Computer Science 330
Operating Systems
Siena College
Fall 2022

Lab 7: Interprocess Communication

Due: 11:59 PM, Wednesday, November 16, 2022

In this lab, you will learn about some Unix operating system mechanisms that support interprocess communication.

You must work individually on this lab.

Learning goals:

1. To learn about pipes in Unix
2. To learn about signal handling in Unix

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `ipc-lab-yourgitname`, for this lab.

Answers to written questions may be given in a PDF document committed and pushed to your repository (give the name in the `README.md` file), by writing them in a readable (reasonably nicely formatted, not all one big line of text) GitHub Markdown form in your repository's `README.md` file, or by linking to a shared document containing your answers from your `README.md` file.

Examples related to this lab are in

`https://github.com/SienaCSISOperatingSystems/sysprog-examples`

(same as in the previous lab).

Pipes

Processes may wish to send data streams (sequences of bytes) to each other. Unix *pipes* are one way to achieve this. You've almost certainly used Unix pipes at the command line. All modern Unix shells allow you to specify multiple commands on the same command line with `|` symbols in between. This indicates that these programs should be running at the same time, and that the output of a program in any (except the last) program in the pipeline should be "piped" to be the input of the subsequent program in the pipeline.

For example, the command line

```
ls | wc
```

runs the `ls` and the `wc` ("word count") programs, piping the output of `ls` to be the input of `wc`.

Question 1: What information is printed by the pipeline above, and what does it mean? (1 point)

Issue this piped command on noreaster:

```
grep ^q /usr/share/dict/words | wc -l
```

Question 2: What is the output? What does each command and parameter do and what does the answer tell us? (2 points)

There's no reason to limit this to just two commands in a pipeline. In your repository, there is a file `namelist.txt` that contains an unsorted list of names, one per line. We want to consider only those names that contain the word "dan" anywhere in the name, and we want to print out the last three alphabetically from that group. This pipelined command line would do it:

```
grep -i dan namelist.txt | sort | tail -3
```

Question 3: Explain what's happening in each component of the above command pipeline and how they combine to work as described. (3 points)

For the following lab questions, describe the effect of the given command pipeline.

Question 4: `ls -1 | wc -l` (1 point) Note: the parameter to `ls` is the number '1' while the parameter to `wc` is the letter 'l'.

Question 5: `head -10 myfile | tail -1` (1 point) The parameter to `tail` is the number '1'. We assume that the file `myfile` contains at least 10 lines.

For the following lab questions, give a single Unix command pipeline that would accomplish the task described.

Question 6: List all of the files in a directory that were last modified on Halloween. (2 points) Hint: start with `ls -la`.

Question 7: Given a file with a list of several hundred words, one per line, print the single word that occurs between lines 100 and 200 of the file which is last alphabetically. (3 points)

Using Pipes in C Programs

Turns out, you can also use pipes directly in programs. Which is a good thing, since Unix shells that support this are just C programs.

An *unnamed pipe* can be created using the `pipe(2)` system call.

Question 8: Review: what does the "2" indicate in "`pipe(2)`" above? (1 point)

The parameter (called `filedes` in the man page) is an array of two `int` values. These are file descriptors, just like the file descriptors we saw previously for file I/O using `open(2)`, `read(2)`, and `write(2)`. `filedes[0]` is the "read end" and `filedes[1]` is the "write end".

Question 9: How can you tell if the creation of an unnamed pipe with the `pipe(2)` system call was successful? (1 point)

As with the file descriptors we can obtain from the `open(2)` system call, we read and write data from a pipe file descriptor using the `read(2)` and `write(2)` system calls. Recall that these operate only on basic streams of bytes – any structure to the data is the responsibility of the programmer using the functions.

We consider the example programs in the `pipes` directory.

`pipe1.c` is an example of communication between two processes, a parent and its child created by `fork()`, communicating via an unnamed pipe.

Question 10: What is the output when you run this program? (1 point)

This required that the values of `fd` are shared between the parent and child processes. This is fine when you create your pipe just before a `fork()`, but what if we have two processes already in existence that wish to communicate through a pipe?

We can create a *named pipe* or *fifo* at the command command line with `mkfifo(1)` or in a program with `mkfifo(2)`.

`pipe2.c` augments our simple example using a named pipe.

Question 11: Run this program without creating the pipe `"testpipe"`. What happens? (1 point)

Question 12: Create the named pipe using `mkfifo(1)`. What is the output of the command `ls -l testpipe` after you do this? (1 point)

Question 13: Now run the program again with the named pipe in place. What is the output? (1 point)

`pipeprocs.c` is an example that's a little more interesting: two independent processes communicate through a pipe.

Run two instances of this program in two different windows, one to read, one to write.

Question 14: What is the output from each program? (2 points)

Question 15: Does it matter which order you create the processes? Why or why not? (2 points)

Duplicating file descriptors

We can use the `dup2(2)` system call to "reroute" input or output from one file descriptor to another file descriptor. This is how your I/O redirection and pipes will work in the shell project.

Back in the `exec` directory of the `sysprog-examples` repository, see and try `execredir.c`.

Question 16: If you run the program with a parameter `"outfile"`, what ends up in `outfile`? Why? (1 point)

Note that we don't close the file here and in fact are not given an opportunity to do so since we lose control once the `execlp` call occurs.

We have seen that you can also obtain file descriptors from `open(2)` and `pipe(2)`. The file descriptors at the ends of a pipe can be passed to `dup2(2)` as well – this will be useful in the shell – set the output of one process to be the input of another through a pipe.

Signals

We next consider a form of interprocess communication in a Unix system known as *signals*.

Question 17: Run `kill -l` (that's the letter 'l') on both a Linux and a FreeBSD system to see the list of signals supported by each. What is the output on each system? (1 point)

We can send a signal "`SIGSOMETHING`" to a process `pid` with the command

```
kill -SIGSOMETHING pid
```

For example, if we launch a program at our Unix prompt to sleep for 60 seconds and put it into the background:

```
-> sleep 60 &
```

you should see output something like:

```
[1] 96132
```

where "96132" would be the process id of the `sleep` process you just created, and `[1]` is the job number within your Unix shell of the process.

We can then send signals to that process by using its pid or `%1` which will refer to job number 1.

For example:

```
-> kill -TERM %1
```

will send the `SIGTERM` signal to try to terminate the process. If you do this, you should see output similar to:

```
[1]+  Terminated                 sleep 60
```

Now launch another `sleep 60` process in the background. Assuming this becomes shell job 1, issue these commands:

```
-> kill -STOP %1
-> kill -CONT %1
```

and wait until the `sleep` command finishes.

Question 18: What do these do, and what output do you see? (2 points)

Every process has *signal handlers* that are used to respond to signals sent to the process. Basically, it's a function that gets called *asynchronously* when a signal is received.

A *default signal handler* is installed when a process begins. `signal(2)` replaces default handler. This lets you *trap* many signals and handle them appropriately.

Be careful not to confuse this `signal()` *with the "signal" operation on semaphores! Unrelated.*

We consider the example programs in the `signals` directory.

The `sigalrm-example.c` example is compute-bound process that "wakes up" every 5 seconds to report on its progress.

The `setitimer(2)` system call is used to set a "timer" which will cause a `SIGALRM` signal to be sent to the process at some time in the future (in this case, every 5 seconds).

Question 19: What line sets up the signal handler for `SIGALRM`? (1 point)

Question 20: What function acts as the signal handler for `SIGALRM`? (1 point)

We can ignore a signal completely by setting its handler to `SIG_IGN`, and restore the default handler with `SIG_DFL`.

Consider this enhanced example: `sigalrm-example2.c`

Question 21: Which signals are handled by the signal handling function in this example? Which ones are ignored completely? (2 points)

A process can also send signals with `kill(2)`. Don't let the name fool you, this function can be used to send any signal, not just `SIGKILL`.

Question 22: One of the signal handlers in this example also sends a signal. Which handler also sends a signal and what signal does it send? (1 point)

Question 23: What happens when you send each of these signals to your running program using the `kill` command from the command line? `SIGALRM`, `SIGINT`, `SIGTERM`, `SIGSTOP`, `SIGCONT`, `SIGUSR1`, and `SIGKILL`. Try each out and paste in your output for each. (3 points)

Final note about signals: `SIGCHLD` will be useful for your shell projects. This is the signal that gets sent to a process's parent when the (child) process terminates.

Submission

Commit and push!

Grading

This assignment will be graded out of 35 points.

Feature	Value	Score
Question 1	1	
Question 2	2	
Question 3	3	
Question 4	1	
Question 5	1	
Question 6	2	
Question 7	3	
Question 8	1	
Question 9	1	
Question 10	1	
Question 11	1	
Question 12	1	
Question 13	1	
Question 14	2	
Question 15	2	
Question 16	1	
Question 17	1	
Question 18	2	
Question 19	1	
Question 20	1	
Question 21	2	
Question 22	1	
Question 23	3	
Total	35	