The basics of file systems

The POSIX file system semantics

Mount points

To make a FS visible, one ”mounts” it below a directory visible to a user.

For the OS kernel, a mount point is a mark that says, “when descending into this directory, jump to the root directory of a file system mounted here”.

```
$ ls -lh ~/testing/mount/
total 0
```

```
$ mount -t ext4 ./img ~/testing/mount/

$ ls -lh ~/testing/mount/
total 8.0K
drwxrwxr-x 1 1002 1002 4.0K Sep 25 16:59 pstorage-fes
drwxr-xr-x 1 1002 1002   83 Sep  6 21:11 rpmbuild
```

Mount points

To make a FS visible, one ”mounts” it below a directory visible to a user.

For the OS kernel, a mount point is a mark that says, “when descending into this directory, jump to the root directory of a file system mounted here”.

```
$ ls -lh ~/testing/mount/
total 0
```

```
$ mount -t ext4 ./img ~/testing/mount/

$ ls -lh ~/testing/mount/
total 8.0K
drwxrwxr-x 1 1002 1002 4.0K Sep 25 16:59 pstorage-fes
drwxr-xr-x 1 1002 1002   83 Sep  6 21:11 rpmbuild
```

The list of mount points can be viewed this way:
- `$ cat /proc/self/mounts`

File systems can be mounted on first access: https://linux.die.net/man/5/auto.master

The POSIX file system semantics

File system objects and their names are separate things

We've seen that files and their names are independent from each other: `link()` and `unlink()` can create files with multiple names and files without names.

A working directory is also not tied to a path. Instead, it is a pointer "directory X in file system Y":

```
$ pwd
/home/artem/testing/students


$ ls -lh .
total 40K
-rw-r--r-- 1 artem artem  234 Sep 28 11:48 example
-rw-r--r-- 1 artem artem  11K Sep 27 21:49 proc
-rw-r--r-- 1 artem artem 1.1K Sep 27 21:49 proc.c
-rw-r--r-- 1 artem artem  13K Sep 28 20:14 ps
-rw-r--r-- 1 artem artem 1.6K Sep 28 20:13 ps.c
```

```
$ pwd
/home/artem/testing/students

$ mount -t ext4 ./img ~/testing/students/

$ ls -lh .

                ???

$ ls -lh ~/testing/students/

                ???
```

The POSIX file system semantics

File system objects and their names are separate things

We’ve seen that files and their names are independent from each other: `link()` and `unlink()` can create files with multiple names and files without names.

A working directory is also not tied to a path. Instead, it is a pointer “directory X in file system Y”:

```
$ pwd
/home/artem/testing/students



$ ls -lh .
total 40K
-rw-r--r-- 1 artem artem  234 Sep 28 11:48 example
-rw-r--r-- 1 artem artem  11K Sep 27 21:49 proc
-rw-r--r-- 1 artem artem 1.1K Sep 27 21:49 proc.c
-rw-r--r-- 1 artem artem  13K Sep 28 20:14 ps
-rw-r--r-- 1 artem artem 1.6K Sep 28 20:13 ps.c
```

```
$ pwd
/home/artem/testing/students

$ mount -t ext4 ./img ~/testing/students/

$ ls -lh .
total 40K
-rw-r--r-- 1 artem artem  234 Sep 28 11:48 example
-rw-r--r-- 1 artem artem  11K Sep 27 21:49 proc
-rw-r--r-- 1 artem artem 1.1K Sep 27 21:49 proc.c
-rw-r--r-- 1 artem artem  13K Sep 28 20:14 ps
-rw-r--r-- 1 artem artem 1.6K Sep 28 20:13 ps.c

$ ls -lh ~/testing/students/
total 8.0K
drwxrwxr-x 1 1002 1002 4.0K Sep 25 16:59 pstorage-fes
drwxr-xr-x 1 1002 1002   83 Sep  6 21:11 rpmbuild
```

The POSIX file system semantics

Virtual file systems in Linux

For a user of the POSIX API a file system does not need to represent data located on a physical device. Any implementation will do as long as it is possible to

- find a file or a directory by name,
- list the content of directories,
- read and write the content of files.

Virtual file systems in Linux: procfs

Linux has a file system where top-level directories represent processes, and files in a directory describe properties of the process.

```
$ ls -lh /proc/self/
total 0

-r--r--r-- 1 artem artem 0 Oct  2 10:08 cmdline
lrwxrwxrwx 1 artem artem 0 Oct  2 10:08 cwd -> /home/artem
lrwxrwxrwx 1 artem artem 0 Oct  2 10:08 exe -> /bin/ls
dr-x------ 2 artem artem 0 Oct  2 10:08 fd
-r--r--r-- 1 artem artem 0 Oct  2 10:08 maps
-r--r--r-- 1 artem artem 0 Oct  2 10:08 stat
......
```

To do at home:

- read `man 5 proc`,
- what does `/proc/PID/auxv` contain, and how does `execve()` fill the stack of a new process?
- write a program that hides its first command line argument from `/proc/PID/cmdline` (a hint: `man prctl` and look for `PR_SET_MM`),
- implement
  - `ps`
  - `lsof`

Virtual file systems in Linux: FUSE

Normally, file systems are implemented in the kernel.

FUSE (Filesystem in USer spacE) is a mechanism to run file system drivers as userspace processes.

FUSE file systems drivers open a pipe to the kernel-space FUSE layer. Over that pipe, they receive commands like “lookup a file in a directory”, “open a file”, “read/write data to a file”, etc.

Example: sshfs.

Advantages of FUSE:
- Userspace processes may use any libraries that may not be fit to the kernel,
- File system drivers can be run by non-privileged users,
- FUSE enables easy experimentation with FS implementations.

Disadvantages of FUSE:
- (Much) lower performance due to numerous switches between the kernel and the userspace.

To do at home:
- Read the documentation about the FUSE high level API,
- Implement a FUSE file system that has only the root directory and one file named “hello”. Reading this file must return the string “hello, world!”. Verify that `ls -l` and `cat hello` work with your file system.

The POSIX file system semantics

POSIX file system API

UNIX is a multiuser OS and needs to isolate files of Alice from access by Bob, unless Alice allows that.

The POSIX file system semantics

POSIX file system API

UNIX is a multiuser OS and needs to isolate files of Alice from access by Bob, unless Alice allows that.

The classical access model:

- the system tracks the set of users and the set of user groups,
- files have one owner user and one owner group,
- files have permission bits that tell what access is granted to the owner user, to the owner group, and to everyone else.

The POSIX file system semantics

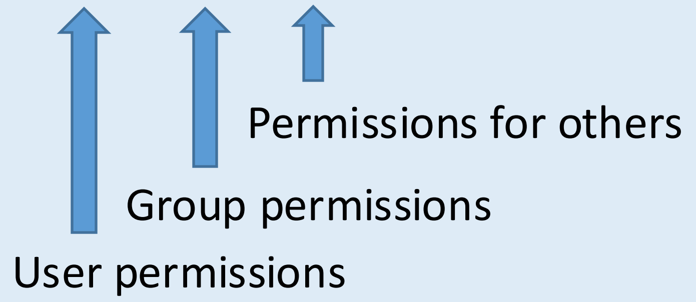
POSIX file system API

UNIX is a multiuser OS and needs to isolate files of Alice from access by Bob, unless Alice allows that.

The classical access model:

- the system tracks the set of users and the set of user groups,
- files have one owner user and one owner group,
- files have permission bits that tell what access is granted to the owner user, to the owner group, and to everyone else.

```
 $ ls -lh /usr/bin
 ...
  rwx   r-x   --x 1 root   root     31K Feb 22  2016 afm2pl
  rwx   r-x   --x 1 root   root     38K Feb 22  2016 afm2tfm
  rwx   r-x   --x 1 root   root    485K Feb 22  2016 aleph
```



POSIX file system API

UNIX is a multiuser OS and needs to isolate files of Alice from access by Bob, unless Alice allows that.

The classical access model:
- the system tracks the set of users and the set of user groups,
- files have one owner user and one owner group,
- files have permission bits that tell what access is granted to the owner user, to the owner group, and to everyone else.

This access model is woefully inadequate for complicated setups. It has no way to grant access to a file both to Gregory and George unless they belong to the owner group of the file.

There is a much more flexible mechanism for assigning access rights to a file, POSIX Access Control Lists. See
- man 5 acl,
- man 1 getfacl,
- man 1 setfacl.

The POSIX file system semantics

POSIX file system API

There are special “access rights” that modify the way the programs are exec()ed:

```
$ ls -lh /usr/bin
...
 rws   r-x   r-x 1 root   root    134K Jan  6  2016 sudo
 rwx   rwx   rwx 1 root   root       4 Jan  6  2016 sudoedit -> sudo
 rwx   r-x   r-x 1 root   root     47K Jan  6  2016 sudoreplay
```

The POSIX file system semantics

POSIX file system API

There are special “access rights” that modify the way the programs are exec()ed:

```
$ ls -lh /usr/bin
...
 rws   r-x   r-x 1 root   root    134K Jan  6  2016 sudo
 rwx   rwx   rwx 1 root   root       4 Jan  6  2016 sudoedit -> sudo
 rwx   r-x   r-x 1 root   root     47K Jan  6  2016 sudoreplay
```

When a set-uid binary is `exec()`ed, it runs with the credentials of the owner user. The similar thing happens with set-gid binaries.

The POSIX file system semantics

POSIX file system API

Files and file descriptors are two separate entities. In particular, they may have unrelated access rights:

```
int fd = open(“/path/to/a/file”, O_RDWR | O_CREAT, S_IRUSR);
write(fd, buffer, size);
close(fd);
```

The POSIX file system semantics

POSIX file system API

Files and file descriptors are two separate entities. In particular, they may have unrelated access rights:

```
int fd = open("/path/to/a/file", O_RDWR | O_CREAT, S_IRUSR);
write(fd, buffer, size);
close(fd);
```

Use cases:

- Split programs into a small trusted and audited core, and into worker processes that handle complex data. What is an example of a very frequently used software that follows this model?
- Binfmt handlers for files that allow only --x--x--x: https://lwn.net/Articles/679310/
- See also: seccomp and seccomp filters: https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

When multiple threads access the same memory region, they need to synchronise their access to avoid races.

What happens when multiple process access the same directory?

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

When multiple threads access the same memory region, they need to synchronise their access to avoid races.

What happens when multiple process access the same directory?

1. Creating a temporary file:

```
int fd = open(“/tmp/tmp.1b42ac00de”, O_RDWR|O_CREAT, 0);
unlink(“/tmp/tmp.1b42ac00de”);
... use fd to keep temp data ...
```

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

When multiple threads access the same memory region, they need to synchronise their access to avoid races.

What happens when multiple process access the same directory?

1. Creating a temporary file:
```
int fd = open(“/tmp/tmp.1b42ac00de”, O_RDWR|O_CREAT, 0);
unlink(“/tmp/tmp.1b42ac00de”);
... use fd to keep temp data ...
```

2. Reading a symbolic link:
```
struct stat st;
lstat(“/path/to/symlink”, &st);
char *buf = malloc(st.st_size + 1);
readlink(“/path/to/symlink”, buf, st.st_size);
buf[st.st_size] = ‘\0’;
```

A file system is a shared resource: concurrent access and races

When multiple threads access the same memory region, they need to synchronise their access to avoid races.

What happens when multiple process access the same directory?

1. Creating a temporary file:

```
int fd = open(“/tmp/tmp.1b42ac00de”, O_RDWR|O_CREAT, 0);
unlink(“/tmp/tmp.1b42ac00de”);
... use fd to keep temp data ...
```

2. Reading a symbolic link:

```
struct stat st;
lstat(“/path/to/symlink”, &st);
char *buf = malloc(st.st_size + 1);
readlink(“/path/to/symlink”, buf, st.st_size);
buf[st.st_size] = ‘\0’;
```

3. Creating two files in a directory:

```
int fd0 = open(“/dir/a”, O_RDWR|O_CREAT, S_IRUSR);
int fd1 = open(“/dir/b”, O_RDWR|O_CREAT, S_IRUSR);
```

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

When multiple threads access the same memory region, they need to synchronise their access to avoid races.

What happens when multiple process access the same directory?

1. Creating a temporary file:
```
int fd = open(“/tmp/tmp.1b42ac00de”, O_RDWR|O_CREAT, 0);
unlink(“/tmp/tmp.1b42ac00de”);
... use fd to keep temp data ...
```

2. Reading a symbolic link:
```
struct stat st;
lstat(“/path/to/symlink”, &st);
char *buf = malloc(st.st_size + 1);
readlink(“/path/to/symlink”, buf, st.st_size);
buf[st.st_size] = ‘\0’;
```

3. Creating two files in a directory:
```
int fd0 = open(“/dir/a”, O_RDWR|O_CREAT, S_IRUSR);
int fd1 = open(“/dir/b”, O_RDWR|O_CREAT, S_IRUSR);
```

1. There is a time frame when another process can open `/tmp/tmp.1b42ac00de` by name. Such process will be able steal the data from the temp file.

2. Another process can change the symlink value between the calls to `lstat()` and `readlink()`. See TOCTTOU (time of check to time of use).

3. In the time frame between two `open()`s another process can rename `/dir` and then create a different directory with the same name.

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

One of the solutions: run applications in containers:

- If processes A and B each have their ”private” `/tmp` directory, then they cannot access files of each other by construction.

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

One of the solutions: run applications in containers:

- If processes A and B each have their ”private” `/tmp` directory, then they cannot access files of each other by construction.
- Linux has “**filesystem namespaces**” which are collections of mount points. Mount points of a FS namespace are visible only to processes that run in that specific FS namespace.
- In two different FS namespaces the path `/tmp` may refer to two different tmpfs instances.

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

One of the solutions: run applications in containers:

- If processes A and B each have their ”private” `/tmp` directory, then they cannot access files of each other by construction.
- Linux has “**filesystem namespaces**” which are collections of mount points. Mount points of a FS namespace are visible only to processes that run in that specific FS namespace.
- In two different FS namespaces the path `/tmp` may refer to two different tmpfs instances.

See also:

- Pid namespaces,
- Network namespaces,
- User namespaces.

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

One of the solutions: run applications in containers:

- If processes A and B each have their ”private” `/tmp` directory, then they cannot access files of each other by construction.
- Linux has “**filesystem namespaces**” which are collections of mount points. Mount points of a FS namespace are visible only to processes that run in that specific FS namespace.
- In two different FS namespaces the path `/tmp` may refer to two different tmpfs instances.

How to populate a FS namespace?

- One can start with an empty directory that will become the root of the FS namespace,
- Add read-only images of directories with standard binaries and libraries like `/usr` and `/lib`,
- Add instances of `/tmp` and `/proc` private to the container,
- Add directories with the data of an application that runs in the container.

The POSIX file system semantics

A file system is a shared resource: concurrent access and races

One of the solutions: run applications in containers:

- If processes A and B each have their ”private” `/tmp` directory, then they cannot access files of each other by construction.
- Linux has “**filesystem namespaces**” which are collections of mount points. Mount points of a FS namespace are visible only to processes that run in that specific FS namespace.
- In two different FS namespaces the path `/tmp` may refer to two different tmpfs instances.

How to populate a FS namespace?

- One can start with an empty directory that will become the root of the FS namespace,
- Add read-only images of directories with standard binaries and libraries like `/usr` and `/lib`,
  - see also: bind mounts
- Add instances of `/tmp` and `/proc` private to the container,
- Add directories with the data of an application that runs in the container.
  - see also: Kubernetes persistent volumes

Bind mounts

Bind mounts are a Linux-specific extension to mount points. When a path traversal descends into a mount point M, it jumps to the root of a file system mounted at M. When descending into a bind mount point M', a path traversal jumps into an arbitrary directory that is the destination of a bind mount.

```
$ ls -lh src/
total 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 1
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 2

$ ls -lh dst/
total 0

$ mount --bind src/ dst/

$ ls -lh dst/
total 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 0
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 1
-rw-r--r-- 1 artem artem 0 Oct  2 00:29 2
```

Bind mounts add a huge number of edge cases:

- one can bind-mount files,
- http://lwn.net/Articles/689856/