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.

Signals may be sent to a process from another process, from the kernel, or from devices such as terminals. The ^C, ^Z, ^S and ^Q terminal commands all generate signals which are sent to the foreground process when pressed.

The kernel handles the delivery of signals to a process. Signals are checked for whenever a process is being rescheduled, put to sleep, or re-executing in user mode after a system call.

Types of Signals

There are 31 different signals defined in "/usr/include/signal.h". 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.

SIGHUP

Hang-up. Sent when a terminal is hung up to every process for which it is the control terminal. Also sent to each process in a process group when the group leader terminates for any reason. This simulates hanging up on terminals that can't be physically hung up, such as a personal computer.

SIGINT

Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-C) is hit. This action of the interrupt key may be suppressed or the interrupt key may be changed using the stty command. Note that suppressing the interrupt key is completely different from ignoring the signal, although the effect (or lack of it) on the process is the same.

SIGTSTP

Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-Z) is hit. This action of the interrupt key may be suppressed or the interrupt key may be changed using the stty command. Note that suppressing the interrupt key is completely different from ignoring the signal, although the effect (or lack of it) on the process is the same.

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. In practice, this signal is preempted by the standard subroutine abort, which a process calls to commit suicide in a way that will produce a core dump.

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.

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.

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. Note that the standard shell (sh) makes each process in a pipeline the parent of the process to its left. Hence, the writer is not the reader's parent (it's the other way around), and would otherwise not be notified of the reader's death.

SIGALRM

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

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.

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 ( ) together with its output.

```
#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>

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

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

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

The declarations here baffle practically everyone at first sight. All they mean is that the second argument to signal is a pointer to a function, and that a pointer to a function is returned.

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.

The signals SIGKILL and SIGSTP may not be reprogrammed.

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.

Defaulting and ignoring signals is easy; the hard part is catching them. To catch a signal you supply a pointer to a function as the second argument to signal.

When the signal arrives two things happen, in this order:

- The signal is reset to its default action, which is usually termination. Exceptions are SIGILL and SIGTRAP, which are not reset because they are signaled too often.
- The designated function is called with a single integer argument equal to the number of the signal that it caught. When and if the function returns, processing resumes from the point where it was interrupted.
- If the signal arrives while the process is waiting for any event at all, and if the signalcatching function returns, the interrupted system call returns with an error return of EINTR -- it is not restarted automatically. You must distinguish this return from a legitimate error. Nothing is wrong -- a signal just happened to arrive while the system call was in progress.

It's extremely difficult to take interrupted system calls into account when programming. You either has to program to restart every system call that can wait or else temporarily ignore signals when executing such a system call. Both approaches are awkward, and the second runs the additional risk of losing a signal during the interval when it's ignored. We therefore offer this rule: Never return from a signal-catching function. Either terminate processing entirely or terminate the current operation by executing a global jump (not described here). If you make it a habit to always print out the value of errno when a system call fails (by calling perror for example) you won't be mystified for long since the EINTR error code will clarify what's going on.

Since the first thing that happens when a caught signal arrives is to change its action to the default (termination), another signal of the same type arriving immediately after the first can terminate the process before it has a chance to even begin the catching function. This is rare but possible, especially on a busy system.

This loophole can be tightened, but not eliminated, by setting the signal to be ignored immediately upon entering the catching function, before doing anything else. Since we're not using signals as messages, we don't care if an arriving signal is thereby missed. We're concerned only with processing the first one correctly and with not terminating prematurely.

‘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 ( )
;

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");
}}

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

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 ( ) {
int (*oldHandler) ( ) ; /* holds old handler value */
printf ("I can be Control-C'ed \n") ;
sleep (5) ;
oldHandler = signal(SIGINT, SIG_IGN) ; /* Ignore Ctrl-C */
printf ("I am protected from Control-C now \
n") ; sleep (5) ;
signal (SIGINT, oldHandler) ; /* Restore old handler */
printf ("\nI can be Control-C'ed again \n") ;
sleep (5) ;
printf ("Bye!!!!!!!\n");
}
```

‘kill’ System Call

```
#include <signal.h>
int kill(pid, sig) /* Send the signal to the named process */
int pid;
int sig;
```

In the previous sections we mainly discussed signals generated by the kernel as a result of some exceptional event. It is also possible for one process to send a signal of any type to another process. pid is the process-ID of the process to receive the signal; sig is the signal number. The effective user-IDs of the sending and receiving processes must be the same, or else the effective user-ID of the sending process must be the super user.

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 (such as a DBMS you happened to have started) 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 user-ID 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:

To terminate one or more processes, usually with SIGTERM, but sometimes with SIGQUIT so that a core dump will be obtained.

To test the error-handling code of a new program by simulating signals such as SIGFPE (floating-point exception).

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.

Using ‘sigaction’ to handle interrupts

Another option that is frequently used to implement signal handling is sigaction, one working example is as follows:

```
#include <signal.h>
#include<stdio.h>
static void handler(int signum)
{
```

```
/* Take appropriate actions for signal delivery */

}
int main()
{
int ret = -1;
struct sigaction sa;
sa.sa_handler = handler;
sigemptyset(&sa.sa_mask)
;
sa.sa_flags = SA_RESTART; /* Restart functions if interrupted by handler */
if(sigaction(SIGINT, &sa, NULL) < 0) {
/* Handle error */

}
/* Further code */
```

Lab Activity

Task1

Convert the following code of signal into sigaction

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signo)
{
if (signo == SIGUSR1)
printf("received SIGUSR1\n");
else if (signo == SIGKILL)
printf("received SIGKILL\n");
else if (signo == SIGSTOP)
printf("received SIGSTOP\n");
}
int main(void)
{
if (signal(SIGUSR1, sig_handler) == SIG_ERR)
printf("\ncan't catch SIGUSR1\n");
if (signal(SIGKILL, sig_handler) == SIG_ERR)
printf("\ncan't catch SIGKILL\n");
if (signal(SIGSTOP, sig_handler) == SIG_ERR)
printf("\ncan't catch SIGSTOP\n");
// A long long wait so that we can easily issue a signal to this
process
while(1)
sleep(1);
return 0;
}
```

Task 2

Write a program to ignore SIGKILL and SIGSTOP

Task 3

Modify the code from last week’s lab of ‘Array Sum’ using pthread and semaphore such that each process needs to spend 5 seconds in critical section i.e. you will set an alarm for it and then exits the critical section using signal.