APELLIDOS:	**DNI:**
NOMBRE:	
FILA:	**COLUMNA:**

Sistemas Operativos – Facultat d´Informàtica de Barcelona - UPC

Fecha: 13 de Junio de 2005 Duración: 3 horas

Notas: Las notas se publicarán el lunes 4 de Julio en el RACÓ.

Revisión: La fecha, hora y lugar exactos de la revisión del examen se publicarán junto con las notas. Estad atentos.

ATENCIÓN: Las preguntas se tienen que contestar en las mismas hojas de examen utilizando el espacio reservado. Asegúrate de poner NOMBRE y APELLIDOS, DNI, fila y columna en cada una de las hojas. El examen se tiene que entregar en bolígrafo negro o azul. Se pueden utilizar las llamadas al sistema y un par de hojas en blanco.

Ejercicio 1 (3 puntos):

Indica si las siguientes sentencias son verdaderas o falsas explicando **brevemente** el porqué.

a) La librería de sistema contiene el código específico para invocar al Sistema Operativo.

b) Cuando la planificación de un sistema es no apropiativa un proceso no puede pasar de *blocked* a *running*.

c) Una diferencia entre pipes y named pipes es que las segundas utilizan bloques de datos para guardar los datos.

d) La E/S salida asíncrona permite a un proceso realizar múltiples operaciones sobre un dispositivo de manera simultánea.

e) Los procesos de usuario pueden provocar una entrada al sistema mediante interrupciones, excepciones y traps.

f) Una posibilidad para evitar *deadlocks* es tener una llamada a sistema que nos permita coger varios *mutex* a la vez.

g) Las tablas de páginas de los procesos siempre tienen que estar en memoria.

h) La técnica de *buffering* permite que los datos enviados o recibidos del dispositivo no se pierdan durante los picos de E/S.

i) Los directorios contienen los *hard links* a los ficheros.

j) Una de las ventajas de la asignación encadenada en tabla es que se pueden replicar los encadenamientos.

k) Para poder implementar los *mutex* hace falta que el hardware nos proporcione una instrucción de consulta atómica.

l) La técnica de *journaling* permite recuperar los datos eliminados del sistema de ficheros.

m) La función principal de los directorios es almacenar los atributos de los archivos.

n) El PCB de un proceso se libera cuando éste finaliza con la llamada de sistema *exit*.

o) Un signal no sirve para sincronizar procesos entre sí. Para eso debemos utilizar semáforos.

p) El algoritmo de planificación Round Robin con prioridades es justo.

q) Una pipe se puede utilizar únicamente para comunicar a un proceso padre con un proceso hijo.

r) El objetivo del *prefetching* es aprovechar la localidad temporal.

s) Una ventaja de la comunicación por memoria compartida es que no hay que invocar al Sistema Operativo.

t) Los ficheros son un dispositivo de tipo lógico.

u) Para utilizar un semáforo para sincronizar varios flujos hay que inicializarlo a ese número de flujos.

v) Lo primero que se hace al entrar al sistema es identificar el servicio que quiere realizar el usuario.

w) La llamada de sistema *alarm* bloquea al proceso durante el número especificado de segundos.

x) Todos los accesos a páginas inválidas son incorrectos y hay que enviar un signal al proceso que los realiza.

APELLIDOS:	**DNI:**
NOMBRE:	
FILA:	**COLUMNA:**

Ejercicio 2 (2.5 puntos):
Tenemos el siguiente sistema de ficheros basado en UNIX:

Inodo	2	3	4	5	6	7	8	9
tipo	dir	dir	dir	dir	dir	data	link	data
bloque	1	2	3	4	5	6	7	8
links	5	3	2	2	2	2	1	1

bloque	1		2		3		4		5		6	7	8
	.	2	.	3	.	4	.	5	.	6	1234	/B/h	abcd
	..	2	..	2	..	2	..	2	..	3			
	A	3	D	6	h	7	i	8	f	9			
	B	4							g	7			
	C	5											

Los Inodos y bloques que no están listados, están libres y se van asignando en orden secuencial.

Dibuja los Inodos y/o bloques de datos que se modifican/crean al ejecutar las siguientes líneas de comandos. No son acumulativas.

a) > mkdir /A/D/E

b) > mv /A/D/f /C/

c) > rm /B/h

d) > cp /C/i   /A/j

e) > ln –s /A/D   /s
sintaxis: *ln –s destino nombre*. Crea un simbolic link con nombre *nombre* que apunta a *destino*.

APELLIDOS:	**DNI:**
NOMBRE:	
FILA:	**COLUMNA:**

Ejercicio 3 (1.5 puntos):

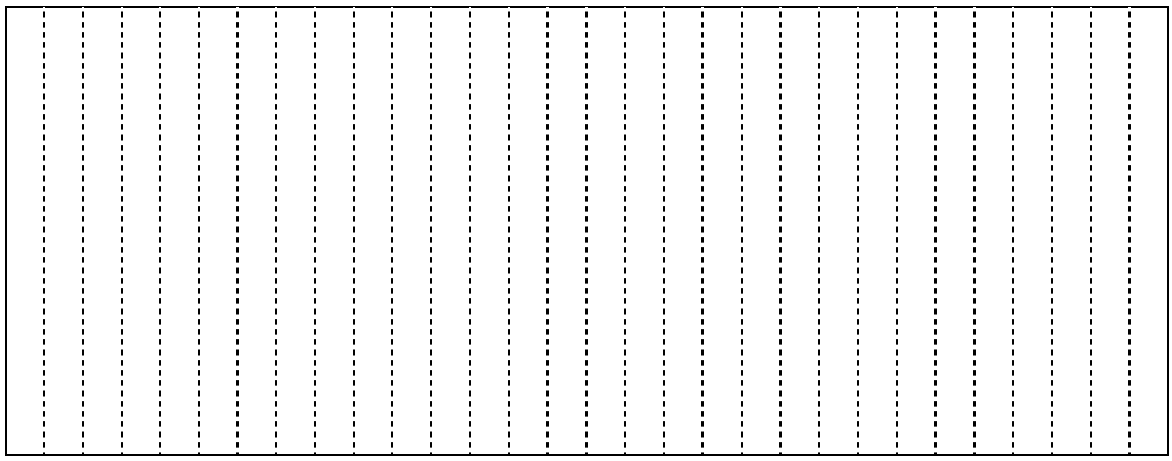
Tenemos un sistema con un algoritmo de planificación Round Robin con prioridades (quantum=3) y apropiación inmediata. En este sistema existen, en el instante 0, los siguientes procesos, por orden de llegada:

Proceso	Prioridad	Ejecución (en ciclos)
P1	1	2 CPU – 2 E/S – 2 CPU
P2	2	5 CPU – 5 E/S – 4 CPU
P3	2	2 CPU – 3 E/S – 3 CPU
P4	3	2 CPU – 4 E/S – 2 CPU

Mayor número de prioridad significa mayor prioridad en el sistema.

Se puede realizar un número infinito de E/S en paralelo.

Dibuja, en la siguiente rejilla, el diagrama de Gantt de la ejecución de estos cuatro procesos en el sistema:



Ejercicio 4 (2 puntos):

Volem fer un codi que faci el següent:

```
_$ myprog una_pipe un_fitxer
```

El programa crearà dos fills (anomenen-los A i B)

1. A llegirà el que li arribi per la named pipe *una_pipe* (assumim que és text), i l'enviarà cap a B, transformant les minúscules en majúscules i deixant la resta igual. A més, cada cop que rebi una minúscula, enviarà un SIGUSR1 al pare. Un cop acabada la transmissió, morirà.
2. B escriurà el que rebi d'A al fitxer *un_fitxer*. Si en un segon no ha acabat la transmissió, enviarà un SIGUSR2 al pare. Si la transmissió acaba abans, morirà normalment.
3. Si el pare rep un SIGUSR2, indicarà que ho ha rebut i morirà, havent matat els fills prèviament. Altrament, el pare morirà quan detecti que han mort els dos fills, indicant el número de minúscules trobat.

Per fer més curt el problema, no afegim el codi de tractament d'errors. Suposarem que la pipe una_pipe està creada.

```
 1: #include ... (suposem que està bé)
 2: void tractar_SIGUSR1(int i){
 3:     numero_siguser1_rebuts++; }
 4: void tractar_SIGUSR2(int i) {
 5:     printf("Rebut un SIGUSR2. Acabo\n");
 6:     kill(pid1,SIGTERM);
 7:     kill(pid2,SIGTERM);
 8:     exit(0); }
 9: void tractar_temps(int i) {
10:     kill(getppid(),SIGUSR2); }
11: void main(int argc; char *argv[]) {
12:     int pid1, pid2, numero_siguser1_rebuts;
13:     int fd, np, p[2];
14:     char c;
15:     pipe(p);
16:     signal(SIGUSR1, tractar_SIGUSR1);
17:     signal(SIGUSR2, tractar_SIGUSR2);
18:     pid1=fork();
19:     if (pid1==0) {
20:       np=open(argv[1], O_RDWR);
21:       while (read(np, c, sizeof(char))>=0) {
22:            if ((c>='a')&&(c<='z')) {
23:              c+='a'-'A'; /*nota: aquesta linia està bé*/
24:              kill(getpid(),SIGUSR1); }
25:            write(p[0],&c,sizeof(char)); }
26:     close(p[0]);
27:     close(p[1]);
28:     close(np);
29:     exit(0); }
30:     pid2=fork();
31:     if (pid2==0) {
32:     signal(SIGALRM, tractar_temps);
33:     alarm(1);
34:     fd=creat(argv[2], 0666);
35:     while(read(p[0],&c, sizeof(char))>0)
36:            write(fd,&c,sizeof(char));
37:     close(p[0]);
38:     close(p[1]); }
39:     waitpid(pid1,NULL,0);
40:     waitpid(pid2,NULL,0);
41:     printf("Acabo havent rebut %d minúscules\n", numero_siguser1_rebuts);
42:     exit(0);
43:}
```

APELLIDOS:	**DNI:**
NOMBRE:	
FILA:	**COLUMNA:**

Aquest codi conté 10 errors. Enumera´ls tot indicant per cada un la línia on es troba, quin es l´error i com el solucionaries.

Error 1:

Error 2:

Error 3:

Error 4:

Error 5:

Error 6:

Error 7:

Error 8:

Error 9:

Error 10:

Ejercicio 5 (1 punto):

Di, de las siguientes secuencias de código, en cuáles y en qué situación se producen *deadlocks*, suponiendo que las ejecutan varios threads a la vez.

a) Inicialización: `sem_init(sem1, 1); sem_init(sem2,0);`

```
sem_wait (sem1);
sem_wait (sem2);
sem_signal (sem2);
sem_signal(sem1);
```

b) Inicialización: `sem_init(sem1, 1); sem_init(sem2, 0);`

```
sem_wait(sem1);
sem_signal(sem2);
sem_signal(sem1);
sem_wait(sem2);
```

c) En este caso, tenemos 2 threads, cada uno ejecuta un código diferente:
Inicialización: `sem_init(sem1, 2); sem_init(sem2, 1); sem_init(sem3, 1);`

```
      T1                                    T2

sem_wait(sem1);                       sem_wait(sem1);
sem_wait(sem2);                       sem_wait(sem3);
sem_wait(sem3);                       sem_wait(sem2);
sem_signal(sem2);                     sem_signal(sem3);
sem_signal(sem3);                     sem_signal(sem1);
sem_signal(sem1);                     sem_signal(sem2);
```

d) Seguimos teniendo 2 threads
Inicialización: `sem_init(sem1, 1); sem_init(sem2, 1);`

```
      T1                                    T2

sem_wait(sem1);                       sem_wait(sem2);
if (sem2->count>0)                    sem_wait(sem1);
      sem_wait(sem2);                 sem_signal(sem2);
sem_signal(sem2);                     sem_signal(sem1);
sem_wait(sem1);
```