Programación de Servicios y Procesos

PROGRAMACIÓN MULTIPROCESO

Unidad 2: Concurrencia

1. OBJETIVOS

- Introducir el concepto de concurrencia.
- Conocer las ventajas y los problemas derivados de la programación concurrente.
- Aprender más acerca de los conceptos de comunicación y sincronización.

ÍNDICE

1. INTRODUCCIÓN

- **Ordenadores modernos**: varias CPU o CPU con varios núcleos.
- **Concurrencia:** varios programas o varias partes de un programa que se ejecutan durante el mismo período de tiempo (**intercalados** o incluso **simultáneos**)
- Los procesos o subprocesos que se ejecutan simultáneamente podrían **colaborar** y/o **competir por los recursos:**
  - se requerirán mecanismos de **comunicación** y **sincronización** entre procesos.
- La **programación concurrente** abarca:
  - Técnicas para **ejecutar** programas en **concurrencia**.
  - Mecanismos para resolver problemas de **comunicación** y **sincronización**.

2. Beneficios

- La concurrencia introduce **complejidad**.
- Sin embargo, existen ventajas a la hora de utilizer concucrrencia:
  - Aumenta el **rendimiento** de la CPU
  - **Velocidad**: Las Aplicaciones trabajan más rápido en concurrencia.
  - Muchos sistemas son concurrentes por **naturaleza**.

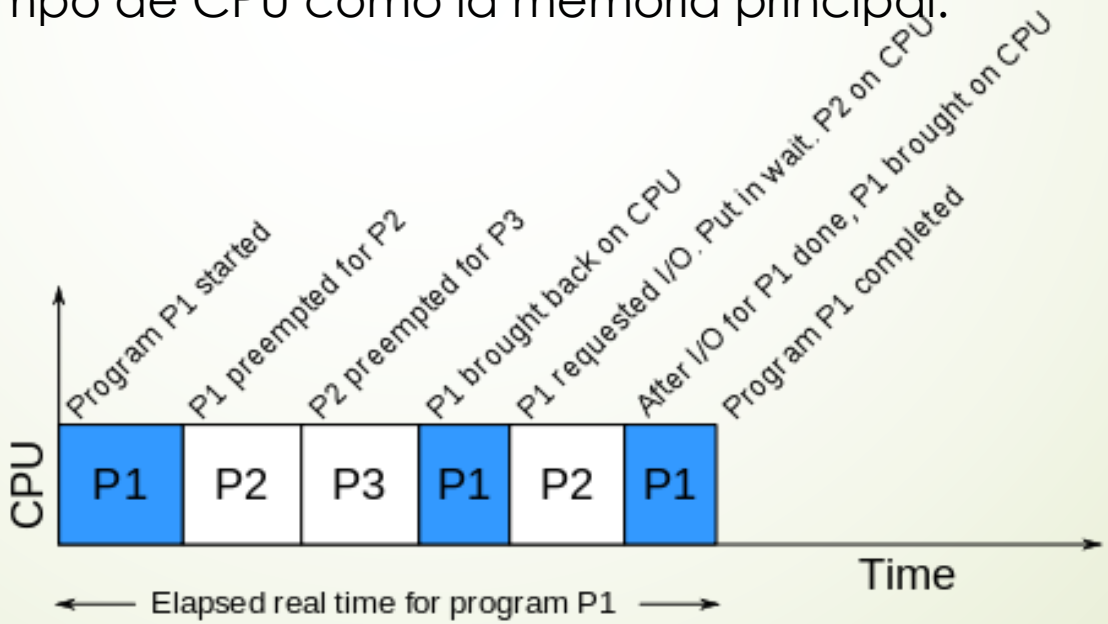
Actividad:

Para cada problema, rellena la tabla con beneficios y pon ejemplos.

Problema	Beneficios	Ejemplos
Sistemas de Controls (dispositivos, sensors para obtener datos, toma decisiones)		
Tecnología Web		
Interacción con la GUI		
Simulación (de procesos/sistemas del mundo Real)		
DBMS (SGBD en Castellano)		

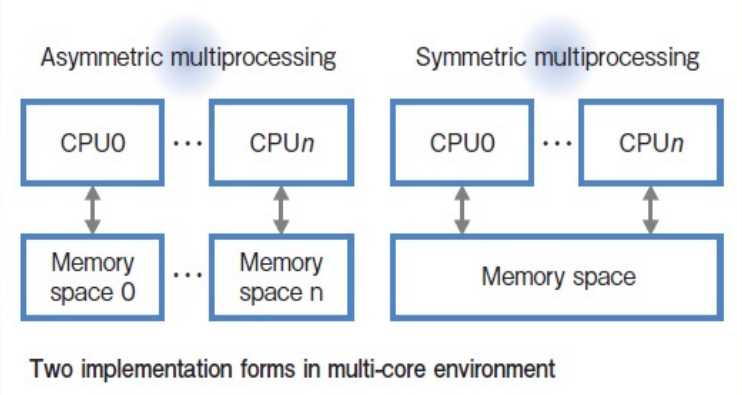
3. Arquitectura de la CPU

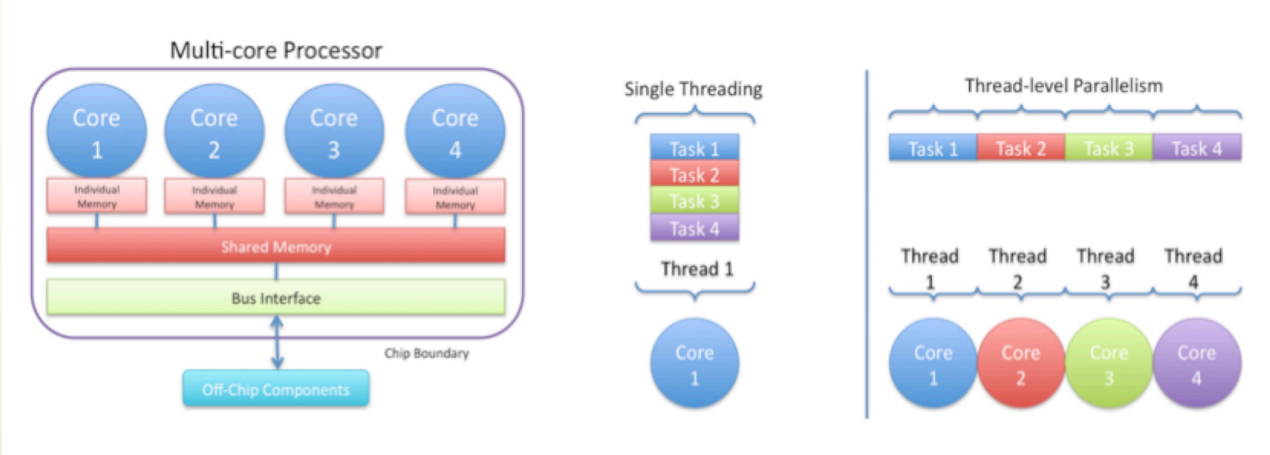
- La concurrencia se implementa tanto en sistemas de un **único procesador** como en sistemas de **múltiples procesadores**
  - **Procesador único**: Los procesos comparten tanto el tiempo de CPU como la memoria principal.



3. Arquitectura de procesador

- Sistemas **Multiprocesador**:





3. CPU Arquitecture

"Independientemente del número de procesadores o de su arquitectura, un programa concurrente siempre se debe funcionar adecuadamente"

4. Complejidad y propiedades

- Programar utilizando concurrencia es una tarea **compleja**.

- **Principios** de los programas concurrentes:
  - **Seguridad**:

    "*Nada malo puede ocurrir*"
  - **Liveness**:

    "*Finalmente algo Bueno ocurre*"

4. Complejidad y Propiedades

Seguridad

- La **seguridad** se encarga de que los datos permanecen consistentes **durante** y **después** las operaciones
- **Ejemplo**: Imagina quen las funciones A y B se ejecutan las dos en concurrencia.

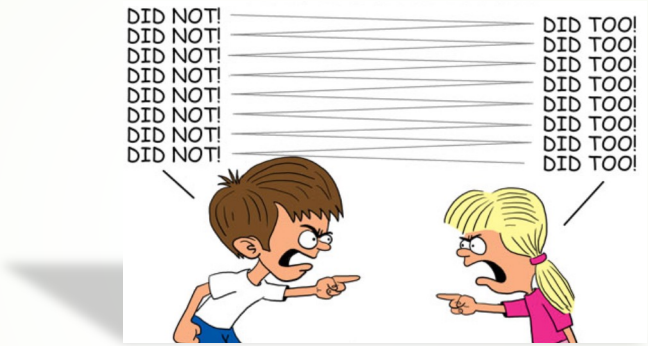
```
var x = 0;
function A()
     {x = x + 1;}

function B()
     {x = x + 2;}
```

¿Cuál es el valor final de **x** ?

- **x = 3** → S las operaciones **x=x+1** y **x=x+2** son ***atómicas***, i.e. no pueden interrumpirse.
- **x = 1, 2 o 3** → Se las opera**x=x+1** and **x=x+2** pueden interrumpirse mutuamente.

6. CONCURRENCIA
Actividad

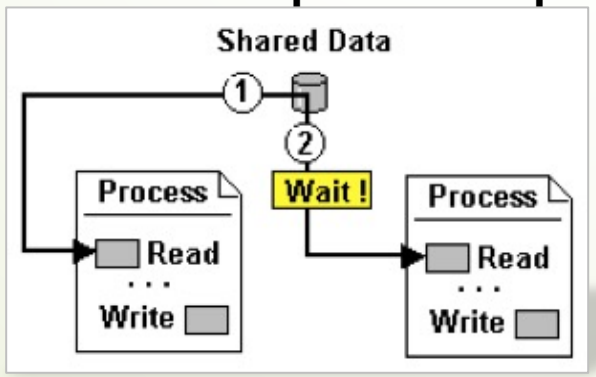


- ¿Que son las "***condiciones de carrera***" ?

- ¿Cómo podemos **prevenirlas**?

6. CONCURRENCIA Complejidad y Propiedades

- La **seguridad** se garantiza cuando, al operar con datos compartidos, se implementa:
  - **Exclusión mutua**: solo cuando un proceso/hilo puede acceder a los datos compartidos al mismo tiempo → **Actualizaciones atómicas**
  - **Sincronización por condición**: las operaciones en datos compartidos son pospuestas si no están en el estado correctos → **¡han de esperar!**



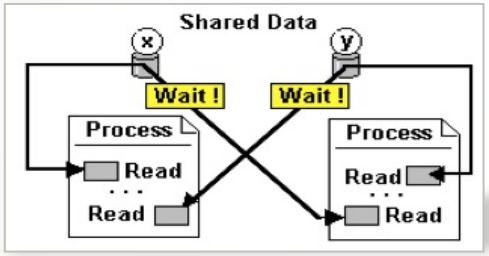
→ *Deben **esperar** si el buffer está vacío*

4. Complejidad y Propiedades
Actividad

- Busca información acerca del **problema productor-consumidor**.
  - ¿Cuales son esos "problemas"?
  - ¿Cómo se solucionan?

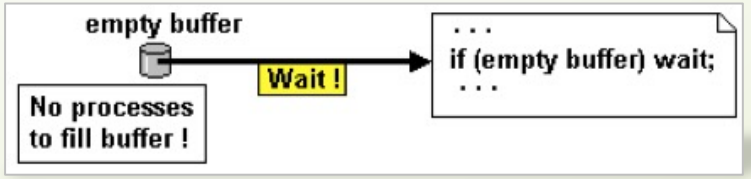
4. Complejidad y propiedades
LIVENESS

- **Exclusión mutua** soluciona muchos problemas de seguridad, pero conduce a otros problemas **graves**:
  - **Deadlock**: (un bug muy común de programación)



Ambos procesos se quedan a la espera de que el otro libere el bloqueo necesario, pero ninguno de los procesos liberará su propio bloqueo porque ambos están bloqueados esperando que el otro bloqueo se libere primero.

  - Deadlock es una forma extrema de inanición: un proceso / hilo no puede continuar porque puede obtener acceso a un recurso que requiere..



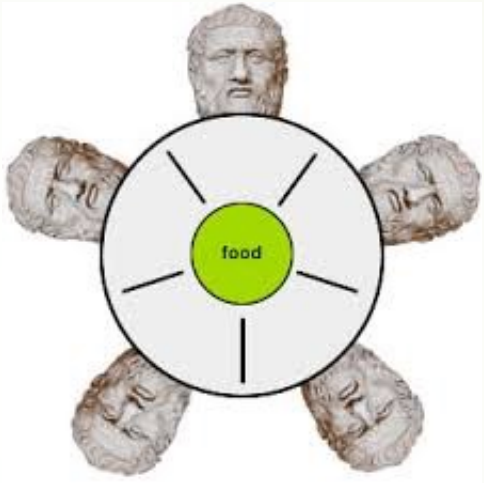
4. Complejidad y propiedades
Actividad

- Que es la **"*espera circular*"**?
- Pon Ejemplos.

4. Complexity and properties
LIVENESS

- **Liveness** asegura que finalmente algo bueno ocurrirá.
- Un programa concurrente tiene la propiedad '**liveness**' si:
  - **No Deadlock:** puede acceder siempre a un recurso compartido.
  - **No Starvation**: puede finalmente acceder a un recurso compartido.
- Liveness es **gradacional**: un programa puede trabajar de manera lenta o mala.

4. Complejidad y Propiedades
Actividad



- Busca información acerca del problema **Problema de la Cena de los Filósofos**
- Describe los "**problemas**"

4. Complejidad y propiedades

Ejercicio: El juego del pañuelo

- Considera el "juego del pañuelo":
  - Identificar cada elemento del juego relacionado con la concurrencia.
  - Rellena en la tabla, explicando cada situación:

Propiedad	Explicación
Exclusión Mutua	
Sincronización de la condición	
Deadlock	
Inanición	

5. Comunicación y Técnicas de Sincronización

- **Orientadas a Procedimiento**: basadas en variables compartidas:
  - **Espera Activa**: Comprobar repetidamente una variable.

```
var iAmReady = false;
    function A()
    {
    . . .
    // signal (!)
    iAmReady = true;
    . . .
    }
```

```
function B()
    {
    . . .
    if (!iAmReady){
    //   wait 300 mseconds
      setTimeout("B()",300);return;}
    . . .
    }
```

  - **Semáforos:** *Los semáforos fueron introducidos por Dijkstra en 1968 como una primitiva de nivel superior para la sincronización. El semáforo es un entero que tiene dos operaciones*:
    - **P**(s): delays (*espera*) hasta s>0 entonces, atómicamente ejecuta s = s - 1
    - **V**(s): atómicamente ejecuta (*señal*) s = s + 1
  - **Monitores**: encapsula recursos y proporciona operaciones de bloqueo/desbloqueo que los manipulan.

5. Comunicación y Técnicas de Sincronización

- ***Orientadas a Mensajes:*** Paso de Mensajes

```
function sender(mX)
    {
    . . .
    m = mX;
    // point of Rendezvous
    // wait if the receiver is not ready
    send(m,"receiver");
    . . .
    }
function receiver()
    {
    var Message = new Buffer(); . . .
    // point of Rendezvous
    // wait if the sender is not ready
    x = Message;
    . . .
    }
```

- ***Orientadas a Operaciones:*** Llamada a Procedimiento Remoto (RPC)