Project 3a: Multiprocess Programming with Shared Memory in Linux

You are likely to find that it is not possible to divide this project up by "server" vs "client" (and have one person work on each part) since they need to work together closely; you are likely to have the most success if you work together. This project will not be a lot of code, but it can be tricky to use the suggested library routines correctly. You may find the discussion here useful.

Objectives

There are numerous objectives to this assignment:

1. To use shared memory across cooperating processes.
2. To use semaphores for mutual exclusion between processes.
3. To catch signals (such as SIGINT) with a signal handler.

Overview

In this assignment, you will implement a **client** and **server** that communicate through a **shared memory page** to display statistics about the client processes. All of the processes are running on the **same machine**. Each client process will periodically **write** to the shared memory page with updates about its recent behavior (e.g., how much CPU it has been allocated); the server process collects this information (by **reading** from the shared memory page) and periodically displays the information for all the client processes. Meanwhile, each client also periodically displays the PIDs of all the processes that are currently running.

As we will soon cover in the lectures, when processes (or threads) cooperate through shared memory, they require synchronization primitives (such as locks or semaphores) to ensure that they do not have race conditions when they are each simultaneously updating the same memory locations. To minimize our need for synchronization in this Project (synchronization is for Project 4!), we will construct the clients and server so that only a single client is (usually) writing to each memory location and the server is only reading (not writing); if the server happens to occasionally read data that is not up-to-date, that is okay, since the data is just usage statistics (and not your bank account). The only time you will need to worry about mutual exclusion will be when clients are first starting and when they exit.

For this project, you will be implementing two components: a server process that displays client statistics every second, and a client process that sleeps and displays the PIDs of all the processes that are currently running every second.

Server Process

Let us consider what the server process, `shm_server`, must do.

First, it is the responsibility of the server process to create and initialize the shared memory page. A shared memory page can be created with `shm_open()` as follows:

```
int fd = shm_open(SHM_NAME, O_RDWR | O_CREAT, 0660);
```

This routine essentially maps the shared memory page to the tmpfs (temporary file system). After calling this routine, you will find this file at `/dev/shm/$SHM_NAME`. Therefore, its very convenient to inspect your shared memory page and debug your code via dumping this file using tools like `xxd`.

Because everyone will create the file at `/dev/shm/`, you have to use a unique file name to avoid conflicts. Therefore, you are required to use the following convention to name this file:

For example, if Zhewen works with Xiangjin for this project, then `SHM_NAME = "zhewen_xiangjin"` where `zhewen` is Zhewen's CS login and `xiangjin` is Xiangjin's CS login. If you are working alone, then simply put your own CS login as the file name.

Second, you should use `ftruncate()` to cause this file to be truncated to a size of precisely length bytes. For this project, you can only create **a single page** of shared memory, i.e. 4096 Bytes.

Third, you should use `mmap()` and the argument `flags` in this routine should be `MAP_SHARED` because we are sharing this mapping.

If anything goes wrong with this setup (e.g., the shared memory page cannot be exclusively created), then the `shm_server` should exit with return code 1.

After the server process creates this single page of shared memory and initializes it appropriately, it should enter an infinite loop where it continually sleeps for one second, reads the contents of shared memory, and then displays the statistics that have been written to shared memory to STDOUT.

There should be one line of output for each client process; the format of each line of output should be as follows:

```
pid : [pid], birth : [dateOfBirth], elapsed : [sec] s [msec] ms
```

The `dateOfBirth` should be the string containing the date and time when the client starts in a human-readable format `Www Mmm dd hh:mm:ss yyyy`, where `Www` is the weekday, `Mmm` the month in letters, `dd` the day of the month, `hh:mm:ss` the time, and `yyyy` the year. You may not need to convert the time to this format yourself. Try to find a C library function to do that for you!

The `elapsed` should be the time that's elapsed since the time of the process started. The `sec` should be an integer and `msec` should be displayed with four digits after the decimal point; you may have either the client or server perform this formatting.

For example, if there are two client processes, the output may look like the following:

```
pid : 14084, birth : Wed Jul  5 22:59:27 2017, elapsed : 30 s 4.8310 ms
pid : 14085, birth : Wed Jul  5 22:59:27 2017, elapsed : 30 s 4.5600 ms
```

If at some time, a new client starts writing to the shared memory page, the output could look like this:

```
pid : 14084, birth : Wed Jul  5 22:59:27 2017, elapsed : 34 s 5.4670 ms
pid : 14085, birth : Wed Jul  5 22:59:27 2017, elapsed : 34 s 5.1960 ms
pid : 14276, birth : Wed Jul  5 23:00:02 2017, elapsed : 0 s 0.0010 ms
```

Finally, if a client (pid : 14085) terminates at some time, the output could look like this:

```
pid : 14084, birth : Wed Jul  5 22:59:27 2017, elapsed : 91 s 14.4980 ms
pid : 14276, birth : Wed Jul  5 23:00:02 2017, elapsed : 57 s 10.0560 ms
```

You will need to define a structure for each client's statistics. The details of this structure are up to you. We suggest treating the shared memory page as an array of these structures. We've provided a definition for the type `stats_t` here.

```
typedef struct {
  int pid;
  char birth[25];
  int elapsed_sec;
  double elapsed_msec;
} stats_t;
```

You must use the fields we have defined without modifying them, but you can add to that structure. However, you cannot add too many fields such that **`sizeof`** `stats_t > 64`. See reasonings below.

Your server must be able to handle **63 clients**. If more than 63 clients at a single time try to use this shared memory page, those clients should receive an error.

When a client terminates, it must reset its segment (or some bytes of its segment depending on your implementations) in the shared memory page so that the segment can be used by another client later.

One of your challenges when implementing the server will be to determine the number of valid clients; don't print out any garbage if there are fewer than 63 running clients!

To ensure that only one client at a time is searching through and modifying the essential structures of the shared memory page, you should use a semaphore to provide mutual exclusion to protect the critical sections.

Now let's explain why we should limit the size of the structure `stats_t` to be at most 64 Bytes. We only created the shared memory for one page, i.e. 4096 Bytes. Now we divide the page into 64 segments, and each segment has size of 64 bytes. We are using 63 of them to store the statistics of the 63 clients. One segment is reserved to store the semaphore. In short, the shared memory page layout should be 1 segment (semaphore) + 63 segments (clients' statistics).

To use the semaphore like a simple mutex lock, you should initialize the semaphore as

```
sem_init(mutex, 1, 1);
```

The clients can then use `sem_wait()` to essentially acquire the mutex lock and `sem_post()` to release the lock.

When your server terminates, you will need to ensure that it correctly removes the shared memory page so that your `/dev/shm/$`**`SHM_NAME`** would not remain there forever. To do so, you need to call `munmap()` and `shm_unlink()`. Again, read the manuals carefully to learn how to correctly use them!

To help you start with, we have provided the code frame of shm_server.c.

Client Process

Your client process, `shm_client`, should behave as follows.

Multiple clients may connect simultaneously to the same server and same shared memory page.

You should first obtain the shared memory page pointer by using `shm_open()` and `mmap()`. Note that there is a subtle difference in the argument of `shm_open()` in the client as opposed to the server because the server has already **created** the shared memory page and the client should just **open** this particular preexisting page.

There is no need to initialize the semaphore (or mutex lock) in the client (because it was already initialized in the server). All you need to do here is to point the semaphore to the segment that stores the semaphore in the shared memory page.

Now your client process will iterate forever. For each iteration, it first updates and writes its statistics to the particular segment of the shared memory page; then it sleeps for one second; finally it prints out the PIDs of current active clients. Below is part of the example output from one client:

```
Active clients : 14084 14085 14276
Active clients : 14084 14085 14276
Active clients : 14084 14276
```

We assume that you will kill your client program by sending it a SIGINT signal (with Ctrl-C) or a SIGTERM signal (by the command line `kill [PID]`). Usually, SIGINT and SIGTERM interrupts your program and kills it. However, you can change this default behavior by specifying a signal handler that should be run when that particular signal is delivered. To do this, use `sigaction()` to specify the routine that you want to run. This new routine should make sure that the segment is reset (or some bytes of its segment depending on your implementations) such that the associated segment can be used by another client later, and then call `exit`. You may find online examples like this very useful.

Similarly, the server will be killed by receiving the signal. So clean the shared memory page and exit elegantly when the server terminates.

To help you start with, we have provided the code frame of shm_client.c.

Compiling, Makefile, and Testing

You should compile your code like

```
gcc shm_server.c -o shm_server -Wall -Werror -lrt -lpthread
```

To ensure that we compile your server and your client process correctly, you will need to create a simple **Makefile**. The makefile must make both of the targets. If you don't know how to write a makefile, you might want to look at the man pages for `make`.

To run the test scripts, go to the directory where your source codes reside and type

```
~cs537-1/ta/tests/3a/runtests
```