

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 \longrightarrow $x = 9$
- 4-5-6-1-2-3 \longrightarrow $x = 9$
- 1-2-4-5-3-6 \longrightarrow $x = 4$
- 1-4-2-5-3-6 \longrightarrow $x = 4$
- 1-4-2-5-6-3 \longrightarrow $x = 6$

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 ($Total < \text{suma de los 4 Parciales}$). Debemos asegurarnos de que el incremento se haga SI o SI de forma atómica \longrightarrow `<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 \longrightarrow Por lo tanto no se debe usar $\langle \text{Parcial} = \text{Parcial} + 1 \rangle$

```
int Total = 0;

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

  while (true)
  { esperar llegada
    Parcial = Parcial + 1;
     $\langle \text{Total} = \text{Total} + 1 \rangle$ ;
  }
}
```

Siempre que se pueda se deben usar variables locales a cada proceso para evitar de este modo usar $\langle \rangle$ 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 estricto orden dado por el Identificador del proceso (hasta no haber atendido al 0 no puede atender al 1...si aún no llegó debe esperar)

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
}
```

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**.

Process Docente

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

```
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 $\langle \text{Actual} = 1 \rangle$

```
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>;
}
```

Es necesario
hacerlo con <>

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

Es necesario
hacerlo con <>

```
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)>;
  }
}
```

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

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
correspondient
e aún no 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)>;
  }
}
```

El alumno debe “avisar” que llegó.



Ejemplo 4

Variante del ejercicio anterior. El docente toma el examen de acuerdo al orden dado por el identificador del proceso (entre los que ya han llegado).

Partimos de la solución anterior

¿Cómo sabe a
quien llamar?

```
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

Debería haber una estructura compartida donde estén almacenados los ID de los alumnos que han llegado, con el orden requerido.

Usaremos una cola ordenado “*espera*” de menor a mayor

¿Y si la cola está vacía?

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

```
Process Alumno [id: 0..29]
```

```
{ agregar (Espera, id);  
  <await (Actual == id)>;  
  Ok = true;  
  //Rinde el examen  
  <await (Listo)>;  
  Listo = false;  
}
```

```
Process Docente
```

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

Ejemplo 4

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

```
Process Alumno [id: 0..29]
```

```
{  agregar (Epera, id);  
  <await (Actual == id)>;  
  Ok = true;  
  //Rinde el examen  
  <await (Listo)>;  
  Listo = false;  
}
```

```
Process Docente
```

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

Debe esperar hasta que no esté vacía la cola para realizar el “sacar”.

Posible interferencia al actualizar ESPERA (tanto por Agregar como por Sacar)

Ejemplo 4

Agregar y Sacar se deben hacer con exclusión mutua.

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

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

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

Ejemplo 5

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 5

colaEspecial C;

Process Persona [id: 0..N-1]

{ int edad = *inicializaci*

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.

colaEspecial C;

Process Persona [id: 0..N-1]

{ int edad =;

<Agregar(C, edad, id)>

//Esperar turno

//Usa el cajero

//Libera el cajero

}

¿Como realiza la espera?

Ejemplo 5

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?

```
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 <>

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

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

Ejemplo 5

```
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 5

```
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 5

```
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)>;
```

```
}
```

Ejemplo 6

Modificamos el ejercicio anterior, donde sólo se debe respetar el orden de llegada.

Partimos de la solución anterior

Si la función **“Agregar”** en la cola C lo hace siguiendo sólo el orden de llegada se resuelve el problema.

Cambiamos Agregar y Sacar por Push y Pop.

```
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)>;  
}
```

Ejemplo 6 – versión 1

```
cola C;  
int Siguiente = -1;  
  
Process Persona [id: 0..N-1]  
{ int edad = ....;  
  
    <if (Siguiente = -1) Siguiente = id  
      else Push (C, id)>;  
    <await (Siguiente == id)>;  
    //Usa el cajero  
    <if (empty(C)) Siguiente = -1  
      else Siguiente = Pop (C)>;  
}
```

Ejemplo 6 – versión 2

También se puede utilizar una solución basada en el algoritmo “*Ticket*”, esta versión sólo sirve para respetar el orden de llegada (no sirve para otro orden) y evitamos el uso de la cola.

```
int Numero = 0;  
int Siguiente = -1;
```

```
Process Persona [id: 0..N-1]
```

```
{ int turno;
```

```
    <turno = Numero; Numero++ >;
```

```
    <await (Siguiente == turno)>;
```

```
    //Usa el cajero
```

```
    <Siguiente++>;
```

```
}
```

Se produce DEADLOCK
ya que Siguiente vale -1 y el
primer turno es 0, por lo
que nadie avanzará.

Ejemplo 6

```
int Numero = 0;
int Siguiente = 0;

Process Persona [id: 0..N-1]
{ int turno;

    <turno = Numero; Numero++ >;
    <await (Siguiente == turno)>;
    //Usa el cajero
    <Siguiente++>;
}
```