

Programación Concurrente

Clase 4



Facultad de Informática
UNLP

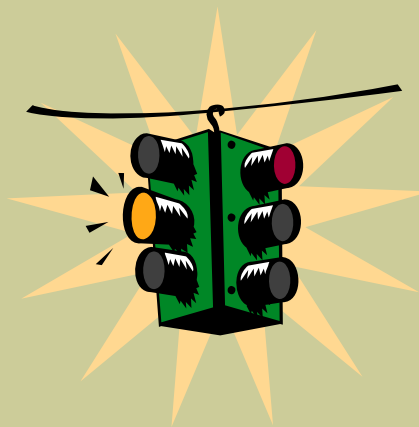
Links a los archivos con audio (formato MP4)

El archivo con la clase con audio está en formato MP4. En el link de abajo está el video comprimido en archivo RAR.

- ◆ Semáforos

<https://drive.google.com/uc?id=1MjOhte-Wv6nQh0V42bwiEJ-HlO2m7rEJ&export=download>

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.*

Semáforos

Descriptos en 1968 por Dijkstra
(www.cs.utexas.edu/users/EWD/welcome.html)

Semáforo \Rightarrow 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*** \rightarrow Señala la **ocurrencia de un evento** (incrementa).
 - ***P*** \rightarrow 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*.

Operaciones Básicas

- **Declaraciones**

sem s; → NO. Si o si se deben inicializar en la declaración

sem mutex = 1;

sem fork[5] = ([5] 1);

- **Semáforo general (o *counting semaphore*)**

$P(s): \langle \text{await } (s > 0) \ s = s-1; \rangle$

$V(s): \langle s = s+1; \rangle$

- **Semáforo binario**

$P(b): \langle \text{await } (b > 0) \ b = b-1; \rangle$

$V(b): \langle \text{await } (b < 1) \ b = b+1; \rangle$

Si la implementación de la demora por operaciones P se produce sobre una *cola*, las operaciones son *fair*

(EN LA MATERIA NO SE PUEDE SUPONER ESTE TIPO DE IMPLEMENTACIÓN)

Problemas básicos y técnicas

Sección Crítica: *Exclusión Mutua*

```
bool lock=false;

process SC[i=1 to n]
{ while (true)
  { <await (not lock) lock = true;>
    sección crítica;
    lock = false;
    sección no crítica;
  }
}
```

Cambio de
variable



```
bool free = true;

process SC[i=1 to n]
{ while (true)
  { <await (free) free = false;>
    sección crítica;
    free = true;
    sección no crítica;
  }
}
```

Podemos representar *free* con un entero, usar 1 para *true* y 0 para *false* \Rightarrow se puede asociar a las operaciones soportadas por los semáforos.

```
int free = 1;

process SC[i=1 to n]
{ while (true)
  { <await (free==1) free = 0;>
    sección crítica;
    free = 1;
    sección no crítica;
  }
}
```

Problemas básicos y técnicas

Sección Crítica: *Exclusión Mutua*

```
int free = 1;

process SC[i=1 to n]
{ while (true)
  { <await (free==1) free = 0;>
    sección crítica;
    free = 1;
    sección no crítica;
  }
}
```



```
int free = 1;

process SC[i=1 to n]
{ while (true)
  { <await (free > 0) free = free - 1;>
    sección crítica;
    <free = free + 1>;
    sección no crítica;
  }
}
```

```
sem free= 1;
process SC[i=1 to n]
{ while (true)
  { P(free);
    sección crítica;
    V(free);
    sección no crítica;
  }
}
```

Es más simple que las soluciones *busy waiting*.

¿Y si inicializo free= 0?

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 \Rightarrow *relacionar los estados de los dos procesos*.

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

```
sem llega1=0, llega2=0;
process Worker1
{ .....
  V(llega1); P(llega2);
  .....
}

process Worker2
{ .....
  V(llega2); P(llega1);
  .....
}
```

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

¿Qué sucede si los procesos primero hacen P y luego V?

Problemas básicos y técnicas

Productores y Consumidores: *semáforos binarios divididos*

Semáforo Binario Dividido (Split Binary Semaphore). Los semáforos binarios b_1, \dots, b_n forman un SBS en un programa si el siguiente es un invariante global:

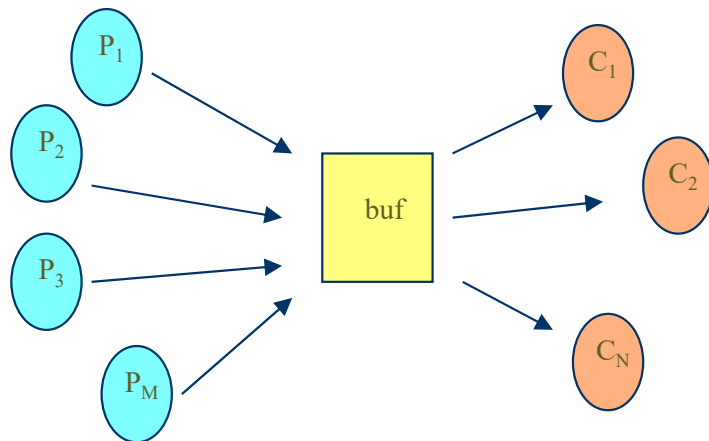
$$SPLIT: 0 \leq b_1 + \dots + b_n \leq 1$$

- Los b_i 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.

Problemas básicos y técnicas

Productores y Consumidores: *semáforos binarios divididos*

Ejemplo: buffer unitario compartido con múltiples productores y consumidores. Dos operaciones: *depositar* y *retirar* que deben alternarse.



```
typeT buf; sem vacio = 1, lleno = 0;

process Productor [i = 1 to M]
{ while(true)
  { ...
    producir mensaje datos
    P(vacio); buf = datos; V(lleno); #depositar
  }
}

process Consumidor[j = 1 to N]
{ while(true)
  { P(lleno); resultado = buf; V(vacio); #retirar
    consumir mensaje resultado
    ...
  }
}
```

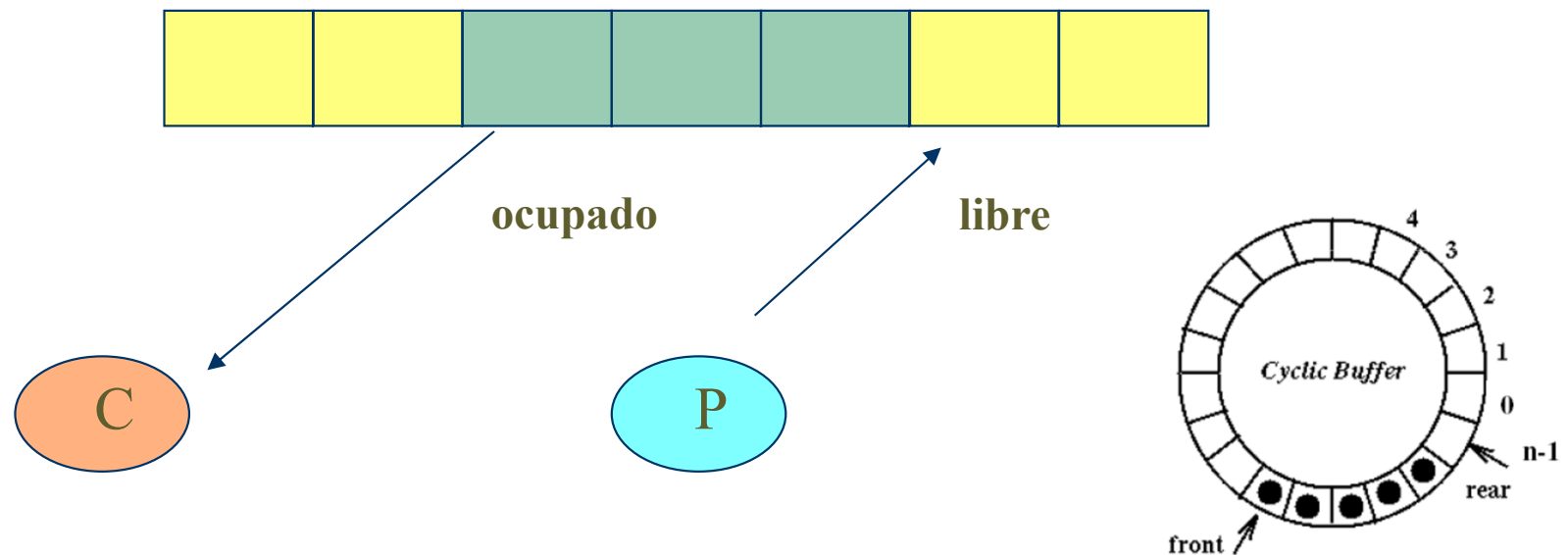
vacio y *lleno* (juntos) forman un “*semáforo binario dividido*”.

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

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*.

Ejemplo: un buffer es una cola de mensajes depositados y aún no buscados. Existe UN productor y UN consumidor que *depositan* y *retiran* elementos del buffer.



Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

```
typeT buf[n]; int ocupado = 0, libre = 0;
sem vacio = n, lleno = 0;

process Productor
{ while(true)
  { ...
    producir mensaje datos
    P(vacio); buf[libre] = datos; libre = (libre+1) mod n; V(lleno); #depositar
  }
}

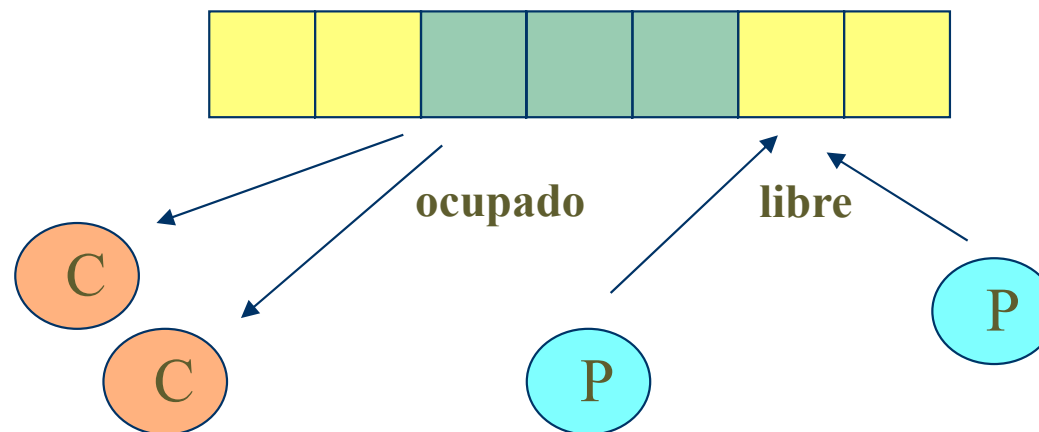
process Consumidor
{ while(true)
  { P(lleno); resultado = buf[ocupado]; ocupado = (ocupado+1) mod n; V(vacio); #retirar
    consumir mensaje resultado
    ...
  }
}
```

- ***vacio*** cuenta los lugares libres, y ***lleno*** los ocupados.
- ***depositar*** y ***retirar*** se pudieron asumir atómicas pues sólo hay un productor y un consumidor.
- ¿Qué ocurre si hay más de un productor y/o consumidor?

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

Si hay más de un productor y/o más de un consumidor, las operaciones de depositar y retirar en sí mismas son SC y deben ejecutar con Exclusión Mutua ¿Cuáles serían las consecuencias de no protegerlas?



Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos al sobrescribirlo.

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

```
typeT buf[n]; int ocupado = 0, libre = 0;
```

```
sem vacio = n, lleno = 0;
```

```
sem mutexD = 1, mutexR = 1;
```

```
process Productor [i = 1..M]
```

```
{ while(true)
```

```
  { producir mensaje datos
```

```
    P(vacio);
```

```
    P(mutexD); buf[libre] = datos; libre = (libre+1) mod n; V(mutexD);
```

```
    V(lleno);
```

```
  }
```

```
}
```

```
process Consumidor [i = 1..N]
```

```
{ while(true)
```

```
  { P(lleno);
```

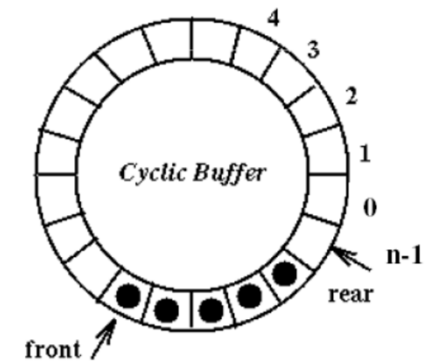
```
    P(mutexR); resultado = buf[ocupado]; ocupado = (ocupado+1) mod n; V(mutexR);
```

```
    V(vacio);
```

```
    consumir mensaje resultado
```

```
  }
```

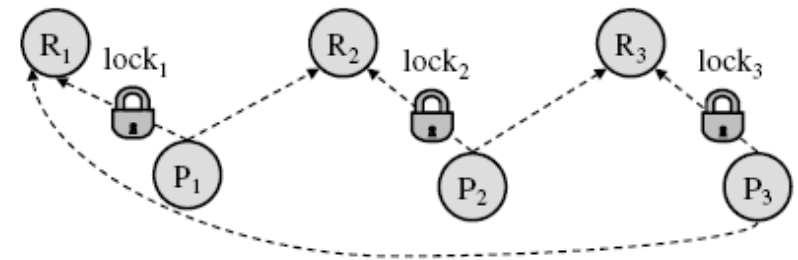
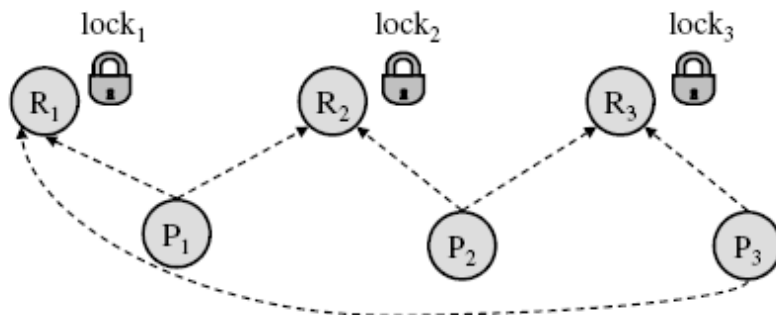
```
}
```



Problemas básicos y técnicas

Varios procesos compitiendo por varios recursos compartidos

- 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.
- Por ejemplo: cada $P[i]$ necesita $R[i]$ y $R[(i+1) \bmod n] \Rightarrow$ ¿Cuándo se da el *Deadlock*?



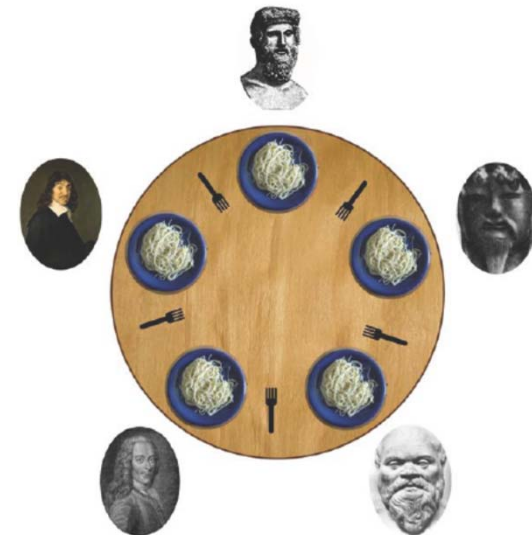
Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

- Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas.

- *Problema de los filósofos:*

```
process Filosofo [i = 0 to 4]
{ while (true)
  { adquiere tenedores;
    come;
    libera tenedores;
    piensa;
  }
}
```



- *Cada tenedor es una SC*: puede ser tomado por un único filósofo a la vez \Rightarrow pueden representarse los tenedores por un arreglo de semáforos.
- Levantar un tenedor \Rightarrow **P** Bajar un tenedor \Rightarrow **V**
- Cada filósofo necesita el tenedor izquierdo y el derecho.
- ¿Qué efecto puede darse si todos los filósofos hacen *exactamente* lo mismo?.

Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

```
sem tenedores [5] = {1,1,1,1,1};

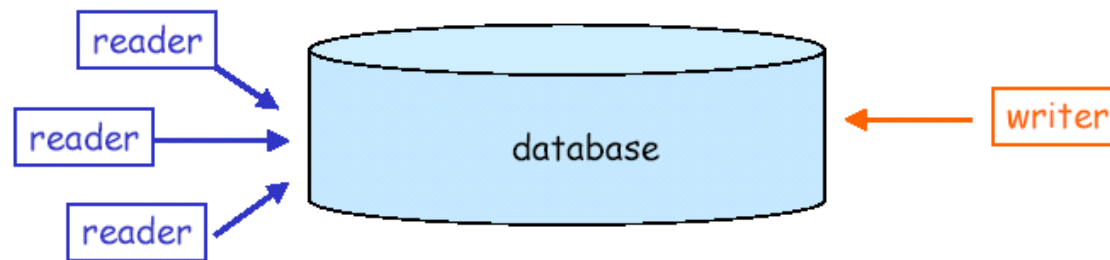
process Filososfos[i = 0..3]
{ while(true)
  { P(tenedor[i]); P(tenedor[i+1]);
    comer;
    V(tenedor[i]); V(tenedor[i+1]);
  }
}

process Filososfos[4]
{ while(true)
  { P(tenedor[0]); P(tenedor[4]);
    comer;
    V(tenedor[0]); V(tenedor[4]);
  }
}
```

Problemas básicos y técnicas

Lectores y escritores

- **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.
 - Como problema de sincronización por condición.

Problemas básicos y técnicas

Lectores y escritores: *como problema de exclusión mutua*

- Los escritores necesitan acceso mutuamente exclusivo.
- Los lectores (como grupo) necesitan acceso exclusivo con respecto a cualquier escritor.

```
sem rw = 1;
process Lector [i = 1 to M]
{ while(true)
  { ...
    P(rw);
    lee la BD;
    V(rw);
  }
}
process Escritor [j = 1 to N]
{ while(true)
  { ...
    P(rw);
    escribe la BD;
    V(rw);
  }
}
```

No hay concurrencia entre lectores

- Los lectores (como grupo) necesitan bloquear a los escritores, pero sólo el primero necesita tomar el *lock* ejecutando $P(rw)$.
- Análogamente, sólo el último lector debe hacer $V(rw)$.

Problemas básicos y técnicas

Lectores y escritores: *como problema de exclusión mutua*

```
int nr = 0;      # número de lectores activos
sem rw = 1;      # bloquea el acceso a la BD
```

```
process Escritor [j = 1 to N]
{ while(true)
  { ...
    P(rw);
    escribe la BD;
    V(rw);
  }
}
```

```
process Lector [i = 1 to M]
{ while(true)
  { ...
    < nr = nr + 1; if (nr == 1) P(rw); >
    lee la BD;
    < nr = nr - 1; if (nr == 0) V(rw); >
  }
}
```

Problemas básicos y técnicas

Lectores y escritores: *como problema de exclusión mutua*

```
int nr = 0;           # número de lectores activos
sem rw = 1;           # bloquea el acceso a la BD
sem mutexR = 1;       # bloquea el acceso de los lectores a nr
```

```
process Escritor [j = 1 to N]
{ while(true)
  { ...
    P(rw);
    escribe la BD;
    V(rw);
  }
}
```

```
process Lector [i = 1 to M]
{ while(true)
  { ...
    P(mutexR);
    nr = nr + 1;
    if (nr == 1) P(rw);
    V(mutexR);
    lee la BD;
    P(mutexR);
    nr = nr - 1;
    if (nr == 0) V(rw);
    V(mutexR);
  }
}
```

Problemas básicos y técnicas

Lectores y escritores: *sincronización por condición*

- Solución anterior \Rightarrow preferencia a los lectores \Rightarrow no es *fair*.
- Otro enfoque \Rightarrow 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
- En este caso, pueden contarse (por medio de *nr* y *nw*) los procesos de cada clase intentando acceder a la BD, y luego restringir el valor de los contadores. ¿Cuáles son los estados buenos y malos de *nr* y *nw*?

```
int nr = 0, nw = 0;
process Lector [i = 1 to M]
{ while(true)
  { ...
    < await (nw == 0) nr = nr + 1; >
    lee la BD;
    < nr = nr - 1; >
  }
}
```

```
process Escritor [j = 1 to N]
{ while(true)
  { ...
    < await (nr==0 and nw==0) nw=nw+1; >
    escribe la BD;
    < nw = nw - 1; >
  }
}
```

Problemas básicos y técnicas

Técnica Passing de 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 entrada 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.

Problemas básicos y técnicas

Técnica Passing de Baton

La sincronización se expresa con sentencias atómicas de la forma:

$$F_1 : \langle S_i \rangle \quad \text{o} \quad F_2 : \langle \text{await } (B_j) S_j \rangle$$

Puede hacerse con semáforos binarios divididos (SBS).

e semáforo binario inicialmente 1 (controla la entrada a sentencias atómicas).

Utilizamos un semáforo b_j y un contador d_j cada uno con guarda diferente B_j ; todos inicialmente 0 .

b_j se usa para demorar procesos esperando que B_j sea *true*.

d_j es un contador del número de procesos demorados sobre b_j .

e y los b_j 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 .

Problemas básicos y técnicas

Técnica Passing de Baton

$F_1 :$ P(e);
S_i;
SIGNAL;

⟨ S_i ⟩

$F_2 :$ P(e);
if (not B_j) {d_j = d_j + 1; V(e); P(b_j); }
S_j;
SIGNAL

⟨ await (B_j) S_j ⟩

SIGNAL: if (B₁ and d₁ > 0) {d₁ = d₁ - 1; V(b₁)}
□ ...
□ (B_n and d_n > 0) {d_n = d_n - 1; V(b_n)}
□ else V(e);
fi

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing de Baton*

```
int nr = 0, nw = 0;

process Lector [i = 1 to M]
{ while(true)
  { ...
    < await (nw == 0) nr = nr + 1; >
    lee la BD;
    < nr = nr - 1; >
  }
}

process Escritor [j = 1 to N]
{ while(true)
  { ...
    < await (nr==0 and nw==0) nw=nw+1; >
    escribe la BD;
    < nw = nw - 1; >
  }
}
```

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M]
{ while(true) {
  P(e);
  if (nw > 0) {dr = dr+1; V(e); P(r); }
  nr = nr + 1;
  SIGNAL1;
  lee la BD;
  P(e); nr = nr - 1; SIGNAL2;
}
}

process Escritor [j = 1 to N]
{ while(true) {
  P(e);
  if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
  nw = nw + 1;
  SIGNAL3;
  escribe la BD;
  P(e); nw = nw - 1; SIGNAL4;
}
}
```

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing de Baton*

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M]
{ while(true)
  { P(e);
    if (nw > 0){dr = dr+1; V(e); P(r); }
    nr = nr + 1;
    SIGNAL1 ;
    lee la BD;
    P(e); nr = nr - 1; SIGNAL2 ;
  }
}

process Escritor [j = 1 to N]
{ while(true)
  { P(e);
    if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
    nw = nw + 1;
    SIGNAL3 ;
    escribe la BD;
    P(e); nw = nw - 1; SIGNAL4 ;
  }
}
```

El rol de los **SIGNAL_i** es el de señalar *exactamente* a uno de los semáforos \Rightarrow los procesos se van pasando el *baton*.

SIGNAL_i es una abreviación de:

```
if (nw == 0 and dr > 0)
  {dr = dr - 1; V(r);}
elsif (nr == 0 and nw == 0 and dw > 0)
  {dw = dw - 1; V(w);}
else V(e);
```

Algunos de los SIGNAL se pueden simplificar.

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing de Baton*

```
int nr = 0, nw = 0, dr = 0, dw = 0;
```

```
sem e = 1, r = 0, w = 0;
```

```
process Lector [i = 1 to M]
{ while(true)
  { P(e);
    if (nw > 0) {dr = dr+1; V(e); P(r); }
    nr = nr + 1;
    if (dr > 0) {dr = dr - 1; V(r); }
    else V(e);
    lee la BD;
    P(e);
    nr = nr - 1;
    if (nr == 0 and dw > 0)
      {dw = dw - 1; V(w); }
    else V(e);
  }
}
```

```
process Escritor [j = 1 to N]
{ while(true)
  { P(e);
    if (nr > 0 or nw > 0)
      {dw=dw+1; V(e); P(w);}
    nw = nw + 1;
    V(e);
    escribe la BD;
    P(e);
    nw = nw - 1;
    if (dr > 0) {dr = dr - 1; V(r); }
    elseif (dw > 0) {dw = dw - 1; V(w); }
    else V(e);
  }
}
```

Da preferencia a los lectores \Rightarrow ¿Cómo puede modificarse?

Alocación de Recursos y Scheduling

Problema: decidir cuándo se le puede dar a un proceso determinado acceso a un recurso.

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;>

- Puede usarse Passing the Baton:

request (parámetros): P(e);
if (request no puede ser satisfecho) DELAY;
tomar las unidades;
SIGNAL;

release (parámetros): P(e);
retornar unidades;
SIGNAL;

Alocación de Recursos y Scheduling

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 asignado inmediatamente al proceso *id*; sino, el proceso *id* se demora.
- **release** (). Cuando el recurso es liberado, es asignado 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 asignado al que esperó más.
- SJN minimiza el tiempo promedio de ejecución, aunque *es unfair* (¿por qué?). Puede mejorarse con la técnica de *aging* (dando preferencia a un proceso que esperó mucho tiempo).
- Para el caso general de asignación de recursos (NO SJN):
 - bool libre = true;
 - request** (tiempo,id): $\langle \text{await (libre) libre} = \text{false}; \rangle$
 - release** (): $\langle \text{libre} = \text{true}; \rangle$

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

- 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.

```
request (tiempo, id):  
    P(e);  
    if (not libre) DELAY;  
    libre = false;  
    SIGNAL;  
  
release ( ):  
    P(e);  
    libre = true;  
    SIGNAL;
```

- 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]*.

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

```
bool libre = true; Pares = set of (int, int) =  $\emptyset$ ; sem e = 1, b[n] = ([n] 0);
```

```
request(tiempo,id):  P(e);  
                      if (! libre){ insertar (tiempo, id) en Pares; V(e); P(b[id]); }  
                      libre = false;  
                      V(e);  
  
    release( ):      P(e);  
                      libre = true;  
                      if (Pares  $\neq \emptyset$  ) { remover el primer par (tiempo,id) de Pares; V(b[id]); }  
                      else V(e);
```

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.

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

```
bool libre = true; Pares = set of (int, int) =  $\emptyset$ ; sem e = 1, b[n] = ([n] 0);
```

Process Cliente [id: 1..n]

```
{ int sig;
```

```
  //Trabaja
```

```
  tiempo = //determina el tiempo de uso del recurso//
```

```
  P(e);
```

```
  if (! libre) { insertar (tiempo, id) en Pares;
```

```
                V(e);
```

```
                P(b[id]);
```

```
            }
```

```
  libre = false;
```

```
  V(e);
```

```
  //USA EL RECURSO
```

```
  P(e);
```

```
  libre = true;
```

```
  if (Pares  $\neq \emptyset$ ) { remover el primer par (tiempo, sig) de Pares;
```

```
                    V(b[sig]);
```

```
                }
```

```
  else V(e);
```

```
}
```

¿Que modificaciones deberían realizarse para respetar el orden de llegada?

¿Que modificaciones deberían realizarse para generalizar la solución a recursos de múltiple unidad?