

### Ejercicio 1:

**P1:**

**y = 4\*2**

(1.1) → Load 4, Reg acumulador

(1.2) → Mul 2, Reg acumulador

(1.3) → Store Acumulador , Y

**x = y + 2;**

(2.1) → Load Y, Reg Acum

(2.2) → Add 2, Acumulador

(2.3) → Store Acumulador X

**P2:**

**x = x + 1**

(3.1) → Load X, Reg Acum

(3.2) → Add 1, Reg Acum

(3.3) → Store Acumulador X

**P3:**

**x = (x\*3) + (x\*2) + 1**

(4.1) → Load X, Reg Acumulador

(4.2) → Mul 3, Reg Acumulador

store

add

(4.3) → Load X, reg acumulador

(4.4) → Mul 2, reg acumulador → add entre (x\*3) **+** (x\*2)

store

(4.5) → Add 1, reg acumulador

(4.6) → Store acumulador, X

P1-P2-P3: 56

P3-P1-P3-P2: 22

Arranca el proceso 3. Realiza la operación x\*3. Arranca el proceso 1, retorna el valor 10, en el proceso 3 hace x\*2 = 20 y le suma 1. Retorna la x = 21 . Finalmente va al proceso 2 y le suma 1 retornando la X = 22.

P3-P1-P2-P3: 23

Realiza la operación x\*3. Arranca el proceso 1, retorna el valor 10

Arranca el proceso 2, y le suma 1 a la X, con lo que X = 11. Sigue el proceso 3 y hace X\*2 = 22. X+1 = 23.

Historias posibles:

(4.1) (4.2)(1.1)(1.2) (1.3) (3.1)(3.2)(3.3) (4.3)(4.4) → x = 56

### Ejercicio 2:

2. Realice una solución concurrente de grano grueso (utilizando <> y/o <await B; S>) para el

siguiente problema. Dado un numero N verifique cuantas veces aparece ese número en un

arreglo de longitud M. Escriba las pre-condiciones que considere necesarias.

*Soluciones de grano grueso con sentencias* ***await*** *para evitar la interferencia. Nos brindan sincronización por Exclusión Mutua y/o por Condición.*

* Hay dos tipos de sentencias await:
  + Para **Exclusión Mutua: <sentencias>**

Se usa para que un proceso ejecute sentencias de forma atómica. En ese caso, solo un proceso a la vez puede estar ejecutando esa Sección Crítica(para variables compartidas). Los procesos que estaban esperando pueden acceder a la Sección crítica una vez que se libera en cualquier orden, sin importar el orden de llegada. Y también la ocupan con Exclusión Mutua.

* + Para **Sincronización por Condición: <await B;>**

Se usa para demorar un proceso hasta que la condición B es verdadera.

Entonces, para este punto, hay que resolver el problema y usar <> para exclusión mutua y

<await B; S> para sincronización por condición.

Pre condiciones: el arreglo es compartido pero solo vamos a leerlo. El contador es compartido así que hay que evitar interferencias.

int [1..M] numbers;

int cant = 0;

int M = algun valor;

Process compare(int N, int value){

if(N == value){

<cant ++;>

}

}

process calculate[i = 1 to M ]{

compare(N, numbers[i]);

}

Calculate es el conjunto de los M procesos independientes, que llaman al proceso compare, el cual identifica si hay una ocurrencia de N e incrementa el contador, evitando la interferencia en la sección crítica por medio del await de exclusión mutua. **Preguntar**

Ejemplo sacado de la teoría 2. con la cláusula process para invocar a M procesos independientes que deben modificar al contador por exclusión mutua.

Otra alternativa podría ser dividir al arreglo en partes, y que cada process calcule las ocurrencias de N en los fragmentos del arreglo, utilizando < > para cuando necesitan incrementar el contador.

Consultar cómo se podría hacer con < await B; S > **Preguntar → No tiene sentido para este ejercicio :)**

Sincronización por condición: por exclusión mutua, intenta chequear si la condición es verdadera, y si la es, procesa las sentencias, sino, abandona la exclusión mutua y espera a que se libere nuevamente.

< await B; > Sirve para esperar una condición, no se ejecuta nada en la SC por exclusión mutua.

(r = m div n), [for y = (i \* r) to ((i++ \* r)--) ] if (n == arr[y]) cantidad++ ; <cant = cant + cantidad>

<await (cant < N); buffer[pri\_vacia] = elemento >

<cant++>;

pri\_vacia = (pri\_vacia + 1) mod N;

Nota Clase martes ej2.

tengo P procesos y un problema de tamaño M, como hago para distribuir el problema de tam M entre los P procesos, cada proc se hace cargo de M/P elementos, cada proceso se encarga de una parte del proceso, for desde el inicio al final que le toca a cada parte, utilizando el id,

for X := id\*m/p hasta posición donde estoy\* parte + m/p → granularidad del proceso

veces:=0

numero:=N

Arreglo[0..M]

Process x[id = 0..P]

parte=M/P

veces \_local:=0

for x:= id\*parte hasta id\*parte + M/P (granularidad del proc){

if(numero==arreglo[X]){

veces local:=veces local+1

}

}

<veces:=veces + veces\_local>

}

### Ejercicio 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  }  } |
| --- | --- |

Deberia setear el elemento, luego sumar la cantidad, sino el consumidor puede chequear un elemento que aún no está

| Process Productor::  { while (true)  { produce elemento  **<await (cant < N); >**  buffer[pri\_vacia] = elemento;  **<cant++>**  pri\_vacia = (pri\_vacia + 1) mod N;  }  } | Process Consumidor::  { while (true)  { **<await (cant > 0); >**  elemento = buffer[pri\_ocupada];  **<cant-->**  pri\_ocupada = (pri\_ocupada + 1) mod N;  consume elemento  }  } |
| --- | --- |

b) Deberíamos tener en cuenta que P != C para sincronizarlos de alguna manera, o solamente hay que proteger los accesos a la SC? ¿Está bien?

| int pri\_vacia, pri\_ocupada, cant;  int buffer[N];  Process Productor[id=0..P]::  { while (true)  { //produce elemento  **<await (cant < N);**  buffer[pri\_vacia] = elemento;  **cant++;**  pri\_vacia = (pri\_vacia + 1) mod N;>  }  } | Process Consumidor[id=0..C]::  { while (true)  { **<await (cant > 0);**  elemento = buffer[pri\_ocupada];  **cant--;**  pri\_ocupada = (pri\_ocupada + 1) mod N;>  //consume elemento  }  } |
| --- | --- |

### Ejercicio 4:

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.

Hay una estructura, que tiene 5 instancias de un recurso. Los procesos desencolan, usan el recurso y vuelven a encolar.

int dato;

int cola\_de\_recursos[5]; estructura que hay que proteger.

<await(cola\_de\_recursos

<await B; S>

B→ mientras que no esté vacía(B);

S → desencolar,

usar recurso,

<encolar nuevamente>

**int dato;**

**int cola\_de\_recursos[5];**

**Process A::**

**{**

**<await cola\_de\_recursos(isnotempty);**

**dato = cola\_de\_recursos.desencolar();>**

**//hacer algo con el dato**

**<cola\_de\_recursos.encolar(dato);>**

**}**

### Ejercicio 5:

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 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.

a)

Documento documento;

Process personas[id = 1 ..N]::

{

<imprimir(documento);>

}

b)

int Actual =-1

Process Personas [id: 1..N]

{

<await (Actual == id)>;

//usa la impresora

imprimir(documento);

<Actual = Actual + 1>;

}

Hay que verificar el orden de llegada de los procesos.

Persona cola;

Actual = -1;

Process Personas[id: 1..N]::

{

< if (Actual == -1) //si esta libre

Actual = id //me toca a mi

else

cola.encolar(id);>//sino me encolo

<await (Actual == id)>; // voy yoooo

imprimir(documento); //uso el recurso

<if(cola.isEmpty())

Actual = -1 // si no hay nadie en la cola, dejo el rec como libre

else

Actual = cola.desencolar();> //desencola el sig

}

c)Si estuviese impresora ocupada demorarme en la cola de espera

solo si la impresora esta ocupada

me encolo

si es encolado espera hast que sea su turno

liberar una vez que deja de usarlo, despertar o avisar al de menor de la cola que puede pasar, si la cola vacía, liberar cartelito libre ponele, pero desocupado solo si no hay nadie, sino avisa al primero de la cola

int minimo = 9999;

int Actual = -1;

Persona cola;

bool Listo = false;

Process Personas [id = 1.. N]::

{

if(bool Listo= true)

<Listo = false>

imprimir(documento);

<Actual = minimo(cola);

Listo = true;>

else

cola.encolar(id)

<await (Actual == id)>;

}

**int minimo = 9999;**

**int Actual = -1;**

**Persona cola;**

**Process Personas [id = 1.. N]::**

**{**

**< if (Actual == -1) //si esta libre**

**Actual = id //me toca a mi**

**else**

**cola.encolar(id);>//sino me encolo**

**<await (Actual == id)>;**

**imprimir(documento);**

**<if(cola.isEmpty())**

**Actual = -1 // si no hay nadie en la cola, dejo el rec como libre**

**else**

**Actual = desencolar\_minimo(cola);> //**

**}**

d)coordinador

tiene que avisar al coordinador que terminó, pero como llama el coordinador a las personas?

puede quedar la persona esperándola q una variable tenga su id, y el coordinador solo cambia esa variable de forma atómica,

Persona cola;

Actual = -1;

bool Listo=fase;

Process Docente

{

while(true){

<if(not cola.isEmpty()) Actual = cola.desencolar();

Listo = true;> //avisa que le toca a ese id

<await (not Listo)>;

}

}

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

{

<cola.encolar(id);>

<await (Actual == id)>;

imprimir(documento);

<Listo=false;>

}

**¡Corrección!**

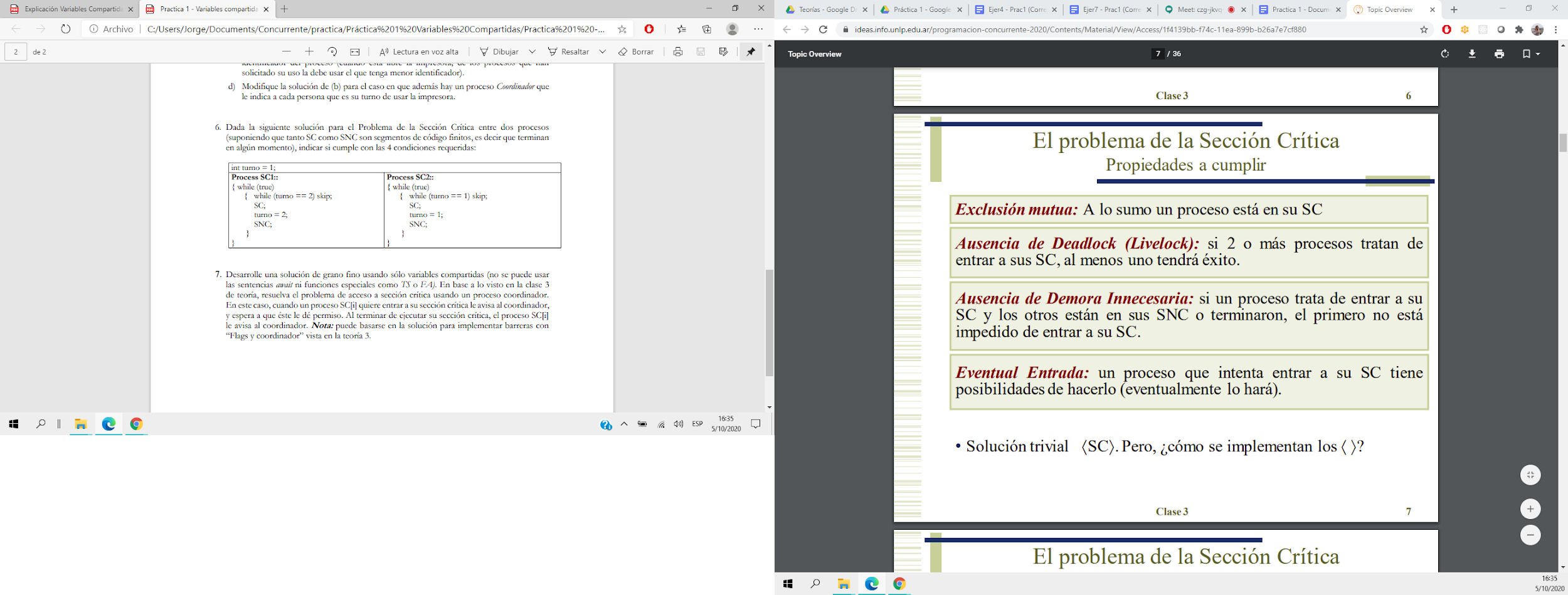
3) Si bien se estaría reduciendo la concurrencia, pondría todas las sentencias en el mismo await, ya que puede que un proceso esté accediendo al primer await y todavía no se actualizó el valor de pri\_vacía.

5a) Está bien

5b) Quién se encarga de desencolar a los procesos? como en este punto no existe un coordinador, son los mismos procesos quienes deben sincronizarse para darse acceso a la impresora. Cuando un proceso llega a ejecutarse lo primero que debe hacer es preguntar si hay gente esperando en la cola, si es así se debe encolar sino ya puede acceder a utilizar el recurso. Cuando termina de utilizarlo él le tiene que pasar "la posta" a otro proceso, por lo que consulta si hay gente esperando, si hay le da el acceso al proceso que está esperando y en caso de no existir procesos en la cola, deja libre el recurso para el primero que lo solicite.

Con esto último revisa el d, el mínimo te lo va a dar una función X.

### Ejercicio 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:*** El proceso SC2 cumple con la propiedad de Exclusión mutua. ¿SC1 se ejecuta?
* ***Ausencia de Deadlock:*** Ambos procesos cumplen, ya que no compiten por el mismo recurso. Y ninguno depende del otro para acceder a su SC.
* ***Ausencia de Demora Innecesaria:***  Cumple porque cada uno espera la condición que necesita y no se interfieren.
* ***Eventual Entrada:***  No cumple, el proceso SC2 cumple pero el SC1 no tendría oportunidad de ejecutarse.