Change the mode of a file

- The mode of a file can be changed with `chmod`(3).
- Again, the file may be addressed via its `path`, or a file descriptor `fd`.

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

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Example Extending the copy program:

```
/* ... */
tgt = open(argv[2], O_WRONLY|O_CREAT|O_EXCL, 0); /* restrict until copied */
/* [... copy the file ...] */
if (fchmod(tgt, st_buf.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO)) != 0)
    err(1, "Cannot change mode of %s", argv[2]);
/* ... */
```

- It is not a good idea to set the permissions to 0. This is just to demonstrate that we are still allowed to write!

The sticky-bit

- The **sticky-bit** (*aka.* **restricted deletion** flag) `S_ISVTX`, usually is 01000 in `st_mode`. (So it's the bit just before the permissions).
- If set for a directory, a file therein can be **removed** or **renamed** *only* if the user has `wx` permission for the directory, **and** at least
  - owns the file, or
  - owns the directory, or
  - is privileged.

  (Normally, `wx` permission on the containing directory is sufficient)
- Typical usage: Temporary storage; you cannot delete files of other users:

```
$ ls -ld /tmp
drwxrwxrwt 11 root root 300 Dec 17 20:48 /tmp
```

- The sticky-bit has no meaning if set on a file.

History Why the name?

- In the old days, an executable with the sticky-bit would stick in memory, even if unused, so that it could be launched faster.
- This use is obsoleted by current memory management policies.

Functions Related to `st_mode`, `st_uid`, `st_gid`

- `access`(3) - check access permissions of a file or pathname
- `umask`(3) - set file creation mode mask
- `chmod`(3), `fchmod`(3) - change mode of file
- `chown`(3), `fchown`(3), `lchown`(3) - change owner/group of a file

Note The `stat`(1) command line utility is a powerful tool to inspect all kinds of file properties.

Fine-grained permissions may be set on many Unix-like systems using `chacl`(1) or `setfacl`(1), if you really need it. More Information in `acl`(5).

9.4 Excursus: Feature Test Macros

- Code using `lstat`(2), `fchmod`(2), *etc.*, may confront you with:

```
$ pk-cc cp_v4.c
cp_v4.c: In function ‘main’:
cp_v4.c:40:2: warning: implicit declaration of function ‘fchmod’
```

  - This indicates that `fchmod` is not declared (*i.e.*, is unknown).
  - This happens **although** you have `#include`d all relevant header files.

- For different **platforms** or language **standards**, the C Standard Library is expected to provide different sets of functions!
  - The set of exposed functions is controlled with the **Feature Test Macros** facility, *cf.* `feature_test_macros`(7).
  - With `-std=c99`, header file `sys/stat.h` does **not expose** `fchmod`.
  - Compare, *e.g.*, the manual pages `open`(2) and `open`(3).

- To use `fchmod`, you must **confirm** that you know your code uses an extension.

- The manual pages list **Feature Test Macro Requirements**, if any. *E.g.*, `fchmod`(2) states:

```
Feature Test Macro Requirements for glibc (see feature_test_macros(7)):
    fchmod():
        _BSD_SOURCE || _XOPEN_SOURCE >= 500 ||
        _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED
        || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
```

- The programmer **needs to choose** which extension to use. In `feature_test_macros`(7) you'll find an explanation of the options.

- For the copy program, we might select `_POSIX_C_SOURCE`:

```
#define _POSIX_C_SOURCE  200809L    /* put this before any includes! */
#include <sys/stat.h>               /* now, this exposes fchmod */

/* ... */
if (fchmod(tgt, st_buf.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO)) != 0)
```

cf. http://www.gnu.org/software/libc/manual/html_node/Feature-Test-Macros.html

10
Processes

10.1 The persona of a process

How is access permission determined?

- There are **users** and **groups** on a Unix system
  - Users are identified by a **user ID** (UID),
  - groups are identified by a **group ID** (GID).
- Every user is member of his **default group**,
- and maybe further **supplementary groups**.

- We have seen that each file has permission bits for its owner (`stat.st_uid`), members of its group (`stat.st_gid`), and the rest of the world.
- How exactly do the permissions apply when I want to access a file?

Note Users do not open files! It's processes that call `open`(2).

Persona

- Every running process has (among others)
  - an **effective user ID** (EUID), *cf.* `geteuid`(2)
  - an **effective group ID** (EGID), and *cf.* `getegid`(2)
  - a set of **supplementary group IDs**. *cf.* `getgroups`(2)

  These form the **persona** of the process.
- A process with EUID 0 is called **privileged**.

File access permission test A simplified version.

- A **privileged process** may
  - read/write/traverse any file/directory, and
  - run any program if at least one of its `x`-bits is set.
- For **non-privileged** processes, exactly one of the following is chosen:
  - If the file's `st_uid` equals the process's EUID, verify against the **owner-permissions** of the file.
  - Otherwise, if the file's `st_gid` equals the process's EGID, or one of its supplementary GIDs, verify against the **group-permissions** of the file.
  - Otherwise, verify against the **other-permissions** of the file.

How does a process get its persona?

- From the **user** that launches it.
  - But wait: Users don't launch processes. Other processes do.
- More precise: Copied from the **process** that launches it.
  - How can a process "run by a normal user" modify, say, the **password database**? See the `passwd`(1) tool.
  - If the first process belongs to `root` (UID=0), then there can be only privileged processes?
- Obviously, there is a problem!
  - A process needs to **change its persona**!

Changing persona

- In addition to EUID and EGID, there are
  - the **real user ID** (UID), and *cf.* `getuid`(2)
  - the **real group ID** (GID), *cf.* `getgid`(2)
  - the **file user ID**, *i.e.*, the UID of the program file, and
  - the **file group ID**, *i.e.*, the GID of the program file.
- When a process is launched,
  - the **real IDs** are copied **from the caller**.
  - If the **set-user-ID** bit `S_ISUID` (04000) is set on the program file, the file UID is used as the process' EUID, and
  - analogous for the **set-group-ID** bit `S_ISGID` (02000).
- Functions `setuid`(2) and `setgid`(2) **change the persona**.
  - A **privileged** process can change it's persona at will.
  - An **unprivileged** process can only change its IDs if the corresponding set-ID-bit is set on the program file. Its **only options** are the real ID, or the file ID.

Example

The passwd tool changes the password of a user.

- Every user needs permission to run it.
- passwd needs permission to modify the password database (/etc/shadow).
- Unprivileged users must not gain access to the password database.

```
$ ls -l /etc/shadow
-rw------- 1 root root 631 Nov 12 23:35 /etc/shadow
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 48k Oct 21 16:33 /usr/bin/passwd
$ passwd
Changing password for marcel.
(current) UNIX password: # waiting for input
```

In a different terminal, while passwd is waiting for input:

```
$ ps -C passwd -o pid,user,ruser,comm                # cf. ps(1)
  PID USER     RUSER    COMMAND
 2180 root     marcel   passwd
```

Dropping privileges

Logging in

- The `login` program runs as privileged process.
- It must access the password database to verify a correct login.
- Then it starts the user's login shell.
- By changing its persona, `login` **drops** its privileges.

Web server

- A web server typically listens on **port 80** (according to the standard).
- However, only privileged processes may bind to ports below 1024.
- Running a privileged server process is **dangerous**!
  - If the server process is hacked, it may execute malicious code.
  - This code would be privileged on the target system!
- Thus, a server should **drop** its privileges after binding to port 80.

10.2 Process creation

- Every process has a unique **process ID**, a non-negative integer.
- Process IDs are **reused at a later time**, *i.e.*, the PID identifies a process only at a given time, not over the entire system uptime..
- Process ID 1 is usually the **init** process.
  - It is invoked by the kernel at the end of the bootstrap procedure.
  - Except for PID 1, every process has a **parent**.

```
#include <unistd.h>

pid_t getpid(void);  /* process ID of calling process, cf. getpid(2) */
pid_t getppid(void);  /* parent process ID of calling process, cf. getppid(2) */
```

Forking a new process

```
#include <unistd.h>
pid_t fork(void);
```

- The function `fork`(2) creates a **new process**.
- The new process is called the **child process**, the process that called `fork` is called the **parent process**.
- The child **is a copy** of the parent.
  - However, parent and child **share the text** segment (*i.e.*, program code).
  - Current implementations perform **copy-on-write** (COW) on the data segment, so this is reasonably efficient.
- Return value:
  - On success, `fork` **returns twice**: `0` is returned in the *child* process, and the **process ID** of the child is returned in the *parent* process.
  - On failure, `-1` is returned to the caller, no child is created, and `errno` is set.

Example

```c
/* ... includes ... */
int global = 6;  /* global variable */

int main(void)
{
    int var = 88;  /* automatic variable on the stack */

    printf("before fork: pid=%d, glob=%d, var=%d\n", getpid(), global, var);

    pid_t pid = fork();
    if (pid < 0)                /* error handling */
        err(1, "fork");
    else if (pid == 0) {        /* only in child */
        global++;
        var++;
    } else {                    /* only in parent */
        printf("child pid is: %d\n", pid);
        sleep(2);
    }

    printf("after fork: pid=%d, glob=%d, var=%d\n", getpid(), global, var);
    return 0;
}
```

Running the example gives:

```
$ ./a.out
before fork: pid=19024, glob=6, var=88          # this is the parent
child pid is: 19025
after fork: pid=19025, glob=7, var=89           # this is the child
after fork: pid=19024, glob=6, var=88           # this is the parent
```

Notes

- The modifications performed in the child process are **invisible** to the parent!
- Obviously, both processes can write to *stdout*. They seem to **share** the same file descriptor:

```
$ ./a.out >foo                  # then look at the generated file
```

File descriptors

- Parent and child share the same file descriptors.
  - This is a **good** thing. We'll see use cases later.
- They also share the same file **offset** (important).
  - You can actually **observe** this with the `seek` functions.
- Some **synchronization** is needed. Very common:
  - The parent `fflush`(3)es all unwritten data before `fork`(2)ing.
  - The parent wait's for child to complete (*cf.* later).
  - The parent and child close descriptors they do not need, and access disjoint sets of descriptors only.

When to use fork

- A process can duplicate itself to execute different code at the same time, *e.g.*, web severs.
- Process wants to execute a different program, *e.g.*, a shell.

Waiting for a child

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

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

- The functions `wait`(2), and `waitpid`(2) **wait for a child** process to change status.
- `wait` **blocks**, until **any child** terminates.
  - Returns **PID** of child that has terminated, or
  - -1 on error, setting `errno` (*e.g.*, if there was no child).
- `waitpid` is **more general**, it can be configured:
  - Which child(ren) to wait for,
  - which status change to observe (there are others than termination),
  - whether to block or not,
  - ...
- If `status` is not `NULL`, the child's status change is described there.
  - This can be examined with the macros described in `wait`(2).

```c
int main(int argc, char **argv)
{
    pid_t pid = fork();
    if (pid < 0) err(1, "fork");

    if (pid == 0) {                                  /* child */
        printf("Sleeping.\n");
        sleep(2);
        return argc > 1 ? atoi(argv[1]) : 0;

    } else {                                         /* parent */
        int status;

        printf("Waiting for child %d.\n", pid);
        if (wait(&status) < 0) err(1, "wait");
        if (WIFEXITED(status))
            printf("Exit status was %d.\n",
                WEXITSTATUS(status));

    }

    return 0;
}
```

```
$ ./a.out 42
Waiting for child 5232.
Sleeping.
Exit status was 42.
```

About orphans and zombies

- If a process terminates, the system stores its **exit status** for collection with `wait` by the parent.
- If the parent **dies before the child** does, the child becomes **reparented** to the `init` process (*i.e.*, the one with PID 1).
  - These processes are called **orphans**.
  - The init process automatically collects the exit status for any child that terminates.
- If the parent process is not yet `wait`ing for the child, the system **must not discard** the exit status — maybe it is requested later.
  - Thus, the **dead child** still consumes one process table entry!
  - These processes are called **zombies**.
  - **Long running** processes (web server, login shell) may accumulate an army of zombies if not cleaned up properly.
  - Only when the **parent dies**, the zombie is reparented to `init`, and subsequently reaped.

```c
int main(int argc, char **argv)          /* observe reparenting with top(1) */
{
    if (argc < 3 || argc > 4)
        errx(1, "Invalid number of arguments.");

    pid_t p = fork();
    if (p < 0) err(1, "fork");

    const char *who = p ? "parent" : "child";
    int s = atoi(argv[p ? 1 : 2]);

    printf("%s[%d]: sleep %d seconds\n", who, getpid(), s);
    sleep((unsigned int)s);

    if (p && argc > 3) {  /* only with three args */
        printf("%s[%d]: waiting\n", who, getpid());
        wait(NULL);
        s = atoi(argv[3]);
        printf("%s[%d]: waited, sleep %d seconds\n", who, getpid(), s);
        sleep((unsigned int)s);
    }

    printf("%s[%d]: exiting\n", who, getpid());
    return 0;
}
```

Fight zombies with a double `fork`

```
pid_t pid = fork();

if (pid < 0)
    err(1, "fork");

else if (pid > 0) {
    waitpid(pid, &status, 0);

    /* do parent stuff */

} else {
    pid = fork();
    if (pid > 0) exit(EXIT_SUCCESS);
    if (pid < 0) err(1, "fork");

    /* do child stuff */

}
```

- The **double fork** is a common means to omit zombies.
- Useful if the parent process does **not care about** the child's exit status.
- Instead, the parent waits for an **intermediate** child, which dies after forking the real child.
- The real child becomes reparented to `init`.

Note Another means to fight zombies will be shown in the chapter on signals, *cf.* page 279.