

Operating System Lab Project 1 Report

Reza Abdoli

Mohammad Reza Alavi

Farbod Azimmohseni

Commit Id: 7450b9a5f599cd135b6b55e3d02a049a7fff67c4

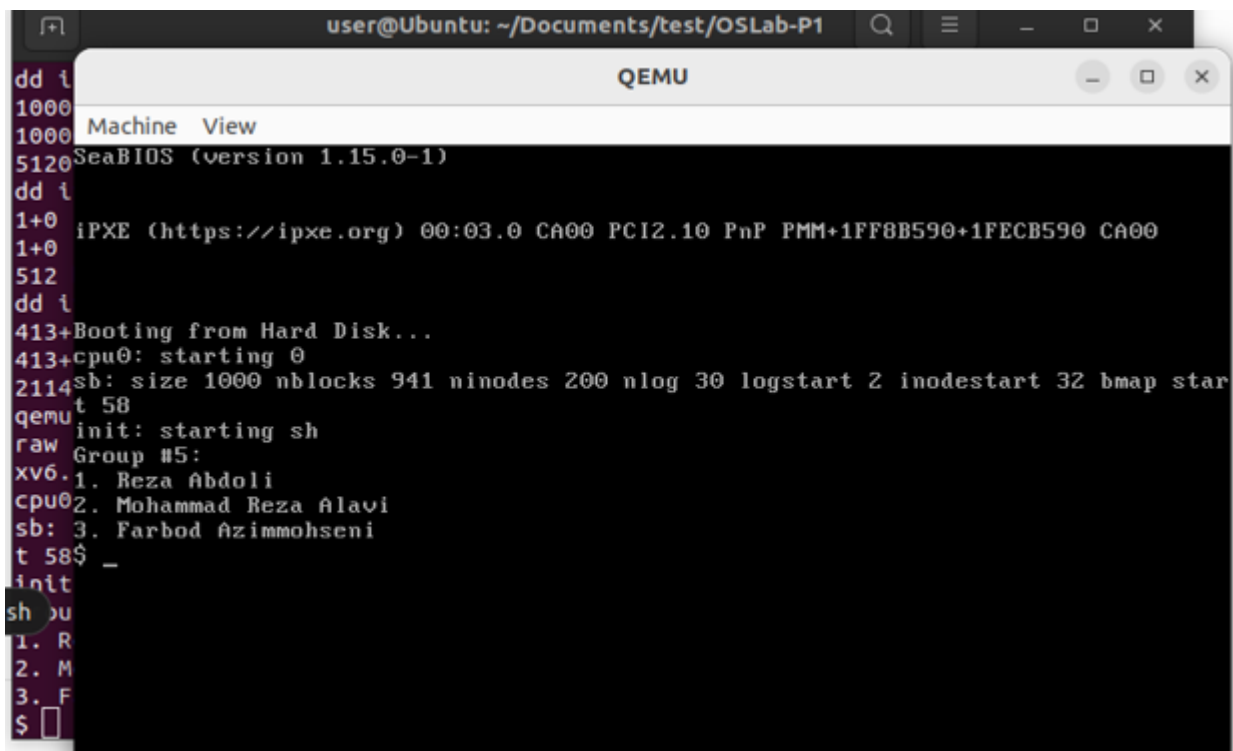
October 22, 2023

Introduction to XV6 Operating System

1. Xv6 is an educational operating system written in ANSI C for Intel x86 and RISC-V processors. It follows a *monolithic kernel* architecture, where all components of the operating system run in kernel mode with full hardware privileges. This design choice eliminates the need for the operating system designer to determine which parts require full hardware access. However, this approach also has its downsides. One downside is that the interfaces in a monolithic kernel can be complex, making it easier for an operating system designer to make mistakes during development. Furthermore, if a single error occurs in a *monolithic* kernel, it can cause the entire system to stop working.
2. Xv6 provides process isolation, preventing one process from manipulating others. To enforce isolation, Xv6 assigns each process its own address space using page tables. This address space gives the program the illusion of having its own machine. The process's address space ensures that it has private memory that other processes cannot access or modify. Each address space contains a user part which contains user text and data, user stack, and heap. The kernel part contains kernel data and text.
4. The *fork* function creates a new child process. It copies the file table and memory contents to the new process, so the code executed in the child process is the same as the parent. These two processes run simultaneously and independently. The *exec* function creates a new process that only copies the file table. When called, *exec* terminates the current process. The advantage of not combining these two functions is that the child process in *fork* can manipulate and change data, and assign and remove new file descriptors without affecting the parent ones. Also, handling errors in processes created by *fork* is relatively simple.

Adding some features to xv6 console

1. Boot message



```
user@Ubuntu: ~/Documents/test/OSLab-P1
QEMU
Machine View
SeaBIOS (version 1.15.0-1)
dd i
1000
1000
5120
dd i
1+0
1+0
512
dd i
413+Booting from Hard Disk...
413+cpu0: starting 0
2114sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
qemu
raw init: starting sh
Group #5:
xv6.1. Reza Abdoli
cpu02. Mohammad Reza Alavi
sb: 3. Farbod Azimmohseni
t 58$ _
init
sh
1. R
2. M
3. F
$
```

Figure 1: boot message

1. Ctrl + B

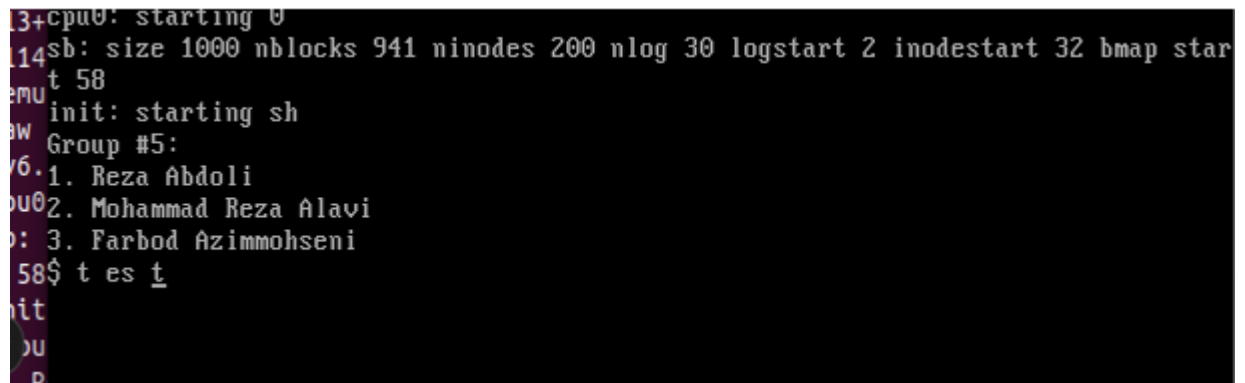


A terminal window showing boot logs. The text is as follows:
413+Booting from Hard Disk...
413+cpu0: starting 0
2114sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
qemu init: starting sh
raw Group #5:
xv6.1. Reza Abdoli
cpu02. Mohammad Reza Alavi
sb: 3. Farbod Azimmohseni
t 58\$ test
init
14sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
mu init: starting sh
w Group #5:
6.1. Reza Abdoli
u02. Mohammad Reza Alavi
: 3. Farbod Azimmohseni
58\$ t est
it
ou

In this screenshot, the characters '413+', '2114', 'qemu', 'raw', 'xv6.', 'cpu0', 'sb:', 't 58', and 'init' are highlighted in red, indicating they were typed after pressing Ctrl+B.

Figure 2: (Ctrl+B)*3 + spacebar

2. Ctrl + F



A terminal window showing boot logs. The text is as follows:
13+cpu0: starting 0
14sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
emu init: starting sh
aw Group #5:
v6.1. Reza Abdoli
ou02. Mohammad Reza Alavi
o: 3. Farbod Azimmohseni
58\$ t es t
hit
ou
P

In this screenshot, the characters '13+', '14', 'emu', 'aw', 'v6.', 'ou0', 'o:', and '58\$' are highlighted in red, indicating they were typed after pressing Ctrl+F.

Figure 3: (Ctrl+F)*2 + spacebar

3. Ctrl + L



Figure 4: Ctrl+L

4. Arrow up

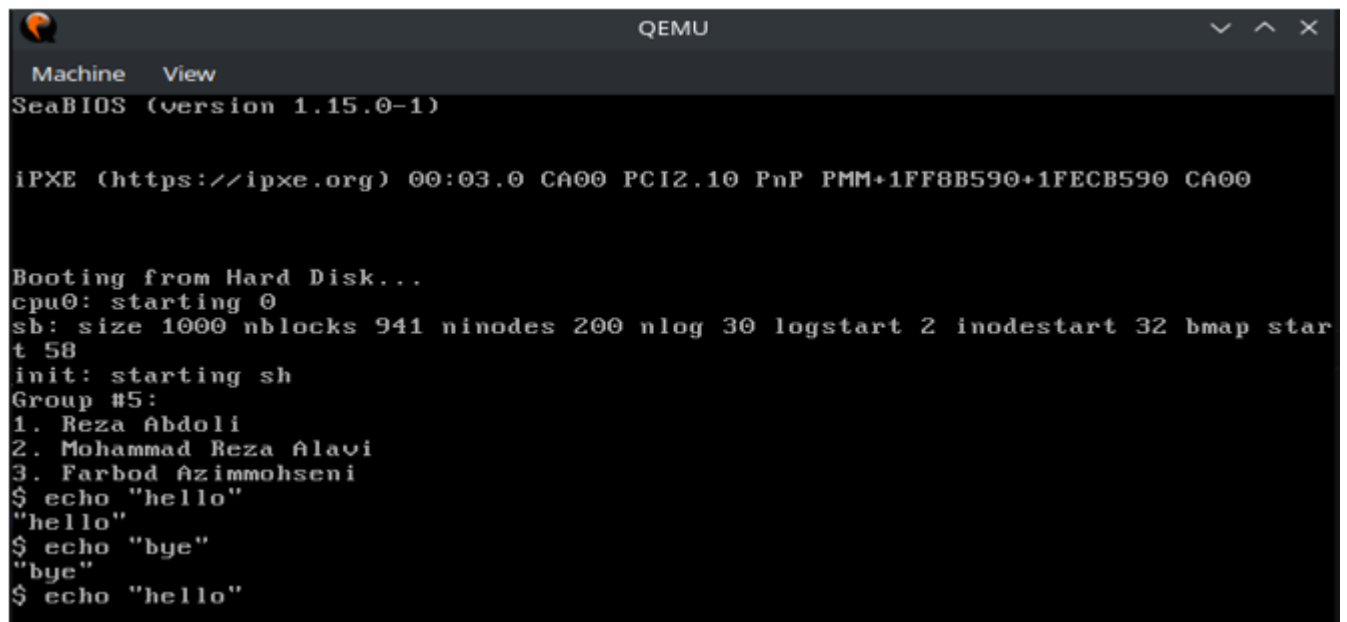
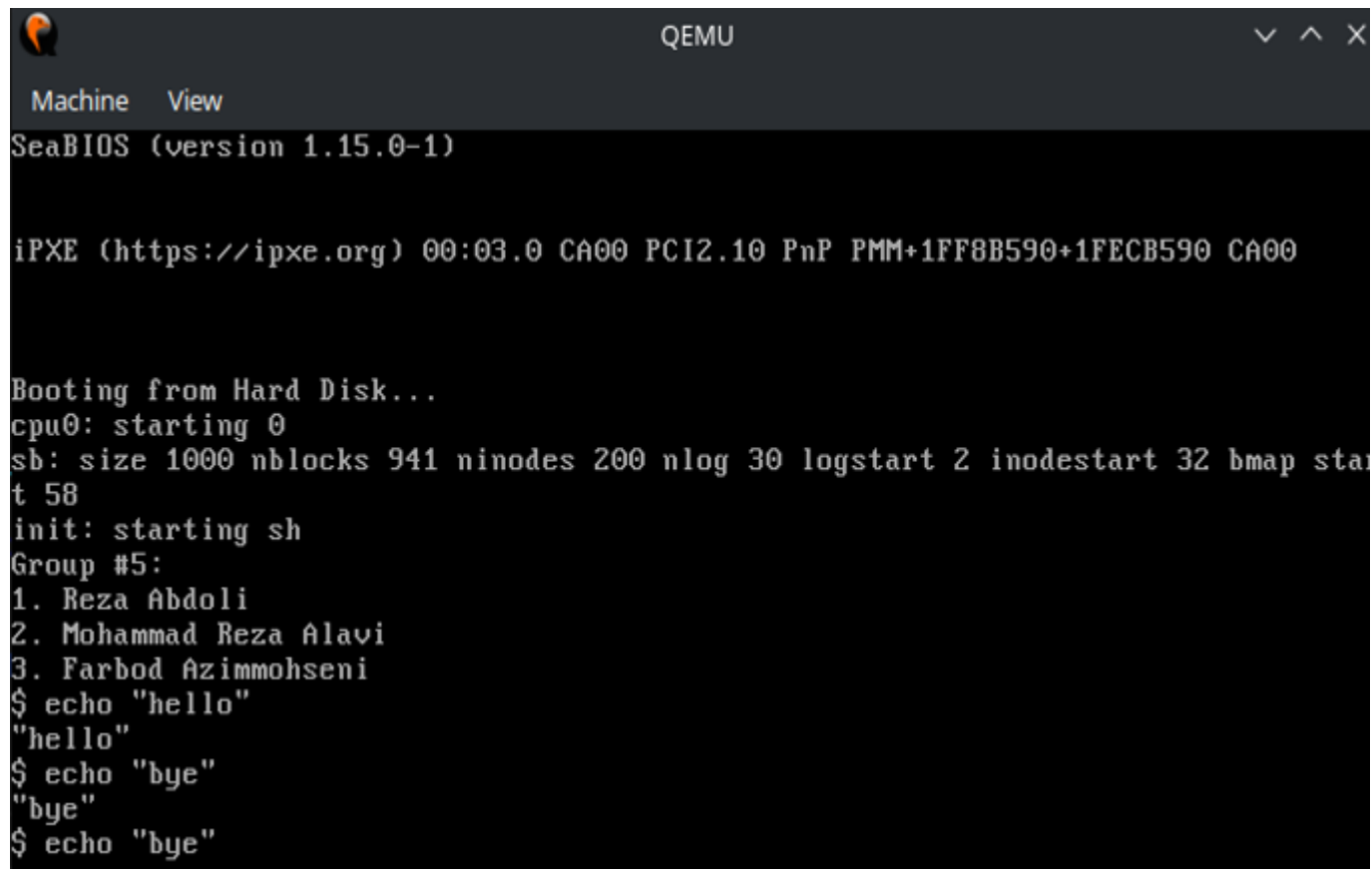


Figure 5: (Arrow up)*2

5. Arrow down



```
QEMU
Machine  View
SeaBIOS (version 1.15.0-1)

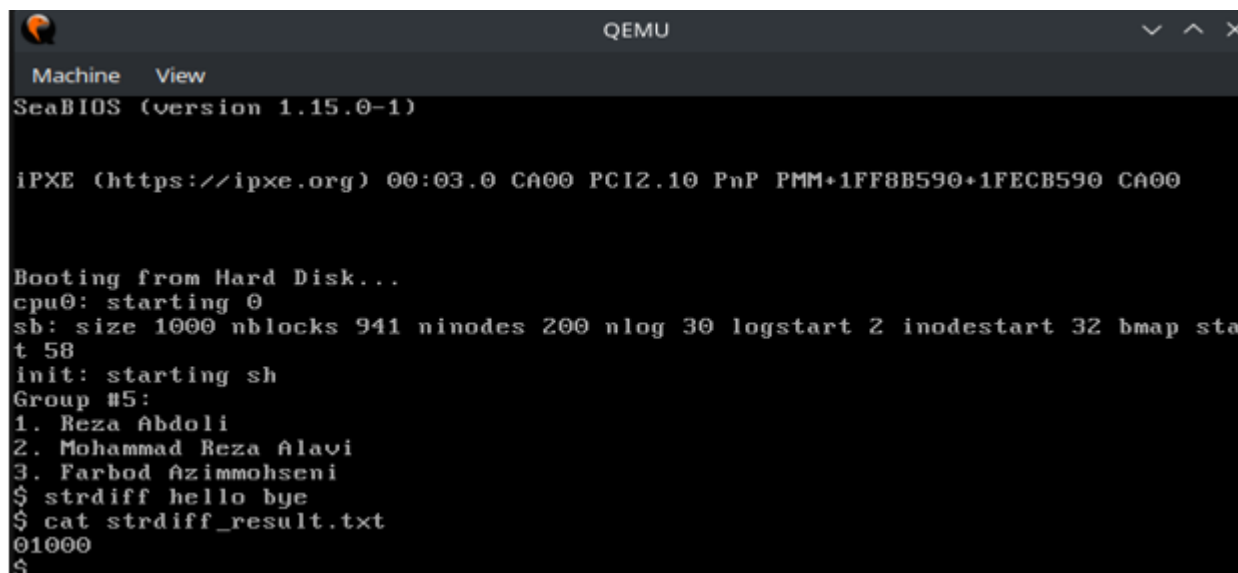
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta
t 58
init: starting sh
Group #5:
1. Reza Abdoli
2. Mohammad Reza Alavi
3. Farbod Azimmohseni
$ echo "hello"
"hello"
$ echo "bye"
"bye"
$ echo "bye"
```

Figure 6: (Arrow up)*2 + Arrow down

Running and implementing a user program

strdiff is implemented as a *C* file.



```
QEMU
Machine  View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group #5:
1. Reza Abdoli
2. Mohammad Reza Alavi
3. Farbod Azimmohseni
$ strdiff hello bye
$ cat strdiff_result.txt
01000
$
```

Figure 7: adding strdiff command to the console

compiling xv6

8. *UPROGS* refers to useful tools and user programs that are used for system calls such as `ls`. *ULIB* refers to a user-level C library that provides a set of functions and methods that user programs can use to interact with the operating system kernel. This library contains functions for making system calls. In other words, this library provides an interface for the user program to perform system calls such as `write()`, `open()`, and `pipe()`.

11. The binary file *bootblock.o* contains the initial instructions that boot the operating system. The main difference between this file and other *.o* files is how they handle memory addresses. In the case of *bootblock.o*, the instructions use physical memory addresses directly. These addresses are fixed and specific to the hardware configuration. The boot block code is designed to work with these physical addresses because the operating system has not yet initialized the memory management system or set up virtual memory. On the other hand, other *.o* files generated from *C* code and compiled to assembly and binary use virtual memory addresses. These addresses are managed by the operating system's memory management system, which includes techniques like page tables. The code of *bootblock.o* is included in the project folder as *objdump.txt*.

12. The *objcopy* function changes a compiled file of the kernel to bootable binary which is necessary for the bootloader to be able to boot the kernel. *objcopy* is commonly used to copy or transfer files from one format to another.

14. **EAX** is a general-purpose register used in xv6 to save function return values and pass arguments to functions and store intermediate results. **CS**(code segment) register is used by the processor to determine the memory location from which the next instruction should be fetched during the execution of code. It is critical in managing the flow of execution and ensuring that the CPU fetches instructions from the correct memory segment. The **Overflow flag** is one of the status flags in xv6 that shows the status of arithmetic operations like addition and multiplication. Additionally, xv6 also uses the **Sign flag** and **Zero flag** as other status flags. Control registers play the role of controlling operations in xv6. For example **CR4** is a register used to control operations such as machine-check exceptions. One of its flags, **PSE**(page size extension) controls size of the memory page.

18. Assembly file **header.S** in Linux does the same thing as **entry.s**.
19. If it was not like that we needed another page table to keep the physical address of the other page table which is not the optimal way and makes it really slow and inefficient.
22. **SEGUSER** flag is set for a segment to keep kernel data and segments from user processes. Because a user program may try to access segments that don't have access.
- 23.

```

struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    volatile int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile [NOFILE];
    struct inode *cwd;           // Current directory
    struct shared *shared;
    char name[16];
};

```

sz saves the size of process memory in bytes. **pgdir** saves the pointer to the page table that translates from virtual memory to physical memory. **kstack** keeps the bottom of kernel stack of this process. **procstate** keeps the state of process (running, sleeping,...). **pid** keeps the process id. **ofile** is a pointer to the open files of the process. The **task-struct** structure does the same thing in Linux and keeps data like process id, parent process id, etc.

27. File system are shared between all cores so that all of them can modify and use it. It is the same for device drivers and also for memory management operations. Process scheduling is a core-specific operation and each core has its own scheduler.

Debugging

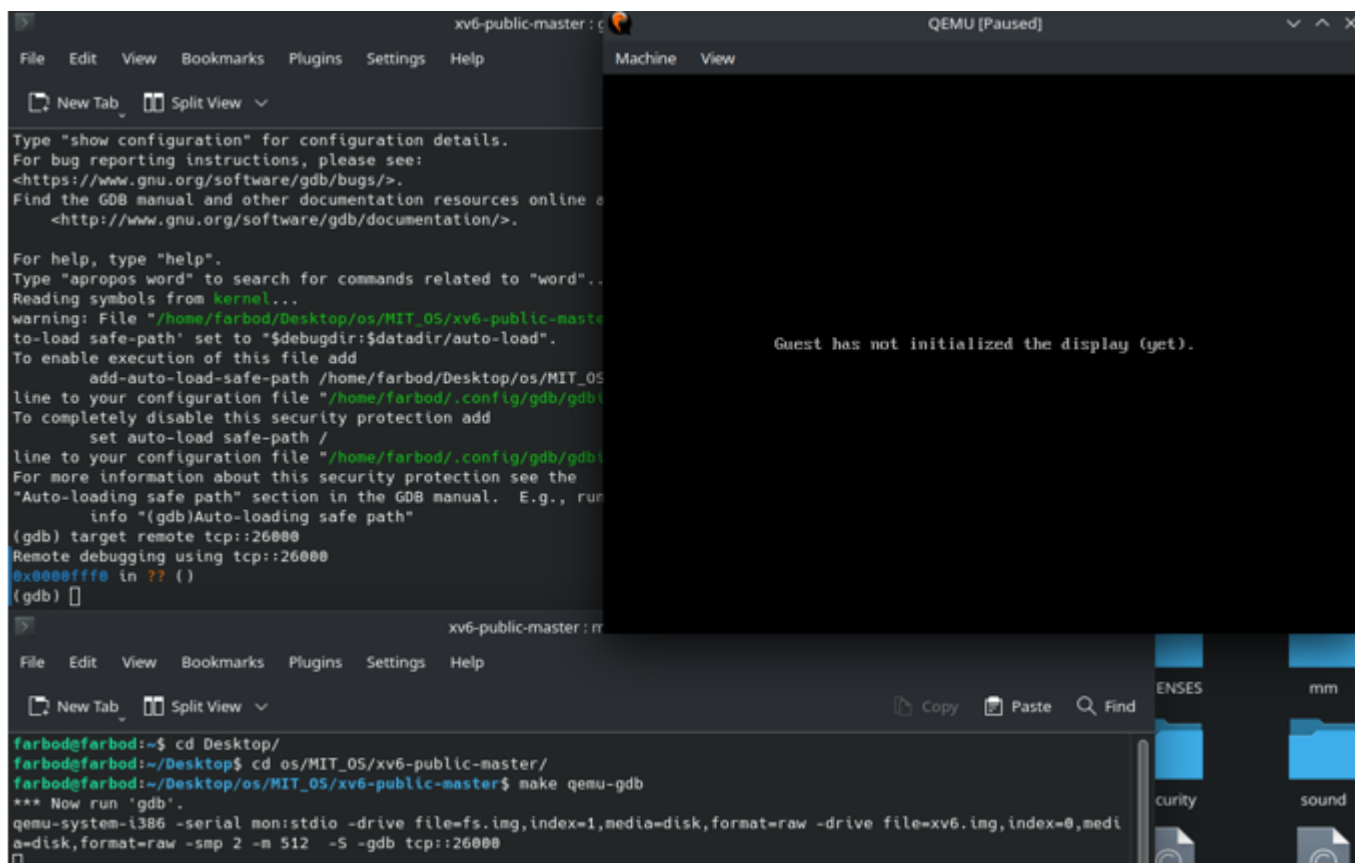


Figure 8: connecting QEMU to GDB

1. using **info break** command.
2. using **delete** followed by breakpoint number.
3. **bt** command shows the backtrace of how our program reached the current step from the first caller up to the top of the stack.

```
(gdb) bt
#0  scheduler () at proc.c:336
#1  0x8010377f in mpmain () at main.c:57
#2  0x801038cc in main () at main.c:37
(gdb)
```

Figure 9: addresses of function calls in stack

4. Command **x** shows the value saved in the address of memory and receives an address as input. Command **print** shows the value of a variable and gets the name of it as input. using **register** followed by the name of register you can see its value.
5. Using **info registers** command you can see the values of each register. For local variables, **info locals** is used. **EDI** and **ESI** registers are two general purpose registers. **EDI** is used to save the destination index in-memory operations. **ESI** is used to keep the source index of a memory operation. They can both be used to pass arguments to functions.


```

(gdb) b console.c:360
Breakpoint 1 at 0x80101008: file console.c, line 360.
(gdb) b console.c:370
Breakpoint 2 at 0x80101178: file console.c, line 370.
(gdb) delete 2
(gdb) break info
Function "info" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) info break
Num      Type           Disp Enb Address      What
1        breakpoint    keep y   0x80101008 in consoleintr at console.c:360
(gdb)

```

Figure 10: break, delete

```

(gdb) info registers
eax      0x0
ecx      0x0
edx      0x663      1635
ebx      0x0
esp      0x0      0x0 <console.read>
ebp      0x0      0x0 <console.read>
esi      0x0
edi      0x0
eip      0xffff      0xffff
eflags   0x2      [ IOPL=0 ]
cs       0xf00      61440
ss       0x0
ds       0x0
es       0x0
fs       0x0
gs       0x0
fs_base  0x0
gs_base  0x0
k_gs_base 0x0
cr0      0x60000010 [ CD NW ET ]
cr2      0x0
cr3      0x0      [ PDBR=0 PCID=0 ]
cr4      0x0      [ ]
cr8      0x0
efer     0x0      [ ]
xmm0     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm2     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm3     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
--Type <RET> for more, q to quit, c to continue without paging--RET
xmm4     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm5     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm6     {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32
x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm7     {v4_float = {0x0, 0x0, 0x0, 0x1f80}, v2_double = {0x0, 0x1f8000000000}, v16_int8 = {0x0 <repeats 12 times>, 0x80, 0x1f, 0x0, 0x0}, v8_int16 = {0x0, 0x0, 0x0,
f80, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x1f80}, v2_int64 = {0x0, 0x1f8000000000}, uint128 = 0x1f80000000000000000000000000000000}
rax      0x1f80      [ IM DM ZM OM UM PM ]
(gdb)

```

Figure 11: information of all registers

- Input structure contains three variables. **input.buf** is a cyclic buffer which has integers **input.e**, **input.w** and **input.r** pointing to different indexes of it. In the **consoleintr** function when a character is entered in the console, it is written on the **input.e** index of **input.buf**. Integer **input.w** is the index of the first character of the line the user is entering his command and when a new line character is entered, **input.w = input.e**. Each time the command that wants to execute reads from **input.r** to **input.w** in the buffer and executes it. and sets **input.r** to the index of new line.

```

(gdb) watch input.e
Hardware watchpoint 1: input.e
(gdb) continue
Continuing.

Thread 1 hit Hardware watchpoint 1: input.e

Old value = <unreadable>
New value = 1
getinput (c=101) at console.c:283
283         input.buf[input.l++ % INPUT_BUF] = c;
(gdb) print input.e
$1 = 1
(gdb)

Old value = 2
New value = 3
getinput (c=<optimized out>) at console.c:275
275         input.w = input.e;
(gdb)

```

Figure 12: changes of input.e when hitting breakpoints

7. The output of **layout asm** is the assembly code of the C file we are debugging. Command **layout src** changes back the layout of code to its source format.
8. Using **Up** and **Down** commands we can move between callers and called function calls.

1 Optional

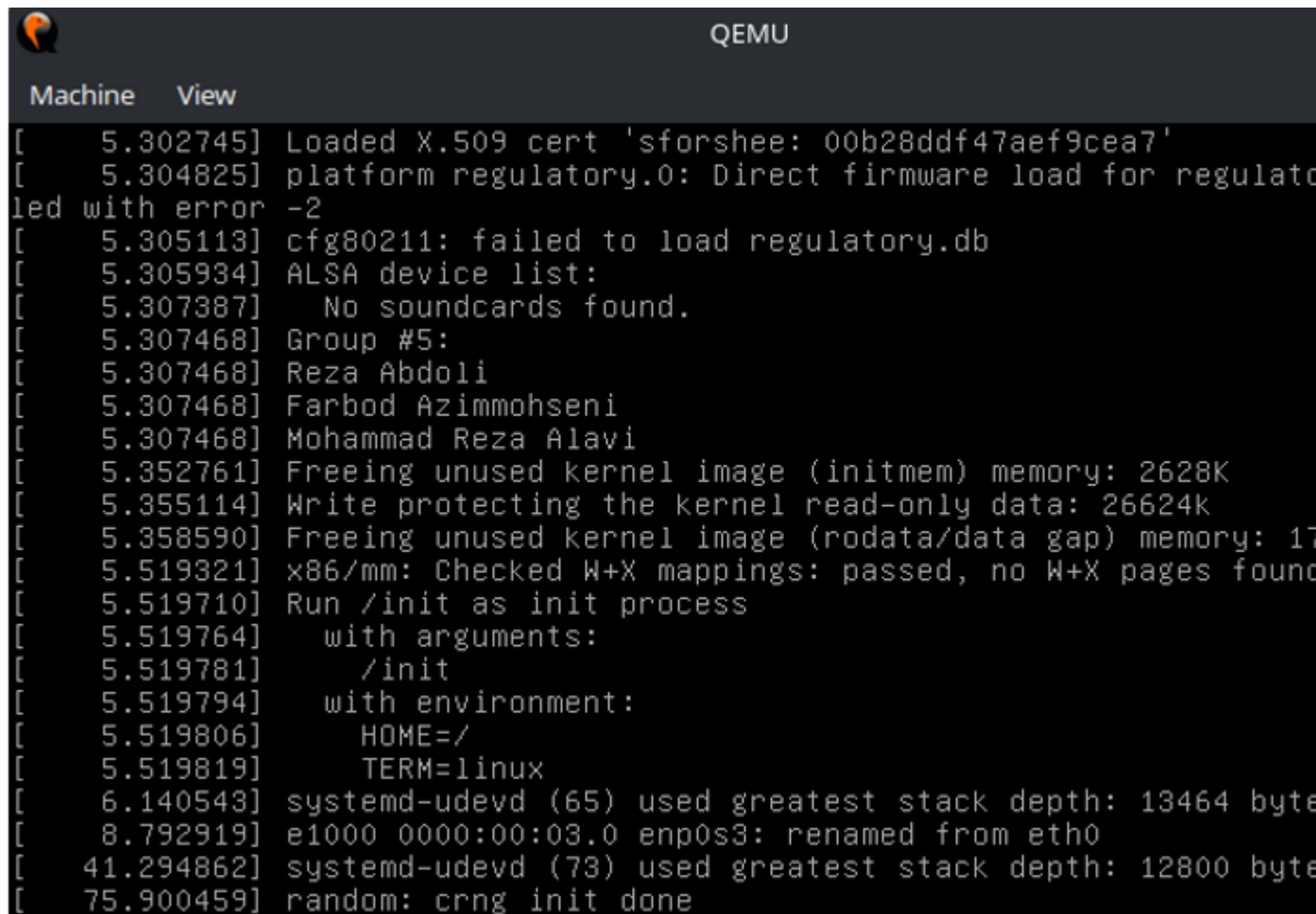
After building the Linux kernel, we boot it on the Qemu. using these commands. First go to directory `arch/x86/boot` and run the this command: **dmesg** command shows the boot message.

```
x86/boot$ mkinitramfs -o initrd.img-6.5.7
```

Figure 13: making ram image

```
ch/x86/boot$ qemu-system-x86_64 -kernel bzImage -initrd initrd.img-6.5.7 -m 1024
```

Figure 14: command that boots console

A screenshot of a QEMU terminal window. The window has a dark gray title bar with the QEMU logo on the left and the text "QEMU" on the right. Below the title bar is a menu bar with "Machine" and "View" options. The main area of the window displays the output of the 'dmesg' command, showing various system boot logs. The text is white on a black background. The logs include timestamps in brackets followed by messages such as "Loaded X.509 cert", "platform regulatory.0: Direct firmware load", "ALSA device list", "Group #5", "Freeing unused kernel image", "x86/mm: Checked W+X mappings", "Run /init as init process", "systemd-udevd", and "random: crng init done".

```
[ 5.302745] Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[ 5.304825] platform regulatory.0: Direct firmware load for regulatory
led with error -2
[ 5.305113] cfg80211: failed to load regulatory.db
[ 5.305934] ALSA device list:
[ 5.307387]   No soundcards found.
[ 5.307468] Group #5:
[ 5.307468] Reza Abdoli
[ 5.307468] Farbod Azimmohseni
[ 5.307468] Mohammad Reza Alavi
[ 5.352761] Freeing unused kernel image (initmem) memory: 2628K
[ 5.355114] Write protecting the kernel read-only data: 26624k
[ 5.358590] Freeing unused kernel image (rodata/data gap) memory: 17
[ 5.519321] x86/mm: Checked W+X mappings: passed, no W+X pages found
[ 5.519710] Run /init as init process
[ 5.519764]   with arguments:
[ 5.519781]   /init
[ 5.519794]   with environment:
[ 5.519806]   HOME=/
[ 5.519819]   TERM=linux
[ 6.140543] systemd-udevd (65) used greatest stack depth: 13464 byte
[ 8.792919] e1000 0000:00:03:0 enp0s3: renamed from eth0
[ 41.294862] systemd-udevd (73) used greatest stack depth: 12800 byte
[ 75.900459] random: crng init done
```

Figure 15: dmesg output