	<b>Sistemas Operativos. 10 de febrero de 2006</b>		
	Nombre _____ Apellidos _____ DNI _____ Titulación _____ Grupo _____		

## TEORÍA

### 1. (Ficheros) 2 puntos

- a) Describa qué son y cómo se implementan los enlaces simbólicos y físicos en un sistema de ficheros como el usado por Linux/UNIX (basada en el uso de nodos-i y tablas de punteros a bloques).
- b) ¿Qué problemas encuentra a la hora de implementar cada uno de dichos tipos de enlace en el sistema de ficheros FAT?

### 2. (Entrada/Salida) 2 puntos

- a) ¿Por qué es necesaria la interrupción de reloj? Indique al menos tres de las funcionalidades que aporta e indica una llamada al sistema relacionada con cada funcionalidad.
- b) ¿De qué factores físicos y organizativos depende el tiempo de búsqueda (seek) para acceder a un bloque en un disco?
- ¿De qué factores físicos y organizativos depende el tiempo de latencia para acceder a un bloque en un disco?
- ¿De qué factores físicos y organizativos depende el tiempo de transferencia de un bloque en un disco?

### 3. (Procesos) 2 puntos

- a) Describa brevemente la política de planificación de procesos Round Robin. En la versión presentada en clase, ¿cómo se implementa la cola de procesos listos para ejecutar?
- b) Considere una variante de la implementación de la cola de procesos listos en la planificación Round-Robin donde la entradas de la cola son punteros a los BCPs. ¿Qué efecto tendría colocar dos punteros al mismo proceso en dicha cola? ¿Qué ventajas proporcionaría este esquema? ¿Y qué problemas presentaría?

### 4 (Sincronización) 2 puntos

Uno de los mecanismos de sincronización de UNIX son las señales, que se pueden usar para forzar precedencias entre procesos. También el mecanismo de semáforos se puede emplear en forma de señalización para forzar precedencias entre procesos.

- a) Comenta las diferencias entre ambos mecanismos,
- b) Establece la correspondencia entre las llamadas a sistema de gestión de señales y de operación de semáforos que cumplen cometidos análogos.

Por ejemplo: Queremos que el código A del proceso P1 vaya antes que el código B del proceso P2. ¿Cómo se especifica la precedencia con señales, y cómo con semáforos?

P1: código inicial 1 + código A + código final 1

P2: código inicial 2 + código B + código final 2

### 5 (Memoria) 2 puntos

- a) ¿En qué consiste el fenómeno de hiperpaginación? ¿Qué “síntomas” en el comportamiento del sistema sugieren que se está produciendo hiperpaginación?

b) Comenta al menos tres razones por la cuales resulta provechoso mantener un conjunto (“pool”) siempre abastecido de marcos de páginas libre. ¿Cómo se llama a esta estrategia de gestión de memoria virtual?

## Soluciones:

1.

a) Enlace físico: cada uno de los nombres con los que se designa un fichero, tal que permite establecer una relación directa entre el nombre y los atributos del fichero. En sistemas de ficheros UNIX se implementa incluyendo en la entrada de directorio la correspondencia entre el nombre del enlace y el número de inodo representativo del fichero; entre los atributos del fichero se incluye un contador de enlaces

0,5

Enlace simbólico: cada uno de los nombres que se asocian como "alias" al nombre directo o "enlace físico" de un fichero. Se implementa creando un nuevo tipo de fichero cuyo contenido es el nombre de ruta del fichero al que se asocia.

0,5

b) Problema de enlace físico en FAT: el nombre del fichero es uno de los campos del conjunto de atributos de un fichero, contenidos todos en la entrada de directorio. Disponer de más de un enlace físico supondría disponer de más de una entrada de directorio con los mismos atributos exactamente y mantener la coherencia de tales entradas sería costosísimo.

0,5

Problema de enlace simbólico en FAT: pasaría por crear un nuevo tipo de fichero cuyo contenido fuera tratado como una referencia indirecta al nombre de ruta de otro fichero. Es factible y de hecho los ficheros de enlace (\*.lnk) de Windows tienen un comportamiento semejante.

0,5

2.

a) Porque es el modo esencial por el cual el Sistema Operativo puede retomar el control del sistema con independencia del comportamiento correcto o incorrecto, cooperador o "egoísta" de las aplicaciones bajo su cargo.

0,5

Función	Llamada
1. Mantenimiento de fecha y hora.	1. time(), stime()
2. Soporte para planificación de procesos.	
3. Gestión de temporizadores.	3. alarm()
4. Contabilidad y estadísticas.	4. times()

0,4 +  
0,1

b)

	Factor físico	Factor organizativo
t. seek	Velocidad lineal del brazo Densidad de pistas	Política de planificación Continuidad de asignación
t. latencia	Velocidad de rotación Densidad de sectores en pista	Intercalado Continuidad de asignación
T. transferencia	Velocidad de rotación Densidad de bits en pista	Tamaño de unidad de transferencia (Bloque, Cluster)

0,3 +  
0,3 +  
0,3

3.

a) Round Robin: reparto equitativo del tiempo de procesador mediante la concesión por turnos de la CPU a cada proceso durante intervalos de tiempo tasados llamados "quanto"

Implementación: mediante una cola circular de procesos (lista circular enlazada de BCPs); un puntero señala al proceso que se encuentra a la cabeza de la cola y que es el elegido para usar la CPU durante un "quanto"; acabado ese tiempo, el puntero se mueve para apuntar al siguiente proceso en la cola.

0,5 +  
0,5

b) En una vuelta a la cola de procesos ese proceso repetido disfrutaría de dos "quantos" de CPU mientras que los demás solo usarían uno.

Ventaja: permitiría regular de forma selectiva los tiempos de CPU concedidos a cada proceso individualmente sin alterar el comportamiento básico del esquema de planificación

Problema: en el caso de que un proceso que está anotado más de una vez en la cola Round-Robin se bloquee y devolviese voluntariamente la CPU, sería obligado que el resto de sus apariciones en la cola fuesen anuladas, complicando el comportamiento básico del esquema de planificación.

0,4 +  
0,3 +  
0,3

4.

a) Señales UNIX: hay un número fijo de señales disponibles, su comportamiento de tipo interrupción; se asocian a una rutina de servicio; están asociadas (por diseño) a eventos específicos, se envían mediante llamada o mediante ocurrencia del evento asociado, interrumpen asíncronamente o pueden ser esperadas mediante llamadas; son enmascarables; no son acumulables.

Semáforos de señalización: su número es variable dinámicamente (se crean y se destruyen), no están asociadas a rutinas de servicio; los eventos que señalizan son establecidos por el programador; son síncronas: envían señales mediante llamadas y reciben señales mediante llamadas; no son enmascarables; son acumulables.

b)

	Señales UNIX	Semáforos de señalización
Armado	signal(nº señal, acción)	
Iniciación		sem_init(), sem_open()
Envío de señal	Aparición del evento, o kill (pid, nº señal)	sem_post(semáforo)
Espera de señal	En cualquier momento o pause()	sem_wait(semáforo)
Finalización		sem_close(), sem_destroy()

P1: código\_inicial\_1 + **código A + sem\_post(sem)** + código\_final\_1

P2: código\_inicial\_2 + **sem\_wait(sem)** + **código B** + código\_final\_2

P1: código\_inicial\_1 + **código A + kill(pid2,SIGUSR1)** + código\_final\_1

P2: código\_inicial\_2 + **pause()** + **código B** + código\_final\_2

5.

a. Hiperpaginación: el espacio residente asignado a un proceso es insuficiente para contener el espacio de trabajo (las necesidades actuales de memoria) del mismo y en consecuencia se genera un número elevado de fallos de página

Síntomas: el proceso pasa más tiempo en la cola de E/S para servicio swap que ejecutándose; el porcentaje de uso de CPU es muy pequeño mientras que el porcentaje de uso del disco de swap es elevado.

b. Razón 1: El tratamiento de un fallo de página supone, en el caso peor, una sola transferencia de E/S (la lectura de la página a memoria)


Razón 2: La aplicación del algoritmo de reemplazo se efectúa seleccionando un gran número de páginas de una vez rentabilizando su invocación (en vez de seleccionarlas de una en una conforme vayan siendo requeridas)

Razón 3: Permite recuperar una página si la decisión de reemplazarla se revela prematura, permitiendo distinguir entre fallos "menores" (servidos desde el "pool" de memoria) y fallos "mayores" (servidos desde disco)

Razón 4: como consecuencia de la razón 3 el comportamiento del algoritmo de reemplazo utilizado ya no es tan determinante

Razón 5: las actualizaciones a disco de las páginas liberadas se van haciendo poco a poco, en tandas, rentabilizando los costosos accesos a disco.

Se llama "Buffering de páginas"

	<b>Sistemas Operativos. 10 de febrero de 2006</b>	
	Nombre_____Apellidos_____	
	DNI_____	Titulación_____Grupo_____

### PROBLEMAS

#### 1. (Ficheros) 3 puntos (a: 0,5; b: 2,5)

Sea un sistema de ficheros similar al de UNIX con un tamaño de bloque de 1KB y un nodo-i con 4 punteros directos y 1 indirecto simple. La política de escritura es *write-through* por lo que todas las operaciones de escritura deben reflejarse de inmediato en disco. Considere el siguiente fragmento de código:

```

.....
buf[512]="ccccccc....."; /* 512B de c's */
mkdir("data");
1:  fd1=creat("data/fichero",0666); /* crea y abre fichero */
2:  write(fd1,buf,512);
3:  pid = fork();
   if (pid==0) {
       char buf2[1024]="bbbbbbbb...."; /* 1KB de b's */
       int fd2;
4:       fd2=open("data/fichero",O_RDWR);
5:       write(fd2,buf2,1024);
6:       lseek(fd2,5200,SEEK_SET); //SEEK_SET: desde inicio
7:       write(fd2,buf2,200);
8:       write(fd1,buf,512);
9:       close(fd2);
       exit(0);
   }
   else {
       wait();
10:     read(fd1,buf_res,1024);
11:     close(fd1);
   }

```

a) ¿Qué contendrá el buffer *buf\_res* tras la llamada a *read*?

b) Indique qué zonas del disco (mapa de bloques libre (**MB**), mapa de nodos-i (**MI**) libres, lista de nodos-i (**LI**) y bloques de datos (**BD**)) y qué tablas del gestor de ficheros del sistema operativo (tabla de descriptores de ficheros abiertos (**tda**), tabla intermedia de posiciones (**TFA**) y tabla de nodos-i (**TIN**)) se modifican en cada una de las llamadas indicadas. Complete la tabla, marcando las casillas de los elementos que se modifican. Comente con detalle las llamadas marcadas en la tabla con un asterisco (\*)

	MB	MI	LI	BD	tda	TFA	TIN
1 *							
2							
3 *							
4 *							
5							
6							
7 *							
8							
9							
10							
11							

*							
---	--	--	--	--	--	--	--

## 2. (Sincronización) 4 puntos (a: 0,5; b: 0,5; c: 2; d: 1)

Tras finalizar el diseño de nuestro sistema operativo, nos damos cuenta de que no proporcionamos soporte para semáforos generales. Por fortuna, sí disponemos de mutexes y variables condicionales. Complete cada uno de los siguientes apartados para crear una biblioteca de rutinas que implemente semáforos generales a partir de mutexes y variables condicionales.

- a) Describa en primer lugar la estructura de datos que representará al semáforo general, completando/modificando la propuesta a continuación:

```
typedef struct sem_t {
    mutex_t *cerrojo;    // mutex
    cond_t *cond;       // variable condicional
    .....
} semaforo;
```

- b) Implemente la función de creación e inicialización del semáforo. ¿Qué parámetro(s) deberá recibir dicha función?

```
semaforo *crear_semaforo (.....) {
    semaforo *sem = malloc(sizeof(semaforo));
    pthread_mutex_init(sem->cerrojo, NULL);
    pthread_cond_init(sem->cond, NULL);
    .....
    return sem;
}
```

- c) Finalmente, implemente las funciones “**wait**” y “**signal (post)**”. Ambas recibirán como parámetro un semáforo (tipo **semaforo** creado en el apartado a) y devuelto por la función **crear\_semaforo**).

- d) Rehaga el ejercicio pero usando tuberías (**pipes**) en lugar de mutexes y variables condicionales.

## 3. (Procesos) 3 puntos

Tres hilos de usuario H1, H2 y H3 de un mismo proceso se ejecutan bajo planificación FIFO con expropiación y prioridades fijas. Los códigos esquematizados de los hilos son los siguientes:

H1 (prioridad <i>baja</i> )	H2 (prioridad <i>media</i> )	H3 (prioridad <i>alta</i> )
funcionA(); // tarda 25 msg lock(mutex1); funcionB(); // tarda 60 msg unlock(mutex1); funcionC(); // tarda 20 msg	funcionM(); // tarda 30 msg read(fd, buffer, 1000); // tarda 80 msg funcionN(); // tarda 5 msg	funcionX(); // tarda 25 msg lock(mutex1); funcionY(); // tarda 5 msg unlock(mutex1); funcionZ(); // tarda 40 msg

H1 comienza a ejecutarse en el instante 0, H2 empieza a los 30 msg, y H3 empieza a los 40 msg.

- Determinar los tiempos de retorno y espera (absolutos, normalizados y promedios) de los tres hilos.
- Repetir el problema a) si la prioridad de un hilo, mientras es propietario de un *mutex*, pasa temporalmente a ser *alta*
- Repetir el problema a) si los hilos son de tipo kernel
- Repetir el problema b) si los hilos son de tipo kernel

0,5

1.

a) Ceros. Lee el segundo bloque del fichero “data/fichero” que no ha sido escrito en ningún momento. Se ha escrito en el bloque primero y en el sexto.

1,0

b)

	MB	MI	LI	BD	tda	TFA	TIN
1 *		X	X	X	X	X	X
2	X		X	X		X	X
3 *					X	X	
4 *					X	X	
5			(X)	X		X	X
6						X	
7 *	X		X	X		X	X
8			(X)	X		X	(X)
9					X	X	
10			(X)			X	(X)
11 *					X	X	X

1,5

Llamada 1:

BD: escribe en el primer bloque del directorio “data” (existente) la entrada “fichero”

MI, LI: reserva un inodo nuevo para “fichero”

Tda, TFA, TIN: abre “fichero”, por primera vez, con un nuevo descriptor

Llamada 3:

Tda: duplica la tda del proceso padre

{TFA: y actualiza los contadores de referencias de las aperturas afectadas}

Llamada 4:

Tda: nuevo descriptor

TFA: para una nueva apertura de “fichero” ya abierto

Llamada 7:

MB, BD: usa dos bloques de datos nuevos; uno para punteros indirectos simples, otro para datos del bloque sexto de “fichero”, posiciones 5200-5400

LI, TIN: actualiza información de localización y tamaño en el inodo

TFA: actualiza la posición actual de fd2 en “fichero”

Llamada 11:

Tda: suprime la entrada correspondiente a fd1

TFA: suprime la apertura correspondiente a fd1

TIN: suprime el inodo en memoria de “fichero” que ya no es referenciado por nadie

2.

a)

0,5

```
typedef struct sem_t {
    mutex_t  *cerrojo;    // mutex
    cond_t   *cond;      // variable condicional
    int      valor ;     // valor entero del semáforo general
} semaforo;
```

b)

0,5

```
semaforo *crear_semaforo(int valor_inicial) {
    if (valor_inicial < 0) return NULL;
    semaforo *sem = malloc(sizeof(semaforo));
    pthread_mutex_init(sem->cerrojo, NULL);
    pthread_cond_init(sem->cond, NULL);
    sem->valor = valor_inicial;
    return sem;
}
```

c)

1,0

```
void wait(semaforo sem) {
    pthread_mutex_lock(sem->cerrojo);
    sem->valor--;
    while (sem->valor < 0)
        pthread_cond_wait(sem->cond, sem->cerrojo);
    pthread_mutex_unlock(sem->cerrojo);
}
```

1,0

```
void post(semaforo sem) {
    pthread_mutex_lock(sem->cerrojo);
    sem->valor++;
    if (sem->valor <= 0)
        pthread_cond_signal(sem->cond, sem->cerrojo);
    pthread_mutex_unlock(sem->cerrojo);
}
```

d)

1,0

```
typedef struct sem_t {
    int      fd[2];      // descriptors del pipe
} semaforo;

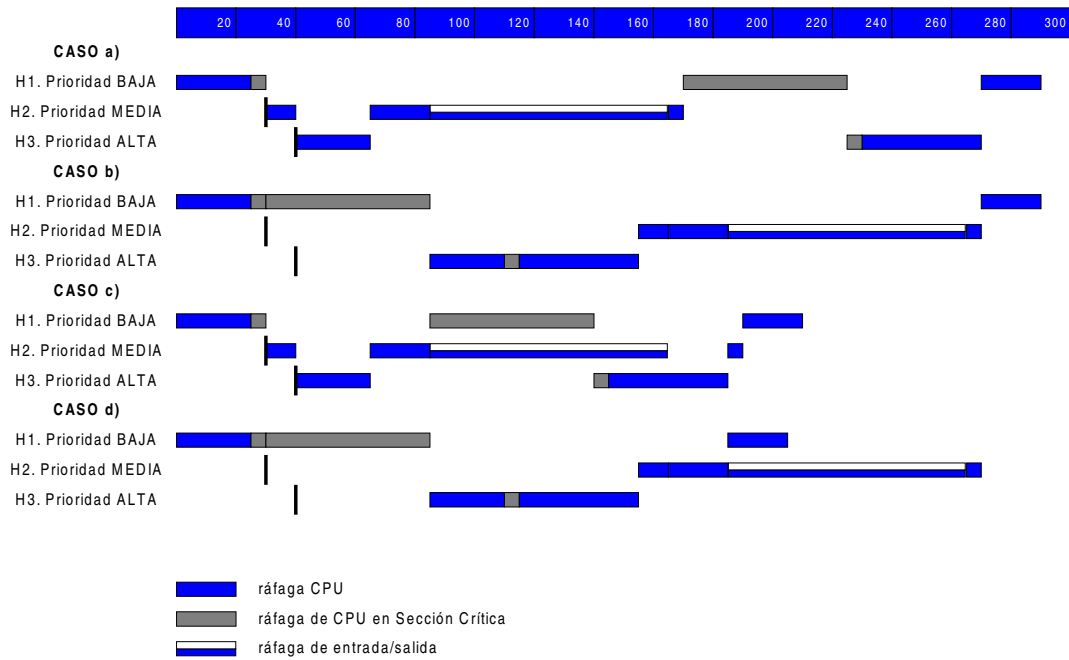
semaforo *crear_semaforo(int valor_inicial) {
    if (valor_inicial < 0) return NULL;
    semaforo *sem = malloc(sizeof(semaforo));
    pipe(sem->fd);
    while (valor_inicial-- > 0)
        write(sem->fd[1], "a", 1);
    return sem;
}

void wait(semaforo sem) {
    char buf[1];

    read(sem->fd[0], buf, 1);
}

void post(semaforo sem) {
    write(sem->fd[1], "a", 1);
}
```

3.



tej	tr	te	trn	ten	Tr-avg	Te-avg	Tm-avg	Ten-avg
105	290	185	2,76	1,76				
115	140	25	1,22	0,22	220	123,33	2,43	1,43
70	230	160	3,3	2,3				
105	290	185	2,76	1,76				
115	140	125	2,09	1,09	215	118,33	2,16	1,16
70	115	45	1,64	0,64				
105	210	105	2	1				
115	160	45	1,39	0,39	171,6	75	1,82	0,82
70	145	75	2,07	1,07				
105	205	100	1,95	0,95				
115	240	125	2,09	1,09	186,6	90	1,89	0,89
70	115	45	1,64	0,64				