Program 1.5

This is a program to see if forked processes will respect a shared memory semaphore. The results and explanation is in the header comments.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>

// Ziad Arafat
// Sep 25 2021

/**
 * Changelog:
 * Sep 25 2021
 * Used my old OS class code to create and manage shared memory
 * Added another part of my old OS class code to create an IPC semaphore
 *
 * Sep 26 2021
 * BUGFIX: Semaphore was not being respected by both processes because the
 * sem_init() call was not enabling shared memory semaphore by setting the
 * second parameter to 1.
 *
 * Sep 27 2021
 * Cleaned up and documented code
 *
 */

/**
 * @brief This program will demonstrate if a forked process will respect a
 * shared memory semaphore created in the parent code. We will achieve this
by
 * forcing the child to enter a critical segment protected by a mutex first
and
 * then forcing the parent to wait 10 seconds on the child. Print
statements
 * will help us know the order in which code was executed. If we succeed
then
 * the parent will never enter the critical section until the child has
exited
 * it. Otherwise the mutex is not being respected.
 */

/**
 * Results:
```

```
 * This is the output from running the code
 * We can know that the semaphore behaves correctly because the parent
 * does not enter the mutex until after the child has left.
 *
 * parent making child and waiting 2 seconds
 * child in mutex
 * child leaving mutex
 * child has left the mutex
 * parent in mutex
 * parent is out of mutex
*/

// create key for shared memory
#define SHMKEY ((key_t)1497)

// define struct to store value in
// This might not be necessary but this is how I did it in OS class.
typedef struct {
        sem_t mut;
} shared_mem;

// initialize memory
static shared_mem *mutex;

int main()
{
        int shmid; // The shared memory id
        int pid1; // The pid returned by the forked process

        char *shmadd_ptr; // The address of the shared memory
        shmadd_ptr = (char *)0;

        // If shmget doesn't return a valid memory address then barf
        if ((shmid = shmget(SHMKEY, sizeof(int), IPC_CREAT | 0666)) < 0) {
                perror("uh oh, shmget failed");
                exit(1);
        }

        // check if the shared mem attached correctly.
        if ((mutex = (shared_mem *)shmat(shmid, shmadd_ptr, 0)) ==
            (shared_mem *)-1) {
                perror("shmat failed");
                exit(1);
        }

        sem_init(&mutex->mut, 1, 1); // initialize the semaphore to 1

        printf("parent making child and waiting 2 seconds\n");

        if ((pid1 = fork()) == 0) { // parent process
                sleep(2); // Wait 2 seconds so the child will enter first

                sem_wait(&mutex->mut); // try to enter critical segment.
```

```
                // This must not run before child leaves the mutex
                printf("parent in mutex\n");

                sem_post(&mutex->mut);

                printf("parent is out of mutex\n");

                waitpid(pid1, NULL, WEXITED); // wait for child to close

                sem_destroy(&mutex->mut); // clean up the mutex.

                // if we can't delete the memory, barf
                if (shmdt(mutex) == -1) { // clean up the shared memory

                        // barf
                        perror("Cant delete the shared memory");
                        exit(-1);
                }
                exit(0); // exit parent

        } else { // child process
                sem_wait(&mutex->mut); // enter the critical segment
                printf("child in mutex\n");

                sleep(10); // sleep for 10 seconds so the parent has to
wait.

                printf("child leaving mutex\n");

                sem_post(&mutex->mut);

                printf("child has left the mutex\n");

                exit(0); // exit child
        }

        return 0;
}
```