**FREE eBook**

# LEARNING POSIX

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#posix

# Table of Contents

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: posix

It is an unofficial and free POSIX ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official POSIX.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Chapter 1: Getting started with POSIX

## Versions

| Version | Standard | Release Year |
|---|---|---|
| POSIX.1 | IEEE Std 1003.1-1988 | 1988-01-01 |
| POSIX.1b | IEEE Std 1003.1b-1993 | 1993-01-01 |
| POSIX.1c | IEEE Std 1003.1c-1995 | 1995-01-01 |
| POSIX.2 | IEEE Std 1003.2-1992 | 1992-01-01 |
| POSIX.1-2001 | IEEE Std 1003.1-2001 | 2001-12-06 |
| POSIX.1-2004 | IEEE Std 1003.1-2004 | 2004-01-01 |
| POSIX.1-2008 | IEEE Std 1003.1-2008 (aka "*Base Specifications, Issue 7*") | 2008-12-01 |
| POSIX.1-2013 | IEEE Std 1003.1-2013 | 2013-04-19 |
| POSIX.1-2016 | IEEE Std 1003.1-2016 | 2016-09-30 |

## Examples

### What is POSIX?

POSIX stands for "*Portable Operating System Interface*" and defines a set of standards to provide compatibility between different computing platforms. The current version of the standard is IEEE 1003.1 2016 and can be accessed from the OpenGroup POSIX specification. Previous versions include POSIX 2004 and POSIX 1997. The POSIX 2016 edition is essentially POSIX 2008 plus errata (there was a POSIX 2013 release too).

POSIX defines various tools interfaces, commands and APIs for UNIX-like operating systems and others.

The following are considered to be within the scope of POSIX standardization:

- System interface (functions, macros and external variables)
- Command interpreter, or Shell (the *sh* utility)
- Utilities (such as *more*, *cat*, *ls*)

Outside of POSIX scope:

- DBMS Interfaces

- Graphical Interfaces
- Binary code portability

## Hello World

A simple `Hello, World` program without error checking:

```
#include <unistd.h> /* For write() and STDOUT_FILENO */
#include <stdlib.h> /* For EXIT_SUCCESS and EXIT_FAILURE */

int main(void) {
        char hello[] = "Hello, World\n";

        /* Attempt to write `hello` to standard output file */
        write(STDOUT_FILENO, hello, sizeof(hello) - 1);

        return EXIT_SUCCESS;
}
```

And with error checking:

```
#include <unistd.h> /* For write() and STDOUT_FILENO */
#include <stdlib.h> /* For EXIT_SUCCESS and EXIT_FAILURE */

int main(void) {
        char hello[] = "Hello, World\n";
        ssize_t ret = 0;

        /* Attempt to write `hello` to standard output file */
        ret = write(STDOUT_FILENO, hello, sizeof(hello) - 1);

        if (ret == -1) {
                /* write() failed. */
                return EXIT_FAILURE;
        } else if (ret != sizeof(hello) - 1) {
                /* Not all bytes of `hello` were written. */
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

## Compiling and running

If the code shown above (either version) is stored in file `hello.c`, then you can compile the code into a program `hello` using either `c99` or `make`. For example, in a strictly POSIX compliant mode, you might in theory compile and run the program using:

```
$ make hello
c99 -o hello hello.c
$ ./hello
Hello, World
$
```

Most actual `make` implementations will use a different C compiler (perhaps `cc`, perhaps `gcc`, `clang`, `xlc` or some other name), and many will use more options to the compiler. Clearly, you could simply type the command that `make` executes directly on the command line.

Read Getting started with POSIX online: https://riptutorial.com/posix/topic/4495/getting-started-with-posix

# Chapter 2: File locks

## Syntax

- int fcntl(int fd, int cmd, struct flock*);
- int lockf(int fd, int cmd, off_t len);

## Examples

### POSIX record locks (fcntl)

This example demonstrates usage of POSIX record locks (a.k.a. process-associated locks), provided by `fcntl` function (POSIX base standard).

Notes:

- Exclusive and shared locks are supported.
- Can be applied to a byte range, optionally automatically expanding when data is appended in future (controlled by `struct flock`).
- Locks are released on first close by the locking process of *any* file descriptor for the file, or when process terminates.

```
#include <stdlib.h>  /* for exit() */
#include <stdio.h>   /* for perror() */
#include <string.h>  /* for memset() */
#include <unistd.h>  /* for close() */
#include <fcntl.h>   /* for open(), fcntl() */

int main(int argc, char **argv) {
    /* open file
     * we need O_RDWR for F_SETLK */
    int fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    struct flock fl;
    memset(&fl, 0, sizeof(fl));

    /* lock entire file */
    fl.l_type = F_RDLCK;    /* F_RDLCK is shared lock */
    fl.l_whence = SEEK_SET; /* offset base is start of the file */
    fl.l_start = 0;         /* starting offset is zero */
    fl.l_len = 0;           /* len is zero, which is a special value
                               representing end of file (no matter
                               how large the file grows in future) */

    /* F_SETLKW specifies blocking mode */
    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl(F_SETLKW)");
        exit(EXIT_FAILURE);
```

```
    }

    /* atomically upgrade shared lock to exclusive lock, but only
     * for bytes in range [10; 15)
     *
     * after this call, the process will hold three lock regions:
     *   [0; 10)        - shared lock
     *   [10; 15)       - exclusive lock
     *   [15; SEEK_END) - shared lock
     */
    fl.l_type = F_WRLCK;    /* F_WRLCK is exclusive lock */
    fl.l_whence = SEEK_SET;
    fl.l_start = 10;
    fl.l_len= 5;

    /* F_SETLKW specifies non-blocking mode */
    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl(F_SETLK)");
        exit(EXIT_FAILURE);
    }

    /* release lock for bytes in range [10; 15) */
    fl.l_type = F_UNLCK;

    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl(F_SETLK)");
        exit(EXIT_FAILURE);
    }

    /* close file and release locks for all regions
     * note that locks are released when process calls close() on any
     * descriptor for a lock file */
    close(fd);

    return EXIT_SUCCESS;
}
```

## lockf function

This example demonstrates usage of `lockf` function (POSIX XSI).

Notes:

- Only exclusive locks are supported.
- Can be applied to a byte range, optionally automatically expanding when data is appended in future (controlled by `len` argument and position set with `lseek` function).
- Locks are released on first close by the locking process of *any* file descriptor for the file, or when process terminates.
- The interaction between `fcntl` and `lockf` locks is unspecified. On Linux, `lockf` is a wrapper for POSIX record locks.

```
#include <stdlib.h>  /* for exit() */
#include <stdio.h>   /* for perror() */
#include <unistd.h>  /* for lockf(), lseek() */
#include <fcntl.h>   /* for open() */
```

```
int main(int argc, char **argv) {
    /* open file
     * we need O_RDWR for lockf */
    int fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    /* set current position to byte 10 */
    if (lseek(fd, 10, SEEK_SET) == -1) {
        perror("lseek");
        exit(EXIT_FAILURE);
    }

    /* acquire exclusive lock for bytes in range [10; 15)
     * F_LOCK specifies blocking mode */
    if (lockf(fd, F_LOCK, 5) == -1) {
        perror("lockf(LOCK)");
        exit(EXIT_FAILURE);
    }

    /* release lock for bytes in range [10; 15) */
    if (lockf(fd, F_ULOCK, 5) == -1) {
        perror("lockf(ULOCK)");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Read File locks online: https://riptutorial.com/posix/topic/5581/file-locks

# Chapter 3: Filesystem

## Examples

### Count number of text files in the directory

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <dirent.h>
#include <ctype.h>

int main(int argc, char **argv)
{
    const char *dname = (argc > 1 ? argv[1] : ".");
    DIR* pdir = opendir(dname);
    if(pdir == NULL) {
        perror("Can't open directory");
        return EXIT_FAILURE;
    }
    int textfiles=0;
    struct dirent* dent;
    while ((dent=readdir(pdir))!=NULL) {
        int d_name_len = strlen(dent->d_name);
        if (d_name_len > 4 &&
                strcmp(dent->d_name + (d_name_len - 4), ".txt") == 0) {
            textfiles++;
        }
    }
    printf("Number of text files: %i\n", textfiles);
    closedir(pdir);
    return 0;
}
```

### Remove files recursively (nftw, not thread-safe)

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>  /* for exit() */
#include <stdio.h>   /* for remove() */
#include <ftw.h>     /* for nftw() */

int unlink_cb(
    const char *fpath, const struct stat *sb, int typeflag, struct FTW *ftwbuf)
{
    return remove(fpath);
}

int rm_rf(const char *path)
{
    return nftw(path,
                unlink_cb,
                64 /* number of simultaneously opened fds, up to OPEN_MAX */,
                FTW_DEPTH | FTW_PHYS);
```

```
}
```

`FTW_PHYS` flag means do not follow symbolic links

`FTW_DEPTH` flag does a post-order traversal, that is, call `unlink_cb()` for the directory itself after handling the contents of the directory and its subdirectories.

`nftw` is interrupted if callback function returns non-zero value.

*Note: this method is not thread-safe , because `nftw` uses `chdir`.*

## Remove files recursively (openat and unlinkat, thread-safe)

```
#include <stddef.h>   /* for offsetof() */
#include <stdlib.h>   /* for exit() */
#include <stdio.h>    /* for perror() */
#include <string.h>   /* for strcmp() */
#include <unistd.h>   /* for close(), unlink() */
#include <fcntl.h>    /* for open() */
#include <dirent.h>   /* for DIR */
#include <sys/stat.h> /* for stat */
#include <limits.h>   /* for NAME_MAX */

int rm_rf(const char* path)
{
    if (unlink(path) == 0) {
        return 0;
    } else {
        int dirfd = open(path, O_RDONLY | O_DIRECTORY);
        if (dirfd == -1) {
            perror("open");
            return -1;
        }
        if (rm_children(dirfd) == -1) {
            return -1;
        }
        if (rmdir(path) == -1) {
            perror("rmdir");
            return -1;
        }
        return 0;
    }
}

int rm_children(int dirfd)
{
    DIR* dir = fdopendir(dirfd);
    if (dir == NULL) {
        perror("fdopendir");
        if (close(dirfd) == -1) {
            perror("close");
        }
        return -1;
    }

    char buf[offsetof(struct dirent, d_name) + NAME_MAX + 1];

    struct dirent* dbuf = (struct dirent*)buf;
```

```
    struct dirent* dent = NULL;

    int ret = 0;

    for (;;) {
        if ((ret = readdir_r(dir, dbuf, &dent)) == -1) {
            perror("readdir_r");
            break;
        }
        if (dent == NULL) {
            break;
        }
        if (strcmp(dent->d_name, ".") == 0 || strcmp(dent->d_name, "..") == 0) {
            continue;
        }
        if ((ret = rm_at(dirfd, dent->d_name)) == -1) {
            break;
        }
    }

    if (closedir(dir) == -1) {
        perror("closedir");
        ret = -1;
    }

    return ret;
}

int rm_at(int dirfd, const char* name)
{
    int fd = openat(dirfd, name, O_RDONLY);
    if (fd == -1) {
        perror("openat");
        return -1;
    }

    int ret = 0;

    struct stat st;
    if ((ret = fstat(fd, &st)) == -1) {
        perror("fstat");
        goto out;
    }

    if (S_ISDIR(st.st_mode)) {
        ret = rm_children(fd);
        fd = -1;
        if (ret == -1) {
            goto out;
        }
    }

    ret = unlinkat(dirfd, name, S_ISDIR(st.st_mode) ? AT_REMOVEDIR : 0);
    if (ret == -1) {
        perror("unlinkat");
        goto out;
    }

out:
    if (fd != -1) {
        if (close(fd) == -1) {
```

```
            perror("close");
            ret = -1;
        }
    }

    return ret;
}
```

*Note: this method is thread-safe, but uses stack for uncontrolled recursion. Each tree level adds at least NAME_MAX bytes + one frame size.*

Read Filesystem online: https://riptutorial.com/posix/topic/5586/filesystem

# Chapter 4: Input/Output multiplexing

## Introduction

IO may be blocking/non-blocking and synchronous/asynchronous. POSIX API provides synchronous blocking API (e.g. classic read, write, send, recv calls), synchronous non-blocking API (same functions, file descriptors opened with `O_NONBLOCK` flag and IO-multiplexing calls) and asynchonous API (functions starting with `aio_`).

Synchronous API is usually used with "one thread/process per fd" style. This is dreadful for resources. Non-blocking API allows to operate with a set of fds in one thread.

## Examples

### Poll

In this example we create a pair of connected sockets and send 4 strings from one to another and print received strings to console. Note, that the number of times we will call send may not be equal to number of times we call recv

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <poll.h>

#define BUFSIZE 512

int main()
{
    #define CKERR(msg) {if(ret < 0) { perror(msg); \
        close(sockp[0]); close(sockp[1]); exit(EXIT_FAILURE); } }
    const char* strs_to_write[] = {"hello ", "from ", "other ", "side "};

    int sockp[2] = {-1, -1};
    ssize_t ret = socketpair (AF_UNIX, SOCK_STREAM, 0, sockp);
    CKERR("Socket pair creation error")

    struct pollfd pfds[2];
    for(int i=0; i<2; ++i) {
        pfds[i] = (struct pollfd){sockp[i], POLLIN|POLLOUT, 0};
        fcntl(sockp[i], F_SETFL|O_NONBLOCK); // nonblocking fds are
                // literally mandatory for IO multiplexing; non-portable
    }
    char buf[BUFSIZE];

    size_t snt = 0, msgs = sizeof(strs_to_write)/sizeof(char*);
    while(1) {
        int ret = poll(pfds,
```

```
            2 /*length of pollfd array*/,
            5 /*milliseconds to wait*/);
        CKERR("Poll error")

        if (pfds[0].revents & POLLOUT && snt < msgs) {
            // Checking POLLOUT before writing to ensure there is space
            // available in socket's kernel buffer to write, otherwise we
            // may face EWOULDBLOCK / EAGAIN error
            ssize_t ret = send(sockp[0], strs_to_write[snt], strlen(strs_to_write[snt]), 0);
            if(++snt >= msgs)
                close(sockp[0]);
            CKERR("send error")
            if (ret == 0) {
                puts("Connection closed");
                break;
            }
            if (ret > 0) {
                // assuming that all bytes were written
                // if ret != %sent bytes number%, send other bytes later
            }
        }
        if (pfds[1].revents & POLLIN) {
            // There is something to read
            ssize_t ret = recv(sockp[1], buf, BUFSIZE, 0);
            CKERR("receive error")
            if (ret == 0) {
                puts("Connection closed");
                break;
            }
            if (ret > 0) {
                printf("received str: %.*s\n", (int)ret, buf);
            }
        }

    }
    close(sockp[1]);
    return EXIT_SUCCESS;
}
```

## Select

Select is another way to do I/O multiplexing. One of it's advantages is an existance in winsock API. Moreover, on Linux, select() modifies timeout to reflect the amount of time not slept; most other implementations do not do this. (POSIX.1 permits either behavior.)

Both poll and select have ppoll and pselect alternatives, which allow handling incoming signals during waiting for event. And both of them become slow with huge amount of file descriptors (one hundred and more), so it would be wise to choose platform specific call, e.g. `epoll` on Linux and `kqueue` on FreeBSD. Or switch to asynchronous API (POSIX `aio` e.g. or something specific like IO Completion Ports).

Select call has the following prototype:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
       fd_set *exceptfds, struct timeval *timeout);
```

`fd_set` is a bitmask array of file descriptors,

`nfds` is the maximum number of all file descriptors in set + 1.

Snippet of working with select:

```
fd_set active_fd_set, read_fd_set;
FD_ZERO (&active_fd_set); // set fd_set to zeros
FD_SET (sock, &active_fd_set); // add sock to the set
// # define FD_SETSIZE sock + 1
while (1) {
    /* Block until input arrives on one or more active sockets. */
    read_fd_set = active_fd_set; // read_fd_set gets overriden each time
    if (select (FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
        // handle error
    }
    // Service all file descriptors with input pending.
    for (i = 0; i < FD_SETSIZE; ++i) {
        if (FD_ISSET (i, &read_fd_set)) {
            // there is data for i
    }
}
```

Note, that on most POSIX implemetations file descriptors associated with files on disk are blocking. So writing to file, even if this file was set in `writefds`, would block until all bytes won't be dumped to disk

Read Input/Output multiplexing online: https://riptutorial.com/posix/topic/6547/input-output-multiplexing

# Chapter 5: Pipes

## Introduction

Pipes are a mechanism for unidirectional interprocess or interthread communication within the scope of a single machine. Logically, a pipe consists of two connected termini, one to which data can be written, and another from which that data can subsequently be read, with a data buffer between such that writes and reads are not required to be synchronous. Pipes should be distinguished from shell *pipelines*, which are an application of pipes.

## Examples

### Basic creation and usage

Anonymous pipes, or simply pipes, are kernel-managed objects exposed to processes as a pair of file descriptors, one for the read terminus and one for the write terminus. They are created via the `pipe`(2) function:

```
int pipefds[2];
int result;

result = pipe(pipefds);
```

On success, `pipe()` records the descriptor for the read end of the pipe at index 0 of the provided array, and the descriptor for the write end at index 1; these indices are analogous to the conventional file descriptor numbers for the standard streams.

Having created a pipe, one uses POSIX I/O functions to write to the write end or read from the read end:

*Process 1:*

```
ssize_t bytes_written = write(pipefds[1], "Hello, World!", 14);
```

*Process 2:*

```
char buffer[256];
ssize_t bytes_read = read(pipefds[0], buffer, sizeof(buffer));
```

Alternatively, one can instead use `fdopen()` to wrap one of both pipe ends in a `FILE` structure for use with C stdio functions:

```
FILE *write_end = fdopen(pipefds[1]);

if (write_end) {
    fputs("Hello, World!");
```

```
}
```

Pipes have finite I/O buffers, and ordinary writes to pipes with full buffers will block. Pipes are therefore not a safe mechanism for a thread to communicate with itself, as if the thread writing to a pipe is also the only one that reads from it, then as soon as a write blocks that thread is deadlocked.

A pipe persists as long as any process has an open file description for either of the pipe ends. The `close`(2) function can be used to close a pipe end represented by a file descriptor, and `fclose`(3) can be used to close a pipe end via a `FILE` wrapped around it.

## Establishing a pipe to a child process

File descriptors and `FILE` objects are per-process resources that cannot themselves be exchanged between processes via ordinary I/O. Therefore, in order for two distinct processes to communicate via an anonymous pipe, one or both participating processes must inherit an open pipe end from the process that created the pipe, which must therefore be the parent process or a more distant ancestor. The simplest case is the one in which a parent process wants to communicate with a child process.

Because the child process must inherit the needed open file description from its parent, the pipe must be created first. The parent then forks. Ordinarily, each process will be either strictly a reader or strictly a writer; in that case, each one should close the pipe end that it does not intend to use.

```
void demo() {
    int pipefds[2];
    pid_t pid;

    // Create the pipe
    if (pipe(pipefds)) {
        // error - abort ...
    }

    switch (pid = fork()) {
        case -1:
            // error - abort ...
            break;
        case 0:   /* child */
            close(pipefds[0]);
            write(pipefds[1], "Goodbye, and thanks for all the fish!", 37);
            exit(0);
        default:  /* parent */
            close(pipefds[1]);

            char buffer[256];
            ssize_t nread = read(pipefds[0], sizeof(buffer) - 1);

            if (nread >= 0) {
                buffer[nread] = '\0';
                printf("My child said '%s'\n", buffer);
            }

            // collect the child
            wait(NULL);
```

```
            break;
    }
}
```

## Connecting two child processes via a pipe

Connecting two child processes via a pipe is performed by connecting each of two children to the parent via different ends of the same pipe. Usually, the parent will not be party to the conversation between the children, so it closes its copies of both pipe ends.

```
int demo() {
    int pipefds[2];
    pid_t child1, child2;

    if (pipe(pipefds)) {
        // error - abort ...
    }

    switch (child1 = fork()) {
        case -1:
            // error - abort
            break;
        case 0:   /* child 1 */
            close(pipefds[0]);
            write(pipefds[1], "Hello, brother!", 15);
            exit(0);
        default:  /* parent */
            // nothing
    }

    switch (child1 = fork()) {
        case -1:
            // error - abort
            break;
        case 0:   /* child 2 */
            char buffer[256];
            ssize_t nread;

            close(pipefds[1]);
            nread = read(pipefds[0], buffer, sizeof(buffer) - 1);
            if (nread < 0) {
                // handle error
            } else {
                buffer[nread] = '\0';
                printf("My brother told me '%s'\n", buffer);
            }
            exit(0);
        default:  /* parent */
            // nothing
    }

    // Only the parent reaches this point
    close(pipefds[0]);
    close(pipefds[1]);
    if (child1 >= 0) {
        wait(NULL);
        if (child2 >= 0) {
```

```
            wait(NULL);
        }
    }
}
```

## Creating a shell-style pipeline

A shell-style pipeline consists of two or more processes, each one with its standard output connected to the standard input of the next. The output-to-input connections are built on pipes. To establish a pipeline-like set of processes, one creates pipe-connected child processes as described in another example, and furthermore uses the `dup2`(2) function to duplicate each pipe end onto the appropriate standard file descriptor of its process. It is generally a good idea to then close the original pipe-end file descriptor, especially if, as is often the case, one intends for the child to afterward exec a different command.

```
// Most error handling omitted for brevity, including ensuring that pipe ends are
// always closed and child processes are always collected as needed; see other
// examples for more detail.
int demo() {
    int pipefds[2];
    pid_t child1 = -1, child2 = -1;

    pipe(pipefds);

    switch (child1 = fork()) {
        case -1:
            // handle error ...
            break;
        case 0:   /* child 1 */
            close(pipefds[0]);
            dup2(pipefds[1], STDOUT_FILENO);
            close(pipefds[1]);
            execl("/bin/cat", "cat", "/etc/motd", NULL);
            exit(1);  // execl() returns only on error
        default:  /* parent */
            // nothing
    }

    switch (child2 = fork()) {
        case -1:
            // handle error ...
            break;
        case 0:   /* child 2 */
            close(pipefds[1]);
            dup2(pipefds[0], STDIN_FILENO);
            close(pipefds[0]);
            execl("/bin/grep", "grep", "[Ww]ombat", NULL);
            exit(1);  // execl() returns only on error
        default:  /* parent */
            // nothing
    }

    // Only the parent reaches this point
    close(pipefds[0]);
    close(pipefds[1]);
    wait(NULL);
    wait(NULL);
```

```
}
```

One of the more common errors in setting up a pipeline is to forget to have one or more of the processes involved close the pipe ends that it is not using. As a rule of thumb, the end result should be that each pipe end is owned by exactly one process, and is open only in that process.

In a case such as the demo function above, failure the second child or of the parent to close `pipefds[1]` will result in the second child hanging, for `grep` will continue to wait for input until it sees EOF, and that will not be observed as long as the write end of the pipe remains open in any process, such as the parent process or the second child itself.

Read Pipes online: https://riptutorial.com/posix/topic/8082/pipes

# Chapter 6: Processes

## Syntax

- pid_t getpid(void);
- pid_t getppid(void);
- pid_t fork(void);
- pid_t waitpid(pid_t pid, int *wstatus, int options);
- int execv(const char *path, char *const argv[]);

## Parameters

| Function, Parameter(s), Return Value | Description |
|---|---|
| `fork()` | function name |
| none | n/a |
| Returns PID, 0, or -1 | The calling process receives the PID of the newly create process or -1 on failure. The child (the newly created process) receive 0. In case of failure set `errno` to either `EAGAIN` or `ENOMEM` |
| - | - |
| `execv()` | function name |
| `const char *path` | String containing name of to executable (might inlcude path) |
| `char *const argv[]` | Array of string pointer as arguments |
| Returns -1 on failure | On success this function does not return. |
| - | - |

## Examples

### Create child process and wait until it exits

This program demonstrates how to run another process using `fork()` and wait its termination using `waitpid()`:

- `fork()` creates an identical copy of the current process. The original process is the parent process, while the newly created one is the child process. Both processes continue exactly

after the `fork()`.

- `waitpid()` blocks until child process exits or terminates and returns its exit code and termination reason.

```
#include <unistd.h>     /* for fork(), getpid() */
#include <sys/types.h>  /* for waitpid() */
#include <sys/wait.h>   /* for waitpid() */
#include <stdlib.h>     /* for exit() */
#include <stdio.h>      /* for printf(), perror() */

int
main(int argc, char *argv[])
{
    /* Create child process.
     *
     * On success, fork() returns the process ID of the child (> 0) to the
     * parent and 0 to the child. On error, -1 is returned.
     */
    pid_t child_pid = fork();

    if (child_pid < 0) {
        perror("fork() failed");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        /* Print message from child process.
         *
         * getpid() returns the PID (process identifier) of current process,
         * which is typically int but doesn't have to be, so we cast it.
         *
         * getppid() returns the PID of the parent process.
         */
        printf("from child: pid=%d, parent_pid=%d\n",
                 (int)getpid(), (int)getppid());

        /* We can do something here, e.g. load another program using exec().
         */
        exit(33);
    } else if (child_pid > 0) {
        /* Print message from parent process.
         */
        printf("from parent: pid=%d child_pid=%d\n",
                  (int)getpid(), (int)child_pid);

        /* Wait until child process exits or terminates.
         *
         * The return value of waitpid() is PID of the child process, while
         * its argument is filled with exit code and termination reason.
         */
        int status;
        pid_t waited_pid = waitpid(child_pid, &status, 0);

        if (waited_pid < 0) {
            perror("waitpid() failed");
            exit(EXIT_FAILURE);
        } else if (waited_pid == child_pid) {
            if (WIFEXITED(status)) {
                /* WIFEXITED(status) returns true if the child has terminated
                 * normally. In this case WEXITSTATUS(status) returns child's
                 * exit code.
```

```
                 */
                printf("from parent: child exited with code %d\n",
                          WEXITSTATUS(status));
            }
        }
    }

    exit(EXIT_SUCCESS);
}
```

---

Example output:

```
from parent: pid=2486 child_pid=2487
from child: pid=2487, parent_pid=2486
from parent: child exited with code 33
```

---

Copied from here, originally created by M.Geiger.

Read Processes online: https://riptutorial.com/posix/topic/4534/processes

# Chapter 7: Signals

## Syntax

- unsigned alarm(unsigned seconds);
- int kill(pid_t pid, int sig);

## Parameters

| Function, Parameter(s), Return Value | Description |
|---|---|
| `alarm()` | function name |
| `unsigned seconds` | Seconds to raise an alarm or 0 to cancel any pending alarm |
| >= 0 | 0 if no other alarm was pending, else the number of seconds the pending alarm still had open. This function won't fail. |
| - | - |
| `kill()` | function name |
| `pid_t pid` | . |
| `int sig` | 0 or signal ID |
| 0, -1 | On success 0 is returned, -1 on failure with setting `errno` to `EINVAL`, `EPERM`or `ESRCH`. |

## Examples

### Raising SIGALARM with the default action

Using `alarm`, user can schedule `SIGALARM` signal to be raised after specified interval. In case user did not blocked, ignored or specified explicit signal handler for this signal, the default action for this signal will be performed on arrival. Per specification default action for `SIGALARM` is to terminate the process:

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

int main (int argc, char** argv)
{
    printf("Hello!\n");
```

```
    // Set the alarm for five second
    alarm(5); // Per POSIX, this cannot fail

    // Now sleep for 15 seconds
    for (int i = 1; i <= 15; i++)
    {
        printf("%d\n", i);
        sleep(1);
    }

    // And print the message before successful exit
    printf("Goodbye!\n");

    return EXIT_SUCCESS;
}
```

This outputs:

```
Hello!
1
2
3
4
5
[2]    35086 alarm      ./a.out
```

## Setting signal handler using sigaction and raising signals using raise

In order for a program to react to a certain signal, other than using default action, custom signal handler can be installed using `sigaction`. `sigaction` receives three arguments - signal to act on, pointer to `sigaction_t` structure which, if not `NULL`, is describing new behaviour and pointer to `sigaction_t` which, if not `NULL` will be filled with the old behaviour (so one can restore it). Raising signals in same process can be done with `raise` method. If more control is needed (to send the signal to some other process, `kill` or `pthread_kill` can be used, which accept the destination process id or thread id).

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

// Signals are numbered from 1, signal 0 doesn't exist
volatile sig_atomic_t last_received_signal = 0;

// Signal handler, will set the global variable
// to indicate what is the last signal received.
// There should be as less work as possible inside
// signal handler routine, and one must take care only
// to call reentrant functions (in case of signal arriving
// while program is already executing same function)
void signal_catcher(int signo, siginfo_t *info, void *context)
{
    last_received_signal = info->si_signo;
}
```

```
int main (int argc, char** argv)
{
    // Setup a signal handler for SIGUSR1 and SIGUSR2
    struct sigaction act;
    memset(&act, 0, sizeof act);

    // sigact structure holding old configuration
    // (will be filled by sigaction):
    struct sigaction old_1;
    memset(&old_1, 0, sizeof old_1);
    struct sigaction old_2;
    memset(&old_2, 0, sizeof old_2);

    act.sa_sigaction = signal_catcher;
    // When passing sa_sigaction, SA_SIGINFO flag
    // must be specified. Otherwise, function pointed
    // by act.sa_handler will be invoked
    act.sa_flags = SA_SIGINFO;

    if (0 != sigaction(SIGUSR1, &act, &old_1))
    {
        perror("sigaction () failed installing SIGUSR1 handler");
        return EXIT_FAILURE;
    }

    if (0 != sigaction(SIGUSR2, &act, &old_2))
    {
        perror("sigaction() failed installing SIGUSR2 handler");
        return EXIT_FAILURE;
    }

    // Main body of "work" during which two signals
    // will be raised, after 5 and 10 seconds, and which
    // will print last received signal
    for (int i = 1; i <= 15; i++)
    {
        if (i == 5)
        {
            if (0 != raise(SIGUSR1))
            {
                perror("Can't raise SIGUSR1");
                return EXIT_FAILURE;
            }
        }

        if (i == 10)
        {
            if (0 != raise(SIGUSR2))
            {
                perror("Can't raise SIGUSR2");
                return EXIT_FAILURE;
            }
        }

        printf("Tick #%d, last caught signal: %d\n",
            i, last_received_signal);

        sleep(1);
    }
```

```
    // Restore old signal handlers
    if (0 != sigaction(SIGUSR1, &old_1, NULL))
    {
        perror("sigaction() failed restoring SIGUSR1 handler");
        return EXIT_FAILURE;
    }

    if (0 != sigaction(SIGUSR2, &old_2, NULL))
    {
        perror("sigaction() failed restoring SIGUSR2 handler");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

This outputs:

```
Tick #1, last caught signal: 0
Tick #2, last caught signal: 0
Tick #3, last caught signal: 0
Tick #4, last caught signal: 0
Tick #5, last caught signal: 30
Tick #6, last caught signal: 30
Tick #7, last caught signal: 30
Tick #8, last caught signal: 30
Tick #9, last caught signal: 30
Tick #10, last caught signal: 31
Tick #11, last caught signal: 31
Tick #12, last caught signal: 31
Tick #13, last caught signal: 31
Tick #14, last caught signal: 31
Tick #15, last caught signal: 31
```

## A process committing suicide using kill()

A process can (try to) send a signal to any other process using the `kill()` function.

To do so, the sending process needs to known the receiving process' PID. As, without introducing a race, a process can only be sure of its own PID (and the PIDs of its children) the most simple example to demonstrate the usage of `kill()` is to have a process send a signal to itself.

Below an example of a process initiating its own termination by sending itself a kill-signal (`SIGKILL`):

```
#define _POSIX_C_SOURCE 1

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


int main (void)
{
  pid_t pid = getpid(); /* Get my iown process ID. */
```

```
  kill(pid, SIGKILL); /* Send myself a KILL signal. */

  puts("Signal delivery initiated.");  /* Although not guaranteed,
                                  practically the program never gets here. */

  pause(); /* Wait to die. */

  puts("This never gets printed.");
}
```

Output:

```
Killed
```

(... or alike, depending on the implementation)

## Handle SIGPIPE generated by write() in a thread-safe manner

When `write()` is called for a named or unnamed pipe or stream socket whose reading end is closed, two things happen:

POSIX.1-2001

1. `SIGPIPE` signal is sent to the *process* that called `write()`

POSIX.1-2004

1. `SIGPIPE` signal is sent to the *thread* that called `write()`
2. `EPIPE` error is returned by `write()`

---

There are several ways to deal with `SIGPIPE`:

- For sockets, `SIGPIPE` may be disabled by setting platform-specific options like `MSG_NOSIGNAL` in Linux and `SO_NOSIGPIPE` in BSD (works only for `send`, but not for `write`). This is not portable.
- For FIFOs (named pipes), `SIGPIPE` will not be generated if writer uses `O_RDWR` instead of `O_WRONLY`, so that reading end is always opened. However, this disables `EPIPE` too.
- We can ignore `SIGPIPE` or set global handler. This is a good solution, but it's not acceptable if you don't control the whole application (e.g. you're writing a library).
- With recent POSIX versions, we can use the fact that `SIGPIPE` is send to the thread that called `write()` and handle it using synchronous signal handling technique.

---

Code below demonstrates thread-safe `SIGPIPE` handling for POSIX.1-2004 and later.

It's inspired by this post:

- First, add `SIGPIPE` to signal mask of current thread using `pthread_sigmask()`.
- Check if there is already pending `SIGPIPE` using `sigpending()`.
- Call `write()`. If reading end is closed, `SIGPIPE` will be added to pending signals mask and `EPIPE` will be returned.
- If `write()` returned `EPIPE`, and `SIGPIPE` was not already pending before `write()`, remove it from pending signals mask using `sigtimedwait()`.
- Restore original signal mask using `pthread_sigmask()`.

Source code:

```
#include <unistd.h>
#include <time.h>
#include <errno.h>
#include <sys/signal.h>

ssize_t safe_write(int fd, const void* buf, size_t bufsz)
{
    sigset_t sig_block, sig_restore, sig_pending;

    sigemptyset(&sig_block);
    sigaddset(&sig_block, SIGPIPE);

    /* Block SIGPIPE for this thread.
     *
     * This works since kernel sends SIGPIPE to the thread that called write(),
     * not to the whole process.
     */
    if (pthread_sigmask(SIG_BLOCK, &sig_block, &sig_restore) != 0) {
        return -1;
    }

    /* Check if SIGPIPE is already pending.
     */
    int sigpipe_pending = -1;
    if (sigpending(&sig_pending) != -1) {
        sigpipe_pending = sigismember(&sig_pending, SIGPIPE);
    }

    if (sigpipe_pending == -1) {
        pthread_sigmask(SIG_SETMASK, &sig_restore, NULL);
        return -1;
    }

    ssize_t ret;
    while ((ret = write(fd, buf, bufsz)) == -1) {
        if (errno != EINTR)
            break;
    }

    /* Fetch generated SIGPIPE if write() failed with EPIPE.
     *
     * However, if SIGPIPE was already pending before calling write(), it was
     * also generated and blocked by caller, and caller may expect that it can
     * fetch it later. Since signals are not queued, we don't fetch it in this
     * case.
     */
    if (ret == -1 && errno == EPIPE && sigpipe_pending == 0) {
        struct timespec ts;
```

```
        ts.tv_sec = 0;
        ts.tv_nsec = 0;

        int sig;
        while ((sig = sigtimedwait(&sig_block, 0, &ts)) == -1) {
            if (errno != EINTR)
                break;
        }
    }

    pthread_sigmask(SIG_SETMASK, &sig_restore, NULL);
    return ret;
}
```

Read Signals online: https://riptutorial.com/posix/topic/4532/signals

# Chapter 8: Sockets

## Examples

### TCP Concurrent Echo Server

In this example, we'll create a simple echo server that will listen on the specified port, and being able to handle new connections:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <time.h>

/**
  Connection handler - this will be executed in
  the new process, after forking, and it will read
  all the data from the socket, while available and
  to echo it on the local terminal.

  Params:
    sd = socket to the client
*/
#define BUF_SIZE (1024)

int echo_client(int sd)
{
    int result = 0;

    char buf[BUF_SIZE + 1] = {0};

    ssize_t n_read;
    while (0 < (n_read = read(sd, buf, BUF_SIZE)))
    {
        buf[n_read] = '\0';
        printf("%s\n", buf);
    }

    if (0 > n_read)
    {
      perror("read() failed");
      result = -1;
    }
    else
    {
      fprintf(stderr, "The other side orderly shut down the connection.\n");
    }

    close(sd);

    return result;
}
```

```
int main(void)
{
    // Create a listening socket
    int listening_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (listening_socket == -1)
    {
        perror("socket() failed");
        return EXIT_FAILURE;
    }

    // Bind it to port 15000.
    unsigned short listening_port = 15000;

    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_port = htons(listening_port);

    socklen_t sock_len = sizeof(addr);

    if (0 > (bind(listening_socket, (const struct sockaddr*) &addr, sock_len)))
    {
        perror("bind() failed");
        return EXIT_FAILURE;
    }

    // Start listening
    if (0 > listen(listening_socket, 0))
    {
        perror("listen() failed");
        return EXIT_FAILURE;
    }

    // Accept new connections, fork the new process for handling
    // and handle the connection in the new process, while the parent
    // is waiting for another connection to arrive.
    int accepted_socket = 0;
    while (0 < (accepted_socket =
                  accept(listening_socket, (struct sockaddr*) &addr, &sock_len)))
    {
        pid_t pid_child = fork();

        if (0 > pid_child)
        {
            perror("fork() failed");
            return EXIT_FAILURE;
        }
        else if (0 == pid_child)
        {
            // inside the forked child here
            close(listening_socket); // The child does not need this any more.

            echo_client(accepted_socket);

            return EXIT_SUCCESS;
        }

        // Inside parent process, since file descriptors are reference
        // counted, we need to close the client socket
        close(accepted_socket);
```

```
    }

    if (0 > accepted_socket)
    {
        perror("accept() failed");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

## Enabling TCP keepalive at server side

This is a client-server example. The process forks and runs client in the parent process and server in the child process:

- client connects to server and waits until server exits;
- server accepts connection from client, enables keepalive, and waits any signal.

Keepalive is configured using the following options described in `socket(7)` and `tcp(7)` man pages:

- `SO_KEEPALIVE` - enables sending of keep-alive messages
- `TCP_KEEPIDLE`- the time (in seconds) the connection needs to remain idle before TCP starts sending keepalive probes
- `TCP_KEEPINTVL` - the time (in seconds) between individual keepalive probes
- `TCP_KEEPCNT` - the maximum number of keepalive probes TCP should send before dropping the connection

Source code:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#define check(expr) if (!(expr)) { perror(#expr); kill(0, SIGTERM); }

void enable_keepalive(int sock) {
    int yes = 1;
    check(setsockopt(
        sock, SOL_SOCKET, SO_KEEPALIVE, &yes, sizeof(int)) != -1);

    int idle = 1;
    check(setsockopt(
        sock, IPPROTO_TCP, TCP_KEEPIDLE, &idle, sizeof(int)) != -1);

    int interval = 1;
    check(setsockopt(
        sock, IPPROTO_TCP, TCP_KEEPINTVL, &interval, sizeof(int)) != -1);
```

```
    int maxpkt = 10;
    check(setsockopt(
        sock, IPPROTO_TCP, TCP_KEEPCNT, &maxpkt, sizeof(int)) != -1);
}

int main(int argc, char** argv) {
    check(argc == 2);

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(12345);
    check(inet_pton(AF_INET, argv[1], &addr.sin_addr) != -1);

    int server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    check(server != -1);

    int yes = 1;
    check(setsockopt(server, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) != -1);

    check(bind(server, (struct sockaddr*)&addr, sizeof(addr)) != -1);
    check(listen(server, 1) != -1);

    if (fork() == 0) {
        int client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        check(client != -1);
        check(connect(client, (struct sockaddr*)&addr, sizeof(addr)) != -1);
        printf("connected\n");
        pause();
    }
    else {
        int client = accept(server, NULL, NULL);
        check(client != -1);
        enable_keepalive(client);
        printf("accepted\n");
        wait(NULL);
    }

    return 0;
}
```

Keepalive packets may be monitored using `tcpdump`.

Example usage:

```
$ ./a.out 127.0.0.1 &
[1] 14010
connected
accepted

$ tcpdump -n -c4 -ilo port 12345
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
18:00:35.173892 IP 127.0.0.1.12345 > 127.0.0.1.60998: Flags [.], ack 510307430, win 342,
options [nop,nop,TS val 389745775 ecr 389745675], length 0
18:00:35.173903 IP 127.0.0.1.60998 > 127.0.0.1.12345: Flags [.], ack 1, win 342, options
[nop,nop,TS val 389745775 ecr 389745075], length 0
18:00:36.173886 IP 127.0.0.1.12345 > 127.0.0.1.60998: Flags [.], ack 1, win 342, options
[nop,nop,TS val 389745875 ecr 389745775], length 0
18:00:36.173898 IP 127.0.0.1.60998 > 127.0.0.1.12345: Flags [.], ack 1, win 342, options
```

```
[nop,nop,TS val 389745875 ecr 389745075], length 0
4 packets captured
8 packets received by filter
0 packets dropped by kernel
```

## TCP Daytime Iterative Server

This is a TCP daytime iterative server kept as simple as possible.

```
#include <sys/types.h>   /* predefined types */
#include <unistd.h>      /* unix standard library */
#include <arpa/inet.h>   /* IP addresses conversion utilities */
#include <netinet/in.h>  /* sockaddr_in structure definition */
#include <sys/socket.h>  /* berkley socket library */
#include <stdio.h>       /* standard I/O library */
#include <string.h>      /* include to have memset */
#include <stdlib.h>      /* include to have exit */
#include <time.h>        /* time manipulation primitives */

#define MAXLINE 80
#define BACKLOG 10

int main(int argc, char *argv[])
{
    int list_fd, conn_fd;
    struct sockaddr_in serv_add;
    char buffer[MAXLINE];
    time_t timeval;

    /* socket creation third parameter should be IPPROTO_TCP but 0 is an
     * accepted value */
    list_fd = socket(AF_INET, SOCK_STREAM, 0);

    /* address initialization */
    memset(&serv_add, 0, sizeof(serv_add));        /* init the server address */
    serv_add.sin_family = AF_INET;                 /* address type is IPV4 */
    serv_add.sin_port = htons(13);                 /* daytime port is 13 */
    serv_add.sin_addr.s_addr = htonl(INADDR_ANY);  /* connect from anywhere */

    /* bind socket */
    bind(list_fd, (struct sockaddr *)&serv_add, sizeof(serv_add));

    /* listen on socket */
    listen(list_fd, BACKLOG);

    while (1)
    {
        /* accept connection */
        conn_fd = accept(list_fd, (struct sockaddr *) NULL, NULL);

        timeval = time(NULL);
        snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));

        write(conn_fd, buffer, strlen(buffer)); /* write daytime to client */

        close(conn_fd);
    }

    /* normal exit */
```

```
    close(list_fd);
    exit(0);
}
```

## TCP Daytime Client

This is a TCP daytime client kept as simple as possible.

```
#include <unistd.h>      /* unix standard library */
#include <arpa/inet.h>   /* IP addresses manipulation utilities */
#include <netinet/in.h>  /* sockaddr_in structure definition */
#include <sys/socket.h>  /* berkley socket library */
#include <stdio.h>       /* standard I/O library */
#include <string.h>      /* include to have memset*/
#include <stdlib.h>      /* include to have exit*/

#define MAXLINE 1024

int main(int argc, char *argv[])
{
    int sock_fd;
    int nread;
    struct sockaddr_in serv_add;
    char buffer[MAXLINE];

    /* socket creation third parameter should be IPPROTO_TCP but 0 is an
     * accepted value*/
    sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    /* address initialization */
    memset(&serv_add, 0, sizeof(serv_add));     /* init the server address */
    serv_add.sin_family = AF_INET;              /* address type is IPV4 */
    serv_add.sin_port = htons(13);              /* daytime post is 13 */

    /* using inet_pton to build address */
    inet_pton(AF_INET, argv[1], &serv_add.sin_addr);

    /* connect to the server */
    connect(sock_fd, (struct sockaddr *)&serv_add, sizeof(serv_add));

    /* read daytime from server */
    while ((nread = read(sock_fd, buffer, MAXLINE)) > 0)
    {
        buffer[nread] = 0;
        if (fputs(buffer, stdout) == EOF)
        {
            perror("fputs error"); /* write daytime on stdout */
            return -1;
        }
    }

    close(sock_fd);
    exit(0);
}
```

## Socket basics

There are four types of sockets available in POSIX API: TCP, UDP, UNIX, and (optionally) RAW.

Unix domain sockets may act like stream sockets or like datagram sockets.

Some of endpoint types:

1. `struct sockaddr` - universal endpoint type. Typically, other concrete endpoint types are converted to this type only in posix calls.
2. `struct sockaddr_in` - IPv4 endpoint

```
struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port;            /* Port number.  */
    struct in_addr sin_addr;       /* Internet address.  */
};
struct in_addr {
    in_addr_t s_addr;
};
```

3. `struct sockaddr_in6` - IPv6 endpoint

```
struct sockaddr_in6 {
    sa_family_t sin6_family;
    in_port_t sin6_port;    /* Transport layer port # */
    uint32_t sin6_flowinfo;    /* IPv6 flow information */
    struct in6_addr sin6_addr;    /* IPv6 address */
    uint32_t sin6_scope_id;    /* IPv6 scope-id */
};
```

4. `struct sockaddr_un`.

```
struct sockaddr_un {
    sa_family_t sun_family;               /* AF_UNIX */
    char        sun_path[108];            /* pathname */
};
```

# Entire program

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <unistd.h>

#define DESIRED_ADDRESS "127.0.0.1"
#define DESIRED_PORT 3500
#define BUFSIZE 512

int main()
{
    // ADDRESS PART
    // MAIN PART
    close(sock);
    return EXIT_SUCCESS;
```

```
}
```

# Creating IPv4 endpoint

```
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
addr.sin_port = htons(DESIRED_PORT); /*converts short to
                                       short with network byte order*/
addr.sin_addr.s_addr = inet_addr(DESIRED_ADDRESS);
```

# TCP server snippet

```
int sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == -1) {
    perror("Socket creation error");
    return EXIT_FAILURE;
}

if (bind(sock, (struct sockaddr*) &addr, sizeof(addr)) == -1) {
    perror("Bind error");
    close(sock);
    return EXIT_FAILURE;
}

if (listen(sock, 1/*length of connections queue*/) == -1) {
    perror("Listen error");
    close(sock);
    return EXIT_FAILURE;
}

socklen_t socklen = sizeof addr;
int client_sock = accept(sock, &addr, &socklen); /* 2nd and 3rd argument may be NULL. */
if (client_sock == -1) {
    perror("Accept error");
    close(sock);
    return EXIT_FAILURE;
}

printf("Client with IP %s connected\n", inet_ntoa(addr.sin_addr));

char buf[BUFSIZE];
if (send(sock, "hello", 5, 0) == -1) {
    perror("Send error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
}

ssize_t readden = recv(sock, buf, BUFSIZE, 0);
if (readden < 0) {
    perror("Receive error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
```

```
}
else if (readden == 0) {
  fprintf(stderr, "Client orderly shut down the connection.\n");
}
else {readden > 0) {
    if (readden < BUFSIZE)
    {
      fprintf(stderr, "Received less bytes (%zd) then requested (%d).\n",
        readden, BUFSIZE);
    }

    write (STDOUT_FILENO, buf, readden);
}
```

# TCP client snippet

```
int sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == -1) {
    perror("Socket creation error");
    return EXIT_FAILURE;
}
if (connect(sock, (struct sockaddr*) &addr, sizeof(addr)) == -1) {
    perror("Connection error");
    close(sock);
    return EXIT_FAILURE;
}

char buf[BUFSIZE];
if (send(sock, "hello", 5, 0); /*write may be also used*/ == -1) {
    perror("Send error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
}

ssize_t readden = recv(sock, buf, BUFSIZE, 0); /*read may be also used*/
if (readden < 0) {
    perror("Receive error");
    close(client_sock);
    close(sock);
    return EXIT_FAILURE;
}
else if (readden == 0)
{
    fprintf(stderr, "Client orderly shut down the connection.\n");
}
else /* if (readden > 0) */ {
    if (readden < BUFSIZE)
    {
      fprintf(stderr, "Received less bytes (%zd) then requested (%d).\n",
        readden, BUFSIZE);
    }

    write (STDOUT_FILENO, buf, readden);
}
```

# UDP server snippet

```
int sock = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
if (sock == -1) {
    perror("Socket creation error");
    return EXIT_FAILURE;
}
if (bind(sock, (struct sockaddr*) &addr, sizeof(addr)) == -1) {
    perror("Bind error");
    close(sock);
    return EXIT_FAILURE;
}

char buf[BUFSIZE];
ssize_t readden = recvfrom(sock, buf, BUFSIZE, 0, &addr, sizeof(addr));
if (readden > 0) {
    printf("Client with IP %s sent datagram\n", inet_ntoa(addr.sin_addr));
    write (STDOUT_FILENO, buf, readden);
}
sendto(sock, "hello", 5, 0, &addr, sizeof(addr));
```

## Accepting connections on a blocking socket

A C program that wishes to accept network connections (act as a "server") should first create a socket bound to the address "INADDR_ANY" and call `listen` on it. Then, it can call `accept` on the server socket to block until a client connects.

```
//Create the server socket
int servsock = socket(AF_INET, SOCK_STREAM, 0);
if(servsock < 0) perror("Failed to create a socket");

int enable = 1;
setsockopt(servsock, SOL_SOCKET, SO_REUSEADDR, (char*)&enable, sizeof(int));

//Bind to "any" address with a specific port to listen on that port
int port = 12345;
sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);

if(bind(servsock, (sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    perror("Error binding to socket");

listen(servsock, 5);

//Accept a client
struct sockaddr_storage client_addr_info;
socklen_t len = sizeof client_addr_info;

int clientsock = accept(servsock, (struct sockaddr*)&client_addr_info, &len);

//Now you can call read, write, etc. on the client socket
```

The `sockaddr_storage` struct that gets passed to `accept` can be used to retrieve information about the client that connected. For example, here's how to determine the client's IP address:

```
char client_ip_str[INET6_ADDRSTRLEN + 1];
if(client_addr_info.ss_family == AF_INET) {
    // Client has an IPv4 address
    struct sockaddr_in *s = (struct sockaddr_in *)&client_addr_info;
    inet_ntop(AF_INET, &s->sin_addr, client_ip_str, sizeof(client_ip_str));
} else {  // AF_INET6
    // Client has an IPv6 address
    struct sockaddr_in6 *s = (struct sockaddr_in6 *)&client_addr_info;
    inet_ntop(AF_INET6, &s->sin6_addr, client_ip_str, sizeof(client_ip_str));
}
```

## Connecting to a remote host

POSIX.1-2008

Given the name of a server as a string, `char* servername`, and a port number, `int port`, the following code creates and opens a socket connected to that server. The "name" of the server can either be a DNS name, such as "www.stackoverflow.com," or an IP address in standard notation, such as "192.30.253.113"; either input format is valid for `gethostbyname` (which had been removed from POSIX.1-2008).

```
char * server = "www.example.com";

int sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock < 0)
  perror("Failed to create a socket");

hostent *server = gethostbyname(servername);
if (server == NULL)
  perror("Host lookup failed");

char server_ip_str[server->h_length];
inet_ntop(AF_INET, server->h_addr, server_ip_str, server->h_length);

sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(port);
memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length);

if (connect(sock, (sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    perror("Failed to connect");

// Now you can call read, write, etc. on the socket.

close(sock);
```

## Reading and writing on a blocking socket

Even when sockets are in "blocking" mode, the `read` and `write` operations on them do not necessarily read and write all the data available to be read or written. In order to write an entire

buffer into a socket, or read a known quantity of data from a socket, they must be called in a loop.

```
/*
 * Writes all bytes from buffer into sock. Returns true on success, false on failure.
 */
bool write_to_socket(int sock, const char* buffer, size_t size) {
    size_t total_bytes = 0;
    while(total_bytes < size) {
        ssize_t bytes_written = write(sock, buffer + total_bytes, size - total_bytes);
        if(bytes_written >= 0) {
            total_bytes += bytes_written;
        } else if(bytes_written == -1 && errno != EINTR) {
            return false;
        }
    }
    return true;
}
```

```
/*
 * Reads size bytes from sock into buffer. Returns true on success; false if
 * the socket returns EOF before size bytes can be read, or if there is an
 * error while reading.
 */
bool read_from_socket(int sock, char* buffer, size_t size) {
    size_t total_bytes = 0;
    while(total_bytes < size) {
        ssize_t new_bytes = read(sock, buffer + total_bytes, size - total_bytes);
        if(new_bytes > 0) {
            total_bytes += new_bytes;
        } else if(new_bytes == 0 || (new_bytes == -1 && errno != EINTR)) {
            return false;
        }
    }
    return true;
}
```

Read Sockets online: https://riptutorial.com/posix/topic/4706/sockets

# Chapter 9: Threads

## Examples

### Simple Thread without Arguments

This basic example counts at different rates on two threads that we name sync (main) and async (new thread). The main thread counts to 15 at 1Hz (1s) while the second counts to 10 at 0.5Hz (2s). Because the main thread finishes earlier, we use `pthread_join` to make it wait async to finish.

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

/* This is the function that will run in the new thread. */
void * async_counter(void * pv_unused) {
    int j = 0;
    while (j < 10) {
        printf("async_counter: %d\n", j);
        sleep(2);
        j++;
    }

    return NULL;
}

int main(void) {
    pthread_t async_counter_t;
    int i;
    /* Create the new thread with the default flags and without passing
     * any data to the function. */
    if (0 != (errno = pthread_create(&async_counter_t, NULL, async_counter, NULL))) {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    i = 0;
    while (i < 15) {
        printf("sync_counter: %d\n", i);
        sleep(1);
        i++;
    }

    printf("Waiting for async counter to finish ...\n");

    if (0 != (errno = pthread_join(async_counter_t, NULL))) {
        perror("pthread_join() failed");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

*Copied from here: http://stackoverflow.com/documentation/c/3873/posix-threads/13405/simple-thread-without-arguments where it had initially been created by M. Rubio-Roy.*

## Simple Mutex Usage

POSIX thread library provides implementation of the mutex primitive, used for the mutual exclusion. Mutex is created using `pthread_mutex_init`, and destroyed using `pthread_mutex_destroy`. Obtaining a mutex can be done using `pthread_mutex_lock` or `pthread_mutex_trylock`, (depending if the timeout is desired) and releasing a mutex is done via `pthread_mutex_unlock`.

A simple example using a mutex to serialize access to critical section follows. First, the example without using a mutex. Note that this program has *data race* due to unsynchronized access to `global_resource` by the two threads. As a result, this program has undefined behaviour:

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

// Global resource accessible to all threads
int global_resource;

// Threading routine which increments the resource 10 times and prints
// it after every increment
void* thread_inc (void* arg)
{
    for (int i = 0; i < 10; i++)
    {
        global_resource++;
        printf("Increment: %d\n", global_resource);
        // Make this thread slower, so the other one
        // can do more work
        sleep(1);
    }

    printf("Thread inc finished.\n");

    return NULL;
}

// Threading routine which decrements the resource 10 times and prints
// it after every decrement
void* thread_dec (void* arg)
{
    for (int i = 0; i < 10; i++)
    {
        global_resource--;
        printf("Decrement: %d\n", global_resource);
    }

    printf("Thread dec finished.\n");

    return NULL;
}

int main (int argc, char** argv)
{
```

```
    pthread_t threads[2];

    if (0 != (errno = pthread_create(&threads[0], NULL, thread_inc, NULL)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    if (0 != (errno = pthread_create(&threads[1], NULL, thread_dec, NULL)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++)
    {
        if (0 != (errno = pthread_join(threads[i], NULL))) {
            perror("pthread_join() failed");
            return EXIT_FAILURE;
        }
    }

    return EXIT_SUCCESS;
}
```

A possible output is:

```
Increment: 1
Decrement: 0
Decrement: -1
Decrement: -2
Decrement: -3
Decrement: -4
Decrement: -5
Decrement: -6
Decrement: -7
Decrement: -8
Decrement: -9
Thread dec finished.
Increment: -8
Increment: -7
Increment: -6
Increment: -5
Increment: -4
Increment: -3
Increment: -2
Increment: -1
Increment: 0
Thread inc finished.
```

Now, if we want to synchronise these threads so that we want first to increment or decrement all the way up or down, and then do it in the different way, we need to use a synchronization primitive, such as mutex:

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

```
#include <errno.h>
#include <stdlib.h>

// Global resource accessible to all threads
int global_resource;
// Mutex protecting the resource
pthread_mutex_t mutex;

// Threading routine which increments the resource 10 times and prints
// it after every increment
void* thread_inc (void* arg)
{
    // Pointer to mutex is passed as an argument
    pthread_mutex_t* mutex = arg;

    // Execute the following code without interrupts, all the way to the
    // point B
    if (0 != (errno = pthread_mutex_lock(mutex)))
    {
        perror("pthread_mutex_lock failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 10; i++)
    {
        global_resource++;
        printf("Increment: %d\n", global_resource);
        // Make this thread slower, so the other one
        // can do more work
        sleep(1);
    }

    printf("Thread inc finished.\n");

    // Point B:
    if (0 != (errno = pthread_mutex_unlock(mutex)))
    {
        perror("pthread_mutex_unlock failed");
        exit(EXIT_FAILURE);
    }

    return NULL;
}

// Threading routine which decrements the resource 10 times and prints
// it after every decrement
void* thread_dec (void* arg)
{
    // Pointer to mutex is passed as an argument
    pthread_mutex_t* mutex = arg;

    if (0 != (errno = pthread_mutex_lock(mutex)))
    {
        perror("pthread_mutex_lock failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 10; i++)
    {
        global_resource--;
        printf("Decrement: %d\n", global_resource);
```

```
    }

    printf("Thread dec finished.\n");

    // Point B:
    if (0 != (errno = pthread_mutex_unlock(mutex)))
    {
        perror("pthread_mutex_unlock failed");
        exit(EXIT_FAILURE);
    }

    return NULL;
}

int main (int argc, char** argv)
{
    pthread_t threads[2];
    pthread_mutex_t mutex;

    // Create a mutex with the default parameters
    if (0 != (errno = pthread_mutex_init(&mutex, NULL)))
    {
        perror("pthread_mutex_init() failed");
        return EXIT_FAILURE;
    }

    if (0 != (errno = pthread_create(&threads[0], NULL, thread_inc, &mutex)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    if (0 != (errno = pthread_create(&threads[1], NULL, thread_dec, &mutex)))
    {
        perror("pthread_create() failed");
        return EXIT_FAILURE;
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++)
    {
        if (0 != (errno = pthread_join(threads[i], NULL))) {
            perror("pthread_join() failed");
            return EXIT_FAILURE;
        }
    }

    // Both threads are guaranteed to be finished here, so we can safely
    // destroy the mutex
    if (0 != (errno = pthread_mutex_destroy(&mutex)))
    {
        perror("pthread_mutex_destroy() failed");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

One of the possible outputs is

```
Increment: 1
Increment: 2
Increment: 3
Increment: 4
Increment: 5
Increment: 6
Increment: 7
Increment: 8
Increment: 9
Increment: 10
Thread inc finished.
Decrement: 9
Decrement: 8
Decrement: 7
Decrement: 6
Decrement: 5
Decrement: 4
Decrement: 3
Decrement: 2
Decrement: 1
Decrement: 0
Thread dec finished.
```

The other possible output would be inverse, in case that `thread_dec` obtained the mutex first.

Read Threads online: https://riptutorial.com/posix/topic/4508/threads

# Chapter 10: Timers

## Examples

### POSIX Timer with SIGEV_THREAD notification

This example demonstrates POSIX Timer usage with `CLOCK_REALTIME` clock and `SIGEV_THREAD` notification method.

```
#include <stdio.h>  /* for puts() */
#include <string.h> /* for memset() */
#include <unistd.h> /* for sleep() */
#include <stdlib.h> /* for EXIT_SUCCESS */

#include <signal.h> /* for `struct sigevent` and SIGEV_THREAD */
#include <time.h>   /* for timer_create(), `struct itimerspec`,
                     * timer_t and CLOCK_REALTIME
                     */

void thread_handler(union sigval sv) {
        char *s = sv.sival_ptr;

        /* Will print "5 seconds elapsed." */
        puts(s);
}

int main(void) {
        char info[] = "5 seconds elapsed.";
        timer_t timerid;
        struct sigevent sev;
        struct itimerspec trigger;

        /* Set all `sev` and `trigger` memory to 0 */
        memset(&sev, 0, sizeof(struct sigevent));
        memset(&trigger, 0, sizeof(struct itimerspec));

        /*
         * Set the notification method as SIGEV_THREAD:
         *
         * Upon timer expiration, `sigev_notify_function` (thread_handler()),
         * will be invoked as if it were the start function of a new thread.
         *
         */
        sev.sigev_notify = SIGEV_THREAD;
        sev.sigev_notify_function = &thread_handler;
        sev.sigev_value.sival_ptr = &info;

        /* Create the timer. In this example, CLOCK_REALTIME is used as the
         * clock, meaning that we're using a system-wide real-time clock for
         * this timer.
         */
        timer_create(CLOCK_REALTIME, &sev, &timerid);

        /* Timer expiration will occur withing 5 seconds after being armed
         * by timer_settime().
         */
```

```
        trigger.it_value.tv_sec = 5;

        /* Arm the timer. No flags are set and no old_value will be retrieved.
         */
        timer_settime(timerid, 0, &trigger, NULL);

        /* Wait 10 seconds under the main thread. In 5 seconds (when the
         * timer expires), a message will be printed to the standard output
         * by the newly created notification thread.
         */
        sleep(10);

        /* Delete (destroy) the timer */
        timer_delete(timerid);

        return EXIT_SUCCESS;
}
```

Read Timers online: https://riptutorial.com/posix/topic/4644/timers

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with POSIX | alk, Community, gavv, Jonathan Leffler, kdhp, P.P., pah |
| 2 | File locks | gavv |
| 3 | Filesystem | gavv, yanpas |
| 4 | Input/Output multiplexing | yanpas |
| 5 | Pipes | John Bollinger |
| 6 | Processes | alk, gavv, P.P., pah |
| 7 | Signals | alk, gavv, Nemanja Boric, P.P., Toby |
| 8 | Sockets | alk, gavv, Giorgio Gambino, kdhp, Nemanja Boric, RamenChef, Toby, yanpas |
| 9 | Threads | alk, Nemanja Boric, P.P., Toby |
| 10 | Timers | gavv, pah |