# Soluciones problemas T1,T2 y T3

*Sistemas Operativos*
*Grau en Enginyeria Informàtica*

Soluciones a los problemas de la sesión PROB_T2

**Profesores SO-Departamento AC**
**13/02/2016**

# Ejercicio 1. (T1,T2,T3)

1. (*T1*) Un usuario busca en google el código de la librería de sistema de Linux (para la misma arquitectura en que trabaja) que provoca la llamada a sistema write, lo copia en un programa suyo, lo compila y lo ejecuta pasándole unos parámetros correctos. ¿Qué crees que pasará? (se ejecutará sin problemas o le dará un error de operación no permitida). Justifícalo.

No debería dar ningún problema si los parámetros son correctos ya que generar la llamada a sistema se realiza desde modo usuario, no se necesitan privilegios

2. (*T2*) En un sistema que aplica una política de planificación tipo *Round Robin*, ¿qué es el Quantum? ¿Se puede ejecutar sin problemas un proceso que tiene ráfagas de CPU que duran, en media, la mitad del quantum?¿ qué pasará al acabar su ráfaga si no ha acabado todavía el quantum?¿Continuará en la cpu hasta el final del quantum o abandonará la cpu?¿cuando vuelva a recibir la cpu… recibirá un quantum completo o sólo lo que le quedaba?

El quantun es la duración máxima de una ráfaga de cpu que se define en políticas de tiempo compartido como Round robin. No hay ningún problema en ejecutar un proceso, simplemente cuando se le acabe la ráfaga de cpu dejará la cpu.El proceso recibirá quantum completo al volver a ejecutarse.

3. (*T3*) Explica que mejoras supone la utilización de librerías compartidas.

Las librerías dinámicas ahorran mucho espacio en disco ya que no se incluye la librería con el binario y también en memoria física ya que no se carga N veces el código sino que se comparte.

4. (*T3*) Enumera y describe brevemente los pasos que debe hacer el SO para cargar un ejecutable desde disco a memoria para implementar la mutación de un proceso en el caso que después de la mutación el proceso continúa ejecutándose.

- Leer e interpretar el binario
- Preparar el esquema en memoria física
- Inicializar las estructuras de datos del proceso
- Actualizar la MMU
- Leer el fichero de disco y copiarlo en memoria
- Hacer los cambios de los registros necesarios para pasar a ejecutar el nuevo espacio de direcciones

5. (*T3*) Tenemos un sistema que ofrece memoria virtual, en el cual nos dicen que se está produciendo el problema de thrashing. Explica brevemente en qué consiste este problema e indica y justifica) que métrica veríamos claramente aumentar en esta situación si estuviéramos analizando los procesos: el tiempo de usuario o el tiempo de sistema.

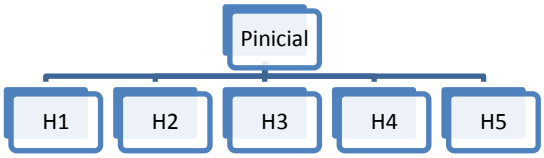
El thrashing es la situación en la que el sistema invierte más tiempo haciendo intercambio de páginas que en ejecutar código de los procesos. Veríamos aumentar claramente el tiempo de sistema ya que el intercambio de páginas lo hace el kernel.

## Ejercicio 2. (T2, T3)

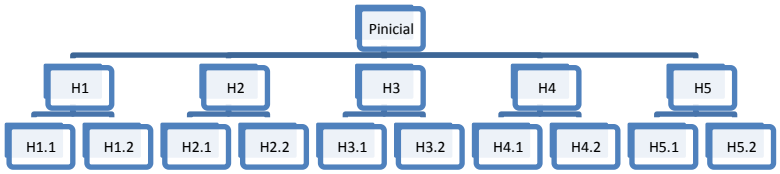
La Figura 1 contiene el código del programa *Mtarea* (se omite el control de errores para simplificar y podemos asumir que ninguna llamada a sistema devuelve error).

1. *(T2)* Dibuja la jerarquía de procesos que se genera si ejecutamos este programa de las dos formas siguientes (asigna identificadores a los procesos para poder referirte después a ellos):

   a) ./Mtarea 5 0




   b) ./Mtarea 5 1




En este caso, también se acepta suponer que se crean 3 procesos en el segundo nivel, pero siempre con la misma estructura

2. Indica, para cada una de las dos ejecuciones del apartado anterior:

   a) *(T2)* ¿Será suficiente el tamaño del vector de pids que hemos reservado para gestionar los procesos hijos del proceso inicial? (Justifícalo y en caso negativo indica que habría que hacer).

a) EJECUCIÓN ./Mtarea 5 0

Es suficiente porque solo controlamos los hijos del proceso inicial, los otros no los detectaremos ya que no son sus hijos. El sbrk reserva espacio para N enteros (donde N es el número de hijos)➔procesosxsizeof(int)

b) EJECUCIÓN ./Mtarea 5 1

b) (T2) Qué procesos ejecutarán las líneas 36+37

- EJECUCIÓN ./Mtarea 5 0

El proceso inicial ya que todos los demás terminan en el exit de la línea 19

- EJECUCIÓN ./Mtarea 5 1

El proceso inicial ya que todos los demás terminan en el exit de la línea 19

c) *(T3)* Qué procesos ejecutarán la función realizatarea

- EJECUCIÓN ./Mtarea 5 0

Todos menos el proceso inicial

- EJECUCIÓN ./Mtarea 5 1

Todos menos el proceso inicial

3. *(T3)* Sabemos que una vez cargado en memoria, el proceso inicial de este programa, tal y como está ahora, ocupa: 2Kb de código, 4bytes de datos, 4kb de pila y el tamaño del heap depende de argv[1] (asumid que como mucho será 4kb). Si ejecutamos este programa en un sistema Linux con una gestión de memoria basada en paginación, sabiendo que una página son 4kb, que las páginas no se comparten entre diferentes regiones y que ofrece la optimización COW a nivel de página. Sin tener en cuenta en las posibles librerías compartidas, CALCULA (desglosa la respuesta en función de cada región de memoria):

a) El espacio lógico que ocupará cada instancia del programa Mtarea (número de páginas)

Espacio lógico: Codigo=(2kb→4kb, 1 pagina), Datos= (4bytes→4kb, 1pagina), pila=1página, heap=tamaño/4kb(máximo 1) ➔ 4páginas

Nota: No se comparten páginas entre regiones, por lo tanto, cada región ocupa N páginas

b) El espacio físico que necesitaremos para ejecutar las dos ejecuciones descritas en el apartado 1

- EJECUCIÓN ./Mtarea 5 0

- Código será compartido por todos los procesos → 1 página
- Datos será compartido ya que el puntero no cambia → 1 página
- Pila será privado x proceso
- Heap será privado x proceso (se modifica el contenido de pids, línea 31)
- Total=(6 procesos) →codigo=1+datos=1+pilas=6+heaps=6=14 páginas

- EJECUCIÓN ./Mtarea 5 0

- Código compartido (igual que antes)
- Datos compartido ya que el puntero no cambia (igual que antes)
- Pila privado x proceso
- Heap privado pero solo el primer nivel, el segundo será compartido con el primero
- Total= (16 procesos) = código=1+datos=1+pilas=16+heap=6(primer nivel)➔24 páginas

```
2. int *pids;
3. void usage()
4. {
5.   char  b[128];
6.   sprintf(b,"./Mtarea procesosnivel1(cuantos) procesosnivel2(0=no/1=si)\n");
7.   write(1,b,strlen(b));
8.   exit(0);
9. }
10. void realizatarea(int i){
11.   // Omitimos su código por simplicidad pero no hay ninguna llamada a sistema relevante
12.   // para el ejercicio
13. }
14.
15. void procesardatos(int i, int multiproceso)
16. {
17.    int it;
18.    if (multiproceso>0){ it=0;  while((fork()>0) && (it<2)) it++;}
19.    realizatarea(i);
20.    exit(1);
21. }
22. void main(int argc,char *argv[])
23. {
24.   int i,ret,procesos;
25.   char buff[128];
26.   if (argc!=3) usage();
27.   procesos=atoi(argv[1]);
28.   pids=sbrk(procesos*sizeof(int));
29.   for(i=0;i<procesos;i++){
30.      ret=fork();
31.     if (ret==0) procesardatos(i,atoi(argv[2]));
32.     pids[i]=ret;
33.   }
34.   while((ret=waitpid(-1,NULL,0))>0){
35.     for(i=0;i<procesos;i++){
36.       if (pids[i]==ret){
37.         sprintf(buff,"acaba el proceso num %d con pid %d \n" ,i,ret);
38.         write(1,buff,strlen(buff));
39.       }
40.     }
41.   }
42. }
```

*Figura 1 Código de Mtarea*

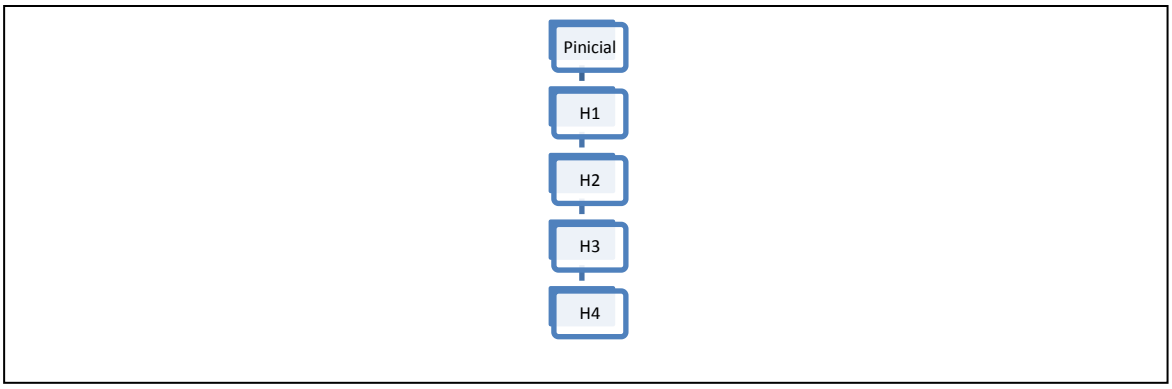
# Ejercicio 3. (T2)

La Figura 2 contiene el código del programa *ejercicio_exec* (se omite el control de errores para simplificar y podemos asumir que ninguna llamada a sistema provoca un error). Contesta a las siguientes preguntas **justificando todas tus respuestas**, suponiendo que los únicos SIGALRM que recibirán los procesos serán consecuencia del uso de la llamada a sistema *alarm* y que ejecutamos el programa de la siguiente manera: %./ejercicio_exec 4

```
1. int sigchld_recibido = 0;
2. int pid_h;
3. void trat_sigalrm(int signum) {
4. char buff[128];
5.    if (!sigchld_recibido) kill(pid_h, SIGKILL);
6.    strcpy(buff, “Timeout!”);
7.    write(1,buff,strlen(buff));
8.    exit(1);
9. }
10. void trat_sigchld(int signum) {
11.         sigchld_recibido = 1;
12. }
13. void main(int argc,char *argv[])
14. {
15.   int ret,n;
16.   int nhijos = 0;
17.   char buff[128];
18.   struct sigaction trat;
19.   trat.sa_flags = 0;
20.   sigempty(&trat.sa_mask);
21.   trat.sa_handler = trat_sigchld;
22.   sigaction(SIGCHLD, &trat, NULL);
23.
24.   n=atoi(argv[1]);
25.   if (n>0) {
26.         pid_h = fork();
27.         if (pid_h == 0){
28.           n--;
29.           trat.sa_flags = 0;
30.           sigempty(&trat.sa_mask);
31.           trat.sa_handler = trat_sigalrm;
32.           sigaction(SIGALRM, &trat, NULL);
33.           sprintf(buff, "%d", n);
34.           execlp("./ejercicio_exec", "ejercicio_exec", buff, (char *)0);
35.         }
36.         strcpy(buff,"Voy a trabajar \n");
37.         write(1,buff,strlen(buff));
38.         alarm (10);
39.         hago_algo_de_trabajo();/*no ejecuta nada relevante para el problema */
40.         alarm(0);
41.         while((ret=waitpid(-1,NULL,0))>0) {
42.             nhijos++;
43.         }
44.         sprintf(buff,"Fin de ejecución. Hijos esperados: %d\n",nhijos);
45.         write(1,buff,strlen(buff));
46.   } else {
47.         strcpy(buff,"Voy a trabajar \n");
48.         write(1,buff,strlen(buff));
49.         alarm(10);
50.         hago_algo_de_trabajo();/*no ejecuta nada relevante para el problema */
51.         alarm(0);
52.         strcpy(buff, “Fin de ejecución\n”);
53.         write(1,buff, strlen(buff));
54.   }
55. }
```

*Figura 2 Código del programa ejercicio_exec*

1. Dibuja la jerarquía de procesos que se crea y asigna a cada proceso un identificador para poder referirte a ellos en las siguientes preguntas.




2. Suponiendo que la ejecución de la función *hago_algo_de trabajo()* dura siempre **MENOS** de 10 segundos:
    a) Para cada proceso, indica qué mensajes mostrará en pantalla:

Si la rutina acaba antes de que pasen los 10 segundos, se desactiva el temporizador y todos los procesos acaban con normalidad. Pinicial empieza la ejecución con n==4, crea a H1 y a continuación pasa e ejecutar el código de la línea 28. H1 ejecuta la rama del if en la línea 23, decrementa n y muta para ejecutar una nueva instancia del mismo programa. Al empezar esta nueva instancia crea a H2 y pasa a ejecutar el código de la línea 28. El comportamiento de H2 y H3 es el mismo. H4 muta con n == 0 y por lo tanto al mutar entra por la rama del else en la línea 39. Los mensajes serían:

Pinicial, H1, H2 y H3 muestran: "Voy a trabajar" y "Fin de ejecución. Hijos esperados: 1"

H4 muestra: "Voy a trabajar" y "Fin de ejecución"

    b) Para cada proceso, indica qué signals recibirá:

Pinicial, H1, H2 y H3 reciben el SIGCHLD cuando sus hijos acaban.

    c) Supón que movemos las sentencias de la línea 29 a la línea 32 a la posición a la línea 23, ¿afectaría de alguna manera a las respuestas del apartado a y b? ¿Cómo?

No, porque la reprogramación del signal sólo afectaría si los procesos reciben ese tipo de signal, y en el supuesto de este apartado ningún proceso recibe el SIGALRM (antes de que pasen los 10 segundos acaba la función y se desactiva el temporizador).

3. Suponiendo que la ejecución de la función *hago_algo_de trabajo()* dura siempre **MÁS** de 10 segundos:
    a) Para cada proceso, indica qué mensajes mostrará en pantalla:

Saltan los temporizadores programados por Pinicial, H1, H2, H3 y H4. Ningún proceso tiene reprogramado el SIGALRM. Pinicial porque no entra en el if de la línea 23; H1, H2 y H3 lo reprograman pero al ejecutar execlp pierden la reprogramación y el nuevo código no lo vuelve a reprogramar (no entran en el if de la línea 23); y H4 porque tampoco ejecuta el código del if donde se reprograma. Por lo tanto el tratamiento es el de por defecto (matar a los procesos) Los mensajes serían:

Pinicial, H1, H2 ,H3 y H4 muestran: "Voy a trabajar"

    b) Para cada proceso, indica qué signals recibirá:

Todos los procesos recibirán el SIGALRM. Si algún proceso hijo acaba antes de que su padre haya muerto, provocará que el padre reciba el SIGCHLD.

    c) Supón que movemos las sentencias de la línea 29 a la línea 32 a la posición a la línea 23, ¿afectaría de alguna manera a las respuestas del apartado a y b? ¿Cómo? ¿Es posible garantizar que el resultado será siempre el mismo?

Si se mueve la reprogramación será efectiva para todos los procesos (aunque los hijos que mutan la pierden, al iniciar la ejecución de la nueva instancia volverán a ejecutar la reprogramación). Por tanto todos los procesos que reciban el SIGALRM ejecutarán la función trat_sigalrm. Los mensajes mostrados y los signals recibidos dependen del orden en el que se ejecuten las sentencias.

Pinicial siempre mostrará "Voy a trabajar" y "Timeout!". Recibirá el SIGALRM y si su hijo (H1) acaba antes de que pasen los 10 segundos recibirá el SIGCHLD.

Los procesos hijos (H1, H2, H3 y H4) pueden recibir un SIGKILL de su padre en cualquier momento (en cuanto acabe el temporizador de su padre). Si no lo reciben escribirán tanto "Voy a trabajar" como "Timeout!". Si lo reciben, en cuanto llegue acabarán la ejecución y los mensajes mostrados dependerán del momento en el que lo hayan recibido. En cuanto a los signals recibidos, si el temporizador del padre salta antes que el suyo recibirán SIGKILL, si no, recibirán SIGALRM. En el caso de H1, H2 y H3, si sus hijos acaban antes que ellos recibirán SIGCHLD.

# Ejercicio 4. (T3)

Tenemos el siguiente código (simplificado) que pertenece al programa suma_vect.c

```
1.  int *x=0,*y,vector_int[10]={1,2,3,4,5,6,7,8,9,10};
2.  void main(int argc,char *argv[])
3.  {
4.  int i=0;
5.  char buffer[32];
6.  // PUNTO A
7.  x=malloc(sizeof(int)*10);
8.  y=&vector_int[0];
9.  fork();
10. Calcula(x,y);  // Realiza un cálculo basándose en x e y y el resultado va a x
11. free(x);
12. }
```

En el **PUNTO A**, observamos el siguiente fichero "maps" (por simplicidad hemos eliminado algunas líneas y algunos datos que no hemos trabajado durante el curso).

```
08048000-08049000  r-xp        /home/alumne/SO/test
08049000-0804a000  rw-         /home/alumne/SO/test
b7dbd000-b7ee2000   r-xp       /lib/tls/i686/cmov/libc-2.3.6.so
b7ee2000-b7ee9000   rw-p       /lib/tls/i686/cmov/libc-2.3.6.so
b7efa000-b7f0f000  r-xp        /lib/ld-2.3.6.so
b7f0f000-b7f10000   rw-p       /lib/ld-2.3.6.so
bfdf9000-bfe0f000  rw-p        [stack]
ffffe000-fffff000  ---p        [vdso]
```

1. Rellena el siguiente cuadro y justifícalo, relacionando el tipo de variable con la región en la que la has ubicado (PUNTO A)

| Variable | Rango direcciones donde podría estar | Nombre región |
|---|---|---|
| x | 08049000-0804a000 | /home/alumne/SO/test |
| y | 08049000-0804a000 | /home/alumne/SO/test |
| vector_int | 08049000-0804a000 | /home/alumne/SO/test |
| i | bfdf9000-bfe0f000 | stack |
| buffer | bfdf9000-bfe0f000 | stack |

**Justificación:** Variable globales en zona de datos y locales en la pila. Respecto al fichero, como correspondía a un fichero "maps", se considera correcto poner como región tanto "data" como la etiqueta del fichero.

2. Justifica por qué no aparece la región del heap en este instante de la ejecución del proceso (PUNTO A)

| |
|---|
| Porque aún no hemos pedido la memoria dinámica, por lo tanto la región del heap está vacía y no se muestra. |

3. Después del malloc, aparece una nueva región con la siguiente definición:

| |
|---|
| 0804a000-0806b000 rw-p [heap] |

a) ¿Qué pasará con el proceso hijo cuando intente acceder a la variable x? ¿Tendrá acceso al mismo espacio de direcciones?

| |
|---|
| No habrá problema porque al hacer el fork se duplica el espacio de direcciones y el espacio lógico es el mismo, por lo tanto, la dirección es válida. Simplemente se traduce a direcciones físicas diferentes. |

b) Justifica el tamaño que observamos en el heap comparado con el tamaño pedido en la línea 7. ¿Que pretende optimizar la librería de C al reservar más espacio del solicitado?

| |
|---|
| La librería de C pide más espacio para evitarse ir al kernel. Pretende evitarse el coste de la llamada a sistema. Pide más espacio porque incluye una estimación extra y espacio para sus datos (sólo la primera vez). |

4. Indica cómo serían las líneas 7 y 11 si quisiéramos hacer este mismo programa utilizando directamente la(s) llamada(s) a sistema correspondiente.

| |
|---|
| L7: sbrk(sizeof(int)*10) → aumenta el tamaño del heap sizeof(int)*10 en bytes<br><br>L11:sbrk(sizeof(int)*-10) → reduce el tamaño del heap en sizeof(int)*10 bytes |

## Ejercicio 5. (T2)

En un sistema de propósito general que aplica una política de planificación Round Robin:

1. ¿Podemos decir que el sistema es preemptivo o no preemptivo? (justifícalo indicando el significado de preemptivo y cómo has llegado a la respuesta que indicas)

   Sí. Si el sistema implementa una política preemptiva el sistema también lo es. Preemptiva significa que le puede quitar la cpu al proceso sin que él la haya liberado voluntariamente como es el caso de Round Robin cuando se termina el quantum del proceso,.

2. Cuando se produce un cambio de contexto... ¿Cuál es el criterio de la política para decidir qué proceso ocupará la CPU?

   Elegirá el primero de la cola de ready

3. ¿En qué estructura de datos almacena el SO la información de un proceso? Indica los campos más relevantes que podemos encontrar en esta estructura de datos.

   En el PCB. Campos relevantes como el PID, PPID, usuario, grupo, tabla de reprogramación de signals, signals pendientes, espacio de direcciones.…

4. ¿En qué casos un proceso estará menos tiempo en la CPU que el especificado por el Quantum? Enuméralos, justifícalo, e indica el estado en que estará al dejar la CPU.

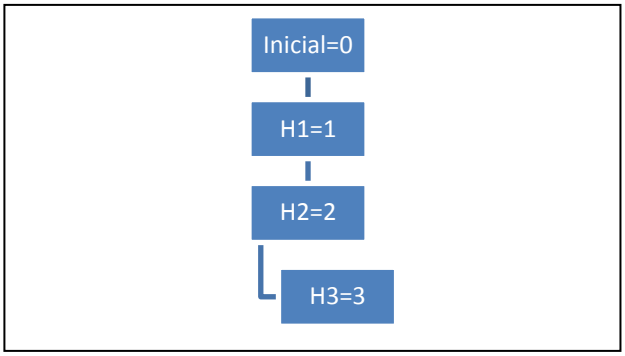
   Si termina, que pasara a zombie, y si se bloquea, que pasará a “bloqueado” o un estado similar

## Ejercicio 6. (T2, T3)

La Figura 3 muestra el código del programa “prog” (por simplicidad, se omite el código de tratamiento de errores). Contesta a las siguientes preguntas, suponiendo que se ejecuta en el Shell de la siguiente manera y que ninguna llamada a sistema provoca error:

%./prog 3

1. (T2) Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un identificador para poder referirte a ellos en el resto de preguntas




2. (T2) ¿Qué proceso(s) ejecutarán las líneas de código 32 y 33?

| |
|---|
| Solo el H3 porque los demás entran por el if de la l17 y ya no salen, ya que ejecutan el exit de la línea 25 |

3. (T2) ¿En qué orden escribirán los procesos en pantalla? ¿Podemos garantizar que el orden será siempre el mismo? Indica el orden en que escribirán los procesos.

| |
|---|
| Sí, porque la secuencia de código lo garantiza, 3,2,1,0. El hecho de tener un waitpid hace que se pueda garantizar. |

4. (T3) Supón que ejecutamos este código en un sistema basado en paginación que utiliza la optimización COW en la creación de procesos y que tiene como tamaño de página 4KB. Supón también que la región de código de este programa ocupa 1KB, la región de datos 1KB y la región de la pila 1KB. Las 3 regiones no comparten ninguna página. ¿Qué cantidad de memoria física será necesaria para cargar y ejecutar simultáneamente todos los procesos que se crean en este código?

| |
|---|
| En COW, sólo se reserva memoria física si se modifica. El código no se modifica, pero si la parte de pila y datos ya que hay variables modificadas en las dos regiones. Además, como no se puede compartir una página entre dos regiones (ni entre dos procesos), tenemos que:<br><br>1 página de código (ya que será compartido) y 4x(1Pag. datos + 1 Pag. pila) = 9 Pag. x 4kb/pag. = 36 KB |

5. (T2) Queremos modificar este código para que se deje pasar un intervalo de 3 segundos antes de crear el siguiente proceso. Indica qué líneas de código añadirías y en qué posición para conseguir este efecto.

Al inicio (1)capturaría el SIGALRM (por ejemplo en la línea 11), (2) bloquearía la recepción del SIGALRM e (3) inicializaría la máscara a usar en el sigsuspend:

```
sigset_t mask;
/* (1) */
trat.sa_flags=0;
trat.sa_handler=f_alarma;
sigemptyset(&trat.sa_mask);
sigaction(SIGALRM,&trat,NULL);
/*(2)*/
sigemptyset(&mask);
sigaddset(&mask,SIGALRM);
sigprocmask(SIG_BLOCK,&mask,NULL);
/*(3)*/
sigfillset(&mask);
sigdelset(&mask, SIGALRM);
```

Entre la 30 y 31 añadiría

```
alarm(3)
sigsuspend(&mask)
```

Y en la función de la alarma podría estar vacía:

```
void f_alarma(int s)
{}
```

```
1.  int recibido = 0;
2.      void trat_sigusr1(int signum) {
3.          recibido = 1;
4.      }
5.
6.      main(int argc, char *argv[]) {
7.      int nhijos, mipid;
8.      int ret, i;
9.      char buf[80];
10.     struct sigaction trat;
11.
12.     trat.sa_handler = trat_sigusr1;
13.     trat.sa_flags = 0;
14.     sigemptyset(&trat.sa_mask);
15.     sigaction(SIGUSR1,&trat,NULL);
16.
17.     nhijos = atoi(argv[1]);
18.     mipid = getpid();
19.     for (i=0; i<nhijos; i++) {
20.         ret = fork();
21.         if (ret > 0){
22.             if (mipid != getpid()) {
23.                 while(!recibido) ;
24.             }
25.             kill(ret, SIGUSR1);
26.             waitpid(-1, NULL, 0);
27.             sprintf(buf, "Soy el proceso %d y acabo la ejecución\n",getpid());
28.             write(1,buf,strlen(buf));
29.             exit(0);
30.         }
31.     }
32.     sprintf(buf, "Soy el proceso %d y acabo la ejecución\n",getpid());
33.     write(1,buf,strlen(buf));
34.     }
35.
```

*Figura 3 Código del programa prog*

# Ejercicio 7. (T4, T5)

1) (T4) ¿Qué es el superbloque de un sistema de ficheros? ¿qué tipo de información podemos encontrar? ¿Qué tipo de información contiene: Datos o metadatos?

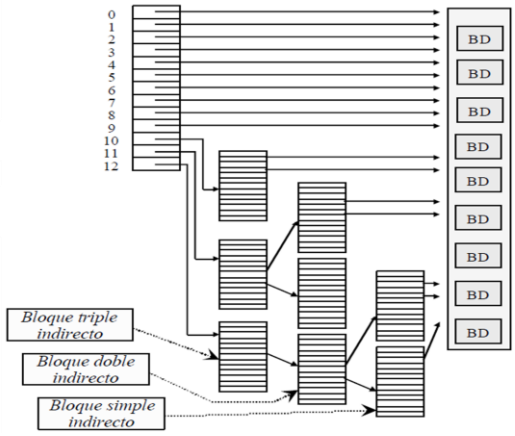
El superbloque es un bloque de disco que contiene información de control del sistema de ficheros, como el tamaño de bloque, la lista de inodos, la lista de inodos libres, etc. Forma parte de los metadatos

2) (T5) ¿Qué es una Race condition? ¿Qué es una region crítica? ¿Qué relación hay entre los dos conceptos?

Race condition (o condición de carrera) es un problema que se produce cuando en un trozo de código se accede a una (o más) variables de lectura/escritura que también se accede desde otro(s) threads que se ejecutan de forma concurrente. Una región crítica es un trozo de código que sufre una race condition

3) (T4) Explica brevemente que es el sistema indexado multinivel que se utiliza en los sistemas basados en UNIX, cuantos índices (y de qué tipo) tienen los inodos. (Haz un dibujo esquemático para mostrar cómo funciona)

En un esquema indexado multinivel se reservan un conjunto de índices fijos (10), y luego unos índices indirectos que solo se irán reservando a medida que se necesiten. En concreto 1 índice indirecto, 1 indirecto doble y 1 indirecto triple

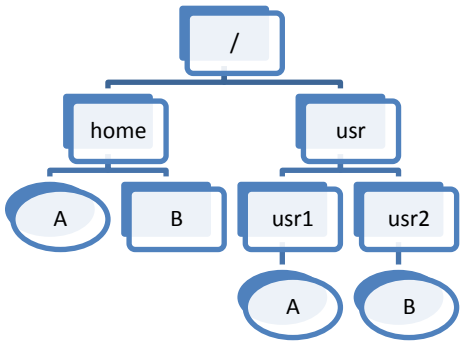



4) (T5) Enumera y explica brevemente qué ventajas tiene utilizar threads respecto a utilizar procesos.

- Coste en tiempo de gestión: creación, destrucción y cambio de contexto
- Aprovechamiento de recursos: no se replica todo el espacio de direcciones ni estructuras de datos del kernel
- Simplicidad del mecanismo de comunicación: memoria compartida

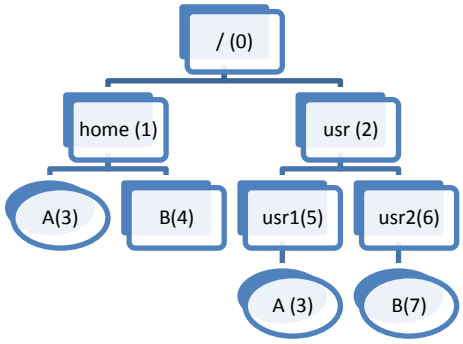
## Ejercicio 8. (T4)

Tenemos el siguiente esquema de directorios con la siguiente información adicional:




- Sabemos que el tamaño de un bloque de disco son 512 bytes y que un inodo ocupa 1 bloque.
- Sabemos que /home/A y /usr/usr1/A son hard-links. El fichero ocupa 2 KB.
- Sabemos que /usr/usr2/B es un soft-link a /home/usr/usr3 (un directorio que ya no existe)
- Los ficheros marcados con cuadros son directorios
- El kernel implementa la optimización de buffer cache, compartida para inodos y bloques de datos.

1. En el siguiente dibujo, asigna números de inodos a todos los ficheros y directorios. Ten en cuenta el tipo de fichero a la hora de hacerlo.




2. Completa el siguiente dibujo añadiendo la información de los inodos y el contenido de los bloques de datos que intervienen. Utiliza las mismas etiquetas que Linux para indicar el tipo de fichero (d=directorio,-=fichero datos, l=soft-link). (por simplicidad hemos puesto espacio solo para 5 bloques de datos). La asignación de bloques de datos se hace de forma consecutiva.

TABLA DE INODOS EN DISCO

| Num_inodo | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| #enlaces | 4 | 3 | 4 | 2 | 2 | 2 | 2 | 1 | | | |
| Tipo_fichero | d | d | d | - | d | d | d | L | | | |
| Bloques_datos | 0 | 1 | 2 | 3 | 7 | 8 | 9 | 10 | | | |
| | | | | 4 | | | | | | | |
| | | | | 5 | | | | | | | |
| | | | | 6 | | | | | | | |
| | | | | | | | | | | | |

**Bloques datos**

| Num Bloque Datos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| | . 0<br>.. 0<br>home 1<br>usr 2 | . 1<br>.. 0<br>A 3<br>B 4 | . 2<br>.. 0<br>usr1 5<br>usr2 6 | AAA | AAAA | AAA | AAA | . 4<br>.. 1 | . 5<br>.. 2<br>A 3 |

| Num Bloque Datos | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| | . 6<br>.. 2<br>B 7 | /home/usr3 | | | | | | | |

3. Si nuestro directorio actual es /home:
   a) ¿Cuál es el path absoluto para referenciar el fichero /usr/usr1/A?

   /usr/usr1/A

   b) ¿Cuál es el path relativo para referenciar el fichero /usr/usr1/A?

   ../usr/usr1/A

4. Dado el siguiente código, ejecutado en el sistema de ficheros de la figura, contesta (justifícalas todas brevemente):

```
1. char c;
2. int fd,fd1,s,pos=0;
3. fd=open("/usr/usr1/A",O_RDONLY);
4. fd1=open("/usr/usr1/A",O_WRONLY);
5. s=lseek(fd1,0,SEEK_END);
6. while(pos!=s){
7.     read(fd,&c,sizeof(char));
8.     write(fd1,&c,sizeof(char));
9.     pos++;
10. }
```

a) Describe brevemente qué hace este código

Este programa duplica el contenido de un fichero en el mismo fichero. Nos ponemos al final y vamos leyendo (del inicio) y escribiendo(al final). Controlamos el final de la operación con el tamaño del fichero.

b) Indica el número de accesos a disco totales que ejecutará la línea 3. Indica exactamente que inodos y bloques de datos habría que leer.

```
fd=open("/usr/usr1/A",O_RDONLY);
```

Leemos el inodo de /(0) , el bloque de datos de /(0). El inodo de usr(2), el bloque de datos de usr(2). El inodo de usr1(5) + el bloque de datos de usr1(8). El inodo de A(3). TOTAL 7 accesos

c) Indica si la línea 4 añadirá o no un nuevo inodo a la tabla de inodos en memoria y por qué

No añadirá ninguna porque el inodo 3 ya está cargado en la tabla y solo se carga una vez

d) Asumiendo que ninguna llamada a sistema da error, ¿cuántas iteraciones dará el bucle de las líneas 6-10?

El fichero tiene tamaño 2kb, por lo tanto dará 2x1024=2048 iteraciones

e) ¿Cuántos accesos a disco, en total, se efectuarán como consecuencia de la llamada a sistema de la línea 7? ¿Qué bloques de datos se leerán?

Los bloques de disco son de 512 bytes, para leer los 2K serán necesarios 4 bloques.

f) ¿Qué valor tendrá el puntero de lectura/escritura del canal indicado en fd al finalizar el bucle?

El puntero valdrá 2x1024=2048.

## Ejercicio 9. (T4)

La Figura 4 muestra el código del programa "prog"(por simplicidad, se omite el código de tratamiento de errores). Contesta de forma **JUSTIFICADA** a las siguientes preguntas, suponiendo que ejecutamos este programa con el siguiente comando:

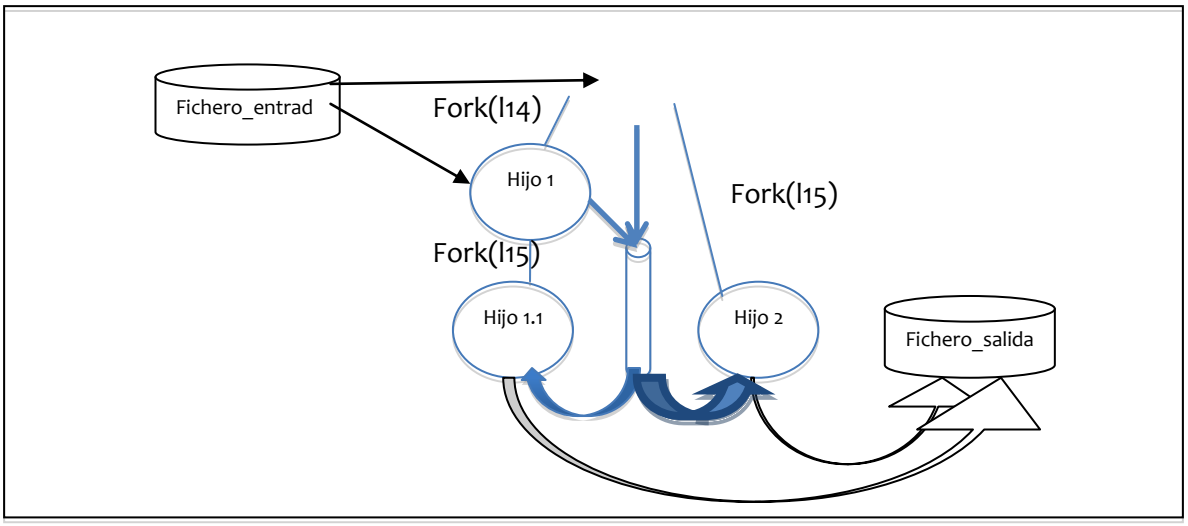
%./prog fichero_salida < fichero_entrada

Y Suponiendo que "fichero_entrada" existe y su contenido es "abcdefghijklmnñopqrstuvwxyz"

1. ¿Cuántos procesos crea este programa? ¿Cuántas pipes? Indica en qué línea de código se crea cada proceso y cada pipe de las que especifiques.

Crea 3 procesos: el proceso principal ejecuta el primer fork para crear un hijo, y luego ambos procesos ejecutan el siguiente fork creando otro hijo cada 1. Se crea una sola pipe, lo hace el proceso principal antes de crear ningún proceso.

2. Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un pid para poder referirte a ellos en el resto de preguntas. Representa también la(s) pipe(s) que se crean indicando qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una. En el dibujo indica también que procesos acceden a los ficheros "fichero_entrada" y "fichero_salida" y el tipo de acceso (lectura/escritura). Represéntalo mediante flechas que indiquen el tipo de acceso.



```
1.  main(int argc char *argv[]) {
2.  int fd_pipe[2];
3.  int fd_file;
4.  int ret, pid1, pid2;
5.  char c;
6.  char buf[80];
7.  int size;
8.
9.  close(1);
10. fd_file = open (argv[1], O_WRONLY|O_TRUNC|O_CREAT, 0660);
11.
12. pipe(fd_pipe);
13.
14. pid1 = fork();
15. pid2 = fork();
16.
17. if (pid2 == 0) {
18.       while ((ret = read(fd_pipe[0],&c, sizeof(c))) > 0)
19.             write (1,&c, sizeof(c));
20. } else {
21.       while ((ret=read(0,&c,sizeof(c)) > 0)
22.             write(pipe_fd[1], &c, ret);
23.
24.       while (waitpid(-1, NULL, 0) > 0);
25.       sprintf(buf, "Fin ejecución\n");
26.       write(2,buf,strlen(buf));
27. }
28. }
```

*Figura 4 Código de prog*

3. Completa la siguiente figura con los campos que faltan para que represente el estado de la tabla de canales, la tabla de ficheros abiertos y la tabla de inodes al inicio de la ejecución de este programa (línea 1).

Tabla Canales

| | Ent. TFA |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 2 | 0 |
| | |
| | |
| | |
| | |

Tabla Ficheros Abiertos

| | #refs | Mod | Punt l/e | Ent t. inodes |
|---|---|---|---|---|
| 0 | 2 | rw | -- | 0 |
| 1 | 1 | r | 0 | 1 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Tabla i-nodes

| | #refs | inode |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 1 | i-fichEnt |
| | | |
| | | |
| | | |
| | | |
| | | |

4. Completa ahora la siguiente figura para representar el estado de las tablas de canales de cada proceso, la tabla de ficheros abiertos y la tabla de inodes, suponiendo que todos los procesos se encuentran en la línea 16.

Tabla Canales

| | Ent. TFA |
|---|---|
| 0 | 1 |
| 1 | 2 |

Tabla Ficheros Abiertos

| | #refs | Mod | Punt l/e | Ent t. inodes |
|---|---|---|---|---|
| 0 | 4 | rw | -- | 0 |
| 1 | 4 | r | 0 | 1 |

Tabla i-nodes

| | #refs | inode |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 1 | i-fichEnt |

| | |
|---|---|
| 2 | 0 |
| 3 | 3 |
| 4 | 4 |
| | |
| | |

| | | | | |
|---|---|---|---|---|
| 2 | 4 | w | 0 | 2 |
| 3 | 4 | r | -- | 3 |
| 4 | 4 | w | -- | 3 |
| | | | | |
| | | | | |

| | | |
|---|---|---|
| 2 | 1 | i_fichSal |
| 3 | 2 | i_pipe |
| | | |
| | | |
| | | |

Tabla Canales

| |
|---|
| 1 |
| 2 |
| 0 |
| 3 |
| 4 |
| |

Tabla Canales

| |
|---|
| 1 |
| 2 |
| 0 |
| 3 |
| 4 |
| |

Tabla Canales

| |
|---|
| 1 |
| 2 |
| 0 |
| 3 |
| 4 |
| |

5. ¿Qué procesos acabarán la ejecución?

Ninguno. Los procesos hijos ejecutan un bucle que lee de la pipe mientras la lectura no devuelva 0. Pero eso nunca pasará porque nunca se cierran los canales de escritura en la pipe. Y los procesos padre se quedan bloqueados esperando

6. ¿Qué líneas de código y en qué posición las añadirías para conseguir que todos los procesos acaben la ejecución?

Hay que añadir close(fd_pipe[1]) entre las líneas 17 y 18, y en la línea 23

7. ¿Qué procesos leen "fichero_entrada"? ¿Podemos saber qué fragmento del fichero lee cada proceso? Si repetimos más veces la ejecución de este programa, con el mismo comando, ¿podemos garantizar que los mismos procesos leerán los mismos fragmentos del fichero?

El padre e hijo 1 pueden leer del fichero. El fragmento que lea cada uno dependerá del orden en el que se intercalen en la cpu: sabemos que se lee hasta el final del fichero, que comparten el puntero de lectura/escritura. Por tanto cada carácter será leído sólo por uno, pero no podemos saber cuántos caracteres leerá cada proceso. Y esto puede cambiar entre ejecuciones.

8. Al final de la ejecución, ¿cuál será el contenido de "fichero_salida"? Si repetimos más veces la ejecución de este programa, con el mismo comando, ¿podemos garantizar que el contenido será siempre el mismo?

No podemos garantizar el contenido del fichero ya que puede haber un cambio de contexto entre la lectura de la pipe y la escritura en el fichero. Y también entre la lectura del fichero y la escritura en la pipe.

## Ejercicio 10. (T4,T5)

1. *(T4)* Explica brevemente que efecto tiene, en las tablas de gestión de entrada/salida, la ejecución de un fork y un exit (explícalos por separado).

   El fork copia la tabla de canales del padre al hijo, lo cual implica actualizar los contadores de referencia de la TFA, incrementando en 1 por cada canal que apuntase a esa entrada. Eso implica que padre e hijo comparten las entradas de la TFA y por lo tanto camos como el puntero de l/e. Un fork no modifica la tabla de inodos.

   exit implica cerrar todos los canales de un proceso. Lo cual implica actualizar las referencias de la TFA, potencialmente cerrar alguna entrada y, en ese caso, actualizar la tabla de inodos y potencialmente cerrar alguna entrada.

2. *(T4)* Enumera las llamadas a sistema de entrada/salida que pueden generar nuevas entradas en la tabla de canales

   Open, dup, dup2, pipe.

   Nota: Esto es lo que considerábamos como base, tampoco se penaliza poner creat o dup3.

3. *(T4)* Tenemos el siguiente código que utilizan dos procesos no emparentados, que se quieren intercambiar sus pids utilizando dos pipes con nombre, una para cada sentido de la comunicación (se muestra sólo el intercambio de datos). Cada uno abre las pipes en el modo correcto y los dos ejecutan este mismo código. Indica si sería correcto o no y justifícalo.

```
…. // Aquí se abrirían las pipes.
int su_pid,mi_pid;
mi_pid=getpid();
read(fd_lect,&su_pid,sizeof(int)); close(fd_lect);
write(fd_esc,&mi_pid,sizeof(int)); close(fd_esc);
```

No es correcto ya que los dos se quedarán bloqueados en el read.

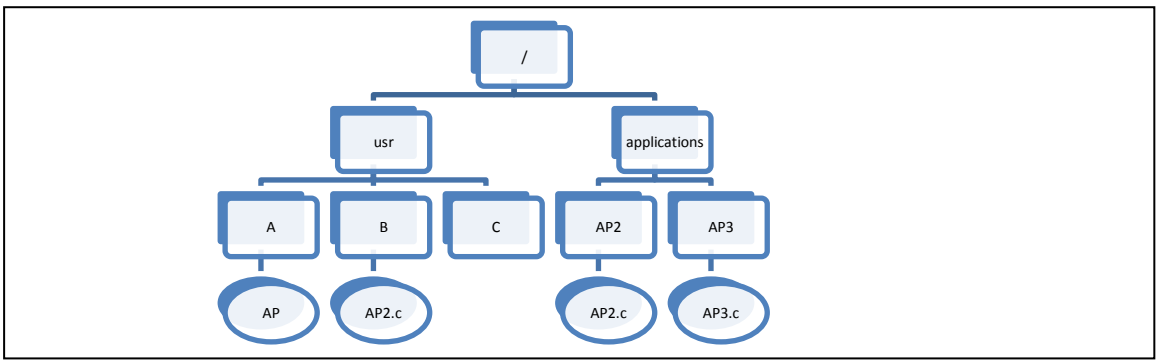
4. *(T4)* En un sistema de ficheros basado en inodos, ¿Cuáles son los dos tipos de enlace (*links*) que podemos encontrar? Explícalos brevemente comentando cómo se implementan en este tipo de sistemas de fichero (fichero especial (si/no), información relacionada, cómo afecta a la hora de acceder al fichero, etc).

Had-link. Es cuando tenemos el nombre del fichero vinculado directamente con el inodo que contiene los datos del fichero. En la información que aparece en el directorio tenemos el nombre del fichero y el número de inodo donde están los datos. Por simplicidad , no se permiten hard-links a directorios.

Soft-link. Es cuando el nombre nos vincula con una referencia indirecta. En este caso es un tipo de fichero especial cuyo contenido es un path a otro fichero o directorio. En este caso en el directorio el nombre tienen un inodo diferente al del inodo que contiene los datos a los que estamos apuntando.

## Ejercicio 11. (T4)

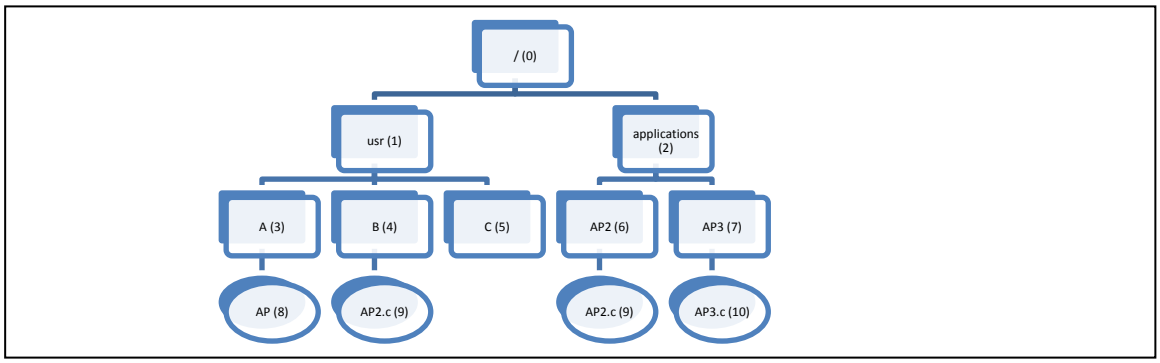
Dado el siguiente esquema de directorios en un sistema de ficheros basado en inodos:



Del cual nos dicen que /usr/A/AP es un soft-link a /applications/AP2/AP2.c y que /usr/B/AP2.c y /applications/AP2/AP2.c son hard-links. Sabiendo que:

- tipo de datos puede ser dir = directorio, dat= fichero normal, link=softlink.
- El directorio raíz es el inodo 0.
- Los cuadrados indican directorios.
- El fichero AP3.c ocupa 3 bloques de datos y el resto 1 bloque de datos.

1. Rellena los siguientes inodos y bloques de datos. Para ello:
   a) Asigna primero un inodo a cada elemento del esquema de ficheros y directorios y anótalo junto a su nombre en la figura.




   b) Asigna bloques de datos a cada elemento del esquema de directorio teniendo en cuenta los tamaños que os hemos indicado.
   c) Completa la figura de la tabla de inodos rellenando los campos que aparecen y los bloques de datos con la información que conozcas sobre su contenido

**Tabla de Inodos en disco**

| Num_inodo | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| #links | 4 | 5 | 4 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | 1 |
| Tipo_fichero | dir | Dir | dir | Dir | dir | Dir | dir | dir | Link | Data | Data |
| Bloques_datos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10,11,12 |

**Bloques datos**

| Num Bloque | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Contenido | . 0<br>.. 0<br>Usr 1 | . 1<br>.. 0<br>A 3 | . 2<br>.. 0<br>AP2 6 | . 3<br>.. 1<br>AP 8 | . 4<br>.. 1<br>AP2.c 9 | . 5<br>.. 1 | . 6<br>.. 2<br>AP2.c 9 | . 7<br>.. 2<br>AP3.c 10 | /applications<br>/AP2/AP2.c |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Applications 2 | B 4<br>C 5 | AP3 7 | | | | | | |

| Num Bloque | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| Contenido | Cont. AP2.c | Cont .AP3.c | Cont. AP3.c | Cont. AP3.c | | | | | |

Justificación:

- Cada fichero o directorio tiene un inodo, excepto los hard-links que apuntan al mismo inodo, por eso no necesitamos 11 sino 10.
- Los soft-links son ficheros especiales cuyo contenido es el path del fichero a que puntan
- El contenido de un directorio es una tabla con dos columnas: nombre fichero y número de inodo del fichero. Un directorio siempre incluye dos ficheros de tipo directorio: el . y ..
- El fichero . hace referencia al inodo del directorio en el que se está en ese momento y el .. al inodo del directorio padre
- Las referencias de un inodo es el número total de nombres de ficheros que apuntan a ese inodo (incluidos . y ..)
- Bloques de datos es la lista de bloques de datos de un fichero. Si hubieran mas de 10

2. Indica cuántos accesos a disco (y cuáles) hacen las siguientes llamadas a sistema (indícalas una por una). Supón que en el momento de ejecutar esta secuencia de llamadas a sistema, el sistema acababa de iniciarse y ningún otro proceso estaba usando ningún fichero del sistema de ficheros. Sabemos que un bloque son 4096 bytes, que el sistema dispone de buffer cache en la que se pueden mantener simultáneamente 1024 bloques, y que al cerrar un canal el inodo siempre se escribe en disco.

Aclaración:

- El open accede a los inodos necesarios hasta llegar al inodo destino, pero no lee ningún bloque de datos del fichero destino
- Si al ya hemos leido un bloque (o inodo) previamente, estará en la buffer cache, por lo que no genera acceso a disco
- Al hacer un open con el flag O_CREAT, hay que crear un fichero nuevo (ya que no existe). Eso implica: un inodo nuevo, escribir el bloque de datos del directorio para añadir un fichero, cambiar el tamaño del directorio para indicar que ha variado su tamaño. Estas modificaciones había que reflejarlas en algún momento. Como máximo al cerrar el fichero.
- El lseek no genera accesos a disco en ningún caso.

```
char buff[4096];

1. fd=open("/applications/AP3/AP3.c",O_RDONLY);
2. fd1=open("/usr/C/AP3.c",O_WRONLY|O_CREAT,0660);
3. ret= read(fd,buff,sizeof(buff));
4. lseek(fd,4096, SEEK_CUR);
5. write(fd1,buff,ret);
6. close(fd1);
```

| | Accesos a disco (cuáles) | Tablas Modificadas (SI/NO) | | |
|---|---|---|---|---|
| | | Canales | F. Abiertos | Inodos |
| 1 | I0+B0+I2+B2+I7+B7+I10 | SI | SI | SI |
| 2 | I1+B1+I5+B5+Nuevo inodo (11?) | SI | SI | SI |
| 3 | B10, primero de AP3.c | NO | SI | NO |
| 4 | Ninguno. | NO | SI | NO |
| 5 | Nuevo Bloque (B13?), para fichero nuevo. | NO | SI | SI |
| 6 | I11+(B5+I5), por el O_CREAT | SI | SI | SI |
| 7 | I10 | SI | SI | SI |

## Ejercicio 12. (T4)

Tenemos el programa "prog" que se genera al compilar el siguiente código (por simplicidad, se omite el código de tratamiento de errores):

```
1.  #include <stdio.h>
2.  #include <unistd.h>
3.  #include <string.h>
4.
5.  main() {
6.
7.  int fd[2];
8.  int ret, pid1, pid2;
9.  char c;
10. char buf[80];
11.
12.
13. pid1 = fork();
14. pipe(fd);
15. pid2 = fork();
16.
17. if (pid2 == 0) {
18.       while ((ret = read(fd[0],&c, sizeof(c))) > 0)
19.             write (1,&c, sizeof(c));
20. } else {
21.       sprintf(buf, "Te notifico mi pid: %d\n",getpid());
22.       write(fd[1], buf, strlen(buf));
23.       while (waitpid(-1, NULL, 0) > 0);
24. }
25.
26. }
27.
```

Contesta de forma justificada a las siguientes preguntas.

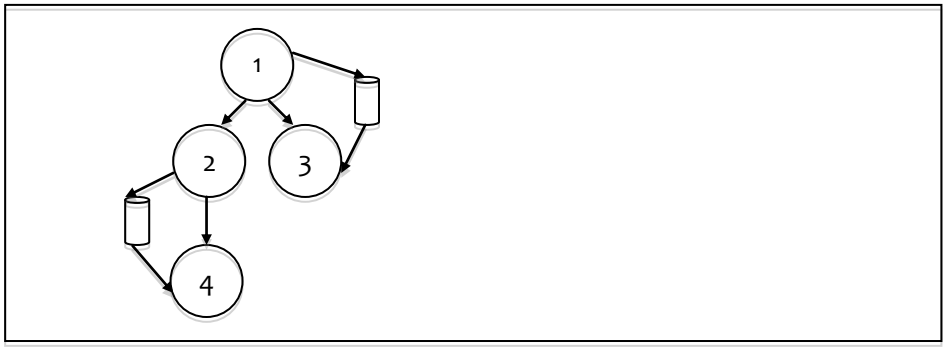
**Suponiendo que el programa se ejecuta de la siguiente manera:**

**%./prog**

1. Análisis del código:

   a) Indica cuántos procesos crea este programa y cuántas pipes. Además representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un pid para poder referirte a ellos en el resto de preguntas. Indica claramente que procesos se comunican entre sí y con qué pipe (asígnale a las pipes alguna etiqueta si crees que puede ayudar). Representa también el uso concreto que hace cada proceso de cada pipe: qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una.

   Crea 3 procesos (a parte del inicial) y 2 pipes, ya que la llamada pipe está justo después del fork y no está condicionada al resultado del fork.




b) ¿Qué proceso(s) ejecutará(n) las líneas de código 18 y 19? ¿Y las líneas entre la 21 y la 23? (justifícalo)

La 18 y la 19 los últimos procesos creados (3 y 4), y de la 21 a la 23 el proceso inicial y su primer hijo (1 y 2)

c) ¿Es necesario añadir algún código para sincronizar a los procesos en el acceso a la(s) pipe(s)?

No, porque en cada pipe sólo hay un lector y un escritor. Y entre parejas no hace falta que se sincronicen.

d) ¿Qué mensajes aparecerán en la pantalla?

Aparecerá "Te notifico mi pid:1 " y "Te notifico mi pid: 2". Ya que los procesos 3 y 4 escriben byte a byte, los textos podrían aparecer mezclados en la salida std. Ya que es compartida.

e) ¿Qué procesos acabaran la ejecución? ¿Por qué? ¿Qué cambios añadirías al código para conseguir que todos los procesos acaben la ejecución sin modificar la funcionalidad del código? Se valorará que el número de cambios sea el menor posible.

Ninguno. Los procesos lectores de las pipes se quedarán bloqueados esperando a que alguien escriba algo en los canales de escritura de las pipes. Y los procesos escritores (que son los padres de los lectores) se quedarán bloqueados en el waitpid esperando a que sus hijos acaben. Sólo hace falta cerrar los canales de escritura de la pipe: los lectores lo deben hacer antes de entrar en el bucle de lectura y los escritores después de escribir.

2. Acceso a las estructuras de datos del kernel:

La siguiente figura representa el estado de la tabla de canales, la tabla de ficheros abiertos y la tabla de inodes al inicio de la ejecución de este programa. Completa la figura representando el

estado de las tablas de canales de todos los procesos que intervienen, la tabla de ficheros abiertos y la tabla de inodes, suponiendo que todos los procesos se encuentran en la línea 16. (en la figura solo se muestra la tabla de canales del proceso inicial)

| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 1 |
| 4 | 2 |
| 5 | |

| TFA | #ref | modo | Puntl/e | Ent.T. inodo |
|---|---|---|---|---|
| 0 | 12 | rw | | 0 |
| 1 | 2 | R | | 1 |
| 2 | 2 | W | | 1 |
| 3 | 2 | R | | 2 |
| 4 | 2 | W | | 2 |
| 5 | | | | |

| T.inodo | #ref | inodo |
|---|---|---|
| 0 | 1 | tty |
| 1 | 2 | Pipe1 |
| 2 | 2 | Pipe2 |
| 3 | | |
| 4 | | |
| 5 | | |

Proceso 2

| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 3 |
| 4 | 4 |
| 5 | |

Proceso 3

| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 1 |
| 4 | 2 |
| 5 | |

Proceso 4

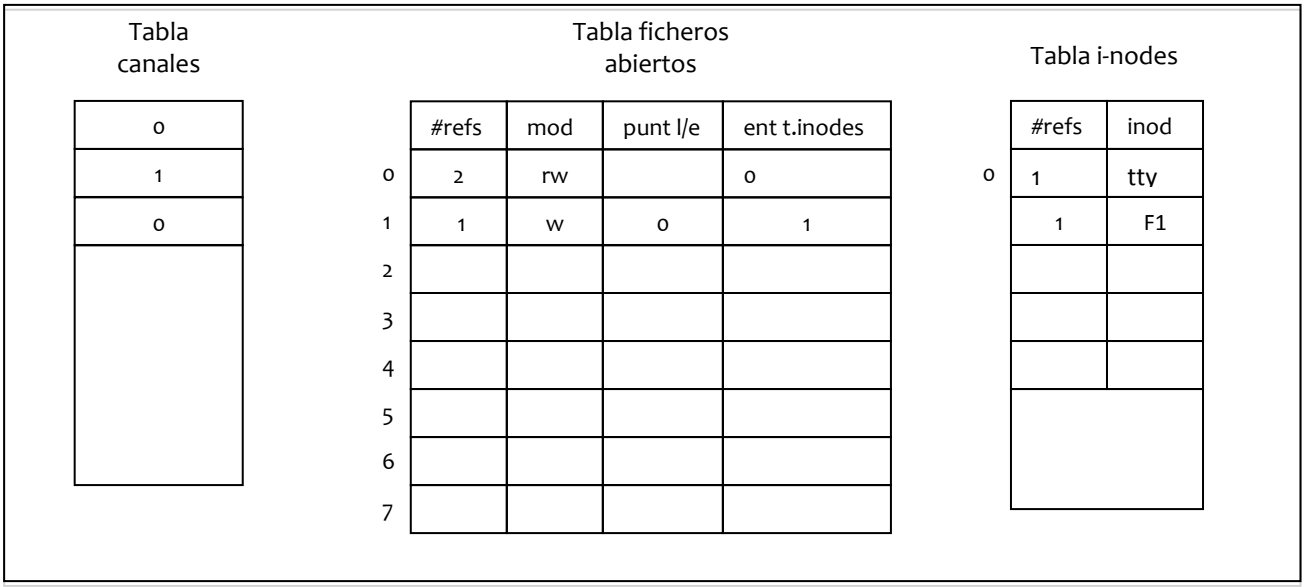
| Canal | Ent. TFA |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 3 |
| 4 | 4 |
| 5 | |

3. Modificamos su comportamiento

**Suponiendo que el programa se ejecuta de la siguiente manera:**

**%./prog > f1**

a) ¿Cambiaría de alguna manera el estado inicial representado en la figura del apartado anterior? Representa en la siguiente figura el estado inicial que tendríamos al ejecutar de esta manera el programa.




b) ¿Qué mensajes aparecerán en pantalla?

Ninguno. Hemos redireccionado la salida estándar para asociarla a f1 y todos los procesos la heredarán.

c) ¿Qué procesos acabarán la ejecución? ¿Por qué? ¿Qué cambios añadirías al código para conseguir que todos los procesos acaben la ejecución sin modificar la funcionalidad del código? Se valorará que el número de cambios sea el menor posible.

Ninguno. El cambio de la redirección no afecta a la causa del bloqueo del ejercicio.

## Ejercicio 13. (T3)

1. ¿Cuáles son las dos principales funcionalidades de la Memory Management Unit (MMU)? (indica cuáles son y en qué consisten)

Traducción de direcciones (lógicas a físicas) y protección (no poder acceder fuera del espacio de direcciones lógicas del proceso)

2. Explica qué aporta la optimización de la memoria virtual respecto a tener simplemente paginación y carga bajo demanda

En la memoria virtual, el sistema puede decidir sacar páginas de memoria y guardarlas en la zona de swap para liberar marcos y poder ser utilizados por otros procesos. La optimización se aplica a nivel de página, por lo que se aprovecha la paginación y es un añadido a la carga bajo demanda, ya que se cargan las páginas a medida que se necesitan

## Ejercicio 14. (T2)

Dado los siguientes códigos

Código 1

```
int recibido=0;
void f(int s)
{ recibido=1; }
void main()
{
struct sigaction trat;
trat.sa_flags = 0;
trat.sa_handler = f;
sigemptyset(&trat.sa_mask);
sigaction(SIGUSR1,&trat, NULL);
…
while(recibido==0);
recibido=0;
…
}
```

Código 2

```
void f(int s)
{ }
void main()
{
struct sigaction trat;
sigset_t mask;
trat.sa_flags = 0;
trat.sa_handler = f;
sigemptyset(&trat.sa_mask);
sigaction(SIGUSR1,&trat, NULL);
…
sigfillset(&mask);
sigdelset(&mask, SIGUSR1);
sigsuspend();
…
}
```

Contesta a las siguientes preguntas:

1. ¿Cuál de los dos códigos corresponde a una espera activa?

El código 1, ya que espera la llegada del evento consultando el valor de la variable, lo cual consume cpu. La opción 2 bloquea al proceso por lo que no consume cpu.

2. ¿Es necesaria la reprogramación del signal SIGUSR1 en el código 2? ¿Qué pasaría si no lo hacemos?

Es necesario ya que la acción por defecto del signal SIGUSR1 es acabar la ejecución del proceso, si no lo reprogramamos el proceso morirá al recibir el signal

3. ¿En qué región de memoria podremos encontrar la variable "recibido"?

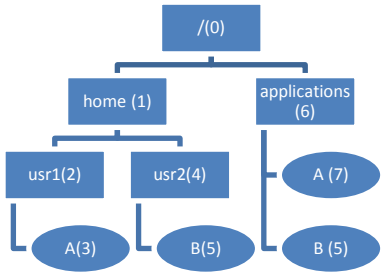
Como es una variable global, estará en la región de data

4. ¿Habrá alguna diferencia de comportamiento entre el código 1 y el 2 si sabemos que estos dos códigos, durante su ejecución, recibirán un único evento SIGUSR1? (en caso afirmativo indica cuál sería el más indicado y por qué).

Si solo recibe 1 evento, el código 2 podría quedarse bloqueado indefinidamente ya que el evento podría llegar antes del sigsuspend. Deberíamos elegir el código 1 si queremos garantizar que se comporta como es esperado. La versión 2 del código funcionaría si en todo el código excepto el sisgsupend la recepción del SIGUSR1 estuviera bloqueada (usando sigprocmask)

## Ejercicio 15. (T4)

Supón un sistema de ficheros basado en inodos que tiene la siguiente estructura (los cuadrados indican directorios, entre paréntesis hemos puesto el número de inodo):




Sabemos que:

- /home/usr1/A es un soft-link a /applications/A
- /home/usr2/B es un hard-link a /applications/B
- /applications/B es un fichero vacío

- /applications/A ocupa 1000 bytes
- Cada directorio y cada inodo ocupan 1 bloque de disco
- El tamaño de un bloque de disco es de 512bytes

Y ejecutamos la siguiente secuencia de código:

```
1. char c[100];int i,fd_in,fd_out,ret;
2. fd_in=open("/home/usr1/A",O_RDONLY);
3. fd_out=open("/home/usr2/B",O_WRONLY);
4. ret=read(fd_in,c,sizeof(c));
5. while(ret>0){
6.    for(i=0;i<ret;i++) write(fd_out,&c[i],1);
7.    ret=read(fd_in,c,sizeof(c));
8. }
```

Contesta las siguientes pregunta :

1. ¿Qué inodo(s) cargaremos en la tabla de inodos en memoria al ejecutar la llamada a sistema de la línea 2?

Cargaremos el inodo 7 que es al que apunta /home/usr1/A →/applications/A. En la tabla de inodos en memoria solo se guarda el inodo destino, no los intermedios.

2. ¿Cuántos y cuáles accesos a disco deberemos hacer para ejecutar la llamada a sistema de la línea 2? Indica cuáles corresponden a inodos y cuáles a bloques de datos (aunque no pongas el número de los bloques de datos). Asume que no hay buffer cache.

Debemos acceder primero /home/usr1/A y luego a /applications/A

Inodo /+Datos dir /+ inodo home + Datos dir. Home + inodo usr1 + BD dir usr1 + inodo A + BD A+ inodo / + BD dir / + inodos applications + BD applications dir +inodo 7= 13 accesos

3. Si sabemos que una llamada a sistema tarda 10ms, ¿Cuánto tiempo invertirá este código (EN TOTAL) sólo en llamadas a sistema? (Indica el coste en ms al lado de cada línea de código y el total al final).

ACLARACIÓN: El fichero tiene 1000 bytes, por lo que se hacen 10 lecturas (sizeof(c) =100) y una última llamada a sistema que devuelve 0. Las escrituras se hacen byte a byte, por lo que hay 1000

```
1.  char c[100];
2.  fd_in=open("/home/usr1/A",O_RDONLY); ➔10ms
3.  fd_out=open("/home/usr2/B",O_WRONLY);➔10ms
4.  ret=read(fd_in,c,sizeof(c)); ➔10ms
5.  while(ret>0){
6.    for(i=0;i<ret;i++) write(fd_out,&c[i],1); ➔10x100=1000ms
7.    ret=read(fd_in,c,sizeof(c));➔10ms
8.  }

TOTAL= 10+10+10+(1000+10)*10=30+10100=10130ms
```

4. ¿Cuántos accesos a disco (en total) realizará el bucle de las líneas 5 a 8 en el caso de …

   a) No tener buffer cache

   Si no hay buffer cache, cada iteración del bucle de escritura ira a disco.

   1000 accesos a disco para escribir

   9 accesos a disco para leer (hay 1 read fuera del bucle)

   b) Tener una buffer cache de 1000 bloques y sabiendo que las escrituras se sincronizan en disco al cerrar el fichero.

   Según el enunciado, las escrituras no generaran accesos y solo quedaran las lecturas. Como el fichero ocupa 2 bloques, haremos 2 accesos a disco en total, pero como ya hemos hecho 1 antes del bucle, solo 1.

## Ejercicio 16. (T4)

Tenemos los siguientes códigos prog1.c y prog2.c, de los que omitimos la gestión de errores para facilitar la legibilidad:

```
1.  / * codigo de prog1.c */
2.  main() {
3.
4.  int pid_h;
5.  int pipe_fd[2];
6.  char buf [80];
7.
8.  pipe(pipe_fd);
9.
10. pid_h =fork();
11.
12. dup2(pipe_fd[0], 0);
13. dup2(pipe_fd[1], 1);
14.
15. close(pipe_fd[0]);
16. close(pipe_fd[1]);
17.
18. sprintf(buf, "%d", pid_h);
19.
20. execlp("./prog2", "prog2", buf, (char *) NULL);
21.
22.
23. }
24.
```

```
1.  / * codigo de prog2.c */
2.  int turno_escr;
3.
4.  void trat_sigusr1(int signum) {
5.      turno_escr = 1;
6.  }
7.
8.  main (int argc, char *argv[]) {
9.
10. char buf [80];
11. int pid_dest;
12. int i,ret,valor_rec;
13. int valor  = getpid();
14. struct sigaction trat;
15. trat.sa_flags = 0;
16. trat.sa_handler=trat_sigusr1;
17. sigemptyset(&trat.sa_mask);
18.
19. sigaction(SIGUSR1, &trat, NULL);
20.
21. pid_dest = atoi(argv[1]);
22.
23. if (pid_dest == 0) {
24.     pid_dest = getppid();
25.     write(1, &valor,sizeof(valor));
26.     turno_escr = 0;
27.     kill(pid_dest,SIGUSR1);
28.     valor ++;
29.
30. } else {
31.     turno_escr = 0;
32. }
33.
34. for (i = 0; i< 5; i++) {
35.     while (!turno_escr);
36.     ret=read (0,&valor_rec,sizeof(valor_rec));
37.     sprintf(buf,"%d",valor_rec);
38.     write(2,buf,ret);
39.     write(2,"\n",1);
40.     write(1,&valor,sizeof(valor));
41.     turno_escr = 0;
42.     kill(pid_dest,SIGUSR1);
43.     valor ++;
44.
```

Supón que en el directorio actual de trabajo tenemos los ejecutables prog1 y prog2, y que ejecutamos el siguiente comando: %./prog1. Contesta razonadamente a las siguientes preguntas

1. ¿Cuántos procesos se crearán? ¿Cuántas pipes?

   2 procesos contanto el inicial. Una pipe

4. Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un pid para poder referirte a ellos en el resto de preguntas. Representa también las pipes que se crean indicando qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una.

5. ¿Qué proceso(s) ejecutará(n) el código de prog2?

Los dos procesos, ya que ambos ejecutan la llamada a sistema exec que tenemos en prog1

6. ¿Qué mensajes aparecerán en el terminal?

El proceso inicial mostrará 5 mensajes, que será un número entero partiendo del pid de su hijo

Y el proceso hijo mostrará 5 mensajes, que también serán números enteros partiendo del pid de su padre

7. Describe brevemente el funcionamiento de este código. ¿Para qué se están utilizando los signals?

Los dos procesos hacen el mismo trabajo pero de manera alternada. El proceso que tiene el turno escribe un valor en la pipe y queda a la espera de recibir de nuevo el turno. Cuando recibe el turno (signal SIGUSR1) lee de la pipe, muestra en salida de errores estándar lo que ha recibido, incrementa el valor y cede el turno enviando al otro proceso un SIGUSR1. El primer proceso en tener el turno es el hijo, ya que cada proceso recibe como parámetro del main de prog2 el valor de retorno del fork. Los signals se están utilizando para sincronizar el acceso a la pipe.

8. Supón que el código prog1 ocupa 1KB y el código de prog2 ocupa 4KB. La máquina en la que ejecutamos estos códigos tiene un sistema de memoria basado en paginación, las páginas miden 4KB y utiliza la optimización copy-on-write en el fork. ¿Cuánta memoria necesitaremos para soportar el código de todos los procesos simultáneamente, suponiendo que cada proceso se encuentra en la última instrucción antes de acabar su ejecución?

Cada proceso carga el programa prog2, que ocupa 4 KB. Como las páginas son de 4KB necesitaremos 8KB para el código de los dos procesos. Justo antes de acabar la ejecución no tiene efecto el copy-on-write porque ambos procesos, padre e hijo, han mutado después del fork.

9. La siguiente figura representa el estado de la tabla de canales, la tabla de ficheros abiertos y la tabla de inodos al inicio de la ejecución de este programa. Completa la figura representando el estado de las tablas de canales de cada proceso, la tabla de ficheros abiertos y la tabla de inodos, suponiendo que todos los procesos están ejecutando la última instrucción antes de acabar su ejecución.

T. canales

| | |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| | |
| | |

T. Ficheros abiertos

| | #refs | Modo | Punt l/e | Ent T. inodos |
|---|---|---|---|---|
| 0 | 3 | rw | -- | 0 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

T. inodos

| | #refs | inodo |
|---|---|---|
| 0 | 1 | i-tty |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

T. canales

| | |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 0 |
| | |
| | |

T. Ficheros abiertos

| | #refs | Modo | Punt l/e | Ent T. inodes |
|---|---|---|---|---|
| 0 | 1 | rw | -- | 0 |
| 1 | 1 | r | -- | 1 |
| 2 | 1 | w | -- | 1 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

T. inodes

| | #refs | inode |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 2 | i-pipe |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

T. canales

| | |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 0 |

# Ejercicio 17. (T1,T3,T4)

1. *(T1)* ¿Un hardware que ofrece cuatro modos de ejecución, uno de ellos de usuario y tres niveles de modos privilegiados, puede ofrecer llamadas a sistema seguras?

   Si, necesitamos al menos 1 modo usuario y 1 modo privilegiado, pero se pueden tener más modos sin problemas

2. *(T3)* Observamos que dos procesos tienen el mismo espacio lógico y físico, ¿Qué optimización ofrece este S.O. para que esta situación sea posible? ¿Tiene que existir alguna relación entre los procesos para que sea posible?

   COW y los procesos tienen que estar relacionados por herencia (por ejemplo, padre e hijo)

3. *(T3)* Explica brevemente para que se utiliza el área de swap cuando aplicamos el algoritmo de reemplazo en la optimización de memoria virtual.

El área de swap se utiliza para almacenar las páginas víctimas (las seleccionadas para ser expulsadas de memoria por el algoritmo de reemplazo). Estas páginas deben guardarse en el almacenamiento secundario por si el proceso las vuelve a referenciar en el futuro.

4. *(T4)* ¿De qué tipo son los ficheros . y .. que encontramos en los directorios y a que hacen referencia?

. and .. son directories que ser refieren al directorio actual y a su padre respectivamente.

## Ejercicio 18. (T2, T3, T5)

CASO 1 PROCESOS: *(T2,T3)* Si sabemos que un proceso ocupa (4 KB de código, 4KB de datos y 4KB de pila) a nivel usuario y que el kernel reserva PCBs de 4KB, que espacio necesitaremos (en total, usuario+sistema, incluido el proceso inicial), ejecutando el siguiente código, en los siguientes casos :

```
1. fork();
2. fork();
3. fork();
```

1. El sistema no implementa COW y todos los procesos están justo después del último fork

Usuario:Hay 8 procesos en total → no se comparte nada→8procesosx12KB=96KB

Kernel: 8 procesos x 4kb→32KB

2. El sistema implementa COW y todos los procesos están justo después del último fork

Usuario: Hay 8 procesos en total → pero se comparte todo → 12KB

Kernel: igual que antes→ 32KB

CASO 2 THREADS: *(T3,T5)* El proceso realiza las siguientes llamadas, sabiendo que todo es correcto y que el sistema no ofrece COW, ¿Qué espacio habrá reservado en total (incluyendo el proceso inicial) al inicio de la línea 4?

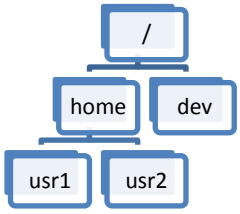
```
1. pthread_create(&th1,función_th,null, null);
2. pthread_create(&th2,función_th,null, null);
3. pthread_create(&th3,función_th,null, null);
4.
```

Usuario: El proceso ocupa 12KB + 3 threads extras→3 pilas →12KB+12KB=24KB

Kernel: 1 proceso=4KB

## Ejercicio 19. (T4)

Tenemos el siguiente sistema de ficheros (todo son directorios):




Y nos dan el siguiente código:

```
1. void f(int i){
2.   char c,buff[64];
3.   int fd;
4.   sprintf(buff,"/home/usr1/%d",i);
5.   fd=open(buff,O_WRONLY|O_CREAT,0777);
6.   while(read(0,&c,1)>0) write(fd,&c,1);
7. }
8. void main(int argv,char *argv[]){
9.    int i,pid;
10.    for(i=0;i<4;i++){
11.     pid=fork();
12.     if(pid==0){
13.      f(i);
14.      exit(0);
15.     }
16.    }
17.    f(i);
18.    while(waitpid(-1,null,0)>0);
19. }
```

1. Describe brevemente que hace este código (procesos que se crean, que hace cada uno , etc)

Este código crea 4 procesos, todos ejecutan la función f que lee de la entrada std y guarda lo que lee en un fichero nuevo para cada proceso. El padre también ejecuta la función y al acabar espera a los hijos.

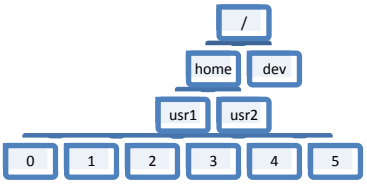
2. Sabiendo que, al inicio del programa, solo están creados los canales 0,1 y 2, ¿qué valor tendrá fd para cada uno de los procesos?

Para todos valdrá 3

3. ¿Cuántas entradas en la Tabla de ficheros abiertos generará este código?

| |
|---|
| 5, ya que hay 5 open's en total |

4. Dibuja como quedará el sistema de ficheros al acabar el programa




5. Sabemos que la entrada std es la consola, y sabemos que el usuario teclea los siguientes caracteres *1234567890* y a continuación apreta ctr-D, ¿Cuál será el contenido de cada fichero que se genera?

| |
|---|
| No lo podemos saber pq se ejecutan de forma concurrente |

6. Si nos dicen que el sistema no ofrece buffer cache y que las escrituras en disco se hacen de forma inmediata, ¿Cuántos accesos a disco se producirán en total como consecuencia del bucle de la línea 6?

| |
|---|
| El read, al ser de la consola, no generará accesos a disco, el write, al ser byte a byte y no tener buffer cache, generará tantos accesos como write's se realicen, es decir 10 (para '*1234567890*'). |

7. Antes antes de ejecutar este código tenemos los inodos y bloques de datos con el siguiente contenido. Modifícalo para representar como quedará para este caso concreto: asume que los procesos se ejecutan en orden de creación (i=0,1,2,3,4) y que cada proceso lee una letra de la entrada std.

TABLA DE INODOS EN DISCO

| Num_inodo | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| #enlaces | 4 | 4 | 2 | 2 | 2 | | | | | | |
| Tipo_fichero | d | d | d | d | d | | | | | | |
| Bloques_datos | 0 | 1 | 2 | 3 | 4 | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |

Bloques datos

| Num Bloque | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|

Datos

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| . 0<br>.. 0<br>home 1<br>dev 2 | . 1<br>.. 0<br>usr1 3<br>usr2 4 | . 2<br>.. 0 | . 3<br>.. 1 | . 4<br>.. 1 | | | | |

Num Bloque Datos

| 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

## Ejercicio 20. (T2, T4)

La Figura 5 contiene el código del programa "prog" (por simplicidad, se omite el código de tratamiento de errores). Contesta de forma **JUSTIFICADA** a las siguientes preguntas, suponiendo que ejecutamos este programa con el siguiente comando y que ninguna llamada a sistema provoca error:

%./prog < fichero_entrada > fichero salida

Y suponiendo que "fichero_entrada" existe y su contenido es "abcdefghijklmnñopqrstuvwxyz"

1. ¿Cuántos procesos crea este programa? ¿Cuántas pipes? Representa la jerarquía de procesos que creará este programa en ejecución. Asigna a cada proceso un identificador para poder referirte a ellos en el resto de preguntas. Representa también las pipes que se crean indicando qué proceso(s) lee(n) de cada una y qué proceso(s) escribe(n) en cada una. En el dibujo indica también (mediante flechas) que procesos acceden a los ficheros "fichero_entrada" y "fichero_salida" y el modo de acceso (lectura/ escritura).

Se crean 2 procesos (contando al inicial en total hay 3 procesos) y se crean dos pipes. El proceso inicial crea una pipe en la línea 6 y a continuación crea a un proceso en la línea 7. Este proceso hijo creará otra pipe en la línea 13 y a otro proceso en la línea 14. El proceso inicial empieza la ejecución con la entrada estándar asociada a fichero_entrada y no modifica esa asociación. Por tanto cuando mute para ejecutar cat y lea de canal 0 estará leyendo de fichero_entrada. También empieza la ejecución con la salida estándar redireccionada , en este caso asociada a fichero_salida y así la hereda su hijo. Sin embargo, despúes de crear al hijo, asocia el canal 1 al extremo de escritura de la pipe 1. Por lo que cuando mute y el código de cat escriba en canal 1 lo hará en esa pipe. El primer hijo (hijo 1) crea a otro hijo antes de redireccionar ningún canal, por lo que este nuevo proceso hereda los canales estándar tal y como los tenía el proceso inicial al empezar la ejecución. El Hijo 1, una vez creado a hijo 1.1., modifica la asociación de los canales estándar de la siguiente manera: asocia canal 0 al extremo de lectura de pipe1 y el canal 1 al extremo de escritura de pipe 2. Por lo que cuando mute para ejecutar cat, leerá de una pipe y escribirá en la otra. Por último Hijo 1.1. únicamente modifica su canal 0 para asociarlo al extremo de lectura de la pipe 2, por lo que cuando mute para ejecutar el cat leerá lo que reciba por esa pipe y escribirá ese contenido en fichero_salida.

```
1.  main() {
2.  int fd_pipe1[2];
3.  int fd_pipe2[2];
4.  int ret, i;
5.  char buf[80];
6.      pipe(fd_pipe1);
7.      ret = fork();
8.      if (ret > 0){
9.          dup2(fd_pipe1[1],1); close(fd_pipe1[1]); close(fd_pipe1[0]);
10.         execlp("cat","cat",(char *) 0);
11.     } else {
12.         close(fd_pipe1[1]);
13.         pipe(fd_pipe2);
14.         ret = fork();
15.
16.         if (ret > 0) {
17.             dup2(fd_pipe1[0],0); close(fd_pipe1[0]);
18.             dup2(fd_pipe2[1],1); close(fd_pipe2[1]); close(fd_pipe2[0]);
19.             execlp("cat","cat",(char *) 0);
20.
21.         } else {
22.             close(fd_pipe1[0]);
23.             dup2(fd_pipe2[0],0), close (fd_pipe2[0]); close(fd_pipe2[1]);
24.             execlp("cat", "cat", (char *) 0);
25.
26.         }
27.     }
28.
29.     write (2, "Fin", 3);
30. }
```

*Figura 5 Código de prog*

2. Completa la siguiente figura con toda la información necesaria para representar el estado de las tablas de canales de todos los procesos, la tabla de ficheros abiertos y la tabla de inodes, suponiendo que todos los procesos están justo antes de acabar la ejecución (nota: en el campo inode de la tabla i-nodes puedes poner algo que represente al i-node del dispositivo).

Tabla Canales

| | Ent. TFA |
|---|---|
| 0 | 1 |
| 1 | 4 |
| 2 | 0 |
| 3 | |
| 4 | |
| | |
| | |

Tabla Ficheros Abiertos

| | #refs | Mod | Punt l/e | Ent t. inodes |
|---|---|---|---|---|
| 0 | 3 | rw | -- | 0 |
| 1 | 1 | r | end | 1 |
| 2 | 1 | w | end | 2 |
| 3 | 1 | r | -- | 3 |
| 4 | 1 | w | -- | 3 |
| 5 | 1 | r | -- | 4 |
| 6 | 1 | w | -- | 4 |

Tabla i-nodes

| | #refs | inode |
|---|---|---|
| 0 | 1 | i-tty |
| 1 | 1 | i-fichEnt |
| 2 | 1 | i_fichSal |
| 3 | 2 | i_pipe1 |
| 4 | 2 | i_pipe2 |
| | | |
| | | |

Tabla Canales

| | |
|---|---|
| 0 | 3 |
| 1 | 6 |
| 2 | 0 |
| | |
| | |
| | |

Tabla Canales

| | |
|---|---|
| 0 | 5 |
| 1 | 2 |
| 2 | 0 |
| | |
| | |
| | |

3. ¿Qué contendrá fichero_salida al final de la ejecución? ¿Qué mensajes aparecerán en pantalla?

Fichero_salida contiene una copia exacta de fichero_entrada. El proceso inicial lee fichero entrada y escribe su contenido en la primera pipe. El proceso hijo 1 lee de esa pipe todo ese contenido y lo escribe en la segunda pipe. Por último el proceso hijo 1.1. lee de esa segunda pipe todo el contenido y lo escribe en fichero_salida. En pantalla no aparece ningún mensaje porque la única línea que intenta escribir en terminal es la 29 (escribe en salida de errores estándar que está asociada al terminal) pero ningún proceso la llega a ejecutar porque todos mutan antes.

4. ¿Acabarán la ejecución todos los procesos?

Sí que acaban todos. La única situación que podría provocar que un proceso no acabara es que no acabara el bucle de lectura que ejecuta el comando cat. En cuanto un proceso acaba de escribir en la pipe muere y por lo tanto se cierran todos sus canales. Y ningún proceso se deja abierto ningún canal de escritura en la pipe.

5. (*T2)* Queremos modificar el código para que cada cierto tiempo (por ejemplo cada segundo) aparezca un mensaje en la pantalla. Se nos ocurre la siguiente modificación del código (los cambios aparecen marcados en negrita):

```
1.  int trat_sigarlm(int signum) {
2.  char buf[80];
3.     strcpy(buf, "Ha llegado alarma!");
4.     write(2,buf,strlen(buf));
5.     alarm(1);
6.  }
7.  main() {
8.  int fd_pipe1[2];
9.  int fd_pipe2[2];
10. int ret, i;
11. char buf[80];
12. struct sigaction trat;
13.    trat.sa_flags = 0;
14.    trat.sa_handler = trat_sigalrm;
15.    sigemtpyset(&trat.sa_mask);
16.    sigaction(SIGARLM ,trat_sigalrm,NULL);
17.    alarm(1);
18.    pipe(fd_pipe1);
19.    ret = fork();
20.    // A PARTIR DE AQUI NO CAMBIA NADA
21.    //...
```

Suponiendo que el proceso inicial tarda más de 1 segundo en acabar, ¿funcionará? ¿Qué mensajes aparecerán ahora por pantalla? ¿Qué procesos los escribirán? ¿Cambiará en algo el resultado de la ejecución?

No funcionará porque al mutar para ejecutar el cat perdemos la reprogramación de la alarma. El único proceso que recibe la alarma es el inicial, si recibiera alguna alarma antes de mutar escribiría el mensaje "ha llegado alarma" y continuaría la ejecución. Pero en cuanto reciba una alarma una vez mutado se ejecutará el tratamiento por defecto de la alarma y el proceso morirá. El fichero de salida contendrá la parte del fichero de entrada que hayamos tenido tiempo de transmitir.

# Ejercicio 21. (T2)

1. ¿Qué diferencia hay entre un signal bloqueado y un signal ignorado? ¿Qué llamada a sistema hay que ejecutar para bloquear un signal? ¿Y para ignorarlo?

   Un signal bloqueado no se recibe hasta que el proceso lo desbloquee y entonces ejecutará el tratamiento que el signal tenga asociado. Para bloquear signals hay que ejecutar la llamada sigprocmask. También es posible bloquearlos temporalmente durante la ejecución de un tratamiento de signal y durante un sigsuspend. Un signal ignorado se recibe pero el tratamiento asociado es no hacer nada. Para ignorarlo hay que usar la constante SIG_IGN como handler que se pasa al sigaction.

2. ¿En qué consiste la atomicidad de la llamada a sistema sigsuspend? Explícalo con un ejemplo.

Al sigsuspend se le pasa la máscara de signals bloqueados que queremos utilizar mientras se está en el sigsuspend. En esa máscara se bloquean todos los signals excepto el que queremos que nos saque del bloqueo y es necesario que los signals que nos van a sacar del bloqueo estén bloqueados fuera del sigsuspend (si no podrían llegar antes y quedarnos para siempre en el sigsuspend). Y es necesario que la activación de esa máscara sea una operación atómica para evitar justamente la situación de que llegue el signal que queremos esperar dentro del sigsuspend pero antes de bloquear al proceso.

3. Indica qué signals hay bloqueados en los puntos de ejecución A, B, C, D, E, F, G, H y I.

```
1.  void sigusr1(int signum)
2.  { sigset_t mascara;
3.    /* C */
4.    sigemptyset(&mascara);
5.    sigaddset(&mascara, SIGINT); sigaddset(&mascara, SIGUSR1);
6.    sigprocmask(SIG_BLOCK, &mascara, NULL);
7.    /* D*/
8.  }
9.  void sigusr2(int signum)
10. { /* B */
11.   kill(getpid(), SIGUSR1);
12. }
13. void sigalrm(int signum)
14. { /* H*/
15. }
16. main()
17. { sigset_t mascara;
18.   struct sigaction new;
19.
20.   new.sa_handler = sigusr1; new.sa_flags = 0;
21.   sigemptyset(&new.sa_mask); sigaddset(&new.sa_mask, SIGALRM);
22.   sigaction(SIGUSR1, &new, NULL);
23.
24.   new.sa_handler = sigalrm; sigemptyset(&new.sa_mask);
25.   sigaction(SIGALRM, &new, NULL);
26.
27.   new.sa_handler = sigusr2;
28.   sigemptyset(&new.sa_mask); sigaddset(&new.sa_mask, SIGPIPE);
29.   sigaction(SIGUSR2, &new, NULL);
30.
31.   /* A */
32.   kill(getpid(), SIGUSR2);
33.   /* E */
34.   sigemptyset(&mascara); sigaddset(&mascara, SIGALRM);
35.   sigprocmask(SIG_BLOCK, &mascara, NULL);
36.   /* F */
37.   sigfillset(&mascara); sigdelset(&mascara, SIGALRM);
38.   alarm(2);
39.   /* G */
40.   sigsuspend(&mascara);
41.   /* I */
42. }
43.
```

| Punto ejecución | Máscara de signals |
|---|---|
| A | Vacía (o la heredara del padre) |
| B | SIGUSR2 y SIGPIPE |
| C | SIGUSR2, SIGPIPE, SIGUSR1,SIGALRM |
| D | SIGUSR2, SIGPIPE, SIGUSR1,SIGALRM,SIGINT |
| E | La misma que en A (al acabar la ejecución de la rutina de atención al signal, se restaura el máscara con el valor que había al recibirlo). |
| F | SIGALRM |
| G | SIGALRM |
| H | Llena |
| I | SIGALRM |