UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

Calificación
1
2
3
4

Fac. de Informática / Escuela Univ. Informática

SISTEMAS OPERATIVOS

Examen Parcial

9 de mayo de 2009

Nombre	Titulación
*** SOLUCIONES ***	

Dispone de tres horas para realizar el examen

---

1 (2.5 puntos) Comentar la validez de la siguiente solución al *problema de la sección crítica para dos procesos*:

Variables compartidas

Int n1=0, n2=0;

```
process P1() {
while (true) {
        //Sección no crítica
        n1:=1;
        n1:=n2+1;
        while(n2<>0) and (n2<n1) do null;
        //Sección Crítica1;
        n1:=0;
        //Sección no crítica
}
}
```

```
process P2() {
while (true) {
        //Sección no crítica
        n2:=1;
        n2:=n1+1;
        while (n1<>0) and (n1<=n2) do null;
        //Sección Crítica2;
        n2:=0;
        //Sección no crítica
}
}
```

Para demostrar que se trata de una solución válida al problema de la sección crítica vamos a comprobar que se cumplen las propiedades de exclusión mutua, progreso y espera limitada. Antes de ver cada una de estas tres propiedades conviene resaltar si analizamos el preprotocolo y postprotocolo de ambos procesos que cada uno de ellos antes de entrar en la sección crítica, inicializa una variable entera (n1 para P1 y n2 para P2) a 1. Estas variables funcionan como indicadores a través de los cuales, cuando tienen un valor mayor que cero, indican la intención de cada proceso de intentar acceder a la sección crítica. La intención de entrar de cada proceso se manifiesta inicializando su correspondiente bandera a 1, para a continuación asignarle el valor inmediatamente superior al del proceso opuesto. Dicho de otra forma, cada proceso coge número antes de entrar, siendo este número el siguiente al que tiene el proceso opuesto. En definitiva se trata de una versión del algoritmo de la panadería para dos procesos.

- Exclusión mutua: veamos por ejemplo que ocurre si el proceso P1 accede a la sección crítica y a continuación el proceso 2 realiza un intento de entrada. Si P1 ejecuta el preprotocolo de la sección crítica, la variable n1 toma el valor 1 (asumiendo que n2 vale 0 al no haber ejecutado éste el preprotocolo). El proceso P1 rebasaría el bucle "while" ya que cuando n2 vale 0 se entiende que el proceso P2 no tiene intención de entrar en la sección crítica y por tanto no hay problema para la entrada del proceso P1. Si estando P1 en SC el proceso P2 ejecuta el preprotocolo de entrada a la SC, vemos como n2 toma el valor 2 y por tanto el proceso 2 quedaría iterando en el bucle "while" de su preprotocolo ya que su número es mayor que el del proceso P1 (n2>=n1) y n1 es distinto de cero (n1<>0). Lo mismo ocurriría exactamente si entra en sección crítica P2 y fuera el proceso P1 el que intentara entrar en SC.
- Progreso: Se puede ver claramente que si un proceso ejecuta el preprotocolo que para entrar en SC y el otro proceso está en su sección NO crítica, su bandera sería 0 y ello no impediría la entrada del proceso que tiene intención de ejecutar la SC. Por tanto la decisión de entrar en la sección crítica depende exclusivamente de aquellos que quieren entrar. Si ambos intentaran acceder a la SC concurrentemente y ejecutaran el

preprotocolo, la decisión de quien entra se toma en un tiempo finito dado que aunque los dos podrían tener el mismo número (igual valor en n1 y n2), las condiciones del bucle con diferentes: "n2<n1" en el proceso P1 y "n1<=n2" en el proceso P2 y por tanto en ese caso entraría primero el proceso P1 y a continuación P2.

- Espera limitada: Si ambos procesos en un momento determinado manifiestan intención de entrar y toman el mismo número, entrará primero el proceso P1 y después aunque este intente acceder nuevamente a la SC, esperará a que entre P2 ya que tomaría un número mayor del que dispone el proceso P2.

Por tanto, podemos decir que la solución planteada verifica las tres propiedades y se trata de una solución por software válida al problema de la sección crítica para dos procesos.

---

2 (2.5 puntos) Una empresa de logística quiere llevar a cabo un control sistemático de su nave en la que se almacenan los productos que distribuye. La nave se debe gestionar de la siguiente manera:

- En la nave se almacenan existencias de un solo producto.
- A la nave acceden vehículos de proveedores y vehículos de compradores. Los vehículos proveedores solicitan el depósito de la mercancía mediante la función **`Solicito_Almacen(cantidad)`**. Los vehículos clientes solicitan retirar un pedido mediante la función **`Solicito_Pedido(cantidad)`**. Siendo Cantidad un valor entero positivo que especifica el número de unidades a depositar o retirar.
- Para que un vehículo proveedor pueda depositar su carga debe haber capacidad en el almacén para recoger toda su carga. Si no hay capacidad suficiente, entonces el vehículo debe quedar a la espera hasta que la haya.
- Para que un vehículo cliente pueda retirar su pedido en la nave debe haber existencias para atender a la totalidad de su pedido. Si no hay existencias suficientes, entonces debe esperar hasta que las haya.

Se pide que implemente las funciones **`Solicito_Almacen(cantidad)`** y **`Solicito_Pedido(cantidad)`** empleando semáforos como herramienta de sincronización.

La solución que se muestra a continuación es uno de los muchos algoritmos posibles y no pretende ser la solución más eficiente. Se trata de una propuesta que cumple los requisitos indicados en el enunciado. En el código propuesto se ha supuesto que el almacén está inicialmente vacío siendo "N" la capacidad total del mismo.

Variables compartidas

```
Int Stock_Artículos = 0;
Int Huecos_Artículos = N;
Int proveedores_bloqueado=0;
Int compardores_bloqueados=0;
Semáforo bloqueo_proveedor ; // Valor inicial a 0
Semáforo bloqueo_comprador; // Valor inicial a 0
Semáforo mutex; // Valor inicial 1
```

```
Solicito_Alamcen( int  cantidad ){
P(mutex);
while ( Huecos_Artículos < cantidad ) {
        Proveedores_bloqueados++;
        V(mutex);
        P(bloqueo_proveedor);
        P(mutex);
}
Stock_Artículos +=  cantidad;
Huecos_Artículos  -=  cantidad;
while ( compradores_bloqueados > 0) {
        V(bloqueo_comprador);
        Compradores_bloqueados--;
}
V(mutex);
}
```

```
Solicito_Pedido(int cantidad) {
P(mutex);
while ( Stock_Artículos < cantidad ) {
        Compradores_bloqueados++;
        V(mutex);
        P(bloqueo_comprador);
        P(mutex);
}
Stock_Artículos -=  cantidad;
Huecos_Artículos + = cantidad;
while ( proveedores_bloqueados > 0 ) {
        V(bloqueo_proveedor);
        Proveedores_bloqueados--;
}
V(mutex);
}
```

---

3 **(2.5 puntos)** Contestar brevemente a las siguientes cuestiones:

a) ¿Sería viable un sistema multiprogramado sin el uso de las interrupciones? (0.5p)

Si un sistema no posee interrupciones, entonces la detección de todos aquellos eventos que son relevantes de cara a su funcionamiento se tendría que realizar mediante la técnica de interrogación o comprando de forma permanente el valor de las variables de estado del sistema. Además, en muchas ocasiones, estas detecciones se realizarían dando lugar a situaciones de espera activa. Por tanto el sistema no tendría un funcionamiento eficiente. Como en un sistema multiprogramado el manejo de eventos juega un papel crítico en su eficiencia, entonces un sistema de este tipo sin interrupciones no sería viable debido a que no tendría un rendimiento aceptable.

b) Estamos desarrollando un intérprete de órdenes o *shell*. Necesitamos escribir el código que lanza el programa que el usuario le encarga ejecutar al *shell*. Para ello podemos elegir entre lanzar un nuevo hilo, o bien lanzar un nuevo proceso pesado. ¿Cuál de las dos opciones le parece más adecuada? (0.5p)

La opción correcta sería lanzar un nuevo proceso pesado, debido a que en general el programa a ejecutar nada tendría en común (código y datos) con respecto a la unidad de ejecución que lo crea. Por tanto, la facilidad para compartir datos y código, característica de los hilos, no tendría sentido en esta situación.

c) En un sistema multiprocesador, para planificar procesos podemos, bien utilizar una única cola de preparados compartida por todos los procesadores, o bien que cada procesador disponga de su propia cola de preparados. ¿Qué ventajas e inconvenientes tiene cada una de estas dos técnicas? En un ordenador tipo PC, ¿qué sistema escogería usted? (0.5p)

Si cada procesador dispone de su propia cola de preparados, debemos gestionar el problema de que pueden darse desequilibrios en la carga de trabajo: un procesador podría llegar a quedarse ocioso, sin procesos en cola, mientras que otro podría seguir cargado. Este problema no ocurriría con una cola única, ya que cada vez que un procesador quedara libre, tomaría uno de los procesos de la cola única y por tanto no se daría la situación de tener a un procesador más cargado de trabajo que otro.

Por otra parte, el esquema de la cola única tiene el inconveniente de que se trata de un recurso compartido por todos los procesadores y hay que controlar el acceso simultáneo a la estructura, para evitar que se corrompa o que ocurran usos indebidos (p.ej. que varios procesadores decidan ejecutar el mismo proceso). Este inconveniente es tanto más grave cuanto mayor sea el número de procesadores del sistema y más frecuentes sean los cambios de contexto.

En un PC multiprocesador y con la tecnología actual tenemos pocos procesadores (dos, cuatro, a lo sumo ocho), así que probablemente el uso de una cola única es más eficiente, ya que no hay demasiada competencia por la cola.

d) ¿Qué aporta la instrucción test-and-set para solucionar el problema de la sección crítica? (0.5p)

La instrucción test-and-set permite realizar de forma atómica dos acciones sobre una variable: recuperar su valor original y modificarlo. Gracias a la indivisibilidad de estas dos acciones, podemos construir un algoritmo sencillo de entrada a la sección crítica en el que utilizamos un indicador booleano global para marcar si la sección crítica está ocupada.

Como muestra, veamos este algoritmo de sección crítica que usa test_and_set:

```
a1. // variable global
a2. bool SC_Ocupada = false;
a3. while ( test_and_set (&SC_Ocupada) ) {}
a4. ... sección crítica ...
a5. SC_Ocupada = false;
```

El código anterior es equivalente algorítmicamente a este otro:

```
b1. // variable global
b2. bool SC_Ocupada = false;
b3. while ( SC_Ocupada ) {}
b4. SC_Ocupada = true;
b5. ... sección crítica ...
b6. SC_Ocupada = false;
```

La diferencia es que en el último ejemplo, las líneas b3 y b4 constituyen una sección crítica que no está controlada y permite que varios procesos observen al mismo tiempo que la variable SC_Ocupada está a false (línea b3) y por tanto escapen todos del bucle de espera y entren simultáneamente en la sección crítica. En el código de ejemplo con test_and_set, la evaluación y manipulación de SC_Ocupada ocurre en un solo paso indivisible (línea a3). Por tanto, ante una multitud de procesos que quieran entrar en sección crítica estando SC_Ocupada a false, sólo uno de ellos se la encontrará con ese valor y podrá escapar del bucle.

Por tanto, la utilización de test_and_set permite simplificar el diseño de algoritmos que regulan el acceso a secciones críticas, y ahorrarnos el esfuerzo de construir algoritmos complejos de entender, ejecutar y mantener.

e) Se considera un algoritmo de planificación del tipo Round Robin de quantum *q*. ¿Puede afirmarse entonces que todo proceso de la cola de preparados disfrutará de *q* unidades de tiempo el recurso CPU? Asimismo, ¿puede ocurrir que un proceso llegue a utilizar la CPU *n* unidades de tiempo sin interrupción, siendo n>q? (0.5p)

Realmente, no es correcta del todo la afirmación, o al menos no podemos asegurar que siempre ocurra. Pueden darse los siguientes casos:

a) Que disfrute exactamente *q* instantes de tiempo de CPU.

b) Que disfrute menos de *q* instantes de tiempo debido a que el proceso finaliza antes de finalizar su quantum asignado o se bloquea esperando por algún evento, por ejemplo, una operación de entrada salida o una operación P sobre un semáforo cuyo valor es cero.

c) Que disfrute más de *q* instantes de tiempo debido a que el proceso no ha finalizado aún y que no hayan más procesos en la *cola de preparados.*

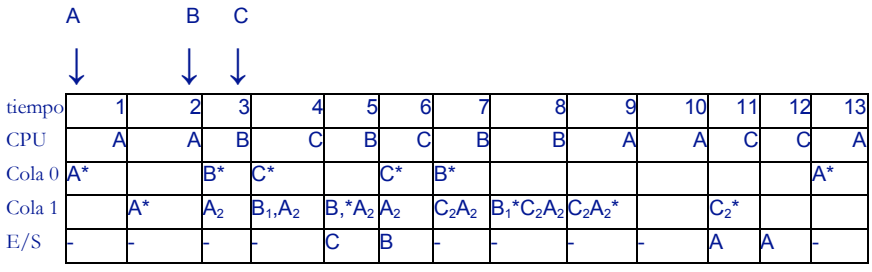
Luego, debido a la razón c) del apartado anterior, sí que puede ocurrir que un proceso disfrute más de un número *q* de instantes de tiempo de CPU.

---

4 (2.5 puntos) Sea un algoritmo de planificación multicolas con realimentación donde la primera cola (cola 0) se gestiona con un Round-Robin de cuanto q=1, y la segunda cola (cola 1) se gestiona con un algoritmo SRTF. La planificación entre colas es del tipo prioridades expulsiva, siendo la cola más prioritaria la 0. Un proceso pasará de la cola 0 a la 1 cuando se agote su cuanto q sin finalizar su ejecución. Los procesos nuevos, y los procedentes del estado SUSPENDIDO, entran por la cola 0. Suponiendo que el sistema operativo no consume tiempo y que las operaciones de E/S se efectúan sobre el mismo dispositivo (gestionado de forma FCFS), se pide obtener el diagrama de Gantt y los tiempos medio de retorno y espera al aplicar la siguiente carga de trabajo:

Proceso	Tiempo de llegada	Duración ráfaga de procesador y entrada/salida
A	0	4(CPU) + 2(E/S)+ 1(CPU)
B	2	2(CPU) + 1(E/S) + 2(CPU)
C	3	1(CPU) + 1(E/S) + 3(CPU)

1)Diagrama de *Gantt*



El * representa el proceso que está en CPU, dentro de cada clase/cola.

2) Tiempo medio de espera

	Tiempos de Espera
A	8-2 = 6
B	4-3 =1
C	10-6=4
Tiempos Medios	(6+1+4)/3 = 3,67

3) Tiempo medio de retorno

	Tiempos de Retorno
A	13-0=13
B	8-2=6
C	12-3=9
Tiempos Medios	(13+6+9)/3=9,33