Leibniz Supercomputing Centre
of the Bavarian Academy of Sciences and Humanities
GNU Make ...
Parallel Programming of
High Performance Systems
Basic Usage

What is GNU *make* good for?

**Common Situation**
Larg(er) Software Project:

- several developers,
- many source files,
- different languages,
... 

**Questions**

- How to *automate build process*?
- (Re)*Build* only *necessary parts* during development?
- *Different build options* (Debug, Release, Hardware, Optimization, ...)?

**Solution:**

*GNU make* (cmake, autotools, ... base on it)

"GNU Make Introduction" | M. Ohlerich
Basic Usage
At its simplest ...

```
$ make clean
$ make
g++ -o prog src/main.cxx
```

```
Makefile

.include/ Makefile
src/

.PROXY: clean
prog: src/main.cxx
<TAB> g++ -o prog src/main.cxx

clean:
<TAB> @rm -rf prog
```

```
$ ls -F
include/ Makefile
src/

$ make
g++ -o prog src/main.cxx

$ ls -F
include/ Makefile
prog* src/

$ make clean
```

“GNU Make Introduction” | M. Ohlerich
Basic Usage
(Default) make-Files, reserved Names

- GNUmakefile, makefile, or Makefile in current directory:
  
  ```bash
  $ make
  g++ -O3 -I./include -o src/file1.o src/file1.cxx
  ...
  
  ```

- Makefile with different name (myMakefile):
  
  ```bash
  $ make -f myMakefile
  ...
  ```
Basic Usage

Command Line Options

- Getting help:
  $ make -h
  ...

- Get GNU make version:
  $ make -v
  GNU Make 4.0 ...

- Special target (clean):
  $ make clean
  ...

- Parallel resources available:
  $ make -j 20
  ...

- Variables used in Makefile:
  $ make VAR=2
  ...

- From different directory:
  $ make -C prog-path
  ...

(reursive calls of make)
Basic Usage

Command Line Options

- Get info about variable setting, implicit rules, etc:
  
  \[
  \begin{align*}
  &\$ \text{make -p} \\
  &\quad \ldots \\
  &\$ \text{make -p | grep VERBOSE} \\
  &\$ \text{make VERBOSE=1 -p | grep VERBOSE} \\
  &\quad \text{VERBOSE} = 1
  \end{align*}
  \]

- Fake: prints cmd sequence, but doesn't do something real:
  
  \[
  \begin{align*}
  &\$ \text{make -n} \\
  &\quad \ldots
  \end{align*}
  \]
Basic Usage
Elements - Anatomy of a Make

- **Variables**: similar to BASH shell variables
- **Rules**: targets, prerequisites, shell commands
- **Directives**: control structures, includes and conditionals
- **Comments**: ignored by make, everything after #
Basic Usage

Basic Rules (by Example)

Makefile version 1 (03_MakefileVersion1)

```
.PHONY: clean
prog: src/main.cxx src/A.cxx include/A.h
<TAB> g++ -o prog -I./include src/main.cxx src/A.cxx
clean:
<TAB> @rm -rf prog src/*.o *~ */*~
```

No Gain! All files are still recompiled!
Basic Usage
Basic Rules (by Example)
... a better solution

Makefile version 2 (03_MakefileVersion2)

```makefile
.PHONY: clean
prog: src/main.o src/A.o
  g++ -o prog src/main.o src/A.o
src/A.o: src/A.cxx include/A.h
  g++ -c -I./include -o src/A.o src/A.cxx
src/main.o: src/main.cxx include/A.h
  g++ -c -I./include -o src/main.o src/main.cxx
clean:
  @rm -rf prog src/*.o *~ */*~
```

**Basic Usage**

Explicit Rules ... Syntax

```
target : prerequisites ...
<TAB> shell command
```

- **target**: file name or label
- **prerequisites**: (dependencies) files names or other targets; separated by white space
- **shell commands**: interpreted by `/bin/sh`

**Note! (02_HelloMakeDiffTab)**

Tab character at beginning of command lines is mandatory!
(unless `.RECIPEPREFIX` is set differently (newer make versions))

"GNU Make Introduction" | M. Ohlerich
Shell Commands (01_HelloMake)

- **Commands** → `/bin/sh`, unless `SHELL` is explicitly set
- **Split line:**

  ```
  @echo "Hello"
  \
  && echo "World!"
  ```

- **@Command**: Command **NOT** printed to screen
- **-Command**: Failure is ignored, e.g. `rm FileNotExist`
- **# Command line**: plainly printed to screen as is
Goal = selected target for Update:

$ make src/main.o

must match exactly one of Makefile's targets

Default Goal:
- can be set via .DEFAULT_GOAL inside Makefile
- else, 1st target in 1st rule of 1st Makefile not starting with dot (".")

Note!

Order of explicit rules not significant, except maybe for determining of the default goal
Basic Usage
Explicit Rules ... .PHONY and Empty Targets

.PHONY targets

- .PHONY targets = targets without real file
- possible files with same name as .PHONY target are ignored
- control Makefile behavior; typical .PHONY targets: clean, all

Declare all .PHONY targets as such!

Empty Targets

print:  src/A.cxx src/main.cxx
       @echo $?
       @touch print

Print only source files that changed since last call of
$ make print

"GNU Make Introduction" | M. Ohlerich
Basic Usage

Variables ... by Example

"GNU Make Introduction" | M. Ohlerich

Makefile version 3 (03_MakefileVersion3)

```
PNAME=prog
CXX=g++
INC=-I./include

.PHONY: clean

$(PNAME): src/main.o src/A.o
    $(CXX) -o $(PNAME) src/main.o src/A.o

src/A.o: src/A.cxx
    $(CXX) -c $(INC) src/A.cxx

src/main.o: src/main.cxx
    $(CXX) -c $(INC) src/main.cxx

clean:
    @rm -rf $(PNAME) src/*.o *~ */*~
```
Variable names **case-sensitive**: `a_bla1 ≠ A_bla1`

- `${variableName}` or `$(variableName)` for substitution

- Variable referenced in **targets**, **prerequisites**, **commands**, most directives, and new variable values
Basic Usage
Variables ... User Defined (Recursively Expanded)

Example (Definition of Recursively Expanded Variables)

```plaintext
name = value
```

variables stored as verbatim text; not expanded until referencing

Advantage
Order of definition NOT significant

Problems

- `name = ${name} anotherValue ⇒ infinite recursion!`
- Function referenced in definition of such variables are executed every time the variable is expanded
  
  *make* execution much slower/unpredictable

"GNU Make Introduction" | M. Ohlerich
Basic Usage

Variables ... User Defined (Simply Expanded)

Example (Definition of Simply Expanded Variables)

```plaintext
name := value
```

value of such variable scanned once

Advantages

- No infinite recursion
- No slow down of *make*
- Better style
Basic Usage
Variables ... User Defined (Appending)

Example (Simply Expanded Variables)

```plaintext
name := value
ame += anotherValue
# is equivalent to
name := value
name := ${name} anotherValue
```

Example (Recursively Expanded Variables)

```plaintext
name = value
name += anotherValue
# is equivalent to
name = value
temp = ${name}
name = ${temp} anotherValue
```
except that temp is not needed using "+="
### Basic Usage

#### Variables ... Overiding

- `make` transforms BASH environment variables into `make` variables with same name/value.
- Variable defined in *Makefile* $\Rightarrow$ any environment variable of same name ignored.
- Variables are overridden on Command Line:
  
  ```
  $ make PNAME=bla -f Makefile
  ```
- Prevent any overriding by (inside *Makefile*)
  
  ```
  override variable = value
  ```
  works also for `"=''` and `"+="` operators.

"GNU Make Introduction" | M. Ohlerich
Similar to BASH wildcards

* Matches any string, including the null string

? Matches any single character

[...] Matches any one of the enclosed characters

~ ~/ denotes your HOME directory

~john/ denotes John's HOME directory
Basic Usage

Searching Directories for Prerequisites

- **VPATH**: directories to look for prerequisites
- Rules can be written as if all prerequisite files in current directory
- **VPATH**: colon- or space-separated list of directories

Example

Suppose `foo.c` is found in `./src`

```
VPATH = src
foo.o : foo.c
```

Caution! (03_MakefileVersion6)

Object files created in CURRENT directory!
### Basic Usage

Variables ... used by Built-in Implicit Rules

**Variables for Compilers**

- `{FC}` Fortran Compiler
- `{CC}` C Compiler
- `{CPP}` C Preprocessor
- `{CXX}` C++ Compiler

**Variables for Flags**

- `{FFLAGS}` Flags for Fortran Compiler
- `{CFLAGS}` Flags for C Compiler
- `{CPPFLAGS}` Flags for C Preprocessor
- `{CXXFLAGS}` Flags for C++ Compiler
- `{LDFLAGS}` Flags for Linker

**Automatic Variables (most important)**

- `@` File name of target of rule
- `<` Name of first prerequisite
- `??` Names of all prerequisites newer than target
- `^^` Names of all prerequisites; space separated

"GNU Make Introduction" | M. Ohlerich
Basic Usage

Implicit Rules ... by Example

Makefile version 4 (03_MakefileVersion4)

```
PNAME=prog
CXX=g++
INC=-I./include
OBJJS=src/main.o src/A.o

.PHONY: clean
$(PNAME): $(OBJJS)
    $(CXX) -o $@ $^  
%.o: %.cxx include/A.h
    $(CXX) -c $(INC) -o $@ $<

clean:
    @rm -rf $(PNAME) $(OBJJS) *~ */*~
```

"GNU Make Introduction" | M. Ohlerich
Basic Usage
Implicit Rules

Syntax for implicit Rules

```
target-pattern : prerequisites
<TAB> shell command
```

target-pattern contains exactly one %!

Note!
Order of appearance of rules in Makefile important!
Basic Usage
Functions by Example

Makefile version 5 (03_MakefileVersion7)

```
PNAME = prog
...
SRC = $(wildcard src/*.cxx)
OBJS = $(SRC:.cxx=.o)
# same as OBJS = $(patsubst %.cxx,.o,$SRC)
.PHONY: clean
bin/$<$(PNAME): $(OBJS)
  @mkdir -p bin
  $(CXX) -o $@ $^
%.o : %.cxx A.h
  $(CXX) -c $(INC) -o $@ $<
...
```

"GNU Make Introduction" | M. Ohlerich
Basic Usage

Functions ... for String Substitution and Analysis

- $(subst \text{from}, \text{to}, \text{text})$
  replace each occurrence of \text{from} in \text{text} by \text{to}

- $(patsubst \text{pattern}, \text{repl}, \text{text})$
  finds whitespace-separated words in \text{text} matching the \text{pattern}
  and replaces with \text{repl}; \% in \text{pattern} is a wildcard;
  may also appear in \text{repl}

- $(filter \text{patterns}..., \text{text})$
  returns all whitespace-separated words in \text{text}, which
  match any of whitespace-separated \text{patterns}

- $(filter-out \text{patterns}..., \text{text})$
  does exact opposite of \text{filter}
Basic Usage

Functions ... illustrating Examples

$\texttt{(subst}\ ee,\ oo,\ \textit{feet})$

$\texttt{== foot}$

$\texttt{(patsubst}\ %.o,\ %.f90,\ \textit{foo.o})$

$\texttt{== foo.f90}$

$\texttt{(filter}\ %.f90,\ \textit{foo.f90}\ \textit{bar.o})$

$\texttt{== foo.f90}$

$\texttt{(filter-out}\ %.f90,\ \textit{foo.f90}\ \textit{bar.o})$

$\texttt{== bar.o}$
Basic Usage

Functions ... more

- $(wildcard pattern)
  request explicitly wildcard expansion for file names; pattern may contain BASH wildcard characters "*", "?", "[...]"

- $(foreach var, list, text)
  similar to BASH's for or Perl's foreach

- $(call variable, arg1, arg2, ...)
  expand appropriately defined variable as function

- $(shell command)
  execute command in a shell and process its output

- Many more functions available ⇒ see GNU make user guide!
Basic Usage
Conditionals ... by Example

Makefile version 5 (03_MakefileVersion8)

PNAME=prog
CXX=g++
ifdef INTEL
   CXX=icpc
endif
INC=-I./include
OBJJS = $(patsubst %.cxx,%.o,$(wildcard src/*.cxx))
...

$ make INTEL=1
or
$ export INTEL=1 && make

"GNU Make Introduction" | M. Ohlerich
Conditionals control what make executes/ignores.

**Example (Syntax of Simple Conditional)**

```make
ifeq (arg1, arg2)
   # execute if arguments are equal
endif
```

All variable references in both arguments are expanded first, then arguments are compared.

**Example (Syntax of another Simple Conditional)**

```make
ifneq (arg1, arg2)
   # execute if arguments are NOT equal
endif
```
Basic Usage
Conditionals

Example (General Syntax of Conditionals)

```
conditional-directive
   # execute, if condition fulfilled
else conditional-directive # available since v3.81
   # execute, if condition fulfilled
[...]
else
   # execute else
endif
```

```
conditional-directive may be one of
```

- ifeq (arg1, arg2)
- ifndef arg
- ifndef (arg1, arg2)
- ifdef arg
- ifndef arg

"GNU Make Introduction" | M. Ohlerich
Basic Usage

Include Directive

Syntax

```
include Make.inc
```

- useful to switch between e.g. compilers (incl. their flags, options, libraries, ...) or hardware
- can be combined with conditionals (*make* command line switches or environmental variables)

Example

Please, have a look in the **PTScotch Library**, or other GNU suite programs!
04_Exercise

Exercise

Create Makefile(s)!
Compile program named prog into bin of top-level directory!
main depends and A.h and B.h.
Class B should become a library, located in lib of the top-level directory!

Hint
How to build dynamic libraries under Linux using gcc:


"GNU Make Introduction" | M. Ohlerich
Always a solution ...
If headers are modified:

\$ make clean && make

Disadvantage: All source files have to be recompiled!
Or ... (05_DependenciesExample1)

PNAME := prog
CXX := g++
CXXFLAGS := -O3 -I./include
SRCS := $(wildcard src/*.cxx)
OBJJS := $(patsubst %.cxx,%.o,$(SRCS))
$(PNAME) : $(OBJJS)
   $(CXX) -o $@ $^
src/main.o: include/A.h include/B.h
src/A.o: include/A.h
src/B.o: include/B.h
%.o : %.cxx
   $(CXX) -c -o $@ $(CXXFLAGS) $<
...

"GNU Make Introduction" | M. Ohlerich
DEPDIR := .d
$(shell mkdir -p $(DEPDIR)/src >/dev/null)
DEPFLAGS = -MT %@ -MMD -MP -MF $(DEPDIR)/%*.Td
COMPILE.cc = $(CXX) $(DEPFLAGS) $(CXXFLAGS) -c
POSTCOMPILE = mv -f $(DEPDIR)/%*.Td $(DEPDIR)/%*.d

%.o : %.cxx
    $(COMPILE.cc) -o $@ $<
    $(POSTCOMPILE)
.PRECIOUS: $(DEPDIR)/%.d
-include $(patsubst %,$(DEPDIR)/%.d,$(basename $(SRCS)))
### Debugging existing Makefile or Variable Settings

$ make -p  
$ make -n

in combination with pipes and grep etc.

### Debugging specific Variable Settings (06_Make_DEBUG)

insert rule into Makefile:

```
print-%:
  @echo \$*=$($*)
```

and call for instance:

```
$ make VAR=1 print-VAR
```
GNU make user manual:
or in PDF form:
Specifically: Quick Reference in Appendix A

GNU make error messages (Appendix B)