Resumen Semáforos - Concurrente

Defectos de la Sincronización por Busy Waiting

- Los protocolos que usan esta sincronización son complejos y no tienen una clara separación entre variables de sincronización y las usadas para computar resultados.
- Es difícil de diseñar estos protocolos para probar la corrección del programa e incluso la verificación del mismo es compleja cuando aumenta el número de procesos.
- Es una técnica ineficiente si se la utiliza en multiprogramación.

Semáforos

- Herramienta que nos permite realizar sincronización entre procesos concurrentes. Es una instancia de un tipo de datos abstracto con sólo 2 operaciones atómicas **P y V.** Internamente el valor de un semáforo es un entero no negativo:
    - **V:** Señala la ocurrencia de un evento incrementando de forma atómica el valor interno del semáforo.
    - **P:** Se usa para demorar un proceso hasta que ocurra un evento decrementando de forma atómica el valor interno del semáforo.
- Permiten realizar sincronización por exclusión mutua (proteger secciones críticas) y por condición.

Sintaxis

- **Declaraciones - Si o si se deben inicializar en la declaración**

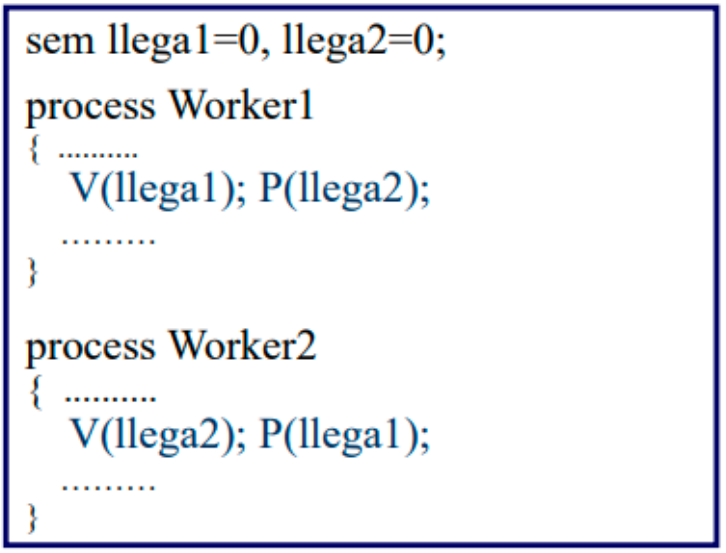
    - sem mutex = 1;
    - sem fork[5] = ([5] 1);
- **Semáforo general o counting semaphore**
    - P(s): <await [s > 0) s = s - 1;>
    - V(s): <s = s + 1;>
    - Estos son los que usamos en la práctica, Cuando el semáforo se vuelve mayor que 0, cualquier proceso puede pasar, no respeta un orden de llegada.
        - Tenemos riesgo de que no se respete la propiedad de Eventual Entrada.
    - Si la implementación de la demora por operaciones **P** se produce sobre una cola, entonces las operaciones si son fair.
- **Semáforo binario**
    - P(b): <await (b > 0) b = b + 1;>
    - V(b): <await (b < 1) b = b + 1;>

Problema de la sección crítica - semáforos por exclusión mutua

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

Problema de la barreras - semáforos por señalización de eventos

- La idea es usar un semáforo para cada flag de sincronización donde un proceso setea el flag ejecutando **V**, y espera a que un flag sea seteado y luego lo limpia ejecutando **P.**
- **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);
   .........
}
```

Problema de productores y consumidores - semáforos binarios divididos

- **Semáforo Binario Dividido (Split Binary Semaphore) - SBS**
    - Los semáforos binarios (en nuestro caso semáforos generales que simulan a los binarios) b1, b2, b3, b..., bN forman un SBS en un programa si se cumple que la suma de todos los semáforos es mayor o igual que 0 y menor o igual que 1, es decir, es binaria.
    - Se puede ver como un semáforo binario que fue dividido en n partes.
    - **Tienen una implementación específica para la exclusión mutua (problema de la sección crítica)**
        - En general la ejecución de los procesos inicia con un **P** sobre un semáforo y termina con un **V** sobre otro de ellos. Esto permite que se genere una alternancia entre procesos para ejecutar la sección crítica.
- **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
      …
    }
}
```

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

Problema de buffers limitados - semáforos como contadores de recursos

- **Semáforos Contadores de Recursos**
    - Cada semáforo cuenta el número de unidades libres de un recurso determinado. Son útiles cuando los procesos compiten por recursos de múltiples unidades que serían, por ejemplo, un buffer con N posiciones limitadas.
- **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. Manejamos el buffer como una cola circular.

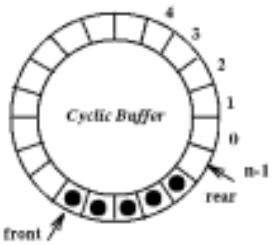
```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;
process Productor
{ while(true)
    {  …
       producir mensaje datos
       P(vacio); buf[libre] = datos; libre = (libre+1) mod n;  V(lleno); #depositar
    }
}
process Consumidor
{ while(true)
    {  P(lleno); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(vacio); #retirar
       consumir mensaje resultado
       …
    }
}
```

vacío cuenta los lugares libres, y **lleno** los ocupados
depositar y **retirar** se asumen atómicas porque solo hay un **consumidor** y un **productor**

- Con más de un productor y un consumidor, las operaciones **depositar** y **retirar** son secciones críticas y se deben ejecutar con exclusión mutua. Solución:

```
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
    }
}
```



Varios procesos compitiendo por varios recursos compartidos

- Hay varios procesos y varios recursos cada uno de ellos protegidos por un lock. Un proceso tiene que adquirir los locks de todos los recursos que requiere para poder trabajar, puede haber deadlock si 2 o más procesos pueden querer el mismo recurso.

Problema de los filósofos - semáforos con exclusión mutua selectiva

- Problema que surge si varios procesos compiten por el acceso a conjuntos superpuestos de variables compartidas.
- **Ejemplo:** Hay 5 filósofos y 5 tenedores. Cada filósofo debe comer y pensar pero para comer necesita 2 tenedores, el que está a su derecha y el que está a su izquierda pero, cada tenedor puede ser tomado por un filósofo a la vez.
    - Cada tenedor es una sección crítica, podemos representar a los tenedores como un arreglo de semáforos.
    - Levantar un tenedor es hacer un **P.**
    - Bajar un tenedor es hacer un **V.**
    - Se genera deadlock si todos los filósofos hacen exactamente lo mismo, por ejemplo, todos primero levantan su tenedor derecho y luego el izquierdo, al querer levantar el izquierdo no pueden porque es el derecho que otro filósofo ya levantó.

```
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]);
    }
}
```

Problema lectores y escritores - semáforos con exclusión mutua selectiva

- **Problema:** Dos clases de procesos (lectores y escritores) comparten una Base de Datos. El acceso de los **escritores** debe ser exclusivo para evitar interferencia entre transacciones. Los **lectores** pueden ejecutar concurrentemente entre ellos si no hay **escritores** actualizando.
    - Los procesos son asimétricos y, según el scheduler, con diferente prioridad

Solución como un problema de exclusión mutua

- Los **escritores** necesitan acceso exclusivo a la base de datos.
- Los **lectores** como grupo necesitan acceso exclusivo con respecto a los **escritores**.
- **Esta solución le da preferencia a los lectores → no es fair.**

```
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);
      }
}
```

Solución como un problema de sincronización por condición

- Cuando las condiciones de espera de los procesos son diferentes pero además son superpuestas, no es viable plantear una solución solo con el uso de semáforos, necesitamos usar la técnica **Passing the Baton.**
- **Técnica Passing the Baton**
    - Técnica que emplea un SBS para brindar exclusión (en este caso a las variables que van a controlar el acceso a la base de datos) y despertar procesos demorados siguiendo algún criterio.
    - Puede usarse para implementar cualquier **await.**
    - Cuando un proceso está dentro de la SC, mantiene el **baton** que significa permiso para ejecutar.
    - Cuando el proceso llega a un **SIGNAL** (sale de la SC), pasa el **baton** a otro proceso. Si ningún proceso está esperando por el **baton** (nadie quiere entrar a la SC) el **baton** se libera para que lo tome el primer proceso que quiera entrar a la SC.
    - Esta técnica puede no cumplir la **eventual entrada** ya que el **baton** no se pasa a un proceso en particular, sino que es no determinístico, por lo tanto, puede quedar algún proceso colgado. Se realiza un orden por **tipo de proceso** y no **por proceso específicamente.**
    - La sincronización se expresa con sentencias atómicas de la forma:
        - **Incondicional** → F_1: <S_I>
        - **Condicional** → F_2: <await (B_J) S_J>
    - **Componentes que necesita la técnica:**
        - Un semáforo “e” inicialmente en 1 que controla la entrada a la SC.
        - Un semáforo “b_j” inicialmente en 0, asociado con cada condición de espera.

        - Por cada semáforo "b_j" un contador "d_j" inicializado en 0, que cuenta la cantidad de procesos dormidos en el semáforo "b_j" esperando que la condición se vuelva verdadera.
        - El conjunto de los semáforos "e" y todos los "b_j" forman un SBS.
    - Traducción de las sentencias:
        - **F_1:**
            - P(e);
              S_i;
              SIGNAL;
        - **F_2:**
            - P(e);
              if (not B_i) {
                  $d_j = d_j + 1$;
                  V(e);
                  $P(b_j)$;
              }
              S_j;
              SIGNAL;
        - **SIGNAL:**
            - if (B_1 and $d_1 > 0$) {
                  $d_1 = d_1 - 1$;
                  $V(b_1)$
              }
              [] ...
              elif (B_n and $d_n > 0$) {
                  $d_n = d_n - 1$;
                  $V(b_n)$;
              }
              else V(e);
              fi

```
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);
   }
}
```

Esta solución sigue dando preferencia a los **lectores** pero estas preferencias se pueden modificar al cambiar las condiciones de los if's

Alocación de Recursos y Scheduling

- **Problema:** Decidir cuándo se le puede dar acceso a un recurso a un proceso en específico. Este recurso puede ser un objeto, elemento, componente, SC, etc. que hace que el proceso se demore al esperar adquirirlo.
- **Definición del Problema:** Procesos que compiten por el uso de unidades de recurso compartido (cada unidad **está libre** o en **uso**)
    - El proceso tiene que hacer un **request(parámetros)** del/los recurso/s, **usar** el/los recurso/s y por último un **release(parámetros)** del/los recurso/s.
- **Podemos hacer uso de la técnica Passing the Baton**

```
request (parámetros): P(e);
                      if (request no puede ser satisfecho) DELAY;
                      tomar las unidades;
                      SIGNAL;

release (parámetros): P(e);
                      retornar unidades;
                      SIGNAL;
```

Alocación Shortest-Job-Next (SJN)

- Esta política determina que si hay más de un proceso esperando por un recurso compartido, lo usará el que lo tenga que usar por menos tiempo.
- **Requerimiento del recurso:**
    - request(tiempo_estimado_de_uso, id)
        - Si el recurso está libre, es alocado inmediatamente al proceso **id**; sino, el proceso **id** se demora.

- **Release del recurso:**
  - release()
    - Cuando se libera el recurso, 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.
- Esta política minimiza el tiempo promedio de ejecución pero es **unfair**. Se puede mejorar usando **aging**.

```
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);
}
```

Si quisiéramos **respetar el orden de llegada**, deberíamos insertar los elementos en una cola y no haría falta enviar el tiempo de uso por parámetro en la request.
Si tuviéramos **recursos de múltiples unidades**, podríamos llevar un contador de recursos libres y una estructura que guarde los recursos disponibles para ir sacando y reponiendo.