

A decorative graphic on the right side of the page. It features three sets of concentric circles in shades of blue. The top set is the largest, the middle set is smaller, and the bottom set is the largest again. Thin blue lines intersect these circles, creating a geometric pattern.

Exámenes de Teoría

Sistemas Operativos

Grau en Enginyeria Informàtica

Este documento contiene ejercicios de exámenes de teoría
que se han hecho anteriormente en esta asignatura

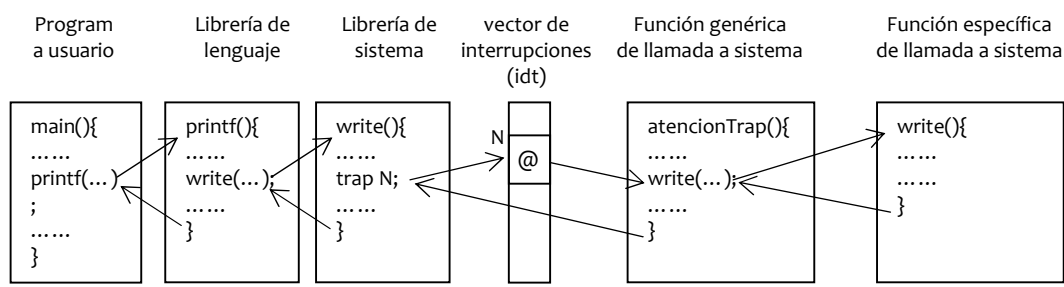
Profesores SO-Departamento AC
Primavera 2012

Índice

Enunciados	3
Preguntas Cortas.....	3
Ejercicios diversos (principalmente orientados a Procesos)	8
Ejercicios diversos (principalmente orientados a E/S y SF)	12
 Enunciados con Respuestas	 24
Preguntas Cortas.....	24
Ejercicios diversos (principalmente orientados a Procesos)	28
Ejercicios diversos (principalmente orientados a E/S y SF)	32

Preguntas cortas. Contesta y justifica TODAS las preguntas en el espacio asignado.

- 1. Indica qué limitaciones tiene el paso de parámetros a llamadas del sistema mediante registros y cómo se puede solucionar.
- 2. Dada la siguiente secuencia de ejecución que se da desde un programa de usuario, indica en qué modo de ejecución está el procesador para ejecutar cada una de ellas.



- 3. ¿Qué es el tiempo de espera de un proceso?
- 4. ¿Es cierta la siguiente afirmación?: Un proceso que está en estado swaped-out NO puede recibir signals porque no está en memoria principal.
- 5. Suponed que tenemos un sistema operativo que implementa una política de planificación de procesos no apropiativa. ¿Es necesario que este sistema disponga de una rutina de cambio de contexto? ¿Por qué?
- 6. Las direcciones que encontramos en un ejecutable ya cargado en memoria, ¿Son direcciones lógicas o físicas?
- 7. En un entorno multiusuario y multiproceso, pero que se ejecuta en una arquitectura que tiene 1 CPU, ¿Podemos tener varios procesos simultáneamente en la CPU? ¿Y cargados en memoria?

8. Explica cuál sería el problema principal que tendría un sistema que fuera basado en paginación con un único nivel de tabla de páginas. ¿Qué alternativas se han propuesto para solucionar este problema? Si para solucionar este problema decidimos aumentar el tamaño de página, ¿Qué problema estamos agravando entonces?
9. ¿Por qué la función `printf` de la librería de C no puede acceder directamente a la consola y ha de utilizar la llamada a sistema correspondiente?
10. Si un proceso que tiene signals pendientes ejecuta la llamada a sistema `fork`, ¿Los signals pendientes son heredados por el proceso hijo?
11. ¿Qué efectos tiene el uso de un quantum demasiado grande y un quantum demasiado pequeño en un planificador a corto plazo que emplee Round Robin?
12. Marca cuales de los elementos de la siguiente lista hereda un proceso de su padre y cuáles no:
 - a. El PID (no se hereda)
 - b. El espacio de direcciones lógico (se hereda)
 - c. La tabla con las rutinas de gestión asociadas a cada tipo de signal (se hereda)
 - d. Los signals pendientes (no se hereda)
 - e. Locks (se hereda)
13. En un sistema actual de propósito general, las direcciones (funciones, referencias a datos, etc) que encontramos en un fichero ejecutable en disco, ¿Son direcciones lógicas o físicas?
14. ¿Tiene sentido disponer de memoria virtual en un ordenador con 1GB de memoria, en el cual el procesador limita el tamaño máximo de un proceso a 512MBs?
15. Comenta las principales diferencias (desde el punto de vista de ventajas e inconvenientes) que tienen los modelos de segmentación y paginación.
16. ¿En qué consiste la optimización de carga bajo demanda? Razona si nos permite ahorrar tiempo, espacio de memoria o ambos.

17. ¿Qué consecuencias tendría en el número de accesos a disco eliminar la Tabla de Inodos?
18. ¿Qué es un dispositivo virtual y para qué se utiliza?
19. Explica qué es la fragmentación y pon un ejemplo de cada tipo de fragmentación que existe en un sistema de ficheros.
20. Qué mensaje aparecerá por pantalla si en nuestro sistema tenemos un proceso que ejecuta el siguiente código, sabiendo que ninguna llamada devuelve error y que las variables “aux” y “buffer” son vectores de caracteres de tamaño suficiente:
- ```
pipe(pd);

A=read (pd[0], buffer, 4096);

sprintf(aux, “%d bytes - contenido: %s\n”, A, buffer);
```
21. ¿Qué son y para qué se utilizan “major” y “minor”?
22. ¿En qué situaciones un “write” puede implicar que el proceso abandone el estado de RUNNING en su ciclo de vida?
23. ¿En qué se diferencia un sistema de ficheros con una configuración RAID que ofrezca robustez frente a un sistema de ficheros con Journaling?
24. Queremos usar el siguiente código para copiar el contenido del fichero  $f_1$  en el fichero  $f_2$

```
int main (int argc, char *argv[])
{
 int ret;
 char c;
 while ((ret=read(0,&c,sizeof(c)) > 0){
 write(1,&c,ret);
 }
}
```

¿Es necesario hacer algún cambio en el código para conseguir hacer la copia o se puede conseguir sin modificar el código? ¿Cómo tenemos que ejecutar el programa para hacer la copia?

25. ¿Qué es un device driver?

26. ¿Qué significa que una operación de entrada/salida sea bloqueante? ¿Cómo afecta a la planificación de procesos?

27. ¿Qué es un i-node?

28. ¿En qué consiste la asignación indexada de bloques de disco? ¿Qué información guarda asociada al nombre de un fichero un sistema de este tipo para poder localizar un bloque del fichero?

29. ¿En qué consiste la asignación encadenada de bloques de disco? ¿Qué información guarda asociada al nombre de un fichero un sistema de este tipo para poder localizar un bloque del fichero?

30. ¿En qué consiste la uniformidad de operaciones del interfaz de entrada/salida? ¿Para qué sirve?

31. Dada las siguientes líneas de código para ejecutar en un sistema Unix:

```
int fd,ret;
char c;

fd= open ("f1", O_RDONLY);
ret = read(fd,&c,sizeof(c));
```

¿Qué estructuras de datos en memoria y en disco consulta y/o modifica Unix en estas dos líneas?

- 32. ¿En qué tipo de dispositivos de almacenamiento tiene sentido utilizar un mecanismo de asignación de bloques contiguo?
- 33. Enumera cómo afecta (o puede afectar) la ejecución de una llamada a sistema open en las estructuras de datos en memoria de gestión de entrada salida de un S.O. Linux.
- 34. ¿Qué significa “redireccionar la salida estándar”?
- 35. ¿Es correcta la siguiente secuencia de código? Indica porqué.

```
int fd[2];
pipe(fd);
dup2(fd[0],1);
write(1,"hola",4);
```

- 36. Explica brevemente en qué consiste la optimización de “caching”, ¿qué ventajas tiene, y qué relación tiene, si crees que tiene, con la “buffer cache” que podemos encontrar en sistemas UNIX.
- 37. En un sistema para gestionar el disco, ¿qué mejora aporta un sistema indexado multinivel respecto a uno con un único nivel?

38. ¿Qué diferencia, en cuanto eficiencia, hay entre estos dos códigos?

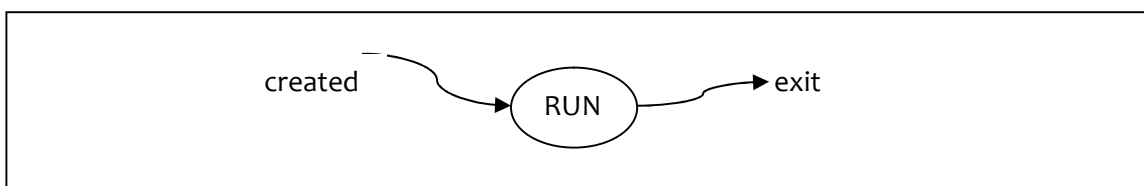
|                                                                                                                                                            |                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <pre>int tamanyo=0,fd; char c; fd=open("A",O_RDONLY); while (read(fd,&amp;c,sizeof(char))&gt;0)     tamanyo++; printf("El tamanyo es %d\n",tamanyo);</pre> | <pre>int tamanyo=0,fd; char c; fd=open("A",O_RDONLY); tamanyo=lseek(fd,0,SEEK_END); printf("El tamanyo es %d\n",tamanyo);</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|

39. ¿Qué diferencia hay entre un “hard link” y un “soft link”?

## Ejercicios diversos (principalmente orientados a Procesos)

### Ejercicio (3 puntos)

Estamos analizando el comportamiento de los procesos en un SO que estamos implantando. En el diseño inicial de nuestro SO disponemos de un sistema uniprogramado (sólo 1 proceso en el sistema en un instante), sin memoria virtual y que posee periféricos muy optimizados donde tan pronto se requiere realizar una comunicación la transferencia del dato es inmediata. Es decir, tarda 0 ciclos en completar la transferencia. Partiendo del diagrama de estados siguiente, suficiente para esta configuración, complétalo si vamos añadiendo las siguientes funcionalidades al sistema operativo:



- Tras una mejora en el SO ahora permite multiprogramación, dibuja el nuevo ciclo de vida y comenta brevemente los cambios que has hecho.
- Tras otra mejora, disponemos de un sistema que ofrece comunicación mediante signals. Dibuja el nuevo ciclo de vida y comenta brevemente los cambios que has hecho.



- c. Tras una última mejora, ahora disponemos de un planificador a medio plazo. Dibuja el nuevo ciclo de vida y comenta brevemente los cambios que has hecho

### Ejercicio (3 puntos)

El siguiente código pertenece al programa “familia”. Sabemos que el programa en memoria ocupa 5KB. Nuestro sistema informático dispone de 1MB de memoria y su gestión de memoria no implementa memoria virtual ni carga bajo demanda. El sistema tiene un máximo global de 100 procesos. Justifica todas las respuestas.

```
int main (int argc, char *argv[])
{
 int i, pid, ret;
 for (i=0; i<5; i++){
 pid=fork();
 If (pid==0){
 execlp("familia", "familia", (char *)0);
 exit(0);
 }
 If (pid<0) control_error();
 }
 for (i=0; i<5; i++) waitpid(-1, &ret, 0);
}
```

- a) Dibuja, en el cuadro de al lado del código, el árbol de procesos que se generaría al ejecutar el programa
- b) Describe en qué momento finalizará la ejecución de dicho código asumiendo que nuestro SO **no tiene** la optimización de Copy-on-Write.
- c) Describe en qué momento finalizará la ejecución de dicho código, suponiendo que nuestro SO **tiene** la optimización de Copy-on-Write.
- d) ¿Cómo se comportaría el programa si en lugar de crear procesos creáramos threads?

### Ejercicio (3 puntos)

Dado el siguiente código, contesta a las siguientes preguntas. Justifica todas las respuestas:

- a) ¿Cuántas veces se ejecutará la función `haz_tarea()`?
- b) Si queremos que el código de los procesos se ejecute de forma concurrente, ¿hay que hacer alguna modificación? Indica cual.
- c) Suponiendo que no hay ningún error, ¿Cuántos procesos se crearán?
- d) Dibuja, en el cuadro de al lado del código, el árbol de procesos que se crearía si quitáramos el `exit` de la función `hacer_tarea` y que valor recibiría cada instancia de la función como parámetro (la `i`)

```
void hacer_tarea(int i)
{
 ...
 exit(0);
}
int main (int argc, char *argv[])
{
 int i, pid, ret;
 for (i=0; i<5; i++){
 pid=fork();
 If (pid==0) hacer_tarea(i);
 If (pid<0) control_error();
 }
 for (i=0; i<5; i++) waitpid(-1, &ret, 0);
}
```

### Ejercicio (3 puntos)

Dado el siguiente código de usuario para ejecutar sobre Linux. Justifica todas las respuestas

- a) Dibuja, en el cuadro de al lado del código, el árbol de procesos que se genera. Indica en el dibujo qué proceso(s) recibirán el signal SIGINT, qué proceso(s) enviarán el signal SIGINT , y qué proceso(s) ejecutará(n) el bucle de la línea 27 (while(1);)
- b) ¿Cuántos procesos ejecutarán las líneas 17 y 18?
- c) ¿Y la rutina gestionSignal?
- d) ¿Qué mensajes veremos en la consola y cuántas veces aparecerán?

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <pre>1: int pidh[2]; 2: void gestionSignal(int signum){ 3:     int valor; 4:         kill(pidh[1],SIGKILL); 5:         waitpid(pidh[1],&amp;valor,0); 6:         exit(1); 7:     } 8:     main(){ 9:         int i, valor, padre; 10:        int proclnicial = getpid(); 11:        signal(SIGINT, gestionSignal); 12:        pidh[0] = fork(); 13:        pidh[1] = fork(); 14:        if (pidh[1] &gt; 0){ 15:            printf("Proceso padre\n"); 16:            waitpid(pidh[1],&amp;valor,0); 17:            kill (pidh[0],SIGINT); 18:            waitpid(pidh[0],&amp;valor,0); 19:        } else { 20:            /* la llamada a sistema getppid 21:             retorna el pid del padre del 22:             proceso que la ejecuta 23:             */ 24:            padre = getppid() 25:            if (padre != proclnicial) { 26:                while(1); 27:            } 28:        } 29:        printf("Final de la ejecución\n"); 30:    } 31: } 32: 33:</pre> |  |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|

# Ejercicios diversos (principalmente orientados a E/S y SF)

## Ejercicio (3 puntos)

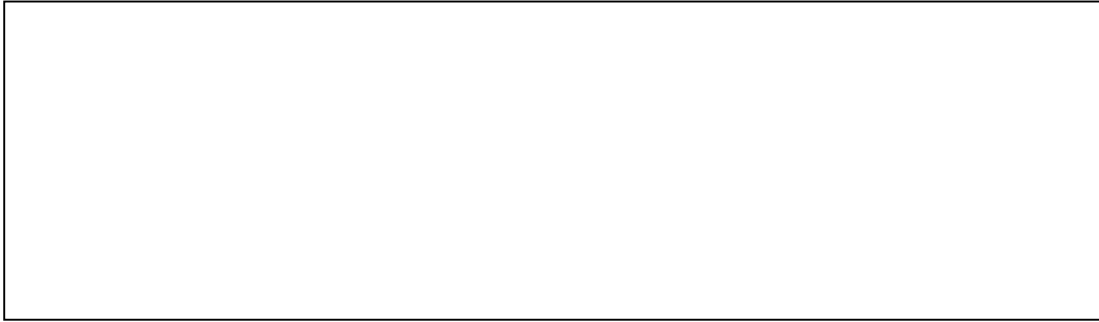
Tenemos los siguientes I-nodos y bloques de datos en un SF basado en I-nodos:

| Inodo   | Inodo | Inodo | Inodo | Inodo | Inodo | Inodo | Inodo |
|---------|-------|-------|-------|-------|-------|-------|-------|
|         | 1     | 2     | 3     | 4     | 5     | 6     | 7     |
| Tipo    | DIR   | DIR   | DIR   | DIR   | Link  | DIR   | Datos |
| #Refs   | 5     | 2     | 3     | 2     | 1     | 2     | 2     |
| Bloques | 1     | 2     | 3     | 4     | 5     | 6     | 7     |

| Bloque 1 |   | Bloque 2 |   | Bloque 3 |   | Bloque 4 |   | Bloque 5   | Bloque 6 |   | Bloque 7                            |
|----------|---|----------|---|----------|---|----------|---|------------|----------|---|-------------------------------------|
| .        | 1 | .        | 2 | .        | 3 | .        | 4 | /homeB/e/h | .        | 6 | Este es un<br>texto de<br>prueba... |
| ..       | 1 | ..       | 1 | ..       | 1 | ..       | 1 |            | ..       | 3 |                                     |
| homeA    | 2 | d        | 5 | e        | 6 | f        | 7 |            | h        | 7 |                                     |
| homeB    | 3 |          |   |          |   |          |   |            |          |   |                                     |
| homeC    | 4 |          |   |          |   |          |   |            |          |   |                                     |

a) (1 punto) Dibuja el grafo de directorios correspondiente al sistema de ficheros descrito en este ejercicio.

b) (1 punto) Queremos implementar un código que copie la segunda mitad del contenido del fichero “homeA/d” en un nuevo fichero llamado “/homeB/hNuevo”. Escribe el código necesario para poder realizar dicha copia, MINIMIZANDO el número de llamadas a sistema que acceden 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).



- c) (1 punto) Indica los accesos a disco que se realizan para cada una de las llamadas al sistema de la siguiente sección de código (es decir, las siguientes instrucciones se corresponden a instrucciones consecutivas dentro de un código determinado). NOTA: No disponemos de buffer cache, el superbloque está cargado en memoria, cada l-nodo tiene el tamaño de un bloque y la variable “buf” está declarada previamente como “char buf[256]”.

```
fd = open("/homeA/d", O_RDWR);
```

```
lseek(fd, 0, SEEK_END);
```

```
read(fd, buf, 100);
```

```
write(fd, buf, 100);
```

## Ejercicio (3,5 puntos)

A partir del siguiente código contesta a las preguntas que se te formulan en los apartados.

```
1: void procesar()
2: {
3: char c;
4:
5: while (read(0, &c, sizeof(char)) > 0)
6: write(1, &c, sizeof(char));
7:
8: exit(0);
9:}

10: int main(int argc, char *argv[])
11: {
12: int fd_ent, fd_sal;
13: int pids[3];
14:
15: fd_ent=open("in.txt", O_RDONLY);
16: fd_sal=open("out.txt", O_WRONLY|O_CREAT|O_TRUNC);
17: for (i=0; i<2; i++){
18: pids[i]=fork();
19: if (pids[i] == 0){
20: dup2(fd_ent, 0);
21: dup2(fd_sal, 1);
22: procesar();
23: }
24: }
25:
26: close(fd_ent);
27: while(waitpid(-1, NULL, 0)>0);
28:}
```

Diagram illustrating the code structure and callout points:

- Point C points to line 4: `char c;`
- Point A points to line 15: `fd_ent=open("in.txt", O_RDONLY);`
- Point B points to line 20: `dup2(fd_ent, 0);`

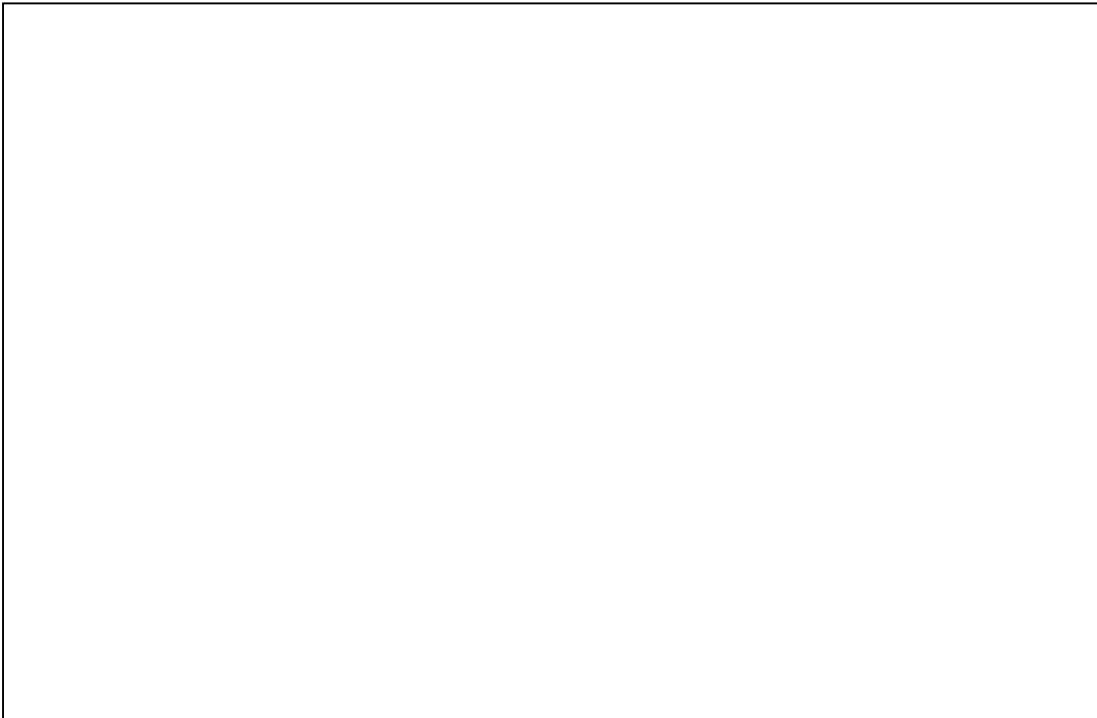
- a) (1 punto) Indica todas las estructuras de datos de memoria y de disco que se acceden y/o actualizan a lo largo de la ejecución de las llamadas al sistema que tenemos en los puntos “A” y “B”.

b) (0,5 puntos) Indica cuál sería el resultado final de la ejecución de este código si ...

... la línea 15 pasa a estar entre las líneas 19 y 20:

... la línea 16 pasa a estar entre las líneas 19 y 20:

c) (1 punto) Dibuja el estado de las TC, TFA y T-Inodos en el instante indicado en el punto “C”. NOTA: asumimos que todos los procesos hijo ya se han creado y se encuentran al mismo tiempo en ese punto (justo antes del read).



d) (1 puntos) Debido a una futura modificación del código, nos interesa que cada uno de los procesos que hemos creado mediante el fork conozcan los PIDs de todos sus otros “procesos hermanos”. Indica si se puede obtener esta información (y en caso afirmativo describe brevemente cómo lo harías) mediante la utilización de:

i. Signals:

ii. Pipes:

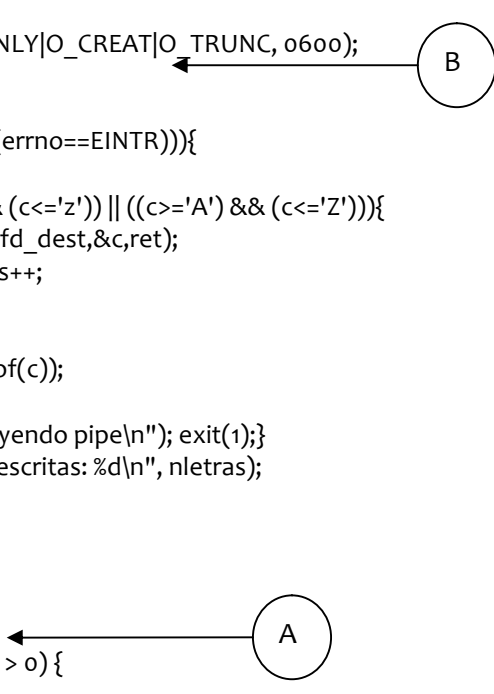
## Ejercicio (4 puntos)

Tenemos el siguiente fichero con un programa para ejecutar en Linux:

```
void trat_alarma(int signum){
char buf[50];
 signal(SIGALRM, trat_alarma);
 strcpy(buf, "Han pasado 10 segundos más\n");
 write(2,buf,strlen(buf));
 alarm(10);
}

main(int argc, char *argv[]){
int fd[2], fd_dest,fd_src;
int ret, pidh, nletras=0;
char c, buf[50];
 siginterrupt(SIGALRM,1);
 signal(SIGALRM, trat_alarma);
 pipe(fd);
 pidh = fork();
 if (pidh == -1) { perror("Error creando hijo"); exit(1);}

 If (pidh == 0) {
 fd_dest=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC, 0600);
 alarm(10);
 ret=read(fd[0],&c,sizeof(c));
 while ((ret>0) || ((ret == -1) && (errno==EINTR))){
 if (ret > 0){
 if (((c>='a') && (c<='z')) || ((c>='A') && (c<='Z'))){
 write(fd_dest,&c,ret);
 nletras++;
 }
 }
 ret=read(fd[0],&c,sizeof(c));
 }
 if (ret<0) { perror("Error hijo leyendo pipe\n"); exit(1);}
 sprintf(buf, "Numero de letras escritas: %d\n", nletras);
 write(2,buf,strlen(buf));
 exit(0);
 }
 close(fd[0]);
 fd_src=open(argv[1],O_RDONLY);
 while ((ret = read(fd_src, &c, sizeof(c))) > 0) {
 write(fd[1], &c, sizeof(c));
 }
 close(fd_src); close(fd[1]);
 waitpid(pidh, NULL, 0);
 strcpy(buf, "Padre acaba la ejecucion\n");
 write(1,buf,strlen(buf));
}
```



Podéis suponer que las llamadas a sistema de este código se ejecutan sin dar error y que el usuario lo ejecuta sin redireccionar ningún canal estándar y pasándole unos parámetros correctos.



- a) Representa el estado de las tablas de canales y de la tabla de ficheros abiertos, suponiendo que el proceso padre se encuentra ejecutando la línea A (justo después de `open(argv[1]...)` ) y el proceso hijo está en la línea B (justo después de `open(argv[2]...)` ).



- b) Este programa contiene un error y si lo ejecutáramos no acabaría ninguno de los dos procesos ¿Por qué? ¿Cómo lo arreglarías?

- c) Queremos que el código de tratamiento que ejecuta el hijo forme parte de un programa separado que el hijo cargue mediante la llamada `execvp`. Y proponemos los siguientes códigos. El código inicial se mantiene igual excepto las líneas marcadas en negrita. Y el código del nuevo programa contiene las líneas que en la versión anterior formaban parte del código ejecutado por el proceso hijo.

```
/* fichero padre.c */
```

```
void trat_alarma(int signum){
char buf[50];
 signal(SIGALRM, trat_alarma);
 strcpy(buf, "Han pasado 10
segundos más\n");
 write(2,buf,strlen(buf));
 alarm(10);
}

main(int argc, char *argv[]){
int fd[2], fd_src;
int ret, pidh;
char c, buf[50];
 siginterrupt(SIGALRM,1);
 signal(SIGALRM, trat_alarma);
 pipe(fd);
 pidh = fork();
 if (pidh == -1) {
 perror("Error creando
hijo");
 exit(1);
 }
 if (pidh == 0) {
 execlp("./hijo", "hijo",
NULL);
 perror("Error en el
execlp"); exit(1);
 }
 close(fd[0]);
 fd_src=open(argv[1],O_RDONLY);
 while ((ret = read(fd_src, &c,
sizeof(c))) > 0) {
 write(fd[1], &c, sizeof(c));
 }
 close(fd_src); close(fd[1]);
 waitpid(pidh, NULL, 0);
 strcpy(buf, "Padre acaba la
ejecucion\n");
 write(1,buf,strlen(buf));
}
```

```
/*fichero hijo.c */
```

```
main() {
int fd_dest, ret, nletras = 0;
int fd[2];
char c, buf[50];

 fd_dest=open(argv[2],
O_WRONLY|O_CREAT|O_TRUNC, 0600);
 alarm(10);
 ret=read(fd[0],&c,sizeof(c));
 while ((ret>0) || ((ret==-1) &&
(errno==EINTR))){
 if (ret > 0){
 if (((c>='a') && (c<='z'))
|| ((c>='A') && (c<='Z'))){
 write(fd_dest,&c,ret);
 nletras++;
 }
 }
 ret=read(fd[0],&c,sizeof(c));
 }

 if (ret<0) {
 perror("Error hijo leyendo
pipe\n");
 exit(1);
 }

 sprintf(buf, "Numero de letras escritas:
%d\n", nletras);
 write(2,buf,strlen(buf));
}
```

¿Cuál será el resultado de ejecutar este código? ¿Tendrá la misma funcionalidad que el código original o es necesario modificar alguna cosa más? En caso de que sea necesario introducir algún cambio más describe claramente qué cambio sería, indicando si afecta al código del fichero padre.c o hijo.c

### Ejercicio (2 puntos)

Queremos escribir un programa que duplique el contenido de un fichero. Por ejemplo, si el contenido de `/users/home/f1` es 'abcd' después de ejecutar nuestro programa su contenido tiene que ser 'abcdabcd'. Proponemos el siguiente código:

```
main(int argc, char *argv[]){
 int fd1, fd2;
 fd1 = open(argv[1], O_RDWR);
 fd2 = dup (fd1);
 lseek(fd2, 0, SEEK_END);
 while ((ret = read(fd1, &c, sizeof(c))) > 0){
 write(fd2, &c, ret);
 }
 close(fd1);
 close(fd2);
}
```

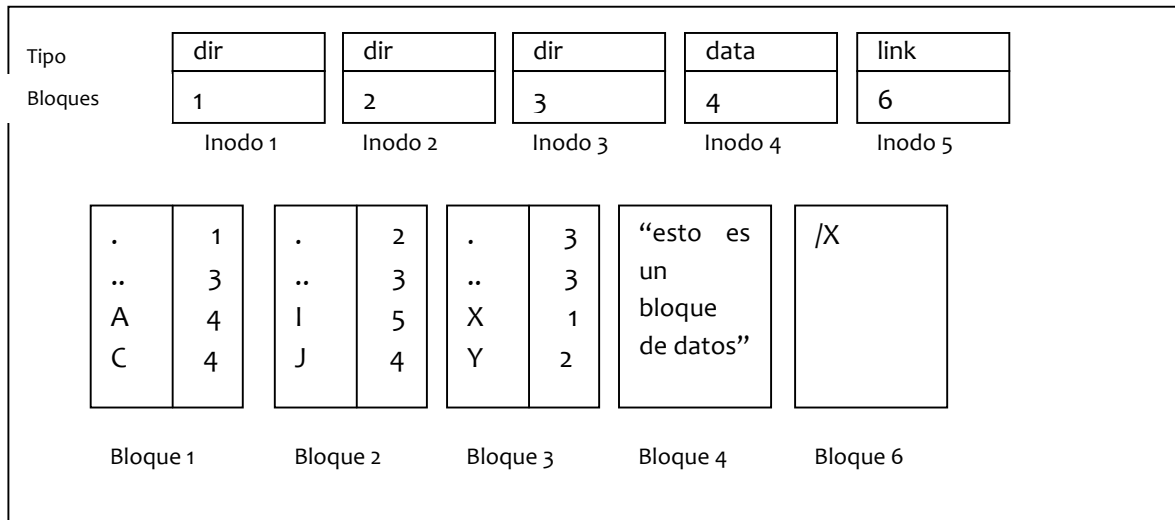
- a) Después de ejecutarlo comprobamos que el resultado no es el esperado. Suponiendo que le pasamos como parámetro `/users/home/f1` que inicialmente contiene 'abcd', ¿cómo se comportará este programa? ¿cómo afectará al fichero `/users/home/f1`? ¿por qué?

b) Indica qué accesos a disco generarían las 4 primeras sentencias de este código (open, dup, lseek y primer read del bucle) si le pasamos como parámetro el fichero `/users/home/f1` asumiendo que:

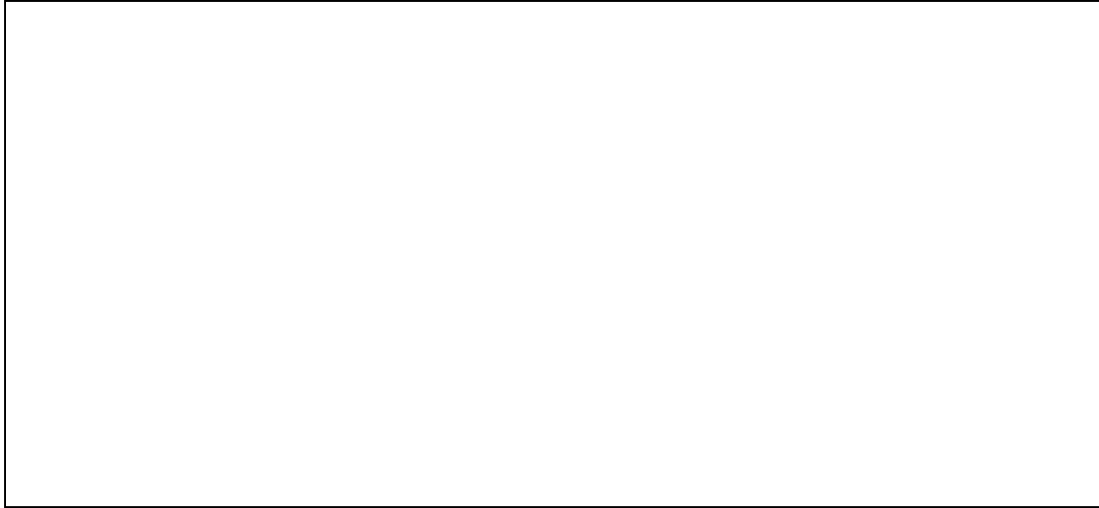
- Acabamos de iniciar la máquina y ningún otro proceso está usando ningún fichero
- El fichero `/users/home/f1` existe y su contenido es 'abcd'
- No tenemos buffer cache
- Los i-nodos ocupan 1 bloque
- Los directorios que leemos también ocupan 1 bloque
- El superbloque está en memoria

### Ejercicio (3 puntos)

Suponed un sistema de ficheros (basado en i-nodos) que tiene los siguientes i-nodos y bloques de datos:



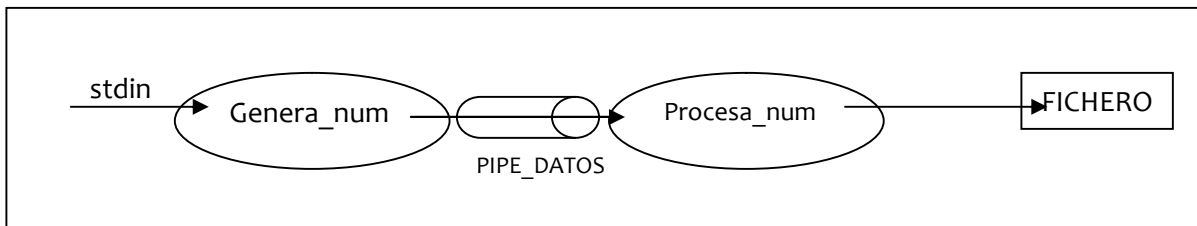
a) Dibuja el grafo de directorios que se puede deducir de este esquema de i-nodos y bloques de datos



- b) ¿Cuál crees que es el i-nodo raíz? (márcalo en el dibujo)
  - c) Indica que ficheros son soft-link y que ficheros son hard-link. En el caso de los soft-links indica mediante flechas los ficheros a los que apunta (márcalo en el dibujo)
  - d) Indica el número de accesos a disco que se generaría si ejecutamos la llamada a sistema `open("/Y/I/A",O_RDONLY)`. Asume que el superbloque está en memoria. Indícalo en el caso que el sistema tenga buffer cache y en el caso que no tenga (asume que los i-nodos ocupan 1 bloque de disco)
- 
- e) Indica además, en el caso anterior, que i-nodo se guardaría en la tabla de i-nodos en memoria

### Ejercicio (3 puntos)

Queremos hacer un programa que estará dividido en dos partes, una que genera números y otra que realiza un cálculo con esos números y guarda el resultado en un fichero que se le pasa como parámetro. El esquema es el siguiente, donde el programa `Genera_num` lee números de su entrada `std`, aplica un cálculo a cada número y el resultado lo escribe en la pipe con nombre `"PIPE_DATOS"` (que ya está creada). El programa `Procesa_num` lee los números de la pipe, hace un segundo cálculo también a cada número, y escribe el resultado en el fichero que recibe como parámetro.



Sin embargo, detectamos que los números se generan más rápido de los que se procesan, por lo que decidimos hacerlo en paralelo, y le pasamos un segundo parámetro al programa `Procesa_num` con el número de procesos que queremos que se creen. Nos ofrecen esta solución, pero nos encontramos que no es correcta, vemos que el fichero de salida no tiene todos los números que debería (hemos quitado el control de error, includes, etc para que ocupe menos):

```

int fd_ent, fd_sal;
int nums, nuevo_num;
int i, num_procs, pids[MAX_PROCS];
int calcula(int n)
{
 int res;
 // Calculo sobre n
 return res;
}
void hijo(void)
{
 //INICIO LECTURAS
 while(read(0, &nums, sizeof(int)) > 0) {
 nuevo_num = calcula(nums);
 write(1, &nuevo_num, sizeof(nuevo_num));
 }
 exit(0);
}
int main(int argc, char *argv[])
{
 num_procs = min(atoi(argv[2]), MAX_PROCS);
 for (i = 0; i < num_procs; i++) {
 pids[i] = fork();
 if (pids[i] == 0) {
 fd_ent = open("PIPE_DATOS", O_RDONLY);
 dup2(fd_ent, 0); close(fd_ent);
 fd_sal = open(argv[1], ...); // Faltan los flags
 dup2(fd_sal, 1); close(fd_sal);
 hijo();
 }
 }
 while(waitpid(-1, NULL, 0) > 0);
}

```

- a) Indica qué valores tendríamos que poner en los flags que le pasamos al `open` para crear un nuevo fichero de salida en cada ejecución del programa.

- b) Dibuja cómo quedarían las estructuras de gestión de E/S en memoria (tabla canales, tabla ficheros abiertos y tabla de i-nodos) en la línea de código marcada como “INICIO LECTURAS” si ejecutamos el programa de la siguiente forma “Procesa\_num NUMS\_EJ1 2” (asumiendo que en ese punto los procesos ya están creados.)
- c) Una vez analizadas las tablas, vemos que con una pequeña modificación se puede arreglar el código. Indica claramente que líneas de código habría que mover o modificar para que el programa se comporte como la versión secuencial (el orden de los números en el fichero de salida no es importante).
- d) Una vez arreglado, queremos hacer una extensión para permitir que la cantidad de procesos que colaboran sea menor al cabo de un tiempo ya que hemos detectado que después de un tiempo la velocidad de generar datos disminuye considerablemente y tenemos a los procesos del programa Procesa\_num bloqueados. ¿En qué línea de código estarán bloqueados los procesos hijos?
- e) Se nos ocurre una solución basada en signals, el programa Procesa\_num cada vez que reciba un SIGUSR1 ejecutará la siguiente función para terminar con un proceso hijo:
- a) ¿Cómo (línea de código) y dónde tenemos que reprogramar el SIGUSR1?

```
void tratar_SIGUSR1(int s)
{
 char b[128];
 sprintf(b, "Termino al proceso %d\n", pids[num_procs]);
 write(1, b, strlen(b));
 kill(pids[num_procs], SIGUSR1);
 num_procs--;
 signal(SIGUSR1, tratar_SIGUSR1);
}
```

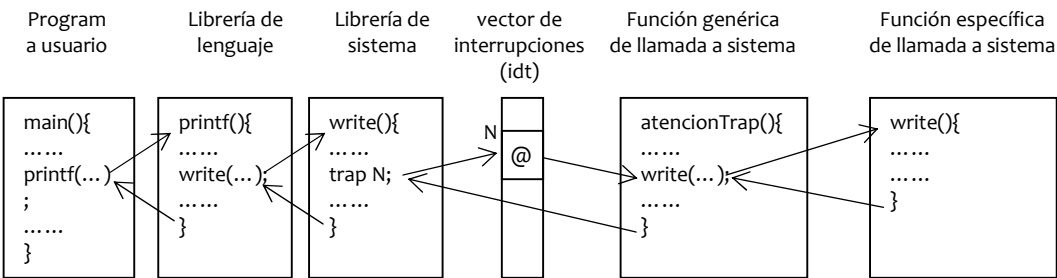
# Preguntas cortas. Contesta y justifica TODAS las preguntas en el espacio asignado.

1. Indica qué limitaciones tiene el paso de parámetros a llamadas del sistema mediante registros y cómo se puede solucionar.

Problema: La cantidad de parámetros que se pueden pasar está limitada. Una posible solución es usar un registro para pasar el puntero a una estructura, de forma que podemos pasar cuantos queramos.

2. Dada la siguiente secuencia de ejecución que se da desde un programa de usuario, indica en qué modo de ejecución está el procesador para ejecutar cada una de ellas.

Modo usuario: Programa usuario, librería de lenguaje y de sistema. Modo kernel: función genérica y función específica de llamada a sistema.



3. ¿Qué es el tiempo de espera de un proceso?

Es el tiempo que un proceso pasa en la cola de ready.

4. ¿Es cierta la siguiente afirmación?: Un proceso que está en estado swaped-out NO puede recibir signals porque no está en memoria principal.

No, sí el proceso existe puede recibir signals, otra cosa es cuando los procesará.

5. Suponed que tenemos un sistema operativo que implementa una política de planificación de procesos no apropiativa. ¿Es necesario que este sistema disponga de una rutina de cambio de contexto? ¿Por qué?

Sí. En cualquier caso es necesaria porque la rutina de cambio de contexto sirve para pasar de un proceso a otro, y aunque la política sea no apropiativa, el proceso puede bloquearse o terminar.

6. Las direcciones que encontramos en un ejecutable ya cargado en memoria, ¿Son direcciones lógicas o físicas?

Son direcciones lógicas.

7. En un entorno multiusuario y multiproceso, pero que se ejecuta en una arquitectura que tiene 1 CPU, ¿Podemos tener varios procesos simultáneamente en la CPU? ¿Y cargados en memoria?

Con 1 CPU podemos tener 1 proceso en la CPU pero varios en memoria.

8. Explica cuál sería el problema principal que tendría un sistema que fuera basado en paginación con un único nivel de tabla de páginas. ¿Qué alternativas se han propuesto para solucionar este problema? Si para solucionar este problema decidimos aumentar el tamaño de página, ¿Qué problema estamos agravando entonces?

El principal problema es el tamaño de la tabla de página. Una posible solución es tener un sistema de paginación multinivel. Si aumentamos el tamaño de la página se agrava el problema de la fragmentación interna.



9. ¿Por qué la función `printf` de la librería de C no puede acceder directamente a la consola y ha de utilizar la llamada a sistema correspondiente?

*Porque la consola es un dispositivo y por tanto el acceso ha de hacerse desde modo sistema.*

10. Si un proceso que tiene signals pendientes ejecuta la llamada a sistema `fork`, ¿Los signals pendientes son heredados por el proceso hijo?

*Los signals pendientes no se heredan, son propios de cada proceso.*

11. ¿Qué efectos tiene el uso de un quantum demasiado grande y un quantum demasiado pequeño en un planificador a corto plazo que emplee Round Robin?

*Un  $\alpha$  demasiado grande generaría un efecto similar a una FIFO, y un  $\alpha$  demasiado pequeño provocaría mucho *overhead* debido a la gran cantidad de cambios de contexto por unidad de tiempo.*

12. Marca cuales de los elementos de la siguiente lista hereda un proceso de su padre y cuáles no:

- a. El PID (no se hereda)
- b. El espacio de direcciones lógico (se hereda)
- c. La tabla con las rutinas de gestión asociadas a cada tipo de signal (se hereda)
- d. Los signals pendientes (no se hereda)
- e. Locks (se hereda)

13. En un sistema actual de propósito general, las direcciones (funciones, referencias a datos, etc) que encontramos en un fichero ejecutable en disco, ¿Son direcciones lógicas o físicas?

*Lógicas*

14. ¿Tiene sentido disponer de memoria virtual en un ordenador con 1GB de memoria, en el cual el procesador limita el tamaño máximo de un proceso a 512MBs?

*Sí, porque en cualquier caso nos permitiría tener más procesos en memoria.*

15. Comenta las principales diferencias (desde el punto de vista de ventajas e inconvenientes) que tienen los modelos de segmentación y paginación.

*Paginación es más sencillo de implementar ya que las partes en que se divide la memoria tienen el mismo tamaño pero sufre de fragmentación interna. Segmentación es más compleja de implementar ya que las partes son de tamaño variable. Sufre de fragmentación externa.*

16. ¿En qué consiste la optimización de carga bajo demanda? Razona si nos permite ahorrar tiempo, espacio de memoria o ambos.

*Consiste en cargar los programas en memoria a medida que los vamos referenciando. Si el programa no ejecuta todo su código permite ahorrar memoria y tiempo.*

17. ¿Qué consecuencias tendría en el número de accesos a disco eliminar la Tabla de Inodos?

*Aumentaría la cantidad de accesos a disco, ya que ahora no disponemos de los inodos de los ficheros (que estamos utilizando) en memoria y no sabemos cómo acceder a sus bloques de datos.*

18. ¿Qué es un dispositivo virtual y para qué se utiliza?

*También conocido como canal, es la asociación entre un nombre simbólico (fichero) y la aplicación. Sirve para ofrecer a la aplicación un acceso independiente del dispositivo lógico/físico que estamos accediendo e independiente del resto de procesos.*

19. Explica qué es la fragmentación y pon un ejemplo de cada tipo de fragmentación que existe en un sistema de ficheros.

*Fragmentación es cuando tenemos espacio libre pero no utilizable. Los esquemas de asignación contiguo, por ejemplo, sufren de fragmentación externa y los esquemas de bloques (por ejemplo inodos) sufren de fragmentación interna.*

20. ¿Qué mensaje aparecerá por pantalla si en nuestro sistema tenemos un proceso que ejecuta el siguiente código, sabiendo que ninguna llamada devuelve error y que las variables "aux" y "buffer" son vectores de caracteres de tamaño suficiente:

Ninguno, ya que el read se quedará bloqueado

```
pipe(pd);

A=read (pd[o], buffer, 4096);

sprintf(aux, "%d bytes - contenido: %s\n", A, buffer);
```

**21. ¿Qué son y para qué se utilizan “major” y “minor”?**

Major y minor son dos números que sirven para identificar un dispositivo físico. El major identifica el tipo de dispositivo y el minor qué instancia del tipo de dispositivo “major”.

**22. ¿En qué situaciones un “write” puede implicar que el proceso abandone el estado de RUNNING en su ciclo de vida?**

En caso que el write sea a un dispositivo bloqueante (y se bloquee), por ejemplo, al escribir en una pipe que está llena.

**23. ¿En qué se diferencia un sistema de ficheros con una configuración RAID que ofrezca robustez frente a un sistema de ficheros con Journaling?**

RAID nos ofrece robustez respecto a fallos físicos del disco, una vez se ha escrito los datos se pueden recuperar. Journaling nos da robustez en el caso de que no se haya llegado a escribir los datos.

**24. Queremos usar el siguiente código para copiar el contenido del fichero f1 en el fichero f2**

```
int main (int argc, char *argv[])
{
 int ret;
 char c;

 while ((ret=read(o,&c,sizeof(c)) > 0){
 write(1,&c,ret);
 }
}
```

¿Es necesario hacer algún cambio en el código para conseguir hacer la copia o se puede conseguir sin modificar el código? ¿Cómo tenemos que ejecutar el programa para hacer la copia?

Se puede conseguir sin modificar el código. Tendremos que ejecutarlo redireccionando la entrada y la salida estándar, desde la Shell: copia < f1 > f2

**25. ¿Qué es un device driver?**

Es el código de más bajo nivel para gestionar un dispositivo físico

**26. ¿Qué significa que una operación de entrada/salida sea bloqueante? ¿Cómo afecta a la planificación de procesos?**

Bloqueante significa que el proceso puede abandonar la CPU y pasar a la cola de bloqueados a esperar que termine la operación de entrada/salida. Implica añadir ese estado de bloqueado.

**27. ¿Qué es un i-node?**

Es la estructura que se utiliza en sistema UNIX para guardar la información relacionada con un fichero.

28.¿En qué consiste la asignación indexada de bloques de disco? ¿Qué información guarda asociada al nombre de un fichero un sistema de este tipo para poder localizar un bloque del fichero?

La asignación indexada consiste en guardar un bloque (o varios) de índices directos a bloques de datos. La información asociada al nombre del fichero es el número de bloque que contiene los índices.

29.¿En qué consiste la asignación encadenada de bloques de disco? ¿Qué información guarda asociada al nombre de un fichero un sistema de este tipo para poder localizar un bloque del fichero?

Consiste en tener los bloques de un fichero “enlazados”, es decir, que un bloque apunta al siguiente. La información puede estar en los mismos bloques de datos o en una tabla aparte (FAT). La información asociada al nombre es el número del primer bloque de datos.

30.¿En qué consiste la uniformidad de operaciones del interfaz de entrada/salida? ¿Para qué sirve?

Consiste en tener las mismas operaciones de entrada/salida independientemente del dispositivo físico al que se acceder. Sirve para ofrecer independencia a las aplicaciones.

31. Dada las siguientes líneas de código para ejecutar en un sistema Unix:

```
int fd,ret;
char c;

fd= open ("f1", O_RDONLY);
ret = read(fd,&c,sizeof(c));
```

¿Qué estructuras de datos en memoria y en disco consulta y/o modifica Unix en estas dos líneas?

En disco: El open accede al i-nodo y al bloque de datos del directorio actual, y al i-nodo del fichero f1. El read lee el primer bloque de datos de f1

En memoria: Se guarda el i-nodo de f1 en la tabla de i-nodos. Se añade una entrada en la TFA y un canal nuevo en la tabla de canales. El read modifica la entrada de la TFA (el puntero de l/e)

32.¿En qué tipo de dispositivos de almacenamiento tiene sentido utilizar un mecanismo de asignación de bloques contiguo?

En aquellos que no se modifiquen, por ejemplo cd-roms, dvd's.

33. Enumera cómo afecta (o puede afectar) la ejecución de una llamada a sistema open en las estructuras de datos en memoria de gestión de entrada salida de un S.O. Linux.

Un open puede añadir una entrada en la tabla de i-nodos, siempre añade una nueva entrada en la TFA y un nuevo canal en la tabla de canales

34.¿Qué significa “redireccionar la salida estándar”?

Significa modificar el fichero que tenemos asociado con el canal 1 .

35.¿Es correcta la siguiente secuencia de código? Indica porqué.

```
int fd[2];
pipe(fd);
dup2(fd[0],1);
write(1,"hola",4);
```

No es correcta porque redireccionamos el canal de lectura de la pipe al canal 1 y luego escribimos en el, pero el canal 1 es de lectura en este caso.

36. Explica brevemente en qué consiste la optimización de “caching”, ¿qué ventajas tiene, y qué relación tiene, si crees que tiene, con la “buffer cache” que podemos encontrar en sistemas UNIX.

La optimización de caching consiste en almacenar en memoria los datos que ya han sido accedidos (una copia) del dispositivo para aumentar la velocidad de acceso. La buffer cache es un caso concreto de caching.

37. En un sistema para gestionar el disco, ¿qué mejora aporta un sistema indexado multinivel respecto a uno con un único nivel?

Un sistema multinivel necesita menos espacio para almacenar los índices que un sistema con un único nivel. Además, requiere de menos accesos a disco para encontrar el bloque de índices/datos que está buscando.

38. ¿Qué diferencia, en cuanto eficiencia, hay entre estos dos códigos?

El segundo es más eficiente ya que el lseek no hace ningún acceso a disco, simplemente modifica el puntero de l/e que hay en la TFA.

|                                                                                                                                                            |                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| <pre>int tamanyo=0,fd; char c; fd=open("A",O_RDONLY); while (read(fd,&amp;c,sizeof(char))&gt;0)     tamanyo++; printf("El tamanyo es %d\n",tamanyo);</pre> | <pre>int tamanyo=0,fd; char c; fd=open("A",O_RDONLY); tamanyo=lseek(fd,0,SEEK_END); printf("El tamanyo es %d\n",tamanyo);</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|

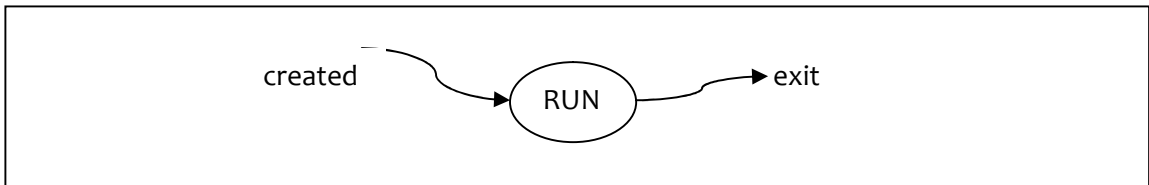
39. ¿Qué diferencia hay entre un “hard link” y un “soft link”?

Un hard-link es tener un enlace directo al i-nodo. No es un tipo de fichero sino simplemente un nombre de fichero vinculado al mismo i-nodo. Un softlink es un tipo de fichero especial cuyo contenido es un path a otro fichero.

## Ejercicios diversos (principalmente orientados a Procesos)

### Ejercicio (3 puntos)

Estamos analizando el comportamiento de los procesos en un SO que estamos implantando. En el diseño inicial de nuestro SO disponemos de un sistema uniprogramado (sólo 1 proceso en el sistema en un instante), sin memoria virtual y que posee periféricos muy optimizados donde tan pronto se requiere realizar una comunicación la transferencia del dato es inmediata. Es decir, tarda 0 ciclos en completar la transferencia. Partiendo del diagrama de estados siguiente, suficiente para esta configuración, complétalo si vamos añadiendo las siguientes funcionalidades al sistema operativo:



- a) Tras una mejora en el SO ahora permite multiprogramación, dibuja el nuevo ciclo de vida y comenta brevemente los cambios que has hecho.

*Habría que añadir, como mínimo, el estado de READY*

- b) Tras otra mejora, disponemos de un sistema que ofrece comunicación mediante signals. Dibuja el nuevo ciclo de vida y comenta brevemente los cambios que has hecho.

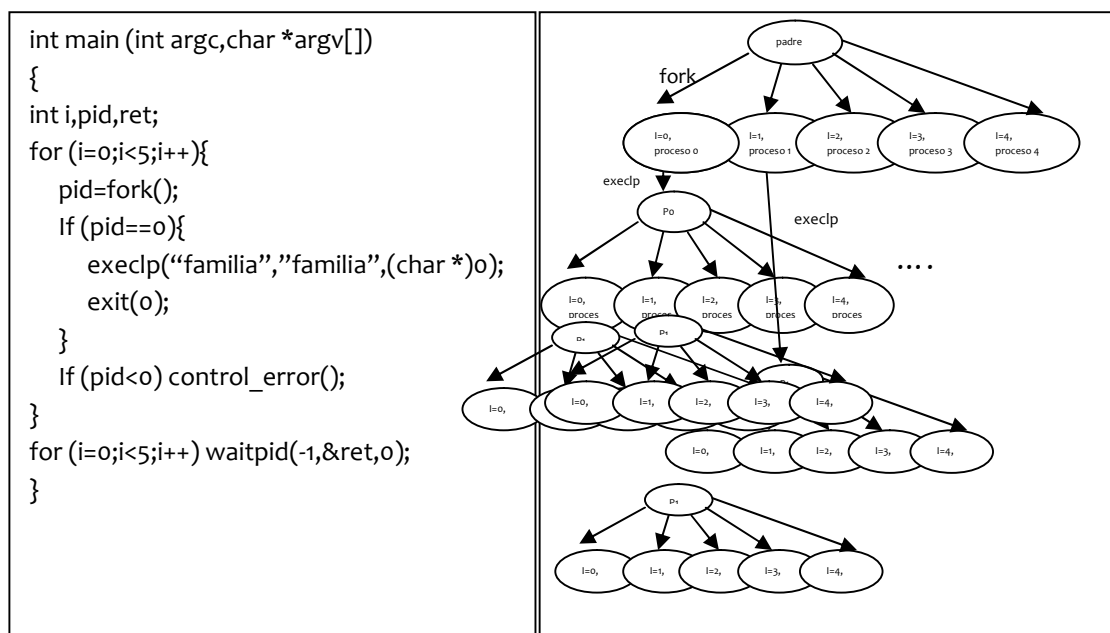
*Al tener signals, los procesos pueden pararse, habría que añadir el estado de STOPPED*

- c) Tras una última mejora, ahora disponemos de un planificador a medio plazo. Dibuja el nuevo ciclo de vida y comenta brevemente los cambios que has hecho

*El planificador a medio plazo puede poner los procesos en estado SWAPPED-OUT, así que hay que añadir este nuevo estado, diferenciando los estados READY-SWAPPED-OUT y STOPPED-SWAPPED-OUT.*

## Ejercicio (3 puntos)

El siguiente código pertenece al programa “familia”. Sabemos que el programa en memoria ocupa 5KB. Nuestro sistema informático dispone de 1MB de memoria y su gestión de memoria no implementa memoria virtual ni carga bajo demanda. El sistema tiene un máximo global de 100 procesos. Justifica todas las respuestas.



- a) Dibuja, en el cuadro de al lado del código, el árbol de procesos que se generaría al ejecutar el programa

*Cada proceso creado ejecuta a su vez el código del programa familia, por lo que el nuevo proceso crea a su vez otros 5 y así sucesivamente*

- b) Describe en qué momento finalizará la ejecución de dicho código asumiendo que nuestro SO **no tiene** la optimización de Copy-on-Write.

*Si el sistema tiene un límite de 100 procesos, que es menor que los procesos que caben en memoria, el programa finalizará cuando lleguemos a este límite y empiecen a fallar los procesos creados debido a que no pueden crearse más procesos.*

- c) Describe en qué momento finalizará la ejecución de dicho código, suponiendo que nuestro SO **tiene** la optimización de Copy-on-Write.

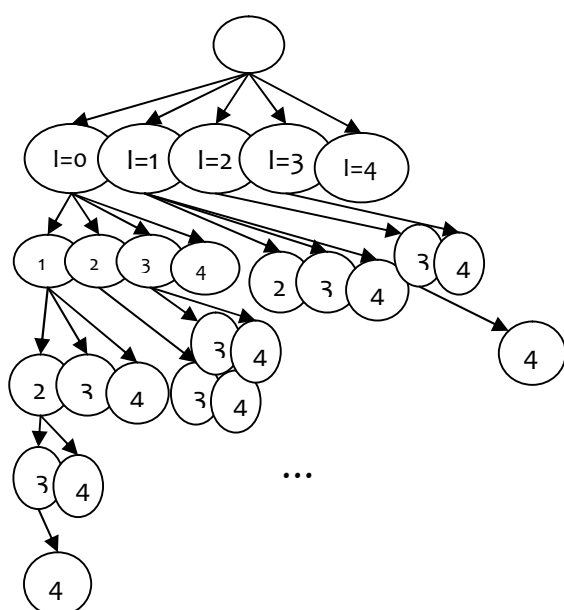
Aunque tenga COW en este caso no afecta porque el máximo lo marca el límite de procesos, no la memoria

- d) ¿Cómo se comportaría el programa si en lugar de crear procesos creáramos threads?

Si creáramos threads, el programa se ejecutaría indefinidamente ya que el primer thread ejecutaría su propio código (ya que muta el código de todo el proceso) y siempre tendríamos un máximo de 1 proceso continuamente

## Ejercicio (3 puntos)

Dado el siguiente código, contesta a las siguientes preguntas. Justifica todas las respuestas:



```
void hacer_tarea(int i)
{
 ...
 exit(0);
}
int main (int argc, char *argv[])
{
 int i, pid, ret;
 for (i=0; i<5; i++){
 pid=fork();
 If (pid==0) hacer_tarea(i);
 If (pid<0) control_error();
 }
 for (i=0; i<5; i++) waitpid(-1, &ret, 0);
}
```

- a) ¿Cuántas veces se ejecutará la función hacer\_tarea()?

5 veces

- b) Si queremos que el código de los procesos se ejecute de forma concurrente, ¿hay que hacer alguna modificación? Indica cual.

Ninguna. El código ya es concurrente

- c) Suponiendo que no hay ningún error, ¿Cuántos procesos se crearán?

5 procesos

- d) Dibuja, en el cuadro de al lado del código, el árbol de procesos que se crearía si quitáramos el exit de la función hacer\_tarea y que valor recibiría cada instancia de la función como parámetro (la i)

Al quitar el exit, cuando termina la función hacer\_tarea el nuevo proceso continúa iterando, como también hereda el valor de la i, el proceso continúa creando nuevos procesos hasta que i=4, los nuevos procesos también iteran.

Ejercicio (3 puntos)

Dado el siguiente código de usuario para ejecutar sobre Linux. Justifica todas las respuestas

- a) Dibuja, en el cuadro de al lado del código, el árbol de procesos que se genera. Indica en el dibujo qué proceso(s) recibirán el signal SIGINT, qué proceso(s) enviarán el signal SIGINT , y qué proceso(s) ejecutará(n) el bucle de la línea 27 (while(1);)
- b) ¿Cuántos procesos ejecutarán las líneas 17 y 18?

Sólo 1, el padre

- c) ¿Y la rutina gestionSignal?

Sólo 1, el hijo 1

- d) ¿Qué mensajes veremos en la consola y cuántas veces aparecerán?

Los 2 mensajes ("Proceso padre" y "Final de la ejecución") y los veremos 2 veces cada uno

```
1: int pidh[2];
2: void gestionSignal(int signum){
3: int valor;
4: kill(pidh[1],SIGKILL);
5: waitpid(pidh[1],&valor,0);
6: exit(1);
7: }
8: main(){
9: int i, valor, padre;
10: int proclnicial = getpid();
11: signal(SIGINT, gestionSignal);
12: pidh[0] = fork();
13: pidh[1] = fork();
14: if (pidh[1] > 0){
15: printf("Proceso padre\n");
16: waitpid(pidh[1],&valor,0);
17: kill (pidh[0],SIGINT);
18: waitpid(pidh[0],&valor,0);
19: } else {
20: /* la llamada a sistema getppid
21: retorna el pid del padre del
22: proceso que la ejecuta
23: */
24: padre = getppid()
25: if (padre != proclnicial){
26: while(1);
27: }
28: }
29: printf("Final de la ejecución\n");
30: }
31: }
32: }
33: }
```

```
graph TD
 padre([padre]) -- Fork1 --> Hijo1([Hijo1])
 padre -- Fork2 --> Hijo3([Hijo3])
 Hijo1 -- Fork2 --> Hijo2([Hijo2])
```

Hijo2 se queda en el while(1)

Hijo 3 imprime la línea 30 y termina

El padre se desbloquea del waitpid (l 16) y envía el signal (l 17)

El hijo 1 ejecuta la función gestionSignal , envía SIGKILL a hijo2 y espera en el waitpid

El hijo 2 termina

El hijo 1 sale del waitpid y termina

El padre sale del waitpid y escribe el mensaje de la línea 30

El padre y el hijo 1 habían escrito el mensaje de la l15

# Ejercicios diversos (principalmente orientados a E/S y SF)

## Ejercicio (3 puntos)

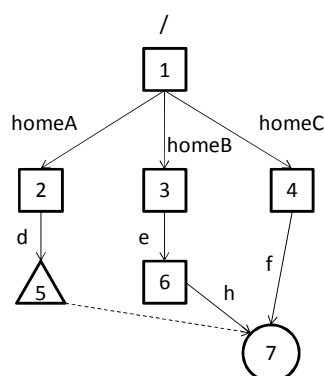
Tenemos los siguientes I-nodos y bloques de datos en un SF basado en I-nodos:

| Inodo   | Inodo | Inodo | Inodo | Inodo | Inodo | Inodo | Inodo |
|---------|-------|-------|-------|-------|-------|-------|-------|
|         | 1     | 2     | 3     | 4     | 5     | 6     | 7     |
| Tipo    | DIR   | DIR   | DIR   | DIR   | Link  | DIR   | Datos |
| #Refs   | 5     | 2     | 3     | 2     | 1     | 2     | 2     |
| Bloques | 1     | 2     | 3     | 4     | 5     | 6     | 7     |

| Bloque 1 |   | Bloque 2 |   | Bloque 3 |   | Bloque 4 |   | Bloque 5   |  | Bloque 6 |   | Bloque 7                            |  |
|----------|---|----------|---|----------|---|----------|---|------------|--|----------|---|-------------------------------------|--|
| .        | 1 | .        | 2 | .        | 3 | .        | 4 | /homeB/e/h |  | .        | 6 | Este es un<br>texto de<br>prueba... |  |
| ..       | 1 | ..       | 1 | ..       | 1 | ..       | 3 |            |  |          |   |                                     |  |
| homeA    | 2 | d        | 5 | e        | 6 | f        | 7 |            |  | h        | 7 |                                     |  |
| homeB    | 3 |          |   |          |   |          |   |            |  |          |   |                                     |  |
| homeC    | 4 |          |   |          |   |          |   |            |  |          |   |                                     |  |

a) (1 punto) Dibuja el grafo de directorios correspondiente al sistema de ficheros descrito en este ejercicio.

"d" es un softlink a "/homeB/e/h", "f" y "h" son hardlinks (los dos apuntan al mismo i-nodo 7)



b) (1 punto) Queremos implementar un código que copie la segunda mitad del contenido del fichero "homeA/d" en un nuevo fichero llamado "/homeB/hNuevo". Escribe el código necesario para poder realizar dicha copia, MINIMIZANDO el número de llamadas a sistema que acceden 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).



```
fd=open("/homeA/d",O_RDONLY);
fd_nuevo=open("/homeB/hNuevo",O_WRONLY|O_CREAT|O_TRUNC,0666);
t=lseek(fd,0,SEEK_END);
lseek(fd,t/2,SEEK_SET);
leidos=read(fd,buf,t/2);
write(fd_nuevo,buf,leidos);
close(fd);
close(fd_nuevo);
```

- c) (1 punto) Indica los accesos a disco que se realizan para cada una de las llamadas al sistema de la siguiente sección de código (es decir, las siguientes instrucciones se corresponden a instrucciones consecutivas dentro de un código determinado). NOTA: No disponemos de buffer cache, el superbloque está cargado en memoria, cada I-nodo tiene el tamaño de un bloque y la variable "buf" está declarada previamente como "char buf[256]".

```
fd = open("/homeA/d", O_RDWR);
```

*i-nodo raíz (1) + Bloque 1 + i-nodo 2 + Bloque 2 + i-nodo 5 + bloque 5 + i-nodo 1 + bloque 1 + i-nodo 3 + bloque 3 + i-nodo 6 + bloque 6 + i-nodo 7 = 13 accesos*

```
lseek(fd, 0, SEEK_END);
```

*EL lseek no hace accesos a disco*

```
read(fd, buf, 100);
```

*Como el lseek anterior nos ha posicionado al final del fichero, este read no hace accesos a disco*

```
write(fd, buf, 100);
```

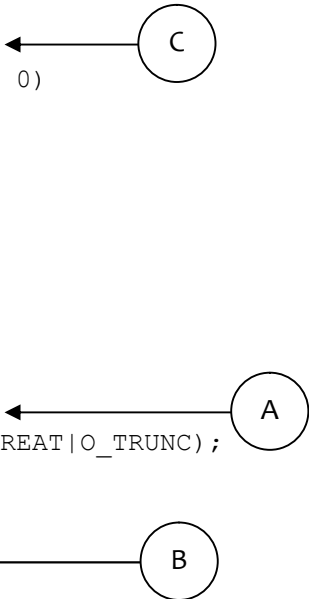
*Haremos 1 acceso a disco para escribir la variable buf*

## Ejercicio (3,5 puntos)

A partir del siguiente código contesta a las preguntas que se te formulan en los apartados.

```
1: void procesar()
2: {
3: char c;
4:
5: while (read(0, &c, sizeof(char)) > 0)
6: write(1, &c, sizeof(char));
7:
8: exit(0);
9:}

10: int main(int argc, char *argv[])
11: {
12: int fd_ent, fd_sal;
13: int pids[3];
14:
15: fd_ent=open("in.txt", O_RDONLY);
16: fd_sal=open("out.txt", O_WRONLY|O_CREAT|O_TRUNC);
17: for (i=0; i<2; i++){
18: pids[i]=fork();
19: if (pids[i] == 0){
20: dup2(fd_ent, 0);
21: dup2(fd_sal, 1);
22: procesar();
23: }
24: }
25:
26: close(fd_ent);
27: while(waitpid(-1, NULL, 0)>0);
28:}
```



- a) (1 punto) Indica todas las estructuras de datos de memoria y de disco que se acceden y/o actualizan a lo largo de la ejecución de las llamadas al sistema que tenemos en los puntos “A” y “B”.

A: El open accede al bloque de datos del directorio y al i-nodo de in.txt. Añade el i-nodo en la tabla de i-nodos, crea una nueva entrada en la TFA y añade un nuevo canal en la tabla de canales.

B: Como cierra el canal 0, actualiza la TFA (y potencialmente la tabla de i-nodos) y modifica la tabla de canales al duplicar fd\_ent y actualiza las referencias de la TFA que apuntaba fd\_ent.

b) (0,5 puntos) Indica cuál sería el resultado final de la ejecución de este código si ...

... la línea 15 pasa a estar entre las líneas 19 y 20:

Todos los procesos harían su propio open, así que el fichero se leería N veces aunque las escrituras estarían mezcladas

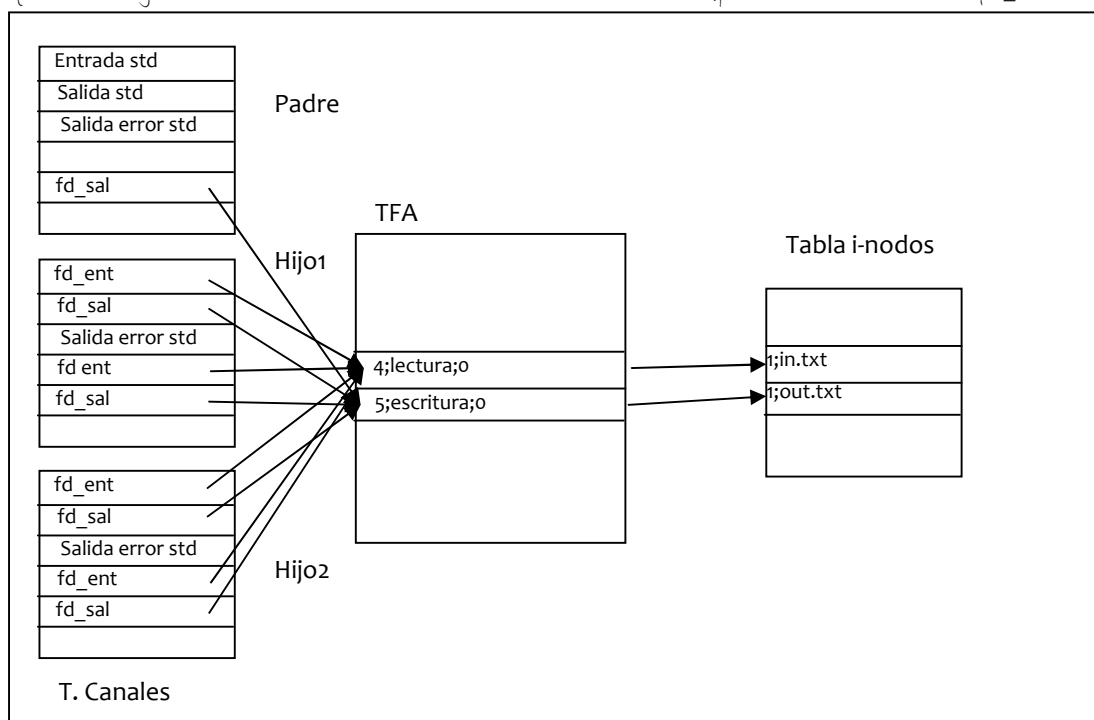
... la línea 16 pasa a estar entre las líneas 19 y 20:

Cada proceso abriría el fichero para escritura. Al tener todos el flags `O_TRUNC` el fichero resultante sería raro. Como no comparten el puntero de l/e, los procesos machacan lo que han escrito los otros procesos

c) (1 punto) Dibuja el estado de las TC, TFA y T-Inodos en el instante indicado en el punto "C".

NOTA: asumimos que todos los procesos hijo ya se han creado y se encuentran al mismo tiempo en ese punto (justo antes del read).

El open se ha realizado antes de hacer los forks, por lo que los canales de los procesos apuntan a las mismas entradas de la TFA (sólo hay dos entradas en la TFA y en la tabla de i-nodos). Además hay que tener en cuenta que el proceso padre hace close de `fd_ent` posteriormente a hacer el fork y que los hijos han redirigido su E/S standard (pero sin cerrar ni `fd_ent` ni `fd_sal`).



d) (1 puntos) Debido a una futura modificación del código, nos interesa que cada uno de los procesos que hemos creado mediante el fork conozcan los PIDs de todos sus otros "procesos hermanos". Indica si se puede obtener esta información (y en caso afirmativo describe brevemente cómo lo harías) mediante la utilización de:

i. Signals:

Con SOLO signals no se puede hacer ya que no se pueden enviar datos, sólo eventos

ii. Pipes:

Con pipes sí se podría hacer. Se podrían conectar los procesos haciendo un círculo con pipes y que se fueran pasando los pids hasta que se detecte que los tienen todos pq se repiten.

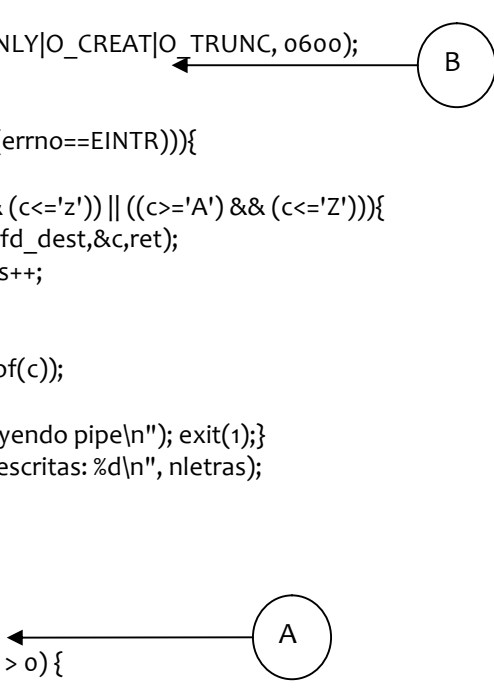
## Ejercicio (4 puntos)

Tenemos el siguiente fichero con un programa para ejecutar en Linux:

```
void trat_alarma(int signum){
char buf[50];
 signal(SIGALRM, trat_alarma);
 strcpy(buf, "Han pasado 10 segundos más\n");
 write(2,buf,strlen(buf));
 alarm(10);
}

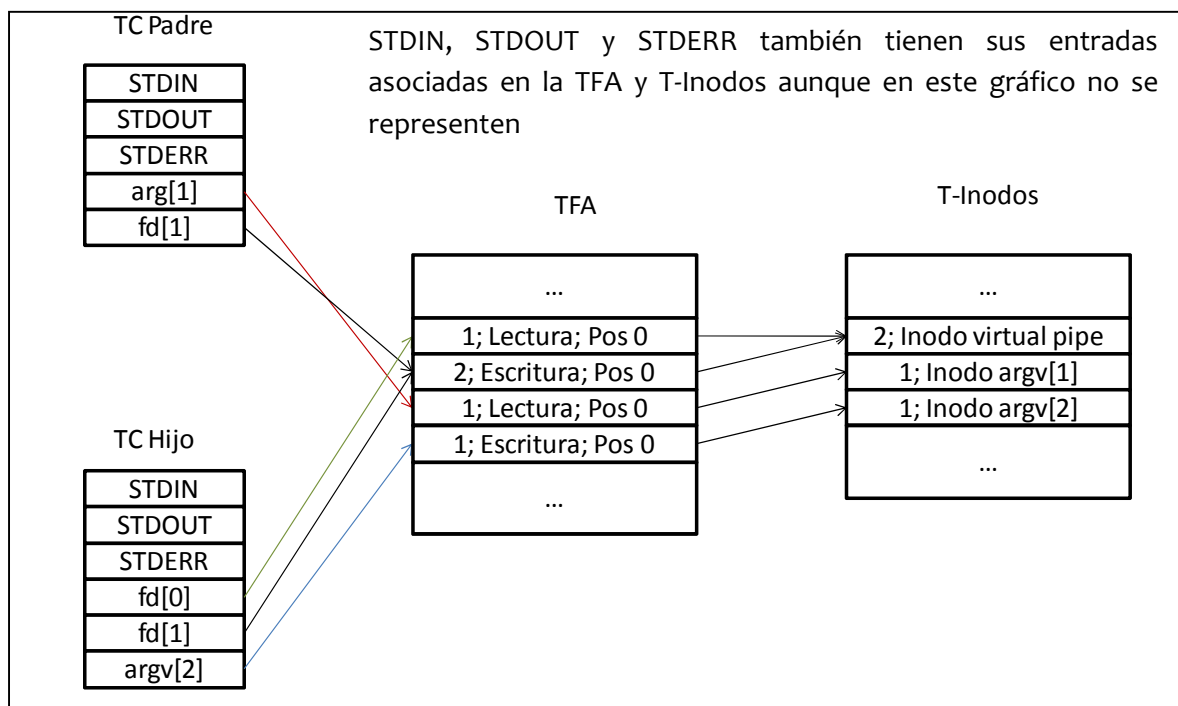
main(int argc, char *argv[]){
int fd[2], fd_dest,fd_src;
int ret, pidh, nletras=0;
char c, buf[50];
 siginterrupt(SIGALRM,1);
 signal(SIGALRM, trat_alarma);
 pipe(fd);
 pidh = fork();
 if (pidh == -1) { perror("Error creando hijo"); exit(1);}

 If (pidh == 0) {
 fd_dest=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC, 0600);
 alarm(10);
 ret=read(fd[0],&c,sizeof(c));
 while ((ret>0) || ((ret == -1) && (errno==EINTR))){
 if (ret > 0){
 if (((c>='a') && (c<='z')) || ((c>='A') && (c<='Z'))){
 write(fd_dest,&c,ret);
 nletras++;
 }
 }
 ret=read(fd[0],&c,sizeof(c));
 }
 if (ret<0) { perror("Error hijo leyendo pipe\n"); exit(1);}
 sprintf(buf, "Numero de letras escritas: %d\n", nletras);
 write(2,buf,strlen(buf));
 exit(0);
 }
 close(fd[0]);
 fd_src=open(argv[1],O_RDONLY);
 while ((ret = read(fd_src, &c, sizeof(c))) > 0) {
 write(fd[1], &c, sizeof(c));
 }
 close(fd_src); close(fd[1]);
 waitpid(pidh, NULL, 0);
 strcpy(buf, "Padre acaba la ejecucion\n");
 write(1,buf,strlen(buf));
}
```



Podéis suponer que las llamadas a sistema de este código se ejecutan sin dar error y que el usuario lo ejecuta sin redireccionar ningún canal estándar y pasándole unos parámetros correctos.

- a) Representa el estado de las tablas de canales y de la tabla de ficheros abiertos, suponiendo que el proceso padre se encuentra ejecutando la línea A (justo después de `open(argv[1]...)` ) y el proceso hijo está en la línea B (justo después de `open(argv[2]...)` ).



- b) Este programa contiene un error y si lo ejecutáramos no acabaría ninguno de los dos procesos ¿Por qué? ¿Cómo lo arreglarías?

No termina porque el proceso hijo no cierra el canal de escritura de la pipe y por lo tanto él mismo provoca que no termine. Como el no termina, el padre tampoco ya que le estará esperando en el `waitpid`. Habría que cerrar el canal antes del bucle.

- c) Queremos que el código de tratamiento que ejecuta el hijo forme parte de un programa separado que el hijo cargue mediante la llamada `execlp`. Y proponemos los siguientes códigos. El código inicial se mantiene igual excepto las líneas marcadas en **negrita**. Y el código del nuevo programa contiene las líneas que en la versión anterior formaban parte del código ejecutado por el proceso hijo.

```
/* fichero padre.c */
```

```
void trat_alarma(int signum){
char buf[50];
 signal(SIGALRM, trat_alarma);
 strcpy(buf, "Han pasado 10
segundos más\n");
 write(2,buf,strlen(buf));
 alarm(10);
}

main(int argc, char *argv[]){
int fd[2], fd_src;
int ret, pidh;
char c, buf[50];
 siginterrupt(SIGALRM,1);
 signal(SIGALRM, trat_alarma);
 pipe(fd);
 pidh = fork();
 if (pidh == -1) {
 perror("Error creando
hijo");
 exit(1);
 }
 if (pidh == 0) {
 execlp("./hijo", "hijo",
NULL);
 perror("Error en el
execlp"); exit(1);
 }
 close(fd[0]);
 fd_src=open(argv[1],O_RDONLY);
 while ((ret = read(fd_src, &c,
sizeof(c))) > 0) {
 write(fd[1], &c, sizeof(c));
 }
 close(fd_src); close(fd[1]);
 waitpid(pidh, NULL, 0);
 strcpy(buf, "Padre acaba la
ejecucion\n");
 write(1,buf,strlen(buf));
}
```

```
/*fichero hijo.c */
```

```
main() {
int fd_dest, ret, nletras = 0;
int fd[2];
char c, buf[50];

 fd_dest=open(argv[2],
O_WRONLY|O_CREAT|O_TRUNC, 0600);
 alarm(10);
 ret=read(fd[0],&c,sizeof(c));
 while ((ret>0) || ((ret==-1) &&
(errno==EINTR))){
 if (ret > 0){
 if (((c>='a') && (c<='z'))
|| ((c>='A') && (c<='Z'))){
 write(fd_dest,&c,ret);
 nletras++;
 }
 }
 ret=read(fd[0],&c,sizeof(c));
 }

 if (ret<0) {
 perror("Error hijo leyendo
pipe\n");
 exit(1);
 }

 sprintf(buf, "Numero de letras escritas:
%d\n", nletras);
 write(2,buf,strlen(buf));
}
```

¿Cuál será el resultado de ejecutar este código? ¿Tendrá la misma funcionalidad que el código original o es necesario modificar alguna cosa más? En caso de que sea necesario introducir algún cambio más describe claramente qué cambio sería, indicando si afecta al código del fichero padre.c o hijo.c

-La función `trat_alarma` tiene que estar en el código del hijo porque sino, al hacer el `execlp`, la rutina asignada al `signal` de alarma se pone por defecto, y al llegar la primera alarma el proceso terminaría. Además hay que mover las líneas de reprogramación del `signal SIGALRM` al código del hijo.

-También hay que modificar la llamada `execlp` para pasar al hijo el parámetro que necesita.

-El proceso hijo utiliza la variable `fd[0]`, pero, al haber mutado, ya no puede utilizar esa variable. En este caso, el código de `hijo.c` debería utilizar el canal `0` y él mismo, antes de mutar, debería redirigirse el canal de lectura de la `pípe` al canal `0`.

## Ejercicio (2 puntos)

Queremos escribir un programa que duplique el contenido de un fichero. Por ejemplo, si el contenido de `/users/home/f1` es 'abcd' después de ejecutar nuestro programa su contenido tiene que ser 'abcdabcd'. Proponemos el siguiente código:

```
main(int argc, char *argv[]){
 int fd1, fd2;
 fd1 = open(argv[1], O_RDWR);
 fd2 = dup (fd1);
 lseek(fd2, 0, SEEK_END);
 while ((ret = read(fd1, &c, sizeof(c))) > 0){
 write(fd2, &c, ret);
 }
 close(fd1);
 close(fd2);
}
```

- a) Después de ejecutarlo comprobamos que el resultado no es el esperado. Suponiendo que le pasamos como parámetro `/users/home/f1` que inicialmente contiene 'abcd', ¿cómo se comportará este programa? ¿cómo afectará al fichero `/users/home/f1`? ¿por qué?

Este programa lee y escribe utilizando el mismo puntero de l/e. Aunque se haya duplicado el canal, solo se ha hecho 1 `open`, por lo que el `lseek` sitúa el puntero al final y el primer `read` devolverá un cero (fin de fichero) y no se escribirá nada. Por lo tanto, el fichero no se modificará.

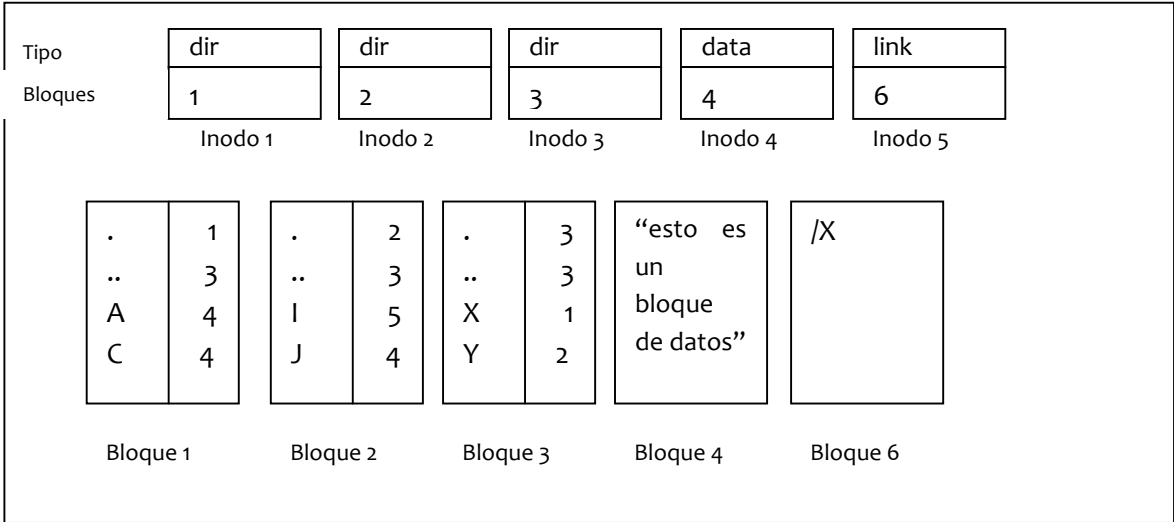
b) Indica qué accesos a disco generarían las 4 primeras sentencias de este código (open, dup, lseek y primer read del bucle) si le pasamos como parámetro el fichero `/users/home/f1` asumiendo que:

- Acabamos de iniciar la máquina y ningún otro proceso está usando ningún fichero
- El fichero `/users/home/f1` existe y su contenido es ‘abcd’
- No tenemos buffer cache
- Los i-nodos ocupan 1 bloque
- Los directorios que leemos también ocupan 1 bloque
- El superbloque está en memoria

Open: lee i-nodo / + bloque datos / + inodo users + bloque datos users + i-nodo home + bloque datos home + i-nodo f1 → 7 accesos  
Dup: no genera accesos a disco  
Lseek: no genera accesos a disco  
Read: no genera accesos a disco porque estamos al final del fichero

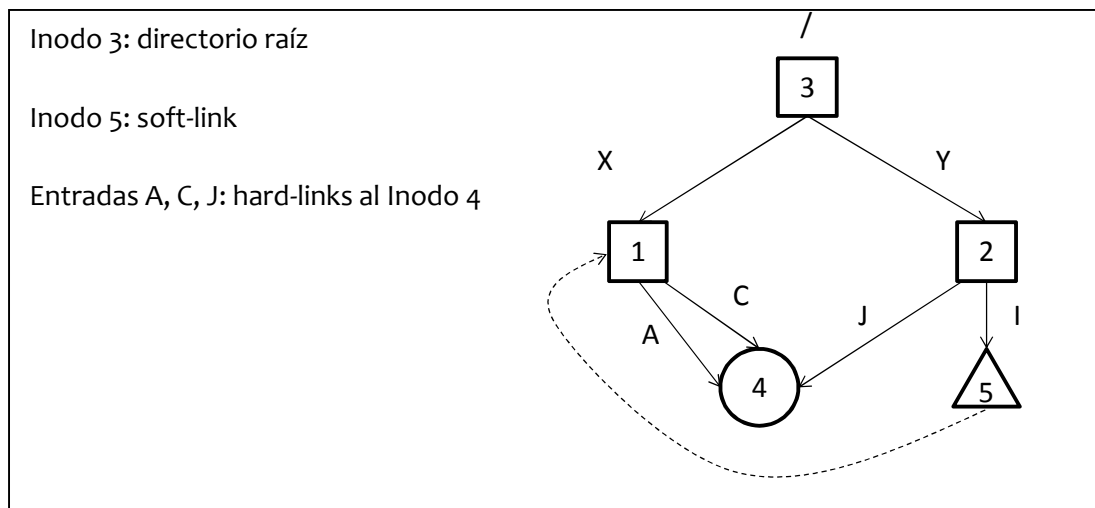
### Ejercicio (3 puntos)

Suponed un sistema de ficheros (basado en i-nodos) que tiene los siguientes i-nodos y bloques de datos:



f) Dibuja el grafo de directorios que se puede deducir de este esquema de i-nodos y bloques de datos





- ¿Cuál crees que es el i-nodo raíz? (márcalo en el dibujo)
- Indica que ficheros son soft-link y que ficheros son hard-link. En el caso de los soft-links indica mediante flechas los ficheros a los que apunta (márcalo en el dibujo)
- Indica el número de accesos a disco que se generaría si ejecutamos la llamada a sistema `open("/Y/I/A", O_RDONLY)`. Asume que el superbloque está en memoria. Indícalo en el caso que el sistema tenga buffer cache y en el caso que no tenga (asume que los i-nodos ocupan 1 bloque de disco)

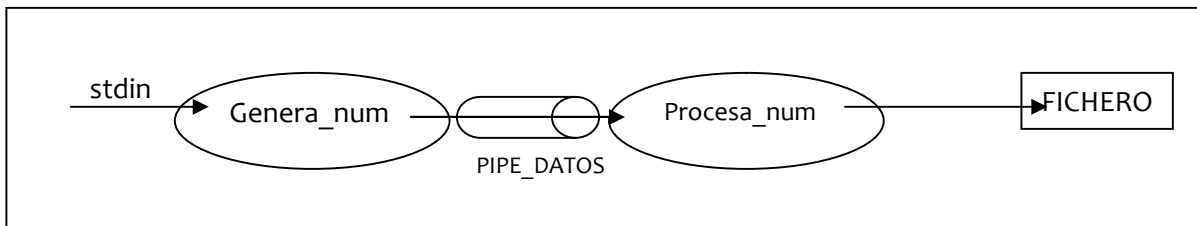
Sin buffer cache: i-nodo 3 (raíz) + bloque 3 + i-nodo 2 + bloque 2 + i-nodo 5 + bloque 6 + i-nodo 3 + bloque 3 + i-nodo 1 + bloque 1 + i-nodo 4 → 11 accesos  
 Con buffer cache: 9 accesos

- Indica además, en el caso anterior, que i-nodo se guardaría en la tabla de i-nodos en memoria

i-nodo 4

## Ejercicio (3 puntos)

Queremos hacer un programa que estará dividido en dos partes, una que genera números y otra que realiza un cálculo con esos números y guarda el resultado en un fichero que se le pasa como parámetro. El esquema es el siguiente, donde el programa `Genera_num` lee números de su entrada `std`, aplica un cálculo a cada número y el resultado lo escribe en la pipe con nombre `"PIPE_DATOS"` (que ya está creada). El programa `Procesa_num` lee los números de la pipe, hace un segundo cálculo también a cada número, y escribe el resultado en el fichero que recibe como parámetro.



Sin embargo, detectamos que los números se generan más rápido de los que se procesan, por lo que decidimos hacerlo en paralelo, y le pasamos un segundo parámetro al programa `Procesa_num` con el número de procesos que queremos que se creen. Nos ofrecen esta solución, pero nos encontramos que no es correcta, vemos que el fichero de salida no tiene todos los números que debería (hemos quitado el control de error, includes, etc para que ocupe menos):

```

int fd_ent, fd_sal;
int nums, nuevo_num;
int i, num_procs, pids[MAX_PROCS];
int calcula(int n)
{
 int res;
 // Calculo sobre n
 return res;
}
void hijo(void)
{
 //INICIO LECTURAS
 while(read(0, &nums, sizeof(int)) > 0) {
 nuevo_num = calcula(nums);
 write(1, &nuevo_num, sizeof(nuevo_num));
 }
 exit(0);
}
int main(int argc, char *argv[])
{
 num_procs = min(atoi(argv[2]), MAX_PROCS);
 for (i = 0; i < num_procs; i++) {
 pids[i] = fork();
 if (pids[i] == 0) {
 fd_ent = open("PIPE_DATOS", O_RDONLY);
 dup2(fd_ent, 0); close(fd_ent);
 fd_sal = open(argv[1], ...); // Faltan los flags
 dup2(fd_sal, 1); close(fd_sal);
 hijo();
 }
 }
 while(waitpid(-1, NULL, 0) > 0);
}

```

- a) Indica qué valores tendríamos que poner en los flags que le pasamos al `open` para crear un nuevo fichero de salida en cada ejecución del programa.

`O_WRONLY|O_CREAT|O_TRUNC, 0666`

- Indicamos que el vamos a escribir en un nuevo fichero a crear en caso que no existiera (con permisos 666). Si ya existía, el contenido se trunca (tamaño de fichero 0).

- b) Dibuja cómo quedarían las estructuras de gestión de E/S en memoria (tabla canales, tabla ficheros abiertos y tabla de i-nodos) en la línea de código marcada como “INICIO LECTURAS” si ejecutamos el programa de la siguiente forma “Procesa\_num NUMS\_EJ1 2” (asumiendo que en ese punto los procesos ya están creados.)

A falta del dibujo: el punto principal es que los open's se han hecho dentro del código de cada hijo, por lo que cada uno tiene entradas diferentes en la TFA y no comparten el puntero de l/e. Aunque cada open implica una nueva entrada de la TFA, todas ellas apuntan a dos únicos ficheros (pipe\_datos y el nuevo fichero de escritura). Por tanto, hay sólo dos entradas en la tabla de i-nodos. Por último, los procesos tienen los canales de ambos ficheros en el 0 y 1 porque los han redirigido previamente.

- c) Una vez analizadas las tablas, vemos que con una pequeña modificación se puede arreglar el código. Indica claramente que líneas de código habría que mover o modificar para que el programa se comporte como la versión secuencial (el orden de los números en el fichero de salida no es importante).

Hay que mover el open del fichero de salida fuera del bucle de creación de procesos para que todos compartan el mismo puntero de l/e

- d) Una vez arreglado, queremos hacer una extensión para permitir que la cantidad de procesos que colaboran sea menor al cabo de un tiempo ya que hemos detectado que después de un tiempo la velocidad de generar datos disminuye considerablemente y tenemos a los procesos del programa Procesa\_num bloqueados. ¿En qué línea de código estarán bloqueados los procesos hijos?

En el read del bucle

- e) Se nos ocurre una solución basada en signals, el programa Procesa\_num cada vez que reciba un SIGUSR1 ejecutará la siguiente función para terminar con un proceso hijo:
- a) ¿Cómo (línea de código) y dónde tenemos que reprogramar el SIGUSR1?

```
void tratar_SIGUSR1(int s)
{
 char b[128];
 sprintf(b, "Termino al proceso %d\n", pids[num_procs]);
 write(1, b, strlen(b));
 kill(pids[num_procs], SIGUSR1);
 num_procs--;
 signal(SIGUSR1, tratar_SIGUSR1);
}
```

El padre tiene que reprogramarlo después de haber creado a los hijos para que éstos tengan la programación por defecto. La línea de código es: `signal(tratar_SIGUSR1, SIGUSR1)`