Lecture Note 1:
OS Introduction

March 4, 2025

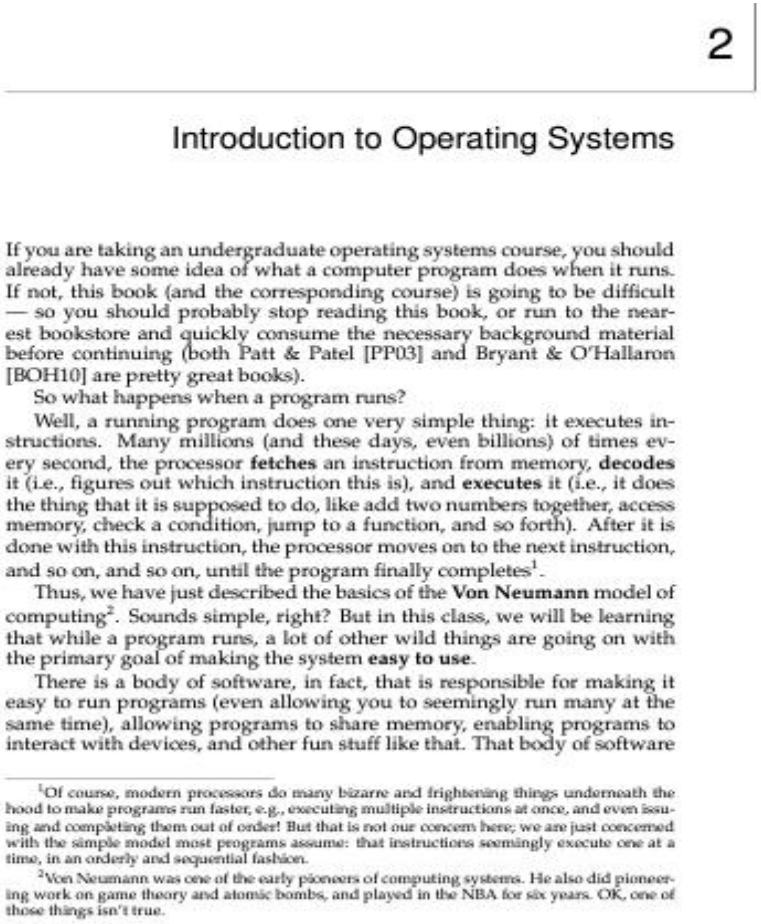
Jongmoo Choi

Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

(본 교재는 **2025**년도 과학기술정보통신부 및 정보통신기획평가원의 '**SW**중심대학사업' 지원을 받아 제작 되었습니다.)

Contents

- From Chap 1~2 of the OSTEP
- Chap 1. A Dialogue on the Book
- Chap 2. Introduction to Operating Systems
  - ✓ 2.1 Virtualizing CPU
  - ✓ 2.2 Virtualizing Memory
  - ✓ 2.3 Concurrency
  - ✓ 2.4 Persistence
  - ✓ 2.5 Design Goals
  - ✓ 2.6 Some History
  - ✓ 2.7 Summary
  - ✓ References



2

Introduction to Operating Systems

If you are taking an undergraduate operating systems course, you should already have some idea of what a computer program does when it runs. If not, this book (and the corresponding course) is going to be difficult — so you should probably stop reading this book, or run to the nearest bookstore and quickly consume the necessary background material before continuing (both Patt & Patel [PP03] and Bryant & O'Hallaron [BOH10] are pretty great books).

So what happens when a program runs?

Well, a running program does one very simple thing: it executes instructions. Many millions (and these days, even billions) of times every second, 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, until the program finally completes[1].

Thus, we have just described the basics of the **Von Neumann** model of computing[2]. Sounds simple, right? But in this class, we will be learning that while a program runs, a lot of other wild things are going on with the primary goal of making the system **easy to use**.

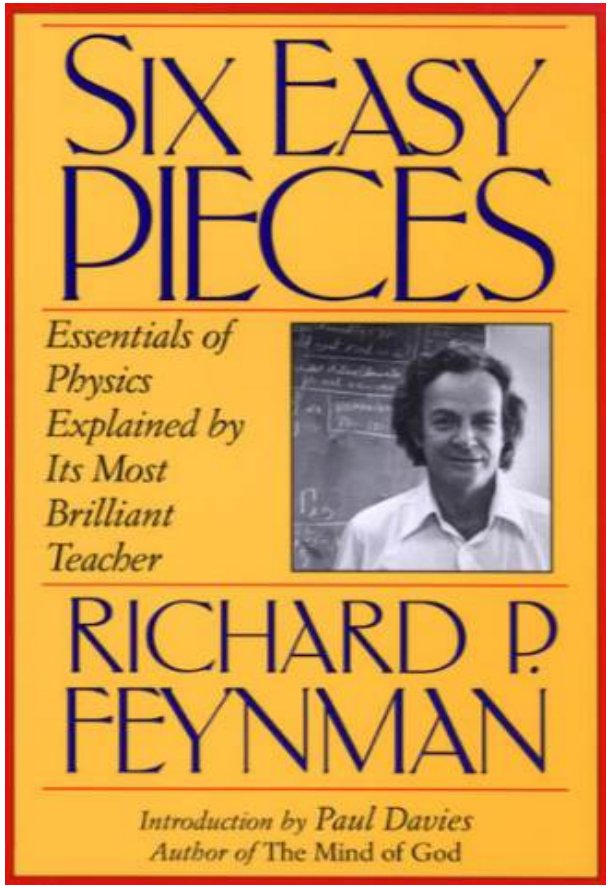
There is a body of software, in fact, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software

[1]Of course, modern processors do many bizarre and frightening things underneath the hood to make programs run faster, e.g., executing multiple instructions at once, and even issuing and completing them out of order! But that is not our concern here; we are just concerned with the simple model most programs assume: that instructions seemingly execute one at a time, in an orderly and sequential fashion.

[2]Von Neumann was one of the early pioneers of computing systems. He also did pioneering work on game theory and atomic bombs, and played in the NBA for six years. OK, one of those things isn't true.

Chap 1. A Dialog on the Book

- OSTEP
  - ✓ Operating Systems: Three Easy Pieces
  - ✓ Homage to the Feynman's famous "Six Easy Pieces on Physics"
    - OS is about half as hard as Physics: from Six to Three Pieces



CONTENTS

(Source: https://www.amazon.com/Six-Easy-Pieces-Essentials-Explained/dp/0465025277)

Chap 1. A Dialog on the Book

- OSTEP
  - ✓ What are Three Pieces: Virtualization, Concurrency, Persistence

Intro	Virtualization		Concurrency	Persistence	Security
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	52 *Dialogue*
TOC	4 Processes	13 Address Spaces code	26 Concurrency and Threads code	36 I/O Devices	53 *Intro Security*
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API code	37 Hard Disk Drives	54 *Authentication*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks code	38 Redundant Disk Arrays (RAID)	55 *Access Control*
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	56 *Cryptography*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables code	40 File System Implementation	57 *Distributed*
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores code	41 Fast File System (FFS)	
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	**Appendices**
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	*Dialogue*
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	Virtual Machines
		22 Swapping: Policies		45 Data Integrity and Protection	*Dialogue*
		23 Complete VM Systems		46 *Summary*	Monitors
		24 *Summary*		47 *Dialogue*	*Dialogue*
				48 Distributed Systems	Lab Tutorial
				49 Network File System (NFS)	Systems Labs
				50 Andrew File System (AFS)	xv6 Labs
				51 *Summary*	

(Source: http://pages.cs.wisc.edu/~remzi/OSTEP/)

Chap 1. A Dialog on the Book

- OSTEP
  - ✓ What to study?

Professor: *They are the three key ideas we're going to learn about:* ***virtualization, concurrency,*** *and* ***persistence****. In learning about these ideas, we'll learn all about how an operating system works, including how it decides what program to run next on a CPU, how it handles memory overload in a virtual memory system, how virtual machine monitors work, how to manage information on disks, and even a little about how to build a distributed system that works when parts have failed. That sort of stuff.*

Student: *I have no idea what you're talking about, really.*

Professor: *Good! That means you are in the right class.*

  - ✓ How to study?

Student: *I have another question: what's the best way to learn this stuff?*

Professor: *Excellent query! Well, each person needs to figure this out on their own, of course, but here is what I would do: go to class, to hear the professor introduce the material. Then, at the end of every week, read these notes, to help the ideas sink into your head a bit better. Of course, some time later (hint: before the exam!), read the notes again to firm up your knowledge. Of course, your professor will no doubt assign some homeworks and projects, so you should do those; in particular, doing projects where you write real code to solve real problems is the best way to put the ideas within these notes into action. As Confucius said...*

Student: *Oh, I know! 'I hear and I forget. I see and I remember. I do and I understand.' Or something like that.*

Professor: *(surprised) How did you know what I was going to say?!*

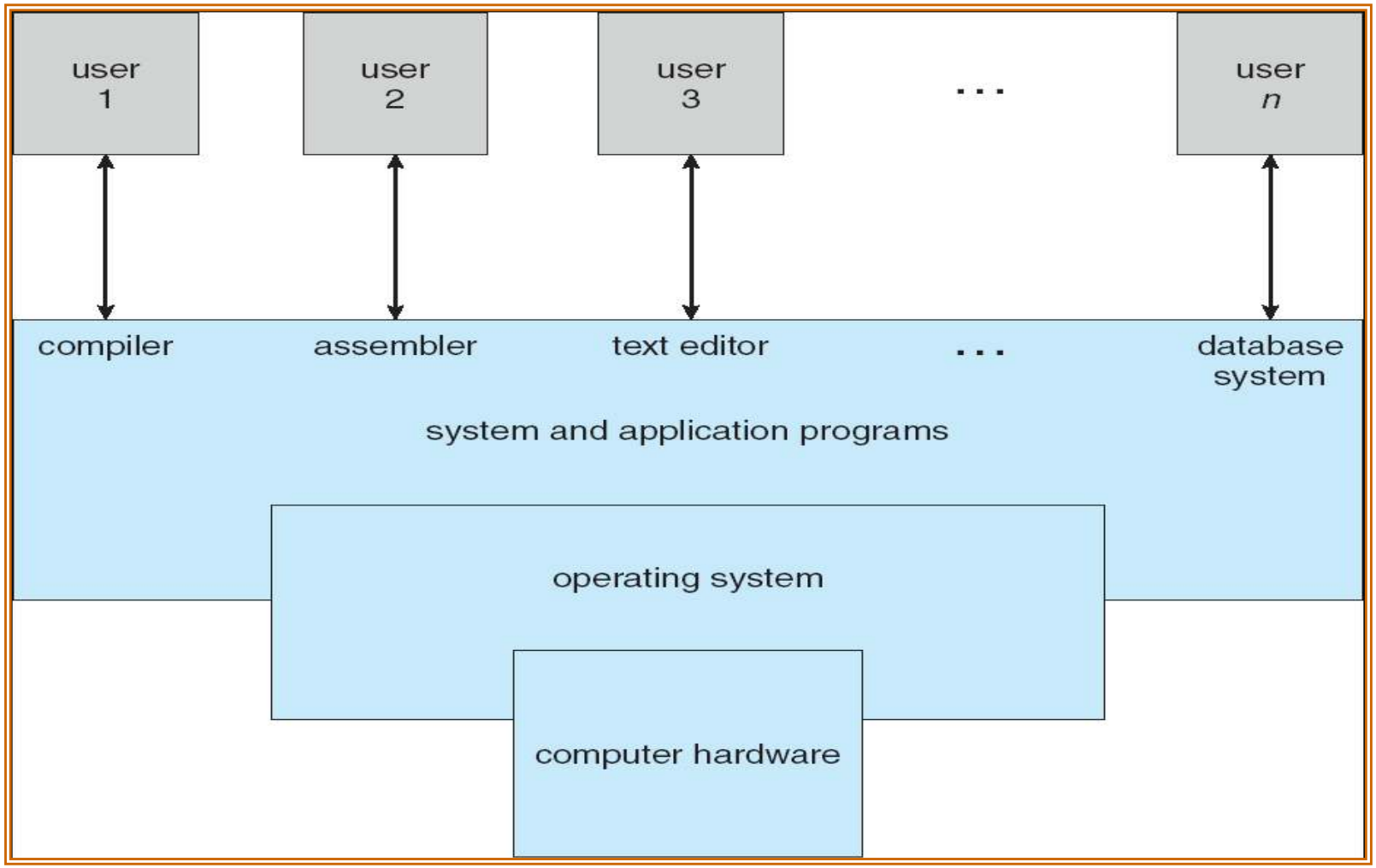
Student: *It seemed to follow. Also, I am a big fan of Confucius, and an even bigger fan of Xunzi, who actually is a better source for this quote*[1]*.*

Chap 2. Introduction to Operating Systems

- 2.1 Virtualizing CPU
- 2.2 Virtualizing Memory
- 2.3 Concurrency
- 2.4 Persistence
- 2.5 Design Goals
- 2.6 Some history
- 2.7 Summary
- References

Introduction

- Layered structure of a computer system



(Source: A. Silberschatz, "Operating system Concept")

Introduction

- What happens when a program runs?
  - ✓ 1. Simple view about running a program

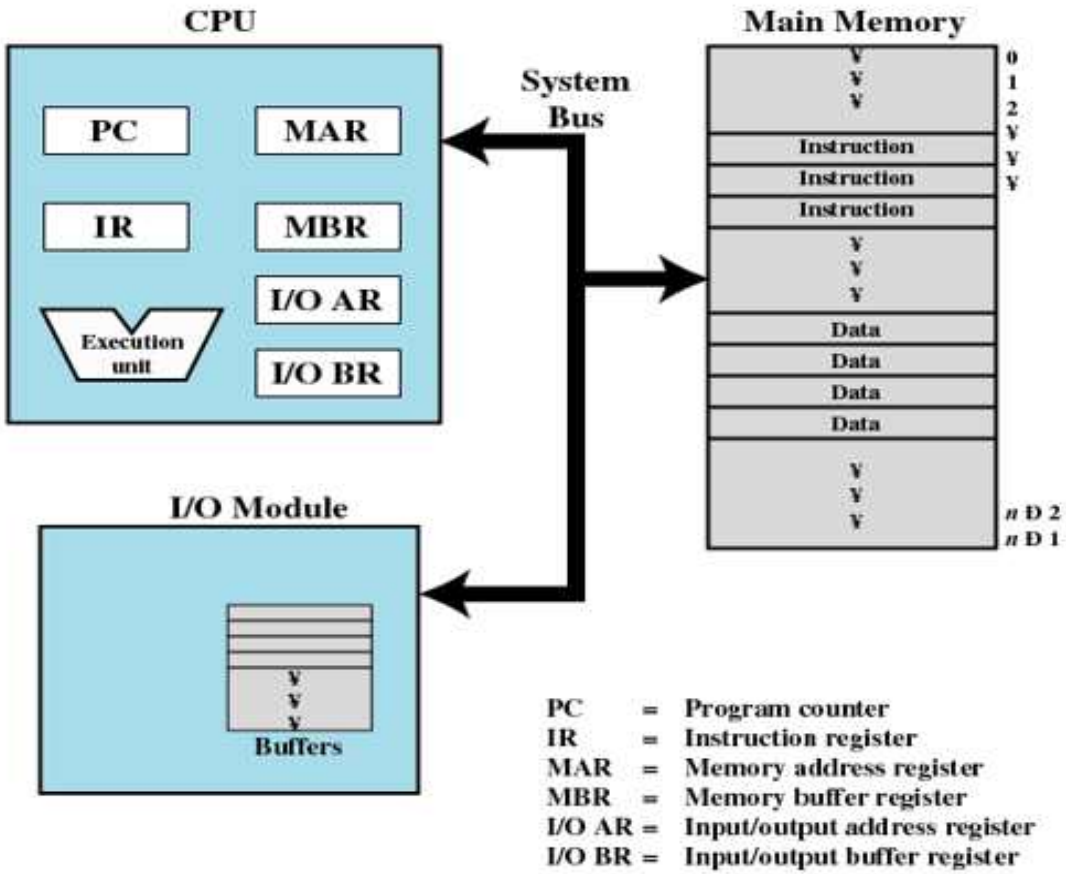
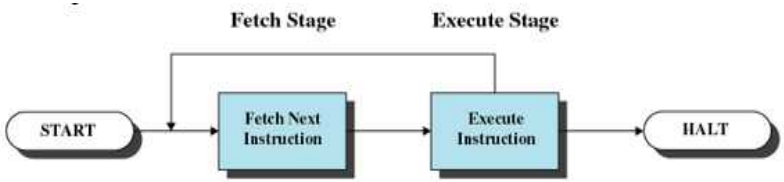


Figure 1.1 Computer Components: Top-Level View

(Source: W. Stalling, "Operating Systems: Internals and Design Principles")

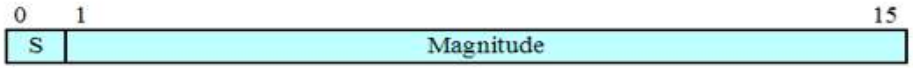
Introduction

- What happens when a program runs?
  - ✓ Details: execute instructions
    - Fetch and Execute



<Instruction cycle>

(a) Instruction format



(b) Integer format

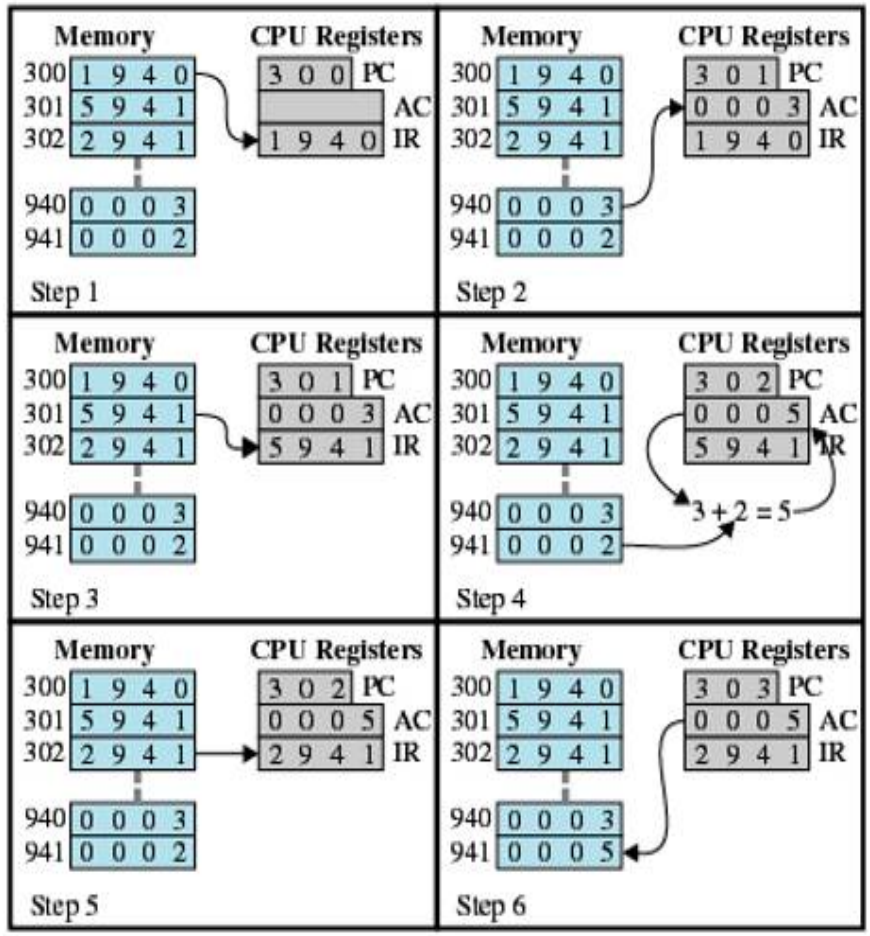
Program Counter (PC) = Address of instruction
Instruction Register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from Memory
0010 = Store AC to Memory
0101 = Add to AC from Memory

(d) Partial list of opcodes

<Hypothetical machine>



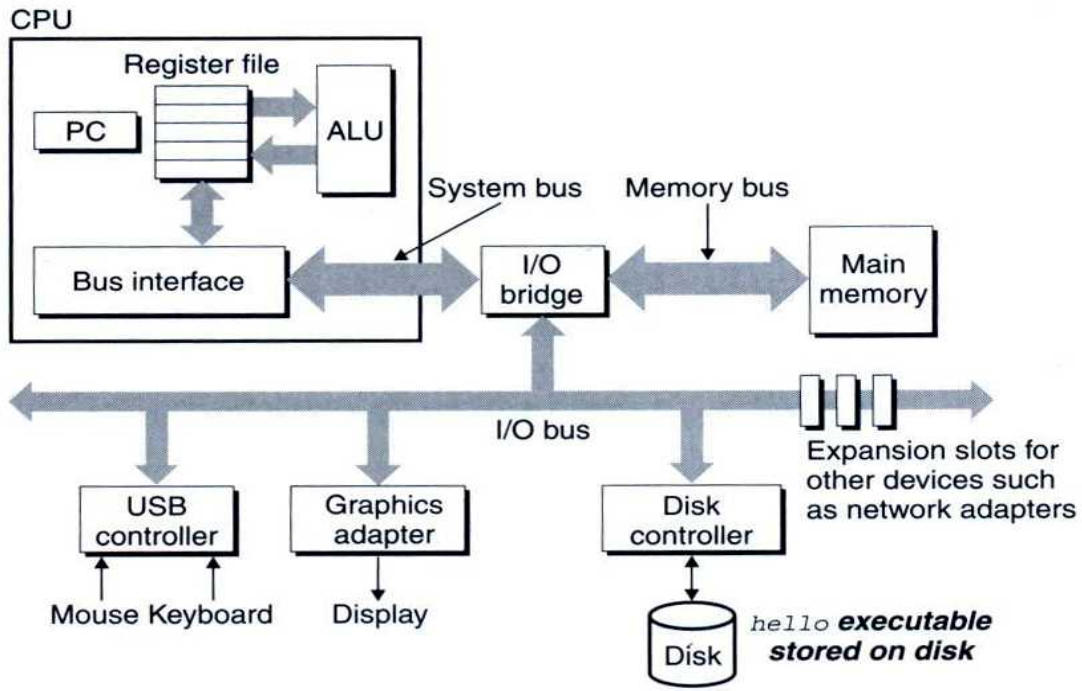
<Run example>

(Source: W. Stalling, "Operating Systems: Internals and Design Principles")

Introduction

- What happens when a program runs?
  - ✓ 2. A lot of stuff for running a program
    - Loading, memory management, scheduling, context switching, I/O processing, file management, IPC, …
    - Operating system: 1) make it easy to run programs, 2) operate a system correctly and efficiently

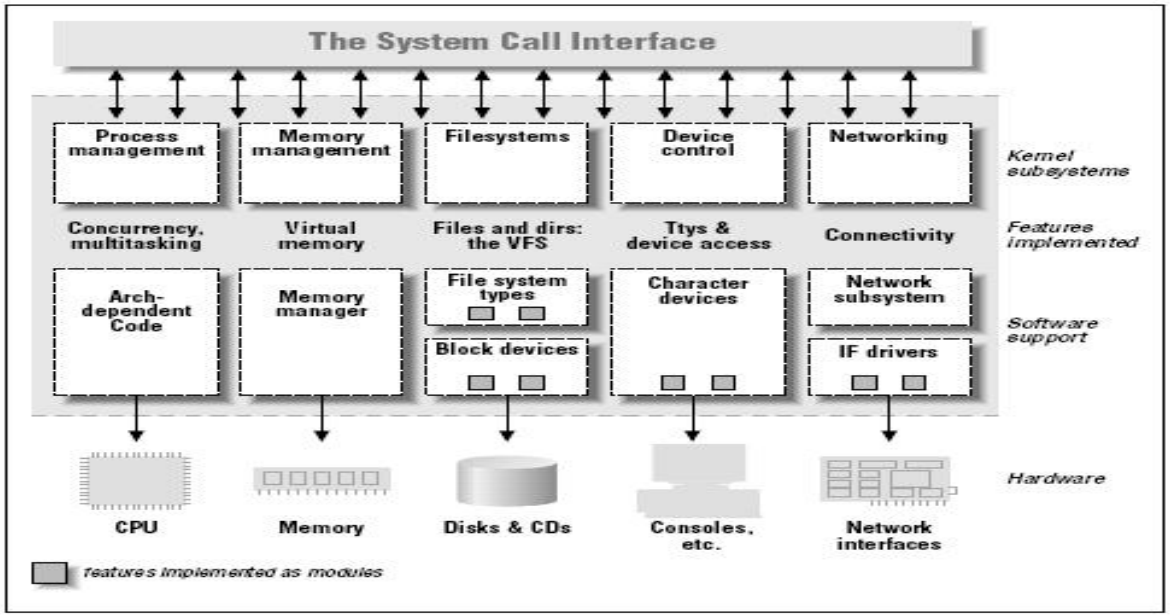
Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program Counter, USB: Universal Serial Bus.



(Source: computer systems: a programmer perspective)

Introduction

- Definition of operating system
  - ✓ Resource manager
    - Physical resources: CPU (core), DRAM, Disk, Flash, KBD, Network, …
    - Virtual resources: Process (task), Thread, Virtual memory, Page, File, Directory, Driver, Protocol, Access control, Security, …
  - ✓ Virtualization (Abstraction)
    - Transform a physical resource into a more general, powerful, and easy-to-use virtual form



(Source: Linux Device Driver, O'Reilly)

Introduction

- System call
  - ✓ Interfaces (APIs) provided by OS

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

(Source: A. Silberschatz, "Operating system Concept")

Introduction

- System call
  - ✓ Standard (e.g.. POSIX, Win32, …)
  - ✓ Mode switch (user mode, kernel mode)

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

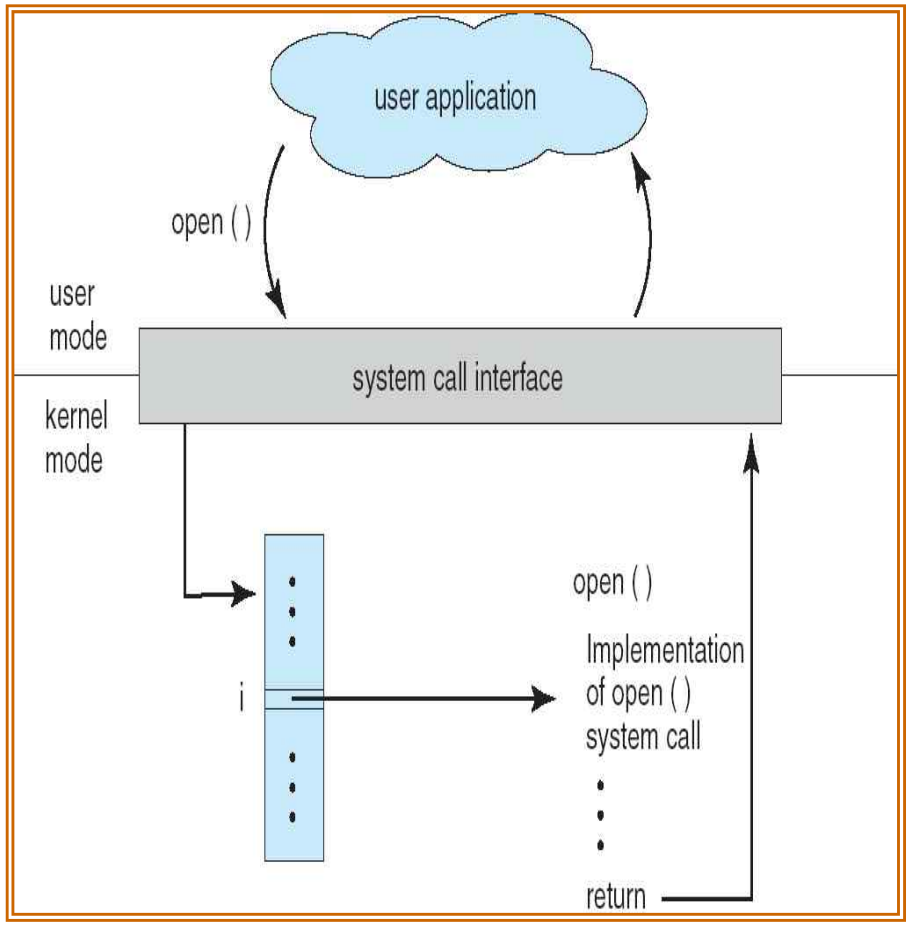
```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.



(Source: A. Silberschatz, "Operating system Concept")

2.1 Virtualizing CPU

- A program for the discussion of virtualizing CPU
  - ✓ call Spin (busy waiting and return when it has run for a second)
  - ✓ print out a string passed in on the command line

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/time.h>
4   #include <assert.h>
5   #include "common.h"
6
7   int
8   main(int argc, char *argv[])
9   {
10      if (argc != 2) {
11          fprintf(stderr, "usage: cpu <string>\n");
12          exit(1);
13      }
14      char *str = argv[1];
15      while (1) {
16          Spin(1);
17          printf("%s\n", str);
18      }
19      return 0;
20  }
```

Figure 2.1: **Simple Example: Code That Loops and Prints (cpu . c)**

2.1 Virtualizing CPU

- Execute the CPU program

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

- Execute the program in parallel

```
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
...
```

☛ Process, Scheduling, …

Figure 2.2: **Running Many Programs At Once**

2.1 Virtualizing CPU

- Issues for Virtualizing CPU
  - ✓ How to run a new program? ➔ process
  - ✓ How to make a new process? ➔ fork()
  - ✓ How to stop a process? ➔ exit()
  - ✓ How to execute a new process? ➔ exec()
  - ✓ How to block a process? ➔ sleep(), pause(), lock(), …
  - ✓ How to select a process to run next? ➔ scheduling
  - ✓ How to run multiple processes? ➔ context switch
  - ✓ How to manage multiple cores (CPUs)? ➔ multi-processor scheduling, cache affinity, load balancing
  - ✓ How to communicate among processes? ➔ IPC (Inter-Process Communication), socket
  - ✓ How to notify an event to a process? ➔ signal (e.g. ^C)
  - ✓ …
  - ☛ Illusion: A process has its own CPU even though there are less CPUs than processes

2.2 Virtualizing Memory

- Memory
  - ✓ Can be considered as an array of bytes
- Another program example
  - ✓ Allocate a portion of memory and access it

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

Figure 2.3: **A Program That Accesses Memory (mem.c)**

2.2 Virtualizing Memory

- Execute the Mem program

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

- Execute the program in parallel

```
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
...
```

Figure 2.4: Running The Memory Program Multiple Times

☛ Same address but independent

2.2 Virtualizing Memory

- Issues for Virtualizing Memory
  - ✓ How to manage the address space of a process? ➔ address space: text, data, stack, heap, …
  - ✓ How to allocate memory to a process? ➔ malloc(), calloc(), brk(), …
  - ✓ How to deallocate memory from a process? ➔ free()
  - ✓ How to manage free space? ➔ buddy, slab, …
  - ✓ How to protect memory among processes? ➔ virtual memory
  - ✓ How to implement virtual memory? ➔ page, segment
  - ✓ How to reduce the overhead of virtual memory? ➔ TLB
  - ✓ How to share memory among processes? ➔ shared memory
  - ✓ How to exploit memory to hide the storage latency? ➔ page cache, buffer cache, …
  - ✓ How to manage NUMA? ➔ local/remote memory
  - ✓ …

  ☛ Illusion: A process has its own unlimited and independent memory even though several processes are sharing limited memory in reality

2.3 Concurrency

- Background: how to create a new scheduling entity?
  - ✓ Two programming model: process (task) and thread
  - ✓ Key difference: data sharing

```
// fork example (Refer to the Chapter 5 in OSTEP)
// by J. Choi  (choijm@dku.edu)
#include <stdio.h>
#include <stdlib.h>

int a = 10;

void *func()
{
      a++;
      printf("pid = %d\n", getpid());
}

int main()
{
      int pid;
      if ((pid = fork()) == 0) { //need exception handle
            func();
            exit(0);
      }
      wait();
      printf("a = %d by pid = %d\n", a, getpid());
}
```

```
// thread example (Refer to the Chapter 27 in OSTEP)
// by J. Choi (choijm@dku.edu)
#include <stdio.h>
#include <stdlib.h>

int a = 10;

void *func()
{
      a++;
      printf("pid = %d\n", getpid());
}

int main()
{
      pthread_t p_thread;
      if ((pthread_create(&p_thread, NULL, func, (void *)NULL))
< 0) {
            exit(0);
      }
      pthread_join(p_thread, (void *)NULL);
      printf("a = %d by pid = %d\n", a, getpid());
}
```

(Source: System programming lecture site, http://embedded.dankook.ac.kr/~choijm/course/course202302.html)

2.3 Concurrency

- Concurrency
  - ✓ Problems arise when working on many things simultaneously on the same data
- A program for discussing concurrency

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "common.h"
4
5   volatile int counter = 0;
6   int loops;
7
8   void *worker(void *arg) {
9       int i;
10      for (i = 0; i < loops; i++) {
11          counter++;
12      }
13      return NULL;
14  }
15
16  int
17  main(int argc, char *argv[])
18  {
19      if (argc != 2) {
20          fprintf(stderr, "usage: threads <value>\n");
21          exit(1);
22      }
23      loops = atoi(argv[1]);
24      pthread_t p1, p2;
25      printf("Initial value : %d\n", counter);
26
27      Pthread_create(&p1, NULL, worker, NULL);
28      Pthread_create(&p2, NULL, worker, NULL);
29      Pthread_join(p1, NULL);
30      Pthread_join(p2, NULL);
31      printf("Final value   : %d\n", counter);
32      return 0;
33  }
```

Figure 2.5: A Multi-threaded Program (threads.c)

2.3 Concurrency

- Execute the multi-thread program

```
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??
```

- ✓ Programing model
  - thread model: share data section (a.k.a data segment)
  - process model: independent, need explicit IPC for sharing
- ✓ Reason for the odd results for the large loop
  - Lack of atomicity, scheduling effect, … ➔ need concurrency control

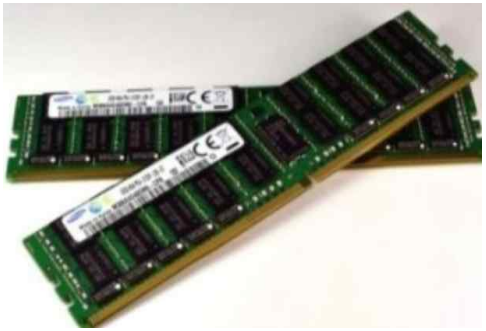
2.3 Concurrency

- Issues for Concurrency
  - ✓ How to support concurrency correctly? ➔ race condition, mutual exclusion, atomicity
  - ✓ How to implement atomicity ➔ lock(), semaphore()
  - ✓ How to implement atomicity in hardware? ➔ test_and_set(), swap()
  - ✓ How to solve the traditional concurrent problems such as producer-consumer, readers-writers and dining philosophers?
  - ✓ What is the semaphore?
  - ✓ What is the monitor?
  - ✓ What is a deadlock?
  - ✓ How to deal with the deadlock?
  - ✓ How to handle the timing bug?
  - ✓ What is the asynchronous I/Os?
  - ✓ …

☛ Illusion: Multiple processes run in a cooperative manner on shared resources even though they actually race with each other on the resources

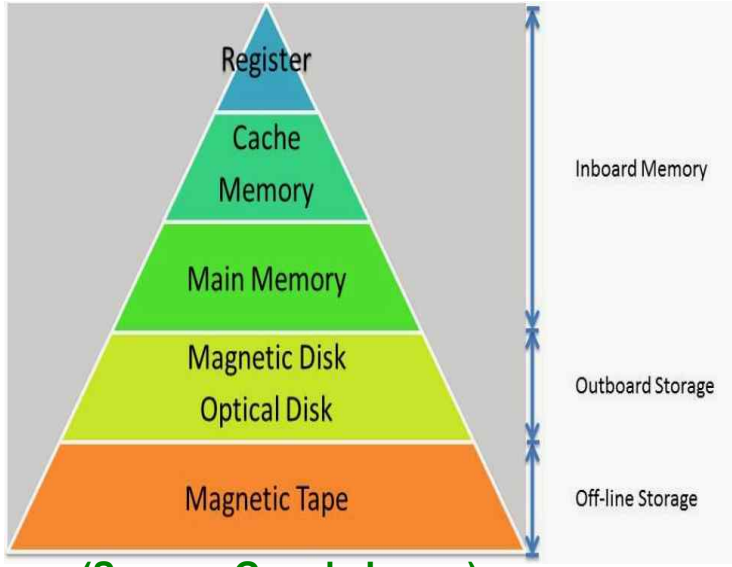
2.4 Persistence

- Background: DRAM vs. Disk



vs

- ✓ Capacity, Speed, …
- ✓ Durability: Volatile vs. Non-volatile
- ✓ Access granularity: Byte vs. Sector



(Source: Google Image)

2.4 Persistence

- Persistence
  - ✓ Users want to maintain data permanently (durability)
  - ✓ DRAM is volatile, requiring write data into storage (disk, SSD) explicitly
- A program for discussing persistence
  - ✓ Use the notion of a file (not handle disk directly)

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <assert.h>
4   #include <fcntl.h>
5   #include <sys/types.h>
6
7   int main(int argc, char *argv[]) {
8       int fd = open("/tmp/file",
9                     O_WRONLY|O_CREAT|O_TRUNC,
10                    S_IRWXU);
11      assert(fd > -1);
12      int rc = write(fd, "hello world\n", 13);
13      assert(rc == 13);
14      close(fd);
15      return 0;
16  }
```

Figure 2.6: A Program That Does I/O (io.c)

2.4 Persistence

- Issues for Persistence
  - ✓ How to access a file? ➔ open(), read(), write(), …
  - ✓ How to manage a file? ➔ inode, FAT, …
  - ✓ How to manipulate a directory?
  - ✓ How to design a file system? ➔ UFS, LFS, Ext2/3/4, FAT, F2FS, NFS, AFS, …
  - ✓ How to find a data in a disk?
  - ✓ How to improve performance in a file system? ➔ cache, delayed write, …
  - ✓ How to handle a fault in a file system? ➔ journaling, copy-on-write
  - ✓ What is a role of a disk device driver?
  - ✓ What are the internals of a disk and SSD?
    - SSD (Solid State Drive): flash memory, FTL, parallelism, error handling
  - ✓ What is the RAID?

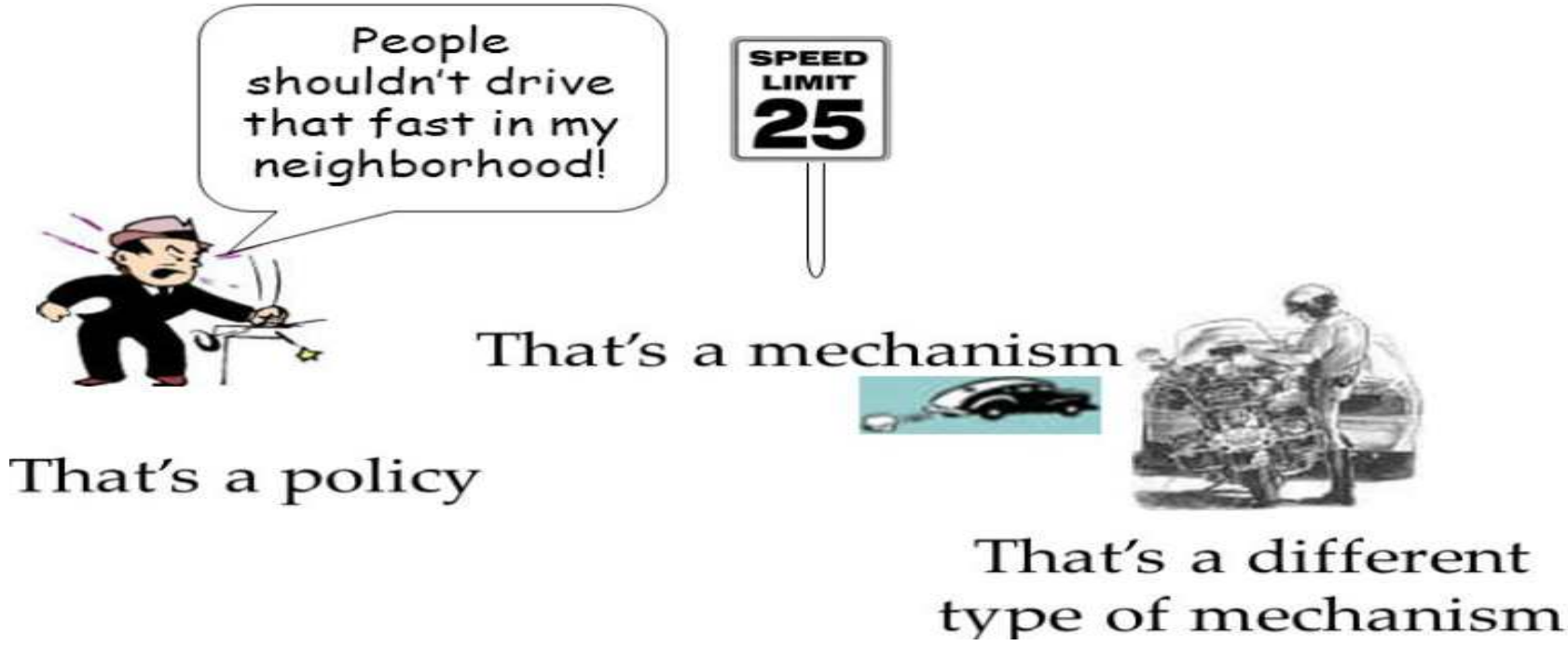
☛ Illusion: Data is always maintained in a reliable non-volatile area while it is often kept in a volatile DRAM (for performance reason) and storage is broken from time to time.

2.5 Design Goals

- Abstraction
  - ✓ Focusing on relevant issues only while hiding details
    - E.g. Car, File system, Make a program without thinking of logic gates
  - ✓ "Abstraction is fundamental to everything we do in computer science" by Remzi
- Performance
  - ✓ Minimize the overhead of the OS (both time and space)
- Protection
  - ✓ Isolate processes from one another
  - ✓ Access control, security, …
- Reliability
  - ✓ Fault-tolerant, redundancy, Cloud system, …
- Others
  - ✓ Depend on the area where OS is employed
  - ✓ Real time, Energy-efficiency, Mobility, Load balancing, Autonomous, …

2.5 Design Goals

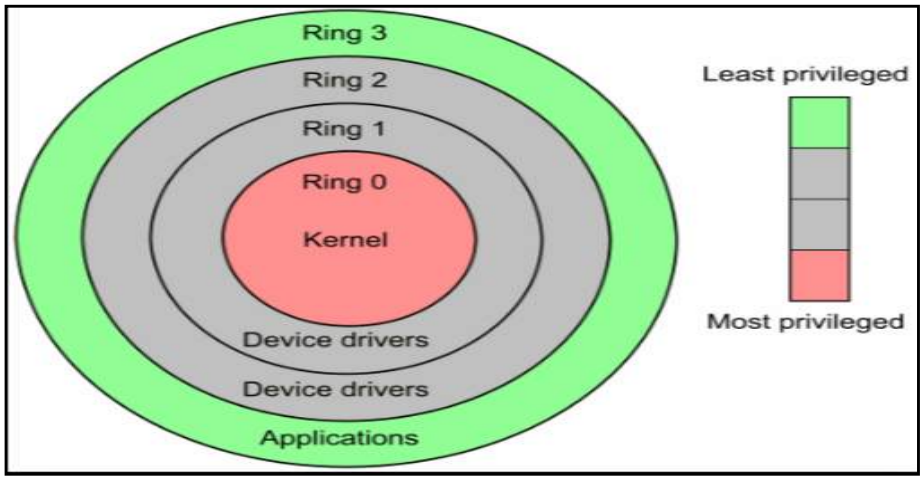
- Separation of Policy and Mechanism
  - ✓ Policy: Which (or What) to do?
    - e.g.) Which process should run next?
  - ✓ Mechanism: How to do?
    - e.g.) Multiple processes are managed by a queue or RB-tree



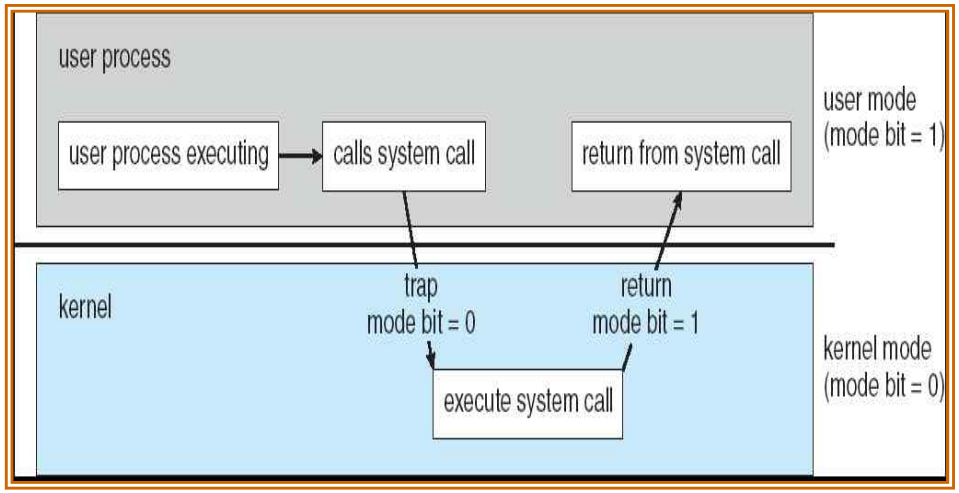
(Source: Security Principles and Policies CS 236 On-Line MS Program Networks and Systems Security, Peter Reiher, Spring, 2008)

2.6 Some History

- Early Operating Systems: Just libraries
  - ✓ Commonly-used functions such as low-level I/Os (e.g. MS-DOS)
  - ✓ Batch processing
    - A number of jobs were set up and then run all together (Not interactive)
- Beyond Libraries: Protection
  - ✓ Require OS to be treated differently than user applications
  - ✓ Separation user/kernel mode, system call
  - ✓ Use trap (special instruction, SW interrupt) to go into the kernel mode
    - Transfer control to a pre-specific trap handler (system_call handler)



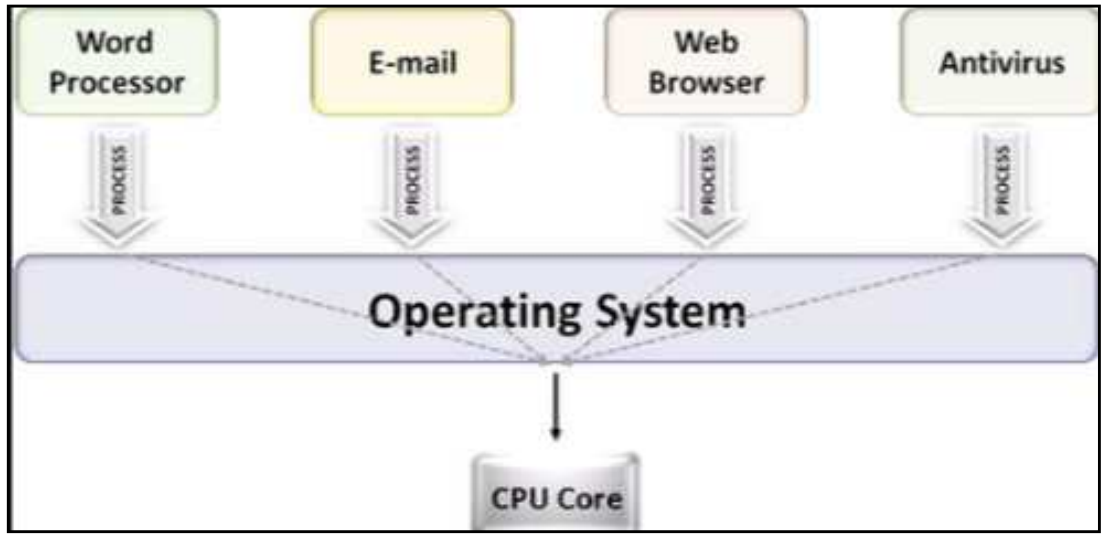
(Source: Google Image)

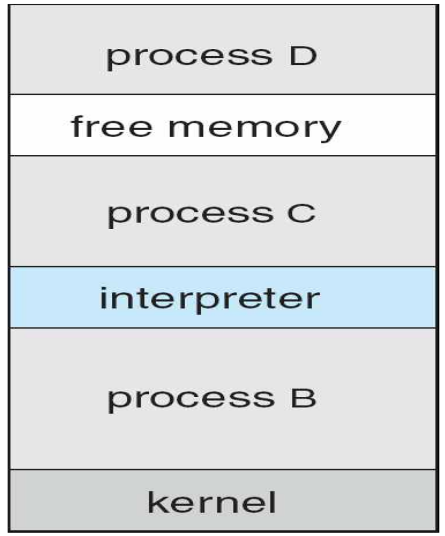


(Source: A. Silberschatz, "Operating system Concept")

2.6 Some History

- The Era of Multiprogramming (c.f. multitasking)
  - ✓ Definition: OS load a number of applications into memory and switch them rapidly
    - Reason: Advanced hardware ➔ Want to utilize machine resources better ➔ Multiple users share a system ➔ multiprogramming (memory virtualization) ➔ multitasking (CPU virtualization)
    - Especially important due to the slow I/O devices ➔ while doing I/O, switch CPU to another process ➔ enhancing CPU utilization
  - ✓ Memory protection and concurrency become quite important ➔ UNIX

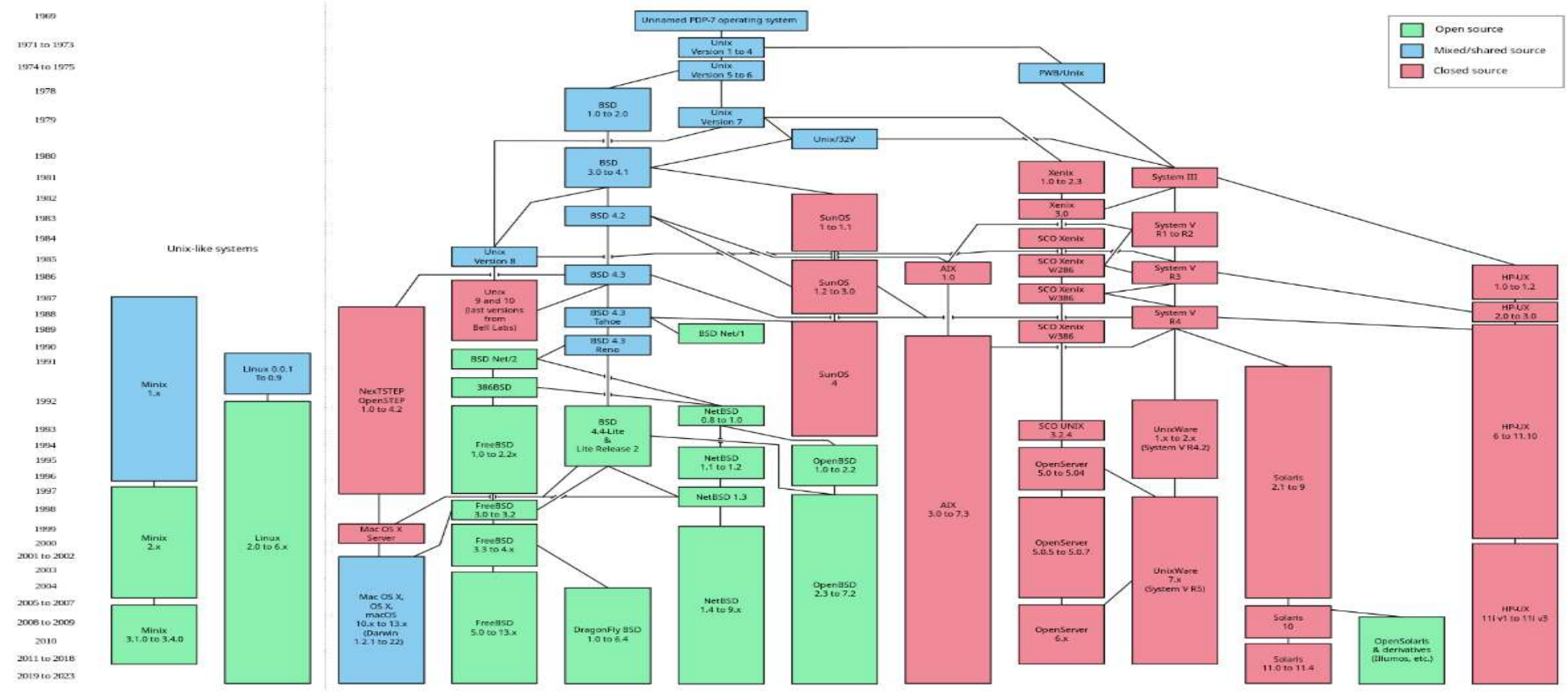




(Source: Google Image)

2.6 Some History

- The Era of Multiprogramming (c.f. multitasking)
  - ✓ UNIX
    - By Ken Thompson and Dennis Ritche (Bell Labs), Influenced by Multics
    - C language based, excellent features such as shell, pipe, inode, small, everything is a file, RR, VM, …
    - Influence OSes such as BSD, SUNOS, Mac OS, Windows NT, and Linux



(Source: https://en.wikipedia.org/wiki/Unix-like#/media/File:Unix_history-simple.svg)

2.6 Some History

- The Modern Era
  - ✓ PC
    - MS Windows, Mac OS X, Linux, …
  - ✓ Smartphone
    - Android, iOS, Windows Mobile, …
  - ✓ IoT
    - What is the next?

(Source: https://topgear-autoguide.com/category/traffic/car-operating-systems-from-tesla-google-daimler-bmw-or-vw1607935498)

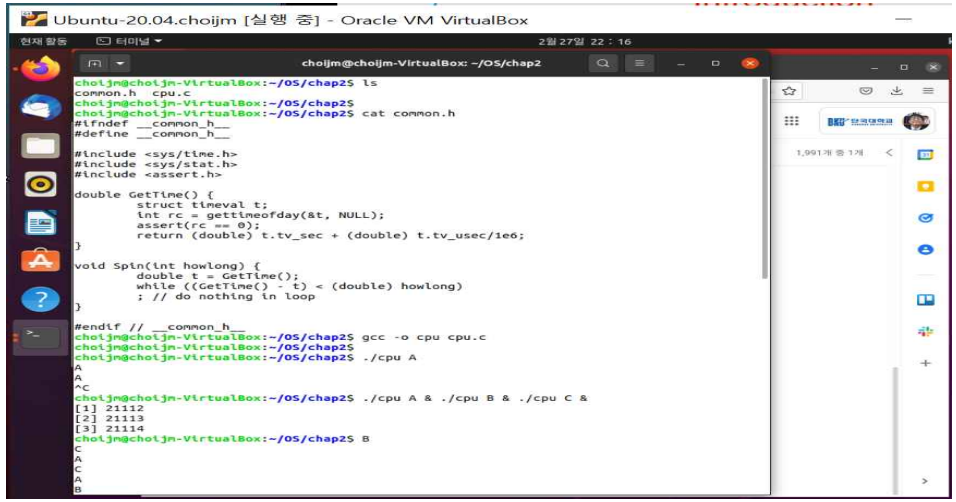
2.7 Summary

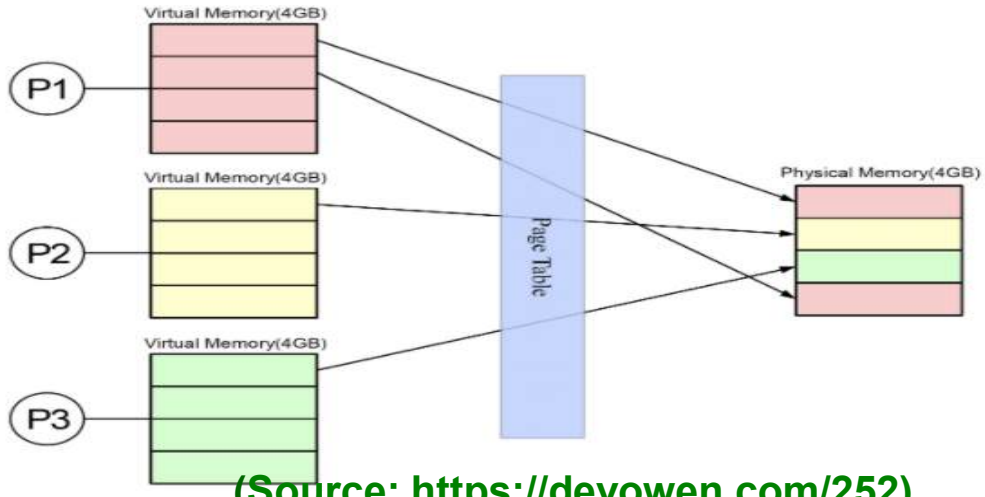
- OS
  - ✓ Resource manager (Efficiency)
  - ✓ Make systems easy to use (Convenience)
- Cover in this book
  - ✓ Virtualization, Concurrency, Persistence
- Not being covered
  - ✓ Security, Network, Graphics
  - ✓ There are several excellent courses for them

☛ Homework 1: Summarize the chap 2 of the OSTEP (reading assignment)
- Requirement: 1) Summary of Chap 2 (**NOT ppt, But text**), 2) Goal of your OS study
- Recommendation: highlight key words or sentences (different size or font), using figures and tables (your own), using itemization or numbering
- Due: Same time of Next week
- Format: pdf
- Bonus: **Snapshot of the results of example programs in a Linux system** (ubuntu on virtual box or wsl or Linux server)

Quiz for this Lecture

- Quiz
  - ✓ 1. OS is defined as a resource manager. What kinds of virtual resources are managed by OS for CPU, DRAM, and disk?
  - ✓ 2. What is the role of "&" in the below example? (I do this experiment using virtualbox + ubuntu in my laptop.)
  - ✓ 3. In today's lecture, we executed two processes (whose pids are 24113 and 24114, respectively). Discuss the differences between physical memory and virtual memory using two these processes.
  - ✓ 4. When we manage a disk, we need to consider the persistence carefully. Explain why the persistence is not considered in DRAM using the difference between disks and DRAM.

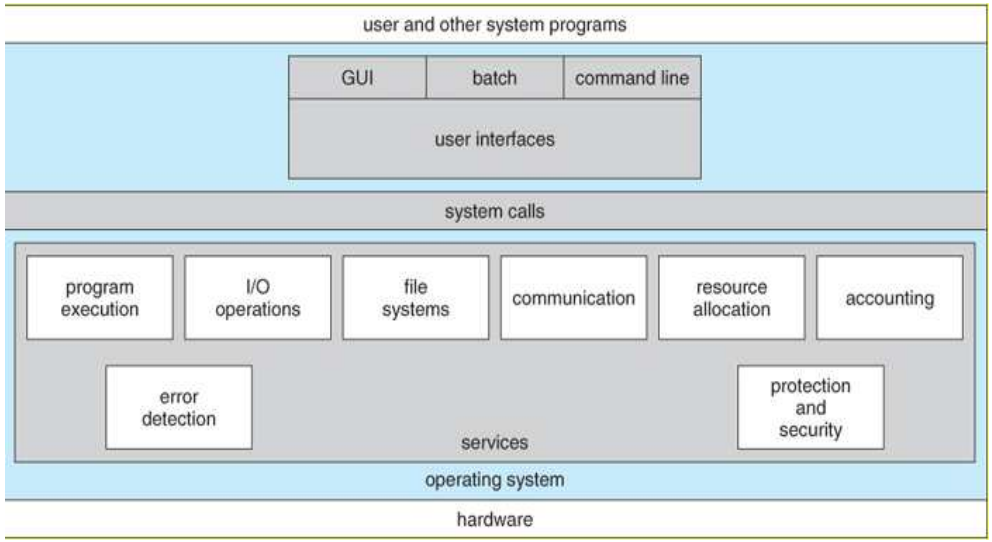




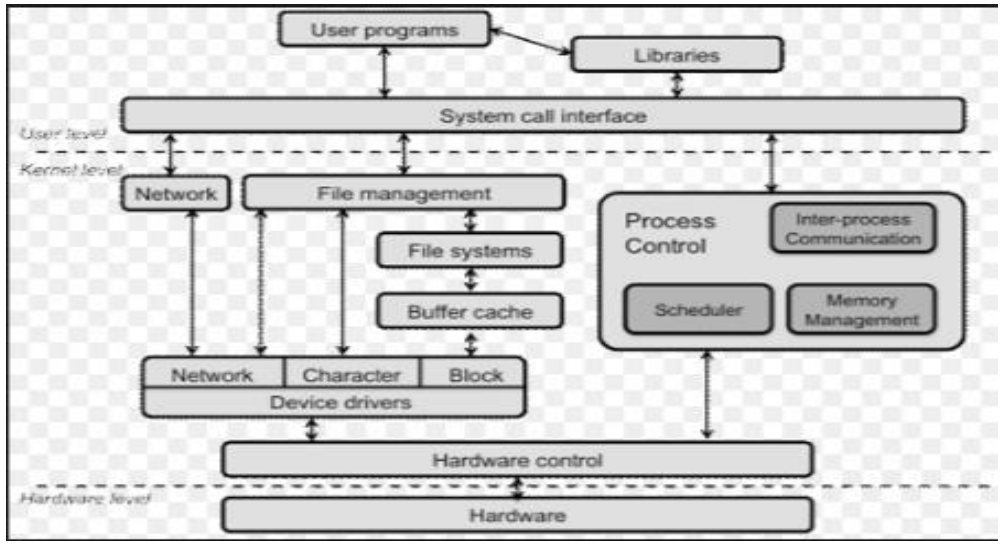
(Source: https://devowen.com/252)

Appendix

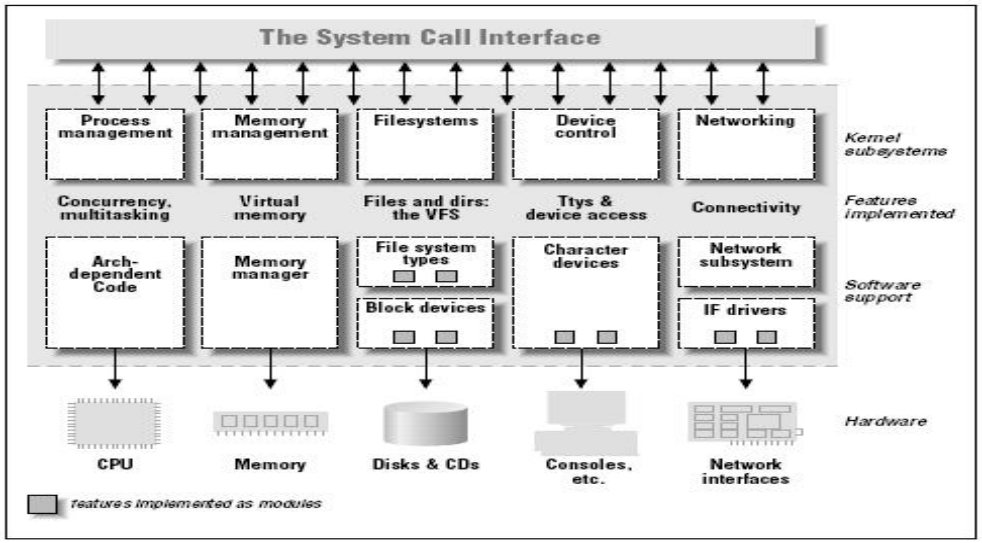
- OS structure in General



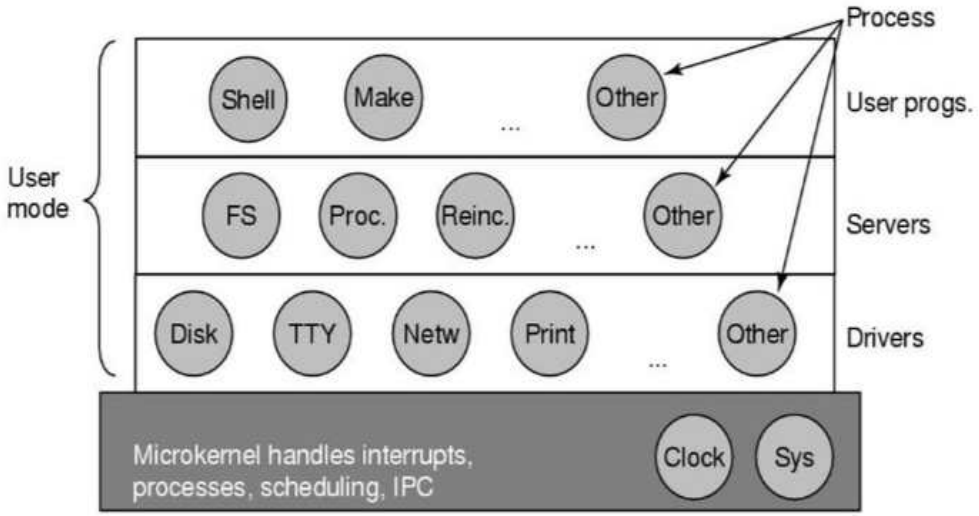
(Source: Operating System Concepts)



(Source: https://www.cs.rutgers.edu/~pxk/416/notes/03-concepts.html)



(Source: Linux Device Driver)



(Source: Modern Operating System)

사사

- 본 교재는 2025년도 과학기술정보통신부 및 정보통신기획평가원의 ‘SW중심대학사업’ 지원을 받아 제작 되었습니다.
- 본 결과물의 내용을 전재할 수 없으며, 인용(재사용)할 때에는 반드시 과학기술정보통신부와 정보통신기획평가원이 지원한 ‘SW중심대학’의 결과물이라는 출처를 밝혀야 합니다.