Remodel (Systems Course Project) Fall 2013

Submitted by
Vijay Kumar Pasikanti
(vijaykp@cs.umass.edu)

School of Computer Science
University of Massachusetts Amherst

December 8, 2013

Contents

1 Introduction

`Remodel` is a replacement program for GNU `make`. `make` takes a series of dependencies and builds targets by executing commands. Its use of UNIX timestamps as a happens-before relation means that it can fail to build properly. This is especially true during parallel builds.

`Remodel` also takes a series of dependencies and targets, etc. (grammar below), but unlike make, it will use MD5 hashes to detect new content and provide, together with the dependency chain, a happens-before relation (md5diff(oldA,newA) =¿ everything that depends on A must be rebuilt). It executes all independent (non-conflicting) commands in parallel, using multiple threads.

1.1 Grammar

`remodel` uses a different grammar than `make`. Dependencies can appear in any order. If you execute remodel with no arguments, it should start with the pseudo-target DEFAULT. Otherwise, the root is the argument to remodel, as in remodel foo.o.

```
program    ::= production*
production ::= target '<-' dependency (':'  '"' command '"")
dependency ::= filename (',' filename)*
target     ::= filename (',' filename)*
```

Here's an example that builds the program baz from two source files, foo.cpp and bar.cpp.

```
DEFAULT <- baz
baz     <- foo.o, bar.o:  "g++ foo.o bar.o -o baz"
foo.o   <- foo.cpp :  "g++ -c foo.cpp -o foo.o"
bar.o   <- bar.cpp:  "g++ -c bar.cpp -o bar.o"
```

1.2 Platform and dependencies

- Platform: linux
- Dependencies: `openssl, glibc` (package name `libssl-dev`)
- Compiler: `gcc 4.6`
- Programming Language: `C`

2 Implementation

`remodel` is completely written in 'C' language using standard `glibc` and `openssl` libraries and works for any flavor of Linux OS. `remodel` uses multiple threads defined by the user while building files which are non-dependent on each other.

`remodel` computes MD5 hash string of each source file to identify the changed files. After the build MD5 hash strings are stored in a file `md5hases` within the directory `.remodel` at the root directory level of the project that was built. File `.remodel/md5hashes` is read and all the hash strings are loaded into `remodel` at the beginning of its execution.

2.1 Algorithm

1. Validate arguments.
2. Read and parse `Remodelfile`. Make lists of targets and source files.
3. Read `.remodel/md5hashes` file and load hash strings from the previous build.
4. Generate dependency tree starting from the given `target`.
5. Generate MD5 hash strings of each source file. Compare and identify the files that changed. If the build is successful, the new MD5 hash strings are saved to disk.
6. Mark changed source files and targets that need build in the *dependency tree* and generate a reduced tree by removing the unnecessary targets.
7. Start N *build threads* for parallel builds and a *monitor thread*. N is calculated based on the number of targets that need build using $N = \sqrt{targets}$
8. Starting from the leafs of the *dependency tree* place each target in a *dispatch queue* where they are picked up by *build threads*. *Build threads* send an acknowledgment to `main` program as soon as the build is done for the target it processed.
9. Based on the responses from the *build threads* and *monitor thread* `main` program processes the *dependency tree* to choose the next target to build and places it in the *dispatch queue*.

2.2 Meta files

- `remodel` uses a meta file `md5hashes` to store md5 hash strings of source files. It resides inside the directory `.remodel` in project's root directory.

- The format it uses is `<src file name> <md5 hash string>`. For example:
  ```
  sched.c 105fcbfaf254826be856c5f166f76e21
  network.h 379058d4f4738993f6e37a60f448e36a
  network.c 8586deee068420812704a88a7274f91b
  filesystem.h 20e986d00cadeb329b0143cf95319548v
  ```
- MD5 hashes are read in the beginning of `remodel` execution to compare with the current source file MD5 hashes.
- MD5 hashes calculated in the current build are stored back into the meta file only if the build is successful.

2.3 Threads for parallel builds

Threads are created and run using pthread library. There are two types of threads used in this program viz. 1.*build thread* and 2.*monitor thread.* Each type of thread is described below.

2.3.1 Build Thread

- *Build threads* are created before the start of the building process. Number of *build threads* are N and N is chosen based on the number of targets that needs build. If *targets* are the number of targets that needs build then N is given by the equation $N = \sqrt{targets}$.
- *Build threads* would be running until the whole build process is finished. `main` program will send a kill message to *build threads* to terminate when the build is done.
- As soon as a *build thread* is done with the build command execution for a target, it sends a completion message to `main` using *response queue.*
- Build threads poll on *dispatch queue* looking for targets to build. `main` program dispatches targets to build threads via *dispatch queue. Dispatch queue* is explained in detail in the queues section.

2.3.2 Monitor Thread

- *Monitor thread* is a single thread. It is created when build threads are created and terminated when the build process is finished. The main function of *monitor thread* is to monitor the targets that appear as a dependency for multiple targets.
- As soon as a target is built by a *build thread* the corresponding target that is being monitored is sent to `main` via *response queue*

2.3.3 `main`

- `main` handles responses from *build threads* and *monitor thread* via *response queue.* For each message in *response queue* `main` updates appropriate build completion flags for each target and dependency. As soon as `main` figures out that all dependencies for a target are built, the target is scheduled for build by being placed in *dispatch queue.*

2.4 Queues for inter-thread communication

`remodel` uses queues for message passing across multiple threads to communicate among each other. For `remodel` queues are implemented from scratch using double-link lists. `remodel` uses 3 queues viz. 1. *dispatch queue* 2.*monitor queue* 3.*response queue.* Each queue is described in detail below.

2.4.1 Dispatch Queue

- *Dispatch queue* holds the targets that are ready for build. *Build threads* pick up a targets from *dispatch queue* and execute the build command.
- `main` fills up the queue, *build threads* and *monitor thread* drain the queue.

2.4.2 Monitor Queue

- *Monitor queue* is used by *monitor thread* as a place holder for monitoring targets that are dependencies for multiple targets. *Monitor thread* is the only producer and consumer of the queue.

2.4.3 Response Queue

- *Response queue* is a place holder for responses from *build threads* and *monitor queue.* `main` handles each response from the *response queue. build* and *monitor* threads fill up the queue and `main` removes elements from the queue.

2.5 Cyclic Dependency Check

`remodel` checks for cyclic dependencies, if any, while creating target dependency tree. For example, one of the test cases in `testcases` folder.

```
   [vijaykp@linux /Remodel/testcases/cyclic_dependency]$ ../../remodel
   remodel: building the default target 'DEFAULT'
   remodel: error: cyclic dependency detected for target/file 'snap'
   remodel: info: detected cyclic dependency: DEFAULT -> all -> snap ->
sched.o -> snap
```

```
remodel:  cleaning up
remodel:  remodel took 2 milliseconds to complete
```

3 Usage

3.1 Remodelfile format

1. All target definitions should be place in a file with the name `Remodelfile` or `remodelfile` and it should be placed in the root directory of the project that is being built.

2. Target definitions should written according the grammar given in the introduction section and they can appear in any order with in the file.

3. Each target definition should be written in a single line.

4. One of the targets should be defined as `DEFAULT`. `remodel` uses `DEFAULT` as the default target when none specified in the command.

5. `remodel` parses `Remodelfile` and checks for any cyclic dependencies before proceeding with the build.

3.2 CLI

`remodel` shoudl be run from within the project directory which has the `remodelfile` or `Remodelfile`. `remodel` command takes `target name` as an argument. If no arguments are specified it used `DEFAUTLT` as the argument.
To show help of the command use `-h` or `--help`. For example:

```
[vijaykp@linux test_project]$ ../remodel --help
remodel:  Usage:  remodel [<target_name> | --help | -h]
          Description:  <target_name> is optional.
                        Default value 'DEFAULT'
                        is used when none specified.
          Options:  '-h' and '--help' will print this message
```

4 Build and test `remodel`

1. Change directory to `remodel` directory.

2. Before building `remodel` make sure that `libssl-dev` and `gcc` are installed.

3. Run `make` for building `remodel` for testing.
   Alternatively, use `make debug` command to build a debug build. Debug build is

useful while debugging. It builds the project with gdb flags and enables debug log messages. Make sure to run `make clean` before switching from `debug` to `prod` build and vice versa.

4. Build generates the binary `remodel` .

5. Change to any test directory and run `remodel` either by copying `remodel` binary to test directory or providing the full path to `remodel` .

4.1 Testcases

Test cases are provided in `testcases/` directory. Shell scripts in `testscripts/` can be used for testing test cases in `testcases/` directory.

For example:

```
   [vijaykp@linux  /test]$ git clone https://github.com/vijkp/Remodel
   Cloning into 'Remodel'...
   [vijaykp@linux  /test]$ cd Remodel/
   [vijaykp@linux  /test/Remodel]$ ls
   Makefile Report file.c main.c maindefs.c md5hash.c misc.c queue.c test.sh
   threads.h README Report.pdf file.h main.h maindefs.h md5hash.h misc.h queue.h
   threads.c testscripts testcases
   [vijaykp@linux  /test/Remodel]$
   [vijaykp@linux  /test/Remodel]$ make
   gcc -D NON_DEBUG -c main.c -o main.o
   gcc -D NON_DEBUG -c misc.c -o misc.o
   gcc -D NON_DEBUG -c file.c -o file.o
   gcc -D NON_DEBUG -c maindefs.c -o maindefs.o
   gcc -D NON_DEBUG -lssl -lcrypto -c md5hash.c -o md5hash.o
   gcc -D NON_DEBUG -c threads.c -o threads.o
   gcc -D NON_DEBUG -c queue.c -o queue.o
   gcc -D NON_DEBUG main.o misc.o file.o maindefs.o md5hash.o threads.o queue.o -lssl
-lcrypto -lpthread -lm -o remodel.new
   mv -f remodel.new remodel
   [vijaykp@linux  /test/Remodel]$ cd testcases/testcase-1/
   [vijaykp@linux  /test/Remodel/testcases/testcase-1]$ ../../remodel
   remodel:  building the default target 'DEFAULT'
   remodel:  using (3) threads for parallel builds
   remodel:  thread 3 running "gcc -c sched.c -o sched.o"
   remodel:  thread 2 running "gcc -c network.c -o network.o"
   remodel:  thread 1 running "gcc -c filesystem.c -o filesystem.o"
   remodel:  thread 2 running "gcc -c main.c -o main.o"
   remodel:  thread 3 running "gcc -c drivers.c -o drivers.o"
   remodel:  thread 3 running "gcc main.o drivers.o filesystem.o network.o sched.o -o
snap"
   remodel:  build done!
   remodel:  cleaning up
   remodel:  remodel took 890 milliseconds to complete
   [vijaykp@linux  /test/Remodel/testcases/testcase-1]$
```

5 Source code

- Total lines of code: `2072`
- Total number of files: `14`
- Language: `C`