CECS 326-01

Operating Systems

Dylan Dang (ID 026158052)

Assignment 5

Due Date: 12/1/2022

Submission Date: 12/2/2022

Program Description

Like the previous assignment, I am tasked make use of the POSIX implementation of the Linux shared memory mechanism. This program will incorporate the fork(), exec(), and wait() system calls to control the program that a child process is to execute. Since the program will make use of the POSIX implementation it will utilize the shm_open(), ftruncate(), mmap(), munmap(), close() functions. All these system calls/functions will be integrated into two C files: master.c and slave.c. Program will also utilize struct from myShm.h. The last assignment tackled the potential problem for race condition due to the concurrent access and modification of the index variable in the shared data structure. However, for this assignment I am to utilize another semaphore to guard against race condition in the display device for output. Hence, mutual exclusion is necessary again. I will be utilizing semaphore to enforce the necessary mutual exclusion. The named semaphore mechanism includes sem_wait(), sem_post(), sem_open(), sem_close(), and sem_unlink(). The unnamed semaphore mechanism includes sem_wait(), sem_post(), sem_init(), and sem_destroy().

Used function/system call & their definitions:

- fork() allows a new process to be created and consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- exec() replaces the process's memory space with a new program. The exec() system call loads a binary file into memory and starts its execution.
- wait() moves the parent process itself off the ready queue until the termination of the child.
- shm_open() returns a file descriptor that is associated with the shared "memory object" specified by name.
- ftruncate() truncates the file indicated by the open file descriptor to the indicated length.

- mmap() establishes a mapping between an address space of a process and a file associated with the file descriptor.
- munmap() remove any mappings for those entire pages containing any part of the address space of the process starting at 'address' (shm base address) and continuing for 'length' bytes.
- close() used to close a file, in this case it is used to close the shared memory segment as if it were a file.
- sem_wait() decrements by one the value of the semaphore. It will decrement when value is greater than zero. If value is zero, then the current thread will block until the value becomes greater than zero.
- sem_post() posts to a semaphore, incrementing its value by one. If the resulting value is greater than zero and if there is a thread waiting on the semaphore, the waiting thread decrements the value by one and continues running.
- sem_open() opens a named semaphore, returning a semaphore pointer that may be used on subsequent calls to sem_post(), sem_wait(), and sem_close().
- sem_close() closes a named semaphore that was previously opened by a thread of the current process using sem_open(). This frees system resources associated with the semaphore on behalf of the process.
- sem_unlink() unlinks a name semaphore. The name of the semaphore is removed from the set of names used by the named semaphores.
- sem_init() initializes the unnamed semaphore at the address pointed to by sem.
- sem_destroy() destroys the unnamed semaphore at the address pointed to by sem.

master.c description

Just like the previous assignment, Assignment 4, the *master.c* program is designed to make use of the fork(), exec(), and wait() system calls to create child processes with each child process executing a *slave* with its child number and the shared memory segment name passed to it from exec(). The given header file, myShm.h is also utilized within this program. The *master* program starts off with identifying itself and then storing the data from the commandline parameters such as the number of children to be created, shared memory segment name, and semaphore name. *Master* will use shm_open() to return a file descriptor which will be used to reference the content from the shared memory. Ftruncate() is also used here to configure the size of the shared memory. After that, mmap() is used to map the memory region between an address space of a process and a file associated with the file descriptor.

Like in the case of assignment 4, where mutual exclusion was used to stop multiple concurrent access to the index variable; we also use mutual exclusion to address the potential issue of race condition due to the concurrent output to the display device. Contrary to the previous assignment, I will make use of a named semaphore this time to combat the race condition regarding the access to the display device and I will use an unnamed semaphore and its mechanisms to combat race condition regarding access to the index variable. To begin, a named semaphore will be created to utilize sem_wait() and sem_post(). These functions will reserve space for the processes' outputs to the display. This step is crucial since I want to make sure that the outputs are accurate to their respective processes. Next, I will be using sem_init() to initialize an unnamed semaphore. The unnamed semaphore is initialized at the address pointed to by sem, in this case is mutex_sem from our struct in myShm.h. This named semaphore will control the access to the index variable of the processes. Now is the time for *master* to use fork() to create new child processes and exec() to pass arguments to the child. Each child process is to execute a *slave* with its child number and the shared memory segment name passed to it from exec(). The *master* should output the number of slaves it has created and must wait for all of them to finish. Sem_close() will be used since the named semaphore will not be used any longer in *master* and it also brings the benefit of freeing system resources. Sem_destroy() will also be used to destroy the unnamed semaphore. Upon termination of all child processes, *master* outputs the content of the shared memory segment filled in by the slaves. Finally, *master* uses the munmap() function to remove any mappings, close the shared memory segment using close(), removes the name of the shared memory segment using sem_unlink(), and then exits.

slave.c description:

The *slave.c* program procedures are quite like *master.c* program. *Slave* program also utilizes the given header file, myShm.h. The program starts off with the *slave* identifying itself. It then proceeds to show its child number and the shared memory segment name it received from the exec() system call from *master*. Like the *master* program, the *slave* program also uses the shm_open function to return a file descriptor, the ftruncate() to configure the size of the shared memory, and the mmap() to create a map between an address space of a process and the file associated with the file descriptor.

Since both named semaphores and unnamed semaphores are used in this assignment, *slave* will also use sem_open() to open a named semaphore using the name of the semaphore that was passed as an argument from master and sem_init() will be used to create an unnamed semaphore from the struct. In the critical section, slave will then use sem_wait() to hold a semaphore or keep it waiting. This is beneficial since I am trying to prevent any race condition problems regarding the device display and the access to the index variable. The function checks the semaphore value and determines whether the semaphore should be blocked. This mechanism allows for *slave* to then writes its child number into the next available slot in the shared memory without the worry of another process accessing it at the same time. After leaving the critical section, sem_post() is used when the process is finished operating on the index and outputting to the display since this function increments the semaphore value by 1 meaning the semaphore is unblocked and free. When all processes are done operating on the index, *slave* will use sem_close() to close the named semaphore and sem_destroy() to destroy the unnamed semaphore. Finally, *slave* uses the munmap() function to remove any mappings and uses close() to close the shared memory segment and then terminates.