Implementing ls in StACSOS

For this practical I implemented directory listing support in StACSOS by adding a dedicated kernel system call and a userspace `ls` utility to consume it. I chose to introduce a new syscall rather than extend `read/pread` because those calls are designed for byte-streamed objects. Directories are structured collections of entries, not linear streams, and forcing them into a file-like model would break the abstraction and make read() semantics ambiguous.

`dirent` Structure

Directory entries are exposed through a simple` dirent` structure containing a fixed-size filename, an entry type, and a size field. This provides a clean, stable interface between kernel and user space and avoids exposing filesystem-internal data structures. Fixed limits on filename and path lengths further simplify memory handling by giving each `dirent` a predictable layout, avoiding dynamic allocation inside the kernel, and reducing the risk of overflows. The structure is also easily extendable should future metadata (e.g., timestamps or permissions) be added.

Kernel-Side Design

I added a new syscall, `get_dir_contents`, whose main logic lives in `do_get_dir_contents`, following the existing pattern used by syscalls such as open and `do_open`. The syscall takes a directory path, a userspace buffer, and a buffer size. It validates all arguments and returns detailed `syscall_result_code` values to distinguish errors such as invalid pointers, non-existent paths, or non-directory nodes.

The kernel resolves the provided path through the VFS and, if the target is a valid directory, populates the user buffer with an array of `dirent` entries. A buffer-based design is necessary because system calls can return only a small fixed-size value; a user-provided memory region allows the kernel to supply an arbitrary number of directory entries without maintaining state between calls. This keeps the kernel simple and stateless, allows efficient batch transfer of entries, and enables clean detection of buffer overflow if the directory contains more entries than fit.

My only change to the FAT filesystem driver was the addition of a public accessor `children` which exposes the list of child directory nodes belonging to a FAT node.

User-space Design

I added a userspace wrapper for the new syscall following the existing conventions in `user-syscall.h`, and then implemented the `ls` utility on top of it. The current implementation supports both short and long listing formats, recursive traversal, single-directory listing, and defaulting to the current directory when no path is provided.

I used a fixed buffer capacity of 64 entries as a reasonable upper bound for typical coursework directories while keeping memory use modest in the constrained environment. This also naturally lends itself to potential future support for pagination of long directory listings, although that extension was not implemented.

Known Bugs and Limitations

There is a known unresolved issue where the shell hangs after ls completes: the prompt does not reappear and input is ignored. This does not occur for recursive listings on valid paths. Debugging statements (still present in the code, jsut commented out) showed that the shell is blocked waiting on a` write` operation in most cases, or on `alloc_mem` during certain recursive error paths. Other limitations include silent truncation when directories exceed 64 entries (due to the lack of pagination), minimal output formatting without column alignment, and reliance on FAT-specific node structures. Generalising the syscall to operate purely on the abstract fs_node interface would make it compatible with future filesystem drivers.