Semaphores
- Provide more general forms of synchronization
- Generalization of mutex locks
  - Mutex allows exclusive access to exactly one thread
  - Semaphore allows controlled access to multiple threads
- Provide the capability of conditional variables
  - Provide wait and signal operations
- Provide a way to synchronize thread execution order

Semaphore: Definition
- Semaphore has a non-negative counter
  - The value of the counter determines the access control and synchronization
- The counter is **internal** to semaphore
  - Can *only* be initialized
  - *Cannot* be directly accessed or examined by the program
- Semaphore can be thought of as a C struct

```c
struct {
  int counter;
  thread_list list;
};
```
Semaphore Operations

- **Wait**
  - Also called P (proberen), down, lock
  - Either decrements counter or blocks if counter is 0

- **Signal**
  - Also called V (verhogen), up, unlock
  - Either increments counter or wakes up a blocked thread

```c
if (S->counter > 0)
    S->counter--;
else
    add to S->list and block;
```

### Semaphore Properties

- Wait and signal are atomic operations
- OS implements these as critical sections
- Can be used for
  - Mutual exclusion
  - Signaling
  - Controlling thread execution order
- Behavior depends on initial value of counter

Semaphore Usage

- Initialize the semaphore
  - Initializes the internal counter
  - Must be initialized with a non-negative value
- Call wait and signal operations
  - Can be called in any order depending on synchronization requirements
  - Cannot directly access or examine counter value
  - Only wait and signal operations allowed

Controlling access to a Critical Section

- Each thread must call wait and signal in correct order
- How many threads can run in critical section if:
  - Initial value of S->counter is 1
  - Initial value of S->counter is 10
  - Initial value of S->counter is 0

```c
wait(S)

Critical section

signal(S)
```
**Synchronizing Execution Order**

- In which order would the threads run if:
  - Initial value of S->counter is 0
  - Initial value of S->counter is 1
- Can we do this kind of synchronization with mutex locks?

**Producer-Consumer: Using Semaphores**

| Buffer_type buffer; int num_items=0; |
| Semaphore mut=1; Semaphore items=0; |

Producer:

```c
while(1) {
    produce(item);
    wait(mut);
    put(item, buffer);
    num_items++;
    signal(mut);
    signal(items);
}
```

Consumer:

```c
while(1) {
    wait(items);
    wait(mut);
    item = get(buffer);
    num_items--;
    signal(mut);
    consume(item);
}
```

- Do we need num_items?

**Bounded Buffer Problem**

- Suppose Buffer is of finite size: capacity of N items
- What happens if buffer is full?
Bounded Buffer: Using Semaphores

Producer:

```c
while(1) {
    produce(item);
    wait(slots);
    wait(mut);
    put(item, buffer);
    signal(mut);
    signal(items);
}
```

Consumer:

```c
while(1) {
    wait(items);
    wait(mut);
    item = get(buffer);
    signal(mut);
    signal(slots);
    consume(item);
}
```

POSIX Semaphore: sem_t

- **Initialization:**
  ```c
  sem_init(sem_t *sem, int pshared, unsigned value);
  ```
- **sem:** Semaphore variable
- **pshared:**
  - 0 => sharing within process
  - Non-zero => shared across processes
- **value:** initial value (must be non-negative)
- **Destruction:** `sem_destroy(sem_t *sem);`

POSIX Semaphore: Operations

- **Wait/signal:**
  ```c
  sem_wait(sem_t *sem);
  sem_post(sem_t *sem);
  sem_trywait(sem_t *sem);
  ```

POSIX Semaphore: Example

```c
sem_t sem;
/* Initialize process-local semaphore with initial value=1 */
sem_init(&sem, 0, 1);
sem_wait(&sem);
/* Critical section */
sem_post(&sem);
```