National University of Computer and Emerging Sciences

Operating System Lab – 12

Lab Manual

Contents

Objective

In Inter-process communication there are many mechanisms through which the processes communicate and in this lab we will discuss one such mechanism: Signals. Signals inform processes of the occurrence of asynchronous events. In this lab we will discuss how user-defined handlers for particular signals can replace the default signals handlers and also how the processes can ignore the signals.

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 UNIX uses signals during everyday operations.

Introduction

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 kind of events are sometimes called interrupts, as they must interrupt the regular flow of a program in order to be processed. When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal.

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.

Signals inform processes of the occurrence of asynchronous events. Every type of signal has a handler which is a function. All signals have default handlers which may be replaced with user-defined handlers. The default signal handlers for each process usually terminate the process or ignore the signal, but this is not always the case.

Types of Signals

A programmer may choose for a particular signal to trigger a user-supplied signal handler, trigger the default kernel-supplied handler, or be ignored. The default handler usually performs one of the following actions:

- terminates the process and generates a core file (dump) terminates
- the process without generating a core image file (quit) ignores and
- discards the signal (ignore)
- suspends the process (suspend)
- resumes the process

Some signals are widely used, while others are extremely obscure and used by only one or two programs. The following list gives a brief explanation of each signal. The default action upon receipt of a signal is for the process to terminate.

```
student@OSLAB-VM:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

```
student@OSLAB-VM:~$ man 7 signal
```

```
       First the signals described in the original POSIX.1-1990 standard.

       Signal      Value     Action   Comment
       ──────────────────────────────────────────────────────────────────────
       SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2       Term    Interrupt from keyboard
       SIGQUIT       3       Core    Quit from keyboard
       SIGILL        4       Core    Illegal Instruction
       SIGABRT       6       Core    Abort signal from abort(3)
       SIGFPE        8       Core    Floating point exception
       SIGKILL       9       Term    Kill signal
       SIGSEGV      11       Core    Invalid memory reference
       SIGPIPE      13       Term    Broken pipe: write to pipe with no
                                     readers
       SIGALRM      14       Term    Timer signal from alarm(2)
       SIGTERM      15       Term    Termination signal
       SIGUSR1   30,10,16    Term    User-defined signal 1
       SIGUSR2   31,12,17    Term    User-defined signal 2
       SIGCHLD   20,17,18    Ign     Child stopped or terminated
       SIGCONT   19,18,25    Cont    Continue if stopped
       SIGSTOP   17,19,23    Stop    Stop process
       SIGTSTP   18,20,24    Stop    Stop typed at terminal
       SIGTTIN   21,21,26    Stop    Terminal input for background process
       SIGTTOU   22,22,27    Stop    Terminal output for background process
```

SIGINT

Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-C) is hit. It is used to kill a process by other words it is a polite kill.

SIGSTOP

Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-Z) is hit. It is used to suspend a process by sending it the signal, which is like a sleep signal, that can be undone and the process can be resumed again.

SIGQUIT

Quit. Similar to SIGINT, but sent when the quit key (normally Control-\) is hit. Commonly sent in order to get a core dump.

SIGILL

Illegal instruction. Sent when the hardware detects an illegal instruction. Sometimes a process using floating point aborts with this signal when it is accidentally linked without the -f option on the cc command. Since C programs are in general unable to modify their instructions, this signal rarely indicates a genuine program bug.

SIGTRAP

Trace trap. Sent after every instruction when a process is run with tracing turned on with ‘ptrace’.

SIGIOT

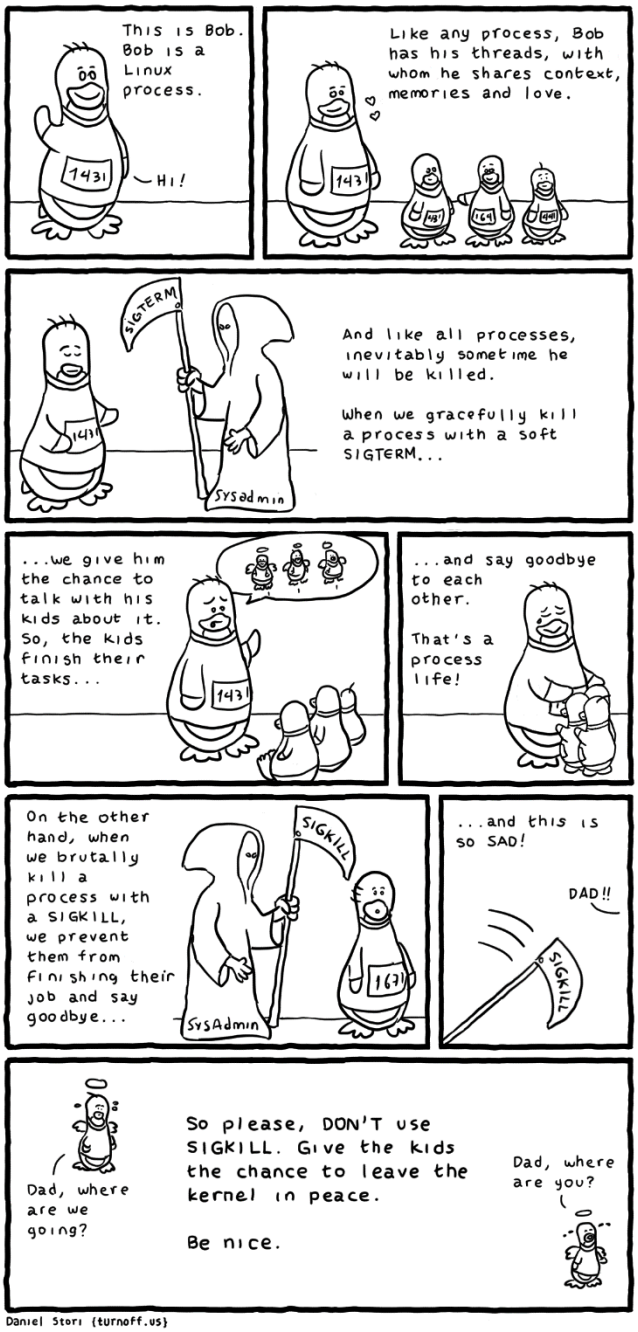
I/O trap instruction. Sent when a hardware fault occurs, the exact nature of which is up to the implementer and is machine-dependent.

SIGKILL

Kill. The one and only sure way to kill a process, since this signal is always fatal (can't be ignored or caught). To be used only in emergencies; SIGTERM is preferred.

SIGTERM

Software termination. The standard termination signal. It's the default signal sent by the kill command, and is also used during system shutdown to terminate all active processes. A program should be coded to either let this signal default or else to clean (e.g., remove temporary files) and call exit.



SIGEMT

Emulator trap instruction. Sent when an implementation-dependent hardware fault occurs. Extremely rare.

SIGFPE

Floating-point exception. Sent when the hardware detects a floating-point error, such as a floating point number with an illegal format. Almost always indicates a program bug.

SIGBUS

Bus error. Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced at an odd address data that should have been word aligned.

SIGSEGV

Segmentation violation. Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced data outside its address space. Trying to use NULL pointers will usually give you a SIGSEGV.

SIGPIPE

Write on a pipe not opened for reading. Sent to a process when it writes on a pipe that has no reader. Usually this means that the reader was another process that terminated abnormally. This signal acts to terminate all processes in a pipeline: When a process terminates abnormally, all processes to its right receive an end-of-file and all processes to its left receive this signal.

SIGALRM

Alarm clock. Sent when a process's alarm clock goes off. The alarm clock is set with the alarm system call.

SIGUSR1

User defined signal 1. This signal may be used by application programs for interprocess communication. This is not recommended however, and consequently this signal is rarely used.

SIGUSR2

User defined signal 2. Similar to SIGUSR1.

SIGPWR

Power-fail restart. Exact meaning is implementation-dependent. One possibility is for it to be sent when power is about to fail (voltage has passed, say, 200 volts and is falling). The process has a very brief time to execute. It should normally clean up and exit (as with SIGTERM). If the process wishes to survive the failure (which might only be a momentary voltage drop), it can clean up and then sleep for a few seconds. If it wakes up it can assume that the disaster was only a dream and resume processing. If it doesn't wake up, no further action is necessary.

Programs that need to clean up before terminating should arrange to catch signals SIGHUP, SIGINT, and SIGTERM. Until the program is solid, SIGQUIT should be left alone so there will be a way to terminate the program (with a core dump) from the keyboard. Arrangements for the other signals are made much less often; usually they are left to terminate the process. But a really polished program will want to catch everything it can, to clean up, possibly log error, and print a nice error message. Psychologically, a message like “internal error 53: contact customer support" is more acceptable than the message ``Bus error – core dumped’’ from the shell. For some signals, the default action of termination is accompanied by a core dump. These are SIGQUIT, SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, and SIGSYS.

Requesting An Alarm Signal: 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:

```
int alarm (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.

Here's a small program that uses alarm ( )

```
#include <stdio.h>
main ( )
{
//alarm (5) ; /* schedule an alarm signal in 5 seconds */
printf ("Looping forever ...\n") ;
while ( 1 ) ;
printf ("This line should never be executed.\n") ;
}
```

‘signal’ System Call

```
#include <signal.h>

signal(sig, func) /* Catch signal with func */

void (func)();/* The function to catch the sig

/*Returns the previous handler or -1 on error */
```

The first argument, sig, is a signal number. The second argument, func, can be one of three things:

- SIG_DFL. This sets the default action for the signal.
- SIG_IGN. This sets the signal to be ignored; the process becomes immune to it. The signal SIGKILL can't be ignored. Generally, only SIGHUP, SIGINT, and SIGQUIT should ever be permanently ignored. The receipt of other signals should at least be logged, since they indicate that something exceptional has occurred.
- A pointer to a function. This arranges to catch the signal; every signal but SIGKILL may be caught. The function is called when the signal arrives.

A child process inherits a parent’s action for a signal. Actions SIG_DFL and SIG_IGN are preserved across an exec, but caught signals are reset to SIG_DFL. This is essential because the catching function will be overwritten by new code. Of course, the new program can set its own signal handlers. Arriving signals are not queued. They are either ignored, they terminate the process, or they are caught. This is the main reason why signals are inappropriate for inter-process communication -- a message in the form of a signal might be lost if it arrives when that type of signal is temporarily ignored. Another problem is that arriving signals are rather rude. They interrupt whatever is currently going on, which is complicated to deal with properly, as we'll see shortly. Signal returns the previous action for the signal. This is used if it's necessary to restore it to the way it was.

‘pause’ System Call

```
Int pause()
```

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.

The following program catches and processes the SIGALRM signal efficiently by having user written signal handler, alarmHandler ( ), by using signal ( ).

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

int alarmFlag = 0 ;
void alarmHandler ( ) ;

int main ( ) {
signal(SIGALRM, alarmHandler) ; /*Install signal Handler*/
alarm (5) ;
printf ("Looping ...\n") ;
while (!alarmFlag) {
pause ( ); /* wait for a signal */
printf ("Loop ends due to alarm signal\n");
}
return 0;}

void alarmHandler ( ) {
printf ("An ALARM clock signal was received\n");
alarmFlag = 1;
}
```

Ha! Ha!

```
#include <stdio.h>
#include <signal.h>
void haha(int sig);
int main()
{
int i;
signal(SIGINT, haha);

for (i = 1; i <= 2000000000; i++)
{
        //do nothing
}
return 0;
}

void haha(int sig)
{
printf("Ha! Ha! \n");
}
```

Protecting Critical Code and Chaining Interrupt Handlers

The same techniques described previously may be used to protect critical pieces of code against Control-C attacks and other signals. In these cases, it's common to save the previous value of the handler so that it can be restored after the critical code has executed. Here's the source code of the program that protects itself against SIGINT signals:

```
#include<stdio.h>
#include<signal.h>
main ( ) {
printf ("I can be Control-C'ed \n") ;
sleep (5) ;
signal(SIGINT, SIG_IGN) ; /* Ignore Ctrl-C */
printf ("I am protected from Control-C now \n") ;
sleep (5) ;
signal (SIGINT, SIG_DFL); /* Restore old handler */
printf ("\nI can be Control-C'ed again \n") ; sleep (5) ;
printf ("Bye!!!!!!!\n");
}
```

'kill' System Call & Inter Process Communication

```
void hello(){
  printf("Hello World!\n");
}

int main(){

  pid_t cpid;
  pid_t ppid;

  //set handler for SIGUSR1 to hello()
  signal(SIGUSR1, hello);

  if ( (cpid = fork()) == 0){
    /*CHILD*/

    //get parent's pid
    ppid = getppid();

    //send SIGUSR1 signal to parrent
    kill(ppid, SIGUSR1);
    exit(0);

  }else{
    /*PARENT*/

    //just wait for child to terminate
    wait(NULL);
  }

}
```

- One process can send a signal to another indicating that an action should be taken. To send a signal to a particular process, we use the kill() system call. The function declaration is below.
- int kill(pid_t pid, int signum);

If pid is equal to zero, the signal is sent to every process in the same process group as the sender. This feature is frequently used with the kill command (kill 0) to kill all background processes without referring to their process-IDs. Processes in other process groups won't receive the signal.

If pid is equal to -1, the signal is sent to all processes whose real user-ID is equal to the effective userID of the sender. This is a handy way to kill all processes you own, regardless of process group.

In practice, kill is used 99% of the time for one of these purposes:

Kill is almost never used simply to inform one or more processes of something (i.e., for inter-process communication), for the reasons outlined in the previous sections. Note also that the kill system call is most often executed via the kill command. It isn't usually built into application programs.

Changing and Reverting to default handler

```c
void handler_3(int signum){
  printf("Don't you dare shoot me one more time!\n");

  //Revert to default handler, will exit on next SIGINT
  signal(SIGINT, SIG_DFL);
}

void handler_2(int signum){
  printf("Hey, you shot me again!\n");

  //switch handler to handler_3
  signal(SIGINT, handler_3);
}

void handler_1(int signum){
  printf("You shot me!\n");

  //switch handler to handler_2
  signal(SIGINT, handler_2);
}


int main(){

  //Handle SIGINT with handler_1
  signal(SIGINT, handler_1);

  //loop forever!
  while(1);

}
```

1) The program first initiates handler_1() as the signal handler for SIGINT.

2) After the first Ctrl-c, in the signal handler, the handler is changed to handler_2(), and after the second Ctrl-c,

3) it is change again to handler_3() from handler_2 ().

4) Finally, in handler_3() the default signal handler is reestablished, which is to terminate on SIGINT, and that is what we see in the output:

Some signals are more equal than others

– The two signals that can never be ignored or handled are: SIGKILL and SIGTSTOP. Let's look at an example:

```
int main(){

  //ignore SIGSTOP ?
  signal(SIGSTOP, SIG_IGN);

  //infinite loop
  while(1);

}
```

```
int main(){

  //ignore SIGSTOP ?
  signal(SIGKILL, SIG_IGN);

  //infinite loop
  while(1);

}
```

Lab Activity

Implement all 6 examples of the following link and submit screenshots.

http://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/signals/signals.html