CS3104 P3 - userspace

StudentID: 230016633

Introduction - In this project, I implemented an ls programme in StACSOS. To accomplish this I implemented two syscalls, opendir and readdir, a new "directory" class in the VFS and FSD and used the Object Manager to store open directories. In this assignment I prioritised conforming to the structures laid out by stacsos to ensure good compatibility with the system.

How directory data is collected in the kernel - I added a directory class to the VFS, which acts as an abstraction to the matching class I implemented in Filesystem Driver. It tells the kernel how it can interact with directory objects and tells the FSD what functions it should implement. It is important this abstraction exists to maintain FSD modularity.
The FAT Directory class stores a list of FAT Nodes (the directory's children) so that the readdir function can obtain the necessary entry data like name, size and type.

Sending data to User Space - User space and Kernel space have different virtual address spaces. If I were to pass a pointer from kernel to user space, it would likely point to somewhere different and would possibly cause an error if that memory address isn't mapped. If I were to set a buffer to point to a value in kernel space, there is no guarantee that value remains the same or still exists when it is used in user space. I used memcpy to copy data in the buffer that user space provided, so the user space's safe access to that memory is already proven.

As required by the -f flag, more than just filename needs to be transfered, so some agreed upon data structure must be used in both the kernel and user space when writing/reading the buffer. I added a Dirent struct to the standard library (available to both spaces), which describes members like entry name, size, and type.
Initially, readdir copied only the next entry into the buffer. To reduce the number of syscalls required, It fills the buffer with as many entries as can fit and returns the number written, which is more helpful as it allows the user space to better monitor which information in the buffer is new.

Storing Directory Objects- I didn't want a limit on the number of entries that could be shown. I needed to support multiple calls of readdir because buffer size limits information passed per call.
Storing a directory object in the object manager stores it in the kernel over multiple syscalls. The object manager stores object tables which make lookup efficient so using it is ideal. The opendir syscall creates the object in the manager and readdir retrieves it and calls its function.
The opendir syscall creates the object in the manager and readdir retrieves it and calls its function, separating concerns between them.

Not storing the object means reconstructing it with each call. That requires a filepath lookup and loading a node. In addition, tracking the current file index is no longer abstracted by the directory class. My implementation using the Object Manager is a cleaner, more efficient approach.
I use a Directory object in user space (using Objects.h) which abstracts both making syscalls and storing the object's ID (which the object manager uses to get the object).

Additional Descision Justifications - The Dirent struct is of a fixed size, rather than using a flexible entry_name field. Though this limits entry name length, it makes sizing buffers easy.
Instead of overloading read so it works for both files and directories, I wanted to make a separate system call. The methods Files and Directories implement are very different, so I wanted clear separation between their system calls. Hence, the opendir and readdir syscall. I created the Directory class for the same reason.