**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS**

**Information and Computer Science Department**

# ICS 431 Operating Systems

**Lab # 6**

**Inter-Process Communication: Signals**

**Objectives:**

 To learn how to use signals

**Signals**

Programs must sometimes deal with unexpected or unpredictable events, such as:

* a floating point error
* a power failure
* an alarm clock "ring"
* the death of a child process
* a termination request from a user (i.e., a *Control-C*)
* a suspend request from a user (i.e., a *Control-Z*)

These kinds of events are sometimes called interrupts, as they must interrupt the regular flow of a program in order to be processed. When Linux recognizes that such an event has occurred, it sends the corresponding process a signal. There is a unique, numbered signal for each possible event. For example, if a process causes a floating point error, the kernel sends the offending process signal number 8.

The kernel isn't the only one that can send a signal; any process can send any other process a signal, as long as it has permissions.

A programmer may arrange for a particular signal to be ignored or to be processed by a special piece of code called a signal handler. In the latter case, the process that receives the signal suspends its current flow of control, executes the signal handler, and then resumes the original flow of control when the signal handler finishes.

By learning about signals, you can "protect" your programs from Control-C, arrange for an alarm clock signal to terminate your program if it takes too long to perform a task, and learn how Linux uses signals during everyday operations.

**Defined Signals:**

Signals are defined in "/usr/include/signal.h" and the other platform-specific header files it includes (the actual signal definitions are in "/usr/include/asm/signal.h"). A programmer may choose that a particular signal triggers a user-supplied signal handler, triggers the default kernel-supplied handler, or is ignored. The default handler usually performs one of the following actions:

* terminates the process and generates a dump of memory in a core file (core)
* terminates the process without generating a core image file (quit)
* ignores and discards the signal (ignore)
* suspends the process (stop)
* resumes the process

**Types of Signals:**

The following is a list of the standard POSIX signals defined in Linux along with their macro definition, numeric value, process's default action, and a brief description.

|  |  |
| --- | --- |
| **Macro # Default action Description** | |
| SIGHUP 1 quit | Hangup or death of controlling process. |
| SIGINT 2 quit | Keyboard interrupt. |
| SIGQUIT 3 core | Quit. |
| SIGILL 4 core | Illegal instruction. |
| SIGABRT 6 core | Abort. |
| SIGFPE 8 core | Arithmetic exception. |
| SIGKILL 9 quit | Kill (cannot be caught, blocked, or ignored). |
| SIGUSR1 10 quit | User-defined signal. |
| SIGSEGV 11 core | Segmentation violation (out of range address). |
| SIGUSR2 12 quit | User-defined signal. |
| SIGPIPE 13 quit | Write on a pipe or other socket with no one to read it. |
| SIGALRM 14 quit | Alarm clock. |
| SIGTERM 15 quit | Software termination signal (default signal sent by kill). |
| SIGCHLD 17 ignore | Status of child process has changed. |
| SIGCONT 18 none | Continue if stopped. |
| SIGSTOP 19 stop | Stop (suspend) the process. |
| SIGTSTP 20 stop | Stop from the keyboard. |
| SIGTTIN 21 stop | Background read from tty device. |
| SIGTTOU 22 stop | Background write to tty device. |

**Terminal Signals:**

The easiest way to send a signal to a foreground process is by pressing Control-C or ControlZ from the keyboard. When the terminal driver (the piece of software that supports the terminal) recognizes a Control-C, it sends a SIGINT signal to all of the processes in the current foreground job. Similarly, Control-Z causes it to send a SIGTSTP signal to all of the processes in the current foreground job. By default, SIGINT terminates a process and SIGTSTP suspends a process.

**System Call alarm ( )**

One of the simplest ways to see a signal in action is to arrange for a process to receive an alarm clock signal, SIGALRM, by using alarm ( ). The default handler for this signal displays the message "Alarm Clock" and terminates the process. Here's how alarm ( ) works:

|  |
| --- |
| System Call: unsigned int alarm (unsigned int count) |
| alarm () instructs the kernel to send the SIGALRM signal to the calling process after count seconds. If an alarm had already been scheduled, it is overwritten. If count is 0, any pending alarm requests are cancelled. alarm () returns the number of seconds that remain until the alarm signal is sent. |

**System Call signal:**

|  |
| --- |
| System Call: void (\*signal (int sigCode, void (\*func)(int))) (int) |
| signal () allows a process to specify the action that it will take when a particular signal is received. The parameter sigCode specifies the number of the signal that is to be reprogrammed, and func may be one of several values: |
| * SIG\_IGN, which indicates that the specified signal should be ignored and discarded. * SIG\_DFL, which indicates that the kernel's default handler should be used. * an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives. |
| The valid signal numbers are included from "/usr/include/signal.h" (and the other header files that includes, the actual signal definitions are in "/usr/include/asm/signal.h" on my Linux machine). The signals SIGKILL and SIGSTP may not be reprogrammed. A child process inherits the signal settings from its parent during a fork (). When a process performs an exec (), previously ignored signals remain ignored but installed handlers are set back to the default handler. |
| With the exception of SIGCHLD, signals are not stacked. This means that if a process is sleeping and three identical signals are sent to it, only one of the signals is actually processed. |
| signal () returns the previous func value associated with sigCode if successful; otherwise it returns -1. |

**System Call pause:**

|  |
| --- |
| System Call: int pause (void) |
| pause () suspends the calling process and returns when the calling process receives a signal. It is most often used to wait efficiently for an alarm signal. pause () doesn't return anything useful. |

**System Call kill:**

A process may send a signal to another process by using the kill () system call. kill () is a misnomer, since many of the signals that it can send do not terminate a process. It's called kill () for historical reasons; back when UNIX was first designed, the main use of signals was to terminate processes.

|  |
| --- |
| System Call: int kill (pid\_t pid, int sigCode) |
| kill () sends the signal with value sigCode to the process with PID pid. kill () succeeds and the signal is sent as long as at least one of the following conditions is satisfied:   * The sending process and the receiving process have the same owner. * The sending process is owned by a super-user.   There are a few variations on the way that kill () works:   * If pid is 0, the signal is sent to all of the processes in the sender's process group. * If pid is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender. * If pid is -1 and the sender is not a super-user, the signal is sent to all of the processes owned by the same owner as the sender, excluding the sending process. * If the pid is negative and not -1, the signal is sent to all of the processes in the process group.   If kill () manages to send at least one signal successfully, it returns 0; otherwise, it returns -1. |

**Example#1:**

#include <stdio.h>

#include <unistd.h>

int main(void)

{

    alarm(5);

    printf("Looping forever.\n");

    while(1);

    return 0;

}

**Example#2:**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

int alarmFlag = 0;

void alarmHandler();

int main() {

    signal(SIGALRM, alarmHandler);

    alarm(5);

    printf("Looping ...\n");    while (!alarmFlag)

    {

        pause();

    }

    printf("Loop ends due to alarm signal\n");      return 0;

}

void alarmHandler()

{

    printf("An ALARM clock signal was received\n");     alarmFlag = 1;

}

**Example#3:**

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

int main() {

    printf("I can be Control-C'ed \n");

    sleep(5);

    signal(SIGINT, SIG\_IGN);

    printf("I am protected from Control-C now \n");

    sleep(5);

    signal(SIGINT, SIG\_DFL);

    printf("I can be Control-C'ed again \n");

    sleep(5);

    printf("Bye!!!!!!!\n");

    return 0;

}

Exercises:

Q. 1: Write a program that generates a multiplication question where the operands are one digit numbers being randomly generated. Display the generated question then prompt the user to enter an answer.

The maximum time allowed for answering the question is 10 seconds. The program should display the correct answer if the user enters a wrong answer or in case the user fails to answer within 10 seconds.

Q. 2: One of the operations which generates a SIGFPE is mod by '0'. Write a program that prompts the user to enter two integers. The program then displays the remainder of the first integer by the second integer. If the second integer is '0', then the program prints a nice message ('Oops! Division by zero') by creating a handler for SIGFPE and then exits (Do not use if-statement to check for '0' value of second integer).