Clase 3

Semáforos

- Descritos por Dijkstra, son una instancia de un tipo de datos abstracto con solo 2 operaciones atómicas, P y V.
- Internamente su valor es un entero no negativo.
- V: Señala la concurrencia de un evento (incrementa).
- P: Se usa para demorar un proceso hasta que ocurra un evento (decrementa).

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;`
    - V en los semáforos binarios se vuelven bloqueantes.

Técnicas y problemas básicos

Sección Crítica: Exclusión Mutua

```
sem free= 1;
process SC[i=1 to n]{
      while (true){
```

```
            P(free);
            sección crítica;
             V(free);
             sección no crítica;
      }
}
```

A la sección crítica solo accede un 1 proceso (debido al free en 1 y las operaciones de semáforo). Tener en cuenta que iniciarlo en 0 no llevará a que ningún proceso pueda entrar.

Barreras: señalización de eventos

- Usamos un semáforo para cada flag de sincronización. Un proceso setea la flag con V, y espera a que una flag sea seteado y luego lo limpia con P.
- Para hacer una barrera de 2 procesos debemos saber cuando llega o parte de la barrera.

Semáforo de señalización: Inicializamos las flags en 0, y cuando llegan a las barreras con V las aumentamos y esperamos.

```
sem llega1=0, llega2=0;
process Worker1
{ ..........
V(llega1); P(llega2);
……..
}
process Worker2
{ ..........
V(llega2); P(llega1);
……..
}
```

Usando este **método para 2** procesos se puede hacer una **butterfly barrier** para **N**.

Productores y Consumidores: semáforos binarios divididos

El **Semáforo Binario Dividido (SBS)** es una técnica de sincronización donde un semáforo binario único se divide en múltiples semáforos binarios (b1, b2, ..., bn). El **invariante global SPLIT** establece que la suma de los valores de estos semáforos está siempre entre 0 y 1:

- $0 \leq b1 + b2 + ... + bn \leq 1$

Esto asegura que los semáforos trabajan de forma coordinada para gestionar la exclusión mutua. La ejecución de los procesos en esta técnica comienza con una operación **P (wait)** sobre uno de los semáforos y finaliza con una operación **V (signal)** sobre otro. Las instrucciones entre estas dos operaciones se ejecutan con **exclusión mutua**, evitando conflictos en secciones críticas.

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
 …
 }
}
```

En este ejemplo vemos cómo podemos hacer para que varios procesos (unos que sacan y otros que quitan) accedan de forma alternada (entre tipos de proceso) al buffer de datos.

Buffers Limitados: Contadores de Recursos

Cada semáforo cuenta la cantidad de recursos libres.

```
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
 …
 }
}
```

Ejemplo con múltiples procesos:

```
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
 }
}
```

Además de tener que manejar el tema del recurso acá debemos acceder por exclusión mutua para la cual compiten con procesos del mismo tipo.

Varios procesos compitiendo por varios recursos compartidos

Todos los recursos que se vayan a usar necesita estar lockeados para poder usarlos por el proceso y esto lleva a deadlock

Problema de los filósofos: exclusión mutua selectiva

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

El problema en este caso es que cada tenedor es un SC solo levantable por un filósofo a la vez. **Los tenedores en su conjunto son** representables por **un arreglo de semáforos**.

Cada filósofo necesita el tenedor izquierdo y el derecho.

```
sem tenedores [5] = {1,1,1,1,1};
process Filósofos[i = 0..3]
{ while(true)
 { P(tenedor[i]); P(tenedor[i+1]);
comer;
V(tenedor[i]); V(tenedor[i+1]);
 }
```

```
}
process Filósofos[4]
{ while(true)
 { P(tenedor[0]); P(tenedor[4]);
comer;
V(tenedor[0]); V(tenedor[4]);
 }
}
```
La gran idea en este caso es romper la idea circular para evitar el deadlock. Es decir alcanza con que uno de los filósofos es que agarre en orden inverso.

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.

Lectores y escritores: como problema de exclusión mutua

```
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);
 }
}
```
Funca para los escritores, pero no permite varios lectores y reduce la concurrencia. La gracias sería que si un lector se metió a leer, que se metan todos los lectores, y luego al salir dejen pasar al escritor.
Solución de grano grueso: Un **lector** si quiere entrar **aumenta la cantidad de lectores esperando entrar**, **si** recién **es el primero** entonces ahí **hacemos P**, **sino entra directamente**.
```
int nr = 0; # número de lectores activos
sem rw = 1; # bloquea el acceso a la BD

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);
}
}
```

Grano fino: introducimos un semáforo para exclusión mutua que solo se usa del lado de los lectores:

- Hay un semáforo sólo para el tema de aumentar y disminuir el número de lectores.
- Hay otro semáforo para ver si pueden entrar al db o no.

```
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);
}
```

```
}
```

La solución tiene preferencia para los lectores, no es fair.

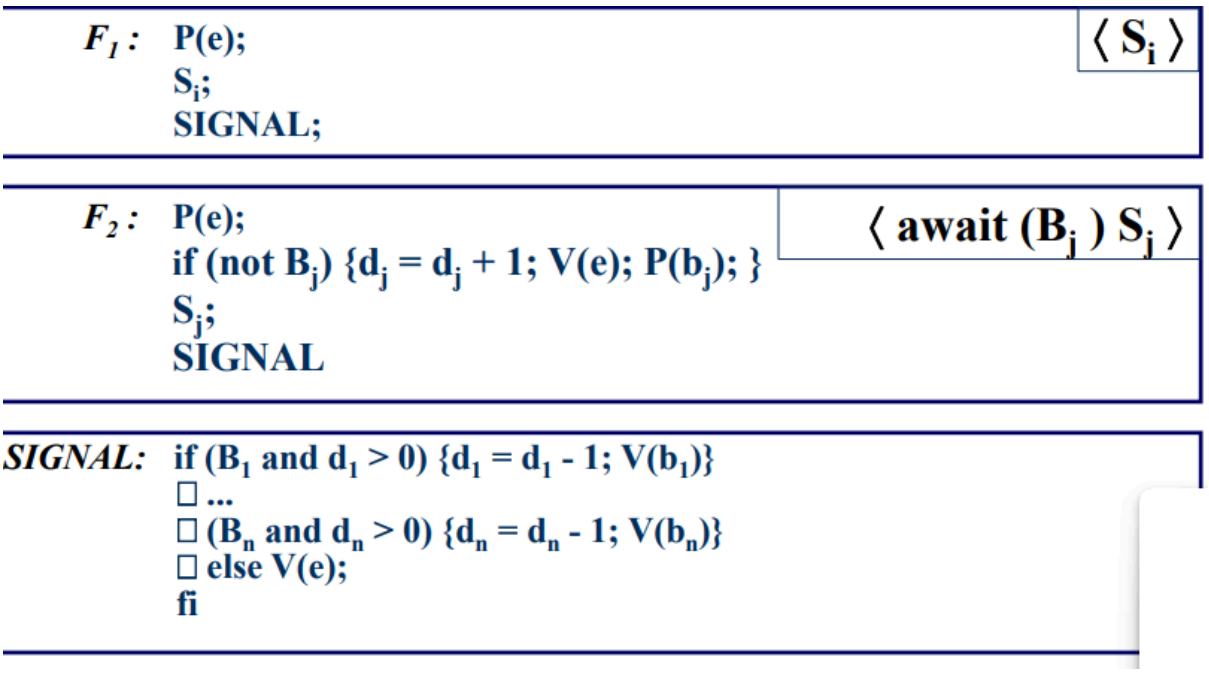
Técnica Passing the Baton

Passing the baton: técnica general para implementar sentencias await.
Proceso mantiene el "bastón" que sería un token que indica su permiso a pasar, y cuando llega a un SIGNAL (sale de la SC) pasa el bastón al siguiente o lo deja libre.

F1 : <Si> o F2 : <await (Bj) Sj>

e -> Semáforo binario inicialmente 1 para poder acceder a la sección crítica (para ver quien puede seguir).
bj -> Semáforo para cada condición de espera
dj -> Contador de procesos dormidos en el semáforo **bj** (inician en 0).

e y los **bj** se usan para formar un semáforo binario dividido: a lo sumo uno a la vez es 1, y cada camino de ejecución empieza con un **P** y termina con un único **V**.




```
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
;
}
}

SIGNALi 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

```
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); // Intento obtener acceso exclusivo para modificar las
variables compartidas.
    if (nw > 0) { // Si hay un escritor activo
      dr = dr + 1; // Incrementó el número de lectores en espera.
      V(e); // Libero la exclusión mutua.
      P(r); // Espero hasta que se me permita leer.
    }
    nr = nr + 1; // Incrementó el número de lectores activos.

    if (dr > 0) { // Si hay lectores esperando.
      dr = dr - 1; // Decrementar el contador de lectores en espera.
      V(r); // Liberó a otro lector.
    } else {
      V(e); // Si no hay lectores esperando, liberó la exclusión
mutua.
    }
```

```
    lee la BD; // Realizó la lectura de la base de datos.

    P(e); // Bloqueo para modificar las variables compartidas
después de leer.
    nr = nr - 1; // Decremento el número de lectores activos.

    if (nr == 0 and dw > 0) { // Si soy el último lector y hay
escritores esperando.
      dw = dw - 1; // Decrementar el contador de escritores en
espera.
      V(w); // Permito a un escritor empezar a escribir.
    } else {
      V(e); // Si no hay escritores esperando, libero la exclusión
mutua.
    }
  }
}


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);
     }
}
```

La explicación de lo anterior:

- **nr**: Número de lectores activos (que están leyendo la base de datos).
- **nw**: Número de escritores activos (que están escribiendo en la base de datos).
- **dr**: Número de lectores esperando para leer (bloqueados esperando que un escritor termine).
- **dw**: Número de escritores esperando para escribir.
- **e**: Semáforo de exclusión mutua (se asegura que ciertas secciones de código se ejecuten de manera crítica).
- **r**: Semáforo para lectores (permite a los lectores esperar si hay un escritor activo).

- **w**: Semáforo para escritores (permite a los escritores esperar si hay lectores o escritores activos).
- **Inicio del lector**:
    - El lector obtiene acceso a la sección crítica (P(e)), es decir, a las variables compartidas.
    - Si hay un escritor activo (nw > 0), el lector incrementa dr (contador de lectores esperando), libera la exclusión mutua (V(e)) y espera a ser notificado mediante el semáforo de lectores r (P(r)).
    - Si no hay escritores, el lector simplemente incrementa nr (número de lectores activos) y continúa.
- **Lectura de la BD**:
    - Una vez que tiene acceso, incrementa el número de lectores activos (nr = nr + 1). Si hay otros lectores esperando, se les permite continuar (V(r)).
    - Después de terminar de leer, el lector decrementa nr (número de lectores activos).
    - Si el lector es el último en salir (nr == 0), y hay escritores esperando, cede el control a un escritor (V(w)).
- **Inicio del escritor**:
    - El escritor entra en la sección crítica (P(e)) para modificar las variables compartidas.
    - Si hay lectores o escritores activos (nr > 0 or nw > 0), incrementa el contador de escritores esperando (dw = dw + 1), libera la exclusión mutua (V(e)) y espera en el semáforo de escritores (P(w)).
    - Si no hay lectores ni escritores activos, incrementa nw (número de escritores activos) y comienza a escribir.
- **Escritura en la BD**:
    - El escritor escribe en la base de datos.
    - Al terminar de escribir, decrementa el número de escritores activos (nw = nw - 1).
- **Salida del escritor**:
    - Si hay lectores esperando, les cede el control para que empiecen a leer (V(r)).
    - Si hay escritores esperando, les cede el control a ellos (V(w)).
    - Si no hay procesos esperando, libera el semáforo de exclusión mutua (V(e)).

Alocación de Recursos y Scheduling

El problema trata de gestionar el acceso de varios procesos a recursos compartidos.

Resumen:

- **Problema**: Decidir cuándo un proceso puede acceder a un recurso compartido.
- **Recurso**: Objeto o dato que puede estar libre o en uso, y por el cual los procesos compiten.

- **Request**: Solicitud para tomar unidades de un recurso; si no están disponibles, el proceso espera.
- **Release**: Devolver las unidades usadas para que otros procesos puedan tomarlas.

Solución: Passing the Baton

1. **Request**: El proceso solicita el recurso. Si no está disponible, espera. Si lo está, lo toma y notifica a otros.
2. **Release**: El proceso libera el recurso y notifica que está disponible para otros.

Esto permite una gestión eficiente y ordenada de recursos compartidos.

Alocación Shortest-Job-Next (SJN)

Comportamiento:

1. **Request (tiempo, id)**:
   - Si el recurso está libre, se asigna inmediatamente al proceso .
   - Si el recurso está ocupado, el proceso debe esperar.
2. **Release**:
   - Cuando el recurso se libera, se asigna al proceso en espera con el menor valor de tiempo.
   - Si varios procesos tienen el mismo tiempo, el recurso se asigna al que ha esperado más.

Ventajas:

- **Minimización del tiempo promedio de ejecución**: SJN reduce el tiempo de espera global al atender primero los procesos más cortos.

Desventajas:

- **Unfair (injusto)**: Los procesos largos pueden quedar en espera indefinidamente si constantemente hay procesos cortos. Este problema puede solucionarse con **aging**, donde se da preferencia a procesos que han esperado mucho tiempo.

En un sistema de planificación SJN (Shortest Job Next):

- **Proceso que invoca a request**: Debe esperar hasta que el recurso esté disponible y su solicitud sea la próxima en ser atendida según la política SJN. El tiempo solo influye si un proceso debe esperar.

```
1. request (tiempo, id):
2. P(e);
3. if (not libre) DELAY;
4. libre = false;
```

```
5. SIGNAL;


6. release ( ):
7. P(e);
8. libre = true;
9. SIGNAL;
```

En DELAY:

- El proceso inserta sus parámetros en una lista de espera (pares).
- Libera la Sección Crítica (SC) ejecutando .
- Se bloquea en un semáforo hasta que su solicitud pueda ser atendida.

En SIGNAL:

- Al liberar el recurso, si la lista de espera no está vacía, el recurso se asigna al proceso con el menor tiempo de ejecución (según SJN).

Cada proceso se retrasa dependiendo de su posición en la lista de espera, bloqueándose en el semáforo .

```
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.

```
bool libre = true; Pares = set of (int, int) = ; sem e = 1, b[n] = ([n] 0);
Process Cliente [id: 1..n]
{ int sig;
```

```
//Trabaja
tiempo = //determina el tiempo de uso del recurso//
P(e);
if (! libre) { insertar (tiempo, id) en Pares;
V(e);
P(b[id]);
}
libre = false;
V(e);
//USA EL RECURSO
P(e);
libre = true;
if (Pares   ) { remover el primer par (tiempo, sig) de Pares;
V(b[sig]);
}
 else V(e);
}
```