Nombre alumno:
DNI:

Primer parcial de teoría de SO

Justifica todas tus respuestas de este examen. Cualquier respuesta sin justificar se considerará errónea.

Preguntas cortas (2 *puntos*)

a) ***(0,5 puntos)*** ¿Por qué la librería de sistema tiene que ser dependiente del HW?

La librería de sistema implementa las llamadas a sistema. Estas invocan a los servicios de kernel alojados en el núcleo del sistema operativo, que se ejecuta en modo privilegiado. Para realizar esta invocación al servicio de kernel, pasarle parámetros y cambiar de modo es necesario invocar a una instrucción específica proporcionada por el hardware (syscall, sysenter, int,…)

b) ***(0,5 puntos)*** Suponiendo que el sistema funcione bien y las llamadas a sistema no dan errores. ¿Qué efectos tiene la ejecución del fork sobre las estructuras de datos de gestión de signals del proceso padre?

Ningún efecto sobre ninguna estructura (Bitmap de signals pendientes, vector de captura, máscara de signals pendientes, temporizador)

c) ***(0,5 puntos)*** Un proceso huérfano recibe un SIGKILL, ¿es posible que se quede en estado Zombie para siempre?

No, el waitpid que libera el estado Zombie lo hará el padre del proceso que recibe el SIGKILL, al ser huérfano su proceso padre pasa a ser el proceso con PID=1

d) ***(0,5 puntos)*** Sea el siguiente código:

Nombre alumno:
DNI:

```
…
sigset_t mask;
sigemptyset(&mask);
sigsuspend(&mask);
…
```

¿Existe algún caso en que **sigsuspend(…)** no se desbloquee al recibir un signal?

Sí, cuando reciba algún signal que esté ignorado

Nombre alumno:
DNI:

Gestión de procesos (3,75 *puntos*)

La figura 1 muestra el código del programa jerarquia.c (se omite el control de errores para facilitar la legibilidad del código):

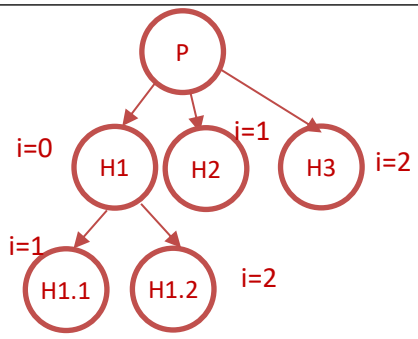
```
1. /* jerarquia */
2.
3. main(int argc, char *argv[]) {
4. int i,ret,nit,nhijos,st;
5. char buf[80];
6.
7. nit = atoi(argv[1]);
8. for (i=0; i<nit;i++){
9.         ret = fork();
10.        if (ret == 0) {
11.                if (i%2 != 0) {
12.                        execlp("ls","ls",(char *)0);
13.                }
14.        }
15.  }
16.  nhijos=0;
17.  while (waitpid(-1,&st,0) > 0) {
18.          nhijos++;
19.  }
20.  sprintf (buf, "Soy %d y he liberado %d hijos\n", getpid(), nhijos);
21.  write (1, buf, strlen(buf));
22.  }
```

figura 1: Código de jerarquia.c

Ponemos el programa en ejecución con el siguiente comando: ./jerarquia 3

Suponiendo que fork, write y execlp se ejecutan sin devolver ningún error, contesta a las siguientes preguntas de manera razonada.

a) ***(1,25 punto)*** Dibuja la jerarquía de procesos que se genera al ejecutar el comando. En el dibujo asigna un identificador a cada proceso para las preguntas posteriores. Indica para cada proceso (excepto para el inicial) qué valor tiene la variable **i** cuando empiezan la ejecución.



Id de proceso	Valor de i
H1	0
H2	1
H3	2
H1.1	1
H1.2	2

Justificación: el bucle que empieza en la línea 8 acaba cuando i vale 2. Dentro del bucle se crea un proceso y el hijo. El padre continúa iterando (y creando hijos) hasta que su i vale 2. El hijo mutará si su i tiene un valor impar (dejará de iterar) y en caso contrario continuará iterando (hasta que su i valga 2) y creará hijos. Cada proceso empieza la ejecución con el valor de i que tuviera su padre cuando le creó.

Nombre alumno:
DNI:

b) ***(1 punto)*** ¿Qué procesos imprimen el mensaje de la línea 21? Para cada uno indica qué valor tiene la variable nhijos.

Todos los procesos excepto los que pasan por la línea 14 y mutan. Es decir, P y los procesos que inician la ejecución con una i par: H1, H1.2, y H3

Id de proceso	Valor de nhijos en 21
P	3
H1	2
H1.2	0
H3	0

c) ***(1 punto)*** Sobre los procesos que pasan por la línea 21 ¿podemos saber cuál será el último proceso en escribir?

Siempre será P porque el waitpid de la línea 17 hace que hasta que sus hijos (H1 y H3) escriban y acaben él no lo hará. Además, H1 no escribirá y acabará hasta que H1.2 lo haga.

d) ***(0,5 puntos)*** ¿Cuál es el grado máximo de concurrencia que podemos tener?

6, porque entre la creación de dos procesos nunca hay ninguna espera hasta que alguno de los ya creados haya acabado.

Nombre alumno:
DNI:

Signals (4,25 *puntos*)

Las figuras 2.a y 2.b muestran los códigos de los programas signaling.c y loop.c (se omite el control de errores para facilitar la legibilidad del código):

```
1.  /* signaling.c */
2.  int cont=0;
3.  void func(int s){
4.        printf("start from signaling\n");
5.        if (cont == 0) {cont++; kill(getpid(), s);}
6.        printf("end from signaling\n");
7.  }
8.
9.  void main(int argc, char *argv[]) {
10.       int i, ret, status;
11.       struct sigaction sh;
12.
13.       sigemptyset(&sh.sa_mask);
14.       sh.sa_flags = 0;
15.       sh.sa_handler = func;
16.       for (i=1; i<16; i++)
17.           sigaction(i, &sh, NULL);
18.
19.       ret = fork();
20.       if (ret == 0) {
21.           alarm(2);
22.           execlp("./loop", "./loop", (char *)0);
23.       }
24.       alarm(1);
25.       waitpid(-1, &status, 0);
26. }
```

figura 2.a

```
1.  /* loop.c */
2.
3.  void func(int s){
4.        printf("start from loop\n");
5.        printf("end from loop\n");
6.  }
7.
8.  int main(int argc, char *argv[]) {
9.        while(1);
10.       return 0;
11. }
```

figura 2.b

Suponiendo que todas las llamadas se ejecutan sin devolver ningún error inesperado, contesta a las siguientes preguntas de manera razonada.

a) ***(0,5 puntos)*** Qué consecuencias implica la ejecución de las líneas 16-17 del código de la figura 2.a?

Cambia el tratamiento de todos los signals, excepto SIGKILL y SIGSTOP, porque están protegidas y no se puede cambiar su tratamiento.

b) **(0,75 puntos)** Qué mensajes veremos por pantalla tras más de 2 segundos de ejecución? Si ejecutamos varias veces el programa, siempre veremos los mensajes en el mismo orden?

start from signaling
end from signaling
start from signaling
end from signaling
El signal está bloqueado durante su tratamiento (se trata secuencial). Mientras que el hijo, al mutar, resetea el tratamiento y finaliza (tratamiento por defecto). Siempre veremos el mismo orden de los mensajes.

Nombre alumno:
DNI:

c) **(1 punto)** Si añadimos el siguiente código entre las líneas 18 y 19:

```
sigset_t mask;
sigfillset(&mask);
sigprocmask(&SIG_BLOCK, &mask, NULL);
```

indica de qué manera cambiarán las respuestas que has dado en el apartado anterior, así como en qué estado estarán los procesos "parent" y "child" tras recibir el signal SIGALRM.

Como SIGALRM está bloqueado, no se muestra ningún mensaje. El proceso "parent" estará en estado bloqueado (por waitpid) y el proceso "child" estará en estado "running".

d) **(1 punto)** Partiendo del código inicial, si cambiamos la línea 14 del código de la figura 2.a y ponemos "sh.sa_flags=SA_RESETHAND", indica de qué manera cambiarán las respuestas que has dado en el apartado "b", así como en qué estado estarán los procesos "parent" y "child" tras recibir el signal SIGALRM.

start from signaling
end from signaling

Debido al flag, se resetea el tratamiento y finaliza la ejecución del proceso "parent" al lanzar el segundo signal. El proceso "child" no cambia y finaliza al cabo de un segundo. Pasa al estado zombie y, como el padre ya no està, el proceso init lo libera.

e) **(1 punto)** Partiendo del código inicial, si quitamos la línea 21 y movemos la línea 24 a la línea 18, indica de qué manera cambiarán las respuestas que has dado en el apartado "b", así como en qué estado estarán los procesos "parent" y "child" tras recibir el signal SIGALRM.

El SIGALRM sólo lo recibe el proceso "parent" mostrando los mismos mensajes, y en el mismo orden, indicados en el apartado "b". Sin embargo, el proceso "child" no recibe ningún signal. Por lo que el proceso "parent" , al tener los sa_flags=0, dejará de estar bloqueado en el waitpid (que devolverá -1) y pasará a "running"y el proceso "child" estará en estado "running".