CMSC 216 Project #3 Spring 2013

Date due: Monday, April 1 (no fooling!), 10:00:00 p.m.

1 Administrative

The TAs' "extra" office hours for this project will run from Thursday, March 28, through Wednesday, April 3 (weekdays only, of course). During that time they will hold the higher–level office hours shown on the office hours handout.

As before, you will have the opportunity to earn extra credit on this project– which will be more difficult than for the previous project– but as before you may also lose points for having an excessive number of submissions. Carefully read Appendix C for full details. On the other hand, if you make too many submissions for this project, you may again lose credit.

Be sure to frequently check the Projects page on ELMS, where this project is posted, to see if any clarifications or corrections to this assignment have been made during the time that it is assigned.

Note: you have seen above that you have a fairly long time to do this project. That is because it is a big step up in difficulty compared to the preceding ones. Spring break occurs during the time when this project is assigned, and due to the break and exam dates, what essentially would have been two separate projects have been combined into this one (otherwise the first of the two separate projects would have had to have been due around the beginning of Spring Break, which of course is not possible). Of course leaving aside the fact that you may want to do other things over Spring Break besides working on this project, there will be no office hours over the break, and the instructional staff will not be available to answer questions about it or provide debugging assistance via email over the break. Lastly, note that religious holidays will take time away from work for some students. Consequently you are advised to begin immediately in order to be sure you complete the project.

The public tests will be provided during the time that the project is assigned. The project will be set up on the submit server after the public tests are provided. In order that you can start working on the project right away, the project tarfile, with the required header files, is available now. When the public tests are available they will be added to the project tarfile (a note will be put on the Projects page on ELMS to let you know when this is), and you can copy it and extract the files from it again to get the public tests. Before the public tests are posted you may refer to the example in Section 9 for an illustration of what the program should do.

You need to know how to use the `gdb` debugger, and to have used it to throughly debug your code before asking for debugging help in the TAs' office hours. The TAs will not debug code for you that you have not already made every effort to debug yourself first.

The following items are being added to the project style guide posted on ELMS:

- It is considered (very) poor style to use numeric ASCII values in place of character constants (literals). If you write code that looks like what is on the left below, it just forces the reader to look up the ASCII table to see what characters are being referred to. On the other hand, no one needs to look up anything when single–quoted character constants are used instead, so code like what is on the right below should always be written instead (note that the ASCII value of `'w'` is 119, and `'z'` is 122).

```
if (ch >= 119 && ch <= 122)
  printf("Goofus\n");
```

```
if (ch >= 'w' && ch <= 'z')
  printf("Gallant\n");
```

- In lecture it was said that `goto` should not be used, and it was also said that `strtok()` library function should not be used. For completeness these are being added to the project style guide.

- It is not good style to write code that duplicates the effects of library functions that are already available. You are not expected to have every single library function memorized, and of course are not responsible for ones we have not covered yet. However, writing code that does the same thing as multiple library functions may result in losing credit. This is sometimes a problem when someone has insufficient familiarity in particular with the string library functions, so be sure to keep them in mind, and use them when applicable, instead of reinventing the wheel.

2 Introduction

In this project you are to write a simulation of some of the functionality of the UNIX filesystem and common commands. Your program will allow simulated files and directories to be created, and permit navigating around the directory structure by changing the current location in the filesystem. The purpose of the project is to use linked structures and memory management in C. Note that you **must** use linked data structures in implementing the project.

For this project you will create your own data structure for storing the components of a filesystem. We are giving you a header file that contains the prototypes of the required functions, but has no type definitions. Although the exact data structure is up to you, you **must**, as mentioned above, use some kind of linked data structure to store a filesystem. You could certainly get all of the functions to work correctly for many cases without using linked structures (for example, by using arrays instead), but since the entire purpose of the project is to use linked data structures and memory management in C, you **will not receive credit** for the project in that case.

You will need to write a makefile for this project. To test your makefile and ensure that it works correctly, write it **as soon as you start coding**, and use it **every time you compile your code**. Do not wait and write it after you finish coding, because then you will not be exercising it as you work, and consequently may not find errors in it.

You are encouraged to think about how to design the project for a day or two, and bounce your ideas off of the TAs in office hours, prior to starting to code.

3 Filesystem components

Your program will simulate the UNIX filesystem and allow both files and directories to be created. Both files and directories have names.

- A file is an entity that, for this project, contains a sequence of zero or more lines of text, each containing zero or more characters. Files are located in directories.
- A directory may store both subdirectories and files. A directory may contain zero or more subdirectories and zero or more files. A directory may not contain multiple things (files or subdirectories) that have the same name, but a directory may contain files and subdirectories with the same names as files or subdirectories that are located in other directories. For example, a directory named `x` can contain a file or subdirectory `z`, and another directory named `y` can also contain a file or subdirectory `z`. A directory named `z` can even contain an immediate subdirectory named `z`.

Names of files and directories consist of sequences of one or more consecutive characters, with the exception that files and directories may not be named either "`.`" (a single period character), "`..`" (two adjacent period characters), or "`/`" (the forward slash character), and they cannot contain a forward slash character anywhere. These symbols all have special meanings in your simulation, as in real UNIX:

- A single period character refers to the current directory. After your functions are used to create directories, the current location in the filesystem may be changed, and functions that refer to or operate upon the contents of the directory `.` use whatever directory happens to be current at the time they are called.
- Two adjacent period characters refer to the parent of the current directory in a filesystem, meaning the directory immediately above it in the filesystem directory structure.
- The forward slash character refers to the root directory, which always exists as the topmost directory of the entire filesystem (for any filesystem that has been created), and is either the direct parent, or the indirect ancestor, of all other files and directories. The root directory is the only file or directory that has no parent. No other directory in a filesystem may be named "`/`" other than the special root directory, or contain a forward slash character. The forward slash character also is used to separate directory names when paths are printed by one of the functions you will write.

File and directory names can begin with a period character, as in the example "`.banana`", and if so they are not treated any differently from other file and directory names, unlike in real UNIX where an initial period in a file or directory name means the file or directory is hidden. Also, file and directory names may contain spaces or start with or contain characters such as `*`, or other punctuation symbols that have special meanings to the shell in real UNIX, but have no special significance effect for this project.

4 Project files, and functions to be written

The `project3.tgz` tarfile in the `216public/project3` directory contains a header file `filesystem.h` that contains the prototypes of the functions you must write, but as previously noted it contains no type definitions. Instead it includes a header file `filesystem-datastructure.h` that is not provided. You must write this file, which must contain a definition of the type `Filesystem` that the functions operate upon. The tarfile also contains object files `driver.o` and `memory-checking.o` (compiled on the Grace machines), described below.

Three functions (`ls`, `pwd`, and `cat`) usually produce output; the rest only modify their `Filesystem` parameter and return a value. Negative return values are described for some error cases in the functions below. All non–`void` functions should return 0 in all cases other than the error situations discussed.

All of the functions have a pointer to a `Filesystem` as a parameter; where the descriptions below refer to a function's "`Filesystem` parameter", this should be understood to mean the `Filesystem` that the pointer parameter points to. Phrases like "the root directory" or "the current directory" should also be understood to refer to that directory in the function's `Filesystem` parameter. The order that is easiest for describing the functions may not be the easiest or best order in which to implement them. Whichever order you do choose to code them in, you should implement one function at a time (entirely or even partially), then **test it** (as soon as that is possible) thoroughly before going on! Furthermore, you should write and test helper functions (for example, utility functions to manipulate your data structures) before writing code that uses them, and test them separately first as well.

In a few places below we say that the effect of performing some operations, or performing them in certain orders, is undefined. By this we mean that this project assignment does not specify any required behavior in those cases; your code can do anything at all, including working perfectly, or failing spectacularly. (Failing spectacularly is probably much easier to implement.) What this means is that it is up to the **user** of your functions to avoid calling them in such situations if they want their programs to work properly, **rather than** being your responsibility as the writer of the functions to detect these situations and have to handle them in any particular way.

The part of the filesystem and the functions that your program simulates is extremely limited, and also differs in some crucial respects from the behavior of the real UNIX filesystem and commands would be. You may want to experiment with the behavior of the commands that your functions simulate, but in all cases you should implement what this project assignment says, not what the real filesystem or UNIX commands do, in any case where these differ.

4.1 Function for initializing or creating a filesystem

```
void mkfs(Filesystem *files);
```

This function should initialize its `Filesystem` parameter, but exactly what that will do depends upon the way you decide to represent and store the components of a filesystem. This function must be called to initialize any `Filesystem` variable, before any other operations are performed on that variable. After calling this function, the initialized `Filesystem` should contain or consist of only a root directory.

The result of calling any of the other functions on a `Filesystem` variable before `mkfs()` is called on it is undefined. The result of calling any of the other functions is also undefined if `mkfs()` was initially called, but its argument was just `NULL`. After calling `mkfs()` with a non–`NULL` parameter any of the other functions can be called. Therefore it's up to the user of your functions (which includes any of our tests) to call itmkfs() on any `Filesystem` variables that are declared or created before calling any of the other functions on them, if they want them to be properly initialized.

Every call to `mkfs()` must initialize the `Filesystem` that its parameter `files` points to in such a way that calling `mkfs()` on different `Filesystem` variables causes each one to be a **different** filesystem. In other words, calling `mkfs` several times on different `Filesystem` variables will not cause them to share any files or directories.

4.2 Functions for creating files and directories

```
int touch(Filesystem *files, const char arg[]);
```

This function's usual effect is to create a file, if it does not already exist. In real UNIX, if the `touch` command is given with the name of a file that does not exist, a new, empty file will be created with that name; the usual effect of this function in your project is analogous. In real UNIX, if `touch` is given the name of an already–existing file, it will update that file's timestamp, however, this project has no notion of file or directory modification times or timestamps.

- If the function's argument `arg` is a name that does not refer to an existing file or directory located in the current directory at the time the function is called, its effect is to create a file with that name in the current directory (the current directory is mentioned further below).
- If `arg` is the name of a subdirectory or a file that already exists in the current directory, or is `.` (a single period), `..`, or `/` (a single forward slash), this function should have no effect.
- If `arg` does not consist solely of a forward–slash character, yet it contains a forward–slash character anywhere, the function should return the value -1 without making any modification to anything in its `Filesystem` parameter.

  In the real UNIX shell a command like "`touch rodent/hamster`" would create a file named `hamster` in the subdirectory `rodent` of the current directory (if it existed), because the forward slash is a directory separator. However, for simplicity in this project, such a command argument will just be invalid.
- If `arg` is an empty string the function should return the value -1 without making any modification to anything in its `Filesystem` parameter.

A file will contain zero lines (no contents at all) when it is first created with touch.

```
int mkdir(Filesystem *files, const char arg[]);
```

The usual effect of this function is to create a subdirectory in the current directory.

- If `arg` is a name that does not refer to an existing file or directory located in the current directory, the function's effect is to create a subdirectory with that name in the current directory.
- If `arg` is an empty string the function should return -1 and exit without modifying anything in its `Filesystem` parameter.
- If `arg` is the name of a file or of a subdirectory that already exists in the current directory, or is `.` (a single period), or `..`, or `/`, the function should return -1 without modifying anything in its `Filesystem` parameter.
- If `arg` does not consist solely of a forward–slash character, yet it contains a forward–slash character anywhere, the function should return the value -1 without making any modification to anything in its `Filesystem` parameter.

4.3 Function for navigating around a filesystem

```
int cd(Filesystem *files, const char arg[]);
```

This function's usual effect is to change the current directory. Note that a `Filesystem` must have some mechanism to store and keep track of its current location (the current directory) at all times.

- If `arg` is a name that does not refer to an existing file or directory located in the current directory the function should return -1 without changing anything in its `Filesystem` parameter.

- If `arg` is the name of a file that exists in the current directory at that time, the function should return −1 without changing anything in its `Filesystem` parameter.
- If `arg` is the name of a directory that exists as an immediate subdirectory of the current directory, that subdirectory will become the new current directory of its `Filesystem` parameter.
- If `arg` is `.` (a single period), or is an empty string, the function has no effect, as this simply changes the current directory to be the current directory.
- If `arg` is `..`, the parent directory of the current directory becomes the new current directory. However, although the root directory has no parent, if the function is called with `arg` being `..`, at a time when the root directory is the current directory, the operation is defined to have no effect. This is not an error, and the current directory remains the root directory.
- If `arg` is `/`, the root directory becomes the new current directory, regardless of what the current directory previously happened to be.
- If `arg` does not consist solely of a forward–slash character, yet it contains a forward–slash character anywhere, the function should return the value −1 without making any modification to anything in its `Filesystem` parameter.

4.4 Informational functions

```
int ls(Filesystem *files, const char arg[]);
```

This function's usual effect is to list the files and subdirectories of the current directory, or the files and subdirectories of its argument if that is a subdirectory, or to list its argument if that is a file.

- If `arg` is a name that does not refer to an existing file or directory located in the current directory at that time, the function should return −1 without producing any output or modifying anything in its `Filesystem` parameter.
- If `arg` is the name of a file that exists in the current directory, the function's effect is to print the name of that file. The file's name must appear on a line by itself, followed by a newline character, but without any other preceding or following whitespace or punctuation.
- If `arg` is the name of a directory that exists as a subdirectory of the current directory, the names of all of the files and subdirectories in that subdirectory should be printed, one per line, each followed by a newline character (including the last one).

  The list of files and subdirectories should be printed in increasing sorted order, where sorted order refers to a lexicographic ordering of file names. For this project, the lexicographic ordering of `str1` is less than (prior to) `str2` if and only if `strcmp(str1, str2)` returns a negative value. (Lexicographic order is usually the same as dictionary order, but file or directory names may contain punctuation or digit characters, or letter characters of different cases. Here `str1` and `str2` will be file or subdirectory names.) Filenames should not be printed with any preceding or following punctuation, but directory names must be printed ending with a single forward–slash character (`/`), to indicate that they are directories, as in the output of the real `ls` command when the `-F` option is used. There may be zero or more files and subdirectories in any directory; if the named directory currently contains no files or directories then the function will produce no output (not even a newline).
- If `arg` is `.` (a single period) or the empty string, the names of all of the files and subdirectories in the current directory should be printed, in the same format as described above.
- If `arg` is `/`, the names of all of the files and subdirectories in the root directory should be printed, in the format described above, regardless of the current location in the filesystem.
- If `arg` is `..`, the names of all of the files and subdirectories in the parent of the current directory should be printed, in the same format as described above. Other than the one possible exception below, this list will be nonempty, since the parent of the current directory must have at least the current directory as a subdirectory.

- If `arg` does not consist solely of a forward–slash character, yet it contains a forward–slash character anywhere, the function should return the value −1 without making any modification to anything in its `Filesystem` parameter.

If the current directory is the root directory, and the filesystem contains nothing other than the root directory, the effect of calling this function with `arg` being `..`, or `/`, or an empty string, will be identical to what would be produced if `arg` was `.` (a single period), namely no output should be produced.

```
void pwd(Filesystem *files);
```

This function's effect is to print the full path from the root to the current directory. The path begins with a forward–slash character (`/`), signifying that the root is the base of the entire filesystem and the parent or ancestor of all other files and directories. Following the slash, the names of all of the directories between the root directory and the current directory are printed in order from the top of the filesystem (the immediate subdirectory of the root), ending with the name of the current directory. Forward slashes must separate the names of the directories along this path, but a slash must not follow the last (current directory) name, and the path should be followed by a newline. The pwd function always produces at least some output. If the root directory is the current directory only a single forward–slash is printed.

Initially, after `mkfs()` is first called on a filesystem, and until any other directories are created in it and moved to, a filesystem's current directory is its root directory.

4.5 Functions for removing an entire filesystem, or individual files or directories

```
void rmfs(Filesystem *files);
```

This function should **deallocate** any dynamically-allocated memory that is used by the `Filesystem` variable that its parameter `files` points to, destroying the filesystem and all its data in the process. The parameter `files` should use no dynamically-allocated memory at all after this function is called.

The effect of calling any of the functions (**other than** `mkfs()`) after `rmfs()` is called is **undefined**. (The effect of calling `rmfs()` **before** `mkfs()` is first called on any `Filesystem` variable is also undefined, because it is stated above that is the case if any of the other functions are called before `mkfs()` is first called on a `Filesystem`.)

It is perfectly valid to call `mkfs()` after `rmfs()` is called, and the effect of `mkfs()` in that case should be to reinitialize its `Filesystem` parameter just as if it was the first time `mkfs()` had ever been called on it. If a `Filesystem` is created, files and directories are added to it, then `rmfs()` is called on it, followed by `mkfs()`, then new files and directories can be added to it just as if it was a newly-created empty `Filesystem`. The effect of the `rmfs()` is that no memory leaks should occur in this process.

If the user of your functions wants to avoid memory leaks they must always call `rmfs()` on any `Filesystem` variables before they go out of scope. (This is their responsibility to ensure, not your code's responsibility to detect or enforce.)

```
int rm(Filesystem *files, const char arg[]);
```

This function's usual effect is to remove a file or a directory from the current directory.

- If the function's argument `arg` is a name that does not refer to an existing file or directory located in the current directory at the time the function is called it should return −1 without changing anything in its `Filesystem` parameter.
- If `arg` is `.` (a single period), `..`, or `/`, the function should return −1 without changing anything in its `Filesystem` parameter.
- If `arg` does not consist solely of a forward–slash character, yet it contains a forward–slash character anywhere, the function should return the value −1 without making any modification to anything in its `Filesystem` parameter.
- If `arg` is an empty string the function should return the value −1 without making any modification to its `Filesystem` parameter.

- If `arg` is the name of a file that exists in the current directory at that time, the function should remove that file from the current directory. Note that this includes removing all lines that the file may contain (Section 4.6 describes the functions that manipulate files' contents).
- If `arg` is the name of a directory that exists as an immediate subdirectory of the current directory, and that directory currently contains **no files or subdirectories**, the function's effect is to cause that subdirectory to be removed from the current directory. However, if `arg` is the name of a directory that exists as an immediate subdirectory of the current directory, and that directory is nonempty (contains one or more files or subdirectories), then the function should return the value -1 without making any modification to its `Filesystem` parameter. (This is analogous to the effects of `rmdir` in real UNIX when called upon a nonempty directory.)

In removing files and directories this function must ensure that no memory leaks occur. The last file or directory could be removed from a directory, causing it to become an empty directory with no contents, but the current directory can never be removed.

4.6 File manipulation functions

```
int addline(Filesystem *files, const char arg[]);
```

This function's usual effect is to read a line of input and append it to a file in the current directory.

- If `arg` is any of the following the function should return -1 without modifying anything in its `Filesystem` parameter:
  - a name that does not refer to an existing file or directory located in the current directory at that time, or
  - the name of a directory that exists as a subdirectory of the current directory, or
  - `.` (a single period), `..` (two period characters), `/`, or the empty string, or
  - something that does not consist solely of a forward–slash character, yet it contains a forward–slash character anywhere.
- If `arg` is the name of a file that exists in the current directory, the function's effect is to read a line of input (including all characters up though newline character that terminates any input line), and append it as the new last line of the file that `arg` refers to. Note that this may be the first line in the file, if the file was previously empty.

Your function may assume, without checking, that any line entered will always be 80 characters or less. We will soon be covering in lecture a library function that will make it easy to read an input line (you should use this library function rather than writing code that duplicates its effects).

Notice that there is no way in this project to remove a subset of the lines of a file. Files may be created and lines added to them, and an entire file, including all of its contents, may be removed from the filesystem, but these are the only ways to modify the contents of a file.

```
int cat(Filesystem *files, const char arg[]);
```

This function should return -1 in exactly the same cases as `addline()` described above. If its parameter `arg` is the name of a file that exists in the current directory, the function's effect is to print all of the contents of the file, with each line appearing on a different output line, in order (meaning the first line that was added should be the first one printed, and the most recently added line should be printed last). Each line of the file, including the last one, should be printed on an output line by itself, meaning followed by a single newline character. If a file is empty (`addline()` has never been called with its name as an argument) then nothing will be printed (not even a newline).

5 Constraints and notes

- You **must** use some type of linked data structure to store filesystems and their components. Linked data structures vs. arrays were discussed in CMSC 132. Arrays are obviously used pervasively in programming,

and have many applications. However, as discussed in CMSC 132, insertion and deletion in an array require linear time, while they require $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ in many linked data structures, so arrays are not ideal for all programming situations. Since one of the main things to be learned in this course is memory management and pointer–based data structures in C, arrays **may not be used** to store the components of a `Filesystem` in this project.

Even if your program passes some– or all– of the project tests, the graders will be looking for use of arrays when your program style is checked, and **any use** of arrays to store the components of a `Filesystem` will result in a score of **zero** for the **entire project**.

If any of your type definitions in `filesystem-datastructure.h` use square braces (`[]`) anywhere, you are obviously using arrays. However, arrays can be used in C even in the absence of square braces, due to pointer arithmetic. If you are using any type of data structure for storing all or some of the components of a `Filesystem`, in which you have to specify a size, or where the data structure can get full and you need to create a larger one (using any of `malloc()`, `calloc()`, or `realloc()`) in order to add more data, then you are using arrays. If you have any doubts about whether you are using arrays, you are urged to discuss your planned design with the TAs during their office hours **before** starting to code.

- Character strings (arrays) **may** be used to store file and directory names. However, since no maximum length is specified for file and directory names, any such character strings **must** be **dynamically-allocated** strings (arrays), rather than fixed-size strings, so that file or directory names of any length may be passed to the functions without errors, and names of any length may be stored without wasting memory space.

  Character strings may also be used to store contents (lines) of files. Notice that you know the maximum size of each line (80 characters typed in), it is not necessary to dynamically allocate string for lines in files. However, no maximum number of lines has been given for the contents of a file, so the list of lines comprising a file's contents must also be stored in some type of linked structure.

- All functions should have no effect if any pointer or array parameter is `NULL` (returning 0 if the function is defined to have a return value). Note that this includes character string parameters as well. Even if a different error condition described above applies to another one of the function's arguments, this takes precedence over whatever return value or handling is specified for that situation. (Recall that Section 4.1 says the effect of calling any of the other functions is undefined if they are called after `mkfs()` is called with a `NULL` parameter.)

- Any functions that need to allocate memory should test whether memory allocation was successful, and print the message "Memory allocation failed!" (terminated by a newline) to its standard output if the memory allocation operation could not be performed. (This message must be spelled **exactly**, with capitalization and punctuation completely as shown.) After that, it does not matter what results are produced by your functions, since subsequent calls to most of them will not succeed, so their effects are undefined.

- Your functions must free any allocated memory when it is no longer needed, so no memory leaks will occur. See Section 7 for further information.

6 Writing a Makefile

You must write and submit a makefile along with your project code. Your makefile will be used to compile your program on all of the public tests; when the tests are provided you will know how many of them there are and what their names are, which you will need to know to write the makefile. Since you will not know before the project due date how many secret tests there will be, or what their names are, you will not be able to put rules in your makefile for compiling the secret tests, so we will use our own makefile to compile those.

Here are the requirements for your makefile:

- It **must** be in a file named `Makefile`.

- It **must** use the compiler options `-Wall`, `-ansi`, `-pedantic-errors`, and `-Werror`, the meanings of which were described earlier this semester, to build object files (not executable files). You may optionally add the `-g` option, to be able to run the `gdb` debugger on your code, and your submitted makefile may contain `-g` with the other options.
- Your `Makefile` **must** contain the following targets:

  all: This target must create executables for all the public tests.

  `filesystem.o`**:** This target must create the object file for `filesystem.c`.

  an object file for each public test source file: Because your makefile is **required** to use separate compilation, for each public test that is a C program you will need to create an object file for it, and use that file to make the corresponding executable.

  an executable for each public test: Your makefile will need to have a rule to build each public test executable. Your makefile **must** build executables with names ending in `.x` (`public01.x`, `public02.x`, etc.).

  clean: This target must delete the object files and executables for all the public tests, and the object file `filesystem.o`, from the current directory.

  Your "clean" target should **not** delete the object files `memory-checking.o` or `driver.o`, otherwise you will have to extract them from the project tarfile again after using this target and removing them. (The easiest way to avoid removing them to just avoid using wildcards when specifying the object files to delete, and just list the names of all files to be removed.)

- Your makefile may contain additional targets, for example to build whatever tests you create to exercise your functions with, so long as it has the targets described.

  Note that your makefile does **not** need to contain a target to create either of the object files `driver.o` or `memory-checking.o`, because they are being given to you.

- Your `Makefile` should have all needed dependencies, otherwise programs may not get built correctly, but it should not have any unnecessary dependencies, because that would lead to unnecessary compilations.

If you do not submit a makefile with your project, or if your makefile does not satisfy all these requirements, either your program will not compile at all on the submit server, or it might compile for the public tests but not for the secret tests, or at the minimum you will lose credit as part of your grade for programming style.

To reiterate, your `Makefile` **must** use separate compilation– each source file needed to form an executable must be separately compiled, then the independent object files linked together, to form the executable.

In this project one header file includes another one; we will be discussing in lecture how this situation should be handled in a makefile.

It is **strongly recommended** that before submitting you should run `make clean`, then run the single command `make`, to ensure that your makefile builds all the public tests correctly. This is the situation in which your makefile has to work on the submit server– being able to build the tests when there are no executable or object files in the directory– so you should test it and ensure it works right in the same circumstances.

Lastly, there are a lot of features of make that were not explained in class, because they are difficult for a beginner to use correctly. Consequently you are advised to avoid all features of make other than those covered in class. If you do use features of make not explained in class your submission may not compile on the submit server for technical reasons not worth fully explaining here, and the TAs will probably not be able to help you in office hours with your makefile. You are advised to just write a simple and straightforward makefile.

7 Memory checking and memory errors

We are supplying you with a compiled object file named `memory-checking.o`, along with its associated header file `memory-checking.h`. These files define two functions, `setup_memory_checking()` and `check_memory_leak()`, which have no parameters or return value. `setup_memory_checking()` will be called

by our tests, and must be called **before** any memory is dynamically allocated; it will set up things to have the consistency and correctness of the heap checked. If it is initially called, and your code has any subsequent memory errors (such as overwriting beyond the end of an allocated memory region in the heap) the program will crash, consequently failing the test. If `check_memory_leak()` is called at any point it will print a message indicating whether there is any memory in use in the heap at all.

Some of our tests will initially call `setup_memory_checking()`, and call `check_memory_leak()` after `rmfs()` is called on all `Filesystem` variables in use. If your `rmfs()` function (or any other function) is not freeing memory properly your program will report that there is some allocated memory in the heap, meaning a memory leak occurred, causing such tests to fail.

If a user of your functions calls `mkfs()` on a `Filesystem` variable that currently contains some files or directories (meaning that it is using some dynamically-allocated memory) **before** `rmfs()` is called on it (in other words, calling `mkfs()` twice without an intervening `rmfs()`), the effect will be that a memory leak occurs. There is nothing that your functions can do to detect or prevent this. As above, it is the responsibility of the user of your functions, if they want to avoid memory leaks, to call `rmfs()` on any `Filesystem` variables that have any contents, before calling `mkfs()` on them. This does not apply to the initial call to `mkfs()`, since a `Filesystem` has no contents before `mkfs()` is first called on it.

The facilities that we are using to perform this memory checking are not compatible with `memcheck` (valgrind) (which will be covered in class very shortly), because `memcheck` is using its own version of the memory management functions that use some memory in the heap. Consequently, if you run any tests that use our memory checking functions under `valgrind`, it will appear that the test is failing with a memory leak, even if your code does not have memory leaks. You can use either our memory checking functions, or `valgrind`, with any program, but both cannot be used together and give correct results.

8 Test driver

A number of functions must be called to create filesystems and navigate around them, and test the various features of the project. As a result, writing test drivers for checking your code could be laborious. To make things easier, when the public tests are posted we will also provide a driver program that calls your functions, simulating the interactive nature of a UNIX shell or command interpreter. The driver enables you to test a variety of project features without writing any test code at all, by typing commands the same way they are input to the UNIX shell.

The driver is supplied as an object file `driver.o` (compiled on the Grace machines), since it uses some C language features that have not been covered in class. If you link it with your project code and run the resulting program, the driver will print a prompt consisting of a single percent sign, and wait for you to enter a command. It recognizes commands corresponding to each function you have to implement (the command `mkfs` for the function `mkfs()`, the command `touch` for the function `touch()`, etc.), each followed by zero or one argument. As each line you type is read the driver will break it up into its constituent fields and call your function that has the same name as the first word on the line entered, passing the remaining values on the line as arguments to your function. Here is some additional information about the driver:

- The driver creates and uses exactly one `Filesystem`, while our tests (and your tests not using the driver) may create and use more than one `Filesystem`.
- The driver will stop reading input and quit if the commands `logout` or `exit` are given, or if the end of its input is seen.
- The driver assumes that no input line is longer than 512 characters, and no word on an input line is longer than 80 characters, so errors may occur if you give it input that exceeds these sizes. (These are assumptions our driver makes, not assumptions that your functions should make; they are not stated anywhere above in the project requirements.)
- If an input line with an invalid command is entered the driver will print a line containing the message "`Command not found.`". If a valid command is followed by an incorrect number of arguments the driver will print a line with the message "`Invalid arguments.`"

Blank input lines, or lines consisting of only whitespace, are ignored by the driver; after one such line a new prompt is printed and a new line will be read.

- The driver prints a generic error message if function returns the error code -1.
- For `ls()` and `cd()`, the driver allows either the command followed by an argument, or the command alone (which calls the function with an empty string) to be entered.
- The driver does not call `mkfs()` on its `Filesystem` variable. Tests using the driver must contain `mkfs` as their first command, if the writer of the tests wants their code to work properly.
- The driver also does not call `rmfs()` on its `Filesystem` variable. Tests using the driver must contain `rmfs` as their last command, if the writer of the tests wants to avoid memory leaks.
- The driver does not use the memory testing functions, so any tests that check for correctness and consistency of the heap will not use the driver.
- To facilitate testing your program so that a long series of commands does not have to be retyped every time the driver is run, you can redirect standard input from a file into which you have typed commands. In order to make it easy to read the output in this case the driver recognizes a command "`set verbose`", that causes every input line to be printed to the output before its command is executed. This is useful because when a program is run using input redirection the input is not displayed. An example is below. A command "`unset verbose`" is also available, although you may not have much need for it.

9 Example

The output of the `main()` function below is shown, assuming the functions for the project are correctly implemented, compiled and run.

```
#include <stdio.h>
#include "filesystem.h"

int main() {
  Filesystem f;

  mkfs(&f);

  mkdir(&f, "ape");

  cd(&f, "ape");
  touch(&f, "hippo");
  ls(&f, ".");  printf("\n");

  cd(&f, "..");
  mkdir(&f, "emu");

  cd(&f, "emu");
  mkdir(&f, "kangaroo");
  touch(&f, "badger");
  mkdir(&f, "frog");

  cd(&f, "/");
  ls(&f, "");  printf("\n");

  ls(&f, "emu");  printf("\n");

  ls(&f, "ape");

  return 0;
}
```

The output would be:

```
hippo

ape/
emu/

badger
frog/
kangaroo/

hippo
```

Assuming you linked the driver program `driver.o` with your code to create an executable program named `driver.x`, below is a similar sample execution that uses input redirection. First the contents of the file `example-commands` are shown, then the input of the program `driver.x` is shown when run with input redirected from that file. If the first command in that file was not "`set verbose`" the only output produced would be what resulted from the `ls` commands, which would make the example hard to follow. Note the two error messages are printed by our driver.


```
mkfs
set verbose
mkdir ape
cd ape
touch hippo
ls .
cd ..
mkdir emu
cd gerbil
cd emu
mkdir kangaroo
touch badger
mkdir frog
cd /
ls
ls emu
ls turtle
rmfs

% driver < example-commands
%
% mkdir ape

% cd ape

% touch hippo

% ls .
hippo

% cd ..
```

```
% mkdir emu

% cd gerbil
Missing, incorrect, or invalid operand.

% cd emu

% mkdir kangaroo

% touch badger

% mkdir frog

% cd /

% ls
ape/
emu/

% ls emu
badger
frog/
kangaroo/

% ls turtle
Missing, incorrect, or invalid operand.

% rmfs

%
```

A Development procedure review

A.1 Obtaining the project files, compiling, and checking your results

You can obtain the necessary project files by logging into one of the Grace machines using commands similar to those used before:

```
cd ~/216
cp ~/216public/project3/project3.tgz .
tar -zxvf project3.tgz
```

The tarfile will create a directory `project3` that contains the necessary files for the project, including the header file `filesystem.h` and the public tests (when they are available). You **must** do your coding in your extra course disk space (using the `cd` command above) for this class, otherwise we **will not accept** your submission. After extracting the files from the tarfile, `cd` to the `project3` directory, create a file named `filesystem.c` that will `#include filesystem.h`, and write the functions whose prototypes are in `filesystem.h` in that source file. (Your `filesystem.c` does not need to include `filesystem-datastructure.h`, since it includes `filesystem.h` and `filesystem.h` includes `filesystem-datastructure.h`.)

See Section 6 regarding how to compile your code.

Commands like those before can be used to run your code and determine whether it passes a public test or not, for example:

```
public1.x | diff -u - public1.output
```

If no differences exist between your output and the correct output diff will produce no output, and your code passed the test. For any tests that have an associated data file from which input should be redirected, the command would be run as follows:

```
public1.x < public1.input | diff -u - public1.output
```


A.2 Submitting your program

As before, the command `submit` will submit your project. **Before** you submit, however, you must first check to make sure you have passed all the public tests, by running them yourself. And **after** you submit, you **must** then log onto the submit server and check whether your program worked right on the public tests there.

Unless you have versions of all required functions that will at least compile– whether they are used in the public tests or not– your program will fail to compile at all on the submit server. (You need to have at least skeleton functions that just contain an appropriate `return` statement for any non–`void` functions, even if the functions don't actually work.)

Do **not** submit programming assignments using the submit server's mechanism for uploading a jarfile or zipfile or individual files. Do not wait until the last minute to submit your program, because the submission server enforces deadlines to the exact second.

Project extensions will not be given to individual students as a result of hardware problems, network problems, power outages, etc., so you are urged to finish **early** so any such situations will not affect you even if they do arise.

A.3 Grading criteria

Your project grade will be determined according to the following weights:

Results of public tests	35%
Results of secret tests	50%
Code style grading	15%

The public tests will not be comprehensive, and you will need to do testing on your own to ensure the correctness of your functions.

A.3.1 Style grading

The only files that will be graded for style are `filesystem.c` and `filesystem-datastructure.h`. Be sure to **carefully read** the course project grading policy posted on ELMS, which contains all details about the criteria that will be used for style grading, so you can follow them and avoid losing credit unnecessarily.

B Project–specific requirements, and notes

- To reiterate again, you **must** use a linked data structure for storing filesystems, not arrays (reread Section section:constraints).
- You **cannot** modify the declaration of anything in the file `filesystem.h`, or add anything to it, because your submission will be compiled on the submit server using our version of this file.

  Your code **may not** comprise any source (.c) files other than `filesystem.c`, so all your code **must** be in that file.
- Be sure to carefully reread the requirements for your makefile in Section 6 before submitting.
- As the project grading policy says, you should use only the features of C that have been covered in class, or that are in the chapters covered, up through the time the project is assigned.
- **Write your own tests, and test each function as you write it, before going on!**
- Recall that the course syllabus says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details. (The compilation penalties mentioned below only apply to submissions made before the end of the project late submission period.)

- For this project you will **lose one point** from your final project score for every submission that you make in excess of six submissions. You will also **lose one point** for every submission that does not compile, in excess of two noncompiling submissions. Therefore be sure to compile, run, and test your project's results **before** submitting it. Also check your makefile carefully, as described in Section 6 above! We hope everyone will check their code themselves carefully, and no one will incur these penalties.

C Extra credit

If you make **only one submission** for this project, and that submission passes all of the **public and secret tests**, we will give you **10 extra–credit bonus points** when your project is graded. In order to make only one submission for the project, and have it pass all the public and secret tests, you are going to have to read this assignment very carefully, and write extensive tests of your methods yourself, as well as use good style while writing your code. Some of the secret tests for the project may be difficult, and the project as a whole is quite challenging, so do not be too disappointed if you do not earn these points, but even if you do not, the harder you try to get them the better your score will be on the secret tests and on the project in general.

(Note that your submission can be on time, one day late, or two days late, and you can still get the extra credit, as long as it is your only submission for the project, and it passes all the public and secret tests.)

As mentioned in the prior section, credit may be lost for students who make an excessive number of submissions, or have too many noncompiling submissions.

D Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.

