Project 3a: Using Shared Memory in Linux

You may do Part A of this project with a partner.

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.

UPDATE: For consistency, all functions, files, and types should begin with the string "stats_" and not "stat_" as some may have been named before.

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 at SIGINT) with a signal handler.
4. To create a dynamically shared library.

Overview

In this assignment, you will implement a **client** and **server** that communicate through a **shared memory segment** 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 segment with updates about its recent behavior (e.g., how much progress it has made and how much CPU it has been allocated); the server process collects this information (by **reading** from the shared memory segment) and periodically displays the information for all the client processes.

As you know by now, 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 three components: a server process that displays client statistics every second, a library containing two functions for updating those statistics, and a client process that sleeps and computes at rates specified by command-line arguments. With these statistics, we hope that you will be able to see more about how the Linux scheduler treats processes at different priorities and with different behavior.

Server Process

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

First, it is the responsibility of the server process to create and intialize the shared memory segment. A shared memory segment can be created with `shmget()`.

Each shared memory segment is identified with a unique key; this key is used to match requests for a particular segment between the server and the clients. You should obtain this key from the command line; that is, when the server is started from the command line, it expects a single flag `-k key`. For example:

```
stats_server -k 931
```

starts the server using 931 as the key for the shared memory segment.

To attach this shared memory segment to the server's address space, you will use the `shmat()` routine. You will need to carefully read the man pages for `shmget()` and `shmat()` to understand how to use them correctly.

If anything goes wrong with this setup (e.g., the shared memory segment cannot be exclusively created), then the `stats_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 (followed by an extra blank line after each second); the format of each line of output should be as follows:

```
[server_iter] [pid] [argv[0]] [counter] [cpu_secs] [priority]
```

For example, if there are two client processes, the 21st time through the server's loop, the output may look like the following:

```
21 1267 hello_world 36 1.26 10
21 1272 cpu_work 4056 0.56 18

```

Then, the next second (the 22nd iteration), the output could look as follows:

```
22 1267 hello_world 52 2.06 10
22 1272 cpu_work 5391 0.86 18

```

If in the next second, a new client starts writing to the shared memory segment, the next output could look like this:

```
23 1267 hello_world 88 2.83 10
23 1272 cpu_work 6183 1.87 18
23 1277 sleep_work 2 0.01 16
```

Finally, if a client terminates, then in the next second, the output could look like this:

```
24 1267 hello_world 126 3.51 10
24 1277 sleep_work 5 0.02 16
```

For the name of the client process, regardless of the actual length of argv[0], the server should display only up to 15 characters; whether the client or the server performs this truncation is up to you.

Similarly, `cpu_secs` must be displayed with two digits after the decimal point (e.g., 5.36); you may have either the client or server perform this formatting.

So that the client and the server agree on the format of the shared memory segment, 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 segment as an array of these structures. We've started a definition for the type `stats_t` in the file stats.h; you must use the fields we have defined without modifying them, but you can add to that structure. Include `stats.h` in all your `.c` files.

Note that the server does not compute or interpret any of these statistics; the server simply displays the information that it reads from the shared memory segment (though it might do truncating or formatting).

The server should create only a **single page** of shared memory. Don't guess or hard-code this value; figure out how you can find the right page size at run-time!

Your server must be able to handle **16 clients**. If more than 16 clients at a single time try to use this shared memory segment, those clients should receive an error.

When a client terminates, it must reset its entry in the shared memory page so that the entry 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 16 running clients!

To ensure that only one client at a time is searching through and modifying the essential structures of the shared memory segment, you should use a semaphore to provide mutual exclusion across both `stats_init()` and `stats_unlink()` (described below)..

To create a semaphore that can be used across unrelated processes, you should use `sem_open`. Again, you will want to look at the man pages for those routines carefully. The server should create this semaphore when it is started. To use the semaphore like a simple mutex lock, you should initialize the semaphore with `sem_open` to have the starting value of 1 with code something like the following:

```
if ((mutex = sem_open("mysemaphore -- key?", O_CREAT, 0644, 1)) == SEM_FAILED) {
  perror("sem_open");
  exit(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 segment and semaphore so that each resource does not remain allocated forever. Look into `shmctl()` with `cmd=IPC_RMID` and `sem_unlink` to delete these resources. How do you find out when your server terminates, given that it runs in an infinite loop? See the same discussion for the client process below.

You may find it useful to run the command `ipcs` from your shell's command line to see information about current shared memory segments and semaphores (especially if you run into problems). While your final working server should delete shared segments itself, if you need to delete them by hand while debugging, the command `ipcrm shm [shmid]` may be extremely useful.

UPDATE: As you all know, when you use sem_open() to create a new semaphore, you specify a name for that semaphore (as the first argument). If you get unlucky with your naming choice and you pick the same name as someone else running on your machine, you may get unexpected behavior (e.g., you won't have permission to open the existing semaphore).

To ensure that your program behaves as expected, we strongly recommend that you use your cslogin as a unique prefix in your semaphore name (e.g., for me, I could use "dusseau_sem" or "dusseauP3" or "dusseauXYZ"). Note that when we run your P3a code for grading, your code could run into this name conflict problem -- so fix this even if you haven't seen this problem in your own experience.

Client Library Functions

To simplify the task of constructing client processes you will encapsulate the functionality of interacting with the server into two library routines: `stats_init()` and `stats_unlink()`.

- `stats_t* stats_init(key_t key)`: attempts to attach to an existing shared memory segment with the specified key. If this is successful, it should return a pointer to the portion of the shared memory segment that this client should write to for its statistics; if it is not successful (e.g., the shared segment with the desired key does not exist or too many clients are already using the segment for statistics), it should return NULL. Each client that wishes to use the statistics monitor must call `stats_init()`.
- `int stats_unlink(key_t key)`: removes the calling process from using the shared memory segment. If successful, it returns 0; if not successful (e.g., this process, or the process with this pid, did not call stats_init), it should return -1.

To define the `stats_t` type you should use the defintion that has been started in stats.h. You are expected to add new fields to this structure so that all of the necessary statistics are shared between the clients and server. However, you may not change the names or types of the fields that are already defined there (i.e., pid, counter, priority, and cpu_secs).

One of your challenges for the client library code will be how to determine which structures in the shared memory segment correspond to currently active clients so that a new client can be assigned to the next (free) structure (and the corresponding address returned).

To ensure that only one client at a time is searching through and modifying the essential structures of the shared memory segment, you should use a semaphore to provide mutual exclusion across both `stats_init()` and `stats_unlink()`.

Within your `stats_init()` and `stats_unlink()` make sure only one client process at a time is modifiying the essential fields that you use to determine whether or not an entry is valid. You can use `sem_wait` to essentially acquire the mutex lock (created by the server) and `sem_post` to release the lock.

You must provide these two routines in a shared library named "libstats.so". Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with your code. There are further advantages to shared (dynamic) libraries over static libraries. When you link with a static library, the code for the entire library is merged with your object code to create your executable; if you link to many static libraries, your executable will be enormous. However, when you link to a shared library, the library's code is not merged with your program's object code; instead, a small amount of stub code is inserted into your object code and the stub code finds and invokes the library code when you execute the program. Therefore, shared libraries have two advantages: they lead to smaller executables and they enable users to use the most recent version of the library at run-time.

To create a shared library named libstats.so, use the following commands (assuming your library code is in a single file "stats.c"):

```
gcc -c -fpic stats.c -Wall -Werror
gcc -shared -o libstats.so stats.o
```

To link with this library, you simply specify the base name of the library with "-lstats" and the path so that the linker can find the library "-L.". You'll need something like this:

```
gcc -o myprogram mymain.c -Wall -Werror -lstats -L.
```

Of course, these commands should be placed in a Makefile. Before you run "myprogram", you will need to set the environment variable, LD_LIBRARY_PATH, so that the system can find your library at run-time. Assuming you always run myprogram from this same directory, you can use the command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Note that export is what you use in bash; if you are not using bash, you'll have to figure out the alternative command to set the environment.

Client Process

To use your library functions, you will need to create a client process. Multiple clients may connect simultaneously to the same server and same shared memory segment.

After performing some initialization (and calling stats_init), your client process will forever iterate between sleeping and computing; at the end of each iteration, it will increment a **counter** and ensure that it has updated all the statistics that are being tracked about it.

This client process, `stats_client` should accept the following flags (which may appear in any order):

```
stats_client -k key -p priority -s sleeptime_ns -c cputime_ns
```

For example:

```
stats_client -k 841 -p 10 -s 1000 -c 1000000
```

If other flags are specified, `stats_client` should print an error message and exit with return code 1. If a flag is not specified, choose a reasonable default value.

These fields should have the following effects:

- **key** : key is the shared memory key that is passed to `stats_init()`
- **priority** : this process should set its priority, using `setpriority`, to this value. Note that the value returned by `getpriority()` should be used for the to update the corresponding field in the `stats_t` structure. You will want to read the man pages for setpriority and getpriority. Note that lower values are higher priorities and that processes that do not have superuser priveleges cannot set their priorities higher than the default of 0.
- **sleeptime_ns** Each iteration, the process should sleep for sleeptime_ns (nanoseconds). The function `nanosleep` is good for this functionality.
- **cputime_ns** : Each iteration, the process should compute for cputime_ns (nanoseconds). To busy-wait (or spin-wait) for this amount of time, the process could repeatedly call a function like `clock_gettime` to see how much cputime has transpired.

If you encounter any invalid flag values, exit with return code 1.

To summarize, the `stats_client` will update its statistics by writing to its entry in the shared memory segment after every iteration. It needs to ensure that the following information is communicated to the server: its `pid`, its value of `argv[0]` (e.g., "stats_client") (perhaps truncated), the current value of the counter that indicates how many iterations have completed so far, the cummulative amount of CPU time this process has acquired since it was started (again, `clock_gettime` could be useful), and its current priority (as returned by `getpriority`).

Finally, when should your client program call `stats_unlink()` given that it runs an infinite loop? We assume that you will kill your client program by sending it a SIGINT signal (with Ctrl-C). Usually, SIGINT 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 me run. This new routine should make sure that `stats_unlink()` is called and then call `exit`.

Compiling, Makefile, and Testing

You are likely to need to link with a number of libraries to create your server and client. To ensure that we compile your server, your library, and your client process correctly for the demo, you will need to create a simple **makefile**. The makefile must make all three targets. If you don't know how to write a makefile, you might want to look at the man pages for `make`.

Make sure the file **stats.h** defines your completed main structure, again, with no changes to the fields we defined.

For testing your stats library, we may also create our own client processes with different behavior. We will include your `stats.h` file and therefore expect to be able to reference the fields we defined.

We will again verify that your code passes lint and valgrind tests, as in Project P1a and P2a.