	Sistemas Operativos. 6 de septiembre de 2006	
	Nombre _____ Apellidos _____	
	DNI _____ Titulación _____ Grupo _____	

TEORÍA

1. (Ficheros) 2 puntos

- Indique qué información de un fichero se guarda en una entrada de directorio en un sistema de ficheros tipo UNIX y en otro tipo FAT.
- En un sistema de ficheros tipo UNIX, ¿cuántas operaciones de disco se necesitarían, como mínimo, para obtener el primer bloque de datos del fichero **/home/usr/paco/so.deb** suponiendo que en memoria se encuentra el i-nodo del directorio raíz pero ninguna otra cosa de las necesarias para recorrer la ruta? Justifique su respuesta.

2. (Entrada/Salida) 2 puntos

- El software de E/S está constituido por una serie de capas funcionales que permiten a los procesos de usuario acceder a los dispositivos físicos. Parte de esas capas son “dependientes de dispositivo” y parte son “independiente de dispositivo”. ¿Qué funciones se reconocen como “independientes de dispositivo”?
- En UNIX los ficheros de dispositivo de E/S están caracterizados con un *major number* y un *minor number*. ¿Cuál es el cometido de dichos números?

3. (Procesos) 2 puntos

Considere que se dispone de las siguientes operaciones internas del sistema operativo:

- Reservar y/o liberar una entrada en la tabla de procesos
 - Rellenar y/o actualizar el BCP
 - Insertar/eliminar un proceso de una cola de procesos
 - Cambiar de contexto
 - Planificar
 - Leer e interpretar un ejecutable
 - Crear pila (especifique su contenido inicial)
 - Crear una región de memoria
 - Compartir/duplicar una región
 - Eliminar una región de memoria
- Se pide especificar, basándose en las anteriores operaciones, cómo se llevan a cabo las llamadas FORK y EXEC. Comente brevemente cada operación.
 - Dado el siguiente código, indicar cuál es el árbol de procesos que se obtiene y el valor final de la variable “j” en cada uno de ellos. Justifique su respuesta.

```
int main() {
    int i, j=0;
    for (i=1; i<4; i++) {
        if (fork() == 0) { j++; }
        else { exit(1); }
        j++;
    }
}
```

4. (Sincronización) 2 puntos

Considere el siguiente código:

```
Proceso A () {
    wait(s);
    escribir( "A");
    signal(t);
}

Proceso B() {
    wait(t);
    wait(t);
    escribir("B");
    signal(s);
    signal(s);
}
```

- Si los semáforos fueron inicializados con los valores $s=2$ y $t=0$, indique qué secuencia de caracteres se escribiría al lanzar en paralelo tres instancias del proceso A y una del B.
- ¿Cómo se implementaría la operación `wait()` de un semáforo con espera activa empleando la instrucción máquina `test&set`? ¿Y cómo se implementaría con espera pasiva?

5. (Memoria) 2 puntos

- Indique al menos cuatro tipos de regiones que pueden formar parte del mapa de un proceso. De cada una de ellas comente si tienen soporte, si son privadas o compartidas, su protección y si su tamaño es fijo o variable.
- Considere el siguiente código:

```
char A[4096*6] = {75, 23, 15.....}; /* vector inicializado */
int x;
int main() {
    int i;
    for (i=0; i<2; i++)
        crear_thread(procesar,i);
    .....
}
void procesar(int index) {
    int elem,i;
    elem=index*4096*3;;
    for (i=elem; i<(elem+4096*3); i++)
        A[i]=A[i]*33;
}
```

Si el tamaño de una página es de 4096 bytes, indique cómo sería el mapa de memoria del proceso (tipo regiones y número de páginas de cada región) una vez que se han creado los dos threads "procesar" pero aún no han ejecutado ninguna sentencia de su código.

SOLUCIONES

1. Ficheros

- a. UNIX: nombre de fichero y número de nodo-i.
FAT: nombre de fichero – entrada FAT del primer bloque – atributos – tamaño- fechas
- b. Hay que leer: bloque de datos de "/" y el nodo-i y bloque de datos de "home", "usr" y "paco". Con eso obtenemos el número de nodo-i de "so.deb". Queda leer el nodo-i de so.deb y su primer bloque de datos. Total: 9 lecturas a disco.

2. Entrada/Salida

- a. Servicios (API), Gestor de archivos (nombrado, protección, control de acceso), gestor de bloques (bloques comunes), gestor de cache (acelerar operaciones), gestor de red y parte del manejador del dispositivo (transparencias 23y 24. Libro pág)
- b. El *major number* identifica una clase de dispositivo (y por tanto un manejador). El *minor number* identifica un dispositivo concreto dentro de una clase (que da información más específica al manejo (transparencias 26, 45. Libro pág.)

3. Procesos

- a. **FORK**: 1, 2 (copia valores del padre, pero modificando algunos específicos del hijo, como el PID), 9 (se duplican las regiones privadas del padre, se comparten las compartidas), 7 (la nueva pila es una copia de la del padre), 3 (insertamos el nuevo BCP en la cola de procesos listos). Podría darse una nueva planificación (en Linux, será siempre el hijo el que continúe la ejecución) **EXEC**: 6 (lo que puede implicar poner el proceso en la cola de bloqueados, y planificar otro proceso), 10, 8 (creamos las regiones que estuvieran definidas en el ejecutable), 7 (la pila contendrá los argumentos del programa y el entorno), 2 (actualizar el BCP del proceso con la nueva información sobre el mapa de memoria, contador del programa, puntero a pila...)
- b. Árbol generado: Proc Original → Proceso 1 → Proceso 2 → Proceso 3

$$j=0 \qquad i=1, j=2 \qquad i=2, j=4 \qquad i=3, j=6$$

4. Sincronización

- a. AABA
- b. Con espera activa:

```
wait(lock) {
    while(test-and-set(&lock)) { }
}
```


Sin espera activa:

```
wait(s) {
    while(test-and-set(&valor_s)) { }
    s=s-1;
    if(s<0) {
        valor_s=false;
        Bloquear al proceso
    }
    valor_s=false;
}
```

5. Memoria

- a. Código (con soporte, compartida, lectura/ejecución, tamaño fijo)
Datos con valor inicial (soporte, privada, lect/esc, tamaño fijo)
Datos sin valor inicial (sin soporte, privada, lect/esc, tamaño fijo)
Pila (sin soporte, privada, lect/esc, tamaño variable)
Otras: heap, memoria compartida, pilas de threads.
- b. Regiones:
Código: 1 página
Datos con valor inicial: 6 páginas
Datos sin valor inicial: 1 página
Pila proceso original: 1 página
Pila de primer thread: 1 página

Pila de segundo thread 1 página

	Sistemas Operativos. 6 de septiembre de 2006	
	Nombre _____ Apellidos _____	
	DNI _____ Titulación _____ Grupo _____	

PROBLEMAS

1. (Ficheros - Procesos) 5 puntos Considere el siguiente código:

```
int fd;
void main() {
    int pid;

    fd = open("tarres", O_RDWR);
    pid = fork();

    if (pid==0) {
        escribe_hijo();
        close(fd);
        exit(0);
    }

    else {
        wait();
        lseek(fd,10270,SEEK_CUR); //avanza puntero 10270 bytes
        escribe_padre();
        close(fd);
    }
}
```

La función “escribe_hijo” (resp. “escribe_padre”), escribe 50 veces “hijo ” (resp. “padre ”) en el fichero apuntado por *fd* (la cadena de caracteres “hijo “ ocupa 5 bytes; “padre “ ocupa 6 bytes). El programa se ejecutará bajo un sistema operativo Linux/UNIX, con un sistema de ficheros basado en nodos-i, y con un tamaño de bloque de 1KB (1024 bytes). Cada nodo-i tiene 6 punteros directos a bloques de datos y 1 indirecto a una tabla de segundo nivel. Inicialmente, el fichero “tarres” tiene ocupado su primer bloque lógico (bloque físico 122) con 1024 caracteres “a” y su tercer bloque lógico (bloque físico 53) con 1024 caracteres “c”.

- (0.5p) ¿Se creará algún fichero nuevo tras la ejecución del código? Justifique su respuesta
- (1.5p) ¿Cuál sería el aspecto final del sistema de ficheros tras ejecutar el código? Indique el contenido de la tabla de punteros a bloques del nodo-i de “tarres”, así como el contenido de los bloques de datos con información relativa al fichero.
- (1p) ¿Cuántos bloques físicos de datos ocupa el fichero finalmente? ¿Coincide el número de bloques físicos ocupados con el de bloques lógicos? Justifique su respuesta
- (1p) Suponga que la orden “*wait*” se sitúa tras la llamada a “*escribe_padre*”. Responda nuevamente al apartado b) asumiendo que el proceso hijo tiene más prioridad que el padre, y que cada operación de E/S provoca un cambio de planificación.
- (1p) Codifique una versión funcionalmente equivalente al programa original, pero usando procesos ligeros (*threads*) en lugar de procesos. Use la función “*crear_thread(nombre_de_funcion)*” para crear un nuevo hilo y “*espera_hilo()*” para esperar la finalización de un hilo hijo.

2. **(Sincronización) 5 puntos** Los clientes que llegan a una oficina de correos para enviar sus giros postales siguen la siguiente conducta:
- En principio tienen que ir a un mostrador en el que se les proporciona un impreso que a continuación deben cumplimentar. Sólo hay **un expendedor de impresos**.
 - Para cumplimentar los impresos se dispone únicamente de **3 bolígrafos**, dispuestos sobre la mesa. Cuando un cliente acude a la mesa y no encuentra ninguno disponible, debe esperar a que algún otro cliente termine de utilizarlo.
 - Una vez cumplimentado el impreso, el cliente dispone de **dos ventanillas** para el pago de la cantidad que desea enviar. El cliente se sitúa en la ventanilla que tiene **menos gente** esperando (no pudiendo cambiarse ya de cola)
 - Finalmente, una vez abonada la cantidad correspondiente, se dirige a **un único buzón**, donde deposita su impreso, junto con el resguardo del pago.

Se va a proceder a modelar la situación anterior mediante el uso de semáforos. Considere el siguiente pseudocódigo del proceso CLIENTE:

```
Proceso CLIENTE {  
  
    <Recoger impreso>  
  
    <Rellenar impreso>  
  
    <Pagar>  
  
    <Depositar impreso en buzón>  
  
}
```

- (2p) Indique qué variables utilizará para la sincronización, especificando su tipo (entero o semáforo), su valor inicial y su cometido en el problema.
- (3p) Implemente el problema completando el pseudocódigo con las estructuras de sincronización necesarias (usando las variables definidas en el apartado anterior)

SOLUCIONES

1.

a) No. Sólo se referencia a “tarres” ya escrito, duplicando el descriptor de fichero

b) Bloque lógico 0 (físico 22) -> 50 veces “hijo “ (250 bytes) y 774 “a”

Bloque lógico 2 (físico 55) -> 1024 “c”

Bloque lógico 10 (necesita usar el puntero indirecto; quinta entrada de la tabla de punteros).: 280 espacios en blanco, 50 veces “padre “ (300 bytes) y final de fichero.

El resto de bloques lógicos NO se usan (no están ocupados ni referenciados. No se reserva ningún espacio para ellos en el disco)

c) Bloques físicos de datos: 3 (4 si contamos el bloque de punteros indirectos. No son datos del fichero, pero está en la zona de Bloques de Datos del sistema de ficheros).

Tenemos 11 bloques lógicos (del 0 a 10) ocupados, pero sólo 3 físicos. Si utilizásemos un sistema FAT, el número de bloques lógicos sí coincidiría con el de bloques físicos.

d) Bloque lógico 0 (físico 22) -> 1 vez “hijo “ (5 bytes) y 1019 “a”

Bloque lógico 2 (físico 55) -> 1024 “c”

Bloque lógico 10 (necesita usar el puntero indirecto; quinta entrada de la tabla de punteros).: 35 espacios en blanco, 49 veces “padre hijo “ (539 bytes) y final de fichero.

```
e) main () {
    thread_id tid;
    fd=open("tarres",O_RDWR);
    tid=crear_thread(func);
    espera_hilo(tid); // join en POSIX
    lseek(...);
    escribe_padre();
    close(fd);
    exit(1);
}

func() {
    escribe_hijo();
    [ pthread_exit(); ] //opcional
}
```

NOTA: la única diferencia es el que el thread “hijo” NO debe realizar un “close”, ya que no permitiría al padre realizar la escritura (pues comparten la tabla de descriptores abiertos).

2.

a) Semáforo y valor inicial

RI=1 //exclusión mutua al recoger impreso

BOLI= 3 //ex. Mutua para coger un bolígrafo y rellenar impresos

exmut=1 // garantiza ex. Mut en el acceso a variables nv1 y nv2

(compartidas)

V1 = 1 // exmut ventanilla 1

V2 = 1 // ex mut. Ventanilla 2
 DB = 1 // ex mut en uso de buzón

Variables enteras:

nv1 = 0; // número de personas en la cola de ventanilla 1
 nv2 = 0; // número de personas en la cola de ventanilla 2

b)

```
Proceso CLIENTE {
    wait(RI);
    <Recoger impreso>
    signal(RI);

    wait(BOLI);
    <Rellenar impreso>
    signal(BOLI);

    wait(exmut);
    if (nv1 >= nv2) {
        nv2++;
        signal(exmut);
        wait(v2);
        <Pagar >
        nv2--; //podría usarse exmut para garantizar excl. mutua
        signal(v2);
    }
    else {
        nv1++;
        signal(exmut);
        wait(v1);
        <Pagar >
        nv1--; //podría usarse exmut para garantizar excl. mutua
        signal(v1);
    }
    wait(DB);
    <Depositar impreso en buzón>
    signal(DB);
}
```