Building Things Concurrently

PCLT Project

Bernardo Toninho

6 October, 2022

This project is due on the 20th of October at 08:59.

Building Things Concurrently

Compilers have to turn source code into some form of executable file. Some generate binaries that are executed by actual processors (e.g. GCC, OCaml's optimizing compiler, the Rust compiler, etc), others generate bytecode files that are executed by some virtual machine (e.g. Oracle's Java compiler, the OCaml bytecode compiler, etc). Regardless of the particular kind of output, a compiler must perform a series of analyses on the source code to ultimately produce an executable output.

One of these analyses is to construct a dependency graph of the various modules or packages that a program relies on. The output of this analysis is of particular importance if the language implementation supports what is often called *separate compilation*. Separate compilation allows a compiler to generate executable code on a per-module (the language-agnostic term is typically compilation unit) basis and then combine the separate pieces into the final executable. Separate compilation has many advantages. For instance, if a program depends on many large library modules (which rarely change), a compiler can simply generate code for the (relatively small) user code and link it with the pre-compiled libraries. The user code itself can also benefit from separate compilation when it is spread out over many compilation units, where only the changed compilation units need be recompiled. In a language implementation *without* separate compilation, all the libraries need to be recompiled with the user code every time. It may surprise you to know that many modern languages (e.g. Rust, Go, C++) do not allow for separate compilation.

Another advantage of separate compilation is that it enables a compiler to generate code concurrently, which is the focus of this project!

Our “Compiler”

Don’t worry, you won’t have to implement a compiler. Instead, we will abstract away most of the specific details of a compiler and instead assume that our imagined compiler produces a **dependency file**.

Dependency Files

A dependency file specifies a series of build rules in a way that is very much like a simplified Makefile. A rule is a line of the form:

```
target <- dependency1 ... dependencyN;
```

specifying that to generate the object code for `target`, code for `dependency1` through `dependencyN` must first be generated. Each `dependency` must either be the target of a rule or be an existing file (i.e., a “source” file on disk). Note that many `target`s may have shared dependencies.

For a dependency file to be well-formed, it cannot specify a cyclic dependency chain (i.e., the transitive closure of the dependencies of any target cannot include the target itself). The first line of a well-formed dependency file specifies its *root* object, which is an object on which no other `target` in the dependency file can depend on. All other targets in the dependency file must be (potentially transitive) dependencies of the root object. In your work, you may **assume** all dependency files are well-formed.

Note that no order on the rules in a dependency file should be assumed, beyond the fact that the first rule must necessarily specify the special root object.

Dependency Files and Dependency Graphs

A well-formed dependency file specifies the dependency graph of its root object. For instance, the dependency file consisting of:

```
project <- utils parser builder;
utils <- os time;
parser <- participle strings os;
builder <- parser;
```

defines a dependency graph rooted at `project` (i.e., nothing depends on `project`). To build `project`, the dependencies `utils`, `parser` and `builder` must be available first. Subsequently, `utils` depends on `os` and `time`; `parser` on `participle`, `strings` and `os`; and, `builder` on `parser`. Some targets have no dependencies at all, such as `os` and `time` – these are the *leaves* of the dependency graph. Internal nodes, such as `parser`, depend on some graph nodes and are dependencies of others.

Since `builder` depends on `parser`, it can only be generated after `parser` has been built. However, we can see that to build the `project`, `utils` and `parser` may be built concurrently, but only after the leaves of the graph (`os`, `time`, `strings`, `participle` and `os`) have been built.

Processing a Dependency File

This project consists of generating a dependency graph from a dependency file, which you must then process (hopefully leveraging the inherent concurrency). The specifics of how you organize the graph processing are up to you (see Grading), but the dependency ordering must be respected (i.e., a target can only be built after its dependencies).

To simulate the actual build operations, we have provided in the `utils` package a `Build` operation that generates a (fake) object file and returns its modification timestamp. This operation will overwrite the file if it already exists. We have also provided functions `Status` and `GetModTime` that check if an object file exists (and returns its last modification time) and return the last modification time of an existing object file, respectively.

Building the Dependency Graph

Once the dependency graph is fully setup, a build of the graph may then take place. Since the build must respect the dependency order, the build should start from the leaves of the graph up to the root. For the purposes of the project, a (re)build is triggered by a call to `utils.Build`.

A leaf whose object file already exists is considered up to date and need not be rebuilt. The remainder of the graph nodes should adhere to an *incremental compilation* policy, where only out of date object files are generated anew. Since separate compilation also enables incremental compilation, you should use the modification timestamp to check whether a call to `utils.Build` is necessary. Specifically, if the timestamp of any dependency is more recent than the timestamp of the target, the target should be rebuilt, otherwise there is no need to issue a call to `Build` for the target.

Overall Setup

While you have complete freedom in choosing how to generate and subsequently process your dependency graph (following the constraints specified above), your overall system should be started by the `builder.MakeController` function, which returns a channel along which one can trigger builds (see `main` and the provided `builder.MakeController` stub for the intended usage). Feel free to add more fields to the `builder.Msg` type.

The actual project executable can run in one of two modes. When started in watch mode, it should recheck whether the dependency graph requires any rebuilds (according to the logic above). Otherwise, a single build cycle is triggered. This basic setup is already defined for you in `main`, but feel free to change the actual printed contents as you see fit.

A simple parser for dependency files is already provided. Check the `parser` package (and `parser/parser_test.go`) for how to use the parser and what structures it returns. Recall that you may assume the dependency file induces a well-formed dependency graph. Your code **does not need** to check for well-formedness.

Suggestions and Hints

- While creating the dependency graph from the dependency file, it will likely be useful to use a `map` whose keys are the various compilation targets.
- You may recall from your algorithms course that a graph traversal that respects the edge-ordering of the graph produces a topological ordering of the graph nodes.
- Note that a dependency may be shared by many compilation targets. This means that while you traverse the parser output you may find the same dependency multiple times. You should structure your program with this in mind (and may even use this fact
for something useful).
- While the provided `builder/MakeController` function refers to a single `leafCh` channel, there will in general be many leaves in a dependency graph. Feel free to change the provided code to use many channels, or "build around" a single channel in whatever creative way you see fit.
- In general, the determination of whether a compilation target needs to be rebuilt requires comparing the timestamps of its dependencies with its own. This suggests that a worker in charge of a given compilation target should *receive* data from the workers in charge of its dependencies and should *send* data (when?) to all workers in charge of compilation units that depend on its target.

Grading

Projects are to be completed preferably in groups of two (at most three), which must match with the groups of the mini-project. The standard plagiarism rules apply and will be enforced.

The project deadline will be on the 20th of October at 08:59 (before the first lecture of the next module), enforced by Github Classroom. You must turn in your code and a **brief** report (add a PDF to the repository), documenting the various design choices in your work. The report should try to address the questions below.

Your code **must not** use locks or mutexes. All synchronization must be done using channels, and Go's channel-based `select`. Waitgroups may be used if you find them appropriate, but only to ensure adequate termination of the program. Any of the techniques seen in lecture may be used (but they need be used).

Your project will receive a better grade according to the following criteria:

- **Correctness:** How is the dependency ordering respected? (Critical!)
- **Worker creation:** How is the dependency graph generated and processed in terms of creating concurrent workers?
- **Worker management:** How are workers organized amongst themselves and in the overall system? How do they signal each other? Are rebuilds triggered only when necessary? Does the solution limit parallelism potential?
- **Data management:** How are data structure accesses managed in order to ensure the absence of data races?
- **Overall code quality**
- **Use of tests to validate your code**

The criteria are not listed in any particular order. The main focus will be on correctness and worker creation/management.