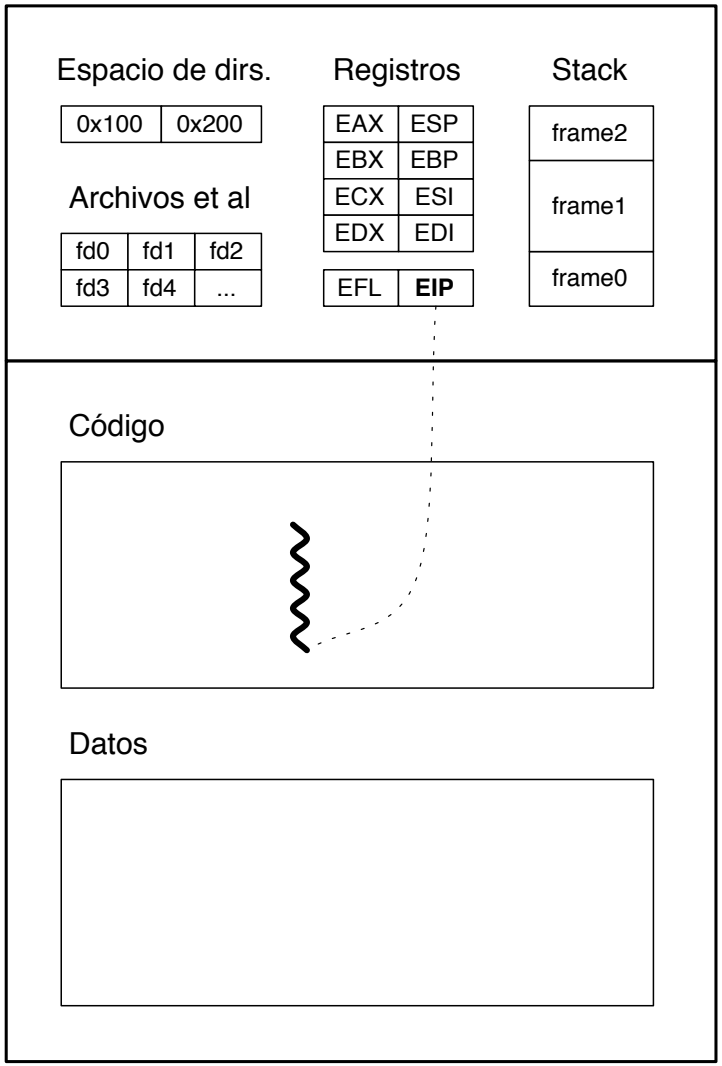
Sincronización entre Procesos

Sistemas Operativos
DC - UBA - FCEN

11 de septiembre de 2025

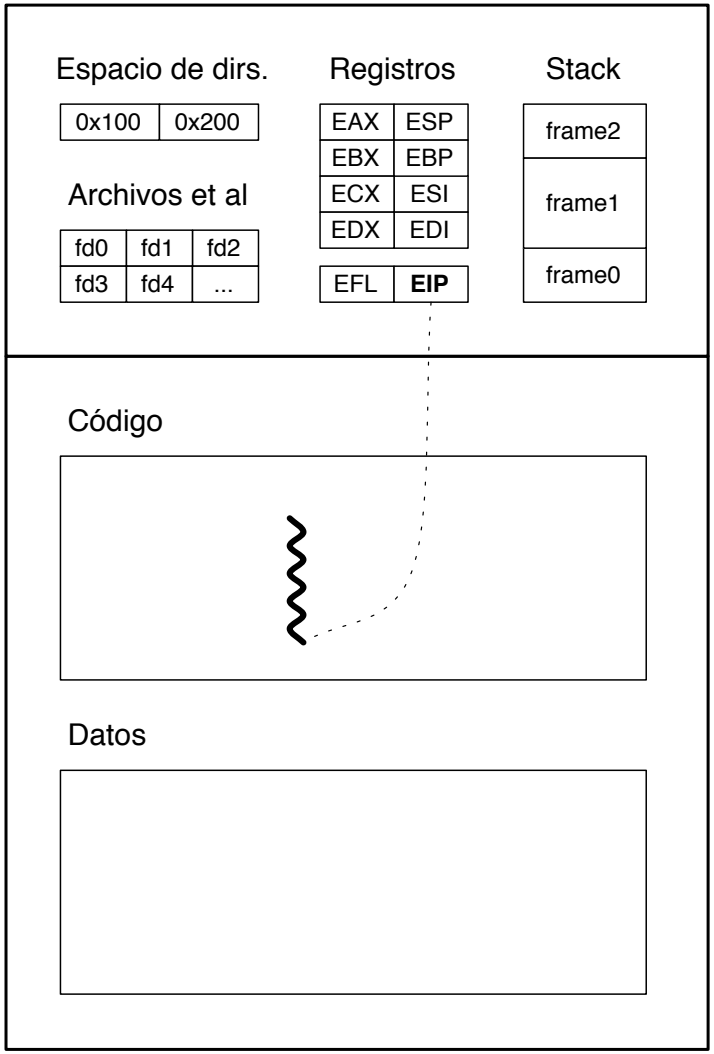
Estructura de la clase

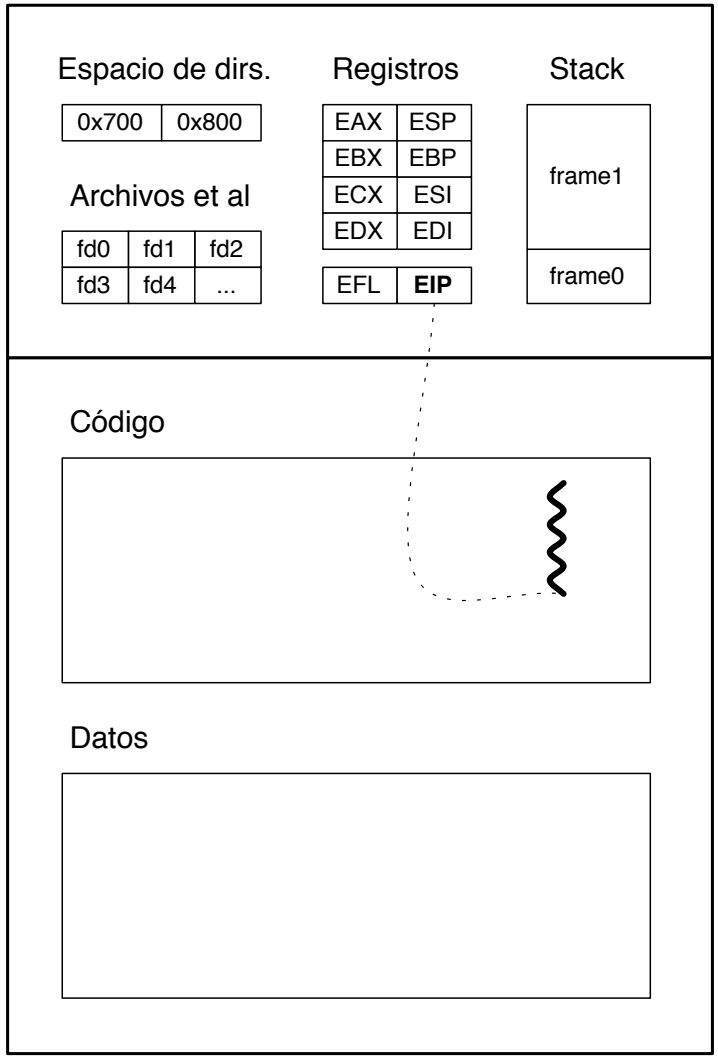
¿Qué es un **proceso**?



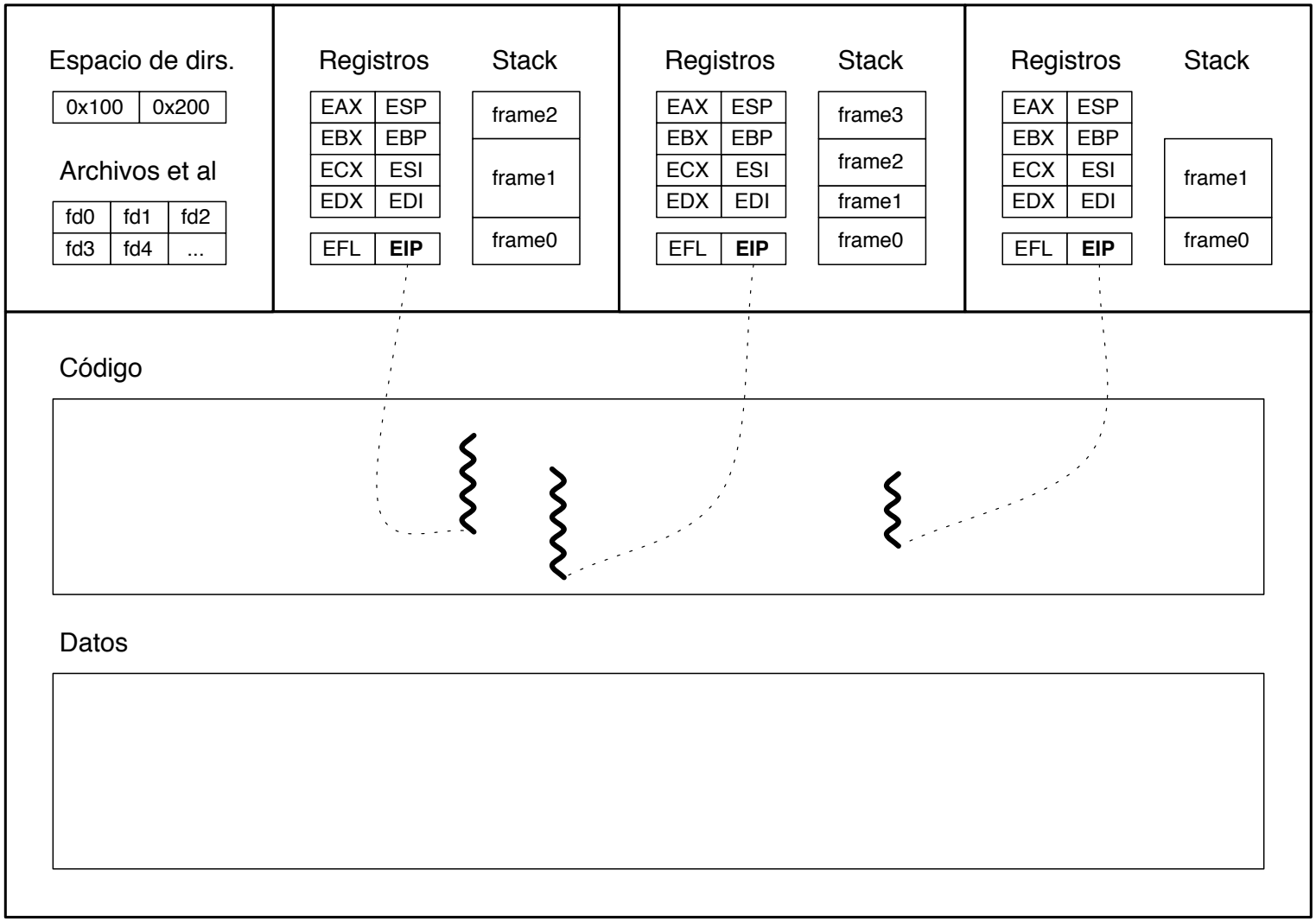
Repaso: Procesos vs. Threads

Podemos tener varios procesos corriendo 'concurrentemente':





¿Qué es un **Thread**?



- Supongamos que tenemos dos **procesos** corriendo en simultáneo: `proceso A` y `proceso B`.

  En estos procesos ocurren **eventos**: `evento A` y `evento B`.

  ¿Cómo sabemos si `evento A` ocurre antes o después que `evento B`?

- En general, no tenemos control sobre el orden de ejecución de los procesos, eso es tarea del scheduler del SO.

Concurrencia

Dos eventos son concurrentes si no podemos decir a simple vista en qué orden ejecutarán.
Por ejemplo porque cualquiera puede ser elegido por el scheduler para ejecutar en un momento dado.

Paralelismo

Entendemos que dos procesos son paralelos si ejecutan literalmente al mismo tiempo (es decir, en distintos procesadores cada uno).

- Nosotros nos vamos a concentrar en procesos concurrentes.

Repaso

Race Condition

- En general los procesos tienen sus variables locales que otros procesos no pueden ver ni acceder.
- Pero a veces tenemos variables compartidas entre procesos/threads.
- ¿Que sucede al mezclar concurrencia y memoria compartida?

  Problemas de sincronización al hacer escrituras, actualizaciones o lecturas concurrentes.

Condición de carrera

Defecto de un programa concurrente por el cual su correctitud depende del orden de ejecución de ciertos eventos.

Estructura de la clase

Ejemplo 1

Varias instancias del mismo programa

Sean `A` y `B` dos instancias en ejecución del mismo programa `'Suma'`.
Ambas corren concurrentemente y comparten una variable `x`.
¿Cuál es la salida esperada si el código que corren es el siguiente?

Memoria compartida:

```
x = 0
```

Suma ():

```
x = x + 1;
print(x);
```

Ojo con la atomicidad

- Pensemos qué pasa a nivel de máquina.
- Esta operación sobre memoria compartida involucra:
  1. Leer valor de `x`
  2. Operar (sumarle 1 a este valor)
  3. Escribir nuevo valor en `x`
- El scheduler puede interrumpir la operación en medio de alguna de estas 3 partes.
- Ejemplo de ejecución (traza):

Proceso A	**Proceso B**	**Estados**
lee `x`		`x=0`
suma 1 a `x`		`x=0`
	lee `x`	`x=0`
almacena 1 en `x`		`x=1`
	suma 1 a `x`	`x=1`
	almacena 1 en `x`	`x=1`

Ojo con la atomicidad

Operación atómica

Una operación es atómica cuando es indivisible. Es decir, cuando no puede ser interrumpida por el procesador hasta terminar.

- En general no sabemos qué operaciones se realizan en un paso y cuáles pueden ser interrumpidas.

¿Qué es una variable atómica?

- ¿Cómo podemos solucionar el problema de race conditions en el problema de recién?

Variable atómica

Una variable atómica es un objeto que nos permite realizar operaciones de escritura y lectura de forma atómica.

- Almacenan un valor entero. Se puede interactuar con la variable mediante algunas primitivas como `getAndInc()` y `getAndAdd()`.
- Estas primitivas son atómicas a efectos de los procesos.

Uso de variables atómicas

Algunas primitivas

- `getAndInc()`: Devuelve el entero atómico sumado 1.
- `getAndAdd(unsigned int value)`: Devuelve el entero atómico sumado la cantidad especificada.
- `set(unsigned int value)`: Asigna al objeto un valor pasado por parámetro.

Más info en `https://en.cppreference.com/w/cpp/atomic/atomic`.

¿Cómo solucionamos la pérdida de sumas del ejemplo?

Esquema de la solución

Memoria compartida:

```
atomic<int> x
x.set(0)
```

Suma ():

```
x.getAndInc()
```

Ejemplo 2

Procesos con programas distintos

Ahora `A` y `B` ejecutan dos programas distintos `Suma_A` y `Suma_B` respectivamente. Ambas corren concurrentemente y comparten una variable `x`.
¿Cuál es la salida esperada si el código que corren es el siguiente?

Memoria compartida:

```
x = 0
```

Suma_A ():

```
x = x + 1;
print(x);
```

Suma_B ():

```
x = x + 2;
print(x);
```

El problema va a ser el mismo. ¿Y la solución? También podemos usar x como variable atómica.

Estructura de la clase

Semáforos

Semáforo

- Es un tipo abstracto de datos que permite controlar el acceso de múltiples procesos a un recurso común.
- Tiene un valor entero, al cuál no podemos acceder. La única manera de interactuar con el semáforo es mediante las primitivas `wait()` y `signal()`.
- Estas primitivas son atómicas a efectos de los procesos.

Uso de semáforos

Primitivas

- `sem(unsigned int value)`: Devuelve un nuevo semáforo inicializado en `value`.
- `wait()`: Mientras el valor sea menor o igual a 0 se bloquea el proceso esperando un `signal`. Luego decrementa el valor de `sem`.
- `signal()`: Incrementa en uno el valor del semáforo y despierta a alguno de los procesos que están esperando en ese semáforo.

Idea detrás de wait() y signal()

```
wait(s):
    while (s<=0) dormir();
    s--;

signal(s):
    s++;
    if (alguien espera por s) despertar a alguno;
```

Importante antes de arrancar

- ¿Puedo saber cuál es el proceso que se despierta en un signal?

  ¡No! Es determinístico pero depende de demasiadas variables.

- ¿Puedo asumir que el proceso que se despierta es el próximo en correr?

  ¡No! Es determinístico pero depende de demasiadas variables.

- ¿Puedo consultar el valor de un semáforo?

  ¡No! Revisar la interfaz, no hay observador.

Ejemplo 1

Solución con semáforos

¿Cómo podemos usar semáforos para resolver el problema del Ejemplo 1?

Esquema de la solución

Memoria:

```
x = 0
mutex = sem(1)
```

Suma():

```
mutex.wait()
x = x + 1
mutex.signal()
```

Ejemplo 2

Solución con semáforos

¿Y si quisiéramos resolver el ejemplo 2 con semáforos?

Esquema de la solución

Memoria:

```
x = 0
mutex = sem(1)
```

Suma_A:

```
mutex.wait()
x = x + 1
mutex.signal()
```

Suma_B:

```
mutex.wait()
x = x + 2
mutex.signal()
```

- ¿Estaría bien usar dos mutex distintos, uno para cada tipo de proceso? **¡NO!**
- Para garantizar que hay un solo proceso ejecutando la sección crítica a la vez, ambos deben estar mirando el mismo semáforo.

Generalización: Sección Crítica

Recordemos: ¿a qué llamamos sección crítica?

Sección crítica

Llamamos sección crítica a la parte del programa que accede a memoria compartida, y queremos que ejecute atómicamente.

¿Y exclusión mutua?

Exclusión mutua

Problema de asegurar que dos threads no ejecutan una sección crítica simultáneamente.

Generalización: Problema de la Sección Crítica

¿Cómo podemos generalizar lo anterior para resolver el problema de la sección crítica?

Esquema de la solución

Memoria:

```
variable_1 = 0
variable_m = 0
mutex = sem(1)
```

Programa_i:

```
Sección no crítica
mutex.wait()
SECCIÓN CRÍTICA
mutex.signal()
Sección no Crítica
```

Sincronización

- Además del problema de la exclusión mutua, los semáforos nos van a permitir sincronizar a los procesos.

- Veamos algunos ejercicios.
- OJO: tenemos que tener cuidado de no generar deadlocks.

Estructura de la clase

Ejercicio 1

Signaling

Enunciado

Se tienen un proceso productor P que puede `producir()` y dos procesos consumidores C_1, C_2 que hacen `consumir1()` y `consumir2()` respectivamente. Se desea sincronizarlos tal que las secuencias de ejecución sean: `producir, producir, consumir1, consumir2, producir, producir, consumir1, consumir2,...`

Solución

`permisoC1 = sem(0), permisoC2 = sem(0), permisoP = sem(1)`

P

```
permisoP.wait()
producir()
producir()
permisoC1.signal()
```

C1

```
permisoC1.wait()
consumir1()
permisoC2.signal()
```

C2

```
permisoC2.wait()
consumir2()
permisoP.signal()
```

Ejercicio 2

Signaling

Enunciado

Se tienen 2 procesos A y B.
El proceso A tiene que ejecutar `A1()` y luego `A2()`.
`B` debe ejecutar `B1()` y después `B2()`.
En cualquier ejecución, `A1()` tiene que ejecutarse antes que `B2()`.
Escribir el código con semáforos tal que cualquier ejecución cumpla lo pedido.

Solución

```
permisoB = sem(0)
```

A

```
A1()
permisoB.signal()
A2()
```

B

```
B1()
permisoB.wait()
B2()
```

Ejercicio 2

Signaling

¿Que problemas tiene esta solución?

NO Solución

```
atomic<int> ejecuto_a1 = false
```

A

```
A1()
ejecuto_a1 = true
A2()
```

B

```
B1()
while(! ejecuto_a1){ }
ejecuto_a1 = false
B2()
```

- Hace **BUSY WAITING**. En el parcial NO hagan busy waiting...
- La comparación y definición de "terminó" no se hacen atómicamente.
- Extra: de asumir que pueden llegar varios procesos de tipo A y B, antes permitíamos que ejecuten tantos B2() como A1(), pero ahora eso ya no necesariamente se cumple.

Ejercicio 3

Rendezvous / Barrera

Enunciado

Usando los procesos A y B del ejercicio anterior, ahora se quiere que `A1()` y `B1()` ejecuten antes de `B2()` y `A2()`.

¿Solución?

```
permisoB = sem(0)
permisoA = sem(0)
```

A

```
A1()
permisoA.wait()
permisoB.signal()
A2()
```

B

```
B1()
permisoB.wait()
permisoA.signal()
B2()
```

Pista: **No**

Problema: Existe un *DEADLOCK*

Algunos tips para identificar deadlocks:

- **Recursos Bloqueados:** Los procesos esperan indefinidamente por recursos que están siendo retenidos por otros procesos.
- **Espera Circular:** Existe una cadena circular de procesos, donde cada uno está esperando el recurso del siguiente proceso en la cadena.
- **No Liberación:** Un proceso mantiene recursos mientras espera otros, sin liberar los recursos que ya posee.
- **Tiempo de Espera Infinito:** Un proceso espera indefinidamente para entrar a una sección crítica.

Ejercicio 3

Rendezvous / Barrera

Enunciado

Usando los procesos A y B del ejercicio anterior, ahora se quiere que `A1()` y `B1()` ejecuten antes de `B2()` y `A2()`.

Solución

```
permisoB = sem(0)
permisoA = sem(0)
```

A

```
A1()
permisoB.signal()
permisoA.wait()
A2()
```

B

```
B1()
permisoA.signal()
permisoB.wait()
B2()
```

Nota: Este patrón se suele llamar rendezvous (punto de encuentro) o barrera.

Ejercicio 4

Muchos estudiantes para un TP

Enunciado

Un grupo de N estudiantes se dispone a hacer un TP de su materia favorita.
Cada estudiante conoce a la perfección cómo `implementarTp()` y cómo `experimentar()`. Curiosamente, cada etapa puede ser llevada acabo de manera independiente por cada une, así que decidieron dividirse el trabajo.
Sin embargo, acordaron que para que alguien pudiera `experimentar()` todes deberían haber terminado de `implementarTp()`.
Se pide diseñar un programa concurrente que utilice procesos y que modele esta situación utilizando semáforos.

Ejercicio 4

Solución con turnstile / molinete

```
barrera = sem(0)
mutex = sem(1)
int counter = 0

ProcesoEstudiantes():
    implementarTp()

    mutex.wait()
    counter++
    if (counter == n): barrera.signal()
    mutex.signal()

    barrera.wait()
    barrera.signal()

    experimentar()
```

Ejercicio 4

Alternativa con múltiples signals

```
barrera.signal(unsigned int n):
        for(i = 0; i < n; i++):
                barrera.signal()
```

Ejercicio 4

Alternativa con múltiples signals

```
barrera = sem(0)
mutex = sem(1)
int counter = 0

ProcesoEstudiantes():
    implementarTp()

    mutex.wait()
    counter++
    if (counter == n): barrera.signal(n)
    mutex.signal()

    barrera.wait()

    experimentar()
```

Ejercicio 4

Reloaded

Para pensar en casa:

Supongamos que el grupo se da cuenta de que su esquema de trabajo es una pésima idea. Ahora quieren dividir el trabajo en varias etapas de implementación-experimentación. Es decir, querrían hacer algo similar a lo anterior, pero repetirlo varias veces.

Esquema

```
// Memoria compartida
ProcesoEstudiantes():
    While(1){
        implementarTp()
        // sincronizacion
        experimentar()
        // sincronizacion
    }
```

Momento para preguntas

Estructura de la clase

Resumen

Hoy vimos...

- Concurrencia
  - ¡Cuidado con las race conditions!
- Variables atómicas
- Semáforos y ejercicios de sincronización
  - ¡Cuidado con los deadlocks!
- Hay más cuestiones que tener en cuenta a la hora de sincronizar procesos, las verán más en detalle en "Programación Concurrente y Paralela".

Resumen

Vimos los patrones de sincronización

- Mutex (Exclusión mutua del acceso a la sección crítica)
- Signaling (Cuando A pasa antes que B)
- Rendezvous (Cuando A1 pasa antes que B2 y B1 pasa antes que A2)
- Barrera con Turnstile
- Barrera reutilizable (les queda para pensar)

Cómo seguimos...

Con esto se puede resolver toda la guía práctica 3.