Programación Concurrente 2017

Clase 4

Facultad de Informática
UNLP

Resumen de la clase anterior

-Concurrencia y sincronización
- **Acciones atómicas e historias de un programa concurrente**
- **Interferencia - Rol de la sincronización**
- **Atomicidad de grano fino y de grano grueso**
- **Propiedad de ASV**
- **Especificación de la sincronización. AA condicionales e incondicionales**

- Propiedades de programa. Seguridad y vida. Scheduling

Concurrencia con variables compartidas

Problema de la sección crítica
- **Soluciones busy waiting**
- **Soluciones fair: tie breaker, ticket, bakery**

-Sincronización barrier
- **Contador compartido**
- **Flags y coordinador**
- **Arbol**
- **Barreras simétricas → Butterfly**

Sincronización Barrier

Problemas resueltos por algoritmos iterativos que computan sucesivas mejores aproximaciones a una rta, y terminan al encontrarla o al converger.

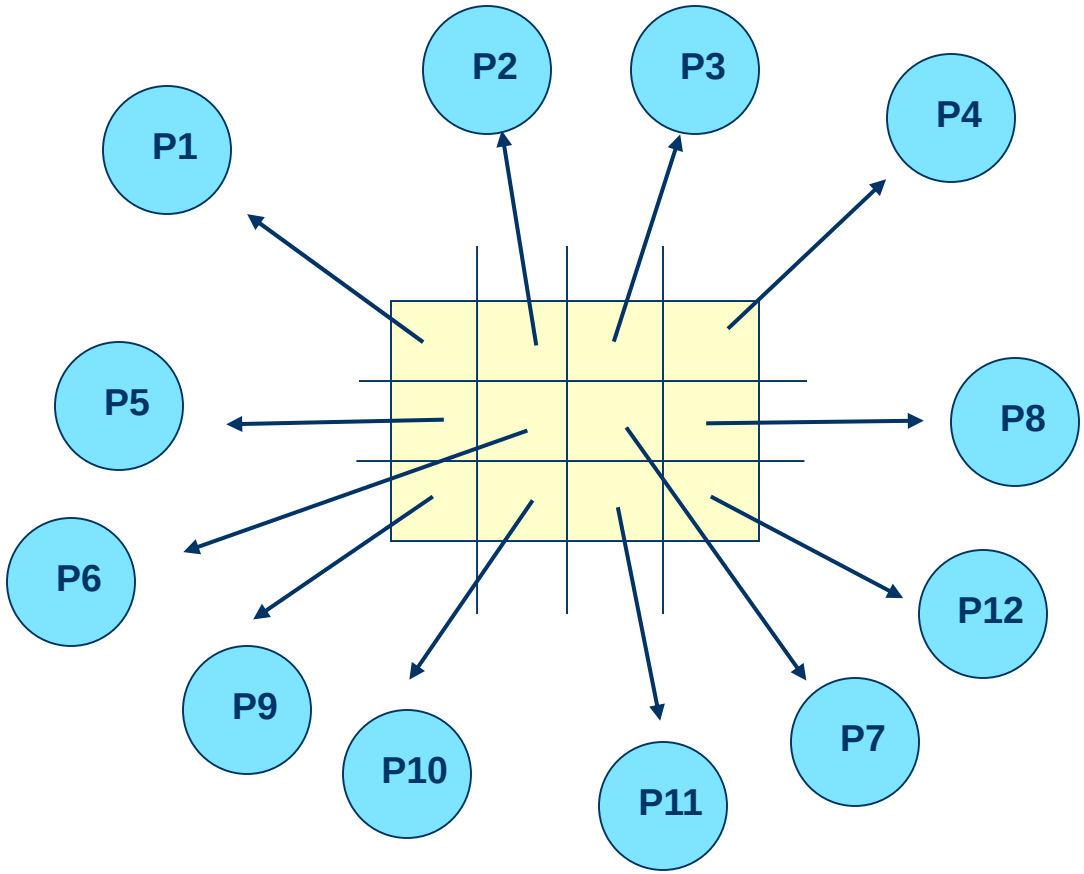
En general manipulan un arreglo, y cada iteración realiza la misma computación sobre todos los elementos del arreglo.

$\Rightarrow$ Múltiples procesos para computar partes disjuntas de la solución en paralelo

En la mayoría de los algoritmos iterativos paralelos cada iteración depende de los resultados de la iteración previa.

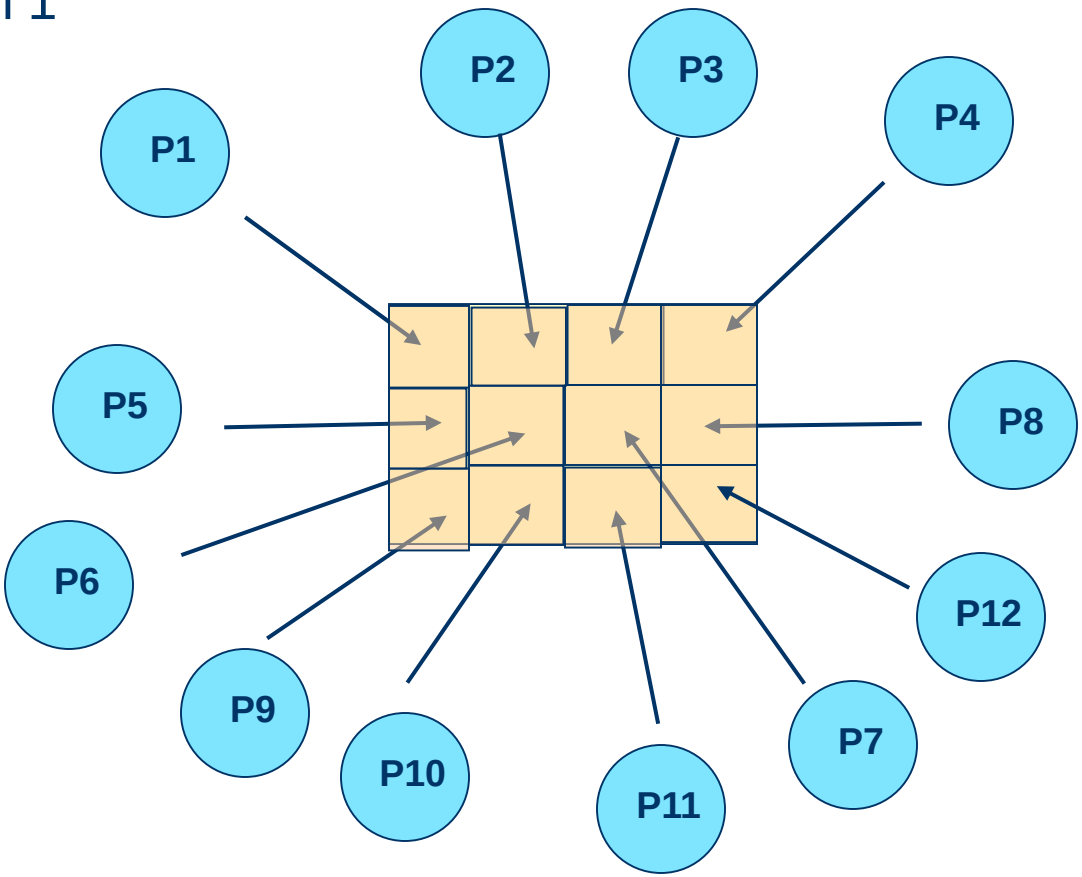
Sincronización Barrier

Inicial



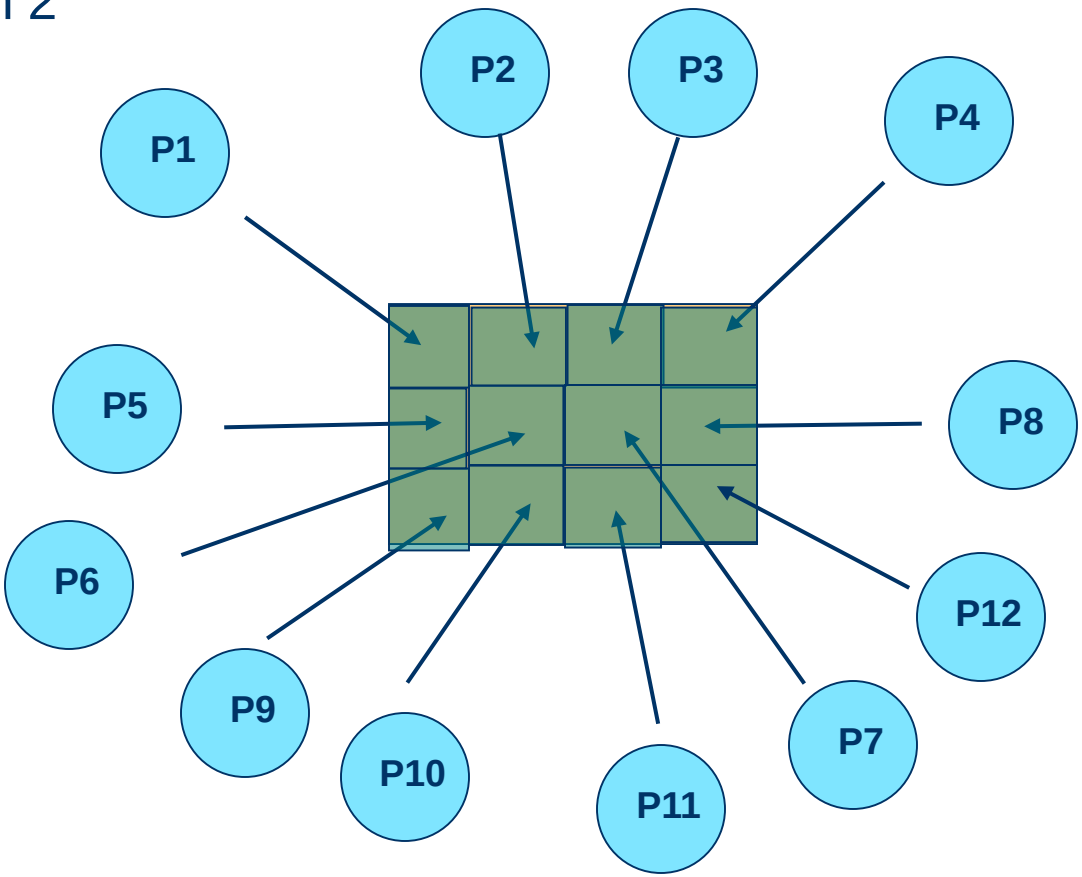
Sincronización Barrier

Iteración 1



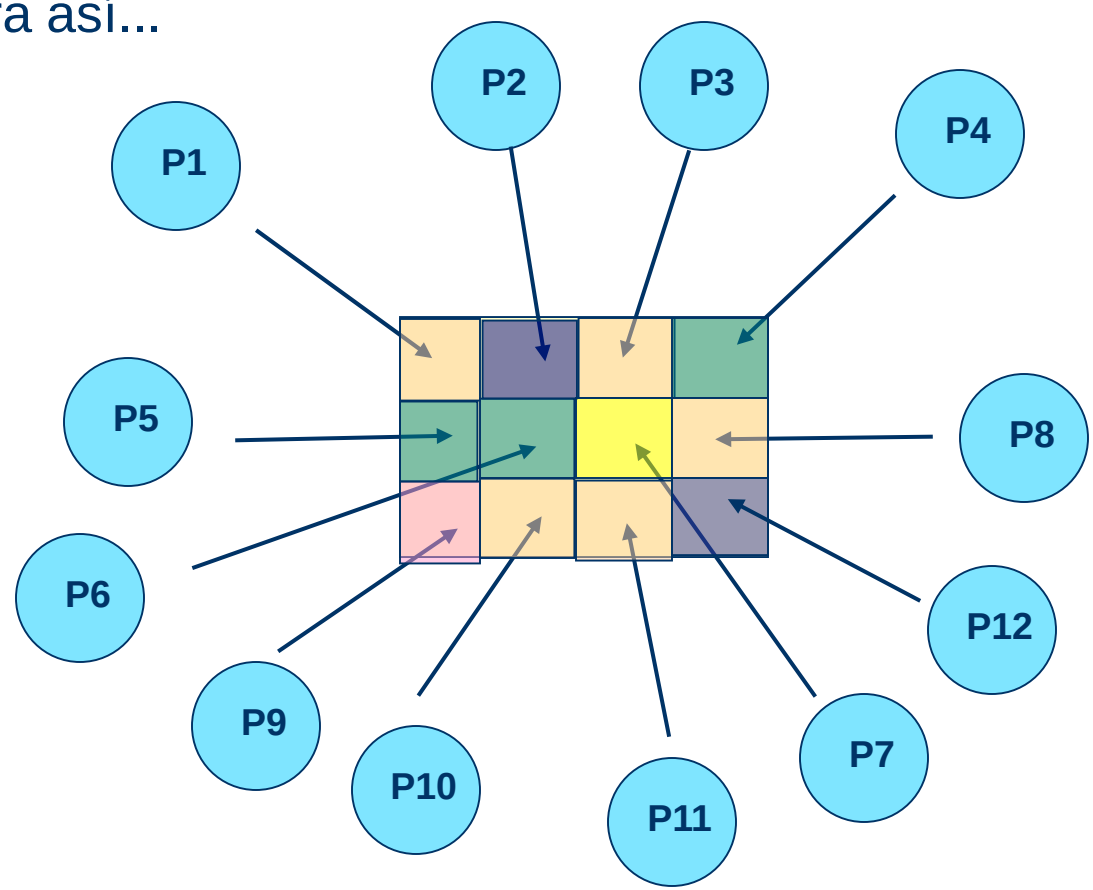
Sincronización Barrier

Iteración 2



Sincronización Barrier

Si no fuera así...



Sincronización Barrier

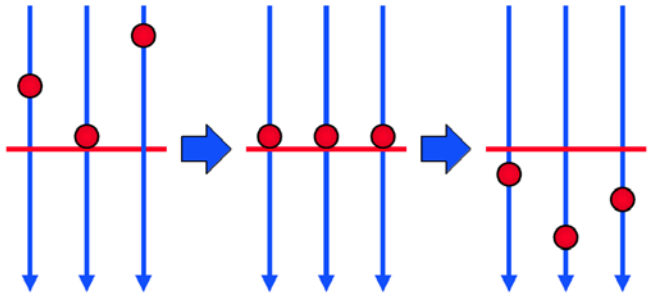
Ignorando terminación, y asumiendo *n* tareas paralelas en cada iteración, se tiene la forma general:

```
while (true) {
    co [i=1 to n]
        código para implementar la tarea i;
    oc   }
```

Ineficiente, ya que produce *n* procesos en cada iteración.

⇒ crear procesos al comienzo y sincronizarlos al final de c/ iteración

```
process Worker[i=1 to n] {
  while (true) {
    código para implementar la tarea i;
    esperar a que se completen las n tareas; }
}
```



Sincronización barrier: el punto de demora al final de c/ iteración es una barrera a la que deben llegar todos antes de permitirles pasar

Sincronización Barrier: Contador Compartido

n workers necesitan encontrarse en una barrera.

⇒ *cantidad* incrementado por c/ worker al llegar

⇒ cuando *cantidad* es *n*, se les permite pasar

```
Int cantidad = 0;
process Worker[i=1 to n] {
    while (true) {
        código para implementar la tarea i;
        ⟨ cantidad = cantidad + 1; ⟩
        ⟨ await (cantidad == n); ⟩
    }
}
```

Se puede implementar con:

```
FA(cantidad,1);
while (cantidad <> n) skip;
```

Problemas: *cantidad* necesita ser 0 en cada iteración, puede haber contención de memoria, coherencia de cache, ...

Sincronización Barrier: Flags y Coordinadores

Puede “distribuirse” *cantidad* usando *n* variables (arreglo *arribo*[1..n])

El await pasaría a ser ⟨ **await (arribo[1] + … + arribo[n] == n);** ⟩

Reintroduce memory contention y es ineficiente

Puede usarse un conjunto de valores adicionales y un proceso más.
⇒ ***Cada Worker espera por un único valor***

Sincronización Barrier: Flags y Coordinadores

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);
# arribo y continuar son "flags"
process Worker[i=1 to n] {
    while (true) {
        código para implementar la tarea i;
        arribo[i] = 1;
        ⟨ await (continuar[i] == 1); ⟩
        continuar[i] = 0;
    }
}
process Coordinador {
    while (true) {
        for [i=1 to n] {
            ⟨ await (arribo[i] == 1); ⟩
            arribo[i] = 0;
        }
        for [i = 1 to n] continuar[i] = 1;
    }
}
```

Sincronización Barrier: Arboles

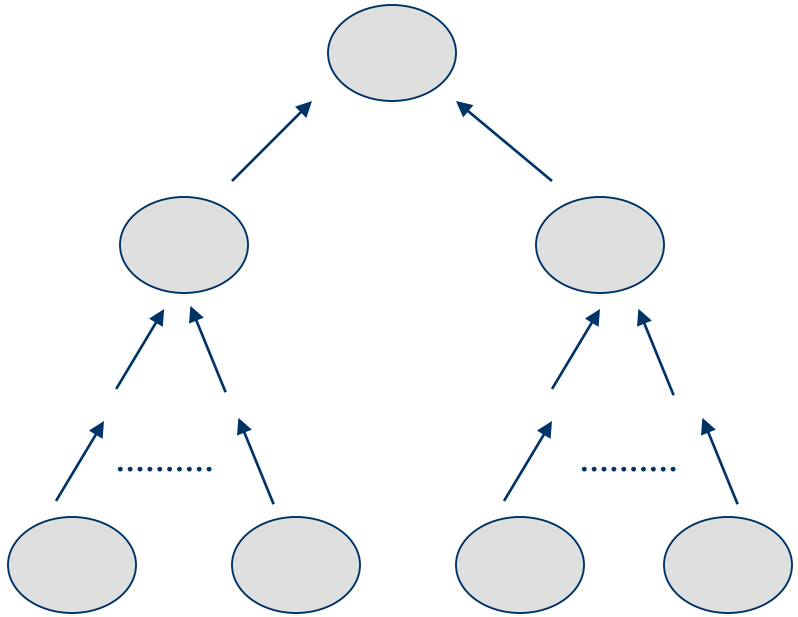
***Problemas*:**
Requiere un proceso (*y procesador*) extra.

El tiempo de ejecución del coordinador es proporcional a *n*

Posible solución: combinar las acciones de workers y coordinador, haciendo que cada worker sea también coordinador.

Por ejemplo, workers en forma de árbol: las señales de *arribo* van hacia arriba en el árbol, y las de *continuar* hacia abajo

⇒ ***combining tree barrier*** (más eficiente para *n* grande)



Sincronización Barrier: Barreras Simétricas

En *combining tree barrier* los procesos juegan diferentes roles.

Una ***barrera simétrica*** para *n* procesos se construye a partir de pares de barreras simples para dos procesos:

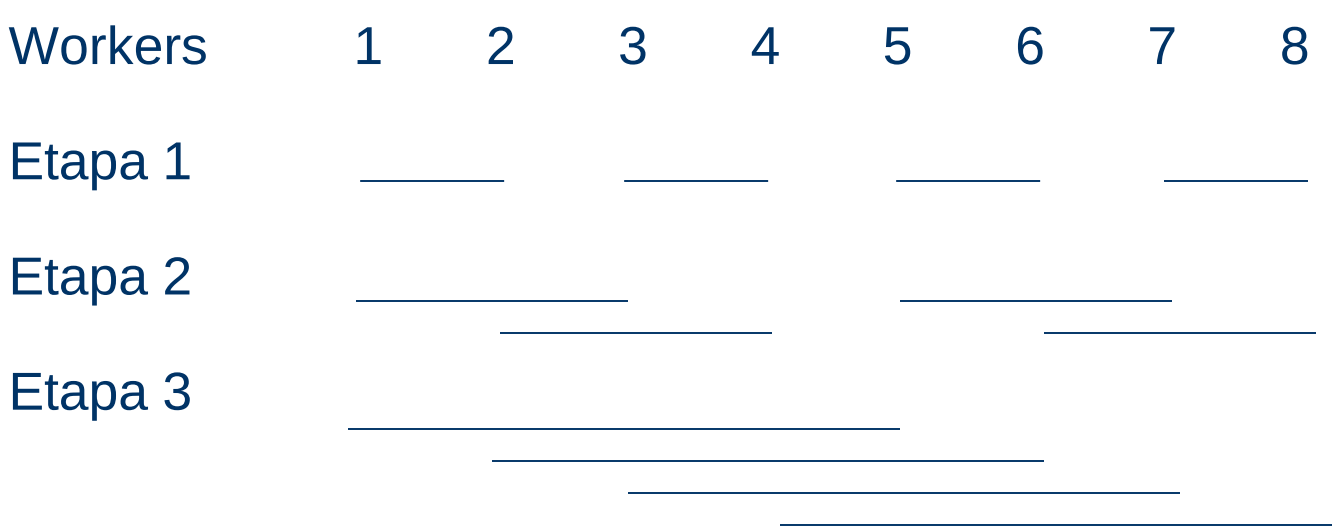
W[i]::⟨ await (arribo[i] == 0); ⟩
arribo[i] = 1;
⟨ await (arribo[j] == 1); ⟩
arribo[j] = 0;

W[j]::⟨ await (arribo[j] == 0); ⟩
arribo[j] = 1;
⟨ await (arribo[i] == 1); ⟩
arribo[i] = 0;

Cómo se combinan para construir una barrera *n* proceso???

Worker[1:n] arreglo de procesos. Si *n* es potencia de 2 ⇒ *butterfly barrier*

Sincronización Barrier: Butterfly Barrier



$\log_2 n$ etapas: cada Worker sincroniza con uno distinto en c/ etapa

En la etapa *s*, un Worker sincroniza con otro a distancia 2^{s-1}

Cuando cada Worker pasó $\log_2 n$ etapas, todos deben haber llegado a la barrera y por lo tanto todos pueden seguir

Computaciones de Prefijo Paralelo

Algoritmos *data parallel* → varios procesos ejecutan el mismo código y trabajan en distintas partes de datos compartidos.

Ejemplo: computar en paralelo las sumas de los prefijos de un arreglo a[n], para obtener sum[n], donde sum[i] es la suma de los primeros i elementos de a.

Secuencialmente:

```
sum[0] = a[0];
for [i=1 to n-1]  sum[i] = sum[i-1] + a[i];
```

Cómo se puede paralelizar?

Computaciones de Prefijo Paralelo

Antes: Cómo se podría obtener en paralelo la suma de todos los elementos??
Idea: Sumar en paralelo pares contiguos, luego pares a distancia 2, luego a distancia 4, etc. $\Rightarrow$ ($\log_2 n$) pasos

1) Setear sum[i]=a[i]
2) En paralelo, sumar sum[i-1] a sum[i], $\forall i > 1$ (suma a distancia 1)
3) Luego, doblar la distancia, sumando sum[i-2] a sum[i], $\forall i > 2$
4) Luego de ($\log_2 n$) rondas se tienen todas las sumas parciales

valores iniciales de a[1:6]	1	2	3	4	5	6
sum después de distancia 1	1	3	5	7	9	11
sum después de distancia 2	1	3	6	10	14	18
sum después de distancia 4	1	3	6	10	15	21

Suma paralela de prefijos

```
int a[n], sum[n], viejo[n];
process Sum[i=0 to n-1] {
      int d = 1;
      sum[i] = a[i];
      barrier(i);
      while (d<n) {
            viejo[i] = sum[i];
            barrier(i);
            if ( (i-d) >= 0 ) sum[i] = viejo[i-d] + sum[i];
            barrier(i);
            d = 2 * d;
       }
}
```

Cuál sería el efecto de usar un multiprocesador sincrónico ??

Defectos de la sincronización por busy waiting

***Protocolos "busy-waiting"*: complejos y sin clara separación e*l* vbles 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.

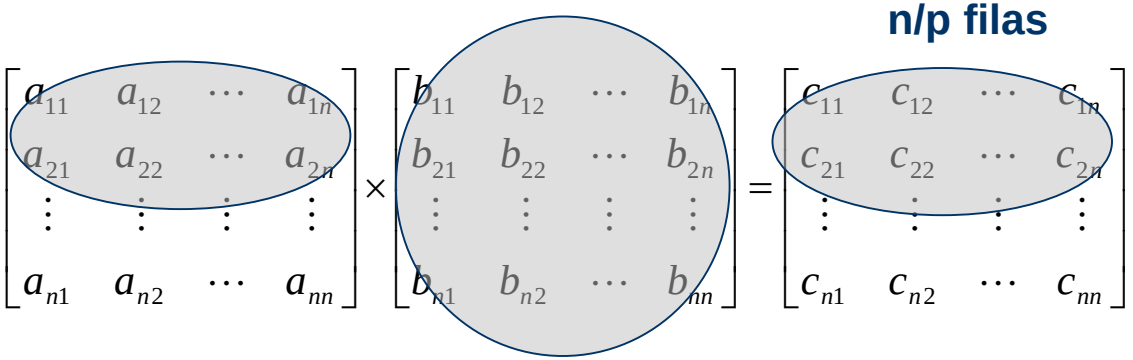
Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso *spinning* puede ser usado de manera más productiva por otro proceso.

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

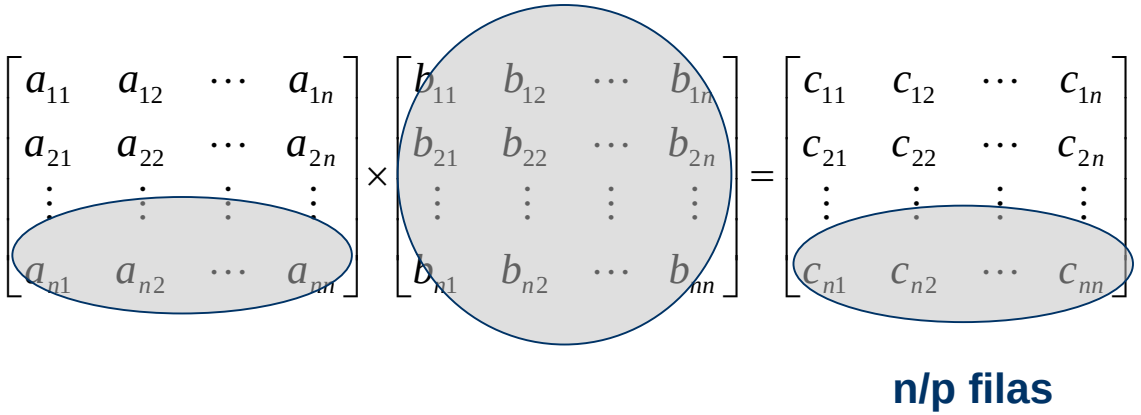
Resolución ejercicio similar al del cuestionario

En paralelo

Worker 1



Worker p



Resolución ejercicio similar al del cuestionario

P= 8, n=128. Cuántas asignaciones, sumas y productos hace cada procesador?
Si P1=...=P7 y los tiempos de asignación son 1, de suma 2 y de producto 3; si P8 es 4 veces más lento, Cuánto tarda el proceso total? Qué puede hacerse para mejorar el speedup?

```
process worker [w = 1 to P] { # strips en paralelo (p strips de n/P filas)
  int first = (w-1) * n/P + 1        # Primera fila del strip
  int last = first + n/P - 1;        # Ultima fila del strip
  for [i = first to last] {
     for [j = 1 to n] {
           c[i,j] = 0.0;
           for [k = 1 to n]
             c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
     }
}
```

Resolución ejercicio similar al del cuestionario

```
process worker [w = 1 to 8] {        # 8 strips en paralelo. n/p = 16
  int first = (w-1) * 16 + 1;        # Primera fila del strip
  int last = first + 16 - 1;         # Ultima fila del strip
  # P1 = 1 a 16, P2 = 17 a 32, P3 = 33 a 48, P4 = 49 a 64
  # P5 = 65 a 80, P6 = 81 a 96, P3 = 97 a 112, P8=113 a 128
  for [i = first to last] {

     for [j = 1 to 128] {
           c[i,j] = 0.0;   #  128 asignaciones

           for [k = 1 to 128]
             c[i,j] = c[i,j] + a[i,k]*b[k,j];   #128 prods, 128 sumas, 128 asign.
        }
   }
}
```

Sin considerar los incrementos de índices:

Total = $128^2*16+128*16 = 264192$ asig,

$128^2*16 = 262144$ sumas, $128^2*16 = 262144$ prod.

Resolución ejercicio similar al del cuestionario

P1 a P8 tienen igual número de operaciones.

Total = 264192 asignaciones 262144 sumas 262144 productos

P1 =...= P7 = 264192 + 524288 + 786432 = 1574912 unid. de tiempo

Para P8 = 1574912 x 4 = 6299648 unidades de tiempo
Hay que esperar a P8 ...

Si todo fuera secuencial:
128^3+128^2 =2113536 as. **128^3** =2097152 su **128^3** =2097152 prod

En P1 = ... = P7 = 2113536 + 4194304 + 6291456 = 12599296 unid. de tiempo

Speedup = 12599296 / 6299648 = 2

Resolución ejercicio similar al del cuestionario

Speedup = 12599296 / 6299648 = 2

Si le damos a P8 solo 2 filas y a P1 a P7 18 filas, podemos corregir los tiempos:

P1=...=P7= 297216 as. 294912 sum 294912 prod
P1=...=P7= 297216 + 589824 + 884736= 1771776 unid. de tiempo

P8 con dos filas = 787456 unid. de tiempo

Speedup = 12599296 / 1771776 = 7.1

⇒ *Mejor balance carga* ⇒ *Mejor Speedup*

⇒ *POR QUE NO el Speedup = 8 ??*

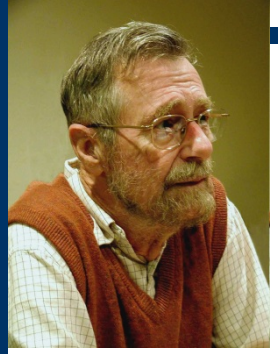
Defectos de la sincronización por busy waiting

***Protocolos "busy-waiting"*: complejos y sin clara separación e*l* vbles de sincronización y las usadas para computar resultados.**

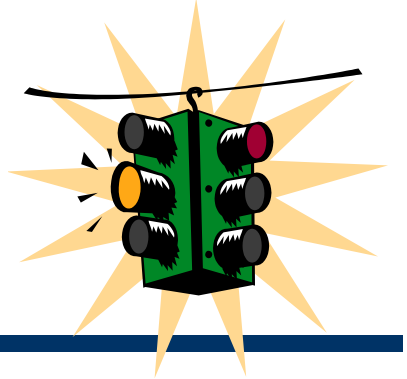
Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.

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 (http://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**

Analogía con la sincronización del tránsito p/ evitar colisiones.

El valor de un semáforo es *no negativo*
V señala la ocurrencia de un evento (incrementa)
P se usa para demorar un proceso hasta que ocurra un evento (decrementa)

Permiten proteger *Secciones Críticas* y pueden usarse para implementar *sincronización por condición*

Semáforos. Operaciones básicas

Declaraciones:
sem s;
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* (asumiremos que NO es así).

Semáforos. Problemas básicos y técnicas de sincronización

PROBLEMA DE LA SECCION CRITICA. EXCLUSION MUTUA

```
bool lock=false;
process SC[i=1 to n] {
   while (true) {   ⟨await (not lock) lock= true; ⟩   # protocolo de entrada
                    sección crítica;
                    lock = false;                      # protocolo de salida
                    sección no crítica;
   }
}
```

```
bool free=true;
process SC[i=1 to n] {
   while (true) {   ⟨await (free) free= false; ⟩  # protocolo de entrada
                    sección crítica;
                    free = true;                    # protocolo de salida
                    sección no crítica;
   }
}
```

Semáforos. Secciones Críticas

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

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

```
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...*

Semáforos. Barreras

BARRERAS. SEÑALIZACION 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)

Semáforos. Barreras

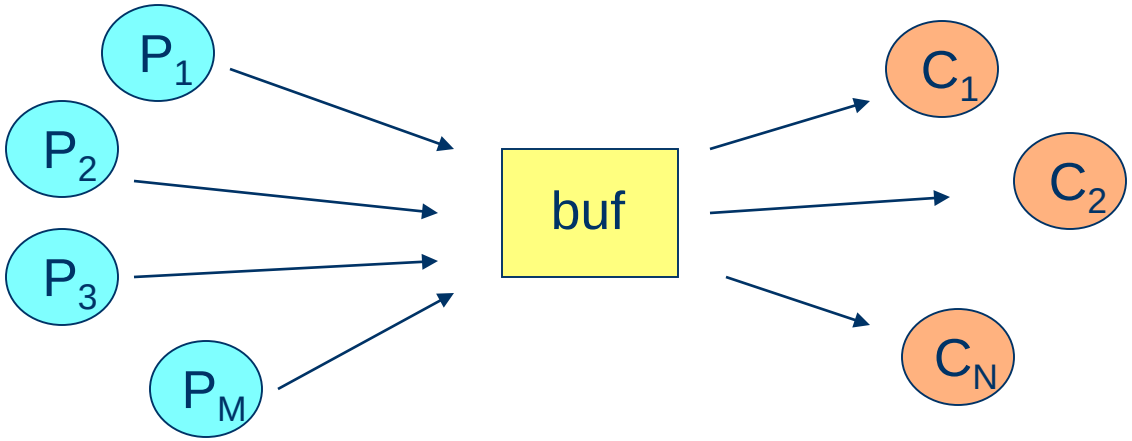
En una barrera p/ 2 procesos, los eventos significativos son las llegadas de los procesos a la barrera $\Rightarrow$ **2 semáforos de señalización.**

```
sem llega1=0, llega2=0;
process Worker1 {
        ..........
        V(llega1);        # señaliza el arribo
        P(llega2);        # espera al otro proceso
        ..........
}
process Worker2 {
        ..........
        V(llega2);        # señaliza el arribo
        P(llega1);        # espera al otro proceso
        ..........
}
```

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 (split)

Buffer unitario compartido.
Múltiples productores y múltiples consumidores.
Dos operaciones: *depositar* y *retirar* $\Rightarrow$ deben alternarse



Dos semáforos (*vacío* y *lleno*)

Inicialización: vacío en 1 y lleno en 0

Ambos semáforos binarios (por qué?)

Productores y Consumidores. Semáforos binarios divididos (split)

```
typeT buf;        # un buffer de algún tipo
sem vacio = 1, lleno = 0;
process Productor[i = 1 to M] {
    while(true) {  …
            producir mensaje datos
            depositar: P(vacio);
                       buf = datos;
                       V(lleno)
    }
}
process Consumidor[j = 1 to N] {
    while(true) {
            retirar:   P(lleno);
                       resultado = buf;
                       V(vacio)
            consumir mensaje resultado
            …
    }
}
```

Productores y Consumidores. Semáforos binarios divididos (*split*)

vacio y lleno, juntos, forman lo que se denomina “semáforo binario dividido” (*split binary semaphore*): a lo sumo uno de ellos puede ser 1 a la vez

***Split Binary Semaphore*. Los semáforos binarios b_1, ..., b_n forman un SBS en un programa si el siguiente es un invariante global:**

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

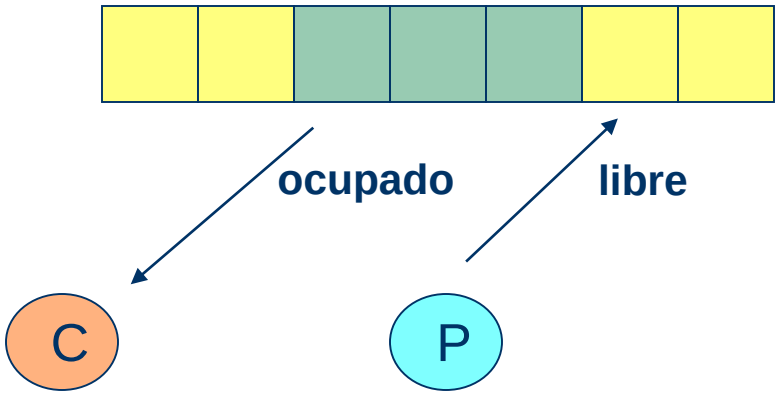
Los b_i pueden verse como un único semáforo binario *b* que fue dividido en *n* semáforos binarios

Importantes por la forma en que pueden usarse para implementar EM (en gral 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. Semáforos como Contadores de recursos

Un productor y un consumidor



El buffer es una cola de mensajes depositados y aún no buscados

Buffers Limitados. Semáforos como Contadores de recursos

```
typeT buf[n];       # un buffer de algún tipo
int ocupado = 0, libre = 0;   sem vacio = n, lleno = 0;

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

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

Buffers Limitados. Semáforos como Contadores de recursos

Los semáforos en este ejemplo son *contadores de recursos*: cada uno cuenta el número de unidades de un recurso.

vacio cuenta los slots vacíos y *lleno* cuenta los slots llenos.

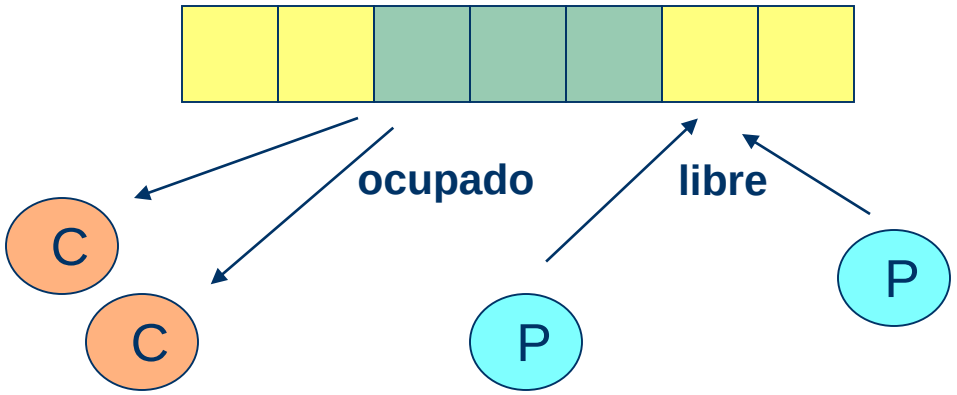
Esta forma de utilización es adecuada cuando los procesos compiten por recursos *de múltiple unidad*

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??

Buffers Limitados. Semáforos como 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 EM (cuáles serían las consecuencias de no protegerlas?)



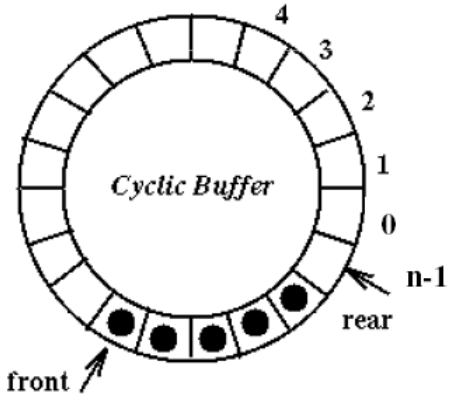
Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos por sobreescritura

Buffers Limitados. Semáforos como Contadores de recursos

```
typeT bu[n];        # un buffer de algún tipo
int ocupado = 0, libre = 0;
sem vacio = n, lleno = 0;
sem mutexD = 1, mutexR = 1;

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

process Consumidor[j = 1 to N] {
    while(true) {
            retirar:   P(lleno);
                       P(mutexR);
                       resultado = buf[ocupado]; ocupado = (ocupado+1) mod n;
                       V(mutexR);
                       V(vacio)
            consumir mensaje resultado
            …   }  }
```



Varios procesos compitiendo por varios recursos compartidos

Problema de varios procesos (P) y varios recursos (R) cada uno protegido por un lock.

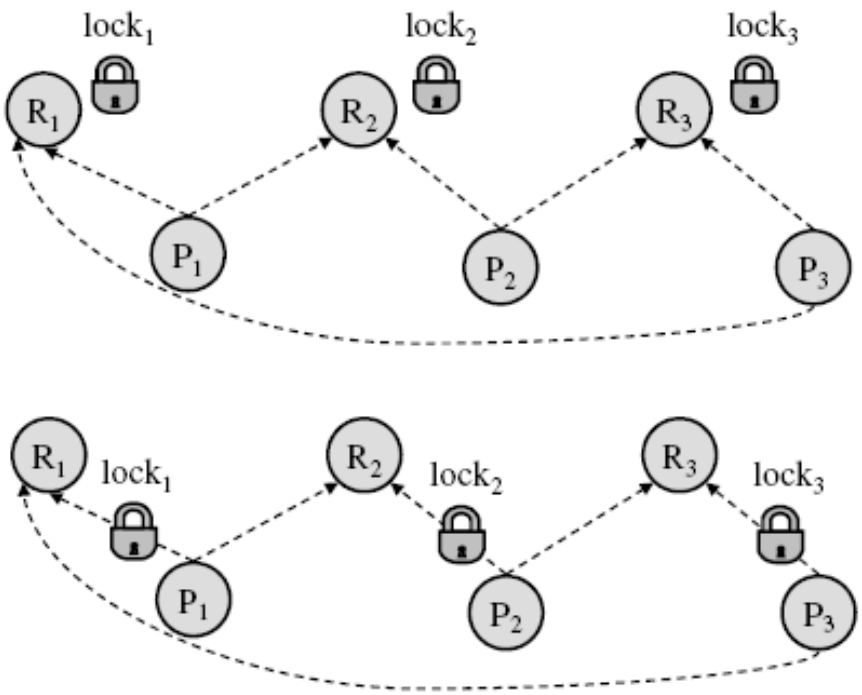
Un proceso debe adquirir los locks de todos los recursos que necesita.

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 mod n]

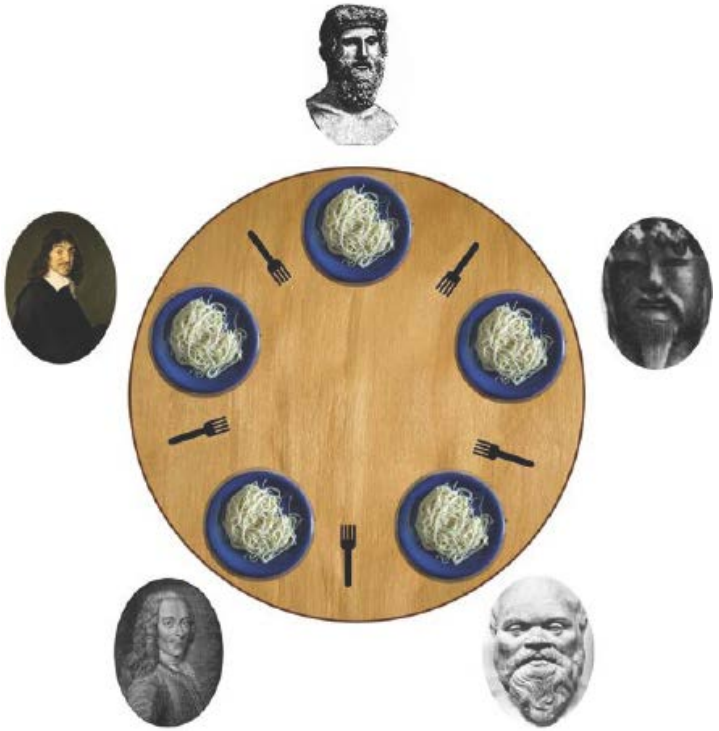
-Deadlock: cada proceso adquiere lock[i] y espera por lock[i+1 mod n] que ya fue adquirido por P [i+1 mod n]



Exclusión mutua selectiva. El problema de los filósofos

Ejemplo de problema de EM entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas.

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



Cómo se especifica el adquirir y liberar los tenedores?

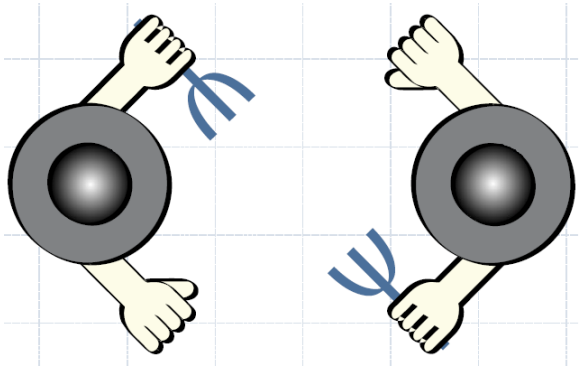
Exclusión mutua selectiva. El problema de los filósofos

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?



Exclusión mutua selectiva. El problema de los filósofos

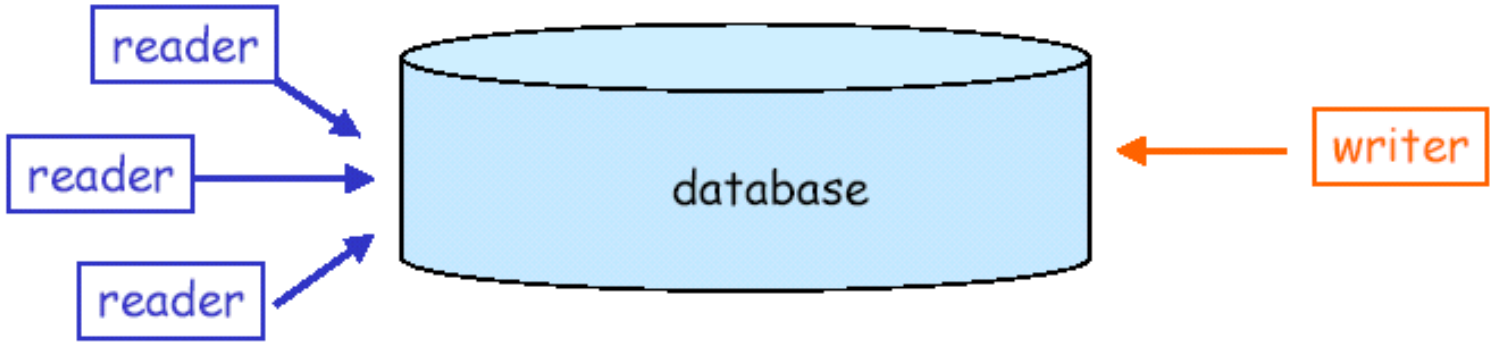
```
sem tenedor[5] = {1, 1, 1, 1, 1};        # Por qué inicializados en 1??
process Filosofo [i = 0 to 3] {
    while(true) {
            P(tenedor[i]); P(tenedor[i+1]);   # toma tenedor izq. y luego der.
            come;
            V(tenedor[i]); V(tenedor[i+1]);
            piensa;
    }
}
process Filosofo [4] {
    while(true) {
            P(tenedor[0]); P(tenedor[4]);     # toma tenedor der. y luego izq.
            come;
            V(tenedor[0]); V(tenedor[4]);
            piensa;
    }
}
```

Lectores y Escritores

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



Lectores y Escritores

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

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);    # toma bloqueo de acceso exclusivo
            lee la BD;
            V(rw);    # libera el bloqueo
    } }
process Escritor [j = 1 to N] {
    while(true) { …
            P(rw);    # toma bloqueo de acceso exclusivo
            escribe la BD;
            V(rw);    # libera el bloqueo
    } }
```

SOLUCION DEMASIADO RESTRICTIVA …

Lectores y Escritores como problema de Exclusión mutua

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).

```
int nr = 0;   # número de lectores activos
sem rw = 1;

process Lector [i = 1 to M] {
    while(true) { …
        ⟨ nr = nr + 1;
          if (nr == 1) P(rw); ⟩  # si es el primero, toma el bloqueo
        lee la BD;
        ⟨ nr = nr - 1;
          if (nr == 0) V(rw); ⟩  # si es el último, libera el bloqueo    } }

process Escritor [j = 1 to N] {
    while(true) { …
        P(rw);    # toma bloqueo de acceso exclusivo
        escribe la BD;
        V(rw);    # libera el bloqueo                  } }
```

Lectores y Escritores como problema de Exclusión mutua

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

process Lector [i = 1 to M] {
    while(true) { …
          P(mutexR);
            nr = nr + 1;
            if (nr == 1) P(rw);   # si es el primero, toma el bloqueo
          V(mutexR);
          lee la BD;
          P(mutexR);
            nr = nr - 1;
            if (nr == 0) V(rw);   # si es el último, libera el bloqueo
          V(mutexR);   } }

process Escritor [j = 1 to N] {
    while(true) { …
          P(rw);    # toma bloqueo de acceso exclusivo
          escribe la BD;
          V(rw);    # libera el bloqueo                } }
```

```
int nr = 0;   sem rw = 1;
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); ⟩   } }
process Escritor [j = 1 to N] {
    while(true) { …
        P(rw);
        escribe la BD;
        V(rw);    } }
```

Lectores y Escritores usando Sincronización por Condición

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

Además, no es sencillo modificarla …

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 los procesos de cada clase intentando acceder a la BD, y luego restringir el valor de los contadores

Lectores y Escritores usando Sincronización por Condición

nr y nw enteros no negativos que registran el número de lectores y escritores accediendo a la BD

Estados malos a evitar: nr y nw positivos, o nw mayor que 1:
BAD: (nr > 0 AND nw > 0) OR nw > 1

⇒ estados buenos caracterizados por NOT BAD
RW: (nr == 0 OR nw == 0) AND nw <= 1

```
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; ⟩              } }
```

La 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 $\Rightarrow$ ***Passing the baton***: *técnica general para implementar sentencias* ***await***

La técnica *Passing the Baton*

La sincronización se expresa con sentencias atómicas de la forma:

$F_1 : \langle S_i \rangle$ **o** **$F_2 : \langle$ await (B_j) $S_j \rangle$**

Puede hacerse con semáforos binarios divididos (SBS)

e sem. binario inicialmente 1 (controla la E/ 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 usa para formar un SBS: a lo sumo uno a la vez es 1, y c/ camino de ejecución empieza con un P y termina con un único V

La técnica *Passing the 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

La técnica *Passing the Baton*

***Passing the baton*:** 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**, pasa el *baton* (control) a otro proceso.

Si ningún proceso está esperando una condición que sea ***true,*** el baton se pasa ***al próximo proceso que trata de entrar a su SC*** por primera vez (es decir, uno que ejecuta P(e)).

Lectores y escritores:
- **e** semáforo p/ controlar acceso a las vbles compartidas, **r** semáforo asociado a la guarda en procesos lectores, y **w** asociado a la guarda en escritores
- **dr** y **dw** contadores de lectores y escritores esperando

Passing the Baton en Lectores y Escritores

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0:        # siempre  0 <= (e+r+w) <= 1
int dr = 0, dw = 0;
process Lector [i = 1 to M] {
    while(true) {
        # ⟨ await (nw == 0) nr = nr + 1; ⟩
        P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL1 ;
        lee la BD;
        # ⟨ nr = nr – 1; ⟩
        P(e);
        nr = nr – 1;
        SIGNAL2 ;
   }
}
```

```
F1:   P(e);
      Si;
      SIGNAL
F2:   P(e)
      if (not Bj) {dj = dj + 1; V(e); P(bj); }
      Sj;
      SIGNAL
```

Passing the Baton en Lectores y Escritores

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

```
F1:   P(e);
      Si;
      SIGNAL
F2:   P(e)
      if (not Bj) {dj = dj + 1; V(e); P(bj); }
      Sj;
      SIGNAL
```

El rol de los SIGNAL ¡ es el de señalizar *exactamente* a uno de los semáforos ⇒ los procesos se van pasando el *baton*

Passing the Baton en Lectores y Escritores

```
F1:  P(e);
     Si;
     SIGNAL
F2:  P(e)
     if (not Bj) {dj = dj + 1; V(e); P(bj); }
     Sj;
     SIGNAL
```

$SIGNAL_i$ es una abreviación de:

```
if (nw == 0 and dr > 0) {
   dr = dr - 1; V(r);          # despierta un lector, o
   }
elseif (nr == 0 and nw == 0 and dw > 0) {
   dw = dw - 1; V(w);          # despierta un escritor, o
   }
else
   V(e);                       # libera el bloqueo de E/
```

e, w y r forman un SBS

Antes de $SIGNAL_1$, nr > 0 y nw = 0 son true ⇒ puede simplificarse a:

```
if (dr > 0) {dr := dr -1; V(r); }
   else V(e);
```

Algo similar sucede con los otros SIGNAL ⇒

Passing the Baton en Lectores y Escritores

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0:     # siempre  0 <= (e+r+w) <= 1
int dr = 0, dw = 0;
process Lector [i = 1 to M] {
    while(true) {
            # ⟨ await (nw == 0) nr = nr + 1; ⟩
            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;
            # ⟨ nr = nr – 1; ⟩
            P(e);
            nr = nr – 1;
            if (nr == 0 and dw > 0) {dw = dw – 1; V(w); }
              else V(e);
   }
}
```

Passing the Baton en Lectores y Escritores

```
process Escritor [j = 1 to N] {
    while(true) {
            # ⟨ await (nr == 0 and nw == 0) nw = nw + 1; ⟩
            P(e);
            if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
            nw = nw + 1;
            V(e);
            escribe la BD;
            # ⟨ nw = nw – 1; ⟩
            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

Recurso ⇒ cualquier objeto, elemento, componente, dato, SC, por la que un proceso puede ser demorado esperando adquirirlo

Hasta ahora aplicamos la política más simple: si algún proceso está esperando y el recurso está disponible, se lo asigna

La política de alocación más compleja fue en R/W, aunque daba preferencia a *clases* de procesos, no a procesos individuales

Cómo se pueden implementar políticas de alocación de recursos generales y cómo controlar explícitamente cuál proceso toma un recurso si hay más de uno esperando??

Alocación de Recursos y Scheduling. Definición del problema y patrón de solución general

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 tiene la forma de F_2
request (parámetros): P(e);
if (request no puede ser satisfecho) DELAY;
tomar las unidades;
SIGNAL;
release tiene la forma de F_1
release (parámetros): P(e);
retornar unidades;
SIGNAL;

Alocación *Shortest-Job-Next*

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

Ejemplos:
- alocación de procesador (*tiempo* es el tiempo de ejecución)
- spooling de una impresora (*tiempo* tiempo de impresión)

Alocación *Shortest-Job-Next*

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 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 (esto es, si *libre* es falso).

Alocación *Shortest-Job-Next*

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

release ( ):
        P(e);
        libre = true;
        SIGNAL;
```

Qué hay que hacer en DELAY y SIGNAL ????

Alocación *Shortest-Job-Next*

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.

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*.

***b*[1:n]** arreglo de semáforos, donde cada entry es inicialmente 0
$\Rightarrow$ el proceso *id* se demora sobre el semáforo b[id]

Alocación *Shortest-Job-Next*

```
bool libre = true;
sem e = 1, b[n] = ([n] 0);      # para entry y demora
typedef Pares = set of (int, int);
Pares =  ∅ ;
# SJN: Pares es un conjunto ordenado ∧ libre ⇒ ( pares== ∅ )

request(tiempo,id): P(e);
                    if (! free) {
                        insertar (tiempo,id) en Pares;
                        V(e);       # libera el bloqueo
                        P(b[id]);  # espera en SU semáforo a ser despertado }
                     libre = false;    # Se está tomando el recurso
                     V(e);        # libera el bloqueo p/ encolar otros procesos

release( ):         P(e);
                    libre = true;
                    if (Pares ≠ ∅ )  {
                        remover el primer par (tiempo,id) de Pares;
                        V(b[id]);      # pasa el baton al proceso id      }
                    else V(e);
```

Alocación *Shortest-Job-Next*

Los semáforos b[id] son ejemplos de *semáforos privados*.

A diferencia de los vistos anteriormente, se asocian con un único proceso

s es un semáforo privado si exactamente un proceso ejecuta operaciones P sobre s

Resultan útiles para señalar procesos individuales.

TAREA: qué modificaciones deberían realizarse para generalizar la solución a recursos de múltiple unidad??

Librería para uso de semáforos - Conceptos de Pthreads

Thread $\rightarrow$ proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).

Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones "multithreading".

En principio estos mecanismos fueron heterogéneos y poco portables $\Rightarrow$ a mediados de los 90 la organización POSIX auspició el desarrolló de una biblioteca en C para multithreading **(*Pthreads*)**

La biblioteca está actualmente en varias versiones de Unix, y con ella se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

Conceptos de Pthreads

```
# include <pthread.h>
```

Declaración de variables para un descriptor de atributos de thread y uno o más descriptores de thread:

```
pthread_attr_t attr;        /* atributos del thread */
pthread_t pid;              /* descriptor del thread */
```

Los atributos incluyen tamaño del stack del thread, prioridad de scheduling, y alcance del scheduling (local o global)

Si es global, el thread compite con todos los otros por el uso de procesador y no sólo con los creados por el mismo thread padre

Inicialización de atributos:

```
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

Conceptos de Pthreads

Creación de thread:

```
pthread_create(&tid, &attr, start_func, arg);
```

donde

- &tid es la dirección de un descriptor que se llena si la creación tiene éxito
- &attr es la dirección de un descriptor inicializado previamente
- el thread comienza la ejecución llamando a start_func con un argumento arg

Un thread termina su propia ejecución llamando a

```
pthread_exit(value);
```

Un thread padre puede esperar a que termine un hijo con

```
pthread_join(tid, value_ptr);
```

donde tid es un descriptor y value_ptr es la dirección de una posición para el valor de retorno (que se llena cuando el hijo llama a exit)

Semáforos con Pthreads

Los threads pueden sincronizar por semáforos (librería semaphore.h)

Declaración y operaciones:

- ***sem_t semaforo*** (se declaran globales a los threads)
- ***sem_init (&semaforo, alcance, inicial)***

Inicializa el semáforo ***semaforo***. ***Inicial*** es el valor con que comienza. ***Alcance*** indica si es compartido por threads de un mismo proceso (0) o por los de todos los procesos (1)

- ***sem_wait(&semaforo)*** equivale a P
- ***sem_post(&semaforo)*** equivale a V

Existen funciones de wait condicional, para obtener el valor de un semáforo y para destruirlo.

Productores/Consumidores con Pthreads

Las funciones de Productor y Consumidor serán ejecutadas por threads independientes.

Acceden a un buffer compartido (*datos*).

El productor deposita una secuencia de enteros de 1 a *numItems* en el buffer.

El consumidor busca estos valores y los suma.

Los semáforos *vacio* y *full* garantizan el acceso alternativo de productor y consumidor sobre el buffer.

Productores/Consumidores con Pthreads

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1
#include <stdio.h>

void *Productor(void *); /* los dos threads */
void *Consumidor(void *);
sem_t vacio, lleno; /* semáforos globales */
int datos; /* buffer compartido */
int numItems;

int main(int argc, char * argv[ ]) {
    ....
    sem_init(&vacio, SHARED, 1); /* sem vacio = 1 */
    sem_init(&lleno, SHARED, 0);  /* sem lleno = 0 */
    numItems = atoi(argv[1]);
    ....
   pthread_create(&pid, &attr, Productor, NULL);
   pthread_create(&cid, &attr, Consumidor, NULL);
   pthread_join(pid, NULL);
   pthread_join(cid, NULL);        }
```

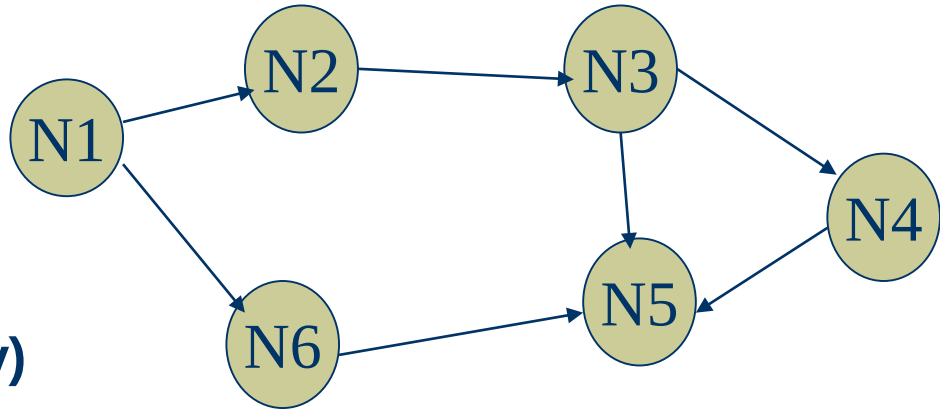
Productores/Consumidores con Pthreads

```
/* deposita 1, ..., numItems en el buffer */
void *Productor(void *arg) {
    int item;
    for (item = 1; produced <= numItems; item++) {
        sem_wait(&vacio);
        datos = item;
        sem_post(&lleno);
        pthreads_exit();        }
}

/* busca numItems items en el buffer y los suma */
void *Consumidor(void *arg) {
    int total = 0, item, aux;
    for (item = 1; item <= numItems; item++) {
        sem_wait(&flleno);
        aux = dato;
        sem_post(&vacio);
        total = total + datos; }
     printf("TOTAL: %d\n", total);
     pthreads_exit();
}
```

Casos problema de interés: 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)



Process N [i:1..6] {
esperar a predecesores (si hay)
ejecutar la tarea
señalizar a sucesores (si hay) }

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

Casos problema de interés: 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 por 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.

Casos problema de interés: 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?

Casos problema de interés: El 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).

Casos problema de interés:
El 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 C/S, 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?

Casos problema de interés: El puente de una sola 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?

Casos problema de interés: Search – Insert – Delete sobre una lista con procesos concurrentes

Una generalización de la EM selectiva e/ 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

Casos problema de interés: 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 ?

Casos problema de interés: 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.

Tareas propuestas

Investigar los monitores como herramienta de sincronización entre procesos

Buscar información sobre problemas clásicos de sincronización entre procesos y su resolución con monitores.