Signals and signal processing

Suntae Hwang

Kookmin University

Introduction

- Signals are software interrupts
- Signals provide a way of handling *asynchronous* events
- Every signal has a name
  - Begin with the three characters SIG
  - These name are all defined by positive integer constants (the signal number) in the header <signal.H>
- Version 7 had 15 different signals
  - **Unreliable signal model** - get lost and hard to turn off.
- SVR4 and 4.3+BSD both have 31 different signals
  - **Reliable signals added**.

Signal concepts

- □ Numerous conditions can generate a signal
  - – The terminal-generated signals occur when user press certain terminal key such as DELETE
  - – Hardware exceptions generate signals
    - ◆ divide by 0, invalid memory reference and the like
  - – The kill(2) function allows a process to send any signal to another process or process group
    - ◆ need to be owner of the target process or we have to be a superuser
  - – The kill(1) command to send signal to other processes
    - ◆ this program is just an interface to the kill function
  - – Software conditions can generate signals
    - ◆ SIGALRM, SIGPIPE (Broken pipe), SIGURG (Out-of-band data)

Dispositions of signals

Disposition or action:

Process has to tell the kernel "if and when this signal occurs, do the following."

□Ignore the signal

- This works for most signals, but SIGKILL and SIGSTOP can never be ignored.

□Catch the signal

- To do this we tell the kernel to call a function of ours whenever the signal occurs

□Let the default action apply

- Every signal has a default action which is to terminate the process in most cases

Unix signals (ANSI, POSIX.1, SVR4, 4.3+BSD)

SIGABRT	abnormal termination(abort)
SIGALRM	time out (alarm)
SIGBUS	hardware fault
SIGCHLD	change in status of a child sent
SIGCONT	continue stopped process
SIGEMT	hardware fault
SIGFPE	arithmetic exception
SIGHUP	hangup
SIGILL	illegal hardware instruction
SIGINFO	status request from keyboard
SIGINT	terminal interrupt character
SIGIO	asynchronous I/O
SIGIOT	hardware fault
SIGKILL	termination
SIGPIPE	write to pipe with no readers
SIGPOLL	pollable event (poll)
SIGPROF	profiling time alarm (setitimer)

SIGPWR	power fail / restart
SIGQUIT	terminal quit character
SIGSEGV	invalid memory reference
SIGSTOP	stop
SIGSYS	invalid system call
SIGTERM	termination
SIGTRAP	hardware fault
SIGTSTP	terminal stop character
SIGTTIN	background read from control tty
SIGTTOU	background write to control tty
SIGURG	urgent condition
SIGUSR1	user-defined signal
SIGUSR2	user-defined signal
SIGVTALRM	virtual time alarm (setitimer)
SIGWINCH	terminal window size change
SIGXCPU	CPU limit exceeded
SIGXFSZ	file size limit exceeded

Signals

- □ SIGART: generated by calling the `abort` function.
- □ SIGALRM: generated when a timer set with the `alarm` expires.
- □ SIGCHLD : Whenever a process terminates or stops, the signal is sent to the parent.
- □ SIGCONT : This signal(job-control) sent to a stopped process when it is continued.
- □ SIGFPE : signals an arithmetic exception, such as divide-by-0, floating point overflow, and so on
- □ SIGHUP :
  - – generated to the controlling process (session leader) associated with a controlling terminal if a disconnect is detected by the terminal interface
  - – generated if the session leader terminates and sent to each process in the foreground process group
  - – commonly used to notify daemon process to reread their configuration files (note that a daemon should not have a controlling terminal and normally never receive this signal)

Signals (cont’d)

- SIGILL : indicates that the process has executed an illegal hardware instruction.
- SIGINT : generated by the terminal driver when we type the interrupt key and sent to all processes in the foreground process group
- SIGIO : indicates an asynchronous I/O event
- SIGKILL : can’t be caught or ignored. a sure way to kill any process.
- SIGPIPE : If we write to a pipeline but the reader has terminated, SIGPIPE is generated
- SIGPWR : related to power failure. (read the book for the detail)
- SIGQUIT : generated by the terminal driver when we type terminal quit key and sent to all processes in the foreground process group

Signals (cont'd)

- SIGSEGV : indicates that ther process has made an invalid memory reference
- SIGSTOP : This signal(job-control) stops a process and can't be caught or ignored
- SIGSYS : signals an invalid system call
- SIGTERM : the termination signal sent by the `kill(1)` command by default.
- SIGTSTP : This is the interactive stop signal generated by the terminal driver when we type the terminal suspend key and sent to all processes in the foreground process group.
- SIGTTIN : generated by the terminal driver when a process in a background process group tries to read from its controlling terminal
- SIGTTOU : generated by the terminal driver when a process in a background process group tries to write to its controlling terminal

Signals (cont'd)

- □SIGURG : notifies the process that an urgent condition has occurred. Optionally generated when out-of-band data is received on a network connection.
- □SIGUSR1[2] : user-defined signals, for use in application programs
- □SIGWINCH : generated to the foreground process group when a process changes the window size from its previous value, with the `ioctl` set-window-size command
- □SIGXCPU : generated if the process exceeds its soft CPU time limit
- □SIGXFSZ : generated if the process exceeds its soft file size limit

Signal Function

```
#include <signal.h>

void ( *signal ( int signo, void (*func) (int))) (int)
```

Returns: previous disposition of signal if OK, SIG_ERR on error

- The simplest interface to the signal features of Unix
  - *signo* : the name of the signal
  - *func:*
    - SIG_IGN - ignore the signal
    - SIG_DFL - take its default action
    - The address of a signal handler (or signal-catching function): a function to be called (catching) when the signal occurs.
  - The signal handler is passed a single integer argument (*the signal number*) and returns nothing.
  - signal() returns the pointer to the previous signal handler

```
typedef void Sigfunc(int);
Sigfunc *signal(int, Sigfunc *);
```

Signal Function Example

```c
static void sig_child(int);

int main(void) {
  pid_t pid; int i;
  signal(SIGCHLD, sig_child);

  pid = fork();
  if (pid == 0) {
    sleep(1);
    exit(0);
  }
  while(1) { i = i; }
}

static void
sig_child(int signo) {
  pid_t pid; int status;

  pid = wait(&status);

  printf("child %d finished\n", pid);
}
```

```
$ a.out
child 17145 finished
```

```c
static void sig_fpe(int);

int main(void) {
  pid_t pid; int i;
  signal(SIGFPE, sig_fpe);

  i = i/0;
}

static void
sig_fpe(int signo) {
  pid_t pid; int status;
  printf("Divide by 0 Error\n");

/* routine that saves all variables*/

  exit(1);
}
```

```
$ a.out
Floating point exception
```

```
$ a.out
Divide by 0 Error
```

Signal Function Example

```
#include        <signal.h>
static void     sig_usr(int);   /* one handler for both signals */
int main(void){
        if (signal(SIGUSR1, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR1");
        if (signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("can't catch SIGUSR2");

        for ( ; ; ) pause();
}

static void
sig_usr(int signo) {             /* argument is signal number */
        if (signo == SIGUSR1)
                printf("received SIGUSR1\n");
        else if (signo == SIGUSR2)
                printf("received SIGUSR2\n");
        else err_dump("received signal %d\n", signo);
        return;
}
```

```
$ a.out &
[1] 4720
$ kill -USR1 4720   send it SIGUSR1
received SIGUSR1
$ kill -USR2 4720   send it SIGUSR2
received SIGUSR2
```

```
$kill 4720    send it SIGTERM
[1] + Terminated a.out &
```

Program Start-up

- When a process is forked, the child inherits the parent’s signal dispositions.
- When a program is *execed*
  - the disposition of any signals that are being caught to their default action
  - the status of all other signals (ignored or default) is left alone
- An interactive shell (w/o job control)
  - sets the disposition of the interrupt and quit signals in the background process to be ignored
  - Many interactive programs catches the signals only when not in the background (the signal is not ignored) by doing the following:

```
int sig_int(), sig_quit()

if (signal(SIGINT, SIG_IGN) != SIG_IGN) signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN) signal(SIGQUIT, sig_quit);
```

Interrupted System Calls (1/2)

- *Slow* system calls : that can block forever
  - reads from/writes to files that can block the caller forever (pipes, terminal, network)
  - open files that block until some condition occurs (opening terminal devices that waits until a modem answers the phone)
  - pause() and wait()
  - certain `ioctl`() operations and some IPC functions
- A *slow* system call is interrupted by a signal
  - returns an error and errno was set to EINTR
  - need to handle the error explicitly

```
Again:
if ((n = read(fd, buff, BUFFSIZE)) < 0) {
        if (errno == EINTR) go to Again; /* interrupted system call */
}
```

Interrupted System Calls (2/2)

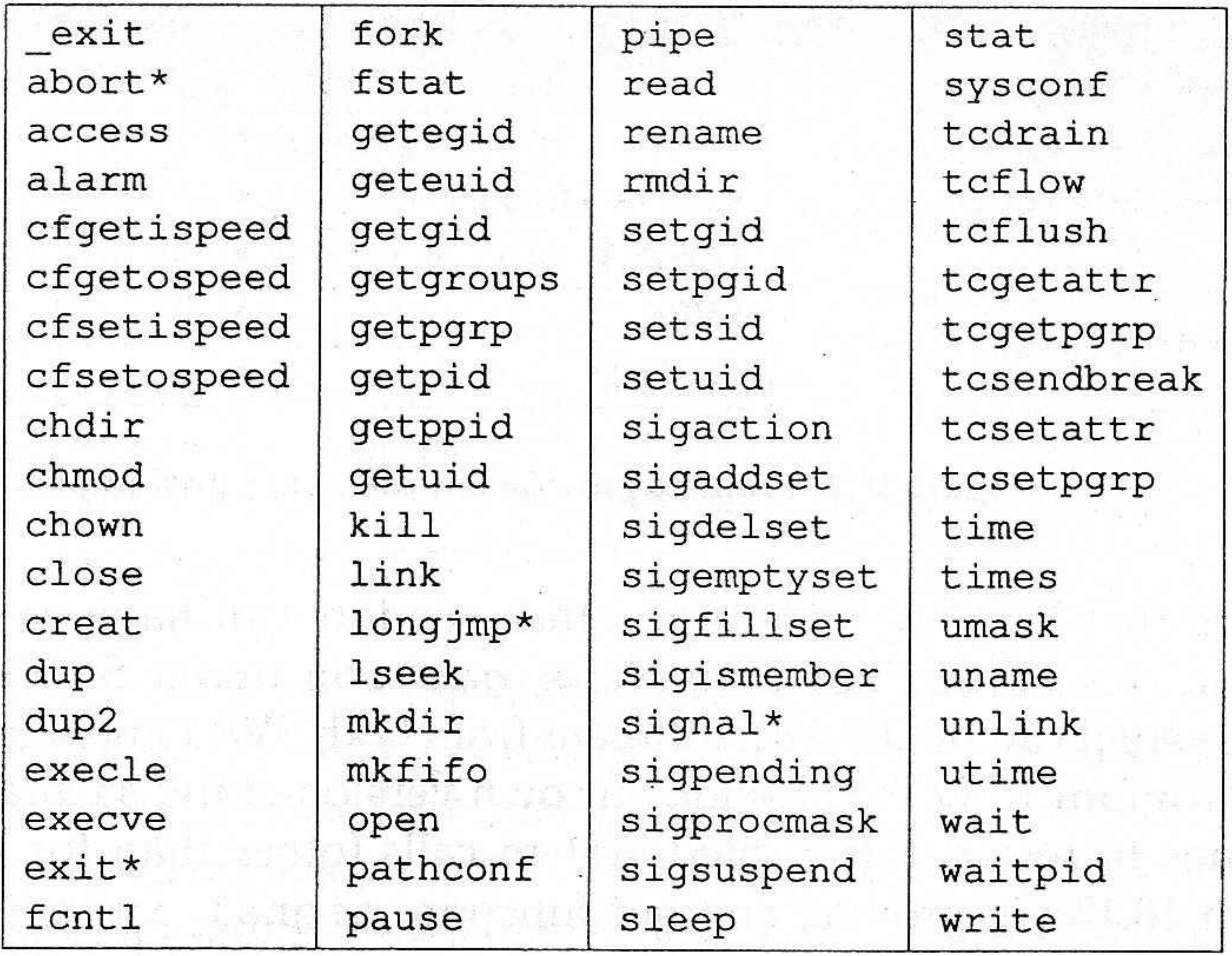
- □Automatic restarting of certain interrupted system calls (4.2BSD)
  - – ioctl, read, readv, write, writev, wait and waitpid

    (wait, waitpid are always interrupted when a signal is caught)
  - – 4.3BSD allow to disable this feature on a per-signal basis
  - – Without the automatic restart feature, we need to test every read/write for the interrupted error return and reissue the read or write.
- □Fast system calls completes before the signal was delivered

Reenturant Functions

- POSIX.1 specifies the functions that are guaranteed to be reentrant
- Calling a nonreentrant function from a signal handler may produce unpredictable results
  - While the main program calls malloc() and interrupted, the signal handler also calls malloc(), then what could happen?
- One errno variable per process even with reentrant guaranteed functions - save the errno and restore it later.

Reentrant functions that may be called from a signal handler



- _exit
- abort*
- access
- alarm
- cfgetispeed
- cfgetospeed
- cfsetispeed
- cfsetospeed
- chdir
- chmod
- chown
- close
- creat
- dup
- dup2
- execle
- execve
- exit*
- fcntl
- fork
- fstat
- getegid
- geteuid
- getgid
- getgroups
- getpgrp
- getpid
- getppid
- getuid
- kill
- link
- longjmp*
- lseek
- mkdir
- mkfifo
- open
- pathconf
- pause
- pipe
- read
- rename
- rmdir
- setgid
- setpgid
- setsid
- setuid
- sigaction
- sigaddset
- sigdelset
- sigemptyset
- sigfillset
- sigismember
- signal*
- sigpending
- sigprocmask
- sigsuspend
- sleep
- stat
- sysconf
- tcdrain
- tcflow
- tcflush
- tcgetattr
- tcgetpgrp
- tcsendbreak
- tcsetattr
- tcsetpgrp
- time
- times
- umask
- uname
- unlink
- utime
- wait
- waitpid
- write

Reenturant Functions (cont’d)

```
err_sys(char *s) { fprintf(stderr,"%s",s); exit(1);}
static void my_alarm(int);

int main(void) {
   struct passwd  *ptr;
   signal(SIGALRM, my_alarm);  alarm(1);
   for ( ; ; ) {
      if ( (ptr = getpwnam(“sthwang")) == NULL) err_sys("getpwnam error");
         if (strcmp(ptr->pw_name, “sthwang") != 0)
         printf("return value corrupted!, pw_name = %s\n", ptr->pw_name);
   }
}
static void my_alarm(int signo) {
   struct passwd  *rootptr;

   printf("in signal handler\n");
   if ( (rootptr = getpwnam("root")) == NULL)
         err_sys("getpwnam(root) error");
   alarm(1);
   return;
}
```

```
$ a.out
in signal handler
Segmentation fault
$ a.out
in signal handler
in signal handler
Segmentation fault
$ a.out
in signal handler
getpwnam(root) error
```

Kill and Raise function (1/2)

```
#include <sys/types.h>
#include <singnal.h>
int     kill(pid_t pid, int signo);
int     raise(int signo);
        Both  return: 0 if OK, 1 on error
```

□The kill function sends a signal to a process or a group of process

- pid > 0 signal to the process whose process ID is pid
- pid == 0 signal to the processes whose process group ID equals that of sender
- pid < 0 signal to the processes whose process group ID equals abs. of pid
- pid == -1 POSIX.1 leaves this condition unspecified (used as a broadcast signal in SVR4, 4.3+BSD)

□The raise function allows a process to send a signal to itself

Kill and Raise function (2/2)

- A process needs permission to send a signal to some other process
  - The superuser can send a signal to any process
  - The real or effective user ID of the sender has to equal the real or effective user ID of the receiver
  - SIGCONT can be sent to any member process of the same session
  - signo = 0: *null signal*
    - normal error checking performed, but no signal is sent
    - used often to determine if a specific process still exists. (If the process doesn't exist, `kill` returns –1 and `errno` is set to `ESRCH`).

alarm and pause function (1/2)

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds) ;
```

Returns: 0 or number of seconds until previously set alarm

□Alarm function

- sets a timer that will expire at a specified time in the future
- When the timer expires, the SIGALRM signal generated
- *seconds* is the number of clock seconds in the future when the signal should be generate
- default action of the signal is to terminate the process.
- There could be a extra delay between when the signal generated and when the signal handler gets the control
- only one alarm clock per process
  - ◆ previously registered alarm clock is replaced by the new value
  - ◆ if *seconds*=0, the previous alarm clock is cancelled

alarm and pause function (2/2)

```
#include <unistd.h>

int pause (void) ;
```

Returns: -1 with errno set to EINTR

□Pause function

- suspends the calling process until a signal is caught.
- returns only if a signal handler is executed and that handler returns.
- returns -1 with errno set to EINTR

Example I (sleep1)

```
static void
sig_alrm(int signo)
{
          return;   /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)    /* sleep the process for nsecs */
{
          if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                    return(nsecs);
          alarm(nsecs);                 /* start the timer */
          pause();                      /* next caught signal wakes us up */
          return( alarm(0) ); /* turn off timer, return unslept time */
}
```

- If the caller of sleep1() already has an alarm set, the alarm is erased by the first call to alarm.
  - Save remaining alarm time and reset the alarm before the return
- Modify the disposition for SIGALRM
  - Save the disposition and reset before the return
- Race condition: alarm may goes off before the pause(); the caller is suspended forever at `pause()` => sigpromask, sigsuspend

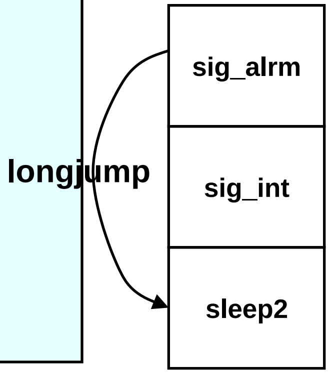
Example II (sleep2)

```
static jmp_buf      env_alrm;

static void
sig_alrm(int signo)
{
          longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
          if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                    return(nsecs);
          if (setjmp(env_alrm) == 0) {
                    alarm(nsecs);       /* start the timer */
                    pause();            /* next caught signal wakes us up */
          }
          return( alarm(0) );           /* turn off timer, return unslept time */
}
```

- The previous race condition was avoided
- Another problem if SIGALRM interrupts some other signal handler and the longjmp() aborts the other signal handler (see the next example)

Example III (sleep2 problem)

```
Int main(void){
        unsigned int        unslept;
        if (signal(SIGINT, sig_int) == SIG_ERR)
                err_sys("signal(SIGINT) error");
        unslept = sleep2(5);
        printf("sleep2 returned: %u\n", unslept);
        exit(0);
}
static void
sig_int(int signo){ /* the for loop executes more than 5 sec */
        int                 i;
        volatile int        j;
        printf("\nsig_int starting\n");
        for (i = 0; i < 2000000; i++)  j += i * i;
        printf("sig_int finished\n");
        return;
}
```



```
$ a.out
                        sleep2 starts running
^?                      Type our interrupt char
sig_int starting        SIGALRM generated while in sig_int()
sleep2 returned: 0      longjmp aborted sig_int
```

Example IV (timeout)

```
Int main(void){
          int n; char line[MAXLINE];
          if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                    err_sys("signal(SIGALRM) error");
          alarm(10);
          if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
                    err_sys("read error");
          alarm(0);

          write(STDOUT_FILENO, line, n);
          exit(0);
}
static void
sig_alrm(int signo){
          return;   /* nothing to do, just return to interrupt the read */
}
```

- A common use for alarm : timeout function
- Race condition: alarm may go off before read()
- If the read system call is automatically restarted, timeout does not work.

Example V (Another timeout)

```
static jmp_buf          env_alrm;
int
main(void){
          int n;char line[MAXLINE];
          if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                    err_sys("signal(SIGALRM) error");
          if (setjmp(env_alrm) != 0)
                    err_quit("read timeout");
          alarm(10);
          if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
                    err_sys("read error");
          alarm(0);
          write(STDOUT_FILENO, line, n);
          exit(0);
}
static void
sig_alrm(int signo)
{
          longjmp(env_alrm, 1);
}
```

- □No problems with automatic restart
- □But still has the race condition and the problem with other signal handler interactions

Abort Function

```
#include <stdlib.h>
void    abort(void);
```

This function never returns

- Causes *abnormal* program termination
- This function sends the SIGABRT signal to the process
- SIGABRT signal handler to perform any cleanup that it wants to do, before the process terminated
- POSIX.1 states that if the process does not terminate itself from this signal handler, when signal handler returns, `abort` *terminates the process.*

Sleep Function

```
#include <signal.h>
unsigned int sleep(unsigned int seconds) ;
```

Returns: 0 or number of unslept seconds

- This function causes the calling process to be suspended until either
  - The amount of wall clock time specified by second has elapsed
    - The return value is 0
  - A signal is caught by the process and the signal handler returns
    - The return value is the number of unslept seconds
  - The actual return may be at a time later than requested, because of other system activity
  - There can be interactions between sleep and alarm if sleep is implemented with the alarm functions (unspecified by POSIX.1)