System Programming

Week 7: Signals

Seongjin Lee

Updated: 2016-11-09
07_signal

insight@hanyang.ac.kr
http://esos.hanyang.ac.kr
Esos Lab. Hanyang University

Table of contents

Introduction

This chapter covers following items

- The concept
- Use cases of signal
- The problems of earlier implementations
- The correct ways

Signal Concepts

Signal Concepts

Every signal has a name that begins with `SIG` and it is assigned with a positive number defined in `<signal.h>`

- `SIGABRT`: generated when a process calls `abort` function
- `SIGALRM`: generated when a timer set byt `alarm` function goes off
- Different versions of Unix have different number of signals

Signal Concepts

Signal generating conditions

1. terminal generated signals (`DELETE` or ^`-C` on many systems) causes the interrupt signal (`SIGINT`)

Signal Concepts

Signal generating conditions

1. terminal generated signals (`DELETE` or ^`-C` on many systems) causes the interrupt signal (`SIGINT`)
2. Hardware exceptions generate singals
   - invalid memory reference (`SIGSEGV`)
   - I/O completed (`SIGIO`)
   - user disconnected from the system (`SIGHUP`)
   - detected by HW and the kernel is notified
   - kernel generates the appropriate signal for the process

Signal Concepts

Signal generating conditions

1. terminal generated signals (`DELETE` or ^`-C` on many systems) causes the interrupt signal (`SIGINT`)
2. Hardware exceptions generate singals
   - invalid memory reference (`SIGSEGV`)
   - I/O completed (`SIGIO`)
   - user disconnected from the system (`SIGHUP`)
   - detected by HW and the kernel is notified
   - kernel generates the appropriate signal for the process
3. `kill(2)` function allows a process to send any signal to another process or group (have to be owner, or the superuser)

Signal Concepts

Signal generating conditions

1. terminal generated signals (`DELETE` or ^`-C` on many systems) causes the interrupt signal (`SIGINT`)
2. Hardware exceptions generate singals
   - invalid memory reference (`SIGSEGV`)
   - I/O completed (`SIGIO`)
   - user disconnected from the system (`SIGHUP`)
   - detected by HW and the kernel is notified
   - kernel generates the appropriate signal for the process
3. `kill(2)` function allows a process to send any signal to another process or group (have to be owner, or the superuser)
4. `kill(1)` command sends signal to other process

Signal Concepts

Signal generating conditions

1. terminal generated signals (`DELETE` or ^`-C` on many systems) causes the interrupt signal (`SIGINT`)
2. Hardware exceptions generate singals
   - invalid memory reference (`SIGSEGV`)
   - I/O completed (`SIGIO`)
   - user disconnected from the system (`SIGHUP`)
   - detected by HW and the kernel is notified
   - kernel generates the appropriate signal for the process
3. `kill(2)` function allows a process to send any signal to another process or group (have to be owner, or the superuser)
4. `kill(1)` command sends signal to other process
5. Software conditions can generate signals

Signal Concepts cont'd

Signals are asynchronous events that occurs randomly

The process has to tell the kernel "if and when this signal occurs, do the following"

Signal Concepts cont'd

Signals are asynchronous events that occurs randomly

The process has to tell the kernel "if and when this signal occurs, do the following"

We can tell the kernel to do one of three things

1. **Ignore the signal** following two can never be ignored: `SIGKILL` and `SIGSTOP`

Signal Concepts cont'd

Signals are asynchronous events that occurs randomly

The process has to tell the kernel "if and when this signal occurs, do the following"

We can tell the kernel to do one of three things

1. **Ignore the signal** following two can never be ignored: `SIGKILL` and `SIGSTOP`
2. **Catch the signal** We tell the kernel to call a customized function whenever the signal occurs
   - if `SIGCHLD` signal is caught, it means child has terminated
   - signal catching function calls `waitpid` to fetch the child's process ID and termination status

Signal Concepts cont'd

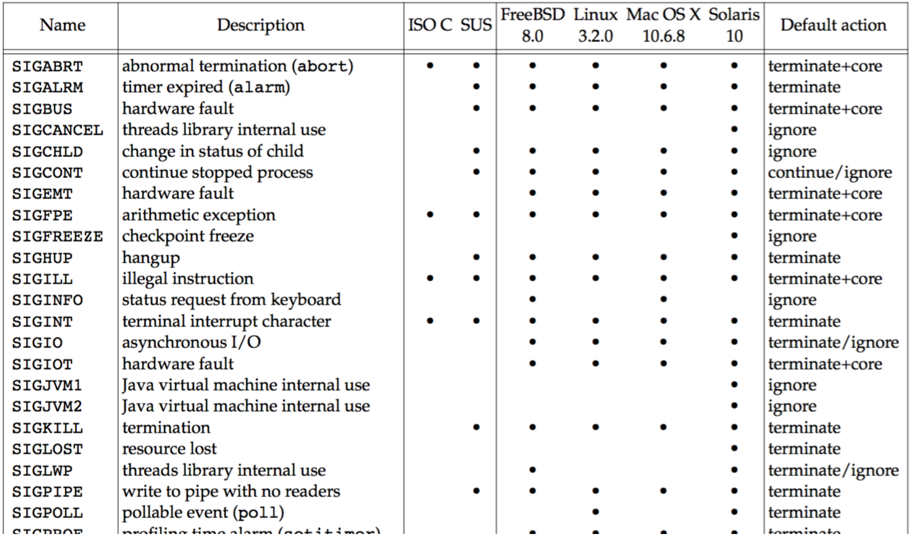
Signals are asynchronous events that occurs randomly

The process has to tell the kernel "if and when this signal occurs, do the following"

We can tell the kernel to do one of three things

1. **Ignore the signal** following two can never be ignored: `SIGKILL` and `SIGSTOP`
2. **Catch the signal** We tell the kernel to call a customized function whenever the signal occurs
   - if `SIGCHLD` signal is caught, it means child has terminated
   - signal catching function calls `waitpid` to fetch the child's process ID and termination status
3. **Use default action** every signal has a default action
   - the default action for most signals is to terminate the process>

Signal Concepts cont'd



Name	Description	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Default action
`SIGABRT`	abnormal termination (`abort`)	•	•	•	•	•	•	terminate+core
`SIGALRM`	timer expired (`alarm`)		•	•	•	•	•	terminate
`SIGBUS`	hardware fault		•	•	•	•	•	terminate+core
`SIGCANCEL`	threads library internal use						•	ignore
`SIGCHLD`	change in status of child		•	•	•	•	•	ignore
`SIGCONT`	continue stopped process		•	•	•	•	•	continue/ignore
`SIGEMT`	hardware fault			•	•	•	•	terminate+core
`SIGFPE`	arithmetic exception	•	•	•	•	•	•	terminate+core
`SIGFREEZE`	checkpoint freeze						•	ignore
`SIGHUP`	hangup		•	•	•	•	•	terminate
`SIGILL`	illegal instruction	•	•	•	•	•	•	terminate+core
`SIGINFO`	status request from keyboard			•		•		ignore
`SIGINT`	terminal interrupt character	•	•	•	•	•	•	terminate
`SIGIO`	asynchronous I/O			•	•	•	•	terminate/ignore
`SIGIOT`	hardware fault			•	•	•	•	terminate+core
`SIGJVM1`	Java virtual machine internal use						•	ignore
`SIGJVM2`	Java virtual machine internal use						•	ignore
`SIGKILL`	termination		•	•	•	•	•	terminate
`SIGLOST`	resource lost						•	terminate
`SIGLWP`	threads library internal use			•			•	terminate/ignore
`SIGPIPE`	write to pipe with no readers		•	•	•	•	•	terminate
`SIGPOLL`	pollable event (`poll`)				•		•	terminate
`SIGPROF`	profiling time alarm (`setitimer`)			•	•	•	•	terminate

Figure: UNIX System signals

Signal Concepts cnt'd

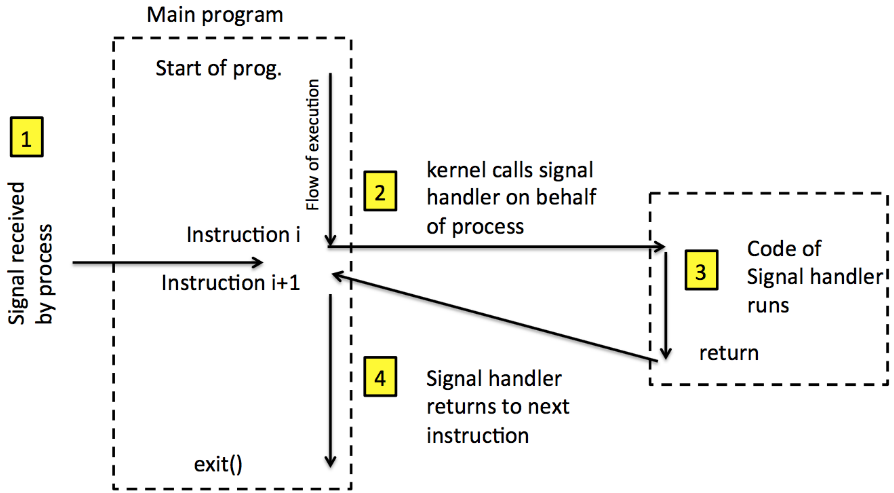

Main program
Start of prog.
Flow of execution
Instruction i
Instruction i+1
exit()
1
Signal received by process
2
kernel calls signal handler on behalf of process
3
Code of Signal handler runs
return
4
Signal handler returns to next instruction

Figure: Signal handling concept

signal Function

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
// Returns: previous disposition of signal (see following) if OK, SIG_ERR on
     error
```

- signo is name of the signal from the Table 10.1
- func is
    - SIG_IGN : to ignore the signal
    - SIG_DFL : to use the default value
    - address of a function to be called when the signal occurs—they are called *signal handler* or *signal-catching function*

Signal Exmple Code: codes/usr_sig.c I

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h> // for signalling
#include <errno.h>

static void sig_usr(int); /* one handler for both signals */

int
main(void) {
   if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
      fprintf(stderr, "Can't catch SIGUSR1: %s", strerror(errno));
      exit(1);
   }
   if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
      fprintf(stderr, "Can't catch SIGUSR2: %s", strerror(errno));
      exit(1);
   }

   if (signal(SIGHUP, sig_usr) == SIG_ERR) {
      fprintf(stderr, "Can't catch SIGHUP: %s", strerror(errno));
```

Signal Exmple Code: `codes/usr_sig.c` II

```
        exit(1);
    }

    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 if (signo == SIGHUP)
        printf("received SIGHUP\n");
    else {
        fprintf(stderr, "received signal: %d\n", signo);
        exit(1);
    }
    return;
}
```

Signal Exmaples

invoke the program in the background and use `kill(1)` to send signal

```
James@maker:codes$ ./usr\_sig &
[2] 4987
James@maker:codes$ kill -USR1 4987
received SIGUSR1
James@maker:codes$ kill -USR2 4987
received SIGUSR2
James@maker:codes$ kill -HUP 4987
received SIGHUP
James@maker:codes$ kill -INT 4987
[2]+ Interrupt: 2 ./usr\_sig
James@maker:codes$
```

Signal Exmaples cnt'd

```
$ sleep 100 &
[ 1 ] 3486
$ pgrep sleep
3486
$ ps aux | grep sleep

$ sleep 100 &
[ 1 ]  3490
$ kill 3490  # kills pid 3490 gracefully
$
[ 1 ]+ Terminated: 15    sleep 100

$ sleep 100 &
[ 1 ] 3437
$ jobs -l
[ 1 ]+ 3637 Running   sleep 100 &
$ kill -s STOP 3637
[ 1 ]+ Stopped    sleep  100
$ kill s CONT 3637
[ 1 ]+ 3637 suspended (signal): 17 sleep 100
```

```
$ sleep 100 &
[ 1 ]  3490
$ kill s SIGKILL 3490
Or
$ kill s KILL 3490
Or
$ kill SIGKILL 3490
Or
$ kill KILL 3490
Or
$ kill s 9 3490
Or
$ kill -9 3490
[ 1 ]+ Killed: 15    sleep 100
```

`signal` Function

At process creation

- When a process calls `fork`, the child inherits the parents signal disposition
- Child starts off with copy of the parent's memory image
- the address of a signal-catching function has meaning in the child

The issues

Unreliable Signals

In earlier versions of Unix system, signals were unreliable

Unreliable Signals

In earlier versions of Unix system, signals were unreliable

1. signals get lost: signal occurs and the process never know about it

Unreliable Signals

In earlier versions of UNIX system, signals were unreliable

1. signals get lost: signal occurs and the process never know about it
2. window of time—after the signal has occured, but before the call to `signal` in the signal handler—when the interrupt signal could occur another time. The second signal would cause the default action to occur (terminate the process)

Unreliable Signals

In earlier versions of Unix system, signals were unreliable

1. signals get lost: signal occurs and the process never know about it
2. window of time—after the signal has occured, but before the call to `signal` in the signal handler—when the interrupt signal could occur another time. The second signal would cause the default action to occur (terminate the process)
3. little control over a signal: unable to turn a signal off when it didn't want the signal to occur. All it can do is to catch or ignore the signal.

Interrupted System Calls cont'd

The slow system calls are those that can block forever

- reads (for pipes, terminal devices, and network devices) that can block the caller
- writes that can block the caller forever if the data can't be accepted immediately
- open on a certain file types (terminal device) that block the caller until some condition occurs
- the `pause` and `wait` function
- certain `ioctl` operations
- some of interprocess communication function

Interrupted System Calls cont'd

The problem with interrupted system calls is that error returns must be explicit

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0 ) {
        if (errno == EINTR)
            goto again; /* just an interrupted system call */
        /* handle other errors */
    }
```

The solution of 4.2BSD was to introduce the automatic restaring of `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`

Reentrant Functions

Scenario

1. while process is running, it catches a signal. process is temporarily interrupted by the signal handler

Reentrant Functions

Scenario

1. while process is running, it catches a signal. process is temporarily interrupted by the signal handler
2. instructions in signal handler executes

Reentrant Functions

Scenario

1. while process is running, it catches a signal. process is temporarily interrupted by the signal handler
2. instructions in signal handler executes
3. upon return of signal handler, the process continues to run

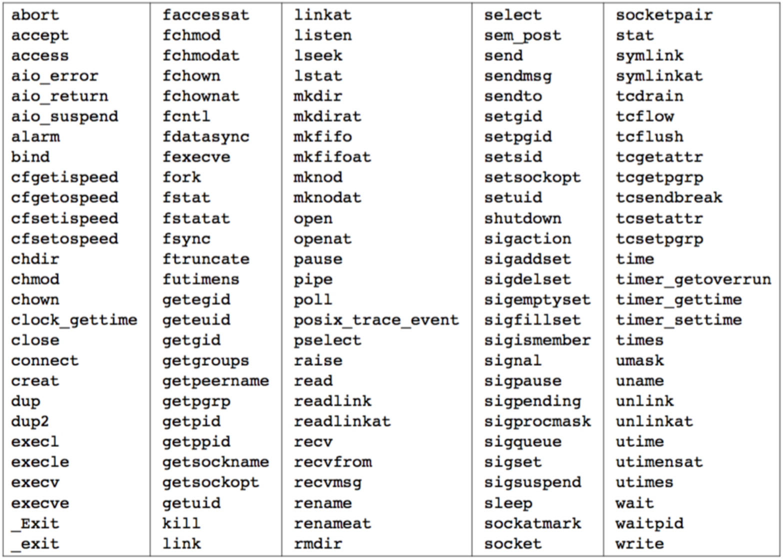
The problem is that signal handler can't tell whether a process was in the middle of execution. *The result becomes unpredictable*

Reentrant Functions cont'd

Reentrant Functions

Reentrant fuctions are guarnateed to be safe to call from within a signal hanlder. They are also called *async-signal safe*. As a general rule, when calling the reentrant functions from a signal handler, we should save and restore `errno`.

Reentrant Functions cont'd



abort	faccessat	linkat	select	socketpair
accept	fchmod	listen	sem_post	stat
access	fchmodat	lseek	send	symlink
aio_error	fchown	lstat	sendmsg	symlinkat
aio_return	fchownat	mkdir	sendto	tcdrain
aio_suspend	fcntl	mkdirat	setgid	tcflow
alarm	fdatasync	mkfifo	setpgid	tcflush
bind	fexecve	mkfifoat	setsid	tcgetattr
cfgetispeed	fork	mknod	setsockopt	tcgetpgrp
cfgetospeed	fstat	mknodat	setuid	tcsendbreak
cfsetispeed	fstatat	open	shutdown	tcsetattr
cfsetospeed	fsync	openat	sigaction	tcsetpgrp
chdir	ftruncate	pause	sigaddset	time
chmod	futimens	pipe	sigdelset	timer_getoverrun
chown	getegid	poll	sigemptyset	timer_gettime
clock_gettime	geteuid	posix_trace_event	sigfillset	timer_settime
close	getgid	pselect	sigismember	times
connect	getgroups	raise	signal	umask
creat	getpeername	read	sigpause	uname
dup	getpgrp	readlink	sigpending	unlink
dup2	getpid	readlinkat	sigprocmask	unlinkat
execl	getppid	recv	sigqueue	utime
execle	getsockname	recvfrom	sigset	utimensat
execv	getsockopt	recvmsg	sigsuspend	utimes
execve	getuid	rename	sleep	wait
_Exit	kill	renameat	sockatmark	waitpid
_exit	link	rmdir	socket	write

Figure: Reentrant functions that may be called from a signal handler

Reentrant Functions example I

codes/reenter.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <pwd.h>

// change the user name
#ifndef USER
#define USER "James"
#endif

static void
my_alarm(int signo)
{
   struct passwd *rootptr;

   write(STDOUT_FILENO, "in signal handler\n", 18);
   if ((rootptr = getpwnam("root")) == NULL){
      write(STDERR_FILENO, "getpwnam(root) error\n", 21);
      exit(1);
   }
```

Reentrant Functions example II

```
    alarm(1);
}

int
main(void)
{
    struct passwd *ptr;

    if (signal(SIGALRM, my_alarm) == SIG_ERR) {
        fprintf(stderr, "Unable to establish signal handler: %s\n", strerror(errno));
        exit(1);
    }
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam(USER)) == NULL){
            fprintf(stderr, "user %s not found, getpwnam error\n", USER);
            exit(1);
        }
        if (strcmp(ptr->pw_name, USER) != 0){
            fprintf(stderr, "return value corrupted!, pw_name = %s\n",
                    ptr->pw_name);
            abort();
        }
    }
}
```

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
  - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
  - when the signal is generated, the kernel usually sets a flag of some form in the process table

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
  - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
  - when the signal is generated, the kernel usually sets a flag of some form in the process table
- **a signal is *delivered* to a process** when the action for a signal is taken
  - during the time between the generation of a signal and its delivery, the signal is said to be ***pending***

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
  - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
  - when the signal is generated, the kernel usually sets a flag of some form in the process table
- **a signal is *delivered* to a process** when the action for a signal is taken
  - during the time between the generation of a signal and its delivery, the signal is said to be ***pending***
- **a process has the option of *blocking* the delievery of a signal**

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
  - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
  - when the signal is generated, the kernel usually sets a flag of some form in the process table
- **a signal is *delivered* to a process** when the action for a signal is taken
  - during the time between the generation of a signal and its delivery, the signal is said to be *pending*
- **a process has the option of *blocking* the delievery of a signal**
- signals are *queued* when system delievers signals

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
    - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
    - when the signal is generated, the kernel usually sets a flag of some form in the process table
- **a signal is *delivered* to a process** when the action for a signal is taken
    - during the time between the generation of a signal and its delivery, the signal is said to be ***pending***
- **a process has the option of *blocking* the delievery of a signal**
- signals are ***queued*** when system delievers signals
- POSIX.1 does not specify the order if more than one signal is ready to be delievered to a process

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
  - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
  - when the signal is generated, the kernel usually sets a flag of some form in the process table
- **a signal is *delivered* to a process** when the action for a signal is taken
  - during the time between the generation of a signal and its delivery, the signal is said to be ***pending***
- **a process has the option of *blocking* the delievery of a signal**
- signals are ***queued*** when system delievers signals
- POSIX.1 does not specify the order if more than one signal is ready to be delievered to a process
- **each process has a *signal mask*** that defines the set of singals currently blocked for delievery to that process

Reliable-Singnal Teminology and Semantics

- **a signal is *generated* (or sent) for a process** when the event that causes the signal occurs
  - The event can be hardware exception, software condition, a terminal-generated signal, or a call to the `kill` function
  - when the signal is generated, the kernel usually sets a flag of some form in the process table
- **a signal is *delivered* to a process** when the action for a signal is taken
  - during the time between the generation of a signal and its delivery, the signal is said to be ***pending***
- **a process has the option of *blocking* the delievery of a signal**
- signals are ***queued*** when system delievers signals
- POSIX.1 does not specify the order if more than one signal is ready to be delievered to a process
- **each process has a *signal mask*** that defines the set of singals currently blocked for delievery to that process
- POSIX.1 defines a data type `sigset_t` that holds a ***signal set***

Use cases of signal

kill and raise Functions

kill function sends a signal to a process or a group of processes

raise function allows a process to send a signal to itself

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

`raise(signo);` == `kill(getpid(), singo);`

- pid>0 sends signal to pid
- pid==0 sends signal to all prosesses in process group ID of the sender
- pid<0 The signal is sent to all process in process group ID of $|pid|$
- pid==-1 send signal to all processes on the system for which the sender has permission to send the signal

`alarm` Function

`alarm` function allows us to set a time that will expire at a specified time in the future. When the timer expires, the `SIGALRM` signal is generated

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
// Returns: 0 or number of seconds until previously set alarm
```

- only one of alarm clocks per process
- if previously registered alarm clock has not expired, the number of seconds left for that alarm clock is returned as the value of this function
- if previous alarm is not expired and *seconds* value is `0`, the previous alarm is cacled
- most processes using alarm, catches this signal

pause Function

pause function suspends the calling process until a signal is caught

```
#include <unistd.h>
int pause(void);
// Returns: 1 with errno set to EINTR
```

alarm and pause Functions Example

codes/sleep-pause.c

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

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

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

`alarm` and `pause` Functions Example

There are three problems in the code

`alarm` and `pause` Functions Example

There are three problems in the code

1. if the caller already has an alarm set, that alarm is erased by the first call to `alarm`
   - we have to check the return value of `alarm`, and make sure we wait only until the existing alarm expires

`alarm` and `pause` Functions Example

There are three problems in the code

1. if the caller already has an alarm set, that alarm is erased by the first call to `alarm`
   - we have to check the return value of `alarm`, and make sure we wait only until the existing alarm expires
2. disposition for `SIGALRM` is modified
   - if others are going to use this call, make sure the function is restored after use

`alarm` and `pause` Functions Example

There are three problems in the code

1. if the caller already has an alarm set, that alarm is erased by the first call to `alarm`
   - we have to check the return value of `alarm`, and make sure we wait only until the existing alarm expires
2. disposition for `SIGALRM` is modified
   - if others are going to use this call, make sure the function is restored after use
3. there is race condition between the first call to `alarm` and the call to `puase`
   - use `setjmp` or `sigprocmask` with `sigsuspend`

Example cont'd codes/sleep-pause2.c

to avoid race condition SVR2 used setjmp and longjmp

```c
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>

static jmp_buf env_alrm;

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

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

Example cont'd `codes/tsleep.c` I

There is subtle problem with `sleep2` function when it interacts with other signals

In this example, we are trying to make it execute longer than 5 seconds (the argument to `sleep2()`

```c
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

unsigned int sleep2(unsigned int);
static void  sig_int(int);

static jmp_buf env_alrm;

int
main(void)
{
   unsigned int unslept;

```

Example cont'd codes/tsleep.c II

```
    if (signal(SIGINT, sig_int) == SIG_ERR){
        fprintf(stderr, "signal(SIGINT) error");
        exit(1);
    }
    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int    i, j;
    volatile int k = 0;

    /*
     * Tune these loops to run for more than 5 seconds
     * on whatever system this test program is run.
     */
    printf("\nsig_int starting\n");
    for (i = 0; i < 900000; i++)
        for (j = 0; j < 10000; j++)
            k += i * j;
    printf("sig_int finished\n");
}

```

Example cont'd `codes/tsleep.c` III

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

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

Example cont'd codes/tsleep.c

```
make tsleep
```

We execute the program by `./tsleep` and interrupt the sleep by typing in the interrupt character

```
James@maker:codes$ time ./tsleep
sleep2 returned: 0

real 0m5.014s
user 0m0.005s
sys 0m0.007s
```

```
James@maker:codes$ time ./tsleep
^C
sig_int starting
sleep2 returned: 0

real 0m5.008s
user 0m4.590s
sys 0m0.023s
```

we can see that the `longjmp` from the `sleep2` aborted the other signal handler, `sig_int`, even though it wasn't finished

Signal Sets

We need a data type to represent multiple signals—*signal set*

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
// All four return: 0 if OK, 1 on error
int sigismember(const sigset_t *set, int signo);
// Returns: 1 if true, 0 if false, 1 on error
```

- `sigemptyset()` initializes the signal set pointed to by set so that all signals are *excluded*
- `sigfillset()` initializes the signal set
- all applications have to call either `sigemptyset` or `sigfillset` once for each signal set, before using signal set
- `sigaddset` adds a single signal to an existing set
- `sigdelset` removes a single signal from a set

Signal Sets cont'd I

An implementation of `sigaddset`, `sigdelset`, and `sigismember`

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

/*
 * <signal.h> usually defines NSIG to include signal number 0.
 */
#define SIGBAD(signo) ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
   if (SIGBAD(signo)) {
      errno = EINVAL;
      return(-1);
   }
   *set |= 1 << (signo - 1);   /* turn bit on */
   return(0);
}


int
sigdelset(sigset_t *set, int signo)
{
```

Signal Sets cont'd II

```
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    *set &= ~(1 << (signo - 1)); /* turn bit off */
    return(0);
}

/* tests a certain bit */
int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    return((*set & (1 << (signo - 1))) != 0);
}
```

POSIX.1 requires us to check the signal number argument for validity and to set `errno` if it is invalid

sigprocmask Function

A process can *examine* its signal mask, *change* its signal mask, *or perform both* operations by calling following function

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
// Returns: 0 if OK, 1 on erro
```

SIG_BLOCK	*set* contains the additional signal that we want to block
SIG_UNBLOCK	*set* contains the additional signal that we want to unblock
SIG_SETMASK	the new signal mask for the process is replaced by the value of the signal set pointed to by *set*

Table: ways to change the current signal mask using sigprocmask

sigprocmask Function I

```
cd codes; vi ex_block.c
```

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int do_block = 0; //toggle
int signum; // received signal

void
hup_handler(int sig) { //hangup signal handler
    signum = sig;
}

void
int_handler(int sig) { //toggle do_block
    signum = sig;
    do_block = !do_block;
}

int
```

sigprocmask Function II

```
main() {
    sigset_t signal_set;

    // Install signal handler.
    signal(SIGHUP, hup_handler); // hangup
    signal(SIGINT, int_handler); // interrupt

    sigemptyset(&signal_set); // empty set of signal_set
    sigaddset(&signal_set, SIGHUP); // add signal hang up to set

    while (1) {
        if (do_block) // if true block signal_set
            sigprocmask(SIG_BLOCK, &signal_set, NULL);
        else // if not unblock signal_set
            sigprocmask(SIG_UNBLOCK, &signal_set, NULL);
        sleep(1000);
        printf("Process received signal %d\n", signum);
    }
}
```

sigprocmask Function cnt'd

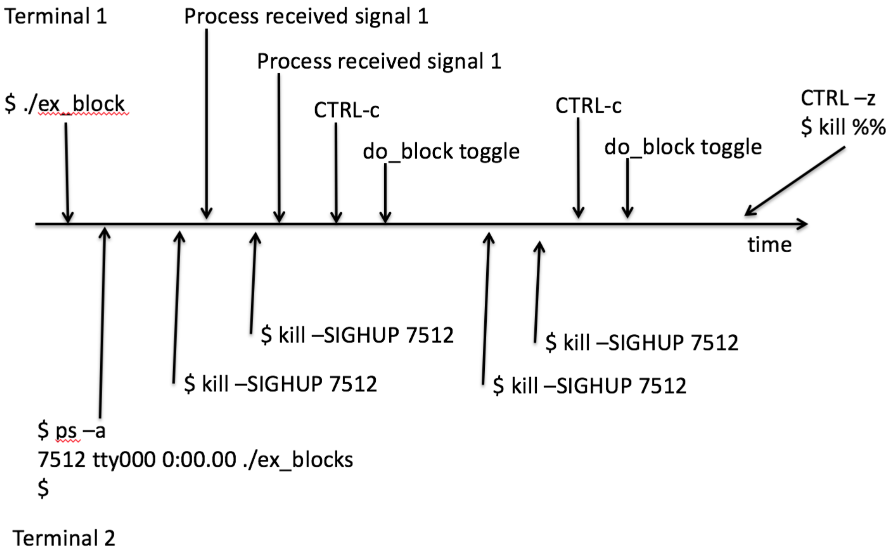


Figure: Example of sigprocmask

sigpending Function

sigpending returns the set of signals that are blocked from delievery and currently pending for the calling process.

the set of signals is returned through the set argument

```
#include <signal.h>
int sigpending(sigset_t *set);
// Returns: 0 if OK, 1 on error
```

sigpending Example I

```
cd codes; make critical
```

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

static void
sig_quit(int signo)
{
   printf("caught SIGQUIT\n");
   if (signal(SIGQUIT, SIG_DFL) == SIG_ERR){
      fprintf(stderr, "can't reset SIGQUIT");
      exit(1);
   }
}

int
main(void)
{
   sigset_t newmask, oldmask, pendmask;

   printf("C-\\ to generate SIGQUIT signal\n");

```

sigpending Example II

```
    // Add user function sig_quit fro SIGQUIT
    if (signal(SIGQUIT, sig_quit) == SIG_ERR){
       fprintf(stderr, "can't catch SIGQUIT");
       exit(1);
    }

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask); // empty the signal set
    sigaddset(&newmask, SIGQUIT); // add SIGQUIT signal to newmask

    // oldmask saves current signal mask for the process
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0){
       fprintf(stderr, "SIG_BLOCK error");
       exit(1);
    }

    sleep(5); /* SIGQUIT here will remain pending */

    // return signals that are blocked from delivery and currently pending for the
         calling process
    if (sigpending(&pendmask) < 0){
       fprintf(stderr, "sigpending error");
       exit(1);
    }
```

sigpending Example III

```
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Restore signal mask which unblocks SIGQUIT.
     * we could use SIG_UNBLOCK.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0){
        fprintf(stderr, "SIG_SETMASK error");
        exit(1);
    }
    printf("SIGQUIT unblocked\n");

    printf("C-\\ to generate SIGQUIT signal again\n");
    sleep(5); /* SIGQUIT here will terminate with core file */
    exit(0);
}
```

sigpending Example

C- to generate SIGQUIT signal SIGQUIT unblocked C- to generate SIGQUIT signal again

```
James@maker:codes$ ./critical
C-\ to generate SIGQUIT signal
^\
SIGQUIT pending
caught SIGQUIT
SIGQUIT unblocked
C-\ to generate SIGQUIT signal again
^\Quit: 3
```

```
James@maker:codes$ ./critical
C-\ to generate SIGQUIT signal
^\^\^\^\^\
SIGQUIT pending
caught SIGQUIT
SIGQUIT unblocked
C-\ to generate SIGQUIT signal again
^\Quit: 3
James@maker:codes$
```

`sigaction` Function

`sigaction` allows to examine or modify the action associated with a particular signal

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
// Returns: 0 if OK, 1 on error
```

- *signo* is the signal number to examine or to modify
- if *act* pointer non-null, we are modifying the actino
- if *oact* pointer is non-null, the system returns the revious action to *oact* pointer

`signal(3)` can be implemented via `sigaction(2)`

sigaction Function cont'd

sigaction uses following structure

```
struct sigaction {
   void     (*sa_handler)(int);  /* addr of signal handler, */
                                 /* or SIG_IGN, or SIG_DFL */
   sigset_t sa_mask;             /* additional signals to block */
   int      sa_flags;            /* signal options, Figure 10.16 */
   /* alternate handler */
   void     (*sa_sigaction)(int, siginfo_t *, void *);
};
```

- sa_handler field contains teh address of signal catching function
- sa_mask specifies a set of singals that are added to the signal mask before the signal-catching function is called
- if and when the signal catching function returns, the signal mask of the process is reset to its previous value
- OS includes the signal being delivered in the signal mask when the handler is invoked
- the signal handler remain installed until explicitly changed

sigaction Function cont'd

```
cd codes; vi sigaction_sig.c
```

```
/* Reliable version of signal(), using POSIX sigaction(). */
Sigfunc *
signal(int signo, Sigfunc *func)
{
   struct sigaction act, oact;
   act.sa_handler = func;

   // must initialize the sa_mask member of the structure
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;

   if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
       act.sa_flags |= SA_INTERRUPT;
#endif
   } else {
      // intentionally set the SA_RESTART flag
      // for all other than SIGALRM
       act.sa_flags |= SA_RESTART;
   }
   if (sigaction(signo, &act, &oact) < 0)
       return(SIG_ERR);
   return(oact.sa_handler);
}
```

abort Function

```
#include <stdlib.h>
void abort(void);
// This function never returns
```

This function sends the SIGABRT signal to the caller

The intent of letting the process catch the SIGABTRT is to allow it to perform any cleanup that it wants to do before the process terminates

Last Words

Last Words

- Read Chapter 11