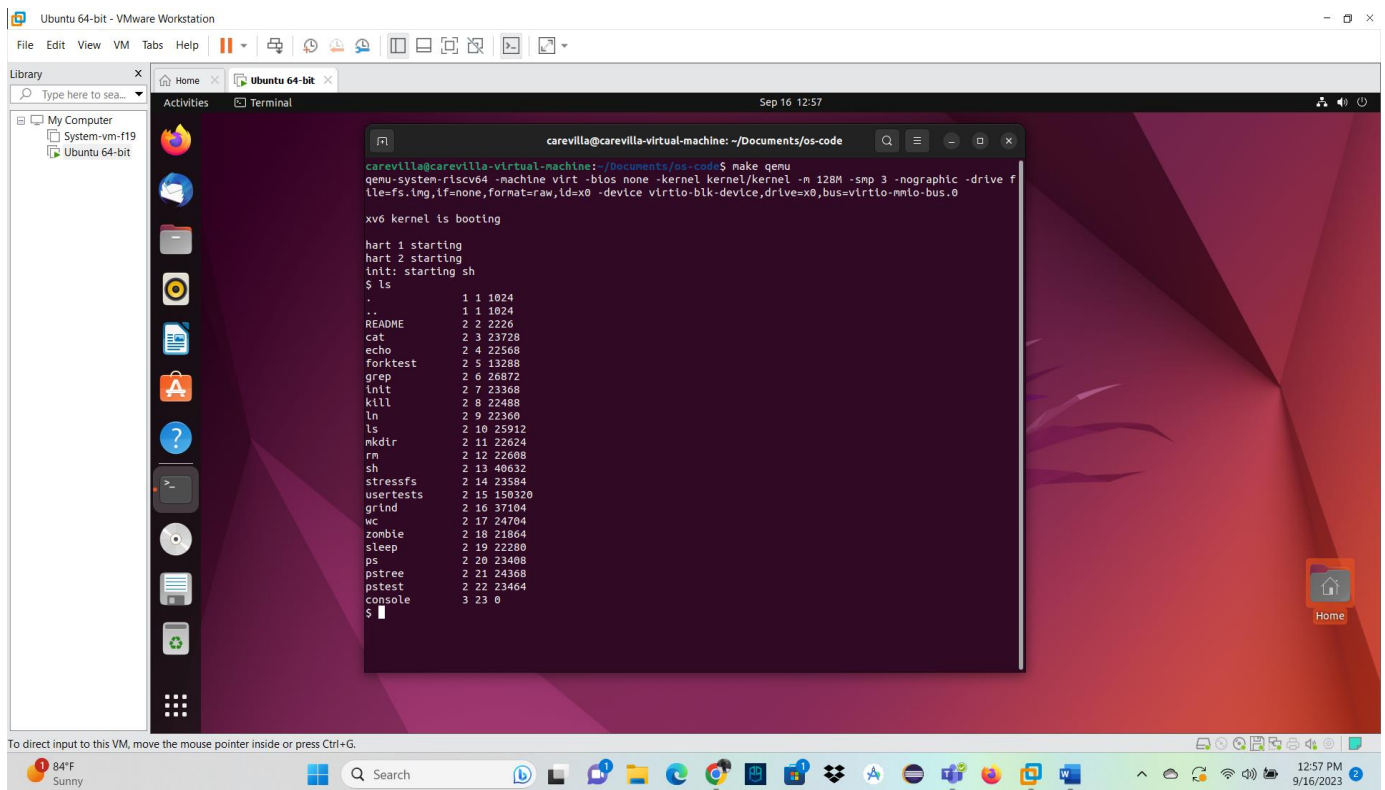


Homework 1: Introduction to xv6

Task 1: Boot xv6 and Explore Utilities

For this semester I will be running Ubuntu 22.04 with the help of VMware to emulate the RISC-V hardware. The first task of this lab was booting up the xv6 emulator which could be obtained from GitHub. I successfully duplicated the repository and called it OS-code. From here I created a separate branch and named it Homework1. All my work is in that repository and will be explained in detail during Task 2 implementation. Below is a photo of the xv6 emulator booted up and running with the ls command.



```
carevilla@carevilla-virtual-machine: ~/Documents/os-code
carevilla@carevilla-virtual-machine:~/Documents/os-code$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive f
llefds.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

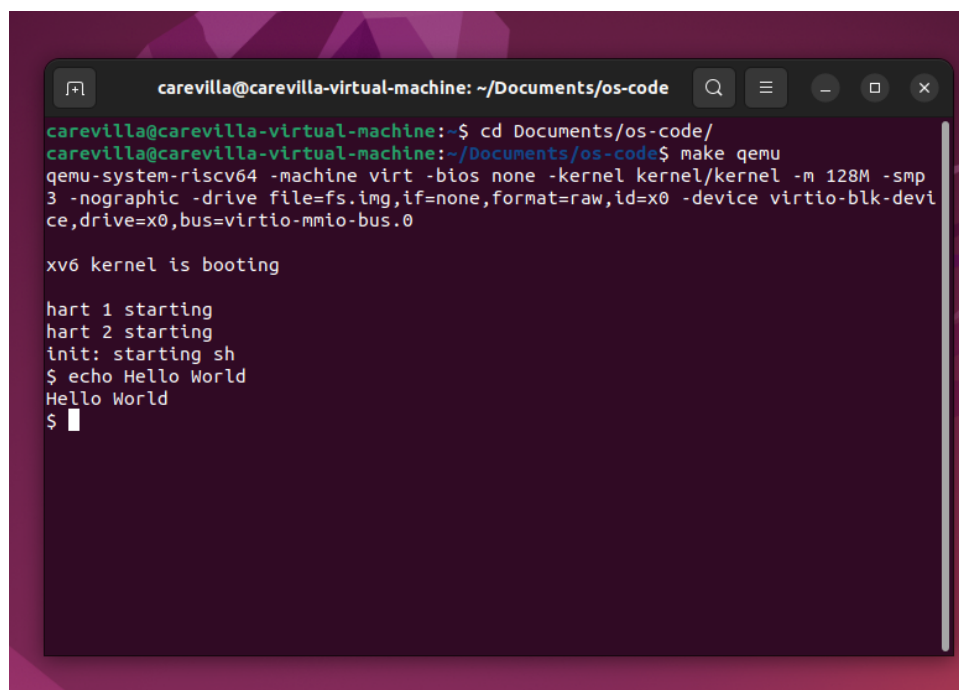
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
cat       2 3 23728
echo      2 4 22568
forktest  2 5 13288
grep      2 6 26872
init      2 7 23368
kill      2 8 22488
ln        2 9 22360
ls        2 10 25912
mkdir     2 11 22624
rm        2 12 22608
sh        2 13 40632
stressfs  2 14 23384
usertests 2 15 150320
grind     2 16 37104
wc        2 17 24704
zombie    2 18 21864
sleep     2 19 22280
ps        2 20 23408
pstree    2 21 24368
pstest    2 22 23464
console   3 23 0
$
```

After successfully booting up and configuring the environment We were asked to explore three additional user commands and explain how they work and what they do.

1. The echo command:

- a. The echo user command is a simple program which will output to the console the input that it is given. That is, it will print to the stdout all the arguments given to it. Arguments are given as strings that are separated by whitespace. While looking at the source code in the file *echo.c* I summarize the code works as follows.
 - i. We can see the program takes two arguments one being an integer, *argc* (argument count), and a pointer to an array of characters. *argv* being the arguments passed.
 - ii. In the example “echo Hello World” the function is called and *argc* gets a value of 3 and the array has for its contents: [“echo”, “Hello”, “World”] from here it’s a standard for loop starting at index one. The reason we start at index 1 is because position 0 is where the command is stored. So the first pass we would write to the *stdout* the string in index one. We have an if statement to help us know when to place a space between each string and a new line when we finish the last element.
 - iii. Below is a quick sample run of the echo command



```
carevilla@carevilla-virtual-machine: ~/Documents/os-code
carevilla@carevilla-virtual-machine:~/Documents/os-code$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

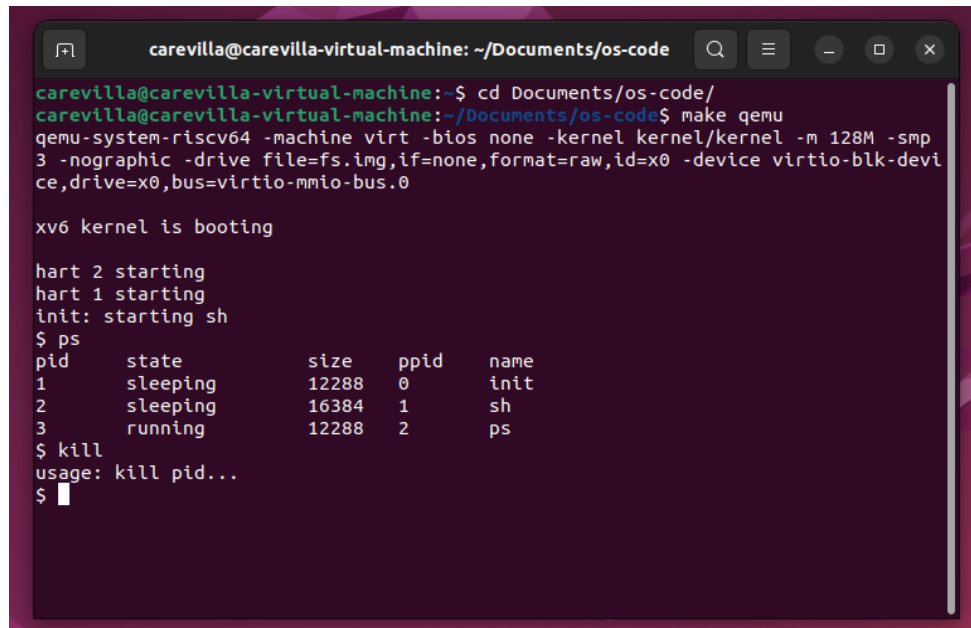
hart 1 starting
hart 2 starting
init: starting sh
$ echo Hello World
Hello World
$
```

2. The kill command:

- a. The user command kill is a program to enable us to kill/terminate processes that are currently running on the operating system by passing it their process ID. Looking at the source code in the file *kill.c* the program works as follows:
 - i. The program takes two values as input, an integer, *argc*, which holds the argument count and a pointer to a pointer that hold the array of process ID’s to be killed in the form of character arrays.
 - ii. The code is straight forward, if the arguments that are passed are less than 2 than the program will print its usage statement and exit with a -1 status

to indicate an error. This message will appear if no process IDs are given. If given multiple PID's it will traverse the length of the argument array starting at index 1 and use the *atoi* function to parse the char array into an integer to pass the PID to the kernel kill function.

- iii. Below is an example of the kill usage error. I did not apply the kill functions to my running programs.



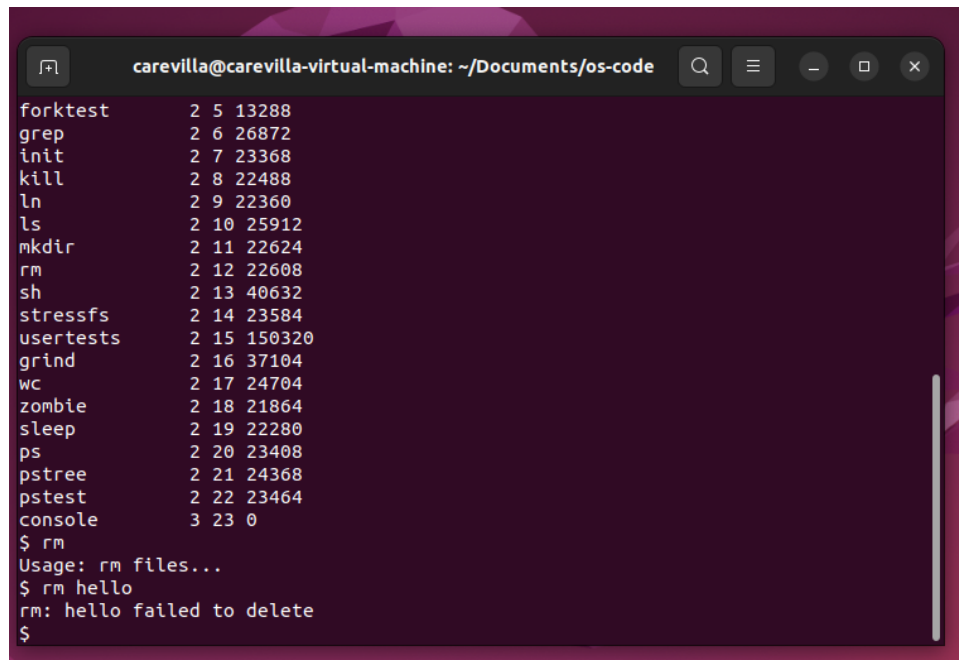
```
carevilla@carevilla-virtual-machine: ~/Documents/os-code
carevilla@carevilla-virtual-machine:~/Documents/os-code$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ ps
pid      state      size      ppid      name
1        sleeping   12288     0         init
2        sleeping   16384     1         sh
3        running    12288     2         ps
$ kill
usage: kill pid...
$
```

3. The rm command:

- a. The *rm* user command is used to remove unwanted files from the file system. It takes in the name of what is to be removed and calls the unlink function with the arguments that are passed to it.
 - i. The program takes two arguments one being an integer, *argc* (argument count), and a pointer to an array of characters being the arguments passed contained in character arrays or strings.
 - ii. Similarly, to the kill code, if less than two arguments are passed it will print a usage message and exit with an error code. If more than two arguments are passed the array is traversed and at each index it ensures the file can properly be unlinked. If at any point the file wasn't unlinked properly it will display a failed message and exit the program.
 - iii. Below is a sample run using the rm command and being prompted with a failed message



```
carevilla@carevilla-virtual-machine: ~/Documents/os-code
forktest      2  5 13288
grep          2  6 26872
init          2  7 23368
kill          2  8 22488
ln            2  9 22360
ls            2 10 25912
mkdir         2 11 22624
rm            2 12 22608
sh            2 13 40632
stressfs      2 14 23584
usertests     2 15 150320
grind         2 16 37104
wc            2 17 24704
zombie        2 18 21864
sleep         2 19 22280
ps            2 20 23408
pstree        2 21 24368
pstest        2 22 23464
console       3 23 0
$ rm
Usage: rm files...
$ rm hello
rm: hello failed to delete
$
```

Task 2: Implement the uptime utility:

For task 2 we were asked to implement the uptime function into the xv6 processor. With the knowledge I obtained from task 1 on how to locate the code for each command, I approached this task in a straightforward manner. I first looked at the user header file (*user.h*) to obtain the syntax for the kernel uptime function which gave me the input arguments, null and the return value, integer. I then looked at the *sysproc.c* code to see how the uptime () system call is implemented. I also reviewed the *usys.S* file for the assembler code that governs the linkage of for the kernel implementation.

I then proceeded to implement my version of uptime in the user directory as requested. To do this, I first created a new file in the user directory called *uptime.c*. Once the file was created, I included the header files that were required to accomplish this task. I then created a main function to start my C code. My code consists of a simple integer that stores the value that is returned when the uptime system call is executed. This is followed by a print statement that formats the output to the console. The last step is to call the *exit()* function to terminate the program and return a 0. Finally, the last task was to add my function to the Makefile to be able to compile and run the command. Below is a picture of the source code and the successful implementation of the uptime command. The source code can also be found on my Github repository branch Homework 1.

