UNIVERSITY OF COPENHAGEN
Department of Computer Science
Jost Berthold berthold@di.ku.dk
Jyrki Katajainen jyrki@di.ku.dk

March 7, 2014

G-Assignment 5 for "Operating Systems and Multiprogramming", 2014

Deadline: March 17 at 23:59
Handing in: via Absalon

Formalities

Please, follow the same procedure as in the previous assignments. Again 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

System calls `syscall_read` and `syscall_write` have been implemented in G-assignment 1, but they only worked on `stdin` and `stdout`. Now these system calls should be extended to work with other file handles as well (`stdin` and `stdout` are just special file handles). Other system calls are to be added, in order to open, close, create, and delete files.

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_DELETE 0x207
#define SYSCALL_FILECOUNT 0x208
#define SYSCALL_FILE 0x209
...
```

Implement the system calls described below (following the Buenos roadmap, Section 6.4 and Chapter 8). Use only calls to the virtual-file-system (VFS) layer of Buenos (declared in `fs/vfs.h` and implemented in `fs/vfs.c`), and VFS error codes defined there. Do not use trivial-file-system (TFS) functions directly.

`int syscall_open(char const* pathname)`
Effects: Prepares the file referenced by `pathname` for reading and writing. A path name includes a volume name and a file name (for instance `[root]a.out`).
Returns: A positive integer greater than 2 (serving as a file handle to the file opened) on success or a negative integer on error.

`int syscall_close(int filehandle)`
Effects: Invalidates the file handle passed as parameter. Hereafter the handle cannot be used in file operations any more.
Returns: 0 on success or a negative integer on error.

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

`int syscall_delete(char const* pathname)`
Effects: Deletes the file referenced by `pathname`. The operation fails if the file is open.
Returns: 0 on success or a negative integer on error.

`int syscall_seek(int filehandle, int offset)`
Effects: Sets the reading or writing position of the open file identified by `filehandle` to the indicated *absolute* offset (in bytes from the beginning).
Returns: 0 on success or a negative integer on error. Seeking beyond the end of the file sets the position to the end and produces an error return value.

`int syscall_read(int filehandle, void* buffer, int length)`
Effects: Reads at most `length` bytes from the file identified by `filehandle` (at the current file position) into the specified buffer, advancing the file position.
Returns: The number of bytes actually read (before reaching the end of the file) or a negative value when an error occurred.

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

Base your implementation on a Buenos version which supports user processes (use the one provided in the model answer for G-assignment 3 or your own), and adapt the user-process implementation to work correctly with the modified system calls. You should manage a list of open files per user process.

Task 2: A simple shell and directory listing support

Together with this problem formulation, we hand out source code for the `osh` shell (our shell), which a simple shell that makes the use of Buenos easier. The shell can already execute programs (when a path to a program is given as a command), echo text (using command `echo`), show the contents of files (using command `show`), and read a line into a file (using command `read`).

Extend `osh` with the following commands that facilitate the manipulation of files in a volume:

`exit`: Terminates the shell (no arguments)
`rm`: Deletes a file from the file system if it exists (1 argument)
`cp`: Copies the contents of one file to another (new or overwritten) file (2 arguments)
`cmp`: Compares the contents of two files byte by byte (2 arguments)
`ls`: Lists the files of a volume (1 argument) (use the system calls described below).

Without any additional support, the user needs to know the contents of a volume to use the shell. To make `ls` possible, add the following two system calls to Buenos VFS, and implement them using TFS. Use the system call numbers given in Task 1.

`int syscall_filecount(char const* name)`
Returns: The number of files in the volume specified by the given name (whose maximum length is `VFS_NAME_LENGTH`). If the name is `NULL`, the return value is the number of mounted file systems. If the name is that of a mounted volume name, the return value is the number of files in this volume. Otherwise, and on errors, a suitable VFS error code (a negative integer) is returned.

`int syscall_file(char const* name, int index, char* buffer)`
Effects: Depending on whether `name` is `NULL` or the name of a mounted volume, this system call operates either on the table of mounted volumes in the system, or on the table of contents of a mounted volume (the "master directory" in Buenos TFS). The call copies the name of the file or volume at the specified `index` in the designated table to the specified buffer, which must be pre-allocated (its size must be at least `VFS_NAME_LENGTH`). The table is traversed sequentially, leaving out unused entries.
Returns: 0 on success or a suitable VFS error code on error.

Bonus task: Background and foreground processes

Design a way for the user to run programs in the background. The user should be able to start a background process and later issue a command `wait` to wait for its termination.