Para las sesiones de problemas

Sistemas Operativos
Grau en Enginyeria Informàtica

Este documento contiene ejercicios que se solucionaran en las sesiones de problemas marcadas en la planificación

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

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
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).
    b. *(T2)* Qué procesos ejecutarán las líneas 36+37
    c. *(T3)* Qué procesos ejecutarán la función realizatarea
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)
    b. El espacio físico que necesitaremos para ejecutar las dos ejecuciones descritas en el apartado 1

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

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:
    b. Para cada proceso, indica qué signals recibirá:
    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?
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:
    b. Para cada proceso, indica qué signals recibirá:
    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?

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		
y		
vector_int		
i		
buffer		

2. Justifica por qué no aparece la región del heap en este instante de la ejecución del proceso (PUNTO A)
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?
   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?
4. Indica cómo serían las líneas 7 y 11 si quisiéramos hacer este mismo programa utilizando directamente la(s) llamada(s) a sistema correspondiente.

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

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?

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.

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?

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.

```
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.
```

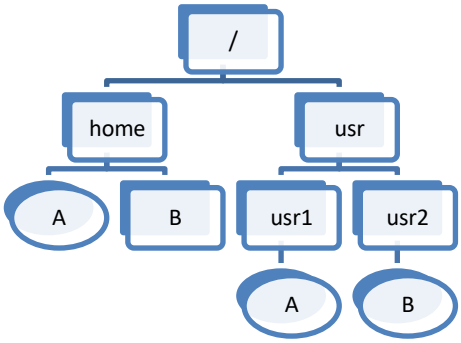
Figura 3 Código del programa prog

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?
2) (T5) ¿Qué es una Race condition? ¿Qué es una region crítica? ¿Qué relación hay entre los dos conceptos?
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)
4) (T5) Enumera y explica brevemente qué ventajas tiene utilizar threads respecto a utilizar procesos.

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 dibujo de la jerarquía, 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											
Tipo_fichero											
Tabla de indices											

Bloques datos

Num Bloque Datos	0	1	2	3	4	5	6	7	8

Num Bloque Datos	9	10	11	12	13	14	15	16	17

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

   b) ¿Cuál es el path relativo para referenciar el fichero /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

b) Indica exactamente qué inodos y bloques de datos accede la línea 3.

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

d) ¿Qué bloques de datos accede la línea 7?

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

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.

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.

3. ¿Qué procesos acabarán la ejecución?

```
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

4. 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	
2	

Tabla Ficheros Abiertos

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

Tabla i-nodes

	#refs	inode
0	1	i-tty

5. 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

Tabla Ficheros Abiertos

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

Tabla i-nodes

	#refs	inode
0	1	i-tty

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

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?

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?

Ejercicio 10. (T4,T5)

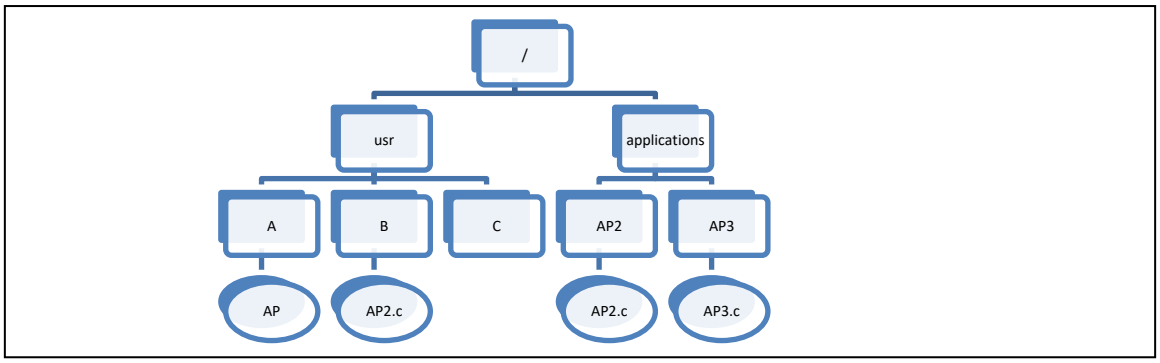
1. *(T4)* Explica brevemente que efecto tiene, en las tablas de gestión de entrada/salida, la ejecución de un fork y un exit (explícalos por separado).
2. *(T4)* Enumera las llamadas a sistema de entrada/salida que pueden generar nuevas entradas en la tabla de canales
3. *(T4)* Tenemos el siguiente código 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);
```

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).

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 de la jerarquía.
   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											
Tipo_fichero											
Tabla de indices											

Bloques datos

Num Bloque	0	1	2	3	4	5	6	7	8
Contenido									

Num Bloque	9	10	11	12	13	14	15	16	17
Contenido									

2. Indica cuántos accesos a bloques e inodos (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.

Aclaración:

- El open accede a los inodos necesarios hasta llegar al inodo destino, pero no lee ningún bloque de datos del fichero destino
- 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 mínimo al cerrar el fichero.
- El lseek no genera accesos a bloques de datos ni 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 bloques e inodos (cuáles)	Tablas Modificadas (SI/NO)		
		Canales	F. Abiertos	Inodos
1				
2				
3				
4				
5				
6				
7				

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

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

Líneas 18 y 19:   Líneas 21 y 23:

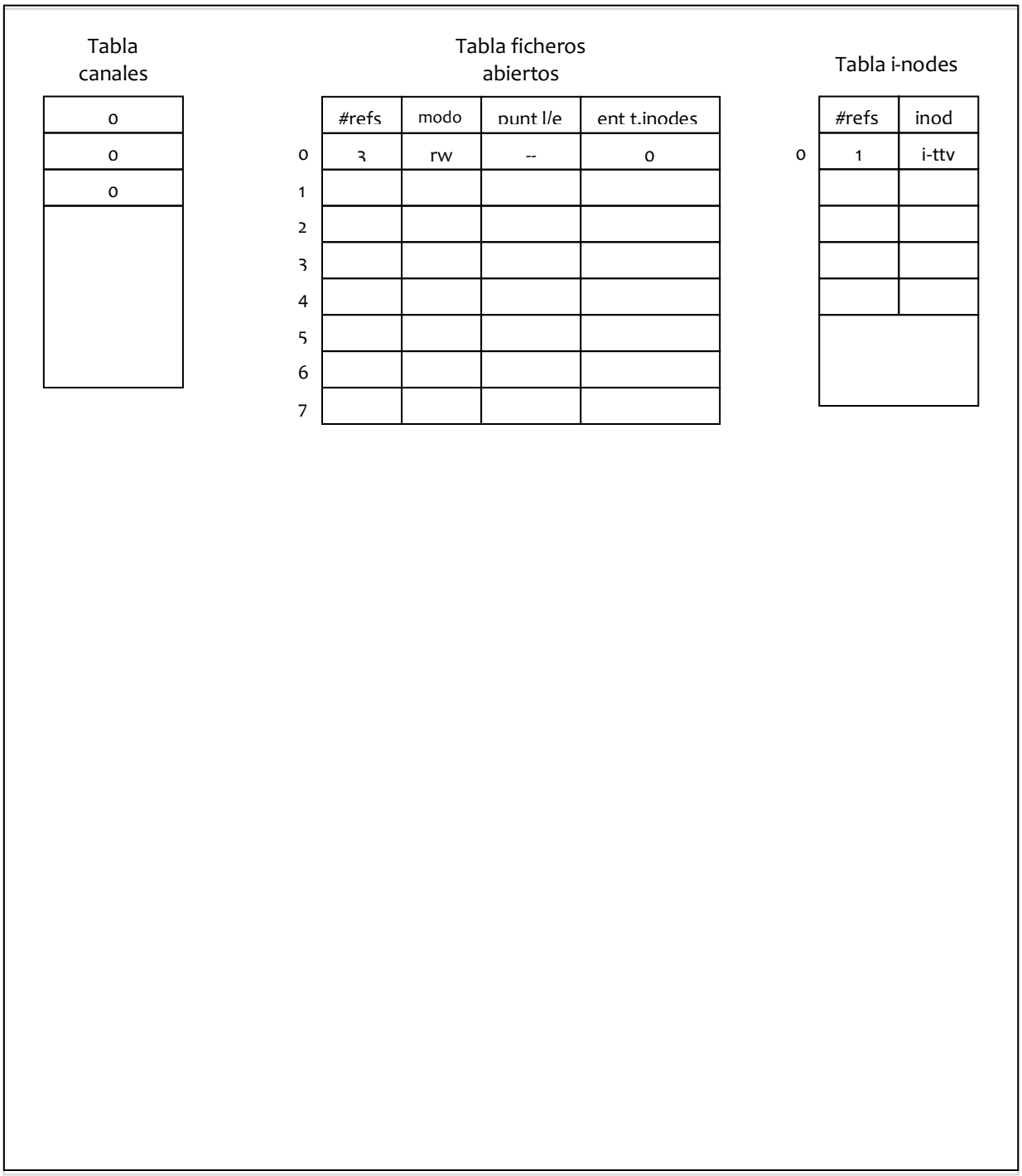
   c) ¿Es necesario añadir algún código para sincronizar a los procesos en el acceso a la(s) pipe(s)?

   d) ¿Qué mensajes aparecerán en la pantalla?

   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.

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)

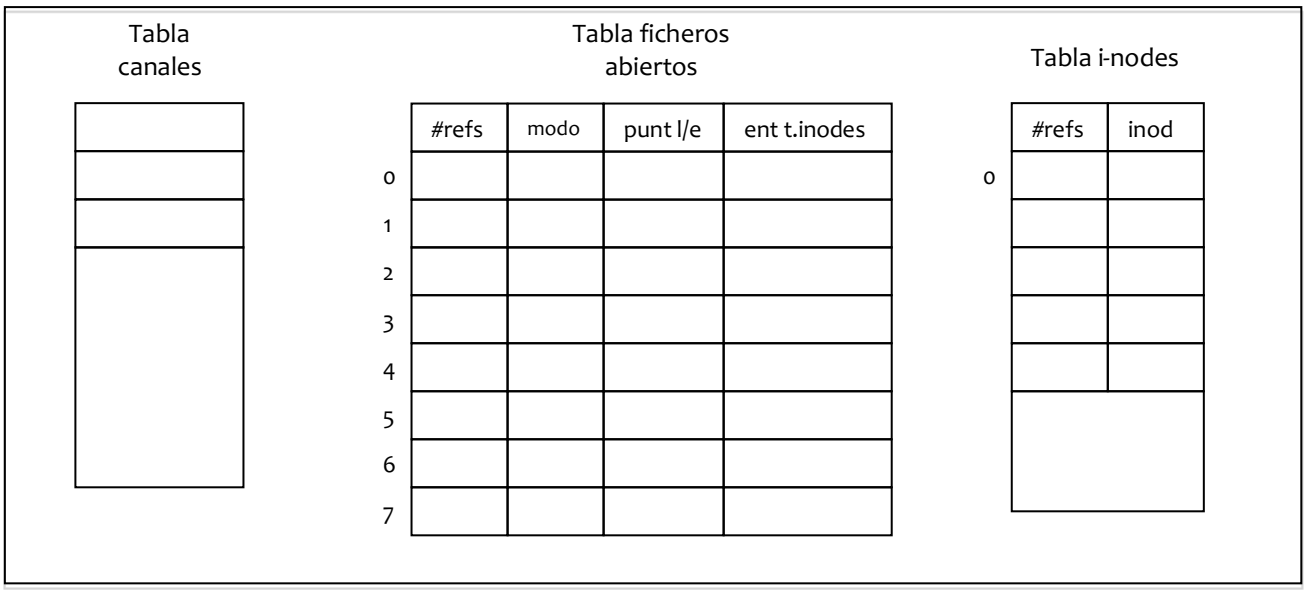


3. Modificamos su comportamiento

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?

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.

Ejercicio 13. (T3)

1. ¿Cuáles son las dos principales funcionalidades de la Memory Management Unit (MMU)? (indica cuáles son y en qué consisten)
2. Explica qué aporta la optimización de la memoria virtual respecto a tener simplemente paginación y carga bajo demanda

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

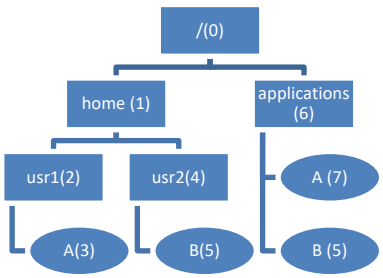
```
void f(int s)
{ }
void main()
{
struct sigaction trat;
sigset_t mask;
trat.sa_flags = 0;
trat.sa_handler = f;
sigemptyset(&trat.sa_mask);
sigaction(SIGUSR1,&trat, NULL);
…
sigfillset(&mask);
sigdelset(&mask, SIGUSR1);
sigsuspend();
…
}
```

Contesta a las siguientes preguntas:

1. ¿Cuál de los dos códigos corresponde a una espera activa?
2. ¿Es necesaria la reprogramación del signal SIGUSR1 en el código 2? ¿Qué pasaría si no lo hacemos?
3. ¿En qué región de memoria podremos encontrar la variable “recibido”?
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).

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?
2. ¿Cuántos y cuáles accesos a inodos y bloques 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).
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).

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?

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?

6. ¿Qué mensajes aparecerán en el terminal?

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

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?

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

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?

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?
3. *(T3)* Explica brevemente para que se utiliza el área de swap cuando aplicamos el algoritmo de reemplazo en la optimización de memoria virtual.
4. *(T4)* ¿De qué tipo son los ficheros . y .. que encontramos en los directorios y a que hacen referencia?

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();
```

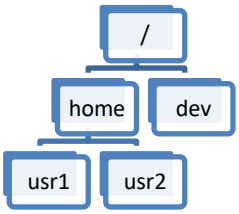
1. El sistema no implementa COW y todos los procesos están justo después del último fork
2. El sistema implementa COW y todos los procesos están justo después del último fork

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.
```

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)

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?

3. ¿Cuántas entradas en la Tabla de ficheros abiertos generará este código?
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?
6. ¿Cuántos accesos a bloques de datos e inodos se producirán en total como consecuencia del bucle de la línea 6?
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											
Tipo_fichero											
Tabla de índices											

Bloques datos

Num Bloque Datos	0	1	2	3	4	5	6	7	8

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).

```
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	
2	

Tabla Ficheros Abiertos

	#refs	Mod	Punt l/e	Ent t. inodes
0				0
1				
2				
3				
4				
5				
6				

Tabla i-nodes

	#refs	inode
0	1	i-tty

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

4. ¿Acabarán la ejecución todos los procesos?

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?

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

Ejercicio 22. (T4)

Supongamos que estamos usando un sistema de ficheros basado en inodos, con un tamaño de bloque de 1024Bytes. En el directorio actual de trabajo tenemos el fichero f1, en la siguiente figura se muestra parte del contenido de su inodo:

Número de inodo	7
	Tipo: fichero Tamaño: 4000 Bytes Tabla de índices: 8 9 15 21

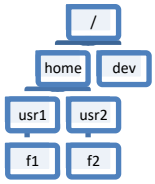
Ejecutamos el siguiente código:

```
1. char buf[1024];
2. int fd,ret;
3. fd=open("f1",O_RDONLY);
4. lseek(fd,1000, SEEK_SET);
5. ret=read(fd,buf,sizeof(buf);
```

Indica qué inodos y bloques de datos accederán las líneas 3 y 4

Ejercicio 23. (T4)

Supongamos que estamos usando un sistema de ficheros basado en inodos, con un tamaño de bloque de 1024Bytes. Tenemos la siguiente jerarquía de directorios.



Nos dicen que f2 es un soft link a f1, f1 es un fichero de datos, y el resto de componentes son directorios. La siguiente figura representa los bloques de datos y metadatos de la jerarquía de directorios

Num_inodo	2	1	2	3	4	5	6	7	8	9	10
#enlaces											
Tipo_fichero											
Tabla de											

índices											

Num Bloque	0	1	2	3	4	5	6	7	8
contenido									

Ejecutamos el siguiente código:

```
1. char buf[2048];
2. int fd,ret;
3. fd=open("/home/usr2/f2",O_RDONLY);
4. lseek(fd,1000, SEEK_SET);
5. ret=read(fd,buf,sizeof(buf);
```

1. ¿Qué inodos y bloques de datos accederá la línea 5?
2. ¿Cuál será el valor de la variable ret, después de ejecutar la línea 5?