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.

Part A: Sharing strings using shared memory

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!
```

Next, you will extend the programs given above, and write two programs, `shm-posix-producer.c` and `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? One way to solve this problem is that 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. Another way is to open another channel of communication between the producer and consumer, say, using named pipes. 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.

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.

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

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...
```

Next, you will extend the above code, and 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 in chunks 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. For example, you may have to read the size of the file using the `fstat` system call at the client, and communicate it to the server before the transfer, so that the server knows when the transfer has completed, and can terminate itself.

Part C: Sharing bitstrings using signals

In this part, we will use signals to communicate a string of 0s and 1s between a parent and child process. You are given template code `bitstring-send.c`, where a parent process forks a child, and reads a bit string of 0s and 1s of length 8 from the user. You must now add code to send this string to the child process via signals, and the child must print out the string it has received. You can use any two different signals to denote 0 and 1. You may want to use user-defined signals to avoid terminating the child accidentally. You can find out more by typing `man 7 signal` in the terminal. The parent must use the `kill` system call to send signals to the child, and when the child receives and handles the signals, it can infer the 0 or 1 being sent by the parent. The child must terminate after receiving the string and must be reaped by the parent, after which the parent terminates too.

We have provided a synchronization mechanism in the code to ensure that the parent process only starts sending the bit after the child process is ready to receive it. We also let the parent sleep between the transmissions (via `sleep 1`) to ensure the child receives all of the signals slowly, one after the other, without the signals getting jumbled up. This sleep is essential for correct communication, but can lead to a high delay in communicating the string.

You can optionally optimize your transmission by adding a way for the child to inform the parent that it has received the previous bit, so that the parent can proceed to the next bit faster. For example, you can send a signal back from the child to the parent when the child receives the bit. We must also make appropriate modifications in the parent process to ensure that the parent does not send the next bit until the child has received the previous one. Hint: Look at the synchronization mechanism used in this code at the start of the communication, could we perhaps do something similar?

A sample execution is shown below.

```
$ ./a.out
Please input a 8-bit bitstring: 10111100
[Parent] Input bitstring is  10111100
[Child] Received bitstring is 10111100
```

Part D: Shell with pipes

Extend the simple shell you have built in the “Shell” lab to add support for pipes. That is, your shell must be able to run commands like `cat foo.txt | grep hello` correctly, where the output from the first command is piped as input to the second command. You can start with support for two commands in a pipe, and optionally extend it to support a series of multiple commands connected by pipes. You may make the same assumptions you made in the shell lab, i.e., all the commands are simple Linux commands, and the input can be tokenized using space as the delimiter.

Your shell must fork two child processes for the two commands that will be connected via a pipe. You will need to use the `pipe` system call in the shell to create the pipe. You will also need to use the `dup2` system call to duplicate the standard input/output file descriptors of the child processes to point to the read/write ends of the pipe. Note that for correct execution of the pipe, the shell and the child processes must close all file descriptors they are not using.