Main Points

- Process concept
  - A process is the OS abstraction for executing a program with limited privileges
- Dual-mode operation: user vs. kernel

  RAM → always work

  - Kernel-mode: execute with complete privileges
  - User-mode: execute with fewer privileges
- Safe control transfer
  - How do we switch from one mode to the other?

switch mode

user

kernel

system call

f1()

f2()

end

interrupt

ตัวเรา u

INT(n)

ไปที่ตำแหน่ง
intrp vector

สัญญาณ

INT(1)

(เกิดได้เป็นสัญญาณ

HW

*

iH : interrupt handler

k

intrp vector

0	ad 0
1	ad 1
2	ad 2
	....
n	ad n

iH1 {...}

iH2 {...}

iH3 {...}

iHn {...}

Mode Switch

system call

1

0

error

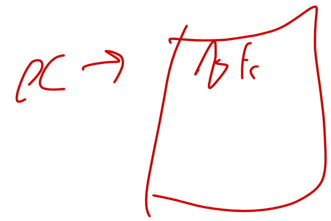
- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

Mode Switch

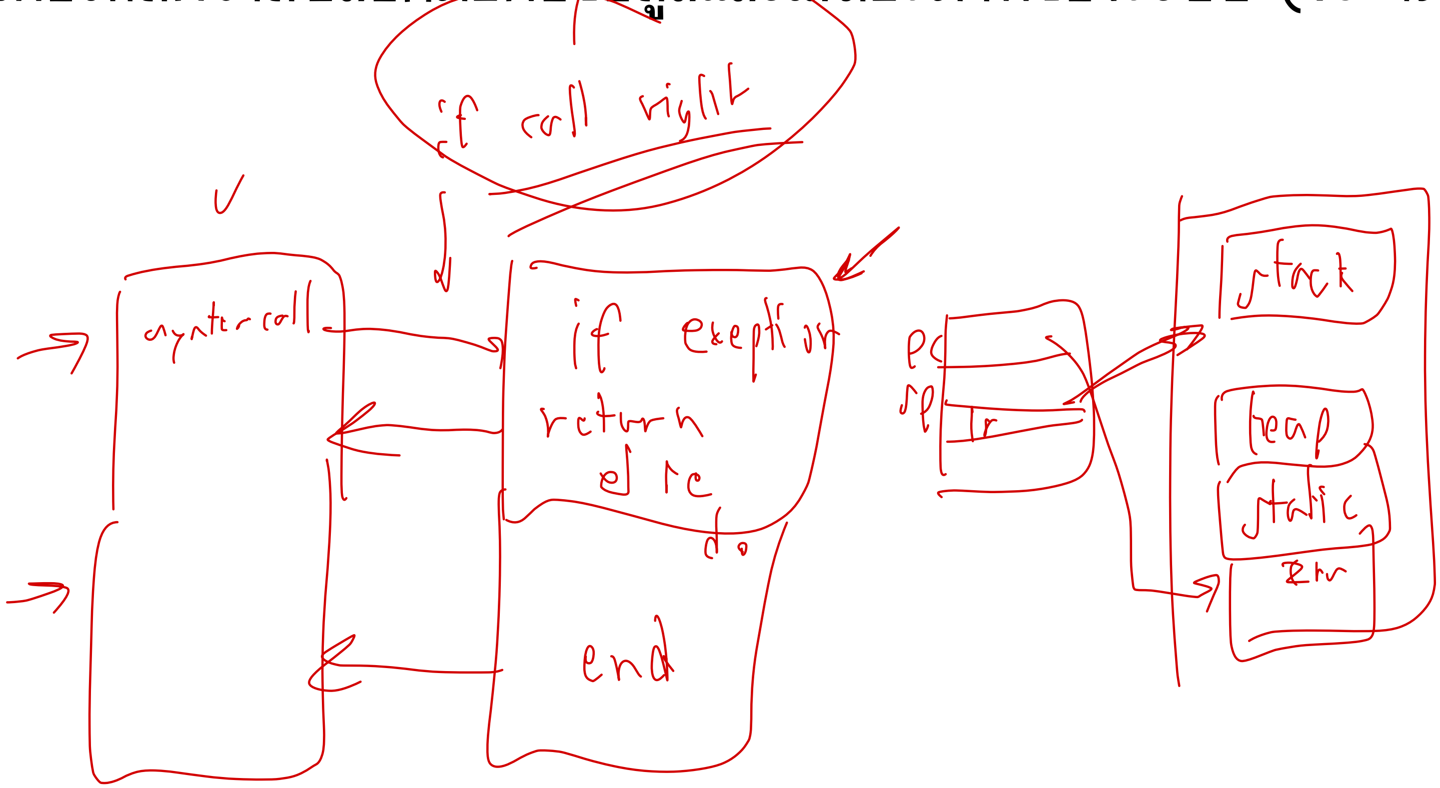
- From kernel mode to user mode ไม่ต้องให้รู้ว่า switch mode
  - create by kernel
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch เกิดใน kernel
    - Resume some other process
  - User-level upcall (UNIX signal) → บริการให้ user register บน program ให้ kernel ใช้
    - Asynchronous notification to user program

Up call   fn → program → user

Activity #1



- ในความเห็นของ นศ การทำ mode switch ควรทำอย่างไรบ้าง เพื่อให้มีความปลอดภัยต่อข้อมูลและเสถียรภาพของระบบ (10 นาที)



Implementing Safe Kernel Mode Transfers

- เก็บ process state ว่า instruction อะไร อยู่แค่ไหน register เวลานั้น แล้ว save เก็บไว้
- ต้อง handle weird / buggy / malicious user state ดักด้วย เช่นว่า
  - system calls with null pointer / call อะไรเนี่ย
  - return instr out of bound / กลับออกจาก fn ได้ไหม
  - user stack pointer out of bound
- ต้องไม่ให้ bug ที่ให้ kernel ทำงานผิดปกติ หรือล่มเลย
- user ต้องไม่รู้ว่ามี interrupt เกิดขึ้น

Device Interrupts

- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - Polling: Kernel waits until I/O is done   kernel เฝ้าดูเป็น ช่วงๆ
  - Interrupts: Kernel can do other work in the meantime

ส่งข้อมูลไปแล้ว รอ แล้วได้

ไม่ต้องรอ   ส่งไป แล้ว ไปทำงานอื่นต่อ

- Device access to memory
  - Programmed I/O: CPU reads and writes to device   ไม่ มี
  - Direct memory access (DMA) by device   รับ
  - Buffer descriptor: sequence of DMA's   ไม่ทำ   เก็บ
    - E.g., packet header and packet body
  - Queue of buffer descriptors
    - Buffer descriptor itself is DMA'ed   พอจบ ก็ส่ง int

program $\xrightarrow[\text{RAM}]{\text{เขียน}}$ file

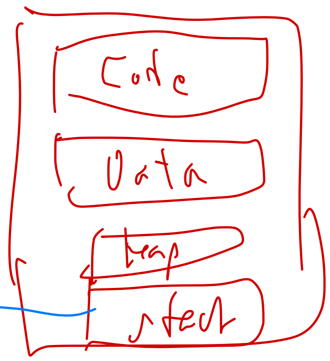
1 descriptor = 1 op   ไม่ต้อง มี

Activity #2

- How do device interrupts work?
  - Where does the CPU run after an interrupt? ก็ทำงานที่เดิม
  - What stack does it use? → ก็มีตัวแปร มีโปรแกรม และใช้ stack ของ process ที่ดู int ใน kernel
  - Is the work the CPU had been doing before the interrupt lost forever? ไม่หายไป
  - If not, how does the CPU know how to resume that work? ก็ดูจาก register ใน stack

(10 นาที)

Process



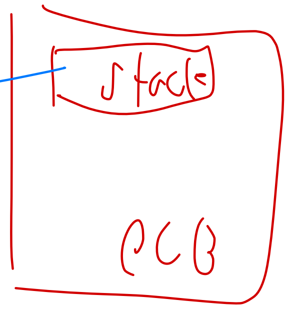
U

User mode
ก็ใช้ stack นี้

ทำไม
ต้องแยก stack

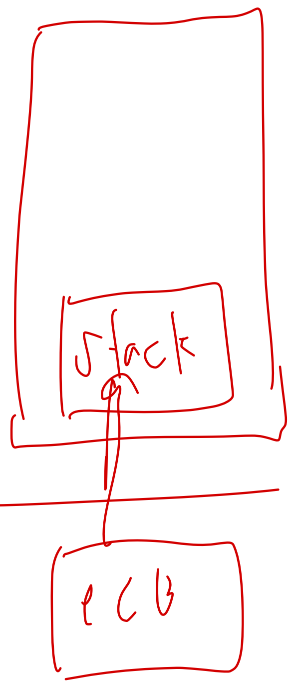
K

ถ้าใช้ kernel mode
ก็ใช้ stack นี้



เป็นเรื่องของ
security

if



U

K

และ หลบบ กฎ

int routine ปลอมเป็นผล ปลด

system call โจมตีได้

เช่น ตอนเกิด ปกติ
แต่ถ้า ไม่ stack แล้วแก้ค่า
ก็อาจ คุม kernel ได้

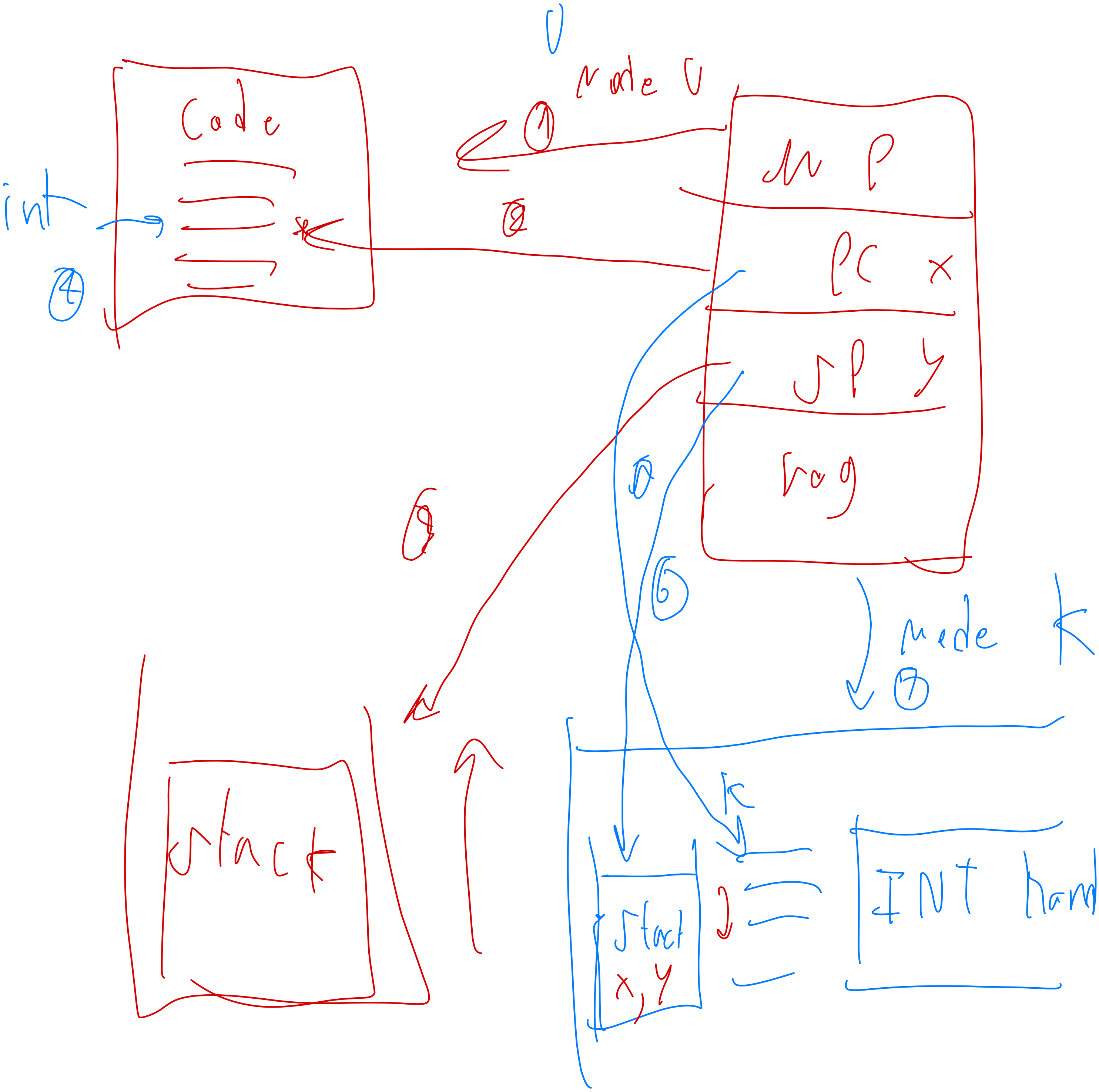
แต่ถ้าถูกแล้ว จะสามารถแก้
ก็แก้ไม่ได้กับ stack
ใน kernel

ดู slide 22 ชุดที่ 02

How do we take interrupts safely?

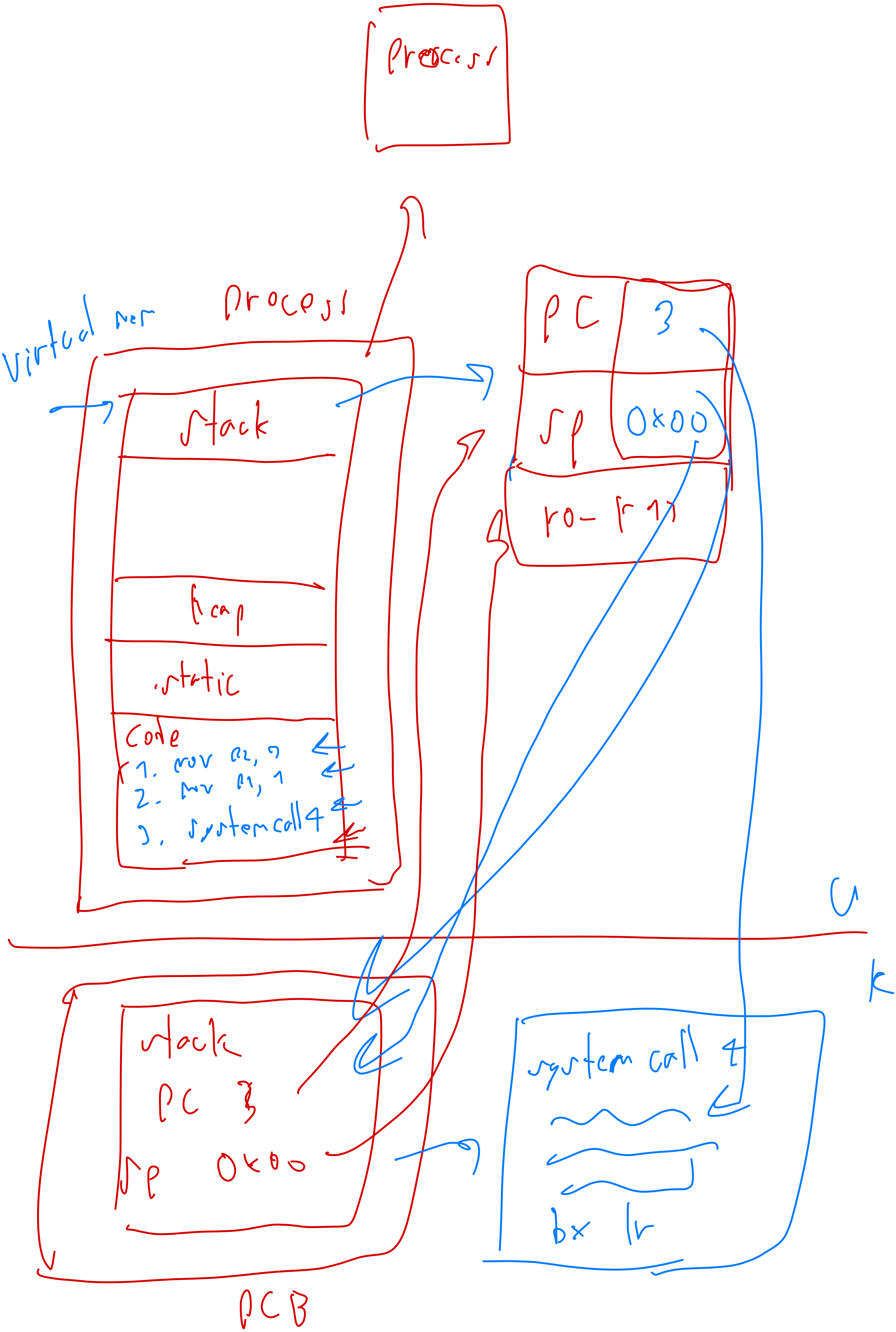
- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking → ในขณะที่เกิด int จะไม่ให้มี int ซ้อน (OS เหมือนกัน)
  - Handler is non-blocking
- Atomic transfer of control
  - "Single instruction"-like to change: (instr ที่ทำทั้งหมด ใน instr เดียว
    - Program counter → pc
    - Stack pointer → sp
    - Memory protection →
    - Kernel/user mode → sw

    1 instr
- Transparent restartable execution
  - User program does not know interrupt occurred



ตอนกลับ ก็เอา x, y แต่ละ ค่ากลับ

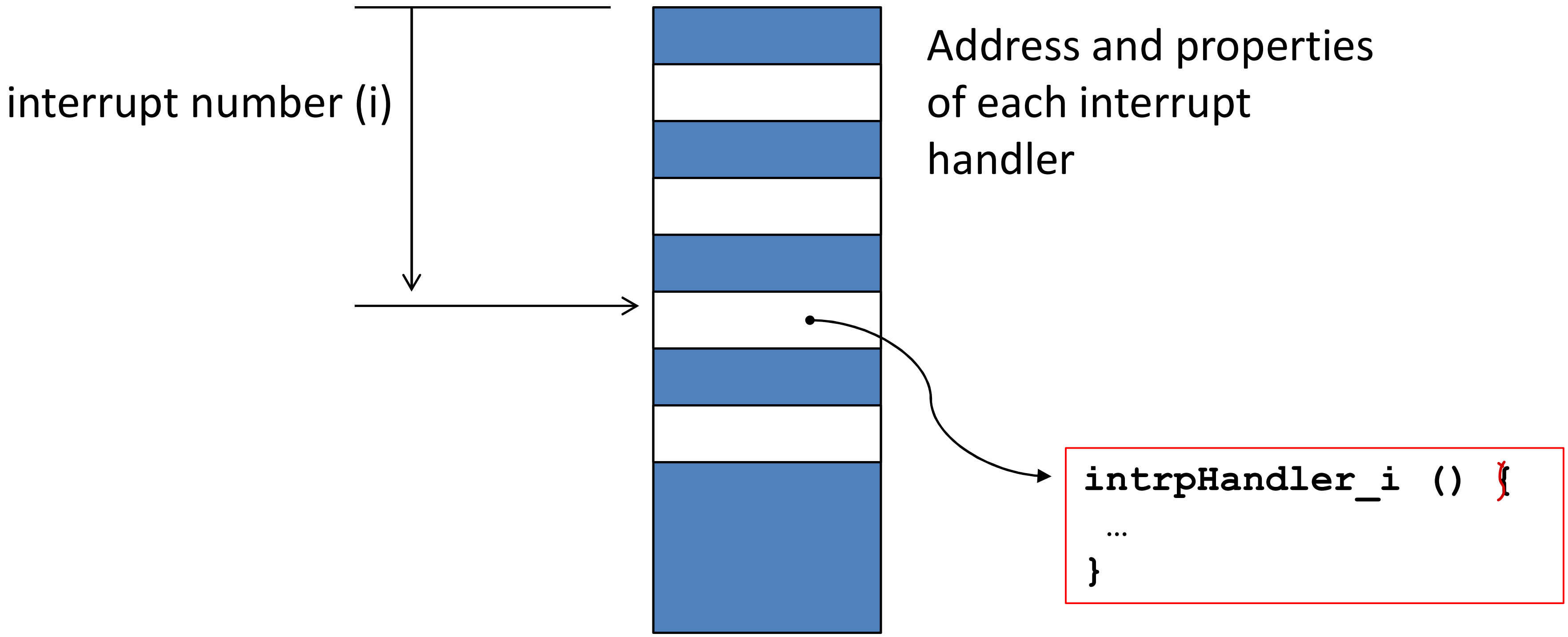
ซึ่ง pointer ก่อน switch mode



Process
virtual mem
Process
stack
heap
static
code
1. mov r2, 7
2. mov r1, 1
3. system call 4
PC
3
sp
0x00
r0 - r1
U
k
stack
PC 3
sp 0x00
PCB
system call 4
bx lr

Where do mode transfers go?

- Solution: ***Interrupt Vector***



The Kernel Stack

- Interrupt handlers want a stack
- System call handlers want a stack
- Can't just use the user stack [why?]

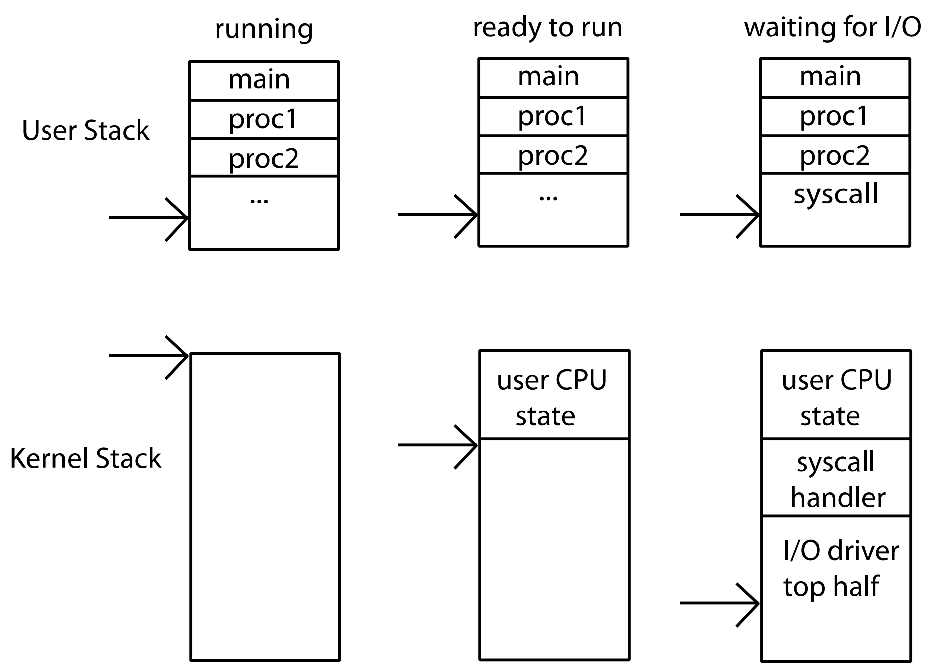
ตอบไปแล้ว

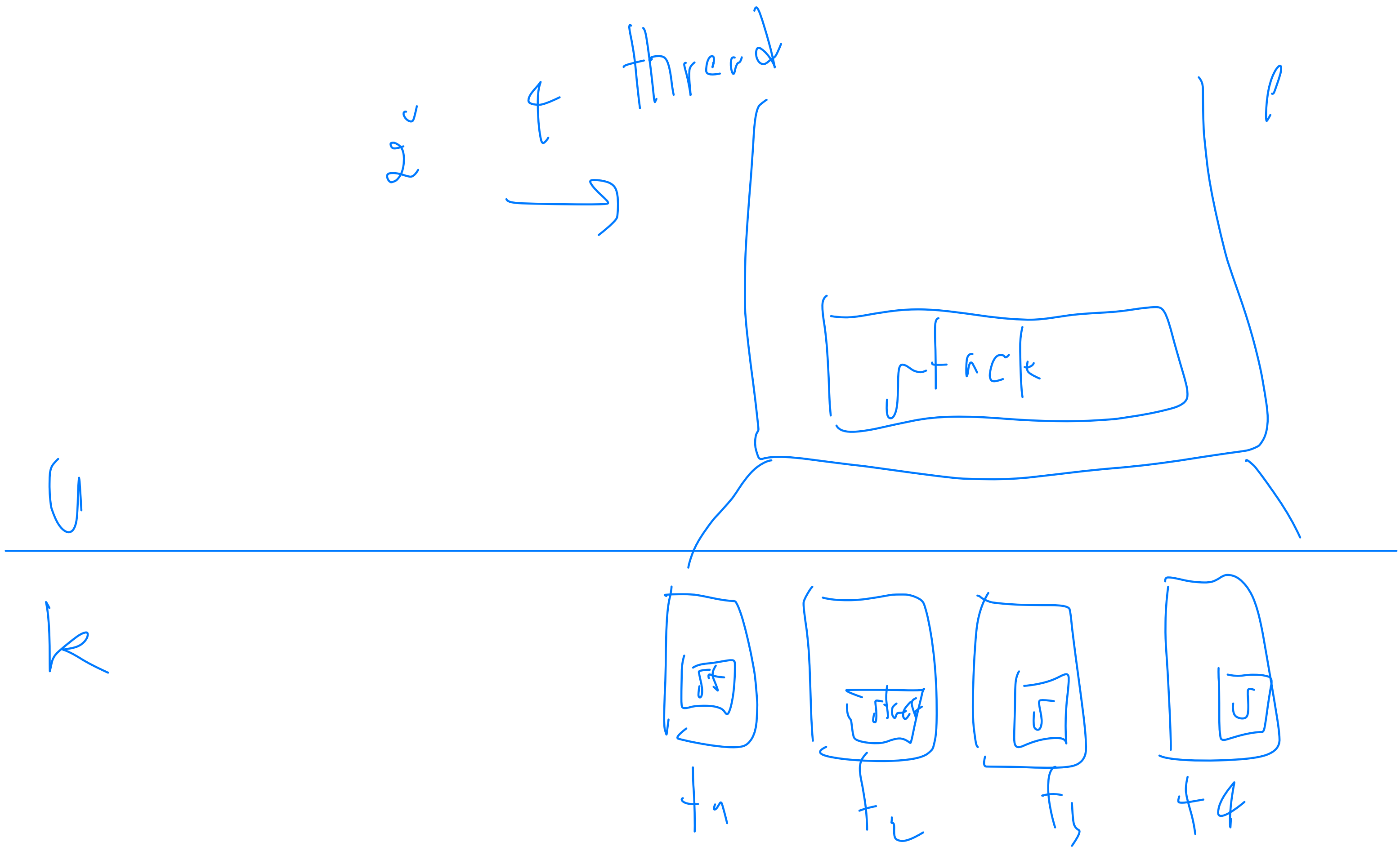
The Kernel Stack

multi-thread like window

process 1 — m thread

- Solution: two-stack model
  - Each OS thread has kernel stack (located in kernel memory) plus user stack (located in user memory)
- Place to save user registers during interrupt




thread
stack
u
k
t1
t2
t3
t4

Interrupt Stack

- Per-processor, located in kernel (not user) memory
  - Usually a process/thread has both: kernel and user stack
- Why can't the interrupt handler run on the stack of the interrupted user process?

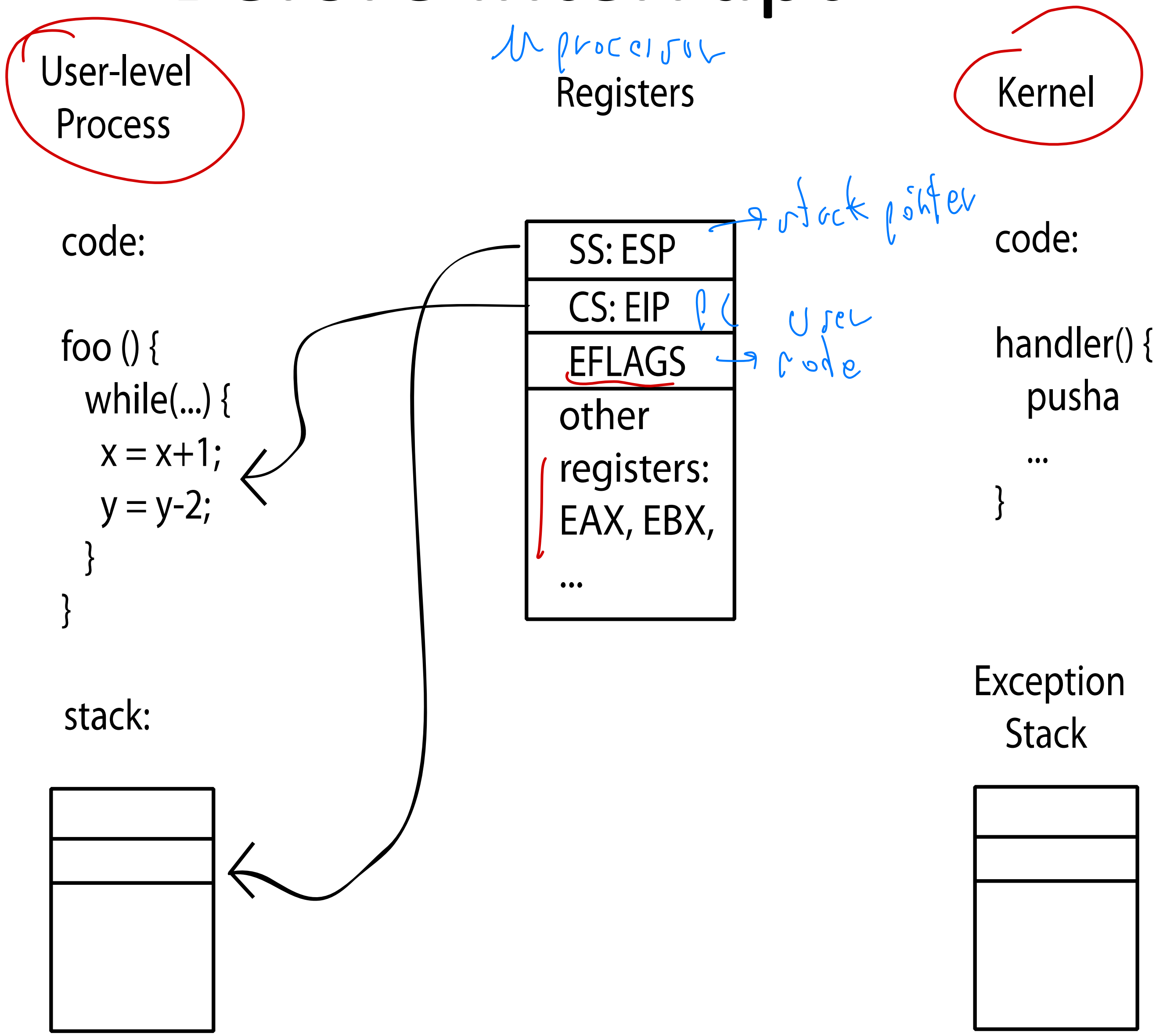
Case Study: x86 Interrupt

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Interrupt handler saves registers it might clobber

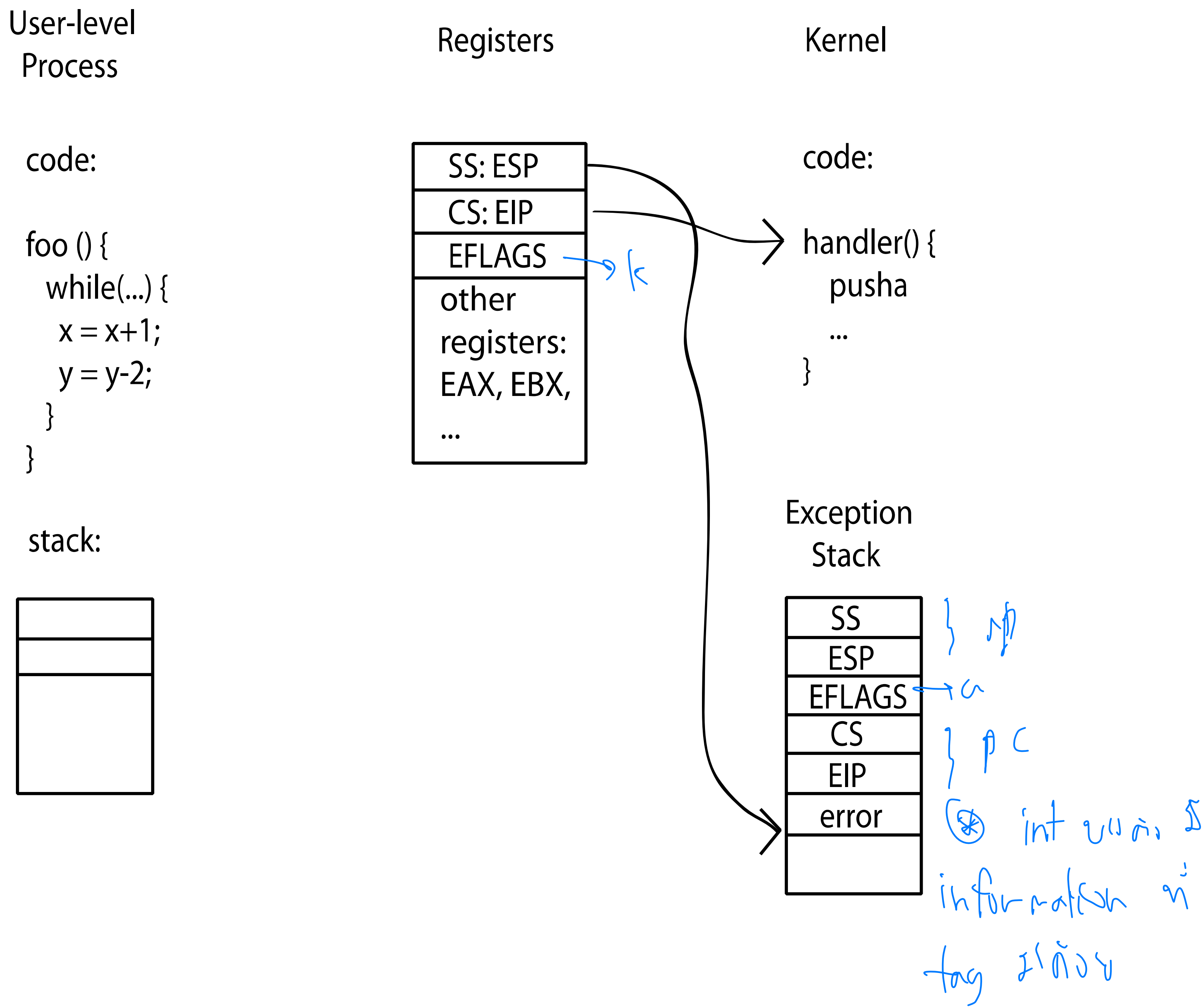
save auto

สลับ

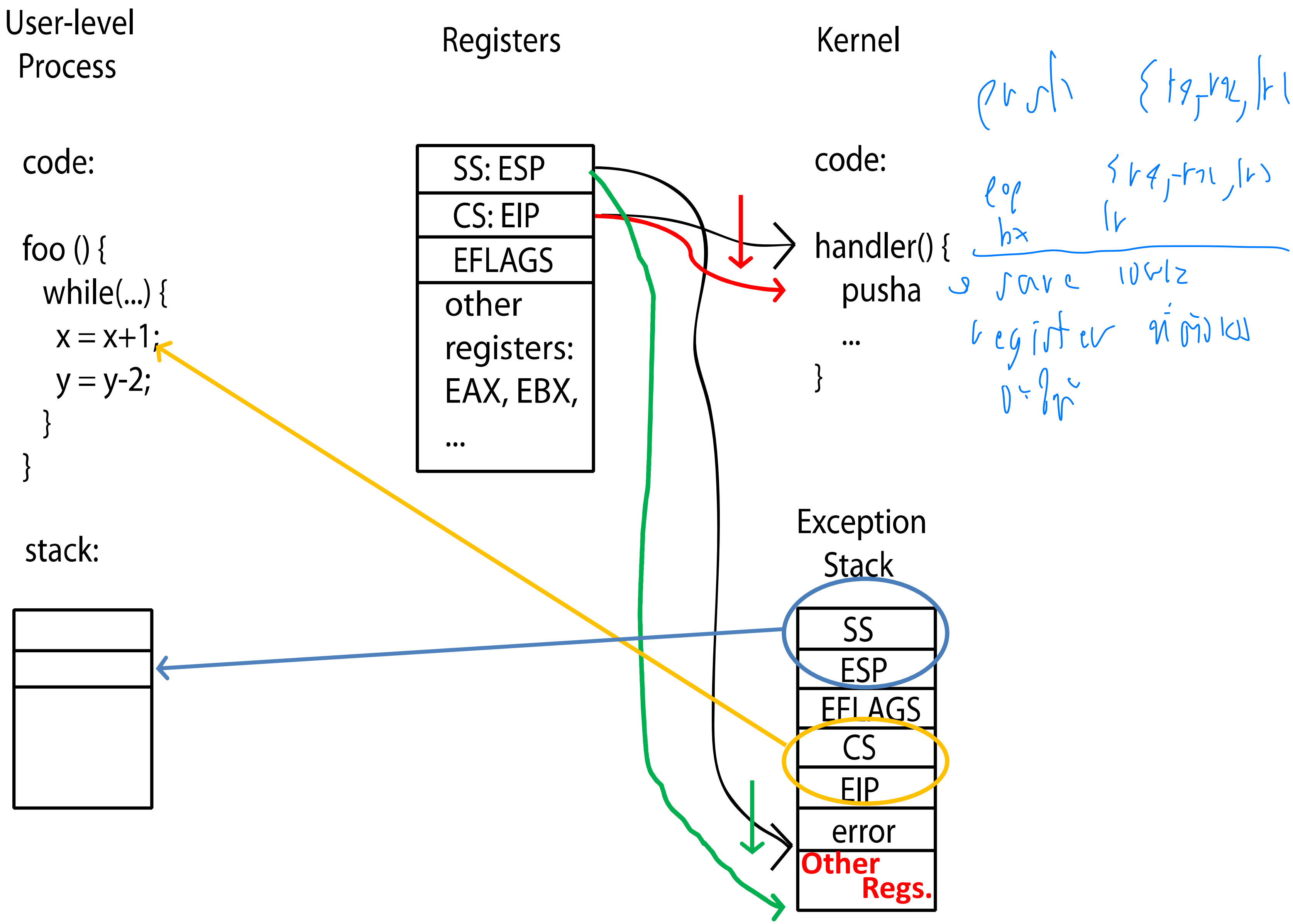
Before Interrupt



During Interrupt



After Interrupt



At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

Interrupt Masking

ถ้าเกิด int ก็disable ขา int ของ cpu

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We'll need this to implement synchronization in chapter 5

Hardware support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - wake up an existing OS thread

Hardware support: Interrupt Control

- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall

ป้องกัน แบ่ง

- HW may have multiple levels of interrupts
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    - e.g., kernel segmentation fault
    - Also: Power about to fail!

แบตหมด

int ที่สำคัญสุด

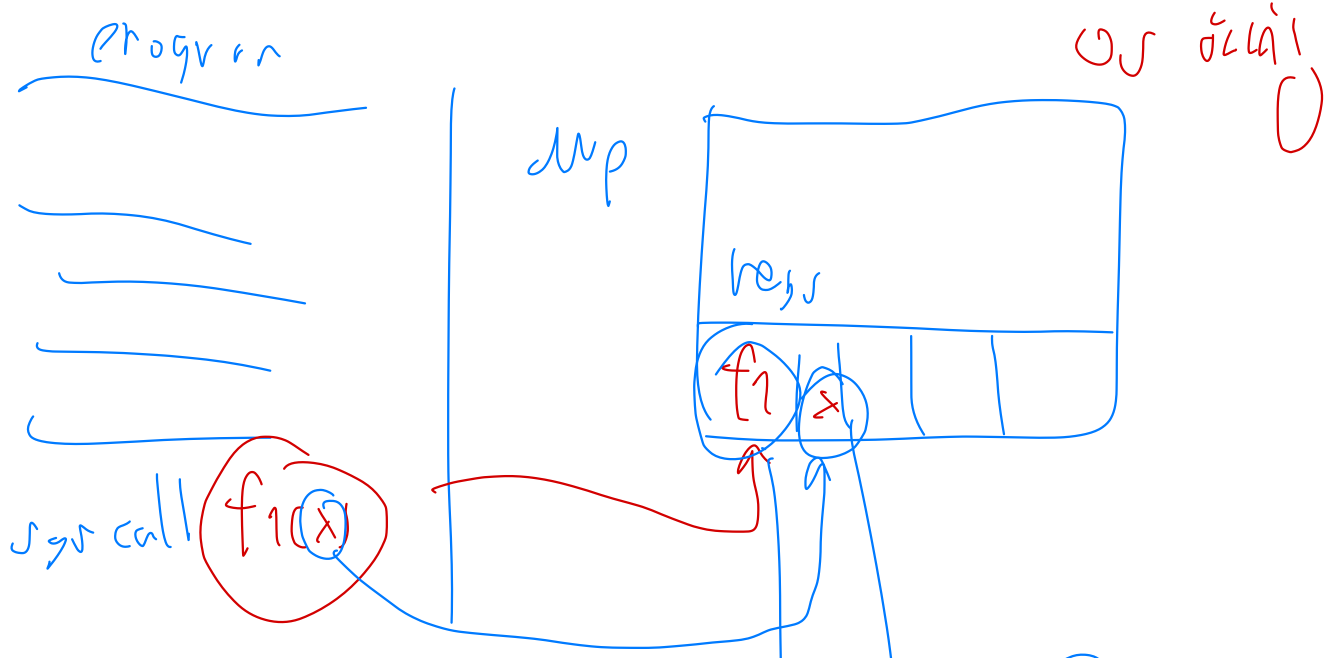
LV
ถ้า int 0
int 1

int Level 0 เกิด
int Level 1 เกิด
ก็ drop อันเก่าทิ้ง

OS [illegible] syscall [illegible] int

Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory – carefully checking locations!
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory – carefully checking locations!



แต่รอบนี้ exec code ต่างๆ = trap

ทำยังไงให้เกิด ซอฟต์แวร์ int

ผลคือ เกิด soft int (แต่ไม่ได้วิ่งไป int routine)

handler

2 ตรวจสอบ param ถูกไหม ถ้าถูกก็ call syscall f1(x)

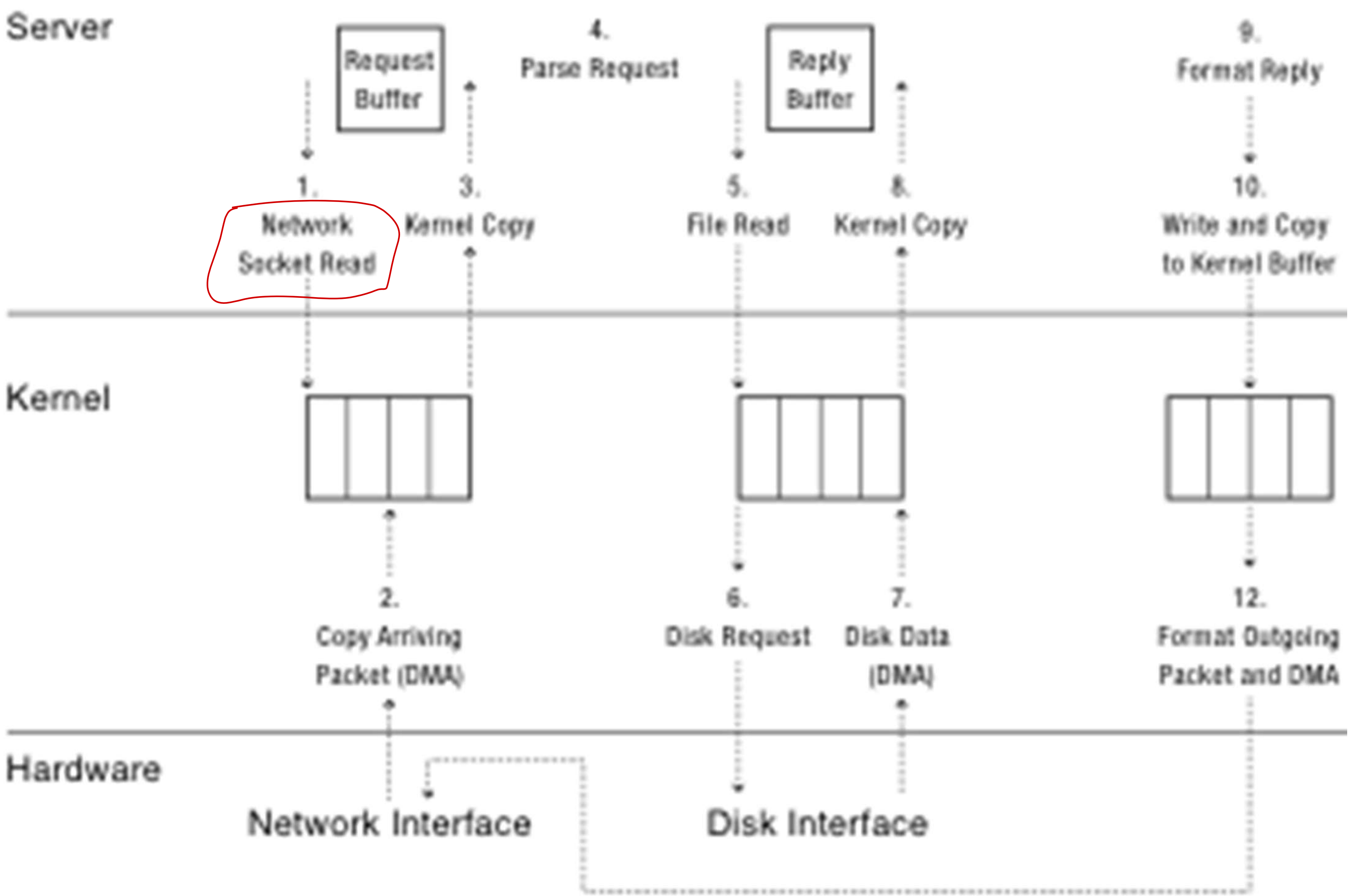
stack

K

ใน x86 trap ก็เปลี่ยนเป็น sysENT

ก็ทำงานคล้าย Trap

network socket read



Today: Four Fundamental OS Concepts

- **Thread: Execution Context** vir Processor
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**) real
  - Program's view of memory is distinct from physical machine
- **Process: an instance of a running program**
  - Address Space + One or more Threads
- **Dual mode operation / Protection** U / K
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other

Protect

kernel