Introduction (ch. 2) part 1

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

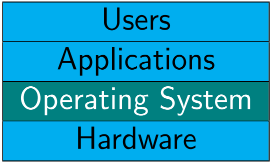
Tel-Aviv Academic College

What is an Operating System?

Not a simple question.

What is an Operating System?

Not a simple question.



Users

Applications

Operating System

Hardware

- Middleware between user programs and hardware
- Abstracts and manages resources
  - CPU
  - Main memory
  - I/O Devices (disk, network card, mouse, keyboard, monitor, etc.)

Why study Operating Systems?

- We study Computer **Science**
  - Not a programming course...
- You use an operating system
  - The machine is (mostly) useless without an OS
  - Understand what you use
  - Why and how is the OS useful?
- Behavior of OS impacts entire machine
  - Understand system performance
  - Useful to know how computers work

Approach

- The course is about ideas and analysis
- We will not build an OS
    - But there will be coding

CPU wise, What happens when a program runs?

- Executes instructions
- The processor:
  - **Fetches** an instruction from memory
  - **Decodes** the instruction
  - **Executes** it
  - Moves on to the **next instruction**
- **Von Neumann** model of computing

What does the OS do?

- Abstraction
  - Virtual resources that correspond to hardware resources
  - Well-defined operations on these resources
    - CPU → Running program (process / thread)
    - Memory → Address space / virtual memory
    - Storage → Files, file system

What does the OS do?

- Resource Management
    - Share resources among running programs
    - Decide who gets how much and when
        - CPU → Who runs next?
        - Memory → Where is data in RAM and when to access it?
        - Storage → Where and how are files stored on disk?

Three Easy Pieces

- **Virtualization**
  - As if each program has resources to itself
- **Concurrency**
  - Juggling many things at once
- **Persistence**
  - Ability to store data beyond termination / computer shutdown

Virtualizing the CPU

`cpu.c`:

```c
int main(int argc, char *argv[])
{
    while (1) {
        spin(1); // returns after 1 second
        printf("%s\n", argv[1]);
    }
}
```

- The program loops and prints

Virtualizing the CPU

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

- Runs forever
- Halt program by pressing “Control-C”

Virtualizing the CPU

```
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
...
```

- We have **one** processor
- All four seem to be running **at the same time**

Virtualizing the CPU

- OS illusion
  - Single CPU → infinite number of virtual CPUs
  - Programs seemingly run at once

Virtualizing the CPU

- OS illusion
  - Single CPU → infinite number of virtual CPUs
  - Programs seemingly run at once
- Under the hood:
  - Multiple instances are started
  - OS picks one to run (use CPU)
  - After a while, OS kicks it off the CPU
  - Picks another one to run

Virtualizing the CPU

- **Context Switch**
    - Running program pauses, another is brought in
    - Program does not know when it is context-switched (in or out)
        - Illusion that it is alone on the CPU
        - Fetch-Decode-Execute cycle continues
    - Fast and frequent
        - Appears to be running at the same time

Virtualizing Memory

- Physical memory (**RAM**) - array of bytes
  - **Read** (load) - specify address to access data
  - **Write** (store) - also specify data to write
- Program code (instructions) is also in memory!

Virtualizing Memory

mem.c:

```c
int main(int argc, char *argv[])
{
    int p;
    printf("(%d) the address of p: %p\n",
        getpid(), &p);
    p = atoi(argv[1]);
    while (1) {
        spin(1);
        p = p + 1;
        printf("(%d) p: %d\n", getpid(), p);
    }
}
```

Virtualizing Memory

```
prompt> ./mem 0
(2134) the address of p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
^C
prompt>
```

- Increments and prints every second

Virtualizing Memory

```
prompt> ./mem 0 & ./mem 100 &
[1] 13526
[2] 13527
(13527) the address of p: 0x200000
(13526) the address of p: 0x200000
(13527) p: 101
(13526) p: 1
(13527) p: 102
(13526) p: 2
(13527) p: 103
(13526) p: 3
...
```

- Same address, different value
- As if each instance as its own private memory

Virtualizing Memory

- Program address is <u>not</u> physical address
    - It is a **virtual address**
- Each process accesses its own **virtual address space**
    - The OS (with hardware help) maps it onto the physical memory
    - Reference in one running program does not affect the other

Virtualizing Memory

- Program address is not physical address
  - It is a **virtual address**
- Each process accesses its own **virtual address space**
  - The OS (with hardware help) maps it onto the physical memory
  - Reference in one running program does not affect the other

- Each program seemingly has all physical memory to itself
  - No knowledge (or responsibility) of other programs
  - **Memory protection**