Ejercicios-examen.pdf

Sanchez01

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Ejercicios de examen

PROBLEMAS DE MONITORES 1

El almacén frigorífico de un supermercado es accedido por los empleados del mismo y por los suministradores. Decimos que el almacén está **lleno cuando hay 20 o más cajas** en el mismo. Los empleados invocan el procedimiento **Retirar** para retirar una caja cada vez. Si no hay cajas en el almacén, se quedan bloqueados en espera de que lleguen cajas. Los suministradores pasan de forma periódica por el almacén, invocando al procedimiento **Depositar**. En el procedimiento **Depositar, si el almacén está lleno, los suministradores no hacen nada** (no tienen porque esperar si el almacén está lleno). Si el almacén no está lleno, **los suministradores depositan normalmente una caja, excepto una de cada cuatro veces que depositan, vez en la cual depositan tres cajas** en lugar de una. Por tanto, depositan tres veces seguidas una caja y en la cuarta vez se depositan tres cajas de golpe, luego el proceso se repite de nuevo. Esto es independiente de qué suministrador concreto deposite, o de si el mismo lo hace varias veces seguidas o se alternan varios suministradores. Suponiendo que inicialmente el almacén tiene 20 cajas, diseñar el monitor Almacén, con semántica SU, que resuelva la sincronización requerida para el problema de acuerdo al siguiente esquema:

```
process Empleado[ i : 1..n]
begin
    while true do begin
        Almacen.retirar();
    end
end
```

```
process Suministrador []
begin
    while true do begin
        Almacen.depositar();
    end
end
```

```
Monitor Almacen
int num_cajas=20;
Condition cola_empleados;
int num_suministros = 0;
```

```
Procedure retirar(){
    if (num_cajas == 0){
      cola_empleados.wait();
    }
    num_cajas--;
    if (num_cajas >0){
      cola_empleados.signal();
    }
}
```

```
Procedure depositar(){
    if (num_cajas <20){
        // Incrementamos el contador
        num_suministros = num_suministros+1;
        if(num_suministros == 4){
            num_cajas +=3;
            num_suministros = 0;
        }else{
            num_cajas++;
        }
        cola_empleados.signal();
    }
}
```

Resueltos por José Miguel Mantas

PROBLEMAS DE MONITORES 2

Una tienda de licores es frecuentada por varios clientes (procesos **Cliente[i]**, hay N de ellos), que periódicamente la visitan para comprar botellas de licor. El tendero (proceso de nombre **Tendero**) dispone de poco espacio para almacenar botellas (**sólo le caben 30 botellas** de licor) y sólo se puede atender a los clientes de uno en uno (que solo pueden comprar **una botella cada iteración**). Para poder contentar a la mayor parte de clientes, **no se permite vender dos o más botellas seguidas al mismo cliente** (es decir, una vez que un cliente compra una botella tiene que esperar a que cualquier otro cliente compre una botella para poder comprar la siguiente), pero **cuando quedan menos de 5 botellas se ignora** esa norma y puede comprar cualquiera. Los clientes compran la botella directamente, sin interactuar con el **tendero, que permanece dormido (bloqueado) hasta que se vacía el almacén, momento en el cual debe rellenarlo**, se vuelve a dormir, y se vuelve a impedir comprar dos veces seguidas al mismo cliente (otra vez hasta que quedan menos de 5 botellas). Siempre que se rellena la tienda, cualquier cliente puede comprar la primera botella. Implementar el monitor Licorería, para que resuelva la sincronización requerida de acuerdo con el código de los procesos que se presenta a continuación. Suponer que la tienda está inicialmente llena y que la semántica de las señales del monitor es señalar y espera urgente (SU).

```
process Cliente[i : 1..N ]
begin
    while true do begin
        /*aquí va el resto de código cliente*/
        Licoreria.comprar(i);
    end
end
```

```
process Tendero;
begin
    while true do begin
        Licoreria.rellenar();
    end
end
```

```
Monitor Licoreria
int num_botellas = 30;
int anterior = -1; //Cliente anterior
Condition cola_clientes, cola_tendero;
```

```
procedure comprar(int i){
    if(num_botellas == 0 or (anterior ==i) and
    (num_botellas <= 5) ){
        cola_clientes.wait();
    }
    //Ahora sabemos que:
    /*
    * num_botellas >0 and (anterior != i or num_botellas<5)
    */
    num_botellas--;
    anterior = i;
    if(num_botellas == 0){
        cola_tendero.signal();
    }else{
        cola_clientes.signal();
    }
}
```

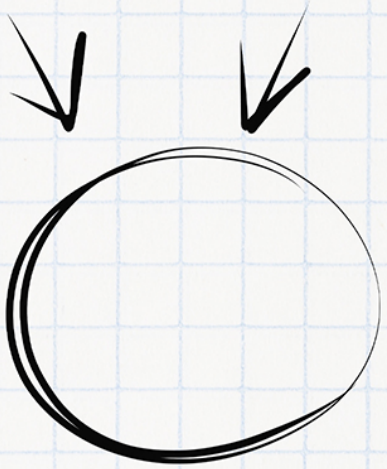
```
procedure rellenar(){
    if (num_botellas >0)
        cola_tendero.wait();
    num_botellas = 30;
    anterior = -1;
    cola_clientes.signal();
}
```

Imagínate aprobando el examen

Necesitas tiempo y concentración

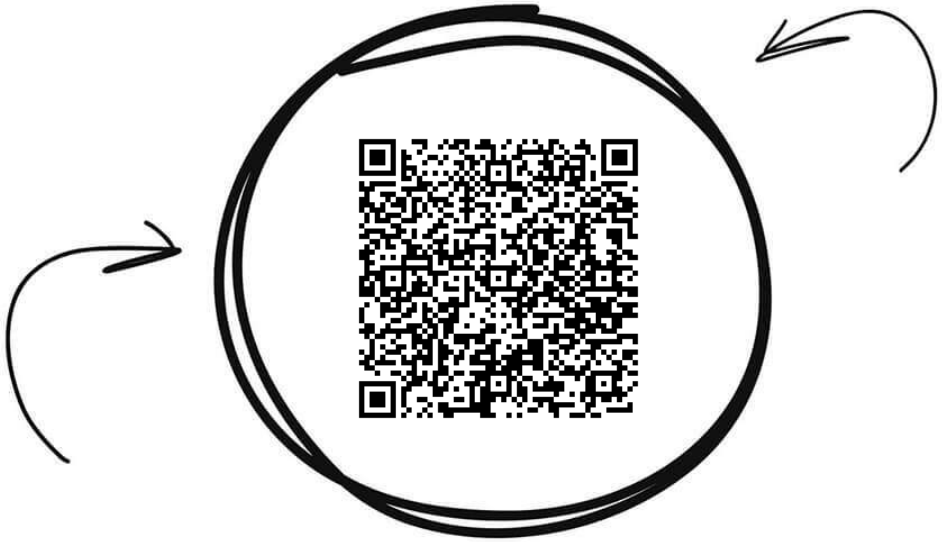
Planes	PLAN TURBO	PLAN PRO	PLAN PRO+
Descargas sin publi al mes	10	40	80
Elimina el video entre descargas	✓	✓	✓
Descarga carpetas	✗	✓	✓
Descarga archivos grandes	✗	✓	✓
Visualiza apuntes online sin publi	✗	✓	✓
Elimina toda la publi web	✗	✗	✓
Precios Anual	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo, ¿Qué nota vas a sacar?



WUOLAH

Sistemas Concurrentes y Dist...



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

PROBLEMAS DE PASO DE MENSAJES 1

Una tienda de licores es frecuentada por varios clientes (procesos **Cliente[i]**, hay N de ellos), que periódicamente la visitan para comprar botellas de licor. El tendero (proceso de nombre **Tendero**) dispone de poco espacio para almacenar botellas (**sólo le caben 30 botellas** de licor) y sólo se puede atender a los clientes de uno en uno (que solo pueden comprar una botella cada iteración). Para poder contentar a la mayor parte de clientes, no se permite vender dos o más botellas seguidas al mismo cliente (es decir, una vez que un cliente compra una botella tiene que esperar a que cualquier otro cliente compre una botella para poder comprar la siguiente), pero cuando quedan menos de 5 botellas se ignora esa norma y puede comprar cualquiera. Siempre que se rellena la tienda, cualquier cliente puede comprar la primera botella. **Será el proceso Tendero el encargado de vender las botellas a petición de los clientes** y cuando se vacía el almacén, y sólo en ese momento, es el propio tendero el que lo rellena. Implementar los procesos Cliente[i] y Tendero[i] usando un paso de mensajes síncrono (**s_send**) y espera selectiva (**select**), de forma que no haya interbloqueos y se cumplan las características del problema.

```
process Cliente [i: 0--N-1]{
      while true do{
          //Resto de código cliente
          <comprar ron>
      }
}
```

```
process Tendero{
      while true do{
          <vender licor o llenar almacen>
      }
}
```

```
process Cliente[i: 0..N-1]{
    int botella;
    while true do{
        //Resto código cliente
        s_send(botella, Tendero);
    }
}
```

```
process Tendero{
    int ultimo =-1;
    int botella, n_botellas = 30;
    while true do{
        select for i =0 to N-1
            when (n_botellas>0 and (n_botellas<5 or
            ultimo!=i)), do receive(botella, Cliente[i]) do{
                n_botellas--;
            }
            if(n_botellas == 0){
                n_botellas = 30;ultimo = -1;
            }else
                ultimo = i;
    }
}
```

Solución correcta para solo 3 clientes:

```
process Cliente[i: 0..2]{
    int botella;
    while true do{
        //Resto código cliente
        s_send(botella, Tendero);
    }
}
```

```
process Tendero{
    int ultimo =-1, botella, n_botellas = 30;
    while true do{
        select
            when (n_botellas>0 and (n_botellas<5 or
            ultimo!=0)), receive(botella, Cliente[0]) do{
                n_botellas--;
                if(n_botellas == 0){
                    n_botellas = 30;
                    ultimo = -1;
                }else
                    ultimo = 0;
            }
            when (n_botellas>0 and  (n_botellas<5 or
            ultimo!=1)), receive(botella, Cliente[1]) do{
                n_botellas--;
                if(n_botellas ==0){
                    n_botellas = 30;
                    ultimo = -1;
                }else
                    ultimo = 1;
            }
            when (n_botellas>0 and (n_botellas<5 or
            ultimo!=2)), receive(botella, Cliente[2]) do{
                n_botellas--;
                if(n_botellas ==0){
                    n_botellas = 30;
                    ultimo = -1;
                }else
                    ultimo =2;
            }
    }
}
```

EJERCICIO

En un sistema distribuido, diversos procesos clientes intentan acceder para sacar e ingresar fondos a una misma cuenta compartida que es gestionada por un proceso llamado Cuenta. Existen dos tipos de clientes: los que tratan de ingresar una cantidad en la cuenta (tipo Cliente1) y los que tratan de sacar una cantidad de dinero de la cuenta (tipo Cliente2). Existen 7 procesos tipo Cliente1 que envían un mensaje al proceso Cuenta con la cantidad a ingresar, y procesos tipo Cliente2 que envían un mensaje con la cantidad a sacar al proceso Cuenta y esperan la confirmación del reintegro. El pseudocódigo de los procesos Cliente1 y Cliente2 se muestra abajo.

```
process Cliente1[ i : 1..? ]
    var cantidad : integer;
begin
    while true do begin
        /* Determina la cantidad a ingresar */
        cantidad = Calcula_Ahorro();
        s_send(cantidad, Cuenta); //Hacer
        ingreso
    end
end
```

```
process Cliente2[ i : 1..5 ]
    var cantidad, confirmacion : integer;
begin
      while true do begin
            /* Determina la cantidad a sacar */
            cantidad = Calcula_necesidades();
            s_send(cantidad, Cuenta); // Hacer
            solic. sacar
            receive(confirmacion, Cuenta); //
            Esperar conf.
      end
end
```

El proceso Cuenta mantendrá el saldo de la cuenta compartida y se encargará de modificarlo en función de las peticiones de los clientes. Adicionalmente, se tendrán que cumplir las siguientes restricciones:

1. El proceso Cuenta sólo confirmará una operación de reintegro (sacar) si la cantidad a reintegrar es menor que la cantidad disponible (el saldo) en la cuenta compartida.
2. El proceso Cuenta podrá recibir una petición de un proceso tipo Cliente2 aunque no pueda confirmarla de forma inmediata (debido a que no hay saldo suficiente para tramitarla) pero, en cuanto recibe una petición de reintegro que no puede tramitar, no recibe más peticiones de reintegro hasta que dicha petición pendiente sea confirmada.
3. Un proceso tipo Cliente2 no podrá realizar dos peticiones consecutivas de reintegro, es decir, dado un proceso cliente2[i] (con $i \in \{1, \ldots, 5\}$) que acaba de hacer un ingreso, este cliente no podrá hacer una petición adicional de ingreso hasta que otro proceso Cliente2[j] (con $i \neq j$) haya completado un ingreso.

Se desea desarrollar una implementación del proceso Cuenta para gestionar el acceso de los clientes a la cuenta compartida usando una orden de espera selectiva y paso de mensajes síncrono seguro.

```
Process Cuenta(){
    int disponible = …; // Lo que hay en la cuenta
    int ultimo2 = -1, cantidad2, confirmacion, indice_pendiente;
    int cantidad; //La que recibe de Cliente1
    boolean reintegro_pendiente = false;
    while true{
        select
            for i = 0 to 7 //Ramas para Cliente1[ i ]
                when receive(cantidad, cliente1[ i ]) do{
```

```
                disponible+=cantidad;
                if(reintegro_pendiente and disponible >= cantidad2){
                    disponible-=cantidad2;
                    s_send(confirmacion, cliente2[indice_pendiente]);
                    reintegro_pendiente = false;
                }
            }//Ramas para Cliente2[ i ]
        for i = 0 to 5
            when (ultimo2 != i and !reintegro_pendiente), receive(cantidad2, cliente2[i])
            do{
                if(disponible >= cantidad2){
                    disponible -= cantidad2;
                    s_send(confirmacion, cliente2[i];
                }else{
                    reintegro_pendiente = true;
                    indice_pendiente = i;
                }
                ultimo2 = i;
            }
    }
}
```