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

- 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

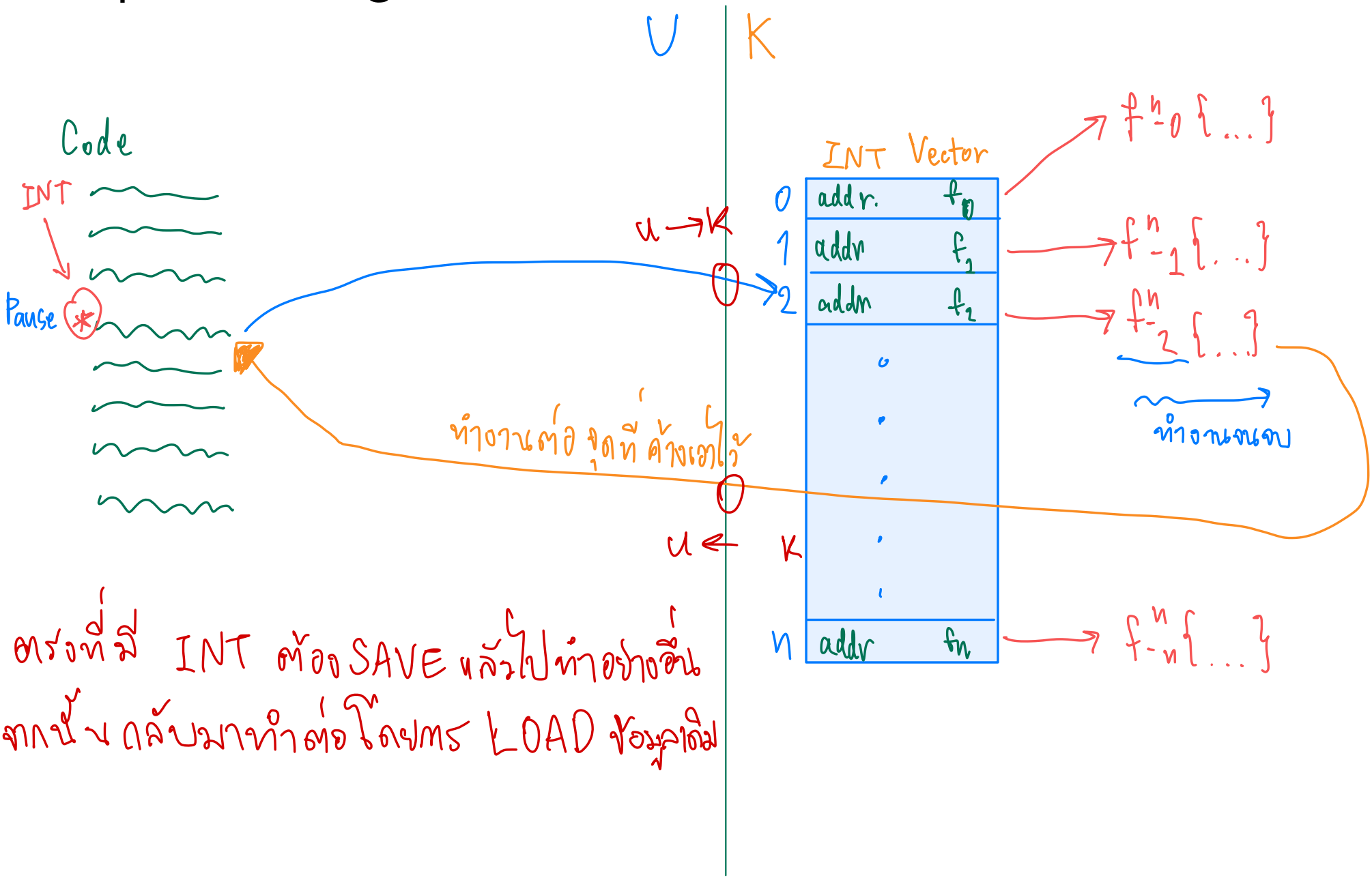
Mode Switch

- From kernel mode to user mode
  - 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



ตรงที่มี INT ต้อง SAVE แล้วไปทำอย่างอื่น
พอทำเสร็จกลับมาทำต่อโดยการ LOAD ข้อมูลเข้ม

Implementing Safe Kernel Mode Transfers

- *Carefully* constructed kernel code packs up the user process state and sets it aside
- Must handle weird/buggy/malicious user state
  - 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)*

Device Interrupts

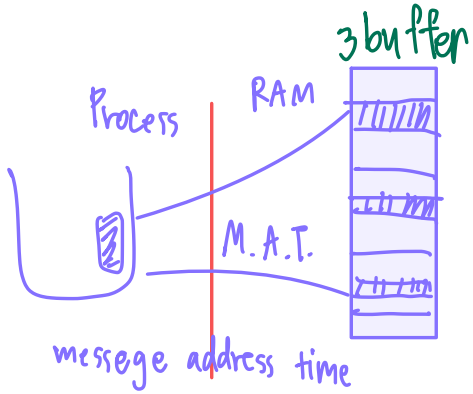
เกี่ยวกับ HW

- 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 Microprecess ข้ามเองรู้(ที่ละ byte)
  - Direct memory access (DMA) by device copy ข้อมูลจาก RAM ไปอีกที่ใน RAM
  - Buffer descriptor: sequence of DMA's
    - E.g., packet header and packet body
  - Queue of buffer descriptors เอามาเรียงทำแบบ Array
    - Buffer descriptor itself is DMA'ed

Messege ที่บอก ตน.ใช้ม หลา จุดสิ้นสุด เพื่อให้ง่ายต่อ ตรย้ายข้อมูล (ค่อนข้างกระจาย) หากมี หลาย ตัวสร้างหลาย รอบ

ระหว่าง Buffer Micro ทำอย่างอื่นก่อน copy เสร็จส่ง INT ให้

Micro รู้ว่าเสร็จแล้ว จะได้ copy อันถัดไปหรือ ทำอย่างอื่นต่อไป



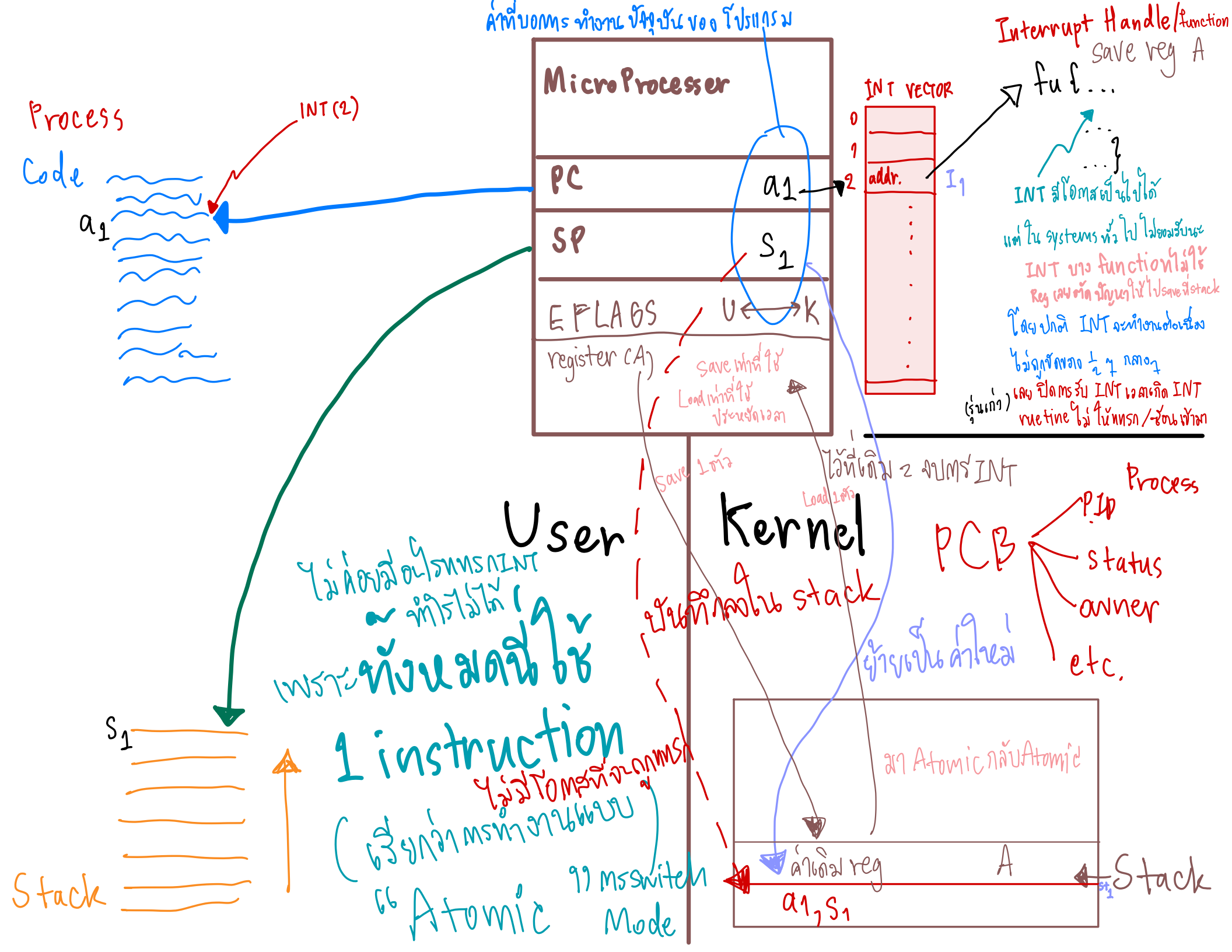
Activity #2

- How do device interrupts work? เกิด INT กระโดดมา INT Vector แล้ว เรียกใช้ function ที่เกี่ยวข้อง กระโดดที่เดิม
  - Where does the CPU run after an interrupt?
  - What stack does it use? เกิดที่ไหน kernel หรือ User
  - Is the work the CPU had been doing before the interrupt lost forever? งานที่ประมวลผลหายไปมั้ย Ans ไม่หาย State ถูก Save ไว้
  - If not, how does the CPU know how to resume that work? ยังไงถ้า Save State แล้ว สั่งให้ load กระทำงานต่อไปข้างหน้า

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

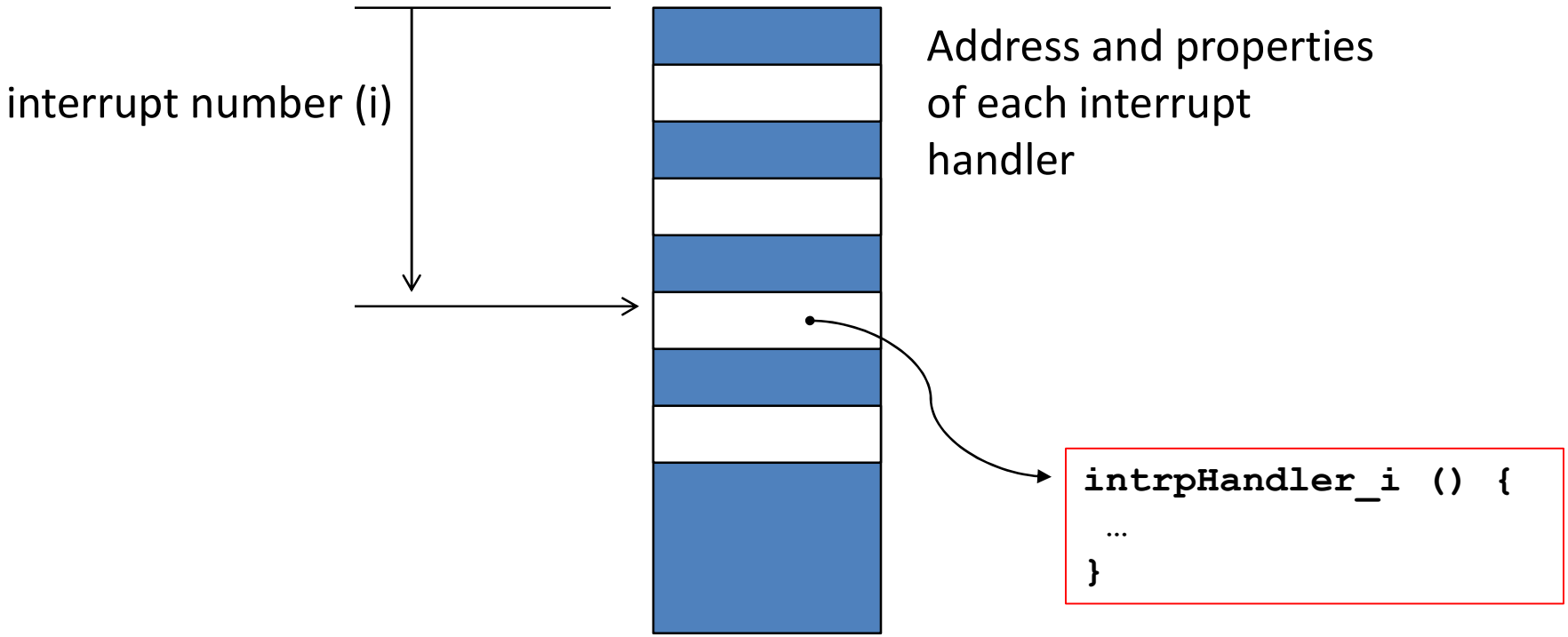

ค่าที่บอกการทำงานปัจจุบันของโปรแกรม
Process
Code
a_1
INT (2)
MicroProcesser
PC
a_1
SP
S_1
EFLAGS
U ↔ K
register (A)
save เท่าที่ใช้
Load เท่าที่ใช้
ประหยัดเวลา
INT VECTOR
0
1
2
addr.
I_1
Interrupt Handle/function
save reg A
fu{...
...}
INT มีโอกาสเป็นไปได้
แต่ใน systems ทั่วไป ไม่ยอมรับนะ
INT บาง function ไม่ใช้
Reg เลย ตัดปัญหาไม่ต้องไปsaveที่stack
โดยปกติ INT จะทำงานต่อเนื่อง
ไม่ถูกขัดจังหวะ $\frac{1}{2}$ ๆ กลางๆ
(รุ่นเก่า) เลย ปิดการรับ INT เวลาเกิด INT
routine ไม่ให้มีการรบกวน
ไว้ที่เดิม 2 ขบการ INT
Save 1ตัว
Load 1ตัว
User
Kernel
ไม่ค่อยสนโปรแกรมINT
~ ทำอะไรไม่ได้
เพราะทั้งหมดนี้ใช้
1 instruction
ไม่มีโอกาสที่จะถูกขัดจังหวะ
(เรียกว่าการทำงานแบบ
"Atomic
ๆ การswitch
Mode
ปัญหาที่กองใน stack
ย้ายเป็นค่าใหม่
Process
PCB
PID
status
owner
etc.
สร้าง Atomic กลับ Atomic
ค่าเดิม reg
A
Stack
a_1, S_1
S_1
Stack

interrupt ruetine = ตรทำงาน ใช้ลงขึ้นอยู่กับงานว่าซับซ้อนหรือไม่

- มีตัวแปร ใช้ registor ในการเก็บค่า มากกว่า 1 registor (เป็นชุด) คำสั่งแรกจะเป็น save register A เป็นตัวแปรไรก็ได้ save ลงไปใน stack
-

Where do mode transfers go?

- Solution: ***Interrupt Vector***



The Kernel Stack

ฝั่ง Kernel

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

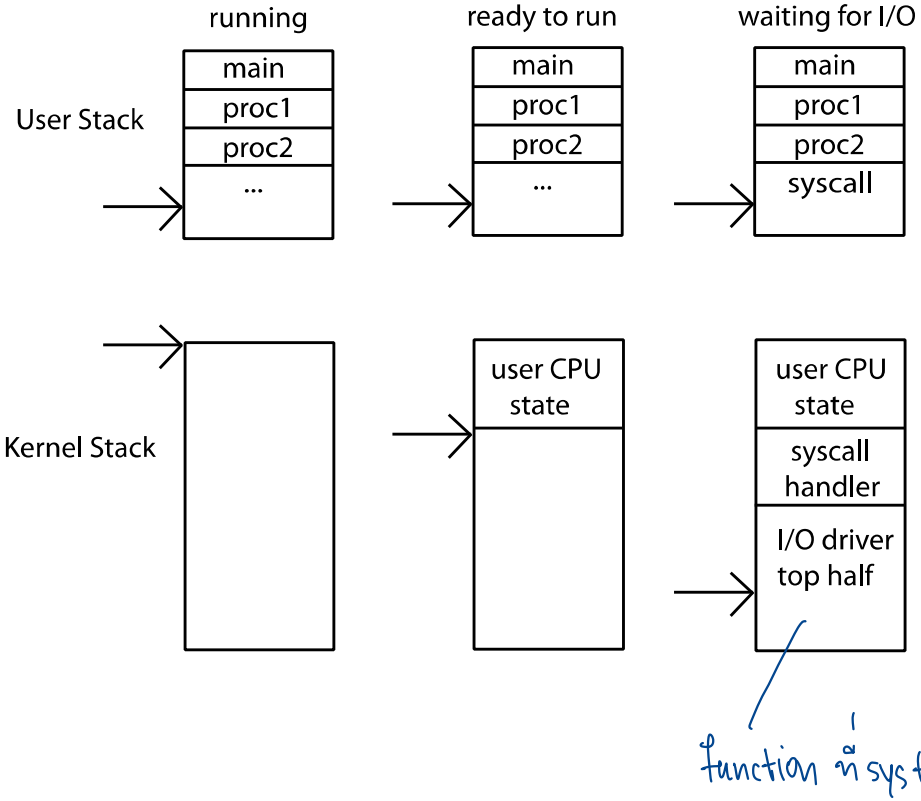
- Security
A. เสียประสิทธิภาพของระบบ ระบบช้าลง ยาก

สามารถ detect การ INT ได้

ใน 1 process มี หลาย thread เวลา เกิด INT thread ตัวอื่นก็ ทำงานได้ ตรวจเช็ค INT ที่จะ detect ค่าที่ไม่ใช่ ถ้ามี ตัวแปร ที่อยู่ตัวแปรเก็บใน stack หากอยู่ User เหมือนกัน ก็ช่วย กัน แก้ไขได้

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



ปกติ มี 1 stack ต่อ 1 process

มี 2 stack ต่อ 1 kernel

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?

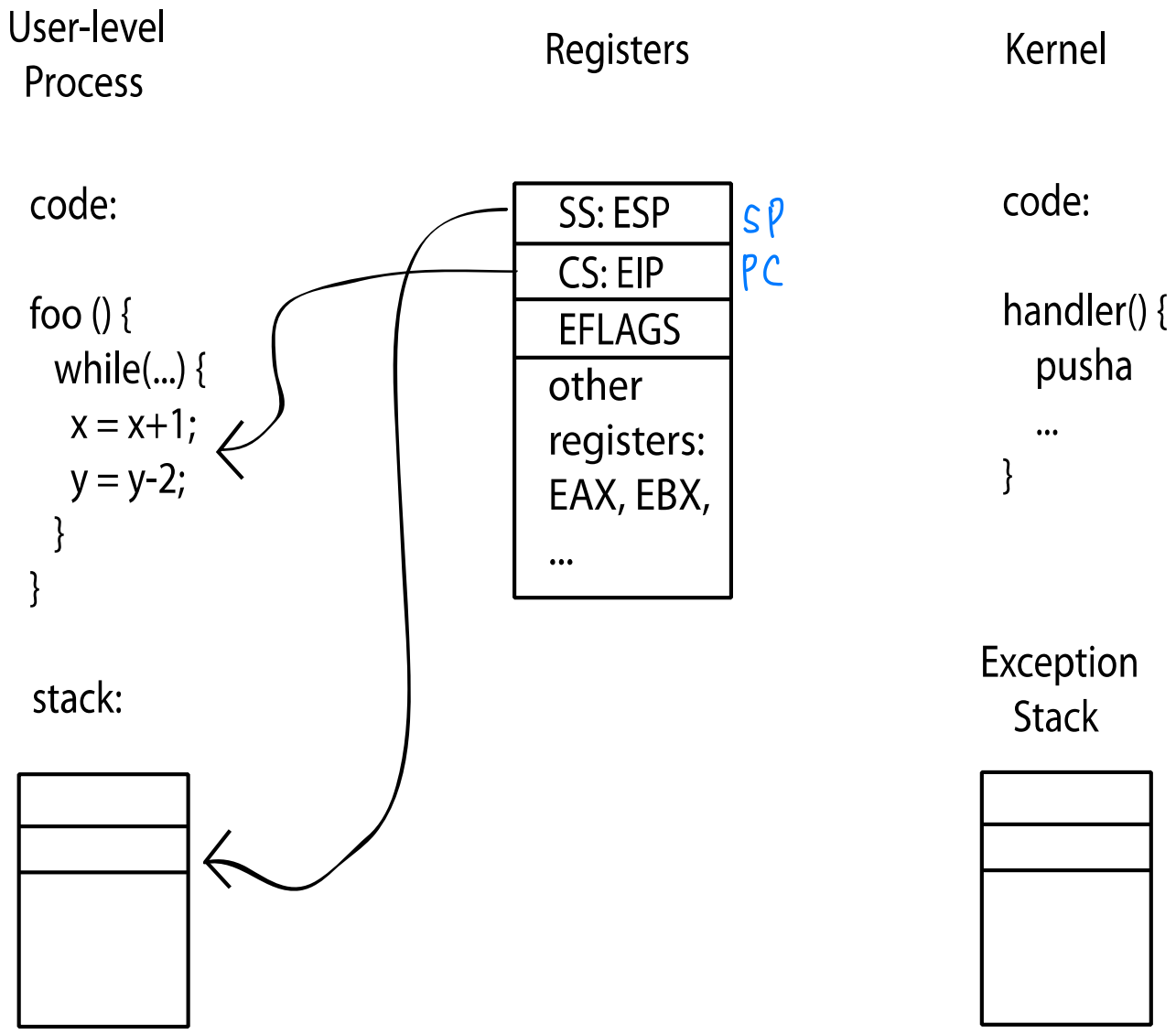
Case Study: x86 Interrupt

- 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

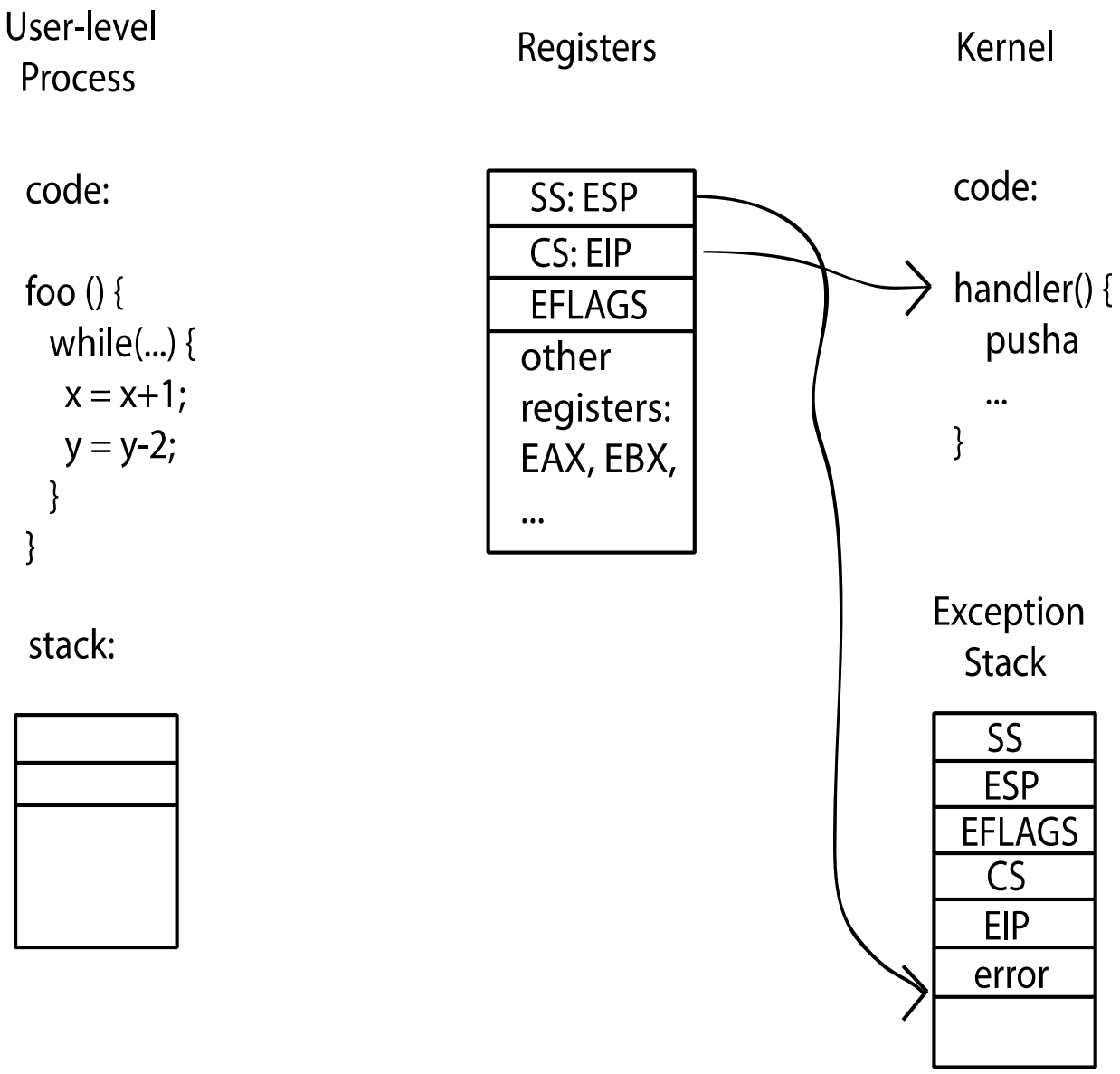
Atomic Instruction
เครื่องใน 1 Instruction

Save reg เท่าที่ใช้
Save ของ load ของเข้าใหม่

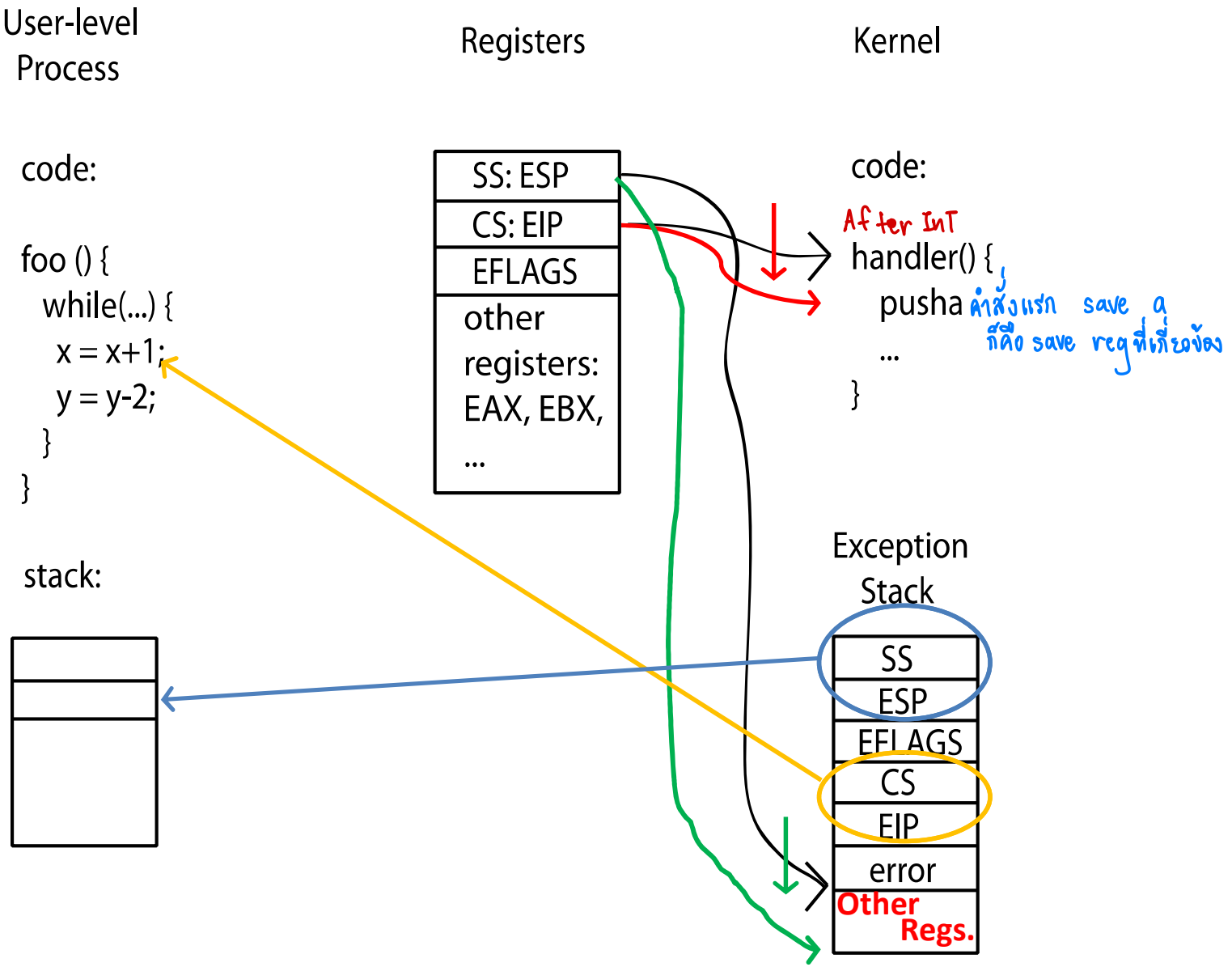
Before Interrupt



During Interrupt

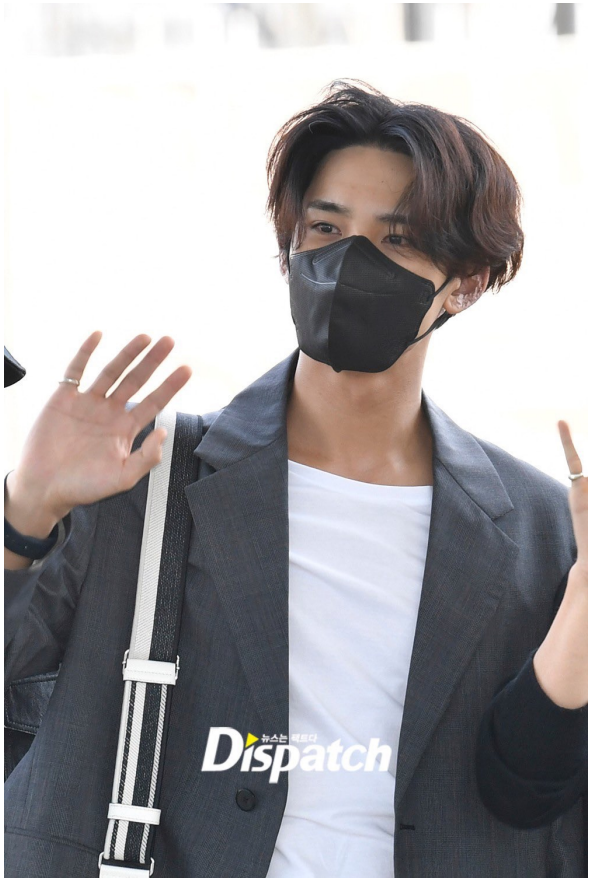


After Interrupt



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



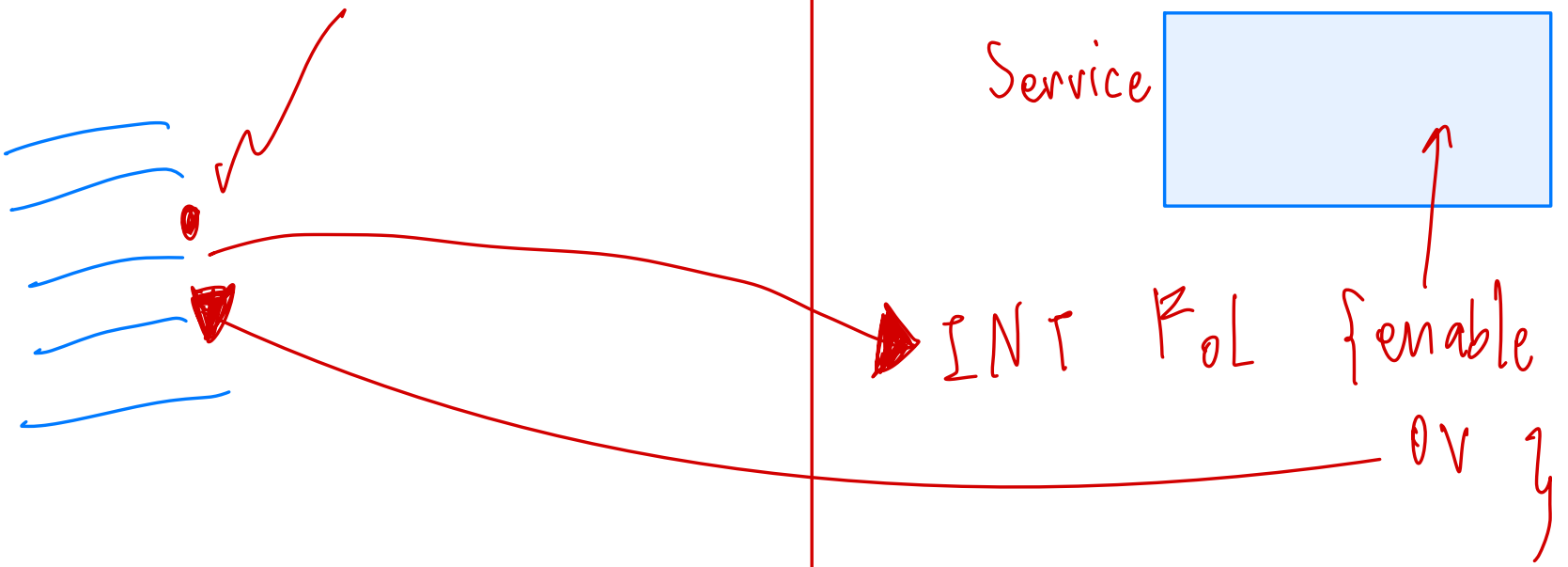
Interrupt Masking

ปกติไม่ยอมให้เกิด interrupt ซ้อน โดยการปิดการรับ INT

- 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 (ของ) x86
    - CLI: disable interrrupts (ปิด)
    - STI: enable interrupts (รับ)
    - Only applies to the current CPU (on a multicore) (มีผลเฉพาะแต่ละ core ที่ทำงานอยู่ ตัวอื่นทำงานต่อ)
- 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


Service
INT FoL {enable
ov

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

INT ประเภท สูงสุด

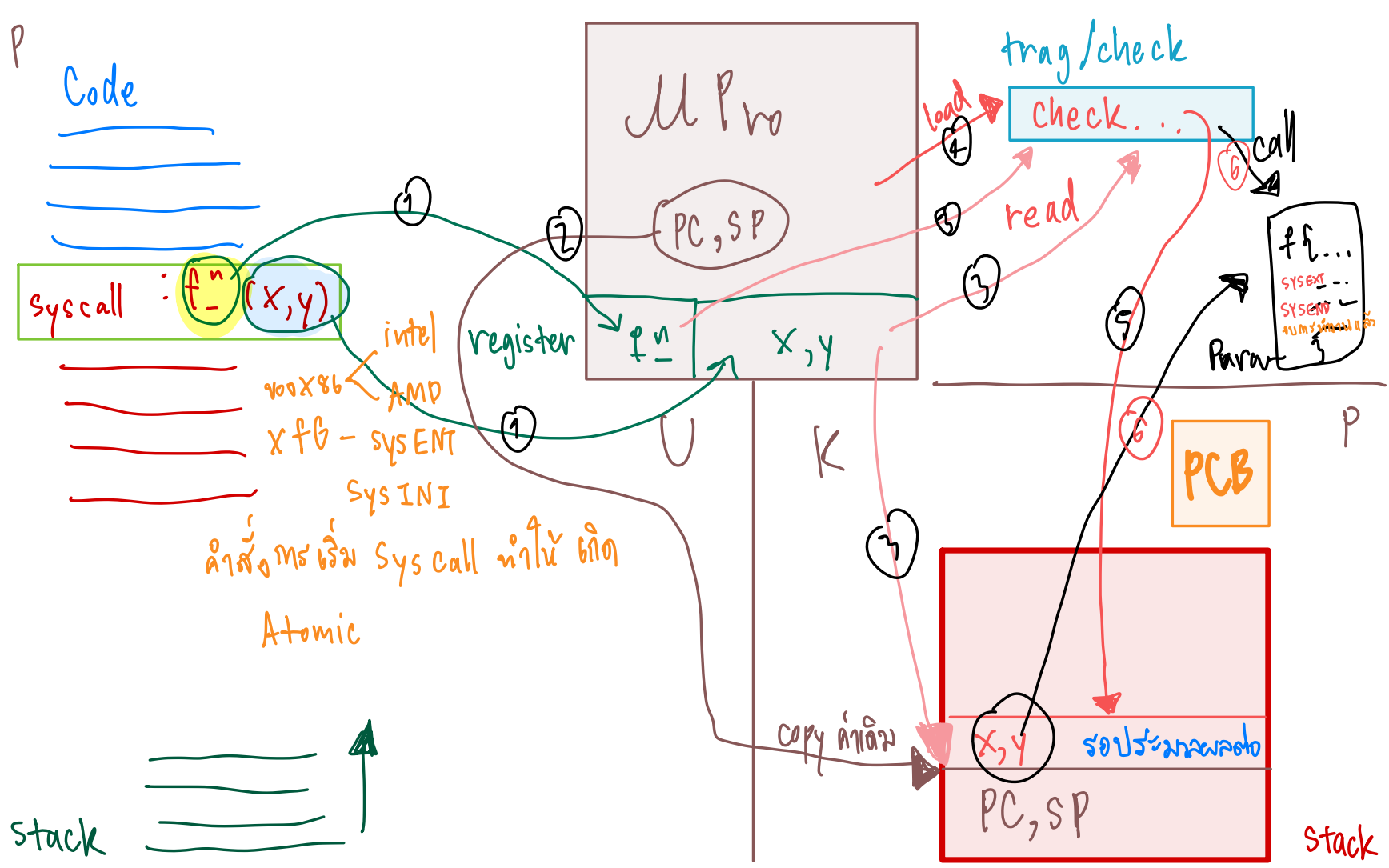
มักเกิดกรณีระบบมีปัญหา ส่งผลต่อประสิทธิภาพการทำงาน

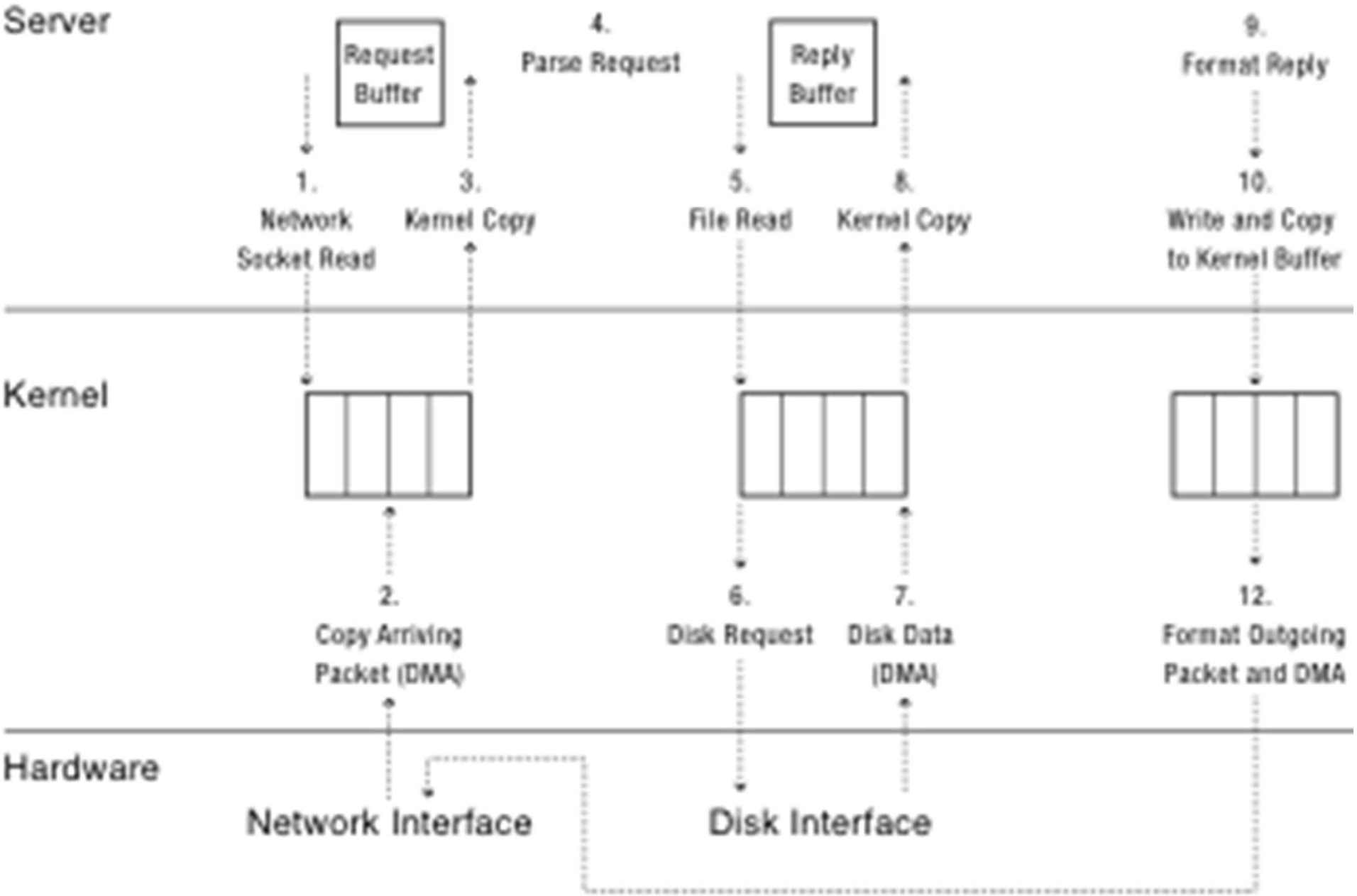
(แบตจะหมด เตือนผู้ใช้งานให้ SAVE งาน)

INT Level สูง สามารถ INT ชั้น Level ต่ำกว่าได้

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
syscall : fn (x,y)
intel
ของx86
AMD
xf6 - sysENT
SysINI
คำสั่งที่ทำให้เริ่ม Sys call ทำให้ เกิด
Atomic
stack
MPro
PC,SP
register
fn
x,y
U
K
trap/check
load
check...
call
read
fn...
SYSEXIT
SYSENT
Parar
P
PCB
copy ค่าเดิม
x,y
รอประมวลผลต่อ
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

Virsual Addr = Logic Addr.

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