OPERATING SYSTEMS LAB

LAB MANUAL # 11

Instructor: Muhammad Abdullah Orakzai

Semester Fall 2021

Course Code: CL2006

Fast National University of Computer and Emerging Sciences, Peshawar

Department of Computer Science

Contents

Inter Process Communication

IPC-Signals

Signals are an inter-process communication mechanism provided by operating systems which facilitate communication between processes. It is usually used for notifying a process regarding a certain event. So, we can say that signals are an ***event-notification system*** provided by the operating system.

Signal Delivery Using Kill

Kill is the delivery mechanism for sending a signal to a process. Unlike the name, a kill() call is used only to send a signal to a process. It does not necessarily mean that a process is going to be killed (although it can do exactly that as well). As mentioned earlier, the signal facility is just an event notification facility provided by the operating system.

For Example, a shutdown signal (SIGHUP) can be sent to all processes currently active in the system in order to notify them about a shutdown process event. Upon receipt of this signal, all processes will prepare to terminate. Once the processes terminate, the system can shut down.

List of Signals

The following table lists out common signals you might encounter and want to use in your programs –

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C)
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D)
SIGFPE	8	Issued if an illegal mathematical operation is attempted

SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default)

The Kill facility can be used in two ways; as a command from your command prompt, or via the kill() call from your program.

1. Kill Command

The syntax of the kill command is as such:

kill -s PID

Here, ***kill*** is the command itself, ***-s*** is an argument which specifies the type of signal to send, and PID is the integer identifier of the process to which a signal is going to be delivered. The list of signals for **-s** argument can be seen from the following:

```
kill -l
```

Hence, supposing that we want to terminate a process, we may enter

```
kill -9 12345
```

Where 12345 will be a process id of any active process in the system.

For Example,

ps

Note pid of bash process and write the following command in terminal

kill -9 3374

By using this command your terminal will closed.

Exercise

Let us kill our bash shell using the TERM signal. Find out:

- The integer representation for the SIGTERM signal
- The PID of your current active bash shell using the ps command

Then, use the kill command to send over this signal.

2. kill() System Call

Usage of the Kill() system call is as such:

```
kill(int, int);
```

For this to work, we will require the **sys/types.h** & **signal.h** C libraries. Here, the 1st parameter is the integer number of signal type (again, can be checkable from kill -l), and the 2nd parameter is the PID of the process to which signal is to be delivered. As an example, if we want to send the Terminate signal to the current process, we will use

~~kill(SIGTERM, getpid())~~

kill(getpid( ), SIGTERM)

or

~~kill(9, getpid())~~

kill(getpid( ), 9)

SIGTERM: A termination request sent to the program.

kill.c

```
#include <unistd.h>
#include <stdio.h>
#include<signal.h>
#include <sys/types.h>
int main()
{
printf("My PID: %d\n",getpid());
printf("Hello\n");
sleep(10);
kill(getpid(),9);
printf("GoodBye\n");
return 0;
}
```

Output:

My PID: 23184

Hello

Killed

Don't think that **kill()** is to terminate a process only. It can send all kinds of signals.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
int main(int argc,char *argv[])
{
   printf("My PID: %d\n",getpid());
   printf("Hello\n");
   sleep(5);
    kill(getpid(),SIGSEGV);  // An invalid access to storage.
   return 0;
}
```

Output:

My PID: 23399

Hello

Segmentation fault (core dumped)

SIGSEGV: An invalid access to storage.

SigSegV means a signal for memory access violation, trying to read or write from/to a memory area that your process does not have access to.

Exercise # 01

Write code which does the following:

- Parent process creates a child process using fork() call (using proper if/else statement blocks).
- Parent sends the terminate signal via kill() call to child and then waits for 120 seconds.

While parent is waiting, the user should check the outcome of **ps au** command.

kill2.c

```
#include <unistd.h>
#include <stdio.h>
#include<signal.h>
#include <sys/types.h>
int main()
{
int pid;
pid = fork();
if(pid == 0)
{
printf("Child PID = %d\n", pid);
printf("I am the Child with PID = %d\n", getpid());
}
else
{
printf("Parent of = %d\n", pid);
printf("I am the Parent with PPID = %d\n",getppid());
kill(pid,SIGTERM);
sleep(120);
}
}
```

Exercise # 02

Modify the code in **Exercise # 01** such that signal is sent from child to parent. and then have your child wait for 120 seconds.

Again, while child is waiting, the user should check the outcome of **ps au** command.

Sending signals via kill()

We can send a signal using kill() to the process.

```
int kill(pid_t pid, int signal);
pid: id of destination process
signal: the type of signal to send
Return value: 0 if signal was sent successfully
```

Example:

```
pid_t iPid = getpid(); /* Process gets its id.*/
kill(iPid, SIGINT);  /* Process sends itself a  SIGINT signal
(commits suicide?)(because of SIGINT
signal default handler terminate the process) */
```

Signals

A signal is a software generated interrupt that is sent to a process by the OS because of when user press ctrl-c or another process tell something to this process. There are fix set of signals that can be sent to a process. Signals are identified by integers. Signal number have symbolic names. **For example, SIGCHLD** is number of the signal sent to the parent process when child terminates.

Examples:

```
#define SIGHUP  1   /* Hangup the process */
#define SIGINT  2   /* Interrupt the process */
#define SIGQUIT 3   /* Quit the process */
#define SIGILL  4   /* Illegal instruction. */
#define SIGTRAP 5   /* Trace trap. */
#define SIGABRT 6   /* Abort. */
```

1. Default Signal Handlers

There are several default signal handler routines. Each signal is associated with one of these default handler routines. The different default handler routines typically have one of the following actions:

- Ign: Ignore the signal; i.e., do nothing, just return
- Term: terminate the process
- Cont: unblock a stopped process
- Stop: block the process

signal1.c

```
// CPP program to illustrate default Signal Handler
#include<stdio.h>
#include<signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
   signal(SIGINT, SIG_DFL);
   while (1)
    {
        printf("hello world\n");
        sleep(1);
    }
        return 0;
}
```

Output: Print hello world infinite times. If user presses ctrl-c to terminate the process because of **SIGINT** signal sent and its default handler to terminate the process.

```
hello world
hello world
hello world
terminated
```

2. User Defined Signal Handlers

A process can replace the default signal handler for almost all signals (but not SIGKILL) by its user's own handler function.

A signal handler function can have any name, but must have return type void and have one int parameter.

Example: you might choose the name sigchld_handler for a signal handler for the **SIGCHLD** signal (termination of a child process). Then the declaration would be:

```
void sigchld_handler(int sig);
```

When a signal handler executes, the parameter passed to it is the number of the signal. A programmer can use the same signal handler function to handle several signals. In this case the handler would need to check the parameter to see which signal was sent. On the other hand, if a signal handler function only handles one signal, it isn't necessary to bother examining the parameter since it will always be that signal number.

signal2.c

```
// CPP program to illustrate User-defined Signal Handler
#include<stdio.h>
#include<signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
// Handler for SIGINT, caused by
// Ctrl-C at keyboard
void handle_sigint(int sig)
{
        printf("Caught signal %d\n", sig);
}
int main()
{
```

```
    signal(SIGINT, handle_sigint);
    while (1) ;
    return 0;
}
```

Output:

```
^CCaught signal 2  // when user presses ctrl-c
^CCaught signal 2
```

Signal Handling Using Signal

Signal delivery is handled by the kill command or the kill() call. The process receiving the signal can behave in a number of ways, which is defined by the signal() call.

The syntax of the call is as such:

```
signal(int, conditions)
```

The signal will require the **signal.h** C library to work. From the syntax above, signal() is the system call, the 1st parameter **int** is the integer identifier of the respective signal which is sent, and the last parameter is either of the following:

- **SIG_DFL** which will perform the default mechanism provided by the operating system for that particular signal.
- **SIG_IGN** which will ignore that particular signal if it is delivered.
- Any function name (for programmer defined signal handling purposes).

As an example, we are going to send a process the **SIGINT** signal and count the total number of times that it is received. See the code below for this purpose:

signal3.c

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int sigCounter = 0;
void sigHandler(int sigNum)
```

```
{
printf("Signal received is %d\n", sigNum);
++sigCounter;
printf("Signals received %d\n", sigCounter);
}
int main()
{
signal(SIGINT, sigHandler);
while(1)
{
printf("Hello Dears\n");
sleep(1);
}
return 0;
}
```

Here, signal is the signal handler call and accepts as input the signal number, and the name of function which is going to behave as signal handler. (If we want it to behave the default way, we specify **SIG_DFL**, or if we want it to ignore a certain signal, we specify **SIG_IGN**).

When we run the program, we are instructing our program that if in case the **SIGINT** signal is detected, we will perform the steps provided in the **sigHandler** function. As long as there is no event, the program will keep on executing the while loop. The event can be delivered by pressing **CTRL+C.**

Try pressing **CTRL+C** with, and without the signal() call and you will understand the difference yourselves.

References

https://www.geeksforgeeks.org/kill-command-in-linux-with-examples/

https://calvinkam.github.io/csci3150-process/kill-system-call.html

https://www.geeksforgeeks.org/signals-c-language/