CS3104 P3

220011263

November 28, 2024

1 Approach to Implementing Directory Listings

The general approach to implementing ls is as follows:

- In main.cpp, the command line string was parsed and the presence of flags (e.g. -l) was evaluated.
- After argument parsing, an object pointer is created using the listdir_() in objects.cpp. This function would then invoke the custom syscall listdir_() (found in user-syscall.h).
- The custom syscall in user-syscall.h invokes the prewritten syscall4() with the syscall number of listdir_ (found in syscalls.h), the directory path and flag booleans as arguments.
- syscall4() would then call the static function listdir_() (found in syscall.cpp) which:
  - Uses lookup() (found in vfs.cpp) to return a pointer to the tarfs_node representation of the supplied directory path.
  - Using this tarfs_node representation, the number of children nodes are received by calling an implemented getter (found in tar-filesystem.h) and the function loops through each index, accessing the child node as an fs_node and its metadata (ie. name, size and file kind), all through implemented getters. This metadata is added to lists of strings, u64 and fs_node_kind.
  - Once the loop concludes, the lists of information are passed into an directory.h object constructor, along with other information such as the booleans for the flags.
  - A shared ptr storing the directory object is also created and a directory object is created in the object manager, allowing the access of the directory object in user space via a unique ID that is passed back through the syscall_result and fa_result structs.
- Back in main.cpp, a buffer is created and the read() method of the directory.h instance is invoked. The information from the object is read into the buffer using read() while keeping in mind what flags were included in the command line. The function can be found in kernel/inc/fs/directory.h. After read() completes, the buffer is printed on the shell screen using console::get().writef().

All of the modified files contain comments which further explain design choices and program flow.

2 Design and Implementation Choices

The choice to use an object oriented route was influenced by the cat folder as it also utilises a 'file' object. Implementing a custom syscall was chosen over overwriting read/pread since read/pread did not include useful functions for which this practical required, plus, from a debugging standpoint, creating additional syscalls would prevent any breaking of original code that may occur from modifying/overwriting read/pread. Arbitrary padding was introduced to improve readability of listings on the shell screen. A singular function was used to handle all combinations of flags so to reduce code repetition and maintainability. To test the -a flag, a new directory, test_hidden, was created with some hidden and non-hidden files (found in the sysroot folder).

2.1 Challenges and Limitations

A slight challenge in implementing the ls command was utilising custom types and custom data structures (ie. stacsos::string and stacsos::list) as you had to read the documentation since resources regarding these are not available online.

The limitations of this ls program are as follows:

- Only 3 flags supported: -a, -l, -U. Incorporating more flags should not be difficult given how codebase is structured, however another flag would require the creation of a new syscall (e.g. syscall5). This requires some knowledge surrounnding assembly and how STACSOS interacts with registers.
- -R was not implemented since STACSOS did not have enough memory for a recursive call stack (the thread and process stopped midway through outputting recursive listing).

2.2 Additional Features

To read more on the additional features of the ls implementation, please read the README.md found in the user/ls folder.