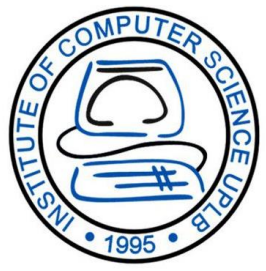
CMSC 125: Operating Systems

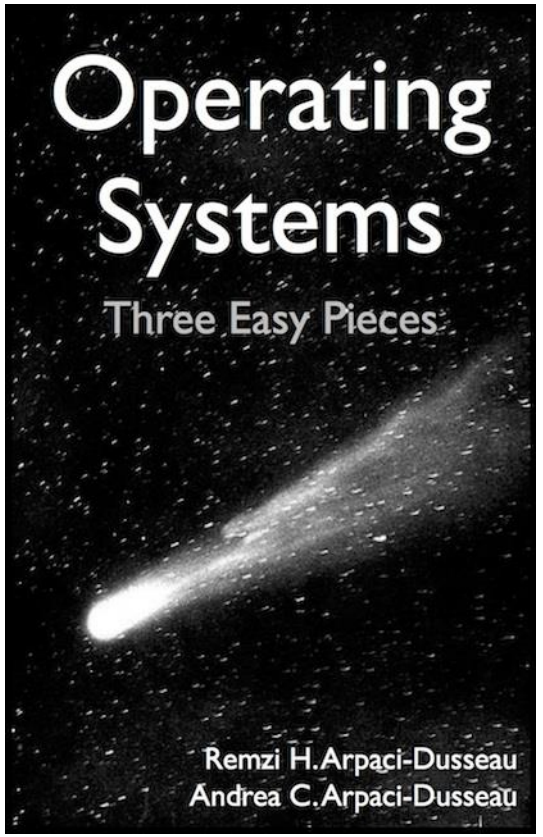
- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org



Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

6. Mechanism: Limited Direct Execution

Operating System: Three Easy Pieces

How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing/multiprogramming.
- Issue
  - **Performance**: How can we implement virtualization without adding excessive overhead to the system?
  - **Control**: How can we run processes efficiently while retaining control over the CPU?
- Will require both **hardware** and **OS** support

Basic Technique: Direct Execution

- Just run the program directly on the CPU.

OS	Program
1. Create entry for **process list** 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` /`argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

Without *limits* on running programs,
the OS wouldn't be in control of anything and
thus would be "just a library"

Problem 1: How to perform restricted operations?

- What if a process wishes to perform some restricted operations such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources, such as CPU or memory
- However, if we just let the process do what it wants, we will be limited in constructing desirable features like security and protection

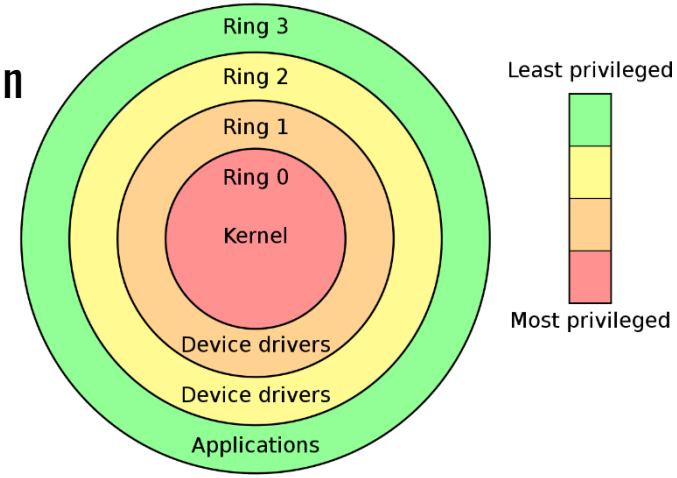
- **Solution**: Using **protected control transfer** by introducing two modes of operation (needs hardware level support)
  - User mode: Applications do not have full/direct access to hardware resources – **user process**
  - Kernel mode: The OS has access to the full resources of the machine - **kernel**
- How then can a user process perform privileged operations like I/O?
  - Answer: The **kernel** performs the operation on behalf of the **user process** via a **system call**

System Call

- Allow the kernel to carefully expose certain key pieces of functionality to a process, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory
- Will need **privileged hardware instructions** because we cannot just use the ordinary function/procedure call mechanism

System Call (Cont.): Privileged Instructions

- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode
  - Examples: In x86 it is usually the `INT` instruction, in x86-64 it is the `SYSCALL` instruction



https://en.wikipedia.org/wiki/Protection_ring#/media/File:Priv_rings.svg

- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode
  - Examples: In x86 it is usually the `IRET` instruction, in x86-64 it is the `SYSRET` instruction
- **Initialize trap table** – a support instruction/s

System Call (Cont.)

- Enough information must be saved by the hardware when executing a **trap** in order to be able to return correctly to the process that invoked the system call
  - PC
  - Flags register
  - Other important registers
- The information is pushed to a per-process **kernel stack** before the **trap**
- **Return-from-trap** will pop the information back to the registers

http://it.uplbosa.org/2014/04/study-hub-hunt-in-elbi.html

System Call (Cont.): Trap Table

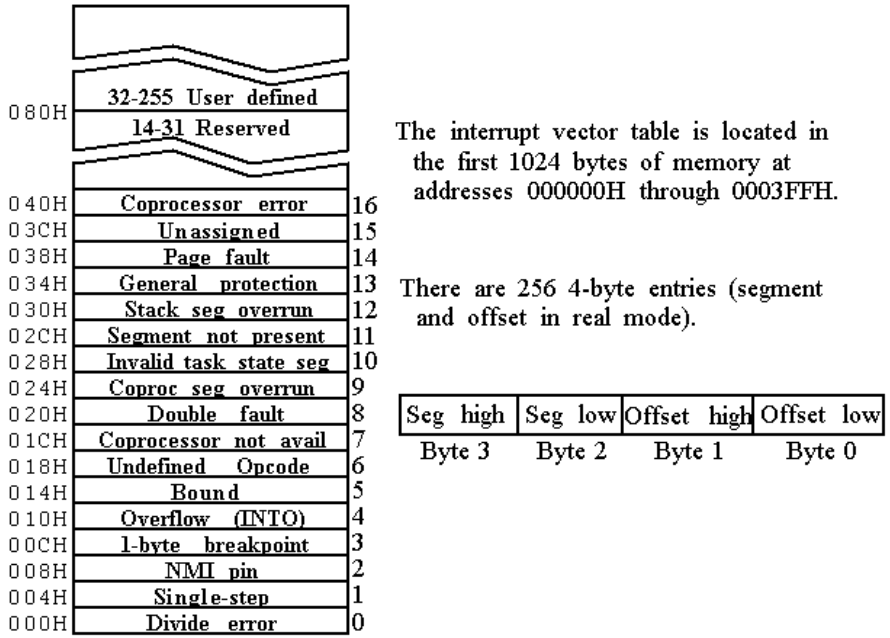
- A user process cannot specify the address to jump to in a system call unlike a typical function/procedure call – why?
- How then does the **trap** knows which code to execute in the kernel (note that we are now in kernel mode)?

Aside: Interrupt Mechanism in Hardware

- If everything is going well, the CPU will just happily execute instructions..
- But sometimes important events need the attention of the CPU
  - Examples: A user process divides by zero, I/O completion, a page fault happens, a system call is invoked, a timer has lapsed
  - Can be handled either through **polling** or **interrupts**(used in x86)
- Interrupts are identified by an **interrupt number**
- Interrupt classification in x86
  - **Interrupts –** asynchronous hardware events, ex: I/O completion or explicit invocation of `INT` instruction
  - **Exceptions –** when error conditions happen, there are three types:
    - **Traps –** reported immediately, process continues execution to the next instruction after trap, ex: a system call, `INT 3` (debugging)
    - **Faults –** reported immediately, process continues execution to the specified fault handler, ex: divide by zero, page fault
    - **Abort –** severe errors happen, hardware component failure (kernel panic, BSOD)

System Call (Cont.): Trap Table

- **Trap table** [aka Interrupt Vector Table(IVT), Interrupt Descriptor Table(IDT)]
  - Set up by the kernel at **boot time** – kernel has full privilege at this point
  - Kernel places in the trap table the memory address of the code (**trap handler**) to execute when an event (generated by interrupts and exceptions) occurs, identified by the **interrupt number**
  - The kernel will set the address of the code for the system call as the **system call handler** to one of the user-defined interrupt numbers (ex. Linux x86 uses `0x80`)
  - Exact/specific operations can be invoked by passing a **service number** in registers or stack – <u>thus using a number instead of address to prevent direct memory access</u>



https://ece-research.unm.edu/jimp/310/slides/8086_interrupts.html

Limited Direction Execution Protocol (LDE)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remembers address of ... ***system call handler***

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for **process list**		
Allocate memory for program		
Load program into memory		
Setup user stack with `argv/argc`		
Fill **kernel stack** with `regs/PC`		
return-from -trap		
	restore `regs/PC` (from **kernel stack**)	
	switch to user mode	
	jump to `main()`	
		Run `main()`
		...
		Call system call (**trap** into OS)

Limited Direction Execution Protocol (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save `regs/PC` to **kernel stack** switch to kernel mode jump to ***system call handler***	
Do work of ***system call handler*** (service specified by a number) **return-from-trap**		
	restore `regs/PC` (from **kernel stack**) switch to user mode jump to restored `PC`	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from **process list**		

Problem 2: Switching Between Processes

- When a user process is running on the CPU, this means that the kernel is not running! (assume we have a single CPU)
- How can the OS regain control of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making **system calls** such as `yield()`.
  - The OS decides to run some other task.
  - Application also transfer control (**trap**) to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - Ex) Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop?
→ Reboot the machine!

A Non-Cooperative Approach: OS Takes Control

A timer interrupt

- The **timer**(a hardware component) raises an **interrupt** every so many milliseconds (configurable)
  - In x86 this chip is the Programmable Interrupt Timer (PIT)
- At boot time, the kernel sets a **timer handler** for the timer interrupt then starts the **timer**
- When the timer interrupt is raised:
  - The currently running process is halted.
  - Save enough of the state of the halted process
  - The **timer handler** for the timer interrupt (set at boot time) runs code in the kernel, such invoking the **scheduler** to run a different process

A timer interrupt gives kernel the ability to run again on the CPU after a number of milliseconds have passed.

Kernel now in control: Saving and Restoring Context

- Scheduler makes a decision (depending on a scheduling policy):
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes a **context switch**.

Context Switch

- A low-level piece of assembly code(**context switch code**) in the kernel which
  1. **Saves a few register values** for the currently running process onto its **kernel stack** for the process
     1. General purpose registers
     2. PC
     3. kernel stack pointer of currently running process
  2. **Restores a few register** for the soon-to-be-executing process from its **kernel stack**
  3. **Switches to the kernel stack** for the soon-to-be-executing process
- By *switching stacks*, the kernel
  - enters the call to the **context switch code** in the context the interrupted process
  - returns in the context of the soon-to-be-executing process

Limited Direct Execution Protocol (with Timer Interrupt)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of ... ***system call handler*** ***timer handler***
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save `regs/PC(A)` to `k-stack(A)` switch to kernel mode jump to ***timer handler***	

Limited Direct Execution Protocol (with Timer Interrupt)

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
Do work of ***timer handler***		
Call `switch()` routine		
save `regs/PC(A)` to `proc-struct(A)`		
restore `regs/PC(B)` from `proc-struct(B)`		
switch to `k-stack(B)`		
return-from-trap (into B)		
	restore `regs/PC(B)` from `k-stack(B)`	
	switch to user mode	
	jump to restored `PC` (which is B's)	
		Process B
		…

The xv6 Context Switch Code

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7          # Save old registers
8          movl 4(%esp), %eax          # put old ptr into eax
9          popl 0(%eax)                # save the old IP
10         movl %esp, 4(%eax)          # and stack
11         movl %ebx, 8(%eax)          # and other registers
12         movl %ecx, 12(%eax)
13         movl %edx, 16(%eax)
14         movl %esi, 20(%eax)
15         movl %edi, 24(%eax)
16         movl %ebp, 28(%eax)
17
18         # Load new registers
19         movl 4(%esp), %eax          # put new ptr into eax
20         movl 28(%eax), %ebp         # restore other registers
21         movl 24(%eax), %edi
22         movl 20(%eax), %esi
23         movl 16(%eax), %edx
24         movl 12(%eax), %ecx
25         movl 8(%eax), %ebx
26         movl 4(%eax), %esp          # stack is switched here
27         pushl 0(%eax)               # return addr put in place
28         ret                         # finally return into new ctxt
```

What about Concurrency?

- What happens if, during interrupt or exception handling, another interrupt occurs?
- Kernel handles these situations by:
  - **Disabling interrupts** during interrupt processing
    - In x86, use the `CLI` and `STI` instructions
  - Use a number of sophisticated **locking** schemes to protect concurrent access to internal data structures.