Operating Systems

Spring 2022

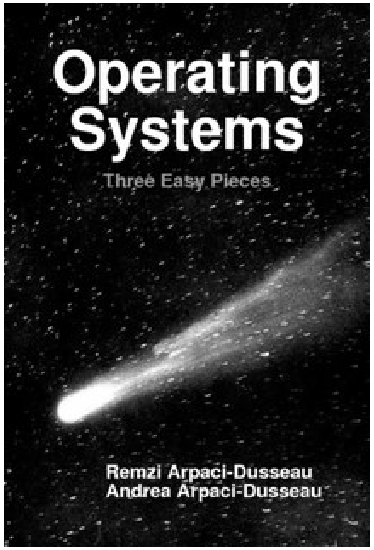
Syllabus

- **Instructors:**
  - Dongkun Shin
  - Office : Room 400310 (반도체관)
  - E-mail : dongkun@skku.edu
  - Office Hours: Mon. 14:00-16:00 or by appointment
  - Lecture Video: icampus
  - Online Class for Q&A (occasionally)
    - You can ask questions in English or Korean

- **Lecture notes**
  - iCampus
  - Lecture notes and talks will be given in English.

Syllabus (cont'd)

- **Main text**
  - Operating Systems: Three Easy Pieces
  - http://pages.cs.wisc.edu/~remzi/OSTEP/
  - free online book!!
  - The book is very easy and funny.
  - You have to read the scheduled chapter before the class time.



- **Grading policy (subject to change)**
  - Midterm exam: 40%
  - Final exam: 40%
  - Assignment (Report, Project): 20%

  - If you miss one or both of exams, you will fail this course.
  - Cheating on tests and other assignments will not be tolerated and you will take no (or a negative) point for the test!

Syllabus (cont'd)

- **Course Outline**
  - Part 1. CPU Virtualization
    - Process
    - CPU Scheduling
  - Part 2. Memory Virtualization
    - Address Space, Allocation
    - Address Translation
    - Paging, TLB, Swapping
  - Part 3. Concurrency
    - Thread
    - Lock, Semaphore
  - Part 4. Persistence
    - I/O Systems
    - Storage
    - File System

Syllabus (cont'd)

- **Assignments**
  - Homework
    - Reports on advanced topics
    - Homework in the textbook
  - A term project

- **Prerequisites**
  - C programming
  - System Programming
  - Computer Architecture

Chap.2
Introduction to Operating Systems

What happens when a program runs?

- **Von Neumann** model of Computing (Instruction Execution)
  - The processor fetches an instruction from memory,
  - Decodes it (i.e., figures out which instruction this is), and
  - Executes it (i.e., it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth).
  - After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on

Operating System (OS)

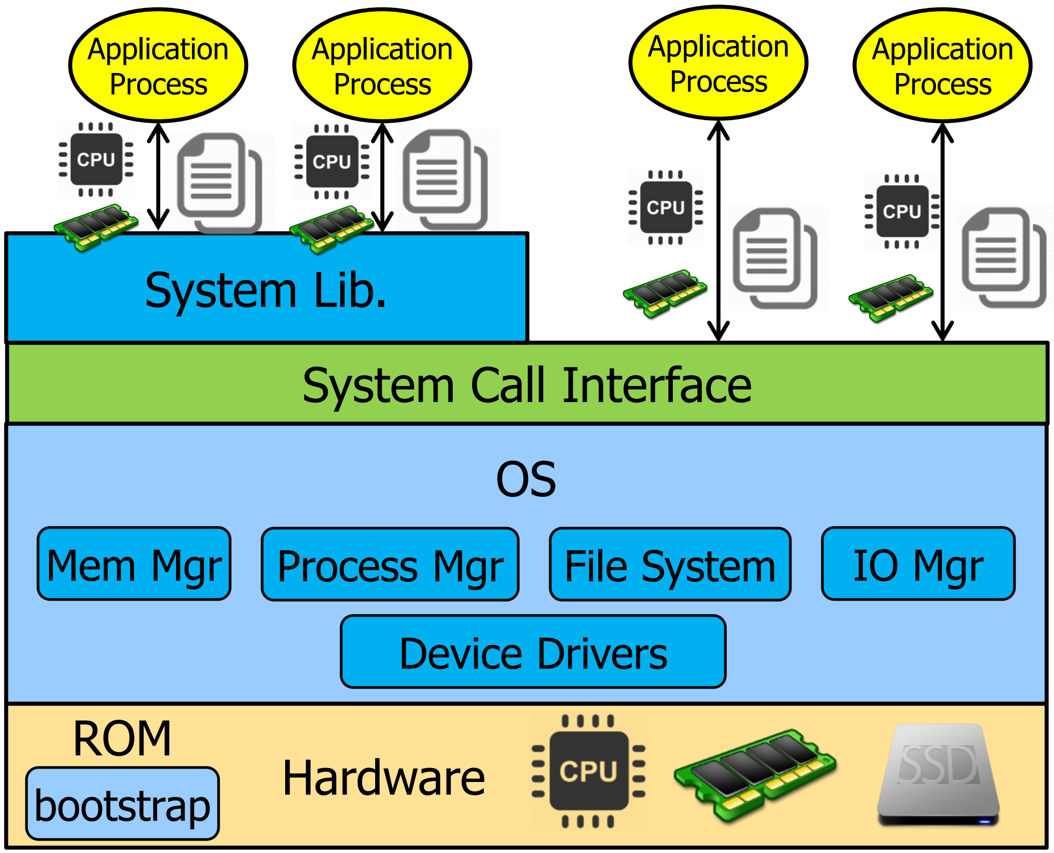
- Makes it easy to run programs
  - even many programs at the same time
- Allows programs to share memory
- Enables programs to interact with devices

➔ in charge of making sure the system operates correctly and efficiently in an easy-to-use manner

Operating System (OS)

- The role of OS
  - Virtualization
    - OS takes a physical resource (such as the processor, or memory, or a disk)
    - transforms it into a more general, powerful, and easy-to-use virtual form of itself.
    - System calls allow users to tell the OS what to do and thus make use of the features of the OS
    - Virtualization allows many programs to run concurrently
  - Resource Manager
    - Resource: CPU, memory, and disk
    - OS manages those resources efficiently or fairly or indeed with many other possible goals in mind.

Computer System Organization



Virtualizing the CPU

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2) {
            fprintf(stderr, "usage: cpu <string>\n");
            exit(1);
    }
    char *str = argv[1];

    while (1) {
            printf("%s\n", str);
            Spin(1);
    }
    return 0;
}
```

Spin(1): repeatedly checks the time and returns once it has run for a second

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Even though we have only one processor, somehow all four of these programs seem to be running at the same time!

Running Many Programs At Once

Virtualizing the CPU

- Illusion that the system has a very large number of virtual CPUs.
- Turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once

- **Two Questions**
  - Policy
    - If more than one program want to run at a particular time, which *should* run?
  - Mechanisms
    - How to implement the ability to run multiple programs at once?

Virtualizing Memory

```
int
main(int argc, char *argv[])
{
   int *p;
   p = malloc(sizeof(int));
   assert(p != NULL);
   printf("(%d) address pointed to by p: %pn",  getpid(), p);
   *p = 0;
   while (1) {
               Spin(1);
               *p = *p + 1;
               printf("(%d)  p: %d\n", getpid(), *p);
   }
   return 0;
}
```

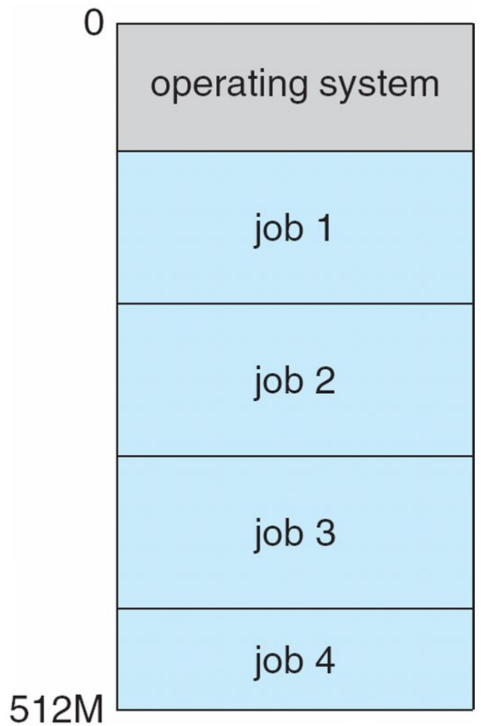
```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
ˆC
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Each running program has allocated memory at the same address (0x200000), and yet each seems to be updating the value at 0x200000 independently!
It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs

Virtualizing Memory

- Each process accesses its own private **virtual address space**
- OS maps the virtual address onto the physical memory of the machine.
- A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself.
- The reality, however, is that physical memory is a shared resource, managed by the operating system.



Concurrency

multi-threaded programs

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
   int i;
   for (i = 0; i < loops; i++) {
               counter = counter + 1;
   }
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   if (argc != 2) {
               fprintf(stderr, "usage: threads <loops>\n");
               exit(1);
   }
   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);
   Pthread_join(p2, NULL);
   printf("Final value   : %d\n", counter);
   return 0;
}
```

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298 // what the??
```

counter = counter + 1; ➔ no atomic

3 instructions

(1) load the value of the counter from memory into a register

(2) increment it

(3) store it back into memory.

How can we build a correctly working multi-threaded program? What primitives are needed from the OS?

Persistence

- DRAM is **volatile**
- We need hardware and software to store data **persistently**
  - H/W: HDD, SSD
  - S/W: **file system** manages the disk, responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system

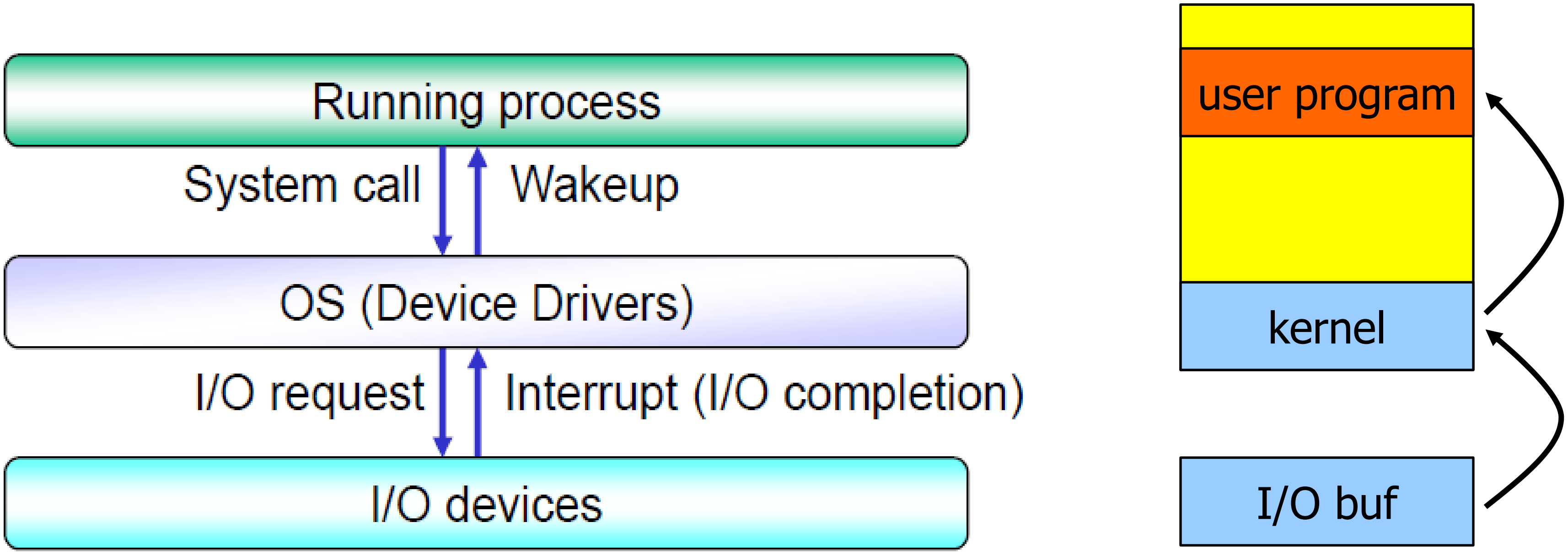
```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
    int fd = open("/tmp/file", O_WRONLY | O_CREAT |
                  O_TRUNC, S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

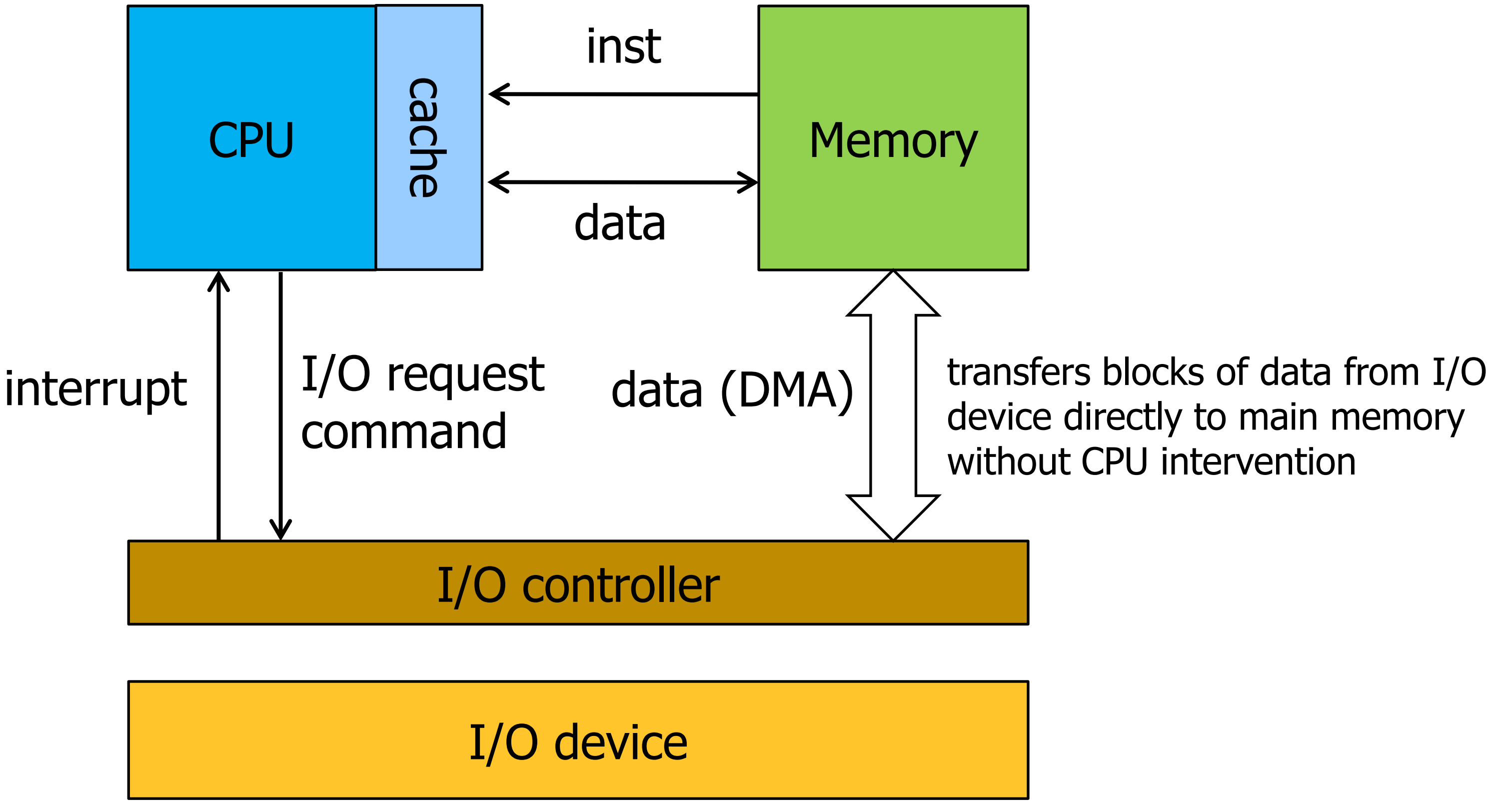
- OS does not create a private, virtualized disk for each application.
- Rather, users will want to share information that is in files.
- System calls: open, read, write, close
- Device driver issues I/O requests to the underlying storage device

Persistence

- **I/O management**



I/O Mechanism



OS includes interrupt handlers

Design Goals

- build up some **abstractions** in order to make the system convenient and easy to use
- provide high **performance**
- minimize the **overheads** of the OS
  - Extra time and space
- provide **protection** between applications, as well as between the OS and applications
  - isolating processes from one another is the key to protection
- **Reliability**
  - The operating system must also run non-stop
  - when it fails, all applications running on the system fail as well
- **energy-efficiency**, **security, mobility**
- Depending on how the system is used, the OS will have different goals and thus likely be implemented in at least slightly different ways.

History

- **Early Operating Systems: Just Libraries**
- **Beyond Libraries: Protection**
  - The idea of a **system call** was invented
  - System call transfers control (i.e., jumps) into the OS while simultaneously raising the **hardware privilege level**.
  - User applications run in what is referred to as **user mode** which means the hardware restricts what applications can do
  - **Trap** raises the privilege level to **kernel mode,** the OS has full access to the hardware of the system
- **The Era of Multiprogramming, Minicomputer**
  - OS loads a number of jobs into memory and switch rapidly between them, thus improving CPU utilization
  - Protection mechanisms are necessary to control access to system resources (including files)
  - **concurrency** issues
  - The introduction of the **UNIX** operating system
    - Ken Thompson (and Dennis Ritchie) at Bell Labs

History

- **The Modern Era**
  - personal computer: Apple II, IBM PC
    - Unfortunately, for operating systems, the PC at first represented a great leap backwards
    - forgot (or never knew of) the lessons learned in the era of minicomputers
    - DOS: no memory protection
    - Mac OS (v9 and earlier): cooperative job scheduling
  - Now
    - Mac OS X
      - Steve Jobs took his UNIX-based NeXTStep operating environment with him to Apple
    - Windows NT
    - Linux
      - **Linus Torvalds** wrote his own version of UNIX which borrowed heavily on the principles and ideas behind the original system, but not from the code base, thus avoiding issues of legality

History of UNIX

NeXTStep ➔ MacOS
1989 **2001**

1960' **1973** **1975~1983** **1991~1994**

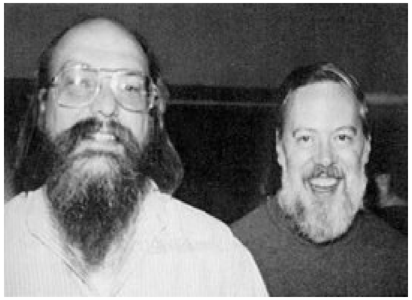
Multics

MIT → **ITS**

Bell Labs → **UNIX, C**

GE

Kenneth Thompson
Dennis Ritchie



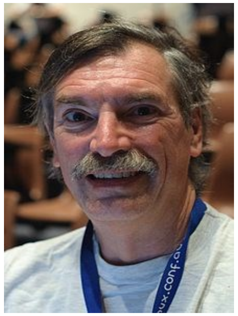
UNIX Source →

vi, Virtual Memory, File System, TCP/IP, …

BSD ver. 1, 2, 3, 4, 4.1, 4.2 → **FreeBSD**

UC Berkeley

Bill Joy, Marshall Kirk McKusick



Lawsuit

AT&T System V

1991
Linux 0.1

AIX (IBM)
HPUX (HP)
IRIX (SGI)

SunOS, Solaris
(Sun Microsystems)

Homework

- **Submit a report on** "The Evolution of the Unix Time-sharing System"