

Make: an automation building tool

(CSCB09 – Software Tools & Systems Programming)

Marcelo Ponce

Week 05 – Winter 2022

Department of Computer & Mathematical Sciences
University of Toronto Scarborough

Today's class

Today we will discuss the following topics:

- Quick overview of IO.
- Review of Modularity and Automation.
- The 'make' tool.
- `make` `makefiles/rules/patterns/...`
- `make` examples.

File I/O

File I/O

File systems

- It's where we keep most data.
- Typically spinning disks
- Logical structure: directories, subdirectories and files.
- At the level of the OS, these are abstracted as *inodes*.
- On disk, these are just blocks of bytes.
- Each I/O operation (IOPS) gets hit by *latency*.

File I/O

What are I/O operations, or IOPS?

- **Finding a file (ls)**

Check if that file exists, read metadata (file size, date stamp etc.)

- **Opening a file:**

Check if that file exists, see if opening the file is allowed, possibly create it, find the block that has the (first part of) the file system.

- **Reading a file:**

Position to the right spot, read a block, take out right part

- **Writing to a file:**

Check where there is space, position to that spot, write the block.

Repeated if the data read/written spans multiple blocks.

- **Move the file pointer (“seek”):**

File system must check where on disk the data is.

- **Close the file.**

Why it matters: disk access rates over time

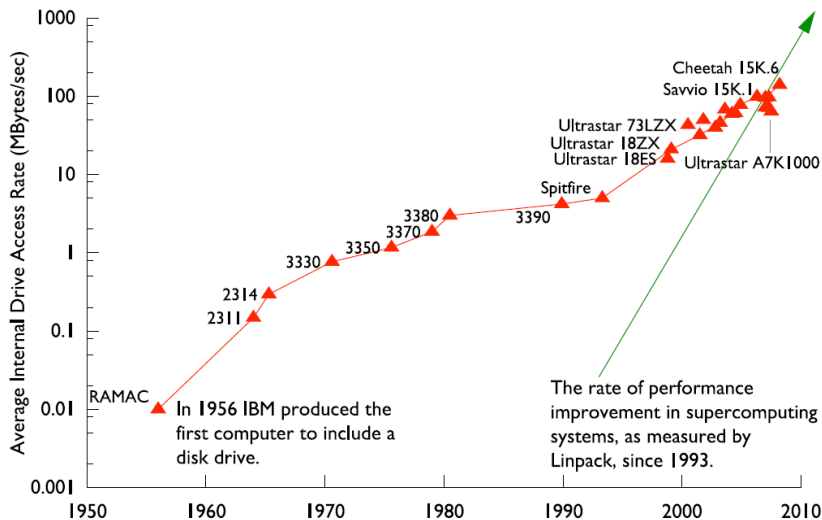


Figure by R. Freitas and L Chiu, IBM Almaden Labs, FAST'10

I/O-aware performance tips

Do's

- Write binary format files Faster I/O and less space than ASCII files.
- Use **parallel I/O** if writing from many nodes
- Maximize size of files. Large block I/O optimal!
- Minimize number of files. Makes filesystem more responsive!

Dont's

- Don't write lots of ASCII files. Lazy, slow, and wastes space!
- Don't write many hundreds of files in a 1 directory. (file locks)
- Don't close files between small reads or writes (no: open, write, close, open for append, write, ...)
- Don't write many small files ($< 10\text{MB}$). System is optimized for large-block I/O.

Binary I/O

- `fgetc` reads characters,
`fputc` writes characters.
- `fread` and `fwrite` operate on bytes.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

read `nmemb * size` bytes into memory at `ptr` returns number of items read

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

write `nmemb * size` bytes from `ptr` to the file pointer `stream` returns number of items written

Examples

```
/* write an integer to the file given by fp */
int num = 1999999;
n = fwrite(&num, sizeof(num), 1, fp);

/* read an integer from the file given by fp */
int num;
n = fread(&num, sizeof(num), 1, fp);
```


File formats

Formats

- ASCII
- Binary
- MetaData (XML)
- Databases
- Standard libraries (HDF5, NetCDF)

ASCII vs. Binary

American Standard Code for Information Interchange

Pros

- Human Readable
- Portable (architecture independent)

Cons

- Inefficient Storage
- Expensive for Read/Write (conversions)

Native Binary

Pros

- Efficient Storage
- Efficient Read/Write (native)

Cons

- Have to know the format to read
- Portability (Endianness)

ASCII vs. binary

Writing 128M doubles

Format	/scratch (GPFS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173 s	174 s	260 s
Binary	6 s	1 s	20 s

Benchmark done on GPC cluster – <https://www.scinethpc.ca/gpc/>

Syntax

Format	C	C++	Python
ASCII	<pre>f=fopen(name,"w"); fprintf(f,...);</pre>	<pre>ofstream f(name); f << ... ;</pre>	<pre>f = open(name, "wb") f.write(...) f.writelines(...)</pre>
Binary	<pre>f=fopen(name,"wb"); fwrite(f, ...);</pre>	<pre>ofstream f(name,ios::binary); f.write(...);</pre>	<pre>f = open(name, "wb") f.write(b...)</pre>

Metadata

But what about that metadata? What is it?

- Metadata is the data about the data. Meaning information that lets you make sense of the data.
- It can (and should) include just about any and all information about how the data was created:
 - what parameters were used in the run?
 - where it was run, when it was run.
 - the version of the code used to perform the run, compiler used to create the code, compiler flags.
 - and anything else that might or not be useful.
- If you're not sure if that bit information should be kept as metadata, then keep it. You never know what information might be needed in the future.

Standard formats

What's the best way to save our metadata? There are several standard file formats which *combine* the metadata with the data:

- HDF5 (Hierarchical Data Format)
- NetCDF (Network Common Data Form)
- discipline-specific formats

What are the benefits?

- Most are provided as libraries.
- Self-describing (metadata is embedded with the data).
- Many are binary agnostic, so portable.
- Many support Parallel I/O and native FS support.
- Broader tool support (visualization, etc.)

ASCII vs. Binary vs. NetCDF

American Standard Code for Information Interchange

Pros

- Human readable
- Could embed metadata
- Portable (architecture independent)

Cons

- Inefficient storage
- Expensive for read/write (conversions)

Native Binary

Pros

- Efficient storage
- Efficient read/write (native)

Cons

- Have to know the format to read
- Portability (Endianness)

NetCDF

Pros

- Efficient storage
- Efficient read/Write
- Portability
- Embedded metadata

Cons

- Only for multi-dimensional arrays

Summary File-IO

- Use file I/O as little as possible. Keep it to big files, with as few IOPs as possible.
- Use a binary format to store you data, not ASCII.
- It's a good practise to make your data "self-describing", meaning store your metadata with your data in the same file.
- NetCDF is a commonly used format to store data that has many useful features.

I/O performance

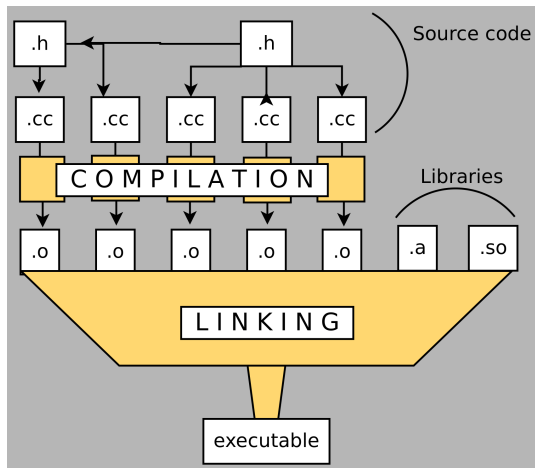
- **Binary data**
- **Large files**
- **Spatial Locality**
- **Reduce number of IOps**

I/O best practices

- **Metadata**
- **Self-describing file format**
- **Use the ones already available via libraries...**
- **NetCDF, HDF5, ...**

make

Modularity and Automatization



- header (.h) files
 - implementation (.c/.cc/.cpp/.cxx) files
 - objects (.o) files – generated by the compilation stage
 - library files (.a/.so/...)
 - an executable – generated in the linking stage
- * **make** can help to automatize and generate each of the steps in the process

What really happens in call to gcc

- `gcc file1.c file2.c -o myprogram`
- First the *pre-processor* runs, it looks for example for `#include` directives and includes the corresponding `.h` file
- Then the *compiler* runs on each `.c` file. It produces machine code (or object code) for each `.c` file.
- Then the *linker* takes all the object code files and combines them into one executable (called `myprogram` in our example).

```
gcc -std=c99 -Wall -O2 -lm file1.c file2.c file3.c -o myprogram
```

The gcc command can grow quite long:

- `gcc file1.c file2.c file3.c file4.c
..... -o myprogram`
- Also recompiles every module, even if it has not changed. This could be addressed by separating compilation & linking:
`gcc -c file1.c # this produces
file1.o
gcc -c file2.c # this produces
file2.o
gcc -o myprogram file1.o file2.o #
produces executable`
- Now we could recompile only files that changed. Still not very convenient

make: Origin

The typical compilation of a large program is a two-step process:

- ① First individually compile all `.c` files, but not the `.h` (header) files, to generate `.o` (library) files.
- ② Link all of the `.o` files together, including external `.so` and `.a` (shared-object and static library files), to generate an executable.

However, it can get complicated and redundant:

- you need to keep track of what depends upon what.
- you need to retype in the entire compilation command every time you need to recompile.
- It's easy to forget all of your compiler flags from one day to the next, as well as the location of external libraries.

It's better to keep all of this information contained in a single file.

This is where the 'make' program enters the picture.

make: Generalities

`make` is a program that is used to build programs from multiple `.c`, `.h`, `.o`, and other files.

- `make` is a very general framework that is used to compile code, of any type.
- `make` takes a '[Makefile](#)' as its input, which specifies what to do, and how.
- The [Makefile](#) contains variables, rules and dependencies.
- The [Makefile](#) specifies executables, compiler flags, library locations, ...
- It is a crucial component of *Professional Software Development*
- Besides building programs, `make` can be used to manage any project where some files must be updated automatically from others whenever the others change (eg. papers, ...)
- [GNU make refs](#)

make: basic usage

- Make is invoked with a list of target file names to build as *command-line arguments*,

```
$ make [TARGET ...]
```

- Without arguments, Make builds the first target that appears in its makefile, which is traditionally a symbolic “**phony**” target named **ALL**.
- Make decides whether a target needs to be regenerated by comparing file modification times (**timestamp**).
This solves the problem of avoiding the building of files which are already up to date, but be aware that sometimes it may fail...
- Make takes many command-line arguments (`make --help`), and can also tell you the available targets in a *makefile*,

```
$ make --help
```

Makefiles

- Make searches the current directory for the `makefile` to use, (eg. `makefile`, `Makefile`, `GNUmakefile`) and then runs the specified (or default) target(s) from (only) that file
- It is possible to specify a different “makefile” by using a ‘`-f`’ flag, eg.



```
$ make -f myMakefile [TARGET ...]
```

- the `makefile` is a plain-text file, with a particular structure
- it may include rules and even use commands from the shell

Rules

- A makefile consists of **rules**.
- Each rule begins with a textual dependency line which defines a **target** followed by a colon (:), and optionally an enumeration of components (files or other targets) on which the target depends.

Eg. a **target** is a file to be created or updated.

- The **dependency line** is arranged so that the target (left hand of the colon) depends on components (right hand of the colon).
- It is common to refer to components as **prerequisites** of the target.
- each command-line must start with a   , to be recognized as a command

```
TARGET: dependencies...
```

```
[command 1]
```

```
:
```

```
[command n]
```

```
TARGET1 [TARGET2 ...]: dep1 dep2 ...
```

```
[command 1]
```

```
:
```

```
[command n]
```

```
Makefile:3: *** missing separator. Stop.
```

Rules – commands

- Each command is executed by a separate shell or command-line interpreter instance.
- backslash `\` can be used to have commands executed by the same shell, it represents line-continuation
- commands can be separated by `;`
- comments are included using `#`

```
# target:  list - List source files
list:
    # Won't work, each cmd is in separate shell
    cd src
    ls
    # Correct, continuation of the same shell
    cd src; \
    ls
```

- an `@`, results in the command not to be printed to standard output

```
# example of a simple makefile
hello: ; @echo 'hello'
```

```
hello:
    @echo 'hello'
```

- A rule may have no command lines def. The dependency line can consist solely of components that refer to targets.

```
# example of a makefile, with
# multiple rules concatenated
realclean:  clean distclean
:
:
clean:  ...
:
:
distclean:  ...
```


Macros & Variables

- Macros are usually referred to as variables when they hold simple string definitions, like “`CC = gcc`”.
- Macros in makefiles may be overridden in the command-line arguments passed to the Make utility.
- Macros allow users to specify the programs invoked and other custom behavior during the build process. For example, the macro “`CC`” is frequently used in makefiles to refer to the location of a C compiler

```
MACRO = definition
```

```
PACKAGE = package
VERSION = `date +%Y.%m%d`
ARCHIVE = $(PACKAGE)-$(VERSION)
dist:
    # Notice that only now macros are
    # expanded for shell to interpret:
    # tar -cf package-`date +%Y.%m%d`.tar

    tar -cf $(ARCHIVE).tar .
```

- Environment variables are also available as macros.

make is sort of two languages in one

- The first language describes **dependency graphs** consisting of **targets and prerequisites**.

Variables

Variables

A variable begins with a `$` and is enclosed within parentheses `(...)` or braces `{...}`.

Single character variables do not need the parentheses.

Egs. `$(CC)`, `$(CC_FLAGS)`, `$@`, `$^`.

Automatic Variables

`$@`: the target filename

`$*`: the target filename without the file extension

`$<`: the first prerequisite filename

`$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.

`$+`: similar to `$^`, but includes duplicates

`$?`: the names of all prerequisites that are newer than the target, separated by spaces

Special Rules

Suffix Rules

have target with names in the form “**.FROM.TO**”, and are used to launch actions based on file extension.

Eg.

```
.SUFFIXES: .txt .html
# From .html to .txt
.html.txt:
    lynx -dump $< > $@
```

\$< refers to the first prerequisite

\$@ refers to the target

```
$ make file.txt
lynx -dump file.html > file.txt
```

Suffix rules cannot have any prerequisites of their own.

Input and Output redirection

- By default, programs read from standard input (keyboard), write results on standard output (screen), and write errors to standard error (screen)
- You can redirect input, output and errors:
 - “ > filename” redirects output to file
 - “>> filename” appends output to file
 - “< inputfile” redirects input (reading from inputfile instead of keyboard)
- 1 means stdout, 2 means stderr

```
1 # saves the output of ls in output.txt (overwriting contents)
2 ls > output.txt
3
4 # saves the output of ls in output.txt (appending)
5 ls >> output.txt
6
7 # does not save output in output.txt ...
8 ls -z > output.txt
9
10 # saves output in output.txt
11 ls -z 2> output.txt
```

Special Rules

Pattern Rules

A pattern rule looks like an ordinary rule, except that its target contains exactly one character '**%**'.

The target is considered a pattern for matching file names: the '**%**' can match any substring of zero or more characters, while other characters match only themselves.

The prerequisites likewise use '**%**' to show how their names relate to the target name.

Eg.

```
# From .html to .txt
%.txt : %.html
    lynx -dump $< > $@
```

Use **\$<** to refer to the first dependency of the current rule.

Use **\$@** to refer to the target of the current rule.

Use **\$\$** to refer to the dependencies of the current rule.

phony target

A target that does not correspond to a file or other object.

Phony targets are usually symbolic names for sequences of actions.

```
.PHONY: variables
variables:
    @echo TXT_FILES: $(TXT_FILES)
```

```
.PHONY: clean
clean:
    rm -f *.dat
    rm -f results.txt
```

Compilation and Linking

MyProg.cc

```
1 // example using the GSL library
2
3 #include <stdio.h>
4
5 #include <gsl/gsl_sf_bessel.h>
6
7 int main()
8 {
9     double x = 5.0;
10
11     double y = gsl_sf_bessel_J0(x);
12
13     printf("J0(%g) = %.18e\n", x, y);
14
15     return 0;
16 }
```

* Compilation

```
$ gcc -std=c99 -I/usr/local/include -c MyProg.cc

$ export GSL_INC=/usr/local/include
$ gcc -std=c99 -I${GSL_INC} -c MyProg.cc
```

* Linking with libraries

```
$ gcc -std=c99 -L/usr/local/lib MyProg.o -lgsl

$ export GSL_LIB=/usr/local/lib
$ gcc -std=c99 -L${GSL_LIB} MyProg.o -lgsl
```

Compiling with make

How does make work?

- A makefile 'rule' is a word followed by a colon (:).
- By default make will execute the first rule it encounters.
- After the colon are the **dependencies** of the rule.
- When make hits a dependency it goes and looks for it.
- When it runs out of rules for the dependencies, it checks the timestamps; if the dependency is newer than the rule the command is executed.

```
# This file is called Makefile
# for compiling a program using GSL

# Define the compiler to use.
CC = gcc

# Compiler and linker flags.
GSL_INC ?= . ; GSL_LIB ?= .
CCFLAGS = -I${GSL_INC} -O2
LDFLAGS = -L${GSL_LIB}
LDLIBS = -lgsl -lgslcblas


all: myProg

myProg: myProg.o
    ${CC} -o myProg myProg.o ${LDFLAGS} ${LDLIBS}

myProg.o: myProg.cc
    ${CC} ${CCFLAGS} -c -o myProg.o myProg.cc
```


Compiling multiple source files with make

How does make work?

- `make` will only recompile those dependencies that have source files that are newer than the library, thus only the code you are working on is modified.
- The most annoying part of `make`: the indentation of the command after the rule is actually a 'tab' (), and it must be a tab.
- The `\` symbol indicates a line-continuation.

```
# Makefile
CC = gcc
GSL_INC ?= . ; GSL_LIB ?= .
CCFLAGS = -I${GSL_INC} -O2
LDFLAGS = -L${GSL_LIB}
LDLIBS = -lgsl -lgslcblas

all: MyArray

MyArray: MyArray.o outputarray.o
    ${CC} -o MyArray MyArray.o \
    outputarray.o ${LDFLAGS} ${LDLIBS}

MyArray.o: MyArray.cc outputarray.h
    ${CC} ${CCFLAGS} -c -o MyArray.o MyArray.cc

outputarray.o: outputarray.cc
    ${CC} ${CCFLAGS} -c -o outputarray.o outputarray.cc
```

Put a 'clean' rule in your Makefile

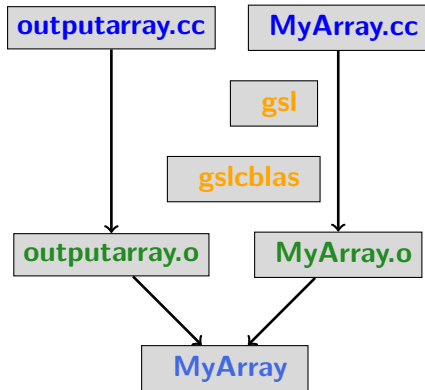
```
CC = gcc
GSL_INC ?= . ; GSL_LIB ?= .
CCFLAGS = -I${GSL_INC} -O2
LDFLAGS = -L${GSL_LIB}
LDLIBS = -lgsl -lgslcblas

MyArray: MyArray.o outputarray.o
    ${CC} -o MyArray MyArray.o outputarray.o ${LDFLAGS}
    ${LDLIBS}

MyArray.o: MyArray.cc outputarray.h
    ${CC} ${CCFLAGS} -c -o MyArray.o MyArray.cc

outputarray.o: outputarray.cc
    ${CC} ${CCFLAGS} -c -o outputarray.o outputarray.cc

clean:
    rm -f MyArray.o outputarray.o MyArray
```



```
$ make clean
$ make
```

Parallel Execution

- Normally, `make` will execute only one recipe at a time, waiting for it to finish before executing the next
- However, the `'-j'` or `'--jobs'` option tells `make` to execute many recipes simultaneously.
- If the `'-j'` option is followed by an integer, this is the number of recipes to execute at once; this is called the number of job slots.
- If there is nothing looking like an integer after the `'-j'` option, there is no limit on the number of job slots.
- The default number of job slots is one, which means serial execution (one thing at a time).
- It is possible to **inhibit** parallelism in a particular makefile with the `.NOTPARALLEL` pseudo-target

Eg.

```
$ make -j 8
```

Summary

`make` utility

- automation tool
- mandatory in software development
- widely used for software installation
- not only used in software compilation, eg. latex-paper generation, ...
- there are several alternatives to make (eg. cmake, ...)

Best Practices on Scientific/Professional Software Development

- Modularity
- Automation Building Tool
- Version Control
- Defensive Programming
- Unit Testing

Example of a Makefile, for compiling a Latex-document

```
# Makefile for compiling a latex paper
NAME=manuscript
TARGET=$(NAME).pdf
SOURCE=$(NAME).tex

JUNK=.aux .bbl .blg .dvi .log .nav .out
      .ps .pdf .snm .tex.backup .tex.bak
      .toc Notes.bib

.PHONY: clean
```

```
$(TARGET): $(SOURCE)
    @pdflatex $(SOURCE)
    @bibtex $(NAME)
    @pdflatex $(SOURCE)
    @pdflatex $(SOURCE)

all: $(TARGET)

clean:
    @for ext in $(JUNK); do \
        rm -v $(NAME)$$ext; \
    done
```

Documenting Makefiles

This might result more useful, either when developing professional software or building pipelines.

`--help` will show available targets in a *makefile*,

```
$ make --help
```

It is possible to document the targets as well, eg

```
:
:
.PHONY : help
help :
    @echo "debug:  compile code with debugging flags."
    @echo "build:   compile code with default options."
    @echo "install:  install package."
    @echo "clean:   Remove auto-generated files."
```

```
$ make help
```

Self-documented Makefiles

It is possible to improve the way in which the targets are documented.

```
## debug:  compile code with debugging flags.
debug :  ...
:
## build:  compile code with default options.
build :  ...
:
## install:  install package.
install :  ...
:
## clean :  Remove auto-generated files.
.PHONY : clean
clean :
:
.PHONY : help
help :  Makefile
    @sed -n 's/^##//p' $<
```

```
$ make help
debug:  compile code with debugging flags.
build:  compile code with default options.
install:  install package.
clean:  Remove auto-generated files.
```

Make – step-by-step example

Make – step-by-step example i

- 1 original source code in multiple files, i.e. multiple compilation proces

```
1 gcc -c file1.c # this produces file1.o
2 gcc -c file2.c # this produces file2.o
3 gcc -o myprogram file1.o file2.o # produces executable
```

- 2 Combine compilation into a single Makefile

```
1 myprog: file1.o file2.o
2     gcc -o myprog file1.o file2.o
3
4 file1.o: file1.c file1.h
5     gcc -c file1.c
6
7 file2.o: file2.c file2.h
8     gcc -c file2.c
```

Make – step-by-step example ii

- ③ Add *macros* for compiler & compilation flags – CC CFLAGS

```
1 CC = gcc
2 CFLAGS = -Wall -g -std=c99 -Werror
3
4 myprog: file1.o file2.o
5     $(CC) $(CFLAGS) -o myprog file1.o file2.o
6
7 file1.o: file1.c file1.h
8     $(CC) $(CFLAGS) -c file1.c
9
10 file2.o: file2.c file2.h
11     $(CC) $(CFLAGS) -c file2.c
```

Make – step-by-step example iii

④ Employ *special rules* and *automatic variables*

```
1 CC = gcc
2 CFLAGS = -Wall -g -std=c99 -Werror
3
4 myprog: file1.o file2.o
5     $(CC) $(CFLAGS) -o $@ $^
6
7 %.o: %.c file.h
8     $(CC) $(CFLAGS) -c $<
```

Make – step-by-step example iv

5 Always add a *clean* rule...

```
1 CC = gcc
2 CFLAGS = -Wall -g -std=c99 -Werror
3
4 myprog: file1.o file2.o
5     $(CC) $(CFLAGS) -o $@ $^
6
7 %.o: %.c file.h
8     $(CC) $(CFLAGS) -c $<
9
10 .PHONY: clean
11 clean:
12     rm *.o
```

6 Make your makefile self-describing (i.e. help!!!)