UTSA The University of Texas at San Antonio™

EE3233 Systems Programming for Engrs

Reference: M. Kerrisk, The Linux Programming Interface

Lecture 9

Signals

Electrical & Computer Engineering

What we will cover in this lecture

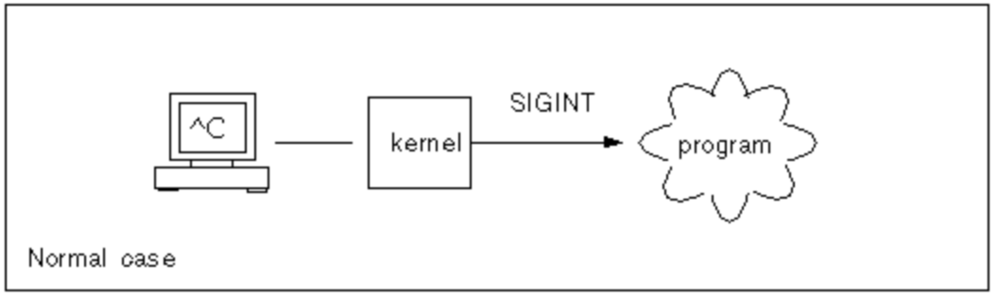
- Concept of Signal
- Types of Signal
- System Calls for Signal

Concepts

- Notification to a process that an event has occurred
    - Software interrupts
    - Interrupts the normal flow of execution of a program
    - One process can send a signal to another process: usually kernel sends it

Types of Events causing Kernel to Generate a Signal

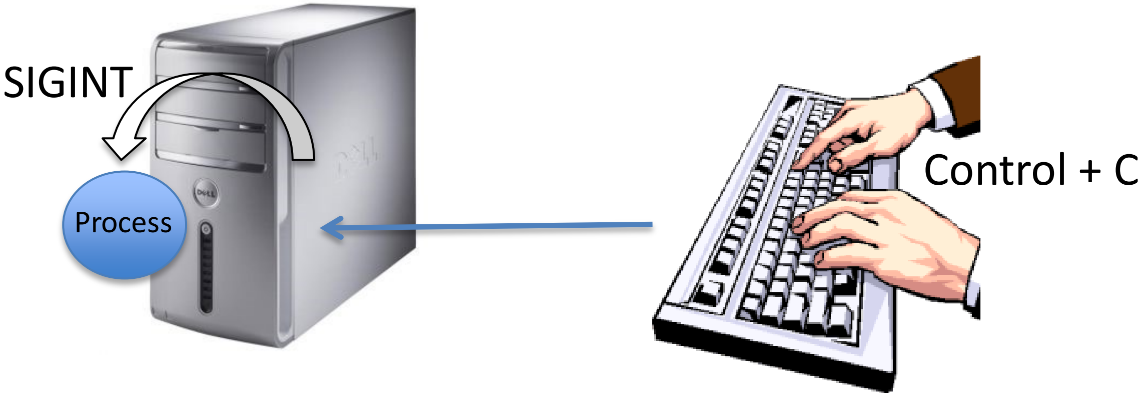
- H/W exception: H/W detected a fault condition and then notifies kernel
  - dividing by 0
  - referencing a part of memory that is inaccessible
- User typed one of the terminal special characters that generate signals
  - interrupt character (*Control + C*)
  - suspend character (*Control + Z*)



- S/W event
  - terminal window resizing
  - child of this process terminated

Concepts and Overview

- Each signal is defined as a unique integer
  - Starts from 1
  - Defined in <signal.h> with symbolic names of the form *SIGxxxx*
  - When the user types the interrupt character, *SIGINT* (signal number **2**) is delivered to a process



SIGINT

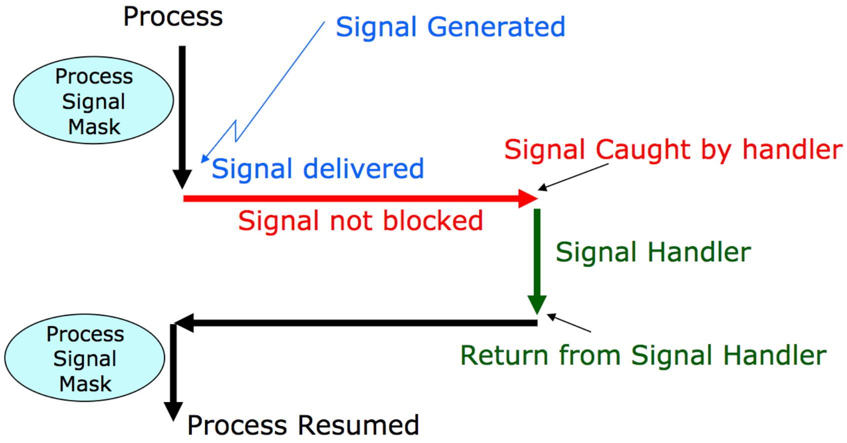
Process

Control + C

Concepts and Overview

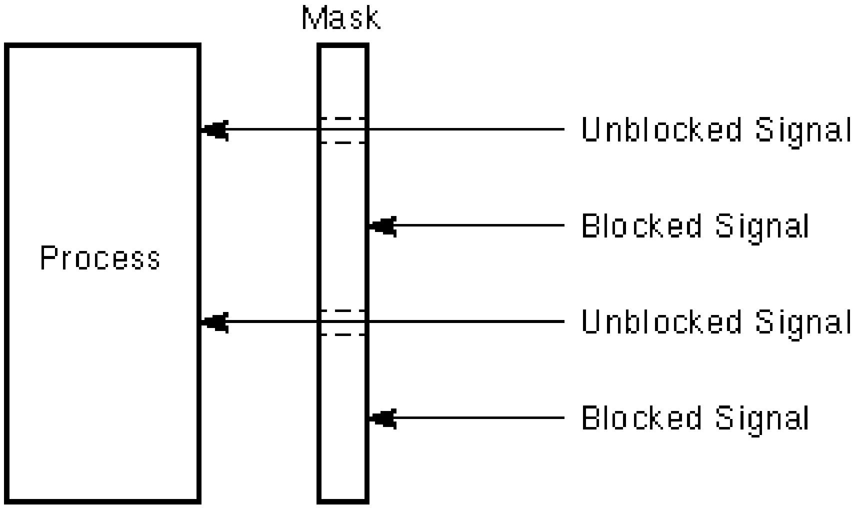
- Standard signal
  - Used by kernel to notify processes of event
  - On Linux, the standard signals are from 1 to 31
  - ***Generated*** by some event and later ***delivered*** to a process, which then takes some action in response to the signal

  → between '*generated*' and '*delivered*' times, a signal is said to be *pending*
- Normally, a pending signal is delivered to a process as soon as it is next scheduled to run, or immediately if the process is running



Concepts and Overview

- Sometimes, we need to ensure that a segment of code is not interrupted by the delivery of signal, then,
  - we can add a signal to the process's *signal mask*
- *Signal mask*: a set of signals whose delivery is currently blocked
  - If a signal is generated while it is blocked, it remains pending until it is later unblocked (removed from the signal mask)



Default Actions

- Signal is *ignored*
  - Discarded by the kernel and has no effect on the process
- The process is *terminated* (killed)
  - Abnormal process termination
  - Opposed to the normal process termination that occurs when a process terminates using *exit( )*
- *A core dump file* is generated, and the process is terminated
  - Core dump file contains an image of the virtual memory of the process
- The process is *stopped*
  - Execution of the process is suspended
  - Execution of the process is resumed after previously being stopped

Disposition

- A program can change the action when the signal is delivered
  - Default action should occur: undo an earlier change
  - The signal is ignored
  - A signal handler is executed

Signal Handler

- A function, written by the programmer, that performs appropriate tasks in response to the delivery of a signal
- Some terminologies:
  - *installing* or *establishing*: notifying the kernel that a handler function should be invoked
  - *handled* or *caught*: when a signal handler is invoked in response to the delivery of a signal

Signal Types and Default Actions

- SIGABRT
  - A process is sent this signal when it calls the *abort()* function
  - By default, this signal terminates the process with a core dump intentionally for debugging
- SIGCHLD
  - is sent (by kernel) to a parent process when one of its children terminates, or stops or resumes by a signal
- SIGCONT
  - When sent to a stopped process, this signal causes the process to resume

Signal Types and Default Actions

- SIGFPE (Floating Point Error)
  - is generated for certain types of arithmetic errors, such as divide-by-zero
- SIGINT
  - When the user type the terminal interrupt character, the terminal driver sends this signal to the foreground process group
  - The default action for this signal is to terminate the process
- SIGKILL
  - *sure kill* signal
  - Can't be blocked, ignored, or caught by a handler, and thus always terminate a process

Signal Types and Default Actions

- SIGQUIT
  - When the user types the quit character (**Control** + ****) on the keyboard
  - This signal is sent to the foreground process group
  - By default, this signal terminates a process and causes it to <u>produce a core dump</u> for debugging
- SIGTERM
  - Terminates a process
  - Sent by the kill commands (Users sometimes explicitly kill a process using kill -9)

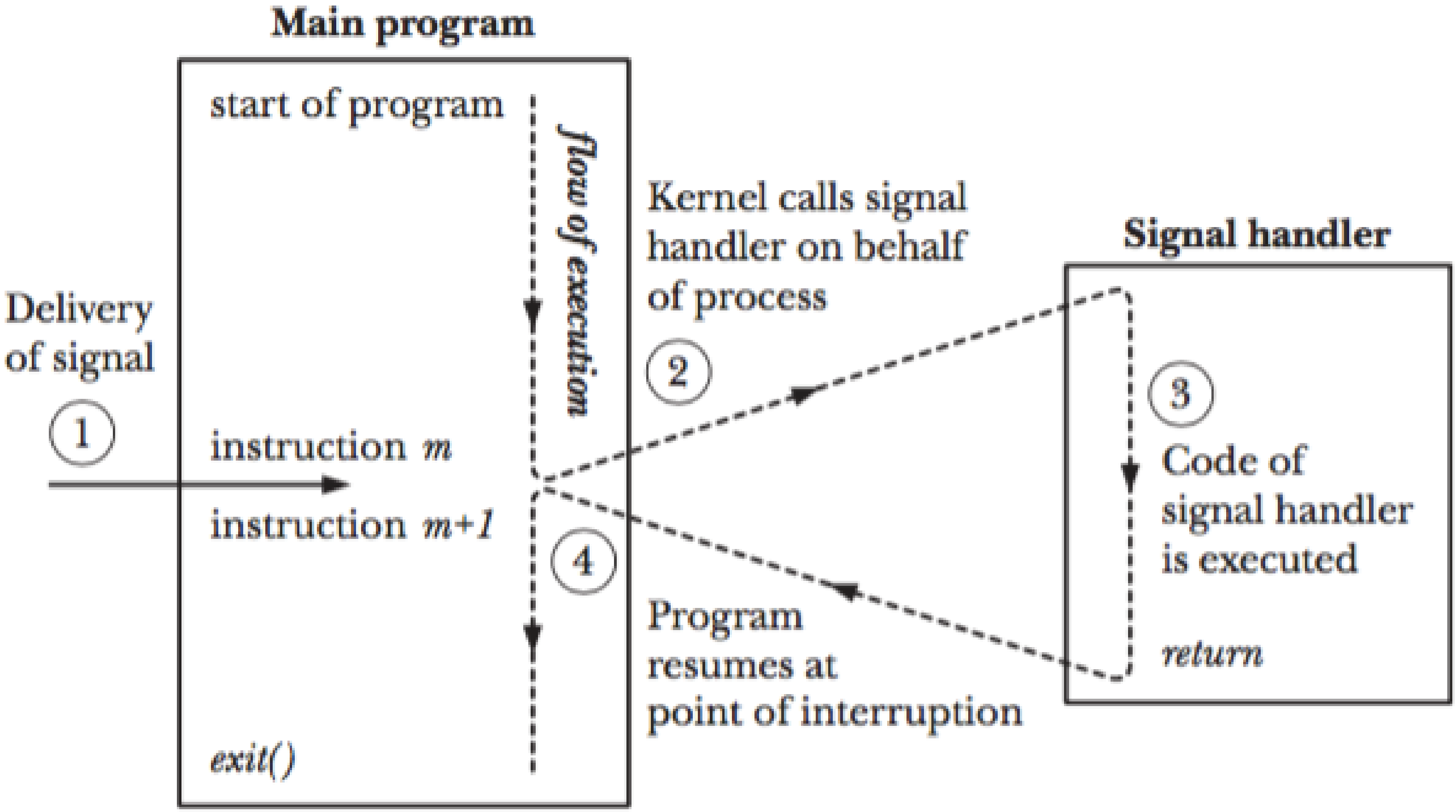
Changing Signal Disposition: *signal()*

```
#include <signal.h>


void ( *signal(int sig, void (*handler)(int)) ) (int);
```

- The first argument, *sig* identifies the signal whose disposition we wish to change
- The second argument, *handler* is the address of the function that should be called when this signal is delivered

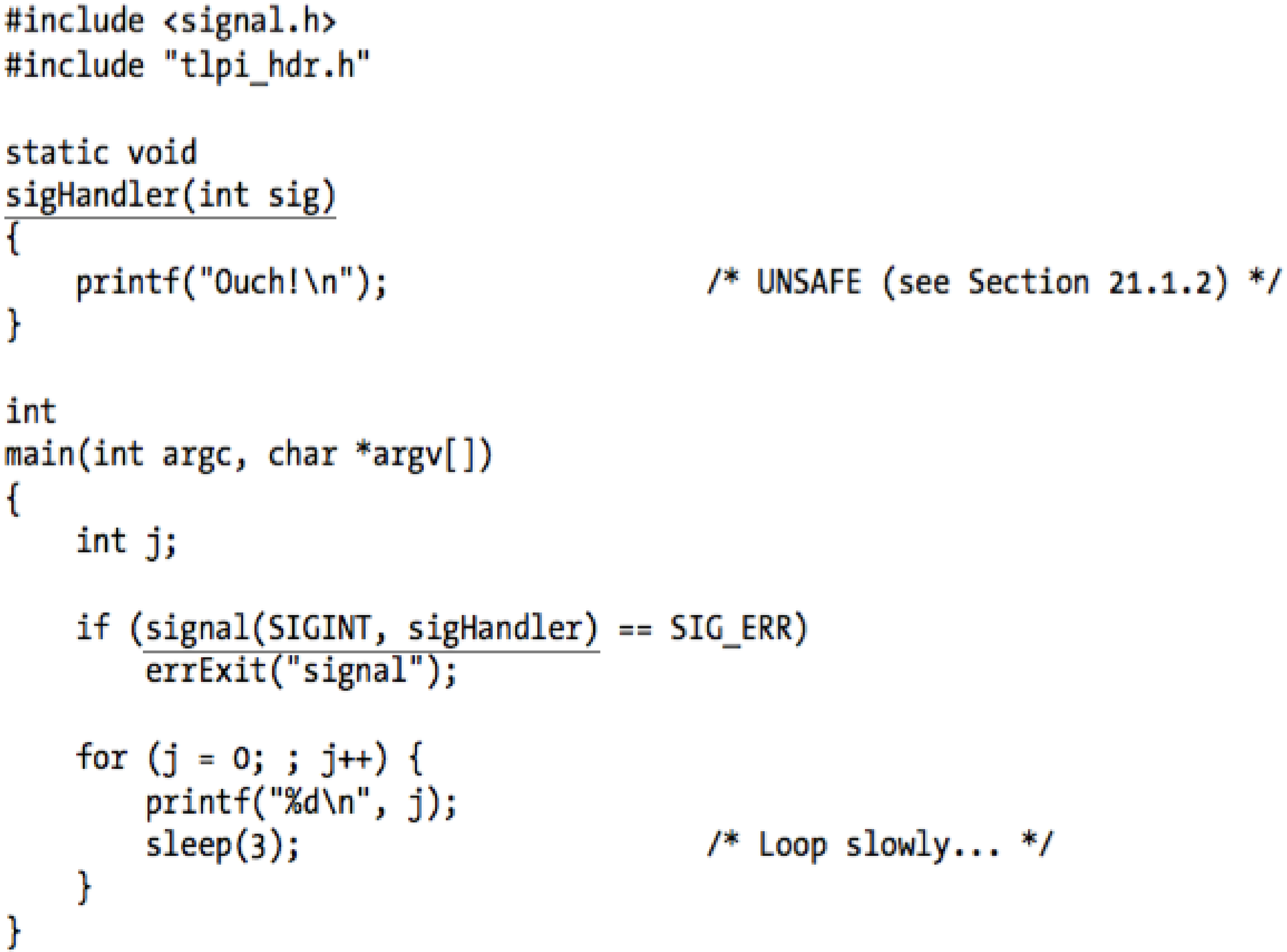
```
void handler (int sig) {
    /*  code for the handler */
}
```

Changing Signal Disposition: *signal()*



Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process's behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it

Changing Signal Disposition: *signal()*



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

static void
sigHandler(int sig)
{
    printf("Ouch!\n");                      /* UNSAFE (see Section 21.1.2) */
}

int
main(int argc, char *argv[])
{
    int j;

    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("signal");

    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3);                           /* Loop slowly... */
    }
}
```

Changing Signal Disposition: *signal()*

```
$ ./ouch
0
Type Control-C
Ouch!
1
2
Type Control-C
Ouch!
3
.
.
```

- The terminal driver generates *SIGINT* signal when we type terminal interrupt character (Control-C)
- The main program continuously loops
- When the kernel invokes a signal handler, it passes the number of the signal that caused the invocation as an integer to handler
- We can establish the same handler to catch different types of signals and use this argument to determine

Changing Signal Disposition: *signal()*

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

static void sigHandler(int sig) {
        static int count = 0;
        if (sig == SIGINT) {
                count++;
                printf("Caught SIGINT (%d)\n", count);
                return; /* Resume execution at point of interruption */ }

        /* Must be SIGQUIT - print a message and terminate the process */
        printf("Caught SIGQUIT - that's all folks!\n");
        exit(EXIT_SUCCESS); }

int main(int argc, char *argv[]) { /* Establish same handler for SIGINT and SIGQUIT */
        if (signal(SIGINT, sigHandler) == SIG_ERR) errExit("signal");
        if (signal(SIGQUIT, sigHandler) == SIG_ERR) errExit("signal");
        for (;;)     /* Loop forever, waiting for signals */
        pause();   /* Block until a signal is caught */
}
```

Changing Signal Disposition: *signal()*

```
$ ./intquit
^CCaught SIGINT(1)
^CCaught SIGINT(2)
^CCaught SIGINT(3)
^CCaught SIGINT(4)
^CCaught SIGINT(5)
^\Caught SIGQUIT – that's all folks!
```

- The code of the handler distinguishes two signals by examining the *sig* argument, and takes different actions
- *pause()* blocks the process until a signal is caught

Sending Signals: kill()

- One process can send a signal to another process using the **kill()** system call

```
#include <signal.h>

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

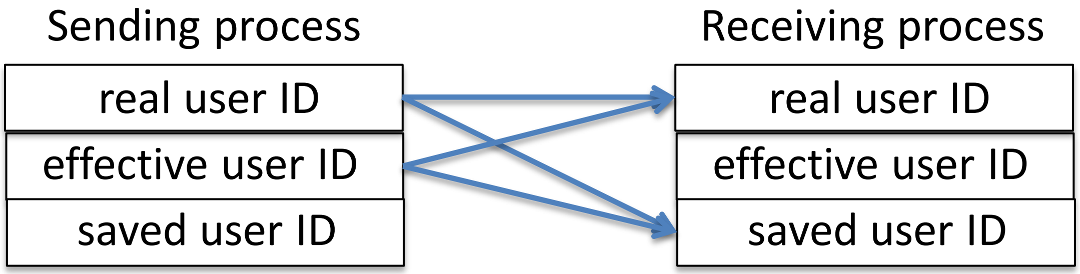
- *pid* identifies one or more processes to which the signal specified by *sig* is to be sent
- Returns 0 on success, or –1 on error

Sending Signals: *kill()*

- ***pid* > 0**
  - the signal is sent to the process with the process ID specified by *pid*
- ***pid* == 0**
  - the signal is sent to every process in the same process group as the calling process (including itself)
- ***pid* < -1**
  - the signal is sent to all of the processes in the process group whose ID equals the absolute value of *pid*
- ***pid* == -1**
  - the signal is sent to every process for which the calling process has permission to send a signal, except *init* and the calling process

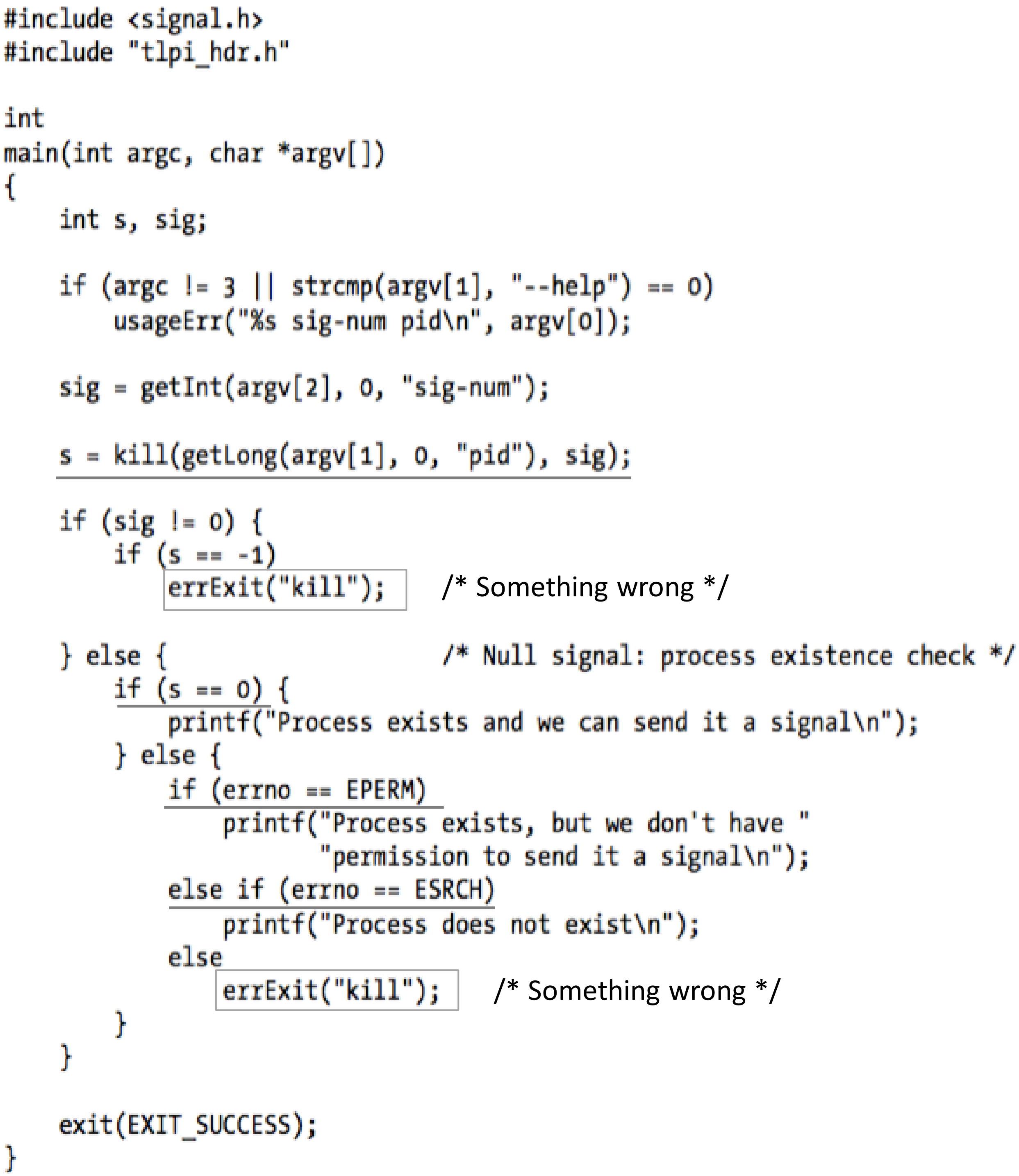
Sending Signals: *kill()*

- A privileged (CAP_KILL) process may send a signal to any process
- The *init* process (PID 1) can be sent only signals for which it has a handler installed
  - this prevents from accidently killing *init*
- An unprivileged process can send a signal to another process if
  - real or effective UID of sending process matches real or saved UID of receiving process



Checking existence of a process

- *kill()* can serve to check existence of a process
- If the *sig* argument is specified as 0, then no signal is sent
  - Instead, merely performs error checking to see if the process can be signaled
  - If sending a null signal fails with the error *ESRCH*, then we know the process doesn't exist
  - If fails with the error *EPERM* (meaning we do not have permission to send a signal to the process) or succeeds, the process exists



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

int
main(int argc, char *argv[])
{
    int s, sig;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sig-num pid\n", argv[0]);

    sig = getInt(argv[2], 0, "sig-num");

    s = kill(getLong(argv[1], 0, "pid"), sig);

    if (sig != 0) {
        if (s == -1)
            errExit("kill");        /* Something wrong */

    } else {                        /* Null signal: process existence check */
        if (s == 0) {
            printf("Process exists and we can send it a signal\n");
        } else {
            if (errno == EPERM)
                printf("Process exists, but we don't have "
                       "permission to send it a signal\n");
            else if (errno == ESRCH)
                printf("Process does not exist\n");
            else
                errExit("kill");    /* Something wrong */
        }
    }

    exit(EXIT_SUCCESS);
}
```

Checking existence of a process

```
$ ./ouch
 1
 2
 3
 4
 5
 *
 *
 *
```

```
$ ps aux | grep ouch
user1 3429 …  ./ouch


$ ./t_kill 3429 0
Process exists and we
can send it a signal


$ ./t_kill 3333 0
Process does not exist
```

Other Ways of Sending Signals: *raise()*

- Sometimes useful to send a signal to itself

```
#include <signal.h>

int raise(int sig);
```

- In a single-threaded program, a call to *raise()* is equivalent to the following call to *kill()*

kill(getpid(), sig);

- When a process sends itself a signal using *raise()*, the signal is delivered immediately

Other Ways of Sending Signals: *killpg()*

- Sometimes useful to send a signal to itself

```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

- equivalent to the following call to kill():

kill(-*pgrp*, *sig*);

If *pgrp* is specified as 0, then the signal is sent to all processes in the same process group as the caller

Displaying Signal Descriptions

- char ***strsignal** (int sig);
  - Each signal has an associated printable description: These descriptions are listed in the array *sys_siglist* – For example, we can refer to *sys_siglist[SIGPIPE]* to get the description for *SIGPIPE* (broken pipe)
  - performs bounds checking on the sig argument, and then returns a pointer to a <u>printable description</u> of the signal

Signal Sets

- **int sigemptyset (sigset_t *set);**
  - initializes a signal set to contain no members
- **int sigfillset (sigset_t *set);**
  - initializes a signal set to contain all signals
- **int sigaddset (const sigset_t *set, int sig);**
  - individual signals can be added to a *set*
- **int sigdelset (const sigset_t *set, int sig);**
  - individual signals can be removed
  - returns true if *set* contains no signals

Signal Sets

- **int sigisemptyset (const sigset_t *set);**
  - returns true if *set* contains no signals
- **int sigismember (const sigset_t *set, int sig);**
  - returns 1 if *sig* is a member of *set*, and 0 otherwise
- **int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);**
  - places the <u>intersection</u> of the sets *left* and *right* in the set *dest*
- **int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);**
  - places the <u>union</u> of the sets *left* and *right* in the set *dest*

Signal Mask (Blocking Signal Delivery)

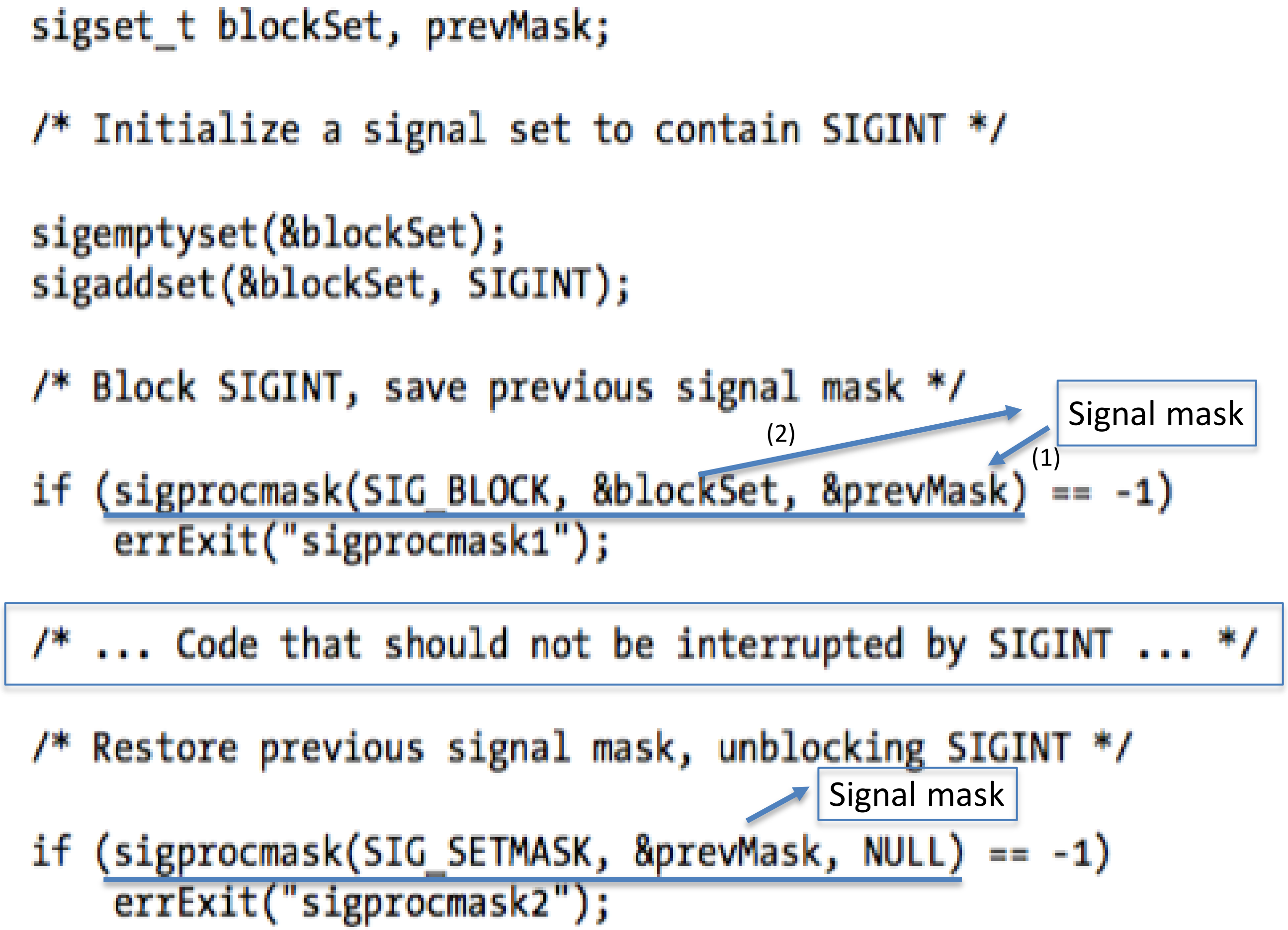
- **int sigprocmask (int *how*, const sigset_t **set*, sigset_t **oldset*);**
  - used to explicitly add signals to and remove signals from the signal mask
  - *how*:

    SIG_BLOCK: *set* is added to the signal mask (union)

    SIG_UNBLOCK: *set* is removed from the signal mask

    SIG_SETMASK: *set* is assigned to the signal mask
  - *oldset*: if the *oldset* is not NULL, it points to a *sigset_t* buffer that is used to return the previous signal mask

Signal Mask (Blocking Signal Delivery)



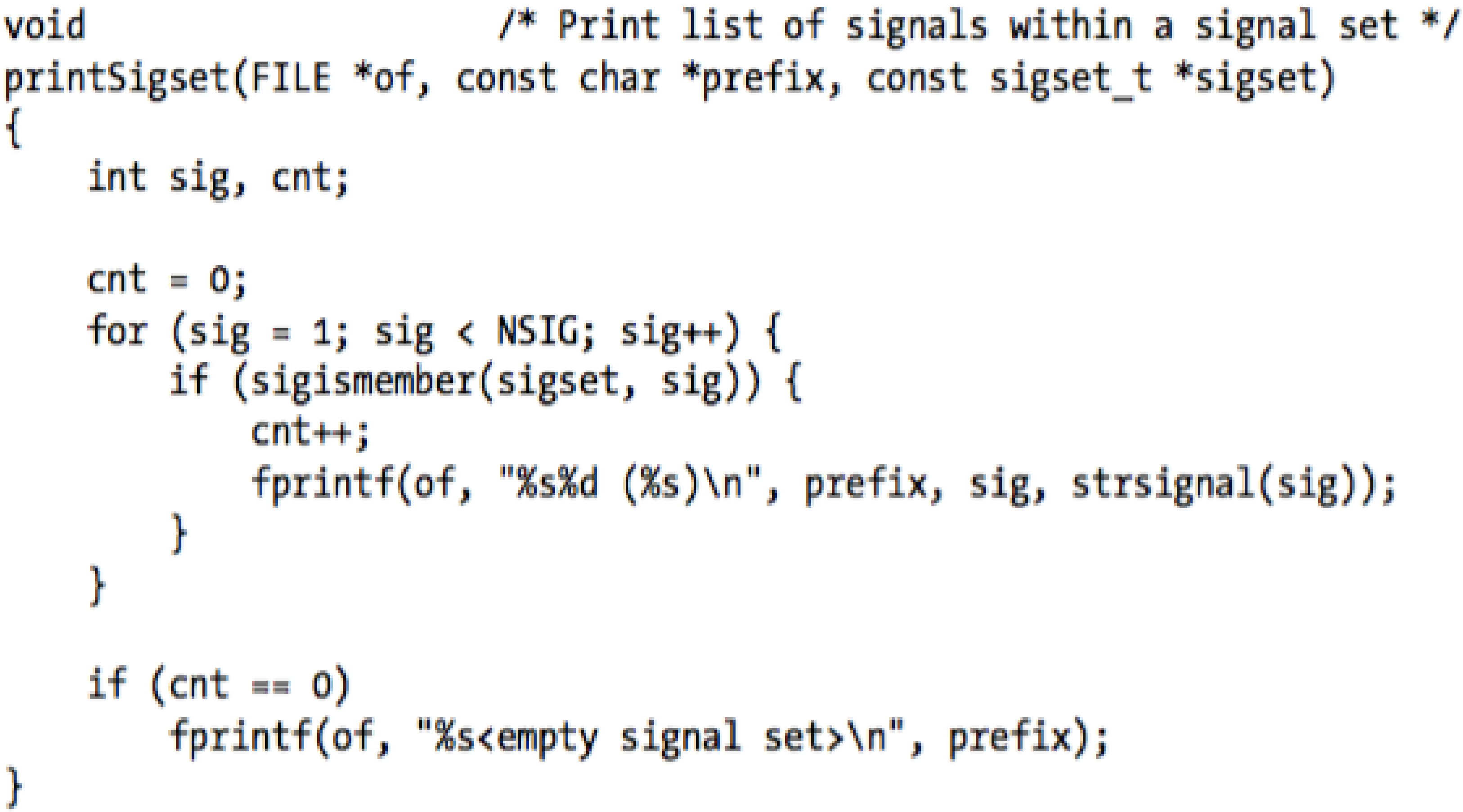
```
sigset_t blockSet, prevMask;

/* Initialize a signal set to contain SIGINT */

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

/* Block SIGINT, save previous signal mask */

if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1)
    errExit("sigprocmask1");

/* ... Code that should not be interrupted by SIGINT ... */

/* Restore previous signal mask, unblocking SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask2");
```

- Temporarily prevents delivery of a signal(SIGINT) and then unblock it by resetting the signal mask to its previous state

Pending Signals

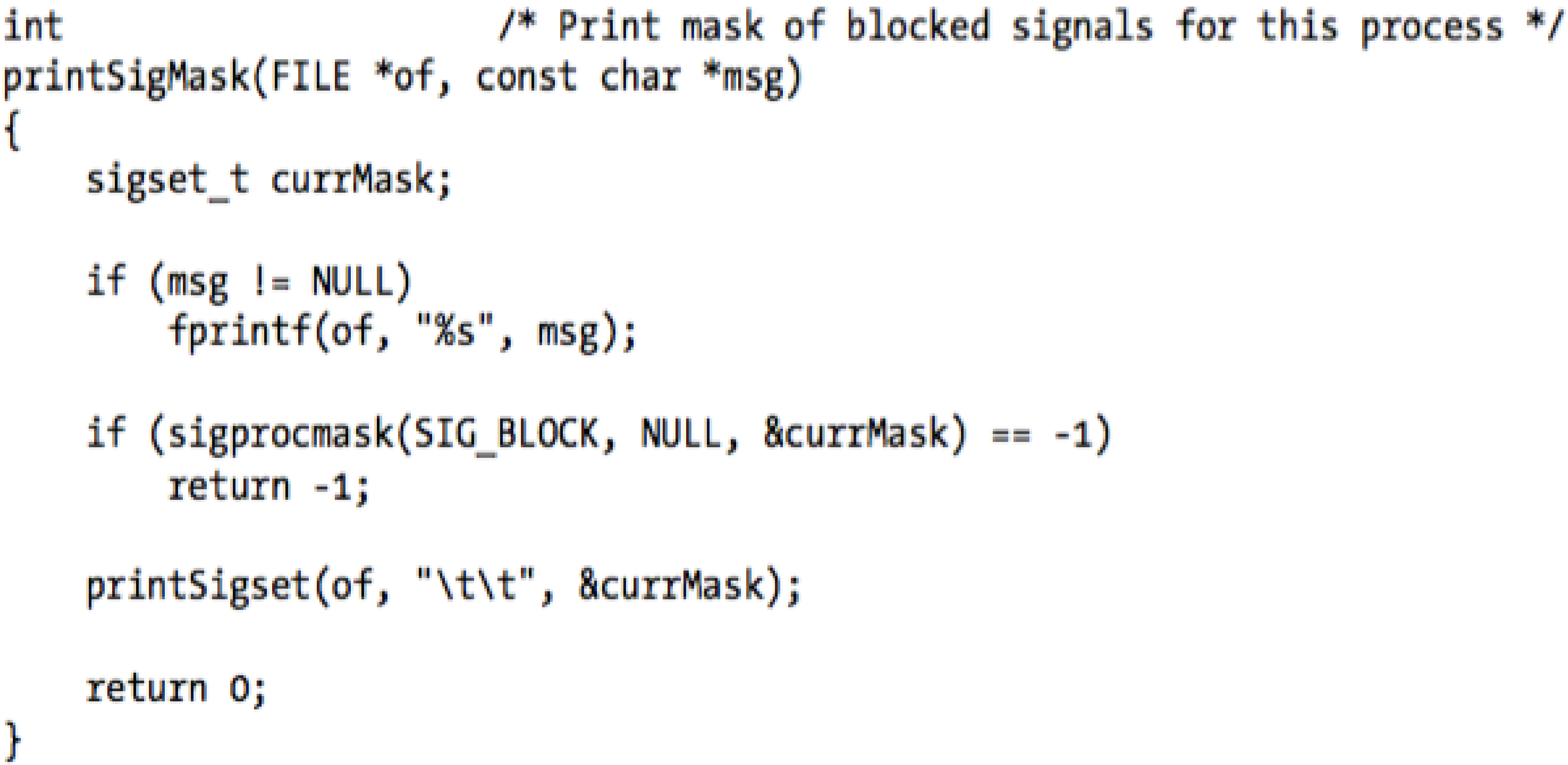
- **int sigpending (sigset_t ****set***);**
  - If a process receives a signal that it is currently blocking, that signal is added to the process's set of pending signals
  - *sigpending()* system call returns the set of signals that are pending for the calling process in the *sigset_t* structure pointed to by *set*

Example Program Displaying Signal Sets



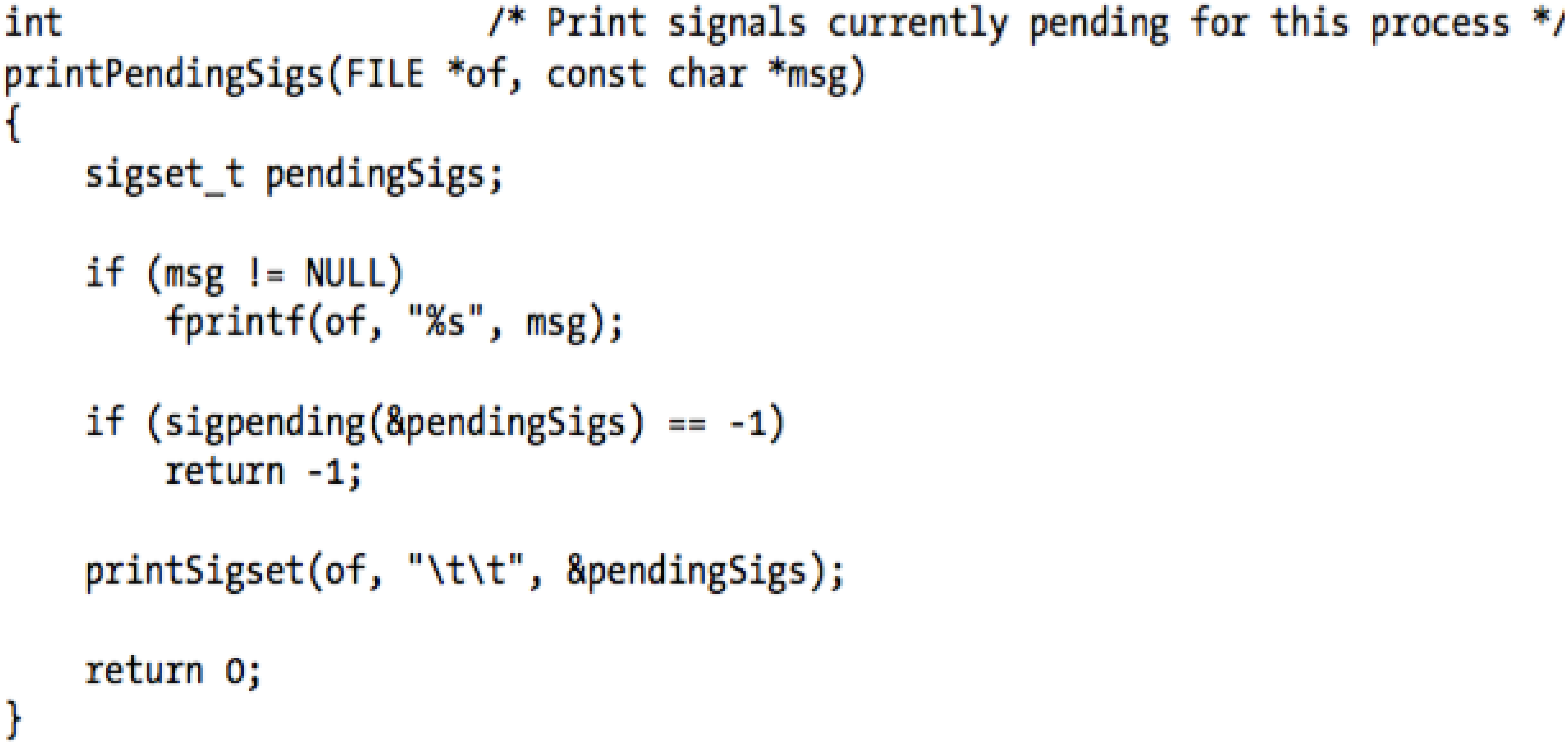
```
void                    /* Print list of signals within a signal set */
printSigset(FILE *of, const char *prefix, const sigset_t *sigset)
{
    int sig, cnt;

    cnt = 0;
    for (sig = 1; sig < NSIG; sig++) {
        if (sigismember(sigset, sig)) {
            cnt++;
            fprintf(of, "%s%d (%s)\n", prefix, sig, strsignal(sig));
        }
    }

    if (cnt == 0)
        fprintf(of, "%s<empty signal set>\n", prefix);
}
```

Example Program Displaying Signal Sets



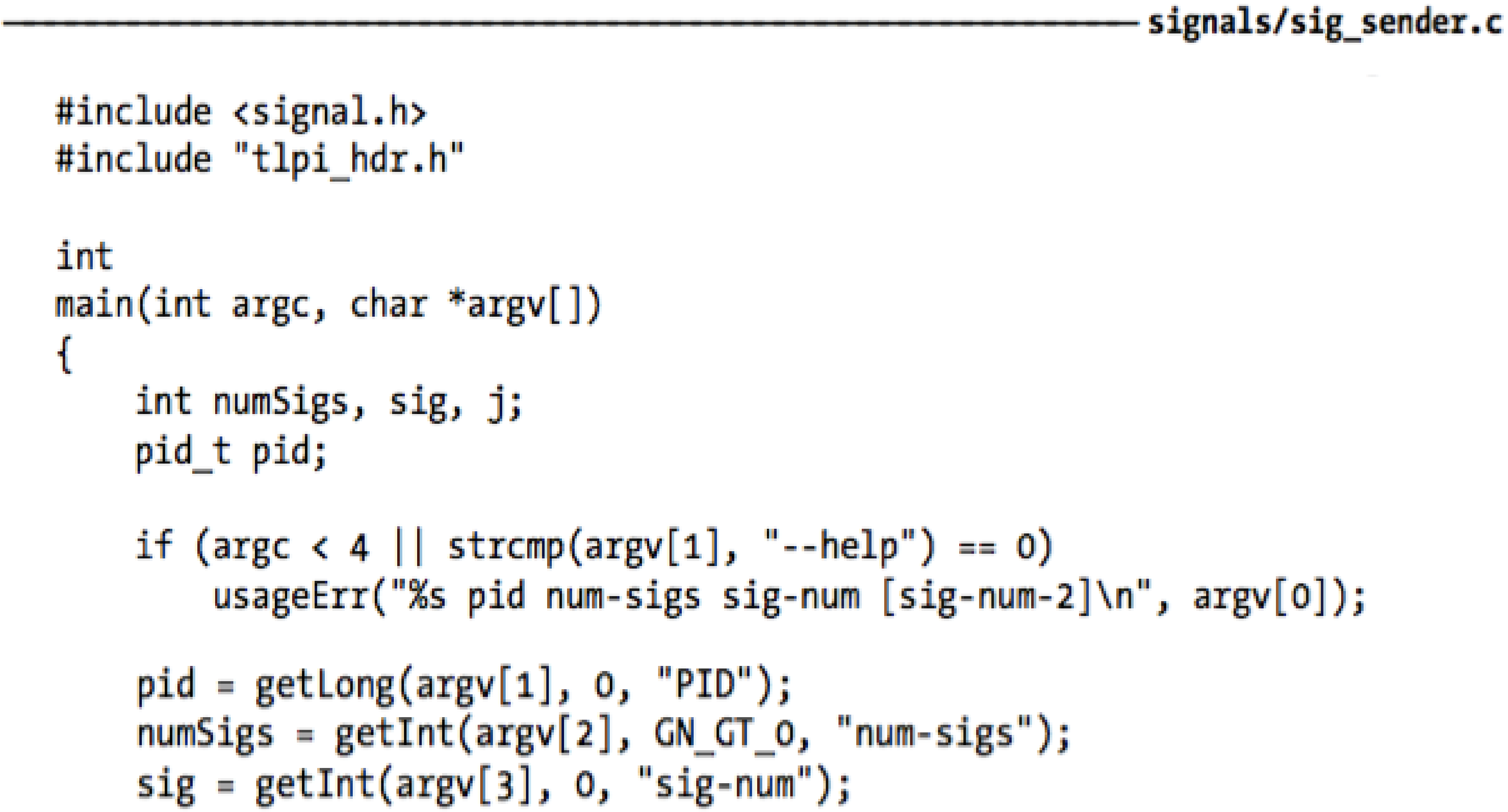
```
int                     /* Print mask of blocked signals for this process */
printSigMask(FILE *of, const char *msg)
{
    sigset_t currMask;

    if (msg != NULL)
        fprintf(of, "%s", msg);

    if (sigprocmask(SIG_BLOCK, NULL, &currMask) == -1)
        return -1;

    printSigset(of, "\t\t", &currMask);

    return 0;
}
```

Example Program Displaying Signal Sets



```
int                     /* Print signals currently pending for this process */
printPendingSigs(FILE *of, const char *msg)
{
    sigset_t pendingSigs;

    if (msg != NULL)
        fprintf(of, "%s", msg);

    if (sigpending(&pendingSigs) == -1)
        return -1;

    printSigset(of, "\t\t", &pendingSigs);

    return 0;
}
```

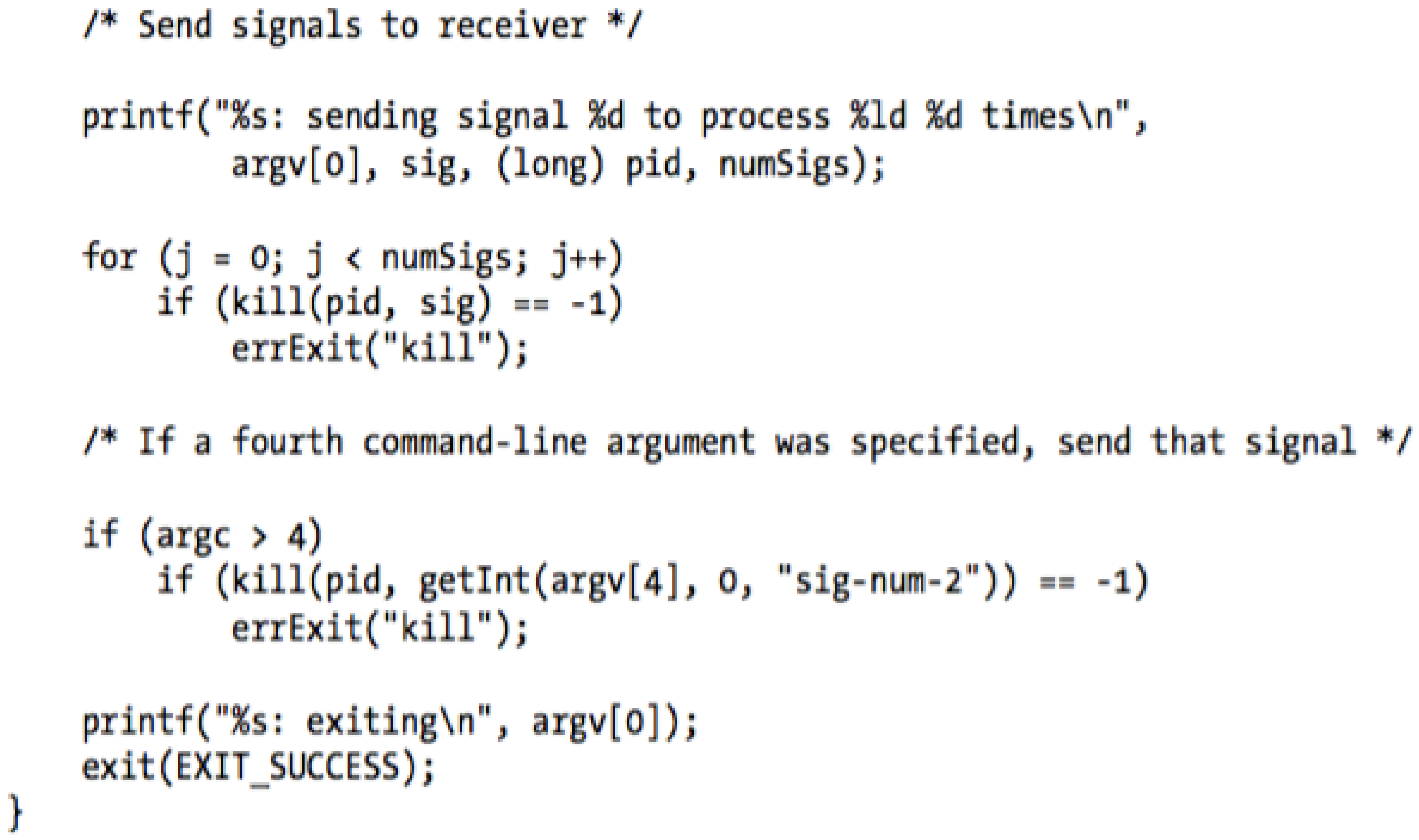
Signals are not queued



signals/sig_sender.c

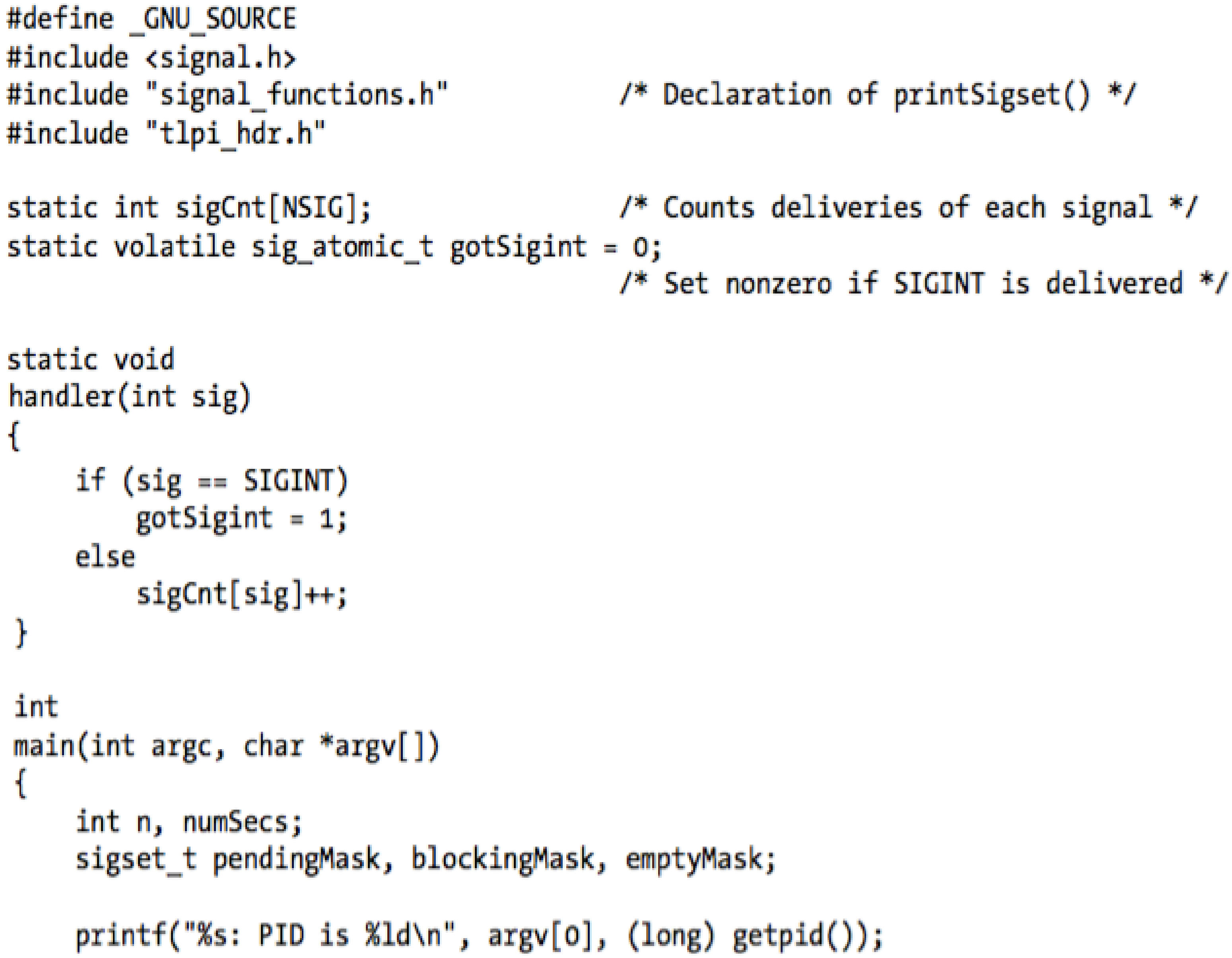
```
#include <signal.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numSigs, sig, j;
    pid_t pid;

    if (argc < 4 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pid num-sigs sig-num [sig-num-2]\n", argv[0]);

    pid = getLong(argv[1], 0, "PID");
    numSigs = getInt(argv[2], GN_GT_0, "num-sigs");
    sig = getInt(argv[3], 0, "sig-num");
```

Signals are not queued

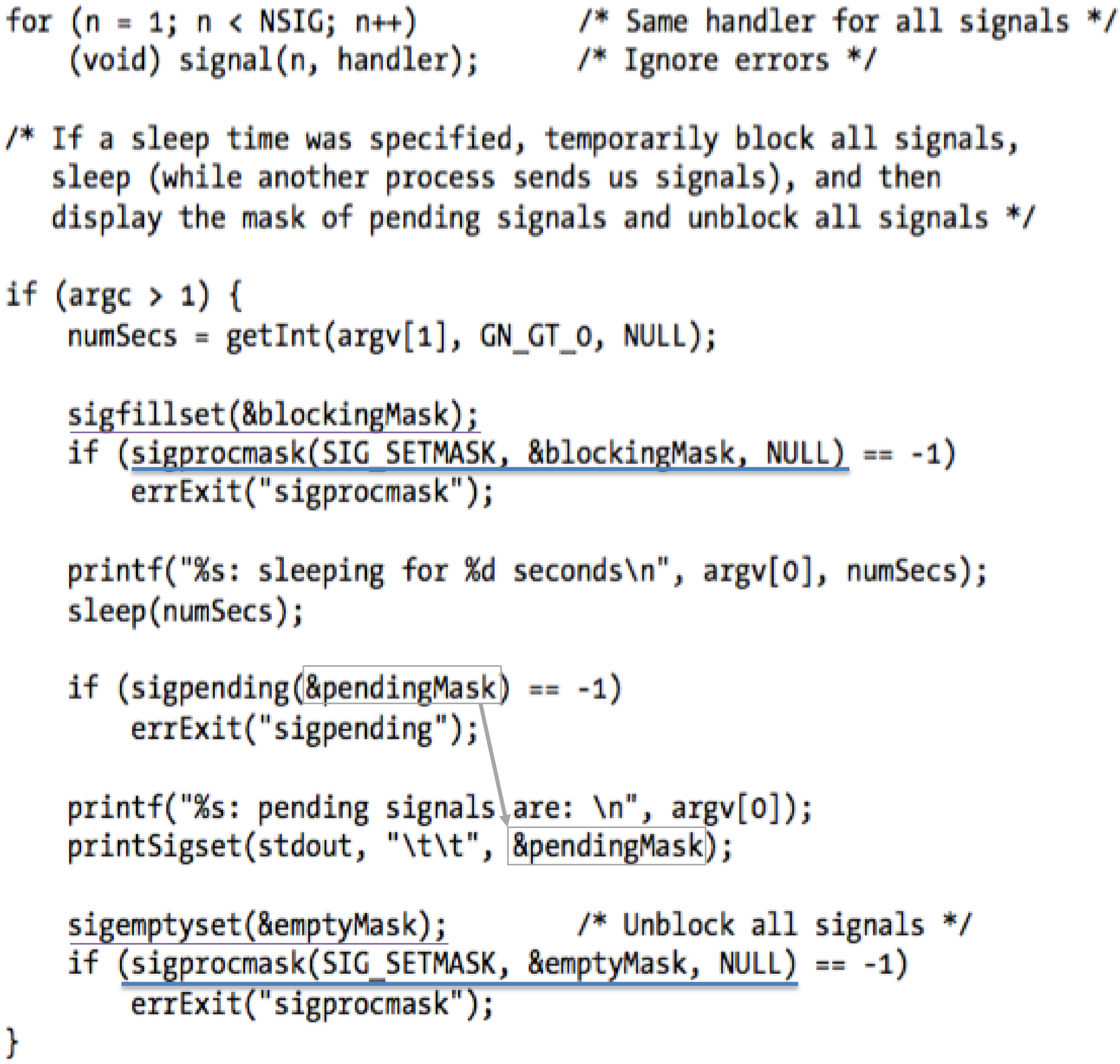


```
    /* Send signals to receiver */

    printf("%s: sending signal %d to process %ld %d times\n",
            argv[0], sig, (long) pid, numSigs);

    for (j = 0; j < numSigs; j++)
        if (kill(pid, sig) == -1)
            errExit("kill");

    /* If a fourth command-line argument was specified, send that signal */

    if (argc > 4)
        if (kill(pid, getInt(argv[4], 0, "sig-num-2")) == -1)
            errExit("kill");

    printf("%s: exiting\n", argv[0]);
    exit(EXIT_SUCCESS);
}
```

Signal receiver

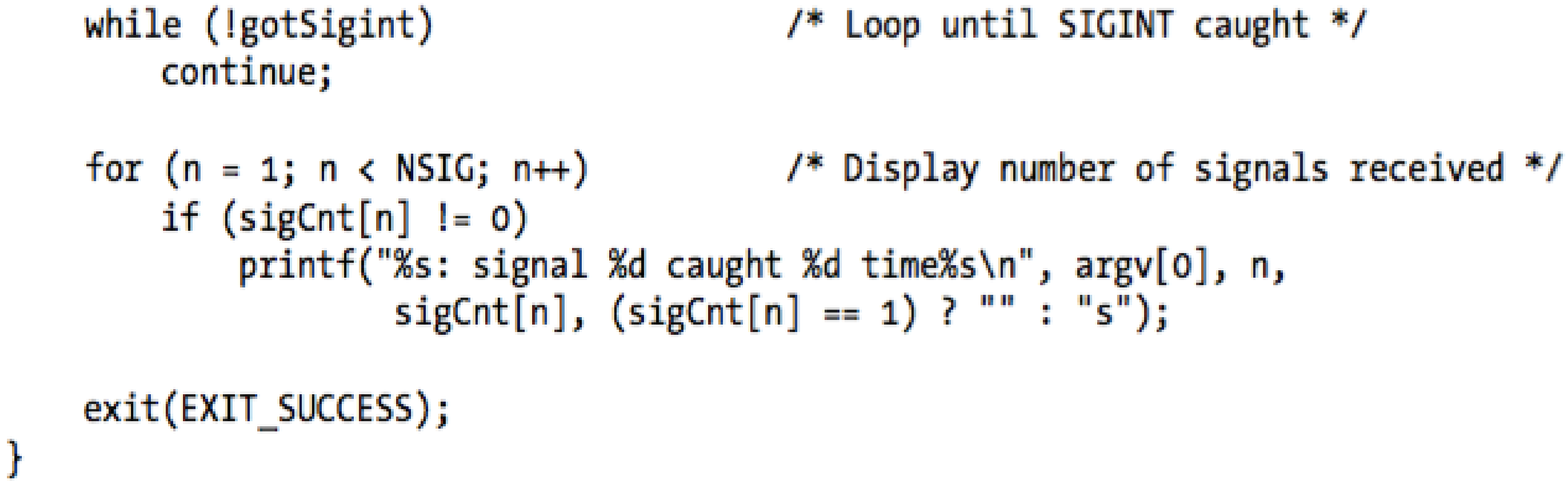


```
#define _GNU_SOURCE
#include <signal.h>
#include "signal_functions.h"          /* Declaration of printSigset() */
#include "tlpi_hdr.h"

static int sigCnt[NSIG];               /* Counts deliveries of each signal */
static volatile sig_atomic_t gotSigint = 0;
                                       /* Set nonzero if SIGINT is delivered */

static void
handler(int sig)
{
    if (sig == SIGINT)
        gotSigint = 1;
    else
        sigCnt[sig]++;
}

int
main(int argc, char *argv[])
{
    int n, numSecs;
    sigset_t pendingMask, blockingMask, emptyMask;

    printf("%s: PID is %ld\n", argv[0], (long) getpid());
```



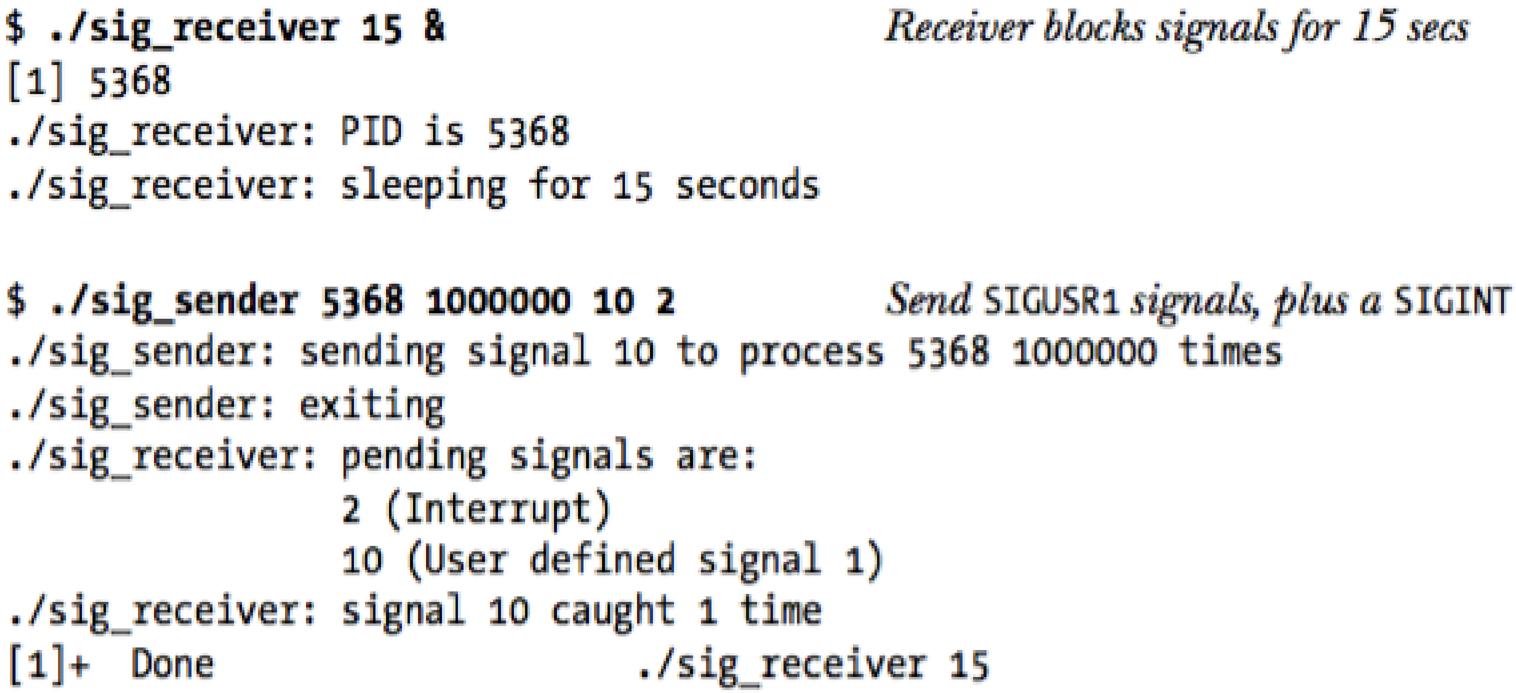
```
for (n = 1; n < NSIG; n++)          /* Same handler for all signals */
    (void) signal(n, handler);      /* Ignore errors */

/* If a sleep time was specified, temporarily block all signals,
   sleep (while another process sends us signals), and then
   display the mask of pending signals and unblock all signals */

if (argc > 1) {
    numSecs = getInt(argv[1], GN_GT_0, NULL);

    sigfillset(&blockingMask);
    if (sigprocmask(SIG_SETMASK, &blockingMask, NULL) == -1)
        errExit("sigprocmask");

    printf("%s: sleeping for %d seconds\n", argv[0], numSecs);
    sleep(numSecs);

    if (sigpending(&pendingMask) == -1)
        errExit("sigpending");

    printf("%s: pending signals are: \n", argv[0]);
    printSigset(stdout, "\t\t", &pendingMask);

    sigemptyset(&emptyMask);        /* Unblock all signals */
    if (sigprocmask(SIG_SETMASK, &emptyMask, NULL) == -1)
        errExit("sigprocmask");
}
```

Signal receiver



```
    while (!gotSigint)                  /* Loop until SIGINT caught */
        continue;

    for (n = 1; n < NSIG; n++)          /* Display number of signals received */
        if (sigCnt[n] != 0)
            printf("%s: signal %d caught %d time%s\n", argv[0], n,
                    sigCnt[n], (sigCnt[n] == 1) ? "" : "s");

    exit(EXIT_SUCCESS);
}
```

Signal receiver



```
$ ./sig_receiver 15 &                         Receiver blocks signals for 15 secs
[1] 5368
./sig_receiver: PID is 5368
./sig_receiver: sleeping for 15 seconds

$ ./sig_sender 5368 1000000 10 2              Send SIGUSR1 signals, plus a SIGINT
./sig_sender: sending signal 10 to process 5368 1000000 times
./sig_sender: exiting
./sig_receiver: pending signals are:
                2 (Interrupt)
                10 (User defined signal 1)
./sig_receiver: signal 10 caught 1 time
[1]+  Done                    ./sig_receiver 15
```

A blocked signal is delivered only once, no matter how many times it is generated