CMSC 216 Project #5 Spring 2013

Date due: Monday, April 29, 10:00:00 p.m.

1 Administrative

The TAs' "extra" office hours for this project will run from Thursday, April 25, through Wednesday, May 1.

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 necessary files, will be available sooner. 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 6 for an illustration of what the program should do.

You **must** know how to use the `gdb` debugger, and `valgrind` (`memcheck`), and have used them, to throughly debug your code before asking for debugging help in the TAs' office hours. The TAs will not debug any code for you that you have not already made every effort to debug yourself first.

2 Introduction

In this project you will write a version of the `make` utility, that can build programs, or execute any commands that depend on other things. Since `make` runs programs, this will involve using process control, which has been discussed in lecture. To distinguish the name "`make`" from what you are implementing, we will call your program "`fake`" (i.e., it's a fake implementation of `make`), and we will call the data that it reads a "fakefile", rather than a makefile.

You will write one function to read a fakefile, another function that will print the components of a fakefile that has been read (so you can ensure that it was read correctly), then the heart of the project will be a function that, given a fakefile that has been read and the name of a target, will try to build that target, if it needs to be built, similar to what the real `make` does. Another function will free the memory of a fakefile.

As in Project #3 you will create your own data structure for storing the components of a fakefile, however, there should be fewer choices of data structure to use. We are giving you a header file that contains the prototypes of the required functions, but has no type definitions; as in Project #3 it includes the header file with type definitions that you must create. You may, but are not required to, use linked data structures. Note though that no maximum limit is specified to the size of a fakefile that should be able to be stored, so somehow your data structure must be expandable.

Your "fake" version of `make` will be very simplified, although it will be able to compile many programs that you could write in this course (even itself!). However, it has far less functionality than the real `make`, which is of course to make the project simpler. Besides simplifications, in a few respects its behavior differs from what real `make` would do as well; these differences are also to make the project easier. Although knowing how the real `make` works will be necessary in writing the project, in all cases your program should behave as described here, rather than doing what the real `make` would do, in any case where these would differ.

As also stated below, you **may not use** the C library function `system()` anywhere in your program.

Important!!! To restrict the number of processes your program can create when you run it on the Grace machines (and avoid a fork bomb if it has an error), add the line `limit maxproc 20` in the `.cshrc.mine` file in your home directory before starting to code. If you add it as the last line of `.cshrc.mine` it **must** end with a newline to be recognized by the shell.

3 Fakefile components and syntax

A fakefile consists of or may contain:

targets: A target is a single word, starting at the very beginning of a line in a fakefile, terminated with a colon. ("Word" means a consecutive sequence of non–whitespace characters).

dependency names: A target will be followed by zero or more dependency names, each of which is a single word.

actions: An action is a line whose first character is a tab character, followed by a UNIX command.

comment lines: A comment line is any line whose very first character is a pound sign (`#`). Comment lines should be ignored and have no effect on fakefile processing.

blank (empty) lines: A blank line consists only of its terminating newline character, with no characters preceding it. Blank lines should also be ignored and have no effect on fakefile processing.

Although the terms "dependencies" (or "recipes") are sometimes used with real `make` to refer to a target, its associated dependencies, and its associated actions, in this project we will just call these rules.

3.1 Fakefile syntax

A fakefile will consist of zero or more rules. Comment lines and blank lines may appear before or after (between) rules, but not internal to a rule. As above, comment lines and blank lines should be ignored. Fakefiles cannot contain variables such as `CC` or `CFLAGS` like real makefiles can, or any features of real `make` that were not covered in lecture.

A fakefile target will be followed by zero or more dependency names. A line consisting of a target and its dependencies will be immediately followed by a **single** action line. (No comment line or blank line will intervene.) Although a real makefile may contain multiple actions in any rule, that are executed top–down, and can also have no actions at all (think of the target "`all`" that you have used in makefiles, that has no associated action), to make the project easier a fakefile rule will only have one action, and will always have one action (so its action will be nonempty).

The command to be executed in a fakefile action may be any UNIX command, or a program that we include with the tests. (In fact, it could be a program that other rules in the same fakefile compiled earlier.)

Note that fakefiles are just a subset of real makefiles, so any fakefile that your program can read and process can also be used by the real `make` utility. However, your program's output will be a bit different from what real `make` would do; the differences (described below) are again just to make writing the project easier.

4 Project files, and functions to be written

The `project5.tgz` tarfile in the `216public/project5` directory contains a header file `fakefile.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 `fakefile-datastructure.h` that is not provided. You must write this file, which must contain a definition of the type `Fakefile` that the functions operate upon (as well as any other definitions that it needs). The tarfile also contains the object file `memory-checking.o`, from Project #3. It also includes a header file `split.h` and a compiled object file `split.o` (both header files have been compiled on the Grace machines), described below.

This project would seem to be an ideal situation where you would write a complete, standalone program, rather than some individual functions that our tests would call. After all, `make` is a complete, standalone program. However, we divided the project up into several required functions, for the following reasons. Firstly, if you can't quite finish all of the functions in the entire project you would still be able to get some or most of the functions to work; if the project just involved writing a complete program more would have to be done to to get most of it to work in order to be able to pass any tests. Secondly, some of the tests may do memory checking, to ensure that your functions do not cause any memory leaks and have not caused any errors in the heap; we need to be able to call our memory checking functions before and after your code runs in order to be able to verify this.

All of the functions have a pointer to a `Fakefile` as a parameter; where the descriptions below refer to a function's "`Fakefile` parameter", this should be understood to mean the `Fakefile` that the pointer parameter points to. It is suggested that you write the functions in the order they are described below.

4.1 `void read_fakefile(Fakefile *fakefile)`

This function will be given the memory address of a `Fakefile` variable that is declared in the calling program, and must initialize it as appropriate, read in the components of a fakefile from its standard input, and store the components in some form in its parameter `fakefile`. The result of calling any of the other functions on a `Fakefile` variable before `read_fakefile()` is called on it is undefined. How things are stored depends on what data structures you use. Your function will have to read lines from the standard input until the end of the input is seen. Each line is guaranteed to be 1000 characters or fewer, and of course every separate line must end with a newline (there will be no more than 1000 characters prior to each line's terminating newline, and the standard I/O library functions that read an entire line will append a null character to an entire line read).

As your function reads lines it must ignore comment lines and blank lines. (There is no reason to store them in your fakefile, although you are not prohibited from doing so.) When it sees a line that is not a comment line or blank line, which begins with a non–tab character, it begins the pair of lines comprising a rule. Your function must read the first of this pair of lines, consisting of a target followed by zero or more dependency names, then read the action command on the next line, which will be a line beginning with a tab character. You may use the `split()` function we are providing, described below, to extract the components of these lines. It must then store the components of the rule contained in the two lines in whatever data structure you use to represent fakefiles.

Every call to `read_fakefile()` on a different parameter must initialize the `Fakefile` its parameter points to in such a way that calling `read_fakefile()` on different `Fakefile` variables causes each one to be a **different** fakefile. In other words, calling `read_fakefile()` several times on different `Fakefile` variables will not cause them to share any components.

To make the project simpler you may assume that any fakefile that your function has to read will be syntactically valid. As just two examples, the first line of every rule will have a colon after the target but no colon anywhere else, and every rule will always have an action command. You are encouraged to add reasonable checks for invalid situations anyway while writing the function, so if you inadvertently create incorrect input files you will get useful information about the problem, rather than just incorrect results or a fatal error.

You may assume that `read_fakefile()` will never be called twice on the same `Fakefile` variable, so it does not have to worry about clearing out any existing data in its parameter `fakefile` before reading data into it. It should just assume that `fakefile` has a garbage value or values.

4.1.1 Our function `char **split(char line[])`

To facilitate the process of reading a fakefile we are supplying you with a function `split()`, with prototype above, in the compiled object file `split.o` (its prototype is in `split.h`). It will take a string representing a line of a fakefile, and break it up into its components. Each component, except as indicated below, is a separate word, separated from other words on the line by either whitespace, or by the beginning or ending of the line (the end of the line will consist of its terminating newline). The function will return a dynamically–allocated array of dynamically–allocated strings, each of which is one word of the line. The array will end with a `NULL` pointer, so your code can detect its last element. So what your function can do is to continually read entire input lines, ignore those that are comment lines or blank lines, and when it sees the line beginning a rule (a non–comment and non–blank line that does not begin with a tab character), read two lines from the input, the first of which will be the target followed by the list of dependency names, and the second of which is the associated action command. It can call split on both lines, and store the components of the rule (the words on the pair of lines) in the data structure you create for your fakefile.

Except as noted below, `split()` will ignore (discard) blank spaces and tabs before and between words, including a tab character if it appears at the beginning of a line, so you do not need to handle an action command line any differently; just pass it to `split()`. `split()` will also remove a trailing colon from a word, so you do not need to handle the target on the first line of a rule any differently.

For example, if called on a line (string) consisting of the characters "`main.x: main.o functions.o\n`" (ending in a null character, of course), `split()` will return a dynamically–allocated array with four elements,

which are (in order) the strings "`main.x`", "`main.o`", "`functions.o`", and the fourth element of which is just `NULL` (the three non–`NULL` strings are all dynamically allocated). Note the result would be the same if more than one blank space separated the words. And if called on a line "`gcc main.o functions.o -o main.x\n`" it would return a dynamically–allocated array with six elements, the first of which is the dynamically–allocated string "`gcc`", the fourth of which is the dynamically–allocated string "`-o`", the fifth of which is the dynamically–allocated string "`main.x`", and the last element of which is just `NULL`.

The `split()` function mimics the behavior of the shell in one respect, which is that a double–quoted string is treated as a single argument. Blank spaces and tabs are preserved inside double–quoted strings. For example, if called on a string consisting of the characters "`echo "Bananas are fun."\n`" (with two internal double quotes), it would return a dynamically–allocated array with three elements, which are the strings "`echo`" (the name of the UNIX command mentioned in lecture that prints something), the string "`"Bananas are fun."`" (with quotes and internal blanks as shown), and `NULL`.

The real UNIX shell has many characters that have special meaning, for example backslashes can be used to escape certain special characters, and single quotes have somewhat different effects than double quotes, but to make things simple our `split()` function does not attempt to emulate the shell's behavior in other respects. (See below for more about this.)

4.2 `void print_fakefile(Fakefile *fakefile)`

This function should print the components of its parameter `fakefile`; its primary purpose is for you to be able to ensure that you read and stored the input fakefile correctly. It should print the fakefile in the following manner:

- The rules must be printed in the same order they appeared in the input fakefile as it was read by `read_fakefile()`.
- Each rule is to be printed in the expected way: in two lines, with the target and its dependencies on the first line and the action command on the second line.
- When printing the first line of the rule the target name must start immediately at the beginning of the line, followed immediately by a colon and a single blank space character, then the list of dependency names, with a single blank space between each of them, but no blank space character after the last one. However, if there are no dependency names for the target, a blank space should not be printed after the colon. (No trailing blank space should be printed at the end of a line.)
- When printing the action command line a single tab character must be printed, then all of the components or words in the action, with a single blank space between each of them, but no blank space character after the last one.
- Even if blank lines did not separate the rules in the input fakefile, a single blank (empty) line must be printed after each rule (following the action command), just to make the output more legible. Note that the last action should **also** be followed by a blank or empty line.
- Other than the single blank or empty line printed after each rule, comments and other blank lines that were in the input fakefile should not appear in the output of this function.

4.3 `void clear_fakefile(Fakefile *fakefile)`

This function should **deallocate** any dynamically–allocated memory that is used by the `Fakefile` variable that its parameter `fakefile` points to, destroying the fakefile and all its components in the process. The parameter `fakefile` should use no dynamically–allocated memory at all after this function is called.

Note that the array of strings returned by `split()` is a dynamically–allocated array of dynamically–allocated strings, so all of its memory must be freed somewhere, to avoid memory leaks. You may need to free it in `read_fakefile()`, depending upon how you decide to store the components of a fakefile, or you may need to free it here, but it has to be freed somewhere.

The effect of calling any of the functions after `clear_fakefile()` is called is **undefined**. (As stated above, the effect of calling `clear_fakefile()`, or any other function, **before** `read_fakefile()` is first called on any `Fakefile` variable is also undefined.)

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

4.4 `int make_target(Fakefile *fakefile, const char target_name[])`

This function will perform the heart of the actual processing of trying to build something based on a fakefile that was read in. Using the fakefile `fakefile`, it should attempt to build the target `target_name` in it. The process of doing this, and the function's return value, are described below.

4.4.1 Fakefile processing

Real `make` may be invoked with no command–line arguments (i.e., just by running the single command "`make`"), and in this case will try to build the default target, which is the first one. This project, and your function `make_target()`, have no notion of a default target. It should try to build the target indicated by `target_name` as described below.

The following example fakefile is used for illustration (note it has a comment, and several blank lines):

```
# This is a fakefile, but could also be a makefile!

main.x: main.o functions.o
        gcc main.o functions.o -o main.x

functions.o: functions.c functions.h
        gcc -c functions.c

main.o: main.c functions.h
        gcc -c main.c

clean:
        rm main.o functions.o main.x
```

4.4.2 Process for determining whether to perform a rule's action command

The following is the procedure for determining whether to perform the action command associated with a rule, which is the most conceptually difficult part of the project:

- When calling `make_target()` with a particular name for `target_name`, if that name does not exist (meaning that there is no file in the filesystem on the disk that has `target_name` as its name), and there is also no rule in `fakefile` with that name as its target, it is an error, and `make_target()` should return -1. (Real `make` would print an error saying "No rule to make target" and quit with exit status 2 in this case.)

  (Note this will automatically be the case if `target_name` is an empty string, since there can't be any rule in a fakefile with an empty string as its target.)

- If there is no rule in `fakefile` with `target_name` as its name, but `target_name` does exist (there is a file in the filesystem on disk with that name) it is not an error, but `make_target()` should just do nothing and return 0.

  For example, in the fakefile above, if `main.c` (the same applies to `functions.c` and `functions.h`) does exist and `make_target()` is called with it them as a target, the fact that there is no target with name `main.c` in the fakefile, is not an error, and the call should cause nothing to happen and 0 to just be

returned. (If real `make` is invoked with the name of a file that exists, and has no rule with it as a target, it will also do nothing.)

Notice that even though we may not usually call `make_target()` with user–written files such as `main.c`, `functions.c`, or `functions.h` as an explicit target, `make_target()` may end up calling itself recursively, as described below, with them as targets.

- When `make_target()` is called on a target name, if neither case above applies then there is a rule in the fakefile that has that target. `make_target()` must first recursively try to build each dependency of that rule, in **left to right** order (there may be zero or more dependencies in the rule, and this process may or may not involve performing their associated action commands, as described below.) If any of them return a nonzero exit code, `make_target()` must return that exit code (note that if checking more than one of the dependencies returns a nonzero exit code, the first one in left to right order that does will terminate processing of the remainder). If they return a zero exit code then processing of the remainder of the dependencies continues.

  For example, in the fakefile above, if `make_target()` is called with `main.x` as a target it will call itself recursively on the same fakefile with targets `main.o` and `functions.o`. (These recursive calls will make further recursive calls on `functions.c`, `functions.h`, `main.c`, and `functions.h`; assuming that all of them exist those recursive calls will return 0 without doing anything else, since they exist but there are no rules with them as targets in the fakefile.)

- Following the recursive calls on the dependencies of `target_name` (if there are any), `make_target()` must determine whether it needs to perform the action command associated with the rule that has `target_name` as a target. It must perform the action command if either `target_name` does not exist, or if `target_name` does exist but is **older** than any of its dependency names that exist. (Obviously it cannot be older than any dependency names that do not exist.)

- After recursively processing the dependencies of `target_name` (if there are any), if the action command does **not** need to be executed (because the target exists and is not older than any dependencies that exist) then `make_target()` should just return 0, without doing anything else after that.

- If the action command associated with `target_name` does have to be performed, `make_target()` will have to create a child process to run the command in the rule's action. Note that the child process must terminate **before** `make_target()` can continue doing anything, because `make_target()` needs to use the child process' return value as follows: if the command run in the child process returns any nonzero value then `make_target()` itself must return that value. If the command returns zero then `make_target()` must return zero. (In other words, it must return whatever the action command returns.)

  Note that performing a rule's action command does not necessarily have to create the rule's target, although this is usually what is done in makefiles (and fakefiles). For example, the following rule is valid in a fakefile, even though it creates a file named `mainprogram.x` if the command is executed, rather than `main.x`. (It may be questionable style, but it's valid.)

```
main.x: main.o functions.o
        gcc main.o functions.o -o mainprogram.x
```

  Of course a target like "`clear`" in the example fakefile above (which in makefiles is called a phony target) also does not create a file named `clear`, and is not considered poor style in a makefile.

When `make_target()` performs or executes an action command, that command might produce some output. For example, an action command might be `ls -l`, or `echo "Bananas are fun"`. Any output that it produces will show up in the output of your function when your function creates a process to run the action command.

Unlike real `make`, `make_target()` must also itself produce output when it performs an action command (meaning output not produced by the action, but rather output produced by your function). This is only for the purpose of making the program easier to write, by seeing what the function is doing. When `make_target()`

determines that it has to perform an action command it must print one output line with the target of the action command followed immediately by a colon and a newline, then print the action command, with a single blank space between each word in the action, then print a completely empty line (just a newline). Then it must run the action command, after printing it as described. (Printing the action command first, then executing it, must be followed, to get the correct output order.)

Note that, as mentioned above, `make_target()` will call itself recursively. If any recursive call returns a nonzero exit code (there were three cases described above where a nonzero value could be returned) then `make_target()` must return that same exit code, so when it calls itself recursively it must capture the return value and potentially use it as its return value.

If a rule has no dependencies, only a target name in its first line, such as the "clear" rule in the example fakefile above, and there is a file with the name of the target in the filesystem, the action command will always be skipped, because the file cannot be older than any of its dependencies, since they do not even exist. But if a rule has no dependencies and there is no file with the name of the target, the action command will be executed, because if a target doesn't exist then its action command is always performed.

4.4.2.1 Testing whether a file exists

In order to test whether a file exists you can use the system call `stat()`, which returns information about a file; its prototype is `int stat(const char *path, struct stat *buf)`. Its first argument is the name of the file to be checked, and as you can see it has a pointer to a structure of type `struct stat` as its second argument. If after the call the value of the global variable `errno` is the symbolic constant `ENOENT` then the file does not exist, otherwise it does exist. Note that `stat()` is going to fill in the fields of `buf` (of type `struct stat`) with some values, but in this case the fields of `buf` can just be ignored after the call; we are only interested in the value of `errno` after calling `stat()`.

To call `stat()` you must include `sys/types.h`, `sys/stat.h`, and `unistd.h`. To use the variable `errno` you must include `errno.h`.

Important: as discussed in class, `errno` is set to a nonzero value by some C standard library functions, and by system calls, to indicate errors. However, library functions and system calls do **not** change or reset the value of `errno` when they start. This means that if you make multiple system calls, and `errno` gets set to some nonzero value by one of them, it will appear to be set to that same nonzero value when checked after the remaining system calls (unless one of them changes it again). The caller (this means your code) must reset `errno` to zero before making any system calls that might set its value, so your code to check whether a file exists must look like this:

set `errno` to 0
make call to `stat()`
test whether `errno` is `ENOENT`

4.4.2.2 Testing which of two files is newer

To test whether one file is newer than another one you can also use `stat()`. Its second parameter is a pointer to a `struct stat`, and its fields are filled in with information about the file whose name is passed into the first character string parameter. A `struct stat` has a field `st_mtime` (of type `time_t`, which is an integer type). If you call `stat()` on two files (that both exist), whichever one has a smaller value for `st_mtime` is the older file. In this case the return value of `stat()` may be ignored, and none of the other fields of the `struct stat` arguments make a difference.

4.4.3 Command output and buffering

As mentioned, in the process of executing action commands in the fakefile, whatever output they produce will just appear in the output of your program. Note that since `make_target()` cannot proceed until each command finishes executing, there will be a definite output order. However, when different processes are printing output to the same destination, for reasons that may be a bit more clear later in class, the output may appear in a different order when it is being piped to another command or redirected to an output file,

due to output buffering issues. To avoid this problem, any of our tests that call `make_target()` will turn off output buffering for the output stream `stdout` by making the system call `setvbuf(stdout, NULL, _IONBF, 0)` (notice the underscore). (The alternative would be for you to have to call `fflush(stdout)` right after every `printf()`, which would have the same effect but be more annoying to do.) To avoid inconsistent results and confusion your own tests should also use `setvbuf()` as well.

4.4.4 Special characters

In real `make` every action command is executed by a shell (a subshell is invoked to execute each command). The shell has a variety of characters that have special meaning. As one example, the asterisk `*` is a wildcard character used to refer to multiple files. Consequently, in real `make` an action command like `rm *.c *.h` would remove all source and header files, because the `*` would be expanded or matched against all of the filenames in the current directory. As another example, the shell uses quotes to delimit argument strings, but then removes them from the argument strings. If you type in the UNIX command:

```
echo "Bananas are fun"
```

the shell runs the `echo` command with the single argument "Bananas are fun."– without the quotes– and the output is just:

```
Bananas are fun
```

However, to make the project simpler, action commands should not be run in a subshell; instead, as described, `make_target()` will create a process and run the command directly in it. That means that all the characters that are special to the shell have no effect in action commands in your project. If an action command was `rm *.c *.h` the asterisk characters would not be expanded, and the `rm` command would try to remove two files whose actual names were literally "`*.c`" and "`*.h`"– with the asterisks– which probably don't exist. And for the action command above, the echo utility would print `"Bananas are fun"`– with the double quotes– because nothing would have removed them like the subshell invoked by real `make` would.

To avoid confusion, any of our tests that call `make_target()` and execute commands will just avoid having any characters in the commands that would have special meaning to the shell. That way your program's results will be similar to those of real `make`. This explanation is just to let you know that if you try to write your own tests that have action commands with special characters in them (characters that have special effects in the shell), your results may be confusing because they would differ from what real `make` would produce.

5 Constraints and notes

- The required functions should have no effect if any pointer or array parameter is `NULL` (returning 0 in the case of `make_target()`). Note that this includes its character string parameter as well. Even if a different error condition described above applies, this takes precedence over whatever return value or handling is specified for that situation.
- 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 their standard **error** 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, the entire program should immediately quit, returning the value -1.
- Some of the tests of this project may be, or use, shell scripts (the tests that test `make_target()`, in particular). A shell script is just a collection of shell commands that can be executed like a program, to avoid having to type them anew every time you want to run them. The shell has features to implement conditionals and loops in shell scripts. The reason we may use shell scripts for some tests is that the shell also has features that can be used in scripts to test whether files exist, whether one file is newer than another one, etc., which we may need to do to verify the correct operation of your `make_target()` function. (Besides knowing that it produced the right output, we need to test whether it actually executed the commands or actions in the fakefile that were supposed to be run.)

- It doesn't matter what results are produced if any system calls fail (for example if a child process cannot be created). This is not the same thing as the exit codes returned by commands– Section 4.4 above describes for example that `make_target()` has to return whatever value executing an action command returns, so an action command may not return 0 when it is run, and if it returns, say, -1, that is what `make_target()` should return. But you can handle however you like any cases where system calls do not succeed.

  Note that if you write an invalid command action (syntactically invalid UNIX command) in a fakefile rule, some system call is going to fail. We only intend to test your program with valid command actions (and, as above, syntactically valid fakefiles).

- Note that infinite recursion in a fakefile is not your problem. For example, if the user of your functions wants to write a rule like the following they will get exactly what they deserve (meaning a segfault) for being so very foolish:

```
main.x: main.x
        gcc main.o functions.o -o main.x
```

  Although real `make` would in fact detect a directly or indirectly infinitely recursive makefile, `make_target()` will not.

- Recall that the facilities that we are using to perform memory checking are not compatible with `memcheck`, 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.

- You should write and use helper functions. Some students have been observed to not write nearly enough helper functions. (In most cases, if you have any functions that are longer than 50 lines or so, you very likely should have broken them up using helper functions.) One advantage of using helper functions is they make understanding what your code is doing much easier. Another advantage is that they are easy to test separately, and when you are confident they work, you can just call them from where you need to without worrying about them. It's not easy to debug 20 lines of code in the middle of a 75–line function, but it's usually much more straightforward to test a 20–line standalone function, then call it from a 55-line function once you're sure it works.

  Write your helper functions **first**, for discrete situations you anticipate needing to handle. A few examples of helper functions (these are just suggestions) are a function to test whether a file exists (given a name, which is a character string), a function to compare whether two files exist and their relative times if so, a function to print all the components of the array of strings returned by `split()`, a function that creates a process and runs a command in it, and a function to search for a target in a fakefile. (Of course what this latter function will return depends entirely on how you store the rules of a fakefile.) Depending on the way you implement the project there are probably various other possible helper functions that you could write in addition. To reiterate, write your helper functions before writing code that uses them, and test them before going on to call them from elsewhere.

- To make things a little easier we will provide a `Makefile` that you can use to compile your code for the public tests; it will be included in the project tarfile when the public tests are posted. (You have to do enough in terms of understanding how makefiles work in this project; having to write a `Makefile` would not add that much useful beyond that.)

6 Example

Suppose the current directory contains the fakefile shown above in Section 4.4.1, which is in a file named `Fakefile`, and the files `main.c`, `functons.c`, and `functions.h` that comprise the program that it builds. It also contains three executable programs `example1.x`, `example2.x`, and `example3.x`. `example1.x` and `example2.x` are executable versions of the program shown on the right and the one below, while `example.x` is exactly the same as the program in `example2.c` except that the call `make_target(&fakefile, "main.x")` has been changed to `make_target(&fakefile, "clean")`.

Below the second program we give the output produced by various commands involved in running these programs. For clarity we have added a few blank lines between some of the commands, and the commands themselves that are typed in by the user are highlighted in green, just to make the different sections of output produced by different commands appear more distinct.

example1.c

```
#include <stdio.h>
#include "fakefile.h"
#include "memory-checking.h"

int main() {
  Fakefile fakefile;

  setup_memory_checking();

  read_fakefile(&fakefile);
  print_fakefile(&fakefile);
  clear_fakefile(&fakefile);

  check_memory_leak();

  return 0;
}
```

example2.c

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

int main() {
  Fakefile fakefile;
  int result;

  setvbuf(stdout, NULL, _IONBF, 0);

  read_fakefile(&fakefile);

  result= make_target(&fakefile, "main.x");
  if (result == -1)
    printf("Some target did not exist, and there was no rule to create it.\n");
  else
    if (result != 0)
      printf("make_target() returned %d.\n", result);

  clear_fakefile(&fakefile);

  return 0;
}
```

The sample execution session using the programs and files above is below; note that the commands are executed in order and their output is shown just as if the commands were typed sequentially in a login session.


```
grace6:~: example1.x < Fakefile
main.x: main.o functions.o
        gcc main.o functions.o -o main.x

functions.o: functions.c functions.h
        gcc -c functions.c

main.o: main.c functions.h
        gcc -c main.c

clean:
        rm main.o functions.o main.x


Looks like your code does not have any memory leaks.

grace6:~: ls
Fakefile     example2.x  functions.c  main.c
example1.x   example3.x  functions.h

grace6:~: example2.x  < Fakefile
main.o:
gcc -c main.c

functions.o:
gcc -c functions.c

main.x:
gcc main.o functions.o -o main.x

grace6:~: ls
Fakefile     example2.x  functions.c  functions.o  main.o
example1.x   example3.x  functions.h  main.c       main.x

grace6:~: touch main.c

grace6:~: example2.x < Fakefile
main.o:
gcc -c main.c

main.x:
gcc main.o functions.o -o main.x

grace6:~: touch functions.h

grace6:~: example2.x < Fakefile
main.o:
gcc -c main.c

functions.o:
gcc -c functions.c

main.x:
gcc main.o functions.o -o main.x

grace6:~: example3.x < Fakefile
clean:
rm main.o functions.o main.x

grace6:~: ls
Fakefile     example2.x  functions.c  main.c
example1.x   example3.x  functions.h
```


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/project5/project5.tgz .
tar -zxvf project5.tgz
```

The tarfile will create a directory `project5` that contains the necessary files for the project, including the three header files `fakefile.h`, `memory-checking.h`, and `split.h`; the public tests and associated files (when they are available). You **must** do your coding in your extra course disk space for this class, otherwise we **will not accept** your submission. After extracting the files from the tarfile, `cd` to the `project5` directory, create a file named `fakefile.c` that will `#include fakefile.h`, and write the functions whose prototypes are in `fakefile.h` in that source file. Of course you also have to write `fakefile-datastructure.h`. (Your `fakefile.c` does not need to include `fakefile-datastructure.h`, since it includes `fakefile.h` and `fakefile.h` includes `fakefile-datastructure.h`.) Note the filenames must be spelled and capitalized **exactly** as shown.

We will have to provide information about how to test your code when the public tests are provided.

A.2 Submitting your program

As before, the command `submit` will submit your project. **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 and style grading

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

Results of public tests	45%
Results of secret tests	40%
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.

The only files that will be graded for style are `fakefile.c` and `fakefile-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

- You **cannot** modify the declaration of anything in the file `fakefile.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 `fakefile.c`, so all your code **must** be in that file. You also **may not** add any header files to the project other than `fakefile-datastructure.h`.

- You **may not use** the C library function `system()` in your program. If you do you will receive **no credit** for the project.
- 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 five submissions. You will also **lose one point** for every submission that does not compile, in excess of two noncompiling submissions.

C 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.

