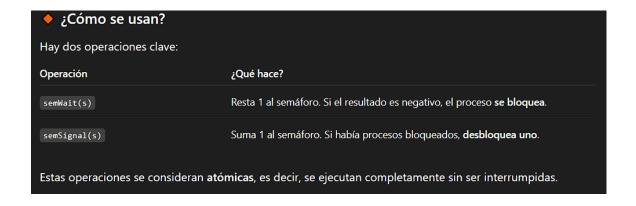
EXPLICACIÓN DEL MATERIAL

¿Qué son los semáforos?

Son **mecanismos de sincronización** entre procesos o hilos. Se usan para evitar que dos procesos accedan al mismo recurso al mismo tiempo (por ejemplo, una variable compartida), y también para coordinar el orden en que se ejecutan.

Origen e idea principal

- El concepto fue introducido por **Dijkstra** en 1965.
- Los procesos se comunican entre sí a través de señales, que indican cuándo un proceso puede continuar.
 - Estas señales se manejan con variables especiales llamadas "semáforos".



Implementación básica (semáforo con contador)

```
struct semaphore {
    int cuenta;
    queueType cola; // cola de procesos bloqueados
};
void semWait(semaphore s) {
    s.cuenta--;
    if (s.cuenta < ∅) {</pre>
        ponerProcesoEn_s.cola;
        bloquearProceso;
    }
}
void semSignal(semaphore s) {
    s.cuenta++;
    if (s.cuenta <= 0) {
        extraerProcesoDe_s.cola;
        ponerProcesoEnListos;
    }
```

Semáforo binario (o mutex)

Versión simplificada que solo toma valores 0 o 1. Ideal para exclusión mutua.

```
struct b sem {
 enum {cero, uno} valor;
 queueType cola;
}
```

```
semWaitB(s)
• Si el valor es 1, lo pone en 0 y entra.
   Si es 0, se bloquea.
semSignalB(s)
• Si no hay nadie esperando, pone el valor en 1.
   Si hay procesos esperando, desbloquea uno.
```

```
    Ejemplo práctico: exclusión mutua

	☐ Copiar

⊘ Editar

  semaphore s.cuenta = 1;
  void p() {
    while (true) {
      semWait(s);
      semSignal(s);
  }
Este código sirve para que solo un proceso a la vez entre en la sección crítica. Si querés permitir más de uno,
```

simplemente inicializás s.cuenta con un valor mayor a 1.

¿Qué es el problema Productor-Consumidor?

Es un problema clásico de concurrencia que describe una situación donde:

- Uno o más productores generan datos y los colocan en un buffer compartido.
- Un consumidor toma esos datos del buffer para procesarlos.
- El buffer tiene una capacidad limitada, por lo tanto:
 - El productor debe esperar si el buffer está lleno.
 - o El consumidor debe esperar si el buffer está vacío.
- Además, no puede haber acceso simultáneo al buffer: eso causaría corrupción de datos ⇒ necesitamos exclusión mutua.

Código sin semáforos (problema error)

```
#define BUFFER_SIZE 100
int cantItems = 0;
void productor() {
   while(1) {
       int item = producirItem();
       if (cantItems == BUFFER_SIZE)
           sleep(); // espera si el buffer está lleno
       ponerItemBuffer(item);
        cantItems++;
       if (cantItems == 1)
           wakeup(consumidor); // avisa que ya hay algo para consumir
   }
void consumidor() {
   while(1) {
       if (cantItems == 0)
           sleep(); // espera si no hay nada
        int item = quitarItemBuffer();
       cantItems--;
       if (cantItems == BUFFER_SIZE - 1)
           wakeup(productor); // avisa que hay espacio para producir
```

X ¿Qué problema tiene este código?

Podría ocurrir una situación como esta:

- cantItems = 0
- El productor produce un ítem y hace wakeup(consumidor), pero justo el consumidor aún no está dormido.
- Luego el consumidor entra y se duerme, pero nadie más lo va a despertar.

Resultado: ambos quedan dormidos para siempre → ¡bloqueo mutuo!

Solución con semáforos

Se usan 3 semáforos:

```
#define BUFFER_SIZE 100

semaphore huecos = BUFFER_SIZE;  // cuenta cuántos huecos quedan
semaphore elementos = 0;  // cuenta cuántos elementos hay
semaphore mutex = 1;  // exclusión mutua (semáforo binario)
```

X Código con semáforos

Productor:

Consumidor:

RESUMEN CLARO		
Concepto	Explicación corta	ð
Productor–Consumidor	Productores meten datos al buffer, consumidor los saca.	
Problema	Si se usan sleep() y wakeup() mal, puede haber bloqueos o pérdic de señales.	da
Solución	Usar semáforos para controlar:	
- mutex (binario)	Asegura exclusión mutua al usar el buffer.	
- huecos (contador)	Cuenta espacios disponibles en el buffer.	
- elementos (contador)	Cuenta cuántos elementos hay disponibles para consumir.	



🔒 ¿Qué son los semáforos en C?

Son mecanismos de sincronización que permiten controlar el acceso de múltiples hilos o procesos a recursos compartidos, como una sección crítica.

En C, se usan gracias a la librería:



Y se declaran con el tipo especial:

sem t nombre;

Este tipo de semáforo (sem_t) puede ser binario (0 o 1) o con contador (valores >1).

Inicialización del semáforo

Se hace con la función sem_init generalmente en el main.

sem_init(&nombre, pshared, valor_inicial);

Parámetro	¿Qué hace?
&nombre	Dirección del semáforo
pshared	0 si es compartido entre hilos del mismo proceso; otro valor si es entre procesos distintos
valor_inicial	0 (cerrado), 1 (abierto), >1 (semaforo contador)

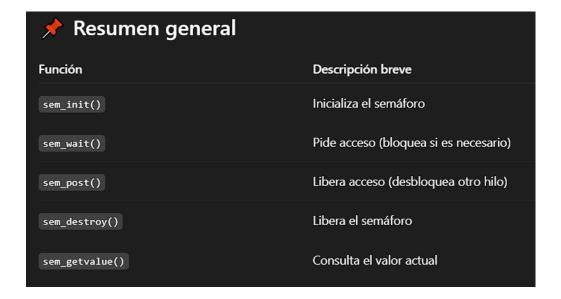
V Funciones importantes

- sem_wait(&sem);
 - Equivale a semWait.
 - Si el semáforo vale 0, bloquea al hilo.
 - Si vale >0, lo decrementa y deja pasar.

- sem_post(&sem);
 - Equivale a semSignal.
 - Incrementa el semáforo.
 - Si había hilos bloqueados, despierta uno.
- sem_post_multiple(&sem, count);
 - Incrementa el semáforo en count.
 - (No siempre está disponible en todas las implementaciones de POSIX)

Otras funciones útiles

- sem_destroy(&sem);
 - Libera los recursos del semáforo.
 - Solo debe llamarse cuando ningún hilo esté usando el semáforo.
- sem_getvalue(&sem, &valor);
 - Guarda el valor actual del semáforo en la variable valor.



Ej en codigo

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t mutex; // Semáforo global
void* seccion_critica(void* arg) {
    sem_wait(&mutex); // Entrar a sección crítica
    printf("Hilo %d dentro de la sección crítica\n", *(int*)arg);
    sem_post(&mutex); // Salir de sección crítica
   return NULL;
}
int main() {
   pthread_t hilos[2];
   int id[2] = \{1, 2\};
    sem_init(&mutex, 0, 1); // Semáforo binario (abierto)
    for(int i = 0; i < 2; i++)
        pthread_create(&hilos[i], NULL, seccion_critica, &id[i]);
    for(int i = 0; i < 2; i++)
        pthread_join(hilos[i], NULL);
    sem_destroy(&mutex); // Limpieza
    return 0;
```