PROGRAMACIÓN CONCURRENTE

Introducción

- El concepto fundamental de la programación concurrente es la noción de **Proceso**.
- **Proceso**: Cálculo secuencial con su propio flujo de control.
- La concurrencia en software implica la existencia de diversos flujos de control en un mismo programa colaborando para resolver un problema.

Procesos vs. Hilos

- En el contexto del Sistema Operativo, un Proceso es una instancia de un Programa que está siendo ejecutado en el ordenador.

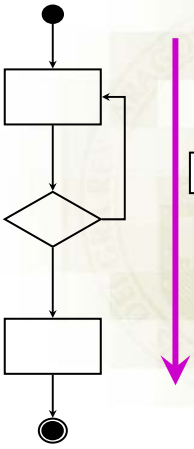
  Proceso = Código de programa + Datos + Recursos

- Un S.O. admite concurrencia si es capaz de manejar diversos procesos simultáneamente.

- En el contexto de un Programa concurrente, un Hilo (*Thread*) es cada uno de los flujos secuenciales de control independientes especificados en el programa.

Procesos Secuenciales

- Un proceso tradicional, correspondiente a un programa secuencial, es un proceso que posee un único hilo de control.

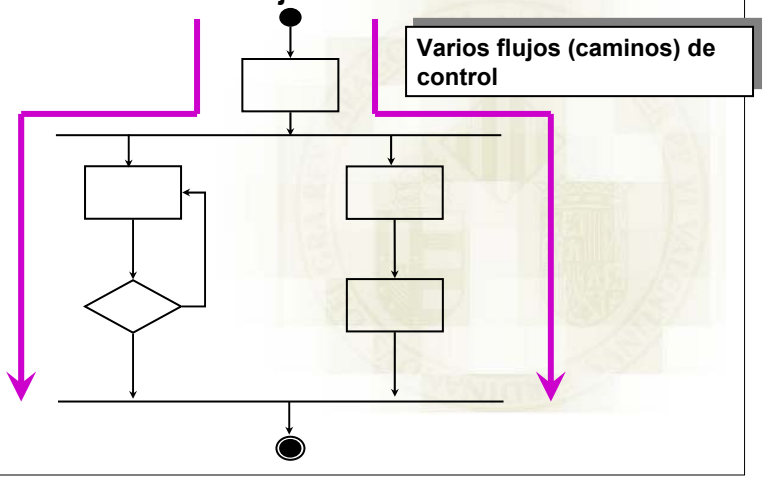


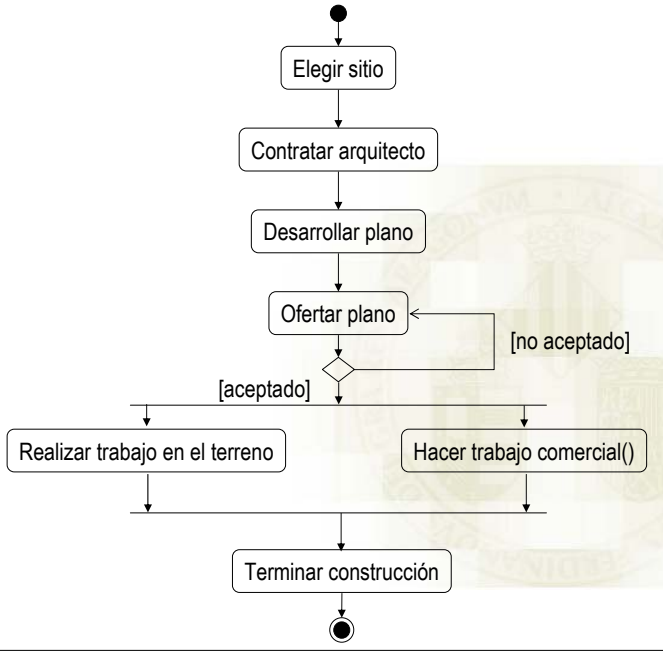
Un flujo (camino) de control

El contador de programa, el estado de la pila y de los objetos definen el estado del sistema.

Procesos Concurrentes

- Un programa concurrente da lugar, durante su ejecución, a un proceso con varios hilos de ejecución.





Concurrencia Software vs. Paralelismo Hardware

- **La concurrencia software es un concepto lógico, no implica la existencia de paralelismo en el hardware:**
  - ✓ Las operaciones hardware ocurren en paralelo si ocurren al mismo tiempo.
  - ✓ Las operaciones (software) en un programa son concurrentes si pueden ejecutarse en paralelo, aunque no necesariamente deben ejecutarse así.

Tipos de concurrencia

- **Concurrencia Física:**
  - ✓ Existe más de un procesador y varias unidades (hilos) de un mismo programa se ejecutan realmente de forma simultánea.
- **Concurrencia Lógica:**
  - ✓ Asumir la existencia de varios procesadores, aunque no existan físicamente. El implementador de tareas del lenguaje se encargará de "mapear" la concurrencia lógica sobre el hardware realmente disponible.
- **La concurrencia lógica es más general, pues el diseño del programa no está condicionado por los recursos de computación disponibles.**

Arquitecturas Multiprocesador: Revisión histórica

- **Años 50: Primeras máquinas con varios procesadores**
  - ✓ Un procesador de propósito general y varios procesadores para controlar la E/S.
  - ✓ Permitían ejecutar un programa mientras se realizaba E o S para otros programas.
  - ✓ No se puede hablar de ejecución concurrente de programas.
- **Años 60 (inicio): Primeras multiproc. reales**
  - ✓ El programador de tareas del SO distribuía programas entre los procesadores a partir de una cola de trabajos.
  - ✓ Permitían concurrencia a nivel de programas.

Arquitecturas Multiprocesador (2)

- **Años 60 (mitad): Ordenadores con hardware específico múltiple**
  - ✓ Multiplican el número de multiplicadores en coma flotante o unidades aritméticas en coma flotante completas.
  - ✓ Estas unidades ejecutan instrucciones procedentes de un único flujo de instrucciones (programa).
  - ✓ Los compiladores determinaban qué instrucciones podían ejecutarse concurrentemente.

Arquitecturas Multiprocesador: Categorías

- **SIMD (*Single Instruction Múltiple Data*):**
  - ✓ Pueden ejecutar simultáneamente la misma instrucción sobre diferentes conjuntos de datos.
  - ✓ Cada procesador tiene su propia memoria local.
  - ✓ Hay un procesador que controla el trabajo del resto.
  - ✓ Como se ejecuta la misma instrucción no es preciso ningún tipo de sincronización entre tareas.
  - ✓ Máquinas típicas: los procesadores vectoriales
    - Específicas para trabajar con datos almacenados en arrays.
    - Aplicación en cálculo científico.

Arquitecturas Multiprocesador: Categorías (2)

- **MIMD (*Multiple Instruction Multiple Data*)**
  - ✓ Cada procesador ejecuta su propio flujo de instrucciones.
  - ✓ Es preciso poder sincronizar tareas.
  - ✓ Dos tipos de configuración:
    - Memoria compartida.
    - Memoria distribuida.
  - ✓ Hay que sincronizar el acceso a los datos.
  - ✓ Soportan concurrencia a nivel de hilos dentro del programa.

Concurrencia en un Programa

- **Desde el punto de vista de la programación, interesa la concurrencia lógica que puede existir en el interior de un programa.**
- **Un Lenguaje de Programación será concurrente si posee las estructuras necesarias para definir y manejar diferentes tareas (hilos de ejecución) dentro de un programa.**
  - ✓ Ejemplos: Java, Ada
- **El compilador y el SO serán los responsables de "mapear" la concurrencia lógica del programa sobre el hardware disponible.**

Sincronización de tareas

- **La Sincronización es el mecanismo que controla el orden en que se ejecutan las tareas.**
- **Tipos de Sincronización:**
  - ✓ Cooperación:
    - La tarea **`A`** debe de esperar que la tarea **`B`** finalice alguna actividad para poder continuar con su ejecución.
  - ✓ Competición:
    - La tarea **`A`** necesita acceder a un dato **`x`** mientras la tarea **`B`** está accediendo al mismo. La tarea **`A`** debe de esperar a que la tarea **`B`** finalice su uso de **`x`** para poder continuar.

Necesidad de Sincronización (ejemplo)

- **Ejemplo de Competición:**
  - ✓ Sea un dato `TOTAL` que inicialmente tiene el valor 3.
    - La tarea `A` realiza: `TOTAL++;`
    - La tarea `B` realiza: `TOTAL *= 2;`
  - ✓ Cada tarea completa su operación sobre `TOTAL` realizando los siguientes pasos:
    - Recuperar el valor de `TOTAL`
    - Realizar una operación aritmética
    - Almacenar el nuevo valor de `TOTAL`

Sincronización para Competición (ejemplo)

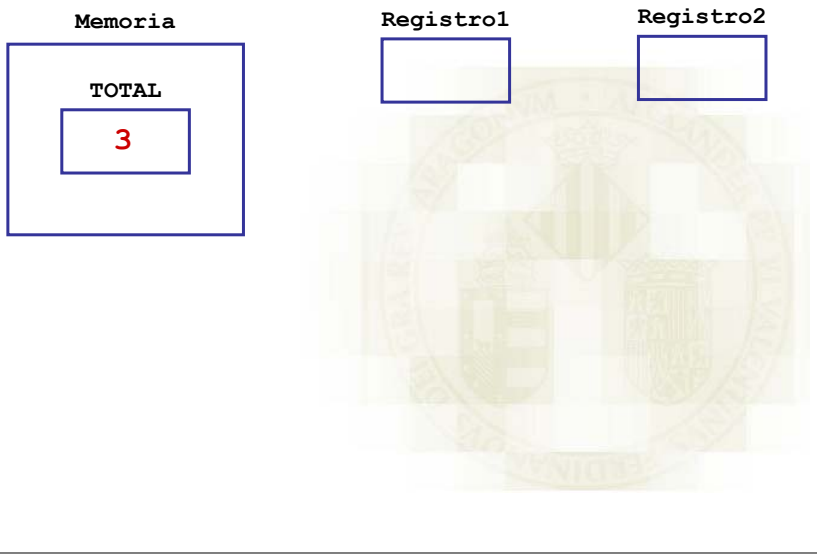
- **Si no existe sincronización entre las tareas `A` y `B`, se pueden dar cuatro resultados diferentes (`TOTAL = 3`):**
  - ✓ Correctos:

  La tarea `A` completa su operación antes de que empiece `B`:
  - Resultado final, `TOTAL = 8`

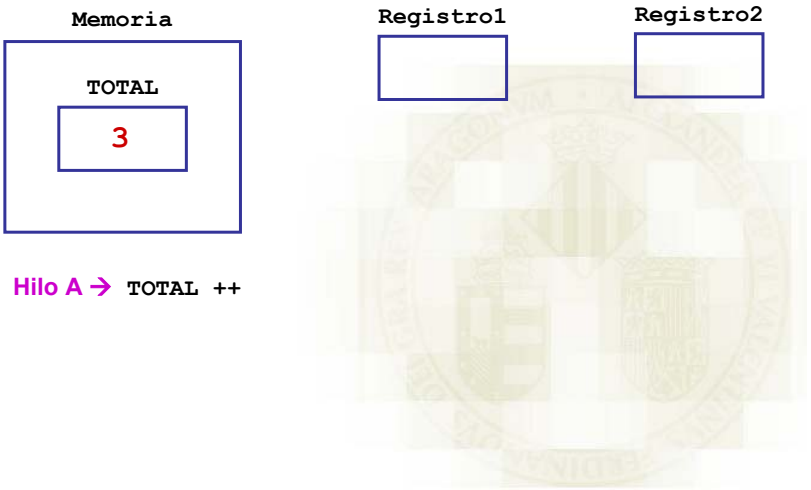
  La tarea `B` completa su operación antes de que empiece `A`:
  - Resultado final, `TOTAL = 7`
  - ✓ Incorrectos:

  Las tareas `A` y `B` recuperan el valor de `TOTAL` sin que ninguna haya llegado a almacenar el nuevo valor:
  - Resultado final, `TOTAL = 6`. Si la última en finalizar es `B`.
  - Resultado final, `TOTAL = 4`. Si la última en finalizar es `A`.

Sincronización (Ejemplo)



Sincronización (Ejemplo: A antes que B)



Hilo A → `TOTAL ++`

Sincronización (Ejemplo: A antes que B)

Hilo A → `TOTAL ++`

```
(i) Recuperar (TOTAL)
```


Sincronización (Ejemplo: A antes que B)

Hilo A → `TOTAL ++`

```
(i) Recuperar (TOTAL)
(ii) Sumar 1
```

Sincronización (Ejemplo: A antes que B)

Hilo A → `TOTAL ++`

```
(i) Recuperar (TOTAL)
(ii) Sumar 1
(iii) Almacenar (TOTAL)
```


Sincronización (Ejemplo: A antes que B)

Hilo A → `TOTAL ++`

```
(i) Recuperar (TOTAL)
(ii) Sumar 1
(iii) Almacenar (TOTAL)
```

Hilo B → `TOTAL *=2`

Sincronización (Ejemplo: A antes que B)

Hilo A → `TOTAL ++`
```
(i) Recuperar (TOTAL)
(ii) Sumar 1
(iii) Almacenar (TOTAL)
```
Hilo B → `TOTAL *=2`
```
(i) Recuperar (TOTAL)
```


Sincronización (Ejemplo: A antes que B)

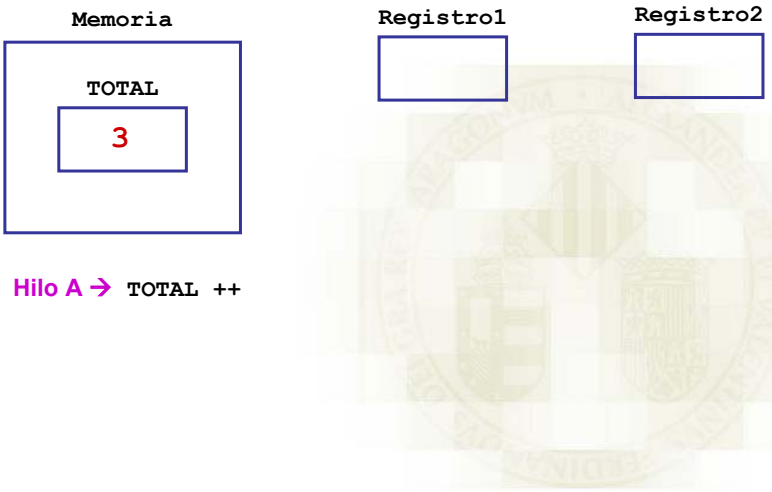
Hilo A → `TOTAL ++`
```
(i) Recuperar (TOTAL)
(ii) Sumar 1
(iii) Almacenar (TOTAL)
```
Hilo B → `TOTAL *=2`
```
(i) Recuperar (TOTAL)
(ii) Multiplicar x 2
```

Sincronización (Ejemplo: A antes que B)

Hilo A → `TOTAL ++`
```
(i) Recuperar (TOTAL)
(ii) Sumar 1
(iii) Almacenar (TOTAL)
```
Hilo B → `TOTAL *=2`
```
(i) Recuperar (TOTAL)
(ii) Multiplicar x 2
(iii) Almacenar (TOTAL)
```

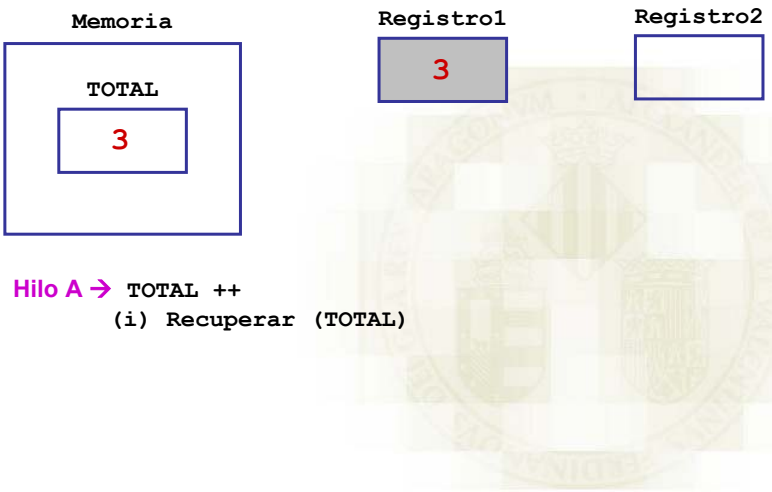

Sincronización (Ejemplo)

Sincronización (Ejemplo: B interfiere con A)



Hilo A → `TOTAL ++`

Sincronización (Ejemplo: B interfiere con A)



Hilo A → `TOTAL ++`
`(i) Recuperar (TOTAL)`

Sincronización (Ejemplo: B interfiere con A)

Hilo A → `TOTAL ++`
`(i) Recuperar (TOTAL)`

Hilo B → `TOTAL *=2`
`(i) Recuperar (TOTAL)`

Sincronización (Ejemplo: B interfiere con A)

Hilo A → `TOTAL ++`
`(i) Recuperar (TOTAL)`
`(ii) Sumar 1`

Hilo B → `TOTAL *=2`
`(i) Recuperar (TOTAL)`

Sincronización (Ejemplo: B interfiere con A)

Hilo A → `TOTAL ++`
`(i) Recuperar (TOTAL)`
`(ii) Sumar 1`

Hilo B → `TOTAL *=2`
`(i) Recuperar (TOTAL)`
`(ii) Multiplicar x 2`

Sincronización (Ejemplo: B interfiere con A)

Hilo A → `TOTAL ++`
`(i) Recuperar (TOTAL)`
`(ii) Sumar 1`
`(iii) Almacenar (TOTAL)`

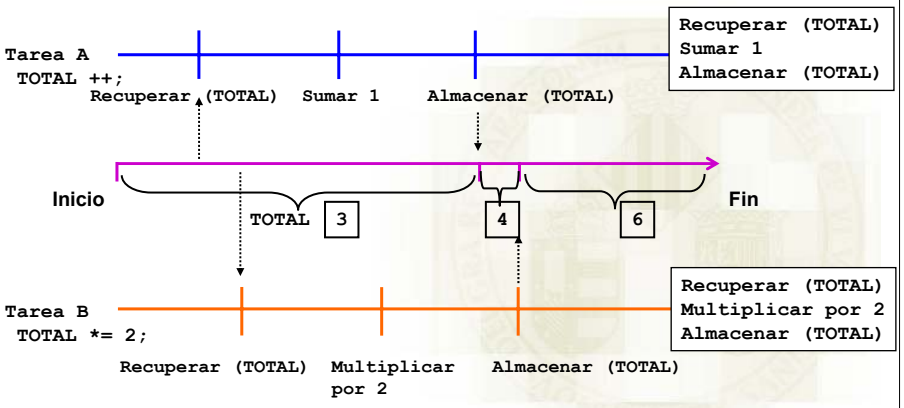
Hilo B → `TOTAL *=2`
`(i) Recuperar (TOTAL)`
`(ii) Multiplicar x 2`

Sincronización (Ejemplo: B interfiere con A)

Hilo A → TOTAL ++
(i) Recuperar (TOTAL)
(ii) Sumar 1
(iii) Almacenar (TOTAL)

Hilo B → TOTAL *=2
(i) Recuperar (TOTAL)
(ii) Multiplicar x 2
(iii) Almacenar (TOTAL)

Sincronización para Competición (ejemplo)



Diseño de lenguajes Concurrentes

- **Los principales conceptos que debe incorporar un lenguaje de programación que soporte Concurrencia son:**
  - ✓ ¿Cómo se soporta la sincronización para Cooperación?
  - ✓ ¿Cómo se soporta la sincronización para Competición?
  - ✓ ¿Cómo y cuándo se inicia y finaliza la ejecución de tareas?
  - ✓ ¿Las tareas se crean estática o dinámicamente?

Semáforos

- **Dijkstra, 1965.**
- **Un semáforo es una estructura de datos que consiste en:**
  - ✓ Un valor entero.
  - ✓ Una cola de tareas (descriptores):
    - Los descriptores de tareas almacenan la información relevante sobre el estado de ejecución de la tarea.
- **Para establecer limitación de acceso a los datos, se establecen protecciones (semáforo) en las operaciones de acceso a dichos datos.**
- **Las protecciones (semáforos) deben permitir acceder a los datos sólo a una tarea en un instante determinado.**

Semáforos (2)

- **Una parte importante del mecanismo de protección consiste en asegurar que todos los intentos de acceso a los datos protegidos se llevarán a cabo en algún momento:**
  - ✓ Cola de tareas.
- **Las únicas operaciones permitidas en los semáforos son:**
  - ✓ `P` (del holandés *passeren*): `esperar`
  - ✓ `V` (del holandés *vrygeren*): `avisar`

Ejemplo de Sincronización para Cooperación

- **Los datos compartidos están en un buffer (situación habitual).**
- **Operaciones de acceso al buffer:**
  - ✓ `ALMACENAR`
  - ✓ `EXTRAER`
- **Hay 2 tareas que acceden al buffer para Cooperar:**
  - ✓ Siguen el esquema Productor-Consumidor:
    - Una tarea produce datos y los almacena en el buffer.
    - La segunda tarea requiere datos y los extrae del buffer.

Ejemplo (2, sin sincronización)

- **Versión no sincronizada de las tareas:**

```
Tarea Productor:
   while ()
   {
      //generar valor...
      buffer.almacenar (valor);
   }
Fin_tarea
```

```
Tarea Consumidor:
   while ()
   {
      valor = buffer.extraer ();
      //utilizar valor...
   }
Fin_tarea
```

- **Problemas:**
  - ✓ El Productor genera valores más rápidamente de lo que los consume el Consumidor → el buffer se llena.
  - ✓ El Consumidor requiere valores más rápidamente de lo que los produce el Productor → el buffer se vacía.

Ejemplo (2, con sincronización)

- **Versión sincronizada con semáforos de las tareas:**

```
Tarea Productor:
   while ()
   {
      //generar valor...
      esperar (posLibres);
      buffer.almacenar (valor);
      avisar (posOcupadas);
   }
Fin_tarea
```

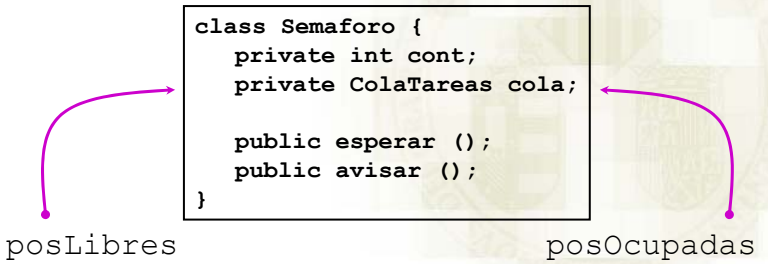
```
Tarea Consumidor:
   while ()
   {
      esperar (posOcupadas);
      valor = buffer.extraer ();
      avisar (posLibres);
      //utilizar valor...
   }
Fin_tarea
```

Espera a que existan posiciones libres en el buffer, mientras tanto la tarea está bloqueada.

Espera a que existan posiciones ocupadas en el buffer, mientras tanto la tarea está bloqueada.

Ejemplo (3)

- **Hay 2 aspectos a proteger sobre el acceso al buffer:**
  - ✓ `posLibres` (necesaria para el Productor)
  - ✓ `posOcupadas` (necesaria para el Consumidor)
- **Se realiza la protección mediante 2 semáforos:**



Ejemplo (4)

`posLibres`:

- `cont` → cuantas posiciones libres hay disponibles en el buffer
- `cola` → las tareas que están esperando poder almacenar datos en el buffer

`posOcupadas`:

- `cont` → cuantas posiciones ocupadas (datos) hay disponibles en el buffer
- `cola` → las tareas que están esperando poder extraer datos en el buffer

Esperar / Avisar

```
esperar () {
    if ( this.cont > 0 ) this.cont --;
    // correcto hay disponibilidad, decremento cont
    // p.q. se va a realizar 1 operación protegida
    else
        this.cola.insertar (tarea que invoca esperar);
        // no hay disponibilidad, encolar la tarea para
        // continuar con ella cuando sea posible
}
```

```
avisar () {
    if ( this.cola.esVacia() ) this.cont ++;
    // no hay tareas esperando este aviso,
    // incremento la disponibilidad
    else
        return ( this.cola.extraer () );
        // había tareas esperando, extraer la primera
}
```


Interpretación de las tareas

```
Tarea Productor:
  while ()
  {
    //generar valor...
    posLibres.esperar();
    buffer.almacenar (valor);
    posOcupadas.avisar();
  }
Fin_tarea
```

¿hay disponibilidad de posiciones libres?
SI: la tarea continúa.
NO: el Productor se guarda en la cola

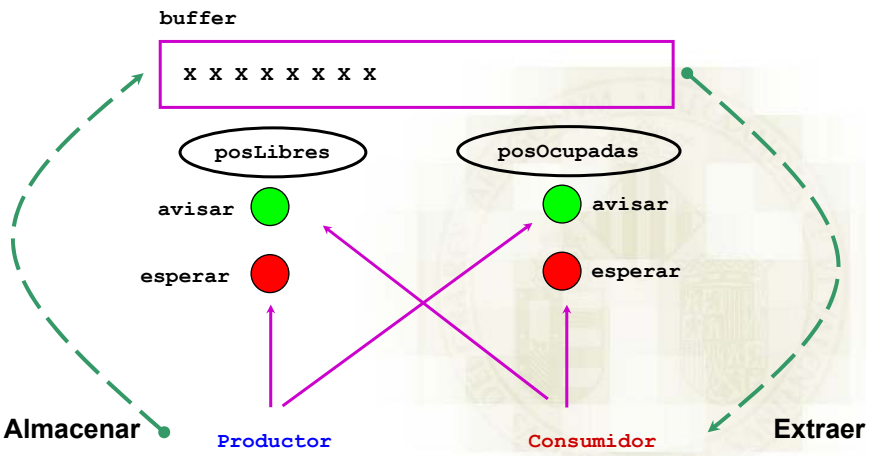
aviso de que hay una nueva posición ocupada.
Si había tareas esperando datos, una tarea de la cola podrá continuar.

```
Tarea Consumidor:
  while ()
  {
    posOcupadas.esperar();
    valor = buffer.extraer ();
    posLibres.avisar();
    //utilizar valor...
  }
Fin_tarea
```

¿hay disponibilidad de datos?
SI: la tarea continúa.
NO: el Consumidor se guarda en la cola

aviso de que hay una nueva posición libre.
Si había tareas esperando guardar datos, una tarea de la cola podrá continuar.

Interpretación gráfica



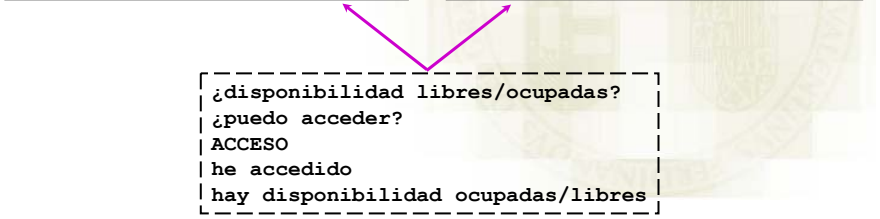
Ejemplo de Sincronización para Competición

- **Controlar el acceso al buffer: sólo una tarea puede estar usándolo.**
- **Un semáforo para controlar el acceso:**
  - ✓ `cont`: no necesita contar nada, sólo debe indicar si se está usando el buffer (0) o no (1) → **Semáforo Binario**.
  - ✓ `esperar`: da permiso para acceder sólo si `cont == 1`
- **Podemos añadir un nuevo semáforo al ejemplo anterior para controlar el acceso exclusivo al buffer (`acceso`).**

Revisión de las tareas

```
Tarea Productor:
   while ()
   {
      //generar valor...
      posLibres.esperar();
      acceso.esperar();
      buffer.almacenar (valor);
      acceso.avisar();
      posOcupadas.avisar();
   }
Fin_tarea
```

```
Tarea Consumidor:
   while ()
   {
      posOcupadas.esperar();
      acceso.esperar();
      valor = buffer.extraer ();
      acceso.avisar();
      posLibres.avisar();
      //utilizar valor...
   }
Fin_tarea
```



Semáforos (fin)

- **Problema: Los semáforos son en sí mismos un recurso**
  - ✓ Varias tareas están solicitando operaciones `esperar/avisar` al semáforo → ¿Ponemos un semáforo al semáforo?
- **Solución: las tareas de un semáforo no se pueden interrumpir.**
- **PL/I: Concurrencia con semáforos binarios.**
- **ALGOL 68: Concurrencia con semáforos.**

Monitores

- Hoare, 1974.
- Monitor = Encapsula datos + mecanismos de sincronización.
- Solución de más alto nivel que los semáforos.
- Si el problema es sincronizar el acceso de tareas a datos comunes → que sean los propios datos los que se encarguen del sincronismo.
- Los lenguajes que soportan monitores aseguran el uso excluyente del monitor:
  - ✓ No hay que controlar la sincronización para Competición.
- Concurrent Pascal y Modula incorporan monitores como tipo.

Estructura de un monitor

```
class bufferMonitor {
   private buffer;
   //incluir todas las declaraciones para implementar
   //el buffer (array, lista circular, etc...)

   //incluir estructuras para sincronizar la
   //Cooperación de tareas.
   private ColaTareas leer, escribir;

   public void almacenar ( valor x ) {...}

   public valor extraer () {...}
}
```

El acceso (Competición) lo controla el compilador.

La implementación debe de incluir la sincronización.

```
Tarea Productor:
  while ()
  {
     //generar valor...
     bufferMonitor.almacenar (valor);
  }
Fin_tarea
```

```
Tarea Consumidor:
  while ()
  {
     valor = bufferMonitor.extraer ();
     //utilizar valor...
  }
Fin_tarea
```

Almacenar/Extraer

```
void almacenar ( valor x ) {
     if ( buffer.esLleno() )
       escribir.delay();
     buffer.insertar (x);
     leer.continue();
}
```

Se bloquea (y encola) la tarea que usa `almacenar` → se le quita la exclusividad de acceso hasta que pueda continuar.

Se continúa con alguna tarea que esté esperando leer.

```
void extraer ( valor x ) {
     if ( buffer.esVacio() )
       leer.delay();
     x = buffer.eliminar ();
     escribir.continue();
}
```


delay/continue

- **`delay`:**
  - ✓ La tarea que ejecuta `delay` se bloquea y se añade a la cola.
  - ✓ Se le quita el acceso exclusivo al monitor
  - ✓ El monitor está disponible para otra tarea
- **`continue`:**
  - ✓ Desconecta del monitor a la tarea que lo ejecuta.
  - ✓ El monitor está disponible para otra tarea.
  - ✓ Si la cola contiene alguna tarea, extrae una tarea y reanuda su ejecución (que había sido detenida por un `delay`).