To implement directory listings in StACSOS, I chose to overload the existing Virtual File System (VFS) interface rather than introducing a new system call. By treating a directory node as a readable file that returned structured metadata rather than raw bytes, I was able to reuse the existing `object::open()` and `pread()` infrastructure in user-space, minimizing changes to the kernel's system call table and maintaining a clean API surface.

The core of the implementation lies in the FAT filesystem driver. I observed that `fat_node::open()` previously returned a `fat_file` regardless of the node type. I modified this method to check the node's kind to return either `fat_file` or `fat_directory_file`, which I created to deal with directories. Additionally, instead of re-parsing raw disk sectors, my `fat_directory_file` implementation leverages the existing list of children stored by `fat_node` when `load()` is called.

To pass data to user-space, I defined a new shared structure, `struct dirent`, that encapsulated the filename, file kind, and file size. I also implemented the `pread` method for `fat_directory_file`, which interprets the read offset as an index into the children list, as opposed to a byte position for a regular file. When `ls` requests data at offset, the kernel first retrieves the child node at `index = offset / sizeof(dirent)`. It then serializes the child's metadata into a `dirent` instance, then copies the struct into the user-space buffer. This allows the user-space program to iterate through the directory one entry at a time after it receives the `dirent` data.

In my main code for the `ls` utility, it first parses command-line arguments to identify flags and the target path. It opens the directory using `object::open()`, then enters a loop where it calls `pread` repeatedly, incrementing the offset by `sizeof(dirent)` after each successful read. The loop terminates when `pread` returns 0, indicating the end of the directory.

A significant challenge arose during testing where `pread` returned an unexpected number of bytes. Adding debugging lines to my code revealed that the user-space compiler and the kernel compiler padded the `dirent` struct differently. I resolved this by applying `__attribute__((packed))` to the struct definition, ensuring a consistent struct size for both the kernel and user space.

One limitation about my implementation of the `ls` command is that it does not display file timestamps. When I was trying to explore this route when looking for extra bits to add to my implementation, I found that the current kernel `fat_node::load` implementation does not extract timestamp bytes from the raw FAT directory entry, so this data is unavailable to the VFS layer. Addressing this would require significant refactoring of the physical filesystem driver to parse and store these additional attributes in the `fat_node` class, which I did not have the time and knowledge to accomplish.

Additionally, I implemented human-readable file sizes, invoked via the `-h` flag. When combined with the long listing format (`-l`), this feature converts raw byte counts into user-friendly units (B, K, M, G). My implementation simply uses floating-point arithmetic to calculate the size relative to the appropriate power of 1024. To ensure readability, it dynamically formats the output: sizes smaller than 10 units include a decimal point (e.g., 4.5K), while larger sizes are rounded to whole numbers (e.g., 15M). This significantly improves the user experience when inspecting larger files.