

Practical 3 Report - User Spaces: ls command

Design & Implementation

To implement the ls command, I decided to create a new system call, `listdir`, specifically for reading directories. This approach allowed me to have a simple and clear implementation, building on the existing filesystem and syscall structure of the OS and only having to add a few additional functionalities.

Kernel - On the kernel side, I added a new syscall number (18) as well as a shared `directory_entry` struct to store the properties of each entry: name, size and type (0 = file, 1 = directory). I opted to store the name in a fixed-size buffer, allowing up to 63 characters plus a null terminator, as most file names fall under that limit, truncating them if not. In `fat.h`, I also added 2 simple public helper methods to access a node's children list (`children()`) and its file size (`data_size()`), to be then used in the syscall handler.

The `listdir` handler can be found in `syscall.cpp`, which interprets the arguments as `arg0` - absolute path, `arg1` - pointer to a userspace buffer of `directory_entry` structs and `arg2` - maximum number of entries. It first validates that the given path is an existing directory, then downcasts the node to a `fat_node*` (due to guaranteed FAT filesystem usage) and retrieves its children list. Iterating over this list, stopping if `max_entries` is reached, each child is transformed into a `directory_entry` struct using `memops::strncpy` and explicit null termination to copy the name, `data_size()` for size and `kind()` for type. Finally, on success, the system call returns "ok" and the number of entries written in the data field. As an extension, I implemented a two-phase API, where the first call with a null buffer and `max_entries=0` only returns the number of children, which the second call then uses instead of a fixed arbitrary limit.

Userspace - In the userspace, the main method of the `ls` program uses the syscall via a small wrapper `listdir(path, buffer, max_entries)` found in `user-syscall.h`. It first parses command-line flags, only accepting: `-l`, `-n`, `-s`, `-ln` and `-ls`, where the `-l` flag enables long mode (printing type and size), while the `-n` and `-s` flags enable sorting by name or size. The functionality is carried out in 2 phases: first phase obtains the total number of entries by calling `listdir(path, nullptr, 0)` and second phase allocates exactly enough memory for all the directory entries and calls `listdir` again to now actually fill the buffer.

After retrieval, entries are optionally sorted using a simple selection sort $O(n^2)$ by name or size. Then, the current and parent directories ("." & "..") are omitted. For long mode, each line prints the type of the entry, its name and its size in bytes. To make my output clearer, I calculate the maximum name length and pad the shorter names so that all the file sizes are aligned to the same column.

Additional Features, Limitations & Overview

For this practical, the extra features I have implemented include additional flags for sorting the output based on file names or sizes, a clearly-readable spaced output and a two-phase API.

The limitations of my code included the truncation of file names that exceed 63 characters and a naive $O(n^2)$ sort, which is acceptable here due to small directory sizes, but would not scale well for larger ones. Nonetheless, the two-phased design avoids buffer overflows and arbitrary fixed limits, and the kernel never exposes internal filesystem structures directly to userspace, as directory entries are passed through an intermediate `directory_entry` struct, ensuring proper well-defined kernel-to-userspace interaction.