Práctica 1 - Variables Compartidas.

# 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.

* Verdadero.
* Si el programa concurrente se ejecuta en el orden:
  + P1 (x = 8+2 = 10)
  + P2 (x = 11)
  + P3 (x = 33 + 22 + 1 = 56).

b) En algún caso el valor de x al terminar el programa es 22.

* Verdadero.
* Si el programa concurrente se ejecuta en el orden:
  + P1.1
  + P1.2
  + P3 (solo se ejecuta hasta x\*3 = 0)
  + P1.3 (x = 8+2 = 10)
  + P3 (lo que le resta, x = 0 + 20 + 1 = 21)
  + P2 (x = 22)

c) En algún caso el valor de x al terminar el programa es 23.

* Verdadero.
* Si el programa concurrente se ejecuta en el orden:
  + P3 (solo se ejecuta hasta x\*3 = 0)
  + P1 (x = 8+2 = 10)
  + P2 (x = 11)
  + P3 (x = 0 + 22 + 1 = 23)

| **P1::**  If (x = 0) then  y := 4\*2;  x := y + 2; | **P2::**  If (x > 0) then  x := x + 1; | **P3::**  x := (x\*3) + (x\*2) + 1 |
| --- | --- | --- |

# 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.

* Precondición: M es un número divisible por 2. Tengo dos procesos.
* Implementación:

M := numero\_par

N := un\_numero

int arreglo[M];

total := 0

Process P1 {

for [i=1 to (M/2)]

if arreglo[i] = N then

<total := total + 1>;

}

Process P2 {

for [i=(M/2)+1 to M]

if arreglo[i] = N then

<total := total + 1>;

}

# 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::** {  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*  }  } |
| --- | --- |

No, el código no funciona. Supongamos que estamos al principio de la ejecución del programa concurrente (cant = 0):

1. Se ejecuta el proceso Consumidor, el cuál se queda esperando porque cant no es mayor que 0.
2. Se ejecuta el proceso Productor, entra en el await y como cant es menor que N (supongamos que N sea 10), incrementa cant (cant = 1)
3. En este punto, se produce un Context Switch y como cant es mayor que 0, Consumidor decrementa cant y además lee del buffer (el cuál solo posee basura en ese momento).

Una solución sería:

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.

int cant = 0; int pri\_ocupada = 0; int pri\_vacia = 0; int buffer[N];

| **Process Productor** [i=1 to P]{  while (true) {  *produce elemento*  <await (cant < N); cant++  buffer[pri\_vacia] = *elemento*;  pri\_vacia = (pri\_vacia + 1) mod N;>  }  } | **Process Consumidor** [i=1 to P]{  while (true) {  <await (cant > 0); cant–;  *elemento* = buffer[pri\_ocupada];  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.

SOLUCIÓN 1:

int cola\_recurso[5];

int recursos\_disponibles = 5;

Process proceso [i=1 to P] {

while True {

<await (recursos\_disponibles > 0)

recurso = cola\_recursos.dequeue()

recursos\_disponibles-->

*Procesar Recurso*

<cola\_recursos.queueup(recurso); recursos\_disponibles++>

}

}

# 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 ítem. 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.

SOLUCIÓN 1:

Process Persona[i=1 to N] {

<Imprimir(documento)>

}

SOLUCIÓN 2:

bool impresora\_libre = true;

Process Persona[i=1 to N] {

<await (impresora\_libre); impresora\_libre = false>

Imprimir(documento)

impresora\_libre = true

}

*Las dos están bien :)*

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

int ticket\_persona = 1

int uso\_de\_impresora = 1

Process Persona[i=1 to N] {

<mi\_ticket = ticket\_persona; ticket\_persona++>

<await (mi\_ticket == uso\_de\_impresora)>

Imprimir(documento)

uso\_de\_impresora++

}

## 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).

int id\_min = N+1

bool impresora\_libre = true

Process Persona[i=1 to N] {

bool no\_use = true

while no\_use {

<if (i < id\_min) then; id\_min = i;>

<if ((i == id\_min) and (impresora\_libre)) then {

impresora\_libre = false>

Imprimir(documento)

id\_min = N+1

no\_use = false

impresora\_libre = true

}

}

}

*No está bien. Porque los procesos se siguen ejecutando cíclicamente (ineficiente).*

int cola\_procesos[n];

bool impresora\_libre = true

Process Persona[i=1 to n] {

*Agregar*(cola\_procesos, i)

<await ((impresora\_libre) and (*OrdenarYObtener*(cola\_procesos) == i); impresora\_libre = false>

Imprimir(documento)

aux = Sacar(cola\_procesos)

impresora\_libre = true

}

## 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.

int ticket\_persona = 1

int uso\_de\_impresora = -1

bool impresora\_libre

Process Persona[i=1 to N] {

<mi\_ticket = ticket\_persona; ticket\_persona++>

<await (mi\_ticket == uso\_de\_impresora)>

Imprimir(documento)

impresora\_libre = true

}

Process Coordinador {

for [i=1 to N] {

impresora\_libre = false

uso\_de\_impresora = i

<await (impresora\_libre)>

}

}

# 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;  SCN;  }  } |
| --- | --- |

* **Exclusión mutua:** Se cumple, ya que solo un proceso a la vez accede a SC.
* **Ausencia de Deadlocks:** Se cumple. Cuando los procesos tratan de acceder a SC, solo uno de ellos lo logra.
* **Ausencia de Demora Innecesaria:** Se cumple, porque primero accede el proceso SC1 a su SC, después setea turno en 1, deja su SC y ahora proceso SC2 accede.
* **Eventual entrada:** Se cumple, ya que en algún momento cada proceso accede a su SC.

# 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), continuar [1:n] = ([n] 0), termino [1:n] = ([n] 0)

process SC[i=1 to n] {

arribo[i] = 1

while (continuar[i] == 0) skip;

SC

termino[i] = 1;

}

process coordinador {

for [i=1 to n] {

while (arribo[i] == 0) skip;

}

for [i=1 to n] {

continuar[i] = 1

while (termino[i] == 0) skip;

}

}

La solución está casi bien. El problema está en que los procesos SC no deberían esperar a que todos lleguen para que puedan acceder a su SC.

int continuar[1:n] = ([n] 0), termino[1:n] = ([n] 0)

process SC[i=1 to n] {

while (continuar[i] == 0) skip;

SC

termino[i] = 1;

}

process coordinador {

for [i=1 to n] {

continuar[i] = 1

while (termino[i] == 0) skip;

}

}