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:

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

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

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

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

kill(getppid(), SIGUSR1);

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

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

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

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

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

while(sig_usr1 == 0);

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

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

	Entrada TFA
0	
1	
2	
3	
4	
5	

Tabla de Ficheros abiertos

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

Tabla de iNodo

	refs	inodo
0		
1		
2		
3		
4		
5		

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

Nombre alumno:
DNI:

Sistema de ficheros (2 puntos)

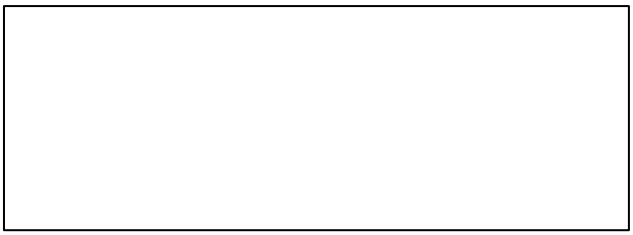
Suponed un sistema de ficheros descrito por los siguientes inodos y bloques de datos:

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

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

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



2. Dado el siguiente código:

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

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

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

Nombre alumno:
DNI:

Accesos:

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
`ret=read(fdr,buf,sizeof(buf))`			
`write(fdw,buf,ret);`			
`unlink("/A/c/g");`			

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

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*)

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

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data		
Stack		
Heap		

JUSTIFICACIÓN:

2. *(0,5 puntos*)

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

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data		
Stack		
Heap		

JUSTIFICACIÓN:

3. *(1 punto*)

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

Región	Páginas asignadas	Memoria Física (KB y/o B)
Data		
Stack		
Heap		

JUSTIFICACIÓN: