Nombre alumno:
DNI:

Examen final de teoría de SO

Justifica todas tus respuestas. Las respuestas sin justificar se considerarán erróneas.

Gestión de procesos (*3 puntos*)

La figura 1 muestra el código del programa examen (se omite la gestión de errores para facilitar la legibilidad del código).

```
1.  void trata_signal(int s) {
2.  }
3.  main() {
4.  int i,ret;
5.  struct sigaction trat;
6.  sigset_t mask;
7.  char buf[80];
8.  int pidh[10];
9.
10. sigemptyset(&mask);
11. sigaddset(&mask,SIGUSR1);
12. sigprocmask(SIG_BLOCK,&mask,NULL);
13.
14. sigemptyset(&trat.sa_mask);
15. trat.sa_flags=0;
16. trat.sa_handler = trata_signal;
17. sigaction(SIGUSR1, &trat, NULL);
18.
19. i=0;
20. ret=1;
21. while ((i<3) && (ret > 0)) {
22.     ret=fork();
23.     pidh[i]=ret;
24.     i++;
25. }
26. if (ret==0) {
27.     sigfillset(&mask);
28.     sigdelset(&mask,SIGUSR1);
29.     sigsuspend(&mask);
30. } else {
31.     for (i=2;i>=0;i--) {
32.         kill(pidh[i],SIGUSR1);
33.         waitpid(-1,NULL,0);
34.     }
35. }
36. sprintf(buf, "Soy el proceso %d\n",getpid());
37. write(1,buf,strlen(buf));
38. }
```

figura 1 Programa examen

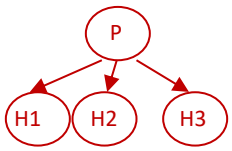
Ponemos en ejecución este código con el siguiente comando:

% ./examen

Suponiendo que todas las llamadas a sistema se ejecutan sin errores y que los únicos signals involucrados con la ejecución del proceso son los generados por el propio código, contesta razonadamente a las siguientes preguntas.

a) ***(0,5 puntos)*** Representa la jerarquía de procesos que genera la ejecución de examen. Asigna a cada proceso un identificador para poder referirte a ellos en el resto de los ejercicios.

Nombre alumno:
DNI:



b) ***(0,5 puntos)*** ¿Qué procesos mostrarán el mensaje de la línea 37?

Todos los procesos.

c) ***(0,5 puntos)*** ¿Se puede saber en qué orden acabarán los procesos? ¿Se puede garantizar que será siempre el mismo orden para todas las ejecuciones?

El orden será los hijos en orden inverso de creación y por último el padre. Siempre será el mismo porque lo garantizan las operaciones de sincronización: cada hijo espera en el sigsuspend su turno. El padre va pasando el turno en orden inverso a la creació y espera a que acabe cada hijo antes de pasarle el turno al siguiente hijo.

d) ***(0,75 puntos)*** ¿Es necesaria la ejecución del sigprocmask de la línea 12? ¿Podemos quitarlo y garantizar que el resultado seguirá siendo siempre el mismo?

Es necesario, porque si lo quitamos algún proceso hijo podría recibir el SIGUSR1 antes de ejecutar el sigsuspend y se quedaría para siempre bloqueado.

e) ***(0,75 puntos)*** Supón que un proceso que está ejecutando la línea del waitpid recibe en ese momento un SIGUSR1 que el usuario le envía con un comando del Shell. ¿Podría afectar de alguna manera a la salida que veríamos en pantalla?

No, porque el único proceso que ejecuta esa línea es el padre que tiene el SIGUSR1 bloqueado. Así que ese SIGUSR1 no cambiaría el comportamiento de la ejecución

Nombre alumno:
DNI:

Gestión de memoria *(2 puntos)*

Tenemos una máquina con un procesador Intel, que implementa un sistema de gestión de memoria basado en paginación con tamaño de página 4KB. El sistema operativo de esta máquina es Linux. Ponemos en ejecución un programa cuyo espacio de direcciones tiene las siguientes regiones: región de código de 1KB, región de pila 1KB y región de datos 2KB. Suponiendo que el proceso está cargado por completo en memoria, contesta razonadamente a las siguientes preguntas

a) ***(0,5 puntos)*** ¿Cuánta memoria física ocupa?

3 páginas (12KB) porque la unidad de asignación es la página, y dos regiones no pueden compartir páginas.

b) ***(0,5 puntos)*** ¿Este proceso tiene fragmentación de memoria? Si la repuesta es que sí, entonces di de qué tipo y la cantidad de memoria que se pierde por la fragmentación. Si la respuesta es que no, justifica el motivo.

Sí, porque debido al tamaño de la unidad de asignación, le estamos asignando más memoria de la que realmente ocupa. Es fragmentación interna y la cantidad de memoria perdida es 8KB.

c) ***(0,5 puntos)*** Supón que este proceso ejecuta la llamada a sistema *fork*. La ejecución de esta llamada (sin tener en cuenta la ejecución de ninguna instrucción posterior), ¿provocará que se reserve alguna cantidad de memoria física? Si la respuesta es que sí, indica cuánta memoria adicional se reservará. En cualquier caso, justifica tu respuesta.

Fork no reserva memoria física porque Linux implementa copy-on-write. Tanto padre como hijo compartirán la memoria mientras accedan sólo de lectura.

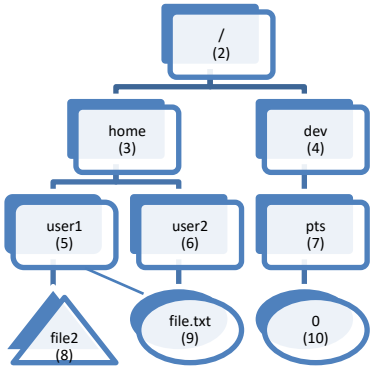
d) ***(0,5 puntos)*** Supón que este proceso en lugar de la llamada *fork* ejecuta la llamada a sistema *exec* para mutar al mismo ejecutable (a sí mismo). La ejecución de esta llamada (sin tener en cuenta la ejecución de ninguna instrucción posterior), ¿provocará que cambie la cantidad de memoria física? Si la respuesta es que sí, indica cómo cambiará. En cualquier caso, justifica tu respuesta.

La llamada a sistema exec provoca que se libere la memoria del proceso y que se reserve memoria para el programa que se quiere cargar. Como Linux implementa carga bajo demanda se irá reservando memoria a medida que se referencie. Tendremos inicialmente 1 página para el código y 1 página para la pila, mientras que la página de datos se reservará cuando se acceda por primera vez a alguna variable global.

Nombre alumno:
DNI:

Sistema de Ficheros *(2 Puntos)*

Tenemos el siguiente SF basado en I-Nodos, con un tamaño de bloque de 1KB. Representamos los ficheros regulares con círculo, los directorios con rectángulo y los soft-links con triángulo. El fichero "file.txt" es un fichero de caracteres que contiene 2KB de datos. El fichero "0" representa el terminal que tenemos abierto y que está en uso. Por último, el fichero "file2" es un soft-link que apunta al terminal "0" mediante un path absoluto.



a) ***(1 punto)*** Rellena las siguientes tablas, asignando I-nodos y Bloques de Datos de forma ordenada. En el campo tipo puedes usar "-", "dir", "link" para representar fichero regular, directorio y soft-link, respectivamente. Puedes assumir que el terminal ("0") no tiene BDs asignados.

Listado de Inodos											
ID Inodo	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
#enlaces	4	4	3	2	2	2	1	2	1		
Tipo	dir	dir	dir	dir	dir	dir	link	-	-		
BDs	0	1	2	3	4	5	6	7, 8	-		

ID BD	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Datos	. 2 .. 2 home 3 dev 4	. 3 .. 2 user1 5 user2 6	. 4 .. 2 pts 7	. 5 .. 3 file2 8 file.txt 9	. 6 .. 3 file.txt 9	. 7 .. 4 0 10	/dev/pts/0	DATA	DATA		

b) ***(0,5 puntos)*** Abrimos otra terminal y, por tanto, se crea otro fichero en el directorio "dev", pero esta vez con el nombre "1". Indica qué campo/s del I-nodo da/n la información necesaria para identificar de manera única las dos terminales que tenemos en la carpeta "dev".

Nombre alumno:
DNI:

Tipo de transferencia (block/char), major (qué tipo de dispositivo) y minor (qué instancia de ese tipo).

Desarrollamos un programa ("prog") con el siguiente código (obviamos el control de errores):

```
1. char c;
2. int fd = open("/home/user1/out.txt", O_WRONLY|O_CREAT|O_TRUNC, 0640);
3. while(read(0, &c, 1) > 0){
4.     write(fd, &c, 1);
5.     lseek(0, 1, SEEK_CUR);
6. }
```

c) ***(0,5 puntos)*** Indica qué cambios reflejará la ejecución de este código en las tablas anteriores si lo lanzamos a ejecutar con la línea de comandos:

./prog < /home/user1/file.txt

Actualizar contenido BD del directorio "student" para añadir la nueva entrada. Un nuevo Inodo que tiene asociado 1 BD (fichero resultante).

Nombre alumno:
DNI:

Pipes *(3 Puntos)*

Tenemos el siguiente código:

```
1. int num;
2. while(read(0, &num, 4) > 0){
3.     write(2, &num, 4);
4. }
          /* endpoint.c */
```

Queremos establecer una comunicación bidireccional entre 2 programas que ejecutan endpoint. Lo haremos mediante 2 pipes con nombre ("PIPE1" y "PIPE2"). Asumimos que las dos pipes ya existen.

```
1. main(){
2.    int npd;
3.    int num = getpid();
4.    if(fork()==0){//Proceso Hijo que lee de PIPE1 y escribe en PIPE2
5.       execlp("./endpoint", "./endpoint", 0);
6.    }
7.    if(fork()==0){//Proceso Hijo que escribe en PIPE1 y lee de PIPE2
8.       execlp("./endpoint", "./endpoint", 0);
9.    }
10.   npd = open("PIPE1", O_WRONLY);
11.   write(npd, &num, sizeof(int));
12.   close(npd);
13.   while(waitpid(-1, null, 0)>0);
14. }
                                /* programa.c */
```

a) ***(1 punto)*** Indica qué cambios harías en "programa.c" para preparar la comunicación entre los dos procesos hijo usando las dos pipes. Hazlo mediante el siguiente formato, indicando las líneas de código exactas que introducirías:

"entre las líneas X-Y poner el código: …"

Entre 4-5:
close(0); open("PIPE1", O_RDONLY);
close(2); open("PIPE2", O_WRONLY);

Entre 7-8:
close(2); open("PIPE1", O_WRONLY);
close(0); open("PIPE2", O_RDONLY);

b) ***(0,5 puntos)*** ¿Qué finalidad tiene la línia de código 11?

Activar la comunicación

Nombre alumno:
DNI:

c) ***(0,5 puntos)*** Si matamos al proceso padre justo **antes de ejecutar** la línea 12, ¿finalizaría la comunicación y, por tanto, la ejecución de los procesos "endpoint"? ¿y si matamos al segundo hijo cuando el padre **ya ha ejecutado** la línea 12, qué sucedería?

Si matamos al proceso padre no sucede nada, ya que el proceso hijo 2 es escritor en la pipe (igual que el padre). Por tanto, habrá una ejecución infinita.

Si matamos el segundo hijo se rompen las pipes de tal forma que el proceso hijo 1, si está en el read, sale del bucle porque ya no hay escritores en la PIPE1. Mientras que si está en el write, recibirá un SIGPIPE que lo matará porque ya no hay lectores en la PIPE2. A continuación el proceso padre ejecutará las dos iteraciones del while(waitpid)

d) ***(1 punto)*** Indica qué tablas se accede y/o modifica al ejecutar las llamadas al sistema de los códigos de éste. En cada caso, si se hace una modificación, indica brevemente qué se ha modificado.

	Llamada al sistema	Tablas Modificadas (SI/NO)			Breve razonamiento de la/s modificación/es
		Canales (F. Descriptors)	F. Abiertos	Inodos	
1	fork()	SI	SI	NO	Se crea una nueva tabla de canales (copia del padre) y se actualizan las #refs de las entradas de la TFO
2	open("PIPE1", O_WRONLY)	SI	SI	SI	Nueva entrada en la T. Canales, en TFO y posible nueva entrada en T.Inodo o actualizar #refs de una entrada
3	execlp("./endpoint", "./endpoint", 0)	NO	NO	NO	
4	close(npd)	SI	SI	SI (depende)	Se libera entrada en T.Canales y en TFO y en la T.Inodos (si #refs llega a cero)
5	waitpid(-1, NULL, 0)	NO	NO	NO	