Build Automation Tools

(Automatizace sestavování)

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

D3S

Pavel Parízek

parizek@d3s.mff.cuni.cz

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

What is “build automation”

- Software building
  - Transforming source code (tree of files) into executable binary code
    - C/C++, C# → Win32 exe, Linux elf
    - Java, Scala → class files (bytecode)
  - Other transformations
    - LaTeX source files → PDF documents

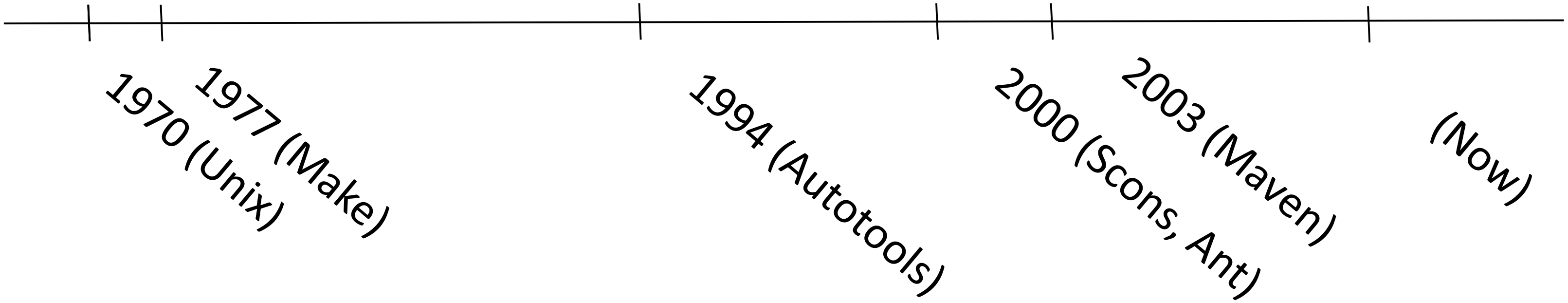
- Automating various processes
  - test execution, packaging, deployment, …

Other “automatable” processes

- Packaging (distribution)
  - generated binary files, metadata, documentation
- Running selected unit tests
- Generating documentation

Popular tools

- Unix (C/C++): Make, Autotools, CMake
- Java/JVM world: Ant, Maven, Gradle
- Windows & .NET/C#: MSBuild, (GUI)



1970 (Unix)

1977 (Make)

1994 (Autotools)

2000 (Scons, Ant)

2003 (Maven)

(Now)

General principles

- Configuration file (script)
  - Declarative specification what should be done
  - Commands to realize specific actions (“how”)
- Ensuring that output (result) matches the most recent available input
  - Dependencies (source → binary)
  - Timestamps (last modification)

Desired features (requirements)

- Automation
  - Minimal interaction with the developer
- Portability
  - Support for multiple platforms
- Efficiency
  - Process each input (source code) file once
  - Reuse previously built (processed) objects
- Robustness
  - Try processing as much input as possible
- Generality
  - Not only for a particular application
- Easy to use
  - Writing and understanding the build scripts

Challenges

- Dependencies
  - Processing files (building modules) in the correct order
    - first binary object files (.o) from source code and then executable
  - Recompile the affected code after modification
    - header file (.h) ➔ source code file (.c, .cpp)
    - class definition (Java, C#) ➔ all files (modules) where it is used
  - How to identify them properly
    - Pre-processor directives ("#include" in C)
    - Source code analysis (bytecode for Java)
    - Metadata and debug symbols in binaries

- Correct build order
  - Logical dependencies between source code files (.c) and intermediate results (.o)
  - Logical dependencies between modules (JARs, assemblies)

Make

Make

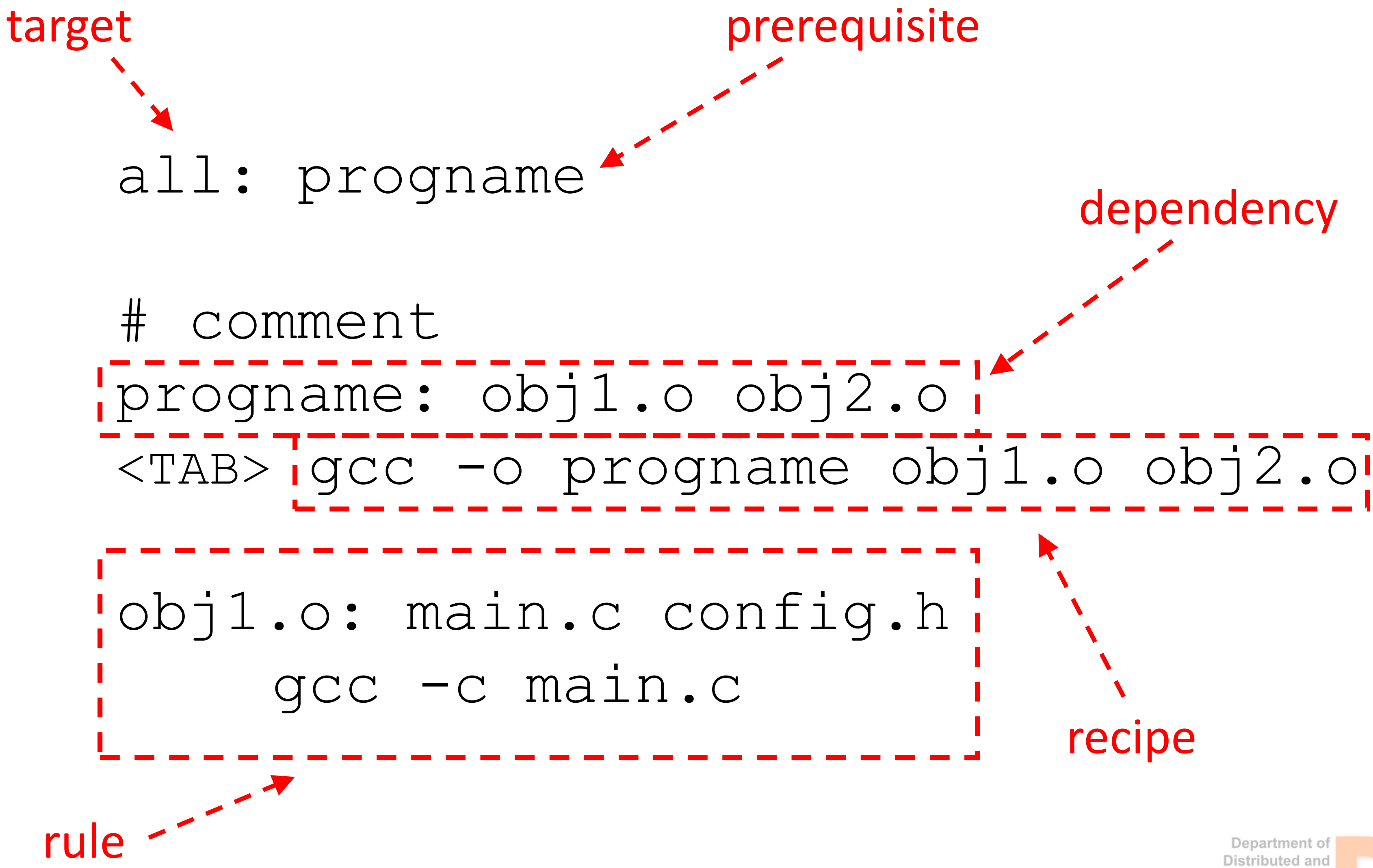
- Standard build automation tool in the Unix and Linux world
- Used mainly for programs that use C/C++ and scripting languages (bash, Awk)
- Many derivatives exist
  - GNU Make, BSD Make, qmake, NMake, ...

- Build script: `Makefile`

Key concepts

- Target
  - Entity to be built: executable program, object file (.o), distribution package (.tgz)
  - Action to be done: clean, build all, prepare something
- Prerequisite
  - Entity that must be **available** and **up-to-date** before the associated target is fulfilled during the build process
- Rules
  - Dependencies between targets and prerequisites
  - Commands that fulfill targets (build entities, ...)

Makefile: structure and syntax



target

prerequisite

dependency

recipe

rule

```
all: progname

# comment
progname: obj1.o obj2.o
<TAB> gcc -o progname obj1.o obj2.o

obj1.o: main.c config.h
    gcc -c main.c
```

Build process with Make

- Running
  - `make` *`target`*
  - `make` // default target
- Two steps
  - Construction of the build tree
    - Root node: target given by the user
    - Leaf nodes: available prerequisites
  - Processing rules in the tree

Example: the “sockets” program

- Simple network client and server
- Both have an UDP and TCP variant
  - Select using the parameter “`-u`”

- http://d3s.mff.cuni.cz/files/teaching/nswi154/sockets.tgz

- Source code written in C++
  - Rather old version of the language

- Script `build.sh`
  - Commands that can be used to compile source files with GCC
- Script `clean.sh`
  - Commands to remove binaries and intermediate object files

Task

- Step 1: basic naive `Makefile` for “sockets”

- What it has to specify
  - Few useful targets
    - `all`, `clean`, program binaries, object files (.o)
  - Dependencies between entities
    - .o → binary, .cpp → .o, etc

Variables

```
objects = obj1.o obj2.o main.o \
          utils.o network.o gui.o

all : progname

progname: $(objects)
    gcc -o prog $(objects) -lcommon
```

Note: wildcard expansion is quite tricky (manual, section 4.4)

https://www.gnu.org/software/make/manual/html_node/Wildcard-Examples.html

Phony targets

- When the target does not represent any file

```
.PHONY : clean

clean :
    rm *.o
```

Guidelines

- Use built-in variables
  - `CC` // C compiler (gcc)
  - `CFLAGS` // C compiler flags
  - `CXX` // C++ compiler (g++)
  - `CXXFLAGS` // C++ compiler flags
  - ... and many more

- Use standard targets
  - `all`, `clean`, `distclean`, `install`

How to use built-in variables

- Define recipes properly

```
$(CC) $(CFLAGS) -c main.c
```

- Set flags when running Make

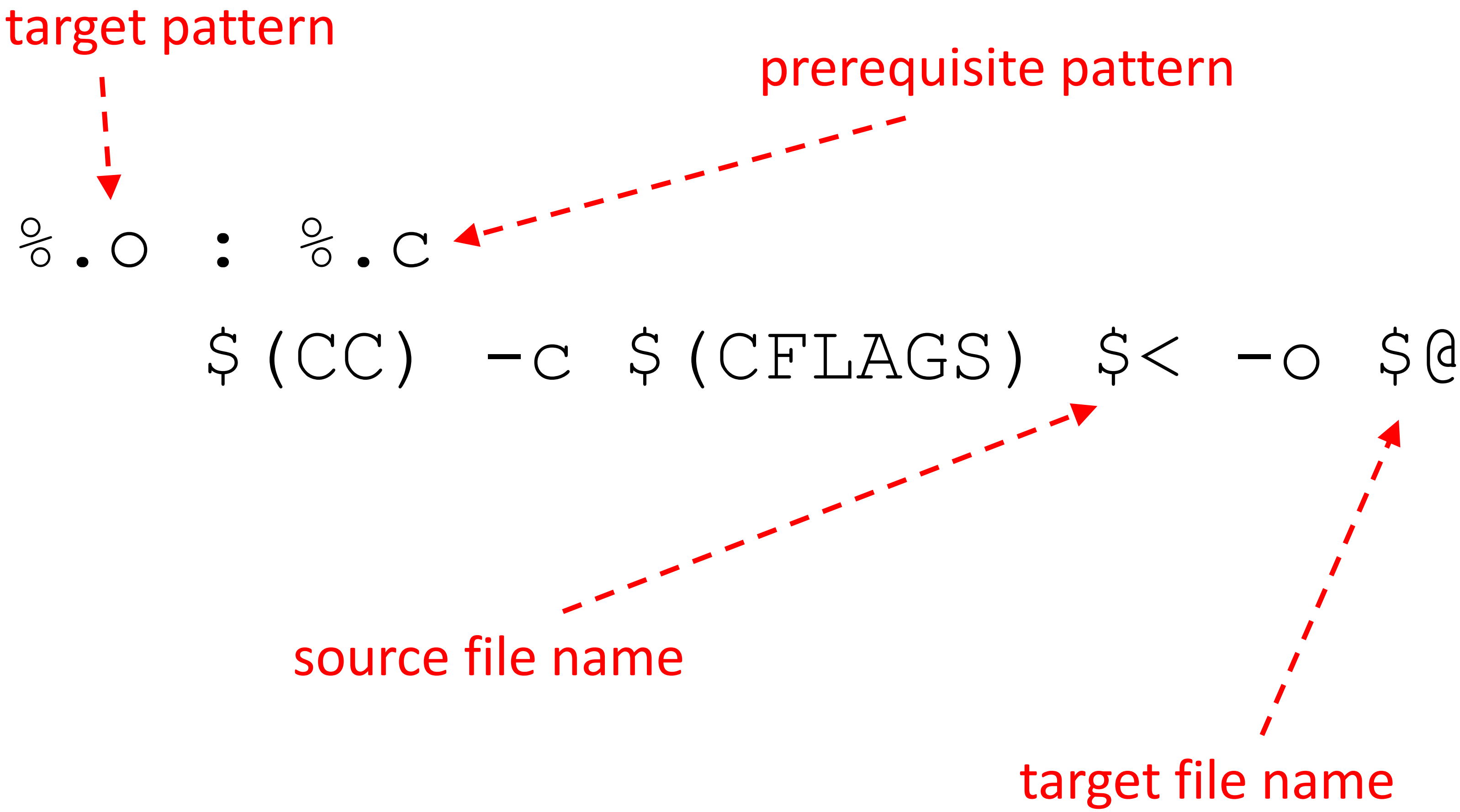
```
CFLAGS=-O2 make
```

Static pattern rules

```
objects = main.o util.o network.o

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

Implicit rules



target pattern

prerequisite pattern

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

source file name

target file name

Task

- Step 2: improve `Makefile` for “sockets”

- Eliminate duplication using these features:
  - Variables (built-in, custom)
  - Implicit rules
  - Static patterns

- Use dependencies between targets properly
- Respect common guidelines (best practices)

Department of Distributed and Dependable Systems

Recursive invocations (subdirectories)

```
SUBDIRS = src doc

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
      $(MAKE) -C $@
```

Two flavors of variables

- Recursively expanded

```
objects = $(core_objs) $(server_objs)
core_objs = tcp.o udp.o
server_objs = srv/main.o
```

```
objs = $(objs) main.o
```

- Simply expanded

```
$(core_objs) := tcp.o udp.o
objects := $(core_objs) srv/main.o
```

Substitutions

```
sources := main.c client.c server.c

objects := $(sources:.c=.o)
```

OR

```
objects := $(sources:%.c=%.o)
```

Operations with variables and values

- Appending

```
objects = main.o util.o
objects += network.o
```

- Functions

```
$(subst from,to,text)
$(patsubst pattern,replacement text)
$(filter pattern1 ... patternN,text)
$(dir path1 ... pathN)
$(basename path1 ... pathN)
$(suffix path1 ... pathN)
```

Automatic variables

- Target: `$@`
- First prerequisite: `$<`
- All prerequisites: `$^`

Other advanced features

- Order-only prerequisites
- Automated generating of files that capture prerequisites (suffix `.d`)
  - Good support by compilers
  - Fallback: `makedepend` tool
- Parallel execution
- Conditional directives

- ... and many more

- See the documentation for GNU Make

Limitations

- Portability over different Unix-like systems
  - Library functions in C (issues with compatibility)
  - Environment: shell, utilities (Awk, sed, grep, …)
- Hard to maintain complex `Makefile`s
- Writing rules by hand can be tedious

- Solution: GNU build system (Autotools)
  - Tools: Autoconf, Automake, Libtool, gettext
  - Previous standard in the open-source world (C/C++)

```
./configure ; make ; make install
```

- Solution: CMake
  - The current standard in open-source projects (C++)

Links

- Make
  - http://www.gnu.org/software/make/
  - http://www.gnu.org/software/make/manual/
- NMake
  - https://learn.microsoft.com/en-us/cpp/build/reference/nmake-reference?view=msvc-170

Build scripts – practice

- Core infrastructure of your project

- Treat in the same way as code
  - readability, modularity

- Possibly very complex

Homework

- Assignment
  - ReCodEx: group associated with this course
  - Web: https://d3s.mff.cuni.cz/files/teaching/nswi154/ukoly/
- Deadline
  - 12.3.2025