Semáforos 🚦

Los semáforos son herramientas que nos permiten realizar sincronización entre procesos concurrentes.

Un semáforo no es más que una instancia de un tipo de dato abstracto (o un objeto) con sólo 2 operaciones (métodos) **atómicas** : P y V

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

- P -> Se usa para demorar un proceso hasta que ocurra un evento (decrementa).
- V -> Señala la ocurrencia de un evento (incrementa).
```

Permiten proteger Secciones Criticas y pueden usarse para implementar Sincronización por Condición.

Sintaxis

Declaraciones :

- "sem mutex = 1" El valor debe ser >= 1
- "sem fork[5] = ([5] 1)" Declaramos un arreglo de semáforos inicializados en uno.

Semáforos generales (Los usamos en la practica) :

- 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>

Técnicas y algoritmos

Sección critica - Exclusión Mutua.

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

- Este es el uso de semáforos, la variable debe estar inicializada en 1 así puede entrar en P y poder realizar las operaciones. Su uso con sentencias await es el siguiente :

```
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;
      }
}
```

- Este seria el código equivalente con sentencias await al uso de semáforos, internamente suma y resta 1 a la variable free. Por eso es que la variable puede tomar valores >= a uno.

Nota : Para entenderlo mejor es que la P lo que hace es esperar que se cumpla el await para entrar, internamente le resta 1 para que no diga que el recurso esta ocupado (eso no lo vemos), cuando salimos con V liberamos el recurso (internamente le suma uno).

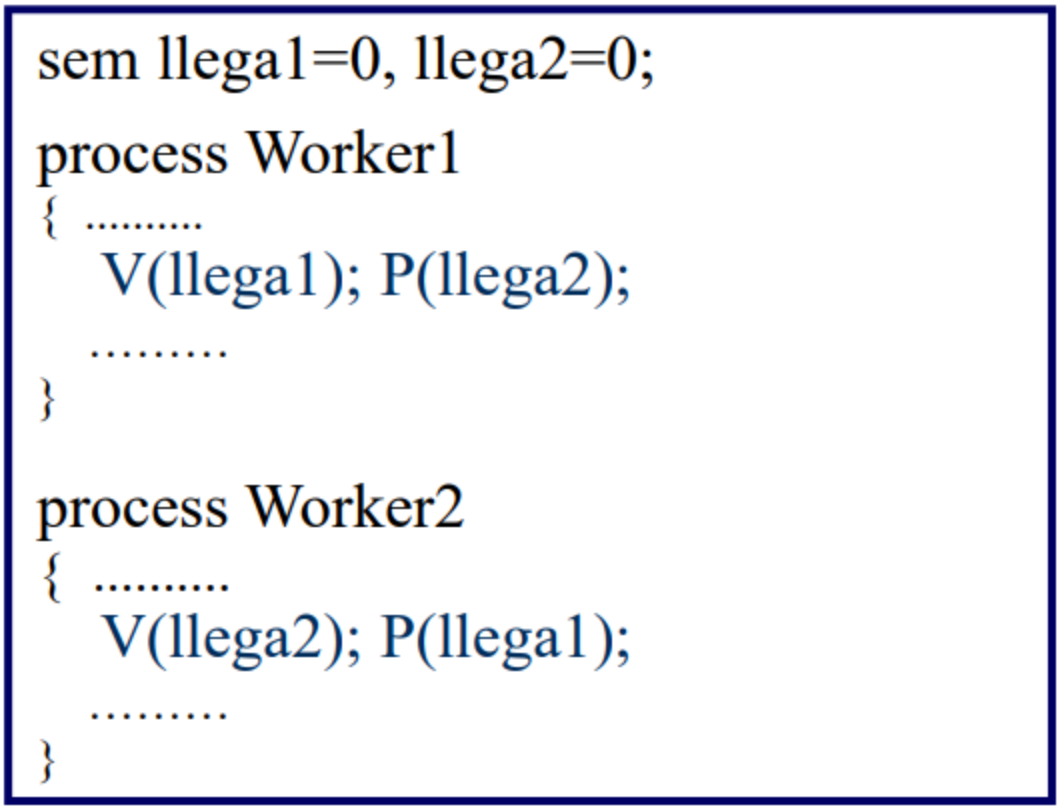
Barreras : Señalización de eventos

la idea es 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.

- **Usamos un semáforo en cero para señalizar !**

Barrera para dos procesos :

- Semáforo de señalización : generalmente inicializados en 0. Un proceso señala el evento con V(); otros procesos esperan la ocurrencia del evento ejecutando P().

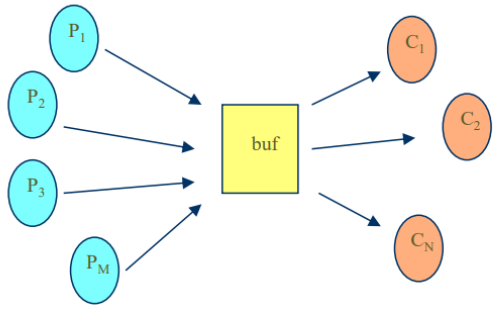


- Si primero se pone P() queda trabado.

Productores y consumidores : semáforos binarios divididos

El producto-consumidor con semáforos binarios es una forma de sincronizar procesos, el cual el producto produce datos y los deposita en un buffer y el consumidor espera a que haya algo en el buffer para poder consumirlo. Todo esto sin preguntar por como esta el buffer, sino que sincronizando con semáforos binarios.

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
       …
    }
}
```

- Para maximizar la concurrencia, no se debe consumir al momento de leer. Se carga en un variable, se libera el recurso y se consume luego de esto.
- El productor NUNCA pregunta si el buffer esta lleno.
- El consumidor NUNCA pregunta si el buffer es vacío.
- Si se hace esto mencionado estaremos haciendo BUSY WAITING, lo cual esta mal.

Buffers limitados : Contadores de recursos

En ejemplo anterior vimos como es con un buffer que posee un recurso, ahora vemos un buffer que posee N recursos.

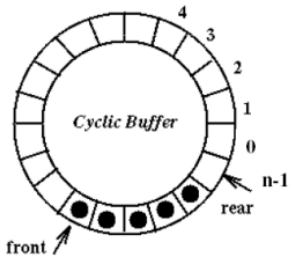
Este ejemplo solo con un consumidor y un productor.

```
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
      …
    }
}
```

- Ahora tenemos que vacío valdrá N en lugar de 1.
- Vacío cuenta los lugares libres y lleno los ocupados.
- Depositar y retirar se pudieron asumir atómicas pues solo hay un productor y un consumidor.

```
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
```

```
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
    }
}
```



Técnica Passing the baton

- Debe respetarse el orden de llegada.... Cuando dice eso el enunciado debemos usar esta técnica.