CSI 385 Design Document

Matthew Fortier & Tony Taylor

December 7, 2016

1 Project Overview

The project folder is divided into two folders `commands` and `package`. The `commands` folder will contain a set of sub-folders for each command which will house the `.c` file as well as the compiled program and any other header files or supporting files. The `package` folder houses all of the given files including all the floppy drives, `fatSupport.c` and `fatSupport.h`. In the main project folder will be located the `shell.c`, `shell.h`, the compiled shell, the `Makefile`, and the `README.txt`. The makefile will go through all of the sub folders and compile all of the commands when `make` is called. The user must then call `./shell` in the main project directory to start the shell.

2 Shell Description

The shell has three main components: Reading input from user, parsing that input into commands and argument, and third executing the proper function.
Author: Matthew Fortier

2.1 Files

shell.c
shell.h

2.2 Required Files

fatSupport.c
fatSupport.h

2.3 Functions

2.3.1 int main(int argc, char** argv)

The main function here will keep the following three functions in a loop until the user wishes to exit. It will continue to read the input, parse it, and execute the proper command. It also has a commands for help that will give the user some information and commands formats. The main also reads in the shared memory to set the cluster and path to `0` and `"/"` respectively at the start of the program. This is the "main" main of the entire shell.

2.3.2 char* readInput()

The `readInput()` function is pretty simple in design. It continues to read in characters to a temporary char array or "string" until it detects that the user has entered `"\n"` which is the enter/newline character. The function uses `getchar()` and then does simple character comparisons.

2.3.3 char ** parseInput(char line[], const char *delimiter)

The `parseInput()` function breaks the command given by the user into an array of arguments that can be passed to the `executeCommand(cmd[])` function. This function makes sure that the command is still the first element in the array returned. The function first counts then number of tokens that it will break the string into. It then allocates enough space for a string array of that size, the tokenizes the string until the full array is created. It then return the array.

2.3.4 int executeCommand(char *cmd[])

The `executeCommand(char *cmd[])` function is arguably the most important function when it comes to the shell. It accepts the array of commands from `parseInput()` and looks for the first element which is the command. First the function forks a child process, then executes the commands using a large if statement comparing the command given. If the command matches, the function uses `execv(filelocation, arugments array)` to execute the compiled c program from another folder. The function then waits for the child process to join and allows for the next command to be processed

2.3.5 void upper_string(char s[])

The `upper_string` function takes in a string of characters, and one by one converts each character to upper case. This allows the user to enter the uppercase or lowercase, and each command will still turn out fine.

2.3.6 int countDirectories(char** directories)

The `countDirectories` function takes in an array of items, typically directories, and essentially returns the length of the array given. In this case, it is returning the number of directories in an path.

3 Commands

3.1 cat

Abbreviation for concatenate.
Author: Matthew Fortier

3.1.1 Requirements

- If x is a file, print the contents of x to the screen.
- If x is a directory, print an error message.
- If number of arguments is not equal to one, print an error message.
- x can be an absolute path or a relative path.
- If x does not exist, print an appropriate error message.

3.1.2 Files

cat.c

3.1.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.1.4 Functions

main():
The main function for `cat` checks to see if the number of arguments are out of bounds, parses the filename using the `parseFilename(char* filename)` function, acquires the global path and cluster from the shared memory, and executes the `searchForFile()` function, and finally the `displayFile()` function.

parseFilename(char* input):
The `parseFilename(char* input)` function for `cat` tokenizes the string given by periods(.) to acquire the filename and the extension. The filename and extension are stored in a global array at positions 0 and 1 respectively, so they can be used later.

char* searchForFile(char* entry):
The `char* searchForFile(char* entry)` function for `cat` searches through each entry in the given sector, determined by the global current cluster from (`global_path->cluster`), and returns the entry where the given file was found or returns an empty entry.

void displayFile(char* buffer):
The `void displayFile(char* buffer)` function for `cat` is given the entry from the `char* searchForFile(char* entry)` function and first records the sector of that entry, the uses `printf` on the entire sector to print its contents. Because this is the `cat` command and if we know that the file exists and we returned the sector where the FAT table tells us the data is located, we can just print the entire sector. The functions then uses the sector that we acquired before to test if the next sector is used or not, it if is, we need to print the following sectors as well.

3.2 cd

Acronym for change directory.
Author: Matthew Fortier

3.2.1 Requirements

- Change to specified directory.
- If there is no argument available, change to the home directory

3.2.2 Files

cd.c
cd.h

3.2.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.2.4 Functions

main():
The `main()` function for `cd` does a lot here. It first retrieves the global_path struct from the shared memory for use in the rest of the command. It will then compare the directory given to a set of conditions:

- If the argument is null, then change directory to the root directory.
- If the argument is "/", then change directory to the root directory.
- If the first character of the argument is "/", than it is an absolute pathname, and we need to start the directory checking at 0.
- Otherwise the argument is a relative pathname, so we should start directory checking at the current cluster.
- If the relative path name is ".." than we need to get the cluster for the parent directory, and pop the end directory off the pathname.
- If the given pathname is absolute, we just need to replace the current path with the new one.

- If the given pathname is relative, we need to add it to the end of the path, unless it is "..", then remove.

The process for checking directories is pretty straight forward:

- Attain the cluster either from the current cluster for relative paths, or start at 0 for absolute paths
- Use the `parseInput(argv[1], "/")` functions to turn the given path into an array of directories
- Use the `countDirectories(char** directories)` function to return the number of directories in the path
- Iterate through each directory calling the `checkDirectory(addSpaces(directories[i]), cluster)` function on each which returns the cluster of the directory being checked.
- If the ending cluster is -1, somewhere along the way, a directory did not exist.
- If the directory does exists, the ending cluster will be greater than -1 and that is what cluster to change directories to.

int countDirectories(char directories):**
The `countDirectories(char** directories)` function for `cd` does a simple for loop that counts the number of items in the array provided and returns that number.

char * addSpaces(char* directory):
The `char * addSpaces(char* directory)` function for `cd` takes in a string, likely a directory or filename, and fills in the gaps with spaces so that comparisons between strings are more accurate.

char * generatePath(char directs, int directoryCount):**
The `addSpaces(char* directory)` function for `cd` takes in an array of directories, and the length of the given array, and turns it into a string. The function iterates through each item in the array, and for each item it uses `strcat(desitnation, given)` to add a "/" and the name of the directory after it. At the end of the loop, it adds an end of line marker('0') to the string and returns the string(path).

int checkDirectory(char *directory, int cluster):
The `checkDirectory(char *directory, int cluster)` function for `cd` takes in the directory the program is searching for and the current cluster it should be in. This function then calls `read_cluster(index, sectorBuffer, directory)` which returns an FLC. If the FLC is greater than 0, than the directory was found, if not it returns a -2 and breaks the loop. This process happens for every entry in the given sector.

int read_cluster(int marker, unsigned char* sect, char* directory):
The `read_cluster(int marker, unsigned char* sect, char* directory)` function for `cd` takes in the current marker for the entry to look at, the buffer to look in, and the directory string it is looking for. This function uses the marker given to access the specific information from the buffer. The function acquires the filename and extension for comparison, and the FLC on the data. If the given directory equals the entry filename and extension, it returns the FLC. The FLC can be acquired by doing some simple bit shifting of two bytes in the buffer:

- (((int) sect[marker + 27]) « 8) & 0x0000ff00
  ((int) sect[marker + 26]) & 0x000000ff

char ** parseInput(char line[], const char *delimiter):
The `parseInput(char line[], const char *delimiter)` function for `cd` takes in a string(char line[]) and a delimiter used to split the string the way the user wants. It returns an array of strings where the string is split where ever the delimiter is found. This is done pretty simply with use of `strtok`

char *trimwhitespace(char *str):
The `trimwhitespace(char *str)` function for `cd` takes in a string, and returns a copy of the given string, just without any spaces. This is the opposite of the `addSpaces(str)` function, used when comparing given string and acquired strings from the buffers.

char ** removeLastElement(char ** directories):
The `removeLastElement(char ** directories)` function for `cd` takes in an array of strings, likely directories, and returns an array without the last element. In hindsight, probably should have written it to remove any element specified.

3.3 df

Acronym for disk free.
Author: Matthew Fortier

3.3.1 Requirements

- Print the number of free logical blocks.
- If there are any arguments, ignore them.

3.3.2 Files

df.c

3.3.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.3.4 Functions

main():
The main function for `df` acquires the global path and cluster from the shared memory, and executes the `searchFileSystem()` function.

void searchFileSystem():
The `searchFileSystem()` function for `df` does all of the work for the command. It is the only function other than the main. There are only a few steps:

- Read in the fat buffer using the `read_sector()` function provided.
- Run a loop that searches all of the sectors starting at 31 and goes to MAX_SECTORS which is 2487.
- Read the current sectors using the `read_sector()` function provided.
- Get the fat entry for the current sector using the `get_fat_entry()` function provided.
- If the fat entry is not equal to 0, than it is used, so increment a used counter;
- Print out the information acquired

3.4 ls

Acronym for list.
Author: Matthew Fortier

3.4.1 Requirements

- When x is the name of a file, list the name of the file, with its extension, the file type, FLC and file size.
- When x is the name of a directory, then list the names of the entries (with extensions) within the directory along with FLC, file size and file type (file or directory). Also, list the "." (current directory) and ".." (parent directory) entries.
- If x is a blank space, for each entity in the current directory, lists its name with extension, FLC, file size in bytes and file type. Also, list the "." and ".." entries.
- If there are two or more arguments, return an error message.
- Keep in mind that "/", "." and ".." are valid values for x.
- x can be a relative or absolute file name.

3.4.2 Files

ls.c

3.4.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.4.4 Functions

main():
The `main()` function for `ls` is where a lot of the logic behind ls happens. It first tests to see if there are too many arguments. Then acquires the struct from the shared memory. It then uses some conditionals to figure out which sector to read in based on the arguments given and the current cluster:

- If there is no argument given, read the sector into the buffer using `read_sector()` function based on the current cluster.
- Otherwise, check to see if the given path is absolute or relative.
- Get the FLC of the given pathname using the same logic as the cd command.
- Read in the sector with the found FLC using `read_sector()`.
- Iterate through the sector given and print the files and directories using `read_cluster`.

int read_cluster(int marker, unsigned char* sect, char* directory):
The `read_cluster(int marker, unsigned char* sect, char* directory)` function for `ls` takes in the current marker for the entry to look at, the buffer to look in, and the directory string it is looking for. This function uses the marker given to access the specific information from the buffer. The function acquires the filename and extension for comparison, and the FLC on the data. If the given directory equals the entry filename and extension, it returns the FLC and prints the filename and extension, type, size and FLC (printf(%s %11s %13d %9d", fullName, type, size, FLC)). The FLC can be acquired by doing some simple bit shifting of two bytes in the buffer:

- (((int) sect[marker + 27]) « 8) & 0x0000ff00
  ((int) sect[marker + 26]) & 0x000000ff

int checkDirectory(char *directory, int cluster):
The `checkDirectory(char *directory, int cluster)` function for `ls` takes in the directory the program is searching for and the current cluster it should be in. This function then calls `read_cluster(index, sectorBuffer, directory)` which returns an FLC. If the FLC is greater than 0, than the directory was found, if not it returns a -2 and breaks the loop. This process happens for every entry in the given sector.

int search_cluster(int marker, unsigned char* sect, char* directory):
The `search_cluster(int marker, unsigned char* sect, char* directory)` function for `ls` is essentially identical to `read_cluster()` except it searches for the specific filename, not just the FLC. This is needed for absolute pathnames to work properly.

int countDirectories(char directories):**
The `countDirectories(char** directories)` function for `ls` does a simple for loop that counts the number of items in the array provided and returns that number.

char * addSpaces(char* directory):
The `char * addSpaces(char* directory)` function for `ls` takes in a string, likely a directory or filename, and fills in the gaps with spaces so that comparisons between strings are more accurate.

char ** parseInput(char line[], const char *delimiter):
The `parseInput(char line[], const char *delimiter)` function for `ls` takes in a string(char line[]) and a delimiter used to split the string the way the user wants. It returns an array of strings where the string is split where ever the delimiter is found. This is done pretty simply with use of `strtok`

char *trimwhitespace(char *str):
The `trimwhitespace(char *str)` function for `ls` takes in a string, and returns a copy of the given string, just without any spaces. This is the opposite of the `addSpaces(str)` function, used when comparing given string and acquired strings from the buffers.

3.5 Make Directory

Removes a File.
Author: Anthony Taylor

Make Directory or `mkdir` allows the user to enter in a relative or aboslute path of a new directory the wish to create. The input is run through a combination of `parseInput`, `checkDirectory`, and `countDirectories` which have been descirbed elsewhere in this document. After the input is parsed `searchForFile` goes through and find an apporpriate location for the directory in its parent sector. After this the directory is created in the parent sector and a FAT table value is marked for it using `createDirectory`. `fillNewDir` is a part of `createDirectory` and it ensures that the `.` and `..` entries are created and properly formatted and linked.

3.6 pbs

Acronym for print boot sector.
Author: Matthew Fortier

3.6.1 Requirements

- Print the contents of the boot sector
- Any arguments specified must be ignored.
- Bytes per sector = 512
  Sectors per cluster = 1
  Number of FATs = 2
  Number of reserved sectors = 1
  Number of root entries = 224
  Total sector count = 2880
  Sectors per FAT = 9
  Sectors per track = 18
  Number of heads = 2
  Boot signature (in hex) = 0x29
  Volume ID (in hex) = 0x244f429d

Volume label = NO NAME
File system type = FAT12

3.6.2 Files

pbs.c
pbs.h

3.6.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.6.4 Functions

main():
The `main()` function for `pbs` does two things:

- Calls readBootSector();
- Calls printBootSector();

void readBootSector():
The `readBootSector()` function for `pbs` does all of the work for the command. It first reads in the boot sectors using the `read_sector` function. A struct was created for this command so all of the info can be stored easily. The function then reads in to the struct each piece of the required content at the specific index. There is a handful of bit shifting going on to read in the integer correctly but strings are easier because each character is only one byte.

void printBootSector():
The `printBootSector()` function for `pbs` is a mess of printf's that print out each item of the struct in a specif format using % in printf.

3.7 pfe

Acronym for print fat entries.
Author: Matthew Fortier

3.7.1 Requirements

- Print the 12-bit FAT entry values representing logical sectors x to y.
- If x > y, then an error message must be displayed.
- If x < 2, an error message must be displayed.
- pfe 2 8
  Entry 3: 4
  Entry 4: 5
  Entry 5: 6
  Entry 6: FFF
  Entry 7: FFF
  Entry 8: 0

3.7.2 Files

pfe.c
pfe.h

3.7.3 Required Files

shell.h
pbs.h
fatSupport.c
fatSupport.h

3.7.4 Functions

main():
The `main()` function for `pfe` follows a few steps:

- Checks to see if there are too many arguments
- Checks to see if there are too few arguments
- Checks to see if arguments are in range with `checkRange(x,y)`
- Read in all of the fat sectors
- Executes a for loop staring at x and going to less than or equal to y and printing each fat entry.
- printf("Entry %d: %X", i, get_fat_entry(i, sect)).

int checkRange(int x, int y):
The `checkRange(int x, int y)` function for `pfe` that checks to see if x is greater or less than y. If x is greater than y it throws and error and returns false. If x $<$ 2, than it throws an error and returns false. Otherwise it returns true(1). Very simple.

3.8 pwd

Acronym for print working directory.
Author: Matthew Fortier

3.8.1 Requirements

- Print the absolute path to the current working directory.
- If there are any arguments, ignore them.

3.8.2 Files

pwd.c

3.8.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.8.4 Functions

main():
The `main()` function for `pwd` follows a few steps and is the only function for the command:

- Gets the global struct containing the path from the shared memory
- Prints the value.
- printf("Path: %s", global_path->cwd).

3.9 Remove

Removes a File.
Author: Anthony Taylor

Remove or `rm` allows the user to enter in a relative or aboslute path of a file to be deleted. The input is run through a combination of `parseInput`, `checkDirectory`, and `countDirectories` which have been descirbed elsewhere in this document. After the input is parsed `searchForFile` goes through and removes the directory from it's parent sector as well as removing it's entry from the FAT table.

3.10 Remove Directory

Removes a Directory.
Author: Anthony Taylor

Remove Directory or `rmdir` allows the user to enter in a relative or aboslute path to a directory to be deleted. The input is run through a combination of `parseInput`, `checkDirectory`, and `countDirectories` which have been descirbed elsewhere in this document. After the input is parsed `searchForFile` goes through and removes the directory from it's parent sector as well as removing it's entry from the FAT table.

3.11 touch

Creates a file.
Author: Matthew Fortier

3.11.1 Requirements

- If x exists, then print a message indicating that it already exists.
- x may be a relative or absolute pathname.
- If x does not exist, then create a file x in the target directory.

3.11.2 Files

touch.c

3.11.3 Required Files

shell.h
fatSupport.c
fatSupport.h

3.11.4 Functions

main():
The `main()` function for `touch` follows a few steps:

- Check the argument count and return 1 if it is not equal to 2.
- Parses the filename using `parseFilename(argv[1])` into a global array containing the filename at index 0 and the extensions at index 1
- Get the shared memory
- See if the file given exists using `searchForFile(buffer)`.
- If the file exists, print an error and end.
- If the file does not exist, create the file using `createFile(buffer)`.

parseFilename(char* input):
The `parseFilename(char* input)` function for `touch` tokenizes the string given by periods(.) to acquire the filename and the extension. The filename and extension are stored in a global array at positions 0 and 1 respectively, so they can be used later.

char* searchForFile(char* entry):
The `searchForFile(char* entry)` function for `touch` searches through each entry in the given sector, determined by the global current cluster from (`global_path->cluster`), and returns a 1 if the given file was found or a 0 if not.

void createFile(char* entry):
The `createFile(char* entry)` function for `touch` now knows if a file the current directory exists or not. This function will then search through the current directory search for an open entry. If it finds an open entry, it sets the filename to the given name, and sets the size to 0. It then searches the following cluster looking for an unused cluster, and then sets the FLC to that cluster. The function then writes the sector, and writes the updated fat entry.