Lecture Note 1: OS Introduction

March 1, 2020
Jongmoo Choi

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

Contents

From Chap 1~2 of the OSTEP

Chap 1. A Dialogue on the Book

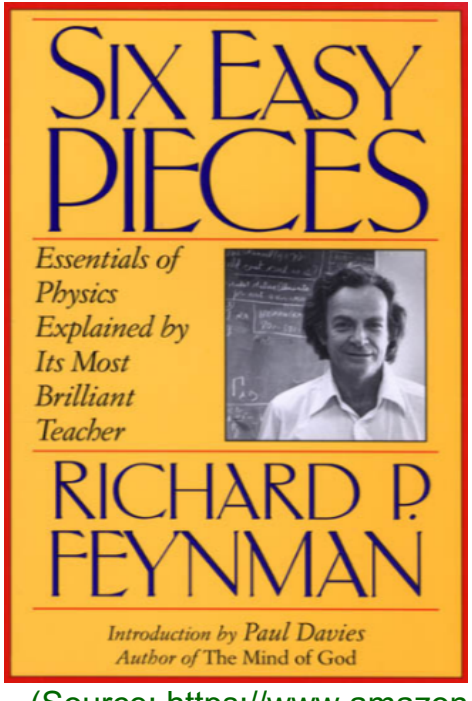
Chap 2. Introduction to Operating System

- ✓ Virtualizing the CPU
- ✓ Virtualizing Memory
- ✓ Concurrency
- ✓ Persistence
- ✓ Design Goals
- ✓ Some History
- ✓ References

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

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...*

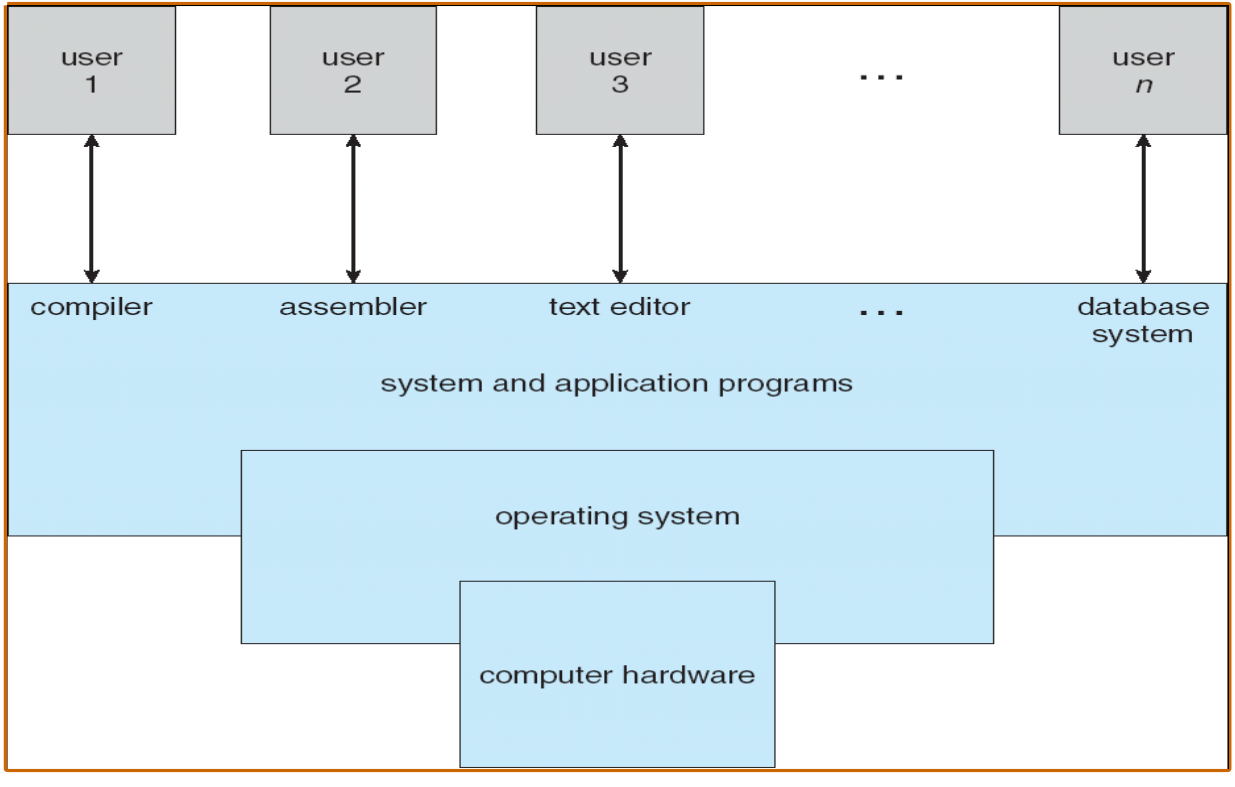
Student: *Oh, I know! 'I hear and I forget. I see and I remember. I do and I understand.' Or something like that.*

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

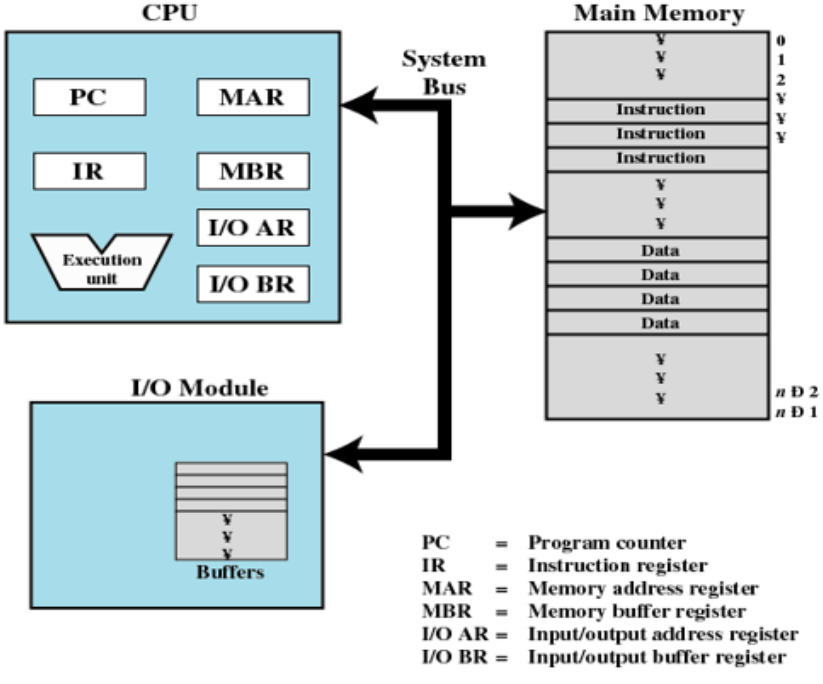


Figure 1.1 Computer Components: Top-Level View

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

Introduction

What happens when a program runs?

- ✓ 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

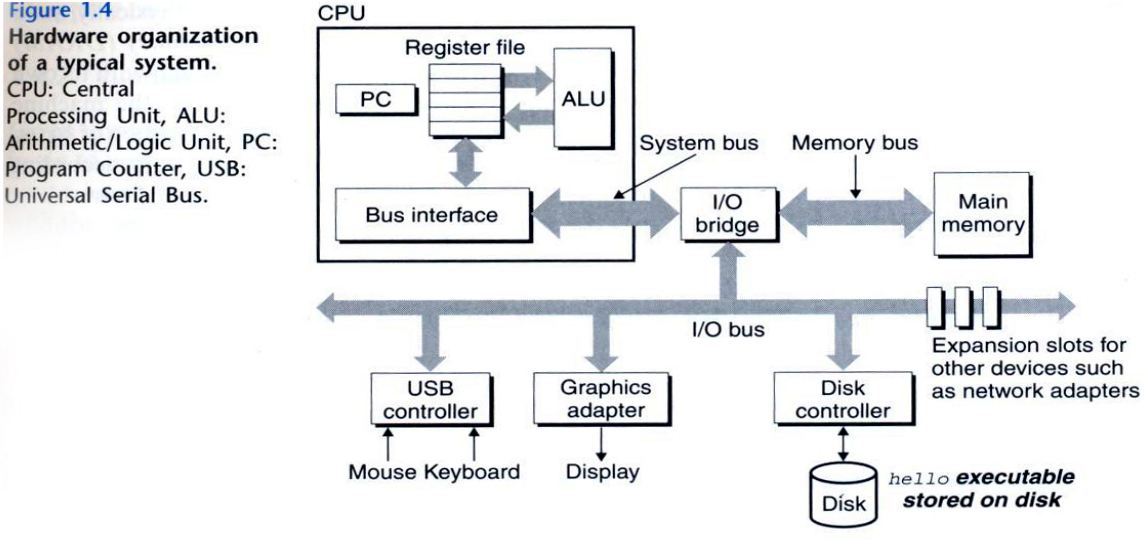


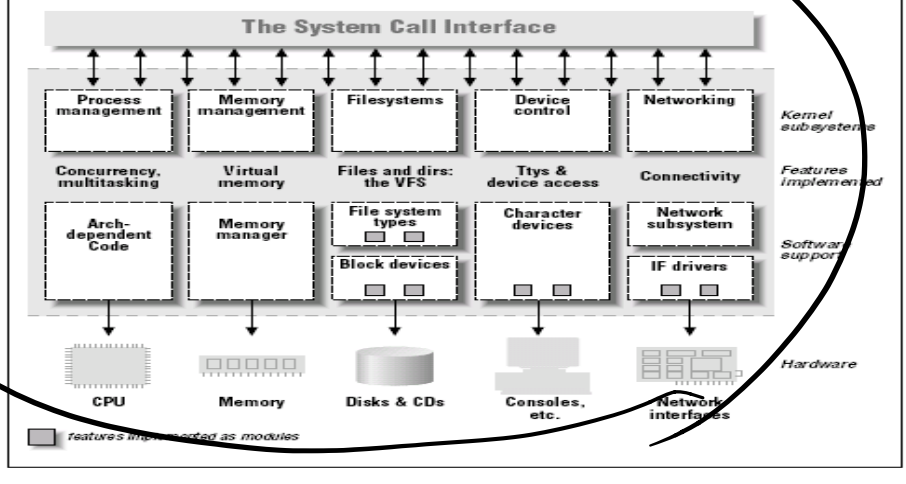
Figure 1.4 **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program Counter, USB: Universal Serial Bus.

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

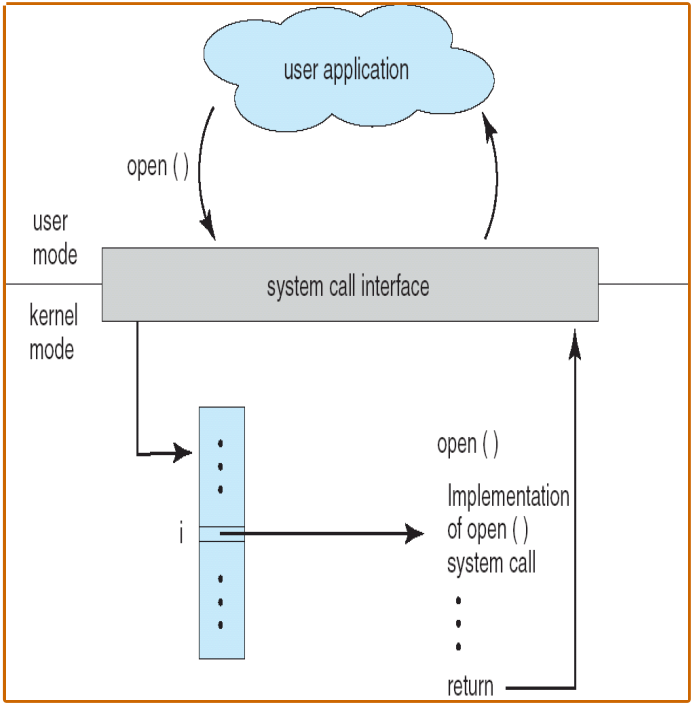
```
#include <unistd.h>

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

```
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

Exec

```
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
```

prog 1

```
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? ? 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
- Any questions? Send an email to me: choijm@dankook.ac.kr
- Simple question to take attendance: Explain why the title of our main text include "three easy pieces"? (Hint: see page 3 in this slide, until 6 PM, March 19th)

? 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 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()
{
      int 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 note)

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)

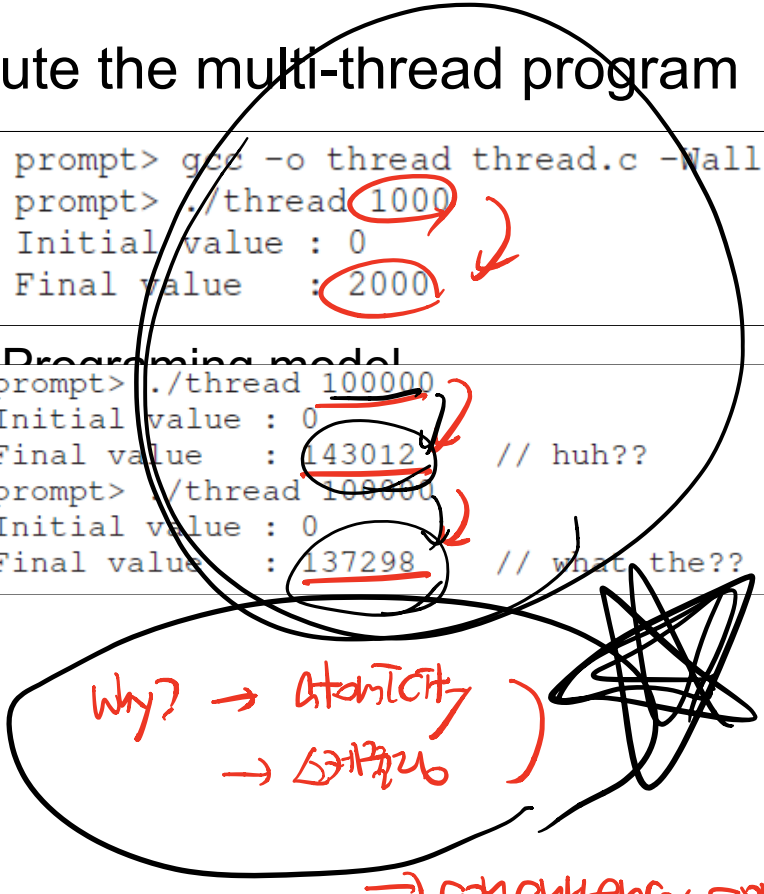
2.3 Concurrency

Execute the multi-thread program

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

- ✓ Programing model
- ✓ ge loop
  ontrol

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



⇒ concurrency control이 필요하다.

2.3 Concurrency

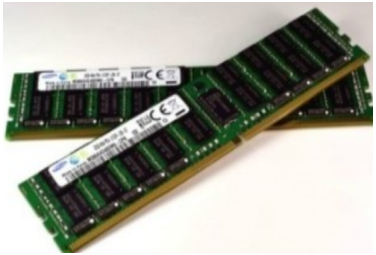
Issues for Concurrency

- How to support concurrency correctly? lock(), semaphore()
- How to implement atomicity in hardware? test_and_set(), swap()
- What is the semaphore?
- What is the monitor?
- How to solve the traditional concurrent problems such as producer-consumer, readers-writers and dining philosophers?
- 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

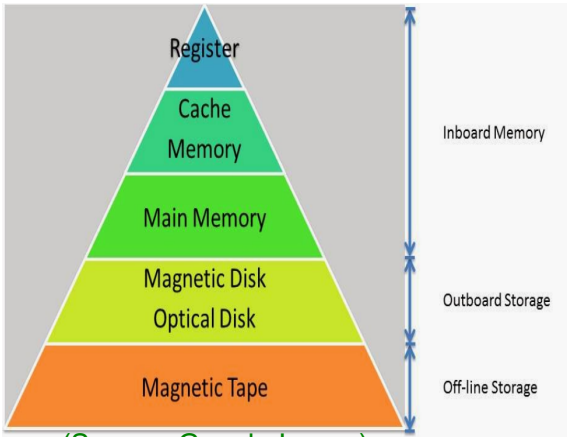
2.4 Persistence

Background: DRAM vs. Disk



VS

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



(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
8   main(int argc, char *argv[])
9   {
10      int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, 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?
- 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

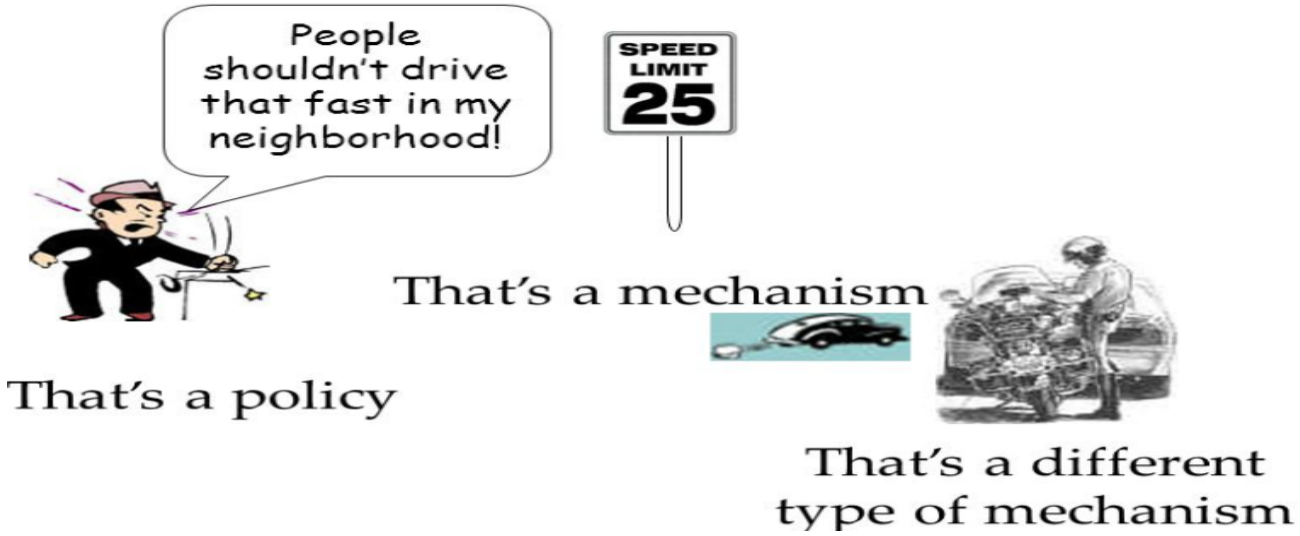
Others

- ✓ 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 scheduling 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

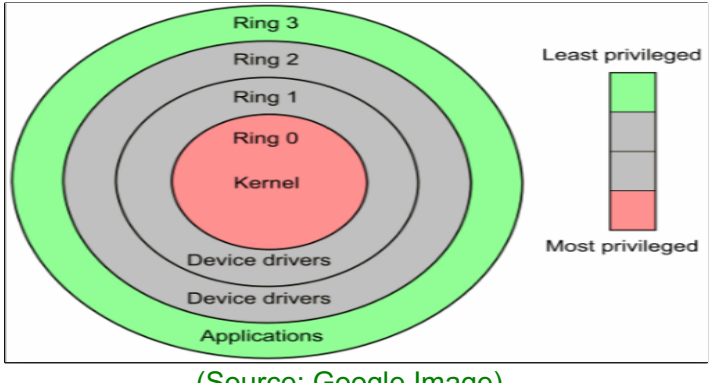
- ✓ Commonly-used functions such as low-level I/Os (e.g. MS-DOS)
- ✓ Batch processing

  a number of jobs were set up and then run all together (Not interactive)

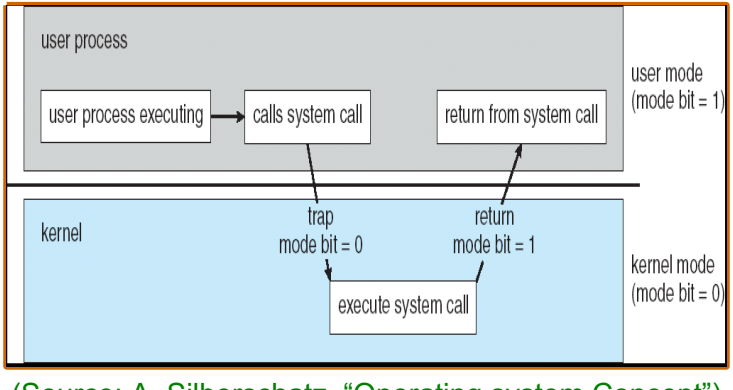
Beyond Libraries: Protection

- ✓ Require OS to be treated differently than user applications
- ✓ Separation user/kernel mode, system call
- ✓ Use trap (special hardware instruction) to switch into the kernel mode

  Transfer control to a pre-specific trap handler (system_call handler)



(Source: Google Image)

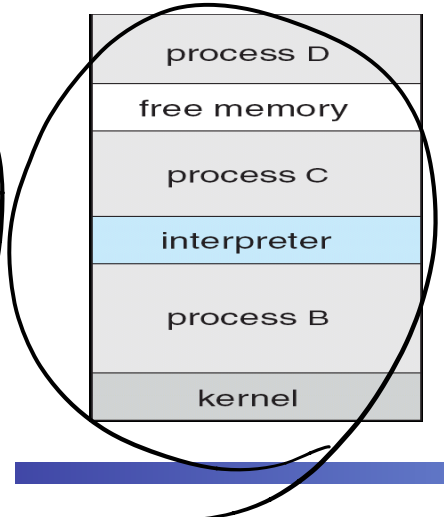


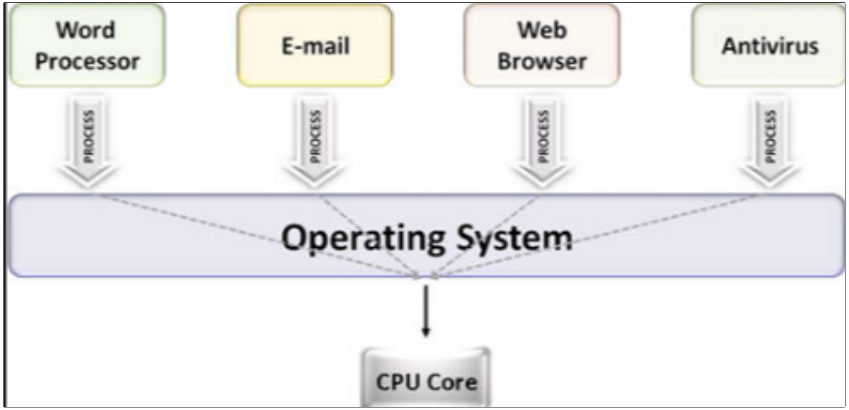
(Source: A. Silberschatz, "Operating system Concept")

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 (workstation, minicomputer) → multiprogramming
- ✓ 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





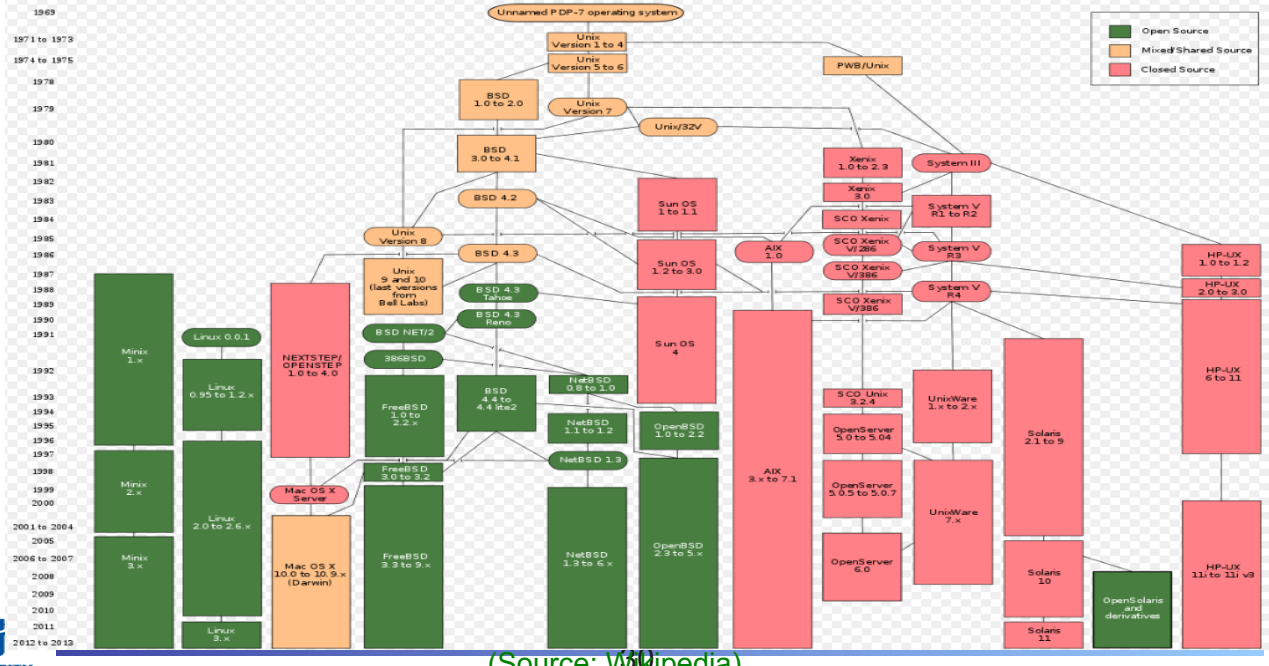
(Source: Google Image)

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, …
  Influence OSes such as BSD, SUNOS, AIX, HPUX, Nextstep and Linux



(Source: Wikipedia)

2.6 Some History

The Modern Era

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

2.7 Summary

OS
- ✓ Resource manager (Efficiency)
- ✓ Make systems easy to use (Convenience)

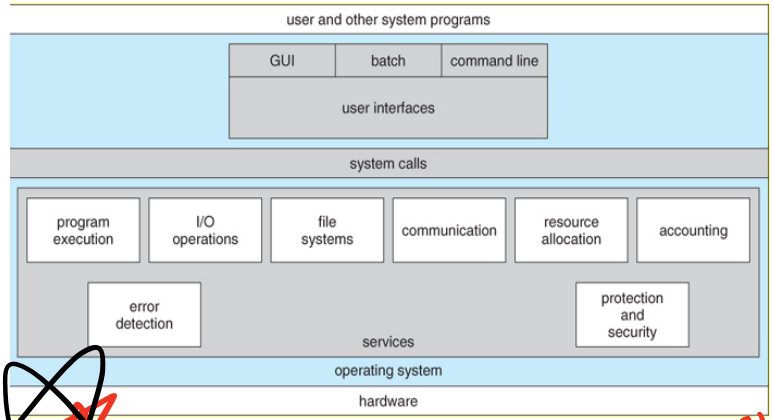
Cover in this book
- ✓ Virtualization, Concurrency, Persistence

Not being covered
- ✓ Network, Security, Graphics
- ✓ There are several excellent courses for them

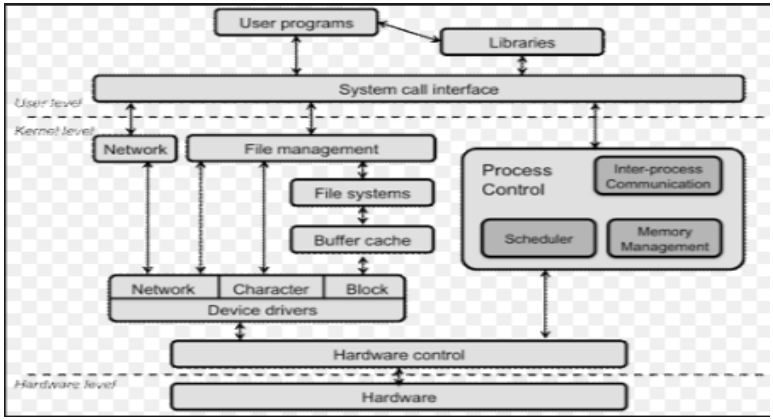
- Homework 1: summarize the chap 2 of the OSTEP
  - Requirement: 1) personal, 2) 3~5 pages for summary, 3) 1 page for the goal you want to study
  - Due: until the same day of the next week
  - Bonus: Snapshot of the results of example programs in a Linux system
- Any questions? Send an email to me: choijm@dankook.ac.kr
- Simple question to take attendance: Explain the differences between interrupt and trap which was discussed in page 28 (until 6 PM, March 24th)

Appendix

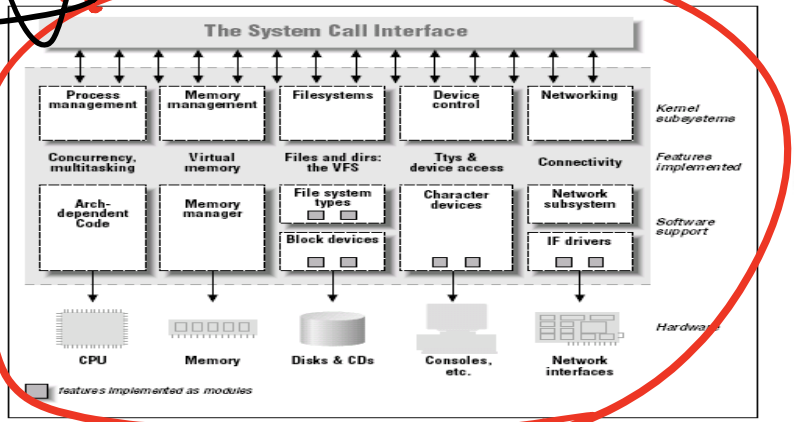
OS structure in General



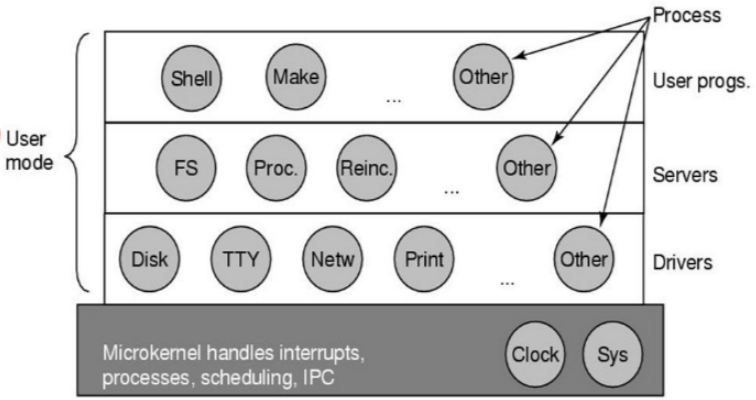
(Source: Operating System Concepts)



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



(Source: Linux Device Driver)



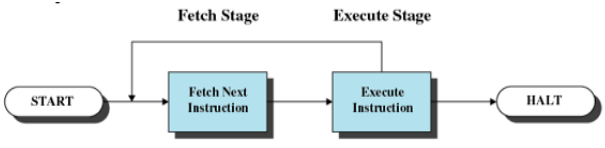
(Source: Modern Operating System)

Appendix

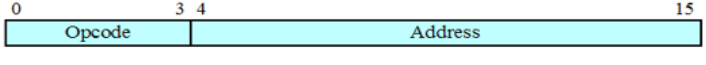
What happens when a program runs?

- ✓ Details: execute instructions

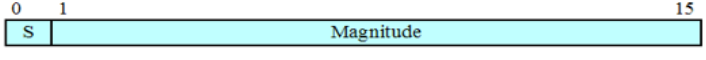
Fetch and Execute



<Instruction cycle>



(a) Instruction format



(b) Integer format

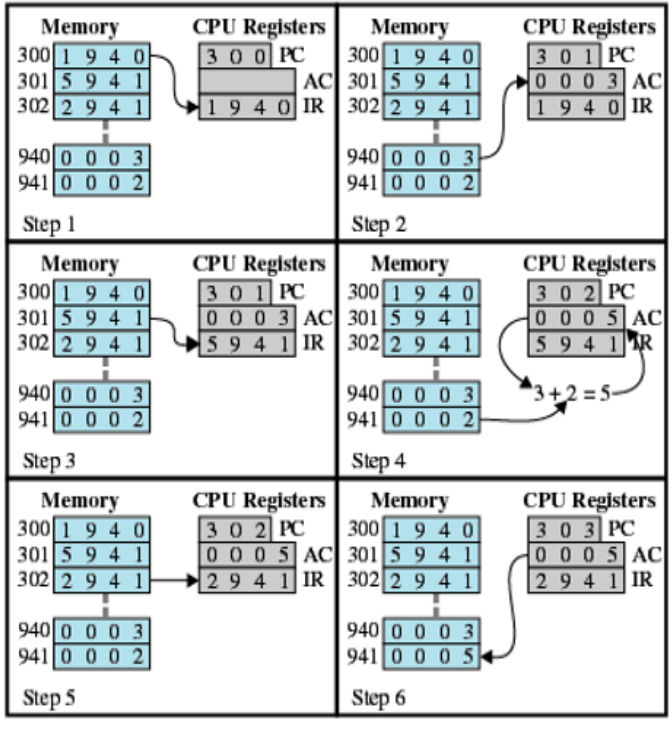
Program Counter (PC) = Address of instruction
Instruction Register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(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")