Clase 4 - Semáforos

# **Defectos de la sincronización por Busy Waiting**

* Protocolos “busy-waiting”: complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados.
* Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.
* Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso spinning puede ser usado de manera más productiva por otro proceso.

| ***⇒Necesidad de herramientas para diseñar protocolos de sincronización.*** |
| --- |

| **Semáforo ⇒** instancia de un tipo de datos abstracto (o un objeto) con sólo 2  operaciones (métodos) atómicas: **P** y **V**.  Internamente el valor de un semáforo es un entero no negativo:  • **V** → Señala la ocurrencia de un evento (incrementa).  • **P** → Se usa para demorar un proceso hasta que ocurra un evento (decrementa). |
| --- |

* Analogía con la sincronización del tránsito para evitar colisiones.
* Permiten proteger Secciones Críticas y pueden usarse para implementar Sincronización por Condición.

En la práctica utilizaremos → **Semáforo general** (o counting semaphore)

**P(s)**: 〈 await (s > 0) s = s-1; 〉

**V(s)**: 〈 s = s+1; 〉

# **Problemas básicos y técnicas**

### Barreras: señalización de eventos

* Idea: un semáforo para cada flag de sincronización. Un proceso setea el flag

ejecutando **V**, y espera a que un flag sea seteado y luego lo limpia ejecutando **P**.

* **Barrera para dos procesos**: necesitamos saber cada vez que un proceso llega o

parte de la barrera ⇒ relacionar los estados de los dos procesos.

| **Semáforo de señalización** ⇒ generalmente inicializado en 0. Un proceso señala el  evento con V(s); otros procesos esperan la ocurrencia del evento ejecutando P(s). |
| --- |

Puede usarse la barrera para dos procesos para implementar una butterfly barrier para n, o sincronización con un coordinador central.

### Productores y Consumidores: semáforos binarios divididos

| **Semáforo Binario Dividido** (Split Binary Semaphore). Los semáforos binarios b1, ...., bn forman un SBS en un programa si el siguiente es un invariante global:  SPLIT: 0 ≤ b1, + ... + bn ≤ 1 |
| --- |

* Los **bi** pueden verse como un único semáforo binario **b** que fue dividido en **n** semáforos binarios.
* Importantes por la forma en que pueden usarse para implementar EM (en general la ejecución de los procesos inicia con un **P** sobre un semáforo y termina con un **V** sobre otro de ellos).
* Las sentencias entre el **P** y el **V** ejecutan con exclusión mutua.

### Buffers Limitados: Contadores de Recursos (Ver ejemplo a partir de diapositiva 11)

| **Contadores de Recursos**: cada semáforo cuenta el número de unidades libres de un recurso determinado. Esta forma de utilización es adecuada cuando los procesos compiten por recursos de múltiples unidades. |
| --- |

### Problema de los filósofos: exclusión mutua selectiva

* Problema de varios procesos (P) y varios recursos (R) cada uno protegido por un lock.
* Un proceso debe adquirir los locks de todos los recursos que necesita.
* Puede caerse en deadlock cuando varios procesos compiten por conjuntos superpuestos de recursos.
* Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de recursos compartidas.
* Problema de los filósofos:

**process Filósofo [i = 0 to 4]**{

while (true){

**adquiere tenedores**;

come;

**libera tenedores;**

piensa;

}

}

* **Cada tenedor es una Sección Crítica**: puede ser tomado por un único filósofo a la vez ⇒ pueden representarse los tenedores por un arreglo de semáforos.
* Levantar un tenedor ⇒ **P**
* Bajar un tenedor ⇒ **V**
* Cada filósofo necesita el tenedor izquierdo y el derecho.

### Lectores y escritores: exclusión mutua selectiva

**Problema**: dos clases de procesos (lectores y escritores) comparten una Base de Datos. El acceso de los escritores debe ser exclusivo para evitar interferencia entre transacciones. Los lectores pueden ejecutar concurrentemente entre ellos si no hay escritores actualizando.

* Procesos asimétricos y, según el scheduler, con diferente prioridad.
* Es también un problema de **exclusión mutua selectiva**: clases de procesos compiten por el acceso a la BD.
* Diferentes soluciones:
* **Como problema de exclusión mutua**

Los escritores necesitan acceso mutuamente exclusivo.Los lectores (como

grupo) necesitan acceso exclusivo con respecto a cualquier escritor. No hay concurrencia entre lectores

(Ver diapositivas 17-20)

* **Como problema de sincronización por condición**

Solución anterior ⇒ preferencia a los lectores ⇒ no es **fair**.

\*Otro enfoque ⇒ introduce la técnica passing the baton: emplea SBS para

brindar exclusión y despertar procesos demorados.

\*Puede usarse para implementar await arbitrarios, controlando de forma

precisa el orden en que los procesos son despertados

### Técnica Passing the Baton

* En algunos casos, await puede ser implementada directamente usando semáforos u otras operaciones primitivas. Pero no siempre…
* En el caso de las guardas de los await en la solución anterior, se superponen en que el protocolo de E/ para escritores necesita que tanto nw como nr sean 0, mientras para lectores sólo que nw sea 0.
* Ningún semáforo podría discriminar entre estas condiciones → **Passing the baton**.

| **Passing the baton**: técnica general para implementar sentencias **await**.  Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar.  Cuando el proceso llega a un **SIGNAL** (sale de la SC), pasa el baton  (control) a otro proceso. Si ningún proceso está esperando por el baton (es decir esperando entrar a la SC) el baton se libera para que lo tome el próximo proceso que trata de entrar. |
| --- |

| La sincronización se expresa con sentencias atómicas de la forma:  **F1 : 〈 Si 〉** o **F2 : 〈 await (Bj ) Sj 〉**  Puede hacerse con semáforos binarios divididos (SBS). |
| --- |

| **e** semáforo binario inicialmente **1** (controla la entrada a sentencias atómicas).  Utilizamos un semáforo **bj** y un contador **dj** cada uno con guarda diferente **B**j ; todos inicialmente 0. |
| --- |

| **bj** se usa para demorar procesos esperando que Bj sea true.  **dj** es un contador del número de procesos demorados sobre bj  **e** y los **bj** se usan para formar un **SBS**: a lo sumo uno a la vez es **1**, y cada camino de ejecución empieza con un **P** y termina con un único **V.** |
| --- |

# Alocación de Recursos y Scheduling

| **Problema**: decidir cuándo se le puede dar a un proceso determinado acceso a un  recurso. Implementar políticas de alocación de recursos generales controlando  explícitamente cuál proceso toma un recurso si hay más de uno esperando.  **Recurso**: cualquier objeto, elemento, componente, dato, SC, por la que un proceso  puede ser demorado esperando adquirirlo.  **Definición del problema**: procesos que compiten por el uso de unidades de un  recurso compartido (cada unidad está libre o en uso).  request (parámetros): 〈await (request puede ser satisfecho) tomar unidades;〉  release (parámetros): 〈retornar unidades;〉 |
| --- |

### 

### Alocación Shortest-Job-Next (SJN)

* Varios procesos que compiten por el uso de un recurso compartido de una sola **unidad**.
* **request (tiempo,id)**. Si el recurso está libre, es alocado inmediatamente al proceso id; sino, el proceso id se demora.
* **release ( )**. Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) con el mínimo valor de tiempo. Si dos o más procesos tienen el mismo valor de tiempo, el recurso es alocado al que esperó más.
* SJN minimiza el tiempo promedio de ejecución, aunque no es fair (¿por qué?). Puede mejorarse con la técnica de **aging** (dando preferencia a un proceso que esperó mucho tiempo).
* Para el caso general de alocación de recursos (NO SJN):

bool libre = true;

request (tiempo,id): 〈await (libre) libre = false;〉

release (): 〈libre = true;〉

En SJN, un proceso que invoca a request debe demorarse hasta que el recurso esté libre y su pedido sea el próximo en ser atendido de acuerdo a la política. El parámetro tiempo entra en juego sólo si un pedido debe ser demorado.

En **DELAY** un proceso:

* Inserta sus parámetros en un conjunto, cola o lista de espera (pares).
* Libera la SC ejecutando V(e).
* Se demora en un semáforo hasta que request puede ser satisfecho.

En **SIGNAL** un proceso:

* Cuando el recurso es liberado, si pares no está vacío, el recurso es asignado a un proceso de acuerdo a SJN.

Cada proceso tiene una condición de demora distinta, dependiendo de su posición en **pares**. El proceso **id** se demora sobre el semáforo **b[id].**

| **s** es un **semáforo privado** si exactamente un proceso ejecuta operaciones **P** sobre **s**.  Resultan útiles para señalar procesos individuales. Los semáforos **b[id]** son de este  tipo. |
| --- |

# Casos problema

### Productores/Consumidores con broadcast

| En el problema del buffer atómico, sea **UN** proceso **productor** y **N** procesos **consumidores**.  El Productor **DEPOSITA** y debe esperar que **TODOS** los consumidores consuman el mismo mensaje (broadcast).  Notar la diferencia entre una solución por memoria compartida y otra por pasaje de mensajes. |
| --- |