VE482 — Introduction to Operating Systems

Project 1 (Compile guide)

Yihao Liu, Modified by TAs later
— UM-JI (Fall 2021)

Goals of the guide

- Install and use LLVM / Clang
- Use GNU make / CMake
- Submit on JOJ

1 Introduction

You are going to know how to build your project that is compatible with JOJ and submit it in this guide.

We are using `llvm/clang` to compile and test your program on JOJ, and we provide two build tools: `GNU make` and `CMake`. You can choose either of them in this project.

2 LLVM / Clang

2.1 Introduction

`clang` is now widely used as a substitute of `gcc`. It has GCC compatibility, fast compilation and low memory use, and expressive diagnostics.

So it is a good choice to install and use `clang` to compile and run your projects locally. In addition, Minix 3 only supports `clang` in default, instead of `gcc`.

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

Find more information on https://llvm.org/.

The Clang project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project.

Find more information on https://clang.llvm.org/.

2.2 Installation

On Windows, you can install and use `clang` for normal C projects, but this project needs some POSIX standard supports, while Windows does not have a full implementation of the standard. So, you are recommended to switch to Linux.

On most Linux distributions, `clang` can be found in the package manager. For example, for Debian (Ubuntu / Linux Mint), you can install it with

```
$ sudo apt install clang
```

On Mac OS X, `clang` is the default compiler installed. The `gcc` and `g++` commands are only aliases of it.

On Minix 3, `clang` is also the default compiler, but you need to install it yourselves.

```
$ pkgin install binutils
$ pkgin install clang
```

2.3 Sanitizers

`clang` provides some sanitizers to detect memory leaks, undefined behaviors and etc.

2.3.1 AddressSanitizer

AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks

Find more information on https://clang.llvm.org/docs/AddressSanitizer.html.

2.3.2 UndefinedBehaviorSanitizer

UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector. UBSan modifies the program at compile-time to catch various kinds of undefined behavior during program execution, for example:

- Using misaligned or null pointer
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination

Find more information on https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

3 Build tools

For simplicity, please put all your source files (`.c` files) in the root path of your project directory.

3.1 GNU Make

Here we have a sample `Makefile` for you. Run `make` to build your project.

```
CC = clang
CFLAGS = -std=gnu11 -O2 -Wall -Wextra -Werror -pedantic -Wno-unused-result -Wconversion
MUMSH_SRC = *.c
MUMSH = mumsh
```

```
MUMSHMC = mumsh_memory_check
MUMSHMC_FLAGS = -fsanitize=address -fno-omit-frame-pointer -fsanitize=undefined
↪  -fsanitize=integer
.PHONY: clean

all: $(MUMSH) $(MUMSHMC)
    @echo mumsh successfully constructed

$(MUMSH): $(MUMSH_SRC)
    $(CC) $(CFLAGS) -o $(MUMSH) $(MUMSH_SRC)

$(MUMSHMC) : $(MUMSH_SRC)
    $(CC) $(CFLAGS) $(MUMSHMC_FLAGS) -o $(MUMSHMC) $(MUMSH_SRC)

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

clean:
    $(RM) *.o *.a *~ $(MUMSH) $(MUMSHMC)
```

3.2 CMake

Here we have a sample `CMakeLists.txt` for you. Make sure you have `cmake` installed on your system (provided on Linux / Mac OS X / Minix 3 by package manager).

```
cmake_minimum_required(VERSION 2.7)

project(ve482p1)

set(CMAKE_C_EXTENSIONS ON)
set(CMAKE_C_STANDARD 11)
set(CMAKE_C_FLAGS "-Wall -Wextra -Werror -pedantic -Wno-unused-result -Wconversion")
file(GLOB SOURCE_FILES "*.c")

add_executable(mumsh ${SOURCE_FILES})
add_executable(mumsh_memory_check ${SOURCE_FILES})

target_compile_options(mumsh_memory_check PUBLIC -fsanitize=address,undefined,integer
↪  -fno-omit-frame-pointer)
target_link_libraries(mumsh_memory_check -fsanitize=address,undefined,integer)
```

Then run the following commands to build your project. If you are using `CLion`, it will be automatically configured with `CMakeLists.txt`.

```
$ mkdir cmake-build-debug && cd cmake-build-debug
$ cmake -DCMAKE_C_COMPILER=clang ..
$ make
```

4 Submission on JOJ

You should archive everything in a tarball (`*.tar`) and submit on JOJ. JOJ will use the default `Makefile` or `CMakeLists.txt` shown before, ignoring the uploaded one. Remember to select your build tool first, and your `Makefile` and `CMakeLists.txt` must be in the first level directory.

Hint: In most cases, you need to submit your answer to JOJ many times to finish all the tasks. Consider using JOJ-Submitter to ease your pain.