Fonaments dels Sistemes Operatius

Departament d'Informàtica de Sistemes i Computadores (DISCA)
Universitat Politècnica de València

fSO

Pràctica 6
Crides UNIX per a arxius
Versió 2.3

Contingut

1. Objectius

El sistema d'arxius Unix presenta una interfície única per al maneig de dispositius i arxius. Les crides al sistema relacionades amb arxius són d'ús ampli en la programació d'aplicacions. La present pràctica s'orienta principalment a l'ús de les mateixes per a la comunicació entre processos i el redireccionament de l'E/S. En concret els objectius de la pràctica són:

- Treballar les crides Unix per al maneig d'arxius: open, close, read, write, pipe i dup2.
- Estudiar el mecanisme de redireccionament de l'E/S a arxius regulars i tubs (pipe).
- Comprendre el mecanisme d'herència que permet la comunicació de processos mitjançant tubs.

2. Maneig d'arxius en Unix

El maneig d'arxius en UNIX segueix el model de sessió. Per a treballar amb arxius primer cal obrir-lo amb la crida `open()`. La funció `open)` retorna un descriptor d'arxiu (*`file descriptor`*), que és un nombre sencer positiu que identificarà a l'arxiu en futures operacions. Aquest valor és la posició de la taula de descriptors del procés que conté el punter a la taula d'apertures del sistema. Finalment cal tancar l'arxiu, amb la crida `close()`, per a alliberar els recursos assignats a l'arxiu.

Unix utilitza la mateixa interfície per a treballar amb arxius que amb dispositius d'E/S. Els descriptors 0, 1 i 2 estan establerts per als dispositius de E/S estàndard, s'hereten de processos pare a fills a través del mecanisme d'herència. El `descriptor 0` correspon a l'entrada estàndard (el teclat), el `descriptor 1` a l'eixida estàndard (la pantalla) i el `descriptor 2` al dispositiu de visualització d'errors (per defecte la pantalla). La utilització de descriptors permet al sistema ser molt més eficient en el treball amb arxius que si s'utilitzés el nom proposat pe l'usuari.

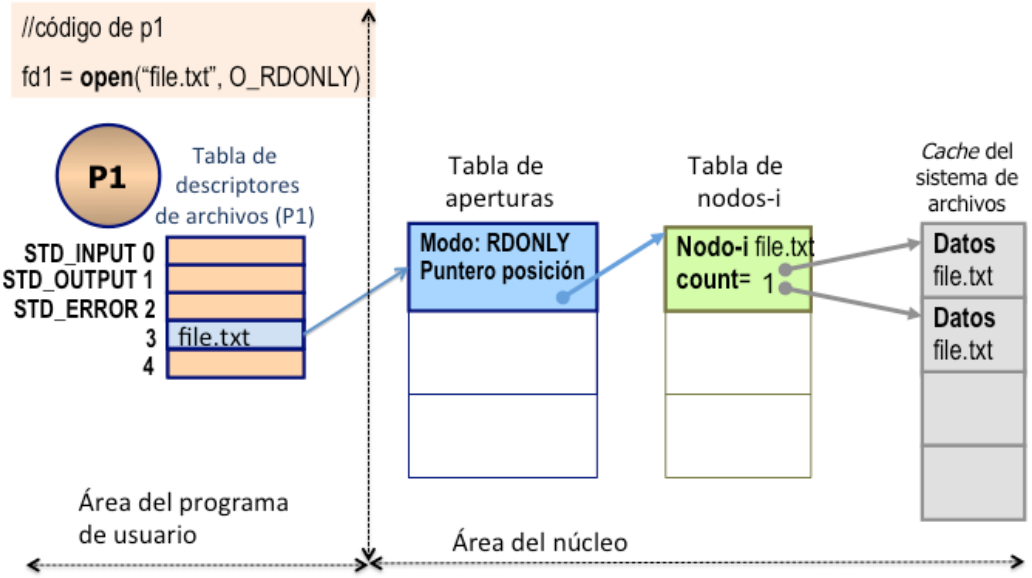

Figura 1: Taula de descriptors d'arxiu per a un procés, després de realitzar la crida *`open()`*

Unix accedeix als arxius de forma seqüencial, tot i que es pot realitzar un accés directe utilitzant la crida `lseek()`. Cada apertura d'arxiu disposa d'un punter de posició que s'incrementa amb cada lectura o escriptura un nombre de bytes igual al nombre de bytes llegits o escrits. La crida `lseek()` permet posicionar el punter en una determinada posició de l'arxiu.

3. Apertura i tancament d'arxius

La figura 1 representa gràficament l'efecte de realitzar una crida `open()`. En Unix els processos reben la taula de descriptors a través del mecanisme d'herència, on els descriptors 0, 1 i 2 corresponen a l'entrada estàndard, eixida estàndard i eixida d'error (`STDIN, STDOUT, STDERR`) respectivament. Unix utilitza la crida `open()` per a assignar un descriptor d'arxiu a un arxiu o dispositiu físic.

3.1 Exercici 1: Descriptors d'arxiu, crida `open()`

El contingut de l'arxiu *descriptor.c* proporcionat amb el material de pràctiques és el següent:

```
// descriptor.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

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

  if (argc!=2) {
      fprintf(stderr,"Required read/write file \n");
      exit(-1);
  }

  if ((fda=open(argv[1],O_RDONLY))<0)
    fprintf(stderr,"Open failed \n");
  else
    fprintf(stderr,"Read %s descriptor = %d \n",argv[1],fda);

  if ((fdb=open(argv[1],O_WRONLY))<0)
    fprintf(stderr,"Open failed \n");
  else
    fprintf(stderr,"Write %s descriptor = %d \n",argv[1],fdb);
  return(0);
}
```

Per a executar *descriptor.c* fique com a paràmetre el nom de l'arxiu a obrir. Compila i executa-ho.

```
$ gcc descriptor.c –o descriptor

$./descriptor  descriptor.c
```

Qüestió 1: Analitza el codi i el resultat de l'execució i respon a les següents qüestions:

1. ¿Quines variables corresponen als descriptors d'arxiu en el codi proposat?
2. Justifica el nombre assignat pel sistema a la variable fda
3. Justifica el nombre assignat pel sistema a la variable fdb

3.2 Exercici 2: Descriptor de l'eixida estàndard, crida `close()`

La crida `close(fd)` allibera el descriptor `fd` de la taula de descriptors. En aquest exercici cal confirmar que el descriptor de l'eixida estàndard i, per tant, del terminal correspon al descriptor nombre 1. Per a això treballe amb el codi de *descriptor_output.c* suministrat amb el material de pràctiques.

```
// descriptor_output.c
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc,char *argv[])
{ char *men1="men1: Writing in descriptor 1 (output)\n";
  char *men2="men2: Writing in descriptor 2 (error)\n";
  char *men3="men3: Writing in descriptor 1 (output)\n";
  char *men4="men4: Writing in descriptor 2 (error)\n";
  char *men5="men5: Writing in descriptor 1 (output)\n";
  char *men6="men6: Writing in descriptor 2 (error)\n";

  write(1,men1, strlen(men1));
  write(2,men2, strlen(men2));
  close(1);
  write(1,men3, strlen(men3));
  write(2,men4,strlen(men4));
  close(2);
  write(1,men5,strlen(men5));
  write(2,men6, strlen(men6));
  return(0);
}
```

Compila *desriptor_output.c* i executa-ho.

```
$ gcc descriptor_output.c -o descriptor_out

$./descriptor_out
```

Qüestió 2: Analitza el codi i el resultat de l'execució i respon a les següents qüestions:

1. ¿Quins missatges s'imprimeixen en la pantalla?

2. Justifica per què no s'imprimeixen cadascun dels missatges que falten

3. Ompliu la taula de descriptors d'arxius oberts corresponent a aquest procés abans del `return(0)`

0	
1	
2	
3	
4	

4. Herència de descriptors d'arxius

Quan un procés realitza una crida `fork()`, el procés fill creat hereta gran quantitat d'atributs del seu pare, com el directori de treball, atributs de planificació, … i la taula de descriptors d'arxius oberts. Degut a aquesta herència els processos comparteixen el punter de posició de lectura/escriptura dels arxius oberts abans de la crida `fork()`.

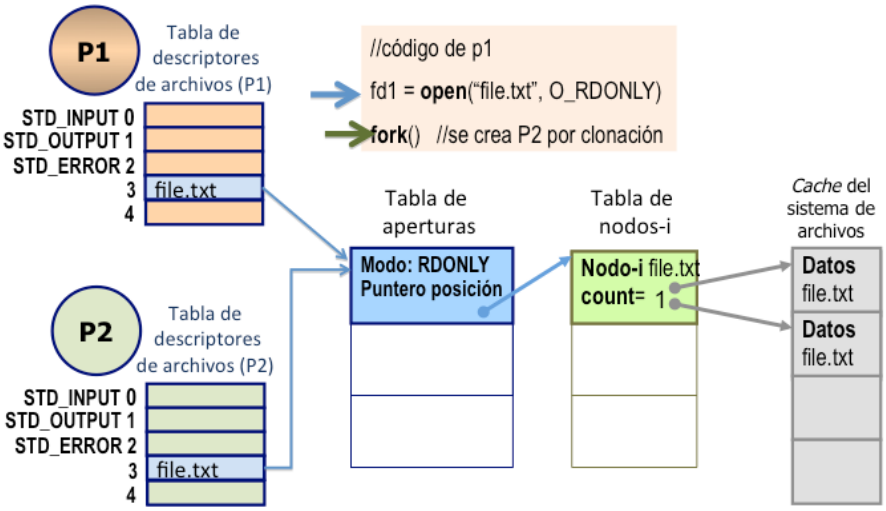

Figura 2: Herencia de la taula de descriptors d'arxius oberts i su relación amb la taula de aperturas, i les estructuras del sistema.

4.1 Exercici 3: Procés pare e fill comparteixen arxiu

En este exercici, l'objectiu és estudiar la compartició d'arxius mitjançant l'herència de descriptors. En concret, el codi de *share_file.c*, que trobaràs entre el material de pràctiques, utilitza el fitxer "`messages.txt`" per a la comunicació entre els processos pare i fill.

```
// share_file.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc,char *argv[]){
  int fd;
  pid_t pid;
  mode_t fd_mode=S_IRWXU;//file permissions
  char *parent_message = "parent message \n";
  char *child_message = "child message \n";

  fd=open("messages.txt",O_RDWR | O_CREAT,fd_mode);
  write(fd,parent_message,strlen(parent_message));

  pid=fork();
  if (pid==0){
     write(fd, child_message,strlen(child_message));
     close(fd);
     exit(0);
  }
  wait();
  write(fd, parent_message,strlen(parent_message));
  close(fd);
  return(0);
}
```

Compila *share_file.c*, executa-ho i mostra el contingut de l'arxiu *mensajes.txt*.

```
$ gcc share_file.c –o share

$./share

$ cat messages.txt
```

Qüestió 3: Analitza el codi i el resultat de l'execució i respon a les següents qüestions:

1. ¿Quin és el contingut de l'arxiu `messages.txt`?

2. Tant el procés pare com el fill han escrit el seu missatge en l'arxiu `messages.txt`. ¿Quins mecanismes/crides ho han fet possible?

3. Ompliu la taula de descriptors d'arxius oberts corresponent al procés pare i fill abans d'executar `close(fd);`

0	
1	
2	
3	
4	

0	
1	
2	
3	
4	

5. Redireccionament: crida dup2

La redirecció de l'entrada estàndard permet a un procés "llegir" dades d'un origen distint del terminal a través del descriptor 0. Per exemple:

```
$ mailx fso10 < missatge
```

El missatge a enviar per l'aplicació `mailx` es troba en l'arxiu `missatge`. La redirecció de l'eixida estàndard permet a un procés "escriure" dades en un destí distint del terminal, a través del descriptor 1. Per exemple:

```
$ echo hola > f1.txt
```

El resultat de l'ordre `echo` s'escriu en el fitxer `f1.txt`. La redirecció de l'eixida estàndard d'errors permet a un procés "escriure" els missatges d'error en un destí distint del terminal, a través del descriptor 2. Per exemple:

```
$ gcc programa1.c –o programa 2 > errores
```

on els errors de compilació (ordre `gcc`) del fitxer `programa1.c` s'escriuen en el fitxer `errors`. El redireccionament de l'entrada, eixida o eixida d'error estàndard en Unix es realitza invocant la crida dup2.

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

`dup2` tanca el descriptor `newfd` i copia el punter associat al descriptor `oldfd` en `newfd`. En l'annex d'aquesta pràctica es descriu amb detall la crida `dup2`.

5.1 Exercici 4: Redirecció de l'eixida estàndard a arxiu

En este exercici es pràctica com utilitzar la crida `dup2()` per a aconseguir que tot allò que s'escriu sobre l'eixida estàndard siga redireccionat a un arxiu. Per a això deurà utilitzar el codi proporcionat amb la pràctica en l'arxiu *redir_output.c* i mostrat a continuació.

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

int main (int argc,char *argv[]) {
  int fd;
  char *arch= "output.txt";
  mode_t fd_mode=S_IRWXU;// file permissions

  fd=open(arch,O_RDWR | O_CREAT,fd_mode);
  if (dup2(fd,STDOUT_FILENO)==-1)
    { printf("Error calling dup2\n");
      exit(-1);
    }

  fprintf(stdout,"out: Output redirected\n");
  fprintf(stderr,"error: not redirected\n");
  fprintf(stderr,"Check file %s\n",arch);
  close(fd);
  return(0);
}
```

Compila *redir_output.c*, executa-ho i mostra el contingut de l'arxiu *output.txt*.

```
$ gcc redir_output.c -o redir_out

$./redir_out

$ more output.txt
```

Qüestió 4: analitza el codi i el resultat de l'execució i respon a les següents qüestions:

1. Justifica, utilitzant les instruccions del codi, el contingut del fitxer `output.txt`

2. Justifica per què la crida open() s'ha invocat amb els flags "`O_RDWR|O_CREAT`"

3. Ompli la taula de descriptors d'arxius oberts corresponent al procés, just abans del "`if(dup…)`"

0	
1	
2	
3	
4	

4. Ompli la taula de descriptors d'arxius oberts corresponent al procés abans d'executar `return(0)`

0	
1	
2	
3	
4	

5.2 Exercici 5: Redirecció de l'eixida estàndard a arxiu

La taula de descriptors d'arxius oberts d'un procés no canvia a l'executar la crida `exec()`, per tant, el procés conserva els arxius oberts i les redireccions que s'hagueren realitzat prèvies al `exec()`. L'objectiu d'aquest exercici és escriure un programa denominat *`ls_redir.c`* que a l'executar-lo execute a la seua vegada l'ordre "`ls -la`", redireccionant l'eixida a l'arxiu *`ls_eixida.txt`*. El resultat final ha de ser equivalent a la següent ordre del shell:

```
$ls -la >ls_eixida.txt
```

Per a fer-h copia *`redir_output.c`* en *`ls_redir.c`*. Edita *`ls_redir.c`* i afegeix en ell la crida `execl()` en el lloc adequat, assegurant-te de que l'eixida es redirecciona a *`ls_eixida.txt`*.

```
execl("/bin/ls","ls","-la",NULL)
```

Compila `ls_redir.c`, executa-ho i mostra el contingut de l'arxiu *`ls_eixida.txt`*.

```
$gcc ls_redir.c -o ls_redir

$./ls_redir

$more ls_eixida.txt
```

Qüestió 5: analitza el codi i el resultat de l'execució i respon a la següent qüestió:

Rere l'execució justifica on s'ha emmagatzemat el contingut del directori actual.

5.3 Exercici 6: Redirecció de l'entrada estàndard des d'arxiu

Escriu un programa denominat *`cat_redir.c`* que al executar-lo execute l'ordre cat redireccionant l'entrada al fitxer *`ls_eixida.txt`*, igual que si fora l'ordre del shell:

```
$cat <ls_eixida.txt
```

Per a fer-ho copia *`ls_redir.c`* en *`cat_redir.c`*. Edita *`cat_redir.c`* i modifica-ho. Assegura't de que l'arxiu *`ls_eixida.txt`* només s'obri per a lectura a l'invocar la crida `open()`.

Compila `cat_redir.c`, i executa-ho. Assegura't de que existeix l'arxiu *`ls_eixida.txt`*.

```
$ gcc cat_redir.c -o cat_redir

$./cat_redir
```

Qüestió 6: Analitza el codi i el resultat de l'execució i respon

¿Què ha sigut necessari modificar en el codi de l'exercici5 per a dur a terme l'exercici 6?

6. Creació de Tubs: `pipe()`

En Unix els tubs són un mecanisme de comunicació entre processos. Un tub és un arxiu sense nom amb dos descriptors, un de lectura i altre d'escriptura. Estos dos descriptors permeten utilitzar punters de posició de lectura i escriptura diferenciats, de manera que el punter de lectura únicament avança quan es realizen operacions de lectura i el d'escriptura quan es realitzen operacions d'escriptura. En UNIX la crida al sistema per a crear un tub es `pipe()` (veure annex):

```
int pipe(int fildes[2])
```

La crida `pipe()` crea un buffer amb un esquema FIFO de gestió del buffer. amb el descriptor `fildes[0]` s'accedeix per a lectura (entrada) i amb `fildes[1]` per a escriptura (eixida). Els tubs no poseen cap nom extern, per tant només poden ser utilitzats mitjançant els seus descriptors pel procés que el crea i pels processos fills que hereden d'ell la taula de descriptors amb `fork()`. Els processos han de compartir el tub i redireccionar la seua entrada o eixida al tub. Per exemple:

```
$ ls | grep txt
```

Aquesta línia de ordres visualitzarà per l'eixida estàndard els noms d'arxius del directori actual que continguen la cadena `txt`.

6.1 Exercici 7: Comunicació de dos processos mitjançant `pipe()`

El objectiu d'aquest exercici es desenvolupar un programa que siga equivalent a l'execució de la següent línia de ordres en el Shell:

```
$ ls -la | wc -l
```

Com mostra la figura-3 l'ordre `ls` ha de redireccionar la seua eixida al tub, mentre que l'ordre `wc` ha de redireccionar la seua entrada per a llegur del tub i la seua eixida a l'arxiu *`eixida.txt`*.

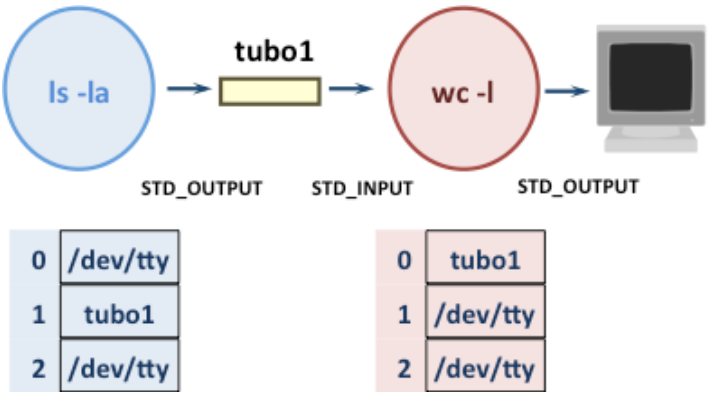

Figura 3: Esquema de redireccionament amb tub que cal implementar en l'exercici 7.

Per tant l'alumne haurà d'implementar un codi on es realitzen les següents accions:

1. El procés pare crea un tub
2. Es crea un procés fill
    a. Fill redirecciona al tub i tanca descriptors
    b. Fill canvia la seua imatge de memòria per a executar l'ordre `ls`
3. Se crea altre procés fill
    a. Fill redirecciona al tub i tanca descriptors
    b. Fill canvia la seua imatge de memòria per a executar l'ordre `wc`
4. Procés pare tanca descriptors i espera als seus fills.

A mode de codi guia es proporciona l'arxiu *a_pipe.c* el qual conté línies de comentaris que l'alumne ha de reemplaçar per instruccions i crides per a terminar l'exercici

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

int main (int argc,char *argv[]) {
  int i;
  char* arguments1 [] = { "ls", "-la", 0 };
  char* arguments2 [] = { "wc", "-l", 0 };
  int fildes[2];
  pid_t pid;
  //Parent process creates a pipe
      if ((pipe(fildes)==-1)){
        fprintf(stderr," Pipe failure  \n");
        exit(-1);
      }
      for (i=0;i<2;i++){
            pid=fork(); //Creates a child process
            if ((pid==0) && (i==0))
            {
              // Child process redirects its output to the pipe
              // Child process closes file descriptors

              // Child process changes its memor i image
              if (execvp("ls",arguments1)<0){
                fprintf(stderr, "ls not found \n");
                exit(-1);
              }
            }
            else if ((pid==0) && (i==1)){

              // Child process redirects its input to the pipe
              // Child process closes pipe descriptors

              // Child process changes its memor i image
              if (execvp("wc",arguments2)<0){
                fprintf(stderr, "wc not found \n");
                exit(-1);
              }
            }
      }

      // Parent process closes pipe descriptors
      close(fildes[0]);
```

```
        close(fildes[1]);
        for (i=0;i<2;i++) wait(NULL);
        return(0);
}
```

Abans de executar el seu programa *a_pipe* modificat comprova quin és el resultat de les ordres del shell que tracta d'implementar. Posteriorment executa el programa i comprova que el resultat es el mateix. Compila *a_pipe.c*, executa-ho.

```
$ gcc a_pipe.c -o a_pipe

$ ls -la | wc -l

$./a_pipe
```

Qüestió 7: Analitza el codi i el resultat de l'execució i respon

1. ¿Què mostra el procés en l'eixida estàndard?

6.2 Exercici 8 (Opcional) : Dos tubs amb tres processos

Basant-te en l'esquema seguit en l'exercici 7, implementa un programa denominat *dos_tubs.c* que execute la següent línia d'ordres:

```
$ ls -la | grep ejemplo | wc -l > result.txt
```

L'estructura de redireccions necessària és la que es mostra en la figura 4.

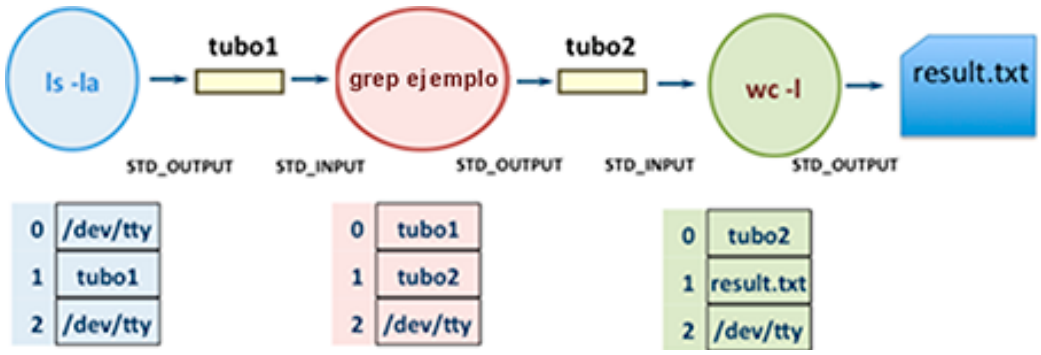

Figura 4: Esquema de redireccionament i tubs que cal implementar en l'exercici 8

7. Annex: sintaxi de crides al sistema

7.1 Crides `open()` i `close ()`

La crida `open()` de Unix

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags [,mode_t mode]);
```

Descripció

La crida `open` obri el fitxer designat per `pathname` i retorna el descriptor de fitxer associat.

`pathname` apunta al nom d'un fitxer.

`flags` s'utilittza per a indicar el mode d'opertura del fitxer. aquest modo es construeix combinant els següents valors:

- `O_RDONLY`: mode lectura.
- `O_WRONLY`: mode escriptura.
- `O_RDWR`: mode lectura i escriptura.
- `O_CREAT`: si el fitxer no existeix, el crea amb els permisos indicats en `mode`.
- `O_EXCL`: si està activat O_CREAT i el fitxer existeix, la crida dona error.
- `O_APPEND`: el fitxer s'obri i l'offset apunta al final del mateix. Sempre que se escriba en el fitxer es farà al final del mateix.

`mode` és opcional i permet especificar els permisos que es desitja que tinga el fitxer en caso de que s'estiga creant.

Retorn

>0: si èxit. Retorna un nombre positiu que correspon al descriptor del fitxer.
-1: si hi ha error.

La crida `close()` de Unix

```
#include <unistd.h>
int close(int fd);
```

Descripció

La crida `close` allibera una posició de la taula de descriptors de fitxers

Retorn

0 : Retorna 0 si no hi ha error
-1 : si hi ha algun error.

7.2 Crides `read ()` i `write()`

La crida `read()` de Unix

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Descripció

La crida `read` llig un nombre de bytes donat per `count` del fitxer al que fa referència el descriptor de fitxer `fd` i els col·loca a partir de l'adreça de memòria apuntada per buf.

Retorn

>0 : si èxit. Retorna el nombre de bytes llegits
0 : si troba el final del fitxer (EOF)
-1 : si hi ha error.

La crida `write()` de Unix.

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

Descripció

La crida `write` escriu un nombre de bytes donat per `count` en el fitxer al que fa referència el descriptor de fitxer `fd`. Els bytes a escriure han de trobar-se a partir de la posició de memòria indicada en buf.

Retorn

>0 : si èxit. Retorna el nombre de bytes escrit

-1 : si hi ha un error

7.3 Crida `pipe()`

La crida pipe() de Unix

```
#include <unistd.h>
int pipe(int fildes[2]);
```

Descripció

Crea un canal de comunicació. El paràmetre `fildes` al retorn conté dos descriptors de fitxer, `fildes[0]` conté el descriptor de lectura i `fildes[1]` el d'escriptura.

L'operació de lectura en `fildes[0]` accedeix a les dades escrites mitjançant `fildes[1]` com en una cua FIFO (primer en arribar, primer en servir-se).

Retorn

0 : si no hi ha error

-1 : si hi ha algun error.

7.4 Crides `dup` i `dup2`

Les crides `dup()` i `dup2()` de Unix

```
#include <unistd.h>
int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

Descripció

Duplica un descriptor de fitxer.

`dup` duplica el descriptor `oldfd` sobre la primera entrada de la taula de descriptors del procés que estiga buida.

`dup2` duplica el descriptor `oldfd` sobre el descriptor `newfd`. En el cas en que aquest ja fera referència a un fitxer, el tanca abans de duplicar.

Retorn

0 : Ambdues retornen el valor del nou descriptor d'arxiu

–1 : en caso d'error