Building Software Systems

- Even small software systems need to use tools to control builds.
- Many, many tools available
- Tools popular with developers often changing, and specific to platform/language.
- We’ll look at a classic tool `make` which is still widely used e.g. Linux kernel
- If you want current alternatives: `cmake + ninja`
- But you should know `make`

`make` allows you to

- document intra-module dependencies
- automatically track of changes

`make` works from a file called `Makefile` (or `makefile`)

A `Makefile` contains a sequence of rules like:

```
target : source1 source2 ...
cmd1 cmd2 ...
```

`Beware`: each command is preceded by a single `tab character`.

Take care using cut-and-paste with `Makefiles`
The make command is based on the notion of **dependencies**.

Each rule in a Makefile describes:

- dependencies between each target and its sources
- commands to build the target from its sources

Make decides that a target needs to be rebuilt if

- it is older than any of its sources (based on file modification times)

---

### Building Multi-module C Program with incremental compilation

```
$ gcc -c -g -Wall world.c
$ gcc -c -g -Wall graphics.c
$ gcc -c -g -Wall main.c
$ gcc -Wall -o game main.o world.o graphics.o
```

---

### Building Large C Program

For systems like Linux kernel with 50,000+ files building is either

- inefficient (recompile everything after any change)
- error-prone (recompile just what's changed + dependents)

- module relationships easy to overlook
  (e.g. graphics.c depends on a typedef in world.h)
- you may not know when a module changes
  (e.g. you work on graphics.c, others work on world.c)
A Makefile for the earlier example program:

```
# Makefile for the earlier example program:
game : main.o graphics.o world.o
    gcc -Wall -o game main.o graphics.o world.o

main.o : main.c graphics.h world.h
    gcc -c main.c

graphics.o : graphics.c world.h
    gcc -c -g -Wall graphics.c

world.o : world.c
    gcc -c -g -Wall world.c
```

Using Make

```
$ make
 gcc -c main.c
 gcc -c graphics.c
 gcc -c world.c
 gcc -o game main.o graphics.o world.o
$ make
 make: 'game' is up to date.
$ vi graphics.h # change graphics.h
$ make
 gcc -c main.c
 gcc -o game main.o graphics.o world.o
$ vi world.h # change world.h
$ make
 make: 'game' is up to date.
$ make
 gcc -c main.c
 gcc -c graphics.c
 gcc -c world.c
 gcc -o game main.o graphics.o world.o
```

Parsing a Makefile in Python

```
def parse_makefile(makefile_name):
    r"""return dict mapping makefile targets to (dependencies, build commands) tuple"
    rules = collections.OrderedDict()
    with open(makefile_name, encoding="utf-8") as f:
        while line := f.readline():
            if not (m := re.match(r"^\S+\s*:\s*(.*)", line)):
                continue
            target = m.group(1)
            dependencies = m.group(2).split()
            build_commands = []
            while (line := f.readline()).startswith("\t"):
                build_commands.append(line.strip())
            rules[target] = (dependencies, build_commands)
    return rules
```

source code for make0.py
The `make` command behaves as:

```
make(target, dependencies, commands):
    # Stage 1
    FOR each D in dependencies
        rebuild D if it needs rebuilding
    # Stage 2
    IF (target does not exist OR any dependency is newer than target) THEN
        run commands to rebuild target
    END
```

### How make Works - Implementation in Python

```python
def build(target, rules, dryrun=False):
    """recursively check dependencies and run commands as needed to build target""
    (dependencies, build_commands) = rules.get(target, ([], []))
    build_needed = not os.path.exists(target)
    for d in dependencies:
        build(d, rules, dryrun)
        build_needed = build_needed or os.path.getmtime(d) > os.path.getmtime(target)
    if not build_needed:
        return
    if not build_commands and not os.path.exists(target):
        print("*** No rule to make target", target)
        sys.exit(1)
    for command in build_commands:
        print(command)
        if not dryrun:
            subprocess.run(command, shell=True)
```

### Make command-line Arguments

If `make` arguments are targets, build just those targets:

```
$ make world.o
$ make clean
```

If no args, build first target in the `Makefile`.

The `-n` option instructs `make`

- to print what it would do to create targets
- but don't execute any of the commands

A different `makefile` name can be optionally specified with `-f`

- to print what it would do to create targets
- but don't execute any of the commands
```python
def main():
    """determine targets to build and build them""
    parser = argparse.ArgumentParser()
    parser.add_argument("-f", "--makefile", default="Makefile")
    parser.add_argument("-n", "--dryrun", action="store_true")
    parser.add_argument("build_targets", nargs="*")
    args = parser.parse_args()
    rules = parse_makefile(args.makefile)
    # if no target is specified use first target in Makefile (if any)
    build_targets = args.build_targets or list(rules.keys())[:1]
    for target in build_targets:
        build(target, rules, args.dryrun)
```

```python
variables = {}
with open(makefile_name, encoding="utf-8") as f:
    while line := f.readline():
        # remove any comment
        line = re.sub(r"#.*", ",", line)
        # check for variable definition
        if m := re.match(r"^\s*(\S+)\s*\=\s*\((.*)\)\\s*$", line):
            variables[m.group(1)] = m.group(2)
            continue
        line = replace_variables(line, variables)

def replace_variables(line, variables):
    """return line with occurrences of $(variable) replaced by variable's value""
    return re.sub(r"\$(.*)\$", lambda m: variables.get(m.group(1), """"), line)
```

```
# string-valued variables/macros
CC = gcc
CFLAGS = -g
LDFLAGS = -lm
BINS = main.o graphics.o world.o

# implicit commands, determined by suffix
main.o : main.c graphics.h world.h
graphics.o : graphics.c world.h
world.o : world.c

# pseudo-targets
clean :
    rm -f game main.o graphics.o world.o
    # or ... rm -f game $(BINS)
```

```
# parsing makefile variables and comments
```

```
variables = {}
with open(makefile_name, encoding="utf-8") as f:
    while line := f.readline():
        # remove any comment
        line = re.sub(r"#.*", ",", line)
        # check for variable definition
        if m := re.match(r"^\s*(\S+)\s*\=\s*\((.*)\)\\s*$", line):
            variables[m.group(1)] = m.group(2)
            continue
        line = replace_variables(line, variables)

def replace_variables(line, variables):
    """return line with occurrences of $(variable) replaced by variable's value""
    return re.sub(r"\$(.*)\$", lambda m: variables.get(m.group(1), """"), line)
```

```
# file to process
```

```
# or ... rm -f game $(BINS)
```

Compiling Python from Sources with `make`

```bash
$ tar xf Python-3.10.5.tar.xz
$ cd Python-3.10.5
$ find . -type f | wc
4302 4304 135014
$ ./configure
... creating Makefile
$ make
gcc ...
...
$ ./python
Python 3.10.5 (main, Jul 28 2022, 10:52:34) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

### make in parallel

The `-jN` option instructs `make` to build dependencies in parallel using up to N parallel processes.

For example an approximately 7x real-time speedup building Python:

```bash
$ make clean
$ time make -j16
...
real 0m13.556s
user 1m55.979s
sys 0m7.663s
$ make clean
$ time make
real 1m19.566s
user 1m15.477s
sys 0m4.032s
```

### Useful other Makefiles functionalities

```
# multiple targets with same sources
stats1 stats2 : data1 data2 data3
  perl analyse1.pl data1 data2 data3 > stats1
  perl analyse2.pl data1 data2 data3 > stats2

# creating subsystems via make
parser:
  cd parser && $(MAKE)
  # assumes parser directory has own Makefile
```