NHR FAU

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

HPC Café – Make (Build automation)

Jan Eitzinger, 08.06.2021

High Performance Computing

What is this good for?

Software development is hard!

Aspect of software engineering

Software configuration management

- Identification, control, status and auditing of configuration
- Build management
- Process management
- Environment management
- Facilitate teamwork
- Defect tracing
- …

Topic of today:
Build systems

What are the benefits?

- **Allow to compile software without knowledge about**
  - Toolchain details
  - Source code internals
  - Target system internals

Automation

Portability

- **Reduce build time**
  - Only build required sources for current configuration
  - Only recompile changed source files on rebuilds
  - Enable parallel builds

Speedup

- **Deterministic compilation for reproducible builds**
  - Select and configure a set of external dependencies that is compatible
  - Configure the source files for specific feature sets
  - Select a compatible set of source files

Configuration

Some history and classification

- Earliest build systems: Collection of OS specific shell scripts
- First Make tool (**Stuart Feldman**, Bell Labs 1976): POSIX standard!

Multiple implementations available

- BSD Make
- GNU Make (de-facto standard on Linux and MacOS)
- Microsoft nmake (Part of Visual Studio)

Efforts to replace Make

- Scons (Python application)
- Rake (Ruby application)
- Ninja (Google)
- Apache Ant (Java application)

Build file generators

- GNU Automake
- CMake (Kitware, 2000)
- qmake (QT)

Modern languages bring their own **build tool**: Golang, Rust

Core concept

- Make is controlled by Makefiles
- A Makefile consists of rules:

```
target : prerequisites
<TAB> recipe
            :
```

Name of file to generate

Input file(s) used to create target

Action to carry out

- Target and prerequisites are assumed to be files!
- In some cases targets are not connected to a file: PHONY targets

Simple Makefile

```
edit : main.o kbd.o command.o display.o
    cc -o edit main.o kbd.o command.o display.o

main.o : main.c defs.h
    cc -c main.c –o main.o
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c –o kbd.o
command.o : command.c defs.h command.h
    cc -c command.c –o command.o
display.o : display.c defs.h buffer.h
    cc -c display.c –o display.o

clean :
    rm edit main.o kbd.o command.o display.o
```

- `$ make` will generate the first target in Makefile, in this case `edit`
- `$ make <target>` builds just the specified target

Perform action if target does not exist or a prerequisite is more recent than target

Target with no file and no prerequisites

Using variables (and behind the scenes)

```
OBJECTS = main.o kbd.o command.o display.o
edit : $(OBJECTS)
   cc -o edit $(OBJECTS)

main.o : main.c defs.h
   cc -c main.c
kbd.o : kbd.c defs.h command.h
   cc -c kbd.c
command.o : command.c defs.h command.h
   cc -c command.c
display.o : display.c defs.h buffer.h
   cc -c display.c
clean :
   rm edit $(OBJECTS)
```

Make operates in two phases

1. Read all Makefiles and build dependency graph of all targets and their prerequisites

2. Use data to determine which targets need to be updated and run the recipes necessary to update them

Make it simpler (using implicit rules)

```
OBJECTS = main.o kbd.o command.o display.o
edit : $(OBJECTS)
   cc -o edit $(OBJECTS)

$(OBJECTS) : def.h
kbd.o command.o : command.h
display.o : buffer.h


.PHONY : clean
clean :
   rm edit $(OBJECTS)
```

Prevent target to be omited if a file named clean exists

Make deduces how targets are built

- Find main.c and match C rule
- Use builtin recipe for C:

`$(CC) $(CPPFLAGS) $(CFLAGS) –c`

The **`VPATH`** variable specifies a list of directories that Make should search for targets and prerequisites.

Pattern rules (write your own implicit rules)

Rule to compile x.c files into x.o files

```
%.o : %.c
	$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Automatic variables

Commonly used automatic variables

$@ file name of the target of the rule

$< name of the first prerequisite

$* stem with which an implicit rule matches

How to set variables

- Shell **environment variables** are also valid inside Makefile
- Make sets many **automatic variables**
- **Variable names** can contain function and variable references

- To set a variable if not already set use

```
FOO ?= bar
```

- Shell assignment operator **!=** to set variable to script output

```
file_list != find . -name '*.c'
file_list = $(shell find . -name '*.c')
```

Equivalent function

- Long lines can be split with a backslash (****) character

Two types of variables

Recursively expanded variables

```
CFLAGS = $(include_dirs) –O
include_dirs = -Ifoo –Ibar
```

Expands to:
-Ifoo -Ibar -O

Disadvantages:

- `CFLAGS = $(CFLAGS) –O`
- If functions are referenced in definitions execution will get very slow

Error: Infinite loop!
+= operator is a possible solution!

Simply expanded variables

```
x := foo
y := $(x) bar
x := later
```

Contains values as of the time this variable was defined

Rules for when expansion happens, during parsing or when using a variable:

https://www.gnu.org/software/make/manual/html_node/Reading-Makefiles.html

Functions (Details on usage in DEMO)

- Syntax: `$(function arguments)`
- Functions allow to do portable text processing (and more) in a Makefile

Commonly used functions:

Many functions operate on whitespace-separated words in `text` !

```
$(patsubst pattern,replacement,text)

$(var:suffix=replacement)

$(filter pattern…,text)

$(filter-out pattern…,text)

$(wildcard pattern)

$(error text…) $(warning text…) $(info text…)
```

Short version for replacing file suffixes!

Useful for debugging Make

Conditionals

Remove surrounding whitespaces

```
ifneq ($(CC),gcc)
    libs = $(normal_libs)
else
    libs = $(libs_for_gcc)
endif
```

```
ifeq ( $(strip $(CC)),)
    CC = gcc
endif
ifdef foo
   frobozz = yes
endif
```

Check if variable is set (not if it is empty)

- Conditionals act on a textual level (in contrast to syntactic level)
- Supported variants: **`ifeq, ifneq, ifdef, ifndef, else, endif`**

Requirements for a production Makefile

- **Generic**: No adaption necessary when adding source files
- **Flat directory structure** based on simple naming conventions
- **Configurable locations** of source and header files
- **Automatic dependency generation**
- **Clear output** with focus on warnings and errors
- **Separation of build configuration** and Makefile
- Support for **multiple tool chains** / build configurations
- Multiple **simultaneous builds** possible in same directory
- Support **C, C++ and Fortran**
- Support **mixed language** applications

Demo

- GitHub Repository with Makefile templates

`https://github.com/RRZE-HPC/Makefile-template`

- Real world examples based on above template

`https://github.com/RRZE-HPC/TheBandwidthBenchmark`

`https://github.com/RRZE-HPC/MD-Bench`

`https://github.com/RRZE-HPC/likwid`

`https://github.com/RRZE-HPC/HPCCG-F90`

`https://aoterodelaroza.github.io/devnotes/modern-fortran-makefiles/`

Tool paths and library dependencies

- For **large projects** you may need **specific compiler** or tool versions
- **Non-standard libraries** may be required to build the application
- **Finding** the **correct libraries** and their **configuration** can be tedious

- Those issues are **not automatically addressed** by Make!
- But they **can be solved** within a **Makefile strategy**

Other tools try to fill the gap: GNU autotools, GNU libtool, CMake, …

Script languages and modern languages (Golang, Rust) come with an **integrated package manager** to address this problem!

Isn’t this oldfashioned? What about CMake?

- **CMake** is a popular software for building, packaging and installing software
- **CMake** is not a build system on its own but generates *native* build files
- **CMake** can be seen as a portable sucessor to **GNU autotools**

Features

- Can handle complicated directory structures
- Can locate system-wide and user-specified executables, files and libraries
- Comes with a graphical configuration editor
- Can generate project files for many IDEs as well as build scripts for native build systems

My opinion: CMake adds complexity and introduces problems and does for 90% of projects not solve any pressing problems!

Best practice recommendation

Keep it simple stupid!

- **Make** provides a **robust and portable environment**
- You can find **simple solutions** for most build requirements
- Handle **dependencies** in a **transparent** and **explicit** way
- **Automatic never comes for free!**

Outlook and further information

- **GNU Make** is a build automation tool that **can meet any requirement**
- As always it is **up to you** to use this **powerful tool** in a **sensible way**
- Things not covered in this talk
  - Recursive Make
  - Advanced topics for writing rules and recipes
  - Integration of Make in editors and IDEs
  - Strategies for install and reinstall targets
  - Dealing with archive files

Topics for next HPC-Café
July: KONWIHR + NHR News
August: No HPC-Café!

- The one stop for documentation of Make are the **official info pages**:

https://www.gnu.org/software/make/manual/html_node/index.html