Lab 6 Lab Report: CprE 308

Matt McKillip

1. Introduction

This lab is introducing us to inter process communications through pipes, message queues, sockets, and shared memory spaces. These tools come in handy when implementing a library. During this lab I will use these tools to turn my Lab5 print-server into a process that can be called from other process through a library.

2. Questions

2.1 Pipe

2.1.1 Output of pipe_test.c

```
[mamckill@co2048-12 ipc-types]$ ./pipe_test
My child asked "Are you my mummy?"
And then returned 42
[mamckill@co2048-12 ipc-types]$
```

The timing of the output was as expected since the thread is waiting 2 seconds, meaning the output will be called after the parent calls the “My child asked” line.

2.1.1 What happens when more than one process tries to writeto a pipe at the same time?

Writing to a pipe needs to be atomic, so a lock should be placed on the pipe or else there will be data leaks.

2.1.1 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?

The output does not change

```
[mamckill@co2048-12 ipc-types]$ ./pipe_test
My child asked "Are you my mummy?"
And then returned 42
[mamckill@co2048-12 ipc-types]$
```

2.1.1 What is maximum size of a pipe in linux since kernel 2.6.11?

65536 bytes

2.1.2 What happens when you run the echo command?

The other terminal will output what was echoed, in this case it was “hello fifo”

2.1.2 What happens if you run the echo first and the the cat?

The terminal that calls the echo first will wait until the second terminal calls cat. Once cat is called, the second terminal prints what was echoed and then the echo command completes.

2.1.2 Where is the data that is sent through the FIFO stored?

The kernel passes all data internally without writing it to the filesystem. Thus, the FIFO special file has no contents on the filesystem; the filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem.

2.2 Socket

2.2.1 What are the six types of sockets?

1. SOCK_STREAM
2. SOCK_DGRAM
3. SOCK_SEQPACKET
4. SOCK_RAW
5. SOCK_RDM
6. SOCK_PACKET

2.2.2 What are the two domains that can be used for local communications?

1. AF_UNIX
2. AF_LOCAL

2.3 Message Queues

2.3.1 What is the output from each program

2.3.2 What happens if you start them in the opposite order

2.3.3 mq_test1.c and mq_test2.c output

2.4 Shared Memory Space

2.4.1 What is the output if you run both at the same time calling shm_test1 first?

2.4.2 What is the output if you run both at the same time calling shm_test2 first?

2.4.3 What if you run each by themselves?

2.4.4 Why is shm_test2 causing a segfault? How could this be fixed?

shared_mem->a_ptr is not declared. by adding the line shared_mem->a_ptr = shared_mem->a_string; before sleep(5);

2.4.5 What is the output if you run both at the same time calling shm_test2 first?

2.4.6 How can a shared memory space be deleted from the system?

You can delete shared memory space through an ipcrm call

2.5 Named Semaphores

2.5.1 How long do semaphores last in the kernel?

Until they are destroyed, either through a call or memory deallocation

2.5.2 What causes them to be destroyed?

memory is deallocated or sem_destroy() is called

2.5.3 What is the basic process for creating and using named semaphores? (list the functions that would need to be called, and their order).

through a sem_open() call then sem_init()

2.6 Signals

2.6.1 What happens when you try to use CTRL+C to break out of the infinite loop?

SIGKILL is called, which causes immediate program termination

2.5.2 What is the signal number that CTRL+C sends?

9

2.6.3 When a process forks, does the child still use the same signal handler?

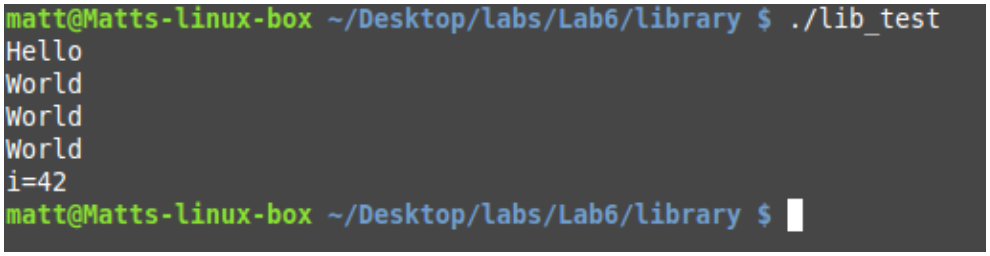
Yes

2.6.4 How about during a exec call?

The new process will have a new signal handler

Dynamically / Statically Linked Libraries

output



3. Results/Output

printer_print()

To test that the library was working properly I created a test file called lib_printserver_test.c.

```c
#include <stdio.h>
#include <stdlib.h>
#include "lib_printserver.h"

int main(int argc, char** argv)
{
        int i;
        int* handle;
        char* driver;
        char* job_name;
        char* description;
        char* data;
        driver = "pdf1";
        job_name = "test1";
        description = "this is a description";
        data = "% start of this file \n new line \n new line \n new line \n%EOF \nstuff goes here";
        i = printer_print(handle, driver, job_name, description, data);
        printf("returned: %d\n", i);

        driver = "pdf2";
        job_name = "test2";
        description = "this is a description";
        data = "% start of this file \n new line \n new line \n new line \n%EOF \nstuff goes here";
        i = printer_print(handle, driver, job_name, description, data);
        printf("returned: %d\n", i);

        return 0;
}
```

The output was as followed

Because I was just sending text data the pdf is blank, but this shows that the test class can send multiple jobs to the printer using the library.

print_file.c

To test this method I created a dummy text file called test.txt.

Calling print_file.c and passing test.txt

```
                ^
lib_printserver.c:161:3: warning: return makes pointer from integer without a cast [e
   return -1;
```

Output from print-server

```
 ^
gcc -shared -Wl,-soname,libprintserver.so -o libprintserver.so lib_printserver.o
gcc -L./ lib_printserver_test.c -lprintserver -lrt -o test
gcc -L./ print_file.c -lprintserver -lrt -o print_file
```

File

printer_list_drivers

I was unable to successfully implement this method.

What my attempt was to build a printer_driver_list during the creation of the printers so that I would be able to pass this list through the message queue if the print-server was asked. Below is my code for the print-server implementation

```
//create and install pdf_printer drivers
printer_t printers[n1+n2];
printer_driver_t printer_driver_list[n1+n2+1];
char *driver_name = "pdf1";
char *driver_version;

for (i = 0; i < n1; i++){
        asprintf(&printer_name, "pdf1-%i", i);
        printers[i] = printer_install(NULL, printer_name);

        printer_driver_list->printer_name = malloc(sizeof(printer_name));
        strcpy(printer_driver_list->printer_name, printer_name);


        printer_driver_list->driver_name = malloc(sizeof(driver_name));
        strcpy(printer_driver_list->driver_name, driver_name);

        asprintf(&driver_version, "%i", i);
        printer_driver_list->driver_version = malloc(sizeof(driver_version));
        strcpy(printer_driver_list->driver_version, driver_version);

}
driver_name = "pdf2";
for (i = n1; i < n1+n2; i++){
        asprintf(&printer_name, "pdf2-%i", i-n1);
        printers[i] = printer_install(NULL, printer_name);

        printer_driver_list->printer_name = malloc(sizeof(printer_name));
        strcpy(printer_driver_list->printer_name, printer_name);


        printer_driver_list->driver_name = malloc(sizeof(driver_name));
        strcpy(printer_driver_list->driver_name, driver_name);

        asprintf(&driver_version, "%i", i);
        printer_driver_list->driver_version = malloc(sizeof(driver_version));
        strcpy(printer_driver_list->driver_version, driver_version);
}
```

On the library side I attempted to read in the array through the buffer, calculate the size, and return the array. I think that the size of the message queue may be too small, or I am not correctly implementing the pointers.

```
// Send the data to the printer
if( mq_send(msg_queue, to_send, strlen(to_send), 10)){
        perror("mq_send");
        return -1;
}

// Wait for message back
// Initialize some buffers to receive the message
struct mq_attr attr;
attr.mq_msgsize = 1024;

printer_driver_t *buffer[5];
if(mq_getattr(msg_queue, &attr)){
        perror("mq_getattr\n");
        return -1;
}

//buffer = malloc(sizeof(buffer));
if (buffer == NULL){
        perror("malloc");
        return -1;
}

int priority;
// Receive the message
size_t size = mq_receive(msg_queue, buffer, attr.mq_msgsize, &priority);
if (size == -1){
        perror("mq_receive");
        return -1;
}

// Unlink message queue
```

4. Design Decision

To implement the IPC I chose to use message queues. I decided upon message queues because during the pre-lab I thought I had a greater understanding of them compared to the other IPC's.
I made major changes to my code from Lab5 for this lab. The first was moving the file reading to a separate function. Since the data is being passed separate from the job name, driver, and description I could not use the same algorithm as Lab5.

To handle the message queue communication with the library I kept the message queue in an infinite while loop. This allows a continuous stream of messages to come into the print-server and be handled. It also works well with the daemon process.

5. Issues

I had issues with setting up a daemon process. I am not very familiar with the concept of daemon processes since it was never gone over in class. I attempted implementation

```
  //check if daemon
  if (daemon_flag){
          // I have this commented out because I seem to get segfaults after running as a daemon
          // It should switch to a deamon process if this line is uncommented
          daemon(0,1);
f DEBUG
          printf("is daemon\n");
f
  }
```

Which would switch the process to daemon mode, but I kept getting unknown errors. So in my final code I commented out that line so I could carry on with the rest of the lab.

The other major issue I had was with the printer_list_drivers function. I was unable to successfully send a list of pointers through a message queue. I am not sure if my issues are with the implementation of the message queue - buffer size is too small. Or if my issue is with pointers.

6. Conclusion

I was able to complete the library printing and the command line printing which was the bulk of the lab. I did run into a few issues, some which I could not resolve in time to still complete this lab on time. But overall, this lab was mildly successful since I was able to get a library that printed through IPC, which seems to be the major point of the lab.

7. Suggestions

I would add more information about daemon processes. I found the topic to be very tricky, and the way it was implemented in this lab had little online examples.