CIS 452 Lab Week #6
Shared Memory Synchronization

Overview

The purpose of this assignment is to become familiar with the methods used for synchronizing process access to shared memory; specifically, the coordination of process' entry into their critical sections. The UNIX IPC package, in the form of shared memory and semaphores, will be investigated and used as the mechanism for controlling process synchronization.

Hand-in:

- A word document containing the answer to the numbered questions.
- Any requested screenshots (uploaded as separate files)
- Program source code (no zip files)

The Shared Memory Problem

Because of its high speed and low overhead, developers often use shared memory as an interprocess communication mechanism. This technique is implemented by allowing processes to share the same data space. However, asynchronous access to shared memory may result in unusual and incorrect results. As mentioned in class and in your textbook, shared memory access presents users with an instance of the critical section problem. Recall the main issues involved with process synchronization while studying the following program:

sampleProgramOne

```
#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>

#define SIZE 16
```

```
int main(int argc, char *argv[])
{
   int status;
   long int i, loop, temp, *sharedMemoryPointer;
   int sharedMemoryID;
   pid_t pid;

   loop = atoi(argv[1]);

   sharedMemoryID = shmget(IPC_PRIVATE, SIZE, IPC_CREAT|S_IRUSR|S_IWUSR);
   if(sharedMemoryID < 0) {
      perror ("Unable to obtain shared memory\n");
      exit (1);
   }

   sharedMemoryPointer = shmat(sharedMemoryID, 0, 0);
   if(sharedMemoryPointer == (void*) -1) {
      perror ("Unable to attach\n");
      exit (1);
   }

   sharedMemoryPointer[0] = 0;
   sharedMemoryPointer[1] = 1;


   pid = fork();
   if(pid < 0){
     printf("Fork failed\n");
   }

   if(pid == 0) { // Child
      for(i=0; i<loop; i++) {
          // swap the contents of sharedMemoryPointer[0] and
sharedMemoryPointer[1]
      }
      if(shmdt(sharedMemoryPointer) < 0) {
         perror ("Unable to detach\n");
         exit (1);
      }
      exit(0);
   }
   else
     for(i=0; i<loop; i++) {
       // swap the contents of sharedMemoryPointer[1] and
sharedMemoryPointer[0]
     }

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

   if(shmdt(sharedMemoryPointer) < 0) {
      perror ("Unable to detach\n");
      exit (1);
   }
   if(shmctl(sharedMemoryID, IPC_RMID, 0) < 0) {
```

```
      perror ("Unable to deallocate\n");
      exit(1);
   }

   return 0;
}
```

Perform the following operations and answer the questions:

Study the sample program. Add the necessary code to perform the 'swap' in the loops. A temp variable has already been declared.

1. What exactly does sampleProgramOne intend to do (i.e., who is responsible for what operations)? (6 points)
2. Given that, what is the program's expected output? (4 points)

Compile Sample Program 1. Run the program multiple times with increasing **loop** values (e.g., 10, 100, 1000, ...) until you observe interesting *and* unexpected results. You may need to run the program on a shared device (EOS/Arch).

3. Describe the output of the sample program as the **loop** values increase (4 points)
4. Describe precisely what is happening to produce the unexpected output. Your answer should tie in to the concepts discussed involving process synchronization. (6 points)

Shared Memory Synchronization

This lab concerns itself primarily with mechanisms for controlling access to shared memory. Like shared memory, semaphores are a kernel resource and follow the familiar resource usage pattern: request, use, release.

For semaphores, the resource usage paradigm is expressed as follows:

a) Obtain a semaphore set (often of size 1)
b) Initialize the semaphore element(s)
c) Use the semaphores correctly (e.g. **wait()...signal()**)
d) Remove the kernel resource data structure

Each of the steps above has an associated system call. The system calls are very similar to the calls used for shared memory. One of the main differences is that the functions are constructed to work with "sets" of semaphores - however, they can easily be used to manipulate a single semaphore. The functions corresponding to the above steps are:

a) **semget()** - this function creates a semaphore set and initializes its elements to 0. It initializes a kernel data structure for operating system management of the resource and returns a resource ID to the user. This ID is used by any process wishing to use the

semaphore.
b) **semctl()** - used for controlling the resource. This function is typically used to set (i.e. initialize) or query the values of semaphore elements.
c) **semop()** - used for incrementing, decrementing and testing the value of semaphore elements.
d) **semctl()** - used for controlling the resource. This function is also used to "free" the resource; removing the semaphore and its associated data structures from the system.

Refer to the man pages for additional details on semaphore operation

The code snippets below demonstrate the use of these system calls:

/* Create a new semaphore set for use by this (and other) processes.. */
semId = semget (IPC_PRIVATE, 1, 00600));

/* Initialize the semaphore set referenced by the previously obtained **semId** handle. */
semctl (semId, 0, SETVAL, 1);

/* The **semop()** function is used to perform operations on semaphore elements (for example, to do the equivalent of the wait() and signal() operations). **semId** is the semaphore set ID obtained previously, **sbuf** is the name of a user variable (i.e., a data object) of type **sembuf**. The values of the **sembuf** structure are filled in by the user and used by the system to determine the nature of the operation to be performed. Refer to the man pages and the .h files for more detail on this structure and its use.
*/
semop (semId, &sbuf, 1);

// Remove the semaphore referenced by **semId**
semctl (semId, 0, IPC_RMID);

Note that the **semctl()** function call is used for initializing and also for removing the semaphore. It uses a different syntax (and a different number of arguments) depending on the usage desired.

Perform the following operations:

5. Name and describe in your own words the use of the three fields of the **sembuf** structure (9 points)
6. What is the purpose of the **SEM_UNDO** flag (i.e., why would you use it)? (6 points)

Miscellaneous Notes

- Recall that the system utility "**ipcs**" (IPC status) displays information about currently allocated kernel IPC resources, including semaphores. The system utility "**ipcrm**" (IPC remove) is used for semaphore removal. Use this utility to clean up resources in the event of programming errors.
- The process that creates shared memory or semaphores should be responsible for:
    - Creation
    - Verifying that no segment or semaphore set with the same ID number exists
    - Establishing access permissions
    - Cleanup and deallocation
- All of these tasks are controlled by the flags passed to the respective **get()** and **ctl()** functions. As usual, read the man pages!
- Similar to signals, there are both System V and POSIX versions of semaphores. All systems implement the SysV version (which you should use for this lab).

Programming Assignment (Controlled Process Synchronization) (25 points)

The programmer's goal is to implement controlled, asynchronous access to shared memory; in this lab that translates to properly synchronizing access to the critical sections of sampleProgramOne. The main idea of this assignment is to demonstrate that with the use of proper synchronization mechanisms, the expected value is *always* obtained from the program.

- Protect the critical sections in sampleProgramOne to prevent memory access conflicts from causing inconsistencies in the output. Your solution should still maximize potential parallelism.
    - Insert the appropriate code to create and initialize a semaphore
    - Use semaphore operations to synchronize the two processes
    - Perform required cleanup operations
    - Submit a screenshot of your program working properly

Note: Semaphore creation and initialization are two different, hence non-atomic operations. Be sure they have both been completed before another process attempts to access the semaphore.