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

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)

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.

**Part 1. Prepare xv6.**

You'll use two sets of tools in this class: an x86 emulator, [QEMU](https://pdos.csail.mit.edu/6.828/2017/tools.html#qemu), for running your kernel; and a [compiler toolchain](https://pdos.csail.mit.edu/6.828/2017/tools.html#chain), 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 Unix commands throughout.

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

To test your toolchain, try the following commands:

% objdump -i

The second line should say elf32-i386.

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

The command should print something like /usr/lib/gcc/i486-linux-gnu/*version*/libgcc.a or /usr/lib/gcc/x86\_64-linux-gnu/*version*/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

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

*#*

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

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

...

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

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

*# 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)

**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 of Wolfie 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 Wolfie 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.