Remodel - Make done right!

Sneha Shankar Narayan

December 8, 2013

CS630 - Graduate Systems

Contents

Chapter 1

Design and Approach

1.1 Introduction

The ask of the project is to write an updated version of the make utility of linux known as remodel. Make suffers from the fact that it uses unix timestamps to deal with the happens before relationship. To avoid this, MD5 hashing is used to find out if a file has actually changed.

The entire project has been coded in C++.

1.2 System Design

In order to successfully build the software being compiled, remodel does the following: processes input, build the dependency graph, take care of md5 hashing, resolve the dependencies and finally do the actual execution of the compiler statements.

1.2.1 Process input

In order to read and understand the input, remodel parses the input file line by line. The target, command, and the list of dependencies are tokenized and understood by the parse module.

1.2.2 Building the dependency graph

The dependency graph is represented as an adjacency list using vectors. Each node of the graph is of a user defined type which is called "dependencyNode". The target, command and the dependencies of the target are stored in each node. Also various other fields required by the node like whether it has been resolved, where it needs to be rebuilt in a new execution are also stored. Nodes, referred to as leaf nodes are created for the nodes that are only the dependencies of a particular node. Therefore leaf nodes do not have a command or a list of dependencies associated with them, they'd just have the targets. This helps while running the dependency resolution algorithm.

1.2.3 MD5 hashing

The filenames are extracted from the dependency graph and this module goes and computes the MD5 hashes using the files that are provided. This module creates a file .remodel/dependency and stores all the MD5 hashes of the input file in the file. On the second build of the same files, this module reads the file that was created and checks if the hashes that were computed the previous execution are same as the ones computed in the current execution. if so, the corresponding file is not built again. Also if a target depends solely on this file, that target is also not built again. This is ensured by marking the "isBuilt" flag in the dependencyNode of that target.

1.2.4 Resolving the dependencies

The dependencies specified in the dependency graph are resolved in this module. Since the leaf nodes are resolved (as they have no dependencies associated with them) we have a starting point for the algorithm. As long as all the nodes are not resolved, the algorithm goes on checking each target, as soon as all the dependencies are resolved the node is marked as resolved. The iteration at which the node is resolved is noted and sorting this field referred to as depth in the ascending order determines the order of execution. Multiple nodes can be in the same 'depth' and the commands associated with the nodes are the same depth can be executed in parallel.

1.2.5 Execution of the compiler statements

This module takes care of execution of the compiler commands in parallel. Parallelism is taken care of using multiple threads. OpenMP is used to make sure that things run in parallel. A maximum of 4 threads are used to run any set of commands in parallel.

1.3 Functionality provided

The order in which the dependencies are written can be anything, remodel takes care of the resolution. All that needs to be done by the user is to provide the dependencies in the specified grammar which is

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

The input file has to be named "remodelFile" in order for the utility to work.

Chapter 2

Results

When remodel is run on a remodelFile, the results seen on the CLI are the same results seen as when running the compiler commands individually. However if a file has changed from the previous execution, that fact is specified on the CLI.

Chapter 3

Testing

This chapter briefly describes the testing that was done on remodel.

3.1 Functional Testing

3.1.1 Test cases

Following type of tests were run:

- Cyclic dependency
- Naive testcases i.e testcases where the input file has a lot of white spaces.

Specific testcases:

- /test/testsuite1: This is a set of basic tests. The script "/basicTest" can be run to execute the tests in this suite.
- /test/testsuite2: This is a set of tests with more complicated dependency resolution.

3.2 Environment testing

Remodel has been tested on the G++ compiler on the Linux platform.

Chapter 4

Setup

This chapter describes the setup required to run the profiler

4.1 Requirements

The following libraries are required:

- OpenSSL
- Standard Template Library (STL)
- OpenMP

4.2 Steps to build remodel

- run 'make'. Remodel will get built and the object file is called 'remodel'

4.3 Using remodel

- Copy the executable into the directory where all the files that have to be built are present.
- Specify the dependencies in the grammar described in the previous sections in a file named 'remodelFile'
- Execute './remodel'