9
Files and IO

9.1 Low-level file IO

- Before data can be red from, or written to a file, it needs to be **opened**.
- When a file is opened, the kernel returns a **file descriptor** to the process, a non-negative integer.
- For the kernel, all open files are referred to by file descriptors, usually **small integers**, starting with 0.
- Whenever IO is to be done on the file, the file descriptor is used to identify the file.
- Each process has a fixed size **descriptor table**. Its size can be figured out with `getrlimit`(3), or `bash`(1) builtin `ulimit`.

High-level IO

- The functions in `stdio`(3) (such as `getchar`(3), `printf`(3), ...), provide a **high-level** interface (*streams*) to the IO **system calls**.
- Buffering is implemented in the streams, *i.e.*, not available at low-level.
- It usually is a **bad idea** to mix high-level and low-level access to the same files.

Tracing system calls

- The tool strace(1) runs a program, and traces the **system calls** issued.
- Each line in the trace lists a system call, its arguments, and its return value.

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

```
$ pk-cc main.c
$ ./a.out
Hello world!
$ strace -o log ./a.out                                  # see strace(1)
Hello world!
$ tail -n5 log                                           # see tail(1)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f1b93a20000
write(1, "Hello world!\n", 13)          = 13      # here the writing happens
exit_group(0)                           = ?
+++ exited with 0 +++
```

9.2 Open a file

A file is opened with the `open`(3) system call:

```
#include <fcntl.h>
int open(const char *path, int flags, mode_t mode);
```

- `path` is the name of the file to open or create.
- `flags` specifies how to open the file. It must be exactly one of:

  `O_RDONLY` Open for reading only.
  `O_WRONLY` Open for writing only.
  `O_RDWR` Open for reading and writing.

  And may be or-ed with **further flags**: (*cf.* `open`(3) for full list)

  `O_CREAT` If the file does not exist it will be created.
  `O_EXCL` Ensure that this call creates the file.
  `O_APPEND` Writing to the file will always append to it.
  `O_TRUNC` Truncate the file length to 0 if possible.

- If the file is created, the permissions from `mode` are used.
- Returns a file descriptor on success, or `-1` on failure, and `errno` is set to indicate the error (*cf.* `errno`(3) and `errno.h`).

errno?

- The C library defines an external variable `errno`, modified by library functions to report errors back to the calling process.
- `#include <errno.h>` brings `errno` into scope for analysis.
- The function `strerror`(3) decodes the error code for you.
- There is a family of functions like `err`(3)[35], or `perror`(3) which print error messages or warnings using `errno`'s value.

Example Open a file with error handling.

```
const char *path = "/tmp/testfile";
int fd = open(path, O_RDONLY, 0);

if (fd < 0)
    err(1, "Opening %s failed", path);
```

```
$ ./a.out
a.out: Opening /tmp/testfile failed: No such file or directory
```

[35] **Note:** Non-standard BSD extension.

Close a file

A file is closed with the `close`(3) system call:

```
#include <unistd.h>
int close(int fd);
```

- `fd` is the **file descriptor** previously returned by `open`.
- `close` returns 0 on success, or `-1` on failure, and `errno` is set to indicate the error.
- When a process **terminates**, all associated file descriptors are closed.

Why bother?

- Proper hygiene.
- Releases any locks the process may have on the file.
- You may run out of available file descriptors otherwise.

Reading and writing files

Reading and writing is done with the read(3) and write(3) system calls.

```
#include <sys/types.h>                                  /* cf. sys_types.h(0) */
#include <unistd.h>

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

- fd is a **file descriptor** previously returned by open.
- buf points to a **buffer** where the data should be stored, or taken from.
- count is the number of bytes **to be transferred**.
- Returns...
  - ...−1 and sets errno on failure. That's why ssize_t is used.
  - ...the number of bytes **actually** transferred. This may be less than count for valid reasons, *e.g.*, EOF while reading.
- The **position** in the file is advanced by the number of transferred bytes.

(*cf.* page 232)

Example: Copy a file

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>

enum { BUFSIZE = 1024, PERMS = 0666 };

int main(int argc, char *argv[])
{
    char buf[BUFSIZE];  /* buffer fot copied data */
    ssize_t c;          /* count bytes */
    int src, tgt;       /* file descriptors */

    if (argc < 3) errx(1, "Need source and target");

    printf("Copying %s to %s\n", argv[1], argv[2]);

    src = open(argv[1], O_RDONLY, 0);
    if (src < 0) err(1, "Cannot read %s", argv[1]);

    tgt = open(argv[2], O_WRONLY|O_CREAT|O_EXCL, PERMS);
    if (tgt < 0) err(1, "Cannot write %s", argv[2]);
```

```
    while ((c = read(src, buf, BUFSIZE)) > 0) {
        if (write(tgt, buf, (size_t)c) < c)
            err(1, "Write failed");
    }

    close(src);
    close(tgt);

    return 0;
}
```

Shortcomings of this program:

- Only copies one file. → Easy: Loop over arguments.
- Cannot copy into a directory.
- Ignorant if first argument is a directory.
  - How can we read a directory? *cf.* `readdir`(3)
- Permissions are not copied, only the `umask`(3) is used.

The standard IO streams

- When the shell runs a program, three files are open already:
  - File descriptor `0` → reads *stdin*.
  - File descriptor `1` → writes *stdout*.
  - File descriptor `2` → writes *stderr*.
- If a program reads `0` and writes `1` and `2`, it can do input and output without worrying about opening files.
- POSIX.1 replaces the magic numbers `0`, `1`, and `2` with `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO` (`unistd.h`)

IO Redirection

- At the shell, the user can **redirect** IO to and from files with < and >:

```
$ ./a.out <data 2>log
```

- In this case, the shell changes the default assignments for file descriptors 0 and 2 to the named files. File descriptor 1 is still attached to the terminal.
- Similar observations hold for IO associated with a pipe.
- In all cases, the file assignments are **set up by the shell**, not by the program. The program does not known where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

Exercise There is no magic here!

- Write a program that writes to a file descriptor, say 23, **without** opening it beforehand.
- Implement error handling, and convince yourself of its functioning.
- Run the program with redirection `23>test`. Look at the generated file.

Buffered implementation of getchar

- A simple **unbuffered** implementation of getchar could be as follows. This requires one **system call** per character to read.

```
int getchar(void)
{
    char c;
    return (read(STDIN_FILENO, &c, 1) == 1) ? c : EOF;
}
```

- Better read **chunks of data** into memory:

```
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static ssize_t n = 0;

    if (n == 0) {      /* buffer is empty */
        n = read(STDIN_FILENO, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? *bufp++ : EOF;
}
```

Reposition read/write file offset

- Every open file has a current **file offset**.
- Indicates **position** of the next `read`/`write` operation, in bytes from the beginning of the file.
- By default the offset is initialized to `0` when a file is opened.

Seeking Change the offest.

```
#include <sys/types.h>                                    /* cf. sys_types.h(0) */
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

- The offset can be adjusted with `lseek`(3) in **seekable** files.
- `whence` indicates how to measure the change:
  - `SEEK_SET` Set to `offset` bytes from the beginning of file.
  - `SEEK_CUR` Set to its current value plus `offset`, which may be negative.
  - `SEEK_END` Set to the size of the file plus `offset`, which may be negative.
- A successful call to `lseek` returns the new file offset. Otherwise, `-1` is returned, `errno` is set, and the offset is not changed.

Seekable files?

- Some files cannot be seeked, *e.g.*, the standard streams *stdin*, *stdout*, and *stderr*.
- We can test this by seeking with offset 0.

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

int main(void)
{
    off_t pos = lseek(STDIN_FILENO, 0, SEEK_CUR);
    if (pos == -1)
        err(1, "Cannot seek (fd=%d)", STDIN_FILENO);
    printf("File position is %d (fd=%d).\n", (int)pos, STDIN_FILENO);
    return 0;
}
```

```
$ ./a.out
a.out: Cannot seek (fd=0): Illegal seek
$ ./a.out <seek.c
File position is 0 (fd=0)
```

Files with holes

If a file is seekable, you may seek **beyond the end** of it.

```
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int main(void)
{
    const char * msg = "hello world";
    int fd = open("/tmp/testfile", O_WRONLY|O_CREAT|O_EXCL, 0666);
    write(fd, msg, 11);
    lseek(fd, 100000, SEEK_SET);  /* advance to 100k bytes */
    write(fd, msg, 11);
    close(fd);
    return 0;
}
```

```
$ ./a.out && ls -l -s /tmp/testfile            # print the allocated size of each file
8.2k -rw------- 1 marcel users 101k Dec 13 16:05 /tmp/testfile # depends on FS
```

- The file's size is 101k, but it consumes only 8.2k diskspace.
- Reading the hole will deliver 0-bytes. (try `hexdump -C /tmp/testfile`)

9.3 Properties of a file

- When interacting with filesystem it is often relevant to determine information *about* a file.
- **Metadata** instead of the file contents, *e.g.*:
  - Is it a directory?
  - Permissions to read/write/execute?
  - Ownership?
  - Time of last modificaton?
  - Size?
- An example is the `ls`(1) program, which prints metadata of files.

The `stat` family of functions

```c
#include <sys/types.h>
#include <sys/stat.h>

struct stat { /* next slide */ };

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Given a `path`, the `stat`(3) function returns a structure of information about the named file.
- `fstat`(3) works with a **file descriptor**, instead of a path.
- `lstat`(3) returns information about the **symbolic link**, not the referenced file.
- Each of them modifies the passed `stat` structure, and return `0` on success, or `-1` on error with `errno` set.

The `stat` structure

```
struct stat {
    dev_t     st_dev;              /* ID of device containing file */
    ino_t     st_ino;              /* inode number */
    mode_t    st_mode;             /* protection */
    nlink_t   st_nlink;            /* number of hard links */
    uid_t     st_uid;              /* user ID of owner */
    gid_t     st_gid;              /* group ID of owner */
    dev_t     st_rdev;             /* device ID (if special file) */
    off_t     st_size;             /* total size, in bytes */
    blksize_t st_blksize;          /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;           /* number of 512B blocks allocated */
    struct timespec st_atim;       /* time of last access */
    struct timespec st_mtim;       /* time of last modification */
    struct timespec st_ctim;       /* time of last status change */
};
```

- The `stat` structure is described in `stat`(2).
- This is where `ls -l` gets its information from.
- Mostly primitive system data types, described in `sys_types.h`(0).

File types

```
struct stat {
    mode_t    st_mode;    /* inode's mode */
    /* ... */
};
```

- The **permission bits** are stored in the `st_mode` member of `stat`,
- as is the **type** of the file (from the OS's perspective), *e.g.*,

Regular file	Text and binary data.
Directory file	Maintains directory data.
Block special file	Typically disk devices.
Socket	Network communication.
Symbolic link	Pointer to another file.
...	

- Typically, `st_mode` is **tested using the macros** described in `stat`(3).

```
struct stat buf;
stat(path, &buf);  /* there should be error handling here */
if (S_ISREG(buf.st_mode) && (buf.st_mode & S_IWUSR)) {
    /* path is a regular file, writable by the owner */
}
```

Example: Copy a file

An extension of the copy program (*cf.* page 227)

```
    struct stat st_buf;   /* for metadata */
/* ... */

    if (fstat(src, &st_buf))
        err(1, "Cannot stat %s", argv[1]);

    if (!S_ISREG(st_buf.st_mode))
        errx(1, "%s is not a regular file", argv[1]);

    tgt = open(argv[2], O_WRONLY|O_CREAT|O_EXCL,
               st_buf.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO)));

    if (tgt < 0)
        err(1, "Cannot write %s", argv[2]);
/* ... */
```

- Passing other than the permission bits to `open` is not specified.
- The final mode of the created file is still **subject to** `umask`(3) modification.