El semaforo es un valor entero que nunca puede ser < 0. Se utilizan para la sincronizacion de procesos.

**sem\_open** crea un nuevo semaforo o abre uno ya existente

**sem\_close** cierra el semaforo cuando un programa lo dejo de usar

**sem\_unlink** elimina el semaforo del sistema cuando todos los programas/procesos lo dejaron de usar.

2 operaciones pueden ejecutarse en un semaforo:

**sem\_wait** -1 al valor del semaforo

**sem\_post** +1 al valor del semaforo

* Semaforos nombrados:

/nombre —> Asigna un nombre al semaforo

Dos procesos pueden operar el mismo semaforo si ambos pasan el mismo /nombre.

Son administrados por el kernel.

Están disponibles globalmente para todos los procesos del sistema operativo

(i.e. no hay semáforos "privados").

Su estado se preserva mientras el SO esté activo (i.e., se pierden entre reinicios).

Cada semáforo tiene un "nombre" que lo identifica con el kernel, en nuestro caso

los nombres son números enteros entre 0 y un límite máximo (idea similar a los

file descriptors).

* Semaforos no nombrados:

Se los guarda en una region de la memoria que se comparte entre varios hilos (thread-shared semaphore) o procesos (process-shared semaphore).

sem\_init —> Inicializacion del semaforo

sem\_destroy —> Destruccion del semaforo

### **Persistence**

POSIX named semaphores have kernel persistence: if not removed by [***sem\_unlink***](https://linux.die.net/man/3/sem_unlink)*(3)*, a semaphore will exist until the system is shut down.

**Syscalls a utilizar:**

● acquire(): Toma un lock haciendo busy waiting hasta que esté

disponible, similar al down de un semáforo. Útil para crear zonas de

exclusión mutua (mutex).

● release(): Libera un lock tomado, similar a un up de un semáforo.

● sleep(): Pone a "dormir" el proceso que realizó la syscall hasta un

wakeup().

● wakeup(): Permite al scheduler volver a ejecutar un proceso que hizo una

llamada a sleep().

● argint: Permite leer un argumento dado en la llamada de la syscall.

● Implementar sem\_down() es la parte más desafiante y requiere una buena

comprensión de sleep() y wakeup(). Pueden buscar ideas e inspiración

en la implementación de pipewrite y piperead en pipe.c.

### **1. acquire()**

* **Función:** Toma un *lock* para garantizar la **exclusión mutua**.
* **Uso:** Similar a la operación down de un semáforo, se utiliza para evitar que otros procesos accedan a una sección crítica del código mientras un proceso ya la está utilizando.
* **Implementación:** La función realiza un **busy waiting**, lo que significa que el proceso espera activamente hasta que pueda tomar el lock.
* **Contexto de uso en semáforos:** Antes de modificar el valor de un semáforo o una estructura compartida, necesitarás adquirir el lock para asegurarte de que ningún otro proceso lo esté modificando al mismo tiempo.

Ej acquire(&lock); // Tomar el lock antes de entrar en la sección crítica

### **2. release()**

* **Función:** Libera un *lock* previamente adquirido con acquire().
* **Uso:** Similar a la operación up de un semáforo, se utiliza para que otros procesos puedan acceder a la sección crítica una vez que el proceso actual haya terminado.
* **Contexto de uso en semáforos:** Después de modificar un semáforo o una estructura compartida, debes liberar el lock para permitir que otros procesos accedan a la sección crítica

release(&lock); // Liberar el lock al salir de la sección crítica

### **3. sleep()**

* **Función:** Suspende la ejecución del proceso actual hasta que ocurra un **evento de wakeup**.
* **Uso:** Pone el proceso en un estado de espera en lugar de hacer busy waiting. La función sleep() es crucial para implementar la operación sem\_down() en semáforos, donde un proceso debe bloquearse si el valor del semáforo es 0.
* **Contexto de uso en semáforos:** Cuando un proceso llama a sem\_down() y el valor del semáforo es 0, debe dormirse hasta que otro proceso lo "despierte" incrementando el semáforo con sem\_up().
* sleep(&semaforo, &lock); // El proceso duerme hasta que se haga un wakeup

### **4. wakeup()**

* **Función:** Despierta a uno o más procesos que están esperando un evento (es decir, que están dormidos en sleep()).
* **Uso:** Cuando un proceso realiza sem\_up() y el valor del semáforo es mayor que 0, esta función se usa para "despertar" a procesos que están esperando para continuar.
* **Contexto de uso en semáforos:** Después de incrementar el semáforo con sem\_up(), si algún proceso estaba bloqueado esperando que el semáforo se incrementara, wakeup() lo activa para que pueda continuar su ejecución.

wakeup(&semaforo); // Despierta procesos dormidos en este semáforo

### **5. argint()**

* **Función:** Recupera un argumento entero de una syscall que ha sido realizada desde el espacio de usuario.
* **Uso:** Dado que las syscalls pasan parámetros desde el espacio de usuario al espacio de kernel, argint() se usa para extraer estos parámetros de la pila de llamadas. En tus syscalls de semáforos, esto se utiliza para recuperar los argumentos como el ID del semáforo o el valor inicial.
* **Contexto de uso en semáforos:** En cada una de tus syscalls (sem\_open(), sem\_close(), etc.), usarás argint() para obtener los valores que los programas de usuario están pasando, como el identificador del semáforo o el valor inicial.

int sem;

argint(0, &sem); // Recupera el primer argumento de la syscall (el ID del semáforo)

### **Resumen del flujo típico para una syscall de semáforos:**

1. **sem\_down()**:
   * Adquiere el lock (acquire()).
   * Verifica si el valor del semáforo es 0.
     + Si es 0, el proceso se duerme (sleep()).
   * Si no es 0, decrementa el valor del semáforo y libera el lock (release()).
2. **sem\_up()**:
   * Adquiere el lock (acquire()).
   * Incrementa el valor del semáforo.
   * Despierta a los procesos dormidos si el valor del semáforo es mayor que 0 (wakeup()).
   * Libera el lock (release()).