Lecture 33 - Meta Programming

Only 3 lectures left (including this one). I appreciate each and every one of you and all the effort that you have put in - especially through the 2nd half of this class.

Friday there will be a "review" for the final. The Final will be easy - multiple choice - mostly based on the Microcode machine and the lectures we have been having on using tools and system programming.

There is a nice article on the web on the architecture behind QEMU, KVM and virtual machines. Please go and read it. https://binarydebt.wordpress.com/2018/10/14/intel-virtualisation-how-vt-x-kvm-and-qemu-work-together/

Videos

https://youtu.be/VfpPLTKQTyw - Lect-33-2150-pt1-meta-programming-make.mp4
https://youtu.be/8qI-fHSZe_I - Lect-33-2150-pt2-meta-programming-DRY.mp4
https://youtu.be/TWOgg_yj9q8 - Lect-33-2150-pt3-meta-programming-Continuous-Integration.mp4

From Amazon S3 - for download (same as youtube videos)

http://uw-s20-2015.s3.amazonaws.com/Lect-33-2150-pt1-meta-programming-make.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-33-2150-pt2-meta-programming-DRY.mp4
http://uw-s20-2015.s3.amazonaws.com/Lect-33-2150-pt3-meta-programming-Continuous-Integration.mp4

make and other build tools

These days there are a plethora of "build" tools - most of them do not do as much as `make`. `make` uses the observation that the output of a process will usually be later in time than the set of inputs. So if you have:

```
an_output: Input1 Input2 Inpute3 ... InputN
        Do-Dometing
```

then you can decide when to do the `Do-Domehing` when you have an error in the ordering of the output is build from the input. This can get quite complex when there are a lot of inputs!

This can also get quite tedious when you have to track and manage the inputs and how they get built.

Skipping the dependency tree - means full re-builds. Docker is notorious for the time it takes to do full rebuilds. Basically by skipping this rule it oworks like this:

1. You are good - 1 sec
2. You are good - 2 sec
3. You are good - 1 sec
4. You are good - 3 sec
5. You are good - 5 sec
6. You are good - 1 sec
7. Oops - full rebuild and start over - 30 mints

Other systems that tend to ignore this like, “maven” and “eclipse” are notorious for builds that take ½ hour to an hour on a large system.

So how dose `"make "` work internally - there is a system called a “topological sort” that allows it to figure out complex dependency trees. Other systems like Excel and Haskell use this also to figure out how to run stuff in order.

So … `"make "`

By default `make` uses the file “Makefile” or “makefile” or you can specify a “-f” option to get a different file.

The basic structure is :

```
default_rule_name:  Dependency1 Dependency2
        Command Line Stuff to run for this rule


Dependency1: DependsOn1 DependsOn2
        Do Something

Dependency2:
        Do Something 1
        Do Something 2
        Do Something 3
```

So if I have a program `aaa.c` and it includes the file `aaa.h` - but we just want to compile to an object file, then we want this to link with `bbb.c` (the main program)

```
all: bbb

bbb: bbb.o aaa.o
        cc -o bbb aaa.o bbb.o

bbb.o: bbb.c aaa.h
        cc -c -o bbb.o bbb.c

aaa.o: aaa.c aaa.h
        cc -c -o aaa.o aaa.c
```

`make` will also stop when there is an error reported from a command.

Let's put a syntax error into `aaa.c` and see what happens.

`make` also has a large set of default rules (the default rules can be extended also).

So... `make1.mk`

```
all: bbb

bbb: bbb.o aaa.o
        cc -o bbb aaa.o bbb.o

bbb.o: bbb.c aaa.h
aaa.o: aaa.c aaa.h
```

Also works.

Webpack is an example of a node.js build system. It lacks any manful dependency checking. if you want to dependency check - then you have to build that into the code.

Ant - is a Java dependency system. Again it requires writing Java code and is missing any clear dependency analysis. There are a number of dependency add-on's for Ant. One of the reason that Java builds take so long is because things like and tend to rebuild all sorts of stuff that they probably don't need to rebuild.

Go has an internal dependency system. It is a fully modern compiler and Go keeps a tree of all of the pre-compiled functions that you have already compiled. This extensive tree will only rebuild what you need to rebuild when a function changes. Quite often you can re-compile and rebuild a 1,000,000 line program in under 2 seconds.

2000s book - Pragmatic Programmer and Don't Repeat Yourself = DRY

The "Pragmatic Programmer" inspired the creation of languages like Ruby and Python. This principal, DRY, is the most important one in the boot (Lot's of stuff in the book is really good - event 20 years later!

Let's take a look at a build system for a database table.

A table in a database is a collection of rows with data types. Let's say this is a "user" table - it has username and password and the persons real name.

Table: user

Column Name	Column Type	Index	Description
id	uuid	PK	Unique generated ID for this tables row
username	text	UK	the name of the user (usually an email address)
real_name	text	P	persons name
password_enc	text		encrypted password for user

What we need is the SQL command to create the table, the indexes for the table, and the comments on the table and the sample query, and the code that performs select, update, delete, insert on the row of the table.

Let's build a "meta program" that reads in the table above and converts that into all of these.

First the makefile, gen_user.mk

```
all: user_table.sql

user_table.sql : user_table.md gen_user
        ./gen_user user_table.md  >user_table.sql

gen_user: gen_user.o

gen_user.o: gen_user.c gen_user.h
```

now when we type "make" it bulds and runs this program

```
$ make -f gen_user.mk
```

And we can take a look at the output:

```
DROP TABLE if exists "user";

CREATE TABLE "user" (
        "id"            uuid DEFAULT uuid_generate_v4() not null primary key,
        "username"      text,
        "real_name"     text,
        "password_enc"  text,
        "password_enc"  text

);

COMMENT ON COLUMN "user"."id" IS 'Unique generated ID for this tables row';
COMMENT ON COLUMN "user"."username" IS 'the name of the user (usually an email address)
COMMENT ON COLUMN "user"."real_name" IS 'persons name';
COMMENT ON COLUMN "user"."password_enc" IS 'encrypted password for user';

CREATE UNIQUE INDEX "user_1_uk" on "user" ( "username" );
CREATE INDEX "user_1" on "user" ( "real_name" );
```

Similarly we can write a program that will use the same data to build all sorts of table-related stuff (Including the C code to access the table!)

Now if we have 1000 or 5000 tables (yes database schemas have that kind of numbers. An insurance company that I am working with has a schema with 11241 tables) We can put the definition into a directory - and use this tool and a makefile to process all of the tables into all the chunks of code that we need to build the foundation of a project. The last time I worked at a phone company we did this and generated over 6,000,000 lines of code for 27 different projects directly from the table definitions.

With the phone company I had a directory with all the table definitions in it. I used “ls” to put that list into a file, then awk to find all the table definitions (There were other files like stored-procedures). The list of tables then went into a automatically generated Makefile, with the commands to build the tools and run them. With an updated Makefile I ran a set of programs on the table definitions to produce code.

Yes - the process was Makefile - run ls - run awk - generate Makefile - run Make (as a tool) to compile and build output files.

Continuous Integration

Wouldn't it be nice if when we changed code it automatically did a bunch of tests to tell us if our change broke anything. If we had tests for all the existing stuff in the code then we could just re-run the test.

In modern development environments this kind of auto-testing is basally required. Not only that but there is an entire set of systems called Continuous Integration when every time you check in a new change to the code the existing tests all get run - this is done before the code you check in can get combined with the "master" branch.

Good examples of this include CircleCI, TravisCI and gitlab.com has its own "Git Actions" or "GitHub Actions" system so that you can run either their system or add your own Continuous Integration system.

You set up a specification file and every change to the code runs the set of tests.

Tests usually come in a set of "flavors":

1. Unit tests - these are developed by the developer to test small chunks of code in isolation.
2. Integration tests - these usually test large integrated chunks of the system. For example if you have an application with an Application Programming Interface (API) then tests that run the entire API with the Database and Servers all at once would be an Integration Test.
3. Regression test - a test that checks that specific fixes to defects stay fixed.
4. Quality Assurance tests - These often test an entire application running with a "mocked" database. An example of this is "mocha" tests for a front end. You use a simulated browser and a complete back end and run a set of tests.

DevOps - Tools and Dev replacement for ID and Operations

DevOps brings rigger via testing to the entire process of system administration. It also brings scale.

Most Important Meta-Programming tools

Compilers! Take a formally specified language - convert it into assembly language, use an assembler to build it into a binary file, link binary files together into an executable!

Compilers started out in the 1950's with Fortran and Cobol. Both of those languages are still in use today - with many additions and modifications to the languages. There are better languages for some stuff but these are still very good for certain applications. If you go and get a mortgage for a house your process will include Cobol programs at a bank! The CERN colder runs on Fortran!

The 2nd most important are databases. PostgreSQL, Redis, MongoDB and other tools that store and allow us to get access to data are the back end behind most modern computer applications. The

first database was the IMS - Information Management System - developed by IBM for the NASA Saturn V mum missions. Today there are specialized databases like Googles Spanner system that is a high speed distributed - world scale - database that uses atomic clocks in multiple data centers around the world to synchronized data!

If you want to learn about Google Spanner - up close and personal and PostgreSQL - next fall I am teaching a database class where we will implement a clone of MongoDB to run on Spanner and PostgreSQL as back ends. We will use Redis as a high-speed cache for temporary data.

Copyright