More and more programs today are being programmed as multithreaded applications. The goal of this MP is to give you more practice writing multithreaded applications and to expose common pitfalls that occur while designing a program to work in a parallel manner. Additionally, you will need to make use of synchronization primitives to protect memory shared amongst the threads.

You are given a task of writing an application which will imitate the common make utility. make is a utility that automatically builds executable programs from source code by reading files called Makefiles which specify how to derive the target program. You have encountered Makefiles in CS 241 MPs as well as in your previous undergraduate CS classes and should be familiar with them.

We have provided the code to parse a Makefile and list the dependencies and commands specified in the file. Once the file is parsed, you will need to perform the actions specified by the Makefile following the rules specified later in the docs. Using a fixed pool of threads, you will parallelize this execution process such that all commands are executed after their dependencies are executed.

Before starting you should read the Wikipedia article on [Make](http://en.wikipedia.org/wiki/Make_(software)).

You might also want to look [here](https://www.cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html) for some notes that explain makefiles really well. (They start with some C++ specific details but you can skip to the ‘Now, makefiles’ section. Also, do note that this MP does use makefile macros.)

**THIS IS A HARD MP. WE RECOMMEND THAT YOU START EARLY.**

The first thing you will need to do is to parse the given command-line options. All handling of options should be done using [getopt()](http://www.gnu.org/software/libc/manual/html_node/Getopt.html) . This function will allow you to specify which options are valid and which require arguments. The usage for parmake looks like:

parmake [ -f makefile ] [ -j threads ] [ targets ]

* If a -f makefile is not specified, ./makefile or ./Makefile should be used (in that order), if they exist (see access()). Return a non-zero value if
  + -f makefile is not specified and neither of ./makefile or ./Makefile exists.
  + the file specified by -f cannot be opened or read.
  + you do not need to print an error message in these cases, but it might help you when you are developing.
* If the number of worker threads -j is not specified, the default value of 1 should be used. The j worker threads are in addition to the one main thread (so if j=1, you will have one worker thread, and one main thread).
* The list of targets will always come last and is optional. You will need to save any targets given for later use.
* The man page for getopt() shows an example of how to locate the position of targets within argc.

Expected inputs to the program

As stated above, the input for this MP will be expected to be in the following format:

./path/to/parmake [-f <path/to/makefile>] [-j <positive-integer>] [targets ...]

This means that all the inputs will either be empty or have a list of string targets with optional flags from {-f, -j} in any order, followed by a space and then the parameter for the flag as specified by the input description. If -f exists in the arguments,then it will only be followed a single space, then by a string (which will be a path). Likewise if -j exists, it will only be followed by a single space, then a positive integer. There will not be any extraneous spaces in the inputs.

Note: the -f flag could come before -j or vice-versa, but the targets will always come at the end.

You may use the sample makefiles provided as a gouge of the kind of the tests that the autograder will run. While you should still write test cases to determine the full functionality of your parmake application as described in the documentation. These input will NOT be tested for by the autograder:

* Any filename string that has more than one word
* Invalid flags (example -j 1.1 or -j mydogatemyhomework)
* Targets which are not defined

Note: Relative paths for makefile names will be resolved with respect to the directory parmake is opened in.

Nota Bene: If you use getopt(), then you can pretty much ignore this section.

Next, the main thread should process the makefile. The makefile will always consist of one or more rules of the form:

target [target ...]: [dependency ...]

[command 1]

.

.

[command n]

For example:

rule1: rule2 rule3

commandtoberun withargs

commandtoberun2 withargs

rule2:

othercommand

rule3 rule4:

finalcommand

If you are unfamiliar with the syntax, do not be afraid. We have provided you with a parsing function,parser\_parse\_makefile(). However, you should still take a look at the [Wikipedia page](https://en.wikipedia.org/wiki/Make_(software)) if you do not know how to read a Makefile.

parser\_parse\_makefile() takes the filename, a NULL-terminated array of strings, and three “call-back” functions. The array of strings specify the targets you are planning to run (specified by the arguments to the program, see the first section). The parser will call parsed\_new\_target() on each target, if any,parsed\_new\_dependency() on each target and each of its dependencies, if any, and parsed\_new\_command()on each target and each of its commands, if any.

For example, suppose there is a rule in the makefile called example\_rule, which depends on another rule calleddepend. The parsed\_new\_target() callback function will be called with example\_rule as an argument, and theparsed\_new\_dependency() callback function would be called with example\_rule as the first argument, anddepend as the second.

Those curious of the implementation can view the source in parser.c although this is not necessary.

We have provided an implementation of a queue for you to store rules. It can be viewed in queue.h and queue.c. Please note that **this queue is not thread safe**.

Each rule depends on a set of other rules and files. It is important to note that each dependency is either the name of another rule or the name of a file on disk or BOTH. A rule can be run if and only if all of rules that it depends on have been satisfied.

parmake must satisfy all of the rules needed to build the specified targets correctly and as quickly as possible. To ensure that rules are executed correctly, a rule can only be run once once it’s dependencies are satisfied. Because we want to run as quickly as possible, we need to be running a rule on each worker thread, if possible.

When a rule is ready to be satisfied, we must determine if we actually need to run the rule’s commands. We run its commands if and only if at least one of the following is true:

* The name of the rule is not the name of a file on disk. **Example:**

clean :

rm -rf \*

or

makenewfile:

touch newfile

* The rule depends on another rule that is not the name of a file on disk. **Example:**

clean : take\_backup

rm -rf \*

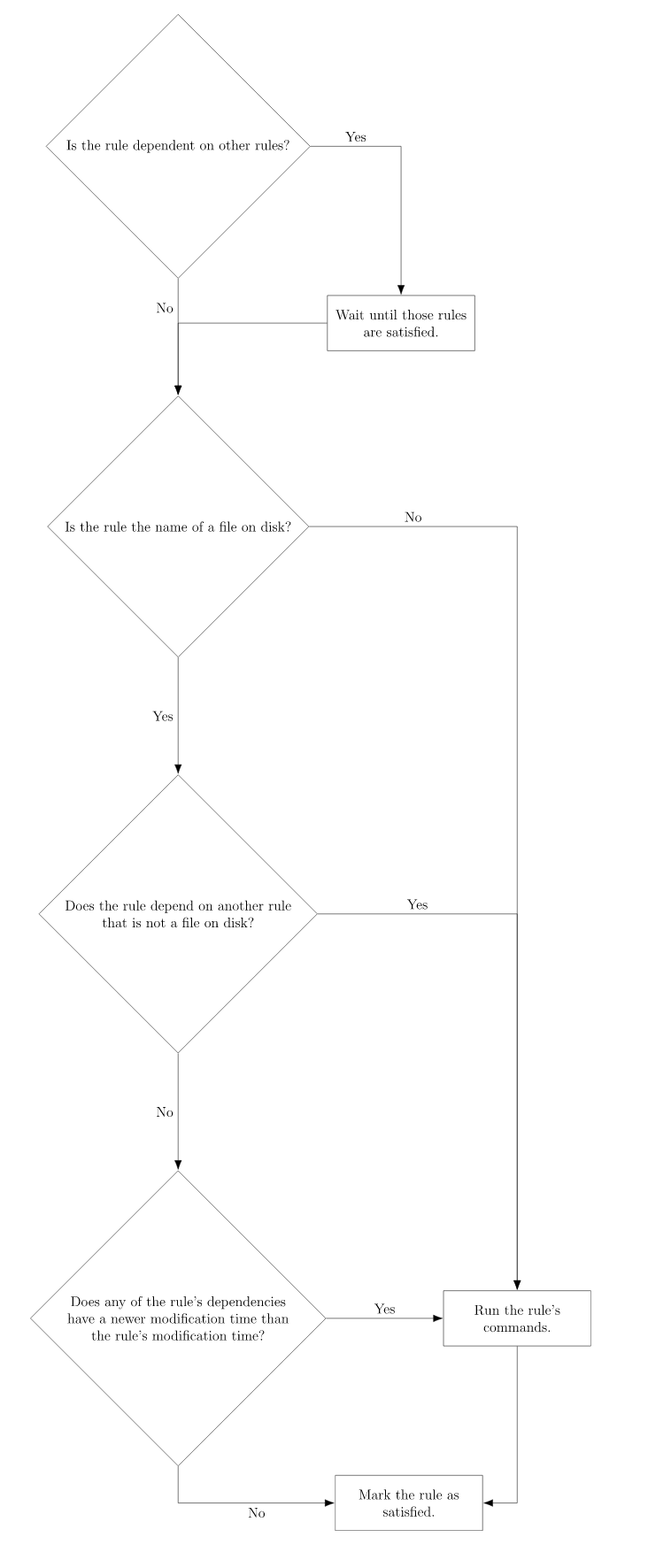
take\_backup :

cp -r \* ../backup

* The rule is the name of a file on disk, and it depends on another file with a NEWER modification time than the modification time of the file which corresponds to the name of the rule.

Once we run a rule’s commands (or decide not to run its commands), we may mark the rule as satisfied.

For your convenience these rules are captured in the following flow chart:



The number of threads running rules is given as the command-line option -j. Each worker thread process rules as they become available. To process a rule, first determine whether its dependencies have been fulfilled. If they have, execute any associated commands. There are several important parallelism requirements:

* You should NOT run any rule unless its dependencies have been met (all dependent rules have been run, see the previous section)
* If a thread is available, and there is at least one rule which is ready to run (all of its dependencies satisfied), the available thread should work on that rule.

Suppose we have makefile:

a: b c

echo A

b: c

echo B

c:

echo C

Running ./parmake should output:

C

B

A

There are many more examples provided in your MP folder.

Notes

* Only make changes in parmake.c
* You can assume all makefiles will be in valid Makefile style syntax as described previously as well as have **no circular dependencies**. You can also expect that there will be no variable references in the Makefiles (e.g. CC=gcc)
* Your are not required to reorder rules that may be run at the same time, to achieve global optimal efficiency. The dependency graph might have natural choke points where one task limits all the others.
* You will receive 0 points if your implementation uses sleep() or busy waiting.
* You must only ever launch T+1 threads, where T is the number of worker threads (+1 comes from the main thread). Do not keep re-spawning new threads for every rule.
* We will try to artificially create spurious wakeups, so think about how you would resolve that.
* GNU make will print out the commands such as ‘echo hello’ before running it. Your parmake application will not be required to do that.
* Remember, the order of the outputs matter for some of the tests!
* As usual, we have provided you with a Makefile which will compile your code in a variety of ways. Unfortunately, you can’t use parmake to compile parmake, because our parser does not support variables and variable expansions.
* To compile in release mode, run make, for debug mode, use make debug.
* ThreadSanitizer
* The provided Makefile also builds a ThreadSanitizer instrumented version of your code. The tsan executable isparmake-tsan. You can run this (instead of parmake) to use the ThreadSanitizer race condition detection tool with parmake. For a tsan example, see [the tsan docs](http://illinois-cs.github.io/tsan)
* **We will be using ThreadSanitizer to grade your code! If the autograder detects a data race, you won’t automatically get 0 points, but a few points will be deducted.**
* (Almost) a reference implementation
* You can use the real version of make to check your code. Note that the real version of make usually prints every command it runs. Your implementation of parmake is not required to print every command it runs, so run make with the flag -s (for silent).
* Example:
* $ ./parmake -f testfile4 -j 2
* This should generate the same output as:
* $ make -s -f testfile4 -j 2