# FAT12 Project
# Design Report
# Team 6
# 2/2/05

## util.c (supporting file util.h)

util.c contains a number of functions that are common across all or most of the utilities we implement or interact at such a low level with the filesystem that it is wiser to keep the related functions together. It is supported by the util.h file, which contains function prototypes and structures so that functions from util.c can be easily included in other code documents.

`void init_util()`

**Arguments:** None
**Returns:** None

1. Attaches the util functions to the shared memory that contains the floppy disk file stream as well as current working directory information
2. Kills the process if an error occurs

`void kill_util()`

**Arguments:** None
**Returns:** None

1. Detaches the util functions from shared memory

`int cleanup_path(char* path, char* newpath)`

**Arguments:** char* path – String containing path with .'s and ..'s to remove
**Returns:** char* newpath – String with all .s and ..s resolved to the actual path
Integer error code

1. Given an absolute path name that contains "." and ".." entries, resolves all entries to actual directory names and returns the "cleaned" path in buffer.

`void get_abs_path(char* buffer, char* path)`

**Arguments:** char* path – String representation of path
**Returns:** char* buffer – Fully expanded absolute path

1. If given a relative path name, prepends the working directory to the relative path and returns it in buffer
2. If given an absolute path name, returns path in buffer

`int parse_path (char* path, int* FLC, int* index)`

**Arguments:** char* path – String containing an absolute or relative pathname
**Returns:** 1 on success, integer error code on error
int* FLC – The first logical cluster of the directory entry containing the file specified
int* index – The index within the FLC that contains info about the file specified

1. Calls get_abs_path() with the pathname
2. Breaks up the path specified using strtok()
3. Iterates through directory structure using the root directory as a starting point until it either reaches the final entry or is unable to continue any further with given path.
4. If the function was able to make it to the final entry, it returns the FLC of the directory entry containing the final entry, and the index within that FLC that contains the entry.
5. If the function was not able to make it to the final entry, it returns an integer error.

`int check_filename (char* filename)`

**Arguments:** char* filename – A FAT12 filename
**Returns:** 1 if filename is FAT12 legal, 0 otherwise

1. Checks if *filename* adhears to the FAT12 filename rules and returns an integer representing the result.

`struct fileinfo get_file_info (int FLC, int index)`

**Arguments:** int FLC – The FLC of the directory entry containing the file
int index – Index within the FLC of the file
**Returns:** Returns a fileinfo structure that contains all the information contained in the directory structure about the given file.

1. Reads the 32-byte file info from the directory structure formed by *FLC* and *index* and places all the information parsed into a *fileinfo* structure.

`int get_file_type (struct fileinfo info)`

**Arguments:** struct fileinfo info – A *fileinfo* structure to read a file type from
**Returns:** Integer representation of the file type (subdirectory/file)

1. Checks the Attribtes field of *info* and checks bit 4 to see if the file is a subdirectory or a file
2. Returns 1 if a file, 2 if a subdirectory, and -1 if an undefined type

`int get_dir_list (struct fileinfo* list, int max, int FLC)`

**Arguments:** struct fileinfo* list – Allocated space for returning a list of files
int max – The maximum number of file entires *list* can hold
int FLC – The FLC of the directory to list
**Returns:** The total number of entries in the directory

1. Starting at the FLC given by *FLC*, iterates through the entire directory and counts the total number of entries. Follows the FAT table if the directory extends beyond one block.
2. If at any time the total number of entries read into *list* is equal to the maximum number of entires allowed by *max*, the function stops reading entries into *list*. However, the function will continue to read until the end of the directory and return a final count of the number of entries in the directory.

`void set_wd (char* path, int FLC)`

**Arguments:** char* path – String representation of current path
**Returns:** int FLC – The FLC of the current path's directory entry

1. Passes the pathname through cleaup_path()
2. Updates the shared memory space containing the current working directory with the new directory represented by *path* and *FLC*

`void get_wd (char* path)`

**Arguments:** none
**Returns:** char* path – A string representation of the current working dir

1. Returns the working directory

`void get_free_space (long* blocks, int* block_size)`

**Arguments:** none
**Returns:** long* blocks – The number of blocks free on the device
int* block_size – The size in kB of each block

1. Starting at the beginning of the FAT, counts the number of free blocks in the FAT table. Returns the number of blocks along with the size of each block.

`int get_free_block (void)`

**Arguments:** none
**Returns:** FLC of a free block in data storage memory, or error if disk full

1. Consults the FAT table, finds a free block, and returns the FLC of one of these blocks.

`void allocate_block (int free_FLC, struct fileinfo info, int root_FLC, int* FLC, int* index)`

**Arguments:** int free_FLC – Free block FLC obtained from get_free_block()
struct fileinfo info – Information to write for the new file
int root_FLC – The FLC of the directory to place the file in
**Returns:** int* FLC – The FLC of the directory where the file was placed
int* index– The index in the directory entry where file was placed

1. Looks at directory specified by root_FLC. If a free directory entry exists at that FLC, places fileinfo *info* into an index in that directory. If no free entries exist, checks the FAT table and goes to the next directory block. If no free directory blocks currently exist, creates a new block, links it in the FAT, and places the fileinfo in that directory block.
2. Places the final FLC and index where the file ended up into *FLC* and *index*

`void fileinfo_to_dentry(struct filename info, char* buffer)`

**Arguments:** struct filename info – Fileinfo structure to convert to 32 byte directory entry
char* buffer – 32 byte buffer space to return directory entry in
**Returns:** None

1. Reads the fileinfo struct *info* and performs the necessary modifications to the data fields to convert the structure into the 32-byte format of a directory entry.

`int unlink_entry (int FLC, int index)`

**Arguments:** int FLC – The FLC that contains the entry to unlink
int index – Index within the FLC of the entry to unlink
**Returns:** 1 on successful exit, negative error code otherwise

1. Calls get_file_info() to obtain the fileinfo structure for the entry.
2. Calls get_file_type() to determine if the entry is a file or a subdirectory.
3. If filetype is a subdirectory:
   (a) Checks to make sure that the directory is empty except for "." and "..". Returns an error otherwise.
   (b) Updates the directory entry to indicate that the entry is free.
4. If filetype is a file:
   (a) Updates the directory entry to indicate that the entry is free.
5. Checks to see if all entries after the removed entry are also free. If so, updates removed entry to indicate all following entries are free.
6. Checks to see if all entries in the block are free. If so, flags the block for removal from the FAT table.
7. Removes all blocks associated with the removed file from the FAT table. If a directory block is also flagged for removal, removes it from the FAT table and updates any entries that point to it or are pointed to.

`int open_file(struct fileinfo info)`

**Arguments:** struct fileinfo info – File to open for reading
**Returns:** The number of bytes read

1. Checks to see if a file is already open (our system only allows one open file at a time for simplcity)
2. If no other files are open, looks up the FLC in *info* and marks the location as the open file, and sets the seek inside that file to 0.
3. Sets flags to indicate no other files can be opened until the current file is closed

`int read_file (char* buffer, int max)`

**Arguments:** char* buffer – Character buffer to read data into
int max – Maximum number of bytes to read into *buffer*
**Returns:** The number of bytes read

1. Seeks to the location set in the previous read/set by open_file()
2. Reads bytes from the clusters by following the FAT table and loading the values into the memory pointed to by *buffer*.
3. Continues to read the file until an End of File (EOF) is encountered, or the FAT indicates it has reached the end of the file.
4. Sets the file seek where it stopped reading
5. Returns the total number of bytes read.

`void close_file()`

**Arguments:** None
**Returns:** None

1. If a file is currently open, close_file() closes it. Otherwise it does nothing.

---

## util.h

### the fileinfo structure

- Members of this structure contain all the information contained in the directory entry for a file/subdirectory

### the shared_info structure

- Members of this structure contain the stream for the current floppy image as well as a string representation of the current working directory and the FLC of the current working directory. One of these structures is allocated and placed in shared memory when the shell initializes.

---

## ls.c

**Arguments:** or 0, directory to list contents of
**Returns:** 1 on successful exit, negative error code otherwise

1. If more than 1 argument is specified, returns with an error.
2. If no arguments are specified calls parse_path() with the current working directory as the path. Then get_file_info() is used to obtain the fileinfo structure for the present directory.
3. If there is an argument, parse_path() is called with the pathname to obtain the FLC and index of the file/subdirectory specified; then get_file_info() is called.
4. Calls get_file_type() with the fileinfo structure obtained before to determine if the path specified a file or a subdirectory
   (a) If filetype indicates a directory, calls printDirectoryListing to obtain a list of all the files in the directory specified
   (b) If filetype indicates a file, calls printEntry to display only the file specified
5. Parses and outputs result of get_dir_list().

### void printUsage ()

**Arguments:** none
**Returns:** nothing

1. Prints the usage information for ls, including what options the command can take.

### void printHeader ()

**Arguments:** none
**Returns:** nothing

1. Prints the first line that ls displays, which are column headers such as “File Size” and “FLC”.

```
int printEntry (struct fileinfo *info)
```

**Arguments:** struct fileinfo *info – file whose information is to be displayed
**Returns:** 0 on success, 1 on failure

1. Constructs the file's full name from the filename and extension in the variable info by calling concatFileName.
2. Prints a single line of information, including the file's name, size, and FLC.

```
void printDirectoryListing (struct fileinfo *directoryInfo,
     int FLC, int index)
```

**Arguments:** struct fileinfo *directoryInfo – directory whose information is to be displayed
int FLC – the first logical cluster of the parent of the directory to be displayed
int index – the index within the FLC of this directory
**Returns:** nothing

1. Uses get_dir_list() to create a list of the files and directories within the current directory.
2. For each file or subdirectory, gets its information using get_file_info() and sends it to printEntry.

```
char * concatFileName (char *fileName, char *extension)
```

**Arguments:** char *fileName – the eight-character filename that will form the initial portion of the new filename.
char *extension – the three character extension that will form the second portion of the new filename.
**Returns:** a pointer to the new filename

1. Removes all spaces that may be padding the fileName.
2. If there is an extension (since not all files and directories have extensions), add a period and the extension to the unpadded fileName.
3. Return this new string.

---

**cd.c**

**Arguments:** Directory to change into
**Returns:** 1 on successful exit, negative error code otherwise

1. If more than 1 argument is specified, returns with an error.
2. If no arguments are specified, calls parse_path() with root (/) specified as the path, otherwise calls parse_path() with the pathname to obtain the FLC and index of the file/subdirectory specified.
3. Calls get_file_info() to obtain the fileinfo structure for the given pathname.
4. Calls get_file_type() with the fileinfo structure obtained to determine if path is a file or subdirectory.
   (a) If path specifies a file, returns with an errors
   (b) If path specifies a directory, calls set_wd() with the new pathname and FLC to set the new working directory

---

**pwd.c**

**Arguments:** none
**Returns:** 1 on successful exit, negative error code otherwise

1. Calls the get_wd() function, parses the return value, and prints the current working directory to the screen.
2. If there are any arguments, they are ignored.

---

**cat.c**

**Arguments:** File to output
**Returns:** 1 on successful exit, negative error code otherwise

1. If not called with exactly one argument, returns with an error
2. Input can either be an absolute or relative path.
3. If file does not exist, print an error.
4. Calls parse_path() with the pathname to obtain the FLC and index of the file/subdirectory specified.
5. Calls get_file_info() to obtain the fileinfo structure for the given pathname.
6. Calls get_file_type() to determine if the path specified a file or a subdirectory.
   (a) If path specified a directory, output an error and exit
   (b) If path specified a file, call read_file() to output the file to the screen

---

**df.c**

**Arguments:** none
**Returns:** 1 on successful exit, negative error code otherwise

1. Calls the get_free_space() function to determine the amount of free space on the disk
2. Parses return values and outputs to the screen

---

**touch.c**

**Arguments:** File to create
**Returns:** 1 on successful exit, negative error code otherwise

1. If not called with exactly one argument, returns with an error
2. Calls parse_path() with the pathname to obtain the FLC and index of the file/subdirectory specified.
   (a) If parse_path() returns that the file exists, print a message that says the file exists (extra idea – make touch update the last access time like the UNIX version)
   (b) If parse_path() returns that the file does not exist:
   (c) Call parse_path() with the file name chopped off to make sure the directory exists.
      i. Calls get_free_block() to allocate a free block of space for the new file
      ii. Call allocate_block() to place the entry for the file in the directory entry and in the FAT table

---

**mkdir.c**

**Arguments:** Directory name to create
**Returns:** 1 on successful exit, negative error code otherwise

1. If not called with exactly one argument, returns with an error
2. Given path may be relative or absolute.
3. If parent directory of new directory does not exist, return an error.
4. If new directory already exists, return an error.
5. Calls parse_path() with the current working directory to get the directory entry that the new directory will be added to. This searches in the parent directory's data sectors. If no room there, looks for a free data sector.
6. Calls allocate_block() to create the new directory entry
7. Calls allocate_block() two more times using the new directory to create the "." and ".." entries.

---

**rm.c**

**Arguments:** File entry to remove
**Returns:** 1 on successful exit, negative error code otherwise

1. If not called with exactly one argument, returns with an error
2. Calls parse_path() with the specified path to get the FLC and the index of the file/subdirectory specified
3. Calls get_file_info() to get the fileinfo structure
4. Calls get_file_type() to determine if path specified is a file or a directory
   (a) If path is a directory, exits with an error
   (b) If path is a file calls unlink_entry() to remove the file

`void printUsage ()`

**Arguments:** none
**Returns:** nothing

1. Prints the usage information for rm, including what options the command can take.

---

**rmdir.c**

**Arguments:** Directory to remove
**Returns:** 1 on successful exit, negative error code otherwise

1. If not called with exactly one argument, returns with an error.
2. Calls parse_path() with the specified path to get the FLC and the index of the file/subdirectory specified. If parse path indicates the file/directory does not exist, returns an error.
3. Calls get_file_info() to get the fileinfo structure for the directory/file.
4. Calls get_file_type() to determine if path specified is a file or a directory
   (a) If path is a file, returns with an error.
   (b) If file is a directory, calls get_dir_list() to determine if directory is empty except for "." and "..". If it is, unlink_entry() is called on the directory entry. Otherwise returns an error.

`void printUsage ()`

**Arguments:** none
**Returns:** nothing

1. Prints the usage information for rmdir, including what options the command can take.

---

**shell.c**

**Arguments:** Device/file to use as filesystem
**Returns:** 1 on successful exit, negative error code otherwise

1. Reads the argument and opens the device/file to use
2. Creates shared memory space to store the current working directory.
3. Presents the "floppy>" prompt and waits for user input. If user presses enter without any input, a new "floppy>" prompt is displayed.
4. Parses valid user input into program name and flags using the strtok() function.
5. Attempts to fork() and run the program specified by the user. If there is an error, reports the error to the console and prints the prompt. Otherwise watches for the program to exit using waitpid().
6. If the program exits successfully, waits until the command returns and then redisplays the prompt. If any error values are returned, prints the corresponding error message.

---

**Division of Labor**

| | |
|---|---|
| Jenn<br>pwd.c<br>cat.c<br>mkdir.c | Kellan<br>shell.c/cd<br>touch.c |
| Lissa<br>ls.c<br>rm.c<br>rmdir.c<br>df.c | Nick<br>util.c<br>util.h |