CprE 308 Laboratory 4: Inter-Process Communication

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

Fill out the provided lab report template and submit it as a PDF on Canvas. Feel free to adjust the answer boxes if needed. For questions that ask to submit source code, kindly **paste the code in the report or upload the source files** to Canvas. This makes our grading easier.

- 10 pts - A summary of what you learned in the lab session. This should be no more than two paragraphs. Try to get at the main idea of the exercises, and include any particular details you found interesting or any problems you encountered.
- 90 pts - A write-up of each experiment in the lab (as dictated by the report template). For output, take screenshots or copy-paste results from the terminal, and summarize when asked. For program code, include comments that explain the important steps within the program. Include all relevant details.

2 Introduction

In this lab session, you will learn about Linux signals through exercises 3.1 - 3.5 and inter-process communication (IPC) mechanisms through exercises 3.6 - 3.8. Before you begin the exercises, read the overview of signals and IPC mechanisms in the manual pages:

- `man 7 signal`
- `man svipc`

Execute the C programs given in the following problems. Observe and interpret the results. You will learn about some of the different methods of communication between Unix processes through these experiments. You are encouraged to alter the programs to run your own experiments or to figure out what's going on. Sometimes a well placed printf will tell you a lot about the program.

3 Excercises

3.1 Introduction to Signals

Read the overview of signals (`man 7 signal`) and the description of the `signal()` call (`man signal`). Create a program with the below code.

```
#include <signal.h>
void my_routine( );
int main( ) {
    signal(SIGINT, my_routine);
    printf("Entering infinite loop\n");
    while(1) {
        sleep(10);
    } /* take an infinite number of naps */
    printf("Can't get here\n");
}

/* will be called asynchronously, even during a sleep */
void my_routine( ) {
    printf("Running my_routine\n");
}
```

Compile and execute the program. Press CTRL-C a few times and then press CTRL-\.

(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?

(3 pts) Omit the `signal(...)` statement in `main`. Recompile and run 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`).)

(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? (Hint: Look up `SIG_IGN` in the man pages)

(3 pts) 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? (You should be able to terminate the process with `CTRL-C`.)

3.2 Signal Handlers

When a signal is received, the number of the signal can also be passed to the signal handler. Run the following program and type `CTRL-C` and `CTRL-\`.

```
#include <signal.h>
void my_routine( );
int main() {
    signal(SIGINT, my_routine);
    signal(SIGQUIT, my_routine);
    printf("Entering infinite loop\n");
    while(1) { sleep(10); }
    printf("Can't get here\n");
}
void my_routine(int signo) {
    printf("The signal number is %d.\n", signo);
}
```

Compile and run the program. Press CTRL-C and CTRL-\few times. You can kill the process using the `kill <pid>` command in another terminal window (look up the process id with `ps`).

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

3.3 Signals for Exceptions

Signals can be used to handle exceptions. In this exercise, you will create a program that handles the division-by-zero exception using signals. The signal sent for division by zero is SIGFPE. Write a main program and a signal handling routine that prints the message “Caught a SIGFPE” The program should create a division-by-zero scenario like the code below:

```
/* the division needs to use a variable or gcc will not compile the instruction */
int a = 4;
a = a/0;
```

Add an exit statement at the end of the signal handler so the program will terminate. Note that you may get a warning about division by zero from the compiler.

(10 pts) Include your source code

(5 pts) Which of the following statements should come first to trigger your signal handler? Explain why?

- the `signal()` statement
- the division-by-zero statement

3.4 Signals using alarm()

Read the following code and identify what this program does. Be sure you understand the previous experiments before running this one. You can look up the function calls `alarm`, `atoi` and `strcpy` in the man pages.

```
#include <signal.h>
#include <stdio.h>
char msg[100];
void my_alarm();
int main(int argc, char * argv[]){
    int time;
    if (argc < 3) {
        printf("not enough parameters\n");
        exit(1);
    }
    time = atoi(argv[2]);
    strcpy(msg, argv[1]);
    signal(SIGALRM, my_alarm);
    alarm(time);
    printf("Entering infinite loop\n");
    while (1) { sleep(10); }
    printf("Can’t get here\n");
}

void my_alarm() {
    printf("%s\n", msg);
    exit(0);
}
```

Note that the variables argc and argv are passed to main. The variable argc is the argument count, or the number of strings in argv. The variable argv is an array of strings filled in from the command line. The first string is always the program name, and subsequent strings are space-delimited arguments that you type in after the command.
For example:

```
$./program argument
```

will result in `argc=2, argv[0]="./program"` and `argv[1]="argument"`.

In your report, answer the following questions:

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

(6 pts) What does the function `alarm()` do? Mention how signals are involved.

3.5 Signals and fork

Observe and explain the behavior of the following program.

```
#include <signal.h>
void my_routine( );
int ret;
int main() {
    ret = fork( );
    signal(SIGINT, my_routine);
    printf("Entering infinite loop\n");
    while(1) { sleep(10); }
    printf("Can't get here\n");
}

void my_routine() {
    printf("Return value from fork = %d\n", ret);
}
```

Run the program and trigger the `SIGINT` signal.

In the report,

(2 pts) Include the output from the program

(2 pts) How many processes are running?

(3 pts) Identify which process sent each message

(3 pts) How many processes received signals?

3.6 Pipes

A Unix pipe is a one-way communication channel. A pipe is used to pass a character stream from one process to another. On the command line, you've already seen a pipe:

```
$ ps -el | less
```

will take output from ps and pipe it to less so you can view it.

Pipes are declared using the pipe function. The pipe function takes an array of two integers and returns two file descriptors: the read end will be `p[0]`, and `p[1]` will be the write end. A file descriptor is a tag that describes a resource to the operating system. You pass the file descriptor to the operating system on read and write calls so the operating system knows which resource you're reading from or writing to. There are limits to how many file descriptors can be open in any process, and because they are system resources, there is also a system limit on the total (i.e., how many pipes can be there one time).

Observe the output of this program and explain.

```
#include <stdio.h>
#define MSGSIZE 16
int main() {
    char *msg = "How are you?";
    char inbuff[MSGSIZE];
    int p[2];
    int ret;
    pipe(p);
    ret = fork( );
    if (ret > 0) {
        write(p[1], msg, MSGSIZE);
    } else {
        sleep(1);
        read(p[0], inbuff, MSGSIZE);
        printf("%s\n", inbuff);
    }
    exit(0);
}
```

Include answers to the following questions in your report:

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

(6 pts) 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.

(2 pts) Why is there a `sleep` statement? What would be a better statement to use instead of `sleep` (Refer to lab 2)? (Hint: You might have to reverse the role of parent and child.)

3.7 Shared Memory Example

In class we have talked extensively about issues that occur when multiple processes try to access the same resource, such as the same variable. However, we have also emphasized that each process has its own address space. How can they share a variable? It turns out that Unix allows processes to share segments of memory through system calls.

Download the example programs `shm_server.c` and `shm_client.c` from Canvas. Read the man pages for the following commands: `shmget, shmat, shmdt`. Compile `shm_server.c` and `shm_client.c`. Run the programs, *starting the server before the client.*

Include answers to the following questions in your report:

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

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

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

(6 pts) 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? (Hint: look in to the man pages of `shmctl`, look for the explanation of `IPC_RMID`)

3.8 Message Queues and Semaphores

This section of the lab does not require any experiments. This section simply explores some of the other IPC mechanisms available to a programmer. For more in depth exploration of IPC mechanisms please see "Inter-process Communications in UNIX: The Nooks and Crannies" (2nd Edition), by John Gray.

Read man pages for the following commands to answer questions in this section.

- IPC Overview - `svipc`
- Semaphores – `semget`, `semctl`, `semop`
- Message Queues – `msgget`, `msgctl`, `msgsnd`, `msgrcv`

Include answers to the following questions in your report:

(2 pts) 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?

(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)?

(3 pts) 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.

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