CIS 452
Lab 6 Report

Ashley Hendrickson
Muna Gigowski
Fall 2019

The Shared Memory Problem

Question One

What exactly does Sample Program 1 intend to do (i.e. who is responsible for what operations)?

After the shared memory is created and attached, the program assigns 0 and 1 to shmptr[0] and shmPtr[1], respectively. After this initial setup, the program forks and within the child's code it loops however many times that were passed in by the command line argument, each iteration swapping the values of shmPtr[0] and shmPtr[1]. After the child is done looping, it calls shmdt() to detach itself from the shared memory, then exits. Meanwhile, the parent's code it is also performing this same loop and operation, swapping shmPtr[1] and shmPtr[0] "loop" number of times. Afterwards, the parent calls wait() until the child has finished. Once wait returns with the child program's exit status, the parent prints out the final values of shmPtr[0] and shmPtr[1] before also detaching from the shared memory and destroying it. Finally, the parent program exits.

Question Two

What is the program's expected output?

The expected output is that the values of shmPtr[0] and shmPtr[1] will print out in the parent's code at the end with the same values they were initially assigned to (before any of the looping happened), since both the child and parent are looping the same number of times, but reverse-swapping the other program's swap (child is swapping 0 -> 1, while at the same time the parent is swapping 1 -> 0, so the one swap is reverted by the other swap).

Coincidentally, when we ran the program many times with a low value for "loop," that is exactly what happened. The values of shmPtr[0] and shmPtr[1] kept printing out at the end as 0 and 1, which were the same values that were initially assigned to them. However, this expected output did not hold as we began testing the program with larger and larger numbers, as detailed in question three.

Question Three

Describe the output of the Sample Program as the loop values increase.

As the loop value increases, the values of shmPtr[0] and shmPtr[1] begin to print out at the end with values different from the ones they were initially assigned - some runs they both printed at the end with a value of 0, sometimes they both printed with a value of 1, and sometimes the values were swapped from their initial values.

Question Four

Describe *precisely* what is happening to produce the observed interesting output. Your answer should tie in to the concepts discussed in Chapter 5 of your textbook -- Process Synchronization.

Even though both the parent and child processes seem to be looping and swapping values at the same time, that is not necessarily true. Because this program does not try to synchronize the two processes in any way, it is possible that during any given iteration the processes fall out of synch - with one performing a swap of the two values at the same time the other program is also swapping the values of the two shared variables. With both programs swapping the values at the same time it is impossible to know what the final output of the two values will be, since both processes are editing the values of the shared variables simultaneously, in no consistent order. This problem becomes much more likely the more iterations we do, which is why this lack of synchronization becomes apparent with the use of very high loop values.

Shared Memory Synchronization

Question Five

Name and describe in your own words the use of the three fields of the sembuf structure.

The sembuf structure contains the following three fields:

1. unsigned short sem_num - the semaphore number
2. short sem_op - operation to be performed on the semaphore
3. short sem_flg - operation flags (either IPC_NOWAIT or SEM_UNDO)

Question Six

What is the purpose of the SEM_UNDO flag (i.e. why would you use it)?
Note: make sure you truly understand what this flag does.

Sem_UNDO flag control makes semop(2) allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to this previous state. If the process dies, the system applies the operation in the undo structures

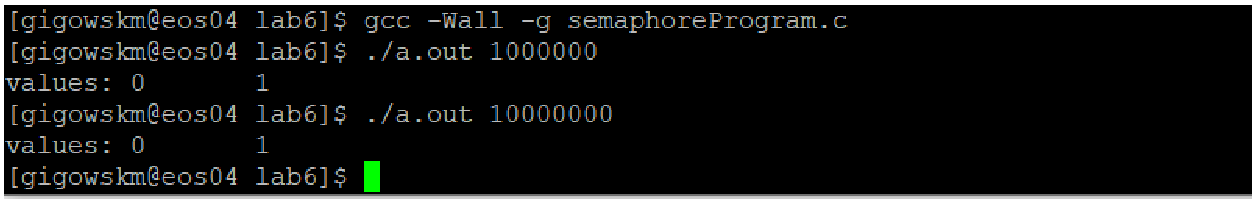
You would want to use it to prevent a process with exclusive use of a semaphore terminating abnormally, and failing to undo the operation freeing the semaphore. This would cause the semaphore to stay locked in memory in the state the process left it in.

Programming Assignment (Controlled Process Synchronization)

Source Code

```
#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
#define MUTEX 2

//sBuf has 3 properties in it: unsigned short sem_num, short sem_op, and
short sem_flg
struct sembuf waitBuf = { 0, -1, 0 };
struct sembuf signalBuf = { 0, 1, 0 };

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

      // get value of loop variable (from command-line argument)
      loop = atoi(argv[1]);

          //create a new semaphore set for use by this (and other) processes
          if((semId = semget (IPC_PRIVATE, 1, 00600)) < 0) {
                perror ("semget failed\n");
                exit (1);
          }

          //initialize the semaphore set referenced by the previously obtained
semId handle
          if(semctl (semId, 0, SETVAL, 1) < 0) {
                perror ("semctl initialization failed\n");
                exit (1);
          }

   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);
   }

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

   if (!(pid = fork())) {
      for (i=0; i<loop; i++) {
               // swap the contents of shmPtr[0] and shmPtr[1]
                          //critical section!!
                          semop(semId, &waitBuf, 1);
               temp = shmPtr[0];
               shmPtr[0] = shmPtr[1];
               shmPtr[1] = temp;
                          semop(semId, &signalBuf, 1);
      }
      if (shmdt (shmPtr) < 0) {
         perror ("just can't let go\n");
         exit (1);
      }
      exit(0);
   }
   else
      for (i=0; i<loop; i++) {
               // swap the contents of shmPtr[1] and shmPtr[0]
                          //critical section!!
                          semop(semId, &waitBuf, 1);
               temp = shmPtr[1];
               shmPtr[1] = shmPtr[0];
               shmPtr[0] = temp;
                          semop(semId, &signalBuf, 1);
      }

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

   // remove the semaphore referenced by semId - may need to move the
location of this line
       if(semctl (semId, 0, IPC_RMID) < 0) {
         perror ("error removing semaphore\n");
      exit(1);
        }

   return 0;
}
```

Sample Output



```
[gigowskm@eos04 lab6]$ gcc -Wall -g semaphoreProgram.c
[gigowskm@eos04 lab6]$ ./a.out 1000000
values: 0        1
[gigowskm@eos04 lab6]$ ./a.out 10000000
values: 0        1
[gigowskm@eos04 lab6]$
```