UNIVERSITY OF COPENHAGEN
Department of Computer Science
OSM 2015 Course Team

Version 1.0, March 6, 2015

G-Assignment 5 for "Operating Systems and Multiprogramming", 2015

Deadline: March 16 at 23:59
Handing in: via Absalon

Formalities

Please, follow the same procedure as in the previous assignments. In the following tasks, you are supposed to extend Buenos. Hand in 1) a source tree for Buenos which contains your work, including a `Makefile`, and the programs used for testing; and 2) a short report where you explain what you have done.

Task 1: System calls for the Buenos file system

The system calls `syscall_read` and `syscall_write` were implemented in G1, but they only worked on `stdin` and `stdout`. We have extended them to work on `stderr` as well (see the package `buenos-g5`). Now your task is to extend these system calls to work with general-purpose file handles. Also, other system calls are to be added, in order to open, close, create, remove, seek, and tell files.

Implement the system calls as described below, following the Buenos Roadmap, § 6.4 and § 8, as closely as possible. In case of ambiguities, clearly state your assumptions in the comments and report. Try to identify all possible error and race conditions.

Use only calls to the virtual-file-system (VFS) layer of Buenos, and the error codes defined there (see `fs/vfs.[hc]`). **Do not** use any parts of the trivial file system (TFS) directly. The use of kernel panic (rather than good use of locks and error return codes) should be your last resort.

proc/syscall.h

```
...
#define SYSCALL_OPEN    0x201
#define SYSCALL_CLOSE   0x202
#define SYSCALL_SEEK    0x203
#define SYSCALL_READ    0x204
#define SYSCALL_WRITE   0x205
#define SYSCALL_CREATE  0x206
#define SYSCALL_REMOVE  0x207
#define SYSCALL_TELL    0x208
...
#define FILEHANDLE_STDIN    0
#define FILEHANDLE_STDOUT   1
#define FILEHANDLE_STDERR   2
...
```

`int syscall_open(const char* pathname)`
Effects: Opens the file identified by `pathname` for reading and writing. A path name is a volume name followed by a file name (for instance, `[disk1]halt`).
Returns: A positive integer greater than 2 (serving as a file handle to the opened file) on success, a suitable VFS error code on error.

`int syscall_close(int filehandle)`
Effects: Closes the (perhaps) open file identified by `filehandle`. The file handle is hereafter invalid for use in file operations.
Returns: 0 on success, a suitable VFS error code on error.

`int syscall_create(const char* pathname, int size)`
Effects: Creates a new file of the given `size` with the name specified by `pathname`.
Returns: 0 on success, a suitable VFS error code on error.

`int syscall_remove(const char* pathname)`
Effects: Remove the file identified by `pathname` from the file system it resides on. If the file is open, it is marked for removal, and removed once the file is closed.
Returns: 0 on success, a suitable VFS error code on error.

`int syscall_seek(int filehandle, int offset)`
Effects: Sets the file position of the file identified by `filehandle` to `offset` (in bytes from the start of the file). Seeking beyond the end of the file leads to undefined behaviour (from the point of view of the VFS).
Returns: 0 on success, a suitable VFS error code on error.

`int syscall_tell(int filehandle)`
Effects: Tells the current file position of the file identified by `filehandle` (in bytes from the start of the file).
Returns: Current file position on success, a suitable VFS error code on error.

`int syscall_read(int filehandle, void* buffer, int length)`
Effects: Reads at most `length` bytes from the file identified by `filehandle` into the specified `buffer`, starting at the current position, and advancing the file position.
Returns: The number of bytes actually read (e.g. before reaching the end of the file), a suitable VFS error code on error.

`int syscall_write(int filehandle, void const* buffer, int length)`
Effects: Writes at most `length` bytes from the specified `buffer` to the file identified by `filehandle`, starting at the current position, and advancing the position.
Returns: The number of bytes actually written, a suitable VFS error code on error.

Base your implementation on the package `buenos-g5`, which supports user processes and semaphores. Adapt the implementation to work correctly with the modified system calls. At the very least, you should manage a list of open files per user process, and close all files opened by the process (except `stdin`, `stdout`, and `stderr`) before a process terminates.

Task 2: GrowFS, an extended file system for Buenos

Extend the trivial file system (TFS) shipped with Buenos such that the write operation is allowed to append data to a file by writing beyond the previous end of the file. Call the extension GrowFS.

1. Base your implementation on the code for TFS (in `fs/tfs.[ch]`, but create a new file system which largely resembles TFS. Leave the TFS block layout and functionality unchanged, except for the implementation of `growfs_write`. Extend this function to write beyond the end of a file by allocating additional blocks on the disk (like `tfs_create` does when creating files) and adding the new block(s) to the file inode. After allocating the additional space, the write operation can be performed using the former `tfs_write` code.

   If no free blocks are available, the operation should fail with `VFS_ERROR`. If the file would grow beyond the maximum file size in TFS/GrowFS (how much is that?), the write operation should fail with `VFS_LIMIT`.

   Discuss other possible error cases in your report, and choose the point at which the modified block allocation table and file inode should be written back to disk to avoid losing data.

2. Use the magic number `0xD00F` to identify the file system in the superblock and add GrowFS to the list of file systems in Buenos (in `fs/filesystems.c`), so that a volume using GrowFS can be mounted.

   Since the overall file system layout of GrowFS is identical to that of TFS, you can create a GrowFS volume using `tfstool` and modify the magic number using a hex editor (or you can modify the `tfstool` source code).

3. Test the functionality of the new file system. Your tests should cover at least:

   - mounting a volume and executing an initial program from it;
   - creating an empty file and then appending data to it;
   - copying a file from a TFS volume to a GrowFS volume and comparing these.

4. **BONUS; only if you are done with the rest; no extra points, but a small prize will be awarded to the team(s) who will come with the best solution.**

   Although files can grow, they cannot grow very large. Drawing inspiration from Figure 11.9 on page 512 of the course book [SGG], add single, double, and triple indirect blocks to a GrowFS inode. How big can the files get now? Test that your implementation can handle files that big. Can you come up with a better[1] inode structure?

[1] The word *better* is intentionally vague.