

Signals

- Notifies a process of an event
- *Generated* when event that causes signal occurs
- *Delivered* when the process takes action based on the signal
- A signal is *pending* if it has been *generated* but is not yet *delivered*.
- The *lifetime* of a signal is the interval between *generation* and *delivery*.

Signal Handling

- A process *catches* a signal if it executes a *signal handler* when the signal is delivered
- A program installs a *signal handler* by making a call to the *sigaction* system call

Signal Handlers

The `sigaction` system call installs either:

- A user defined function
- `SIG_DFL` – a default routine
- `SIG_IGN` – ignore the signal

Masking Signals

- The action taken on receipt of a signal depends on the current signal handler for that signal and the signal mask for the process
- The mask identifies the currently blocked signals
- You change a signal mask with the *sigprocmask* system call
- blocking is different from ignoring – you ignore a signal by installing SIG_IGN with *sigaction*

POSIX Required Signals

Symbol	Meaning
SIGABRT	Abnormal termination as initiated by abort
SIGBUS	Access undefined part of memory object
SIGALRM	Timeout signal as initiated by alarm
SIGFPE	Error in arithmetic operation as in division by zero
SIGHUP	Hang up (deat) on controlling terminal (process)
SIGILL	Invalid hardware instruction
SIGINT	Interactive attention signal
SIGKILL	Terminate (cannot be caught or ignored)
SIGPIPE	Write on a pipe with no readers
SIGQUIT	Interactive termination
SIGSEGV	Invalid memory reference
SITTERM	Termination
SIGURG	High bandwidth data available at a socket
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

POSIX Job Control Signals

Symbol	Meaning
SIGCHLD	Indicates child process terminated or stopped
SIGCONT	Continue if stopped (done when generated)
SIGSTOP	Stop signal (cannot be caught or ignored)
SIGTSTP	Interactive stop signal
SIGTTIN	Background process attempts to read from controlling terminal
SIGTTOU	Background process attempts to write to controlling terminal

Generation of Signals

- Some signals are generated with errors occur (i.e., SIGFPE or SIGEGV)
- A user can send signals to a process that it owns (the effective user ID of the sending and receiving processes are the same)
- Signals can be generated from the shell with a kill system call

kill Command

SYNOPSIS

kill -s signal_name pid...

kill -l [exit status]

kill [-signal_name] pid...

kill [-signal number] pid...

POSIX: *Shell and Utilities*

Signal_Name Example

kill -USR1 3423 or kill -s USR1 3423

- Sends SIGUSR1 to process 3423
- USR1 is a symbolic name for the signal SIGUSR1

Signal_Number Example

```
kill -9 3423
```

- Sends signal number 9 (SIGKILL) to process 3423
- The only supported signal number values are:
 - 0 for signal 0
 - 1 for signal SIGHUP
 - 2 for signal SIGINT
 - 3 for signal SIGQUIT
 - 6 for signal SIGABRT
 - 9 for signal SIGKILL
 - 14 for signal SIGALRM
 - 15 for signal SIGTERM

Signal Symbolic Names

Typing `kill -l` on the ect-unix machines causes the following to be output:

```
EXIT HUP INT QUIT ILL TRAP ABRT EMT  
FPE KILL BUS SEGV SYS PIPE ALRM  
TERM USR1 USR2 CLD PWR WINCH  
URG POLL STOP TSTP CONT TTIN  
TTOU VTALRM PROF XCPU XFSZ  
WAITING LWP FREEZE THAW CANCEL  
LOST RTMIN RTMIN+1 RTMIN+2  
RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1  
RTMAX
```

kill System Call

SYNOPSIS

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

```
int kill(pid_t pid, int sig);
```

POSIX: CX

errno for kill System Call

errno	cause
EINVAL	sig is an invalid or unsupported signal
EPERM	caller does not have the appropriate privileges
ESRCH	no process or process group corresponds to pid

kill System Call Example

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

```
#include <sys/types.h>
```

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

```
if (kill(3423, SIGUSR1) == -1)  
    perror("Could not send SIGUSR1");
```

```
if (kill (getppid(), SIGTERM) == -1)  
    perror("Error in kill");
```

raise System Call

SYNOPSIS

```
#include <signal.h>  
int raise(int sig);
```

POSIX: CX

A process can send a signal to itself with a raise system call – Example:

```
if (raise (SIGUSR1) != 0)  
    perror (“Failed to raise SIGUSR1”);
```

stty -a

Among other things, the `stty -a` output associates signals with control keystroke inputs

Output of stty -a on ect-unix Machine

```
speed 9600 baud; rows 24; columns 80;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D;
eol = <undef>; eol2 = <undef>; swtch = <undef>; start
= ^Q; stop = ^S; susp = ^Z; dsusp = ^Y; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O;
-parenb parodd cs8 hupcl cstopb cread -clocal crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr
icrnl ixon ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0
tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase
-tostop -echoprt
echoctl echoke
```

Signal Generating Characters

- The INTR character, ctrl-c, generates SIGINT for the foreground process
- The QUIT character, ctrl-|, generates SIGQUIT
- The SUSP character, ctrl-z, generates SIGSTOP
- The DSUP character, ctrl-y, generates SIGCONT

alarm System Call

SYNOPSIS

```
include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```

Posix.1 Spec 1170

Causes SIGALRM signal to be sent to calling process after the specified number of seconds have elapsed –
Example:

```
#include <unistd.h>
void main(void) {
    alarm(10);
    for( ; ; ) { } }
```

Signal Mask and Signal Sets

- A process can temporarily prevent a signal from being delivered by blocking it
- The process signal mask contains the set of signals that are currently blocked
- When a process blocks a signal, an occurrence of the signal is held until the signal is unblocked (blocked signals do not get lost – ignored signals do get lost)

Signal Set Functions

- sigemptyset – remove all signals from the signal set
- sigfillset – initializes a signal set to contain all signals
- sigaddset – adds a specified signal to a signal set
- sigdelset – removes a specified signal from a signal set
- sigismember – tests to see if a specified signal is in a signal set

sigprocmask System Call

SYNOPSIS

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

```
int sigprocmask(int how, const sigset_t *set, sigset_t  
    *oset);
```

Posix.1, Spec 1170

sigprocmask – how parameter

- SIG_BLOCK – add the collection of signals in the parameter *set* to those already in the signal set
- SIG_UNBLOCK – delete the collection of signals in the parameter *set* from the signal set
- SIG_SETMASK – replace the entire current signal set with the collection of signals in the parameter *set*

sigprocmask – Example 1

```
#include <stdio.h>
#include <signal.h>
sigset_t newsigset;
sigemptyset(&newsigset); /* newsigset is empty */
sigaddset(&newsigset, SIGINT); /* add SIGINT to
                                newsigset */
if (sigprocmask (SIG_BLOCK, &newsigset, NULL) < 0)
    perror("Could not block the signal");
```

The above code causes the signal SIGINT to be blocked.

sigprocmask – Example 2

...

```
sigemptyset (&intmask);
```

```
sigaddset (&intmask, SIGINT);
```

```
sigprocmask(SIG_BLOCK, &intmask, NULL);
```

...

```
    /* if ctrl-c is typed while this code is executing,  
       the SIGINT signal is blocked */
```

...

```
sigprocmask(SIG_UNBLOCK, &intmask, NULL);
```

...

```
    /* a ctrl-c typed while either this OR THE ABOVE  
       code is excuting is handled here */
```

...

sigprocmask – Example 3

...

```
sigfillset(&blockmask);
```

```
sigprocmask(SIG_SETMASK, &blockmask, &oldset);
```

...

```
    /* all signals are blocked here */
```

...

```
sigprocmask(SIG_SETMASK, &oldset, NULL);
```

...

```
    /* the old signal set goes back into effect here */
```

...

sigaction System Call

SYNOPSIS

```
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
```

```
struct sigaction {  
    void (*sa_handler)(); /* SIG_DFL, SIG_IGN, or pointer to function */  
    sigset_t sa_mask      /* additional signals to be blocked during  
                           execution of handler */  
    int sa_flags          /* special flags and options */  
};
```

The `sigaction` system call installs signal handlers for a process. The data structure *struct sigaction* holds the handler information

sigaction – Example 1

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
char handmsg[] = "I found ^c\n";

void mysighand(int signo)
{
    write (STDERR_FILENO, handmsg, strlen(handmsg)); }
...
struct sigaction newact;
...
newact.sa_handler = mysighand; /* set the new handler */
sigemptyset(&newact.sa_mask); /* no other signals blocked */
newact.sa_flags = 0;          /* no special options */
if (sigaction(SIGINT, &newact, NULL) == -1)
    perror("Could not install SIGINT signal handler");
```

Installs *mysighand* signal handler for SIGINT

sigaction – Example 2

```
struct sigaction act;
struct sigaction oact;

/* Download old signal handler */
if (sigaction(SIGINT, NULL, &oact) == -1)
    perror("Could not get old handler for SIGINT");
else if (oact.sa_handler == SIG_DFL) { /* ignore SIGINT */
    act.sa_handler = SIG_IGN; /* set new handler to ignore */
    if (sigaction(SIGINT, &act, NULL) == -1)
        perror("Could not ignore SIGINT");
    else {
        printf("Now ignoring SIGINT for 10 seconds\n");
        sleep(10);
        if (sigaction(SIGINT, &oact, NULL) == -1)
            perror("Could not restore old handler for SIGINT");
        printf("Old signal handler restored\n");
    }
}
```

This code downloads the current sigaction structure, checks to see if the current handler is the default handler, and if so, ignores SIGINT for 10 seconds

Set Handler to Default

```
struct sigaction newact;
```

```
newact.sa_handler = SIG_DFL; /* new handler set to default */  
sigemptyset(&newact.sa_mask); /* no other signals blocked */  
newact.sa_flags = 0; /* no special options */  
if (sigaction(SIGINT, &newact, NULL) == -1)  
    perror("Could not set SIGINT handler to default action");
```

pause System Call

SYNOPSIS

```
#include<unistd.h>  
int pause (void);
```

POSIX

Pause suspends the calling process until a signal that is not being blocked or ignored is delivered to the process. It does not allow resetting of the signal mask.

Pause – Example 1

```
#include <unistd.h>
int signal_received = 0; /* external static variable */
...
while(signal_received == 0)
    pause();
```

If the signal is delivered in between the test of `signal_received` and `pause`, the `pause` will not return until another signal is delivered to the process.

pause – Example 2

...

```
int signal_received = 0; /* external static variable */
```

...

```
sigset_t sigset;
```

```
int signum;
```

...

```
sigemptyset(&sigset);
```

```
sigaddset(&sigset, signum);
```

```
sigprocmask(SIG_BLOCK, &sigset, NULL);
```

```
While(signal_received == 0)
```

```
    pause();
```

This does not solve the Example 1 pause problem because pause is executed while signum is blocked.

sigsuspend System Call

SYNOPSIS

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

```
int sigsuspend(const sigset_t *sigmask)
```

POSIX:CX

Solves the problems encountered with pause by temporarily installing sigmask when sigsuspend is called.

sigsuspend – Example 1

```
int signum = SIGUSR1;  
sigfillset (&sigmost);  
sigdelset(&sigmost, signum);  
sigsuspend(&sigmost);
```

If SIGUSR1 is delivered before sigsuspend, the process still suspends and deadlocks until another SIGUSR1 is delivered

sigsuspend – Example 2

```
static volatile sig_atomic_t sigreceived = 0
sigset_t maskall, maskmost, maskold
int signum = SIGUSR1;
sigfillset(&maskall);
sigfillset(&maskmost);
sigdelset(&maskmost, signum);
sigprocmask(SIG_SETMASK, &maskall, &maskold);
if (sigreceived == 0)
    sigsuspend(&maskmost);
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

Assume that sigreceived is set to 1 at the end of an interrupt handling routine called on SIGUSR1

sigsuspend – Example 3

```
static volatile sig_atomic_t sigreceived = 0; /* external  
static variable */
```

```
...
```

```
sigset_t maskblocked, maskold, maskunblocked;
```

```
int signum = SIGUSR1;
```

```
...
```

```
sigprocmask(SIG_SETMAK, NULL, &maskblocked);
```

```
sigprocmask(SIG_SETMASK, NULL, &maskunblocked);
```

```
sigaddset(&maskblocked, signum);
```

```
sigdelset(&maskunblocked, signum);
```

```
sigprocmask(SIG_BLOCK, &maskblocked, maskold);
```

```
while(sigreceived == 0)
```

```
    sigsuspend(&maskunblocked);
```

```
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

sigsuspend – Example 4

```
static volatile sig_atomic_t sigreceived = 0; /* external  
static variable */
```

```
...
```

```
sigset_t masknew, maskold;  
int signum = SIGUSR1;  
sigprocmask(SIG_SETMAK, NULL, &masknew);  
sigaddset(&masknew, signum);  
sigprocmask(SIG_SETMASK, &masknew, maskold);  
sigdelset(&masknew, signum);  
while(sigreceived == 0)  
    sigsuspend(&masknew);  
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

sigsuspend – Example 5

```
int simplesuspend(void) {  
    ...  
    if ((sigprocmask(SIG_SETMASK, NULL, &maskblocked) == -1) ||  
        (sigaddset(&maskblocked, signum) == -1) ||  
        (sigprocmask(SIG_SETMASK, NULL, &maskunblocked) == -1) ||  
        (sigdelset(&maskunblocked, signum) == -1) ||  
        (sigprocmask(SIG_SETMASK, &maskblocked, &maskold) == -1))  
        return -1;  
    while(sigreceived == 0)  
        sigsuspend(&maskunblocked);  
    sigreceived = 0  
    return sigprocmask(SIG_SETMASK, &maskold, NULL); }
```

sigwait

SYNOPSIS

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

```
int sigwait(const sigset_t *restrict sigmask, int *restrict signo);
```

POSIX:CX

- Blocks until any signal specified by sigmask is pending
- Then it removes that signal from the set of pending signals and unblocks
- The number of the signal removed from pending signals is stored at location signo

sigwait – Counts SIGUSR1 Occurrences

```
int main(void) {
    int signalcount = 0;
    int signo;
    int signum = SIGUSR1;
    sigset_t sigset;
    if((sigemptyset(&sigset) == -1) ||
        (sigaddset(&sigset,signum) == -1) ||
        (sigprocmask(SIG_BLOCK, &sigset,NULL) == -1))
        perror("Failed to block signals before sigwait");
    fprintf(stderr,"This process has ID %ld\n",(long)getpid());
    for(;;) { if(sigwait(&sigset,&signo)== -1) {
                perror("Failed to wait using sigwait");
                return 1; }
            signalcount++;
            fprintf(stderr,"Number of signals so far: %d\n", signalcount); }
}
```

Biff get_file_size

```
/* Return the size of file if no error.  Otherwise if  
 * file does not exist return 0 otherwise return -1.  
 */
```

```
long get_file_size(const char *filename)  
{  
    struct stat buf;  
  
    if (stat(filename, &buf) == -1) {  
        if (errno == ENOENT) return 0;  
        else return -1;  
    }  
    return (long)buf.st_size;  
}
```

```
/* Notify the user that mail has arrived */  
void send_mail_notification()  
{  
    fprintf(stderr, "Mail has arrived\007\n");  
}
```

Biff Interrupt Handlers

```
/* Turn on mail notification */  
void turn_notify_on(int s)  
{  
    notifyflag = 1;  
}
```

```
/* Turn off mail notification */  
void turn_notify_off(int s)  
{  
    notifyflag = 0;  
}
```

Biff notify_of_mail

```
/* Continuously check for mail and notify user of new mail */
int notify_of_mail(const char *filename)
{ ...
  sigemptyset(&emptyset);
  sigemptyset(&blockset);
  sigaddset(&blockset, SIGUSR1);
  sigaddset(&blockset, SIGUSR2);
  get mail;
  if mail is empty return(1) else call send_mail_notification;
  sleep(SLEEPTIME);
  for( ; ; ) {
    if (sigprocmask(SIG_BLOCK, &blockset, NULL) < 0)
      return 1;
    while (notifyflag == 0)
      sigsuspend(&emptyset);
    if (sigprocmask(SIG_SETMASK, &emptyset, NULL) < 0)
      return 1;
    compare new mail with old mail;
    if new mail is larger call send_mail_notification;
    sleep(SLEEPTIME);
  } }
```

Biff main()

```
void main(void)
{
    ...
    newact.sa_handler = turn_notify_on;
    newact.sa_flags = 0;
    sigemptyset(&newact.sa_mask);
    sigaddset(&newact.sa_mask, SIGUSR1);
    sigaddset(&newact.sa_mask, SIGUSR2);
    if (sigaction(SIGUSR1, &newact, NULL) < 0)
        perror("Could not set signal to turn on notification");
    newact.sa_handler = turn_notify_off;
    if (sigaction(SIGUSR2, &newact, NULL) < 0)
        perror("Could not set signal to turn off notification");
    notify_of_mail(mailfile);
    fprintf(stderr, "Fatal error, terminating program\n");
    exit(1);
}
```

Interrupted read

```
...
while (retval = read(fd, buf, size),
       retval == -1 && errno == EINTR)
    ;
if (retval == -1) {
    /* handle errors here */
}
```

Notice the use of the comma expression.

Timeout read from pipe

...

```
alarm(10);
if (read(fd, buf, size) < 0) {
    if (errno == EINTR)
        fprintf(stderr, "Timeout occurred\n");
        /* handle timeout here */
    else
        fprintf(stderr, "An error occurred\n");
        /* handle other errors here */
}
alarm(0);
```

Alarm signal is assumed to be caught and does NOT terminate the program.

Async-Signal Safe

- A function is async-signal safe if it can be safely called within a signal handler
- Functions are NOT async-signal safe if they
 - Use static data structures
 - Call malloc or free
 - Use global data structures in a non-reentrant way
- A single process cannot execute two occurrences of these calls concurrently
- Signals add concurrency to a program

Signal Handling Rules

- When in doubt, explicitly restart system calls within program
- Do not use functions that are not async-signal safe
- Analyze interactions – block signals that can cause unwanted interactions

Async-Signal Safe Functions

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

```

...
static volatile sig_atomic_t jumpok = 0;
static sigjmp_buf jmpbuf;
void int_handler(int signo)
{
    if (jumpok == 0) return;
    siglongjmp(jmpbuf, 1);
}
void main(void)
{
    struct sigaction act;
    int i;
    ...
    act.sa_handler = int_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGINT, &act, NULL) < 0) {
        perror("Error setting up SIGINT handler");
        exit(1);
    }
    ...
    if (sigsetjmp(jmpbuf, 1))
        fprintf(stderr, "Returned to main loop due to ^c\n");
    ...
    jumpok = 1;
        /* start of main loop */
... }

```

siglongjump