# Operating System: Limited Direct Execution

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

# How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing

- Issue

성능면에서는 시스템적인 과도한 오버헤드를 어떻게 줄일수 있는지?

  – **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?

컨트롤면에서는 자원을 더 효율적으로 사용하기 위한 방법은?

# Direct Execution

문제없이 그냥 시작부터 종료까지 쭉 실행되는거 이때 OS는 그냥 라이브러리 수준의 일만 하는 거.

- 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"**

OS는 추가적인 문제에 대한 컨트롤을 할 줄 알아야함.

# Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory 할당 받은 자원 이상을 사용하려할때.

- **Solution**: Using protected control transfer
  - User mode: Applications do not have full access to hardware resources.
  - Kernel mode: The OS has access to the full resources of the machine

접근권한을 나눠서 문제를 해결.

# System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

# System Call (Cont.)

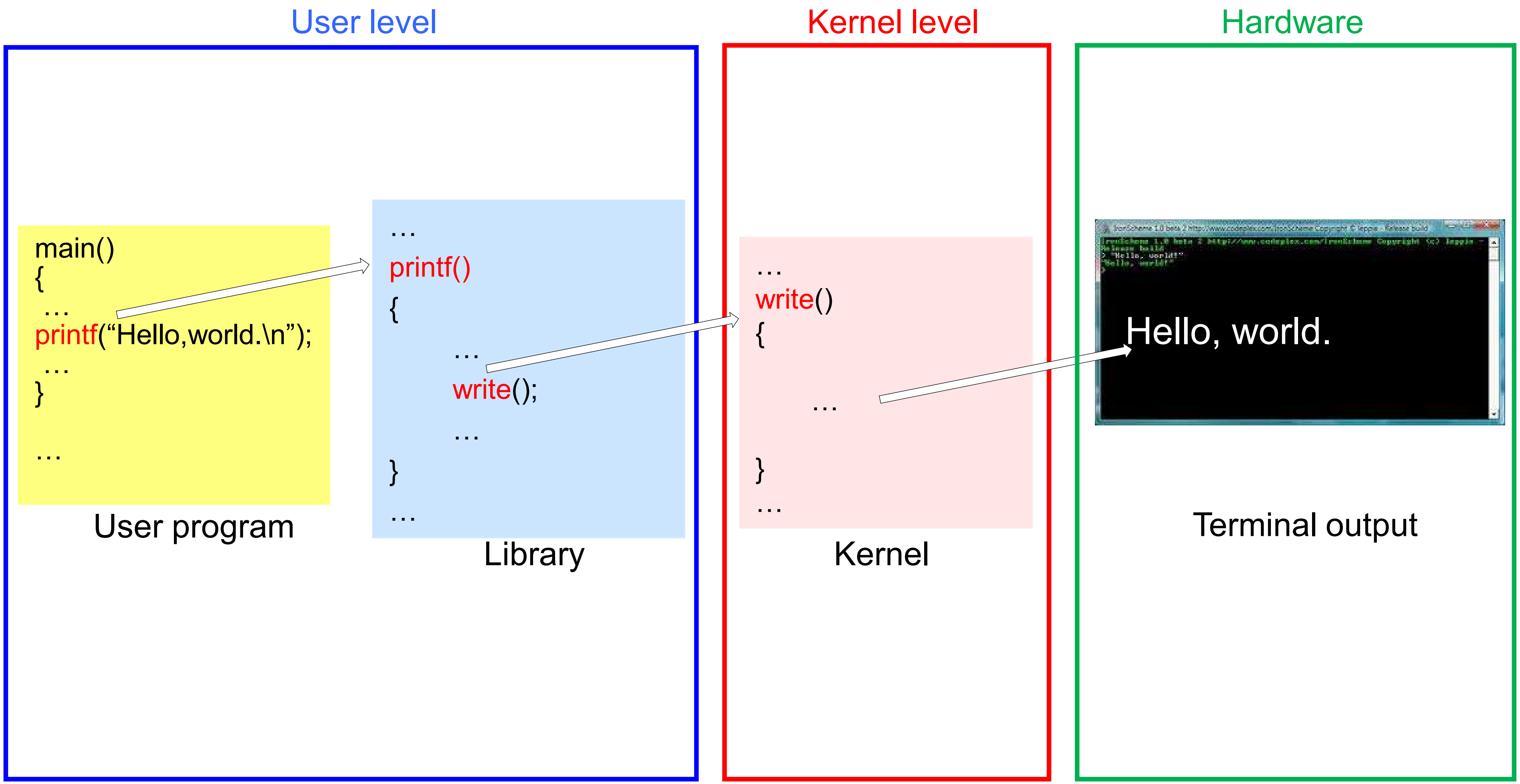
user mode → kernal mode
넘어갈때 trap.

- Trap instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode
- Return-from-trap instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode
- How does the trap know which code to run inside the OS?
  - trap table
    - also called interrupt descriptor table or interrupt vector table
  - trap handler
    - The code should be run when a program executes a trap instruction

# System Call (Cont.)

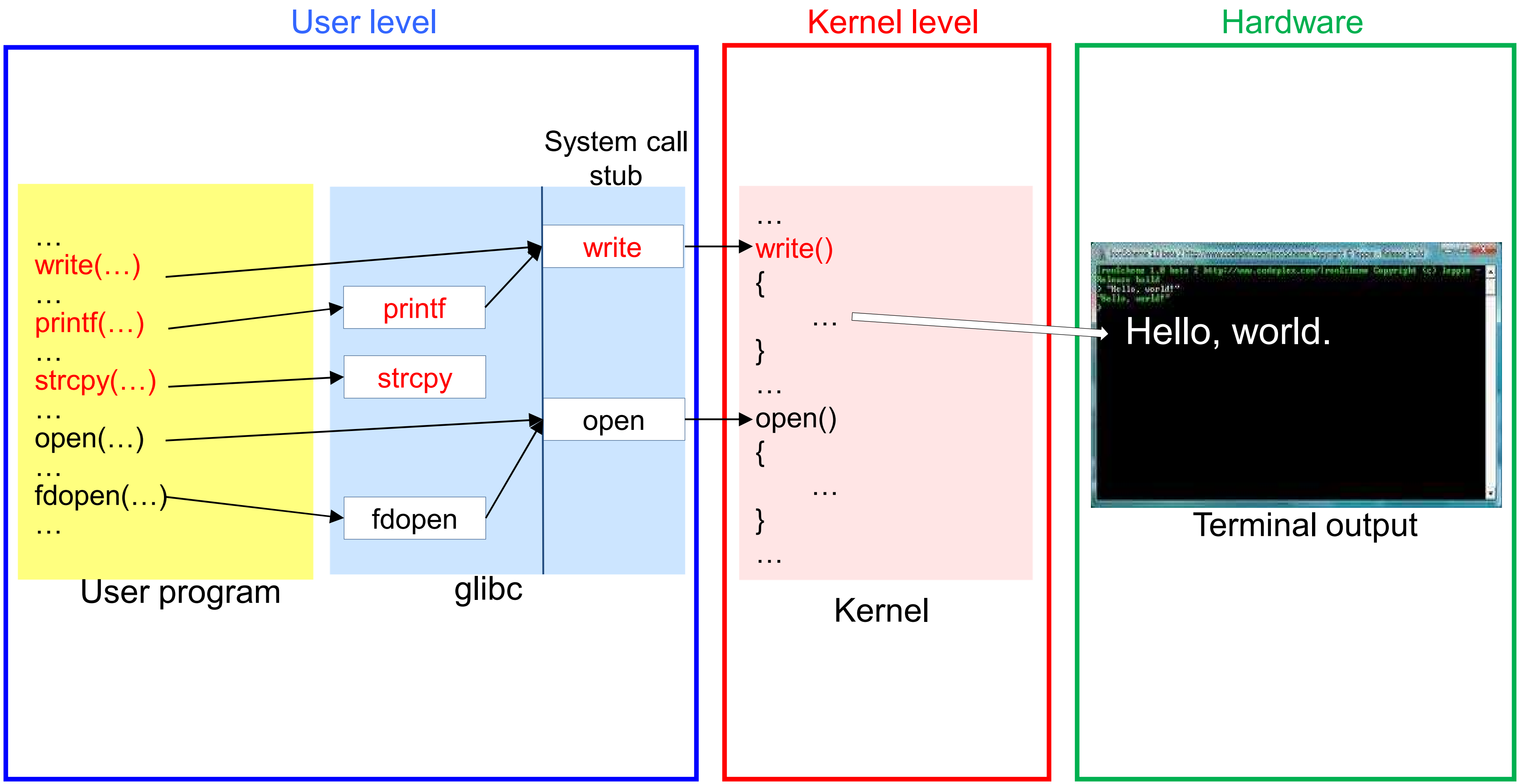
- System-call number
  - Assigned to each system call
  - The user code is thus responsible for placing the desired system-call number in a register

# System Call Handling



# System Call Handling (Cont.)

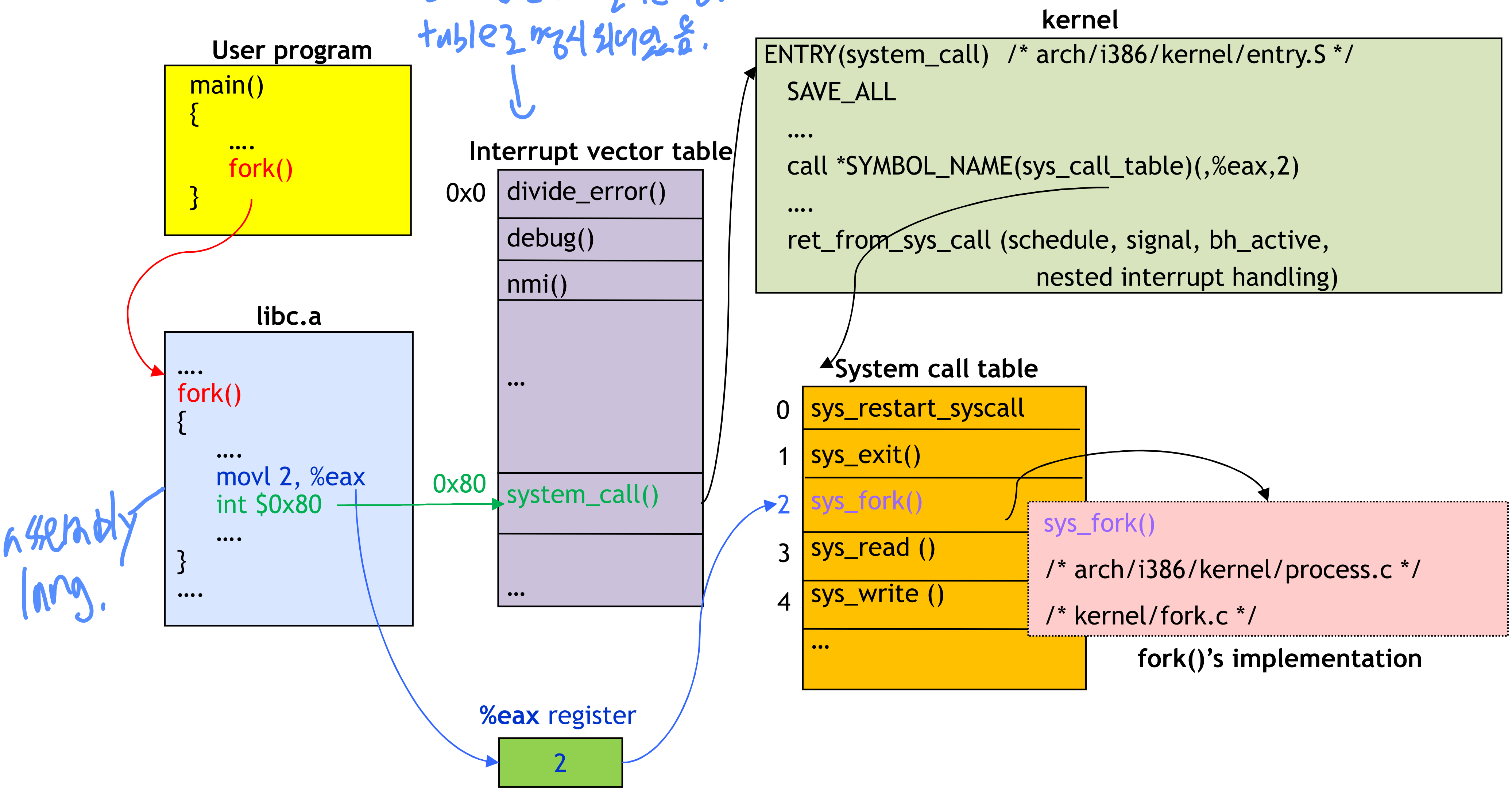
user mode의 write와
kernel mode의 write는 다름.



# System Call Handling (Cont.)

- Invoke fork() system call (i386-linux system)

인터럽트가 걸리는 경우가
table로 명시되어있음.



# Limited Direction Execution Protocol

| OS @ boot (kernel mode) | Hardware | |
|---|---|---|
| **initialize trap table** | | |
| | remember address of …<br>syscall handler | |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| Create entry for process list<br>Allocate memory for program<br>Load program into memory<br>Setup user stack with argv<br>Fill kernel stack with reg/PC<br>**return-from -trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to main | |
| | | Run main()<br>…<br>Call system<br>**trap** into OS |

user stack :

kernel stack : system call 함수내용 .

# Limited Direction Execution Protocol (Cont.)

| **OS @ run (kernel mode)** | **Hardware** | **Program (user mode)** |
|---|---|---|
| | *(Cont.)* | |
| | save regs to kernel stack<br>move to kernel mode<br>jump to trap handler | |
| Handle trap<br>Do work of syscall<br>**return-from-trap** | | |
| | restore regs from kernel stack<br>move to user mode<br>jump to PC after trap | |
| | | …<br>return from main<br>trap (via `exit()`) |
| Free memory of process<br>Remove from process list | | |

# Problem 2: Switching Between Processes

- 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**

1 2

context

CPU switching.

프로그램에 CPU를 어떤식으로 할당할것인가?

# A cooperative Approach: Wait for system calls

프로세스가 자발적으로 yield라는 함수로 CPU를 내어줌. OS는 이를 기다림.

- 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 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

yield도 안하고 illegal 한것도 안하면 하나가 계속 써먹음.

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

# A Non-Cooperative Approach: OS Takes Control

- A timer interrupt
  - During the boot sequence, the OS start the timer
  - The timer raise an interrupt every so many milliseconds
  - When the interrupt is raised :
    - ➢ The currently running process is halted
    - ➢ Save enough of the state of the program
    - ➢ A pre-configured interrupt handler in the OS runs

**A timer interrupt gives OS the ability to run again on a CPU**

# Saving and Restoring Context

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**
  - If the decision is made to switch, the OS executes context switch

# Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore a few** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

# Limited Direction Execution Protocol (Timer interrupt)

| OS @ boot (kernel mode) | Hardware |
|---|---|
| **initialize trap table** | |
| | remember address of …<br>syscall handler<br>timer handler |
| **start interrupt timer** | |
| | start timer<br>interrupt CPU in X ms |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | | Process A<br>… |
| | **timer interrupt**<br>save regs(A) to k-stack(A)<br>move to kernel mode<br>jump to trap handler | |

# Limited Direction Execution Protocol (Timer interrupt)

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| | *(Cont.)* | |
| Handle the trap<br>Call switch() routine<br>  save regs(A) to proc-struct(A)<br>  restore regs(B) from proc-struct(B)<br>  switch to k-stack(B)<br>**return-from-trap (into B)** | | |
| | restore regs(B) from k-stack(B)<br>move to user mode<br>jump to B's PC | |
| | | Process B<br>… |

# Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?

- OS handles these situations:
  - **Disable interrupts** during interrupt processing 무시
  - Use a number of sophisticate **locking** schemes to protect concurrent access to internal data structures