Introduction to Semaphores

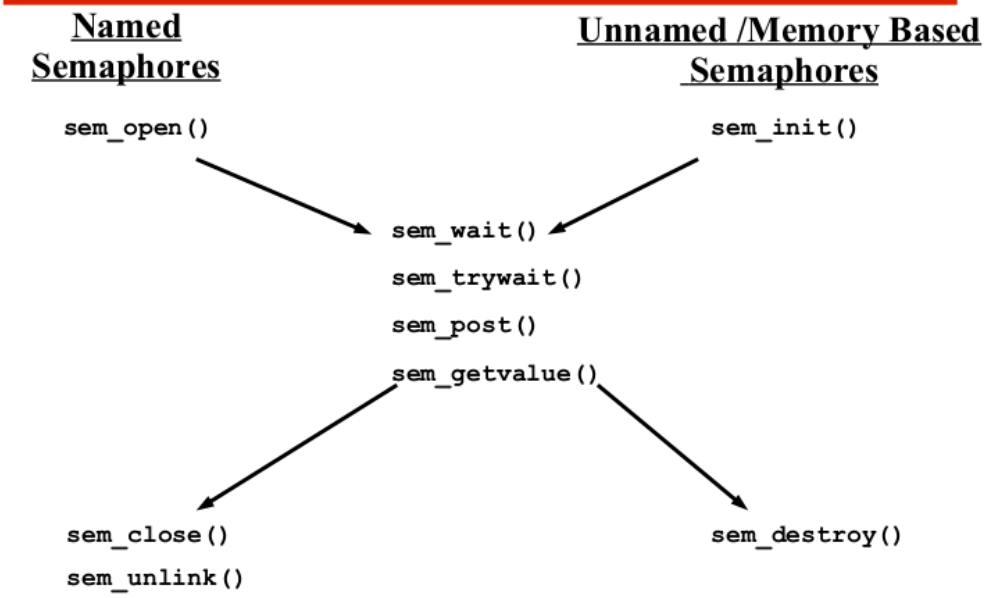
- Semaphores are a kind of generalized locks first defined by Dijkstra in late 1960s. A primitive used to provide synchronization between various processes or between various threads of a process. It can be considered as an integer variable, with three differences:
  - When you create a semaphore, you can initialize it to any integer value, but after that you can perform two operations on it, increment (verhogen, post, signal) and decrement (proberen, wait)
  - When a process/thread decrements the semaphore, if the semaphore currently has the value zero, then the thread blocks until the value of semaphore value rises above zero
  - When a process/thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked. Which one? (strong semaphore, weak semaphores)

Mutex, Condition Variable and Semaphore

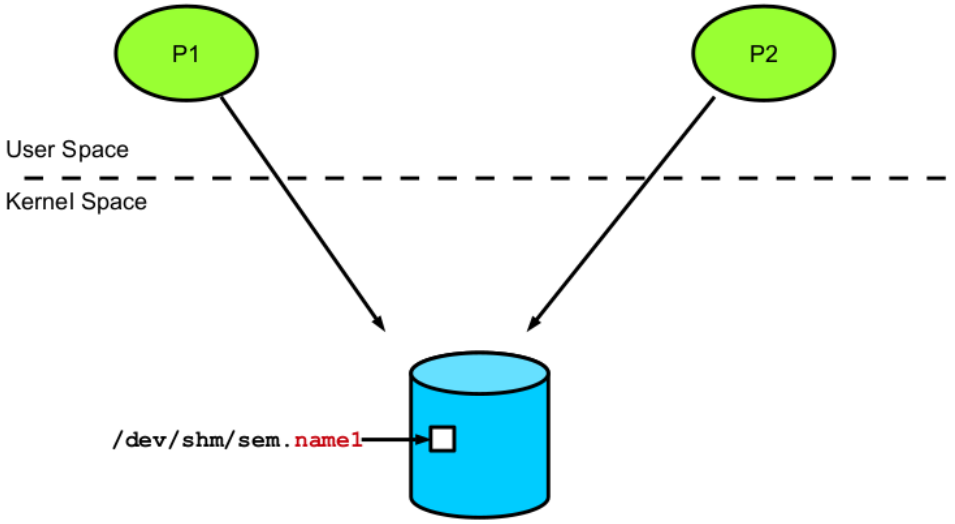
- A mutex can have only two values 0 or 1, and is used to achieve mutual exclusion, while semaphores can also be used as counting semaphores in order to access a shared pool of resources
- A mutex must always be unlocked by the thread that locked the mutex, whereas, a semaphore post need not be performed by the same thread that did the semaphore wait
- When a condition variable is signaled, if no thread is waiting for this condition variable, the signal is lost, while a semaphore post is always remembered
- Out of various synchronization techniques, the only function that can be called from a signal handler is semaphore post

Mutexes are optimized for locking, condition variables are optimized for waiting, and a semaphore can do both

Implementations of POSIX Semaphores



Creating a Named Semaphore



Creating a Named Semaphore

```
sem_t *sem_open(const char* name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

- The `sem_open()` library call creates a new semaphore or opens an existing semaphore identified by its first argument `name` of the form `/somename`; that is, a null-terminated string of up to NAME_MAX-4 (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes
- The second argument `oflag` is mostly O_CREAT, in which case the semaphore is created if it does not already exist. If both O_CREAT and O_EXCL are specified, then an error is returned if a semaphore with the given name already exists
- If O_CREAT is specified in `oflag`, then two additional arguments must be supplied. The `mode` argument specifies the permissions to be placed on the new semaphore. The `value` argument specifies the initial value for the new semaphore. Binary semaphores usually have an initial value of 1, whereas counting semaphores often have an initial value greater than 1
- The return value is a pointer to `sem_t` datatype, which is then used as the argument to `sem_wait()`, `sem_post()` and `sem_close()` calls

Incrementing and Decrementing Semaphores

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- The `sem_wait()` library call decrements the semaphore pointed to by `sem`. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until the value of semaphore value rises above zero
- The `sem_post()` library call increments the semaphore pointed to by `sem`. If the semaphore's value becomes greater than zero, then another process or thread blocked in a `sem_wait()` call will be woken up and proceed to lock the semaphore
- On success both the functions returns 0. On error, the value of the semaphore is left unchanged, a -1 is returned and `errno` is set to indicate the error

Incrementing and Decrementing Semaphores

```
int sem_trywait(sem_t *sem);
int sem_getvalue(sem_t *sem, int* sval);
```

- The `sem_trywait()` library call is the same as `sem_wait()`, except that if the decrement cannot be immediately performed, then the call returns an error instead of blocking
- The `sem_getvalue()` library call places the current value of the semaphore pointed to by `sem` into the integer pointed to by `sval`. POSIX permits two possibilities for the value of `sval` less than zero:
  - Either 0 is returned (Linux adopts this behavior)
  - Or a negative number is returned showing the count of blocked threads

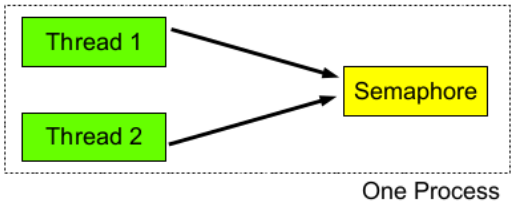
Closing and Removing a Named Semaphore

```
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

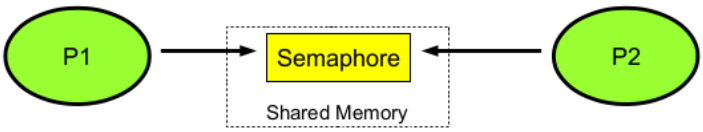
- A semaphore is automatically closed on process termination. A named semaphore can be closed by using the `sem_close()` library call and passing it the `sem_t` variable received via a previous `sem_open()` call
- Closing a named semaphore does not remove it from the system, as they are at least kernel-persistent. They retain their value even if no process currently has the semaphore open
- So to remove a named semaphore from the system we can use the `sem_unlink()` call. A semaphore has a reference count of how many times they are currently open. Removing of semaphore from filesystem occur when the reference count becomes zero and also after the last process that has opened the semaphore calls `sem_close()`
- On the shell on Linux, you can use the `rm(1)` command to delete the related file in the `/dev/shm/` directory

Creating a Un-Named Semaphore

Memory Based Semaphore Shared between two Threads



Memory Based Semaphore Shared between two Processes



Creating a Un-named Semaphore

```
int sem_init(sem_t *sem, int pshared, int value);
```

- The `sem_init()` library call initializes the unnamed semaphore at the address pointed to by its first argument `sem` with value mentioned as third argument
- If `pshared` is zero, then semaphore is shared between the threads of a process, and `sem` has to be global, so that it is accessible among all the threads of a process
- If `pshared` is non-zero, then semaphore is shared between processes, and `sem` has to be located in a region of shared memory
- After a successful call, the address of semaphore `sem` can be used as the argument to `sem_wait()` and `sem_post()` calls by the processes or threads
- Initializing a semaphore that has already been initialized results in undefined behavior

Destroying an Un-Named Semaphore

```
int sem_destroy(sem_t *sem);
```

- The `sem_destroy()` call destroys the unnamed semaphore at the address pointed to by `sem`. Only a semaphore that has been initialized by `sem_init()` should be destroyed using `sem_destroy()`
- Destroying a semaphore that other processes or threads are currently blocked on produces undefined behavior.
- Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using `sem_init()`
- On success the call returns 0. On error a -1 is returned, and `errno` is set to indicate the error