Mechanism:
Limited Direct Execution

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Direct Execution

→프로세서 직접 하드웨어를 접근.

끝나면

운영체제는 이 모든 걸 책임지고 보호 메커니즘을 유지하면서 프로세스에게 하드웨어 접근 권한을 줌

- **The OS needs to share the physical CPU by time sharing, but there are two challenges to virtualize the CPU:**
  - Performance: how can we implement virtualization without adding excessive overhead to the system?

    과도한
  - Control: how can we run processes efficiently while retaining control over the CPU? – control is particularly important to the OS as it's in charge of resources
- **The direct execution gives rise to a few problems to virtualize**
  - How can the OS make sure the program doesn't do anything we don't want?
  - How does the OS stop it from running and switch to another process?

PCB

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	"무한루프 돌면...?"
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

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

Problem #1: Restricted Operations

CPU가 < user mode / kernel mode 를 구별해주어야 한다.

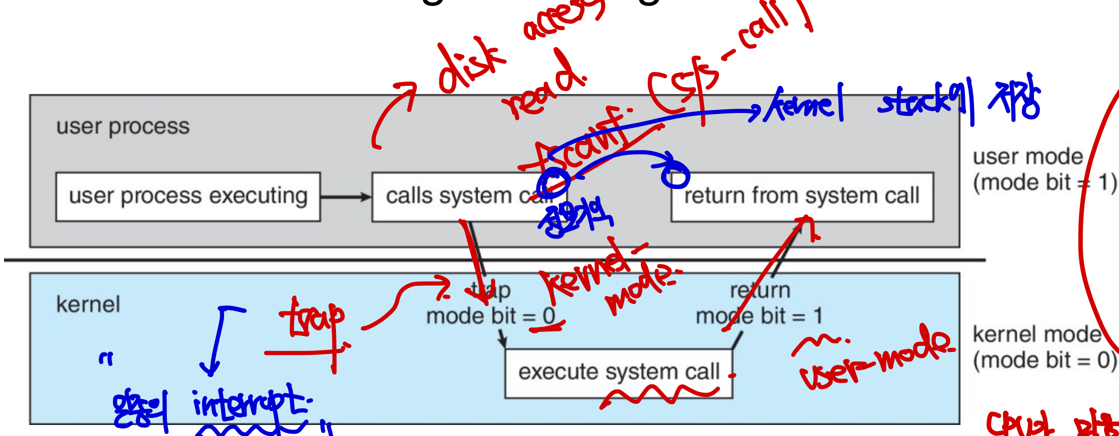
- **What if the process wishes to perform restricted operations?**
  - Issuing an I/O request to a disk, gaining access to more CPU and memory
  - Then, the process can access an entire disk and destroy important data → 파괴하면..?
  - Most of operations (arith, loop) can run directly but some should run indirectly
    - 산술
    - ↳ 운영체제를 통해서.
- **A new CPU mode, user mode, is introduced to restrict access**
  - → 제한을 줌.
  - Code that runs in user mode is restricted in what it can do (e.g. no I/O request)
  - Doing so results in the processor raising an exception and OS kills the process
    - ↳ 비정상
- **In contrast to user mode is kernel mode, which the OS runs in**
  - Code that runs can do what it likes including privileged operations (e.g. I/O)
- **Thus, the CPU must support at least two mode of operations**
  - 더 많을 수도
  - e.g.) 4 privilege levels in x86 and its level is set by current privilege level (CPL) in CS register
  - The privileged instructions can only be executed in the corresponding level; otherwise, the CPU raises an exception and the OS kills the process

System Call and Trap

- **What if a user program wants to perform privileged operations?**
  - For this, hardware provides the ability for user programs to perform a system call
  - System calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing file system, allocating more memory
- **For system call, a program must execute a special trap instruction**

OS로 제어권

  - This simultaneously jumps into the kernel and raises the privilege level to kernel mode; Then, the system can perform the privilege operations
  - When finished, the OS calls a special **return-from-trap** instruction that returns to the calling user program and reduces the privilege level to user mode



↔ hardware interrupt

- **Trap is known as software interrupt**
- **In x86,**
  "INT" is trap instruction and
  "IRET" is return-from-trap instruction

Trap Handling

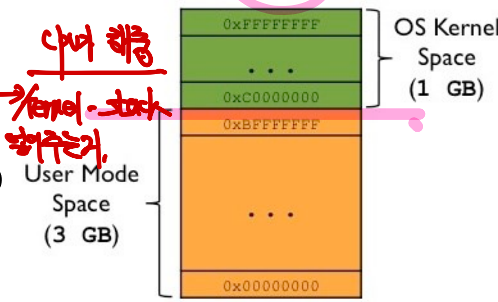
user의 cpu의 정보 trap : 찾다 찾자 overhead ↑

cpu는 1개! → user. kernel register override

P1 P2

- **The hardware needs to be careful to when executing a trap**
  - It must save caller's registers to be able to return correctly when the OS issues the return-from-trap instruction
  - In x86, the CPU pushes PC, flags, and some registers onto a per-process kernel stack; the return-from-trap pops these values and resumes user-mode program

cpu의 레지 kernel-stack에 넣어주는거



IDT

INT #	Description
0x00	Divide by 0
0x01	Reserved
0x02	NMI Interrupt
0x03	Breakpoint (INT3)
0x04	Overflow (INTO)
0x05	Bounds range exceeded (BOUND)
0x06	Invalid opcode (UD2)
0x07	Device not available (WAIT/FWAIT)
0x08	Double fault
0x09	Coprocessor segment overrun
0x0A	Invalid TSS
0x0B	Segment not present
0x0C	Stack-segment fault
0x0D	General protection fault
0x0E	Page fault
0x0F	Reserved
0x10	x87 FPU error
0x11	Alignment check
0x12	Machine check
0x13	SIMD Floating-Point Exception
0x14-0x1F	Reserved
0x20-0xFF	User definable

- **How does the trap know which code to run in OS?**
  - The kernel does so by setting up a trap table (at boot time)
  - When the machine boots up, it's in privileged (kernel) mode and the OS initializes the trap table ← trap table의 위치도 알아야 함
  - The OS informs the CPU of the locations of these trap handlers

trap table은 발생된 예외에 대한 handler의 위치가 저장됨 또 수행

- **A system-call number is usually assigned to each system call**
  - The OS, when handling the system call inside the trap handler, examines this number, ensures it is valid and executes the codes; a form of protection
  - Informing the CPU of trap table location (IDTR) is done by a privileged operation

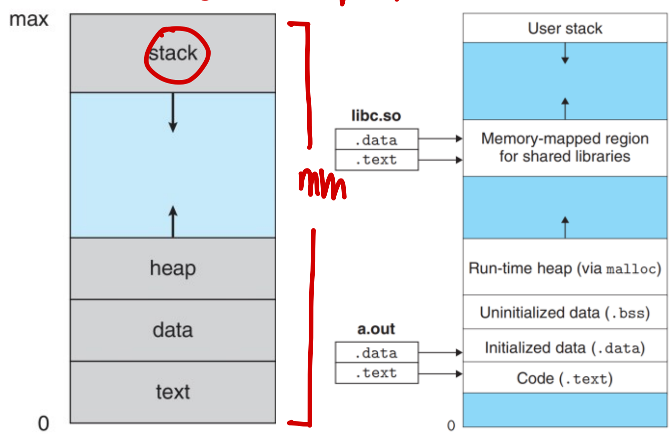
trap table의 위치

Where does the OS live?

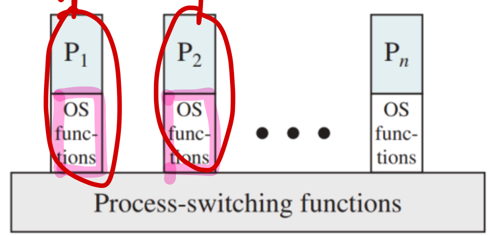
user
kernel
process
OS ..? where

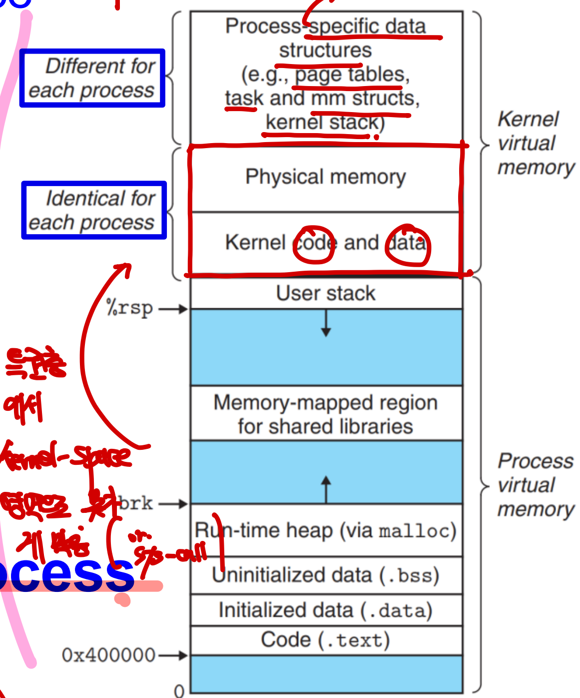
- **Memory Layout Revisited: The memory layout of a process is typically divided into four segments of text, data, heap, and stack**
  – Each process has its own private virtual address space

user process
kernel-stack



OS는 user process 안에 기생해서 삶



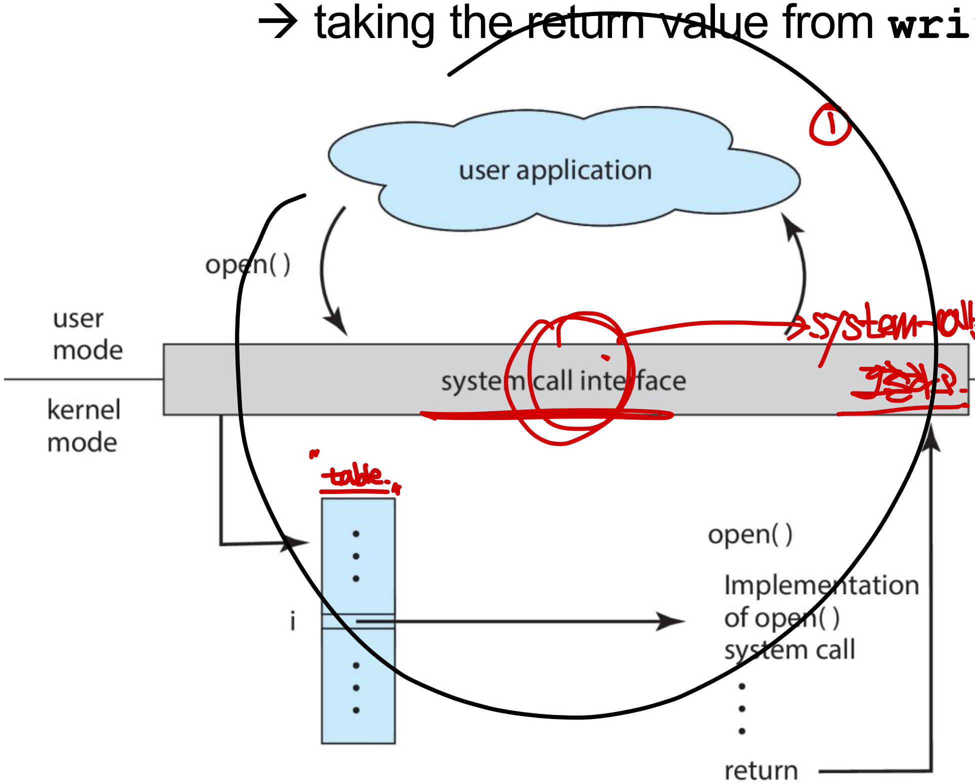


- **OS lives in the same address at the user process**
  – The other approach can be used according to the OS but modern OS is executed in context of user process
  – The privileged mode does not allow the user process to jump into the kernel space and prevent it from accessing the kernel space
  – User programs must access kernel code and data indirectly via the system call

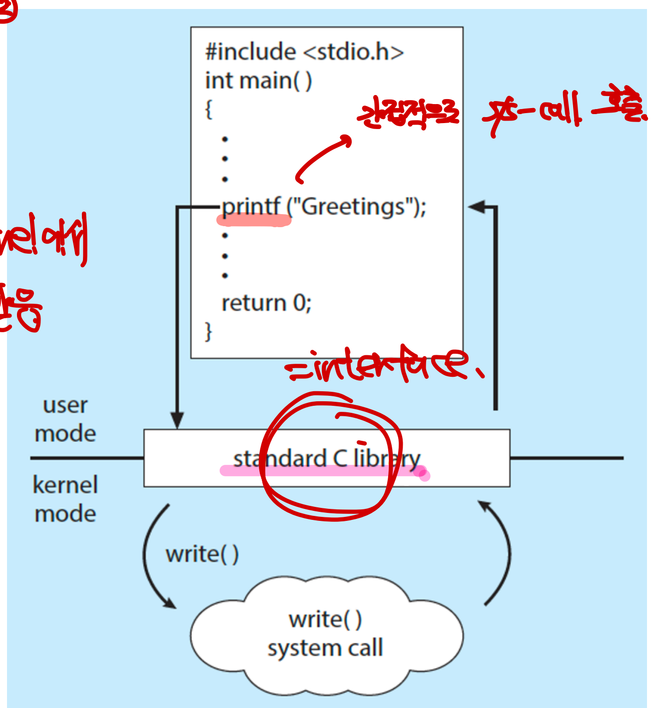
System Call Interface and Standard Library

- **System call interface maintains a table indexed to the numbers**
  - The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- **The standard library provides a portion of system call interface**
  - e.g.) invoking `printf()` → intercepting this call by lib → invoking syscall `write()` → taking the return value from `write()` → passing it back to user program

wrapping



②



Limited Direct Execution Protocol

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall 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 Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to PC after trap	
		... return from main **trap** (via exit())
Free memory of process Remove from process list		

- **Initializing when booting up**
- **Creating process**
- **Processing system call by trap**
  - system call → trap → save context and switch stack → jump to the trap handler → processing in kernel mode
  - return-from-trap → switch stack and restore context → jump to the next of the system call → continue in user mode
- **Destroying process**

OS

(kernel-mode)

HW

program.

initialize
trap-table

remember IDTR
trap table
address.

Create entry process
list.

Allocate memory for pc

Load program into memory

Setup user stack with argv

Fill kernel stack with
pc / regs
범용 레지스터

Return-from-trap

pop kernel-stack
restore regs
move to user-mode
jump to main

Run main()

Call sys-call
trap into OS

save regs into
kernel-stack

kernel-mode

jump to trap handler

handle trap

Do work for sys-call

return-from-trap

restore regs

user mode

jump to code after trap

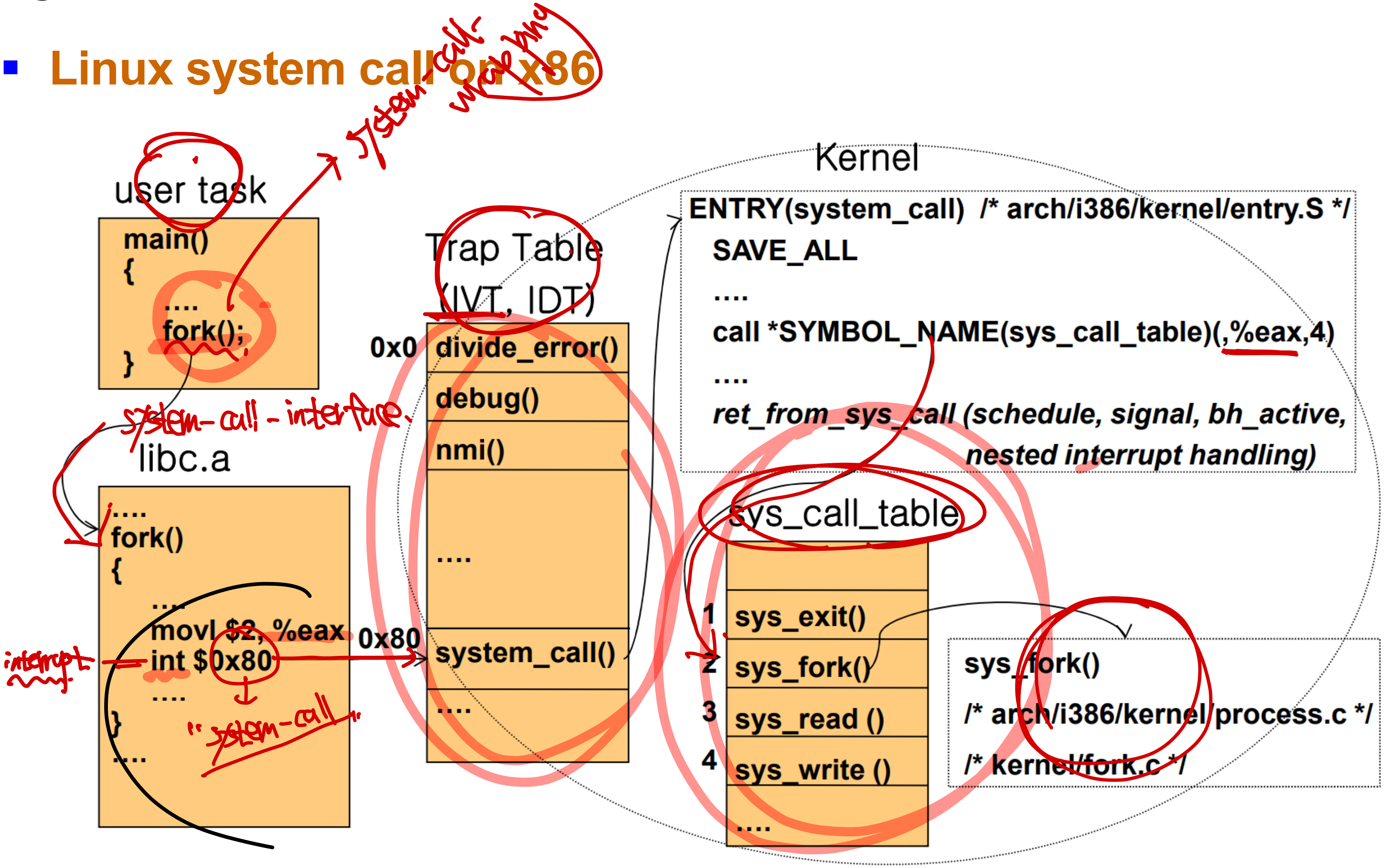
return from main

trap

loose

System Call Implementation

- Linux system call on x86



Courtesy of Prof. Jongmoo Choi @ Dankook Univ.

Problem #2: Switching between Processes

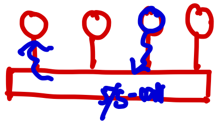
여러개를 조작

- **If a process is running on the CPU, then, the OS is not running**
  - Then, there is no way for the OS to do for switching between processes
  - How can the OS regain control of the CPU so it can switch between processes?

협조적

- **One way is a cooperative approach: wait for system calls** → 안쓰
  - Processes are assumed to periodically give up the CPU so the OS can decide to run some other task
  - Most processes transfer control of the CPU to the OS frequently by system calls
  - Application also transfer control to the OS when they do something illegal, such as dividing by zero, illegal memory access, which will cause a trap
  - yield system call is provided in case processes seldomly use a system call
  - Big issue: what if a process gets stuck in an infinite loop?➔ reboot the machine

- **In short, a cooperative approach makes thee steps**
  1) processes use a system call
  2) control transfer to the OS
  3) do scheduling (switching)



A Non-Cooperative Approach: The OS Takes Control

- **How can the OS gain control of the CPU without cooperation?**
  Hardware interrupt
  timer device 가 Hardware 에 있음
  - The answer is simple and was discovered long time ago: a timer interrupt
  - A timer device can be programmed to raise an interrupt every so many millisec.
  - When 1) the interrupt is raised, 2) the running process is halted and 3) a pre-configured interrupt handler in the OS runs ➔ the OS has regained control

- **The OS must inform the hardware of which code to run when the timer interrupt occurs**
  - At boot time, the OS do that and must start the timer by a privileged operation

- **The hardware has some responsibility when an interrupt occurs**
  - In particular to save enough of the state of the running process such that a subsequent return-from-trap instruction will be able to resume it correctly
  - This is similar to the behavior of the hardware during an explicit system-call trap into the kernel

Saving and Restoring Context

- **The scheduler, a part of OS, makes a decision whether**
  - To continue running the "currently-running process" OR switch to a different one
  - If the decision is made to "switch", the OS execute a low-level piece of code, which is referred to as a context switch

정보를 알 필요

이전의 cpu가 kernel-stack 저장 ...①

- **A context switch is conceptually simple**

2번 해야 함!

  - A few register values for current process are saved onto the memory and a few for the soon-to-be-executing process are restored from the memory "PCB 저장"
  - By doing so, the OS ensures that when return-from-trap instruction is executed, the system resumes execution of another process, instead of the original one

- **The OS will execute some low-level assembly code to**
  - save general purpose registers, PC, and the kernel stack pointer of currently-running process
  - then, restore the registers and PC, and switch to the kernel stack (by changing kernel stack pointer) for the soon-to-be-executing process

Limited Direct Execution Protocol (Timer Interrupt)

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

STC

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B) return-from-trap (into B)		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

HW

CPU

k-stack (A)

k-stack B

- Initializing when booting up

kernel stack

regs(A)

regs(B)

OS

PCB(A)

PCB(B)

process

- Running process A & Timer interrupt

k-stack

stack-pointer

- Scheduling & Context switch (A → B)
  By changing the stack pointer to use the B's kernel stack (not A's)

PCB

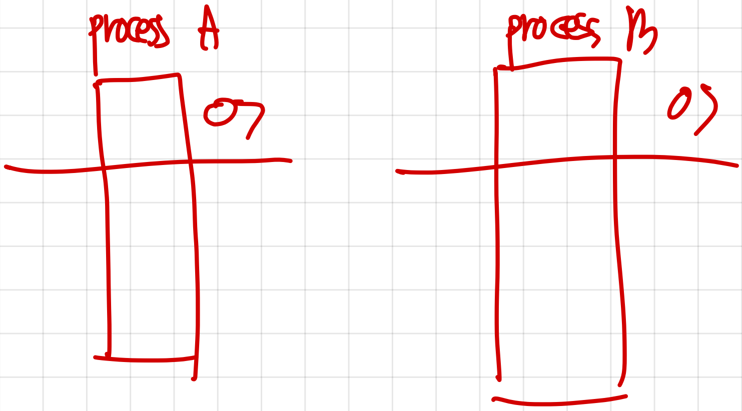
kernel-stack

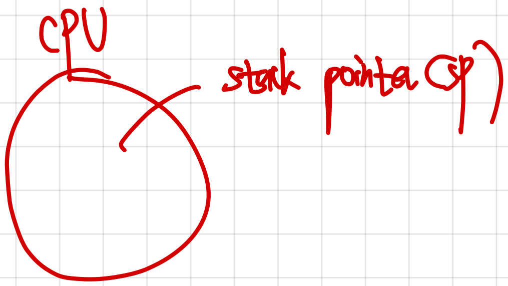
- Returning from interrupt & Running process B

register

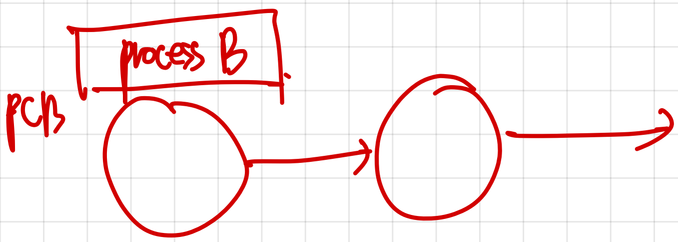
stack pointer

- There are two types of register saves and restores in LDE
  1) The user registers of the running process are implicitly saved to its kernel stack by hardware (CPU) when a timer interrupt occurs
  2) The kernel registers are explicitly saved to its PCB in memory by software (OS) when the OS decides to switch





kernel



IDT IVT
trap table

1. timer interrupt

2. (CPU) regs (A) → kernel (A)

↓ (kernel-mode)

3.

OS boot
(kernel-mode)

Hardware

initialize trap table → (remember address of syscall-handler, timer-interrupt handler)

start interrupt timer → (X msec 지난후 CPU를 인터럽트 한다)

운영체제
(커널모드)

하드웨어

프로그램

프로세스 A
...

타이머 인터럽트 → OS에 제어권을 넘기기 위함

A의 레지스터를 kernel-stack에 저장

커널 모드로 이동

트랩 핸들러.

트랩 핸들

switch() 루틴

A의 레지스터 값을 A의 proc 구조에 저장

B의 proc 구조로부터 B의 레지스터를 복원

B의 kernel-stack으로 전환

return-from-trap → B의 커널 스택을 B의 레지스터에 저장

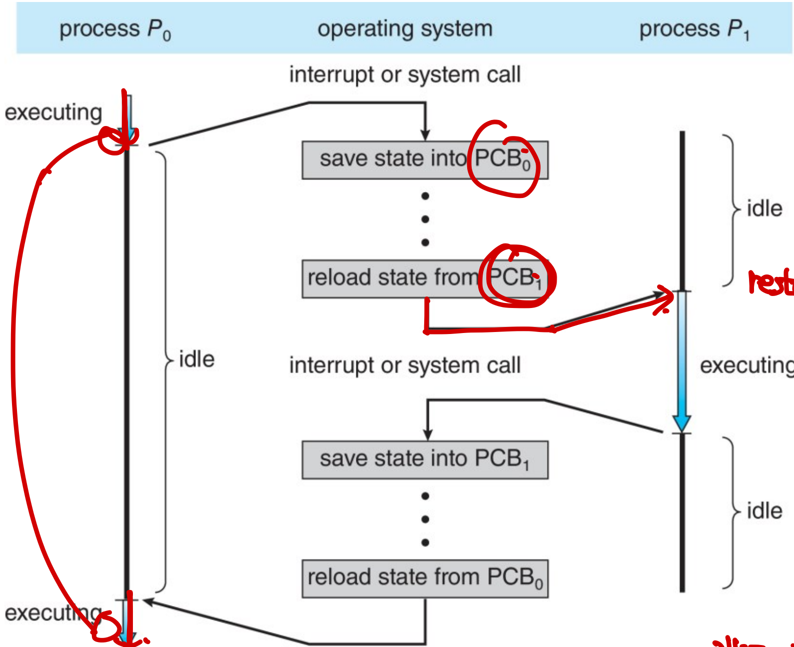
사용자 모드

B의 PC로 이동

Context Switching Revisited and Concurrency Issue

CPU가 상태를 인식하고 있어야 한다.

- **The context of a process is represented in the PCB**



- **Statistics of context switches in Linux**

켜져 있는 시간

  - Total uptime: 2,176,310 sec (25 days)
  - Total 14,115,783,018 context switches
  - Average 6486 context switches/sec
  - Average 541 context switches/sec/CPU (for all 12 CPUs)

12 core

```
[tahoe: /home/yongtae]$ cat /proc/uptime
2176310.83 66103492.27
[tahoe: /home/yongtae]$ grep ctxt /proc/stat
ctxt 14115783018
[tahoe: /home/yongtae]$ echo "14115783018 / 2176310.83" | bc -l
6486.10613126434701425439
[tahoe: /home/yongtae]$ echo "6486.10613126434701425439 / 12" | bc -l
540.50884427202891785453
[tahoe: /home/yongtae]$
```

idle 한 상태가 많았던 서버

커널모드 처리중 → 마우스 처리

- **Worried about concurrency?**

HW-interrupt

  - What happens when, during a system call, a timer interrupt occurs?
  - What happens when you're handling one interrupt and another one happened?
  - Then, the OS can disable interrupt during the interrupt handling (dangerous), and use some sophisticated locking schemes ➔ concurrency issue