Programación de Sistemas con

Ansi C sobre UNIX

Pedro Merino Gómez
Jesus Martínez Cruz

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Programación de Sistemas

- **Llamadas al sistema**
  - **Gestión de errores**
  - **Manejo de ficheros y directorios**
  - **Control de procesos**
  - **Manejo de señales**
  - **Comunicación entre procesos con sockets**

- **Algunos ejemplos extraidos de**
- `http://users.actcom.co.il/~choo/lupg/tutorials/index.html`

Gestión de errores

```
#include  <stdio.h>
void perror(const char *s)
#include <errno.h>
int errno;
```

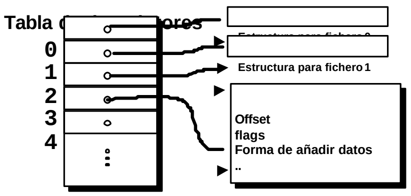
- **Las llamadas al sistema pueden devolver errores (-1).**
- **El código de error se examina en una variable concreta**

Gestión de errores

```
if (fd_read < 0) {
    perror("open");
    exit(1);
}
```

Ficheros regulares

Un proceso puede disponer de 20 descriptores de ficheros. Los tres primeros se abren al comenzar la ejecución.



Ficheros regulares

```
int open(const char *path, int oflag,
           /* mode_t mode */...);
Oflag:
 O_RDONLY   O_WRONLY  O_RDWR  O_APPEND
```

- **oflag es el OR de los indicadores**
- **Hay otras funciones estándar de C (fopen(), ..)**

Ficheros regulares

```
  int     fd;              /* descriptor fichero      */
…...

  if ( (fd = open(“ejemplo”, O_RDONLY)) < 0) {
                fprintf(stderr,“open failed\n");
                exit(1);
  }

if ( (fd = open(“ejemplo”, O_WRONLY | O_TRUNC | O_CREAT, 0600)) < 0) {
                fprintf(stderr,“creation failed\n");
                exit(1);
  }
```

Ficheros regulares

```
/* these hold file descriptors returned from open(). */
int fd_read;
int fd_write;
int fd_readwrite;
int fd_append;

/* Open the file /etc/passwd in read-only mode. */
fd_read = open("/etc/passwd", O_RDONLY);
if (fd_read < 0) {
    perror("open");
    exit(1);
}

/* Open the file run.log (current directory) in write-only mode. */
/* and truncate it, if it has any contents.                      */
fd_write = open("run.log", O_WRONLY | O_TRUNC);
if (fd_write < 0) {
    perror("open");
    exit(1);
}
```

Ficheros regulares

```
/* Open the file /var/data/food.db in read-write mode. */
fd_readwrite = open("/var/data/food.db", O_RDWR);
if (fd_readwrite < 0) {
    perror("open");
    exit(1);
}

/* Open the file /var/log/messages in append mode. */
fd_append = open("/var/log/messages", O_WRONLY | O_APPEND);
if (fd_append < 0) {
    perror("open");
    exit(1);
}
```

Ficheros regulares

```
ssize_t read(int fildes, void *buf, size_t
  nbyte);

ssize_t write(int fildes, const void *buf,
  size_t nbyte);
```

- read() devuelve cero al final de fichero
- write() puede devolver un valor menor que nbyte

Ficheros regulares

```
/* return value from the read() call. */
size_t rc;

/* buffer to read data into.          */
char buf[20];

/* read 20 bytes from the file.       */
rc = read(fd, buf, 20);
if (rc == 0) {
    printf("End of file encountered\n");
}
else if (rc < 0) {
    perror("read");
    exit(1);
}
else {
    printf("read in '%d' bytes\n", rc);
}
```

Ficheros regulares

```
/* return value from the write() call. */
size_t rc;

/* write the given string to the file. */
rc = write(fd, "hello world\n", strlen("hello world\n"));
if (rc < 0) {
    perror("write");
    exit(1);
}
else {
    printf("wrote in '%d' bytes\n", rc);
}
```

Ficheros regulares

```
int close(int fildes);
```

- Cierra un fichero
- Comprobar que no devuelve -1 !!

```
if (close(fd) == -1) {
    perror("close");
    exit(1);
}
```

Ficheros regulares

```
int stat(const char *path, struct stat
  *buf);
```

- Devuelve información sobre el fichero
- Útil para recorrer directorios
- ver estructura stat en <sys/stat.h>

Ficheros regulares

```
•       mode_t   st_mode;     /* File mode (see mknod(2)) */
•       ino_t    st_ino;      /* Inode number */
•       dev_t    st_dev;      /* ID of device containing */
•                             /* a directory entry for this file */
•       dev_t    st_rdev;     /* ID of device */
•                             /* This entry is defined only for */
•                             /* char special or block special files */
•       nlink_t  st_nlink;    /* Number of links */
•       uid_t    st_uid;      /* User ID of the file's owner */
•       gid_t    st_gid;      /* Group ID of the file's group */
•       off_t    st_size;     /* File size in bytes */
•       time_t   st_atime;    /* Time of last access */
•       time_t   st_mtime;    /* Time of last data modification */
•       time_t   st_ctime;    /* Time of last file status change */
•                             /* Times measured in seconds since */
•                             /* 00:00:00 UTC, Jan. 1, 1970 */
•       long     st_blksize;  /* Preferred I/O block size */
•       blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
```

Especialmente útil: S_ISDIR(st_mode)

S_ISREG(st_mode)

Ficheros regulares

```
/* structure passed to the stat() system call, to get its results. */
struct stat file_status;

/* check the status information of file "foo.txt", and print its */
/* type on screen.                                               */
if (stat("foo.txt", &file_status) == 0) {
    if (S_ISDIR(file_status.st_mode))
        printf("foo.txt is a directory\n");
    if (S_ISLNK(file_status.st_mode))
        printf("foo.txt is a symbolic link\n");
    if (S_ISSOCK(file_status.st_mode))
        printf("foo.txt is a (Unix domain) socket file\n");
    if (S_ISREG(file_status.st_mode))
        printf("foo.txt is a normal file\n");
}
else { /* stat() call failed and returned '-1'. */
    perror("stat");
}
```

Directorios

```
int mkdir(const char *path, mode_t mode);

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

- opendir devuelve un puntero a estructure tipo DIR, para ser empleada en otras funciones

Directorios

```
/* open the directory "/home/users" for reading. */
DIR* dir = opendir("/home/users");
if (!dir) {
    perror("opendir");
    exit(1);
}


if (closedir(dir) == -1) {
    perror("closedir");
    exit(1);
}
```

Directorios

```
struct dirent *readdir(DIR *dirp);
```

●**Devuelve u puntero a estructure tipo dirent, que contiene, entre otros:**

`d_namlen` y `d_name[]`

Directorios

```
dirp = opendir(".");

while (dirp) {
    errno = 0;
    if ((dp = readdir(dirp)) != NULL) {

        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        }
    } else {
        if (errno == 0) {
            closedir(dirp);
            return NOT_FOUND;
        }
        closedir(dirp);
        return READ_ERROR;
    }
}

return OPEN_ERROR;
```

Directorios

```
 /* cdir.c program to emulate unix cd command */

/* cc -o cdir cdir.c */


#include<stdio.h>
#include<sys/dir.h>

main(int argc,char **argv)
{
        if (argc < 2)
          { printf("Usage: %s <pathname>\n",argv[0]);
            exit(1);
          }

        if (chdir(argv[1]) != 0)
          { printf("Error in \"chdir\"\n");
            exit(1);
          }
}}
```

Procesos

```
pid_t fork(void);

pid_t wait(int *stat_loc);

void exit(int status);
```

- **fork () crea imagen del proceso que llama**
- **wait() chequea la terminación de procesos hijos**

Procesos

```
#include <unistd.h>    /* defines fork(), and pid_t.      */
#include <sys/wait.h> /* defines the wait() system call. */

/* place for the pid of the child process, and its exit status. */
pid_t child_pid;
int child_status;

/* lets fork off a child process... */
child_pid = fork();

/* check what the fork() call actually did */
switch (child_pid) {
     case -1:  /* fork() failed */
     perror("fork");  /* print a system-defined error message */
   exit(1);
     case 0:   /* fork() succeeded, we're inside the child process */
     printf("hello world\n");
     exit(0);  /* here the CHILD process exits, not the parent. */
   default:    /* fork() succeeded, we're inside the parent process */
   wait(&child_status);       /* wait till the child process exits */
}
/* parent's process code may continue here... */
```

Procesos

```
int execl(const char *path, const  char  *arg0,  ...,  const char
  *argn, char * /*NULL*/);

int execv(const char *path, char *const argv[]);

int execle(const char *path, const char  *arg0,  ...,  const char
  *argn, char * /*NULL*/, char *const envp[]);

int execlp(const char *file, const char  *arg0,  ...,  const char
  *argn, char * /*NULL*/);

int execvp(const char *file, char *const argv[]);
```

- **Cambia la imagen del proceso que llama**

Procesos

```
/* fork.c - example of a fork in a program */

#include <stdio.h>

main()
{
    char buf[1024];
    char *args[64];

    for (;;) {
       printf("Command: ");
        if (gets(buf) == NULL) {
             printf("\n");
             exit(0);
        }
        parse(buf, args);

       execute(args);
    }
}
```

Procesos

```
void parse(char *buf, char **args)
{
    while (*buf != NULL) {
        while ((*buf == ' ') || (*buf == '\t'))
            *buf++ = NULL;

            *args++ = buf;
            while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
              buf++;
    }

    *args = NULL;
}
```

Procesos

```
void execute(char **args)
{
    int pid, status;
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }


    if (pid == 0) {
        execvp(*args, args);
        perror(*args);
        exit(1);

    }

    /*
     * The parent executes the wait.
     */
    while (wait(&status) != pid)
        /* empty */ ;
}
```

Señales

● En el mundo UNIX una señal es similar a una interrupción hardware.

•Es un evento asíncrono que hace saltar el flujo de ejecución.

• Su puede trabajar con señales en consola:
kill *nº_señal* *pid*

Señales

```
void (*signal (int sig, void (*disp)(int)))(int);
```

● **signal () asocia el manejador a una señal**
●**Señales usuales:** SIGKILL, SIGTERM, SIGCHLD, SIGPIPE...
●Algunas NO se pueden capturar, como SIGKILL

Señales

```
#include <stdio.h>     /* basic I/O routines.   */
#include <unistd.h>    /* define fork(), etc.   */
#include <sys/types.h> /* define pid_t, etc.    */
#include <sys/wait.h>  /* define wait(), etc.   */
#include <signal.h>    /* define signal(), etc. */

/* first, here is the code for the signal handler */
void catch_child(int sig_num)
{
    /* when we get here, we know there's a zombie child waiting
   */
    int child_status;

    wait(&child_status);
    printf("child exited.\n");
}
.
/* and somewhere in the main() function ... */
.
```

Señales

```
/* define the signal handler for the CHLD signal */
signal(SIGCHLD, catch_child);

/* and the child process forking code... */
{
    int child_pid;
    int i;
    child_pid = fork();
    switch (child_pid) {
        case -1:         /* fork() failed */
            perror("fork");
            exit(1);
        case 0:          /* inside child process  */
            printf("hello world\n");
            sleep(5);    /* sleep a little, so we'll have */*/
            exit(0);
        default:         /* inside parent process */
            break;
```

Señales

```
}
  /* parent process goes on, minding its own
business... */
  /* for example, some output...
*/
  for (i=0; i<10; i++) {
      printf("%d\n", i);
      sleep(1);    /* sleep for a second, so we'll have
time to see the mix */
  }
}
```