

Programación Concurrente ATIC

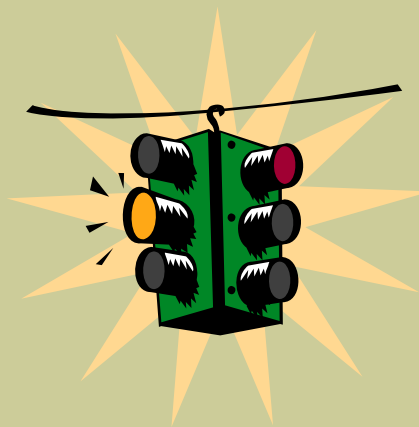
Redictado de Programación Concurrente

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 \Rightarrow instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones (métodos) atómicas: ***P*** y ***V***.

Internamente el valor de un semáforo es un entero *no negativo*:

- ***V*** \rightarrow Señala la **ocurrencia de un evento** (incrementa).
 - ***P*** \rightarrow Se usa para **demorar** un proceso **hasta que ocurra un evento** (decrementa).
- Analogía con la sincronización del tránsito para evitar colisiones.
 - Permiten proteger *Secciones Críticas* y pueden usarse para implementar *Sincronización por Condición*.

Operaciones Básicas

- **Declaraciones**

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

sem mutex = 1;

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

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

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

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

- **Semáforo binario**

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

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

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

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

Problemas básicos y técnicas

Sección Crítica: *Exclusión Mutua*

```
bool lock=false;

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

Cambio de
variable



```
bool free = true;

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

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

```
int free = 1;

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



```
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

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

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

Es más simple que las soluciones *busy waiting*.

¿Y si inicializo free= 0?

Problemas básicos y técnicas

Barreras: señalización de eventos

- **Idea:** un semáforo para cada *flag* de sincronización. Un proceso setea el *flag* ejecutando *V*, y espera a que un *flag* sea seteado y luego lo limpia ejecutando *P*.
- **Barrera para dos procesos:** necesitamos saber cada vez que un proceso llega o parte de la barrera \Rightarrow *relacionar los estados de los dos procesos*.

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

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

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

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

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

Problemas básicos y técnicas

Productores y Consumidores: *semáforos binarios divididos*

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

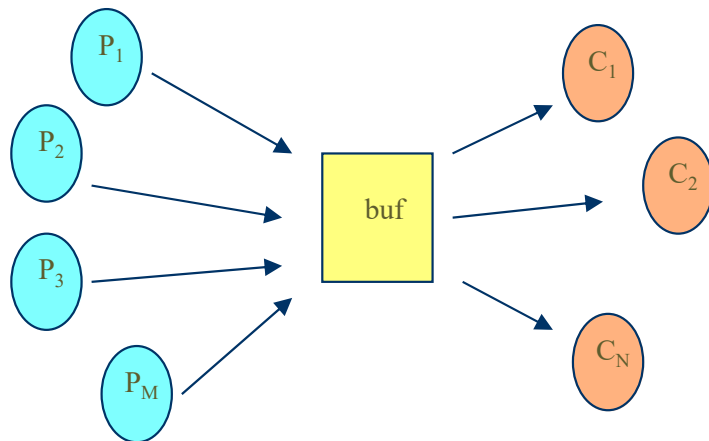
$$SPLIT: 0 \leq b_1 + \dots + b_n \leq 1$$

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

Problemas básicos y técnicas

Productores y Consumidores: *semáforos binarios divididos*

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



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

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

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

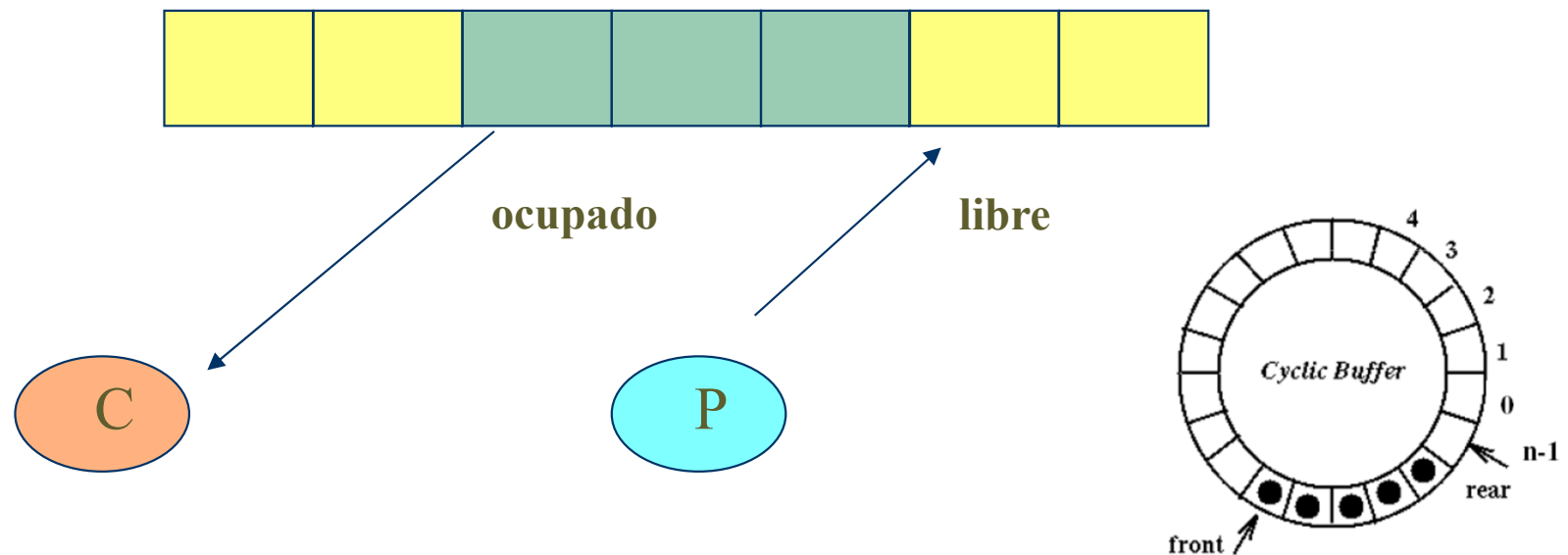
vacio y *lleno* (juntos) forman un “*semáforo binario dividido*”.

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

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

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



Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

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

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

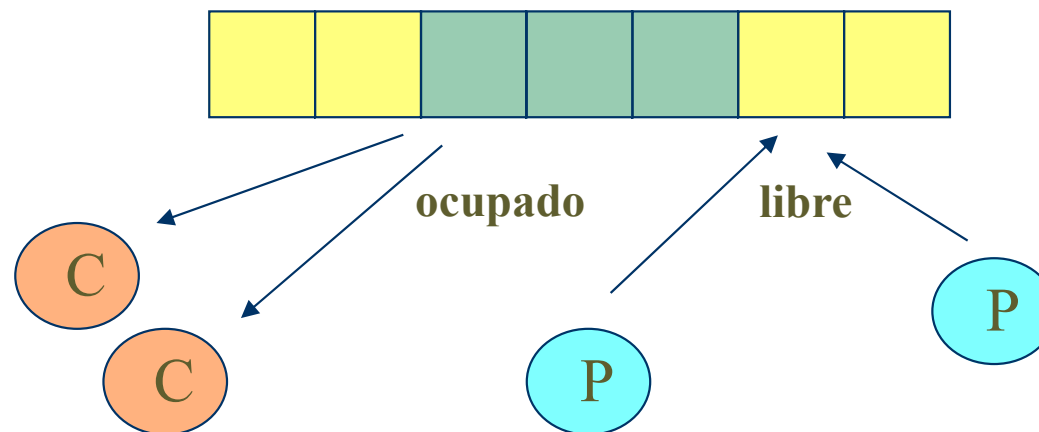
process Consumidor
{ while(true)
  { P(lleno); resultado = buf[ocupado]; ocupado = (ocupado+1) mod n; V(vacio); #retirar
    consumir mensaje resultado
    ...
  }
}
```

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

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

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



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

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

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

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

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

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

```
{ while(true)
```

```
  { producir mensaje datos
```

```
    P(vacio);
```

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

```
    V(lleno);
```

```
  }
```

```
}
```

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

```
{ while(true)
```

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

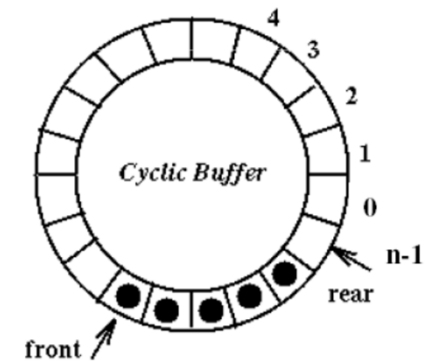
```
    P(mutexR); resultado = buf[ocupado]; ocupado = (ocupado+1) mod n; V(mutexR);
```

```
    V(vacio);
```

```
    consumir mensaje resultado
```

```
  }
```

```
}
```



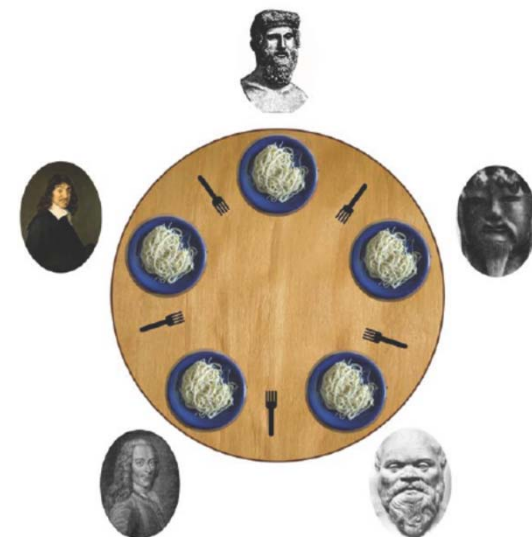
Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

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

- *Problema de los filósofos:*

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



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

Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

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

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

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


Problemas básicos y técnicas

Alocación de Recursos y Scheduling: *Passing de Baton*

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): $\langle \text{await (request puede ser satisfecho) tomar unidades;} \rangle$

release (parámetros): $\langle \text{retornar unidades;} \rangle$

Problemas básicos y técnicas

Alocación de Recursos y Scheduling: *Passing de Baton*

- 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): $\langle \text{await (libre) libre} = \text{false}; \rangle$

release (): $\langle \text{libre} = \text{true}; \rangle$



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: *Passing de Baton*

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

request (tiempo, id): $\langle \text{await (libre and } miTurno) \text{ libre} = \text{false}; \rangle$

release (): $\langle \text{libre} = \text{true}; \rangle$

request

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

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

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.

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?

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