CMSC 216 Project #8 Spring 2022

Date due: Thursday, April 21, 11:59:59 p.m.

1 Introduction and purpose

Different kinds of programs have to run other programs. One example discussed in lecture is a UNIX shell (or the command prompt window in Windows, or a terminal in Mac OS). Another example is the `make` utility, which you should have a good basic understanding of by now. In this project you will write a version of `make`, which can build programs and, more generally, execute any commands that depend on other things. Since `make` runs programs, this will involve using process control. To distinguish the name "`make`" from what you are implementing, we will call your program "`fake`", and we call the files that your program will operate upon "fakefiles" rather than makefiles.

You will write utility functions to perform various operations on fakefiles, such as reading a fakefile, printing the components of a fakefile that has been read, looking up targets in a fakefile, performing actions in a fakefile, etc. The process of trying to build a target in a fakefile is really the heart of the project. We are providing you with a function `make_target()` that does this by making calls to the functions that you will write. To reduce one thing for you to do we are providing a `Makefile` in the project tarfile that you can use to compile your code for the public tests.

Your 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 omits much of the functionality of real `make`, and in a few respects its behavior also differs from that of real `make`; these differences are just to make the project easier.

As in Project #6, some public and secret tests of this project will check for memory leaks or corruption of the heap, so besides getting your functions to produce the right results, you will again need to use `valgrind` to check for memory problems. When your code has a problem (of any type), **don't use the public tests** to try to figure out what's wrong. Write the **smallest** test of your own that shows the problem (you can just copy the public test and remove as much of it as possible while still having the bug). Also experiment with which cases cause the problem and which ones don't. Trying to find a problem in a small program that is testing just one or a few things will almost certainly be much easier.

Before starting to code, look at the public tests to see what they are testing. If you are not passing a public test, look at the problem test or tests to see what they are doing and how. Then write your own simple tests of those things.

Do not use the C library function `system()` anywhere in your code, or you will **not receive credit** for the project.

This project assignment is lengthy because it includes a lot of background information. But the descriptions of the functions you have to write are just three pages of this assignment.

1.1 Extra credit and number of submissions

You can again get extra credit for this project. If you make **only one submission** that passes **all the public tests** you will get **5 extra–credit bonus points**. And if your single submission is made **at least 48 hours before the on–time deadline** you will get **5 additional extra credit bonus points**, for 10 extra credit points total. (Obviously you can't get the second 5 extra credit points if you don't get the first 5, but you can get the first 5 without getting these second 5.)

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 **run `valgrind`**.

2 Fakefile components and syntax

A fakefile consists of zero or more *rules* with blank lines optionally appearing before or after any rule, but not internal to a rule. (A blank or empty line consisting only of a newline is ignored and has no effect on fakefile processing.) Each rule consists of exactly two lines: a *target line* followed immediately by a single *action line*.

A target line consists of a *target* and *dependency names*:

- A target is a single word, starting at the very beginning of the line, ending with a colon. ("Word" just means a consecutive sequence of non–whitespace characters). No space can separate the colon from the target name.
- The target and colon of a target line will be followed by zero or more dependency names. A dependency name is just a single word.

An action line consists of a tab character followed by a single command:

The command can be any UNIX command, or a program included with the tests. (In fact, it could be a program that other rules in the same fakefile compiled earlier.)

Fakefiles are simplified makefiles, which have **only** the features explicitly described here. For example, fakefiles do not have variables such as `CC` or `CFLAGS`, they do not have rules with no action lines like the target "`all`" you have used in makefiles in earlier projects, they do not have rules with multiple action lines, etc.

This example fakefile is used for illustration and explanation in several places below:

```
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
```

3 Functions to be written

This project would seem to be an ideal situation where you would write a complete, standalone program, rather than individual functions that our tests would call. After all, `make` is a standalone program. However the project is divided up into functions because that allows getting credit for passing tests of some functions, even if you can't get them all to work perfectly.

Most of the functions have a pointer to a `Fakefile` (i.e., `Fakefile *`) parameter. Where the descriptions below refer to a function's "`Fakefile` parameter", this means the `Fakefile` variable that the pointer parameter points to.

The effects of all of the functions are undefined if a fakefile contains multiple rules that have the same target name.

3.1 Data structure and data structure requirements

The data you have to store in this project is the components of fakefiles. As in some other projects you will create your own data structure for storing fakefiles. We have given you a header file `fakefile.h` that contains the prototypes of the required functions, assuming an undefined type `Fakefile`. It includes a header file `fakefile-datastructure.h` which you must provide (with its name spelled **exactly** as shown), containing your definition of `Fakefile`.

You could determine what properties and behavior your data structure has to have by reading the descriptions of the functions below, but here is some information about the functionality that your data structure has to support:

- There is no maximum size for a fakefile, so your data structure must not have a fixed size limit (i.e., it must allow a fakefile o have an arbitrary number of rules).
- It **must** use **only** dynamically–allocated memory to store fakefiles. **Otherwise, you will not receive credit** (even if your code passes every test).

  Two examples of dynamically allocated data structures are linked lists and dynamically–allocated arrays whose size is not known at compile time.
- You **cannot** use any fixed-size arrays **anywhere** in your entire data structure. This means there should not be **any square braces (`[]`) anywhere in your header file `fakefile-datastructure.h`**. (This is to ensure that you are not using any fixed–size arrays.) (You can use a fixed–size array where you are reading each line from the input files; this is not in your data structure.)
- Your data structure has to be able to store the rules of fakefiles, and to be able to print a fakefile, with all the rules in the same order they were read from the input (by the function `read_fakefile()` described in Section 3.4).

- Your data structure has to allow looking up a rule based on the name of its target, and returning the number of the rule. Fakefile rules use zero–based numbering, so returning the number of a rule means returning whether it was the zero'th rule in the fakefile, the first one, etc.
- Your data structure has to allow determining and returning how many dependencies a target has, and returning a dependency based upon its rule number and dependency number in that rule. Dependency numbers are also zero–based. For example, in the sample fakefile in Section 2, dependency #1 of rule #0 is `functions.o`, and dependency #0 of rule #2 is `main.c`.
- Of course your data structure has to allow the action lines of rules to be executed, which is the purpose in life of a fakefile.
- Efficiency of your data structure should not matter, meaning even if it is slower to perform operations than another data structure might be, this should not cause you to fail any tests.

3.2 `int exists(const char filename[])`

This function and the next one do not have `Fakefile` parameters. They just return information about files in the filesystem given their filenames. They are needed to be able to determine whether targets are out of date with respect to their dependencies or not.

This function should return 1 if a file named `filename` exists in the filesystem, otherwise, if there is no file with that name, or if `filename` is `NULL`, it should return 0.

To test whether a file exists you could just try to open it for reading and see if that operation failed, but in UNIX there is a better way. The system call `stat()` returns information about a file; its prototype is `int stat(const char *path, struct stat *buf)`. Its first argument is the name of a file. Its second argument is a pointer to a structure of type `struct stat`, which `stat()` fills with information that is not needed here but is used in the next function. `stat()` returns -1 on error, and there are various reasons that it could fail. However, if it returns -1 and sets the global variable `errno` to the symbolic constant `ENOENT`, it means that the file with name `path` does not exist; otherwise (if it does not return -1, or if it does but it does not set `errno` to `ENOENT`) then it means that the file does exist. Note that `stat()` is going to fill in the fields of `buf` (of type `struct stat`) with some values, but they can just be ignored after the call; for this function only the value of `errno` after calling `stat()` matters.

To call `stat()` and declare a `struct 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 the Reek text, some system calls and C standard library functions set `errno` to a nonzero value to indicate errors. However, these functions and system calls do **not** change or reset `errno` when they begin. This means that the caller (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 a call to stat()
if stat() returns −1, test whether errno is ENOENT
```

3.3 `int is_older(const char filename1[], const char filename2[])`

This function should return 1 if `filename1` and `filename2` both exist and `filename1` is older (was created or modified earlier) than `filename2`. Otherwise, or if either parameter is `NULL`, it should return 0.

You can use `stat()` to test whether one file is newer than another one. As mentioned above, its `struct stat` parameter is filled with information about the file whose name is passed into the first string parameter (assuming the file exists). 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. None of the other fields of the `struct stat` arguments are important in this project.

3.4 `Fakefile *read_fakefile(const char filename[])`

This function will need to dynamically allocate memory to store a `Fakefile` variable, initialize it as appropriate, open a file with the name `filename`, read the components of a fakefile from it, store the components in the allocated `Fakefile`, and return the memory (meaning its address). This will require using C's standard I/O library functions (Chapter 15 of the Reek text, which you were supposed to read). **Do not use low–level UNIX I/O system calls to do file manipulation**– this is more difficult, and unnecessary in this project. Just use the C standard I/O library functions covered in Chapter 15 of the Reek text.

If there is no file with name `filename` that can be opened (perhaps it doesn't exist), or if `filename` is `NULL`, the function should just return `NULL` without using any allocated memory. Note that no matter what causes it to return, before doing so the function must close the file if it was opened, otherwise memory leaks could occur, because when the C standard I/O library opens a file it may allocate memory, which is only released when the file is closed. The results of calling any of the other functions on a `Fakefile` variable that has not been returned earlier by `read_fakefile()` are undefined.

Your function will have to read lines from the opened file until the end of the file is seen. There is no maximum limit to the number of lines in a fakefile, but each line is guaranteed to be at most 1000 characters, which does not include its terminating newline. The standard I/O library functions that read an entire line will append a null character to the line read.

This function is the only one that modifies a fakefile, in that it reads and returns one. The other functions produce output or run programs based upon the contents of a fakefile that has been read, but do not modify their fakefile parameter.

As your function reads lines it should ignore blank ones. When it sees a line that is not blank, that will be a target line, to be followed immediately by an action line (starting with a tab character). It should read both these lines. Use the `split()` function we are providing, described in Section 5 below, to extract the components of these lines (either when the lines are read, or later when they are used). `read_fakefile()` must then store the components of the rule represented by the two lines in whatever data structure you use to represent fakefiles; how this is done depends on what data structure you use.

Every call to `read_fakefile()` must create and return a different `Fakefile` variable, which will not share components with any `Fakefile` returned by a different call.

To make things simpler you may assume that any fakefile to be read will be syntactically valid. You are encouraged to add reasonable checks for invalid situations anyway, so if you inadvertently create incorrect input files you will get useful information about the problem, rather than just incorrect results with no explanation, or a fatal error.

3.5 `int lookup_target(Fakefile *const fakefile, const char target_name[])`

This function should search for a rule in its parameter `fakefile` that has target name equal to `target_name` and, if such a rule exists, return its number. Each rule in a fakefile is considered to have a number; as mentioned above the first rule has number 0, the second is number 1, etc. (These are not the same as their line numbers in the fakefile, since rule numbers ignore blank lines, and a rule number is for the pair of lines comprising the rule.) If either parameter is `NULL` or if there is no rule in `fakefile` with target name `target_name` the function should just return -1. Other functions below will use a rule's number after it has been looked up by this function.

For example, if called on the example fakefile above: if `target_name` is `main.x` the function should return 0; it should return 1 if `target_name` is `functions.o`; it should return 2 if `target_name` is `main.o`; it should return 3 if `target_name` is `clean`; and -1 if `target_name` is anything else at all. -1 should also be returned if `target_name` is `NULL`.

How this function will work depends on your data structure, but you obviously have to choose one that allows looking up rules by the names of their targets, and enables returning a number indicating which rule in a fakefile had that target.

3.6 `void print_action(Fakefile *const fakefile, int rule_num)`

If `fakefile` is `NULL` or if `rule_num` is not a valid rule number for `fakefile` this function should just return without doing anything or producing any output. Otherwise it should print the action line of the rule numbered `rule_num` in its parameter `fakefile` (rule numbers were described under `lookup_target()` above). To print the action means to print all of the components or words in the action, with a single blank space between each of them, but neither whitespace nor a tab character should be printed before the first one. A newline should be printed immediately after the last word.

3.7 void print_fakefile(Fakefile *const fakefile)

This function should print the components of its parameter `fakefile` as described below. However, if `fakefile` is `NULL` it should just return without producing any output.

- 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 two lines: the target line followed by the action line.
- When printing the target line, 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 a blank space should **not** be printed after the last one. However, if there are no dependency names for a target, a blank space should not be printed after its colon. (No trailing blank space should ever be printed at the end of a line in a rule.)
- An action line should be preceded by a tab character, then printed as described under `print_action()` above.
- A single blank line must separate every rule (pair of target/action lines) in the printed fakefile, regardless of whether blank lines appeared in the input fakefile when it was first read, or how many of them there were. But a blank line should not be printed before the first rule in the fakefile or after the last one.

3.8 int num_dependencies(Fakefile *const fakefile, int rule_num)

This function should return the number of dependencies that the rule with number `rule_num` has in its parameter `fakefile`. If its parameters are valid the result will be zero or more. But if `fakefile` is `NULL` or if `rule_num` is not a valid number of a rule in `fakefile` then -1 should just be returned.

3.9 char *get_dependency(Fakefile *const fakefile, int rule_num, int dependency_num)

This function should return a pointer to the name of the dependency (a string) with number `dependency_num` in rule number `rule_num` in its parameter `fakefile`. As above, the first dependency in a rule has number 0. For example, in rule 1 of the example fakefile in Section 2 (the rule with target `functions.o`), `functions.c` is dependency number 0 and `functions.h` is dependency 1. However, if `fakefile` is `NULL` or if `rule_num` is not valid (there is no rule with that number in the parameter `fakefile`), or if `dependency_num` is not valid (there is no dependency with that number in rule number `rule_num` in `fakefile`) the function should just return `NULL`.

Note that the function is returning a pointer to a string stored somewhere in your data structure. The function is **not** supposed to allocate new memory for the return value, and (if they know what is good for them) the caller should **not** try to free the returned string after calling `get_dependency()`, because they will be clobbering part of your data structure. (This is their responsibility to avoid, not something you can check for or prevent. All you can do is hope that they are not so deranged as to do this.)

3.10 int do_action(Fakefile *const fakefile, int rule_num)

This function is the crux of fakefile processing. It should actually perform the action (command) of the rule with number `rule_num` in its parameter `fakefile`, and **if the command exited normally** `do_action()` should return the exit status (return value) that the action returned when performed. However, if `fakefile` is `NULL` or if it has no rule number `rule_num`, or if the command **didn't** exit normally when `do_action()` tried to perform it the function should just return -1 without doing anything else

In order to perform the action command of a rule, the function will have to create a child process to execute it. As above, the function should return the child process' exit status if the child process exited normally– this is needed by our supplied function `make_target()` (discussed in Section 4 below), which calls this function. Hence **this function must ensure that the child process terminates before the rest of the program can continue doing anything**, so the child's exit status can be obtained. (None of the tests require that the child process have access to the current environment.)

This function should **not use a pipe**– any output produced by the child process will show up in the standard output.

When `do_action()` performs an action command, the command might produce output. For example, a command might be `ls -l`, or `echo "Sheep are fuzzy"`. The output just shows up in the output of the program that is directly or indirectly calling `do_action()`.

Important! To create the child process, **do <u>not</u> call the `fork()` system call directly**. **Instead** call the function `safe_fork()`, which we have provided in the object file `safe-fork.o` from the previous project. Your code will **<u>not compile</u>** on the submit server if you call `fork()` directly.

3.11 `void clear_fakefile(Fakefile **const fakefile)`

This function should **deallocate** any dynamically–allocated memory that is used by the `Fakefile` variable that its parameter `fakefile` indirectly points to (it is a pointer to a pointer), destroying the fakefile and all its components in the process. The `fakefile` should not use any dynamically–allocated memory at all after this function is called, and the pointer that `fakefile` points to should be `NULL`. If `fakefile` itself is initially `NULL`, or if the pointer that it points to is `NULL`, the function should just have no effect.

After `clear_fakefile()` is called on (the address of) a fakefile pointer variable it is **valid** to call any of the other functions on the pointer, because the pointer will be `NULL` and it is valid to pass `NULL` into any function that has a `Fakefile *` parameter (although the functions won't do anything in that case).

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

4 Our function `int make_target(Fakefile *const fakefile, const char target_name[])`

As mentioned above we provide this function in the object file `make-target.o` (include `make-target.h` to use it). Given a fakefile and the name of a target, it will try to build that target if it needs to be built, similar to what real `make` does, as described below. For instance, if `make_target()` is called with the example fakefile in Section 2 above, and with `target_name` being `main.o`, it will see if `main.o` needs to be rebuilt (based on its relationship to its dependencies) and run the rule's command to rebuild it if so. Tests that call `make_target()` will have to be linked with `make-target.o`. (Note that `make_target()` is called by **tests** of your code– your code should **not** be calling it.) We explain here what `make_target()` does so you can understand how tests work, but we wrote this function so you do not have to.

(Real `make`, when invoked with no command–line arguments, tries to build the first target in a makefile, but our `make_target()` has no default target; the target to be built will be passed into the second parameter.)

4.1 Procedure for determining whether to perform a rule's action command

This is how `make_target()` determines whether to perform the action command associated with a rule:

- If there is no rule in its parameter `fakefile` with `target_name` as its target and no file named `target_name` exists, it is an error and `make_target()` returns -1. (Real `make` would print an error saying "`No rule to make target`" and quit with exit status 2 in this case.)

- If there is no rule in its parameter `fakefile` with `target_name` as its target but a file named `target_name` exists, this is not an error: `make_target()` just does nothing and returns 0 in this case.

  For instance, in the example fakefile in Section 2 above, if `make_target()` is called with `target_name` being `main.c` and `main.c` exists, the fact that there is no target in the fakefile with name `main.c` is not an error, and the call just does nothing and returns 0. (Real `make` does the same thing, but just prints a message.)

  Note that even though it may not be typical to call `make_target()` with user–written files like `main.c` as targets, `make_target()` may end up calling itself recursively with them as targets.

- If neither case above applies then there is a rule in the fakefile that has target `target_name`, and `make_target()` does the following:

  – First, `make_target()` recursively tries to build each dependency of that rule, in **left to right** order (there may be zero or more dependencies, and this process may or may not involve performing their associated actions, as described below). If any of the recursive calls returns a nonzero exit code, make_target() stops and returns that exit code (this will terminate processing of the remainder of the dependencies).

For example, if make_target() is called using the example fakefile in Section 2 above with target main.x, it will call itself recursively 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 these files all exist, the recursive calls will return 0, since there are no rules with those names as targets in the example fakefile.)

- Following the recursive calls, make_target() will perform the action command of the rule that has the target target_name in two cases: (a) if target_name does not exist, or (b) if target_name does exist but is **older** than any of its dependency names that exist (some of them may have just been created).

  If the action command associated with target_name does **not** need to be executed then make_target() just returns 0 without doing anything else. If the action **does** have to be performed, make_target() calls your do_action() to perform it. then make_target() returns whatever value do_action() does.

Unlike real make, make_target() itself produces output when it performs an action command (meaning output not produced by the action, but rather output produced by our function). This is only for the purpose of making it easier to understand what it is doing. When make_target() determines that it has to perform an action command, it prints the target and action of that rule, on two successive output lines, before calling do_action(). The output of some of the public tests illustrates this.

4.2 Command output and buffering

As mentioned, whatever output is produced by executing action commands will appear in the output of a program calling make_target(). Note that since do_action() should not return until an action finishes executing, there will be a definite output order. However, when different processes (like a parent and child) are printing output to the same destination, the output may appear in different orders when it is being piped to another command or redirected to an output file, due to buffering issues. (This refers to redirecting the output of the program calling make_target().) To avoid this, any tests (ours or yours) that call make_target() need to turn off output buffering for the stream stdout by initially calling setvbuf(stdout, NULL, _IONBF, 0) (notice the underscore). Some of the public tests illustrate this.

4.3 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, such as *, which is a wildcard character used to refer to multiple files. This is why an action command in a real makefile can remove multiple files using something like rm *.x. However, your do_action() will not perform actions in a subshell. This means that (other than double quotes, described above) all the characters that are special to the shell have no effect in actions in the project. If an action of a rule was rm *.o *.x the asterisks would not refer to multiple files, and the rm command would try to remove two files whose actual names were literally "*.o" and "*.x", with the asterisks.

To avoid confusion, our tests that call make_target() generally avoid having 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 write your own test fakefiles that have action commands with special characters, the results may differ from what real make would do.

4.4 Other notes about making targets and about fakefiles

- Your code does not have to try to detect or handle a fakefile that has a circular dependency causing infinite recursion. For example, if the user of your functions writes a rule like this, where main.x depends on main.x, they will get exactly what they deserve (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 recursive makefile, make_target() will not. The effects of calling make_target() on such a fakefile are undefined.

- 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 if the command is executed it creates a file named `mainprogram.x`, 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
  ```

  If you call `make_target()` multiple times on target `main.x` using this rule (assuming `main.x` does not exist or is initially older than its dependencies), every call will cause the action to be performed, because `main.x` will not be created by the rule's command so it will never be newer than its dependencies.

  Note that a target like "`clean`" in the example fakefile above in Section 2 (which in makefiles is called a phony target) also does not create a file named `clean`, and is not considered poor style in a makefile.

5 Our function `char **split(const char line[])`

At some point you will have to break up each nonempty fakefile line that is read into its constituent words. To facilitate this we are supplying you with a compiled function `split()`, with prototype above, in the object file `split.o` (with associated `split.h`). It will take a string and break it up into its components, where each component (except as indicated below) is a word, separated from other words by either whitespace, or by the beginning or end of the string. You will have to link `split.o` with your code to use `split()`.

`split()` returns 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 its last element can be detected. (Conveniently, this is the exact format that the execv system calls expect.) `split()` will ignore (discard) blank spaces and tabs before and between words. It will also ignore (remove) a colon that ends a word.

For example, suppose we were to call `char **words= split("I looooove C!!")`. Then a dynamically–allocated array with four elements will be returned, which are (in order) the strings `"I"`, `"looooove"`, `"C!!"`, and `NULL` (the three non–`NULL` strings, and the array itself, are all dynamically allocated). The result would be the same if there were multiple spaces or tabs before, after, or between the words.

`split()` treats a double–quoted string as a single argument, and whitespace is preserved inside double–quoted strings. For example, an array of six elements will be returned by `split("split␣says␣\"hello␣friend\"␣to␣you")`, the third of which is the string `hello␣friend` (the quotes are removed but what's inside them is treated as a single word).

Note that `split()` returns a dynamically allocated array of dynamically allocated strings, so all of its memory must be freed somewhere to avoid memory leaks. Where you need to free it depends upon how you decide to store the components of a fakefile.

You do not need to make a copy of the memory that `split()` returns. It is already going to the work of creating a dynamically allocated array of dynamically allocated strings, whose strings are copies of the words in its parameter, and returning a pointer to it. All you need to do is to make something **point** to the array returned by `split()`, **without** having to make an unnecessary copy if it.

Do not reinvent the wheel and try to write your own `split()` function. (If you make mistakes in it your code may not pass tests.) Just call the one we provided in the object file `split.o`.

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/project08/project08.tgz
```

This will create a directory `project08` that contains the necessary files for the project, including the header and object files `memory-checking.h`, `memory-checking.o`, `safe-fork.h`, `safe-fork.o`, `split.h`, `split.o`, `make_target.h`, and `make_target.o`, the public tests, and our supplied makefile that we are donating to you our of our extreme goodness of heart. `cd` to the `project08` directory and create a header file named `fakefile-datastructure.h`, containing your type definitions. Also in the `project08` directory create a source file named exactly `fakefile.c` (spelled **exactly** that way) that will include `fakefile.h`, `safe-fork.h`, and, `split.h`, and in it write the functions whose prototypes are in `fakefile.h`. All of your code for this project must be in `fakefile.c`. `fakefile.c` will **not** include `memory-checking.h`, only tests that call our memory checking functions will need to include it.

Optionally (as a matter of personal preference)– **only if** you have added correct conditional compilation directives to your `fakefile-datastructure.h` header file as discussed in Appendix B below, `fakefile.c` can also include `fakefile-datastructure.h` as well, just to be explicit, but this is not required, as it includes `fakefile.h`, which includes `fakefile-datastructure.h`.

As in Project #6 you will not be compiling your code from the command line; you will use `make` and our makefile to compile everything. `diff` can be used as before to check whether your code passes a public test, for example:

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

(You can also run `make` from Emacs if desired.)

In discussion section recently your TA showed you a UNIX shell script `run-tests` (which is on Grace so it can be used without changing anything in your account setup), which runs a program on all of its tests at once. Using it will save time compared to running all of the public tests manually.

As before, the command `submit` will submit your project. **Before** you submit you **must** make sure you have passed the public tests, by running them yourself. You must also run `valgrind` on all of the public tests as well. Appendix B below explains how to run `valgrind` for this project, which is different from Project #6.

After you submit you **must** log onto the submit server and see if your program worked there.

Unless your `fakefile.c` has 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.)

A.2 Grading criteria

Your grade for this project will be based on:

public tests	40 points
secret tests	45 points
programming style	15 points

B Project–specific requirements, suggestions, and other notes

- Of course the program could be written without explicitly using process control (e.g., by using the C library function `system()`). But since the entire point of the project is to write a more advanced program using process control, you will **not get any credit on the project** unless you write it as described. So do not call the `system()` function anywhere in your code. Do **not call `sleep()`** anywhere in your code either. Your program will **not even compile** on the submit server if it calls `system()` or `sleep()` anywhere.

  Also do **not** use a loop to try to make a process wait for another one to do something. This is called busy–waiting.

- If any system call fails your program should print some sort of descriptive message saying what didn't work right and where. It doesn't matter what happens after that. The exact wording of the message doesn't matter and it doesn't matter what your code does if this occurs, as long as an explanatory message is printed.

- For simplicity you may assume that any memory allocations performed will always be successful. You are encouraged, but not required, to check whether memory allocations succeed, and to have code handling it in an appropriate way if not. (After all, during development you may have bugs that cause the heap to fill up– if your program just crashes in a cryptic way then, it will be harder to find the bug.)

- One public test, and some secret tests, call our functions from Project #6 in `memory-checking.o` that check the heap, with associated header file `memory-checking.h`. And your code will call functions in both `split.o` and `safe-fork.o`, with associated header files `split.h` and `safe-fork.h`.

- You **cannot** modify the header files provided to you, because your code will be compiled on the submit server using our versions of them.

  Your code **may not** comprise any source (.c) files other than `fakefile.c` so all your code **must** be in that file. You cannot write any header files of your own either other than `fakefile-datastructure.h`.

- Any helper functions that you write in `fakefile.c`, whose prototypes are not in `fakefile.h`, will be "private", so they should be declared `static`, and their prototypes should be placed at the **top of `fakefile.c`**, **not** in `fakefile-datastructure.h` (or `fakefile.h`).

- **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 will **lose credit** if you cast the return value of the memory allocation functions. Besides being completely unnecessary, in some cases this can mask certain errors in code.

- The project style guide says that you **cannot** use the `strtok()` function (which we did not cover).

- Now that the preprocessor has been covered, your `fakefile-datastructure.h` header file **must** have correct conditional compilation directives, as explained in class.

  Note that part of using conditional compilation correctly is that you must spell the name of the preprocessor symbol used for conditional compilation either as described in class and in the lecture slides, or in the form that the Reek text uses, or you may lose some credit.

- Project #7 explained how to use `gdb` with a program that creates child processes.

- As mentioned earlier, you cannot use `valgrind` together with our memory checking functions. We want to make it easy for you to use `valgrind` if you have memory problems in your code, so now that the preprocessor has been covered ,we wrote the tests that use our memory checking facilities to allow you to easily turn off our memory checking by just compiling with the preprocessor symbol `ENABLE_VALGRIND` defined.

  To be able to use `valgrind` (this assumes your `Makefile` is correct):

  – Add a definition of `ENABLE_VALGRIND` (e.g., `-D ENABLE_VALGRIND`) to the compiler options in your `Makefile`. (You also have to add the `-g` option to be able to use `valgrind`, as well as `gdb`.)

  – Force all the tests to be recompiled using `make clean` (or touching all source/header files would also work), and recompile everything (our tests and yours).

  – Now you can use `valgrind` on any test. When you've found the problem and are done using `valgrind`, don't forget to **remove** the definition of `ENABLE_VALGRIND` from the compilation options and recompile everything again, otherwise you will not be running the same versions of the tests as the submit server will, because the submit server will be using our memory checking for some tests. After re–enabling our memory checking by removing -D ENABLE_VALGRIND and recompiling, rerun the tests to confirm that they pass.

  – When running `valgrind`, add the option `--leak-check=no` to turn off checking for memory leaks for any test that is not calling `clear_fakefile()` at the end.

- If you have a problem with your code and have to come to the TAs' office hours for debugging help you **must** come with tests you have written yourself that illustrate the problem (**not** the public tests), what cases it occurs in, and what cases it doesn't occur in. You will need to show the **smallest** test you were able to write that illustrates the problem, so the cause can be narrowed down as much as possible before the TAs even start helping you. To emphasize: **the TAs will not look at your program's results on any of the public tests in office hours. You must have written your own tests to receive any help with your program code.**

  You also **must** have used `gdb` yourself to try to find any bugs, and be prepared to show the TAs how you attempted to debug your program using it and what results you got.

  And **besides** having used `gdb`, you must now have **also** run `valgrind` if your code has any bugs.

  If you aren't sure how to do these things then the TAs can help you with that.

- **Be sure to make frequent backups of your project in a different directory in your course disk space.**
- Recall that the course project policies handout on ELMS 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.

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.