Nombre alumno:
DNI:

Examen final de teoría de SO

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

Preguntas cortas

1. *(0,5 puntos*) Explica brevemente qué operaciones realiza la siguiente llamada (asume que no hay optimizaciones de memoria):
   `int *ptr = malloc(2000);`

malloc() es la llamada de librería de lenguaje C para reservar memoria dinámicamente. La función busca en el heap actual si hay 2000 bytes contiguos:

- En caso afirmativo: asignará a ptr esos 2000 bytes
- En caso negativo: la librería pedirá al kernel que asigne al heap del proceso al menos 2000 bytes nuevos mediante la syscall sbrk(). De esta asignación del kernel la librería otorgará 2000 bytes a ptr.

2. (*0,5 puntos*) Un proceso lanza una dirección lógica válida pero la MMU no puede hacer la traducción. ¿A qué puede ser debido?

- El dato o la instrucción pueden estar en el área de intercambio (SWAP)
- A causa de la optimización de carga bajo demanda esa dirección no se ha cargado en memoria en el momento de la carga del ejecutable

Nombre alumno:
DNI:

3. (*0,5 puntos*) Un proceso recibe SIGPIPE. Indica todas las causas a que puede ser debido.

- Otro proceso (o él mismo) le ha enviado SIGPIPE usando la syscall o el comando "kill"
- Ese proceso ha intentado escribir en una pipe en la que no hay procesos lectores.

4. (*0,5 puntos*) En una política de planificación no apropiativa, ¿qué eventos hacen que un proceso pase de RUN a READY?

- Ningún evento. Las políticas no apropiativas no admiten la transición de RUN a READY

Nombre alumno:
DNI:

Gestión de procesos (2 *puntos*)

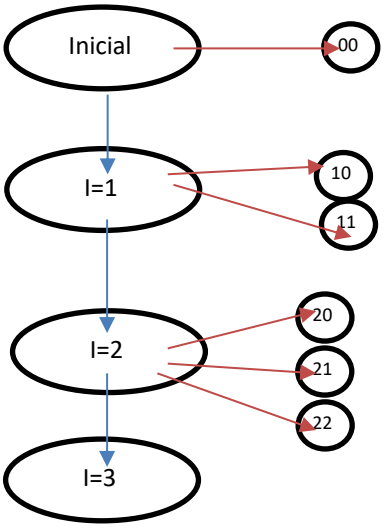
1. *(2 puntos*) El código de la izquierda corresponde con el programa n_computation y el de la derecha al programa compute. El programa do_work no es relevante, no hace llamadas a sistema, solo realiza un cálculo. Asume que las funciones usage existen y que las funciones error_y_exit y tratarExitCode son las usadas durante el curso. Asume también que el vector de 10 pids es suficiente. Analiza los códigos y responde a las preguntas:

```
/* n_computation */
void f_alarm(int s)
{}

void main(int argc, char *argv[])
{
int i, ec, ret;
pid_t pids[10];
uint levels;
char curr_level[64], buffer[128];
if (argc != 2) usage(argv[0]);
struct sigaction sa;
sigset_t m;

sigemptyset(&m);
sigaddset(&m,SIGUSR1);
sigprocmask(SIG_BLOCK, &m, NULL);

levels = atoi(argv[1]);
for (i = 0; i < levels; i++){
    pids[i] = fork();
    if (pids[i] > 0){
      sprintf(curr_level,"%d", i + 1);
      execlp("./compute", "compute", curr_level,
               NULL);
      error_y_exit("Error execlp", 2);
    }else if (pids[i] < 0 ){
      error_y_exit("Error fork", 2);
    }
}

kill(getppid(), SIGUSR1);

while((ret = waitpid(-1, &ec, 0)) > 0){
  trataExitCode( ret, ec);
}
exit(0);
}
```

```
/* compute */
int sig_usr1 = 0;
void f_usr1(int s)
{
sig_usr1 = 1;
}

void main(int argc, char *argv[])
{
int i, ec;
pid_t pids[10];
char buffer[128];
struct sigaction sa;
sigemptyset(&m);

sa.sa_handler = f_usr1;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGUSR1, &sa, NULL);

sigaddset(&m, SIGUSR1);
sigprocmask(SIG_UNBLOCK, &m, NULL);


while(sig_usr1 == 0);

for (i = 0; i < atoi(argv[1]); i++){
  pids[i] = fork();
  if (pids[i] == 0){
    execlp("./do_work", "do_work",  NULL);
    error_y_exit("Error execlp", 2);
  }else if (pids[i] < 0 ){
    error_y_exit("Error fork", 2);
  }
}
while(waitpid(-1, NULL, 0) >0 );
kill( getppid(), SIGUSR1);
exit(atoi(argv[1]));
}
```

a) (0.75 puntos) Dibuja la jerarquia de procesos que se genera al ejecuta el programa n_computation de la siguiente forma: ./n_computation 3. En el dibujo asigna un número a cada proceso para las preguntas posteriores. I

Nombre alumno:
DNI:



Las flechas azules indican procesos creados en el programa n_computations y las rojas en el compute

b) (0.25) ¿Qué podría pasar si eliminamos el sigprocmask de los dos programas?

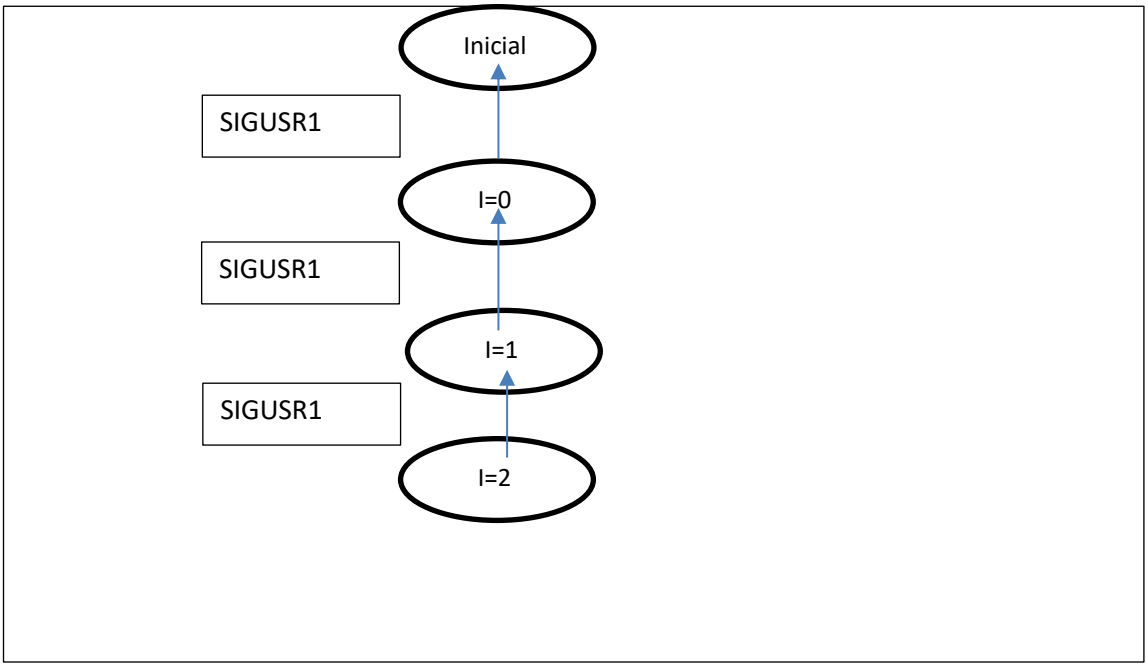
Podría pasar que el signal SIGUSR1 llegara antes del sigaction y por lo tanto se ejecutara la acción por defecto que es terminar el proceso.

c) (0.25) ¿Qué pasaría si movemos la función f_sigusr1 (antes del main) y el sigaction (antes de la creación de procesos) del programa compute al n_computation?

Al hacer el execlp se pierde las modificaciones que se hubieran hecho con el sigaction por lo que se ejecutaría la acción por defecto del SIGUSR1

d) (0.5) Replica la jerarquía de procesos incluyendo SOLO los procesos que envían o reciben algún signal. Indica claramente quien envía/recibe y que signal envía/recibe.

Nombre alumno:
DNI:



e) (0.25) El bucle que ejecuta la función trataExitCode, ¿Cuántos y qué mensajes escribirá?

Ninguna, ya que ese proceso no tiene hijos.

Nombre alumno:
DNI:

Pipes (*2 puntos*)

La Figura 1 contiene el código del programa *pipes*.

```
1.  void ras(int s) {
2.          write(2,"suspes\n",7);
3.          exit(1);
4.  }
5.  int main() {
6.  int fd[2],r,pid;
7.  struct sigaction sa,antic;
8.  sa.sa_handler=ras;
9.  sigemptyset(&sa.sa_mask);
10. sa.sa_flags=0;
11. if (sigaction(SIGPIPE,&sa,&antic)<0) perror("sig");
12. close(1);
13. pipe(fd);
14. pid=fork();
15. if (pid==0) {
16.         dup2(fd[0],0);
17.         dup2(2,1);
18.         close(fd[1]);
19.         execlp("cat","cat",(char*)0);
20. }
21. else {
22.         close(fd[0]);
23.         write(3,"Examen ",7);
24.         waitpid(-1,NULL,0);
25. }
26. write(2,"aprovat\n",8);
27. sigaction(SIGPIPE,&antic,NULL);
28. exit(0);
29. }
```

Figura 1 Código de pipes

Nota: el programa “cat”, en ausencia de ficheros de entrada, lee de la entrada estándar y concatena lo leído, escribiéndolo por la salida estándar.

Ponemos en ejecución este programa con el siguiente comando: *./pipes*

Nombre alumno:
DNI:

1. *(1 punto)* ¿Qué es una pipe?¿Para qué se utilizan las pipes? (en general y para este caso en particular). ¿Qué tipo de pipes utiliza este código?

Una pipe és un dispositiu, un canal de comunicació unidireccional entre processos. És una estructura FIFO, tot el que entra surt en l'ordre d'entrada o es queda dins la pipe fins que algun procés drena el contingut.

En aquest cas, la pipe comunica dos processos, pare i fill. El procés fill llegeix de la pipe. El procés pare escriu a la pipe.

Aquest codi fa servir *unnamed pipes*.

2. *(0,75 puntos)* Completa la siguiente figura con el estado de la **tabla de canales solo del proceso hijo**, tabla de ficheros abiertos y tabla de inodos, suponiendo que el hijo está en la línea 19 y el padre en la 24.

Tabla de Canales

	Entrada TFA
0	1
1	0
2	0
3	
4	
5	

Tabla de Ficheros abiertos

	refs	modo	Posición l/e	Entrada T.inodo
0	4	RW	---	0
1	1	R	---	1
2	1	W	---	1
3				
4				
5				
6				

Tabla de iNodo

	refs	inodo
0	1	I-tty
1	2	I-pipe
2		
3		
4		
5		

3. *(0,25 puntos)* ¿Qué mensaje saldrá por pantalla? ¿El programa acaba?

"Examen "

El programa no acaba perquè el procés fill espera indefinidament que el pare tanqui el canal d'escriptura de la pipe. El pare espera indefinidament que el fill acabi.

Nombre alumno:
DNI:

Sistema de ficheros (2 *puntos*)

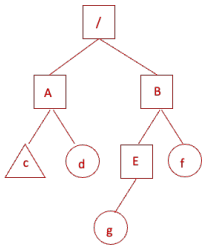
Suponed un sistema de ficheros descrito por los siguientes inodos y bloques de datos:

Listado de Inodos											
ID Inodo	2	3	4	5	6	7	8				
#enlaces	4	2	3	1	1	2	2				
Tipo	d	d	d	l	-	d	-				
path	-	-	-	/B/E	-	-	-				
BDs	0	1	2	-	3	4	5 6				

ID BD	0		1		2		3	4		5	6				
Datos	. .. A B	2 2 3 4	. .. c d	3 2 5 6	. .. E f	4 2 7 8	Texto	. .. g	7 4 8	Texto	Texto				

El campo *Tipo* del inodo puede tomar como valor d, l o – en función de si representa a un directorio, a un soft link o a un fichero de datos respectivamente. El campo path sólo se usa para el caso de los soft links (cuando el path del fichero apuntado cabe en el inodo). El tamaño de bloque es 512 bytes.

1. *(0,5 puntos)* Dibuja la jerarquía de ficheros que representan estos inodos y bloques. Usa un cuadrado para representar directorios, un triángulo para soft links y un círculo para ficheros de datos.



2. Dado el siguiente código:

```
1. int fdr, fdw,ret;
2. char buf[512];
3. fdr=open("/A/c/g", O_RDONLY);
4. fdw=open("/A/h",O_WRONLY|0_CREAT, S_IRUSR| S_IWUSR); /* S_IRUSR| S_IWUSR == 0600 */
5. while ((ret=read(fdr,buf,sizeof(buf)))>0)
6.    write(fdw,buf,ret);
7. close(fdr);close(fdw);
8. unlink("/A/c/g"); /* borra el fichero /A/c/g */
```

Suponiendo que todas las llamadas a sistema se ejecutan sin devolver error, contesta a las siguientes preguntas de manera justificada.

a) *(0,5 puntos)* Indica la secuencia de accesos a inodos y bloques de datos que hará la llamada a sistema de la línea 3 (*fdr=open("/A/c/g", O_RDONLY)*) .

Nombre alumno:
DNI:

Accesos: (superbloque) Inodo raíz, bloque raíz, inodo A, Bloque A, inodo C (es un soft link, obtenemos el path real a resolver), (superbloque) Inodo raíz, bloque raíz, inodo B, bloque B, inodo E, Bloque E, inodo g

- I2 B0 I3 B1 I5 I2 B0 I4 B2 I7 B4 I8

Justificación: Hay que acceder a todos los inodos y directorios del path para poder averiguar cuál es el inodo que hay que cargar en la tabla de inodos. En este caso atravesamos un soft link, que tiene en su inodo el path real del inodo que hay que resolver1

b) *(0,5 puntos)* Para cada llamada a sistema, indica cuál de las siguientes tablas de gestión de entrada salida **modificará**. En la justificación indica cómo las modifica.

MODIFICA SI/NO	Tabla de canales	Tabla de ficheros abiertos	Tabla de inodos
`ret=read(fdr,buf,sizeof(buf))`	NO	SI	NO
`write(fdw,buf,ret);`	NO	SI	SI
`unlink("/A/c/g");`	NO	NO	NO

Justificación:

Read: modifica puntero de lectura y escritura

Write: modifica puntero de lectura y escritura y actualiza tamaño en inodo

Unlink : no es un fichero en uso, no aparece en las tablas.

c) *(0,5 puntos)* Modifica las estructuras de datos para representar cómo quedarán después de ejecutar el código anterior.

Listado de Inodos											
ID Inodo	2	3	4	5	6	7	8	**9**			
#enlaces	4	2	3	1	1	2	~~2~~ **1**	**1**			
Tipo	d	d	d	l	-	d	-	**-**			
path	-	-	-	/B/E	-	-	-	**-**			
BDs	0	1	2	-	3	4	5 6	**7** **8**			

ID BD	0		1		2		3	4		5	6	**7**	**8**		
Datos	. .. A B	2 2 3 4	. .. c d **h**	3 2 5 6 **9**	. .. E f	4 2 7 8	Texto	. .. ~~g~~	7 4 ~~8~~	Texto	Texto	**Texto**	**Texto**		

Nombre alumno:
DNI:

Justificación: Creamos un nuevo fichero en el directorio A (nuevo inodo y nueva entrada en el directorio A) con una copia de g (ocupa 2 bloques nuevos). Además se elimina g del directorio E y eso implica decrementar el número de enlaces en su inodo

Memoria (2 *puntos*)

Tenemos una máquina que tiene una gestión de memoria basada en paginación, con un tamaño de página de 4KB. En esta máquina un entero y un puntero ocupan 4 bytes. El sistema de gestión de memoria dispone de carga bajo demanda y de COW. No tenemos en cuenta ninguna variable de ninguna librería y cada programa se pone en ejecución por separado. Indica en las tablas, justificando tus respuestas, qué cantidad de páginas lógicas asignadas, así como memoria física (en KB y/o Bytes) será necesaria justo antes de acabar la ejecución, para cada una de las regiones que aparecen en las tablas, teniendo en cuenta las regiones de **TODOS** los procesos involucrados.

1. *(0,5 puntos*)

```
1.  int res[2048];
2.  int *ptr;
3.  main(){
4.     int A[2048], B[2048], i;
5.     ptr = res;
6.     for (i=0;i<2048;i++)
7.        A[i] = B[i] = i;
8.     for (i=0;i<2048;i++)
9.        ptr[i] = A[i] + B[i];
10. }
```

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data	3	8KB + 4B
Stack	5	16KB + 4B
Heap	0	0KB

JUSTIFICACIÓN: Al haber un tamaño de página de 4KB, las cantidades de memoria física asignadas serán múltiplo de este espacio. En la región Data nos encontraremos las variables globales ("res" y "ptr"). Como "res" es un array de enteros (4 bytes c/u) el espacio total que ocupa es de 8KB (2048*4), es decir 2 páginas. Ahora bien, como "ptr" también necesita espacio para guardar la dirección del puntero (4 bytes), usará una página más sólo para esa variable. Por tanto 3 páginas en total (12KB). Por otro lado, el tamaño del Stack se calcula de forma parecida, ya que existen 2 arrays de 2 páginas c/u, así como también una variable entera. Por tanto, 20KB en total. Por último, no hay espacio de Heap utilizado (0 Bytes).

2. *(0,5 puntos*)

```
1.  int *ptr;
2.  main(){
3.     int A[2048], B[2048], i;
4.     ptr = sbrk(2048 * sizeof(int));
5.     for (i=0;i<2048;i++)
6.        A[i] = B[i] = i;
7.     for (i=0;i<2048;i++)
8.        ptr[i] = A[i] + B[i];
9.  }
```

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data	1	4B
Stack	5	16KB + 4B
Heap	2	8KB

JUSTIFICACIÓN: La explicación para la región de Stack es la misma que en el código anterior. En cambio, para la región de Data ahora sólo se utiliza 4KB para poder guardar la variable "ptr" (4Bytes). Por último, dado que se emplea sbrk para asignar memoria dinámica, la región de Heap tiene 8KB asignados (2048 * 4Bytes).

3. *(1 punto*)

```
1.  int *ptr;
2.  main(){
3.     int *A, *B, i;
4.     A = sbrk(2048 * sizeof(int));
5.     B = sbrk(2048 * sizeof(int));
6.     ptr = sbrk(2048 * sizeof(int));
7.     for (i=0;i<2048;i++)
8.        A[i] = B[i] = i;
9.     fork();
10.    for (i=0;i<2048;i++)
11.       ptr[i] = A[i] + B[i];
12. }
```

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data	1	4B
Stack	2	12B + 12B
Heap	8	24KB + 8KB

Nombre alumno:
DNI:

JUSTIFICACIÓN: Primero analizamos el proceso padre. La explicación para la región de Data es la misma que en el código anterior. En cambio, para la región de Stack ahora sólo se utiliza 4KB para poder guardar las variables “i”, “A” y “B” (4Bytes c/u). Por último, dado que se emplea sbrk para asignar memoria dinámica, la región de Heap tiene 24KB asignados (2048 * 4Bytes * 3). En el proceso hijo, al existir la optimización COW, sólo se duplican aquellas páginas que requieren ser actualizadas por alguna escritura. En este caso, sólo se duplican: una página de la región de Stack (debido a la variable “i”) y 2 páginas de la zona Heap correspondientes a “ptr”, pero teniendo en cuenta que la variable “ptr” no se modifica, sino el contenido de las direcciones dinámicas que le han sido asignadas para el vector.