Operating System Capstone

Lecture 2: OS Introduction

Tsung Tai Yeh

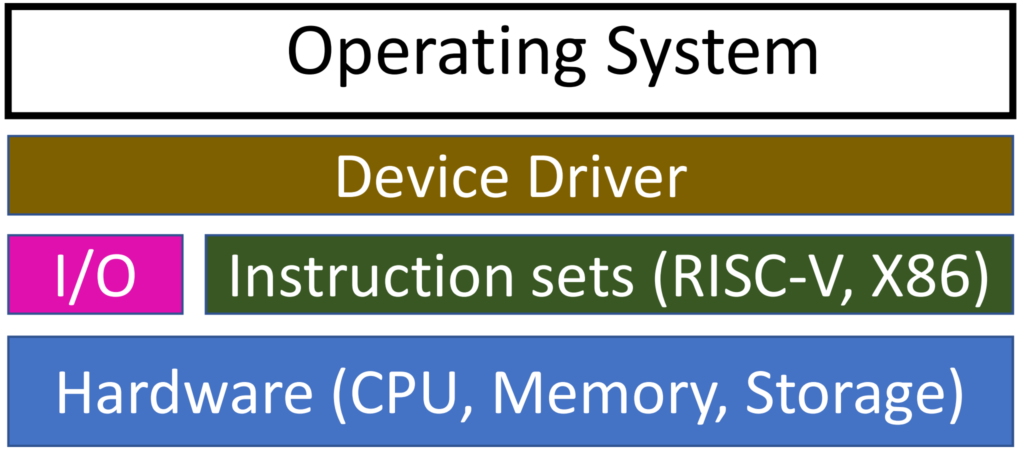
Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Christopher Hallinan, Embedded Linux Primer, A Practical Real-World Approach, Prentice Hall, 2010

What is the purpose of an OS

- **Abstract the hardware**
  - Convenience and portability
- **Multiplex the hardware**
  - Share the hardware among multiple applications
- **Isolate applications**
  - Security and privacy issue
- **Provide high performance**
  - Hardware resource management



Operating System

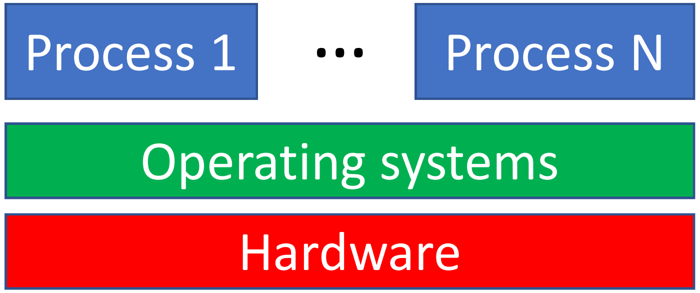
Device Driver

I/O

Instruction sets (RISC-V, X86)

Hardware (CPU, Memory, Storage)

What is the OS design approach ?

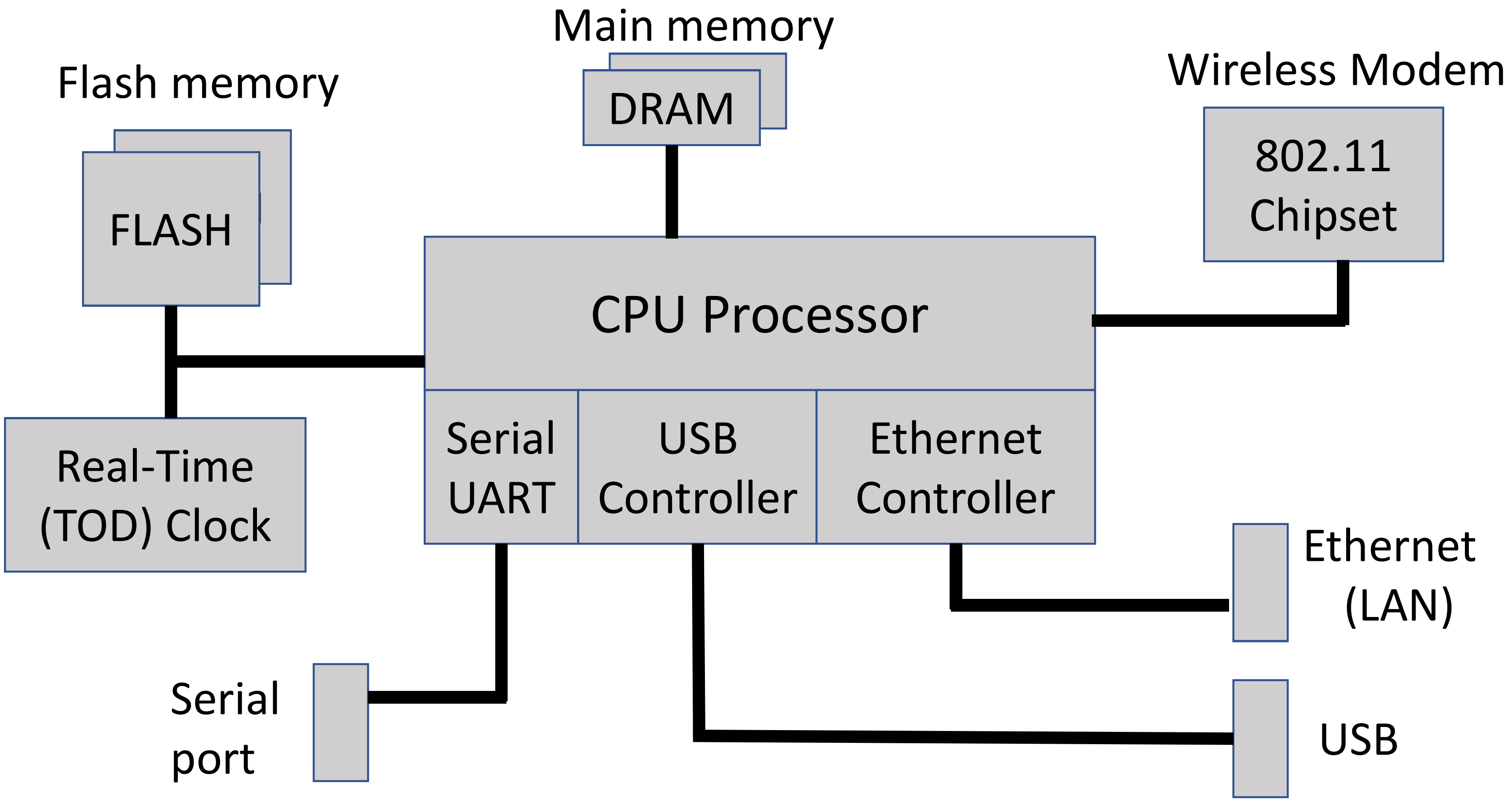


- **The small view**
  - A hardware management library
  - Problem of this method ?
- **The big view**
  - **Abstraction**
    - Hide details of underlying hardware
    - E.g. processes open and access files instead of issuing raw commands to hard drive
  - **Resource management**
    - Controls how processes share hardware resources (CPU, memory, disk, etc.)

Outline

- Operating system basics
  - MIT JOS kernel
  - Bootstrap
  - Basic Input Output System (BIOS)
  - Boot loader
  - Unix Shell

Anatomy of an Embedded System



Main memory

DRAM

Flash memory

FLASH

Wireless Modem

802.11 Chipset

CPU Processor

Real-Time (TOD) Clock

Serial UART

USB Controller

Ethernet Controller

Ethernet (LAN)

Serial port

USB

A Small OS kernel -JOS

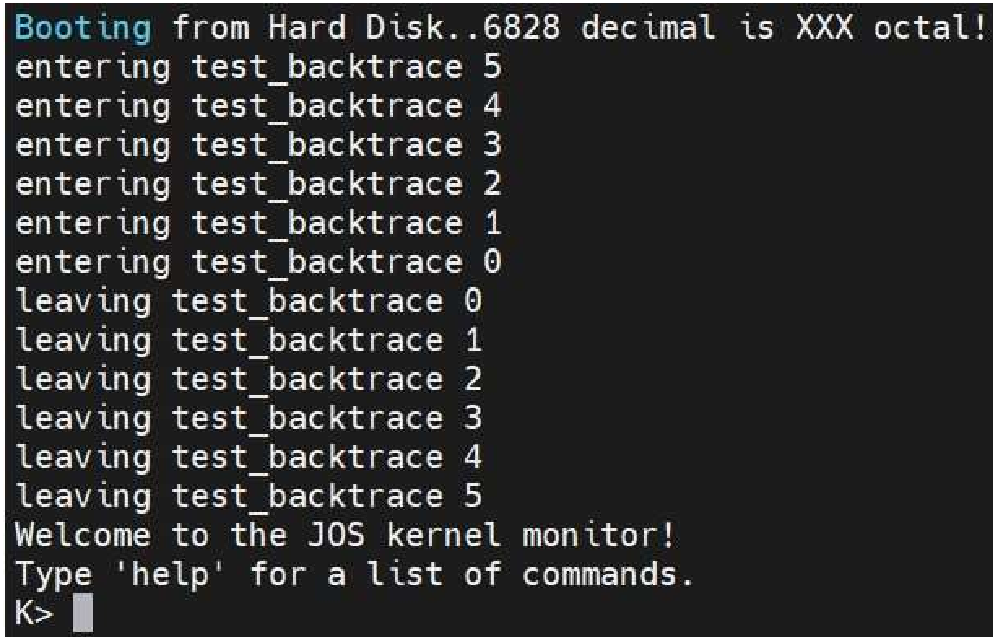
- JOS was from MIT that includes the skeleton of OS and helps you get through booting procedure
  - Run JOS on QEMU emulator
  - Include a boot loader
  - Loading the kernel
  - Kernel message can be prompt printed by the small monitor (console)
  - QEMU wiki: https://wiki.qemu.org/Hosts/Linux
  - Demo JOS

```
git clone https://pdos.csail.mit.edu/6.828/2018/jos.git
```

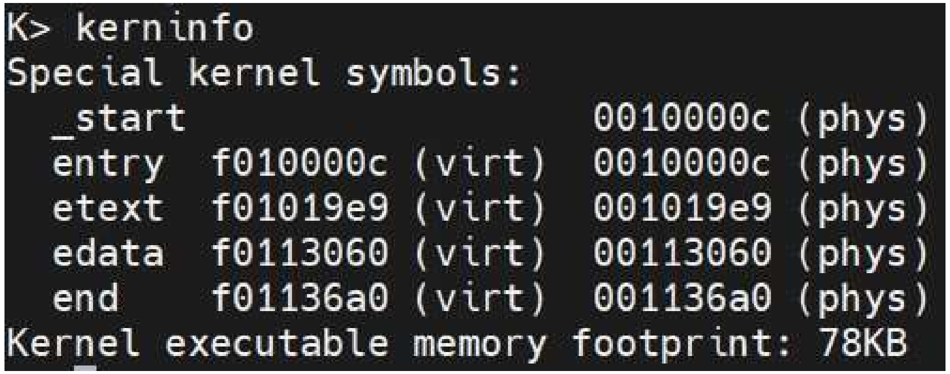
```
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
```

The bootstrap

- The contents of emulated PC’s virtual hard disk
  - Supplying the file obj/kern/kernel.img as
- The hard disk image contains
  - the bootloader (obj/boot/boot)
  - the kernel (obj/kernel)



```
Booting from Hard Disk..6828 decimal is XXX octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```



```
K> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f01019e9 (virt)  001019e9 (phys)
  edata  f0113060 (virt)  00113060 (phys)
  end    f01136a0 (virt)  001136a0 (phys)
Kernel executable memory footprint: 78KB
```

Basic Input/Output System (BIOS)

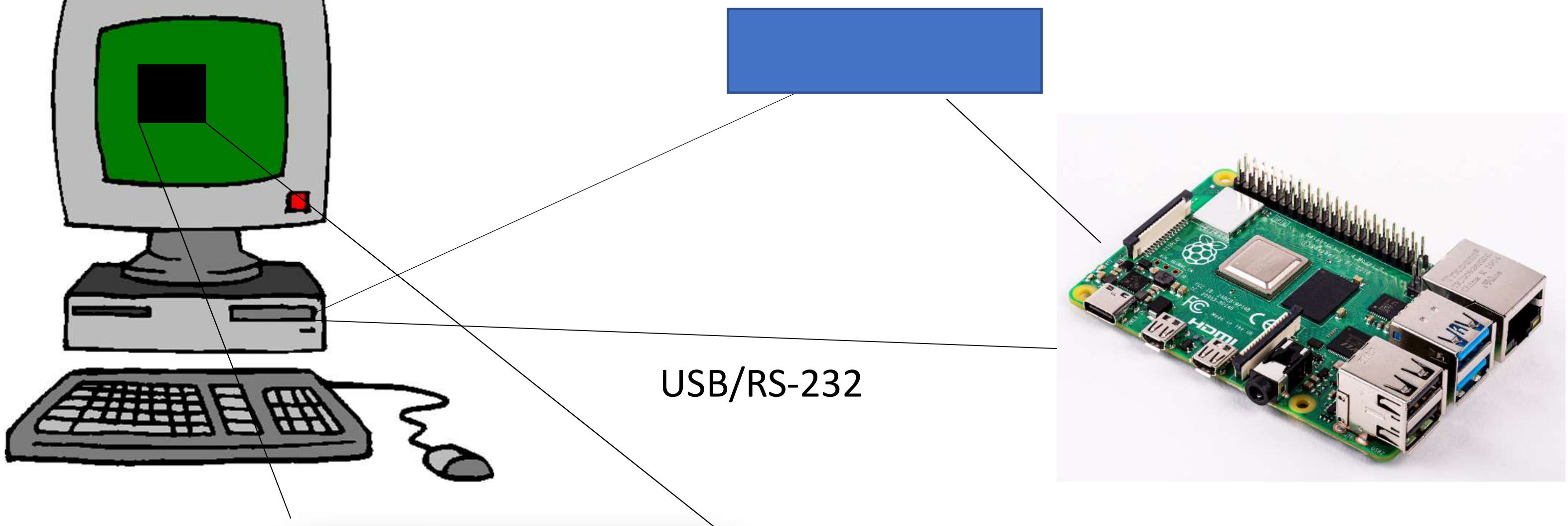
- **Basic input/output system (BIOS)**
  - A set of system-configuration software routines
  - Know the low-level details of hardware architecture
  - When power is first applied to the computer, BIOS immediately takes control of the processor
  - Stored in Flash memory
- **Primary responsibility**
  - Initialize the hardware
  - checking the amount of memory installed
  - Load an operating system from the storage device

The bootloader

What is difference between bootloader and BIOS ?

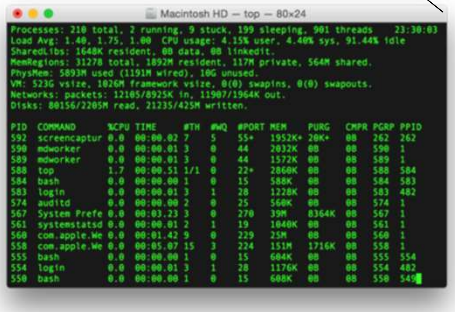
- The bootloader
  - The software program that performs same functions as BIOS
- The bootloader’s jobs on power-up
  - **Initializes hardware components** such as memory, I/O, graphics controllers
  - **Initializes system memory**
    - In preparation for passing control to the operating system
  - **Allocates system resources**
    - Memory and interrupt circuits to peripheral controllers
  - **Loading the operating system image**
  - **Passing any required startup information**
    - Total memory size, clock rates, serial port speeds and other low-level hardware specific configuration data
- Check the bootloader files: boot/boot.S, boot/main.c in JOS

Typical Embedded Linux Setup



Ethernet Hub

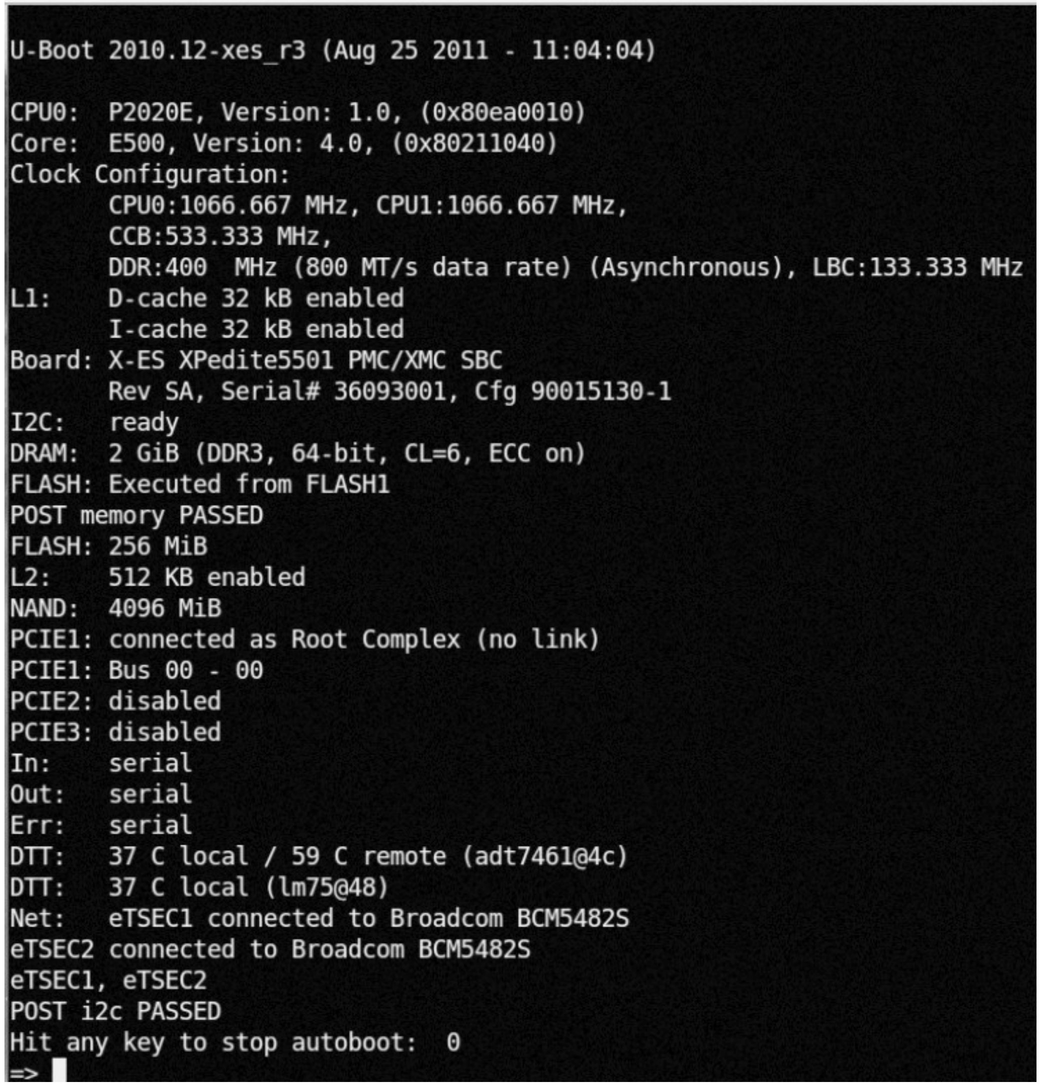
USB/RS-232



```
Processes: 210 total, 2 running, 9 stuck, 199 sleeping, 901 threads     23:30:03
Load Avg: 1.40, 1.75, 1.00  CPU usage: 4.15% user, 4.40% sys, 91.44% idle
SharedLibs: 1648K resident, 0B data, 0B linkedit.
MemRegions: 31278 total, 1892M resident, 117M private, 564M shared.
PhysMem: 5893M used (1191M wired), 10G unused.
VM: 523G vsize, 1026M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 12105/8925K in, 11907/1964K out.
Disks: 80156/2205M read, 21235/425M written.

PID  COMMAND      %CPU TIME     #TH  #WQ  #PORT MEM    PURG   CMPR PGRP PPID
592  screencaptur 0.0  00:00.02 7    5    55+   1952K+ 20K+   0B   262  262
590  mdworker     0.0  00:00.01 3    0    44    2032K  0B     0B   590  1
589  mdworker     0.0  00:00.01 3    0    44    1572K  0B     0B   589  1
588  top          1.7  00:00.51 1/1  0    22+   2860K  0B     0B   588  584
584  bash         0.0  00:00.00 1    0    15    588K   0B     0B   584  583
583  login        0.0  00:00.01 3    1    28    1228K  0B     0B   583  482
574  auditd       0.0  00:00.00 2    0    25    560K   0B     0B   574  1
567  System Prefe 0.0  00:03.23 3    0    270   39M    8364K  0B   567  1
561  systemstatsd 0.0  00:00.01 2    1    19    1040K  0B     0B   561  1
560  com.apple.We 0.0  00:01.42 9    0    229   25M    0B     0B   560  1
558  com.apple.We 0.0  00:05.07 15   3    224   151M   1716K  0B   558  1
555  bash         0.0  00:00.00 1    0    15    604K   0B     0B   555  554
554  login        0.0  00:00.01 3    1    28    1176K  0B     0B   554  482
550  bash         0.0  00:00.00 1    0    15    608K   0B     0B   550  549
```

Starting the target board

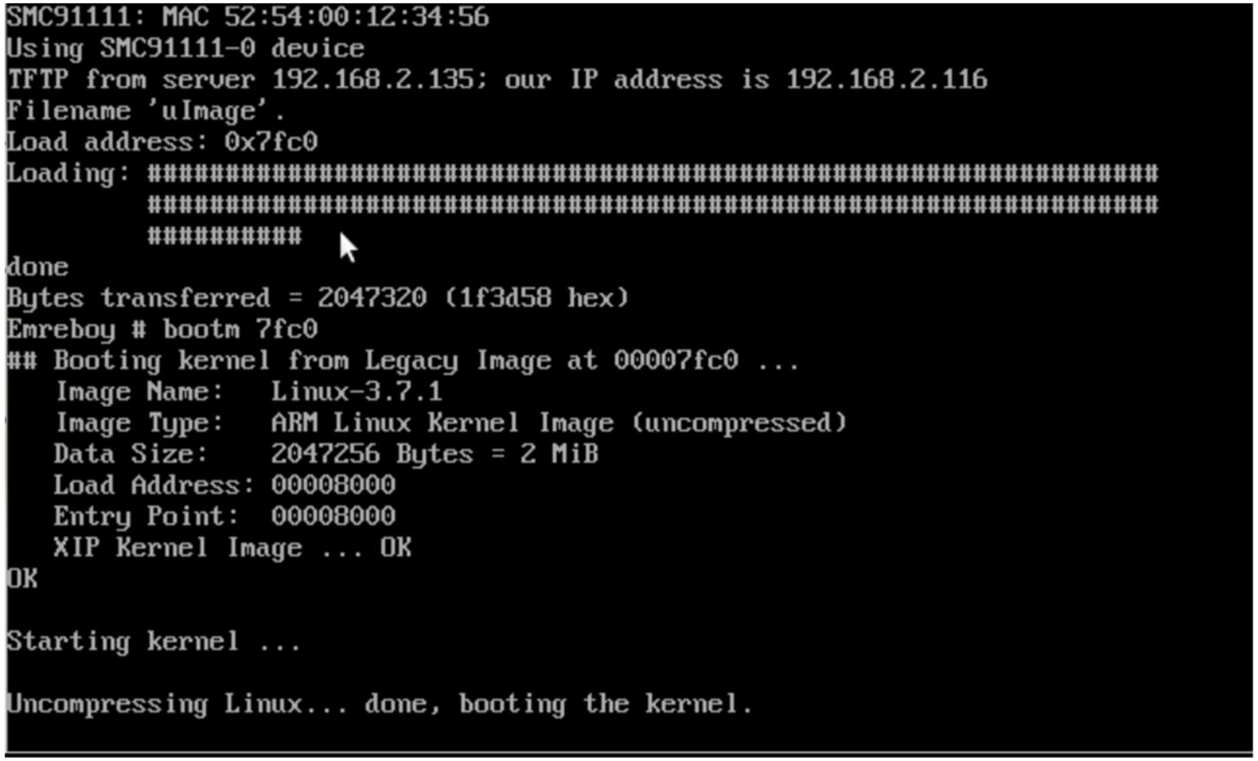
- When power is first applied
  - A bootloader in the target board takes control of the processor
  - Initializes low-level hardware
    - Processor and memory setup
    - Initializes UART and Ethernet controller
    - Configures serial ports



```
U-Boot 2010.12-xes_r3 (Aug 25 2011 - 11:04:04)

CPU0:  P2020E, Version: 1.0, (0x80ea0010)
Core:  E500, Version: 4.0, (0x80211040)
Clock Configuration:
       CPU0:1066.667 MHz, CPU1:1066.667 MHz,
       CCB:533.333 MHz,
       DDR:400  MHz (800 MT/s data rate) (Asynchronous), LBC:133.333 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Board: X-ES XPedite5501 PMC/XMC SBC
       Rev SA, Serial# 36093001, Cfg 90015130-1
I2C:   ready
DRAM:  2 GiB (DDR3, 64-bit, CL=6, ECC on)
FLASH: Executed from FLASH1
POST memory PASSED
FLASH: 256 MiB
L2:    512 KB enabled
NAND:  4096 MiB
PCIE1: connected as Root Complex (no link)
PCIE1: Bus 00 - 00
PCIE2: disabled
PCIE3: disabled
In:    serial
Out:   serial
Err:   serial
DTT:   37 C local / 59 C remote (adt7461@4c)
DTT:   37 C local (lm75@48)
Net:   eTSEC1 connected to Broadcom BCM5482S
eTSEC2 connected to Broadcom BCM5482S
eTSEC1, eTSEC2
POST i2c PASSED
Hit any key to stop autoboot:  0
=>
```

Booting the kernel

- Load kernel image
  - Uses “tftpboot” instructs U-Boot to load the kernel image into memory
  - The “bootm” (boot from memory image) command instructs U-boot to boot the kernel

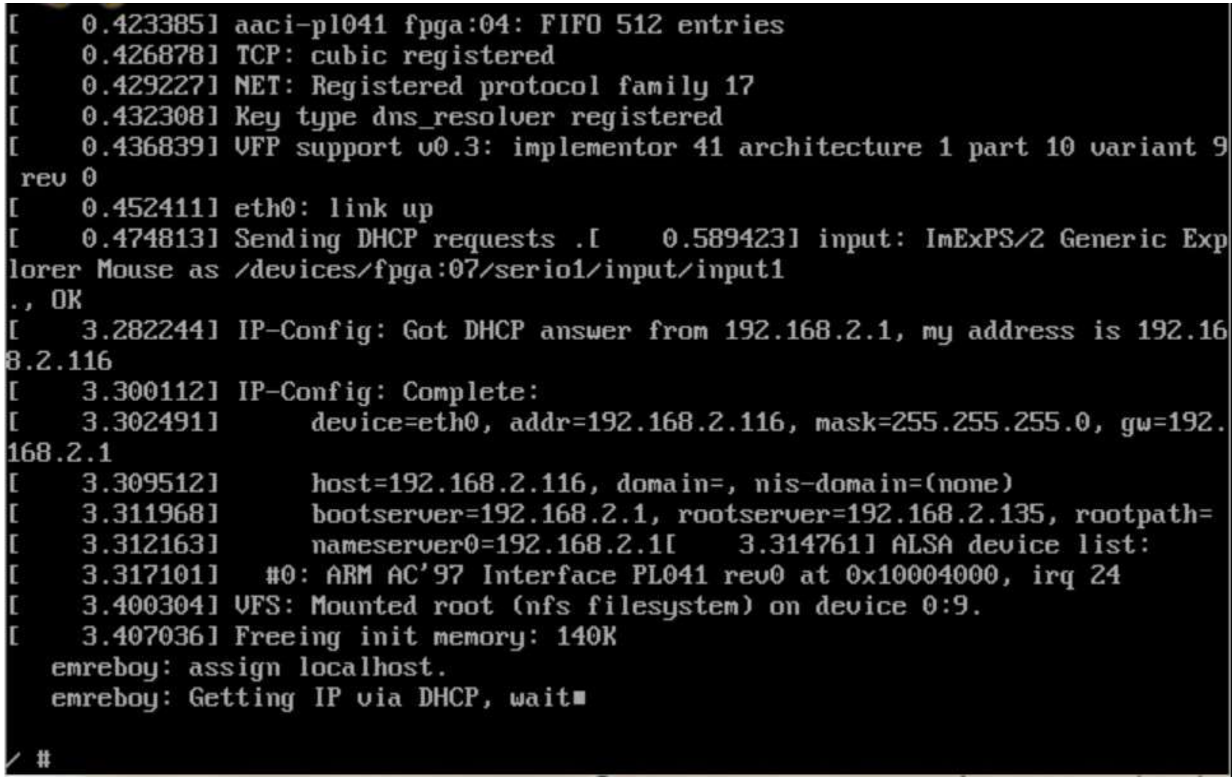


```
SMC91111: MAC 52:54:00:12:34:56
Using SMC91111-0 device
TFTP from server 192.168.2.135; our IP address is 192.168.2.116
Filename 'uImage'.
Load address: 0x7fc0
Loading: #################################################################
         #################################################################
         ##########
done
Bytes transferred = 2047320 (1f3d58 hex)
Emreboy # bootm 7fc0
## Booting kernel from Legacy Image at 00007fc0 ...
   Image Name:   Linux-3.7.1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2047256 Bytes = 2 MiB
   Load Address: 00008000
   Entry Point:  00008000
   XIP Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
```

- Unlike the BIOS in a desktop
  - The bootloader ceases to exist when the Linux kernel takes control

Booting the kernel

- Before issuing a login prompt
    - Linux mounts a root file system
    - A root file system contains
        - Application systems
        - System libraries
        - Utilities that make up a GNU/Linux system



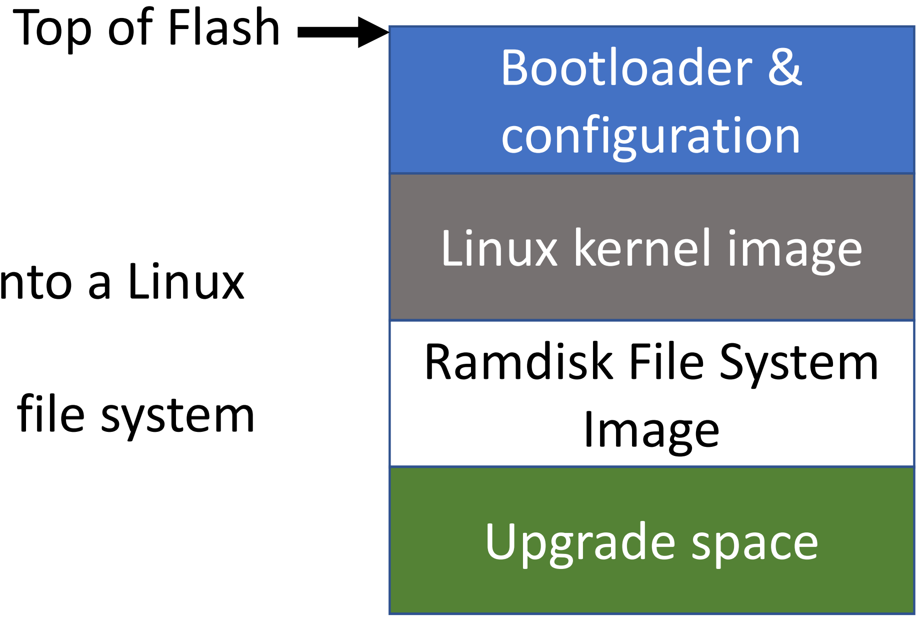
```
[    0.423385] aaci-pl041 fpga:04: FIFO 512 entries
[    0.426878] TCP: cubic registered
[    0.429227] NET: Registered protocol family 17
[    0.432308] Key type dns_resolver registered
[    0.436839] VFP support v0.3: implementor 41 architecture 1 part 10 variant 9
 rev 0
[    0.452411] eth0: link up
[    0.474813] Sending DHCP requests .[    0.589423] input: ImExPS/2 Generic Exp
lorer Mouse as /devices/fpga:07/serio1/input/input1
., OK
[    3.282244] IP-Config: Got DHCP answer from 192.168.2.1, my address is 192.16
8.2.116
[    3.300112] IP-Config: Complete:
[    3.302491]      device=eth0, addr=192.168.2.116, mask=255.255.255.0, gw=192.
168.2.1
[    3.309512]      host=192.168.2.116, domain=, nis-domain=(none)
[    3.311968]      bootserver=192.168.2.1, rootserver=192.168.2.135, rootpath=
[    3.312163]      nameserver0=192.168.2.1[    3.314761] ALSA device list:
[    3.317101]   #0: ARM AC'97 Interface PL041 rev0 at 0x10004000, irq 24
[    3.400304] VFS: Mounted root (nfs filesystem) on device 0:9.
[    3.407036] Freeing init memory: 140K
  emreboy: assign localhost.
  emreboy: Getting IP via DHCP, wait■

/ #
```

First User Space Process: init

```
INIT: version 2.78 booting
```

- After “init” stage
  - The kernel owns all system memory and operates with full authority over all system resources
- “init” application program
  - The Linux kernel spawns after completing internal initialization and mounting its root file system
  - When the kernel start “init”, it is said to be running in user space
  - Then, the user space process has restricted access to the system
  - Must use kernel system calls to request kernel services

Flash usage



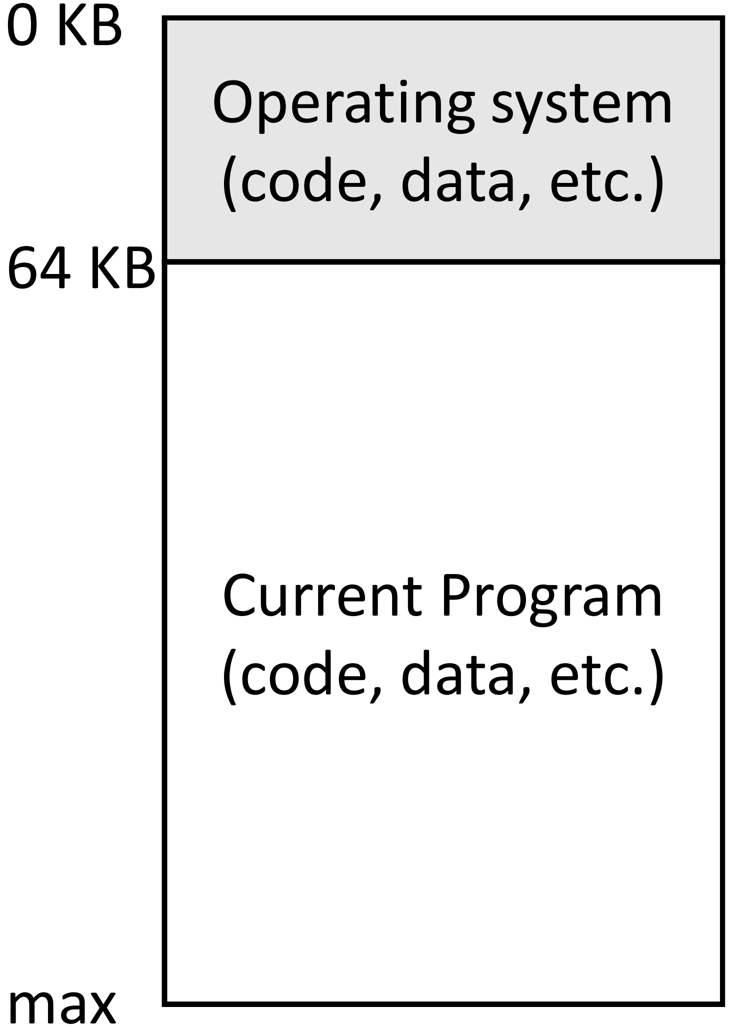
- When booted
  - A file system image stored in Flash is read into a Linux ramdisk block device
  - Linux ramdisk block device is mounted as a file system and accessed only from RAM
- Flash memory layout
  - The bootloader is often placed in the top or bottom of the Flash memory array
  - The Linux kernel and ramdisk file system images are compressed
  - The bootloader handles the decompression during the boot cycle

Memory Management Unit (MMU)

- MMU is a **hardware engine**
  - Enable an operating system to control over its address space and the address space it allocates to processes
- The purpose of MMU
  - **Access rights**
    - Allow an operating system to assign specific memory-access privileges to specific tasks
  - **Memory translation**
    - Allow an operating system to virtualize its address space

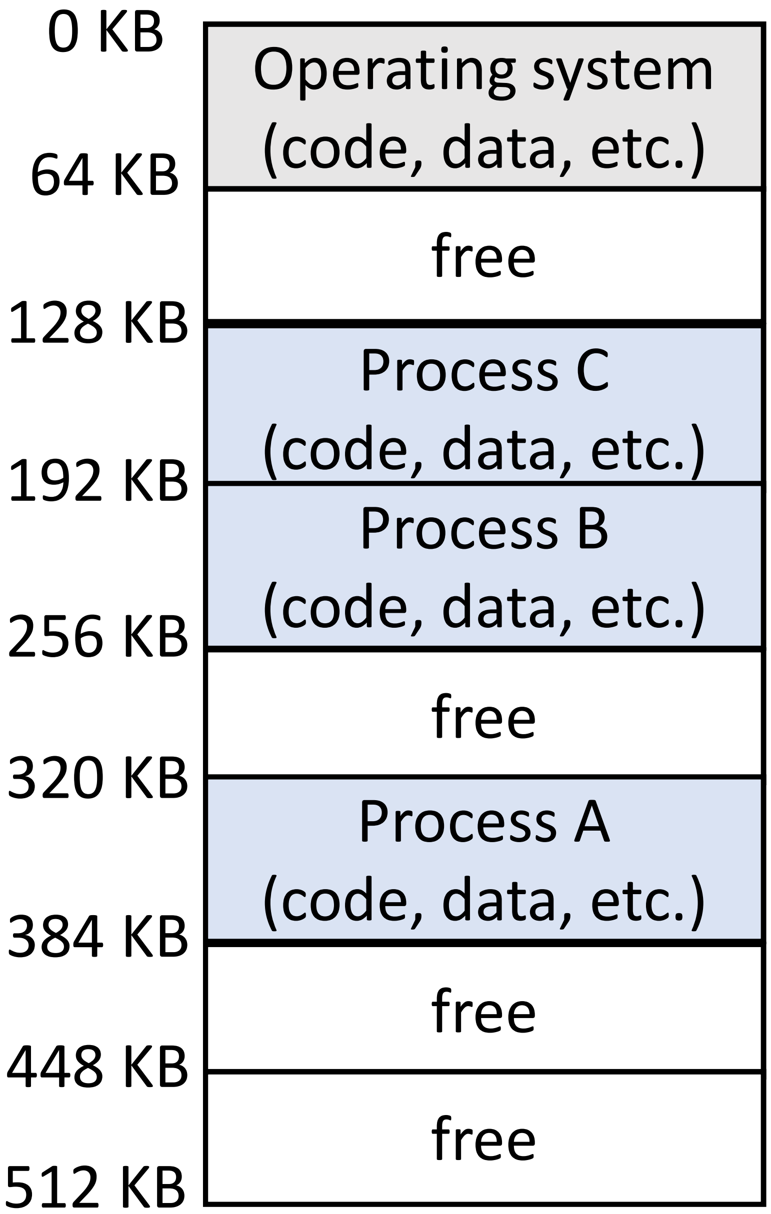
Address space: Single user machine

- Early systems
  - The OS was set of routines (a library)
  - One running program (a process)
    - Starting at physical address 64k and use the rest of memory
  - Life was sure easy
  - What are problems of this physical address space ?



Address space: multiprogramming

- Multiprogramming & Time sharing
  - Assuming a single CPU
  - The OS chooses one of the processes, while the others sit in the ready queue waiting to run
  - What are problem of this address space ?



0 KB

Operating system (code, data, etc.)

64 KB

free

128 KB

Process C (code, data, etc.)

192 KB

Process B (code, data, etc.)

256 KB

free

320 KB

Process A (code, data, etc.)

384 KB

free

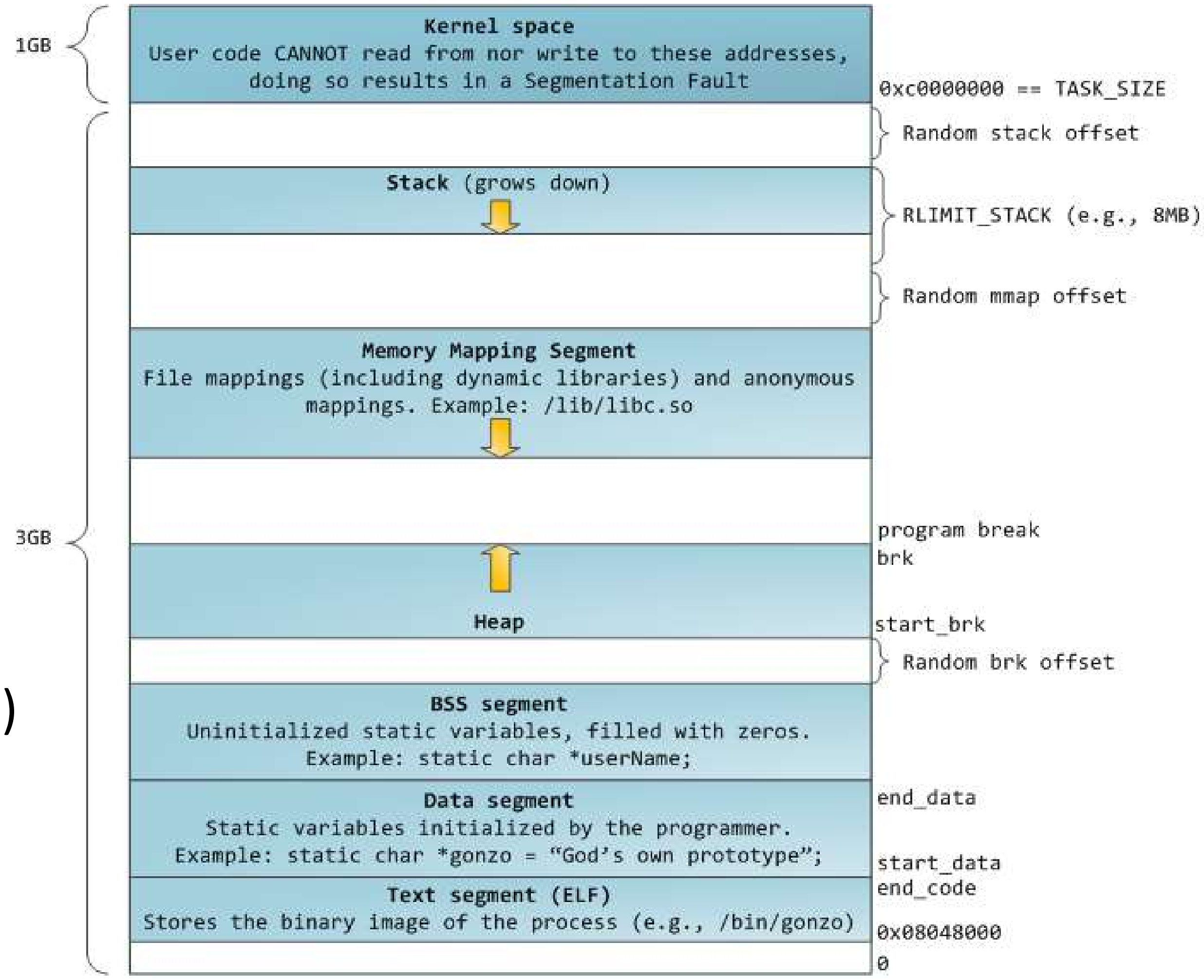
448 KB

free

512 KB

Address space

- It is the running program's view of memory in the system
- **Stack**
  - Keep track of where it is in the function call chain
  - Allocate local variables
- **Heap**
  - Used for dynamically-allocated user-managed memory, malloc()
- **Code**
  - Segments



Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

https://hackmd.io/@ofAlpaca/r1L5Ecc_7?type=view

Virtualizing memory

- **Virtual memory**
  - Illusion of a large, private, uniform store for multiple running processes on top of a single, physical memory
- **Benefit:**
  - **Efficient use of physical memory**
    - By presenting the appearance that the system has more memory than is physical present
  - **Prevent one process from errantly accessing memory**
    - The kernel can enforce access rights to each range of system memory that it allocates to a task or process

Case Study: Virtualizing memory

- Calling malloc() to allocate memory

```
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
int main(int argc, char *argv[]) {
		int *p = malloc (sizeof(int));
		assert(p != NULL);
		printf(“(%d) address pointed to by p: %p\n”, getpid(), p);
		*p = 0;
		while(1){
				Spin(1);
				*p = *p + 1;
				printf(“(%d) p: %d\n”, getpid(), *p);
		}
				return 0; }
```

```
Prompt> ./mem
```

What are outputs of *p ?

```
Address pointed to by p: 0x200000
P: 1
P: 2
P: 3
P: 4
P: 5
…
```

mem.c

Case study: Virtualizing memory

```
Prompt> ./mem &; ./mem &
```

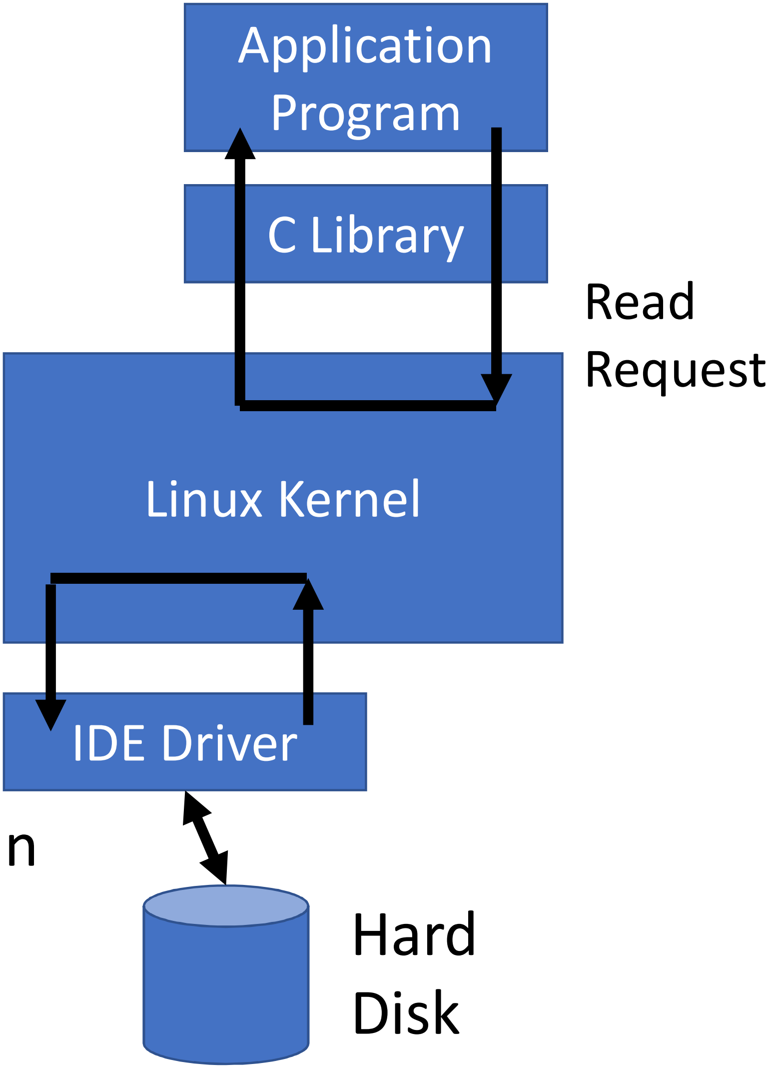
- Process identifier (PID) is unique per running process
- Each running program has allocated memory at the same address (0x200000)
- How to update the value at 0x200000 independently ?
  - Each process accesses its own private virtual address space

What are outputs of *p ?

```
[1] 24113
[2] 24114
(24113) Address pointed to by p: 0x200000
(24114) Address pointed to by p: 0x200000
(24113) P: 1
(24114) P: 1
(24113) P: 2
(24114) P: 2
(24113) P: 3
(24114) P: 3
(24113) P: 4
(24114) P: 4
…
```

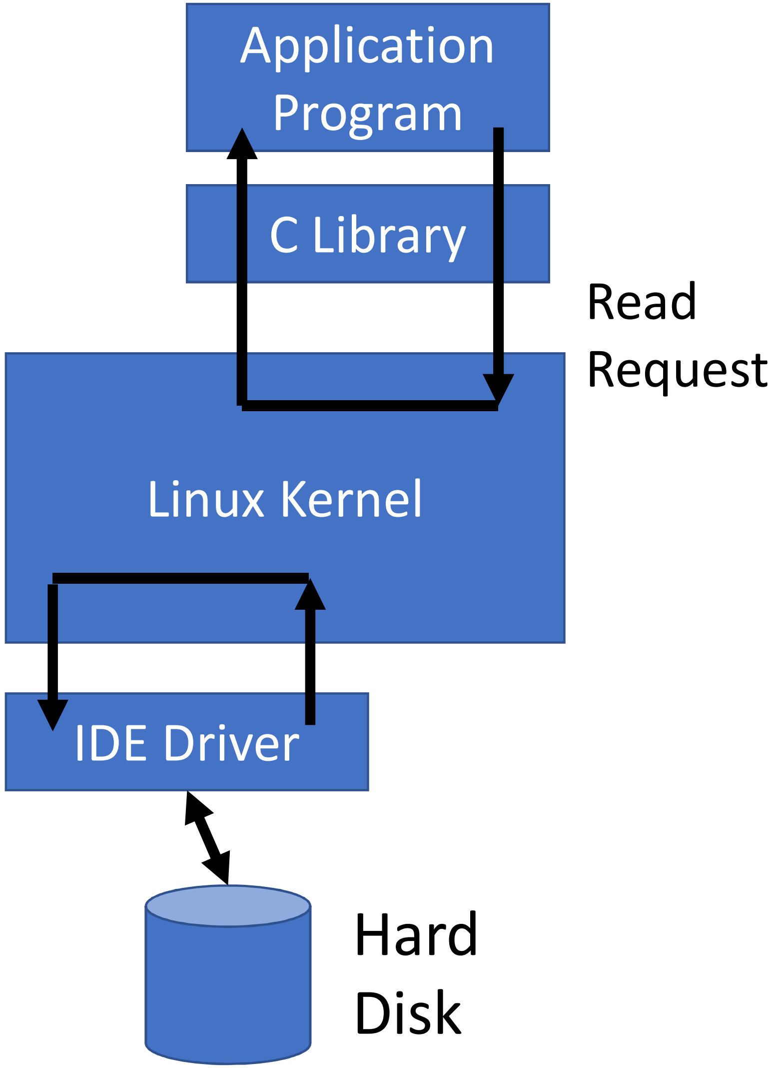
Execution Contexts

- Consider to open a file and issue a read request
  - The **read** function call begins in user space, in the C library read() function
  - The C library issues a read request **to the kernel**
  - A context switch from the user to kernel space
  - Inside the kernel, the read requests results in a hard-drive access requesting the sectors containing the file's data
  - The hard-drive read is asynchronous issued



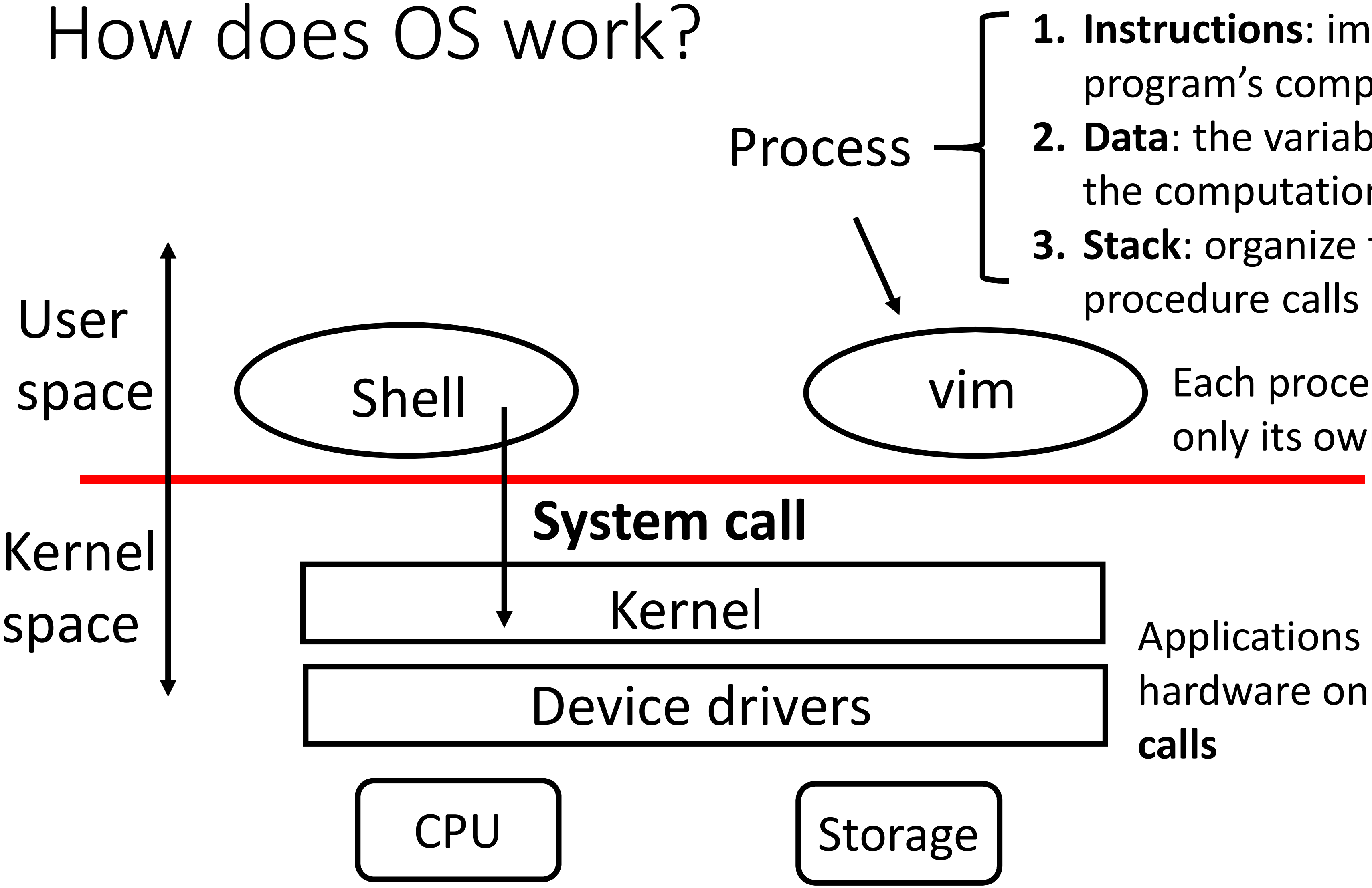
Execution Contexts cont.

- Consider to open a file and issue a read request
  - When the data is ready, **the hardware interrupts the processor**
  - When the hard disk has the data ready, **its posts a hardware interrupt**
  - When **the kernel receives the hardware interrupt**, its suspends whatever process was executing and proceeds to read the waiting data from the drive



How does OS work?

Each running program is call **process**



1. **Instructions**: implement the program's computation
2. **Data**: the variables on which the computation acts
3. **Stack**: organize the program's procedure calls

Each process can access only its own memory

Applications access beneath hardware only via **system calls**

Many system calls

System call	Description
fork()	Create a process
exit()	Terminate the current process
wait()	Wait for a child process to exit
open(filename, flag)	Open a file; the flags indicate read/write
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd’s in p
fstat(fd)	Return info about an open file
unlink(filename)	Remove a file

Case study 1: How to use system calls ?

- Applications access the hardware only through system calls
  - Spin(): A function that repeatedly checks the time and returns once it has run for a second

```
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
int main(int argc, char *argv[]) {
        if (argc != 2)
                fprint(stderr, “usage: cpu <string>\n”); exit(1);
        char *str = argv[1];
        while (1){
                Spin(1);
                printf(“%s\n”, str); }
        return 0;
 }
```

How to use system calls ?

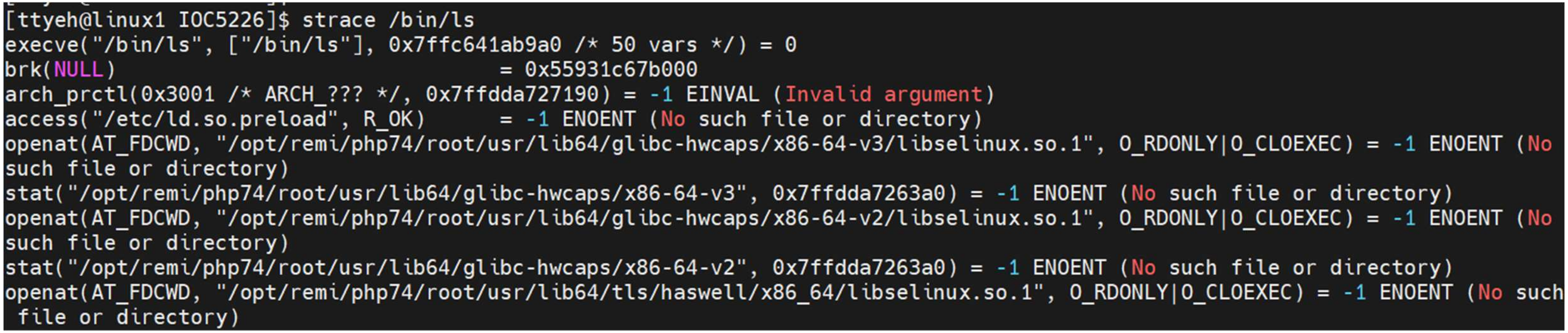
- gcc -o cpu cpu.c –Wall
- Prompt> ./cpu “A”
- What do outputs look like?
  - A
  - A
  - A ….
- How to stop this program?
  - Control-c in Unix to halt this program
- How about
  - prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &

```
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
int main(int argc, char *argv[]) {
        if (argc != 2)
                fprint(stderr, “usage: cpu <string>\n”); exit(1);
        char *str = argv[1];
        while (1){
                Spin(1);
                printf(“%s\n”, str); }
        return 0;
 }
```

cpu.c

How to trace system call ?

- In Linux
    - strace /bin/ls



```
[ttyeh@linux1 IOC5226]$ strace /bin/ls
execve("/bin/ls", ["/bin/ls"], 0x7ffc641ab9a0 /* 50 vars */) = 0
brk(NULL)                               = 0x55931c67b000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffdda727190) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/remi/php74/root/usr/lib64/glibc-hwcaps/x86-64-v3/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No
such file or directory)
stat("/opt/remi/php74/root/usr/lib64/glibc-hwcaps/x86-64-v3", 0x7ffdda7263a0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/remi/php74/root/usr/lib64/glibc-hwcaps/x86-64-v2/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No
such file or directory)
stat("/opt/remi/php74/root/usr/lib64/glibc-hwcaps/x86-64-v2", 0x7ffdda7263a0) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/opt/remi/php74/root/usr/lib64/tls/haswell/x86_64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such
 file or directory)
```

    - “ls” command uses many system calls such as “execve”

I/O and File Descriptor

- **A file descriptor**
  - a small integer representing a kernel-managed object (read/write)
  - an index into a per-process table
  - Every process has a private space of file descriptors start at zero
- **A process**
  - Reads from file descriptor 0 (standard input)
  - Write to file descriptor 1 (standard output)
  - Write error messages to file descriptor 2 (standard error)

I/O and File Descriptor

- The read and write system calls read bytes from and write bytes to open files named by file descriptors

```
char buf[512]; int n;
for( ; ; ) {
        n = read(0, buf, sizeof(buf));
        if(n == 0) break;
        if(n < 0) {
                fprintf(2, “read error\n”);
                exit();
        }
        if(write(1, buf, n) != n) {
                fprintf(2, “write error\n”);
                exit();
        }
}
```

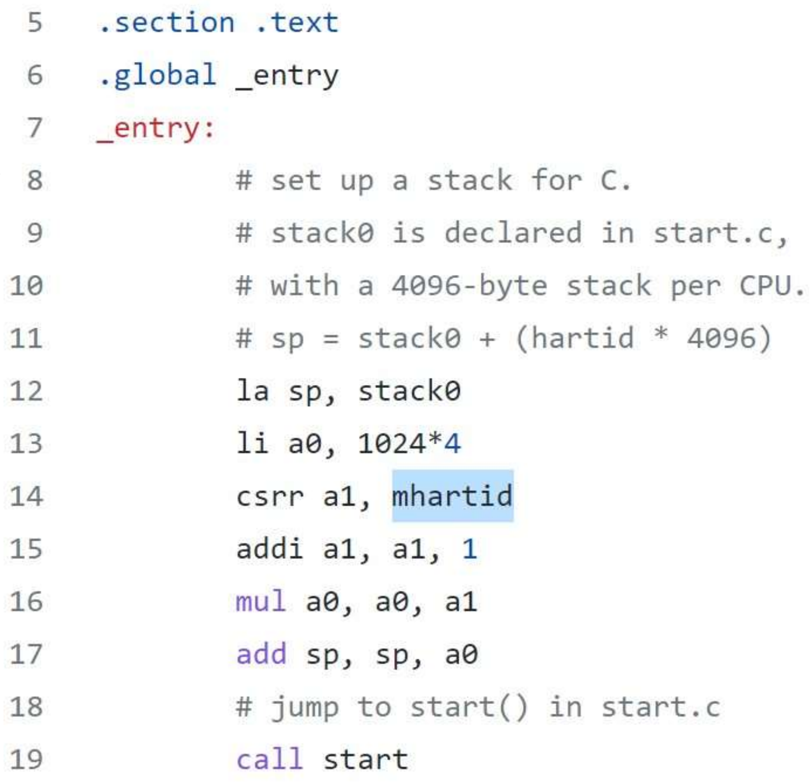
1. read (fd, buf, n) reads at most n bytes from the file descriptor fd, copies them into buf and return the number of bytes read
2. write(fd, buf, n) writes n bytes from buf to the file descriptor fd and return the number of bytes written
3. What does this program fragment work ?

   a. Copy data from its standard input to its standard output.

   b. If an error occurs, it writes a message to the standard error https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/entry.S

The first process and system call in JOS

- When the RISC-V computer powers on
  - **Runs a boot loader** in read-only memory
  - The boot loader **loads kernel** into memory
  - The CPU executes OS **starting at _entry**
  - The instructions at _entry set up a stack so that the OS can run C code
  - **An initial stack**: stack0
  - **The code at _entry loads the stack pointer register sp** with the address stack0 + 4096
  - The loader loads the kernel at physical address 0x80000000
  - The address range 0x0:0x80000000 contains I/O devices

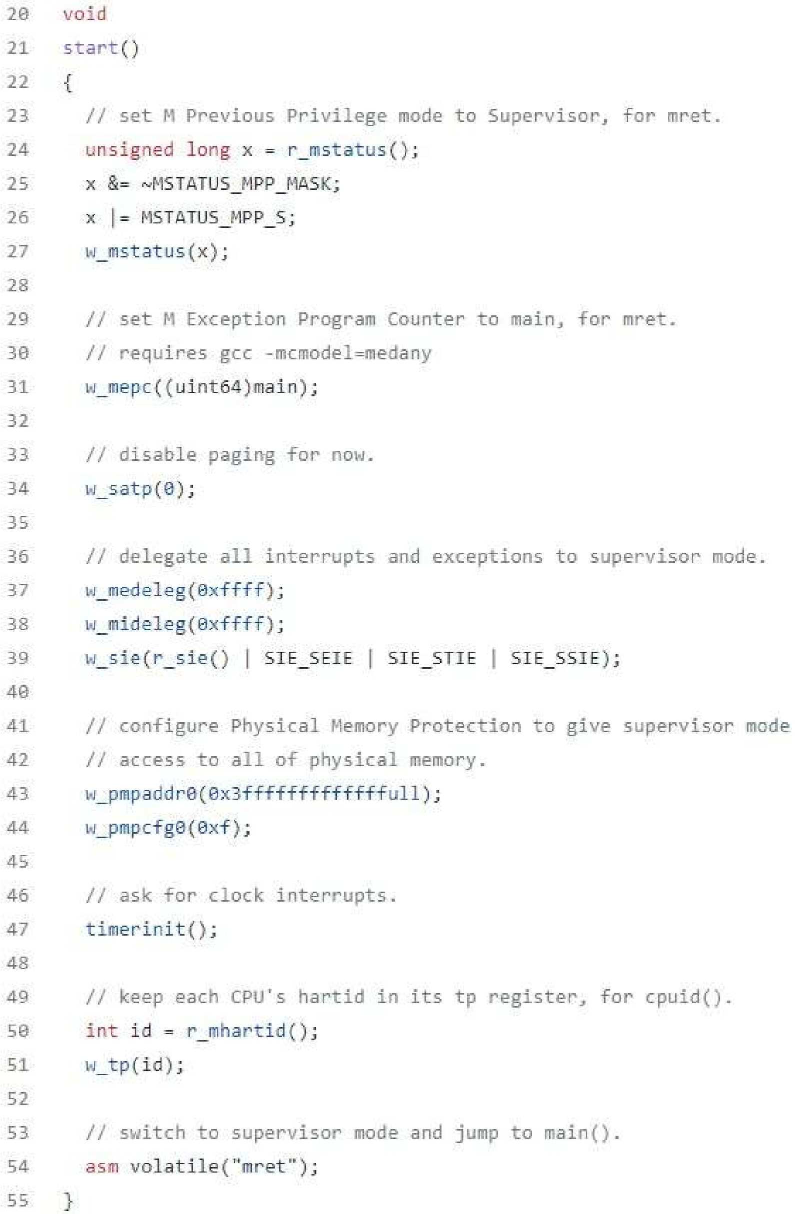


```
.section .text
.global _entry
_entry:
        # set up a stack for C.
        # stack0 is declared in start.c,
        # with a 4096-byte stack per CPU.
        # sp = stack0 + (hartid * 4096)
        la sp, stack0
        li a0, 1024*4
        csrr a1, mhartid
        addi a1, a1, 1
        mul a0, a0, a1
        add sp, sp, a0
        # jump to start() in start.c
        call start
```

The first process

- The function “start” performs configurations that switches to supervisor mode.
  - **Set the privilege mode** to supervisor in the register “mstatus”
  - Set the return address to main by **writing main’s address into the register “mepc”**
  - **Writing 0 into the page-table register** “**satp**” to disable virtual address translation
  - “start” returns to supervisor mode by calling “**mret**”

https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/start.c

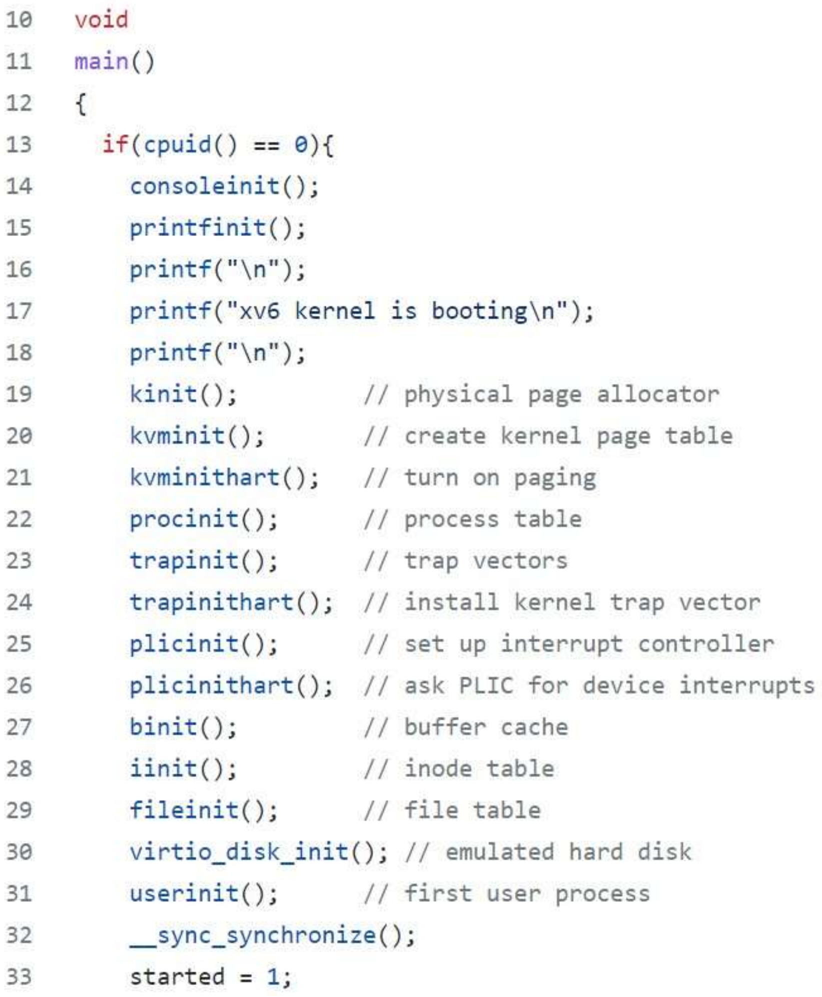


```c
void
start()
{
  // set M Previous Privilege mode to Supervisor, for mret.
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK;
  x |= MSTATUS_MPP_S;
  w_mstatus(x);

  // set M Exception Program Counter to main, for mret.
  // requires gcc -mcmodel=medany
  w_mepc((uint64)main);

  // disable paging for now.
  w_satp(0);

  // delegate all interrupts and exceptions to supervisor mode.
  w_medeleg(0xffff);
  w_mideleg(0xffff);
  w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);

  // configure Physical Memory Protection to give supervisor mode
  // access to all of physical memory.
  w_pmpaddr0(0x3fffffffffffffull);
  w_pmpcfg0(0xf);

  // ask for clock interrupts.
  timerinit();

  // keep each CPU's hartid in its tp register, for cpuid().
  int id = r_mhartid();
  w_tp(id);

  // switch to supervisor mode and jump to main().
  asm volatile("mret");
}
```

https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/main.c

The first process

- After main initializes several devices and subsystems, it creates the first process by calling “**userinit**”
  - The first process makes the first system call
  - **initcode.S** loads the number for the exec system call, SYS_EXEC into register a7
  - The kernel uses the number in register a7 to call the desired system call
  - Calls “ecall” to re-enter the kernel



```c
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
```

The Unix Shell

- It’s the Unix **command-line user interface**
  - Bourne Again Shell (BASH), C Shell, Korn Shell
  - Primary purpose is to read commands and run other programs
  - Programs can be run in Bash by entering commands at the command-line prompt
  - Advantages
    - Automating repetitive tasks
- Bash shell script language
  - The shell uses system calls to set up redirection, and pipes …
  - ls > file, ls | wc – l

Case Study: How to implement “cat < input.txt”?

- The system call “fork” copies the parent’s file descriptor table along with its memory
- The child starts with exactly the same open files as the parent

```
char *argv[2];

argv[0] = “cat”;
argv[1] = 0;
If(fork() == 0) { // child process
            close (0);
            open(“input.txt”, O_RDONLY);
            exec(“cat”, argv);
}
```

1. After the child closes file descriptor 0
2. Open uses file descriptor 0 for the newly opened input.txt
3. The system call “exec” replaces the calling process’s memory but preserves its file table.

Implement “echo hello; echo world > output”

- Fork copies the file descriptor table, each underlying file offset is shared between parent and child

```
if (fork() == 0) { // child process
            write(1, “hello”, 6);
            exit();
} else {
            wait();
            write(1, “world\n”, 6);
}
```

1. The file attached to file descriptor 1 will contain the data “hello world”.
2. The wait system call ensures the parent to run only after the child is done
3. The write in the parent picks up where the child’s write left off.

Implement “echo hello; echo world > output”

- The dup system call duplicates an existing file descriptor, return a new one that refers to the same underlying I/O object.

```
fd = dup(1);
write(1, “hello”, 6);
write(fd, “world\n”, 6)
```

1. Both file descriptors share an offset, just as the file descriptors duplicated by fork do

- Using dup to implement commands: 2>&1
  - The 2>&1 tells the shell to give the command a file descriptor 2 that is a duplicate of descriptor 1
  - “ls existing-file non-existing-file > tmp1”
  - Both the name of the existing file and the error message for the non-existing file will show up in the file tmp1

Pipes

- **A pipe**
  - A small kernel buffer as a pair of file descriptors, one for reading and one for writing
  - Provide a way for processes to communicate
- “dup” system call
  - Duplicates an existing file descriptor, return a new one that refers to the same underlying I/O object.
- “echo hello world | wc”

```
int p[2];
char *argv[2];
argv[0] = “wc”;
argv[1] = 0;
pipe(p);
if(fork() == 0) { // child process
        close(0);
        dup(p[0]);
        close(p[0]);
        close(p[1]);
        exec(“/bin/wc”, argv);
} else { // parent process
        close(p[0]);
        write(p[1], “hello world\n”, 12);
        close(p[1]);
}
```

How to implement a shell?

- How to implement a shell in rpi3 board in Lab 1 ?
  - Follow this tutorial: https://github.com/bztsrc/raspi3-tutorial
  - Using uart_init(), uart_getc(), uart_send() and uart_puts() routines

```
#include “uart.h”
int strcmp(const char *a, const char *b){ …}
int main()
{
        uart_init(); // set up serial console
        // read command
        …..
        return 0;
}
```

Concurrency support of the OS

```
#include <stdlib.h>
#include <stdio.h>
volatile int counter = 0;
int loops;
void *worker (void *arg) {
        int i;
        for(i = 0; i < loops; i++) counter ++
        return NULL; }
int main(int argc, char *argv[]) {
        loops = atoi (argv[1]);
        pthread_t p1, p2;
        printf(Initial value : %d\n, counter);
        pthread_create(&p1, NULL, worker, NULL);
        pthread_create(&p2, NULL, worker, NULL);
        pthread_join(&p1, NULL, worker, NULL);
        pthread_join(&p2, NULL, worker, NULL);
        printf(“Final value : %d\n”, counter);
        return 0; }
```

1. The main program creates two threads using pthread_create()
2. A thread as a function running within the same memory space as other functions
3. More than one of them active at a time
4. Each thread starts running in a routine called worker()
5. It simply increments a counter in a loop

thread.c

Concurrency support of the OS

```
#include <stdlib.h>
#include <stdio.h>
volatile int counter = 0;
int loops;
void *worker (void *arg) {
        int i;
        for(i = 0; i < loops; i++) counter ++
        return NULL; }
int main(int argc, char *argv[]) {
        loops = atoi (argv[1]);
        pthread_t p1, p2;
        printf(Initial value : %d\n, counter);
        pthread_create(&p1, NULL, worker, NULL);
        pthread_create(&p2, NULL, worker, NULL);
        pthread_join(&p1, NULL, worker, NULL);
        pthread_join(&p2, NULL, worker, NULL);
        printf(“Final value : %d\n”, counter);
        return 0; }
```

```
prompt> gcc –o thread thread.c –Wall –pthread
Prompt> ./thread 1000
```

What are outputs of this program?

```
Initial value : 0
Final value: 2000
```

Concurrency support of the OS

```
#include <stdlib.h>
#include <stdio.h>
volatile int counter = 0;
int loops;
void *worker (void *arg) {
        int i;
        for(i = 0; i < loops; i++) counter ++
        return NULL; }
int main(int argc, char *argv[]) {
        loops = atoi (argv[1]);
        pthread_t p1, p2;
        printf(Initial value : %d\n, counter);
        pthread_create(&p1, NULL, worker, NULL);
        pthread_create(&p2, NULL, worker, NULL);
        pthread_join(&p1, NULL, worker, NULL);
        pthread_join(&p2, NULL, worker, NULL);
        printf("Final value : %d\n", counter);
        return 0; }
```

```
Prompt> ./thread 100000
Initial value: 0
Final value : 143012
Prompt> ./thread 100000
Initial value: 0
Final value : 137298
```

Why are outputs are different in these two runs ?

Ans: the update of “counter” doesn’t execute atomically.