CSc 332 (L) - Operating Systems

Lab – Spring 2018

Instructor: Michael Iannelli, email: miannelli@gradcenter.cuny.edu

Process Synchronization using POSIX thread libraries & Semaphores

March 23, 2018

Race Condition

A *race condition* occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

One common scenario where a race condition might occur is when more than one process share a file descriptor. Recall, when forking a child process, file descriptors are copied to the child process, which can result in concurrent operations on the file. Concurrent operations on the same file can cause data to be read or written in a nondeterministic order, creating race conditions and unpredictable behavior. One of the most common process synchronization technique used to prevent race condition is *semaphore*.

The rest of the handout consists of two parts describing process synchronization methods to avoid race condition. The first part discuss the pthread library and a semaphore-based technique is presented in the second part.

PART 1

POSIX thread (pthread) libraries

In the previous two assignments, we have seen how to use `fork()` system call to create new processes. When you create a new process using `fork()` system call, a copy of the parent process's address space, code and created for the child. Thread is another way a main or parent process create a concurrent process flow.

Please refer to this excellent online resource to learn about the pthread libraries: `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`. Also practice the example codes for thread creation, synchronization, termination, and so on given in the above tutorial.

PART 2

Semaphores

Semaphores are a process synchronization technique that prevents race conditions over a shared resource. Simply put, a semaphore is counter that keeps track of how many accesses to a single resource are allowed. There are two types of semaphores.

- Binary Semaphore(aka mutex semaphore)
  - most simple form of semaphores
  - guarantees mutually exclusive access to resource
  - semaphore is initialized to 1 and when something accesses it, it will decrement to 0

  - any other user of this resource will now be blocked until the initial resource releases the semaphore by incrementing the value back to 1

- Counting Semaphore
  - represents a resource with many units available
  - allows process to enter as long as more units are available
  - counter is initialized to N (the number of units available)

A semaphore is accessed only through two standard atomic operations: **wait** and **signal**.

Note: "atomic" operation means that the operation must be executed indivisibly and without interruption. That is, no two or more processes are allowed to modify the semaphore simultaneously. In addition, no interruption is allowed during modification.

There are three semaphore system calls. To use them, we need to include three header files, *<sys/types.h>*, *<sys/ipc.h>* , and *<sys/shm.h>*.

Like a process, a semaphore has its own life cycle:

- creation
- initialization
- operation
- removal.

`semget()` system call is used for semaphore creation; `semctl()` system call for initialization and removal; `semop()` system call for operation. It's up to the owner process (which creates the semaphore) to create, initialize, and remove the semaphore. For the other processes, apart from obtaining the semaphore identifier through `semget()`, they do semaphore operations on semaphores by calling `semop()`

Note: The initialization and operations (wait and signal) system calls are defined in the header file, `sem.h`. You need to only pass the required arguments to the functions the defined in this header.

Producer Consumer Problem [Description Reference: Wikipedia]

The consumer producer problem (also known as the bounded-buffer problem) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Producer Consumer Problem – Implementation

Create two unrelated processes producer & consumer. Create a semaphore using `semget()` system call. Create a shared memory using `shmget()` system call and attach a dynamic character array to it using

`shmat()` system call. The access to this shared memory is treated as the *critical section*. In process Producer define three semaphores **mutex** with intial value 1, **full** with initial value 0, **empty** with initial value *N*. Set this value by using system call `semctl()` with SETVAL option. `sem_op` value to decrement (-1) and increment (+1) respectively by the system call `semop()`. `P()` is used for *entry* section and `V()` is used in *exit* section.

For solution to the producer-consumer problem, refer the attachment `ProducerConsumer.c`

Note: The producer consumer problem requires both types of semaphores – binary and counting.

TO DO:

- There is no graded assignment as part of this handout.
- Use the solution to Producer-Consumer Problem to understand the need for process synchronization using semaphore variables.
- Construct a solution to the Producer-Consumer Problem using pthread libraries and apply the mutex-based solution to the problem.

***