Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

# Lab: Inter-process communication

In this lab, you will understand how to write programs using various Inter-Process Communication (IPC) mechanisms.

## Before you begin

Familiarize yourself with various IPC mechanisms, including shared memory, named pipes, and Unix domain sockets.

## Warm-up exercises

Do the following exercises before you begin the lab.

1. You are given two programs that use POSIX shared memory primitives to communicate with each other. The producer program in the file `shm-posix-producer-orig.c` creates a shared memory segment, attaches it to its memory using the `mmap` system call, and writes some text into that segment. You can see the shared memory file under `/dev/shm` after the producer writes to it. The consumer program in `shm-posix-consumer-orig.c` opens the same shared memory segment, reads the text written by the producer, and displays it to the screen. Read, understand and execute both programs to understand how POSIX shared memory works. These examples are from the famous OS textbook by Gagne, Galvin, Silberschatz. A sample run of this code is shown below. Note the library used during compilation.

```
$ gcc -o prod shm-posix-producer-orig.c -lrt
$ gcc -o cons shm-posix-consumer-orig.c -lrt
$ ./prod
$ cat /dev/shm/OS
Studying Operating Systems Is Fun!
$ ./cons
Studying Operating Systems Is Fun!
```

2. You are given two programs that communicate with each other using Unix domain sockets. The server program `socket-server-orig.c` opens a Unix domain socket and waits for messages. The client program `socket-client-orig.c` reads a message from the user, and sends it to the server over the socket. The server then displays it. The programs can be compiled as follows.

```
$ gcc -o client socket-client-orig.c
$ gcc -o server socket-server-orig.c
```

You must first start the server:

```
$ ./server
Server ready
```

Then, start the client, type a message, and check that it is displayed at the server.

```
$ ./client
Please enter the message: hello
Sending data...
```

3. Write two simple programs that communicate with each other using a named pipe or FIFO. Make one of the processes send a simple message to the other process via the pipe, and let the other process display the message on the screen.

## Part A: Sharing strings using shared memory

In this part of the lab, you will write two programs, a producer `shm-posix-producer.c` and consumer `shm-posix-consumer.c`. The producer and consumer share a 4KB shared memory segment. The producer first fills the shared memory segment with 512 copies of the 8-byte string "freeeee" (7 characters plus null termination character) indicating that the shared memory is empty. Then, the producer repeatedly produces 8-byte strings, e.g., "OSisFUN", and writes them to the shared memory segment. The consumer must read these strings from the shared memory segment, display them to the screen, and "erase" the string from the shared memory segment by replacing them with the free string. The consumer should also sleep for some time (say, 1 second) after consuming each string, in order to digest what it has consumed! The producer and consumer should exchange 1000 strings in this manner. Since there are only 512 slots in the shared memory segment, the producer will have to reuse previously used memory locations that have been consumed and freed up by the consumer as well.

You can use the starter code `shm-posix-producer-orig.c`/`shm-posix-consumer-orig.c` given to you to get started. But note that in the original programs, the shared memory segment is opened only for reading at the consumer, while this part of the lab requires the consumer to write to the shared memory as well when freeing it up. So you will have to change permission flags to the various system calls suitably.

How does the consumer know when and where the producer has written a string to the shared memory? The consumer can constantly keep reading the shared memory segment for a string that is different from the free string pattern, but this is inefficient. Instead, you must open another channel of communication between the producer and consumer, using named pipes or message queues or any other IPC mechanism. Whenever the producer writes a string to the shared memory segment, it sends a message to the consumer specifying the location (you can use byte offset or any other way to encode the location) of the string it has written. The consumer repeatedly reads messages from the producer on this channel, finds out the location of the string it must consume, and then consumes it. You must be careful in ensuring that the consumer reads the exact same number of bytes written by the producer on this channel.

How does the producer know when a string has been consumed, and the coresponding location freed-up, by the consumer? Once again, you can make the producer scan the shared memory segment to find empty slots to produce in, or the consumer can send a message to the producer via some channel to inform it about free slots. This design choice is left up to you, and you can use the inefficient method of scanning for free slots if you desire.

Once you write both programs, test them for a smaller number of iterations (instead of 1000) to check that the shared memory is being used correctly. You may also print out the contents of the shared memory segment for debugging purposes. You must also test your code for varying amounts of sleep time at the consumer. When the sleep time is small or 0, you will see that the producer and consumer finish quickly. However, for longer sleep times, and for more than 512 iterations, you will notice that the producer slows down while waiting for space to be freed up by the consumer. Play around with different sleep times to convince yourself that your code is working correctly.

## Part B: File transfer using Unix domain sockets

In this part of the lab, you will write two programs, a client program `socket-client.c` and a server program `socket-server.c` which communicate with each other over Unix domain sockets to transfer a file. The client takes a filename as argument, opens and reads the file in chunks of some size (say, 256 bytes) from disk using open/read system calls, and sends this file data over the socket to the server. The server receives data from the client and displays it on screen. When you run the server in one window, and the client in another, you should see that the content of the file whose name you have given to the client is displayed in the server terminal. Ideally, the programs should also terminate when the file transfer is complete. though this needs a bit of work to achieve, so doing this is optional.

## Submission instructions

- You must submit the files `shm-posix-producer.c` and `shm-posix-consumer.c` for part A, and the files `socket-client.c` and `socket-server.c` for part B.
- Place these files and any other files you wish to submit in your submission directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.