Assignment 2

Now that you have an understanding of ELF files, the EXT2 file system, and basic disk loading operations, you are ready to tackle the "TO DO" sections of the code that were discussed in the lecture. (**This assignment requires you to have a fully working version of the previous bootloader assignment. If you do not, you should not attempt this one until the previous one is working correctly. No inline assembly is permitted.**)

Using the solution directory you created for assignment 1, rename the directory to SimpleOSv4-FileSystem. (You should not keep the directory named bootloader-base. For each assignment, you will need to rename the directory for clarity.) Now, in Github, download and merge the files from SimpleOSv4-2-Filesystem-Base with your previous solution. There are two files present in the folder. Take those two files and replace the corresponding files in your assignment 1 solution directory. Similar to assignment 1, you cannot modify any function signatures, add additional functions, etc. You are simply to implement the functions marked as "ASSIGNMENT 2 TO DO" in fs.cpp.

For this assignment, you will need to implement the following functions in fs.cpp. Be sure to become familiar with constants.h and fs.h for this assignment as well as review the links to relevant material in the current module in Canvas. Don't forget to consult the SimpleOS v4 Reference Manual in Canvas and the online Doxygen documentation.

diskReadSector()

The purpose of this function is to read a logical sector number from the hard drive and load it to the provided memory address. I have included the high-level steps this function must perform below.

1. Perform a diskStatusCheck() call
2. Set the primary ATA sector count to 1 sector
3. Set the low-byte and mid-byte sector bytes (using masks and possibly shifts) from sectorNumber
4. Set the high-byte sector number to zero (not needed in this small implementation)
5. Set the primary ATA driver header register to 0xE0
6. Issue an ATA read command to the primary ATA device
7. Perform another diskStatusCheck() call
8. Perform an ioPortWordToMem() call to the correct destination memory and for the number of words appropriate for the sector size

diskWriteSector()

The purpose of this function is to write a sector of data from the provided memory address to the provided sector on the hard drive. You will repeat the process outlined in the function above except it will be performing a write instead of a read.

1. Repeat the process of diskReadSector() here but do it for writes (making any appropriate modifications)

loadElfFile()

The purpose of this function is to parse an ELF header at the provided memory address and load the text and data segments to their preferred memory locations.

1. This function will parse the ELF header at the incoming destination memory address and determine the text segment p_vaddr location.
2. Once that is known, perform a memoryCopy() call to copy the entire text segment to that destination memory using that segment's p_memsz value. (*** p_memsz is in bytes, memoryCopy() uses words, not bytes.)
3. Repeat for the data segment. Only copy those two segments.

loadFileFromInodeStruct()

The purpose of this function is to load a file based on the provided inode struct to the provided file buffer (as a temporary location). (An inode structure will be stored at the incoming inodeStructMemory address by a call to fsFindFile(). You don't have to call fsFindFile() as it will be called for you in the bootloader-stage2.cpp code.)

1. Parse all direct blocks in the inode struct and copy those blocks to the destination memory address (char *fileBuffer). If the direct block is zero, increment the fileBuffer pointer by the BLOCK_SIZE.
2. Read the first indirect block, if not zero, copy all the indirect block pointers to EXT2_INDIRECT_BLOCK.
3. Parse the EXT2_INDIRECT_BLOCK as if it were an array of 256 more direct blocks, repeating the process above.

Additionally, be sure to use the constants (from constants.h) whenever possible. You should not be using a hard-coded value if there is a corresponding constant in constants.h, (but do not modify, add to, or alter constants.h in any way). You cannot create new helper functions and you should not create any new global variables. **Be sure to follow a professional coding standard, including the naming of your variables, structures, etc. You should use descriptive names that are appropriate. Nonsensical or whimsical names will lose points. You must also use the custom types where I have supplied them (e.g., uint8_t, uint16_t, uint32_t) unless you need a signed typed or one I haven't supplied. You will lose points if you don't follow this convention.**

Once you have everything working, you should see the kernel binary display the following message (see screenshot below). When done, submit only the SUBMIT-ME.zip file to Canvas. Most of the grade is based on your implementations working but I will also be looking for the use of constants from constants.h and generally clean code (refer back to the lecture videos for good and bad examples).

Additionally, you must attest and add the following honor statement to your Canvas submission (in the Canvas submission comments):

I, <state your name>, attest that this submission is my own work. I have not worked with anyone else on my implementation and have not used AI or other unpermitted sources to help me with my solution, in accordance with our definition of class cheating shown in Lecture 1.

Any submission that does not include this attestation will not receive any credit.