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)* Indica cuál de las siguientes llamadas a sistema pueden provocar en Linux una transición del ready a blocked y en qué circunstancias.

open(nombre, O_RDONLY) (siendo "nombre" el nombre de un fichero):

Sí, cuando las operaciones para cargar el inodo de "nombre" requieran acceder al SF en disco (lento).

open(nombre, O_RDONLY) (siendo "nombre" el nombre de una pipe):

Sí, cuando el extremo de escritura de la pipe esté todavía cerrado

pipe(fd):

No, la llamada pipe() no es bloqueante

read(fd,buf,n) (siendo fd un canal asociado a un fichero de datos):

sí, si el fichero está almacenado en un dispositivo lento y se debe acceder a el para recuperar los dados

2. (*0,5 puntos*) En las mismas condiciones (cantidad memoria física, tamaño de página y tamaño del espacio lógico del procesador), si se aplica la optimización COW puede ser que un proceso provoque más excepciones durante su ejecución. ¿Verdadero o falso? Justifica tu respuesta.

Verdadero.

Se producirá una excepción que resuelve el kernel, cada vez que se accedan direcciones cuyas traducciones estén afectadas por el COW (no copiadas desde su padre hasta ese proceso).

Nombre alumno:
DNI:

3. (*0,5 puntos*) Un proceso de usuario, ¿puede provocar la ejecución de código en modo privilegiado?

Sí, como consecuencia de una llamada a sistema o de una excepción

4. (*0,5 puntos*) En una máquina con Linux existe el fichero de datos f1 dentro del directorio /A. El directorio actual es /A. Ejecutamos el siguiente comando:

%ln f1 f2

Para crear un hard link a f1 llamado f2. ¿Disminuirá la cantidad de bloques libres del sistema? ¿Y la cantidad de inodos libres?

Bloques del sistema – En principio no disminuirá. Se crea un hard-link, que únicamente representa la creación de un nombre y su asociación a un inodo existente. Solo se crearía un bloque si el nuevo nombre necesita un nuevo bloque en el directorio /A para almacenarlo.

Inodos libres – No disminuye, se mantiene. La creació de este hardlink solo asocia un nuevo nombre al inodo correspondiente.

Nombre alumno:
DNI:

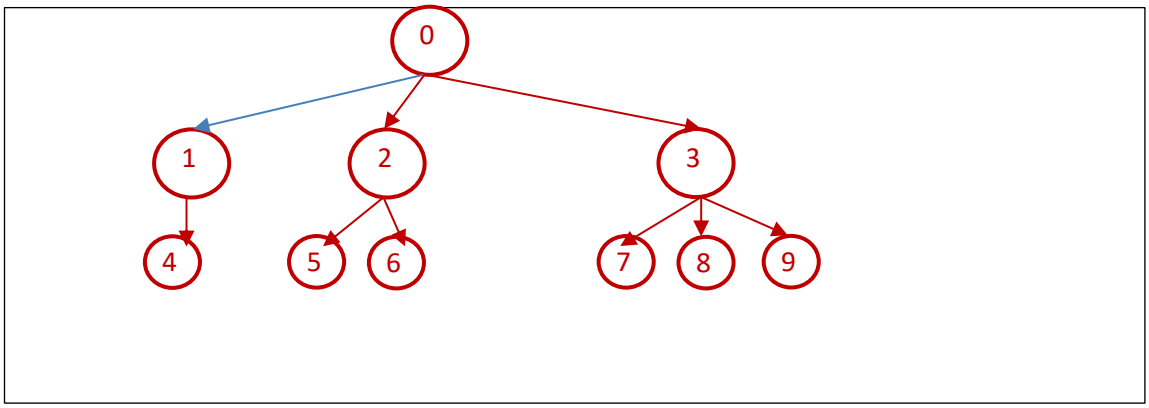
Gestión de procesos *(2 puntos)*

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, suponiendo que ejecutamos el siguiente comando: ./n_computation 3

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

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

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

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

for (i=levels-1;i>=0;i--)
      kill(pid[i], SIGUSR1);
      ret=waitpid(-1, &ec, 0
      trataExitCode( ret, ec);
}
exit(0);
}
```

```
/* compute */
void f_usr1(int s)
{
}

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

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

sigfillset(&m);
sigdelset(&m, SIGUSR1);
sigsuspend(&m);

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

1. *(0.75 puntos)* Dibuja la jerarquía de procesos que se genera. En el dibujo asigna un número a cada proceso para las preguntas posteriores.

2. 

Nombre alumno:
DNI:

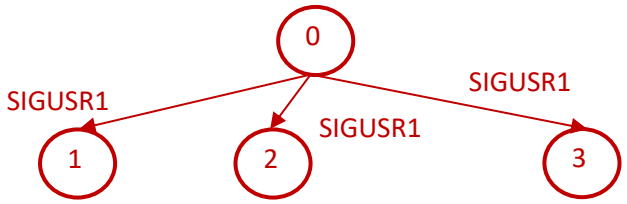
3. *(0,25 puntos)* ¿Qué podría pasar si eliminamos el sigprocmask de n_computation?

Podría pasar que se envíe el signal antes de que el los procesos que ejecutan el programa compute hayan hecho el sigaction. En ese caso se ejecutaría la acción por defecto que es acabar el proceso.

4. *(0,25 puntos)* Para la ejecución ¿Cuántos programas do_work podremos tener como mucho en ejecución al mismo tiempo?

Máximo 3 ya que hay una sincronización antes de la creación de los procesos para cada caso, Por lo tanto aunque los procesos que ejecutan "compute" están creados, estarán bloqueados en el sigsuspend y solo crearán los "do_work" a medida que n_computation les vaya enciando el SIGUSR1

5. *(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.



6. *(0,25 puntos)* El bucle que ejecuta la función trataExitCode, ¿Cuántos y qué mensajes escribirá?

Escribirá tres mensajes indicando el PID y el exit code cada proceso 1, 2 y 3. Los exit code serán respectivamente 1 , 2 y 3.

Nombre alumno:
DNI:

Pipes (*2 puntos*)

La **Error! Reference source not found.** contiene el código del programa *pipes*. Podéis suponer que "mipipe" es una pipe con nombre que existe y que tiene los permisos adecuados.

```
1.  void ras(int s) {
2.          write(2,"suspes\n",7);
3.          exit(1);
4.  }
5.  int main() {
6.  int fd,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. close(0);
14. pid=fork();
15. if (pid==0) {
16.         fd=open("mipipe",O_RDONLY);
17.         dup2(2,1);
18.         execlp("cat","cat",(char*)0);
19. }
20. else {
21.          fd=open("mipipe",O_WRONLY);
22.          write(0,"Examen ",7);
23.           close(0);
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). En este código, ¿podríamos substituir la pipe por un fichero de datos?

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.

No, depenent de la política de planificació podria passar que el fill intenti llegir del fitxer abans de que el pare hagi escrit y llavors el cat acabaría sense haver llegit res

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 justo antes de ejecutar línea 18 y el padre en la 24.

Tabla de Canales

	Entrada TFA
0	1
1	0
2	0
3	
4	
5	

Tabla de Ficheros abiertos

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

Tabla de iNodo

	refs	inodo
0	1	I_tty
1	1	I_pipe
2		
3		
4		
5		

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

Examen aprovat

Sí, el programa acaba.

Nombre alumno:
DNI:

Sistema de ficheros (2 puntos)

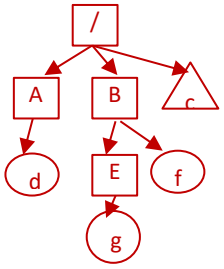
Suponed un sistema de ficheros descrito por los siguientes inodos y bloques de datos:

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	-	-	-	/A	-	-	-				
BDs	0	1	2	-	3	4	5 6				

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

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

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



2. Dado el siguiente código:

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

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

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

Nombre alumno:
DNI:

Accesos: (superbloque), inodo raíz, bloque raíz, inodo c (c es un soft link, obtenemos el path real a resolver), (superbloque), inodo raíz, bloque raíz, inodo A, bloque A, inodo d

- (superbloque) I2 b0 i5 (superbloque) i2 b0 i3 b1 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 resolver

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
`fdr=open("/c/d", O_RDONLY);`	SI	Si	Si
`write(fdw,buf,ret);`	No	Si	si
`unlink("/c/d");`	No	No	No

Justificación:

open: crea una nueva entrada en la tc y en la tfa. La entrada en la tc apunta a la entrada en la tfa. En la entrada de tfa el número de referencias es 1, el modo de acceso lectura y el puntero de l/e vale 0. Si el fichero no estaba en uso una nueva en la tabla de inodos con el inodo del fichero abierto y si ya estaba en uso incrementa el número de referencias de su entrada en la tabla de inodos. La entrada de la tfa apunta a esta entrada de la tabla de inodos

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

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

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

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

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

Nombre alumno:
DNI:

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

Memoria (2 *puntos*)

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

1. *(0,5 puntos*)

```
1.  int res[1024];
2.  int *ptr;
3.  main(){
4.     int A[1024], B[1024], i;
5.     ptr = res;
6.     for (i=0;i<1024;i++)
7.        A[i] = B[i] = i;
8.     for (i=0;i<1024;i++)
9.        ptr[i] = A[i] + B[i];
10. }
```

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data	2	4KB + 4B
Stack	3	8KB + 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 4KB (1024*4), es decir 1 página. 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 2 páginas en total (8KB). Por otro lado, el tamaño del Stack se calcula de forma parecida, ya que existen 2 arrays de 1 página c/u, así como también una variable entera. Por tanto, 12KB en total. Por último, no hay espacio de Heap utilizado (0 Bytes).

2. *(0,5 puntos*)

```
1.  int *ptr;
2.  main(){
3.     int A[1024], B[1024], i;
4.     ptr = sbrk(1024 * sizeof(int));
5.     for (i=0;i<1024;i++)
6.        A[i] = B[i] = i;
7.     for (i=0;i<1024;i++)
8.        ptr[i] = A[i] + B[i];
9.  }
```

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data	1	4B
Stack	3	8KB + 4B
Heap	1	4KB

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 4KB asignados (1024 * 4Bytes).

3. *(1 punto*)

```
1.  int *ptr;
2.  main(){
3.     int *A, *B, i;
4.     A = sbrk(1024 * sizeof(int));
5.     B = sbrk(1024 * sizeof(int));
6.     for (i=0;i<1024;i++)
7.        A[i] = B[i] = i;
8.     if (fork() == 0) {
9.        ptr = sbrk(1024 * sizeof(int));
10.       for (i=0;i<1024;i++)
11.          ptr[i] = A[i] + B[i];
12.    }
13. }
```

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

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 12KB asignados (1024 * 4Bytes * 3). En el proceso hijo, al existir la optimización COW, sólo se duplican aquellas páginas que requieren ser actualizadas por alguna escritura. En este caso, sólo se duplican: una página de la región de Stack (debido a la variable “i”), otra página en Data (debido a “ptr”) y 1 página 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.