An Introduction to POSIX

These notes aim at giving a brief introduction to the Portable Operating System Interface (POSIX). Most of the concepts described in this notes are taken from *Advanced Programming in the UNIX Environment* (3rd Ed., W. Stevens and S. Rago, Addison-Wesley, 2013). POSIX is a family of standards specified by the IEEE for maintaining compatibility between UNIX operating systems. UNIX refers to a family of operating systems. In general, you can think of the operating system as the software that controls the hardware resources of the computer providing the environment under which programs can run. At its core, the operating system includes the *kernel* whose main role is to interact with the hardware to provide the necessary abstractions for processes, file systems, memory, etc. The kernel offers its functionalities to applications via a software layer called *system calls.* On UNIX systems, each system call has a corresponding C function that is provided as part of the standard C library (so you don't need to link a separate library). When writing an application that uses POSIX functions, one only needs to include the specific header files that declare those functions. The main objective of this document is to provide an introduction to some of the system calls supported by the POSIX standard.

I/O Interface

This section describes the functions available to perform I/O operations. Before describing the I/O interface, it is essential to introduce the concept of a *file descriptor*. A file descriptor is a non-negative number used to identify a file. In practice, when a process opens or creates a new file, the kernel will return a file descriptor to the process. Then, the process can use the same file descriptor to identify the file from which it wants to read or write.

You can create or open a file with the `open` function, declared in the `fcntl.h` header as follows:

```
int open(const char *path, int oflag, ... /* mode_t mode */);
```

The return value is the file descriptor for the newly opened/created file. On error, the function returns -1 (recall that file descriptors are *non-negative* numbers, so a negative number can be used to notify the caller about an error). The *path* argument is the name of the file. The *oflags* argument defines the opening options. Each option is identified by a constant declared in the `fcntl.h` header. For instance, `O_RDONLY` defines read-only access, `O_WRONLY` means write-only, and `O_RDWR` opens the file for both reading and writing. Finally, the *mode* argument allows to define the file permissions when the file is opened with the `O_CREAT` option. For both the *oflags* and *mode* arguments, different options can be combined by passing their bitwise-or values (see examples below).

As usual, a program that acquires a resource must also properly release it. For files, this means *closing* them. An open file can be closed with the `close` function, declared in the `unistd.h` header as follows:

```
int close(int fd);
```

The return value is 0 on success, and -1 on error. The *fd* argument is the file descriptor.

We can read and write data from and to an open file using the `read` and `write` functions, respectively. The two functions are declared in `unistd.h` as follows:

```
ssize_t read(int fd, void *buf, size_t nbytes);
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Both functions return an `ssize_t` value to indicate the number of bytes that were actually transferred (read or written). In case of error, `read` and `write` return -1. For both `read` and `write`, the *fd* argument identifies the input or output file. The use of the *buf* and *nbytes* arguments differ slightly for the two functions. The `read` function uses *buf* as the destination for the bytes read, and *nbytes* to indicate the maximum amount of bytes to read and copy into *buf* from the input file. Conversely, `write` uses *buf* as a source for the data to be written, and *nbytes* to indicate the number of bytes to be written from *buf* onto the output file. Notice that both read and write operations may transfer less than *nbytes* bytes, in which case, the return value of `read` and `write` will be less than *nbytes*. For example, a read operation may return less than *nbytes* simply because the input has reached the end-of-file condition. However, in general, there is no guarantee that read or write operations would transfer the requested amount of data even if that amount of data is actually available. It is therefore the responsibility of the programmer to keep track of the progress made in reading and writing.

Using the basic functions seen so far, namely `open`, `close`, `read`, and `write`, we can already start implementing some small but fully functional tools. For example, we could implement a file-copy utility similar to the `cp` command:

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>                  /* for open. */
#include <unistd.h>                 /* for read, write, close. */
#include <errno.h>                  /* for interrupt while writing, and

int main(int argc, const char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "usage: %s <src> <dst>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Open the source file for reading. */
    int srcfd = open(argv[1], O_RDONLY);
    if (srcfd < 0) {
        fprintf(stderr, "failed to open source file: %s\n", argv[1]
        return EXIT_FAILURE;
    }

    /* Open the destination file for writing and create it if it it
     * does not exist.  The `mode' sets read and write permissions
     * only for the file owner.  (A better implementation would als
     * copy the permissions from the source file, which can be read
     * using the `stat' function.
     */
    int dstfd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR
    if (dstfd < 0) {
        fprintf(stderr, "failed to open destination file: %s\n", ar
        close(srcfd);
        return EXIT_FAILURE;
    }

    while (1) {
        char buf[1024];
        int n = read(srcfd, buf, 1024);
        if (n < 0) {
            /* The read call might be interrupted by a signal -- ye
             * the world is complicated -- and in this case, the
             * result is considered an error condition reported
             * through a global variable called `errno' and a const
             * value EINTR.
             */
            if (errno == EINTR)
                continue;
            fprintf(stderr, "failed to read from source file\n");
            goto error_handling;
        } else if (n == 0)      /* end of file */
            break;

        /* Copy the bytes read from the source file into the destin
        size_t l = 0;           /* number of bytes written onto the
        while (l < n) {
            /* We must keep track of our progress in writing the n
             * bytes we read this time around.
             */
            ssize_t wres = write(dstfd, buf + l, n - l);
            if (wres < 0) {
                /* The write call might also be interrupted by a si
                if (errno == EINTR)
                    continue;
                fprintf(stderr, "failed to copy to destination file
                goto error_handling;
            }
            l += wres;
        }
    }

    /* Once done, close both the source and destination files. */
    close(srcfd);
    close(dstfd);
    return EXIT_SUCCESS;

 error_handling:
    close(srcfd);
    close(dstfd);
    return EXIT_FAILURE;
}
```

I/O multiplexing using *select*

By default, `read` and `write` operations are *blocking*, meaning that the a `read` call returns only when data becomes available, and `write` blocks until the stream is available for writing. This is okay if you need to read or write from a single file or stream. But what if you need to work with multiple streams? For example, imagine implementing a server that needs to handle multiple client connections at the same time. The server could go round-robin over the client connections, read from each one, and process the client requests as needed.

However, this way, an inactive client would block the server while there might be complete requests from other clients.

One solution to this problem is to use I/O multiplexing. The idea of I/O multiplexing is that the server would “select” one or more file descriptors for which data is indeed available, and then read from (or write to) those files without blocking. This can be done with the `select` function declared in `sys/select.h` as follows:

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

The `select` function takes three *sets* of file descriptors, *readfds* for reading, *writefds* for writing, and *exceptfds* for error conditions, plus a *timeout* interval. The `select` call blocks until either data is available from one or more files in *readfds*, data can be written into one or more files in *readfds*, an error condition arises in one or more files in *exceptfds*, or the *timeout* period elapses.

The return value is the total number of file descriptors ready for I/O across the provided sets. Upon returning, `select` also changes the file-descriptor sets to let the caller know exactly which file descriptors are ready for their corresponding operations. On error, the function returns -1. In case of a timeout, which means that no file descriptor is ready for I/O, the function returns 0.

The *nfds* argument specifies the range of file descriptors to monitor. In practice, it is the maximum file descriptor (value) plus one. The *readfds*, *writefds*, and *exceptfds* arguments are pointers to sets of file descriptors to monitor for reading, writing, and exceptional conditions, respectively. These sets are manipulated using the `FD_SET`, `FD_CLR`, `FD_ISSET`, and `FD_ZERO` macros. (The example below illustrates the use of the file descriptors sets and their macros.) The *timeout* argument is a pointer to a structure defining the maximum time to wait for an event. If this parameter is `NULL`, `select` does not use a timeout and therefore may block indefinitely. The `struct timeval` is defined in the `sys/time.h` header as follows:

```
struct timeval {
    long tv_sec;    /* Seconds */
    long tv_usec;   /* Microseconds */
};
```

Using the `select` function, we can solve the problem of a server supporting multiple concurrent connections as follows:

```
#include <stdint.h>
/* Include this header for error handling */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
/* Include headers to use networking functions. */
#include <sys/socket.h>
#include <arpa/inet.h>
/* Include this header to use close, read, write. */
#include <unistd.h>
/* Include this header to use the select function. */
#include <sys/select.h>

#define BACKLOG 128
```

```c
#define BUFSIZE 1024

/* Defining a helper function to log an error and quit. */
void err_quit(const char *msg)
{
    if (errno != 0)
        perror(msg);
    else
        fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}


/* Defining a helper function to setup a TCP listening socket. If y
 * are curious about what this code does, look up the UNIX socket A
 * but you will be exposed to these concepts during your computer
 * networking classes.
 */
int server_socket(uint16_t port)
{
    struct sockaddr_in addr;
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        err_quit("Failed to create a socket");

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr *) &addr, sizeof(addr)) < 0)
        close(sockfd);
        err_quit("Failed to bind the socket");
    }

    if (listen(sockfd, BACKLOG) < 0) {
        close(sockfd);
        err_quit("Failed to make the socket listening");
    }

    return sockfd;
}


/* An helper function to close a list of file descriptors. */
void close_all_fds(int fds[], size_t count)
{
    for (size_t j = 0; j < count; ++j)
        if (fds[j] >= 0)
            close(fds[j]);
}


/* An helper function to write n bytes into a file. */
ssize_t writen(int fd, const char *data, size_t n)
{
    size_t nleft = n;
    const char *p = data;
    ssize_t nwritten;
    while (nleft > 0) {
        if ((nwritten = write(fd, p, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        p += nwritten;
    }
    return n;
}


int main(int argc, const char *argv[])
{
    int sockfd = server_socket(8080);
    ssize_t n;

    /* Define the list of file descriptors we want to monitor,
     * corresponding to the client connections.  We implement the l
     * with an array.  FD_SETSIZE is a constant defining the maximu
     * number of file descriptors that can be store in a fd_set.  W
     * use that one as the maximum number of concurrent clients.
     */
    int fds[FD_SETSIZE];
    unsigned fds_count = 0;
    /* Define two fd_set: one will be passed to the select call,
     * and while the other will keep track of the file descriptors
     * we should monitor.
     */
    fd_set rset, allset;

    int maxi = -1;
    /* The maxfd variable will keep track of the highest file
     * descriptor seen so far.
     */
    int maxfd = sockfd;
    char buf[BUFSIZE];

    /* Initialize the fd_set structure. */
    FD_ZERO(&allset);
    /* Using the FD_SET to register the server file descriptor. */
    FD_SET(sockfd, &allset);

    while (1) {
        /* fd_set is a struct, so we can copy it. Select does not
         * preserve the provided fd_set since it has to modify it
         * to indicate which file desciptors are ready for I/O.
         * Hence, we define two fd_set and we copy allset into rset
         * in each iteration.
         */
        rset = allset;

        /* Use the select function to wait until some file in the
         * set is ready for I/O. Notice the NULL for the timeout
         * argument indicates that we will wait until some event
         * happens.
         */
        int nready = select(maxfd + 1, &rset,
                            NULL, NULL, NULL);
        if (nready == -1) {
            close_all_fds(fds, fds_count);
            err_quit("Select failed");
        }

        /* If we reach this point, it means that some file descript
         * in the rset is ready for I/O. We start by checking if th
         * is any new connection on the server socket. We can check
         * the file descriptor is ready for I/O by using the FD_ISS
         * macro.
         */
        if (FD_ISSET(sockfd, &rset)) {
            if (fds_count == FD_SETSIZE) {
                close_all_fds(fds, fds_count);
                err_quit("Too many files open");
            }
            int client = accept(sockfd, NULL, NULL);
            fds[fds_count++] = client;

            /* We use the FD_SET macro to register the new client
             * into the set.
             */
            FD_SET(client, &allset);
            shutdown(client, SHUT_WR);
            if (client > maxfd)
                maxfd = client;
            if (--nready <= 0)
                continue;
        }

        // Read data from the connected clients
        for (unsigned int i = 0; i < fds_count; ++i) {
            /* If the file descriptor is ready for I/O handle it. *
            if (FD_ISSET(fds[i], &rset)) {
            reading:
                // Simply read data from the client and copy it to
                if ((n = read(fds[i], buf, BUFSIZE)) < 0) {
                    if (errno == ECONNRESET) {
                        close(fds[i]);
                        /* The client closed the connection, so rem
                         * file descriptor from the fd_set
                         */
                        FD_CLR(fds[i], &allset);
                        fds[i] = fds[--fds_count]; /* remove fds[i]
                    } else if (errno == EINTR) {
                        goto reading;
                    } else {
                        close_all_fds(fds, fds_count);
                        err_quit("Failed reading from client");
                    }
                } else if (n == 0) {
                    close(fds[i]);
                    /* We use the FD_CLR macro to delete the discon
                     * client from the set.
```

```
                         */
                        FD_CLR(fds[i], &allset);
                        fds[i] = fds[--fds_count]; /* remove fds[i] by
                    } else if (writen(STDOUT_FILENO, buf, n) < 0) {
                        close_all_fds(fds, fds_count);
                        err_quit("Failed to write client data to stdout
                    }
                    if (--nready <= 0)
                        break;
                }
            }
        }
    }
}
```

After compiling and running the application, you can test it using the following commands:

```
$ nc localhost 8080
```

I/O multiplexing using *poll*

The `poll` function is conceptually similar to `select`, and therefore can also be used to check for the immediate availability of input/output data from multiple streams. The declaration, which can be included from `poll.h`, is as follows:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

The *fds* parameter points to an array of `struct pollfd` objects; *nfds* is the length of the array. Each `struct pollfd` object specifies a file descriptor together with the I/O events of interest for it. Specifically, `struct pollfd` is defined as follows:

```
struct pollfd {
    int fd;                 /* File descriptor to monitor */
    short int events;       /* Events we monitor (POLLIN, POLLERR, etc.)
    short int revents;      /* Occurred events (POLLIN, POLLERR, etc.)
};
```

The `poll` functions also takes a *timeout* parameter that specifies the maximum time to wait for an event in milliseconds. If *timeout* is 0, `poll` returns immediately. If `timeout` is -1, poll blocks indefinitely until an event occurs.

The return value is the total number of file descriptors ready for I/O in the provided array. Specifically, `poll` sets the value of `revents` for each `struct pollfd` object to indicate which file is ready for I/O or shows an error condition. On error, `poll` returns -1. In case of a timeout with no file descriptor ready for I/O, the function returns 0.

The following example demonstrate the use of `poll`:

```
#include <stdint.h>
/* Include this header for error handling */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
/* Include headers to use networking functions. */
```

```
#include <sys/poll.h>
#include <sys/socket.h>
#include <arpa/inet.h>
/* Include this header to use close, read, write. */
#include <unistd.h>
/* Include this header to use the poll function. */
#include <poll.h>

#define BACKLOG 128
#define BUFSIZE 1024

/* Defining a helper function to log an error and quit. */
void err_quit(const char *msg)
{
    if (errno != 0)
        perror(msg);
    else
        fprintf(stderr, "%s\n", msg);
    exit(EXIT_FAILURE);
}


/* Defining a helper function to setup a TCP listening socket.
 * If you are curious about what this code does, look up the
 * UNIX socket API, but you will be exposed to these concepts
 * during your computer networking classes.
 */
int server_socket(uint16_t port)
{
    struct sockaddr_in addr;
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        err_quit("Failed to create a socket");

    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr *) &addr, sizeof(addr)) < 0)
        close(sockfd);
        err_quit("Failed to bind the socket");
    }

    if (listen(sockfd, BACKLOG) < 0) {
        close(sockfd);
        err_quit("Failed to make the socket listening");
    }

    return sockfd;
}


/* An helper function to close a list of file descriptors. */
void close_all_fds(struct pollfd fds[], size_t count)
{
    for (size_t j = 0; j < count; ++j)
        if (fds[j].fd >= 0)
            close(fds[j].fd);
```

```
}


/* An helper function to write n bytes into a file. */
ssize_t writen(int fd, const char *data, size_t n)
{
    size_t nleft = n;
    const char *p = data;
    ssize_t nwritten;

    while (nleft > 0) {
        if ((nwritten = write(fd, p, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        p += nwritten;
    }
    return n;
}


int main(int argc, const char *argv[])
{
    int sockfd = server_socket(8080);
    ssize_t n;

    /* Define the list of file descriptors we want to monitor.
     * FOPEN_MAX is a constant defining the maximum number of
     * open file descriptors.
     */
    struct pollfd fds[FOPEN_MAX];

    /* Register the server socket as the first item in the
     * list to be notified about new connecting clients.
     */
    fds[0].fd = sockfd;
    fds[0].events = POLLIN;

    /* We have 1 file decriptor, i.e. the server socker. */
    unsigned fds_count = 1;
    char buf[BUFSIZE];

    while (1) {
        /* Use the poll function to wait until some
         * file in the set is ready for I/O. Notice the
         * -1 argument indicates that we will wait until
         * some event happens.
         */
        int nready = poll(fds, fds_count, -1);
        if (nready == -1) {
            close_all_fds(fds, fds_count);
            err_quit("Polling failed");
        }

        /* If we reach this point, it means that some file
```

```
             * descriptor in the fds list is ready for I/O.
             * We start by checking if there is any new connection
             * on the server socket. We can check if the file descripto
             * is ready for I/O by looking at revents.
             */
            if (fds[0].revents & POLLIN) {
                if (fds_count == FOPEN_MAX) {
                    close_all_fds(fds, fds_count);
                    err_quit("Too many files open");
                }

                int client = accept(sockfd, NULL, NULL);
                fds[fds_count].fd = client;
                fds[fds_count++].events = POLLIN;
                shutdown(client, SHUT_WR);
                if (--nready <= 0)
                    continue;
            }

            // Read data from the connected clients
            for (unsigned i = 1; i < fds_count; ++i) {
                /* If the file descriptor is ready for I/O handle it. *
                if (fds[i].revents & (POLLIN | POLLERR)) {
                    reading:
                    // Simply read data from the client and discard it.
                    if ((n = read(fds[i].fd, buf, BUFSIZE)) < 0) {
                        if (errno == ECONNRESET) {
                            close(fds[i].fd);
                            fds[i] = fds[--fds_count];
                        } else if (errno == EINTR) {
                            goto reading;
                        } else {
                            close_all_fds(fds, fds_count);
                            err_quit("Failed reading from client");
                        }
                    } else if (n == 0) {
                        close(fds[i].fd);
                        fds[i] = fds[--fds_count];
                    } else if (writen(STDOUT_FILENO, buf, n) < 0) {
                        close_all_fds(fds, fds_count);
                        err_quit("Failed to write client data to stdout
                    }
                    if (--nready <= 0)
                        break;
                }
            }


        }
    }
```

Notice that `poll` is semantically identical to `select`. However, the interface of `poll` might be more convenient, since it does not require a re-initialization of the list of file descriptors at each iteration.

Process Interface

We now introduce some of the functions that allow us to control processes. When you run a program on your computer, the operating system executes the code of the program within a *process.* You can think of a process as the execution of the program, and also as all the data that the operating system associates with that execution. This includes the list of open files, including network connections, the identity of the user who started the process, which will be used to check for permissions to access system resources, the memory map that implements the virtual memory of the process, and much more.

When you run a C program from your command shell, the shell effectively creates a new process to execute that program. This is not a just a prerogative of the shell. Any program, which is running in a process, can create other processes. The mechanism by which a process creates another process is called *fork.* Intuitively, this term alludes to a fork in a road that represents the execution of a process. The original process proceeds in a single sequential line of instructions. Then, the fork operation executed by the original process creates another process, after which the two processes proceed along two independent lines. You can also think of the creation of a process as a cloning operation, since the new process is created as an almost identical copy of the initial process. In this fork/cloning operation, the initial process is also called the *parent,* while the new process is called the *child.* Thus all the processes in a system can be arranged in a hierarchical structure (a tree) according to this parent–child relation.

Each process is identified by a unique process ID (PID), a non-negative number assigned to the process at its creation. Note that even though PIDs are unique at any given time, the same PID can be reused after its process terminates. Also, some processes are special. Usually, PID 0 is reserved for some kernel operations. PID 1 is the process that starts right after the “bootstrap” phase of the execution of the operating systems and that is responsible for starting the system processes. This initialization process typically runs a program called *init* or, on more modern systems, *launchd* or *systemd.* The initialization process is the root of the tree of processes in a system.

In a C program, you can create a new process with the `fork` function defined in the `unistd.h` header as follows:

```
pid_t fork(void);
```

The return type of this function is quite special, since it will return *twice,* meaning that it will return in the parent process, and also independently in the child process. In fact, the return *value* of the function is different in the parent and the child process. In the parent process, `fork` returns the PID of the child process, or -1 in case of error. In the child process, the return value is 0. After `fork` returns, both child and parent continue executing with the instructions that follow the `fork` call. In fact, as we said, the child process is essentially a clone of the parent process, in the sense that the child process executes exactly the same program as the parent process. The child process also inherits some resources from its parent, such as open files and environment variables. However, the execution of the child process is independent, so the child gets a copy of the parent’s data in memory.[1]

The *exec* family of functions.

A newly “forked” child process will continue executing the same program as the parent process. So, how does the shell manage to run any program other than a copy of itself? Simple: the shell—or any other process—can call a special *exec* function that effectively terminates the current program and continues the execution of the same process with a completely new program. So, when you write a command in your shell, the shell forks off a new process such that the parent continues with the shell code, while the child immediately calls *exec* to run the program indicated in the command.

The `exec` family of functions is defined in the `unistd.h` header file as follows:

```
int execl(const char *pathname, const char *arg, ...
          /*, (char *) NULL */);
int execle(const char *pathname, const char *arg, ...
             /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execve(const char *pathname, char *const _Nullable argv[],
           char *const _Nullable envp[]);

int execlp(const char *filename, const char *arg, ...
           /*, (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execvpe(const char *filename, char *const argv[], char *const e
```

Normally, *exec* functions do not return. This is because the semantics of the *exec* function is to immediately terminate the program and to replace it by another program that starts from scratch. However, the return value is still important in case of error. In that case, all *exec* functions return -1.

The various specific functions differ in the way the new execution is set up. These differences are indicated mnemonically in the suffix letters of the function names. Thus, `execl` takes the arguments to be passed to the new program as a *list* of string parameters passed individually as arguments to the `execl` call. Instead, `execv` takes the program arguments as a *vector* passed as a single argument: an array of `char *` terminated by a `NULL` pointer. Then, `execle` is like `execl` but it also sets the *environment* of the starting process, and so does `execve` as a variant of `execv`. Lastly, the `p` suffix in `execlp`, `execvp`, and `execvpe` indicates that, if the given *filename* does not contain a slash character (`/`), then that program name is looked up in the `PATH` environment variable.

Waiting for and collecting the exit status from a process

A process terminates when the program that is executed within that process terminates. This can be a normal termination, such as when the `main` function terminates or when the program calls the `exit` function. Or it can be an abnormal termination, for example when the program calls the `abort` function. In any such case, the execution of the process stops. However, at that point, the operating system does not immediately clear the process data. In this situation, when the process is effectively dead but the operating system has not yet disposed of its body, the process is said to be in *zombie* state.

The purpose of keeping a zombie process is to allow its parent process to collect the exist status of the terminated process. The parent process can do that with the `wait` and `waitpid` functions. These functions return immediately if the target process has already terminated, or otherwise block until the target process terminates. These *wait* functions are defined in the `sys/wait.h` header as follows:

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

The main difference between the two functions is that `wait` will block until one of the child processes terminate, whereas `waitpid` has a number of options that allow to control which process to wait for. Both functions return the PID of the child process that terminated. On error, the functions return -1. The *status* argument is a pointer to an integer where the exit status of the terminated child process is stored. The value in the status variable can be analyzed with the `WIFEXITED`, `WIFSIGNALED`, `WIFSTOPPED` macros. In the `waitpid` function, the *pid* argument allows to specify which child process to wait for. A positive PID waits for the child process with the specified PID. A pid value of -1 waits for any child process (it is essentially equivalent to `wait`). A pid value of 0 waits for any child process in the same process group as the caller. A pid value less than -1 waits for any child process whose process group ID is equal to -pid. Finally, the *options* argument allows to modify the behavior of the call. For instance, the `WNOHANG` causes `waitpid` not block, and instead return immediately even if no child process has terminated, in which case the return value will be 0.

The following code shows an example of how these functions interact to create a child process that simply uses `exec` to invoke the `echo` utility to print a message:

```
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
/* Include this header for the fork and exec functions. */
#include <unistd.h>
/* Include this header for the wait function. */
#include <sys/wait.h>

int main(void)
{
    pid_t pid;

    /* Create a child process. */
    pid = fork();
    switch (pid) {
    case -1:
        /* In case the child process creation failed
         * simply report the error.
         */
        fprintf(stderr, "Failed to fork process\n");
        return EXIT_FAILURE;

    case 0: {
        char *const envp[] = {"MYVAR=somevalue", NULL};

        /* The child process should run the echo command using the
         * exec call.
         */
        if (execle("/bin/sh", "sh", "-c", "echo Variable has value
            fprintf(stderr, "Failed to exec process\n");
            return EXIT_FAILURE;
        }
    }

    default: {
        int status;
        printf("Child has PID %jd\n", (intmax_t) pid);

        /* Wait for the child process to terminate. */
        if (wait(&status) == -1) {
            fprintf(stderr, "Wait failed\n");
            return EXIT_FAILURE;
        } else if (WIFEXITED(status)) {
            /* Use sys/wait.h macros to obtain child process exit s
            printf("Child exited with status %d\n", WEXITSTATUS(sta
        }
        return EXIT_SUCCESS;
    }}
}
```

IPC Interface

With Inter-Process Communication (IPC), we refer to the mechanisms provided by the operating system to exchange and share data among multiple processes. In this section, we provide an introduction to UNIX pipes. You can think of a pipe as a channel where whatever gets written at one end will be available for reading at the other end. So, when one process writes into a pipe and another process reads from the same pipe, the pipe is effectively providing a way for the two process to communicate.

Pipes have two important limitations. First, the channel they provide is half-duplex. That is, data flows in only one direction. Second, a pipe can only be used between processes that have a common ancestor. There are other IPC mechanisms that overcome these limitations. An example, which we do not cover in this document, is UNIX domain sockets.

You can create a pipe using the `pipe` function defined in `unistd.h` as follows:

```
int pipe(int fd[2]);
```

The return value is 0 on success, and -1 on error. The pipe is returned via the *fd* argument. Notice, that `pipe` takes an array of file descriptors of length 2. `fd[0]` is the reading end of the pipe, and `fd[1]` is the writing end of the pipe. In practice, whatever data you write in `fd[1]` will be available to `fd[0]` for reading.

The following code show an example about how to use a UNIX pipe to have the parent and child process sharing some data. In practice, the parent will generate a “random” number of “random” lines, and will send them over a pipe to the child process. The child process will read the line from the pipe, and it will output it to standard output.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>            /* for pipe, fork, write, etc. */
#include <sys/wait.h>          /* for the wait. */
#include <errno.h>             /* for interrupt while writing, and

#define MAXLINE 1024


/* An helper function to write n bytes into a file. */
ssize_t writen(int fd, const char *data, size_t n)
{
    size_t nleft = n;
    const char *p = data;
    ssize_t nwritten;
    while (nleft > 0) {
        if ((nwritten = write(fd, p, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;
            else
                return -1;
        }
        nleft -= nwritten;
        p += nwritten;
    }
    return n;
}


int main(void)
{
    int fd[2];
    ssize_t n;
    char line[MAXLINE];

    if (pipe(fd) < 0) {
        fprintf(stderr, "failed to create pipe\n");
        return EXIT_FAILURE;
    }

    pid_t pid = fork();
    switch(pid) {
    case -1:
        /* If the fork fails, simply close the pipe and report the
        close(fd[0]);
        close(fd[1]);
        fprintf(stderr, "Failed to fork process\n");
        return EXIT_FAILURE;

    case 0:
        /* The child does not write anyting in the pipe in this exa
         * so we can just close the write end of the pipe for the c
         */
        close(fd[1]);

    reading:
        /* Reading data sent by the parent from the pipe. . */
        while ((n = read(fd[0], line, MAXLINE)) > 0)
            if (writen(STDOUT_FILENO, line, n) < 0) {
                close(fd[0]);
                fprintf(stderr, "Failed to output parent messages\n
                return EXIT_FAILURE;
            }

        if (n < 0 && errno == EINTR) goto reading;

        /* Closing also the read side of the pipe once done */
        close(fd[0]);

        if (n < 0) {
            fprintf(stderr, "Failed to receive message from parent\
            return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;

    default: {
        /* The parent does not read anything from the child, so we
         * the read end of the pipe.
         */
        close(fd[0]);

        srand(time(NULL));
        int ret = EXIT_SUCCESS;
        int nlines = rand() % 20;
        for (int i = 0; i < nlines; ++i) {
            char line[MAXLINE];
            int linelen = rand()%(MAXLINE - 2);
            int j = 0;

            for (; j < linelen; ++j)
                line[j] = rand()%26 + 'a';

            line[j++] = '\n';

            /* Write line over the pipe for the child process to re
            if (writen(fd[1], line, j) < 0) {
                fprintf(stderr, "Failed to write to pipe\n");
                ret = EXIT_FAILURE;
                break;
            }
        }

        /* Closing the pipe once done */
        close(fd[1]);

        /* Wait for the child process to terminate. Not that we pas
         * as status because we do not care about the return status
         * for this demo.
         */
        if (wait(NULL) == -1) {
            fprintf(stderr, "Wait failed\n");
            ret = EXIT_FAILURE;
        }

        return ret;
    }}
}
```

Footnotes:

[1] Actually, it is not a real copy. Initially, there is no copy at all, and instead the memory regions of the parent process are simply shared with the child process. Then, if one of the processes executes a write operation (in memory), the corresponding page will be copied. This way of managing memory pages is called “copy-on-write”, and it is what makes the spawning and use of processes very efficient.