CS3104 - P3 - ls

November 29, 2024

220015759

Introduction

In our final Operating Systems practical, we were tasked with implementing the `ls` function for STACSOS. The specifications provided hints and some guidance regarding how the file system is structured and where children are stored.

Design

The `ls` function - when given a directory path - will list out the entries of that directory. The base requirement also includes the `-l` flag which will display some more information on screen like the type, name and size.

The idea is to have a *directory cursor* obtained from the `opendir` syscall. The cursor will act like an iterator where the `readdir(cursor, ptr)` syscall will return the next directory entry until there are no more children left. This way, the syscall is quite small and provides a simple and well defined interface to create and get directory entries. Then it is up to the userspace program (`ls`) to collect the data, perform operations and display them. I believe this approach follows the Unix philosophy of "do one thing and do it well".

The cursor will be different from a file (although everything is a file). The `opendir` syscall will take in a path and call `vfs::get().lookup(path)` to traverse the file structure and retrieve the corresponding `fs_node`. Then, a `directory` object is created and stored in kernel space through `object_manager` as a `shared_ptr<fs::directory>`. This is wrapped up within a `dir_object` which is a subtype of `object`. The userspace process will get an id (`u64`) which is treated as the directory object and passed into the `readdir` syscall. The kernel then accesses the object using that id which returns the directory object that we've created. In order to get the contents of the data (since it will return an `object` type), the `pread` function is overwritten to return the underlying directory struct and then the `fs_node` that we had located in the beginning. Additional abstract and concrete methods have been added to `fs_node` such as `get_next_child()`, `total_children()` and `get_name()`. These changes are also in place for all classes that inherit from `fs_node` such as `tarfs_node`, `rootfs_nods` and `devfs_node`. Moreover, a `dirdata` struct has been defined under `lib` making it visible to both the kernel space and user space. Due to the restrictions between how kernel and user space communicate with one another, the `dirdata` struct is created in the user space and the address is passed as a `void *` to the kernel space where directory information is written from kernel space to user space.

For more functionality, the `-r` flag can be set to recursively perform `ls` on subdirectories. The flags are represented as an integer bitmask which is easily extended upon and passed through for subsequent recursive function calls. Moreover, as an extension, relative path resolution has been implemented on `vfs` so we can just type `ls` on the command line instead of `/usr/ls`. To implement sorting, there has to be a way for STACSOS to store all directory entries. Due to limited memory, this is not feasible since it'll simply run out of memory and page fault for larger directories.

Conclusion

To conclude, I found this practical quite enjoyable. Through the lecture slides I could say that I know how it works under the hood. After this practical, I can say I know how it works inside the engines. Had I have more time for this, I would look into implementing the `getcwd()` syscall by creating a directory stack for each process and implement the `cd` command that will push and pop from that stack. Additionally, I've noticed that the close syscall does not do anything, and perhaps a delete method for the AVL tree could be implemented for something more concrete. At the moment, the LS command may very well be a memory leak.