ITF22519 - Introduction to Operating Systems

Lab8: Interprocess Communication (IPC)

Fall 2020

We have now began to see how two (or more) processes in a machine can communicate. In some applications, there is a clear need for some processes to communicate with each other. There are several approaches of IPC in a system. These approaches can simply be a way for one process to communicate or share some information to another process. In this lab, we will take a look at the following approaches:

- Pipe (unnamed and named pipe)
- Signals
- Shared Memory
- Named Semaphores
- Message Queues
- Sockets

Before you start, remember to `commit` and `push` your previous lab to your git repository. Then, try to `pull` the new lab:

```
$ cd ITF22519/labs
$ git pull upstream master
$ cd lab8
```

1 Pipe

The simplest form of IPC is a `pipe`. In the textbook, pipe is a pseduo file which is considered as a special file. There are unnamed pipe and named pipe (FIFO)

1.1 Unnamed Pipe

When you pipe the output of one program into the input of another, you are creating an unnamed pipe. For example, this command in Bash would create a pipe between `lsmod` and `grep` processes.

```
$ lsmod | grep acpi
```

Another way to create an unnamed pipe is to use the `pipe(2)` system call in C. This function creates a unidirectional pipe and returns a two element array where the first element is the read and the second element is to write end of the pipe. The `pipe()` function can be called before the `fork()` system call. Then, the child and parent processes can use it to communicate. For more information about pipes, use `pipe(2)` and `pipe(7)`.

Exercise 1 (10pts)

The program `pipe_test.c` uses unnamed pipe.

- Run the code example and explain the output.
- How the output changes if the `sleep` statement in child process is moved before `fgets` statement of the parent?

1.2 Named Pipe

Another approach to create a pipe is a named pipe, known as a FIFO due to its behavior, by using `mkfifo` command in your terminal or `mkfifo(3)` library call in C. A FIFO behaves exactly the same way as a pipe except that it has a name so that any process can reference to it. Now, open two terminals in the same directory and create a new FIFO using `mkfifo`.

```
$ mkfifo fifo_test
```

In one terminal, run the following command:

```
$ cat fifo_test
```

This command is now watching FIFO and will show anything written to the FIFO. Now, in another terminal, type the following command:

```
$ echo "Hi FIFO" > fifo_test
```

Exercise 2 (15 pts)

- What happens when you run the `echo` command?
- What happens if you run `echo` command first and then `cat` command?
- Use `man fifo(7)`, where is the data is sent through FIFO store?

Exercise 3 (15pts)

Write two programs `read_fifo.c` and `write_fifo.c` which will be run on two terminals:

- `write_fifo.c` while true: reads one line that user enters into one terminal and writes it to a named fifo.
- `read_fifo.c` while true: reads any input from the same named fifo and print them out to the other terminal.
- Do we need to synchronize the two programs to ensure that the read operation will always happen after the write operation?

2 Signals

Signals are something you have already dealt with, you just didn't know it! Simply speaking, a signal is a special command that a program receives from the kernel. Anytime you have typed `CTRL+C` to close a program, you have been sending the `SIGKILL` signal to that program. Another signal you have already seen is the segmentation fault signal `SIGSEGV`. To find a list of signals and to learn more about them, look at the man page `signal(7)`. These signals have well defined actions they perform when a program receives them (e.g., quit the program). You can define your own signals and signal handlers and use them for IPC. Note that this is a little different than the other forms of IPC because it can't really be used to send data, only signals.

Exercise 4 (10 pts)

Compile and run the program `sig_test.c` and answer the following questions. To exit the program, press `CTRL + Z`.

- What happens when you try to quit the program by using `CTRL + C`?. Explain the result.
- What is the signal number that `CTRL + C` send?

Exercise 5 (10 pts)

Compile and run the program `sig_fork.c` and answer the following questions:

- What is the output of the program?
- Explain the output, specify how variable `count` is increased in child and parent processes.

Exercise 6 (10 pts)

Write a program `sig_com.c` that exploits signal for the communication between child and parents processes as follow:

- Child process:

```
while(true)
   read input from user
   if input is "quit"
      break the loop
   else
      send SIGUSR1 to the parent using kill function
```

- Parent process waits for the child to exit. On receiving `SIGUSR1` signal, print a message: "Received SIGUSR1 from my child".

3 Shared Memory

A shared memory space (SHM) is a method of IPC which allows the applications to share a region of memory between them. Any variables or data in this shared memory area is accessible to all processes which open the SHM. Please read the man page overview `shm_overview(7)`, `shm_open(3)` for more information.

Exercise 7 (25pts)

Compile and run the files `shm_test1.c` and `shm_test2.c`. You will need to compile with the `-lrt` flag to link against `librt`. Observe the output by running both applications at the same time. Answer the following questions

- What is the output if you run both at the same time calling `shm_test1.c` first?
- What is the output if you run both at the same time calling `shm_test2.c` first?
- Why is `shm_test2.c` causing a segfault? How could this be fixed?
- What happens if the two applications both try to read and set a variable at the same time? (e.g. `shared_mem->count++`)?
- How can a shared memory space be deleted from the system?

4 Named Semaphores

Named semaphores are the same as the semaphores you used in Lab7 except they can be shared between multiple processes without being in shared memory space. The naming convention is the same as for shared memory spaces, a name starting with a / followed by an alpha-numeric string. For more information, look at `sem_overview(7)` and `sem_open(3)`. Semaphores are not very useful on their own for IPC, but they can be used with these other types of IPC as a very powerful tool. For example, a semaphore may be used to protect a shared pipe or shared memory to prevent multiple processes from writing at the same time.

Exercise 8 (15 pts)

Please answer the following questions in your report:

- How long do semaphores last in the kernel?
- What causes them to be destroyed?
- What is the basic process for creating and using named semaphores? (list the functions that would need to be called and their order).

Exercise 9 (15pts)

Write two programs `read_shm.c` and `write_shm.c` which will be run on two terminals:

- `write_shm.c` while true: reads one line that a user enters into one terminal (maximum 1000 characters) and copy it to a string inside a shared memory area.
- `read_shm.c` while true: reads the string from the same shared memory area and prints it out to the other terminal.
- Use named semaphore to synchronize the two programs so that `read_shm.c` will always start after `write_shm.c`.

5 Sockets

Sockets are really a special type of pipe in UNIX systems. They allow two-way communication between two processes which may be running on two different computers connected by a network or may be on the same system.

Exercise 10 (10pts)

Please use man pages `socket(7), socket(2)` answer the following questions:

- What are the six types of sockets?
- What are the two domains that can be used for local communications?

6 Message Queues

Message queues are a way of passing priority messages from one process to another. One nice feature of message queues is the ability to subscribe to events on the queue and to handle the events asynchronously. Look at the man page `mq_overview(7)`, `mq_open(3)`, `mq_send(3)`, `mq_receive(3)` and `mq_notify(3)` for more information.

Exercise 11 (15pts)

Compile the files `mq_test1.c` and `mq_test2.c` with the `-lrt` flag to link with `-lrt`. Have two terminals open; in the first start `mq_test1` and then in the other start `mq_test2`. In your report answer the following questions:

- What is the output from each program?
- What happens if you start them in the opposite order?
- Change `mq_test2.c` to send a second message which is "My name is X" where "X" is your name. Change `mq_test1.c` to wait for and print this second message before exiting.

Exercise 12 (15pts)

Write two programs `read_mq.c` and `write_mq.c` will be run on two terminals:

- `write_mq.c` while true, reads one line that user enters into one terminal and send it to a message queue.
- `read_mq.c` while true reads the message from the same message queue and print it out to the other terminal.
- Do we need to synchronize the two programs to ensure read operations will always happen after write operations?

7 What To Submit

Complete all tasks under each section and submit your source files to GitHub. Complete the exercises in this lab, and make your `lab8_report` file. After that, put all of files into the **lab8** directory of your repository. Run `git add.` and `git status` to ensure the file has been added and commit the changes by running `git commit -m "Commit Message"`. Finally, submit your files to GitHub by running `git push`. Check the GitHub website to make sure all files have been submitted.