

Name: Nihaal K Zaheer
Section: 2
University ID: 105254206

Lab 4 Report

Summary (10pts):

This lab involves learning about Linux signals and inter-process communication mechanisms through various exercises. The lab provides an overview of signals and IPC mechanisms in the manual pages, and then moves on to various exercises.

In Exercise 3.1, students learn about signals by creating a program and observing the behavior when CTRL-C or CTRL-\ is pressed. In Exercise 3.2, students learn about signal handlers by passing the number of the signal to the signal handler. In Exercise 3.3, students learn to handle exceptions using signals by creating a program that handles the division-by-zero exception using signals. Finally, in Exercise 3.4, students learn to use signals with the alarm() function.

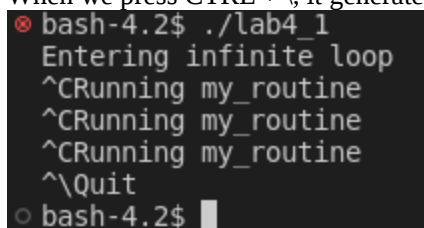
The lab encourages students to alter the programs to run their own experiments and to figure out what is happening in the programs by observing the behavior or by placing well-placed printf statements.

Lab Questions:

3.1:

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

On reading the man pages of signal and SIGINT, I understood that SIGINT acts as an interrupt from keyboard.
When we press CTRL + C, the signal function is called, it then prints "Running my routine", and then returns back to the infinite while loop.
When we press CTRL + \, it generates a SIGQUIT signal which terminates the program.



```
bash-4.2$ ./lab4_1
Entering infinite loop
^CRunning my_routine
^CRunning my_routine
^CRunning my_routine
^\\Quit
bash-4.2$
```

3 pts Omit the signal(...) statement in main. Recompile and run the program. Type CTRL-C. Why did the program terminate?

```
⊗ bash-4.2$ ./lab4_1
  Entering infinite loop
  ^C
  ○ bash-4.2$ █
```

CTRL +C still sends a keyboard interrupt, which in this case is 'SIGINT'. But since we don't have a registered signal handler, it takes the default action, which is to terminate immediately.

3 pts In main, replace the `signal()` statement with `signal(SIGINT, SIG_IGN)`. Recompile, and run the program then type CTRL-C. What's happening now?

```
⊗ bash-4.2$ ./lab4_1
  Entering infinite loop

  ^C
  ^\Quit
```

SIG_IGN ignores the signal that it receives. So when CTRL + C is pressed, it triggers the SIGINT interrupt, which is assigned to do SIG_IGN which is ignore the signal. So this does nothing.

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

```

❌ bash-4.2$ ./lab4_1
Entering infinite loop
^Running my_routine
^Running my_routine
^Running my_routine
^Running my_routine
^Running my_routine
^C

```

The default signal when CTRL + \ is pressed is SIGQUIT. On adding `signal(SIGQUIT, my_routine)`, we assign an interrupt so that when the SIGQUIT signal is called in the foreground, it would go to the function `my_routine()`. This is why we can't kill the process with CTRL + \. But the SIGINT signal interrupt is not assigned to anything, so by default it will terminate the program. But if we gave a function for SIGINT too, we would not be able to terminate.

```

❌ bash-4.2$ ./lab4_1
Entering infinite loop
^C^Running my_routine
^Running my_routine
^Running my_routine
^C^Running my_routine
^Running my_routine
^C^Running my_routine

```

```

{
    // signal(SIGINT, my_routine);
    signal(SIGINT, SIG_IGN);
    signal(SIGQUIT, my_routine);
    printf("Entering infinite loop\n");
    while (1)

```

3.2:

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

```

bash-4.2$ ./lab4_2
Entering infinite loop
^CThe signal number is 2.
^CThe signal number is 2.
^The signal number is 3.
^The signal number is 3.
^

```

SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard

The value of SIGINT and SIGQUIT are defined as 2 and 3 respectively by the OS. So when each of these signals are called by interrupts, 'signo' gets assigned these values.

3.3:

10 pts Include your source code

```
lab4_3.c  X
lab4 > lab_code > C lab4_3.c > catch_zero()
1  #include <signal.h>
2  // #include <stdio.h>
3  void catch_zero();
4  int main()
5  {
6      signal(SIGFPE, catch_zero);
7      int a = 4;
8      a = a/0;
9      printf("Result: %d", a);
10     return 0;
11 }
12
13 void catch_zero()
14 {
15     printf("Caught a SIGFPE \n");
16     exit(1);
17 }
bash-4.2$ ./lab4_3
Caught a SIGFPE
```

5pts Explain which line should come first to trigger your signal handler: the signal() statement or the division-by-zero statement? Explain why.

The signal statement must come first.
If we call the signal after performing the zero error, the signal might not be raised.
So signal() is called before to ensure the handler will take care of the zero error if it occurs in the future.

3.4:

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

```
bash-4.2$ ./lab4_4 printed_after_two 2
Entering infinite loop
printed_after_two
```

We need two input parameters, one text followed by a number which is time in seconds to wait. In the above screenshot, we print, “printed_after_two” after 2 seconds.

6pts What does the function “alarm” do?? Mention how signals are involved.

The alarm() function takes a parameter which sets a timer for the number of seconds provided. Once the time runs out, the SIGALRM signal is called, this in turn called the my_alarm() function, which prints the msg, which in our case is the first argument provided.

3.5:

2pts Include the output from the program.

```
bash-4.2$ ./lab4_5
Entering infinite loop
Entering infinite loop
^CReturn value from fork = 0
Return value from fork = 7216
```

2pt How many processes are running?

2 process are running, the parent and the child.

```
bash-4.2$ ps
  PID TTY          TIME CMD
 7215 pts/3        00:00:00 lab4_5
 7216 pts/3        00:00:00 lab4_5
 7374 pts/3        00:00:00 ps
30985 pts/3        00:00:00 bash
```

3pts Identify which process sent each message.

Since we fork the process, the parent and the child process will print outputs. This is why we have 2 “Entering infinite loop ” statements. This is then followed by printing ‘ret’ which is the value of fork(). The first one printed is the child process since value is shown as 0. The second statement is printed once the child process finishes executing, this is the parent process.

3pts How many processes received signals?

Both the child and parents process receives the signal. But the execution happens only one at a time. In this case, first the child, then the parent.

3.6:

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

There are 2 processes running, a parent and a child. The 'if' block references the parent, because the value of fork greater than 0 is for a parent. The 'else' statement references the child process.

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.

First we initialize the pipe. Then the write() function takes the message stored in the variable msg and writes it to the write end of the pipe (p[1]). Then in the child process the process sleeps for a second which lets the parent process write correctly. Then the read function is called. We use p[0] to retrieve the message from the pipe and store this to the variable 'inbuff'. The writing to inbuff is done by the read() function.

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

The sleep() function is used so that the parent process has enough time to write the message before the child process starts reading. Referencing lab2, we might be able to use a wait() or waitpid() function instead of sleep, but this might not be the case because wait() is used to complete a child process, but in this case we want the parent process to terminate first. Another thing we could use instead of sleep would be mutexes or semaphores.

3.7:

3pts How do the separate processes locate the same memory space?

This is done by the key. Here, the key is generated using the key_t data type. In this case, the key is given a value of 5678. The shmget function then reads the key and creates a shared memory segment for this specific key. Since both the client and server use the same key. It uses the same shared memory address space.

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

The flaw in the program is that, it does not account for any mechanism for synchronization or anything that can account for race conditions. In a case where the client reads before the server finishes running, the client could print garbage value. To fix this we could use mutexes or semaphores.

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

```
bash-4.2$ ./shm_client
Message read: *bcdefghijklmnopqrstuvwxyz
Client done reading memory
```

Once the client is done reading from the server, it prints everything it reads, that is alphabets from a to z. After this it sets **shm** = '*'. This tells the server that the client has received all the data and that the server can terminate. But on doing so, it replaces the first element in the string. This shows us that the server is not running and that client is reading old data.

6pts Now add the following two lines to the server program just before the exit at the end of main:

```
shmdt(shm)
```

```
shmctl(shmid, IPC_RMID, 0)
```

Recompile the server. Run the server and client together again. Now run the client without the server. What do you observe? What did the two added lines do?

```
bash-4.2$ ./shm_client
shmget failed: No such file or directory
```

Shmdt() detaches the shared memory segment. The shmctl() function with IPC_RMID, marks segment to be destroyed and marked for deletion. When the last process detaches from the shared memory, the segment detaches, which is why we get the error. The last two lines ensure that any resource associated with it, is properly released.

3.8:

2pts 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 amount of data that can be sent in a single message using message queues depends on the system implementation and the size of the message buffer allocated when creating the message queue. In most systems, the default size of the message buffer is relatively small, typically around 8KB. The maximum message size can be increased by setting the appropriate system limits or by using a custom message queue implementation.

2 pts What will happen to a process that tries to read from a message queue that has no messages (hint: there is more than one possibility)?

It depends on how the process is implemented. But there can be the following possibilities;

1. Blocking read: If the process is using a blocking read operation, it will be suspended until a message arrives on the queue.
2. Non-blocking read: If the process is using a non-blocking read operation, it will return immediately with an error code indicating that there are no messages in the queue.
3. Interruptible read: If the process is using an interruptible read operation, such as a system call that can be interrupted by a signal, it will be interrupted by the signal and the read operation will return an error code.

3pt 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.

Queues and Shared memories are created using unique keys. They can be removed in several ways;

Explicit removal: A process can explicitly remove a Message Queue or Shared Memory segment using the `msgctl` or `shmctl` functions, respectively.

System reboot: If the system is rebooted, any Message Queues or Shared Memory segments that were not explicitly removed will be destroyed.

Manual removal: In some cases, an administrator may manually remove a Message Queue or Shared Memory segment using system tools or commands.

3pt Are the semaphores in Linux general or binary? Describe in brief how to acquire and initialize them.

Then can be either.

Binary semaphores only have two conditions, 0 or 1.

General semaphores can have any non-negative integer value. It can be used to represent a count of available resources.

To acquire and initialize, we can;

Call `semget()` to obtain a semaphore set identifier (`semid`).

Call `semctl()` to initialize the semaphore set with `SETVAL` command, which sets the value of the semaphore to a specified value.

Use `semop()` to perform semaphore operations on the semaphore set. Once the semaphore is no longer needed, it can be released using `semctl()` with the `IPC_RMID` command.