Explicación Práctica de Variables Compartidas

Ejercicios

EJERCICIO 1

Suponiendo el siguiente código para un programa concurrente con dos procesos (***A*** y ***B***), debemos analizar con que valores puede terminar una variable compartida ***x***.

```
int x = 1;

Process A
{ int y = 5
  x = x + y;
 }


Process B
{ int z = 3
  x = x + z;
 }
```

Se descomponen los procesos en acciones atómicas de grano fino

1- Load Pos Memoria x, Reg Acumulador
2- Add Pos Memoria y, Reg Acumulador
3- Store Reg Acumulador, Pos Memoria x

4- Load Pos Memoria x, Reg Acumulador
5- Add Pos Memoria z, Reg Acumulador
6- Store Reg Acumulador, Pos Memoria x

EJERCICIO 1

Se deben analizar las diferentes historias que se pueden generar y ver los posibles resultados de ***x***

```
int x = 1;

Process A
{ int y = 5

  1- Load Pos Memoria x, Reg Acumulador
  2- Add Pos Memoria y, Reg Acumulador
  3- Store Reg Acumulador, Pos Memoria x
}


Process B
{ int z = 3

  4- Load Pos Memoria x, Reg Acumulador
  5- Add Pos Memoria z, Reg Acumulador
  6- Store Reg Acumulador, Pos Memoria x
}
```

Algunas historias posibles

- 1-2-3-4-5-6 → x = 9
- 4-5-6-1-2-3 → x = 9
- 1-2-4-5-3-6 → x = 4
- 1-4-2-5-3-6 → x = 4
- 1-4-2-5-6-3 → x = 6

Soluciones de grano grueso con sentencias await

Forma general del ***await***

< await (B); sentencias >

El proceso se demora en este punto hasta que la condición ***B*** es verdadera y en ese momento y en forma atómica ejecuta las ***sentencias***.
La atomicidad comienza desde el momento que se está haciendo el último chequeo de la condición ***B*** (el que la encuentra en verdadero) y NO cuando ya termino de chequearla. De tal forma que no hay posibilidad de que alguien modifique ***B*** en el lapso transcurrido entre que encontró la condición ***B*** verdadera y comienza a ejecutar ***sentencias***.

Por supuesto que un único proceso a la vez podrá estar ejecutando ***sentencias.***

Nos brinda sincronización por ***Exclusión Mutua*** y/o ***por Condición.***

Soluciones de grano grueso con sentencias await

Forma del ***await*** sólo para ***Exclusión Mutua***

< sentencias >

El proceso ejecuta ***sentencias*** en forma atómica. Sólo un proceso a la vez puede estar ejecutando esa "Sección Crítica" (SC).

Se debe tener en cuenta que si hay varios procesos esperando ejecutar la SC y esta se libera (el que estaba ejecutando termino) cualquiera de los que está esperando accederá a usarla con Exclusión Mutua, y NO de acuerdo al orden de llegada.

Forma del ***await*** sólo para ***Sincronización por Condición***

< await B;>

El proceso únicamente se demora hasta que ***B*** es verdadero.

Ejemplo 2

Se tiene un salón con cuatro puertas por donde entran los alumnos a un examen. Cada puerta lleva la cuenta de los que entraron por ella y a su vez se lleva la cuenta del total de personas en el salón.

Necesitamos una variable compartida para llevar el total de personas en el salón y un variable local para cada puerta que tenga la cantidad de personas que entraron por ella.

```
int Total = 0;

Process Puerta[id: 0..3]
  {  int Parcial = 0;

     while (true)
      {  esperar llegada
         Parcial = Parcial + 1;
         Total= Total+ 1;
      }
  }
```

Que ocurre con esta sentencia que no es atómica

Ejemplo 2

El incremento de ***Total*** no es atómico, ya que (como vimos en el ejemplo 1) se divide en 3 acciones atómicas de grano grueso. Podría pasar que más de una puerta intente ejecutar esa sentencia al mismo tiempo, por lo que podría solaparse algún incremento y esto generar que el resultado de Total sea menos a que corresponda (*Total* < suma de los 4 *Parciales*). Debemos asegurarnos de que el incremento se haga SI o SI de forma atómica → <Total = Total +1 >;

```
int Total = 0;

Process Puerta[id: 0..3]
 {  int Parcial = 0;

    while (true)
     {  esperar llegada
        Parcial = Parcial + 1;
        <Total= Total+ 1>;
     }
 }
```

Que ocurre con esta sentencia que tampoco es atómica

Ejemplo 2

En este caso, a pesar de que el incremento no sea una acción atómica, se comporta como tal, ya que cada proceso tiene su propia instancia local de ***Parcial*** (cada uno tiene su propia variable), por lo que las modificaciones realizadas por un proceso a su variable ***Parcial*** no afectan o interfieren sobre los otros → Por lo tanto no se debe usar <Parcial = Parcial +1>

```
int Total = 0;

Process Puerta[id: 0..3]
  {  int Parcial = 0;

     while (true)
      {  esperar llegada
         Parcial = Parcial + 1;
         <Total= Total+ 1>;
      }
  }
```

Siempre que se pueda se deben usar variables locales a cada proceso para evitar de este modo usar <> que reducen la concurrencia. Dejarlo sólo para cuando es necesario (en este caso *Total*).

Ejemplo 3

Hay un docente que les debe tomar examen oral a 30 alumnos (de a uno a la vez) de acuerdo al orden dado por el Identificador del proceso

Cada alumno debe esperar a que el docente lo llame, luego debe esperar a que el docente le avise que el examen termino y se va; mientras que el docente llama al siguiente alumno.

```
Process Alumno [id: 0..29]
 {  //Espera a que lo llamen
    //Rinde el examen
    //Espera a que termine el examen
 }


Process Docente
 {   for i = 0..29
       {  //Llama al alumno "i"
          //Toma el examen
          //Avisa a "i" que termino
       }
 }
```

Ejemplo 3

```
Process Alumno [id: 0..29]
 { //Espera a que lo llamen
   //Rinde el examen
   //Espera a que termine el examen
 }


Process Docente
 {  for i = 0..29
      { //Llama al alumno "i"
        //Toma el examen
        //Avisa a "i" que termino
      }
 }
```

Cómo se resuelve la espera

Debemos usar una variable compartida ***Actual*** que indique cual es el alumno que debe pasar a rendir el examen, y cada alumno debe demorarse hasta que ese valor sea igual a su ***id***.

```
int Actual = -1;

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   //Espera a que termine el examen
 }
```

Ejemplo 3

```
int Actual = -1

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   //Espera a que termine el examen
 }


Process Docente
 {  for i = 0..29
      { //Llama al alumno "i"
        //Toma el examen
        //Avisa a "i" que termino
      }
 }
```

Cómo llama al alumno?

Llama a un determinado alumno modificando el valor de ***Actual*** con el identificador del alumno a atender.

```
Process Docente
 {  for i = 0..29
      { <Actual = i>
        //Toma el examen
        //Avisa a "i" que termino
      }
 }
```

Es necesario hacerlo con <>

Ejemplo 3

El único proceso que puede modificar el valor de ***Actual*** es el ***Docente***, y hasta que el Alumno correspondiente no vea ese valor modificado y termine el examen no lo va a poder volver a modificar, por lo que NO hay riesgo de que el alumno no vea que es su turno de rendir → Por lo tanto no se debe usar <Actual = 1>

```
int Actual = -1

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   //Espera a que termine el examen
 }

Process Docente
 {  for i = 0..29
      { Actual = i
        //Toma el examen
        //Avisa a "i" que termino
      }
 }
```

Debemos sincronizar la finalización del examen para que el alumno no se vaya antes de terminar y el docente no llame a otro alumno al mismo tiempo.

Usamos una variable compartida ***Listo*** para esta sincronización.

Ejemplo 3

```
int Actual = -1
bool Listo = False;

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   <await Listo>;
 }

Process Docente
 {  for i = 0..29
      { Actual = i
        //Toma el examen
        <Listo = true;>
      }
 }
```

En que momento se resetea a False

Lo debe resetear a ***false*** el alumno después de ver que estaba en true. Y el ***Docente*** debe esperar a que el alumno lo haya puesto en false antes de llamar al siguiente alumno (sino este otro alumno inmediatamente que pasa a rendir el examen ve que la variable ***Listo*** es ***true*** y se retira sin rendir.

Ejemplo 3

```
int Actual = -1
bool Listo = False;

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   <await (Listo); Listo = false>;
 }

Process Docente
 {  for i = 0..29
      { Actual = i
        //Toma el examen
        <Listo = true;>
        <await (not Listo)>
      }
 }
```

Es necesario hacerlo con <>

Es necesario hacerlo con <>

No es necesario el <> por las mismas razones que Actual = i

```
int Actual = -1
bool Listo = False;

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   <await (Listo)>;
   Listo = false;
 }

Process Docente
 {  for i = 0..29
      { Actual = i
        //Toma el examen
        Listo = true;
        <await (not Listo)>;
      }
 }
```

Ejemplo 3

```
int Actual = -1
bool Listo = False;

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   //Rinde el examen
   <await (Listo)>;
   Listo = false;
 }

Process Docente
 {  for i = 0..29
      { Actual = i
        //Toma el examen
        Listo = true;
        <await (not Listo)>;
      }
 }
```

Que ocurre si el alumno correspondiente aún no llegó?

El alumno debe "avisar" que llegó.

```
int Actual = -1;  bool Listo = False, Ok = false;

Process Alumno [id: 0..29]
 { <await (Actual == id)>;
   Ok = true;
   //Rinde el examen
   <await (Listo)>;
   Listo = false;
 }

Process Docente
 {  for i = 0..29
      { Actual = i
        <await (Ok)>; Ok = false;
        //Toma el examen
        Listo = true;
        <await (not Listo)>;
      }
 }
```

Ejemplo 4

Un cajero automático debe ser usado por ***N personas*** de a uno a la vez y según el orden de llegada al mismo. En caso de que llegue una persona anciana, la deben dejar ubicarse al principio de la cola.

Cuando una persona llega al cajero, si esté está ocupado y/o hay gente esperando en la cola, debe demorarse hasta que sea su turno: si es una persona mayor debe ubicarse al frente de la cola, y sino al final.
Cuando es su turno, debe usar el cajero y luego, al terminar debe avisarle al siguiente que pase (en caso de que haya alguien en la cola).

```
Process Persona [id: 0..N-1]
 { //Si el cajero no está libre debe encolarse
   //Usa el cajero
   //Libera el cajero
 }
```

Vamos a suponer que existe una estructura de datos "colaEspecial" con una función "Agregar(edad, id)" que dependiendo de ese valor lo inserta la tupla al principio o al final de la cola; y existe una función "Sacar" que devuelve el ***id*** de quien está al principio en la cola.

Ejemplo 4

```
colaEspecial C;

Process Persona [id: 0..N-1]
 { int edad = inicializac
    Agregar(C, edad, id)
     //Esperar turno
    //Usa el cajero
    //Libera el cajero
 }
```

Es necesario hacerlo con <>

SI, sino se puede generar una interferencia entre dos personas que se quieren encolar, ocupando el mismo lugar y, de ese modo, una reemplaza a la otra.

¿Como realiza la espera?

```
colaEspecial C;

Process Persona [id: 0..N-1]
 { int edad = ….;
    <Agregar(C, edad, id)>
    //Esperar turno
     //Usa el cajero
    //Libera el cajero
 }
```

Ejemplo 4

Se debe usar como en el ejemplo anterior una variable que indique quien es el siguiente que debe usar el cajero.

```
colaEspecial C;
int Siguiente = -1;

Process Persona [id: 0..N-1]
 {  int edad = ….;

    <Agregar(C, edad, id)>;
    <await (Siguiente == id)>;
    //Usa el cajero
    //Libera el cajero
 }
```

¿Como realiza la liberación?

Debe sacar al primero de la cola y poner su ***id*** en ***Siguiente***, pasándole el turno de usar el cajero a esa persona.

```
colaEspecial C;
int Siguiente = -1;

Process Persona [id: 0..N-1]
 {  int edad = ….;

    <Agregar(C, edad, id)>;
    <await (Siguiente == id)>;
    //Usa el cajero
    <Siguiente = Sacar(C)>;
 }
```

Es necesario hacerlo con <>

SI, porque la función ***Sacar*** podría interferir con la función ***Agregar***

Ejemplo 4

```
colaEspecial C;
int Siguiente = -1;

Process Persona [id: 0..N-1]
 {  int edad = ….;

    <Agregar(C, edad, id)>;
    <await (Siguiente == id)>;
    //Usa el cajero
    <Siguiente = Sacar(C)>;
 }
```

A quien despierta si en ese momento no hay nadie en la cola

En ese caso no debe despertar a nadie en particular (no sabría a quien) ni tampoco debe esperar a que llegue alguien al cajero para pasarle el turno y recién ahí retirarse. Debe indicar que el cajero está libre e irse.

Ponemos un dato inválido en Siguiente indicando que está libre. Por ejemplo -1.

Ejemplo 4

```
colaEspecial C;
int Siguiente = -1;

Process Persona [id: 0..N-1]
  {  int edad = ….;

     <Agregar(C, edad, id)>;
     <await (Siguiente == id)>;
     //Usa el cajero
     <if (empty(C))  Siguiente = -1
       else Siguiente = Sacar(C)>;
  }
```

Si cuando la persona llega el cajero está libre (Siguiente = -1) la persona se encolará y nunca nadie la despertará, ya que todas las personas que lleguen harán lo mismo y nadie pasará a usar el cajero y luego a despertar a una persona encolada.
Primero debe verificar si el cajero está libre y en ese caso "auto" asignarse el turno, y sino recién ahí encolarse.

Ejemplo 4

```
colaEspecial C;
int Siguiente = -1;

Process Persona [id: 0..N-1]
 {  int edad = ….;

    <if (Siguiente = -1) Siguiente = id
      else Agregar(C, edad, id)>;
    <await (Siguiente == id)>;
    //Usa el cajero
    <if (empty(C)) Siguiente = -1
      else Siguiente = Sacar(C)>;
 }
```

Problema de la Sección Crítica (SC)

Implementación de acciones atómicas en software (locks).

Propiedades que debe cumplir la solución:

- 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).

Problema de la Sección Crítica (SC) – Solución "*Spin Locks*"

Solución tipo ***"spin locks"***: los procesos se quedan iterando (spinning) mientras esperan que se limpie ***lock***.

```
bool lock=false;

process SC[i=1 to n]
{  while (true)
     {   <await (not lock); lock = true>;
         sección crítica;
         lock = false;
         sección no crítica;
     }
}
```

```
bool lock=false;

process SC[i=1 to n]
{  while (true)
     {   while (TS(lock)) skip ;
         sección crítica;
         lock = false;
         sección no crítica;
     }
}
```

Problema de la Sección Crítica (SC) – Solución Fair - "*Tie-Breaker*"

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
   while (true) {    in1 = true; ultimo = 1;
                     while (in2 and ultimo == 1) skip;
                     sección crítica;
                     in1 = false;
                     sección no crítica;
   }
}

process SC2 {
   while (true) {    in2 = true; ultimo = 2;
                     while (in1 and ultimo == 2) skip;
                     sección crítica;
                     in2 = false;
                     sección no crítica;
   }
}
```

Problema de la Sección Crítica (SC) – Solución Fair - "*Ticket*"

Se reparten números y se espera a que sea el turno → *Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los procesos con número más chico han sido atendidos.*

```
int numero = 1;
int proximo = 1;

process SC [i: 1..n]
{ int turno;
  while (true)
    {   <turno = numero; numero ++>;
        <await (turno ==proximo)>;
        sección crítica;
        proximo = proximo++;
        sección no crítica;
    }
}
```

```
int numero = 1;
int proximo = 1;

process SC [i: 1..n]
{ int turno;
  while (true)
    {   turno = FA (numero, 1);
        while (turno <> proximo) skip;
        sección crítica;
        proximo = proximo++;
        sección no crítica;
    }
}
```

Problema de la Sección Crítica (SC) – Solución Fair - "*Bakery*"

Los procesos se chequean entre ellos y no contra un global → *Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna uno mayor. Luego espera a que su número sea el menor de los que esperan.*

```
int turno[1:n] = ([n] 0);

process SC[i = 1 to n]
{   while (true)
        {  turno[i] = 1;   //indica que comenzó el protocolo de entrada
           turno[i] = max(turno[1:n]) + 1;
           for [j = 1 to n st j != i]        //espera su turno
                 while (turno[j] != 0)  and (  (turno[i],i) > (turno[j],j) ) →  skip;
            sección crítica
            turno[i] = 0;
            sección no crítica
        }
}
```

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ (∀j : 1 ≤ j ≤ n, j ≠ i: turno[j] = 0 ∨ turno[i] < turno[j])) }

Sincronización *Barrier* "*Contador Compartido*"

Se utiliza un contador para llevar la cuenta de cuantos han llegado a la barrera, y los procesos esperan hasta que el valor es *n*.

```
int cantidad = 0;

process Worker[i=1 to n]
{ …..
   <cantidad = cantidad +1>;
   <await (cantidad = n)>;
   …...
}
```

```
int cantidad = 0;

process Worker[i=1 to n]
{ …..
   FA (cantidad, 1);
   while (cantidad <> n) skip;
   …...
}
```

Sincronización *Barrier*
"*Barrera con Coordinador*"

Para múltiples barreras se puede incorporar un "coordinador" evitando el uso de un contador compartido.

```
int arribo[1:n] = ([n] 0);
int continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{  ....
   arribo[i] = 1;
   <await (continuar[i] == 1) >;
   continuar[i] = 0;
    .....
}

process Coordinador
{  for [i = 1 to n]
       { <await (arribo[i] == 1)>;
         arribo[i] = 0;
       }
   for [i = 1 to n] continuar[i] = 1;
}
```

Sincronización *Barrier* "*Flags y Coordinadores*"

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{  ….
   arribo[i] = 1;
   while (continuar[i] == 0) skip;
   continuar[i] = 0;
   …..
}

process Coordinador
{  for [i = 1 to n]
       { while (arribo[i] == 0)  skip;
         arribo[i] = 0;
       }
   for [i = 1 to n] continuar[i] = 1;
}
```