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.

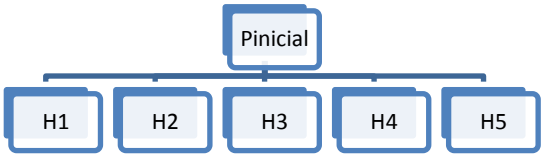
El thrashing es la situación en la que el sistema invierte más tiempo haciendo intercambio de páginas que en ejecutar código de los procesos. Veríamos aumentar claramente el tiempo de sistema ya que el intercambio de páginas lo hace el kernel.

Ejercicio 2. (T2, T3)

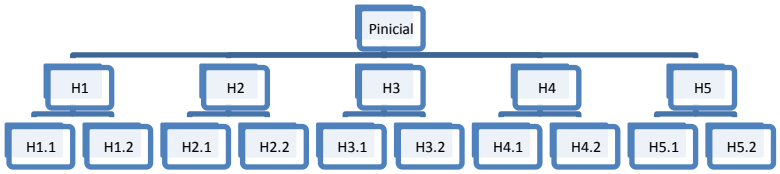
La Figura 1 contiene el código del programa *Mtarea* (se omite el control de errores para simplificar y podemos asumir que ninguna llamada a sistema devuelve error).

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

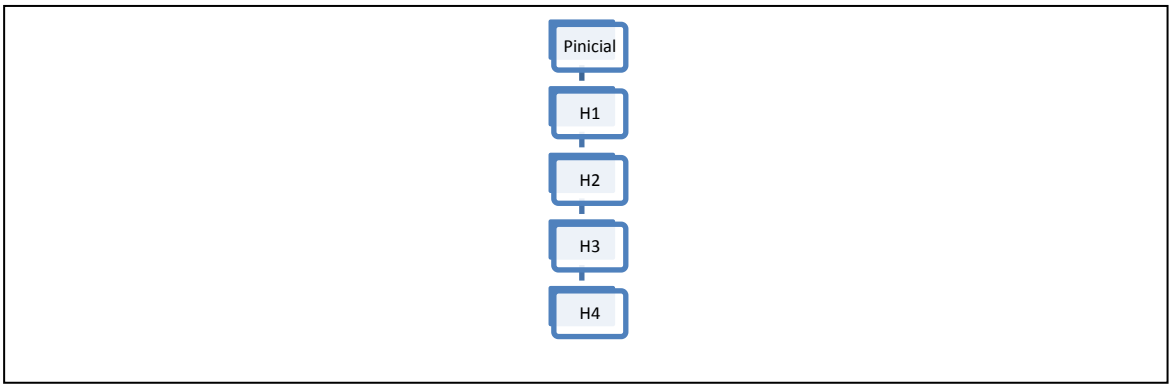
Ejercicio 3. (T2)

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 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.

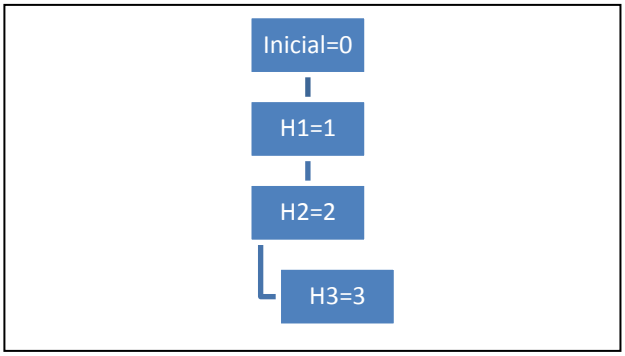
   Si termina, que pasara a zombie, y si se bloquea, que pasará a "bloqueado" o un estado similar

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: 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. (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 8. (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 9. (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