Programación Concurrente - Resolución práctica 1

Tomatis, Vicens, Torchia, Olmos, Manzin, Petraccaro, Traberg, Villareal

Bibliografía (1)

(1)

Consigna: 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:

- ☑ En algún caso el valor de x al terminar el programa es 56.
- ☑ En algún caso el valor de x al terminar el programa es 22.
- ☑ 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;

Respuesta:

a) P1 completo -> P2 completo -> P3 completo

b) x:= x*3 -> P1 Completo -> P3 (x*2)+1 -> P2 Completo

c) x:= x*3 -> P1 Completo -> P2 completo -> P3 (x*2)+1

---

(2)

Consigna: 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.
Respuesta:

Ejercicio 2

```
int total=0;
int arr[M];
Proccess Arrays [id:0..P-1]::
{
  suma=0
  parte= M/P
  extra=0
  inicio= id*parte
  if id= M-1 then extra= M mod P
  for (i=inicio, inicio+parte+extra, i++){
    if (arr[i]=N) then suma++
  }
  if suma!=0{
  <total+=suma>}
}
```

---

(3)

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::
{ while (true)
{ *produce 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_ocupada];
	pri_ocupada = (pri_ocupada + 1) mod N;
	consume elemento
	}
	}

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

Respuesta:

a)

Compartido

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

Productor

```
Process Productor::
{ while (true)
 { produce elemento
 <await (cant < N); cant++;
 buffer[pri_vacia] = elemento;>
 pri_vacia =(pri_vacia+1) mod N;
 }
}
```

Consumidor

```
Process Consumidor::
{ while (true)
 { <await (cant > 0); cant--;
 elemento = buffer[pri_ocupada];>
 pri_ocupada =(pri_ocupada+1) mod N;
 consume elemento
 }
}
```

b)

Compartido

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

Productor

```
Process Productor [id:0..P-1]::
{ while (true)
 { produce elemento
 <await (cant < N); cant++
 buffer[pri_vacia] = elemento;
 pri_vacia =(pri_vacia+1) mod N;>
 }
}
```

Consumidor

```
Process Consumidor [id:0..C-1]::
{ while (true)
 { <await (cant > 0); cant--
 elemento = buffer[pri_ocupada];
 pri_ocupada =(pri_ocupada+1) mod N;>
 consume elemento
 }
}
```

---

(4)

Consigna:

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.

Respuesta:

Ejercicio 4

```
colaEspecial C;
Process Consumidor[id:0..N-1]
{
    T rec;
    while(true)
        {
        {
            <await !empty(C); rec=Sacar(C) >
            usar recurso
            <Agregar(C,rec)>
        }
}
```

---

(5)

Consigna:

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.

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

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).

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.

Respuesta:

Ejercicio 5a

```
bool libre=true;
Process Persona[0..P-1]::
{
    <Imprimir(documento);>
}
```

Ejercicio 5b

```
cola C;
int sig=-1;
Process Persona[id:0..P-1]::
{
    <if (sig=-1) sig=id
    else Agregar(C,id)>;
    <await (sig==id)>;
    Imprimir(documento)
    <if(empty(C) sig=-1;
    else sig=Sacar(C)>;
}
```

Ejercicio 5c

```
colaEspecial C;
int sig=-1;
Process Persona[id:0..P-1]::
{
    <if (sig=-1) sig=id
    else AgregarOrdenado(C,id,id)>;
    <await (sig==id)>;
    Imprimir(documento)
    <if(empty(C) sig=-1;
    else sig=Sacar(C)>;
}
```

Ejercicio 5d

```
libre=true;
int siguiente=-1
```

Personas

```
Process Persona[1..N]::
{
    <Agregar(C,id);>
    <await (siguiente==id)>;
    Imprimir(documento);
    libre=true
}
```

Coordinador

```
Process Coordinador::
{
    int sig
    while(true)
    {
        <await !empty(C); sig=Sacar(C)>
        <await libre; libre=false>
        siguiente=sig
    }
}
```

(6)

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::
{ while (true)
{ while (turno == 2) skip;
SC;
turno = 2;
SNC;
}
}
	Process SC2::
	{ while (true)
	{ while (turno == 1) skip;
	SC;
	turno = 1;
	SNC;
	}
	}

☑ **Exclusión mutua:** A lo sumo un proceso está en su SC

☑ **Ausencia de Deadlock (Livelock):** si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.

☒ **Ausencia de Demora Innecesaria:** si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.

☑ **Eventual Entrada:** un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

En este caso, el while (turno==otro) skip evita que ambos procesos se encuentren en su SC a la vez.
Como solo hay 1 recurso para pedir a la vez, no puede haber deadlock
Hay demora innecesaria, cuando un proceso sale de su SC, inmediatamente el otro puede entrar a la suya, pero nunca puede entrar de forma no intercalada
No hay nada que impida que los procesos entren a su SC

Esta respuesta seguro puede ser mejorada. Preguntar

---

(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.

Ejercicio 7 - Area compartida

```
bool libre= true; int arribo[1:n]]=([n] false)
```

Procesos

```
Process Proceso[id:1..N]::
{
    while (true)
    {
        arribo[id] = true
        while (arribo[id]) { skip };
        SC
        libre=true;
        SNC
    }
}
```

Coordinador

```
Process Coordinador::
{
    while(true)
    {
        for [i=1 to n st arribo[i]==true]{
            while (not libre) skip;
            libre=false;
            arribo[i] = false;
        }
    }
}
```

Ejercicio 7 - Area compartida - Otra solucion

```
int next=-1; int arribo[1:n]=([n] false)
```

Procesos

```
Process Proceso[id:1..N]::
{
    while (true)
    {
        arribo[id] = true
        while (id != next) { skip };
        SC
        next = -1;
        SNC
    }
}
```

Coordinador

```
Process Coordinador::
{
    while(true)
    {
        for [i=1 to n st arribo[i]==true]{
            while (next!=-1) skip;
            arribo[i] = false;
            next = i;
        }
    }
}
```

Bibliografía

[1] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*. 1900 E Lake Ave Glenview, IL 60025 United States: Addison-Wesley, 1999.