Name: Sean Gordon

Section: 3

University ID: Sgordon4

Lab 4 Report

Summary (10pts):

This lab covered various small but important functions and concepts for writing code for linux systems. The exercises focused specifically on common and useful library functions with clear, concise explanations and guidance.

I personally had never thought to capture Ctrl-c input to do something else, and it is causing me to reevaluate my toolbox of signal uses. I am also thankful for a lab that doesn't beat around the bush and tells you what you need to know efficiently. This class so far has been a breath of fresh air in the smog that is the engineering department.

Lab Questions (80 pts):

3.1 - Introduction to Signals

3pts After reading through the man page on signals and studying the code, what happens in this program when you type CTRL-C?

The program captures the signal and calls my_routine() rather than allowing it to propagate.

3pts What is the signal handler function for this code?

my_routine()

3pts Remove the signal(...) statement in main. Recompile and rerun the program. Type CTRL-C. Why did the program terminate? Hint: find out how a program usually reacts to receiving what CTRL-C sends (man 7 signal).

The program did not capture the signal and allowed it to propagate to its usual destination, thus terminating the program.

3pts Replace the signal( ) statement with *signal(SIGINT, SIG_IGN)*. Run the program and press CTRL-C. Look up SIG_IGN in the man pages. What's happening now? (HINT: Look up SIG_IGN in man signal)

The signal is being ignored completely, not triggering any function.

3pts The signal sent when CTRL-\ is pressed is SIGQUIT. Replace the *signal()* statement with *signal(SIGQUIT, my_routine)* and run the program. Press CTRL-\. Why can't you kill the process with CTRL-\ now?

The program is capturing the SIGQUIT signal now (spawned from CTRL-\).

3.2 – Signal Handlers

5pts What are the integer values of the two signals? What causes each signal to be sent?

SIGINT (Ctrl-c) – 2
SIGQUIT (Ctrl-\) – 3

3.3 – Signals for Exceptions

10pts Include your source code

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void my_routine();

int main() {
   //Capture any SIGFPEs received
   signal(SIGFPE, my_routine);

   //Perform div-by-zero
   printf("Performing division by zero: \n");
   int a = 4, b = 0;
   a = a/b;

   printf("Will never be reached\n");
   return 0;
}
/* will be called asynchronously, even during a sleep */
void my_routine() {
   printf("Caught a SIGFPE\n");
   exit(0);
}
```

5pts Which of the following statements should come first to trigger your signal handler? The signal() statement, or the division-by-zero statements. Explain why.

The signal() statement must come before the div-by-zero statements as handlers must be registered to their respective signals before the signals are received or the signal will propagate as normal, not calling the handler.

3.4 – Signals using alarm()

4pts What are the input parameters to this program, and how do they affect the program?

argv[0] = process name
argv[1] = time to wait
argv[2] = message to be printed

6pts What does the statement *alarm(time)* do? Mention how signals are involved.

Alarm arranges for a SIGALRM signal to be delivered to the calling process after *time* seconds.

3.5 – Signals and fork

2pts How many processes are running?

2 processes

2pts Identify which process prints out which message.

Main prints:
Entering infinite loop
Return value from fork = ####

Child prints:
Entering infinite loop
Return value from fork = 0

2pts How many processes received signals?

2 processes

3.6 - Pipes

2pts How many processes are running? Which is which (refer to the if/else block)?

There are two processes, with the child writing and the main reading.

6pts Trace the steps the message takes before printing to the screen, from the array *msg* to the array *inbuff*, and identify which process is doing each step.

Message is created
Buffer is created with the size of the message
Child writes the message to the pipe
Main reads the message from the pipe into *inbuff*
Main prints the contents of *inbuff,* and therefore the message

2pts Why is there a sleep statement? What would be a better statement to use instead of sleep?

The sleep statement pauses the main process for long enough for the child to finish working. A better statement to use would be wait().

3.7 – Shared Memory

2pts How do the separate processes locate the same memory space?

The processes use the defined key to tell which memory spaces to use.

3pts There is a major flaw in these programs, what is it? (Hint: Think about the concerns we had with threads)

The server never deletes the message from memory, but the message is changed after the first client reads it, leaving it corrupted for any client that comes after.

3pts Now run the client without the server. What do you observe? Why?

The client reads the corrupted message without knowing anything is wrong because the server never removed it from memory.

3pts Recompile the server. First run the server and client together. Then run the client without the server. What do you observe? What did the two added lines do? (HINT: look at the man pages of shmdt and shmctl; look for the explanation of IPC_RMID in the shmctl man pages)

The client now cannot find the message in memory. The two added lines detached the segment from the server's memory and marked it for removal.

3.8 – Message Queues and Semaphores

2pts Read the man pages of *msgsnd*. Message queues allow for programs to synchronize their operations as well as transfer data. How much data can be sent in a single message using this mechanism?

The default size of a message queue is defined by MSGMNB – 16384 bytes
After initialization the limit can be modified using msgctl()

2pts Read the man pages of *msgrcv*. What will happen to a process that tries to read from a message queue that has no messages? (hint: there are more than one cases)

If IPC_NOWAIT is not specified, the calling process is blocked, otherwise the call fails with ENOMSG.

2pts Both Message Queues and Shared Memory create semi-permanent structures that are owned by the operating system (not owned by an individual process). Although not permanent like a file, they can remain on the system after a process exits. Describe how and when these structures can be destroyed.

Shared Memory will be destroyed after it is marked with IPC_RMID and the last process using it detaches, and Message Queues can be destroyed using msgctl(). Both processes can also be removed using ipcrm, with the command ipcs used as a counterpart.

2pts Are the semaphores in Linux general or binary? Describe in brief how to acquire and initialize them.

Linux provides both general/counting and binary semaphores.

```
#include <semaphore.h>
sem_t semaphore;
sem_init(…);
```

This semaphore can be initialized with a count of 1 to create a binary semaphore (mutex), or with a larger value to create a counting/general semaphore.