Date due: Thursday, April 7, 11:59:59 p.m.

1 Introduction and purpose

The Nelovo computer company wants to develop a new operating system to run on its new CPUs, which you are familiar with from Projects #3 and #4. The operating system's code name during development is Ournix. Due to your increasing proficiency in C, which Nelovo has heard about, they want to hire you as an intern to write a small simulation of the filesystem component of the planned operating system, so they can see how it will appear to users. Your simulation will model the filesystem and a few of the most common commands. Your code 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 create and manipulate a dynamically–allocated linked data structure in C. You will have to write a makefile that will compile your code with all of the public tests, so you will get more practice with makefiles. Another purpose is to get some basic experience with a few C string library functions. You **must** use the string library functions to manipulate string data (names) rather than write code (loops or recursion) that duplicates their effects.

You will create your own data structure for storing the components of a simulated Ournix filesystem. Think about how to design the project and ask the TAs in office hours if you have questions prior to starting to code. Keep in mind that your design for the project, just like your implementation, is individual work **only**, and other than asking the instructional staff in office hours, you cannot discuss your design or data structures with fellow students, or anyone else.

This project will be difficult, because although a working data structure is not more complex than ones you would have created in CMSC 132, many bugs can arise when first using dynamically–allocated linked data structures in C.

The way to find bugs that will arise when using linked data structures in C is to use the `gdb` debugger, so you **must** know how to use `gdb`, 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. If you need help in knowing how to use `gdb`, ask the TAs in office hours.

As explained below, some of the tests will check for memory leaks and memory errors resulting in heap corruption (having invalid data in the heap). You can find these types of errors by running `valgrind` on all the public tests. **You must also know how to use `valgrind`, and to have used it to try to fix any problems that it identifies (see below) before asking for debugging help in the TAs' office hours.** The TAs will not look at code that you have not already run `valgrind` on and tried to fix memory errors in. (See Appendix B regarding using `valgrind` on the project tests.)

1.1 Extra credit

You can again get extra credit for this project. If you make **only one submission** that passes **all the public tests** you will get **7 extra–credit bonus points**. And if your single submission is made **at least 48 hours before the on–time deadline** you will get **8 additional extra credit bonus points**, for 15 extra credit points total. (Obviously you can't get the second 8 extra credit points if you don't get the first 7, but you can get the first 7 without getting these second 8.)

However, as before, if you make too many submissions you may again lose credit. To avoid making submissions that don't compile or don't pass all of the public tests you should compile the tests, run them, check your output, and fix any problems, all **before** submitting. **And** you should **test your makefile** and run `valgrind`, as explained below.

2 Filesystem components

Your code will allow both regular files and directories to be simulated. Note that Ournix is superficially rather similar to UNIX, although your simulation will be much simpler. Both files and directories have names.

- A file for this project just consists of a name; it does not have any contents, although it does have a simulated timestamp (see below). Files are located in directories.
- A directory can store both files and other directories (subdirectories). A directory can contain zero or more files and zero or more subdirectories. A directory cannot contain multiple components (files or subdirectories) that have the same name, but it can contain components with the same names as components located in other directories. For example, a directory `a` can contain a file or subdirectory named `z`, and another directory `b` can also contain a file or subdirectory `z`. A directory named `z` can even contain a file or subdirectory named `z`.

Note that an Ournix variable must have some mechanism to keep track of its current location (the current directory) at all times.

Names of files and directories consist of sequences of one or more characters, with the exceptions that they 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 they do in 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 whichever directory happens to be current at the time they are called.
- Two adjacent period characters refer to the parent of the current directory, meaning the one immediately above it in the directory structure. (For the root directory, ".." is the same as "." because there is no directory above it.)
- The forward–slash character refers to the root directory, which always exists as the topmost directory of an entire filesystem, 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 in its name.

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

3 Data structure requirements, and memory management

There is **no maximum size** for a simulated Ournix filesystem– it may contain any number of files and any number of directories and subdirectories, limited only by the amount of memory in the computer running your program. Similarly, the name of a file or directory may be an arbitrarily long string. Situations where programs have to store an unknown amount of data, or data structures where there is no limit to the amount of data that needs to be stored, are natural applications of dynamic memory allocation and linked data structures.

As mentioned, you will create your own data structure for storing the components of a filesystem. The header file `ournix.h` in the project tarfile contains the prototypes of the required functions, but it has no type definitions. Instead it includes another header file `ournix-datastructure.h` that is not provided. You must write this file (whose name must be spelled and capitalized **exactly** as shown), which must contain a definition of the type `Ournix` that represents a filesystem, and which all of the functions operate upon. `ournix-datastructure.h` may also contain any other definitions that are needed for your `Ournix` definition. The requirements that your data structure **must** follow are:

- You **must** use **only** dynamically–allocated data structures to store **all** of the data and components of filesystems. You could certainly get the functions to work for many cases without using dynamically–allocated data structures (for example, by using fixed–size 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.
- Your filesystem data structure can **only** use linked, dynamically–allocated data structures. Other than for storing file and directory names, **no arrays can be used**– and even file and directory names must be stored in **dynamically–allocated** arrays (strings). You **cannot** use any **fixed–size** arrays **anywhere** in your filesystem data structure. If any of your type definitions in `ournix-datastructure.h` use square braces (`[]`) anywhere, you are using fixed–size arrays, which is disallowed. (Function parameters with square braces are not part of your filesystem data structure, so they are fine.)

If you have any doubts about whether you are violating these requirements you can always discuss your planned design with the TAs during office hours before starting to code.

There are no requirements for your own tests, so these restrictions do not apply to them, only to your simulated filesystem implementation.

3.1 Memory management

A robust C program should always check that all memory allocations succeed, and take appropriate action if not. (The appropriate action might just be gracefully exiting the program, but it at least should not be just crashing). However, to avoid one thing to have to do in the project you may **assume that all memory allocations always succeed**, so you are encouraged but not required to check the return values of calls to memory allocation functions.

A robust C program should always free all dynamically allocated memory when it's no longer needed, to prevent memory leaks. However, most of the public tests do not require that memory is freed, so most of them will pass even if there are memory leaks. You will write functions that free the memory of a `Ournix` variable and remove components from it, but only a few of the public tests use these functions. However, many of the secret tests will use them, meaning that many of the secret tests will pass only if memory is being correctly freed where your code would need to free it.

Any tests that are not requiring that memory be freed will have memory leaks, which you don't have to worry about. (Of course if you leak the entire heap memory and ask for more the operating system will terminate your program, but given the sizes of inputs your functions will be run on, that should not occur.)

4 Compiling using `make`

You must write a makefile, which will be used to compile your code for all the public tests, so it must have a rule for each public test present in the project tarfile. Since you won't know how many secret tests there are, or what their dependencies should be, we will use our own makefile to compile them.

Do not wait to write your makefile. Write it first– before you even start writing your code. (That is why this section appears before the required functions are even described.) Writing your makefile first will preclude the need to type commands for compiling the public tests, so you can save time and avoid making mistakes. Much more importantly, mistakes in writing a makefile can cause your code to not compile correctly– writing your makefile first and using it all during development, every time you compile your code, will allow you to ensure it works right. So what you should do is to read the rest of this assignment to understand what you have to do, look at the public tests and figure out how they would have to be compiled and what their dependencies are, then write your makefile first. Then write your `Ournix` definition in `ournix-datastructure.h` and write the functions. The requirements for your makefile are:

- It **must** be in a file named `Makefile` (with a capital first letter). (Your code will not compile on the submit server if your makefile name is wrong.)
- It **must** use the five `gcc` options `-ansi`, `-pedantic-errors`, `-Wall`, `-fstack-protector-all`, and `-Werror`, which were mentioned in the first project, to build object files. (But do **not** use these options to create executable files.) It does not matter if your makefile contains `-g` (to be able to run `gdb` or `valgrind`) with the other options.
- Your `Makefile` **must** contain the following targets:

  `all:` This target must appear first and must create executables for all the public tests (and only for the public tests).

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

  an object file for each public test source file: Because your makefile is **required** to use separate compilation, it will need to create an object file for each public test source file.

  Note that different tests will have different dependencies, which you can see by looking at what they include.

  an executable for each public test: Your makefile will need to have a separate target to build each public test executable. These targets must have names ending in `.x` (`public01.x`, `public02.x`, etc.) and **must** build executables with the same filenames as those targets. (Your code will not work on the submit server if your makefile has incorrect target names or builds executables with the wrong names.) By looking at the source file for each public test you can see what object files have to be linked together to create it.

  Because your makefile is **required** to use separate compilation, it must link the appropriate object files necessary to build each executable, **not** directly compile multiple source files together to form executables. So there should never be more than **one** source (`.c`) file in any compilation command in your makefile.

  As above, different tests will have different dependencies and will require different object files to be linked together to create their executable.

clean: This target must remove the object files and executables for all the public tests and `ournix.o` from the current directory.

- Your makefile may contain additional targets, for example to build your own tests, as long as it has the targets described above. However, do **not** add any of your own tests to the **all** target, because things may not work on the submit server if you do.
- Your makefile should **not** directly compile header files in compilation commands (e.g., do **not** have compilation commands like `gcc -c ournix.c ournix.h`).
- 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.
- Section 6 below explains that a few public tests use two functions that are defined in a provided compiled object file named `memory-checking.o` and (with associated header file `memory-checking.h`). Section 7 explains that one test also uses a function that is defined in a provided object file `driver.o` (with associated header file `driver.h`). You will have to add these two object files to the compilation commands in your makefile for the tests that use the functions in them. The public test executables that use the functions in these object files, and the object files for the tests that use them, will have different dependencies than the executables and object files for the other tests, and an additional object file must be linked to form the executable for these tests compared to the other public test executables.
- Do **not** have targets in your makefile to create `memory-checking.o` or `driver.o`, because you are not being given their source files `memory-checking.c` or `driver.c`. Just use `memory-checking.o` and `driver.o` in the linking rules for tests that call the functions in the associated header files, without having rule creating these object files. But your makefile must **recompile anything that uses these object files** if they were to change (for example, if we were to give you new versions of these object files to fix a bug).

  And if any files include `memory-checking.h` or `driver.h` they must be recompiled if these header files were changed as well. (Even though you should not change them, we could need to give you a new version of them.)
- Your `clean` target should remove all object files **other than** `memory-checking.o` and `driver.o`. (If did remove these object files it would just force you to extract them again from the project tarfile after running `make clean`.) So don't use a wildcard in your `clean` target to remove `*.o` (instead just explicitly list all the object files that should be removed).
- Make has many features that were not explained in class, because they are difficult for a beginner to use correctly. You can **only use the features of make that were covered in class**. If you use features not explained in class your code may not compile on the submit server for reasons not worth explaining, and the TAs can not help fix makefiles using features not covered. **You will lose credit for using any makefile features not covered in class.**

  Almost every past student who tried to use advanced features of make in this course made at least some mistakes. So you should write a simple and straightforward makefile, using just the features of make that were covered.

If there is no makefile with your submission, or if your makefile does not satisfy 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.

Before submitting, you **must** run `make clean`, then run the commands `make public01.x`, `make public02.x`, `make public03.x`, etc., to ensure your makefile builds all the public tests correctly. (Use these separate commands, not `make` or `make all`.) Your makefile has to able to build the tests like this on the submit server– meaning when there are no executable or object files in the directory– so test it in the same situation to ensure it works right then.

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.

5 Functions to be written

The functions to be written all have a pointer to an Ournix (i.e., `Ournix *`) parameter. For brevity, wording like "the function's Ournix parameter" is used to mean "the Ournix variable that the function's pointer parameter points to". Phrases like "the root directory" or "the current directory" refer to that directory in the function's Ournix parameter. Two functions (`ls()` and `pwd()`) can produce output; the rest only modify their `Ournix` parameter or return a value.

Some cases below are described as error cases. For any error cases the functions should not have any effect, meaning they should not modify anything in their `Ournix` parameter, and if they have a return value they should just return 0 in error cases. All non–`void` functions should return 1 in **all cases other than** the error cases described.

To avoid repetition these are stated just once here, but don't forget about them. **It is an error case if any function's pointer or array parameter is `NULL`.** (The function should have no effect in this case, and just return 0 if it is a non–`void` function.) For the functions that have a character string argument, it is also an error case if `name` contains a forward–slash character and does not consist solely of a forward–slash character (e.g., "`/`" is not an error but "`/a`", "`a/b`", and "`//`" are errors).

To get a good idea of what your functions have to do and how they will be called, study the public tests carefully and be sure to understand their output, after reading the descriptions of the functions below, but before starting to code.

Be sure to carefully read the requirements in Appendix B before starting to code, because you will lose credit for not following them.

Write one function at a time (entirely or even partially), then **test it** (as soon as that is possible) thoroughly before going on! 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.

Your project will consist of three required user–written files: `Makefile`, `ournix-datastructure.h`, and `ournix.c`. The required functions will all be in `ournix.c`.

The part of the filesystem and the commands that your functions simulate are extremely limited compared to what the behavior of the real UNIX filesystem and commands would be, and also differs in various respects. (An example is that file and directory names can contain punctuation characters here that have special meanings in real UNIX.) You should implement what this project assignment says, not what real UNIX would do, in any case where these differ. The reason that your project does not faithfully imitate real UNIX in various respects is just to make the project much easier.

5.1 The function for initializing a filesystem `void mkfs(Ournix *const filesystem)`

This function should initialize its `Ournix` parameter, but exactly what that will do depends upon the way you decide to represent and store the components of a filesystem (meaning what your definition of `Ournix` is, and how it will need to be initialized). This function will be called to initialize any `Ournix` variable, before any other operations are performed on that variable. After calling this function the initialized parameter should contain or consist of only a root directory, and the root directory should be that filesystem's current directory.

Note that the **caller** created the variable that the parameter `filesystem` points to, and is just passing its address into this function. `mkfs()` may or may not have to allocate memory for the `Ournix`– depending on your data structure– but the variable that its parameter `filesystem` points to **already exists**– it was created in whatever code is **calling** this function– so it should **not** itself be allocated by this function. The parameter should just be **initialized**, assuming its contents are currently just uninitialized (garbage).

A call to `mkfs()` must initialize its parameter such that calling `mkfs()` on different `Ournix` variables causes each one to represent a **different** filesystem. In other words, calling `mkfs()` on different `Ournix` variables will not cause them to share any files or directories.

5.2 Functions for creating files and directories

5.2.1 `int touch(Ournix *const filesystem, const char name[])`

This function's effect is to create a file, if it does not already exist, or to update the timestamp of a file if it does already exist.

- If the function's argument `name` is a name that does not refer to an existing file or directory located in the current directory when the function is called, and it is not one of the special values mentioned below, its effect is to create a file with that name in the current directory (the current directory is discussed further below). (Recall that it should then return 1, since the beginning of this section said that functions should return 1 in any cases other than specifically–indicated error cases.) In our simple simulation a file's timestamp is just a simple integer. The newly created file should have a timestamp of 1.
- If `name` is the name of a **file** (not a subdirectory) that already exists in the current directory the function's effect should be to increment its timestamp by 1.

- If `name` is the name of a **subdirectory** (not 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 and should not modify anything but should return 1 (these are not error cases).
- It is an error case if `name` is an empty string. The error case mentioned above, where `name` does not consist solely of a forward–slash character but contains a forward–slash character somewhere, applies to this function (and also to the other ones even if not explicitly repeated for each one).

5.2.2 `int mkdir(Ournix *const filesystem, const char name[])`

The usual effect of this function is to create a subdirectory in the current directory, if it does not already exist.

- If `name` is a name that does not refer to an existing file or directory located in the current directory, and it is not one of the special values mentioned below, its effect is to create a subdirectory with that name in the current directory.
- These are all error cases: if `name` is the name of a file or a subdirectory that already exists in the current directory, or is `.` (a single period), or `..`, or `/`, or an empty string.

5.2.3 Notes about `touch()` and `mkdir()`

When these functions have to manipulate file or directory names they should **use the string library functions to do so**. **Do <u>not</u> write loops (or use recursion)** to process strings, or you will **lose credit**. If these or other functions have to allocate memory do **<u>not</u>** cast the return value of the memory allocation functions, or you will **also** lose credit.

Unlike real UNIX, in this project you can only create files and directories in the current directory– there is no analogue to commands like `mkdir sheep/baa` or `touch wool/sweater`, which are possible in real UNIX (if the directories `sheep` and `wool` exist in the current directory). This is a consequence of the descriptions of the effects of the functions above. If you want to create a file or directory somewhere other than the current directory, you must first use `cd()` to navigate there (which may take multiple calls), then create it.

When a new file or directory is created, these functions must copy their parameter `name` into a **newly–created dynamically–allocated** array (meaning a deep copy) of size just sufficient to store the string. This ensures that the string, regardless of its size, is stored without wasting memory space. It also ensures correctness if the caller of the function later modifies or frees whatever they passed into `name`. These functions **cannot just make something in your data structure point to their parameter `name`**, because many errors will result if they just do pointer aliasing like this.

5.3 The function for navigating around a filesystem `int cd(Ournix *const filesystem, const char name[])`

This function's usual effect is to change the current directory of its `Ournix` parameter. As mentioned above, an `Ournix` variable must have some mechanism to keep track of its current director location at all times.

- If `name` is the name of a directory that exists as an immediate subdirectory of the current directory, that subdirectory should become the new current directory of its Ournix parameter.
- If `name` is `.` (a single period), or is an empty string, the function should not have any effect, as this simply changes the current directory to be the current directory (but this is not an error case).
- If `name` is `..`, the current directory should change to be the parent directory of the current directory. However, although the root directory has no parent, if the function is called with `name` being `..` at a time when the root directory is the current directory, the operation is defined to just have no effect and not modify anything. This is not an error case. (The special root directory effectively acts as if it were its own parent.)
- If `name` is `/` the current directory should change to be the root directory, regardless of what the current directory previously happened to be.
- It is an error case if `name` is a name that does not refer to an existing file or directory located in the current directory, and it is not one of the special values `.`, `..`, `/`, or an empty string. It is also an error case if `name` is the name of a file (not a directory) that exists in the current directory.

Note that, unlike real UNIX, there is no way to change to a directory more than one directory away in the filesystem in one call to this function (except if `name` is `/`). In real UNIX, one command could change to a location far away in the

filesystem, like for example `cd i/love/cute/fuzzy/sheep`. But in this project you would have to call `cd()` to change to the subdirectory `i`, then `cd()` to change to `love`, then `cd()` to change to `cute`, then `cd()` to change to `fuzzy`, and lastly `cd()` to change to `sheep`, to have the same effect. (This is due to the descriptions above of how names containing forward slashes are error cases.)

5.4 Informational functions

5.4.1 `int ls(Ournix *const filesystem, const char name[])`

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

- If `name` is the name of a file that exists in the current directory, the function's effect is to print the single name of that file followed by a single blank space and the timestamp of that file, which will always be 1 or more. The file's name and timestamp must appear on a line by themselves, and must be followed by a newline.
- If `name` is the name of a directory that exists as an immediate subdirectory of the current directory, the names of all of the files and subdirectories that have been created in that subdirectory should be printed, one per line, each followed by a newline (including the last one).

  The list of files and subdirectories should be printed in **increasing lexicographic (dictionary) order**. For this project, `str1` precedes `str2` in lexicographic order (`str1` and `str2` will be file or subdirectory names) if and only if `strcmp(str1, str2)` returns a negative value. Filenames should be printed with no preceding or following punctuation or whitespace, 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. When the entire contents of a directory are printed the timestamps of files are **omitted** after the files' names themselves.

  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 should not print any output (not even a newline).
- If `name` is `.` (a single period), or is an empty string, the names of all the files and subdirectories in the current directory should be printed, in the same format described immediately above. If the current directory has no files or directories then the function should not print any output (not even a newline).
- If `name` is `..`, the names of all the files and subdirectories in the parent of the current directory should be printed, in the same format described above. Other than the one possible exception at the end of this section this list will be nonempty, since the parent of the current directory must have at least the current directory as a subdirectory.
- If `name` 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 directory or location in the filesystem.
- It is an error case if `name` is a name that does not refer to an existing file or directory located in the current directory at that time, and it is not one of the special values `.`, `..`, `/`, or an empty string.

Note that `ls()` does not change the current directory, even if a different directory's contents should be printed (for example, if `name` is `/` or the name of a subdirectory).

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 `name` being `..`, or `/`, or an empty string, will be identical to what would be produced if `name` was `.` (a single period), namely no output should be produced.

5.4.2 `void pwd(Ournix *const filesystem)`

This function's effect is to print the full path from the root to the current directory. The path must begin 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 should be 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. The path should be followed by a newline. Unless its parameter is `NULL` the `pwd()` function always produces at least some output. If the root directory is the current directory only a single forward slash should be printed.

5.5 Functions to remove all or part of a simulated filesystem

5.5.1 `void rmfs(Ournix *const filesystem)`

This function should **deallocate** all dynamically–allocated memory that is used by the entire Ournix variable that its parameter `filesystem` points to, destroying the filesystem and all its data in the process. The filesystem data structure should not use any dynamically–allocated memory at all after this function is called.

If the user of your functions wants to avoid memory leaks they must call `rmfs()` on any `Ournix` variable after they are finished using it but before it goes out of scope. Not calling `rmfs()` will result in memory leaks, which in practice an actual user of C code would want to avoid.

5.5.2 `int rm(Ournix *const filesystem, const char name[])`

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

- The following are error conditions: if the function's argument `name` is a name that does not refer to an existing file or directory located in the current directory, if it is `.` (a single period), `..`, `/`, or an empty string, or if it contains a forward–slash character somewhere.
- If `name` is the name of a file that exists in the current directory, the function should remove that file from the current directory and return 1.
- If `name` is the name of a directory that exists as an immediate subdirectory of the current directory, the function should remove that directory **and** all of its contents from the current directory and return 1. (Note that its contents may also be directories that themselves have contents, etc.)

In removing files and directories the function must ensure that no memory leaks occur. Note that the last file or directory could be removed from a directory, causing it to become an empty directory with no contents. Of course a subdirectory could be removed, which would also remove all of its contents, but as a consequence of the description above, the current directory and the root directory can never be removed by this function. (Of course `rmfs()` described above could be called to remove or destroy the entire filesystem.)

5.6 Valid sequences of calls to the functions

All valid sequences of calls to your functions on an `Ournix` variable must obey the following:

- `mkfs()` must be the first function called on any `Ournix` variable.
- After `rmfs()` is called on an `Ournix` variable the **only** function that can be called on that same variable is `mkfs()` again. After calling `rmfs()`, the effect of calling any of the functions **other than `mkfs()`** is undefined.
- After `rmfs()` and then `mkfs()` are called on an `Ournix` variable any of the other functions may again be called on that variable.

  It is perfectly valid to call `mkfs()` on an `Ournix` variable after `rmfs()` is called on it, and `mkfs()` should reinitialize it just as if it was the first time it had ever been called on it. In other words, if a filesystem is created, files and directories are added, then `rmfs()` is called on it, followed by `mkfs()`, new files and directories can be added just as if it was a newly–created empty filesystem. No memory leaks should occur in this process.
- The only time that `mkfs()` can be called on an `Ournix` variable is if it has just been created (no functions at all have been called on it yet), or immediately after a call to `rmfs()`.

  The effect of calling `mkfs()` on an already initialized `Ournix` variable– one that `mkfs()` has already been called on, which possibly has had files and directories added to it)– is also undefined.

The effect of any sequences of calls to the functions that do not obey these properties is **undefined**, which means that your code can do anything in such cases. **Ensuring that these properties are not violated is the responsibility of the caller of your functions** (which includes your own tests); **your code does not have to try to detect violations of these properties**, and in fact it has no way to do so.

As above, if the user of your functions wants to avoid memory leaks they must call `rmfs()` on any `Ournix` variable after they are finished using it and before it goes out of scope. Note that a sequence of calls to the functions that does not call `rmfs()` on a `Ournix` variable before it goes out of scope will still be a valid sequence if it obeys the properties

listed here. Not calling `rmfs()` will just result in memory leaks. Although a realistic production C program will always need to avoid memory leaks, i f the user doesn't care about memory leaks in some programs then they just don't have to call `rmfs()` in them.

6 Memory checking and memory errors

We are supplying you (in the project tarfile) with a compiled object file named `memory-checking.o` and its associated header file `memory-checking.h`. These files define two functions, `setup_memory_checking()` and `check_heap()`, which have no parameters or return value. We use them as follows in a few of the public tests (and many of the secret tests) to detect memory leaks and heap corruption:

- `setup_memory_checking()` must be called **once** in a program **before** any memory is dynamically allocated; it sets up things to check the consistency and correctness of the heap. If this function has been called initially, and the program has any memory errors later (such as overwriting beyond the end of an allocated memory region in the heap) the program will crash, consequently failing the test.
- `check_heap()` prints a message indicating whether there is any memory in use in the heap at all.

Some tests will initially call `setup_memory_checking()`, then call `check_heap()` after functions are called on `Ournix` variables and their memory is then cleared. If the functions that are supposed to remove all or part of your data structure are not freeing memory properly your program will report that there is some allocated memory in the heap– meaning a memory leak occurred- causing these tests to fail. (This could also happen if the user calls your functions in an invalid order, for example by calling `mkfs()` on a nonempty `Ournix` variable without calling `rmfs()` on it first), but this is their fault. However, you are using our memory checking functions in your tests and you make an invalid sequence of calls to the functions in these tests, they would fail.)

The facilities we use to perform this memory checking are not compatible with `memcheck` (`valgrind`), because `memcheck` uses its own versions of the memory management functions, that themselves use some memory in the heap. Consequently, **if you run any tests that use our memory checking functions under `valgrind`, the test will appear to fail with a memory leak, even if your code does not have memory leaks.** You can use **either** our memory checking functions, **or** `valgrind`, but both cannot be used at the same time with a program and give correct results. Appendix B describes how to correctly use `valgrind` in this project.

7 Test driver

A number of functions must be called to create filesystems and navigate around them to test the various features of the project. As a result, writing tests to check your code throughly could be time–consuming. To make things easier we have provided 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 situations without writing any test code at all, by typing commands the same way they are input to the UNIX shell.

The driver is supplied in the project tarfile as an object file `driver.o`, since it uses C features that have not been covered in class. It has an associated header file `driver.h`, which must be included by any program that uses it. It provides a single function `void driver(Ournix *filesystem)`. If you link the driver with your project code and call this function, it 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 a following argument, if there is one, to the function. The tenth public test uses the driver so you can see how it is used there. Here is some additional information about the driver:

- The driver creates and uses exactly one `Ournix` variable, while our tests (and your tests not using the driver) could create and use more than one `Ournix` variable.
- The driver will stop reading input and quit if the commands `logout` or `exit` are given, or at the end of the input.
- 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 limits. (These are assumptions our driver makes, **not** assumptions your functions should make or need to make.)

- If an input line with an invalid command is entered the driver will print the message "`Command not found.`" If a valid command is followed by an incorrect number of arguments the driver will print "`Invalid arguments.`"

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

- The driver prints a generic error message "Missing, incorrect, or invalid operand" if any call to your functions returns the error value 0. If you see this when you don't expect it then some function of yours must be returning 0 when it should not be. Write tests **of your own** to figure out which function(s) are returning the wrong value and in what cases.

- 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 `Ournix` parameter. Tests using the driver must first perform `mkfs` before any other filesystem commands, if the test writer wants their code to work properly. (See the input file for the tenth public test that uses the driver an example.)

- To avoid having to retype many commands every time the driver is run, you can redirect standard input from a file containing pretyped commands. (Hint: you used input redirection to run every test in Project #2.) 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 itself is not displayed. A command "`unset verbose`" is also available, although you may not have much need for it.

A Development procedure review

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

Log into the Grace machines and use commands similar to those from before:

```
cd ~/216
tar -zxvf ~/216public/project06/project06.tgz
```

This will create a directory `project06` that contains the necessary files for the project, including the header files `ournix.h`, `memory-checking.h`, and `driver.h`, the public tests, and the `memory-checking.o` and `driver.o` object files. `cd` to the `project06` directory, write your `Makefile` in it, and create a file in it named `ournix.c` (spelled **exactly** that way) that will include the header file `ournix.h`, and in it write the functions whose prototypes are in `ournix.h`. You will first have to write your type definitions in a header file `ournix-datastructure.h`.

Commands similar to those before can be used to run tests and determine whether a public test passes, for example:

```
public01.x | diff - public01.output
```

Because one public test uses the driver, you must redirect input from its corresponding input file when running it:

```
public10.x < public10.input | diff - public10.output
```

As before, the command `submit` from the project directory will submit your project. **Before** you submit you **must** make sure you have passed all the public tests, by compiling and running them yourself. Unless you have versions of all required functions that will at least compile, your program will fail to compile at all on the submit server. (Suggestion– create skeleton versions of all functions when starting to code, that just have an appropriate return statement.) **Before** submitting, also be sure to test your makefile as explained in Section 4, and run `valgrind` on all of the tests (see the information in Section B below). Using uninitialized data– which `valgrind` will detect– can cause C programs to work differently on different machines or different times they're compiled or run. If you don't run `valgrind` on all the tests before submitting you could find that you pass all the tests on Grace but fail some or all of them on the submit server.

A.2 Grading criteria

Your grade for this project will be based on:

public tests	35 points
secret tests	50 points
programming style	15 points

Your code's programming style will be graded, but in this project your makefile will not be graded for stylistic issues. Of course if your makefile doesn't work right your code might not compile on the submit server. And some secret tests might test your makefile.

B Project–specific requirements, and notes

- Be sure to reread the data structure requirements in Section 3 before starting to code.

  To be completely clear, this is repeated here: you **must** use a dynamically–allocated linked data structure for storing filesystems, and cannot use any fixed–size arrays in your data structure.

- **Do not** write code using loops (or recursion) that has the same effect as any string library functions. If you need to perform an operation on strings and there is a string library function already written that accomplishes that task, you are expected to use it, otherwise you will **lose credit**. **You are expected to use the string library functions whenever there is one that can be used.**

- **Do not** cast the return value of the memory allocation functions or you will **lose credit**. Besides being completely unnecessary, in some cases this can mask certain errors in code.

- Remember that the project style says that global variables **may not be used** in projects unless you are specifically told to use them. **You will lose credit for using any global variables.**

- As the project grading policy handout on ELMS says you should use only the features of C that have been covered so far, up through the time the project is assigned (subject to what the project style guide handout says is good style).

- Remember that our checking functions are not compatible with `valgrind`. To use `valgrind` on any public tests, copy the test to another file, edit the copy to **remove** the calls to `setup_memory_checking()` and `check_heap()`, compile the copy with your functions, and run the compiled program under `valgrind`. (As was shown when valgrind was explained in discussion section, you need to compile your program using the `-g` option to use `valgrind`, so you may want to add it to `CFLAGS` in your makefile if your makefile is compiling your own tests.)

- You **cannot** modify anything in the header file `ournix.h` or add anything to `ournix.h`, because your submission will be compiled on the submit server using our version of this file. You cannot write any new header files of your own either, besides the required `ournix-datastructure.h`.

  Your code **may not** comprise any source (.c) files other than `ournix.c`, so all your code **must** be in that file.

- If your code compiles on Grace but not on the submit server either your account setup is wrong, or you modified `ournix.h`, or your `Makefile` is wrong, **and** you did not test it as described in Section 4.

  If your code passes tests on Grace but not on the submit server, remember that uninitialized variables can cause C programs to work differently on different machines or different times they're compiled. In this case **run `valgrind`** to look for any use of variables before they are given a value.

- Be sure to make frequent backups of your project files in a different directory in your course disk space.

- For this project you will **lose one point** from your final project score for every submission that you make in excess of four submissions, for any reason.

- Recall 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 TAs will not look at your code's results on any of the public tests– you must have written your own tests. And the TAs will not help you fix any program bugs unless you have first used `gdb` and `valgrind`. (If you need help using these you can ask the TAs.)

C Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, 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 projects. 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. More information is in the course syllabus – please review it now.

The academic integrity requirements also apply to any test data for projects, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.