Anteriormente vimos 4 aproximaciones diferentes a la solución y dos algoritmos Dekker- Peterson para el manejo de los procesos concurrentes y la protección del acceso de los recursos compartidos.

Dijkstra Define las reglas por las cuales se tenía que bazar el diseño de un SO, compone ciertos mecanismos de desarrollo para asegurar que estos sean eficientes y dar un soporte a la cooperación entre procesos. Estos mecanismos podrían ser usados por el usuario y por los diferentes procesos del SO, mientras colaboran entre sí en la utilización de los recursos.

# SEMÁFOROS

Se propone una solución para el manejo de los procesos concurrentes, que simulan el funcionamiento de los semáforos de tránsito.

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

Su función es indicar si un proceso puede ejecutar o no dependiendo del semáforo, que protegen estos recursos compartidos.
Estos semáforos los utilizamos para permitir bloquear el acceso de un proceso a un recurso compartido y evitar el acceso simultaneo de 2 procesos que producirían resultados ambiguos o errores.

Para la señalización se utilizan variables especiales llamadas **«semáforos».**

Para **transmitir** una señal vía el semáforo que lo llamaremos **(s),** el proceso ejecutara la primitiva **semSignal(s) (señal de semáforo)**
Un proceso emitirá una señal semSignal para **activar** un semáforo, encender una bandera que avisa que está utilizando el recurso compartido.

Para **recibir** una señal vía el semáforo que lo llamaremos **(s)**, el proceso ejecutará la primitiva **semWait(S) (espera del semáforo)**
Utilizará una primitiva semWait para **preguntar** si debemos detenernos ante ese semáforo o podemos continuar con la ejecución.

Con las funciones de **semWait** y **semSignal**, el SO las maneja de forma atómica. El semWait y semSignal, decremento, espera y verificación de todo el manejo de ese semáforo, se hace en un solo ciclo de reloj.

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 (s >=0)**

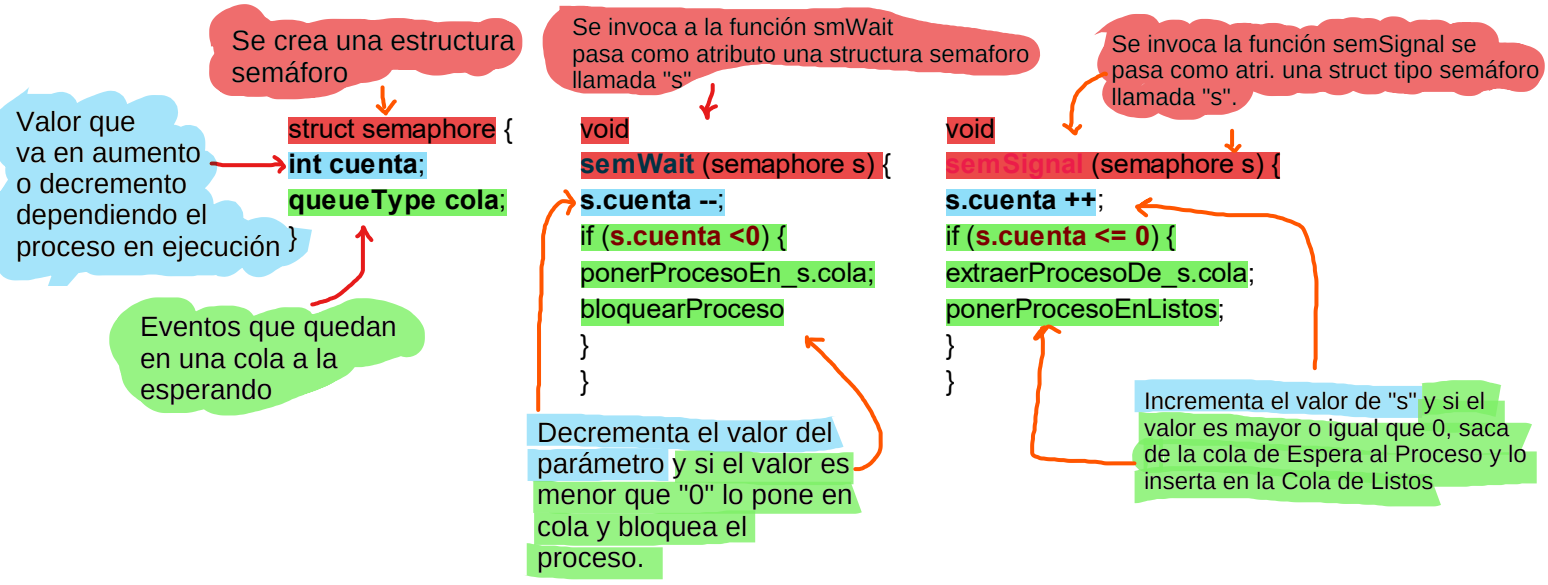
2. La operación **semWait** *decrementa el valor del semáforo.*

semWait = s – 1

3. La operación **semSignal** *incrementa el valor del semáforo.*

semSignal = s + 1

# Semáforos Contador




Cada una de estas acciones que realizamos tanto para analizar el semWait y el semSignal corresponden a una acción Atómica ( una unidad de tiempo relog).
- Los atributos que pasamos a la función del tipo estructura "semaphone" serán con el nombre "s" en cualquiera de los dos casos solo para indicar que estamos hablando del semaforo en si -

Se los conoce como semáforo contador, nos va permitir proteger un recurso de más de un acceso concurrente.

Este semáforo contador lo iniciaremos con un valor, dependiendo la cantidad de procesos que queremos administrar con este proceso a un recurso - También puede ser utilizado como Semáforo Binario asignando su valor en 1.

INFO ADICIONAL DE USO DE SEMAFORO NORMAL

## Paso a paso del Código

**Estructura semáforo** que almacena los valores.
Tenemos un **valor entero** el cual se va a ir **incrementando** o **decrementando**.
Una **cola** donde los procesos irán esperando (intentan hacer el semWait, no pueden ejecutarse y quedan bloqueados).

**Función semWait** recibe una estructura de tipo semáforo.
Decrementa el contador.
Si **cuenta < 0** pone el proceso en cola de ese semáforo. Indicando que ese proceso está esperando que se libere el recurso para poder utilizarlo.
Luego lo bloquea.

**Función semSignal** recibe una estructura de tipo semáforo.
Incrementa el contador.
Si **cuenta <=0** saca el proceso de la cola.
Lo libera y cambia a estado Listo.

Permite proteger un recurso, de mas de un acceso concurrente. Puedo tener una cantidad de "n" procesos concurrentes. Con el uso del semáforo se puede determinar hasta cuantos procesos, se puede aceptar en ese recurso.

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

Estructura (s) alamacena valores.
int deceremenda o incrementa
cola de bloquedos

```
void
semWait (semaphore s) {
s.cuenta --;
if (s.cuenta <0) {
ponerProcesoEn_s.cola;
bloquearProceso
}
}
```

**Funcion Wait tipo semaforo**

cuenta -1 (disminuye contador)
**Condicion**
**Ponen en cola** a espera recurso
**BLOQUEA** procesos

```
void
semSignal (semaphore s) {
s.cuenta ++;
if (s.cuenta <= 0) {
extraerProcesoDe_s.cola;
ponerProcesoEnListos;
}
}
```

**Funcion Signal tipo semaforo**

cuenta +1 (aumenta contador)
**Condicion**
**Saca el proceso cola** para darle el recurso
Pone en **LISTO,** el recurso queda libre

El semáforo protege un recurso y este recurso puede ser usando por “n” usuarios o “n” proceso concurrentemente.

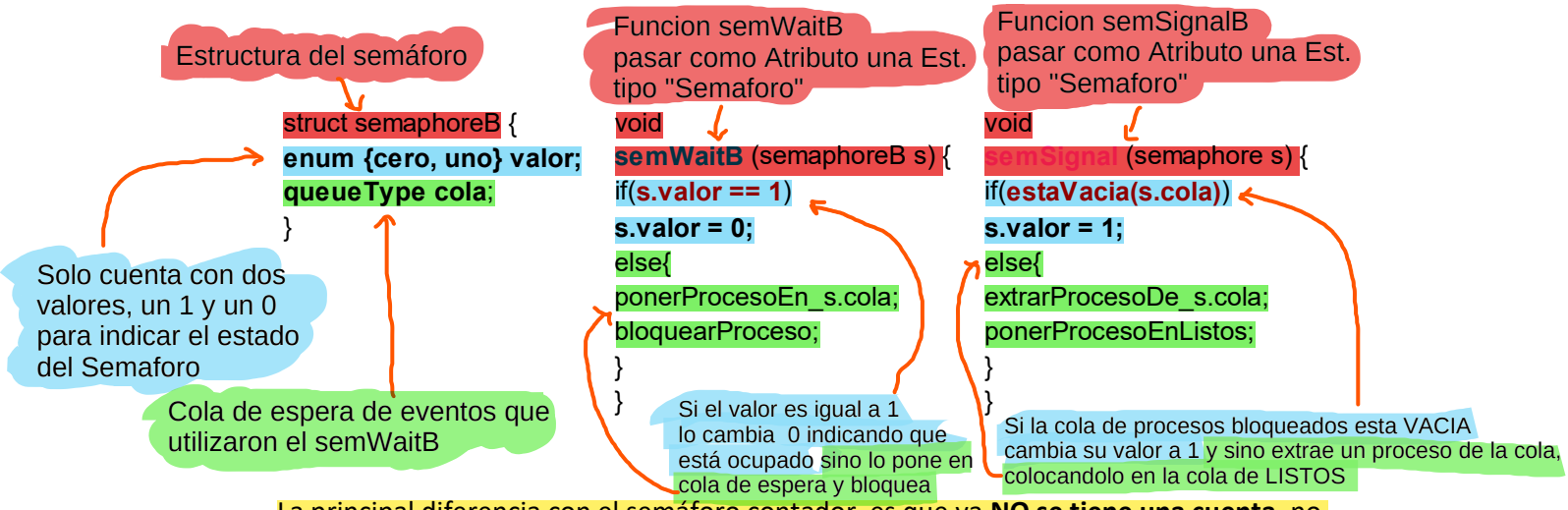
La primitiva **SemWait** va a intentar decrementar ese semáforo en 1.
Intenta decrementar el valor del semáforo y si el valor al decrementarlo es negativo o si ese semáforo ya se encuentra en 0, no lo puedo decrementar. Si el valor es menor que cero no se puede continuar con la ejecución, **queda bloqueado**. La cantidad de procesos que están utilizando ese recurso que este semáforo protege, **no admite más usuarios.**

Entonces por algún motivo el semáforo llego a cero, el próximo proceso que intente hacer un semWait de ese semáforo quedará bloqueado, (el semáforo no puede bajar de cero, no puede ser negativo). Entonces todo aquel que quiera hacer un semWait de ese semáforo va a quedar bloqueado, esperando.

Cuando algún otro proceso que esté utilizando el recurso que protege ese semáforo lo libera, hará un **semSignal**, esa primitiva va a incrementar ese semáforo. Al incrementar ese semáforo aquellos procesos que están en semWhait podrán decrementarlo en un orden, Schedule planificación de procesos, podrán pasar a ejecución.

# Semáforos Binario

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




La principal diferencia con el semáforo contador, es que ya **NO se tiene una cuenta**, no se incrementa/decrementa el semáforo.
Es utilizado para recursos de exclusión mutua. Para recursos de **uso exclusivo**. Quien logra activar el semáforo, excluye al resto hasta que él lo libere.

## Paso a paso del Código

Un semáforo binario solo puede tomar los valores 0 y 1 y se puede definir las siguientes operaciones.

**Estructura semáforo** Un semáforo binario **solo puede ser inicializado en 0 o 1**.
Ya no almacena, le asigna un valor 0 o 1 (ocupado o libre).
Tiene una cola donde los procesos irán esperando.

**Función semWait** **Comprueba el valor de semáforo.**
Si el valor que tengo en el **semáforo = 1**, lo cambia a 0. **Caso contrario**, el proceso que intenta hacer el wait **quedará bloqueado**.

**Función semSignal** **Verifica si la cola está vacía**.
**SI** está vacía COLA. **NO hay procesos esperando el recurso compartido**, pone el **semáforo en 1**. Lo que indica que **el recurso esta libre.**
**NO** está vacía **COLA. Hay procesos esperando el recurso compartido.**
Saca de la cola cada vez que un proceso haga un semSignalB, para que utilice el recurso compartido y luego cambia el estado, de bloqueado a Listo. Y así sucesivamente vuelve a verificar si hay procesos encolados recién pondrá el semáforo en 1 cuando la cola está vacía. Si hay procesos en cola, lo mantiene en 0 hasta que esa cola se libere**.**

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

Estructura (s) alamacena valores.
binario 0 - 1
cola de bloquedos

```
void
semWaitB (semaphoreB s) {
if(s.valor == 1)
s.valor = 0;
else{
ponerProcesoEn_s.cola;
bloquearProceso;
}
}
```

**Funcion Wait tipo semaforo**

Condicion: **comprueba el valor del semaforo**
Cambia a valor a 0
**NO** cumple condicion
**Ponen en cola** a espera recurso
No cumple condición **BLOQUEA**

```
void
semSignal (semaphore s) {
if(estaVacia(s.cola))
s.valor = 1;
else{
extrarProcesoDe_s.cola;
ponerProcesoEnListos;
}
}
```

**Funcion Signal tipo semaforo**

Condicion: **verifica si hay procesos en cola**
cambia a valor a 1 (no hay procesos esperando)
**NO** cumple condicion
**Saca el proceso cola** para darle el recurso
Pone en **LISTO,** el recurso queda libre

# Ejemplo Semáforo

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 < 0**, el proceso se bloquea.
- Si el valor de **s >= 0**, decrementa el valor del semáforo y el proceso entra a su S.C. Luego hará un **SemSignial**(s), incrementa el valor, indicando que el recurso está libre.

```
/* programa exclusión mutua */
semaphore s=1;                  inicializa semaforo en 1
void p() {
while(true){                    bucle infinito
semWait(s);                     s= s - 1; si s< 0 se bloq, sino sigue
/* sección crítica */           entra SC
semSignal(s);                   s= s+1
/* resto */
}
}
```

Inicializa el semáforo en 1, tengo un proceso en un bucle infinito que hace un semWait del semáforo (s), poniendo el semáforo en cero. Luego ejecuta su código relacionado con la SC, una vez que termina hará su semSignal (s) para volver al semáforo al valor 1.

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.
Ej. s=2 En ese momento, ese recurso admite dos procesos más ejecutando.

• **s.cuenta< 0**: la magnitud **(valor absoluto)** de s.cuenta es el número de procesos suspendidos en s.cola.
Ej. S=-10 cantidad de procesos esperando en la cola

Si queremos que el semáforo pueda tener tres procesos a la vez, hay que inicializar el semáforo con 3. Permitiremos entrar a tres procesos a una SC utilizando el mismo semáforo.

**Secuencia SEMAFORO CONTADOR**:

```
void p(1){                    void p(2){
while (true) {                while (true) {
semWait(s);   // s=2          semWait(s);     // s=1
/*SC*/                        /*SC*/
```

```
void p(3){                    void p(4){
while (true) {                while (true) {
semWait(s);   // s=0          semWait(s);     // No hace el wait porque s >=0
/*SC*/                                          proceso  bloqueado
```

Si llegara a entrar un (4) proceso, no va a poder entrar a la SC, porque el semáforo ya está en 0. Quedará bloqueado.
Únicamente va a poder entrar si alguno de los 3 procesos haga un **semSignial** de ese semáforo, va a cambiar a 1 el valor del semáforo y permitirá entrar el proceso (4).

**Secuencia SEMAFORO BINARIO**:

Suponemos se inicializa en s=1

```
void p(1){
while (true) {
semWaitB(s);  //cambia s=0
/*SC*/
semSignial(s); cola? SI
```

```
void p(2){
while (true) {
semWaitB(s);    //  s=0
                bloq -cola
/*SC*/
semSignial(s);  cola? SI
```

```
void p(3){
while (true) {
semWaitB(s);    //  s=0
                bloq - cola
                bloq - cola

/*SC*/
semSignial(s);  cola? NO
                cambia s=1 Recurso libre
```

Importante: Cuando un proceso después de pasar por la SC, hace un **semSignial**, verifica si hay procesos en cola. Si hay, no cambia el valor del semáforo, porque el proceso que libera el recurso, le da paso al siguiente, dejando al recurso bloqueado. No tiene sentido que lo desboque, gastaría recursos de procesamiento en desbloquearlo y luego bloquearlo. Directamente lo deja bloqueado y libera la cola.
El recurso se mantiene bloqueado y va liberando. Si entra algún otro proceso irá a la cola de espera.
Se evita el choque de liberar y que entre otro. (ej. justo que lo libero otro hace el senWaitB y gana la prioridad en la cola).

## Entonces deja el recurso bloqueado hasta que la cola quede vacía.

# PRODUCTOR -CONSUMIDOR

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

**Emisor- Receptor**
Un **proceso que genera información** – **proceso que consume información**.
Ej. Impresión, la maquina produce, la impresora consume.
Uno o mas procesos generando información, los coloca en un buffer**. Buffer es el lugar donde el emisor pone la información**. Hay un único consumidor que extrae los datos de ese buffer. Tenemos 3 situaciones:

- Productor no pueda producir más**, buffer lleno.**
- Consumidor se quede sin información, **buffer vació.**
- **Falta de sincronización para utilizar el buffer**. Buffer es compartido. Uno coloca y otro extrae. Si no se sincroniza esos accesos al buffer, puedo tener problemas de comunicación entre 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 dato.

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

Defimos el tamaño del buffer
Inicializamos en 0 la variable global cantItems

## Función productor

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

## Función consumidor

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

## Función productor

| | |
|---|---|
| void productor(){ | |
| while(1){ | Mientras |
| int item = producirItem(); | productor produce un item |
| if(cantItems == BUFFER_SIZE) | pregunta si tiene lugar en buffer, si es igual está lleno |
| sleep(); | entonces productor duerme |
| ponerItemBuffer(item); | sino está lleno pone el item en el buffer |
| cantItems++; | aumenta la cantidad de ítems. |
| if(cantItems == 1) | Pregunta si la cantidad de ítems =1 |
| wakeup(consumidor); | Despierta al consumidor, avisa tenes info en el buffer |

## Función consumidor

| | |
|---|---|
| void consumidor(){ | |
| while(1){ | Mientras |
| if(cantItems == 0) | pregunta si se queda sin elementos para retirar del buffer |
| **sleep();** | entonces consumidor duerme |
| int item = quitarItemBuffer(); | Sino saca un item del buffer |
| cantItems--; | decrementa la cantidad de ítems. |
| if(cantItems == BUFFER_SIZE-1) | Pregunta si por lo menos tiene 1 espacio libre en el buffer |
| **wakeup(productor);** | Despierta al productor, avisa hay lugar continua produciendo |

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

1. Cuando **cantItems valga 0**, el consumidor no tendrá ítems 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 válida, pero tiene un problema muy grande. Veamos la siguiente traza de ejecución:

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

| Tiempo | Productor | Consumidor |
|---|---|---|
| 0 | | if(cantItems == 0 ) **Ok** |
| 1 | int item = producirItem(); | |
| 2 | if(cantItems == BUFFER_SIZE) **X** | |
| 3 | ponerItemBuffer(item); | |
| 4 | cantItems++; | |
| 5 | if(cantItems == 1) **Ok** | |
| 6 | **wakeup(consumidor);** | |
| 7 | | **sleep();** |

Ambos quedan bloqueados, por tema de sincronización de procesos.

Productor hace wakeup al consumidor, el proceso del consumidor continua, le corresponde un sleep. En la siguiente vuelta del bucle, como la cantItmes quedo en 1 luego incrementará en cantItems=2, consumidor no despertará y seguirá productor continuará incrementando el buffer (3,4,5.100) hasta llenarlo.

Se podría solucionar el problema planteado anteriormente implementando semáforos.

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.

| | |
|---|---|
| #define **BUFFER_SIZE** 100 | Defimos el tamaño del buffer |
| **semaphore huecos** = BUFFER_SIZE; | Tipo Contador / inicializamos el contador con el tamaño del buffer |
| **semaphore elementos** = 0; | Tipo Contador / inicializamos el contador en 0 |
| **semaphore mutex** = 1; | Tipo Binario / inicializamos en 1 el semaforo |

**Productor**

```
void productor(){
while(1){
int item = producirItem();
semWait(huecos);
semWait(mutex);
ponerItemBuffer(item);
semSignal(mutex);
semSignal(elementos);
}
}
```

**Consumidor**

```
void consumidor(){
while(1){
semWait(elementos);
semWait(mutex);
int item = quitarItemBuffer();
semSignal(mutex);
semSignal(huecos);
consumirItem(item);
}
}
```

Supongamos una secuencia en donde no se interfieren, ejecuta Productor y luego Consumidor.

| Huecos | Elementos | Mutex | Items |
|---|---|---|---|
| 100 | 0 | 1 | 1 |
| 99 | | | |
| | | 0 | |
| | | | |
| | | 1 | |
| | 1 | | |
| | 0 | | |
| | | 0 | |
| | | | 0 |
| | | 1 | |
| 100 | | | |

| Tiempo | Productor | Consumidor |
|---|---|---|
| 0 | Produce items+1 | |
| 1 | Wait huecos | |
| 2 | | |
| 3 | Pone buffer | |
| 4 | Signal mutex | |
| 5 | Signal elementos | |
| 6 | | Wait elementos |
| 7 | | Wait mutex |
| 8 | | Quita buffer itmes -1 |
| 9 | | Signal mutex |
| 10 | | Signal huecos |

Suponemos que ejecuta una instrucción Productor y otra de Consumidor.

| Huecos | Elementos | Mutex | Items |
|---|---|---|---|
| 100 | 0 | 1 | 1 |
| 99 | | | |
| | | | |
| | | 0 | |
| | | | |
| | | 1 | |
| | 1 | | |
| | 0 | | |
| | | | 2 |
| | | 0 | |
| 98 | | | |
| | | | 1 |
| | | 1 | |
| | | 0 | |
| | | | |
| | | | |
| 99 | | | |
| 98 | | | |
| 97 | | | |
| ... | | | |
| 0 | | | |
| 0 | | | |
| 1 | | | |
| 0 | | | |

| Tiempo | Productor | Consumidor | Cola Prod. | Cola Cons. |
|---|---|---|---|---|
| 0 | Produce items+1 | | | |
| 1 | Wait huecos | | | |
| 2 | | Wait elementos | Bloqueado | Bloqueado |
| 3 | Wait mutex | Wait elementos | Bloqueado | Bloqueado |
| 4 | Pone buffer | Wait elementos | Bloqueado | Bloqueado |
| 5 | Signal mutex | Wait elementos | Bloqueado | Bloqueado |
| 6 | Signal elementos | Wait elementos | Bloqueado | Bloqueado |
| 7 | | Wait elementos | | |
| 8 | Produce items+1 | | | |
| 9 | | Wait mutex | | |
| 10 | Wait huecos | | | |
| 11 | | Quita buffer itmes -1 | | |
| 12 | Wait mutex | | | |
| 13 | | Signal mutex | | |
| 14 | Pone buffer | | | |
| ... | | | | |
| 130 | | | | |
| 131 | | | | |
| 132 | | | | |
| 133 | | | | |
| 134 | Wait huecos | | Bloqueado | |
| 135 | Wait huecos | | Bloqueado | |
| 136 | Wait huecos | Signal huecos | Bloqueado | |
| 137 | Wait huecos | | | |

**Semáforo del mutex** es utilizado para protección del buffer, es para evitar que ambos accedan en forma simultánea.
**Protección del recurso** para que no lo accedan los dos a la vez

- semWait(mutex);
- Pone productor o Saca consumidor
- semSignal(mutex);

**Semáforo de huecos y elementos** los utiliza para llevar cuenta espacios y elementos en el buffer.

# Semáforos en C

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

**#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()**
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()**
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.

• **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.