Brian Reber
CprE308, Section D
Lab 04

In this lab, I learned about signals that processes can receive. I learned how to set up a signal handler. Before working on this lab, I didn't know that you could catch the Ctrl+C signal in your code and prevent it from ending the process. I also learned how processes can interact with each other using signals, and how processes can share memory using the Unix shared memory APIs.

3.1 Introduction to Signals

In this program, we set the handler for the signal SIGINT to be the function my_routine. That means that whenever our program receives the SIGINT signal, it will execute the function my_routine. Hitting Ctrl+C sends a SIGINT signal to the program, so by pressing Ctrl+C, we get a print out of "Running my_routine".

By creating a signal handler, we are "catching the signal" according to the man page for signal.

The name of the signal handler function this program uses for the SIGINT signal is my_routine.

When I comment out the signal function call, and then run the program, pressing Ctrl+C causes the program to end. SIG_DFL means that the default signal handler is to be used. Pressing Ctrl+C causes the shell to send a Keyboard Interrupt to the running program, which causes it to quit.

When I use the function call signal(SIGINT, SIG_IGN), pressing Ctrl+C doesn't do anything. That is because the SIG_IGN has the effect of ignoring the signal. Therefore, there is no indication that any signal had been sent.

When I use the function call signal(SIGQUIT, my_routine), pressing Ctrl+\ causes the printing of "Running my_routine", and pressing Ctrl+C causes the program to quit. This is because we now mapped the signal handler for the Quit signal (what gets sent when pressing Ctrl+\) to our function.

Here is the output of the program when pressing Ctrl+\ a couple times.

```
bash-3.2$ ./3.1
Entering infinite loop
Running my_routine
Running my_routine
Running my_routine
```

3.2 Signal Handlers

When running the program in this part, we can see that the two signals have different numbers. When pressing Ctrl+\, we get a signal number of 3, and when we press Ctrl+C, we get a signal number of 2. As you can tell in my output below, the first signal number occurred when I pressed Ctrl+C, and the second occurred when I pressed Ctrl+\

```
bash-3.2$ ./3.2
Entering infinite loop
The signal number is 2.
The signal number is 3.
```

3.3 Signals For Exceptions

See my source code (in the submission tar.gz) for the division by zero program.

In order for the signal handler to be called in the program, the division by zero needs to happen after we set up the signal handler. If it isn't, we won't be able to catch the exception because it isn't set up by the time the exception occurs.

3.4 Signals using alarm()

There are two parameters to this program. One is the message you want to display, and the other is the number of seconds you want to delay the message. It should be run like this:

```
./3.4 "Hello World!" 2
```

This will print "Hello World!" two seconds after starting the program.

The alarm() function sets a timer that will send a signal (SIGALRM) after a certain number of seconds. We set the signal handler to our own function that prints out the message, so when the signal SIGALRM is sent, our function is called, therefore printing out the message. The signal is delayed a certain number of seconds after the alarm function is called.

```
bash-3.2$ ./3.4 "test msg" 3
Entering infinite loop
test msg
```

3.5 Pipes

There are two processes that run during the execution of this program. We fork once, so there is the parent (original process), and a child process. The parent thread is the one that writes the message, as the result from fork is greater than zero for the parent, causing it to enter the if block. The child process is the one that sleeps and then reads, and prints (the else block).

We first create the msg array, and statically define the value of this array to be "How are you?". We then write the message to the pipe we created, which then gets read from the pipe into the variable inbuff. From there, we print the value of inbuff to the screen.

The way this currently works is that there is a sleep statement in the child process, so that it can be sure that the parent has written to the pipe. If the sleep statement wasn't there, we couldn't guarantee that the pipe had been written to because we could try to read from the pipe before the other process had the chance to write to it. An alternate way of doing this would be to have the child process write the message, and the parent would use a waitpid on the child. This way, we could print, and then make sure the child had completed before the parent reads from

the pipe. This would probably be the ideal solution for this example, as all the child is doing is writing to the pipe and then exiting. This may not be the best solution if the child is executing a bunch of other code, and we can therefore not wait for it to complete before allowing the parent to complete. (See 3.5.wait.c in my tar.gz)

3.6 Signals and Forks

Here is the output from the program, with me hitting Ctrl+C a couple times:

```
bash-3.2$ ./3.6
Entering infinite loop
Entering infinite loop
Return value from fork = 5964
Return value from fork = 0
Quit
```

As you can tell, there are two processes running. We forked once, so there are two processes. We set up a signal handler on both of them, so whenever the signal SIGINT is sent, we will enter the my_routine function in both processes.

The parent (original process) is the one that prints out “Return value from fork = 5964”, because it has the value of ret to be greater than zero. Since ret is the return value from the fork function call, we know that if it is greater than zero, it is in the parent process. The child process is the one that prints out “Return value from fork = 0”. We know this for the same reason that we figured out the parent’s result. Since it is the child of a forked process, we know that the return value from fork is 0.

Both the child and the parent processes received the signals, so therefore there would be two processes that received the signal. Since the shell sent the signal to the parent process, the parent must have forwarded that signal on to its children as well.

3.7 Shared Memory Example

Here is the output of the programs, when starting the server in the background, and then starting the client in the foreground.

```
bash-3.2$ ./shm_client
Message read: abcdefghijklmnopqrstuvwxyz
Client done reading memory
Server detected client read
```

The two processes locate the same memory by calling the shmget function with the same key value.

One flaw with these programs is that they are accessing the same memory area at the same time. While one is writing, the other could also be writing. This could cause corruption of the data located in the shared memory. This is similar to the issue we faced when dealing with threads accessing and writing to the same variable at the same time. It could cause undesirable results.

When I run the client without the server, I get this output:

```
bash-3.2$ ./shm_client
Message read: *bcdefghijklmnopqrstuvwxyz
Client done reading memory
```

From looking at the code of the server and client, this appears to be what is left in shared memory when the programs terminate. The fact that we are locating the same location in memory in sequential runs of this program seems to mean that this shared memory isn't cleared immediately (at least not the way the programs are written right now).

When I add the code to the server, run the server and the client, and then the client without the server, this is the output I get.

```
bash-3.2$ ./shm_client
shmget failed: No such file or directory
```

So from that output, it appears that the shared memory is considered a file stored somewhere where the shmget can locate it by key. When we call shmctl, with IPC_RMID as a parameter, we are marking the area of memory available to be destroyed. That is why the client is unable to locate it after the server marks it as destroyed.

3.8 Message Queues and Semaphores

The maximum size of a message in the message queue is MSGMAX, which is 8192 bytes by default. The maximum size of the message queue is specified by MSGMNB, which is 16384 bytes by default.

If there are no messages available, the calling process is blocked until one of the following conditions occurs:

- A message of the desired type is placed in the queue.
- The message queue is removed from the system. In this case the system call fails with errno set to EIDRM.
- The calling process catches a signal. In this case the system call fails with errno set to EINTR.

The message queue can be marked for destruction at any time, through the call of the msgctl function with IPC_RMID as one of the arguments.

The semaphores provided by the semget/semctl/semop functions are general semaphores. They can be set to any value, and are not limited to 0/1. In order to use a semaphore, you need to use semget. Then to use the semaphore, you would need to use semop with the correct semaphore id, and the operation you want to do.