CprE 308 Laboratory 4: Inter-Process Communication

Department of Electrical and Computer Engineering
Iowa State University

Spring 2014

1 Submission

Please work through all the experiments in the lab and prepare a write-up of each experiment. Submit your report through Blackboard. Include the following in your lab report:

- Your full name and lab section in the upper right corner of the first page in large print.
- A cohesive summary of what you learned through the various experiments performed in this lab. 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.
- A write-up of each experiment in the lab. Each experiment has a list of items you need to include. For output, cut and paste results from your terminal and summarize when necessary. For explanations, keep your answers concise, but include all relevant details.

2 Resources

Some of the Unix calls used in the following problems are: signal, exit, alarm, pipe, read, and write. Check out the links on the course web page for further information on Inter Process Communication in Unix. Some helpful man pages:

- `man signal`
- `man 7 signal`
- `man 2 read`
- `man 2 write`

Also, some of the functions used in Lab 2 are used in this lab, so review them if you need to.

3 Excercises

Execute the C programs given in the following problems. Observe and interpret the results. You will learn about some of the different ways of communication between Unix processes by performing the suggested 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.

3.1 Introduction to Signals

The objective of this experiment is to learn about signals. Execute the following program and do the exercises below.

```
#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");
}
```

Press CTRL-C a few times.

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

(1 pt) According to the man page, what is it called when a signal handler function is used?

(1 pt) What is name of the signal handler function for this code?
When you're done, you can kill the process with `CTRL-\`.

(2 pts) Omit the signal( ) statement in main and run the program. Type `CTRL-C`. Look up `SIG_DFL` in the man pages (man signal). Why did the program terminate? Hint: find out how a program usually reacts to receiving what `CTRL-C` sends (man 7 signal).

(2 pts) In main, replace the signal( ) statement with `signal(SIGINT, SIG_IGN)`. Run the program and type CTRL-C. Look up `SIG_IGN` in the man pages. What's happening now?

(2 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`.

In your report, provide any output or lack thereof, and answer all of the questions above.

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);
}
```

(4 pts) In your report, note what causes each signal to be sent and what the number of the signal is. You can kill the process using the kill command without any options (look up the process id with ps in a new terminal window).

3.3 Signals for Exceptions

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" for a floating point error (e.g. division by zero). Include a division by zero in the main program like the one below:

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

If you don't see your message, then you didn't set up the signal handler correctly.
Hint: Add an exit statement at the end of the signal handler so the program will terminate.
In the report, please

(5 pts) include your source code

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

3.4 Signals using alarm()

Figure out what this program does and explain it. Be sure you understand the previous experiments before running this one.

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".
You can look up the functions 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);
}
```

In your report, answer the following questions:

4 pts What are the 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 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? Think about what happens if one process runs first. Think back to lab 2 – what would be a better statement to use instead of sleep for this small example? You might have to reverse the role of parent and child.

3.6 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);
}
```

In the report,

1 pt Include the output from the program

1 pt How many processes are running?

2 pts Identify which process sent each message.

2 pts How many processes received signals?

3.7 Shared Memory Example

Read man pages for the following commands.

- Shared Memory – shmget, shmctl, shmat, shmdt

Review the source code for `shm_server.c` and `shm_client.c`

Compile and run `shm_server.c` and `shm_client.c`

Be sure to start the server program prior to the client.

Include answers to the following questions in your report:

(1 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?

(3 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?

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.

- 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?

(1 pt) 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)?

(1 pt) 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.

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