Operating Systems

Lecture 1

Course Overview

Who am I?

Associate Professor

School of Cybersecurity, Korea Univ.

Research Interests

System Security on OS and Hypervisor-based Virtualization
CPU Microarchitecture Security
Security Analysis of Carrier-grade Network Devices
Cloud System Security

Education

Ph.D., Computer Science, KAIST
M.S., Computer Science, KAIST
B.S., Computer Science and Engineering, Korea University

Work Experience

2020.9 ~ Current : Korea University
2017.3 ~ 2020.8 : Kwangwoon University
2008.4 ~ 2017.2 : National Security Research Institute (NSR)
2007.11 ~ 2008.3 : LG Electronics, Digital Media Lab

Honors & Awards

Official Commendation from the Director of National Intelligence Service (NIS) of Korea, 2012
Best Researcher Award by National Security Research Institute (NSR), 2011
Best Paper Awards by ACM Q2SWinet (2007), National Cryptography Contest (2008, 2019), CISC-W (2019), etc

Class Goals

- Understand the basic concepts of operating systems
  - *designing* & *building* operating systems, not *using* them!
- Gain some practical experience so that it is not just words!

Expectations

Reading textbook *before* class

Active participation in class discussions

No cheating!

The contract:

- You try to learn!
- I try to teach you!

Grading

Participation – F will be given if the attendance rate is less than 70%

Exams

- Mid-term - 30%
- Final - 30%

Attendance (Attitude) – 10%

Assignments – 30%

Text books

“Operating Systems: Three Easy Pieces” by Remzi Arpaci-Dusseau

Useful Links

Class web site

Introduction to Operating Systems

1. What is an Operating System?
2. Review of OS-Related Hardware

What is an Operating System?

"A *program* ... that controls the execution of application programs and implements an interface between the user of a computer and the computer hardware"

Runs on PCs, workstations, servers, smart phones, routers, embedded systems, etc

Operating System Roles

- Abstract Machine
  - Hides complex details of the underlying hardware
  - Provides common API to applications and services
  - Simplifies application writing
- Resource Manager
  - Controls accesses to *shared* resources
    - CPU, memory, disks, network, ...
  - Allows for global policies to be implemented

The Abstract Machine Role

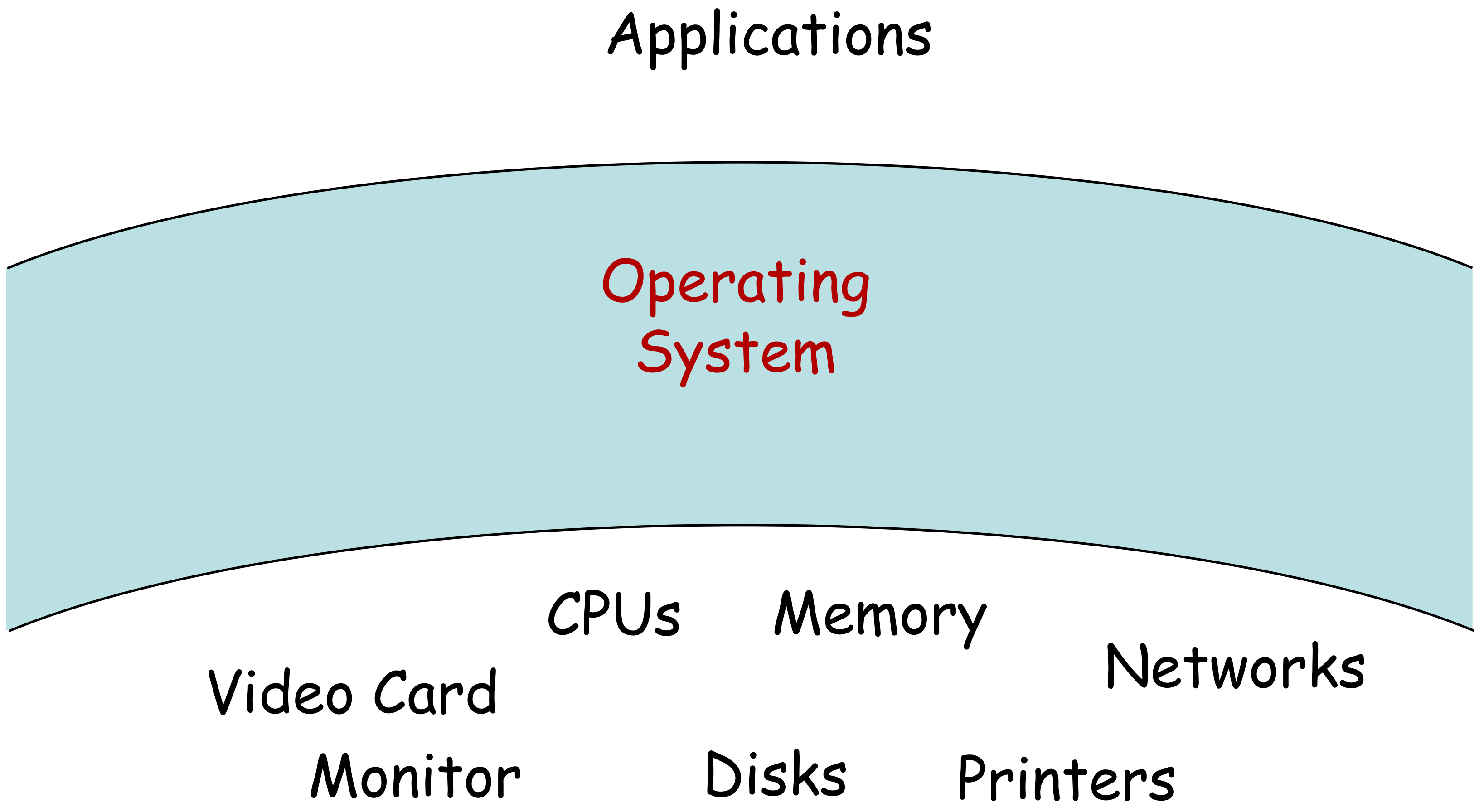
Without operating systems, application writers would have to program all device access directly:

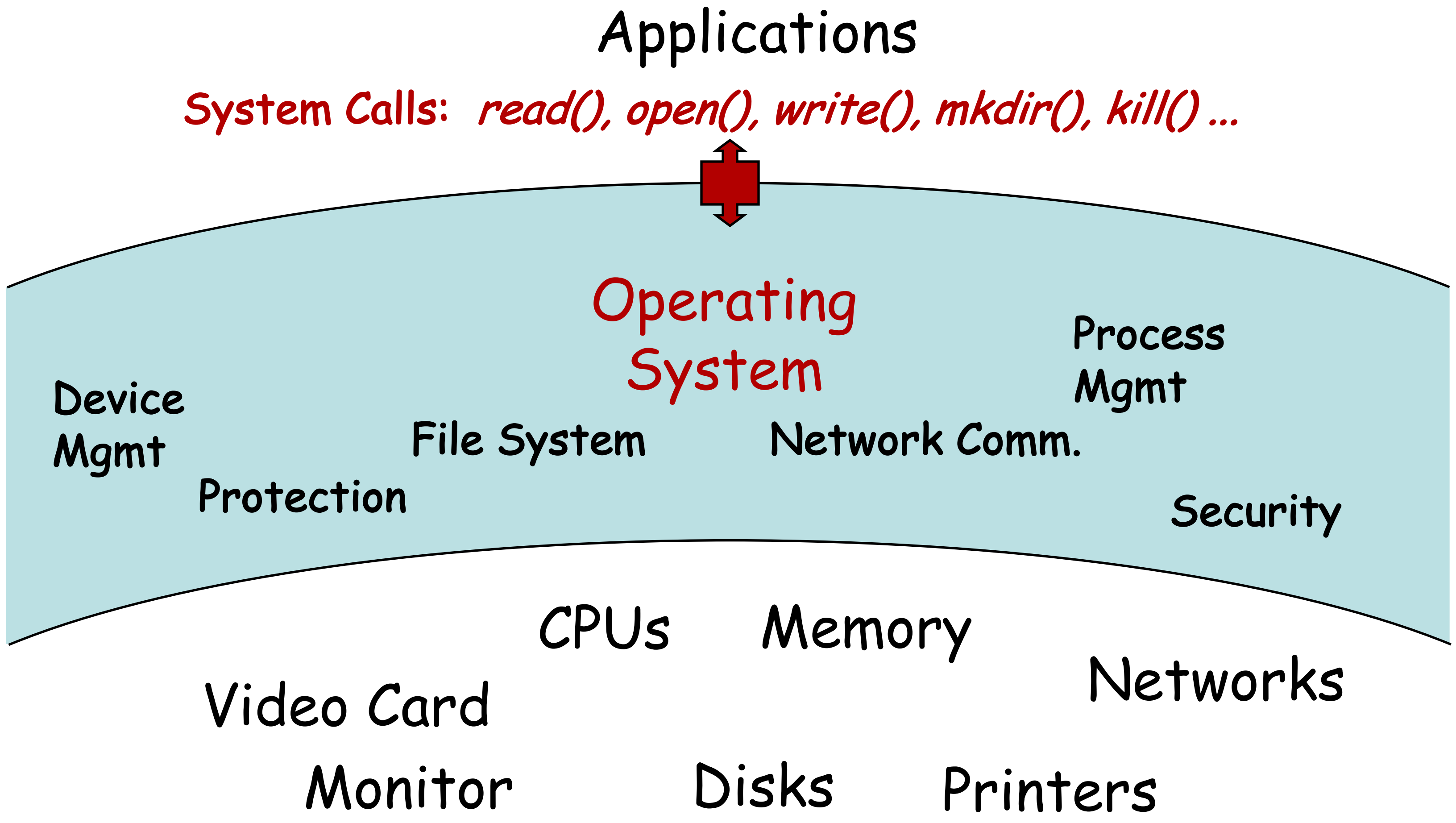
- Load device command codes into device registers
- Understand physical characteristics and data layout
- Interpret return codes
- ...

Application programming would be complicated

Applications would be difficult to maintain, upgrade and port

- This OS code could be written just once and then shared!


Applications
Operating System
CPUs
Memory
Video Card
Networks
Monitor
Disks
Printers


Applications
System Calls: read(), open(), write(), mkdir(), kill() ...
Operating System
Device Mgmt
Protection
File System
Network Comm.
Process Mgmt
Security
CPUs
Memory
Networks
Video Card
Monitor
Disks
Printers

The Resource Manager Role

Allocating resources to applications
- time sharing resources
- space sharing resources

Making efficient use of limited resources
- improving utilization
- minimizing overhead
- improving throughput/good put

Protecting applications from each other

Resources to Allocate

Time sharing the CPU

Space sharing the memory

Space sharing the disk

Time sharing the network

Problems Solved by OS

Time sharing the CPU among applications

Space sharing the memory among applications

Space sharing the disk among users

Time sharing access to the network

More Problems Solved by OS

Protection of applications from each other, of user data from other users and of I/O devices

Protection of the OS itself!

Prevention of direct access to hardware, where this would cause problems

But the OS is just a program! How can it do all this?

OS Needs Help from Hardware

The OS is just a program!

When it is not running, it can't do anything!

Its goal is to run applications, not itself!

The OS needs help from the hardware in order to detect and prevent certain activities, and to maintain control

Brief Review of Hardware

Instruction sets define all that a CPU can do, and differ among CPU architectures

All have load and store instructions to move data between memory and registers

Many instructions for comparing and combining values in registers

The OS is Just a Program!

The OS is just a sequence of instructions that the CPU will fetch/decode/execute

How can the OS cause application programs to run?

How can applications cause the OS to run?

How Can an OS Run Applications?

The OS must load the address of the application's starting instruction into the PC (Program Counter)

Example:

- computer boots and begins running the OS
- OS code must get into memory somehow
- fetch/decode/execute OS instructions
- OS requests user input to identify application program/file
- OS loads application (executable file) into memory
- OS loads the address of the app's first instruction into the PC
- CPU fetches/decodes/executes the application's instructions

The OS is Just a Program!

How does the OS ever get to run again?

How can the OS switch the CPU to run a new application (and later resume the first one)?

How can the OS maintain control of what the application does when the OS is not running?

In what ways can application try to seize control indefinitely (ie. cheat)?

How can the OS prevent such cheating?

How Can the OS Regain Control?

What if an application doesn't call the OS and instead just hogs the CPU?

- OS needs *interrupts* from a timer device!
- OS must register a future timer interrupt before handing control of the CPU over to an application
- When the timer interrupt goes off the hardware starts running the OS at a pre-specified location called an interrupt handler
- The interrupt handler is part of the OS program
- The address of the interrupt handler's first instruction is placed in the PC by the h/w

Can the Application Cheat?

Can the application disable the future timer interrupt so that the OS can not take control back from it?

Disabling interrupts must be a privileged instruction that is not executable by applications

The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the PSW!

Privileged instructions can only be executed when the mode bit is set

- eg. disabling interrupts, setting the mode bit!
- attempted execution in non-privileged mode generally causes an interrupt (trap) to occur

Are There Other Ways to Cheat?

What stops the running application from modifying the OS?

- eg. modifying the timer interrupt handler to jump control back to the application?

What Stops Applications From Modifying the OS?

Memory protection!

Memory protection instructions must be privileged

- i.e., they can only be executed with the mode bit set …

Why must the OS clear the mode bit before it hands control to an application?

How Can Applications Invoke the OS?

Why not just set PC to an OS instruction address and transfer control that way?

How Can Applications Invoke the OS?

Special trap instruction causes a kind of interrupt

- changes PC to point to a predetermined OS entry point instruction
- simultaneously sets the mode bit
- CPU is now running in privileged mode

Application calls a library procedure that includes the appropriate trap instruction

fetch/decode/execute cycle begins at a pre-specified OS entry point called a system call handler

Are Traps Interrupts?

Traps, like interrupts, are hardware events

But traps are synchronous where as interrupts are asynchronous

i.e. traps are caused by the executing program rather than a device external to the CPU

Switching to a New Application?

To suspend execution of an application the OS must run!

After that, simply

- capture the application's memory state and processor state
- preserve all the memory values of this application
- copy values of all CPU registers into a data structure which is saved in memory
- restarting the application from the same point just requires reloading the register values

Q and A