Main Points

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

Mode Switch U→k

- From user mode to kernel mode
  - Interrupts เหตุการณ์ที่ถูกสร้างโดย Hw & มีเลข
    - Triggered by timer and I/O devices
  - Exceptions error แบบอิกฤต พวก ÷0 & ที่ให้เกิดการ switch mode
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (aka protected procedure call) ↳ เลือกต้อง switch mode
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

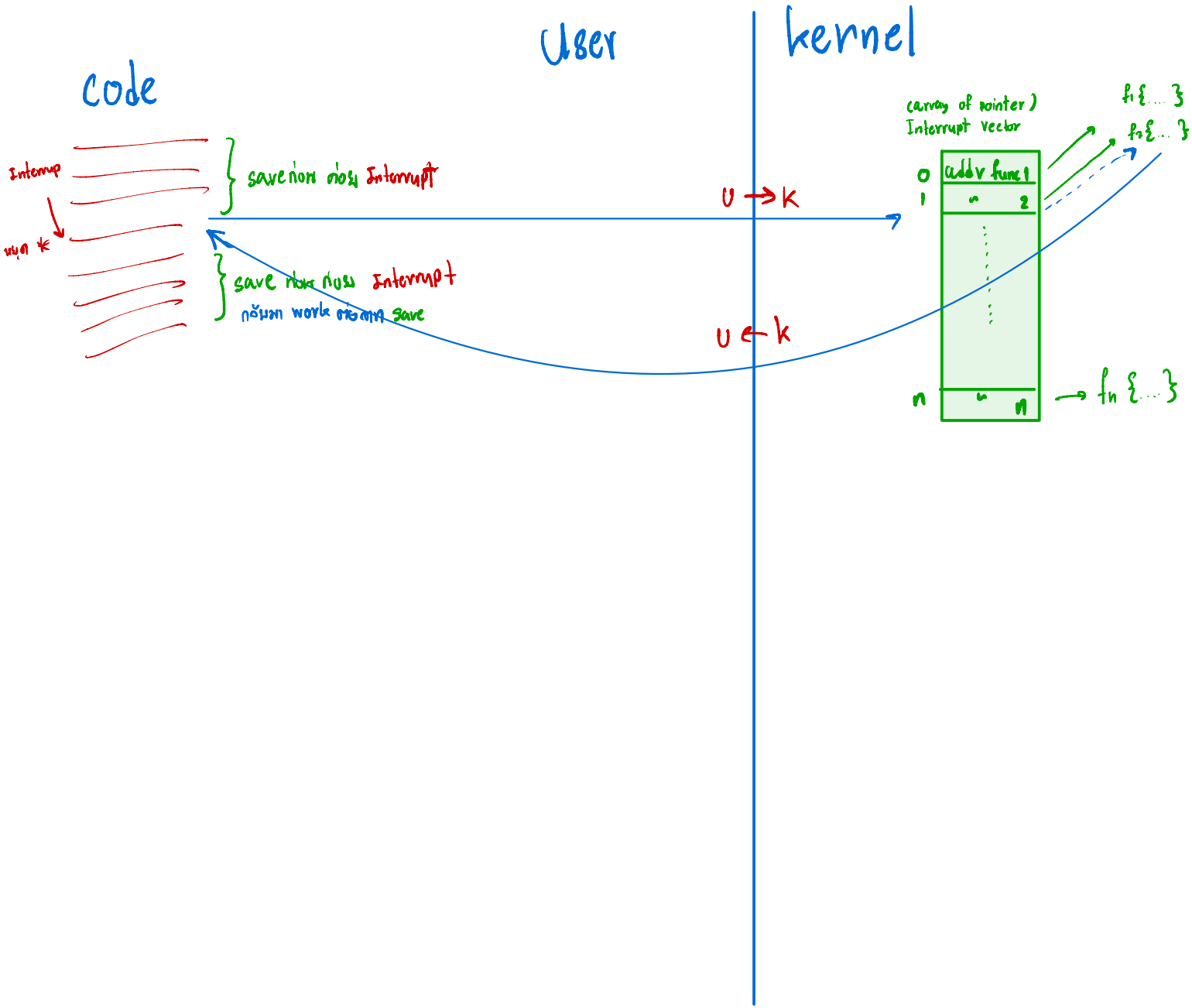
Mode Switch K → U

- From kernel mode to user mode
  - 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
    - Resume some other process
  - User-level upcall (UNIX signal)
    - Asynchronous notification to user program

Activity #1

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

Implementing Safe Kernel Mode Transfers

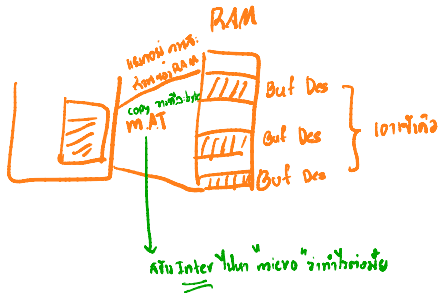


Implementing Safe Kernel Mode Transfers

- *Carefully* constructed kernel code packs up the user process state and sets it aside → save file
- Must handle weird/buggy/malicious user state ตรวจสอบ
  - Syscalls with null pointers
  - Return instruction out of bounds
  - User stack pointer out of bounds
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself
- User program should not know interrupt has occurred *(transparency)* ไม่ให้ Hacker ออกแบบการเข้าสู่ระบบได้

Device Interrupts

- OS kernel needs to communicate with physical devices
- Devices operate asynchronously from the CPU
  - Polling: Kernel waits until I/O is done
  - Interrupts: Kernel can do other work in the meantime
- Device access to memory
  - Programmed I/O: CPU reads and writes to device
  - Direct memory access (DMA) by device
  - Buffer descriptor: sequence of DMA's
    - E.g., packet header and packet body
  - Queue of buffer descriptors
    - Buffer descriptor itself is DMA'ed



Activity #2

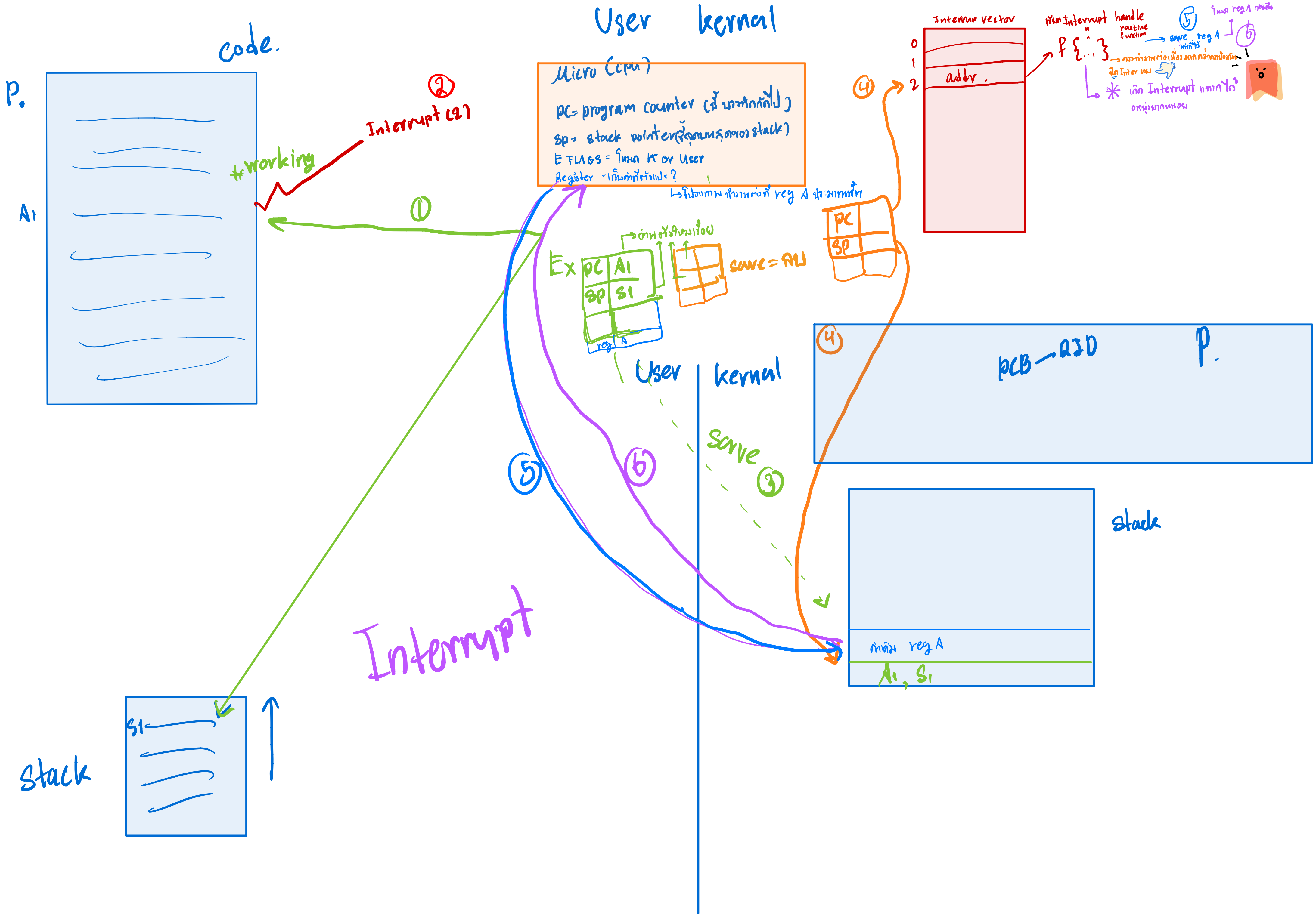
- How do device interrupts work?
  - Where does the CPU run after an interrupt? ตอน Interrupt ไปไหน
  - What stack does it use? เมื่อเกิด Inter ใช้ stack ใน U หรือ ?
  - Is the work the CPU had been doing before the interrupt lost forever? ข้อมูลไม่หาย
  - If not, how does the CPU know how to resume that work? รู้ได้ไงว่าไม่หาย save ไว้

(10 นาที)

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 ปิด Inter
  - Handler is non-blocking
- Atomic transfer of control
  - "Single instruction"-like to change: ฝ่ายที่ ฝ่ายpointer
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

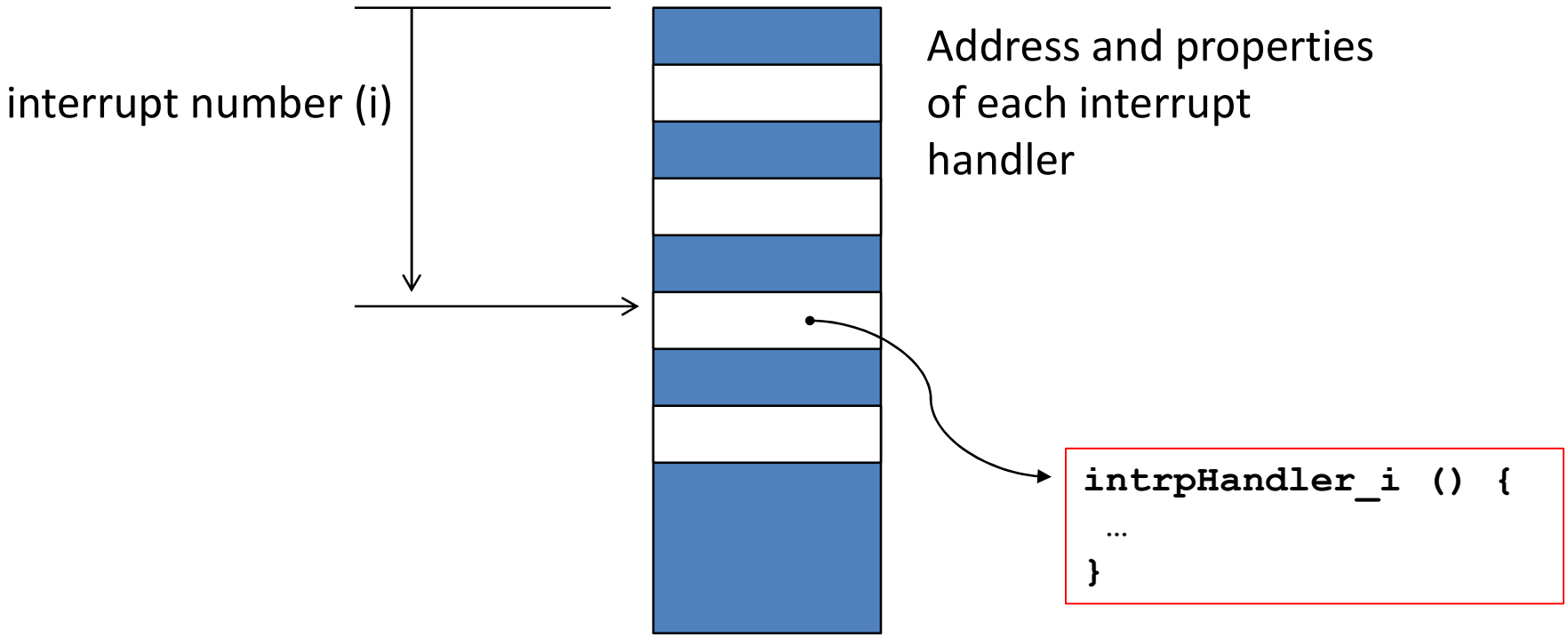
นอกไม่ปัด ว่า มี Inter เกิดขึ้น



code.
P.
A1
+working
Interrupt (2)
②
①
stack
S1
User
kernal
Micro (cpu)
pc = program counter (ชี้บรรทัดถัดไป)
sp = stack pointer (ชี้จุดบนสุดของ stack)
E FLAGS = โหมด K or User
Register = เก็บค่าที่ตัวแปร?
โปรแกรมทำงานต่อที่ reg A ประมาณนี้
Ex
pc
A1
sp
S1
reg
A
save
pc
sp
④
Interrupt vector
0
1
2
addr.
เรียก Interrupt handle
routine
function
f {...}
⑤
save reg A
เท่าที่ใช้
⑥
โหลด reg A กลับคืน
ปิด Inter ด้วย
เกิด Interrupt แทรกได้
User
kernal
Save
③
⑤
⑥
Interrupt
④
pcB — QID
P.
stack
ค่าเดิม reg A
A1, S1

Where do mode transfers go?

- Solution: ***Interrupt Vector***



The Kernel Stack

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

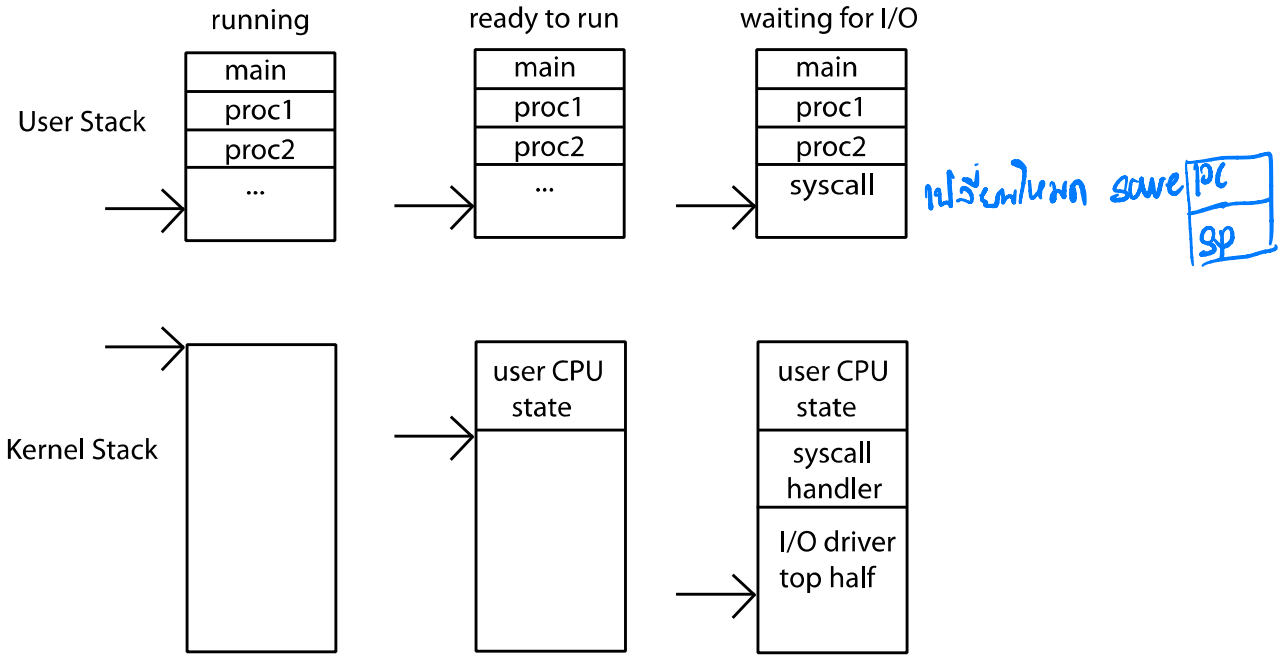
ทำไมไม่ใช้ user stack เพียงแค่อันเดียว

เพราะ:

- process อื่น/ตัวเอง ที่ไม่เกี่ยวข้อง มีสิทธิ์เข้ามาที่ frame เข้าถึง state ของตัวเอง
  - เสถียรภาพ ระบบจะหยุด
  - security Hacker เข้า

The Kernel Stack

- 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



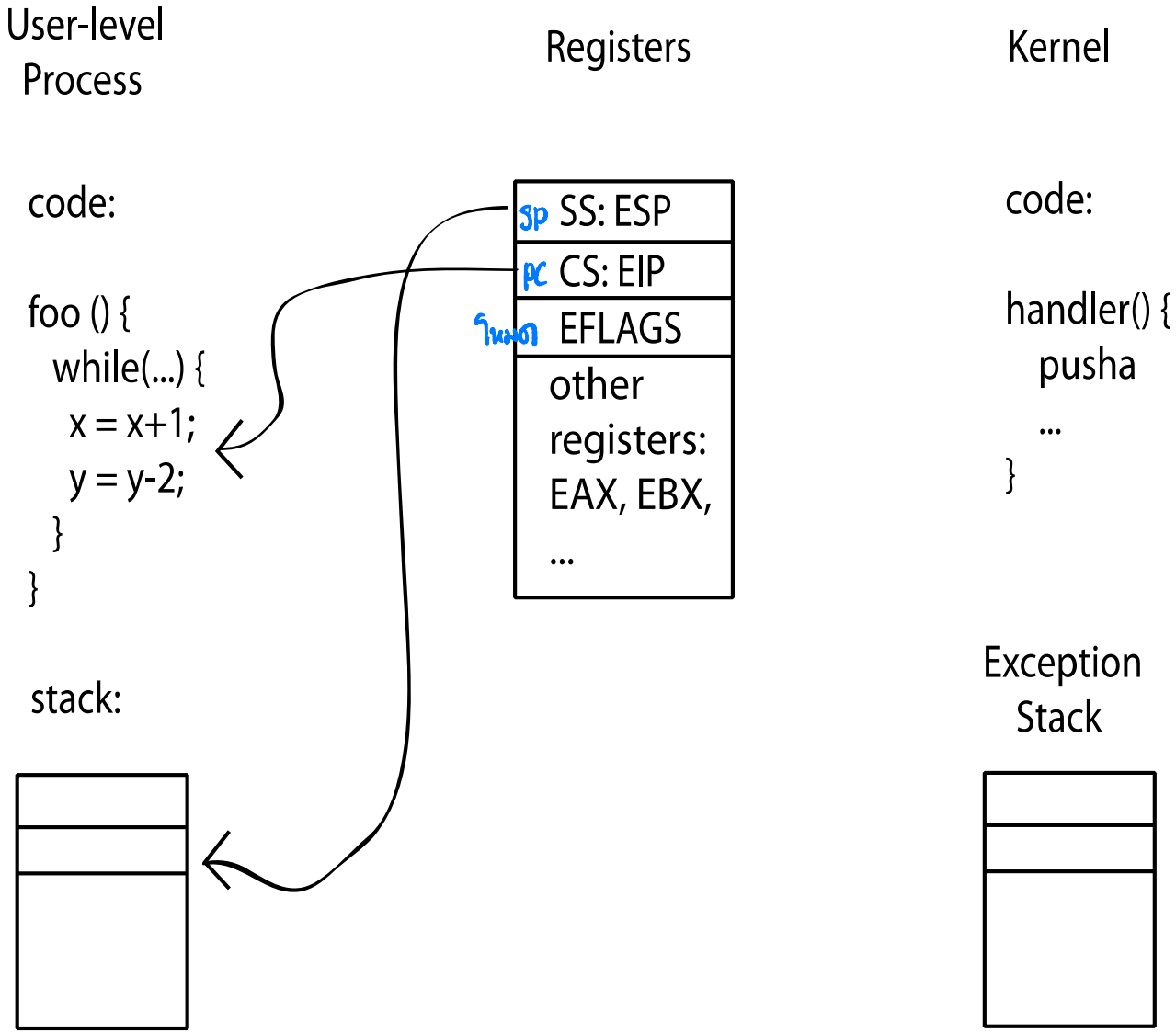
Interrupt Stack

- 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?

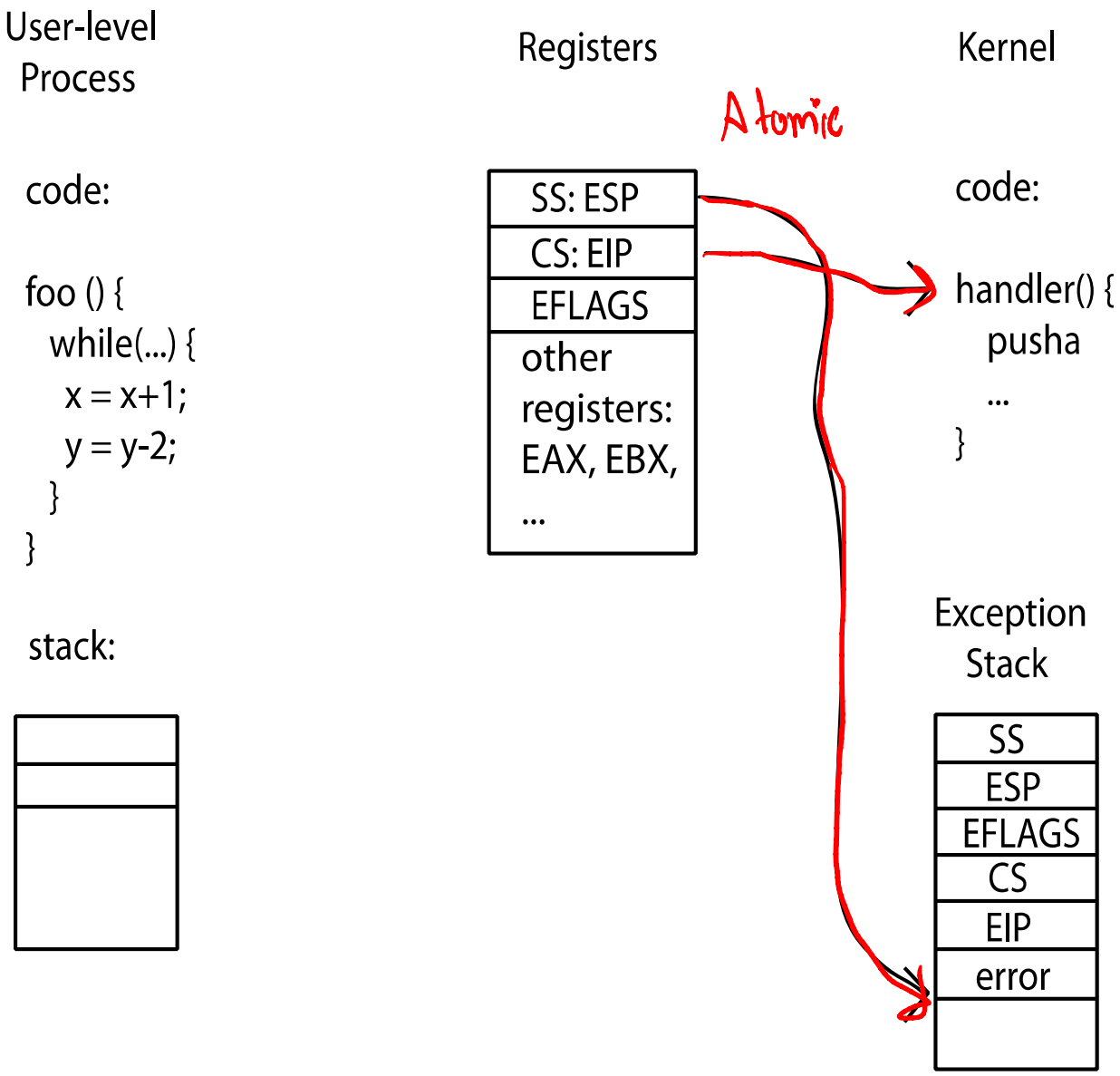
Case Study: x86 Interrupt

- 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

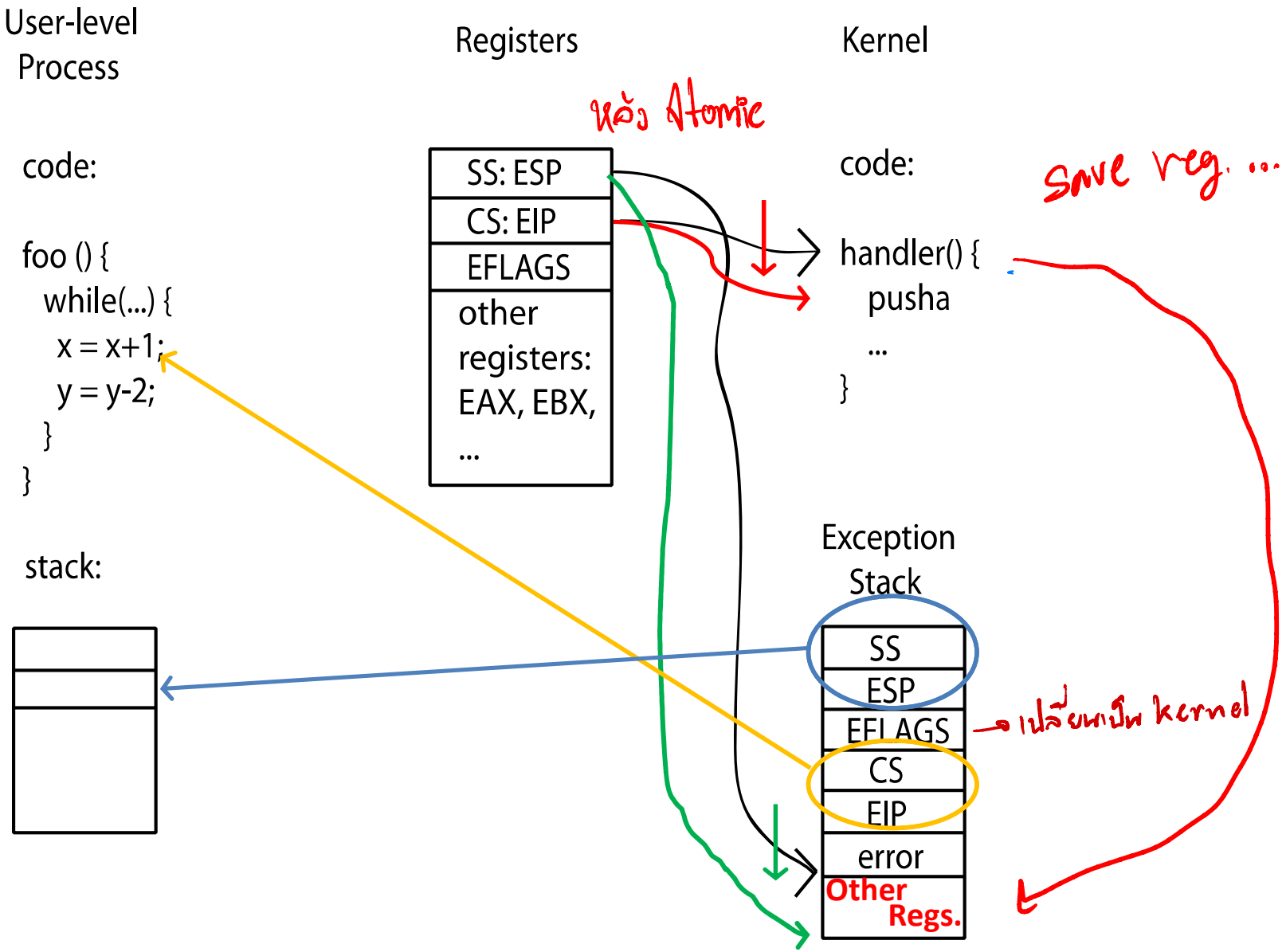
Before Interrupt



During Interrupt



After Interrupt



At end of handler

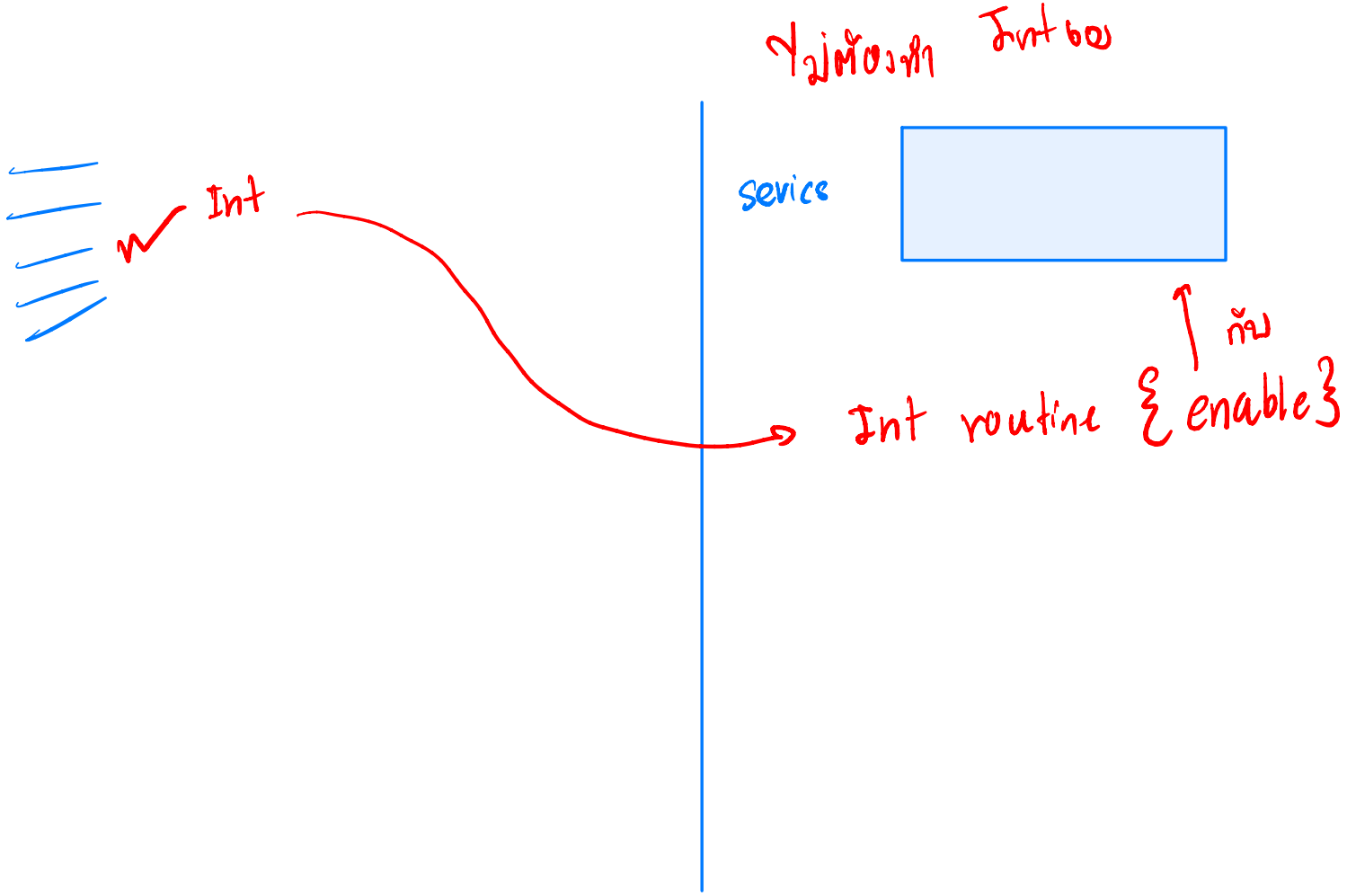
- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

Interrupt Masking

- 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


ไม่ต้องการ Int ตอน
Int
sevice
กลับ
Int routine {enable}

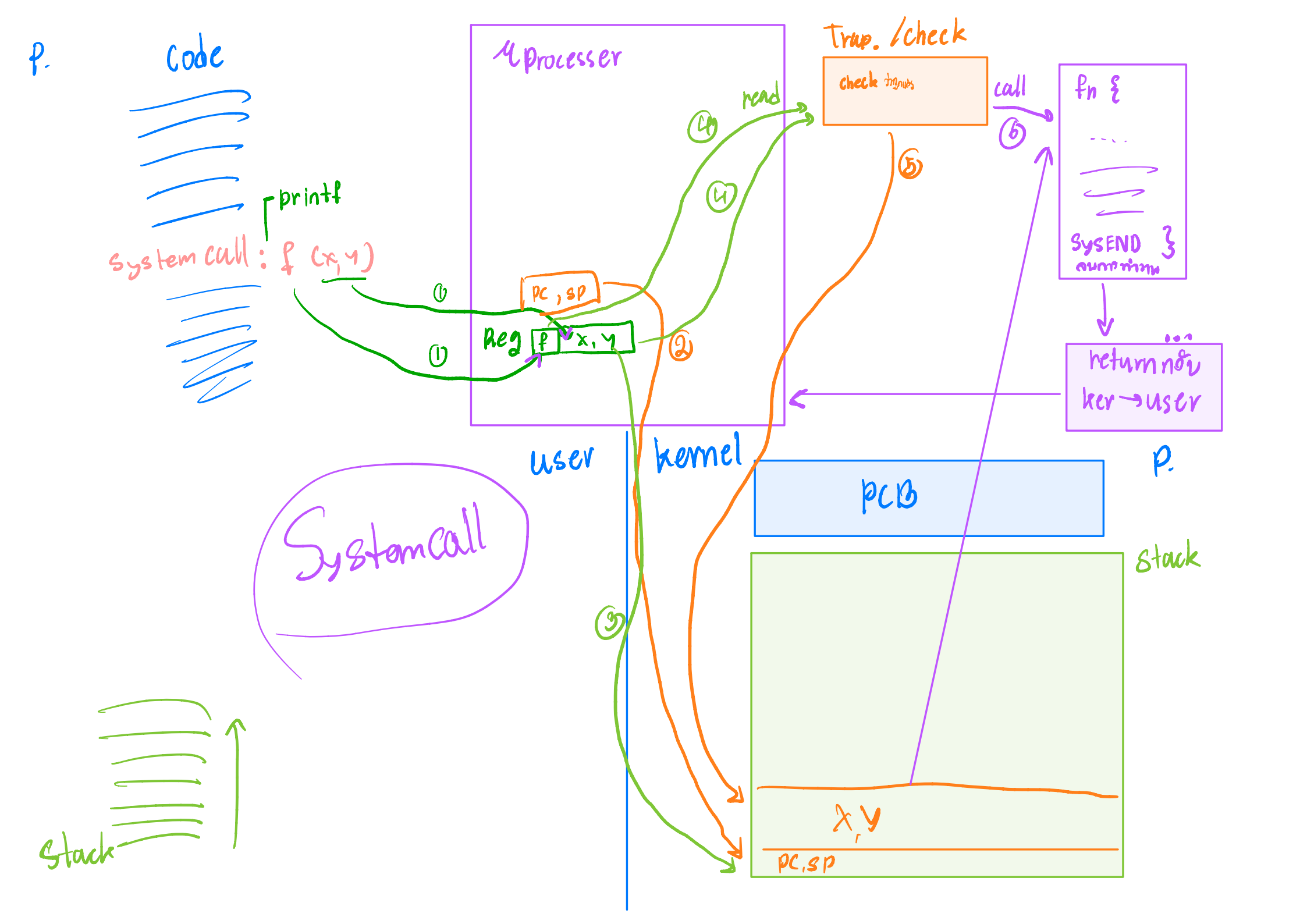
Hardware support: Interrupt Control

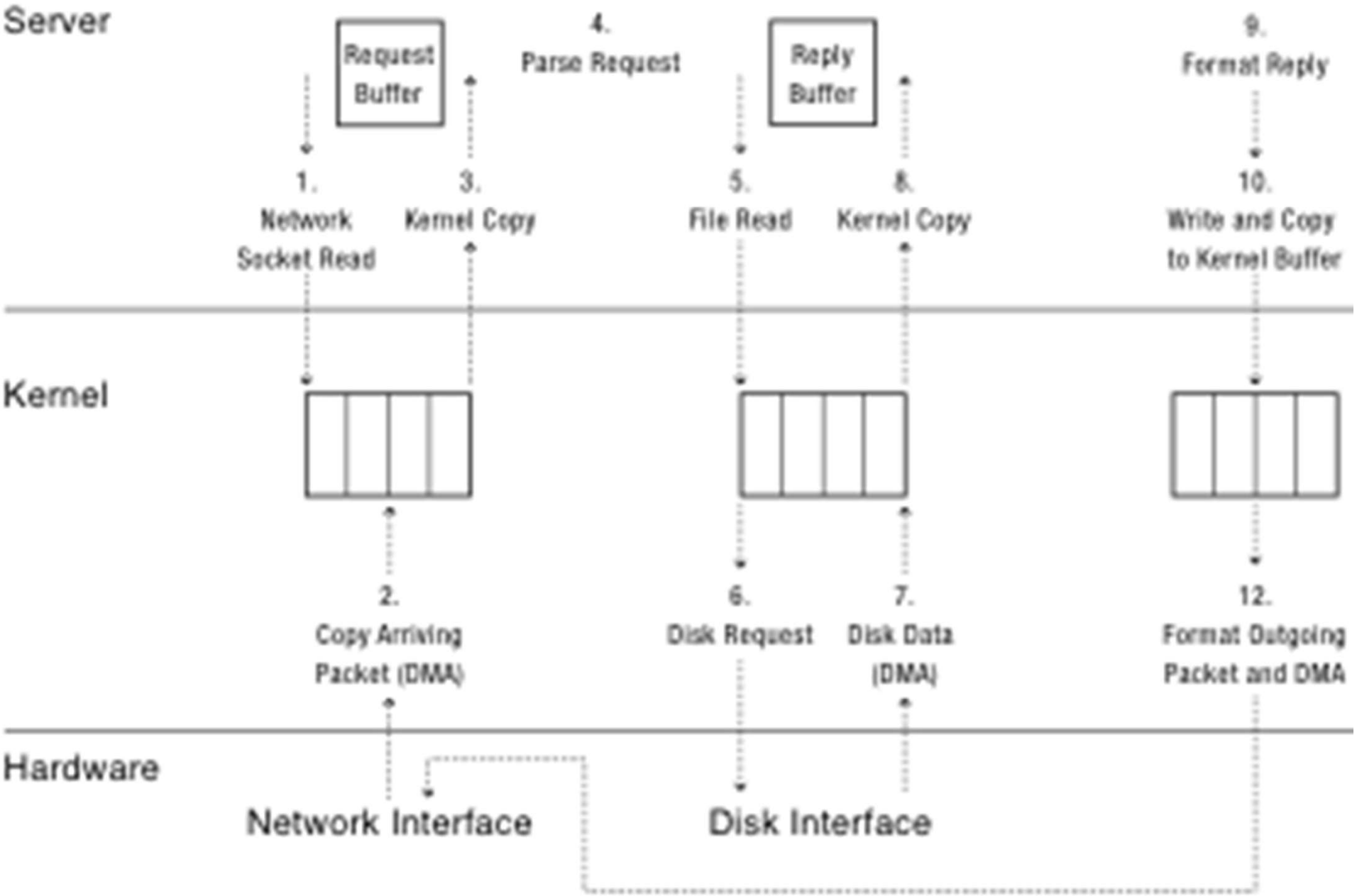
- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall

- 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!

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!


P.
Code
printf
system call : f (x, y)
Processer
Trap. /check
check
call
fn {
SysEND }
read
①
②
③
④
⑤
⑥
PC , SP
Reg
f
x, y
return ที่
ker → user
P.
user
kernel
Systemcall
PCB
Stack
x, y
PC, SP
Stack


Server
Request Buffer
4.
Parse Request
Reply Buffer
9.
Format Reply
1.
Network Socket Read
3.
Kernel Copy
5.
File Read
8.
Kernel Copy
10.
Write and Copy to Kernel Buffer
Kernel
2.
Copy Arriving Packet (DMA)
6.
Disk Request
7.
Disk Data (DMA)
12.
Format Outgoing Packet and DMA
Hardware
Network Interface
Disk Interface

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Program Counter, Registers, Execution Flags, Stack
- **Address space** (with **translation**)
  - 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**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other