Signals

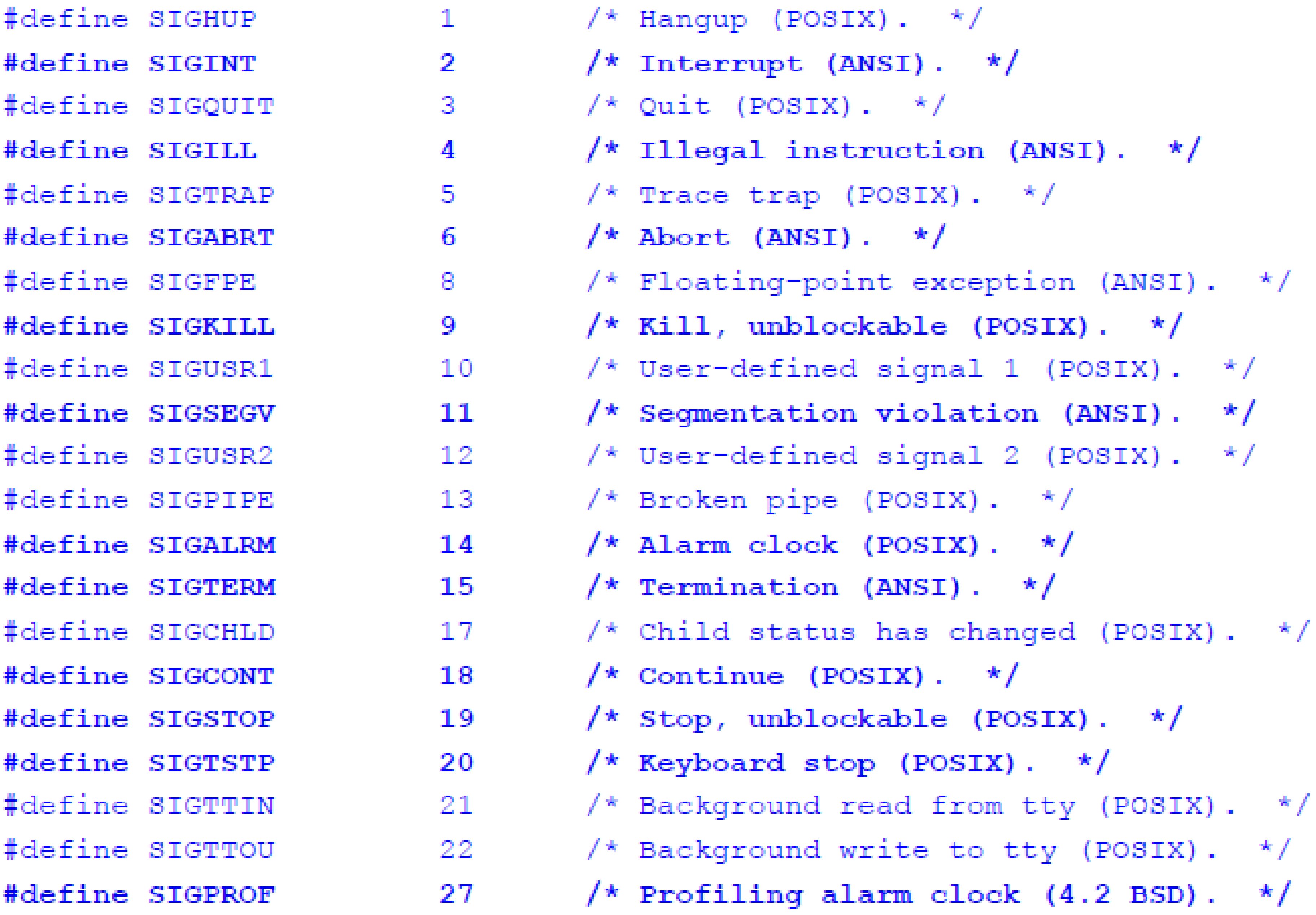
Dr. Jibi Abraham

College of Engineering, Pune

Signal - Definition

- A signal is an *asynchronous* event which is delivered to a process.
- Asynchronous means that the event can occur at any time, may be unrelated to the execution of the process
  - an illegal operation (e.g., divide by 0)
  - a power failure
  - an alarm clock
  - the death of a child process
  - a termination request from a user (Ctrl-C)
  - a suspend request from a user (Ctrl-Z)
- Every signal has a name begins with ‘SIG’
- 31 signals in SVR4 and BSD 4.3+

signals



```
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  */
#define SIGTTIN         21      /* Background read from tty (POSIX).  */
#define SIGTTOU         22      /* Background write to tty (POSIX).  */
#define SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
```

Example Signals

- **SIGABRT** abnormal termination - system function **abort** has been called
- **SIGCHLD** child process status change - parent process is notified whenever a child process stops or terminates (default: ignore)
- **SIGFPE** arithmetic error - When supported by hardware and the O/S, an illegal operation such as divide by zero has been attempted
- **SIGSTOP** process stop order - Issued by **Control-Z** in many shells, stop but do not terminate a process. This cannot be caught or ignored (default: stop process)
- **SIGCONT** signal sent to a stopped process when it is continued
- **SIGTSTP** process stop request - Stop but do not terminate a process. Unlike **SIGSTOP**, this can be caught or ignored (default: stop process)
- **SIGHUP** hang-up terminal connection - A controlling process is notified of a terminal disconnect
- **SIGTERM** process termination request - The default signal for the **kill** command
- **SIGINT** generated by the terminal driver when we type the interrupt key and sent to all processes in the foreground process group

Conditions can generate a signal

- Terminal-generated signals
  - CTRL-C → SIGINT
  - CTRL-Z → SIGSTP signal
- Hardware excepts generate signals
  - divide by 0 → SIGFPE
  - invalid memory reference → SIGSEGV
- kill() function
  - sends any signal to a process or process group
  - need to be owner or super-user
- kill command
  - Used to terminate a runaway background process
- Software conditions
  - SIGALRM: alarm clock expires
  - SIGPIPE: broken pipe
  - SIGURG: out-of-band network data

Signal Dispositions

- Process has to tell the kernel “if and when this signal occurs, do the following.”
- Three types
  - Ignore the signal
    - all signals can be ignored, except SIGKILL and SIGSTOP
  - Catch the signal
  - Let the default action apply
    - most are to terminate process

Signal Default Disposition

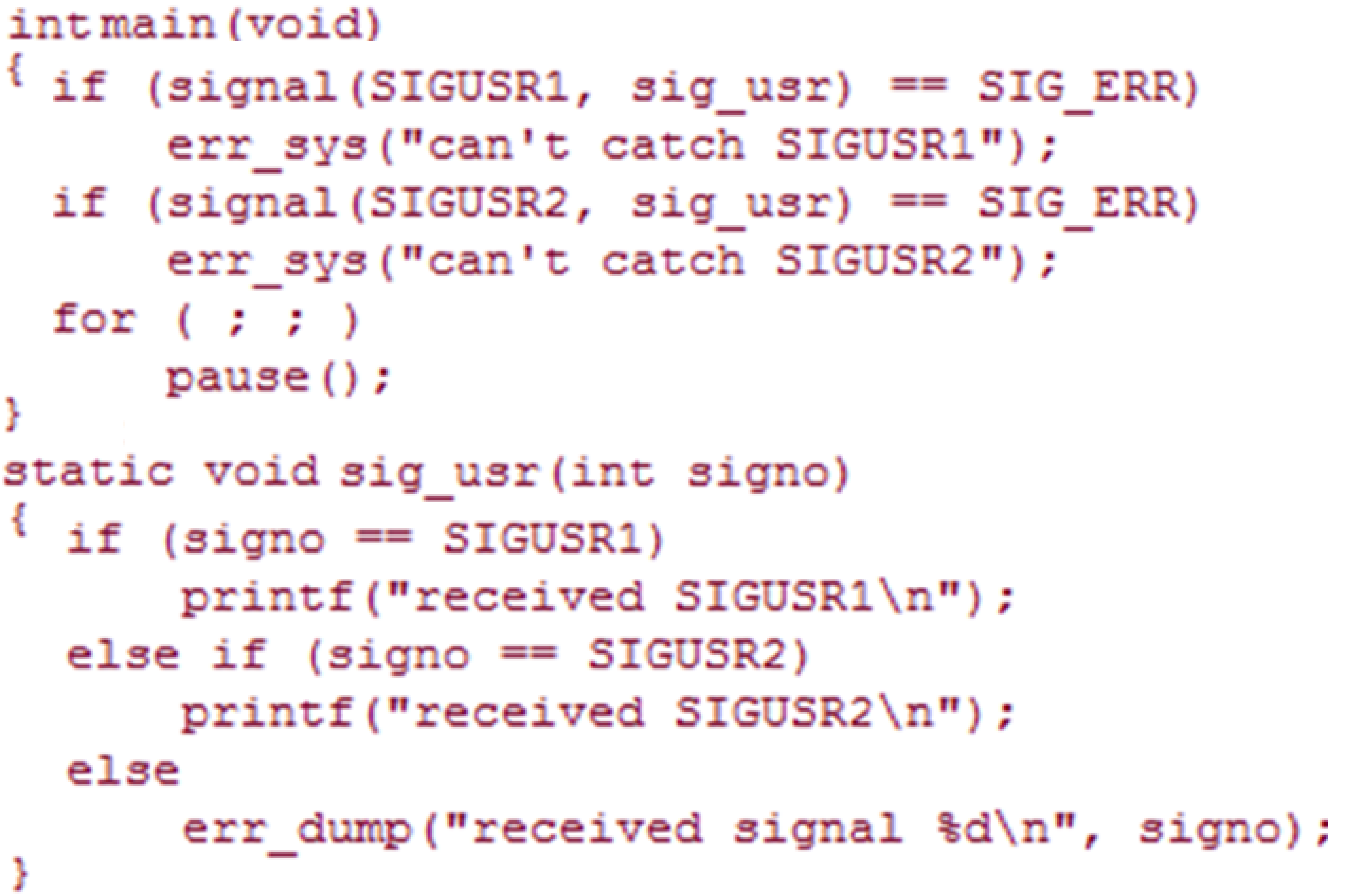
Name	*Description*	*Default Action*
SIGINT	Interrupt character typed	terminate process
SIGQUIT	Quit character typed (^\)	create core image
SIGKILL	kill -9	terminate process
SIGSEGV	Invalid memory reference	create core image
SIGPIPE	Write on pipe but no reader	terminate process
SIGALRM	alarm() clock ‘rings’	terminate process
SIGUSR1	user-defined signal type	terminate process
SIGUSR2	user-defined signal type	terminate process

- See man 7 signal

signal() function

- Signal Handler Registration
- **`void (* signal(int signo, void(*func)(int)))(int);`**
  - specify the action for a signal (*signo* → *func*)
- signal function requires two arguments and returns a pointer to a function that returns void (the previous func)
- signal function's first argument, signo, is an integer
- Second argument is a pointer to a function that takes a single integer argument and returns nothing
- *func*
  - SIG_IGN (ignore)　　　#define SIG_IGN (void (*) ()) 1
  - SIG_DFL (default)　　　#define SIG_IGN (void (*) ()) 0
  - user-defined function

Program to catch sigusr1 and sigusr2



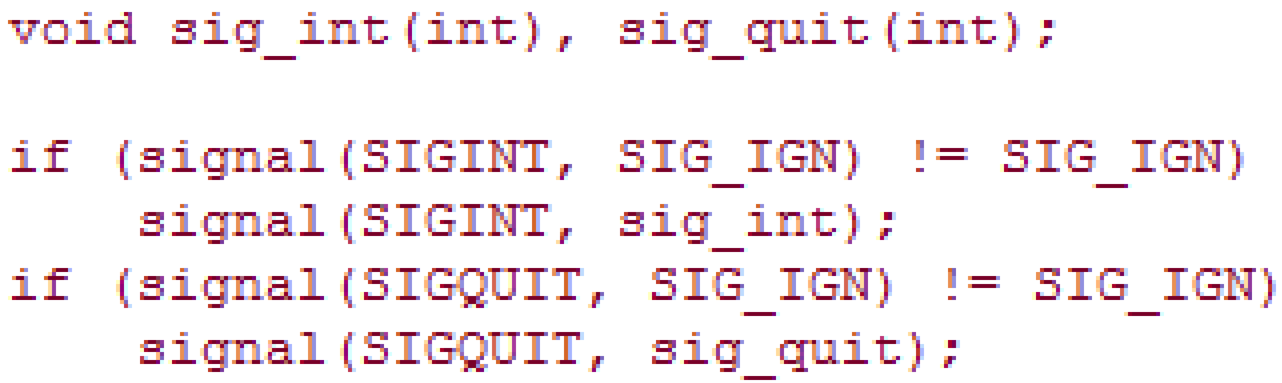
```
intmain(void)
{ if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
  for ( ; ; )
        pause();
}
static void sig_usr(int signo)
{  if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
   else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
   else
        err_dump("received signal %d\n", signo);
}
```

Program execution

```
$a.out &
    [1] 4270
$ kill –USR1 4270
    received SIGUSR1
$ kill –USR2 4270
    received SIGUSR2
$ kill 4270
    [1] + terminated a.out & // SIGTERM is sent
```

Limitation of signal()

- Not able to determine the current disposition of a signal without change the current disposition
- Example: many interactive programs catch SIGINT and SIGQUIT, if they are not currently ignored by coding:



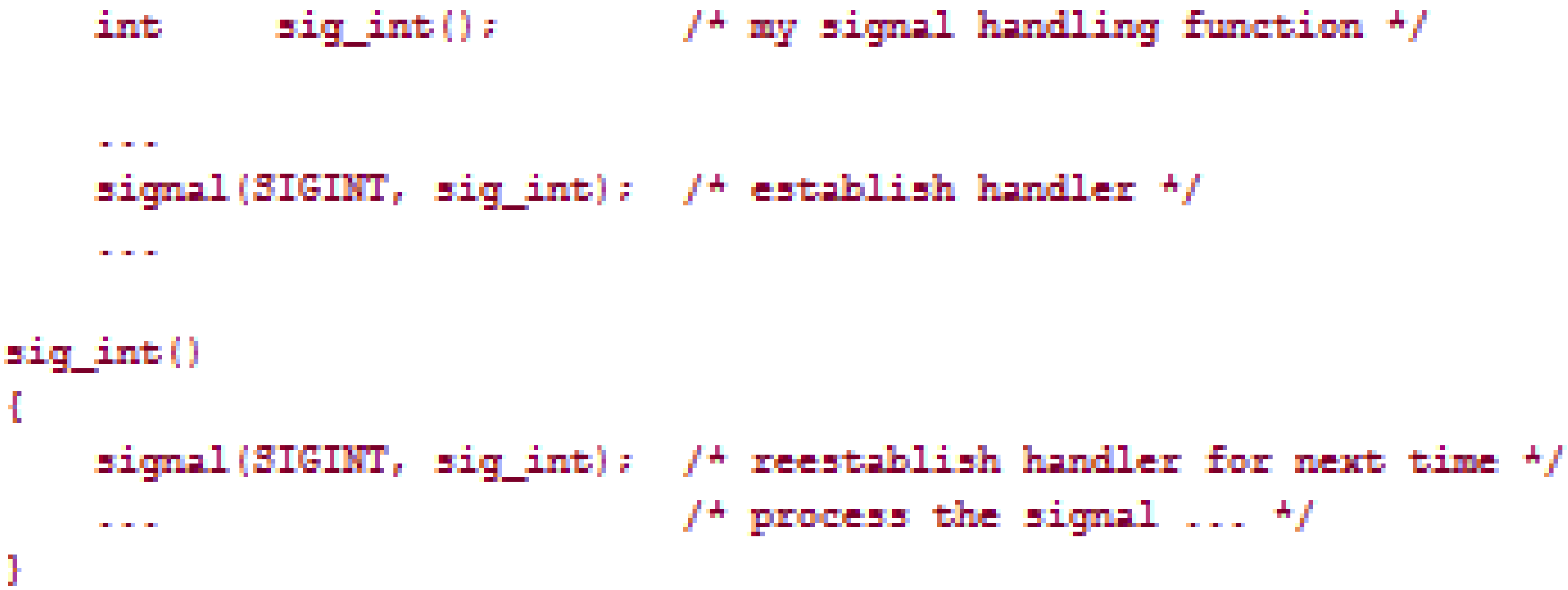
```
void sig_int(int), sig_quit(int);

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

Unreliable Signals

- In earlier versions of the UNIX System (such as Version 7), signals were unreliable
- Signals could get lost: a signal could occur and the process would never know about it
- A process had little control over a signal: a process could catch the signal or ignore it, could not able to block a signal, just remember if it occurs and tell later when the process will be ready
- Changes were made with 4.2BSD to provide what are called reliable signals

Unreliable Signals (contd)

- Signal disposition is reset to its default action immediately after the signal has been delivered. Call *signal()* again to reinstall signal handler function.

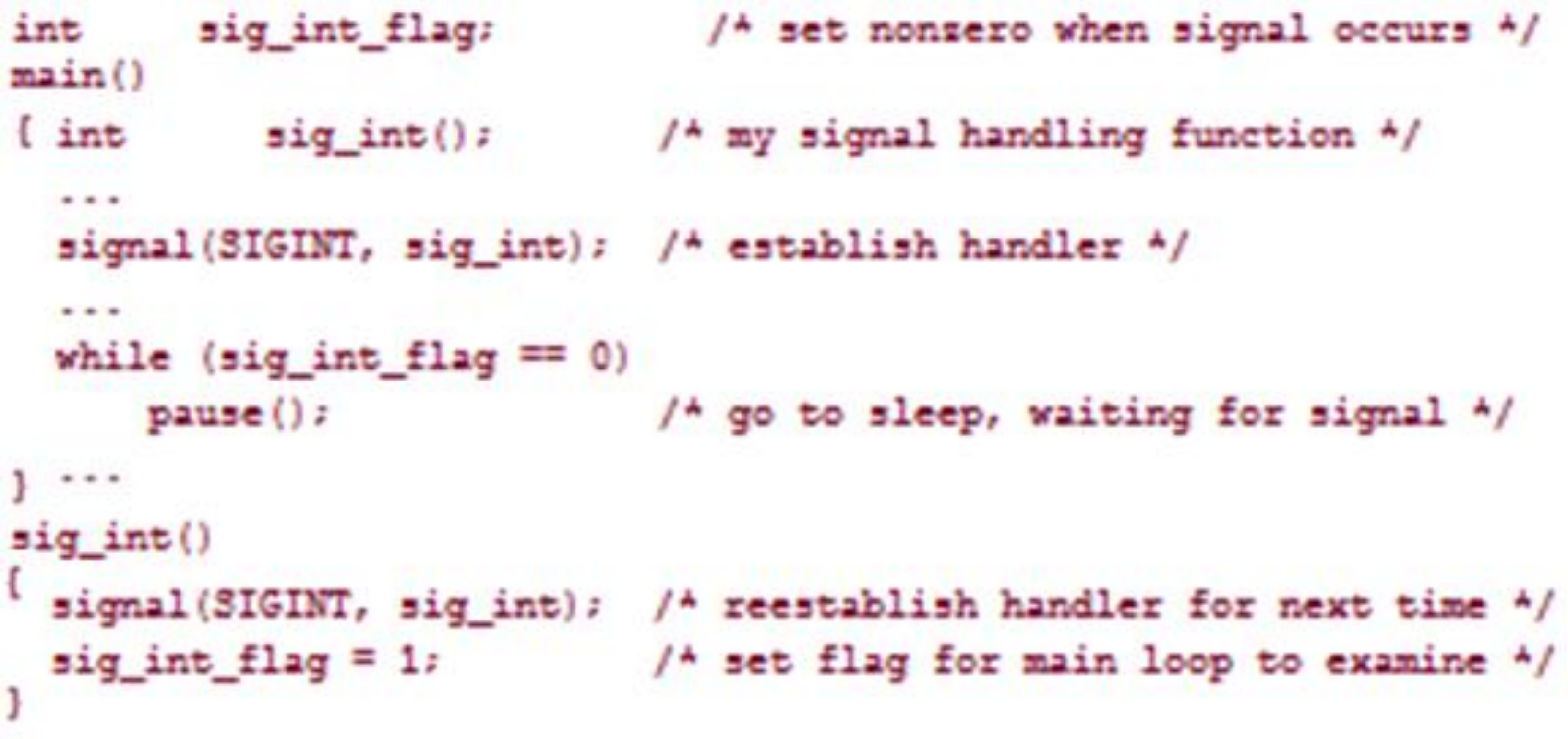


```
    int     sig_int();         /* my signal handling function */

    ...
    signal(SIGINT, sig_int);   /* establish handler */
    ...

sig_int()
{
    signal(SIGINT, sig_int);   /* reestablish handler for next time */
    ...                        /* process the signal ... */
}
```

- Problem: window of time after the signal occurred, but before call to signal in the handler when another interrupt signal occurs, would cause the default action to occur

Unreliable signals

- Another problem: process was unable to turn a signal off when it didn't want the signal to occur.



```
int     sig_int_flag;          /* set nonzero when signal occurs */
main()
{ int       sig_int();        /* my signal handling function */
   ...
   signal(SIGINT, sig_int);  /* establish handler */
   ...
   while (sig_int_flag == 0)
       pause();              /* go to sleep, waiting for signal */
} ...
sig_int()
{
  signal(SIGINT, sig_int);  /* reestablish handler for next time */
  sig_int_flag = 1;         /* set flag for main loop to examine */
}
```

- Problem: If the signal occurs after the test of sig_int_flag, but before call to pause, process could go to sleep forever

Interrupted System Calls

- Slow system functions carry out I/O on things that can possibly block the caller forever:
  - pipes, terminal drivers, networks
  - some IPC functions
  - pause(),
  - some uses of ioctl()
- When a system call (e.g. read()) is interrupted by a signal, a signal handler is called, returns, and then what?
- On many UNIXs, slow system function calls do not resume. Instead they return an error and errno is assigned EINTR.
- Can use signals on slow system functions to code up timeouts

Interrupted System Calls

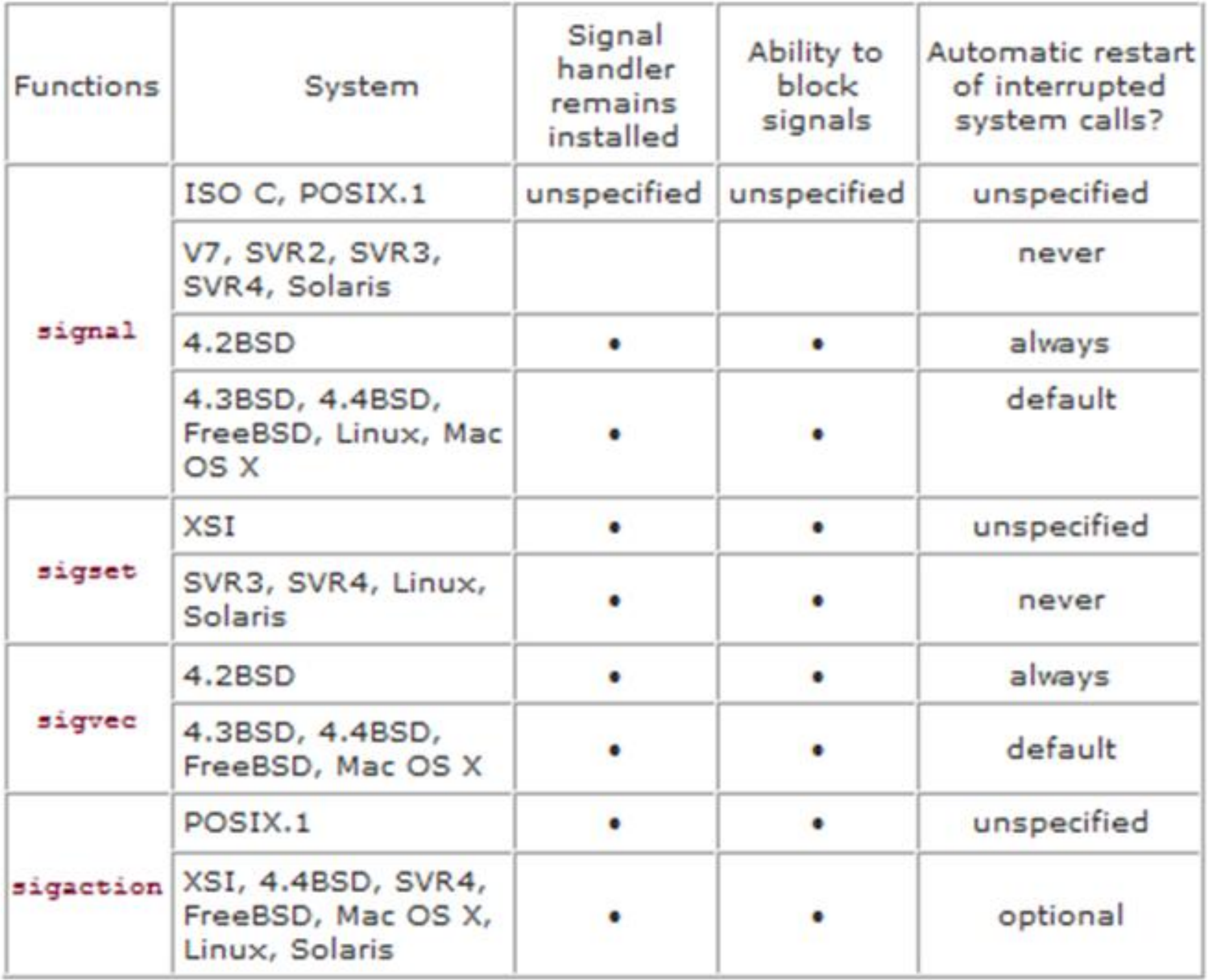
- Most system functions are non-slow, including ones that do *disk* I/O
  - e.g. read() of a disk file
  - read() is sometimes a slow function, sometimes not
- Some UNIXs resume non-slow system functions after the handler has finished
- Some UNIXs only call the handler after the non-slow system function call has finished
- Typical code sequence to restart interrupted system call:

```
again:
        if ((n= read(fd, buf, SIZE))<0) {
                if (errno == EINTR)
                        goto again;
        }
```

Interrupted System Calls

- 4.2 BSD introduced automatic restarting of certain interrupted systems calls
  - ioctl, read, readv, write, writev are interrupted by signal only if they are operating on a slow device
  - wait, and waitpid are always interrupted when a signal is caught
  - POSIX.1 allows system call restart, System V never restarted, 4.2BSD allows automatic restart

Features Provided by Different Signal Implementation



Functions	System	Signal handler remains installed	Ability to block signals	Automatic restart of interrupted system calls?
signal	ISO C, POSIX.1	unspecified	unspecified	unspecified
	V7, SVR2, SVR3, SVR4, Solaris			never
	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	default
sigset	XSI	•	•	unspecified
	SVR3, SVR4, Linux, Solaris	•	•	never
sigvec	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Mac OS X	•	•	default
sigaction	POSIX.1	•	•	unspecified
	XSI, 4.4BSD, SVR4, FreeBSD, Mac OS X, Linux, Solaris	•	•	optional

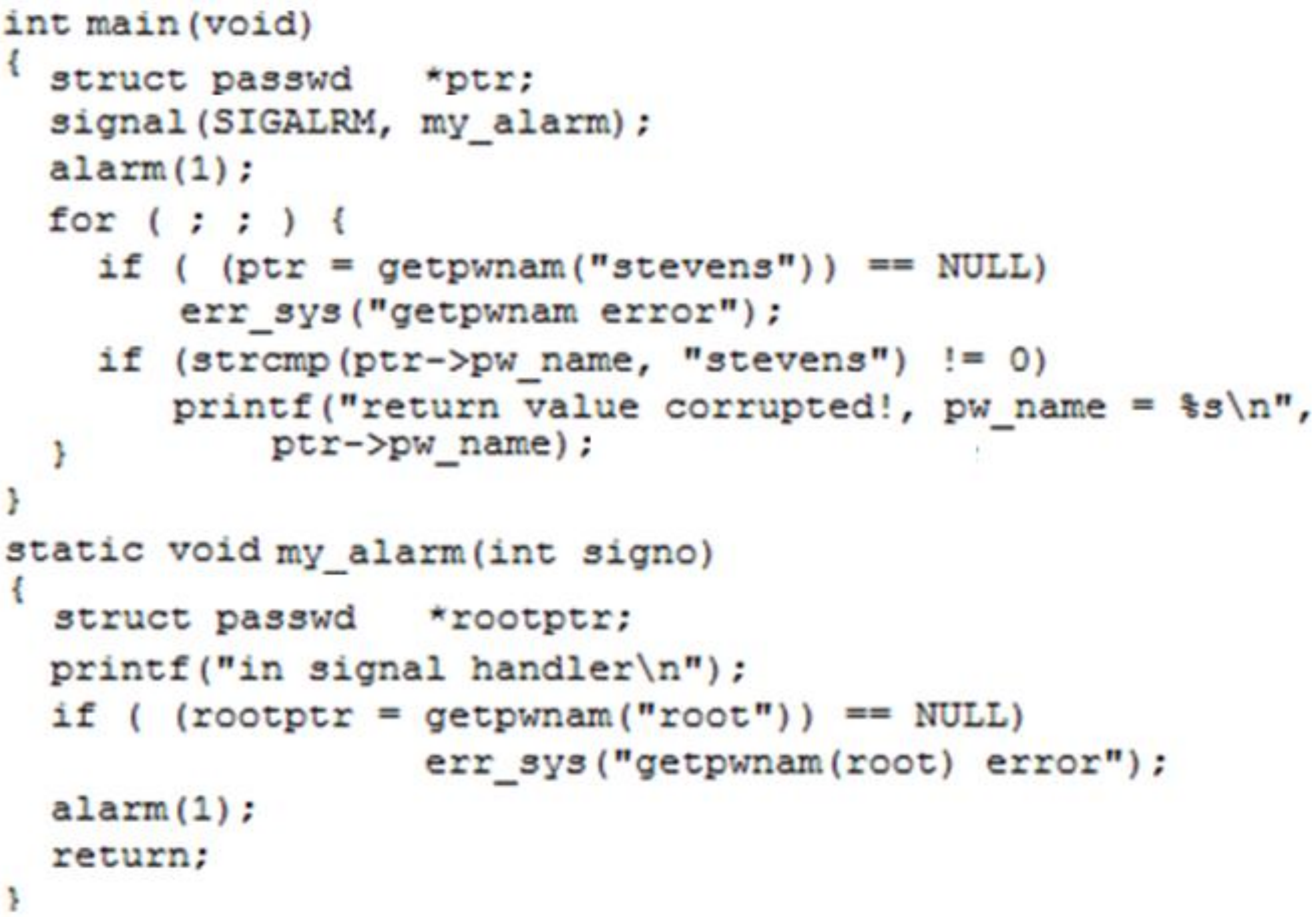
Reentrant functions

- If a system function is called inside a signal handler then it may interact with an interrupted call to the same function in the main code
  - e.g. `malloc()`
    - malloc() operates on a global heap
    - It is possible that two different invocations of malloc that happen at the same time, return the same memory block
    - 2nd malloc call should happen before an address of the chunk is fetched, but the chunk is not marked as unavailable
- This is not a problem if the function is *reentrant*
  - a process can contain multiple calls to these functions at the same time
  - e.g. `read()`, `write()`, `fork()`, many more

Interrupted System Calls

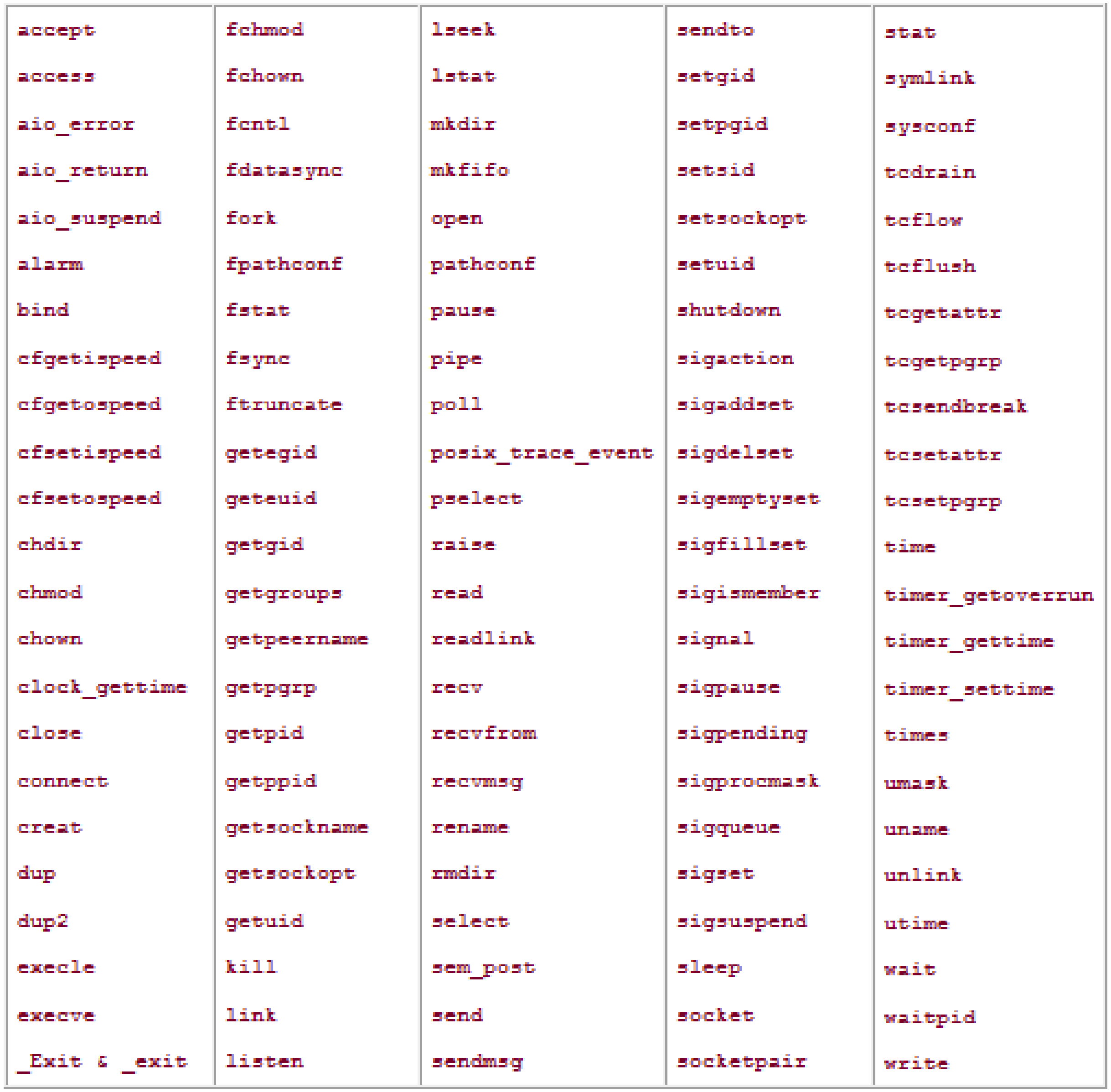
- A functions may be non-reentrant (only one call to it at once) for a number of reasons:
  - it uses a static data structure
  - it manipulates the heap: `malloc()`, `free()`, etc.
  - it uses the standard I/O library
    - e,g, `scanf()`,`printf()`
    - I/O library uses global data structures in a non-reentrant way
    - printf modifies a global variable FILE* stout
  - Gethostbyname returns its value in a static object, multiple calls reuses the same object each time

Example Call a Non-reentrant Function



```
int main(void)
{
  struct passwd   *ptr;
  signal(SIGALRM, my_alarm);
  alarm(1);
  for ( ; ; ) {
    if ( (ptr = getpwnam("stevens")) == NULL)
        err_sys("getpwnam error");
    if (strcmp(ptr->pw_name, "stevens") != 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;
}
```

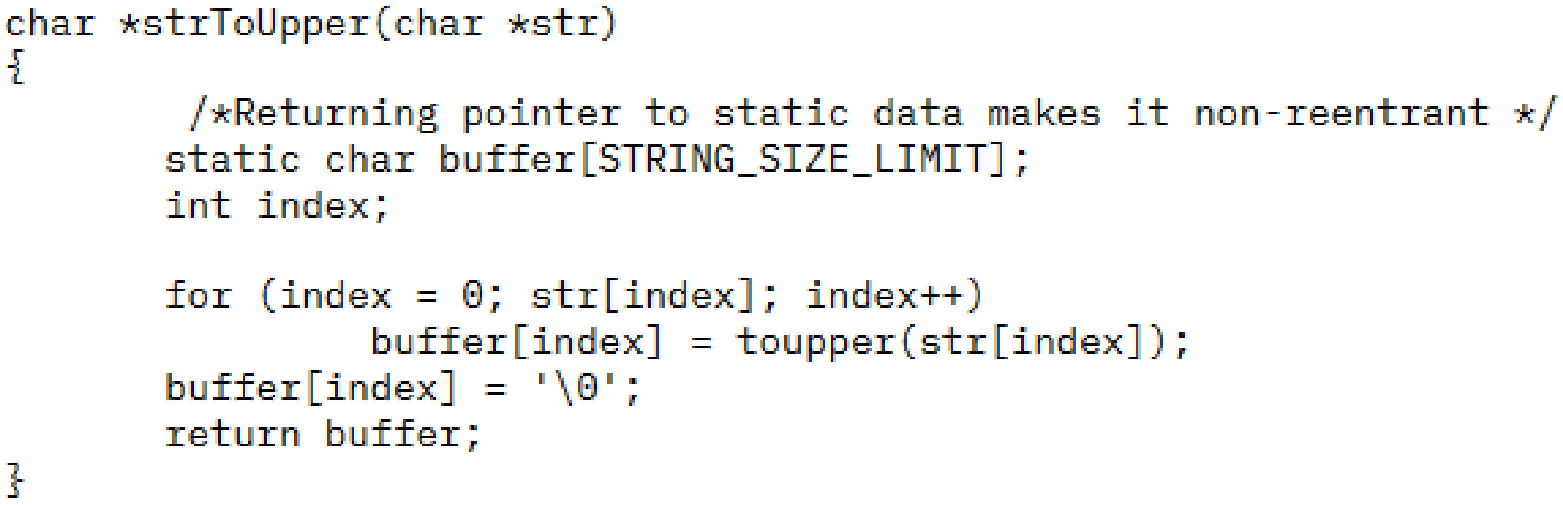
Reentrant Functions that may be Called from a Signal Handler



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

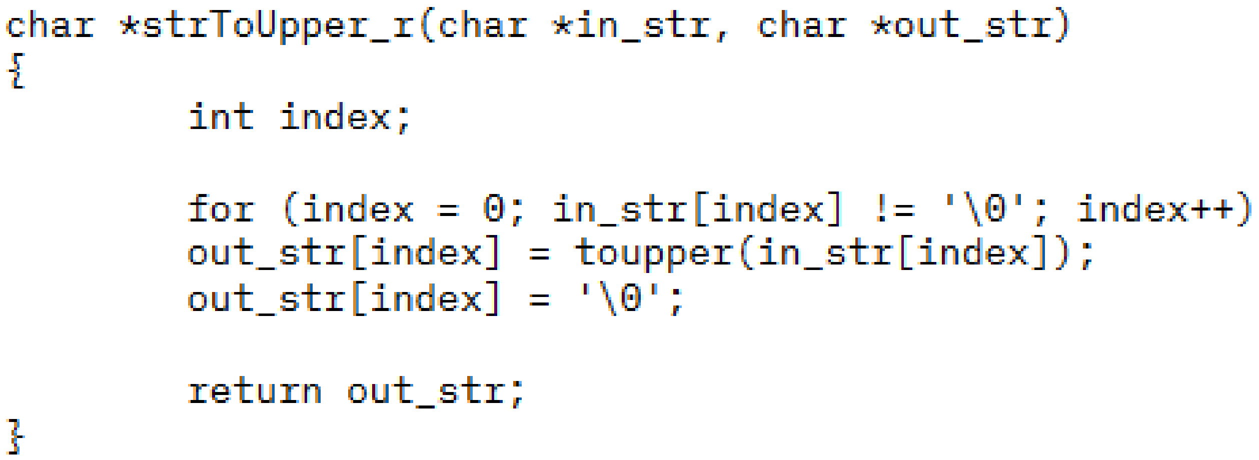
Best Practices to Re-entrancy

- Non-reentrant version of strToUpper



```
char *strToUpper(char *str)
{
         /*Returning pointer to static data makes it non-reentrant */
        static char buffer[STRING_SIZE_LIMIT];
        int index;

        for (index = 0; str[index]; index++)
                buffer[index] = toupper(str[index]);
        buffer[index] = '\0';
        return buffer;
}
```

- Re-entrant version of strToUpper

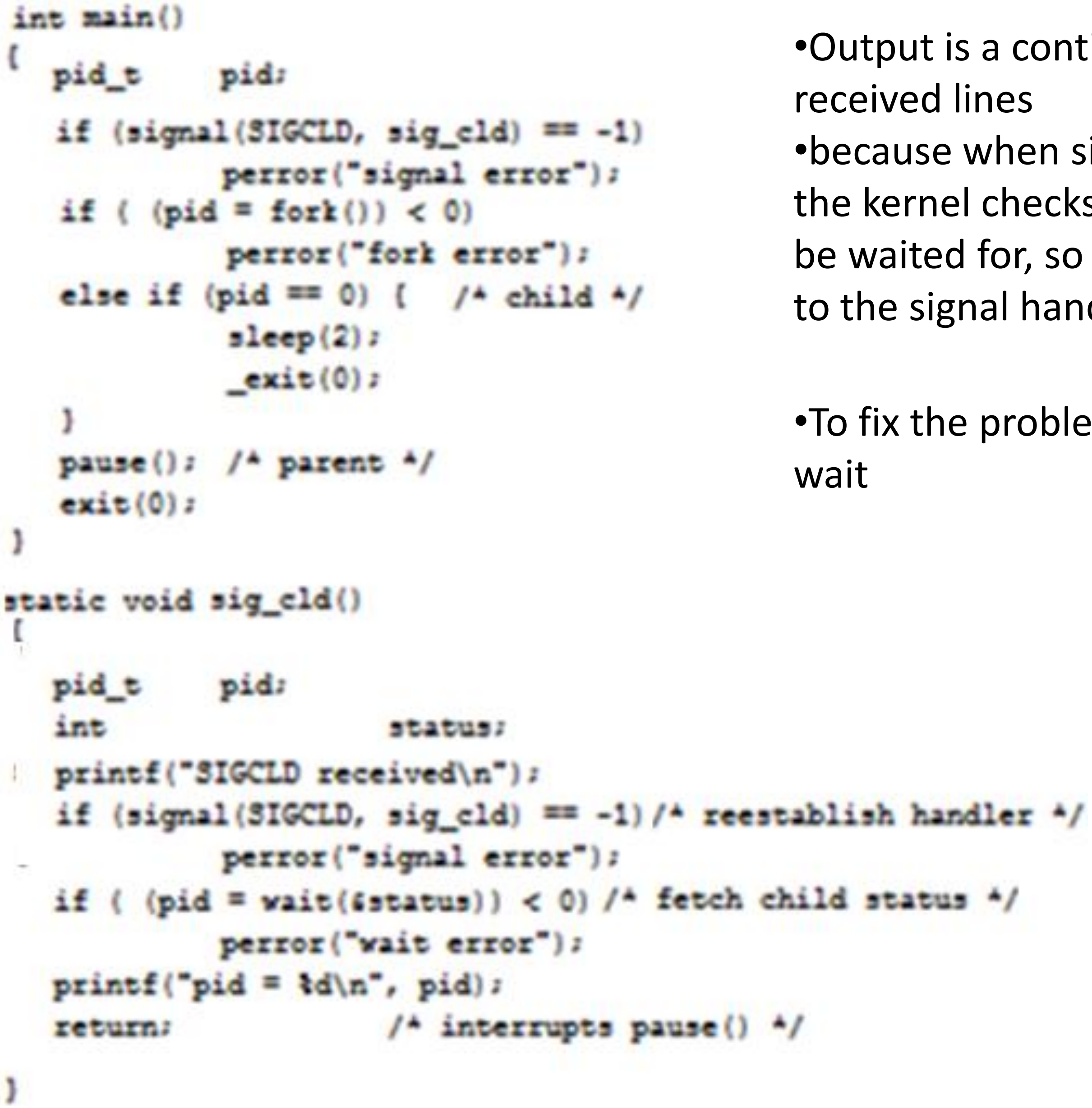


```
char *strToUpper_r(char *in_str, char *out_str)
{
        int index;

        for (index = 0; in_str[index] != '\0'; index++)
        out_str[index] = toupper(in_str[index]);
        out_str[index] = '\0';

        return out_str;
}
```

SIGCLD Semantics

- SIGCHLD – when the signal occurs, the status of the a child has changed and parent needs to call one of the wait functions. Default to ignore
- System V has SIGCLD signal
  - If the disposition is specifically set to SIG_IGN, children of the calling process will not generate zombie process
    - This is different from default action to ignore
    - Instead, on child termination, status is just ignored
  - If the disposition is to catch, kernel immediately checks if there are any child process ready to be waited and if so calls the SIGCLD handler

System V SIGCLD handler that doesn’t work



```
int main()
{
    pid_t     pid;

    if (signal(SIGCLD, sig_cld) == -1)
            perror("signal error");
    if ( (pid = fork()) < 0)
            perror("fork error");
    else if (pid == 0) {   /* child */
            sleep(2);
            _exit(0);
    }
    pause();  /* parent */
    exit(0);
}

static void sig_cld()
{

    pid_t     pid;
    int               status;

    printf("SIGCLD received\n");
    if (signal(SIGCLD, sig_cld) == -1) /* reestablish handler */
            perror("signal error");
    if ( (pid = wait(&status)) < 0) /* fetch child status */
            perror("wait error");
    printf("pid = %d\n", pid);
    return;             /* interrupts pause() */

}
```

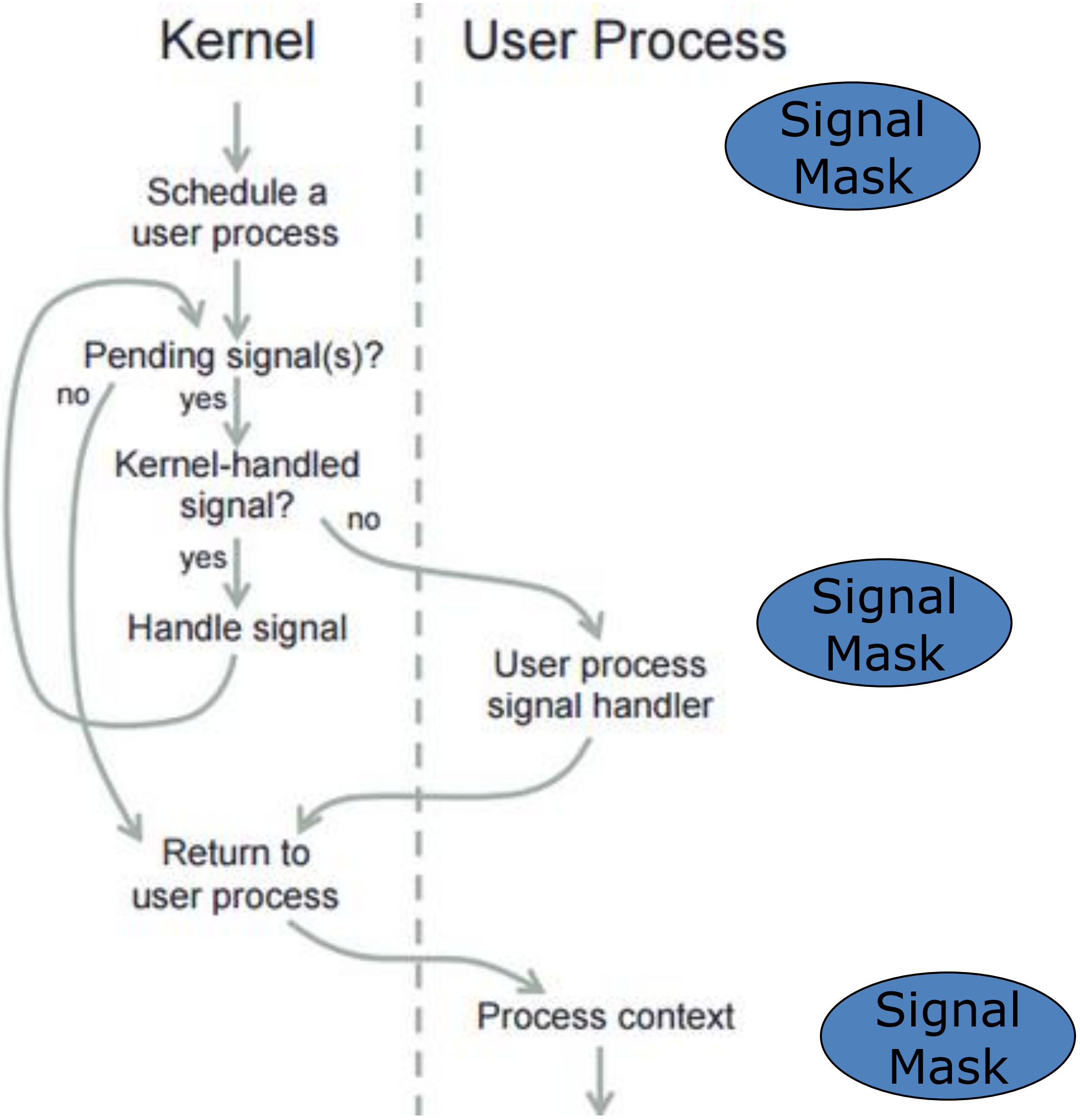
- Output is a continual string of SIGCLD received lines
- because when signal handler is called, the kernel checks whether a child needs to be waited for, so it generates another call to the signal handler

- To fix the problem , move signal after the wait

Reliable Signal Terminology and Semantics

- A signal is generated for a process when the event that causes the signal occurs
- A signal is delivered when the action for the signal is taken
- Time between the generation of the signal and its delivery, a signal is said to be pending
- A process has an option of blocking the delivery of a signal
  - During blocking, the signal is pending
  - If many signals of the *same* type are waiting to be handled (e.g. two SIGINTs), then most UNIXs will only deliver one of them.
    - the others are thrown away
  - If many signals of *different* types are waiting to be handled (e.g. a SIGINT, SIGSEGV, SIGUSR1), they are not delivered in any fixed order.

How Signals Work



kill Function

- int kill(pid_t pid, int signo);

  return: 0 if OK, -1 on error

- *kill* 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 absolute of pid*
- *pid == -1: unspecified (used as a broadcast signal in SVR4, 4.3 + BSD)*

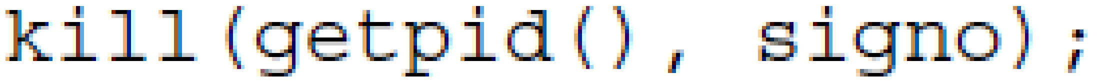
kill Function (contd)

- Permission to send signals
  - Super-user can send a signal to any process.
  - Real or effective user ID of the sender has to equal the real or effective user ID of the receiver
- Signal number 0 or *null* signal used with *kill*, no signal is sent, used to determine if a specific process still exists. If does not exist, *kill* returns -1 with *errno* set to *ESRCH*
- If the call to *kill* causes the signal to be generated for the calling process and if the signal is not blocked, either *signo* or some other pending unblocked signal is delivered to the process before *kill* returns

raise function

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

int raise(int signo);
                              return: 0 if OK, -1 on error
```

- **raise** - function allows a process to send a signal to itself
- Implement raise function using kill



```
kill(getpid(), signo);
```

alarm Function

- unsigned int alarm (unsigned int seconds);

  Returns: 0 or number of seconds until previously set alarm
- alarm() sets a timer to expire at a specified time in future.
  - when timer expires, SIGALRM signal is generated
- Only one alarm clock per process
  - previously registered alarm clock is replaced by the new value
- if alarm(0), a previous unexpired alarm is cancelled
- Default action for *SIGALRM* is to terminate the process
  - Most processes use alarm clock catch the signal
  - If process wants to terminate, performs cleanup before terminates

pause() function

- **`int pause (void);`**

  `Returns: -1 with errno set to EINTR`
- Suspends the calling process until a signal is caught
- Returns only if a signal handler is executed and that handler returns
  - If signal handler is not registered, just quit
  - If signal handler is registered, return after the handler is processed

sleep Function

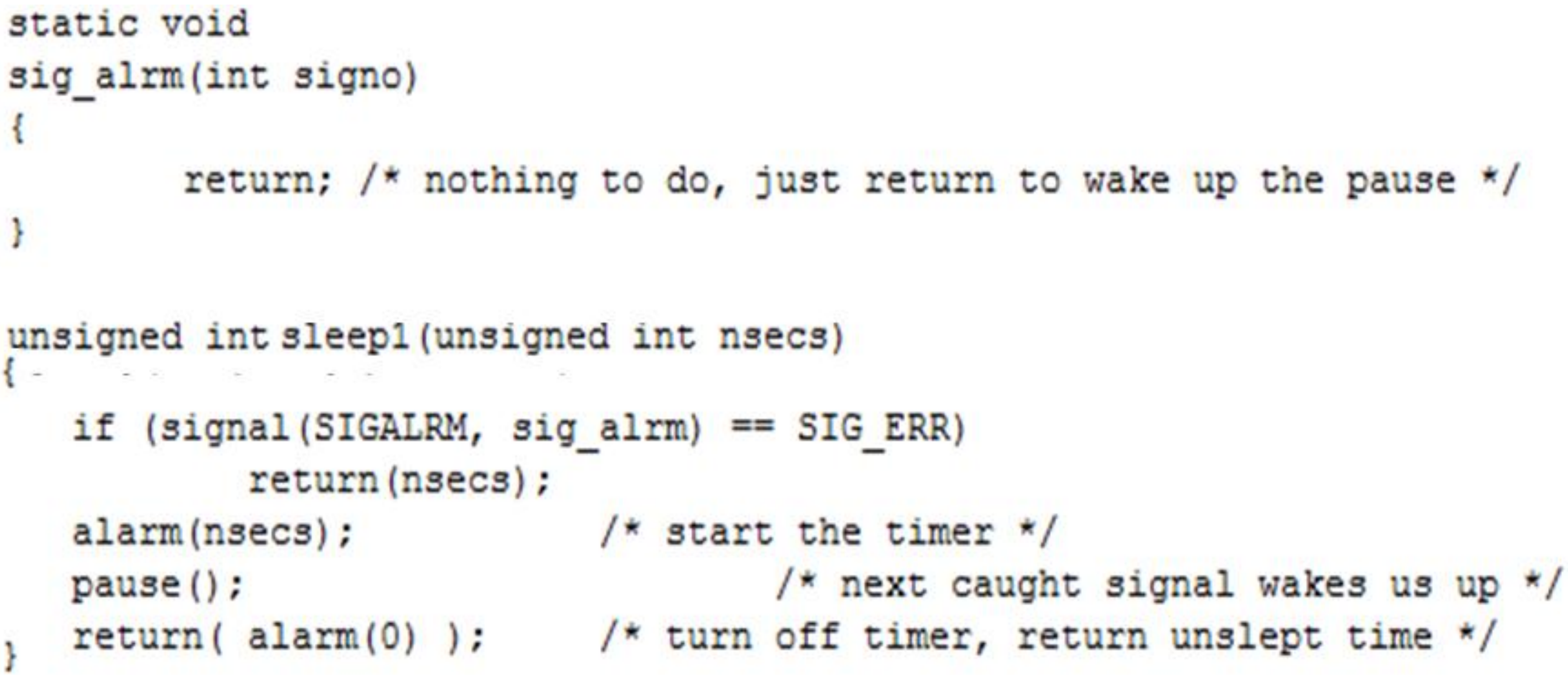
- unsigned int sleep(unsigned int seconds);

  Returns: 0 or number of unslept seconds

- This function causes the calling process to be suspended until either

  1. The amount of wall clock time specified by seconds has elapsed
     - Returns 0
  2. A signal is caught by the process and the signal handler returns
     - Returns number of unslept seconds

Simple, Incomplete Implementation of sleep

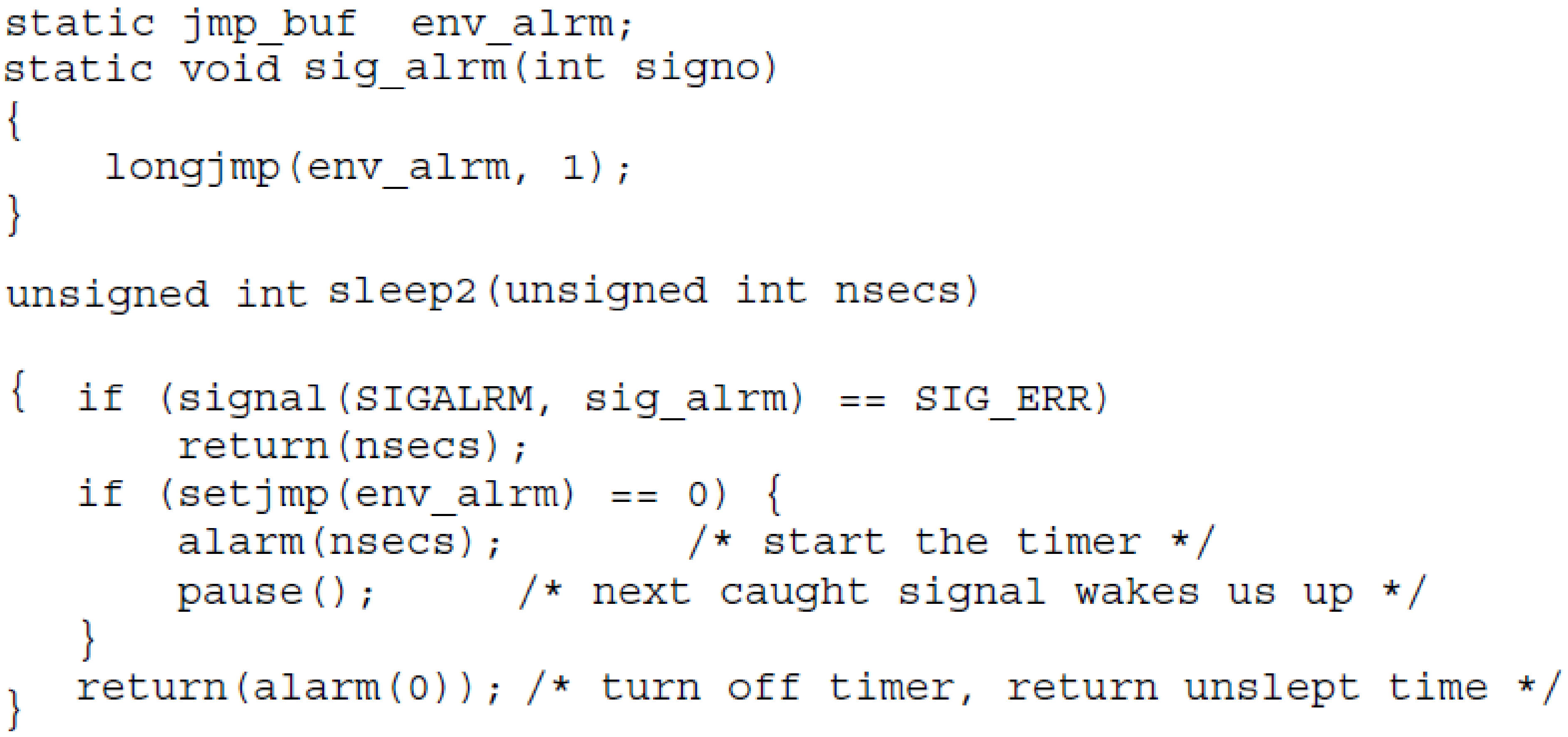


```
static void
sig_alrm(int signo)
{
        return; /* nothing to do, just return to wake up the pause */
}

unsigned int sleep1(unsigned int 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 */
}
```

Problems with sleep Implementation

- If the caller has already an alarm set, that alarm is erased by the first call to alarm. Solution is
  - Look at the return value from the first call to alarm
  - If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the previously set alarm expires
  - If the previously set alarm will go off after ours, then before returning, we should reset this alarm to occur at its designated time in the future
- Modified the disposition of SIGALRM
  - we should save the disposition and restore it
- Race condition between first call to alarm and pause
- If the alarm expires before executing pause, the system will be in pause until another signal occurs

Another (imperfect) implementation of sleep (SVR2)

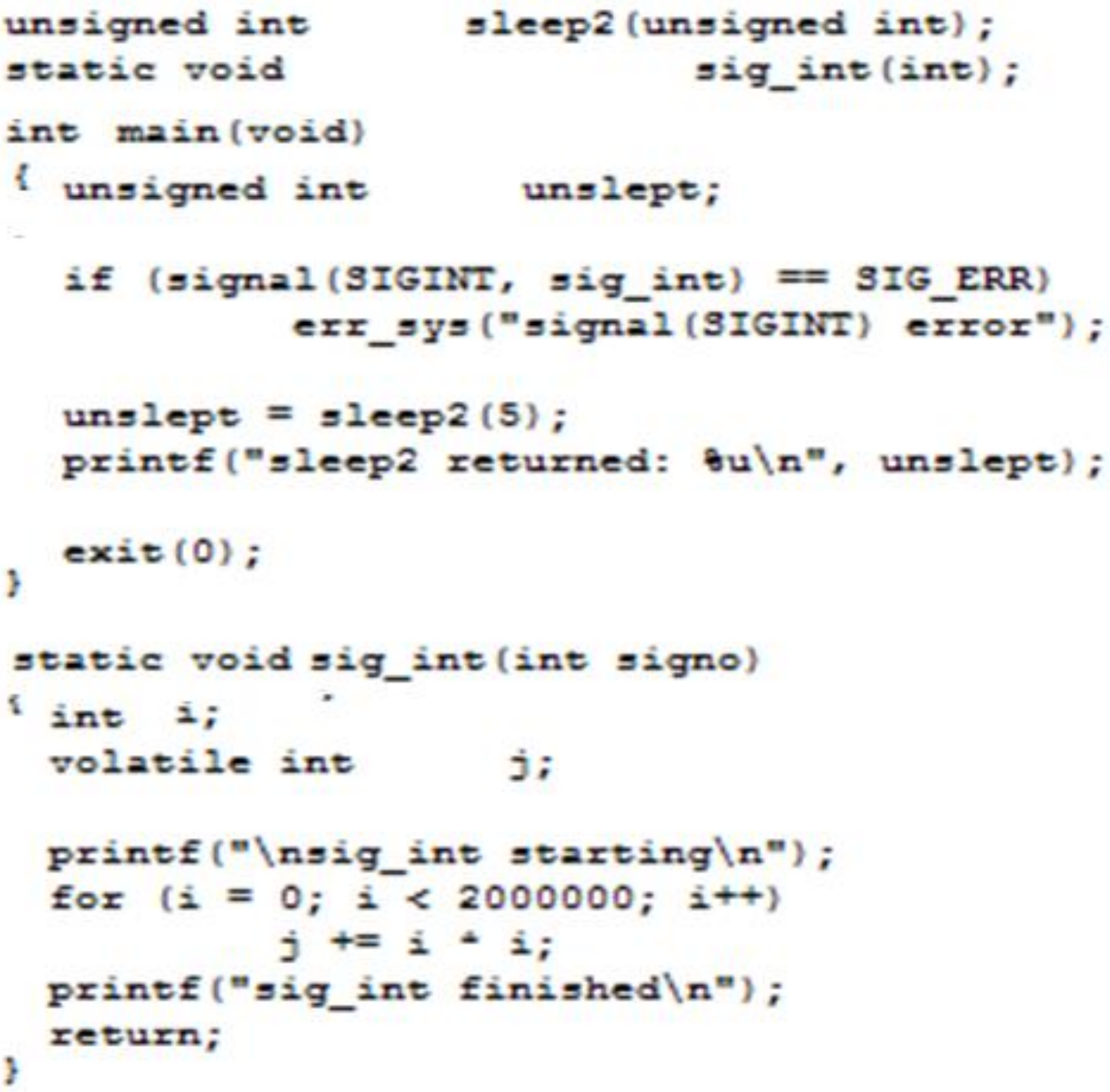


```
static jmp_buf  env_alrm;
static void sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

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 */
}
```

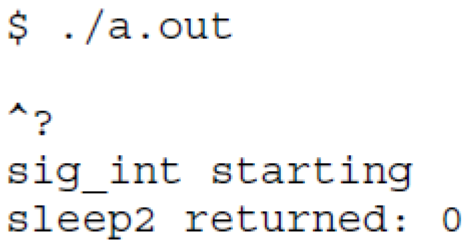
Problem with implementation

- Race condition is eliminated
- Interaction with other signals
  - If SIGALRM interrupts some other signal handler, when longjmp is called, it aborts the other signal handler

Calling sleep2 from a program that catches other signals

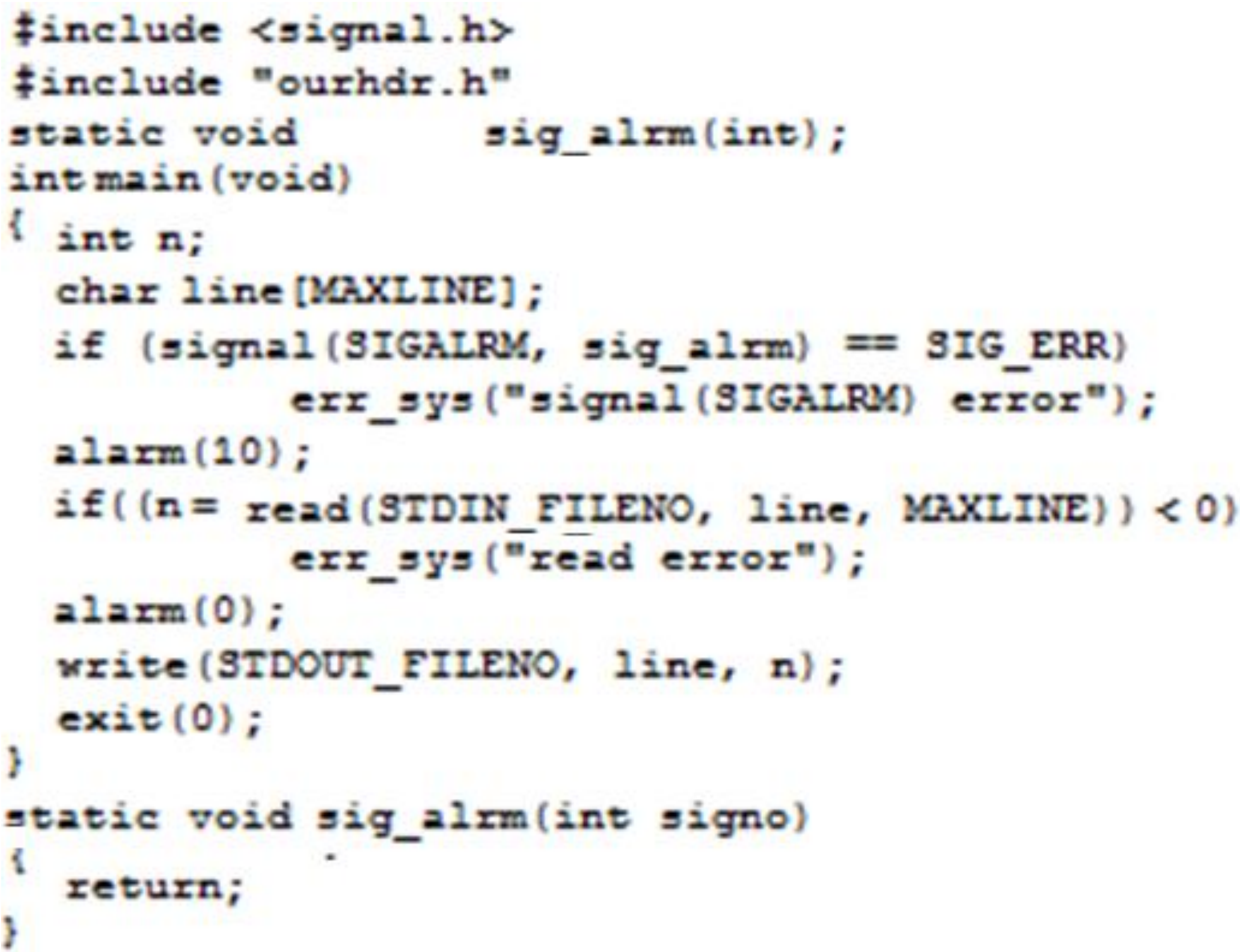


```
unsigned int       sleep2(unsigned int);
static void                  sig_int(int);

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)
{ 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

^?
sig_int starting
sleep2 returned: 0
```

Alarm to Put Upper Limit on read()

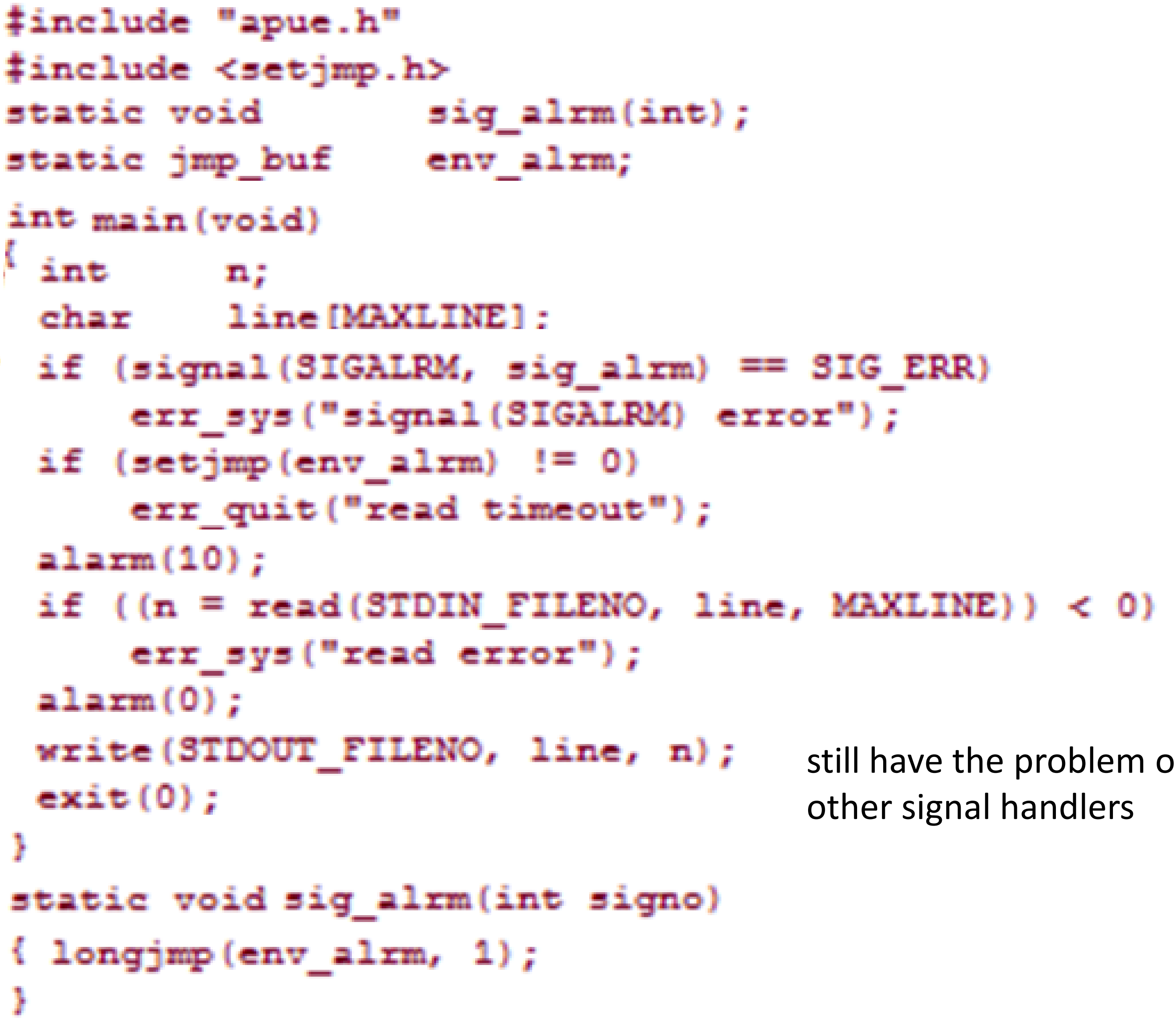


```c
#include <signal.h>
#include "ourhdr.h"
static void        sig_alrm(int);
int main(void)
{  int n;
   char line[MAXLINE];
   if (signal(SIGALRM, sig_alrm) == SIG_ERR)
            err_sys("signal(SIGALRM) error");
   alarm(10);
   if((n= read(STDIN_FILENO, line, MAXLINE)) < 0)
            err_sys("read error");
   alarm(0);
   write(STDOUT_FILENO, line, n);
   exit(0);
}
static void sig_alrm(int signo)
{  return;
}
```

Alarm to Put Upper Limit on read()

- Race condition between alarm and read
  - If the kernel blocks the process between these two function calls for longer than the alarm period, the read could block forever.
- If read is automatically restarted, read does not get interrupted when SIGALRM signal handler returns

Calling read with a timeout, using longjmp

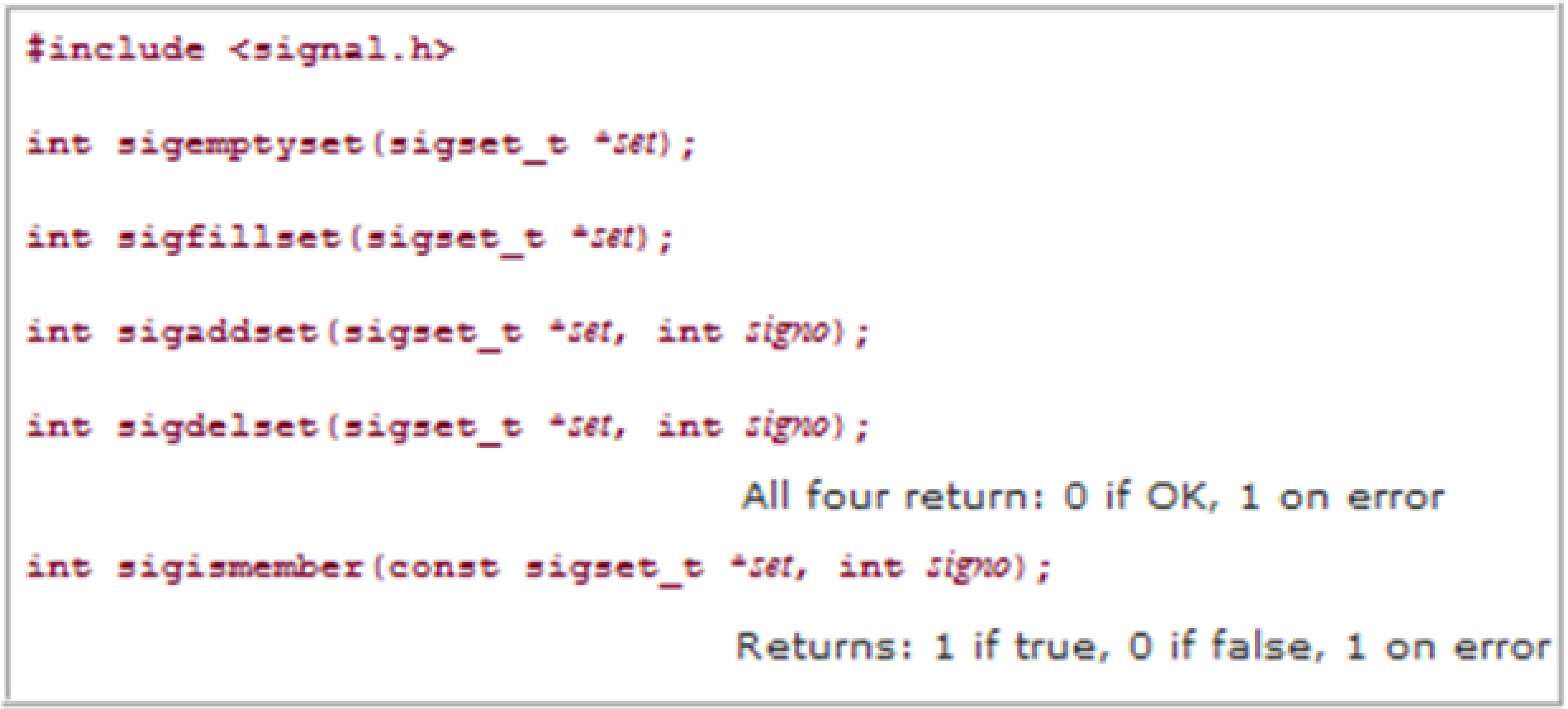


```
#include "apue.h"
#include <setjmp.h>
static void       sig_alrm(int);
static jmp_buf    env_alrm;

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);
}
```

still have the problem of interactions with other signal handlers

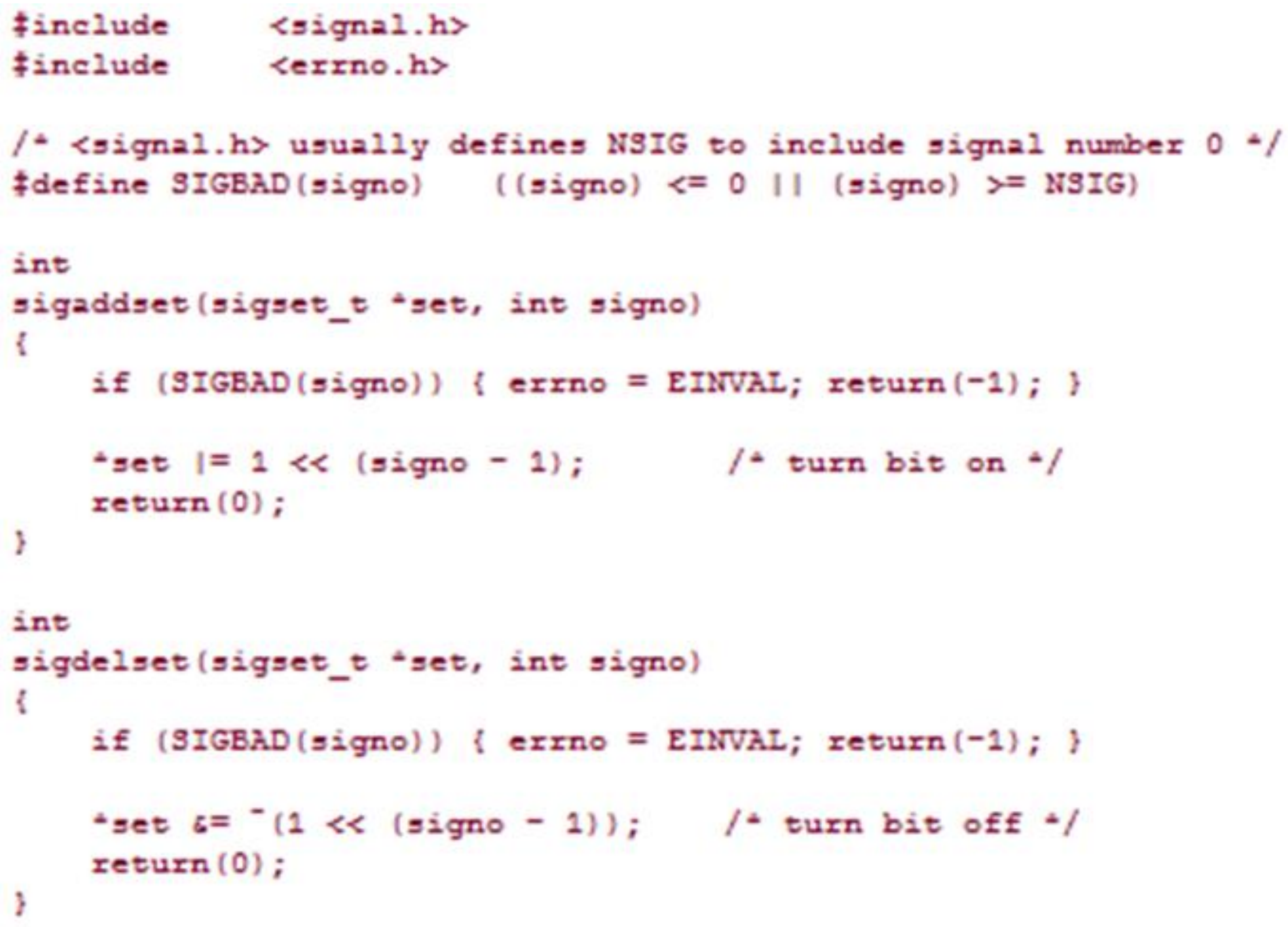
Signal Sets

- Posix.1 defines the data type *sigset_t* to represent multiple signals since *int* data type is insufficient to hold all signals
- Deals with pending signals that might otherwise be missed while a signal is being processed
- POSIX contains several functions for creating, changing and examining signal sets.



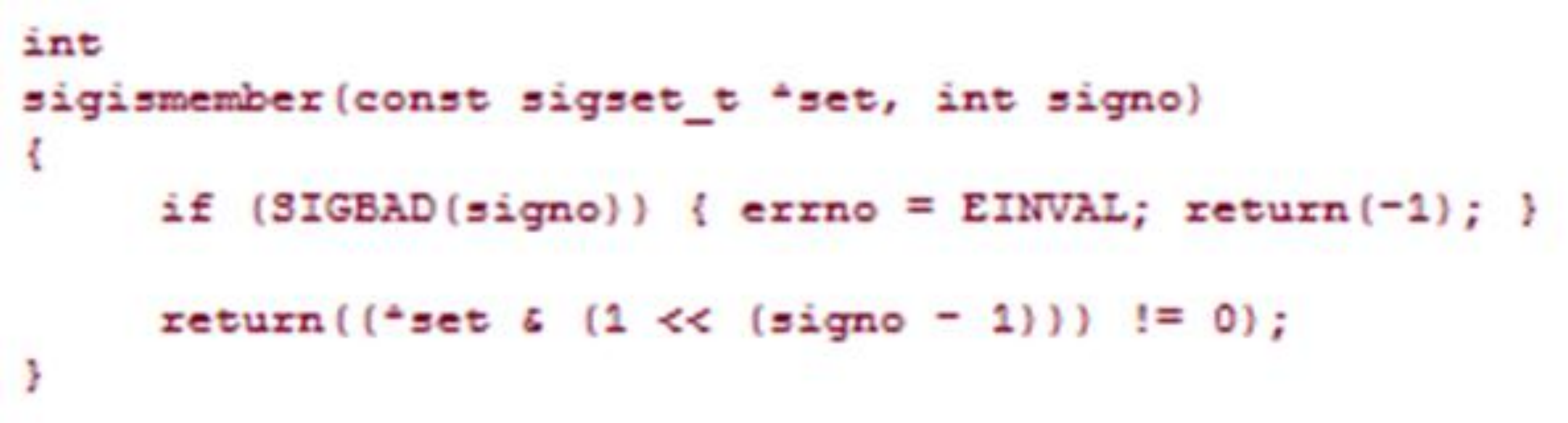
```
#include <signal.h>

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
```

sigaddset, sigdelset



```
#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)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

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

sigismember



```
int
sigismember(const sigset_t *set, int signo)
{
     if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

     return((*set & (1 << (signo - 1))) != 0);
}
```

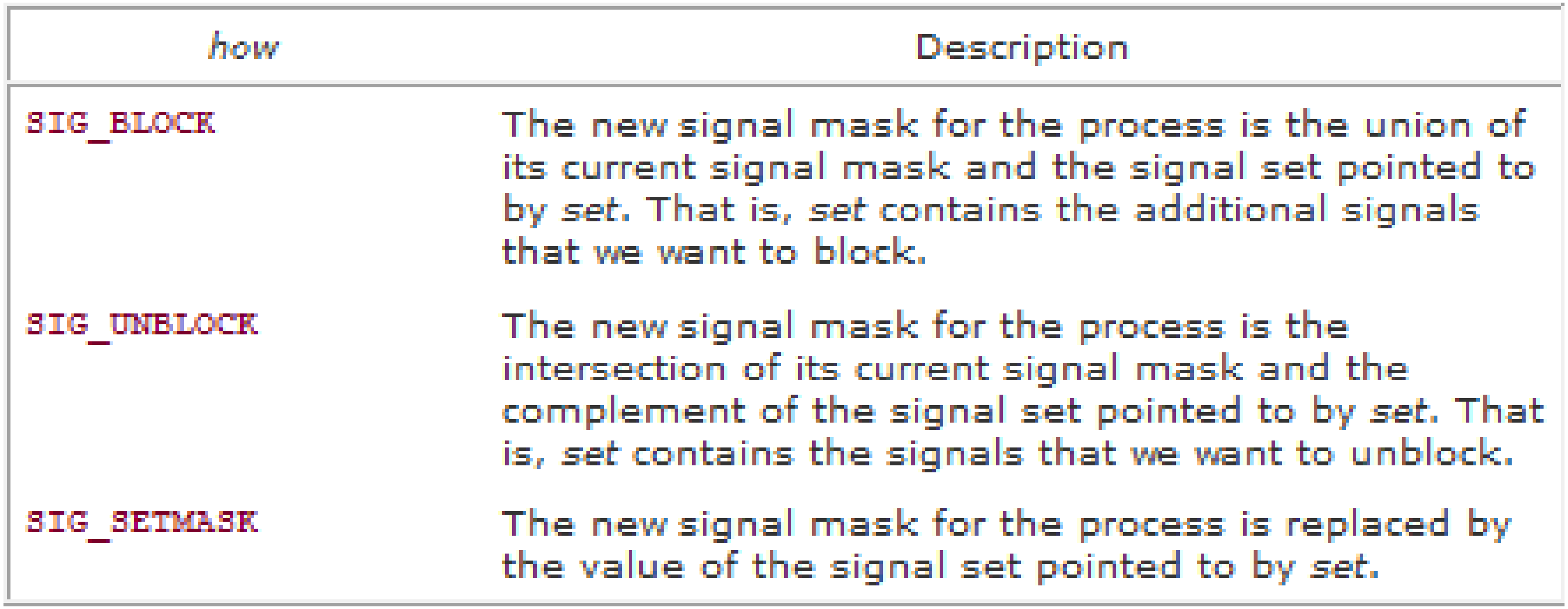
Signal Blocking

- A Process can temporarily prevent signal from being delivered by *blocking* it
- **Important!** Blocking a signal is different from ignoring signal.
  - When a process blocks a signal, the OS does not deliver signal until the process unblocks the signal
  - When a process ignores signal, signal is delivered and the process handles it by throwing it away

sigprocmask() function

- *Signal Mask* contains a set of signals which are currently ***blocked***
- A process can examine or change its signal mask by calling *sigprocmask*
- int sigprocmask(int how, sigset_t *restrict set, sigset_t *restrict oset);

returns: 0 if OK, -1 on error

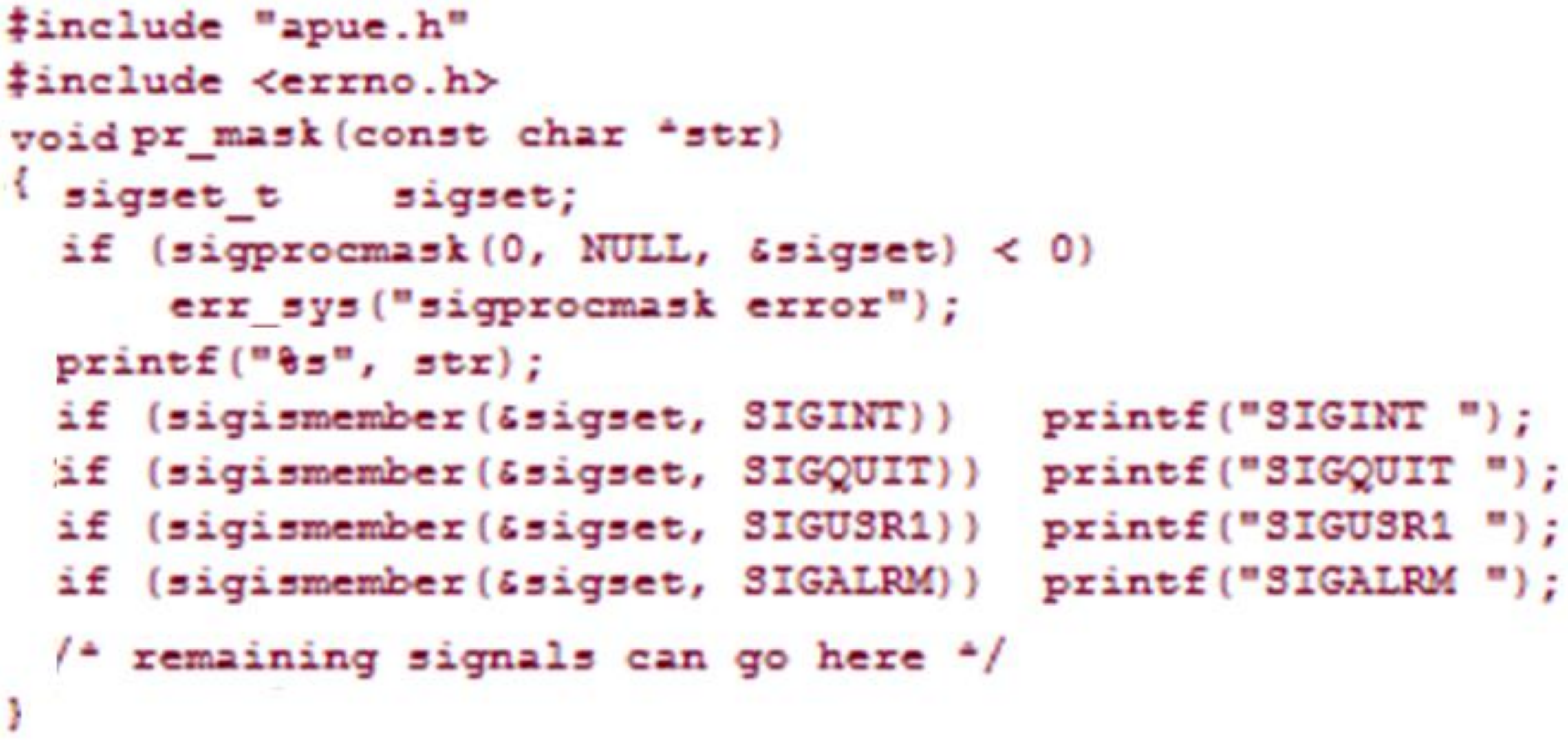


how	Description
SIG_BLOCK	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by *set*. That is, *set* contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by *set*. That is, *set* contains the signals 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*.

sigprocmask() function

- int sigprocmask(int how, sigset_t *restrict set, sigset_t *restrict oset);
- If *oset* is not null, *oset* returns current mask
- If *set* is null, signal mask is not changed
- If there are any pending, unblocked signals after the call to *sigprocmask*, atleast one of the signals is delivered to the process before *sigprocmask* returns

Prints the signal mask for the process



```
#include "apue.h"
#include <errno.h>
void pr_mask(const char *str)
{ sigset_t    sigset;
  if (sigprocmask(0, NULL, &sigset) < 0)
      err_sys("sigprocmask error");
  printf("%s", str);
  if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
  if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
  if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
  if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");
  /* remaining signals can go here */
}
```

A Critical Code Region

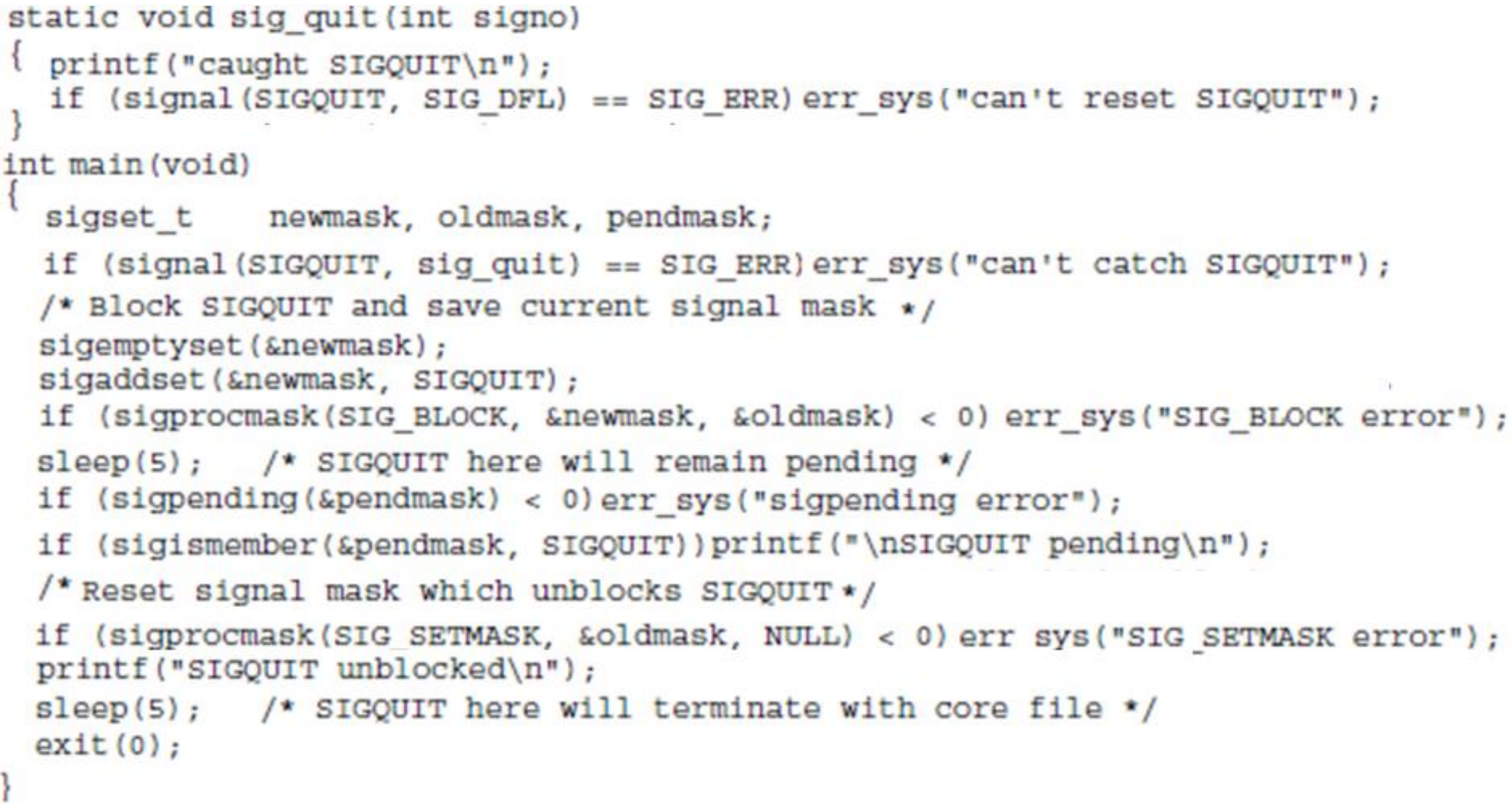
```
sigset_t newmask, oldmask;

sigemptyset( &newmask );
sigaddset( &newmask, SIGINT );


/* block SIGINT; save old mask */
sigprocmask( SIG_BLOCK, &newmask, &oldmask );

/* critical region of code */

/* reset mask which unblocks SIGINT */
sigprocmask( SIG_SETMASK, &oldmask, NULL );
```

sigpending() Function

- Returns the set of signals that are blocked from delivery and currently pending for the calling process

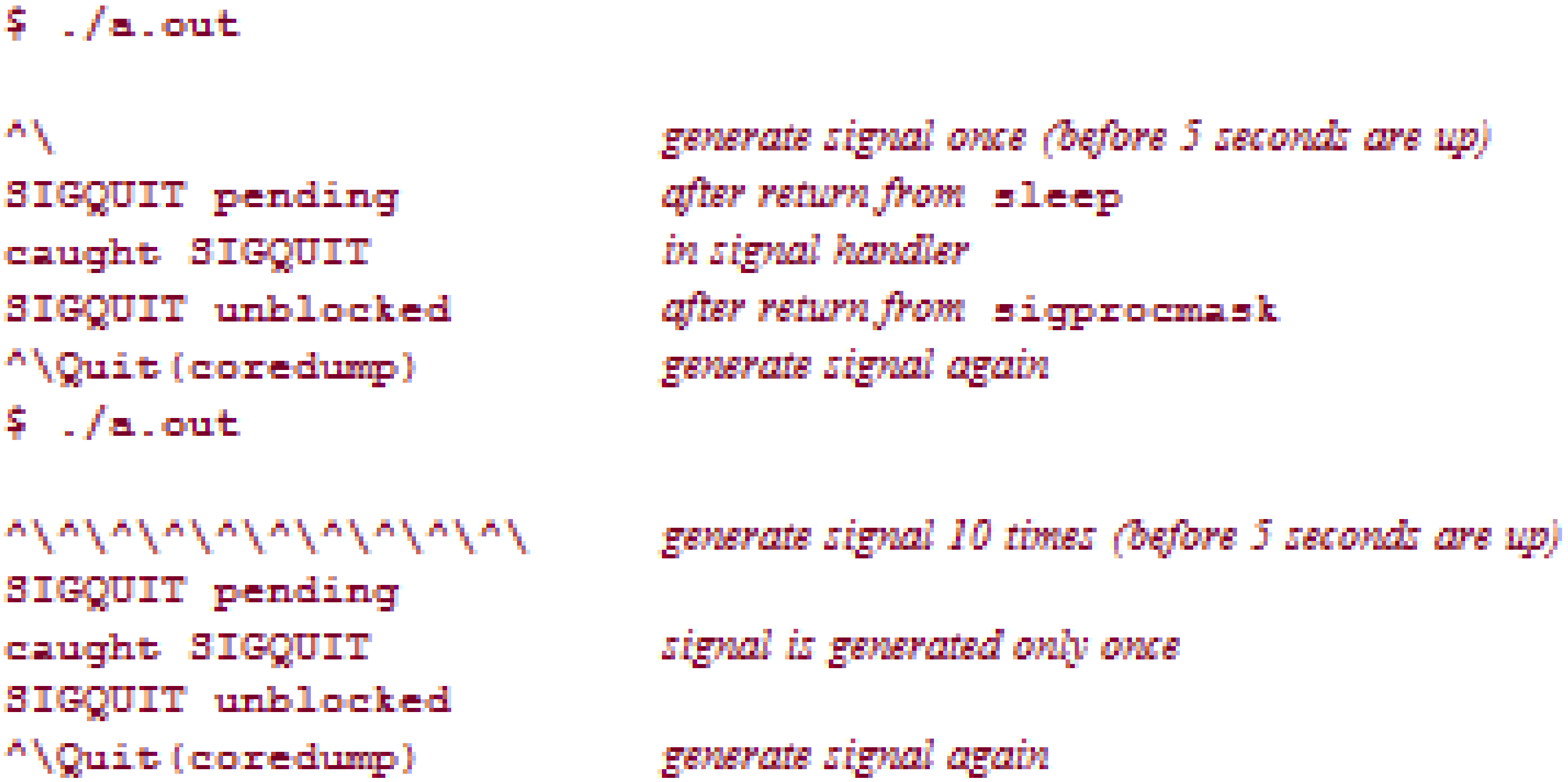
```
int sigpending(sigset_t *set);
                    Returns: 0 if OK, -1 on error
```

Example of signal sets and sigprocmask



```
static void sig_quit(int signo)
{ printf("caught SIGQUIT\n");
  if (signal(SIGQUIT, SIG_DFL) == SIG_ERR) err_sys("can't reset SIGQUIT");
}
int main(void)
{
  sigset_t    newmask, oldmask, pendmask;
  if (signal(SIGQUIT, sig_quit) == SIG_ERR) err_sys("can't catch SIGQUIT");
  /* Block SIGQUIT and save current signal mask */
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGQUIT);
  if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) err_sys("SIG_BLOCK error");
  sleep(5);    /* SIGQUIT here will remain pending */
  if (sigpending(&pendmask) < 0) err_sys("sigpending error");
  if (sigismember(&pendmask, SIGQUIT)) printf("\nSIGQUIT pending\n");
  /* Reset signal mask which unblocks SIGQUIT */
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) err sys("SIG_SETMASK error");
  printf("SIGQUIT unblocked\n");
  sleep(5);    /* SIGQUIT here will terminate with core file */
  exit(0);
}
```

Example (Contd)



```
$ ./a.out

^\                              generate signal once (before 5 seconds are up)
SIGQUIT pending                 after return from  sleep
caught SIGQUIT                  in signal handler
SIGQUIT unblocked               after return from  sigprocmask
^\Quit(coredump)                generate signal again
$ ./a.out

^\^\^\^\^\^\^\^\^\^\            generate signal 10 times (before 5 seconds are up)
SIGQUIT pending
caught SIGQUIT                  signal is generated only once
SIGQUIT unblocked
^\Quit(coredump)                generate signal again
```

- OS includes the signal being delivered in the signal mask when the handler is invoked
- Hence, guaranteed that whenever processing a given signal, another occurrence of that same signal is blocked

sigaction() Function

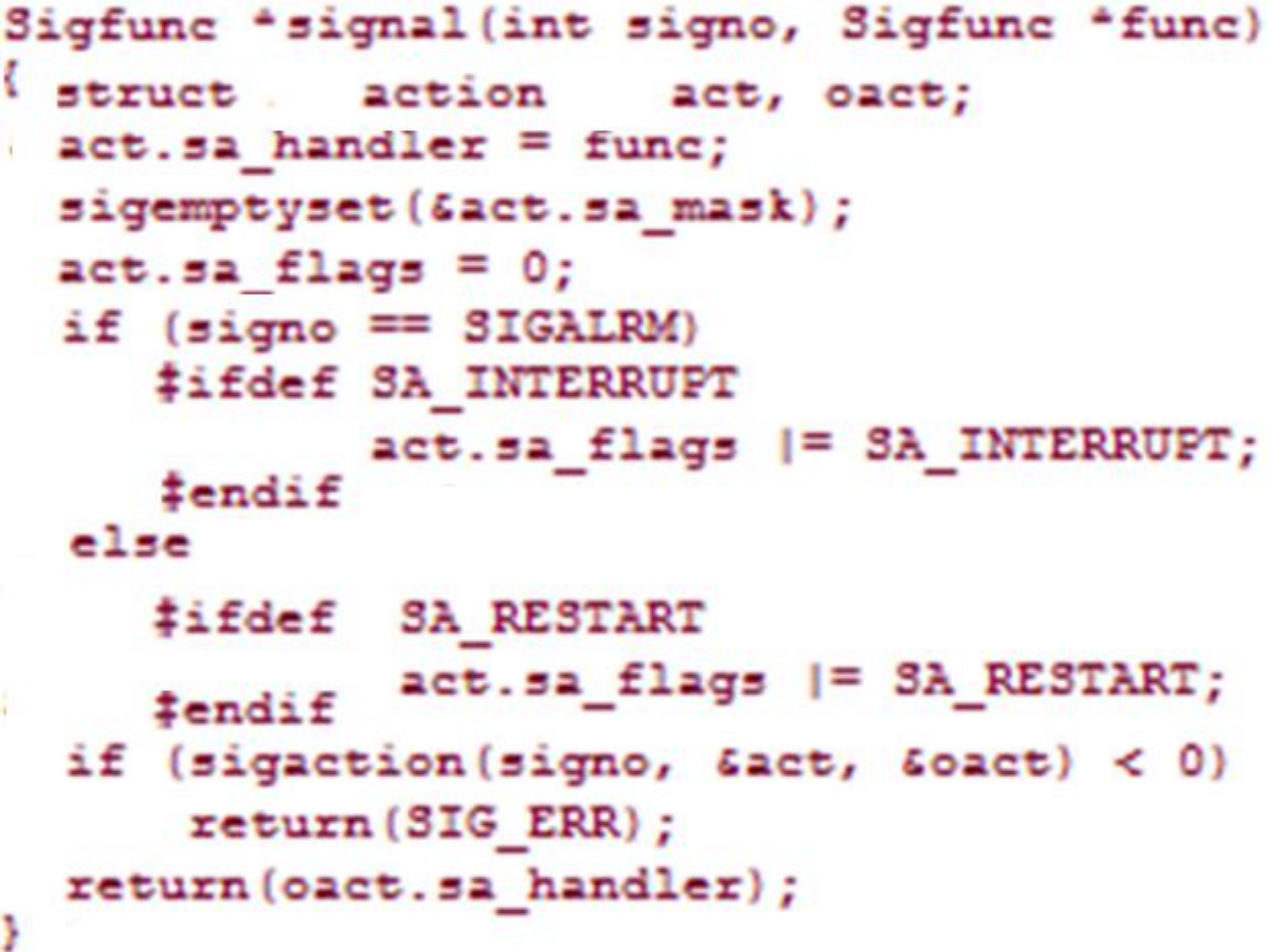
- To overcome the deficiencies of signal function:
  - signal() function does not block other signals from arriving while the current handler is executing
  - In earlier systems, signal() function resets the signal action back to SIG_DFL for almost all signals
- sigaction function allows the caller to examine or modify or specify action associated with a specific signal
- Program installs *signal handler* by calling sigaction with the name of a user-written function
- Allow blocking of additional signals while execution of a signal handler

sigaction() Function

```
int sigaction(int signo, struct sigaction *act, struct sigaction *oact);
        Returns: 0 if OK, -1 on error


struct sigaction {
   void (*sa_handler)(int); /* signal handler/SIG_IGN/SIG_DFL */
   sigset_t      sigset_t sa_mask;     /* additional signals to block */
   int sa_flags;                      /* signal options */
};
```

- Either **act** or **oact** may be NULL.
- sa_flags – SA_RESTART: interrupted system call is automatically restarted
- A signo signal causes the sa_handler signal handler to be called
- While sa_handler executes, the signals in sa_mask are blocked in addition to signo
- sa_handler remains installed until it is changed by another sigaction() call. No reset problem.

Implementation of *signal* using *sigaction*



```
Sigfunc *signal(int signo, Sigfunc *func)
{  struct     action     act, oact;
   act.sa_handler = func;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   if (signo == SIGALRM)
        #ifdef SA_INTERRUPT
                act.sa_flags |= SA_INTERRUPT;
        #endif
    else
        #ifdef  SA_RESTART
                 act.sa_flags |= SA_RESTART;
        #endif
    if (sigaction(signo, &act, &oact) < 0)
         return(SIG_ERR);
    return(oact.sa_handler);
}
```

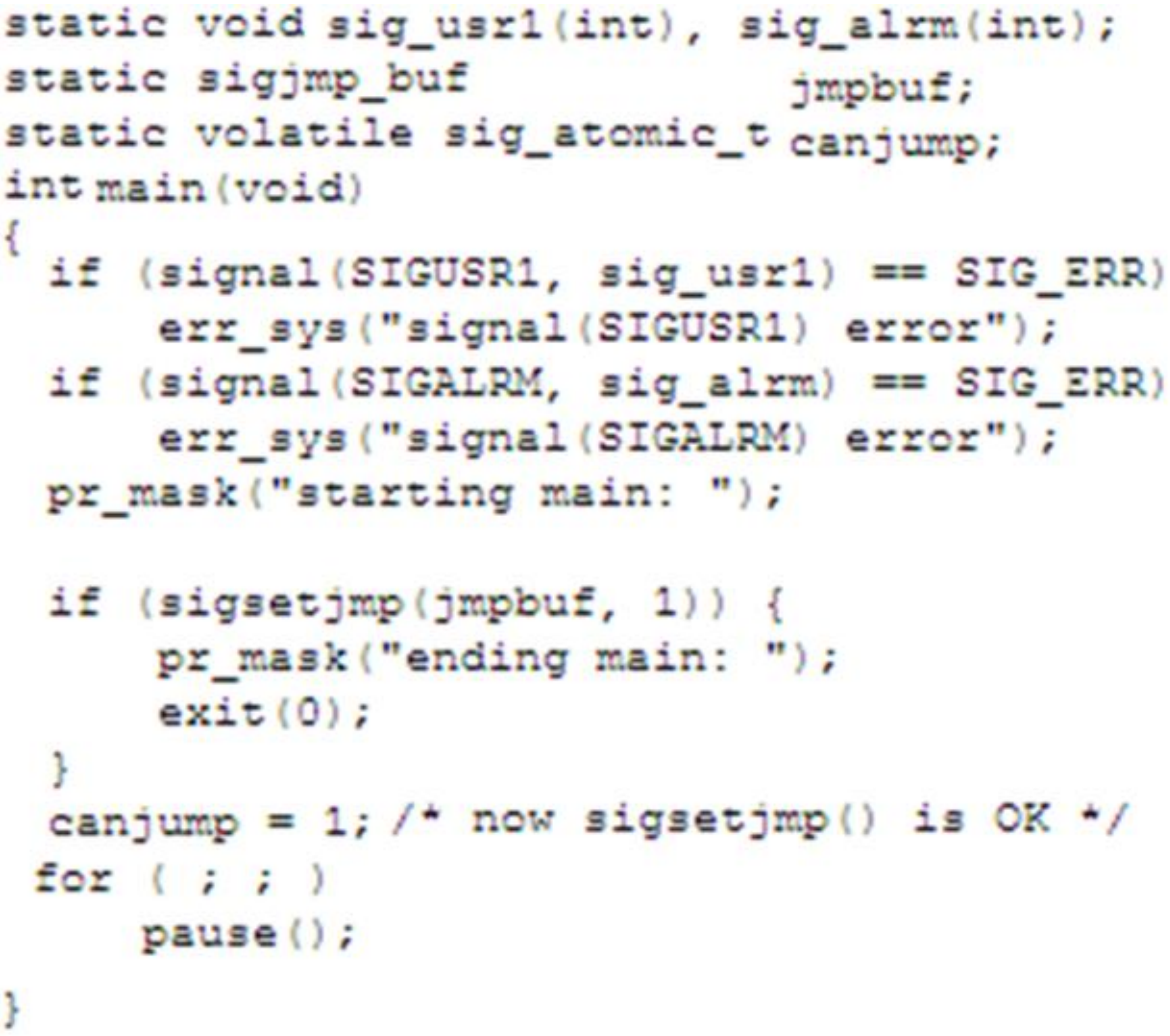
sigsetjmp and siglongjmp Functions

- *longjmp* is often called from a signal handler to return to main loop, instead of returning from the handler
- Problem with *longjmp*
  - When signal is caught, the signal catching function is entered with the current signal automatically being added to signal mask, if *longjmp*, what happens to the signal mask for the process?
  - POSIX.1 does not specify the effect of *setjmp* and *longjmp* on signal masks
- int sigsetjmp(sigjmp_buf env, int savemask);

  Returns: 0 if called directly,

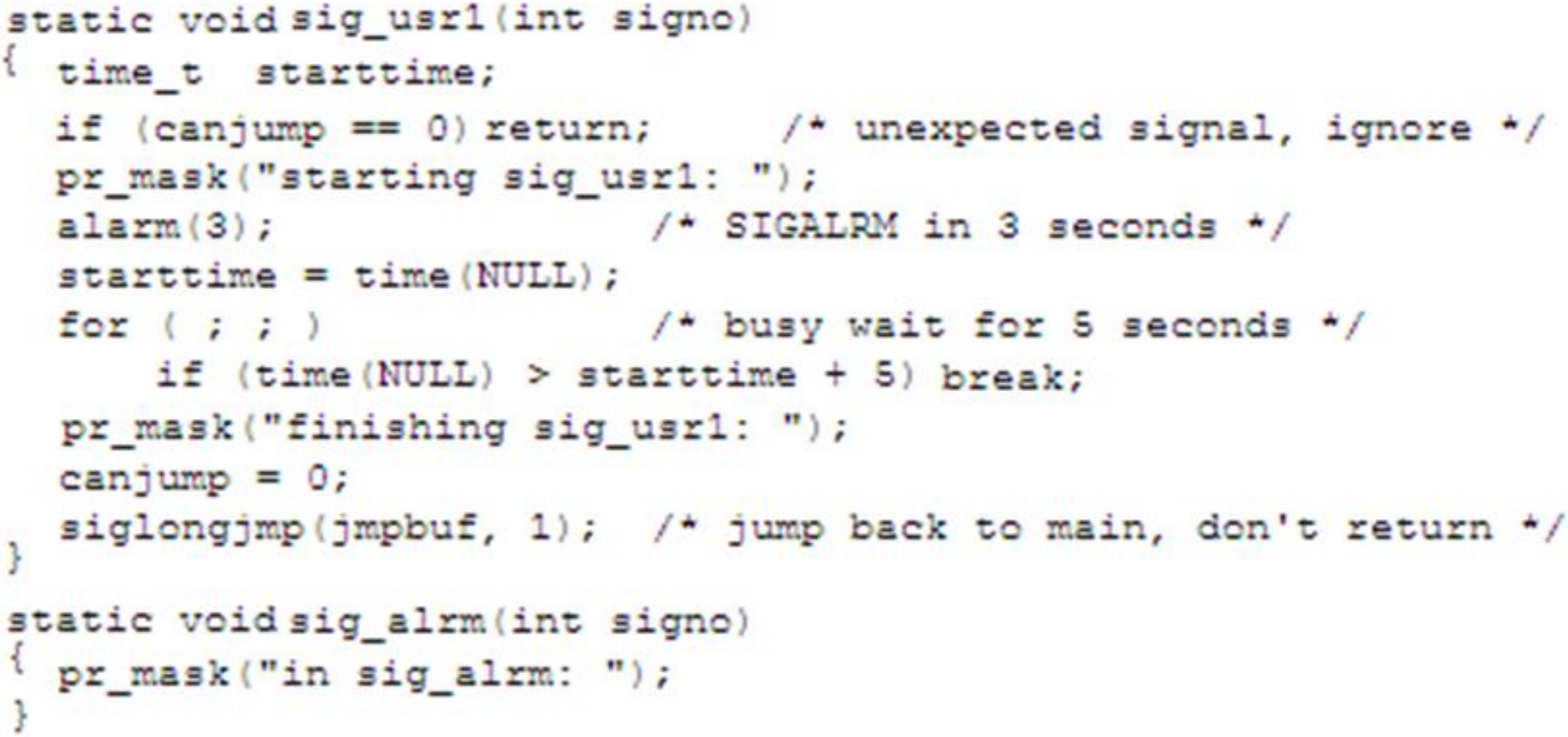
  nonzero if returning from a call to siglongjmp
- If *savemask* is nonzero, the *sigsetjmp* saves the current mask of the process in *env*
- void siglongjmp(sigjmp_buf env, int val);
- When *siglongjmp* is called, *siglongjmp* restores the mask

Example -sigsetjmp and siglongjmp



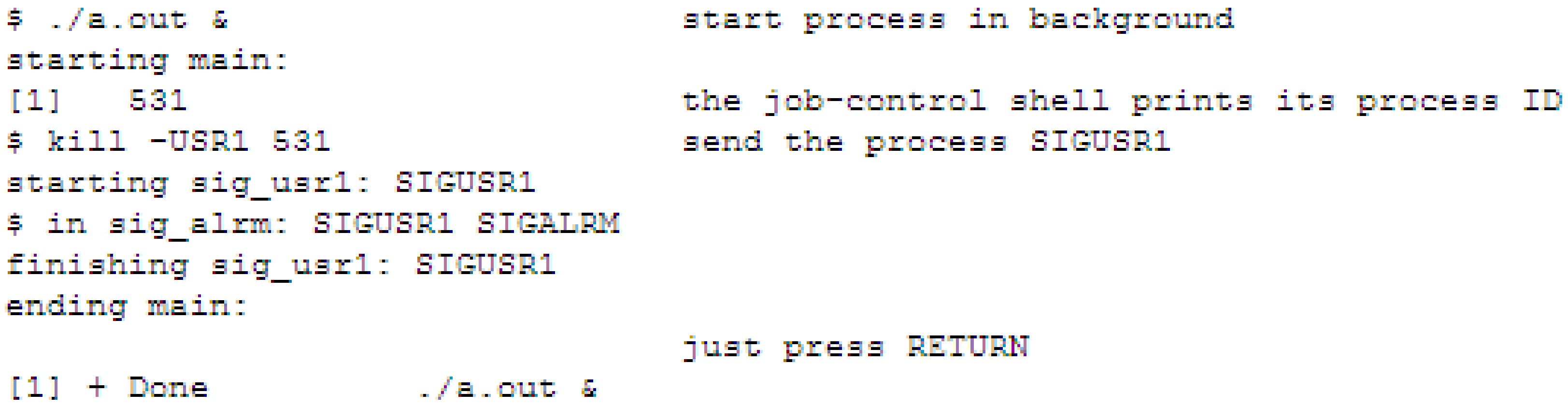
```
static void sig_usr1(int), sig_alrm(int);
static sigjmp_buf                jmpbuf;
static volatile sig_atomic_t canjump;
int main(void)
{
  if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
      err_sys("signal(SIGUSR1) error");
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
      err_sys("signal(SIGALRM) error");
  pr_mask("starting main: ");

  if (sigsetjmp(jmpbuf, 1)) {
      pr_mask("ending main: ");
      exit(0);
  }
  canjump = 1; /* now sigsetjmp() is OK */
 for ( ; ; )
     pause();

}
```

Example Contd...

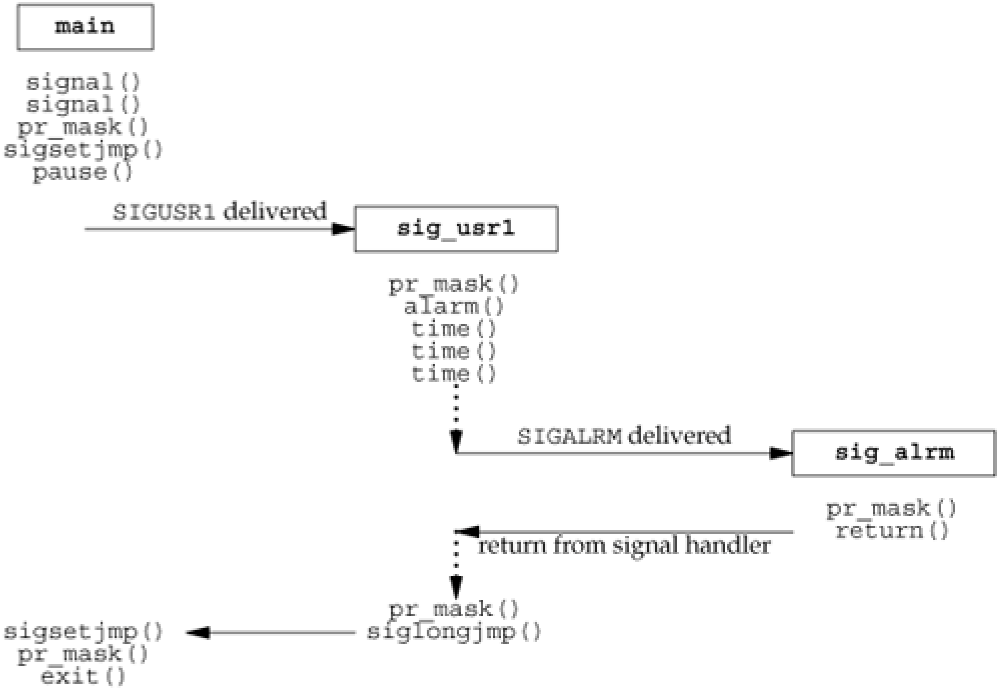


```
static void sig_usr1(int signo)
{  time_t  starttime;
   if (canjump == 0) return;      /* unexpected signal, ignore */
   pr_mask("starting sig_usr1: ");
   alarm(3);                      /* SIGALRM in 3 seconds */
   starttime = time(NULL);
   for ( ; ; )                    /* busy wait for 5 seconds */
       if (time(NULL) > starttime + 5) break;
   pr_mask("finishing sig_usr1: ");
   canjump = 0;
   siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void sig_alrm(int signo)
{  pr_mask("in sig_alrm: ");
}
```

Execution



```
$ ./a.out &                       start process in background
starting main:
[1]    531                        the job-control shell prints its process ID
$ kill -USR1 531                  send the process SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                  just press RETURN
[1] + Done          ./a.out &
```



Protecting a Critical Region from a Signal

After the critical region, if we want to unblock a signal and then pause, waiting for the previously blocked signal to occur

```
sigset_t newmask, oldmask;

sigemptyset( &newmask );
sigaddset( &newmask, SIGINT );

/* block SIGINT; save old mask */
sigprocmask( SIG_BLOCK, &newmask, &oldmask );

/* critical region of code */

/* reset mask which unblocks SIGINT */
sigprocmask( SIG_SETMASK, &oldmask, NULL );
pause();
```

If the signal occurs between the unblocking and the pause, the signal is lost and the pause will block indefinitely

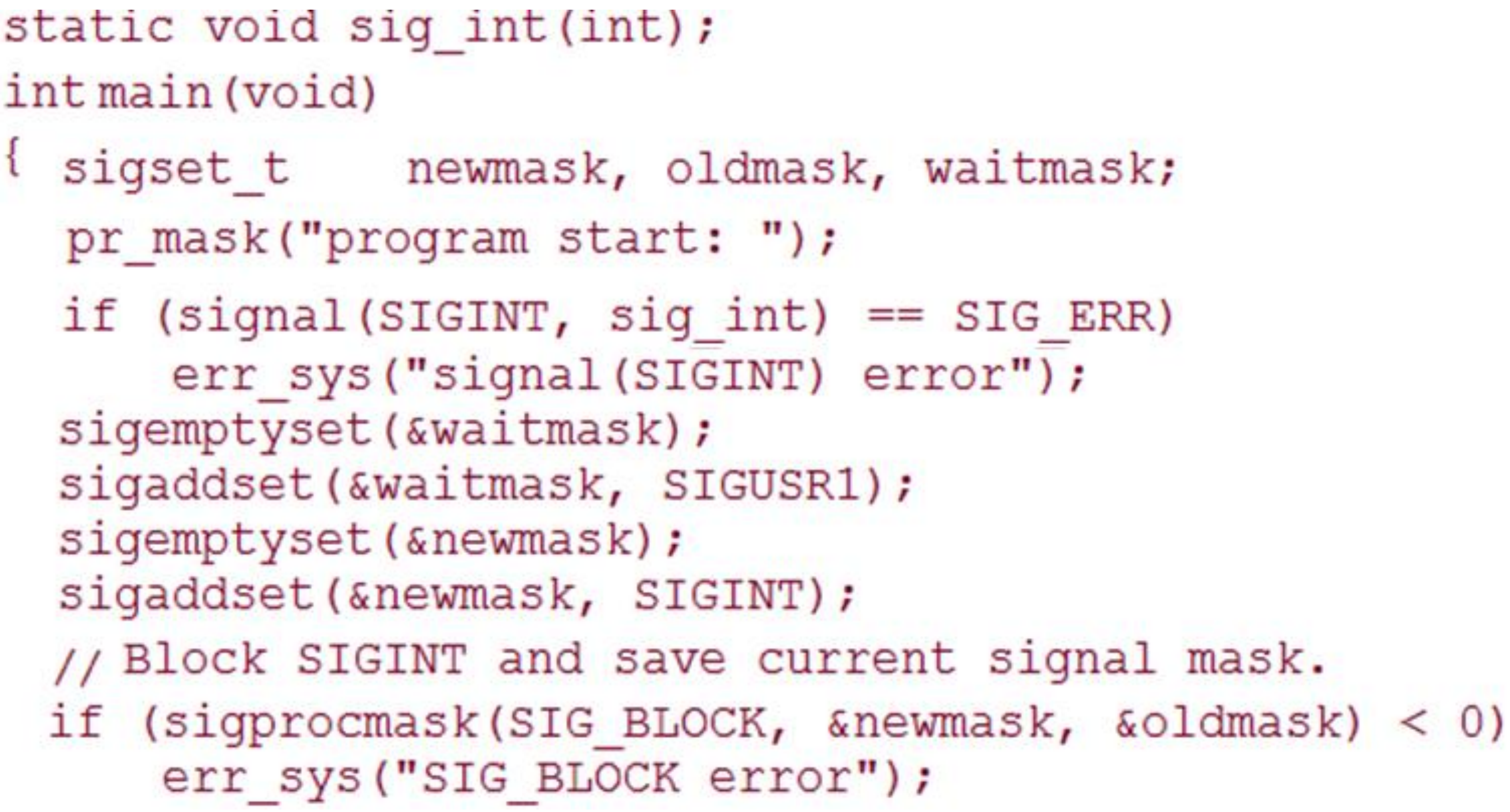
sigsuspend Function

- sigsuspend resets the signal mask and puts the process to sleep as an atomic operation
- int sigsuspend(sigset_t *sigmask);

  Returns: -1 with errno set to EINTR

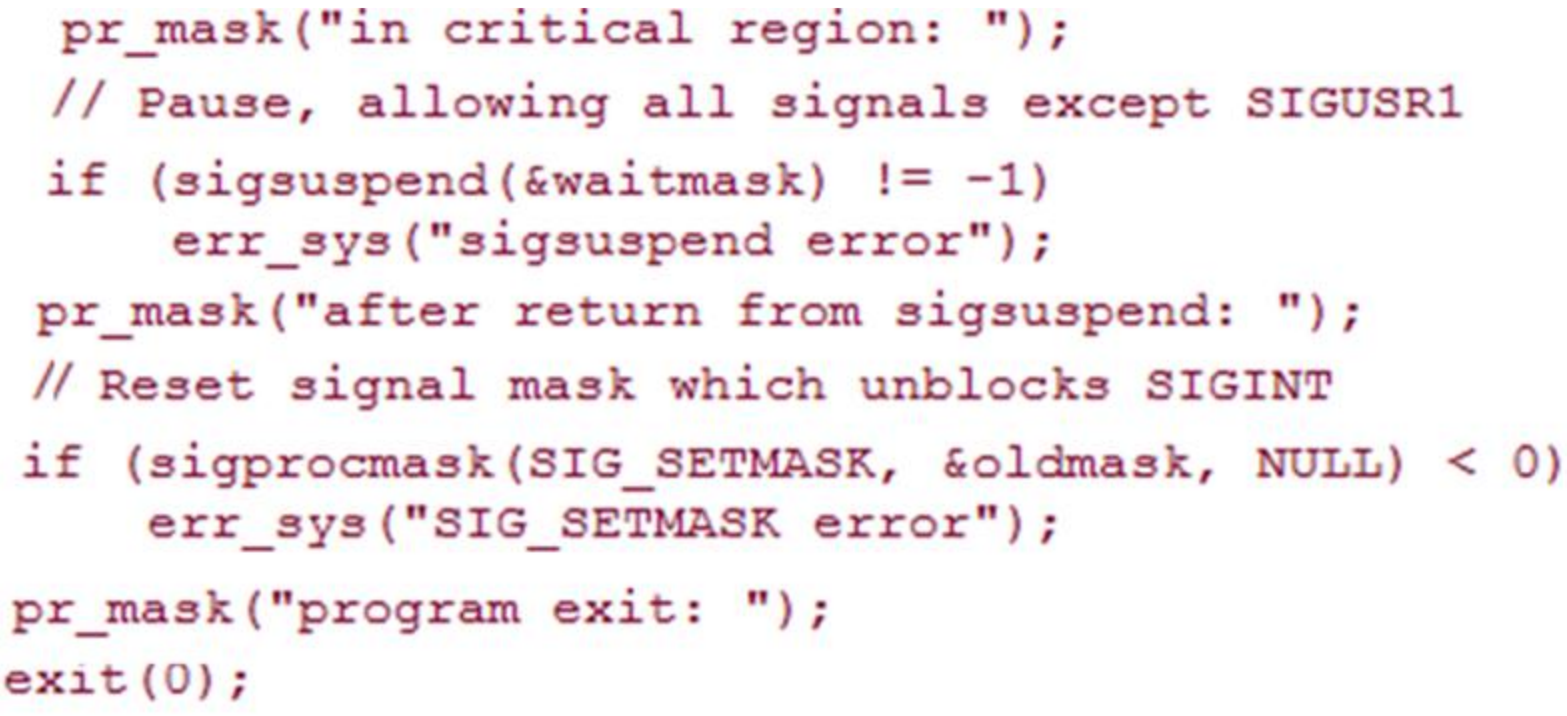
Protecting a Critical Region from a signal

- correct way

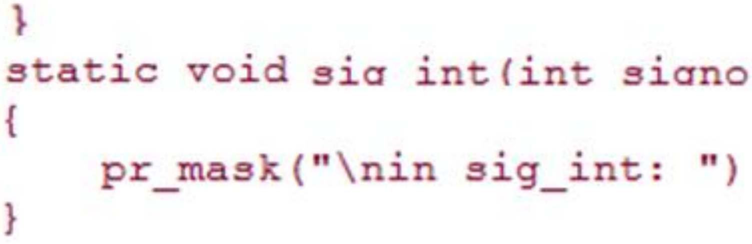


```
static void sig_int(int);
int main(void)
{ sigset_t    newmask, oldmask, waitmask;
  pr_mask("program start: ");
  if (signal(SIGINT, sig_int) == SIG_ERR)
      err_sys("signal(SIGINT) error");
  sigemptyset(&waitmask);
  sigaddset(&waitmask, SIGUSR1);
  sigemptyset(&newmask);
  sigaddset(&newmask, SIGINT);
  // Block SIGINT and save current signal mask.
  if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
      err_sys("SIG_BLOCK error");
```

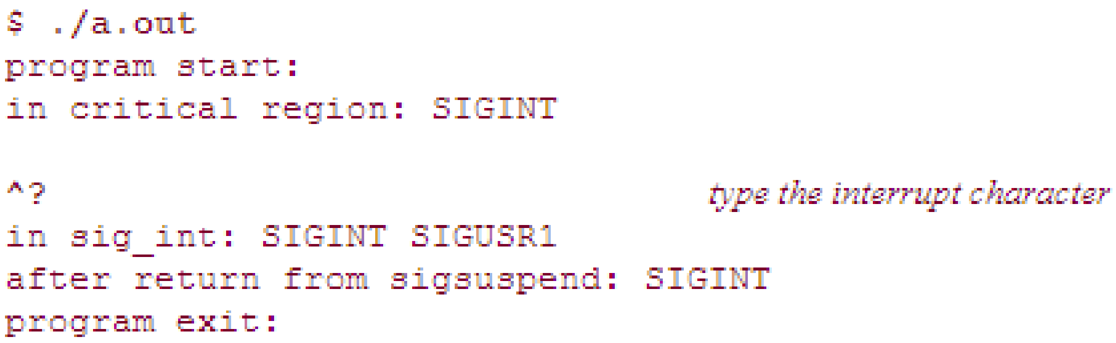
Protecting a Critical Region from a signal



```
 pr_mask("in critical region: ");
 // Pause, allowing all signals except SIGUSR1
 if (sigsuspend(&waitmask) != -1)
     err_sys("sigsuspend error");
pr_mask("after return from sigsuspend: ");
// Reset signal mask which unblocks SIGINT
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
     err_sys("SIG_SETMASK error");
pr_mask("program exit: ");
exit(0);
```

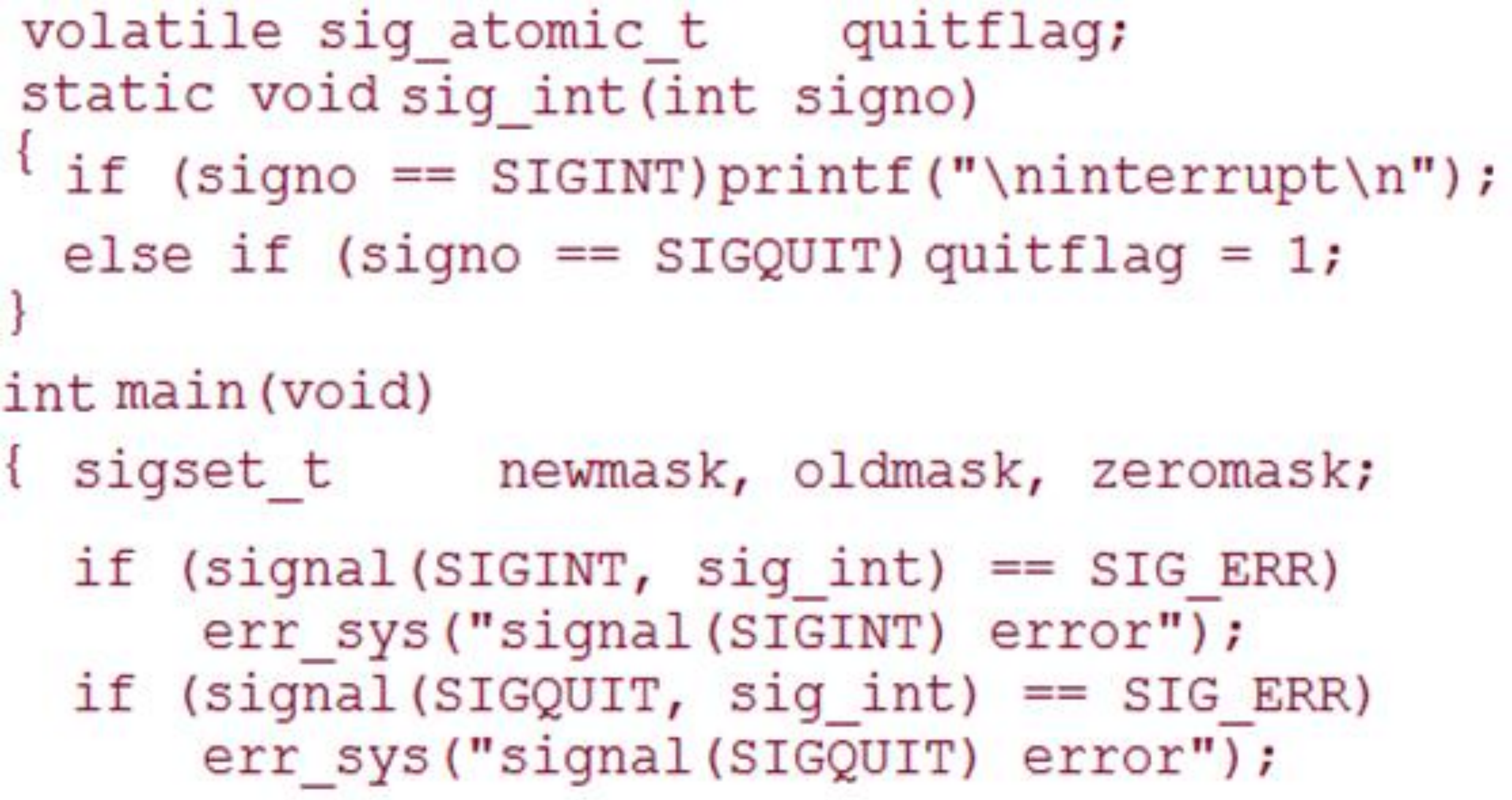


```
}
static void sig_int(int signo)
{
    pr_mask("\nin sig_int: ")
}
```



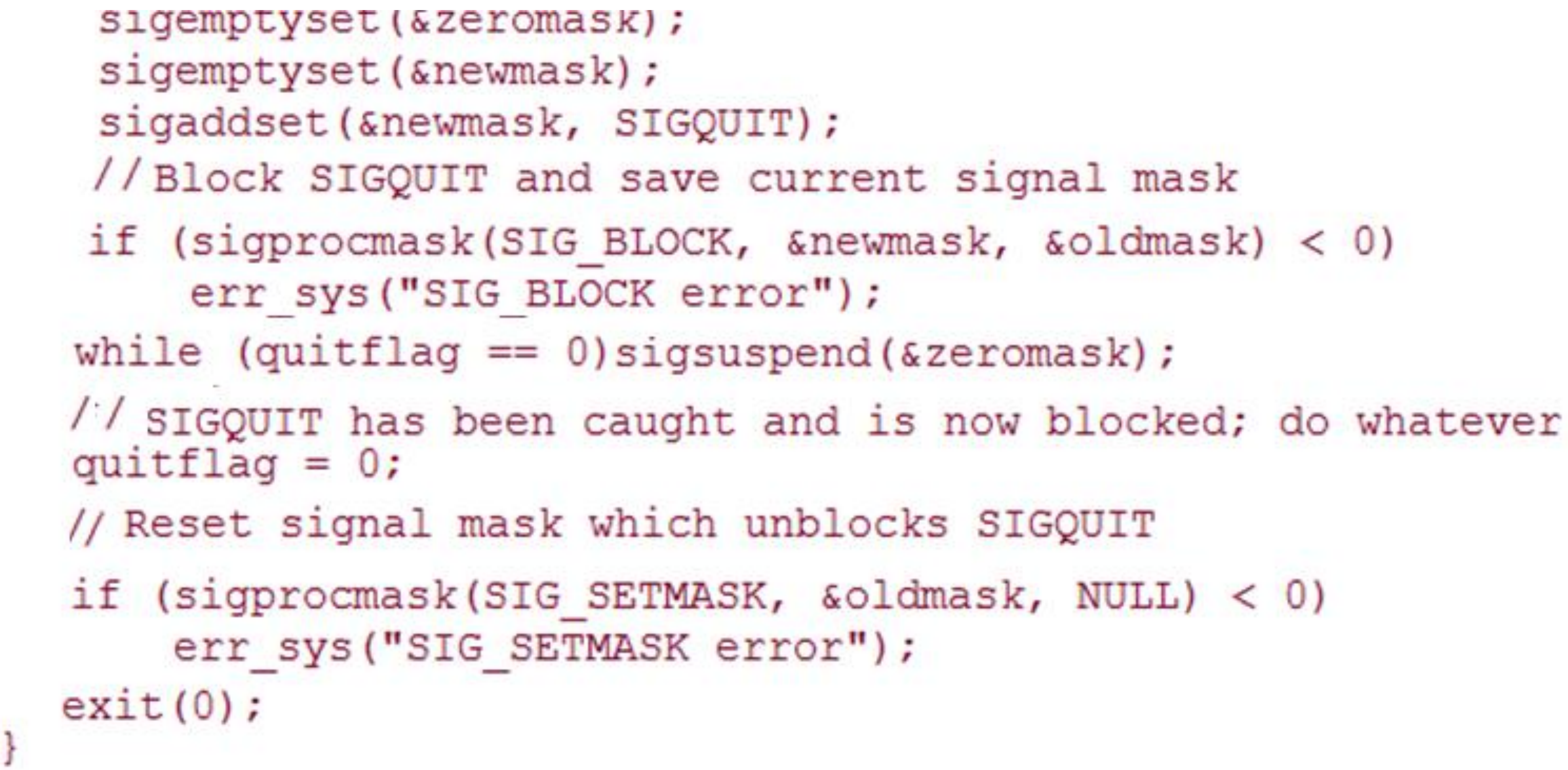
```
$ ./a.out
program start:
in critical region: SIGINT

^?                                              type the interrupt character
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

Using sigsuspend to wait for a global variable to be set



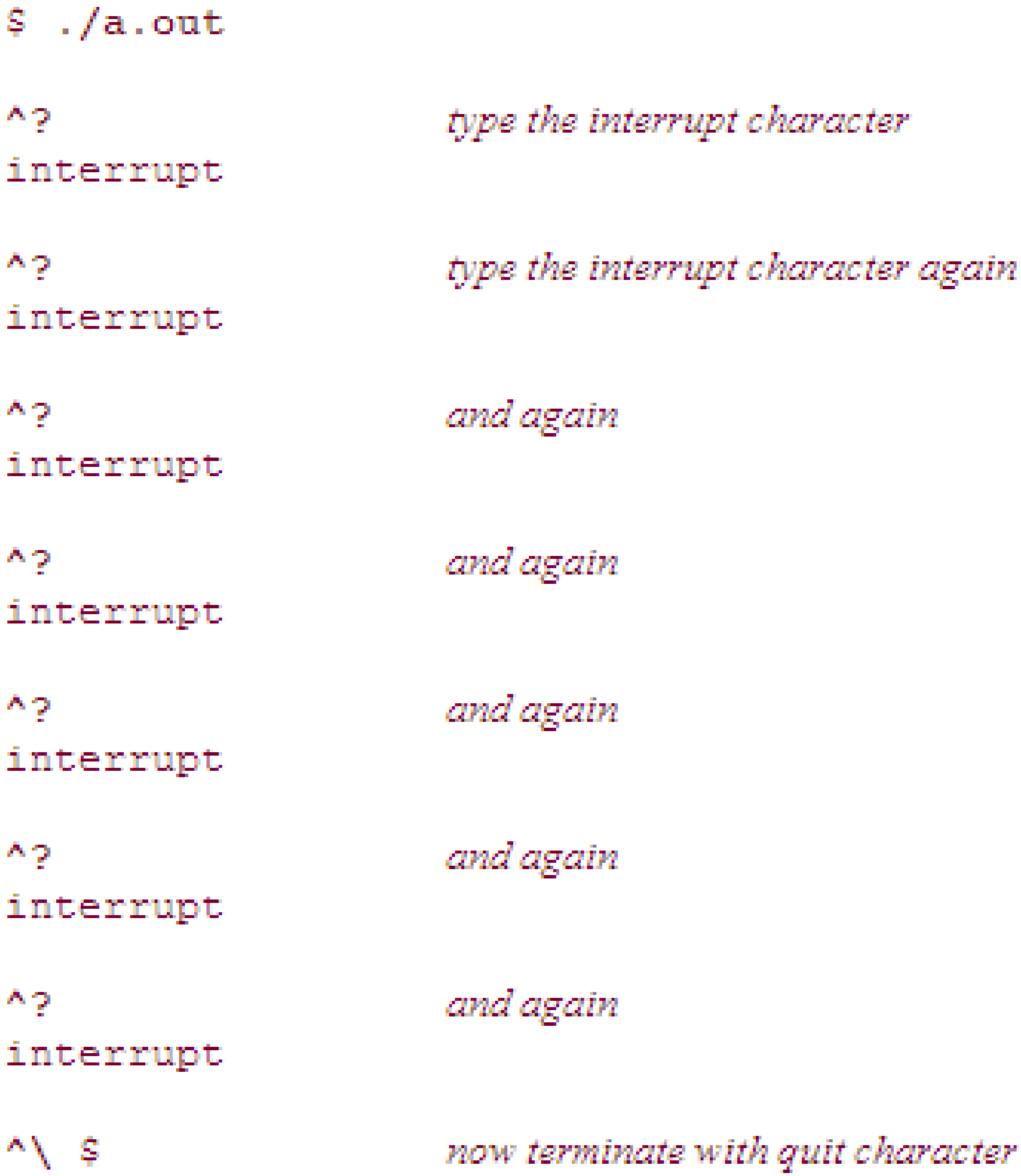
```
volatile sig_atomic_t    quitflag;
static void sig_int(int signo)
{
  if (signo == SIGINT)printf("\ninterrupt\n");
  else if (signo == SIGQUIT) quitflag = 1;
}

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

  if (signal(SIGINT, sig_int) == SIG_ERR)
      err_sys("signal(SIGINT) error");
  if (signal(SIGQUIT, sig_int) == SIG_ERR)
      err_sys("signal(SIGQUIT) error");
```

To set global variable



```
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
   //Block SIGQUIT and save current signal mask
   if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
       err_sys("SIG_BLOCK error");
  while (quitflag == 0)sigsuspend(&zeromask);
  // SIGQUIT has been caught and is now blocked; do whatever
  quitflag = 0;
  // Reset signal mask which unblocks SIGQUIT
  if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
      err_sys("SIG_SETMASK error");
  exit(0);
}
```

Execution



```
$ ./a.out

^?                      type the interrupt character
interrupt

^?                      type the interrupt character again
interrupt

^?                      and again
interrupt

^?                      and again
interrupt

^?                      and again
interrupt

^?                      and again
interrupt

^?                      and again
interrupt

^\ $                    now terminate with quit character
```

Upon fork()

- Child inherits from parent signal mask and dispositions
- Pending alarms are cleared for the child
- Set of pending signals for the child is set to empty set