

# SEMAFOROS

## INTRODUCCION

El primer avance fundamental en el tratamiento de los problemas concurrentes ocurre en 1965 con el tratado de Dijkstra, que estaba involucrado en el diseño de un sistema operativo como una colección de procesos secuenciales cooperantes y con el desarrollo de mecanismos eficientes y fiables para dar soporte a la cooperación. Estos mecanismos podrían ser fácilmente usados por procesos de usuario si el procesador y el sistema operativo colaborarán en hacerlos disponibles.

El principio fundamental es: dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar determinado hasta que haya recibido una señal específica.

Para la señalización se utilizan variables especiales llamadas «semáforos». Para transmitir una señal vía el semáforo **s**, el proceso ejecuta la primitiva **semSignal(s)**. Para recibir una señal vía el semáforo **s**, el proceso ejecutará la primitiva **semWait(s)**.

Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que tiene un valor entero sobre el cual sólo están definidas tres operaciones:

1. Un semáforo puede ser inicializado a un valor no negativo.
2. La operación **semWait** decrementa el valor del semáforo. Si el valor pasa a ser negativo entonces el proceso que está ejecutando el **semWait** se bloquea.
3. La operación **semSignal** incrementa el valor del semáforo.

Las primitivas **semWait** y **semSignal** se asumen atómicas. Podemos ver una definición formal de las primitivas del semáforo:

```
struct semaphore{  
  
    int cuenta;  
  
    queueType cola;  
  
    }
```

```
Void semWait(semaphore s){
```

```
    s.cuenta--;
```

```
    if(s.cuenta)<0{
```

```
        ponerProcesoEn_s.cola;
```

```
        bloquearProceso;
```

```
    }
```

```
}
```

```
Void semSignal(semaphore s){
```

```
    s.cuenta++;
```

```
    if(s.cuenta >=0){
```

```
        extraerProcesoDe_s.cola;
```

```
        ponerProcesoEnListos;
```

```
    }
```

```
}
```

Este tipo de semáforos se los conoce como **semáforos con contador**

Existe una versión más restringida de los semáforos, conocida como **semáforo binario** o **mutex**, definido de la siguiente manera:

```
    struct b_sem {
```

```
        enum {cero, uno} valor;
```

```
        queueType cola;
```

```
    }
```

```
void semWaitB(b_sem s){
```

```
    if(s.valor == 1)
```

```
        s.valor = 0;
```

```
    else{
```

```
        ponerProcesoEn_s.cola;
```

```
        bloquearProceso;
```

```
    }
```

```
}
```

```
void semSignalB(b_sem s){
```

```
    if(estaVacia(s.cola))
```

```
        s.valor = 1;
```

```
    else{
```

```
        extrarProcesoDe_s.cola;
```

```
        ponerProcesoEnListos;
```

```
    }
```

```
}
```

Un semáforo binario sólo puede tomar los valores 0 y 1 y se puede definir las siguientes operaciones:

1. Un semáforo binario sólo puede ser inicializado en 0 o 1
2. Las operaciones **semWaitB** comprueba el valor de semáforo. Si el valor es 0, entonces el proceso que está ejecutando el **semWaitB** se bloquea.
3. La operación **semSignalB** comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en la operación **semWaitB**. Si no hay, entonces el valor del semáforo se pone a uno.

### EJEMPLO DE SEMAFORO

```
/* programa exclusión mutua */  
  
semaphore s.cuenta = 1;  
  
void p(){  
    while(true){  
        semWait(s);  
  
        /* sección crítica */  
        semSignal(s);  
  
        /* resto */  
    }  
}
```

Consideremos **n** procesos, los cuales necesitan todos acceder al mismo recurso. Cada proceso ejecuta el **semWait(s)** justo antes de entrar a su sección crítica. Si el valor de **s** pasa a ser negativo, el proceso se bloquea. Si el valor es 1, entonces decrementa a 0 y el proceso entra a su s.c. inmediatamente.

Este ejemplo puede servir igualmente si el requisito es que se permita más de un proceso en su sección crítica a la vez. Este requisito se cumple simplemente inicializando el

semáforo al valor especificado. Así, el valor de `s.cuenta` puede ser interpretado como sigue:

- **`s.cuenta >= 0`**: `s.cuenta` es el número de procesos que pueden ejecutar **`semWait(s)`** sin suspensión. Tal situación permitirá a los semáforos admitir sincronización así como exclusión mutua.
- **`s.cuenta < 0`**: la magnitud de `s.cuenta` es el número de procesos suspendidos en `s.cola`.

## PRODUCTOR – CONSUMIDOR

Veremos uno de los problemas más comunes enfrentados en la programación concurrente: el problema productor/consumidor.

### El enunciado:

«Hay uno o más procesos generando algún tipo de dato y poniéndolos en un **buffer**. Hay un único consumidor que está extrayendo datos de dicho **buffer** de uno en uno.»

El sistema está obligado a impedir la superposición de las operaciones sobre los datos. Es decir, sólo un agente (productor o consumidor) puede acceder al **buffer** en un momento dado.

**Veremos algunas soluciones...**

**Analicemos el siguiente código:**

```
#define BUFFER_SIZE 100 int cantItems = 0;





void productor(){
    while(1){
        int item = producirItem();
        if(cantItems == BUFFER_SIZE)
            sleep();
        ponerItemBuffer(item);
        cantItems++;
        if(cantItems == 1)
            wakeup(consumidor);
    }
}

void consumidor(){
    while(1){
        if(cantItems == 0)
            sleep();
        int item = quitarItemBuffer();
        cantItems--;
        if(cantItems == BUFFER_SIZE-1)
            wakeup(productor);
    }
}
```

Podemos ver como el productor introduce datos en un buffer y cómo el consumidor los extrae. Tenemos dos situaciones:

1. Cuando `cantItems` valga 0, el consumidor no tendrá items para extraer y consumir, por lo tanto ejecutará la instrucción **sleep** y se bloqueará. Cuando el productor ingrese un nuevo item al buffer ejecutará la instrucción **wakeup** despertando así, al consumidor.
2. Cuando `cantItems` valga el tamaño máximo del buffer (100), el productor no tendrá más espacio para seguir produciendo por lo que ejecutará la instrucción **sleep** y se bloqueará. Cuando el consumidor extraiga un elemento del buffer, mandará una señal a través de **wakeup** para despertar al productor y que éste pueda seguir produciendo.

Pareciera una solución valida, pero tiene un problema muy grande. Veamos la siguiente traza de ejecución:

	<b>cantItems = 0;</b>	
	<b>Productor</b>	<b>Consumidor</b>
0		if(cantItems == 0) 
1	item=producirItem();	
2	if(cantItems==BUFFER_SIZE) 	
3	ponerItemBuffer(item);	
4	cantItems++;	
5	if(cantItems==1) 	
6	wakeup(consumidor);	
7		sleep();
	<b>AMBOS QUEDAN BLOQUEADOS</b> 	

Se podría solucionar el problema planteado anteriormente implementando semáforos:

```
#define BUFFER_SIZE 100

semaphore huecos = BUFFER_SIZE;

semaphore elementos = 0;

semaphore mutex = 1;
```

Consideremos 3 semáforos que son declarados de manera global. Los semáforos elementos y huecos funcionan como **semáforos con contador** y el semáforo mutex como **semáforo binario**.

```
Void productor(){
    While(1){
        Int item = producirItem();

        semWait(huecos);

        semWait(mutex);

        ponerItemBuffer(item);

        semSignal(mutex);

        semSignal(elementos);

    }}

```

```
Void consumidor(){
    While(1){

        semWait(elementos);

        semWait(mutex);

        int item = quitarItemBuffer();

        semSignal(mutex);

        semSignal(huecos);

        consumirItem(item);

    }}

```

## SEMAFOROS EN C

Los semáforos pueden ser implementados en C junto con los hilos para esto, debemos agregar una nueva librería: semaphore.

***#include <semaphore.h>***

Para que los semáforos sirvan para todos los hilos, deben ser declarados de manera global. Para eso usamos el tipo de dato **sem\_t**:

**sem\_t mutex;**

Los semáforos deben ser inicializados para poder ser utilizados, esta inicialización generalmente se realiza en el main:

***sem\_init(sem\_t\* sem, int pshared, unsigned int value);***

**Donde:**

- **sem\_t\* sem:** dirección de memoria del semáforo a ser inicializado
- **int pshared:** indica si el semáforo va a ser compartido por hilos de un mismo proceso o entre procesos distintos.
  - Si se pone el valor 0, el semáforo es compartido entre hilos del proceso.
  - Un valor distinto de 0 el semáforo es compartido entre procesos y el ese valor representa un bloque de memoria compartida por los procesos.
- **Unsigned int value:** valor en el cual va a ser inicializado el semáforo:
  - 0: indica que el semáforo comienza cerrado
  - 1: el semáforo comienza abierto
  - >1: se considera un semáforo con contador

### Funciones para manejo de semáforos:

- **sem\_wait(sem\_t\* sem);**
  - Equivalente al semWait() visto anteriormente, verifica el valor del semáforo, si vale 0 bloquea al proceso/hilo impidiéndole continuar. Si vale

1, cambia el valor del semáforo a 0 y lo deja continuar. Si vale >1, le resta uno al semáforo y deja pasar al proceso/hilo.

- **sem\_post(sem\_t\* sem);**
  - Equivalente al semSignal() visto anteriormente. Incrementa el valor del semáforo.
- **sem\_post\_multiple(sem\_t\* sem, unsigned int count);**
  - Para semáforos con contador, cambia el valor del semáforo por el valor contenido en el parámetro count.

#### **Algunas funciones más:**

- **sem\_destroy(sem\_t\* sem);**
  - Destruye el semáforo pasado por parámetro. Sólo un semáforo que fue inicializado con el sem\_init puede ser destruido. Destruir un semáforo en donde hay procesos o hilos bloqueados, o utilizar un semáforo previamente destruido puede provocar comportamientos indefinidos.
- **sem\_getvalue(sem\_t\* sem, int \* value);**
  - Almacena el valor actual del semáforo en el entero apuntado por value.