OPERATING SYSTEM
TD 4: File systems

Global instructions

Sources for part 1 are available on the web page: `http://adrien.krahenbuhl.fr/en/teaching/`

1 Virtual File System with FUSE

You must install the FUSE development packages to obtain the `/usr/include/fuse.h` file, for example the `libfuse-dev` package under Debian/Ubuntu. Make sure your user is in the `fuse` group or have access to `/dev/fuse`.

FUSE (Filesystem in USErspace) allows to implement a file system without writing a kernel module. An existing driver, the fuse kernel module, route the user requests made through the VFS to a process executing our code. The system calls are translated into calls at functions that you will define. This allows you to implement exotic file systems, such as the reading of a tar.gz file without decompressing it.

The VFS Hello

(1.1) Download the sources (see global instructions) and compile them. Mount the virtual file system Hello in `./mnt` by executing `./hello ./mnt`. The file system logs are sent to the `hello.log` file in the current folder. Display them as we go along in another terminal with `tail -f hello.log`.

(1.2) List the content of the mounted folder, display the contents of the file. Explain what is happening and the contents of the logs.

(1.3) For the debug, you can attach `gdb` after mounting with `gdb -p $ (pidof hello)`. Later, you will unmount the file system with `fusermount -u ./mnt`.

(1.4) Add a second virtual file with a different content.

The VFS grepfs

In this section you will use FUSE to create a file system that applies the `grep` command transparently. You will work on a single folder whose the sub-folders are ignored. If the `toto` file exists in this folder, it will appear:

- ✓ without modification in the file system,
- ✓ in `foo/toto` as a file whose content is restricted to lines containing `foo`,
- ✓ in `foo/bar/toto` with only the lines containing `foo` and `bar`.

(1.5) Mount the file system in `./mnt` with

```
./grepfs . ./mnt
```

and display the logs in another terminal with `tail -f ./grepfs.log`. Explain the output of the three following commands:

```
ls -l ./mnt
ls ./mnt/foo/bar
cat ./mnt/foo/bar/grepfs.c
```

(1.6) Complete the `grep_realpath()` and `grep_getattr()` implementation to retrieve the true attributes of the real files. If it fail, look at the folder. You will use `LOG()` if necessary to check that `stat()` is called on the right files. Explain the output of the following commands:

```
ls ./mnt/foo/bar
cat ./mnt/foo/bar/grepfs.c
```

(1.7) In `grep_read()`, parse the path to discover the tags at the beginning. For example, extract `foo` from `./mnt/foo/grepfs.c` or `foo` and `bar` from `./mnt/foo/bar/grepfs.c`. Construct the corresponding command to execute with `popen()`.

(1.8) Improve `readdir()` to only display the files, not the subfolders. Then, only display the files whose content matches the path. For example `ls ./mnt/foo/` should only list files containing `foo`.

2 Size and disk occupation

The results observed in the questions below may vary depending on the type of used file system (ext4, ext3, NFS, . . .). Open `/proc/mounts` and find the different mount points to find out what file system you are using. To avoid writing a lot of programs in C, you can use the `dd` tool to fill pieces of file. For example, to copy 1 block (`count`) of size 1 (`bs`) from the file `/dev/zero` (`if`) to the file `file` (`of`) by skipping 1 000 000 blocks (`seek`), you can use the following command:

```
dd if=/dev/zero of=file count=1 bs=1 seek=1000000
```

(2.1) Create a file containing a single character. Observe its size with the commands `du -h` and `stat`. Explain the observed numbers.

(2.2) Expand your file until the disk occupancy increases and explain the behavior. Continue several times and try to predict the disk occupancy based on the actual size. Expand the file until megabytes or even gigabytes. Was your prediction good? Why does the file system behaves like this?

(2.3) Create an empty file (null size) and explain how its disk occupancy can also be null.

(2.4) Create a file containing only one byte at position 10 000. Explain its size and disk space. And if you replace 10 000 by 10 000 000 , or 1 000 000 000 000 ?

3 Concurrent accesses

Consider the two following processes `R` and `W`:

```
r () /* Process R */
{
    int fd;
    char buf1[512];
    char buf2[512];
    fd = open ("test", O_RDONLY);
    /* Reading R1 */
    read (fd, buf1, sizeof (buf1));
    /* Reading R2 */
    read (fd, buf2, sizeof (buf2));
    close (fd);
}
```

```
w () /* Process W */
{
    int fd, i;
    char buf[512];
    fd = open ("test", O_WRONLY);
    for (i = 0; i < 512; i++) buf[i] = 'a';
    /* Writing W1 */
    write (fd, buf, sizeof (buf));
    for (i = 0; i < 512; i++) buf[i] = 'b';
    /* Writing W2 */
    write (fd, buf, sizeof (buf));
    close (fd);
}
```

The test file initially contains 1 024 `'c'` characters. The two processes are running in parallel.

(3.1) What can happen? What can be the states of the file and the buffers `buf` of the two processes after each input/output?

(3.2) Can we have in a buffer `buf` of the process `R` a sequence like `aaa ...aaaccc ...ccc` or `bbb ...bbbccc ...ccc`?

(3.3) Can we have an "atomic" system call? Is it desirable? Is that reasonable? How the system can manage this?

(3.4) How can tou modified these two processes to ensure that the process making its first input/output make its second before the other process? In other words, how two cooperating processes can ensure their mutual exclusion for the file access?