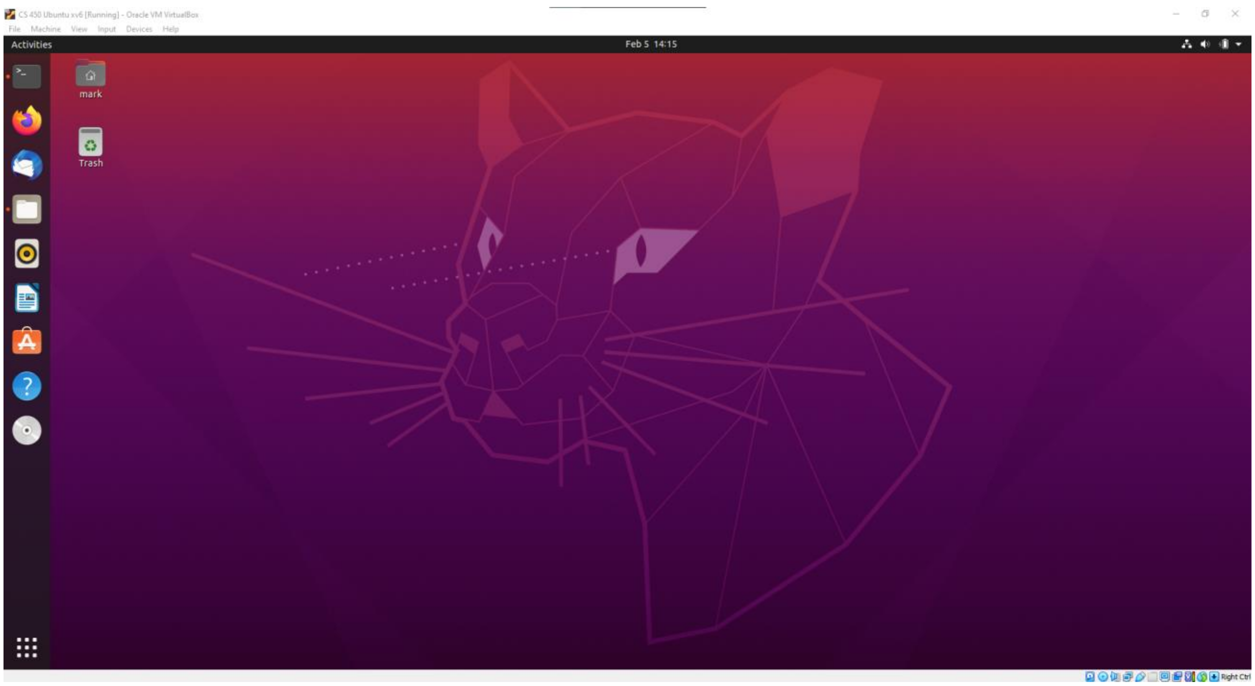
Mark Gameng

CS 450 – Duan Yue

Lab 1

To setup xv6 environment, I used VirtualBox to create a virtual machine running Ubuntu. Once you finish setting up Ubuntu on a VM, you can proceed with the normal instructions to install xv6 on Ubuntu.

The VM screen should look like this



Then, you can go to the terminal using the bar to the right and run the following commands to get xv6.

sudo apt-get update && sudo apt-get install git nasm build-essential qemu gdb emacs
git clone https://github.com/mit-pdos/xv6-public.git
cd xv6-public
make
make qemu-nox

This launches xv6 and the command line is now running the qemu operating system. To exit out of this press CTRL-A then X.

For me, this command resulted in an error. Saying it couldn't find a working QEMU executable. Luckily, there was a post on Piazza about fixing this specific problem. The following commands below is how I fixed this problem.

sudo apt remove qemu
git clone https://github.com/qemu/qemu.git
cd qemu

sudo apt install -y libglib2.0-dev libfdt-dev libpixman-1-dev zlib1g-dev ninja-build
./configure --disable-kvm --target-list="i386-softmmu x86_64-softmmu"
make
sudo make install

After all of this, you can now find qemu-system-i386 on ur system and can now launch xv6 by **make qemu-nox**

make qemu-gdb

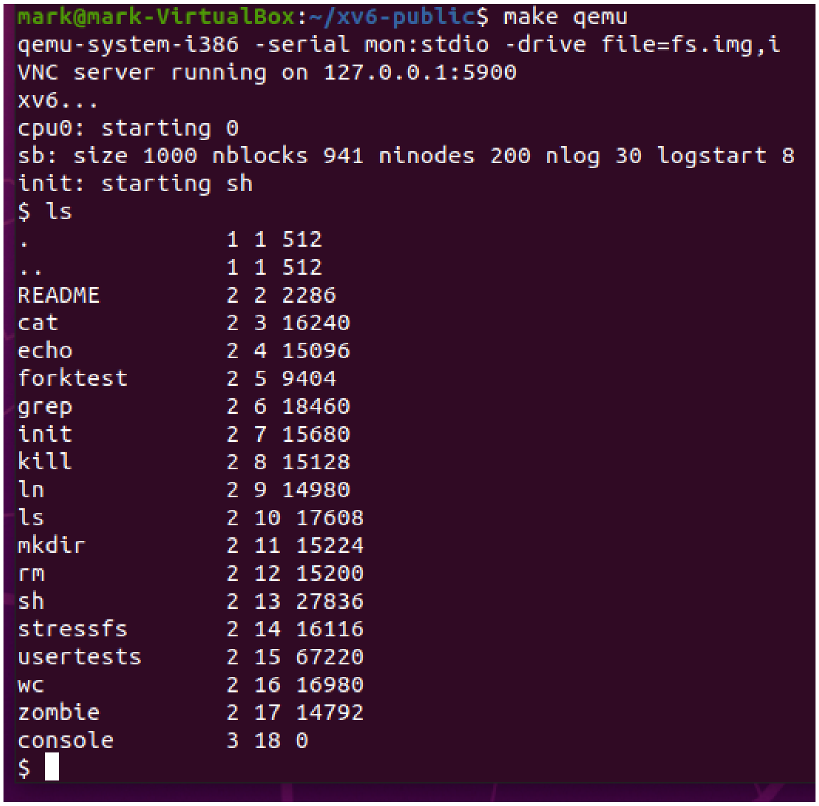
The command above is to debug xv6 with gdb. To exit out of this press CTRL-A then X. Then in another window go in the same directory and you can enter gdb in the cmd or in emacs.

Below are some pictures of me compiling xv6 and running commands and doing some debugging.

```
mark@mark-VirtualBox:~$ git clone https://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 | 9.69 MiB/s, done.
Resolving deltas: 100% (9537/9537), done.
mark@mark-VirtualBox:~$ cd xv6-public
mark@mark-VirtualBox:~/xv6-public$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32
-Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno-pic -
O -nostdinc -I. -c bootmain.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32
-Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -fno-pic -
nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
```

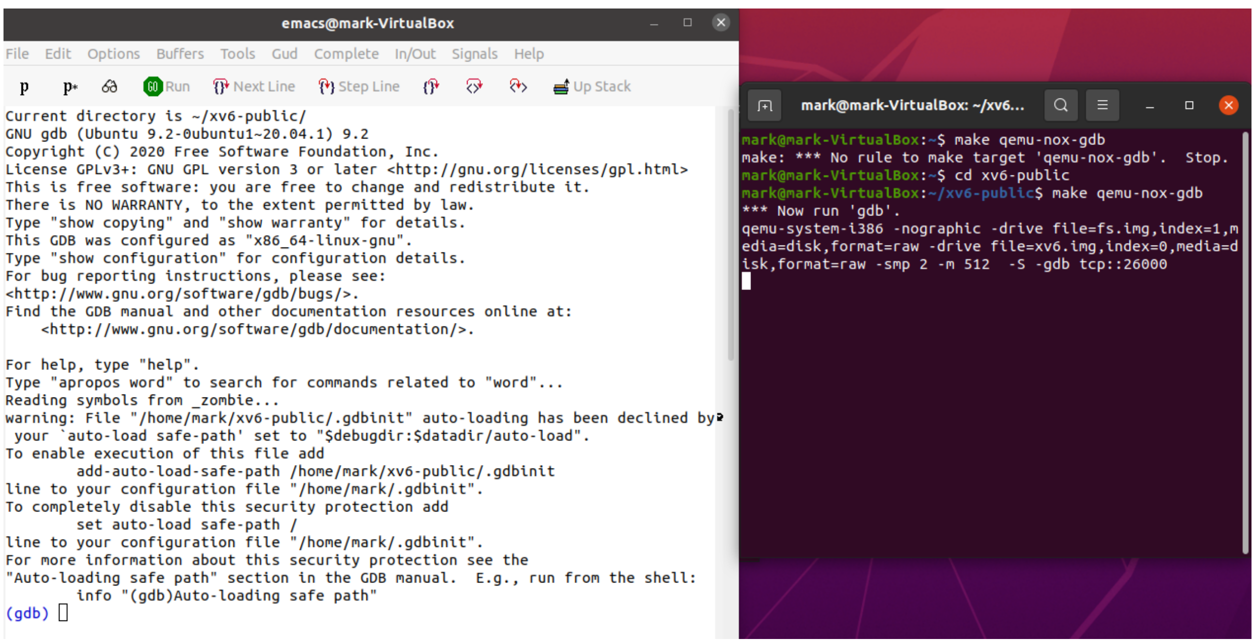
Running **make qemu-nox** …

```
SeaBIOS (version rel-1.15.0-29-g6a62e0cb0dfe-prebuilt.qemu.org)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF91260+1FEF1260 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$
```



```
mark@mark-VirtualBox:~/xv6-public$ make qemu
qemu-system-i386 -serial mon:stdio -drive file=fs.img,i
VNC server running on 127.0.0.1:5900
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 8
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16240
echo           2 4 15096
forktest       2 5 9404
grep           2 6 18460
init           2 7 15680
kill           2 8 15128
ln             2 9 14980
ls             2 10 17608
mkdir          2 11 15224
rm             2 12 15200
sh             2 13 27836
stressfs       2 14 16116
usertests      2 15 67220
wc             2 16 16980
zombie         2 17 14792
console        3 18 0
$
```

Running **make qemu-nox-gdb** and debugging

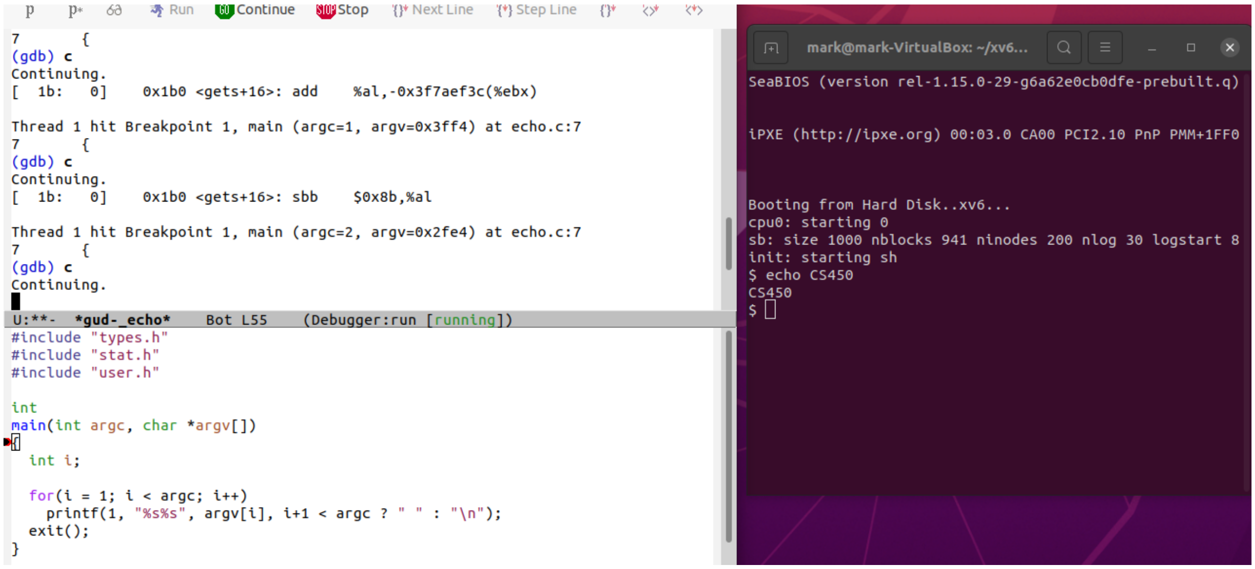



```
Current directory is ~/xv6-public/
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from _zombie...
warning: File "/home/mark/xv6-public/.gdbinit" auto-loading has been declined by
 your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/mark/xv6-public/.gdbinit
line to your configuration file "/home/mark/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/mark/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb)
```

```
mark@mark-VirtualBox:~$ make qemu-nox-gdb
make: *** No rule to make target 'qemu-nox-gdb'.  Stop.
mark@mark-VirtualBox:~$ cd xv6-public
mark@mark-VirtualBox:~/xv6-public$ make qemu-nox-gdb
*** Now run 'gdb'.
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512  -S -gdb tcp::26000
```

Need to add autoload. So I ran **echo "add-auto-load-safe-path $HOME/xv6-public/.gdbinit" > ~/.gdbinit**

Some debugging using **echo.**



By having a breakpoint at main, I can step through **echo.c** and look at what it is doing. Using **s** runs the next line of the program.

Now I am at **printf(…)** as indicated by the black line. The red line is the breakpoint.

```
int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++)
    printf(1, "%s%s", argv[i], i+1 < argc ? " " : "\n");
  exit();
}
```

s one more time will go inside printf method and start stepping through the code there.

Running **print** in gdb also allows me to run expressions and print addresses and variables and such.

```
(gdb) print argv
$5 = (char **) 0x2fe4
(gdb) print argv[0]
$6 = 0x2ff8 "echo"
(gdb) print argv[1]
$7 = 0x2ff0 "CS450"
```

Stepping more and more, you will go to **putc** method, which eventually ends up to **SYSCALL**. This is now kernel debugging.

```
(gdb) si
[  1b: 308]    0x4b8 <printf+120>:      testl  $0xf9830000,(%eax)
0x00000308      16      SYSCALL(write)
(gdb) si
The target architecture is assumed to be i386
=> 0x80105f5b:  push   $0x0
0x80105f5b in ?? ()
(gdb) 
U:**-  *gud-_echo*    Bot L166    (Debugger:run [end-stepping-range
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
```

As you can see there is **?? ()**. Now that we are kernel space, we need to change **symbol-file.**

```
(gdb) symbol-file kernel
Reading symbols from kernel...
(gdb)
U:**-  *gud-_echo*    Bot L168    (Debugger:ru
  jmp alltraps
.globl vector63
vector63:
  pushl $0
  pushl $63
  jmp alltraps
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
.globl vector65
vector65:
  pushl $0
  pushl $65
  jmp alltraps
.globl vector66
-:---  vectors.S      19% L319    (Assembler)
```

Now we know where we are. Stepping through more results in **trap.c** which runs **syscall()**

```
(gdb) p num
$5 = 16
(gdb)
U:**-  *gud-_echo*    Bot L396    (Debugger:run [breakpoint-
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};

void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
-:---  syscall.c      88% L138  Git-master  (C/*l Abbrev)
```

Continue stepping along and you can see it progressively prints out “CS450” one character at a time due to running “echo CS450”