CS 452 Lab: Week #5
Managing Shared Memory

Overview

The purpose of this assignment is to become familiar with the methods used to enable processes to create and access shared memory (the fastest form of InterProcess Communication). The UNIX IPC package will be investigated as the mechanism for implementing a multiple-process project that communicates via shared memory. Of course, whenever executing entities share data, it poses an instance of the Critical Section problem. The programming assignment suggests a simple synchronization method to solve this problem.

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)

Shared Memory

One of the interprocess communication (IPC) methods mentioned in class was the use of shared memory. Because of its speed of access (as opposed to say, files or sockets), it is a useful tool for application programmers. This is why shared memory is commonly supported by modern operating systems and utilized in many advanced applications. However, shared data presents users with an instance of what is called the critical section problem.

Normally, each process has its own unique address space. The purpose of shared memory is to allow processes to communicate by sharing the *same* data area. Each process still has its own data segment, but with an additional shared segment mapped into its virtual address space. Processes may access this shared segment only via pointers; this is known as anonymous memory. The reading and modifying of memory, and hence process communication, occurs in the usual manner but with pointers.

This lab is concerned with the access mechanism itself: learning to use the system calls that implement and manage shared memory. Since shared memory is a kernel resource, programs that use it should adhere to the established resource usage pattern: request, use, release. For shared memory specifically, the resource usage paradigm is expressed as follows:

a) Create a shared memory region
b) Attach the shared memory segment to a process' address space
c) Detach the shared memory region from a process' address space
d) Remove the kernel resource data structure

All of the above steps have an associated system call. The system calls corresponding to the above steps are:

a) **`shmget()`** - this function creates a shared memory segment. It initializes a kernel data structure to manage the resource and returns a resource ID to the user. This ID, or handle, is used by any process wishing to access the shared segment.
b) **`shmat()`** - attaches the specified shared memory region to the address space of the calling process.
c) **`shmdt()`** - detaches the specified shared memory region.
d) **`shmctl()`** - used for controlling the resource. This function has different uses that range from returning information about the shared memory segment to locking the memory. It can also be used to "free" the resource; removing the shared memory segment and its associated data structures from the system.

Refer to the man pages for additional details on these functions.

The sample code below demonstrates the use of these system calls (i.e., the mechanics of obtaining and using shared memory). Note that this program doesn't really do anything useful - because it doesn't make sense for a single process to use shared memory.

sampleProgramOne

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define FOO 4096

int main ()
{
   int shmId;
   char *sharedMemoryPtr;

   if((shmId = shmget(IPC_PRIVATE, FOO, IPC_CREAT|S_IRUSR|S_IWUSR)) < 0) {
      perror ("Unable to get shared memory\n");
```

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

   printf("Value a: %p\t Value b: %p\n", (void *) sharedMemoryPtr, (void *)
sharedMemoryPtr + FOO);

   if(shmdt (sharedMemoryPtr) < 0) {
      perror ("Unable to detach\n");
      exit (1);
   }
   if(shmctl (shmId, IPC_RMID, 0) < 0) {
      perror ("Unable to deallocate\n");
      exit(1);
   }

   return 0;
}
```

Perform the following operations and answer the questions:

Compile and run sampleProgramOne

1. What is being output by sampleProgramOne (i.e., what is the *meaning* of the output values) ?
2. Read the man pages; then describe the meaning/purpose of each argument used by the **`shmget()`** function call.
3. Describe two specific uses of the **`shmctl()`** function call.
4. Read the man pages, then use **`shmctl()`** to modify sampleProgramOne so that it prints out the size of the shared memory segment. What changes/lines do you have to add to the program?

Useful System Utilities

The system utility "**`ipcs`**" (IPC status) displays information about all currently allocated kernel IPC resources. Use the man pages to familiarize yourself with this program.

Since shared memory is a kernel resource, it is persistent (i.e., it may remain in the system even after the creating process has terminated). Shared memory is normally deallocated via the **`shmctl()`** function, but the system utility "**`ipcrm`**" (IPC remove) can also be used when necessary. Use this utility in the event of programming errors. For example, if your program terminates before cleaning up, this utility can be used to free the resources.

Perform the following operations:

- Modify the print statement in sampleProgramOne to determine the ID of the shared memory segment
- Insert a `pause()` after the print statement, recompile and run
- Terminate the Sample Program (`^C`) and run the `ipcs` utility
- Take a screenshot
- Use the `ipcrm` utility to remove the shared memory segment
- Re-run the `ipcs` utility to verify that it worked
- Take a screenshot

Programming Assignment (Readers and Writer)

Process communication using shared memory involves at least two processes by definition, often referred to as a Reader and a Writer. The programming assignment for this lab is to implement a Writer program that creates a shared memory segment, then repeatedly accepts an arbitrary user input string from the terminal and writes it into shared memory. You must also implement a corresponding Reader program; two instances of it will attach to the same shared memory segment. The Readers will repeatedly read each new string from the shared memory and display the string to the screen. The idea is similar to the "echo" command, or to the operation of the pipe demo sample program from a previous lab, except that your solution must implement two Reader processes. In this sense, it functions a little like the typescript utility, which sends output to both the display and to a file. Your program sends output to two different window displays.

Write separate programs for each process type and then run each process in its own terminal window (3 windows total). This clearly and visually demonstrates the communication mechanism involved in using shared memory.

- The Writer needs to know that the current string has been displayed, so that it may write a new string into shared memory (otherwise it would overwrite the first string). The Readers need to know when a new string has been written into shared memory, so that they can display it (otherwise they would re-display the current string). The solution to this problem does not require sophisticated synchronization mechanisms such as semaphores or mutexes (the topic of a future lab). You should be able to synchronize for the desired behavior with simple elements such as flag(s) and/or turn variables, which must also be stored in shared memory so that all processes can access them. Note that the functionality of this system constitutes a type of "lockstep" synchronization, which is acceptable in this case since that is the

desired behavior of the programs (writer goes, then both readers go, then writer goes, etc.).

- The Readers need to know which segment to attach to. The preferred method is for the Readers and Writer to agree upon a *passkey* that is used to get access to the resource. See the man pages for information on using the function **`ftok()`** (file_to_key), which is used to generate a common key. Please use the source code in the current path to allow for easy grading (as opposed to a file that will not be on my machine).
- The programs should conclude with a graceful shutdown, releasing all system resources before terminating (this would be a good use of the **`signal()`** mechanism). The graceful shutdown should be able to occur from the user typing 'quit' or Control-C.

- **Important:** don't lose your shared memory pointer! Otherwise, you will not be able to properly detach.

If you are developing this program on a shared resource (EOS/Arch) please be sure to use the utilities discussed in this lab to release any resources your program may have failed to release.

Submit screenshots of your successful program execution.