# COMP 310

**Winter 2024**

# Assignment 3

Posted: Monday, March 11

Due: Wednesday, April $3^{rd}$, 11:59 p.m.

Similar to Assignment 2, this assignment is heavyweight.
Plan your time wisely. Don't hesitate to ask questions on Ed if you get stuck.

There will not be any extension for this assignment
except for documented cases of medical/personal issues or SAA considerations.

**Please read the entire PDF before starting. It is very important that you follow the directions as closely as possible.** The directions, while perhaps tedious, are designed to make it as easy as possible for us to run and grade your code.

**This assignment will count for 20% of your final grade. To get full marks, you must follow all directions below:**

- You must write your code in the C programming language. We use C in this class because most practical contemporary OS kernels are written in C/C++ (e.g., Mac, Linux, Windows and others).
- Make sure that all filenames and any specified functions/data structures are **spelled exactly** as described in this document; otherwise, the code will likely receive a zero. You are free to modify function signatures.
- The output from your program must **exactly match** the output that we specify, except for spaces.
- Make sure that your code **compiles**. Code that does not compile will likely result in a zero.
- Write your name and student ID in a comment at the top of all files modified from the starter code.
- Your code must follow our style guidelines, which are listed on the next page. **Up to 30% can be removed for code that does not comply to our style guidelines.**

**Hints & tips**

- **Start early.** Programming projects always take more time than you estimate!
- You are free to include any header files from the C standard library that are available for including by default in the given Docker image.
- Do not wait until the last minute to submit your code. **Submit early and often**—a good rule of thumb is to submit every time you finish writing and testing a function.
- Write your code **incrementally**. Don't try to write everything at once. That never works well. Start off with something small and make sure that it works, then add to it gradually, making sure that it works every step of the way.

- Seek help when you get stuck! Check our discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to a TA during office hours if you are having difficulties with programming. Go to the instructor's office hours if you need extra help with understanding a part of the course material.
  - Note that small amounts of code can be posted on the discussion board as *private* posts, which only the instructors and T.A.'s can see. But if more than a few lines of code are in question, then it is better to seek help in person. Often the problem requires a different *approach* such as proper use of the debugger, and the discussion board is a poor medium for this kind of help. (You could also post a single line of code and error message (error traceback) as a public post.)
- If you come to see us in office hours, please do not ask "Here is my program. What's wrong with it?" We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. Reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.
  - However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried in order to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

## Policy on Academic Integrity

See section 5 of our syllabus for our course policies on academic integrity. In short, you must write the assignment on your own (or with a partner, if you have registered them as such). Do not copy someone else's work, nor share your work with someone else. Do not copy from any online sources.

## Late policy

Late assignments will be accepted up to two days late and will be penalized by 10 points out of 100 per day. Note that if you submit one minute late, it is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.

## Style guidelines

As students at the 300 and 400 level, it is expected that you code at the level of a beginner software engineer. Please follow the brief coding style below in all your OS assignments.

- Indentation: All blocks of code must be indented by a minimum of 4 spaces, or with a 1-tab. (Pretend you are writing Python code.)
- Horizontal spacing: Parts of the code that are logically different should be separated by a blank line.
- Comments: Provide useful explanations to the reader of the code. We do not expect you to comment all your code, but we do expect explanations for more complex parts. In other words, do not write comments that just restate the code; here are some situations where a comment may be appropriate:
  - describing the purpose of a function
  - explaining the high-level steps of a function
  - explaining magic numbers
  - describing side-effects of a function
- Names: Name your variables, constants, functions, objects, methods, data structures and files appropriately. The purpose of each should be obvious from the name and can save you from writing a comment.

## Compilation environment

Because of the many different versions of gcc and C library implementations, we will ask you to use a standardized testing environment when compiling and testing your code. We will be using the same environment when grading your code, so it's important that we all use the same one. If your code does not compile in this environment, you will receive a very low mark, and possibly a zero.

We will use **Docker** for our testing environment. Here are the steps you should follow:

1. Install Docker by following the steps at `https://docs.docker.com/get-docker`. Docker is available for Mac, Linux and Windows.

   Note: You do not need to sign up for a Docker account. You can skip that step.

2. Open up a command line and enter the following command:

   ```
   docker pull gcc:13.2
   ```

3. `cd` to the directory containing your assignment code files (*.c, *.h and Makefile). Then enter the following command to run a Docker shell. This command is long, so you may want to alias it:

   ```
   docker run --rm -it --mount type=bind,source=.,target=/code --entrypoint /bin/bash -w /code gcc:13.2
   ```

4. Once you are in the Docker shell, you should find yourself in the `/code` directory by default. This directory will contain all the files in the folder you were in before launching the shell. You can now run `make` to compile your code. (Don't forget to pass the `framesize=X` and `varmemsize=Y` arguments.)

   Note: Any changes made to your files here (including the creation of your executable) **will** be reflected in your original folder (on your hard drive). So, be careful not to delete any files from within Docker.

You are also free to modify the Docker image by adding additional packages (such as vim) to it, as long as these packages do not change the compilation environment (e.g., by adding a new compiler or new libraries). You can add packages by doing the following:

1. Create a file called `Dockerfile` with the following contents. This one installs `vim` and `valgrind`:

   ```
   FROM gcc:13.2
   RUN apt-get update && apt-get install -y --no-install-recommends vim valgrind
   ```

2. In the command line, cd to the folder containing your Dockerfile and enter the following command:

   ```
   docker build . -t 310 -f Dockerfile
   ```

3. Then run the same command as in step 3 above (first cd'ing to your code folder), but replace gcc:13.2 by 310 at the end of the command.

## Submitting your code and running the public tests

For each question, we provide **examples** of how your code should behave. All examples are given as if you were to run the commands while inside the shell.

During grading, we will run these examples and verify that your code outputs the same as given in the examples. Part of your assignment mark will be determined by the results of these tests. We will also run additional, private tests on your code. Therefore, it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples. **You are free to handle corner cases as you wish, unless we explicitly mention how to handle such a case.**

To run your code on our public tests, as well as to submit your code for grading, please check post #72 (Submission instructions) on Ed. For submission of late assignments (within the two-day late period), check post #274.

In this assignment we will write functions that operate on a filesystem.

We provide starter code for this assignment which contains our solution for Assignment 2 as well as our filesystem implementation. We begin by describing the properties of our filesystem, the new shell commands, the list of new files included in the starter code, an example of how to use the filesystem and a note about using your own A2 solution code. We then discuss your tasks for the assignment.

## Filesystem architecture

Our filesystem has a similar structure to the simple filesystem discussed in class. So if you attended the lectures and did the practice filesystem problems, you are already in a good position to understand some part of our implementation. If not, **you are strongly suggested to catch up on those lectures before beginning this assignment.**

First, we will store our filesystem inside a single file on our computer. We will call this file the hard drive file or **disk image**. A regular hard drive has sectors that are read from and written to; our hard drive file will also have 'sectors' (of size 512 bytes) which are really just bytes in the file. Sector 0 will be the first 512 bytes of the file; sector 1 will be from bytes 512 to 1024, and so on. In this way, we can approximate a real hard drive by using this one file.

Our shell will be able to use only one disk image at a time. The file will be specified as the first argument to the `myshell` executable. For instance, to run the shell with `blank.dsk`, which is a disk image of size 1MB provided with the starter code, you can run

```
./myshell blank.dsk
```

A disk image by itself is just an empty file with some metadata at the beginning (sector 0) relating to partitions. In order for our shell to use it, we must first **format** it for our filesystem. To do this, we must pass the `-f` flag to `myshell`, as follows:

```
./myshell blank.dsk -f
```

Note that formatting will destroy any metadata already existing in the disk image. So if your disk image already has some files on it and you then format it by passing the `-f` flag, your files will no longer be accessible.

Now for the basics of the filesystem. The filesystem stores a file by creating an **inode** for it in a free sector, and then stores the data of the file in other sectors (called the **data blocks**); the inode for a file will store the sector numbers for the file's data blocks.

In particular, the inode will directly store 123 sector numbers for data blocks (recall that each sector is 512 bytes, so that means that $123 * 512 = 62,976$ bytes can be addressed by these direct block pointers). If a file needs more than that, the filesystem will create (in a new sector) an indirect block which has space for another 128 sector numbers (pointing to data blocks). If a file still needs more, it will create (in a new sector) a double indirect block, which will hold 128 sector numbers that each point to indirect blocks (each allocated as needed). Note that this scheme is very similar to the one discussed in class. See Figure 1 (next page) for an illustration.

When making a new file, space on disk must be found for the inode and for the data blocks (and possibly the indirect block(s)). Finding free space is accomplished by checking the **free space bitmap**, which itself is written to disk in a reserved sector near the start of the disk. The data blocks for a file do not have to be all stored in one contiguous chunk, so a file's data blocks may be spread across the disk.

The filesystem has both on-disk and in-memory data structures. For example, the on-disk inode stores the block sectors. There is also an in-memory inode, into which the on-disk inode is loaded when a file is accessed; the in-memory inode also contains some additional information.

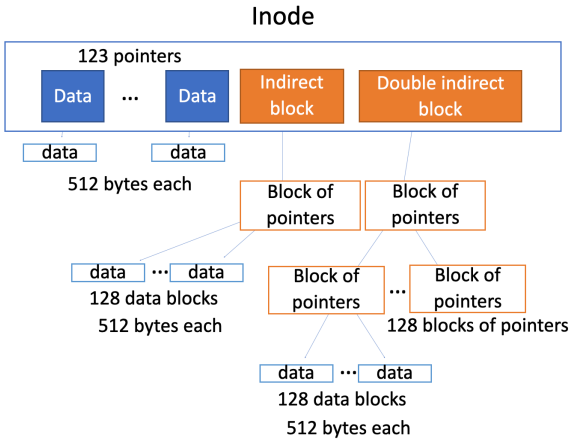


Figure 1: On-disk inode struct (defined in `inode.h`)

Also as discussed in class, the filesystem has a **write-behind cache** for both inodes and data blocks. When a request is made to read a block, the cache is checked first before we go to the hard disk. If the block is not in the cache, then we read it from the hard disk (disk image) and put it in the cache. Similarly, if we write a block, we put it in the cache. The cache has a limited size, so when it is full, we evict another block (writing to disk if needed), and on the close of the shell we flush all blocks to disk.

Finally, although our filesystem supports directories, we will assume that all files are stored in the root directory.

## Shell commands

To make our shell work with this new filesystem, we have added shell commands which we describe below. Note that some of these commands had already existed, but they operated on your real filesystem and not a disk image and its filesystem as described above; we have replaced those implementations with versions that do operate on the disk image. (Note that we've also renamed commands to remove their `my_` prefix.)

- `ls`: Lists all files (in the root directory).
- `cat fname`: Displays the contents of the specified file.
- `rm fname`: Removes the specified file.
- `create fname size`: Creates a (blank) file with the given filename and initial size.
- `write fname data1 ... ...`: Writes data into the specified file. (Spaces are supported; any reasonable number of tokens can be given.) Note that any write will move forward the current offset of the file, so subsequent reads/writes will take place after the written data.
- `read fname bytes`: Displays the given number of bytes of the specified file. Note that any read will move forward the current offset of the file, so subsequent reads/writes will take place after the written data.
- `size fname`: Displays the number of bytes taken up in data blocks by the given file.
- `seek fname offset`: Updates the given file's offset to the given amount, to be used for subsequent reads/writes.
- `freespace`: Prints out the number of free sectors on the disk.

(You will implement some further commands.)

## New files in starter code

The following files (inside the `fs` folder) make up our filesystem implementation. As part of your tasks for this assignment, you will need to understand the various `structs` and call on the various functions exposed in these files. Please take a moment to look through these files as you read through this list so that you get a bit of an idea of what is available to you. We point out some useful functions below, but when to use them and what other functions to use are left for you to understand and find.

Note that we do not expect you to modify these files. You can if you like (and you may need to for some of the optional parts), but if you do so, please be careful, because if your filesystem implementation starts to diverge from ours, it may no longer pass the tests used for grading your work. (Of course, it is ok to add new functions that only your code calls.)

- `cache.c`/`cache.h`: Implementation of the write-behind cache. To read or write blocks, you should use the cache functions `buffer_cache_read` and `buffer_cache_write`.
- `directory.c`/`directory.h`: directory operations; `dir_readdir_inode` can be called successively to return each filename/inode in the directory. Note that we will assume that all files are in the root directory.
- `file.c`/`file.h`: Operations on the `file` struct. This struct contains an (in-memory) inode and the current file position (for reading/writing). Also contains the open file table.
- `filesys.c`/`filesys.h`: Operations at the filesystem level: creating a new file, removing a file, opening a file, and others, given the filename.
- `free-map.c`/`free-map.h`: Operations on the free space bitmap.
- `fsutil.c`/`fsutil.h`: Operations at the shell level: code for the shell commands ls, cat, rm, create, read, write, size, seek, freespace. Understanding how these functions work may be useful to you as you may want to replicate some of their behaviours.
- `inode.c`/`inode.h`: Operations on the inode `struct`, including creating, opening, closing, removing, and reading/writing into the data blocks of the inode. Important stuff!

These files are also present, however, they are slightly less important.

- `bitmap.c`/`bitmap.h`: A general purpose implementation of a bitmap. Used for the free map.
- `block.c`/`block.h`: Block level operations: reading and writing to blocks (sectors). (You will want to use the cache functions instead.)
- `debug.c`/`debug.h`: some miscellaneous debugging functions; some may be useful to you.
- `ide.c`/`ide.h`: 'controller' code to handle the ATA drive at the physical level (in reality just handling the disk image).
- `list.c`/`list.h`: a general-purpose linked-list implementation. Some functions could be useful if you need to modify lists.
- `off_t.h`/`round.h`: misc definitions.
- `partition.c`/`partition.h`: code to deal with partitions on a disk image. (We will assume there is only one partition.)

## Example of filesystem use

Here is sample output from running `myshell` with the provided blank disk image `blank.dsk` (after passing the flag `-f` to format it).

```
$ ls
Files in the root directory:
End of listing.
$ create ysub.txt 128
$ ls
Files in the root directory:
ysub.txt
End of listing.
$ write ysub.txt In the town where I was born Lived a man who sailed to sea
$ cat ysub.txt
Printing ysub.txt to the console...
00000000  49 6e 20 74 68 65 20 74-6f 77 6e 20 77 68 65 72 |In the town wher|
00000010  65 20 49 20 77 61 73 20-62 6f 72 6e 20 4c 69 76 |e I was born Liv|
00000020  65 64 20 61 20 6d 61 6e-20 77 68 6f 20 73 61 69 |ed a man who sai|
00000030  6c 65 64 20 74 6f 20 73-65 61 00 00 00 00 00 00 |led to sea......|
00000040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |................|
00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |................|
00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |................|
00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |................|
```

We encourage you to compile the starter code now and play around a bit with the filesystem shell commands so that you can get some experience with them before continuing to read this PDF. Part of this assignment involves understanding the existing code given to you so that you can use it or parts of it as needed.

As before, to compile your code, you will run `make clean; make myshell framesize=X varmemsize=Y` in the Docker container as detailed earlier in this PDF. Also like the last assignment, to run your shell, you can either run in interactive mode (by typing `./myshell diskname.dsk` at the prompt), or in batch mode, by making a file of commands and then typing `./myshell diskname.dsk < commands.txt`. We will be using the latter mode to test your code.

## Using your own Assignment 2 solution code

You are free to use your current Assignment 2 solution, in which case you will have to make the following modifications:

- copy the provided `fs` folder into your code folder.
- merge the provided `Makefile` with your own (if you made any changes to your Makefile); otherwise, simply replace your current one with the new version.
- merge the provided `shell.c` file with your own; it is likely you did not make any changes to that file, so you can simply replace your current one with the new version.
- merge the provided `interpreter.c` file with your own. This file contains substantial changes, including: new headers, error messages, and a bunch of new `else if`'s in `interpreter()` for the new shell commands.

(To see the changes in these files in detail, you can use a file diff program like FileMerge on the Mac, which comes with Xcode. Or you could put them in your Git repo and use Github Desktop to view the change in the files before committing.)

## Your tasks

Your tasks for this assignment are as follows:

1. Implement some basic file utilities.
2. Implement a simple defragmentation algorithm and fragmentation statistic.
3. Implement data recovery and forensics routines.

Details on the required behaviour of your functions follow. Note that we will make some recommendations as to how to proceed, but you have full freedom for your implementation, as long as it complies with all instructions and edge cases.

For each sub-part below, we also note how (relatively) lengthly it may take you to accomplish, based somewhat on the length of the required solution code (or, new code needed in excess of code written for prior parts) by indicating (Short), (Medium) or (Long). If you find that you are taking a very long time on a short part, then it may be helpful to rethink your implementation and possibly seek advice in office hours/Ed as there could be a much easier way to solve it. Note however that these notes are only our best approximations.

Note that the assignment is designed to be done in the order below. The reason is that parts of the code that you write in earlier parts may be reusable in later parts and, likewise, the understanding that you build up in earlier parts will help you in later parts. That's why we recommend, if working with a partner, to actually work together on the assignment (e.g., pair programming) instead of splitting up the work.

A reminder that we will be having a **lab** on this assignment in the days following its release (date to be announced). You are strongly encouraged to attend the lab or watch the recording afterwards as it will go through some helpful tips for getting started with the assignment.

Finally, note that we provide two sample disk images (each of size 1MB) with this starter code. One is blank (and thus needs to be formatted) and the other has a few files in it already (so do not format that one). You will probably want to make backups of these files so that you can recover from a clean copy in case they get corrupted. There may also be a way to obtain other disk images, but I don't know anything about that.

**Write all your code for this assignment in the file `fsutil2.c`**. As described above, you may also modify other files if you like.

### Part 1: File utilities

- (Short) Part 1-1: Implement the shell commands **`copy_in`** and **`copy_out`**. `copy_in` should be followed by a filename of a file on your real hard drive. It should read in the file, and then store it as a new file into the shell's hard drive with the same name in the root directory. `copy_out` should be followed by a filename of a file in the shell's hard drive. It should create a file on your real hard drive in the current working directory (normally the same directory as the `myshell` executable), with the same name as the given file and with the same contents.

  For `copy_in`, if you are able to only partially copy the file in (because of lack of free space), print out the message `"Warning: could only write %d out of %ld bytes (reached end of file"` where `%d` is the number of bytes you could write and `%ld` is the total file size of the file to be copied in.

  Both functions should return the appropriate error flags (defined in `interpreter.h`) if there is an issue with the file. E.g., if the file does not exist, file could not be read, created, written, etc.

```
/code# echo abcdef > test.txt
/code# ./myshell < blank.dsk -f
Shell v2.0
Frame Store Size = 120; Variable Store Size = 120
hda: 2049 sectors (1 MB)
hda1: 2048 sectors (1 MB)
Formatting file system...done.
Num free sectors: 1974
$ copy_in test.txt
$ ls
Files in the root directory:
test.txt
End of listing.
$ cat test.txt
Printing <test.txt> to the console...
00000000  61 62 63 64 65 66 0a                              |abcdef.         |
Done printing
```

```
/code# ./myshell blank.dsk -f
Shell v2.0
Frame Store Size = 120; Variable Store Size = 120
hda: 2049 sectors (1 MB)
hda1: 2048 sectors (1 MB)
Formatting file system...done.
Num free sectors: 1974
$ create myfile.txt 10
$ write myfile.txt testdata
$ copy_out myfile.txt
$ exit
Bye!
/code# cat myfile.txt
testdata
```

- (Short) Part 1-2: Implement a file finding function `find_file`. The function should take one argument - a pattern to search for. It should go through all files in the root directory and print out the name of each file whose contents contain the given pattern.

```
/code# ./myshell tswift.dsk
Shell v2.0
Frame Store Size = 120; Variable Store Size = 120
hda: 2049 sectors (1 MB)
hda1: 2048 sectors (1 MB)
Num free sectors: 1771
$ ff cat
vigilante-sh*t.txt
karma.txt
welcome-to-new-york.txt
style.txt
bad-blood.txt
all-you-had-to-do-was-stay.txt
22.txt
miss-americana.txt
i-wish-you-would.txt
```

### Part 2: Fragmentation

- (Medium) Part 2-1: Write the function `fragmentation_degree` to implement a simple filesystem fragmentation metric. The function should print out the degree of fragmentation of the filesystem. The degree is calculated by dividing the number of fragmented files by the number of fragmentable files.

  A fragmented file is a file which contains at least two consecutive data blocks which are in sectors that are more than three away from each other. For example, a file with three data blocks in sectors 3, 4, 5, or 3, 4, 7, is not fragmented. But a file that has two data blocks in sectors 4 and 10 (and not in 5,6,7,8,9) is fragmented.

  A fragmentable file is any file that has more than one data block.

- (Long) Part 2-2: Write the function `defragment`. This function should reduce the number of fragmented files to zero without any data loss.

  This function can be partly implemented by reading all files into memory (not necessarily the shell memory, just a buffer); we will assume that the memory is large enough to store the contents of all files on the disk. Note that real defragmentation algorithms cannot make this assumption and must try to defragment in-place.

### Part 3: Data recovery & forensics

Write your code for this part in the function `recover`. You may (as always) use helper functions.

When the shell command `recover %d` is executed, the function `recover` will be executed. The integer `%d` will be passed as argument and serve as a flag. Depending on the flag, you will execute a different routine, listed below. Each routine involves recovering files on the hard drive.

- (Short) Part 3-1 (flag 0): Recovering deleted files. Your code should scan all free sectors on the drive and check if any contain inodes which were previously deleted from the root directory (i.e., through the `rm` command) but still exist on disk. If it finds any, it should restore the inodes so that the files can be accessed normally via the shell. The filename for each recovered file should be `'recovered0-%d'` where `%d` is the sector number containing the deleted inode. All recovered files of this type should be stored in the root directory of the disk image. (We will assume that the data blocks pointed to by the inode will not have been overwritten.)

- (Short) Part 3-2 (flag 1): Recovering individual, non-zeroed out data blocks (since it may be that a deleted file's inode has been overwritten, but some or all of its data blocks may remain intact). If you find any non-zero data blocks (starting at sector 4), save the contents of the sector outside of the shell (on your real filesystem) in a file called `'recovered1-%d.txt'` where `%s` is the sector in which you found the data.

- (Long) Part 3-3 (flag 2): Finding data hidden past the end of a current file. A file can be split over multiple data blocks, but it may not fully fill its final data block. For example, a file might be created with size 768, which will fit into two data blocks (since blocks/sectors are 512 bytes each). However, the last 256 bytes of the second data block will go unused, and the contents of this part of the data block will not be shown using `cat`. It is possible that data is hidden in this area. Your function should go through all files and check for this hidden data, if any. If any such (non-zeroed) data is found, save it outside of the shell (on your real filesystem) in a file called `'recovered2-%s.txt'` where `%s` is the name of the file in which you found the data.

Note: your recovered filed must contain only the requested data recovered. If it contains less or more than that, it will not be judged as correct.

### (Optional) Part 4

*The following sub-parts are not worth any marks and will not be graded. However, if you pass the tests for this part, you will be awarded the specified number of* `COMP310COIN`*. Coins will be awarded upon submission if the relevant test(s) are passed.*

Note: For those working with a partner, you will split the coins with your partner. However, your leaderboard score will appear as if you had each obtained the full amount.

500 coins Recover more hidden data from the hard drive file beyond the points mentioned in part 3 above. You will need to implement a routine for flag 3. Store the recovered data in a file `recovered3.txt`. We do not say where the data may be hidden.

500 coins Recover more hidden data from the hard drive file beyond the points mentioned in part 3 above. You will need to implement a routine for flag 4. Store the recovered data in a file `recovered4.txt`. We do not say where the data may be hidden.

500 coins Recover more hidden data from the hard drive file beyond the points mentioned in part 3 above. You will need to implement a routine for flag 5. Store the recovered data in a file `recovered5.txt`. We do not say where the data may be hidden.

500+ coins Extend the capacity of the disk image *without making the disk image file larger*. We provide a 1MB blank disk image with this assignment. Here we ask you to update the filesystem code so that it can store more files than in the initial implementation. Some possibilities here are compression or sharing sectors that multiple files have in common; or looking for unused parts of the disk image in the current implementation and finding out why they are unused (you can check where empty space may be by using a hex-editor, e.g., http://hexfiend.com for Mac, on a disk image filled to capacity). The more amount of data that can be stored, the more coins you will receive (in fixed increments).

1000 coins Implement the shell commands `cd` and `mkdir` which should operate on the shell's hard drive (not your real hard drive). The other file commands (`create`, `read`, `write`, etc.) should all be modified to look for the file in the current working directory instead of the root directory.

### README

Finally, add a **`README.md`** file into the same folder as your code files. In this file, state your name(s) and McGill ID(s), any comments you would like the TA to see, and whether you have implemented any optional parts, and if so, which parts, so that the TA will know to check for this.


HEY EVERY !!
IT'S ME!!


EV3RY BUDDY 'S FAVORITE
[[Number 1 Rated Salesman1997]]


[Come One Come ALl] TO OUR NEWEST
[[100% Legal Storefront Site]]!!


HURRY UP AND BUY
OUR DELICIOUS [[Disk Images]]!!!


[[100% Virus]] GUARANTEE!


OPENING SOON!
IN THE [[??? Floor]] OF THE
[[Trottier Building]]!!

Spamton sprite from the game DELTARUNE by Toby Fox