Lab Exercise #3 -- Linux System Calls and Signals

This exercise focuses on the system calls and library functions available under the Linux operating system. Recall that providing a high-level interface to the hardware resources is one of the main functions of an operating system.

A. Linux System Calls

The heart of the Linux operating system is a set of system calls that are used to manage the system’s resources (primarily the CPU, memory, and the file system). For example, the following system call could be used inside a C/C++ program to create a file named "lab03.example" in the current directory:

```
   fd = open( "lab03.example", O_CREAT|O_RDONLY, S_IRUSR );
```

An overview of the system calls is available through the "man" utility:

```
   man 2 intro
```

Information about individual system calls is also available. For example:

```
   man 2 open
```

Users interact with Linux by executing programs which invoke the appropriate system call(s) to perform the desired task(s).

Fortunately, a typical Linux environment already contains many programs written by system programmers to perform common tasks, such as GUIs (graphical user interfaces), shells (interactive command interpreters), and utility programs ("ls", "rm", etc).

In addition, most programming languages provide libraries of functions which make it easier for users to invoke system calls correctly. For example, the standard C library contains a function "fopen" to create and open files. Of course, "fopen" invokes "open" to do some of the work.

Thus, most programmers employ a combination of a GUI, a shell, and their own programs containing library functions and system calls to perform tasks on a Linux platform.

Use "man 2 chdir" to display the information about the "chdir" system call. Review that information, then perform the experiments below.

a) Write a C/C++ program which changes the current directory using the "chdir" system call, where the destination directory is supplied as a command-line argument to the program. After changing the current directory, the program will use the "getcwd" library function (described in Section 3 of the on-line manual) to retrieve and display the full pathname of the current directory.

b) Error checking is a critical part of systems programming. All of the system calls return a flag (usually -1 or NULL) when an error is encountered. Use "man 2 intro" and "man 3 errno" to review information about the error handling used in system calls. In particular, examine the portions having to do with the variable "errno" and the symbolic names defined for various error conditions.

Include the following preprocessor directive in your program:

```
   #include <errno.h>
```

Modify your program so that it determines whether or not the call to "chdir" was successful, and prints out an appropriate error message if it was not. Note that the manual page lists the possible errors that could be encountered for a given system call. Your program should specifically check for the error conditions "ENOENT", "EACCES", and "ENOTDIR". Design test cases that exercise your program’s functionality.

B. Linux Signals

A signal is the software equivalent of a hardware interrupt. A signal is sent to a process to notify it of some event; the process may selectively execute a different portion of its code in response to the signal. Or, it may cause the process to terminate.

Some signals are directly related to events in the process receiving the signal. For example, the "SIGSEGV" signal is sent to a process when it attempts to reference memory in an illegal way, and the "SIGFPE" signal is sent to a process is if it attempts an invalid floating point operation.

Other signals are generated by the operating system in response to events that are external to the process receiving the signal. For example, the "SIGCHLD" signal is sent to the parent of a child process when that child terminates.

Use "man 7 signal" to review general information about signals. Note that this manual page has some discussion about threads; at this point, you can treat those as references to ordinary processes (executing programs). Part of the process of learning to read manual pages is learning how to skim them to find the information that is important, while ignoring details that may only be important in certain situations.

a) Briefly describe the options a process has when it receives a signal from the operating system.

b) There are two ways that a process can register the fact that it wants a certain function to be executed when a signal is sent to the process: using the "sigaction" system call or the "signal" library function. Review the information about "signal" in the on-line manual.

The "signal" function accepts two parameters and is used to indicate the particular function the program should call when a specific signal is received. The first parameter is an integer identifying the signal (always use the symbolic constants defined for these signals rather than the integer values themselves). The second parameter is the constant "SIG_IGN", the constant "SIG_DFL", or the address of a function. Note: if you use the name of a function without parentheses indicating parameters, it is treated as a reference to the address where that function is located in memory.

Review the contents of "˜cse410/Labs/lab03.signal.c". Compile, link and execute the program, then briefly describe the functionality of the program.

c) Review the information about "ps" and "kill" in Section 1 of the on-line manual. Then, start up a second shell session (in a different window).

Execute the program from "lab03.signal.c" in one window, then use "kill" in the second window to terminate that program. Note that you’ll need the process identification number (PID) of the program, which you can find using "ps". Give the "ps" and "kill" commands which you used.

d) Review the information about "getpid" and "kill" in Section 2 of the on-line manual.