Faculty of Engineering and Applied Science

SOFE - 3950U Operating Systems

Winter 2023

Tutorial 6 - POSIX Threads Part II

Vishan Patel - 100784201
Akshat Kapoor - 100781511
Steven Mai - 100781485
Evidence Okeke - 100755328

Wednesday, March 15th, 2023

CRN 74171, Group 2

Github: https://github.com/23Vishan/OS-Tutorial-6

1. What is fork(), how does it differ from multi-threading (Pthreads)?

Fork() is a system call that creates a new process by duplicating the calling process. The new process is called the child process. The process that makes the fork() call is called the parent process.

Pthread is a process that can contain multiple threads all of which are executing the same program.

In fork(), when a child process is created, it shares the same address space, code and stack as the parent process. In Pthreads, the new thread in the process can share data, files and communicate with other threads in the process. However, the new thread will get its own stack, thread ID and registers.

2. What is inter-process communication (IPC)? Describe methods of performing IPC.

Inter-process communication is a mechanism that allows cooperating processes to exchange data and information.

a. Shared-Memory Method: Inter-process communication using shared memory method requires that the processes establish a region of shared memory. This shared-memory region typically resides in the process that creates the shared memory segment. To communicate with the process, other processes must attach it to their address space. This removes the restriction by the operating system and allows these processes to share data and communication in the shared area. An example of this is the producer-consumer process.

b. Message-Passing Method: Message-passing provides a mechanism for processes to communicate without sharing the same address space. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

3. Provide an explanation of semaphores, how do they work, how do they differ from mutual exclusion?

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). Semaphores are used to enforce mutual exclusion, avoid race conditions and implement synchronization between processes.

When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. If the value of the semaphore is zero, any process that performs a wait() operation can block itself. The block operation will place the process in a waiting queue and is restarted when some other process executes a signal() operation.

Mutual Exclusion is an object used to control access to shared resources. In mutual exclusion, at least one resource must be non shareable.

Mutual Exclusion (mutex) is a locking mechanism	Semaphore is a signaling mechanism
Mutual Exclusion is an object	Semaphore is an integer
Mutual Exclusion is only modified by the process that requests or releases a resource	Semaphore can be modified by wait() or signal() operations

4. Provide an explanation of wait (P) and signal (V).

Wait(P) - When a process executes the wait function, if semaphore is less than or equal to zero, the process must wait. The semaphore decrements.

```
wait(S) {
 while (S <= 0) ;
   // busy wait
      S--;
       }
```

Signal(V) - When a process executes the signal function, the semaphore value is incremented.

```
signal(S) {
        S++;
       }
```

5. Research the main functions used for semaphores in <semaphore.h> and explain each function.

- **a.** sem_wait(sem_t *) - decrements the semaphore pointed to by *sem.* If the value is greater than zero, then the decrement proceeds and it returns a function. If it is zero, then the call blocks until it becomes possible to decrement or there is an interruption.
- **b.** sem_close(sem_t *) - closes the semaphore referred to by *sem*. It frees every resource that was allocated to the calling process for the semaphore.
- **c.** sem_post (sem_t *) - increments the function pointed to by *sem.* If the semaphore value continues to increment and is greater than zero, then a process blocked in a sem_wait() call will be woken up and continues to decrement the semaphore.
- **d.** sem_destroy(sem_t *) - destroys the unnamed semaphore at the address pointed to by *sem*. Only a semaphore that has been initialized by sem_init() should be destroyed by sem_destroy()
- **e.** sem_init(sem_t *, int, unsigned) - initializes an unnamed semaphore at the address pointed to by *sem.*