XV6 Installation and sample Program

Home page of the xv6 project: http://pdos.csail.mit.edu/6.828/2017/xv6.html
Full code of xv6 (PDF): https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf
Code explanation: https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf

XV6 installation steps on Ubuntu or any Debian Flavor Linux

vishnu@mannava ~ $ sudo apt-get update
vishnu@mannava ~ $ sudo apt-get upgrade
vishnu@mannava ~ $ sudo apt install make
vishnu@mannava ~ $ sudo apt-get install build-essential
vishnu@mannava ~ $ sudo apt-get install gcc-multilib
vishnu@mannava ~ $ sudo apt-get install qemu
vishnu@mannava ~ $ sudo apt-get install git
vishnu@mannava ~ $ git clone git://github.com/mit-pdos/xv6-public.git xv6
vishnu@mannava ~ $ ls
vishnu@mannava ~ $ cd xv6/
vishnu@mannava xv6 $ ls
vishnu@mannava xv6 $ gedit Makefile
vishnu@mannava xv6 $ make
vishnu@mannava xv6 $ make qemu-nox

XV6 installation steps on Fedora, Centos and Redhat Linux

vishnu@mannava ~ # yum check-update
vishnu@mannava ~ # yum update
vishnu@mannava ~ # yum install qemu
vishnu@mannava ~ # yum install git
vishnu@mannava ~ # git clone git://github.com/mit-pdos/xv6-public.git xv6
vishnu@mannava ~ # ls
vishnu@mannava ~ # cd xv6
vishnu@mannava ~ # ls
vishnu@mannava ~ # nano Makefile
vishnu@mannava ~ # make
vishnu@mannava ~ # make qemu-nox

A separate window should appear containing the display of the virtual machine. After a few seconds, QEMU's virtual BIOS will load xv6's boot loader from a virtual hard drive image contained in the file `xv6.img`, and the boot loader will in turn load and run the xv6 kernel. After everything is loaded, you should get a '`$`' prompt in the xv6 display window and be able to enter commands into the rudimentary but functional xv6 shell. For example, try:

```
$ ls
.               1 1 512
..              1 1 512
README          2 2 1844
cat             2 3 12129
...
$ echo Hello!
Hello!
$ cat README
xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix
Version 6 (v6).  xv6 loosely follows the structure and style of v6,
...
$ grep run README
To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make".
Then run "make TOOLPREFIX=i386-jos-elf-".
...
$ cat README | grep run | wc
6 70 376
$ echo My New File >newfile
$ cat newfile
My New File
```

The small file system you're examining and modifying here is resides on a second virtual disk, whose initial contents QEMU initializes from the file `fs.img`. Later in the course you will examine how xv6 accesses and modifies this file system.

Adding a system call in XV6 OS. We will just add a simple helloworld systemm call which will print hello world

Write a simple system call on XV6 in order to understand how they're implemented. I've used these steps

1. In `syscall.c`, declared `extern int sys_hello(void)` and added `[SYS_hello] sys_hello` into `static int (*syscalls[])(void)` array.

```
 99 extern int sys_write(void);
100 extern int sys_uptime(void);
101 extern int sys_hello(void);
102
103 static int (*syscalls[])(void) = {
120 [SYS_unlink]  sys_unlink,
121 [SYS_link]    sys_link,
122 [SYS_mkdir]   sys_mkdir,
123 [SYS_close]   sys_close,
124 [SYS_hello]   sys_hello,
125 };
126
127 void
128 syscall(void)
129 {
```

2. In `syscall.h`, defined `SYS_hello` as call number 22

```
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 #define SYS_hello  22
24
```

3. In `user.h`, declared the function prototype as `int hello (void);`

```
24 int sleep(int);
25 int uptime(void);
26 int hello(int);
27
28 // ulib.c
29 int stat(char*, struct stat*);
```

4. In `usys.S`, added `SYSCALL(hello)` to the macro

```
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(hello)
```

5. In `sysproc.c`, added the function `sys_hello(void)` at the bottom

```
int sys_hello(void)
{
  cprintf ("Hello World System Call\n");
  return 0;
      }
```

6. Created `hello.c` which simply calls the `hello()` system call

```
#include "types.h"
#include "user.h"
int main (void) {
        //printf (1,"Hello World\n");
        hello();
        exit();
}
```

7. Added `hello.c` to the `Makefile` and ran the code

The xv6 kernel is a port of an old UNIX version 6 from PDP-11 (the machine it was originally built to run on) to a modern system, Intel x86, and the idea to make it a source of study stems from Lions' famous commentary on Unix SV6. The xv6 kernel does is compact and concise, and thus represents a great way to understand many of the fundamentals that underly operating systems without the code deluge that of- ten accompanies such a pursuit. We'll specifically trace what happens in the code in order to understand a **system call**. System calls allow the operating system to run code on the behalf of user requests but in a protected manner, both by jumping into the kernel (in a very specific and restricted way) and also by simultaneously raising the privilege level of the hardware, so that the OS can perform certain restricted operations.

Adding a User Program to xv6

It is pretty much simple and straightforward to write a user level C program for xv6 and making it available for the user at the shell prompt. When we go forward and implement much advanced things on the xv6 OS, most of the times we need to test those features by using a user level program. Therefore, knowing how to write such programs and adding it into the xv6 source code for compilation is a necessary thing.

First of all, let's create a C program like the following. We save it inside the source code directory of xv6 operating system with the name **my.c** or whatever the name you prefer.

```
1 // A simple program which just prints something on screen
2
3 #include "types.h"
4 #include "stat.h"
5 #include "user.h"
6
7 int
8 main(void)
9 {
10  printf(1, "My first user program on xv6\n");
11  exit();
12 }
```

Now, we have to edit the **Makefile** of the xv6 source code in order to let the compiler know that we have a program like this and we need it to be compiled with other system programs. In the **Makefile**, there are two places in which we need to put entries.

Find the place with some lines like the following. We have to add a line as shown below to notify about our new program.

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _my\
```

Similarly, find the place with the lines like below. Add an entry as shown to indicate that we have a program called **my.c** there.

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    my.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Now, our Makefile and our user program is ready to be tested. Enter the following commands to compile the whole system.

```
make clean
make
```

Now, start xv6 system on QEMU and when it booted up, run **ls** command to check whether our program is available for the user. If yes, give the name of that executable program, which is in my case is **my** to see the program output on the QEMU emulator window.