

Nombre y apellidos alumno:

DNI:

## Examen final de teoría de SO

**Justifica todas tus respuestas del examen.** Las respuestas no justificadas se considerarán erróneas.

### Gestión de memoria(2puntos)

Analiza el siguiente programa (mem.c) y responde a las siguientes preguntas. El programa mem.c se ejecuta sobre un sistema Linux, con la optimización Copy On Write, el tamaño de página es de 4KB, el tamaño de un entero (int) es de 4 bytes, el tamaño de las direcciones de memoria es de 32 bits, y el código del programa cabe en 1 página.

```
1. #define pagesize 4096
2. int array10[100 * pagesize];
3. int i;
4. int main ()
5. {
6.     int array20[100 * pagesize];
7.     int *array30;
8.     for (i = 0; i < 100 * pagesize; i++) array10[i] = 10;
9.
10.    array30 = malloc (100 * sizeof(int) * pagesize);
11.
12.    if (fork () == 0)
13.    {
14.        for (i = 0; i < 100 * pagesize; i++) array20[i] = 20;
15.    }
16.    else
17.        for (i = 0; i < 100 * pagesize; i++) array30[i] = 30;
18.
19.    free(array30);
20. }
```

- a) **(0,5 puntos)** Indica para las variables *array10*, *array20* y *array30* en qué región de memoria del proceso están localizadas y dónde está localizado su contenido.

Nombre y apellidos alumno:

DNI:

b) **(0,5 puntos)** Justifica cuántos frames de región de pila se están consumiendo en la línea 17

c) **(0,5 puntos)** Supón que paramos la ejecución en la línea 17. Consultamos las herramientas del sistema y vemos que el tamaño del heap para el proceso padre es de 533 páginas. Justifica este tamaño del heap.

d) **(0,5 puntos)** Modifica las líneas de código que consideres para que la reserva y liberación de memoria dinámica sea mediante llamadas a sistema

Nombre y apellidos alumno:

DNI:

**Procesos y signals (3 puntos)**

Nos dan el siguiente código (programa n\_steps) que genera una jerarquía de procesos.

```

1. uint sigusr1_received = 0;
2. void notifica(char *pid_str)
3. {
4.     int pid;
5.     pid = atoi(pid_str);
6.     if (pid) kill(pid, SIGUSR1);
7. }
8. void f_sigusr1(int s)
9. {
10. sigusr1_received = 1;
11.}
12.void espera()
13.{
14. alarm(5);
15. while(sigusr1_received == 0);
16. alarm(0);
17.}
18.void do_work(int step){// Ejecuta cálculo }
19.void espera_hijos()
20.{
21. char buffer[256];
22. int hijos = 0;
23. while(waitpid(-1,NULL, WNOHANG)> 0) hijos++;
24. sprintf(buffer,"Hijos terminados %d\n", hijos);
25. write(1, buffer, strlen(buffer));
26.}

```

```

27.void main(int argc, char *argv[])
28.{
29. int ret , step;
30. char buffer[256], buffer1[256];
31. step = atoi(argv[1]);
32.
33. if (step > 0 ){
34.     ret = fork();
35.     if (ret > 0){
36.         sprintf(buffer,"%d", ret);
37.         sprintf(buffer1,"%d", step-1);
38.         execlp(argv[0], argv[0], buffer1, buffer, NULL);
39.     }
40. }else{
41.     do_work(step);
42.     notifica(argv[2]);
43.     espera_hijos();
44.     exit(0);
45. }
46. espera();
47. do_work(step);
48. notifica(argv[2]);
49. exit(0);
50.}

```

El objetivo del código es que cada proceso (excepto uno) espere la recepción del signal SIGUSR, luego ejecute la función do\_work y notifique al siguiente que ya puede ejecutarse. Hay uno que no espera sino que ejecuta la función e inicia la cadena. Conociendo esta funcionalidad básica, y sabiendo que el código está incompleto, contesta las siguientes preguntas. Se ha eliminado el control de errores por simplicidad. Sabemos, además, que al inicio del main los signals están bloqueados y las acciones asociadas son las acciones por defecto.

- a) **(0,5 puntos)** Dibuja la jerarquía de procesos que genera si ejecutamos el programa de la siguiente forma “n\_steps 5 0”. Asigna un número a cada proceso para referirte a ellos e indica que parámetros recibiría cada proceso. Si el proceso muta muestra el último código y argumentos.

- b) **(0,25 puntos)** La función “espera\_hijos”, ¿Realiza una espera activa o bloqueante? ¿Podemos saber qué valor se imprimiría para la variable “hijos”?

Nombre y apellidos alumno:

DNI:

- c) **(0,25 puntos)** La función “espera\_hijos”, ¿Es correcto el flag que utiliza si queremos que haga una espera bloqueante y la variable “hijos” tenga el número de hijos creados por cada proceso? Si no es correcto indica que valor deberíamos usar en el flag.

- d) **(0,25 puntos)** Indica qué signals recibirá cada proceso y qué procesos ejecutarán la función do\_work

- e) **(0,5 puntos)** Tal y como está el código, ¿qué deberíamos añadir y donde (línea de código) para garantizar que los procesos puedan recibir los señales que se usan y tratar el SIGUSR1 mediante la función f\_sigusr1 ? (asume este cambio en los siguientes apartados).

- f) **(0,5 puntos)** ¿Qué modificaciones harías en la función “espera” para que no consuma tiempo de CPU? (Conservando el control de los 5 segundos tal cual está ahora)

- g) **(0,5 puntos)** ¿Qué modificación harías en la función “espera\_hijos” para detectar, para cada proceso, si ha ejecutado o no la función do\_work?

- h) **(0,25 puntos)** ¿Se cumplen los requisitos para poder ejecutar la llamada a sistema kill de la línea 6?

Nombre y apellidos alumno:

DNI:

**Procesos y pipes (3 puntos)**

La figura 1 muestra el código de los programas `fusion_encryptada` y `encryptar` (se omite el código de control de errores).

```

1. /* fusion_encryptada */
2. #define N 10
3. main(int argc, char *argv[]){
4.     int i,ret,fd_pipe[N][2];
5.     char c;
6.     i=0; ret=1;
7.     while ((i<argc-1) && (ret>0)){
8.         pipe(fd_pipe[i]);
9.         ret=fork();
10.        if (ret>0) i++;
11.    }
12.    if (ret == 0) {
13.        close(0); open(argv[i+1], O_RDONLY);
14.        dup2(fd_pipe[i][1],1);
15.        execlp("./encryptar","encryptar",(char *)0);
16.    }
17.    for (i=0;i<argc-1;i++){
18.        close(fd_pipe[i][1]);
19.    }
20.    for (i=0;i<argc-1;i++){
21.        while((ret=read(fd_pipe[i][0],&c,sizeof(char))>0){
22.            write(1,&c,sizeof(char));
23.        }
24.    }
25.    exit(0);
26. }

```

```

27. /* encryptar */
28. char crypt (char *c) {
29.     /* Codigo para encriptar el carácter c
30.      * No es relevante para el ejercicio
31.      */
32. }
33. main(int argc, char *argv[]){
34.     char c1,c2;
35.     while ((ret=read(0, &c1, sizeof(char))>0) {
36.         c2=crypt(c1);
37.         write(1, &c2, sizeof(char));
38.     }
39.     exit(0);
40. }

```

figura 1 Código del programa `fusion_encryptada` y `encryptar`

Desde el directorio en el que tenemos los dos códigos ejecutamos el siguiente comando: `./fusion_encryptada f1 f2 > f3`. Donde `f1`, `f2`, y `f3` son ficheros que existen y su contenido es respectivamente: "Feliz", "año", y "nuevo".

Suponiendo que ninguna llamada a sistema devuelve error, responde razonadamente a las siguientes preguntas

- a) **(0,5 puntos)** Dibuja la jerarquía de procesos que se creará como consecuencia de este comando. Representa también la(s) pipe(s) creada(s), indicando de qué pipe(s) lee y/o escribe cada proceso. Añade un identificador a cada proceso y pipe para poder referirte a ellos en los siguientes apartados.

- b) **(0,5 puntos)** Completa la figura con el estado de las tablas, considerando a todos los procesos de la jerarquía y suponiendo que todos se encuentran entre la línea 11 y la línea 12. Añade las filas y tablas que necesites.

Nombre y apellidos alumno:

DNI:

Tabla de Canales		Tabla de Ficheros abiertos				Tabla de iNodo	
Entrada TFA		refs	modo	Pos. l/e	Entrada T.inodo	refs	inodo
0		0	rw	-	0	0	1   tty1
1		1				1	
2		2				2	
3		3				3	

- c) **(0,5 puntos)** Completa la figura con el estado de las tablas, suponiendo que el proceso inicial va a ejecutar el exit. Considera los procesos de la jerarquía que estén vivos en ese momento. Añade las filas y tablas que necesites.

Tabla de Canales		Tabla de Ficheros abiertos				Tabla de iNodo	
Entrada TFA		refs	modo	Pos. l/e	Entrada T.inodo	refs	inodo
0		0	rw	-	0	0	1   tty1
1		1				1	
2		2				2	
3		3				3	

- d) **(0,5 puntos)** Rellena la siguiente tabla indicando qué procesos de la jerarquía leen y/o escriben en los dispositivos.

	Lectores	Escritores	Justificación
<i>f1</i>			
<i>f2</i>			
<i>f3</i>			
<i>terminal</i>			

- e) **(0,5 puntos)** ¿Cuál será el contenido de f3 después de ejecutar este comando? ¿Se puede garantizar que si repetimos varias veces el mismo comando el contenido de f3 será siempre el mismo?

- f) **(0,5 puntos)** ¿Qué procesos acabarán la ejecución?


Nombre y apellidos alumno:

DNI:

### Sistema de ficheros (2 puntos)

Tenemos un sistema de ficheros tipo Unix, con 12 punteros directos a bloques de datos y tres punteros indirectos (simple, doble y triple direcciones a bloque). El tamaño del puntero (a bloque, a inodo y el de la Tabla de Ficheros Abiertos) es de 4 Bytes. El tamaño de los bloques es de 4096 Bytes. Los tipos de ficheros son: directory, regular, fifo y softlink. Las tablas de Inodos y de Bloques de Datos se muestran en la figura 2. Ejecutamos un ls con las opciones long, inode (primera columna), all y Recursive (para ver el subdirectorio Datei):

```
murphy@numenor: /home/murphy/Examen$ ls -liaR
total 56
800170 drwxr-xr-x 3 murphy students 4096 Dec 23 11:47 .
787670 drwxr-xr-x 3 murphy students 4096 Dec 18 14:03 ..
800210 -rwxr-xr-x 1 murphy students 16920 Dec 23 11:47 caps
800177 drwxr-xr-x 2 murphy students 4096 Dec 23 11:06 Datei
800178 lrwxrwxrwx 1 murphy students 5 Dec 18 11:46 Folder -> Datei
800179 -rw-r--r-- 3 murphy students 4211 Dec 18 11:49 Lorem.bp
800179 -rw-r--r-- 3 murphy students 4211 Dec 18 11:49 text.txt
800176 prw-r--r-- 2 murphy students 0 Dec 18 11:45 tube
./Datei:
total 16
800177 drwxr-xr-x 2 murphy students 4096 Dec 23 11:06 .
800170 drwxr-xr-x 3 murphy students 4096 Dec 23 11:47 ..
800179 -rw-r--r-- 3 murphy students 4211 Dec 18 11:49 Ipsum.txt
800176 prw-r--r-- 2 murphy students 0 Dec 18 11:45 mypipe
```

- a) **(0,5 puntos)** Rellenad las celdas marcadas con  de las tablas de la figura 2 que tenéis al final del ejercicio.
- b) **(0,25 puntos)** En este sistema de ficheros, ¿cuál es el tamaño máximo de un fichero? ¿Por qué?

- c) **(1,25 puntos)** Ejecutamos sin errores, desde el terminal, el programa de la figura 2.
- a. **(0,5 pts.)** Al bucle (lin. 6-9), ¿cuántas llamadas a la `syscall` read se han hecho? ¿Cuántos bloques de datos se han leído?

- b. **(0,5 pts.)** Escribe los números de inodos y di cuántos bloques de datos se han visitado al ejecutar la línea 4?

- c. **(0,25 pts.)** ¿Qué estructuras de datos del kernel se han modificado al ejecutar la línea 5? ¿Qué inodos y cuántos bloques de datos se liberan?

Nombre y apellidos alumno:

DNI:

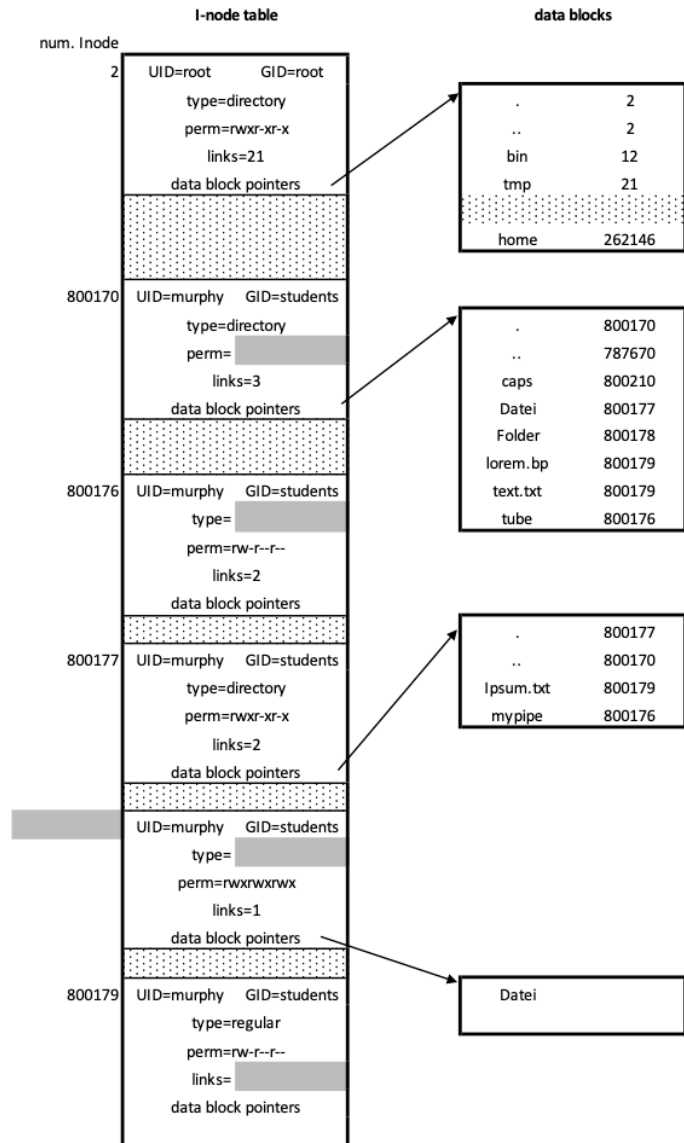


figura 2 Tabla de i-nodos y bloques

```

1. int main(int argc, char**argv) {
2.   char c;
3.   int ifd, ofd;
4.   ifd =
5.     open("/home/murphy/Examen/Folder/Ipsum.txt", O_RDONLY);
6.   ofd =
7.     open("new_IPSUM.TXT", O_WRONLY|O_CREAT|O_TRUNC, 0644);
8.   while (read(ifd, &c, 1) > 0) {
9.     if (c > 96 && c < 123) c = c - 32;
10.    write(ofd, &c, 1);
11.  }
12.  close(ifd); close(ofd);

```

figura 3 Código del programa caps.c



Nombre y apellidos alumno:

DNI:

## Examen final de teoría de SO

**Justifica todas tus respuestas del examen.** Las respuestas no justificadas se considerarán erróneas.

### Gestión de memoria(2puntos)

Analiza el siguiente programa (mem.c) y responde a las siguientes preguntas. El programa mem.c se ejecuta sobre un sistema Linux, con la optimización Copy On Write, el tamaño de página es de 4KB, el tamaño de un entero (int) es de 4 bytes, el tamaño de las direcciones de memoria es de 32 bits, y el código del programa cabe en 1 página.

```
1. #define pagesize 4096
2. int array10[100 * pagesize];
3. int i;
4. int main ()
5. {
6.     int array20[100 * pagesize];
7.     int *array30;
8.     for (i = 0; i < 100 * pagesize; i++) array10[i] = 10;
9.
10.    array30 = malloc (100 * sizeof(int) * pagesize);
11.
12.    if (fork () == 0)
13.    {
14.        for (i = 0; i < 100 * pagesize; i++) array20[i] = 20;
15.    }
16.    else
17.        for (i = 0; i < 100 * pagesize; i++) array30[i] = 30;
18.
19.    free(array30);
20. }
```

- a) **(0,5 puntos)** Indica para las variables *array10*, *array20* y *array30* en qué región de memoria del proceso están localizadas y dónde está localizado su contenido.

Array10 está es una variable global, está en la región de datos y su contenido también

Array20 es un array local a la funcion main y ella misma y su contenido estan en la pila

Array30 es un puntero a entero, está localizado en la pila y su contenido en el heap

- b) **(0,5 puntos)** Justifica cuántos frames de región de pila se están consumiendo en la línea 17

En la pila se almacenan las variables *array20* y *array30*, que ocupan respectivamente 1600 KB y 4 bytes, es decir, 401 frames (400 para *array20* y 1 para *array30* ).

En la línea 17 el padre ha actualizado completamente el *array20* y COW ha tenido que duplicar los frames que ocupa (en principio compartidos entre padre e hijo). El valor de *array30* al asignarse antes del fork y no variar se comparte entre padre e hijo (el contenido se modifica en el heap). USO de pila: 400 frames para el padre y 400 para el hijo (*array20*), más 1 frame compartido (*array30*)→ 801 frames de pila

Nombre y apellidos alumno:

DNI:

- c) **(0,5 puntos)** Supón que paramos la ejecución en la línea 17. Consultamos las herramientas del sistema y vemos que el tamaño del heap para el proceso padre es de 533 páginas. Justifica este tamaño del heap.

El programa mem.c està usando funciones de librería de C para la reserva/liberación de memòria. El programa pide específicamente 400 páginas para almacenar la variable array30 que el padre actualiza completamente en la línea 16.

El resto de pàgines asignadas al heap son datos de la pròpia librería para su pròpia gestión de la memòria dinàmica del proceso, y también para reserva prèvia con el fin de ahorrar llamadas a sistema en posteriores peticiones de nueva memoria

- d) **(0,5 puntos)** Modifica las líneas de código que consideres para que la reserva y liberación de memoria dinámica sea mediante llamadas a sistema

Línea 10: `array30 = sbrk (100 * sizeof(int) * pagesize);`

Línea 19: `sbrk (-100 * sizeof(int) * pagesize);`

## Procesos y signals (3 puntos)

Nos dan el siguiente código (programa n\_steps) que genera una jerarquía de procesos.

```
1. uint sigusr1_received = 0;
2. void notifica(char *pid_str)
3. {
4.     int pid;
5.     pid = atoi(pid_str);
6.     if (pid) kill(pid, SIGUSR1);
7. }
8. void f_sigusr1(int s)
9. {
10.    sigusr1_received = 1;
11. }
12. void espera()
13. {
14.    alarm(5);
15.    while(sigusr1_received == 0);
16.    alarm(0);
17. }
18. void do_work(int step){// Ejecuta cálculo }
19. void espera_hijos()
20. {
21.    char buffer[256];
22.    int hijos = 0;
23.    while(waitpid(-1,NULL, WNOHANG)> 0) hijos++;
24.    sprintf(buffer,"Hijos terminados %d\n", hijos);
25.    write(1, buffer, strlen(buffer));
26. }
```

```
27. void main(int argc, char *argv[])
28. {
29.    int ret , step;
30.    char buffer[256], buffer1[256];
31.    step = atoi(argv[1]);
32.
33.    if (step > 0 ){
34.        ret = fork();
35.        if (ret > 0){
36.            sprintf(buffer,"%d", ret);
37.            sprintf(buffer1,"%d", step-1);
38.            execlp(argv[0], argv[0], buffer1, buffer, NULL);
39.        }
40.    }else{
41.        do_work(step);
42.        notifica(argv[2]);
43.        espera_hijos();
44.        exit(0);
45.    }
46.    espera();
47.    do_work(step);
48.    notifica(argv[2]);
49.    exit(0);
50. }
```

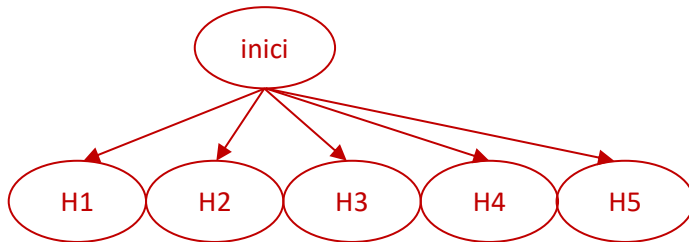
El objetivo del código es que cada proceso (excepto uno) espere la recepción del signal SIGUSR, luego ejecute la función do\_work y notifique al siguiente que ya puede ejecutarse. Hay uno que no espera sino que ejecuta la función e inicia la cadena. Conociendo esta funcionalidad básica, y sabiendo que el código está incompleto, contesta las siguientes

Nombre y apellidos alumno:

DNI:

preguntas. Se ha eliminado el control de errores por simplicidad. Sabemos, además, que al inicio del main los signals están bloqueados y las acciones asociadas son las acciones por defecto.

- a) **(0,5 puntos)** Dibuja la jerarquía de procesos que genera si ejecutamos el programa de la siguiente forma “n\_steps 5 0”. Asigna un número a cada proceso para referirte a ellos e indica que parámetros recibiría cada proceso. Si el proceso muta muestra el último código y argumentos.



Cada proceso hijo recibe 5,4,3,2,1 como argv[1] respectivamente y 0, Pid H1, Pid H2, Pid H4 respectivamente. El padre al final tiene 0, pidH5.

- b) **(0,25 puntos)** La función “espera\_hijos”, ¿Realiza una espera activa o bloqueante? ¿Podemos saber qué valor se imprimiría para la variable “hijos”?

La función espera\_hijos hace una espera activa por el flag WNOHANG y el hecho de estar en un bucle. Sin embargo, la condición de salida del bucle y el hecho de que los procesos tengan los signals bloqueados, hará que el valor de hijos valga 0.

- c) **(0,25 puntos)** La función “espera\_hijos”, ¿Es correcto el flag que utiliza si queremos que haga una espera bloqueante y la variable “hijos” tenga el número de hijos creados por cada proceso? Si no es correcto indica que valor deberíamos usar en el flag.

No, deberíamos usar un 0 como flag.

- d) **(0,25 puntos)** Indica qué signals recibirá cada proceso y qué procesos ejecutarán la función do\_work

La función do\_work la hará solo el inicial. Los demás recibirán el SIGALARM pero como está bloqueado no se tratarán. El H5 recibirá (pero no tratará) el SIGUSR1.

- e) **(0,5 puntos)** Tal y como está el código, ¿qué deberíamos añadir y donde (línea de código) para garantizar que los procesos puedan recibir los señales que se usan y tratar el SIGUSR1 mediante la función f\_sigusr1 ? (asume este cambio en los siguientes apartados).

Habría que añadir un sigaction del SIGUSR1 para asociándolo con f\_sigusr1 y desbloquear el SIGUSR1 y SIGALRM mediante la llamada sigprocmask.

- f) **(0,5 puntos)** ¿Qué modificaciones harías en la función “espera” para que no consuma tiempo de CPU? (Conservando el control de los 5 segundos tal cual está ahora)

Nombre y apellidos alumno:

DNI:

Utilizaríamos un sigsuspend en substitución de bucle de espera del SIGUSR1. El sigsuspend debería tener una máscara con el SIGALRM y SIGUSR1 desbloqueados.

```
sigseg_t mask;
sigfullset(&mask);
sigdelset(&mask, SIGUSR1);
sigdelset(&mask, SIGALRM);
sigsuspend(&mask);
```

g) **(0,5 puntos)** ¿Qué modificación harías en la función “espera\_hijos” para detectar, para cada proceso, si ha ejecutado o no la función do\_work?

Deberíamos detectar si el proceso hijo a terminado por un signal o por un exit.

```
int res;
while(waitpid(-1,&res, 0) > 0){
    if (WIFEXITED(res)) → do_work
    else → no do_work
}
```

h) **(0,25 puntos)** ¿Se cumplen los requisitos para poder ejecutar la llamada a sistema kill de la línea 6?

Si, se comprueba que el pid no sea cero, los pids son válidos y los procesos son del mismo usuario.

## Procesos y pipes (3 puntos)

La figura 1 muestra el código de los programas fusion\_encryptada y encriptar (se omite el código de control de errores).

<pre>1. /* fusion_encryptada */ 2. #define N 10 3. main(int argc, char *argv[]){ 4.     int i,ret,fd_pipe[N][2]; 5.     char c; 6.     i=0; ret=1; 7.     while ((i&lt;argc-1) &amp;&amp; (ret&gt;0)){ 8.         pipe(fd_pipe[i]); 9.         ret=fork(); 10.        if (ret&gt;0) i++; 11.    } 12.    if (ret == 0) { 13.        close(0); open(argv[i+1], O_RDONLY); 14.        dup2(fd_pipe[i][1],1); 15.        execlp("./encriptar","encriptar",(char *)0); 16.    } 17.    for (i=0;i&lt;argc-1;i++){ 18.        close(fd_pipe[i][1]); 19.    } 20.    for (i=0;i&lt;argc-1;i++){ 21.        while((ret=read(fd_pipe[i][0],&amp;c,sizeof(char))&gt;0){ 22.            write(1,&amp;c,sizeof(char)); 23.        } 24.    } 25.    exit(0); 26. }</pre>	<pre>27. /* encriptar */ 28. char crypt (char *c) { 29.     /* Codigo para encriptar el carácter c 30.      * No es relevante para el ejercicio 31.      */ 32. } 33. main(int argc, char *argv[]){ 34.     char c1,c2; 35.     while ((ret=read(0, &amp;c1, sizeof(char))&gt;0) { 36.         c2=crypt(c1); 37.         write(1, &amp;c2, sizeof(char)); 38.     } 39.     exit(0); 40. }</pre>
---	--

figura 1 Código del programa fusion\_encryptada y encriptar

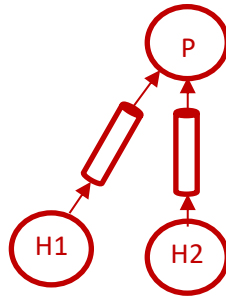
Desde el directorio en el que tenemos los dos códigos ejecutamos el siguiente comando: `./fusion_encryptada f1 f2 > f3`  
 Donde f1, f2, y f3 son ficheros que existen y su contenido es respectivamente: “Feliz”, “año”, y “nuevo”.

Nombre y apellidos alumno:

DNI:

Suponiendo que ninguna llamada a sistema devuelve error, responde razonadamente a las siguientes preguntas

- a) **(0,5 puntos)** Dibuja la jerarquía de procesos que se creará como consecuencia de este comando. Representa también la(s) pipe(s) creada(s), indicando de qué pipe(s) lee y/o escribe cada proceso. Añade un identificador a cada proceso y pipe para poder referirte a ellos en los siguientes apartados.



- b) **(0,5 puntos)** Completa la figura con el estado de las tablas, considerando a todos los procesos de la jerarquía y suponiendo que todos se encuentran entre la línea 11 y la línea 12. Añade las filas y tablas que necesites.

Tabla de Canales P

Entrada TFA

0	0
1	1
2	0
3	2
4	3
5	4
6	5

Tabla de Ficheros abiertos

refs modo Pos. l/e Entrada T.inodo

0	6	rw	-	0
1	3	w	0	1
2	3	r	-	2
3	3	w	-	2
	2	r	-	3
	2	w	-	3

Tabla de iNodo

refs inodo

0	1	l tty1
1	1	l_f3
2	2	l_pipe1
3	2	l_pipe2

Tabla de Canales H1

0	0
1	1
2	0
3	2
4	3
5	
6	

Tabla de Canales H2

0	0
1	1
2	0
3	2
4	3
5	4
6	5

Nombre y apellidos alumno:

DNI:

- c) **(0,5 puntos)** Completa la figura con el estado de las tablas, suponiendo que el proceso inicial va a ejecutar el exit. Considera los procesos de la jerarquía que estén vivos en ese momento. Añade las filas y tablas que necesites.

Tabla de Canales		Tabla de Ficheros abiertos				Tabla de iNodo		
Entrada TFA		refs	modo	Pos.l/e	Entrada T.inodo	refs	inodo	
0	0	0	2	rw	-	0	1	l tty1
1	1	1	1	w	size of f3	1	1	l_f3
2	0	2	1	r	-	2	1	l_pipe1
3	2	3				3	1	l_pipe2
			1	r	-			
5	4							

- d) **(0,5 puntos)** Rellena la siguiente tabla indicando qué procesos de la jerarquía leen y/o escriben en los dispositivos.

	Lectores	Escritores	Justificación
f1	H1	--	f1 está asociado al canal 0 de H1 (por las líneas 9 y 10). Por eso cuando H1 mute a encriptar e intente leer de 0 lo hará de f1. Lo mismo con f2 y H2. F3 está asociado al canal 1 de P (los hijos pierden empiezan la ejecución con esta asociación pero la pierden en la línea 11), por eso cuando P escriba en 1 lo hará en f3. El terminal está asociado a canal 0 de P (los hijos pierden a asociación en las líneas 9 y 10) y al canal 2 de todos. No hay ningún acceso al canal 2 y los únicos accesos al 0 lo hacen los hijos dentro del programa encriptar
f2	H2	--	
f3	--	P	
terminal	--	--	

- e) **(0,5 puntos)** ¿Cuál será el contenido de f3 después de ejecutar este comando? ¿Se puede garantizar que si repetimos varias veces el mismo comando el contenido de f3 será siempre el mismo?

El contenido será las palabras: "feliz" "año" encriptado y en este orden. Y siempre será el mismo resultado. El código del Shell trunca el contenido de f3 al interpretar el operado ">". P escribe en f3 lo que va recibiendo de las pipes (una por hijo). Hasta que un hijo no acaba de escribir el padre no pasa a leer de la pipe en la que escribe el siguiente hijo.

- f) **(0,5 puntos)** ¿Qué procesos acabarán la ejecución?

Todos. Cada hijo lee un fichero y escribe en la pipe que le toca su contenido encriptado. Cuando la lectura del fichero llegue al final devolverá 0 y el hijo acabará la ejecución. Al acabar la ejecución se cierran todos sus canales (y deja de constar como escritor en la pipe). El padre crea tantas pipes y procesos como ficheros recibe como parámetro. A continuación cierra su extremo de escritura de todas las pipes porque en ningún momento va a escribir en las pipes. Y luego ejecuta un bucle con tantas iteraciones como ficheros, en cada iteración lee de la pipe que corresponde con la iteración hasta que la lectura devuelve 0. Esto ocurrirá cuando el hijo correspondiente (único escritor en la pipe a través de dos canales, fd\_pipe[i][1] y 1) muera.

## Sistema de ficheros (2 puntos)

Tenemos un sistema de ficheros tipo Unix, con 12 punteros directos a bloques de datos y tres punteros indirectos (simple, doble y triple direcciones a bloque). El tamaño del puntero (a bloque, a inodo i el de la Tabla de Ficheros

Nombre y apellidos alumno:

DNI:

Abiertos) es de 4 Bytes. El tamaño de los bloques es de 4096 Bytes. Los tipos de ficheros son: directory, regular, fifo i softlink. Las tablas de Inodos y de Bloques de Datos se muestran en la figura 2. Ejecutamos un ls con las opciones long, inode (primera columna), all y Recursive (para ver el subdirectorio Datei):

```
murphy@numenor:/home/murphy/Examen$ ls -liaR
total 56
800170 drwxr-xr-x 3 murphy students 4096 Dec 23 11:47 .
787670 drwxr-xr-x 3 murphy students 4096 Dec 18 14:03 ..
800210 -rwxr-xr-x 1 murphy students 16920 Dec 23 11:47 caps
800177 drwxr-xr-x 2 murphy students 4096 Dec 23 11:06 Datei
800178 lrwxrwxrwx 1 murphy students 5 Dec 18 11:46 Folder -> Datei
800179 -rw-r--r-- 3 murphy students 4211 Dec 18 11:49 Lorem.bp
800179 -rw-r--r-- 3 murphy students 4211 Dec 18 11:49 text.txt
800176 prw-r--r-- 2 murphy students 0 Dec 18 11:45 tube
./Datei:
total 16
800177 drwxr-xr-x 2 murphy students 4096 Dec 23 11:06 .
800170 drwxr-xr-x 3 murphy students 4096 Dec 23 11:47 ..
800179 -rw-r--r-- 3 murphy students 4211 Dec 18 11:49 Ipsum.txt
800176 prw-r--r-- 2 murphy students 0 Dec 18 11:45 mypipe
```

- a) **(0,5 puntos)** Rellenad las celdas marcadas con   de las tablas de la figura 2 que tenéis al final del ejercicio.
- b) **(0,25 puntos)** En este sistema de ficheros, ¿cuál es el tamaño máximo de un fichero? ¿Por qué?

El file pointer es de 4 bytes (32 bits), por tanto, el tamaño del fichero está limitado a  $2^{32} = 4 \text{ GiB}$ . Aunque los punteros a bloques nos permitirían  $4 \text{ TiB} = 4 \text{ KiB}(12 + 1 \text{ KiB} + (1 \text{ KiB})^2 + (1 \text{ KiB})^3)$ .

- c) **(1,25 puntos)** Ejecutamos sin errores, desde el terminal, el programa de la figura 2.
- a. **(0,5 pts.)** Al bucle (lin. 6-9), ¿cuántas llamadas a la `syscall` read se han hecho? ¿Cuántos bloques de datos se han leído?

4212 reads, solo el último retorna 0. Se han leído dos bloques de disco.

- b. **(0,5 pts.)** Escribe los números de inodos y di cuántos bloques de datos se han visitado al ejecutar la línea 4?

Inodos: 2, 262146, 787670, 800170, 800178, 800177, 800179. BDs: 6 (los de los directorios /, home, murphy, Examen, Folder, Datei).

- c. **(0,25 pts.)** ¿Qué estructuras de datos del kernel se han modificado al ejecutar la línea 5? ¿Qué inodos y cuántos bloques de datos se liberan?

Se pide un inodo libre, se crea una nueva entrada al BD del inodo 800170 con el nombre del fichero y el nuevo número de inodo. Se añade una entrada en las tablas de inodos, de ficheros abiertos y de canales. Dado que el fichero new\_IPSUM.TXT no existe, no se libera nada. Si ya existiera new\_IPSUM.TXT, se liberarían todos sus bloques de datos al hacer el O\_TRUNC.

Nombre y apellidos alumno:

DNI:

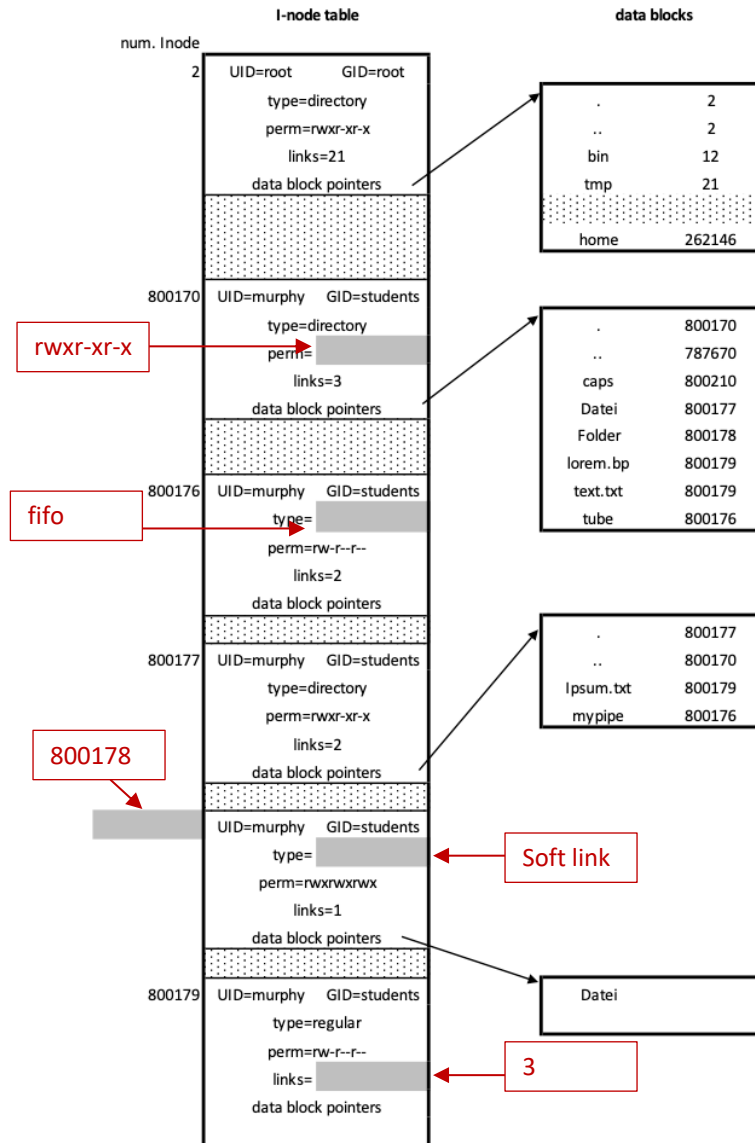


figura 2 Tabla de i-nodos y bloques

```

1. int main(int argc, char**argv) {
2.   char c;
3.   int ifd, ofd;
4.   ifd =
     open("/home/murphy/Examen/Folder/Ipsum.txt", O_RDONLY);
5.   ofd =
     open("new_IPSUM.TXT", O_WRONLY|O_CREAT|O_TRUNC, 0644);
6.   while (read(ifd, &c, 1) > 0) {
7.     if (c > 96 && c < 123) c = c - 32;
8.     write(ofd, &c, 1);
9.   }
10. close(ifd); close(ofd);

```

figura 3 Código del programa caps.c



Nombre alumno:

DNI:

## Examen final de teoría de SO

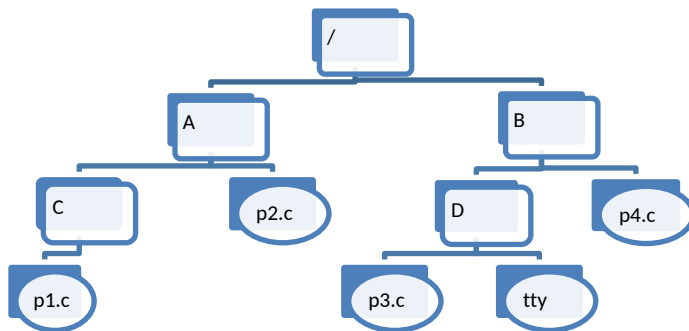
**Justifica todas tus respuestas del examen.** Las respuestas no justificadas se considerarán erróneas.

### Sistema de Ficheros (2 puntos)

Tenemos el siguiente sistema de ficheros basado en inodos. Datos:

<ul style="list-style-type: none"> <li>Los rectángulos representan directorios, los óvalos representan otros elementos</li> </ul>	<ul style="list-style-type: none"> <li>Las rutas destino de los softlinks se guardan en bloques de datos</li> </ul>
<ul style="list-style-type: none"> <li>El tamaño de bloque es de 4 KB</li> </ul>	<ul style="list-style-type: none"> <li>p1.c y p4.c son hardlinks al mismo fichero</li> </ul>
<ul style="list-style-type: none"> <li>p3.c es un soft-link a p2.c. Asume que el path no cabe en el inodo.</li> </ul>	<ul style="list-style-type: none"> <li>p1.c ocupa 2 KB</li> </ul>
<ul style="list-style-type: none"> <li>p2.c ocupa 12 KB</li> </ul>	<ul style="list-style-type: none"> <li>tty es un dispositivo especial de carácter (una consola)</li> </ul>

- a) **(0,25 puntos)** asigna a los elementos de la jerarquía siguiente un número de inodo. Indícalo justo a la derecha de cada figura. Usa el cuadro a la derecha para justificar tu asignación.



- b) **(0,75 puntos)** Completa las siguientes tablas con la información necesaria para representar la jerarquía anterior:

ID Inodo											
#enlaces											
Tipo											
Tabla de índices a BD											

ID BD	0	1	2	3	4	5	6	7	8	9	10
Contenido											

Nombre alumno:

DNI:

c) **(0,5 puntos)** Estando situados en el directorio /A ejecutamos el comando “**mkdir SubA**”.

Enumera los cambios que habría que hacer en las tablas anteriores (y en la jerarquía) para completar el comando:

d) **(0,5 puntos)** Analiza el siguiente código. Indica para cada línea qué accesos, consultas o modificaciones, se realizan a inodos y bloques de datos del sistema de ficheros. Supón que el sistema tiene **buffer cache de 1000 bloques vacía**. Indica qué accesos serían a través de la buffer cache.

```
1. int fd=open("/B/D/p3.c",O_RDWR);
2. int ret=lseek(fd,0,SEEK_END);
3. int w=write(fd,"a",1);
4. close(fd);
```

Sigue la siguiente nomenclatura, de ejemplo, como GUÍA para escribir tu respuesta:

- I3C significa que se accede al Inodo 3 para Consulta (lectura)
- B6M significa que se accede al Bloque de datos 6 para Modificarlo
- Subraya claramente los accesos que sean en buffer cache

	Accesos y justificación
Linea 1	
Linea 2	
Linea 3	
Linea 4	

Nombre alumno:

DNI:

**Gestión de memoria (2 puntos)**

Analiza el siguiente código. Responde a la siguientes preguntas de forma razonada. El código se ejecuta sobre un sistema operativo Linux de 32 bits paginado, con tamaño de página de 4096 bytes, y **no** ofrece la optimización Copy-On-Write pero **sí** ofrece SWAP (memoria virtual). El tamaño del tipo **int** es de 4 bytes. La región de código del programa ocupa 2400 bytes. Suponemos que el tamaño de las regiones para los datos viene dado únicamente por lo definido en este código. Supón que el programa "miprogram" existe, la ruta correcta y hay permiso para ejecutarlo.

```
int *a;
int b=4;

int main ()
{
    int c, ret;

    ret=fork();
    -----PUNTO A
    a=sbrk(sizeof(int));

    if (ret>0){
        waitpid(-1,NULL,0);
        a=sbrk(sizeof(int));
        *a=1;
        c>(*a)*b;
    }
    else if (ret==0)
        execlp("./miprogram", "miprogram", (char*)NULL);
    -----PUNTO B
    else error_y_exit("fork");
}
```

- A) (0,5 puntos) Indica y justifica cuánta memoria física ocupa por región (en marcos de página) la ejecución de este programa en el **punto A**.

- B) (0,25 puntos) Razona si añadir la optimización COW ofrecería alguna mejora en la gestión de memoria si hacemos avanzar el código desde el punto A hasta el **punto B**.

Nombre alumno:

DNI:

- C) (0,5 puntos) Estando en el **punto A**, justifica si con el esquema de gestión de memoria de este sistema se produce algún tipo de fragmentación de memoria, qué tipo de fragmentación sería, en caso afirmativo, y cuánta memoria se desaprovecharía.

- D) (0,5 puntos) Al ejecutar la instrucción “ **\*a=1;** ” se produce un MAJOR PAGE FAULT. Indica 3 razones que lo puedan provocar:

- E) (0,25 puntos) Indica las operaciones que realizaría el sistema para solucionar el MAJOR PAGE FAULT

Nombre alumno:

DNI:

**Procesos y signals (3 puntos)**

Analiza el siguiente código y contesta justificadamente a las preguntas. (se omite el control de errores por simplicidad). El programa "A" existe, es correcto, y no modifica nada relacionado con signals ni hace nada relevante para el ejercicio. El objetivo de este código es crear un nuevo proceso cada vez que se reciba un SIGUSR1 y terminar cuando se reciba un SIGUSR2.

```
1. uint sig1 = 0, sig2 = 0;
2. void create_process()
3. {
4.     int ret;
5.     ret = fork();
6.     if (ret == 0){
7.         execlp("./A", "A", NULL);
8.     }
9. }
10. void f_sig(int s)
11. {
12.     if (s == SIGUSR1) sig1 = 1;
13.     if (s == SIGUSR2) sig2 = 1;
14. }
15.
16. void main(int argc, char *argv[])
17. {
18.     struct sigaction new_action;
19.     sigset_t      action_mask;
20.
21.     new_action.sa_handler = f_sig;
22.     new_action.sa_flags = SA_RESTART;
23.     sigfillset(&new_action.sa_mask);
24.     sigaction(SIGUSR1, &new_action, NULL);
25.     sigaction(SIGUSR2, &new_action, NULL);
26.
27.     sigfillset(&action_mask);
28.     sigdelset(&action_mask, SIGUSR1);
29.     sigdelset(&action_mask, SIGUSR2);
30.
31.     while(1){
32.         sigsuspend(&action_mask);
33.         if (sig1) create_process();
34.         if (sig2) exit(0);
35.         sig1 = 0;
36.         waitpid(-1, NULL, 0);
37.     }
38. }
39.
```

Nombre alumno:

DNI:

Si el programa se ejecuta en un terminal y devuelve el PID 1000, y en otro terminal ejecutamos la siguiente secuencia:

```
kill -USR1 1000  
kill -USR1 1000  
kill -USR1 1000  
kill -USR2 1000
```

- a) (0,5 puntos) ¿Qué pasaría si enviamos los dos SIGUSR1 antes de llegar al sigsuspend?

- b) (0,5 puntos) Si enviamos un signal cada minuto y el programa A tarda 5 segundos en ejecutarse, ¿Al recibir el SIGUSR2, cuantos procesos hijos tendrá activos o zombies el proceso 1000?

- c) (0,5 puntos) Si movemos la llamada de la función create\_process() a la línea 12 (donde se gestiona la recepción del SIGUSR1), ¿Cuál sería la máscara de signals bloqueados de los nuevos procesos?

- d) (0,5 puntos) ¿Es necesario modificar el código para evitar que el SIGINT, en ningún momento de la ejecución, termine la ejecución de proceso? Justifica la respuesta y en caso afirmativo indica el código que añadirías y en que líneas.

Nombre alumno:

DNI:

- e) (0,5 puntos) Si quisiéramos que la gestión de la finalización de los procesos hijos fuera asíncrona, ¿Qué cambios en el código habría que hacer? Enumera los cambios de forma resumida (lista de cambios).

- f) (0,5 puntos) Si queremos hacer una mejora limitando el tiempo de ejecución de los procesos que ejecutan el programa A (por ejemplo 1000 segundos) utilizando signals, indica: (1) Si cambiarías el código del padre o de los hijos, (2) que signal elegirías, y (3) que llamada(s) a sistema añadirías. (solo se pide que termine, no que haga ninguna acción concreta)

### Procesos y pipes (3 puntos)

Analiza el siguiente código, que está incompleto, y contesta justificadamente las preguntas. (la función toupper pasa de minúsculas a mayúsculas). El fichero "pipeA" es una pipe. Asume que ejecutamos el programa (B) en línea de comandos de la siguiente forma (el fichero f es un fichero de datos con un texto de 1024 caracteres):

\$ B < f

Nombre alumno:

DNI:

```
1. void process_data()
2. {
3.     char c;
4.     int fd = open("pipeA", O_RDONLY);
5.     while(read(fd, &c, sizeof(char)) > 0){
6.         char uc = toupper(c);
7.         write(1, &uc, sizeof(char));
8.     }
9. }
10.
11. int main(int argc, char * argv[])
12. {
13.     int fd, ret, fd_out, c;
14.     char f_name[128], buffer[128];
15.
16.     sprintf(f_name, "f_out.%d", getpid());
17.
18.     fd_out = open(f_name, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
19.
20.     ret = fork();
21.     if (ret == 0){
22.         process_data();
23.     }
24.
25.     ret = fork();
26.     if (ret == 0){
27.         process_data();
28.     }
29.
30.     int turn = 0;
31.     fd = open("pipeA", O_WRONLY);
32.     while(read(0, &c, sizeof(c)) > 0){
33.         write(fd, &c, sizeof(c));
34.     }
35.
36.     while(waitpid(-1, NULL, 0) > 0);
37.     sprintf(buffer, "Ready\n");
38.     write(1, buffer, strlen(buffer));
39.
40. }
```

- a) (0,25 puntos) Dibuja la jerarquía de procesos que genera este código y el acceso a los ficheros y pipes que harían cada uno de los procesos.



Nombre alumno:

DNI:

- b) (0,25 puntos) ¿Deberíamos utilizar el flag O\_TRUNC en el open de la línea 18 para asegurar que el fichero de salida contiene únicamente el resultado de cada ejecución?

- c) (0,5 puntos) ¿Tendría algún efecto en el comportamiento si movemos el open de la línea 31 a la línea 17?

- d) (0,5 puntos) Si queremos que las escrituras de la línea 7 vayan al fichero que hemos abierto en la línea 18, indica que cambios propondrías en la función process\_data para garantizarlo.

- e) (1 punto) Rellena el estado de las tablas de E/S asumiendo que el proceso inicial está en el waitpid y que los hijos están creados y están justo al inicio del bucle de lectura. Añade las filas o tablas que necesites para reflejar el estado de todos los procesos. Ten en cuenta como se ha ejecutado el programa tal y como se indica al inicio del ejercicio. Incluye el cambio que hayas propuesto en la pregunta d. Las tablas muestran su contenido antes de que la shell cree el proceso que luego ejecutará el programa.

Nombre alumno:

DNI:

Tabla de Canales		Tabla de Ficheros abiertos				Tabla de iNodo	
Entrada TFA		refs	modo	Posición l/e	Entrada T.inodo	refs	inodo
0		0				0	
1		1				1	
2		2				2	
3		3				3	
4		4				4	
5		5				5	
6		6				6	
7		7				7	
8		8				8	
9		9				9	
10		10				10	
11		11				11	

- f) (0,5 puntos) ¿Debemos añadir alguna modificación a este código para asegurar que termina? En caso afirmativo indica que código y en que líneas.

Nombre alumno:

DNI:

## Examen final de teoría de SO

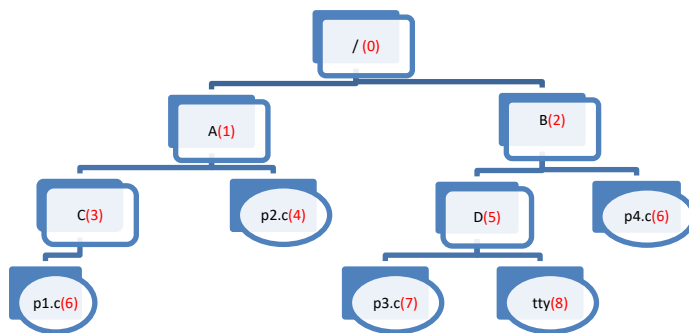
Justifica todas tus respuestas del examen. Las respuestas no justificadas se considerarán erróneas.

### Sistema de Ficheros (2 puntos)

Tenemos el siguiente sistema de ficheros basado en inodos. Datos:

<ul style="list-style-type: none"> <li>Los rectángulos representan directorios, los óvalos representan otros elementos</li> </ul>	<ul style="list-style-type: none"> <li>Las rutas destino de los softlinks se guardan en bloques de datos</li> </ul>
<ul style="list-style-type: none"> <li>El tamaño de bloque es de 4 KB</li> </ul>	<ul style="list-style-type: none"> <li>p1.c y p4.c son hardlinks al mismo fichero</li> </ul>
<ul style="list-style-type: none"> <li>p3.c es un soft-link a p2.c</li> </ul>	<ul style="list-style-type: none"> <li>p1.c ocupa 2 KB</li> </ul>
<ul style="list-style-type: none"> <li>p2.c ocupa 12 KB</li> </ul>	<ul style="list-style-type: none"> <li>tty es un dispositivo especial de carácter (una consola)</li> </ul>

- a) **(0,25 puntos)** asigna a los elementos anteriores un número de inodo. Indícalo justo a la derecha de cada figura. Usa el cuadro a la derecha para justificar tu asignación.



Todos los elementos están descritos por un inodo diferente, excepto p4.c y p1.c que son hardlinks al mismo fichero y por tanto comparten inodo

- b) **(0,75 puntos)** Completa las siguientes tablas con la información necesaria para representar la jerarquía anterior:

ID Inodo	0	1	2	3	4	5	6	7	8		
#enlaces											
Tipo	d	d	d	d	--	d	--	link	c		
Tabla de índices a BD	0	1	2	3	4,5,6	7	8	9	(vacío)		

ID BD	0	1	2	3	4	5	6	7	8	9	10
Contenido	. 0 .. 0 A 1 B 2	. 1 .. 0 C 3 p2.c 4	. 2 .. 0 D 5 P4.c 6	. 3 .. 1 P1.c 6	BD1 de p2.c	BD2 de p2.c	BD3 de p2.c	. 2 .. 0 p3.c 7 tty 8	B1 de p1.c y p4.c	/A/p2.c	

Nombre alumno:

DNI:

c) **(0,5 puntos)** Estando situados en el directorio /A ejecutamos el comando “**mkdir SubA**”.

Enumera los cambios que habría que hacer en las tablas anteriores (ya la jerarquía) para completar el comando:

- Solicitar al superbloque un nuevo inodo para directorio SubA (inodo 9)
- Solicitar al superbloque un nuevo bloque de datos para subA, que se tendrá que ocupar con dos entradas: “.” apuntando al nuevo inodo (9) y “..” al inodo del directorio padre (1)
- Crear el nombre en /A (subA apuntando al inodo 9) y actualizar el tamaño del directorio en su inodo (1) y las estadísticas de uso (fecha de modificación, usuario modificador)

d) **(0,5 puntos)** Analiza el siguiente código. Indica para cada línea qué accesos, consultas o modificaciones, se realizan a inodos y bloques de datos del sistema de ficheros. Supón que el sistema tiene **buffercaché de 1000 bloques vacía**. Indica que accesos serían en la buffer caché

```
1. int fd=open("/B/D/p3.c",O_RDWR);
2. int ret=lseek(fd,0,SEEK_END);
3. int w=write(fd,"a",1);
4. close(fd);
```

Sigue la siguiente nomenclatura como GUIA para escribir tu respuesta:

- I3C significa que se accede al Inodo 3 para Consulta
- B6M significa que se accede al Bloque de datos 6 para Modificarlo
- Subraya claramente los accesos que sean en buffercache

	Accesos y justificación
Línea 1	I0C + B0C + I2C + B2C + I5C + B7C + I7C + B9C + <u>I0C</u> + <u>B0C</u> + I1C + B1C + I4C
Línea 2	La llamada a sistema lseek en principio no accede ni a inodos ni a bloques, pero al hacer SEEK_END se debe consultar el I4 (I4C) para conocer el tamaño.
Línea 3	Como la escritura necesita un bloque nuevo (B10 p.e.) se solicita al superbloque, se acceden: B10M + I4M (para tamaño e índices de bloques, en la Tabla de inodos)
Línea 4	Si B10 no se ha guardado en disco por estar modificados solo en buffer caché, se debe hacer en este momento. Modificar la estadísticas de acceso (usuario, fecha) en el inodo (I4M) en el cierre si no se ha hecho en la línea anterior.

Nombre alumno:

DNI:

**Gestión de memoria (2 puntos)**

Analiza el siguiente código. Responde a la siguientes preguntas de forma razonada. El código se ejecuta sobre un sistema operativo Linux de 32 bits paginado, con tamaño de página de 4096 bytes, y **no** ofrece la optimización Copy-On-Write pero **sí** ofrece SWAP. El tamaño del tipo **int** es de 4 bytes. La región de código del programa ocupa 2400 bytes. Suponemos que el tamaño de las regiones para los datos viene dado únicamente por lo definido en este código. Supón que el programa “miprogram” existe, la ruta correcta y hay permiso para ejecutarlo.

```
int *a;
int b=4;

int main ()
{
    int c, ret;

    ret=fork();
    -----PUNTO A
    a=sbrk(sizeof(int);

    if (ret>0){
        waitpid(-1,NULL,0);
        a=sbrk(sizeof(int);
        *a=1;
        c=(*a)*b;
    }
    else if (ret==0)
        execlp("./miprogram", "miprogram", (char*)NULL);
    else return 1;
}
```

**A) (0,5 puntos) Indica y justifica cuánta memoria física ocupa por región posible (en marcos de página) la ejecución de este programa en el punto A.**

Al no haber COW se deben replicar y copiar el contenido de todas las secciones del proceso padre en el hijo:

- Sección de código: 2 frames (él código ocupa 2400 bytes que caben en 1 frame para el padre y otro para el hijo)
- Sección de pila: 2 frames (las variables c y ret ocupan 8 bytes en total que caben en un frame para el padre y otro para el hijo )
- Sección de datos: 2 frames (la variables a y b ocupan 8 bytes en total que caben en 1 frame para el padre y otro para el hijo)
- Heap: en el punto A no se usa el heap, por tanto ocupa 0 frames

**B) (0,25 puntos) Razona si añadir la optimización COW ofrecería alguna mejora en la gestión de memoria si hacemos avanzar el código desde el punto A hasta el punto B.**

Nombre alumno:

DNI:

Sí. Si hubiera COW el sistema se ahorraría duplicar las páginas del padre al hijo para inmediatamente después mutar y cambiar por completo el esquema de memoria del hijo (a pesar que junto después del fork habría que duplicar PILA, HEAP y DATOS). El ahorro consistiría en no tener que duplicar la página de código.

- C) (0, 5 puntos) Estando en el **punto A**, justifica si con el esquema de gestión de memoria de este sistema se produce algún tipo de fragmentación de memoria, que tipo de fragmentación sería, en caso afirmativo, y cuanta memoria se desaprovecharía.

En el punto A hay 2 procesos y no se dispone de COW. Al ser un sistema paginado se produce FRAGMENTACION INTERNA. La memoria desaprovechada por proceso es:

REGION CODIGO: 2400 bytes aprovechados en 1 página. Desaprovechado:  $4096 - 2400 = 1696B$

REGION DATOS: 1 puntero y 1 entero, aprovechado 8 bytes: desaprovechado  $4096 - 8 = 4088B$

REGIÓN PILA: 2 enteros, aprovechado 8 bytes, desaprovechado  $4096 - 8 = 4088B$

REGIÓN HEAP: no usada.

TOTAL desaprovechado= 2 procesos \* (1696+4088+4088) bytes

- D) (0,5 puntos) Al ejecutar la instrucción “**\*a=1**; ” se produce un MAJOR PAGE FAULT. Indica 3 razones que lo puedan provocar:

- La página de código que contiene la instrucción está en el SWAP
- La página de datos donde se encuentra al variable a está en el SWAP
- La página de heap que almacena el contenido de a está en el SWAP

- E) (0,25 puntos) Indica las operaciones que realizaría el sistema para solucionar el MAJOR PAGE FAULT

El sistema recibe la excepción de fallo de página, para resolverla debe bloquear el proceso, buscar la página correspondiente en el SWAP e intentar llevarla a memoria. Si la memoria estuviese llena, el SO debe seleccionar una página víctima y trasladarla al SWAP (reemplazo). Hay que actualizar la tabla de páginas del proceso con la nueva traducción (y eventualmente actualizar la TP del proceso víctima del reemplazo). Una vez hecho, se desbloquea el proceso y se reintenta la instrucción.

### Procesos y signals (3 puntos)

Analiza el siguiente código y contesta justificadamente a las preguntas. (se omite el control de errores por simplicidad). El programa “A” existe, es correcto, y no modifica nada relacionado con signals ni hace nada relevante para el ejercicio. El objetivo de este código es crear un nuevo proceso cada vez que se reciba un SIGUSR1 y terminar cuando se reciba un SIGUSR2.

Nombre alumno:

DNI:

```
1. uint sig1 = 0, sig2 = 0;
2. void create_process()
3. {
4.     int ret;
5.     ret = fork();
6.     if (ret == 0){
7.         execlp("./A", "A", NULL);
8.     }
9. }
10. void f_sig(int s)
11. {
12.     if (s == SIGUSR1) sig1 = 1;
13.     if (s == SIGUSR2) sig2 = 1;
14. }
15.
16. void main(int argc, char *argv[])
17. {
18.     struct sigaction new_action;
19.     sigset_t          action_mask;
20.
21.     new_action.sa_handler = f_sig;
22.     new_action.sa_flags   = SA_RESTART;
23.     sigfillset(&new_action.sa_mask);
24.     sigaction(SIGUSR1, &new_action, NULL);
25.     sigaction(SIGUSR2, &new_action, NULL);
26.
27.     sigfillset(&action_mask);
28.     sigdelset(&action_mask, SIGUSR1);
29.     sigdelset(&action_mask, SIGUSR2);
30.
31.     while(1){
32.         sigsuspend(&action_mask);
33.         if (sig1) create_process();
34.         if (sig2) exit(0);
35.         sig1 = 0;
36.         waitpid(-1, NULL, 0);
37.     }
38. }
39.
```

Si el programa se ejecuta en un terminal y devuelve el PID 1000, y en otro terminal ejecutamos la siguiente secuencia:

```
kill -USR1 1000
kill -USR1 1000
kill -USR1 1000
kill -USR2 1000
```

a) (0,5 puntos) ¿Qué pasaría si enviamos los dos SIGUSR1 antes de llegar al sigsuspend?

Nombre alumno:

DNI:

Si los signals llegan antes del sigaction, se ejecutará la acción por defecto que es acabar el proceso. Los signals no están bloqueados. Si llegan entre el sigaction y el sigpause, se ejecutará la acción de f\_sig pero no se cumplirá el objetivo del programa que es crear un proceso por cada SIGUSR1 y solo se creará 1 cuando llegue el tercer SIGUSR1

- b) (0,5 puntos) Si enviamos un signal cada minuto y el programa A tarda 5 segundos en ejecutarse, ¿Al recibir el SIGUSR2, cuantos procesos hijos tendrá activos o zombies el proceso 1000?

Ninguno porque tenemos un waitpid que garantiza que los procesos ya se habrán acabado y se habrá liberado el PCB. Las condiciones de la pregunta garantizan que haya tiempo suficiente.

- c) (0,5 puntos) Si movemos la llamada de la función create\_process() a la línea 12 (donde se gestiona la recepción del SIGUSR1), ¿Cuál sería la máscara de signals bloqueados de los nuevos procesos?

En ese punto el proceso tiene todos los signals bloqueados por lo que el proceso hijo también lo tendría.

- d) (0,5 puntos) ¿Es necesario modificar el código para evitar que el SIGINT, en ningún momento de la ejecución, termine la ejecución de proceso? Justifica la respuesta y en caso afirmativo indica el código que añadirías y en que líneas.

Si ya que el proceso no tiene ningún signal bloqueado Deberíamos añadir al principio un sigprocmask (línea 20). (También podríamos capturar el SIGINT con una función vacía)

```
sigset_t m;  
sigemptyset(&m)  
sigaddset(&m, SIGINT)  
sigprocmask(SIG_BLOCK, &m, NULL)
```

- e) (0,5 puntos) Si quisiéramos que la espera de los procesos hijos fuera asíncrona, ¿Qué cambios en el código habría que hacer? Enumera los cambios de forma resumida (lista de cambios).



Nombre alumno:

DNI:

- Capturar el SIGCHLD con un sigaction
- Mover el waitpid a la función de atención al SIGCHLD y hacerlo de modo no bloqueante WNOHANG e iterativo
- Permitir el SIGCHLD en el sigsuspend

f) (0,5 puntos) Si queremos hacer una mejora limitando el tiempo de ejecución de los procesos que ejecutan el programa A (por ejemplo 1000 segundos) utilizando signals, indica: (1) Si cambiarías el código del padre o de los hijos, (2) que signal elegirías, y (3) que llamada(s) a sistema añadirías. (solo se pide que termine, no que haga ninguna acción concreta)

Cambiaría el código de los hijos, pondría un alarm(1000) entre las líneas 6-7

### Procesos y pipes (3 puntos)

Analiza el siguiente código, que está incompleto, y contesta justificadamente las preguntas. (la función toupper pasa de minúsculas a mayúsculas). El fichero "pipeA" es una pipe. Asume que ejecutamos el programa (B) en línea de comandos de la siguiente forma (el fichero f es un fichero de datos con un texto de 1024 caracteres):

\$ B < f

Nombre alumno:

DNI:

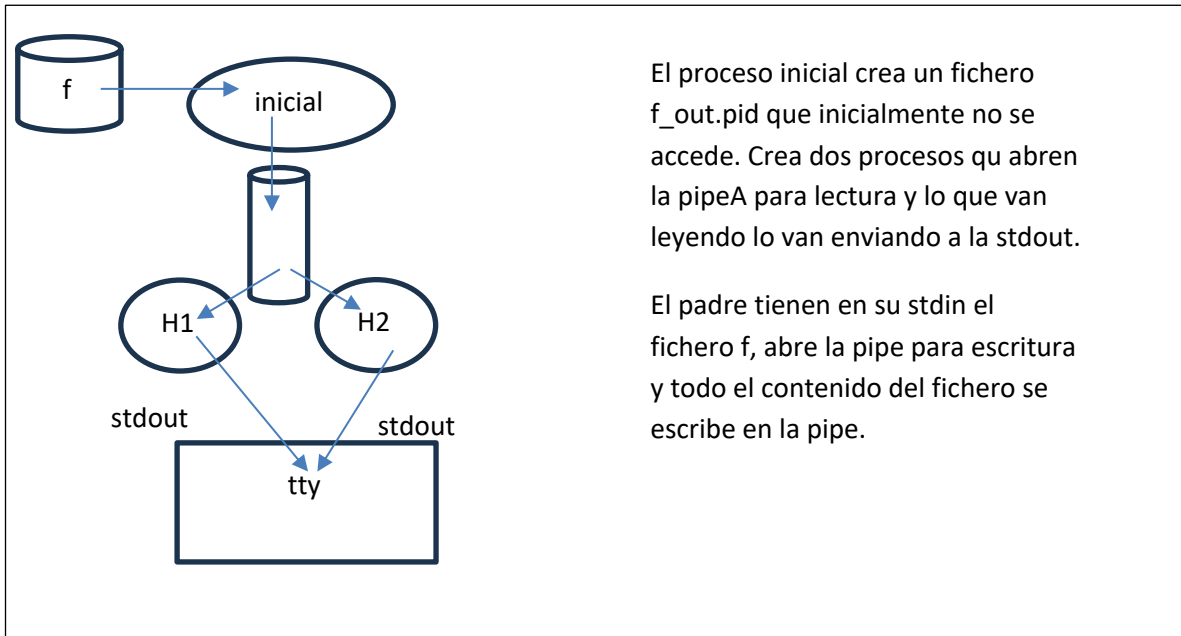
```
1. void process_data()
2. {
3.     char c;
4.     int fd = open("pipeA", O_RDONLY);
5.     while(read(fd, &c, sizeof(char)) > 0){
6.         char uc = toupper(c);
7.         write(1, &uc, sizeof(char));
8.     }
9.     exit(0);
10. }
11.
12. int main(int argc, char * argv[])
13. {
14.     int fd, ret, fd_out, c;
15.     char f_name[128], buffer[128];
16.
17.     sprintf(f_name, "f_out.%d", getpid());
18.
19.     fd_out = open(f_name, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
20.
21.     ret = fork();
22.     if (ret == 0){
23.         process_data();
24.     }
25.
26.     ret = fork();
27.     if (ret == 0){
28.         process_data();
29.     }
30.
31.
32.     fd = open("pipeA", O_WRONLY);
33.     while(read(0, &c, sizeof(c)) > 0){
34.         write(fd, &c, sizeof(c));
35.     }
36.
37.     while(waitpid(-1, NULL, 0) > 0);
38.     sprintf(buffer, "Ready\n");
39.     write(1, buffer, strlen(buffer));
40.
41. }
```

40.

- a) (0,25 puntos) Dibuja la jerarquía de procesos que genera este código y el acceso a los ficheros y pipes que harían cada uno de los procesos.

Nombre alumno:

DNI:



- b) (0,25 puntos) ¿Deberíamos utilizar el flag `O_TRUNC` en el `open` de la línea 18 para asegurar que el fichero de salida contiene únicamente el resultado de cada ejecución?

No ya que el nombre del fichero depende del PID y es un identificador por lo tanto es imposible que se repita

- c) (0,5 puntos) ¿Tendría algún efecto en el comportamiento si movemos el `open` de la línea 31 a la línea 17?

Si, el `open` de las pipes es bloqueante por lo que el proceso padre no continuaría su ejecución y no se llegarían a crear los procesos hijos

- d) (0,5 puntos) Si queremos que las escrituras de la línea 7 vayan al fichero que hemos abierto en la línea 18, indica que cambios propondrías en la función `process_data` para garantizarlo.

Haria un `dup2(fd_out,1)` (y pondría la variable `fd_out` global o la pasaríamos por parámetro)

Nombre alumno:

DNI:

- e) (1 punto) Rellena el estado de las tablas de E/S asumiendo que el proceso inicial está en el waitpid y que los hijos están creados y están justo al inicio del bucle de lectura. Añade las filas o tablas que necesites para reflejar el estado de todos los procesos. Ten en cuenta como se ha ejecutado el programa tal y como se indica al inicio del ejercicio. Incluye el cambio que hayas propuesto en la pregunta d. Las tablas muestran su contenido antes de que la shell cree el proceso que luego ejecutará el programa.

Tabla de Canales Entrada TFA		Tabla de Ficheros abiertos				Tabla de iNodo	
		refs	modo	Posición l/e	Entrada T.inodo	refs	inodo
0	1	0	3	rw	-	0	1 /dev/tty
1	2	1	3	r	1024	1	1 F
2	0	2	6	w	0	2	1 F_out.pid
3	2	3	1	w	-	3	3 pipeA
4	3	4	1	r	-	4	
5		5	1	r	-	5	
6		6				6	
7		7				7	
8		8				8	
9		9				9	
10		10				10	
11		11				11	

Tabla de Canales Entrada TFA		Tabla de Canales Entrada TFA	
0	1	0	1
1	2	1	2
2	0	2	0
3	2	3	2
4	4	4	5

El proceso inicial tiene en su stdin el fichero f, que es heredado por sus hijos. El canal de fd\_out también lo heredan y además cada uno de ellos hace una redirección a la stdout para que el write(1...) realmente escriba en el fichero. Por otro lado, el padre ya ha leído el contenido del fichero f, por lo que la pos l/e está en 1024. Los canales de la pipe son independiente en cada proceso ya que cada uno de ellos hace su propio open.

- f) (0,5 puntos) ¿Debemos añadir alguna modificación a este código para asegurar que termina? En caso afirmativo indica que código y en que líneas.

Nombre alumno:

DNI:

Si, como el padre no cierra el canal de escritura de la pipe, los procesos hijos se bloquean en el read. EL padre debe cerrar el canal de la pipe antes del bucle de waitpids. Asumiendo que queremos que haya dos procesos hijos, los hijos deberán hacer un exit al final de la función procesar\_data. En caso que hayais asumido que los hijos no han de acabar en el proces\_data la solución es mucho más compleja ya que habría que asegurar que el último proceso creado no se quede bloqueado indefinidamente en lectura en caso que el padre ya haya terminado.

Nombre alumno:

DNI:

## Examen final de teoría de SO

**Justifica todas tus respuestas del examen.** Las respuestas no justificadas se considerarán erróneas.

### Preguntas Cortas (2 puntos)

---

a) **(0,5 puntos)** Ejecutamos el siguiente comando:

```
$> ln A B
```

Sabiendo que “A” existe, “B” no existe, que disponemos permisos para crear elementos en el directorio actual y que hay espacio libre en el disco, el comando falla. ¿A qué puede ser debido?

b) **(0,5 puntos)** Define el concepto de fragmentación externa de memoria, indicando en qué modelo de gestión de memoria se puede producir.

Nombre alumno:

DNI:

c) **(0,5 puntos)** En un programa se ejecuta la siguiente línea:

```
ret = read(fd, buffer, 100);
```

Asumiendo que **fd** apunta a un dispositivo virtual válido y que **buffer** está bien declarado y tiene tamaño suficiente, indica al menos un caso en que tras finalizar **read** la variable **ret** tenga el siguiente intervalo de valores:  $1 \leq \text{ret} \leq 99$

d) **(0,5 puntos)** Supón que conoces la dirección de entrada de la rutina de kernel que implementa el servicio de la llamada a sistema write. ¿Se podría hacer un *call* directamente a esa rutina desde una aplicación de usuario, asumiendo que pasamos correctamente los parámetros?

Nombre alumno:

DNI:

**Gestión de memoria (1 Punto)**

Tenemos el siguiente código del programa "memoria". Este programa se ejecuta en un sistema Linux con tamaño de página 4096 bytes y nuestro sistema implementa la optimización de COW.

```
1. #define M_SIZE 4096
2. void print_limit()
3. {
4.     char buffer[256];
5.     void* limit;
6.     limit = sbrk(0);
7.     sprintf(buffer, "Limite %p\n", limit);
8.     write(1, buffer, strlen(buffer));
9. }
10. void main(int argc, char *argv[])
11. {
12.     int ret, i, *p1;
13.     print_limit();
14.     p1 = sbrk(M_SIZE * sizeof(int));
15.     print_limit();
16.     for (i = 0; i < M_SIZE; i++) p1[i] = 0;
17.     ret = fork();
18.     /* A */
19.     sbrk(-1 * M_SIZE * sizeof(int));
20.     print_limit();
21. }
```

Al ejecutarlo tenemos la siguiente salida:

```
[user@login]$ ./memoria
Limite 0x1971000
Limite 0x1975000
Limite XXXXXXXXX
Limite YYYYYYYYY
```

Contesta a las siguientes preguntas:

a) Rellena las líneas punteadas con la respuesta correcta.

1. La variable p1 está en la región de memoria ....., la variable limit está en la región de memoria .....
2. El valor que veremos en las XXXXXXXX es ..... y en las YYYYYYYY es .....

b) Si nos dan la siguiente información: el tamaño de la pila es 4096 bytes, el código de este programa ocupa 1000 bytes, la variable p1 ocupa 8 bytes y el tamaño de un int son 4 bytes:

1. ¿Cuántos marcos de página (páginas físicas) tendrán reservados para las regiones de pila en el punto A incluyendo los dos procesos?



Nombre alumno:

DNI:

2. ¿Cuántos marcos de página (páginas físicas) tendrán reservados para las regiones de heap en el punto A incluyendo los dos procesos?

Nombre alumno:

DNI:

**Procesos y Signals (3 Puntos)**

Analiza el código que te mostramos a continuación y responde a las preguntas de la manera más detallada posible dentro del espacio de que dispones. Supón que ejecutamos el programa desde el terminal con la siguiente línea de comandos: \$ ./a.out 2

- a) **(0,5 puntos)** Dibuja la jerarquía de procesos creada. Etiquétalos para poder referirte a ellos durante el resto de tu respuesta.

```
1. int beep = 0;
2. void ras(int s) {
3.     sigset_t nores, m;
4.     sigfillset(&nores);
5.     if (s == SIGALRM) {
6.         write(1, " FINAL!\n", 8);
7.         beep++;
8.     }
9.     if (beep < 1) {
10.        sigprocmask(SIG_SETMASK, &nores, &m);
11.        sigdelset(&m, SIGALRM);
12.        sigsuspend(&m);
13.    }
14.}
15.

16.int main(int argc, char *argv[]) {
17.    int i;
18.    struct sigaction sa;
19.    int n = atoi(argv[1]);
20.    sa.sa_handler = ras;
21.    sigfillset(&sa.sa_mask);
22.    sa.sa_flags = SA_RESTART;
23.    for (i = 0; i < 32; i++) {
24.        sigaction(i, &sa, NULL);
25.        alarm(i);
26.    }
27.    for (i = 0; i < n; i++) fork();
28.    while (waitpid(-1, NULL, 0) > 0);
29.    write(1, "\tEXAMEN!", 8);
30.    exit(0);
31.}
```

- b) **(0,5 puntos)** ¿Qué signals están bloqueados cuando comenzamos a ejecutar la función ras()?

Nombre alumno:

DNI:

c) **(1,5 puntos)** ¿Qué procesos escriben "EXAMEN!"? ¿Y "FINAL!"?

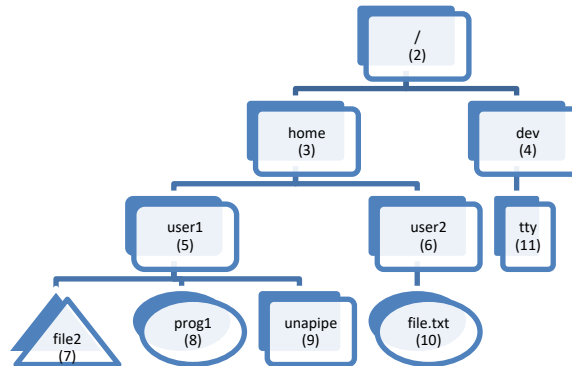
d) **(0,5 puntos)**. ¿Qué línea de comandos tendrías que ejecutar para que todos los procesos llegasen a la línea 29, escribiendo por pantalla?

Nombre alumno:

DNI:

**Entrada/Salida (4 Puntos)**

Tenemos el siguiente SF basado en I-Nodos, con un tamaño de bloque de 1KB.



El fichero “file.txt” es un fichero de caracteres que contiene 2KB de datos. El fichero “file2” es un soft-link que apunta /home/user2/file.txt mediante un path absoluto. El fichero “prog1” es un ejecutable que ocupa menos de un bloque y es el resultado de compilar el siguiente código fuente:

```

1. main() {
2.   char c;
3.   int ret, fd;
4.   ret = fork();
5.   if (ret == 0) {
6.     fd=open("/home/user1/unapipe", O_WRONLY);
7.     while (read(0, &c, sizeof(c)) > 0)
8.       write(fd, &c, sizeof(c));
9.     exit(0);
10.  }
11. fd=open("/home/user1/unapipe", O_RDONLY);
12. while(read(fd, &c, sizeof(c)) > 0)
13.   write(1, &c, sizeof(c));
14. waitpid(-1, NULL, 0);
15. }
  
```

- a) **(0,75 puntos)** Completa las siguientes tablas con la información que falta para representar esta jerarquía. El campo “path” sólo se utiliza para los soft links y contiene el path del fichero referenciado.

ID Inodo	2	3	4	5	6	7	8	9	10	11	12
#enlaces						1	1	1	1	1	
Tipo	d	d	d	d	d	l	-	p	-	c	
Path	-	-	-	-	-	/home/user2/file.txt	-	-	-	-	
Tabla de índices a BD	0	1	2	3	4		5		6,7		

ID BD	0	1	2	3	4	5	6	7	8	9
Contenido	. 2 .. 2 home dev	. .. user1 user2				codigo	datos	datos		

Nombre alumno:

DNI:

b) **(3,25 puntos)** Supón que el directorio actual de trabajo es /home/user1 y que ejecutamos el siguiente comando: `./prog1 < /home/user1/file2 > /home/user1/file3.txt`

1. **(0,5 puntos)** Completa el estado de las siguientes estructuras de datos suponiendo que los procesos se encuentran justo después del fork (entre la línea 4 y 5).

Tabla de Canales		Tabla de Ficheros abiertos				Tabla de iNodo	
Entrada TFA		refs	modo	Posición l/e	Entrada T.inodo	refs	inodo
0		0				0	1
1		1	rw	-	0	1	l tty1
2		2				2	
3		3				3	
4		4				4	

2. **(0,5 puntos)** Describe brevemente lo que hace este comando. ¿Acabará la ejecución? Justifica tu respuesta

3. **(0,5 puntos)** Indica cómo quedarán las estructuras de datos del apartado a) cuando el bucle de la línea 11 haya completado 2048 iteraciones. Si necesitas utilizar nuevos inodos o nuevos bloques asígnalos secuencialmente.

ID Inodo	2	3	4	5	6	7	8	9	10	11	12
#enlaces						1	1	1	1	1	
Tipo	d	d	d	d	d	l	-	p	-	c	
Path	-	-	-	-	-	/home/user2/file.txt	-	-	-	-	
Tabla de índices a BD	0	1	2	3	4		5		6,7		

ID BD	0	1	2	3	4	5	6	7	8	9
Contenido	. 2 .. 2 home dev	. .. user1 user2				codig o	datos	datos		

Justificación:



Nombre alumno:

DNI:

## Examen de teoría de SO

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

### Preguntas Cortas (2 puntos)

1. (0,5 puntos) Ejecutamos el siguiente comando:

```
$> ln A B
```

Sabiendo que “A” existe, “B” no existe, que disponemos permisos para crear elementos en el directorio actual y que hay espacio libre en el disco, el comando falla. ¿A qué puede ser debido?

Porque “A” es un directorio y no podemos crear hardlinks entre directorios.

2. (0,5 puntos) Define el concepto de fragmentación externa de memoria, indicando en qué modelo de gestión de memoria se puede producir.

Es memoria libre y no asignada pero no se puede asignar a un proceso por no estar contigua. No está reservada pero no sirve. Se produce en el modelo de segmentación.

3. (0,5 puntos). En un programa se ejecuta la siguiente línea:

```
ret = read(fd, buffer, 100);
```

Asumiendo que **fd** apunta a un dispositivo virtual válido y que **buffer** está bien declarado y tiene tamaño suficiente, indica al menos un caso en que tras finalizar **read** la variable **ret** tenga el siguiente intervalo de valores:  $1 \leq \text{ret} \leq 99$

- Cuando el dispositivo asociado a **fd** solo dispone de entre 1 y 99 bytes disponibles para lectura.
- Cuando, mientras se ejecuta la transferencia de datos al buffer el proceso recibe un signal capturado donde el flag **SA\_RESTART** está desactivado.

Nombre alumno:

DNI:

4. (0,5 puntos) Supón que conoces la dirección de entrada de la rutina de kernel que implementa el servicio de la llamada a sistema write. ¿Se podría hacer un *call* directamente a esa rutina desde una aplicación de usuario, asumiendo que pasamos correctamente los parámetros?

- No es posible. Desde el espacio lógico de un proceso de usuario no es accesible el espacio de direcciones del kernel.

### Gestión de memoria (1 Punto)

Tenemos el siguiente código del programa "memoria". Este programa se ejecuta en un sistema Linux con tamaño de página 4096 bytes y nuestro sistema implementa la optimización de COW.

```
1. #define M_SIZE 4096
2. void print_limit()
3. {
4.     char buffer[256];
5.     void* limit;
6.     limit = sbrk(0);
7.     sprintf(buffer, "Limite %p\n", limit);
8.     write(1, buffer, strlen(buffer));
9. }
10. void main(int argc, char *argv[])
11. {
12.     int ret, i, *p1;
13.     print_limit();
14.     p1 = sbrk(M_SIZE * sizeof(int));
15.     print_limit();
16.     for (i = 0; i < M_SIZE; i++) p1[i] = 0;
17.     ret = fork();
18.     /* A */
19.     sbrk(-1 * M_SIZE * sizeof(int));
20.     print_limit();
21. }
```

Al ejecutarlo tenemos la siguiente salida:

```
[user@login]$ ./memoria
Limite 0x1971000
Limite 0x1975000
Limite XXXXXXXXX
Limite YYYYYYYYY
```

Contesta a las siguientes preguntas:

1. Rellena las líneas punteadas con la respuesta correcta.
  - a. La variable p1 está en la región de memoria .....pila/stack....., la variable limit está en la región de memoria .....pila/stack.....



Nombre alumno:

DNI:

- b. El valor que veremos en las XXXXXXXX es .....0x1971000..... y en las YYYYYYYY es .....0x1971000.....
2. Si nos dan la siguiente información: el tamaño de la pila es 4096 bytes, el código de este programa ocupa 1000 bytes, la variable p1 ocupa 8 bytes y el tamaño de un int son 4 bytes:

- a. ¿Cuántos marcos de página (páginas físicas) tendrán reservados para las regiones de pila en el punto A incluyendo los dos procesos?

En ese punto la pila ya no se comparte ya que tanto i como ret se han modificado. En total 1 página por proceso = 2 páginas.

- b. ¿Cuántos marcos de página (páginas físicas) tendrán reservados para las regiones de heap en el punto A incluyendo los dos procesos?

Se ha reservado 4 páginas (4096 x 4). El padre inicializa los datos, por lo que se reservan 4 páginas. Al hacer fork, como se aplica COW y ni padre ni hijo modifican los datos, entre los dos tendrán 4 páginas.

### Procesos y Signals (3 Puntos)

Analiza el código que te mostramos a continuación y responde a las preguntas de la manera más detallada posible dentro del espacio de que dispones. Supón que ejecutamos el programa desde el

```

1. int beep = 0;
2. void ras(int s) {
3.     sigset_t nores, m;
4.     sigfillset(&nores);
5.     if (s == SIGALRM) {
6.         write(1, " FINAL!\n", 8);
7.         beep++;
8.     }
9.     if (beep < 1) {
10.        sigprocmask(SIG_SETMASK, &nores, &m);
11.        sigdelset(&m, SIGALRM);
12.        sigsuspend(&m);
13.    }
14.}
15.

16.int main(int argc, char *argv[]) {
17.    int i;
18.    struct sigaction sa;
19.    int n = atoi(argv[1]);
20.    sa.sa_handler = ras;
21.    sigfillset(&sa.sa_mask);
22.    sa.sa_flags = SA_RESTART;
23.    for (i = 0; i < 32; i++) {
24.        sigaction(i, &sa, NULL);
25.        alarm(i);
26.    }
27.    for (i = 0; i < n; i++) fork();
28.    while (waitpid(-1, NULL, 0) > 0);
29.    write(1, "\tEXAMEN!", 8);
30.    exit(0);
31.}

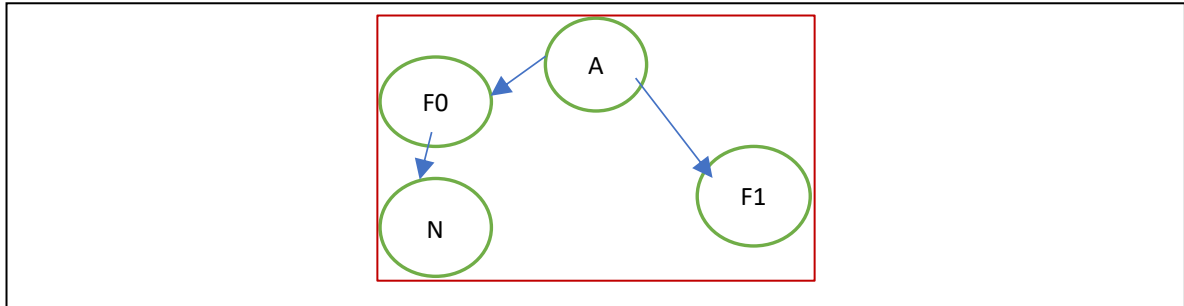
```

terminal con la siguiente línea de comandos: \$ ./a.out 2

- a) **(0,5 puntos)** Dibuja la jerarquía de procesos creada. Etiquétalos para poder referirte a ellos durante el resto de tu respuesta.

Nombre alumno:

DNI:



- b) **(0,5 puntos)** ¿Qué signals están bloqueados cuando comenzamos a ejecutar la función `ras()`?

Todos. En la línea 21, llenamos la máscara `sa.sa_mask` y, en la línea 24, la fijamos como máscara de signals del proceso mientras se ejecute la rutina de atención al signal. En la línea 10, inicializamos la `m` con la máscara de signals actual del proceso.

- c) **(1,5 puntos)** ¿Qué procesos escriben "EXAMEN! "? ¿Y "FINAL! "?

EXAMEN lo escriben, enseguida, los procesos N y F1 (ie, las hojas del árbol, que no tienen hijos). Esto provoca que F0 y A (los padres respectivos) reciban un SIGCHLD y queden suspendidos en línea 12 esperando un SIGALRM. Sólo A tiene programado un temporizador y, pasados 32 segundos, lo recibirá y escribirá FINAL. Entonces A queda esperando, en la línea 28, la muerte de F0. Pero F0 nunca morirá (a menos que reciba un SIGALRM desde otro proceso).

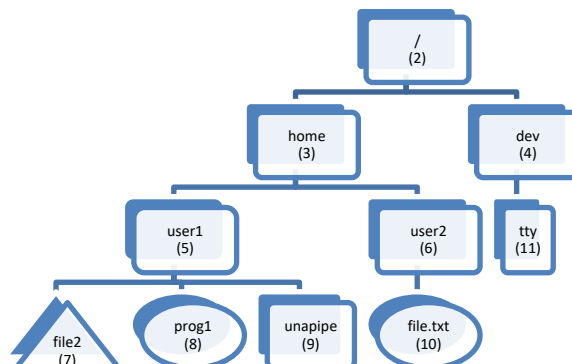
- a) **(0,5 puntos)**. ¿Qué línea de comandos tendrías que ejecutar para que todos los procesos llegasen a la línea 29, escribiendo por pantalla?

```
$ kill -ALRM pidF0
```

Cuando F0 reciba un SIGALRM, saldrá del `sigsuspend` y escribirá FINAL. Como su hijo ya está muerto y `beep` vale 1, no se bloqueará en la línea 28, ni en la 12. Escribirá EXAMEN y morirá. Entonces su padre, A, saldrá del `waitpid` de la línea 28. Tampoco entrará en el `if` de la línea 9, porque su variable `beep` vale 1. Escribirá EXAMEN y morirá.

### Entrada/Salida (4 Puntos)

Tenemos el siguiente SF basado en I-Nodos, con un tamaño de bloque de 1KB.



Nombre alumno:

DNI:

El fichero “file.txt” es un fichero de caracteres que contiene 2KB de datos. El fichero “file2” es un soft-link que apunta /home/user2/file.txt mediante un path absoluto. El fichero “prog1” es un ejecutable que ocupa menos de un bloque y es el resultado de compilar el siguiente código fuente:

```

1. main() {
2.     char c;
3.     int ret, fd;
4.     ret = fork();
5.     if (ret == 0) {
6.         fd=open("/home/user1/unapipe",O_WRONLY);
7.         while (read(0,&c,sizeof(c))>0)
8.             write(fd,&c,sizeof(c));
9.         exit(0);
10.    }
11.    fd=open("/home/user1/unapipe",O_RDONLY);
12.    while(read(fd,&c,sizeof(c)) > 0)
13.        write(1,&c,sizeof(c));
14.    waitpid(-1,NULL,0);
15. }
```

- d) **(0,75 puntos)** Completa las siguientes tablas con la información que falta para representar esta jerarquía. El campo “path” sólo se utiliza para los soft links y contiene el path del fichero referenciado.

ID Inodo	2	3	4	5	6	7	8	9	10	11
#enlaces	4	4	2	2	2	1	1	1	1	1
Tipo	d	d	d	d	d	l	-	p	-	c
Path	-	-	-	-	-	/home/user2/file.txt	-	-	-	-
Tabla de índices a BD	0	1	2	3	4		5		6,7	

ID BD	0	1	2	3	4	5	6	7	8	9
Contenido	. 2 .. 2 home 3 dev 4	. 3 .. 2 user1 5 user2 6	. 4 .. 2 tty 11	. 5 .. 3 file2 7 prog1 8 unapipe 9	. 6 .. 3 file.txt 10	codigo	datos	datos		

- e) **(3,25 puntos)** Supón que el directorio actual de trabajo es /home/user1 y que ejecutamos el siguiente comando:

```
%. /prog1 < /home/user1/file2 > /home/user1/file3.txt
```

Nombre alumno:

DNI:

- 1) **(0,5 puntos)** Completa el estado de las siguientes estructuras de datos suponiendo que los procesos se encuentran justo después del fork (entre la línea 4 y 5).

Tabla de Canales

Entrada TFA

0	1
1	2
2	0
3	
4	

Tabla de Ficheros abiertos

refs	modo	Posición I/e	Entrada T.inodo
0	2	rw	-
1	2	r	0
2	2	w	0
3			
4			

Tabla de iNodo

refs	inodo
0	1
1	1
2	1
3	
4	

Tabla de Canales

Entrada TFA

0	1
1	2
2	0
3	
4	

Tabla de Ficheros abiertos

refs	modo	Posición I/e	Entrada T.inodo
0	2	rw	-
1	2	r	0
2	2	w	0
3			
4			

Tabla de iNodo

refs	inodo
0	1
1	1
2	1
3	
4	

- 2) **(0,5 puntos)** Describe brevemente lo que hace este comando. ¿Acabará la ejecución? Justifica tu respuesta

La ejecución del comando provocará la creación de /home/user1/file3.txt con una copia de /home/user2/file.txt.

Acaba la ejecución. El proceso hijo acabará cuando haya completado la lectura de file1.txt. En ese momento, desaparecerá el único canal de escritura abierto para la pipe. El padre cuando vacíe la pipe, detectará que no quedan escritores y también saldrá de su bucle y acabará la ejecución.

- 3) **(0,5 puntos)**. Indica cómo quedarán las estructuras de datos del apartado a) cuando el bucle de la línea 11 haya completado 2048 iteraciones. Si necesitas utilizar nuevos inodos o nuevos bloques asígnalos secuencialmente.

ID Inodo	2	3	4	5	6	7	8	9	10	11	12
#enlaces	4	4	2	2	2	1	1	1	1	1	1
Tipo	d	d	d	d	d	l	-	p	-	c	-

Nombre alumno:

DNI:

Path	-	-	-	-	-	/home/user2/file.txt	-	-	-	-	-
Tabla de índices a BD	0	1	2	3	4		5		6,7		8,9

ID BD	0	1	2	3	4	5	6	7	8	9
Contenido	. 2 .. 2 home 3 dev 4	. 3 .. 2 user1 5 user2 6	. 4 .. 2	. 5 .. 3 file2 7 prog1 8  unapipe 9  file3.txt 12	. 6 .. 3 file.txt 10	codi go	datos	datos	COPIA DE DATOS DE 6	COPIA DE DATOS DE 7

**Justificación:** La creación de un nuevo fichero y la copia del contenido implica reservar un nuevo inodo, reservar dos nuevos bloques que contendrán lo mismo que los bloques de file.txt y añadir una nueva entrada en el directorio /home/user1

- 4) **(0,5 puntos)** Rellena la siguiente tabla indicando qué bloques de datos y qué inodos accederán los procesos (independientemente de si están en disco o es una copia en alguna estructura de datos en memoria) durante la ejecución de las siguientes sentencias del código.

Sentencia	Bloques e inodos accedidos	Justificación
7. read(0, &c, 1)	Bloque 6 y 7, Inodo 10	Lee todos los bloques de file.txt. La información sobre cuáles son esos bloques está en el inodo del fichero.
12. write(1,&c,1)	Bloque 8,9, Inodo 12	Escribe en los bloques asignados a file3.txt. La información sobre qué bloques son está en el inodo del fichero

- 5) **(0,5 puntos)** La línea 6 (`open("/home/user1/unapipe",...);`), ¿necesita acceder a algún bloque de datos? ¿Y a algún inodo? De ser así, indica la secuencia de accesos que realizará. En cualquier caso, justifica tu respuesta.

Nombre alumno:

DNI:

(superbloque) I2,B0,I3,B1,I5,B3,I9

- 6) **(0,75 puntos)** Indica cuáles de las siguientes sentencias modificarán la tabla de ficheros abiertos y cuáles modificarán la tabla de inodos (modificar puede ser añadir nueva entrada o cambiar el contenido de una ya existente). En la justificación describe los cambios que harán.

Sentencia	Modifica TFA (si/no)	Modifica T. Inodos (si/no)
6. fd=open("/home/user1/unapipe",O_WRONLY);	Si	Si
8. write(fd,&c,sizeof(c));	No	Si
11. fd=open("/home/user1/unapipe",O_RDONLY);	Si	Si
13. write(1,&c,sizeof(c));	Si	Si

**Justificación**

6 y 11: Cada open crea una nueva entrada en la TFA. Si el dispositivo no estaba en uso, crea una nueva entrada en la T. Inodos, si ya estaba en uso incrementa el número de referencias del inodo.

8. La escritura en una pipe no afecta a la entrada de la TFA pero deberemos actualizar en el inodo la fecha de último acceso

13. La escritura en un fichero modifica el puntero de lectura/escritura en la TFA. Además estamos haciendo crecer el fichero, así que en el inodo hay que actualizar tamaño, bloques (cuando se añaden) y fecha de último acceso.

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.

Nombre alumno:

DNI:

- 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.

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

- 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?

- 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?

- 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?



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?

b) **(0,5 puntos)** ¿Este proceso tiene fragmentación de memoria? Si la respuesta 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.

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.

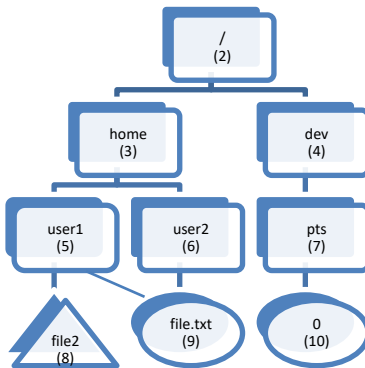
d) **(0,5 puntos)** Supón que este proceso en lugar de la llamada *fork* ejecuta la llamada a sistema *exec* para mutar al mismo ejecutable (a sí mismo). La ejecución de esta llamada (sin tener en cuenta la ejecución de ninguna instrucción posterior), ¿provocará que cambie la cantidad de memoria física?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

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 asumir 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											
Tipo	dir	dir	dir	dir	dir	dir	link	-	-		
BDs	0								-		

ID BD	0	1	2	3	4	5	6	7	8	9	10
Datos											

Nombre alumno:

DNI:

- 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 l-nodo da/n la información necesaria para identificar de manera única las dos terminales que tenemos en la carpeta “dev”.

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*

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: ..."

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

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?

- 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()				
2	open("PIPE1", O_WRONLY)				
3	execlp("./endpoint", "./endpoint", 0)				
4	close(npd)				
5	waitpid(-1, NULL, 0)				

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:

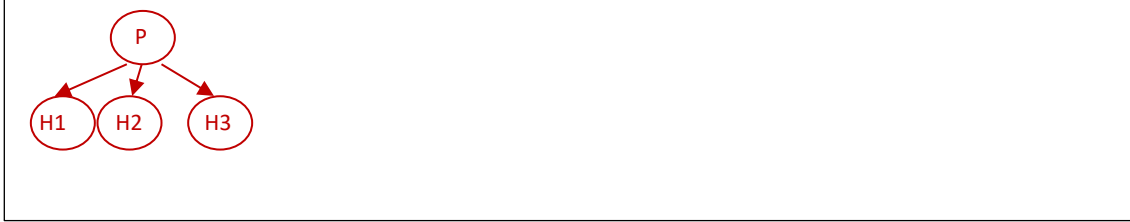
```
% ./examen
```

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 creación 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 respuesta 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.

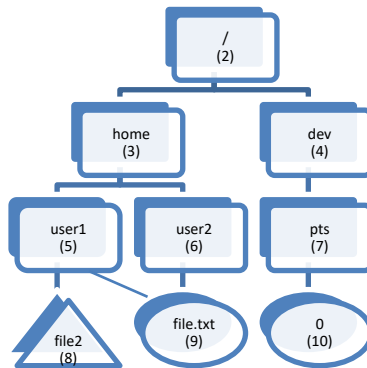


Nombre alumno:

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 asumir 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), mayor (qué tipo de dispositivo) y menor (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 línea 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	

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 (2 puntos)

---

1. (0,5 puntos) Explica brevemente qué operaciones realiza la siguiente llamada (asume que no hay optimizaciones de memoria):

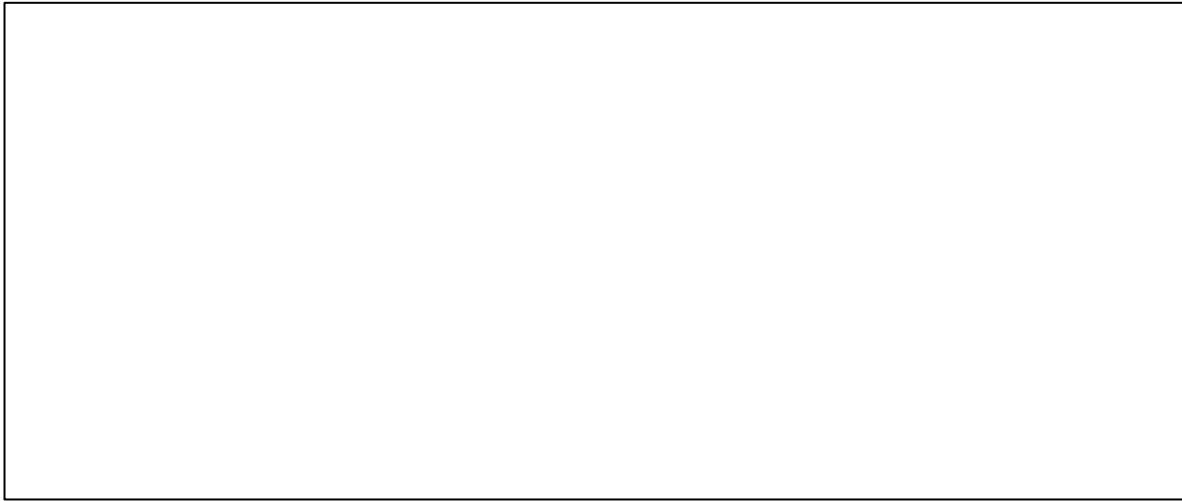
```
int *ptr = malloc(2000);
```

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

Nombre alumno:

DNI:

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



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



Nombre alumno:

DNI:

**Gestión de procesos (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:

<pre> /* 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(&amp;m);     sigaddset(&amp;m, SIGUSR1);     sigprocmask(SIG_BLOCK, &amp;m, NULL);      levels = atoi(argv[1]);     for (i = 0; i &lt; levels; i++){         pids[i] = fork();         if (pids[i] &gt; 0){             sprintf(curr_level, "%d", i + 1);             execlp("./compute", "compute", curr_level,                 NULL);             error_y_exit("Error execlp", 2);         } else if (pids[i] &lt; 0){             error_y_exit("Error fork", 2);         }     }      kill(getppid(), SIGUSR1);      while((ret = waitpid(-1, &amp;ec, 0)) &gt; 0){         trataExitCode( ret, ec);     }     exit(0); } </pre>	<pre> /* 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(&amp;m);      sa.sa_handler = f_usr1;     sigemptyset(&amp;sa.sa_mask);     sa.sa_flags = 0;     sigaction(SIGUSR1, &amp;sa, NULL);      sigaddset(&amp;m, SIGUSR1);     sigprocmask(SIG_UNBLOCK, &amp;m, NULL);      while(sig_usr1 == 0);      for (i = 0; i &lt; 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] &lt; 0){             error_y_exit("Error fork", 2);         }     }      while(waitpid(-1, NULL, 0) &gt; 0 );     kill( getppid(), SIGUSR1);     exit(atoi(argv[1])); } </pre>
--	--

1. (0.75 puntos) Dibuja la jerarquía 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.

Nombre alumno:

DNI:

2. (0,25 puntos) ¿Qué podría pasar si eliminamos el sigprocmask de los dos programas?

3. (0,25 puntos) ¿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`?

4. (0,5 puntos) 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.

5. (0,25 puntos) El bucle que ejecuta la función `trataExitCode`, ¿Cuántos y qué mensajes escribirá?



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?

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		Tabla de Ficheros abiertos				Tabla de iNodo	
Entrada TFA		refs	modo	Posición l/e	Entrada T.inodo	refs	inodo
0		0				0	
1		1				1	
2		2				2	
3		3				3	
4		4				4	
5		5				5	
		6					

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

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	.	2	.	3	.	4	Texto	.	7	Texto	Texto				
	..	2	..	2	..	2		..	4						
	A	3	c	5	E	7		g	8						
	B	4	d	6	f	8									

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.

- (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.

- 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|O_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.

- (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:****Justificación:**

- 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
<code>ret=read(fdr,buf,sizeof(buf))</code>			
<code>write(fdw,buf,ret);</code>			
<code>unlink("/A/c/g");</code>			

**Justificación:**

- 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				
#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	. 2 .. 2 A 3 B 4	. 3 .. 2 c 5 d 6	. 4 .. 2 E 7 f 8	Texto	. 7 .. 4 g 8	Texto	Texto				

**Justificación**

Nombre alumno:

DNI:

**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)**

<pre> 1. int res[2048]; 2. int *ptr; 3. main(){ 4.     int A[2048], B[2048], i; 5.     ptr = res; 6.     for (i=0;i&lt;2048;i++) 7.         A[i] = B[i] = i; 8.     for (i=0;i&lt;2048;i++) 9.         ptr[i] = A[i] + B[i]; 10. }</pre>	Región	Páginas asignadas	Memoria Física (KB y/o B)
	Data		
	Stack		
	Heap		
<b>JUSTIFICACIÓN:</b>			

**2. (0,5 puntos)**

<pre> 1. int *ptr; 2. main(){ 3.     int A[2048], B[2048], i; 4.     ptr = sbrk(2048 * sizeof(int)); 5.     for (i=0;i&lt;2048;i++) 6.         A[i] = B[i] = i; 7.     for (i=0;i&lt;2048;i++) 8.         ptr[i] = A[i] + B[i]; 9. }</pre>	Región	Páginas asignadas	Memoria Física (KB y/o B)
	Data		
	Stack		
	Heap		
<b>JUSTIFICACIÓN:</b>			

**3. (1 punto)**

<pre> 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&lt;2048;i++) 8.         A[i] = B[i] = i; 9.     fork(); 10.    for (i=0;i&lt;2048;i++) 11.        ptr[i] = A[i] + B[i]; 12. }</pre>	Región	Páginas asignadas	Memoria Física (KB y/o B)
	Data		
	Stack		
	Heap		
<b>JUSTIFICACIÓN:</b>			

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 `trataExitCode` son las usadas durante el curso. Asume también que el vector de 10 pids es suficiente. Analiza los códigos y responde a las preguntas:

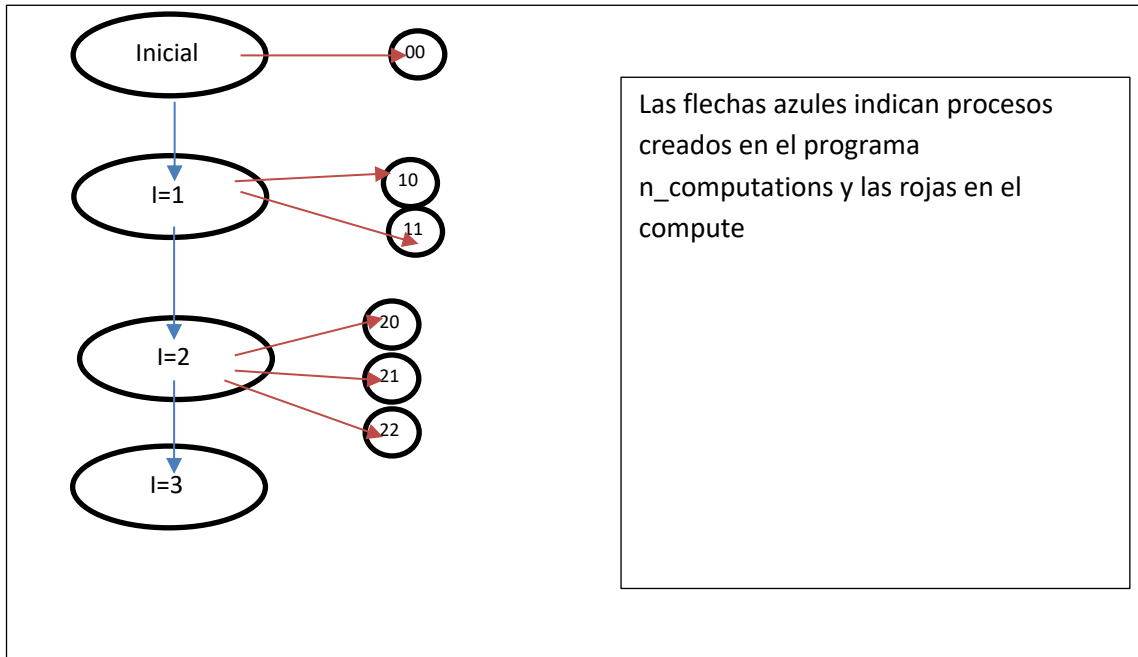
<pre> /* n_computation */ void f_alarm(int s) { }  void main(int argc, char *argv[]) {     int i, ec, ret;     pid_t pids[10];     uint levels;     char curr_level[64], buffer[128];     if (argc != 2) usage(argv[0]);     struct sigaction sa;     sigset_t m;      sigemptyset(&amp;m);     sigaddset(&amp;m, SIGUSR1);     sigprocmask(SIG_BLOCK, &amp;m, NULL);      levels = atoi(argv[1]);     for (i = 0; i &lt; levels; i++){         pids[i] = fork();         if (pids[i] &gt; 0){             sprintf(curr_level, "%d", i + 1);             execlp("./compute", "compute", curr_level,                 NULL);             error_y_exit("Error execlp", 2);         } else if (pids[i] &lt; 0 ){             error_y_exit("Error fork", 2);         }     }      kill(getppid(), SIGUSR1);      while((ret = waitpid(-1, &amp;ec, 0)) &gt; 0){         trataExitCode( ret, ec);     }     exit(0); } </pre>	<pre> /* 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(&amp;m);      sa.sa_handler = f_usr1;     sigemptyset(&amp;sa.sa_mask);     sa.sa_flags = 0;     sigaction(SIGUSR1, &amp;sa, NULL);      sigaddset(&amp;m, SIGUSR1);     sigprocmask(SIG_UNBLOCK, &amp;m, NULL);      while(sig_usr1 == 0);      for (i = 0; i &lt; 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] &lt; 0 ){             error_y_exit("Error fork", 2);         }     }      while(waitpid(-1, NULL, 0) &gt; 0 );     kill( getppid(), SIGUSR1);     exit(atoi(argv[1])); } </pre>
---	---

- a) (0.75 puntos) Dibuja la jerarquía de procesos que se genera al ejecutar 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:



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

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

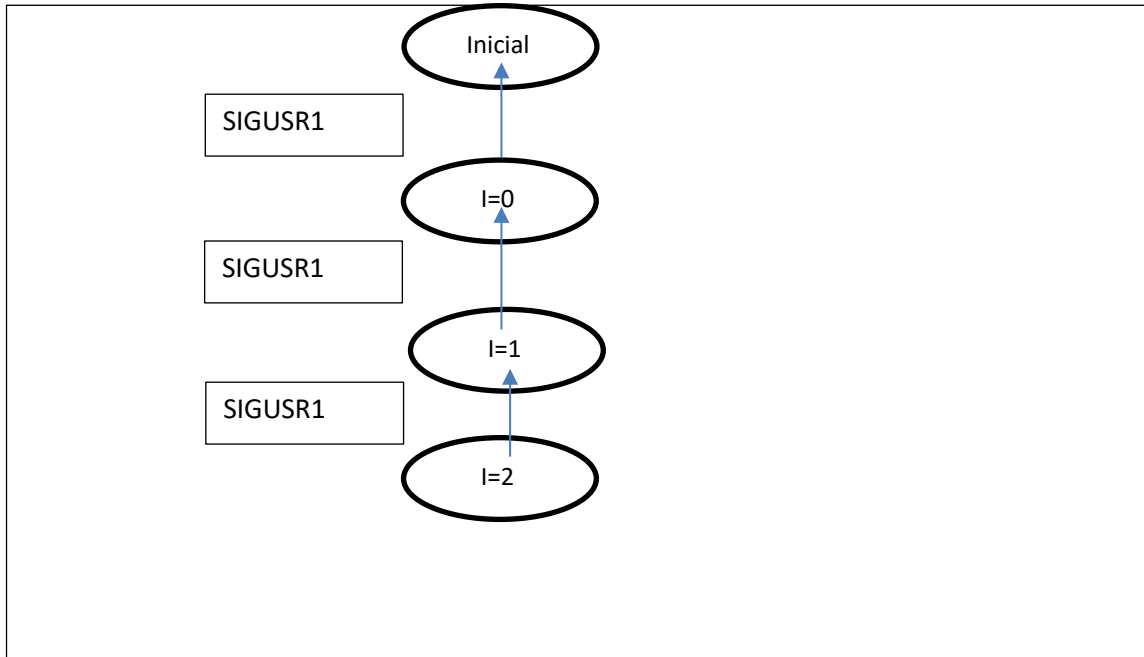
c) (0.25) ¿Qué pasaría si movemos la función `f_sigusr1` (antes del `main`) y el `sigaction` (antes de la creación de procesos) del programa `compute` al `n_computation`?

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		Tabla de Ficheros abiertos				Tabla de iNodo		
Entrada TFA		refs	modo	Posición l/e	Entrada T.inodo	refs	inodo	
0	1	0	4	RW	---	0	1	l-tty
1	0	1	1	R	---	1	2	l-pipe
2	0	2	1	W	---	2		
3		3				3		
4		4				4		
5		5				5		
		6						

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)**

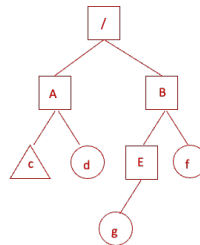
Suponed un sistema de ficheros descrito por los siguientes inodos y bloques de datos:

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	.	2	.	3	.	4	Texto	.	7	Texto	Texto
	..	2	..	2	..	2		..	4		
	A	3	c	5	E	7		g	8		
	B	4	d	6	f	8					

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.

- (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.



- 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|O_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.

- (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	<del>2</del> 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	. 2 .. 2 A 3 B 4	. 3 .. 2 c 5 d 6 h 9	. 4 .. 2 E 7 f 8	Texto	. 7 .. 4 g 8	Texto	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)

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

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

<pre> 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&lt;2048;i++) 8.         A[i] = B[i] = i; 9.     fork(); 10.    for (i=0;i&lt;2048;i++) 11.        ptr[i] = A[i] + B[i]; 12. }</pre>	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 ( $2048 * 4\text{Bytes} * 3$ ). 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.