

# CMake

2023.04.06

SWPP Practice Session

Seunghyeon Nam

# Build System

- Also called “build automation software”
- Why do we need one?

# Build System

- Also called “build automation software”
- Why do we need one?
  - We write a lot of code in **multiple files**
  - We write a lot of code on top of **multiple external libraries**
  - **We have to compile & link hundreds of files every time!**

# Make

- De facto standard build system in \*nix
- Manually configure dependencies between source files
- Recompile only when the dependent file(s) have changed

# Problems with Make

```
version_h := include/generated/uapi/linux/version.h

clean-targets := %clean mrproper cleandocs
no-dot-config-targets := $(clean-targets) \
    cscope gtags TAGS tags help% %docs check% coccicheck \
    $(version_h) headers headers_% archheaders archscripts \
    %asm-generic kernelversion %src-pkg dt_binding_check \
    outputmakefile rustavailable rustfmt rustfmtcheck
# Installation targets should not require compiler. Unfortunately, vdso_install
# is an exception where build artifacts may be updated. This must be fixed.
no-compiler-targets := $(no-dot-config-targets) install dtbs_install \
    headers_install modules_install kernelrelease image_name
no-sync-config-targets := $(no-dot-config-targets) %install kernelrelease \
    image_name
single-targets := %.a %.i %.ko %.lds %.ll %.lst %.mod %.o %.rsi %.s %.syntypes %/

config-build :=
mixed-build :=
need-config := 1
need-compiler := 1
may-sync-config := 1
single-build :=

ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
    ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
        need-config :=
    endif
endif

ifneq ($(filter $(no-compiler-targets), $(MAKECMDGOALS)),)
    ifeq ($(filter-out $(no-compiler-targets), $(MAKECMDGOALS)),)
        need-compiler :=
    endif
endif

ifneq ($(filter $(no-sync-config-targets), $(MAKECMDGOALS)),)
    ifeq ($(filter-out $(no-sync-config-targets), $(MAKECMDGOALS)),)
        may-sync-config :=
    endif
endif
endif
```

```
KBUILD_HOSTCFLAGS := $(KBUILD_USERHOSTCFLAGS) $(HOST_LFS_CFLAGS) $(HOSTCFLAGS)
KBUILD_HOSTCXXFLAGS := -Wall -O2 $(HOST_LFS_CFLAGS) $(HOSTCXXFLAGS)
KBUILD_HOSTRUSTFLAGS := $(rust_common_flags) -O -Cstrip=debuginfo \
    -Zallow-features= $(HOSTRUSTFLAGS)
KBUILD_HOSTLDFLAGS := $(HOST_LFS_LDFLAGS) $(HOSTLDFLAGS)
KBUILD_HOSTLDLIBS := $(HOST_LFS_LIBS) $(HOSTLDLIBS)

# Make variables (CC, etc...)
CPP = $(CC) -E
ifneq ($(LLVM),)
    CC = $(LLVM_PREFIX)clang$(LLVM_SUFFIX)
    LD = $(LLVM_PREFIX)ld.lld$(LLVM_SUFFIX)
    AR = $(LLVM_PREFIX)llvm-ar$(LLVM_SUFFIX)
    NM = $(LLVM_PREFIX)llvm-nm$(LLVM_SUFFIX)
    OBJCOPY = $(LLVM_PREFIX)llvm-objcopy$(LLVM_SUFFIX)
    OBJDUMP = $(LLVM_PREFIX)llvm-objdump$(LLVM_SUFFIX)
    READELF = $(LLVM_PREFIX)llvm-readelf$(LLVM_SUFFIX)
    STRIP = $(LLVM_PREFIX)llvm-strip$(LLVM_SUFFIX)
else
    CC = $(CROSS_COMPILE)gcc
    LD = $(CROSS_COMPILE)ld
    AR = $(CROSS_COMPILE)ar
    NM = $(CROSS_COMPILE)nm
    OBJCOPY = $(CROSS_COMPILE)objcopy
    OBJDUMP = $(CROSS_COMPILE)objdump
    READELF = $(CROSS_COMPILE)readelf
    STRIP = $(CROSS_COMPILE)strip
endif
RUSTC = rustc
RUSTDOC = rustdoc
RUSTFMT = rustfmt
CLIPPY_DRIVER = clippy-driver
BINDGEN = bindgen
CARGO = cargo
PAHOLE = pahole
RESOLVE_BTIDS = $(objtree)/tools/bpf/resolve_btfids/resolve_btfids
```

```
# Allows the usage of unstable features in stable compilers.
export RUSTC_BOOTSTRAP := 1

export ARCH SRCARCH CONFIG_SHELL BASH HOSTCC KBUILD_HOSTCFLAGS CROSS_COMPILE LD CC HOSTPKG_CONFIG
export RUSTC RUSTDOC RUSTFMT RUSTC_OR_CLIPPY_QUIET RUSTC_OR_CLIPPY_BINDGEN CARGO
export HOSTRUSTC KBUILD_HOSTRUSTFLAGS
export CPP AR NM STRIP OBJCOPY OBJDUMP READELF PAHOLE RESOLVE_BTIDS LEX YACC AWK INSTALLKERNEL
export PERL PYTHON3 CHECK CHECKFLAGS MAKE UTS_MACHINE HOSTCXX
export KGZIP KBZIP2 KLZOP LZMA LZ4 XZ ZSTD
export KBUILD_HOSTCXXFLAGS KBUILD_HOSTLDFLAGS KBUILD_HOSTLDLIBS LDFLAGS_MODULE
export KBUILD_USERCFLAGS KBUILD_USERLDFLAGS

export KBUILD_CPPFLAGS NOSTDINC_FLAGS LINUXINCLUDE OBJCOPYFLAGS KBUILD_LDFLAGS
export KBUILD_CFLAGS CFLAGS_KERNEL CFLAGS_MODULE
export KBUILD_RUSTFLAGS RUSTFLAGS_KERNEL RUSTFLAGS_MODULE
export KBUILD_AFLAGS AFLAGS_KERNEL AFLAGS_MODULE
export KBUILD_AFLAGS_MODULE KBUILD_CFLAGS_MODULE KBUILD_RUSTFLAGS_MODULE KBUILD_LDFLAGS_MODULE
export KBUILD_AFLAGS_KERNEL KBUILD_CFLAGS_KERNEL KBUILD_RUSTFLAGS_KERNEL
export PAHOLE_FLAGS

# Files to ignore in find ... statements

export RCS_FIND_IGNORE := \
    \
    -name SCCS -o -name BitKeeper -o -name .svn -o \
    -name CVS -o -name .pc -o -name .hg -o -name .git \
    -prune -o
export RCS_TAR_IGNORE := --exclude SCCS --exclude BitKeeper --exclude .svn \
    --exclude CVS --exclude .pc --exclude .hg --exclude .git

# =====
# Rules shared between *config targets and build targets

# Basic helpers built in scripts/basic
PHONY += scripts_basic
scripts_basic:
    $(Q)$(MAKE) $(build)=scripts/basic

PHONY += outputmakefile
ifdef building_out_of_srctree
```

# Problems with Make

- Makefile is somewhat verbose
  - Dependency hierarchy can quickly grow into unmanageable mess
- Makefile is really just a list of compile commands
  - We still have to specify a lot of things manually
  - Include path, libraries to link, etc...

# CMake

- Configure build with higher level scripts (`CMakeFile.txt`)
  - Reduce complexity
  - Automatically configures compile options
- Meta build system
  - CMake itself does not compile the files
  - Instead, it generates build scripts for the ‘generators’

# Using CMake

Configure prior to build

```
cmake -G Ninja -S llvm -B build \  
-DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \  
-DLLVM_INSTALL_UTILS=ON \  
-DLLVM_TARGETS_TO_BUILD="X86" \  
-DBUILD_SHARED_LIBS=ON \  
-DCMAKE_BUILD_TYPE=Release \  
-DCMAKE_INSTALL_PREFIX=$LLVM_DIR  
cmake --build build
```



# Using CMake

Generator to use (Ninja, Makefile, ...)

```
cmake -G Ninja -S llvm -B build \  
-DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \  
-DLLVM_INSTALL_UTILS=ON \  
-DLLVM_TARGETS_TO_BUILD="X86" \  
-DBUILD_SHARED_LIBS=ON \  
-DCMAKE_BUILD_TYPE=Release \  
-DCMAKE_INSTALL_PREFIX=$LLVM_DIR  
cmake --build build
```

# Using CMake

Root CMakeFile location

```
cmake -G Ninja -S llvm -B build \  
  -DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \  
  -DLLVM_INSTALL_UTILS=ON \  
  -DLLVM_TARGETS_TO_BUILD="X86" \  
  -DBUILD_SHARED_LIBS=ON \  
  -DCMAKE_BUILD_TYPE=Release \  
  -DCMAKE_INSTALL_PREFIX=$LLVM_DIR  
cmake --build build
```

# Using CMake

Build directory

```
cmake -G Ninja -S llvm -B build \
  -DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \
  -DLLVM_INSTALL_UTILS=ON \
  -DLLVM_TARGETS_TO_BUILD="X86" \
  -DBUILD_SHARED_LIBS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=$LLVM_DIR
cmake --build build
```

# Using CMake

```
cmake -G Ninja -S llvm -B build \  
  -DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \  
  -DLLVM_INSTALL_UTILS=ON \  
  -DLLVM_TARGETS_TO_BUILD="X86" \  
  -DBUILD_SHARED_LIBS=ON \  
  -DCMAKE_BUILD_TYPE=Release \  
  -DCMAKE_INSTALL_PREFIX=$LLVM_DIR  
cmake --build build
```

Variables declared in CMakeFile

# Using CMake

```
cmake -G Ninja -S llvm -B build \  
  -DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \  
  -DLLVM_INSTALL_UTILS=ON \  
  -DLLVM_TARGETS_TO_BUILD="X86" \  
  -DBUILD_SHARED_LIBS=ON \  
  -DCMAKE_BUILD_TYPE=Release \  
  -DCMAKE_INSTALL_PREFIX=$LLVM_DIR  
cmake --build build
```

CMake config

# Using CMake

```
cmake -G Ninja -S llvm -B build \  
  -DLLVM_ENABLE_PROJECTS="clang;lldb;compiler-rt" \  
  -DLLVM_INSTALL_UTILS=ON \  
  -DLLVM_TARGETS_TO_BUILD="X86" \  
  -DBUILD_SHARED_LIBS=ON \  
  -DCMAKE_BUILD_TYPE=Release \  
  -DCMAKE_INSTALL_PREFIX=$LLVM_DIR
```

```
cmake --build build
```

After configuration, build!

# Writing CMakeFile

```
cmake_minimum_required(VERSION 3.13.0)
project(example VERSION 0.1.0)

add_executable(example main.cpp)
```

- ‘Minimal’ CMakeFile
- `add_executable` adds a **build target**
  - Build target: something that can be built from this CMakeFile
- `add_executable(<target_name> <sources...>)`

# Writing CMakeFile

```
add_library(SCParser OBJECT ${SRC_DIR}/lib/parser.cpp)
target_compile_options(SCParser PRIVATE -fPIC)
target_include_directories(SCParser PRIVATE ${LLVM_INCLUDE_DIRS})
llvm_map_components_to_libnames(parser_llvm_libs asmparser)
target_link_libraries(SCParser ${parser_llvm_libs})
```

- `add_library` adds a `library` build target
- `add_library(<target_name> <sources...>)`
- Library target can be later linked against other build targets.



# Writing CMakeFile

```
add_library(SCParser OBJECT ${SRC_DIR}/lib/parser.cpp)
target_compile_options(SCParser PRIVATE -fPIC)
target_include_directories(SCParser PRIVATE ${LLVM_INCLUDE_DIRS})
llvm_map_components_to_libnames(parser_llvm_libs asmparser)
target_link_libraries(SCParser ${parser_llvm_libs})
```

- `target_compile_options` adds **compiler flag** to a build target
- `target_compile_options(<target_name> [<VIS>] <flags...>)`
- Features with <VIS> option can inherit its values to other targets
  - <VIS> is used to control the inheritance

# Writing CMakeFile

```
add_library(SCParser OBJECT ${SRC_DIR}/lib/parser.cpp)
target_compile_options(SCParser PRIVATE -fPIC)
target_include_directories(SCParser PRIVATE ${LLVM_INCLUDE_DIRS})
llvm_map_components_to_libnames(parser_llvm_libs asmparser)
target_link_libraries(SCParser ${parser_llvm_libs})
```

- `target_include_directories` adds `include path` to a build target
- `target_include_directories(<target_name> [<VIS>] <paths...>)`

# Writing CMakeFile

```
add_library(SCParser OBJECT ${SRC_DIR}/lib/parser.cpp)
target_compile_options(SCParser PRIVATE -fPIC)
target_include_directories(SCParser PRIVATE ${LLVM_INCLUDE_DIRS})
llvm_map_components_to_libnames(parser_llvm_libs asmparser)
target_link_libraries(SCParser ${parser_llvm_libs})
```

- `target_link_libraries` adds **library to link** to a build target
- `target_link_libraries(<target_name> [<VIS>] <libs...>)`
- **Library build targets** can be linked like library files

# Writing CMakeFile

```
function(add_opt_pass pass_name file_name)
    add_library(${pass_name} ${SRC_OPT_DIR}/${file_name})
    target_include_directories(${pass_name} PRIVATE ${LLVM_INCLUDE_DIRS})
    target_link_libraries(${pass_name} PRIVATE ${pass_llvm_libs})
    target_link_libraries(OptPasses INTERFACE ${pass_name})
endfunction()
```

- `function` can be helpful when certain tasks have to be repeated
- `function(<name> [<arg1...>])`

# Writing CMakeFile

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(SRC_DIR ${CMAKE_CURRENT_SOURCE_DIR}/src)
option(BUILD_SHARED_LIBS "Build using shared libraries" ON)
```

- `set` is used to declare & set an internal variable
- `option` is used to declare, set and cache a boolean internal variable

# Writing CMakeFile

```
find_package(LLVM REQUIRED CONFIG)
message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVM in: ${LLVM_BINARY_DIR}")
```

- `find_package` is used to import features from external library
- `find_package(<lib_name> [REQUIRED] <feature...>)`
- `message` is used to print some helpful message during configuration
- `message(<type> <message>)`

# Inspecting Configuration

- Configuration yields 2+@ files
  - Generation script, such as `Makefile` or `ninja.build`
  - `CMakeCache.txt` that lists all the declared variables
- Usually, looking at CMakeCache should suffice
- Generation script might be helpful, but it's extremely verbose