CIS 452 Lab: Week #3
InterProcess Communication (IPC)

Overview

The purpose of this assignment is to become familiar with some fundamental methods of communication between processes (IPC). Every operating system provides multiple mechanisms for communicating between processes and/or threads; this lab specifically explores the use of signals and pipes. In order to better understand pipes, it also provides an introduction (or refresher) to basic UNIX file operations.

Hand-in:

- A word document containing the answer to the numbered questions.
- Any requested screenshots (uploaded as separate files)
- Program source code (no zip files)

Signals

Signals represent a simple, one-directional form of communication between processes. There are a number of predefined signals that exhibit a default behavior (look in `/usr/include/signal.h` and particularly in `/usr/include/bits/signum.h` for details)(Specific folders may vary from system to system) . For certain signals, it is possible to override the default behavior by installing a signal handler, essentially a function that defines what to do when the associated signal is received. The signal handler is installed (and registered) via the **`signal()`** system call.

Signals can originate with the user, with the operating system, or with another process. They can be sent from one process to another using the **`kill()`** system call - a mechanism known as raising a signal. (Note: it is called "kill" for historical reasons - it is basically a "send".)

The following sample program illustrates the installation of a signal handler that overrides the built-in interrupt signal (Control-C). This technique allows interrupts to be used, for example, to gracefully shut down a long-running process such as a daemon: by closing all

files, pipes and sockets, and terminating all child processes before exiting. Refer to the man pages to understand the use of the `signal()` system call and the meaning of its arguments. Note that this section of the lab refers to the simpler ANSI **`signal`** mechanism; modern signaling often uses the POSIX **`sigaction`** mechanism.

sampleProgramOne

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

void sigHandler (int);

int main()
{
    signal (SIGINT, sigHandler);
    printf ("waiting...\n");
    pause();
    return 0;
}

void
sigHandler (int sigNum)
{
    printf (" received an interrupt.\n");
    // this is where shutdown code would be inserted
    sleep (1);
    printf ("time to exit\n");
    exit(0);
}
```

Perform the following operations and answer the questions:

Study, compile and run sampleProgramOne. Send the program an interrupt via the keyboard.

1. What does the program print, and in what order?
2. Describe exactly what is happening to produce the answer observed for the above question.

Reliable Signals

The original ANSI signal() mechanism had a sporadic flaw called a "race condition", basically a concurrency problem (more on that in future labs). Newer

versions of Linux have fixed this problem. However, this means that code using **`signal()`** will work correctly on newer systems, but occasionally have issues if run on older systems; hence it is referred to as unreliable.

The POSIX **`sigaction()`** mechanism is implemented uniformly (i.e., reliably) across all versions (new and old).

ANSI signals are generally easier to learn and work with, which is why they are introduced in this lab. However, be aware that the POSIX **`sigaction()`** mechanism is generally preferred over the simpler ANSI **`signal()`** functions.

File I/O

This section describes some of the basics of UNIX file I/O - it is a prelude to understanding interprocess communication via pipes (see any UNIX reference book for more details). UNIX promotes device-independent input and output by the use of *handles*. This means that handles are used for many different types of I/O (e.g., files, pipes, sockets, physical devices). A handle is also known as a file descriptor, which is basically an integer index into a table of entries, each of which is associated with a file or device. Each process is initially given the file descriptors 0, 1, 2 (accessible via the **`fileno()`** function call or the macros STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO). These file descriptors correspond to standard input (keyboard), standard output (display), and standard error (unbuffered display). The file descriptor table for a process initially looks like this:

[0]	standard input
[1]	standard output
[2]	standard error

UNIX also permits file redirection, by modifying an entry in the file descriptor table. Suppose the process that owns the above table is executing the "ls" utility, and its output is redirected to a file as in: "`ls > directory.txt`". Then its file descriptor table would now appear as:

[0]	standard input

[1]	WRITE directory.txt
[2]	standard error

indicating that the normal stdout (to the display) is instead being written to the file "directory.txt".

This redirection can also be performed from within a program by using the **`dup2()`** system call, which closes an existing file descriptor and points it at the same file as another file descriptor (see the man pages for details). As its name implies, **`dup2`** is used to copy or duplicate a file handle. The file redirection described above is implemented in the shell by using **`dup2()`**.

Basic I/O-related System Calls

Most UNIX I/O can be performed using the system calls **`open()`**,**`close()`**,**`read()`**, and **`write()`**. A file is opened by giving its name, the mode (read/write), and file permissions (if you are creating it). The associated file descriptor is returned. For example:

```
  fd = open("input.dat", O_RDONLY);
```

or

```
fd = open("record.dat", O_RDWR | O_CREAT, 0644);
```

A file is closed by giving its file descriptor as the only argument:

```
  close(fd);
```

Input and output (reading/writing) are symmetric system calls:

```
  numRead = read(fd, buffer, bufSize);
```

and

```
  numSent = write(fd, buffer, bufSize);
```

Familiarize yourself with the use of the `open()`,`close()`,`read()`,`write()`, and `dup2()` system calls.

Answer the following questions:

Suppose a process has used `dup2()` to make its standard output point to a file named `temp`. The process then issues a `fork()` system call to spawn a child. The child code issues an `exec()` system call to execute a different program.

1. Where does the standard output of the child process go? Explain.

Suppose a process issues a `fork()` system call to spawn a child process. The parent then uses `dup2()` to make its standard output point to a file named `temp`. The child code issues an `exec()` system call to execute a different program.

2. Where does the standard output of the child process go? Explain.

Pipes

The **`pipe()`** system call creates a unidirectional communication buffer space managed by the operating system. It also initializes two file descriptors - one associated with each end of the pipe (the "writing" end and the "reading" end). This is similar to opening a file and getting a file handle back, except there is both a reading and a writing handle. Using the `read()` and `write()` system calls, the pipe can be used for communication by connecting a process to each end, and proceeding as if you are reading/writing a file. Pipe communication is *unidirectional*; so typically one process (the Producer) writes to a pipe and the other process (the Consumer) reads from the pipe. Therefore, the writer closes their "read" handle, and the reader closes their "write" handle.

Note: redirection is not required; i.e., when appropriate, pipe handles can be used directly for I/O, just like file handles. It's simply convenient in certain cases.

The following program illustrates the general concept of interprocess communication and many of the system calls discussed in this section of the lab.

sampleProgramTwo

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

#define READ 0
#define WRITE 1

int main()
```

```
{
  int fd[2];
  int pipeCreationResult;
  int pid;

  pipeCreationResult = pipe(fd);

  if(pipeCreationResult < 0){
    perror("Failed pipe creation\n");
    exit(1);
  }

  pid = fork();

  if(pid < 0)    // Fork failed
    {
      perror("Fork failed");
      exit(1);
    }



  int output = 3;
  int input;

  if(pid == 0)
    {  // Child process
      write(fd[1], &output, sizeof(int));
      printf("Child wrote [%d]\n", output);
    }
  else
```

```
  {
    read(fd[0], &input, sizeof(int));
    printf("Parent received [%d] from child process\n", input);
  }


 return 0;



}
```

Perform the following and answer the questions:

Study, compile and run sampleProgramTwo

3. What exactly does the program do (i.e., describe its high-level functionality)?

Add the following variables to sample program 2:

```
char myStringOutput[] = "This a test!";

char myStringInput[50];
```

Modify the sample program to have the child write *myStringOutput* to the pipe (instead of '*output*').

Modify the sample program to have the parent read the string into *myStringInput* and output it along with the number of bytes that were read (instead of '*input*').

Close each end of the pipe that isn't being used before you write to the pipe.

Submit the modified source code along with a screenshot of the execution.

Lab Programming Assignment (Communicating Processes)

Interprocess communication of some sort occurs in most significant programming projects. Asynchronous communication is an IPC technique often

used in systems programming, where processes must be able to respond to a variety of possible inputs and events. For example, a child process may need to communicate to its parent that some condition has occurred, or not. Experiment with asynchronous IPC by writing the following program that demonstrates the use of signals for simple communication.

- Write a parent program that:
  - Spawns off a child process
  - Installs signal handler(s) for the two *user-defined* signals (`SIGUSR1/SIGUSR2`)
  - When a user-defined signal is received, it reports the type of signal sent
    - note: it may be necessary to reinstall your signal handler after a signal is received
  - Terminates gracefully upon receiving a Control-C
- The child process should repeatedly:
  - Wait a random amount of time (e.g. one to five seconds)
  - Randomly send one of the two user-defined signals to its parent

The program that produced the sample output listed below begins by spawning a child process. It then reports the identity of any signal that is sent to it at random times by its child. Finally, the main program is terminated and shutdown gracefully via an interrupt (Control-C) sigHandler.

Sample Output:

```
[eos12:~/cs452]$ labThreeProgrammingAssignment
spawned child PID# 19772
waiting...       received a SIGUSR2 signal
waiting...       received a SIGUSR1 signal
waiting...       received a SIGUSR2 signal
waiting...       received a SIGUSR2 signal
waiting...       received a SIGUSR1 signal
waiting...       received a SIGUSR2 signal
waiting...       received a SIGUSR1 signal
waiting...       received a SIGUSR1 signal
waiting...       received a SIGUSR1 signal
waiting...       received a SIGUSR1 signal
waiting...      ^C received.  That's it, I'm shutting you down...
[eos12:~/cs452]$
```

Submit the modified source code along with a screenshot of the execution.