Sistemas Operativos
D.A.T.S.I.
Facultad de Informática
Universidad Politécnica de Madrid

PROBLEMAS DE SISTEMAS OPERATIVOS DE 2º

15ª ed. Madrid, noviembre de 2014

Pedro de Miguel Anasagasti
Victor Robles Forcada
Francisco Rosales García
Francisco Sánchez Moreno
María de los Santos Pérez Hernández
Angel Rodríguez Martínez de Bartolomé
Antonio Ruiz Mayor

ISB N
Depósito Legal:

Edita e imprime:

Fundación General de la Universidad Politécnica de Madrid
Servicio de Publicaciones - Facultad de Informática
Campus de Montegancedo -Boadilla del Monte - 28660 - Madrid

Ediciones:
1ª ed. Madrid, noviembre de 1998
2ª ed. Madrid, noviembre de 1999
3ª ed. Madrid, noviembre de 2000
4ª ed. Madrid, noviembre de 2001
5ª ed. Madrid, noviembre de 2002
6ª ed. Madrid, febrero de 2004
7ª ed. Madrid, marzo de 2005
8ª ed. Madrid, febrero de 2006
9ª ed. Madrid, mayo de 2007
10ª ed. Madrid, junio de 2009
11ª ed. Madrid, julio de 2010
12ª ed. Madrid, abril de 2012
13ª ed. Madrid, julio de 2012
14ª ed. Madrid, enero de 2013
15ª ed. Madrid, noviembre de 2014

PRÓLOGO

Este libro trata de problemas de sistemas operativos y sus soluciones. La fuente de estos problemas son los exámenes propuestos durante varios años por el grupo docente de Sistemas Operativos de la Facultad de Informática de la Universidad Politécnica de Madrid en la asignatura de Sistemas Operativos de 2º curso.

Conviene advertir que las soluciones propuestas en este libro no son las únicas posibles. Como bien se sabe, en programación pueden usarse diferentes algoritmos para resolver el mismo problema y por supuesto el estilo del programador puede variar grandemente en claridad, eficiencia, y concisión del código. Se debe advertir también que muchas de las soluciones no han sido compiladas ni ejecutadas por lo que aunque se ha hecho una revisión bastante exhaustiva se pueden encontrar errores. Por otra parte, no es nada fácil asegurar que un programa concurrente funciona correctamente. A diferencia de los programas secuenciales, los concurrentes no son reproducibles y por tanto resulta muy difícil de caracterizar perfectamente un error concreto en los mismos.

En algunos casos los problemas propuestos han tratado de simplificar la realidad para evitar la complejidad de los detalles y realzar más los aspectos de diseño.

No seáis faratones,
que lo bien hecho bien queda
y más vale no hacerlo que hacerlo mal.

"Logarte"

Índice

1
PROCESOS

Problema 1.1

Se quiere realizar un programa que cree un conjunto de procesos que acceden en exclusión mutua a un fichero compartido por todos ellos. Para ello, se deben seguir los siguientes pasos:

*a) Escribir un programa que **cree** N procesos hijos. Estos procesos deben formar un anillo como el que se muestra en la figura 1.1. Cada proceso en el anillo se enlaza de forma unidireccional con su antecesor y su sucesor mediante un pipe. Los procesos **no** deben redirigir su entrada y salida estándar. El valor de N se recibirá como argumento en la línea de mandatos. Este programa debe crear además, el fichero a compartir por todos los procesos y que se denomina `anillo.txt`*

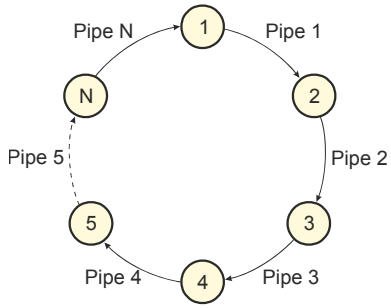


Figura 1.1

b) El proceso que crea el anillo inserta en el mismo un único carácter que hará de testigo, escribiendo en el pipe de entrada al proceso 1. Este testigo recorrerá el anillo indefinidamente de la siguiente forma: cada proceso en el anillo espera la recepción del testigo; cuando un proceso recibe el testigo lo conserva durante 5 segundos; una vez transcurridos estos 5 segundos lo envía al siguiente proceso en el anillo. Codifique la función que realiza la tarea anteriormente descrita. El prototipo de esta función es:

```
void tratar_testigo(int ent, int sal);
```

donde `ent` es el descriptor de lectura del pipe y `sal` el descriptor de escritura (véase figura 1.2).

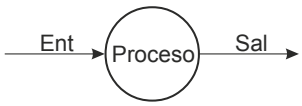


Figura 1.2

c) Escribir una función que lea de la entrada estándar un carácter y escriba ese carácter en un fichero cuyo descriptor se pasa como argumento a la misma. Una vez escrito en el fichero el carácter leído, la función escribirá por la salida estándar el identificador del proceso que ejecuta la función.

d) Cada proceso del anillo crea dos procesos ligeros que ejecutan indefinidamente los códigos de las funciones desarrolladas en los apartados b y c respectivamente. Para asegurar que los procesos escriben en el fichero en exclusión mutua se utilizará el paso del testigo por el anillo. Para que el proceso pueda escribir en el fichero debe estar en posesión del testigo. Si el proceso no tiene el testigo esperará a que le llegue éste. Nótese que el testigo se ha de conservar en el proceso mientras dure la escritura al fichero. Modificar las funciones desarrolladas en los apartados b y c para que se sincronicen correctamente utilizando semáforos.

Solución FG

a) El programa propuesto es el siguiente:

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
   int fd[2];
   int ent;
   int sal;
   pid_t pid;
   int i;
   int N;
   char c;

   if (argc != 2)
   {
      printf("Uso: anillo <N> \n");
      return 0;
   }

   N = atoi(argv[1]);

   /* se crea el pipe de entrada al proceso 1 */
   if (pipe(fd) < 0)
   {
      perror("Error al crear el pipe\n");
      return 0;
   }
   ent = fd[0];
   sal = fd[1];
   write(sal, &c, 1);    /* se escribe el testigo en el primer pipe */

   for(i = 0; i < N; i++)
   {
      if (i != N-1)
         if (pipe(fd) < 0)
         {
            perror("Error al crear el pipe\n");
            return 0;
         }

      pid = fork();
      switch(pid)
      {
         case -1: /* error */
            perror("Error en el fork \n");
            return 0;
         case 0: /* proceso hijo */
            if (i != N-1)
            {
```

```
            close(sal);
            sal = dup (fd[1]);
            close(fd[0]);
            close(fd[1]);
         }
         i = N
         break;
      default: /* proceso padre */
         if (i == N-1)   /* último proceso */
            return 0;
         else
         {
            close(ent);
            close(fd[1]);
            ent = fd[0];
         }
         break;
   }
}

   /* a continuación los procesos del anillo continuarían sus acciones */
   return 0;
}
```

b) El programa propuesto es el siguiente:

```
void tratar_testigo (int ent, int sal)
{
   char c;

   for(;;)
   {
      read(ent, &c, 1);
      sleep(5);
      write(sal, &c, 1);
   }
}
```

c) El programa propuesto es el siguiente:

```
void escribir_en_fichero(int fd)
{
   char c;
   pid_t pid;

   read(0, &c, 1);
   write(fd, &c, 1);
   pid = getpid();
   printf("Proceso %d escribe en el fichero\n", pid);
   return;
}
```

d) Los procesos ligeros ejecutan los códigos de las funciones desarrolladas en b y c de forma indefinida. Para sincronizar correctamente su ejecución es necesario utilizar un semáforo con valor inicial 0 y que denominaremos sincro.

Los códigos de las funciones tratar_testigo y escribir_en_fichero quedan de la siguiente forma:

```
void tratar_testigo (int ent, int sal)
{
   char c;

   for(;;)
   {
      read(ent, &c, 1);
      sem_post(&sincro);
```

```
        /* se permite escribir en el fichero */
        sleep(5);
        sem_wait(&sincro);
        /* se espera hasta que se haya escrito en el fichero */
        write(sal, &c, 1);
    }
}

void escribir_en_fichero(int fd)
{
    char c;
    pid_t pid;

    for(;;)
    {
        read(0, &c, 1);
        sem_wait(&sincro);
        /* se espera a estar en posesión del testigo */
        write(fd, &c, 1);
        pid = getpid();
        printf("Proceso %d escribe en el fichero\n", pid);
        sem_post(&sincro);
        /* se permite enviar el testigo al siguiente proceso */
    }
}
```

Problema 1.2 (septiembre 1998)

Escribir un programa en C, que implemente el mandato cp. La sintaxis de este programa será la siguiente:

```
cp  f1 f2
```

donde f1 será el fichero origen y f2 el fichero destino respectivamente (si el fichero existe lo trunca y sino lo crea). El programa deberá realizar un correcto tratamiento de errores.

Se quiere mejorar el rendimiento del programa anterior desarrollando una versión paralela del mandato cp, que denominaremos cpp (véase figura 1.3). La sintaxis de este mandato será:

```
cpp n f1 f2
```

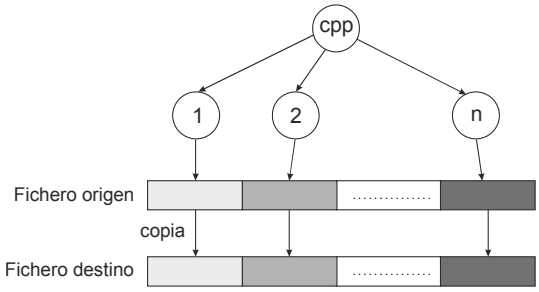


Figura 1.3

donde f1 será el fichero origen y f2 el fichero destino respectivamente, y n el número de procesos que debe crear el programa cpp para realizar la copia en paralelo (ver figura 1.3). Este programa se encargará de:

a) Crear el fichero destino.
b) Calcular el tamaño que tiene que copiar cada uno de los procesos hijo y la posición desde la cual debe comenzar la copia cada uno de ellos
c) Crear los procesos hijos, encargados de realizar la copia en paralelo
d) Deberá esperar la terminación de todos los procesos hijos.

***Nota**: Suponga que el tamaño del fichero a copiar es mayor que n. Recuerde que en C, el operador de división, /, devuelve la división entera, cuando se aplica sobre dos cantidades enteras, y que el operador modulo es %.*

Solución

a) El programa propuesto es el siguiente:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#define MAX_BUF    4096

int main(int argc, char **argv)
{
   int fde, fds;
   int leidos;
   char buffer[MAX_BUF];

   if (argc != 3) {
      printf("Uso: cp f1 f2 \n");
      exit(1);
   }

   fde = open(argv[1], O_RDONLY); /* se abre el fichero de entrada */
   if (fde < 0) {
      perror("Error al abrir el fichero de entrada\n");
      exit(1);
   }

   fds = creat(argv[2], 0644);   /* se crea el fichero de salida */
   if (fds < 0) {
      perror("Error al crear el fichero de salida\n");
      close(fde);
      exit(1);
   }

   /* bucle de lectura del fichero de entrada y escritura en el fichero de salida */
   while ((leidos = read(fde, buffer, MAX_BUF)) > 0)
      if (write(fds, buffer, leidos) != leidos) {
         perror("Error al escribir en el fichero\n");
         close(fde);
         close(fds);
         exit(1);
      }

   if (leidos == -1)
      perror("Error al leer del fichero\n");

   if ((close(fde) == -1) || (close(fds) == -1))
      perror("Error al cerrar los ficheros\n");

   return 0;
}
```

b) En este caso el programa se encargará de:

- Crear el fichero destino.
- Calcular el tamaño que tiene que copiar cada uno de los procesos hijo y la posición desde la cual debe comenzar la copia cada uno de ellos
- Crear los procesos hijos, encargados de realizar la copia en paralelo
- Deberá esperar la terminación de todos los procesos hijos.

Cada uno de los procesos hijos debe abrir de forma explícita tanto el fichero de entrada como el fichero de salida para disponer de sus propios punteros de posición. En caso contrario todos los procesos heredarían y compartirían el puntero de la posición sobre el fichero de entrada y salida y el acceso a los ficheros no podría hacerse en paralelo.

```
#include <sys/types.h>
#include <wait.h>#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define MAX_BUF  4096

int main(int argc, char **argv)
{
   char buffer[MAX_BUF];
   int fde;                /* descriptor del fichero de entrada */
   int fds;                /* descriptor del fichero de salida */
   int n;                  /* número de procesos */
   int size_of_file;       /* tamaño del fichero de entrada */
   int size_proc;          /* tamaño a copiar por cada proceso */
   int resto;              /* resto que copia el último proceso */
   int aux;                /* variables auxiliares */
   int leidos;
   int j;

   if (argc != 4){
      printf("Error, uso: cpp n f1 f2 \n");
      return 0;
   }

   n = atoi(argv[1]); /* numero de procesos */

   fde = open(argv[2], O_RDONLY); /* se abre el fichero de entrada */
      if (fde < 0) {
      perror("Error al abrir el fichero de entrada \n");
      return 0;
   }

   fds = creat(argv[3], 0644);   /* se crea el fichero de salida */
   if (fds < 0) {
      close(fde);
      perror("Error al crear el fichero de salida \n");
      return 0;
   }

   /* obtener el tamaño del fichero a copiar */
   size_of_file = lseek(fde, 0, SEEK_END);

   /* calcular el tamaño que tiene que escribir cada proceso */
   size_proc = size_of_file / n;

   /* El último proceso escribe el resto */
   resto = size_of_file % n;

   /* el proceso padre cierra los ficheros ya que no los necesita*/
   /* cada uno de los procesos hijo debe abrir los ficheros*/
   /* de entrada y salida para que cada uno tenga sus propios*/
   /* punteros de posición */

   for (j = 0; j < n; j++) {
      if (fork() == 0) {
         /* se abren los ficheros de entrada y salida */
```

```
            fde = open(argv[2], O_RDONLY);
            if (fde < 0) {
               perror("Error al abrir el fichero de entrada \n");
               return 0;
         }

            fds = open(argv[3], O_WRONLY);
            if (fds < 0) {
               perror("Error al abrir el fichero de entrada \n");
               return 0;
            }

            /* Cada hijo sitúa el puntero en el lugar correspondiente */
            lseek(fde, j * size_proc, SEEK_SET);
            lseek(fds, j * size_proc, SEEK_SET);

            /* el ultimo proceso copia el resto */
            if (j == n - 1)   /* último */
               size_proc = size_proc + resto;

            /* bucle de lectura y escritura */
            while (size_proc > 0) {
               aux = (size_proc > MAX_BUF ? MAX_BUF : size_proc);
               leidos = read(fde, buffer, aux);
               write(fds, buffer, leidos);
               size = size - leidos;
            }

            close(fde);
            close(fds);
            return 0;
         }
      }

      /* esperar la terminación de todos los procesos hijos */
      while (n > 0) {
         wait(NULL);
         n --;
      }
      return 0;
   }
```

Problema 1.3

Escribir un programa que cree dos procesos que actúen de productor y de consumidor respectivamente. El productor abrirá el fichero denominado /tmp/datos.txt y leerá los caracteres almacenados en él. El consumidor tendrá que calcular e imprimir por la salida estándar el número de caracteres almacenados en ese fichero sin leer del fichero. La comunicación entre los dos procesos debe hacerse utilizando un pipe. Por otra parte, el proceso padre tendrá que abortar la ejecución de los dos procesos hijos, si transcurridos 60 segundos éstos no han acabado su ejecución.

Solución

```
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
```

```
#define MAXBUF   512

pid_t pid1;
pid_t pid2;

void MatarProcesos(void)
{
   /* Esta función se encargará de matar a los procesos hijos */
   kill(pid1, SIGKILL);
   kill(pid2, SIGKILL);
   exit(0);
}

void Productor(int f)
{
   /* El proceso productor lee del fichero y escribe los datos en el pipe
 utilizando el descriptor de escritura que se pasa como parámetro */
   int fd;
   int n;
   char buf[MAXBUF];

   fd = open("/tmp/datos.txt", O_RDONLY);
   if (fd == -1)
     perror("Error en open");
   else {
     while ((n = read(fd, buf, MAXBUF)) != 0)
       write(f, buf, n);
     if (n == -1)
       perror("Error en write");
   }

   close(fd);
   return;
}

void Consumidor(int f)
{
   /* El proceso consumidor lee del pipe utilizando el escriptor de lectura
 que se pasa como parámetro y escribe por la salida estándar (descript. 1) */
   int n;
   char buf[MAXBUF];

   while ((n = read(f, buf, MAXBUF)) != 0)
     write(1, buf, n);

   if (n == -1)
     perror("Error en read ");

   return;
}

int main(void)
{
   int fd[2];
   struct sigaction act;

   /* se crea el pipe */
   if (pipe(fd) < 0) {
     perror("Error al crear el pipe");
     return 0;
   }

   pid1 = fork();    /* Se crea el primer hijo */
```

```
   switch (pid1) {
      case -1:  /* error */
         perror("Error en el fork");
         return 0;
      case 0:/* proceso hijo Productor */
         close(fd[0]);
         Productor(fd[1]);
         close(fd[1]);
         return 0;
      default: /* proceso padre */
         pid2 = fork(); /* Se crea el segundo hijo */
         switch (pid2) {
            case -1:    /* error */
               perror("Error en el fork");
               return 0;
            case 0:     /* proceso hijo consumidor */
               close(fd[1]);
               Consumidor(fd[0]);
               close(fd[0]);
               return 0;
            default: /* proceso padre */
               /* Cierra los descriptores de lectura y escritura porque no
                  Los necesita */
               close(fd[0]);
               close(fd[1]);

               /* Se prepara para recibir la señal SIGALRM */
               act.sa_handler = MatarProcesos;
               act.sa_flags = 0;
               sigaction(SIGALRM, &act, NULL);
               alarm(60);
               /* Cuando llegue la señal SIGLARM se ejecuta el código
                  De la función MatarProcesos que mata a los dos procesos hijo */

               /* Se espera a que acaben los dos procesos hijos. Si vence el
                  Temporizador antes se les mata */
               wait(NULL);
               wait(NULL);
         }
   }
   return 0;
}
```

Problema 1.4

Se quiere desarrollar una aplicación que se encargue de ejecutar los archivos que se encuentren en una serie de directorios (todos los archivos serán ejecutables). Para ello, se deben seguir los siguientes pasos:

a) *Escribir un programa que cree cuatro procesos, que denominaremos de forma lógica 0, 1, 2 y 3. Estos cuatro procesos estarán comunicados entre ellos a través de un único pipe. Uno de los procesos (por ejemplo el 0) ejecutará el código de una función que se denomina* `Ejecutar()` *y el resto (procesos 1, 2 y 3) ejecutarán una función que se denomina* `Distribuir()`**.** *Los códigos de estas funciones se describen en los siguientes apartados.*

b) *La función* `Distribuir()` *en el proceso i abrirá el directorio con nombre "directorio_i" y leerá las entradas de este directorio (todas las entradas se refieren a nombres de archivos ejecutables). Cada vez que lea una entrada enviará, utilizando el pipe creado en el apartado a, el nombre del archivo al proceso que está ejecutando la función* `Ejecutar()`**.** *Cuando el proceso que está ejecutando la función* `Distribuir()` *llega al fin del directorio acaba su ejecución. Escribir el código de la función* `Distribuir()` *para el proceso i.*

c) *El proceso que ejecuta la función* `Ejecutar()` *leerá del pipe que le comunica con el resto de procesos nombres de archivos. Cada vez que lee un archivo lo ejecuta (recuerde que todos los archivos de los directorios son ejecutables) esperando a que acabe su ejecución. Este proceso finaliza su trabajo cuando todos los procesos que ejecutan la función* `Distribuir()` *han acabado. Escribir el código de la función* `Ejecutar()`*. ¿Existe algún problema en la lectura que realiza este proceso del pipe?*

Nota: *Para el desarrollo de este ejercicio considere que todos los nombres de archivos tienen 20 caracteres y que todos los directorios se encuentran en la variable de entorno PATH.*

Solución

a) El programa propuesto es el siguiente:

```
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>

#define NOMBRE_SIZE   20

int main(void)
{
   int fd[2];   /* pipe que comunica a todos los procesos*/
   int j;
   int pid;

   if (pipe(fd) < 0)  {
      perror("Error en pipe");
      exit(1);
   }

   for(j=0; j < 4; j++)  {
      pid = fork();
      switch (pid){
         case -1:
            perror("Error en fork");
            close(fd[0]);
            close(fd[1]);
            exit(1);
         case 0:
            if (j == 0){
               close(fd[1]);  /* Ejecutar no lo utiliza */
               Ejecutar(fd[0]);
               close(fd[0]);
               return 0;
            }
            else {
               close(fd[0]);  /* Distribuir no lo utiliza */
               Distribuir(fd[1], j);
               close(fd[1]);
               return 0;
            }
         default:
            break;
      }
   }

   /* el proceso inicial cierra los descriptores del pipe y acaba su ejecución*/
   close(fd[0]);
   close(fd[1]);
   return 0;
}
```

b) El programa propuesto es el siguiente:

```
void Distribuir(int fd, int i)
{
   DIR *dirp;
   struct dirent *dp;

   dirp = opendir("directorio_i");
   if (dirp == NULL)  {
      perror("Error en opendir");
      return;
   }

   while ((dp=readdir(dirp))!= NULL){
      /* Se saltan las entradas "." y ".." */
      if ((strcmp(dp->d_name, ".") == 0) ||
         (strcmp(dp->d_name, "..") == 0))
         continue;

      /* se escribe el nombre en el pipe. La escritura es atómica */
      if (write(fd, dp->d_name, NOMBRE_SIZE) < NOMBRE_SIZE){
         perror("Error en write");
         closedir(dirp);
         return;
      }
   }
   return;
}
```

c) El programa propuesto es el siguiente:

```
void Ejecutar(int fd)
{
   int pid;
   char nombre[NOMBRE_SIZE];

   while(read(fd, nombre, NOMBRE_SIZE) > 0){
      pid = fork();
      switch(pid){
         case -1:
            perror("Error en fork");
            break;
         case 0:
            close (fd);
            execlp(nombre, nombre, NULL);
            perror("Error en execlp");
            exit(1);
         Default:
            /* Espera la terminación del proceso creado */
            if (wait(NULL) < 0)
               perror("Error en wait");
      }
   }

   return;
}
```

El proceso que ejecuta la función Ejecutar() acaba cuando no quedan escritores en el pipe (los únicos procesos escritores del pipe son aquellos que ejecutan la función Distribuir), por ello es importante que este proceso cierre el descriptor fd[1] antes de comenzar la lectura del pipe. Recuérdese que una lectura de un pipe sin escritores no bloquea y devuelve 0.

La lectura del pipe no plantea ningún problema ya que las operaciones de acceso al pipe (`read` y `write`) se realizan de forma atómica. Además el proceso acaba cuando no queda ningún proceso escritor (proceso que ejecuta la función `Distribuir`).

Problema 1.5 (junio 2000)

Se desea construir la función int pipe_two(int fd[4]), que asigna a las 4 entradas del vector fd el valor de los 4 nuevos descriptores, según la siguiente relación:

- *fd[0]: Primer descriptor de lectura, que lee los datos que se escriban a través de fd[2]*
- *fd[1]: Segundo descriptor de lectura, que lee los datos que se escriban a través de fd[3]*
- *fd[2]: Primer descriptor de escritura.*
- *fd[3]: Segundo descriptor de escritura.*

Dicha función devuelve 0 si su ejecución ha sido correcta y –1 en caso de error.

Sobre este pipe doble, se definen las siguientes operaciones de lectura y escritura:

- *int read_from2 (int fd[2], char *buffer, int n_bytes): almacena en buffer los datos leídos de fd[0] y fd[1], de forma que se lee un byte de fd[0] y se almacena en la primera posición del buffer; a continuación se lee un byte de fd[1] y se almacena en la siguiente posición del buffer y así sucesivamente. n_bytes contiene el número de bytes totales a leer. Se supone que n_bytes es un número par.*

  La función devuelve el número total de bytes leídos si su ejecución ha sido correcta y –1 en caso de error.

- *int write_to2 (int fd[2], char *buffer, int n_bytes): escribe los bytes que se encuentren en las posiciones pares del buffer en el fichero descrito por fd[0] y las posiciones impares en el fichero descrito por fd[1]. n_bytes contiene el número de bytes totales que se van a escribir.*

Se considera que la primera posición del buffer es la posición 0 y que es una posición par.

La función devuelve el número de bytes escritos si su ejecución ha sido correcta y –1 en caso de error.

Se pide:

a) Implementar las funciones pipe_two( ), read_from2( ) y write_to2( ).
b) Utilizando las anteriores funciones, programar un proceso que realice los siguientes pasos:
   - *Crea 2 procesos hijos.*
   - *Lee los caracteres del fichero "/usr/datos.txt" y a uno de los procesos hijos le pasa los caracteres pares y a otro los caracteres impares.*

*c) Los procesos hijos leen los datos y los procesan mediante la función char * tratar_datos(char *buffer). La función devuelve una cadena de caracteres, que contiene los datos procesados. Esta función **no** hay que implementarla. Sólo se debe utilizar.*
d) Los procesos hijos envían los datos procesados de nuevo al padre.
e) El padre recibe los datos de ambos hijos, mezclándolos en un buffer; es decir, lee los caracteres que les envía los procesos hijos de forma alternativa.

El proceso que se pide implementar sigue la estructura que se representa en la figura 1.4:

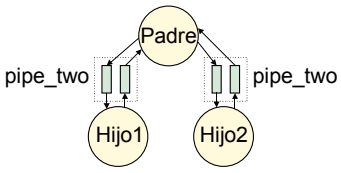


Figura 1.4

Solución

a) La función *pipe_two* construye 2 pipes y los enlaza de la forma indicada en el enunciado, es decir, enlaza el descriptor fd[0] con el descriptor fd[2] y el descriptor fd[1] con el descriptor fd[3]:

```
int pipe_two(int fd[4]) {
   int pipe1[2], pipe2[2];
   int status=0;
```

```
   if (pipe(pipe1) == -1)
      status = -1;
   else if (pipe(pipe2) == -1){
      close(pipe1[0];
      close(pipe1[1];
      status = -1;
   }

   if (status == 0) {
      fd[0]= pipe1[0];
      fd[1]= pipe2[0];
      fd[2]= pipe1[1];
      fd[3]= pipe2[1];
   }
   return status;
}
```

La función *read_from2* hace una lectura alternativa de los ficheros cuyos descriptores recibe como argumento. Vamos a suponer que si uno de los ficheros llega al final, *read_from2* deja de leer en esa vuelta del bucle y devuelve el número de bytes leídos hasta entonces. Por otro lado, si hay una lectura incorrecta de alguno de los ficheros, termina inmediatamente devolviendo –1:

```
int read_from2(int fd[2], char *buffer, int n_bytes) {

   int bytes_read=1;
   int bytes_total=0;

   for (i=0; ((i<n_bytes) && (bytes_read != 0)); i=i+2) {
      if ((bytes_read = read(fd[0],(buffer+i),1)) == -1) {
         bytes_total = -1;
         break;
      }
      else {
         bytes_total + = bytes_read;
      }

      if ((bytes_read = read(fd[1],(buffer+i+1),1) == -1) {
         bytes_total = -1;
         break;
      }
      else {
         bytes_total + = bytes_read;
      }
   }
   return bytes_total;
}
```

La función write_to2 escribe de forma alternativa en los ficheros cuyos descriptores recibe como argumento. Si hay una escritura incorrecta en alguno de los ficheros, termina inmediatamente devolviendo -1:

```
int write_to2 (int fd[2], char *buffer, int n_bytes) {

   int bytes_write;
   int bytes_total = 0;

   for (i=0; i<n_bytes ; i=i+2) {
      if ((bytes_write = write(fd[0],(buffer+i),1)) == -1) {
         bytes_total = -1;
         break;
      }
      else {
```

```
            bytes_total + = bytes_read;
        }

        if ((bytes_read = write(fd[1],(buffer+i+1),1) == -1) {
            bytes_total = -1;
            break;
        }
        else {
            bytes_total + = bytes_read;
        }
    }
    return bytes_total;
}
```

b) Se va a suponer que el fichero "/usr/datos.txt" tiene un número par de caracteres y que ocupa un máximo de 8192 bytes. (Esto se define en la constante MAX_BUF).

```
#define MAX_BUF 8192
/* En la función main() no se han tratado todos los casos de errores.
*/
int main(void) {
    int fd[4], fd_file, bytes_read;
    char buffer_write[MAX_BUF];
    char buffer_read[MAX_BUF];

    /* Creación del pipe doble */
    if (pipe_two(fd) < 0)
    {
        perror("pipe_two");
        exit(1);
    }

    /* Creación del primer proceso hijo */
    switch (pid =fork())  {
    case -1:
        perror("fork 1");
        close(fd[0]);
        close(fd[1]);
        close(fd[2]);
        close(fd[3]);
        exit(1);
    case 0:  /* Primer proceso hijo */

        /* Cierro los descriptores que no utilizo */
        close(fd[1]);
        close(fd[3]);
        /* Lee del primer descriptor de lectura */
        bytes_read=read(fd[0],buffer_read, MAX_BUF);

        buffer_write = tratar_datos(buffer_read);
        /* Escribo en el primer descriptor de escritura */
        write(fd[2],buffer_write, bytes_read);

        /* Cierro el resto de descriptores */
        close(fd[0]);
        close(fd[2]);
        return 0;

    default:  /* Proceso padre */
        /* Creación del segundo proceso hijo */
        switch (pid =fork())  {
        case -1:
            perror("fork 2");
```

```
            close(fd[0]);
            close(fd[1]);
            close(fd[2]);
            close(fd[3]);
            exit(1);
        case 0: /* Segundo proceso hijo */

            /* Cierro los descriptores que no utilizo */
            close(fd[0]);
            close(fd[2]);

            /* Lee del segundo descriptor de lectura */
            bytes_read=read(fd[1],buffer_read, MAX_BUF);
            buffer_write = tratar_datos(buffer_read);
            /* Escribo en el segundo descriptor de escritura */
            write(fd[3],buffer_write, bytes_read);

            /* Cierro el resto de descriptores */
            close(fd[1]);
            close(fd[3]);
            return 0;

        default: /* Proceso padre */
            /* Leo los caracteres del fichero datos */
            fd_file = open("/usr/datos.txt",O_RDONLY);
            bytes_read=read(fd_file,buffer_write,MAX_BUF);

            /* Escribo los bytes leidos a los
            descriptores de escritura: fd[2], fd[3] */
            write_to_2(fd+2,buffer_write,bytes_read);

            /* Leo los datos que me pasan los procesos hijos
            en el buffer de lectura de los descriptores de
            lectura: fd[0] y fd[1] */
            bytes_read= read_from2(fd,buffer_read,MAX_BUF);

            /* Cierro todos los descriptores */
            close(fd[0]);
            close(fd[1]);
            close(fd[2]);
            close(fd[3]);

            /* Aquí iría el código de uso de buffer_read */
        }
    }
    return 0;
}
```

Problema 1.6 (junio 2002)

*a) Se desea **implementar** un programa que, utilizando los servicios de Unix, ejecute un "ls recursivo", es decir, un programa que liste los contenidos de un directorio, de todos sus subdirectorios y así sucesivamente hasta llegar a las hojas del árbol que representa dicho directorio. No se debe utilizar en ningún caso el mandato "ls" ya implementado. El programa debe imprimir el árbol de directorios en profundidad.*

Por ejemplo, para el árbol de directorios de la figura 1.5, ejecutando:

```
$ ls_recursivo /home
```

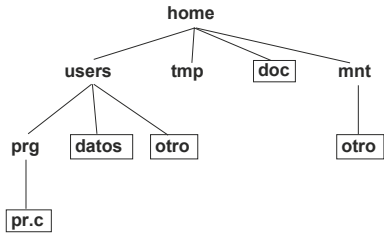


Figura 1.5

Se obtendría la siguiente salida:

```
/home/users
/home/users/prg
/home/users/prg/pr.c
/home/users/datos
/home/users/otro
/home/tmp
/home/doc
/home/mnt
/home/mnt/otro
```

*b) Se pide **implementar** el programa "mifind_recursivo", a partir del ls implementado en el apartado a) y a partir del mandato "grep". No se debe implementar este último mandato.*

El programa "mifind_recursivo" debe tener el mismo comportamiento que el mandato "find" de Unix, que busca ficheros en un árbol de directorios. Recibe como argumentos el directorio a partir del cuál se hace la búsqueda y el nombre del fichero que se busca. La búsqueda se realiza de forma recursiva visitando todos los subdirectorios a partir del inicial.

Ejemplo:

```
$ mifind_recursivo /home otro
```

La ejecución de este mandato tendrá la siguiente salida:

```
/home/users/otro
/home/mnt/otro
```

Aclaración: El mandato "grep" permite mostrar líneas que concuerden con un patrón. Este mandato lee de la entrada estandar y encuentra el patrón que se le pase como argumento. Ejemplo:

```
$ grep hola
```

La ejecución de este mandato lee de la entrada estándar e imprime todas las líneas que contengan la palabra "hola".

c) Imagine que existe una implementación multi-thread del "ls recursivo", que crea un nuevo thread para explorar cada directorio encontrado.

Plantear** qué mecanismos de sincronización son necesarios en esta implementación, de forma que la solución multi-thread produzca un resultado idéntico (en el mismo orden) al producido por la solución secuencial. **No implementar** nada. **Responder brevemente.

Solución

a) Supongo que se pasa a la función el nombre absoluto del directorio. Si no fuera así, habría que concatenar el directorio actual.

```
#define MAX_BUF  256
/* ls_recursivo */
int main(int argc, char *argv[])
{
   struct stat inf_dir;
   char buf[MAX_BUF];

   if (argc != 2)
   {
      fprintf(stderr,"Error. Uso: ls_recursivo nombre_direct\n");
      exit(1);
```

```
   }

   /* Comprobar si es un directorio */
   stats(argv[1],&inf_dir);
   if (!S_ISDIR(inf_dir.st_mode)){
      fprintf(stderr,"Error. No es un directorio\n");
      exit(1);
}

   if (funcion_ls_recursivo(argv[1])<0)
   {
      fprintf(stderr,"Error: ls_recursivo\n");
      exit(1);
   }
   return 0;
}

int funcion_ls_recursivo(char *nombre_directorio)
{
   char buf[MAX_BUF];
   struct dirent *dp;
   DIR *dirp;
   struct stat inf_dir;
   char path_completo[MAX_BUF];

   dirp = opendir(nombre_directorio);
   if (dirp==NULL)
   {
      return -1;
   }

   while ((dp = readdir(dirp)) ¡= NULL)
   {
      /* Imprime el directorio o fichero */
      path_completo = strcat(nombre_directorio,dp->d_name);
      fprintf(stdout,"%s\n",path_completo);
      stat(path_completo,&inf_dir);
      /* Si es un directorio, hacemos una
      llamada recursiva */
      if (S_ISDIR(inf_dir.st_mode)){
         funcion_ls_recursivo(path_completo);
      }
   }
   return 0;
}
```

b) La solución consiste en unir mediante un pipe los procesos “ls_recursivo” y “grep”, de forma que la salida del primero sea la entrada del segundo.

```
/* mifind_recursivo */
int main (int argc, char *argv[])
{
   int fd[2];
   int pid, pid2,pid3;

   if (argc != 3)
   {
      fprintf(stderr,"Error. Uso: mi_find_recursivo directorio fichero\n");
      exit(1);
   }
   if (pipe(fd)<0){
      perror("pipe");
      exit(1);
```

```
   }

   pid = fork();
   switch (pid)
   {
      case -1:
         close(fd[0]);
         close(fd[1]);
         perror("fork");
         exit(1);
      case 0:
         close(fd[0]);
         close(1);
         dup(fd[1]);
         close(fd[1]);
         pid2=fork();
         switch(pid2)
         {
            case -1:
               perror("fork");
               exit(1);
            case 0:
               execl("ls_recursivo",argv[1],NULL);
               perror("execl");
            default:
               wait(NULL);
               return 0;
         }
      default:
         close(fd[1]);
         close(0);
         dup(fd[0]);
         close(fd[0]);
         pid3 = fork();
         switch (pid3)
         {
            case -1:
               perror("fork");
               exit(1);
            case 0:
               execl("grep",argv[2],NULL);
               perror("execl");
            default:
               wait(NULL);
               wait(NULL);
               return 0;
         }
      }
   return 0;
}
```

c) La solución multi-thread debe utilizar mecanismos de sincronización de cara a imprimir la salida, ya que tiene que seguir el orden expuesto en el problema. El resto de la operación (lectura de los directorios) debe hacerse concurrentemente, para beneficiarse de la característica multi-thread. Se puede utilizar cualquier mecanismo de sincronización, siendo bastante aconsejable utilizar mútex y variables condicionales, por su afinidad con los threads. Habría que utilizar varios mútex para secuenciar la salida en el orden predeterminado. Un único mútex no asegura ese orden.

Problema 1.7

Se desea desarrollar una aplicación que debe realizar dos tareas que se pueden ejecutar de forma independiente. Los códigos de estas dos tareas se encuentran definidos en dos funciones cuyos prototipos en lenguaje de programación C, son los siguientes:

```
void tarea_A(void);
void tarea_B(void);
```

Se pide:

a) Programar la aplicación anterior utilizando tres modelos distintos:

- *1. Un único proceso.*
- *2. Procesos para aprovechar paralelismo. Plantear una solución que minimice el número de procesos creados y solicite el mínimo número de servicios al sistema operativo.*
- *3. Procesos ligeros para aprovechar paralelismo. En este caso también se minimizará el número de procesos ligeros creados y el número de servicios.*

En cualquiera de los tres casos la aplicación debe terminar cuando se hayan acabado las tareas A y B.

b) Suponiendo que las tareas tienen las siguientes fases de ejecución:

- *A: 60 ms de CPU, lectura de 120 ms sobre el dispositivo 1, 60 ms de CPU*
- *B: 40 ms de CPU, escritura de 100 ms sobre el dispositivo 2, 40 ms de CPU*

Realizar un diagrama que muestre la ejecución de los procesos en el sistema, obteniendo el tiempo que tardará la aplicación en terminar en cada uno de los modelos anteriores. Se tomará como t = 0 el instante en el que empieza la ejecución del primer proceso.

Para la realización del apartado b se considerarán los siguientes supuestos:

***1.** El sistema operativo emplea un algoritmo de planificación FIFO.*

***2.** Se comenzará con la ejecución de la tarea A.*

***3.** Se tendrá en cuenta el tiempo de ejecución de los siguientes servicios que ofrece el sistema operativo:*

- fork: 80 ms
- exec: 100 ms
- Tratamiento de una interrupción de periférico: 10 ms
  - *Otros servicios (read, write, wait, pthread_create, pthread_join, exit, pthread_exit, etc.): 10 ms*

NOTA: los tiempos del SO no son realistas, puesto que están por debajo del ms. Se han supuesto estas cifras para que la gráfica con la solución quede más clara.

***4.** En el sistema no ejecuta ningún otro proceso y no se tendrán en cuenta las interrupciones del reloj.*

***5.** Cuando un proceso crea a otro, continúa la ejecución el primero.*

Solución

a) Con un único proceso el programa quedaría de la siguiente forma:

```
int main(void)
{
tarea_A();
tarea_B();
return 0;
}
```

En este caso las dos tareas se ejecutan de forma secuencial.

Cuando se utilizan procesos para aprovechar paralelismo, la solución sería la siguiente:

```
int main(void)
{
pid_t pid;
```

```
pid = fork();
if (pid == 0)       /* el hijo ejecuta la tarea B */
Tarea_B;
else {              /* el padre ejecuta la tarea A */
tarea_A();
wait(NULL);     /* espera a que el hijo acabe */
}
return 0;
}
```

Empleando procesos ligeros la solución sería:

```
int main(void)
{
pthread_t t;
int pid;

/* Se crea un proceso ligero para que ejecute la tarea B*/
pthread_create(&t, NULL, tarea_B, NULL);

/* El proceso ligero principal ejecuta la tarea A de forma concurrente con el proceso ligero creado anteriormente */
tarea_A();

/* En este punto el proceso ligero principal ha acabado la tarea A. Espera a que acabe el proceso ligero que ejecuta la tarea B
*/
pthread_join(t, NULL);
return 0;
}
```

El proceso ligero que ejecuta la tarea B, acaba cuando se llega al fin de la función tarea_B().

b) Los diagramas de tiempo se encuentran en las figuras 1.6, 1.7 y 1.8.

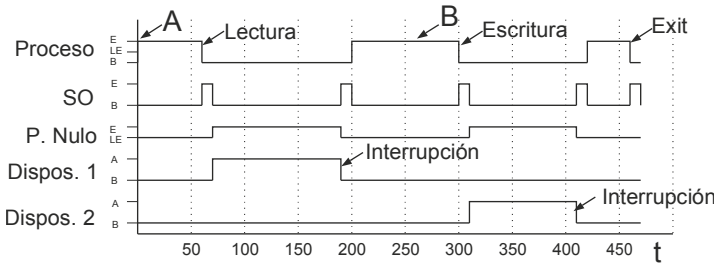


Figura 1.6 Caso de un solo proceso

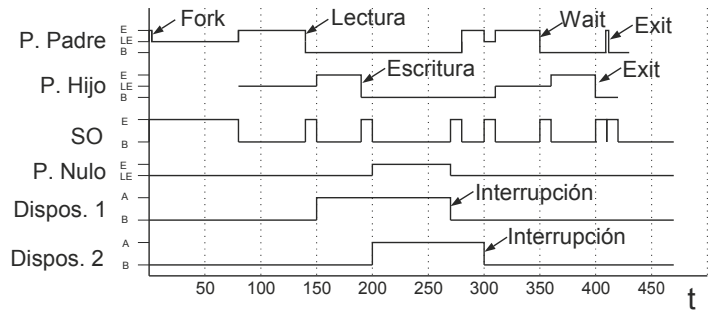


Figura 1.7 Caso de dos procesos

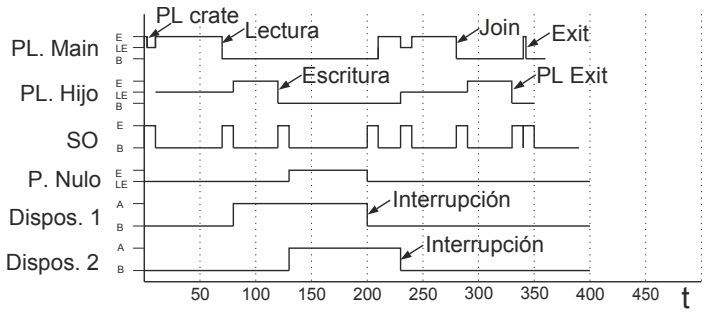


Figura 1.8 Caso de procesos ligeros

Problema 1.8

El objetivo de este ejercicio es mostrar el uso de los dos mecanismos disponibles en UNIX para construir aplicaciones concurrentes: los procesos convencionales, creados mediante la llamada `fork`*, y los procesos ligeros o threads.*

Para ello, se va a construir una aplicación, denominada `lincon`*, que recibe como argumento un conjunto de ficheros y que muestra por la salida estándar el número de líneas que tiene cada uno de ellos y el número de líneas total. Se pretende que dicha aplicación procese de forma concurrente cada uno de los ficheros especificados por el usuario. Además, la aplicación debe cumplir los siguientes requisitos:*

a) Si no se puede acceder a alguno de los ficheros especificados, deberán procesarse normalmente los ficheros restantes, pero el programa deberá terminar devolviendo un 1. Si todo va bien, devolverá un 0. Nótese que se está siguiendo la típica convención usada en UNIX.

b) Los ficheros se leerán realizando lecturas de 8 KiB.

Se desarrollarán dos versiones de la aplicación: una con threads y otra con procesos convencionales.

Solución

Programa basado en *threads*. Se presentan dos versiones de este programa:

- Uso de una variable global para acumular el número total de líneas (`lincon_thr_v1.c`). Es necesario crear una sección crítica para evitar los problemas de carrera que pueden producirse al actualizar la variable global.
- Cada *thread* devuelve el número de líneas del fichero que ha procesado (`lincon_thr_v2.c`). Elimina la necesidad de la sección crítica.

Recuerde que debe especificar la biblioteca de threads a la hora de generar el programa:

```
cc -o lincon lincon.c -lpthread
```

Programa basado en procesos convencionales. Se presentan tres versiones de este programa:

a) Uso de una variable global para acumular el número total de líneas (`lincon_proc_v1.c`). Nótese que esta versión no calcula bien el total: siempre sale 0. El problema no se debe a ninguna condición de carrera, sino a que cada proceso tiene su propio mapa de memoria. Por tanto, cuando un proceso hijo actualiza la variable global, este cambio no afecta al proceso padre ni a ninguno de sus hermanos. Ésta es una de las principales diferencias entre los procesos convencionales y los *threads*.

b) Cada proceso devuelve el número de líneas del fichero que ha procesado (`lincon_proc_v2.c`). Esta versión sólo funciona si el fichero que se procesa tienen como mucho 254 líneas. Esto se debe a que en la llamada `wait` sólo se reciben los 8 bits de menos peso del valor especificado en el `exit` correspondiente. Para entender esta extraña limitación, hay que tener en cuenta que en el valor devuelto por la llamada `wait` hay más información, aparte del valor devuelto por el programa (por ejemplo, se indica sí el programa terminó normalmente o debido a una señal). Nótese que tampoco sería válida esta versión para ficheros con 255 líneas, pues ese valor corresponde con el caso en el que el proceso hijo devuelve un -1 para indicar que no ha podido procesar el fichero.

c) Versión que usa una tubería para que cada hijo mande al padre el número de líneas del fichero que le ha tocado procesar (lincon_proc_v1.c).

Fichero `lincon_thr_v1.c`

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define TAMBUF 8192

int totlin=0;


pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER;

int contar(char *arg) {
   int fo, nbytes, i, nlineas=0;
   char buf[TAMBUF];

   if ((fo=open(arg, O_RDONLY))<0) {
      perror("Error abriendo fichero");
      return -1;
      }
   while ( (nbytes= read(fo, buf, TAMBUF)) >0)
   for (i=0; i<nbytes; i++)
   if (buf[i]=='\n') nlineas++;

   if (nbytes<0) {
      perror("Error leyendo en fichero");
      return -1;
      }
   printf("%s %d\n", arg, nlineas);

   /* Sección crítica para actualizar el total */
   pthread_mutex_lock(&mutex);
   totlin+= nlineas;
   pthread_mutex_unlock(&mutex);

   return 0;
   }

int main(int argc, char *argv[]) {
   int i, estado=0, retorno;
   pthread_t *thid;

   if (argc<2) {
      fprintf(stderr, "Uso: %s fichero ...\n", argv[0]);
      exit(1);
      }

   /* Reserva en tiempo de ejecución el espacio para almacenar los
descriptores de los threads */
   thid= malloc((argc -1)*sizeof(pthread_t));

   for (i=0; i<argc-1; i++)
   if (pthread_create(&thid[i], NULL, (void * (*)(void *))contar,  (void
*)argv[i+1])!=0) {
      perror("Error creando thread");
      exit (1);
      }
```

```
   for (i=0; i<argc-1; i++) {
      pthread_join(thid[i], (void **)&retorno);
      if (retorno<0)
      estado=1;
      }
   printf("total %d\n",totlin);
   exit(estado);
   }
```

Fichero `lincon_thr_v2.c`

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define TAMBUF 8192

int contar(char *arg) {
   int fo, nbytes, i, nlineas=0;
   char buf[TAMBUF];

   if ((fo=open(arg, O_RDONLY))<0) {
      perror("Error abriendo fichero");
      return -1;
      }
   while ( (nbytes= read(fo, buf, TAMBUF)) >0)
   for (i=0; i<nbytes; i++)
   if (buf[i]=='\n') nlineas++;

   if (nbytes<0){
      perror("Error leyendo en fichero");
      return -1;
      }
   printf("%s %d\n", arg, nlineas);
   return nlineas;
   }

int main(int argc, char *argv[]) {
   int i, estado=0;
   pthread_t *thid;
   int nlineas, totlin=0;

   if (argc<2) {
      fprintf(stderr, "Uso: %s fichero ...\n", argv[0]);
      exit(1);
      }

   /* Reserva en tiempo de ejecución el espacio para almacenar los
   descriptores de los threads */
   thid= malloc((argc -1)*sizeof(pthread_t));

   for (i=0; i<argc-1; i++)
   if (pthread_create(&thid[i], NULL, (void * (*)(void *))contar,  (void
*)argv[i+1])!=0) {
      perror("Error creando thread");
      exit (1);
      }
   for (i=0; i<argc-1; i++) {
      pthread_join(thid[i], (void **)&nlineas);
      if (nlineas>=0)
      totlin+=nlineas;
```

```
    else
    estado=1;
    }
  printf("total %d\n",totlin);
  exit(estado);
  }
```

Fichero `lincon_proc_v1.c`

```
/* Esta versión no funciona ya que no se actualiza el total */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

#define TAMBUF 8192


int contar(char *arg) {
  int fo, nbytes, i, nlineas=0;
  char buf[TAMBUF];

  if ((fo=open(arg, O_RDONLY))<0) {
    perror("Error abriendo fichero");
    return -1;
    }
  while ( (nbytes= read(fo, buf, TAMBUF)) >0)
  for (i=0; i<nbytes; i++)
  if (buf[i]=='\n') nlineas++;

  if (nbytes<0) {
    perror("Error leyendo en fichero");
    return -1;
    }
  printf("%s %d\n", arg, nlineas);

  /* Se pretende actualizar el total, pero no funciona... */
  totlin+= nlineas;

  return 0;
  }

int main(int argc, char *argv[]) {
  int i, estado=0, retorno, valor;

  if (argc<2) {
    fprintf(stderr, "Uso: %s fichero ...\n", argv[0]);
    exit(1);
    }

  for (i=0; i<argc-1; i++)
  if (fork()==0) {
    valor=contar(argv[i+1]);
    exit(valor);
    }
  for (i=0; i<argc-1; i++) {
    wait(&retorno);
    if (retorno!=0)
    estado=1;
    }
  printf("total %d\n",totlin);
  exit(estado);
```

```
   }
```

Fichero `lincon_proc_v2.c`

```
/* Esta versión no funciona para ficheros que tengan más de 254 líneas por la limitación existente en el rango del valor devuelto por el exit */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

#define TAMBUF 8192

int contar(char *arg) {
   int fo, nbytes, i, nlineas=0;
   char buf[TAMBUF];

   if ((fo=open(arg, O_RDONLY))<0) {
      perror("Error abriendo fichero");
      return -1;
      }
   while ( (nbytes= read(fo, buf, TAMBUF)) >0)
   for (i=0; i<nbytes; i++)
   if (buf[i]=='\n') nlineas++;

   if (nbytes<0) {
      perror("Error leyendo en fichero");
      return -1;
      }
   printf("%s %d\n", arg, nlineas);
   return nlineas;
   }

int main(int argc, char *argv[]) {
   int i, estado=0, retorno, valor, totlin=0;

   if (argc<2) {
      fprintf(stderr, "Uso: %s fichero ...\n", argv[0]);
      exit(1);
      }

   for (i=0; i<argc-1; i++)
   if (fork()==0) {
      valor=contar(argv[i+1]);
      exit(valor);
      }
   for (i=0; i<argc-1; i++) {
      wait(&retorno);
      if (!WIFEXITED(retorno))
      /* el hijo ha terminado debido a una señal */
      estado=1;
      else {
         /* el hijo ha terminado normalmente */

         /* extraigo el valor devuelto */
         retorno=WEXITSTATUS(retorno);

         /* el -1 devuelto se ha convertido en un 255 */
         if (retorno==255)
         estado=1;
         else
         totlin+= retorno;
```

```
      }
    }
  printf("total %d\n",totlin);
  exit(estado);
  }
```

Fichero `lincon_proc_v3.c`

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

#define TAMBUF 8192

int tuberia[2];

int contar(char *arg) {
  int fo, nbytes, i, nlineas=0;
  char buf[TAMBUF];

  if ((fo=open(arg, O_RDONLY))<0) {
    perror("Error abriendo fichero");
    return -1;
    }
  while ( (nbytes= read(fo, buf, TAMBUF)) >0)
  for (i=0; i<nbytes; i++)
  if (buf[i]=='\n') nlineas++;

  if (nbytes<0) {
    perror("Error leyendo en fichero");
    return -1;
    }
  write(tuberia[1], &nlineas, sizeof(nlineas));
  printf("%s %d\n", arg, nlineas);
  return 0;
}

int main(int argc, char *argv[]) {
  int i, estado=0, retorno, valor, totlin=0, nlineas;

  if (argc<2) {
    fprintf(stderr, "Uso: %s fichero ...\n", argv[0]);
    exit(1);
    }

  pipe(tuberia);
  for (i=0; i<argc-1; i++)
  if (fork()==0) {
    close(tuberia[0]);
    valor=contar(argv[i+1]);
    exit(valor);
    }
  close(tuberia[1]);
  for (i=0; i<argc-1; i++) {
    wait(&retorno);
    if (retorno!=0)
    estado=1;
      else {
      read(tuberia[0], &nlineas, sizeof(nlineas));
      totlin+= nlineas;
      }
    }
```

```
printf("total %d\n",totlin);
exit(estado);
}
```

Problema 1.9

Se debe escribir en C el programa paralelo pwc que cuente el número de líneas y los caracteres que no sean espacio de un determinado fichero.

La sintaxis del programa será: pwc n fich, donde n es el número de procesos hijo que debe crear el programa pwc y fich es el nombre del fichero de entrada.

El programa deberá:

a) *Proyectar en memoria el fichero fich recibido como parámetro de entrada. Se recuerda que la llamada al sistema mmap tiene el siguiente prototipo:*
   *void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)*
b) *Calcular el tamaño que debe procesar cada uno de los procesos hijo y la posición desde la cual debe comenzar cada uno de ellos (veáse la figura 1.9). El proceso almacenará estos valores en dos vectores int offset[n], int size[n].*
c) *Crear los procesos hijo encargados de realizar la cuenta en paralelo, pasándoles la posición k que referencia a los valores offset[k] y size[k] con los valores que debe utilizar.*
d) *Esperar la finalización de los procesos hijo y recoger, utilizando pipes, los valores calculados por cada uno de ellos.*

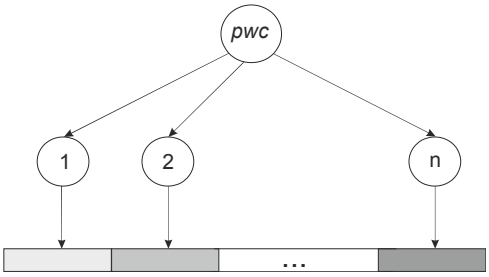


Figura 1.9

Se pide:

a) *Desarrollar el programa descrito.*
b) *Explicar de forma breve y razonada si los procesos hijos podrían comunicar los valores calculados al padre a través de memoria compartida.*
c) *¿Se podrían haber creado los vectores int numLineas[n] e int palabras[n] y que los procesos hijos escribieran en ellos sus resultados?*

Solución

a) El programa propuesto es el siguiente:

```
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define MAX_PROC 20

struct datos{
   int numLineas;
   int numCaracteres;
```

```
};

int main (int argc, char **argv){

   int fde;                  // Descriptor del fichero de entrada
   int fd[2];                   // Descriptores del pipe
   struct datos resul;          // Resultado obtenido por cada uno de los hijos
   int tamFichero;              // Tamaño del fichero a proyectar
   void *p, *q;                 // Punteros al fichero proyectado en memoria
   int size_proc;            // Tamaño del fichero entre número de procesos
   int resto;                   // Tamaño restante del fichero
   int offset[MAX_PROC];     // Dirección de comienzo para cada proceso hijo
   int size[MAX_PROC];          // Tamaño que debe procesar cada proceso hijo
   int n;                    // Número de procesos hijo
   int i,j;                  // Para bucles
   int pid;                  // Identificador de los procesos hijo
   char *c;                  // Carácter actual que se trata
   int nLineas, nCaracteres; // Líneas y caracteres no espacio contados por
   // cada uno de los hijos
   int lineasTot, caracteresTot;  // Líneas y caracteres no espacio totales
   //(del padre)

   if (argc!=3){
      printf("Lo siento, este programa necesita dos argumentos %d");
      return 0;
   }

   // Número de procesos hijo
   n = atoi(argv[1]);

   // Antes de proyectar el fichero hay que abrirlo
   fde = open(argv[2], O_RDONLY);
   if (fde<0){
      perror("Error abriendo el fichero!");
      return 0;
   }

   // Calculo el tamaño del fichero
   tamFichero = lseek (fde, 0, SEEK_END);

   // Proyecto el fichero como MAP_SHARED, para que pueda ser compartido
   // por los procesos hijo
   p = mmap(0,(size_t) tamFichero, PROT_READ, MAP_SHARED, fde, 0);
   if (p<0){
      perror("Error proyectando el fichero");
      return 0;
   }

   // Se calcula el tamaño que tiene que procesar cada proceso
   size_proc = tamFichero / n;

   // El último proceso procesará también el resto
   resto = tamFichero % n;

   // Vectores size y offset, con los tamaños y direcciones
   for (i=0; i<n; i++){
      size[i] = size_proc;
      offset[i] = i * size_proc;
   }

   // El último proceso se encarga del resto del fichero
   size[n-1] += resto;
```

```
   // Se crea el pipe
   if (pipe(fd) < 0) {
      perror("Error al crear el pipe");
      return 0;
   }

   // Creo los procesos hijo
   // Cada uno se encarga de su parte del fichero proyectado
   for (i=0; i<n; i++){
      pid = fork();

      // Código del proceso hijo
      if (pid == 0){

         // Puntero en el punto de comienzo
         q = p + offset[i];
         nLineas=0;
         nCaracteres=0;

         // Leo carácter por carácter y compruebo si es salto
         // de línea o si no es espacio
         for (j=0 ; j<size[i];j++) {
            c=q;

            if (*c=='\n')
            nLineas++;

            if (*c != ' ')
            nCaracteres++;

            q++;
         }

         // Escribo la estructura
         resul.numLineas = nLineas;
         resul.numCaracteres = nCaracteres;

         // Escribo la estructura en el pipe
         write (fd[1], &resul, sizeof(resul));
         return 0;
      }
   }

   // El padre espera los valores mandados por todos los procesos hijo
   lineasTot=0;
   caracteresTot=0;

   // Por cada uno de los hijos tengo que leer del pipe
   for (i=0;i<n;i++) {
      read(fd[0],&resul,sizeof(resul));
      lineasTot += lineasTot + resul.numLineas;
      caracteresTot += caracteresTot + resul.numCaracteres;
   }

   printf("Caracteres totales no espacio %d\n Lineas Totales %d\n",
   caracteresTot, lineasTot);
   return(0);
}
```

b) Sí que sería posible usar el mecanismo de memoria compartida para pasar los resultados al padre. Por ejemplo, se podría utilizar un fichero proyectado en memoria, en el cual los hijos escribieran su resultado y el padre lo leyera.

c) No sería posible utilizar los dos vectores propuestos, ya que los hijos, al escribir en ellos, se crearían una copia privada, siendo imposible para al proceso padre acceder a dichos valores.

Problema 1.10 (2004)

Dado el siguiente código de un ejecutable cuyo nombre es `run_processes`:

```
1  int main(int argc, char *argv[]){
2    int pid, number, status;
3
4    number = atoi(argv[1]);
5    if (number<=0)
6      return 0;
7    pid = fork();
8    if (pid == 0) {
9      fork();
10     number=number-1;
11     sprintf(argv[1], "%d", number);
12     execvp(argv[0],argv);
13     perror("exec");
14     exit(1);
15   } else {
16     while (wait(&status)!= pid)
17       continue;
18   }
19   return 0;
20 }
```

***Aclaración**:La función `atoi()` devuelve el valor entero representado por la cadena de caracteres dada. Por otro lado, dado un valor entero y un buffer, `sprintf()` almacena en el buffer la tira de caracteres que representa el entero. Ejemplo:*

`atoi("35")` devuelve 35.

`sprintf(cadena, "%d", 35)` almacena en la variable `cadena` el texto "35".

Se supone que los servicios fork() y execvp() se ejecutan de forma correcta.

a) *Indicar qué sucede si se ejecuta `run_processes` sin ningún argumento.*
   Para el resto del ejercicio considere la ejecución del mandato `run_processes 3`.
b) *Indicar qué hará el primer proceso hijo creado durante la ejecución de `run_processes 3`.*
c) *Dibujar el árbol de procesos resultantes de la ejecución de `run_processes 3`.*
d) *¿Cuál sería el número total de procesos creados durante la ejecución de `run_processes 3`, sin contar el proceso padre original?*
e) *Razonar si al realizar la ejecución de `run_processes 3`, podrá existir algún proceso huérfano y/o algún proceso zombie. Se considera proceso huérfano aquel cuyo padre ha muerto, y es adoptado por el proceso `init`.*
f) *Si se incluye delante de la línea 16 el siguiente código:*

```
     act.sa_handler = tratar_alarma;
act.sa_flags = 0;
sigaction(SIGALRM, &act, NULL);
alarm(10);
while (pid != wait(&status)) continue;
return 0;
```

Se declara dentro del main(), la variable: `struct sigaction act;` y se declara como variable global `int pid;`, para que pueda ser vista por la función `tratar_alarma`, cuyo código se muestra a continuación:

```
void tratar_alarma(void) {
  kill(pid, SIGKILL);
}
```

En el caso de que se lleve a cabo la ejecución `run_processes 3`*, ¿cuál es el número máximo de procesos que podrían morir por la ejecución del servicio* `kill` *invocado en la función* `tratar_alarma`*?*

SOLUCIÓN

a) Se produciría un acceso a memoria incorrecto al acceder a `argv[1]`.

b) El primer proceso hijo creará otro proceso y ambos ejecutarán recursivamente `run_processes 2`.

c) El árbol de procesos se encuentra en la figura . Dicha figura también muestra los procesos que realizan un wait por el proceso hijo así como el valor que toma la variable `number` en cada uno de ellos.

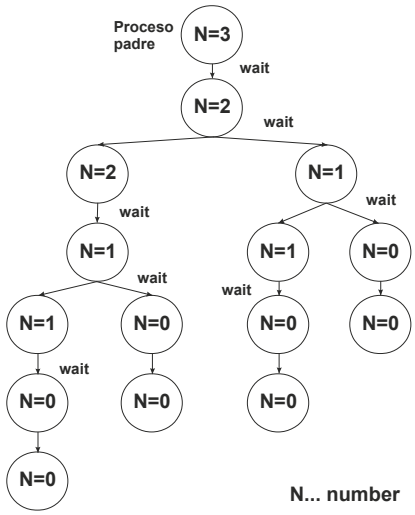


Figura 1.10

d) 14

e) En algún orden de ejecución se produciría la situación de procesos hijos huérfanos y procesos zombies. En el primer caso, los procesos padres que no hacen wait por sus procesos hijos (tal y como está representado en la figura), podrían morir antes que los procesos hijos correspondientes. En el segundo caso, si los procesos hijos por los cuales el proceso padre no hace un wait mueren, se convertirían en procesos zombies.

f) 7 (el mismo número de procesos que hacen wait por el proceso hijo y que tienen su pid).

Problema 1.11 (abril 2005)

Se tiene desarrollada una aplicación que realiza tres procedimientos que son independientes entre sí. Las cabeceras de estos procedimientos son las siguientes:

```
void Procedimiento1(void);
void Procedimiento2(void);
void Procedimiento3(void);
```

Suponga que los procedimientos tienen las siguientes fases de ejecución y que se usa E/S por DMA:

a) Procedimiento 1: 25 ms de CPU, lectura de dispositivo1 25 ms, 25 ms de CPU
b) Procedimiento 2: 30 ms de CPU, lectura de dispositivo1 35 ms, 20 ms de CPU
c) Procedimiento 3: 50 ms de CPU

Esta aplicación se ha desarrollado utilizando dos modelos diferentes, cuyo código se detalla a continuación:

```
/* Modelo 1 - Un único proceso */
1  int main (void)
2  {
3     Procedimiento1();
4     Procedimiento2();
5     Procedimiento3();
6     return 0;
```

```
7   }
```

```
/* Modelo 2 - Tres procesos concurrentes */
1  int main (void)
2  {
3     if (fork()!=0)
4     {
5        if (fork()!=0)
6           Procedimiento1();
7        else
8           Procedimiento3();
9     }
10    else
11    {
12       Procedimiento2();
13    }
14    wait(NULL);
15    return 0;
16 }
```

Se consideran los siguientes supuestos:

a) El sistema operativo emplea un algoritmo de planificación basado en prioridades (en caso de que existan varios procesos listos para ejecutar, se seleccionará el más prioritario). En el caso del Modelo 2 el proceso padre tiene prioridad 1, el primer proceso hijo prioridad 2 y el segundo proceso hijo prioridad 3 (números más bajos indican mayores prioridades).
b) Se tomará como t=0 el instante en el que comienza la ejecución de la línea 3 en el Modelo 1 y la ejecución de la línea 3 en el Modelo 2.
c) El SO no ejecuta ningún otro proceso y no se tendrá en cuenta ningún tipo de interrupción de reloj.
d) El tiempo de ejecución de los servicios del sistema operativo es el siguiente (estos tiempos incluyen la planificación y activación del siguiente proceso. En realidad el SO consume mucho menos tiemo que los indicados, se utilizan estas cifras para que la figura sea más clara):
- *Tratamiento de la llamada al sistema fork: 20 ms*
- *Tratamiento de la llamada al sistema para la lectura de dispositivo: 10 ms*
- *Tratamiento de cualquier otra llamada al sistema: 10 ms*
- *Tratamiento de interrupción de fin de operación de E/S: 15 ms*

e) Considere cualquier otro tiempo despreciable.

Se pide:

a) ¿Qué tiempo total tarda en ejecutar el Modelo 1?
b) ¿Qué jerarquía de procesos surge de la ejecución del Modelo 2?
c) ¿Qué sucede si estando en ejecución un proceso menos prioritario termina una operación de E/S de un proceso más prioritario?
d) En el Modelo 2, ¿quién está ejecutando en el instante 110 ms?
e) ¿Qué tiempo total tarda en ejecutar el Modelo 2?
f) En el Modelo 2, ¿qué pasa con el `wait()` de la línea 14 en el segundo proceso hijo creado?

Solución

a) El diagrama de tiempos se puede encontrar en la figura 1.11.

b) La jerarquía resultante es un proceso padre que ejecuta el Procedimiento1 con dos procesos hijo que ejecutan Procedimiento2 y Procedimiento3.

c) Al finalizar una operación de E/S se genera una interrupción, por lo que entra a ejecutar el sistema operativo. Cuando se termine el tratamiento de la interrupción se planificará al siguiente proceso a ejecutar, seleccionándose el más prioritario.

d) En el instante 110 está ejecutando el Sistema Operativo, tal y como se puede observar en la figura .

e) El diagrama de tiempos se puede encontrar en la figura 1.12.

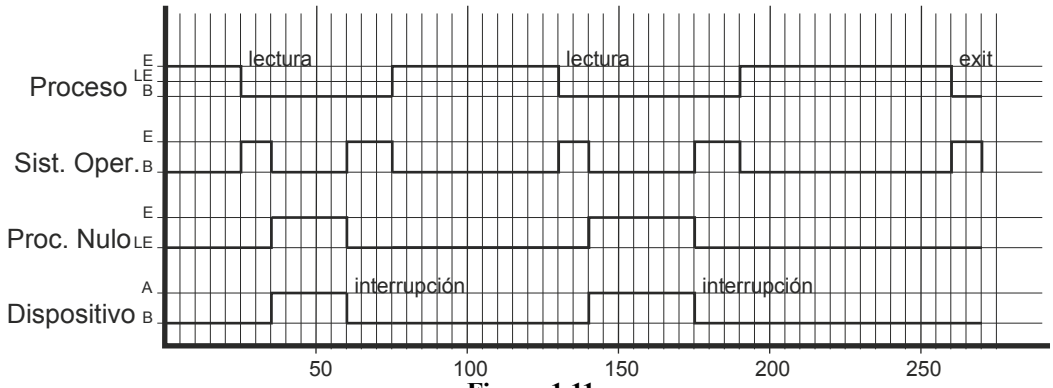


Figura 1.11

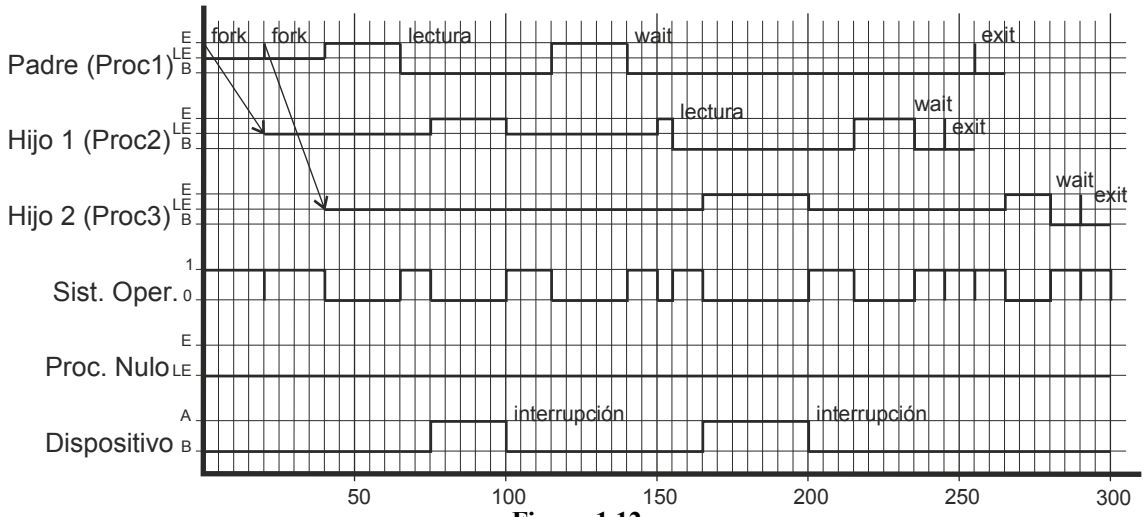


Figura 1.12

f) El segundo proceso hijo no es padre de ningún otro proceso, por lo que la llamada wait devuelve un error directamente.

Problema 1.12 (junio 2005)

Se pide implementar en C y para UNIX un programa con las siguientes características:

a) Se debe invocar de la siguiente manera:

```
verificar_actividad minutos mandato [args...]
```

b) Su objetivo es verificar, con cierta periodicidad, que un operario de una central nuclear está alerta.

c) El programa debe alternar fases de espera de los minutos especificados como primer argumento en la invocación, con fases de verificación de presencia del operario.

d) Para la fase de verificación deberá crearse un proceso hijo cuyo estado de terminación indique si todo ha ido de forma correcta (terminación 0) o si ha habido algún error (terminación no 0).

e) En la fase de verificación se debe presentar un mensaje como "`Introduzca 4:`", siendo el dígito mostrado el menos significativo del valor del pid del proceso hijo. El operario dispondrá de 30 segundos para introducir correctamente el dígito presentado. Durante este tiempo, cada 2 segundos se llamará la atención del operario con una señal acústica producida por la función `void beep (void)` de la biblioteca `curses`.

f) Si la segunda fase va mal el proceso principal deberá terminar ejecutando el mandato (con sus respectivos argumentos) indicado en la invocación del programa.

g) Tanto en el proceso padre como en los procesos hijos creados se deberán enmascarar las señales generables desde el teclado `INT` y `QUIT`.

***RECUERDE QUE:** Cuando llega una señal a un proceso que está bloqueado en una llamada al sistema, la llamada se aborta, devolviendo un -1, y la variable `errno` toma el valor `EINTR`.*

Solución

```
#include <curses.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int beep(void);

void tratar_alarma()
{
   beep();
}

int main(int argc, char **argv) {
   int status;
   int tiempo;
   int i;
   sigset_t mascara;
   struct sigaction act;
   char car;
   char ultimoDigitoPid;
   pid_t pid;
   int resul;

   sigemptyset(&mascara);
   sigaddset(&mascara, SIGINT);
   sigaddset(&mascara, SIGQUIT);
   sigprocmask(SIG_SETMASK, &mascara, NULL);

   tiempo = atoi(argv[1])*60;

   while(1)
   {
      sleep(tiempo);
      pid = fork();
      switch(pid)
      {
         case -1: /* error del fork() */
            perror ("fork");
            exit(1);
         case 0: /* proceso hijo */
            /* La máscara de señales se mantiene, no hay que cambiarla */
            /* Establecer el manejador */
            act.sa_handler = tratar_alarma; /* función a ejecutar */
            act.sa_flags = 0; /* ninguna acción específica */
            sigaction(SIGALRM, &act, NULL);

            ultimoDigitoPid = '0' + getpid() % 10;
            printf("\nIntroduce %c:", ultimoDigitoPid);

            for (i=0;i<15;i++)
            {
               alarm(2);

               do
               {
                  /* Si vence la alarma en resul obtenemos un -1 */
                  resul = read(0,&car,1);
```

```
            } while (resul >=0 && car != ultimoDigitoPid);

            if (car==ultimoDigitoPid)
            {
               return 0;
            }
         }
         exit(1); /* Salimos dando error */
      default:    /* padre */
         wait(&status);
         if (status != 0)
         {
            execvp(argv[2], &argv[2]);
            perror("exec");
            exit(1);
         }
   }
  }
  return 0;
}
```

Problema 1.13 (abril 2006)

Dados los siguientes códigos:

Fichero arbol.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int total, x, i;

  total = atoi(argv[1]);
  i = 0;
  x = 5;

  while (i < total)
  {
     fork();
     i++;
     x = x + 2;
  }

  printf("%d \n", x);
  return 0;
}
```

Fichero arbolv2.c

```
void func(void)
{
     ...
     pthread_exit(0);
}
int main(int argc, char *argv[])
{
pid_t pid;
  int total, x, i, fd;
```

```
   pthread_attr_t attr;
   pthread_t thid1,thid2;

   pthread_attr_init(&attr);
   total = atoi(argv[1]);
   pid = getpid();
   i = 0;
   x = 5;
   fd = open("/home/paco/datos.txt", O_WRONLY)
   while (i < total)
   {
      fork();
      i++;
      x = x + 2;
   }
   if (getpid()!=pid)
   {
      pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
      pthread_create(&thid1, &attr, func, NULL);
      pthread_create(&thid2, &attr, func, NULL);
   }
   printf("%d\n", x);
   return 0;
}
```

Se generan los ejecutables arbol y arbolv2 de los ficheros anteriores.

a) Suponiendo que como procesos arbol se entienden todos los procesos creados durante la ejecución del programa, incluyendo al proceso inicial, ¿cuántos procesos arbol se crearán durante la ejecución de arbol 4?

b) ¿Qué valor de x imprimirá el segundo de los procesos creados directamente por el proceso inicial durante la ejecución de arbol 4?

c) ¿Cuántos procesos zombies podríamos llegar a tener, como máximo, durante la ejecución de arbol n, independientemente del valor de n?

d) ¿Cuántas llamadas al sistema close serían necesarias para que el fichero no tuviera ningún descriptor abierto al ejecutar arbolv2 n, independientemente del valor de n?

e) Se desea que el proceso arbol inicial, a los 10 segundos, escriba el número de procesos que ya han finalizado y termine su ejecución. Escriba el código que se debe incluir en el programa para conseguir dicho objetivo. Subraye las llamadas al sistema que utilice. Nota: para solucionar este apartado no se pueden utilizar pipes.

SOLUCION

a) La jerarquía de procesos que se obtiene es la que se puede ver en la figura 1.13. El número que se puede observar dentro de cada proceso corresponde al valor inicial de la variable i para dicho proceso.

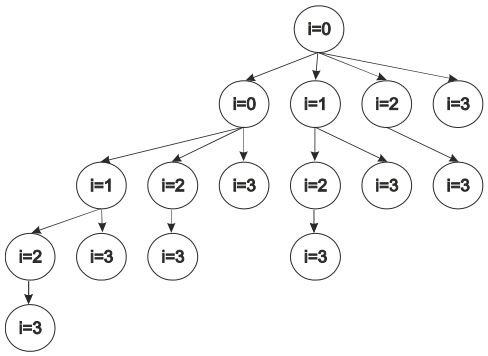


Figura 1.13

Es importante destacar que en el momento en que se realiza el fork el hijo se crea como una copia exacta del padre, pero a partir de dicho momento las variables del padre y del hijo son completamente independientes. Por tanto,

a partir del momento de la creación si el padre toca su variable i, este cambio no afecta a la variable i del hijo y viceversa.

b) El valor de x que se imprime por pantalla es 13.

Al comenzar x = 5. A continuación, el proceso inicial crea su primer proceso hijo y, a continuación, incrementa en 2 el valor de x, por lo que pasa a valer 7. En la segunda iteración, cuando x=7, crea a su segundo proceso hijo. Es decir, el segundo proceso hijo del proceso inicial comienza con x = 7.

Por último, el segundo proceso hijo realiza tres iteraciones en el bucle, por lo que el resultado final es de 13.

De hecho, si se realiza este mismo procedimiento para cualquier de los procesos, se podrá ver como el valor final que se imprime por pantalla es 13 en todos los casos. La única diferencia existente es en dónde se realizan los incrementos de x.

c) Podríamos llegar a tener a tener tantos procesos zombies como procesos hay en la jerarquía, ya que en ningún caso se está realizando un wait por ninguno de los hijos. De hecho, el proceso inicial también podría llegar a quedarse zombie porque, tal y como está definido el problema no es posible saber si su padre está o no haciendo un wait por él.

Por tanto, podríamos llegar a tener tantos procesos zombies como procesos arbol hay en la jerarquía de procesos que se genera.

d) Efectivamente sólo se está realizando un open del fichero, el que realiza el proceso inicial. Sin embargo, cada vez que se crea un nuevo hijo, se realiza una copia de la Tabla de Descriptores. Sin embargo, la creación de threads dentro de cada proceso no altera las Tablas de Descriptores.

Por tanto, para que el fichero no tenga ningún descriptor abierto habrá que hacer tantos close como procesos arbolv2.

e) Para que el proceso inicial escriba por pantalla a los 10 segundos el número de procesos que ya han terminado es necesario establecer una alarma (alarm) y capturarla (sigaction). Transcurrido el tiempo de la alarma se sacará el mensaje por pantalla.

La parte de contabilizar el número de procesos que ya han finalizado es más compleja. Lo que no se puede hacer, y es un error muy grave, es usar una variable global para llevar esta cuenta. Los procesos de nuestra jerarquía no comparten sus variables, una vez creados los nuevos procesos sus variables sólo le pertenecen a él, y los cambios que hagan sobre los valores de sus variables no afectan al resto de los procesos.

Otro error grave bastante común es que el proceso inicial realice un wait por el resto de los procesos. Si bien es posible realizar esta operación, sólo estaría esperando por sus cuatro hijos directos, pero no por el resto de procesos de la jerarquía. En este caso el contador de procesos del padre sólo podría llegar a tener un valor de 4.

Un posible procedimiento a utilizar es que los procesos, cuando finalicen, manden una señal al proceso inicial para indicarle esta situación. El proceso inicial puede tener una variable en la que ir contando el número de señales de este tipo que ha recibido. Cuando llegue la alarma, simplemente habrá que imprimir ese valor por pantalla y finalizar la ejecución.

Un posible código es el siguiente:

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define SENAL 40
int contador;

void tratar_alarma(int signum)
{
   printf("Contador = %d\n", contador);
   exit(0);
}

void incrementar_contador(int signum)
{
   contador++;
}

int main(int argc, char *argv[])
```

```
{
   pid_t pid;
   int total, x, i;
   struct sigaction act;

   total = atoi(argv[1]);
   i = 0;
   x = 5;
   contador = 0;

   /* establecer el manejador para SIGALRM */
   act.sa_handler = tratar_alarma;
   act.sa_flags = 0 ;
   sigaction(SIGALRM, &act, NULL);

   /* establecer el manejador para SIGUSR1 */
   act.sa_handler = incrementar_contador;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   sigaction(SENAL, &act, NULL);

   pid = getpid();

   /* Se crean los procesos hijos */
   while (i < total)
   {
      fork();
      i++;
      x = x + 2;
   }

   /* Si soy el proceso inicial establezco la alarma y espero */
   if (getpid()==pid)
   {
      alarm(10);
      while (1){
         pause();
      }
   }
   else /* Cualquier otro proceso: manda señal a proc ini */
   {
      kill(pid, SENAL);
   }

   return 0;
}
```

Nota avanzada: Tal y como está solucionado el ejercicio su ejecución funciona perfectamente porque los procesos envían una señal de tiempo real al proceso inicial y estas señales siempre se encolan. Estas señales se pueden reconocer porque su número es mayor o igual a 34. Sin embargo, si hubiéramos usado una señal como SIGUSR1, se podrían estar perdiendo señales. Según esta definido el estándar POSIX, si mientras se esta tratando una señal llega otra del mismo tipo, esta se encola, pero si en ese momento llegara otra mas, esa señal ya no se encola, con la correspondiente pérdida de señales.

A continuación se muestra otra posible solución al problema basada en ficheros. En este caso se crea un fichero auxiliar en modo O_APPEND en donde los procesos escriben un carácter cuando finalizan. Para saber el número de procesos que han finalizado nos basta con comprobar el tamaño final de fichero.

```
#include <signal.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```c
#include <sys/stat.h>
#include <fcntl.h>

int contador;

void tratar_alarma(int signum)
{
   struct stat buf;

   stat("aux.txt", &buf);
   printf("Contador = %d\n", buf.st_size);
   exit(0);
}

void incrementar_contador(int signum)
{
   contador++;
}

int main(int argc, char *argv[])
{
   pid_t pid;
   int total, x, i;
   struct sigaction act;
   int fd;

   total = atoi(argv[1]);
   i = 0;
   x = 5;
   contador = 0;

   fd = open("aux.txt", O_CREAT | O_APPEND | O_WRONLY | O_TRUNC, 0644);

   /* establecer el manejador para SIGALRM */
   act.sa_handler = tratar_alarma;
   act.sa_flags = 0 ;
   sigaction(SIGALRM, &act, NULL);

   pid = getpid();

   /* Se crean los procesos hijos */
   while (i < total)
   {
      fork();
      i++;
      x = x + 2;
   }

   /* Si soy el proceso inicial establezco la alarma y espero */
   if (getpid()==pid)
   {
      alarm(10);
      pause();
   }
   }
   else  /* Si soy cualquier otro proceso mando una señal al proceso inicial */
   {
      write(fd,"1",1);
      close(fd);
   }

   return 0;
}
```

Problema 1.14 (abril 2007)

El siguiente programa, denominado `watchdog_timer`, nos permite comprobar cada cierto tiempo que un operario está alerta en su terminal de trabajo. Este programa recibe dos argumentos: (i) `espera1`, que nos indica cada cuánto tiempo hay que verificar la presencia del operario y (ii) `espera2`, que nos indica el tiempo máximo que tiene el operario para responder correctamente. El código del programa, a falta de completar algunas partes, es el siguiente.

```
#define true 1
pid_t pid;

void tratar_alarma()
{
   // Código de tratamiento de la alarma
}

int main (int argc, char **argv)
{
   struct sigaction act;
   int num, ch, espera1, espera2, numErrores;
   int valor;

   espera1 = atoi(argv[1]);
   espera2 = atoi(argv[2]);

   while (true)
   {
      sleep(espera1);

      pid = fork();
      switch (pid)
      {
         case -1:
            perror("Error en la llamada fork");
            exit(1);
         case 0:
            numErrores = 0;
            while (true)
            {
               num = generar_numero();
               printf("Introduzca el número %d", num);
               scanf("%d\n", &ch);
               if (ch==num)
                  return 0;
               numErrores++;
               if (numErrores==5)
                  exit(1);
            }
         default:
            act.sa_handler = tratar_alarma;
            act.sa_flags = 0;
            sigaction(SIGALRM, &act, NULL);
            alarm(espera2);
            while (pid != wait(&valor));
               // Código para la comprobación de la finalización del proceso hijo
      }
   }
   return 0;
}
```

a) *¿Están manejando correctamente la alarma el proceso padre y los procesos hijo que se van creando?*
b) *Indica el contenido correcto de la pila nada mas comenzar la ejecución de* `watchdog_timer 240 30`
c) *Para verificar la presencia del operario se crea un proceso hijo, y el proceso padre se queda bloqueado esperando por él. Sin embargo, es necesario verificar que el proceso hijo ha finalizado correctamente y no por la recepción de una señal. Codifique el fragmento de código que deberíamos añadir a nuestro programa después de la sentencia* `while(pid!=wait(&valor))`.
d) *En caso de que el operario no responda correctamente en el tiempo máximo establecido, se activa la función tratar_alarma, que debe matar al proceso hijo y, a continuación, ejecutar un programa denominado alarmon que no recibe ningún parámetro de entrada. Codifique la función tratar_alarma.*
e) *Queremos que nuestro programa no se vea afectado por las señales generadas desde teclado con* `Ctrl+C` *y con* `Ctrl+\`. *Codifique, utilizando una máscara de señales, una posible solución, teniendo en cuenta que tanto el proceso padre como los procesos hijo que se vayan creando deben estar protegidos ante dichas señales.*
f) *En esta ocasión, deseamos que tanto el proceso inicial como los procesos hijo estén protegidos ante las señales generadas con* `Ctrl+C` *y con* `Ctrl+\`, *pero no queremos proteger la ejecución del programa alarmon. Para resolver este problema, ¿sería posible codificar una solución basada en máscaras?*

SOLUCIÓN

a) Los procesos hijo no tienen nada que ver con la alarma, ya que quien la activa es el proceso padre. Por tanto, en el caso de los procesos hijo todo es correcto. Sin embargo, el proceso padre no maneja correctamente la alarma cuando el proceso hijo finaliza antes del vencimiento de la misma. De esta manera, y justo después de la instrucción `while (pid != wait(&valor))`, el proceso padre debería ejecutar la sentencia `alarm(0)` para desconectar la alarma. En caso de que la alarma no fuera desconectada se ejecutaría de forma incorrecta la función de tratamiento de alarma `tratar_alarma()`.

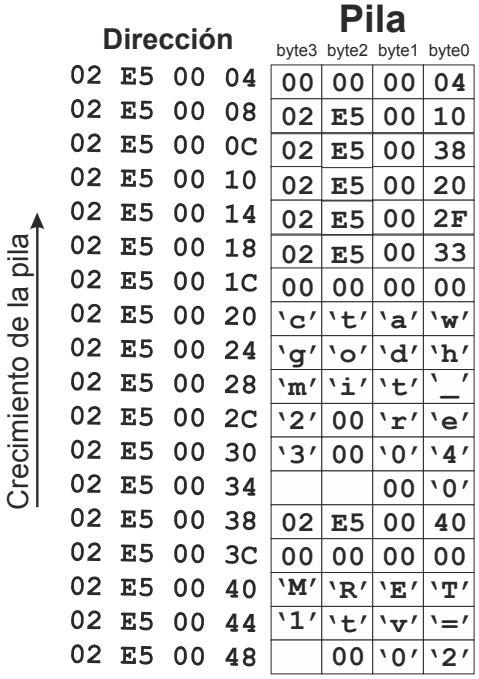


Figura 1.14

b) En la figura 1.14 se puede observar el contenido completo de la pila. Se debe tener en cuenta que el número de parámetros de invocación al programa es 4, siendo `argv[0]` el nombre del programa y `argv[3]=NULL`. La parte superior de la pila debe contener, además del número de parámetros del programa, un puntero a los argumentos y un puntero al entorno. Por último, los valores de los parámetros y de las variables de entorno deben estar separados entre sí con valores 00, de forma que se pueda saber dónde finaliza cada uno de ellos.

c) El código a insertar sería el siguiente:

```
if (WIFEXITED(valor))
  if (WEXITEDSTATUS(valor)==0)
    continue;
```

```
   else
     exit(1);
else if (WIFSIGNALED(valor))
   exit(2);
```

En primer lugar debemos comprobar que el proceso hijo ha finalizado correctamente, para lo cual verificamos que la macro `WIFEXITED` ha devuelto un valor positivo. Si esta verificación es positiva, debemos comprobar que el valor de finalización del proceso hijo es 0. Este valor se comprueba a través de la macro `WEXITEDSTATUS`. En cualquier caso, si el proceso hijo ha finalizado por la recepción de una señal (macro `WIFSIGNALED`), terminamos la ejecución devolviendo un código de error.

d) El código de la función `tratar_alarma` sería el siguiente:

```
void tratar_alarma() {
   kill(pid, SIGKILL);
   execlp("alarmon", "alarmon", NULL);
   exit(1);
}
```

La función comienza con el envío de la señal `SIGKILL`, a través de la llamada al sistema `kill`, al proceso hijo. El envío de esta señal garantiza que el proceso hijo morirá, ya que esta señal no se puede capturar. A continuación, a través del servicio `execlp` se pasa a ejecutar el programa `alarmon` que, tal y como se dice en el enunciado, no recibe ningún parámetro de entrada. En caso de error en la llamada `exec`, se finaliza la ejecución a través de la llamada `exit`.

e) Debemos establecer una máscara para la señales `SIGINT` (`Ctrl+C`) y `SIGQUIT` (`Ctrl+\`). Con establecer la máscara en el proceso padre es suficiente, ya que los procesos hijo que se vayan creando heredarán esta máscara. Se deberá insertar al comienzo del main el siguiente fragmento de código:

```
sigset_t mascara;
sigaddset(&mascara, SIGINT);
sigaddset(&mascara, SIGQUIT);
sigprocmask(SIG_SETMASK, &mascara, NULL);
```

f) No, ninguna solución basada en máscaras es factible. Si se usa una máscara en el proceso padre, tanto los procesos hijo como el programa alarmon heredarían dicha máscara. Si queremos que las señales `SIGINT` y `SIGQUIT` puedan llegar al programa alarmon, deberíamos quitar la máscara justo antes de hacer el `exec`, lo que significaría que la señales pendientes (en caso de existir) llegarían al proceso padre. Por tanto, el proceso padre no estaría correctamente protegido.

Problema 1.15 (junio 2009)

Dado el siguiente programa:

```
#define NPH 10
int main(int argc, char *argv[])
{
int i, sig;
int fd[NPH][2];

   for (i=0; i<NPH; i++)
     pipe(fd[i]);
   for (i=0; i<NPH; i++) {
     if (fork()==0) {
        sig = (i+1)%NPH;
        <<CÓDIGO_A_RELLENAR>>
        /*  Ejecución de cada mandato.  *
         *   Por simplicidad no se muestra  *
         * el paso de argumentos a execvp. */
        execvp(...);
        perror("exec");
        exit(1);
```

```
      }
   }
   write(fd[0][1], "ABCDEFGHIJ", 10);
   for (i=0; i<NPH; i++) {
      close(fd[i][0]);
      close(fd[i][1]);
   }
   while (wait(NULL)>0) ;
   return 0;
}
```

Se pretende que los procesos hijo ejecuten mandatos estándar conectados en una secuencia cíclica, esto es, el primer hijo mandará su salida a la entrada del segundo, el segundo mandará al tercero, y así sucesivamente, hasta el último hijo, que tendrá su salida estándar conectada a la entrada estándar del primer hijo.

Se pide:

a) Dibuje el diagrama de procesos en el momento en que todos los hijos están en ejecución. Muestre la jerarquía de procesos y los mecanismos de comunicación entre ellos y las asociaciones de los primeros con los segundos mediante descriptores.

b) Implemente el `<<CÓDIGO_A_RELLENAR>>`, para que se puedan ejecutar mandatos estándar en secuencia cíclica.

c) Implemente el mandato estándar `cat` en la versión sin argumentos.

d) Si el primer hijo ejecuta el mandato `ls` y el resto el mandato `cat`, responda:

1. ¿Cuándo finalizan todos los procesos?
2. ¿Qué estarán haciendo todos los procesos cuando se haya puesto todo en marcha?
3. ¿Cómo se llama la situación que alcanzan estos procesos?

e) Responda a las preguntas del apartado anterior para el caso en que el proceso padre no realice la llamada `write` y todos los hijos ejecuten el mandato `cat` (sin argumentos).

f) Responda las mismas preguntas para el caso en que el proceso padre sí realice la llamada `write` y todos los hijos ejecuten el mandato `cat` (sin argumentos).

g) En el escenario anterior, explique qué ocurre en el caso de que al hijo 1 le llegue una señal que lo mata. Indicar:

1. ¿En qué orden terminarían todos los procesos?
2. ¿En qué estado finalizarían cada uno de ellos?

Solución.-

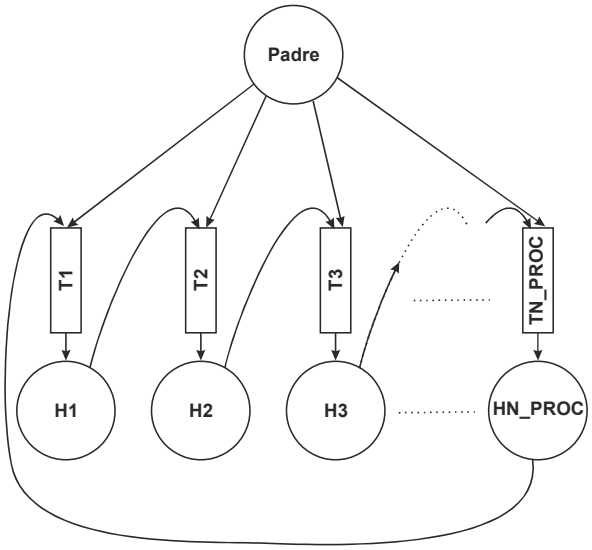


Figura 1.15

a) En la figura 1.15 se muestra el diagrama de procesos así como las tuberías que permiten comunicar dichos procesos.

b) En el código a rellenar hay que redirigir la entrada y salida de cada proceso para que se comunique con el proceso anterior y posterior. Además, se tienen que cerrar todos los descriptores abiertos que no se vayan a utilizar. El código quedaría del siguiente modo:

```
close(0);
dup(fd[i][0]);
close(1);
dup(fd[sig][1]);
for (i=0; i<NPH; i++){
   close(fd[i][0]);
   close(fd[i][1]);
}
```

c) El mandato `cat` escribe por la salida estándar lo que lee por la entrada estándar:

```
#define BUFSIZE 1024
int main(int argc, char *argv[])
    {
char buffer[BUFSIZE];
int leidos;

while ((leidos = read(0, buffer, BUFSIZE)) > 0)
   write(1,buffer,leidos);

if (leidos < 0)
   return 1;
else
   return 0;
    }
```

d) El primer hijo ejecuta el mandato `ls`, teniendo redirigida la salida al extremo de escritura de la primera tubería. El segundo proceso ejecuta el mandato `cat`, escribiendo por la salida estándar (asociada al extremo de escritura de la segunda tubería) lo que previamente leyó por su entrada estándar (asociada al extremo de lectura de la primera tubería). Esto se repite para cada uno de los procesos de manera circular. Por tanto:

1. El primer proceso termina cuando ejecuta el mandato `ls`. El segundo proceso lee de su entrada estándar la salida de la ejecución de este mandato, que lo escribe por su salida estándar y muere. Así sucesivamente, hasta que el último de los procesos lee de su entrada estándar y al escribir en su tubería, como no

tendrá lectores (salvo que el proceso padre no hubiera cerrado el descriptor correspondiente), terminará con error.

2. Una vez que los procesos arrancan, el primero ejecutará el mandato `ls`. Los demás ejecutarán el mandato `cat`, leyendo de la entrada estándar asociada a la tubería correspondiente. El proceso padre cerrará los descriptores y esperará a que finalicen todos los procesos hijos.

3. Los procesos finalizan de forma correcta, excepto el último que finaliza por un error (le llega la señal SIGPIPE) al intentar escribir sobre una tubería sin lectores (excepto en el caso de que escriba en la tubería antes de que el proceso padre haya cerrado el descriptor correspondiente). Hay también una situación en la que se produce un interbloqueo. Si el mandato `ls` produce muchos datos, éstos se escribirán de manera sucesiva en todas las tuberías, hasta llegar a la última, que como no se vacía provoca de manera encadenada que todas las tuberías se llenen.

e) En este segundo escenario, todos los procesos hijos estarían bloqueados esperando leer algo por la entrada estándar.

1. Los procesos estarían bloqueados eternamente, excepto si les llega una señal que les mata.

2. Una vez que los procesos arrancan, todos los procesos hijos estarían bloqueados en la llamada `read` del mandato `cat`. El proceso padre estaría bloqueado esperando a la finalización de sus hijos.

3. Los procesos estarían en una situación de *deadlock*.

f) En el tercer escenario, todos los procesos hijos estarían continuamente pasándose los datos escritos por el padre.

1. Los procesos no finalizarían, excepto si les llega una señal que les mata.

2. Una vez que los procesos arrancan, todos los procesos hijos estarían ejecutando el mandato `cat`. El proceso padre estaría bloqueado esperando a la finalización de sus hijos.

3. Todos los procesos llevan a cabo su trabajo. Se encuentran en un bucle infinito de trabajo.

g) Al proceso hijo le llega una señal que provoca que muere. El orden y el estado de terminación de los procesos depende del orden de planificación de los mismos por parte del sistema operativo. Si por ejemplo, después de morir el primer proceso hijo, se planifica el último proceso hijo, éste morirá al intentar escribir en una tubería sin lectores. Le llegaría la señal SIGPIPE. Si después, se planifica el penúltimo proceso hijo, le ocurriría lo mismo y así sucesivamente. Por el contrario, si después de morir el primer proceso hijo, se planifica el segundo, éste terminará de manera correcta, pues ya no hay más procesos escritores sobre la tubería de la que lee y por tanto finalizará y morirá correctamente. Lo mismo le ocurrirá al tercer proceso si se planificara a continuación. Todo depende si el proceso posterior al planificado ha finalizado o no. Si ha finalizado, el proceso correspondiente terminará de manera incorrecta al intentar escribir sobre una tubería sin lectores. Si no ha terminado y el proceso anterior sí, terminará correctamente al no tener ya más procesos escritores sobre la tubería de la que está leyendo.

Problema 1.16 (abril 2010)

Dado el siguiente fuente:

```
01 int bucle_fork(int M)
02 {
03   int n;
04 for(n = 0; n < M; n++){
05   switch(fork()){
06   case -1:
07      perror("fork()");
08      return -1;
09   case 0:
10
11   default:
12
13   }
14 }
15 return 0;
16 }
```

a) Dibujar el diagrama del árbol genealógico de procesos (relaciones padre-hijo) que resulta al invocar "bucle_fork(3)", incluyendo el proceso que originalmente hace la invocación. Identifique a cada hijo con el valor de n en el momento de su creación.

b) ¿Cuántos procesos (incluyendo al primer padre) se producen al invocar "bucle_fork(M)"? Suponer que todos los fork() retornan sin error.

c) ¿Qué código hay que escribir en la línea 10 y en la línea 12 para conseguir crear solo M procesos hijo (todos hijos del mismo padre)?

d) ¿Qué código hay que escribir en la línea 10 y en la línea 12 para conseguir que el proceso inicial tenga una descendencia de M generaciones de procesos (padre → hijo → nieto → bisnieto ...)?

e) ¿Cómo temporizar para que cada hijo muera a los n+1 segundos? Suponer que el proceso inicial, ante la señal SIGALRM, realiza la acción por defecto (terminar)

f) ¿Qué comportamiento obtenemos si línea 10 = "exit(0);" línea 12 = "pause(); wait(NULL);"?

Solución

a)

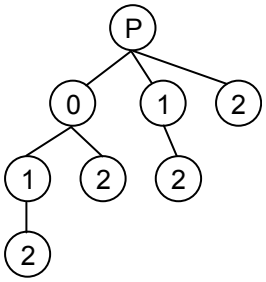


Figura 1.16

b) 2^M

c) En general:

línea 10 = Debe acabar en alguna sentencia que termine al hijo (que no tenga descendencia)

línea 12 = El padre debe continuar iterando

Por ejemplo.

línea 10 = "pause(); exit(0);"

línea 12 = "break;"

d) En general:

línea 10 = El hijo debe continuar iterando (para seguir generando la siguiente generación)

línea 12 = El padre debe terminar

Por ejemplo.

línea 10 = "break;"

línea 12 = "exit(0);"

e) En general:

línea 10 = Dado que cada hijo hereda la disposición de señales de su padre, ante SIGALRM terminará, por lo que basta con solicitar SIGALRM a los n+1 segundos. Cuando reciba SIGALRM, terminará.

línea 12 = El padre debe seguir iterando, para que se generen los hijos.

Por ejemplo:

línea 10 = "alarm(n+1);"

línea 12 = "break;"

f) El primer hijo se convierte en zombie. En efecto, el primer hijo termina, pero el padre no está ejecutando wait(), sino pause(), por lo que está esperando la llegada de la siguiente señal. En principio no se indica que el padre haya

armado ninguna señal. Cuando llegue (si llega) una señal al padre puede que el padre termine involuntariamente (por ejemplo ante una señal no armada cuyo comportamiento por defecto sea la terminación) y wait(NULL) nunca ejecute, por lo que el proceso Init adopta al zombie, ejecuta wait() y el zombie desaparece. Si el padre no termina, ejecuta wait(NULL) y el zombie desaparece igualmente.

Problema 1.17 (junio 2010)

Dados los programas programa_a y programa_b, rellenar las casillas vacías de la tabla 1.1, indicando el tiempo que dura la ejecución del proceso hijo de programa_a y la causa de su terminación:

- *Suponer despreciables todos los tiempos de ejecución (SO, otros procesos, etc.) distintos de los indicados en las invocaciones a `sleep()`.*
- *Suponer que la acción por defecto al recibir cualquier señal es terminar involuntariamente el proceso.*

programa_a:

```
void funcion1(int s){ }
int main(void){
   pid_t p;
   struct sigaction act;
   sigemptyset(&act.sa_mask);
   act.sa_handler = &funcion1;
   act.sa_flags = 0;
   sigaction(SIGUSR1, &act, NULL);
   p = fork();
   switch(p){
   case -1:
      perror("fork() = -1");
      exit(1);
   case 0:
      sleep(2);
      act.sa_handler = SIG_IGN;
      sigaction(SIGUSR2, &act, NULL);
      sleep(2);
      execl("./programa_b", "programa_b", (char *)NULL);
      perror("exec");
      exit(1);
   default:
      sleep(T1);
      kill(p, SIG_X);
      sleep(T2);
      kill(p, SIGQUIT);
      wait(NULL);
   }
   return 0;
}
```

programa_b:

```
int main(void){
   sleep(2);
   return 0;
}
```

SIG_X	T1(s)	T2(s)	Tiempo vida hijo (s)	Causa terminación hijo
SIGUSR1	1	6		
	3	5		
	3	1		
	5	4		
SIGUSR2	1	5		
	3	4		
	5	2		
SIGINT	1	2		
	5	3		
SIGKILL	5	2		

Tabla 1.1

SOLUCIÓN

SIG_X	T1(s)	T2(s)	Tiempo vida hijo (s)	Causa terminación hijo
SIGUSR1	1	6	5 (a)	Terminación voluntaria normal
	3	5	5 (a)	Terminación voluntaria normal
	3	1	4 (a)	Acción por defecto al recibir SIGQUIT
	5	4	5 (a)	Acción por defecto al recibir SIGUSR1
SIGUSR2	1	5	1 (a)	Acción por defecto al recibir SIGUSR2
	3	4	6	Terminación voluntaria normal
	5	2	6	Terminación voluntaria normal
SIGINT	1	2	1	Acción por defecto al recibir SIGINT
	5	3	5	Acción por defecto al recibir SIGINT
SIGKILL	5	2	5	Acción por defecto al recibir SIGKILL

Tabla 1.2

(a) La señal interrumpe la temporización de `sleep()` en proceso hijo, pero sin terminar el proceso, por lo que parte de la temporización de sleep no se contabiliza en la duración total del proceso hijo.

Problema 1.18 (sep-2010)

Dado el siguiente programa:

```
int main(void){
   int s, n, k = 0;
   pid_t p1;
   for(n = 0; n < 3; ){
      p1 = fork();
      switch(p1){
      case -1:
         perror("fork"); exit(1);
      case 0:
         if(getpid()%2 == 0)
            exit(0);
         else
            exit(1);
      default:
         k++;
         wait(&s);
         if(WIFEXITED(s))
            if(WEXITSTATUS(s) == 0)
               n++;
      }
   }
   printf("k = %i\n", k);
   return 0;
}
```

a) ¿Qué contiene la variable p1?

b) ¿Qué sentencias del cuerpo del switch corresponden al proceso hijo?

c) ¿Qué función realiza el proceso hijo? (Recuerde que "a % b" es el resto de la división entera de a entre b)

d) ¿Qué función realiza la macro WEXITSTATUS()?

e) ¿Qué operaciones hace el proceso padre tras crear al proceso hijo?

f) ¿Cuántos procesos hijo pueden existir al mismo tiempo? ¿Porqué?

g) ¿Qué información contienen las variables k y n al final de cada iteración del bucle for?

h) ¿Puede en algún caso caso iterar indefinidamente el bucle for? Si es así, ¿En cuál?

i) ¿Qué información contiene la variable k al final del programa?

j) ¿Qué escribe el programa en la salida estándar si el sistema operativo asigna la siguiente secuencia de PIDs a los procesos hijo: 29180, 29181, 29182, 29183, 29184, 29185, 29186, 29187, etc.?

SOLUCIÓN

a) El código retornado por fork(). En el proceso hijo vale 0 y en el proceso padre contiene el PID del proceso hijo.

b) Las sentencias de la etiqueta "case 0": if(getpid()%2 == 0) exit(0); else exit(1);

c) El proceso hijo termina con código de terminación 0 si su PID es par y con 1 si es impar

d) Retorna el código de terminación del hijo.

e) Incrementa k, espera a la terminación del hijo y si éste termina con código 0 (pid par), entonces incrementa n.

f) Solo uno, por que el proceso padre espera a la terminación de cada hijo.

g) k = Número de procesos hijo creados hasta el momento

n = Número de procesos hijo que han tenido pid par hasta el momento.

h) Sí. En el caso en que n siempre sea menor que 3, es decir, en el que el sistema operativo no asigne más de 2 pid pares a los sucesivos procesos hijo.

i) k = Número de procesos hijo que ha sido necesario crear para conseguir que 3 de ellos tengan pid par.

j) k = 5

Problema 1.19 (nov-2010)

a) Escribir un programa que cree N procesos hijo (hijos del proceso principal). Los hijos se deben comunicar con el padre mediante un pipe único, en el que escribe cada hijo y sólamente lee el padre. T segundos después de crear los procesos hijo el padre enviará a cada hijo una señal SIGUSR1. Cada hijo, al recibir tal señal, escribirá en el pipe su número de hijo mediante un carácter único 0, 1...N-1 y terminará (suponer que N < 11 para simplificar esta eta - pa). Despues el padre leerá todos los caracteres y terminará.

NOTA: Definir N y T como constantes internas del programa.

b) Modificar el programa anterior para que la temporización de T segundos empiece antes de crear los procesos hijo.

SOLUCIÓN

a)

```
#define N 10
#define T 3
int f[2];
char buf;
int i;

void f1(int s){
   write(f[1], &buf, sizeof(buf));
}

int main(void){
```

```
   pid_t p[N];
   if (pipe(f) < 0){
    perror("pipe\n"); return 1;
   }
   struct sigaction a1;
   sigemptyset(&a1.sa_mask);
   a1.sa_handler = f1;
   a1.sa_flags = 0;
   sigaction(SIGUSR1, &a1, NULL);
   for(i = 0; i < N; i++){
      if((p[i] = fork()) == -1){
         perror("fork"); return 1;
      }else if(p[i] == 0){
         close(f[0]);
         buf = '0' + i;
         pause();
         return 0;
      }
   }
   close(f[1]);
   sleep(T);
   for(i = 0; i < N; i++){
      kill(p[i], SIGUSR1);
   }
   for(i = 0; i < N; i++){
      if(read(f[0], &buf, 1) < 0){
         perror("read"); return 1;
      }
   }
   return 0;
}
```

b) En paralelo con la temporización se deben crear los hijos, por lo que no podemos usar sleep(T). El padre debe temporizar con alarm(T) antes del primer for (armando previamente SIGALRM). Cada hijo no hereda las señales pendientes del padre, por lo que SIGALRM no se hereda y llegará exclusivamente al padre, permitiendo un funcionamiento correcto. La acción asociada a SIGALRM serán los bucles de envío de SIGUSR1 a hijos y lectura del pipe.

```
#define N 10
#define T 3
int f[2];
char buf;
int i;
pid_t p[N];


void f2(int s){
   for(i = 0; i < N; i++){
      kill(p[i], SIGUSR1);
   }
   for(i = 0; i < N; i++){
      if(read(f[0], &buf, 1) < 0){
         perror("read"); exit(1);
      }
   }
}


void f1(int s){
   write(f[1], &buf, sizeof(buf));
}

int main(void){

   struct sigaction a1;
   sigemptyset(&a1.sa_mask);
   a1.sa_handler = f2;
   a1.sa_flags = 0;
```

```
   sigaction(SIGALRM, &a1, NULL);

   sigemptyset(&a1.sa_mask);
   a1.sa_handler = f1;
   a1.sa_flags = 0;
   sigaction(SIGUSR1, &a1, NULL);


   if (pipe(f) < 0){
    perror("pipe\n"); return 1;
   }

   alarm(T);
   for(i = 0; i < N; i++){
      if((p[i] = fork()) == -1){
         perror("fork"); return 1;
      }else if(p[i] == 0){
         close(f[0]);
         buf = '0' + i;


         pause();
         return 0;
      }

   }
   close(f[1]);
   pause();
   return 0;
}
```

Problema 1.20 (febrero 2011)

Dado el siguiente programa:

```
#define N 10
int i, p1[2];
pid_t h1;
char c1;

void f1(int s){
   c1 = '0' + i - 1;
   write(p1[1], &c1, 1);
}
void f2(int s){
   for(i = 0; i < N; i++) /*I*/
      read(p1[0], &c1, 1);
}
int main(void){
   struct sigaction a1;
   sigemptyset(&a1.sa_mask);
   a1.sa_handler = f1;
   a1.sa_flags = 0;
   sigaction(SIGUSR1, &a1, NULL); /*A*/

   pipe(p1); /*B*/
   for(i = 0; i < N; i++) /*C*/
      if((h1 = fork()) > 0)
         break;

   if(i == 0){ /*D*/
      a1.sa_handler = f2;
      sigaction(SIGUSR1, &a1, NULL);
```

```
      pause();
      waitpid(h1, NULL, 0);
   }else if(i < N){ /*E*/
      pause();
      kill(getppid(), SIGUSR1);
      waitpid(h1, NULL, 0);
   }else{ /*F*/
      sleep(1);
      f1(0);
      kill(getppid(), SIGUSR1);
   }
   return 0;
} //(FIN PROGRAMA)
```

k) ¿Qué acción realiza la sentencia A? ¿Qué consecuencias tiene la acción de A en los procesos hijo del proceso principal (si los hay)?
l) ¿Qué acción realiza la sentencia B? ¿Qué consecuencias tiene en los procesos hijo (si los hay)?
m) ¿Cuántos procesos genera el bucle C? ¿Qué relaciones padre-hijo hay entre ellos?
n) ¿Qué proceso ejecuta el cuerpo del if de D?
o) ¿Qué procesos ejecutan el cuerpo del if de la sentencia E?
p) ¿Que proceso ejecuta el cuerpo de else de la sentencia F?
q) ¿Cuántas señales SIGUSR1 se envían en total? Explicar su origen, destino y en qué orden se envían.
r) ¿En qué orden acceden al pipe p1 los procesos creados por fork? ¿Leen, escriben o ambas cosas?
s) ¿Qué secuencia de caracteres se ha leído del pipe p1 al terminar el for de la línea I?
t) ¿Garantiza este código los mismos resultados en un sistema muy cargado con otros procesos? ¿Porqué? Si no es así, proponer una solución.

SOLUCIÓN

a) ¿Qué acción realiza la sentencia A? ¿Qué consecuencias tiene la acción de A en los procesos hijo del proceso principal (si los hay)?

Arma la señal SIGUSR1 con la función f1. Los procesos hijo heredan el armado de la señal SIGUSR1.

b) ¿Qué acción realiza la sentencia B? ¿Qué consecuencias tiene en los procesos hijo (si los hay)?

Crea un fichero pipe cuyos descriptores están en registrados en p1. Los hijos heredan el pipe abierto.

c) ¿Cuántos procesos genera el bucle C? ¿Qué relaciones padre-hijo hay entre ellos?

Genera N procesos. N generaciones a partir del proceso principal (cada generación tiene un único hijo).

d) ¿Qué proceso ejecuta el cuerpo del if de D?

El proceso principal.

e) ¿Qué procesos ejecutan el cuerpo del if de la sentencia E?

Los procesos hijo de generación i = 1, 2 ...N-1, o sea, todas las generaciones excepto la última.

f) ¿Que proceso ejecuta el cuerpo de else de la sentencia F?

El proceso de la última generación (i = N), último hijo creado.

g) ¿Cuántas señales SIGUSR1 se envían en total? Explicar su origen, destino y en qué orden se envían.

En total se envían N señales SIGUSR1. Se envía una desde cada proceso hijo creado por fork. El destino es su proceso padre, que espera a su recepción para enviar a su vez su señal. El orden de envío comienza por el último hijo (i = N) y va ascendiendo por el árbol de procesos, de cada hijo su padre, hasta llegar al proceso principal.

h) ¿En qué orden acceden al pipe p1 los procesos creados por fork? ¿Leen, escriben o ambas cosas?

En el mismo orden de la cadena de señales, ascendente por el árbol de procesos, comenzando por el último hijo (i = N) y terminando en el proceso principal (i = 0). Cada proceso solo escribe el caracter correspondiente al dígito i-1.

i) ¿Qué secuencia de caracteres se ha leído del pipe p1 al terminar el for de la línea I?

9876543210

j) ¿Garantiza este código los mismos resultados en un sistema muy cargado con otros procesos? ¿Porqué? Si no es así, proponer una solución.

No lo garantiza. Si alguna de las señales llega antes de que se ejecute su correspondiente pause(), el proceso de destino queda esperando indefinidamente y no enviará nunca la señal a su padre. Esto puede ocurrir si el proceso de destino tarda en entrar a ejecución en un sistema muy cargado. La solución consiste en eliminar los pause() y mover los kill(getppid(), SIGUSR1) al final de f1(). Así la función de armado f1() maneja completamente la cadena de señales. Los procesos pueden recibir la señal mientras esperan en waitpid().

Problema 1.21 (abril 2011)

Dado un planificador ROUND ROBIN con rodaja de tiempo de 3ms, obtener el cronograma de ejecución de los siguientes procesos y fases de ejecución:

Proceso A: 5ms de CPU, lectura de 5ms sobre el dispositivo D1, 3ms de CPU

Proceso B: 2ms de CPU, escritura de 4ms sobre el dispositivo D2, 1ms de CPU

Consideraciones: Al comenzar, asignar el procesador al proceso A y colocar el proceso B en la cola de listos. Asignar 1ms en cada intervención del SO. El sistema no tiene que atender a ningún otro evento o proceso de usuario. Los tiempos de las operaciones de entrada/salida indicadas comienzan al terminar la intervención previa del SO.

SOLUCIÓN

(sombreado = origen del siguiente cambio de estado)

t(ms) 0 3 4 6 7 9 10 11 12 13 14 15 16 19

dt	3	1	2	1	2	1	1	1	1	1	1	1	3	
A	E	p	p	p	E	-	-	-	-	-	-	-p	E	fin
B	p	p	E	-	-	-	-	-p	E	fin				
SO	p	E	p	E	p	E	p	E	p	E	p	E	p	
nul	p	p	p	p	p	p	E	p	p	p	E	p	p	
D1							1	1	1	1	1			
D2					2	1	1							

"dt" = duración del estado (columna de la tabla), "nul" = proceso nulo

"E"= Ejecutando, "p"= preparado (listo), "-" = bloqueado

D1 y D2: <número> = duración de cada intervalo de tiempo

Problema 1.22 (abril 2011)

Escribir un programa que realice lo siguiente. El programa genera un único proceso, que ejecuta 3 tareas cuyo código fuente ya está programado en funciones que tienen los siguientes prototipos:

```
int tarea_A(void);
int tarea_B(void);
int tarea_C(int n);
```

Sean a, b y c los valores que retornan la tarea_A, tarea_B y tarea_C, respectivamente. Si a > 0, la tarea C debe ser invocada con el valor n = a. Sino, debe ser invocada con n = b. Las tareas no tienen ninguna otra dependencia entre ellas. El proceso debe maximizar la concurrencia entre tareas. Antes de terminar, el proceso debe escribir los valores de a, b y c en la salida estándar.

SOLUCIÓN

```
int n;
pthread_t thid = 0;
int tarea_A(void);
int tarea_B(void);
int tarea_C(int n);
void *thread_B(void *p) {pthread_exit((void *)tarea_B());}
int main(void){
   int ret, a, b, c;
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
   ret = pthread_create(&thid, &attr, &thread_B, NULL);
   if(ret != 0){perror("p_create()"); exit(1);}
   if((a = tarea_A()) > 0){
      c = tarea_C(a);
      ret = pthread_join(thid, (void **)(&b));
      if(ret != 0){perror("p_join()"); exit(1);}
   }else{
      ret = pthread_join(thid, (void **)(&b));
      if(ret != 0){perror("p_join()"); exit(1);}
      c = tarea_C(b);
   }
   pthread_attr_destroy(&attr);
   printf("a=%i b=%i c=%i\n", a, b, c);
   return 0;
}
```

Problema 1.23 (junio 2011)

Dado el siguiente programa:

```
#define N 10
int i, pipe_datos[2], pipe_pid[2];
pid_t pid[N], pid_aux;
void f1(int s){
   if(i < N-1)
      read(pipe_pid[0], &pid_aux, sizeof(pid_t));
   pid_t pid2 = getpid();
   write(pipe_pid[1], &pid2, sizeof(pid_t));
```

```
   if(i == 0)
     f2(0);
   else
     kill(pid[i-1], SIGUSR1);
}
void f2(int s){
   char c1 = '0' + i;
   write(pipe_datos[1], &c1, 1);
   if(i < N-1)
     kill(pid_aux, SIGUSR2);
   exit(0);
}
int main(void){
   struct sigaction a1;
   sigemptyset(&a1.sa_mask);
   a1.sa_flags = 0;
   a1.sa_handler = f1;
   sigaction(SIGUSR1, &a1, NULL); /*A*/
   a1.sa_handler = f2;
   sigaction(SIGUSR2, &a1, NULL); /*B*/
   pipe(pipe_datos);
   pipe(pipe_pid);
   for(i = 0; i < N; i++){
     if((pid[i] = fork()) == 0){
       if(i == N-1) f1(0); /*D*/
       while(1){}
     }
   }
   for(i = 0; i < N; i++){
     char c2;
     waitpid(pid[i], NULL, 0);
     read(pipe_datos[0], &c2, 1); /*C*/
   }
   return 0;
} /*FIN PROGRAMA*/
```

u) ¿Cuántos procesos genera este programa? ¿Qué relaciones padre-hijo hay entre ellos?

v) ¿Qué realizan las sentencias A y B? ¿A cuántos y a qué procesos afectan estas sentencias?

w) ¿Qué procesos escriben en pipe_datos? ¿Qué procesos leen de pipe_datos? ¿Qué procesos escriben en pipe_pid? ¿Qué procesos leen de pipe_pid?

x) Nótese que la sentencia D hace que el último hijo N-1 comience su ejecución invocando a la función f1. ¿Qué procesos envían la señal SIGUSR1? ¿Hacia qué procesos van destinadas? ¿Cuántas señales SIGUSR1 se envían en total?

y) ¿Qué proceso escribe primero en pipe_pid? ¿Qué dato escribe? ¿Qué proceso lee ese dato?

z) Nótese que la variable pid_aux es global. Se asigna en la función f1 y se usa en la función f2. Al terminar de tratar la última señal SIGUSR1, ¿Qué dato contiene la variable pid_aux de cada proceso hijo?

aa) ¿Qué procesos envían la señal SIGUSR2? ¿Hacia qué procesos van destinadas? ¿Cuántas señales SIGUSR2 se envían en total?

ab) ¿Qué proceso envía la primera señal? ¿Y la última señal?

ac) Justo antes de terminar el proceso padre, ¿Qué dato contiene el fichero de pipe_pid?

ad) Qué secuencia de caracteres lee el proceso padre en el bucle final (sentencia C)

SOLUCIÓN

a) ¿Cuántos procesos genera este programa? ¿Qué relaciones padre-hijo hay entre ellos?

Genera N+1 procesos. 1 padre y N hijos (hermanos entre sí).

b)¿Qué realizan las sentencias A y B? ¿A cuántos y a qué procesos afectan estas sentencias?

Arman las señales SIGUSR1 y SIGUSR2 con las funciones de armado f1 y f2, respectivamente. Afectan al proceso padre y a los N procesos hijo (N+1 procesos)

c)¿Qué procesos escriben en pipe_datos? ¿Qué procesos leen de pipe_datos? ¿Qué procesos escriben en pipe_pid? ¿Qué procesos leen de pipe_pid?

Escriben en pipe_datos: Los N procesos hijo

Leen de pipe_datos: El proceso padre

Escriben en pipe_pid: Los N procesos hijo

Leen de pipe_pid: Los procesos hijo 0, 1, 2...N-2

d) Nótese que la sentencia D hace que el último hijo N-1 comience su ejecución invocando a la función f1. ¿Qué procesos envían la señal SIGUSR1? ¿Hacia qué procesos van destinadas? ¿Cuántas señales SIGUSR1 se envían en total?

SIGUSR1 es enviada por los procesos hijo 1, 2, ... N-1. Cada hijo i envia SIGUSR1 a su hermano mayor i-1. En total se envían N-1 señales SIGUSR1.

e) ¿Qué proceso escribe primero en pipe_pid? ¿Qué dato escribe? ¿Qué proceso lee ese dato?

El primer proceso en escribir en pipe_pid es el hijo N-1. Escribe su PID. Ese dato lo lee el proceso hijo N-2.

f) Nótese que la variable pid_aux es global. Se asigna en la función f1 y se usa en la función f2. Al terminar de tratar la última señal SIGUSR1, ¿Qué dato contiene la variable pid_aux de cada proceso hijo?

Contiene el PID del hermano menor inmediato (proceso i+1)

g) ¿Qué procesos envían la señal SIGUSR2? ¿Hacia qué procesos van destinadas? ¿Cuántas señales SIGUSR2 se envían en total?

SIGUSR2 es enviada por los procesos hijo 0, 1, 2,.. N-2. Cada hijo i envía SIGUSR2 a su hermano menor i+1. En total se envían N-1 señales SIGUSR2.

h) ¿Qué proceso envía la primera señal? ¿Y la última señal?

Primera señal (SIGUSR1): Proceso hijo N-1.

Última señal (SIGUSR2): Proceso hijo N-2.

i) Justo antes de terminar el proceso padre, ¿Qué dato contiene el fichero de pipe_pid?

El pid del primer proceso hijo (hijo 0)

j) Qué secuencia de caracteres lee el proceso padre en el bucle final (sentencia C)

0123456789

Problema 1.24 (julio 2011)

Dado el siguiente programa:

```
#define N 10
#define T 8
```

```
int f[2];
char buf;
int i;
pid_t p[N];
void f2(int s){
   for(i = 0; i < N; i++){
      kill(p[i], SIGUSR1);
   }
   for(i = 0; i < N; i++){/*E*/
      while(read(f[0],&buf,1) < 1){
      }
   }
}
void f1(int s){
   buf = '0' + i;
   write(f[1], &buf, sizeof(buf));/*F*/
}
int main(void){
   struct sigaction a1;
   sigemptyset(&a1.sa_mask);
   a1.sa_handler = f2;
   a1.sa_flags = 0;
   sigaction(SIGALRM, &a1, NULL);/*A*/

   sigemptyset(&a1.sa_mask);
   a1.sa_handler = f1;
   a1.sa_flags = 0;
   sigaction(SIGUSR1, &a1, NULL);/*B*/

   if(pipe(f) < 0){
     perror("pipe"); return 1;
   }
   /*C*/
   alarm(T); /*D*/
   for(i = 0; i < N; i++){
      if((p[i] = fork()) == -1){
         perror("fork"); return 1;
      }else if(p[i] == 0){
         close(f[0]);
         pause();
         return 0;
      }
   }
   close(f[1]);
   pause();
   return 0;
}
```

a) ¿Qué hacen la sentencias A y B?

b) ¿Qué contiene la variable f en el punto C del código?

c) ¿Qué acción realiza la sentencia D? ¿Qué consecuencias produce?

d) Dibujar el árbol de procesos generados por el programa

e) Qué señales recibe el proceso padre? ¿Qué señales recibe cada proceso hijo? (Indicar cuantas señales en cada caso)

f) ¿Cuántos bytes se leen del pipe f en cada iteración del bucle for de la sentencia E?

g) ¿Cuántos bytes lee cada proceso hijo del pipe? ¿Cuántos escribe?

h) Nótese que el proceso padre no ejecuta ningún servicio wait. ¿Podría algún hijo quedar huérfano antes de ejecutar la sentencia F? ¿Porqué?

i) Nótese que el programa puede tener un resultado inesperado si el sistema está muy cargado y T es pequeño. Explicar qué problema puede aparecer.

j) Proponer una solución para el problema de la pregunta anterior, para cualquier valor de T (no se admite aumentar T) y manteniendo la sentencia D (alarm(T)) en su ubicación actual. No es necesario escribir el código fuente, pero indicar qué servicios del sistema operativo son necesarios.

SOLUCIÓN

a) ¿Qué hacen la sentencias A y B?

A: Arma la señal SIGALRM con la función de armado f2

B: Arma la señal SIGUSR1 con la función de armado f1

b) ¿Qué contiene la variable f en el punto C del código?

Contiene los descriptores de lectura/escritura del fichero pipe recién creado. f[0] = fd de lectura, f[1] = fd de escritura.

c) ¿Qué acción realiza la sentencia D? ¿Qué consecuencias produce?

Inicia un temporizador de T segundos. Cuando termine, el sistema operativo enviará al proceso principal la señal SIGALRM.

d) Dibujar el árbol de procesos generados por el programa

e) Qué señales recibe el proceso padre? ¿Qué señales recibe cada proceso hijo? (Indicar cuantas señales en cada caso)

El padre recibe 1 SIGALRM.

Cada hijo recibe 1 SIGUSR1.

f) ¿Cuántos bytes se leen del pipe f en cada iteración del bucle for de la sentencia E?

1

g) ¿Cuántos bytes lee cada proceso hijo del pipe? ¿Cuántos escribe?

Lee 0 bytes. Escribe 1 byte.

h) Nótese que el proceso padre no ejecuta ningún servicio wait. ¿Podría algún hijo quedar huérfano antes de ejecutar la sentencia F? ¿Porqué?

No. El proceso padre antes de morir tiene que haber leído n bytes del pipe, que deben haber sido escritos por los hijos (ejecutando F).

i) Nótese que el programa puede tener un resultado inesperado si el sistema está muy cargado y T es pequeño. Explicar qué problema puede aparecer.

No se crearían todos los hijos. SIGALRM llegaría al padre antes de que todos los hijos hayan sido creados. Se ejecutaría la función de armado f2 pero el envío de SIGUSR1 sería incorrecto. El pipe contendría menos de los N bytes esperados, por lo que el padre quedaría esperando indefinidamente en read(f[0]...), etc.

j) Proponer una solución para el problema de la pregunta anterior, para cualquier valor de T (no se admite aumentar T) y manteniendo la sentencia D (alarm(T)) en su ubicación actual. No es necesario escribir el código fuente, pero indicar qué servicios del sistema operativo son necesarios.

Hay que garantizar que SIGALRM llega después de haber creado todos los hijos. Para ello:

- Enmascarar SIGALRM (servicio sigprocmask) antes de invocar 'alarm(T)'
- Desenmascarar SIGALRM (servicio sigprocmask) después del bucle for de creación de hijos.

Problema 1.25 (nov-2011)

```
1  #include <error.h>
```

```
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/types.h>
7  #include <sys/wait.h>
8
9  pid_t hpid, ppid;
10 int i; int n; int estado;
11
12 int main (int argc, char **argv)
13 {
14
15  n = atoi (argv[1]);
16  ppid = getpid ();
17  fprintf (stdout, "Proceso padre: %d Proceso P:  %d\n", (int) getppid (),
18    (int) ppid);
19
20  for (i = 1; i < n; i++) {
21      hpid = fork ();
22      if (hpid == -1) {
23   perror ("P: fork"); exit (1);
24    }
25      if (hpid == 0) {
26   fprintf (stdout, "Hijo: %d pid: %d Padre: %d\n", i, (int) getpid (),
27    (int) getppid ());
28   }
29      else break;
30  }
31
32  if (hpid != 0) {
33      hpid = wait (&estado);
34      if (hpid == -1)
35  {
36   perror ("wait"); return 1;
37  }
38      else if (ppid != getpid ())
39  {
40   fprintf (stdout, "Finalizado el hijo: %d\n", (int) getpid ());
41   return 0;
42  }
43      else fprintf (stdout, "Finalizado el proceso P: %d\n", (int) getpid ());
44  }
45  else if (i == n)
46    fprintf (stdout, "Finalizado el ultimo hijo: %d\n", (int) getpid ());
47
48  return 0;
49 }
```

Suponiendo que el proceso padre P tiene PID 100, que los procesos hijos incrementan en una unidad el PID del padre según se vayan creando y que durante la ejecución de P no se crean más procesos en el sistema:

a) *Obtenga el diagrama jerárquico de procesos creados indicando los PID de los procesos cuando se invoca la ejecución del proceso de la siguiente forma:* `./P 3`
b) *Indique una posible traza de ejecución obtenida con la invocación anterior.*
c) *Si se elimina el servicio* `wait` *de la línea 33, ¿cómo afecta esta moficación a la terminación de los procesos?*
d) *Sobre la versión inicial del código adjuntado en el enunciado, añada el código necesario para bloquear la recepción de la señal SIGINT en todos los procesos salvo en el proceso P, indicando dónde lo incluiría.*
e) *Sobre la versión inicial del código adjuntado en el enunciado, añada el código necesario para que los hijos ignoren la recepción de la señal SIGSTOP, indicando dónde lo incluiría.*
f) *Sobre la versión inicial del código adjuntado en el enunciado, añada el código necesario para que los hijos creen cada uno un thread de tipo ULT que ejecute el código asociado a la función* `void *terminar(void *p)` *en modo detached:*

```
void *terminar(void *p) {
  printf("Thread %u \n", (int)pthread_self());
  exit(0);
}
```

g) Tomando como referencia el código desarrollado para el apartado anterior, indique el resultado de una posible traza de ejecución al invocar: `./P 2`

SOLUCIÓN

a) P: 100 -> 101 -> 102

b)

Proceso padre: 50 Proceso P: 100
Hijo: 1 pid: 101 Padre: 100
Hijo: 2 pid: 102 Padre: 101
Finalizado el ultimo hijo: 102
Finalizado el hijo: 101
Finalizado el proceso P: 100

c) Los procesos que son padres no esperan por sus hijos. La traza de ejecución se modificará al perder el control sobre el orden de finalización de los procesos. Esto afecta al orden en el que se imprimen los mensajes. Probablemente, aparecerán en el mismo orden en el que se han creado los procesos. Respecto al estado en el que quedan los hijos, al no esperar sus padres por ellos, se convierten en zombies.

d) Bastaría con declarar una variable local a la función main que recogiera el valor de la máscara que se quiere definir y añadir a continuación de la línea 25 (primera línea de código ejecutada por los hijos) las siguientes líneas de código:

```
sigset_t mascara; // Variable local de la función main
25      if (hpid == 0) {
  sigemptyset(&mascara);
  sigaddset(&mascara,SIGINT);
  sigprocmask(SIG_SETMASK, &mascara, NULL);
26   fprintf (stdout, "Hijo: %d pid: %d Padre: %d\n", i, (int) getpid (),
27    (int) getppid ());
28   }
```

e) No se puede hacer porque las señales SIGKILL y SIGSTOP son las dos únicas que no se pueden ignorar.

f) Declaración de variables locales de la función `main`:

```
pthread_attr_t attr;
pthread_t thid;
```

Antes del bucle de creación de los hijos, definición de los atributos de los threads:

```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr,PTHREAD_SCOPE_PROCESS);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
```

A continuación de la sentencia de la línea 26 (fprintf), se añade la creación de los threads:

```
if (pthread_create(&thid,&attr,&terminar,NULL) == -1) {
perror("pthread_create"); exit(1);
}
```

g) La traza de ejecución sería:

Proceso padre: 50 Proceso P: 100
Hijo: 1 pid: 101 Padre: 100
Thread: 21678761
Finalizado el proceso P: 100

Al invocar el exit desde la función terminar, no solo se termina el thread sino el proceso desde el cual se ha invocado, así como todos los threads que se hubieran creado dentro de ese proceso, por eso no se imprime el mensaje de finalización del proceso hijo.

Problema 1.26 (abril 2012)

Sea un proceso padre con un número indeterminado de hijos (al menos 1). Escribir un tramo de código fuente que permita al proceso padre esperar la terminación del proceso hijo con PID `p1` *y todos los que terminen antes. Si el hijo* `p1` *ha terminado involuntariamente por la llegada de la señal SIGKILL el proceso padre debe esperar la terminación del resto de hijos durante un tiempo máximo de 5 segundos y después terminar voluntariamente con código de terminación 2.*

NOTA: No usar ninguna variable `N`*, o similar, para indicar el número total de hijos.*

SOLUCIÓN

```
void f1(int s){exit(2);}
main(...){
   ...
   struct sigaction act;
   act.sa_handler = &f1;
   act.sa_flags = 0;
   sigaction(SIGALRM, &act, NULL);
   ...
   while (wait(&status)!= p1)
      continue;
   if(WIFSIGNALED(status))
      if(WTERMSIG(status) == SIGKILL){
         alarm(5);
         while(wait(&status) > 0)
            continue;
         exit(2);
      }
}
```

Problema 1.27 (abril 2012)

Escribir un tramo de código que realice las mismas operaciones que el código que se muestra a continuación, pero maximizando la concurrencia entre las funciones invocadas y generando un único proceso.

```
   int a, b, c, d, e;
   a = tarea_A(0);
   b = tarea_B(0);
   d = tarea_D(a);
   c = tarea_C(a);
   e = tarea_E(b+c);
   printf("%i %i %i %i %i\n", a, b, c, d, e);
```

SOLUCIÓN

```
pthread_t Id_B = 0, Id_D = 0;
int a, b, c, d, e;
void *B(void *p){
   b = tarea_B(0);
   pthread_exit(NULL);
}
void *D(void *p){
   d = tarea_D(a);
   pthread_exit(NULL);
}
int main(void){
   pthread_attr_t attr;
```

```
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);

   pthread_create(&Id_B, &attr, &B, NULL);
   a = tarea_A(0);
   pthread_create(&Id_D, &attr, &D, NULL);
   c = tarea_C(a);
   pthread_join(Id_B, NULL);
   e = tarea_E(b+c);
   pthread_join(Id_D, NULL);
   pthread_attr_destroy(&attr);
   printf("%i %i %i %i %i\n", a, b, c, d, e);
}
```

OTRA SOLUCIÓN:

```
pthread_t Id_CE = 0, Id_B = 0;
int a, b, c, d, e;
void *B(void *p){
   b = tarea_B(0);
   pthread_exit(NULL);
}
void *CE(void *p){
   c = tarea_C(a);
   pthread_join(Id_B, NULL);
   e = tarea_E(b+c);
   pthread_exit(NULL);
}
int main(void){
   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
   pthread_create(&Id_B, &attr, &B, NULL);
   a = tarea_A(0);
   pthread_create(&Id_CE, &attr, &CE, NULL);
   d = tarea_D(a);
   pthread_join(Id_CE, NULL);
   pthread_attr_destroy(&attr);
   printf("%i %i %i %i %i\n", a, b, c, d, e);
   return 0;
}
```

Problema 1.28 (abril 2012)

Se desea escribir un programa servidor de peticiones externas que van llegando indefinidamente en el tiempo. El programa debe generar un proceso padre con N procesos hijo. Las peticiones van llegando por la entrada estándar del proceso padre. Cada petición está codificada mediante un byte único que se debe procesar invocando a la función ya programada: `void procesa_peticion(char c)`*, siendo* `c` *el byte que identifica la petición. Los procesos (padre e hijos) se deben turnar para leer y procesar la siguiente petición, de manera que cada proceso solo debe atender una petición cada vez que le toca el turno. Se debe maximizar la concurrencia (cada proceso debe pasar el turno de lectura al siguiente proceso lo antes posible).*

a) Escribir el programa para que los turnos tengan un orden predeterminado que pase por los N+1 procesos y que se repita indefinidamente. Sugerencia: Usar señales para pasar el turno al siguiente proceso. Suponer que durante la ejecución de una función de armado de una señal A el SO enmascara automáticamente la misma señal A.

b) Escribir el programa sin un orden predeterminado de los turnos.

c) Explicar ventajas e inconvenientes de ambas soluciones.

SOLUCIÓN

a) Turnos en anillo Padre -> HijoN -> Hijo N-1... -> Hijo 1 -> Padre... El turno se pasa mediante la señal SIGUSR1. La invocación a procesa_peticion() debe estar dentro de la función de armado para protegerla de la siguiente llegada de SIGUSR1.

```
pid_t p1;
void procesa_peticion(char c);
void f1(int IDsegnal){
   char b;
   read(0, &b, 1); //LEE PETICIÓN
   kill(p1, SIGUSR1); //PASA EL TURNO
   procesa_peticion(b); // PROCESA PETICIÓN
}
int main(void){
   pid_t p;
   int i;
   struct sigaction act;
   act.sa_handler = &f1;
   act.sa_flags = 0;
   sigaction(SIGUSR1, &act, NULL);

   p1 = getpid();
   for(i=0; i<N; i++){
      switch(p = fork()){
      case -1:
         perror("fork() = -1"); exit(1);
      case 0: //HIJO:
         while(1) pause();
      default: //PADRE:
         p1 = p;
      }
   }
   //PADRE:
   kill(p1, SIGUSR1);
   while(1) pause();
}
```

b) Cada proceso simplemente ejecuta un bucle infinito de leer y procesar petición. Así, puede haber varios procesos compitiendo por leer del mismo fichero (la entrada estándar, heredada por todos los hijos). El SO entrega cada byte a solo 1 proceso. El orden de turno no está predefinido.

```
int main(void){
   char b;
   pid_t p;
   int i;
   for(i=0; i<N; i++)
      if((p = fork()) == -1){
         perror("fork() = -1"); exit(1);
      }else if(p == 0) //HIJO:
         while(1){
            read(0, &b, 1);
            procesa_peticion(b);
         }
   //PADRE:
   while(1){
      read(0, &b, 1);
      procesa_peticion(b);
   }
}
```

c)

Programa con turnos predefinidos:

> **Ventaja**: Orden conocido permitiría implementar funciones adicionales al servidor. Por ejemplo, cada petición podría ser procesada con diferente algoritmo según su orden de llegada.
>
> **Inconveniente**: Si uno de los procesamientos de petición tarda mucho, la señal puede dar una vuelta completa al anillo y el servicio de peticiones se detendría hasta terminar tal procesamiento, habiendo procesos ociosos que podían haberlo atendido.

Programa con turnos no predefinidos:

> **Ventaja**: Más eficiente. Procesos no se entorpecen.

Problema 1.29 (julio 2014)

Se dispone de dos versiones de la función B:

- *int B1(int fd, int init, int size);*
- *int B2(char *name, int init, int size);*

Dicha función recorre secuencialmente el fragmento del fichero que empieza en la posición init y que tiene un tamaño size, generando un entero que entrega como valor de retorno. En caso de error en el acceso al fichero retorna un -1.

Se quiere diseñar un mandato A que reciba como argumento un nombre de fichero y que genere un proceso hijo por cada fragmento de 10 KiB del fichero. Cada hijo ejecutará la función B sobre su fragmento y enviará al padre el entero generado, en formato binario y a través de un pipe. El padre sumará dichos enteros y sacará la suma por su salida estándar.

a) 1 Indicar las ventajas e inconvenientes que presenta cada versión B1 y B2 para su utilización en el mandato A.

b) 1 Dibujar dos esquemas de procesos. El primero deberá indicar la jerarquía de procesos que genera el mandato A y el segundo deberá indicar el esquema de comunicación entre los procesos.

c) 1 Explicar qué ocurre si un hijo muere antes de devolver el valor generado. Explicar qué ocurre si el proceso padre muere antes de que terminen los hijos.

d) 1 Explicar si sería posible que el padre leyese solamente la mitad de los bytes que forman un entero, produciendo, por tanto, un resultado erróneo.

e) 3 Desarrollar el programa A, destacando todos los servicios utilizados.

f) 1 Incluir un mecanismo para evitar que el proceso padre se quede eternamente esperando ante cualquier tipo de problema que pueda tener alguno de los hijos. Escoja la acción que realizará el padre si se da esta situación.

g) 2 El programa A se monta estáticamente y tiene los siguientes tamaños: text: X bytes, data Y bytes, bss Z bytes

Suponga que el tamaño de página es de 4KiB, que están en ejecución el padre y 9 hijos, y que todas las páginas de todos los procesos han migrado y residen en memoria principal. Desarrolle en función de X, Y y Z la expresión que calcule el total de la memoria principal ocupada por los 10 procesos.

Nota. Suponga que dispone de las funciones siguientes:

- *Redondeo al más próximo: int RedProxo(float m);*
- *Redondeo por exceso: int RedEx(float m);*
- *Redondeo por defecto: int RedDef(float m);*

Solución

a) Lo primero que hay que plantearse es que vamos a tener n procesos hijos coutilizando el fichero. Cada uno tiene que acceder a una parte específica del fichero, por lo que necesita su propio puntero. Se necesitarán tantas aperturas de fichero como hijos. Si se utiliza la función B2, la apertura la hará dicha función. Si utilizamos la función B1, cada hijo deberá abrir el fichero antes de llamar a la función B1. No sirve que el padre abra el fichero y que lo hereden los hijos, puesto que, en ese caso, se compartiría el puntero dando resultados erróneos.

Por tanto, no hay una gran diferencia entre usar una función u otra, salvo el tener que incluir en el código de los hijos la apertura del fichero.

b) Se generarán RedEx(FileSize / 10·1024) hijos. En relación con el esquema de comunicaciones nos podemos plantear un solo pipe para comunicar todos los hijos con el padre o utilizar un pipe por hijo. Es más sencilla de programar y utiliza menos recursos la solución de un solo pipe. En todo caso, es muy importante destacar que los hijos deben cerrar el descriptor de lectura del pipe y que el padre debe cerrar el descriptor de escritura del pipe o de los pipes de comunicación con los hijos.

c) Si el hijo muere antes de devolver el valor generado ocurren tres cosas. UNO: El padre recibe una señal SIGCHLD, que por defecto es ignorada. DOS: Mientras el padre no haga el correspondiente wait, el hijo queda en estado zombie. TRES: si se hacen las cosas bien, es decir, si se cierran los descriptores no utilizados de los pipes, cuando el padre lea del pipe recibirá 0 bits leídos, lo que le indica que el hijo ha muerto, por lo que deberá hacer el correspondiente wait. Para el caso de un solo pipe esto ocurrirá cuando todos los demás hijos hayan completado su trabajo o hayan muerto. Para el caso de pipe por hijo, esto ocurrirá cuando el padre lea de ese pipe.

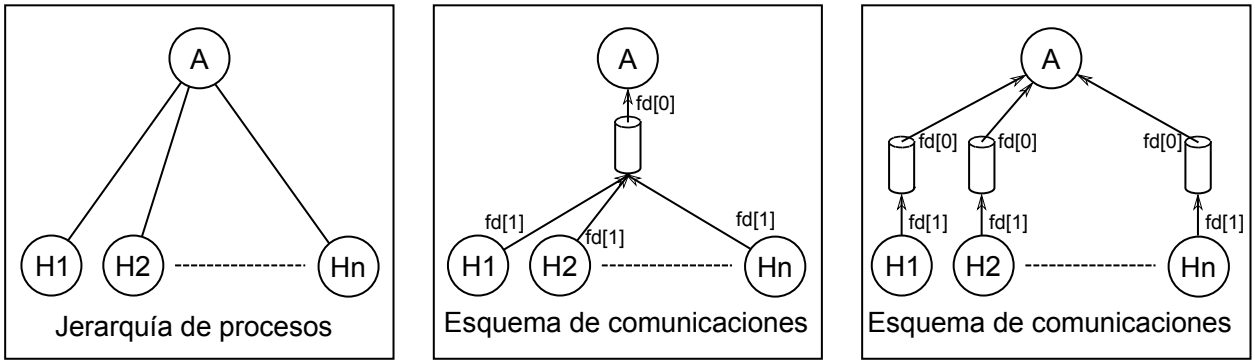


Figura 1.17

Si el padre muere antes que alguno de los hijos ocurren dos cosas. UNO: Los procesos se quedan huérfanos y el proceso INIT los hereda. DOS: si se hacen las cosas bien no habrá lectores en el pipe. Cuando el hijo intente escribir en el pipe recibirá la señal SIGPIPE, por lo que morirá (a menos que la tenga tratada).

d) Dado que lo que se manda por el pipe es un entero en formato binario, ocupará sizeof(int) bytes. Los servicios de escritura y lectura sobre el pipe se harán con ese tamaño (por ejemplo: write(pipe[1], &n, sizeof(n)); read(pipe[0], &n, sizeof(n));. Evidentemente, si hacemos un read o write indicando un tamaño menor de sizeof(int) el padre recibirá menos bytes. En el supuesto de que el mandato esté bien programado **no es posible** que se produzca esta situación puesto que las operaciones de pocos bytes sobre un pipe son **atómicas**, es decir, que se hacen completamente o no se hacen, no se interrumpen mientras se ejecutan.

e)

```
#include <stdio.h>

int main(int argc, char *argv[])  {
struct stat buffer;
int tamfich, i, pp[2], n, size, suma = 0;
   // Obtenemos la información del fichero
   if (stat(argv[1], &buffer)<0)  {
      perror("Error en la llamada stat.\n");
      return 1;
   }
   //Comprobamos si es directorio
   if (S_ISDIR(buffer.st_mode)) {
      perror("El nombre corresponde a un directorio.\n");
      return 1;
   }
   tamfich = (int) buffer.st_size; // Obtenemos tamaño del fichero
   size = 10 * 1024; //Tamaño partición
   pipe(pp);
   for (i = 0; i < tamfich; i+= size) {
      switch(fork())  {
      case -1: // Error
         perror("fork()");
```

```
            return 1;
         case 0: // Hijo
            close(pp[0]);
            if (i + size > tamfich) size = tamfich - i;
            if (size != 0) n = B2(argv[1], i, size);
            write(pp[1], &n, sizeof(n));
            return 0;
         default:   // Padre
         }
      }
      close(pp[1]);
      for (i = 0; i < tamfich; i+= size) {
         read(pp[0], &n, sizeof(n));
         suma += n;
         wait(NULL);
      }
      close(pp[0]);
      write(STDOUT_FILENO, &suma, sizeof(suma));
      return 0;
}
```

f) La forma de evitar que el proceso padre se quede indefinidamente en el último bucle es poniendo un temporizador. La acción lógica sería sacar un mensaje de error y matar a todos los procesos. Para matar a los hijos se pueden hacer dos cosas o bien salvar los `pid` de los hijos y hacer un bucle con `kill` en la función de la alarma o bien hacer un kill al grupo de procesos. En este último caso, si no queremos que el padre muera este tendría que ignorar la señal enviada en el `kill`, por ejemplo con `signal(SIGQUIT, SIG_IGN);`.

Abría que añadir el siguiente código en el main:

```
//Incluir en las declaraciones
struct sigaction act;

//Incluir justo antes o después del  close(pp[1]);
//Se prepara para recibir la señal SIGALRM
   act.sa_handler = AccionTempor;
   act.sa_flags = 0;
   sigaction(SIGALRM, &act, NULL);
   alarm(120);   //Se esperan 2 minutos
```

Y añadir la siguiente función

```
void AccionTempor(void) {
   pid_t parent_pid;
   perror ("Error de temporización.\n")
   parent_pid = getpid();
   kill(-parent_pid, SIGQUIT); //Se matan todos los procesos del grupo parent_pid (padre e hijos)
   return 0;
}
```

g) Suponiendo que los datos con valor inicial, los datos sin valor incial y el heap forman una sola región, el tamaño en páginas de la imagen de memoria de cada proceso es el siguiente:

Texto = RedEx(X / 4·1024).
Datos = RedEx((Y+Z+H) / 4·1024), DCVI + DSVI + Heap expresados todos ellos en bytes.
Pila = Pila expresada en páginas.

Ahora bien, el Texto es compartido, por lo tanto, suponiendo que el tamaño del heap y de la pila es el mismo en todos los procesos, la ocupación en memoria es:

$$4\cdot(\text{Texto} + 10\cdot(\text{Datos} + \text{Pila}))\ \text{KiB}$$

De acuerdo a los prototipo de las funciones B no tiene sentido plantear que estas hagan proyección de memoria. En caso de utilizar el mecanismo de proyección de memoria, ésta la debería hacer el padre de todo el fichero y en modo compartido. En esta caso los argumentos de B deberían ser dirección de memoria de comienzo de su zona y tamaño.

2

SISTEMA DE FICHEROS

Problema 2.1 (septiembre 1998)

Dado un sistema con memoria virtual y con preasignación de swap, y con páginas y bloques de 4 KiB. Sea el código adjunto, que pertenece al proceso 345, con UID real = 34.

```
if (fork () == 0)
   execl ("/doc/edit.exe", "/doc/edit.exe", "/doc/pri1.txt", NULL);
else
   wait (NULL);
```

a) *Indicar en detalle las operaciones que realiza el SO al ejecutar el código anterior, suponiendo que no se producen errores en las llamadas.*

b) *Especificar qué estructuras de información se utilizan, indicando las modificaciones que se realizan sobre ellas.*

c) *Suponiendo que el programa edit debe acceder en lectura-escritura al fichero cuyo nombre se suministra como parámetro, indicar qué servicio debe solicitar y justificar si se realizará con éxito.*

Algunas de las estructuras del sistema son las mostradas en la figura 2.1 y en las tablas 2.1 y 2.2.

Nodo-i	Dueño (UID)	Permisos	Tipo	Agrupaciones	Tamaño	SUID, SGID
2	Root (0)	rwx r-x r-x	directorio	7644	4 KiB	
23	Pepa (34)	rwx r-x r-x	directorio	173	4 KiB	
87	Root (0)	rws --x --x	archivo	79752 y 79753	7,5 KiB	Activo SUID
273	Root (0)	rwx r-x r-x	directorio	7635	4 KiB	
324	Tere (622)	rwx --- ---	directorio	23	4 KiB	
753	Juana (62)	rwx --- ---	archivo	4546, 3874 y 22993	9,2 KiB	
823	Root (0)	rw- --- ---	archivo	12764	3 KiB	
876	Pepa (34)	rwx --- ---	directorio	453	4 KiB	

Tabla 2.1 Contenido de algunos nodos-i

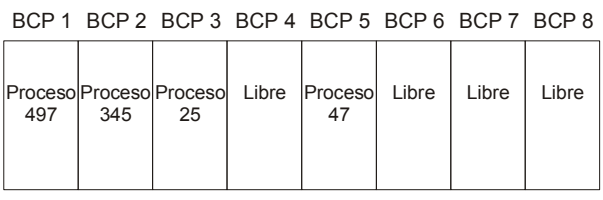


Figura 2.1

Agrupación	Contenido
56	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 .....
443	qwei oier oeiru qoieruower qpoweioru qpwoeir....
453	. 876
	.. 2
453	iue 8234
569	1234567890...r
4413	abcdefghijklme.....
7635	. 273
	.. 2
7635	pri1.txt 823
	edit.exe 87
7644	. 2
	.. 2
	doc 273
	Proa 876
	Prob 23
12764	ieqo ieoqo eir93o3hnf ‘wqn34209e rn349 wheew.....
79752	ABCDEFGHIJKLM.....

Tabla 2.2 Contenido de algunas agrupaciones

Solución PM

Las operaciones que realiza el sistema operativo, suponiendo que no hay errores en las llamadas, son las siguientes:

- 1.- **fork** crea un proceso hijo idéntico al proceso padre. Devuelve al padre el *pid* del hijo y al hijo un cero. Para ello, el S.O. busca una entrada libre en la tabla de procesos y se la asigna al nuevo proceso.

  En este caso, se elige *BCP4*. Sobre dicha entrada se copia la del proceso padre (*pid* 345) situada en *BCP2*. Sobre *BCP4* se escribe el *pid* del hijo. La tabla de ficheros no debe ser modificada porque son compartidos. A continuación, el S.O. crea una tabla de páginas para el proceso hijo. Dicha tabla es una copia idéntica de la del padre, con las páginas de datos y pila marcadas como de *copy-on-write* en la tabla del padre. Cuando ejecuta el hijo, se hace la llamada al sistema exec.

- 2.- **exec** Para ejecutar el mandato **edit.exe**, en primer lugar comprueba si el fichero con el código del mandato tiene permisos de lectura. Para acceder al fichero se parte del directorio raíz “/” con *nodo-i* 2. Se solicita dicho *nodo-i* (que habitualmente está en memoria) y se trae el primer bloque de disco de la agrupación 7644 y se encuentra la entrada al directorio doc. En este caso, como el *UID* real del usuario es 34 (Pepa), se comprueba el acceso del directorio *doc* (*nodo-i* 273) y se ve que tiene permisos para acceder al mismo (*rwxr-xr-x*). Dicha entrada nos remite al *nodo-i* 273 que está en la agrupación 7635. Se trae el primer bloque de dicha agrupación bloques de dicha agrupación hasta que se encuentra la entrada de edit.exe, que nos remite al *nodo-i* 87. Se trae a memoria dicho *nodo-i* y se ve que todo el mundo puede ejecutarlo (*rwx--x--x*) y que además tiene el SUID activo, con lo que Pepa pasa a tener el UID efectivo 0 (superusuario). A continuación se comprueba que se puede acceder al fichero **pri1.txt**. Dicho fichero está en la misma agrupación que **edit.exe** (7635), por lo que se trae a memoria su *nodo-i* y se comprueban los permisos de acceso. En este caso tiene *rwx*------, pero se puede acceder porque Pepa tiene el UID efectivo del *superusuario*. Una vez comprobados los permisos de acceso al fichero, se vacía la tabla de páginas del proceso hijo, se trae al swap el fichero con el código del mandato (preasignación de swap), se actualiza la tabla de páginas y se elimina el flag de COW en la tabla del padre. A continuación empieza la ejecución preasignando una página nueva para datos y pila del nuevo mandato edit.exe. En este momento, ambos procesos solamente comparten la tabla de ficheros. También se carga en memoria los primeros 4 KiB del fichero pri1.txt, que están en la agrupación 12764.

- 3.- **wait** El proceso padre se queda esperando hasta que el hijo termina, evento que le comunica el S.O. con una señal.

Las estructuras de datos utilizadas son:

- Tabla de procesos. Copia de *BCP2* sobre *BCP4* y modificación para introducir los datos del hijo, de su nueva imagen de memoria cuando se hace el exec y de la entrada al fichero pri1.txt cuando se abre.
- Tabla de páginas del proceso padre para activar y desactivar el COW.
- Tabla de páginas del proceso hijo para cambiar las entradas de las páginas por las del nuevo ejecutable edit.exe.
- Mapa de bloques libres del *swap* para asignar las páginas del nuevo proceso.
- *Nodos-i* de los ficheros compartidos para indicar que el número de procesos que los tienen abiertos es 2.

Para que el programa **edit** acceda en lectura-escritura el fichero, debe solicitar el siguiente servicio:

```
open("/doc/pri1.txt",O_RDWR);
```

La solicitud del servicio se realiza con éxito si el fichero existe y si los permisos de acceso son los adecuados. Como ya hemos visto anteriormente, cuando se ejecuta edit.exe se adquiere el UID efectivo del *superusuario* (0), por lo que el directorio "/doc", con permisos *rwxr-xr-x*, es accesible y el fichero *pri1.txt*, con permisos *rw-------*, puede ser accedido con éxito por *Pepa*. Por tanto, el servicio será efectuado sin problemas.

Problema 2.2

Sean los códigos programa 1 y programa 2 que ejecutará el usuario jperez

```
/* Programa 1 */
int main (void) {
int fd; int pid; int cont;

fd = open ("/alfa", O_RDWR);
cont = write(fd, "Escribo alfa", strlen("Escribo alfa"));
close(fd);
fd = open ("/beta/b4/tyz", O_RDWR);
lseek(fd, 13,SEEK_SET);
cont = write(fd, "del ejemplo", strlen("del ejemplo"));
pid = fork();
/* PUNTO A */
switch (pid) {
   case -1:
      break;
   case 0: /* hijo */
      cont = write(fd, "del hijo", strlen("del hijo"));
      /* PUNTO C */
      break;
   default: /* padre */
      cont = write(fd, "del padre", strlen("del padre"));
   /* PUNTO D */
   }
}
```

```
/* Programa 2 */
int main (void) {
int fd; int cont;
fd = open ("/beta/b4/tyz", O_RDWR);
cont = write(fd, "Proceso 2 escribe", strlen("Proceso 2 escribe"));
/* PUNTO B */
close(fd);
}
```

Suponiendo que el orden de ejecución es el siguiente:

- *Proceso padre con código programa 1 ejecuta hasta el punto A*
- *El proceso con código programa 2 ejecuta hasta el punto B*
- *El proceso hijo con código programa 1 ejecuta hasta el punto C*
- *Vuelve a ejecutar el padre hasta el punto D*

Y que los contenidos de algunas agrupaciones y nodos-i son los mostrados en las tablas adjuntas, se pide:

a) *Establecer el esquema de directorios.*
b) *Indicar el tamaño de la agrupación*
c) *Indicar, para cada uno de los procesos, los valores devueltos por cada uno de los servicios del SO que ejecuta.*
d) *Indicar el contenido del fichero /alfa en el punto A y el contenido del fichero /beta/b4/tyz en los puntos A, B y C.*

Agrupación	***Contenido***
32	*. 2; .. 2; usr 121; alfa 243; bi 453; beta 54*
73	*. 417; .. 453 .....*
74	*. 628; .. 54; abc 742; tyz 375*
112	*. 412; .. 453 .....*
138	*. 453; .. 2; bi 430; apli1 412; apli2 417*
255	*¿Cómo se aplica la sección 3.3 del plan de seguridad .....*
271	*. 121; .. 2; prof1 301; prof2 317; prof3 371*
1300	*. 301; .. 121 .....*
2342	*CALLE SANDOVAL, 3 .....*
3207	*. 430; .. 453 .....*
3765	*Desde el punto de vista del usuario, la tabla Panel ....*
4731	*El usuario jramirez está .....*
6734	*. 54; .. 2: b1 621; b2 612; b3 614; b4 628*
6777	*En el esquema correspondiente se representan estas líneas .....*
15301	*Para el ejemplo de prueba se han propuesto 5 casos que ....*

Nodo-i	***Dueño (UID)***	***Permisos***	***Tipo***	***Agrupaciones***	***Tamaño***	***SUID, SGID***
2	*Root (0)*	*rwx r-x r-x*	*directorio*	*32*	*180*	
54	*Root (0)*	*rwx r-x r-x*	*directorio*	*6734*	*180*	
121	*Root (0)*	*rwx r-x r-x*	*directorio*	*271*	*150*	
243	*Root (0)*	*rwx --- ---*	*usuario*	*4731*	*653*	
301	*jperez*	*rwx r-x r-x*	*directorio*	*1300*	*300*	
317	*lgomez*	*rw- r-- r--*	*usuario*	*255 y 257*	*1568*	
371	*rfung*	*rw- r-- r--*	*usuario*	*15301 a 15305*	*4354*	
375	*jperez*	*rw- r-- r--*	*usuario*	*6777 a 6779*	*2564*	
412	*Root (0)*	*rwx r-x r-x*	*directorio*	*112*	*120*	
417	*Root (0)*	*rwx r-x r-x*	*directorio*	*73*	*180*	
430	*Root (0)*	*rwx r-x r-x*	*directorio*	*3207*	*330*	
453	*Root (0)*	*rwx r-x r-x*	*directorio*	*138*	*150*	
612	*lgomez*	*rw- r-- r--*	*usuario*	*3765*	*473*	
614	*lgomez*	*rw- r-- r--*	*usuario*	*4321 y 486*	*1324*	
621	*lgomez*	*rw- r-- r--*	*usuario*	*2342*	*875*	
628	*jperez*	*rwx r-x r-x*	*directorio*	*74*	*120*	
742	*jperez*	*rw- r-- r--*	*usuario*	*5701 a 5708*	*7432*	

Solución

a) El directorio raíz tiene "." y ".." apuntando a sí mismo. El esquema de directorio es el siguiente:

```
/  |Usr    |prof1
         |prof2
         |prof3
   |alfa
```

```
|bi      |bi
         |apli1
         |apli2
|beta    |b1
         |b2
         |b3
         |b4      |abc
                  |tyz
```

b) El tamaño de la agrupación ha de ser un múltiplo del tamaño del sector, es decir nxTsector, donde n es generalmente una potencia de dos. Por otro lado, el sector tiene m bytes, siendo m una potencia de dos. Por tanto, el tamaño de la agrupación ha de ser una potencia de dos.

Viendo el número de agrupaciones que tienen los distintos ficheros, y comparando este número con el correspondiente tamaño del fichero, se deduce fácilmente que el tamaño de la agrupación es de 1 KiB. Por ejemplo, el fichero de nodo_i 317 tiene dos agrupaciones y ocupa 1.568 B. Dos agrupaciones de 1 KiB permiten tamaños de fichero entre 1.025 y 2.048 B.

c) Los valores devueltos son los que se indican seguidamente. Son de destacar los siguientes extremos:

- El usuario jperez no puede abrir el fichero alfa, por lo que los tres primeros servicios devuelven error.
- El valor devuelto al padre por el fork puede ser un -1 en caso de que el SO no pueda crear al hijo.
- Los blancos también son caracteres que ocupan su byte.
- Los valores devueltos por el servicio open dependen de la ocupación de la tabla de fd. Se ha supuesto que solamente están abiertas la entrada y salidas estándar. El open puede devolver un -1 en caso de que esté llena la tabla de fd. Sin embargo, los write no `pueden devolver error, puesto que se sobreescribe en espacio ya asignado (solamente se podría dar error en caso de avería del controlador o del disco).

```
        int main (void) {
          int fd; int pid; int cont;

-1 error  fd = open ("/alfa", O_RDWR);
-1        cont = write(fd, "Escribo alfa", strlen("Escribo alfa"));
-1        close(fd);
3         fd = open ("/beta/b4/tyz", O_RDWR);
13        lseek(fd, 13,SEEK_SET);
11        cont = write(fd, "del ejemplo", strlen("del ejemplo"));
0 y n     pid = fork();
          /* PUNTO A */
          switch (pid) {
            case -1:
              break;
            case 0: /* hijo */
8             cont = write(fd, "del hijo", strlen("del hijo"));
              /* PUNTO C */
              break;
            default: /* padre */
9             cont = write(fd, "del padre", strlen("del padre"));
            /* PUNTO D */
            }
          }

        int main (void) {
          int fd; int cont;
3         fd = open ("/beta/b4/tyz", O_RDWR);
17        cont = write(fd, "Proceso 2 escribe", strlen("Proceso 2 escribe"));
          /* PUNTO B */
0         close(fd);
          }
```

d) El fichero alfa no cambia, puesto que no se puede abrir.

El fichero tyz sufre las siguientes modificaciones:

Inicialmente:	En el esquema correspondiente se representan estas líneas....
Punto A:	En el esquemadel ejemploiente se representan estas líneas ....

Punto B: Proceso 2 escribeejemploiente se representan estas líneas …
Punto C: Proceso 2 escribeejemplodel hijo representan estas líneas …
Punto D: Proceso 2 escribeejemplodel hijodel padretan estas líneas ...

Problema 2.3 (septiembre 2000)

Dado un disco de 4 GiBytes con tamaño de bloque de 1 KiB se quieren analizar los dos siguientes sistemas de ficheros:

1.- Sistema de ficheros tipo UNIX con las siguientes características:

- Representación del fichero mediante nodos-i con 10 direcciones directas a bloque, un indirecto simple, un indirecto doble y un indirecto triple y direcciones de bloque de 4 bytes. El sistema utiliza un mapa de bits para la gestión del espacio vacío.

2.- Sistema de ficheros tipo MS-DOS (FAT) con las siguientes características:

- Entradas de 4 bytes y tamaño de agrupaciones de 4 bloques.

Se pide:

a) ¿Cuál es el tamaño máximo de los ficheros en cada sistema de ficheros?
b) ¿Qué tamaño ocupan la FAT y el mapa de bits en cada caso?
c) Se desea abrir un fichero llamado datos.txt que se encuentra en el directorio user y acceder a los bytes 273.780.000 y 281.450.500. Si nos encontramos en el directorio raíz, ¿cuál será el número de accesos al disco para realizar la anterior operación en cada sistema de ficheros?
d) ¿Dónde se almacenan los atributos del fichero en cada sistema de ficheros?. ¿Qué problemas puede presentar este sistema de atributos en MS-DOS?

Solución

a) UNIX:

10 bloques directos a bloque
1024/4 bloques con indirección simple
(1024/4)^2 bloques con indirección doble
(1024/4)^3 bloques con indirección triple

Tamaño máximo de fichero:

- (10 + 256 + 256^2 + 256^3) bloques * 1024 Bytes/bloque = 16 GiB
- La máxima longitud de fichero que podría ser alcanzada con este esquema es de aproximadamente 16 GiBytes.

MS-DOS: Usando la FAT descrita en el enunciado el tamaño máximo de fichero viene determinado por el tamaño de disco, 4 GiB, ya que la FAT es simplemente una lista enlazada y el direccionamiento especificado es suficiente para apuntar a todas las agrupaciones.

b) MS-DOS: Tenemos un disco con 4 GiB con 1 KiB de tamaño de bloque y 4 KiB de tamaño de agrupación. De esta forma en la FAT se necesita 1 Mega entradas. Cada entrada de la FAT apunta a una agrupación y tiene un tamaño de 4 B, por lo que el tamaño total de la FAT es de 4 MiB.

UNIX: El mapa de bits incluye 1 bit para cada recurso existe, es decir, un bit por cada bloque de disco o nodo-i. En total se tienen que controlar 4 Mega bloques por lo que son necesarios 512 bloques para el mapa de bits, es decir 512 KiB.

Nota: hay que tener en cuenta que en MS-DOS se ha direccionado a nivel de agrupación y en UNIX a nivel de bloque.

c) MS-DOS: Supondremos el fichero esté muy disperso en la FAT, de forma que cada acceso a ella exige un acceso al disco. Además, consideraremos que el sistema mantiene el valor de la agrupación accedida y su posición de puntero, de forma que accesos sucesivos no requieran volver a recorrer toda la FAT. Para poder acceder al fichero se necesitan los siguientes accesos a disco:

- 1 lectura del directorio raíz para localizar el directorio user

- Supondremos que el directorio user cabe en un bloque. Hará falta un acceso a la FAT para conocer la ubicación del directorio y 1 lectura del bloque que ocupa el directorio user para localizar el fichero datos.txt, así como su primera agrupación (agrupación 0).
- Una vez localizado el fichero son necesarios los siguientes accesos:
- Los bytes a leer se encuentran en las agrupaciones 66.840 y 68.713. Por tanto, se necesitan 68.713 accesos a la FAT para localizar las agrupaciones más 2 accesos para leer los datos pedidos de disco.
- En total 68.718 accesos.

UNIX: Para acceder al fichero:

- Traer el bloque / a memoria y buscar la entrada /user
- Traer el nodo-i de /user a memoria
- Traer un bloque de /user a memoria y buscar la entrada de datos.txt
- Traer el nodo-i de datos.txt a memoria
- Por tanto para acceder al fichero necesitamos 4 accesos a memoria.
- Para leer los datos hay que acceder a los bloques 32500 * 4 = 130000 y 32750 * 4 = 131000
- Para direccionar estos bloques necesitamos los punteros de triple indirección del nodo-i (con los punteros de doble indirección se llega al bloque: 10+256+256*256= 65802). Accesos requeridos:
- 3 para acceder a los 3 niveles de indirección
- 1 para acceder al bloque donde se encuentra el primer byte
- 3 para acceder a los 3 niveles de indirección
- 1 para acceder al bloque donde se encuentra el segundo byte
- Total 12 accesos.

d) En MS-DOS los atributos se guardan en la entrada del directorio, mientras que en UNIX se almacenan en el nodo-i.

Los problemas que se presentan en MS-DOS son que al modificar los atributos hay que acceder a la información de los directorios y la complicación que supone el crear enlaces a archivos.

Problema 2.4 (junio 2001)

Dado el siguiente código:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>

6  #define BUFFSIZE 27

7  int main (int argc, char *argv[]) {
8     int miFd, miFd2, miFd3, pid;
9     char buff[BUFFSIZE]="abcdefghijklmnopqrstuvwxyz";
10    char buff2[10], buff3[10];
11    int pos1, pos2, pos3;

12    if (argc != 3) {
13       fprintf(stderr,"Error al introducir argumentos");
14       exit(1);  }

15    miFd= open(argv[1],O_CREAT|O_TRUNC|O_RDWR, 0700);

16    pid = fork();
17    switch (pid) {
18       case -1: perror("fork");
19          exit(2);
20       case 0:     //proceso hijo
21          write(miFd,buff,BUFFSIZE);
22          link(argv[1],argv[2]);
23          miFd2=open(argv[2],O_RDWR);
```

```
24          miFd3=dup(miFd);
25          pos1=lseek(miFd,4,SEEK_SET);
26          read(miFd2,buff2,10);
27          read(miFd3,buff3,10);
28          pos2=lseek(miFd2,0,SEEK_END);
29          write(miFd2,buff2,10);
30          write(miFd3,buff3,10);
31          pos3=lseek(miFd3,0,SEEK_CUR);
32          close(miFd);
33          close(miFd2);
34          close(miFd3);
35          return 0;
36       default:
37          write(miFd,buff,BUFFSIZE);
38          close(miFd);
39          return 0;
40    }
41 }
```

Resolver las siguientes cuestiones:

a) Explicar el código, dibujando la relación existente entre los distintos descriptores y ficheros para los procesos involucrados. Dibujar los nodos-i de los ficheros con los campos más significativos.

b) ¿En qué consisten los argumentos que recibe el programa principal? Teniendo en cuenta las llamadas al sistema que se realizan en el código, ¿qué restricciones deben cumplir dichos argumentos para que la ejecución sea correcta?

c) ¿Cuál es el contenido final de los ficheros involucrados? ¿Depende del orden de ejecución de los procesos padre e hijo? Si es así, indicar el contenido de los ficheros suponiendo que el proceso padre se ejecuta de forma completa antes que el proceso hijo.

d) ¿Cuáles son los valores que reciben las variables `pos1`, `pos2` y `pos3` en las líneas 25, 28 y 31? Suponer que el proceso padre se ha ejecutado de forma completa antes de que comience a ejecutar el proceso hijo.

e) Dibujar un diagrama indicando cómo se modifican las entradas de directorios y nodos-i correspondientes cuando se lleva a cabo la llamada al sistema link() (línea 22).

f) ¿Sería equivalente sustituir la línea 15 por:

```
int miFd=creat(argv[1],0700); ?
```

Si no lo es, explicar cómo se ve modificada la ejecución del programa.

g) Utilizar un mecanismo de sincronización para lograr que el proceso padre realice todas las operaciones antes de que el proceso hijo realice cualquier operación de escritura.

Solución

a) El programa principal recibe dos argumentos, correspondientes a nombres de ficheros. Lo primero que se hace en el código es comprobar que el número de argumentos es correcto (líneas 12 a 14). A continuación, se crea el fichero pasado como primer argumento, abriéndolo en modo lectura/escritura (línea 15). El proceso crea un proceso hijo (llamada al sistema `fork()`). Se comprueba que la llamada ha sido correcta. El código que ejecuta el hijo se encuentra entre las líneas 21 y 25. El código del padre es el código situado entre las líneas 37 y 39. Padre e hijo comparten el descriptor de fichero `fd`.

El proceso padre escribe en el fichero el buffer `buff`, que contiene una cadena de caracteres con las letras del abecedario. A continuación, cierra el descriptor correspondiente y finaliza.

Por su parte, el proceso hijo también escribe en dicho fichero el buffer `buff`. A continuación crea un enlace físico al primer fichero, usando como nombre el segundo argumento (línea 22). Por tanto, ambos nombres referencian al mismo fichero. Se abre el fichero correspondiente al segundo fichero, asociandole el descriptor `miFd2`. En la línea 24, se duplica el descriptor `miFd` y el descriptor duplicado se recoge en la variable `miFd3`. A continuación, se posiciona el fichero en el cuarto byte, utilizando el descriptor `miFd`. Los descriptores `miFd` y `miFd3` comparten puntero, frente al descriptor `miFd2`, que no lo comparte. Se leen 10 bytes del fichero a través del descriptor `miFd2` y se almacena en `buff2`. A continuación, se leen otros 10 bytes del fichero a través del descriptor `miFd3` y se almacena en `buff3`. Se posiciona el fichero al final del mismo, utilizando el descriptor `miFd2`. Posteriormente, se

escriben 10 bytes situados en `buff2` en el fichero a través del descriptor `miFd2` y otros 10 bytes situados en `buff3` a través del descriptor `miFd3`. En la línea 31 el puntero no se ve modificado, ya que se posiciona en el lugar actual. A continuación se cierran todos los descriptores y el proceso hijo finaliza.

Sólo hay un nodo-i involucrado, ya que se trata de un enlace físico. La representación gráfica de las entradas de directorio correspondientes y el nodo-i se encuentra en la figura 2.2, suponiendo que se ha ejecutado el programa con los argumentos `texto.txt` y `texto2.txt`:

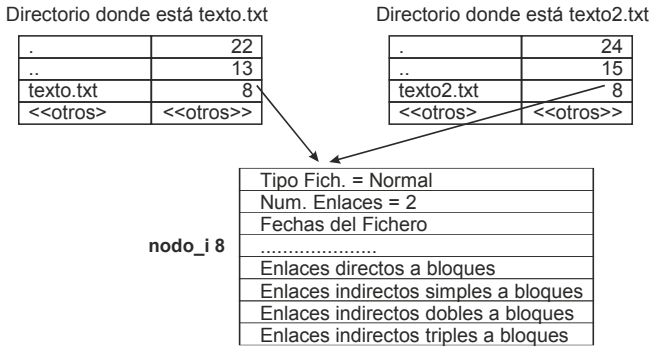


Figura 2.2

Suponemos que el fichero no tenía ningún enlace inicialmente.

b) Los argumentos que recibe el programa principal son los dos nombres del fichero. Las restricciones que deben cumplir ambos argumentos vienen determinadas por las llamadas al sistema `open()` y `link()`: ambos argumentos deben corresponder a nombres de ficheros normales (no directorios existentes) y además ambos ficheros deben situarse en un mismo volumen. Por otro lado, se deben tener los permisos adecuados sobre el directorio en el cuál se van a crear. Además, si ya existe un fichero con el nombre correspondiente al primer argumento, éste debe tener permisos de lectura/escritura. No debe existir ya un fichero con un nombre igual al segundo argumento.

c) El contenido de ambos ficheros es el mismo, porque de hecho se trata del mismo fichero, pero con dos nombres diferentes. El contenido del mismo depende del orden de ejecución de los procesos padre e hijo. Si el proceso padre se ejecuta de forma completa antes que el hijo, el contenido final del fichero será igual a:

"abcdefghijklmnefghijklmnyz'/0'abcdefghijklmnopqrstuvwxyz'/0'abcdefghij" (Los caracteres '/0' corresponden a caracteres nulos, debidos a las escrituras de una cadena de caracteres)

d) Teniendo en cuenta que los descriptores miFd y miFd3 comparten puntero, mientras que el descriptor miFd2 no lo comparte, los valores que reciben las variables pos1, pos2 y pos3 son:

```
pos1 = 4
pos2 = 54
pos3 = 24
```

e) La situación antes de la llamada al sistema `link()` (suponiendo que el fichero no tuviera más de un enlace) se encuentra en al figura 2.3:

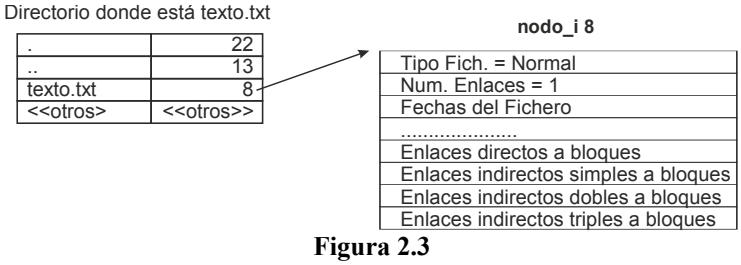


Figura 2.3

La situación después de realizar la llamada al sistema `link()` se encuentra en la figura 2.2.

f) No es equivalente. La llamada `creat(argv[1],0700)` equivaldría a: `open(argv[1],O_CREAT|O_TRUNC|O_WRONLY,0700)`. Si se utilizara la llamada `creat()` no se podría leer del fichero correspondiente y las llamada `read()` de la línea 27 daría un error, ya que `miFd` y `miFd3` comparten modos de acceso (llamada `dup`); no así la llamada `read()` de la línea 26, ya que en este caso el fichero se ha abierto en modo lectura/escritura.

g) Podemos utilizar cualquier mecanismo de sincronización para lograr que el proceso padre realice todas las operaciones antes de que el proceso hijo realice cualquier operación de escritura. Si utilizamos un pipe, la solución sería la siguiente:

```
   int pp[2];    char testigo;
   pipe(pp);
   ........
20      case 0: /* proceso hijo */
21         read(pp[0],&testigo,1);
22         close(pp[0]);
23         close(pp[1]);
24    /* resto proceso hijo */
   ..............
36      default: /* proceso padre */
37         write(miFd,buff,BUFFSIZE);
38         close(miFd);
39         write(pp[1],&testigo,1);
40         close(pp[0]);
41         close(pp[1]);
42         exit(0);
43    }
44 }
```

Falta realizar el tratamiento de error correspondiente a las llamadas al sistema.

Problema 2.5 (septiembre 2001)

Bloque	*Agrupación*	*CONTENIDO*
0		*BOOT*
1		*SUPERBLOQUE*
2		*001011110000111000……..00000*
3		*011100100100101011……..00000*
4		
5		*NODOS-I DEL SISTEMA*
6		
7		
8		
....		
600		
601	*0*	*AGRUPACIONES DE DATOS Y DIRECTORIOS*
602	*1*	
603	*2*	
604	*3*	
605	*4*	
606	*5*	
607	*6*	
608	*7*	
...	...	
...	...	
...	...	
4095	*3494*	

Tabla 2.3

La tabla 2.3 muestra el esquema de un microsistema de ficheros tipo UNIX, que está simulado sobre memoria. En él se pueden distinguir seis partes: el bloque de boot, el superbloque, el mapa de bits para los nodos-i del sistema, el mapa de bits para las agrupaciones de datos, los nodos-i (uno por cada entrada de la tabla) y finalmente las agrupaciones de datos y directorios. Todos los bloques de disco simulados aparecen numerados y tienen un tamaño de 256 bytes. Nótese que las agrupaciones son de un bloque.

- *Bloque de BOOT: Ocupa el bloque 0. No es utilizado en el problema.*
- *Superbloque: Ocupa el bloque 1. No es utilizado en el problema.*
- *Mapa de bits para los nodos-i del sistema: Ocupa el bloque 2. Cada bit representa el estado de un nodo-i del sistema: 0 significa que el nodo-i está libre y 1 implica que el nodo-i está ocupado. En la figura se ha representado un posible estado del mapa de bits.*
- *Mapa de bits para los bloques de datos: Ocupa los bloques 3 y 4. Cada bit representa el estado de un bloque de datos: 0 significa que el bloque está libre y 1 implica que el bloque está ocupado. En la figura se ha representado un posible estado del mapa de bits.*
- *Nodos-i del sistema: En este área se almacenan los nodos-i del sistema. Este área ocupa 596 bloques (bloques 5 a 600, ambos inclusive).*
- *Agrupaciones de datos y directorios: en este área se almacenan los bloques de datos y directorios. Ocupa 3495 bloques (desde el bloque 601 hasta el 4095, ambos inclusive).*

Responder razonadamente a las siguientes preguntas:

a) Calcular el espacio que puede ocupar un nodo-i en este sistema, considerando que la representación del fichero se realiza mediante nodos-i con 10 direcciones directas a bloque, un indirecto simple, un indirecto doble y un indirecto triple. Comprobar cuantos nodo-i caben en un bloque del sistema.

b) Tamaño máximo de un fichero en este sistema.

c) Número máximo de ficheros del sistema.

d) ¿Por qué los bloques de datos y los directorios aparecen mezclados en un sistema de ficheros tipo UNIX?

e) Número máximo de caracteres que puede tener el nombre de un fichero de directorio en este sistema, suponiendo que cada entrada de directorio es de tamaño fijo y ocupa un bloque.

f) Rellenar la tabla que representa el sistema en el caso de que se tenga la estructura de ficheros de la figura 2.4.

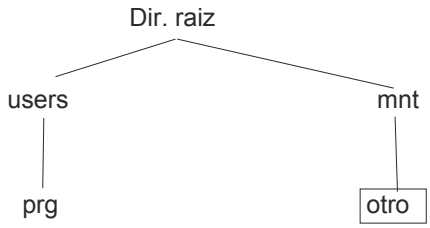


Figura 2.4

donde los ficheros ordinarios aparecen encuadrados. El fichero otro contiene las 10 primeras letras del abecedario. Los nodos-i y los bloques de datos y directorios se pueden rellenar de forma esquemática.

g) ¿Se pueden utilizar ficheros para llevar a cabo la comunicación de procesos? ¿Es aconsejable? ¿Por qué?

Solución

a) Los atributos son los siguientes, para los cuales se ha supuesto un tamaño adecuado para un microsistema de ficheros.

Atributo	**Tamaño en B**	**Comentario**
Identificador único del fichero	2	Permite direcciar 64 Ki ficheros
Tipo de fichero	1	
Dueño y grupo	4	Dueño y grupo de 16 bits.
Información de protección	2	
Tamaño real en bytes	3	Permite ficheros de hasta 16 MiB
Hora y fecha de creación	4	
Hora y fecha del último acceso	4	
Hora y fecha de la última modificación	4	
Número de enlaces (número de nombres)	1	Permite hasta 256 enlaces por fichero

Total de 25

Además, tenemos 13 punteros (10 punteros directos, 1 puntero indirecto simple, 1 puntero indirecto doble y 1 puntero indirecto triple). Hay que calcular lo que ocupa 1 puntero. El mapa de bits sirve para 256 · 8 · 2 = 4.096 agrupaciones, que son más de las 3.495 de las que consta el sistema.

Para direccionar esas 3.495 agrupaciones, necesitamos 12 bits, que permite hasta 4.096 agrupaciones. Sin embargo, dado que el servidor de ficheros que maje este sistema también debería poder soportar implementaciones con más capacidad, supondremos direcciones de 16 bits. Lo que permitiría tener 64 Ki agrupaciones, es decir, un sistema de ficheros con 64 Ki · 256 B = 16 MiB de espacio para datos y directorios.

Un nodo-i ocupa: 2 B ·13 punteros + 25 B = 51 B. Por lo que caben 256 / 51 = 5 nodos-i por bloque.

Como tenemos 596 bloques para nodos-i, podemos tener hasta 596 · 5 = 2.980 ficheros. Esta cantidad parece adecuada ya que tenemos solamente 3.495 agrupaciones.

b) El máximo tamaño del fichero puede venir limitado por varias rozones.

- Dado que el atributo tamaño tiene 3 B, significa que el tamaño máximo del fichero es de 16 MiB.
- El tamaño máximo de un fichero viene determinado por la estructura del nodo-i y según dicha estructura, la expresión para calcularla es la siguiente:

  Tam. Max = Tam(punt. directos) + Tam(punt. indir. simple) + Tam(punt. indir. doble) + Tam(punt. indir. triple) = $(10 + (Sb/n) + (Sb/n)^2 + (Sb/n)^3) \cdot Sb$

  Donde Sb es el tamaño de una agrupación y n es lo que ocupa la dirección de una agrupación, que hemos establecido en el apartado anterior en 2 B.

  De la expresión anterior, se deduce que Tam. Max = 2.113.674 agrupaciones = 541.100.544 B ≈ 516 MiB.
- Hemos dedicado 16 bits para direccionar las agrupaciones, por lo que no podemos tener más de 64 Ki agrupaciones, lo que da una capacidad de 64Ki · 265 = 16 MiB. El fichero sería algo menor, puesto que hay que descontar las agrupaciones necesarias para directorios y para punteros indirectos.
- Finalmente, si las 3.495 agrupaciones disponibles se asignan a un solo fichero este tendría 3.466 agrupaciones (3.495 menos una para el directorio raíz y 28 agrupaciones para los punteros indirectos), es decir, 3.466 · 256 = 887.296 B = 866,5 KiB.

De estos cuatro límites el menor es el debido al espacio físico de almacenamiento, lo que indica que el mayor fichero será de 866,5 KiB.

c) El número máximo de ficheros viene limitado por el número de nodos-i, que se ha indicado anteriormente que es de 2.980 ficheros.

d) Los directorios son ficheros ordinarios que contienen un registro por cada entrada del directorio. Este registro tiene la siguiente información: (nombre, num. nodo-i)

e) Nos indican que cada entrada del directorio ocupa un bloque de 256 bytes. Hay que calcular el número de bytes necesarios para representar un nodo-i. Necesitamos 12 bits. Sin embargo, por las consideraciones que hemos indicado en el apartado a), dedicaremos 2 bytes para representar el nodo-i. Luego, nos restan 254 bytes para representar el nombre.

f) Supongamos que los nodos-i de nuestro sistema son los siguientes (los nodos-i 0 y 1 están ocupados por el sistema) (véase figura 2.5).

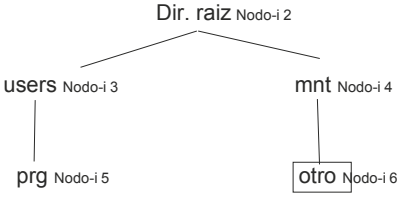


Figura 2.5

Entonces, la tabla de la figura podría quedar como se indica en la tabla 2.4.

Bloque	Agrupación	CONTENIDO
0		BOOT
1		SUPERBLOQUE
2		1111111000000000000……..00000
3		1111111111111000000……..00000
4		
5		NODOS-I DEL SISTEMA
6		
7		
8		
....		
600		
601	0	AGRUPACIONES DE DATOS Y DIRECTORIOS
602	1	
603	2	
604	3	
605	4	
606	5	
607	6	
608	7	
...	...	
...	...	
...	...	
4095	3494	

Tabla 2.4

Nodos-i:

Nodo-i 2: Agrupaciones 0, 1, 2 y 3
Nodo-i 3: Agrupaciones 4, 5 y 6
Nodo-i 4: Agrupaciones 7, 8 y 9
Nodo-i 5: Agrupaciones 10 y 11
Nodo-i 6: Agrupación 12

Agrupaciones de datos y directorios:

0	.	2
1	..	2
2	users	3
3	mnt	4
4	.	3
5	..	2
6	prg	5
7	.	4
8	..	2
9	otro	6
10	.	5
11	..	3
12	ABCDEFGHIJ000000.....0	

g) Sí, un archivo puede utilizarse para comunicar procesos. Por ejemplo, un proceso puede escribir datos en un archivo y otro puede leerlos.

Es un mecanismo simple que, sin embargo, presenta una serie de inconvenientes que hacen que en general no sea un mecanismo de comunicación ampliamente utilizado, a saber:

- Es un mecanismo bastante ineficiente, ya que la escritura y lectura en disco es lenta.
- Necesitan algún otro mecanismo que permita que los procesos se sincronicen en el acceso a los datos almacenados en un archivo.

Problema 2.6 (septiembre 2002)

La siguiente figura muestra el esquema de un pequeño sistema de ficheros tipo UNIX. En él se pueden distinguir seis partes: el bloque de boot, el superbloque, el mapa de bits para los nodos-i del sistema, el *mapa de bits para los bloques de datos*, los *nodos-i (uno por cada entrada de la tabla) y finalmente los bloques de datos y directorios*. Todos los bloques de disco aparecen numerados y tienen un tamaño de 4 KiB.

0	*BOOT*
1	*SUPERBLOQUE*
2	*1110100100000100000000.....0000000000000*
3	*1111110110000001000000.....0000000000000*
4	***Nodo-i 2***
5	
...	*NODOS-I DEL SISTEMA*
258	
259	
260	***Bloque Datos 0***
261	
262	
...	*BLOQUES DE DATOS Y DIRECTORIOS*
2306	
2307	***Bloque Datos 2047***

- *Bloque de BOOT: Ocupa el bloque 0.*
- *Superbloque: Ocupa el bloque 1.*
- *Mapa de bits para los nodos-i del sistema y mapa de bits para los bloques de datos: Ocupan los bloques 2 y 3 respectivamente. Cada bit representa el estado de un nodo-i o bloque de datos del sistema: 0 significa que está libre y 1 implica que está ocupado. En la figura se ha representado el estado actual de ambos mapas de bits. El mapa de bits de nodos-i comienza por el nodo-i 2 (nodo-i situado en el bloque de disco 4) y el mapa de bits de los bloques de datos comienza por el bloque de datos 0 (bloque de disco 260).*
- *Nodos-i del sistema: En este área se almacenan los nodos-i del sistema. Cada nodo-i ocupa un bloque. Este área ocupa 256 bloques (bloques 4 a 259, ambos inclusive).*
- *Bloques de datos y directorios: en este área se almacenan los bloques de datos y directorios. Ocupa 2048 bloques (desde el bloque 260 hasta el 2307, ambos inclusive).*

Los nodos-i del sistema tienen la siguiente estructura:

Tipo Fichero (Normal o Directorio)
Num. Enlaces
UID
GID
Bits de protección
Num. Bloques Directos
1. Bloque directo
2. Bloque directo
.....
N. Bloque directo
....

Sólo se utilizan bloques directos y además sólo se distingue entre fichero normal y directorio. Se ha simplificado la estructura habitual del nodo-i, eliminando algunos atributos. El nodo-i 2 es el nodo-i del directorio raíz. El nodo-i 0 queda reservado para indicar que la correspondiente entrada de directorio no está ocupada. El nodo-i 1 está reservado por el sistema.

Los nodos-i del sistema quedan representados en la siguiente figura:

Nodo-i 2	Nodo-i 3	...	Nodo-i 6	Nodo-i 7	...	Nodo-i 9	Nodo-i 10	...	Nodo-i 15
Directorio 4	Directorio 3		Directorio 3	Directorio 3		Normal 1	Directorio 2		Normal 1

0 1 777 1 0 #### .... ####	0 1 777 1 1 #### .... ####	0 1 755 1 2 #### .... ####	101 30 755 1 3 #### .... ####	101 30 640 2 5 6 #### ..... ####	101 30 755 1 4 #### .... ####	101 30 640 1 5 #### .... ####

El contenido de los bloques de datos se representa en la siguiente figura:

Bloque 0	"."	2	;	".."	2	;	"tmp"	3	;	"home"	6	;			
Bloque 1	"."	3	;	".."	2	;	"fich.txt"	9	;	"proyecto"	10	;			
Bloque 2	"."	6	;	".."	2	;	"pepe"	7	;						
Bloque 3	"."	7	;	".."	6	;	"fich2.txt"	9	;						
Bloque 4	"."	10	;	".."	3	;	"fich3.txt"	15	;						
Bloque 5	456700.....			.....			.....			.....			.....		000
Bloque 6	00000.....			.....			.....			.....			.....		000
Bloque 7	00000.....			.....			.....			.....			.....		000
Bloque 8	00000.....			.....			.....			.....			.....		000
....															

Responder razonadamente a las siguientes preguntas:

a) ¿Es el sistema de ficheros consistente? Si no es así, indicar qué problemas tiene y modificar las estructuras de datos mostradas en el problema para que el sistema sea consistente. Indicar cuáles son los pasos que tendría que seguir una herramienta tipo fsck para buscar las inconsistencias y corregirlas. Describir las estructuras de datos adicionales que podría utilizar para llevar a cabo dicho proceso.

b) Indicar el tamaño máximo de un fichero de este sistema.

c) ¿Existe algún problema respecto a la protección de los ficheros en el escenario descrito? Si es así, indicar la solución.

d) Rellenar la metainformación del sistema que queda modificada después de la ejecución de cada una de las llamadas al sistema que se enumeran a continuación por parte del usuario con uid 101 y gid 30. Representar en forma de árbol la estructura de directorios resultante en cada uno de los pasos e indicar si las llamadas al sistema se realizan de forma correcta.

```
1. link("/tmp/fich.txt", "/home/fich4.txt");
2. unlink("/tmp/fich.txt");
3. mkdir("/tmp/datos", 0755);
```

Las llamadas se llevan a cabo en el orden descrito.

e) Se desea abrir el fichero fich2.txt y acceder al byte 5000. Si nos encontramos en el directorio raíz, ¿cuál será el número de accesos a disco para realizar dicha operación en este sistema de ficheros?

Solución

a) El sistema de ficheros no es consistente. Para deducirlo, se han comprobado los siguientes aspectos:

1.- Nodos-i utilizados en el sistema

- Según el mapa de bits de nodos-i ⇒ Nodos-i utilizados: 2, 3, 4, 6, 9, 15
- Según el recuento de nodos-i ocupados en el sistema (Recorriendo los bloques de directorios) ⇒ Nodos-i utilizados: 2, 3, 6, 7, 9, 10, 15

El estado es inconsistente. La solución consiste en marcar como libre el nodo-i 4 y como ocupados los nodos-i 7 y 10 en el mapa de bits de nodos-i:

Mapa de bits de nodos-i: 1100110110001000.....

2.- Bloques de datos utilizados en el sistema

- Según el mapa de bits de bloques de datos ⇒ Bloques utilizados: 0, 1, 2, 3, 4, 5, 7, 8, 15
- Según el recuento de bloques ocupados en el sistema (Recorriendo los nodos-i que los referencian) ⇒ Bloques de datos utilizados: 0, 1, 2, 3, 4, 5 (2 veces), 6

El estado es inconsistente. Los errores afectan al mapa de bits de bloques de datos y a los ficheros “fich.txt” y “fich3.txt”. La solución al mapa de bits consiste en marcar como libre los bloques 7, 8 y 15, quedando del siguiente modo: 1111111000000.....

Para resolver el problema del bloque compartido por los ficheros, la mejor solución sería copiar el bloque 5 a otro bloque nuevo, por ejemplo el bloque 7 y asignar este bloque a uno de los dos ficheros, por ejemplo “fich3.txt”.

El mapa de bits de bloques de datos quedaría: 1111111100000.....

Además, el nodo-i 15 deberá referenciar al bloque 7 en lugar del bloque 5.

3.- Número de enlaces

- Según el nodo-i 9: Número de enlaces=1. Según las referencias a dicho nodo-i: Número de enlaces=2
- Según el nodo-i 7: Número de enlaces=3. Según las referencias a dicho nodo-i: Número de enlaces=2

El estado es inconsistente. La solución consiste en incrementar el campo número de enlaces correspondiente al nodo-i 9 y decrementar el campo número de enlaces correspondiente al nodo-i 7.

La herramienta para la comprobación de la consistencia tendría que seguir los pasos utilizados anteriormente. Las estructuras de datos que podría utilizar serían:

- Un contador para cada nodo-i, que se debe incrementar a partir de los nodos-i referenciados al recorrer el árbol de directorios. Esta estructura nos permite comprobar la consistencia del mapa de bits de nodos-i (contador>0) y número de enlaces del mismo (contador=campo nº de enlaces).
- Un contador por cada bloque de datos, que debe incrementarse a partir del recuento de los mismos.

Otras comprobaciones adicionales son:

- Existencia de los UID y GID utilizados.
- Que todos los números de nodo-i estén dentro del rango (2-257)

b) El tamaño máximo de un fichero viene determinado por la estructura del nodo-i y por el tamaño del área de datos.

Según la estructura del nodo-i, la expresión para calcula el tamaño máximo es la siguiente:

Tam. Max = Num. Punteros directos · Sb, donde Sb es el tamaño de un bloque (4 KiB).

Se necesita conocer cuántos punteros directos puedo almacenar en el nodo-i. Dado el tamaño del área de datos, bastaría con 11 bits para direccionar los 2048 bloques de datos.

Al ser un sistema de ficheros pequeño, sería razonable utilizar 16 bits para los campos de cada nodo-i. Por tanto, un nodo-i tiene 2048 campos, por lo que permite direccionar: 2048-6=2042 bloques de datos. Luego, Tam. Max = 2042 · 4 KiB = 8168 KiB

Los bloques que se pueden dedicar a datos son: 2048-1 (directorio raíz) =2047. Por tanto, el tamaño máximo limitado por el número de bloques del sistema permite almacenar el fichero calculado previamente.

c) El nodo raíz no debería tener permisos 777, que implica permisos de lectura, escritura y ejecución para todos los usuarios del sistema. La solución sería modificarle los permisos, estableciéndolos a 755.

Otro problema de seguridad proviene de la no-existencia de un campo que establezca el tamaño real del fichero. Un usuario puede crear un fichero y escribir un byte y posteriormente leer todo el bloque con el contenido que dejó otro usuario previamente.

d) Se supone que se han corregido los problemas de consistencia del sistema de ficheros y en concreto, que el número de enlaces del nodo-i 9 vale 2.

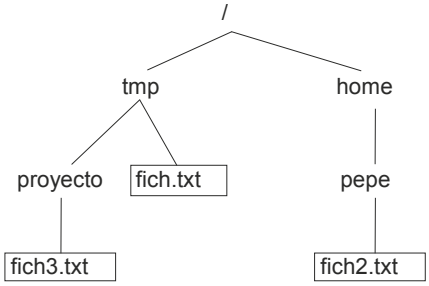


Figura 2.6

El árbol de directorios inicial es el de la figura 2.6.

1.- Error. El usuario con uid 101 y gid 30 no tiene permisos para crear un fichero en el directorio *home*. La estructura de árbol no se modifica.

2.- La llamada se realiza de forma correcta. La operación modifica el bloque de datos 1 y el nodo-i 9 de la siguiente forma:

Tipo	Num. Enlaces	UID	GID	Bits de protec.	Bloques Directos	Bloque directo 1	Bloque directo 2	Bloque directo 3	....
Normal	1	101	30	640	2	5	6	####	

Se decrementa el número de enlaces del nodo-i. Como antes valía 2, ahora pasa a tomar el valor 1

Bloque 1	"." 3 ;	".." 2 ;	"proyecto" 10 ;

Se elimina la entrada correspondiente a "fich.txt".

Esta operación no afecta a los mapas de bits, dado que no se eliminan ni bloque ni nodos-i.

El árbol de directorios resultante es el de la figura 2.7.

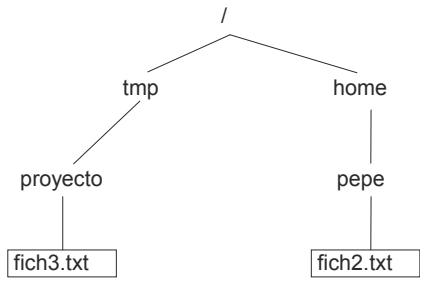


Figura 2.7

3.- La llamada se realiza de forma correcta. Se va a crear un nuevo directorio en tmp, por lo que se habilita una nueva entrada en el bloque 1. Además, necesita un nuevo nodo-i libre, por ejemplo, el 4:

Bloque 1	"." 3 ;	".." 2 ;	"proyecto" 10 ;	"datos" 4

El nodo-i 4 tendrá la siguiente estructura:

Tipo	Num. Enlaces	UID	GID	Bits de protec.	Bloques Directos	Bloque directo 1	Bloque directo 2	Bloque directo 3	.....
Normal	2	101	30	750	1	7	####	####	

Se le asigna un bloque libre, por ejemplo, el 7

El bloque de datos 7 contendrá:

Bloque 7	"." 4 ;	".." 3 ;

Además, el nodo-i 3 también es modificado al incrementarse el número de enlaces:

Tipo	Num. Enlaces	UID	GID	Bits de protec.	Bloques Directos	Bloque directo 1	Bloque directo 2	Bloque directo 3	.....
Directorio	4	0	1	777	1	1	####	####	

Se incrementa en 1 el número de enlaces

Respecto a la metainformación, en el mapa de nodos-i se marca el nodo-i 4 como ocupado y en el mapa de bloques de datos se marca el bloque 7 como ocupado.

El árbol de directorios resultante es el de la figura 2.8.

e) Se supone que el nodo-i 2 (raíz) está ya en memoria.

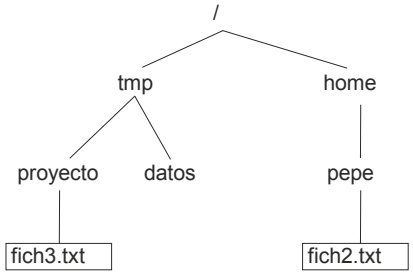


Figura 2.8

El número de accesos a disco para acceder al fichero:

- Acceso 1: Traer el bloque / a memoria y buscar la entrada /home
- Acceso 2: Traer el nodo-i de /home a memoria
- Acceso 3: Traer el bloque de /home a memoria y buscar la entrada de /home/pepe
- Acceso 4: Traer el nodo-i de /home/pepe a memoria
- Acceso 5: Traer el bloque de /home/pepe a memoria y buscar la entrada de /home/pepe/fich2.txt
- Acceso 6: Traer el nodo-i de /home/pepe/fich2.txt

Por tanto para acceder al fichero necesitamos 6 accesos a disco.

- Acceso 7: Para leer el byte 5000 del fichero hay que acceder al bloque 6.

Total: 7 accesos.

Problema 2.7 (junio 2003)

a) Implementar el programa `GenerarClave` *con los siguientes requisitos:*

- *El programa recibe como único parámetro el nombre de un fichero o directorio.*
- *En caso de haber recibido un directorio el programa emite un –1 por su salida estándar y finaliza.*
- *Si el parámetro recibido es un fichero el programa deberá emitir por su salida estándar un número entero calculado de la siguiente manera: tamaño en bytes del fichero más la suma de todos los bytes del fichero.*

b) Se desea realizar el programa `InsertarClave` *con los siguientes requisitos:*

- *El programa recibe como argumentos el nombre de un directorio y el nombre de un fichero. Se debe suponer que tanto el directorio como el fichero existen y son válidos.*
- *Calcula la clave del fichero indicado como segundo argumento, usando para su cálculo el programa* `GenerarClave` *realizado en el anterior apartado.*
- *Una vez calculada la clave, crea en el directorio especificado como primer argumento un enlace simbólico con nombre el valor de la clave calculada, apuntando al fichero recibido como entrada.*

Por ejemplo, en la siguiente invocación del programa:

InsertarClave /var/claves/ /tmp/fichDatos.txt

si la clave calculada es 67534, se debería crear la siguiente entrada en el directorio `/var/claves/`**:**

lrwxr-xr-x 1 luis luis 13 Jun 2 16:36 67534 -> /tmp/fichDatos.txt

c) Realizar el programa `ComprobarClaves` *con los siguientes requisitos:*

- *Recibe como parámetro de entrada el nombre de un directorio. Se debe suponer que este directorio solamente contiene entradas creadas por el programa* `InsertarClave`**.**
- *Cada 20 segundos, de forma predeterminada, recorre uno por uno todos los ficheros del directorio, comprobando que su clave es correcta.*
- *En caso de encontrar una clave incorrecta, imprime por su salida estándar el nombre del fichero correspondiente.*

d) Responda razonadamente. ¿Habría alguna diferencia si en vez de enlaces simbólicos el programa `InsertarClave` *creara enlaces físicos?, ¿qué ventajas e inconvenientes podría suponer esta situación?*

Solución

a) El programa propuesto es el siguiente.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   struct stat buffer;
   int clave=0;
   FILE *file;
   int s;

   /* Obtenemos la información del fichero */
   if (stat(argv[1], &buffer)<0)
   {
      perror("Error en la llamada stat");
      return 1;
   }

   /* Comprobamos si es directorio */
   if (S_ISDIR(buffer.st_mode))
   {
      clave = -1;
      printf("%d\n", clave);
      return 0;
   }
   else
   {
      /* Obtenemos tamaño del fichero */
      clave = (int) buffer.st_size;
      file = fopen(argv[1],"r");
      if (!file)
      {
         perror("Error al abrir el fichero");
         return 1;
      }

      /* Cálculo de la suma de los bytes del fichero */
      s = getc(file);
      while (s!=-1)
      {
         clave += s;
         s = getc(file);
      }
   }

   /* Imprimimos por la salida estándar la clave */
   printf("%d\n", clave);
   return 0;
}
```

b) Debemos recoger la salida estándar del programa anterior, por lo que se debe crear un hijo que ejecute dicho programa y recoger su salida a través de una tubería.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
```

```
{
   int fd[2];
   FILE *file;
   pid_t pid;
   int n;
   int clave;
   char nuevoPath[200];
   char origPath[200];
   char claveStr[100];

   /* Creamos la tubería */
   if (pipe(fd)<0)
   {
      perror("error en la tuberia");
      return 1;
   }

   /* Creamos el hijo */
   pid = fork();

   switch(pid)
   {
      case -1:
         perror("Error en el fork");
         return 1;
      case 0:
         /* El hijo ejecuta GenerarClave */
         close(fd[0]);
         close(1);
         dup(fd[1]);
         close(fd[1]);
         execlp("GenerarClave", "GenerarClave", argv[2], NULL);
         perror("Error en el exec de GenerarClave");
         return 1;
         break;
      default:
         /* El padre lee del pipe */
         close(fd[1]);
         file = fdopen(fd[0], "r");
         fscanf(file, "%d", &clave);

         /* Ruta original */
         strcpy(origPath,"");
         if (argv[2][0]!='/')
         {
            getcwd(origPath, 200);
            strcat(origPath, "/");
         }
         /* Ahora es ruta absoluta */
         strcat(origPath, argv[2]);

         /* Nueva ruta */
         sprintf(claveStr,"/%d", clave);
         strcpy(nuevoPath,"");
         strcat(nuevoPath, argv[1]);
         strcat(nuevoPath, claveStr);

         /* Creación del enlace simbólico */
         symlink(origPath,nuevoPath);
   }
   return 0;
}
```

c) El programa propuesto es el siguiente.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <dirent.h>

int main(int argc, char *argv[])
{
   int fd[2];
   pid_t pid;
   int n;
   int clave;
   char dirActual[200];
   char origPath[200];
   char claveStr[100];
   DIR *dirp;
   FILE *file;
   struct dirent *dp;

   dirp = opendir(argv[1]);

   if (dirp == NULL)
   {
      perror("No puedo abrir el directorio");
      exit(1);
   }
   else
   {
      /* Nos saltamos las dos primeras entradas: "." y ".." */
      readdir(dirp);
      readdir(dirp);

      /* Leemos el directorio entrada por entrada */
      while ((dp = readdir(dirp)) != NULL)
      {
         printf("Nombre fich: %s\n",dp->d_name);

         /* Se crea la tubería */
         if (pipe(fd)<0)
         {
            perror("error en la tuberia");
            return 1;
         }

         /* Ruta original */
         sprintf(origPath,"%s%s%s",argv[1],"/",dp->d_name);

         pid = fork();

         switch(pid)
         {
            case -1:
               perror("Error en el fork");
               return 1;
            case 0:
               /* El hijo ejecuta GenerarClave */
               close(fd[0]);
               close(1);
               dup(fd[1]);
               close(fd[1]);
               execlp("GenerarClave", "GenerarClave", origPath, NULL);
```

```
            perror("Error en el exec de GenerarClave");
            return 1;
            break;
         default:
            /* El padre lee del pipe y comprueba que la clave es correcta*/
            close(fd[1]);
            file = fdopen(fd[0], "r");
            fscanf(file, "%d", &clave);
            sprintf(claveStr,"%d", clave);

            if (strcmp(claveStr,dp->d_name))
               fprintf(stderr,"Error en la clave de %s\n",dp->d_name);
      }
   }
  }
  return 0;
}
```

d) La primera diferencia es que no se pueden crear enlaces físicos entre diferentes volúmenes. La segunda diferencia vendría a la hora del borrado de los ficheros. Dependiendo de la política que nos interese podemos utilizar cada uno de los enlaces. Si usamos enlaces simbólicos podemos tener enlaces erróneos en nuestro directorio. Si usamos enlaces físicos podemos estar generando problemas con el borrado de los ficheros, ya que estamos aumentando el contador del número de enlaces en los mismos.

Desde el punto de vista de la programación, sólo necesitamos cambiar la llamada symlink por link.

Problema 2.8

a) Dado el siguiente código, en el cual no se ha realizado ninguna comprobación de errores:

```
1  int main(int argc, char *argv[]) {
2     int fd, fd2, fd3;
3     pid_t pid1, pid2;
4     char buffer[1024]="12345678901234567890123456789012345";
5
6     fd=open(argv[1],O_RDWR);
7     pid1=fork();
8     if (pid1==0) {
9        fd2 = creat(argv[2], 0640);
10       fd3 = dup(fd);
11       read(fd3,buffer,10);
12       read(fd,buffer,10);
13       write(fd2,buffer,20);
14       close(fd);
15       close(fd2);
16       close(fd3);
17       return 0;
18    }
19    else {
20       pid2=fork();
21       if (pid2==0) {
22          fd2=open(argv[1],O_RDONLY);
23          fd3=open(argv[2],O_RDWR);
24          write(fd3, buffer, 10);
25          read(fd,buffer,10);
26          write(fd,buffer+10,10);
27          close(fd);
28          close(fd2);
29          close(fd3);
30          return 0;
31       }
```

```
32      else {
33         fd2=creat(argv[2],0755);
34         fd3=dup(fd2);
35         read(fd,buffer,10);
36         write(fd2,buffer,10);
37         write(fd3,buffer,10);
38         close(fd);
39         close(fd2);
40         close(fd3);
41         return 0;
42      }
43   }
44 }
```

Si se ejecuta este programa con los argumentos fich1 fich2 y el fichero fich1 tiene el contenido "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ", responder razonadamente a las siguientes cuestiones:

1. Si el orden en el que se ejecutan los procesos de forma completa es: proceso padre, primer proceso hijo, segundo proceso hijo, ¿cuál es el contenido del fichero fich2 al final de la ejecución?

2. Realizar el mismo ejercicio para el caso de que el orden de ejecución sea el inverso (segundo proceso hijo, primer proceso hijo, proceso padre).

3. Para el orden de ejecución del apartado 1, rellenar la siguiente tabla, con los valores correspondientes de la tabla en la línea ejecutada:

Num. línea	Puntero de posición del fichero fd1	Puntero de posición del fichero fd2	Puntero de posición del fichero fd3	Contenido de buffer

4. Una vez que el proceso correspondiente ejecuta el código de la línea 10, ¿qué modo de acceso tiene sobre el fichero cuyo descriptor es fd3? Responder a la misma cuestión para el caso del proceso que ejecuta el código de la línea 34.

5. El código correspondiente a la línea 26, ¿produciría un error de compilación?, ¿produciría un error de ejecución? Explicar gráficamente en qué consiste esta operación, dibujando el buffer y el fichero cuyo descriptor es fd.

6. ¿Qué diferencias habría en la ejecución del proceso que ejecuta el código de la línea 36, si dicha línea se sustituyera por `write(fd3,buffer,10);`?

7. ¿Qué diferencias habría en la ejecución si en lugar de utilizar procesos pesados utilizaramos procesos ligeros?

b) Escribir en C una pequeña función `int generar_fichero(int offset)`, que realice las siguientes tareas:

1. Abre el fichero que se encuentra en el directorio de trabajo inicial del usuario del proceso, cuyo nombre es el identificador del usuario que está ejecutando el programa y que tiene extension ".dat".

2. Crea un fichero en el directorio actual, con nombre el identificador del proceso y con extensión ".dat".

3. Lee los bytes del primer fichero, a partir del desplazamiento del fichero `offset` y los escribe en el segundo fichero.

4. Cierra ambos ficheros.

5. Devuelve 0 si todo ha ido bien y –1 en caso contrario.

Solución

a) La solución de cada apartado es la siguiente:

a.1. 12345678901234567890

a.2. Si el fichero fich2 no existiera, habría un error en la llamada de la línea 23. No obstante, el contenido del fichero al final de la ejecución quedaría: OPQRSTUVWXOPQRSTUVWX

a.3.

Num. línea	Puntero fd	Puntero fd2	Puntero fd3	Buffer

6	0	#	#	123456789012345678901234 5
33	0	0	#	123456789012345678901234 5
34	0	0	0	123456789012345678901234 5
35	10	0	0	abcdefghij12345678901234 5
36	10	10	10	abcdefghij12345678901234 5
37	10	20	20	abcdefghij12345678901234 5
38	#	20	20	abcdefghij12345678901234 5
39	#	#	20	abcdefghij12345678901234 5
40	#	#	#	abcdefghij12345678901234 5
9	10	0	#	123456789012345678901234 5
10	10	0	10	123456789012345678901234 5
11	20	0	20	klmnopqrst12345678901234 5
12	30	0	30	uvwxyzABCD12345678901234 5
13	30	20	30	uvwxyzABCD12345678901234 5
14	#	20	30	uvwxyzABCD12345678901234 5
15	#	#	30	uvwxyzABCD12345678901234 5
16	#	#	#	uvwxyzABCD12345678901234 5
22	30	0	#	123456789012345678901234 5
23	30	0	0	123456789012345678901234 5
24	30	0	10	123456789012345678901234 5
25	40	0	10	EFGHIJKLMN12345678901234 5
26	50	0	10	EFGHIJKLMN12345678901234 5
27	#	0	10	EFGHIJKLMN12345678901234 5
28	#	#	10	EFGHIJKLMN12345678901234 5
29	#	#	#	EFGHIJKLMN12345678901234 5

a.4. Línea 10: Modo de acceso lectura y escritura

Línea 34: Modo de acceso sólo escritura.

a.5. No, es una línea de código perfectamente válida. La figura 2.9 muestra el resultado de la operación gráficamente,

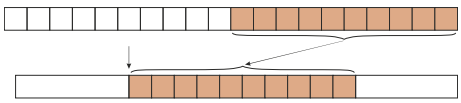


Figura 2.9

a.6. Ninguna, dado que estos dos descriptores comparten el púntero de posición.

a.7. En el caso de utilizar procesos ligeros, se comparte la memoria, por lo que la variable buffer es compartida y todos los cambios sobre esta variable son vistos por dichos procesos ligeros. En este caso, además habría que utilizar mecanismos de sincronización, para evitar que se dieran condiciones de carrera.

b) El programa siguiente muestra una posible solución.

```
#define TAM_FICH 256
#define TAM_BUFFER 1024

int generar_fichero(int offset) {
   char *fichero_origen;
   char *fichero_destino;
   char buffer[TAM_BUFFER];
   int fd_origen, fd_destino;
   int leidos;

   fichero_origen = malloc(TAM_FICH*sizeof(char));
   if (fichero_origen == NULL)
      return -1;

   fichero_destino= malloc(TAM_FICH*sizeof(char));
   if (fichero_destino== NULL)
      return -1;
```

```
   sprintf(fichero_destino, "%d", (int) getpid());
   fichero_destino=strcat(fichero_destino,".dat");
   fd_destino=creat(fichero_destino, 0640);
   if (fd_origen <0)
      return -1;

   chdir(getenv("HOME"));
   sprintf(fichero_origen,"%d", (int) getuid());
   fichero_origen=strcat(fichero_origen,".dat");
   fd_origen=open(fichero_origen, O_RDONLY);
   if (fd_destino <0)
      return -1;

   lseek(fd_origen, offset, SEEK_SET);
   while ((leidos = read(fd_origen, buffer, TAM_BUFFER))>0)
      write(fd_destino, buffer, leidos);

   free(fichero_origen);
   free(fichero_destino);
   close(fd_origen);
   close(fd_destino);

   if (leidos <0)
      return -1;
   else
      return 0;
}
```

Problema 2.9 (mayo 2004)

*En un sistema operativo de tipo **UNIX** tenemos, entre otros muchos, los siguientes ficheros:*

Nodoi	*Permisos*	*Usuario*	*Grupo*	*Tamaño B*	*Nombre_Fichero*
123	*d rwx r-x --x*	*Pedro*	*Datsi*	*1024*	*/home/Pedro*
656	*- r-x r-s ---*	*root*	*root*	*1546*	*/home/Pedro/Cancelar*
678	*- rw- rw- ---*	*Pedro*	*Datsi*	*30*	*/home/Pedro/Informe.txt*
697	*- rw- --- ---*	*Pedro*	*Datsi*	*30*	*/home/Pedro/Informe2.txt*
708	*- r-x r-x ---*	*Pedro*	*Datsi*	*998*	*/home/Pedro/Prog1*
351	*d rwx --- ---*	*Maria*	*Datsi*	*1024*	*/home/Maria*
697	*- rw- --- ---*	*Pedro*	*Datsi*	*30*	*/home/Maria/Informe3.txt*
720	*- r-x r-x ---*	*Maria*	*Datsi*	*345*	*/home/Maria/Prog2*

*Un permiso **s** implica que, además del correspondiente permiso de ejecución, está activo el bit `SETUID` si aparece en el primer trio ó el bit `SETGID` si aparece en el segundo trio.*

*El código fuente de **Prog1** es:*

```
1  int main (int argc, char **argv)
2  {
3     int fd=-1, fd2=-1;
4     pid_t pid;
5     char cadena[512]="122333444455555";
6
7     access("Cancelar", X_OK);
8     chmod("Informe2.txt",0660);
9
10    fd = open ("Informe2.txt", O_RDWR);
11
12    pid=fork();
```

```
13
14    if (pid!=0) {
15       lseek(fd, 12, SEEK_CUR);
16       write(fd2, cadena, 10);
17       write(fd, cadena, 4);
18       read(fd, cadena+2, 5);
19       close(fd2);
20       close(fd);
21    }
22    else {
23       fd2 = dup2(fd, 10);
24       write(fd, "77777", 3);
25       read(fd2, cadena, 5);
26       close(fd);
27       close(fd2);
28    }
29 }
```

*El código fuente de **Prog2** es:*

```
1  int main (int argc, char **argv)
2  {
3     int sd, fd;
4     char cadena[256]="55555555555555555555";
5
6     link("../Pedro/Informe.txt","Informe.txt");
7     sd = open("Informe.txt",O_RDWR);
8     lseek(sd, 0, SEEK_END);
9     fd = open ("Informe3.txt", O_RDWR|O_CREAT, 0666);
10
11    lseek(fd, -10, SEEK_END);
12    write(fd, cadena, 20);
13    lseek(fd, 20, SEEK_END);
14    close(fd);
15 }
```

El fichero descrito por el nodo-i 678 comienza por: `12345123455432154321123455432 1...`

El fichero descrito por el nodo-i 697 comienza por: `12345678901234567890123456789 0...`

En este entorno, los usuarios Pedro y María, que pertenecen al grupo Datsi, ejecutan desde sus respectivos `HOME`*, los programas **Prog1** y **Prog2** respectivamente.*

Suponiendo el siguiente orden de ejecución:

- *Prog1:padre hasta la línea 15 (incluida).*
- *Prog1:hijo hasta la línea 25 (incluida).*
- *Prog1:padre hasta la línea 17 (incluida).*
- *Prog2 hasta la línea 12 (incluida).*
- *Prog1:padre hasta que termina.*
- *Prog1:hijo hasta que termina.*
- *Prog2 hasta que termina.*

Responder a las siguientes preguntas.

a) ¿Qué devuelve la llamada `access` *de la línea 7 de **Prog1**?*
b) ¿Qué devuelve la llamada `write` *de la línea 24 de **Prog1**?*
c) ¿Qué devuelve la llamada `write` *de la línea 16 de **Prog1**?*
d) ¿Qué devuelve la llamada `lseek` *de la línea 8 de **Prog2**?*
*e) ¿Cuál es el tamaño final del fichero **Informe2.txt**?*
f) ¿Cuál es el contenido de las 5 primeras posiciones de la variable `cadena` *después de ejecutar la línea 18 de **Prog1**?*

SOLUCION

a) La llamada access sólo comprueba los UID y GID reales, y no los efectivos por lo que no tiene permisos suficientes y devuelve –1.

b) La llamada write, que tiene permisos, hace la operación y devuelve 3, ya que es el número de bytes que realmente consigue escribir, aunque la cadena sea de mayor tamaño.

c) La llamada write devuelve –1 ya que fd2 tiene un valor de –1 en el proceso padre, que no es un valor válido de descriptor.

d) La llamada `lseek` sitúa el puntero al final del fichero, por lo que el valor devuelto será el tamaño del mismo, es decir 30. Este valor se encuentra especificado en el nodo-i y se ha suministrado en la tabla de descripción de ficheros (hay que recordar que el tamaño físico menor del fichero será un bloque o agrupación, que tendrá un cierto contenido).

e y f) En la tabla 2.5 se analizan los programas Prog1 y Prog2 línea por línea, según el orden de ejecución especificado

Por lo tanto, el tamaño el final de *Informe2.txt* es **40** bytes y las cinco primeras posiciones de la variable cadena en el padre son **12555**.

Prog	***Línea***	***Resultado***
1	*7*	*access devuelve -1*
1	*8*	*Informe2.txt con permisos rw- rw- ---*
1	*10*	*Se abre Informe2.txt y, por tanto, fd=3*
1	*12*	*Creamos proceso hijo. Padre e hijo de Prog1 comparten puntero fd*
1	*15*	*lseek cambia el puntero de posición y devuelve 12*
1	*23*	*fd2=10 y se comparte puntero con fd*
1	*24*	*Se escriben 3 bytes.Informe2.txt = 1234567890127776789012345678901*
1	*25*	*Se leen 5 bytes desde el puntero. Cadena(hijo) = 678903444455555*
1	*16*	*La llamada devuelve –1*
1	*17*	*Se escriben 4 bytes de la variable cadena(padre) en el fichero.Informe2.txt = 1234567890127776789012235678901*
2	*6*	*link devuelve 0 por que se realiza correctamente*
2	*7*	*sd=3*
2	*8*	*lseek devuelve 30*
2	*9*	*fd=4. Se abre Informe3.txt que es un enlace físico a Informe2.txt.El puntero de acceso es independiente al de Prog1*
2	*11*	*lseek devuelve 20 y sitúa al puntero en esa posición*
2	*12*	*Escribimos 20 veces el carácter 5.Informe2.txt = 1234567890127776789055555555555555555555Tamaño Informe2.txt = 40*
1	*18*	*read devuelve un 5. El puntero estaba en la posición 25. Se lee lo que ha escrito Prog2.Cadena(padre) = 125555544455555*
1	*19*	*close devuelve –1, ya que fd2 era –1*
1	*20*	*Se cierra fd en el padre*
1	*26*	*Se cierra fd en el hijo*
2	*13*	*lseek devuelve 60, pero no cambia el tamaño del fichero*
2	*14*	*Se cierra fd en el Prog2*

Tabla 2.5

Problema 2.10 (febrero 2005)

Sea un sistema que utiliza páginas de 4 KiB y cuyos bloques de sistema de ficheros son de 4 KiB. En este sistema existe un fichero denominado 'fichpru' de 8742 B que ocupa, por ese orden, los bloques 4, 7 y 1.

a) Recordando la forma en que el sistema de ficheros accede al disco, indicar las operaciones que hace el sistema de ficheros sobre el disco para atender el servicio: `write(fd, "X", 1);`*. Supondremos que el sistema de ficheros no tiene cache, por lo que hace todas las operaciones de lectura y escritura directamente sobre disco.*

b) Sean los dos programas adjuntos, que sobrescriben las 3000 primeras posiciones pares del fichero, respetando el valor original de las impares.

Seguimos suponiendo que el sistema de ficheros no tiene cache.

Para cada uno de los programas de la tabla 2.6, indicar por orden los 6 primeros accesos a disco que se generan dentro del bucle `for` *(externo para el programa 2). Detallar el bloque accedido y el tipo de acceso.*

c) Realizar un programa que realice la misma función pero proyectando el fichero en memoria. Analizar las operaciones de disco que se producen en este caso.

Programa 1

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
   int fd, i;
   fd = open("fichpru", O_WRONLY);
   for (i=0; i<3000; i++) {
      write(fd, "X", 1);
      lseek(fd, +1, SEEK_CUR);
   }
   close(fd);
   return 0;
}
```

Programa 2

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
   int fd, i, j;
   char buff[20];
   fd = open("fichpru", O_RDWR);
   for (i=0; i<300; i++) {
      read(fd, buff, 20);
      for (j=0; j<10; j++)
         buff[2*j] = 'X';
      lseek(fd, -20, SEEK_CUR);
      write(fd, buff, 20);
   }
   close(fd);
   return 0;
}
```

Tabla 2.6

Solución

a) Para atender el servicio mencionado, el sistema de ficheros debe determinar el bloque de disco afectado. Esto exige leer la tabla de descriptores de fichero del proceso así como la tabla de punteros (FILP). Ambas están en memoria, por lo que no afecta al disco. Con el valor del puntero debe obtener el bloque. Esta información está almacenada en el nodo-i (o estructura equivalente), por lo que pueden ser necesarias 0, 1 o más operaciones de lectura en el disco, dependiendo de que el nodo-i o estructura equivalente esté mantenido en memoria (lo que, por ejemplo, ocurre en los sistemas Unix), y de que el bloque sea directo, simple indirecto, doble indirecto o triple indirecto.

Una vez conocido el bloque afectado, el sistema de ficheros ha de modificar un solo byte de dicho bloque. Pero el disco no admite operaciones de 1 byte, solamente operaciones de un sector completo. Por tanto, el sistema de ficheros, escribirá siempre **bloques enteros**. Para no modificar el resto de los bytes del bloque lo que tiene que hacer es: primero, leer el contenido del bloque a memoria, segundo, modificar en memoria ese contenido con el nuevo byte y, tercero, reescribir el bloque completo al disco.

Solamente en el caso de la primera escritura en un nuevo bloque añadido a un fichero (o en el caso de que se escribiera un bloque completo) no hay que conservar información, por lo que no es necesario hacer la lectura.

b) Los accesos de datos a disco del programa 1 son : lee bloque 4, escribe bloque 4, lee bloque 4, escribe bloque 4, lee bloque 4, escribe bloque 4 ..... escribe bloque 7, lee bloque 7, escribe bloque 7.

El bucle se repite 3000 veces, por lo que se generan 3000 lecturas y 3000 escrituras al disco.

Los accesos de datos a disco del programa 2 son : lee bloque 4, lee bloque 4, escribe bloque 4, lee bloque 4, lee bloque 4, escribe bloque 4 ..... , lee bloque 7, lee bloque 7, escribe bloque 7.

El bucle se repite 300 veces, por lo que se generan 600 lecturas y 300 escrituras al disco.

c) Un posible programa es el siguiente:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
   int fd, i;
   char * data;

   fd = open("fichpru", O_RDWR);
   data = mmap(0, 6000, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
   close(fd);
   for (i=0; i<3000; i++)
      data[2*i] = 'X';
   munmap(data, 6000);
   return 0;
}
```

Al pasar por primera vez por la sentencia `data[2*i] = 'X';` se produce un fallo de página, por lo que el bloque 4 es leído. Más adelante se sobrepasa el bloque 4, por lo que se produce un segundo fallo de página, por lo que se lee el bloque 7.

Finalizado el bucle se libera la imagen de memoria, pero como está en MAP_SHARED el sistema operativo salva lo escrito. Esto significa que se escriben en el disco los bloques 4 y 7. El orden en que se escribe depende de la implementación del sistema operativo, por lo que podemos suponer uno cualquiera.

Es de notar que esta solución ha requerido solamente 2 lecturas y 2 escrituras al disco (más las posibles lecturas y escrituras relativas a la metainformación que no se está considerando).

Dentro del bucle no se produce ninguna llamada al sistema operativo.

Problema 2.11 (mayo 2005)

En un sistema UNIX existen los siguientes usuarios:

- *alumno1 (UID = 101)*
- *alumno2 (UID = 102)*
- *alumno3 (UID = 103)*
- *alumno4 (UID = 104)*
- *root (UID = 0)*

El grupo alumnosSO, cuyo identificador es el 300, incluye los siguientes usuarios: alumno1, alumno2. Por su parte, el grupo alumnosDSO, cuyo identificador es el 400, incluye los siguientes usuarios: alumno3, alumno4. El usuario root pertenece al grupo root (GID = 0).

Si se lleva a cabo un listado del directorio /fichs en formato largo e incluyendo información sobre los nodos-i (ls –lai) se obtiene la siguiente información:

Nodo-i	Permisos	Enlaces	Usuario	Grupo	Tamaño	Fecha	Nombre
41190	drwxrwxrwx	2	root	root	4096	2004-05-11	.
2	drwxr-xr-x	26	root	root	4096	2004-05-11	..
36890	-rws--x--x	1	root	root	28004	2004-08-12	cambiarClaves
37820	-rwxr-s--x	1	alumno2	alumnosSO	28004	2004-03-16	cambiarClaves2
54321	-rw-------	1	root	root	80	2004-04-10	claves
46230	-rw-r--r--	1	alumno2	alumnosSO	10	2004-07-22	datos
13357	-rw-r--r--	1	alumno3	alumnosDSO	334	2004-09-23	pre.dat
12356	-rw-r--r--	1	root	root	16170	2004-05-11	variables.pdf

Los permisos `rws------` implican que el propietario del fichero tiene permiso de lectura, escritura y ejecución y además que el fichero tiene activado el bit SETUID. Los permisos `---r-s---` implican que el grupo tiene permiso de lectura y ejecución y además que el fichero tiene activado el bit SETGID.

Se va a suponer que el sistema en caso de asignar nuevos nodos-i, lo hará utilizando valores crecientes a partir de 58743 inclusive.

Los ficheros cambiarClaves y cambiarClaves2 son dos ejecutables con el mismo código. A continuación se describe el código de ambos:

```
int main (int argc, char *argv[])
{  int fd, acceso, pos;
   struct stat estado;
   char buffer[1024]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";

   if (argc!=3) {
      fprintf(stderr,"Error\n");
      exit(1);
   }
   link (argv[1], argv[2]);

fd = open(argv[2], O_RDWR | O_APPEND);
if (fd>=0)
{
      write(fd, buffer, 10);
pos = lseek(fd, 20, SEEK_CUR);
      fstat(fd, &estado);
      printf("%d\n",estado.st_ino);
   }
   acceso = access(argv[2], R_OK);
   if (acceso ==0)
      unlink(argv[1]);
   return 0;
}
```

Responder a las siguientes cuestiones:

Si el usuario alumno1 lleva a cabo la siguiente ejecución:

```
$ /fichs/cambiarClaves /fichs/claves /fichs/claves2
```

a) ¿Cuál es el valor de la variable `fd`?
b) Después de ejecutar el servicio `lseek`, ¿cuál es el valor de la variable `pos`?
c) ¿Cuál es el valor impreso por la salida estándar?
d) Al final de la ejecución, ¿Cuántos enlaces físicos tiene el fichero cuyo nodo-i es 54321?

Si el usuario alumno3 lleva a cabo la siguiente ejecución:

```
$ /fichs/cambiarClaves2 /fichs/datos /fichs/datos2
```

e) ¿Cuál es el valor de la variable `fd`?
f) Al final de la ejecución, ¿cuántos enlaces físicos tiene el fichero cuyo nodo-i es 46230?

Solución

a) Dado que el ejecutable `/fichs/cambiarClaves` tiene activado el bit SETUID y pertenece al superusuario, el proceso tiene como identidad efectiva la del superusuario. Por tanto, puede acceder en modo lectura y escritura sobre el fichero `/fichs/claves`. La llamada `open` devolverá el descriptor más bajo disponible del proceso.

b) Dado que la apertura se ha realizado utilizando el modo `O_APPEND`, el puntero se coloca al final del fichero antes de cada escritura. Por tanto, los 10 bytes del buffer se escriben al final del fichero y a continuación se posiciona el puntero 20 bytes más allá del final de dicho fichero. Por tanto, la variable pos toma el valor 110 (80 (tam. Inicial de fichero) + 10 bytes escritos + 20 (posicionamiento)).

c) El valor impreso por la salida estándar corresponde al valor del nodo-i del fichero cuyo nombre es `/fichs/claves2`, que al ser un enlace físico del fichero con nombre `/fichs/claves`, tiene el mismo nodo-i. Por tanto, el valor impreso es: 54321

d) Dado que hemos creado un enlace físico y no hacemos el `unlink` correspondiente (`access` devuelve -1, debido a que comprueba los derechos de acceso de lectura con la identidad real del proceso, es decir, la del usuario alumno1), el número de enlaces del nodo-i 54321 es 2.

e) El proceso adquiere como GID efectivo el grupo alumnosSO. No obstante, el grupo sólo tiene permisos de lectura sobre el fichero `/fichs/datos` y, por tanto, también sobre el fichero `/fichs/datos2`. Esto provoca que la llamada open devuelva -1.

f) Dado que el proceso realiza un `link` y posteriormente un `unlink` (`access` devuelve 0, debido a que comprueba los derechos de acceso de lectura del fichero `/fichs/datos2` con la identidad real del proceso, es decir, la del usuario alumno3), el nodo-i 46230 tendrá un único enlace.

Problema 2.12 (febrero 2006)

En un sistema tipo UNIX se dispone de dos sistemas de ficheros A y B, soportados respectivamente en las particiones A y B. El sistema de ficheros B está montado en `/usr` del sistema A, según se muestra en la figura 2.10. Dicha figura muestra, para cada fichero, su nombre después del montaje, si es un fichero directorio o normal, el nombre del dueño y de su grupo, así como los permisos.

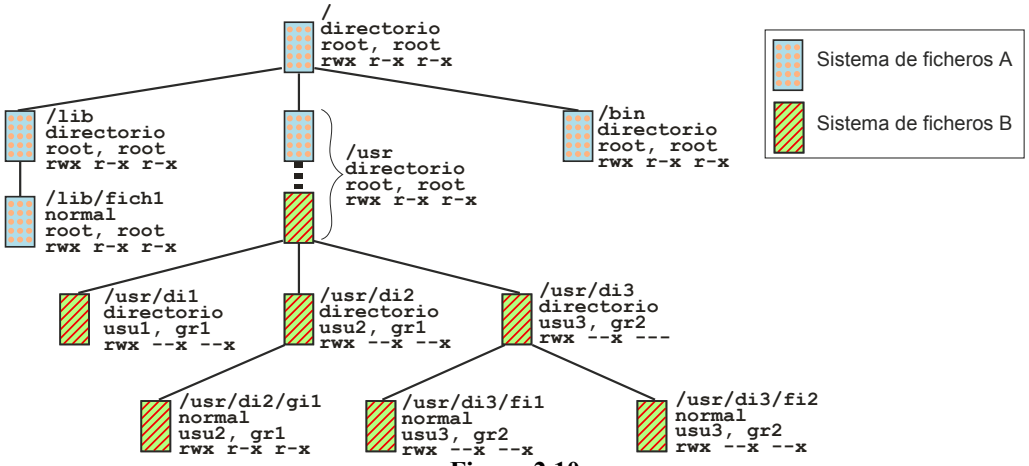


Figura 2.10

Consideraremos tres procesos: el X del usuario usu3 y grupo gr2, el Y del usuario usu1 y grupo gr1, y el Z del usuario root y grupo root. Inicialmente los procesos solamente tienen ocupadas las tres primeras entradas de sus tablas de descriptores fd.

Para cada paso de ejecución indicar todos y cada uno de los puntos siguientes, en caso de no modificarse alguno indicarlo expresamente:

1.- El valor devuelto por el servicio (por ejemplo, 0).

2.- Campos modificados en la tabla de descriptores (fd) del o de los procesos involucrados (por ejemplo, se rellena la entrada número 0 de la tabla fd del proceso M).

3.- Campos modificados en la tabla intermedia residente en memoria (por ejemplo, la tabla intermedia no se modifica).

4.- Modificaciones en nodos-i de ficheros de usuario en la tabla de nodos-i residente en memoria (por ejemplo, se incrementa el campo xxx del nodo-i del fichero yyy).

5.- Modificaciones en la partición A (por ejemplo, en la partición A se rellena un nuevo nodo-i y se añade una entrada en el directorio xxx con el valor yyy).

6.- Modificaciones en la partición B (por ejemplo, la partición B no se modifica).

Los pasos son los siguientes, que se ejecutan en el orden dado:

a) Proceso X: `fd1 = open ("/usr/di2/gi1", O_RDONLY);`
b) ProcesoY: `fd1 = open ("/usr/di2/fi1", O_RDONLY);`
c) Proceso X: `pid = fork ( );` *,ejecuta con éxito, generando el proceso XH*
d) Proceso XH: `fd1 = open("/lib/fich1", O_RDWR);`
e) Proceso Y: `li = symlink ("/lib/fich1", "/usr/di1/fich1");`
f) Proceso XH: `li = link ("/usr/di2/gi1", "/usr/di3/fich1");`
g) Proceso XH: `fd2 = dup(3);`

h) Proceso Y: `fd2 = open ("/usr/di1/fich1", O_RDONLY);`
i) Proceso Z: `lf = unlink ("/lib/fich1");`
j) Proceso Y: `n = read(fd2, buf, 128);`

Solución

a) En principio, el servicio ejecutará correctamente, puesto que se tienen permisos para llegar al fichero y para leer de él. Por tanto:

1.-Devuelve 3 (el enunciado dice que solamente están ocupados los fd estándar).

2.- Se modifica la entrada cuarta de la tabla de descriptores, incluyendo el valor de la entrada en la tabla intermedia.

3.- Se rellena una entrada en la tabla intermedia con el número de nodo-i del fichero `/usr/di2/gi1`, puntero ← 0 y nº de referencias ← 1.

4.- Se añade el nodo-i del fichero en la tabla de memoria (si no estaba antes) y nº de opens ← 1. Si el nodo-i ya estaba en la tabla, se incrementa el nº de opens.

5.- y 6.- No se modifican.

b) El servicio fracasa puesto que no existe el fichero, por lo que:

1.- Devuelve -1.

2.-, 3.-, 4.-, 5.- y 6.- No se modifican.

c) Nos dicen que el fork se ejecuta con éxito, luego:

1.- X recibe el PID de XH y XH recibe un 0.

2.- Se crea la tabla de descriptores de XH, copiándola de X.

3.- Se incrementan los nº de referencias de todos las entradas de la tabla intermedia referenciadas en la tabla de descriptores de HX.

5.- y 6.- No se modifican.

d) El servicio fracasa puesto que usu3 y gr2, si bien tienen permisos para llegar al fichero, no tienen derechos de escritura sobre el fichero.

1.- Devuelve -1.

2.-, 3.-, 4.-, 5.- y 6.- No se modifican.

e) En principio, el servicio se ejecuta con éxito, puesto que se tienen permisos para llegar al fichero `"/usr/di1/fich1"`, así como permisos de escritura en el mismo. Dado que es un enlace simbólico no se comprueba `"/lib/fich1"`. Por lo que:

1.- Devuelve 0.

2.-,3.- y 4.- No se modifican.

5.-No se modifica, no se incrementa el nº de enlaces del correspondiente nodo-i.

6.-Se utiliza un nuevo nodo-i que se marcará como enlace simbólico y que indicara un bloque que contendrá el enlace, es decir, la cadena: `/lib/fich1`. Se añade una entrada en el directorio `/usr/di1` que asocia el nombre `fich1` y el número de nodo-i anterior. También se actualiza el tiempo de modificado en el nodo-i del directorio `/usr/di1`.

f) En principio, el servicio se ejecuta con éxito, puesto que se tienen permisos para llegar al fichero `"/usr/di3/fich1"` así como permisos de escritura en el mismo, Por otro lado, se tienen derechos de acceso en el camino `"/usr/di2/"` y existe el fichero `gi1`, por lo que se puede obtener su nodo_i. Por tanto:

1.- Devuelve 0.

2.- ,3.-, 4.- y 5.- No se modifican.

6.- Se modifica el nodo-i del fichero `/usr/di2/gi1` incrementado el nº de enlaces. Se añade una entrada en el directorio `/usr/di3` que asocia el nombre `fich1` y el número de nodo-i del fichero `/usr/di2/gi1`. También se actualiza el tiempo de modificado en el nodo-i del directorio `/usr/di3`.

g) En principio, el servicio se ejecuta con éxito, dado que el descriptor 3 es válido (fue heredado). Como están utilizadas las entradas 0, 1, 2 y 3, se asigna la 4.

1.-Devuelve 4.

2.- Se modifica la entrada quinta de la tabla de descriptores, copiando el valor de la entrada en la tabla intermedia que tiene la entrada cuarta.

3.- Se modifica la correspondiente entrada en la tabla intermedia incrementado el nº de referencias.

4.-, 5.- y 6.- No se modifican

h) En principio, el servicio ejecutará correctamente, puesto que se está abriendo el fichero enlazado en el paso e) y se tienen derechos, por tanto:

1.-Devuelve 3 (el enunciado dice que solamente están ocupados los fd estándar).

2.- Se modifica la entrada cuarta de la tabla de descriptores, incluyendo el valor de la entrada en la tabla intermedia.

3.- Se rellena una entrada en la tabla intermedia con el número de nodo-i del fichero `/lib/fich1`, puntero ← 0 y no de referencias ← 1. Observe que el nodo-i es el del fichero real, no el del enlace simbólico.

4.-Se añade el nodo-i del fichero `/lib/fich1` en la tabla de nodos-i en memoria (si no estaba antes) y nº de opens ← 1. Si el nodo-i ya estaba en la tabla, se incrementa el nº de opens.

5.- y 6.- No se modifican.

i) En principio, el servicio se ejecuta con éxito. El fichero está abierto y seguirá abierto. No se borra hasta que no tenga ningún usuario, por lo que:

1.-Devuelve 0.

2.- y 3.- No se modifican.

4.- En la tabla de nodos-i en memoria se decrementa el nº de enlaces del nodo-i del fichero `/lib/fich1`. No se elimina este nodo-i, puesto que todavía tiene usuarios como es el proceso Y.

5.-Se modifica el nodo-i del fichero `/lib/fich1` decrementado el nº de enlaces. Si el contador de enlaces llega a 0, en principio, habría que recuperar los recursos asignados al fichero (nodo-i y bloques). Sin embargo, el nodo-i en memoria tiene nº de opens = 1 (paso h), por lo tanto, el fichero no se puede borrar aún. Cuando el proceso Y cierre dicho fichero o termine, es cuando se recuperarán realmente el nodo-i y los bloques. Sí se borra la entrada de valor `fich1` en el directorio `/lib`. También se actualiza el tiempo de modificado en el nodo-i del directorio `/lib`.

6.- No se modifica.

j) En principio, el servicio se ejecuta con éxito, puesto que el proceso tiene abierto el fichero, por lo que:

1.- Devuelve 128, siempre que el fichero tenga datos suficientes.

2.- No se modifica.

3.-Se incrementa el puntero de posición de la entrada correspondiente en la tabla intermedia.

4.-, 5.-y 6.- No se modifican. La actualización de los tiempos de último acceso dependen de la implementación, por lo que no se han contemplado.

Problema 2.13 (abril 2006)

En un sistema consideraremos los tres usuarios juan (uid 100, gid 100), pablo (uid 101, gid 100) y fran (uid 202, gid 200). En dicho sistema, los contenidos en el instante X de algunas agrupaciones (no todas) y algunos nodos-i (no todos) de dos dispositivos ***hda*** *y* ***sda*** *que tiene el sistema son los mostrados en las tablas adjuntas. Además, el sistema de ficheros sda está montado sobre `/home` del hda.*

Dispositivo hda

Agrupación	***Contenido (nombre - nº nodo_i)***
3	*. 2; .. 2; usr 121; home 243; etc 453; boot 54; dev 38; sys 15; sbin 73*
62	*. 73; .. 2; link 174;……*

73	*. 243; .. 2; abc 742; tyz 375*
213	*abcdefghijk...*
532	*^?ELF^A^A^A^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@P....*
7	*. 453; .. 2; passwd 13;……*
56	*root:x:0:0:Superusuario:/root:/bin/bash……*

Nodo-i	***Dueño (UID-GID)***	***Permisos***	***Agrupaciones***	***Tamaño***	***Nº enlaces***
2	*Root (0-0)*	*d rwx r-x r-x*	*3*	*180*	*9*
243	*Root (0-0)*	*d rwx r-x r-x*	*73*	*273*	*2*
73	*Root (0-0)*	*d rwx r-x r-x*	*62*	*512*	*2*
174	*Root (0-0)*	*rwx r-x r-x*	*532, 732, 555, ...*	*270053*	*1*
742	*Root (0-0)*	*rw- r-- r--*	*213*	*654*	*1*
453	*Root (0-0)*	*d rwx r-x r-x*	*7*	*304*	*9*
13	*Root (0-0)*	*rw- r-- r--*	*56*	*978*	*1*
375	*Root (0-0)*	*rw- r-- r--*	*333*		

Dispositivo sd*a*

Agrupación	***Contenido***
3	*. 2; .. 2; juan 15; pablo 243; fran 453*
25	*.15; ..2; j1 460; j2 461*
30	*.243; ..2; p1 701; p2 650*
43	*.453; ..2; f1 534; f2 475*
67	*000*
123	*671482*
234	*000000*

Nodo-i	***Dueño (UID-GID)***	***Permisos***	***Agrupaciones***	***Tamaño***	***Nº enlaces***
2	*Root (0-0)*	*d rwx r-x r-x*	*3*	*345*	*5*
15	*juan (100-100)*	*d rwx r-x r-x*	*25*	*274*	*2*
243	*pablo (101-100)*	*d rwx rwx r-x*	*30*	*325*	*2*
453	*fran (202-200)*	*d rwx r-x r-x*	*43*	*156*	*2*
460	*juan (100-100)*	*rw- rw- ---*	*234*	*6*	*1*
701	*pablo (101-100)*	*rw- rw- rw-*	*123*	*4*	*1*
475	*fran (202-200)*	*rw- r-- r--*	*67*	*3*	*1*

En el mismo instante X, el sistema operativo tiene, entre otras, las siguientes tablas de descriptores

Proceso 273 (juan)

fd	***Entrada Tabla Intermedia***
0	*1*
1	*2*
2	*2*
3	*0*
4	*0*
5	*0*
6	*0*

Proceso 678 (pablo)

fd	***Entrada Tabla Intermedia***
0	*3*
1	*4*
2	*4*
3	*8*
4	*0*
5	*10*
6	*0*

Proceso 534 (fran)

fd	***Entrada Tabla Intermedia***
0	*5*
1	*6*
2	*6*
3	*7*
4	*0*
5	*9*
6	*0*

Y la siguiente tabla intermedia:

	Nodo-i	***Posición***	***Referencias***
.....	*.....*	*.....*	*.....*
7	*sda-460*	*2*	*1*

8	*sda-701*	*2*	*1*
9	*sda-701*	*1*	*1*
10	*sda-460*	*0*	*1*
11	.....	.....	.....

Partiendo de la situación de las tablas descritas anteriormente, responder a las siguientes preguntas.

a) El proceso 534 ejecuta la cláusula siguiente: `fd = open ("/home/abc", O_RDWR);`

¿Qué valor devuelve el servicio? ¿Por qué?

b) Para que el proceso 678 abra para lectura con éxito el fichero de nombre local `j1` *(nº nodo_i = sda-460), ¿cuál debe ser el valor del primer argumento del open?*

c) ¿El superusuario podría realizar la operación de umount del sistema de ficheros sda en el instante X?

d) El proceso 534 ejecuta: `n = write(5, "65414",2);`

seguidamente el proceso 678 ejecuta: `n = write(3, "3453",4);`

Indique los cinco primeros caracteres contenidos en el fichero de nº nodo_i = sda-701

e) El proceso 678 crea un hijo (proceso 789) y, seguidamente, se ejecutan las siguientes escrituras:

Proceso 789: `n = write(5, "21345", 2);`

Proceso 678: `n = write(5, "18212", 3);`

Indique los cinco primeros caracteres contenidos en el fichero de nº nodo_i = sda-460

f) El programa `/sbin/link` *es el mandato de sistema link que produce enlaces físicos siendo el primer argumento el fichero existente. El usuario fran está ejecutando un interprete de mandatos (proceso 345) y desea establecer un enlace: ¿Cuál de los siguientes mandatos produciría realmente un enlace?*

```
$ /sbin/link /home/juan /home/fran/juan
$ /sbin/link /etc/passwd /home/fran/pw
$ /sbin/link /home/juan/j1 /home/juan/j3
$ /sbin/link /home/juan/j2 /home/fran/j2
```

g) Cuántos subdirectorios tiene el directorio /etc.

h) El proceso 273 tiene como máscara 011 y el proceso 678 tiene 033. El proceso 273 ejecuta:

```
fd = open("/home/pablo/dat.txt", O_WRONLY|O_CREAT|O_TRUNC, 0777);
```

Indicar si se crea o no el fichero, y en caso de crearse los permisos con los que se crea.

Solución

a) Al estar montado el dispositivo sda sobre `/home` del hda, los subdirectorios y ficheros del `/home/` dejan de ser accesibles, por lo que nadie puede abrir el fichero "/home/abc". El usuario fran puede atravesar el directorio `/home/` , pero no encuentra el fichero `abc`. Si se desmontase el dispositivo sda volvería a ser accesible el mencionado fichero `abc`.

b) El valor ha de ser: "`/home`/juan/j1"

c) No se puede desmontar el dispositivo sda puesto que hay ficheros del mismo abiertos, como se observa en la tabla intermedia del enunciado.

d) El resultado es 66345

e) El resultado es 21182

f) Para poder hacer un enlace físico es necesario disponer de permisos de escritura en el directorio destino y tener permisos de acceso al fichero origen. El único mandato que cumple estas condiciones es el siguiente:$ /sbin/link /home/juan/j1 /home/juan/j3

g) El directorio `/etc` tiene 7 subdirectorios. Dado que tiene 9 enlaces, de los cuales uno pertenece al nombre «etc» que se encuentra en la tabla directorio raíz / y otro pertenece al nombre «.» de la tabla directorio /etc/. Los 7 restantes corresponden al nombre «..» de otros tantos subdirectorios del /etc/.

h) El fichero, en principio, se crea puesto que el usuario juan tiene permisos de acceso al directorio `/home/pablo` y permisos de escritura en el mismo. Teniendo en cuenta que la máscara del proceso 273 es 011 y que en el open se solicitan los permisos 0777, los permisos con los que se crea el fichero `dat.txt`, son 0766

Problema 2.14 (junio 2006)

Se desea implementar un spooler, un programa que recorre un directorio de trabajos pendientes (`/var/spool/works/`) y los envía a otro programa que los ejecuta. Cuando se desea que un trabajo se lance por medio del spooler se escribe un nuevo fichero en el directorio compartido, que este proceso inspeccionará periódicamente. Las características del funcionamiento del programa spooler son las siguientes:

- *Durante el recorrido del directorio se comprueba si hay nuevas entradas (saltándose las entradas '.' y '..'). Esta operación la realizará la función `recorrer_directorio`.*
- *La verificación de si hay trabajos pendientes en el directorio se ejecuta cada segundo. La función `main` programará la temporización y preparará a la función anterior (`recorrer_directorio`) para que se ejecute en ese momento.*
- *Por cada fichero encontrado en el directorio se debe ejecutar el programa `/bin/ejecutor`, al que se le pasará por su entrada estándar el contenido del fichero. La ejecución de este programa se hará desde la función `procesar_fichero`.*
- *El fichero de un trabajo, una vez procesado, se debe borrar. Esto también lo hará la función `procesar_fichero`.*

a) Implementar la función `recorrer_directorio`

b) Implementar el mecanismo de temporización.

c) Implementar la función `procesar_fichero`.

d) La implementación de este spooler presenta una posible condición de carrera, ya que podría estar a medias de escribir un fichero en el directorio mientras el spooler lee, manda el trabajo y lo borra. Sin implementar nada, proponga cuál sería un buen mecanismo para asegurar la sincronización del spooler y los potenciales procesos pesados no emparentados que escriben trabajos en el directorio compartido.

e) En la escritura y posterior borrado de trabajos se plantea otro problema. Los usuarios que escriben trabajos en el directorio no tienen por qué ser los mismos que el usuario que ejecuta el spooler (que será, probablemente, un usuario del sistema llamado `spool`). Supongamos que los usuarios usan un programa llamado submit, para programar trabajos (escribirlos en el directorio), y que el procesamiento es por medio del programa spooler que ya hemos visto. ¿Cuáles serían las consideraciones sobre propietarios y permisos que habría que tener en cuenta en los ficheros, directorios y programas que participan?

Solución

a) Complete el código de la función `recorrer_directorio` para que realice las operaciones descritas anteriormente.

```
void recorrer_directorio(char* dir)
{
   /* Declaración de variables */
   DIR* dir_handler;
   struct dirent* entrada;
   char buffer[1024];

   /* Apertura del directorio */
   dir_handler=opendir(dir);

   while((entrada=readdir(dir_handler))!=NULL)

   {
      /* Verificación de la entrada (saltarse '.' y '..'), e invocación de la función procesa_fichero(...) */
      if((strcmp(entrada->d_name,"..")==0)||(strcmp(entrada->d_name,".") ==0))
```

```
        continue;

      sprintf(buffer,"%s/%s",dir, entrada->d_name);
      procesar_fichero(buffer);
   }
   /* Cierre del directorio */
   closedir(dir_handler);
}
```

b) La función `recorrer_directorio` se debe ejecutar con la periodicidad descrita anteriormente. Implemente la programación de esa periodicidad (sin usar `sleep` o cualquier otro mecanismo de espera bloqueante, ya que se puede requerir del proceso que ejecute otras operaciones mientras no se active la temporización).

```
int main(int argc , char *argv[])
{
   /* Temporización */
   signal(SIGALARM,manejador);
   alarm(1);
   /* Seguiría haciendo operaciones *)
   ...
}
/* Funciones auxiliares */
void manejador (int sig)
{
    recorrer_directorio("/var/spool/work");
    signal(SIGALARM,manejador); // Volver a montar la señal
    alarm(1);
}
```

c) Complete el código de la función `procesar_fichero` para que realice las operaciones descritas anteriormente.

```
void procesar_fichero(char* fichero)
{
   /* Declaración de variables */
   int pp[2];
   char buffer[4096];
   int fd, leidos;

   /* Construcción del mecanismo de comunicación entre ambos procesos */
   pipe(pp);

   switch(fork())
   {
      case -1:
           /* Error */
      perror("fork()");
      exit(1);

      case 0:
           /* Hijo */
      close(0);
      dup(pp[0]);
      close(pp[0]);
      close(pp[1]);
      execlp("ejecutor",NULL);
      perror("execlp()");

      default:
           /* Padre */
      close(pp[0]);
      if((fd=open(fichero,O_RDONLY))<0))
        perror("open()"); exit();
      while((leidos=read(fd,buffer,4096))>0)
```

```
            write(pp[1],buffer,leidos);
         close(pp[1]);
         wait(NULL);
      }
      /* Borrado del fichero */
      unlink(fichero);
}
```

Existe una implementación alternativa a este apartado c) que es más compacta

```
void procesar_fichero(char* fichero)
{
   /* Declaración de variables */
   int fd;

   switch(fork())
   {
      case -1:
            /* Error */
      perror("fork()");
      exit(1);

      case 0:
            /* Hijo */
      close(0);
      if((fd=open(fichero,O_RDONLY))<0))
         perror("open()"); exit();
      execlp("ejecutor",NULL);
      perror("execlp()");

      default:
            /* Padre */
      wait(NULL);
   }
   /* Borrado del fichero */
   unlink(fichero);
}
```

d) Hay varias alternativas a este nivel, lo fundamental es que hay que considerar que los procesos que participan, el `spooler` y los procesos que dejan trabajos en el directorio no están emparentados. De esta forma se tiene que optar por un mecanismo que valga para ese caso.

Las alternativas posibles serían (todas ellas válidas):

- Semáforos con nombre: Creado por el `spooler` y cuyo nombre es conocido por los otros programas. Este semáforo gestionaría la región crítica en un modelo lectores-escritores.
- Un pipe con nombre (o FIFO). Se podría crear en el mismo o en otro directorio y podría tener un byte de contenido que haría de token para acceder al directorio. Antes de hacer operaciones sobre el mismo lee del FIFO, si lee un byte (el FIFO se queda vacío) y se pueden hacer las operaciones sobre el directorio. Al salir del directorio y terminar de hacer las operaciones se vuelve a escribir el byte en el FIFO. Este mecanismo asegura exclusión mutua al ser operaciones atómicas el `read` y `write` de ese tamaño.
- Una última alternativa podría ser el uso de cerrojos de ficheros. Esta alternativa no se ha visto en el curso, pero es muy utilizada en procesos de sistema. Las llamadas `flock()` y `lockf()` permiten establecer cerrojos sobre ficheros.

e) La opción más sencilla es hacer que el propietario del directorio `/var/spool/work` sea el propio usuario `spool`. De forma que, al menos el permiso de escritura, sea sólo para este usuario, el propietario del directorio. Lo que se haría es que el programa `submit` tenga activado el bit "s" (*setuid*), de forma que la identidad efectiva de cualquiera que ejecute ese programa sea la del propietario del mismo. Es decir que cualquiera que ejecute `submit` se convierta en el usuario `spool` de forma efectiva. De esta forma el programa podrá escribir en el directorio nuevas entradas sin problemas.

Cualquier otra solución es problemática, por ejemplo que todos los usuarios de un grupo o todos los de una máquina puedan escribir en el directorio, porque implicaría que programas que no sólo ponen trabajos en el directorio pudieran manipular su contenido. Uno podría copiar, mover o renombrar sus ficheros en ese directorio. Adicionalmente el `spooler` no podría borrar los ficheros de otros usuarios al no ser propietario de los mismos (como ocurre en el `/tmp`).

Problema 2.15 (septiembre 2006)

El programador A debe escribir el código de un ejecutable denominado escribirRodaja que cumpla los siguientes requisitos:

- *1. Recibe tres argumentos: el nombre de un fichero y dos números enteros, que representan dos desplazamientos (offsets) dentro del fichero: desp1 y desp2. Se supone que $0 \leq desp1 \leq desp2$.*
- *2. Si el fichero no existe, lo crea con un tamaño igual a desp2 y con contenido todos los bytes a cero.*
- *3. Si el fichero existe y su tamaño es superior a desp2, se leen los bytes existentes entre desp1 y desp2 (ámbos inclusive) y se escriben por la salida estándar. En el caso de que los dos desplazamientos, desp1 y desp2, sean iguales, sólo se escribe el byte correspondiente.*
- *4. Si el fichero existe y desp2 es superior o igual al tamaño del fichero, se cierra dicho fichero y no se escribe nada por la salida estándar.*

El programador A desea estructurar de forma modular el código fuente correspondiente al ejecutable escribirRodaja, haciendo uso, entre otras, de las dos siguientes funciones:

- *int crear_fichero(char *nombre_fich, int tamanyo);, que se utiliza para resolver el requisito número 2. Esta función recibe como argumentos el nombre y el tamaño del fichero y crea un nuevo fichero con dicho nombre y tamaño y relleno a ceros. La función devuelve el descriptor del nuevo fichero creado o -1 en caso de error.*
- *int leer_fichero_despl (int fd, int desp_min, int desp_max);, que se utiliza para resolver el requisito número 3. Esta función recibe un descriptor de fichero y dos desplazamientos, uno mínimo y otro máximo. La función lee del fichero a partir de la posición desp_min hasta desp_max y escribe dichos bytes por la salida estándar. La función devuelve el número de bytes escritos por la salida estándar o -1 en caso de error. Se supone que $0 \leq desp\_min \leq desp\_max <$ tamaño del fichero.*

Poniéndose en el caso del programador A:

a) Implementar la función int crear_fichero(int tamanyo); correspondiente al ejecutable escribirRodaja.

b) Implementar la función int leer_fichero_despl (int fd, int desp_min, int desp_max); correspondiente al ejecutable escribirRodaja.

El programador B recibe el ejecutable escribirRodaja, teniendo que programar el código correspondiente a otro ejecutable (llamado crearSubficheros), que haga lo siguiente:

- *Recibe dos argumentos: el nombre de un fichero y un número entero mayor que 0.*
- *El segundo argumento representa el número de subficheros en los que se debe dividir el contenido del fichero de entrada, al que hace referencia el primer argumento. La concatenación del contenido de todos los subficheros, ordenados por nombre o sufijo, debe ser igual al contenido del fichero de entrada. Los nombres de estos subficheros son iguales a los del fichero de entrada, añadiéndoles el sufijo "_x", donde x vale 0 para el primer subfichero, 1 para el segundo y así sucesivamente. Para crear los nombres de los subficheros se dispone de una función char * crear_nombre_subfichero (char *nombre_fich, int numero); que dado un nombre de fichero y un número devuelve el nombre del subfichero correspondiente. Esta función NO HAY QUE IMPLEMENTARLA.*
- *El ejecutable crearSubficheros divide el número de bytes del fichero original por el número de subficheros para calcular cuántos bytes secuenciales deben escribirse en cada subfichero. Se supone que el número de bytes del fichero original es múltiplo exacto del número de subficheros.*
- *Se supone que el usuario que ejecute el programa tiene permisos de lectura del fichero de entrada y permisos de escritura sobre el directorio donde está el fichero de entrada y donde debe crear los subficheros.*

Poniéndose en el caso del programador B:

c) Implementar el código correspondiente al ejecutable crearSubficheros, haciendo uso del ejecutable escribirRodaja y de la función crear_nombre_subfichero.

d) Suponga que se desea implementar el código del ejecutable crearSubficheros sin hacer uso del ejecutable escribirRodaja y utilizando una solución multihilo, en la cual cada hilo se encargue de resolver la escritura de cada subfichero. ¿Qué ventajas y desventajas ofrecería esta solución respecto a la solución del apartado anterior?

Solución

a) Hay varias alternativas. Dos de ellas son:

```
int crear_fichero_v1(char *nombre_fich, int tamanyo)
{
   int fd;
   fd = creat(nombre_fich, 0666);
   ftruncate(fd, tamanyo);
   return fd;
}

int crear_fichero_v2(char *nombre_fich, int tamanyo)
{
   int fd;
   char byte = 0;
   fd = creat(nombre_fich, 0666);
   lseek(fd, tamanyo-1, SEEK_SET);

   write(fd, &byte, 1);
   return fd;
}
```

b)

```
int leer_fichero_despl(int fd, int desp_min, int desp_max)
{
   char *buffer;
   int escritos = 0;

   buffer = (char *) malloc(desp_max-desp_min+1);
   if (!buffer)
      return -1;

lseek(fd, desp_min-1, SEEK_SET);
   escritos = desp_max-desp_min+1;
   read(fd, buffer, escritos);
   write(1, buffer, escritos);

free(buffer);
   return escritos;
}
```

c)

```
// ejecutable crearSubficheros
int main (int argc, char *argv[])
{

   int num_subfich;
   int i;
   int fd;
   char *nombre_subfich;
   char desp1[6];
   char desp2[6];
   struct stat inf_fich;
   int num_bytes_total, num_bytes_fich;
   int offset;

   if (argc != 3)
```

```
   {
      fprintf(stderr, "Error: Uso: ./crearSubficheros nombre_fich
num_subfich\n");
      return 1;
   }

   num_subfich = atoi(argv[2]);
   if (num_subfich <=0)
   {
      fprintf(stderr, "Error: Numero de subficheros debe ser mayor que 0\n");
      return 1;
   }

   stat(argv[1], &inf_fich);
   num_bytes_total = inf_fich.st_size;
   num_bytes_fich = num_bytes_total/num_subfich;

   offset = 0;

   for (i=0; i<num_subfich; i++)
   {
      switch(fork())
{
case -1: perror("fork");
return 1;
   case 0:
nombre_subfich=crear_nombre_subfichero(
argv[1]);
     close(1);
fd=creat(nombre_subfich, 0666);
sprintf(desp1, "%d",offset);
sprintf(desp2, "%d", offset+num_bytes_total);
execlp("escribirRodaja", "escribirRodaja", argv[1], desp1, desp2, NULL);
          perror("execlp escribirRodaja 1");
          close(fd);
          return 2;
        default:
          offset = offset+num_bytes_fich;
     }
}
for (i=0; i<num_subfich; i++)
   wait(NULL);

   return 0;
}
```

d) En general, el uso de *threads* es adecuado para soluciones concurrentes, que permitan dividir un proceso en tareas que se puedan ejecutar concurrentemente dentro del propio proceso. En este tipo de escenario, los *threads* pueden mejorar el rendimiento de la solución, dado que la creación de un proceso pesado implica mayor carga que la creación de un *thread*. No obstante, en este caso, al tratarse de la lectura del fichero de entrada, habría problemas para sincronizar la lectura del mismo de una forma correcta. La creación de los ficheros de salida no sería un problema. Si se siguiera la filosofía de la solución anterior (utilizando escribirRodaja) tendríamos problemas, dado que todos los procesos ligeros tendrían la misma salida estándar y se mezclarían los resultados. Si se hace de otro modo, tendríamos que sincronizar el acceso al fichero de entrada.

Problema 2.16 (febrero 2007)

Se desea implementar un programa que cifre los contenidos de todos los ficheros de un directorio. El programa recibirá dos argumentos: (1) el nombre de un ejecutable que hará de cifrador y (2) el directorio a cifrar. El programa

no tiene que cifrar subdirectorios, sólo los ficheros que se encuentre en el directorio pasado como argumento. Las características del funcionamiento del programa son las siguientes:

- *Se recorre el directorio dado con la función recorrer_directorio.*
- *Durante el recorrido del directorio, por cada fichero que se localice, se llamará a la función cifrar_fichero.*
- *Para cifrar el fichero se ejecutará el cifrador, este ejecutable recibe por la entrada estándar los contenidos a cifrar y genera por la salida estándar dichos contenidos cifrados. Esta tarea la realiza la función cifrar_fichero.*
- *Al final de la ejecución, el directorio debe contener los mismos nombres de ficheros, pero sus contenidos deben de estar cifrados.*

a) Implementar la función recorrer_directorio.

b) El proceso de cifrado manipula dos ficheros, el original y el cifrado. Potencialmente, que estos dos ficheros tengan el mismo nombre puede ser un problema. Considerando que no se puede dar un fallo o caída en el programa durante el proceso de cifrado, existen tres alternativas para hacerlo:

1. Renombrar el fichero original, crear el fichero cifrado con el antiguo nombre del original, cifrar y finalmente borrar el fichero original renombrado. (Usando rename).
2. Abrir el fichero original, borrar el nombre de fichero manteniéndolo abierto y crear el fichero cifrado con dicho nombre. Luego cifrar y cerrar los ficheros. (Usando unlink)
3. Abrir el fichero original, enganchar la salida del cifrador a una tubería (pipe), cifrar, borrar el fichero original y crear un fichero con el mismo nombre y volcar el contenido del pipe a dicho fichero. (Usando pipe).
¿Alguna de estas alternativas no funcionaría correctamente? Indique para cada alternativa sus ventajas e inconvenientes.

c) Implementar la función cifrar_fichero.

d) Si reconsideramos el apartado b), planteándonos la posibilidad de que el proceso falle bajo las siguientes condiciones:

d.i) Que el proceso de cifrado y/o el que recorre el directorio se caiga en cualquier punto de la ejecución.
d.ii) Que el espacio en disco sea muy escaso y pueda agotarse, sobretodo al cifrar ficheros de gran tamaño.
Bajo estos escenarios, comente las alternativas 1, 2 y 3, del apartado b) bajo estas dos posibles causas de error ¿Cómo calificaría la fragilidad de cada una de las soluciones?

Solución

a)

```
int main(int argc, char *argv[])
{
  if(argc!=3)
  {
     fprintf(stderr,"uso: %s prog_cifrador directorio\n",argv[0]);
     return 1;
  }
  recorrer_directorio(argv[2],argv[1]);
  return 0;
}

void recorrer_directorio(char* cifrador, char* directorio)
{
  DIR* dir;
  struct dirent* ent;
  char buff[MAXPATHLEN];

  dir=opendir(directorio);

  while((ent=readdir(dir))!=NULL)
    {
```

```
        /* Si son las entradas . y ..  nos las saltamos */
        if(!strcmp(ent->d_name,".") || !strcmp(ent->d_name,".."))
          continue;
        /* Componemos el path del directorio */
        sprintf(buff,"%s/%s",directorio,ent->d_name);
        cifrar_fichero(buff,cifrador);
      }
    closedir(dir);
}
```

b) Todas las alternativas funcionan correctamente, la diferencia radica en cómo utiliza los recursos (bloques de disco y espacio en los directorios), en el caso de b.1) lo que se hace es duplicar los ficheros, es decir mantener el original bajo otro nombre e ir construyendo el cifrado. Esta opción, como se puede observar es menos eficiente en el consume de recurso (espacio en disco y entradas de directorio). La solución es quizás la más intuitiva y la que puede generar menos problemas.

La opción b.2) aparentemente parece que puede dar un error, pero no es así. Un fichero cuando se borra no se liberar sus bloques de datos del disco hasta que lo ha cerrado el último programa que lo ha abierto. Lo que sí que es verdad, es que el nombre del fichero, la entrada dentro del directorio se elimina, de forma que en el intervalo entre el borrado (unlink) y el momento en el que finalice el proceso que lo tiene abierto ningún otro proceso puede acceder al fichero. El consumo de bloques de disco es el mismo, lo bloques originales no se liberan hasta el close del fichero y los nuevos contenidos del fichero ya codificado empezarán a consumir bloques. Un aspecto interesante de esta alternativa es que la solución es más elegante al no aparecer varios nombres de fichero en el directorio y haciendo que los ficheros en fase de cifrado sólo sean accesibles por el proceso que realiza la operación.

La opción b.3) puede parecer aún más refinada que la anterior, aquí los bloques de disco se liberan antes de escribir el nuevo fichero. Comparte con la solución anterior el no generar nombre intermedios de ficheros o entradas extrañas en el directorio, pero esta vez usando el buffer del pipe como almacén de los datos. Esta solución sin embargo sí tiene un claro problema que puede hacer que no sea válida. La capacidad de almacenamiento del pipe es limitada, si se intentan meter más datos de los 4 KiB que generalmente tiene como espacio disponible hace que el write se bloquee. Esto haría que el programa cifrador se quedase también bloqueado. Es, con diferencia, la peor de las soluciones.

- b.1) La más segura
- b.2) La más refinada
- b.3) No funciona en muchos casos.

c)

```
void cifrar_fichero(char* fichero, char* prog_cifrador)
{
  /* Seleccionada la opción b.2), borrar el fichero abierto */

  switch(fork())
    {
    case -1: /* Error */
      perror("fork");
      exit(1);
    case 0:   /* Hijo */
      /* Cerramos la entrada estándar */
      close(0);
      /* Al abrir el fichero toma el descriptor 0 */
      open(fichero, O_RDONLY);
      /* Borrar el fichero, no se liberan los bloques hasta que
         se haga el close del fichero que está abierto */
      unlink(fichero);
      /* Cerramos la salida */
      close(1);
      /* Abrirmos el descriptor 1 como el mismo nombre de
         fichero */
      creat(fichero, 0660);
      execlp(prog_cifrador, prog_cifrador, NULL);
      perror("execlp");
      exit(1);
```

```
    default:
      wait(NULL);
    }
}
```

d. i) Que el proceso de cifrado y/o el que recorre el directorio se caiga en cualquier punto de la ejecución.

En este caso las alternativas b.2 y b.3 pueden plantear problemas, en ambos casos el contenido del fichero original se perdería y nos quedaría una versión parcial del cifrado.

La opción d.1 evita ese problema, si se corta el proceso quedaría dos ficheros, el original y parte del contenido del cifrado, bastaría con borrar el cifrado, volver a renombrar el original y repetir el proceso.

d. ii) Que el espacio en disco sea muy escaso y pueda agotarse, sobretodo al cifrar ficheros de gran tamaño.

Aquí el problema con b.1 y b.2 es el mismo, aunque b.2 borre el nombre del fichero sus bloques de datos siguen existiendo y al ir cifrando se duplicaría este espacio de disco.

La opción b.3 no tiene ese problema pero, como ya vimos, su funcionamiento está restringido a ficheros que entren en el buffer del pipe (4 KiB). Esa restricción no se cumple en ficheros grandes así que no es tampoco una buena alternativa.

Quizás la mejor solución efectiva a este caso sería almacenar el fichero en memoria y desde ella cifrar y sobrescribir el fichero. La única restricción sería la memoria virtual disponible para los procesos (a priori, la memoria física más el espacio de swap, siempre y cuando los procesos de usuario no tengan restricciones adicionales por parte del administrador).

Problema 2.17 (abril 2007)

Dado el siguiente código:

```
1  int main(int argc, char *argv[])
2  {
3     int fd1, fd2, bytes;
4     char ch1, ch2, buff[64];
5     umask(0022);
6     fd1=open("f.txt", O_CREAT|O_RDWR|O_TRUNC, 0660);
7     fd2=open("f.txt", O_RDWR);
8
9     write(fd1,"9876543210",6);
10    lseek(fd1,2,SEEK_SET);
11    read(fd1,&ch1,1);
12 /* INSTANTE A */
13
14    lseek(fd2,-2,SEEK_END);
15    read(fd2,&ch2,1);
16    write(fd2,&ch2,1);
17 /* INSTANTE B */
18
19    bytes=read(fd1,buff,64);
20 /* INSTANTE C */
21
22    ...
23 }
```

a) ¿Con qué bits de protección se crearía el fichero `f.txt`? Indíquelo en formato numérico octal.

b) Suponiendo que el proceso tiene abiertos sólo los descriptores por defecto (entrada, salida y error estándar), ¿cuál sería la situación de las estructuras de gestión de ficheros tras la realización de los dos open?

c) En el INSTANTE A, ¿qué dígito se habrá leído en la variable `ch1`?

d) Si se hiciera un `lseek` más allá del tamaño actual del fichero ¿qué ocurre?

e) En el INSTANTE B, ¿qué dígito se habrá escrito mediante la última llamada `write`?

f) Si tras haber hecho los open, otro proceso quiere borrar el fichero `f.txt`, ¿qué ocurre?

g) En el INSTANTE C, ¿cuántos bytes se habrán leído del fichero en esa última operación?

h) Supongamos que `f.txt` ya existiese (siendo del mismo usuario y con permisos de lectura y escritura).¿Qué ocurriría si fuese un enlace?

i) Si después del INSTANTE C se ejecuta:

```
if(fork())
close(fd1);
```

¿Qué ocurre?

SOLUCIÓN

a) El fichero se intenta crear con protección 0660, pero antes de ello se ha definido una máscara de protección 0022. Esta máscara representa los bits que quedan excluidos (la operación lógica aplicable sería 0660 & ~0022). Los permisos resultantes son 0640.

b) Los descriptores 3 y 4 estarían asignados, apuntando a dos entradas diferentes de la tabla intermedia, pero ambas apuntarían a la misma copia del i-nodo. Cada `open`, siempre implica una nueva entrada en la tabla intermedia. Hay que recordar que esta tabla incluye la posición del puntero de lectura/escritura. Por lo tanto, para cada descriptor de fichero tendríamos.

c) Hay que tener en cuenta que el primer `write`, aunque tiene como buffer de partida una cadena de 10 elementos sólo se escriben los 6 primeros "987654". Después de esa operación el fichero estará al final de esa cadena. Sin embargo, se hace un *`lseek`* a la posición absoluta (`SEEK_SET`) 2. Es decir, sobre el "7" (se comienza por la posición, 0, luego 1 y luego 2). Y luego se lee un byte, por lo tanto, el carácter leído es el "7".

d) Una escritura posterior haría crecer el fichero hasta ese tamaño rellenándose el espacio extra con el carácter '\0'. Si no se hiciera esa escritura, y sólo el `lseek`, el fichero no crecería. Hay que resaltar que para hacer crecer un fichero no hace falta la opción `O_APPEND`, puesto que esta opción sólo implica que el puntero de lectura/escritura se coloque de partida al final del fichero. Con o sin esa opción el fichero puede crecer.

e) Como se ha comentado en el tercer apartado, el fichero contiene la cadena "987654". La llamada a `lseek` sobre el descriptor `fd2` se coloca en la antepenúltima posición (`SEEK_END` -2). `SEEK_END` es el final del fichero, y retroceder dos posiciones coloca el puntero sobre el dígito "5". Dicho dígito se lee y se escribe de nuevo en el fichero, dejando su contenido como "987655". El carácter escrito es el "5".

f) Curiosamente, el fichero se ha borrado, pero permanece abierto y sus datos accesibles hasta que se haga un `close` del último descriptor que lo manipule. El borrado de ficheros no interfiere con los descriptores ya abiertos que lo recorren. Lo que se hace es eliminar el nombre de dicha entrada en el directorio que la contiene, el contenido del fichero seguirá existiendo pero no estará accesible para otros procesos debido a que su nombre se ha eliminado. En cuanto se haga el último `close`, todo el espacio reservado se eliminará.

g) Recordemos en que posición dejamos el descriptor de `fd1`. Dicho descriptor tenía posicionado su puntero de lectura/escritura en la posición inmediatamente posterior al "7" que había leído. Es importante recalcar que las operaciones realizadas con el descriptor `fd2` no han afectado a este puntero (cada `open` implica un puntero independiente). Así pues desde la posición actual del puntero de `fd1` se invoca una lectura de 64 bytes. La llamada `read` intenta proporcionará todos los bytes solicitados (64 bytes), pero al no haber suficientes datos se leen únicamente los disponibles, es decir "655", un total de 3 bytes.

h) Si `f.txt` es un enlace (de cualquier tipo) a un fichero, el contenido se modificaría para cualquier otro enlace que lo apunte. Los parámetros de la llamada `open`, `O_CREAT|O_RDWR|O_TRUNC` indican, respectivamente, que si no existe se cree, que se abra con permisos de lectura/escritura y, por último, que si había algún contenido éste se trunca. Este último argumento no quiere decir que el fichero se borre y se cree otro, indica que se elimine su contenido. Al conservarse el fichero, el i-nodo, tanto los enlaces físicos como los simbólicos seguirían compartiendo contenido (cada uno por medio de la redirección correspondiente).

i) Se crea un proceso hijo y el proceso padre cierra el descriptor 3, que permanecería abierto en el hijo.

Problema 2.18 (junio 2007)

*Sea el programa `TT.c`, al que le faltan las líneas 21 a 51 y al que se le han tachado ciertos textos con la notación XXX**##**XXX, siendo ## es el número de línea donde están.*

```
___1 /* TT.c */
___2 #include <fcntl.h>
___3 #include <stdio.h>
___4 #include <stdlib.h>
___5 #include <unistd.h>
___6 #define SIZE 1024
___7
___8 int main(int argc, char *argv[])
___9 {
__10     int pp[2], fd;
__11     char buff[SIZE];
__12     int ret;
__13
__14     /* Comprobar Argumentos */
__15     if (!argv[1]) {
__16         fprintf(stderr, "USO: %s mandato [args...]\n", argv[0]);
__17         exit(1);
__18     }
__19
__20     /* Almacenar Copia de Entrada Estándar */
....
....
__52
__53     /* Almacenar Copia de Salida Estándar */
__54     ret = pipe(pp);
__55     if (ret < 0) {
__56         perror("pipe");
__57         exit(1);
__58     }
__59     switch(fork()) {
__60     case -1:
__61         perror("fork");
__62         exit(1);
__63     case  0:
__64         close(XXX64XXX);
__65         fd = creat("salida", 0666);
__66         if (fd < 0) {
__67             perror("salida");
__68             exit(1);
__69         }
__70         while((ret = read(pp[0], buff, XXX70XXX)) > 0) {
__71             write(fd, buff, XXX71XXX);
__72             write(1, buff, XXX72XXX);
__73         }
__74         if (ret < 0) {
__75             perror("salida");
__76             exit(1);
__77         }
__78         return 0;
__79     default:
__80         close(XXX80XXX);
__81         dup(XXX81XXX);
__82         close(pp[0]);
__83         close(pp[1]);
__84     }
```

```
__85
__86      /* Ejecutar Mandato */
__87      execvp(argv[1], &argv[1]);
__88      perror(argv[1]);
__89      exit(1);
__90
__91      return 0;
__92 }
```

Dado el código presentado (sin considerar las líneas 21 a 51 que faltan) se pide que deduzca cuál es la utilidad del programa. Para ello:

- ***a)*** *Suponga que el fichero* `dias.txt` *contiene, uno por línea, los nombres de los días de la semana. Estudie el siguiente ejemplo de uso del mandato* `TT`*.*

  ```
  $ ./TT sort < dias.txt
  ```

  Identifique claramente qué información llegará por la entrada estándar, qué información producirá por la salida estándar y cuál será la información finalmente contenida en los ficheros auxiliares que se hayan creado.
- ***b)*** *Dibuje un diagrama de procesos comunicados, donde quede clara la jerarquía de los procesos, los descriptores que utilizan y para qué los utilizan.*
- ***c)*** *Explique con sus propias palabras para qué sirve el mandato* `TT`*.*
- ***d)*** *Razone y explique cuál sería el valor correcto del texto tachado en las líneas 70 a 72.*
- ***e)*** *Razone y explique cuál sería el valor correcto del texto tachado en las líneas 80 y 81.*
- ***f)*** *Razone y explique cuál sería el valor correcto del texto tachado en la línea 64. ¿Qué le sucedería a la ejecución del mandato si se hubiese omitido esta llamada a* `close`*?*

Dado el código expuesto, no es difícil imaginar el código de las líneas 21 a 51 que faltan, ya que su misión es simétrica al de las líneas 54 a 84.

- ***g)*** *Codifique las líneas que faltan. No se acelere. Preste especial atención a las diferencias necesarias y subráyelas.*
- ***h)*** *En vista al nuevo código que ha escrito, ¿sería posible eliminar las líneas 54 a 58?, es decir, ¿serían necesarias una o dos llamadas a* `pipe`*?, y en este último caso, ¿bastaría con la declaración de* `pp` *en la línea 10 o haría falta una segunda?*
- ***i)*** *Conteste nuevamente al apartado* ***a)****.*
- ***j)*** *Conteste nuevamente al apartado* ***b)****.*

Solución

```
___1 /*·TT.c·*/
___2 #include·<fcntl.h>
___3 #include·<stdio.h>
___4 #include·<stdlib.h>
___5 #include·<unistd.h>
___6 #define·SIZE·1024
___7
___8 int·main(int·argc,·char *argv[])
___9 {
__10     int·pp[2],·fd;
__11     char·buff[SIZE];
__12     int·ret;
__13
__14     /*·Comprobar·Argumentos·*/
__15     if·(!argv[1])·{
__16         fprintf(stderr,·"USO:·%s·mandato·[args...]\n",·argv[0]);
__17         exit(1);
__18     }
__19
__20     /*·Almacenar·Copia·de·Entrada·Estándar·*/
```

```
__21      ret = pipe(pp);
__22      if (ret < 0) {
__23          perror("pipe");
__24          exit(1);
__25      }
__26      switch(fork()) {
__27      case -1:
__28          perror("fork");
__29          exit(1);
__30      case 0:
__31          close(pp[0]);
__32          fd = creat("entrada", 0666);
__33          if (fd < 0) {
__34              perror("entrada");
__35              exit(1);
__36          }
__37          while((ret = read(0, buff, SIZE)) > 0) {
__38              write(fd, buff, ret);
__39              write(pp[1], buff, ret);
__40          }
__41          if (ret < 0) {
__42              perror("entrada");
__43              exit(1);
__44          }
__45          return 0;
__46      default:
__47          close(0);
__48          dup(pp[0]);
__49          close(pp[0]);
__50          close(pp[1]);
__51      }
__52
__53      /* Almacenar Copia de Salida Estándar */
__54      ret = pipe(pp);
__55      if (ret < 0) {
__56          perror("pipe");
__57          exit(1);
__58      }
__59      switch(fork()) {
__60      case -1:
__61          perror("fork");
__62          exit(1);
__63      case 0:
__64          close(pp[1]);
__65          fd = creat("salida", 0666);
__66          if (fd < 0) {
__67              perror("salida");
__68              exit(1);
__69          }
__70          while((ret = read(pp[0], buff, SIZE)) > 0) {
__71              write(fd, buff, ret);
__72              write(1, buff, ret);
__73          }
__74          if (ret < 0) {
__75              perror("salida");
__76              exit(1);
__77          }
__78          return 0;
__79      default:
__80          close(1);
__81          dup(pp[1]);
```

```
__82          close(pp[0]);
__83          close(pp[1]);
__84      }
__85
__86      /*·Ejecutar·Mandato·*/
__87      execvp(argv[1],·&argv[1]);
__88      perror(argv[1]);
__89      exit(1);
__90
__91      return·0;
__92  }
```

a) El ejemplo propuesto de uso del mandato: `$ ./TT sort < dias.txt`
sugiere la puesta en ejecución, desde el *shell* que nos muestra el *prompt* `$`, del ejecutable `TT` (que se supone que es el resultado de la compilación correcta del programa `TT.c`), que se encuentra en el directorio actual de trabajo (de ahí la ruta indicada `./`), pasándole un único argumento (`sort`) y asociándole la entrada estándar al fichero `dias.txt` mediante la notación "<". Todo esto lo realiza automáticamente el *shell*. La salida estándar y la salida estándar de error no han sido redirigidas, luego permanecen asociadas a donde estuvieran en el momento de introducir esta línea de mandatos. Digamos, por ejemplo, que permanecen asociadas al terminal desde el que un usuario ha introducido el mandato.

Viendo el código de `TT.c` se observa la creación de una tubería (pipe) y su uso para la comunicación del proceso original (que llamaremos padre) con otro proceso hijo creado al efecto. El proceso padre está destinado a ejecutar el mandato pasado como primer argumento (en este caso concreto, el mandato `sort`) parametrizado con el resto de argumentos (en este caso concreto no hay tales). Estos mandatos (denominados genéricamente mandatos estándar) utilizan los descriptores estándar, y el proceso hijo lee de la tubería. De ello debemos deducir que el proceso padre en las líneas 81 y 82 redirige su salida estándar a la tubería.

Así pues, la información que el conjunto de estos procesos presentará por la salida estándar, será la que presente el proceso hijo (con la llamada de la línea 72) que quedará simultáneamente registrada en el fichero auxiliar `salida`, y que, en definitiva, será trascripción de la que el proceso padre inyecte por el otro extremo de la tubería. Esto es, la salida producida por el mandato indicado como argumentos a `TT` por su salida estándar.

En resumen: La información que llegará por la entrada estándar será el contenido del fichero `dias.txt`, esto es, los días de la semana, uno por línea. La información producida por la salida estándar serán, uno por línea, los días de la semana, pero ordenados alfabéticamente (esto es lo que `sort` realiza). Y la información finalmente registrada en el archivo `salida`, será copia de la producida por la salida estándar.

b) Dos procesos: padre he hijo, como muestra la figura 2.11.

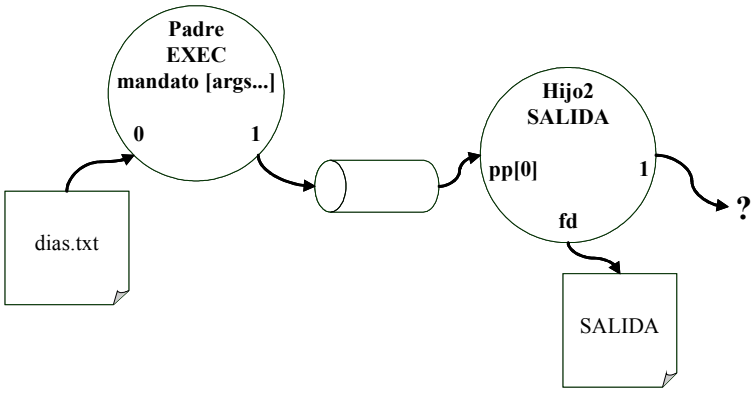


Figura 2.11:

El padre tiene su entrada estándar asociada al fichero `dias.txt`, su salida estándar asociada a una tubería y ejecuta el mandato `sort`.

El proceso hijo lee del otro extremo de la tubería, a través del descriptor `pp[0]`, y lo que lee lo escribe por el descriptor `fd` (asociado al fichero `salida` creado al efecto) y lo repite por la salida estándar.

c) El mandato `tee` de UNIX (`man 2 tee`) funciona al igual que una tubería en forma de T, y de ahí le viene el nombre. Copia lo que lee de su entrada estándar sobre su salida estándar y además en otro archivo indicado como argumento.

De forma semejante (pero no idéntica) el mandato presentado como TT tiene como objetivo final, ejecutar el conjunto mandato+argumentos indicado en sus argumentos, pero añadiendo una T a la entrada y otra T a la salida de dicho mandato.

Un proceso central, ejecutará el mandato+argumentos indicado. Dos procesos auxiliares, hijos del primero, filtrarán la entrada y salida estándar respectivamente, registrando copia de dichas informaciones sobre sendos archivos auxiliares de nombre prefijado: `entrada` y `salida`.

d) El proceso hijo intenta procesar la información que lee de la tubería en cantidades de tamaño razonable. En la línea 70, intenta leer sobre `buff SIZE` bytes cada vez. Por tratarse de una tubería, se leerá la información que haya disponible en cada momento hasta el límite indicado (`SIZE`). Es por eso que las dos operaciones de escritura siguientes deben transferir sólo la información que fue realmente leída de la tubería y ahora está en nuestro *buffer*. Dicha cantidad de información son `ret` bytes. El tercer parámetro correcto para las llamadas `write` de las líneas 71 y 72 será pues la variable `ret`.

```
__70          while((ret·=·read(pp[0],·buff,·SIZE))·>·0)·{
__71              write(fd,·buff,·ret);
__72              write(1,·buff,·ret);
```

e) Por razones que se han expuesto detalladamente en el apartado **a**, el proceso padre está destinado a enviar lo que emita por su salida estándar a la tubería que hemos creado en la línea 54. Lo que debe hacer pues es redirigir su salida estándar al extremo de escritura de la tubería, y eso es exactamente lo que hacen las dos instrucciones siguientes:

```
__80          close(1);
__81          dup(pp[1]);
```

f) Una regla de buena programación en UNIX dice que deben cerrarse todos los descriptores (a excepción de los estándar) que no vayan a utilizarse. Pero en este caso hay otra razón de mayor peso.

La línea 64 se corresponde con el código que ejecuta el proceso hijo, que lee de la tubería y lo copia sobre `salida` y sobre la salida estándar. Esta copia se realiza mientras se lea información de la tubería, y, por lo tanto, terminará cuando la llamada de lectura devuelva un valor 0. Como se ha estudiado en teoría, para que una llamada de lectura sobre una tubería devuelva 0 bytes leídos, es preciso que se den simultáneamente dos circunstancias: que no haya datos en la tubería (de haberlos la llamada los leería) y que no los vaya ha haber en el futuro, y la única forma de garantizar esto último es que no quede abierto ningún descriptor asociado al extremo de escritura de la tubería.

```
__64          close(pp[1]);
```

Es un fallo común al comunicar procesos por tuberías dejar sin cerrar un descriptor de escritura a una tubería, con el efecto de dejar permanentemente colgado al proceso lector en la llamada de lectura del otro extremo de la tubería.

g) Anteriormente se ha mostrado el código completo del mandato `TT`. A continuación se muestran, de manera comparada, sólo los cambios entre las correspondientes líneas 21 a 51 y 54 a 84.

Es necesario entender que, en este caso, el sentido de la comunicación a través de la tubería es del hijo hacia el padre.

- Se cierran los descriptores que no se utilizan.

```
ENTRADA __31          close(pp[0]);          no va ha leer de la tubería
SALIDA  __64          close(pp[1]);          no va ha escribir en la tubería
```

- El nombre del fichero auxiliar de registro.

```
ENTRADA __32    __34    __42        "entrada"
SALIDA  __65    __67    __75        "salida"
```

- El sentido de la comunicación a través de la tubería.

```
ENTRADA __37          read(0,...             del hijo al padre
ENTRADA __38                write(fd,...
ENTRADA __39                write(pp[1],...
SALIDA  __70          read(pp[0],...         del padre al hijo
SALIDA  __71                write(fd,...
SALIDA  __72                write(1,...
```

- La redirección. El hijo utiliza los descriptores de manera explícita, pero el padre, al ir a ejecutar mandatos, debe redirigir sus descriptores estándar.

```
ENTRADA __47        close(0);          entrada estándar
ENTRADA __48        dup(pp[0]);              de la tubería
SALIDA  __80        close(1);          salida estándar
SALIDA  __81        dup(pp[1]);              a la tubería
```

h) No es posible eliminar las líneas 54 a 58. Si, son necesarias dos llamadas a `pipe`, para crear las dos tuberías que comunican al priper hijo (`entrada`) con el padre y a este con su segundo hijo (`salida`).

Por otro lado, y dado el código, basta con la declaración de `pp` en la línea 10 y no haría falta una segunda. Esta variable no es distinta las otras de este mismo programa, que hemos venido reutilizando. La variable `pp` está destinada a contener, temporalmente, el valor de los descriptores asociados a uno y otro extremo de las tuberías. Primero de una y luego de la otra.

i) Ahora, tenemos implementadas las dos Tes, no sólo la de salida, sino también la de entrada. La ejecución del mandato sugerido, será prácticamente la misma. La única diferencia respecto al apartado **a)** es que ahora tenemos un primer hijo destinado a filtrar la entrada.

Así pues, la información que el conjunto de estos procesos tomará de la entrada estándar, será la que tome el primer proceso hijo (con la llamada de la línea 37) que será simultáneamente registrada en el fichero auxiliar `entrada`, y será inyectada por el extremo de la tubería para que le llegue al proceso padre como entrada estándar.

En resumen: La información que llegará por la entrada estándar será el contenido del fichero `dias.txt`, esto es, los días de la semana, uno por línea. Y la información registrada en el archivo `entrada`, será copia de la recibida por la entrada estándar. La información producida por la salida estándar será, uno por línea, los días de la semana, pero ordenados alfabéticamente (esto es lo que `sort` realiza). Y la información finalmente registrada en el archivo `salida`, será copia de la producida por la salida estándar.

j) Tres procesos: padre y dos hijos, `ENTRADA` y `SALIDA`., como se muestra en la figura 2.12.

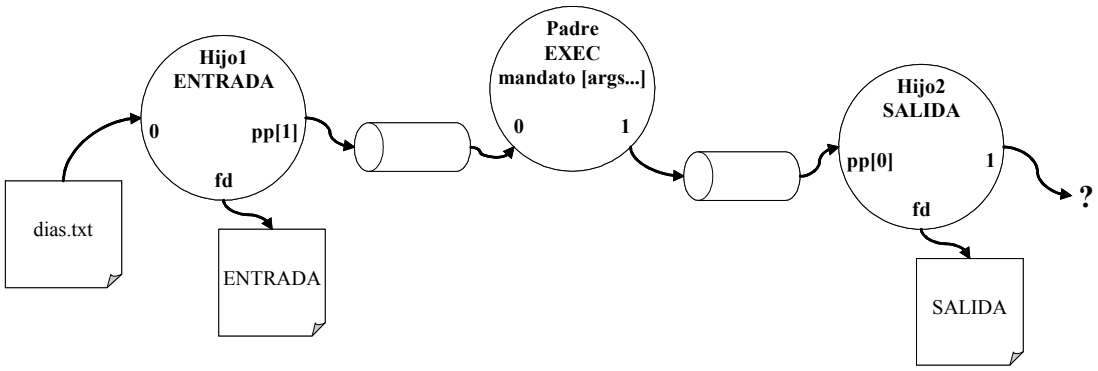


Figura 2.12

El padre tiene su entrada estándar asociada a una tubería que lo comunica con un proceso hijo denominado `ENTRADA`. Así mismo, su salida estándar está asociada a una tubería que lo comunica con un proceso hijo denominado `SALIDA`. El proceso padre ejecuta el mandato indicado con los argumentos indicados: `sort`, sin argumentos.

El proceso hijo `ENTRADA` lee de la entrada estándar que tiene asociada al fichero `dias.txt`, y lo que lee lo escribe por el descriptor `fd` (asociado al fichero `entrada` creado al efecto) y lo repite por la tubería que lo comunica con su padre a través del descriptor `pp[1]`.

El proceso hijo SALIDA lee de la tubería que lo comunica con su padre a través del descriptor pp[0], y lo que lee lo escribe por el descriptor fd (asociado al fichero salida creado al efecto) y lo repite por la salida estándar.

Problema 2.19 (abril 2008)

En el directorio `/home` de un sistema de ficheros A se ha montado el sistema de ficheros B, sin enmascarar ningún permiso. ***Parte del árbol*** *de nombres resultante se muestra en la figura 2.13, en base a una salida parcial de varios mandatos `ls` (se recuerda que el segundo dato del `ls` es el número de enlaces).*

Las máscaras de creación de ficheros y los grupos de algunos usuarios son los siguientes:

- *Usuario* `pepe`*, máscara = 022 y grupo div1*
- *Usuario* `macu`*, máscara = 027 y grupo div1*
- *Usuario* `mari`*, máscara = 077 y grupo div2*

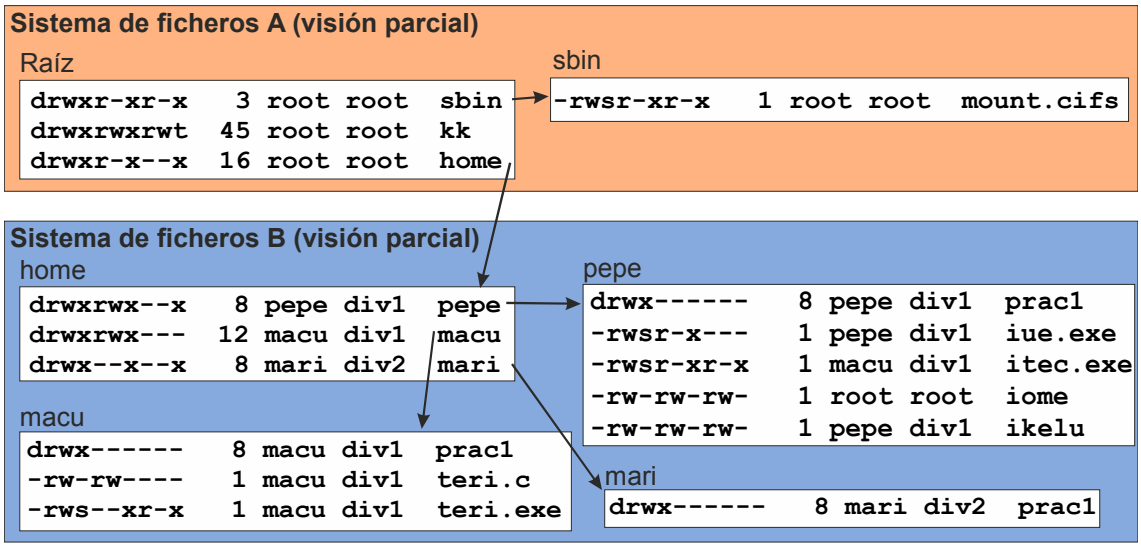


Figura 2.13

El programa `itec.exe` *contiene el siguiente esquema de código*

```
fd = open("/home/macu/teri.c", O_RDWR);
n = fork();
```

a) El usuario `mari` *crea un nuevo programa* `xx` *que incluye la siguiente línea:*

```
m = execl("/home/pepe/itec.exe", "itec.exe", NULL);
```

Teniendo en cuenta que `m` *es una variable de* `xx` *y* `fd` *de* `itec.exe`*, y que no hay errores inesperados, indicar el valor de* `m` *y el de* `fd` *cuando* `mari` *ejecuta el programa* `xx`*..*

b) Los usuarios `pepe` *y* `macu` *lanzan al tiempo la ejecución del programa* `itec.exe`*. ¿Cuál sería el máximo número de referencias (también llamados* `ndups`*) y de* `nopens` *que podrían llegar a alcanzarse?*

c) El programa `iue.exe` *contiene la siguiente línea de código:*

```
n = creat("/home/pepe/prac1/siet.c", 0666);
```

El usuario `macu` *intenta ejecutar dicho programa desde su home, indicar si se crea el fichero y, en su caso, indicar el dueño y los derechos.*

Ahora, el programa que ha lanzado el usuario `pepe` *ejecuta la siguiente línea de código:*

```
fd = open("/home/pepe/ikelu", O_RDWR|O_CREAT|O_TRUNC,0640);
```

obteniendo `fd= 5`*. Seguidamente, el programa crea dos hijos H1 y H2. Inmediatamente, H2 hace un* `exec`*.*

Al mismo tiempo, pero cuando `pepe` *ya ha ejecutado el* `open`*, el usuario* `mari` *ejecuta otro programa (proceso M) que incluye:*

```
fd = open("/home/pepe/ikelu", O_RDWR);
```

recibiendo fd = 3.

A partir de este punto se produce la siguiente secuencia de ejecuciones

	Proceso	***Servicio***
1	*H1*	`write(5, "11111111111111111111111111111", 23);`
2	*H2*	`write(5, "22222222222222222222222222222", 26);`
3	*M*	`write(1, "3333", 4);`
4	*H1*	`lseek(5, 17, SEEK_END);`
5	*H2*	`write(5, "9999", 4);`

6	M	`lseek(3, 2, SEEK_SET);`
7	H1	`write(5, "4444", 4);`
8	M	`write(3, "45", 2);`
9	H2	`lseek(5, 4, SEEK_SET);`
10	H1	`write(5, "8901", 4);`
11	H1	`close(5);`
12	H2	`close(5);`
13	M	`Close(3);`

Aplicando la política de coutilización de UNIX:

d) Indicar el tamaño real del fichero al final de esta secuencia.

e) Indicar el contenido del fichero.

Solución

a) Tanto `home` como `pepe` tienen derechos de visita para el mundo, además, el fichero `itec.exe` tiene derechos de ejecución para el mundo, por lo que `mari` lo puede ejecutar. Al tratarse de un `exec` exitoso no se devuelve ningún valor, por lo que `m` no llega a cambiar de valor.

Observamos que el programa `itec.exe` tiene activo el SUID, por lo que ejecutará con la identidad de su dueño, es decir, `macu`. Como `macu` es el dueño, puede visitar su directorio y llegar al fichero `/home/macu/teri.c`, por tanto, lo podrá abrir para escritura y lectura. Esto significa que `fd` recibirá el valor del primer descriptor libre, por ejemplo, 3.

b) Tanto `pepe` como `macu` pueden ejecutar el programa `itec.exe`, que ejecutará con identidad efectiva `macu` en ambos casos, por lo que cada ejecución abrirá el fichero `/home/macu/teri.c`. Esto significa que `nopens` = 2 (suponiendo que el fichero no estaba abierto con anterioridad). Por otro lado, aparecerán dos entradas en la tabla intermedia, cada una de las cuales llegará a tener dos referencias, dado que cada ejecución producirá un proceso hijo.

c) El usuario `macu` tiene derechos de ejecución de `iue.exe`. Además, dicho fichero tiene activo su SUID, por lo que ejecuta con la identidad efectiva de `pepe`. Se aplica la máscara de creación de ficheros que tiene el proceso, por tanto, la de `macu` que es 027. Luego, se crea el fichero con dueño `pepe` y con derechos `rw-r-----`

d) El tamaño del fichero será de 23 + 26 + 17 + 4 + 4 = 74 bytes.

e) El contenido del fichero será el siguiente, de acuerdo a la línea ejecutada:

```
1     11111111111111111111
2     11111111111111111111122222222222222222222222
3     (se escribe en otro fichero)
5     111111111111111111111122222222222222222222222200000000000000009999
7     111111111111111111111122222222222222222222222200000000000000099994444
8     114511111111111111111122222222222222222222222200000000000000099994444
10    114589011111111111111122222222222222222222222200000000000000099994444
```

Problema 2.20 (mayo 2009)

Sea un sistema de ficheros UNIX con las siguientes características:

- *Bloque de 1 KiB y agrupación de 1 bloque.*
- *Nodos i ocupados del 2 al 29, el resto libres.*
- *Bloques ocupados del 1 al 79, el resto libres.*
- *Para asignar un bloque o un nodo i se selecciona el primero libre.*
- *Entradas libres de la tabla intermedia: a partir de la 11.*
- *El directorio "/" (raíz), pertenece al usuario y grupo root y tiene los permisos: rwx rwx --x*
- *El directorio "/home", pertenece al usuario y grupo root, tiene los permisos: rwx rwx rwx y su nodo i es el 14.*
- *El fichero "/home/user2/prog" tiene: UID = 400, GID = 7 y permisos = 06755.*

- *Enteros de 4 bytes.*

En el instante inicial existen dos procesos PA y PB cuyas identidades y descriptores de fichero se indican en la figura 2.14.

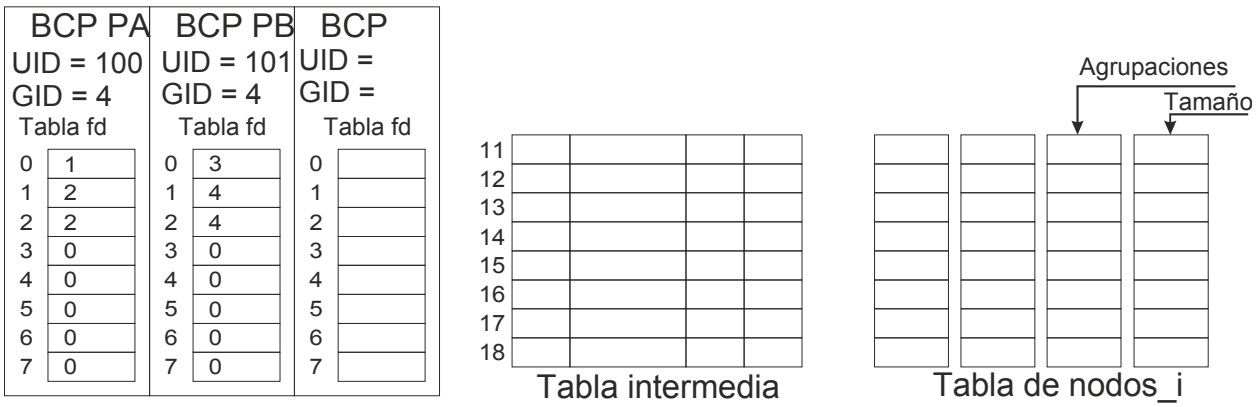


Figura 2.14

Se ejecuta la secuencia siguiente:

Proceso	**código ejecutado**

PA	`umask(0023);`
	`mkdir("/home/user1", 0753);  // El directorio no existe inicialmente.`
	`fd=creat("/home/user1/file1.txt", 0666);`
	`for (i=0;i<100;i++)`
	`  n=write(fd, "0123456789ABCDEFGHJK", 20);`
	`pid=fork();  // Suponer que no falla. Llamar PC al hijo.`

PC	`lseek(fd, 3, SEEK_SET);` `n=write(fd, "xter",4);`
PB	`fd2=open("/home/user1/file1.txt", O_RDWR);`
Punto 1 de la secuencia de ejecución.	
PB	`lseek(fd2, 5*1024, SEEK_CUR);` `n=write(fd2, "ProcesoB",8);`
Punto 2 de la secuencia de ejecución.	
PB	`execl("/home/user2/prog", "/home/user2/prog", NULL);` `exit (1);`

PB	`(nuevo código)`
	`fd3=creat("/home/user1/file2.txt", 0666);  // El fichero no existe inicialmente.`
	`n3=access("/home/user1", R_OK);`

Punto 3 de la secuencia de ejecución.	

Se pide:

a) Determinar los permisos con los que se crea el fichero "/home/user1/file1.txt".

b) Completar la figura 2.14 para el punto 1 de la secuencia de ejecución. Deberá indicar los títulos de las columnas de la tabla intermedia y de nodos i que faltan.

c) Indicar el contenido de los bloques que se modifican por la secuencia anterior en su ejecución hasta el punto 1.Usar para los bloques de directorio el siguiente formato: nombre – nodo i; nombre – nodo i; ... Limitar el contenido presentado a los primeros caracteres de cada bloque seguido de algunos puntos, si éste tiene un contenido mayor. Indicar el comienzo de la basura poniendo interrogaciones ???.

d) En el punto 2 de la secuencia de ejecución indicar las agrupaciones asignadas al fichero "/home/user1/file1.txt"

e) En el punto 3 de la secuencia de ejecución indicar los valores devueltos en las variables fd3 y n3.

Solución

a) Como el directorio `/home` tienen permisos de escritura tanto para el dueño, como para el grupo como para el mundo, todos los usuarios pueden escribir en él, por lo que el `mkdir` se ejecutará con éxito. El fichero "/home/user1/file1.txt" se crea con los permisos: rw- r-- r-- .

b) La creación del directorio `/home/user1` sopone la creación de un fichero tipo directorio, al que se le asignará el nodo i nº 30. El proceso PB no puede abrir el fichero "/home/user1/file1.txt", por lo que solamente lo tienen abierto los procesos PA y PC. La figura 2.15 muestra la situación de las tablas.

Figura 2.15

En el supuesto de que PB hubiese podido abrir l fichero "/home/user1/file1.txt" las tablas quedarían como se indica en la figura 2.16.

Figura 2.16

c) El contenido de los bloques es el siguiente:

Bloque	**Contenido**
80	. – 30; .. – 14; file1.txt – 31;?????
81	012xter789ABCDEFGHJK0123456789ABCDEFGHJK0123456789ABCDEFGHJK…..
82	456789ABCDEFGHJK0123456789ABCDEFGHJK0123456789ABCDEFGHJK…....???????

El bloque 82 tendrá los últimos 48 bytes con basura, dado que no han sido escritos.

d) Igual que para el apartado b) consideraremos dos situaciones, la real consistente en que el proceso PB no consigue abrir el fichero "/home/user1/file1.txt" y la hipotética en la que sí consigue abrir el fichero.

En el primer supuesto, el fichero no es modificado por el proceso PB, por lo que sus agrupaciones son la 81 y 82, cono se indica en el apartado c). Hay que tener en cuenta que la agrupación 80 es ocupada por el directorio "/home/user1".

En el segundo supuesto, las agrupaciones ocupadas por el fichero "/home/user1/file1.txt" son las 81, 82 y 83. Además el fichero tiene un hueco desde la posición 2.000 a la 5*1.024 = 5.120, ocupándose las posiciones 5.120 a 5.120 por el texto "ProcesoB". Las agrupaciones 81 y 82 dan soporte a los 2.000 caracteres escritos, debiendo ser re-

llenados con ceros al crearse el hueco los 48 bytes no escritos previamente. Los bytes 2.048 a 5.119 (que corresponden a hueco) no tienen agrupación asignada dado que no tiene sentido asignar unas agrupaciones que tendríamos, además, que rellenar con ceros. Finalmente, la agrupación 83 dará soporte a los 8 caracteres "ProcesoB", quedando el resto fuera del tamaño del fichero.

e) El fichero "/home/user1/file2.txt" no puede ser creado, puesto que la identidad efectiva del proceso PB después del `exec` es UID = 400, GID = 7, que no tiene permisos de escritura en "/home/user1". Por lo tando, fd3 = -1

Dado que el servicio `access` utiliza la identidad real del proceso PB, que es UID = 101, GID = 4, por lo que si tiene permisos de lectura. Por tanto n3 = 0.

Problema 2.21 (abril 2010)

Sean los fragmentos de programas de la tabla adjunta, dichos programas se ejecutan de forma entrelazada de acuerdo a la secuencia establecida en dicha tabla. Supondremos que el fichero fich existe y tiene un tamaño de 43 bytes, que el tamaño del bloque es de 1 KiB y que ninguno de los servicios produce error.

Programa 1	***Programa 2***	***Orden ejecución (línea)***
`fd = open(”/us/pe/fich”, O_RDWR);`		1
`write(fd, ”145243”, 4);`		2
	`fd = open(”/us/pe/fich”, O_WRONLY);`	3
	`lseek(fd, 3, SEEK_CUR);`	4
`lseek(fd, 50, SEEK_SET);`		5
	`write(fd, ”7824”, 4);`	6
`write(fd, ”123434”, 5);`		7
`lseek(fd, -53, SEEK_CUR);`		8
`write(fd, ”86”, 2);`		9
	`close(fd);`	10

a) Indicar el contenido del fichero al final de la ejecución de ambos fragmentos de programa, línea 10.

b) Indicar el tamaño del fichero al final de la ejecución de ambos fragmentos de programa.

c) Suponiendo que el nodo_i raíz ya está en memoria, que cada directorio ocupa un bloque y que no existe cache de bloques, determinar el número de accesos a disco que se producen hasta la ejecución de la línea 2 inclusive.

d) Repetir la pregunta c para el caso de que exista cache de bloques, considerando la ejecución hasta la línea 6 inclusive y que la escritura es write-through.

e) Representar de forma gráfica las estructuras de información afectadas por dichos programas tal y como quedan el final de la ejecución de la línea 10 inclusive.

Solución

a) Los contenidos son los siguientes, representando los valores originales del fichero con ? y todo ello expresado como cadena de caracteres.

Línea 2: **1452**??
Línea 6: 145**7824**????????????????????????????????????
Línea 7: 1457824????????????????????????????????????**000000012343**
Línea 9: 14**86**824????????????????????????????????????000000012343

b) El tamaño final del fichero lo obtenemos de la respuesta anterior, siendo de 55 B.

c) Los accesos son los siguientes:

- Lectura del directorio raíz (un bloque). Línea 1
- Lectura del nodo_i del directorio us (un bloque). Línea 1
- Lectura del directorio us (un bloque). Línea 1

- Lectura del nodo_i del directorio pe (un bloque). Línea 1
- Lectura del directorio pe (un bloque). Línea 1
- Lectura del nodo_i del fichero fich (un bloque). Línea 1
- Lectura del primer bloque del fichero fich. Línea 2
- Escritura del primer bloque del fichero fich. Línea 2

Total 8 accesos al disco. El nodo_i queda modificado, puesto que se modifican los instantes de acceso y escritura. Esta modificación se hace primero en la copia que está en memoria y más adelante se lleva al disco.

d) Los accesos son los siguientes:

- Lectura del directorio raíz (un bloque). Línea 1
- Lectura del nodo_i del directorio us (un bloque). Línea 1
- Lectura del directorio us (un bloque). Línea 1
- Lectura del nodo_i del directorio pe (un bloque). Línea 1
- Lectura del directorio pe (un bloque). Línea 1
- Lectura del nodo_i del fichero fich (un bloque). Línea 1
- Lectura del primer bloque del fichero fich. Línea 2
- Escritura del primer bloque del fichero fich. Línea 2
- Escritura del primer bloque del fichero fich. Línea 6

Total 9 accesos al disco. Es de destacar que la línea 3 ya encuentra todos los bloques en la cache y la línea 6 encuentra en la cache el valor escrito por la línea 2, por lo que no necesita leerlo. Igual que en el caso anterior, más adelante habrá que actualizar el nodo_i.

e) Las estructuras quedan como se indica en la figura 2.17. Se ha supuesto que los procesos que ejecutan los programas dados solamente tienen abiertos los descriptores estándar y que las líneas 5 y 6 de la tabla intermedia están libres. También hemos considerado que el nodo_i del fichero es el 33. La línea 6 queda ocupada por el proceso del programa 2, pero despúes del close queda libre, lo que se refleja puesto que el número de referancias es 0:

En la tabla de nodos_i también quedarán modificados los instantes de acceso y modificación del fichero.

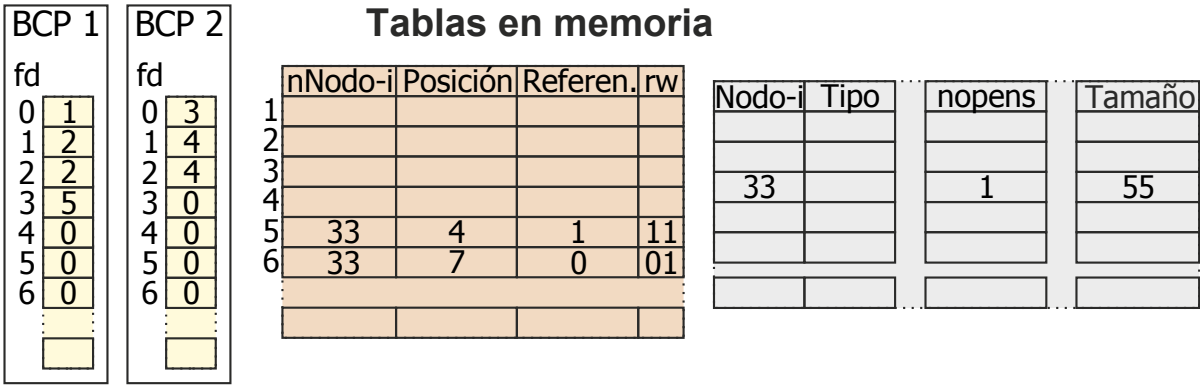


Figura 2.17

Dado que no se asignan bloques nuevos ni nodos_i, los mapas de bits del disco no quedan modificados.

Problema 2.22 (junio 2010)

Debe usted implementar en lenguaje C y para UNIX el código fundamental del mandato `cyp` *(CopiaYPega) cuya utilidad es poder copiar una parte de un fichero, dadas su posición y longitud (de bytes a gigabytes), a otro fichero en otra posición dada. Realice el código más correcto y claro que pueda (NO se admitirá pseudocódigo), realice un tratamiento correcto de los errores y libere todos los recursos previamente reservados, cuando dejen de usarse.*

a) Siga estrictamente las siguientes instrucciones de diseño:

- *Implemente la función* `copia` *que copia* `size` *bytes del fichero abierto* `fdorg` *al fichero abierto* `fddst`*:*

```
copia(fdorg, size, fddst)
```

- *Haciendo uso de la función anterior, implemente la función* `copiaypega`*, que añade a la anterior la posición absoluta de origen y destino,* `offorg` *y* `offdst`*:*

```
copiaypega(fdorg, offorg, size, fddst, offdst)
```

- *Haciendo uso de la función anterior, implemente la función* `cyp` *que realiza la operación de* `copiaypega` *entre los ficheros* `origen` *y* `destino` *de nombre dado:*

```
cyp(origen, offorg, size, destino, offdst)
```

b) Considere la siguiente secuencia de mandatos y analice el comportamiento de `cyp`*:*

```
(1)  echo -n 0123456789 > A
(2)  echo -n abcdefghij > B
(3)  cyp A 7 5 B 2
(4)  cyp A 1 5 B 8
```

Razone cuál sería el contenido del fichero `B` *al final de los pasos (3) y (4) y porqué.*

c) Ahora, piense y rehaga sólo la parte imprescindible del código anterior, para realizar una implementación equivalente de `cyp` *pero mediante ficheros proyectados en memoria. Observe que conseguir el mismo comportamiento anterior implica una solución un poco más completa que la trivial.*

d) Razone:

- *¿Cuál de estas implementaciones será más rápida (en bytes copiados por unidad de tiempo)?*
- *¿Cuál de estas implementaciones será más apta para copiar cantidades muy grandes de datos (decenas de gigas) (considere un sistema de 32 bits)?*

e) Considere ahora la necesidad de poder ejecutar varios mandatos `cyp` *concurrentes haciendo copias cruzadas de información entre un conjunto de ficheros.*

- *Implemente el código necesario para asegurar que cada cyp sucede sin verse afectado por otros en ejecución simultánea.*
- *¿Podría suceder interbloqueo?, ¿cómo podría evitarse?*

Solución

a) Las funciones solicitadas serían:

```c
/* Copia size bytes
 * del fichero abierto fdorg
 *  al fichero abierto fddst.
 */
int copia(int fdorg, int size, int fddst)
{
    char buf[4096];
    int ret;
    do {
       ret = size;
       if (ret > sizeof(buf)) ret = sizeof(buf);
       ret = read (fdorg, buf, ret);
       if (ret <= 0) break;
       ret = write(fddst, buf, ret);
       if (ret <= 0) break;
       size -= ret;
    } while(size > 0);
    if (ret < 0) return -1;
    return size;
}
/* Copia size bytes
 * del fichero abierto fdorg en su posición absoluta offorg
 *  al fichero abierto fddst en su posición absoluta offdst.
 */
int copiaypega_rdwr(int fdorg, int offorg, int size, int fddst, int offdst)
{
    int ret;
    ret = lseek(fdorg, offorg, SEEK_SET);
    if (ret < 0) return -1;
    ret = lseek(fddst, offdst, SEEK_SET);
    if (ret < 0) return -1;
    return copia(fdorg, size, fddst);
}
/* Copia size bytes
 * del fichero de nombre org en su posición absoluta offorg
 *  al fichero de nombre dst en su posición absoluta offdst.
 */
```

```
int cyp(char * org, int offorg, int size, char * dst, int offdst)
{
    int fdorg = open(org, O_RDONLY);
    int fddst = open(dst, O_RDWR|O_CREAT, 0666);
    int ret = 0;
    if (fdorg < 0) ret = -1;
    if (fddst < 0) ret = -1;
    if (ret >= 0) ret = copiaypega_rdwr(fdorg, offorg, size, fddst, offdst);
    if (fdorg >= 0) close(fdorg);
    if (fddst >= 0) close(fddst);
    return ret;
}
```

b) El fichero `A` contiene los diez dígitos del 0 al 9. El fichero `B` contiene las 10 primeras letras minúsculas.

En el paso (3), al intentar leer más allá del tamaño del fichero `A`, 5 bytes desde el 7, cuando su tamaño son 10, sólo podremos leer los 3 que hay de la posición 7 en adelante "`789`". Estos datos son los que deberán ser copiados sobre `B` en su posición 2, luego `B` quedará como "`ab789fghij`".

En el paso (4), se leerán sin problemas los 5 caracteres contenidos en el fichero `A` desde su posición 1, estos son "`12345`". Estos datos se tratarán de escribir llegando más allá del tamaño actual del fichero `B`, que son 10 bytes. Al intentar escribir más allá de su tamaño, el fichero crecerá automáticamente. El contenido final de `B` será "`ab789-fgh12345`" y su tamaño final serán 13 bytes. Si hubiese quedado hueco entre el tamaño anterior del fichero y la posición de escritura, este hueco se rellenaría con el bytes nulos.

c) La versión basada en ficheros proyectados en memoria consistiría en implementar `copiaypega` como sigue (la implementación tiene en cuenta la realidad de que los offset deben estár alineados a página):

```
/* Copia por proyección en memoria size bytes
 * del fichero abierto fdorg en su posición absoluta offorg
 *  al fichero abierto fddst en su posición absoluta offdst.
 */
int copiaypega_mmap(int fdorg, int offorg, int size, int fddst, int offdst)
{
    int page = sysconf(_SC_PAGE_SIZE);
    /* Los offset deben estar alineados a página. */
    void * org = mmap(NULL, size+offorg%page, PROT_READ , MAP_SHARED, fdorg, offorg/page*page);
    void * dst = mmap(NULL, size+offdst%page, PROT_WRITE, MAP_SHARED, fddst, offdst/page*page);
    int ret = 0;
    offorg %= page;
    offdst %= page;
    if (org == MAP_FAILED) ret = -1;
    if (dst == MAP_FAILED) ret = -1;
    if (ret >= 0) memcpy(dst+offdst, org+offorg, size);
    if (org != MAP_FAILED) ret = munmap(org, size+offorg);
    if (dst != MAP_FAILED) ret = munmap(dst, size+offdst);
    return ret;
}
```

La simple proyección en memoria de los ficheros ofrecería un comportamiento distinto, por dos razones:

- Cuando el sistema de memoria virtual subyacente se basa en paginación, las zonas de memoria proyectadas se redondean a página. Esto implicaría en nuestro caso, poder leer bytes nulos al acceder más allá del tamaño real del fichero proyectado, por no ser múltiplo exacto del tamaño de página. Así pues, en el caso (3) anterior el resultado habría sido "`ab789__hij`", indicando con '`_`' el byte nulo.
- Por la misma razón, si el fichero no tiene un tamaño múltiplo del tamaño de página, será posible escribir más allá del tamaño real del fichero proyectado. Sin embargo, esto no se verá reflejado en el fichero asociado, ni aunque, como es lógico la proyección sea `MAP_SHARED`. El resultado pues del paso (4) anterior sería "`ab789__h12345___...`" en memoria, pero "`ab789__h12`" sobre `B`.

Así pues, para corregir estos dos efectos, debemos proceder a limitar el espacio de lectura y a ampliar el de escritura, a saber, habría que añadir la siguiente función a antes de llamar a la anterior en `cyp`.

```
/* Ajusta tamaños de copia y destino para tener
 * el mismo comportamiento proyectando que por E/S.
 */
int copiaypega_ajst(int fdorg, int offorg, int*sizeptr, int fddst, int offdst)
{
    struct stat st;
    int size;
    int ret;
    /* Corregir máximos datos de origen. */
```

```
    ret = fstat(fdorg, &st);
    if (ret < 0) return ret;
    size = st.st_size - offorg;
    if (*sizeptr > size)
       *sizeptr = (size >= 0) ? size : 0;
    /* Corregir tamaño final del destino. */
    ret = fstat(fddst, &st);
    if (ret < 0) return ret;
    size = offdst + *sizeptr;
    if (size > st.st_size)
       if (*sizeptr > 0)
           ret = ftruncate(fddst, size);
    return ret;
}
```

d) Respecto a la velocidad de las soluciones hay que notar que la solución basada en operaciones de E/S realiza más llamadas al sistema operativo que la basada en proyección y, además, exige copiar los datos al espacio de memoria del proceso de usuario (en el buffer `buf`). Por su lado, la solución basada en proyección requiere atender los fallos de página de las regiones proyectadas y, por razones de seguridad, limpiar las páginas nuevas que se asignen al fichero en el que se copia. La velocidad de cada solución depende mucho de la implementación del sistema operativo y de las características de la operación solicitada.

Sobre cuál de las soluciones es más apta para grandes cantidades de datos, está muy claro, si consideramos un sistema de 32 bits malamente podremos proyectar zonas de ficheros por decenas de gigas. Simplemente no caben en nuestro mapa de memoria. Esto se podría solucionar realizando proyecciones parciales. Por otro lado, la implementación basada en E/S no plantea ningún límite en el tamaño de la copia.

e) Estamos hablando de un mecanismo de sincronización adecuado para coordinar la coutilización de ficheros por parte de procesos pesados arrancados independientemente. La definición/adecuación está muy clara. Debemos usar cerrojos sobre ficheros. Además, estos permiten distinguir entre uso compartido y exclusivo, para accesos concurrentes de lectura o accesos exclusivos de escritura, respectivamente.

Por supuesto que, si no hacemos nada, podrían darse situaciones de interbloqueo al requerirse dos cerrojos por ejecución de `cyp`. La solución más cómoda para esto es establecer algún orden global que nos permita tomar todos los recursos que se requieran de manera ordenada. De este modo será imposible que aparezca un ciclo de posesión, reclamo de recursos.

```
/* Bloquea las zonas a leer/escribir,
 * para evitar la corrupción de los datos.
 * Aplica in orden global único para evitar interbloqueos.
 */
int copiaypega_lock(int fdorg, int offorg, int size, int fddst, int offdst)
{
    struct flock florg;
    struct flock fldst;
    struct stat  storg;
    struct stat  stdst;
    int ret;
    florg.l_whence = SEEK_SET;
    florg.l_start = offorg;
    florg.l_len = size;
    florg.l_pid = getpid();
    florg.l_type = F_RDLCK;
    fldst.l_whence = SEEK_SET;
    fldst.l_start = offdst;
    fldst.l_len = size;
    fldst.l_pid = getpid();
    fldst.l_type = F_WRLCK;
    ret = fstat(fdorg, &storg);
    if (ret < 0) return ret;
    ret = fstat(fddst, &stdst);
    if (ret < 0) return ret;
    if (storg.st_dev < stdst.st_dev) {
       ret = fcntl(fdorg, F_SETLKW, &florg);
       if (ret < 0) return ret;
       ret = fcntl(fddst, F_SETLKW, &fldst);
    } else if (storg.st_dev > stdst.st_dev) {
       ret = fcntl(fddst, F_SETLKW, &fldst);
       if (ret < 0) return ret;
       ret = fcntl(fdorg, F_SETLKW, &florg);
    } else {
       if (storg.st_ino < stdst.st_ino) {
           ret = fcntl(fdorg, F_SETLKW, &florg);
```

```
            if (ret < 0) return ret;
            ret = fcntl(fddst, F_SETLKW, &fldst);
        } else if (storg.st_ino > stdst.st_ino) {
            ret = fcntl(fddst, F_SETLKW, &fldst);
            if (ret < 0) return ret;
            ret = fcntl(fdorg, F_SETLKW, &florg);
        } else {
            if (offorg < offdst) {
                ret = fcntl(fdorg, F_SETLKW, &florg);
                if (ret < 0) return ret;
                ret = fcntl(fddst, F_SETLKW, &fldst);
            } else if (offorg > offdst) {
                ret = fcntl(fddst, F_SETLKW, &fldst);
                if (ret < 0) return ret;
                ret = fcntl(fdorg, F_SETLKW, &florg);
            } else {
                /* Mismo fichero, misma zona. Mala cosa */
                ret = -1;
            }
        }
    }
    return ret;
}
/* Desbloquea las zonas a leer/escribir.
 */
int copiaypega_ulck(int fdorg, int offorg, int size, int fddst, int offdst)
{
    struct flock fl;
    int ret;
    fl.l_whence = SEEK_SET;
    fl.l_start = offorg;
    fl.l_len = size;
    fl.l_pid = getpid();
    fl.l_type = F_UNLCK;
    ret = fcntl(fdorg, F_SETLK, &fl);
    if (ret < 0) return ret;
    fl.l_whence = SEEK_SET;
    fl.l_start = offdst;
    fl.l_len = size;
    fl.l_pid = getpid();
    fl.l_type = F_UNLCK;
    ret = fcntl(fddst, F_SETLK, &fl);
    if (ret < 0) return ret;
    return ret;
}
/* Copia size bytes
 * del fichero de nombre org en su posición absoluta offorg
 *  al fichero de nombre dst en su posición absoluta offdst.
 */
int cyp(char * org, int offorg, int size, char * dst, int offdst)
{
    int fdorg = open(org, O_RDONLY);
    int fddst = open(dst, O_RDWR|O_CREAT, 0666);
    int ret = 0;
    if (fdorg < 0) ret = -1;
    if (fddst < 0) ret = -1;
#if 0
    /* Mediante Entrada/Salida convencional. */
    if (ret >= 0) ret = copiaypega_lock(fdorg, offorg, size, fddst, offdst);
    if (ret >= 0) ret = copiaypega_rdwr(fdorg, offorg, size, fddst, offdst);
    if (ret >= 0) ret = copiaypega_ulck(fdorg, offorg, size, fddst, offdst);
#else
    /* Mediante Proyección de Ficheros en Memoria. */
    if (ret >= 0) ret = copiaypega_ajst(fdorg, offorg,&size, fddst, offdst);
    if (ret >= 0) ret = copiaypega_lock(fdorg, offorg, size, fddst, offdst);
    if (ret >= 0) ret = copiaypega_mmap(fdorg, offorg, size, fddst, offdst);
    if (ret >= 0) ret = copiaypega_ulck(fdorg, offorg, size, fddst, offdst);
#endif
    if (fdorg >= 0) close(fdorg);
    if (fddst >= 0) close(fddst);
    return ret;
}
```

Problema 2.23 (sep 2010)

Sea la siguiente descripción de un programa.

- *El programa consistirá en tres procesos que llamaremos **padre**, **hijo** y **nieto**.*
- *Se abrirá como **origen** de datos el fichero indicado como primer argumento.*
- *Se creará como **destino** de datos el fichero indicado como segundo argumento.*
- *Los procesos deberán estar convenientemente comunicados para intercambiar información y comportarse según se describe.*
- *El **hijo** leerá los datos de **origen**, en trozos de medio KiB y los enviará al **nieto** y simultáneamente irá calculando el número de bytes de **origen**.*
- *El **nieto** habrá de ejecutar el mandato estándar indicado como tercer argumento (que denominaremos **filtro**), para procesar la información que le llegue del **hijo** y enviar su resultado al **destino**.*
- *NOTA: Se ha de entender que el mandato **filtro**, como otros mandatos estándar, si se invoca sin argumentos, se comporta como un filtro entre su entrada estándar y su salida estándar.*
- *Cuando el **nieto** haya concluido, el **hijo** emitirá por su salida estándar una línea de texto indicando el numero total de bytes de **origen**.*
- *Cuando el **hijo** haya concluido, el **padre** emitirá por su salida estándar una línea de texto indicando el numero total de bytes de **destino**. Esta información habrá de obtenerla directamente de dicho descriptor.*

Se pide:

a) Dibuje un esquema con la jerarquía de procesos requerida donde aparezcan los descriptores abiertos por estos procesos y los objetos de datos asociados a estos descriptores.

b) Se pide que implemente en C y para UNIX el programa descrito, haciendo uso directo de los servicios del sistema operativo.

Suponga que el sistema donde se ejecuta este programa dispone de dos discos `hda1` y `hdb1`, usados para el sistema de ficheros raíz y para las cuentas de usuario (montado en `/home`), respectivamente.

*c) Detalle los accesos al sistema de ficheros (a inodos y a bloques datos) necesarios para completar la llamada de apertura del fichero de **origen** `/home/alumno/tmp/datos.txt`.*

d) Suponga que en el sistema de ficheros de `hda1` existe (antes de montar `hdb2`) el archivo `/home/archivo`, así como dos enlaces a este, uno físico en la ruta `/tmp/fisico` y otro simbólico en `/tmp/simbolico`.

*e) Determine cuál sería el comportamiento del programa si el **superusuario** invoca al programa indicando como **destino** las siguientes rutas:*

```
/home/archivo
/tmp/fisico
/tmp/simbolico
```

SOLUCIÓN

a) Esquema:

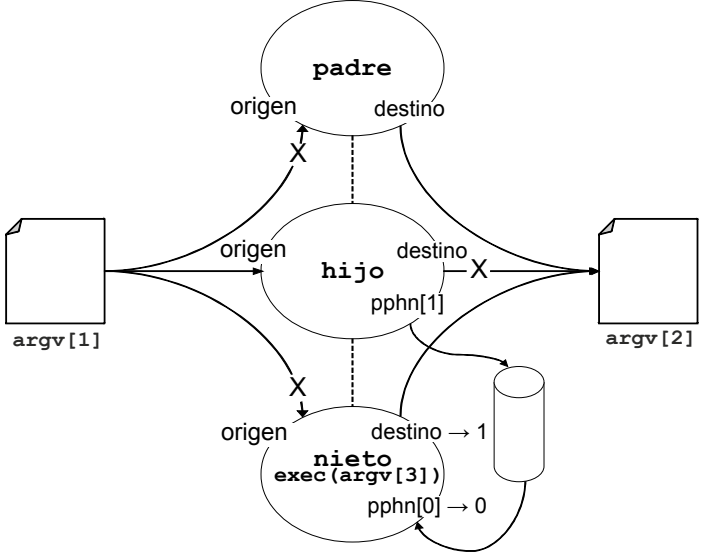


Figura 2.18

b) Programa:

```
/* programa */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SIZE 512

int main(int argc, char*argv[])
{
   int origen;
   int destino;
   int padre = getpid();
   int hijo;
   int nieto;
   int pphn[2];
   unsigned char datos[SIZE];
   int ret;
   int nbytes = 0;
   int status;

   origen = open(argv[1], O_RDONLY);
   if (origen < 0)
   {
      perror(argv[1]);
      exit(1);
   }
   destino = creat(argv[2], 0666);
   if (destino < 0)
   {
      perror(argv[2]);
      exit(1);
   }
   switch(hijo=fork())
   {
   default: /* padre */
      close(origen);
      waitpid(hijo,&status,0);
      nbytes = lseek(destino,0,SEEK_CUR);
      printf("Total bytes destino = %d\n", nbytes);
      break;
   case  0: /* hijo  */
      ret = pipe(pphn);
      if (ret < 0)
      {
         perror(argv[0]);
         exit(1);
      }
      switch(nieto=fork())
      {
```

```
      default: /* hijo */
         close(destino);
         close(pphn[0]);
         while ((ret = read(origen,datos,SIZE)) > 0)
         {
            nbytes += ret;
            write(pphn[1],datos,ret);
         }
         close(pphn[1]);
         waitpid(nieto,&status,0);
         printf("Total bytes  origen = %d\n", nbytes);
         break;
      case  0: /* nieto  */
         close(origen);
         close(0); dup(pphn[0]); close(pphn[0]); close(pphn[1]);
         close(1); dup(destino); close(destino);
         execvp(argv[3],&(argv[3]));
         perror(argv[3]);
         exit(1);
      case -1: /* error */
         perror(argv[0]);
         exit(1);
      }
      break;
   case -1: /* error */
      perror(argv[0]);
      exit(1);
   }

   return 0;
}
```

c) Accesos a inodos y bloques.

acceso a	num.	device	path	luego
inodo	2	hda1	/	DIR, bloques: 100, ...
bloque	100	hda1	//	entrada: home, 20
inodo	20	hda1	/home	flag: sistema montado hdb1
inodo	2	hdb1	/	DIR, bloques: 333, ...
bloque	333	hdb1	/	entrada: alumno, 44
inodo	44	hdb1	/alumno	DIR; bloques: 555, ...
bloque	555	hdb1	/alumno/	entrada: tmp, 66
inodo	66	hdb1	/alumno/tmp	DIR, bloques: 777, ...
bloque	777	hdb1	/alumno/tmp/	entrada: datos.txt, 88
inodo	88	hdb1	/alumno/tmp/datos.txt	FILE, permisos RD

d) Comportamiento si destino es:

`/home/archivo`

Dicho archivo está oculto bajo el punto de montaje `/home`. Luego la llamada `creat` no lo vería, puesto que iría a ver si existe la entrada `archivo` en el directorio raíz (inodo 2) del sistema montado `hdb1`. Al no existir, se crearía este.ooo

`/tmp/fisico`

Un enlace físico es otro nombre para un mismo inodo y datos. Digamos que el inodo de `/home/archivo` y el de `/tmp/fisico`, es el 18553. La llamada `creat` estaría truncando el fichero `/tmp/fisico` y, por lo tanto, también el de `/home/archivo`.

`/tmp/simbolico`

Al decodificarse esta ruta durante la llamada `creat`, el enlace simbólico nos llevaría a intentar decodificar aquella a la que apunta, esto es: `/home/archivo`. Lo cual no s lleva al primer caso que ya hemos visto.

Problema 2.24 (oct-2010)

Sea un sistema de ficheros de tipo UNIX con bloques de 1 KiB, cuya estructura de directorios se indica, parcialmente, a continuación. Asuma que el directorio tmp está vacío inicialmente.

```
tmp  drwxrwxrwt root root
bin  drwxr-xr-x root root
etc  drwxr-xr-x root root
home drwxr-xr-x root root
     rodri drwxr-xr-x rodri prof
     perez drwxr-xr-x perez adm
     lopez drwxr-xr-x lopez adm
     ruiz  drwxr-xr-x ruiz prof
           fich1 -rwxrwxr-- ruiz prof
```

Tres procesos P1 (usuario `rodri`, grupo `prof`) P2 (usuario `ruiz`, grupo `prof`) y P3 (usuario `lopez`, grupo `adm`) ejecutan las sentencias de la tabla adjunta, en el orden de dicha tabla.

Orden	Proceso	
1	P1	`umask(007);`
2	P1	`fd1 = open("/home/ruiz/fich1", O_WRONLY|O_CREAT|O_TRUNC, 0640);`
3	P1	`fd2 = open("/tmp/pru", O_WRONLY|O_CREAT|O_TRUNC, 0660);`
4	P1	`buf = "ABCDEFGHIJK"`
5	P1	`write(fd2, buf, 10);`
6	P1	`lseek(fd2,-8,SEEK_END);`
7	P2	`umask(000);`
8	P2	`fd1 = open("/tmp/pru", O_RDWR, 0666);`
9	P2	`i = 24;  //i está definido como un entero y ocupa 4 bytes`
10	P2	`lseek(fd1,2,SEEK_CUR);`
11	P2	`write(fd1, &i, sizeof(int));`
12	P1	`fork();  //Genera el proceso P4`
13	P4	`lseek(fd2,5200,SEEK_SET);`
14	P2	`lseek(fd1,1024,SEEK_CUR);`
15	P1	`write(fd2, buf, 10);`
16	P2	`write(fd1, &i, sizeof(int));`
17	P3	`fd1 = open("/home/ruiz/fich1", O_RDWR, 0666);`
18	P1	`write(fd1, buf, 10);`
19	P3	`write(fd1, "BB", 2);`

Contestar las siguientes preguntas:

a) Indicar los permisos de `/home/ruiz/fich1` y de `/tmp/pru` después de ejecutarse la orden nº 8.

b) Indicar el contenido de `/tmp/pru` después de ejecutarse la orden nº 11

c) Indicar el tamaño de `/tmp/pru` después de ejecutarse la orden nº 15, así como el espacio físico que ocupa.

d) Indicar el tamaño de `/tmp/pru` después de ejecutarse la orden nº 16, así como el espacio físico que ocupa.

e) Indicar el contenido de los 15 primeros bytes de `/home/ruiz/fich1` después de ejecutarse la orden nº 19.

f) Realizar una gráfica con las estructuras del servidor de ficheros después de ejecutarse la orden nº 19.

g) Indicar qué ocurre si el proceso P3 ejecuta después de la orden nº 19 la sentencia `unlink("/tmp/pru");`. Indicar qué ocurre si es el proceso P2 quién la ejecuta.

SOLUCIÓN

a) El fichero `/home/ruiz/fich1` ya existe, por lo que el open no cambia sus permisos, que son: 774. El fichero `/tmp/pru` no existe previamente, por lo que el open lo crea y establece sus permisos, que son: 660.

b) Después de la orden 5 el contenido de `/tmp/pru`, expresado en ASCII, es: ABCDEFGHIJ y expresado en exadecimal es: 41 42 43 44 45 46 47 48 49 4A. En la orden 8, el proceso P2 abre, a su vez, este fichero, por lo que tiene

su propio puntero, que está a 0. En la orden 10 el proceso P2 mueve su puntero a la posisción 2 y en la orden 11 escribe cuatro bytes con el contenido de la variable i, que exadecimal es 00 00 00 18. El resultado depende de si el computador es de tipo little endian o big endian. Para little endian será: 41 42 18 00 00 00 47 48 49 4A y para big endian será: 41 42 00 00 00 18 47 48 49 4A

c) El proceso P4 y P1 comparten puntero, por lo que la orden 15 escribe desde la posición 5200, haciendo que el tamaño del fichero pase a ser de 5210 B. Dado que exite un hueco que va desde la posición 11 a la 5199, el servidor de ficheros no asignará espacio físico a los bloque intermedios, es decir, solamente exitirá un bloque que da soporte a las posiciones 0 a 1023 del fichero y cuyo contenido será 41 42 18 00 00 00 47 48 49 4A más 1014 blancos (el bloque ha de quedar limpio con la orden 15) y un bloque da soporte a las posiciones 5120 a 6133 del fichero y contendrá blancos en sus posiciones 0 a 79, seguido de 41 42 43 44 45 46 47 48 49 4A, siendo el resto basura. En resumen el fichero tiene asignados dos bloques, es decir, 2 KiB de espacio físico,

d) En la orden 16 el proceso P2 escribe en las posiciones 1030 a 1034 del fichero, lo que supone que el servidor de ficheros debe asignar un nuevo bloque. El fihero no cambia de tamaño, pero ahora ocupa 3 KiB de espacio físico.

e) El proceso P3 no tiene permisos de escritura en el fichero `/home/ruiz/fich1`, por lo tanto la apertura fracasa y como intenta escribir en el descriptor -1, la escritura también fracasa. El fichero ha sido truncado en la apertura por el proceso P1, por lo que solamente tiene un tamaño de 10 B que adquiere en la escritura de la línea 18. Su contenido expresado en ASCII, es: ABCDEFGHIJ y expresado en exadecimal es: 41 42 43 44 45 46 47 48 49 4A.

f) La figura 2.19 muestra las estructuras de datos del servidor de ficheros

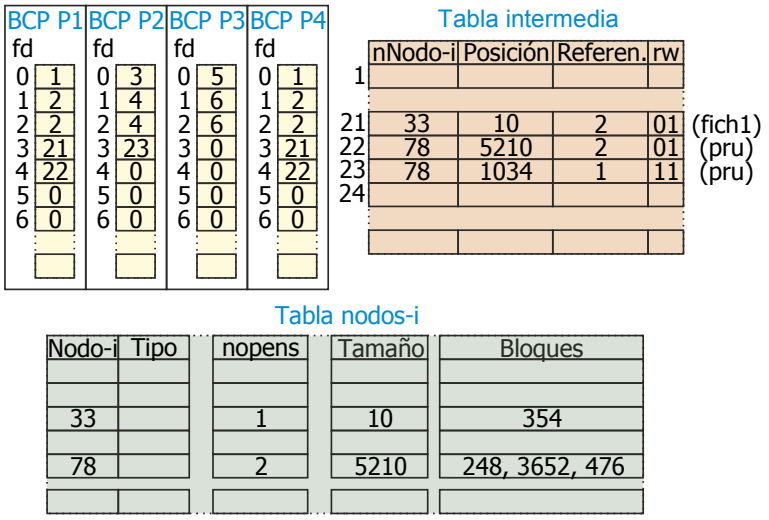


Figura 2.19

g) Observar que el directorio `/tmp` tiene activo el bit t, lo que significa que solamente el dueño de un fichero lo puede borrar. Dado que el dueño del fichero `/tmp/pru` es el usuario `rodri`, grupo `prof`, ni el proceso P2 ni el P3 tienen permisos para realizar un `unlink`, por lo que fracasará.

Problema 2.25 (octubre 2011)

Sean los fragmentos de programas de la tabla adjunta, dichos programas se ejecutan de forma entrelazada de acuerdo a la secuencia establecida en dicha tabla. Supondremos que el fichero "fich" existe y tiene un tamaño de 20 bytes, que el tamaño del bloque es de 1 KiB y que ninguno de los servicios produce error. Indicar el contenido del fichero y el valor de los punteros de posición de ambos procesos al final de la ejecución (línea 9).

Orden de ejecución	**Programa 1**	**Programa 2**
1	`fd = creat("/tmp/fich",0666);`	
2	`write(fd, "01234", 4);`	
3		`fd = open("/tmp/fich", O_WRONLY);`
4	`lseek(fd, 10, SEEK_SET);`	
5		`lseek(fd, -4, SEEK_END);`
6	`write(fd, "8888", 4);`	
7		`write(fd, "abc", 3);`

8	`lseek(fd, -6, SEEK_CUR);`	
9	`write(fd, "A", 1);`	

SOLUCIÓN

Línea 2: 0123

Línea 6: 012300000008888

Línea 7: abc30000008888

Línea 9: abc30000A08888 (tamaño = 14 bytes)

Al terminar, p1 apunta a byte 9 ("0"), p2 apunta a byte 3 ("3")

Problema 2.26 (octubre 2011)

Dado un sistema de ficheros tipo UNIX con direcciones de 32 bits, bloques de 4 KiB y agrupaciones de 2 bloques:

j) Calcular el número de direcciones por agrupación.
k) Calcular el máximo número de agrupaciones direccionables.
l) Calcular el tamaño máximo (en bytes) de fichero suponiendo un nodo-i con estructura interna igual a la presentada en las transparencias de teoría.
m) Calcular el número de accesos a disco necesarios para leer el byte 32 MiB de un fichero cuyo nodo-i ya está en memoria principal y suponiendo que el sistema no dispone de caché de acceso a disco.

SOLUCIÓN

a) Calcular el número de direcciones por agrupación.

Tamaño agrupación = tamaño bloque * nº bloques por agrup. = 4 KiB/bloque * 2 blq/agrup = **8 KiB / agrupación**

Nº direcciones/agrup = tamaño agrup. / ancho dirección = (8 KiB/agr) / (4B/dir) = **2 Kidirecciones / agrupación**

b) Calcular el máximo número de agrupaciones direccionables.

Máximo nº de agrupaciones direccionables = 2^32 agr = **4 Gi agrupaciones**

c) Calcular el tamaño máximo (en bytes) de fichero suponiendo un nodo-i con estructura interna igual a la presentada en las transparencias de teoría.

- 10 punt. directos a agr: 10 agr

- 1 punt. ind. Simple: 2 Ki agr

- 1 punt. ind. Doble: 2 Ki^2 agr = 2^2 Mi agr

- 1 punt. ind. triple: 2Ki^3 agr = 2^3 Gi agr

- Total: (2^3 Gi + 2^2 Mi + 2 Ki + 10) agr * 8 KiB/agr = 2^6 TiB + 2^5 GiB + 2^4 MiB + 80 KiB =

= **64 TiB + 32 GiB + 16 MiB + 80 KiB**

d) Calcular el número de accesos a disco necesarios para leer el byte 32 MiB de un fichero cuyo nodo-i ya está en memoria principal y suponiendo que el sistema no dispone de caché de acceso a disco.

El byte 32MiB corresponde a un puntero indirecto doble, que (dado que el nodo-i ya está en memoria) requiere **3 accesos a disco** (2 accesos a agrupaciones de punteros y 1 acceso final a la agrupación de datos)

Problema 2.27 (octubre 2011)

Completar el siguiente programa fuente (escribir el código que falta en los puntos A, B y C) para que realice lo siguiente:

- Comunicar con un pipe la salida estándar del padre con la entrada estándar del hijo.

- El padre abre el fichero "fich1" y lo copia al pipe.

- El hijo lee del pipe y lo copia al fichero "fich2"

- Al terminar, "fich2" debe contener los mismos datos que "fich1"

Notas:

- Minimizar el número de servicios usados.

- Minimizar el número de descriptores de fichero abiertos.

```
main(void){
   <A>
   switch (fork()){
      case -1: perror("fork()");  exit(1);
      case 0:  /* HIJO */
         <B>
         return 0;
      default: /* PADRE */
         <C>
         return 0;
   }
}
```

SOLUCIÓN

```
#define SZ 1024
<A>
   int pp1[2], fd, n1;
   char buf[SZ];
   pipe(pp1);

<B>
   close(pp1[1]);
   dup2(pp1[0], 0);
   close(pp1[0]);
   fd = open("fich2", O_WRONLY|O_CREAT|O_TRUNC, 0666);/*Alternativa: creat()*/
   while((n1 = read(0, buf, SZ)) > 0)
      write(fd, buf, n1);

<C>
   close(pp1[0]);
   dup2(pp1[1], 1);
   close(pp1[1]);
   fd = open("fich1", O_RDONLY);
   while((n1 = read(fd, buf, SZ)) > 0)
      write(1, buf, n1)
```

Problema 2.28 (marzo 2012)

a) Sea un dispositivo de almacenamiento de 240 MiB con un sistema de ficheros de tipo UNIX. Contestar las siguientes preguntas. En cada caso indique las unidades en que expresa los cálculos.

a1) Escoja un tamaño razonable de agrupación para este sistema de ficheros. ¿Cuántas agrupaciones habrá? ¿De qué tamaño serán las direcciones de agrupación? ¿Cuánto ocupará el mapa de bits correspondiente?

a2) Considere un único archivo de 128 MiB de datos en este sistema de ficheros. Calcule la cantidad de metadatos (información necesaria para poder localizar los datos) que requiere este archivo.

a3) Suponiendo que el tamaño medio de los archivos a almacenar en el sistema es de 8 KiB, ¿cuánto espacio ocupará el mapa de bits de nodos_i? ¿y cuánto el vector de nodos_i?

b) El dispositivo anterior tiene, entre otros, los siguientes directorios:

```
drwxr-xr-x  root  root   /
drwxrwxrwt  root  root   /tmp
drwxr-xr-x  root  root   /bin
drwxr-xr-x  root  root   /etc
drwxr-xr-x  root  root   /home
drwxrwxr-x  rodri prof   /home/rodri
drwxr-xr-x  perez  adm   /home/perez
drw-r-xr-x  root  root   /home2
```

Existe otro dispositivo, de características similares al anterior, que está montado, con la opción MS_NOSUID, sobre home2 y que tiene los siguientes directorios y ficheros:

```
drwxr-xr-x  root  root   /
drwxr-xr-x  lopez  adm   /lopez
drwxr-xr-x  ruiz  prof   /ruiz
-rwxrwxr--  ruiz  prof   /ruiz/fich1
-rwsrwxr--  ruiz  prof   /ruiz/p.exe
```

Se ponen en ejecución los procesos A (usuario y grupo efectivo: ruiz y prof) y B (usuario y grupo efectivo: rodri y prof).

b1) El proceso B crea un hijo que intenta ejecutar el programa p.exe. Explicar si puede ejecutar dicho programa. Indicar y explicar los valores reales y efectivos de usuario y grupo de dicho proceso.

b2) Completar la tabla adjunta que indica la secuencia ordenada en tiempo de servicios ejecutados por los procesos. Inicialmente los procesos tienen como directorio de trabajo su 'home' y como máscara 0750.

Nº	**Proceso**	**Servicio**	**Valor devuelto**	**Permisos, dueño, contenido, … del fichero usado en el servicio**
1	A	mimasc = umask(0037);		
2	A	dir = chdir("/home/rodri");		
3	A	fda = creat("F1.txt", 0771);		Permisos: Dueño:
4	B	fdb1 = open("F1.txt", O_WRONLY| O_CREAT|O_TRUNC, 0640);		Permisos:
5	A	n = write(fda, “1234”, 3);		Contenido:
6	A	m = chown("F1.txt", rodri, prof);		Dueño:
7	A	n = write(fda, “5678”, 4);		Contenido:
8	B	fdb2 = open("F1.txt", O_RDWR);		
9	B	n = write(fdb2, “abcd”, 4);		Contenido:
10	A	fork, creando AH		
11	AH	n = write(fda, “1234”, 3);		Contenido:

b3) Realizara una figura con las principales estructuras de información utilizadas por el servidor de ficheros que refleje la situación existente al final de secuencia anterior.

Solución

a1) El tamaño del bloque varía entre 1 KiB y 8 KiB. Por otro lado, en Unix la agrupación suele ser de un bloque. En la tabla adjunta se muestra el número de agrupaciones y de bits para varios tamaños de agrupación.

Tamaño de la agrupación	**Nº de agrupaciones**	**Mínimo nº de bits de la dirección**	**Mapa bits**
1 KiB	240 Ki agrupaciones	18 bits	30 bloques
2 KiB	120 Ki agrupaciones	17 bits	8 bloques
4 KiB	60 Ki agrupaciones	16 bits	2 bloques
8 KiB	30 Ki agrupaciones	15 bits	1 bloque

En realidad el tamaño de las direcciones de las agrupaciones viene determinada por el software del servidor de ficheros y no por el tamaño del disco. Por lo que este tamaño en un gran porcentaje de las implementaciones actuales será de 32 bits.

El mapa de bits debe ocupar un número entero de bloques, por lo que hay que redondear por exceso.

a2) El tamaño de la metainformación dependerá del tamaño de la agrupación y de las direcciones, por lo que plantearemos los cuatro casos anteriores:

Tamaño agrupación (R)	1 KiB	2 KiB	4 KiB	8 KiB
Direcciones/agrupación (S = R/4)	256	512	1 Ki	2 Ki
Agrupaciones del fichero (T = 128 MiB / R)	128 Ki	64 Ki	32 Ki	16 Ki
Direcciones en nodo_1 (U)	10	10	10	10
Agrupación indirectos (Y)	1	1	1	1
Agrupaciones para doble indirecto usados (V)	256 (más el índice)	127 (más el índice)	31 (más el índice)	7 (más el índice)
Agrupaciones para triple indirecto usados (X)	255 (más dos de índice)	ninguno	ninguno	ninguno
Tamaño metainformación ((V + X + Y)R + 128 B)	516 KiB + 128 B	258 KiB + 128 B	132 KiB + 128 B	72 KiB + 128 B

V = (T – U – Y*S)/S, con la condición de que V=< S. Esta condición solo se aplica para la agrupación de 1 KiB

X = (T – U – Y*S – V*S)/S = (T – U)/S – Y – V

Se observa que con agrupaciones más grandes se requiere menos espacio de metainformación, pero aumenta la fragmentación interna (por término medio se pierde la mitad de la última agrupación).

Un cálculo aproximado, algo menor que el real, se puede hacer considerando solamente el número de direcciones de bloque que se necesitan multiplicado por el tamaño de la dirección. Esto daría, respectivamente, para las agrupaciones de 1, 2, 4 y 8 KiB, los tamaños de 512 KiB, 256 KiB, 128 KiB y 64 KiB.

a3) Si el tamaño medio de los ficheros es de 8 KiB, cabrían en el mismo 240 MiB / 8 KiB = 30 Ki ficheros. Se necesitarían 30 Kib en el mapa de bits y 30 Ki nodos_i en el vector de nodos_i (en la realidad se pondrían más, puesto que es muy negativo quedarse sin nodos_i mientras quede espacio en el disco). Por otro lado, el tamaño típico del nodo_i es de 128 B.

Dado que tendremos un número entero de bloques para el mapa y el vector, el espacio realmente ocupado se muestra en la tabla adjunta.

Tamaño agrupación (R)	1 KiB	2 KiB	4 KiB	8 KiB
Agrupaciones del mapa bits (30 Kib/R)	4	2	1	1
Tamaño ocupado por el mapa bits	4 KiB	4 KiB	4 KiB	8 KiB
Agrupaciones del vector nodos_i (30x128 KiB/R)	3840	1920	960	480
Tamaño ocupado por el vector de nodos_i	3.840 KiB	3.840 KiB	3.840 KiB	3.840 KiB

b1) Todos los directorios tienen la x activa en los tres grupos, por lo que se puede llegar al fichero p.exe. El proceso B tiene usuario y grupo efectivo: rodri y prof. Por tanto, puede ejecutar el programa p.exe, que tiene permisos de ejecución para el grupo prof.

El programa p.exe tiene activo el bit SETUID, por lo que parecería que el hijo del proceso B tomaría la identidad del dueño del fichero, es decir usuario y grupo efectivo: ruiz y prof. Sin embargo, no es así, puesto que el disco se ha montado con la opción MS_NOSUID que inhibe los bits SETUID y SETGID, manteniendo el usuario y grupo efectivo: rodri y prof. Por otro lado, la identidad real es la heredada del proceso B, que supondremos es rodri y prof.

b2) .

Nº	**Proceso**	**Servicio**	**Valor devuelto**	**Permisos, dueño y contenido del fichero usado en el servicio**
1	A	mimasc = umask(0037);	0750	

2	A	dir = chdir("/home/rodri");	0	
3	A	fda = creat("F1.txt", 0771);	3	Permisos: 0740 Dueño: ruiz, prof
4	B	fdb1 = open("F1.txt", O_WRONLY| O_CREAT|O_TRUNC, 0640);	-1	Permisos: 0740
5	A	n = write(fda, “1234”, 3);	3	Contenido: 123
6	A	m = chown("F1.txt", rodri, prof);	0	Dueño: rodri, prof
7	A	n = write(fda, “5678”, 4);	4	Contenido:1235678
8	B	fdb2 = open("F1.txt", O_RDWR);	3	
9	B	n = write(fdb2, “abcd”, 4);	4	Contenido:abcd678
10	A	fork, creando AH		
11	AH	n = write(fda, “1234”, 3);	3	Contenido: abcd678123

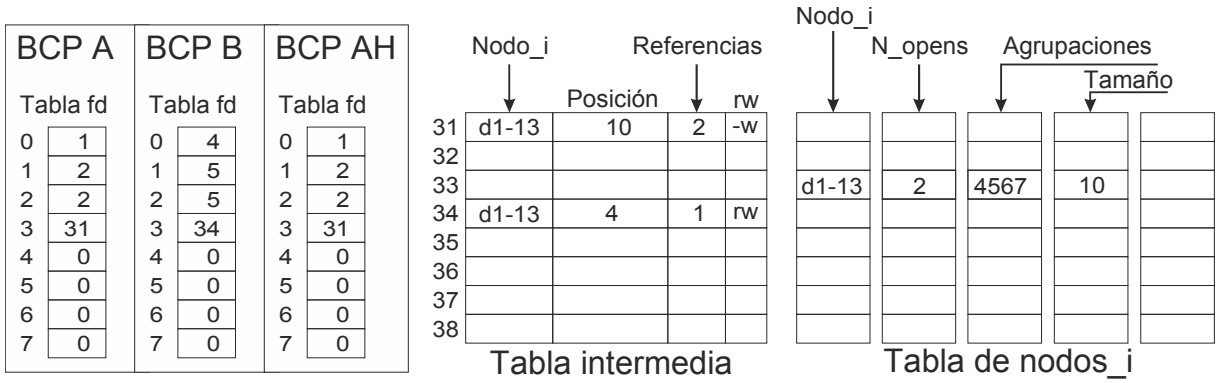


Figura 2.20

b3) Supondremos que los procesos solamente tienen abiertas la entrada y salidas estándar, que el nodo_i del fichero es el 13 del disco d1 y que las primeras líneas libres de la tabla intermedia son la 31 y la 34. La figura 2.20 muestra el contenido de las estructuras pedidas.

Problema 2.29 (julio 2012)

Se desea diseñar un programa Lanzador que reciba como argumentos las rutas de dos ficheros fich1 y fich2 más un número variable de nombres de programas, que llamaremos worker1, worker2, worker3, etc. El programa Lanzador debe poner en ejecución los workers que deberán acceder a los dos ficheros.

fich1 está formado por registros de 124 B, que contienen información para los workers, mientras que fich2 está formado por registros de 12 B, que se rellenarán por los workers.

Se contemplan dos escenarios: Lanzador1 y Lanzador2.

Para el caso de Lanzador1, cada worker lee un registro (no leído por nadie más) de fich1, lo procesa y escribe el resultado en un registro de fich2. El orden de los registros de fich2 no es importante, es decir, el resultado del registro 5 de fich1 no tiene por qué estar en el registro 5 de fich2.

a) Plantear justificadamente cómo pasará Lanzador1 los ficheros a los workers.

b) Plantear justificadamente si es necesario que Lanzador1 espere a que terminen todos los workers antes de terminar. En cualquier caso, codificar cómo esperaría.

c) Diseñar Lanzador1 para el supuesto de no esperar a que terminen los workers.

d) Diseñar los workers, suponiendo que el tratamiento del registro lo hace una función fun1, fun2, fun3, etc. respectivamente para cada worker.

Para el caso de Lanzador2 el worker1 lee el primer registro de fich1 lo procesa con fun1 y le pasa el resultado al worker2, seguidamente lee el segundo registro, lo procesa y le pasa el resultado al worker2 y así sucesivamente

hasta el final de fich1. El Worker2 vuelve a leer el mismo registro de fich1 que utilizó el worker1 y lo combina mediante fun2 con el resultado producido por el worker1, generando un resultado que pasa al worker3, y así sucesivamente. Cada worker recibe del worker anterior un resultado rlativo al registro n, lee el registro n de fich1 y genera un realtado que pasa al siguiente worker. El último worker de la cadena escribe en fich2 el resultado final de haber procesado cada registro de fich1 por todos los workers.

e) Plantear justificadamente cómo pasará Lanzador2 los ficheros a los workers.

f) Plantear detalladamente cómo pasa cada worker su resultado al siguiente, indicando los cambios a introducir en el diseño de Lanzador1 para pasar a Lanzador2.

g) Diseñar los workers.

Solución

a) Nos interesa que los workers compartan el puntero a los ficheros, para que leean automáticamente registros consecutivos de fich1 y escriban automáticamente registros consecutivos en fich2. Dado que tanto el fork como el exec conservan los ficheros abiertos, basta con que Lanzador1 abra ambos ficheros y redirija la entrada estándar al fich1 y la salida estándar al fich2 en los procesos hijos antes de realizar el exec. Otra alternativa, no muy recomendable, sería que Lanzador1 les pase a los workers como argumentos los correspondientes descriptores de fichero.

b) No hace falta que Lanzador1 espere a que terminen los procesos hijos workers. Lo que ocurrirá será que dichos workers se quedarán huérfanos y los heredará el proceso init.

Dado que el número de workers será de argc - 3, puesto que primero vienen los nombres de los ficheros y luego los de los workers, Lanzador1 debe esperar por argc – 3 procesos:

```
for (i=0; i< argc-3; i++)
{
   wait(NULL);
}
```

Se puede observar que el código propusto simplemente espera por argc – 3, que, en un caso general, podrían no ser los workers si Lanzador1 generase más hijos. Una solución que garantice que se espera por los workers exige almacenar los pid de los workers en un vector `int workerpid[argc-3];` y hacer `wait(workerpid[i-3])`

c) Un posible código para Lanzador1 es el siguiente.

```
int main (int argc, char *argv[]) {
   int fde, fds, i, pid;
   fde = open(argv[1],O_RDONLY);
   fds = open(argv[2],O_RDWR);
      if (fde = -1 || fds = -1)  {
      perror("Error en los ficheros.");
      exit(1);
   }
   for (i=3; i< argc; i++) {
      pid = fork;
      switch( fork()) {
      case -1:
         perror("fork");
         exit(1);
      case 0:
         dup2(0, fde);
         dup2(1, fds);
         execl(argv[i],argv[i]);     //Se lanza el worker
         perror(argv[i]);
         exit(1);
      }
   }
   return 0;
}
```

d) El código de los workers puede ser el siguiente:

```
#define TAM_REGE 124
#define TAM_REGS 12
int main (void) {
   char rege[TAM_REGE], regs[TAM_REGS] ;
   while (read(0, rege, TAM_REGE)>0){
```

```
      funi(rege, regs);
      write (1, regs, TAM_REGS);
   }
   return 0;
}
```

e) En este caso, todos los worker necesitan su propio puntero a fich1, puesto que, para procesar el registro n, lo tienen que leer previemente y combinar con el resultado del worker anterior. Por ello Lanzador2 deberá pasar como argumento la ruta de dicho fichero. Además, el último worker escribe en el fichero fich2, por lo que Lanzador2 puede abrir dicho fichero y redirigir la salida estándar del último worker a dicho fichero (también podría pasar el nombre del fichero al último worker, pero esto le haría diferente de los otros).

f) Para pasar los resultados entre los workers, lo mejor es utilizar un pipe entre cada pareja de workers, redirigiendo la entrada y salida estándar en cada hijo adecuadamente. De esta forma los workers leerán el resultado del worker anterior por su entrada estándar y escribirán por la salida estándar.

En la solución propuesta se han lanzado todos los workers de la misma forma, pero hay que tener en cuenta que el primero no leerá del pipe de entrada.

```
int main (int argc, char *argv[]) {
   int fds, i, pid;
   int entrada, pipe[2];
   entrada = dup(0);
   for (i=3; i< argc; i++) {
      pipe(pipe);   //En realidad para i = argc no haría falta crear el pipe porque no se usa
      pid = fork ;
      switch( fork()) {
      case -1:
         perror("fork");
         exit(1);
      case 0:
         dup2(0, entrada);
         close (entrada);
         if (i = argc - 1) {      //En el último worker cambiamos el pipe de salida por el fichero fich2
            close (pipe[1]);
            fds = open(argv[2],O_RDWR);
            pipe[1] = fds
         }
         dup2(1, pipe[1]);
         close (pipe[0]);
         execl(argv[i], argv[i], argv[2],NULL);            //Se lanza el worker
         perror(argv[i]);
         exit(1);
      default:
         close (entrada);
         entrada = pipe[0];
         close (pipe[1]);
      }
   }
   close (entrada);
   return 0;
}
```

g) Suponiendo que el tamaño de los resultados parciales que envía un worker al siguiente es también de 12 B, el código general propuesto para los workers es el siguiente:

```
#define TAM_REGE 124
#define TAM_REGS 12
#define TAM_RESULTADO 12
int main (void) {
   int fde;
   char rege[TAM_REGE], regs[TAM_REGS], resultado[TAM_RESULTADO];
   fde = open(argv[1],O_RDONLY);
   while (read(0, resultado, TAM_REGE)>0){
      read(fde, rege, TAM_REGE);
      funi(rege, resultado, regs);
      write (1, regs, TAM_REGS);
   }
   return 0;
```

}

El código del primer worker es distinto, puesto que en el control de while no se lee de la entrada estándar sino del descriptor fde y el segundo read no existe.

Problema 2.30 (marzo 2013)

Considere el contenido en un sistema de ficheros de un equipo Unix mostrado en la figura 2.21.

Exponer, en todas las preguntas, los pasos a seguir por el servidor de ficheros para determinar la respuesta dada, indicando el número de línea (de 1 a 18) y el grupo de permisos que interviene.

a) ¿Pueden los usuarios pepe, macu *y* mari *acceder a sus directorios?*

b) ¿Puede mari *ejecutar el siguiente mandato?*

```
/home/pepe/itec   /home/macu/teri.c   /home/macu/teri.bak
```

Indicar las identidades reales y efectivas del proceso itec.

c) ¿Podrá itec *abrir para leer y escribir el fichero dado como primer argumento (*teri.c*)? ¿Podrá* itec *crear el fichero dado como segundo argumento (*teri.bak*)?*

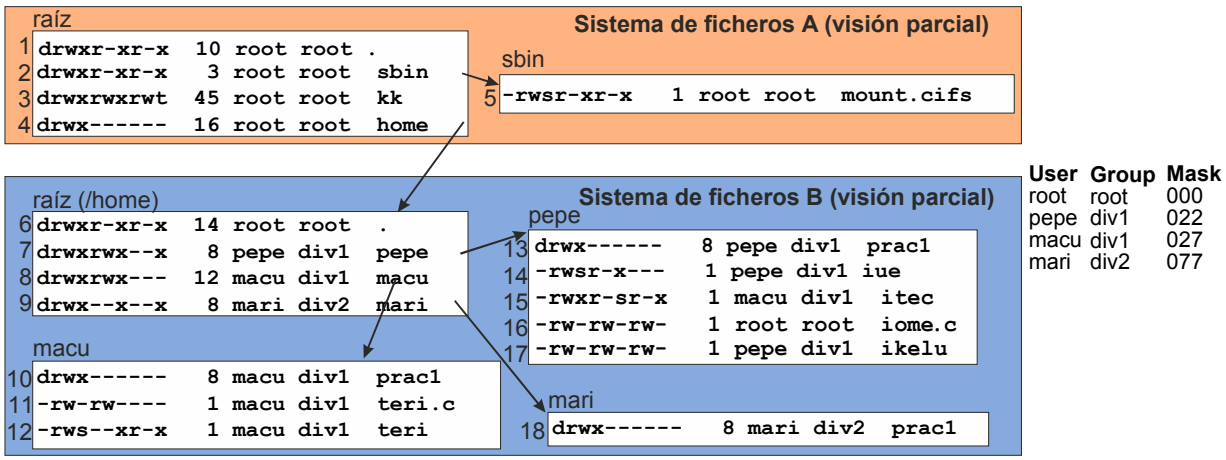


Figura 2.21

Solución

a) Tendremos que analizar los permisos del directorio raíz, del directorio home y de cada uno de los directorios pepe, macu, mari. Marcaremos en negrita, subrayado y rojo el permiso que nos permite acceder. Es de destacar que los permisos del directorio home se indican en la línea 6 **no la 4**, puesto que, al estar montado el sistema de ficheros B en home, los permisos del home original (línea 4) no se consideran. Por otro lado, la mascara no tiene nada que ver con los permisos de acceso, se utiliza exclusivamente para determinar los permisos con los que se crean los ficheros.

Usuario	Línea	Permisos	Acceso
pepe	1	`drwxr-xr-`**`x`**	Puede atravesar
	6	`drwxr-xr-`**`x`**	Puede atravesar
	7	`drw`**`x`**`rwx--x`	**Puede atravesar**
macu	1	`drwxr-xr-`**`x`**	Puede atravesar
	6	`drwxr-xr-`**`x`**	Puede atravesar
	7	`drw`**`x`**`rwx---`	**Puede atravesar**
mari	1	`drwxr-xr-`**`x`**	Puede atravesar
	6	`drwxr-xr-`**`x`**	Puede atravesar
	7	`drw`**`x`**`--x--x`	**Puede atravesar**

b) Hay que ver los permisos para ejecutar `/home/pepe/itec`, los argumentos **no** hay que analizarlos para ver si `/home/pepe/itec` se puede ejecutar.

Usuario	Línea	Permisos	Acceso
mari	1	`drwxr-xr-x`	Puede atravesar
	6	`drwxr-xr-x`	Puede atravesar
	7	`drwxrwx--x`	Puede atravesar
	15	`-rwxr-sr-x`	**Puede ejecutar**

Dado que itec tiene activo el bit SETGID, el grupo efectivo se cambiará al grupo del fichero (es decir div1, no a superusuario), por lo tanto, el proceso itec ejecutado por mari tendrá las siguientes identidades

UID real = mari, GID real = div2, UID efectivo = mari. GID efectivo = div1

c) Se puede observar que, al igual que mari, todo usuario podrá ejecutar itec y que el grupo efectivo de ese proceso será siempre div1. Para teri.c hay que ver que se puede llegar a acceder al fichero y que se tienen permisos de rw. Para el fichero teri.bak hay que ver que se puede llegar la directorio macu y que se tienen permisos de escritura en el mismo. Dado que teri.bak no existe, los permisos con los que se cree dependerán de los permisos que se pongan en el servicio open o creat así como de la máscara que tenga el usuario que ejecuta itec.

Fichero	Línea	Permisos	Acceso	Comentario
teri.c	1	`drwxr-xr-x`	Puede atravesar	
	6	`drwxr-xr-x`	Puede atravesar	
	8	`drwxrwx--x`	Puede atravesar	Si el usuario fuese macu se utilizaría la x del dueño. En otros casos se usa la x del grupo
	11	`-rw-rw----`	**Puede abrir para rw**	Si el usuario fuese macu se utilizaría rw del dueño
teri.bak	1	`drwxr-xr-x`	Puede atravesar	
	6	`drwxr-xr-x`	Puede atravesar	
	8	`drwxrwx--x`	**Puede crear teri.bak**	Si el usuario fuese macu se utilizaría la w del dueño

Problema 2.31 (marzo 2013)

Tenemos que dar formato a un disco de 256 GiB que se va a emplear para almacenar fotografías de alta definición, con un tamaño medio de 8 MiB por fotografía. El mandato que utilizaremos será:

```
mke2fs  -b 4096  -N numeronodos_i
```

La opción -b permite definir el tamaño de la agrupación en bytes (valores válidos son 1024, 2048 y 4096).

La opción -N permite definir el número total de nodos_i del sistema de ficheros.

a) Calcular el valor que se debe dar a la opción -N.

b) Calcular el espacio que se reservará en el sistema de ficheros para la gestión de las agrupaciones libres.

Solución

a) Se necesita un nodo-i por cada fichero, por lo tanto, debemos ajustar el número de nodos-i al número previsto de ficheros. No conviene quedarse cortos, puesto que podríamos llegar a tener espacio libre, pero no nodos-i libres. Tampoco conviene poner muchos más de los necesario, puesto que ocupan espacio en el disco que se podría dedicar a ficheros de usuario.

El numero de ficheros previstos será el tamaño del disco partido por el tamaño de los ficheros, es decir, 256 GiB / 8 MiB = 32 Ki ficheros. El número mínimo que deberíamos poner para la opción -N es de 32 Ki (en la práctica se pondrían bastante más).

Observación: Suponiendo que el nodo-i ocupa 128 B, cada agrupación alberga 4 KiB / 128 = 32 nodos-i. El número de nodos-i debería se múltiplo de 32, porque no tiene sentido dejar una agrupación dedicada a nodos-i medio llena.

b) El espacio libre se puede gestionar mediante un mapa de bits que tenga un bit por cada agrupación del espacio libre del disco. Para calcular el número de agrupaciones del disco, dividimos su tamaño por el de la agrupación: 256 GiB / 4 KiB = 64 Mi agrupaciones. Por tanto, el mapa de bits tendrá 64 Mib = 8 MiB = 2 Ki agrupación.

Por su lado, el mapa de bits de nodos-i necesita 32 Kib = 4KiB = 1 agrupación.

De forma más precisa habría que descontar, del número de agrupaciones del espacio libre, las empleadas por los mapas de bits y por los nodos-i. Los nodos-i ocuparán 32 Ki nodos-i * 128 B/nodo-i = 4 MiB = 1 Ki agrupación.

El espacio liebre es, realmente, 64 Mi agrupación - 2 Ki agrupación - 1 Ki agrupación - 1 agrupación, siendo éste el número de bits del mapa de bits necesario para su gestión.

Problema 2.32 (julio 2013)

Un sistema Linux tiene los usuarios siguientes: A1 del grupo G1, A2 del grupo G2 y root del grupo root. Las características de algunas de las entradas de directorio son las siguientes:

Entrada	Permisos	Enlaces	Dueño	Grupo
/	drwxr-xr-x	32	root	root
/mnt	drwxr-xr-x	5	root	root
/mnt/usbflsh	d---------	2	root	root

Se dispone de una memoria USB en la que se ha establecido un sistema de ficheros con formato ext2. Las características de algunas de las entradas de dicho sistema de ficheros son las siguientes:

Entrada	Permisos	Enlaces	Dueño	Grupo
/	drwx--x--x	10	A1	G1
/Pro1	drwx--x--x	5	A1	G1
/Pro1/doc1	-rw-rw-rw-	2	A1	G1

El usuario A1 ejecuta un programa que contiene la siguiente línea de código:

```
n = mount("/dev/sda1", "/mnt/usbflsh", "ext2", MS_NOEXEC | MS_RDONLY);
```

a) ¿Que valor recibirá en `n`? Decir la razón por la cuál recibe dicho valor.

b) Suponer ahora que el administrador ejecuta la línea de código de la pregunta anterior con éxito. ¿Ha tenido que hacer algún cambio en permisos, dueños o grupos para que la ejecución sea exitosa?

Una vez montada la memoria USB con las opciones detalladas en la línea de código anterior, el usuario A2 ejecuta un programa que contiene la siguiente línea de código:

```
fd = open("/mnt/usbflsh/Pro1/doc1", O_RDWR, 0666);
```

c) ¿Qué valor tendrá `fd`? ¿Por qué?

d) Ahora el que ejecuta el programa con la línea anterior es el administrador. ¿Qué valor tendrá `fd`? ¿Por qué?

Solución

a) El usuario A2 no es root, por lo tanto no puede ejecutar el mandato mount, con independencia de los permisos.

b) El administrador o root puede ejecutar el mandato sin problemas y sin tener que modificar nada. Es de observar que los permisos, dueño y grupo de /mnt/usbflsh no se tienen en cuenta, puesto que quedarán sustituidos por los del directorio raíz del dispositivo montado.

c) Una vez montado el dispositivo flash vemos que el usuario A2 tiene los siguientes permisos

```
/                     drwxr-xr-x permiso de atravesar
/mnt                  drwxr-xr-x permiso de atravesar
/mnt/usbflsh          dr-xr-xr-x permiso de atravesar
/mnt/usbflsh/Pro1     dr-xr-xr-x permiso de atravesar
/mnt/usbflsh/Pro1/doc1  -r--r--r-- permiso de lectura, pero no de escritura
```

Por lo tanto, el servicio open devuelve error, es decir, fd = -1.

Adicionalmente, el mandato incluye el argumento mode, que solamente se aplica si está la opción O_CREAT. Este argumento, por tanto, se ignora.

d) En principio al administrador o root no se le comprueban permisos, por lo que cabría pensar que puede abrir el fichero, aunque éste no tenga permisos de escritura. El servicio devolvería la posición de la primera entra de la tabla de descriptores que esté a 0, por ejemplo la 3, es decir, fd = 3. Sin embargo, una unidad montada como `MS_RDONLY` es como si físicamente no se pudiese escribir en ella, por lo que el root tampoco puede abrir el fichero para escritura, recibiendo, por tanto, fd = -1 y produciéndose el error EROFS.

Problema 2.33 (marzo 2014)

En un sistema Linux se desea disponer de un mandato que cambie el dueño y los permisos de los elementos contenidos en un subdirectorio dado. Por ejemplo, para que el jefe pueda acceder a los ficheros que tiene en su directorio home un empleado que deja el trabajo.

El mandato deberá estar basado en la siguiente función:

```
int CambiaSubDir(char *DirectRecorrer, uid_t owner, gid_t group);
```

Esta función ha de recorrer el subdirectorio DirectRecorrer *y procesar cada elemento encontrado cambiando su dueño a* owner *y* group *y cambiando sus permisos de acuerdo a las siguientes reglas:*

- *Si es un enlace simbólico no deberá ser procesado.*
- *Si es un fichero sin permiso de ejecución para su propietario, le pondrá permisos 0600.*
- *Si es un fichero con permiso de ejecución para su propietario, le dejará los permisos que tiene, incluidos los posibles bits SETUID y SETGID.*
- *Si es un directorio le pondrá permisos 0700.*
- *En cualquier otro caso, dejará los permisos que tenga.*

Nota: Para determinar si un fichero tiene activo un bit de permiso basta con hacer un AND con un máscara que tenga todos ceros menos un 1 en la posición del bit a analizar. Por ejemplo, (mode & 00001) será cierto si el fichero tiene permiso de ejecución para el mundo, (mode & 00002) si tiene permiso de escritura para el mundo, etc. y (mode & 04000) si tiene activo el bit SETUID.

a) Indicar justificadamente la identidad efectiva que deberá tener dicho proceso para poder ejecutar con éxito.

b) Indicar justificadamente en qué orden deben hacerse los siguientes pasos: cambiar el dueño, determinar los permisos, determinar el tipo de fichero y cambiar los permisos.

c) Escribir la función CambiaSubDir.

d) Modificar la función para que se llame recursivamente en cada directorio encontrado.cSe puede pensar, en un principio, que sería suficiente con la identidad del empleado, puesto que se trata de sus ficheros. Sin embargo, al cambiar el dueño se pierden los bits de SETUID y SETGID, por lo que el dueño original del fichero si quisiese restituirlos no podría porque el fichero ya no sería suyo. Por lo tanto, solamente un proceso con identidad efectiva de root podría ejecutar dichas llamadas.

b) Lo primero es ver el tipo de fichero, puesto que si es un enlace simbólico no hay que procesarlo.

Seguidamente, hay que obtener los permisos, puesto que si cambiamos el dueño antes de obtener los permisos perderíamos los bits de SETUID y SETGID.

Luego se cambia el dueño y grupo y, finalmente se cambian los permisos, incluyendo, en su caso, los bits de SETUID y SETGID.

c) Un aspecto importante a tener en cuenta en el diseño del programa es que los nombres que se obtienen en la estructura de tipo `dirent` es el nombre local dentro del directorio. Si queremos utilizar ese nombre, en servicios que requieran el nombre como `chown` y `chmod`, deberemos concatenarlos al nombre del directorio.

En caso de fallo en alguno de los servicios del sistema operativo la función termina y devuelve un valor que indica el lugar donde se ha producido el problema.

La entrada ".." no hay que modificarla, puesto que es el directorio padre. La entrada "." significa que cambiamos el propio directorio pasado como argumento, lo que es correcto o no según se interprete el enunciado. En la sulución presentada se procesa la entrada "." .

En algunos casos, por ejemplo en Linux, la estructura `dirent` contiene un campo `d_type` que permite conocer el tipo de fichero sin necesidad de llamar al servicio `stat`. Sin embargo, como se necesita analizar el modo del fichero en nuestro caso es necesario la llamada `stat`. Deberemos utilizar la versión `lstat`, para poder reconocer los enlaces simbólicos.

```
#define MAX_BUF   300
int CambiaSubDir(char *DirectRecorrer, uid_t owner, gid_t group) {
   DIR *dirp;
   struct dirent *dp;
   struct stat atributos
   char fichero[MAX_BUF];
   mode_t mascaraset;
   //Se abre el directorio DirectRecorrer. Puede ser dirección absoluta o relativa
   if (dirp = opendir(DirectRecorrer) == NULL) return 1;
   //Se lee entrada a entrada del directorio empezando por el "." y el ".."
   while ((dp = readdir(dirp)) != NULL) {  //El readdir obtiene el nombre local
      if (strcmp(dp->d_name, "..") == 0) continue;  //El directorio padre no hay que tocarlo
      strcpy (fichero, DirectRecorrer);     //Hay que concatenar el nombre local a DirectRecorrer
      strcat (fichero, "/");                //para tener el nombre completo
      strcat (fichero, dp->d_name);
      if (lstat(fichero, &atributos) < 0) return 2;
      if (S_ISLNK(atributos.st_mode)) continue;  //Caso de enlace simbólico
// Otra altenativa en Linux para enlace simbólico es poner: if (dp->d_type == DT_LNK) continue;
      if (S_ISDIR(atributos.st_mode)) {          //Caso de directorio
         if (chown(fichero, owner, group) < 0) return 3;
         if (chmod(fichero, 00700) < 0) return 4;
         continue;
      }
      mascaraset = atributos.st_mode & 06000;     //Salvamos los bits de SETUID y SETGID
      if (chown(fichero, owner, group) < 0) return 5;
      if (S_ISREG(atributos.st_mode)) {          //Caso de fichero de usuario
         if (atributos.st_mode & 0100)           //Bit de ejecución del dueño
            if (chmod(fichero, atributos.st_mode | mascaraset) < 0) return 6;
         else
            if (chmod(fichero, 00600) < 0) return 7;
         continue;
      }
      if (chmod(fichero, atributos.st_mode | mascaraset) < 0) return 8;
   }
   closedir(dirp);
   return 0;
}
```

d) Lo primero que hay que tener en cuenta es que si no se elimina el caso "‥" de la función, al hacerla recursiva cambiaría todo el árbol directorio y, además, entraría en un bucle infinito. Si no se elimina el caso "." el programa entraría también en un bucle infinito.

Hay que tocar el if del caso directorio cambiándolo, por ejemplo, por el siguiente código:

```
      if (S_ISDIR(atributos.st_mode)) {    //Caso de directorio
         if (chown(fichero, owner, group) < 0) return 2;
         if (chmod(fichero, 00700) < 0) return 3;
//Evitar bucle infinito
         if (strcmp((dp->d_name, ".") != 0) CambiaSubDir(fichero, owner, group);
         continue;
         }
```

Con la solución propuesta los directorios, menos el original, se tratarían dos veces, una al tratar su nombre y otra al tratar el "." .

También hay que destacar que ha incluido en la llamada la ruta completa al subdirectorio.

Problema 2.34 (abril 2014)

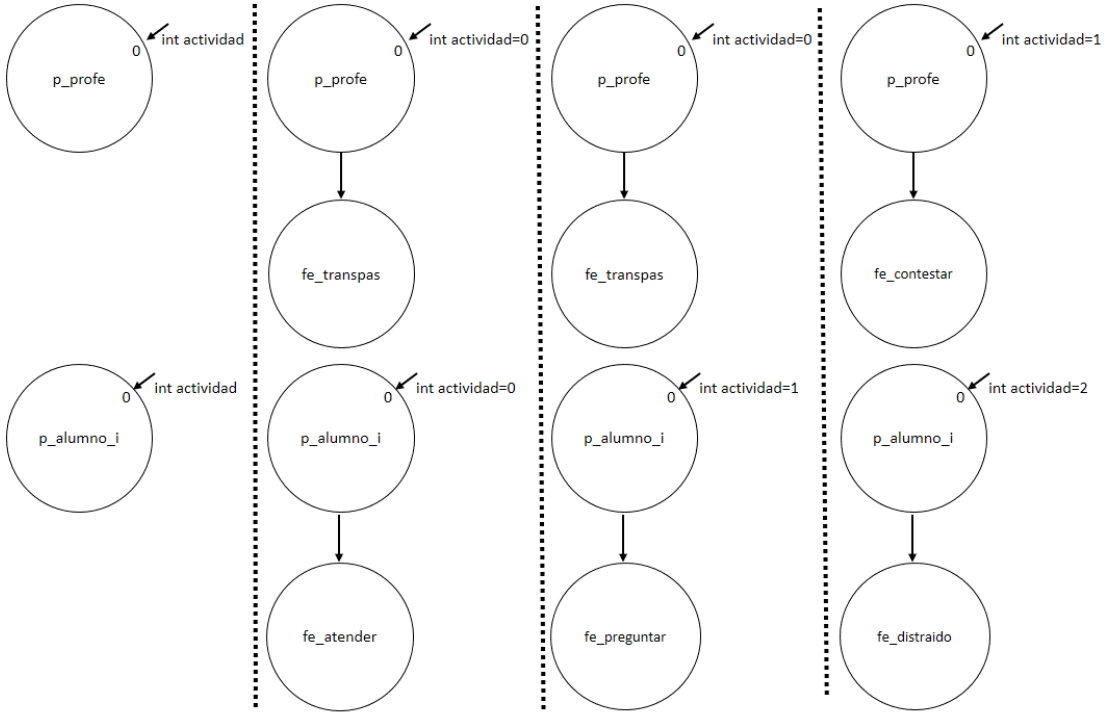
En un aula se encuentran en ejecución el proceso p_profe y los procesos p_alumno_i, donde i puede tomar valores de 1 a n. Durante una clase de teoría de 60 minutos, el proceso p_profe puede invocar los ficheros ejecutables fe_transpas y fe_contestar_p. La elección del fichero ejecutable se realiza con la lectura por la entrada estándar de un valor entero.

Transcurridos los 60 minutos, el proceso p_profe debe recibir un aviso de que su actividad debe terminar y concluye su ejecución.

Los procesos p_alumno_i ejecutan los siguientes ficheros ejecutables: fe_atender, fe_preguntar_a, fe_distraido.

En todos los casos, la ejecución de estos códigos implica la creación de un proceso que al finalizar devuelve un estado de terminación al correspondiente proceso padre (p_profe o p_alumno_i). Antes de crear un nuevo proceso hijo, los procesos p_alumno_i matan al hijo que estuviera previamente en ejecución.

En el siguiente diagrama se ilustran distintas situaciones por las que pueden pasar los procesos, desde el comienzo de la clase (izquierda).



a) *Enumerar los servicios del sistema, funciones de biblioteca y sentencias de lenguaje C que permiten terminar con la ejecución:*
   - *De un proceso*
   - *De un thread*

b) *¿Se podría implementar con threads la ejecución de fe_transpas y fe_contestar? Justifique la respuesta. Indique el máximo número de procesos o threads que podrían coexistir con las restricciones dadas en el enunciado.*

c) *Justifique cuál es la solución o combinación de soluciones más adecuada (ignorar, armar, bloquear) para que el proceso p_profe atienda las señales SIGUSR1 que generan los procesos asociados a la ejecución de fe_preguntar_a. La señal SIGUSR1 la genera fe_preguntar_a cuando se quiere realizar una pregunta al proceso p_profe.*

d) *Especifique el código necesario para implementar el desarrollo de las actividades del profesor durante la clase de tal forma que el proceso p_profe se ejecuta en el hilo principal del código, los ficheros ejecutables fe_transpas y fe_contestar_p en un hijo y se debe controlar la duración de la clase. No hay que implementar la función de tratamiento de la señal SIGUSR1.*

e) Suponiendo que se dispone de los fuentes de fe_atender, fe_preguntar_a y fe_distraido de los ficheros ejecutables que invocan los procesos p_alumno_i, especifique el código necesario para implementar con threads el desarrollo de las actividades de los alumnos. La actividad de los procesos p_alumno_i finaliza tras la recepción de una señal SIGKILL enviada por el proceso p_profe al grupo de procesos p_alumno_i.

Solución

a) De un proceso: kill, return (desde la función main), exit. De un thread: pthread_kill, pthread_exit, return.

b) Al tener que invocarse la ejecución de cualquiera de los 2 programas con el servicio exec, si se hiciera con una solución basada en threads, no volvería a ejecutarse el código del proceso p_profe una vez cargada en memoria la imagen de memoria del nuevo ejecutable que entra en ejecución, por lo que no es posible utilizar threads. El número máximo de procesos será: 2 (correspondientes al p_profe y al hijo creado) + 2 x n (correspondientes a los n alumnos presentes y por cada alumno existirán tanto el proceso p_alumno_i como el hijo creado).

c) El proceso p_profe no puede ignorar la señal enviada por estos procesos. El armado de la señal implica la atención inmediata con su recepción, por lo que se podría interrumpir constantemente el hilo del discurso desarrollado por el proceso p_profe. La solución más adecuada está basada en armar la señal y definir una máscara que incluya el bloqueo de la señal SIGUSR1 de tal forma que la entrega de esta señal quede pendiente hasta que el proceso p_profe la pueda atender desactivando en la máscara el bit de esta señal. Una vez atendida, habría que reactivar la máscara con el bit de la señal SIGUSR1 activo.

d)

```
pid_t pid=0;
void tratar_alarma(void)
{
        kill(pid,SIGKILL);
        fprintf(stderr,"Fin de la clase\n");
        exit(0);
}
int main(void){
int actividad;

    sigaction struct act;
    act.sa_handler=&tratar_alarma;
    act.sa_flags=0;
    sigaction(SIGALRM, &act, NULL);
    alarm(3600);
    while (1) {
        read(0,&actividad,sizeof(int));
        if (pid != 0)
              kill(pid,SIGKILL);
        pid=fork;
        if (pid ==-1)
             {perror("fork"); exit(1);}
        else if (pid==0)
             {switch (actividad)
             {
             case 0:
                   execlp("fe_transpas","fe_transpas",NULL);
                   perror("exec fe_transpas"); exit(1);
             break;
             case 1:
                   execlp("fe_contestar_p","fe_contestar_p",NULL);
                   perror("exec fe_contestar_p"); exit(1);
             break;
             default:
                   fprintf(stderr,"Actividad %d incorrecta\n",actividad);
```

```
                        exit (2);
                  break;
                  }
            }
            else  waitpid(pid,&estado,0);
            }
        return 0;
    }
```

e)

```
    void * func_atender (void *p){
    /* Código fuente del fichero ejecutable fe_atender */
    pthread_exit(0);}
    /* Habría que hacer lo mismo para la especificación del código de los ficheros
ejecutables fe_preguntar_a, fe_distraido que invocan los threads creados por
p_alumno_i */
    int main (void){
    int actividad;
    pthread_attr_t attr;
    pthread_t thid;
      pthread_attr_init(&attr);
      pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
      while (1) {
            read(0,&actividad,sizeof(int);
            switch(actividad) { // Suponemos que los threads se crean correctamente
                  case 0: pthread_create(&thid, &attr, &func_atender, NULL); break;
                  case 1: pthread_create(&thid, &attr, &func_preguntar, NULL); break;

                  case 2: pthread_create(&thid, &attr, &func_distraido, NULL); break;
                  default: fprintf(stderr,"Actividad %d incorrecta\n",actividad); exit
    (1);
                 break;
            }
            pthread_join(thid,NULL);
      }
      pthread_attr_destroy (&attr);
      return 0;
    }
```

3
GESTIÓN DE MEMORIA

Problema 3.1

Sea un sistema monousuario con memoria virtual, que planteamos de muy pequeñas dimensiones para no complicar el problema, que dispone de:

- *Una memoria principal de 8 páginas de 4 KiB*
- *Un SO cuya parte residente ocupa 4 KiB*
- *Una zona de swap con capacidad para 10 páginas.*
- *Una política de no preasignación de swap.*
- *Un disco con dos particiones una para ficheros y otra para swap.*

El SO operativo arranca un único proceso shell y el usuario solicita la ejecución de un proceso compuesto por:

- *Segmento de texto de 3,4 KiB*
- *Segmento de pila, con un tamaño inicial de 200 B*
- *Segmento de datos de sólo lectura de 14 KiB*
- *Segmento de datos de escritura y lectura con un tamaño inicial de 6 KiB*

Se pide:

a) *Suponer que el sistema de ficheros es de tipo UNIX y que la capacidad de datos útil que se consigue es de 160 KiB. Establecer la estructura para este volumen o partición, proponiendo justificadamente el tamaño de cada parte.*
b) *Establecer la tabla de páginas del proceso de usuario. Justificar su estructura y proponer su contenido completo.*
c) *Indicar el contenido de la memoria principal y del swap al iniciar la ejecución del proceso.*
d) *Suponer que, en un instante determinado, el proceso tiene asignado una sola página para la pila, que la página está llena (la pila ocupa toda la página) y que se ejecuta una instrucción de PUSH. Exponer las acciones que se producen.*

Solución PM

a) El dispositivo está dividido en dos particiones, una para ficheros y otra para swap. Nos dicen que la partición de ficheros tiene una capacidad útil (o neta) de 160 KiB. Suponiendo bloques de 1 KiB y sectores de 512 B la partición de ficheros tiene 160 bloques, con estos datos podemos hacer una buena estimación de los tamaños de los diferentes componentes de la partición (véase figura 3.1).

- Boot: Este tamaño puede ser de un sector (p.e. 512 B)

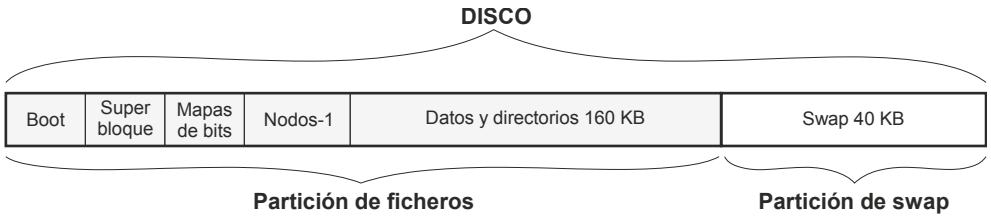


Figura 3.1

- Superbloque: Este tamaño puede ser de un sector (p.e. 512 B)
- Nodos-i: El máximo serían 160 nodos-i, suponiendo que cada fichero ocupa un solo bloque. Sería más lógico tomar un valor menor p.e. 80. Cada nodo-i tiene los 13 punteros más propietario, grupo del propietario, tamaño, instante de creación, instante de último acceso, instante de última modificación e información de control, es decir un tamaño de unas 20 palabras. Por tanto, 80 nodos-i ocupan 80·20·4 = 6400 B.
- Mapas de bits: Un mapa de bits para los bloques: 160 b, más un mapa de bits para los nodos-i: 80 b. Hace falta, por tanto, 30 B.

b) Como sabemos se debe emplear una tabla multinivel, para evitar los huecos. Tendremos un primer nivel con cuatro elementos, uno por segmento, y una tabla de segundo nivel por cada segmento. Las capacidades son las siguientes:

- Segmento de texto: 1 página
- Segmento de pila: 1 página
- Segmento de datos R: 4 páginas
- Segmento de datos RW: 2 páginas

Como suele ser normal (véase figura 3.2), hemos incluido bits de protección en los dos niveles, aunque en el ejemplo propuesto bastaría con incluirlos en uno solo de ellos. Nótese que se han incluido bits de uso, para indicar los elementos de cada tabla que están en uso.

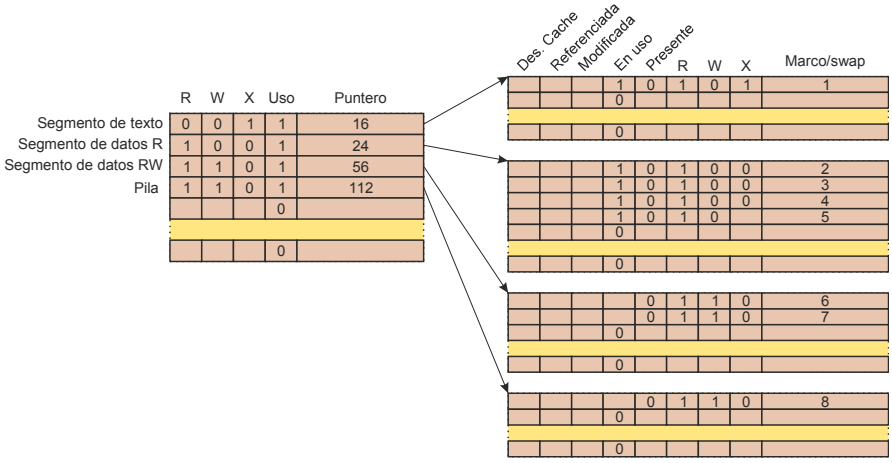


Figura 3.2

c) Al iniciar la ejecución del proceso no tenemos ninguna página en memoria principal. De hecho, la tabla de páginas propuesta en la sección anterior refleja esta situación. Dependiendo del SO, la pila inicial podría estar en un marco de página, puesto que es allí donde la crea el SO.

Al intentar ejecutar el proceso dará un fallo de página que hará traer a un marco de memoria principal la página de texto. Seguidamente, se irán produciendo otros fallos de página que harán que otras páginas deban pasar a marcos.

d) El proceso intenta ejecutar la instrucción de máquina PUSH. Según crezca la pila se incrementa o decrementa el SP, lo que genera una dirección virtual que corresponde a una página inexistente. La MMU recibe esta dirección para traducirla y, al acceder a la tabla de páginas, detecta que se sale de rango. En efecto, la pila tiene asignada una sola página, por lo que el tamaño de su tabla es 1. Sin embargo, la dirección presentada a la MMU se refiere a la siguiente página, inexistente. Por tanto, la MMU genera un trap de violación de memoria.

El SO trata la interrupción y, como se trata de un trap de memoria, pasa a ejecutar el gestor de memoria. Éste analiza la situación y detecta que se trata de un desbordamiento de pila, por lo que tiene dos posibilidades:

- Si existe espacio disponible se asigna más memoria al proceso. Dado que se funciona sin preasignación de swap el soporte físico que se asigna es un marco de página. En caso de no tener marcos libres habría de liberar uno previamente. En la figura 3.3 se ha asignado el marco 5.
- Si no existe espacio disponible no puede continuar la ejecución del proceso, por lo que el SO le envía una señal, para que muera.

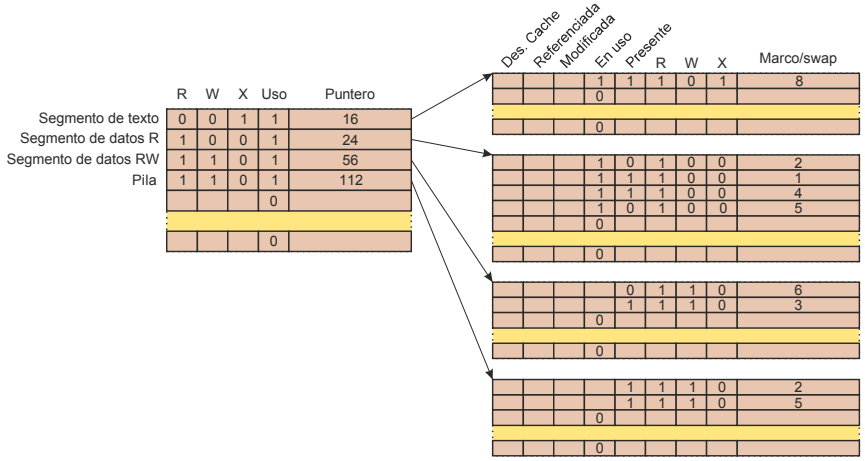


Figura 3.3

Problema 3.2

Sea un sistema con las siguientes características:

- *Memoria virtual con direcciones de 32 bits, en el que cada usuario tiene su propio espacio virtual que va desde la dirección 00000000 hasta la 7FFFFFFF.*
- *Se utilizan páginas de 1 KiB y se realiza la asignación de espacio de disco en bloques de 4 KiB.*
- *A los discos se les ha dado formato con sectores de 512 B.*

Se desea ejecutar la aplicación definida seguidamente:

Fichero datos.h

```
char vector [23000];
int a = 5; int b = 15; int c = 23; int d = 7; int e ;
```

Fichero programa.c

```
#include <datos.h>
int main (void)
{
   char x; char *w; char *m;
   int v = 10; int fd;
   e = v;
   w = malloc (27000);
A
   f (e);
   fd = open ("dat.txt", O_RDONLY);
   m = mmap(0, 4000, PROT_READ, MAP_SHARED, fd, 0);
B
   .
   .
   exit (0);
   }

void f (int a)
```

```
{
   printf ("%d\n", a);
   return;
}
```

Suponiendo que la cabecera del ejecutable ocupa 1 KiB y que el código de la mencionada aplicación ocupa 211 KiB, determinar el tamaño real y el espacio ocupado en disco por el fichero ejecutable.

Para los tres casos siguientes: al iniciar la ejecución del mencionado programa, al llegar al punto A y al llegar al punto B, definir las regiones o segmentos de la imagen de memoria del proceso especificando para cada una de ellas:

- *Dirección de comienzo en hexadecimal y tamaño en KiB.*
- *Soporte físico y, en su caso, técnica de optimización empleada.*
- *Contenido de la región o segmento.*

Solución

1º Determinación del tamaño real y en disco del fichero ejecutable.

Supondremos 4 bytes para los enteros (depende del sistema, podrían ser 2 bytes). La cabecera ocupa 1 KiB y el código 211 KiB, como sólo se inicializan 4 enteros en el fichero de cabecera (a,b,c y d), estos ocuparan 16 bytes.

Así el tamaño real será 212 KiB + 16 B, igual a 217.104 bytes.

El espacio se asigna en agrupaciones de 4 KiB, luego, para cubrir los 212 KiB + 16 B, se necesitan 54 agrupaciones = 54·4 KiB. Es de notar que el ejecutable también podría incluir una tabla de símbolos cuyo tamaño desconocemos.

2º-a Regiones o segmentos de la imagen de memoria del proceso al comienzo, en A y en B.

Al comienzo:

<table>
<tr><td>00000000-00034BFF</td><td colspan="2">Código, 211 KiB</td><td></td></tr>
<tr><td>00034C00-00034C0F</td><td colspan="2">DVI[1], 16 B</td><td>4 B por 4 int definidos e iniciados en datos.h.</td></tr>
<tr><td>00034C10-0003A5EB</td><td colspan="2">DSVI[2], 23.004 B</td><td>23.000 B char[ ] + 4 B por 1 int en datos.h</td></tr>
<tr><td>0003A5EC-xxxxxxxx</td><td colspan="2">Heap (Espacio libre)</td><td>El Heap crece hacia abajo ⇓</td></tr>
<tr><td></td><td colspan="2"></td><td></td></tr>
<tr><td></td><td colspan="2"></td><td></td></tr>
<tr><td></td><td>PILA:Var. Locales 17 B</td><td rowspan="4">Bloque de Activación de main()</td><td>1 B char + 4 B por 2 char* + 4 B por 2 int</td></tr>
<tr><td></td><td>PILA: puntero a marco anterior 4 B</td><td>32 bits = 4 bytes</td></tr>
<tr><td></td><td>PILA: Dir. Retorno 4 B</td><td>32 bits = 4 bytes</td></tr>
<tr><td></td><td>PILA: argc 4B</td><td>32 bits = 4 bytes</td></tr>
<tr><td>yyyyyyyy-7FFFFFFF</td><td colspan="2">PILA:Var. Entorno</td><td>La pila crece hacia arriba ⇑</td></tr>
</table>

[1] Datos con Valor Inicial.

[2] Datos Sin Valor Inicial.

En el punto A: Se incluyen en el heap los 27.000 B del malloc. Si el heap no tenía ese tamaño habrá de crecer.

0003A5EC-00040F63	27.000 bytes	Reservado por el *malloc(...)*
00040F64-xxxxxxxx	Espacio libre	

En el punto B: Se añade una región A al ejecutar el sistema operativo el mmap. La dirección de comienzo de la región del mmap dependerá del sistema operativo. Supondremos la 01000000. Además crece la pila con el bloque de activación de la función f, compuesto por el argumento a, dirección de retorno de f y puntero a marco anterior. En total 3 enteros de 4B.

01000000-01000F9F	4.000 bytes	Reservado por el *mmap(...)*
01000FA0-zzzzzzzz	Espacio libre	
yyyyyyyy-7FFFFFFF		Crece la pila con la llamada a la función *f*

2º-b Soporte físico al comienzo, en A y en B.

Comienzo:	
Código	Paginar sobre fichero *programa*
DVI	Paginar sobre fichero *programa*
DSVI	Rellenar 0
Heap	Rellenar 0
Región A	No existe
Pila	Rellenar 0, marco con pila inicial.

En A:	
Ejecuta	Algún marco
e = 10	1 marco
malloc()	Rellenar 0
No existe	
	Igual que al comienzo

En B:	
Ejecuta	Algún marco
mmap()	Fichero *datos.txt*
	Puede necesitar un marco adicional.

2º-c Contenido de la región o segmento comienzo, en A y en B.

Comienzo:	
Código	Fichero *programa*
DVI	Fichero *programa*
DSVI	Rellenar 0
Heap	
Región A	
Pila	Entorno activación *main()*

En A:
Fichero *programa*
Fichero *programa*
e = 10
Rellenar 0
No existe
w = dir malloc

En B:
Fichero *programa*
Fichero *programa*
e = 10
Rellenar 0
Fichero *datos.txt*
Se añade el marco de activación de f

Problema 3.3

Un sistema tiene un disco de 128 KiB organizado en bloques e intercalado simple y una memoria principal de 44 KiB. La gestión de la memoria virtual se realiza con páginas y con tablas de páginas de dos niveles. Las direcciones virtuales son de 24 bits, de los cuales 6 bits sirven para acceder a la tabla de páginas de primer nivel y 6bits para acceder a la tabla de páginas de segundo nivel.

Además, en nuestro sistema existe una biblioteca dinámica lib.so que contiene la función func.Se tiene, además, el siguiente fragmento de programa.

```
char x[1024];
char y[1024];
...

int main (void)
{
   int i, pid;

   /* A  Situación inicial */
   for (i=0; i<1024; i++)
      x[i]=y[i];
   func("datos");
   /* B  Después del for y de usar la biblioteca dinámica */
   ...
}
```

a) *Sabiendo que una vez compilado el programa el código ocupa 5K, los datos con valor inicial (d.v.i.) ocupan 6K y que los datos sin valor inicial (d.s.v.i.) tienen un tamaño de 2K, se pide* ***representar gráficamente*** *el fichero* ***ejecutable*** *resultante, suponiendo que utilizamos el formato ELF y que el tamaño de la cabecera del ejecutable es de 1 KiB.*

b) ***Representar gráficamente*** *el* ***mapa de memoria*** *del proceso en la situación inicial A.*

*c) Con el sistema de gestión de memoria propuesto inicialmente, indicar qué **tipo de fragmentación** se está generando.*

*d) Sabiendo que se tiene un sistema sin preasignación de swap, se pide **representar las tablas de páginas** tanto de primer como de segundo nivel en el instante A. Para cada Entrada de la Tabla de Páginas (ETP) se tendrá en cuenta la siguiente información: bit de validez, bit presente/ausente, bit modificado, bit referenciado, bit COW, bit Rellenar a Ceros (RC), bit de Rellenar de Fichero (RF), protección y número de marco o bloque de disco.*

*e) Responder **brevemente** a las siguientes preguntas: ¿qué ocurre si una misma página es expulsada varias veces al área de swap?, ¿quién y cuándo escribe el bit de modificado y quién y cuándo lo inicializa?*

*f) Indicar qué **cambios** se realizarían en el mapa de memoria del proceso y en la tabla de páginas al llegar el proceso al instante B. Suponer que la llamada a función func necesita solamente la primera página de la región de texto y la primera página de la región de datos de la biblioteca dinámica.*

*g) Explicar **brevemente** qué ocurre si el proceso crea un proceso hijo y éste escribe en la zona de datos de la biblioteca dinámica.*

Estado inicial del disco

0	*1*	*2*	*3*
4 *lib.so texto*	*5* *lib.so texto*	*6* *lib.so datos*	*7* *lib.so datos*
8 *lib.so datos*	*9*	*10*	*11*
12	*13*	*14*	*15*
16 *SWAP*	*17* *SWAP*	*18* *SWAP*	*19* *SWAP*
20 *SWAP*	*21* *SWAP*	*22* *SWAP*	*23* *SWAP*
24 *SWAP*	*25* *SWAP*	*26* *SWAP*	*27* *SWAP*
28 *SWAP*	*29* *SWAP*	*30* *SWAP*	*31* *SWAP*

Siglas utilizadas:

RF: Rellenar de fichero
RC: Rellenar a ceros
COW: Copy on write
X: Ejecución
R: Lectura
W: Escritura

Solución

a) El fichero ejecutable está formado por la cabecera, y una serie de secciones que son descritas en la cabecera. Por cada sección viene su dirección inicial y su desplazamiento, excepto para la sección de datos sin valor inicial, de la que sólo se especifica su tamaño, véase figura 3.4.

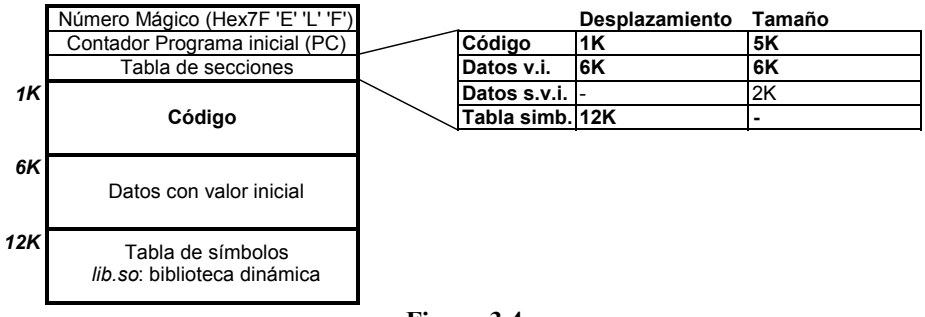


Figura 3.4

b) El mapa de memoria contiene: código, datos con valor inicial, datos sin valor inicial y pila. Dado que se dedican 12 bits de la dirección para seleccionar la páginas, quedan otros 12 para la direción dentro de la página, lo que significa que la página es de 4 KiB. Consideraremos que las tres primeras regiones están en la zona alta y la última en la zona más baja. Además, se ha supuesto que tanto los datos con valor inicial como los sin valor inicial están en su propia región. El espacio total de direccionamiento es de 2^24 = 16 MiB. Véase figura 3.5.

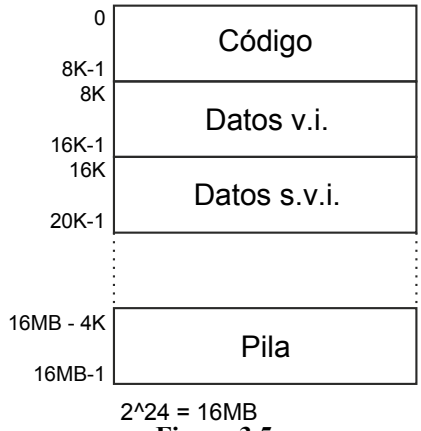


Figura 3.5

c) Debido a que tenemos un esquema de paginación, se produce fragmentación interna. Puede desperdiciarse parte de último marco asignado cada región del proceso.

d) Véase figura 3.6.

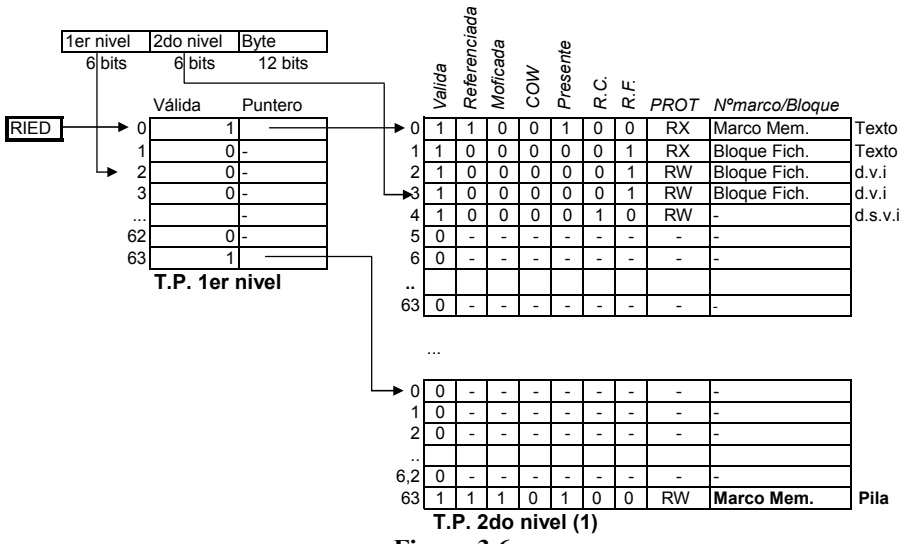


Figura 3.6

e) Asignación de espacio de swap: la primera vez se le reserva espacio en el swap, las siguientes veces usará el espacio que se le reservó la primera vez. El bit de modificado lo inicializa el sistema operativo cada vez que lleva una página de memoria secundaria a memoria principal. Es escrito por la MMU cuando hace un acceso de escritura a la página. Sirve para saber si la página está sucia (con cambios), a la hora de expulsarla a Memoria Secundaria.

f) Véase figura 3.7.

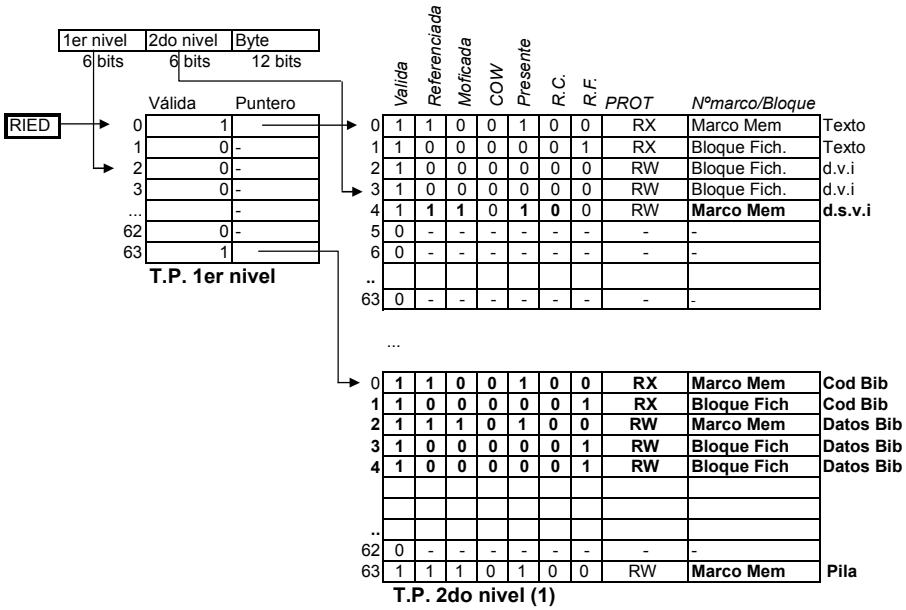


Figura 3.7

Los cambios son:

- Accedo y escribo en la pila por la variable i
- Accedo y escribo en los d.s.v.i. por los vectores x e y
- Creo dos nuevas regiones para la biblioteca dinámica
  - Región de datos: Se proyectan los datos de la Bibl dinámica de forma privada y de lectura/escritura
  - Región de código: Se proyecta el código de la Bibl dinámica de forma compartida y de lectura/ejecución
- Dentro de la Bibl. Dinámica se accede a la primera página del código y a la primera de la región de datos.

Luego en el mapa de memoria del proceso se crean dos nuevas regiones para la Bibl. Dinámica. Estas regiones se crean a partir de la dirección 10 MiB del mapa de memoria y tienen un tamaño de 2 páginas (8 KiB) para la zona de código y 3 páginas (12 KiB) para la zona de datos.

Los cambios están en negrita. Se han creado las regiones de la biblioteca dinámica algo por encima de la pila.

g) Inicialmente la región se comparte como COW y el hijo, que escribe primero, crea una copia privada cuando va a modificar los datos.

Problema 3.4 (junio 2003)

Sea el código siguiente, con montaje dinámico, que ejecuta en una máquina con memoria virtual y páginas de 4 KiB. La función `cos` *está en la biblioteca libm y* `printf` *en libc. Los tamaños de estas bibliotecas son los siguientes:*

Biblioteca	Texto	Datos con valor inicial	Datos sin valor inicial
libm	191 KiB	8 KiB	3 KiB
libc	147 KiB	3 KiB	2 KiB

mi_cos.c:

```
#include <stdio.h>
#include <math.h>
   double coseno(double v) {
      return cos(v);
   }
   int main (void) {
      double a, b = 2.223;
      a = coseno(b);
      printf("coseno de %f:%f\n",b,a);
```

```
    return 0;
}
```

a) *Establecer la imagen de memoria en el instante en el que el proceso mi_cos está ejecutando la función* `cos`. *Suponer unos tamaños razonables para el texto y las variables de entorno de mi_cos.*

b) *Repetir para el instante en el que el proceso está ejecutando la función* `printf`.

c) *Una máquina con palabras de 64 bits y con alineación de datos ejecuta el fragmento de código siguiente:*

```
int cnt;
struct {
   char caracter;
   double flotante;
} * estructura;

estructura = malloc(720);
for (cnt = 0; cnt < MAXIMO_POSIBLE; cnt++) {
   estructura[cnt].flotante = cnt * 100.0;
}
```

Suponiendo que el double ocupa 64 bits, se pide determinar el máximo valor que se le podría asignar razonablemente a 'MAXIMO_POSIBLE'. Explicar el resultado obtenido.

d) *Sean dos programas A y B que contienen los fragmentos de códigos listados en la tabla 3.1.*

Fragmento de programa A

```
int fd;
struct {
   void * direccion;
   int array[100];
} * area;
/* Se supone que "BUFFER" existe. */
fd = open("BUFFER", O_RDWR);
area = mmap(0, sizeof(*area),
PROT_READ|PROT_WRITE, MAP_SHARED,
fd,0);
area->direccion = area;
area->array[0] = 1;
area->array[1] = 2;
munmap(area, sizeof(*area));
close(fd);
```

Fragmento de programa B

```
int fd;
struct {
   void * direccion;
   int array[100];
} * area;
/* Se supone que "BUFFER" existe. */
fd = open("BUFFER", O_RDWR);
area = mmap(0, sizeof(*area),
PROT_READ|PROT_WRITE, MAP_SHARED,
fd,0);
area = area->direccion;
area->array[2] = 3;
area->array[3] = 4;
munmap(area, sizeof(*area));
close(fd);
```

Tabla 3.1

Se supondrá que el programa A requiere mucha más memoria que el programa B para su ejecución y que ejecuta esta sección de código antes que el programa B ejecute la suya. Se pide determinar el contenido del fichero BUFFER al final de la ejecución de ambos programas. Explicar el resultado obtenido.

Solución

a) y **b)** Las soluciones a las preguntas a y b se encuentran en la figura 3.8, en la que son de destacar las siguientes consideraciones:

El programa mi_cos no tiene ni datos con valor inicial ni datos sin valor inicial, por lo que su imagen no tendrá estas subregiones. Además, no genera datos dinámicamente, por lo que no hace uso del heap. Sin embargo, el SO no sabe que el programa no hará uso del heap, por lo que, como siempre, dejará espacio para que se pueda crear el heap y para que éste pueda crecer.

Dado que el montaje es dinámico, las bibliotecas no se encuentran en el código del programa. Sabemos que las bibliotecas dinámicas se pueden cargar según tres procedimientos, de los cuales descartamos directamente el método de montaje explícito, puesto que el código no incluye ninguna cláusula para ello. Nos queda, por tanto, el montaje en tiempo de carga en memoria y el montaje al invocar el procedimiento. Como sabemos, el montaje más eficiente y utilizado es al invocar el procedimiento, por lo que es el que hemos considerado en esta solución.

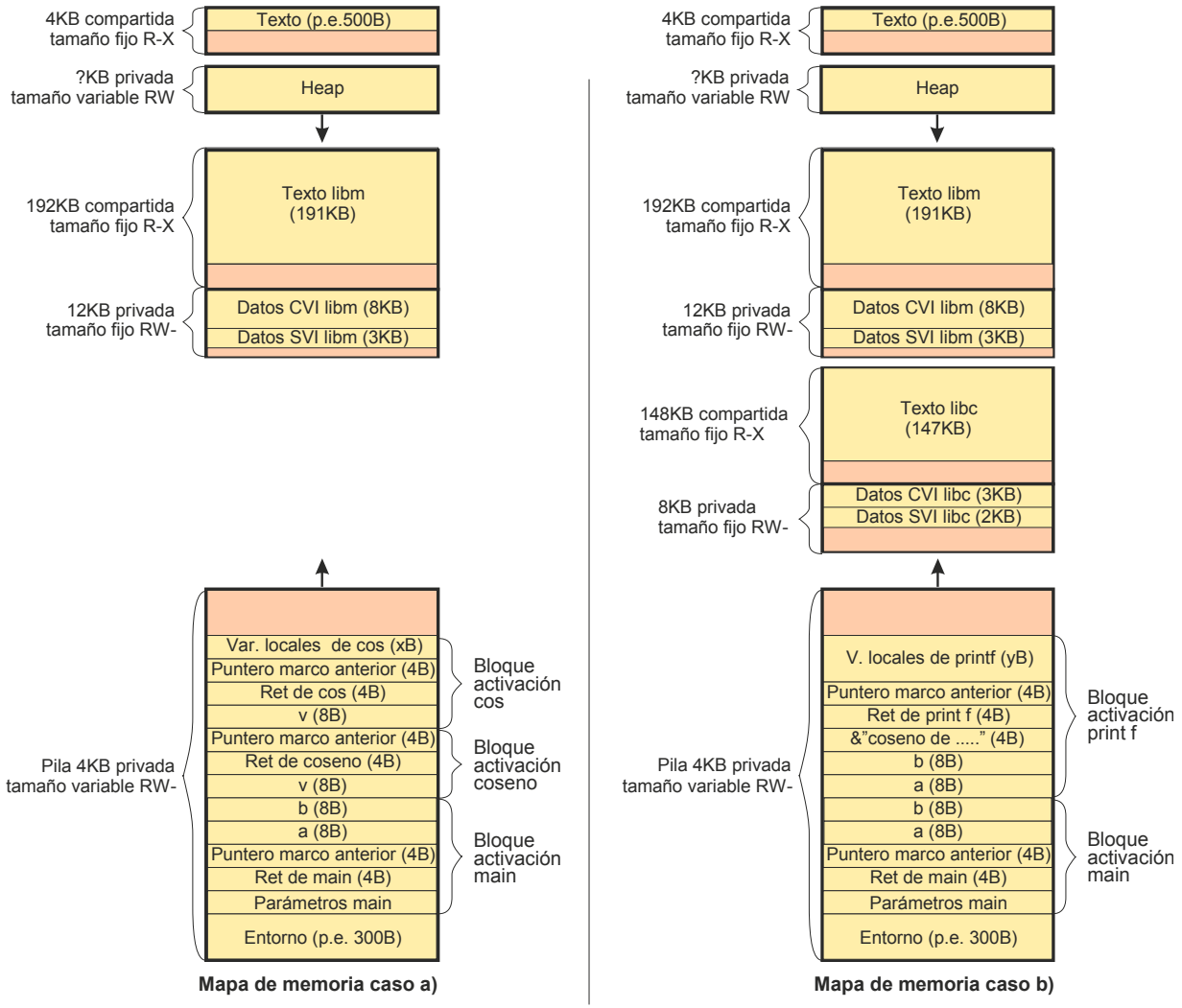


Figura 3.8

Por ello, en la situación a) la imagen de memoria solamente incluye la biblioteca libm, que es la que contiene la función cos. Sin embargo, en la situación b) encontramos en la imagen de memoria ambas bibliotecas, puesto que el SO **no sabe** que el programa ya no utilizará la biblioteca libm, por lo que **no la elimina** de la imagen de memoria del proceso.

Nótese también que todas las regiones de memoria contienen un número entero de páginas, por lo que quedan trozos de las mismas sin ocupar. Además, es de destacar que las zonas de datos con y sin valor inicial se pueden juntar en una única región, puesto que son de tamaño fijo. Así, la figura incluye una región de datos para cada una de las bibliotecas.

La figura no inlcuye, sin embargo, una región de datos del proceso, pues su código no contiene datos globales. Dicha región podría también albergar el heap. Se trataría de una región privada con derechos RW- que podría estar pegada a la región de texto del proceso.

Las bibliotecas **no tienen** pila ni heap propio, utilizan las del proceso.

El SO podría haber pegado la región de texto de la biblioteca libc a la región de datos de la biblioteca libm, puesto que es de tamaño fijo. Por el contrario, siempre ha de dejar espacio para el crecimiento de la pila y del heap.

c) El char ocupa un byte y el double ocupa 8 bytes. Sin embargo, nos dicen que la máquina hace alineación de datos, por lo que cada estructura ocupa dos palabras, como muestra la figura 3.9, al no poderse almacenar el double en la misma palabra que el char.

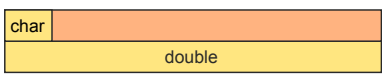


Figura 3.9

Por lo tanto, para que no se desborde el espacio de 720 bytes reservado para las “estructuras” solamente podemos almacenar 720/16 = 45 “estructuras”, por lo que MAXIMO_POSIBLE = 45.

d) El programa A proyecta el fichero en modo shared, por lo que se reflejará en el mismo todo lo que se escriba en la región de memoria. En concreto se escriben tres enteros. El primero corresponde a la dirección virtual en la que comienza la región de memoria ‘area’, que llamaremos ‘dir_area’. El segundo y tercero reciben un 0001 y un 0002 respectivamente. El resto del fichero no queda modificado, por lo que mantiene los valores anteriores.

El programa B proyecta el mismo fichero también en modo shared, por lo que éste reflejará los cambios que haga el programa en la región de memoria.

Nos dicen que el programa A require mucha más memoria que el B, lo que lleva a que el SO deberá crear unas imágenes de memoria muy distintas, por lo que la dirección de la región ‘area’ será muy distinta en el programa A que en el programa B.

Si nos fijamos en el programa B, podemos observar que toma la primera palabra de la región (que contiene el valor ‘dir_area’ dejado en el fichero por el programa A) e intenta escribir en las direcciones dir_area + 3 y dir_area + 4. Estas direcciones eran válidas para el programa A, pero no tienen porqué serlo para el B. Es más, lo más probable es que estas direcciones no correspondan a ninguna región de dicho programa, por lo que el resultado más probable será un error de violación de memoria.

Podemos afirmar que el contenido final del fichero será, seguramente, el de la figura 3.10.

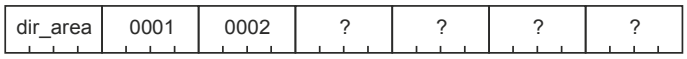


Figura 3.10

Problema 3.5 (septiembre 2003)

Sea el código adjunto que se monta en modo dinámico para ser ejecutado en una máquina con memoria virtual y páginas de 2 KiB. Considere que el código del programa ocupa 3.573 B.

```
01 #include <stdio.h>   /* fichero ejemplo.c */
02 #include <stdlib.h>
03
04 int a;
05 int b = 50;
06
07 void funcion (int c)
08 {
09     int d;
10     static int e = 2;
11     d = e
12     c = d + 5;
13     printf ("Esto lo imprime la funcion funcion()\n");
14 }
15
16 int main (void)
17 {
18     char *f;
19     f = (char *) malloc (512);
20     funcion (b);
21     free (f);
22     exit (0);
23 }
```

a) Haga un gráfico con las diferentes secciones del fichero ejecutable correspondiente, especificando en qué sección del mismo se almacena cada una de las variables del programa y por qué se almacena en dicha sección.

b) Para cada uno de los puntos de ejecución correspondientes a la finalización de las cláusulas 16, 12 y 21 haga un gráfico con las distintas regiones de memoria del proceso que ejecuta el programa anterior, indicando:

- *Dirección de comienzo y fin de cada región.*
- *Características de la región.*
- *Contenido de cada región, indicando expresamente las variables incluidas en cada una de ellas, así como su valor.*

Como no se indican las necesidades de memoria de la librería dinámica libc utilizada por el programa, especifique los tamaños necesarios mediante nombres simbólicos.

Solución

a) El fichero ejecutable constará de las siguientes secciones:

- **Cabecera**, donde se almacena el número mágico y el valor inicial de los registros del procesador en el momento de la carga.
- **Código**, donde se almacena el código máquina del programa ya enlazado con las librerías estáticas necesarias. El código se dice en el enunciado que ocupa 3.573Bytes.
- **Datos** con valor inicial, en este caso sería 'b=50' (línea 05) y 'static e=2' (línea 10). Se supone un sistema en que cada entero ocupa 4bytes, por tanto esta región ocuparía 8 bytes. El resto de variables del programa se almacenarán en el código.
- **Tablas** para montaje dinámico y otra información necesaria para la formación del ejecutable o para depuración.

b) Nos piden la **imagen de memoria** al terminar la ejecución de las líneas 16, 12 y 21. Suponemos tamaño de memoria virtual de 32 bits y el tamaño de los enteros de 4 bytes. El enunciado dice que el tamaño de página es de 2 KiB.

Línea 16: En esta línea empieza a ejecutarse el programa y, por tanto, la imagen de memoria del proceso tendrá sólo el bloque de activación de main(). Véase tabla 3.2.

Línea 12: En esta línea se ha ejecutado buena parte de la función main() incluido la llamada a la función malloc(), por tanto se habrá enlazado la librería dinámica correspondiente a funciones estándar. En este momento se encuentra en mitad de una llamada a la función 'funcion()', por tanto se habrá escrito en la pila el correspondiente bloque de activación. Véase tabla 3.3.

Línea 21: En esta línea será igual que lo anterior pero habrá desaparecido de la pila el bloque de activación de la función funcion() y también se habrá liberado del Heap los 512 bytes solicitados por malloc(). Véase tabla 3.4.

Dirección	**Tamaño**	**Características**	**Contenido**		
0000.0000-0000.07FF	2 KiB	RX (Compartido y soporte en disco)	El **código** de 3.573 B utiliza dos páginas con permisos de lectura y ejecución. Queda espacio libre hasta 4 KiB		
0000.0800-0000.0-FFF	2KiB				
0000.1000-0000.17FF	2 KiB	RW (privado)	**Datos con valor inicicial**: 'b=50' y 'static e=2', solo 8 bytes		
			Datos sin valor inicial: 'int a' (línea 04), ocupa 4 bytes		
			Heap: Crecerá hacia abajo según se vaya demandando mas espacio de forma dinámica. Los datos y el Heap tienen los mismos permisos de lectura y escritura y estarán por tanto en la misma página. El Sistema Operativo asignará mas páginas al Heap según las vaya necesitando.		
....					
XXXX.F800-XX-XX.FFFF	2 KiB	RW (privado)	**Pila**	La pila crece hacia arriba	
				Variables locales 'char *f' (línea 18) 4 bytes	Bloque activación main
				Dirección de retorno y puntero marco anterior (4 + 4 bytes)	
				Parámetros (0 bytes)	
				Variables de entorno del proceso	

Tabla 3.2

Dirección	**Tamaño**	**Características**	**Contenido**
0000.0000-0000.07FF	2 KiB	RX (Compartido)	El **código** de 3.573 B utiliza dos páginas con permisos de lectura y ejecución. Queda espacio libre hasta 4 KiB
0000.0800-0000.0FFF	2 KiB		
0000.1000-0000.17FF	2 KiB	RW (privado)	**Datos con valor inicicial**: 'b=50' y 'static e=2', solo 8 bytes, pero se ocupa
			Datos sin valor inicial: 'int a' (línea 04), ocupa 4 bytes
			Heap: Crecerá hacia abajo según se vaya demandando mas espacio de forma dinámica. Los datos y el Heap tienen los mismos permisos de lectura y escritura y estarán por tanto en la misma página. El Sistema Operativo asignará mas páginas al Heap según las vaya necesitando. El **Heap habrá crecido** 512 bytes del **malloc()**.
....			
8000.0000-8006.3FFF	N·2 KiB	RX (Compartido)	**Código librería dinámica**: Se carga por el medio del mapa. En este caso se han supuesto 200 páginas (N=200). Los permisos son de lectura y ejecución y es memoria compartida. En este caso sería la librería estándar.
8006.4000-8007.CFFF	M·2 KiB	RW (privado)	**DSVI y DCVI librería dinámica:** Estos datos son privados del proceso. Se ha supuesto que se necesitan 50 páginas (M=50)
....			
FFFF.F800-FFFF.FFFF	2 KiB	RW (privado)	**Pila** — La pila crece hacia arriba Variables locales 'int d' (línea 09) (4 bytes). La variable 'e' es estática y está inicializada a un valor, por tanto se encontrará en "Datos con valor inical" — Bloque activación de funcion() Dirección de retorno y puntero marco anterior (4 + 4 bytes) — Bloque activación de funcion() Parámetro c (4 bytes) — Bloque activación de funcion() Variables locales 'char *f' (línea 18) 4 bytes — Bloque activación de main() Dirección de retorno y puntero marco anterior (4 + 4 bytes) — Bloque activación de main() Parámetros (0 bytes) — Bloque activación de main() Variables de entorno del proceso

Tabla 3.3

Dirección	**Tamaño**	**Características**	**Contenido**
0000.0000-0000.07FF	2 KiB	RX (Compartido)	El **código** de 3.573 B utiliza dos páginas con permisos de lectura y ejecución. Queda espacio libre hasta 4 KiB
0000.0800-0000.0FFF	2 KiB		
0000.1000-0000.17FF	2 KiB	RW (privado)	**Datos con valor inicicial**: 'b=50' y 'static e=2', solo 8 bytes, pero se ocupa
			Datos sin valor inicial: 'int a' (línea 04), ocupa 4 bytes
			Heap: Crecerá hacia abajo según se vaya demandando mas espacio de forma dinámica. Los datos y el Heap tienen los mismos permisos de lectura y escritura y estarán por tanto en la misma página. El Sistema Operativo asignará mas páginas al Heap según las vaya necesitando. Se han liberado los 512 bytes del malloc()
....			

8000.0000-8006.3FFF	N·2 KiB	RX (Compartido)	**Código librería dinámica**: Se carga por el medio del mapa. En este caso se han supuesto 200 páginas (N=200). Los permisos son de lectura y ejecución y es memoria compartida. En este caso sería la librería estándar.		
8006.4000-8007.CFFF	M·2 KiB	RW (privado)	**DSVI y DCVI librería dinámica:** Estos datos son privados del proceso. Se ha supuesto que se necesitan 50 páginas (M=50)		
....					
XXXX.F800-XXXX.FFFF	2 KiB	RW (privado)	**Pila**	La pila crece hacia arriba	
				Variables locales 'char *f' (línea 18) 4 bytes	Bloque activación main()
				Dirección de retorno y puntero marco anterior (4 + 4 bytes)	
				Parámetros (0 bytes)	
				Variables de entorno del proceso	

Tabla 3.4

Problema 3.6 (abril 2004)

El programa adjunto ejecuta en un sistema con memoria virtual siendo el tamaño de página de 8 KiB. Además, el montaje dinámico se realiza en el momento de carga en memoria del programa.

```
 1  #include <sys/mman.h>
 2  #include <sys/stat.h>
 3  #include <sys/types.h>
 4  #include <fcntl.h>
 5  #include <stdio.h>
 6  #include <unistd.h>
 7
 8  int v[1000];    /* Considere enteros de 4 bytes. */
 9
10  int main(void)
11  {
12  int fd, pid;
13  int length = 1024, offset = 0;
14  char * area, caracter;
15
16  fd = open("compartido", O_RDWR);
17  area = mmap(0, length, PROT_READ|PROT_WRITE, MAP_SHARED, fd, offset);
18
19  pid = fork();
20  if (pid == 0) {
21  strcpy(area, "12345");
22  caracter = area[2];
23  area[4] = '7';
24  printf("valor de area[4] = %c\n", area[4]);
25  } else {
26  area = area + 2;
27  strcpy(area, "123");
28  munmap(area-2,length);
29  caracter = area[2];
30  }
31  return 0;
32  }
```

Supóngase que:

- *El texto del programa adjunto con compilación dinámica tiene un tamaño de 5032 B, lo que incluye el código y las cadenas de caracteres.*

- *Dicho programa utiliza la biblioteca libc que tiene un texto de 123 KiB, unos datos con valor inicial de 13 KiB y unos datos sin valor inicial de 4 KiB.*
- *Por defecto el sistema operativo reserva 20 KiB para el heap, que incluye en la región de datos, y 48 KiB para la pila, incluyendo la pila inicial. Resaltamos que las regiones de datos con valor inicial, datos sin valor inicial y heap se incluyen en la correspondiente región de datos.*

a) Calcular el número de páginas que requiere la imagen de memoria del proceso que ejecuta dicho programa al iniciar la ejecución de la línea 19.

b) Calcular el tamaño de la sección de datos del fichero ejecutable. Exprese el resultado en Bytes.

Supóngase ahora que el orden de ejecución es el siguiente:

- *El proceso padre ejecuta de forma seguida hasta la línea 26 inclusive.*
- *El hijo ejecuta hasta la línea 21 inclusive.*
- *El proceso padre ejecuta la línea 27.*
- *El proceso hijo ejecuta la línea 22.*
- *El proceso padre ejecuta la línea 28.*
- *El proceso hijo ejecuta hasta terminar.*
- *El proceso padre ejecuta hasta terminar.*

c) Determinar en el proceso hijo el valor de la variable caracter una vez ejecutada la línea 22.

d) Determinar el caracter que se imprimirá al ejecutarse la línea 24.

e) Determinar en el proceso padre el valor de la variable caracter una vez ejecutada la línea 29.

Solución

a) Las regiones que tendrá la imagen de memoria del proceso son las siguientes:

- Texto o código: 5032 B, lo que requiere 1 página. Derechos de lectura y ejecución.
- Datos: No hay datos con valor inicial y hay 1000*4 B de datos sin valor incial. A ello hay que añadir los 20 KiB del heap, por lo que la región es de 24 KiB, lo que requiere 3 páginas. Derechos de lectura y escritura.
- Pila: 48 KiB, lo que requiere 6 páginas. Derechos de lectura y escritura.
- Texto de la librería: 123 KiB, lo que requiere 16 páginas. Derechos de lectura y ejecución.
- Datos de la librería: 13 KiB + 4 KiB = 17 KiB, lo que requiere 3 páginas. Derechos de lectura y escritura.
- Región compartida creada por el mmap: 1024 B, lo que requiere 1 página. Derechos de lectura y escritura.

La suma de las regiones anteriores es de 30 páginas.

b) El fichero ejecutable incluye una sección con los datos que tienen valor inicial. Dado que no hay datos con valor incial, la respuesta es 0 B.

c) El hijo es un clon del padre, por lo tanto tiene las mismas regiones de memoria que el padre y con los mismos derechos. En concreto el hijo también tiene la región creada por el mmap, y de forma compartida con el padre. El hijo escribe en los bytes 0 a 4 los valores 1, 2, 3, 4 y 5. Seguidamente el padre escribe en los bytes 2 a 4 los valores 1, 2 y 3 (sobreescribiendo el 3, 4 y 5 escrito por el hijo).

Por tanto, el valor que obtiene el hijo es un '1'.

d) El padre desproyecta la región del mmap. Esto no afecta al hijo, que sigue teniendo la región hasta que termina, puesto que no ejecuta ningún munmap. Por tanto, el resultado del printf es:

valor de area[4] = 7

e) Al ejecutar la línea 29, el padre ha desproyectado la región, por lo que no tiene derechos de acceso a esa zona de memoria. Al intentar acceder area[2] se generará un error de violación de memoria.

Problema 3.7 (junio 2004)

Dado un sistema de ficheros de tipo UNIX, se tiene el siguiente conjunto de ficheros y directorios:

Nodo-i	Dueño	Permisos	Tamaño	Agrupaciones
2	0	0755	180	21
28	100	0600	1435	35 y 40

32	100	0600	20	94
41	100	0755	180	345
50	100	0600	30	155
115	100	0755	180	87
213	100	0755	180	350
233	100	0600	30	103
241	100	0755	180	200
256	100	0600	40	330
335	100	0755	180	210

El contenido de las agrupaciones asociadas a los ficheros y directorios se muestra a continuación:

Agrupaciones	**Contenido**
21	. 2; .. 2; usr 115; home 335; etc 41
35	*00001111222233334444555566666....*
40	*1111111111111111111111111111....*
87	. 115; .. 2; fich2 233; fich3 256
94	*3333333333333333333...*
103	*22222222222222222222222222…*
155	*12345678901234567890123456789....*
200	. 241; .. 41
210	. 335; .. 2; fich4 50; fich6 28
330	*99998888777766665555444433332222111110000.....*
345	. 41; .. 2; prueba 213; prueba2 241; fich5 28
350	. 213; .. 41; fich1 32

a) Si se desea abrir el fichero ``fich1'' utilizando su nombre absoluto, ¿cuántas tablas de directorios se necesitan leer?

b) ¿Cuántos enlaces físicos tiene el directorio "/etc"?

Suponga que en este sistema tenemos 3 procesos, un proceso A, un proceso hijo del proceso A, que denominaremos proceso B, y un tercer proceso independiente, llamado C. Los 3 procesos se ejecutan con identificación efectiva de usuario 100.

A continuación, se muestra el código de los 3 procesos:

```
1  int main(void)
2  { /* Proceso A */
3     int fd2, fd3, fd3_2, fd4;
4     int leidos, escritos, longitud=2048, offset = 0;
5     char *dir_mem;
6     fd3 = creat("/usr/fich3", 0600);
7     fd4 = open("/home/fich4", O_RDWR);
8     dir_mem = mmap(0,longitud, PROT_READ|PROT_WRITE,MAP_SHARED, fd4, offset);
9     switch(fork()) {
10    case -1: perror("fork");
11       exit(1);
12    case 0: /* Proceso B */
13       fd3_2 = dup(fd3);
14       fd2 = open("/usr/fich2", O_RDWR);
15       lseek(fd3, 10, SEEK_CUR);
16       escritos = write(fd3_2, dir_mem, 5);
17       escritos = write(fd2, dir_mem, 3);
18       leidos = read(fd3, dir_mem, 11);
19       leidos = read(fd2, dir_mem, 10);
20       lseek(fd3_2, 0, SEEK_SET);
21       munmap(dir_mem,longitud);
22       return 0;
```

```
23    default:
24       leidos = read(fd3, dir_mem, 4);
25       munmap(dir_mem, longitud);
26    }
27    return 0;
28 }
```

```
/* Proceso C */
1  char buffer[20];
2  int main(void)
3  {
4     int fd2, fd3;
5     fd2 = open("/usr/fich2", O_RDONLY);
6     fd3 = open("/usr/fich3", O_WRONLY);
7     read(fd2, buffer,6);
8     write(fd3, buffer,3);
9     return 0;
10 }
```

Supóngase que:

- *El programa correspondiente al proceso A ejecuta en un sistema con memoria virtual siendo el tamaño de página de 8 KiB.*
- *El código del programa más las cadenas de caracteres ocupan 5 KiB.*
- *El programa correspondiente al proceso A se enlaza de forma estática a la biblioteca libc. Dicha biblioteca tiene un texto de 200 KiB, datos con valor inicial de 20480 Bytes y datos sin valor inicial de 10 KiB.*
- *El sistema operativo construye la zona de datos mínima para las necesidades iniciales del proceso. Resaltamos que la región de datos incluye a las regiones de datos con valor inicial y datos sin valor inicial.*
- *La pila inicial ocupa 24 KiB.*
- *Se consideran enteros de 4 Bytes.*

c) Calcular el número de páginas que requiere la imagen de memoria del proceso A al iniciar la ejecución de la línea 9.

d) Calcular el tamaño de la sección de datos del fichero ejecutable correspondiente al proceso A, expresando el resultado en Bytes.

Suponiendo que el orden de ejecución de los 3 procesos es el siguiente:

- *El proceso A ejecuta hasta la línea 9 inclusive.*
- *El proceso B ejecuta hasta la línea 20 inclusive.*
- *El proceso C ejecuta de forma completa*
- *El proceso A ejecuta hasta terminar*
- *El proceso B ejecuta hasta terminar*

e) ¿Qué valor toma la variable leidos justo después de la ejecución de la línea 18?

f) Tras finalizar todos los procesos, ¿cuál es el contenido de las 5 primeras posiciones del fichero "/home/fich4"?

SOLUCIÓN

a) Analizando el árbol de directorios correspondiente a este sistema de ficheros, observamos que el fichero "fich1" tiene como ruta absoluta "/etc/prueba/fich1", por lo que se necesitan leer 3 tablas de directorios, correspondientes a la tabla del directorio raiz, la tabla del directorio etc y la tabla del directorio prueba.

b) Observando la tabla de agrupaciones, podemos calcular el número de enlaces físicos de cualquier fichero o directorio, simplemente contando el número de referencias al nodo-i. En el caso del directorio /etc, su nodo-i es 41, y es referenciado 4 veces. Por lo tanto, el número de enlaces físicos es 4.

c) Para calcular el número de páginas que requiere la imagen de memoria del proceso A, necesitamos calcular:

- *Texto: 5 KiB (código programa + cadenas de caracteres) + 200 KiB (Biblioteca estática) = 205 KiB -> 26 páginas de 8 KiB.*
- *Datos:*
  - *Datos sin valor inicial: 10 KiB*

- *Datos con valor inicial: 20480 Bytes = 20 KiB*
- *Equivale a 30 KiB -> 4 páginas de 8 KiB*

- *Pila: 24 KiB -> 3 páginas de 8 KiB.*
- *Memoria compartida: 2048 Bytes -> 1 página de 8 KiB*

Por tanto, el número de páginas que requiere la imagen de memoria del proceso A es 34 (26+4+3+1).

d) Para calcular el tamaño de la sección de datos del fichero ejecutable, debemos calcular el número entero de páginas que dé soporte a los datos con valor inicial. En este caso, los datos con valor inicial ocupan 20 KiB, lo que requiere 3 páginas. La sección de datos del ejecutable ocupará por tanto en bytes: 3*8*1024 = 24576 bytes.

e) Debido a que fd3 utiliza modo sólo escritura (O_WRONLY), no se puede leer a partir de dicho descriptor. Por lo que la operación read devuelve -1.

f) Tras la ejecución del proceso A hasta la línea 9, el fichero /home/fich4 no tiene ningún cambio. A continuación, ejecuta el proceso B hasta la línea 20. En la línea 18 se modifica el contenido de la zona de memoria compartida. Las primeras posiciones son 0. Después ejecuta el proceso C de forma completa. Este proceso modifica el fichero /usr/fich3. El proceso A vuelve a ejecutar hasta terminar, modificando las 4 primeras posiciones de la zona de memoria compartida. Finalmente, ejecuta el proceso B hasta terminar. Lo único que hace es desproyectar la zona de memoria compartida.

De este modo, el contenido de las 5 primeras posiciones del fichero /home/fich4 al final de la ejecución queda del siguiente modo:

1. carácter del fichero 4 – 2. carácter del fichero 4 – 3. carácter del fichero 4 – 0 – 8. carácter del fichero 2.

Por tanto, las 5 primeras posiciones del fichero /home/fich4 tienen como contenido: 12302.

Problema 3.8 (septiembre 2004)

En un sistema con páginas de 4 KiB y palabras de 32 bits se tienen las tablas de páginas de la figura 3.11.

a) Compare las tablas de los dos procesos e indique las conclusiones a las que llegue.

b) Supongamos que el proceso 1 está ejecutando, que el tamaño ocupado de la pila es de 400 B y que realiza una llamada a un procedimiento que contiene exclusivamente la declaración de una matriz local de 100x10 enteros con valor inicial. Indicar los cambios que se producirían en las tablas de páginas al ejecutarse dicha llamada. ¿Qué ocurre si la matriz es de 100x100 enteros con valor inicial? ¿Y si es de 100x100 enteros sin valor inicial?

c) Supóngase que el proceso 1, partiendo de la situación de la figura, ejecuta un bucle que recorre todo su segmento 4. Justo después de ello el proceso 2 ejecuta a su vez un bucle que recorre todo su segmento 4. ¿Es probable que el proceso 2 requiera el intercambio de alguna página? Justificar la respuesta.

d) Sea ahora una máquina con palabras de 64 bits y que requiere datos alineados. Los enteros ocupan 4 bytes mientras que los double ocupan 8 bytes. Calcular el espacio real ocupado por la siguiente estructura suponiendo que el compilador no realiza ninguna optimización.

```
struct ditur {
   char b;
   double c;
   int v[5];
   double m;
   char n;
   char s;
   int r;
}
```

¿Qué optimización se podría realizar?

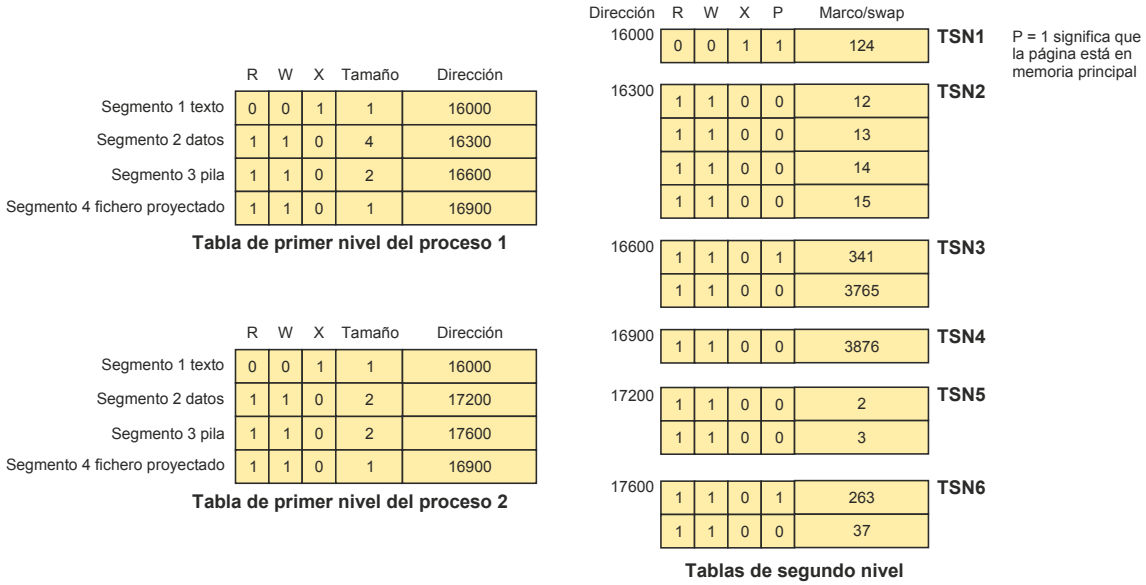


Figura 3.11

Solución

a.- A la vista de las tablas de páginas se observa que los dos procesos comparten dos regiones, la de texto y el fichero proyectado en memoria. Dado que ambos procesos tienen derechos de escritura sobre el fichero proyectado podemos asegurar que se ha proyectado de forma MAP_SHARED.

Las regiones de texto no tienen derechos de lectura. Esto significa que no pueden contener cadenas de caracteres ni constantes.

Los procesos no tienen ninguna página de datos en marcos de memoria, lo que parece indicar que se han puesto en ejecución recientemente.

Se podría pensar que uno de los procesos desciende del otro, pero el que sus regiones de datos tengan tamaños distintos no apoya dicha idea, sobre todo por lo dicho en el párrafo anterior.

b.- La llamada a un procedimiento implica asignación de memoria para los argumentos de la llamada, la dirección de retorno, el puntero al bloque de activación anterior y las variables locales. Dicha asignación se realiza en la pila, con el denominado bloque de activación. Hay que reservar espacio para 100x10 = 1.000 enteros, lo que supone 4.000B (el tamaño del entero se ha supuesto de 4 bytes), más los argumentos (que desconocemos), la dirección de retorno (4B) y el puntero de bloque (4B). Como la pila tiene ocupados 400 B llegaremos hasta 4.408 B, es decir, a algo más de 1 página. Ahora bien, se dispone de dos páginas, por lo que no es necesario incrementar la región de pila. Sin embargo, al tener la matriz valores iniciales, es necesario rellenar dicha estructura con esos valores, lo que implica que ha de ser traída a memoria principal. Por todo ello, quedará modificada la tabla TSN3, en la que se producen dos cambios: se pone a "1" el bit P de la segunda página y se pone el valor del marco asignado a la segunda página.

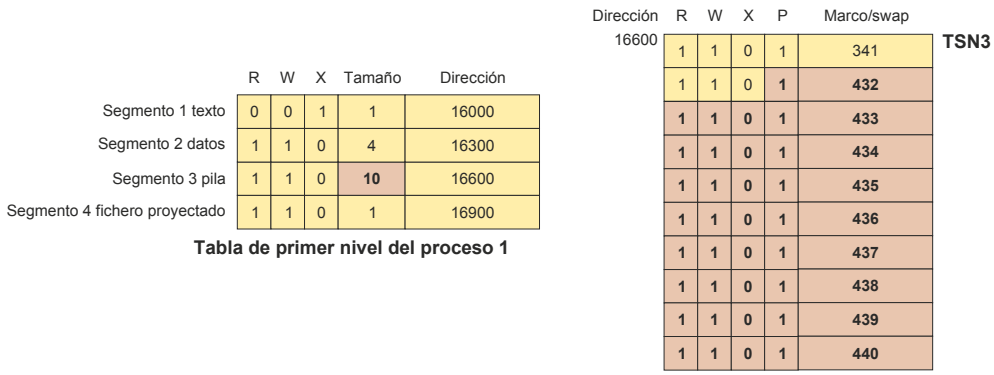


Figura 3.12

En el caso de una matriz de 100x100 = 10.000 enteros con valor inicial hacen falta 40.000 B más los argumentos, retorno y puntero de bloque, es decir, unos 40.008 B. Como están ocupados 400 B se necesita un total de 40.408 B, por lo que bastan 10 páginas (= 40.960 B) . La pila dispone de 2 páginas, por lo que es necesario aumentarla en 8 páginas. Además, hay que rellenar todas estas páginas con los valores iniciales, por lo que hay que llevarlas a marcos de memoria principal. Todo ello afecta a la tabla de primer nivel y a la tabla TSN3, que quedrían como muestra la figura 3.12, en la que se han resaltado en gris los elementos modificados (los valores de marco de página serían los que el sistema operativo seleccionase).

Para el caso de la matriz de 100x100 sin valor inicial la situación es bastante parecida a la anterior. Hay que añadir 8 páginas a la pila, pero al no tener que rellenar su contenido con valores iniciales no es obligado que se carguen en marcos de página (bastaría con dejar marcadas esas páginas como de rellenar a cero). Las tablas quedarían de forma similar a la figura , pero los bits de presente P estarían todos a “0” menos el de la primera página. Sin embargo, hay una sutileza importante:¿cuándo se entera el sistema operativo que tiene que aumentar la pila? Si el hardware dispone de control de desbordamiento de pila, en cuanto se aumenta el puntero de pila por encima de su máximo valor. Si no dispone de ese mecanismo se enterará cuando se acceda (para leer o escribir) a zonas prohibidas. En ese momento será cuando se aumente realmente la tabla de páginas.

c.- Dado que el proceso 2 ejecuta el mismo programa sobre el mismo fichero proyectado, al pasar del primer al segundo proceso nos encontraremos con que tanto la página del texto como la página del fichero proyectado estarán en sendos marcos de página. Hay que notar que el proceso 1, al hacer un bucle sobre los datos del fichero proyectado trae a un marco la página de dicho fichero. Por lo tanto, el proceso 2, mientas no realice accesos a su región de datos o incremente su pila, no requerirá ningún intercambio de página.

d.- Los caracteres ocupan un byte por lo que la distribución de las variables en memoria en una máquina que realice alineamiento de datos es la mostrada en la figura 3.13. El espacio requerido es de 7 palabras, desperdiciándose un total de 13 B (marcado en gris en la figura).

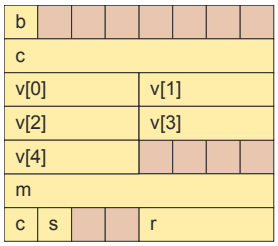


Figura 3.13

La optimización consiste en reordenar la definición de la estructura, ya sea por el programador o por el compilador. Por ejemplo se puede redefinir la estructura de la forma mostrada a continuación, por lo que la ocupación en memoria sería la de la figura 3.14 (basta con 6 palabras, perdiéndose solamente 5 bytes, que podrían llegar a ser utilizados por otra variable o estructura en caso de encajar en dicho espacio).

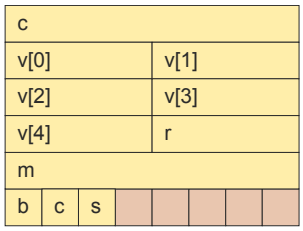


Figura 3.14

```
struct ditur {
   double c;
   int v[5];
   int r;
   double m;
   char b;
   char n;
   char s;
}
```

Problema 3.9 (abril 2005)

Sea un sistema operativo que utiliza páginas de 4 KiB, tablas de páginas de 2 niveles y regiones de texto compartidas.

Existen simultáneamente dos procesos A y B que ejecutan el mismo programa y de los que sabemos que en un instante de tiempo determinado Ta tienen la siguiente situación:

Proceso	***Región***	***Nº páginas***	***Nº páginas presentes***	***Dirección comienzo***
A	*Texto*	*24*	*7*	*0*
	Datos	*13*	*5*	2^{23}
	Texto Biblioteca dinámica Z	*45*	*12*	2^{24}
	Datos Biblioteca dinámica Z	*5*	*3*	$2^{24}+2^{23}$
	Fichero M proyectado compartido	*27*	*7*	2^{25}
	Pila	*11*	*3*	2^{30}
B	*Texto*	*24*	*7*	*0*
	Datos	*15*	*5*	2^{23}
	Fichero M proyectado compartido	*27*	*7*	2^{24}
	Texto Biblioteca dinámica Z	*45*	*12*	2^{25}
	Datos Biblioteca dinámica Z	*5*	*3*	$2^{25}+2^{23}$
	Pila	*16*	*7*	2^{30}

a) Calcular el nº total de marcos de páginas que tienen asignados entre los dos procesos en ese instante.

b) Seguidamente, A ejecuta un bucle de lectura que recorre todo el fichero proyectado. Suponiendo que no se reemplaza ninguna página de los procesos A y B, indicar el nº de fallos de página que se producen así como el total de marcos de página que tienen ahora asignados entre los dos procesos.

Tomando como referencia el instante de tiempo Ta, Indicar si se produce un error de ejecución y, en su caso, el valor de la variable v en cada uno de los procesos en los supuestos c, d, e, y f siguientes. (NOTA: recuerde que la notación C 1<<x equivale a 2^x).

*c) El proceso A ejecuta: v = *p (donde p vale 24); e inmediatamente el proceso B ejecuta el mismo trozo de código, pero ahora p vale 25. En este caso, indicar, además, si se puede producir un fallo de página considerando que las variables v de cada proceso están cada una en su correspondiente marco de página.*

*d) El proceso A ejecuta: p = (1<<24) + 1; *p = 234; v = *p; e inmediatamente el proceso B ejecuta v = p.*

*e) El proceso A ejecuta: p = (1<<24) + (1<<23) + 12; *p = 234; e inmediatamente el proceso B ejecuta p = 1<<24) + (1<<23) +*

*f) El proceso A ejecuta: p = (1<<25) + 10012; *p = 234; e inmediatamente el proceso B ejecuta p = (1<<24) + 10012; v = *p.*

Solución

a) Las regiones privadas del problema son las de datos, las de datos de las bibliotecas dinámicas y las pilas. Hemos de sumar todos los marcos de página del proceso A: 7 + 5 + 12 + 3 + 7 + 3 = 37 más los marcos de página del proceso B que no comparte con A, es decir, 5 + 3 + 7 = 15. Lo que nos da un total de 52 marcos de página.

b) Como la región del fichero proyectado tiene 27 páginas, de las que 7 ya están en marcos, el recorrer todo el fichero hará que se traigan a marcos las 27 – 7 = 20 páginas restantes, lo que produce 20 fallos de página. Después de esta bucle, entre los dos procesos se tendrán 52 + 20 = 72 marcos de página, dado que no se produce ningún reemplazo.

c) No se produce error de ejecución $v_A = 24$ y $v_B = 25$. No es posible un fallo de página. Justo después de ejecutar A, B ejecuta el mismo código, por lo que ya tiene que estar en un marco. Por otro lado v = *p con p = 24 implica un acceso a la primera página de la región de código, al igual que v = *p con p = 25, página que es compartida por los dos procesos.

d) Se produce un error al ejecutar A puesto que se intenta escribir en una región (texto de biblioteca) que no tiene derechos de escritura. Del valor v_B no podemos decir su valor, puesto que no conocemos el valor de p_B.

e) Se produce un error al ejecutar B, dado que la dirección $2^{24} + 2^{23} + 12 = 2^{24} + 2^{12} \bullet 2^{11} + 12$ no está en el mapa de memoria de ese proceso.

f) En este caso no se produce error de ejecución, siendo $v_B = 234$.

Problema 3.10 (junio 2005)

Sea un sistema operativo que utiliza páginas de 4 KiB, tablas de páginas de 2 niveles, direcciones de 4 bytes y regiones de texto compartidas.

Dado el siguiente código, correspondiente al ejecutable "ejec1":

```
#define TAM_BUF 1024
#define TAM_BUF_PROY    16384
char buffer[TAM_BUF];
char * buffer_proy;

int main(void)
{
   int fd;
   int i;
   int leidos, total;
   int pfd[2];

   fd = open("FICHERO_PROY", O_RDWR, 0666);
   buffer_proy = mmap(0, TAM_BUF_PROY, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd , 0);
   if (buffer_proy == MAP_FAILED) perror("mmap");
   pipe(pfd);

   switch (fork()){
   case -1:
      perror("fork");
      munmap(buffer_proy, TAM_BUF_PROY);
      close(fd); close(pfd[0]); close(pfd[1]);
      exit(1);
   case 0:
      read(pfd[0],&jtotal, sizeof(int));
      close(fd); close(pfd[0]); close(pfd[1]);
      write(1, buffer_proy, total);
      munmap(buffer_proy, TAM_BUF_PROY);
      return 0;
   default:
      total=0;
      while ((leidos=read(0, buffer, TAM_BUF))>0){
         for (i=0; i<leidos; i++) {
            buffer_proy[j] = buffer[i];
            total++;
         }
      }
      write(pfd[1],&total,sizeof(int));
      wait(NULL);
      close(fd); close(pfd[0]); close(pfd[1]);
      munmap(buffer_proy, TAM_BUF_PROY);
      return 0;
   }
   return 0;
}
```

Se supone que el fichero cuyo nombre es "FICHERO_PROY" existe y tiene un tamaño de TAM_BUF_PROY bytes o más.

La versión compilada correspondiente al código anterior ocupa 25200 bytes. El montaje del ejecutable utiliza una biblioteca estática, cuyo texto ocupa 52 KiB y cuyos datos sin valor inicial ocupan 12 KiB. Dicha biblioteca no tiene datos con valor inicial.

Conteste de forma razonada a las siguientes preguntas:

a) *¿Cuántos bytes ocupa el fichero ejecutable "ejec1"?*

b) *Dibuje el mapa de memoria de los dos procesos justo tras la llamada* `fork`*, indicando la disposición y tamaño de cada región del mapa y su carácter compartido o privado. Haga las suposiciones que considere convenientes.*

c) *Describir de forma resumida la funcionalidad de "ejec1".*

d) *Codifique el programa "ejec2", que, sin rescribir el código de "ejec1", sino haciendo uso de su ejecutable, imprima por la salida estándar el contenido de un determinado fichero dado como argumento, es decir:*

```
$ ./ejec2 nombre_fichero
```

e) *El programa "ejec1" tiene la limitación de que sólo permite comunicar datos con un tamaño inferior o igual a TAM_BUF_PROY. Esboce, sin programar nada, una solución que permitiese reutilizar el buffer compartido eliminando esta limitación.*

Solución

a) El fichero ejecutable estará compuesto por el texto correspondiente a *"ejec1"*, el texto de la biblioteca estática, los datos con valor inicial de "ejec1" y los datos de valor inicial correspondientes a la biblioteca estática. Es decir:

- Texto ejec1: 25200 bytes
- Texto biblioteca estática: 52 KiB = 53284 bytes
- La suma del texto es igual a 78484 bytes, que ocupan 20 páginas
- Datos v.i. ejec1: 0 bytes
- Datos v.i. biblioteca: 0 bytes

Para hacer el cálculo, se ha considerado despreciable el tamaño de la cabecera del ejecutable.

b) Justo antes de la llamada `fork()`, el proceso padre tiene la siguiente imagen de memoria:

Texto ejec1 + Texto biblioteca	Compartido (20 páginas)
Datos sin v.i. ejec1 + biblioteca	Privado (4 páginas)
Fichero proyectado	Compartido (4 páginas)
Pila	Privado (1 página)

Para calcular el número de páginas:

- Texto compartido ejec1: 25200 bytes + Texto compartido biblioteca estática: 52 KiB (53284 bytes) = 78448 bytes -> 20 páginas
- Datos con v.i.: 0
- Datos sin v.i. ejec1: 1024 bytes (buffer) + 1 dirección (4 bytes) + Datos sin v.i. biblioteca estática: 12 KiB -> 4 páginas
- Fichero proyectado: 16 KiB -> 4 páginas
- Pila: Necesitamos espacio para entorno, variables locales, etc… Tenemos 6 enteros como variables locales. Tenemos suficiente con 1 página.

Una vez ejecutada la llamada `fork()`, el proceso hijo tiene las mismas regiones, aunque pueden estar en direcciones diferentes. Si las regiones son compartidas, el proceso hijo comparte con el proceso padre dichas regiones. Si las regiones son privadas, el proceso hijo tiene una copia de dichas regiones.

c) El código tiene la siguiente funcionalidad. Un proceso crea otro proceso hijo y se comunica con él a través de un fichero proyectado en memoria y de un pipe. El proceso padre lee de la entrada estándar, escribiendo el contenido en la zona de memoria compartida. Calcula el número de caracteres escritos en la zona de memoria compartida y envía este dato a través del pipe al proceso hijo. El proceso hijo recoge el número de caracteres y escribe a través de la salida estándar la parte de la memoria compartida que ha sido escrita por el padre. Se trata de una especie de filtro,

en el cual el proceso padre lee de la entrada estándar, pasándole dicha información al proceso hijo, que lo escribe por la salida estándar.

d) Se debe redirigir la entrada estándar al fichero que se le pasa como argumento a "ejec2". El código es el siguiente

```
int main(int argc, char *argv[])
{
   if (argc != 2)
   {
      fprintf(stderr, "Error. Uso: ./ejerc2 nom_fich\n");
      exit(1);
   }
   close(0);
   open(argv[1],O_RDONLY);
   execlp("./ejerc1", "./ejerc1", NULL);
   perror("exec");
   return(1);
}
```

e) El padre tendría que detectar que ha llegado al final del buffer compartido (tamaño TAM_BUF_PROY), en cuyo caso se comunicaría con el proceso hijo para enviarle el número de datos leídos hasta ese momento. El proceso hijo leería los datos de la memoria compartida, los escribiría por la salida estándar y pasaría algún testigo al proceso padre para permitirle volver a escribir a partir del inicio de la memoria compartida. Este proceso de sincronización se tendría que realizar con algún mecanismo de sincronización, como puede ser un semáforo, otro pipe, etc.

Problema 3.11 (junio 2006)

Sea un sistema con memoria virtual que tiene un tamaño de página de 4 KiB y direcciones de 4 bytes. Dado el siguiente código:

```
struct datos {
   int a;
   char b;
   int c[2048];
};

struct datos varDat;
char varChar = 'c';

int main (int argc, char *argv[])
{
   static int varInt = 5;
   int fd;
   int i=0;
   char *p, *q;
   struct stat bstat;

   if (argc != 2) {
      printf("Error. Uso: ./ejec nombre_fichero");
      return 1;
   }

   fd = open(argv[1], O_RDONLY);
   if (fd < 0)
   {
      perror("open");
      return 1;
   }
   fstat(fd, &bstat);
   p = mmap(NULL, bstat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
   close(fd);
```

```
   varDat.a = varInt;
   varDat.b = varChar;

   q = p;
   for (i=0; i< bstat.st_size; i++)
   {
      varDat.c[i] = *q;
      q++;
   }

   munmap(p, bstat.st_size);
   funcBib(varDat);
   return 0;
}
```

Teniendo en cuenta los siguientes supuestos:

- *Esta arquitectura requiere alineación de datos.*
- *`sizeof(int)` es igual a 4 bytes.*
- *Existe una única región de datos donde se incluyen los datos con valor inicial y sin valor inicial.*
- *El código utiliza una biblioteca dinámica, donde se encuentra la función funcBib. El montaje se lleva a cabo al invocar el procedimiento y se supone que dicha biblioteca no se ha cargado previamente por ningún otro proceso.*
- *El código del ejecutable ocupa 4020 bytes y el código de la biblioteca dinámica 24 KiB.*
- *Los datos con valor inicial de la biblioteca dinámica ocupan 6 KiB. No tiene datos sin valor inicial.*
- *La tabla de símbolos del fichero ejecutable ocupa 4 KiB.*
- *Se reservan 4 páginas para la pila inicial. Se supone que el sistema operativo no tiene que ampliar esta región en ningún momento de la ejecución.*
- *La proyección del fichero (uso de mmap) se lleva a cabo de forma correcta.*

Responder a las siguientes preguntas:

*a) ¿Cuál es el número de páginas ocupado por todas las regiones del mapa de memoria del proceso justo después de ejecutar **`munmap(p, bstat.st_size)`**?*
b) ¿Cuántas variables del código forman parte de los datos con valor inicial?
c) ¿Cuál es el tamaño en bytes que ocupan los datos sin valor inicial en la región de datos del proceso?
d) ¿Qué tipo de error se puede producir en la ejecución de la sentencia `varDat.c[i] = *q`?

Solución

a) Las regiones que tiene el proceso junto después de desproyectar el fichero son las siguientes:

Región de código: 4020 bytes (la biblioteca es dinámica), lo que ocupa 1 página

Región de datos:

- Datos c.v.i.: 1 variable entera (la variable estática) y 1 variable de tipo char: 4 bytes + 4 bytes (éstos últimos por alineación de datos)
- Datos s.v.i: 1 variable de tipo struct datos, que ocupa: 4 bytes del campo entero 4 bytes del campo char (por alienación de datos) 2048*4 del vector de enteros

En total la región de datos ocupa 3 páginas.

Región de pila: 4 páginas

En total, 8 páginas

Nota: Hay que tener en cuenta que todavía la biblioteca dinámica no ha sido montada y el fichero ya se ha desproyectado.

Por supuesto, la tabla de símbolos no forma parte del mapa de memoria del proceso como una región independiente.

b) 2, la variable entera estática *varInt* y la variable de tipo char *varChar*.

c) Ya se ha calculado para responder a la pregunta a):

- 4 bytes del campo entero

- 4 bytes del campo char (por alienación de datos)
- 2048*4 del vector de enteros

Total: 8200 bytes

d) Se puede dar un desbordamiento de un dato múltiple (el vector *varDat.c*) si el tamaño del fichero es superior a 2048.

Problema 3.12 (junio 2008)

Un sistema operativo con memoria virtual sigue el modelo clásico UNIX con una región de datos que engloba los datos estáticos y los dinámicos (heap). Además, asigna por defecto un heap de, al menos, 16 KiB y una pila de 32 KiB. El tamaño de página es de 4 KiB.

Mediante el mandato size obtenemos los siguientes valores:

```
# size mi_programa
      text    data     bss       dec      hex      filename
     12096    1080    8724     21900     558C      mi_programa
# size mi_biblio
      text    data     bss       dec      hex      filename
    645024   14720   13424    658448    A0C10      mi_biblio
```

a) Dibujar el mapa de memoria del proceso que ejecuta mi_programa al comienzo de la ejecución, considerando que el montaje de las bibliotecas se hace al invocar una de sus funciones. Indicar para cada región el espacio que ocupa, los permisos de acceso y el contenido. Marcar los huecos existentes.

b) mi_programa incluye una línea que utiliza la función mi_funcion incluida en la biblioteca dinámica mi_biblio. Dibujar el mapa de memoria del proceso justo después de la llamada a mi_funcion, de igual forma que en el caso anterior.

c) mi_programa incluye las siguientes líneas de código:

```
p = mmap(NULL, 8192, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
q = p;
n = fork();
if (n == 0) {
   for (i = 0; i < 2; i++){*q = 1; q++;}
   m = *p;
   ......
}else{
   for (i = 0; i < 2; i++){*q = 3; q++;}
   r = *p;
   ......
}
```

Indicar el valor que tendrán las variables m y r después de que el padre y el hijo hayan ejecutado completamente las sentencias for.

*d) mi_programa tiene definida como global la variable int *p; Además tiene dos threads que ejecutan las líneas de código siguientes:*

Thread 1

```
p = mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
for (i = 0; i < 2; i++){*p = i; p++;}
```

Thread 2

```
for (j = 0; j < 5; j++){*p = j+5; p++;}
```

Suponiendo que primero el thread 1 ejecuta completamente las dos líneas indicadas y luego el thread 2 completa la ejecución del for, indicar el contenido de la región generada con el mmap.

e) mi_programa realiza montaje explícito de la biblioteca mis_numeros, utilizando la función mi_maximo, que se define en dicha biblioteca. Esta función recibe dos números enteros y devuelve otro número entero. Escriba un fragmento de código que utilice dicha función mi_maximo.

Solución

a) La región de texto, como toda región en un sistema con memoria virtual, se compone de un número entero de páginas. En este caso el texto requiere 3 páginas, puesto que 3x4 KiB = 12.288 B > 12.096. Esta región es compartida, de tamaño fijo, tiene permisos de lectura y ejecución, y su fuente es el ejecutable. Quedan sin usar 192 B al final de la última página.

La región de datos incluye los datos con valor inicial más los datos sin valor inicial más el heap, es decir, 1.080 B + 8.724 B + 16 KiB = 9.804 B + 16 KiB. Se requieren, por tanto 7 páginas. El heap quedará de 7x4 KiB - 9.804 B = 18.868 B. Esta región es privada, de tamaño variable (el heap puede crecer), tiene permisos de lectura y escritura. La fuente de los datos con valor inicial es el ejecutable, para el resto es rellenar a ceros.

La pila incluirá las variables de entorno más la pila inicial y tendrá un tamaño de 32 KiB = 8 páginas. Esta región es privada, de tamaño variable, tiene permisos de lectura y escritura y su fuente es rellenar a ceros.

La figura 3.15 muestra la imagen de memoria del proceso. Dependiendo del tipo de segmentos que permita el sistema, la región de datos puede o no estar pegada a la de texto. En la figura se ha dejado un hueco, puesto que es el caso más general, dado que el texto es de tamaño fijo, este hueco no tiene por objeto que la región pueda crecer. Entre heap y pila sí existirá un hueco, para que el heap pueda crecer.

Figura 3.15

b) La biblioteca dinámica requiere dos regiones la de texto y la de datos. El tamaño de la región de datos de la biblioteca es de 645.024 B, por lo que son necesarias 158 páginas = 647.168 B. Esta región es compartida, de tamaño fijo, tiene permisos de lectura y ejecución, y su fuente es el fichero de la biblioteca.

Para datos son necesarios 14.720 B + 13.424 B = 28.144 B. Son necesarias 7 páginas = 28.672 B. Esta región es privada, de tamaño fijo, tiene permisos de lectura y escritura. La fuente de los datos con valor inicial es el fichero de la biblioteca, mientras que para los datos sin valor inicial es rellenar a ceros.

La figura 3.16 muestra la imagen de memoria del proceso. Como se ha indicado anteriormente, el hueco entre texto y datos puede no existir, dependiendo del sistema. Lo que siempre existirá será el hueco para que pueda crecer el heap y la pila.

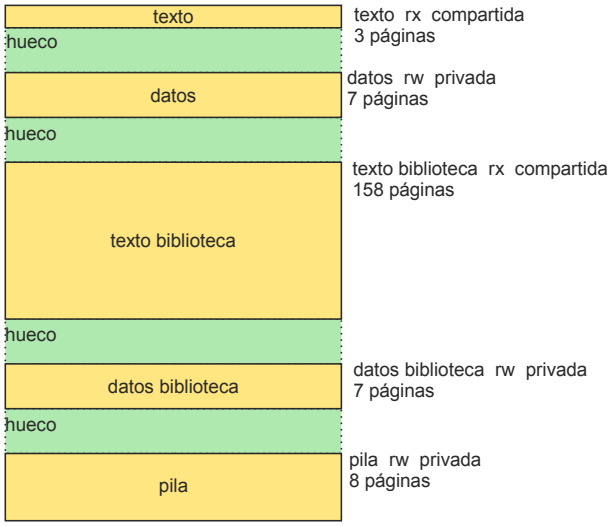


Figura 3.16

c) Tanto el padre como el hijo escriben en la región compartida, por lo que los valores de la región dependen del orden de ejecución. En relación con el contenido final de la región se podrían dar los siguientes casos:

11.....
33.....
13.....
31.....

siendo los dos primeros más probables que los dos últimos, que exigen más cambios de contexto.

En cualquier caso, no se puede determinar si el valor de r y m será un 1 o un 3.

d) Ahora nos especifican el orden de ejecución, por lo que podemos determinar el resultado. Dado que p es una variable global, los dos threads pueden acceder a la región de memoria.

Al final del primer bucle la región contiene: 01....

Al final del segundo bucle la región contiene: 0156789....

e) El código propuesto es el siguiente:

```
void *h;
int (*max) (int,int);

h = dlopen("/lib/mis_numeros.so", RTLD_LAZY);
max = dlsym(h,"mi_maximo");
printf("%d\n", (*max)(3,5));
También sería correcto poner:
printf("%d\n", max(3,5));
```

Problema 3.13 (junio 2009)

Sea un sistema con las siguientes características:

- *Ancho de palabra de 32 bits.*
- *Memoria virtual con direcciones de 32 bits, de tipo segmentado paginado. Se dedican 9 bits para especificar el segmento.*
- *Se utilizan páginas de 1 KiB.*
- *El segmento de datos incluye los datos con y sin valor inicial además del heap.*
- *Los programas son de montaje dinámico, realizándose el montaje de una biblioteca cuando se utiliza por primera vez una función de la misma.Observe que malloc y mmap están en libc y que cos está en libm.*
- *Se van asignando los segmentos 0, 1, 2, 3, ... en el orden en que se van creando los segmentos del proceso.*
- *Suponer que los enteros (int) son de 4 bytes.*
- *Las funciones malloc, printf, etc. están en la biblioteca dinámica libc, y la función cos en la biblioteca dinámica libm.*

prog1.c

```
int *f(void)
{
  int miv[2000]; int i; int *ret;
  for (i = 0; i < 2000; i++) miv[i] = i;
  ret = miv;
  return ret;
}
char vect[3*1024];
int a1 = 5; int a2 = 15; int a3 = 23;
int main(void)
{
  int *w; int *v; int *m; int des; int af; int i;
  w = malloc(4*1024);
  v = f()+7;
  af = *v;
  des = creat("dat.txt", 0640);
  ftruncate(des, 12*1024 +16);
  m = mmap(0, 12*1024+16, PROT_READ|PROT_WRITE,MAP_SHARED, des, 0);
```

```
        close(des);
        m[0] = (int)m;
/*A*/   for (i = 1; i < (3*1024+4); i++) m[i] = i;
        printf("%f\n", cos(2));
/*B*/   .
        .
        return 0;
    }
```

prog2.c

```
    int main(void)
    {
        int *n; int *m; int des; int fr; int p; int* q; int i;
        fr = creat("prep.txt", 0640);
        ftruncate(fr, 8*1024);
        n = mmap(0, 8*1024, PROT_READ|PROT_WRITE, MAP_SHARED, fr, 0);
        close(fr);
        for (i = 0; i < 2*1024; i++) n[i] = i+200;
        des = open("dat.txt", O_RDONLY);
        m = mmap(0, 12*1024, PROT_READ, MAP_SHARED, des, 0);
        close(des);
        q = (int*)(m[0] + 4);
        p = *q;
        .
        .
        return 0;
    }
```

a) Determine de forma exacta el número máximo de páginas que puede tener cada segmento.

b) Determine en bytes el tamaño mínimo exacto que puede tener el segmento de datos de prog1 justo antes del mmap.

c) Determinar el valor que adquiere la variable af del prog1. Indique si considera razonable utilizar dicho valor.

d) Suponiendo que primero ejecuta el prog1 hasta la Línea B y seguidamente ejecuta el prog2, determinar el valor que adquiere la variable p de este ´ultimo programa, indicando si es razonable utilizar dicho valor.

*e) Suponiendo que la Línea A fuese: for (i = 1; i <(3*1024+100); i++) m[i] = i; ¿Cuál sería el efecto de la ejecución de dicha línea?*

*f) Teniendo en cuenta los siguientes datos: Tamaño de texto de prog1 = 2500 B. Tamaño de texto de libc = 148*1024+35 B. Tamaño de datos de libc = 945 B. Tamaño de texto de libm = 136*1024 + 563 B. Tamaño de datos de libm = 10*1024 + 75 B. Representar la imagen de memoria del proceso prog1 cuando está ejecutando la línea B. Identificar cada región e indicar su tamaño y características.*

Solución

a) La dirección de 32 bits se descompone en 9 bits para el segmento, 13 bits para la página del segmento y 10 bits para el byte de la página. Por tanto habrá $2^{13} = 8.192$ páginas como máximo en un segmento.

b) Primero hay que destacar que el tamaño ha de ser un múltiplo de página. Tenemos 3x4 bytes de datos con valor inicial (variables a1, a2 y a3), 3 páginas de datos sin valor inicial (vector vect) más el heap de, al menos, 4 páginas. El total es de 8 páginas = 8.192 B como mínimo.

c) La variable af toma el valor 7. La función f devuelve un puntero con la dirección del vector miv. A ese puntero se le suma 7, por lo tanto, v apuntará al séptimo elemento del vector, que tiene un valor = 7, valor que es asignado a la variable af.

Hay que notar que el vector miv ya no existe como tal, puesto que sólo es válido dentro de la función f(), pero su contenido sigue estando presente en la pila, puesto que no se ha invocado ninguna otra función. Eso significa que efectivamente af tomará el valor 7, pero claramente se está cometiendo un error al usar el valor de una variable obsoleta. Por tanto, NO debe utilizarse dicho valor.

d) La variable p tomará el valor 201. Para llegar a esa conclusión hemos de considerar lo siguiente: dado que las regiones se crean por orden en los segmentos 0, 1, 2 …, las regiones del proceso prog1 cuando ha ejecutado hasta la Línea B son las siguientes:

- Segmento 0 Texto, que empieza en la dirección 0.
- Segmento 1 Datos, que empieza en la dirección $1*2^{13}*2^{10}$.

- Segmento 2 pila, que empieza en la dirección $2*2^{13}*2^{10}$.
- Segmento 3 Texto libc, que empieza en la dirección $3*2^{13}*2^{10}$.
- Segmento 4 Datos libc, que empieza en la dirección $4*2^{13}*2^{10}$.
- Segmento 5 Proyección dat.txt, que empieza en la dirección$5*2^{13}*2^{10}$.
- Segmento 6 Texto libm, que empieza en la dirección $6*2^{13}*2^{10}$.
- Segmento 7 Datos libm, que empieza en la dirección $7*2^{13}*2^{10}$.

Por tanto, la dirección de m será $5*2^{13}*2^{10}$.

Por su lado, el proceso de prog2 cuando esté ejecutando la línea p = *q tiene las siguientes regiones (hay que tener en cuenta que las rutinas de llama al sistema operativo están en la libc, por lo que esta se carga con el creat):

- Segmento 0 Texto, que empieza en la dirección 0.
- Segmento 1 Datos, que empieza en la dirección $1*2^{13}*2^{10}$.
- Segmento 2 pila, que empieza en la dirección$21*2^{13}*2^{10}$.
- Segmento 3 Texto libc, que empieza en la dirección $3*2^{13}*2^{10}$.
- Segmento 4 Datos libc, que empieza en la dirección $4*2^{13}*2^{10}$.
- Segmento 5 Proyección prep.txt, que empieza en la dirección $5*2^{13}*2^{10}$.
- Segmento 6 Proyección dat.txt, que empieza en la dirección $6*2^{13}*2^{10}$.

El programa prog2 obtiene $m[0] = 5*2^{13}*2^{10}$, puesto que comparte la región dat.txt con el prog1, y éste ha puesto ese valor al comienzo de la región. Ahora bien, en prog2, esta dirección corresponde a la región de la proyección prep.txt. Por tanto, m[0] + 4 corresponderá a la dirección del segundo dato entero de la región prep.txt, entero que tiene el valor 201.

Nótese que si las regiones se asignasen con otro criterio, la dirección $5*2^{13}*2^{10}$ podría corresponder a otra región o, incluso, no estar asignada al proceso prog2. Por tanto, es claro que es un error de programación y que no debería en ningún caso utilizarse dicho valor.

e) La nueva línea se ejecuta sin problemas, puesto que la región tiene 13 páginas, lo que significa que caben 13x1024/4 = 3.328 enteros y el bucle solamente llega hasta el 3.172. Se escriben en memoria los 3.172 enteros. Como la proyección es más pequeña (3x1024 + 4 = 3076), los 96 enteros sobrantes constituyen basura.

f) Las regiones son siempre múltiplo de página, por lo que daremos su tamaño en páginas. La imagen se refleja en la tabla adjunta.

Segm.	**Título**	**Tamaño (páginas)**	**Permisos**	**Tamaño**	**Compartido/ privado**	**Fuente**
0	Texto	3	rx	Fijo	Compartido	Fichero
1	Datos	8	rw	Variable	Privado	Fichero y rellenar 0
2	Pila	8	rw	Variable	Privado	Inicial y rellenar 0
3	Texto libc	149	rx	Fijo	Compartido	Fichero
4	Datos libc	1	rw	Fijo	Privado	Fichero y rellenar 0
5	Proyección dat.txt	13	rw	Fijo	Compartido	Rellenar 0 (fichero vacío)
6	Texto libm	137	rx	Fijo	Compartido	Fichero
7	Datos libm	11	rw	Fijo	Privado	Fichero y rellenar 0

El tamaño de la región de pila viene determinado fundamentalmente por la llamada a la función f(), dado que dicha función crea un vector de 2000 enteros más otros dos enteros en la pila. Esto supone que necesita 2002x4 = 8.008 B. Como 8 páginas son 8.192 B, es posible que los 184 bytes adicionales sean suficientes para el entorno y el resto de los bloques de activación de main y f. La pila tendrá asignada una página adicional, marcada de solo-escritura, para avisar de que se reqiere más pila.

Problema 3.14 (septiembre 2009)

Sea un sistema con páginas y bloques de e/s de 1 KiB y con agrupaciones de 4 KiB. El sistema tiene dos discos D1 y D2 con tiempos medio de acceso de 15 y 12 ms respectivamente. Consideraremos que el SO consume 1 ms cada vez que entra a ejecutar y que los enteros son de 32 bits.

El programa prog1 es ejecutado con el argumento `/home/pepe/fich1`, formando el proceso P1. El fichero `fich1` tiene un tamaño de 2.000 B y se encuentra en D1.

a) Analice el código prog1 e indique si añadiría algo al mismo, indicando qué y dónde.

b) Suponiendo que el nodo_i del directorio raíz está en memoria y que no se producen errores, indicar la secuencia de accesos a disco que es necesario realizar para ejecutar el servicio open de P1.

c) Suponiendo que el proceso P1, que ejecuta prog1, es el único proceso en el sistema y que en instante t0 su ejecución se encuentra justo terminado el `mmap`, dibujar el cronograma de ejecución hasta su finalización. Considerar que el `printf` se realiza sobre el monitor, por lo que no existe tiempo de espera sobre el periférico. (Reflejar en el cronograma solamente los tiempos de la función `Funcion1`, del SO y de la e/s).

d) Supondremos, ahora, que el único proceso es el P2, que ejecuta prog2 sobre un fichero de 1.000 B que se encuentra en D1. Considerando que en el instante t0 su ejecución se encuentra justo terminado el bucle for, dibujar el cronograma de ejecución hasta su finalización. (Reflejar en el cronograma solamente los tiempos de la función `Funcion2`, del SO y de la e/s).

e) De forma similar, el proceso P3 ejecuta prog2 sobre un fichero de 3.000 B que se encuentra en D2. Dibujar el cronograma a partir del bucle, como en el caso anterior.

f) Ahora supondremos que los tres procesos P1, P2 y P3 están activos y que P1 es más prioritario que P2 y éste más que P3. Supondremos que en el instante t0 se encuentran en los puntos de ejecución indicados en las preguntas anteriores. Representar el correspondiente cronograma.

```c
/*prog1*/
int main(int argc,char *argv[])
{
   int i,desc,size;
   int*ptr;
   struct stat st;
   double total=0.0;

   desc=open(argv[1],O_RDONLY);
   fstat(desc,&st);
   size=st.st_size;
   ptr=mmap(0,size,PROT_READ, MAP_SHARED,desc,0);
   Funcion1();    /* Funcion1 usa 7 msecs de CPU */
   size=size/sizeof(int);
   for(i=0;i<size;i++)
      total=total+ptr[i];
   printf("suma = %g\n",total);
   return 0;
}
/*prog2*/
int main(int argc,char *argv[])
{
   int i,desc,size;
   char*buff;
   struct stat st;

   desc=open(argv[1],O_WRONLY);
   fstat(desc,&st);
   size=st.st_size/2;
   buff=malloc(size);
   for(i=0;i<size;i++)
      buff[i]=(char)i;
   write(desc,buff,size);
   free(buff);
   close(desc);
   Funcion2();    /* Funcion2 usa 5 msecs de CPU */
   return 0;
}
```

Solución.-

a) En el código `prog1` se echa en falta las dos cosas siguientes:

- La devolución de los recursos que ya no son necesarios. En concreto, debería haber un close después del `mmap` y debería haber un `munmap` antes del `printf`.
- El tratamiento de los errores, que debe ir detrás de cada llamada a un servicio del SO. Por ejemplo, si el `mmap` falla, los accesos `ptr[i]` darán violación de memoria.
- *Comprobar* el argumento de llamada.

b) Los accesos al disco, para abrir el fichero `/home/pepe/fich1`, son los siguientes:

Leer el directorio raíz.
Leer el nodo_i del directorio `home`.
Leer el directorio `home`.
Leer el nodo_i del directorio `pepe`.
Leer el directorio `pepe`**.**
Leer el nodo_i del fichero `fich1`.

Dependiendo del tamaño de dicho directorio se puede necesitar más de un acceso. Aquí consideraremos que los directorios son pequeños, por lo que ser necesita un solo acceso para su lectura.

c) El cronograma se encuentra en la figura 3.17. Es de destacar que el acceso a ptr[i] produce dos fallos de página, puesto que recorren 2.000 B.

d) El cronograma se encuentra en la figura 3.17. Con el `write` se escriben 1.000/2 = 500 B, lo que afecta a un solo bloque, que no se escribe entero. Hay que leer primero el bloque y luego hay que escribirlo.

e) El cronograma se encuentra en la figura 3.17. Con el `write` se escriben 3.000/2 = 1.500 B, lo que afecta a dos bloques. El primer bloque no hace falta leerlo, porque se escribe entero. El segundo sí hay que leerlo. Teniendo en cuenta que los dos bloques afectados pertenecen a la primera agrupación del disco, significa que son contiguos, por lo que basta con una operación de escritura conjunta para los dos bloques.

f) El cronograma se encuentra en la figura 3.17.

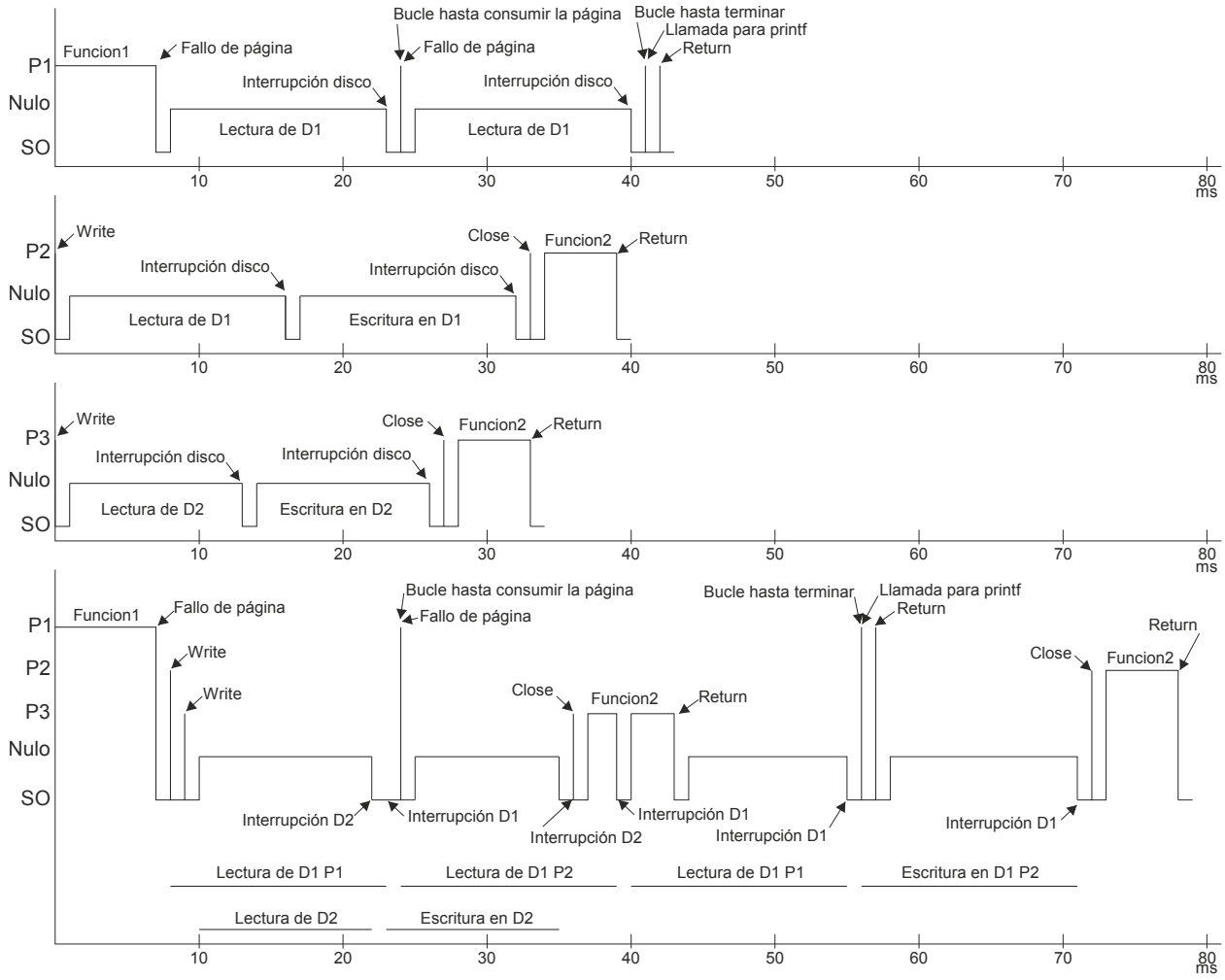


Figura 3.17

Problema 3.15 (junio 2010)

Sea un sistema con las siguientes características:

- *Memoria virtual con direcciones de 32 bits.*
- *Se utilizan páginas de 4 KiB y se realiza la asignación de espacio de disco en bloques de 4 KiB.*
- *Los discos están preformados con sectores de 1 KiB.*
- *Los datos de tipo int ocupan 4 bytes y los de tipo short 2 bytes.*
- *El mapa de direcciones de la imagen de memoria del proceso comienza en la dirección 0, y finaliza en la 1 GiB-1.*

Se desea ejecutar una aplicación que está compuesta por los siguientes módulos:

- *Fichero cabecera.h: contiene declaración de variables globales*
- *Fichero proyecta.c: contiene el código fuente de una función que está incluida en una biblioteca libproy y que se invoca desde el programa principal*
- *Fichero fuente.c: contiene el código fuente de la aplicación*
- *Fichero datos.txt: contiene datos*
- *Los ficheros de cabecera donde se declaran las funciones del lenguaje y los servicios del sistema utilizados por la aplicación (no mostrados en el código).*

El contenido de cada uno de los ficheros es el siguiente:

Fichero datos.txt:

```
abcdefghijklmnopqrstuvxyz0123456789
```

Fichero cabecera.h:

```
int a, b=10, c=15, d;
struct tipo_datos{
   char a;
   char b;
   short c;
   int d;
   char e[5];
}
```

Fichero proyecta.c:

```
#include "cabecera.h"
#include <sys/mman.h>
char *proyecta(char * nombre) {
   int fd;
   char *t=NULL;
   fd = open (nombre, O_RDWR);
   t=mmap (0, sizeof(struct tipo_datos), PROT_READ|PROT_WRITE, MAP_SHARED, fd,
   0);
   close(fd);
return (t);
}
```

Fichero fuente.c:

```
#include "cabecera.h"
#include <stdio.h>
#include <stdlib.h>
extern char * proyecta(char * nombre);
char literal[36]="abcdefghijklmnopqrstuvxyz0123456789";
int main (void) {
   char * aux;
   int fd=0, i=5;
   aux=malloc(sizeof(struct tipo_datos));
   struct tipo_datos p;
   printf("p.a %c, p.b %c, p.c %d, p.d %d, p.e %d, p.e[4] %c\n",p.a, p.b, p.c,
   p.d, (int) p.e, p.e[4]);
   /* Punto A */
   aux=proyecta("datos.txt");
   p.c=(short) c;
   p.d=b;
   printf("p.a %c, p.b %c, p.c %d, p.d %d, p.e %d, p.e[4] %c\n", p.a, p.b,
   p.c, p.d, (int) p.e, p.e[4]);
   /* Punto B */
   for (i=3;i<6;i++)
   aux[i]= literal[i+22];
   munmap(aux,sizeof(struct tipo_datos));
   return 0;
   /* Punto C */
}
```

Se supone que el fichero ejecutable fuente y la biblioteca librproy estructuran su contenido de la siguiente forma:

	Texto	***Datos con valor inicial***	***Datos sin valor inicial***
fuente	*35 KiB*	*¿?*	*¿?*
libproy	*165 KiB*	*16 KiB+¿?*	*27 KiB+¿?*

donde ¿? se corresponde con el valor deducido de la información disponible en los módulos del enunciado.

La biblioteca se montará dinámicamente al invocar el procedimiento.

Conteste a las siguientes preguntas.

a) Describir las regiones de la imagen de memoria del proceso al iniciar la ejecución de la aplicación, y en los Puntos A y B, indicando su contenido, niveles de protección de la misma, dirección de comienzo y tamaño de la región en páginas en memoria.

b) Indicar los valores impresos en los Puntos A y B.

c) Indicar el contenido del fichero "datos.txt" en los Puntos A y B.

Solución

a) Comienzo de la ejecución:

- Región de código: Código, R-X, 0 - $35 \cdot 2^{10}$-1, Tam. memoria: 9 páginas.
- Región de datos con valor inicial (DVI): variables b, c y literal: 44 bytes, RW-, $36 \cdot 2^{10}$ – $36 \cdot 2^{10}$+43 bytes, Tam. memoria: 1 página.
- Región de datos sin valor inicial (DSVI): variables a y d: 8 bytes, RW-, $40 \cdot 2^{10}$ – $40 \cdot 2^{10}$+7 bytes, Tam. memoria: 1 página.
- Heap: vacío, RW-, $44 \cdot 2^{10}$ - , Tam. memoria: 0 o el que asigne por defecto el montador.
- Pila (direccionamiento en sentido descendente): Var. de entorno: X bytes, var. locales de la función main (fd, i, p, aux): 25 bytes, retorno main (4 bytes) RW-, 2^{20}-1 – 2^{20}-X-30, Tam. memoria: 1 página (si X+29 < 4096 bytes).

Punto A:

Tanto la pila -invocación de funciones- como la región de datos –actualización de variables- son regiones que se han utilizado hasta llegar al Punto A, pero por el código propuesto, la única región que ha sufrido una modificación es el heap si se ha supuesto que inicialmente estaba vacío. El resto de regiones no varían respecto de la situación inicial.

- Heap: se reservan 13 bytes diseccionables desde la variable aux, RW-, $44 \cdot 2^{10}$ – $44 \cdot 2^{10}$ +12, Tam. memoria: 1 página

Punto B:

Mismo comentario que en el Punto A aplicable a todas las regiones que existían en ese momento.

- Región de código de la biblioteca *libproy*: Código, R-X, Y – Y+$168 \cdot 2^{10}$ –1, Tam. memoria: 42 páginas. (Y es una posición de memoria situada entre la pila y el heap).
- DVI de la biblioteca *libproy*: variables b y c más las variables con valor inicial definidas en otros módulos de la biblioteca: 16 KiB + 8 bytes, RW-, Y+$168 \cdot 2^{10}$ – Y+$184 \cdot 2^{10}$ +7, Tam. memoria: 5 páginas.
- DVSI de la biblioteca *libproy*: variables a y d más las variables con valor inicial definidas en otros módulos de la biblioteca: 27 KiB + 8 bytes, RW-, Y+$188 \cdot 2^{10}$ – Y+$215 \cdot 2^{10}$+7, Tam. memoria: 7 páginas.
- Región del fichero proyectado: 13 primeros bytes del fichero "datos.txt", RW-, Z-Z+13,Tam. Memoria: 1 página. (Z es una posición de memoria situada entre la pila y la DVSI de la biblioteca *libproy*).
- Pila: misma situación que en el Punto A, más el bloque de activación de la función proyecta. Se supone que esta acción no modifica el tamaño de la región.

b) Punto A: la variable p es una variable local de la función main y está sin inicializar, por lo tanto se almacena en la pila. Se puede considerar que la variable contendrá basura porque el programador no tiene ningún control sobre el contenido inicial de las posiciones donde se ha almacenado. Hay que recordar que el lenguaje C no es un lenguaje interpretado, como puede serlo Java, por lo que las variables locales se alojan en la pila en el bloque de activación de la función aunque su declaración se haya realizado en una zona donde ya hay sentencias de código. En el campo "p.e" se imprimirá la dirección de la variable p.e.

Punto B: Idem que en el Punto A salvo para: p.c 15, p.d 10

c) En ninguno de los dos puntos se llega a modificar el contenido del fichero. Sólo se ha modificado en el Punto C.

Problema 3.16 (junio 2011)

Sea un sistema de memoria virtual con páginas de 4 KiB y con 4 GiB de memoria principal y 8 GiB de swap.

Suponiendo que el SO residente ocupa 1 GiB, indicar el mayor tamaño que podría tener la imagen de memoria de un proceso para los dos casos siguientes:

a. Con preasignación de swap

b. Sin preasignación de swap

Sea un sistema con memoria virtual y con páginas de 2 KiB, donde el SO mantiene una sola región con los datos con y sin valor inicial más el heap. Un proceso tiene 512 B de datos con valor inicial, 1.300 B de datos sin valor inicial y 2 KiB de heap. Supondremos que el primer dato sin valor inicial es: char car[50]. Dicho proceso ejecuta el siguiente bucle:

```
for (i=0 ; i<10000 ; i++) {
   car[i] = car[i+1];
}
```

c. ¿Cuál es el último valor alcanzado por i?

d. ¿Por qué causa se alcanza dicho valor?

SOLUCIÓN

a) Con preasignación de swap significa que toda página tiene su ubicación en swap. Por tanto, un único proceso que tuviese todo el swap tendría una imagen de memoria de 8GiB.

b) Sin preasignación de swap significa que cada página reside en memoria o en swap. Por tanto, un único proceso que tuviese todo el swap y la memoria libre tendría una imagen de memoria de 8 GiB + 3 GiB = 11 GiB, dado que el SO ocupa 1 GiB de memoria.

c y d) La región de datos más heap tiene que ocupar 2 páginas, es decir 4 KiB. Las posiciones relativas dentro de la región son desde la 0 hasta la 4.095. En la primera ejecución del bucle (i = 0) se acceden a la primera y segunda posición de car[], es decir, a las posiciones 512 y 513 de la región, dado que primero están los datos con valor inicial que ocupan desde la 0 hasta la 511.

El bucle ejecutaría 10.000 veces a menos que se produzca un error. Efectivamente, se producirá un error de acceso a memoria (segmentation fault) cuando se intente acceder a la posición 4.096, puesto que nos habremos salido de la región y, por tanto, la MMU detectará que se está accediendo a una dirección no asignada al proceso. Por lo tanto, se recorrerán las direcciones desde la 512 hasta la 4.095. El intento de acceso a la posición 4.096 es la que genera el error. Con i = 0 se accede a las posiciones 512 y 513. Con i = 4.095 - 512 = 3.583, se accede a la posición 4.095 y se intenta la 4.096, produciéndose el error.

Problema 3.17 (feb 2011)

Los procesos siguientes aparecen en la cola del planificador en un determinado momento:

Trabajos	***Unidades de Tiempo***	***Prioridad***
1	*8*	*1*
2	*5*	*4*
3	*2*	*2*
4	*7*	*3*

Los procesos llegan en el orden 1, 2, 3, 4 y la prioridad más alta es la de valor 1.

Se pide :

a) Obtener los diagramas de tiempo que ilustren la ejecución de estos procesos usando:

1- Planificación de prioridades no expulsiva

2- Round Robin (quantum = 2)

3- FCFS (First Come First Served)

b) Calcular cual es el algoritmo de planificación con menor tiempo medio de espera (de todos los procesos).

c) Para la traza formada por las páginas: 4, 2, 1, 5, 4, 2, 3, 4, 2, 3, 4, 2, 1, 5, 3

Calcular el funcionamiento de los algoritmos de reemplazamiento FIFO y LRU con 3 y 4 marcos de página, obteniendo el número de fallos de página producidos en cada uno de los cuatro casos. Comentar los resultados obtenidos.

d) Suponiendo que:

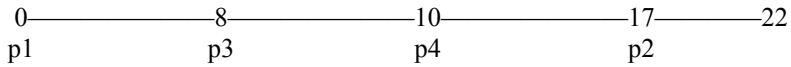
- *Los 4 procesos de la tabla siguen ejecutando una vez cumplida la rodaja de tiempo asignada en la tabla.*
- *El modelo de memoria se basa en memoria virtual.*
- *El gestor de memoria utiliza la técnica del primer ajuste para asignar huecos libres.*
- *En t=50 ut. el proceso 2 proyecta en memoria un fichero denominado 'datos.txt'.*

Indicar las regiones de memoria involucradas en los procesos recogidos en la tabla anterior, señalando los niveles de protección de las mismas.

Solución

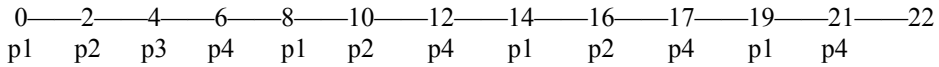
a) Diagramas de tiempo.

1- Planificación con prioridades no expulsiva.



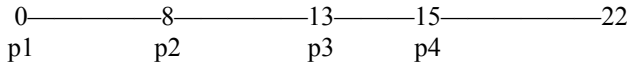
Tiempo medio espera = (0+8+10+17)/4 = 8,75 u.t.
Tiempo medio respuesta = (8+10+17+22)/4 = 14,25 u.t.

2- Planificación con Roun Robin (quantum = 2)



Tiempo medio espera = (13+12+4+15)/4=11 u.t.
Tiempo medio respuesta = (21+17+6+22)/4=16,5 u.t.

3- Planificación FCFS



Tiempo medio espera = (0+8+13+15) / 4 = 9 u.t.
Tiempo medio respuesta = (8+13+15+22) /4 = 14.5 u.t.

b) Se observa que el caso con menor tiempo de espera es el de prioridades no expulsiva. Este tiempo resulta parecido al del FCFS. En realidad el ejemplo es muy sencillo y limitado para sacar ninguna conclusión general válida, puesto que sólo considera cuatro procesos que aparecen todos en el mismo instante.

c) 1) FIFO y 3 marcos de página.

PET	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M1	4	4	4	5	5	5	3	3	3	3	3	3	3	3	3
M2		2	2	2	4	4	4	4	4	4	4	4	1	1	1
M3			1	1	1	2	2	2	2	2	2	2	2	5	5
FP	P	P	P	P	P	P	P						P	P	

2) FIFO y 4 marcos de página.

PET	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M1	4	4	4	4	4	4	3	3	3	3	3	3	3	5	5
M2		2	2	2	2	2	2	4	4	4	4	4	4	4	3
M3			1	1	1	1	1	1	2	2	2	2	2	2	2
M4				5	5	5	5	5	5	5	5	5	1	1	1
PP	P	P	P	P			P	P	P				P	P	P

Comentario: Anomalía de Belady.

3) LRU y 3 marcos de página.

M1	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M2	4	4	4	5	5	5	3	3	3	3	3	3	1	1	1

M3		2	2	2	4	4	4	4	4	4	4	4	4	5	5
M4			1	1	1	2	2	2	2	2	2	2	2	2	3
PP	P	P	P	P	P	P	P						P	P	P

4) LRU y 4 marcos de página.

M1	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3
M3		2	2	2	2	2	2	2	2	2	2	2	2	2	2
M4			1	1	1	1	3	3	3	3	3	3	3	5	5
M5				5	5	5	5	5	5	5	5	5	1	1	1
PP	P	P	P	P			P						P	P	P

Comentario: Con FIFO la tasa de fallos decrece al aumentar el número de marcos de página. Con LRU no.

d) Como no nos dan más datos, se supone que cada proceso (1, 2, 3, 4) dispondrá de su región de código (R-X), datos con valor inicial (RW-), datos sin valor inicial (RW-), heap (RW-) y pila (RW-). El proceso 2 proyecta un fichero en memoria, por lo que habrá una nueva región específica para este fichero (RW-). Muy probablemente, también permanezcan en memoria las regiones de código y datos correspondientes a las bibliotecas que contienen los servicios del sistema y las funciones del lenguaje que utilizan los mencionados procesos.

Problema 3.18 (feb-2011)

Sean dos SO, A y B que tienen un sistema de memoria segmentado (NO paginado). En la implementación de los servicios ***mmap*** *y* ***munmap*** *el SO A emplea la siguiente política:*

- *El gestor de memoria utiliza la variable DA, que originalmente apunta a la primera dirección libre después de la región de datos sin valor inicial.*
- *Al atender un servicio* ***mmap*** *con tamaño de X bytes, se concede el espacio lógico delimitado por DA y DA + X -1, y se modifica DA de la siguiente forma: DA ← DA + X.*
- *Al atender un servicio* ***munmap*** *se intenta recuperar el espacio liberado, como se indica en la figura adjunta, en la que se parte de la situación (1) y se llega a las situaciones (2) y (3) al ejecutarse en distinto orden los* ***munmap*** *1 y 2. Es decir,* ***DA se decrementa*** *para reutilizar el espacio liberado.*

Por su lado el SO B también utiliza la variable DA, que originalmente apunta a la primera dirección libre después de la región de datos sin valor inicial. Igualmente, al atender un servicio ***mmap*** *con tamaño de X bytes, se concede el espacio lógico delimitado por DA y DA + X – 1, y se modifica DA de la siguiente forma: DA ← DA + X. Sin embargo, a diferencia del caso anterior, este SO* ***no decrementa*** *nunca DA.*

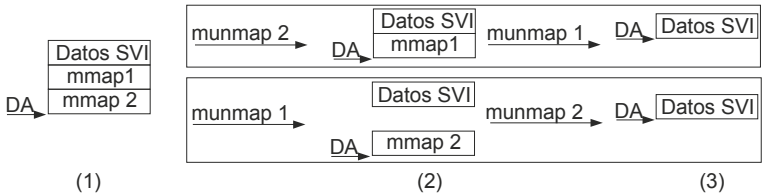


Figura 3.18

Sea el código

```
/* P1 */
1  struct datos {
2      int a;
3      int b;
4      int c;
5      int v[10];
6  }
7  int main (void) {
8      struct datos *p = NULL;
9      struct datos *t = NULL;
10     int fd, i;
11     fd = open("compartido", O_RDWR);
12     p = mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
13     close(fd);
```

```
14    p->a = 7;
15    p->b = 527;
16    p->c = 9;
17    for (i = 0; i <10; i++)
18          p->v[i] = i*i;
19    printf("%d %d %d %d %d\n", p->a, p->b, p->c, p->v[0], p->v[1]); //X
20    munmap(p, 1024);
21
22    fd = open("/dev/zero", O_RDWR);
23    t = mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
24    close(fd);
25    t->a = 1024;
26    for (i = 0; i <10; i++)
27          t->v[i] = 100-i;
28    printf("%d %d %d %d %d\n", t->a, t->b, t->c, t->v[0], t->v[1]);
//Y
29    printf("%d %d %d %d %d\n", p->a, p->b, p->c, p->v[0], p->v[1]);
//Z
30    munmap(t, 1024);
31  }
```

```
/* P2 */
1  struct datos {
2     int v[10];
3     int a;
4     int b;
5     int c;
6  }
7  int main (void) {
8     struct datos *p = NULL;
9     int fd;
10    fd = open("compartido", O_RDWR);
11    p = mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
12    close(fd);
13    printf("%d %d %d %d %d\n", p->a, p->b, p->c, p->v[0], p->v[1]);
//W
14    munmap(p, 1024);
15  }
```

a) Para el caso de SO A indicar los valores impresos por cada uno de los printf X,Y,Z y W para la siguiente secuencia de ejecución: P1 de línea 1 a línea 19 inclusive, P2 completo y P1 de línea 20 al final.

b) Para el caso de SO A indicar los valores impresos por el printf W en el caso de la siguiente secuencia de ejecución: P1 de línea 1 a línea 21 inclusive, P2 completo y P1 de línea 22 al final.

c) Para el caso de SO A indicar los valores impresos por el printf W en el caso de la siguiente secuencia de ejecución: P1 de línea 1 a línea 28 inclusive, P2 completo y P1 de línea 28 al final.

d) Para el caso de SO A indicar los valores impresos por el printf W en el caso de la siguiente secuencia de ejecución: P1 completo y después el P2 completo.

e) Repetir los casos a, b c y d para el SO B.

Solución

a) Los valores impresos son los siguientes, de acuerdo al orden de impresión:

	Impresión	**Comentario**
X	7 527 9 0 1	
W	49 64 81 7 527	El proceso P2 comparte la memoria con P1. Obsérvese que las estructuras utilizadas en P1 y P2 para acceder a la memoria son distintas
Y	1024 0 0 100 99	Se ha supuesto que los valores no inicializados están a 0.
Z	1024 0 0 100 99	Dado el funcionamiento del SO A, el segundo mmap devuelve el mismo valor que el primero, por lo que el puntero p tiene, en este caso, el mismo valor que el t. Esto significa que con p se accede a la zona de memoria del segundo mmap.

b), c) y **d)**

	Impresión	Comentario
W	49 64 81 7 527	En estos tres casos el valor impreso en W es el mismo, puesto que P2 proyecta el fichero previamente modificado por P1.

e)

	Impresión	Comentario
X	7 527 9 0 1	
W	49 64 81 7 527	El proceso P2 comparte la memoria con P1.
Y	1024 0 0 100 99	
Z		En este caso el puntero p apunta a direcciones que ya no son válidas, por lo que al intentar la lectura de p->a se produce un intento de violación de memoria. El SO matará al proceso, por lo que no se imprime nada más.

f), g) y **h)**

	Impresión	Comentario
W	49 64 81 7 527	En estos tres casos el valor impreso en W es el mismo, puesto que P2 proyecta el fichero previamente modificado por P1.

i) No sería factible plantear este caso en un sistema paginado pues las direcciones de comienzo de las regiones deben ser múltiplos del tamaño de la página del sistema.

Problema 3.19 (junio 2011)

a) Detalle el código necesario para que un proceso P1 acceda a un fichero de texto denominado 'texto.txt' mediante el mecanismo de proyección en memoria. El proceso debe invertir el orden de los bytes del fichero, pasando el primer byte a ser el último y el último se convierte en el primer byte. Cuando el proceso ha completado la reordenación, libera esta región de memoria, termina, y el fichero 'texto.txt' habrá quedado reordenado. Durante la ejecución se debe permitir que otros procesos puedan acceder simultáneamente a la región de memoria proyectada.

b) Indique qué le sucede al fichero 'texto.txt' si el proceso P1 lo proyecta en memoria en modo privado. Explique porqué.

c) Detalle el código necesario para que un proceso P2 copie el contenido del fichero 'texto.txt' sobre otro fichero denominado 'backup.txt' garantizando que son finalmente idénticos. Como puede haber otros procesos en ejecución accediendo al fichero 'texto.txt', el proceso P2 debe garantizar, utilizando cerrojos como mecanismo de sincronización, que este fichero no cambia mientras que dura la copia. Del mismo modo, debe garantizar que el fichero 'backup.txt'

d) Suponiendo que el proceso P2 cierra el descriptor de fichero asociado al fichero 'backup.txt' sin haber finalizado aún el proceso de copia, ¿es posible que otros procesos sobreescriban este fichero mientras continua la ejecución del proceso P2? En caso de contestación afirmativa, explique por qué.

Solución

a)

```
/* Proceso P1 */
int main(int argc, char ** argv) {
   int fd; char *p; char *org; char *pos_final;
   int i; struct stat bstat; char aux;
   fd=open("texto.txt", O_RDWR);          //Abre fichero
   fstat(fd, &bstat);                     //Averigua longitud del fichero
   /* Se proyecta el fichero */
   org=mmap(NULL, bstat.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd,0);
   close(fd);                             //Cierra el descriptor del fichero
   p=org;
   pos_final=org+bstat.st_size-1;
   for (i=0; i<(bstat.st_size/2); i++) { //Bucle de acceso
      aux=*p;
      *p=*pos_final;
      *pos_final=aux;
      p++;
```

```
      pos_final--;
   }
   munmap(org, bstat.st_size);          //Se elimina la proyección
   return 0;
}
```

b) Al proyectar P1 en modo privado, las modificaciones que se realizan sobre la región de memoria correspondiente a la proyección del fichero 'texto.txt' no se reflejan en el fichero original.

```
/* Proceso P2 */
#define TAM_BUFFER 4096;
int main(int argc, char ** argv) {
   int fd_ent;
   int fd_sal;
   char buffer[TAM_BUFFER];
   int bytes;
   int cod_error;
   fd_ent=open("texto.txt", O_RDONLY);  //Abre fichero de entrada
   fd_sal=creat("backup.txt",0644);     //Crea el fichero de salida
   cod_error=flock(fd_ent, LOCK_SH);    //Se bloquea el cerrojo sobre el fichero de entrada
   if (cod_error == -1){
      perror("flock texto.txt");
      close(fd_ent);
      close(fd_sal); return 0;
   }
   cod_error=flock(fd_sal, LOCK_EX);    //Se bloquea el cerrojo sobre el fichero de salida
   if (cod_error == -1){
      perror("flock backup.txt");
      flock(fd_ent, LOCK_UN);
      close(fd_ent);
      close(fd_sal); return 0;
   }
   while ((bytes =read(fd_ent, &buffer, TAM_BUFFER)) > 0){
      if (write(fd_sal, buffer, bytes) < bytes) {
         perror("Escritura");
         close(fd_ent);
         close(fd_sal); return 0;
      };
   }
   if (bytes < 0) {
      perror("Lectura");
      close(fd_ent);
      close(fd_sal); return 0;
   }
   flock(fd_ent, LOCK_UN);
   flock(fd_sal, LOCK_UN);              //Se liberan los cerrojos
   close(fd_ent);
   close(fd_sal); return 0;
}
```

d) Al cerrarse el descriptor de fichero, se pierden los cerrojos asociados a ese descriptor, por lo que el fichero se podría sobreescribir por otros procesos.

Problema 3.20 (julio-2011)

Sea una aplicación compuesta por tres tipos de procesos, que se crean de forma independiente y que se comunican mediante una región de memoria compartida. La memoria compartida está formada por 30 registros de 512 bytes. Cada registro está formado por dos cadenas de caracteres C1 de 64 bytes y C2 de 448 caracteres.

Los procesos P1i generan un dato de tipo D1, buscan un registro libre y rellenan la cadena C1 del mismo. Un registro está libre si su primer byte $C1_0$ contiene el valor 32 (lo que corresponde a un blanco " ").

Los procesos P2j buscan un registro con dato D1 y sin dato D2, leen el dato D1 y generan otro dato de tipo D2 que deben almacenar en el mismo registro del que leyeron el dato D1, ocupando la cadena C2. El primer byte de la cadena C2 de un registro sin dato D2 contiene el valor 32.

Los procesos P3k buscan un registro completo (es decir, la pareja de datos D1-D2) para realizar su función. Por tanto, los procesos P3k deberán seleccionar registros que no tengan el valor 32 en el primer byte de sus cadenas C1 y C2. Una vez leído el registro, el proceso P3k pondrá dichos bytes 0 a 32 para indicar que queda totalmente libre.

Cuando un proceso no encuentre un registro con el que trabajar, se quedará bloqueado esperando a que cambie esta situación.

Existe un proceso inicial PA cuya misión será crear aquellos elementos necesarios para que funcionen correctamente los procesos de tipo P1, P2 y P3. Lo primero que hacen dichos procesos al arrancar es comprobar que existen los elementos necesarios creados por PA. En caso de no encontrarlos terminan con un 1.

El sistema en el que ejecutan dichos procesos es de tipo UNIX con memoria virtual segmentada paginada y con páginas de 2 KiB.

a) Codificar para cada tipo de proceso PA, P1, P2 y P3 el establecimiento de la región de memoria compartida.

Consideraremos tres funciones de búsqueda Acceso1, Acceso2 y Acceso3, que usarán, respectivamente, los procesos de tipo P1, P2 y P3 para comprobar que un registro cumple las condiciones necesarias para que el correspondiente proceso pueda trabajar con él. Dichas funciones se utilizarán para encontrar registros con los que trabajar. El prototipo de estas funciones es el siguiente:

```
int Accesoi (struct registro *r);   // Devuelve 0 si registro no está disponible.
```

b) Codificar la función Acceso3, indicando si se puede producir un problema de carrera.

c) Codificar las operaciones de acceso al registro seleccionado que realiza el proceso de tipo P2.

El programa P1 se compila dinámicamente dando el siguiente resultado

 # size P1

```
   text      data       bss       dec       hex   filename
  14096       480      1024     15600      3CF0    P1
```

Dicho programa utiliza la biblioteca dinámica `libdin`

 # size libdin

```
   text      data       bss       dec       hex   filename
   3814      2148      2352      8314      207A    libdin
```

Además, el SO al cargar un programa le asigna, por defecto, una pila de 8 KiB y un heap de unos 3 KiB.

d) Mostrar la tabla de páginas del proceso de tipo P1 justo después de su carga, antes de haber ejecutado, así como después de haber creado y almacenado el primer dato D1 (suponer que no ha ejecutado ningún otro proceso hasta ese momento).

e) Indicar el valor real que tendrá el heap inicialmente.

f) Seleccionar el tipo o los tipos de mecanismos de sincronización que considera más adecuados para esta aplicación, indicando las razones que justifican su selección. Definir cada mecanismo que se piensan utilizar, indicando la función que cumple cada uno.

g) Diseñar las secciones críticas de los programas PA, P1, P2 y P3.

Solución

a) El proceso PA crea la región a compartir

```
struct registro {
   char C1[64];
   char C1[448];
};
struct registro *p;
fd=open("/home/pepe/apli", O_CREAT|O_TRUNC|O_RDWR, 0640); //Crea fichero
ftruncate(fd, 30*512);     //Tamaño para los 30 registros de 512 B
p=mmap(NULL, 30*512, PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0);
close(fd);
```

Todos los procesos de tipo P1, P2 y P3 leen y escriben de la región compartida.

```
struct registro *p;
fd=open("/home/pepe/apli", O_RDWR); //Abre fichero
p=mmap(NULL, 30*512, PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0);
close(fd);
```

Este código debería se completado con los tratamientos de error en los servicios. Por ejemplo, si un proceso Pi arranca antes de que complete el PA su secuencia puede no tener el fichero /home/pepe/apli disponible, o su tamaño puede ser erróneo por lo que el mmap daría error.

b) Una posible solución sería la siguiente:

```
int Acceso3 (strut registro *r) {
   if ((r->C1[0] == ' ') || (r->C2[0] == ' '))  {
      return = 0;
   else
      return = 1;
    }
}
```

Condiciones de carrera:

- Condición de carrera de procesos de tipo P3 con procesos de tipo P1. No puede haber, puesto que no se cumple la condición de r.C2[0] distinto de " ".
- Condición de carrera de procesos de tipo P3 con procesos de tipo P2. Se puede evitar si los procesos P2 escriben C2 de forma que el último carácter que se escriba sea el C2[0].
- Condición de carrera entre procesos de tipo P3. Puede haber si ambos leen C1[0] y C2[0] antes de seguir avanzando.
- Es de notar que los procesos de P1 presentan condición de carrera entre ellos, así como los de tipo P2.
- Se puede evitar la condición de carrera entre procesos de tipo P1 y P2 haciendo que el último carácter que escriban los P1 sea el C1[0].

c) Una posible solución sería la siguiente:

```
j=0;
while (j == 0) {
   for (i = 0; i < 30; i++){     //Recorremos todos los registros
      if (Acceso2(&p[i]) == 1){  //Encontramos registro a tratar
         j=1;
         break;
      }
   }                             //Si se completa el if volvemos a repetir
}
<<obtemos p[i].C1 con el que se genera la cadena X de tipo C2>>
strcpy(p[i].C2, X);              //Se copia la cadena X en el registro i.
```

La solución planteada tiene dos problemas: Presenta problemas de carrera y realiza espera activa, cuando no hay registros. Estos problemas se deben resolver con mecanismos de sincronización.

d) Supondremos que las librerías dinámicas se cargan en el momento de cargar el programa. En caso contrario las regiones no aparecerían hasta que se utilizase la librería. Las regiones son las siguientes:

- Texto 7 Páginas
- Datos + heap (480 + 1024 + 3·1.024 = 4.576 B) 3 páginas
- Pila 4 páginas
- Texto librería 2 páginas
- Datos librería (2.148 + 2.352 = 4.500 B) 3 páginas

Es de destacar que ninguna página está en memoria, con excepción de la parte de la pila con el entorno y los parámetros de llamada del main. La figura 3.19 muestra la tabla de páginas así como el mapa de memoria con la imagen de memoria del proceso.

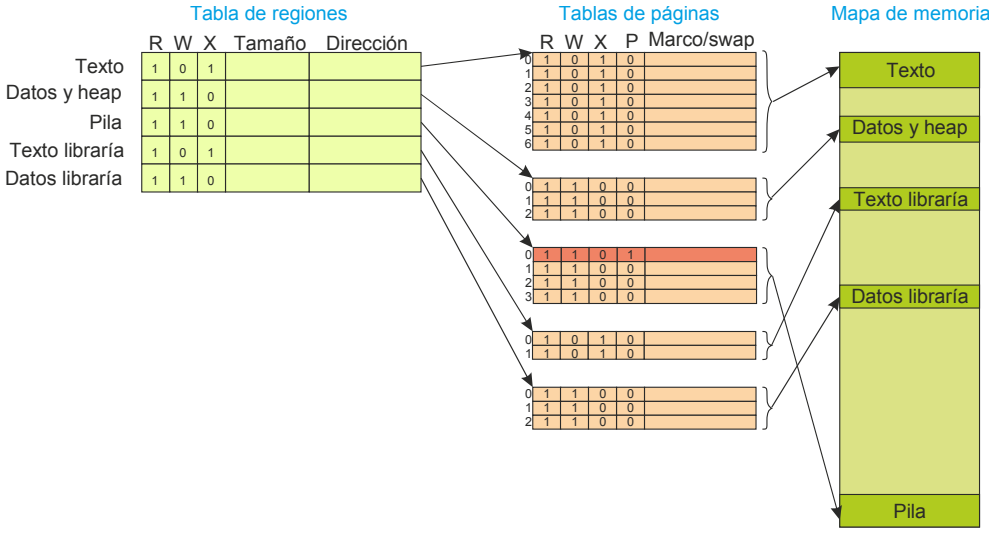


Figura 3.19

Antes de haber almacenado el primer dato se proyecta el fichero, por lo que aparece otra región de 30*512 = 15 KiB, es decir de 8 páginas. Por lo tanto, se crea una nueva entrada en la tabla de regiones y se añade una tabla de páginas de 8 entradas. Al haber ejecutado el programa, parte de las páginas de texto y datos estarán presentes en memoria.

e) Los tamaños que tenemos para datos y heap son: 480 + 1024 + 3*1024 = 4576 B. Se tienen que asignar 3 páginas, por lo que el heap quedará de 3*2*1.024 - (480 + 1024) = 4.640 B.

f) Como se trata de procesos no podemos usar los mutex, al comunicarse por memoria común tampoco tiene sentido intentar usar cerrojos de fichero. Por lo tanto se utilizarán semáforos con nombre (nos dicen que los procesos se crean de forma independiente, por lo que no están emparentados). Necesitamos un semáforo por registro que garantice el acceso exclusivo al mismo.

Existe un problema con los procesos de tipo P2, puesto que toman un registro y leen C1 para generar C2. La generación de C2 puede llevar mucho tiempo, por lo que el registro puede estar tomado mucho tiempo. Esto puede llevar a una alta contención, si se permite que los procesos de tipo P1 o P3 se queden esperando a registros tomados por un proceso de tipo P2.

g) El proceso PA que crea la región de memoria común, así como los mecanismos de sincronización, debe ejecutar previamente a los otros procesos y no necesita sincronizarse con ellos.

Supondremos las funciones siguientes

```
int GeneraVectorSemaforos(char *nombre, int n, sem_t *misem[]);
```

Función que genera n semáforos con los nombres nombre1, nombre2, etc., e inicializados a 1. Esta función la ejecuta el proceso PA para crear los semáforos.

```
int AbreVectorSemaforos(char *nombre, int n, sem_t *misem[]);
```

Función que que abre n semáforos con los nombres nombre nombre1, nombre2, etc. Esta función la ejecutan los procesos de tipo P1, P2 y P3 para abrir dichos semáforos.

Se puede modificar el código del apartado c), utilizando el semáforo que protege cada registro, quedando el código siguiente:

```
sem_t * misem[30];
AbreVectorSemaforos("MisSemaforos", 30, misem);
j=0;
while (j == 0) {
   i=0;
   for (i = 0; i < 30; i++){      //Recorremos todos los registros
      sem_wait(misem[i]);
      if (Acceso2(&p[i]) == 1){  //Encontramos registro a tratar
         j=1;
         break;
      }
      sem_post (misem[i]);
      i++;
```

```
   }                                    //Si se completa el if volvemos a repetir
}
<<obtemos r.C1 con el que se genera la cadena X de tipo C2>>
strcpy(p[i].C2, X);                     //Se copia la cadena X en el registro.
sem_post (misem[i]);
```

El problema de esta solución es la alta contención que produce. En cuanto se encuentra un registro tomado, se queda el proceso bloqueado. Esto es especialmente preocupante dado que los procesos de tipo P2 pueden tener tomado un registro un tiempo apreciable.

Una mejora consiste en hacer la comprobación primero sin semáforo, y, en caso de éxito, repetirla con semáforo. Con ello disminuimos la contención, pero no la eliminamos del todo:

```
sem_t * misem[30];
AbreVectorSemaforos("MisSemaforos", 30, misem);
j=0;
while (j == 0) {
   i=0;
   for (i = 0; i < 30; i++){          //Recorremos todos los registros
      if (Acceso2(&p[i]) == 1){
         sem_wait(misem[i]);
         if (Acceso2(&p[i]) == 1){  //Encontramos registro a tratar
            j=1;
            break;
         }
         sem_post (misem[i]);
      }
      i++;
   }                                  //Si se completa el if volvemos a repetir
}
<<obtemos r.C1 con el que se genera la cadena X de tipo C2>>
strcpy(p[i].C2, X);                   //Se copia la cadena X en el elemento C2 del registro.
sem_post (misem[i]);
```

La solución planteada sigue teniendo el problema de la espera activa cuando no hay un registro libre del tipo requerido por el proceso. Una forma de mitigar este problema sería añadiendo un sleep después del for. El tiempo del sleep se debería establecer teniendo en cuenta el tiempo durante el cual los procesos de tipo P2 retienen su registro.

Problema 3.21 (mayo 2011)

Complete el código siguiente para recorrer el fichero `caracteres.txt` proyectándolo en memoria con el servicio `mmap`. En el fichero `caracteres.txt` se tiene almacenada una secuencia de letras que no se repiten. Además, durante el recorrido del fichero deberá comprobar si en la biblioteca dinámica `simbolos.so` existe una variable definida con el mismo nombre que la letra leída en `caracteres.txt`. Para ello, el alumno debe enlazar la biblioteca explícitamente en tiempo de ejecución. Por último, indique al final de la ejecución el número de búsquedas realizadas con éxito.

```
int main(int argc, char ** argv) {
int fd; char *p; char *org; void *handle; char *error;
double *variable; int i; int cuenta=0; struct stat bstat;
char nombre[2]; // Almacena la letra leída y el carácter '\0'
fd=open("caracteres.txt", O_RDONLY)); //Abre fichero
// ...
// Bucle de acceso donde se buscan las variables y se lleva la
// cuenta de las búsquedas exitosas
// ...
     else cuenta++;
// ...
printf("%d\n", cuenta);
   return 0;
}
```

Solución

```
int main(int argc, char ** argv) {
int fd; char *p; char *org; void *handle; char *error;
double *variable; int i; int cuenta=0; struct stat bstat;
char nombre[2]; // Almacena la letra leída y el carácter '\0'

fd=open("caracteres.txt", O_RDONLY)); //Abre fichero
fstat(fd, &bstat);            //Averigua longitud del fichero
/* Se proyecta el fichero */
org=mmap(NULL, bstat.st_size, PROT_READ, MAP_PRIVATE, fd,0);
close(fd);                    //Cierra el descriptor del fichero
handle = dlopen ("simbolos.so", RTLD_LAZY); //Abre la biblioteca
if (!handle) {
fprintf (stderr, "%s\n", dlerror());
return 1;
}
nombre[1]= "\0";
p=org;
for (i=0; i<bstat.st_size; i++, p++) //Bucle de acceso
     {
     nombre[0]=*p;
variable = dlsym(handle, nombre);   //Se busca la variable
if ((error = dlerror()) != NULL)  {
fprintf (stderr, "%s\n", error);
return 1;
}
     else cuenta++;
   }
dlclose(handle);          // Se cierra la biblioteca
munmap(org, bstat.st_size);          //Se elimina la proyección
printf("%d\n", cuenta);
   return 0;
}
```

Problema 3.22 (septiembre 2011)

Dado el fichero `/home/fich` con un tamaño inicial de 20 KiB, se desea ejecutar sobre un sistema UNIX el siguiente fragmento de código:

```
struct point {
   int x;
   int y;
};
struct rect {
   struct point pt1;
   struct point pt2;
};
struct point *pp;
struct rect *pr;
void main(void)
{
int fd;
struct stat bstat;
int i;
void *p;
int pid;
... /* Código de sincronización */
fd = open ("/home/fich", O_RDWR);
fstat(fd, &bstat);
p= mmap((c_addr_t) 0,bstat.st_size,PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
close(fd);
pid=fork();
if (pid =! 0){   /* Código del padre */
```

```
   pr = p;
   for ( i=0; i<=MAX_IT; i++){
      ... /* Código de sincronización con el hijo*/
      pr->pt1.x = 7*i;
      pr->pt1.y =2*i;
      pr->pt2.x = 3;
      pr->pt2.y = 9*i;
      pr++;    /* Avanza hasta el inicio de la siguiente estructura */
      ... /* Código de sincronización con el hijo*/
      }
   }else{   /* Código del hijo */
   pp = p;
   for ( i=0; i<=MAX_IT; i++){
      ... /* Código de sincronización con el padre*/
      printf("coordenada x:%d,coordenada y:%d\n",pp->x, pp->y)
      pp++;   /* Avanza hasta el inicio de la siguiente estructura */
      ... /* Código de sincronización con el padre*/
      }
   }
}
```

Si el código de sincronización hace que ejecute primero el proceso padre, después el proceso hijo y así sucesivamente, se pide:

a) Indicar qué valores escribe el proceso hijo por pantalla durante las cinco primeras iteraciones. Justificar razonadamente la respuesta.

b) Escribir el código para realizar la sincronización necesaria utilizando el mecanismo de semáforos.

c) Si se tiene un sistema que crea una región independiente para el fichero proyectado, ¿qué ocurre si MAX_IT vale 2000? Suponga que un entero ocupa 4 bytes.

Solución

a) En el código del programa se proyecta el fichero /home/fich en memoria como una zona de datos compartida. De esta forma, tanto el padre como el hijo están accediendo a la misma zona de memoria y los cambios que realice cualquiera de ellos serán vistos por el otro. Además hay que tener en cuenta que el padre está accediendo al fichero proyectado con una estructura diferente a la del hijo, y que esta estructura tiene exactamente el doble de tamaño.

Los valores escritos en pantalla por el hijo son los siguientes:

Iteración	Valor 1	Valor 2
1	0	0
2	3	0
3	7	2
4	3	9
5	14	4

b) No es posible realizar la implementación con un solo semáforo, ya que entonces no se podría tener control total sobre la ejecución del padre y del hijo y no se podría realizar la alternancia necesaria. Es por tanto necesario utilizar dos semáforos, uno para controlar la ejecución del padre y otro para controlar la ejecución del hijo. El código quedaría de la siguiente manera:

```
/* Inicialización de los semáforos */
sem_t sem_padre, sem_hijo;
sem_init (&sem_padre, 1, 1); /* El padre es el primero en ejecutar */
sem_init (&sem_hijo, 1,0); /* A la espera de que el padre le de paso */
   /* Código del padre */
   for (...){
      sem_wait(sem_padre);
      …
      sem_post(sem_hijo);
   }
   /* Código del hijo */
```

```
   for (...){
      sem_wait(sem_hijo);
      …
      sem_post(sem_padre);
   }
/* Liberamos los recursos utilizados */
sem_destroy(&sem_padre);
```

sem_destroy(&sem_hijo);

c) Si el número máximo de iteraciones es 2000 implica que el padre estaría intentando escribir fuera de la imagen del proceso, pues el fichero ocupa 20 KiBytes (20.480 bytes) y el padre intenta acceder a partir de la iteración 1281 a una posición de la región proyectada que se encuentra fuera del rango direccionable por el proceso. La MMU al realizar la traducción de la dirección errónea generaría una excepción de *referencia a memoria inválida* y el kernel mandaría una señal SIGSEGV al proceso para matarle.

Problema 3.23 (julio 2013)

Sea un sistema Linux instalado en un instrumento de medida que tiene las siguientes características físicas: Memoria RAM de 2 MiB y disco flash de 12 GiB. Dicho instrumento debe tomar cada minuto (aproximadamente) un registro de medida, registro que ocupa 1KiB. Para el almacenamiento de los registros se utiliza el fichero Reg de tamaño fijo de 8 GiB, que se trata como un almacenamiento circular, es decir, una vez lleno, los nuevos registros irán sobrescribiendo los más antiguos.

El programador, partiendo de la función "`int getdata(struct registro *registro);`" que rellena el buffer registro de 1 KiB con una medida, establece el siguiente bucle infinito de medida:

```
struct registro * unregistro;
int fd;
int cont = 0;
fd = creat(Reg, 0600);
while (TRUE) {
   unregistro = malloc(sizeof(struct registro));
   getdata(unregistro);
   write(fd, unregistro, sizeof(struct registro));
   cont++;
   if (cont >= 8*1024*1024) {
      lseek(fd, 0, SEEK_SET);
      cont = 0;
   }
   sleep(60);
}
```

a) Indicar cuantas medidas podrá tomar el proceso antes de que se produzca un error, así como la razón por la que se produce dicho error. NOTA: tenga en cuenta todos los elementos involucrados en la gestión de la imagen de memoria del proceso.

b) Rescribir el código anterior proyectando en memoria el fichero Reg.

Solución

a) El problema con el que nos encontramos es que hay un malloc en el bucle sin su correspondiente free, por lo que el heap irá creciendo hasta que el sistema operativo no pueda asignar más memoria al proceso. Para saber cuándo ocurrirá esto hay que hacer ciertas suposiciones, pudiéndose dar los casos siguientes:

Sistema sin memoria virtual. Para el proceso se dispondrá, como máximo, del tamaño de la memoria principal menos lo que ocupe el sistema operativo (suponiendo que no hay otros procesos). Suponiendo que el sistema operativo ocupe 512 KiB nos quedan 1536 KiB para el proceso. Suponiendo que el código ocupa 120 KiB, que otros datos ocupan 1 KiB y que para la pila se han reservado 50 KiB, nos quedaría para el heap 1536 – 120 –1 –50 = 1365 KiB. Como cada pasada del bucle reserva sizeof(struct registro) = 1 KiB, al cabo de 1366 el sistema operativo daría un error de falta de memoria.

Sistema con memoria virtual. En este caso el límite nos viene impuesto o bien por que se alcanza el tamaño máximo del mapa de memoria o bien porque se alcanza el límite impuesto por el soporte físico de la memoria virtual, es decir, por el swap del disco.

Por un lado tenemos un disco con 8 GiB libres, que se podrían dedicar a swap. Por otro lado, la aplicación es de un instrumento de medida, por lo que es lógico que incorpore un pequeño procesador, por lo que supondremos que sus direcciones son de 32 bits. En este caso el mapa de memoria del proceso será la mitad del mapa de memoria total es decir 2^{31} = 2 GiB. Algo de ese mapa estará ocupado por el código, otros datos y la pila, por lo que al llegar a algo menos de 2 Mi medidas el sistema operativo dará error.

b) El programa queda como sigue:

```
   struct registro * unregistro;
   int fd;
   int cont = 0;
   fd = creat("Reg", 0600);
   unregistro = malloc(sizeof(struct registro)); //el registro ocupa 1024 B
   char *org;
   ftruncate (fd, 8*1024*1024*1024); //hacemos que el fichero tenga 8GiB
   org=mmap(NULL, 8*1024*1024*1024, PROT_WRITE, MAP_SHARED, fd, 0);
   close(fd);
   while (TRUE) {
      getdata(unregistro); //obtenemos una medida
      memcpy (org + cont * 1024, unregistro, 1024); //copiamos a memoria
      cont++;
      if (cont >= 8*1024*1024) {
         cont = 0;
      }
      sleep(60);
   }
free(unregistro);
munmap(org, 8*1024*1024*1024);
```

Problema 3.24 (noviembre 2013)

Sea un sistema UNIX con memoria virtual de páginas de 4KiB, con direcciones de 64 bits y que crea una región independiente para el heap.

a) Indicar el tamaño máximo que podría tener una imagen de memoria en dicho sistema.

b) Suponiendo que en un instante determinado la región de heap tiene un tamaño de 8 KiB y que está vacía, calcular justificadamente el tamaño que tendrá la región del heap después de ejecutar la secuencia siguiente.

```
p1 = (char *) malloc (2*1024);
p2 = (char *) malloc (9*1024);
free(p1);
p3 = (char *) malloc (1024);
p4 = (char *) malloc (3*1024);
free(p2);
```

c) Indicar justificadamente cuantas llamadas al SO se habrán producido en la secuencia anterior.

d) Sea un fichero con registros de 1KiB. La cabecera de cada registro es un entero de 4B que indica su tipo. Escribir una función que, utilizando la técnica de proyección de ficheros en memoria, calcule el número de registros de un cierto tipo que hay en el fichero. El prototipo de la función será el siguiente:

```
int CaculaReg(char *FileName, int TipoRegistro);
```

Solución

a) El tamaño de la imgen puede vernir limitado por los recursos físicos de los que se disponga para soportar dicha imagen (marcos de página y paginas de swap) o por el número de direcciones disponible. Desde el punto de vista de las direcciones hay que tener en cuenta que, en principio, una máquina de 64 bits tiene direcciones de 64 bits, lo que indica que tiene un espacio de direcciones de 2^{64} = 16 EiB. Suponiendo que el computador reserva la mitad de ese espacio para el modo de ejecución de núcleo, queda un espacio de 8 EiB para el proceso, siendo éste el mayor tama-

ño de imagen posible. Evidentemente, es un espacio inmenso, por lo que la imagen vendrá realmente limitada por los recursos físicos disponibles.

b) La figura 3.20 muestra la evolución del heap. Se puede observar que ha crecido en dos páginas, por lo que tiene un tamaño de 16 KiB.

c) La figura 3.20 muestra que el heap ha aumentado por dos veces. Por tanto, se han tenido que realizar dos llamadas al SO. Se ha supuesto que la librería del lenguaje solamente solicita el espacio que necesita, no pidiendo más espacio por adelantado.

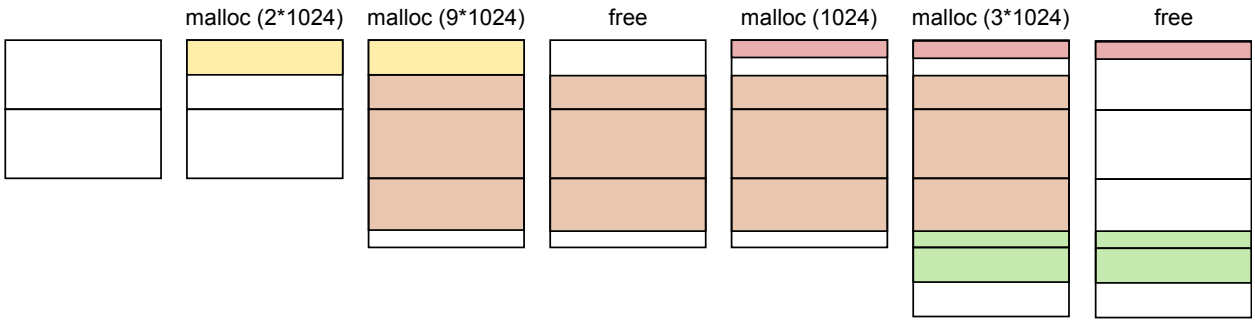


Figura 3.20

d) Se han planteado dos soluciones alternativas, mediante el uso de punteros y mediante el uso de arrays.

```
int CaculaReg(char *FileName, int TipoRegistro){
   int *p, fd, NumRegist = 0;
   struct stat bstat;
   fd = open(FileName, O_RDONLY);
   if (fd < 0) {
      perror("open");
      return 1;
   }
   if (fstat(fd, &bstat) < 0) return 2;
   p = mmap(NULL, bstat.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
   if (p == MAP_FAILED) {
      perror("mmap");
      return 1;
   }
   close(fd);
   for (int i = 0; i< bstat.st_size / 1024; i++)   {
      if (p[i * 256] == TipoRegistro) {//un registro equivale a 1024/4 = 256 enteros
      NumRegist++;
      }
   }
   munmap(p, bstat.st_size);
   return NumRegist;
}
```

```
int CaculaReg(char *FileName, int TipoRegistro){
   int *p, *q, fd, NumRegist = 0, SizeFich;
   struct stat bstat;
   fd = open(FileName, O_RDONLY);
   if (fd < 0) {
      perror("open");
      return 1;
   }
   if (SizeFich = lseek(fd, 0, SEEK_END) < 0) return 2;
   p = mmap(NULL, SizeFich, PROT_READ, MAP_PRIVATE, fd, 0);
   if (p == MAP_FAILED) {
      perror("mmap");
      return 1;
```

```
    }
    close(fd);
    q = p;
    for (int i = 0; i< SizeFich / 1024; i++)  {
      if (*q == TipoRegistro) NumRegist++;
      q = q + 256    //un registro equivale a 1024/4 = 256 enteros
    }
    munmap(p, SizeFich);
    return NumRegist;
}
```

Problema 3.25 (enero 2014) Tiene parte de sincronización

Sea un sistema Linux con páginas de 2 KiB y palabras de 64 bits. Tenemos dos funciones llamadas `Recept` *y* `Distribut`*. La función* `Recept` *es un bucle infinito que se encarga de recibir mensajes a través de un dispositivo de comunicaciones leyendo de su descriptor de fichero, que se le pasa como un parámetro. El mensaje es de tamaño variable (de 64 B hasta 4 KiB) e incluye una cabecera con el destino y el tamaño del mensaje.* `Recept` *crea un buffer por mensaje mediante un* `malloc` *y, seguidamente, encola la dirección del buffer creado en un buffer circular llamado* `bufcir`*, con capacidad para 512 mensajes.*

La función `Distribut` *es un bucle infinito que se encarga de enviar, con otro formato, los mensajes previamente encolados en* `bufcir`*, a través del dispositivo de comunicaciones cuyo descriptor de fichero recibe como parámetro.*

El montaje es dinámico y se tienen los siguientes tamaños del programa y de las dos bibliotecas utilizadas:

```
   text    data     bss     dec     hex filename
  11189     168       9   11366    2c66 Miprogr.o
1719061   11508   11316 1741885  1a943d libc.so
  92630     868    8352  101850   18dda libpthread.so
```

En una primera versión se ejecuta el programa siguiente:

```
#include <stdio.h>
#include <pthread.h>
void* Recept (void* arg)
{     ...
      while (1) {...}
}
void* Distribut (void* arg)
{     ...
      while (1) {...}
}
int main(void)
{
      pthread_t distrib;
      int fdin, fdout;
            //Aquí hay un código que abre  fdin y fdout
      pthread_create(&distrib, NULL, Distribut, (void *)fdout);
      Recept((void *)fdin);
}
```

a) Indicar las regiones de memoria que formarán la imagen de memoria del proceso cuando `Recept` *esté ejecutando su bucle.*

b) Representar gráficamente cómo estará el heap después de la secuencia siguiente: Se recibe un mensaje de 2 KiB, se recibe un mensaje de 1 KiB, se envía el primer mensaje, se recibe un mensaje de 3 KiB, se envía el segundo mensaje y se recibe un mensaje de 1 KiB.

c) Indicar en qué parte del código se debe poner la declaración del buffer circular `bufcir` *y cómo sería dicha declaración.*

d) Considerar ahora que `main` *crea varios threads con* `Recept` *y varios con* `Distribut`*. Indicar el modelo de sincronización a aplicar entre dichos threads.*

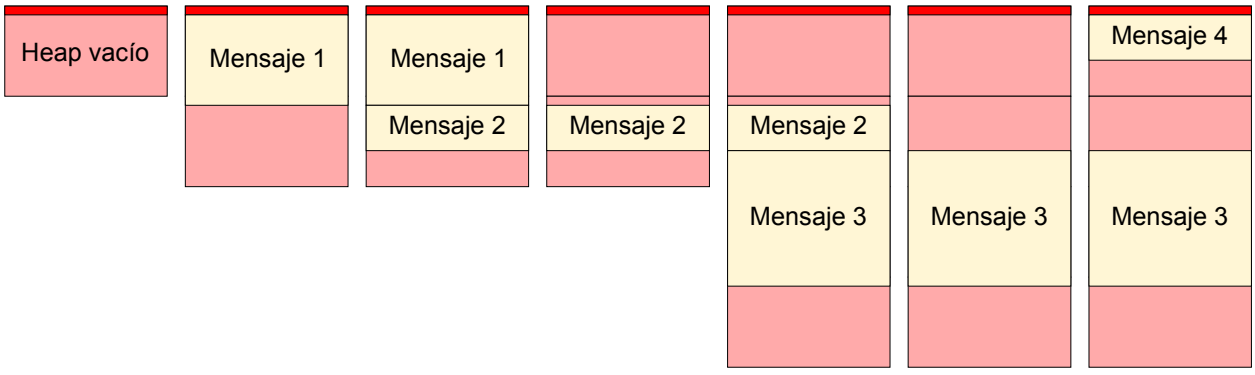
e) Plantear, para el caso anterior, el mecanismo de sincronización a utilizar, desarrollando el código correspondiente.

Solución

a) Las regiones se muestran en la tabla adjunta. Tenemos dos threads, el inicial (que usa la pila original) y el creado (que requiere una nueva pila), por lo tanto necesitamos dos pilas. Las pilas requieren una página de salvaguarda, que esté asignada como r-- o como ---. De esta forma, si el proceso intenta escribir en dicha página es que necesita más pila. Por eso, consideramos que, como mínimo, las pilas han de tener dos páginas (una con la pila usada hasta ese momento y la otra de salvaguarda).

Región	Tamaño		Permisos	Origen	Tipo
	Nº de páginas	Fijo/Variable			
Texto Miprogr	11.189 B → 6 páginas	Fijo	r-x	Ejecutable	Compartida
Datos CVI y SVI de Miprogr más heap	177 B + heap, inicialmente 1 página	Variable	rw-	CVI: ejecut. SVI y heap: ← 0	Privada
Texto libc	1.719.061 B → 840 páginas	Fijo	r-x	Librería	Compartida
Datos CVI y SVI de libc	22.824 B → 12 páginas	Fijo	rw-	CVI: librer. SVI: ← 0	Privada
Texto libpthread	92.630 B → 46 páginas	Fijo	r-x	Librería	Compartida
Datos CVI y SVI de libpthread	9.220 B → 5 páginas	Fijo	rw-	CVI: librer. SVI: ← 0	Privada
Pila del thread	Dos o más páginas. La última con permisos r--	Variable	rw- (r--)	← 0	Privada
Pila del thread primario del proceso	Dos o más páginas. La última con permisos r--	Variable	rw- (r--)	Pila inicial resto ← 0	Privada

b) La figura adjunta muestra la evolución de la región de datos que incluye el heap. Los datos con valor y sin valor inicial ocupan los primeros 177 B, el resto es el heap, que va creciendo a medida que se va necesitando. Lógicamente, la función `Distribut` se ha de encargar de liberar la memoria de los mensajes creados por `Recept`.



c) La declaración ha de ponerse antes de las funciones, en la zona de datos globales, debajo de los includes. La declaración es:

```
void * buffdir[512];
```

d) Se trata de un clásico problema productor consumidor con varios procesos de cada tipo.Los procesos `Recept` son los productores y los procesos `Distribut` son los consumidores.

e) Dado que hay que sincronizar threads, la solución más recomendable es la de utilizar mutex y condiciones. La información de control que necesitamos es una variable con el número de mensajes activos en el sistema. Esta variable

ha de ser global. Por otro lado las condiciones que nos encontramos es que `buffdir` esté totalmente lleno, condición por la que `Recept` debe esperar o que esté totalmente vacío, condición por la que `Distribut` ha de esperar.

Una posible solución es la siguiente:

```
#define BUFCIR_SIZE      512
#define MAXMENSAJE       4*1024
int NumMensajes, PrimerHueco, PrimerMensaje;
void * buffdir[BUFCIR_SIZE];
pthread_mutex_t MiMutex;   // Acceso a sección crítica
pthread_cond_t no_lleno, no_vacio; // Condiciones de espera

void* Recept (void* arg)
{
   void * p;
   char buf[MAXMENSAJE]; //
   int fdin = (int)arg; //Se convierte el parámtro, que viene en forma de puntero
   while(1) {
      /*Aquí va el código de lectura del mensaje que suponemos lo deja en buf terminado en '\0'.
      Primero se lee la cabecera para conocer el tamaño del mensaje. Luego se lee el cuerpo.*/
      p = malloc(strlen(buf) + 1) //Creamos el espacio para el mensaje.
      strncpy(p, buf, strlen(buf) + 1);
      pthread_mutex_lock(&MiMutex);
      while(NumMensajes == BUFCIR_SIZE) //Condición de espera
         pthread_cond_wait(&no_lleno, &MiMutex);
      buffdir[PrimerHueco] = p;
      PrimerHueco = (PrimerHueco + 1) % BUFCIR_SIZE;
      NumMensajes++;
      pthread_cond_signal(&no_vacio);
      pthread_mutex_unlock(&MiMutex);
   }
}

void* Distribut (void* arg)
{
   void * p;
   int fdout = (int)arg; //Se convierte el parámtro, que viene en forma de puntero
   while(1)  {
      pthread_mutex_lock(&MiMutex);
      while(NumMensajes == 0) //Condición de espera
         pthread_cond_wait(&no_vacio, &MiMutex);
      p = buffdir[PrimerMensaje];
      PrimerMensaje = (PrimerMensaje + 1) % BUFCIR_SIZE;
      NumMensajes--;
      pthread_cond_signal(&no_lleno);
      pthread_mutex_unlock(&MiMutex);
      //Aquí está el código que envía el mensaje a través de fdout
      free(p); //Liberamos el espacio de memoria que contiene el mensaje
   }
}

int main(void)
{
   pthread_t distrib;
   int fdin, fdout;
      //Aquí hay un código que abre fdin y fdout
   pthread_mutex_init(&MiMutex, NULL);  //Inicialización
   pthread_cond_init(&no_lleno, NULL);
   pthread_cond_init(&no_vacio, NULL);
   NumMensajes = 0;
   PrimerHueco = 0; //Los mensajes los consideramos numerados de 0 a BUFCIR_SIZE - 1
   PrimerMensaje = 0;
   pthread_create(&distrib, NULL, Distribut, (void *)fdout);
   Recept((void *)fdin);
   return 0;
```

```
}
```

Problema 3.26 (mayo 2014)

Sea un sistema con memoria virtual que presenta las siguientes características: páginas de 8 KiB; palabras de 32 bits; el sistema funciona con alineamiento de los datos. Al ejecutar el mandato size, sobre un fichero ejecutable y sobre una biblioteca, la salida que se obtiene es:

 # size mi_programa

text	*data*	*bss*	*dec*	*hex*	*filename*
12096	*1080+ ¿?*	*8724+ ¿?*	*21900+ ¿?*	*558C + ¿?*	*mi_programa*

 # size mi_biblio

text	*data*	*bss*	*dec*	*hex*	*filename*
645024	*16384 + ¿?*	*13424 + ¿?*	*674832 + ¿?*	*A4C10 + ¿?*	*mi_biblio*

donde ¿? se corresponde con el valor deducido de la información disponible en el código que está distribuido entre los siguientes ficheros (suponga que también están incluidos todos los ficheros de cabecera necesarios para las declaraciones de las funciones del lenguaje y los servicios del sistema pero esta información no se considera para el cálculo de ¿?):

Fichero cabecera.h:

```
int a=3, b=5;   // Considere que el tipo int requiere 4 bytes
float *c;       // Considere que el tipo float requiere 4 bytes
struct tipo_estruct{
    short a;    // Considere que el tipo short requiere 2 bytes
    int b; char c; char d;
    double e;   // Considere que el tipo double requiere 8 bytes
    } var_estruct;
```

Fichero load_dyn.c:

```
#include "cabecera.h"
void *load_dyn(char * nombre) {
    void * hd;
    hd=dlopen (nombre, RTLD_NOW));
/* Punto A */
    return (hd);}
```

Fichero fuente.c:

```
#include "cabecera.h"
extern int load_dyn(char * nombre);
char literal[20]="0123456789=!?$%&/()";
int main (void) {
    int fd=0, j=5;
    void *hd;
    hd=load_dyn("mi_biblio");
    c=dlsym(hd,"mi_variable"); // En la biblioteca, int mi_variable=7;
    struct tipo_estruct *vest, *aux;
    vest=malloc(sizeof(struct tipo_estruct));
    vest->a=(short) a;
    vest->d='d';
    vest->e=(double) mi_variable;
    aux=vest;
/* Punto B */
    free(vest);
    fd = open ("datos.txt", O_RDWR);
    vest=mmap       (0,sizeof(struct       tipo_estruct),       PROT_READ|PROT_WRITE,
    MAP_SHARED,fd,0); close(fd);
    aux=vest;
    aux->c=literal[10]; aux->d=literal[11];
    munmap(aux,sizeof(struct tipo_estruct));
    printf("c: %c, d: %c, e: %f\n",vest->c, vest->d, vest->e);
    return 0;
/* Punto C */
}
```

El fichero datos.txt es un fichero ASCII que ocupa 40 B.

Se pide:

a) *Indique el espacio ocupado por el tipo de datos tipo_estruct justificando la respuesta.*
b) *Describa el mapa de memoria del proceso en los puntos A y B. Para cada región existente, indique el espacio ocupado en B y el número de páginas que abarca la región, sus permisos, si es una región privada o compartida, el soporte físico que tiene la región y el origen de su contenido. Suponga que a la región asociada a la pila se le asignan inicialmente 32 KiB y las variables de entorno ocupan 500 B.*
c) *Explicar cómo le afecta al heap y a la región proyectada en memoria la ejecución de free(vest).*
d) *Indicar a qué región pertenece la dirección a la que apunta la variable vest en los puntos A, B y C.*
e) *Indique los valores impresos al invocar la función printf.*

Solución

a) El campo b de tipo int no puede ajustarse al campo a de tipo short al existir alineamiento de los datos y trabajar con palabras de 32 bits, por lo que queda un hueco de 2 bytes sin poder utilizarse entre estos dos campos. Lo mismo ocurre entre los campos d y e. En este caso, el hueco es de 1 byte. Por lo tanto, la estructura ocupa 20 bytes. Solamente si se activara la optimización del compilador, éste podría reordenar los distintos campos de la estructura para que el orden de almacenamiento de éstos permitiera acceder a cada uno de ellos en un solo acceso a memoria. Una posible reordenación sería: `struct tipo_estruct{ short a; char c; char d; int b; double e; }` y en este caso, la estructura pasaría a ocupar 16 B.

b) Punto A:

- Región de código: 12096 B; 2 páginas; R-X; compartida; fichero ejecutable; fichero ejecutable.
- Región de datos con valor inicial: 1080 B + 8 B (2 int); 1 página; RW-; privada; swap; fichero ejecutable.
- Región de datos sin valor inicial: 8724 B + 4 B (float *) + 20 B (tipo_estruct); 2 páginas; RW-; privada; swap; rellenar con ceros.
- Región de heap: 0 B; 0 páginas; privada; swap; rellenar con ceros.
- Región de pila: a la pila se le asignan inicialmente 32 KiB, que permiten almacenar sin problemas las variables de entorno, el bloque de activación de la función main con sus variables locales (20 B; 2 int + 3 punteros), el bloque de activación de la función load_dyn, la variable local de esta función (4 B de un puntero) y el bloque de activación de la función dlopen; 4 páginas; privada; swap; rellenar con ceros.

La invocación del montaje de la biblioteca se ha realizado con la opción RTLD_NOW, por lo que en ese momento se monta explícitamente la biblioteca.

- Región de código de la biblioteca: 645024 B; 79 páginas; R-X; compartida; fichero biblioteca; fichero biblioteca.
- Región de datos con valor inicial de la biblioteca: 16384 B + 8 B (2 int); 3 páginas; RW-; privada; swap; fichero biblioteca.
- Región de datos sin valor inicial de la biblioteca: 13424 B + 4 B (float) + 20 B (tipo_estruct); 2 páginas; RW-; privada; swap; rellenar con ceros.

Si en la solución se ha considerado el modelo clásico de UNIX con una única región de datos que aglutina las regiones de datos con y sin valor inicial y el heap, se necesitarían 2 páginas (2 páginas > 1080 B + 8 B (2 int) + 8724 B + 4 B (float) + 20 B); privada; swap; los datos con valor inicial se encontrarían en el fichero ejecutable y el resto de datos proceden del mecanismo de rellenar con ceros los marcos de página. En este caso, la región de datos de la biblioteca se describiría: 16384 B + 8 B (2 int) + 13424 B + 4 B (float) + 20 B (tipo_estruct); 5 páginas; RW-; privada; swap; rellenar con ceros.

Punto B:

Además de las regiones existentes en el punto A, en el punto B se produce la siguiente modificación en el heap:

- Región de heap: 20 B; 1 páginas; privada; swap; rellenar con ceros.

Si en la solución se ha considerado el modelo clásico de UNIX con una única región de datos que aglutina las regiones de datos con y sin valor inicial y el heap, se necesitarían las mismas 2 páginas (2 páginas > 1080 B + 8 B (2 int) + 8724 B + 4 B (float) + 20 B + 20 B (heap)); privada; swap; los datos con valor inicial se encontrarían en el fichero ejecutable y el resto de datos proceden del mecanismo de rellenar con ceros los marcos de página.

c) Las funciones malloc y free actúan solamente sobre el heap, mientras que el fichero proyectado en memoria es una región completamente independiente del heap que se gestiona con los servicios mmap y munmap. Al invocar la función free pasando como argumento la variable vest de tipo tipo_estruct, se liberará una zona de memoria en el heap correspondiente al tamaño de sizeof(struct tipo_estruct).

d) La variable vest es una variable local de la función main y se almacenará en la pila, pero no está inicializada. Por lo tanto, en el instante A, al no estar inicializada, se puede considerar que la variable apuntará a posiciones basura porque el programador no tiene ningún control sobre el contenido inicial de la posición donde se ha almacenado vest.

En el B, apunta al heap tras la invocación de malloc.

Tras la proyección del fichero en memoria, recoge el valor de comienzo de la región creada. Al llegar al instante C, esa región ya no existe en la imagen de memoria del proceso tras la invocación del servicio munmap, por lo que apunta a una dirección no válida en el espacio de memoria del proceso.

e) Al invocar la función printf se está accediendo a posiciones de memoria que ya no son válidas en el proceso porque se ha eliminado la región de memoria en la que estaban incluidas, por lo tanto se producirá un error de acceso a memoria y se abortará la ejecución del proceso.

4

COMUNICACIÓN Y SINCRONIZACIÓN

Problema 4.1 (septiembre 1999)

a) Se desea construir una función llamada pizarra, que reciba 3 parámetros:

num_escritores: Número de procesos que escribirán en dicha pizarra.

num_lectores: Número de procesos que leerán los datos de la pizarra. Cuando se lee un dato, se borra de la misma.

fichero_sal: Fichero de salida donde se copiarán los datos que permanezcan en la pizarra una vez finalizados todos los procesos lectores y escritores.

La función creará num_escritores procesos escritores y num_lectores procesos lectores.

La simulación de la pizarra se llevará a cabo a través de un ***pipe****.*

Para la escritura de los datos en la pizarra, suponed que existe una función, llamada escribir_pizarra( ). Esta función escribe los datos por la salida estándar. Su prototipo es el siguiente:

```
void escribir_pizarra (void);
```

Los procesos escritores realizarán una única invocación a esta función.

De forma análoga, para la lectura de los datos en la pizarra, suponed que existe una función llamada leer_pizarra( ). Esta función lee los datos de la entrada estándar. Su prototipo es el siguiente:

```
void leer_pizarra (void);
```

Los procesos lectores realizarán una única invocación a esta función.

Las escrituras y lecturas de la pizarra se realizan de forma concurrente.

Cuando finalizan todos los procesos lectores y escritores, la función pizarra, debe escribir los datos que queden en la pizarra en el fichero cuyo nombre es fichero_sal, y que se debe crear en el directorio actual.

b) Considerad ahora que las funciones escribir_pizarra y leer_pizarra no escriben ni leen de la salida/entrada estándar, sino del descriptor de fichero que reciben como parámetro. Los prototipos de las funciones, en este caso, son:

```
void escribir_pizarra (int fd);
void leer_pizarra(int fd);
```

¿Cómo quedaría modificado el código del apartado a)? Reescribidlo teniendo en cuenta estas consideraciones.

Solución

a) La comunicación se va a llevar a cabo mediante un *pipe*. Como el número de escrituras máximo es menor que 100 (por la restricción de 100 escritores) y éstos escriben un carácter cada uno, estamos seguros de que el *pipe* no se va a llenar.

El código correspondiente a este apartado es el siguiente:

```
/* El tamaño es mayor de 100 caracteres, lo que garantiza su almacenamiento */
#define MAX_BUFF 1024

/* Prototipos de las funciones de escritura y lectura en la pizarra */
void escribir_pizarra(void);
void leer_pizarra(void);

/* Función pizarra: */
void pizarra (int num_esc, int num_lect, char *fichero_sal)
{

   int fd_pipe[2];
   int fd_sal;
   int i, pid, n_leidos;
   char

   if (num_esc >= 100)
   {
      fprintf(stderr,"Número de escritores no valido\n");
      exit(1);
   }


   if (pipe(fd_pipe) < 0)
   {
      perror("pipe");
      exit(1);
   }

   /* Creación de procesos escritores */
   for (i=0; i<num_esc; i++)
   {

      switch( fork())
      {

      case -1:
         perror("fork");
         close(fd_pipe[0]);
         close(fd_pipe[1]);
         exit(1);
      case 0:
         close(fd_pipe[0]);
         close(STDOUT_FILENO);
         dup(fd_pipe[1]);
         close(fd_pipe[1]);
         escribir_pizarra();
         /* El proceso hijo finaliza su ejecución */
         exit(0);
      }
   }


   /* Creación de procesos lectores */
```

```
for (i=0; i<num_lec; i++)
{

   switch( fork())
   {

   case -1:
      perror("fork");
      close(fd_pipe[0]);
      close(fd_pipe[1]);
      exit(1);
   case 0:
      close(fd_pipe[1]);
      close(STDIN_FILENO);
      dup(fd_pipe[0]);
      close(fd_pipe[0]);
      leer_pizarra();
      /* El proceso hijo finaliza su ejecución */
      exit(0);
   }
}

/* El padre cierra el descriptor del pipe de escritura, ya que si hay algún proceso intentando leer del pipe
vacío, se quedaría bloqueado */
close(fd_pipe[1]);

/* El padre espera por la finalización de todos los procesos hijos */
for (i=0; i< (num_esc+num_lec); i++)
{
   pid= wait(NULL);
   if (pid == -1)
   {
      perror("wait");
      close(fd_pipe[0]);
      close(fd_pipe[1]);
      exit(1);
   }
}

/* Se crea el fichero de salida y se escribe lo que queda en el pipe en el mismo */

fd = creat(fichero_sal, 0600);

if (fd <0)
{
   perror("creat");
   close(fd_pipe[0]);
   exit(1);
}

while (n_leidos = read(fd_pipe[0], buff, MAX_BUFF) > 0)
{
   if (write(fd,buff,n_leidos) != n_leidos)
   {
      perror("write");
      close(fd_pipe[0]);
      close(fd);
      exit(1);
   }
}
if (n_leidos < 0)
{
```

```
        perror("read");
        close(fd_pipe[0]);
        close(fd);
        exit(1);
    }

    /* Se cierran todos los descriptores y se finaliza el programa */
    close(fd_pipe[0]);
    close(fd);

}
```

b) Este apartado sólo queda modificado de la siguiente forma:

Los procesos escritores sólo tendrán que ejecutar este código:

```
escribir_pizarra(fd_pipe[1]);
close(fd_pipe[0]);
close(fd_pipe[1]);
exit(0);
```

Los procesos lectores ejecutarán este código**:**

```
leer_pizarra(fd_pipe[0]);
close(fd_pipe[0]);
close(fd_pipe[1]);
exit(0);
```

Problema 4.2 (abril 2000)

Se desea programar un proceso (secuenciador), que realiza las siguientes operaciones:

- *Crea un número de procesos (P1, P2, P3, ...) determinado por la constante NUM_PROC.*
- *Cada uno de los procesos hijos Pi debe bloquearse hasta que el secuenciador (proceso padre) lo desbloquee.*
- *El orden en el que el proceso secuenciador debe desbloquear los procesos es el de creación de los mismos (P1, P2, P3, ...). El desbloqueo se hace de forma secuencial, es decir, en primer lugar se desbloquea al primer proceso creado (P1); cuando éste finalice y haya pasado cierto tiempo, se desbloquea al segundo (P2) y así sucesivamente hasta que finalicen todos los procesos hijos.*
- *El proceso secuenciador desbloquea al proceso que le corresponda cuando haya pasado un número determinado de segundos (el proceso secuenciador recibe dicho valor como único parámetro) y siempre que el anterior proceso hijo desbloqueado haya finalizado.*
- *Suponed que los hijos siempre finalizan de forma correcta.*
- *Los procesos hijos deben comunicarse entre ellos. El proceso Pi debe comunicarse con el proceso P(i+1), donde el orden se define por el momento de creación del mismo. La comunicación se lleva a cabo mediante el envío de los datos generados por una función llamada procesar_datos. Esta función se define como:*

  ```
  char *procesar_datos (int fd);
  ```

  Recibe como argumento un descriptor de fichero del cual se leerán los datos que se deben procesar. Esta función devuelve una cadena de caracteres, que contendrá los datos que hay que enviar al siguiente proceso en la cadena.

  El proceso padre (secuenciador) se encargará de escribir un dato inicialmente para que el primer proceso (P1) pueda generar los datos que le pasará al segundo proceso (P2).

Se pide programar el código del proceso secuenciador así como de los procesos hijos. No hay que implementar la función procesar_datos( ).

La estructura del código de cada uno de los procesos hijos será:

< Bloqueo esperando a que el proceso padre (secuenciador) lo despierte>

< Llamada a procesar_datos, para que se generen nuevos datos a partir de los datos que le ha comunicado el proceso anterior en la cadena>

< Los datos generados se envían al siguiente proceso en la cadena>

< Termina la ejecución, señalizándolo si fuera necesario>

Para sincronizar los procesos, utilizad el mecanismo de semáforos. Para la comunicación de los procesos, utilizad el mecanismo de pipes. Para la espera del proceso padre, se puede utilizar la función sleep( ). Se debe minimizar el número de mecanismos a crear por el programa.

Solución

La figura Error: Reference source not found muestra el esquema de la solución propuesta.

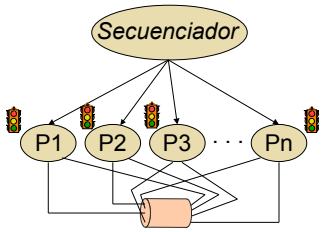


Figura 4.1

```
int main (int argc, char *argv[])
{
   sem_t sem_inicio_ejecucion_hijo[NUM_PROC];


   int i, fildes[2];
   char *datos;
   char c;

   /* Creación de los pipes */
   if (pipe (fildes) < 0) {
      perror("pipe");
      exit(1);
   }

   /* Inicialización de los semáforos */
   for (i=0; i<NUM_PROC; i++) {
      sem_init(&(sem_inicio_ejecucion_hijo[i]),1,0);
   }


   /* El proceso Secuenciador crea NUM_PROC procesos hijos */
   for (i=0; i<NUM_PROC; i++) {
      if (fork() == 0 ) { /* Proceso hijo */
         /* El proceso hijo se bloquea esperando a que le dé paso el proceso secuenciador */
         sem_wait(sem_inicio_ejecucion_hijo[i]);

         /* Procesa los datos y los escribe en el descriptor de escritura del pipe */
         datos = procesar_datos(fildes[0]);
         write(fildes[1],datos,strlen(datos));

         /* Cierre del pipe */
         close(fildes[0]);
         close(fildes[1]);

         /* Se destruye el semáforo correspondiente */
         sem_destroy(&(sem_inicio_ejecucion_hijo[i]));

         /* Sale de forma ordenada */
         return 0;
      }
   }
```

```
    /* continua el proceso padre */

    /* el proceso padre escribe un caracter en el pipe */
    write(fildes[1],&c,1);

    for (i=0; i<NUM_PROC; i++) {
       /* Desbloquea a un proceso hijo en orden */
       sem_post(sem_inicio_ejecucion_hijo[i]);

       /* Duerme durante el tiempo especificado por la rodaja */
       sleep(atoi(argv[1]));

       /* Se bloquea esperando que termine el proceso hijo que corresponda */
       wait(NULL);
    }

    /* Cierra el pipe */
    close(fildes[0]);
    close(fildes[1]);

    /* Finaliza */
    return 0;

}
```

Problema 4.3 (septiembre 2000)

Una aplicación consta de 20 procesos, P1 a P20, que cooperan entre sí de diversas formas, que se describen a continuación.

Cooperación A: Es un esquema productor-consumidor, en el que el cada proceso Pi genera de vez en cuando una estructura de información que tiene un tamaño de 1 KiB y que debe utilizar seguidamente otro de los procesos Pj.

Cooperación B: En este caso todos los procesos deben utilizar una misma información compuesta por 100.000 registros de 200B. Cada proceso, de acuerdo a determinadas solicitudes que van llegando a la aplicación, de vez en cuando debe obtener el contenido de 7 registros y debe modificar el contenido de 2 registros. La identidad de los registros leídos y modificados viene determinada por cada solicitud recibida, siendo éstas completamente aleatorias.

Cooperación C: El proceso Pi genera una información de 4 MiB. Cuando la ha completado el proceso Pj ha de tratar dicha información.

Supondremos que el coste en tiempo de procesador de las operaciones de comunicación y sincronización entre procesos es proporcional al número de llamadas al SO, con la excepción del coste de realizar un paso de mensaje de más de 300B, que consideraremos equivalente a 1+ KiB·0,1 llamadas al sistema.

Proponer para cada uno de los casos anteriores dos soluciones en pseudocódigo que garanticen el correcto funcionamiento y la coherencia de la información. Una solución utilizará la técnica de memoria compartida y la otra utilizará la técnica de paso de mensajes. Marcar (con un asterisco o subrayado) cada uno de los servicios del SO utilizados.

Calcular el coste computacional de cada una de ellas y comentar los resultados obtenidos.

Solución

El problema plantea una serie de situaciones de comunicación entre procesos y nos pide que comparemos la solución basada en memoria compartida con la de paso de mensajes. Por tanto, las soluciones que se planteen con ambas técnicas han de ser lo más parecidas posibles.

Por otro lado, en los supuestos A y B la comunicación se dice que es repetitiva. Ello hace que se tenga que diferenciar claramente entre la fase de creación y eliminación de los mecanismos de comunicación y sincronización, que solamente se realizará una vez, y la fase de utilización de los mismos, que se utilizará repetidamente. Aunque en el

supuesto C no se especifica si la comunicación es repetitiva o no, se seguirá la misma filosofía que para los casos A y B.

a) Cooperación A

Consideraremos dos situaciones similares: una región de memoria común por proceso, que permita almacenar 1 KiB, o una cola de mensajes por proceso, que permita almacenar un mensaje de 1 KiB. Esta memoria o cola estará asociada al proceso destinatario de la información (Pj). Consideraremos que este KiB contiene una estructura con campos como 'edad', 'altura', etc. Con estas condiciones las soluciones son las siguientes:

Memoria compartida

Los mecanismos necesarios son la memoria común asociada al proceso Pj, cuyo puntero estará almacenado en mpj, y dos semáforos asociados a esta memoria. Un semáforo (lector) tendrá por misión bloquear al lector hasta que el productor no introduzca un dato, y otro semáforo (escritor) tendrá por misión impedir que el escritor entre a modificar la memoria si el lector no ha terminado con su acceso.

El trozo del proceso Pi que realiza la comunicación es el siguiente:

```
.....
sem_wait (escritor); //se garantiza que solamente se escribe cuando se debe
mpj->edad = 7; //se asignan a los campos de la estructura establecida en la
                //memoria compartida los valores deseados.
mpj->altura = 112;
.....
sem_post (lector); //se permite que entre el lector
.....
```

Mientras que el trozo de código el proceso Pj que realiza la comunicación es el siguiente:

```
......
sem_wait (lector);        //se garantiza que solamente se lee cuando se debe
miedad = mpj->edad;       //se asignan los valores de la estructura establecida
mialtura = mpj->altura; //en la memoria compartida a las variables
....                      //deseadas
sem_post (escritor);      //se permite que entre el escritor
....
```

En total cada comunicación requiere 4 llamadas al SO.

Paso de mensajes

El mecanismo necesario es la cola de mensajes asociada al proceso Pj, cuyo descriptor denominaremos cmpj.

El trozo del proceso Pi que realiza la comunicación es el siguiente:

```
....
mq_send (cmpj, &buffer, 1024, 0);
....
```

Mientras que el trozo de código del proceso Pj que realiza la comunicación es el siguiente**:**

```
....
mq_receive (cmpj, &buffer,1024, 0);
....
```

En total cada comunicación requiere 2 llamadas al SO.

b) Cooperación B

En este caso se presenta una disyuntiva de diseño muy importante, relativa al paralelismo en el acceso al almacén de registros. En efecto, si la protección de acceso se extiende a todo el almacén, en cada instante solamente habrá un proceso que pueda estar trabajando con él. En el caso de que el tiempo que tarde el proceso en realizar su función sea grande sería más interesante establecer un mecanismo que permitiera bloquear solamente los registros que están en uso. Esta será la solución propuesta.

Memoria compartida

Para el caso de memoria compartida consideraremos que hay una única memoria común a todos los procesos (mcomun). Dado que establecer un semáforo por registro nos llevaría a un número exagerado de ellos, se propone tener otra memoria común que mantenga la información de qué registros están bloqueados (mbloqueados). Esta memoria vendrá protegida por un semáforo (semaforo), de forma que solamente un proceso pueda accederla en cada

instante para bloquear o desbloquear los registros pertinentes. Nótese que el tiempo que se tarda en realizar este acceso es muy pequeño, por lo que esta solución no debe producir una contención apreciable entre los procesos.

El trozo del proceso Pi que realiza el acceso es el siguiente:

```
....
sem_wait (semaforo); //se garantiza que solamente uno acceda a mbloqueados
....        //se comprueba que los registros están libres y, en su caso, se marcan como ocupados. Para los leídos se
            //incrementará el número de lectores y los que hay que escribir se marcan cerrados para lectura y escritura.
            //En caso de no poder tomar control de los registros, hay que cancelar la operación e intentarla más tarde.
sem_post (semaforo); //se permite que entre otro proceso

alfa = mcomun[4700].campo + mcomun[3826].campo;
....        //se procesan los registros y se graban los resultados
mcomun[37652].campo = alfa*3;
....
sem_wait (semaforo); //se garantiza que solamente uno acceda a mbloqueados
....                 //se liberan los registros utilizados.
sem_post (semaforo); //se permite que entre otro proceso
```

En total cada comunicación requiere 4 llamadas al SO en el supuesto de que los registros estén libres.

Cola de mensajes

En el caso de paso de mensajes la solución pasa por dedicar un proceso a gestionar el almacén de datos. Dicho proceso tendrá asociada una cola de mensajes (cmgestor). Además, cada proceso cliente Pi tendrá su cola de mensajes para las contestaciones. Cada proceso Pi le mandará la petición con los registros que necesita. El proceso gestor comprobará que los registros estén disponibles y, en caso afirmativo, los bloqueará y enviará un mensaje con sus contenidos. En caso contrario retendrá la petición hasta que se liberen los registros necesarios.

El trozo del proceso Pi que realiza el acceso es el siguiente:

```
....        //prepara mensaje de petición en 'buffer1'
mq_send (cmgestor, &buffer1, sizeof(buffer1), 0);
mq_receive (cmpi, &buffer2, sizeof(buffer2), 0);
....        //procesa los registros y los introduce en 'buffer3'
mq_send (cmgestor, &buffer3, sizeof(buffer3), 0);
```

Mientras que el trozo de código del proceso gestor es el siguiente:

```
....
mq_receive (cmgestor, &buffer1, sizeof(buffer1), 0); //espera orden o respuesta
   //en caso de ser orden
....     //comprueba que los registro están libres y los carga en 'buffer2'
mq_send (cmpi, &buffer2, sizeof(buffer2), 0);
   //en caso de respuesta
.....    //graba los registros enviados y analiza si alguna petición retenida puede ser ahora atendida.
```

El número de llamadas al sistema es de 6.

c) Cooperación C

El código es idéntico al del caso A, por lo que no se repite.

Comparación de prestaciones:

	Llamadas SO	**Llamadas equivalentes SO**
A memoria común	4	4
A paso de mensajes	2	2(1+0,1) = 2,2
B memoria común	4	4
B paso de mensajes	6	2 + 2(1+0,14) + 2(1+0,04) = 6,38
C memoria común	4	4
C paso de mensajes	2	2(1+ 409,6) = 821,2

Para los casos A y B el coste es muy parecido. Habría que considerar otros factores para seleccionar la mejor solución al problema.

Para el caso C la solución de paso de mensajes es muy costosa, por la sobrecarga introducida en el envío de grandes mensajes.

Problema 4.4

Un pool de threads es un mecanismo para ejecutar de forma concurrente varias tareas, por medio de threads, pero que adicionalmente proporciona un control y una gestión adecuada para determinados problemas.

Un pool de threads consiste en un conjunto de procesos ligeros (threads) fijo o de tamaño dinámico que de forma permanente están en ejecución en el proceso. Si existe una tarea pendiente, ésta es asignada al thread, si no existe, el thread no se destruye sino que se deja bloqueado a esperas de una nueva tarea a realizar. Esta estrategia permite ahorrarse los costes de creación y destrucción de threads en aplicaciones que crean muchos de estos elementos para pequeñas tareas.

Se propone el desarrollo de una pequeña biblioteca que implemente una versión muy elemental de un pool de threads. Esta biblioteca gestionará un número de hilos de ejecución fijo (definidos en fase de inicialización). La biblioteca dispondrá de un mecanismo para solicitar un hilo que se encuentre ocioso (estado `IDLE`) y asignarle una tarea. Las tareas serán un conjunto de funciones `TRABAJO_A`, `TRABAJO_B`, `TRABAJO_C`, `TRABAJO_D` y `TRABAJO_E` que serán pasadas como parámetro al thread.

Solución

El código está dividido en tres ficheros de implementación:

- **`thread_pool.c`**: Contiene la implementación de las funciones proporcionadas por la biblioteca. Éste código será válido para cualquier problema con la misma taxonomía. Las cabeceras de las funciones están en el fichero `thread_pool.h`.
- **`main.c`**: Código del programa principal, invoca las funciones de la biblioteca. Este código abre un fichero de órdenes que contiene la secuencia de trabajos a realizar, así como periodos en los cuales el sistema esta "dormido" (no recibe nuevas peticiones de trabajos y está ejecutando las actuales).
- **`jobs.c`**: Este fichero incluye las implementaciones de cada una de las funciones que representan las posibles tareas que realizará el sistema. Son funciones simples que ejecutan uno o varios bucles anidados y representan lo que sería posibles trabajos del sistema Las cabeceras de las funciones están en el fichero `jobs.h`. Las implementaciones de estas funciones son unos casos de ejemplo cualquiera.

Fichero `main.c`

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "thread_pool.h"
#include "jobs.h"

struct thread_pool_t TP;     /* Variable pool de threads */

/* Estructura usada para traducir los nombres de los trabajos (tal y  como aparecen el fichero de ordenes) a las funciones
  que los implementan. */
struct jobs_st
{
   char job_name[24];   /* Nombre del trabajo */
   void (*job)(int);    /* Puntero a función */
} job_list[]={
   {"TRABAJO_A",trabajo_A},
   {"TRABAJO_B",trabajo_B},
   {"TRABAJO_C",trabajo_C},
   {"TRABAJO_D",trabajo_D},
   {"TRABAJO_E",trabajo_E}};

/* Funcion auxiliar que busca el trabajo en la tabla anterior y  devuelve su indice en dicha tabla.*/
```

```
static int buscar_trabajo(const char* trabajo)
{
  int i,tam;
  tam=sizeof(job_list)/sizeof(struct jobs_st);

  for(i=0;i<tam;i++)
    if(!strcmp(trabajo,job_list[i].job_name))
      return i;

  return -1;
}

int main(int argc, char *argv[])
{
  int     i,j;
  FILE*   ordenes;
  char    linea[80];
  char    *accion,*aux;
  int     parametro;
  struct timespec tiempo;

  if (argc!=3)
    {
      fprintf(stderr,"Uso: thread_pool <num_threads> <fichero_ordenes>\n");
      return 1;
    }

  /* Inicialización del pool de threads */
  thread_pool_init(&TP,atoi(argv[1]));

  /* Apertura del fichero de prdenes */
  if((ordenes=fopen(argv[2],"r"))==NULL)
    {
      fprintf(stderr,"Error al abrir el fichero %s\n",argv[2]);
      fprintf(stderr,"Uso: thread_pool <num_threads> <fichero_ordenes>\n");
      perror("fopen");
      return 1;
    }

  while(fgets(linea,80,ordenes)!=NULL)    /* Mientras no se termine el fichero */
    {
      /* Se divide cada linea del fichero en accion (al principio) y parametro. */
      aux=strchr(linea,' ');
      accion=linea;
      if(aux!=NULL)
 {
   *aux='\0';
   parametro=atoi(aux+1);
 }
      else
 parametro=0;

      /* Si la accion se llama SLEEP, entonces se duerme el proceso durante varios segundos. */
      if(!strcmp(accion,"SLEEP"))
 {
   tiempo.tv_sec=parametro;
   tiempo.tv_nsec=0;
   fprintf(stdout,
     "Durmiendo: No hay más trabajos en los próximos %d seg.\n",
     parametro);
   nanosleep(&tiempo,NULL); /* Sleep */
   fprintf(stdout,"Recibiendo nuevos trabajos\n");
```

```
    }
        else
    {
      /* En otro caso se busca en la tabla de trabajos la entrada asociada. */
      if((j=buscar_trabajo(accion))==-1)
        fprintf(stderr,"Trabajo '%s' no es válido\n",accion);
      else
        {
          /* Se obtiene un thread disponible. Si no hay se espera */
          i=get_idle_thread(&TP);
          fprintf(stdout,
             "Asignando %s(%d) a thread %d\n",
             accion,parametro,i);
          /* Se asigna el trabajo al thread */
          if(assign_job(&TP,i,job_list[j].job,parametro))
     fprintf(stderr,"Error asignando trabajo a %d\n",i);
        }
    }
      }

    fprintf(stdout,"No hay más trabajos\n");

    /* Una vez finalizada la asignación de trabajos se espera la terminación de todos los threads. */
    for(j=0;j<atoi(argv[1]);j++)
      {
        i=get_idle_thread(&TP);
        fprintf(stdout,"Liberando thread %d\n",i);
        release_idle_thread(&TP,i);
      }

    return 0;
}
```

Fichero thread_pool.h

```
#ifndef _THREAD_POOL_H_
#define _THREAD_POOL_H_

#include <stdlib.h>
#include <pthread.h>

/* Entrada del thread_pool */
struct thread_pool_entry_t
{
  pthread_t       thread_id;   /* ID del thread */

  void            (*job)(int);/* Puntero a una función que recibe un entero como parámetro.
                                 Esta función se inicializará */
  int             param;       /* Parámetro del trabajo */
  int             status;      /* Indica el estado del thread */
};

#define THREAD_IS_WORKING   1
#define THREAD_IS_IDLE      2
#define THREAD_IS_ASSIGNED  3
#define THREAD_IS_CANCELLED 4

struct thread_pool_t
{
  struct thread_pool_entry_t* thread_pool;    /* Vector de entradas */
  unsigned int                num_threads;    /* Número de threads */
  pthread_cond_t              tp_condition;   /* Variable condición que se usara para asignar
```

```
                                                    trabajos */
  pthread_mutex_t               tp_mutex;       /* Mutex asociado a la condición anterior */
};

/* thread_pool_init
  ================
  Inicializa la estructura del pool de threads. Arranca todos los threads  y los coloca en estado IDLE y a la espera de
  trabajos. Esta función recibe como argumento el número de threads a crear en el pool.*/
void thread_pool_init(struct thread_pool_t* tp,
            unsigned int           num_threads);

/* get_idle_thread
  ===============
  Esta función busca un thread que se encuentre en estado IDLE. Si no hay ningún thread en ese estado espera hasta que
  quede uno disponible.

  NOTA: Esta es una función bloquente.
  El indice devuelto estará reservado hasta que se le asigne una tarea o
  se invoque a la función release_idle_thread que implicara que dicho
  thread se libera y ya no podrá ser solicitado por otra llamada. */
int   get_idle_thread(struct thread_pool_t* tp);

/* assign_job
  ==========
  Esta función asigna el trabajo 'job' al thread pasado como indice. Ademas
  del puntero a la función a ejecutar es necesario pasarle un parámetro.
  Si el thread indicado no está en estado IDLE esta función devuelve un -1
  en otro caso un 0.
*/
int assign_job(struct thread_pool_t* tp,
          int                   index,
          void                  (*job)(int),
          int                   param);

/* release_idle_thread
  ===================
  Libera el thread que se le pasa como parámetro. Este thread sera
  cancelado y ya no podrá ser usado en sucesivas asignaciones de trabajos.*/
void release_idle_thread(struct thread_pool_t* tp,int index);

#Endif
```

Fichero thread_pool.c

```
#include <stdio.h>
#include <signal.h>
#include <strings.h>
#include "thread_pool.h"
/* Implementación de la biblioteca del pool de threads. */


static void *thread_entry_handler(void* arg)
{
  int t;
  struct thread_pool_entry_t* myself=NULL;
  struct thread_pool_t* tp;
  tp=(struct thread_pool_t*)arg;   /* Conversión del tipo del argumento. */

  /* Se hace una espera para que todos los threads arranquen a la vez. */
  pthread_mutex_lock(&tp->tp_mutex);
  pthread_mutex_unlock(&tp->tp_mutex);

  /* Identifico dentro de la estructura TP mi propia entrada */
```

```
  for(t=0;t<tp->num_threads;t++)
    if(tp->thread_pool[t].thread_id==pthread_self())
      myself=&(tp->thread_pool[t]);

  if(!myself)
    {
      fprintf(stderr,"Fallo en thread [%ld]\n",pthread_self());
      pthread_exit(NULL);
    }

  while(1)
    {
      /* Se espera a que se nos asigne un trabajo o se cancele el thread. */
      pthread_mutex_lock(&tp->tp_mutex);
      while(myself->status!=THREAD_IS_ASSIGNED &&
       myself->status!=THREAD_IS_CANCELLED)
   pthread_cond_wait(&(tp->tp_condition),
          &(tp->tp_mutex));

      /* Si el thread se cancela, se libera el semáforo y se hace  un thread_exist. */
      if(myself->status==THREAD_IS_CANCELLED)
   {
     pthread_mutex_unlock(&tp->tp_mutex);
     pthread_exit(NULL);
   }

      /* En otros casos se cambia de estado al thread y se libera el
    semáforo. */
      myself->status=THREAD_IS_WORKING;
      pthread_mutex_unlock(&tp->tp_mutex);

      /* Se ejecuta el trabajo. Esto puede llevar tiempo. */
      (*myself->job)(myself->param);

      /* Se vuelve a tomar el semáforo para cambiar el thread de estado y se manda una señal a los que estén esperando
      en la variable condición. */
      pthread_mutex_lock(&tp->tp_mutex);
      myself->status=THREAD_IS_IDLE;
      pthread_cond_broadcast(&(tp->tp_condition));
      pthread_mutex_unlock(&tp->tp_mutex);
    }

  return NULL;
}


void thread_pool_init(struct thread_pool_t* tp,
           unsigned int          num_threads)
{
  int i;

  /* Se reserva la estructura del tamaño indicado */
  tp->num_threads=num_threads;
  tp->thread_pool=
    (struct thread_pool_entry_t*) malloc(sizeof(struct thread_pool_entry_t)*
             num_threads);

  /* Se inicializan las variables semafor y condicion */
  pthread_cond_init(&(tp->tp_condition),NULL);
  pthread_mutex_init(&(tp->tp_mutex),NULL);

  /* Se cierra el semáforo para hacer que no se disparen los threads hasta que no se ha concluido la inicialización de
  todos. */
```

```
  pthread_mutex_lock(&tp->tp_mutex);

  for(i=0;i<num_threads;i++)
    {
      tp->thread_pool[i].job=NULL;
      tp->thread_pool[i].status=THREAD_IS_IDLE;
    }

  /* Se crea cada thread. */
  for(i=0;i<num_threads;i++)
    pthread_create(&(tp->thread_pool[i].thread_id),   /* Thread ID */
        NULL,                          /* Sin atributos */
        thread_entry_handler,          /* Manejador del thread */
        (void*)tp);                    /* Como argumento pasamos el pool de threads */
  pthread_mutex_unlock(&tp->tp_mutex);
}

int  get_idle_thread(struct thread_pool_t* tp)
{
  int i=0;
  /* Se solicita el semáforo */
  pthread_mutex_lock(&tp->tp_mutex);

  /* La condición de espera es que se libere alguno de los threads */
  while (1){
    for(i=0;i<tp->num_threads;i++)
      if(tp->thread_pool[i].status==THREAD_IS_IDLE)
   return i;
    pthread_cond_wait(&(tp->tp_condition),
           &(tp->tp_mutex));
  }
}

int assign_job(struct thread_pool_t* tp,
         int                   index,
         void                  (*job)(int),
         int                   param)
{
  /* Si el thread no estaba idle se debe a un error */
  if(tp->thread_pool[index].status!=THREAD_IS_IDLE)
    {
      pthread_mutex_unlock(&tp->tp_mutex);
      return -1;
    }

  /* Se asigna función y parámetro al thread y se cambia su estado. */
  tp->thread_pool[index].job=job;
  tp->thread_pool[index].param=param;
  tp->thread_pool[index].status=THREAD_IS_ASSIGNED;

  /* Se notifica que la variable condición ha cambiado */
  pthread_cond_broadcast(&(tp->tp_condition));
  pthread_mutex_unlock(&tp->tp_mutex);

  return 0;
}

void release_idle_thread(struct thread_pool_t* tp, int index)
{
  void* ret;

  /* Se cambia el thread a estado de cancelación y se notifica de posibles cambios en la condición*/
  tp->thread_pool[index].status=THREAD_IS_CANCELLED;
```

```
    pthread_cond_broadcast(&(tp->tp_condition));
    pthread_mutex_unlock(&tp->tp_mutex);

    /* Se espera a recoger el estado del thread cancelado */
    pthread_join(tp->thread_pool[index].thread_id,&ret);
}
```

Problema 4.5 (septiembre 2001)

Se desea realizar una comunicación entre dos procesos con tuberías sin nombre. Se plantean dos posibilidades.

A.- Comunicación unidireccional entre dos procesos. (Con una única tubería)

***El padre** deberá leer de la entrada estándar (teclado) y enviarle la información al hijo a través de una tubería sin nombre. Si el mensaje que recibe el padre por la entrada estándar es 'FIN', después de mandárselo al hijo por la tubería, el padre terminará.*

***El hijo** deberá estar leyendo de forma continua del otro extremo de la tubería sin nombre y desviará el mensaje hacia la salida estándar (monitor). Si recibe el mensaje 'FIN' el proceso hijo terminará correctamente.*

B.- Comunicación bidireccional entre dos procesos. (Con dos tuberías)

***El padre** será igual que en el caso anterior, pero antes de volver a leer de la entrada estándar, esperará la recepción del mensaje 'LISTO' que le deberá llegar por una segunda tubería sin nombre que le conecta con el hijo.*

***El hijo** será igual que el caso anterior, pero inmediatamente después de escribir el mensaje por la salida estándar, enviará al padre un mensaje de 'LISTO' para recibir más datos. Este mensaje se lo enviará por la segunda tubería sin nombre.*

Se pide:

*a) Realizar un esquema de cómo sería el **programa A**, señalando el sentido de ejecución, llamadas al sistema, situación de las tuberías, etc.*
*b) Ídem para el **programa B**.*
c) Escribir el seudo código el programa A.
d) Escribir el seudo código el programa B.

Solución

a) En la figura Error: Reference source not found aparece el esquema de lo que se pide para una comunicación unidireccional entre dos procesos. El padre crea una tubería para comunicarse con el hijo(llamada *pipe()*). Seguidamente crea el proceso hijo (llamada *fork()*). El padre estará en un bucle infinito esperando datos del teclado y mandándoselos al hijo por la entrada de la tubería. Si se recibe el mensaje FIN por teclado, el padre escribirá este mensaje en la tubería y terminará. El hijo estará en otro bucle infinito leyendo de la salida de la tubería y enviando lo recibido a pantalla. Si lo que recibe por esta es el mensaje FIN, éste terminará.

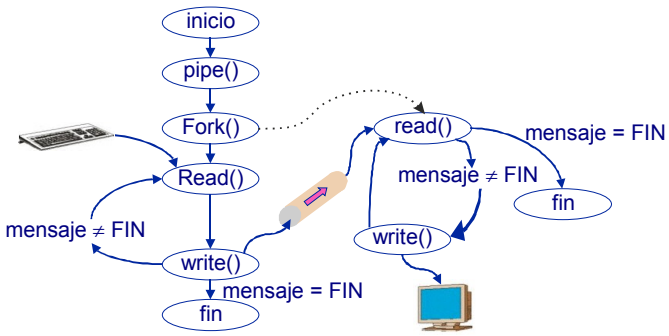


Figura 4.2

b) La iniciación es similar al caso anterior, con diferencia que ahora se crean dos tuberías. El padre estará en un bucle infinito esperando por recibir información del teclado. Al igual que en el caso anterior todo lo recibido lo mandará por una tubería. En caso de recibir el mensaje FIN del teclado, también se mandará este mensaje y se finalizará. Para el resto de mensajes, el padre esperará, leyendo en otra tubería, la confirmación de que el hijo ha leído el men-

saje, esperando por la palabra LISTO. Cuando haya recibido este reconocimiento, el padre volverá a leer datos del teclado.

El hijo, al igual que el caso anterior, espera los datos por la primara tubería, sacará lo recibido por pantalla, salvo que el mensaje recibido sea FIN. En este caso terminará su ejecución. Después de escribir el mensaje por pantalla, el hijo enviará el mensaje LISTO al padre, escribiendo en la entrada de la segunda tubería. Véase la figura Error: Reference source not found.

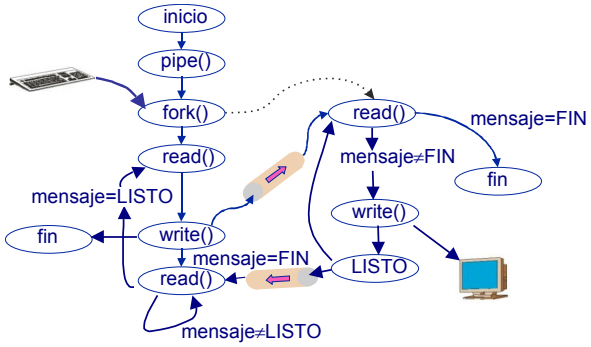


Figura 4.3

c) El seudo código para el caso de comunicación unidireccional es:

```
comienzo
  pipe(tubería)
  pid = fork()

/* Proceso Padre */

  Si pid != 0
    close(tuberia[0])
      repetir
        read (0 , mensaje)
        write(tuberia[1], mensaje)
      hasta mensaje = "FIN"
    close (tubería[0])
    exit(0)
  fin-si

/*Proceso hijo */

  Si pid = 0
    close(tuberia[1])
      repetir
        read (tubería[0], mensaje)
        si mensaje != "FIN"
           write(1, mensaje)
        fin-si
      hasta mensaje = "FIN"
    close (tubería[0])
    exit(0)
  fin-si
fin
```

d) El seudo código para el caso de comunicación bidireccional es:

```
comienzo
  pipe(tuberia_padre_al_hijo)
  pipe(tuberia_hijo_al_padre)
  pid = fork()

/* Proceso Padre */

  Si pid != 0
    close(tuberia_padre_al_hijo[0])
    close(tuberia_hijo_al_padre[1])
      repetir
        read(0, mensaje)
```

```
        write(tuberia_padre_al_hijo[1], mensaje)
        si mensaje != "FIN"
          repetir
             read(tuberia_hijo_al_padre[0],mensaje2)
          hasta mensaje2 = "LISTO"
        fin-si
      hasta mensaje = "FIN"
    close(tuberia_padre_al_hijo[1])
    close(tuberia_hijo_al_padre[0])
    exit(0)
  fin-si

/*Proceso hijo */


  Si pid != 0
    close(tuberia_padre_al_hijo[1])
    close(tuberia_hijo_al_padre[0])
      repetir
        read(tuberia_padre_al_hijo[0], mensaje)
        si mensaje != "FIN"
          write(1, mensaje)
          write(tuberia_hijo_al_padre[1], "LISTO")
        fin-si
      hasta mensaje = "FIN"
    close(tuberia_padre_al_hijo[0])
    close(tuberia_hijo_al_padre[1])
    exit(0)
  Fin-si
```

Problema 4.6

Dado el fichero '/home/fich' con un tamaño inicial de 16 KiB, se desea ejecutar sobre un sistema UNIX el siguiente fragmento de código:

```
struct point {
   int x;
   int y;
};
struct rect {
   struct point pt1;
   struct point pt2;
};
struct point *pp;
struct rect *pr;

int main(void)
{
int fd;
struct stat bstat;
int i;
void *p;
... /* Código de sincronización */
fd = open ("/home/fich", O_RDWR);
fstat(fd, &bstat);
p= mmap((c_addr_t) 0,bstat.st_size,PROT_READ | PROT_WRITE, MAP_SHARED, fd,0);
close(fd);
if (fork() =! 0){   /* Código del padre */
   pr = p;
   for ( i=0; i<=MAX_IT; i++){
      ... /* Código de sincronización con el hijo*/
     pr->pt1.x = 3*i;
     pr->pt1.y =5*i;
```

```
      pr->pt2.x = 1;
      pr->pt2.y = 7*i;
      pr++;     /* Avanza hasta el inicio de la siguiente estructura */
      ... /* Código de sincronización con el hijo*/
      }
   }else{   /* Código del hijo */
   pp = p;
   for ( i=0; i<=MAX_IT; i++){
      ... /* Código de sincronización con el padre*/
      printf("coordenada x:%d,coordenada y:%d\n",pp->x, pp->y)
      pp++;   /* Avanza hasta el inicio de la siguiente estructura */
      ... /* Código de sincronización con el padre*/
      }
   }
}
```

Si el código de sincronización hace que ejecute primero el proceso padre, después el proceso hijo y así sucesivamente, se pide:

a) Indicar qué valores escribe el proceso hijo por pantalla durante las cinco primeras iteraciones. Justificar razonadamente la respuesta.

b) Escribir el código para realizar la sincronización necesaria utilizando el mecanismo de semáforos.

c) Si MAX_IT vale 1000 y un entero ocupa 4 bytes, calcular el número de veces que se activa el sistema operativo durante la ejecución del programa, si se tiene un sistema de memoria virtual con paginación por demanda, sin preasignación de swap y con un tamaño de página de 4K. Considerar que en memoria principal hay espacio suficiente para albergar todas las páginas necesarias, sin necesidad de realizar ninguna expulsión.

d) Si se tiene un sistema que crea una región independiente para el fichero proyectado, ¿qué ocurre si MAX_IT vale 2000?

Solución

a) En el código del programa se proyecta el fichero '/home/fich' en memoria como una zona de datos compartida. De esta forma, tanto el padre como el hijo están accediendo a la misma zona de memoria y los cambios que realice cualquiera de ellos serán vistos por el otro. Además hay que tener en cuenta que el padre está accediendo al fichero proyectado con una estructura diferente a la del hijo, y que esta estructura tiene exactamente el doble de tamaño.

Los valores escritos en pantalla por el hijo son los siguientes:

Iteración	Valor 1	Valor 2
1	0	0
2	1	0
3	3	5
4	1	7
5	6	10

b) No es posible realizar la implementación con un solo semáforo, ya que entonces no se podría tener control total sobre la ejecución del padre y del hijo y no se podría realizar la alternancia necesaria. Es por tanto necesario utilizar dos semáforos, uno para controlar la ejecución del padre y otro para controlar la ejecución del hijo. El código quedaría de la siguiente manera:

```
/* Inicialización de los semáforos */
sem_t sem_padre, sem_hijo;
sem_init (&sem_padre, 1, 1); /* El padre es el primero en ejecutar */
sem_init (&sem_hijo, 1,0); /* A la espera de que el padre le de paso */
   /* Código del padre */
   for (...){
      sem_wait(sem_padre);
      …
      sem_post(sem_hijo);
```

```
    }
    /* Código del hijo */
    for (...){
       sem_wait(sem_hijo);
       …
       sem_post(sem_padre);
    }
/* Liberamos los recursos utilizados */
sem_destroy(&sem_padre);
sem_destroy(&sem_hijo);
```

c) Se deberá tener en cuenta que el sistema operativo se activa con las llamadas al sistema y con los fallos de página. Del total de 16 KiB (16.384 bytes) que ocupa el fichero, el padre va a acceder a: 4 enteros * 4 bytes/entero * 1001 iteraciones = 16.016 bytes. Inicialmente, al proyectar el fichero, no se produce ningún fallo de página, éstos se irán produciendo según el padre vaya escribiendo sus valores. Al finalizar la ejecución habremos accedido a 4 páginas.

En cuanto a llamadas al sistema las que se ejecutan una vez son las siguientes: sem_init y sem_destroy del padre, sem_init y sem_destroy del hijo, open, fstat, mmap y close del fichero y fork para la creación del proceso hijo. Las llamadas que se repiten más de una vez son sem_wait y sem_post tanto en el padre como en el hijo y se realizan un total de 1001 veces cada una.

Por tanto, el número total de activaciones es:

4 + 9 + 4*1001 = 4017 activaciones

d) Si el número máximo de iteraciones es 2000 implica que el padre estaría intentando escribir fuera de la imagen del proceso. La MMU al realizar la traducción de la dirección errónea generaría una excepción de *referancia a memoria inválida* y el kernel mandaría una señal SIGSEGV al proceso para matarle.

Problema 4.7

Se pretende resolver un problema de comunicación y sincronización entre procesos ligeros empleando semáforos. Se trata del típico problema de lectores y escritores. Se dan las siguientes directrices:

- *El proceso ligero principal debe crear 'MAX_LECTORES' procesos ligeros lectores y 'MAX_ESCRITORES' procesos ligeros escritores, que van a competir por un recurso común. En este caso, el recurso común es un simple dato de tipo entero (dato).*
- *Cada proceso ligero lector deberá leer el recurso en exclusión mutua frente a escritores. Es decir, mientras un lector esté accediendo al recurso compartido ningún proceso ligero escritor podrá acceder a este.*
- *De manera contraria, si el recurso está ocupado por un escritor, el lector deberá esperar que el escritor termine. Sin embargo el proceso ligero lector no debe mantener exclusión mutua frente a otros procesos ligeros lectores, ya que la lectura no es destructiva.*
- *El proceso ligero escritor debe mantener la exclusión mutua frente a procesos lectores, como ya se dijo, y también frente a procesos escritores.*

Se plantea en un primer momento el siguiente código, pero la experiencia comprueba que es erróneo:

```
01 int dato = VALOR_INICIAL;        /* recurso */
02 sem_t semaforo;                  /* acceso a dato */
03
04 int main(void)
05 {
06   pthread_t th_lector[MAX_LECTORES], th_escritor[MAX_ESCRITORES];
07 int i;
08
09   sem_init(&semaforo, 0, 1);
10   for (i=0; i<MAX_LECTORES; I++)
11        pthread_create(&th_lector[i], NULL, Lector, NULL);
12   for (i=0; i<MAX_ESCRITORES; I++)
13        pthread_create(&th_escritor[i], NULL, Escritor, NULL);
14
```

```
15    for (i=0; i<MAX_LECTORES; i++)
16          pthread_join(th_lector[i], NULL);
17    for (i=0; i<MAX_ESCRITORES; i++)
18          pthread_join(th_escritor[i], NULL);
19
20    /*Cerrar todos los semáforos*/
21    sem_destroy(&semaforo);
22
23    return 0;
24  }
25
26  void Lector(void)  /* código del lector */
27  {
28    sem_wait(&semaforo);
29    printf("%d\n", dato);   /* leer dato y mostrar el valor*/
30    sem_post(&semaforo);
31    pthread_exit(0);
32  }
33
34  void Escritor(void)   /* código escritor */
35  {
36    sem_wait(&semaforo);
37    dato = dato + 2;   /* modificar recurso */
38    sem_post(&semaforo);
39
40    pthread_exit(0);
41  }
```

NOTA: *Supóngase* `VALOR_INICIAL, MAX_ESCRITORES, MAX_LECTORES` *cualquier valor.*

Se pide:

a) ¿Qué fallo tiene el código?

b) ¿Qué instrucciones darías a un informático para que pueda corregirlo?

c) Escribir el código corregido con el consejo que propones. Solamente escribe las líneas de código que hay que añadir, indicando entre que líneas deben ser colocadas.

d) Escribir el código del escritor realizado con con mutex y variables de condición.

Solución

a) El fallo es que está haciendo exclusión mutua entre lectores, y el enunciado dice explícitamente que no debe hacerse. Varios lectores deberían poder acceder simultáneamente a la sección crítica.

b) Ya que se está inicializando la variable semáforo a 1, es decir se está empleando como un *mutex*, una posibilidad sería que se establezca un contador de procesos lectores. Este contador debe ser accesible entre todos los procesos ligeros lectores, y por tanto debe ser una variable global. Antes de acceder al recurso, y por tanto, de intentar decrementar la variable semáforo, el proceso ligero lector deberá incrementar este contador, y cuando se abandone el recurso, incrementar el semáforo, deberá decrementar este contador.

Mientras el contador sea distinto de cero, indicará que hay lectores, y por tanto la variable semáforo no deberá ser incrementada. El recurso es de los lectores.

Cuando un lector termine, si el número de lectores es cero, entonces éste debe liberar el recurso compartido, para permitir que puedan entrar escritores.

Nótese que esta variable contador de lectores, es una variable global entre todos los lectores, ya que todos la pueden modificar. Cualquier operación de modificación debe ser atómica, y por tanto hace falta otro semáforo binario o mutex para asegurar esto.

c) Las líneas de código que faltan serían las siguientes:

En la línea 3 inicializo las dos variables globales que voy a usar.

```
int n_lectores = 0;      /* num. Lectores */
  sem_t sem_lec;        /* acceso n_lectores */
```

En la línea 9

```
sem_init(&sem_lec, 0, 1);
```

En la línea 22

```
sem_destroy(&sem_lec);
```

Antes ocupar el semáforo (línea 28) me aseguro que no hay otros lectores, si los hay ya tienen el semáforo

```
sem_wait(&sem_lec);
n_lectores = n_lectores + 1;
if (n_lectores == 1)
    Sem_wait(&semaforo);   /* Se bloquea si está ocupado por el escritor */
sem_post(&sem_lec);        /* Se libera el semáforo de lectores */
printf("%d\n", dato);       /* Leer dato y mostrar el valor */
```

Antes de liberar el semáforo (línea 30) me aseguro que no quedan lectores. Si quedan no libero el recurso.

```
sem_wait(&sem_lec);
n_lectores = n_lectores - 1;

if (n_lectores == 0)
   sem_post(&semaforo);    /* Libero el recurso */
sem_post(&sem_lec);
pthread_exit(0);
```

d) Sería el mismo que las líneas 34 a 41, pero sustituyendo sem_wait() por pthread_mutex_lock() y sem_post() por pthread_mutex_unlock().

Problema 4.8 (septiembre 2003)

Se desea implementar un programa (multi_grep) que se encargue de llevar a cabo la búsqueda de un determinado patrón en múltiples ficheros de texto. Su sintaxis es la siguiente:

```
$ multi_grep patron fichero1 fichero2 ... ficheroN
```

Para cada uno de los ficheros, el programa multi_grep debe crear un thread diferente, cada uno de los cuales debe buscar el patrón sobre su correspondiente fichero. La búsqueda del patrón sobre los ficheros debe ***hacerse en paralelo****.*

Cuando un thread encuentra el patrón buscado en alguna línea, debe insertar dicha linea en una estructura en memoria compartida por todos los threads. Esta estructura no debe modificarse de forma concurrente. Para llevar a cabo la inserción, se debe utilizar la función insertar_linea (), que tiene la siguiente sintaxis:

void *insertar_linea (****char*** **nombre_fich,* ***int*** *num_linea,* ***char*** **linea);*

Esta función recibe el nombre del fichero, el número de línea y la línea donde se ha encontrado un determinado patrón. La función insertar_linea() no debe utilizarse de forma concurrente, puesto que modifica la estructura que almacena las líneas encontradas. La función insertar_linea ***no hay que implementarla****, sólo utilizarla de forma correcta.*

Al finalizar la búsqueda del patrón en todos los ficheros, el programa deberá imprimir por la salida estándar todas aquellas líneas en las cuales aparece el patrón buscado. Para ello, se podrá utilizar la función imprimir_lineas(), que tiene la siguiente sintaxis:

void *imprimir_lineas ();*

y que escribe por la salida estándar y en orden todas las líneas almacenadas en la estructura.

Para buscar un patrón sobre un determinado fichero, se pueden utilizar las siguientes funciones, que ***no hay que implementar****:*

char ** leer_linea (* ***int*** *fd);*

Esta función lee línea a línea un fichero cuyo descriptor es fd. Devuelve NULL cuando llega al final de fichero.

int *esta_patron (char *patron, char *linea);*

Esta función devuelve 1 si el patrón se encuentra en la línea y 0 si no se encuentra.

No considere los casos de error. Si necesita utilizar algún mecanismo de sincronización, utilice el mecanismo de mutex.

Se pide:

a) Implementar el programa multi_grep, así como el código asociado a cada uno de los threads que tratan los ficheros, utilizando las funciones descritas.

b) ¿Sería equivalente la solución si en lugar de utilizar mutex se utilizaran semáforos?

c) Implementar un nuevo programa `coincidencias` que búsque las líneas en las que aparece el patrón "`mensaje`" en los ficheros `data1.txt`, `data2.txt` y `data3.txt` y contabilice el número de ellas, escribiéndolo en el fichero `num_ocurrencias.dat`. Para ello utilice:

- *El ejecutable `multi_grep` visto anteriormente.*
- *El mandato `wc` con el argumento `-l`. Dicho mandato cuenta el número de lineas que recibe por la entrada estándar y devolve el valor por la salida estándar.*
- *La salida debe ser escrita en el fichero anteriormente indicado.*

***Nota**: No modifique el código de `multi_grep`, utilícelo con la sintaxis indicada al comienzo del ejercicio (suponga que sólo se dispone del fichero binario).*

Solución

a)

```
#include <pthread.h>
   /* resto de includes */
struct argumentos {
   char *patron;
   char *nombre_fichero;
};
pthread_mutex_t mutex;

/* Programa multi_grep */
int main (int argc, char *argv[]) {
     pthread_t *thread_id;
     struct argumentos *arg;
     int i;

     /* Comprobación de los argumentos */
     if (argc < 3) {
        fprintf(stderr, "Error. Uso: multigrep patron f1 [f2].. [fn]\n");
        exit(1);
     }
     /* Reservamos espacio para los identificadores de los threads */
     thread_id = (pthread_t *) malloc ((argc-2)*sizeof(pthread_t));

     /* Inicializamos el mutex */
     pthread_mutex_init(&mutex,  NULL);

     /* Creamos los threads */
     for (i=2; i<argc; i++) {
        arg = (struct argumentos *) malloc(sizeof(struct argumentos));
        arg.patron=argv[1];
        arg.nombre_fichero=argv[i];
        pthread_create (&(thread_id[i-2]), NULL, simple_grep, &arg);
     }

     /* Esperamos a la  finalización de los threads */
     for (i=0; i<argc-2; i++) {
        pthread_join(thread_id[i],NULL);
     }
     /* Destruimos el mutex */
```

```
      pthread_mutex_destroy(&mutex);

   /* Imprimimos las líneas y salimos */
      imprimir_lineas();
      return 0;
}

/* Código de los threads */
void simple_grep (struct argumentos *arg){
   int fd;
   char *linea;
   int num_linea = 0;

   fd = open(arg->nombre_fichero, O_RDONLY);
   linea = leer_linea(fd);
   while (linea != NULL) {
      num_linea++;
      if (esta_patron(arg->patron, linea))  {
         /* Entrada en la sección crítica */
         pthread_mutex_lock(&mutex);
         insertar_linea(arg->nombre_fich, num_linea, linea);
         /* Salida de la sección crítica */
         pthread_mutex_unlock(&mutex);
      }
      linea = leer_linea(fd);
   }
   free(arg);
   pthread_exit(NULL);
}
```

b) Dado que utilizamos un mutex para crear una sección crítica, el uso de un semáforo binario sería equivalente. Este semáforo debería ser inicializado a 1. No obstante, habría que modificar las directivas correspondientes (sem_wait y sem_post, para la entrada y la salida de la sección crítica respectivamente). Por otro lado, los mutex suelen ser más apropiados para threads que los semáforos.

c)

```
/* Programa coincidencias */
int main(void){
   int fd, pfd;
   int pid;

   pipe(pfd);
   pid=fork();

   switch (pid){
      case -1:
         perror("fork");
         exit(1);
      case 0:
         close(pfd[0]);
         close(1);
         dup(pfd[1]);
         close(pfd[1]);
         execlp("multi_grep", "multi_grep", "data1.txt", "data2.txt",
            "data3.txt", NULL);
         perror("exec");
         exit(1);
      default:
         close(pfd[1]);
         close(0);
         dup(pfd[0]);
         close(pfd[0]);
```

```
        close(1);
        fd=open("num_ocurrencias.dat", O_WRONLY | O_CREAT | O_TRUNC, 0600);
        execlp("wc", "wc", "-l", NULL);
        perror("exec");
        exit(1);
    }
}
```

Problema 4.9 (2004)

Sea el código adjunto, que es un esqueleto incompleto y/o erróneo de un servidor genérico que presta su servicio a procesos remotos vía sockets.

```
    /* ServidorGenerico.c   */
/* 1*/  int main(int argc, char *argv[])
/* 2*/  {
/* 3*/       int sd, cd, size;
/* 4*/       struct sockaddr_in s_ain, c_ain;
/* 5*/
/* 6*/       sd = socket( ???? );
/* 7*/
/* 8*/       bzero((char *) &s_ain, sizeof(s_ain));
/* 9*/       s_ain.sin_family = AF_INET;
/*10*/       s_ain.sin_addr.s_addr = INADDR_ANY;
/*11*/       s_ain.sin_port = htons(atoi(argv[1]));
/*12*/       bind(sd, (struct sockaddr *) &s_ain, sizeof(s_ain));
/*13*/
/*14*/       listen(sd, 5);
/*15*/       while (1) {
/*16*/           size = sizeof(c_ain);
/*17*/           cd = accept(sd, (struct sockaddr *) &c_ain, &size);
/*18*/
/*19*/           ServidorDedicado(sd, cd);
/*20*/       }
/*21*/  }
```

Conteste a las siguientes preguntas para completar y/o corregir los errores presentes en este código.

a) En vista del resto del código presentado, ¿cuál sería la versión de la llamada a `socket` de la línea 6?

Considere ahora las siguientes estrategias alternativas para el diseño de la función `ServidorDedicado`:

- *Estrategia 1, basada en Procesos Pesados:*
- *El proceso `ServidorGenerico` original crea un proceso hijo por cada cliente para que preste un determinado `ServicioConcreto`. Habrá múltiples procesos, con un único hilo de ejecución cada uno. El proceso padre no espera a que terminen sus hijos.*
- *Estrategia 2, basada en Procesos Ligeros:*
- *El proceso `ServidorGenerico` original crea un hilo de ejecución por cada cliente para que le preste un determinado `ServicioConcreto`. Habrá un único proceso (el original) con múltiples hilos de ejecución. El hilo principal no espera a que terminen los demás hilos.*

b) Suponiendo que el `ServicioConcreto` que finalmente se desea implementar sea equivalente al `telnet` (terminal remoto), vía el cual un usuario remoto pueda trabajar en el sistema interactuando con un intérprete de mandatos, ¿cuál de las dos estrategias de diseño se adaptaría mejor a este uso?

Sean las siguientes implementaciones de las estrategias de diseño indicadas anteriormente:

```
    /* Implementación 1: Procesos Pesados   */
/* 1*/  void ServidorDedicado(int sd, int cd)
/* 2*/  {
/* 3*/       switch (fork()) {
```

```
/* 4*/       case -1:
/* 5*/           perror("Servidor");
/* 6*/           exit(1);
/* 7*/       case 0:
/* 8*/           close(sd);
/* 9*/           ServicioConcreto(cd);
/*10*/
/*11*/
/*12*/       default:
/*13*/           close(cd);
/*14*/       }
/*15*/   }
```

```
   /* Implementación 2: Procesos Ligeros  */
/* 1*/   void * HiloServicio(void * cdp)
/* 2*/   {
/* 3*/       int cd = *(int *)cdp;
/* 4*/       free(cdp);
/* 5*/       ServicioConcreto(cd);
/* 6*/       close(cd);
/* 7*/       return NULL;
/* 8*/   }
/* 9*/   void ServidorDedicado(int sd, int cd)
/*10*/   {
/*11*/       pthread_t th;
/*12*/       int * cdp = malloc(sizeof(int));
/*13*/       *cdp = cd;
/*14*/       pthread_create(&th, NULL, HiloServicio, cdp);
/*15*/   }
```

c) ¿Cuál de las dos implementaciones es incorrecta (presenta errores de concepto)?

d) Escriba el código de la función `ServicioConcreto` para que implemente un servicio de eco, donde cada cliente remoto reciba de vuelta cada byte de información que envíe al servidor.

En un instante determinado de su ejecución el `ServidorGenerico` tiene:

- *7 clientes que fueron ya completamente servidos y terminaron.*
- *5 clientes que están siendo servidos actualmente, es decir, hay un `ServicioConcreto` ejecutando para cada uno.*
- *4 clientes que están intentando obtener servicio, pero su conexión no ha sido aceptada todavía.*

e) Si `ServidorDedicado` sigue la implementación 2, de procesos ligeros, ¿cuál será el número total de descriptores de fichero asociados a sockets (incluido el `sd` original) en uso en el proceso `ServidorGenerico` en dicho instante determinado?

A partir de este momento, considere exclusivamente la estrategia 2, de procesos ligeros.

```
   /* ServicioConcreto */
001: void ServicioConcreto(int cd) {
002: /* …
...      …
017:     … */
018:     BEGIN REGION_CRITICA 0
019:         BEGIN REGION_CRITICA 1
020:             cnt = cnt + 1;
021:             first_cmd = cnt;
022:         END   REGION_CRITICA 1
023:     /* …
...          …
105:         … */
106:         BEGIN REGION_CRITICA 2
107:             colocar en la cola la petición etiquetada con first_cmd
```

```
108:          END   REGION_CRITICA 2
109:      END   REGION_CRITICA 0
110:      /* …
...           …
218:          … */
219:      BEGIN REGION_CRITICA 0
220:          BEGIN REGION_CRITICA 1
221:              cnt = cnt + 1;
222:              second_cmd = cnt;
223:          END   REGION_CRITICA 1
224:      /* …
...           …
300:          … */
301:          BEGIN REGION_CRITICA 2
302:              colocar en la cola la petición etiquetada con second_cmd
303:          END   REGION_CRITICA 2
304:      END   REGION_CRITICA 0
305:      return;
306: }
```

La función `ServicioConcreto` representa el grueso del proceso de servicio. Esta rutina utiliza y modifica varias variables globales. Para evitar que los múltiples hilos concurrentes provoquen valores incoherentes de estas variables, consideraremos "región crítica" cada fragmento de código donde se accedan dichas variables.

f) ¿Cuáles son los servicios POSIX del sistema operativo más apropiados para la defensa de este tipo de región crítica?

Si en un determinado instante T de la ejecución de este programa en una máquina monoprocesador: el valor de `cnt` es 5 y hay tres hilos de dentro `ServicioConcreto` en los puntos thA:017, thB:107 y thC:218.

g) Si, a partir del instante T, continuase ejecutando el hilo A, ¿hasta qué línea podría ejecutar (o dónde se bloquearía)?

h) Si, a partir del instante T, continuase ejecutando el hilo B, ¿hasta qué línea podría ejecutar (o dónde se bloquearía)?

i) Si, a partir del instante T, continuase ejecutando el hilo C, ¿hasta qué línea podría ejecutar (o dónde se bloquearía)?

j) Si, a partir del instante T, primero ejecutase A (hasta donde pudiese) luego B y luego C, ¿qué valor habría tomado la variable compartida `cnt`?

SOLUCIÓN

a) Dado que el código de ServidorGenerico contiene un bucle donde se utiliza la llamada accept, y dado que esta llamada se utiliza para aceptar conexiones, el socket necesario debe estar orientado a conexión, es decir, debe ser de tipo stream, y por lo tanto, el protocolo de transporte necesario debe ser TCP.

La llamada correcta será `sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);`

b) El servicio concreto descrito en el enunciado (equivalente a telnet) es un ejemplo muy claro de un servidor "con estado". Todo usuario, al interactuar con un intérprete de mandatos "cambia" el estado de este intérprete (implícita o explícitamente) al ejecutar mandatos internos (ej. cd) o al establecer variables de entorno.

La estratégia de implementación más adecuada para un servidor con estado es aquella basada en procesos pesados ya que el programador del servicio no tendrá que preocuparse de mantener esta idea de estado por cada cliente servido, sino que será el sistema operativo el que se encargue automáticamente de hacerlo.

Luego la estratégia más adecuada es la 1, basada en Procesos Pesados.

c) La segunda implementación, basada en procesos ligeros es completamente correcta. El paso del único parámetro que el hilo de servicio necesita se realiza a través de memoria dinámica, y es este hilo el que se encarga de liberar la memoria dinámica así como de cerrar el descriptor del socket de servicio cuando este termine.

Sin embargo, la primera implementación no es completamente correcta. El problema reside en que el proceso creado para la prestación del servicio concreto no termina dónde debería. Haría falta una llamada exit(0) en las líneas 10 u 11. Sin esta llamada, cada proceso hijo creado continuará ejecutando el código que le corresponde al pro-

ceso Servidor Genérico original. Si quisiesemos hacer más evidente que, una vez concluida la prestación del servicio concreto no nos hace falta ya el descriptor de servicio, podríamos hacerlo haciendo una llamada close(cd) en la línea 10.

d)

```
   /* ServicioConcreto */
void ServicioConcreto(int cd)
{
        unsigned char byte;
        while( recv(cd, &byte, 1, 0) > 0 )
                send(cd, &byte, 1, 0);
}
```

e) Estarán en uso: el descriptor de aceptación de conexiones (sd) más un descriptor de servicio (cd) por cada cliente que esté actualmente en servicio.

Los descriptores de servicio de los clientes que fueron ya completamente servidos y terminaron fueron cerrados por el correspondiente hilo de servicio.

Los clientes que están intentando obtener servicio, no habrán superado todavía la fase de aceptación de conexión (llamada accept del servidor), luego, todavía no existirá para ellos un descriptor de conexión.

Luego la respuesta correcta es 6.

f) Para sincronizar procesos ligeros (también llamados hilos, en inglés threads) las primitivas más adecuadas, por ligéras y cómodas de usar, son los mutex y las variables condicionales.

Otros mecanismos de sincronización, como por ejemplo semáforos sobre memoria proyectada compartida, serían los adecuados si lo que quisiéramos sincronizar fueran procesos pesados.

g) Lo primero que hay que hacer notar es que los dos segmentos código: de la línea 19 a la 108 y de la línea 220 a la 303, son "la misma región crítica", la región crítica número 0. Los son porque, muy probablemente, en ambos segmentos se manipulan las mismas variables, y no se desea que pueda haber hilos ejecutando simultáneamente dentro de un segmento y del otro.

Lo mismo ocurre con las regiones críticas 1 (líneas 20 a 21 y 221 a 222) y 2 (líneas 107 y 302).

En el instante T, el hilo A está fuera de toda sección crítica, y lo siguiente que hará será intentar entrar en (el primer segmento de) la región crítica 0. La situación del hilo C es semejante, fuera de toda sección crítica e intentará entrar en el segundo segmento de la región crítica 0.

Sin embargo, el hilo B está en posesión de las regiones críticas 0 y 2, pero no de la 1.

Dada esta situación en el intante T, si el siguiente en ejecutar fuese el hilo A, quedaría bloqueado en la línea 18, compitiendo por entrar en la sección crítica 0 (en posesión de B).

h) Dada la explicación del apartado anterior, y entendiendo que los demás hilos están parados, el hilo B podría ejecutar libremente hasta salir de la función por el return de la línea 305.

Primero liberaría las regiones críticas 2 y 0, para luego volver a tomar la 0, pasar por la 1 y la 2, liberar la 0 y terminar.

i) Al igual que en el caso del hilo A, el hilo C no podrá ir más allá de la línea 219, donde quedará bloqueado intentando entrar en la región crítica número 0 que está en posesión del hilo B.

j) Vista la explicación dada en los apartados anteriores, queda claro que A no modificará la variable cnt. En la ejecución del hilo B la variable cnt será incrementada una vez, en la línea 221. Una vez terminado el hilo B, el hilo C podrá ejecutar libremente hasta salir de la función, y en su camino, volverá a incrementar la variable compartida cnt al ejecutar la línea 221.

Luego finalmente la variable cnt habrá sido incrementada dos veces, y su valor final será 7.

Problema 4.10 (2004)

Se desea realizar un sistema de envío de mensajes cortos (SMS), a través de una serie de módems GSM.

Todos los mensajes a ser enviados tienen que almacenarse en un buffer circular acotado de tamaño N mensajes. De esta forma, los procesos que deseen enviar algún mensaje, deberán rellenar el siguiente hueco libre de dicho buffer con la siguiente información:

```
struct mensaje{
   long int destino; // Número teléfono de destino
   long int origen;  // Número de teléfono de origen
   char sms[160];    // Cuerpo del mensaje
}
```

Para implementar este sistema se plantea la estructura de procesos de la figura Error: Reference source not found:

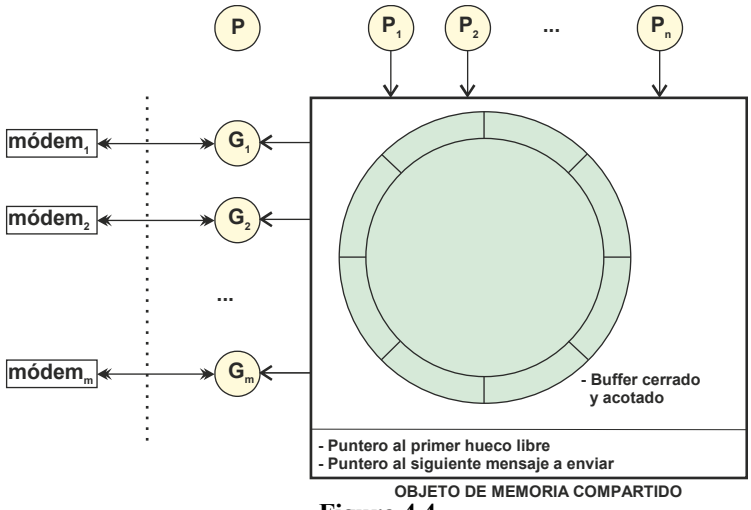


Figura 4.4

- *Un proceso general P, que solicitará al SO la creación del buffer (objeto de memoria compartido con nombre) y todos los mecanismos de sincronización necesarios.*
- *Los procesos P_i que son los que desean enviar mensajes. Para enviar un mensaje, P_i debe rellenar el siguiente hueco vacío del buffer. P_i recibe los datos del mensaje a enviar como parámetros de salida de la siguiente función, que retorna cuando tiene los datos (suponer que ya está implementada):*

  `datosMensaje (long int * destino, long int * origen, char * sms)`

- *Existe un proceso G_i asociado a cada uno de los módems, que son los responsables del envío de mensajes. Estos procesos recogerán los mensajes pendientes de envío del buffer y se lo mandarán a su correspondiente módem GSM. Cada uno de estos procesos G_i debe abrir el dispositivo "`/dev/modemi`" y, por cada mensaje dado, llamar a la siguiente función bloqueante (suponer que ya está implementada):*

  `enviarMensaje(int descriptorModem, long int destino, long int origen, char * sms)`

  Lógicamente, cada uno de los mensajes del buffer debe ser enviado a través de un único módem (el que se encuentre disponible en cada momento).

Se debe suponer que todos los procesos anteriormente descritos son independientes entre sí y que todos ellos conocen los nombres de los mecanismos de comunicación y sincronización creados por el proceso P.

Se pide:

a) *Como se ve en la Figura, los punteros al primer hueco libre y al siguiente mensaje a enviar están localizados en el objeto de memoria compartido. ¿Sería más eficiente tener almacenados estos valores en cada uno de los procesos?, ¿en cuáles de ellos? ¿Qué problemas plantea tener los punteros en el objeto de memoria compartida?*

b) *Implementar el proceso P siguiendo el esquema de la figura .*

c) *Implementar uno de los procesos P_i siguiendo el esquema de la figura .*

d) *Implementar uno de los procesos G_i siguiendo el esquema de la figura .*

e) Se desea realizar un log de todos los mensajes enviados por los procesos G_i. Para ello, se debe crear el fichero "`/usr/enviados.txt`" y, cada vez que uno de los procesos G_i envía un mensaje, se debe añadir al final de dicho fichero el mensaje enviado y el número de módem utilizado para su envío. Implementar los cambios necesarios en los procesos anteriores para dar soporte a esta nueva funcionalidad.

Solución

a) Los punteros deben ser compartidos por todos los procesos. Es necesario, por tanto, que estén almacenados o bien en el propio objeto de memoria compartida, o bien en algún otro objeto que pueda ser compartido por todos los procesos. En ningún caso deben estar en los propios procesos ya que de esa forma sería muy difícil actualizar los valores de los punteros en todos los procesos.

Tener los punteros en el objeto de memoria compartida exige que los procesos accedan en exclusión mutua al objeto al modificar los punteros. Por último, el objeto de memoria compartida estará en el mapa de memoria de todos los procesos, y en cada proceso en una dirección diferente, por lo que los punteros deben contener valores relativos, no pudiéndose utilizar direccionamiento absoluto.

b) Código del proceso *P*. Los mecanismos de comunicación y sincronización que crea *P* deben ser con nombre, ya que todos procesos de la solución son independientes entre sí. Este proceso no destruye ningún mecanismo ya que no se pide en el enunciado.

```
#define MAX_BUFFER 1000
struct mensaje{
   long int destino; // Número teléfono de destino
   long int origen;  // Número de teléfono de origen
   char sms[160];    // Cuerpo del mensaje
}
struct compartido{
   mensaje buffer[MAX_BUFFER];
   int pHueco; int pElemento;
}
sem_t *elementos;   /* elementos en el buffer */
sem_t *huecos;      /* huecos en el buffer */
sem_t *sembin;  /* acceso a compartido */


int main (void)
{
   int shd;
   struct compartido * objCompartido;

   /* El proceso P crea el fichero para memoria compartida */
   shd = open("BUFFER", O_CREAT, 0700);
   /* Le asigna el tamaño necesario */
   ftruncate(shd, sizeof(struct compartido));

   /* Proyectar el objeto de memoria compartida en su espacio
   de direcciones */
   objCompartido = (struct compartido *) mmap(NULL,

   sizeof(struct compartido), PROT_WRITE, MAP_SHARED, shd, 0);
   /* Inicializa punteros de huecos y elementos */
   objCompartido->pHueco = 0;
   objCompartido->pElemento = 0;
   /* Desproyecta el objeto de memoria compartida */
   munmap (objCompartido, sizeof(struct compartido) );

   elementos = sem_open("ELEMENTOS", O_CREAT, 0700, 0);
   huecos = sem_open("HUECOS", O_CREAT, 0700, MAX_BUFFER);
   sembin = sem_open("SEMBIN", O_CREAT, 0700, 1);
}
```

c) Código del proceso P_i.

```
#define MAX_BUFFER 1000
struct mensaje{
   long int destino; // Número teléfono de destino
   long int origen;  // Número de teléfono de origen
   char sms[160];    // Cuerpo del mensaje
}
struct compartido{
   mensaje buffer[MAX_BUFFER];
   int pHueco;
   int pElemento;
}
sem_t *elementos;   /* elementos en el buffer */
sem_t *huecos;      /* huecos en el buffer */
sem_t *sembin;  /* acceso a compartido */
void Pi (void)
{
   int i, shd;
   struct compartido * objCompartido;
   long int origen, destino;
   char sms[160];
   /* Abre el fichero para memoria compartida */
   shd = open("BUFFER", O_RDONLY);

   /*Proyectar el objeto de memoria compartida en su espacio de direcciones */
   objCompartido = (struct compartido *) mmap(NULL,

   sizeof(struct compartido), PROT_WRITE|PROT_READ, MAP_SHARED, shd, 0);

   /* Abre los semáforos */
   elementos = sem_open("ELEMENTOS", 0);
   huecos = sem_open("HUECOS", 0);
   sembin = sem_open("SEMBIN", 0);

   for(i=0; i < SMS_A_PRODUCIR; i++ ) {
      /* Se obtiene mensaje a enviar */
      datosMensaje(&destino, &origen, sms);
      sem_wait(huecos);    /* Un hueco menos */
      sem_wait(sembin);         /* Acceso con exclusión mutua */
      (objCompartido -> buffer[objCompartido->pHueco]).destino = destino;
      (objCompartido -> buffer[objCompartido->pHueco]).origen = origen;
      strcpy(sms ,(objCompartido -> buffer[objCompartido->pHueco]).sms);
      objCompartido->pHueco = (objCompartido->pHueco + 1) % MAX_BUFFER;
      sem_post(sembin);
      sem_post(elementos);       /* Un elemento mas */
   }

   /* Desproyecta el objeto de memoria compartida */
   munmap (objCompartido, sizeof(struct compartido) );
}
```

d) Código del proceso G_i.

```
#define MAX_BUFFER 1000
struct mensaje{
   long int destino; // Número teléfono de destino
   long int origen;  // Número de teléfono de origen
   char sms[160];    // Cuerpo del mensaje
}
struct compartido{
   mensaje buffer[MAX_BUFFER];
   int pHueco;
   int pElemento;
```

```
}
sem_t *elementos;   /* elementos en el buffer */
sem_t *huecos;      /* huecos en el buffer */
sem_t *sembin;  /* acceso a compartido */

void Gi (void)
{
   int i, shd, dm;
   struct compartido * objCompartido;
   long int origen, destino;
   char sms[160];
   /* Abre el fichero para memoria compartida */
   shd = open("BUFFER", O_RDONLY);

   /*Proyectar el objeto de memoria compartida en su espacio de direcciones */
   objCompartido = (struct compartido *) mmap(NULL,

   sizeof(struct compartido), PROT_READ|PROT_WRITE, MAP_SHARED, shd, 0);

   /* Abre los semáforos */
   elementos = sem_open("ELEMENTOS", 0);
   huecos = sem_open("HUECOS", 0);
   sembin = sem_open("SEMBIN", 0);
   /* Abre el modem con el que está relacionado */
   md = open("/dev/modemi");

   for(i=0; i < SMS_A_CONSUMIR; i++ ) {
      sem_wait(elementos);    /* Un elemento menos */
      sem_wait(sembin);         /* acceso con exclusión mutua */
      /* Lee mensaje a enviar */
      destino = (objCompartido -> buffer[objCompartido->pElemento]).destino;
      origen = (objCompartido -> buffer[objCompartido->pElemento]).origen;
      strcpy(sms ,(objCompartido -> buffer[objCompartido->pElemento]).sms);
      objCompartido->pElemento = (objCompartido->pElemento + 1) % MAX_BUFFER;
      sem_post(sembin);
      sem_post(huecos);       /* Un hueco mas */
      /* Ya liberados los recursos, envía el mensaje */
      enviarMensaje (md, destino, origen, sms);
   }

   /* Desproyecta el objeto de memoria compartida */
   munmap (objCompartido, sizeof(struct compartido) );
}
```

e) El proceso *P* puede ser el encargado de crear el fichero de *log*. Los procesos G_i, cada vez que envíen un mensaje, deben añadir al final del fichero de *log* dicho mensaje. La opción *O_APPEND* de la llamada *open* es suficiente, ya que garantiza que en todo momento estamos escribiendo al final del fichero, aunque éste esté compartido por varios procesos.

Para esta solución se debe añadir:

En el programa P

```
creat("/usr/enviados.txt", 0700);
```

En los programas G_i

```
/* Al principio del programa */
fd = open("/usr/enviados.txt", O_WRONLY|O_APPEND);
/* Justo después de enviar mensaje */
sprintf(fd, "MODEM: %d Mensaje: %s\n", md, sms);
write(fd, buffer, strlen(buffer));
```

Son posibles otras soluciones sin abrir el fichero con el flag *O_APPEND*. En este caso, antes de cada escritura, habría que hacer un *lseek* al final del fichero. En esta solución habría que garantizar que el *lseek* y el *fprintf* se hacen

de forma atómica, ya que en caso contrario se podría meter otro proceso en medio de esas operaciones. Para garantizar esta atomicidad, o bien utilizamos un semáforo adicional con valor inicial 1 que bloquee el acceso al fichero, o bien realizamos las operaciones *enviarMensaje*, *lseek* y *fprintf,* antes de las llamadas *sem_post* que se realizan en G_i (esta solución, aunque factible, implica tener bloqueado el objeto de memoria compartida más tiempo del debido).

Problema 4.11 (mayo 2005)

Sean las siguientes dos utilidades cuyo código se adjunta, respectivamente diseñadas para ser invocadas como:

- *emite segundos texto*
- *filtro mandato [args...]*

emite.c

```
/* 1*/  int main(int argc, char *argv[])
/* 2*/  {
/* 3*/       int secs=atoi(argv[1]);
/* 4*/       while(secs > 0) {
/* 5*/            sleep(4); /*segundos*/
/* 6*/            secs -= 4;
/* 7*/            write(1, argv[2], strlen(argv[2]));
/* 8*/       }
/* 9*/       return 0;
/*10*/  }
```

filtro.c

```
/* 1*/  int main(int argc, char **argv)
/* 2*/  {
/* 3*/       int pp[2], ret, cnt=0, ttl=0;
/* 4*/       char buff[10];
/* 5*/       pipe(pp);
/* 6*/       switch(fork()) {
/* 7*/       case  0:
/* 8*/            close(1);
/* 9*/            dup(pp[1]);
/*10*/            argv++;
/*11*/            execvp(*argv, argv);
/*12*/       case -1:
/*13*/            perror(*argv);
/*14*/            exit(1);
/*15*/       default:
/*16*/            close(0);
/*17*/            dup(pp[0]);
/*18*/            while((ret = read(0, buff, 10)) > 0) {
/*19*/                 cnt++; ttl += ret;  /* Filtrado */
/*20*/                 write(1, buff, ret);
/*21*/            }
/*22*/       }
/*23*/       fprintf(stderr, "%d,%d\n", cnt, ttl);
/*24*/       return 0;
/*25*/  }
```

Responda:

a) ¿Cuál es el objetivo aparente de `filtro`*?*

b) Si se invocase a `filtro inexistente` *indicando un mandato inexistente, ¿qué líneas se ejecutarían tras la invocación a* `execvp` *de la línea 11?*

c) *Desgraciadamente, la implementación dada de* `filtro` *contiene un error que (independientemente de quién esté al otro lado del pipe) impide que concluya la que debería ser última invocación a* `read`*. Indique qué descriptores deberían haberse cerrado,* ***como mínimo****, para corregir el problema.*

Suponiendo corregidas las deficiencias anteriormente indicadas y considerando la ejecución de:
`filtro emite 12 AEIOU`

d) *¿Cuántos segundos,* ***aproximadamente****, tardará en completarse la primera llamada a* `read`*?*

e) *¿Qué dos valores (separados por una coma) se mostrarán finalmente por el* `stderr`*?*

Solución

a) `filtro` ejecuta el mandato indicado como un proceso hijo, conectándose padre e hijo con una tubería. El hijo escribe en la tubería y el padre lee de ella. Luego `filtro` filtrará la salida del mandato.

b) Si la llamada a `exec` falla, y dado que en C los diferentes casos de la sentencia `switch` no son excluyentes, la llamada a `exec` retornará (devolviendo un -1) y el hilo continuará ejecutando las líneas 13 y 14. Esta última línea hace un `exit`, donde el proceso terminará.

c) Para que la última llamada a `read` (que está leyendo de un pipe) concluya y devuelva un cero, deben darse dos circunstancias, que no haya datos en el pipe y que no queden descriptores asociados al extremo de escritura del pipe. Las anteriores llamadas a `read` irán consumiendo los datos que el mandato introduzca en el pipe, pero la última no concluirá mientras existan descriptores asociados al extremo de escritura. Ni el padre ni el hijo cerraron `pp[1]`, no obstante, cuando el hijo concluya se cerrarán automáticamente todos sus descriptores. De manera que lo mínimo que deberíamos hacer para que el programa `filtro` concluya correctamente es que el padre cierre `pp[1]`, descriptor que, desde luego, no va ha usar.

d) El mandato `emite` emite el texto indicado como segundo argumento (“AEIOU”) cada 4 segundos y hasta que se cumplan el número de segundos (12) especificado en su invocación. Por otro lado, el mandato `filtro` esta intentando leer del pipe de 10 en 10 bytes. Hay que recordar que la lectura de un pipe devuelve la cantidad de información disponible en ese preciso instante hasta el máximo especificado en como tercer argumento a `read`. Dicho lo cual, la primera llamada a `read` leerá los 5 caracteres “AEIOU” al cabo de 4 segundos aproximadamente.

e) Para un total de 12 segundos, el mandato `emite` realizará 3 iteraciones, en cada una de las cuales emitirá los 5 caracteres “AEIOU”. Los dos valores `cnt` y `ttl` que filtro muestra finalmente por la salida estándar de error contabilizan respectivamente el número de llamadas a `read` que devolvieron un valor mayor que cero y el número total de caracteres leídos. Luego la información finalmente mostrada será “`3,15`”.

Problema 4.12 (septiembre 2005)

Se desea implementar un nuevo mecanismo de comunicación de procesos denominado tubito con las siguientes características:

- *Con este mecanismo se pueden leer y escribir cadenas de caracteres.*
- *Sólo puede ser utilizado por dos procesos relacionados en la jerarquía.*
- *Es un mecanismo de comunicación unidireccional. El padre puede realizar operaciones de escritura y el hijo de lectura.*
- *Es un mecanismo de comunicación asíncrono.*
- *El tubito tendrá las siguientes operaciones no bloqueantes:*

1. `void *crear_tubito ()`

   Esta operación la invocará el proceso padre, antes de hacer un fork para crear al proceso hijo con el que se desea comunicar, y creará las estructuras necesarias para la implementación de un tubito de tamaño 10 KiB. La operación devuelve un descriptor que deberá ser utilizado en las funciones de lectura y escritura.

2. `int leer_tubito (void *descriptor, int tam, char *datos)`

Esta operación la invocará el proceso hijo para leer tam bytes del tubito. La función devolverá el número de bytes que se han conseguido leer y en datos el resultado de la lectura.

3. `int escribir_tubito (void *descriptor, int tam, char *resul)`

Esta operación la invocará el proceso padre para escribir tam bytes en el tubito. La función devolverá el número de bytes que se han conseguido escribir.

La implementación de los tubitos se realizará de la siguiente manera.

- *La operación* `crear_tubito` *proyectará el fichero* `/dev/zero` *como MAP_SHARED en memoria y devolverá la dirección a partir de la cual se ha proyectado el fichero. Esta dirección será la que se utilice como descriptor del mecanismo de comunicación.*

  La zona proyectada contiene esta estructura de datos:

```
struct tubito{
   int pos_lect; //Posición actual de lectura
   int pos_escr; //Posición actual de escritura
   char bufferCircular[10000]; // buffer con la información
int escritos_pendientes_leer; // num de caracteres pendientes de leer
}
```

- *El manejo del buffer es circular, por la que cuando se llega al tamaño máximo se debe comenzar a leer/escribir desde la posición 0 del buffer.*
- *La operación de escritura rellenará los siguientes caracteres vacíos del buffer e incrementará* `pos_escr`.
- *La operación de lectura leerá del buffer e incrementará el índice* `pos_lect`.
- *La variable* `escritos_pendientes_leer` *llevará la cuenta del número de bytes que ha escrito el padre y que el hijo todavía no ha leído. Lógicamente esta variable nunca podrá ser mayor de 10.000, ya que el buffer estaría lleno, ni más pequeña que 0, situación en la que el buffer estaría vacío. Con esta variable se debe controlar que el hijo sólo lee cosas que ha escrito el padre, y que el padre no escribe más datos de los posibles.*
- *No es necesario controlar quién lee o escribe en el tubito (se supondrá que el padre siempre escribe y que el hijo siempre lee).*

Se pide:

a) Realice un ejemplo sencillo donde se vea cómo el padre y el hijo pueden utilizar el tubito.

b) En caso de que existir la operación `void destruir_tubito (void *descriptor)`*, ¿quién debería invocarla?*

c) ¿Es necesario el uso de algún mecanismo de sincronización para la implementación de los tubitos?

d) Suponiendo que el descriptor es válido, implemente las tres operaciones descritas de los tubitos.

e) Las operaciones de lectura y escritura reciben un descriptor a través del cual son capaces de acceder a la estructura de memoria. ¿Cómo se podría verificar que este descriptor es correcto y que, por tanto, estamos accediendo a una zona de memoria correcta?

f) ¿Podría hacerse bidireccional el mecanismo? Describir brevemente (no más de 10 líneas) cómo podría implementarse.

SOLUCIÓN

a) El programa propuesto es el siguiente.

```
int main(void)
{
   void *descriptor;
   pid_t pid;
   int status;

   char cadena[4]="hola";
   char resul[4];
```

```
   descriptor=crear_tubito();
   pid = fork();

   if (pid!=0)
   {
        escribir_tubito(descriptor,sizeof(cadena),cadena);
        wait(&status);
   }
   else
   {
     leer_tubito(descriptor,sizeof(resul),resul);
     return 0;
   }

   return 0;
}
```

Tal y como está implementado este programa y debido a que es un mecanismo de comunicación no bloqueante, no se puede asegurar que el hijo reciba lo que ha escrito el padre, ya que depende del orden de ejecución de los procesos.

b) Debido a que es un mecanismo basado en una zona de memoria compartida, deberán ser tanto el proceso padre como el hijo los que cierren el mecanismo, a fin de liberar todos los recursos.

c) En principio, al ser un mecanismo de comunicación no bloqueante, no se debe considerar el uso de variables condicionales a la hora de la lectura y escritura en el buffer compartido. Es decir, cuando el buffer esté lleno o vacío, e intentemos escribir o leer respectivamente, la llamada no se bloquea. Simplemente, las funciones `leer_tubito` y `escribir_tubito` devolverán el número de caracteres que han podido leer o escribir.

Por otra parte, es posible que un proceso padre y un proceso hijo estén realizando una operación al mismo tiempo, situación en la cual debemos proteger la variable `escritos_pendientes_leer`, a la que se debe acceder en exclusión mutua. Esta exclusión mutua se podría implementar con semáforos.

e) Se podría utilizar un mecanismo similar al que se usa para saber que un fichero ejecutable es válido, es decir, se podría añadir a la estructura del mecanismo un número mágico, de forma que cada vez que nos manden un descriptor verifiquemos el número mágico para ver que es un descriptor correcto.

f) No hay ningún problema en hacer este mecanismo bidireccional tal y como está definido, siempre y cuando las operaciones de lectura y escritura sobre el buffer se hagan en exclusión mutua. Es decir, no sólo es necesario proteger la variable `escritos_pendientes_leer`, sino que hay que proteger todo el acceso a la estructura compartida en memoria.

Otra posible opción sería tener un sistema de doble buffer, de forma que cada buffer fuera utilizado en una dirección.

d) El programa propuesto es el siguiente.

```
#define TAM_BUFFER 10000
void *crear_tubito()
{
   int fd,

   fd = open("/dev/zero", O_RDWR);
   t = mmap(NULL, (sizeof(int)*3)+10000 , PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, 0);
   close(fd);
   t->pos_lect=0;
   t->pos_escr=0;
   t->escritos_pendientes_leer=0;

   return(t);
}

int leer_tubito(void *descriptor, int tam, char *datos)
{
```

```
   int leidos=0;

   while(descriptor->escritos_pendientes_leer)
   {
      datos[leidos]=descriptor->bufferCircular[descriptor->pos_lect];
      descriptor->pos_lect = (descriptor->pos_lect+1) % TAM_BUFFER;
      leidos++;
      // Inicio exclusion mutua
      descriptor->escritos_pendientes_leer--;
      // Fin exclusion mutua
      if (descriptor->escritos_pendientes_leer==0)
         break;
   }
   return(leidos);
}

int escribir_tubito(void *descriptor, int tam, char *datos)
{
   int escritos=0;

   while(descriptor->escritos_pendientes_leer)
   {
      descriptor->bufferCircular[descriptor->pos_escr]=datos[escritos];
      descriptor->pos_escr = (descriptor->pos_escr+1) % TAM_BUFFER;
      escritos++;
      // Inicio exclusion mutua
      descriptor->escritos_pendientes_leer++;
      // Fin exclusion mutua
      if (descriptor->escritos_pendientes_leer==TAM_BUFFER)
         break;
   }
   return(escritos);
}
```

Problema 4.13 (junio 2006)

En una práctica que se realiza por parejas se pide desarrollar el programa cifrar con las siguientes características:

- *Toma como entrada un fichero que contiene números enteros en binario y lo cifra, dando como salida ese mismo fichero transformado. El nombre del fichero será el primer argumento del programa.*
- *La ejecución de este programa se debe realizar a través de una serie de procesos, cada uno de los cuales se encargará de parte del fichero. El número de procesos vendrá definido por el segundo argumento del programa. La jerarquía de procesos será un proceso padre y tantos procesos hijo como se nos indique.*
- *A fin de mejorar el rendimiento de este programa se ha decidido usar la técnica de proyección de ficheros en memoria.*

Tu compañero de prácticas ha realizado el siguiente código, que según te cuenta, compila y ejecuta correctamente, pero no realiza la labor para la que ha sido diseñado.

```
int func_transform(int entrada)
{ ... } // Código correcto para el cifrado de un entero

void tratar_parte(int *p, int inicio, int n_enteros)
{
   int i;
   p = p + inicio;
   for (i = 0;  i < n_enteros ;i++){
      *p = func_transform(*p);
      p++;
   }
```

```
}

int main(int argc, char **argv)
{
   int fd, n, size_proc, tam, resto, j, size_of_file, num_enteros;
   int *p, *q;

   fd = open(argv[1],O_RDWR);
   n = atoi(argv[2]);   // Conversión de una cadena de caracteres a un entero

   size_of_file = lseek(fd, 0, SEEK_END);
   p = mmap(NULL, size_of_file, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);

   num_enteros = size_of_file/sizeof(int);
   size_proc = num_enteros / n;
   resto = num_enteros % n;

   // Punto A

   for (j = 0; j < n; j++){
      if (fork() == 0){
         tam = size_proc;
         if (j == (n-1))
            tam = tam + resto;
         tratar_parte(p, j * size_proc, tam);
      }
   }

   while (n > 0){
      wait(NULL);
      n --;
   }
}
```

Se pide:

a) La jerarquía de procesos creada por este código, ¿es la especificada en el enunciado de la práctica? Razona la respuesta.

b) Centrándose en las llamadas al sistema, ¿encuentras algún fallo en este código?

c) Para hacer el código más seguro y profesional, ¿qué recomendaciones harías a tu compañero?

d) En la segunda parte de la práctica nos piden una pequeña mejora. La mejora consiste en añadir al final del fichero un número mágico predefinido, el 999. Para ello, nuestro compañero nos propone añadir las siguientes líneas en el Punto A:

```
// Añadido en Punto A
q = p + num_enteros;
*q = 999;
```

¿Esta modificación funcionaría correctamente? Razona la respuesta.

e) Suponiendo que el número mágico ya funciona correctamente, nos piden una nueva mejora del programa. Esta vez, cuando un proceso hijo finalice, éste sumará uno al valor del número mágico. Para realizar esta labor nuestro compañero nos propone el siguiente código:

```
/* Suponiendo que el puntero q apunta correctamente al último entero del
fichero proyectado */
for (j = 0; j < n; j++){
   if (fork() == 0) {
      ...
      tratar_parte(p, j * size_proc, tam);
      *q = *q + 1;
      ...
   }
}
```

¿Está consiguiendo nuestro compañero realizar la labor correctamente? En caso de existir algún problema, describe brevemente cómo lo solucionarías.

f) Ya está todo funcionando, cuando nuestro compañero decide hacer más potente la función func_transform, con tanta torpeza que la función se llama a sí misma infinitamente de forma recursiva:

```
int func_transform(int entrada)
{
   char buffer[50000];
   ...
   return (func_transform(aux));
}
```

Suponiendo que el programa tiene tiempo ilimitado para ejecutar, ¿qué error se produciría en el sistema?

g) Tu compañero te pregunta qué cambios serían necesarios en el código para que, en vez de estar basado en procesos, estuviera basado en hilos. ¿Qué cambios le explicarías a tu compañero?, ¿qué llamadas al sistema necesitarías para su implementación? No hace falta codificar esos cambios, sólo indicar qué debería hacerse.

SOLUCIÓN

a) La jerarquía de procesos creada por nuestro compañero no es la correcta. La jerarquía especificada en el enunciado consiste en un proceso padre con **n** procesos hijos. Sin embargo, en el código de nuestro compañero los procesos hijos, después de llamar a la función `tratar_parte`, no finalizan y se vuelven a meter en el bucle for, creando a su vez otros procesos. La solución es tan sencilla como incluir un `exit` después de la llamada a la función `tratar_parte`.

La solución del hacer un `break` en vez de un `exit` no es la más correcta, ya que el proceso hijo seguiría ejecutando y realizaría la llamada wait, cuando realmente no tiene ningún hijo por el que esperar. En este caso la llama wait nos devuelve un error indicando que no tenemos procesos hijo.

b) La llamada mmap no se está haciendo correctamente, ya que el fichero se está proyectando como `MAP_PRIVATE` cuando debería proyectarse como `MAP_SHARED` para que los cambios realizados se graben a fichero cuando se desproyecta. El resto de llamadas al sistema están perfectamente definidas, incluyendo el `lseek`, que además de posicionar el puntero en la última posición del fichero (ya que se especifica `SEEK_END`) nos devuelve dicha posición.

c) Respecto a la seguridad y profesionalidad del código de nuestro compañero se le podrían hacer dos recomendaciones: (i) es muy importante realizar una correcta verificación de los posibles errores de las llamadas al sistema, y (ii) es muy recomendable liberar todos los recursos asignados. Continuando con este último punto se debería cerrar el fichero abierto una vez proyectado –llamada `close(fd)`– y el proceso padre debería cerrar la proyección del fichero al final del programa –llamada `munmap(p, size_of_file)`–.

d) La modificación de nuestro compañero no funciona correctamente. Si bien la aritmética de punteros es correcta, es decir, `q = p + num_enteros` nos posiciona al final del fichero proyectado, al realizar la sentencia `*q = 999` estamos escribiendo fuera de la proyección. De esta forma, no estamos consiguiendo lo que se nos indica en el enunciado. La solución pasa por cambiar el tamaño del fichero antes su proyección, por ejemplo con la llamada `ftruncate(fd, size_of_file + sizeof(int))`.

e) Suponiendo que el puntero q apunta correctamente al último entero del fichero, la sentencia `*q = *q + 1` funciona correctamente. Sin embargo, tenemos que tener en cuenta que se podrían producir condiciones de carrera al intentar actualizar el valor del número mágico. La solución es sencilla, nos bastaría con utilizar un mecanismo de sincronización como los semáforos.

f) Cada vez que se realiza una llamada a una función en la pila del proceso se crea su registro de activación. Por tanto, si hacemos llamadas recursivas de manera infinita a una función, estaremos haciendo crecer la región de pila del mapa de memoria del proceso de manera infinita. Las llamadas se podrían seguir realizando mientras la pila pueda seguir creciendo.

g) Comentaría a mi compañero que en este caso cambiar nuestro programa para que use hilos es muy sencillo. En primer lugar debemos establecer el estado de terminación de los hilos. En nuestro caso, es conveniente usar hilos de tipo `JOINABLE`, para que el hilo principal pueda saber cuando finalizan los hilos trabajadores. El resto de los cambios es casi inmediato. Debemos crear la estructura para los atributos de los hilos (`pthread_attr_t`) y las variables para almacenar los identificadores de los hilos (`pthread_t`). A continuación, debemos inicializar los hilos

(`pthread_attr_init`). De esta forma, ya estamos preparados para crear los hilos, sustituyendo la llamada `fork` por la llamada `pthread_create` en donde debemos especificar que cada hilo debe ejecutar la función `tratar_parte`. Por último, se debe sustituir la llamada wait por la llamada `pthread_join` para esperar la finalización de los hilos y destruir el objeto de tipo atributo previamente creado con la llamada `pthread_attr_destroy`. Para no dejarnos ningún detalle, la función `tratar_parte` debe finalizar con la llamada `pthread_exit` para que los hilos trabajadores finalicen correctamente.

Problema 4.14 (junio 2007)

*Se tiene un fichero de 1000 bytes sobre el que interactúan procesos lectores y escritores cuyo código se adjunta. Se observará que cada uno de ellos lee o escribe 8 bytes, empezando en argv[1]*8. Considere que los lectores y escritores nunca son llamados con argv[1] > 124.*

a) *¿Cuál sería el tamaño del fichero después de realizar 20 operaciones de lectura (lector B) seguido de 5 escrituras (escritor A)?*

b) *Suponiendo que existe un máximo de 25.000 posibles lectores y que tenemos un escritor que está modificando «val», ¿Cuantos lectores que estén leyendo «val» se podría llegar tener?*

c) *Partiendo del fichero inicial, indicar el máximo número de escritores A simultáneos (todos ellos modificando «val») que se podrían tener, en caso de existir tres lectores B que han recibido argv[1] = 0, argv[1] = 40 y argv[1] = 120 y que están leyendo «val».*

Código escritor A

```
int main(int argc, char *argv[]) {
   int fd, val, argv1;
   struct flock fl;
   argv1 = atoi(argv[1])*8
   fd = open("BD", O_RDWR);
   fl.l_whence = SEEK_SET;
   fl.l_start = argv1;
   fl.l_len = 8;
   fl.l_pid = getpid();
   fl.l_type = F_WRLCK;
   fcntl(fd, F_SETLKW, &fl);
   lseek(fd, argv1, SEEK_SET);
   read(fd, &val, 8);
   <<se modifica val>>
   lseek(fd, argv1, SEEK_SET);
   write(fd, &val, 8);
   fl.l_type = F_UNLCK;
   fcntl(fd, F_SETLK, &fl);
   return 0;
}
```

Código lector B

```
int main(int argc, char *argv[]) {
   int fd, val, argv1;
   struct flock fl;
   argv1 = atoi(argv[1])*8
   fd = open("BD", O_RDONLY);
   fl.l_whence = SEEK_SET;
   fl.l_start = argv1;
   fl.l_len = 8;
   fl.l_pid = getpid();
   fl.l_type = F_RDLCK;
   fcntl(fd, F_SETLKW, &fl);
   lseek(fd, argv1, SEEK_SET);
   read(fd, &val, 8);
   <<se manipula val>>
   printf("%d\n", val);
   fl.l_type = F_UNLCK;
   fcntl(fd, F_SETLK, &fl);
   return 0;
}
```

Solución

Primero observemos que el fichero está compuesto por 1000/8 = 125 datos que pueden ser utilizados de forma independiente. Cada lector establece un cerrojo de tipo compartido sobre un dato de 8 bytes y cada escritor establece un cerrojo exclusivo también sobre un dato de 8 bytes.

a) Dado que las operaciones de escritura siempre se realizan dentro del fichero su tamaño no se modifica, por lo que seguirá siendo de 1000 bytes.

b) El escritor tendrá establecido un cerrojo exclusivo sobre el dato que esté escribiendo (8 bytes), en el resto del fichero podemos tener tantos lectores como se quiera. Nos dicen que el hay un máximo de 25.000 lectores, por lo que se podría llegar a tener esos 25.000 lectores, cada uno de los cuales estará leyendo uno de los 124 datos sobre los que no ha cerrojo exclusivo. Nótese que todos ellos podrían llegar a estar leyendo el mismo dato.

c) Al existir tres lectores sobre datos distintos, existen tres cerrojos de tipo compartido sobre los que no se puede hacer cerrojo exclusivo. Se pueden establecer cerrojos exclusivo (necesarios para los escritores) sobre 125 - 3 = 122 datos, por tanto, se pueden tener hasta 122 escritores.

Problema 4.15 (junio 2007)

Sea el siguiente código, que compila correctamente.

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>

int *p;
int tamThread=0;

void escribir_pares(void)
{
   int i, dato=0;

   for (i=0; i < tamThread; i++ ) {
      *p=dato;
      dato=dato+2;
      p++;
      }
}

void escribir_impares(void)
{
   int i, dato=1;

   for (i=0; i < tamThread; i++ ) {
      *p=dato;
      dato=dato+2;
      p++;
   }
}

int main(void)
{
   int fd, tam;
   pthread_attr_t attr;
   pthread_t th1, th2;

   fd=open("numbers.dat", O_RDWR);
   // Obtención del tamaño del fichero
   tam = ...;
   p=mmap(NULL, tam, PROT_WRITE, MAP_SHARED, fd, 0);
   close(fd);
   tamThread = tam/sizeof(int)/2;
   pthread_attr_init(&attr);
   pthread_create(&th1, &attr, escribir_pares, NULL);
   pthread_create(&th2, &attr, escribir_impares, NULL);
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);
   pthread_attr_destroy (&attr);

   munmap(p,tam);
   return(0);
```

```
}
```

Suponer que el fichero `numbers.dat` tiene un tamaño de 56 bytes y que está localizado en el mismo directorio que el programa ejecutable. Además, se debe suponer que un número entero ocupa 4 bytes.

Se pide:

a) Codificar dos formas diferentes de obtener el tamaño del fichero `numbers.dat` antes de su proyección.

b) Enumera los segmentos que tendrá el mapa de memoria del proceso justo antes de la primera llamada a `pthread_join` teniendo en cuenta que el programa hace uso de bibliotecas estáticas. Indicar, para cada uno de los segmentos identificados, su protección correspondiente (RWX).

c) ¿Qué valor devolverá la llamada al sistema `munmap`? En caso de que la llamada devuelva un código de error, realizar las correcciones necesarias al código para solucionarlo.

d) Dado que `p` es una variable compartida, ¿qué estrategia de uso es necesaria para que el resultado sea el esperado?

e) Tras una ejecución del programa, se verifica que en el fichero `numbers.dat` se quedan almacenados los siguientes números enteros:

`0 2 4 6 8 10 12 1 3 5 7 9 11 13`

Sin embargo, se desea que el programa escriba el siguiente contenido:

`0 1 2 3 4 5 6 7 8 9 10 11 12 13`

*Introducir los cambios necesarios en el programa de forma que los threads `escribir_pares` y `escribir_impares` **se turnen** en el acceso al fichero proyectado. Plantear una solución basada en alguna técnica de **sincronización**.*

La solución codificada, ¿corrige también el problema del apartado d)?

SOLUCIÓN

a) Solución con fstat:

```
struct stat bstat;
fstat(fd, &bstat);
tam =  bstat.st_size;
```

Solución con `lseek`:

```
tam = lseek(fd, 0, SEEK_END);
```

b) El mapa de memoria del proceso tendrá los siguientes segmentos:

- Código (RX)
- Datos con valor inicial (RW)
- Datos sin valor inicial (RW)
- Pila (RW)
- Fichero proyectado `numbers.dat` (W)
- Pila del thread `escribir_pares` (RW)
- Pila del thread `escribir_impares` (RW)

c) La llamada al sistema munmap fallará, devolviendo -1, ya que estamos intentando desproyectar desde la dirección p, que los threads escribir_pares y escribir_impares están modificando en sus respectivas sentencias p++. La solución es muy sencilla, se puede realizar una copia del puntero p justo después de la proyección (q=p), y realizar el munmap usando esta nueva variable (munmap(q,tam)).

d) Los threads escribir_pares y escribir_impares están accediendo a la misma variable compartida sin ningún tipo de control, por lo que se pueden producir problemas de carrera. Para evitar este problema, se deben convertir las zonas conflictivas en zonas de exclusión mutua, con alguna de las técnicas de sincronización vistas: semáforos o mutex y condiciones.

e) En este caso se necesita alternar entre los threads escribir_pares y escribir_impares. Una posible solución sería con semáforos.

Declarar dos variables globales (los semáforos)

```
sem_t pares;
sem_t impares;
```

Estos semáforos deben ser inicializados en el thread principal

```
sem_init(&pares,0,1);
sem_init(&impares,0,0);
```

El thread escribir_pares quedaría

```
for(...) {
   sem_wait(&pares);
   *p=dato;
   dato=dato+2;
   p++;
   sem_post(&impares);
}
```

El thread escribir_impares quedaría

```
for(...) {
   sem_wait(&impares);
   *p=dato;
   dato=dato+2;
   p++;
   sem_post(&pares);
}
```

Justo antes de finalizar la ejecución debemos liberar los recursos:

```
sem_destroy(&par);
sem_destroy(&impar);
```

Por último, el acceso al dato compartido p quedaría protegido por los semáforos, por lo que ya no podría darse el problema de carrera que se analizó en el apartado d).

Se podría plantear otra solución con mutex y variables condicionales.

Declaración de variables globales

```
int par=0;
pthread_mutex_t mutex;
pthread_cond_t toca_par, toca_impar;
```

Inicializaciones en el thread principal

```
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&toca_par, NULL);
pthread_cond_init(&toca_impar, NULL);
```

El thread escribir_pares quedaría

```
for(...) {
   pthread_mutex_lock(&mutex);
   while (par==0){
      pthread_cond_wait(&toca_par,&mutex);
   }
   *p=dato;
   dato=dato+2;
   p++;
   par = 0;
   pthread_cond_signal(&toca_impar);
   pthread_mutex_unlock(&m);
}
```

El thread escribir_impares quedaría

```
for(...) {
```

```
   pthread_mutex_lock(&mutex);
   while (par==1){
      pthread_cond_wait(&toca_impar,&mutex);
   }
   *p=dato;
   dato=dato+2;
   p++;
   par = 1;
   pthread_cond_signal(&toca_par);
   pthread_mutex_unlock(&m);
}
```

Por último, debemos liberar los recursos

```
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&toca_par);
pthread_cond_destroy(&toca_impar);
```

Problema 4.16 (junio 2010)

Parte A. *Sea el programa siguiente:*

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
   int dp[2];

   pipe(dp);
   if (fork()==0) {
      int yo, mipadre;

      fork();
      close(dp[0]);
      yo = getpid();
      mipadre = getppid();
      write(dp[1], &yo, sizeof(int));
      write(dp[1], &mipadre, sizeof(int));
   }
   else {
      int dato;
      if (fork()==0)
         if (fork()==0)
            fork();
      read(dp[0], &dato, sizeof(int));
      /* .Se realiza el procesamiento del dato......*/
   }

   return 0;
}
```

1.- ¿Existe alguna circunstancia por la cual alguno de los procesos generados por dicho código quedase bloqueado de forma indefinida? En caso afirmativo indique dicha circunstancia y plantee una solución.

2.- Suponiendo que el programa ejecuta sin que se produzca ningún error, indicar, justificadamente, el número máximo de lectores simultáneos que puede llegar a tener el pipe.

3.- Se desea que los procesos lectores ejecuten en un determinado orden. Explicar qué solución utilizaría para alcanzar dicho objetivo.

Parte B. *Sea, ahora, un sistema de venta en el que coexisten distintos compradores, implementados como threads, que siguen la secuencia siguiente:*

- *Se selecciona el producto a comprar*
- *Se analiza la disponibilidad del mismo así como su precio*
- *Se determina el número de unidades a comprar (función "seleccion_usuario")*
- *Se realiza la compra.*

El precio del producto varía con el número de ellos en stock, de acuerdo a la siguiente fórmula: Precio = a + b·(c – n), siendo a, b y c constantes que dependen del producto y n el número de productos que quedan en stock. Los valores a, b, c y n de cada producto se almacenan en un registro en memoria.

Se plantea el siguiente esquema para resolver dicho problema:

```
#include <pthread.h>

struct registro {        /* Registro de producto */
   pthread_mutex_t cerrojo;
   int n;
   float a, b, c;
};
/* Una vez seleccionado el producto en una fase anterior, se ejecuta la compra */
void compra(struct registro * r) {
   int disponibles;
   float precio;
   int p;

   pthread_mutex_lock(&(r->cerrojo));
   disponibles = r->n;
   precio = r->a + r->b * (r->c - r->n);
   pthread_mutex_unlock(&(r->cerrojo));

   muestra_usuario(disponibles, precio);
/* El usuario selecciona el número p de unidades del producto a comprar, siendo p ≤
disponible */
   if ((p = seleccion_usuario(disponibles)) > 0) {
      pthread_mutex_lock(&(r->cerrojo));
      precio = r->a + r->b * (r->c - r->n);
      r->n -= p;
      pthread_mutex_unlock(&(r->cerrojo));
      realizar_cargo(precio * p);
   }
}
```

4.- ¿Cuántos compradores simultáneos podríamos llegar a tener un producto del que quedan n unidades?

5.- ¿Es posible que el precio aplicado sea distinto del precio mostrado? En caso afirmativo explique la secuencia que produciría dicha situación.

6.- ¿Es posible llegar a vender más unidades de las disponibles? En caso afirmativo desarrolle una solución para evitarlo.

SOLUCIÓN

1.- Si alguno de los procesos escritores no se crea o muere antes de escribir, hay lectores que se quedarían esperando eternamente. La solución consiste en cerrar en los lectores el descriptor de escritura del pipe. Por ejemplo, poniendo close(dp[1]); después del else, lo que haría que se un escritor no se crea o muere antes de escribir, el read se completaría devolviendo 0 bytes.

2.- Se crean cuatro lectores. Todos ellos pueden quedar bloqueados en el read, si ninguno de los dos escritores ha escrito nada.

3.- Hay muchas formas de conseguir que los lectores ejecuten en un cierto orden. Si queremos que ejecute primero el padre luego el primer hijo y así sucesivamente, pondríamos el read y el tratamiento de los datos antes de cada fork:

```
   else {
```

```
    int dato;
    read(dp[0], &dato, sizeof(int));
    /* .Se realiza el procesamiento del dato......*/
    if (fork()==0)
       read(dp[0], &dato, sizeof(int));
       /* .Se realiza el procesamiento del dato......*/
       if (fork()==0)
          read(dp[0], &dato, sizeof(int));
          /* .Se realiza el procesamiento del dato......*/
          if (fork()==0)
             read(dp[0], &dato, sizeof(int));
             /* .Se realiza el procesamiento del dato......*/
}
```

Si queremos que ejecute primero el último hijo, podemos poner un waitpid en cada proceso para que espere a que termine su hijo.

Finalmente, también podríamos utilizar tres semáforos (no mutex, puesto que se trata de procesos). Dichos semáforos deben ser creados con valor 0 por el padre. Tres procesos lectores quedan esperando cada uno en un semáforo. Al finalizar el lector sin semáforo debe hacer un post del semáforo del lector siguiente, y así sucesivamente.

4.- Puede haber un número ilimitado de lectores que han obtenido el precio y disponibles, pero que no han terminado la compra.

5.- Dado que una vez obtenido el precio se libera el mutex, otros compradores pueden modificar el número de unidades disponibles y, por tanto, el precio que se ha de aplicar.

6.- Efectivamente, se pueden vender más unidades de las disponibles, puesto que el límite que puede solicitar un comprador viene en función del valor de la variable disponible, que puede estar obsoleta si otro comprador ha realizado la compra mientras tanto.

No es buena solución englobar todo el proceso en el mutex puesto que se produciría una gran contención. Solamente podría haber un comprador en cada instante.

Tampoco es buena solución meter el `if ((p = seleccion_usuario(disponibles)) > 0)` dentro de la zona protegida por el mutex. También se produciría gran contención. Por ejemplo, si un cliente no termina de completar la seleccion_usuario el sistema de venta se quedaría bloqueado.

Lo más sencillo es justo antes de la sentencia `r->n -= p;` añadir `if (r->n < p) {p = r->n; }`

Es de destacar que la solución general propuesta en este problema no es nada satisfactoria para el cliente, puesto que se le puede aplicar un precio superior al mostrado inicialmente y se le pueden vender menos unidades de las pedidas por el cliente.

Problema 4.17 (junio 2010)

Sea un encaminador de red (switch) que puede tener n puertos de entrada y m de salida. Por sencillez consideraremos que cada trama contiene solamente un paquete.

Cada puerto está manejado por un thread, por lo que existen n+m threads, creados por el mismo proceso. Los threads de entrada repetidamente reciben una trama, obtienen el paquete, determinan su puerto de salida y lo depositan en memoria. Los threads de salida repetidamente van tomando cada paquete, forman la trama y la envían por su puerto.

Cada thread de entrada tiene un buffer capaz de almacenar una trama y cada thread de salida tiene un buffer capaz de almacenar un paquete. Cada thread de entrada guarda la trama en su buffer y copia, cuando pueda, el paquete en el correspondiente buffer del thread de salida.

a) Determinar la(s) estructura(s) de información a utilizar.

b) Codificar las regiones críticas de los threads de entrada y salida.

SOLUCIÓN

a) Consideraciones previas:

- Cada buffer de entrada no tiene necesidad de región crítica (sólo tiene 1 escritor), pero la escritura requiere esperar a que el buffer esté vacío
- Cada buffer de salida tiene 1 único lector, pero puede tener n escritores. Sólo puede escribir 1 escritor en cada momento, y con acceso exclusivo (ni lector ni otros escritores). Dado que la capacidad de cada buffer es 1 único elemento (paquete), solo tiene 2 estados posibles, vacío o lleno. El estado vacío permite escribir al primer escritor que esté esperando, y transitar a lleno. El estado lleno permite leer al thread lector, y transitar de nuevo a vacío.

Se necesitan:

- n buffers de entrada (locales a su thread) y m buffers de salida (globales)
- m mutex para gestionar el acceso exclusivo en escritura a los buffers de salida, así como diversos vectores de m flags para control de sincronización.

Variables globales:

```
<<tipo_puerto_entrada>> puerto_e[n];
<<tipo_puerto_salida>> puerto_s[m];
<<tipo_paquete>> buffer_salida[m]; /* m buffers de salida */
int buffer_lleno[m]; /* (booleano 0,1) 1 = buffer lleno */
int escribiendo[m]; /* (booleano 0,1) 1 = escribiendo este buffer */
pthread_mutex_t mutex[m]; /* Control de acceso */
pthread_cond_t a_leer[m], a_escribir[m]; /* Condiciones de espera */
```

b)

```
void *Thread_de_Entrada(void *p)
/* RECIBE TRAMA POR PUERTO #i Y ESCRIBE SU PAQUETE EN BUFFER DE SALIDA #j*/
{
   int i = (int)p; /*nº puerto de entrada*/
   int j; /*nº puerto salida */
   while(1){
      buftrama = recibe_trama(puerto_e[i]);
      j = puerto_de_salida(buftrama);

      /* 1) ESPERA CONDICIÓN "buffer #j vacío AND no escribiendo buffer #j" */
      pthread_mutex_lock(&mutex[j]);
      while(buffer_lleno[j] || escribiendo[j])
         pthread_cond_wait(&a_escribir[j], &mutex[j]);
      escribiendo[j] = 1;
      pthread_mutex_unlock(&mutex[j]);

      /* 2) ESCRITURA EN EXCLUSIVA EN EL BUFFER DE SALIDA #j*/
      buffer_salida[j] = paquete(buftrama);

      /* 3) LIBERA EL RECURSO (BUFFER #j) */
      pthread_mutex_lock(&mutex[j]);
      /* Avisa que puede leer el lector #j*/
      buffer_lleno[j] = 1;
      escribiendo[j] = 0;
      pthread_cond_signal(&a_leer[j]);
      pthread_mutex_unlock(&mutex[j]);
   }
}

void *Thread_de_salida(void *p)
/* LEE PAQUETE DEL BUFFER DE SALIDA #j Y ENVÍA TRAMA POR PUERTO #j*/
{
   int j = (int)p; /*nº puerto de salida*/
   while(1){
      /* 1) ESPERA CONDICIÓN "buffer #j lleno" */
      pthread_mutex_lock(&mutex[j]);
      while(!buffer_lleno[j]) //condición espera
         pthread_cond_wait(&a_leer[j], &mutex[j]);
      pthread_mutex_unlock(&mutex[j]);
```

```
      /* 2) LECTURA EN EXCLUSIVA DEL BUFFER DE SALIDA #j Y ENVÍO DE TRAMA*/
      envia_trama(puerto[j], buffer_salida[j]);

      /* 3) LIBERA EL RECURSO (BUFFER #j) */
      pthread_mutex_lock(&mutex[j]);
      /* Avisa que puede escribir cualquier escritor en #j*/
      buffer_lleno[j] = 0;
      pthread_cond_signal(&a_escribir[j]);
      pthread_mutex_unlock(&mutex[j]);
   }
}
```

NOTA. Funciones auxiliares (pseudocódigo):

- `recibe_trama(puerto[i])`: Lee el puerto i
- `envia_trama(puerto[j], paquete)`: Compone y envía trama por el puerto j
- `puerto_de_salida(buftrama)`: Extrae el puerto de salida de una trama
- `paquete(buftrama)`: Extrae el paquete de una trama

Problema 4.18 (enero 2011)

Un sistema de ficheros de tipo UNIX, (diseñado para un pequeño sistema portátil) presenta las siguientes características:

- *Representación de ficheros mediante nodos-i con 12 direcciones directas a bloque, un indirecto simple y un indirecto doble. Direcciones de bloque de 4 bytes.*
- *El tamaño del bloque del sistema de ficheros es de 2 KiB y emplea una cache de 1 MiB con una política de reemplazo LRU.*

Sobre este sistema se ejecutan un proceso Escritor *y cuatro procesos* Lectores *con los fragmentos de código indicados más adelante.*

Escritor

```
#define DATA_SIZE     1024
#define N             2048
char buffer[DATA_SIZE];
int fd, i;
struct flock fl;
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = DATA_SIZE;
fl.l_pid = getpid();
fl.l_type = F_WRLCK;
fd = open("fich", O_CREAT | O_WRONLY,
0666);
ftruncate(fd, DATA_SIZE*N);
fcntl(fd, F_SETLKW, &fl);

for (i = 0 ; i < N; i++) {
   write(fd, buffer, DATA_SIZE);
   if (i < N-1) {
      fl.l_type = F_WRLCK;
      fl.l_start = DATA_SIZE*(i+1);
      fcntl(fd, F_SETLKW, &fl);
   }
   fl.l_type = F_UNLCK;
   fl.l_start = DATA_SIZE*i;
   fcntl(fd, F_SETLK, &fl);
```

Lector

```
#define DATA_SIZE     1024
#define N             2048
char buffer[DATA_SIZE];
int fd, i;
struct flock fl;
fl.l_whence = SEEK_SET;
fl.l_len = DATA_SIZE;
fl.l_pid = getpid();
fd = open("fich", O_RDONLY);

for (i = 0 ; i < N; i++) {
   fl.l_start = DATA_SIZE*i;
   fl.l_type = F_RDLCK;
   fcntl(fd, F_SETLKW, &fl);
   read(fd, buffer, DATA_SIZE);
   fl.l_type = F_UNLCK;
   fcntl(fd, F_SETLK, &fl);
}
close(fd);
```

```
}
close(fd);
```

a) Calcule el tamaño máximo que puede tener un fichero en dicho sistema.

Teniendo en cuenta que la cache se encuentra inicialmente vacía, que no se realiza ninguna operación de limpieza de cache (`sync`) durante la ejecución del mencionado programa, que el escritor ejecuta su primer `fcntl` antes que ningún código lector *ejecute el `open` y que el fichero `"fich"` existía previamente con un tamaño de 16 MiB, se pide:*

b) ¿Puede un lector adelantar al escritor, es decir, leer una zona todavía no escrita por el escritor? Justifique la respuesta.

c) ¿Cuántos accesos a disco se producen durante la ejecución de los programas en cada uno los bucles de lectura y escritura utilizando una política de actualización write-through (escritura inmediata)? Justifique la respuesta. ¿El resultado sería distinto si el fichero `"fich"` no existiese previamente?

d) Repetir la pregunta anterior para la política de actualización write-back (escritura diferida).

e) Calcule la mejor y la peor tasa de aciertos en la cache que se puede producir durante un bucle de lectura, para los dos casos siguientes: cache de 1 MiB y cache de 16 MiB.

f) Seleccionar el mecanismo de sincronización adecuado y plantear el código necesario para garantizar que el código escritor *ejecuta su primer `fcntl` antes que ningún código* lector *ejecute el `open`.*

SOLUCIÓN

a) Cada bloque permite 2 KiB / 4 B = 512 direcciones.

nodo_i almacena 12 direcciones
Indirecto simple almacena 512 direcciones = 0,5 K direcciones
Indirecto doble almacena 512*512 direcciones = 256 K direcciones

Tamaño máximo del fichero = 2 KiB·(12 + 0,5 K + 256 K) = 513 MiB + 24 KiB

b) El Escritor da al fichero un tamaño de DATA_SIZE*N y establece un cerrojo de tamaño DATA_SIZE, antes de que cualquier Lector pueda establecer su cerrojo. Seguidamente escribe en el bloque y amplía el cerrojo al siguiente bloque de tamaño DATA_SIZE. Finalmente, libera el cerrojo del bloque escrito. Por tanto, los Lectores no pueden adelantar al Escritor.

c) Al truncar el fichero se le da un tamaño de 2 MiB, lo que supone 1 K bloque. El Escritor escribe de forma secuencial todo el fichero en bloques de 1K.

Analizaremos primero el bucle del escritor.

- Accesos debidos a los datos: Dada la política write-through, cada escritura produce un acceso al disco, por tanto, se producen 2.048 accesos a disco. Además, como el fichero tiene inicialmente 8 bloques las escrituras 0, 2, 4, 6, 8, 10, 12 y 14 procen una lectura del bloque afectado, para su modificación.Por tanto, hay 1.056 accesos.
- Accesos debidos a metadatos:
  - Mapa nodos_i: Cada bloque que se va asignado supone una escritura en el mapa de nodos_i. Como se acceden a 1024 bloques, pero el fichero ya tenía 8 bloques, se realizan 1.016 accesos para modificar el mapa de bits.
  - Direcciones de bloques. Los 12 primeros están el nodo_i. Se modifica el nodo_i por cada nuevo bloque asignado, es decir, hay 12-8 = 4 escrituras. La dirección de los siguientes 512 bloques se almacena en el indirecto simple, por tanto, son 513 accesos: uno para incluir en el nodo_i la dirección de indirecto simple y 512 para ir inclyendo en dicho bloque indirecto los nuevos bloques de datos asignados. Las direcciones de los 1024 – 512 -8 = 504 bloques restantes implican un bloque indirecto doble, por tanto, hay que modificar su dirección en el nodo_1, el indirecto doble, para el nuevo bloque de direcciones y este último 504 veces para los nuevos 504 nuevos bloque de datos, es decir, 506 acceso
    El total es de 4 + 513 + 506 = 1.023 accesos.

  Por tanto, el total de accesos debidos a los metadatos es de: 1.016 + 1.023 = 2.029.

En relación con los accesos debidos a los bucles lectores hay que hacer notar que se dispone de una cache que permite almacenar aproximadamente la mitad del fichero (habría que descontar los tres bloques de direcciones, así como los bloques del mapa de bits afectados y el del nodos_i, que teambién residirán en memoria). Esto significa,

que, a menos que se retrasase mucho en su ejecución algún lector con respecto al escritor, los lecotres siempre encontrarán en memoria tanto los blques de datos como la metainformación necesaria, no generando accesos al disco.

d) Si la escritura es write-back, las escrituras de datos no se trasladan al disco hasta que los bloques sean expulsados de la cache (sin embargo, se leerían los 8 primeros bloques que tiene el fichero inicialmente). Como las escrituras afectan a 1024 bloques de datos, se necesitan 1024 accesos a disco. Es de notar que hasta que no se ha llegado aproximadamente a la miltad del fichero no se producen escrituras de datos al disco, puesto que se dispone de suficiente memoria cahe. A partir de ese momento se irán expuldando bloques, produciéndose los consiguientes accesos a disco. Más adelante, cuando se produzca un sync se volcarán las demás.

Normalmente la metainformación utiliza, por seguridad, una técnica write-through, por lo que se aplica lo visto en la sección anterior. En caso de aplicar write-back a la metainformación los accesos serían uno por le nodo_i, otro por cada uno de los tres bloques dedicados a direcciones y el o los bloques del mapa debits afectado. Se observa que es mucho menor que en el caso anterior, pero mucho menos fiable frente a una caída del sistema.

e) Con una cache de 16 MiB cabe toda la información del fichero y de su metainformación en la misma, por lo que la tasa de aciertos en el bucle de lectura es del 100%.

Con una cache de 1 MiB, como se ha indicado anteriormente es muy probable que la tasa de aciertos también sea del 100%. Sin embargo, si un lector se retrasa mucho frente al escritor y a los otros lectores, puede encontrar en la cache la última mitad del fichero, por lo que irá sustituyendo bloques continuamente, dando una tasa de aciertos del 0%. En relación con la metainformación la tasa de aciertos será sin embargo del 100% o muy próxima al 100%, puesto que el bloque indirecto simple puede haber sido expulsado.

f) El mecanismo a utilizar depende que lectores y escritores ejecuten como procesos o como threads. En el primer caso utilizaremos semáforos y el en el segundo mutex y condiciones. Una solución para el caso primero sería la siguiente:

Cada proceso escritor o lector crea un semáforo con el contador a cero:

```
sem_t * inicio;
inicio = sem_open("INIT", O_CREAT, 0700, 0);
```

Realmente, el único que crea el semáforo es el primero que ejecuta el sem_open. Los demas simplemente lo abren, con los valores que tenga.

El proceso Escritor incrementa el contador en una unidad justo después de su primer fcntl

```
....
fcntl(fd, F_SETLKW, &fl);
sem_post(inicio);    //el proceso Escritor abre el semáforo
sem_close(inicio);      //el proceso Escritor cierra el semáforo, puesto que yo no
                        //lo vuelve a utilizar
....
```

Cada proceso lector se queda esperando en el semáforo justo antes de abrir el fichero. Seguidamente incrementa el samáforo para que otro lector pueda entar:

```
....
fl.l_pid = getpid();
sem_wait(inicio);       //el proceso Lector espera por el semáforo
sem_post(inicio);       //el proceso Lector abre el semáforo
sem_close(inicio);      //el proceso Lector cierra el semáforo, puesto que yo no
                        //lo vuelve a utilizar
fd = open("fich", O_RDONLY);
                        //lo vuelve a utilizar
....
```

La solución planteada no contempla la eliminación del semáforo, es dicir no inclue el sem_unlink. El semáforo se queda en el sistema y con valor 1. Una alternativa de diseño sería plantear un proceso que crease el semáforo con valor 0, que luego generase los procesos Escritor y Lectores, que esperase por la finalización de todos ellos y que, finalmente, eliminase el semáforo.

Problema 4.19 (junio 2011)

Estamos desarrollando la aplicación de un cajero automático. En concreto nos ha encargado la operación de retirada de dinero. Se descompone la operación en los siguientes pasos:

Pasos genéricos del cajero, (partiendo de la pantalla de bienvenida):

- *Detección y lectura de la tarjeta de crédito.*
- *Lectura del pin de acceso y validación con la propia tarjeta. Al tercer error el cajero da por finalizada la sesión, se queda con la tarjeta y muestra un mensaje de que el usuario debe ponerse en contacto con su banco.*
- *Selección de la operación por parte del usuario.*

Pasos de la operación de retirada de dinero (esta parte es la que nos han encargado):

  - *Selección del importe.*
  - *Confirmación de la orden por el usuario.*
  - *Comprobación de que existen fondos en el cajero y en la cuenta del usuario.*
  - *Emisión del dinero.*
  - *Emisión del recibo.*

- *¿Desea realizar una nueva operación?*
- *….*
- *Expulsión de la tarjeta de crédito.*
- *Pantalla de bienvenida*

a) Indique justificadamente qué mecanismo de comunicación utilizaría entre la aplicación del cajero y la aplicación del sistema central que atiende a los cajeros.

b) Indique justificadamente la estructura de la aplicación del sistema central que atiende a los cajeros del banco.

c) Indique justificadamente qué mecanismo de sincronización utilizaría para resolver los problemas de carrera que pueden surgir por parte de los cajeros.

d) Indique en qué puntos de la operación del cajero indicados en el enunciado incluiría los mecanismos anteriores.

SOLUCIÓN

a) Deseamos una comunicación fiable entre el programa del cajero y el del sistema central, por lo que se utilizarán sockets TCP (stream).

b) Se trata de una clásica aplicación cliente-servidor en la que el servidor ejecuta en el sistema central y el cliente en el cajero. Dado que deseamos que se puedan atender a muchos cajeros simultáneamente, el servidor deberá generar un proceso o thread por cliente, que se encargue de atenderle de forma dedicada.

c) En esta aplicación nos encontramos con los siguientes problemas de carrera:

1. Hay que comprobar que hay suficiente dinero en el cajero. Como solamente hay un usuario por cajero, esta comprobación no es concurrente, por lo que no plantea problemas de carrera.
2. Dado que se han de modificar los datos bancarios de la cuenta afectada por la operación, hay que garantizar que el acceso a los mismos se haga de forma exclusiva: desde el momento en el que se leen, para comprobar que hay saldo, hasta el momento en el que quedan modificados.
3. Durante la emisión de los billetes se pueden producir problemas, puesto que la máquina expendedora se puede atascar o la corriente se puede cortar. Por tanto, es necesario garantizar que quedan anotados los billetes que realmente se expiden.
4. El que se imprima o no el recibo no es excesivamente problemático, por tanto esta operación no es necesario protegerla.

El mecanismo de sincronización más adecuado en esta aplicación es la transacción para resolver los problemas 2 y 3. Se trata, en este caso, de una transacción distribuida, puesto que afecta tanto al cajero como al sistema central.

d) El inicio de la transacción se debe establecer al recibirse la confirmación de la operación por parte del usuario (cuando se lea que ha pulsado la tecla de confirmación). El final de la transacción se debe producir cuando se ha terminado de expender todos los billetes. Ahora bien, como nos podemos quedar a mitad de la emisión de los billetes, la solución real tiene mayor complejidad.

Una alternativa es anidar, dentro de la transacción anterior, una transacción adicional por cada billete. En caso de fracasar una de esas transacciones se aborta la emisión de más billetes y se completa la transacción principal reflejando solamente el dinero realmente emitido.

Problema 4.20 (junio 2011)

Sea un sistema de reservas de consultas médicas por red para un centro médico. El servidor de reservas generará un proceso "reserva" de acceso a la tabla de reservas específico por cada cliente que se conecte para hacer una reserva y que le atiende durante toda la reserva. Las consultas tienen una duración de 10 minutos y la tabla de reservas está organizada por médico, mes, día, y hora.

El proceso de reservas es el siguiente:

- *El usuario selecciona un médico.*
- *Se le presenta una tabla con las horas de consulta libres, desde el primer día que tiene un hueco libre hasta 5 días laborables más.*
- *El usuario selecciona un hueco y pulsa "Reservar".*
- *Se le confirma la reserva con una interfaz que incluye un botón para imprimir la reserva.*

a) Indicar justificadamente en qué puntos de la secuencia anterior deberán introducirse los mecanismos de sincronización que eviten las condiciones de carrera.

b) Supondremos primero que la tabla de reservas se almacena en un conjunto de ficheros, teniendo cada fichero las reservas de un médico y un mes, y que el mecanismo de sincronización es el cerrojo. Plantear una solución en pseudocódigo, destacando el uso de los mecanismos de sincronización.

c) Supondremos ahora que la tabla de reservas se almacena en memoria. Seleccionar los mecanismos de sincronización más adecuados y plantear una solución en pseudocódigo, destacando el uso de los mecanismos de sincronización.

SOLUCIÓN

a) Para decidir en qué punto se deben poner los mecanismos de sincronización primero hay que determinar el comportamiento que queremos. Por un lado, queremos que el sistema tenga la mínima contención, es decir, que permita la máxima concurrencia. Por otro lado, tenemos que garantizar que dos usuarios no obtienen la misma cita (médico, día y hora).

El proceso tiene dos partes. Primero hay que leer uno o más ficheros para encontrar las consultas libres (puede que se tenga que acceder a dos ficheros de meses consecutivos para completar los 6 días laborables necesarios para construir la tabla que se presenta al usuario). Seguidamente, una vez realizada la selección por el usuario, hay que escribir el fichero para marcar la cita como comprometida.

Dado que un usuario puede tardar mucho en decidir qué cita seleccionar (incluso puede dejar la aplicación colgada), no parece que tenga sentido el considerar que la sección crítica incluya las lecturas más la posible escritura.

Es de destacar que si protegemos la sección de lectura del programa de forma independiente de la escritura (siguiendo el clásico modelo lector-escritor) no tenemos ninguna garantía, a la hora de formalizar la reserva, de que otro usuario más rápido no haya seleccionado ya la misma consulta. La protección de la lectura en el modelo lector-escritor evita simplemente que el lector pueda leer un estado inconsistente de la información. En nuestro caso, la protección de la lectura no es necesaria puesto que no puede retornar un estado inconsistente, dado que la operación de reserva consiste en una única escritura en el fichero de consultas incluyendo en el registro de una consulta el nombre del paciente, escritura que se hace de forma atómica.

Se propone entonces proteger el fichero correspondiente desde que se recibe el "Reservar" hasta que se graba en el mismo. El problema que puede aparecer es que la consulta haya sido asignada a otro usuario que no se demoró tanto en la selección. Para garantizar un funcionamiento correcto, la sección crítica, antes de escribir en el fichero, deberá volver a leerlo, para comprobar que la consulta sigue estando libre. En caso contrario deberá informar al usuario para que realice otra selección.

Igual ocurre en el caso de que la tabla de consultas resida en memoria.

b) Al haber seleccionado una consulta, ya queda definido el fichero a modificar (definido por el conjunto médico-mes) así como la región dentro del fichero (definido por el conjunto día-hora). La secuencia del proceso "Reserva" quedaría como sigue:

```
struct consult{
   long fechahora;
   char paciente[50];
};
```

En el servidor principal

```
.....
struct consult consulta;
<<Mediante un fork se crea el servidor particular de cada cliente>>
.....
```

En cada servidor particular

```
.....
<<Selección del médico y de la consulta a reservar>>
MedicoMes = <<Nombre del fichero con la reserva deseada>>
PosicionConsulta = <<Posición de la reserva deseada dentro del fichero>>
<<El usuario pulsa aceptar>>
fd = open(MedicoMes, O_RDWR);
fl.l_whence = SEEK_SET;
fl.l_start = PosicionConsulta;
fl.l_len = sizeof(consulta);
fl.l_pid = getpid();
fl.l_type = F_WRLCK
fcntl(fd, F_SETLKW, &fl);                  //Se espera por el cerrojo si está cogido
lseek (fd, PosicionConsulta, SEEK_SET);
read (fd, &consulta, sizeof(consulta));
if (strcmp(consulta.paciente,"") == 0) {       //Sigue libre la consulta
   lseek (fd,PosicionConsulta, SEEK_SET);
   strcpy(consulta.paciente, usuario);         //se añade el nombre del paciente a la consulta
   write (fd, &consulta, sizeof(consulta));
   reserva = 1;       //Indicamos que la reserva se ha realizado correctamente
} else {              //La consulta ha sido ocupada y no se puede hacer
   reserva = 0;       //Indicamos que la reserva NO se ha realizado
};
fl.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &fl);    //se libera el cerrojo
<<Si reserva == 0 hay que repetir la selección>>
.....
```

c) Dado que nos dicen que el servidor genera un proceso por cada cliente, dichos procesos son emparentados, por lo que se pueden utilizar semáforos sin nombre. Por lo mencionado en la sección a), necesitaremos un semáforo para proteger cada zona de la tabla que pueda ser accedida al tiempo, de forma que solamente un proceso "reserva" pueda realizar las operaciones de comprobar que la consulta sigue libre y de asignarla.

Dependiendo del grado de granularidad que queramos debemos generar más o menos semáforos. Por ejemplo, si el grano es toda la tabla, necesitaremos un solo semáforo. Si el grano es la tabla de un médico y un mes, necesitamos un semáforo por médico-mes. Si el grano es médico día, necesitamos un semáforo por médico y día. Evidentemente, mientras mayor sea el grano mayor será la contención. En nuestro caso, el código de la sección crítica (comprobación-reserva) es muy rápido, por lo que los procesos permanecen en la sección crítica muy poco tiempo, produciendo muy poca contención, por lo que consideremos que la granularidad médico-mes es la adecuada.

Vamos a suponer que la información en memoria cubre 6 meses (habrá una operación mensual que corra la ventana de tiempo de la aplicación, de forma que siempre se tengan residentes en memoria el mes actual y los cinco meses siguientes). También supondremos que tenemos MEDICOS médicos y que seleccionamos granularidad mensual. Por simplicidad consideraremos que todas las tablas de mes tienen 31 días y 60 consultas por día.

```
struct consult{
   long fechahora;
   char paciente[50];
};
```

En el servidor principal

```
.....
struct consult *consulta;
sem_t MedMes[6][MEDICOS];
for (i=0; i<6; i++)                        //Inicializamos los semáforos
   for(j=0; j<MEDICOS; j++)
```

```
      sem_init(&MedMes[i][j], 1, 1);      //Los semáforos son compartido y binarios
tamagno = 6*MEDICOS*31*60*sizeof(struct consult);
//Generamos la región compartida con las tablas de reservas. El uso de MAP_ANON evita el tener que
//utilizar un fichero y resulta en una región sin nombre.
org=mmap(NULL, tamagno, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1,0);
<<Se inicializa la tabla de reservas>>
.....
<<Mediante un fork se crea el servidor particular de cada cliente>>
.....
```

En cada servidor particular

```
....
<<Selección del médico y de la consulta a reservar>>
Medico=<<Nombre del médico seleccionado>>
Mes=<<Mes de la consulta seleccionada>>
Dia=<<Día de la consulta seleccionada>>
Hora=<<Índice de la hora de la consulta seleccionada>>
<<El usuario pulsa aceptar>>
sem_wait (&MedMes[Mes][Medico]);
posicion=org + Medico*Mes*Dia*Hora*sizeof(struct consult);
consulta = (struct consult *) posicion;        //Seleccionamos la consulta
if (strcmp(consulta->paciente,"") == 0) {      //Sigue libre la consulta
   strcpy(consulta->paciente,usuario);         //se añade el nombre del paciente a la consulta
   reserva = 1;     //Indicamos que la reserva se ha realizado correctamente
} else {            //La consulta ha sido ocupada y no se puede hacer
   reserva = 0;     //Indicamos que la reserva NO se ha realizado
};
sem_post (&MedMes[Mes][Medico]);
<<Si reserva == 0 hay que repetir la selección>>
....
```

Problema 4.21 (junio 2011)

Queremos diseñar un sistema para jugar a los barquitos. Cada usuario primeramente posicionará sus barcos en su consola. Seguidamente, y en turnos alternos, cada usuario marcará su disparo y, automáticamente, se le indicará si ha caído en "agua" o ha "tocado" un barco contrario. Se plantean tres posibles soluciones:

La primera consiste en un computador con dos consolas. Cada consola está atendida por un proceso.

a. Determinar los mecanismos de comunicación se podrían utilizar entre los dos procesos.

b. Indicar las ventajas e inconvenientes que tendría cada uno de ellos para esta aplicación.

c. Para cada mecanismo de comunicación anterior indicar cómo se consigue alternancia en los turnos.

La segunda es como la primera pero cada consola está atendida por un thread.

d. Seleccionar el mecanismo de comunicación más adecuado en este caso.

e. Indicar cómo se consigue la alternancia.

La tercera solución consiste en tener dos computadores personales comunicados por wifi.

f. Seleccionar el mecanismo de comunicación más adecuado en este caso.

g. Indicar cómo se consigue la alternancia.

h. Indicar, adicionalmente, el esquema de procesos involucrados en la aplicación.

SOLUCIÓN

a, b y c) Se podría utilizar uno cualquiera de los siguientes mecanismos:

Pipes. Exige que los que los procesos que atienden ambas consolas estén emparentados, bien porque el de la consola B es hijo del de la consola A, bien porque ambos sean hijos del proceso que inicia el juego. Es la solución

más adecuada, ya que es un mecanismo simple y bloqueante. El pipe permite sincronizar los procesos que los utilizan, en este caso para garantizar la alternancia de los jugadores, con un esquema como el siguiente:

Consola A que inicia el juego

```
primeravez = 1;
while (jugando){
   if  (!primeravez) {
      read (pipe2[0], &jugada, sizeof(jugada));
      <<Genera resultado jugada del contrario>>
      write (pipe1[1], &resultado, sizeof(resultado));
   }
   primeravez = 0;
   <<Compone jugada>>
   write (pipe1[1], &jugada, sizeof(jugada));
   read (pipe2[0], &resultado, sizeof(resultado));
   <<Muestra resultado jugada y comprueba si ha terminado el juego>>
}
```

Consola B

```
while (jugando) {
   read (pipe1[0], &jugada, sizeof(jugada));
   <<Genera resultado jugada del contrario>>
   write (pipe2[1], &resultado, sizeof(resultado));
   <<Compone jugada>>
   write (pipe2[1], &jugada, sizeof(jugada));
   read (pipe1[0], &resultado, sizeof(resultado));
   <<Muestra resultado jugada y comprueba si ha terminado el juego>>
}
```

FIFOS. Se puede utilizar aunque los procesos no estén emparentados. La forma de sincronizar los procesos sería muy similar a la propuesta para los pipes, puesto que utilizan igualmente los servicios read y write bloqueantes.

Memoria compartida. En este caso es necesario utilizar un mecanismo de sincronización para ir garantizando el turno. Como se trata de procesos podemos utilizar semáforos (con o sin nombre dependiendo de que los procesos estén emparentados. Nos basta un semáforo binario por consola, que esté inicializado a 1 para el caso de la consola que juega primero y a cero para la otra. El esquema sería el siguiente:

Para iniciar el juego

```
struct jug{
   int X;
   int Y;
   int Resultado;
};
.....
struct jug *JugadaA, *JugadaB;
sem_t ConsolaA;
sem_t ConsolaB;
sem_init(&ConsolaA, 1, 1);    //Los semáforos son compartido y binarios
sem_init(&ConsolaB, 1, 0);
//Generamos la región compartida con espacio para dos jugadas
jugadas = sizeof(struct jug);
JugadaA=mmap(NULL, 2*jugadas, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1,0);
JugadaB = JugadaA + 1;   //Avanzamos el tamaño de la estructura jug
.....
<<Mediante un fork se crea el proceso de la otra consola>>
.....
```

Consola A que inicia el juego

```
primeravez = 1;
while (jugando) {
   sem_wait (&ConsolaA);
   if (!primeravez) {
      <<Muestra resultado jugada de acuerdo al valor de JugadaA->Resultado>>
      <<Calcula el resultado de la jugada de A y lo pone en JugadaB->Resultado>>
      <<Comprueba si ha terminado el juego>>
      <<Compone jugada generando MiX y MiY>>
   }
```

```
   primeravez = 0;
   JugadaA->X = MiX;
   JugadaA->Y = MiY;
   sem_post (&ConsolaB);
}
```

Consola B

```
while (jugando) {
   sem_wait (&ConsolaB);
   <<Muestra resultado jugada de acuerdo al valor de JugadaB->Resultado>>
   <<Calcula el resultado de la jugada de B y lo pone en JugadaA->Resultado>>
   <<Comprueba si ha terminado el juego>>
   <<Compone jugada generando MiX y MiY>>
   JugadaB->X = MiX;
   JugadaB->Y = MiY;
   sem_post (&ConsolaA);
}
```

Sockets de cualquiera de los tipos stream o datagrama en dominio UNIX o INET. Dentro de un computador no hay problema sde pérdidas de paquetes, ni de duplicados, por lo que los sockets datagrama constituyen una buena solución, siendo más ligeros que los stream. Dado que las operaciones de send y recv pueden ser bloqueantes, se puede lograr la alternancia con un esquema similar al planteado para los pipes. En la sección f se detalla la solución.

Ficheros. Evidentemente podemos comunicar los procesos a través de un fichero, pero no tiene ninguna ventaja, puesto que exige sincronización como en el caso de memoria compartida y es menos eficiente que ésta, puesto que hay que hacer más llamadas al sistema operativo (operaciones lseek, read y write).

d y e) En el caso de los threads el mecanismo natural de comunicación es la memoria, dado que se comparte toda la imagen de memoria. Se definen dos estructuras jug, JugadaA y JugadaB como variables globales.

Para sincronizar los threads se podrían utilizar semáforos sin nombre con el esquema planteado en la sección c). Sin embargo, podemos utilizar también mutex y condiciones con el esquema siguiente.

```
pthread_mutex_t mutex;  //Control de acceso
pthread_cond_t a_jugarA, a_jugarB;//Condiciones de espera
int juegaA =1, juegaB=0;   //Estado del acceso, empieza a jugar el A
```

Consola A que inicia el juego

```
primeravez = 1;
while (jugando) {
   pthread_mutex_lock(&mutex);
   while(juegaA !=0) //cond. espera
      pthread_cond_wait(&a_jugarA, &mutex);
   juegaA=0;
   pthread_mutex_unlock(&mutex);
   if (!primeravez) {
      <<Muestra resultado jugada de acuerdo al valor de JugadaA->Resultado>>
      <<Calcula el resultado de la jugada de B y lo pone en JugadaB->Resultado>>
      <<Comprueba si ha terminado el juego>>
      <<Compone jugada generando MiX y MiY>>
   }
   primeravez = 0;
   JugadaA->X = MiX;
   JugadaA->Y = MiY;
   pthread_mutex_lock(&mutex);
   juegaB=1;
   pthread_cond_signal(&a_jugarB);
   pthread_mutex_unlock(&mutex);
}
```

Consola B

```
while jugando {
   pthread_mutex_lock(&mutex);
   while(juegaB !=0) //cond. espera
      pthread_cond_wait(&a_jugarB &mutex);
   juegaB=0;
   pthread_mutex_unlock(&mutex);
```

```
      <<Muestra resultado jugada de acuerdo al valor de JugadaB->Resultado>>
      <<Calcula el resultado de la jugada de A y lo pone en JugadaA->Resultado>>
      <<Comprueba si ha terminado el juego>>
      <<Compone jugada generando MiX y MiY>>
      JugadaB->X = MiX;
      JugadaB->Y = MiY;
      pthread_mutex_lock(&mutex);
      juegaA=1;
      pthread_cond_signal(&a_jugarA);
      pthread_mutex_unlock(&mutex);
   }
```

f, g y h) Al tratarse de un sistema distribuido, el mecanismo de comunicación es el socket en el dominio INET. Aunque estemos en una red local (wifi) la fiabilidad de estas redes no es muy grande, por lo que se seleccionan los socket de tipo stream.

En este caso, los procesos son totalmente independientes, por lo que primero se tienen que poner de acuerdo para realizar la conexión TCP. La solución más simple es que uno de ellos sea el maestro y actúe como servidor, estando a la espera de que se conecte el otro proceso que actuará como cliente. Llamando A al que hace de servidor, el esquema de procesos que podemos plantear es uno de los dos mostrados en la figura Error: Reference source not found. En este caso, puesto que solamente hay un posible cliente, la solución 2, que implica un servidor exclusivo, es la más adecuada, aunque la solución clásica del servidor delegado también serviría.

Una vez establecido el socket, la alternancia se consigue de forma similar a con pipes, utilizando servicios bloqueantes:

Consola A que inicia el juego

```
primeravez = 1;
while (jugando) {
   if (!primeravez) {
      recv(cd, &jugadaB, sizeof(jugada), 0);  //Bloqueante
      <<Genera resultado jugada del contrario>>
      send(cd, &jugadaB, sizeof(jugada), 0);  //Bloqueante
   }
   primeravez = 0;
   <<Compone jugada>>
   send(cd, &jugadaA, sizeof(jugada), 0);  //Bloqueante
   recv(cd, &jugadaA, sizeof(jugada), 0);  //Bloqueante
   <<Muestra resultado jugada y comprueba si ha terminado el juego>>
}
```

Consola B

```
while (jugando) {
   recv(cd, &jugadaA, sizeof(jugada), 0);  //Bloqueante
   <<Genera resultado jugada del contrario>>
   send(cd, &jugadaA, sizeof(jugada), 0);  //Bloqueante
   }
   <<Compone jugada>>
   send(cd, &jugadaB, sizeof(jugada), 0);  //Bloqueante
   recv(cd, &jugadaB, sizeof(jugada), 0);  //Bloqueante
   <<Muestra resultado jugada y comprueba si ha terminado el juego>>
}
```

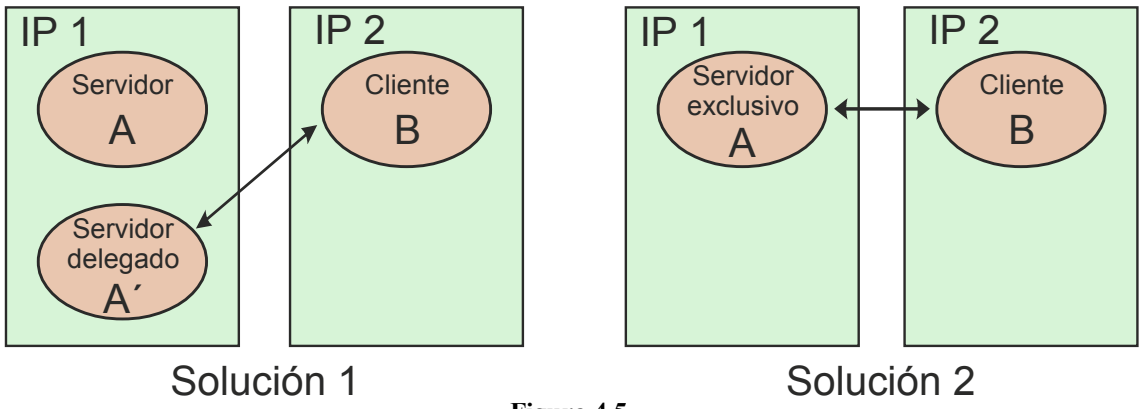


Figura 4.5

Problema 4.22 (enero 2012)

Se desea diseñar una aplicación de trabajo colaborativo por Internet. Una de las funciones a incluir es una aplicación de edición que permita trabajar a varios usuarios simultáneamente sobre un mismo documento. Otra es una agenda para reserva de la sala de reuniones.

a) Uno de los diseñadores plantea una solución cliente-servidor, mientras que otro aboga por una solución P2P. Indicar las ventajas e inconvenientes que presentan estas dos soluciones para implementar la aplicación de edición.

b) El grupo de diseñadores opta por la solución cliente-servidor, que debemos concretar. Realizar una gráfica que represente la máquina que albergue al servidor así como tres máquinas con clientes, mostrando los procesos involucrados en la aplicación de edición y la relación entre ellos. Justificar la solución planteada.

Para la agenda de reservas se decide utilizar un fichero en la máquina servidora que contenga el mes en curso y los dos meses siguientes. El día uno de cada mes se lanza un proceso de mantenimiento que rescribe el fichero, eliminado el mes que acaba de terminar y añadiendo otro mes.

c) Un diseñador sugiere emplear un proceso servidor con funcionamiento serie (hay un solo proceso que atiende de forma secuencial a los clientes), mientras que otro plantea la solución clásica que utiliza un proceso para atender a cada cliente. Indicar las razones que alega el diseñador que sugiere el servidor con funcionamiento serie y la replica que realiza el que sugiere la solución clásica.

El grupo se decanta por la solución clásica.

d) Indicar el mecanismo de comunicación a utilizar entre los clientes y el servicio.

e) Determinar los posibles problemas de concurrencia que existen (tener en cuenta tanto los clientes como el proceso de mantenimiento).

f) Seleccionar los mecanismos de sincronización a utilizar para resolver los problemas de concurrencia, indicando las razones en las que se basa la selección.

g) Plantear el pseudocódigo de sincronización necesario en los distintos procesos, incluyendo el de mantenimiento.

Solución

a) Tanto la edición de un documento como la gestión de la agenda de reservas exige que los usuarios tengan acceso a las actualizaciones que realicen otros usuarios. Esto es mucho más fácil de conseguir en una solución centralizada, en la que se dispone de un servidor que almacena dichos documentos, que con un esquema P2P. Por otro lado, el número de usuarios que se puede esperar no es muy elevado, dado que es impensable tener cientos de personas trabajando sobre un mismo documento en un instante determinado o accediendo a la agenda, por lo que la solución centralizada cliente-servidor no plantea problemas de congestión.

b) Se podrían plantear soluciones diferentes para el editor y para la agenda. Seleccionamos la solución de servidor dedicado, por lo que el esquema de procesos involucrados es la de la figura 4.6. Es de notar que los servidores dedicados deben sincronizarse entre sí, puesto que acceden al mismo documento.

c) El uso de un servidor serie para la agenda elimina la concurrencia, puesto que tendríamos un solo cliente accediendo a la misma en cada instante. Sin embargo, esta solución no parece aceptable puesto que introduce una contención inadmisible (el cliente que está accediendo a la agenda puede dejar al servidor esperando indefinidamente). La solución clásica evita la contención, pero, al presentar concurrencia, debe ser diseñada de forma que se imposibiliten las condiciones de carrera.

d) Nos indican que es una solución Internet, es decir, distribuida en WAN. Por lo tanto, se requiere un mecanismo de comunicación distribuido y fiable, es decir, sockets AF_INET de tipo Stream, es decir con protocolo TCP.

e) Los problemas de concurrencia ocurren entre los servidores dedicados de la agenda y entre estos y el proceso de mantenimiento. Es necesario evitar que dos clientes puedan reservar la misma hora del mismo día, y es necesario que el proceso de mantenimiento acceda de forma exclusiva a todo el fichero para reescribirlo correctamente.

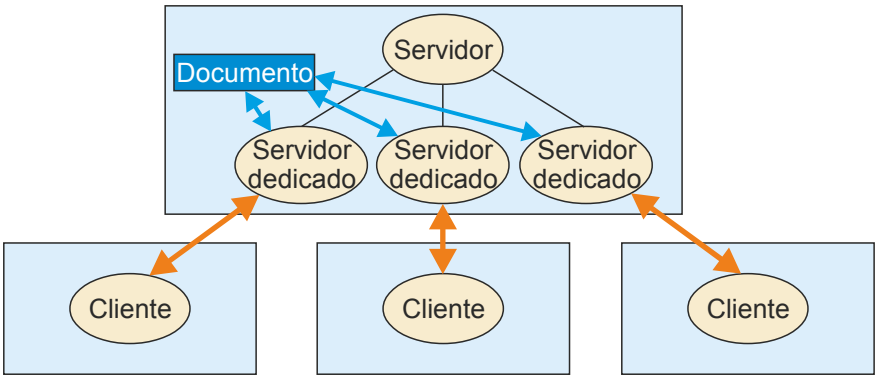


Figura 4.6

e) Los problemas de concurrencia ocurren entre los servidores dedicados de la agenda y entre estos y el proceso de mantenimiento. Es necesario evitar que dos clientes puedan reservar la misma hora del mismo día, y es necesario que el proceso de mantenimiento acceda de forma exclusiva a todo el fichero para reescribirlo correctamente.

f) Los servidores dedicados de la agenda afectan a una pequeña parte del fichero de reservas, la parte donde se marca la reserva de la hora y día. Por otro lado, se trata de procesos y no threads, que acceden a un fichero compartido, por lo que los mecanismos de sincronización disponibles son fundamentalmente dos: el semáforo y el cerrojo. Dado que cada reserva afecta a una pequeña zona del fichero, la solución más adecuada para sincronizar los servidores dedicados es la basada en cerrojos. Por otro lado, el proceso de mantenimiento requiere acceso exclusivo a todo el fichero, por lo que es incompatible con los servidores dedicados. En este caso, se podría utilizar indistintamente como mecanismo de sincronización un semáforo o un cerrojo sobre todo el fichero. Otra solución consistiría en parar el servicio de agenda cuando se lance el proceso de mantenimiento, operación que se podría plantear durante el fin de semana.

g) Los servidores dedicados leen y escriben en el fichero. Hay que determinar si se trata de un problema tipo lector-escritor o es un problema simplemente de escritores.

La secuencia de reserva es la siguiente: el usuario deberá ser presentado con las reservas disponibles (por ejemplo, las de una semana), lo que requiere leer del fichero (el proceso actúa como lector). El usuario selecciona la hora y pulsa aceptar. Se modifica el fichero (el proceso actúa como escritor) y se informa al usuario del éxito de la operación.

Si utilizamos una solución tipo lector-escritor en la fase de lectura debemos establecer un cerrojo compartido sobre toda la semana leída. Dos o más procesos podrían leer la misma semana. Sin embargo, a la hora de seleccionar una reserva se verían imposibilitados de realizar la escritura hasta que ningún proceso tuviera un cerrojo sobre la zona (la escritura requiere un cerrojo exclusivo). Esto produciría una alta contención en el sistema, lo que no es recomendable.

Si utilizamos una solución de escritores, en donde solamente los escritores solicitan cerrojos, se puede dar la situación de que varios usuarios reserven simultáneamente una misma hora, que se les ha presentado como libre. Sin embargo, solamente uno será capaz de establecer el cerrojo exclusivo, realizando realmente la reserva. Para todos los demás la reserva fracasará. Un aspecto adicional a considerar es si en esta solución la operación de lectura puede producir información inconsistente. Este no es el caso, puesto que la reserva se puede hacer con una sola operación de escritura en el disco, que es una operación atómica, por lo que la información contenida en el fichero será siempre consistente.

Por otro lado, el proceso de mantenimiento requiere acceso exclusivo a todo el fichero, puesto que se modifica en su totalidad. Es muy posible que la actualización del fichero se realice mediante una serie de operaciones de lec-

tura y escritura, lo que podría producir estados inconsistentes en el mismo durante algunos instantes, por lo que sería conveniente parar el servicio mientras se hace el mantenimiento. En el caso de parar el servicio no sería necesario incluir en proceso de mantenimiento ningún mecanismo de sincronización adicional.

El código de sincronización de los servidores dedicados es el siguiente:

```
int fd;
struct flock fl;
fd = open("Agenda", O_RDWR);
   <<Presentación de agenda>>
   <<Selección reserva>>
fl.l_whence = SEEK_SET;
fl.l_start = comienzo;        //comienzo:variable que indica el comienzo de la sección a modificar
fl.l_len = longitud;          //longitud: variable con el nº de bytes a modificar
fl.l_pid = getpid();
fl.l_type = F_WRLCK;          //cerrojo exclusivo
fcntl(fd, F_SETLKW , &fl);    //establece cerrojo de forma síncrona
//Comprobamos si sigue libre la reserva
lseek(fd, comienzo, SEEK_SET);
read (fd, &val, longitud);    //obtenemos la reserva. val: contiene la información de la reserva
if (reservalibre(val)) {      //Reserva libre u ocupada
   <<Se introducen los datos de la reserva en val>>
   lseek(fd, comienzo, SEEK_SET);
   write(fd, &val, longitud);
   fl.l_type = F_UNLCK;
   fcntl(fd, F_SETLK, &fl);      //se libera el cerrojo
   <<Avisar que la operación se ha realizado>>
} else {
   <<Avisar que la operación ha fracasado>>
}
```

El código de sincronización del proceso de mantenimiento, para el caso de usar cerrojos, sería el siguiente:

```
int fd;
struct flock fl;
fd = open("Agenda", O_RDWR);
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = 0;      //cerrojo sobre todo el fichero
fl.l_pid = getpid();
fl.l_type = F_WRLCK;          //cerrojo exclusivo
fcntl(fd, F_SETLKW, &fl);     //establece cerrojo de forma síncrona
   <<operación de mantemiento>>
fl.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &fl);        //se libera el cerrojo
```

Problema 4.23 (enero 2012)

Un sistema de ficheros de tipo UNIX, (diseñado para un pequeño sistema portátil) presenta las siguientes características:

- Representación de ficheros mediante nodos-i con 12 direcciones directas a bloque, un indirecto simple y un indirecto doble. Direcciones de bloque de 4 bytes.
- El tamaño del bloque del sistema de ficheros es de 2 KiB y emplea una cache de 1 MiB con una política de reemplazo LRU.

Sobre este sistema se ejecutan un proceso **escritor** y cuatro procesos **lectores** con los fragmentos de código indicados más adelante.

Escritor	**Lector**

```
#define DATA_SIZE     1024
#define N             2048
char buffer[DATA_SIZE];
int fd, i;
struct flock fl;
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = DATA_SIZE;
fl.l_pid = getpid();
fl.l_type = F_WRLCK;
fd = open("fich", O_CREAT | O_WRONLY,
0666);
ftruncate(fd, DATA_SIZE*N);
fcntl(fd, F_SETLKW, &fl);

for (i = 0 ; i < N; i++) {
   write(fd, buffer, DATA_SIZE);
   if (i < N-1) {
      fl.l_type = F_WRLCK;
      fl.l_start = DATA_SIZE*(i+1);
      fcntl(fd, F_SETLKW, &fl);
   }
   fl.l_type = F_UNLCK;
   fl.l_start = DATA_SIZE*i;
   fcntl(fd, F_SETLK, &fl);
}
close(fd);
```

```
#define DATA_SIZE     1024
#define N             2048
char buffer[DATA_SIZE];
int fd, i;
struct flock fl;
fl.l_whence = SEEK_SET;
fl.l_len = DATA_SIZE;
fl.l_pid = getpid();
fd = open("fich", O_RDONLY);

for (i = 0 ; i < N; i++) {
   fl.l_start = DATA_SIZE*i;
   fl.l_type = F_RDLCK;
   fcntl(fd, F_SETLKW, &fl);
   read(fd, buffer, DATA_SIZE);
   fl.l_type = F_UNLCK;
   fcntl(fd, F_SETLK, &fl);
}
close(fd);
```

a) Calcule el tamaño máximo que puede tener un fichero en dicho sistema.

Suponiendo que la cache se encuentra inicialmente vacía, que no se realiza ninguna operación de limpieza de cache (sync) durante la ejecución del mencionado programa, que el escritor ejecuta su primer fcntl antes que ningún código lector ejecute el open y que el fichero "fich" existía previamente con un tamaño de 16 MiB, se pide:

b) ¿Puede un lector adelantar al escritor, es decir, leer una zona todavía no escrita por el escritor? Justifique la respuesta.

c) ¿Cuántos accesos a disco se producen durante la ejecución del bucle "for" del proceso escritor? Suponer una política de actualización write-through (escritura inmediata). Justificar la respuesta, e indicar cuántos accesos son de lectura y cuántos son de escritura.

d) Responder a la pregunta anterior, pero para el bucle "for" de cada proceso lector.

SOLUCIÓN

a) (K, M, G = Prefijos binarios)

(Como no se indica otra cosa, suponemos 1 agrupación = 1 bloque y lo denominaremos 'bloque' en adelante)

Primero tenemos que calcular el número de punteros que caben en 1 bloque = 2 KiB / 4B = 1/2 K = **512 punteros por bloque** = 2^9

12 directos: 12 bloques

1 indirecto simple: 512 bloques

1 indirecto doble: (512)^2 bloques = (2^9)^2 = 2^18 = 2^8 * 2^10 = 256 Ki bloques

TOTAL: (256 K + 512 + 12)blq * 2 KiB/blq = 512 MiB + 1 MiB + 24 KiB = **513 MiB + 24 KiB**

b) El escritor da al fichero un tamaño de DATA_SIZE*N y establece un cerrojo de tamaño DATA_SIZE, antes de que cualquier lector pueda establecer su cerrojo. Seguidamente escribe en el bloque y amplía el cerrojo al siguiente bloque de tamaño DATA_SIZE. Finalmente, libera el cerrojo del bloque escrito. Por tanto, **los lectores no pueden adelantar al escritor**.

c) 'ftruncate' trunca el fichero de 16 MiB a 2 MiB. El bucle del escritor realiza 2048 iteraciones y en cada una escribe 1 KiB de datos (secuencialmente desde el principio de fichero), por lo que sobrescribe los 2 MiB. Así, cada

iteración escribe 1/2 bloque de disco. La primera iteración (i=0) requiere 1 acceso a disco de lectura (la caché está vacía y el bloque tenía datos) y 1 acceso a disco de escritura (por la política *write-through*). La segunda iteración (i=1) solo requiere 1 acceso a disco de escritura, pues el bloque ya ha sido traído a caché en la iteración anterior. Así, las iteraciones con i impar provocan 1 acceso y las de i par 2 accesos.

Esto no se ve afectado por que la memoria caché sea menor (1 MiB) que el total a escribir. Cuando la caché esté llena, el siguiente bloque se reemplaza sin accesos adicionales a los mencionados, debido a la política *write-through*.

En conclusión, el bucle del escritor provoca **1024 accesos de lectura a disco y 2048 accesos de escritura**.

d) El bucle de cada proceso lector realiza también 2048 iteraciones y en cada una lee secuencialmente 1 KiB de datos. En condiciones normales de carga del sistema, cada proceso lector recibirá rodaja de procesador con una frecuencia similar al proceso escritor, por lo que la ejecución de cada iteración de lector no debe retrasarse mucho respecto de la del escritor. Así, cada lectura pide un bloque que se encuentra en la caché por lo que no provocará **ningún acceso a disco**.

Problema 4.24 (junio 2012)

Una aplicación se compone de dos procesos pesados cuyo código se adjunta, que comparten un buffer circular formado por registros de tamaño fijo y que utilizan un FIFO (llamado miFIFO).

a) Indicar los problemas que tiene el código propuesto.
b) Plantear una o varias soluciones a los problemas encontrados.
c) ¿Qué cambios se deberían introducir en la solución propuesta si en vez de procesos se utilizasen threads?
d) ¿Qué cambios habría que hacer si se quisiese que los procesos ejecutasen en máquinas distintas?

Ahora tenemos varios productores y varios consumidores que comparten el buffer circular utilizado en los códigos adjuntos, con la siguiente modificación: el buffer incluye una cabecera con información de gestión y/o sincronización del buffer.

e) ¿Cómo resolveríamos el problema de sincronización para procesos pesados? Plantear la información a incluir en la cabecera así como el o los mecanismos de sincronización a utilizar.

```
// Definición del registro
struct registro {
   int numero;
   char texto[280];
};

int main(void)  // Programa productor.
{
   int fd_buff,fd_fifo;
   struct stat bstat;
   int i, NR;
   struct registro * tabla;
   struct registro * registro;

   fd_buff = open ("buffer", O_RDWR);
   fstat(fd_buff, &bstat);
   tabla = mmap(0,bstat.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd_buff,0);
   close(fd_buff);
   fd_fifo = creat("miFIFO", 0666); // Se supone que miFIFO es un FIFO y que ya está creado.
   NR = bstat.st_size / sizeof(struct registro);

   i = 0;
   while(1) {
      // buffer circular
      registro = &(tabla[i%NR]);
      registro->numero = i;
      <<se genera el contenido del registro en el buffer, lo que puede tardar bastante>>
      write(fd_fifo,&registro,sizeof(struct registro *)); // envio de la dirección del registro
      i++;
   }
   return 0;
```

```
}

int main(void) // Programa consumidor.
{
   int fd_buff, fd_fifo;
   struct stat bstat;
   int i;
   struct registro * tabla;
   struct registro * registro;

   fd_buff = open ("buffer", O_RDWR);
   fstat(fd_buff, &bstat);
   tabla = mmap(0,bstat.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd_buff,0);
   close(fd_buff);
   fd_fifo = open("miFIFO", O_RDONLY);

   i = 0;
   while(1) {
      read(fd_fifo,&registro,sizeof(struct registro *));
      << se procesa el registro, lo que puede tardar bastante>>
      printf("%d %d\n", i, registro->numero);
      i++;
   }
   return 0;
}
```

SOLUCIÓN

a) El código propuesto tiene dos problemas principales. Por un lado, el productor pasa una dirección absoluta al consumidor. Dado que las dos regiones de memoria no tienen por qué estar en la misma dirección, se producirá un error al acceder al registro por parte del consumidor.

Por otro lado, la sincronización entre ambos procesos no es correcta. Si bien, el consumidor se sincroniza correctamente con el productor mediante miFIFO, cada dirección indica que está disponible un nuevo registro. El productor desconoce el avance del productor, lo que puede llevar a que llene completamente el buffer y llegue a sobrescribir registros no consumidos, si el consumidor es más lento que él.

b) Para que el consumidor haga correctamente el acceso al buffer el productor debe enviar una dirección relativa al comienzo de la región compartida. Por ejemplo, puede enviar el índice i.

Para evitar que el productor sobrescriba registros no consumidos podemos utilizar varias soluciones.

Una primera alternativa podría consistir en que el consumidor, una vez consumido el registro, pusiese `registro->numero = 0`, de esta forma el productor pudiese comprobar que el registro está vacío antes de escribirlo. Sin embargo, el productor tiene el siguiente problema: si encuentra el registro lleno, porque ha adelantado al consumidor, ¿cómo se entera de que queda vacío? Una espera activa leyendo reiteradamente el registro para ver que ya está vacío no es muy recomendable.

Una segunda alternativa consiste en utilizar otro FIFO, que llamaremos suFIFO. En suFIFO introduciremos inicialmente NR bytes (NR es el número de registros de buffer). Cada byte se utilizará como testigo de un registro, indicando que está libre. Esta operación inicial la hace el productor antes de producir ningún registro.

El productor antes de escribir en el buffer lee un byte del suFIFO, captando el testigo del registro sobre el que va a escribir. Por su parte, el consumidor escribe un byte en suFIFO cuando ha terminado con el registro, devolviendo el testigo.

Si el buffer está lleno, no habrá ningún byte en suFIFO, por lo que el productor se quedará esperando en la lectura de suFIFO hasta que el consumidor escriba un byte. Por otro lado, las direcciones que contenga miFIFO indica el número de registros útiles que todavía no han sido consumidos. Si no hay ninguno, el consumidor se queda esperando a que el productor escriba una dirección.

Una tercera alternativa consiste en crear un semáforo con nombre, inicializado a NR (número de registros de buffer). El productor hará un `sem_wait` sobre dicho semáforo antes de escribir en el registro y el consumidor hará un `sem_post` después de recoger un registro.

Una cuarta alternativa sería eliminar la región de memoria compartida y enviar el registro completo por el FIFO.

Finalmente se podría haber planteado el uso de cerrojos sobre ficheros. Una primera consideración es que lo que se comparte es una región de memoria compartida no un fichero. Sin embargo, si no se cierra el descriptor del fichero proyectado, se podrían usar los cerrojos, puesto que de forma indirecta nos protegerían las mismas zonas de la región de memoria.

Una segunda consideración es que no es un mecanismo adecuado para nuestro problema, que consiste en que el productor reescriba un registro que todavía no ha sido consumido. Puesto que si no está siendo consumido no habrá cerrojo sobre el mismo.

c) Para el caso de threads lo primero que hay que hacer es cambiar la región de memoria compartida por una declaración de buffer como variable global. Tampoco tiene sentido plantear una comunicación mediante un FIFO, por lo que la sincronización se debería hacer mediante mutex y condiciones.

La información de control que necesitamos es el número de registros libres en el buffer. Dicha información se inicia a NR. El productor la decrementa cada vez que produce un registro y el consumidor la incrementa cada vez que consume uno. Los códigos de sincronización serían los siguientes:

```
//Bucle del productor
while(1) {
   <<se genera el contenido del registro en el buffer, lo que puede tardar bastante>>
   mutex_lock(mimutex);
   while (numeroregistroslibre < 0)
      condition_wait(cond, mimutex);
   numeroregistroslibres--
   //se guarda en el registro i%NR del buffer
   registro = &(tabla[i%NR]);
   registro->numero = i;
   strcpy(registro->texto, mitexto);   //Se supone que mitexto es un string válido
   condition_signal(cond);
   mutex_unlock(mimutex);

   i++;
}

//Bucle del consumidor
while(1) {
   mutex_lock(mimutex);
   while (numeroregistroslibres < NR)
      condition_wait(cond, mimutex);
   numeroregistroslibres++
   //se lee el  registro i%NR del buffer
   registro = &(tabla[i%NR]);
   mii= registro->numero;
   strcpy(mitexto, registro->texto);
   condition_signal(cond);
   mutex_unlock(mimutex);
   <<se consume el registro, lo que puede tardar bastante>>

   i++;
}
```

La contención producida por la solución propuesta es pequeña, puesto que el código ejecutado en la sección crítica es muy pequeño.

d) Para el caso de máquinas distintas los procesos no pueden compartir regiones de memoria ni FIFO.

La solución más simple sería utilizar sockets de dominio AF_INET. Como la comunicación debe ser segura usaremos el tipo Stream y protocolo el IPPROTO_TCP. A través del socket se envía el registro completo, de forma similar a la cuarta alternativa del apartado b).

Si se desea conservar la filosofía propuesta, se podría utilizar un fichero en red en vez de la región de memoria compartida y valdría la segunda alternativa del apartado b), pero usando dos socket en vez de dos FIFO. Sin embargo, esta solución puede tener problemas debido a la semántica de los sistemas de ficheros distribuidos.

e) En caso de haber varios productores y consumidores, se podría utilizar la alternativa cuarta del apartado b). Un FIFO en el que los productores escriben el registro completo y del que los consumidores leen el registro completo.

Si conservamos la filosofía de la región de memoria compartida, necesitamos saber qué registros están ocupados en cada instante, de forma que un productor elija de forma exclusiva uno libre y un consumidor elija uno relleno y no consumido.

Podemos añadir una cabecera que sea un mapa de bits que nos indique los registros libres del buffer. El acceso a esta cabecera debe ser exclusivo, lo que podemos conseguir mediante un semáforo binario inicializado a 1. Por otro lado, hay que evitar que un productor haga espera activa si no hay registros libres así como que un consumidor haga espera activa si no hay registros rellenos. Para eso podemos utilizar un semáforo que indique los registros libres, que estará inicializado a NR y otro que indique los ocupados, inicializado a 0. Las secuencias son las siguientes:

```
//Bucle del productor
while(1) {
   <<se genera el contenido del registro en el buffer, lo que puede tardar bastante>>
   sem_wait(registroslibres)
   sem_wait(cabecera)
   <<Se busca un registro libre reglibre  y se marca como ocupado en la cabecera>>
   //se guarda en el registro
   registro = &(tabla[reglibre]);
   registro->numero = reglibre;
   registro->texto = mitexto;
   sem_post(cabecera)
   sem_post(registrosocupados)

   i++;
}

//Bucle del consumidor
while(1) {
   sem_wait(registrosocupados)
   sem_wait(cabecera)
   <<Se busca un registro ocupado regocupado  y se marca como libre en la cabecera>>
   //Se lee el registro
   registro = &(tabla[regocupado]);
   mii= registro->numero;
   mitexto = registro->texto;
   sem_post(cabecera)
   sem_post(registroslibres)
   <<se consume el registro, lo que puede tardar bastante>>

   i++;
}
```

La contención producida por la solución propuesta es pequeña, puesto que el código ejecutado en la sección crítica es pequeño.

En este caso el uso de un cerrojo sobre la cabecera tendría la misma funcionalidad que el semáforo cabecera. Sin embargo, por lo comentado en la sección b) nos parece mucho más adecuado el semáforo.

Problema 4.25 (junio 2012)

Se quiere escribir un programa que con un único proceso pesado implemente una cadena de procesamiento compuesta por N etapas concurrentes en serie, implementadas con procesos ligeros. Cada etapa debe recibir (rol de consumidor) un número entero procedente de la etapa anterior, procesarlo y enviar (rol de productor) el resultado (otro entero) a la etapa siguiente. No se permite el uso de pipes ni FIFOs para comunicar las etapas. El procesamiento del dato en cada etapa consiste en la invocación de la función de biblioteca "`int realiza_operacion(int Netapa, int dato)`*", siendo* `Netapa` *el número de etapa de la cadena y* `dato` *el dato a procesar. No programar esta función. La entrada de la primera etapa debe ser la entrada estándar (flujo de números enteros) y la salida de la última etapa la salida estándar. Elegir libremente la representación (binario, ascii...) de los datos en la entrada y salida estándar. Se pide:*

a) Escribir una primera versión del programa sin protección de secciones críticas. Sugerencia: Programar en primer lugar una etapa intermedia genérica y después añadir el código adicional para las etapas de los extremos.

b) Indicar (en el programa escrito) qué tramos de código son secciones críticas y por qué.

c) Añadir el código necesario para proteger las secciones críticas.

d) Explicar brevemente (2-5 líneas) si el mecanismo de sincronización empleado produce esperas activas. Si es así, ¿Qué otro mecanismo podría evitarlo?

SOLUCIÓN

a)

(NOTA: Se han omitido las comprobaciones de retornos de error)

Cuerpo de texto Times New Roman 10pt. Siguientes párrafos

```
/* N-1 buffers entre etapas (1 int/buffer):
    Etapa#0 -> buffer[0] -> Etapa#1 -> buffer[1]...
*/
int buffer[N-1];

int realiza_operacion(int Netapa, int dato);

void *etapa(void *p){
  int dato_in, dato_out;
  int i = (int)p; //Restaura i de main

  while(1){
    if(i == 0){//Etapa #0 lee de stdin
      scanf("%i", &dato_in);
    }else{
      dato_in = buffer[i-1]; // CONSUMIDOR de buffer[i-1]
    }

    dato_out = realiza_operacion(i, dato_in);

    if(i == N-1){//Etapa #N-1 escribe en stdout
      printf("%i\n", dato_out);
    }
    else{
      buffer[i] = dato_out; // PRODUCTOR de buffer[i]
    }
  }
}

int main(void){
  int i;
  pthread_t thid[N];

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

  for(i = 0; i < N; i++){
    ret = pthread_create(&thid[i], &attr, &etapa, (void *)i);
  }
  while(1);
}
```

b)

Sección crítica 1 (compite con productor de etapa anterior):

```
   dato_in = buffer[i-1]; // CONSUMIDOR de buffer[i-1]
```

Sección crítica 2 (compite con consumidor de etapa siguiente):

```
   buffer[i] = dato_out; // PRODUCTOR de buffer[i]
```

c) (En negrita el código añadido)

(NOTA: Se han omitido las comprobaciones de retornos de error)

```
pthread_mutex_t mutex[N-1];
pthread_cond_t dato_disponible[N-1], vacio[N-1];
int n_datos[N-1]; //1= buffer[.] tiene dato valido

/* N-1 buffers entre etapas (1 int/buffer):
    Etapa#0 -> buffer[0] -> Etapa#1 -> buffer[1]...
*/
int buffer[N-1];

int realiza_operacion(int Netapa, int dato);

void *etapa(void *p){
   int dato_in, dato_out;
   int i = (int)p; //Restaura i de main

   while(1){
     if(i == 0){//Etapa #0 lee de stdin
       scanf("%i", &dato_in);
     }else{
       pthread_mutex_lock(&mutex[i-1]);
       while(n_datos[i-1] == 0) //ESPERA mientras buffer vacío
         pthread_cond_wait(&dato_disponible[i-1], &mutex[i-1]);
       pthread_mutex_unlock(&mutex[i-1]); //(Para ESPERA PASIVA de PRODUCTOR)
       dato_in = buffer[i-1]; // CONSUMIDOR de buffer[i-1]
       n_datos[i-1] = 0; //Consumidor ha vaciado buffer
       pthread_mutex_lock(&mutex[i-1]); //(Para ESPERA PASIVA de PRODUCTOR)
       pthread_cond_signal(&vacio[i-1]);
       pthread_mutex_unlock(&mutex[i-1]);
     }

     dato_out = realiza_operacion(i, dato_in);

     if(i == N-1){//Etapa #N-1 escribe en stdout
       printf("%i\n", dato_out);
     }
     else{
       pthread_mutex_lock(&mutex[i]);
       while(n_datos[i] == 1) //ESPERA mientras buffer ocupado
         pthread_cond_wait(&vacio[i], &mutex[i]);
       pthread_mutex_unlock(&mutex[i]); //(Para ESPERA PASIVA de CONSUMIDOR)
       buffer[i] = dato_out; // PRODUCTOR de buffer[i]
       n_datos[i] = 1; //Productor ha llenado buffer
       pthread_mutex_lock(&mutex[i]); //(Para ESPERA PASIVA de CONSUMIDOR)
       pthread_cond_signal(&dato_disponible[i]);
       pthread_mutex_unlock(&mutex[i]);
     }
   }
}
```

```
int main(void){
   int i;
   pthread_t thid[N];

   //Inicia estado de N-1 buffers = VACÍOS
   for(i = 0; i < N-1; i++){
      n_datos[i] = 0;
   }

   pthread_attr_t attr;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

   for(i = 0; i < N; i++){
      ret = pthread_create(&thid[i], &attr, &etapa, (void *)i);
   }

   while(1);
}
```

Problema 4.26 (dic 2012)

Se desea diseñar un sencillo dispositivo H que haga de puente entre un conjunto de puertos USB y los computadores de una nube. El hardware del dispositivo H incluye n controladores USB, un puerto Ethernet, un microprocesador, memoria RAM y memoria FLASH para el programa y los parámetros de configuración. El objetivo es poder establecer puertos USB virtuales en las máquinas de la nube, de forma que los programas de estas puedan acceder a los clientes USB como si fuesen locales, con una relación biunívoca entre cada puerto real USB y un puerto virtual. En dicho dispositivo H consideraremos dos aplicaciones: ApA y ApB.

ApA *aplicación encargada de las comunicaciones. Recoge lo que cada cliente USB escribe en el puerto USB físico y lo envía a la máquina con el puerto USB virtual asociado. Inversamente, debe recoger lo que le envíe la máquina de la nube y debe retransmitirlo por el puerto físico.*

ApB *aplicación encargada de asignar puertos USB. Cuando una máquina de la nube quiere acceder a un cliente USB debe contactar con esta aplicación para ver si está libre y para que se le asigne.*

Además, es necesario incluir una aplicación ***ApC*** *en cada uno de los computadores de la nube para que presenten los correspondientes puertos USB virtuales, de forma que se acceda a ellos como si fuesen locales.*

a) Para la aplicación ApA, teniendo especialmente en cuenta las comunicaciones, exponer detalladamente las ventajas e inconvenientes que presentan las siguientes arquitecturas: a1) Un único proceso sin threads, a2) Un único proceso con threads, a3) Varios procesos sin threads.

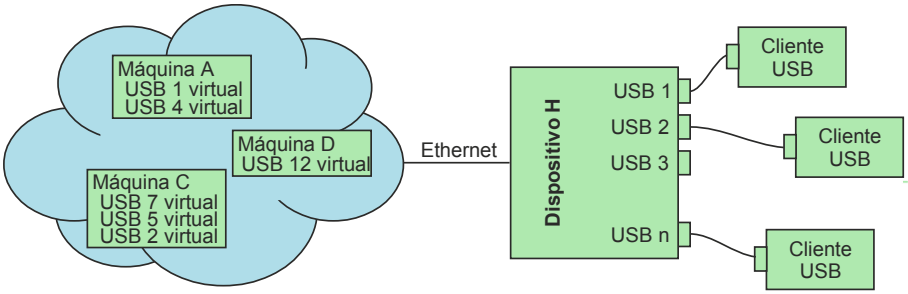


Figura 4.7

b) Para la solución a3, seleccionar el mecanismo de comunicación más adecuado, justificando dicha selección. Indicar las instancias de dicho mecanismo que serían necesarias, explicando el uso de cada de ellas así como su relación con los procesos del dispositivo H.

c) ¿Es previsible que aparezcan problemas de concurrencia en la aplicación ApA? ¿y en la aplicación ApC? Explicar las razones en las que basa su respuesta.

d) ¿Tendría sentido plantear la aplicación ApB como un servicio serie? En este supuesto, ¿qué mecanismos de comunicación se utilizarían entre las aplicaciones ApC y la aplicación ApB? Justifique ambas respuestas.

e) La aplicación ApA detectará que un cliente de la nube ha cerrado el puerto virtual y deberá anotar dicho puerto como libre, planteándose un problema de concurrencia con ApB. Indicar la información que se comparte, su ubicación y el mecanismo de comunicación a utilizar entre ApA y ApB.

f) Plantear una solución al problema anterior, indicando la información de control que se necesita. Justificar el tipo de mecanismo a utilizar, así como las instancias del mismo y las características de cada una de ellas.

g) Codificar el acceso a la información compartida de acuerdo a la solución anterior.

Solución

a) Evidentemente se necesitará una comunicación fiable que garantice que la información que circula entre las máquinas de la nube y los dispositivos USB no se pierda y llegue ordenada. Pora ello nos harán falta sockets del dominio AF_INET, de tipo Stream y protocolo TCP. Es decir, sockets con conexión.

Un único proceso sin threads con servicios bloqueantes no sería una buena solución. El problema de ApA aparece con las lecturas de los sockets o de los puertos USB locales, puesto que se quedaría bloqueado en una de las lecturas, dejando de atender a las demás, hasta que se llegara algo que leer. Es una solución de servidor serie, que solamente atiende a un cliente, dejando bloqueados a los demás. Si usamos servicios no bloqueantes (por ejemplo, select o poll), sí puede ser una buena solución.

Un proceso con threads, siempre que estos sean KLT, permite tener threads dedicados que se quedarán bloqueados en las lecturas, pero sin afectar a los demás, por lo que se puede atender simultáneamente a todos los clientes.

La solución clásica de tener un proceso por puerto presenta la misma ventaja que los threads KLT.

Para construir la aplicación ApA mediante primitivas de comunicación síncronas se necesitar un proceso o thread por cada primitiva de lectura, puesto que dejará bloqueado a dicho proceso o thread. La aplicación ApA, por un lado, tiene que leer del puerto USB local, para reenviar lo leído al puerto virtual USB y, por otro, tiene que leer lo que le llega del puerto virtual USB para re-enviárselo al local. Por lo tanto, la aplicación ApA ha de generar dos procesos o threads por puerto USB, tal y como indica la figura 4.8.

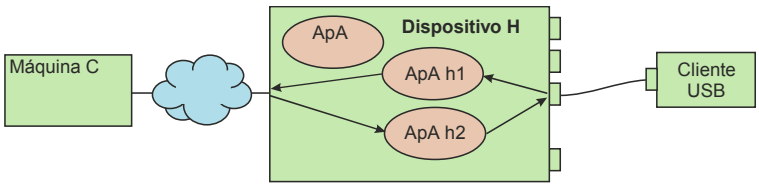


Figura 4.8

El proceso maestro ApA estará esperando peticiones de conexión de las máquinas de la nube. Una vez aceptada una solicitud generará un socket específico para la misma y creará dos procesos. Uno se quedará bloqueado leyendo del socket, mientras que el otro se queda bloqueado leyendo del descriptor de fichero asociado al puerto USB local.

b) Como se ha indicado anteriormente, se necesita un socket del dominio AF_INET, de tipo Stream y protocolo TCP por cada USB virtual que se establezca. Dichos sockets son bidireccionales, por lo que cada uno será atendido por dos procesos en el dispositivo H (uno por cada sentido de la comunicación). Por otro lado, la comunicación con el puerto local se realizará mediante un descriptor de fichero resultado de abrir el dispositivo (/dev) correspondiente.

c) Los procesos derivados del ApA que hacen de puente entre el puerto físico USB y el virtual, no comparten ninguna información entre ellos, por lo que no presentan ningún problema de concurrencia entre ellos. Solamente presentan un problema con la aplicación ApB, como se dice en el apartado e. Igualmente, los procesos ApC no comparten información entre sí, por lo que, aunque tengamos N puertos virtuales en la misma máquina, los N procesos ApC son totalmente independientes sin ningún problema de concurrencia.

d) Teniendo en cuenta que la operación de solicitar una conexión USB no será muy frecuente y que la función que ejecuta ApB es muy simple, pues consiste en comprobar en una tabla si está libre el puerto USB correspondiente, la solución de un servidor serie parece muy adecuada por su simplicidad. Como necesitamos una conexión segura entre ApC y ApB, utilizaremos un único socket del dominio AF_INET, de tipo Stream y protocolo TCP. El proceso ApB estará normalmente bloqueado en el servicio accept.

e) y **f)** La información que se comparte es el estado de ocupado o libre de los puertos USB locales, que llamaremos tabla Ocupa. La comunicación entre ApA y ApB se puede realizar básicamente de dos formas: mediante memoria compartida o mediante un FIFO (o pipe en el caso de que estén emparentados). Mediante memoria compartida sería necesario implementar un mecanismo de sincronización. Podemos utilizar un semáforo para proteger toda la tabla

Ocupa, puesto que las acciones sobre la misma son muy breves e infrecuentes, por lo que no se producirá contención apreciable. La utilización de un FIFO sería, en principio, adecuada, puesto que lo que necesita ApA es informar a ApB que se ha quedado libre un puerto USB, lo que se puede hacer mediante un mensaje. Sin embargo, presenta un problema en ApB, que deberá estar atendiendo simultáneamente las peticiones que le lleguen a través del socket y los mensajes que le lleguen por el FIFO, lo que no se puede hacer usando exclusivamente servicios síncronos.

g) Vamos a suponer que el proceso ApB es hijo del ApA, por lo que puede heredar el semáforo y la región de memoria compartida. Consideraremos que Ocupa dedica un byte en vez de un bit por USB, dado que simplifica la programación y que el número de puertos USB en una misma máquina no puede ser muy grande. Siendo Nusb el número total de puertos USB, los fragmentos de código son los siguientes:

```
//Inicialización
sem_t sem_Ocupa;
char *p;
sem_init (&sem_Ocupa, 1, 1); //Iniciamos a 1
p = mmap(0, Nusb, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON , -1, 0);

//Acceso a  Ocupa por ApA
sem_wait(sem_Ocupa);
p[n] = 0; //marca como libre el USB n
sem_post(sem_Ocupa);


//Acceso a  Ocupa por ApB
sem_wait(sem_Ocupa);
if (p[n] == 0) {
   p[n] = 1; //marca como ocupado el USB n
   sem_post(sem_Ocupa);
   <<contesta asignando el puerto y avisa a ApA para que establezca la conexión>>
}
else {
   sem_post(sem_Ocupa);
<<contesta avisando que el puerto está ocupado>>
}
```

5
PROBLEMAS PROPUESTOS

A.- Tema de introducción

Problema .1

Sea un sistema multitarea sin memoria virtual que tiene una memoria principal de 24 MiB. Conociendo que la parte residente del SO ocupa 5 MiB y que cada proceso ocupa 3 MiB, calcular el grado de multiprogramación que se puede alcanzar.

Problema .2

Un sistema con memoria virtual y páginas de 2 KiB tiene 24 MiB de memoria principal y 300 MiB de zona de swap. El SO residente ocupa 6 MiB y cada proceso requiere 6 MiB. Calcular el grado de multiprogramación que se puede alcanzar. Para este grado de multiprogramación calcular la memoria principal en marcos y en KiB que le corresponde a cada proceso.

Problema .3

En un sistema multiprogramado se van a ejecutar tres procesos, que tienen las características de ejecución de la tabla siguiente, expresada en ms:

Proceso	UCP	E/S	UCP	E/S	UCP
A	5	25	13	27	3
B	2	43	3	36	4
C	68	12	54		

El proceso A empieza su ejecución en el instante t = 0. El proceso B se crea en el instante t = 10 ms y el C en el instante t = 25 ms. Considerando que el SO utiliza 3 ms de UCP cada vez que entra a ejecutar y que el planificador da más prioridad a los procesos más antiguos, hacer un diagrama de tiempos que indique el avance de los procesos.

Problema .4

Repetir el problema anterior considerando que el sistema tiene un reloj que interrumpe cada 10 ms.

Problema .5

Un fichero almacena enteros de 32 bits, desde el 1 hasta el 4G. El planificador del SO va d ando control a los procesos de forma que se produce la siguiente secuencia de operaciones:

- *Proceso A fd = open("/tmp/felipe/datos.txt", O_RDONLY);*
- *Proceso A fork(); Crea el proceso C*
- *Proceso B fd = open("/tmp/felipe/datos.txt", O_RDONLY);*
- *Proceso C lseek(fd, 1000, SEEK_SET); Posiciona desde el origen*
- *Proceso B read(fd, buff, 20);*
- *Proceso A read(fd, buff, 20);*
- *Proceso C read(fd, buff, 20);*

(Considerar que buff se ha definido como un vector de 100 bytes)

Indicar los contenidos de buff después de esta secuencia de operaciones.

Problema .6

Un sistema de ficheros utiliza el esquema de nodos-i para almacenar la estructura de los ficheros y reserva el nodo-i = 2 para el directorio raíz. La información del sistema de ficheros incluye lo siguiente:

Contenido de algunos nodos-i

Nodo-i	Tipo	Dueño (uid/gid)	Protección	Tamaño en bytes	Agrupaciones del fichero
2	directorio	0/0	rwx r–x r–x	160	7644
23	directorio	34/20	rwx —x ——	80	173
87	usuario	65/20	rw– rw– ——	2532	79752, 4739, 119
273	directorio	0/0	rwx r–x —x	120	7635
324	directorio	6/5	rwx r–x r–x	40	23
753	usuario	34/20	rw– rw– ——	856	4546
823	usuario	4/5	rw– rw– ——	3848	12764, 2384, 8763, 6635
876	directorio	0/0	rwx r–x r–x	40	453
2073	usuario	6/5	rw– rw– r—	852	56
3784	usuario	58/20	rw– rw– ——	796	443
8234	usuario	0/0	rw– —— ——	374	4413
9835	usuario	34/20	rw– —— ——	535	569

Contenido de algunas agrupaciones *(el ";" se ha utilizado para separar las líneas de los directorios)*

Agrupación	Contenido de la agrupación
23	udjri 2073
56	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 .....
173	dec 753; Jfuie 9835
443	qwei oier oeiru qoieruower qpoweioru qpwoeir....
453	iue 8234
569	1234567890 ...
4413	abcdefghijklme.....
4546	Esto es un fichero de texto.....
7635	pri1 823; Pro2 87; Pro3 324
7644	doc 273; Proa 876; Prob 23; Proc 3784
12764	ieqo ieoqo eir93o3hnf 'wqn34209e rn349 wheew.....
79752	ABCDEFGHIJKLM.....

Se pide:

a) *Hacer un diagrama con el esquema de directorios.*
b) *Indicar las operaciones que tiene que hacer el sistema de ficheros para interpretar el nombre:*
   /doc/Pro3/udjri
c) *Indicar el contenido del fichero antes mencionado.*

B.- Procesos y llamadas al sistema

Problema .7

Se dispone de un sistema que tiene 128 MiB de memoria principal y 500 MiB de swap. El SO residente ocupa 32 MiB y los procesos ocupan una media de 23 MiB.

Suponiendo que el sistema opera con preasignación de swap y con expulsión, indicar el máximo nivel de multi-programación que se puede alcanzar y la cantidad de memoria principal que se asignará a cada proceso.

Problema .8

Un proceso Unix tiene la tabla de páginas de la figura 5.1, siendo éstas de 1 KiB.

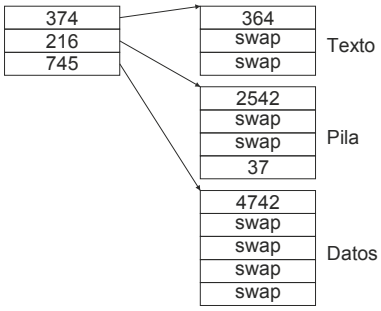


Figura 5.1

Suponer que el proceso solicita 3 KiB de memoria al SO, ¿Cómo quedará la tabla de páginas?

Problema .9

Un sistema operativo tiene 4 niveles de prioridad, siendo el 3 el de menor prioridad, y planifica de acuerdo a las siguientes reglas:

- *Los procesos de cada nivel se planifican en Round-Robin con rodajas de 50 ms de UCP.*
- *Cuando un proceso lleva más de 500 ms en un nivel sin ejecutar es elevado de nivel.*

Suponer que los procesos sólo desean UCP y no E/S, y que se parte de una situación en t = 0 en la cual hay 1 proceso de nivel 0, 2 de nivel 1, 2 de nivel 2 y 1 de nivel 3.

Indicar el tiempo que transcurre hasta que se ejecute por primera vez el proceso de nivel 3.

Problema .10

El planificador de un SO va dando control de forma que se produce la siguiente secuencia de eventos.

- *Proceso A fork (); crea el proceso B*
- *Proceso B fork (); crea el proceso C*
- *Proceso B exit (estado);*
- *Proceso A wait (&sta);*
- *Proceso C exit (status)*
- *Proceso A exit (stat)*

Indicar, mediante una serie de esquemas de Jerarquía de procesos, las distintas fases por las que se pasa al producirse los eventos anteriores.

Problema .11

Un proceso multihebra produce la siguiente secuencia de ejecución:

Hebra principal A	*p_create*	*Crea hebra B*
Hebra B	*p_create*	*Crea hebra C*
Hebra B	*p_create*	*Crea hebra D*
Hebra D	*p_join*	*sobre la hebra B*
Hebra B	*p_create*	*Crea hebra E*
Hebra E	*read (.....)*	

Indicar en este momento cuál es la situación del proceso desde el punto de vista de sus posibilidades de ejecución. (Hebra = proceso ligero)

Problema .12

Un sistema dispone de 10 recursos que llamaremos R0 a R9. Sea la situación siguiente:

- *Proceso A tiene R2 y desea R4*
- *Proceso B tiene R5 y desea R1*
- *Proceso C tiene R3 y R4 y desea R9*
- *Proceso D tiene R9 y desea R2 y R1*

Indicar en que situación se encuentran estos procesos. Verificar que con una política de asignación adecuada se podría haber evitado el problema.

Problema .13

Dos procesos se comunican a través de un fichero, de forma que uno escribe en el fichero y el otro lee del mismo. Para sincronizarse el proceso escritor envía una señal al lector. Proponer un esquema del código de ambos procesos.

Problema .14

Consideremos procesos que tiene un perfil de ejecución repetitivo con 30 ms de UCP seguido por 60 Ms de E/S. Supondremos que se dispone de un sistema operativo para trabajos batch con un nivel de multiprogramación de 3. En el instante t = 0 no hay ningún proceso en memoria y se tiene la cola de procesos pendientes siguiente:

P1 (160), P2 (1000), P3(600), P4 (30), P5 (100) y P6 (60)

donde el número entre paréntesis indica el tiempo de UCP necesario para que ejecute el proceso.

Calcular el tiempo medio de espera para los dos casos de planificación FIFO y planificación según el más corto.

C.- Comunicación y sincronización entre procesos

Problema .15

El siguiente fragmento de código intenta resolver el problema de la sección crítica para dos procesos

```
P0 y P1.
while (turno != i)
;
SECCIÓN CRÍTICA;
turno = j;
```

La variable turno tiene valor inicial 0. La variable i vale 0 en el proceso P0 y 1 en el proceso P1. La variable j vale 1 en el proceso P0 y 0 en el proceso P1. ¿Resuelve este código el problema de la sección crítica?

Problema .16

¿Cómo podría implementarse un semáforo utilizando un pipe?

Problema .17

Considérese un sistema en el que existe un proceso agente y tres procesos fumadores. Cada fumador está continuamente liando un cigarrillo y fumándolo. Para fumar un cigarrillo son necesarios tres ingredientes: papel, tabaco y cerillas. Cada fumador tiene reserva de un solo ingrediente distinto de los otros dos. El agente tiene infinita reserva de los otros tres ingredientes, poniendo cada vez dos de ellos sobre la mesa. De esta forma, el fumador con el tercer ingrediente puede liar el cigarrillo y fumárselo, indicando al agente cuando ha terminado, momento en que el agente pone en la mesa otro par de ingredientes y continua el ciclo.

a) Escribir un programa que sincronice al agente y a los tres fumadores utilizando semáforos.
b) Resolver el mismo problema utilizando mensajes.

D.- E/S y sistema de ficheros

Problema .18

Dado el siguiente programa:

```
int fd;
pid_t pid;

fd = creat("fichero", 0755);
pid = fork();
if (pid == 0)
{
   write(fd, "Soy yo\n", 7);
   lseek(fd, 1024, SEEK_CUR);
   exit(1);
}
else
{
   write(fd, "Soy otro\n", 10);
   write(fd, "Adios\n", 6);
```

```
    exit(1);
}
```

Conteste a las siguientes preguntas:

a) ¿Cuál sería el tamaño final del fichero? ¿Sería distinto si la operación lseek() se realizara en último lugar?
b) ¿Cuál será el contenido final del fichero?

Problema .19

Escribir un programa que implemente el mandato tee. Este mandato escribe lo que lee por la entrada estándar en su salida estándar y en un fichero especificado como argumento.

Problema .20

*Un usuario con uid=12 y gid=1 es el dueño de un fichero con modo de protección **rwxr-x---**. Otro usuario con uid=3 y gid=1 quiere ejecutar el fichero. ¿Puede hacerlo?*

Problema .21

¿Cuántos nodos-i estarán ocupados en un sistema de ficheros UNIX que contiene únicamente los siguientes ficheros: "/f1", "/f2" (enlace simbólico a "/f1"), "/f3" (enlace no simbólico a "/f1"), y "/dir" que es un directorio vacío?

Problema .22

Considérese la siguiente política de backup de los ficheros contenidos en un disco: el primer lunes de cada mes se realiza una copia de seguridad completa, el resto de lunes se realiza una copia de seguridad incremental con respecto a la última copia completa, y el resto de los días se hace una copia incremental con respecto a la última copia incremental. Si se rompe el disco el jueves de la tercera semana, ¿qué pasos habría que seguirse para restaurar el sistema?

Problema .23

Considere el siguiente fragmento de programa

```
int fd;
char buf[512];
fd1 = open("fichero1", O_RDONLY);
fd2 = creat("fichero2", 0755);
while ((n_bytes = read(fd1, buf, 512)) > 0)
   write(fd2, buf, n_bytes);
```

Suponiendo que el fichero fichero1 tiene un tamaño de 512 KiB y que el sistema de ficheros emplea un tamaño de bloque de 4 KiB, se pide:

a) ¿Cuántos accesos a disco se realizarían durante la ejecución del bucle anterior en un sistema que no emplea cache de bloques?
b) ¿Cuántos accesos a disco se realizarían durante la ejecución del bucle anterior en un sistema que emplea un cache de 2 MiB y una política de escritura inmediata?