File Operation Syscalls

TABLE OF CONTENTS

In addition to the process control syscalls, you will also need to implement the following file operation syscalls: `create`, `remove`, `open`, `filesize`, `read`, `write`, `seek`, `tell`, and `close`. Pintos already contains a basic file system which implements a lot of these functionalities, meaning you will not need to implement any file operations yourself. Your implementations will simply call the appropriate functions in the file system library.

Keep in mind that most testing programs use the `write` syscall for output. **Until you implement `write` syscall, most test programs will not work**.

Implementation details

Pintos' file system is not thread-safe, so you must make sure that your file operation syscalls do not call multiple file system functions concurrently. In Project File Systems, you will add more sophisticated synchronization to the Pintos file system, but for this project, you are permitted to use a global lock on file operation syscalls (i.e. treat it as a single critical section to ensure thread safety). **We recommend that you avoid modifying the `filesys/` directory in this project**.

While a user process is running, you must ensure that nobody can modify its executable on disk. The `rox-*` tests check that this has been implemented correctly. The functions `file_deny_write` and `file_allow_write` can assist with this feature. Denying writes to executables backing live processes is important because an operating system may load code pages from the file lazily, or may page out some code pages and reload them from the file later. In Pintos, this is technically not a concern because the file is loaded into memory in its entirety before execution begins, and Pintos does not implement demand paging of any sort. However, you are still required to implement this, as it is good practice.

Your final code for Project User Programs will be used as a starting point for Project File Systems. The tests for Project File Systems depend on some of the same syscalls that you are implementing for this project, and you may have to modify your implementations of some of these syscalls to support additional features required for Project File Systems. While you certainly will not be able to plan too far in advance for Project File Systems, we recommend you write good code that can be easily modified by keeping good documentation.

Syscall signatures

create

```
bool create (const char *file, unsigned initial_size)
```

Creates a new file called `file` initially `initial_size` bytes in size. Returns true if successful, false otherwise. Creating a new file does not open it: opening the new file is a separate operation which would require an `open` system call.

remove

```
bool remove (const char *file)
```

Deletes the file named `file`. Returns true if successful, false otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it. See this section of the FAQ for more details.

open

```
int open (const char *file)
```

Opens the file named `file`. Returns a nonnegative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened.

File descriptors numbered 0 and 1 are reserved for the console: 0 (`STDIN_FILENO`) is standard input and 1 (`STDOUT_FILENO`) is standard output. `open` should never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

Each process has an independent set of file descriptors. File descriptors in Pintos are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each open returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to `close` and they do **not** share a file position.

filesize

```
int filesize (int fd)
```

Returns the size, in bytes, of the open file with file descriptor `fd`. Returns `-1` if `fd` does not correspond to an entry in the file descriptor table.

read

```
int read (int fd, void *buffer, unsigned size)
```

Reads `size` bytes from the file open as `fd` into `buffer`. Returns the number of bytes actually read (0 at end of file), or -1 if the file could not be read (due to a condition other than end of file, such as `fd` not corresponding to an entry in the file descriptor table). `STDIN_FILENO` reads from the keyboard using the `input_getc` function in `devices/input.c`.

write

```
int write (int fd, const void *buffer, unsigned size)
```

Writes `size` bytes from `buffer` to the open file with file descriptor `fd`. Returns the number of bytes actually written, which may be less than `size` if some bytes could not be written. Returns `-1` if `fd` does not correspond to an entry in the file descriptor table.

Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or 0 if no bytes could be written at all.

File descriptor 1 writes to the console. Your code to write to the console should write all of `buffer` in one call to the `putbuf` function `lib/kernel/console.c`, at least as long as size is not bigger than a few hundred bytes. It is reasonable to break up larger buffers. Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our autograder.

seek

```
void seek (int fd, unsigned position)
```

Changes the next byte to be read or written in open file `fd` to `position`, expressed in bytes from the beginning of the file. Thus, a position of 0 is the file's start. If `fd` does not correspond to an entry in the file descriptor table, this function should do nothing.

A `seek` past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. However, in Pintos files have a fixed length until Project File Systems is complete, so writes past end-of-file will return an error. These semantics are implemented in the file system and do not require any special effort in the syscall implementation.

tell

```
int tell(int fd)
```

Returns the position of the next byte to be read or written in open file `fd`, expressed in bytes from the beginning of the file. If the operation is unsuccessful, it can either `exit` with `-1` or it can just fail silently.

close

```
void close (int fd)
```

Closes file descriptor `fd`. Exiting or terminating a process must implicitly close all its open file descriptors, as if by calling this function for each one. If the operation is unsuccessful, it can either

`exit` with `-1` or it can just fail silently.
