PRACTICAL 10
SIGNAL HANDLING

Program 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., Control-C)
- a suspend request from a user (i.e., 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.

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:

Signal Process
#8
FIGURE Floating point error signal

Any process can send any other process a signal, as long as it has permission to do so. 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***.

The process that receives the signal

1) suspends its current flow of control,
2) executes the signal handler,
3) and then resumes the original flow of control when the signal handler finishes.

Signals are defined in “/usr/include/sys/signal.h”.
The default handler usually performs one of the following actions:

- terminate the process and generate a core file (*dump*)
- terminate the process without generating a core image file (*quit*)
- ignore and discard the signal (*ignore*)
- suspend the process (*suspend*)
- resume the process

A List of Signals

Here's a list of the predefined signals, along with their respective macro definitions, numerical values, and default actions, as well as a brief description of each:

Macro # Default Description

SIGHUP 1 quit hang up
SIGINT 2 quit interrupt
SIGQUIT 3 dump quit
SIGILL 4 dump illegal instruction
SIGTRAP 5 dump trace trap(used by debuggers)
SIGABRT 6 dump abort
SIGEMT 7 dump emulator trap instruction
SIGFPE 8 dump arithmetic execution
SIGKILL 9 quit kill(cannot be caught, blocked, or ignored)
SIGBUS 10 dump bus error(bad format address)
SIGSEGV 11 dump segmentation violation(out-of-range address)
SIGSYS 12 dump bad argument to system call
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)
SIGUSR1 16 quit user signal 1
SIGUSR2 17 quit user signal 2
SIGCHLD 18 ignore child status changed
SIGPWR 19 ignore power fail or restart
SIGWINCH 20 ignore window size change
SIGURG 21 ignore urgent socket condition
SIGPOLL 22 ignore pollable event
SIGSTOP 23 quit stopped(signal)
SIGSTP 24 quit stopped(user)
SIGCONT 25 ignore continued
SIGTTIN 26 quit stopped(tty input)
SIGTTOU 27 quit stopped(tty output)
SIGVTALRM 28 quit virtual timer expired
SIGPROF 29 quit profiling timer expired
SIGXCPU 30 dump CPU time limit exceeded
SIGXFSZ 31 dump file size limit exceeded

Terminal Signals

The easiest way to send a signal to a foreground process is by pressing Control-C or Control-Z from the keyboard.

When the terminal driver (the piece of software that supports the terminal) recognizes that Control-C was pressed. 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

Requesting an Alarm Signal : alarm()

SIGALRM, by using "alarm()".

The default handler for this signal displays the message "Alarm clock" and terminates the process.

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, that alarm 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.

A small program that uses "alarm()", together with its output:

```
$ cat alarm.c ---> list the program.
#include <stdio.h>
main()
{
alarm(3); /* Schedule an alarm signal in three seconds */
printf("Looping forever… \n");
while(1)
printf("This line should never be executed \n");
}
$ alarm ---> run the program.
Looping forever…
Alarm clock ---> occurs three seconds later.
$ -
```

Handling Signals : 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 stored in "/usr/include/signal.h".

The signals SIGKILL and SIGSTP may not be reprogrammed. A child process inherits the signal settings from its parent during a "fork()".

"signal()" returns the previous *func* value associated with *sigCode* if successful; otherwise, it returns a value of -1.

The "signal()" system call may be used to override the default action.

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.

Sending Signals: 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 to do not terminate a process.

It's called "kill()" because of historical reasons; the main use of signals when UNIX was first designed 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 zero, 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 owned by a super-user, the signal is sent to all of the processes owned by the same owner as that of the sender, excluding the sending process.
- If the *pid* is negative, but not -1, the signal is sent to all of the processes in the process group.

Death of Children

When a parent’s child terminates, the child process sends its parent a SIGCHLD signal.

A parent process often installs a handler to deal with this signal; this handler typically executes a “wait()” system call to accept the child’s termination code and let the child dezombify.

The parent can choose to ignore SIGCHLD signals, in which case the child dezombifies automatically.

Suspending and Resuming Processes

The SIGSTOP and SIGCONT signals suspend and resume a process, respectively. They are used by the UNIX shells that support job control to implement built-in commands like *stop*.

Program 1:

//PROGRAM TO FIND THE NUMBER OF SIGNALS AVAILABLE ON YOUR SYSTEM; //IGNORE ALL OF THEM

```
#include<stdio.h>
#include<signal.h>
int main(void)
{
int i;
for(i = 1; i <= 32; i++)
If(signal(I,SIG_IGN)==SIG_ERR)
printf(“Signal %d can’t be ignored\n”,i);
}
```

Program 2:

/*PROGRAM TO HANDLE ALARM SIGNAL */

```
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0; /* Global alarm flag */
void alarmHandler(); /* Forward declaration of alarm handler */
main()
{
signal(SIGALRM,alarmHandler); /* Install signal handler */
alarm(3); /* Schedule an alarm signal in three seconds */
printf("Looping...\n");
while( !alarmFlag ) /* Loop until flag set */
{
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;
}
```

$ ./a.out ---> run the program.
Looping…
An alarm clock signal was received ---> occurs three seconds later.
Loop ends due to alarm signal
$ _

Program 3:

/* The program will print out Hello World forever until something interrupts it */

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
printf("OUCH! - I got signal %d\n", sig);
(void) signal(SIGINT, SIG_DFL);
}
int main()
{
(void) signal(SIGINT, ouch);
while(1) {
printf("Hello World!\n");
sleep(1);
}
}
```

Program 4:

/* Program waits for 5 seconds for user input and then generates SIGALRM that has a handler specified. */

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<stdlib.h>
#define BUFSIZE 100
void alrm_handler(int signo);
void quit(char *message,int exit_status);
char buf[BUFSIZE] = "foo\0";
int main()
{
int n;
if(signal(SIGALRM,alrm_handler)==SIG_ERR) /* signal returns SIG_ERR on
error */
quit("sigalrm",1);
fprintf(stderr,"Enter Filename : ");
/* sets alarm clock; will deliver SIGALRM in 5 seconds*/
alarm(5);
        /* Will come here if user inputs a string within 5 seconds*/
```

```
n = read(STDIN_FILENO,buf,BUFSIZE);
if ( n > 1)
printf("Filename = %s\n",buf);
exit(0);
}
/* Invoked with the process received SIGALRM */
void alrm_handler(int signo)
{
signal(SIGALRM,alrm_handler); /*Resetting signal handler*/
fprintf(stderr,"\nSignal %d received default filename : %s\n",signo,buf);
exit(1);
}
void quit(char *msg, int si)
{
printf("%s %d\n",msg, si);
}
```

Run this program twice with the file name and without filename and verify the correctness of signal number with kill –l output

```
$ ./a.out
Enter filename : f1.txt
Filename : f1.txt
$ ./a.out
Enter filename : Nothing enter in 5 second
Signal 14 received, default Filename : foo
$ kill –l | grep 14
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGUSR1
+++++++
```

Program 5:

/* Program to demostrate Kill system call along with fork() */
// Suspending and Resuming Processes

```
/* The main program created two children that both entered an infinite loop and
displayed a message every second.
The main program waited for three seconds and then suspended the first child.
After another three seconds, the parent restarted the first child, waited a little while
longer, and then terminated both children. */
#include <signal.h>
#include <stdio.h>
```

```
main()
{
        int pid1;
int pid2;
pid1 = fork();
if (pid1== 0) /* First child */
{
while(1) /* Infinite loop */
{
printf("pid1 is alive \n");
sleep(1);
}
}
pid2 = fork(); /* Second child */
if ( pid2 == 0 )
{
while(1) /* Infinite loop */
{
printf("pid2 is alive \n");
sleep(1);
}
}
sleep(3);
kill(pid1,SIGSTOP); /* Suspend first child */
sleep(3);

kill(pid1,SIGCONT); /* Resume first child */
sleep(3);
kill(pid1,SIGINT); /* Kill first child */
kill(pid2,SIGINT); /* Kill second child */
}
$ ./a.out ---> run the program.
pid1 is alive ---> both run in first three seconds.
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
pid2 is alive ---> just the second child runs now.
pid2 is alive
```

pid2 is alive
pid1 is alive ---> the first child is resumed.
pid2 is alive
pid1 is alive
pid2 is alive
pid1 is alive
pid2 is alive
++++++

Program 6:
/* **Protecting Critical Code and Chaining Interrupt Handlers** */

/* To protect critical pieces of code against Control-C attacks and other such signals.
- it can be restored after the critical code has executed. Here's the source code of a program that protects itself against SIGINT signals: */

```
#include <stdio.h>
#include <signal.h>
main()
{
        void (*oldHandler) (); /* To hold old handler value */
        printf("I can be Control-C.ed \n");
        sleep(3);
        oldHandler = signal(SIGINT, SIG_IGN); /* Ignore Control-C */
        printf("Im protected from Control-C now\n");
sleep(3);
signal(SIGINT, oldHandler); /* Restore old handler */
printf("I can be Control-Ced again \n");
sleep(3);
printf("Bye! \n");
}
```

$ ./a.out ---> run the program.
I can be Control-C'ed
^C ---> Control-C works here.
$ ./a.out ---> run the program again.
I can be Control-C'ed
I'm protected from Control-C now
^C
I can be Control-C'ed again
Bye!