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. **V**
b) En algún caso el valor de x al terminar el programa es 22. **V**
c) En algún caso el valor de x al terminar el programa es 23. **V**

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;

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 n;
int m;
int [0..m] arreglo;
int apariciones = 0;
Process Arreglo [id: 1..m]::
{
   if (arreglo[id]==n) then
      <apariciones = apariciones + 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*
	}
	}

Puede suceder que cuando el productor aumente la cantidad el consumidor intente asginar un elemento del buffer que aún se encuentre vacio porque el productor no realizo previamente la asignación del elemento al buffer:

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];   int P;   int C;
```

Process Productor[id: 1..P]::	**Process Consumidor[id: 1..C]::**

{ 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*
	}
	}

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.

```
ColaRecursos[5] c; enUso=0;
Process Recurso[id: 1..N]::{
   while(true){
    <await(enUso<5);enUso++;
    instanciaRecurso = c.pop>
    //Recurso en uso
    <c.push(instanciaRecurso);
    enUso—;>
   }
}
```

5. En cada ítem debe realizar una solución concurrente de grano grueso (utilizando <> y/o) 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*.

```
Process Persona [id: 1..n]::{
   Documento documento;
   <Imprimir(documento);>
   //Persona se retira;
}
```

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

```
Cola[n] c; int siguiente = 0;
Process Persona [id: 1..n]::{
   Documento documento;
   <if  (siguiente = 0) then siguiente = id;
   else Agregar(c,id)>;
   <await (siguiente==id)>;
   Imprimir(documento);
   <if empty(c) then siguiente = 0
   else siguiente=c.pop>
```

```
   //Persona se retira
}
```

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 actual=1;
Process Persona [id: 1..n]::{
   <await actual==id>;
   Imprimir(documento);
   <actual++;>
   //Persona se retira
}
```

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.

```
ok=false; Persona[n] p; int contador = 1; int actual = 1;
Process Persona [id: 1..n]::{
   Documento documento;
   <p[id]=contador;
   contador++>;
   <await p[id]==actual>;
   Imprimir(documento);
   ok=true
   //Persona se retira
}

Process Coordinador::{
   for i = 1 to n do begin
      actual = i;
      <await ok>
      ok = false;
}
```

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;
	  }
	}

Exclusión mutua: Se respeta ya que en la cláusula del while que contiene el turno se verifica que el número de turno sea el que le corresponde.

Ausencia de Deadlock (Livelock): Hay ausencia de deadlock ya que ambos procesos se manejan por turnos por ende no hay ningún caso en que un proceso no pueda entrar a su SC.

Ausencia de demora innecesaria: Esta propiedad también se cumple ya que cuando uno de los procesos cumple con su SC el mismo cambia su turno y permite al otro proceso entrar a su SC.

Eventual entrada: Ambos procesos pueden entrar a su SC, en el momento que sea su turno lo hará.

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 llegada[1..n] = ([n]0), continuar [1..n] = ([n]0)	

Process trabajador[id= 1..n]::{
while(true){
llegada[id]=1;
while (continuar[id]==0) skip;
//SC
continuar[id]=0;
}
}
	Process coordinador[id= 1..n]::{
	while (true){
	for i=1 to n{
	while(llegada[id]==0) skip;
	llegada[id]=0;
	}
	for i=1 to n{
	continuar [id]=1;
	}
	}
	}