

Semaforos

En un entorno de multiprogramación, en donde varios procesos se ejecutan concurrentemente, y dado que los procesos pueden compartir memoria, ya sea porciones de código o alguna variable (llamado **Seccion Critica**), podemos tener inconsistencia de datos. Para poder mantener la consistencia de los mismos, uno de los mecanismos para asegurar la ejecución ordenada de procesos cooperativos son los **“semáforos”**

Un semáforo es un tipo de variable especial (Tipo de dato Abstracto), que pertenece al dominio de los enteros, esta variable es de tipo protegido y solo la maneja el **“Sistema Operativo”**

A parte de la operación de inicialización del semáforo, esta variable se manipula a través de dos operaciones atómicas bien definidas que se puede llamar de distintas maneras, dependiendo de la bibliografía.

Wait - - Signal
Down -- Up

Wait = down = P(S)

La función P es conocida como la que pide el recurso, y si no está disponible queda en un ciclo infinito, ahora si está disponible no entra al while y lo que hace es decrementar la variable **semáforo S**, esto quiere decir que toma el recurso, variable, etc o lo que sería acceder a la “Seccion Critica” hasta que lo libere con el V(S)

Signal = Up = V(S)

Esta función libera el recurso e incrementa el valor de la variable semáforo S

Inicializar_semaforo (S, valor)

```
Wait = down = P(S) :  
    while S <= 0 do  
        Begin  
            Recurso no disponible  
        End  
        S=S-1; -----> Toma el recurso  
Signal = Up = V(S) : S=S+1; ----- > Libera el recurso
```

Tipos:

Binarios (Mutex): toman valor 0 o 1

Contadores: toman valores mayores a 1, se usan cuando por ejemplo tengo 10 recursos, aunque son compartidos, cuando llega el proceso 11 este se bloquea.

Espera Activa: ...

Etc....

Usos:

- Para el manejo de secciones criticas (Sincronizacion de procesos, variables compartidas)
- Para el acceso y uso de recursos

Inicializar_semaforo (S, ?)

```
Procedure P1;  
Begin  
    While condición 1 do  
        Begin  
            Actividades Preliminares_1  
            P(Activo);  
            Seccion Critica;  
            V(Activo);  
            Otras Actividades_1  
        End  
    End  
End
```

```
Procedure P2;  
Begin  
    While condición 2 do  
        Begin  
            Actividades Preliminares_2  
            P(Activo);  
            Seccion Critica;  
            V(Activo);  
            Otras Actividades_2  
        End  
    End  
End
```

Se lanzan 2 procesos en paralelo, suponemos que S esta disponible en 1, por lo tanto cualquiera de los dos lo puede tomar, supongamos que lo toma P1 primero, como S no es ≤ 0 , lo toma y decrementa S (ahora en 0) y ejecuta su “sección critica”, ahora cuando el P2 quiere ejecutar su sesión critica se encuentra con que S es ≤ 0 por lo tanto el P2 queda en lo que se denomina “espera activa” hasta que el P1 libere el recurso e incremente el valor de S a 1.

Este sistema tiene el problema de cuando un proceso queda en espera activa sigue consumiendo CPU, por lo tanto lo que debe hacer es:

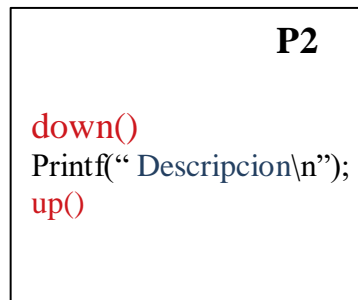
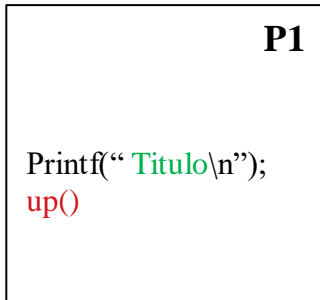
```
wait(s)  
{  
    s = s - 1; ✓  
    if (s < 0) {  
        <Bloquear al proceso>  
    }  
}  
  
signal(s)  
{  
    s = s + 1;  
    if (s <= 0)  
        <Desbloquear a un proceso bloqueado por la  
        operacion wait>  
    }  
}
```

Seria una implementación más óptima sin espera activa

Vamos a ver un ejemplo donde yo quiero sincronizar dos procesos, para que si o si uno deba ejecutarse primero que el otro, Por ejemplo debo imprimir un documento, donde un proceso (P1) imprime el titulo, y el proceso (P2) imprime la descripción. Es evidente que primero se tiene que ejecutar el P1 y luego P2.

Como debería inicializar el semáforo para esta problemática?

Inicializar_semaforo (S, 0)



Titulo

Descripcion

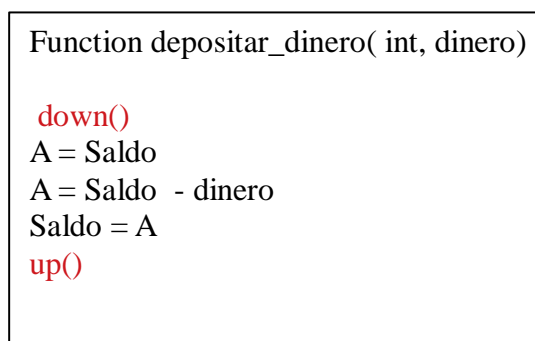
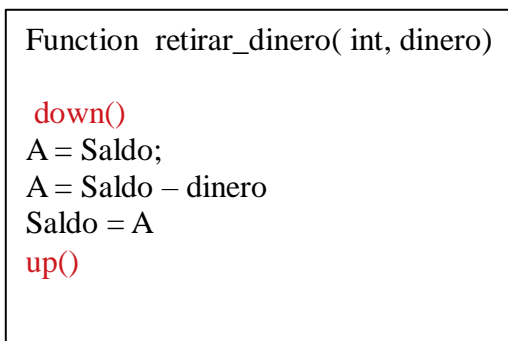
Como dijimos que también se puede usar para recursos compartidos, o variables compartidas por distintos procesos.

Para esto podemos ver un ejemplo clásico donde tenemos una cuenta bancaria, suponemos que alguien puede retirar dinero de esa cuenta (Hijo), y otra persona podría estar ingresando dinero a la misma (Padre).

Variable (compartida por dos procesos) **Saldo = 500 1100 100**

Como debería estar inicializado el semáforo para que al menos uno lo pueda ejecutar?

Inicializar_semaforo (S, 1)



Retiro = 400

Ingresa = 600

Aux = 500

Aux = 500 1100

La nomenclatura para llamar a este tipo de semáforos, se lo llama de exclusión mutua o mutex

Para resolver los ejemplos de la práctica vamos a usar lo siguiente, siempre en el lenguaje de programación ANSI C:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```



Vamos a incluir estas 2 librerías, la primera de ellas para trabajar con la creación de hilos y la segunda para semaforos

Vamos a usar hilos para trabajar con semaforos

Declaramos los hilos

```
pthread_t hilo_1, hilo_2;
```

Suponemos que vamos a crear 2 hilos de proceso

```
pthread_create(&hilo_1, NULL, *hilo_1_function, NULL);
pthread_create(&hilo_2, NULL, *hilo_2_function, NULL);
```

hilo_1: nombre del hilo, NULL: atributos del hilo (quedan por defecto)

hilo_1_function: función que se va a ejecutar al crearse el hilo, NULL: argumento de entrada a esa función

Lanzamiento en paralelo de los hilos de procesos

```
pthread_join(hilo_1, NULL);
pthread_join(hilo_2, NULL);
```


pthread_join: lanzo la ejecución en paralelo de los hilos donde el primer argumento es el nombre del hilo y el segundo (NULL) valor de retorno.

Declaración de funciones

```
static void * hilo_1_function(void* arg);
static void * hilo_2_function(void* arg);
```


Para controlar la sincronización de los 2 hilos usamos semaforos

```
sem_t S;
```



En este caso estamos declarando una variable **S** de tipo semaforo

```
sem_init(&S, 0, 1);
```



Inicializando el semaforo, lo primero (&S) es una referencia a ese semáforo, el 0 si es para procesos, en este caso no, el 1 es el valor de inicialización del semáforo.

Las dos operaciones sobre el semaforo

```
sem_wait(&S);
sem_post(&S);
```

La función que pide el recurso
la función que libera el recurso