Lab 6

By Jarred Parr and Alexander Fountain

1. The program uses the child to swap two values in shared memory and then the parent swaps those values back.
2. The output, in an ideal case, will always be `values: 0 1`
3. There is a race condition present where the value may be double read and can cause the output to be `0 0` or `1 1` as a result.
4. As was stated in problem 3, there is an issue in read and write times for the shared memory. Because of the context switch in an undefined location in the loop, we can see that `temp` is assingned a value, say 0, then the context switches and the other loop replaces `temp` with the value, say 1, and then swaps the pointers. Because this loop finishes, the program context switches back to the first process, but temp now has a value of 1 instead of zero, this causes us to see that both values would end up at 1 in each index of the shared memory.
5. The 3 options for `struct sembuf` are `sem_num`, `sem_op`, and `sem_flg`. These 3 options are specified as the following:

- `sem_num` - The semaphore number. This specifies the semaphore that the operation will be performed on.
- `sem_op` - The semaphore operation. This specifies the operation that the semaphore will be performing.
- `sem_flg` - The semaphore flag. You can initialize some of these values with certain flags that may affect creation constraints or runtime aspects within the semaphore. Flags faciliate that feature of the semop. The options `IPC_NOWAIT` and `SEM_UNDO` allow the program to run and undo the opertation when the process is done respectively. These options allow the field to control how semaphores work without needing to be extremely strict.

6. `SEM_UNDO` undoes the specified operation after the program running has terminated. This allows the kernel to back out of whatever the operation is running if the program exits unexpectedly. This gives the ability of cleaning up shared namespaces and memory, at the cost of all of the calls needing to now be able to access things in the kernel space if you're running it in that level of privilge. This is most useful in situations like this, because when running in the kernel space, a user could send a SIGKILL to a process and it might not effectively clean up the semaphore since it was not explicitly flagged.

Practical:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

```c
#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, sem_id;
  pid_t pid;
  struct sembuf sem[2];

  loop = atol(argv[1]);
  printf("Loop count: %ld\n", loop);

  if ((shmId = shmget (IPC_PRIVATE, SIZE, IPC_CREAT|S_IRUSR|S_IWUSR)) < 0) {
    perror ("i can't get no..\n");
    exit (1);
  }
  if ((shmPtr = shmat (shmId, 0, 0)) == (void*) -1) {
    perror ("can't attach\n");
    exit (1);
  }

  key_t key;
  key = ftok("Secret", 7);

  if ((sem_id = semget(key, 1, IPC_CREAT | 0666)) == -1) {
    perror("Failed to make the semaphore");
    return EXIT_FAILURE;
  }

  shmPtr[0] = 0;
  shmPtr[1] = 1;
  semctl(sem_id, 0, SETVAL, 1);
  sem[0].sem_num = 0;
  sem[0].sem_op = -1;
  sem[0].sem_flg = 0;
  sem[1].sem_num = 0;
  sem[1].sem_op = 1;
  sem[1].sem_flg = 0;

  if (!(pid = fork())) {
    for (i=0; i<loop; i++) {
      semop(sem_id, &sem[0], 1);
      temp = shmPtr[0];
      shmPtr[0] = shmPtr[1];
      shmPtr[1] = temp;
```

```
      semop(sem_id, &sem[1], 1);
    }
    if (shmdt (shmPtr) < 0) {
       perror ("just can't let go\n");
       exit (1);
    }
    exit(0);
  }
  else {
    for (i=0; i<loop; i++) {
      semop(sem_id, &sem[0], 1);
      temp = shmPtr[1];
      shmPtr[1] = shmPtr[0];
      shmPtr[0] = temp;
      semop(sem_id, &sem[1], 1);
    }
  }

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

  if (shmdt (shmPtr) < 0) {
    perror ("just can't let go\n");
    exit (1);
  }
  if (shmctl (shmId, IPC_RMID, 0) < 0) {
    perror ("can't deallocate\n");
    exit(1);
  }

  return 0;
}
```

Library

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <time.h>
#include "lib.h"

#define LOCK -1
#define UNLOCK 1
#define NO_CHANGE -4
#define CHANGE 4
```

```c
void wow() {
  printf("WOWZERS");
}

sem_t* sem_t_create(key_t key, struct sembuf* buf, int flags, int nsems) {
  sem_t sem;

  if ((sem.sem_id = semget(key, nsems, flags)) < 0) {
    perror("Error, failed to initialize semaphore!");
  }

  sem.buf = *buf;

  return &sem;
}

void sem_t_initialize(sem_t* sem, int sem_op, int sem_flg) {
  sem->buf.sem_num = sem->sem_id;
  sem->buf.sem_op = sem_op;
  sem->buf.sem_flg = sem_flg;
}

int sem_t_timed_wait(sem_t* sem, const struct timespec* timeout) {
  if (sizeof(sem->buf) == 0) {
    goto not_initialized;
  }
  /* The sem buffer must be initialized here*/
  if (semtimedop(sem->sem_id, sem->buf, 1, timeout) < 0) {
    return -1;
  }

  return 0;

not_initialized:
  perror("semaphore not initialized!");
  return EXIT_FAILURE;
}

void sem_t_lock(sem_t* sem) {
  sem->waiting = 1;
  sem->buf.sem_flg = LOCK;
}

void sem_t_unlock(sem_t* sem) {
  sem->waiting = 0;
  sem->buf.sem_flg = UNLOCK;
}

int sem_t_signal(sem_t* sem) {
  /* A process can signal if the semaphore is waiting */
```

```
  if (sem->waiting) {
    sem_t_unlock(sem);
    return CHANGE;
  }

  /* Otherwise, do nothing */
  return NO_CHANGE;
}

void destroy(sem_t* sem) {
  semctl(sem->sem_id, 0, IPC_RMID);
}
```