Concurrente- práctica

lunes, 26 de agosto de 2024 14:12

1. Para el siguiente programa concurrente suponga que todas las variables están inicializadas en 0 antes de empezar. Indique cual/es de las siguientes opciones son verdaderas:
   a) En algún caso el valor de x al terminar el programa es 56.
   b) En algún caso el valor de x al terminar el programa es 22.
   c) En algún caso el valor de x al terminar el programa es 23.

P1::	P2::	P3::

If (x = 0) then
y:= 4*2;
x:= y + 2;
	If (x > 0) then
	x:= x + 1;
		x:= (x*3) + (x*2) + 1;

P1—>p2—>p3=56
P3—>p2=2
Primera parte de p3—>p1—>p2=22

2. Realice una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el siguiente problema. Dado un número N verifique cuántas veces aparece ese número en un arreglo de longitud M. Escriba las pre-condiciones que considere necesarias.

```
Int total=0;
Int arreglo[0..m]
Int n=?
Process contador[id:0..x]
{
  For int i=0..m-1
  {
    If (arreglo[i]==n){
      <total++>
    }
  }
}
int cant = 0; int pri_ocupada = 0; int pri_vacia = 0; int buffer[N];
```

3. Dada la siguiente solución de grano grueso:
   a) Indicar si el siguiente código funciona para resolver el problema de Productor/Consumidor con un buffer de tamaño N. En caso de no funcionar, debe hacer las modificaciones necesarias.

int cant = 0; int pri_ocupada = 0; int pri_vacia = 0; int buffer[N];

Process Productor::	Process Consumidor::

{ while (true)
{ *produce elemento*
<await (cant < N); cant++>
buffer[pri_vacia] = *elemento*;
pri_vacia = (pri_vacia + 1) mod N;
}
}
	{ while (true)
	{ <await (cant > 0); cant-- >
	elemento = buffer[pri_ocupada];
	pri_ocupada = (pri_ocupada + 1) mod N;
	consume elemento
	}
	}

   b) Modificar el código para que funcione para C consumidores y P productores.

a- Hay que poner todo lo que tiene que ver con el buffer dentro de los <> porque se podría dar que primero el productor haga el await, luego el consumidor intente sacar el elemento antes de que el productor haya puesto algo ahí.

Quedaría:

```
process productor {
   while (true){
      // producir elemento
      <await (cant < N); cant++;
      buffer[pri_vacia] = elemento>
      pri_vacia = (pri_vacia + 1) mod N;
   }
}
```

```
process consumidor{
   while (true) {
      <await (cant > 0); cant—
      elemento = buffer[pri];>
      pri_ocupada = (pri_ocupada + 1) mod N;
      //consume elemento
   }
}
```

b-

Para que funcione con muchos productores y consumidores es necesario proteger la actualización del libre.

```
int cant = 0; int pri_ocupada = 0; int pri_vacia = 0; int buffer[N];
process productor[id:0..P] {
    while (true){
        // producir elemento
        <await (cant < N); cant++;
        buffer[pri_vacia] = elemento;
        pri_vacia = (pri_vacia + 1) mod N;>
    }
}
process consumidor[id:0..C]{
    while (true) {
        <await (cant > 0); cant--
        elemento = buffer[pri];
        pri_ocupada = (pri_ocupada + 1) mod N;>
        //consume elemento
    }
}
```

4. Resolver con SENTENCIAS AWAIT (<> y <await B; S>). Un sistema operativo mantiene 5 instancias de un recurso almacenadas en una cola, cuando un proceso necesita usar una instancia del recurso la saca de la cola, la usa y cuando termina de usarla la vuelve a depositar.

```
Cola c[5]
process proceso[id:0..4]{
    while (true) {
  <await Cola.lenght!=0; recurso= pop cola>
  //usar recurso
  <Push recurso>
       }
}
```

5. En cada ítem debe realizar una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el siguiente problema, teniendo en cuenta las condiciones indicadas en el item. Existen *N* personas que deben imprimir un trabajo cada una.

   a) Implemente una solución suponiendo que existe una única impresora compartida por todas las personas, y las mismas la deben usar de a una persona a la vez, sin importar el orden. Existe una función *Imprimir(documento)* llamada por la persona que simula el uso de la impresora. Sólo se deben usar los procesos que representan a las *Personas*.

```
bool ocupada=false;
process persona[id:0..n]{
  <await (ocupada=false); ocupada=true;>
```

```
  imprimir(documento)
  <ocupada=false>
}
//mucho más simple
process persona[id:0..4]{
  <imprimir(documento)>
}
```

b) Modifique la solución de (a) para el caso en que se deba respetar el orden de llegada.

```
cola cola[5]
siguiente=-1
process persona[id:0..n]{
  <if (siguiente==-1) siguiente=id
    else push cola(id)>
  <await(siguiente==id)>
    imprimir(documento)
  <if (cola.lenght==0) siguiente=-1;
    else siguiente=pop cola>
}
```

c) Modifique la solución de (a) para el caso en que se deba respetar el orden dado por el identificador del proceso (cuando está libre la impresora, de los procesos que han solicitado su uso la debe usar el que tenga menor identificador).

Esto se resuelve con que el de arriba tenga una cola que respete orden

d) Modifique la solución de (b) para el caso en que además hay un proceso *Coordinador* que le indica a cada persona que es su turno de usar la impresora.

```
process persona[id:0..n]{
  <push cola(id)>
  <await(siguiente==id), ocupada=true>
    imprimir(documento)
    ocupada=false

}
process coordinador[]{
   while (true) {
     <await (ocupada==false && cola.lenght > 0), siguiente = pop cola>
   }
}
```

Preguntaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaar

6. Dada la siguiente solución para el Problema de la Sección Crítica entre dos procesos (suponiendo que tanto SC como SNC son segmentos de código finitos, es decir que terminan en algún momento), indicar si cumple con las 4 condiciones requeridas:

int turno = 1;

Process SC1::	**Process SC2::**

{ while (true)
{ while (turno == 2) skip;
SC;
turno = 2;
SNC;
}
}
	{ while (true)
	{ while (turno == 1) skip;
	SC;
	turno = 1;
	SNC;
	}
	}

1. Exclusión mutua: se cumple. Por cómo está planteado el ejercicio solo uno de los procesos puede estar en su sección crítica a la vez
2. Ausencia de deadlock: se cumple, nunca se va a dar que turno sea 1 y 2 a la vez por lo que siempre uno podrá entrar
3. Ausencia de demora innecesaria: no se da, si un proceso quiere entrar a ejecutar, cuando sea su turno va a poder hacerlo.
4. Eventual entrada: formalmente no se cumple porque puede ser que el proce siempre le asigne prioridad a uno de los procesos y el otro no se puede ejecutar.

7. Desarrolle una solución de grano fino usando sólo variables compartidas (no se puede usar las sentencias *await* ni funciones especiales como *TS* o *FA)*. En base a lo visto en la clase 3 de teoría, resuelva el problema de acceso a sección crítica usando un proceso coordinador. En este caso, cuando un proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le dé permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. ***Nota:*** puede basarse en la solución para implementar barreras con "Flags y coordinador" vista en la teoría 3.

```
int arribo[1:n]=([n]0), ejecutar[1:n]=([n]0)
process coordinador[]{
  while (true) {
  for [i=1..n]{
    while (arribo[i]==0) skip{//CONSULTAR, no sería un if???? sino es re estúpido porque se queda esperando a que sea 1, sin avanzar al siguiente número del for
      ejecutar[i]=1
      while (ejecutar[i]==1) skip{
        arribo[i]=0
      }
    }
  }
  }
}
process proceso[id:0..n]{
  while (true) {
    arribo[i]=1
    while (ejecutar[i]==0) skip{
      Sección crítica
      ejecutar[i]=0
    }
  }
}
```

★ Igna usó un bakery, preguntaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaar