Instructions Document For The Minor Project Setup

1. Working with QEMU and xv6 OS

1.1. What is QEMU?

QEMU is a CPU and machine emulator that allows you to run one operating system inside another by emulating hardware components (CPU, memory, and devices).

- QEMU (Quick EMUlator) is a powerful, free, and open-source platform for hardware virtualization and emulation.
- It enables running operating systems and applications designed for one hardware architecture on a different one (e.g., Android (ARM) on an x86 PC).
- supports diverse guest operating systems and emulates a wide range of hardware components.

In OS labs, we use QEMU to boot small teaching kernels (like xv6) without touching the host machine. Think of it as a virtual computer where you can quickly rebuild and reboot your own kernel.

1.2. What is xv6 OS?

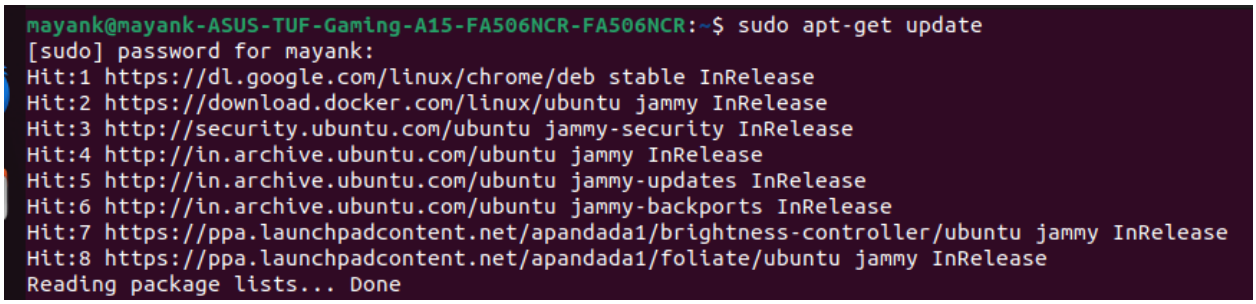
xv6 is a small, Unix-like teaching operating system from MIT: a clean reimplementation of Sixth Edition Unix in ANSI C for modern x86 and RISC-V, designed to teach core OS concepts in a codebase that students can read end-to-end in a semester. It pairs a compact kernel and userland with an openly available commentary book that walks through processes, memory, traps/system calls, locking, scheduling, and a simple file system

1.3. Why QEMU for xv6?

- No need to install to bare metal.
- Fast rebuild–reboot cycle for kernel hacking.
- Deterministic and portable across lab machines.

1.4. Setting up QEMU with xv6 OS

```
sudo apt-get update
```

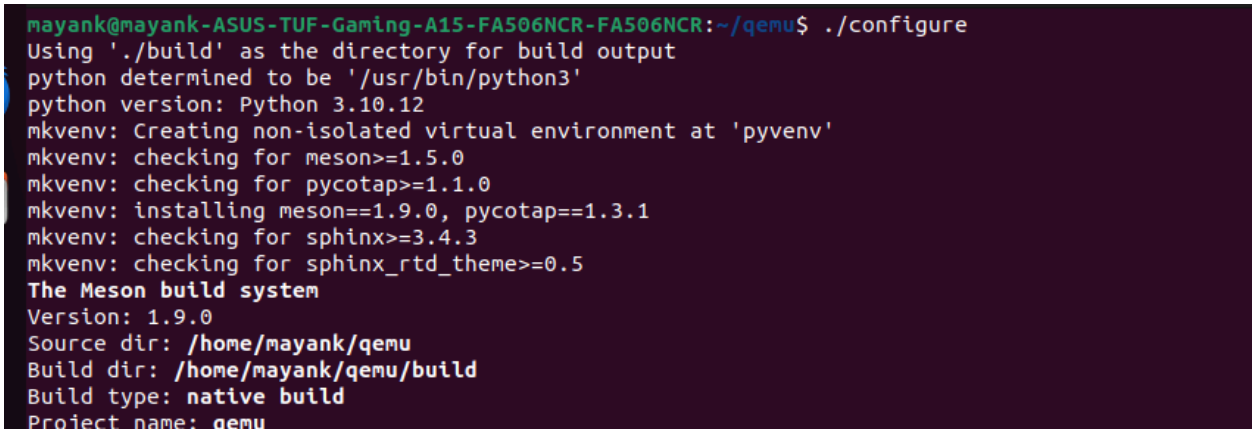



```
git clone https://gitlab.com/qemu-project/qemu.git
   cd qemu
   ./configure
   make
```

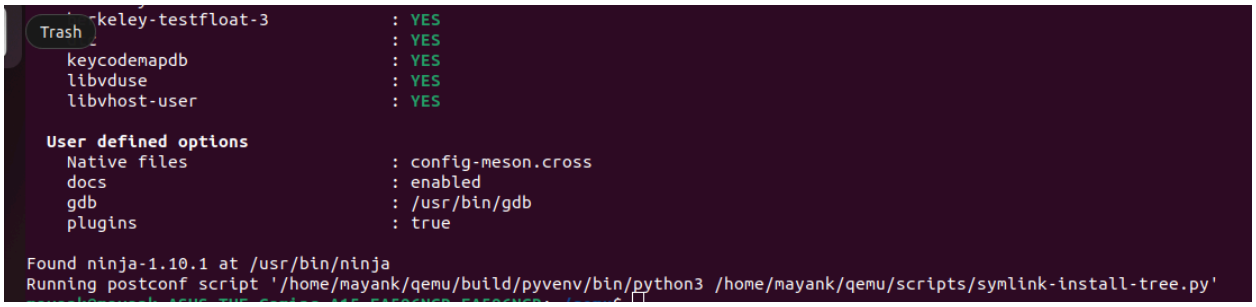
Note: ./configure may give errors if sphinx==6.2.1 and 'sphinx_rtd_theme==1.2.2' are not available. You can simply run:
`pip install sphinx==6.2.1` , and
`pip install sphinx-rtd-theme==1.2.2`

If it gives ERROR: cannot find ninja:
`sudo apt-get install ninja-build`

If everything runs fine you will get a message, something like “Running postconf script '/home/mayank/qemu/build/pyvenv/bin/python3 /home/mayank/qemu/scripts/symlink-install-tree.py' “



.....



Then we proceed to build it using the `make` command. It might take a while. It everything is fine you will get a message; something similar to “make[1]: Leaving directory '/home/mayank/qemu/build' “

.....

```
[7140/7152] Linking target tests/qtest/pnv-spi-seeprom-test
[7141/7152] Compiling C object tests/qtest/pnv-host-i2c-test.p/pnv-host-i2c-test.c.o
[7142/7152] Linking target tests/qtest/pnv-host-i2c-test
[7143/7152] Compiling C object tests/qtest/rtas-test.p/rtas-test.c.o
[7144/7152] Linking target tests/qtest/rtas-test
[7145/7152] Compiling C object tests/qtest/virtio-ccw-test.p/virtio-ccw-test.c.o
[7146/7152] Compiling C object tests/qtest/sifive-e-aon-watchdog-test.p/sifive-e-aon-watchdog-tes
[7147/7152] Linking target tests/qtest/sifive-e-aon-watchdog-test
[7148/7152] Compiling C object tests/qtest/riscv-csr-test.p/riscv-csr-test.c.o
[7149/7152] Linking target tests/qtest/riscv-csr-test
[7150/7152] Linking target tests/qtest/virtio-ccw-test
[7151/7152] Compiling C object tests/qtest/m48t59-test.p/m48t59-test.c.o
[7152/7152] Linking target tests/qtest/m48t59-test
make[1]: Leaving directory '/home/mayank/qemu/build'
mayank@mayank-ASUS-TUF-Gaming-A15-FA506NCR-FA506NCR:~/qemu$
```

Next, Download the xv6 code tarball we have put together for this miniproject: [x64 image compressed](). Please note that there are minor changes to the the [original xv6 code tarball]() (which is the latest x86 version of the code from [the xv6 github repo]()), to handle some gcc compiler related issues, and also add a simple test case. The original xv6 code should also work fine on most Linux machines.

Decompress (untar) the folder: `tar -zxvf cs236-xv6-linux.tgz`

```
mayank@mayank-ASUS-TUF-Gaming-A15-FA506NCR-FA506NCR:~/qemu$ tar -zxvf cs236-xv6-linux.tgz
xv6-public/
xv6-public/._cuth
xv6-public/._file.h
xv6-public/._.git
xv6-public/._entryother.S
xv6-public/._defs.h
xv6-public/._stat.h
xv6-public/._dot-bochsrc
xv6-public/fs.c
xv6-public/mmu.h
xv6-public/._swtch.S
xv6-public/._gdbutil
xv6-public/._zombie.c
xv6-public/sleeplock.h
xv6-public/._fs.h
xv6-public/sleep1.p
xv6-public/._README
xv6-public/ioapic.c
xv6-public/._printpcs
xv6-public/lapic.c
xv6-public/file.h
xv6-public/vm.c
xv6-public/fs.h
```

Go to the uncompressed xv6 folder, and compile xv6 by typing for following inside the folder.

```
cd xv6-public
make clean
make
```

```
mayank@mayank-ASUS-TUF-Gaming-A15-FA506NCR-FA506NCR:~/qemu$ cd xv6-public/
mayank@mayank-ASUS-TUF-Gaming-A15-FA506NCR-FA506NCR:~/qemu/xv6-public$ make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie _mytest
```

If everything runs fine, then you will get the output similar to the screenshot below:

.....

```
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0161036 s, 318 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000134801 s, 3.8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
393+1 records in
393+1 records out
201372 bytes (201 kB, 197 KiB) copied, 0.000984293 s, 205 MB/s
```

Now, Run xv6 inside QEMU as follows.

```
make qemu
```

Or

```
make qemu-nox
```

If everything is fine, you will see the following screen:

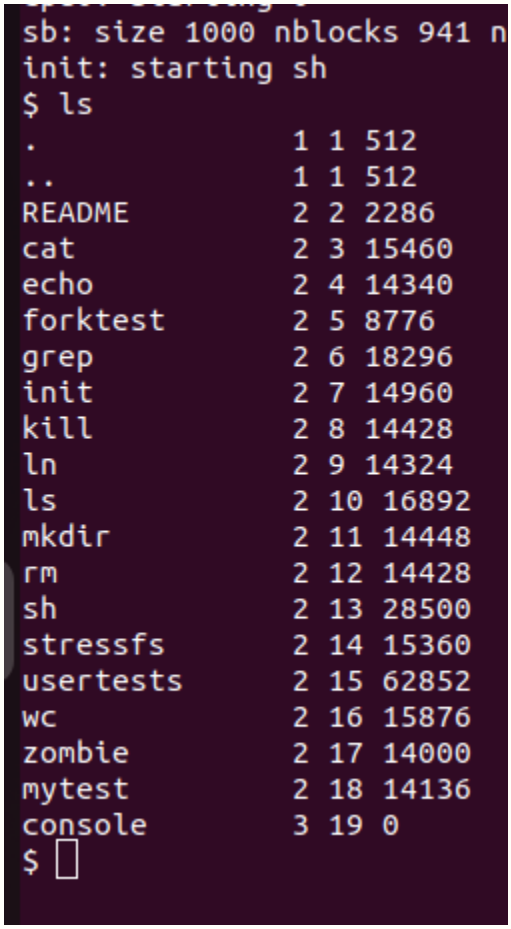
```
SeaBIOS (version rel-1.17.0-0-gb52ca86e094d-prebuilt.qemu.org)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFD1EC0+1EF31EC0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

It starts with running its shell/terminal.

You can now run simple commands like "ls" in the xv6 terminal. You will see something like this printed on screen.

```
$ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15492
echo           2 4 14376
forktest       2 5 8812
grep           2 6 18328
init           2 7 14996
kill           2 8 14460
ln             2 9 14356
ls             2 10 16924
```

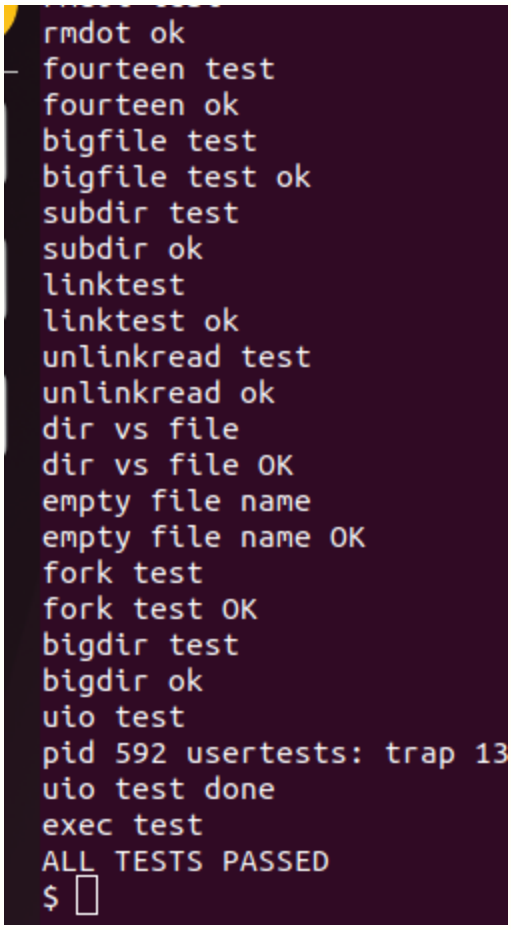
```
mkdir           2 11 14484
rm              2 12 14464
sh              2 13 28512
stressfs        2 14 15392
usertests       2 15 62884
wc              2 16 15912
zombie          2 17 14032
console         3 18 0
```



You can also test your code by running all the user tests in xv6. Run the following command in the xv6 terminal.

```
usertests
```

If this command displays "All tests passed", then you are good to go.



To exit QEMU, type Ctrl+A X (First press Ctrl + A, where "A" is just key a, not the alt key, then release the keys, afterwards press X.)

2. Writing and testing new code

In this miniproject, you will write code inside the xv6 kernel to add and extend kernel features. It is recommended that you create separate copies of this xv6 folder for every assignment you solve (so that you do not disturb the original files).

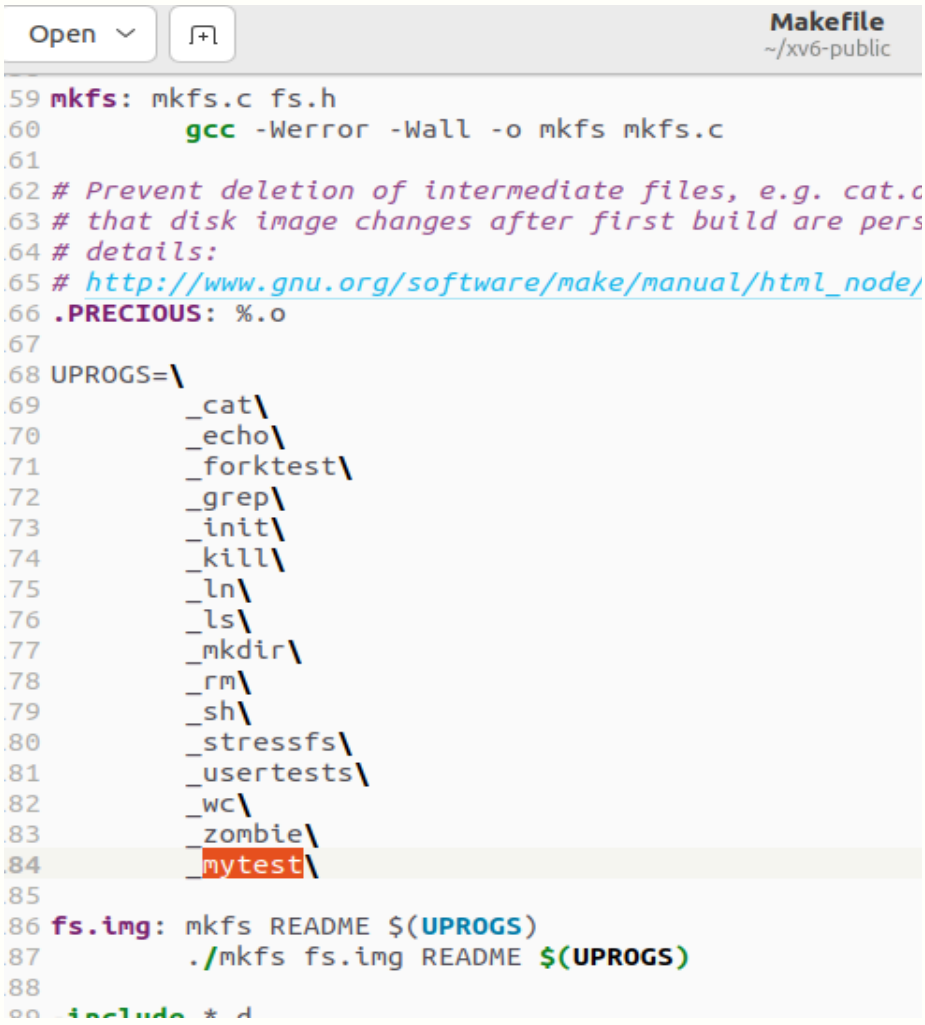
Once you have added new kernel features, you will want to test your code. For this, you will need to write simple user programs or testcases that invoke OS functionality. Below are instructions on how you can write simple user programs in xv6.

We have provided you with a simple user program "mytest.c" as part of the custom xv6 tarball provided on this page. This program, shown below, just prints a message to screen, but you can write more complex code that invokes other xv6 system calls as well, as you solve the other labs.

```
#include "types.h"
#include "user.h"
int main(int argc, char *argv[])
{
    printf(1, "Hey, I am trying xv6\n");
    exit();
}
```

This program has been added to the xv6 folder given to you.

In order to compile this file as part of the xv6 compilation process, we have made the following changes to the xv6 Makefile.



1. Add “_mytest\” to the list of executables
2. Add “mytest.c” to the list of programs to compile

```
50
51 EXTRA=\
52         mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
53         ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
54         printf.c umalloc.c mytest.c\
55         README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
56         .gdbinit.tmpl gdbutil\
```

Now, run “make clean && make && make qemu” to start xv6. On typing “mytest” on the terminal, your compiled program will run and print.

```
$ mytest
Hey, I am trying xv6
$
```

3. Add a new system call: a simple `hello()` example

Goal: create a tiny syscall that prints a kernel message hello from kernel and returns 0.

3.1 Some important files in XV6 OS :

`syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.

`syscall.c` contains helper functions to parse system call arguments, and pointers to the actual system call implementations.

`user.h` contains the system call definitions in xv6.

`usys.S` contains a list of system calls exported by the kernel, and the corresponding invocation of the trap instruction.

Then, there are some other important files in xv6 for process management, memory management and threads and synchronization, which you will have to study based on your specific mini-project topic out of these (these will be mentioned in the problem statement for your mini-project topic, you will have to study that codes in detail to be able to implement your own “new” system calls as per the topic).

3.2. Pick a syscall number

Open `syscall.h` and add a new number at the end of the list (keep order contiguous):

Open

*syscall.h
~/xv6-public

```
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
15 #define SYS_uptime 14
16 #define SYS_open   15
17 #define SYS_write  16
18 #define SYS_mknod  17
19 #define SYS_unlink 18
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 #define SYS_hello  22
```

3.3. Declare the user-space wrapper

Add a prototype to `user.h` so user programs can call it:

```
// in user.h
int hello(void);
```

3.4. Add the stub to `usys.S`

Map the user call to a trap into the kernel:

```
// in usys.S
SYSCALL(hello)
```

Open

*usys.S
~/xv6-public

```
#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)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(hello)
```

3.5. Kernel side: add syscall entry in the table

Tell the kernel which C function to call for `SYS_hello`.

In `syscall.c`, add an external declaration and an entry in the `syscalls[]` table:

```
// in syscall.c
extern int sys_hello(void);

static int (*syscalls[])(void) = {
[SYS_fork] sys_fork,
[SYS_exit] sys_exit,
...
[SYS_hello] sys_hello,
```

```
};
```

Open ▾ *syscall.c ~/xv6-public

```
94 extern int sys_link(void);
95 extern int sys_mkdir(void);
96 extern int sys_mknod(void);
97 extern int sys_open(void);
98 extern int sys_pipe(void);
99 extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_hello(void);
107
108 static int (*syscalls[])(void) = {
109 [SYS_fork]    sys_fork,
110 [SYS_exit]    sys_exit,
111 [SYS_wait]    sys_wait,
112 [SYS_pipe]    sys_pipe,
113 [SYS_read]    sys_read,
114 [SYS_kill]    sys_kill,
115 [SYS_exec]    sys_exec,
116 [SYS_fstat]   sys_fstat,
117 [SYS_chdir]   sys_chdir,
118 [SYS_dup]     sys_dup,
119 [SYS_getpid]  sys_getpid,
120 [SYS_sbrk]    sys_sbrk,
121 [SYS_sleep]   sys_sleep,
122 [SYS_uptime]  sys_uptime,
123 [SYS_open]    sys_open,
124 [SYS_write]   sys_write,
125 [SYS_mknod]   sys_mknod,
126 [SYS_unlink]  sys_unlink,
127 [SYS_link]    sys_link,
128 [SYS_mkdir]   sys_mkdir,
129 [SYS_close]   sys_close,
130 [SYS_hello]   sys_hello,
131 };
132
133 void
134 syscall(void)
135 {
```

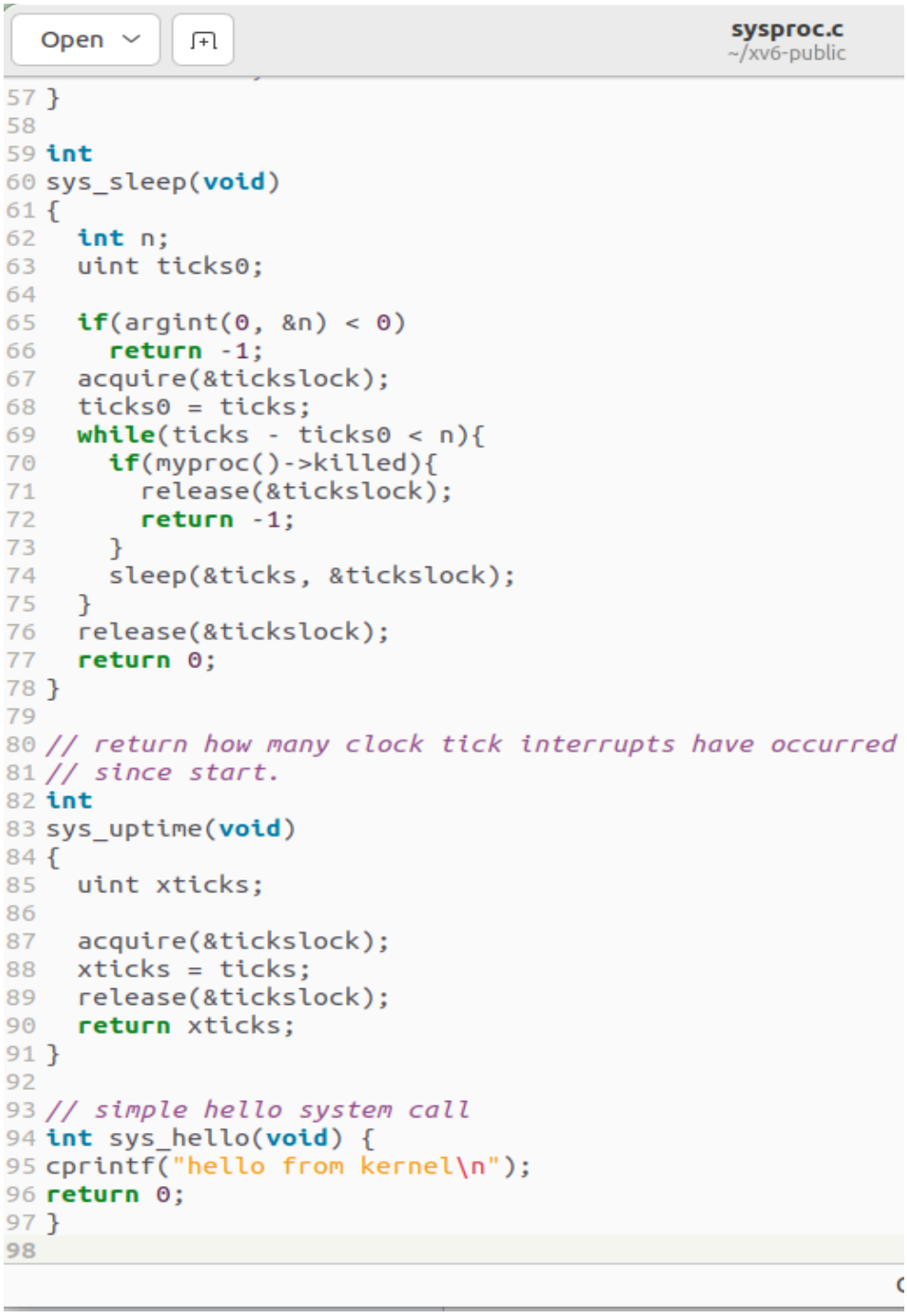
3.6. Implement the kernel function

A simple place is `sysproc.c` (or create a new file and add it to Makefile).

```
// in sysproc.c
#include "types.h"
#include "defs.h"

int sys_hello(void) {
```

```
cprintf("hello from kernel\n");
return 0;
}
```

Here, `cprintf` prints to the xv6 console from kernel space.



3.7. Write a user program to test `hello()`

Create `hello_test.c` under the xv6 tree (usually alongside other user programs like `echo.c`).

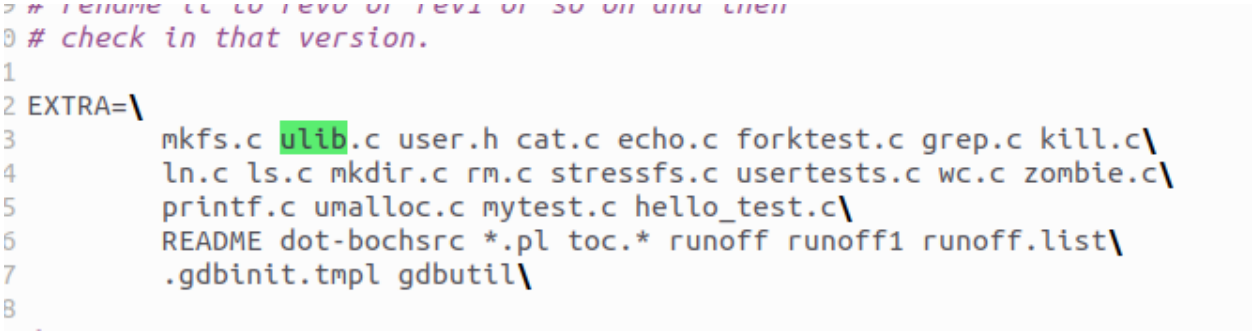
```
// hello_test.c
#include "types.h"
#include "user.h"
```

```
int main(int argc, char *argv[]) {
int r = hello();
printf(1, "hello() returned %d\n", r);
exit();
}
```

3.8. Add it to the build

Open `Makefile` and:

Add the new program to the UPROGS list (note the leading underscore):

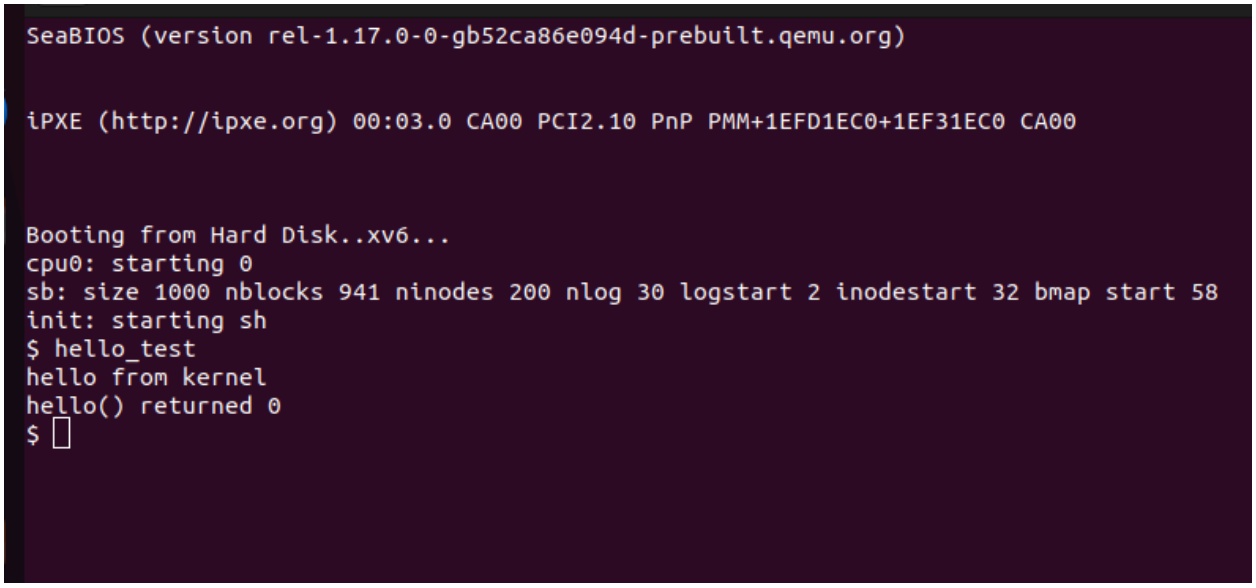


```
UPROGS=\
        _cat\
        _echo\
        _forktest\
        _grep\
        _init\
        _kill\
        _ln\
        _ls\
        _mkdir\
        _rm\
        _sh\
        _stressfs\
        _usertests\
        _wc\
        _zombie\
        _mytest\
        _hello_test\
```

Then rebuild and run: `make clean && make && make qemu-nox`

OR `make clean && make && make QEMU=/home/mayank/qemu/build/qemu-system-i386 qemu-nox`

Then try typing `hello_test` in the xv6 shell:



Based on this information, you will e asked to create your own “new” system calls!