COGNOMS: **FILA:**

NOM: **COLUMNA:**

Examen final SO (pla 2003). 21 de gener de 2005.

Les notes i les dates de revisió sortiran el dijous 27 de gener a migdia.

Problema 1. Contesta breument (2 punts)

1.- Explica en que consisteix l'algorisme de planificació de disc C-SCAN

2.- Quina és la diferència entre un dispositiu lògic i un dispositiu virtual ?

3.- El hardware proporciona com a mínim dos modes d'execució. Per què ?

4.- Explica com funciona la tècnica de buffering i quins avantatges / desavantatges té.

5.- Què es modifica quan executem execlp en un procés?

6.- Enumera els passos per crear un nou procés si el SO té l'optimització CopyOnWrite.

7.- A què ens referim quan diem que un programa és resident en memòria?

8.- Què comparteixen i què no comparteixen els threads d'un procés?

9.- Perquè és necessari tenir una política de reemplaçament de pàgines de memòria?

Problema 2. (2 punts)

Donat el següent sistema de fitxers tipus UNIX:

Inodes

tipus	dir	data	link	dir	dir
mida	4096	200	8	6000	4096
BDs	2	4	6	5 3	7
	2	3	4	5	6

Blocs dades

.	2	c	4	/tmp/a		.	5	/tmp/c		.	6
..	2	d	10	.		..	2			..	2
etc	6					a	9			files	8
tmp	5					b	4			data	4
2		3		4		5		6		7	

Els blocs són de 4Kb. El superbloc es troba sempre a memòria i no hi ha cap tipus de buffercache. Si tenim el següent programa

```
int fd;

void thread1()
{
      char buf[100];
      int n,m;

      n = read(fd,buf,sizeof(buf));       2
      m = n;
      n = lseek(fd,SEEK_END,0);           3
      n = write(fd,buf,m);                5
}

void thread2()
{
         char buf[100];
         int n;

         n = read(fd,buf,sizeof(buf));    4
         n = read(fd,buf,sizeof(buf));    6
}

int main ()
{
      fd = open("/etc/data",O_RDWR);      1
      create_thread(thread1);
      create_thread(thread2);
}
```

(segueix a un altre full)

COGNOMS: **FILA:**

NOM: **COLUMNA:**

Problema 2 (continuació).

Apartat 1. Calculeu quants accessos, indicant quins són, es fan al disc en cadascun dels punts indicats (suposant que es fan en l'ordre especificat). Quant val n en cada punt?

1. Accessos:

   Total d'accessos per aquesta instrucció =

2. Accessos:

   Total d'accessos per aquesta instrucció = n =

3. Accessos:

   Total d'accessos per aquesta instrucció = n =

4. Accessos:

   Total d'accessos per aquesta instrucció = n =

5. Accessos:

   Total d'accessos per aquesta instrucció = n =

6. Accessos:

   Total d'accessos per aquesta instrucció = n =

Apartat 2. A quines estructures de dades s'accedeix al punt 4? Quines es modifiquen?

Problema 3. Per pensar una mica (1.5 punts)

1.- Quan canvia el PID d'un procés i perquè?

2.- Si en un sistema de fitxers tipus UNIX tenim blocs de dades lliures però ens dona error al intentar crear un nou fitxer, A què pot ser degut ?

3.- Si per accident esborro un fitxer, explica com ens ajuda la tècnica RAID?

4.- Tinc localitzada l'entrada de directori corresponent a un fitxer. Quants accessos a disc calen per accedir al bloc 200 en els següents casos:

a) Assignació encadenada

b) Assignació encadenada en taula (FAT)

c) Assignació indexada amb 50 indexos per bloc d'índex

d) Assignació indexada multinivell amb 10 índexs directes, 1 doble indirecte i un triple indirecte. Els blocs d'índexs tenen 50 índexs cadascun.

5.- Quan es millor fer servir una estructura tipus hash enlloc d'una llista simple com a estructura interna dels directoris?

6.- Què escriu per la sortida estàndard aquest fragment de codi?

```
int fd;
char *p=”prueba\n”, c, b[5];

mknod (“pi”, S_IFIFO|0666, 0);
fd=open(“pi”, O_RDWR);
write (fd, p, 6);
read (fd, &c, 1);
sprintf(b, “%c\n”, c); write (1, b, 2);
write (fd, p, 6);
read (fd, &c, 1);
sprintf(b, “%c\n”, c); write (1, b, 2);
```

7.- Què passaria en un SO si cada flux (thread) no tingués la seva pròpia variable errno?

COGNOMS: **FILA:**

NOM: **COLUMNA:**

Problema 4 (2.5 puntos)

Tenemos un proceso que ejecuta el siguiente código:

```
1:int pd[2];
2:char buffer[256];
3:switch (fork()) /* El padre continúa en Run y el hijo está en Ready */
4:{
5: case 0:
6:       mknod("mipipe", S_IFIFO|0666,0);
7:       pd[1]=open("mipipe", O_WRONLY);
8:       sprintf(buffer, "pipe\n");
9:       write(pd[1], buffer, strlen(buffer));
10:      close (pd[1]);
11:      unlink ("mipipe");
12:      break;
13:default:
14:      mknod("mipipe", S_IFIFO|0666,0);
15:      pd[0]=open("mipipe", O_RDONLY);
16:      while(read(pd[0], buffer, 1)>0);
17:      close (pd[0]);
18:      unlink ("mipipe");
19:}
20:exit(0);
```

Suponemos que en nuestro sistema existen más procesos que no interfieren con este. También suponemos que todas las llamadas al sistema funcionan perfectamente. La política de planificación es FIFO sin prioridades ni apropiación de la CPU. Trabajaremos con el grafo de estados básico (READY, RUN, BLOCKED). La named pipe "mipipe" no existe antes de la ejecución de este proceso.

Apartado 1. Escribe las llamadas al sistema, en orden de ejecución, junto con su resultado, al ejecutar el código del proceso. Para simplificar, escribe el número de línea junto con el resultado de la llamada al sistema.

Apartado 2. Supongamos que el código que ejecuta ahora el proceso es el siguiente:

```
1:int pd[2];
2:char buffer[256];
3:pipe(pd);
4:switch(fork()) /* El padre sigue en Run y el hijo se crea en Ready */
5:{
6:  case 0:
7:        sprintf(buffer, "pipe\n");
8:        write(pd[1],buffer, strlen(buffer));
9:        close(pd[1]);
10:       break;
11: default:
12:       while (read(pd[0], buffer, 1)>0);
13:       close(pd[0]);
14:}
15:exit(0);
```

Escribe las llamadas al sistema, en orden de ejecución, junto con su resultado, que se ejecutan ahora. Como en el caso anterior, escribe el número de línea junto con el resultado de la llamada al sistema.

Apartado 3.¿Cuantas llamadas al sistema acceden a disco en cada uno de los dos casos?

Apartado 4. Explica, en palabras, como modificarías el código del apartado 1 para que realice menos llamadas al sistema.

COGNOMS: **FILA:**

NOM: **COLUMNA:**

Problema 5 (2 punts)

Sigui un programa amb 4 threads:

thread principal	thread 1	thread 2	thread 3
...	...	...	...
	BI1	BI2	BI3
// crear els	...	...	...
// 3 threads	BI4	BI5	
	...	...	

Volem que els blocs d'instruccions BI1, ..., BI5 s'executin amb aquestes restriccions:
1.- Cap instrucció de BI2 hauria d'executar-se abans de que s'executessin totes les de BI1.
2.- Cap instrucció de BI2 ni de BI3 hauria d'executar-se simultàniament.
3.- Les instruccions de BI5 haurien d'executar-se quan tant les de BI4 com les de BI2 s'hagin executat al complert.

Es demanen 2 solucions:

La primera, amb semàfors (i només semàfors, no podeu fer servir cap variable auxiliar). La solució hauria de ser tal que cap thread esperi més temps del necessari. Intenteu-lo fer amb el mínim nombre de semàfors possible. Contesteu a continuació:

thread principal	thread 1	thread 2	thread 3
...	...	...	...
	BI1	BI2	BI3
// crear els			
// 3 threads			
	...	...	...
	BI4	BI5	

La segona solució seria fer-lo amb instruccions de mutex com les descrites a classe. En aquest cas podeu fer servir variables addicionals. Contesteu a continuació:

thread principal	thread 1	thread 2	thread 3
...	...	...	...
	BI1	BI2	BI3
// crear els			
// 3 threads			
	...	...	...
	BI4	BI5	