Lab 08 - Shared Memory

Objectives

1. Get an introduction to programming with shared memory.
2. Use the shared memory to solve a typical IPC problem.

Pre-reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Overview

As seen in the lab on synchronization methods, shared blocks of memory are a common solution when programing. A block of memory is 'shared' if it is mapped into the address space of two or process. This is different than having threads started by a process and sharing certain variables. Although care is required to synchronize access to this block of memory, this is a useful technique for applications were it is not practical to move/copy large chunks of data around. In some cases it is faster to just work on the same buffer. On the other hand, we can also use the buffer itself to move chunks of data between processes in a producer/consumer relationship.

When a process stores information into a location in the shared memory, it can be read by any other process that is using the same segment of shared memory. Within an individual computer, uniprocessor or multiprocessor, shared memory is normally the fastest way for two processes to share information.

The shared memory API commonly used in contemporary versions of POSIX was introduced in System V Unix. Even though the mechanism allows multiple processes to map a common memory segment into their own address spaces – it is logically part of the memory manager – it was designed and implemented in System V Unix as part of the IPC (inter-process communication) mechanism.

The Shared Memory API

Shared memory can be used to allow any process to dynamically define a new block of new that is independent of the address space created for the process before it began to execute. Every Linux process is created with a large virtual address space, only a portion of which is used to reference the compiled code, static data, stack, and heap. The remaining addresses in the virtual address space are initially unused. A new block of shared memory, once defined, is mapped into a block of the unused virtual addresses, after which the process can read and write the shared memory as if it were ordinary memory. Of course, more than one process can map the shared memory block into its own address space, so code segments that read or write shared memory are normally considered to be critical sections (and should be protected by a synchronization primitive such as a mutex, a semaphore or a monitor as we've seen in the bounded buffer lab).

The following four system calls define the kernel interface for shared memory support.

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

int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
void *shmdt(char *shaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- **`shmget()`** creates the shared memory block.
- **`shmat()`** maps an existing shared memory block into a process's address space.
- **`shmdt()`** removes (unmaps) a shared memory block from the process's address space.
- **`shmctl()`** is a general-purpose function (in the style of **`ioctl()`**) used to perform all other commands to the shared memory block.

To create a new block of shared memory, the process must call **`shmget()`**. If **`shmget()`** successfully creates a new block of memory, then it returns a shared memory identifier of type int. The shared memory identifier is a handle into a kernel data structure.

The first arguments to **`shmget()`** is of type **`key_t`**. This may be either the key of an existing memory block, 0, or **`IPC_PRIVATE`**. When it is **`IPC_PRIVATE`**, the **`shmget()`** call creates a new block of shared memory. When it is 0, and the **`IPC_CREAT`** flag is set in the third argument, **`shmflg`**, a new block of memory will be created. A process that wants to reference a shared memory block that was created by another process can set the key argument to the key value of an existing memory block. The **`shmflg`** also defines the access permissions for user, group, and world accesses to the memory block as its lower 9 bits (by using the same bit pattern as for file protection). The second argument specifies the size of the new block of memory. All memory allocation operations are in terms of pages. That is, if a process requests 1 byte of memory, then the memory manager allocates a full page (**`PAGE_SIZE`** = 4,096 bytes on i386 machines). Thus the size of the new shared memory block will be the value of the size argument rounded up to the next multiple of the page size. A size of 1 to 4,096 results in a 4K (one-page) block, 4,097 to 8,192 results in an 8K (two-page) block, and so on.

Often different processes want to attach to the same memory segment, which requires that both of them use the same key. A useful function in this situation is:

```
key_t ftok(const char *pathname, int proj_id);
```

which converts a pathname (which must refer to an existing, accessible file) and a project ID (a nonzero integer) to a key.

The `void *shmat(int shmid, const void*shmaddr, int shmflg)` system call maps the memory block into the calling process's address space. The **`shmid`** argument is the result returned by the **`shmget()`** call that created the block. The **`shmaddr`** pointer is the virtual address to which the first location in the shared memory block should be mapped. The value of **`shmaddr`** should be aligned with a page boundary. If the calling process does not wish to choose the address to which the memory block will be mapped, it should pass a value of 0 for **`shmaddr`** and the operating system will choose a suitable address at which to attach the memory segment. If **`shmaddr`** is specified, and **`SHM_RND`** is asserted in **`shmflg`**, then the address will be rounded down to a multiple of the **`SHMLBA`** constant. The **`shmflg`** is used in the same way as the corresponding flag in **`shmget()`**, that is, to assert a number of different 1-bit flags to the **`shmat()`** system call. In addition to asserting the **`SHM_RND`** flag, the calling process can assert **`SHM_RDONLY`** to attach the memory block so that it can be only read, and not also written.

When a process finishes using a shared memory block, it calls `void *shmdt(char *shaddr)`, where **`shaddr`** is the address used to attach the memory block. Note that the call to **`shmdt`** only detaches the memory segment from the address space of the current process; it does not delete the shared memory segment (even if the process detached was the last process using it). To delete the shared memory segment **`shmctl()`** must be used.

The final shared memory call is `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`. This call performs control operations on the shared memory block descriptor. The **`shmid`** argument identifies the shared memory block, the **`cmd`** argument specifies the command to be applied and the **`buf`** argument represent a pointer to a structure containing the different properties to be read from or applied to the shared memory segment. Check the manual page of **`shmctl`** to see the possible values of the **`cmd`** argument and the effect of each of them. If the **`cmd`** argument is **`IPC_RMID`**, then the shared memory segment is removed (if it's not in use) or it is marked for removal when the last process detaches from it.

Here is a simple example of creating a block of shared memory:

```
#include <sys/shm.h>

#define SHM_SIZE 2000
...
int shm_handle;
char* shm_ptr;

// create shared memory
shm_handle = shmget(IPC_PRIVATE,SHM_SIZE,IPC_CREAT | 0644);

// valid handle?
if(shm_handle == -1) {
      printf("shared memory creation failed\n");
      exit(0);
}

// attach to shared block, see man pages for detail
shm_ptr = (char*)shmat(shm_handle,0,0);
if (shm_ptr == (char *) -1) {
      printf("Shared memory attach failed");
      exit(0);
}

// do work, share results through the shared memory segment
// have other processes start and also use the same shared memory
...

// detach from the shared memory segment
shmdt(shm_ptr);

// remove the shared memory segment
shmctl(shm_handle,IPC_RMID,0);
```

Semaphores

Like in the case of any shared resource, the access to the shared memory must be synchronized. A simple way of synchronizing the access to shared memory regions is to use semaphores. A semaphore is a synchronization primitive represented by an integer value on which two operations can be performed: *signal* and *wait*. These are both atomic operations,   signal increments the value of the semaphore and wait decrements the value of the semaphore. If the value of the semaphore is zero, the process that performs a wait operation on that semaphore will block until another process performs a signal operation on the semaphore.

The POSIX standard specifies the following functions for working with semaphores:

`sem_open()`: initializes and opens a new semaphore

```
sem_t *sem_open(const char *name, int oflag,
                  mode_t mode, unsigned int value);
```

`sem_wait()`: decrements and locks the semaphore (the wait operation)

```
int sem_wait(sem_t *sem);
```

`sem_post()`: increments and unlocks the semaphore (the signal operation)

```
int sem_post(sem_t *sem);
```

Check the manual pages of this function for additional information on how to use them.

Procedure

We have already done a lab about the producer-consumer problem. In this assignment you have to solve this problem again, but this time using a shared memory region as the common buffer shared between two distinct processes (last time we used two threads within the same process) and semaphores for synchronizing the access to the memory region.

You will have to create two distinct programs, *producer* and *consumer* each of them taking a single argument, the name of a file:

```
$ ./producer input_file
```

```
$ ./consumer output_file
```

The producer will read data from the input_file and store it into the shared buffer and the consumer will read the data from the shared buffer and save it into the output_file. Both processes repeat this until the entire content of the input file has been transferred to the output file, after which they terminate. At the end, the output_file should be identical to the input file.

The input_file can be any type of file (image, audio or video file), so you shouldn't assume that it's a text file.

Deliverables

All source and header files, and a make file. Name your source files producer.c and consumer.c.