CprE 308 Project 2: Inter-process Communication

You have created a print server application which would take in jobs from the stdin of the application and printed to a single backend driver. This has a very clear flaw: it can only print jobs from programs piped into it at start time. This week you will modify the program so any program can send print jobs to it at any time using inter-process communication (IPC).

Please note that this is a **TWO-WEEK LAB**. The code and report are due two weeks after your scheduled lab section. This is also twice as long as a normal lab so you should work on it early and not put it off to the last minute.

Types of IPC

There are various methods of IPC in a system. At their root all IPC methods are simply a way for one process to communicate some information to another process. By this definition, what you implemented last week is IPC by piping data from the test script into the server, but that only allowed one program to talk to one other program. Now we want to make any number of processes be able to talk to the server program. All of the following example programs can be found in the ipc-types subdirectory of the Lab 6 repository.

Pipe Unnamed Pipe

One of the simplest forms of IPC is a pipe. When you pipe the output of one program into the input of another you are creating an unnamed pipe. For example, the following would create a pipe between lsmod and grep .

```
$ lsmod | grep acpi
```

Another way to create an unnamed pipe is using the pipe(2) system call in C. The pipe function will create a unidirectional pipe and return a two-element array where element 0 is the read end and element 1 is the write end of the pipe. They can be called before a fork call and then the child and parent can use it to communicate. For example, try running this code example pipe_test.c . In your report and record the output of this program along with anything you notice about the timing of when things are printed.

Learn more about pipes by reading the man pages pipe(2) and pipe(7) . In your report answer the following questions:

- What happens when more than one process tries to write to a pipe at the same time? Be specific: using the number of bytes that each might be trying to write and how that effects what happens.
- How does the output of pipe_test.c change if you move the sleep statement from the child process before the fgets of the parent?
- What is maximum size of a pipe in Linux since kernel 2.6.11?

Named Pipe (FIFO)

Another method of creating a pipe is a named pipe or FIFO using the

mkfifo command in the terminal or mkfifo(3) library call in C. A FIFO behaves exactly the same as a pipe except it has a name that any process can use to reference it. Open up two terminals to the same directory. Create a new FIFO using mkfifo

```
 $ mkfifo test_fifo
```

Now in one terminal run the following command

```
 $ cat test_fifo
```

This is now watching that FIFO and will show anything written to it. Now in your other terminal type the following command:

```
$ echo "hello fifo" > test_fifo
```

In your report answer the following questions:

- What happens when you run the echo command?
- What happens if you run the echo first and the cat ?
- Look at the man page fifo(7) . Where is the data that is sent through the FIFO stored?

Write a short program that uses named FIFO (mkfifo(3)) to print any line entered into the program on one terminal out on the other terminal (imagine a really simple, one direction, one user communication system).

Socket

Sockets are really a special type of pipe in UNIX systems. They allow for 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. For this lab you only need to worry about local communication using AF_UNIX domain. Please use man pages socket(7) , socket(2) and linked pages to answer the following questions:

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

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 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. Compile the files mq_test1.c and mq_test2.c with the -lrt flag to link with librt . 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 reads “I am X” where ‘X’ is your favorite companion.

- Change mq_test1.c to wait for and print this second message before exiting. Include the output of these programs in your report. Note: if you are unsure what we mean by companion just have it send “I am Rose”.

Shared Memory Space

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) , and linked man pages for more information. 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 vs. each by themselves. Answer the following questions

- What is the output if you run both at the same time calling shm_test1 first
- What is the output if you run both at the same time calling shm_test2 first
- What if you run each by themselves
- Why is shm_test2 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?

Convince yourself that you understand what is going on here, and if not, please ask questions. Then change the code to share some useful piece of information. Use your imagination for how this might be used. Include your new code in your write up.

Unnamed Semaphores

You used unnamed semaphores last week that were considered “thread-shared.” It is also possible to create semaphores which are stored in a shared memory space and can be shared among all processes which have mapped that memory space. In your lab report include the function call that would be needed to create an unnamed semaphore in a shared memory space called shared_mem->my_sem and assign it an intial value of 5.

Named Semaphores

Named semaphores are the same as the semaphores you used in lab last week 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 to prevent multiple processes from writing at the same time. 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).

Signals

Signals are something you have already dealt with, you just didn't know it! Simply put, a signal is a special command that a program receives from the kernel. Anytime you have typed CTRL+C to close out of a program you have been sending that program the SIGKILL signal. Another signal you have already seen is the segfault signal SIGSEGV . To find a list of signals and to learn more about them look at the man page signal(7) . These signals above have well defined actions that they perform when a program receives them (namely, quit). You can define your own signals and signal handlers and use these 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. Compile and run the program sig_test.c and answer the following questions. (note, you can still exit using CTRL+\ )

- What happens when you try to use CTRL+C to break out of the infinite loop?
- What is the signal number that CTRL+C sends?
- When a process forks, does the child still use the same signal handler?
- How about during an exec call?

Write two programs. One which will send a signal of number 42 (using kill(2)) to the other process. The other program should catch that signal a print out the message “I got the signal!”

One thing of note is that many of these above methods of IPC can generate signals. For example, when a pipe is opened with the option flag O_ASYNC it raises a signal SIGIO whenever new data arrives on the pipe. Similar signals exist for message queues and sockets. More details and examples of this can be found in the man pages.

Signals as exceptions

It is also worth mentioning that signals can be used for exception handling similar to most object-oriented languages. For example you could have a signal handler which catches a certain error and then when that error occurs use raise(3) to raise the exception. With some clever use of setjmp(3) and longjmp(3) one can achieve a simple try-catch block. This is out of scope for this lab however.

Dynamically / Statically Linked

Libraries

In the past all of the code you have probably written was contained in .c files which you compiled into object files and linked together. This

is fine when all of the files are code written for one project, but what if you want to implement the same functionality into multiple programs? Instead we can compile the functionality into a library and then link against that library. Now each time the program runs it will find those functions inside the library file. This allows several programs on the computer to all share the same library code and not duplicate the functionality that the library provides. Look in the directory library inside the git repository. You will see two C files and a header file. Let's compile lib_hello.c as a dynamically linked library. Inside the

library directory enters the following commands:

```
$ gcc -Wall -fPIC -c lib_hello.c

$ gcc -shared -Wl,-soname,libhello.so -o libhello.so lib_
hello.o

$ ls
```

You now should see the file libhello.so in the directory. The flags that were passed to gcc were

- -Wall : Enable all warnings (any small bugs in your library can cause major problems in the software that includes them)
- -fPIC : Compile as relocatable code. This is needed so the library can be loaded at runtime
- -c : Compile to object file
- -shared : Link as a shared library instead of an executable
- -Wl : Pass these commands to the linker

- -soname,libhello.so : Name the library libhello.so
- -o : The output file name

Now that you have a shared library you can compile and link against it using the following command

```
$ gcc -L./ lib_test.c -lhello -o lib_test
```

The flags are

- -L : Search a directory for shared libraries, in this case the current directory.
- -l : Include the libhello.so library. Always drop off the lib and .so from the name for the -l flag
- -o : output file name

Now try to run the program ./lib_test and record the output in your lab report. It didn't work because the program didn't know where to find the library. To tell the operating system where to search for libraries use the LD_LIBRARY_PATH environment variable. Run this command to prepend the current working directory into this path variable: Note the use of backticks instead of apostrophes around pwd.

```
$ export LD_LIBRARY_PATH=`pwd`
```

```
$ $LD_LIBRARY_PATH
```

Now run the program ./lib_test again and record the output in your lab report. Change the file lib_hello.c so that world() returns a number other than 42. Recompile the library but do not recompile lib_test . Now run lib_test again and you should see that the output has changed!

Tasks for This Lab

This is a two-week lab. There are a lot of requirements for this lab so make sure you work on it early and don't put it off to the last minute.

The rest of this lab will be extending your code from Lab4. In Lab4 you created a print server program which would accept jobs from the standard input and serve those jobs to printers. This has a clear problem: only one process can send print jobs to the print server at a time. In a real system we want to have our print server running in the background as a daemon (see section Print Server as Daemon below for more details on daemons and how to turn a process into a daemon). Anytime a program has a file it wants printed it should be able to send it to the printer using an IPC method. Additionally, programs which want to print should not need to worry about the internal workings of the print server or the IPC calls, it should be able to link in a library and use that to do printing. Please read all of the tasks before starting as they will each effect the others.

Print Server IPC

In this lab you have seen several different ways to do inter-process communication. You will now apply one or more of the above methods in your print server program. Some of the methods make more sense than others; however, most of the above methods of IPC can be applied to this problem. Here is a list of possible ways to complete this task:

A named pipe which generates a signal each time new data is written to it Print jobs packeted into messages and transmitted over a message queue

A shared memory space with a set number of print job slots and a semaphore protecting the memory A local domain socket with print jobs sent as network packets Any other combination or method you feel best solves the problem

This a very open-ended problem and there is no one right way to solve it. Some of the solutions might be more efficient, others may be easier or simpler, while others yet might be more extendable if the server was used in a real system. Use what you have learned from this lab and what you feel most comfortable with to solve this problem. Note that with some solutions you can turn the print server into a multiple writer problem. If you choose to do this, you will need to ensure that you properly protect any shared memory that all the writers will use.

Hint: Highly recommend using **Socket** to solve this problem!

Print Server Client Library

Users of your print server should not have to worry about your internal implementation of the print server when they are writing software. Instead they should be able to include a single header file and link against your pre-compiled library. Provided in the src\libprintserver directory of the git repository is a header file called print_server_client.h . Write a statically link library that provides this public API called libprintserver.so . Your library must implement all of the function prototypes listed in print_server_client.h and will do all IPC calls to the print server daemon.

Extra Credit

There are several functions listed in print_server_client.h which you can choose to implement for extra credit. You can choose which ones you would like to implement and will receive extra credit for those that you do implement. Make sure to mention in your lab report if you do choose to add any of these functions.

Print Server as Daemon

As stated in daemon(7) “a daemon is a service process that runs in the background and supervises the system or provides functionality to other processes.” Daemons are processes which run in the background and often as the root user. All interaction with daemons is done through some form of IPC calls, often times on modern systems using an IPC library and standard called DBUS. Although this lab has not talked about the DBUS due to its complexity and higher-level implementation, it is worth noting that on most modern Linux distributions most

IPC is done using the DBUS. On most modern distributions daemons are controlled using a program called systemd which allows the user to start, stop, and otherwise change the settings of running daemons. Examples of daemons on modern systems include the network manager, audio subsystem, and much more. The man page daemon(7) lists a lot of information about how modern daemons should run; however, to minimize the complexity of this lab (and because you do not have permissions to edit daemons running on the lab machines) you will be implementing a much simpler daemon system. The man page daemon(3) provides a library call which will switch a user space process to a background daemon.

A command line argument should be added to your print server program such as -d or --daemon . Whenever this flag is provided your program should switch itself to a daemon by calling daemon(3) .

Command Line Printer Program

A program called cli-printer has been provided for you. The program should take the all of the following command line arguments:

- Required
    - The file to print (will always be the last parameter passed) (optional if doing the extra credit for this section)
    - -l , --list : List all of the drivers the print server currently has installed and exit immediately.
- Optional
    - -d , --driver : The printer driver to use. If not provided provide the user with a prompt asking them to select a driver
    - -o , --output : The name of the output file. If not provided use the input file name removing the .ps from it
    - -d , --description : The description of this print job. If not provided just give the print server an empty string
    - -v , --version : Display a version string and exit

- -u , --usage : Display a usage string and exit
- -? , --help : Display a help message and exit

The program should only include the print_server_client.h header and link against the libprintserver.so by calling gcc as follows:

```
$ gcc -o print_file print_file.c -I../libprintserver/ -
L. ./libprintserver/ -lprintserver
```

Extra Credit

To take this a step further you can optionally have an interactive mode

where the user is given a prompt and can list the available drivers, select files to print, and do any other tasks you feel might be useful. If you choose to implement this make sure to mention it in your report and have a command in your program called help that will tell the user how to interact with your program.

What to submit

Make sure that you answer **ALL** of the questions asked in this lab in your report. Also talk about what method you choose for the IPC of the last part and why you made this decision. Talk about any problems you ran into and how you solved those problems. In addition, you should submit your code for last part as follows:

The print server program should be called print-server and contained in the src/print-server directory.

You will need to copy most of your Lab 4 code into this directory as well (No need, a solution print_server_single.c is given). The print server library should be called libprintserver.so and contained in the src/libprintserver directory. The command line printer program should be called cli- printer and contained in the src/cli-printer directory.

All code should be well documented using doxygen style comments.

There are a lot of requirements for this lab so make sure you work on it early and don't put it off to the last minute.

License