CECS 326-01

Operating Systems

Dylan Dang (ID 026158052)

Assignment 4

Due Date: 11/3/2022

Submission Date: 11/3/2022

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 given myShm.h. However, it won't stop there. This assignment tackles the potential problem for race condition due to the concurrent access and modification of the index variable in the shared data structure. Hence, mutual exclusion is necessary. 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().

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.

master.c description

Just like the previous assignment, Assignment 3, 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 3, this is the step where master uses fork() to create new child processes and exec() to pass arguments to the child; however, before that, there is a need to address the potential issue of race condition due to the concurrent access and modification of the index variable in the shared data structure. To combat this potential problem, I must make use of mutual exclusion to limit the number of processes operating on the index. Utilization of semaphores and a shared memory buffer is necessary to solve this dilemma. Master uses sem_open() to open a named semaphore which allows for the use of the other named semaphore mechanisms. This will be followed by a sem_unlink() call to indicate that the named semaphore will be destroyed once all processes have ceased. 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 semaphore will not be used any longer in *master* and it also brings the benefit of freeing system resources. 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 shm_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 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. 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. 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. Sem_post() is used when the process is finished operating on the index 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 semaphore. Finally, *slave* uses the munmap() function to remove any mappings and uses close() to close the shared memory segment and then terminates.

master.c, slave.c, compilation screenshot:

```
dylandang@Dylan-Blade:~/Assignment 4$ gcc -o master master.c -lrt -lpthread
dylandang@Dylan-Blade:~/Assignment 4$ gcc -o slave slave.c -lrt -lpthread
dylandang@Dylan-Blade:~/Assignment 4$ ls
master  master.c  myShm.h  slave  slave.c
```

running master executable screenshot:

```
dylandang@Dylan-Blade:~/Assignment 4$ ./master 3 my_shm_name my_sem_name
Master begins execution
Master created a shared memory segment named my_shm_name
Master created a semaphore named my_sem_name
Master initializes index in the shared struct to 0
Master created 3 child processes to execute slave
Master waits for all child processes to terminate
Slave 1 begins execution
I am child number 1, received shared memory name, my_shm_name, and semaphore name, my_sem_name
Slave 1 acquires access to shared memory segment, and structures it according to struct CLASS
Slave 1 copies index to a local variable i
Slave 1 writes its child number in response[0]
Slave 1 increments index
Slave 1 closes access to shared memory and semaphore and terminates
Slave 1 exits
Slave 3 begins execution
I am child number 3, received shared memory name, my_shm_name, and semaphore name, my_sem_name
Slave 3 acquires access to shared memory segment, and structures it according to struct CLASS
Slave 3 copies index to a local variable i
Slave 3 writes its child number in response[1]
Slave 3 increments index
Slave 3 closes access to shared memory and semaphore and terminates
Slave 3 exits
Slave 2 begins execution
I am child number 2, received shared memory name, my_shm_name, and semaphore name, my_sem_name
Slave 2 acquires access to shared memory segment, and structures it according to struct CLASS
Slave 2 copies index to a local variable i
Slave 2 writes its child number in response[2]
Slave 2 increments index
Slave 2 closes access to shared memory and semaphore and terminates
Slave 2 exits
Master received termination signals from all 3 child processes
Content of shared memory segment filled by child processes:
--- content of shared memory ---
1
3
2
```

master.c code:

C master.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <semaphore.h>
#include "myShm.h"

int main(int argc, char** argv) {

    /*Shared memory file descriptor*/
    int shm_fileDes;

    /*Takes user input from commandline argument such as
    number of children, shared memory name, semaphore name and stores it*/
    int numChildren = atoi(argv[1]);
    const char *segmentName = argv[2];
    const char *semaphoreName = argv[3];

    /*Declare required size of memory request*/
    const int SIZE = sizeof(struct CLASS);

    /*Create base address as pointer to struct CLASS*/
    struct CLASS  *shm_baseAdd;

    /*When executed, master outputs a message to identify itself*/
    printf("Master begins execution\n");

    /*Output shared memory segment name*/
    printf("Master created a shared memory segment named %s\n", segmentName);

    /*Output semaphore name*/
    printf("Master created a semaphore named %s\n", semaphoreName);
```

```c

    /*Output starting index of response*/
    printf("Master initializes index in the shared struct to 0\n");

    /*The shm_open() function returns a file descriptor that is associated with the shared "memory object" specified by name.*/
    shm_fileDes = shm_open(segmentName,O_CREAT | O_RDWR,0666);

    /*Checking for failure*/
    if (shm_fileDes == -1) {
        printf("ERROR: Shared memory failed; shm_open() failed\n");
        exit(1);
    }

    /*The ftruncate() function truncates the file indicated by the open file descriptor to the indicated length. */
    ftruncate(shm_fileDes, SIZE);

    /*The mmap() function establishes a mapping between an address space of a process and a file associated with the file descriptor
    at the offset for length of bytes.*/
    shm_baseAdd = (struct CLASS *)  mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fileDes,0);

    /*Checking for failure*/
    if (shm_baseAdd == MAP_FAILED) {
        printf("ERROR: Map failed; mmap() failed\n");
        exit(1);
    }

    /*Update/Track index*/
    shm_baseAdd->index = 0;

    /*The sem_open() function opens a named semaphore, returning a semaphore pointer that may be used on
    subsequent calls to sem_post(), sem_wait(), and sem_close().

    The parameters of sem_open() are:
    1. name - a pointer to the null-terminated name of the semaphore to be opened
    2. oflag - option flags
    3. mode - permission flags
    4. value - initial value of the named semaphore */
    sem_t *mutex_sem = sem_open(semaphoreName, O_CREAT, 0666, 1);

```

```c
    /*Checking for failure*/
    if (mutex_sem == SEM_FAILED) {
        printf("ERROR: sem_open failed(): %s\n", strerror(errno));
        exit(1);
    }

    /*The sem_unlink() function unlinks a named semaphore.
    The name of the semaphore is removed from the set of names used by named semaphores.

    The parameter of sem_unlink() is:
    1. name - a pointer to the null-terminated name of the semaphore to be unlinked */
    if (sem_unlink(semaphoreName) == -1) {
            printf("ERROR: sem_unlink() failed: %s\n", strerror(errno));
            exit(1);
    }

    /*Determine number of child process to create*/
    for (int i = 0; i < numChildren; i++) {

        /*fork() system call is used to create child process*/
        pid_t childPID = fork();

        /*buffer for second argument in execlp()*/
        char childNum[5];

        /*Convert integer to char in order to be passed as an argument*/
        sprintf(childNum,"%d",i + 1);

        /*execute child process via execlp() function call*/
        if (childPID == 0){
            execlp("./slave",segmentName,childNum,semaphoreName, NULL);
        }
    }

    /*Output number of child process created by master to be executed by slave*/
    printf("Master created %d child processes to execute slave\n", numChildren);

    /*Waiting for all child processes to exit*/
    printf("Master waits for all child processes to terminate\n");

    /*Output acknowledgement message*/
    while(wait(NULL) != -1);
    printf("Master received termination signals from all %d child processes\n",numChildren);

    /*Output the content of the shared memory segment*/
    printf("Content of shared memory segment filled by child processes:\n");
    printf("--- content of shared memory ---\n");

    for (int i = 0; i < numChildren; i++) {
        /*For every child process, output the content of the shared memory filled in by the slave*/
        printf("%d\n", shm_baseAdd -> response[i]);
    }

    /*The sem_close() function closes a named semaphore that was previously opened by a thread of the current process using sem_open().
    The sem_close() function frees system resources associated with the semaphore on behalf of the process.sem_close()

    The parameter for sem_close() is:
    1. sem - a pointer to an opened named semaphore. This semaphore is closed for this process.*/
    if (sem_close(mutex_sem) == -1) {
        printf("ERROR from Master: sem_close failed: %s\n", strerror(errno));
        exit(1);
    }

    /*The munmap() function shall remove any mappings for those entire pages containing
    any part of the address space of the process starting at (shared memory base) address and continuing for (SIZE) length bytes*/
    if (munmap(shm_baseAdd, SIZE) == -1) {
        printf("ERROR from Master: Unmap failed; munmap() failed: %s\n", strerror(errno));
        exit(1);
    }

    /*The close() function is used to close a file, in this case it is used to close the shared memory segment as if it was a file*/
    if (close(shm_fileDes) == -1) {
        printf("ERROR from Master: Close failed; close() failed: %s\n", strerror(errno));
        exit(1);
     }

    /*Exit program*/
    exit(0);
}
```

slave.c code:

```
slave.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <errno.h>
#include <semaphore.h>
#include "myShm.h"

int main(int argc, char **argv) {

    /*Shared memory file descriptor*/
    int shm_fileDes;

    /*Takes user input from commandline argument such as
    number of children, shared memory name, semaphore name and stores it*/
    const char *segmentName = argv[0];
    int childNum = atoi(argv[1]);
    const char *semaphoreName = argv[2];

    /*Declare required size of memory request*/
    const int SIZE = sizeof(struct CLASS);

    /*Local variable that holds next free index*/
    int *i;

    /*Create base address as pointer to struct CLASS*/
    struct CLASS *shm_baseAdd;

    /*Output identifying message*/
    printf("Slave %d begins execution\n",childNum);

    /*Output child number and segment name from execlp() system call */
    printf("I am child number %d, received shared memory name, %s, and semaphore name, %s\n",childNum,segmentName,semaphoreName);

    /*The shm_open() function returns a file descriptor that is associated with the shared "memory object" specified by name.*/
    shm_fileDes = shm_open(segmentName, O_RDWR, 0666);

    /*Checking for failure*/
    if (shm_fileDes == -1) {
        printf("ERROR from Slave %d. Shared Memory failed; shm_open() failed: %s\n",childNum, strerror(errno));
        exit(1);
    }

    /*The mmap() function establishes a mapping between an address space of a process and a file associated with the file descriptor
    at the offset for length of bytes.*/
    shm_baseAdd = (struct CLASS *) mmap(0, SIZE,PROT_READ | PROT_WRITE, MAP_SHARED, shm_fileDes, 0);

    /*Checking for failure*/
    if (shm_baseAdd == MAP_FAILED) {
        printf("ERROR from Slave %d. Map failed; shm_mmap() failed: %s\n",childNum, strerror(errno));
        exit(1);
    }

    /*The sem_open() function opens a named semaphore, returning a semaphore pointer that may be used on
    subsequent calls to sem_post(), sem_wait(), and sem_close().

    The parameters of sem_open() are:
    1. name - a pointer to the null-terminated name of the semaphore to be opened
    2. oflag - option flags
    3. mode - permission flags
    4. value - initial value of the named semaphore */
    sem_t *mutex_sem = sem_open(semaphoreName, O_CREAT, 0666, 1);

    /*Checking for failure*/
    if (mutex_sem == SEM_FAILED) {
        printf("ERROR from Slave %d: sem_open failed(): %s\n",childNum, strerror(errno));
        exit(1);
    }

    /*The sem_wait() function decrements by one the value of the semaphore.
    The semaphore will be decremented when its value is greater than zero.
    If the value of the semaphore is zero, then the current thread will block until the semaphore's value becomes greater than zero.

```

```c
    The parameter of sem_wait() is:
    1. sem - a pointer to an initialized unnamed semaphore or opened named semaphore */
    if (sem_wait(mutex_sem) == -1) {
        printf("ERROR from Slave %d sem_wait() failed: %s/n",childNum, strerror(errno));
        exit(1);
    }

    /*Output acknowledgement message*/
    printf("Slave %d acquires access to shared memory segment, and structures it according to struct CLASS\n",childNum);

    /*Update index value*/
    i = &(shm_baseAdd->index);
    printf("Slave %d copies index to a local variable i\n",childNum);

    /*Write to shared memory*/
    shm_baseAdd->response[*i] = childNum;
    printf("Slave %d writes its child number in response[%d]\n",childNum,*i);

    /*Increment shared memory base address index*/
    *i += 1;
    printf("Slave %d increments index\n",childNum);

    /*The sem_post() function 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 semaphore value by one and continues running.

    The parameter of sem_post() is:
    1. sem - a pointer to an initialized unnamed semaphore or opened named semaphore */
    if (sem_post(mutex_sem) == -1) {
        printf("ERROR from Slave %d: sem_post() failed: %s\n",childNum, strerror(errno));
        exit(1);
    }

    /*The sem_close() function closes a named semaphore that was previously opened by a thread of the current process using sem_open().
    The sem_close() function frees system resources associated with the semaphore on behalf of the process.sem_close()

    The parameter for sem_close() is:
    1. sem - a pointer to an opened named semaphore. This semaphore is closed for this process.*/
    if (sem_close(mutex_sem) == -1) {
        printf("ERROR from Slave %d: sem_close() failed: %s\n",childNum, strerror(errno));
        exit(1);
    }

    /*The munmap() function shall remove any mappings for those entire pages containing
    any part of the address space of the process starting at (shared memory base) address and continuing for (SIZE) length bytes*/
    if (munmap(shm_baseAdd, SIZE) == -1) {
        printf("ERROR from Slave %d. Unmap failed; munmap() failed; %s\n",childNum, strerror(errno));
        exit(1);
    }

    /*The close() function is used to close a file, in this case it is used to close the shared memory segment as if it was a file*/
    if (close(shm_fileDes) == -1) {
        printf("ERROR from Slave %d. Close failed; close() failed: %s\n",childNum,strerror(errno));
        exit(1);
    }

    printf("Slave %d closes access to shared memory and semaphore and terminates\n",childNum);
    printf("Slave %d exits\n",childNum);

    /*Exit program*/
    exit(1);
}
```

mySh.h code:

mySh.h

```c
/* mySh.h*/
/* Header file to be used with master.c and slave.c
*/
struct CLASS {

    /*Index to next available response slot*/
    int index;

    /*Each child writes its child number here*/
    int response[10];
};

```