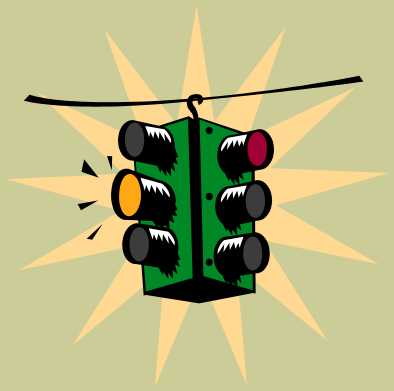
Programación Concurrente ATIC
Programación Concurrente (redictado)

Clase 4

Facultad de Informática

UNLP

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 ⇒ 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*.

Operaciones Básicas

- **Declaraciones**

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

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

  ***P(s)*:** ⟨ await (s > 0) s = s-1; ⟩
  ***V(s)*:** ⟨ s = s+1; ⟩

- **Semáforo binario**

  ***P(b)*:** ⟨ await (b > 0) b = b-1; ⟩
  ***V(b)*:** ⟨ await (b < 1) b = b+1; ⟩

Si la implementación de la demora por operaciones ***P*** se produce sobre una ***cola***, las operaciones son ***fair***
(EN LA PRÁCTICA DE 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*** ⇒ se puede asociar a las operaciones soportadas por los semáforos.

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

Es más simple que las soluciones ***busy waiting***.

¿Y si inicializo mutex = 0?

Problemas básicos y técnicas

Barreras: señalización de eventos

- Recordar la utilización de barreras …
- **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).

```
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, ..., b_n forman un SBS en un programa si el siguiente es un invariante global:

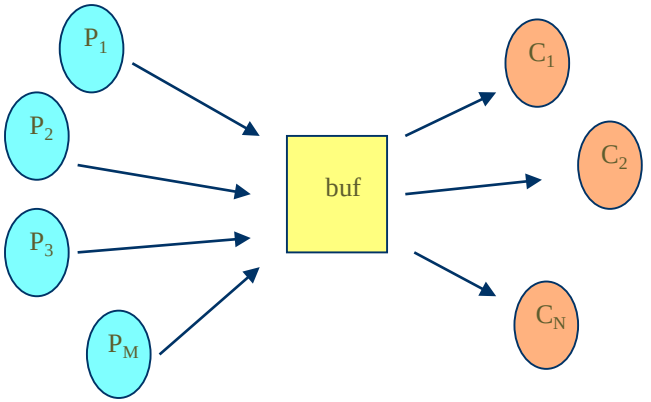
$$SPLIT: 0 \leq b_1, + ... + b_n \leq 1$$

- 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
        …
     }
}
```

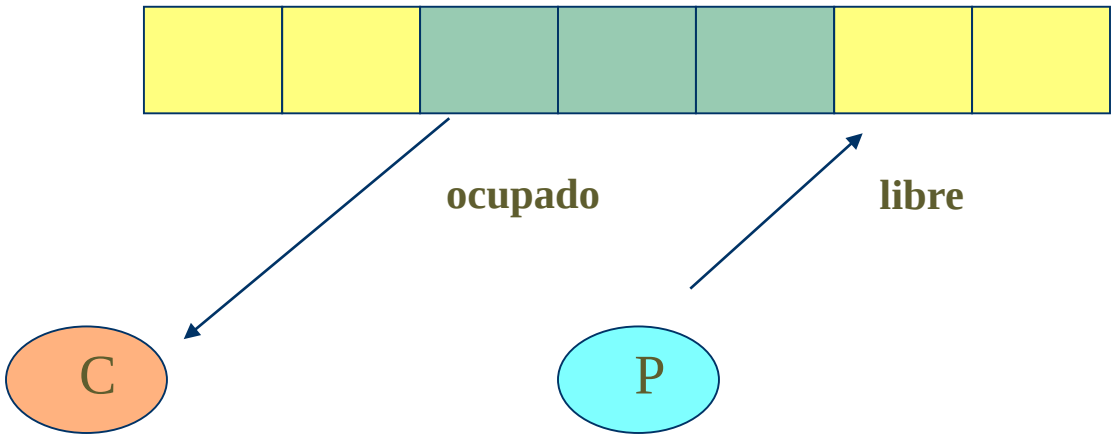
vacio y ***lleno*** (juntos) forman un ***"semáforo binario dividido"***.

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*

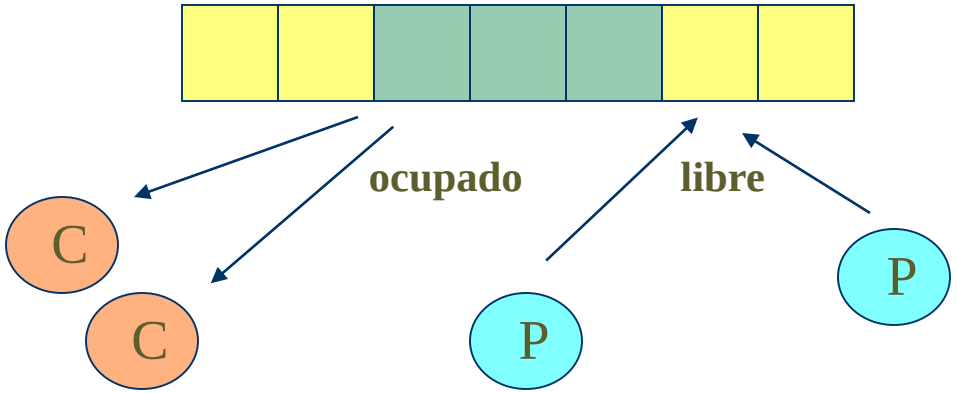
```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;

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?

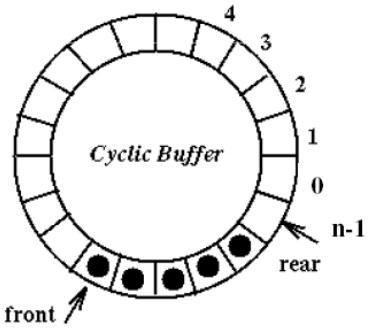


Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos por sobreescritura.

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;

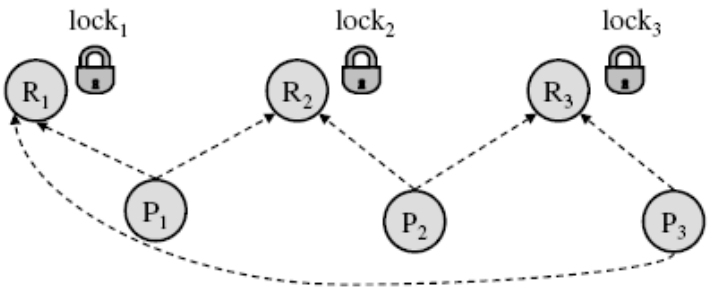


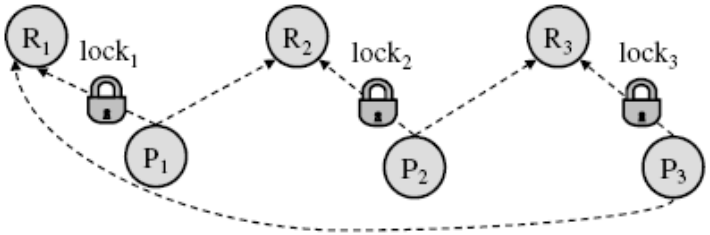

```
process Productor [i = 1..M]
{ while(true)
     {  producir mensaje datos
        P(vacio);
        P(mutexD); buf[libre] = datos; libre = (libre+1) mod n; V(mutexD);
        V(lleno);
     }
}

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

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



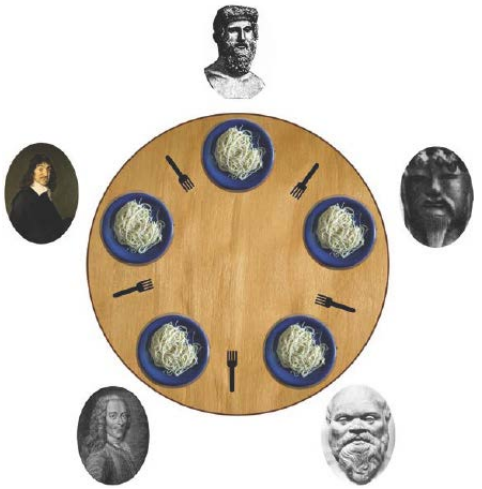


Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

- Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de recursos 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 ⇒ 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.
- ¿Qué efecto puede darse si todos los filósofos hacen ***exactamente*** lo mismo?.

Problemas básicos y técnicas

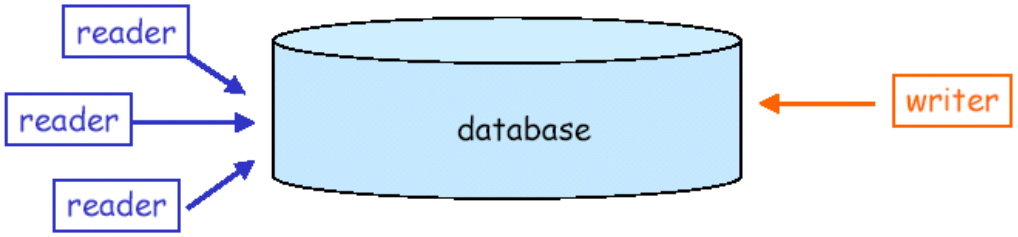
Problema de los filósofos: *exclusión mutua selectiva*

```
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: *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.
  - 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 ⇒ 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
- 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 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.

Problemas básicos y técnicas

Técnica Passing de Baton

La sincronización se expresa con sentencias atómicas de la forma:

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

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;

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

SIGNAL: if (B_1 and $d_1 > 0$) {$d_1 = d_1 - 1$; V(b_1)}
□ ...
□ (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; ⟩
   }
}
```

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

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

SIGNAL:
if (B_1 and $d_1 > 0$) {$d_1 = d_1 - 1$; V(b_1)}
□ ...
□ (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, 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 señalizar ***exactamente*** a uno de los semáforos ⇒ 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 ⇒ ¿Cómo puede modificarse?

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;⟩

- 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 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;⟩

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) = ∅;  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 ≠ ∅ ) { 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) = ∅;
sem e = 1, b[n] = ([n] 0);

Process Cliente[id: 0..N-1]::
{  int tiempo = …;
   …
   //request
   P(e);
   if (! Libre){ insertar (tiempo, id) en Pares; V(e); P(b[id]); }
   libre = false;
   V(e);
   //Usa el recurso compartido
   //reléase
   P(e);
   libre = true;
   if (Pares ≠ ∅ ) { remover el primer par (tiempo,id) de Pares; V(b[id]); }
   else V(e);
   …
}
```

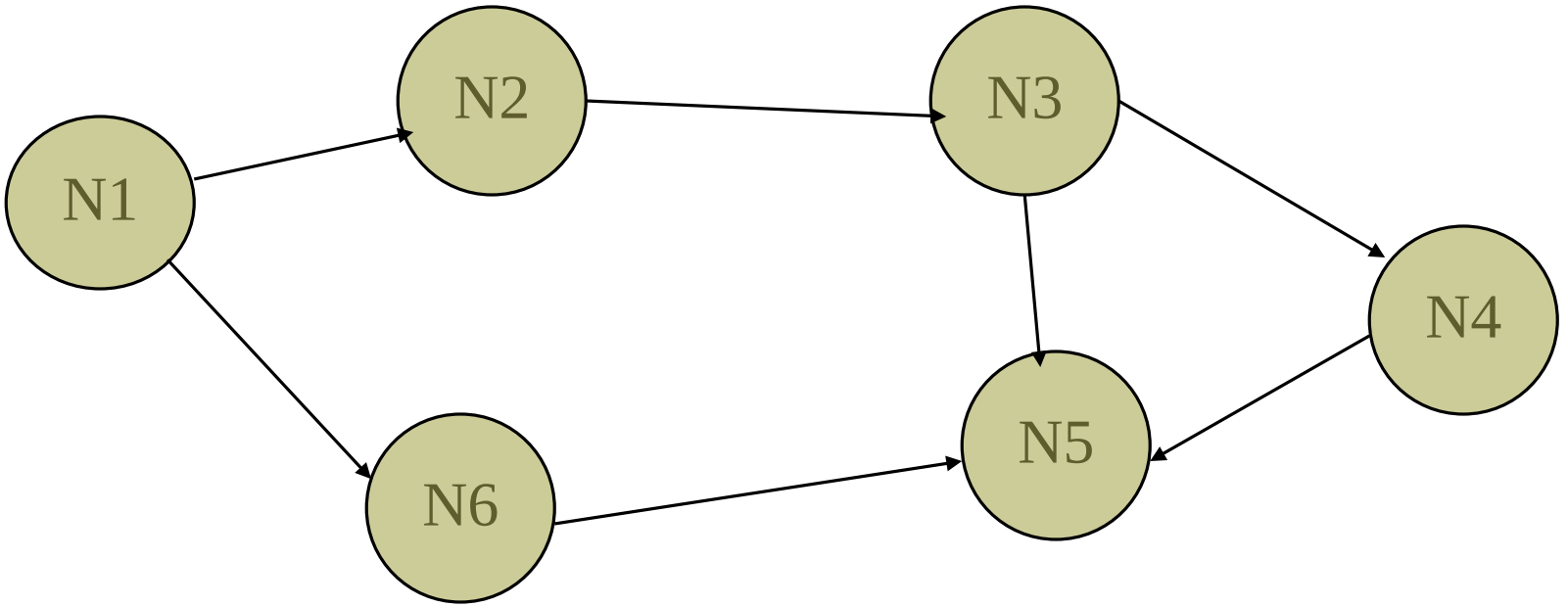
¿Que modificaciones deberían realizarse para generalizar la solución a recursos de múltiple unidad?

Casos Problema

Grafo de precedencia

Analizar el problema de modelizar N procesos que deben sincronizar, de acuerdo a lo especificado por un grafo de precedencia arbitrario (con semáforos).

Ejemplo: en este caso N4 hará P(N3) y V(N5), N1 hará V(N2) y V(N6).



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.

Versión más compleja: buffer con ***K*** lugares, ***UN*** productor y ***N*** consumidores.

El productor puede depositar hasta ***K*** mensajes, los ***N*** consumidores deben leer cada mensaje para que el lugar se libere y el orden de lectura de cada consumidor debe ser ***FIFO***.

Analizar el tema con semáforos.

Variantes del problema de los filósofos

- Si en lugar de administrar tenedores, cada filósofo tiene un estado (comiendo, pensando, con hambre) y debe consultar a sus dos filósofos laterales para saber si puede comer, tendremos una solución distribuida. ¿Se podría usar la técnica de “passing the baton” para resolverlo?.
- Otra alternativa es tener una solución de filósofos centralizada, en la que un scheduler administra por ejemplo los tenedores y los asigna con posiciones fijas o variables (notar la diferencia).
- En la solución que vimos de filósofos, para evitar deadlock utilizamos un código distinto para un filósofo (orden de toma de los tenedores). Otra alternativa es la de la elección al azar del primer tenedor a tratar de tomar... ¿Cómo?

Problema de los baños

- Un baño único para varones o mujeres (excluyente) sin límite de usuarios.
- Un baño único para varones o mujeres (excluyente) con un número máximo de ocupantes simultáneos (que puede ser diferente para varones y mujeres)
- Dos baños utilizables simultáneamente por un número máximo de varones (K1) y de mujeres (K2), con una restricción adicional respecto que el total de usuarios debe ser menor que K3 (K3 < K1 + K2).

Problema de la molécula de agua

- Existen procesos O (oxígeno) y H (hidrógeno) que en un determinado momento toman un estado “listo” y se buscan para combinarse formando una molécula de agua (HH0).
- Puede pensarse en un esquema Cliente/Servidor, donde el servidor recibe los pedidos y cuando tiene 2 H listos y 1 O listo concreta la molécula de agua y libera los procesos H y O.
- También puede pensarse como un esquema “passing the baton” que puede iniciarse por cualquiera de los dos H o por el O, pero tiene un orden determinado (analizar con cuidado).
- ¿Podría pensar la solución con un esquema de productores-consumidores? ¿Quiénes serían los productores y quienes los consumidores?

Puente de una vía

- Suponga un puente de una sola vía que es accedido por sus extremos (procesos Norte y procesos Sur).
- Cómo especificaría la exclusión mutua sobre el puente, de modo que circulen los vehículos (procesos) en una sola dirección.
- Es un caso típico donde es difícil asegurar fairness y no inanición. ¿Por qué? ¿Cuál podría ser un método para asegurar no inanición con un scheduler? ¿Qué ideas propone para tener fairness entre Norte y Sur ?
- Suponga que cruzar el puente requiere a lo sumo 3 minutos y Ud. quiere implementar una solución tipo “time sharing” entre los procesos Norte y Sur, habilitando alternativamente el puente 15 minutos en una y otra dirección. ¿Cómo lo esquematizaría?

Search – Insert – Delete sobre una lista con procesos concurrentes

- Una generalización de la Exclusión Mutua Selectiva entre clases de procesos visto con lectores-escritores es el problema de procesos que acceden a una lista enlazada para ***Buscar*** (search), ***Insertar*** al final o ***Borrar*** un elemento en cualquier posición.
- Los procesos que BORRAN se excluyen entre sí y además excluyen a los procesos que buscan e insertan. Sólo un proceso de borrado puede acceder a la lista.
- Los procesos de inserción se excluyen entre sí, pero pueden coexistir con los de búsqueda. A su vez, los de búsqueda NO se excluyen entre sí

Modificaciones a SJN

- Suponga que se quiere cambiar el esquema de alocación SJN por uno LJN (longest JOB next). Analice el código y comente los cambios. ¿Cuál de los dos esquemas le parece más fair? ¿Cuál generará una cola mayor de procesos pendientes?.

- En el esquema SJN, suponga que lo quiere cambiar por un esquema FIFO. Analice el código y comente los cambios. ¿Cuál de los dos esquemas le parece más eficiente? ¿Más Fair ?.

Un problema para investigar: Drinking Philosophers

- Investigar el tema definido por Chandy y Misra en 1984. Se trata de una generalización del problema de los ***dining philosophers***.

- Analizar las diferencias de los dos problemas, estudiar los mecanismos de sincronización y discutir las ventajas/dificultades de utilizar semáforos en la sincronización.