```c
/**
 * Lab 6 assignment - Semaphores
 *
 * Create and use a semaphore correctly to allow two processes to
 * access the shared memory space exclusively.
 *
 * @author Ron Rounsifer
 * */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/sem.h>


#define SIZE 16

int main (int argc, char **argv) {
    int status;
    long int i, loop, temp, *shmPtr;
    int shmId, semID;
    pid_t pid;

    // semaphore instructions
    struct sembuf sem_wait = { 0, -1, 0 };
    struct sembuf sem_signal = { 0, 1, 0 };


    sscanf(argv[1], "%ld", &loop);

    printf("loop: %ld\n", loop);


    // create shared memory
    if ((shmId = shmget (IPC_PRIVATE, SIZE,
                         IPC_CREAT | S_IRUSR | S_IWUSR)) < 0) {
        perror ("i can't get no..\n");
        exit (1);
    }

    // attach pointer to first spot in memory location
    if ((shmPtr = shmat (shmId, 0, 0)) == (void *) -1) {
        perror ("can't attach\n");
        exit (1);
    }

    shmPtr[0] = 0;
    shmPtr[1] = 1;

    // create semaphore
    if ( (semID = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600)) == -1)
    {
        perror("main: semget");
        exit(1);
    }

    // init the semaphore to 1
    if (semctl(semID, 0, SETVAL, 1) == -1)
    {
        perror("main: semctl");
        exit(1);
    }

    // fork into a child process
    if (!(pid = fork ())) {

        // this loop is for the parent process
        for (i = 0; i < loop; i++) {
```

```
            // waits until the semaphore is non-negative
            if ( semop(semID, &sem_wait, 1) == -1)
            {
                perror("parent: sem_wait");
                exit(1);
            }

            // access shared memory
            temp = shmPtr[0];
            shmPtr[0] = shmPtr[1];
            shmPtr[1] = temp;

            // signal the semaphore
            if ( semop(semID, &sem_signal, 1) == -1)
            {
                perror("parent: sem_signal");
                exit(1);
            }

        }
        if (shmdt (shmPtr) < 0) {
            perror ("just can 't let go\n");
            exit (1);
        }
        exit (0);
    }
    else {

        // this loop is for the child process
        for (i = 0; i < loop; i++) {

            // wait until memory is free
            if (semop(semID, &sem_wait, 1) == -1)
            {
                perror("child: sem_wait");
                exit(1);
            }

            // access shared memory
            temp = shmPtr[0];
            shmPtr[0] = shmPtr[1];
            shmPtr[1] = temp;

            // signal semaphore
            if (semop(semID, &sem_signal, 1) == -1)
            {
                perror("child: sem_signal");
                exit(1);
            }

        }
    }

    wait (&status);
    printf ("values: %li\t%li\n", shmPtr[0], shmPtr[1]);

    // remove the semaphore
    if (semctl(semID, 0, IPC_RMID) < 0)
    {
        perror("main: semaphore removal");
        exit(1);
    }

    // detach from shared memory
    if (shmdt (shmPtr) < 0) {
        perror ("just can't let go\n");
        exit (1);
    }

    // deallocate shared memory
    if (shmctl (shmId, IPC_RMID, 0) < 0) {
        perror ("can't deallocate\n");
        exit (1);
```

```
    }
    return 0;
}
```