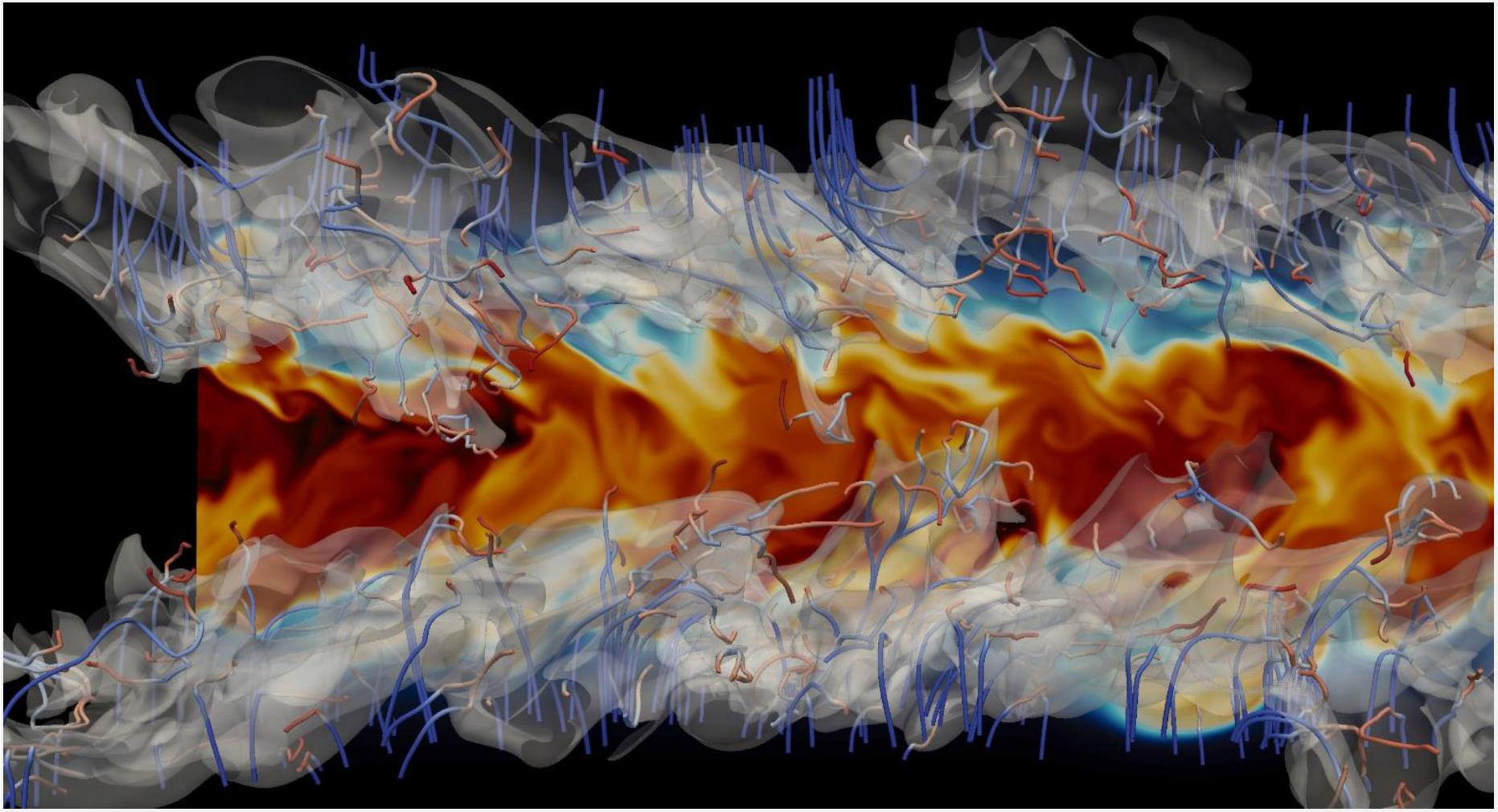
Software Tools for UNIX/Linux Systems

Part 8: Make, Debugging, Profiling

C. Hasse

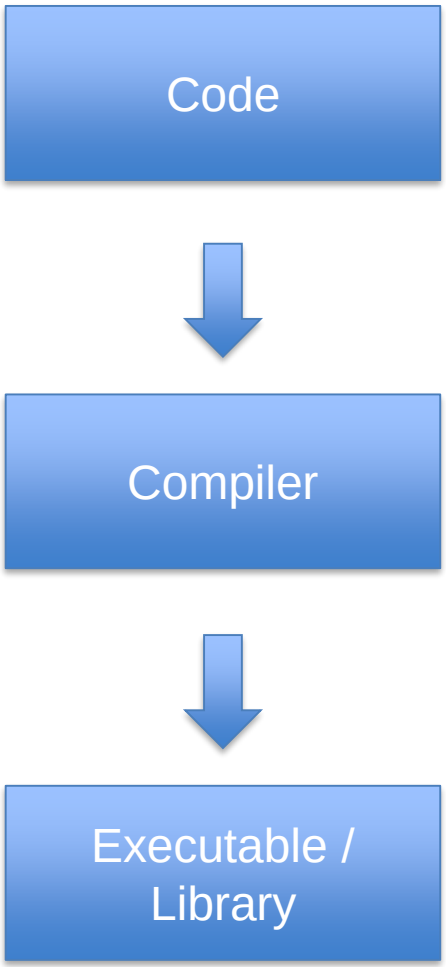
TECHNISCHE UNIVERSITÄT DARMSTADT

STFS



GNU Make

Motivation

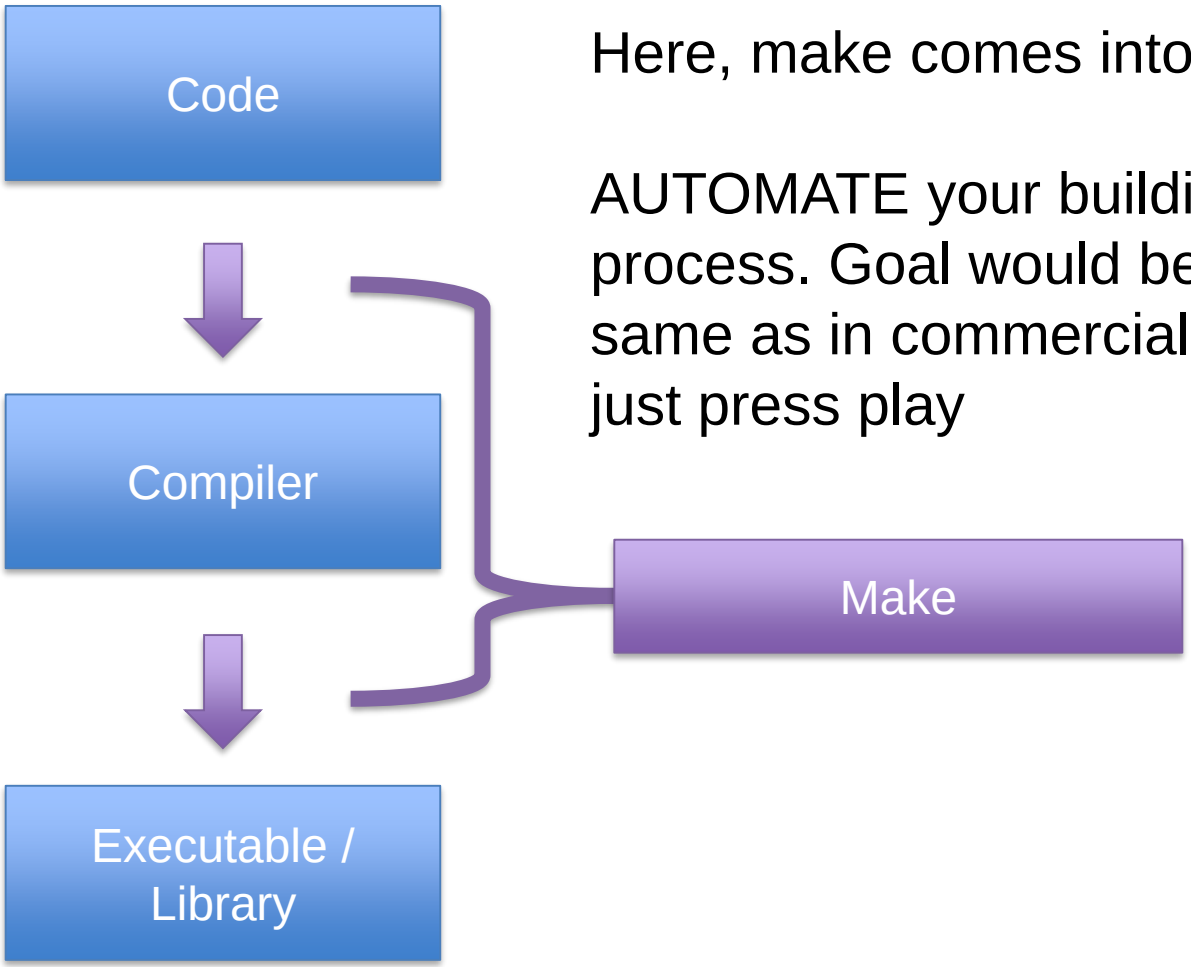


So you have written some code – and what now?

We know how to compile stuff – just do it for every file, right?

We know how to link stuff – just do it for every file, right?

Motivation



Here, make comes into play

AUTOMATE your building process. Goal would be the same as in commercial IDE – just press play

Introduction - Make

- primarily designed for software development
- potentially useful for anything else – if timeline constrains and inter-file dependencies are central point of interest
- due to it’s ability of following different goals it‘s a flexible, yet powerful tool for boosting your everyday chores…compiling, linking, (la-)texing, cleaning up
- targets are -usually- achieved faster and overall process is less error-prone (compared to shell-scripts or execution by hand)

GNU make - execution

make (or gmake)

- scans current PWD for a file named “makefile” or “Makefile”
- creates dependencies tree of projects‘ files and executes script-like commands associated with target

syntax: make [options] [target]

-d debug
-f file use file as makefile
-j [N] parallel mode
-I dir search dir for included files
-p print data base

makefile – general remarks

- is a script and is *interpreted* during runtime
- is *not* read in a linear manner
- can be designed recursively (make calling make calling make…)
- is *naturally* capable of building in parallel
- is parsed several times
  - resolve all include files and generate them if not existing
  - get list of all targets and dependencies
  - build the project by executing shell-script snippets for each target starting at first possible target to be built

make uses script files named “makefile”

header

default-target : dependency
<TAB> <script to create/update target>

target : dependency
<TAB> <…>

targets are *filenames* by default

recursive definition

```
bar = text-to-vanish
foo = $(bar)
bar = hello world!
all:
	echo $(foo)
```

⇨ “hello world!”

```
foo = hello
foo = $(foo) world!

all:
	echo $(foo)
```

⇨ infinite loop
(results in an error)

simply expanded definition

```
bar  = hello world!
foo := $(bar)
bar  = text-to-vanish
all:
	echo $(foo)
```

⇨ “hello world!”

```
foo  = hello
foo := $(foo) world!

all:
	echo $(foo)
```

⇨ “hello world!”

special variables

$@ name of target
$% in case target is part of a library: target name w.o. library-part (otherwise $% is empty)
$< name of first dependency
$? space separated list of all dependencies newer than target
$^ space separated list of all dependencies w.o. duplicates
$+ space separated list of all dependencies with duplicates
$* base name of target (w.o. suffix)

adding a “D” results in directory-only part of the name
adding a “F” results in filename-only part of the name

```
all : ../include/main.h ../src/main.cpp ../include/math.h
	@echo $(^F)
```

“main.h main.cpp math.h”