Nombre alumno:
DNI:

Examen final de teoría de SO

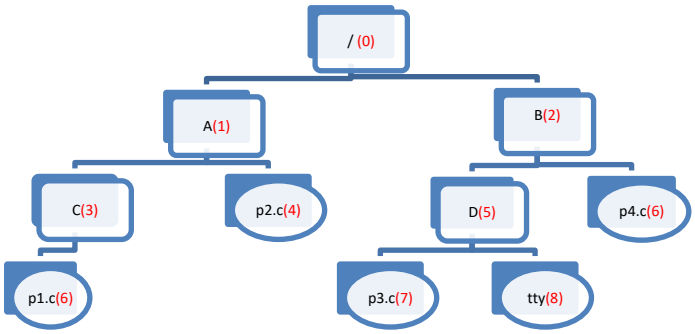
Justifica todas tus respuestas del examen. Las respuestas no justificadas se considerarán erróneas.

Sistema de Ficheros (*2 puntos*)

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

• Los rectángulos representan directorios, los óvalos representan otros elementos	• Las rutas destino de los softlinks se guardan en bloques de datos
• El tamaño de bloque es de 4 KB	• p1.c y p4.c son hardlinks al mismo fichero
• p3.c es un soft-link a p2.c	• p1.c ocupa 2 KB
• p2.c ocupa 12 KB	• tty es un dispositivo especial de carácter (una consola)

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**	(vacio)		

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 serian 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
Linea 1	I0C + B0C + I2C + B2C + I5C + B7C + I7C + B9C + I0C + B0C + I1C + B1C + I4C
Linea 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.
Linea 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)
Linea 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 "miprog" 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("./miprog","miprog", (char*)NULL);
----------------PUNTO B
else error("Error");
}
```

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 victima 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 preguntan 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 cambiarias 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
```

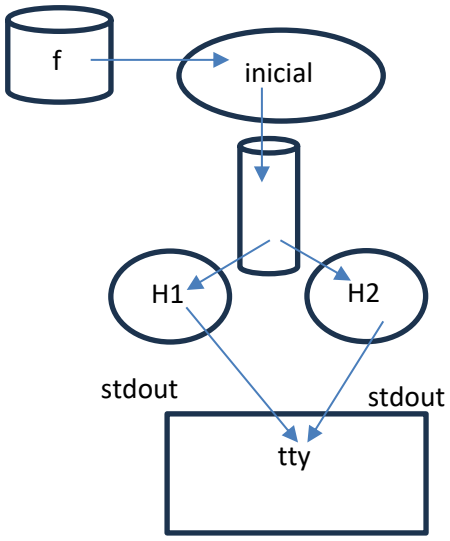
Nombre alumno:

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:



El proceso inicial crea un fichero f_out.pid que inicialmente no se accede. Crea dos procesos qu abren la pipeA para lectura y lo que van leyendo lo van enviando a la stdout.

El padre tienen en su stdin el fichero f, abre la pipe para escritura y todo el contenido del fichero se escribe en la pipe.

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
0	1
1	2
2	0
3	2
4	3
5	
6	
7	
8	
9	
10	
11	

Tabla de Ficheros abiertos

	refs	modo	Posición l/e	Entrada T.inodo
0	3	rw	-	0
1	3	r	1024	1
2	6	w	0	2
3	1	w	-	3
4	1	r	-	3
5	1	r	-	3
6				
7				
8				
9				
10				
11				

Tabla de iNodo

	refs	inodo
0	1	/dev/tty
1	1	F
2	1	F_out.pid
3	3	pipeA
4		
5		
6		
7		
8		
9		
10		
11		

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

	Tabla de Canales Entrada TFA
0	1
1	2
2	0
3	2
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.