Colección de problemas

Sistemas Operativos
Grau en Enginyeria Informàtica

Este documento contiene ejercicios para ser utilizados por el alumno como parte de su auto-aprendizaje.

Profesores SO-Departamento AC
Primavera 2012

Índice

Comandos

 Comandos útiles para realizar los ejercicios

- ls, ps : para MOSTRAR ficheros y procesos
- wc: para CONTAR
- grep: para BUSCAR y FILTRAR
- head, tail: Para SELECCIONAR las primeras/últimas LÏNEAS de un fichero (o entrada de datos)
- sort: para ORDENAR
- less,cat: pata MOSTRAR el contenido de un fichero

**** Ejercicio 1 ****

Mostrar la ultima librería creada en el directorio /usr/lib

 Comprueba con el man las opciones de los comandos, en este caso ls

**** Ejercicio 2 ****

Mostrar el nombre de los usuarios que están actualmente conectados en la máquina

 Utiliza el comando who. Si no lo conoces, busca en el man.

**** Ejercicio 3 ****

¿**Cuántos** procesos tienes actualmente en ejecución?

 Utiliza el comando ps como base, pero fíjate que pedimos "cuantos", no cuales. Cuando queremos contar, en general podemos combinar el comando base con el comando wc. Búscalo en el man.

**** Ejercicio 4 ****

¿Cuáles son los PIDs de tus procesos?

 Utiliza el comando ps. Intenta conseguir que solamente se muestre el PID, no el resto de datos, ni siquiera la cabecera

**** Ejercicio 5 ****

Escribe una línea de comandos que cuente **cuantos** ficheros hay en el directorio actual.

 Puedes combinar varios comandos con el carácter especial "|", que conecta la salida de un comando con la entrada de otro. En este caso, intenta combinar ls y wc

**** Ejercicio 6 ****

Escribe una línea de comandos que devuelva **cuántos** procesos se están ejecutando en la máquina

**** Ejercicio 7 ****

Escribe una línea de comandos que devuelva el username y el número de procesos que está ejecutando el usuario que más procesos tiene en ejecución en la máquina

**** Ejercicio 8 ****

Escribe una línea de comandos que devuelva el nombre del fichero del directorio actual de trabajo que hace más tiempo que no se modifica.

**** Ejercicio 9 ****

Escribe una línea de comandos que muestre página a página el contenido de todos los ficheros (no directorios) del directorio actual de trabajo.

**** Ejercicio 10 ****

Lanza la aplicación "sleep 3600" en modo background. A continuación, escribe una línea de comandos que averigüe su PID y haga un kill de él.

**** Ejercicio 11 ****

Crea dos ficheros con los siguientes nombres y contenidos:

>cat Fichero1.txt

Este
finde
este
no

>cat Fichero2.txt

no
hay
futbol
no
hay

Muestra por pantalla el número de palabras únicas que hay en total . En el ejemplo del enunciado, el resultado sería 5.

Programación en C

**** Ejercicio 1 ****

Escribe un programa llamado calc.c que implemente una pequeña calculadora. El programa acepta sólo 3 parámetros: el primero y el tercero son números y el segundo parámetro puede ser +, -, / y *. Tiene que mostrar cual es el resultado de efectuar la operación.

> calc op1 operando op2

Ejemplo:

> calc 3 + 4

7

 Recuerda que un string en C no es más que una cadena de caracteres que acaba en el carácter especial '\0'. No existe el tipo de datos string. Todos los parámetros de un programa se reciben, en la función principal (main), a través del vector argv, y todos se reciben como strings.

 Crea una función Usage que compruebe que hay 3 parámetros y que el del medio es un operador válido

 Crea un Makefile

Mejoras:

1. Encapsulación de las operaciones: implementa las operaciones (+, -, * y /) en módulos separados y después enlázalas en un único ejecutable.
2. Encapsula las operaciones en una librería y después enlázala con el programa principal.

**** Ejercicio 2 ****

Escribir un programa en C, llamado calc2.c, que reciba 1 o 2 parámetros. Si recibe otra cantidad de parámetros, se debe mostrar un mensaje de error.

- Si recibe 2 parámetros, se debe comprobar que éstos sean números enteros y se tiene que mostrar por pantalla el resultado de sumarlos, restarlos, multiplicarlos y dividirlos (división entera). Si has hecho el ejercicio anterior, aprovecha el código.
- Si recibe 1 parámetro, se tiene comprobar que es un número real y se tiene que mostrar por la salida estándar el resultado de multiplicarlo por 10. Esta multiplicación por 10 debe hacerse únicamente mediante un shift del punto hacia la derecha, en la cadena de caracteres que representa el número. NO SE DEBE REALIZAR LA MULTIPLICACIÓN ARITMÉTICA POR 10.

ej 45.45544 ---> shift del punto ---> 454.5544.

**** Ejercicio 3 ****

Implementa la función:

char * buscachar (char *str, char c)

Esta función busca en la cadena str la primera aparición de c i retorna un puntero a ésta. Si no se encuentra el carácter, devuelve NULL. Haz un programa que recibe dos parámetros. El primero será una palabra y el segundo un carácter a buscar en esa palabra. Utilizando la función anterior, si el carácter se encuentra en la palabra, el programa mostrará los caracteres que hay desde donde se encuentra el carácter buscado hasta el final. Si no se encuentra, se mostrará un mensaje indicándolo.

Ejemplo:

$Busca hola o

ola

**** Ejercicio 4 ****

Implementa la función: int stringiguals (char *str1, char *str2)

que retorna TRUE si los dos strings son iguales y FALSE en cualquier otro caso.

Haz un programa que reciba dos parámetros. El programa comprobará si son iguales mediante la función anterior y escribirá un mensaje indicado si son iguales o no.

Ejemplo

$Compara azul azul

Los dos parámetros son iguales

**** Ejercicio 5 ****

Hacer un programa llamado param.c que acepta 2 o 3 parámetros.

- En el caso de que sean 2 parámetros, serán: entero palabra
- En el caso de que sean 3 parámetros, serán: entero float palabra
- Para el entero: decir el numero de dígitos validos (003 tiene 1 dígito valido).
- Para el float: dividirlo entre 10 sin convertirlo a float
- Para la palabra: contar el número de consonantes que tiene.
- Tanto para el entero como para el float, se tiene que convertir a formato interno de la maquina cuando se vayan a mostrar por pantalla.

**** Ejercicio 6 ****

Escribe una función que dado un string convierta todos los caracteres a mayúsculas y devuelva el número de conversiones realizadas. Haz un programa para probar su funcionamiento.

int mayusculas(char *fuente, char *destino)

**** Ejercicio 7 ****

Escribe una función que dado un string convierta todos los caracteres a minúsculas y devuelva el número de conversiones realizadas. Haz un programa para probar su funcionamiento.

int minusculas(char *fuente, char *destino)

**** Ejercicio 8 ****

Escribe una función que dado un string imprima por salida estándar el mismo string pero separando cada palabra en una línea y imprima seguidamente el número de palabras detectadas en el string de entrada. Suponed que el carácter separador es el espacio en blanco. Haz un programa para probar su funcionamiento..

int separar (char *fuente)

**** Ejercicio 9 ****

Escribe una función que dado un string genere un string eliminando todos los caracteres que no sean dígitos y devuelva el número de caracteres eliminados:

int digitos(char *fuente, char *dest)

Haz un programa para probar su funcionamiento.

**** Ejercicio 10 ****

Escribe una función que dado un string genere un string eliminando todos los caracteres que no sean letras y devuelva el número de caracteres eliminados:

int letras(char *fuente, char *dest)

Haz un programa para probar su funcionamiento.

Procesos y Signals

**** Ejercicio 1 ****

Haz un programa llamado hijos.c, que cree dos hijos. Cada hijo escribirá por la salida estándar su número de hijo y su PID en ascii y terminará su ejecución.

 Utiliza las llamadas a sistema fork (para **crear** un proceso, exit (para **finalizar** la ejecución de un proceso y , si se quiere, devolver un estado de finalización) write(para **escribir** en un dispositivo).

 Puedes utilizar la función sprintf para convertir cualquier cosa a un *string*. El siguiente ejemplo crea un *string* y substituye el patrón %d por el primer parámetro pid. %d indica que el parámetro es un entero. Si no pones texto adicional, "%d", simplemente convertirías el número a *string*.

```
char buffer[64];
int pid=111;
sprintf(buffer, "el pid es %d\n",pid);
write(1,buffer,strlen(buffer));
```

Salida de ejemplo:
Soy el hijo 2 y mi PID es el 124
Soy el hijo 1 y mi PID es el 123

**** Ejercicio 2 ****

Si has hecho el ejercicio 1 este es una extensión. Haz un programa llamado hijos2.c, que cree dos hijos. Cada hijo escribirá por la salida estándar su número de hijo y su PID en ascii y terminará su ejecución con estado 0. El padre esperará a que acabe cada hijo y escribirá por la salida estándar el PID del hijo que ha acabado.

 Utiliza las llamadas a sistema fork (para **crear** un proceso, exit (para **finalizar** la ejecución de un proceso y , si se quiere, devolver un estado de finalización), waitpid (para **esperar, si no ha terminado aún,** a un proceso hijo y consultar su estado de finalización). Consulta en el man cómo sabe, el proceso padre, mediante waitpid, el PID del proceso hijo que ha terminado. Usa write(para **escribir** en un dispositivo).

Salida de ejemplo:
Soy el hijo 2 y mi PID es el 124
Soy el hijo 1 y mi PID es el 123
El hijo con PID 124 ha finalizado
El hijo con PID 123 ha finalizado

**** Ejercicio 3 ****

Si has hecho el ejercicio 2 este es una extensión. Haz un programa llamado hijos3.c, que cree dos hijos. Cada hijo escribirá por la salida estándar su número de hijo y su PID en ascii. Cada hijo, además, indicará como estado de finalización su número de hijo (0 ó 1). El padre esperará a que acabe cada hijo y escribirá por la salida estándar el PID del hijo que ha acabado y su estado de finalización.

 Utiliza la macro WEXITSTATUS para extraer el estado de finalización de un proceso del "status" que devuelve el waitpid. Esta macro sólo sirve para aquellos casos en que el proceso termina con un exit.

Salida de ejemplo:
Soy el hijo 2 y mi PID es el 124
Soy el hijo 1 y mi PID es el 123
El hijo 0 con PID 124 ha finalizado
El hijo 1 con PID 123 ha finalizado

**** Ejercicio 4 ****

Haz un programa, Ejecutador, que recibirá como parámetro el nombre de 2 comandos (por ejemplo ls y ps). El programa recibirá un tercer parámetro que será el texto sec o conc (secuencial o concurrente). Si el parámetro es "sec", los dos comandos recibidos se deberán ejecutar en secuencias. Si el parámetro es "conc", se deberán ejecutar de forma concurrente. El proceso

padre esperará a que acaben los comandos y escribirá, por su salida estándar, un mensaje indicando el PID del proceso que ha terminado y su estado de finalización.

- Haz un función Usage que comprueba que haya 3 parámetros y que el tercero sea o sec o conc.
- Haz un makefile

 Puedes utilizar la función strcmp para comparar *strings* en C. Utiliza la llamada execlp para mutar el código que ejecuta un proceso.

**** Ejercicio 5 ****

Haz el código del siguiente programa:

> plauncher n cmd1 cmd2 ... cmdi

Este programa ejecutará, de forma concurrente, cada uno de los diferentes comandos especificados como argumentos y escribirá por la salida estándar aquellos comandos que no acabaron correctamente (indicando su PID). Asumiremos que un estado de finalización diferente de 0 significa que ha habido alguna incidencia. El primer parámetro "n" indica cuantos procesos pueden existir a la vez (sin contar al padre).

Ejemplo:

>plauncher 2 ls uptime ps

Ejecutará ls y uptime y cuando cualquiera de ellos acabe entonces ejecutara ps.

- Haz una función Usage para comprobar que, como mínimo, hay dos parámetros: n y un comando
- Haz un Makefile
- Haz primero el programa ignorando la n, es decir, ejecutando todos los comandos concurrentemente y luego limitando la concurrencia a n.

**** Ejercicio 6 ****

Haz una variante del programa plauncher, que llamaremos plauncher_din, en el que el programa recibirá solamente la lista de comandos. El programa ejecutará un programa de la lista cada vez que reciba un signal SIGUSR1. Si en algún momento el programa recibe un SIGUSR2, el programa terminará su ejecución indicando el nombre de los comandos que le quedaban por ejecutar. El proceso debe controlar, mediante el signal SIGCHLD, la terminación de los procesos que va creando. Cuando termine el último programa de su lista, el programa terminará su ejecución escribiendo un mensaje por su salida estándar indicándolo.

- Haz dos versiones del programa: una en utilizas una función diferente para cada signal y otra donde utilices la misma función para todas

 Utiliza las llamadas a sistema signal (para reprogramar un signal) y pause(para bloquear al proceso hasta recibir un signal). Cuidado que esta última función puede bloquear al proceso indefinidamente si el signal que esperamos ya hubiera llegado al ejecutar el pause.

**** Ejercicio 7 ****

Haz un programa que llamaremos envía_sigusr1 que recibirá tres parámetros: un pid, una letra y un número. Este programa enviará el signal SIGUSR1 al proceso con el pid que recibe como primer parámetro de dos formas diferentes.

Si el segundo parámetros es una "t", significa que ha de enviar N SIGUSR1 seguidos, donde N es el tercer parámetro. Si el segundo es una "p" significa que se ha de enviar el signal periódicamente cada N segundos, donde N será también el tercer parámetro.

- Haz una función Usage para comprobar que hay 3 parámetros y que el segundo es una t o una p. Haz también un Makefile
- Pruébalo con el programa plauncher_din. Comprueba que pasa con los dos modos de utilizar el programa envía_sigusr1.

 Utiliza las llamadas a sistema kill (para enviar un signal) y alarm (para programar un envío automático del signal SIGALRM). Cuidado que esta función tiene como acción por defecto terminar la ejecución del proceso, por lo tanto deberás reprogramar el SIGALRM.

**** Ejercicio 8 ****

Haz el siguiente programa:

> plauncher_con_param cmd1 [arg1 ... argn] - cmd2 [arg1 ... argn] - cmdi [arg1 ... argn]

Este programa ejecutará cada uno de los diferentes comandos especificados como argumento (con sus respectivos argumentos) secuencialmente. Los comandos se separan mediante "-".

Para cada comando indicará como ha sido la finalización (con qué resultado acabó). El número máximo de argumentos por comando será de 10.

Ejemplo:

> plauncher_con_param ls -la /etc - ps -x - uptime - cat /etc/passwd

ejecutará primeramente un ls -la /etc, posteriormente ps -x, después uptime y finalmente cat /etc/passwd, informando del resultado de cada uno.

- Incluye una función Usage y un Makefile

**** Ejercicio 9 ****

Implementa un programa que se llame migrep.c con la siguiente sintaxis:

>migrep palabra fichero [listaficheros]

Este programa mostrará el nombre de los ficheros pasados como parámetro que contengan "palabra". Para ello se tiene que utilizar el comando grep.

La salida tiene que ser:

El fichero xxxx contiene la palabra yyy.
El fichero ssss no continene la palabra yyy.

La primera versión ejecutará secuencialmente grep para mirar fichero por fichero si éstos contienen palabra. Ayuda: Utiliza la opción –q del comando grep para que no escriba sus mensajes por su salida estándar. Consulta también en el man, los valores que devuelve el comando grep, en su estado de finalización, en función de si encuentra o no la palabra.

**** Ejercicio 10 ****

Igual que el ejercicio anterior pero en este caso, tienen que mirarse todos los ficheros en paralelo.

Se puede suponer que, como mucho, la lista de ficheros contendrá 10 ficheros.

**** Ejercicio 11 ****

Escribid un programa, que llamaremos *aleatorio*, que reciba como parámetro un número entero, que utilizará como semilla para la generación de números aleatorios (función srand). Este programa sólo debe generar un número aleatorio (función rand) y devolver como parámetro del exit un 1 si el número generado es impar y un 2 si es par.

Escribid un segundo programa que reciba como parámetro un número entero que le indicará el número de procesos hijo que debe crear. Los hijos se ejecutarán de forma secuencial (es decir, hasta que un hijo acabe el padre no creará el siguiente hijo) y cada hijo ejecutará el programa del apartado anterior, *aleatorio*. Al primer hijo se le pasará como parámetro el pid del padre y al resto de los hijos el pid del hijo creado en la iteración anterior. Este programa recogerá el código de fin de cada hijo y mostrará por pantalla un mensaje indicando si el número generado por el hijo era par o impar.

**** Ejercicio 12 ****

Escribe un programa que reciba como parámetro el número de procesos que debe crear. Todos los procesos hijo se ejecutarán en paralelo y cada uno escribirá un mensaje por pantalla con su pid, los pid de sus hermanos mayores (procesos creados antes que él) y diciendo cuántos hermanos pequeños tendrá. El padre esperará hasta que todos los hijos acaben.

**** Ejercicio 13 ****

Queremos implementar un programa, que llamaremos Npls, que reciba como parámetros una lista de directorios y ficheros y haga un análisis sobre ellos (usando un programa que nos pasan, de nombre pls). Para conseguirlo, se quiere que el programa cree tantos procesos hijos como parámetros reciba, todos esos hijos se tienen que ejecutar de forma concurrente y cada uno debe mutar para ejecutar el programa pls con la opción –l y el parámetro que le toca tratar. Se propone la siguiente implementación:

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <unistd.h>
4: main(int argc, char **argv){
 5: int i,pid_h;
6:  if (argc < 2) {
7:      fprintf(stdout, "usage: Npls f1 [f2].... [fn]\n");
8:      exit(1);
 9:}
10:for (i=0;i<argc-1;i++){
11:    pid_h=fork();
12:    if (pid_h < 0) {
13:       perror("Error creando hijo");
14:       while (waitpid(-1,NULL,0) > 0);
15:       exit(1);
16:    }
17:    If (pid_h==0){
18:        execlp("pls", "pls", "-l", argv[i+1], (char *) 0);
19:        perror("Error mutando al pls");
20:     }
21: }
22:   while (waitpid(-1,NULL,0) > 0);
23:   fprintf(stdout, "Han acabado todos los pls\n");
24:}
```

Supón que lanzamos este programa pasándole como parámetro el nombre de dos ficheros que se encuentran en el directorio actual de trabajo (f1.txt y f2.txt)

1. Representa la jerarquía de procesos que creará este código indicando, exactamente, el programa que ejecutará cada uno de ellos.
2. Después de varias pruebas decidimos que la ejecución concurrente no es lo más adecuado para este tipo de programa y decidimos hacerla secuencial. Indica que líneas de código

habría que añadir/mover/cambiar/etc para conseguir, con el mínimo número de cambios, que este código sea secuencial. (utiliza las líneas de código como guía para describir los cambios)

3. Suponed que queremos controlar que cada ejecución del programa pls ejecutado por los procesos hijos no esté más de 100 segundos en ejecución, ya que eso significaría un problema en la ejecución del programa pls y habría que abortarlo. Nos dicen que el programa pls termina de forma controlada si recibe un SIGINT, por lo de que decidimos controlar, desde el proceso padre, si han pasado los 100 segundos y en ese caso enviar un SIGINT al proceso que se esté ejecutando en ese momento y continuar con el siguiente. Indica que cambios habría que realizar en el código para controlar el tiempo y enviar el signal SIGINT al proceso hijo que haya en ese momento activo.
    a. Piensa como lo harías en la versión original (secuencial) y en la versión modificada por ti (concurrente)


Piensa que en el caso secuencial, sólo tendrás activo un proceso, pero en el caso concurrente tendrás varios, por lo que deberás almacenar los PIDs en un vector. También has de tener en cuenta que pasa si el proceso tarda menos de 100 segundos.

**** Ejercicio 14 ****

Tenemos el siguiente código en el fichero test.c:

```
int status;
int main()
{
  // PUNTO A
  switch(fork())
  {
    case 0:
            // PUNTO B
            execlp("ls", "ls", (char*)0);
            exit(0);
  }
  // PUNTO C
  wait(&status);
  // PUNTO D
  exit(0);
}
```

Lo compilamos y ejecutamos en background:

> ./test &

[2] 29640

y consultamos el espacio de direcciones del proceso cuando se encuentra en el PUNTO A:

```
> cat /proc/29640/maps
00c40000-00c41000 ---p 00157000 08:01 266449     /lib/libc-2.12.1.so
00c41000-00c43000 r--p 00157000 08:01 266449     /lib/libc-2.12.1.so
00c43000-00c44000 rw-p 00159000 08:01 266449     /lib/libc-2.12.1.so
00c44000-00c47000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:01 154767     /home/SO/test
08049000-0804a000 r--p 00000000 08:01 154767     /home/SO/test
0804a000-0804b000 rw-p 00001000 08:01 154767     /home/SO/test
b772c000-b772d000 rw-p 00000000 00:00 0
b7740000-b7742000 rw-p 00000000 00:00 0
bfebc000-bfedd000 rw-p 00000000 00:00 0          [stack]
```

Al proceso hijo se le asigna el PID 29641. Responde a las siguientes preguntas, justificando tu respuesta:

1. Describe cual será el contenido (no pongas el contenido, solamente describe los cambios principales: regiones nuevas/eliminadas, aumento/disminución de tamaño, etc) del fichero maps del hijo cuando:
    a. el proceso hijo alcance la línea marcada como PUNTO B
    b. el proceso hijo mute
    c. el proceso padre alcance la línea marcada como PUNTO D
2. ¿Cambian las respuestas anteriores si el sistema operativo incorpora la optimización de copy-on-write?
3. Al ejecutar test, vemos que el orden de ejecución del código es: PUNTO A – PUNTO B – PUNTO C – PUNTO D

¿Se puede asegurar que siempre se va a mantener este orden en cualquier ejecución del programa test?

4. ¿Cuál será el PID del proceso hijo cuando haya acabado de mutar para ejecutar ls?
5. La variable status está en la dirección lógica 0804a024 en el espacio de direcciones del proceso padre. En el sistema operativo, las páginas lógicas tienen un tamaño de 4KBs. ¿Cuál será la dirección lógica de esta variable dentro del proceso hijo?

 Para realizar este ejercicio es vital recordar la diferencia entre direcciones lógicas y físicas. Comprueba de que tipo son las direcciones del fichero maps. También debes recordar a que nivel se aplica la optimización de CoW, si de páginas lógicas o físicas.

Planificación procesos

**** Ejercicio 1 ****

Tenemos un sistema que implementa la política de planificación *Round Robin*. En un momento dado, tenemos 3 procesos activos y se observa el siguiente diagrama de Gantt. Analiza el diagrama y contesta a las siguientes preguntas.

	P0	P0	P0	P0	P1	P2	P2	P2	P2	P0	P0	P0	P0	P1	P1	P2	P2	P2	P2	P0	P0	P0	P0	P1	P0	P0	P0	P0	P1
T	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

1. A la vista de este diagrama, y en función de la duración de las ráfagas de cpu que se observa ¿Cuál crees que podría ser la duración del quantum del sistema?
2. ¿Cuántos cambios de contexto (y en que ciclos) se han producido durante este intervalo?

 Piensa en qué casos la política de planificación Round Robin decide que hay que hacer un cambio de contexto.

Procesos, Signals y Entrada/Salida

**** Ejercicio 1 ****

Haz un programa que escriba N veces un número (en formato binario) por su salida estándar (sin espacios ni ningún carácter separador). El programa se llamará de la siguiente forma:

$genera_nun num_escribir cantidad

 Utiliza la llamada write para escribir los números

**** Ejercicio 2 ****

Haz un programa, que llamaremos suma_num, que lea de su entrada estándar una secuencia de números (en formato binario), asumiendo que no hay ningún espacio ni carácter separador entre ellos. Los números se almacenarán en un vector de enteros, los sumará y escribirá por su salida estándar, esta vez en formato ascii, el resultado de la suma. El programa recibirá como único parámetro la dimensión del vector. Reserva el espacio para el vector mediante memoria dinámica.

- Haz una función Usage para comprobar que nos pasan la dimensión del vector
- Haz dos versiones: mediante la función de la librería de C para reservar el vector y utilizando la llamada a sistema. En ambos casos debes liberar la memoria después de haber hecho la suma.
- Utiliza el programa del ejercicio anterior como en el siguiente ejemplo, conectaremos ambos programas mediante una pipe.

  $ genera_num 2 10 | suma_num 10

  La suma es 20

 Utiliza la llamada read (para leer los números), las funciones malloc/free (para reservar/liberar memoria dinámica mediante la librería de C) y sbrk (para reservar/liberar la memoria dinámica con llamada a sistema)

**** Ejercicio 3 ****

Haz una versión nueva del programa suma_num que acepte como parámetro un fichero donde estarán los datos de entrada. Este fichero será opcional, si no existe, los datos se leerán de la entrada estándar. Por lo tanto, estas opciones deben funcionar igual

```
$ genera_num 2 10 | suma_num 10
La suma es 20
$genera_num 2 10 > f_datos
La suma es 20
$suma_num 10 <f_datos
La suma es 20
$suma_num 10 f_datos
La suma es 20
```

 Utiliza la llamada open (para obtener el acceso a un fichero, abrirlo)

**** Ejercicio 4 ****

Haz un programa que ejecutará el mismo código que haría la Shell si ejecutáramos la siguiente línea de comandos:

$ genera_num Y X | suma_num X

El programa recibirá como parámetros la X e Y, creará los dos procesos que ejecutarán los comandos genera_num y suma_num. El proceso padre será el encargado de crear la pipe sin nombre en el lugar adecuado, redireccionar los canales, cerrar canales, etc.

 Utiliza las llamadas a sistema pipe (para crear una pipe sin nombre), dup2 para redireccionar canales, y close (para cerrar canales).

**** Ejercicio 5 ****

Queremos crear un proceso que se llame:

>demonio pipe num_segundos \&

Este proceso crea una named pipe llamada "pipe" (primer parámetro) por donde va recibiendo palabras.

- Cuando demonio recibe SIGUSR1, lee un único directorio que se le ha pasado por la named pipe, y crea un hijo que ejecuta un ls del directorio pasado.
- Cuando demonio recibe SIGUSR2, lee el contenido de la named pipe, y crea un hijo que ejecutará ps utilizando como parámetro lo que se le haya pasado por la named pipe.

Además, demonio tiene un temporizador que cada num_segundos segundos vacía la named pipe.

El proceso demonio solo acaba con SIGKILL.

- Nota: la named pipe se tiene que abrir con O_NONBLOCK.
- Nota 2: cuidado con los strings que se leen por la named pipe
- Nota 3: para enviar algo a la named pipe se hace con el comando echo. Ejemplo:

> echo -la > pipe

 Utiliza la llamada mknod (para crear un dispositivo tipo pipe)

**** Ejercicio 6 ****

Haz un programa que se llamará generador.c. Por defecto, el programa escribirá, cada segundo, el carácter "@" por su salida estándar. Este programa terminará cuando reciba SIGUSR1.

Además, se quiere que, de forma externa, se pueda modificar el comportamiento del programa generador mientras se está ejecutando. Para esto, generador creará una named pipe antes de empezar la generación de caracteres. El nombre de esta named pipe se pasará como primer parámetro al programa.

Las órdenes que se pueden enviar al generador a través de la named pipe son (cada orden es una letra):

- P: Este comando pausará la generación de caracteres. Imprimira una "|" por su salida estándar para indicar que está pausado. El tiempo en pausa no cuenta como tiempo entre caracteres.
- C: Este comando pondrá otra vez en marcha la generación de caracteres. Mostrará ">" para indicar este hecho.
- @: Permite cambiar el carácter que se muestra cada segundo. El nuevo carácter vendrá a continuación de @.
- I: Permite especificar un nuevo intervalo en segundos. El nuevo intervalo (entre 0 y 9 segundos) se indicará con un entero (en formato ascii) pasado a continuación de la I.
- X: Obliga a que se genere un nuevo carácter aunque el generador esté en pausa o no se haya agotado el tiempo entre generación de caracteres. No cambia el estado de pausa pero sí el intervalo de generación de caracteres, ya para mostrar el siguiente carácter se tiene que esperar un intervalo entero.

Para enviar comandos al generador, se tiene que utilizar el comando echo. Por ejemplo, si la named pipe se llama control:

- echo P > control
- echo @% > control
- echo I5 > control
- echo P@%I3C > control

**** Ejercicio 7 ****

Escribe un programa que reciba como parámetro el tiempo máximo que se va a esperar para cada nueva lectura de entrada estándar. El programa ejecutará un bucle que muestra por salida estándar lo que recibe por entrada estándar, hasta que la lectura le devuelva final de datos. Para cada lectura se limita el tiempo máximo que tiene el usuario para introducir los datos, si pasado ese tiempo el usuario no ha introducido los datos se sacará por salida de errores estándar un aviso y se reintentará la lectura durante la misma cantidad de tiempo. Si vuelve a expirar el tiempo entonces se mostrará en salida estándar un aviso de que se acaba la introducción de datos y el programa finalizará.

**** Ejercicio 8 ****

Escribe un programa que reciba como parámetro el número de procesos hijos que debe crear y el número de iteraciones que deberá ejecutar. Este programa creará una pipe sin nombre que usará para comunicarse con los hijos y a continuación ejecutará un bucle durante el número de iteraciones que ha recibido como parámetro. En cada iteración del bucle, seleccionará aleatoriamente un proceso hijo y le enviará un signal de tipo SIGUSR1, a continuación leerá de la pipe el mensaje que el hijo le enviará y lo mostrará por pantalla.

Cuando acabe el bucle deberá enviar un signal de tipo SIGTERM a todos sus hijos y esperar su finalización. Por su parte cada hijo ejecutará un bucle que finalizará al recibir el signal SIGTERM. En ese bucle cada hijo esperará hasta recibir un signal de tipo SIGUSR1, escribirá en la pipe un mensaje que contenga su pid y el número de signals que ha recibido y se quedará bloqueado hasta recibir el siguiente signal.

**** Ejercicio 9 ****

Escribe un programa que escriba en salida estándar lo que reciba por entrada estándar hasta que la lectura le devuelva que no quedan datos por leer. Si el programa recibe un signal de tipo SIGINT (por ejemplo, si el usuario pulsa \^C) el programa deberá mostrar por salida estándar un mensaje con el número de bytes leídos hasta el momento y continuar la ejecución.

**** Ejercicio 10 ****

"Temporizador": este programa recibirá como parámetro un tiempo límite de espera. El proceso lee de su entrada estándar el pid de otro proceso (en formato binario), y, a continuación, calcula de forma aleatoria el tiempo que tiene que esperar (que no debe sobrepasar el valor que ha recibido como parámetro). Pasado ese tiempo enviará un signal de tipo SIGKILL al proceso cuyo

pid ha recibido por la entrada estándar. Para calcular el tiempo aleatorio podéis usar la rutina aleatorio que os damos implementada.

```
int aleatorio(int lim_sup){
        srand(getpid());
        return((1 + (int) ((float)lim_sup * (rand() / (RAND_MAX + 1.0)))));
}
```

"Carrerahermanos": un programa que cree dos procesos con los que se comunicará usando pipes sin nombre (¿cuántas?) para mandar a cada uno el pid de su hermano. Cada proceso ejecutará el programa *temporizador*, pasándole como parámetro el tiempo límite de espera (una constante del programa). A continuación quedará a la espera de que los dos hijos acaben y mostrará por salida estándar un mensaje con los pids de los dos procesos, indicando cuál ha sido más rápido (cuál ha acabado con exit y cuál ha muerto por el SIGKILL).

**** Ejercicio 11 ****

Implementar un programa llamado "familia" que aceptará un parámetro de entrada que indicará el número de hijos que creará el proceso principal.

Todos los procesos hijo se bloquearán (piensa cómo hacerlo), excepto el primero que previamente activará una alarma. Esta alarma enviará una señal a los dos segundos para que dicho proceso hijo envíe un SIGUSR1 al padre.

Mientras tanto el padre va a estar leyendo de la entrada estándar. Después de finalizar una lectura el proceso padre duerme durante 1 segundo.

Si el proceso padre recibe el SIGUSR1 cuando no ha terminado de leer, entonces envía un SIGUSR2 al proceso que le envió la señal (matando a dicho proceso) y despertará al siguiente hijo, el cual tendrá el mismo comportamiento que su proceso hermano.

Si por el contrario el proceso padre recibe el SIGUSR1 cuando está "descansando", entonces enviará un SIGUSR1 al proceso hijo que le envió la señal. Este, al recibir la señal, reseteará la alarma para que otra vez, al cabo de 2 segundos, envíe un SIGUSR1 al proceso padre.

**** Ejercicio 12 ****

Haz el siguiente programa:

> multipipe [-w] n comando1 arg1 ... argn -- comando2 arg1 ... argn

Este programa creará n instancias de comando1 redireccionando su salida estándar a la entrada estándar de una única instancia de comando2. Comando1 y comando2 siempre irán separados por "--".

Ejemplo:

> multipipe 3 ls / -- grep usr.

El programa por defecto esperará a que todos los procesos acaben excepto si se le pasa la opción -w, en cuyo caso retornará después de haber creados los procesos.

**** Ejercicio 13 ****

Haced dos programas: uno que se llamará suma.c y otra que se llamará psuma.c.

Suma.c lee enteros (en formato binario) de su entrada estándar y los va sumando. Cuando ya no hay más números, escribe por su salida estándar el resultado de la suma (también en formato binario).

Psuma.c tiene la siguiente sintaxis:

> psuma nhijos n1 n2 n3 n4 .... nn

Funciona de la siguiente manera: creará tantos hijos como indique el argumento nhijos. Cada uno de estos hijos ejecutará el programa suma.

Además, utilizando una pipe, suministrará como números a sumar a los hijos el resto de argumentos. Después recogerá los resultados de las sumas parciales de cada hijo, las acumulará y escribirá por la salida estándar el resultado final. Utilizad únicamente dos pipes en total.

**** Ejercicio 14 ****

Implementar un programa llamado pr.c con la siguiente sintaxis:

> pr userid

que mediante forks, pipes y execs, tiene que ejecutar el equivalente a la siguiente línea de comandos:

> ps -e | grep userid | wc -l

**** Ejercicio 15 ****

Escribir un programa, llamado conectado.c, que nos diga en qué terminal está conectado un determinado usuario pasado como parámetro al programa. El proceso principal deberá crear dos hijos que ejecuten lo equivalente a la siguiente línea de comandos:

> who | grep usuario

El proceso padre deberá escribir por pantalla el terminal donde está conectado el usuario o indicar que no está conectado.

**** Ejercicio 16 ****

Escribir un programa en C, llamado migrep.c, cuya sintaxis es:

> migrep fichero cadena

Este programa nos dice en cuántas líneas del fichero pasado como parámetro aparece la cadena que se pasa como segundo parámetro. El proceso principal deberá crear dos hijos que ejecuten el equivalente a la siguiente línea de comandos:

> grep palabra fichero | wc -l

El proceso padre deberá escribir por pantalla el resultado.

**** Ejercicio 17 ****

Escribir un programa en C, llamado mips.c, que nos diga cuántos procesos está ejecutando un determinado usuario, pasado como parámetro. El proceso principal deberá crear tres hijos que ejecuten el equivalente a la línea de comandos:

> ps -eaf | grep usuario | wc -l

El proceso padre deberá escribir por pantalla el resultado.

**** Ejercicio 18 ****

Escribir un programa en C, llamado miping.c, que reciba un número entero como parámetro. Este programa creará un proceso hijo. Ambos procesos, padre e hijo, se irán intercambiando e incrementado un número entero comenzando por 0 hasta que uno de los dos procesos al incrementar llegue al valor pasado como parámetro. En ese momento los dos procesos deben acabar de forma elegante. Utilizar dos pipes para comunicar los procesos.

Mejora: Utiliza solamente una pipe.

**** Ejercicio 19 ****

Implementa un programa llamado avisos.c, que cree dos hijos. El primer hijo, tan rápido como pueda, se quedará "dormido" (utiliza la función sleep). El segundo hijo, 5 segundos después de haberse creado, acabará. Una vez ha acabado el segundo hijo, el primer hijo se despertará y mostrará: "Mi hermano ha terminado y yo voy a hacerlo ahora". El proceso padre tiene que esperar la finalización de los procesos hijos.

**** Ejercicio 20 ****

Escribid los siguientes programas:

pingpong: este programa creará dos procesos hijos que intercambiarán mediante pipes sin nombre un valor. El programa recibe como parámetro el valor inicial y el número de intercambios que deberán hacer sus hijos. Uno de los hijos ejecutará el programa ping y el otro el programa pong.

ping: este programa ejecuta un bucle que consiste en escribir un valor por salida estándar, leer de entrada estándar el nuevo valor e incrementarlo antes de volverlo a escribirlo. Los parámetros del programa son el valor inicial y el número de iteraciones que debe ejecutar el bucle.

pong: ejecuta un bucle en el que lee de entrada estándar un valor, lo incrementa y escribe el valor modificado en salida estándar. La condición de fin del bucle es que la lectura de entrada estándar devuelva final de datos.

**** Ejercicio 21 ****

Haz un programa, que llamaremos esta_conectado, que comprueba si un usuario está o no conectado. El programa recibe como parámetro el username que queremos comprobar. Para realizarlo, utilizaremos los comandos ya existentes *who* y *grep*. El programa, terminará con un 1 en su estado de finalización si el usuario está conectado y con un 0 si no lo está.

**** Ejercicio 22 ****

Queremos implementar un programa que dado un username nos avise de cuándo ese usuario entra en el sistema. El funcionamiento deseado es que cada cierto tiempo T (por defecto 180 segundos) el proceso compruebe si el usuario ha entrado en la máquina. Si ha sido así mostrará un mensaje por pantalla y acabará. Si por el contrario, el usuario no ha entrado, el proceso se quedará bloqueado durante T segundos, y pasado ese tiempo volverá a realizar la comprobación. Este programa aceptará como parámetros el username y, opcionalmente, el número de segundos que tiene que esperar entre comprobaciones.

Para hacerlo, utiliza el programa esta_conectado realizado en el ejercicio anterior.

**** Ejercicio 23 ****

Haz un programa que llamaremos anillo_de_procesos, que reibe como parámetros un número de hijos a crear. En él, todos los procesos (hijos y padre) estarán conectados mediante pipes sin nombre formando un anillo. Cada proceso mandará por la pipe a su hermano "pequeño" (el siguiente en el orden de reación lo que ha recibido de su hermano "mayor" (el anterior en el orden de creación) sumando su propio pid. El padre es que el que inicia la cadena enviando su pid

a su hijo mayor y el que la finaliza mostrando por salida de errores estándar lo que ha recibido de su hijo menor.

Cada proceso hijo ejecutará el siguiente código: un bucle que leerá de entrada estándar (hasta que la lectura le devuelva que no quedan más datos que leer) y que escribirá en salida estándar lo que ha leído.

Todos los datos se envían/reciben en formato binario.

**** Ejercicio 24 ****

Pidordenado. Escribe un programa que reciba como parámetros el número de procesos que debe crear de forma concurrente. Queremos que los procesos hijos escriban por su salida estándar su PID pero queremos que salga en el orden de creación, por lo que habrá que implementar una sincronización entre ellos. Para implementar la sincronización entre procesos y que el orden de escritura sea el requerido, se utilizarán pipes ordinarias. El padre esperará hasta que todos sus hijos acaben y mostrará el mensaje fin de pids.

 Piensa como explotar las características bloqueantes de las pipes para forzar una sincronización

**** Ejercicio 25 ****

Implementa un programa que llamaremos messenger. El programa messenger recibirá dos parámetros que serán los nombres de dos nicks (usernames). El programa implementará una comunicación entre los dos usuarios que usan los nicks indicados. El primer nick será el del usuario que ejecuta el programa y el segundo el del usuario con el que se quiere conectar. Todos los nicks serán de 4 caracteres.

Para ello, el programa realizará las siguientes acciones:

- Creará una pipe con nombre, de nombre /tmp/nick1_nick2
- Escribe en el fichero /tmp/mess_pids su Nick y un 0 ó 1 (en ascii) para indicar si está o no conectado. El programa deberá comprobar si el usuario ya había estado antes conectado, en ese caso cambiar el 0 por el 1. Sinó, se añadirá al final del fichero su Nick y un 1.
- Comprueba si el usuario con el que quiere hablar está o no conectado (usando el fichero /tmp/mess_pids). En caso de no estar conectado, se escribirá un mensaje por la salida estándar indicando que el usuario no está conectado y por lo tanto habrá que esperar. En ese caso, el proceso comprobará cada segundo si el usuario está conectado antes de continuar. Si está conectado, daremos un mensaje indicando que se conectará de forma inmediata.

- Una vez esté conectado el usuario con el que queremos hablar, el proceso abrirá la pipe /tmp/nick1_nick2 en modo escritura y la pipe /tmp/nick2_nick1 en modo lectura.
- Se creará un proceso nuevo que lea de la pipe /tmp/nick2_nick1 y lo escriba por la salida estándar. El proceso padre se encargará de leer de la entrada estándar y escribir en la pipe /tmp/nick1_nick2
- Para utilizarlo abre dos terminales ejecuta "messenger nick1 nick2" y en el otro "messenger nick2 nick1"

Sistemas de ficheros

**** Ejercicio 1 ****

Tenemos la siguiente lista de Inodos y bloques de un sistema de ficheros basado en UNIX. Los Inodos 9 y 10 están libres, así como el bloque 9.

Inodo	2	3	4	5	6	7	8	9	10
Tipo	dir	dir	Dat	dat	dir	dir	Link		
Bloques	2	3	5	6	7	4	8		
#ref	5	2	1	1	2	2	2		

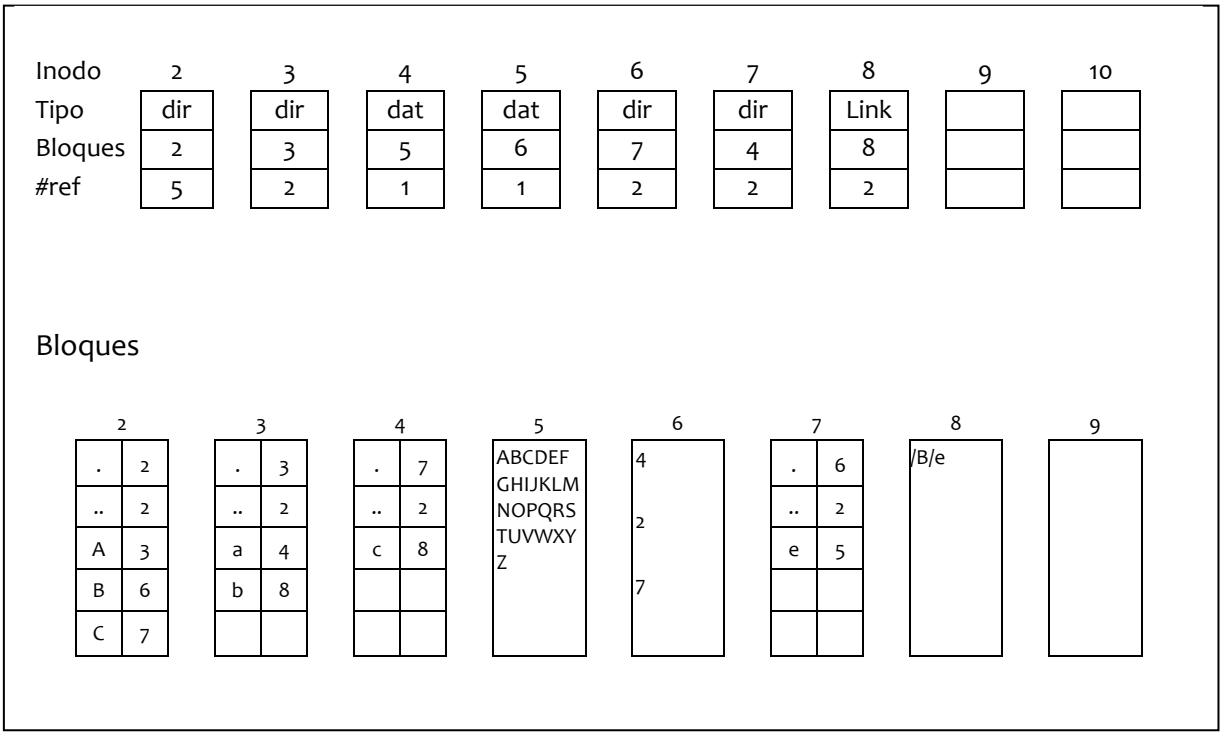
Bloques

2		3		4		5	6	7		8	9
.	2	.	3	.	7	ABCDEF GHIJKLM NOPQRS TUVWXY Z	4 2 7	.	6	/B/e	
..	2	..	2	..	2			..	2		
A	3	a	4	C	8			e	5		
B	6	b	8								
C	7										

a) Dibuja el grafo de directorios.
b) Indica qué actualizaciones se producirán tanto en los Inodos como en los bloques mostrados en el enunciado si ejecutamos las siguientes líneas de comandos (NOTA: "wc –c" devuelve por la salida estándar el número de caracteres que lee por la entrada estándar):

```
> ls / > /C/d

> cat /A/b | wc -c >> /C/d
```

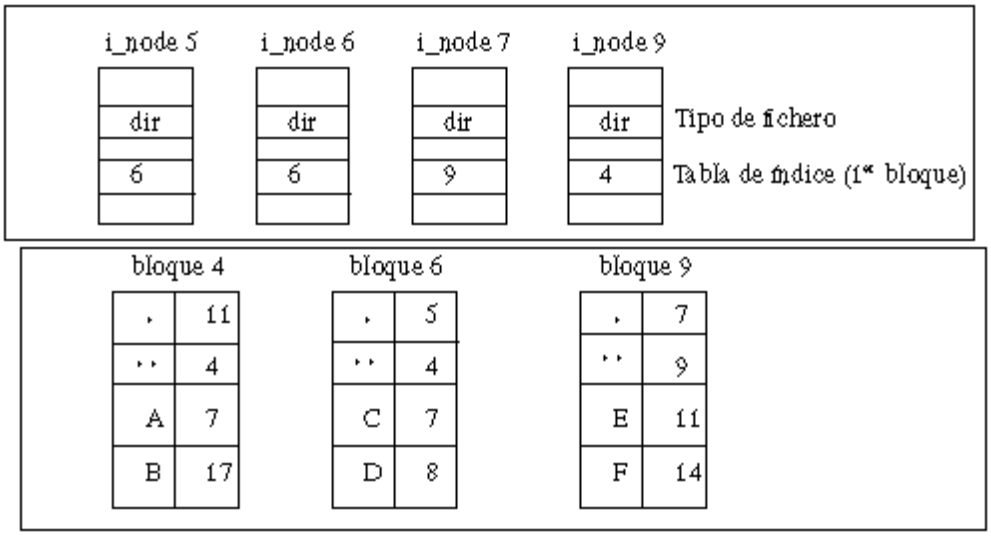


c) Describe qué accesos a disco se hacen al ejecutar la siguiente secuencia de sentencias, asumiendo que no hay buffer cache. Puedes asumir que el superbloque está cargado en memoria y que cada I-nodo tiene el tamaño de un bloque.

```
main(){
int fd;
int ret;
char c;
fd = open("/A/b",O_RDONLY);
ret = read(fd,&c,sizeof(c));
while (ret > 0){
        write(1, &c, sizeof(c));
        ret = read(fd,&c,sizeof(c));
}
close(fd);
}
```

**** Ejercicio 2 ****

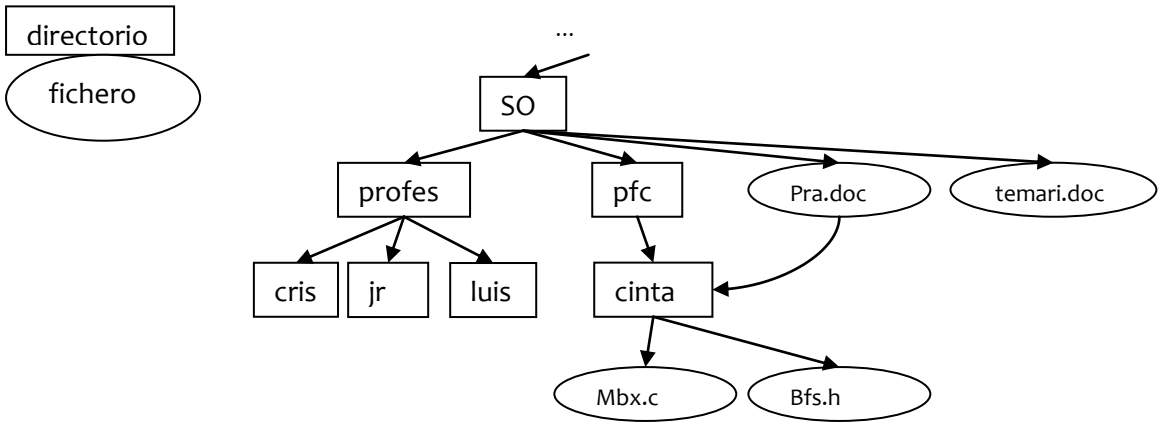
Dado este sistema de ficheros de UNIX, responde a las siguientes preguntas.



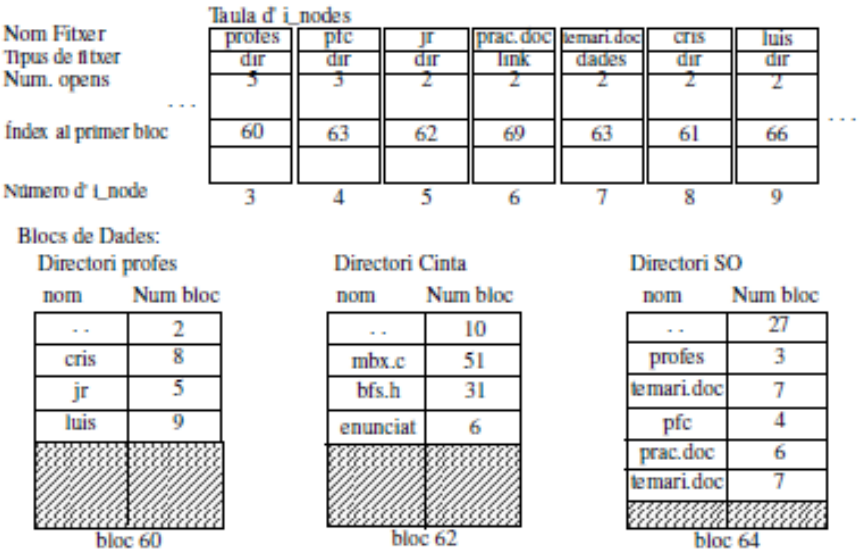
a) ¿Es coincidencia que el bloque 9 tenga como valor de i-nodo asociado a su fichero ".." el i-nodo 9?
b) ¿Es correcto que el primer bloque en los i_nodes 5 y 6 es común?
c) ¿Por qué el i-node 7 aparece en dos directorios?¿Qué tipo de fichero son A y C?

**** Ejercicio 3 ****

Tenemos el siguiente árbol de directorios

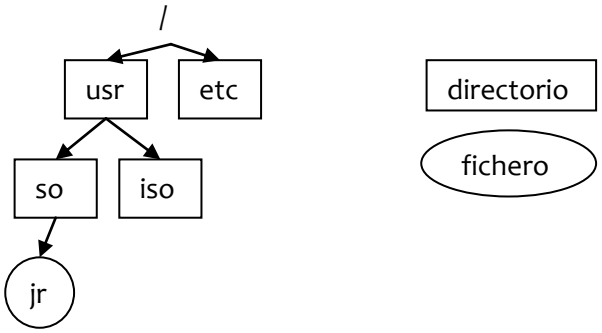


Sabiendo que está basado en un sistema UNIX (I-nodos), encuentra los errores en las siguientes estructuras de datos.

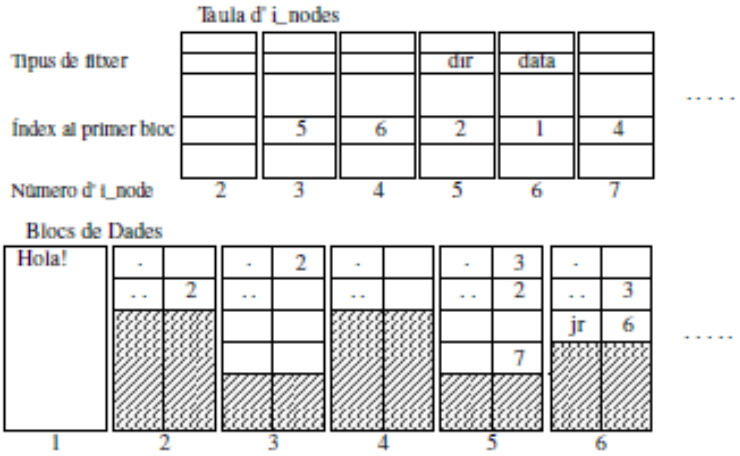


**** Ejercicio 4 ****

Dado el siguiente árbol de directorios



Sabiendo que es un sistema basado en i-nodos, rellena los campos que faltan



**** Ejercicio 5 ****

Bloque 1	
.	1
..	1
homeA	2
homeB	3
homeC	4

Bloque 2	
.	2
..	1
d	5

Bloque 3	
.	3
..	1
e	6

Bloque 4	
.	4
..	1
f	6

Bloque 5
/homeB/e/h

Bloque 6	
.	6
..	3
h	7

Tenemos los siguientes bloques de datos en un sistema de ficheros basado en UNIX:

En el cual:

- El I-nodo raíz es el 1.
- El orden de creación es: /, homeA, homeB, homeC, homeD, e, f, g, h.
- El orden de asignación de todos los I-nodos y bloques es el de creación.
- El contenido de "h" es texto que ocupa un tamaño par de bytes.
- El contenido de "d" es un soft-link.

a) Dibuja el grafo de directorios correspondiente al sistema de ficheros descrito en este ejercicio.

b) Siguiendo el orden de creación indicado anteriormente, ¿tendremos problemas al crear "d" por no existir "/homeB/e" en el momento de establecer el soft-link? Razona tu respuesta.

c) ¿Realmente puedo tener implementado este Sistema de Ficheros en un Sistema Operativo basado en UNIX? Si no es así, indica el motivo y qué alternativa propones para solucionarlo.
d) Si sabemos que "h" es un fichero muy grande (varios GBs) y queremos leer el último byte del fichero. ¿Qué sistema de asignación, de los vistos en clase, necesitará menos accesos a disco para leer ese byte? Razona brevemente tu respuesta.
e) Queremos implementar un código que copie la segunda mitad del contenido del fichero "h" en un nuevo fichero llamado "/homeB/hNuevo". Escribe el código necesario para poder realizar dicha copia, minimizando el número de transacciones a disco. NOTA: podemos declarar un vector de char de tamaño infinito. No es necesario comprobar los errores (podemos asumir que no hay errores en ninguna de las llamadas al sistema que hacemos).
f) Según el código que has escrito en el apartado anterior, indica los accesos a disco que se realizan para cada una de las llamadas al sistema que has puesto en el código. NOTA: No disponemos de buffer cache y el superbloque está cargado en memoria.