**Lab. 3: Adding a system call**

**This is a group lab. You can make groups of 3 to 4 students. Please specify the contributions of the group members in terms of percentage in your lab report. The whole group will share 100%. For example, in a three students group, if all members contribute to the lab equally, you could list the contributions as follows:**

**Student A, 112019xxxx, 33%**

**Student B, 112019xxxx, 33%**

**Student C, 112019xxxx, 33%**

**Submission：A lab report and the files that you created or modified**

From Lab. 3, we are going to work on a small OS kernel called xv6, which is an instructional OS consisting of a stripped-down version of Unix. To get ready to work within xv6, please start reading the [xv6 book](https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf) as well as the other xv6 resources listed below.

**Resources**

* [xv6 book](https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf)
* [xv6 indexed/cross referenced code](https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf)
* [Official website](https://pdos.csail.mit.edu/6.828/2017/xv6.html)
* [xv6 x86 source code](https://github.com/mit-pdos/xv6-public)
* [xv6 RISC-V source code](https://github.com/mit-pdos/xv6-riscv)

Lab. 3 has two parts, first is to prepare a runnable xv6 environment. The second part is to add a new system call to xv6.

**NOTE: This instruction is for students using xv6 on x86 architecture. If you could finish the labs on RISC-V architecture, you could get 10% extra marks.**

**Part 1. Prepare xv6.**

You'll use two sets of tools in this class: an x86 emulator, [QEMU](https://www.qemu.org/index.html), for running your kernel; and a compiler toolchain, including assembler, linker, C compiler, and debugger, for compiling and testing your kernel. This page has the information you'll need to download and install your own copies. This class assumes familiarity with Linux commands throughout.

We are going to use the openKylin virtual machine installed in Lab. 0.

To test your toolchain, try the following commands:

% objdump -i

The second line should say elf64-x86-64.

% gcc -m32 -print-libgcc-file-name

The command should print something like ：

/usr/lib/gcc/x86\_64-linux-gnu/12/32/libgcc.a

If both these commands succeed, you're all set, and don't need to compile your own toolchain. If the gcc command fails, please check Lab.0 document and install the toolchain.

Once you have the VM up and running, let’s install QEMU and xv6 for this course:

*# install the QEMU*

*#*

**[**vm**]** $ sudo apt-get install qemu-system

# （this command will install simulators for different CPU architectures. In this lab, we are going to use qemu-system-i386）

Run qemu-system-i386 --version command to check your qemu installation.

**[**vm**]** $ qemu-system-i386 --version

**QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-ok1)**

**Copyright (c) 2003-2023 Fabrice Bellard and the QEMU Project developers**

*# finally it's time for setting up xv6*

*#*

**[**vm**]** $ cd ~

**[**vm**]** $ git clone https://github.com/mit-pdos/xv6-public.git xv6

# (it may fail sometimes, you clould retry the command. Or you could just go to website <https://github.com/mit-pdos/xv6-public> to manually download the xv6 source code archive and then unzip it into your VM )

# enter the created xv6 folder

**[**vm**]** $ cd xv6

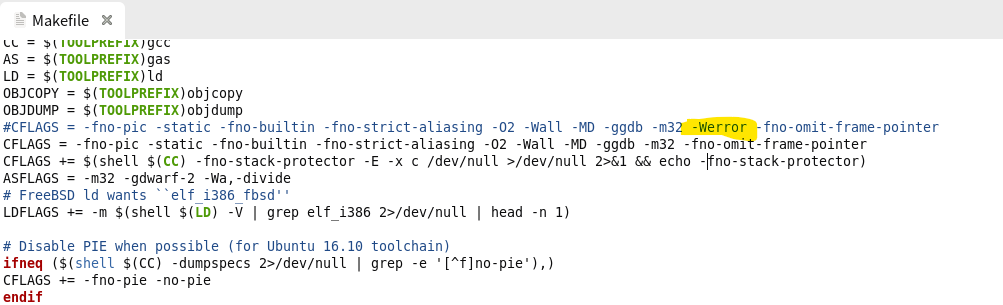
**[**vm**]** $ make

If your compilation faild because of some warnings:

cc1: all warnings being treated as errors

make: \*\*\* [<内置>：mp.o] 错误 1

You could modify the **Makefile** file to turn off the -Werror flag.



When you see something like:

dd if=bootblock of=xv6.img conv=notrunc

输入了 1+0 块记录

输出了 1+0 块记录

512 字节已复制，0.00359033 s，143 kB/s

dd if=kernel of=xv6.img seek=1 conv=notrunc

输入了 394+1 块记录

输出了 394+1 块记录

202224 字节 (202 kB, 197 KiB) 已复制，0.00355693 s，56.9 MB/s

It means your OS image file is ready. Now, you clould use the qemu simulator to start the system.

*# NOTE: allow local gdbinit to be loaded (only done once)*

*#*

**[**vm**]** $ echo "add-auto-load-safe-path $HOME/xv6/.gdbinit" > ~/.gdbinit

To start the xv6, open two terminal windows.

In one terminal, enter make qemu-gdb (or make qemu-nox-gdb). This starts up QEMU, but QEMU stops just before the processor executes the first instruction and waits for a debugging connection from GDB.

$ make qemu-gdb

In the second terminal, from the same directory you ran make, run gdb. (Briefly, gdb -q -iex "set auto-load safe-path /home/csprofs/nael/xv6-master/" . Change the last part to your path to the xv6 directory. You should see something like this,

$ gdb .

GNU gdb 6.8-debian

Copyright (C) 2008 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later

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".

+ target remote localhost:26000

The target architecture is assumed to be i8086

[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b

0x0000fff0 in ?? ()

+ symbol-file kernel

Now set a breakpoint at 0x0010000c by typing br \* 0x0010000c in the gdb window

(gdb) br \* 0x0010000c

type continue You should see something like:

$ continue

The target architecture is assumed to be i386

=> 0x10000c: mov %cr4,%eax

Breakpoint 1, 0x0010000c in ?? ()

(gdb)

IF you don’t want to debug the kernel, you could just rum the OS using command:

$ make qemu

qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512

VNC server running on 127.0.0.1:5900

xv6...

cpu0: starting 0

sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58

init: starting sh

$ ls

. 1 1 512

.. 1 1 512

README 2 2 2286

cat 2 3 15252

echo 2 4 14128

forktest 2 5 8648

grep 2 6 18160

init 2 7 14752

kill 2 8 14220

ln 2 9 14112

ls 2 10 16688

mkdir 2 11 14240

rm 2 12 14220

sh 2 13 28176

stressfs 2 14 14988

usertests 2 15 62300

wc 2 16 15620

zombie 2 17 13792

console 3 18 0

**Part 2. Adding a system call**

Your task is to add a system call to xv6. It will help to start by reading syscall.c (the kernel side of the system call table), user.h (the user-level header for the system calls), and usys.S (the user-level system call definitions). You may add additional files to xv6 to implement this call. For more information about how xv6 implement system calls, you should read the Chapter 3 of xv6 book.

**Exercise 1.**  Create a system call int sys\_wolfie(void \*buf, uint size), which copies an [ASCII art](http://en.wikipedia.org/wiki/ASCII_art) image to a user-supplied buffer, provided that the buffer is large enough. You are welcome to use an ASCII art generator, or draw your own by hand.  
If the buffer is too small, or not valid, return a negative value. If the call succeeds, return the number of bytes copied.

You may find it helpful to review how other system calls are implemented and compiled into the kernel, such as read.

**Exercise 2.**

You will also write a user-level application, called wolfietest.c that gets the image from the kernel, and prints it to the console.

You will have to modify the Makefile to add your user application in the compile task.