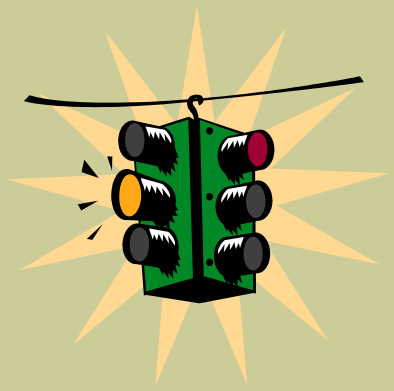
Programación Concurrente

Clase 3

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 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)*:** ⟨ 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 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.

```
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;
   }
}
```

Problemas básicos y técnicas

Sección Crítica: *Exclusión Mutua*

```
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;
   }
}
```

Definición de las operaciones P y V

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

***V(s)*: ⟨ s = s+1; ⟩**

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 ⇒ *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:

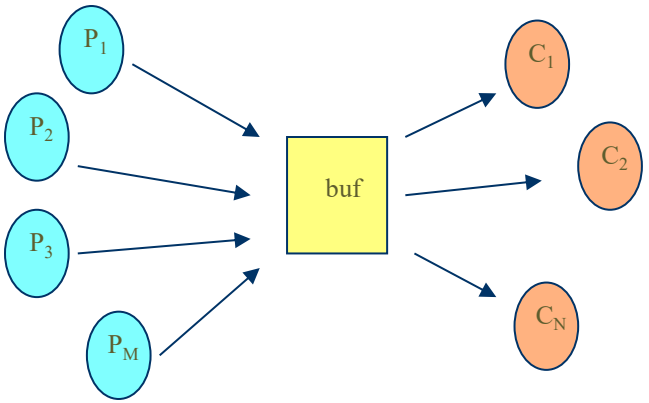
$$SPLIT: 0 \leq b_1, + \ldots + b_n \leq 1$$

- 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
         …
      }
}
```

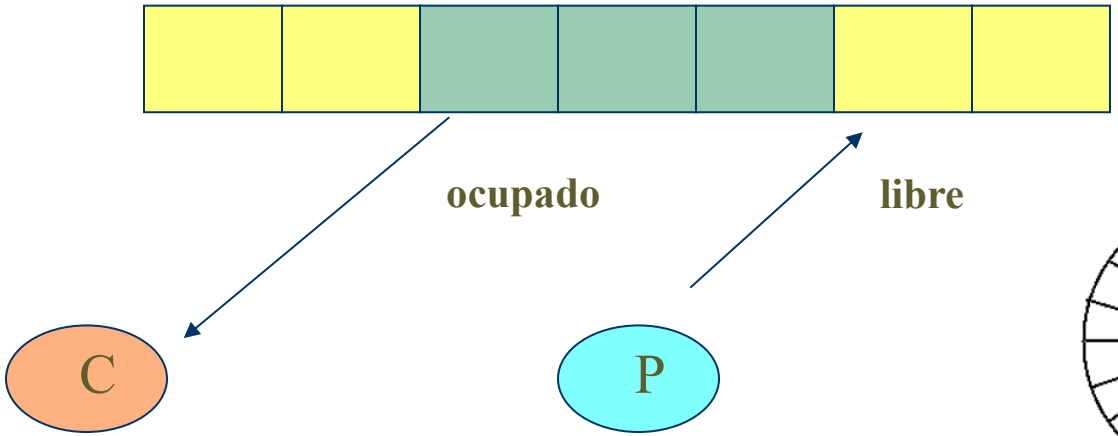
vacio y ***lleno*** (juntos) forman un ***"semáforo binario dividido"***.

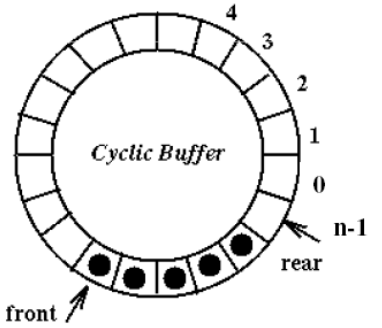
Problemas básicos y técnicas

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*

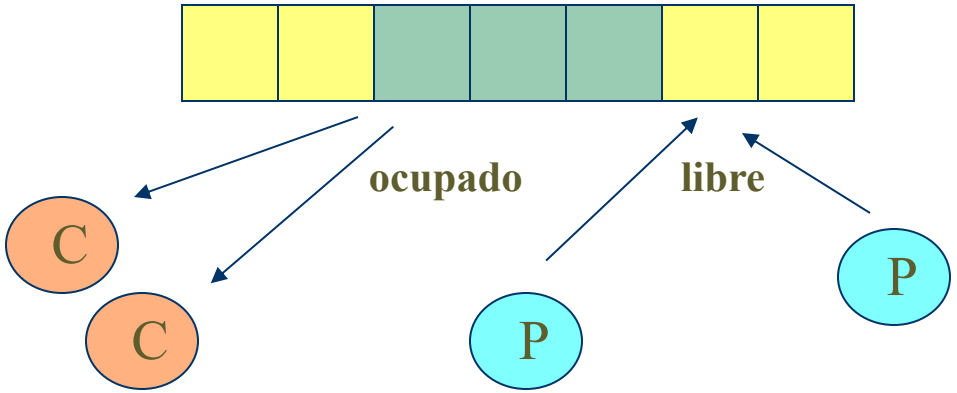
```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;

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?

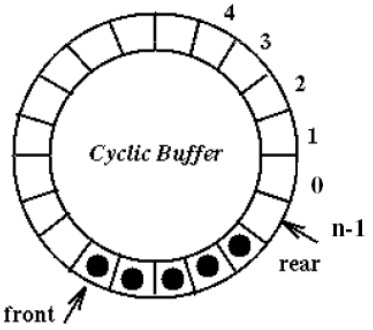


Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos al sobrescribirlo.

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
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;⟩

Problemas básicos y técnicas
Alocación de Recursos y Scheduling

- Varios procesos que compiten por el uso de un recurso compartido *de **una sola unidad***.
- Para el caso general de alocación de recursos SIN ORDEN:

bool libre = true;
request (id): ⟨await (libre) libre = false;⟩
release (): ⟨libre = true; ⟩

Solución al problema de la SC

sem mutex = 1;

Request: P(mutex)
//Usa Recurso Compartido
Release: V(mutex)

Problemas básicos y técnicas
Alocación de Recursos y Scheduling

- Para el caso general de alocación de recursos **por ORDEN de llegada**:

request (id): ⟨await (libre and *miTurno*) libre = false;⟩
release (): ⟨libre = true;⟩

```
sem mutex = 1;
cola espera;          bool libre = true;

request

P(mutex);
push(espera, id);
while ((libre = false) or (top(espera) <> id))
        {  V(mutex);
           P(mutex);}
libre = false;
V(mutex);
```

Busy Waiting

```
release

P(mutex);
libre = true;
pop(espera, ….);
V(mutex);
```

Problemas básicos y técnicas
Alocación de Recursos y Scheduling: *Passing de 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 debe salir 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.

Permite controlar de forma precisa el orden en que los procesos son despertados para continuar su ejecución en la SC utilizando *Semáforos Binarios Divididos (SBS).*

Semáforos Privados: s es un ***semáforo privado*** si exactamente un proceso ejecuta operaciones ***P*** sobre ***s***. Resultan útiles para señalar procesos individuales.

Problemas básicos y técnicas
Alocación de Recursos y Scheduling: *Passing de Baton*

```
bool libre = true;      cola espera;
sem baton = 1
sem b[n] = ([n] 0);

    request(id):
                        P(baton);
                        if (! libre){ push (espera, id);
                                      V(baton);
                                      P(b[id]);
                                    }
                        libre = false;
                        V(baton);

    release( ):
                        P(baton);
                        libre = true;
                        if (not empty(espera)) { pop (espera, id);
                                                 V(b[id]); }
                        else
                              V(baton);
```

Problemas básicos y técnicas
Alocación de Recursos y Scheduling: *Passing de Baton*

```
bool libre = true;        cola espera;        sem baton = 1, b[n] = ([n] 0);
Process Cliente [id: 1..n]
{  int sig, tiempo;
   //Trabaja
   P(baton);
   if (! libre) { push (espera, id);
                  V(baton);
                  P(b[id]);
                }
   libre = false;
   V(baton);
   //USA EL RECURSO
   P(baton);
   libre = true;
   if (not empty(espera)) { pop (espera, sig);
                            V(b[sig]);
                          }
   else V(baton);
}
```

¿Que modificaciones deberían realizarse para respetar otro orden?

Problemas básicos y técnicas
Alocación Shortest-Job-Next (SJN): *Passing de Baton*

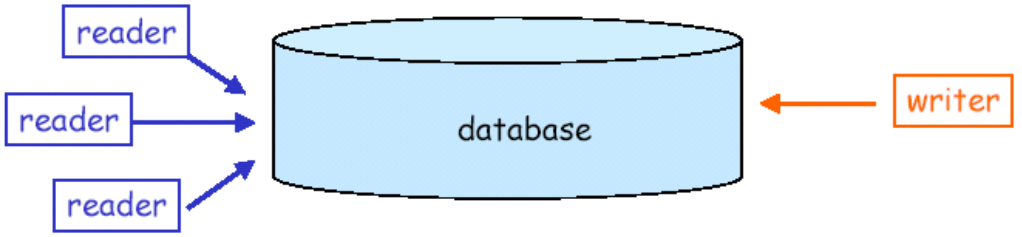
```
bool libre = true;        cola espera;        sem baton = 1, b[n] = ([n] 0);
Process Cliente [id: 1..n]
{  int sig, tiempo;
   //Trabaja
   P(baton);
   if (! libre) { insertar (espera, id, tiempo);
                  V(baton);
                  P(b[id]);
                }
   libre = false;
   V(baton);
   //USA EL RECURSO
   P(baton);
   libre = true;
   if (not empty(espera)) { sacar (espera, sig);
                            V(b[sig]);
                          }
   else V(baton);
}
```

Sólo se debe modificar el orden con que se inserta en la cola de espera

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

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 un problema de ***exclusión mutua selectiva***: procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas.
- 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 ⇒ 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.

```
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; ⟩
    }
}
```

- En algunos casos, await puede ser implementada directamente usando semáforos u otras operaciones primitivas, pero no siempre…En este caso las guardas de los await se superponen en: para los 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***.

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing the 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;
int dr = 0, dw = 0;
sem e = 1;
sem 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);
      }
}
```

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing the 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;
int dr = 0, dw = 0;
sem e = 1;
sem r = 0, w = 0:
```

```
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);
  }
}
```

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing the 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?