운영체제
4190.307

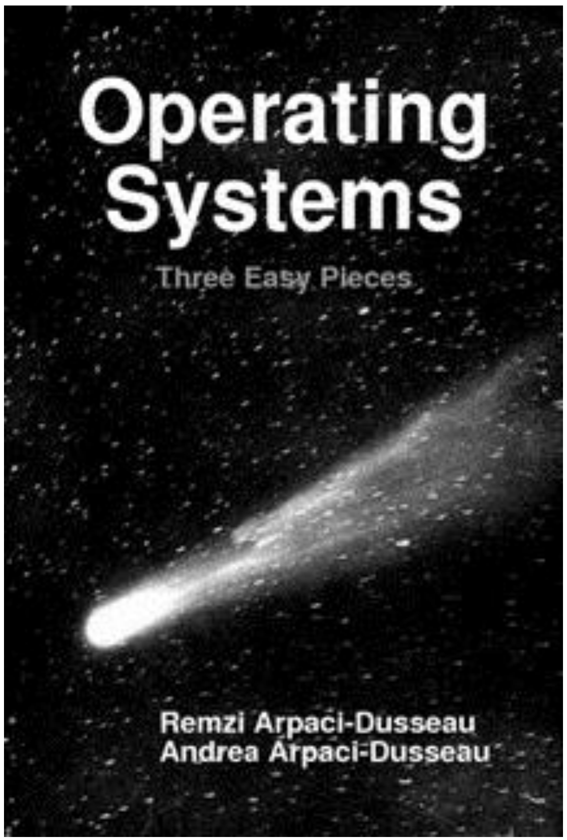
2017.2학기

월/수 14:00-15:15

Prerequites..

- C Programming
- 컴퓨터구조
- 시스템 프로그래밍
- 자료구조

Textbook



Operating Systems

Three Easy Pieces

by

Remzi Arpaci-Dusseau

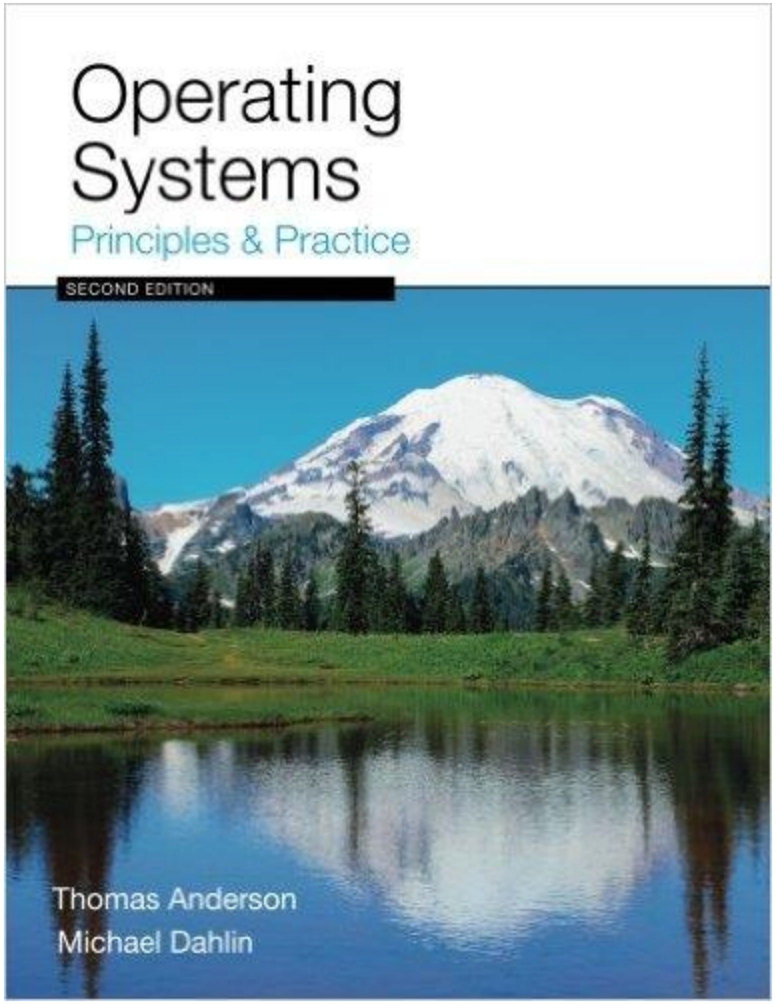
Andrea Arpaci-Dusseau

PDF file available from

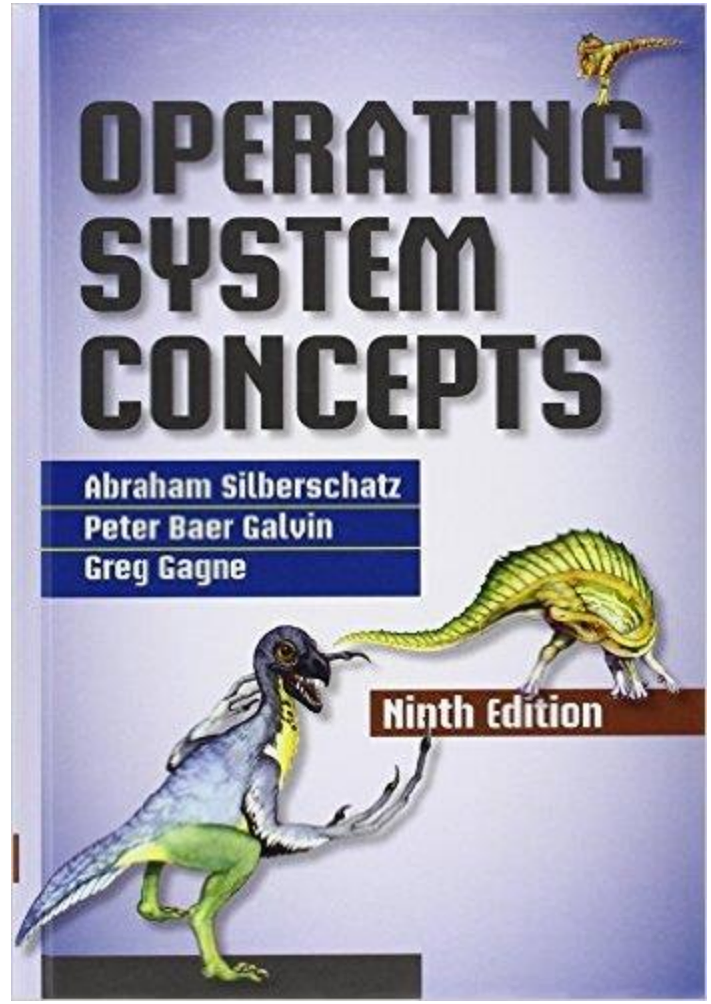
http://pages.cs.wisc.edu/~remzi/OSTEP/

Or Hardcopy (come to 302-319-1)

References

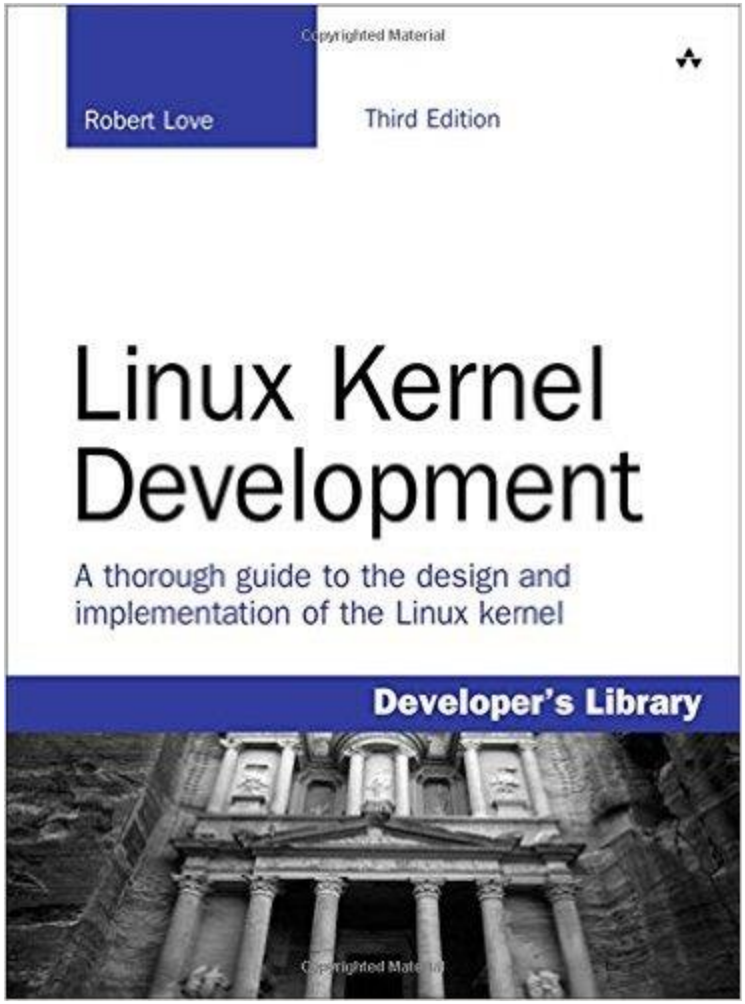


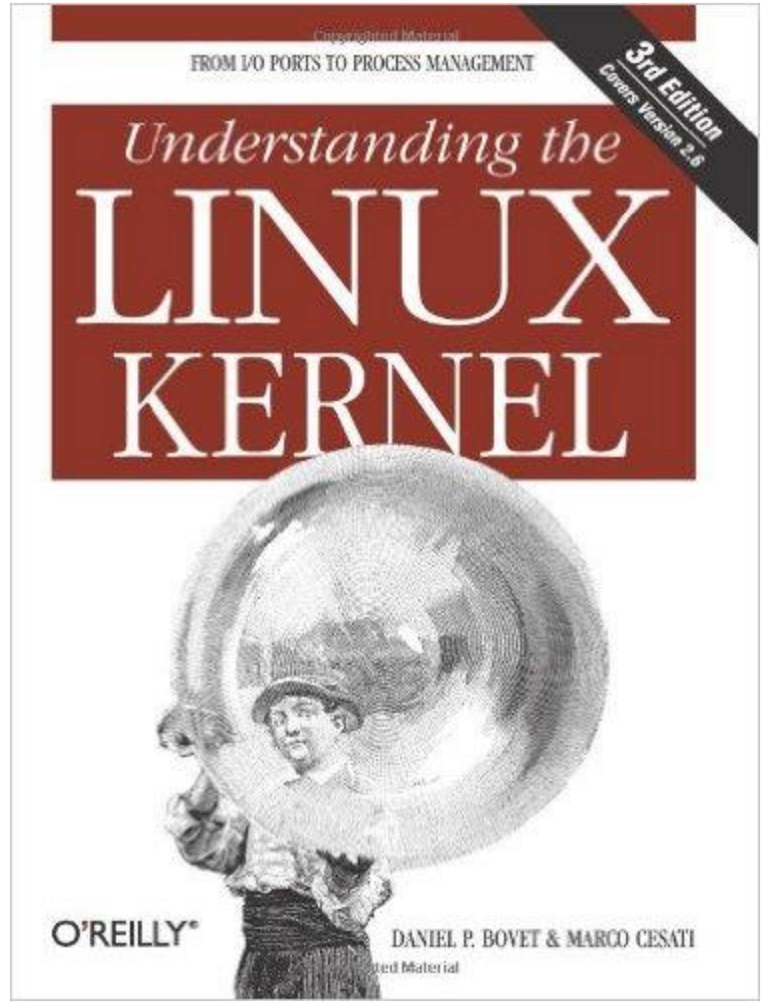
Operating Systems
Principles & Practice
SECOND EDITION
Thomas Anderson
Michael Dahlin



OPERATING SYSTEM CONCEPTS
Abraham Silberschatz
Peter Baer Galvin
Greg Gagne
Ninth Edition

References


Copyrighted Material
Robert Love
Third Edition
Linux Kernel Development
A thorough guide to the design and implementation of the Linux kernel
Developer's Library


FROM I/O PORTS TO PROCESS MANAGEMENT
3rd Edition
Covers Version 2.6
Understanding the LINUX KERNEL
O'REILLY
DANIEL P. BOVET & MARCO CESATI

Teaching Staff

- Instructor : Yeom, Heonyoung (염헌영)
  - Email : yeom AT snu.ac.kr
  - Office : 302-321
  - Office Hours : M W 13-14pm

- TAs
  - Park, Jiwoong (박지웅) pjbear AT gmail.com
  - Min, Cheol Gi (민철기) m47341 AT gmail.com
  - Office : 302-319-1

Course Components

- Lectures
- Programming Projects using Artik-10 board + Tizen 3.0
  - Provide in-depth understanding of the internals of OS
  - Learning by doing !
  - Could be done in 2-3 person teams
- Midterm Exam (10/23)
- Final Exam (12/13)
- No class on 10/4(추석) classes on 10/2, 10/9
- Lab(302-310-2) sessions : to be determined…

When you need help

- Class eTL
  - Lecture slides
  - Q & A

Class Rules

- No open laptops
  - Unless you are asked to do something…
- No cell phones in my sight
  - Better have them mute if they are in your possesion.
- Use common senses !
- No more begging for better grades !
  - I have to report the incidents to the president !!!

Questions answered in this lecture:

What will you do in this course?

What is an OS and why do you want one?

Why study operating systems?

What happens when a program runs?

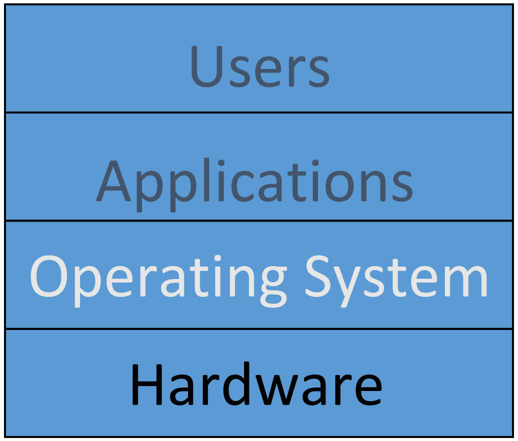
A running program executes instructions.

1. The processor **fetches** an instruction from memory.
2. **Decode**: Figure out which instruction this is
3. **Execute**: i.e., add two numbers, access memory, check a condition, jump to function, and so forth.
4. The processor moves on to the **next instruction** and so on.

Operating System (OS)

Responsible for

- Making it easy to **run** programs
- Allowing programs to **share** memory
- Enabling programs to **interact** with devices



OS

- Software that converts hardware into a useful form for applications
- in charge of making sure the system operates correctly and efficiently

What DOES OS Provide?

Role #1: Abstraction - Provide standard library for resources

What is a resource?

Anything valuable (e.g., CPU, memory, disk)

What abstraction does modern OS typically provide for each resource?

- CPU: process and/or thread
- Memory: address space
- Disk: files

Advantages of OS providing abstraction?

- Allow applications to reuse common facilities
- Make different devices look the same
- Provide higher-level or more useful functionality

Challenges

- What are the correct abstractions?
- How much of hardware should be exposed?

What DOES OS Provide?

Role #2: Resource management – Share resources well

Advantages of OS providing resource management?

- Protect applications from one another
- Provide efficient access to resources (cost, time, energy)
- Provide fair access to resources

Challenges

- What are the correct mechanisms?
- What are the correct policies?

OS Organization

How to cover all the topics relevant to operating systems?

OS Organization

How to cover all the topics relevant to operating systems?

Virtualization

Concurrency

persistency

Three PIECES: FIRST

Virtualization

Make each application believe it has each resource to itself

Virtualization

The OS takes a physical resource(Processor, Memory, Disk…) and transforms it into a virtual form of itself.

- The virtual form is more general, powerful and easy-to-use. Sometimes, we refer to the OS as a **virtual machine**.

System call

System call allows user **to tell the OS what to do**.

The OS provides some interface (APIs, standard library).

A typical OS exports a few hundred system calls.

- Run programs
- Access memory
- Access devices

The OS is a resource manager

The OS **manage resources** such as

CPU, *memory* and *disk*.

The OS allows

- Many programs to run → Sharing the <u>CPU</u>
- Many programs to *concurrently* access their own instructions and data → Sharing <u>memory</u>
- Many programs to access devices → Sharing <u>disks</u>

Virtualizing the CPU

The system has a very large number of virtual CPUs.

- Turning a single CPU into a seemingly infinite number of CPUs.
- Allowing many programs to seemingly run at once
  → **Virtualizing the CPU**

Virtualizing the CPU (Cont.)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.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) {
                Spin(1); // Repeatedly checks the time and
                         returns once it has run for a second
                printf("%s\n", str);
        }
        return 0;
}
```

Simple Example(`cpu.c`): Code That Loops and Prints

Virtualizing the CPU (Cont.)

Execution result 1.

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

Run forever; Only by pressing “Control-c” can we halt the program

Virtualizing the CPU (Cont.)

Execution result 2.

```
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, all four of programs seem to be running at the same time!

Virtualizing Memory

The physical memory is *an array of bytes*.

A program keeps all of its data structures in memory.

Read memory (load):

Specify an address to be able to access the data

Write memory (store):

Specify the data to be written to the given address

Virtualizing Memory (Cont.)

A program that Accesses Memory (`mem.c`)

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

int
main(int argc, char *argv[])
{
        int *p = malloc(sizeof(int));  // a1: allocate some memory
        assert(p != NULL);
        printf("(%d) address of p: %08x\n",
                getpid(), (unsigned) p);  // a2: print out the address of the memory
        *p = 0;  // a3: put zero into the first slot of the memory
        while (1) {
                Spin(1);
                *p = *p + 1;
                printf("(%d) p: %d\n", getpid(), *p); // a4
        }
        return 0;
}
```

Virtualizing Memory (Cont.)

The output of the program `mem.c`

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

The newly allocated memory is at address `00200000`.
It updates the value and prints out the result.

Virtualizing Memory (Cont.)

Running `mem.c` multiple times

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
...
```

It is as if each running program has its **own private memory**.

- Each running program has allocated memory at <u>the same address</u>.
- Each seems to be updating the value at `00200000` independently.

Virtualizing Memory (Cont.)

Each process accesses its own private **virtual address space**.

- The OS maps address space onto the physical memory.
- A memory reference within one running program does not affect the address space of other processes.
- Physical memory is a shared resource, managed by the OS.

Three PIECES: SECOND

Concurrency:

How can you handle all the different things at once ?

Three PIECES: SECOND

Concurrency:

- Events are occurring simultaneously and may interact with one another
- OS must be able to handle concurrent events

Easier case

Hide concurrency from **independent** processes

Trickier case

Manage concurrency with **interacting** processes

- Provide abstractions (locks, semaphores, condition variables, shared memory, critical sections) to processes
- Ensure processes do not deadlock

Demo

Interacting threads must coordinate access to shared data

The problem of Concurrency

The OS is juggling many things at once, first running one process, then another, and so forth.

Modern multi-threaded programs also exhibit the concurrency problem.

Concurrency Example

A Multi-threaded Program (`thread.c`)

```
#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++;
        }
        return NULL;
}
...
```

Concurrency Example (Cont.)

```
int
main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: threads <value>\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;
}
```

The main program creates **two threads**.

- Thread: a function running within the same memory space. Each thread start running in a routine called `worker()`.
- `worker()`: increments a counter

Concurrency Example (Cont.)

`loops` determines how many times each of the two workers will **increment the shared counter** in a loop.

- `loops`: 1000.

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

- `loops`: 100000.

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

Why is this happening?

Increment a shared counter → take three instructions.

1. Load the value of the counter from memory into register.
2. Increment it
3. Store it back into memory

These three instructions do not execute atomically.
→ Problem of **concurrency** happen.

Three PIECES: Third

Persistence:

The data should be there even after you turn off the computer !

Three PIECES: Third

Persistence: Access information permanently

- Lifetime of information is longer than lifetime of any one process
- Machine may be rebooted, machine may lose power or crash unexpecte dly

Issues:

- Provide abstraction so applications do not know how data is stored : Fil es, directories (folders), links
- Correctness with unexpected failures
- Performance: disks are very slow; many optimizations needed!

Demo

File system does work to ensure data updated correctly

Persistence

Devices such as DRAM store values in a volatile.

Hardware and *software* are needed to store data persistently.

- **Hardware**: I/O device such as a hard drive, solid-state drives(SSDs)
- **Software**:
  - File system manages the disk.
  - File system is responsible for storing any files the user creates.

Persistence (Cont.)

Create a file (`/tmp/file`) that contains the string "hello world"

```c
#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;
}
```

`open()`, `write()`, and `close()` system calls are routed to the part of OS called the file system, which handles the requests

Persistence (Cont.)

What does OS do in order to write to disk?

- Figure out **where** on disk this new data will reside
- **Issue I/O** requests to the underlying storage device

File system handles system crashes during write.

- **Journaling** or **copy-on-write**
- Carefully <u>ordering</u> writes to disk

Design Goals

Build up **abstraction**

- Make the system convenient and easy to use.

Provide high **performance**

- Minimize the overhead of the OS.
- OS must strive to provide virtualization without excessive overhead.

Protection between applications

- Isolation: Bad behavior of one does not harm other and the OS itself.

Design Goals (Cont.)

High degree of **reliability**

- The OS must also run non-stop.

Other issues

- Energy-efficiency
- Security
- Mobility

Advanced Topics

Current systems

- Multiprocessors
- Networked and distributed systems
- Virtual machines

Why study Operating Systems ?

Build, modify, or administer an operating system

Understand system performance

- Behavior of OS impacts entire machine
- Tune workload performance
- Apply knowledge across many layers

  Computer architecture, programming languages, data structures and algorithms, and performance modeling

Fun and challenging to understand large, complex systems

Disclaimer: This lecture slide set was made by merging/modifying/revising below three.

1) Andrea’s at University of Wisconsin.
2) Tyler’s at University of Wisconsin
3) Youzip’s Operating System course in Computer Science Dept. at Hanyang University