Cuestión 1 (1 punto)
Razone sobre la decodificación de la ruta indicada, durante la realización de la siguiente llamada, entendiendo que esta termina correctamente y suponiendo que las caches están inicialmente vacías.

```
open("/home/alumno/fichero.txt", O_RDONLY);
```

a) ¿Cuál sería el número mínimo de accesos a disco necesario?
b) ¿Cuál sería el número máximo de accesos a disco necesario?
c) Explique en qué circunstancias estos números pueden no coincidir aun siendo la misma ruta.
d) Indique qué permisos habrá de ir verificando durante la decodificación de esta ruta.

Solución

a) y d) Durante la decodificación de la ruta `"/home/alumno/fichero.txt"`, suponiendo caches vacias, el número mínimo de accesos a disco necesario sería:

- Acceso al inodo raiz, comprobación de que es directorio y verificación del permiso x.
- Busqueda en el directorio `/` de la entrada `home` para conocer su número de inodo.
- Acceso al inodo de `/home`, comprobación de directorio y de permiso x.
- Busqueda en `/home/` de la entrada `alumno` para conocer su número de inodo.
- Acceso al inodo de `/home/alumno`, comprobación de directorio y de permiso x.
- Busqueda en `/home/alumno/` de `fichero.txt` para conocer su número de inodo.
- Acceso al inodo de `/home/alumno/fichero.txt`, es archivo y tenemos permiso r.

Son un total de 7 accesos a disco como mínimo. Validandose el permiso de paso (x) en cada directorio y de lectura (r) en el objeto pues se quiere abrir para lectura.

b) y c) Esta ruta aparentemente tan sencilla, podría en realidad contener enlaces simbólicos y pasar por puntos de montaje (todo ello en cualquier cantidad) que habría que ir recorriendo durante su decodificación, de cara a intentar abrir el objeto indicado.
En estas cinrcunstancias el número máximo de accesos necesario no está acotado, aunque el sistema internamete impone un máximo de pasos en la decodificación de cualquier ruta para evitar caer en un bucle que podría dejar "colgado" el sistema.

Ejercicio 1 (1 punto)

Implemente en C y con llamadas al sistema Unix, el programa cuya ejecución equivale a la siguiente secuencia de mandatos:

```
$  ls -laR /  |  sort -u  2> /dev/null
```

Solución

```
int main(void) {
      int ret, pp[2];
      ret=creat("/dev/null",0666); if(ret==-1) { perror("2>"); exit(1); }
      dup2(ret,2); close(ret);
      ret=pipe(pp); if(ret==-1) { perror("pipe"); exit(1); }
      switch(fork()) {
      case -1: perror("fork"); exit(1);
      case  0: dup2(pp[1],1); close(pp[0]); close(pp[1]);
            execlp("ls","ls","-laR","/",NULL);
            perror("ls"); exit(1);
      default: dup2(pp[0],0); close(pp[0]); close(pp[1]);
            execlp("sort","sort","-u",NULL);
            perror("sort"); exit(1);
      }
      return 0;
}
```

Ejercicio 2 (2 puntos)
Se desea implementar la función `int Sustituir(int fd,char viejo,char nuevo);` que, en archivo indicado por el descriptor `fd`, sustituye todas las apariciones del carácter `viejo` por el carácter `nuevo`. Caben dos implementaciones posibles, que deberá desarrollar.

a) **(1 punto)** Impleméntela usando los servicios de manejo de ficheros (`read`, `write`, etc.).
b) **(1 punto)** Impleméntela proyectando el archivo en memoria (sin utilizar `read` ni `write`).

Solución

```
int memsust(void*ptr,int len,char viejo,char nuevo) {
      int pos, cnt=0;
      for (pos=0;pos<len;pos++)
            if (ptr[pos]==viejo) { ptr[pos]=nuevo; cnt++; }
      return cnt;
}
#define SIZE 1024
int Sustituir_a(int fd,char viejo,char nuevo) {
      char buff[SIZE];
      int ret,cnt,ttl=0;
      while((ret=read(fd,buff,SIZE))>0) {
            cnt=memsust(buff,ret,viejo,nuevo);
            if (cnt>0) {
                  lseek(fd,-ret,SEEK_CUR);
                  write(fd,buff,ret);
                  ttl+=cnt;
            }
      }
      return ttl;
}
int Sustituir_b(int fd,char viejo,char nuevo) {
      void*ptr;
      int size,cnt;
      size=lseek(fd,0,SEEK_END);
      ptr=mmap(0,size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
      cnt=memsust(ptr,size,viejo,nuevo);
      munmap(ptr,size);
      return cnt;
}
```

Ejercicio 3 (3 puntos)
Unos grandes almacenes van a cambiar su sistema de gestión de ventas, pasando de un sistema centralizado en la sede principal de Madrid a un sistema distribuido en tres ciudades: Madrid, Barcelona y Valencia. El sistema está compuesto por tres subsistemas:

El primer subsistema, denominado Subsist_Tienda, se encuentra localizado en cada tienda y permite realizar la contabilidad, estadísticas y predicciones de ventas de la misma. Este subsistema se va a instalar para cada tienda en una única máquina. Hay varias tiendas en cada ciudad.

El segundo subsistema, denominado Subsist_Ciudad, es el que recolecta toda la contabilidad, estadísticas y predicciones de ventas de una misma ciudad. Cada Subsist_Tienda se comunica con cierta periodicidad con el sistema Subsist_Ciudad, para enviarle la información correspondiente a dicha tienda.

El tercer subsistema, denominado Subsist_Central, es único y se encuentra localizado en Madrid. Cada subsistema Subsist_Ciudad de Madrid, Barcelona y Valencia se comunica con Subsist_Central con cierta periodicidad para enviarle información sobre dicha ciudad, de manera que Subsist_Central pueda realizar la contabilidad, estadísticas y predicciones de ventas globales.

a) **[0.5 ptos]** Dibujar a nivel de proceso y máquinas todos los subsistemas, indicando los mecanismos de comunicación (su tipo) entre los diferentes subsistemas.

b) **[0.5 ptos]** La aplicación Subsist_Tienda se implementa como un conjunto de threads que comparten acceso a una variable que constituye el estado interno del subsistema. Los threads son de los siguientes tipos: `thread_cont`, que procesan la contabilidad de la tienda modificando el estado, `thread_est`, que procesan las estadísticas de ventas consultando el estado y `thread_pred`, que realizan predicciones de venta consultando el estado. Se requiere que el estado accedido sea siempre coherente tanto a la hora de consultarlo como de modificarlo.

   En este contexto, definir las variables de control necesarias así como los mecanismos de sincronización que necesitamos, justificando la selección.

c) **[0.5 ptos]** Suponga el siguiente código para implementar la entrada y salida de la sección crítica de los thread de tipo `thread_cont`. Indicar si la implementación es correcta y/o si maximiza la concurrencia. En caso contrario, proponer una solución alternativa.

```
pthread_mutex_lock(&mutex);
      <<Acceso en modo escritura al estado compartido>>
pthread_mutex_unlock(&mutex);
```

d) **[0.5 ptos]** Análogamente suponga el siguiente código para los threads de tipo `thread_est` y `thread_pred`. Indicar si la implementación es correcta y/o si maximiza la concurrencia. En caso contrario, proponer una solución alternativa.

```
pthread_mutex_lock(&mutex);
      <<Acceso en modo lectura al estado compartido>>
pthread_mutex_unlock(&mutex);
```

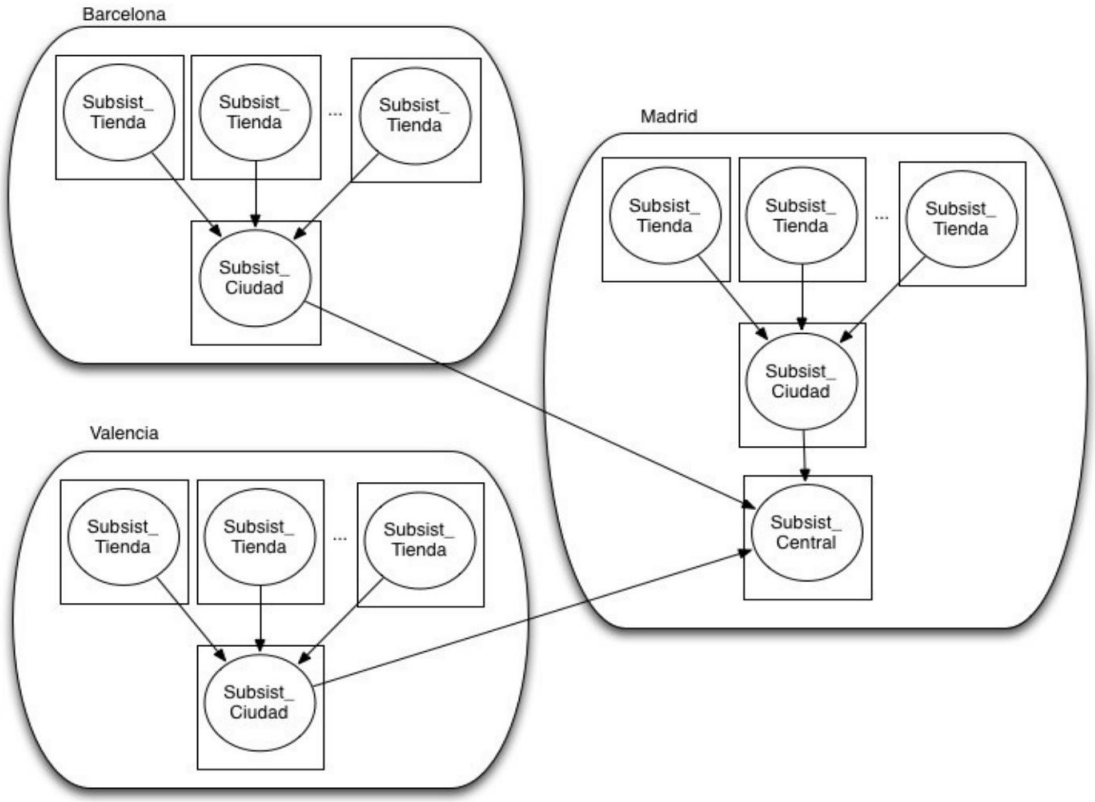
e) **[0.5 ptos]** Si se pretende intercambiar el siguiente tipo de datos entre Subsist_Tienda y Subsist_Ciudad y entre Subsist_Ciudad y Subsist_Central, indicar qué problema existe y cómo se debe resolver.

```
struct info_ventas {
      int contabilidad;
      int estadistica;
      int prediccion;
}
```

f) **[0.5 ptos]** Suponiendo que la periodicidad con la que Subsist_Tienda envía información a Subsist_Ciudad es 1 hora, implementar el código necesario que habría que incorporar a Subsist_Tienda para controlar esta periodicidad.

Solución

a) Dibujo de los subsistemas a nivel de máquinas, procesos y comunicaciones.



b) Tenemos que definir un conjunto de variables que nos permitan saber cuántos threads de cada tipo están accediendo a la variable que constituye el estado del subsistema. Las restricciones que tenemos es que mientras los threads `thread_cont` tienen que acceder de manera exclusiva, los threads `thread_est` y `thread_pred` pueden acceder concurrentemente. Por tanto, definiremos tres variables de tipo int, del siguiente modo, iniciándolas a 0:

```
int num_cont = 0; // número de threads de contabilidad accediendo al estado
int num_est  = 0; // número de threads de estadísticas accediendo al estado
int num_pred = 0; // número de threads de predicciones accediendo al estado
```

Dado que se trata de threads que se pueden comunicar a través de variables globales, el mecanismo más apropiado será el de mútex + condiciones. En concreto, necesitamos una variable de tipo mutex y dos condiciones, una que dé acceso de lectura a la variable que constituye el estado y otra que dé acceso de escritura a dicha variable:

```
pthread_mutex_t mutex; // Exclusión mutua en acceso a las variables de control
pthread_cond_t a_leer;     // Condición de acceso de lectura
pthread_cond_t a_escribir; // Condición de acceso de escritura
```

c) La solución es correcta, dado que no ocasiona condiciones de carrera en el acceso al estado compartido. Sin embargo, no es la solución más apropiada, desde el punto de vista de maximizar la concurrencia. Para ello, usaremos una solución lector-escritor, implementando la parte de escritura con los mecanismos definidos en el apartado anterior:

```
pthread_mutex_lock(&mutex)
while ((num_est > 0) || (num_pred>0))
     pthread_cond_wait(&a_escribir, &mutex);
num_cont++;
pthread_mutex_unlock(&mutex);

<< Acceso en modo escritura al estado compartido >>
```

```
pthread_mutex_lock(&mutex);
num_cont--;
pthread_cond_signal(&a_escribir);
pthread_cond_broadcast(&a_leer);
pthread_mutex_unlock(&mutex);
```

d) Al igual que en el caso anterior, la solución es correcta, dado que no ocasiona condiciones de carrera en el acceso al estado compartido. Sin embargo, no es la solución más apropiada, desde el punto de vista de maximizar la concurrencia. Para ello, usaremos una solución lector-escritor, implementando la parte de lectura con los mecanismos definidos anteriormente (se pone como ejemplo el thread de estadísticas, pero el de predicciones sería similar y está comentado en el código):

```
pthread_mutex_lock(&mutex);
while (num_cont!=0)
     pthread_cond_wait(&a_leer, &mutex);
num_est++; // o num_pred++ si fuera el thread de predicción
pthread_mutex_unlock(&mutex);

<< Acceso en modo lectura al estado compartido >>

pthread_mutex_lock(&mutex);
num_est--; // o num_pred-- si fuera el thread de predicción
if ( (num_est == 0) && (num_pred ==0))
     pthread_cond_signal(&a_escribir);
pthread_mutex_unlock(&mutex);
```

e) Podría haber un problema a la hora de enviar valores enteros si las arquitecturas de los sistemas son diferentes. Para resolverlo, utilizaremos las funciones `htonl()` y `ntohl()`.

f) Se podría resolver programando una alarma que venza en 1 hora (3600 segundos), o bien, si el proceso no hace nada hasta que pase la hora, bastaría con utilizar un sleep de 3600 segundos, del siguiente modo:

```
sleep(3600);
/** mandar información a Subsist_Ciudad **/
```

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos – GMI y GII

Examen Extraordinario. 7 de julio de 2017

Dispone de 2 horas. Las notas saldrán el 11 de julio. La revisión será el 13 a las 16:00 en la sala de cristal S4200.

C1	C2	C3	P1a	P1b	P1c	C4	P2a	P2b	P2c

Cuestiones (conteste las cuestiones 1, 2 y 3 y el problema 1 en el espacio reservado al efecto en esta misma hoja y la cuestión 4 y el problema 2 en la otra hoja)

C1) [0,5 ptos] Describa los posibles situaciones que pueden darse ante la finalización de un proceso hijo con pid 4444 en función de si el padre con pid 333 termina antes que él. Suponga la siguiente jerarquía de procesos: 1-22-333-4444-55555

El proceso hijo 4444 puede quedar huérfano si el padre 333 ha invocado el servicio wait antes de su finalización. En este caso, el hijo es heredado por el proceso init 1 que espera en un bucle infinito la terminación de sus hijos.
También puede quedar como zombi si termina y el proceso padre 333 no ha hecho un wait. En este caso, no puede entregar el estado de terminación al padre y no se libera su entrada en la tabla de procesos.

C2) [0,5 ptos] ¿Qué diferencia hay al utilizar el flag RTL_NOW o el RTL_LAZY en la invocación de la función dlopen?

Al invocar la función dlopen se está solicitando el montaje dinámico explícito de la biblioteca que se pasa como primer argumento. Los flags permiten diferenciar el momento en el que se carga la biblioteca en memoria. En el caso del RTL_NOW se resuelven todos los símbolos de la biblioteca en el momento de la invocación de la función y se cargan en memoria. En el caso de RTL_LAZY, solo se resuelven las variables y respecto a las referencias a las funciones, se pospone su resolución hasta que no sea estrictamente necesario. Es decir, cuando se utilizan en el código. Si no se llegan a utilizar nunca en el código, estas referencias no llegan a resolverse nunca.

C3) [0,5 ptos] Enumere señales que al recibirlas un proceso, su acción por defecto no provoca su muerte.

SIGCHLD, SIGSTOP, SIGCONT

Problema 1 (3,5 puntos)

a) (1 punto) En un sistema UNIX que dispone de memoria virtual con páginas de 8 KiB, indique cómo pueden compartir una región de memoria un proceso padre con sus hijos sin que exista soporte físico de esta región en el sistema de ficheros. El tamaño solicitado para la región será de 15 KB. Especifique la invocación del servicio que permite crear esta región e indique cuál será el tamaño de la región tras su creación.

Antes de la invocación del servicio fork para crear los hijos, el proceso padre crea una región en memoria con el servicio mmap activando el flag MAP_SHARED, que permite compartir la región entre varios procesos, y utilizando el flag MAP_ANONYMOUS, que no asocia la región a ningún fichero en disco:

```
char *p;
p=mmap(NULL,15000,PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS,-1,0);
```

Como el tamaño de la región debe ser múltiplo de la página del sistema, la solicitud de 15 KB se convertirá en una región de 16 KiB.

b) (1,5 puntos) Un usuario invoca desde el terminal el mandato `programaA`. Este proceso crea un hijo. El proceso padre envía al hijo la señal SIGUSR1 transcurridos 3 segundos desde el inicio de la ejecución del proceso padre y el proceso hijo imprime un mensaje por la salida estándar indicando que se ha producido la recepción de la señal. El proceso hijo debe ejecutar el mandato "`ls -alt /tmp`" una vez que han pasado 10 segundos desde su creación. Al concluir la ejecución del proceso hijo tras la ejecución del mandato `ls`, también debe mostrarse al usuario en el terminal los identificadores de ambos procesos. Especifique el código necesario para que los procesos completen la ejecución correctamente y que ambos siempre devuelvan el estado en el que terminan su ejecución.

```
pid_t pid;

void funcion1(int senal){
        if (senal == SIGUSR1)
           printf("Se ha recibido la Senal SIGUSR1\n");
        else if (senal == SIGALRM)
        {
           printf("Han transcurrido los 10 segundos\n");
           execlp("ls","ls","-alt","/tmp",NULL);
           perror("No se ha podido ejecutar el exec");
           exit(1);
        }
}

void funcion2(int senal){
        aux=kill(pid,SIGUSR1);
        if (aux == -1)
             perror("kill SIGCONF");
}

int main(void)
{
        int status; struct sigaction act;

        act.sa_handler=&funcion2;
        act.sa_flags=SA_RESTART;
        sigaction(SIGALRM,&act,NULL);
        alarm(3);
        pid=fork();
        if (pid == -1)
             {perror("Error en fork"); exit(1);}
        else if (pid == 0){
                    act.sa_handler=&funcion1;
                    act.sa_flags=SA_RESTART;
                    sigaction(SIGUSR1,&act,NULL);
                    sigaction(SIGALRM,&act,NULL);
                    alarm(10);
                    while (1) {}
        }
        else {
             pid=wait(&status);
             if (pid>0)
                    printf("Fin del hijo: %d\n",pid);
             else
                    perror("Error en el wait");
             printf("Fin del padre: %d\n",getpid());
        }
        return 0;
}
```

c) (1 punto) Indique si sería posible resolver el apartado b) utilizando una solución basada en threads en lugar de procesos pesados y en caso afirmativo especifique el código.

No es posible porque al invocar el servicio exec desde algún thread del proceso se sustituye su imagen de memoria por la del nuevo mandato que se pasa como argumento, en este caso el mandato ls. Por lo tanto, los mensajes con los identificadores de threads o del proceso que debían mostrarse al usuario una vez concluida la ejecución del mandato ls no podrían mostrarse en este caso.

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos – GMI y GII

Examen Extraordinario. 7 de julio de 2017

C1	C2	C3	P1a	P1b	P1c	C4	P2a	P2b	P2c

					Tabla intermedia de punteros					Tabla de Nodos-i			
Paso núm.	BCP A		BCP B		ID Interm.	Núm Nodo-i	Posic.	Núm de Refs	Modo R/W	Núm Nodo-i	nopens	Agrupaciones.	Tam
1	0	1	0	3	**7**	**42**	**0**	**1**	**11**	42	**1**	184-186	2500
	1	2	1	4									
	2	2	2	4									
	3	**7**											
2	0	1	0	3	7	42	0	1	11	42	**2**	184-186	2500
	1	2	1	4	**8**	**42**	**0**	**1**	**11**				
	2	2	2	4									
	3	7	**3**	**8**									
3	0	1	0	3	7	42	**1800**	1	11	42	2	184-186	2500
	1	2	1	4	8	42	0	1	11				
	2	2	2	4									
	3	7	3	8									
4	0	1	0	3	7	42	1800	1	11	42	2	184-186	2500
	1	2	1	4	8	42	**2200**	1	11				
	2	2	2	4									
	3	7	3	8									
5	0	1	0	3	7	42	**2700**	1	11	42	2	184-186	**2700**
	1	2	1	4	8	42	2200	1	11				
	2	2	2	4									
	3	7	3	8									
6	0	1	0	3	7	42	2700	1	11	42	2	**184-187**	**3100**
	1	2	1	4	8	42	**3100**	1	11				
	2	2	2	4									
	3	7	3	8									
7	0	1	0	3	7	42	3100	**0**	11	42	**1**	184-187	3100
	1	2	1	4	8	42	3100	1	11				
	2	2	2	4									
	3	**0**	3	8									
8	0	1	0	3	7	42	3100	0	11	42	**0**	184-187	3100
	1	2	1	4	8	42	3100	**0**	11				
	2	2	2	4									
	3	0	3	**0**									

C4) [1,5 ptos] En un servicio de edición de documentos online, supóngase el caso en el que múltiples usuarios quieren leer y editar un documento. Discutir qué modelo/arquitectura de comunicaciones sería más conveniente usar para el escenario propuesto.

La edición/visualización de un documento 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, por lo que la solución centralizada cliente-servidor no plantea problemas de congestión.

Problema 2 (3,5 puntos)

a) (0,5 puntos) Suponiendo que se escoge un modelo de comunicación cliente y servidor serie en el servicio de edición de documentos online mencionado anteriormente, escribir el código necesario para la función de lectura de un trozo del fichero que se ejecuta en el servidor. La función tiene la siguiente cabecera:

```
int leerParteDeFichero( int fd, off_t offset, size_t tam_a_leer,char* parte_leida );
```

Dicha función deberá ser capaz de detectar si se ha podido leer el número de bytes solicitados, devolviendo un código de error en caso de que no haya sido posible.

```
int leerParteDeFichero( int fd, off_t offset, size_t tam_a_leer,
                        char* parte_leida )
{
  int n;
  lseek(fd, offset, SEEK_SET);
  n = read(fd, parte_leida, tam_a_leer);

  if ( n != tam_a_escribir )
    return -1;
  return 0;
}
```

b) (1,5 puntos) Suponiendo que se escoge un modelo de comunicación cliente y servidor dedicado, escribir el código necesario para la función de escritura de un trozo del fichero usando como mecanismo de sincronización cerrojos de ficheros. La función, que se ejecuta en el servidor, tiene la siguiente cabecera:

```
int escribirParteDeFichero(int fd,off_t desp,size_t tam_a_escribir,char*parte_a_escribir);
```

Como en el caso anterior, la función debe ser capaz de detectar si no se ha podido escribir todo el contenido del buffer, y devolver un código de error en caso contrario.
Nota: No será tendrán en cuenta posibles sobre-escrituras entre un cliente y otro

```
int escribirParteDeFichero( int fd, off_t offset,
                            size_t tam_a_escribir, char* parte_a_escribir )
{
  int n;
  struct flock fl;
  fl.l_whence = SEEK_SET;
  fl.l_start = offset;
  fl.l_len = tam_a_escribir;
  fl.l_pid = getpid();
  fl.l_type = F_WRLCK;
  fcntl(fd, F_SETLKW , &fl);
  lseek(fd, offset, SEEK_SET);
  n = write(fd, parte_a_escribir, tam_a_escribir);
  fl.l_type = F_UNLCK;
  fcntl(fd, F_SETLK, &fl);

  if ( n != tam_a_escribir )
    return -1;

  return 0;
}
```

c) (1,5 puntos) Dado el siguiente escenario, donde dos clientes A y B acceden a un fichero:

- Fichero que se va a manipular de nombre: “/tmp/fich_compartido.txt”
- Bloque de 1 KiB y agrupaciones de 1 bloque
- Número de inodo donde se almacena la información: 42
- Agrupaciones ocupadas: del 2 al 183
- Entradas libres de la tabla intermedia: a partir de la 7
- Tamaño inicial del fichero: 2500 bytes

Rellenar las tablas de descriptores de ficheros de los procesos, la tabla de nodos-i en memoria y la tabla intermedia de punteros (identificadores intermedios) después de ejecutar cada una de las siguientes llamadas a las funciones anteriormente descritas y abrir / cerrar el fichero en cada uno de los servidores dedicados a cada cliente. La información debe rellenarse en la tabla del dorso de esta hoja, actualizando la información según se van ejecutando los pasos (indicando en la primera columna el número de paso correspondiente a la actualización de valores).

Paso 1. [Proceso A] Abrir fichero
Paso 2. [Proceso B] Abrir fichero
Paso 3. [Proceso A] leerParteDeFichero(fd, 1500, 300, buff1A);
Paso 4. [Proceso B] leerParteDeFichero(fd, 1800, 400, buff1B);
Paso 5. [Proceso A] escribirParteDeFichero(fd, 2400, 300, buff2A);
Paso 6. [Proceso B] escribirParteDeFichero(fd, 2700, 400, buff2B);
Paso 7. [Proceso A] Cerrar fichero
Paso 8. [Proceso B] Cerrar fichero

Asumiendo que todos los *buffers* de lectura y escritura tienen el tamaño necesario para almacenar la información leída/escrita. Se debe proporcionar también el código necesario para la apertura del fichero antes de la llamada a ***leerParteDeFichero*** y para el cierre del mismo después de la llamada a ***escribirParteDeFichero***, en cada proceso.

Tras la apertura del fichero por ambos procesos, se trae a memoria la información del fichero, se rellenan sendas entradas en la tabla intermedia de punteros (con los permisos necesarios para poder leer y escribir usando el mismo descriptor de fichero) y se asocia el primer descriptor libre de cada proceso con el correspondiente identificador intermedio. (Pasos 1 y 2)

Después de las dos operaciones de lectura, se modifica la posición del puntero en ambas entradas de la tabla intermedia de punteros. (Pasos 3 y 4)

Con la escritura por parte del proceso A, el puntero puntero se desplaza hasta la posición 2700, haciendo crecer al fichero. (Paso 5)

Con la escritura por parte del proceso B, el puntero se desplaza hasta la posición 3100, haciendo crecer al fichero e incrementando el número de agrupaciones asignadas (asumimos que la siguiente asignación está disponible. (Paso 6)

Finalmente, tras cerrar ambos ficheros, se marcan los descriptores de fichero de cada proceso como disponibles y se invalidan ambas entradas de la tabla intermedia de punteros poniendo a cero el número de referencias. (Pasos 7 y 8)

Sistemas Operativos 5º semestre. Grado II

Primer Parcial. Sistema de Ficheros. 6 de Noviembre de 2017.

SOLUCIÓN

Ejercicio 1 (2 puntos)

Realice un dibujo que muestre claramente las estructuras de datos donde el Sistema Operativo almacena los siguientes campos relativos a un fichero abierto. El dibujo debe ser consecuente con el valor indicado para cada campo. Añada las explicaciones necesarias.

- **a)** Descriptor de fichero. Valor: salida estándar.
- **b)** Número de inodo de dicho fichero. Valor: 12345.
- **c)** Número de duplicados o referencias (`ndups`). Valor: 3.
- **d)** Posición sobre el fichero. Valor: 9876.
- **e)** Modo de apertura. Valor: `O_WRITE|O_APPEND`.
- **f)** Número de veces abierto (`nopens`). Valor: 2.
- **g)** Número de enlaces o de nombres (`nlinks`). Valor: 2.
- **h)** Tipo de objeto: Valor: fichero regular.
- **i)** Propietario y Grupo. Valores: 0 y 0.
- **j)** Permisos de acceso. Valor: 0751 con SETUID activo.

Solución

Estructuras de datos:

Tabla de Procesos: (única en el sistema, con un BCP por cada proceso)
BCP del proceso: (con todos los atributos de ese proceso, y entre ellos la T. fds)
Tabla de descriptores de fichero:
[1] (salida estándar): apuntando a la entrada XX de la Tabla Intermedia.

Tabla Intermedia: (única, con una entrada por cada apertura de fichero realizada)
Campos: `[[ #inodo | ndups | posición | modo_open ]]`
[XX]: `[[ 12345 | 3 | 9876 | O_WRITE OAPPEND ]]`

Nota: ndups=3 indica que hay otros dos descriptores de fichero (duplicados) más (de ese proceso o de otros) apuntando a la entrada XX de la T. Intermedia.

Tabla de inodos: (única, conteniendo o referenciando a los inodos en uso)
Campos: `[[ #inodo | nopens | <<copia_del_inodo>> ]]`
[12345]: `[[ 12345 | 2 | <<mostrado aparte>> ]]`

Nota: nopens=2 indica que hay otra entrada más de la T.Intermedia apuntado a este inodo, correspondiente a una apertura independiente del mismo archivo.

Inodo 12345: (copia única de este inodo en memoria, llevado a memoria mientras se use)
Campos: `[[ <<permisos>> | <<propietario>> | nlinks | resto... ]]`
`[[ -,rwsr-x—x | (UID=0, GID=0) | 2 | ... ]]`

Nota: Los permisos contienen el tipo de objeto y los bists SETUID y SETGID, y UID y GID especifican la identidad del propietario del objeto (root en este caso).
Nota: nlinks=2 indica que hay dos entradas de directorio (o nombres de archivo o enlaces físicos) asociados a este número de inodo.

Ejercicio 2 (3 puntos)

Considere un sistema de ficheros tipo UNIX (basado en inodos) para un disco de 2 TiB, con sectores de tamaño estándar, bloques de 2 KiB y agrupaciones de 2 bloques. Explique con detalle:

- **a)** ¿A qué unidad "apuntan" los punteros directos e indirectos de un inodo? ¿Qué ancho deben tener estos punteros/direcciones en este sistema? ¿Por qué?
- **b)** ¿El mapa de bits para gestión de espacio libre utiliza un bit por cada...? (nombre la unidad) ¿Cuánto espacio de disco (en agrupaciones) ocupará este mapa de bits? Haga los cálculos.
- **c)** ¿Cuánto espacio de disco ocupan los datos de ficheros de: 0, 1, 3 y 5 KiB respectivamente?
- **d)** El otro mapa de bits ¿cuál es? ¿para qué sirve? ¿de qué decisión/es depende su tamaño?
- **e)** Exprese con una fórmula la capacidad máxima de direccionamiento (en bytes) del inodo.
- **f)** ¿Qué accesos a disco se necesitan para leer el byte 409600000 de un archivo recién abierto?

Solución

a) Las direcciones del Sistema de Ficheros apuntan a espacio asignado, esto es, a agrupaciones. Los punteros directos apuntan a agrupaciones de datos y los indirectos a agrupaciones de indirección. Hay que calcular cuantas agrupaciones tiene este SF:

2^41(B/disco) / 2^12(B/agrup) = 2^29(agrup/disco)

Luego necesitaríamos 29 bits, que redondeamos a potencia de 2 (32 bits) para definir el ancho necesario para los campos de dirección (o punteros).

b) El mapa de bits para gestión de espacio libre utiliza un bit por cada agrupación tenga este SF. Este mapa almacena 2^29(bits), que son 2^26(B), que a 2^12(B/agrup) son 2^14(agrup).

c) El espacio asignado para los datos de los ficheros será un número entero de agrupaciones. Para ficheros de 0, 1, 3 y 5 KiB, serán 0, 1, 1 y 2 agrupaciones respectivamente.

d) El otro mapa de bits es el de inodos, y se utiliza para saber el estado de libre u ocupado de cada inodo. Su tamaño depende del número máximo de inodos que vayamos a querer tener en este SF, y esto a su vez suele calcularse como el número de archivos de tamaño medio que entrarían en este SF, o lo que es igual, tamaño del disco entre tamaño medio del archivo.

e) El número máximo de agrupaciones que puede llegar a direccionar un inodo, es función del número de punteros de cada nivel de indirección que tenga y del número de punteros por agrupación de indirección, según la siguiente fórmula: $CMD(agrup) = \#pd + \#psi*dpa + \#pdi*dpa^2 + \#pti*dpa^3$

donde los punteros por agrupación son: dpa = 2^12(B/agrup) / 2^2(B/direc) = 2^10(direc/agrup)

normalmente hablamos de 10 punteros directos y tres niveles, #pd = 10 y #psi = #pdi = #pti = 1

y finalmente habría que pasarlo a bytes, CMD(B) = CMD(agrup)*Tam.Agrp.(B)

f) Se trata de acceder directamente a un byte, un acceso directo que de ningún modo precisa el acceso secuencial a toda la información anterior. Considerando que las numeraciones empiezan desde cero, el byte 409600000 será el primero de la agrupación 100000 del fichero. Con los 10 punteros directos no llega (10 < 100000), ni con el simple indirecto (1024 < (100000-10)), pero con el doble indirecto podremos acceder (2^20 > (100000-10-1024)).

Luego, considerando que el inodo ya se encuentra en memoria, necesitaremos 3 accesos a disco. Usamos el puntero doble indirecto del inodo y accedemos a una agrupación de indirección con punteros de tipo simple indirecto. Se calcula la entrada adecuada (la 96) y accedemos a una agrupación de indirección con punteros directos. Se calcula la entrada adecuada (la 662) y accedemos a la agrupación de datos que contendrá el byte 409600000 del fichero.

Ejercicio 3 (5 puntos)

a) Implemente en C y para UNIX el mandato `cat01` (equivalente a un `cat` sin argumentos) que emite por la salida estándar todo lo que lee de su entrada estándar, usando un buffer de 1024 bytes.

b) Ponga y explique un ejemplo de uso del mandato `cat01` desde el intérprete de mandatos, que fuerce la situación en la que el proceso correspondiente a este mandato recibiría la señal SIGPIPE.

Considere ahora el bucle interno de copia de `cat01` y describa cómo será su avance según sea la naturaleza del objeto asociado a su entrada estándar, indicando:

- ¿qué valores devolverá cada llamada `read`?
- ¿cuál será el total de bytes retransmitidos?
- ¿cuánto durará (aproximadamente) la ejecución de `cat01`?

Todo ello para cada uno de los tres casos siguientes donde el objeto asociado a la entrada estándar es:

c) Un archivo de 3500 bytes.

d) Un terminal por el que un usuario introduce los 7 días de la semana (lunes, martes, etc.), uno por línea y uno cada minuto.

e) Una tubería por la que llega una línea de texto cada segundo: la primera vacía, la segunda con una letra, la tercera con dos, y así sucesivamente, hasta 1000 líneas.

Solución

a)

```
int main(void) {
     int cnt;
     char buf[1024];
     while((cnt=read(0,buf,1024))>0)
          write(1,buf,cnt);
     return 0;
}
```

b) `$  echo “Algo para leer” | cat01 | true`

El mandato `true` termina bien inmediatamente, de manera que el pipe a la salida de `cat01` se quedará sin lectores y cuando `cat01` escriba en dicho pipe recibirá SIGPIPE. Pero para que `cat01` escriba primero debe leer (véase el bucle de trabajo), es por eso que hemos conectado su entrada con otro mandato que le proporcione algo para leer.

c) Un archivo se recorrerá secuencialmente, devolviendo: 1024, 1024, 1024, 428 y finalmente 0, este último indicando fin de fichero y permitiendo salir del bucle y terminar el programa. Se habrán transmitido los 3500 bytes. La duración no está acotada, pero será muy muy breve (milisegundos).

d) De un terminal se leen líneas completas, devolviendo: 6, 7, 10, 7, 8, 7, 8 y finalmente 0, o la longitud de cada linea (fin de línea incluido) más la indicación de EOF (Ctrl-D por terminal). Total 53 bytes en 7 minutos aproximados.

e) De una tubería se lee lo que haya en cada momento. Como llega una línea de texto cada segundo, cada una de ellas se leerá por separado devolviendose el número de caracteres leídos que incluirá los caracteres de fin de línea: 1, 2, 3, … 999, 1000 y finalmente 0, (la indicación de EOF). Total de 1001*1000/2 bytes en 1000 segundos.

Sistemas Operativos 5º semestre. Grado II

Segundo Parcial. Gestión de Procesos. 6 de noviembre de 2017.

Dispone de 50 minutos. Publicación el 24 de noviembre. Revisión el 28 de noviembre.

Ejercicio 1 (1.5 puntos)
En algunas ocasiones un proceso puede no querer recibir alguna señal, para lo cual puede ignorar la señal o bloquearla. ¿Cuál es la diferencia entre bloquear e ignorar una señal? ¿Son acciones incompatibles o excluyentes? ¿Qué servicios UNIX utilizaríamos para llevar a cabo una u otra acción?

Cuando se genera una señal, ésta debe ser entregada al proceso al que va dirigida. Si la señal está bloqueada, ésta queda pendiente de entrega, y será entregada cuando el proceso la desbloquee en su máscara de señales. Por otro lado, si la señal está ignorada, ésta es descartada, independientemente de la máscara de señales y no quedando pendiente para su posterior entrega. Ambas acciones son compatibles y pueden llevarse a cabo simultáneamente.

Los servicios que utilizaremos para bloquear una señal son los que nos permiten establecer/modificar una máscara de señales: sigprocmask() para manipular la máscara y sigemptyset(), sigaddset(), sigdelset(), etc. para establecer el conjunto de señales se aplicará el bloqueo/desbloqueo.

El servicio que usaremos para ignorar una señal será sigaction() con el valor SIG_IGN como handler.

Ejercicio 2 (1 punto)
¿Qué es el BCP y qué información contiene?

El BCP (Bloque de Control de Proceso) es una estructura de datos que maneja el sistema operativo y que almacena información relativa a un proceso. Cada proceso debe estar representado por un BCP que contiene, entre otra, la siguiente información:

- *Información de identificación (PID, PPID, UID y GID reales y efectivos, etc.).*
- *Estado del procesador: copia de los registros del procesador cuando el proceso no está en ejecución.*
- *Información de control del proceso:*
  - *Información de planificación y estado*
  - *Regiones de memoria asociadas al proceso*
  - *Recursos asignados: descriptores de ficheros abiertos, puertos, temporizadores, etc.*
  - *Información sobre señales: armado y máscara de señales*
  - *Información de contabilidad*
  - *Etc.*

Ejercicio 3 (1.5 puntos)
¿Para qué sirve el servicio `alarm()`? ¿Cómo funciona? ¿Qué efecto tiene sobre el proceso que lo invoca? ¿Qué uso típico podemos hacer de él?

El servicio alarm() establece un temporizador en el proceso que lo invoca. Dicho temporizador provoca que, una vez transcurrido el tiempo establecido en la invocación del servicio, se envíe al proceso la señal SIGALRM. Dicha señal tiene asociado como comportamiento por defecto la muerte del proceso que la recibe, por lo que es una forma muy sencilla de establecer un tiempo máximo de ejecución para un proceso hijo que ejecute una tarea potencialmente larga: después del fork(), en el código del proceso hijo, se puede establecer el temporizador por el tiempo máximo deseado y, a continuación, lanzar la tarea que queramos que pueda ser finalizada si excede el tiempo máximo establecido.

Ejercicio 4 (6 puntos)
Una conocida empresa de Internet quiere desarrollar una "App" para que sus usuarios puedan retocar sus fotos antes de subirlas a la Web. Durante el diseño del programa los ingenieros dudan de si es más conveniente una implementación basada en procesos pesados o en *threads*. El diseño actual contempla que el programa reciba como argumentos el nombre del fichero que contiene la imagen y los nombres de los filtros que se quieren aplicar. Un ejemplo de invocación del programa podría ser:

```
./instamiligram   pict20171031.jpg   ludwig   juno   slumber
```

El programa principal deberá abrir la imagen pasada como primer argumento y, a continuación, crear tantos procesos (o hilos) como filtros se hayan indicado en la línea de mandatos. Los procesos deberán quedar conectados formando un anillo de manera que los filtros se apliquen como un *pipeline* donde cada filtro se aplica sobre el resultado producido por el filtro anterior, salvo el primero que recibirá la imagen original del proceso padre. Finalmente el proceso padre deberá recoger el resultado de la aplicación del último y escribir dicho resultado en un nuevo fichero con el mismo nombre que el de la imagen original, pero con el añadido "_processed" antes de la extensión. Para simplificar la implementación, asumiremos que existen el tipo de datos `image_t` y las tres funciones siguientes:

```
typedef unsigned char image_t [ImageSize];
image_t * readImage(char * fname);
int writeImge(char * fname, image_t * outputImage);
image_t * applyFilter(image_t * inputImage, char * filterName);
```

La primera de las funciones se encarga de abrir y leer la imagen cuyo nombre se pasa como argumento y devuelve un puntero a un `image_t` que contiene la información de la imagen. La segunda de las funciones recibe como primer argumento el nombre de la imagen original y el resultado de aplicar el último filtro y lo escribe en un fichero del mismo nombre que el original, pero con el añadido anteriormente mencionado. Finalmente, la tercera recibe como argumentos un `image_t` que representa la imagen y el nombre del filtro que se debe aplicar y devuelve un nuevo `image_t` con el resultado de aplicar el filtro sobre la imagen de entrada.

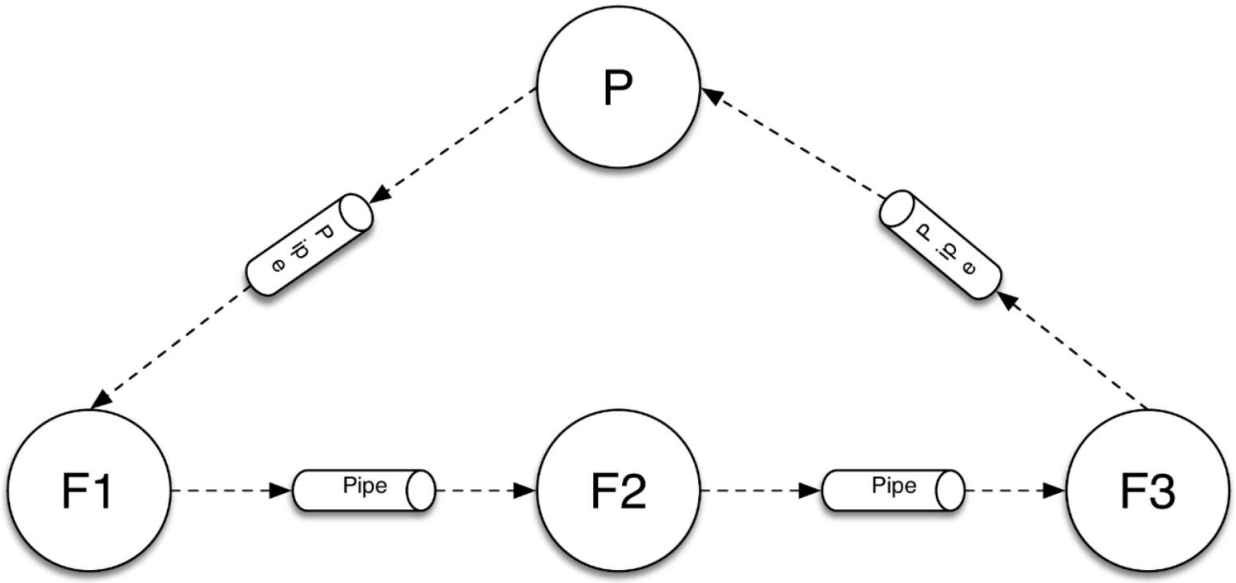
NOTAS:

- Para simplificar el problema, asumiremos que el tamaño de las imágenes es constante, tal y como se muestra en el que código que tenemos a continuación.
- Asimismo, no se tendrá en cuenta en esta implementación la necesidad de liberar los recursos reservados en las funciones auxiliares para almacenar la información de las imágenes.

Asumiendo que invocamos al programa de la siguiente forma:

```
./instamiligram   pict20171031.jpg   ludwig   juno   slumber
```

a) Dibuje el diagrama de procesos en el momento en que todos los procesos están en ejecución. Dicho diagrama debe mostrar tanto la jerarquía de procesos como los mecanismos de comunicación entre los mismos.



El proceso padre crea las 4 tuberías necesarias para conectar los cuatro procesos que ejecutarán simultáneamente de acuerdo a la línea de ejecución de más arriba: el mismo proceso padre y los tres procesos hijos, uno por cada filtro solicitado. En esta jerarquía el proceso padre se encuentra al principio y al final de la secuencia de procesos encadenados, mientras que los procesos hijos ejecutan secuencialmente, uno tras otro.

b) Indique en qué momento los procesos hijos terminarán su ejecución por completo y discuta en qué estado se encontrarían hasta ese momento.

Los procesos hijos no pueden terminar hasta que el proceso padre haga wait() por ellos. Por lo tanto, todos ellos quedarán en estado zombie hasta que el padre espera por ellos (línea 53 del código proporcionado).

c) Si antes de que un proceso A termine su ejecución el proceso posterior a éste en la secuencia de filtros muere (por ejemplo, porque le llega una señal con este comportamiento por defecto), ¿qué le ocurriría al proceso A?

En ese escenario el proceso A recibirá la señal SIGPIPE al intentar escribir en un pipe sin lectores. Como el comportamiento por defecto ante la recepción de esta señal es la muerte del proceso que la recibe, el proceso A morirá.

d) Escriba una implementación equivalente utilizando hilos en vez de procesos.

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

#define ImageSize 1024*768
typedef unsigned char image_t [ImageSize];

image_t* readImage(char* fname);
int writeImage(char* fname, image_t* outputImage);
image_t* applyFilter(image_t* inputImage, char* filterName);

image_t *inputImage, *outputImage;

void* runTask(void *p) {
      inputImage=outputImage;

      outputImage=applyFilter(inputImage, (char*)p);
      return (void*)NULL;
}

int main(int argc, char*argv[]) {
      int nFiltros=argc-2;
      int i;

      pthread_t thid[nFiltros];

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

      inputImage=readImage(argv[1]);
      printf("Imagen original: %s\n", *inputImage);

      for (i=0; i<nFiltros; i++) {
            int ret = pthread_create(&thid[i], &attr, &runTask, argv[i+2]);
            if (ret != 0) {perror("p_create()"); exit(1);}

            ret = pthread_join(thid[i], (void**)NULL);
            if (ret != 0) {perror("p_join()"); exit(1);}
      }

      pthread_attr_destroy(&attr);

      writeImage(argv[1], outputImage);

      return 0;
}
```

A continuación, se muestra la implementación basada en procesos pesados:

```
 1: #include <stdlib.h>
 2: #include <stdio.h>
 3: #include <unistd.h>
 4: #include <sys/wait.h>
 5:
 6: #define ImageSize 1024*768
 7: typedef unsigned char image_t [ImageSize];
 8: image_t * readImage(char * fname);
 9: int writeImage(char * fname, image_t * outputImage);
10: image_t * applyFilter(image_t * inputImage, char * filterName);
11:
12: int main(int argc, char*argv[]) {
13:   int nFiltros=argc-2;
14:   int i;
15:
16:   int fd[(nFiltros+1)][2];
17:
18:   for (i=0; i<(nFiltros+1); i++)
19:     pipe(fd[i]);
20:
21:   for (i=0; i<nFiltros; i++) {
22:     if (fork()==0) {
23:       int j;
24:       for (j=0; j<(nFiltros+1); j++) {
25:         if (j!=i)
26:           close(fd[j][0]);
27:         if(j!=(i+1))
28:           close(fd[j][1]);
29:       }
30:
31:       image_t inputImage, *outputImage;
32:       read(fd[i][0], &inputImage, ImageSize);
33:       outputImage=applyFilter(&inputImage, argv[i+2]);
34:       write(fd[i+1][1], outputImage, ImageSize);
35:       exit(0);
36:     }
37:   }
38:
39:   for (i=0; i<(nFiltros+1); i++) {
40:     if (i!=nFiltros)
41:       close(fd[i][0]);
42:     if (i!=0)
43:       close(fd[i][1]);
44:   }
45:
46:   image_t * origImage=readImage(argv[1]);
47:   write(fd[0][1], origImage, ImageSize);
48:   image_t finalImage;
49:   read(fd[nFiltros][0], &finalImage, ImageSize);
50:
51:   writeImage(argv[1], &finalImage);
52:
53:   while (wait(NULL)>0) continue;
54:
55:   return 0;
56: }
```

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos – GII

Tercer Parcial. Gestión de Memoria. 18 de diciembre de 2017

Dispone de 50 minutos. Las notas saldrán el 12 de enero. La revisión será el 15 de enero a las 12:00 en la sala de cristal S4200.

C1	C2	C3	P1a	P1b	P1c	P1d

Cuestiones (conteste las cuestiones en el espacio reservado al efecto en esta misma hoja y el problema en una hoja aparte)

Responder brevemente pero de forma razonada a las siguientes preguntas.

1) [1 punto] Sea un programa del usuario root que hace uso de una biblioteca dinámica con montaje automático en tiempo de carga. Especifique qué regiones de memoria tendría al iniciar su ejecución así como los permisos, la fuente, si son privadas o compartidas y si son de tamaño fijo o variable.

Las regiones que tendría serían (1) código, (2) datos con valor inicial, (3) datos sin valor inicial, (4) heap, (5) código de biblioteca dinámica, (6) datos con valor inicial de biblioteca dinámica, (7) datos sin valor inicial biblioteca dinámica y (8) pila

Y sus características serían:

(1): compartida, tamaño fijo, R-X, fichero ejecutable
(2): privada, tamaño fijo, RW-, fichero ejecutable
(3) y (7): privada, tamaño fijo, RW-, rellenar a 0s
(4) y (8): privada, tamaño variable, RW-, rellenar a 0s
(5): compartida, R-X, fichero biblioteca
(6): privada RW-, fichero biblioteca

2) [1 punto] Explicar en qué consiste la técnica de optimización “Buffering de páginas”.

Consiste en mantener una reserva mínima de marcos libres, realizando las operaciones de reemplazo de forma anticipada. De esta forma, el tratamiento del fallo de página es más rápido puesto que basta con coger un marco de página libre. Cuando el sistema operativo detecta que el número de marcos de página disminuye por debajo de un cierto umbral, aplica repetidamente el algoritmo de reemplazo hasta que el número de marcos libres llegue a otro umbral que corresponda a una situación de estabilidad. Las páginas liberadas limpias pasan a una lista de marcos libres, mientras que las páginas sucias pasan a una lista de marcos modificados que deberán limpiarse (escribirse en memoria secundaria) antes de poder volver a utilizarse. Esta limpieza de marcos se puede intentar hacer cuando el sistema esté menos cargado y en lotes para obtener un mejor rendimiento del dispositivo de swap.

3) [1 punto] Un programador quiere hacer uso de la biblioteca plug-in.so solamente en el caso de que exista dicho fichero de biblioteca y en caso contrario escribir un mensaje de aviso por la salida estándar. Razonar qué tipo de montaje de biblioteca dinámica necesitaría usar y escribir el extracto del código necesario. Se podrá hacer uso de la función int existe_fichero (char* nombre_fichero); que devuelve 0 si no existe y 1 si existe el fichero pasado como argumento.

Debería usar montaje explícito para poder realizar o no la llamada a dlopen, con el nombre del fichero, dependiendo de si se cumple la condición.

```
void* handler;
if (existe_fichero("plug-in.so" ))
      handler = dlopen("plug-in.so", RTLD_LAZY);
else
      fprintf(stderr, "Aviso: no se ha encontrado la biblioteca plug-in.so\n");
```

Problema 1 (7 puntos)

Suponga un sistema de memoria virtual con páginas de 4 KiB que dispone de la optimización Copy-On-Write. El programa descrito a continuación tiene un fichero ejecutable con una cabecera de 1 KiB, una sección de código de 5 KiB y el espacio correspondiente a los datos descritos en el código. La pila y el heap ocupan inicialmente 52 KiB. Suponga además que se realiza montaje dinámico al invocar el procedimiento y que las regiones de código y datos de la biblioteca del lenguaje ocupan respectivamente 313 KiB y 85 KiB.

programa.c:

```
pid_t pid[10];
int array_aux[2000];

int main(void)
{
  int status;
  int a=5, i;
  printf("%d\n",a);
  for (i=0;i<10;i++){
    switch (fork()){
       case -1: break;
     case 0: break;
     default: break;
     }
  }
  for (i=0;i<10;i++){
    wait(&status);
  }
  return 0;
}
```

a) (1 punto) A partir del código descrito anteriormente, indique el número de páginas total que necesitarán las imágenes de memoria de todos los procesos una vez que se han creado todos. Explique la respuesta.

El proceso cuenta inicialmente con regiones de código (2 páginas), DVI+DSVI+Heap (0+2+13=15 páginas), pila (13 páginas) y monta dinámicamente la biblioteca libc que tiene una región de código (79 páginas) y una región de datos (22 páginas). Los imágenes de memoria de los procesos hijo son clones del padre. Los procesos hijos no realizan ninguna modificación en ninguna variable, así que con la optimización COW compartirán las páginas asignadas al padre hasta que algún proceso realice alguna escritura en las páginas compartidas. Si en el código del hijo se hubiera invocado exit(0), como hubiera sido deseable, solo existiría un nivel de recursión en la creación de los hijos, pero al no incorporarse, se crea una genealogía de procesos con 10 niveles de profundidad. Si nos fijamos solamente en la primera generación, solo el proceso padre va modificando la variable i en cada iteración del bucle y que sirve de índice del bucle, por lo que la página de la pila que contiene esa variable y que era compartida entre el proceso padre y el proceso hijo creado en cada iteración se desdobla. Por lo tanto, el número de páginas será la suma de las regiones enumeradas, 131 páginas, más las páginas de la pila desdobladas, 10. En total, 141 páginas.

b) (1 punto) Si en lugar de utilizar el servicio fork se utilizara la función de biblioteca pthread_create para crear threads, ¿cuántas regiones tendría el proceso? Explique el resultado.

Cada thread dispondría de su propia pila, más las regiones mencionadas anteriormente para el proceso padre: región de código, datos junto con heap, pila, y las dos regiones correspondientes a la biblioteca. En total, 15.

c) (2 puntos) Incorpore las modificaciones que crea pertinentes en el código anterior para que cada proceso en ejecución disponga de una imagen de memoria distinta durante su ejecución. Explique la respuesta.

Basta con incorporar la invocación del servicio exec en el código de los hijos en donde el argumento del programa a ejecutar sea distinto en la ejecución de cada hijo. Con el servicio exec se cambia la imagen de memoria del proceso en ejecución, por lo que la imagen del padre se modificará en los hijos con la mera invocación del exec. En el caso de los procesos hijos, basta con incorporar un argumento diferente en cada caso, p. ej. el índice del bucle o el pid del proceso. De esta forma, la pila de cada proceso hijo tendrá un contenido distinto.

```
      ...
 case 0:    char literal[100];
            sprintf(literal,"%d",i);
            execl("./programaA","programaA",literal,(char *)NULL);
            perror("Error en el exec"); exit(1);
            break;
```

d) (3 puntos) Incorpore las mínimas modificaciones necesarias para que todos los procesos puedan acceder a una región cuyo contenido sea el del fichero "./datos.txt".

Basta con realizar la proyección del fichero en memoria con el flag MAP_SHARED antes del bucle de creación de los procesos hijo. De esta forma, las imágenes de memoria de los procesos hijos ya tendrán el contenido del fichero con los permisos adecuados para realizar las modificaciones:

```
char * orig;
int fd;
struct stat fich;

fd=open("datos.txt",O_RDWR);
if (fd == -1){
      perror("Error en la apertura del fichero datos.txt");
      exit(1);
      }
if (fstat(fd,&fich)==-1){
      perror("Error al obtener el tamaño del fichero");
      exit(2);
      };
orig=mmap(NULL,fich.st_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
if (orig == MAP_FAILED){
      perror("Error en la creación de la región");
      exit(3);
      }
close(fd);
for (i=0;i<10;i++){
      switch(fork()){
      ...
```

Sistemas Operativos – GII

Cuarto Parcial. Sincronización y comunicación. 18 de diciembre de 2017

Cuestiones:

Responder **brevemente pero de forma razonada** a las siguientes preguntas.

1) [1 punto] Se desea diseñar un sistema de reservas online de entradas de cine. ¿Qué modelo de comunicación sería preferible usar?

Este tipo de sistema se ajusta mejor a un modelo de comunicaciones cliente-servidor, frente a un modelo P2P, dado que la base de datos de reservas estaría centralizada en el servidor, facilitando el control del acceso sincronizado de los clientes y, aunque se tendrán múltiples clientes, el número no será tan elevado.

2) [1 punto] Se desean comunicar y asegurar exclusión mutua entre N procesos ligeros. Indicar qué opciones existen.

Para comunicar se podría usar las regiones de la imagen de memoria que comparten o bien un fichero, siendo la primera opción la más sencilla y más rápida.
Para asegurar exclusión mutua se podrían usar mutex/condición o bien semáforos.

3) [1 punto] Se desea comunicar dos procesos remotos. Uno de ellos recogerá imágenes de una cámara conectada por USB y le enviará los fotogramas adquiridos al otro proceso. Detallar qué mecanismo de comunicación se debe usar.

El mecanismo adecuado para comunicar dos procesos remotos serían los sockets. Y, si en este caso asumimos que la pérdida o llegada desordenada de algún mensaje con un fotograma no supone un problema, el tipo de sockets que se usarían serían de tipo datagrama.

4) [1 punto] Explique qué ocurre si un proceso solicita leer 10 bytes de una tubería cuando dicha tubería: (a) está vacía y hay un descriptor de escritura asociado a la misma; (b) está vacía y no hay ningún descriptor de escritura asociado a la misma; (c) contiene 5 bytes; (d) contiene 15 bytes.

a) **La operación de lectura se queda bloqueada**
b) **La lectura devuelve 0 (EOF)**
c) **Se leen 5 bytes**
d) Se leen 10 bytes

Problema

La barrera es un mecanismo que permite sincronizar múltiples flujos de ejecución, tengan éstos el mismo código o diferente, tal que cuando un flujo llega a una barrera, no proseguirá su ejecución hasta que todos los demás flujos alcancen la misma. Este mecanismo ofrece una función para iniciar la barrera (`inicio_barrera(barr_t *b, int nflujos)`), que recibe como parámetros el descriptor de la barrera y el número total de flujos que usarán la misma, y otra función que se invoca cada vez que se quiere realizar la sincronización propiamente dicha (`barrera(barr_t *b, int flujoID)`), que recibe como parámetros el descriptor de la barrera y el identificador del flujo que invoca la función, que será un valor entre 0 y *`nflujos-1`*. Este mecanismo puede implementarse usando cualquiera de las herramientas de sincronización que ofrece el sistema operativo.

a) Solución basada en semáforos sin nombre pero **sólo para dos flujos de ejecución**. Considere un escenario en el que el programa inicia la barrera y, a continuación, crea **dos** *threads* a cada uno de los cuales les pasa como parámetros el descriptor de la barrera y su identificador de flujo. Se plantea una solución que usa un semáforo por cada flujo. **Se pide identificar a qué corresponde cada uno de los 5 símbolos que incluyen una interrogación** (**NOTA**: Tenga en cuenta que, al tratarse de sólo 2 flujos, si uno tiene como identificador *`x`*, al otro le corresponderá *`1-x`*).

```
typedef struct {
  int nflujos;
  sem_t *sems; //vector de semáforos
} barr_t;
void inicio_barrera(barr_t *b, int nflujos){
  // implementada de forma genérica
  b->nflujos = nflujos;
  b->sems = malloc(nflujos * sizeof(sem_t));
  for (int i=0; i<nflujos; i++)
    sem_init(&(b->sems[i]), 0, ?1);
}
```

```
void barrera(barr_t *b, int flujoID){
  // implementada sólo para 2 flujos (0 y 1)
  sem_?2(&(b->sems[?3]));
  sem_?4(&(b->sems[?5]));
}
```

b) Solución basada en *mutex* y variables de condición. Considere el mismo escenario que en el apartado anterior, pero **para un número cualquiera de flujos**. Se plantea una solución que usa un *mutex* y una variable de condición, así como una variable (`nflujos_sinc`) que refleja cuántos flujos han alcanzado la barrera en cada momento (inicialmente, ninguno). **Se pide identificar a qué corresponde cada una de las dos operaciones de sincronización**.

```
typedef struct {
  int nflujos; int nflujos_sinc;
  pthread_mutex_t m; pthread_cond_t c;
} barr_t;
void inicio_barrera(barr_t *b,int nflujos){
  b->nflujos = nflujos;
  b->nflujos_sinc = 0;
  pthread_mutex_init(&(b->m), NULL);
  pthread_cond_init(&(b->c), NULL);
}
```

```
void barrera(barr_t *b, int flujoID){
  pthread_mutex_lock(&(b->m));
  if (++b->nflujos_sinc == b->nflujos) {
     b->nflujos_sinc=0;
     // OP1 de sincronización
  } else {
     // OP2 de sincronización
  }
  pthread_mutex_unlock(&(b->m));
}
```

c) Solución basada en *sockets stream*. Considere un escenario en el que se ejecutan en máquinas distintas **dos programas** independientes que reciben los argumentos pertinentes (dirección IP, puerto, …). Se plantea una solución asimétrica en el sentido de que cada proceso ejecuta un código

diferente. Suponga que el código que aparece a continuación corresponde a la iniciación de la barrera en cada flujo.

Código de inicio de la barrera del flujo 0

```
a = socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);

struct sockaddr_in dir = {
   .sin_family = AF_INET,
   .sin_port = htons(atoi(argv[1])),
   .sin_addr ={INADDR_ANY}};

b = bind(a,(struct sockaddr *)&dir,
         sizeof(dir));
c = listen(a,1);
d = accept(a, NULL, NULL);
```

Código de inicio de la barrera del flujo 1

```
a = socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);

struct sockaddr_in dir = {
   .sin_family = AF_INET,
   .sin_port = htons(atoi(argv[1])),
   .sin_addr={inet_addr(argv[2])}};

b = connect(a,(struct sockaddr *)&dir,
            sizeof(dir));
```

Una vez realizada esa iniciación, cada vez que se tengan que sincronizar los procesos en la barrera, se intercambiarán entre sí un mensaje que contenga un byte con un valor cualquiera usando un esquema cliente-servidor. **Se pide detallar únicamente el código de cada flujo que realiza esa sincronización siguiendo el esquema propuesto.**

Solución

a) En la solución planteada en el enunciado para el caso de dos flujos de ejecución con un semáforo por cada uno, cada flujo debe notificar al otro que ha llegado a la barrera y esperar a que suceda la misma circunstancia con el otro flujo. Por tanto, en primer lugar, cada flujo realizará un `sem_post` del semáforo del otro flujo y, a continuación, un `sem_wait` de su semáforo, que, al estar iniciado a 0, lo hará esperar hasta que haya llegado la notificación del otro flujo.

`?1 → 0`

`?2 → sem_post`

`?3 → 1-flujoID`

`?4 → sem_wait`

`?5 → flujoID`

Dando como resultado, el siguiente código:

```
  . . . . .
  sem_init(&(b->sems[i]), 0, 0);
}
void barrera(barr_t *b, int flujoID){
  // implementada sólo para 2 flujos (0 y 1)
  sem_post(&(b->sems[1-flujoID]));
  sem_wait(&(b->sems[flujoID]));
}
```

Nótese que también sería válida una solución donde los índices del vector de semáforos se usen justo al contrario: `?3 → flujoID` y `?5 → 1-flujoID`

b) En la solución propuesta con *mutex* y variables de sincronización, la rama `else` del `if` corresponde a la llegada a la barrera de los sucesivos flujos, exceptuando el último, y, por tanto, la acción de sincronización debe significar el bloqueo del flujo mediante `pthread_cond_wait`. Por otro lado, la rama afirmativa del `if` corresponde a cuando alcanza la barrera el último flujo y, por tanto, tiene que desbloquear a todos los demás flujos que están esperando mediante `pthread_cond_broadcast`.

```
void barrera(barr_t *b, int flujoID){
```

```
  pthread_mutex_lock(&(b->m));
  if (++b->nflujos_sinc == b->nflujos) {
     b->nflujos_sinc=0;
     pthread_cond_broadcast(&(b->c));        // OP1 de sincronización
  } else {
     pthread_cond_wait(&(b->c), &(b->m));    // OP2 de sincronización
  }
  pthread_mutex_unlock(&(b->m));
}
```

c) En la solución basada en *sockets stream* planteada en el enunciado, en la fase inicial los dos flujos establecen una conexión entre sí. A la hora de realizar la sincronización asociada a la barrera, tal como establece el enunciado, los flujos tienen que intercambiar un byte siguiendo un esquema cliente-servidor.

Código de la barrera del flujo 0

```
char c;
read(d, &c, 1);
write(d, &c, 1);
```

Código de la barrera del flujo 1

```
char c;
write(a, &c, 1);
read(a, &c, 1);
```

Tenga en cuenta que el envío de datos por un *socket*, sea mediante `write` o usando `send`, no conlleva ningún tipo de sincronización entre el emisor y el receptor: los datos se copian a un *buffer* local y se completa la llamada de petición de envío.

Nótese que en el flujo 0, que actúa de servidor, hay que usar en la comunicación el valor devuelto por la llamada `accept`, que corresponde al descriptor de *socket* conectado con el cliente, mientras que en el flujo 1 se utiliza directamente el valor devuelto por la llamada `socket`.

Departamento de Arquitectura y Tecnología de Sistemas Informáticos.
Sistemas Operativos 5º Semestre Grado II

Examen Final del Semestre. 22 de Enero de 2018. Teoría y Problemas.

Para la realización del examen en total se dispone de 120 minutos.
Las notas se publicarán el 30. La revisión será el 1 a las 12:00 en la sala S4200.

PRIMERA PARTE

Cuestión 1 (1 punto)

Considere un proceso que ejecuta satisfactoriamente una llamada exec y explique qué sucede con los siguientes recursos que tenía previamente asignados: (**a**) regiones de memoria; (**b**) ficheros abiertos; (**c**) disposición de las señales; (**d**) *threads* activos.

a) La ejecución de esta llamada conlleva la destrucción de todas las regiones del mapa de memoria que tenía el proceso antes de hacer la llamada, para, a continuación, la construcción del nuevo mapa basado en el ejecutable especificado en la misma.
b) Los ficheros abiertos se mantienen en ese estado después de la llamada. Nótese que esta característica permite, entre otras cosas, realizar en el proceso hijo las redirecciones de descriptores requeridas antes de llevar a cabo esta llamada.
c) Las señales capturadas pasan al estado por defecto, puesto que el código que realiza su tratamiento va a desaparecer del mapa de memoria del proceso como parte de esta llamada. En cuanto a las señales ignoradas o con el tratamiento por defecto, se mantienen en ese mismo estado.
d) Cuando un *thread* de un proceso realiza esta llamada, se termina incondicionalmente la ejecución de todos los otros *threads* del proceso. Observe que no pueden continuar su ejecución puesto que el código que están ejecutando va a desaparecer del mapa de memoria del proceso como parte de esta llamada.

Cuestión 2 (1 punto)

Identifique qué tipo de enlace, físico o simbólico, se debería usar en las siguientes circunstancias: (**a**) es un enlace a un directorio; (**b**) es un enlace a un fichero almacenado en otro sistema de ficheros; (**c**) se pretende reducir la posibilidad de que el fichero se elimine por error; (**d**) se pretende crear un enlace a un fichero que todavía no existe y que se creará posteriormente.

- **a) Simbólico**: no se permite crear enlaces físicos a directorios, puesto que podrían generar ciclos en el sistema de ficheros.
- **b) Simbólico**: no es posible crear enlaces físicos a ficheros almacenados en otro sistema de ficheros.
- **c) Físico**: si es usa un enlace simbólico y se borra el fichero original, se pierde el fichero, cosa que no ocurre con un enlace físico, que asegura que el fichero existe mientras tenga al menos un enlace de este tipo asociado.
- **d) Simbólico**: a la hora de crear un enlace físico, hay que especificar el nombre de un fichero que ya exista. Sin embargo, en el caso de los enlaces simbólicos, el nombre de fichero especificado no tiene que existir a priori.

Cuestión 3 (1 punto)

Explique cómo afectan al mapa de memoria de un proceso las siguientes operaciones, identificando qué regiones de memoria crean, en caso de que lo hagan: (**a**) open; (**b**) mmap; (**c**) pthread_create con los atributos por defecto; (**d**) dlopen.

- **a)** open: no crea ninguna región de memoria.
- **b)** mmap: crea una región de memoria asociada al fichero con las características que correspondan a los parámetros especificados en la llamada

- **c)** pthread_create: cuando se invoca esta función con el valor de los atributos por defecto, se crea una región de memoria para que almacene la pila de ese nuevo *thead*.
- **d)** dlopen: esta función carga una biblioteca dinámica, creando las regiones requeridas por la misma: una región para el código de la biblioteca y las regiones asociadas a los datos con y sin valor inicial de la misma, respectivamente.

Cuestión 4 (1 punto)
Se requiere crear una sección crítica para asegurar la exclusión mutua en la ejecución de un determinado fragmento de código por parte de *N* flujos de ejecución en un sistema UNIX usando semáforos. Especifique (**a**) qué valor inicial debería tener el semáforo, (**b**) qué servicio del sistema operativo se debería usar al entrar en la sección crítica (**c**) y cuál al salir.

a) El valor inicial del semáforo debería ser igual a 1 para permitir que un flujo pueda entrar en la sección crítica.
b) Debería usar `sem_wait` para competir en la entrada de la sección crítica.
c) Debería usar `sem_post` para indicar que ha salido de la sección crítica.

SEGUNDA PARTE

Ejercicio 1 (3 puntos)

Sea el código que se proporciona en la página siguiente. En este código, un proceso comprueba periódicamente la existencia de un fichero en una determinada ruta (dada por `FNAME`). Para ello, duerme una serie de segundos y, cada vez que despierta, comprueba si el fichero existe. Si dicho fichero existe, debe despertar a un conjunto de procesos auxiliares (su número viene determinado por `N_PROC`) que se encuentran en espera. Para ello, se utilizará como sistema de comunicaciones el envío de una señal (`SIGUSR1`, por ejemplo). Al recibir la señal, estos procesos deben despertar y leer la información que el proceso principal les envía a través de un *pipe* (nombre del fichero que deben abrir, un carácter que deben buscar y las posiciones inicial y final para cada uno de ellos). El objetivo de estos procesos es buscar el carácter indicado en el fichero pasado en la región delimitada por las posiciones inicial y final que se les indique. Una vez hecho el trabajo, deben devolver por otro *pipe* el número de ocurrencias del carácter en la región en que han buscado y el proceso principal sumará todos estos valores.

Para simplificar el código, se han omitido los *includes* necesarios (asumiremos que todos son correctos) y el código de la función `strsplit` (usada en la línea 27). Esta función tiene como objetivo *"des-serializar"* los datos que el proceso principal envía a través del pipe (línea 58) donde los 4 valores que se quieren comunicar se han combinado en una cadena y separado por "`\t`". Se asume que dicha función funciona correctamente y devuelve los valores correctos una vez extraídos de la cadena.

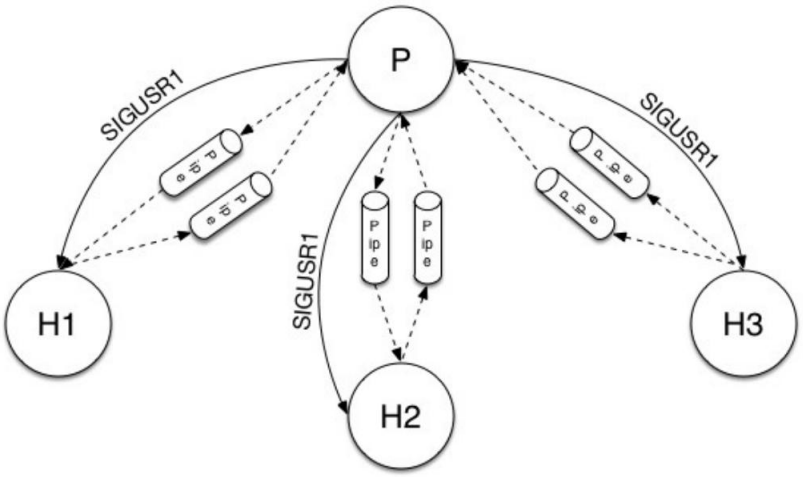
1) [0.25 puntos] Dibuje la jerarquía de procesos creada por el programa y los mecanismos de comunicación existentes entre los mismos.
2) [0.75 puntos] Añada el código necesario para que el proceso principal envíe a los procesos auxiliares la señal `SIGUSR1` para que éstos despierten (línea 48). Adicionalmente, discuta si es necesario añadir algún código adicional para los hijos (línea 16) justificando su respuesta. Si dicho código es necesario, inclúyalo.
3) [0.50 puntos] Proporcione el código necesario para calcular el tamaño del fichero en la línea 45. El tamaño del fichero debe almacenarse en una variable `size` que se usa posteriormente en el programa. ¿Se le ocurren más de una manera de calcular el tamaño del fichero?
4) [0.50 puntos] Rellene los huecos en las llamadas a `read` y `write` señalados con `XXXX` con el código necesario.
5) [1 punto] Complete la implementación de la función `find_in_file` (líneas marcadas con `XXXX`). Asegúrese de sólo leer el número de caracteres necesarios, ya que el `buffer` de la línea 7 se crea únicamente con esa capacidad.

```
01: int find_in_file(char* fname, char c, int init_pos, int end_pos) {
02:   int n=0;
03:   XXXX
04:   XXXX
05:
06:   int sz=end_pos-init_pos+1;
07:   char buffer[sz];
08:   XXXX
09:
10:   int i;
11:   for(i=0; i<sz; i++) {
12:     if(buffer[i]==c)
13:       n++;
14:   }
15:
16:   return n;
17: }
```

Código adjunto al Ejercicio 1.

```
01: #define N_PROC 3
02: #define BUFF_SIZE 1024
03: #define FNAME "/tmp/important_file"
04: /* AQUÍ IRÍAN TODOS LOS INCLUDES NECESARIOS */
05:
06: void strsplit(char* buffer, char* fname, char* c, int* init_p, int* end_p);
07:
08: int main(int argc, char const *argv[]) {
09:   int i, fd_ptc[N_PROC][2], fd_ctp[N_PROC][2], children[N_PROC];
10:
11:   for (i=0; i<N_PROC; i++) {
12:     pipe(fd_ptc[i]);
13:     pipe(fd_ctp[i]);
14:
15:     if ((children[i]=fork())==0) {
16:
17:
18:       while(1) {
19:         pause();
20:
21:         char buffer[BUFF_SIZE];
22:         int n=read(XXXX, XXXX, BUFF_SIZE);
23:         buffer[n]='\0';
24:
25:         char c, fname[BUFF_SIZE];
26:         int init_pos, end_pos;
27:         strsplit(buffer, fname, &c, &init_pos, &end_pos);
28:
29:         n=find_in_file(fname, c, init_pos, end_pos);
30:
31:         snprintf(buffer, BUFF_SIZE, "%d", n);
32:         write(XXXX, XXXX, strlen(buffer));
33:       }
34:     }
35:   }
36:
37:   while(1) {
38:     sleep(3);
39:
40:     char seekedc="abcdefghijklmnopqrstuvwxyz"[random()%26];
41:
42:     if (access(FNAME, F_OK)!=-1) {
43:       char buffer[BUFF_SIZE];
44:
45:
46:
47:       for (i=0; i<N_PROC; i++) {
48:
49:
50:         int init_p=size/N_PROC*i;
51:         int end_p;
52:
53:         if (i!=N_PROC-1)
54:           end_p=(size/N_PROC*(i+1))-1;
55:         else
56:           end_p=size-1;
57:
58:         snprintf(buffer, BUFF_SIZE, "%s\t%c\t%d\t%d", FNAME, seekedc, init_p, end_p);
59:         write(XXXX, XXXX, strlen(buffer));
60:       }
61:
62:       int total=0;
63:       for (i=0; i<N_PROC; i++) {
64:         int n=read(XXXX, XXXX, BUFF_SIZE);
65:         buffer[n]='\0';
66:         total+=atoi(buffer);
67:       }
68:     }
69:   }
70:   return 0;
71: }
```

1) **Jerarquía de procesos.**



2) **Código señal SIGUSR1.**

El código necesario en la línea 48 es:

```
kill(children[i], SIGUSR1);
```

Además, sí sería necesario añadir código adicional en la línea 16 que permita capturar la señal recibida por parte de los hijos y que éstos no mueran (comportamiento por defecto):

```
struct sigaction act;
act.sa_handler=handler;
sigemptyset(&act.sa_mask);
act.sa_flags=0;
sigaction(SIGUSR1, &act, NULL);
```

La función `handler`, que habría que definir antes del main, podría ser simplemente:

```
void handler(int signum) {
      return;
}
```

3) **Código tamaño fichero.**

Una forma de calcular el tamaño podría ser usando el servicio `stat`:

```
struct stat st;
stat(FNAME, &st);
int size=st.st_size;
```

También podría hacerse abriendo el fichero y usando `lseek` para posicionarse al final del mismo:

```
int fd=open(FNAME, O_RDONLY);
int size=lseek(fd, (size_t)0, SEEK_END);
close(fd);
```

4) **Huecos en las llamadas a `read` y `write`.**

Línea 22: `int n=read(fd_ptc[i][0], buffer, BUFF_SIZE);`
Línea 32: `write(fd_ctp[i][1], buffer, strlen(buffer));`
Línea 59: `write(fd_ptc[i][1], buffer, strlen(buffer));`
Línea 64: `int n=read(fd_ctp[i][0], buffer, BUFF_SIZE);`

5) Implementación de la función `find_in_file`.

```
int find_in_file(char* fname, char c, int init_pos, int end_pos) {
  int n=0;
  int fd=open(fname, O_RDONLY);
  int pos=lseek(fd, init_pos, SEEK_SET);

  int sz=end_pos-init_pos+1;
  char buffer[sz];
  int l=read(fd, buffer, sz);

  int i;
  for(i=0; i<sz; i++) {
    if(buffer[i]==c)
      n++;
  }

  return n;
}
```

TERCERA PARTE

Ejercicio 2 (3 puntos)
Sean dos procesos que van a actuar como PRODUCTOR y CONSUMIDOR usando ambos el código adjunto pero uno recibe PRODUCTOR como primer argumento y el otro CONSUMIDOR. Ambos usan un mecanismo de comunicación mediante memoria compartida usando un fichero proyectado.

Además, se tienen los siguientes datos:

- Los procedimientos `producir_registro` y `consumir_registro` pertenecen a la biblioteca dinámica "`registros.so`".
- El montaje de bibliotecas dinámicas es automático al invocar el procedimiento.
- El tamaño de página es de 4KiB.
- La pila y la región de *heap* inicialmente serán de 12KiB cada una.
- Asumir que no se produce ningún error.
- Las regiones de datos con valor inicial, datos sin valor inicial, *heap* y pila son regiones independientes.
- El tamaño del texto es 13340B.
- La salida del mandato `size` para "`registros.so`" es: `text 10432 data 2340 bss 5670`

```
01:   #define TAM_REGISTRO 100
02:   #define NUM_REGISTROS 10
03:
04:   char* f = NULL;
05:   int fd;
06:
07:   int main (int argc, char *argv[]) {
08:     int i;
09:
10:     fd = open("bd", O_RDWR);
11:     f = mmap(0, TAM_REGISTRO*NUM_REGISTROS,
12:              PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
13:     close(fd);
14:
15:     for (i = 0; i < NUM_REGISTROS; i++) {
16:       if (strcmp(argv[1], "PRODUCTOR") == 0)
17:         producir_registro(f + i * TAM_REGISTRO);
18:       else if (strcmp(argv[1], "CONSUMIDOR") == 0)
19:         consumir_registro(f + i * TAM_REGISTRO);
20:     }
21:
       ...
       return 0;
     }
```

1) [1 punto] Describir qué regiones tendrá el mapa de memoria de un proceso así como su tamaño en bytes, número de páginas, fuente, permisos y si es compartida o privada, antes de ejecutar la línea 10.

NOMBRE	TAM BYTES	NÚM. PÁGS.	FUENTE	PERMISOS	Comp/Priv
Texto	13340B	4	Fich. ejec.	R-X	Comp.
DVI	4B	1	Fich. ejec.	RW-	Priv
DSVI	4B	1	Rellenar a 0s	RW-	Priv
Heap	12KiB	3	Rellenar a 0s	RW-	Priv
Pila	12KiB	3	Rellenar a 0s	RW-	Priv

2) [1 punto] Describir qué cambios habría en el mapa de memoria al llegar a la línea 21.

Suponiendo que la pila y el heap no han necesitado más de las 3 páginas que tenían asignadas inicialmente, solamente se habrían creado las siguientes regiones nuevas:

NOMBRE	TAM BYTES	NÚM. PÁGS.	FUENTE	PERMISOS	Comp/Priv
Texto Bib.	10432B	3	Fich. bib.	R-X	Comp.
DVI Bib.	2340B	1	Fich. bib.	RW-	Priv
DSVI	5670B	2	Rellenar a 0s	RW-	Priv
Fich. proy.	1000B	1	Fich. proy.	RW-	Comp.

3) [1 punto] Describir el problema de sincronización que existe en este escenario y modificar el código para sincronizar ambos procesos usando un FIFO (que ya existe y se llama "`FIFO`"), pero manteniendo la comunicación mediante el fichero proyectado.

El problema es que la lectura no está sincronizada con la escritura, y por tanto el consumidor podría adelantar al productor. Si se quiere mantener la comunicación a través del fichero proyectado y sincronizar usando un FIFO, se podría mandar un token a través del FIFO cada vez que se escriba un registro. Habría que añadir el código que está en negrita.

```
char* f = NULL;
int fd;

int main (int argc, char *argv[])
{
  int i, fifo, token;

  fd = open("bd", O_RDWR);
  f = mmap(0, TAM_REGISTRO*NUM_REGISTROS,
           PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);

  fifo = open("FIFO", strcmp( argv[1], "PRODUCTOR" ) == 0 ? O_WRONLY :
O_RDONLY );

  for ( i = 0; i < NUM_REGISTROS; i++ )
  {
    if ( strcmp( argv[1], "PRODUCTOR" ) == 0 )
    {
      producir_registro( f + i * TAM_REGISTRO );
      write( fifo, &token, sizeof(token));
    }
    else if ( strcmp( argv[1], "CONSUMIDOR" ) == 0 )
    {
      read( fifo, &token, sizeof(token));
      consumir_registro( f + i * TAM_REGISTRO );
    }
  }

  return 0;
}
```

Sistemas Operativos, GII y GMI

Examen Extraordinario. 13 de Julio de 2018.

Problema 1 (5 ptos)

Sea un equipo Unix en el que las cuentas de usuario están en un sistema de archivos montado (`/home`), distinto del sistema de archivos raíz (`/`). Sea la variable: `char * RUTA = "/home/userA/datos.dat";`

El programa `progA`, ejecutado por el usuario `userA` (del grupo `users`), crea el archivo indicado por `RUTA` y a continuación, en un bucle, espera un minuto, escribe una letra en él e itera, y así para cada una de las 27 letras del abecedario, y termina.

El programa `progB`, ejecutado por el usuario `userB` (del grupo `users`), abre para lectura el archivo `RUTA` y a continuación arranca 5 hijos. Cada hijo, en un bucle, lee un carácter del archivo, espera un minuto, lo escribe por su salida estándar e itera. Los hijos terminan cuando detectan EOF. El padre espera a que terminen sus 5 hijos y termina.

A) **(1 pto)** Indique qué propietario, grupo y permisos debe tener cada nodo-i de `RUTA` para que las llamadas de apertura de `progA` y `progB` sean exitosas. ¿Cómo afectaría el `umask` de `userA` a dichas llamadas de apertura? ¿y cómo afectaría el `umask` de `userB`?

B) **(1 pto)** Codifique el programa `progB`.

C) **(1 pto)** Considere que `progB` es puesto en ejecución por `userB` exactamente 10:30 minutos después de que `userA` haya puesto en ejecución `progA`.
Describa detalladamente como avanzará la ejecución de los procesos involucrados.
Determine el instante de terminación de cada uno de los procesos involucrados.

Suponga ahora que el objeto con el nombre indicado por `RUTA` ya existía con anterioridad y que se trata del siguiente FIFO: `prw-rw-r-- 1 userA users 0 Jul 2018 /home/userA/datos.dat`

Conteste:

D) **(1 pto)** Explique el mecanismo de comunicación FIFO: cómo se crea, cómo se abre para usarlo y cómo se comporta cuando se lee o se escribe en él.
¿En qué sentidos el FIFO ofrece sincronización entre Productor y Consumidor?

E) **(1 pto)** Suponiendo ahora que `RUTA` es un FIFO, vuelva a contestar detalladamente a la pregunta del apartado **C**.

Solución:

A) El `progB` ejecutado por `userB` realizará un `open(RUTA,O_RDONLY);`. El `progA` ejecutado por `userA` realizó un `creat(RUTA,0666);` equivalente a un `open` con modo `O_WRONLY|O_CREAT|O_TRUNC`. Si `RUTA` ya existía `creat` no necesita escribir en el directorio y trunca el archivo. Si `RUTA` no existía se crea el archivo con los permisos indicados en el `creat` previa aplicación de la máscara del usuario `userA`. Esta máscara suele ser `007` o más restrictiva. Si tuviese activo el bit `0040`, el `open` de `progB` fallaría por falta del permiso de lectura necesario. La máscara de `userB` no afecta, pues `progB` no crea ningún objeto.

Componente de la `RUTA`:	Usuario propietario	Grupo propietario	Permisos que suele tener.	Necesario para `creat` de `progA`	Necesario para `open` de `progB`
`/`	`root`	`root`	`rwx r-x r-x`	`--- --- --x`	`--- --- --x`
`/home/ (punto de montaje)`	`root`	`root`	`rwx r-x r-x`	`No importa, queda tapado`	`No importa, queda tapado`
`/home/ (raíz del SF montado)`	`root`	`root`	`rwx r-x r-x`	`--- --- --x`	`--- --- --x`
`/home/userA/`	`userA`	`users`	`rwx r-x ---`	`-wx --- --- ó --x --- --- si ya existía`	`--- --x ---`
`/home/userA/ datos.dat`	`userA`	`users`	`rw- rw- --- 0666+umaskA`	`-w- --- --- si ya existía`	`--- r-- ---`

B)

```
// progB
#define N 5
int main(void) {
      int fd = open(RUTA,O_RDONLY);
      char ch;
      for (int i=0;i<N;i++) {
            switch(fork()) {
            case -1: perror("fork"); exit(1);
            case  0:
                  while(read(fd,&ch,1)>0) {
                        sleep(60);
                        write(1,&ch,1);
                  }
                  exit(0);
            default:
      }
      for (int i=0;i<N;i++) wait(NULL);
      return 0;
}
```

C) Cuando `progB` se pone en ejecución, `progA` ya llevará emitidas las letras `ABCDEFGHIJ`. Los 5 procesos hijo de `progB` comparten la posición (_) sobre el archivo abierto, luego cada carácter se lee una vez. El orden de ejecución de los procesos hijo no es fijo. Quién lee cada letra y en qué orden se emiten, no es predecible.

tiempo	datos.dat:A	progA	tiempo	datos.dat:B	H1	H2	H3	H4	H5	progB	emitido
01:00	_	A									
...	A_	B-I	...								
10:00	ABCDEFGHI_	J	10:30	A_BCDEFGHIJ|	E	B	A	D	C	wait	DBAEC
11:00	ABCDEFGHIJ_	K	11:30	...E_FGHIJK|	G	H	J	F	I	wait	FHGIJ
12:00	...EFGHIJK_	L	12:30	...FGHIJ_KL|	EOF	K	EOF	EOF	L	**H1,3,4**	LK
13:00	...FGHIJKL_	M	13:30	...GHIJKLM_|		M			EOF	**H5**	M
14:00	...GHIJKLM_	N	14:30	...HIJKLMN_|		N				wait	N
15:00	...HIJKLMN_	Ñ	15:30	...IJKLMNÑ_|		Ñ				wait	Ñ
16:00	...IJKLMNÑ_	O	16:30	...JKLMNÑO_|		O				wait	O
...	...JKLMNÑO_	P-Y	...	...STUVWXY_|		P-Y				wait	P-Y
27:00	...STUVWXY_	Z **END**	27:30	...TUVWXYZ_|		Z				wait	Z
			28:30	...TUVWXYZ_		EOF				**H2,END**	

D) Un FIFO del servicio de ficheros Unix es un `pipe` con nombre. Este mecanismo de comunicación se usa entre procesos locales no emparentados. Se crea con el servicio `mkfifo(char*name,int mode);` indicando qué nombre se le quiere dar y los permisos que tendrá (sobre los que se aplica el `umask`). Una vez abierto tiene la semántica de un `pipe`, sincronizando escrituras y lecturas siguiendo el modelo Productor-Consumidor. Se abre como si fuese un archivo, con un `open`, indicando nombre y modo de apertura, pero el sistema operativo fuerza la sincronización las aperturas de los dos extremos (el de lectura y el de escritura) haciendo esperar a quien abre primero hasta que la suceda la apertura del otro extremo.

E) En este caso `RUTA` se refiere a un FIFO creado previamente. Por lo tanto cuando `progB` arranque, `progA` llevará 10:30 minutos esperando en el `creat`, a que algún otro proceso abra el extremo de lectura del FIFO. Todavía `progA` no habrá escrito nada. Un minuto más tarde (11:30) escribe 'A' y 26 después (37:30) escribe 'Z' y termina.

Los 5 hijos de `progB` competirán por leer cada letra escrita por `progA`. No se puede determinar qué hijo consigue cada letra y la imprime 60 segundos después, pero todas ellas serán impresas en su orden.

En el instante 37:30, 4 de los hijos detectarán EOF y terminarán. Un minuto más tarde, el hijo restante emitirá la 'Z', detectará EOF y terminará. Luego terminará el proceso padre de `progB`, a los 38:30.

Sistemas Operativos 5º Semestre. Grado II

Primer Parcial. Sistema de Ficheros. 17 de Octubre de 2018.

Dispone de 60 minutos. Publicación: el 29 o antes. Revisión: el 31 o antes.

Ejercicio 1) (2 pts) Implemente en C y para Unix el mandato: `anular fich pos tam` que emite por la salida estándar la parte del contenido actual del archivo `fich` consistente en `tam` bytes a partir de la posición `pos` de dicho archivo, y además reescribe esa parte del archivo con bytes nulos. Realice las suposiciones que considere oportunas. Use `int atoi(char*str);`

Solución:

```
___1 #include <sys/types.h>
___2 #include <sys/stat.h>
___3 #include <fcntl.h>
___4 #include <stdlib.h>
___5 #include <unistd.h>
___6 
___7 int main(int argc, char*argv[])
___8 {
___9    int fd = open(argv[1],O_RDWR);
__10            if(fd<0) { perror(argv[1]); exit(1); }
__11    int pos = atoi(argv[2]);
__12    int tam = atoi(argv[3]);
__13    int cnt;
__14    int ret;
__15    char buf[tam];
__16 
__17    ret = lseek(fd,pos,SEEK_SET);
__18    cnt = read(fd,buf,tam);
__19    ret = write(1,buf,cnt);
__20    ret = lseek(fd,-cnt,SEEK_CUR);
__21    for(int i=0;i<cnt;i++) buf[i]='\0';
__22    ret = write(fd,buf,cnt);
__23    close(fd);
__24 
__25    return 0;
__26 }
```

Ejercicio 2) (2 pts) Según la implementación de `anular` realizada por usted en el ejercicio 1, **razone** cuál será su comportamiento explicando qué tamaño y contenido tendrán finalmente los archivos auxiliares `A` y `B`, en de cada uno de los siguientes casos independientes:

a) `echo 0123456789 > A ; ./anular A 5 10 > B`

b) `echo HOLA > H ; ln -s H A ; rm H ; ./anular A 0 5 > B`

Solución:

a) `echo 0123456789 > A`

El archivo `A` se crea con 11 bytes, los dígitos del 0 al 9 más un final de línea.

`./anular A 5 10 > B`

Se crea el archivo `B` que recogerá lo emitido por `anular` por su salida estándar, esto es, los caracteres del `5` (en la posición 5) en adelante hasta el final de fichero, pues se pretenden leer 10 bytes pero sólo hay 6.

El programa anular reescribe con el byte nulo sólo los mencionados bytes del archivo `A`. El archivo no crece de tamaño pues no se escribe más de lo que se leyó ni más allá de su tamaño.

b) `echo HOLA > H`

El archivo `H` se crea con 5 bytes, (`H`, `O`, `L`, `A`) más un final de línea.

`ln -s H A`

Se crea un enlace simbólico a `H`, con nombre `A`.

`rm H`

Se borra `H`. Ahora `A`. apunta a una ruta inexistente.

`./anular A 0 5 > B`

Se crea el archivo `B`, vacío de contenido y con 0 bytes, que recogerá lo emitido por `anular` por su salida estándar, que en este caso será nada, pues el intento de apertura del enlace `A` fallará, pues el `open` sólo indica modo `O_RDWR`.

Si la apertura hubiese sido "`open(...,O_RDWR|O_CREAT,0666)`", entonces la ruta apuntada por `A` (el archivo de nombre `H`) se habría creado vacío, el intento de lectura desde la posición 0 devolvería un 0 y no se emitiría nada por la salida estándar.

Ejercicio 3) (3 pts) Sea la siguiente visión parcial del contenido de un sistema de ficheros:

NUM	T	U	G	O	USER	GROUP	PATH(RUTA)
1	d	rwx	r-x	r-x	root	root	/
2	d	rwx	rwx	rwt	root	root	/tmp/
3	-	r--	rw-	r--	luke	jedi	/tmp/datos.bin
4	-	rw-	rw-	rw-	r2d2	robot	/tmp/datos2.bin
5	d	rwx	r-x	--x	root	root	/home/
6	d	rwx	rwx	---	c3po	robot	/home/c3po/
7	d	rwx	---	---	r2d2	robot	/home/r2d2/
8	l	rwx	rwx	rwx	c3po	robot	/home/c3po/temp -> /tmp/
9	-	rws	r-x	---	r2d2	robot	/home/c3po/anular

Donde: `NUM` es el número de ruta; `T` es el tipo de objeto; `U`, `G` y `O` son los grupos de permisos (*user*, *group* y *other*); y la notación `A -> B` indica que el enlace simbólico `A` apunta a la ruta `B`.

Considere que el usuario `c3po` (del grupo `robot`) invoca, desde su *home*, el siguiente mandato:

```
./anular ./temp/datos.bin 10 10
```

Para cada uno de los siguientes casos, **conteste detallando** cómo el S.O. decodifica la ruta indicada. Muestre: **(nº de) ruta, grupo de permisos y permiso concreto, que se validará a cada paso**.

a) ¿Podrá ponerse en ejecución el mandato indicado? ¿Qué identidad tendrá el proceso resultante? Detalle cada paso de la decodificación de la ruta.

b) ¿Podrá el proceso abrir el archivo indicado? ¿Qué valor devolverá la llamada al sistema de apertura del archivo? Detalle cada paso de la decodificación de la ruta.

Considere ahora que el usuario `c3po` invoca, desde `/tmp`, el mandato: `rm *`

c) ¿Conseguirá `c3po` borrar todos los archivos de este directorio? Razone la respuesta.

Solución:

a)

[9 G x] c3po pertenece al grupo robot. Con estos permisos se permite la ejecución.
[9 U s] Por otra parta está activado el bit SETUID, por lo que el usuario efectivo pasa a ser el dueño del ejecutable.
Identidad del proceso resultante:
UID real: c3po
GID real: robot
UID efectivo: r2d2
GID efectivo: robot
Al realizarse la ejecución desde el directorio home del usuario c3po, no es necesario realizar ningún paso adicional de decodificación de la ruta.

b)

Con la identidad del proceso antes descrita
[8 - -] No se comprueban los permisos de este nodoi, por ser un enlace simbólico
[1 o x] Se permite atravesar el directorio.
[2 o t] Se permite atravesar el directorio. Además, el directorio tiene activo el sticky bit.
[3 o r] En este caso solo tenemos permisos de lectura sobre el archivo.
El ejecutable anular necesita tener permisos de lectura/escritura sobre el archivo para poder hacer la llamada open con el flag O_RDWR. De esta forma, la llamada al sistema devolverá -1.

c) El directorio tiene activado el stycki bit, por lo que se permite crear y borrar entradas en este directorio con el UID efectivo del proceso. De esta forma, no se podrán borrar las entradas de otros usuarios.

Ejercicio 4) (3 pts) Considere un sistema de ficheros de tipo UNIX (basado en inodos con 12 punteros directos, 1 indirecto simple, 1 doble y 1 triple), con agrupaciones de 8 KiB, para un disco de 2 TiB y archivos con tamaño medio de 2 agrupaciones.

a) ¿Cuántas agrupaciones ocupará el mapa de bits de inodos en este sistema?

Suponga que estando este sistema de ficheros vacío (solo el directorio raíz cuyo inodo es el 2) y con un *umask* de 0027, se ejecuta la siguiente secuencia:

```
fd=creat("/F", 0644); link("/F", "/G"); write(fd, "a", 1);
```

b) Especifique en detalle el contenido de **todos** los inodos (para cada inodo indique el tipo de fichero, los permisos, el nº de enlaces y los punteros) y agrupaciones en uso en el sistema.

Solución:

a) El mapa de bits de inodos almacena un bit por cada inodo para indicar si ese inodo está libre u ocupado. Por tanto, ese mapa tendrá tantos bits como inodos pueda haber en ese sistema de ficheros. Teniendo en cuenta que cada inodo corresponde a un fichero, para determinar el tamaño de esta estructura es necesario calcular cuál es el número de ficheros que habrá en el sistema.

Dado que en el enunciado se especifica que el tamaño medio de un fichero es de dos agrupaciones, en promedio, cada fichero gastará 1 inodo y 2 agrupaciones, por lo que, para mantener equilibrado el gasto de inodos y de agrupaciones de datos, el número de inodos debería ser la mitad que el de agrupaciones. En consecuencia, el primer paso sería calcular cuántas agrupaciones hay en el disco:

- 2 TiB bytes / 8KiB bytes/agrupación = 2^{41} / 2^{13} agrupaciones = 2^{28} agrupaciones

A partir de ese valor, se puede determinar el número de ficheros y, por tanto, el número de bits que contiene el mapa de bits de inodos:

- (2^{28} agrupaciones / 2 agrupaciones/fichero) * 1 bit/fichero = 2^{27} bits

Puesto que nos piden cuántas agrupaciones ocupa ese mapa, se calcula teniendo en cuenta cuántos bits caben en una agrupación:

- 2^{27} bits/(8 bits/byte * 8KiB bytes/agrupación) = 2^{11} agrupaciones=**2048 agrupaciones**
- Ocupa el (2^{11} /2^{28}) * 100% del disco = 100/2^{17} % ≈ 0,0008 %

Expresado en una sola fórmula, el cálculo sería:

- Tamaño del mapa de bits de inodos = ((2^{41} / 2^{13}) / 2) / (2^{3} * 2^{13}) = 2048 agrupaciones

b) Teniendo en cuenta el estado inicial del sistema de ficheros, donde solo existe el directorio raíz, después de la ejecución de esa secuencia, en el sistema estarán en uso 2 inodos y 2 agrupaciones, que corresponden al directorio raíz y al único fichero creado.

Hay que tener en cuenta que el servicio *link* no crea un nuevo fichero, sino que añade una nueva entrada denominada "G" en el directorio raíz que hace referencia al mismo inodo que la entrada "F", que se incluyó como parte del servicio *creat*, incrementando el número de enlaces de ese inodo.

Con respecto a los permisos del fichero creado, hay que calcular cuáles son los permisos resultantes teniendo en cuenta los permisos solicitados (0644) y el valor de la máscara de *umask* (0027):

- 0644 & ~(0027) = 0644 & 0750 = 0640 (*rw-r*-----)

A continuación, se detalla el contenido de los 2 inodos y las 2 agrupaciones en uso:

- Contenido del inodo correspondiente al directorio raíz:
  - Inodo 2 (convención habitual de la familia UNIX).
  - Tipo directorio
  - Permisos 755 (*rwxr-xr-x*: permisos habituales del directorio raíz).

- 2 enlaces: al tratarse del directorio raíz, la entrada “.” y “..” apuntar al propio directorio.
- Solo usa el primer puntero directo que apunta a la primera agrupación de datos del sistema.

- Contenido del inodo correspondiente al nuevo fichero:
  - Inodo 3 (primero libre).
  - Tipo fichero
  - Permisos 640 (*rwxr*-----).
  - 2 enlaces: las entradas “F” y “G” del directorio raíz referencian a este inodo.
  - Solo usa el primer puntero directo que apunta a la segunda agrupación de datos del sistema.

- Contenido de la agrupación asociada al directorio raíz. Habrá 4 entradas tal que cada una incluye el nombre de la entrada y el número de inodo asociado a la misma:
  - . 2
  - .. 2
  - F 3
  - G 3
- Contenido de la agrupación de datos asociada al nuevo fichero. Dado que únicamente se ha escrito un byte, tiene asociada una sola agrupación que contiene ese valor:
  - ‘a’

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos 5º Semestre. Grado GII

Segundo Parcial. Gestión de Procesos. 7 de Noviembre de 2018

Dispone de 30 minutos. Las notas saldrán el 22 de noviembre. La revisión será el 26 de noviembre a las 13:00 en la sala de cristal S4200.

P1a	P1b	P2a	P2b	P2c	P2d

Conteste a los problemas en el espacio reservado al efecto en esta misma hoja)

Problema 1 (3 puntos)

A continuación se especifica el esqueleto del bucle principal de un **servidor paralelo**.

```
#define NUM_ARGS 10
pid_t pid; char *arg_accion[NUM_ARGS];
int main(void)
{       int status;   pid_t aux;
       /* Se inicializan los valores por defecto en el servidor y
        * se crean los mecanismos de comunicación con los clientes */
        while (1) {
           /* Se queda a la espera de las solicitudes de los clientes */
           /* Se recibe la solicitud del cliente */
           /* Se atiende la solicitud del cliente y se responde desde
            * el mandato accion */
               default: /* Se invoca el wait sin bloquear el servidor */
                     aux= waitpid(-1,&status,WNOHANG);
     }
     return 0;
}
```

a) (1,5 puntos) Complete en el bucle del servidor la implementación de la etapa correspondiente a la ejecución de la solicitud del cliente. Los servidores hijos se encargarán de ejecutar un programa denominado `accion` que recibe 10 argumentos, entre ellos la orden del cliente y el puerto de comunicación desde el que se debe contestar al cliente. Puede considerar que los argumentos están almacenados en la variable `char *arg_accion[NUM_ARGS]`. Tanto en este como en el siguiente apartado, tenga en cuenta el control de errores en la invocación de todos los servicios del sistema.

`Solución (no es necesario especificar los ficheros .h):`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#define NUM_ARGS 10
pid_t pid; char *arg_accion[NUM_ARGS];
int main(void)
{       int status;    pid_t aux;
       /* Se inicializan los valores por defecto en el servidor y
        * se crean los mecanismos de comunicación con los clientes */
       while (1) {
          /* Se queda a la espera de las solicitudes de los clientes */
          /* Se recibe la solicitud del cliente */
          /* Se atiende la solicitud del cliente y se responde desde
           * el mandato accion */
           pid= fork();
           switch (pid) {
                case -1:
                     perror("Error en la llamada a fork");
                     exit(1);
                case 0:
                     execvp("accion",arg_accion);
                     perror("Error en la llamada a exec");
                     exit(2);
                default:
                     aux= waitpid(-1,&status,WNOHANG);
                     if (aux == -1)
                     {
                          perror("Error en la llamada a wait");
                          exit(3);
                     }
           }
       }
       return 0;
}
```

b) (1,5 puntos) Realice el apartado a) con una implementación basada en threads independientes (también denominados asíncronos). En este caso, la invocación del mandato `accion` se sustituye por la invocación de la función `fun_accion` a la que se le pasa como argumento la variable `char *arg_accion[NUM_ARGS]`.

`Solución (no es necesario especificar los ficheros .h):`

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#define NUM_ARGS 10
void fun_accion(char *argumentos[NUM_ARGS])
{    /* Ejecución de la solicitud del cliente y escritura de la
        * respuesta en el puerto de comunicaciones */
}
int main(void)
{  pthread_t thid; pthread_attr_t attr; int aux= 0;
   char *arg_accion[NUM_ARGS];
   /* Se inicializan los valores por defecto en el servidor y
    * se crean los mecanismos de comunicación con los clientes */
   aux= pthread_attr_init(&attr);
if (aux== -1){perror("Error en la inicialización del thread");return 1; }
aux= pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
if (aux== -1) {
          perror("Error en la definición el estado del thread"); return 1; }
while (1) {
  /* Se queda a la espera de las solicitudes de los clientes */
  /* Se recibe la solicitud del cliente */
  /* Se atiende la solicitud del cliente y se responde desde
   * el mandato accion */
  aux= pthread_create(&thid,&attr,(void *) fun_accion, arg_accion);
  if (aux== -1)
          perror("Error en la creación del thread");
   }
   pthread_attr_destroy(&attr);
   return 0;
}
```

Problema 2 (3 puntos)

El fragmento de código mostrado a continuación corresponde con un programa que desea paralelizar la ejecución de dos tareas usando, para ello, dos hilos de ejecución. Además, el programador ha decidido gestionar determinadas señales tal y como se muestra en el código. Se puede asumir que todas las señales están sujetas al comportamiento por defecto al comienzo del programa.

```
L1:  #include <pthread.h>
L2:  #include <signal.h>
L3:  #include <stdio.h>
L4:  #include <stdlib.h>
L5:  void* task:(void *arg) {
L6:        printf("Task 1 running...\n");
L7:        return NULL;
L8:  }
L9:  void* task2(void *arg) {
L10:       printf("Task 2 running...\n");
L11:       return NULL;
L12: }
L13: void captura(int s) {
L14:       // Signal treatment code...
L15: }
L16: int main(int argc, char const *argv[]) {
L17:       struct sigaction acc1, acc2;
L18:       pthread_t thid[2];
L19:       pthread_attr_t tattr;
L20:       acc1.sa_handler= captura;
L21:       acc1.sa_flags= SA_RESTART;
L22:
L23:       sigaction(SIGINT, &acc1, &acc2);
L24:       pthread_attr_init(&tattr);
L25:       pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_JOINABLE);
L26:       if (pthread_create(&thid[0], &tattr, &task1, NULL) != 0) {
L27:             perror("pthread_create");
L28:             exit(1);
L29:       }
L30:       if (pthread_create(&thid[1], &tattr, &task2, NULL) != 0) {
L31:             perror("pthread_create");
L32:             exit(1);
L33:       }
L34:       printf("Tasks launched...\n");
L35:       sigaction(SIGINT, &acc2, &acc1);
L36:       pthread_join(thid[0], NULL);
L37:       pthread_join(thid[1], NULL);
L38:       printf("Main program finishing...\n");
L39:       return 0;
L40: }
```

1) (0,75 Puntos) ¿Qué ocurriría si el programa recibe la señal SIGINT mientras está ejecutando en la línea 20? ¿Y si la recibe mientras ejecuta en la línea 34? ¿Qué hilo atendería el tratamiento de la señal en este último caso?

Si el programa recibe la señal SIGINT mientras ejecuta la línea 20, el proceso moriría, pues es el comportamiento por defecto ante la recepción de esta señal y ésta no ha sido ignorada, armada ni bloqueada en este punto.

Si, en lugar de en la 20 la señal llega mientras el proceso ejecuta en la línea 34 del programa, entraría en ejecución la función de armado de la señal, que sería invocada desde cualquiera de los hilos de ejecución (los dos creados explícitamente o el principal) indistintamente.

b) (0,75 Puntos) ¿Qué hace la línea 35?

Devuelve el comportamiento frente a la recepción de la señal SIGINT a la configuración por defecto (que, en el caso de esta señal, supondría la muerte del proceso).

c) (1 Punto) Si el usuario pulsa la combinación de teclas CTRL-C mientras ejecuta la línea 24, ¿qué efecto tendría sobre el programa? Y si en lugar de en la línea 24 lo hace en la 36, ¿se llegaría a imprimir el mensaje de la línea 38?

La pulsación de la combinación de teclas CTRL-C hace que se envíe la señal SIGINT al proceso. Si se hace mientras el proceso ejecuta en la línea 24, el proceso estará protegido frente a la señal y entrará en ejecución la función de tratamiento de señal.

Sin embargo, si ésta llega mientras el programa se encuentra en la línea 36, ésta provocará la muerte del proceso y el mensaje de la línea 38 nunca se llegará a ejecutar dado que el armado de la señal se ha eliminado en la línea 35, devolviendo el comportamiento frente a la recepción de esta señal al valor por defecto, que supone la muerte del proceso.

d) (0,50 Puntos) ¿Qué hacen las líneas 36 y 37?

Estas dos líneas hacen que el hilo principal espere por la finalización de los dos hilos creados anteriormente para ejecutar las tareas encomendadas.

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos 5º Semestre. Grado GII

Tercer Parcial. Gestión de Memoria. 28 de Noviembre de 2018

Dispone de 40 minutos. Las notas saldrán el 13 de diciembre. La revisión será el 17 de diciembre a las 13:00 en la sala de cristal S4200.

P1a	P1b	P1c1	P1c2	P2

Conteste a los problemas en el espacio reservado al efecto en esta misma hoja

Problema 1 (3 puntos)

Sea un proceso que ejecuta el programa adjunto y con las siguientes premisas:

- Direcciones de 32 bits.
- La función "`malloc`" pertenece a la biblioteca "`libc.so`", que se montará dinámicamente al invocar el procedimiento
- El tamaño de página es de 1KiB
- La pila y la región de heap inicialmente estarán vacías.
- Asumir que no se produce ningún error
- Las regiones de datos con valor inicial, datos sin valor incial y heap son regiones independientes
- El tamaño del texto del programa es 5250B
- La salida del size para "libc.so" es: text 6543 data 543 bss 234
- Asumir que las variables de entorno serán 100B y que el programa se llama sin argumentos

```
1   char* r = NULL, s, *t;
2   int f, b = 3;
3
4   int main (int argc, char *argv[]) {
5     int i;
6     static char c;
7
8     b = 4;
9     r = malloc(500);
10    f = open("fich.txt", R_ONLY);
11    t = mmap(0, 1500, PROT_READ, MAP_SHARED, f, 0);
12    close(f);
```

a) [1 punto] Describir qué regiones tendrá el mapa de memoria del proceso incluyendo su tamaño en bytes, número de páginas, fuente, permisos y si es compartida o privada, al ejecutar la línea 8

b) [1 punto] Describir qué cambios habría en el mapa de memoria al ejecutar la línea 12

Asumiendo que `open`, `mmap` y `close` no hacen uso del heap,

c1) [0,5 puntos] describir qué cambios habría en el mapa de memoria si después de la línea 12 se ejecutara `s = malloc(1000); free(r);`

c2) [0,5 puntos] Ídem para `free(r); s = malloc(500);`

Solución a)

NOMBRE	TAM BYTES	NÚM.PÁGS.	FUENTE	PERMISOS	Comp/Priv
Texto	5250	6	Fich. ejec.	R-X	Comp.
DVI	4(r)+4(b)	1	Fich. ejec.	RW-	Priv
DSVI	4(f)+4(t)+1(s)+1(c)	1	Rellenar a 0s	RW-	Priv
Pila	100(entorno)+4(argc)+4(argv)+4(Ret. Main)+4(Puntero marco anterior)+4(i)+ **(longitud_del_nombre_del_ejecutable+1)**	1	Rellenar a 0s	RW-	Priv

Solución b) Suponiendo que la pila no ha necesitado crecer más de la página que ya tenía, solamente se habrían creado las siguientes regiones nuevas:

NOMBRE	TAM BYTES	NÚM. PÁGS.	FUENTE	PERMISOS	Comp/Priv
Texto Bib.	6543	7	Fich. bib.	R-X	Comp.
DVI Bib.	543	1	Fich. bib.	RW-	Priv
DSVI Bib.	234	1	Rellenar a 0s	RW-	Priv
Fich. proy.	1500	2	Fich "fich.txt"	R--	Comp.
Heap	500	1	Rellenar a 0s	RW-	Priv

Solución c1) El Heap crece en una página más (tendría 2 en total) por el malloc y el free no afecta a las regiones.

Solución c2) El free no afecta y como el malloc es de un tamaño menor que el tamaño de la región no será necesario hacer crecer el heap. Por tanto, no habría cambios.

Problema 2 (3 puntos)

A continuación se especifica el código fuente de un programa que utiliza montaje explícito para acceder a las funciones `mi_mmap` y `mi_munmap` almacenadas en la biblioteca dinámica "`mi_mmap.so`". Complete el código para que los procesos compartan una región utilizando estas funciones en lugar de la invocación directa a los servicios del sistema. La variable `direccion` almacenará la ubicación de esta región. Tenga en cuenta los posibles errores en la ejecución del nuevo código. No replique todo el código, solo algunas líneas del enunciado que sirvan de referencia.

```
/* Código fuente de la biblioteca mi_mmap.so */
 void *mi_mmap(void *addr, size_t length, int prot, int flags,
               int fd, off_t offset){
       void *aux;  size_t tamano= 0;
       tamano= length+ (2*sysconf(_SC_PAGESIZE));
       aux= mmap(addr, tamano, prot, flags, fd, offset);
       return aux; }
int mi_munmap(void *addr, size_t length){
       int aux; size_t tamano= 0;
       tamano= length+ (2*sysconf(_SC_PAGESIZE));
       aux= munmap(addr, tamano);
       return aux; }
/* Fin del código fuente de la biblioteca mi_mmap.so */

/* Código fuente del programa que hay que completar */
pid_t pid, aux; int *direccion, *direccion_aux;
int main (void){
 int status, errores, i, longitud = 4000;   char *str_error;
 struct sigaction act;  void *manejador;

 pid = fork ();
 switch (pid)   {
  case -1: perror ("Error en la llamada a fork"); exit (1);
    case 0:
     for (i = 0; i < 10; i++ ) {
        *direccion = i;
        printf ("DirecciónH: %p ContenidoH: %d\n", direccion, *direccion);
        direccion++ ;
       } /* for */

     errores = mi_munmap (direccion - 10, longitud);

     exit (0);
    default:
     aux = waitpid (-1, &status, pid);
     direccion_aux = direccion;
     for (i = 0; i < 10; i++ ) {
        *direccion_aux = i + 100;
        printf ("DirecciónP: %p ContenidoP: %d\n", direccion_aux,
              *direccion_aux);
        direccion_aux++ ;
       } /* for */

     errores = mi_munmap (direccion, longitud);

     exit (0);
    } /* switch */
}
```

Solución

```
/* Código fuente completo */
pid_t pid, aux; int *direccion, *direccion_aux;
int main (void)
{
 int status, errores, i, longitud = 4000;   char *str_error;
 struct sigaction act;  void *manejador;
/* Declaraciones de símbolos externos */
 void *(*mi_mmap) (void *addr, size_t length, int prot, int flags,
                   int fd, off_t offset);
 int (*mi_munmap) (void *addr, size_t length);

 manejador = dlopen ("./mi_mmap.so", RTLD_LAZY);
 if (!manejador) {fprintf (stderr, "%s\n", dlerror ()); exit (1); }
 mi_mmap = dlsym (manejador, "mi_mmap");
 str_error = dlerror ();
 if (str_error != NULL)
   {fprintf (stderr, "dlsym: %s\n", str_error); exit (1);}
 direccion = mi_mmap (NULL, longitud, PROT_READ | PROT_WRITE,
          MAP_ANONYMOUS | MAP_SHARED, -1, 0);
 if (direccion == (void *) -1) {perror ("Error mmap"); exit (1);}
 mi_munmap = dlsym (manejador, "mi_munmap");
    str_error = dlerror ();
    if (str_error != NULL)
      {fprintf (stderr, "dlsym: %s\n", str_error); exit (1);}
 pid = fork ();
 switch (pid) {
  case -1: perror ("Error en la llamada a fork"); exit (1);
   case 0:
    for (i = 0; i < 10; i++ ) {
       *direccion = i;
       printf ("DirecciónH: %p ContenidoH: %d\n", direccion, *direccion);
       direccion++ ;
      } /* for */
    errores = mi_munmap (direccion - 10, longitud);
    if (errores == -1){perror ("Error munmap"); exit (1);}
    errores= dlclose(manejador);
    if (errores != 0){perror ("Error dlclose"); exit (1);}
    exit (0);
   default:
    aux = waitpid (-1, &status, pid);
    direccion_aux = direccion;
    for (i = 0; i < 10; i++ ) {
       *direccion_aux = i + 100;
       printf ("DirecciónP: %p ContenidoP: %d\n", direccion_aux,
             *direccion_aux);
       direccion_aux++ ;
      } /* for */

    errores = mi_munmap (direccion, longitud);
    if (errores == -1) {perror ("Error munmap"); exit (1);}
    errores= dlclose(manejador);
    if (errores != 0){perror ("Error dlclose"); exit (1);}
    exit (0);
   } /* switch */
}
```

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos 5º Semestre. Grado GII

Tercer Parcial. Sincronización y comunicación. 19 de diciembre de 2018

Dispone de 40 minutos. Las notas saldrán el 8 de enero La revisión será el 10 de enero a las 11:00 en la sala de cristal S4200.

1	2	3	4	5	6

Problema (6 puntos)

Sean las siguientes piezas de código:

```
/* Código A */

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

  int pid;

  pid = fork();
    if (pid == 0) {
      << acceso a recurso  compartido
         con proceso padre >>
    }
    else {
      << acceso a recurso compartido
         con proceso hijo >>
    }
}
```

```
/* Código B */

/* Ejecutable1 */
int main(int argc, char * argv[]) {

  << acceso a recurso compartido
     con proceso Ejecutable2 >>

}

/* Ejecutable2 */
int main(int argc, char * argv[]) {

  << acceso a recurso compartido
     con proceso Ejecutable1 >>

}
```

```
/* Código C */

void *func (void *p) {
  << acceso a recurso compartido
     con el otro thread >>
}

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

  pthread_t th1, th2;


  pthread_create(&th1, NULL, &func, NULL);
  pthread_create(&th2, NULL, &func, NULL);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  return 0;
}
```

```
/* Código D*/

/* Ejecutable en máquina 1 */
int main(int argc, char * argv[]){

  << acceso a recurso compartido
     con proceso de máquina 2 >>

}

/* Ejecutable en máquina 2*/
int main(int argc, char * argv[]){

  << acceso a recurso compartido
     con proceso de máquina 1 >>

}
```

1) [1 punto] Rellenar la tabla de la siguiente página, indicando aquellos mecanismos que sean factibles y/o recomendables. En cada posición de la tabla, seleccionar una de las tres opciones: [No factible/Factible/Recomendable], para los distintos escenarios (códigos).

2) [1'5 puntos] Indicar para cada posición que hayáis rellenado como "No factible" la razón de dicha selección.

3) [1'5 puntos] Indicar para cada posición que hayáis rellenado como "Recomendable" la razón de dicha selección.

4) [2'5 puntos] Modificar el código A para que implemente una solución donde el proceso padre va leyendo de su entrada estándar números enteros, los incrementa en 1 y se los manda al proceso hijo para que los escriba por su salida estándar. Los procesos terminarán cuando el proceso padre lea un fin de fichero por su entrada estándar.

5) [1 punto] Indicar las modificaciones que serían necesarias para que la solución del apartado anterior se adapte al código B. No programar nada en este apartado.

6) [2'5 puntos] Modificar el código C para que implemente una solución donde los hilos deben alternarse en escribir un byte que leen por su entrada estándar en un fichero cuyo nombre se pasa como parámetro al ejecutable.

a)

Mecanismo / Código	Código A	Código B	Código C	Código D
Imagen de memoria única + semáforos sin nombre				
Imagen de memoria única + semáforos con nombre				
Imagen de memoria única + mútex y cond				
Región de memoria compartida (mmap + MAP_ANON+MAP_SHARED) + semáforos sin nombre				
Región de memoria compartida (mmap + MAP_ANON+MAP_SHARED) + semáforos con nombre				
Fichero proyectado + semáforos sin nombre				
Fichero proyectado + semáforos con nombre				
Pipes (tuberías sin nombre)				
FIFOS (tuberías con nombre)				
Fichero + cerrojos				
Sockets UNIX				
Sockets INET				

SOLUCIÓN:

1 La tabla rellena quedaría del siguiente modo:

Mecanismo / Código	Código A	Código B	Código C	Código D
Imagen de memoria única + semáforos sin nombre	*No factible*	*No factible*	*Factible*	*No factible*
Imagen de memoria única + semáforos con nombre	*No factible*	*No factible*	*Factible*	*No factible*
Imagen de memoria única + mútex y cond	*No factible*	*No factible*	*Recomendable*	*No factible*
Región de memoria compartida (mmap + MAP_ANON+MAP_SHARED) + semáforos sin nombre	*Recomendable*	*No factible*	*Factible*	*No factible*
Región de memoria compartida (mmap + MAP_ANON+MAP_SHARED) + semáforos con nombre	*Factible*	*No factible*	*Factible*	*No factible*
Fichero proyectado + semáforos sin nombre	*Factible*	*Factible*	*Factible*	*No factible*
Fichero proyectado + semáforos con nombre	*Factible*	*Recomendable*	*Factible*	*No factible*
Pipes (tuberías sin nombre)	*Recomendable*	*No factible*	*Factible*	*No factible*
FIFOS (tuberías con nombre)	*Factible*	*Recomendable*	*Factible*	*No factible*
Fichero + cerrojos	*Factible*	*Recomendable*	*No factible*	*No factible*
Sockets UNIX	*Factible*	*Factible*	*Factible*	*No factible*
Sockets INET	*Factible*	*Factible*	*Factible*	*Recomendable*

1 En el caso del código A, las únicas soluciones no factibles son aquellas que implican una imagen de memoria única, dado que los procesos padre e hijo no comparten esa imagen de memoria.

En el caso del código B, sólo son factibles aquellas soluciones en las que los mecanismos de comunicación y/o sincronización tienen nombre, tales como ficheros, FIFOS o sockets. L a combinación fichero proyectado con semáforos sin nombre es factible, definiendo los semáforos dentro de la región asociada al fichero proyectado.

En el caso del código C, la mayoría de las soluciones son factibles, aunque muchas de ellas no son recomendables, al tener todos los hilos una imagen de memoria única y ser más sencillo utilizar dicha imagen con algún mecanismo de sincronización ligero. La solución de ficheros más cerrojos no es factible, dado que el cerrojo se establece a nivel de proceso y no de hilo.

En el caso del código D, la única solución factible es la de los sockets de dominio INET, dado que ambos códigos se encuentran en máquinas diferentes.

2 En el caso del código A, son recomendables aquellas soluciones sin nombre, que pueden heredarse a través de descriptores de ficheros o como parte de una región de memoria compartida, dado que se trata de procesos padre e hijo.

En el caso del código B, las soluciones recomendables son aquellas en las que hay un mecanismo con nombre detrás, tales como ficheros, semáforos con nombre y FIFOS. Los sockets no son necesarios, dado que los procesos están en la misma máquina.

En el caso del código C, la solución recomendable es el uso de las variables compartidas, por tratarse de hilos que tienen una imagen de memoria única, más mútex y condiciones, por ser mecanismos ligeros, apropiados para los hilos.

En el caso del código D, la única solución recomendable y factible es la de los sockets de dominio INET, dado que ambos códigos se encuentran en máquinas diferentes.

3 El código A quedaría del siguiente modo:

```
int main(int argc, char * argv[]){
   int tub[2];
   int dato;

   pipe(tub);

   pid = fork();
   if (pid != 0) {
      close(tub[0]);
      while (scanf("%d",&dato)!= EOF) {
         dato++;
         write(tub[1],&dato, sizeof(int));
      }
      close(tub[1]);
   }
   else {
      close(tub[1]);
      while (read(tub[0], &dato, sizeof(int)) > 0) {
          printf("%d\n", dato);
      }
      close(tub[0]);
   }
      return 0;
}
```

4 El código del apartado anterior tendría que modificarse, de forma que, en lugar de utilizar una tubería sin nombre, ambos procesos usaran una tubería con nombre, dado que en este caso los procesos no tienen ningún parentesco.

5 Una posible solución sería utilizar una variable de control compartida, que determinara el turno, y mútex y condiciones para lograr los bloqueos necesarios:

```
int turno = 0, fd;
pthread_mutex_t mutex;
pthread_cond_t turno1, turno2;

void *func1 (void *p) {
   int dato;

   while (scanf("%d",&dato) != EOF) {
      pthread_mutex_lock(&mutex);
      while (turno != 0)
         pthread_cond_wait(&turno1,&mutex);
      pthread_mutex_unlock(&mutex);

       write(fd, &dato, sizeof(int));

       pthread_mutex_lock(&mutex);
       turno = 1;
       pthread_cond_signal (&turno2);
       pthread_mutex_unlock(&mutex);
   }
   pthread_exit(NULL);
}

void *func2 (void *p) {
   int dato;

   while (scanf("%d",&dato) != EOF) {
      pthread_mutex_lock(&mutex);
      while (turno != 1)
         pthread_cond_wait(&turno2,&mutex);
      pthread_mutex_unlock(&mutex);

      write(fd, &dato, sizeof(int));

      pthread_mutex_lock(&mutex);
      turno = 0 ;
      pthread_cond_signal (&turno1);
      pthread_mutex_unlock(&mutex);
   }
   pthread_exit(NULL);
}


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

   pthread_t th1, th2;

   pthread_mutex_init(&mutex,NULL);
   pthread_cond_init(&turno1, NULL);
   pthread_cond_init(&turno2, NULL);

   fd = creat(argv[1], 0640);

   pthread_create(&th1, NULL, &func1, NULL);
   pthread_create(&th2, NULL, &func2, NULL);

   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   close (fd);
   return 0;
}
```

Ejercicio 1 (3 puntos)

En el directorio /home/user/alumno1, invocando el mandato “ls -l”, se obtiene el siguiente listado:

```
-rwxr-xr-x  1  alumno1   alumnos  9526  6 jun 17:50   codigo
-rwxr-xr-x  1  alumno1   alumnos  1040  6 jun 17:20   codigo.c
drwxr-xr-x  3  alumno1   alumnos   128  6 jun 16:20   dirA
drwxr-xr-x  3  alumno1   alumnos    64  6 jun 16:21   dirB
```

Por su parte, si en el directorio /home/user/alumno1/dirA, se invoca el mandato “ls -l”, se obtiene el siguiente listado:

```
-rwxr-xr-x  1  alumno1   alumnos    28  6 jun 15:20   fich1
-rwxr-xr-x  1  alumno1   alumnos  1024  6 jun 16:20   fich2
drwxr-xr-x  3  alumno1   alumnos    64  6 jun 17:01   dirA1
```

Finalmente, si en el directorio /home/user/alumno1/dirB, se invoca el mandato “ls -l”, no se imprime nada en la salida estándar.

El código fuente correspondiente a *codigo.c*, cuyo ejecutable es *codigo*, es el siguiente:

```
 1: #include <dirent.h>
 2: #include <stdio.h>
 3: #include <unistd.h>
 4: #include <sys/stat.h>
 5:
 6: #define TAM_PATH 1024
 7:
 8: int main(int argc, char const *argv[]) {
 9:    DIR *dirp;
10:    struct dirent *dp;
11:    char entrada[TAM_PATH];
12:    struct stat st;
13:
14:    if (argc!=2){
15:           fprintf(stderr, "Error. Uso: %s directorio\n”, argv[0]);
16:           return 1;
17:    }
18:
19:    dirp=opendir(argv[1]);
20:    if (dirp==NULL)
21:           fprintf(stderr, "No puedo abrir %s\n”, argv[1]);
22:    else {
23:           while ((dp = readdir(dirp)) != NULL){
24:                  snprintf(entrada, sizeof(entrada),“%s/%s”, argv[1],dp->d_name);
25:                  stat(entrada,&st);
26:                  if (S_ISREG(st.st_mode))
27:                         printf(“%s tamaño %d bytes\n”, dp->d_name, st.st_size);
28:           }
29:           closedir(dirp);
30:    }
31:    return 0;
32: }
```

1) **[0.25 puntos]** ¿Qué se obtendría por la salida estándar cuando el usuario alumno1 ejecuta desde el directorio /home/user/alumno1/ la siguiente secuencia “./codigo /home/user/alumno1/dirA”?
2) **[0.25 puntos]** Realizar el mismo ejercicio anterior, en el caso de ejecutar la secuencia “./codigo /home/user/alumno1/dirB”.
3) **[0.25 puntos]** Indicar si hay alguna diferencia si en lugar de ejecutar “./codigo /home/user/alumno1/dirA”, se ejecuta “./codigo dirA”. Justificar la respuesta.
4) **[1 punto]** Sustituya el código de la línea 27 por un código equivalente que, utilizando la llamada lseek(), logre imprimir por la salida estándar el mismo resultado, pero **sin usar el valor de st.st_size**. Declare las variables necesarias.
5) **[1.25 puntos]** Modifique el código original, para que las operaciones que se hacen con cada una de las entradas del directorio que se pasa como argumento, las lleven a cabo n threads, uno por cada entrada. Se asume que el directorio no tiene más de 10 entradas.

Solución

1) El código recorre el directorio pasado como argumento, imprimiendo el tamaño en bytes de aquellos ficheros que son normales, precedido por su nombre. Por tanto, lo que se obtendría por la salida estándar, sería:

fich1 tamaño 28 bytes

fich2 tamaño 1024 bytes

2) Análogamente, si lo hacemos con el directorio dirB, como dicho directorio no tiene ningún fichero normal, no se imprimiría nada por la salida estándar.

3) No habría ninguna diferencia, dado que en un caso se utiliza el nombre absoluto y en el otro caso, un nombre relativo al directorio actual, pero que se refiere al mismo fichero. Los permisos habilitan al usuario para poder llevar a cabo todas las operaciones.

4) Se podría obtener el tamaño del fichero mediante la llamada lseek(), del siguiente modo:

```
int tam, fd;

. . .
if (S_ISREG(st.st_mode)){
        fd = open(entrada, O_RDONLY);
        if (fd>0) {
                tam = lseek(fd, 0, SEEK_END);
                printf("%s tamaño %d bytes\n", dp->d_name, tam);
                close(fd);
        }
}
. . ..
```

5) Al suponer que el número máximo de entradas es 10, el número máximo de threads tampoco excederá este número, por lo que podemos declarar un vector estático de threads, dentro del main(). Además, declararemos una función que realice todas las operaciones correspondientes a cada entrada. Dicha función recibirá la estructura correspondiente a la entrada del directorio (struct dirent *). El código quedaría del siguiente modo:

```
#define MAX_THREADS 10
. . .

char *directorio;

void *imp_entrada(void *p){
        char entrada[TAM_PATH];
        struct stat st;
        struct dirent *dpt = (struct dirent*)p;

        snprintf(entrada, sizeof(entrada), "%s/%s", directorio, dpt->d_name);
        stat(entrada, &st);
        if (S_ISREG(st.st_mode))
                printf("%s tamaño %d bytes\n", dpt->d_name, st.st_size);

        pthread_exit(NULL);
}

int main(int argc, char **argv){
        . . .
        pthread_t thid[MAX_THREADS];
        pthread_attr_t attr;
        int i, j;

        . . .
        directorio = argv[1];
        dirp = opendir(argv[1]);
        if (dirp == NULL)
                fprintf(stderr,"No puedo abrir %s\n", argv[1]);
        else {
                pthread_attr_init(&attr);
                pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
                i = 0;
                /* asigna cada entrada a un thread */
                while ((dp = readdir(dirp)) != NULL) {
                        pthread_create(&thid[i], &attr, &imp_entrada, (void *) dp);
                        i++;
                }
                closedir(dirp);
```

```
                /* espera por la finalización de todos los threads */
                for (j=0; j<i; j++)
                        pthread_join(thid[j], NULL);

                pthread_attr_destroy(&attr);
        }
        return 0;
}
```

Ejercicio 2 (3 puntos)

Se desea implementar un servicio que permita a distintos usuarios (usando el programa **P1**) consultar el resultado, a través de una web, de un dispositivo remoto que hace de marcador deportivo (sólo guarda dos enteros que representan el resultado de los dos equipos). Dicho marcador estará conectado por USB a un computador donde se ejecutarán los siguientes procesos: un proceso (ejecutando el programa **P2**) que leerá cada 10 segundos el resultado del dispositivo marcador y lo escribirá en un fichero; un proceso (ejecutando el programa **P3**) que creará un nuevo proceso para atender cada petición de los usuarios de la web. Nota: los nombres de los equipos no se guardan en el dispositivo marcador y son configurables por parte del cliente mediante argumentos del programa **P1.**

Código P1:

```
1   #include <stdlib.h>
2   #include <string.h>
3   char url[30] = "https://marcadorendirecto.com";
4   struct Marcador {
5    char nombreEquipoA[40];
6    char nombreEquipoB[40];
7    int puntuacionEquipoA;
8    int puntuacionEquipoB;
9  };
10  int main(int argc, char *argv[])
11  {
12   struct Marcador *m;
13   m = malloc(sizeof(struct Marcador));
14   strcpy(m->nombreEquipoA, argv[1]);
15   strcpy(m->nombreEquipoB, argv[2]);
     ...
L    free(m);
     return 0;
   }
```

Código P2

```
int main(int argc, char *argv[])
{
  ...
  leer_dispositivo_marcador(&marcador);
  escribir_fich_marcador(marcador, fich_fd);
  ...
}
```

Código P3

```
int main(int argc, char *argv[])
{
  ...
  leer_fich_marcador(&marcador, fich_fd);
  enviar_marcador(marcador);
  ...
}
```

1) **[0.5 puntos]** Razonar **a) qué comunicaciones ocurrirían**, **b) qué modelo/s de comunicación** sería/n el/los adecuado/s para resolver el problema y **c) qué mecanismos de comunicación** podrían usarse.
2) **[0.75 puntos]** Razonar **a) qué problemas de concurrencia** se pueden dar en dicho escenario, **b) cuáles son los mecanismos** adecuados para resolverlos y **c) esbozar el código** necesario que habría que añadir a los programas correspondientes y que genere la menor contención posible.

Teniendo en cuenta que los procesos que ejecuten el programa P1 correrán en máquinas: con direcciones de memoria de 64 bits; con tamaño de página de 4KiB; la pila y heap iniciales estarán vacíos; el entorno será de 500B; regiones de datos con valor inicial, datos sin valor inicial y heap como regiones independientes; el tamaño del código de P1 será de 15677B; suponiendo que la llamada `malloc` pertenece a la biblioteca libc.so (text 11567 data 5432 bss 3246) y la llamada `strcpy` a la biblioteca libstring.so (text 3567, data 8901, bss 1234) y el montaje de bibliotecas dinámicas es automático al invocar el procedimiento

3) Indicar las regiones de memoria existentes para un proceso ejecutando el programa P1, en los siguientes casos, así como su tamaño en bytes y número de páginas, su fuente, sus permisos y su soporte. Indicar también en qué regiones se encuentran las distintas variables que aparecen en el código:
   a) **[0.75 puntos] Antes** de ejecutar la línea 13.
   b) **[0.5 puntos] Después** de ejecutar la línea 14.
   c) **[0.5 puntos] Después** de ejecutar la línea L.

Solución:

1) En este escenario existen dos tipos de comunicación. En la primera, los procesos que ejecutan los programas P1 y P3 se comunicarán usando un modelo cliente-servidor, ya que la gestión del marcador es centralizada. Además, el proceso servidor será dedicado, ya que se indica que se creará un proceso hijo (servidor dedicado) para atender a cada uno de los procesos que ejecutan P1 (clientes). Con respecto al mecanismo de comunicación, los procesos que ejecutan P1 y P3, lo harán en máquinas remotas, por lo que se deberán comunicar usando sockets de tipo INET y, si se desea que la comunicación sea fiable, deberá usarse protocolo TCP y, en caso contrario, podría usarse protocolo UDP. En la segunda, los procesos que ejecuten los programas P2 y P3 se comunicarán usando un fichero a través del cual se intercambiarán la información del marcador.

2) En este escenario, el problema principal de concurrencia se da a la hora de leer y escribir el fichero con la información extraída del marcador. A este fichero podrán estar accediendo en lectura N procesos (ejecutando el programa P3) y un proceso en escritura (ejecutando P2). Este problema se puede considerar como un problema clásico de lectores-escritores donde puede haber N lectores pero sólo un escritor.
Como el mecanismo de comunicación en este caso es un fichero, se podría solventar el problema de concurrencia usando cerrojos sobre ficheros. En este caso, como la lectura y escritura del marcador es completa, sería necesario bloquear el fichero completo. El proceso escritor (P2) tendrá que solicitar un cerrojo exclusivo, de forma que se garantice que ningún escritor pueda estar leyendo al mismo tiempo. Por otro lado, los procesos lectores (P3) podrán usar un cerrojo compartido. El problema que puede surgir en este tipo de esquema, es que el escritor sufra de inanición ya que, si llegan procesos lectores de forma continuada el proceso escritor no conseguirá nunca el cerrojo.

El código necesario sería el siguiente (se ha obviado el control de errores):

P3:

```
int main(int argc, char *argv[])
{
  ...
  fich_fd = open("nombre_del_fichero_marcador", O_RDONLY);
  struct flock fl;
  fl.l_type = F_RDLCK;
  fl.l_whence = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 0;
  fl.l_pid = getpid();
  fcntl(fich_fd, F_SETLKW, &fl);
  leer_fich_marcador(&marcador, fich_fd);
  fl.l_type = F_UNLCK;
  fcntl(fich_fd, F_SETLK, &fl);
  enviar_marcador(marcador);
  ...
}
```

P2:

```
int main(int argc, char *argv[])
{
  ...
  leer_dispositivo_marcador(&marcador);
  fich_fd = open("nombre_del_fichero_marcador", O_RDWR);
  struct flock fl;
  fl.l_type = F_WRLCK;
  fl.l_whence = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 0;
  fl.l_pid = getpid();
```

```
        fcntl(fich_fd, F_SETLKW, &fl);
        escribir_fich_marcador(marcador, fich_fd);
        fl.l_type = F_UNLCK;
        fcntl(fich_fd, F_SETLK, &fl);

        ...
    }
```

3)

a) Se crean las siguientes regiones

NOMBRE	TAM BYES	NÚM. PÁGS.	FUENTE	PERMISOS	SOPORTE
Texto P1	15677	4	Fich. ejec.	R-X	Fich. ejec.
DVI P1	30	1	Fich. ejec.	RW-	Swap
Pila (Entorno+sizeof(argc) +∑(strlen(argv[i] +1)+sizeof(struct Marcador *)	500 + 4 + ∑(strlen(argv[i]+1) + 8	1	Rellenar a 0s	RW-	Swap

b) Se añaden las siguientes regiones:

NOMBRE	TAM BYES	NÚM. PÁGS.	FUENTE	PERMISOS	SOPORTE
Heap	40+40+4+4 = 88	1	Rellenar a 0s	RW-	Swap
Texto libc.so	11567	3	Fich. bib.	R-X	Fich. bib.
DVI libc.so	5432	2	Fich. bib.	RW-	Swap
DSVI libc.so	3246	1	Rellenar a 0s	RW-	Swap
Texto libstring.so	3567	1	Fich. bib.	R-X	Fich. bib.
DVI libstring.so	8901	3	Fich. bib.	RW-	Swap
DSVI libstring.so	1234	1	Rellenar a 0s	RW-	Swap

c) No habría ningún cambio porque la biblioteca libstring.so ya estaría montada y porque la liberación de uso de memoria de heap no conlleva un decremento del tamaño de dicha región.

Sistemas Operativos 5º Semestre. Grado Ingeniería Informática

Primer Parcial. Procesos. 16 de octubre de 2019. Ejercicio

Dispone de 25 minutos. Publicación de notas: 28 de octubre. Revisión: 30 de octubre 16:00h.

Ejercicio 1) (6 puntos)

```
 1. #include <stdio.h>
 2. #include <stdlib.h>
 3. #include <unistd.h>

 4. int main(int argc, char * argv[])
 5. {
 6.     int pid, pid2, status, i;
 7.     for (i=1; i<argc; i++)
 8.     {
 9.         switch(pid = fork())
10.         {
11.             case -1:
12.                 perror("fork");
13.                 break;
14.             case 0:
15.                 switch(pid2 = fork())
16.                 {
17.                     case -1:
18.                         perror("fork");
19.                         break;
20.                     case 0:
21.                         execlp(argv[i],argv[i],NULL);
22.                         perror("exec");
23.                         exit(1);
24.                     default:
25.                         wait(&status);
26.                         exit(0);
27.                 }
28.             default:
29.                 break;
30.         }
31.     }
32.     return 0;
33. }
```

Dado el código del enunciado (correspondiente al programa *ejecutor*), se compila correctamente y se ejecuta con los siguientes argumentos:

./ejecutor ./prog1 ./prog2 ./prog3 ./prog4 ./prog5

Teniendo en cuenta que los programas *prog1* a *prog5* son correctos, y que la ejecución del programa *ejecutor* se produce sin errores, se pide responder a las siguientes preguntas:

1.- (1 punto) Dibujar la jerarquía de procesos que se genera.

2.- (1 punto) Durante la ejecución del programa *ejecutor*, ¿podría quedarse algún proceso *zombie*? Razona la respuesta, y en caso afirmativo, implementa el código necesario para que esto no suceda.

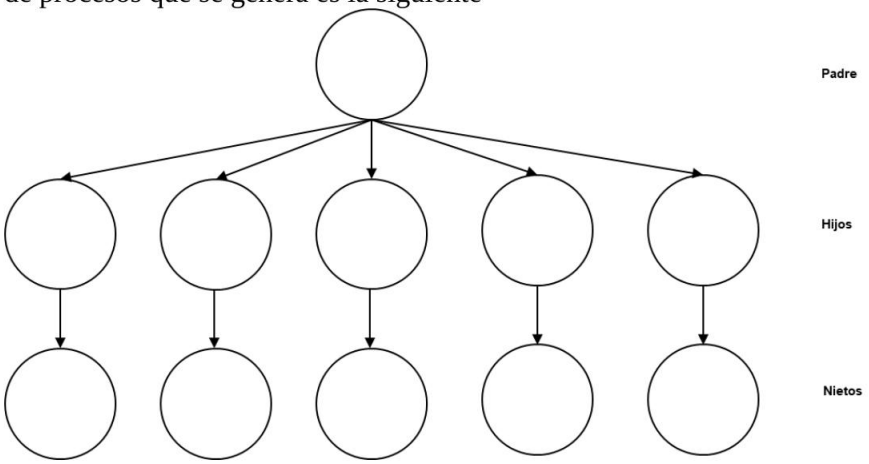
3.- (1 punto) Se desea temporizar la ejecución de los programas *prog1* a *prog5*. De esta forma, en caso de que su ejecución dure más de 10 segundos, deberemos matar al proceso correspondiente. Implementa la solución necesaria.

4.- (1,5 puntos) Se desea proteger a los programas *prog1* a *prog5* de las señales SIGINT y SIGQUIT. Responde a las siguientes cuestiones: (i) ¿Sería posible implementar una solución basada en la llamada *sigaction*?, (ii) ¿Sería posible implementar una solución basada en la llamada *sigprocmask*? Razona las respuestas. Implementa el código necesario para proteger únicamente a estos programas, sin proteger al resto de los procesos de la jerarquía.

5.- (1,5 puntos) En caso de que cualquiera de los programas de *prog1* a *prog5* muera por la recepción de una señal, se pide imprimir un mensaje por pantalla indicándolo. Implementa la solución necesaria.

SOLUCIÓN

1.- La jerarquía de procesos que se genera es la siguiente



2.- Efectivamente podría darse la situación de procesos *zombie*. Si bien los procesos hijos hacen un *wait* a la espera de los procesos nietos, el proceso padre no hace un *wait* por sus 5 procesos hijo. En el primer *switch* definido en el bucle *for*, el padre simplemente hace un break para salir del *switch*, sin operaciones adicionales.

Cualquiera de la siguientes opciones es válida:
(i) Añadir en la línea 29
wait(&status);
Si bien esta solución es válida, presenta el inconveniente de que la llamada es bloqueante, por lo hasta que no finalice el correspondiente hijo, no se haría la siguiente iteración del bucle for, ralentizando la ejecución del programa completo.
(ii) Añadir en la línea 29
waitpid(pid, &status, WNOHANG);
Esta solución es más adecuada, pues no es una llamada bloqueante, lo que permite seguir ejecutando el bucle for sin esperar por cada uno de los hijos.
(iii) Dejar el bucle for tal y como está, y añadir en la línea 32

```
 for (i=1; i<argc; i++)
    wait(&status);
```

3.-

```
void tratar_alarma(int n) // Línea 4
{
        kill(pid2, SIGKILL);
}

struct sigaction act;  // Línea 6

act.sa_handler = &tratar_alarma; // Línea 25
act.sa_flags = 0;
sigaction(SIGALRM, &act, NULL);
alarm(10);
wait(&status);
exit(0);
```

Además, la variable pid2 deberíamos declararla global, y no local en el main.

4.-
(i) **No es posible implementar una solución basada en *sigaction***, pues el armado o el ignorado de las señales que se realizara, no se heredarían después de exec.
(ii) **Sí es posible una solución basada en *sigprocmask***, pues con la máscara de señales aseguramos que las señales no llegan al proceso, y además la máscara se hereda después de realizar el exec.

```
sigset_t mascara; // Línea 21
sigemptyset(&mascara);
sigaddset(&mascara, SIGINT);
sigaddset(&mascara, SIGQUIT);
sigprocmask(SIG_BLOCK, &mascara, NULL);
```

5.-
```
if (WIFSIGNALED(status)) // Línea 26
        printf("El proceso ha muerto por la recepción de una señal");
```

Sistemas Operativos 5º Semestre. Grado II

Segundo Parcial. Sistema de Ficheros. 13 de noviembre de 2019

Dispone de 25 minutos. Publicación: 25 de noviembre o antes. Revisión: 27 de noviembre o antes

Sea un sistema de ficheros UNIX con las siguientes características:

- Tamaño de bloque: 1024 bytes.
- Tamaño de agrupación: 1 bloque.
- Campos del nodo-i: Atributos (permisos), tipo de elemento: fichero regular (fich) o directorio (dir), usuario, grupo, número de enlaces, localización (agrupaciones)

Además, se tiene la siguiente visión parcial del contenido del sistema de ficheros:

nodo-i	permisos	tipo	usuario	grupo	enlaces	Localización
2	rwx r-x r-x	dir	root	root	**xxx**	100
3	rwx r-x r-x	dir	root	root	**yyy**	101
4	rwx rwx ---	dir	juan	graf	2	102
5	rwx r-x ---	dir	blanca	graf	2	103
6	rwx r-x ---	dir	irene	prog	2	104
7	rwx r-s ---	fich	juan	graf	1	107,108
8	rw- rw- rw-	fich	juan	graf	1	106
9	rw- --- ---	fich	blanca	graf	**zzz**	105

Agrup 100		Agrup 101		Agrup 102		Agrup 103		Agrup 104		Agrup 105	Agrup 106
.	2	.	3	.	4	.	5	.	6	1234567890	123abc456def
..	2	..	2	..	3	..	3	..	3		
home	3	juan	4	ejecutor	7	fich2.txt	9	dat.txt	21		
tmp	15	blanca	5	fich.txt	8	new	12	real	18		
bin	37	irene	6			increm.dat	19	fich2.txt	9		

Responda a las siguientes preguntas

a) (i) Indica los valores **xxx** **yyy** y **zzz** de la primera tabla del sistema de ficheros.
(ii) Pinta la estructura del árbol de ficheros/directorios resultante.

b) Considere que el usuario *blanca* (del grupo *graf*) invoca, desde su home, el siguiente mandato:
/home/juan/ejecutor ./fich2.txt
(i) ¿Podrá ponerse en ejecución dicho mandato?
(ii) ¿Qué identidad tendrá el proceso resultante?

c) El usario irene del grupo prog ejecuta el siguiente mandato:
rm /home/irene/fich2.txt
Describa, paso a paso, cómo se realizaría en el sistema de ficheros

d) El usuario blanca invoca el mandato **/home/juan/ejecutor ./fich2.txt** desde su home.
Parte del código del programa ejecutor es el siguiente:

Proceso	**código ejecutado**

ejecutor	umask(0027);
	fd = creat(argv[1], 0666);
	n = write(fd, "0123456789ABCDEFGHJK", 20);
	fd2 = open(argv[1], O_RDWR);
	n = write(fd2, "0123456789", 5);

(i) ¿Qué sucede en la llamada creat? ¿Con qué permisos se queda el fichero fich2.txt?
(ii) Indique la situación en la que se encuentran las tablas internas del sistema operativo relacionadas con la gestión de ficheros (tablas de descriptores, tabla intermedia y tabla de copia de nodos-i) después de la ejecución del código descrito, suponiendo que no se produce error alguno.
(iii) Implemente el código necesario, para que, escribiendo un único byte adicional en el fichero pasado como primer argumento al programa (argv[1]), el tamaño del mismo se quede en 10000 bytes. ¿Cuántas agrupaciones ocuparía el fichero después de esta operación?

Evaluación: a) 1 punto, b) 1 punto, c) 1.5 puntos, d) 2.5 puntos

SOLUCIÓN

a) (i)
xxx=3 (5 si contamos con los directorios tmp y bin)
yyy=5
zzz=2

(ii)
/
/tmp
/bin
/home
    /home/juan
        /home/juan/ejecutor
        /home/juan/fich.txt
    /home/blanca
        /home/blanca/fich2.txt
        /home/blanca/new
        /home/blanca/increm.dat
    /home/irene
        /home/irene/dat.txt
        /home/irene/real
        /home/irene/fich2.txt

b) (i) El usuario blanca del grupo graf invoca desde su home un mandato con una dirección absoluta.
Tras comprobar la ruta completa tenemos permiso de atravesar (x) en todos los directorios necesarios. Además, el fichero ejecutor (nodo-i 7) tiene permisos rwx r-s ---. De esta forma, aplicando los permisos de grupo (r-s) tenemos permiso de ejecución, y además se puede apreciar que está activo el bir SETGID.
En formato [nodo-i, permiso, terna]
[2,x,otros]
[3,x,otros]
[4,x,grupo]
[7,s,grupo] => SETUID activo

No es necesario hacer comprobación alguna sobre fich2.txt, pues es solo un argumento de llamada al programa.

(ii)
UID real=blanca
GID real=graf
UID efectivo=blanca (copia del UID real)
GID efectivo=graf (GID del propietario del fichero, que en este caso coincide con GID real)

c) Permisos aplicados
[2,x,otros]
[3,x,otros]
[6,x,propietario]
Una vez en el directorio, debemos tener permisos de escritura para poder borrar una entrada de directorio [6,w,propietario]. De esta forma, e independientemente de los permisos que tenga dicho fichero, podemos borrar la entrada del directorio.
Debido a que estamos ante la presencia de un enlace físico, en el nodo-i 9 vendrá como número de enlaces 2. Al hacer el rm se realizan las siguientes operaciones:
- Se borra la entrada de directorio /home/irene/fich2.txt
- Se decrementa al número de enlaces que pasará a ser 1
- No se libera ningún nodo-i, ni agrupación

d) (i) En la llamada creat el fichero ya existe, por lo tanto, simplemente se trunca (se deja de tamaño 0), se abre en modo de solo escritura, y los permisos no se tocan, por lo que seguirá teniendo permisos (0600).

(ii) El estado de las tablas internas del sistema operativo será el siguiente (se ha supuesto que en la tabla intermedia los primeros valores disponibles empezaban en el 11).

BCP ejecutor

Tabla fd

0	aaa
1	bbb
2	ccc
3	11
4	12
5	
6	
7	

Tabla intermedia

	nodo-i	Puntero	nRefs	Permisos
11	9	20	1	-W
12	9	5	1	RW
13				
14				
15				
16				
17				
18				

Tabla copia nodos i

nodo-i	nOpens
9	2

(iii)

Para que el fichero quede con el tamaño de 10.000B, haciendo una única escritura de 1B, se podría por ejemplo hacer con el siguiente código:

```
lseek(fd,9999,SEEK_SET);
write(fd,”1”,1);
```

El número de agrupaciones que ocuparía sería de 2, pues el sistema de ficheros es capaz de optimizar estas situaciones, y no usar agrupaciones que están todas rellenas a cero hasta que sea necesario.

EJERCICIO 1 (3 ptos)

Dado el siguiente código, correspondiente a *pipeline.c*, donde, por simplicidad, se han eliminado los ficheros de cabecera necesarios y cuyo ejecutable es *pipeline*:

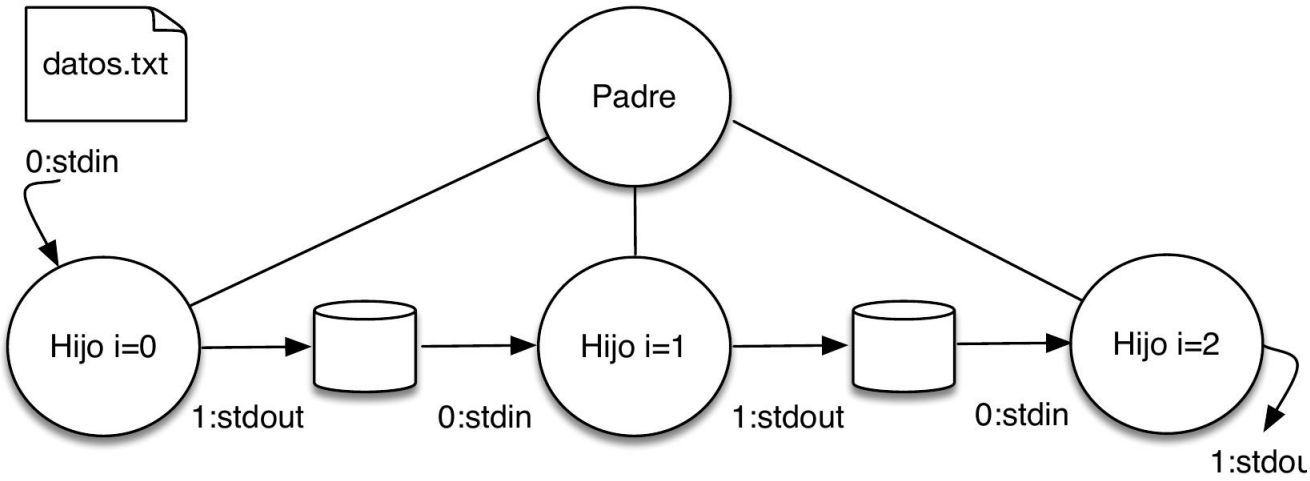
```
 1: int main(int argc, char const *argv[]) {
 2:     int n, pid, i;
 3:     int tub_ant_lectura, tub_post[2], sal_est, fd;
 4:
 5:     if (argc!=3){
 6:             fprintf(stderr, "Error. Uso: ./pipeline num_hijos ejec\n");
 7:             return 1;
 8:     }
 9:
10:     sal_est=dup(1);
11:     n = atoi(argv[1]);
12:
13:     for (i=0; i<n; i++){
14:             if (i != 0){
15:                     tub_ant_lectura=tub_post[0];
16:                     dup2(tub_ant_lectura,0);
17:                     close(tub_ant_lectura);
18:             }
19:             if (i != n-1){
20:                     pipe(tub_post);
21:                     dup2(tub_post[1],1);
22:                     close(tub_post[1]);
23:             }
24:             if (i == n-1){
25:                     dup2(sal_est,1);
26:                     close(sal_est);
27:             }
28:             pid = fork();
29:             if (pid == 0){
30:                     if (i==0){
31:                             fd = open("./datos.txt", O_RDONLY);
32:                             dup2(fd,0);
33:                             close(fd);
34:                     }
35:                     if (i != n-1){
36:                             close(tub_post[0]);
37:                             close(sal_est);
38:                     }
39:                     execlp(argv[2],argv[2],NULL);
40:                     perror("exec");
41:                     return 2;
42:             }
43:     }
44:     return 0;
45:}
```

Se pide:

1) **[0.5 puntos]** Para la ejecución de la siguiente secuencia "`./pipeline 3 sort`", dibujar el diagrama de procesos, dibujando también las tuberías, así como indicando la entrada y salida estándar de cada proceso.
2) **[0.5 puntos]** Para la ejecución anterior y en el caso de que el fichero "`datos.txt`" contenga tres líneas con los siguientes caracteres: `abcdefg, cdefghi, bcdefgh`, indicar qué se imprimirá por la salida estándar.
3) **[0.5 puntos]** Realizar el mismo ejercicio anterior, pero para la ejecución de la secuencia "`./pipeline 3 cat`".
4) **[0.25 puntos]** Indicar para qué se utiliza la variable `sal_est` en el código.
5) **[0.25 puntos]** Justificar si puede haber procesos huérfanos y/o procesos zombies.
6) **[1 punto]** Modificar el código para que el proceso inicial espere por el último hijo que cree. En caso de que dicho proceso hijo no finalice antes de 10 segundos, el proceso padre debería matarlo. Mientras realiza la espera por el proceso hijo, el proceso padre no debería ser interrumpido por una señal de atención interactiva (ctrl+C).

SOLUCIÓN

1)



2) Como los 3 procesos hijos se comunican a través de tuberías, “filtrando” los datos de entrada mediante el mandato `sort`, lo que se imprimirá al final serán los datos procedentes del fichero, pero ordenados. Por tanto, la salida sería:

```
abcdefg
bcdefgh
cdefghi
```

3) Al igual que en el apartado anterior, los procesos se comunican a través de tuberías, “filtrando” los datos de entrada, pero en esta ocasión mediante el mandato `cat`, por lo que la salida no vendría ordenada alfabéticamente, sino en el orden en el que aparecen en el fichero. Por tanto, la salida sería:

```
abcdefg
cdefghi
bcdefgh
```

4) La variable `sal_est` se utiliza para guardar la salida estándar inicial, y reponerla en el último proceso hijo creado.

5) Todos los procesos hijos pueden ser huérfanos, si el proceso padre muere antes que los procesos hijos. También podrían ser zombies, si los procesos hijos mueren antes que el padre, ya que éste no hace un `wait()` por ellos.

6) El código sería:

```
pid_t pid;

void tratar_senyal(int sig){
       kill(pid, SIGKILL);
}

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

       int n, i;
       int tub_ant_lectura, tub_post[2], sal_est, fd;
       struct sigaction accion;
       sigset_t senyales;

       . . . // mismo código anterior
```

```
                        execlp(argv[2],argv[2],NULL);
                        perror("exec");
                        return 2;
                }
                else {
                        if (i == n-1){
                                        sigemptyset(&senyales);
                                sigaddset(&senyales, SIGINT);
                                sigprocmask(SIG_BLOCK, &senyales, NULL);
                                accion.sa_handler = &tratar_senyal;
                                accion.sa_flags = SA_RESTART;
                                sigaction(SIGALRM, &accion, NULL);
                                alarm(10);
                                waitpid(pid,NULL,0);
                                sigprocmask(SIG_UNBLOCK, &senyales, NULL);
                        }
                }
        }
        return 0;
}
```

EJERCICIO 2 (3 ptos)

a) (1 pto) Sea un sistema de ficheros tipo Unix (basado en inodos) para un dispositivo orientado a bloque de 3 TiB brutos. Si el espacio asignado a ficheros se gestiona en unidades de 2 KiB, ¿cuántas direcciones cabrán en el espacio apuntado por el puntero simple indirecto del inodo? Detalle cada paso de sus cálculos y las unidades.

b) (1 pto) ¿Qué optimización ofrece el Gestor de Memoria del Sistema Operativo que permite hacer la implementación del servicio FORK mucho más eficiente? Explique qué se ahorra, comparando la implementación con y sin dicha optimización.

c) (1pto) Implemente en C y para UNIX int CountRegs(int fd,int RegSize,int RegType); que dado un descriptor de fichero (fd), que se sabe que contiene registros del tamaño dado (RegSize), retorna cuántos registros del tipo dado (RegType) contiene. Se entiende que los registros tienen un primer campo entero que indica el tipo de registro.
Dada su solución, razone que restricciones (si las hubiera) impone a la naturaleza del objeto asociado al descriptor dado y al tamaño máximo de dicho objeto.

SOLUCIÓN

a)
3TiB < 2^42 (B/dev) ==> 2^(42-11) (agr/dev) = 2^31 ==> Valen direcciones de 32 bits (4B).
El número de direcciones en cada agrupación de indirección serán: 2^(11-2) = 2^9 (dir/agr).

b)
Es la técnica de copy on write (COW) que consiste en que se marcan todas las páginas de la región a copiar como COW, y se copian las tablas de páginas con lo que los procesos comparten los mismos marcos de páginas y páginas de intercambio, pero cada uno a través de su propia tabla de páginas. Seguidamente, si un proceso intenta escribir en la región la MMU producirá una excepción que trata el sistema operativo y, al reconocer que se trata de una página en COW, se encarga de desdoblar la página afectada, copiándola en un nuevo marco de memoria. De esta forma, cada proceso mantiene la información que le es propia, sin afectar a los demás.

La ejecución del servicio fork exige que la imagen de memoria del proceso hijo sea una copia de la de su padre.
Sin la técnica COW sería necesario realizar una copia explícita y completa de cada página de cada región privada del proceso padre (se vaya a intentar modificar en el futuro esta memoria o no) requiriendose espacio de swap, operaciones de E/S y considerable tiempo.
Con COW basta con compartir todas las regiones de padre y marcar las privadas como COW. Como resultado de esta optimización, en vez de duplicar el espacio de memoria sólo se duplica la tabla de páginas.

c)
```
int CountRegs(int fd, int RegSize, int RegType)
{
	int ret;
	int FileSize, NumRegs;
	void *ptr;
	struct stat st;
	ret = fstat(fd, &st);
	if (ret < 0) { perror("stat"); return -1; }
	FileSize = st.st_size;
	ptr = mmap(NULL, FileSize, PROT_READ, MAP_PRIVATE, fd, 0);
	if (ptr == MAP_FAILED) { perror("mmap"); return -2; }
	ret = 0;
	NumRegs = FileSize / RegSize;
	while(NumRegs > 0)
	{
		if (*(int*)ptr == RegType) ret++;
		ptr += RegSize;
		NumRegs--;
	}
	munmap(ptr, FileSize);
	return ret;
}
```

EJERCICIO 1 (3 ptos)

Dado el siguiente código, correspondiente a *pipeline.c*, donde, por simplicidad, se han eliminado los ficheros de cabecera necesarios y cuyo ejecutable es *pipeline*:

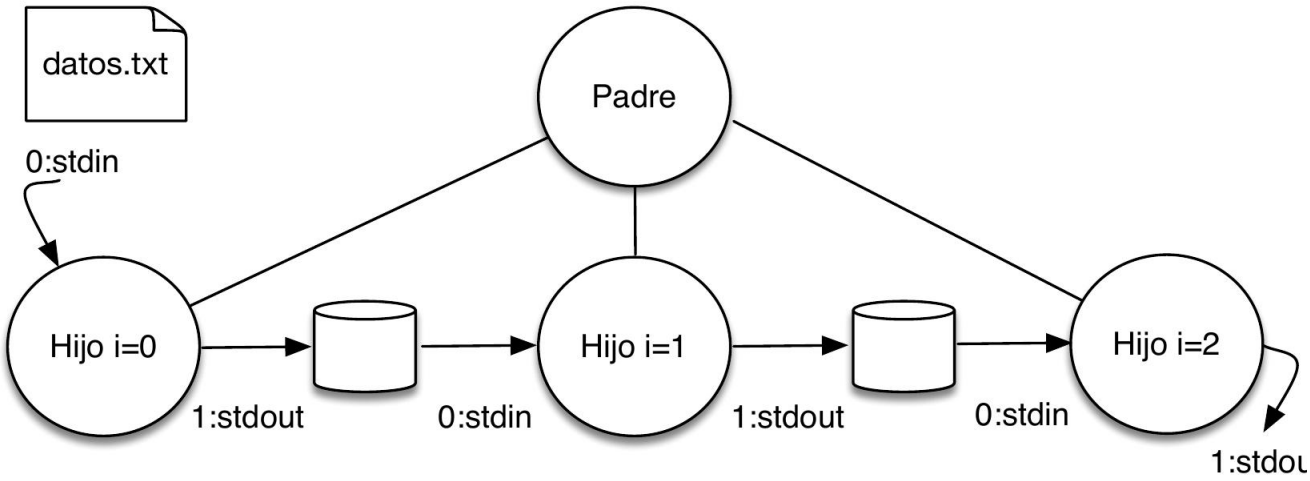
```
 1: int main(int argc, char const *argv[]) {
 2:     int n, pid, i;
 3:     int tub_ant_lectura, tub_post[2], sal_est, fd;
 4:
 5:     if (argc!=3){
 6:             fprintf(stderr, "Error. Uso: ./pipeline num_hijos ejec\n");
 7:             return 1;
 8:     }
 9:
10:     sal_est=dup(1);
11:     n = atoi(argv[1]);
12:
13:     for (i=0; i<n; i++){
14:             if (i != 0){
15:                     tub_ant_lectura=tub_post[0];
16:                     dup2(tub_ant_lectura,0);
17:                     close(tub_ant_lectura);
18:             }
19:             if (i != n-1){
20:                     pipe(tub_post);
21:                     dup2(tub_post[1],1);
22:                     close(tub_post[1]);
23:             }
24:             if (i == n-1){
25:                     dup2(sal_est,1);
26:                     close(sal_est);
27:             }
28:             pid = fork();
29:             if (pid == 0){
30:                     if (i==0){
31:                             fd = open("./datos.txt", O_RDONLY);
32:                             dup2(fd,0);
33:                             close(fd);
34:                     }
35:                     if (i != n-1){
36:                             close(tub_post[0]);
37:                             close(sal_est);
38:                     }
39:                     execlp(argv[2],argv[2],NULL);
40:                     perror("exec");
41:                     return 2;
42:             }
43:     }
44:     return 0;
45:}
```

Se pide:

1) **[0.5 puntos]** Para la ejecución de la siguiente secuencia "`./pipeline 3 sort`", dibujar el diagrama de procesos, dibujando también las tuberías, así como indicando la entrada y salida estándar de cada proceso.
2) **[0.5 puntos]** Para la ejecución anterior y en el caso de que el fichero "`datos.txt`" contenga tres líneas con los siguientes caracteres: `abcdefg, cdefghi, bcdefgh`, indicar qué se imprimirá por la salida estándar.
3) **[0.5 puntos]** Realizar el mismo ejercicio anterior, pero para la ejecución de la secuencia "`./pipeline 3 cat`".
4) **[0.25 puntos]** Indicar para qué se utiliza la variable `sal_est` en el código.
5) **[0.25 puntos]** Justificar si puede haber procesos huérfanos y/o procesos zombies.
6) **[1 punto]** Modificar el código para que el proceso inicial espere por el último hijo que cree. En caso de que dicho proceso hijo no finalice antes de 10 segundos, el proceso padre debería matarlo. Mientras realiza la espera por el proceso hijo, el proceso padre no debería ser interrumpido por una señal de atención interactiva (ctrl+C).

SOLUCIÓN

1)



2) Como los 3 procesos hijos se comunican a través de tuberías, “filtrando” los datos de entrada mediante el mandato `sort`, lo que se imprimirá al final serán los datos procedentes del fichero, pero ordenados. Por tanto, la salida sería:

```
abcdefg
bcdefgh
cdefghi
```

3) Al igual que en el apartado anterior, los procesos se comunican a través de tuberías, “filtrando” los datos de entrada, pero en esta ocasión mediante el mandato `cat`, por lo que la salida no vendría ordenada alfabéticamente, sino en el orden en el que aparecen en el fichero. Por tanto, la salida sería:

```
abcdefg
cdefghi
bcdefgh
```

4) La variable `sal_est` se utiliza para guardar la salida estándar inicial, y reponerla en el último proceso hijo creado.

5) Todos los procesos hijos pueden ser huérfanos, si el proceso padre muere antes que los procesos hijos. También podrían ser zombies, si los procesos hijos mueren antes que el padre, ya que éste no hace un `wait()` por ellos.

6) El código sería:

```
pid_t pid;

void tratar_senyal(int sig){
       kill(pid, SIGKILL);
}

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

       int n, i;
       int tub_ant_lectura, tub_post[2], sal_est, fd;
       struct sigaction accion;
       sigset_t senyales;

       . . . // mismo código anterior
```

```
                        execlp(argv[2],argv[2],NULL);
                        perror("exec");
                        return 2;
                }
                else {
                        if (i == n-1){
                                        sigemptyset(&senyales);
                                sigaddset(&senyales, SIGINT);
                                sigprocmask(SIG_BLOCK, &senyales, NULL);
                                accion.sa_handler = &tratar_senyal;
                                accion.sa_flags = SA_RESTART;
                                sigaction(SIGALRM, &accion, NULL);
                                alarm(10);
                                waitpid(pid,NULL,0);
                                sigprocmask(SIG_UNBLOCK, &senyales, NULL);
                        }
                }
        }
        return 0;
}
```

EJERCICIO 2 (3 ptos)

a) (1 pto) Sea un sistema de ficheros tipo Unix (basado en inodos) para un dispositivo orientado a bloque de 3 TiB brutos. Si el espacio asignado a ficheros se gestiona en unidades de 2 KiB, ¿cuántas direcciones cabrán en el espacio apuntado por el puntero simple indirecto del inodo? Detalle cada paso de sus cálculos y las unidades.

b) (1 pto) ¿Qué optimización ofrece el Gestor de Memoria del Sistema Operativo que permite hacer la implementación del servicio FORK mucho más eficiente? Explique qué se ahorra, comparando la implementación con y sin dicha optimización.

c) (1pto) Implemente en C y para UNIX int CountRegs(int fd,int RegSize,int RegType); que dado un descriptor de fichero (fd), que se sabe que contiene registros del tamaño dado (RegSize), retorna cuántos registros del tipo dado (RegType) contiene. Se entiende que los registros tienen un primer campo entero que indica el tipo de registro.
Dada su solución, razone que restricciones (si las hubiera) impone a la naturaleza del objeto asociado al descriptor dado y al tamaño máximo de dicho objeto.

SOLUCIÓN

a)
3TiB < 2^42 (B/dev) ==> 2^(42-11) (agr/dev) = 2^31 ==> Valen direcciones de 32 bits (4B).
El número de direcciones en cada agrupación de indirección serán: 2^(11-2) = 2^9 (dir/agr).

b)
Es la técnica de copy on write (COW) que consiste en que se marcan todas las páginas de la región a copiar como COW, y se copian las tablas de páginas con lo que los procesos comparten los mismos marcos de páginas y páginas de intercambio, pero cada uno a través de su propia tabla de páginas. Seguidamente, si un proceso intenta escribir en la región la MMU producirá una excepción que trata el sistema operativo y, al reconocer que se trata de una página en COW, se encarga de desdoblar la página afectada, copiándola en un nuevo marco de memoria. De esta forma, cada proceso mantiene la información que le es propia, sin afectar a los demás.

La ejecución del servicio fork exige que la imagen de memoria del proceso hijo sea una copia de la de su padre.
Sin la técnica COW sería necesario realizar una copia explícita y completa de cada página de cada región privada del proceso padre (se vaya a intentar modificar en el futuro esta memoria o no) requiriendose espacio de swap, operaciones de E/S y considerable tiempo.
Con COW basta con compartir todas las regiones de padre y marcar las privadas como COW. Como resultado de esta optimización, en vez de duplicar el espacio de memoria sólo se duplica la tabla de páginas.

c)
```
int CountRegs(int fd, int RegSize, int RegType)
{
	int ret;
	int FileSize, NumRegs;
	void *ptr;
	struct stat st;
	ret = fstat(fd, &st);
	if (ret < 0) { perror("stat"); return -1; }
	FileSize = st.st_size;
	ptr = mmap(NULL, FileSize, PROT_READ, MAP_PRIVATE, fd, 0);
	if (ptr == MAP_FAILED) { perror("mmap"); return -2; }
	ret = 0;
	NumRegs = FileSize / RegSize;
	while(NumRegs > 0)
	{
		if (*(int*)ptr == RegType) ret++;
		ptr += RegSize;
		NumRegs--;
	}
	munmap(ptr, FileSize);
	return ret;
}
```