MNNIT ALLAHABAD

OPERATING SYSTEM
PART 1

SUBMITTED BY :
NAME : SHISHU
ROLL/REG ID : 2020CA089
SUBMIT DATE : 24/11/2021

SUBMITTED TO :
TEACHER _____
DEPPT : COMPUTER SCIENCE
DEADLINE: 24/11/2021

1 **Generate make file for template-example-1. Its output (*.o) files must be stored in sub-directory. Define clean macro to clean the output file and directory.**

MAKE FILE

```
compiler = g++
source = $(wildcard *.cpp)

compile:
    rm -rf ./output
    mkdir ./output
    ${compiler} ${source} -o ./output/main.o

clean:
    rm -rvf output
```

OUTPUT

```
[redacted]-os:~/college/os/a2/template-example-1$ make
rm -rf ./output
mkdir ./output
g++ function1.cpp function2.cpp main.cpp -o ./output/main.o
[redacted]-os:~/college/os/a2/template-example-1$
```

```
[redacted]-os:~/college/os/a2/template-example-1$ tree
.
├── function1.cpp
├── function2.cpp
├── functions.h
├── main.cpp
├── makefile
└── output
    └── main.o
```

2 **While calling make, pass an argument USE_C with value 1 (“make USE_C=1”)Test this variable using ifeq in your makefile. Use c compiler or java compiler accordingly, to build the targets, depending on the condition being true or false respectively.**

MAKE FILE

```
compiler = g++
source = $(wildcard *.cpp)

ifeq ($(USE_C), 1)
    compiler = gcc
    source = $(wildcard *.c)
endif

compile:
    rm -rf ./output
    mkdir ./output
    ${compiler} ${source} -o ./output/main.o

clean:
    rm -rvf output
```

3 **The process in which a makefile calls other makefiles is called a recursive make. You are given a parent folder with sub-directory, each containing base source C files You have to design a makefile in the parent directory that recursively calls the individual makefiles of the sub-directory. The makefiles in the sub-directories compile their respective code files and store the output there only.**

MAKE FILE ROOT DIRECTORY

```
compile:
    $(MAKE) -C subdir
    gcc file1.c -c -o file1.o
    gcc file2.c -c -o file2.o
```

MAKE FILE SUB DIRECTORY

```
compile:
    $(MAKE) -C tmp
    gcc subfile.c -c -o subfile.o
```

MAKE FILE TMP DIRECTORY

```
compile:
    gcc subsubfile.c -c -o subsubfile.o
```

PART – 2

1 Write your xv-6 installation experience.

Installation of xv-6 operating system involves two steps:

- Installing qemu
- Installing xv6.

Installing qemu

- Install qemu using the command : **sudo apt install qemu**

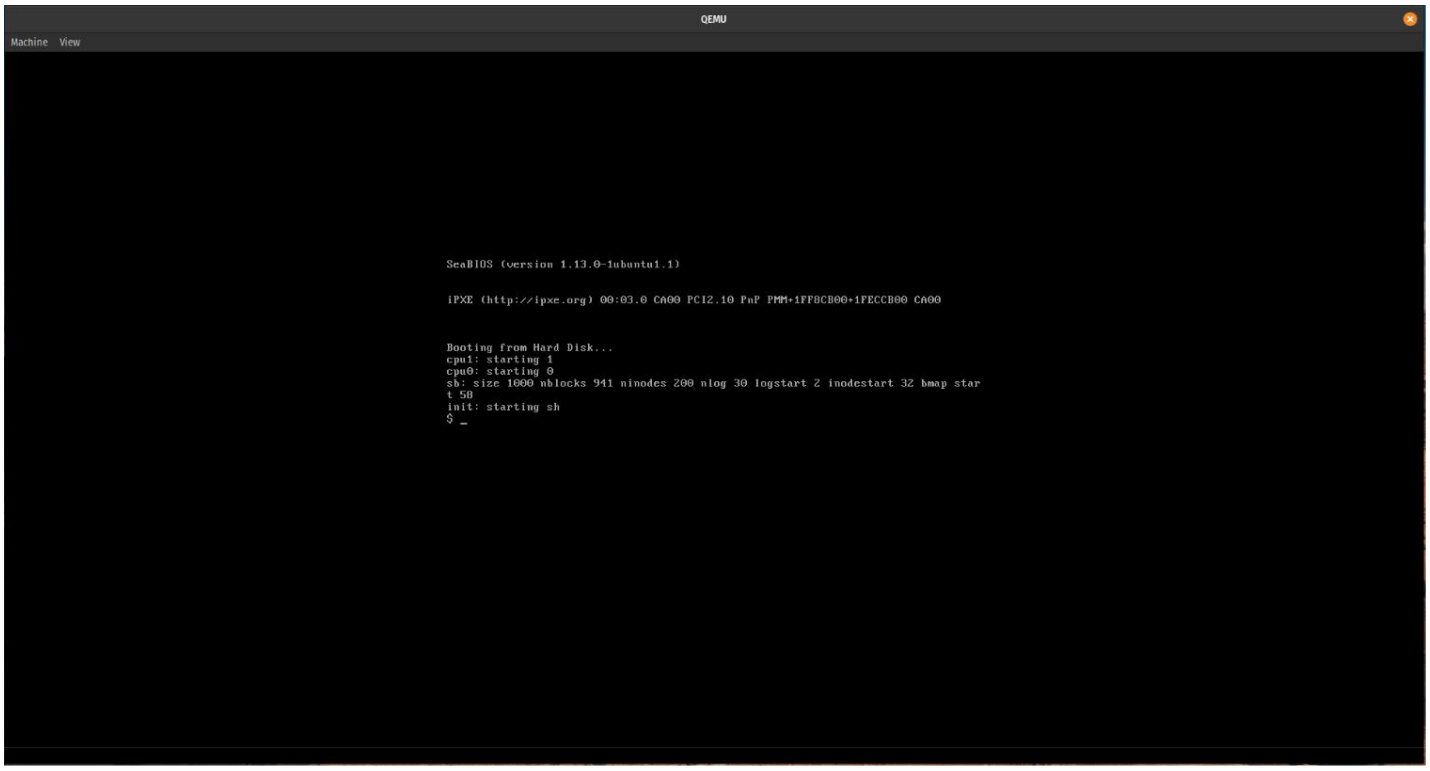
Installing xv6

- First we need to clone xv6 from the git repository using command

git clone git://github.com/mit-pdos/xv6-public.git

```
chandan@pop-os:~$ git clone git://github.com/mit-pdos/xv6-public.git
Cloning into 'xv6-public'...
remote: Enumerating objects: 13990, done.
remote: Total 13990 (delta 0), reused 0 (delta 0), pack-reused 13990
Receiving objects: 100% (13990/13990), 17.18 MiB | 3.85 MiB/s, done.
Resolving deltas: 100% (9537/9537), done.
chandan@pop-os:~$
```

- Then we need to compile xv6 using the command **make.**
- Then we compile and run the emulator qemu



2 Write about the xv6 operating system in 1000 words.

Xv6 is a re-implementation of the Unix sixth edition in ANSI C in order to use as a learning tool. xv6 was developed by MIT as a teaching operating system for their “6.828” course. A vital fact about xv6 is that it contains all the core Unix concepts and has a similar structure to Unix even though it lacks some functionality that you would expect from a modern operating system. This is a lightweight operating system where the time to compile is very low and it also allow remote debugging.

Xv6 source code can be downloaded and compiled from https://github.com/mit-pdos/xv6-public. Users can add their programs to xv6 and change the makefile to compile them so that they can be executed from the shell.

3 Write about the ‘qemu’ in 1500 words.

QEMU is a generic and open source machine emulator and virtualizer.

When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.

When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, 64-bit POWER, S390, 32-bit and 64-bit ARM, and MIPS guests. QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.

QEMU has multiple operating modes:

- **User-mode emulation**

  In this mode QEMU runs single Linux or Darwin/macOS programs that were compiled for a different instruction set. System calls are thunked for endianness and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.

- **System emulation**

  In this mode QEMU emulates a full computer system, including peripherals. It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD it supports emulating several instruction sets, including x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC, SPARC, ETRAX CRIS and MicroBlaze.

- **KVM Hosting**

  Here QEMU deals with the setting up and migration of KVM images. It is still involved in the emulation of hardware, but the execution of the guest is done by KVM as requested by QEMU.

- **Xen Hosting**

  QEMU is involved only in the emulation of hardware; the execution of the guest is done within Xen and is totally hidden from QEMU.