CECS 326-01

Operating Systems

Dylan Dang (ID 026158052)

Assignment 3

Due Date: 10/18/2022

Submission Date: 10/18/2022

Program Description

The purpose of this assignment is to 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(), shm_unlink() 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.

- 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
- Shm_unlink() remove the name of the shared memory object named by the string pointed to by the 'name' (shared memory segment name).

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 and the shared memory segment 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. The next big step for *master* is 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. 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.

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. The *slave* then writes its child number into the next available slot in the shared memory. 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 3$ gcc -o master master.c -lrt
dylandang@Dylan-Blade:~/Assignment 3$ gcc -o slave slave.c -lrt
dylandang@Dylan-Blade:~/Assignment 3$ ls
master  master.c  myShm.h  slave  slave.c
dylandang@Dylan-Blade:~/Assignment 3$
```

running master executable screenshot:

```
dylandang@Dylan-Blade:~/Assignment 3$ ./master 3 my_shm_name
Master begins execution
Master created a shared memory segment named my_shm_name
Master created 3 child processes to execute slave
Master waits for all child processes to terminate
Slave begins execution
I am child number 1, received shared memory name my_shm_name
I have written my child number to shared memory
Slave closed access to shared memory and terminates
Slave begins execution
I am child number 2, received shared memory name my_shm_name
I have written my child number to shared memory
Slave closed access to shared memory and terminates
Slave begins execution
I am child number 3, received shared memory name my_shm_name
I have written my child number to shared memory
Slave closed access to shared memory and terminates
Master received termination signals from all 3 child processes
Content of shared memory segment filled by child processes:
--- content of shared memory ---
1
2
3
Master removed shared memory segment, and is exiting
```

master.c code:

```
GNU nano 4.8                                                    master.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 "myShm.h"

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

    /*Before starting, make sure argument count is valid*/
    if (argc >= 3) {
        /*When executed, master outputs a message to identify itself*/
        printf("Master begins execution\n");

        /*Takes user input from commandline argument such as
        number of children and shared memory name and stores it*/
        char* numChildren = argv[1];
        char* segmentName = argv[2];

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

        /*Declare required size of memory request*/
        const int SIZE = 4096;

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

        /*Create shared memory file descriptor to reference values from response array*/
        int shm_fileDes;

        /*The shm_open() function returns a file descriptor that is associated with the shared "memory object" specified by name.
        This file descriptor is used by other functions such as mmap() & mprotect() to refer to the shared memory object.
        The state of the shared memory object, including all data associated with it, persists until the shared memory object is unlinked
        and all other references are gone.

        The parameters of shm_open() are:
        1. The name of the shared memory segment
        2. The value of oflag (O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_TRUNC) = read-only, read and write, create, error if create and file exists, trunca>
        3. The permission of the shared memory object*/
        shm_fileDes = shm_open(segmentName, O_CREAT | O_RDWR, 0666);

        /*The ftruncate() function truncates the file indicated by the open file descriptor to the indicated length (in this case 4096).
        If the file size exceeds length, any extra data is discarded. If the file size is smaller than length, bytes between the old and new lengths
        are read as zero. A change to the size of the file has no impact on the file offset.

        The parameters of ftruncate() are:
        1. The file descriptor
        2. The indicated length to be truncated to*/
        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.

        The parameters of mmap() are:
        1. The starting address of the memory region to be mapped
        2. The length
        3. The permitted access to the pages being mapped (read, write, execute, etc.)
        4. The handling of the mapped region (MAP_SHARED write references to the memory region by the calling process may change the file and are visib>
        5. The file descriptor
        6. The file byte offset at which the mapping starts*/
        shm_baseAdd = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fileDes, 0);

        /*Must convert numChildren from char to int so it can be used in the for loop*/
        int intChildren = atoi(numChildren);

        for (int i = 0; i < intChildren; i++) {
            /*fork() system call is used to create child process*/
            pid_t childPID = fork();
```

```
            /*Make sure that fork was successful. A '0' means the fork was successful*/
            if (childPID == 0) {
                char numList[10];

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

                /*Arguments that will be passed to the child program*/
                char *args[] = {numList, segmentName, NULL};

                /*The child process is executed via execv() system call which executes the file at the specific pathname
                which in this case is /slave */
                execv("./slave", args);

                /*Exiting child process*/
                exit(0);
            }
        }

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

        /*Waiting for all child processes to exit*/
        printf("Master waits for all child processes to terminate\n");
        while(wait(NULL) != -1);

        /*Output acknowledgement message*/
        printf("Master received termination signals from all %s 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 = 1; i < intChildren + 1; 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 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*/
        munmap(shm_baseAdd, SIZE);

        /*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*/
        close(shm_fileDes);

        /*The shm_unlink() function shall remove the name of the shared memory object named by the string pointed to by (segmentName)*/
        shm_unlink(segmentName);

        /*Removes the shared memory and then exits*/
        printf("Master removed shared memory segment, and is exiting\n");
    }

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

slave.c code:

```
 GNU nano 4.8                                                    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 "myShm.h"

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

    /*Before starting, make sure argument count is valid*/
    if (argc >= 2) {
        /*Output identifying message*/
        printf("Slave begins execution\n");

        /*Store number of children and shared memory name*/
        char* childNum = argv[0];
        char* segmentName = argv[1];

        /*Convert char to integer*/
        int int_childNum = atoi(childNum);

        /*Declare required size of memory request*/
        const int SIZE = 4096;

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

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

        /*Create shared memory file descriptor to reference values from response array*/
        int shm_fileDes;

        /*The shm_open() function returns a file descriptor that is associated with the shared "memory object" specified by name.
        This file descriptor is used by other functions such as mmap() & mprotect() to refer to the shared memory object.
        The state of the shared memory object, including all data associated with it, persists until the shared memory object is unlinekd
        and all other references are gone.

        The parameters of shm_open() are:
        1. The name of the shared memory segment
        2. The value of oflag (O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_TRUNC) = read-only, read and write, create, error if create and file exists, truncat>
        3. The permission of the shared memory object*/
        shm_fileDes = shm_open(segmentName, O_CREAT | O_RDWR, 0666);

        /*The ftruncate() function truncates the file indicated by the open file descriptor to the indicated length (in this case 4096).
        If the file size exceeds length, any extra data is discarded. If the file size is smaller than length, bytes between the old and new lengths
        are read as zero. A change to the size of the file has no impact on the file offset.

        The parameters of ftruncate() are:
        1. The file descriptor
        2. The indicated length to be truncated to*/
        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.

        The parameters of mmap() are:
        1. The starting address of the memory region to be mapped
        2. The length
        3. The permitted access to the pages being mapped (read, write, execute, etc.)
        4. The handling of the mapped region (MAP_SHARED write references to the memory region by the calling process may change the file and are visibl>
        5. The file descriptor
        6. The file byte offset at which the mapping starts*/
        shm_baseAdd = mmap(0,SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fileDes, 0);

        /*Writes its child number into the next available slot in the shared memory*/
        shm_baseAdd -> index = int_childNum;
        shm_baseAdd -> response[int_childNum] = int_childNum;

        /*Output acknowledgement message*/
        printf("I have written my child number to shared memory\n");
```

```
76
77          /*The munmap() function shall remove any mappings for those entire pages containing
78          any part of the address space of the process starting at (shared memory base) address and continuing for (SIZE) length bytes*/
79          munmap(shm_baseAdd, SIZE);
80
81          /*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*/
82          close(shm_fileDes);
83
84          /*Removes shared memory*/
85          printf("Slave closed access to shared memory and terminates\n");
86      }
87
88      /*Exit process*/
89      exit(0);
90  }
91
```

myShm.h code:

```
  GNU nano 4.8
 1  /* myShm.h*/
 2  /* Header file to be used with master.c and slave.c
 3  */
 4  struct CLASS {
 5
 6          /*Index to next available response slot*/
 7          int index;
 8
 9          /*Each child writes its child number here*/
10          int response[10];
11  };
12
```