

#OperatingSystemByNeso #os #OperatingSystem

## System Calls

การดู 282,254 ครั้ง • 15 มี.ค. 2018 • Operating System: System Calls...

แสดงเพิ่มเติม

# The Kernel Abstraction

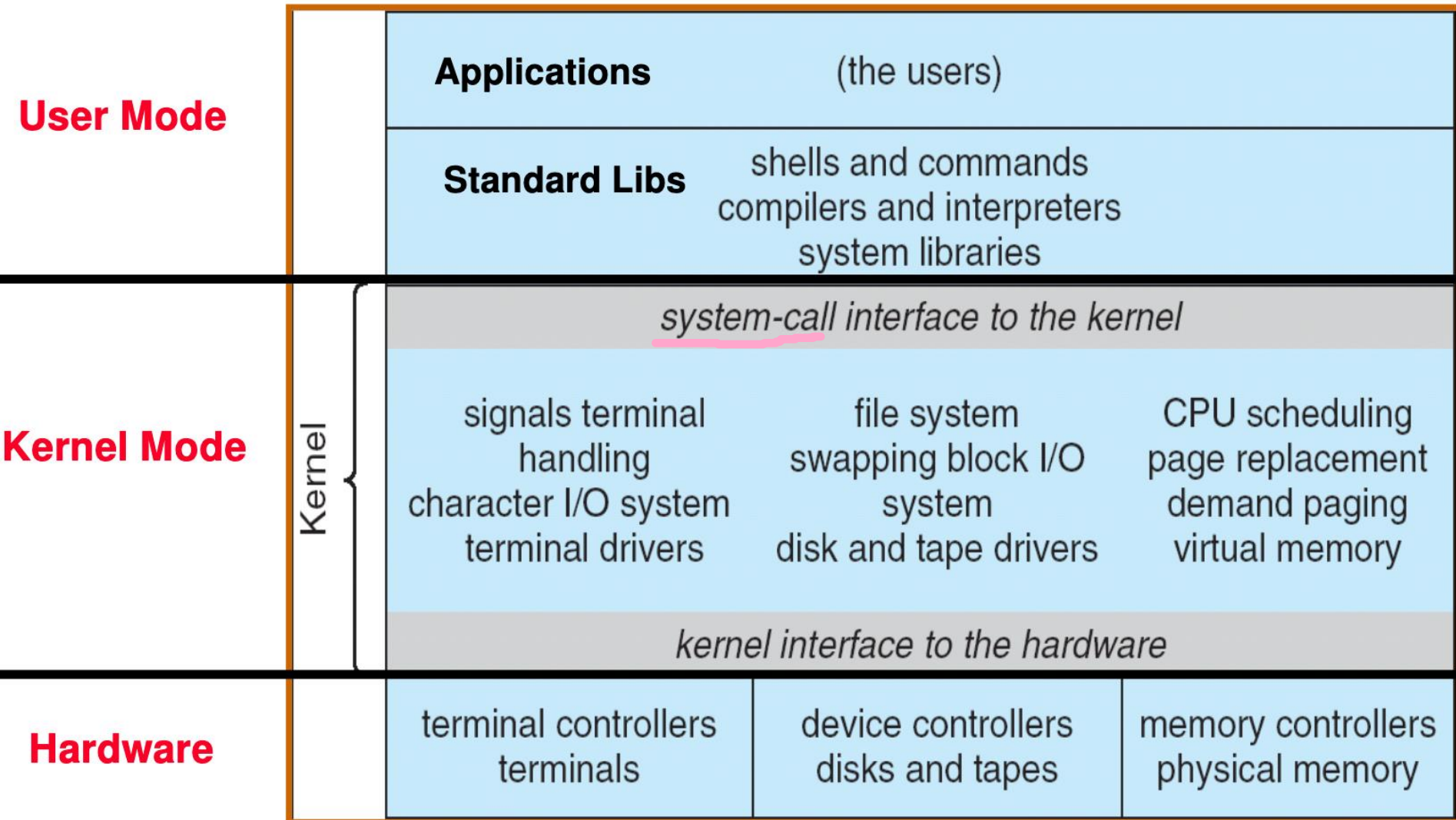
ไปดูคลิปมาอีกทีเพราะว่าจด (แค็บ) มาไม่หมด

# What is an OS...

- Hiding Complexity
  - Variety of HW
    - E.g. different CPU, amount of RAM, I/O devices
- **Kernel** is the part of the OS that running all the time on the computer
  - Core part of the OS
  - Manages system resources
  - Acts as a bridge between apps and HW

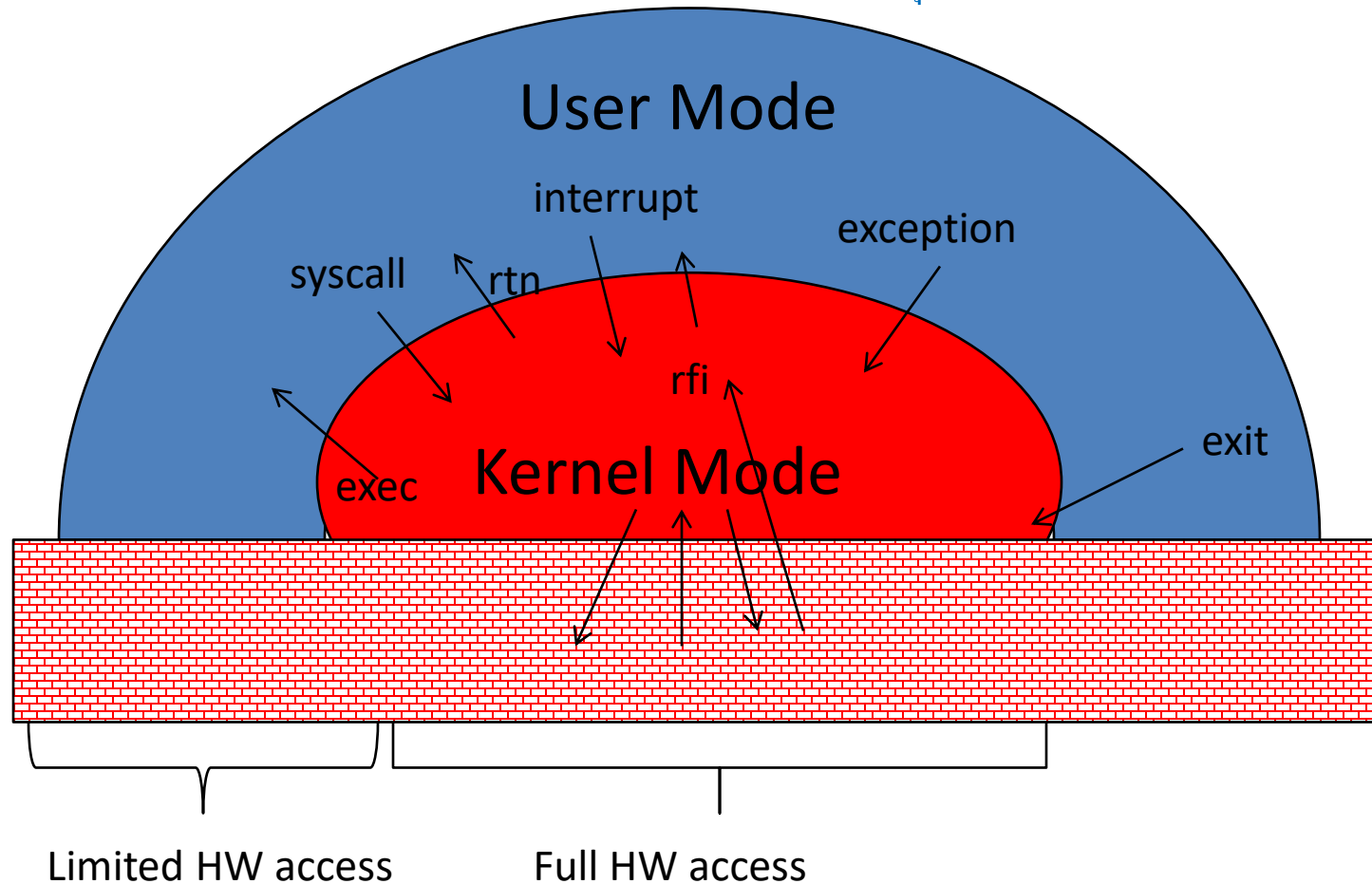
# UNIX Structure

แบบปัจจุบันจะแยกชั้นชัดเจน  
รองรับ Multitask Multituser



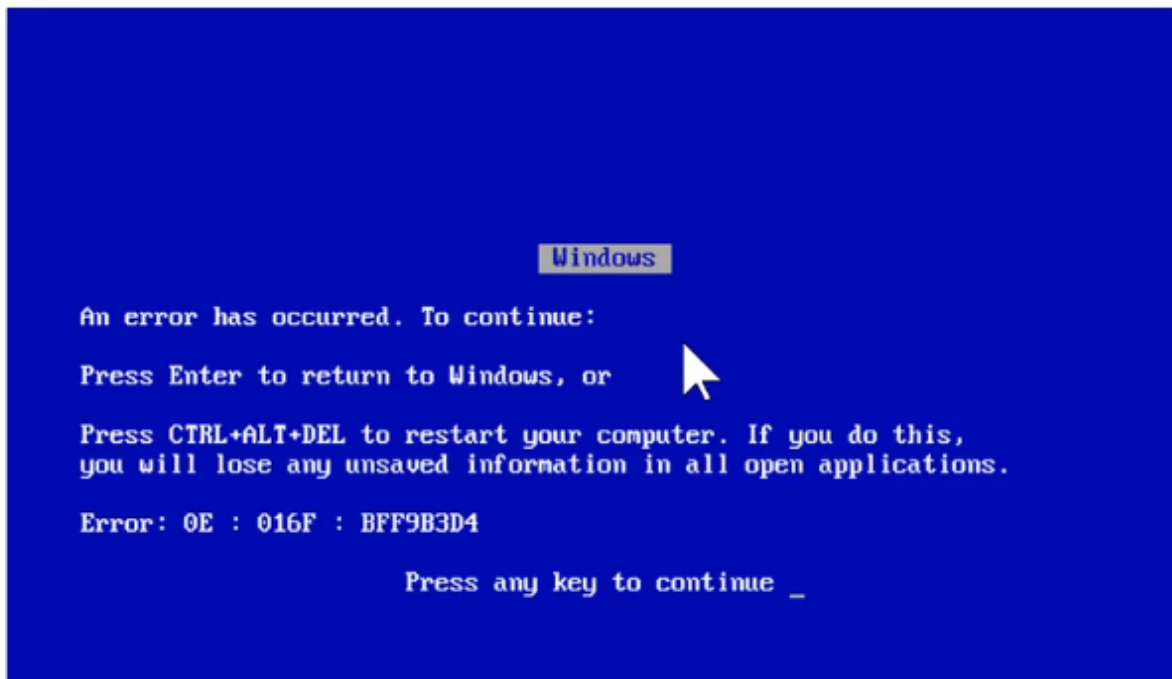
# User/Kernel (Privileged) Mode

แบบยุคเก่าที่ User เข้าถึง HW ได้โดยไม่ต้องผ่าน Kernel



# One of the major goals of OS is...

- Protecting **Process** and the **Kernel**
  - Running multiple programs
    - Keep them from interfering with the OS – kernel
    - Keep them from interfering with each other



จอฟ้านี้คือขั้นสุดแล้ว  
มีการรบกวนกันของโปรแ  
กรมสองอัน  
หรือโปรแกรมกับ Kernal  
แล้ว Os หยุดไม่ได้

# Protection: WHY?

เวลา 10 นาที

- **Reliability**: buggy programs only hurt themselves
  - **Security** and privacy: trust programs less
  - **Fairness**: enforce shares of disk, CPU
- SW
- HW

# Protection: How? (HW/SW)

เวลา 10 นาที

- 2 Main HW mechanism

- Memory address translation
- Dual mode operation

- Privileged/non-privileged mode
- System calls

- SW

- Process
- System calls

# Hardware Support: Dual-Mode Operation

- Kernel mode
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
  - Limited privileges
  - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register
- On the MIPS, mode in the status register




# โจทย์

- ในปัจจุบัน น.ศ. คิดว่า HW (CPU) supports การทำ Dual-mode Operation อย่างไรบ้าง

เวลา 10 นาที

# Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - To prevent user code from overwriting the kernel
- Timer
  - To regain  interrupt control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

# Privileged instructions

- Examples?
- What should happen if a user program attempts to execute a privileged instruction?

# User Mode

- Application program
  - Running in process

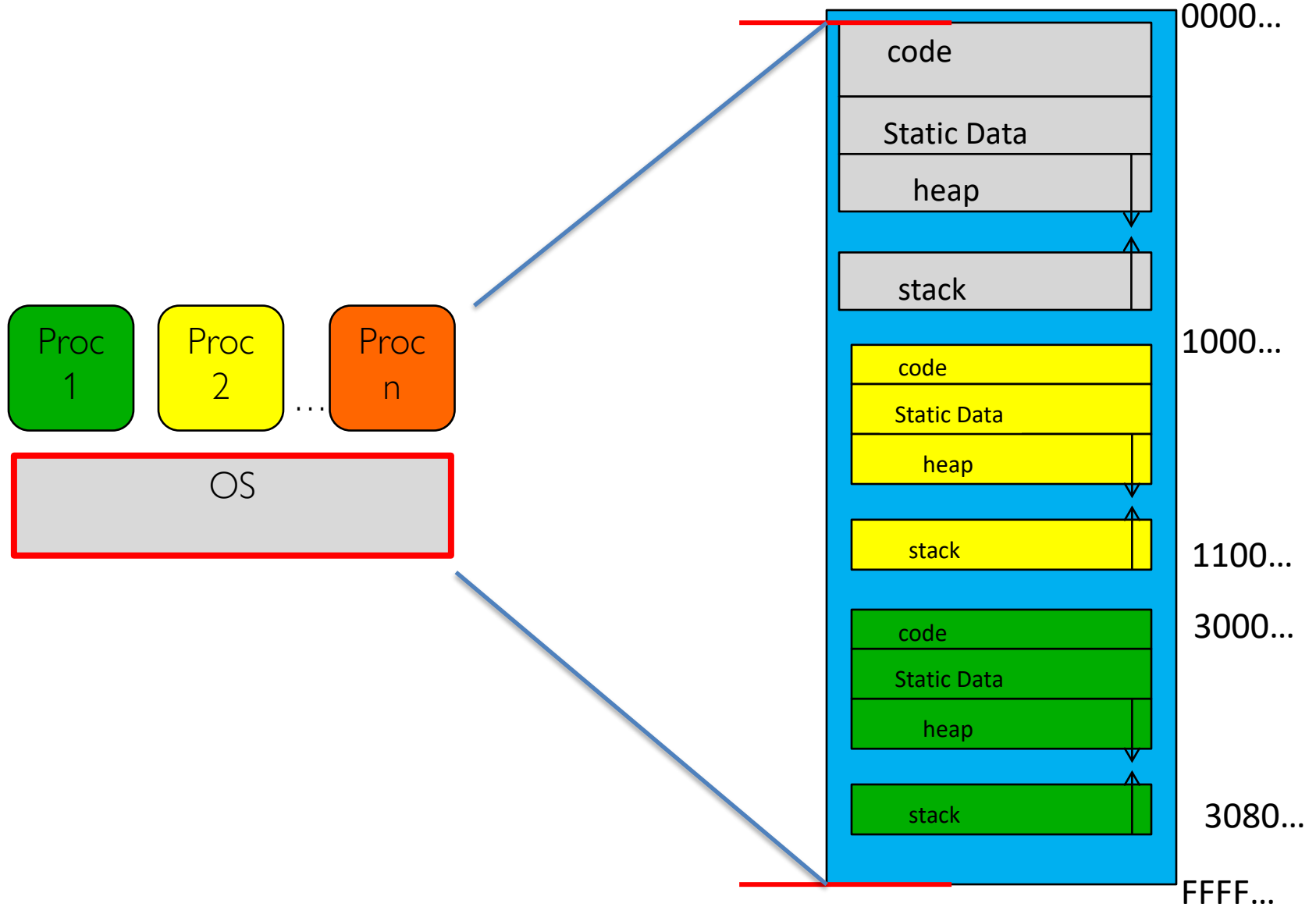
# Virtual Machine:VM

- Software emulation of an abstract machine
  - Give programs illusion they own the machine
  - Make it look like HW has feature you want
- 2 types of VM
  - Process VM
    - Supports the execution of a single program (one of the basic function of the OS)
  - System VM
    - Supports the execution of an entire OS and its applications

# Process VMs

- GOAL:
  - Provide an isolation to a program
    - Processes unable to directly impact other processes
      - Boundary to the usage of a memory
    - Fault isolation
      - Bugs in program cannot crash the computer
  - Portability

# Kernel mode & User mode



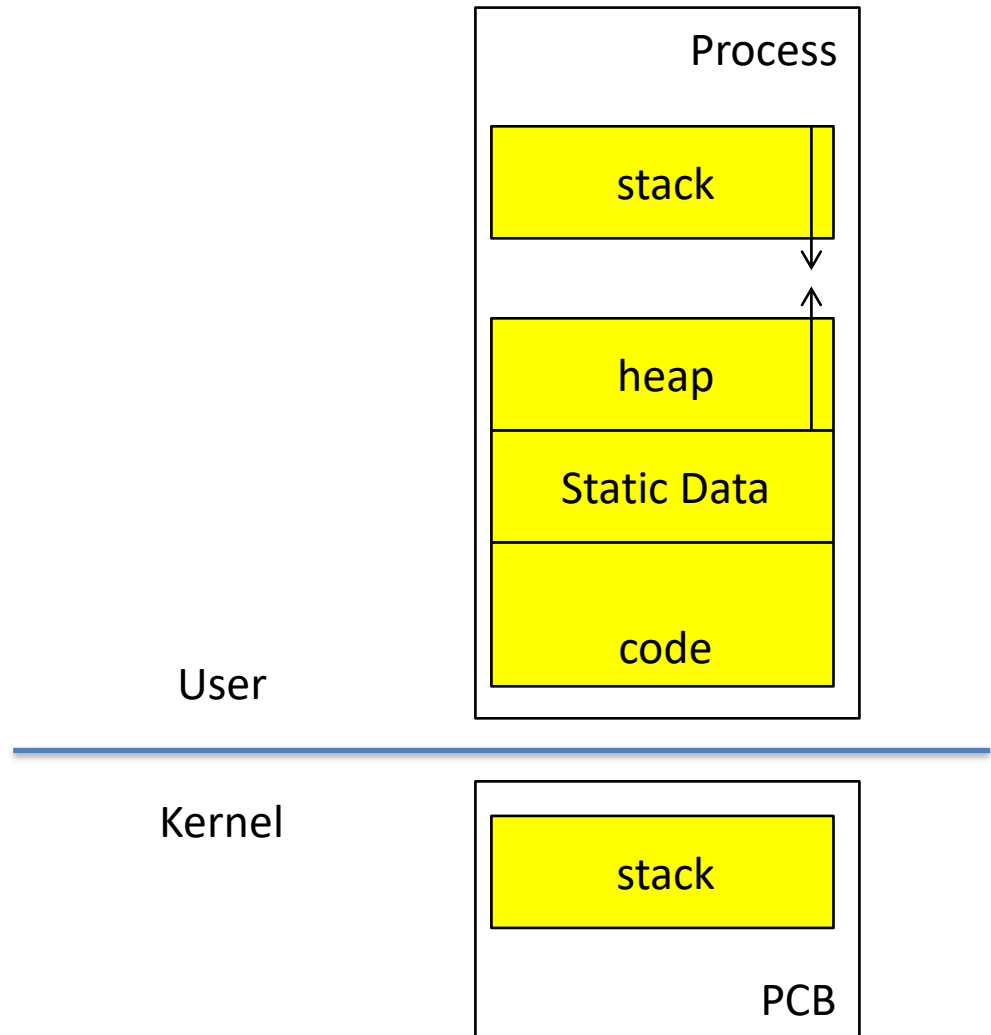
# Process Abstraction

- Process: an *instance* of a program, running with limited rights
  - Thread: a sequence of instructions within a process
    - Potentially many threads per process (for now 1:1)
  - Address space: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)



# Process

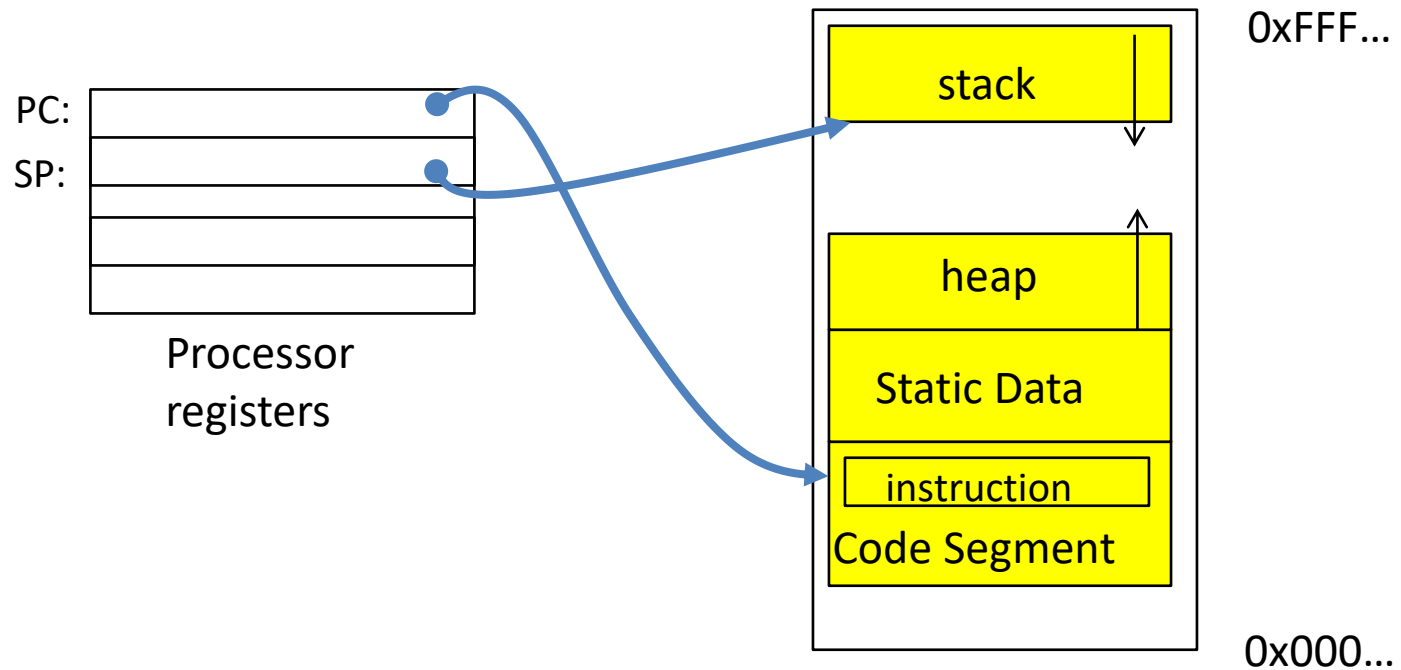
- 2 parts
  - PCB in kernel
  - Others in user



# Process Control Block: PCB

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, ...)
  - Registers, SP, ... (when not running)
  - Process ID (PID), User, Executable, Priority, ...
  - Execution time, ...
  - Memory space, translation tables, ...
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

# Address Space: In a Picture



**Break**

# 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 (เกิดทำงานผิดปกติ)
    - 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

# Implementing Safe Kernel Mode Transfers

เซฟ user state ไว้ก่อนตอน return กลับมาทำ

- *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

- 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

# Device Interrupts

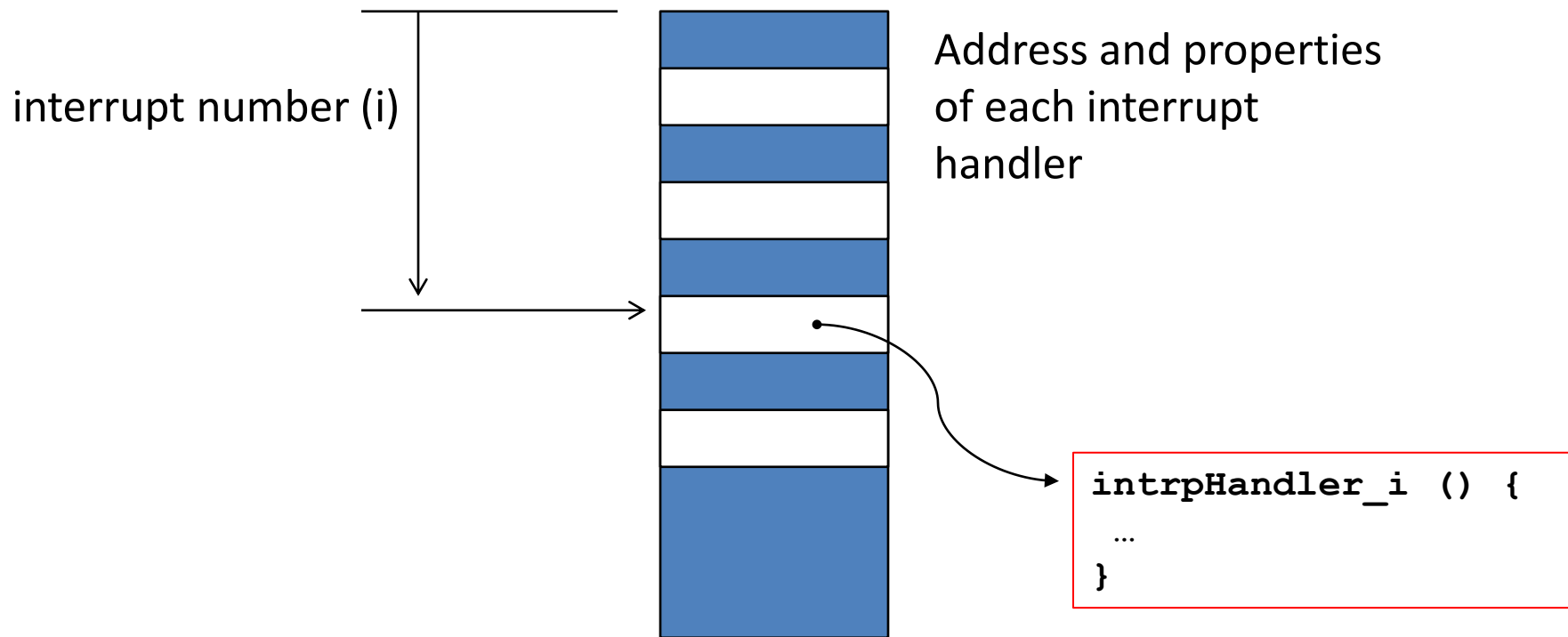
- How do device interrupts work?
  - Where does the CPU run after an interrupt?
  - What stack does it use?
  - Is the work the CPU had been doing before the interrupt lost forever?
  - If not, how does the CPU know how to resume that work?

# 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

# Where do mode transfers go?

- Solution: ***Interrupt Vector***

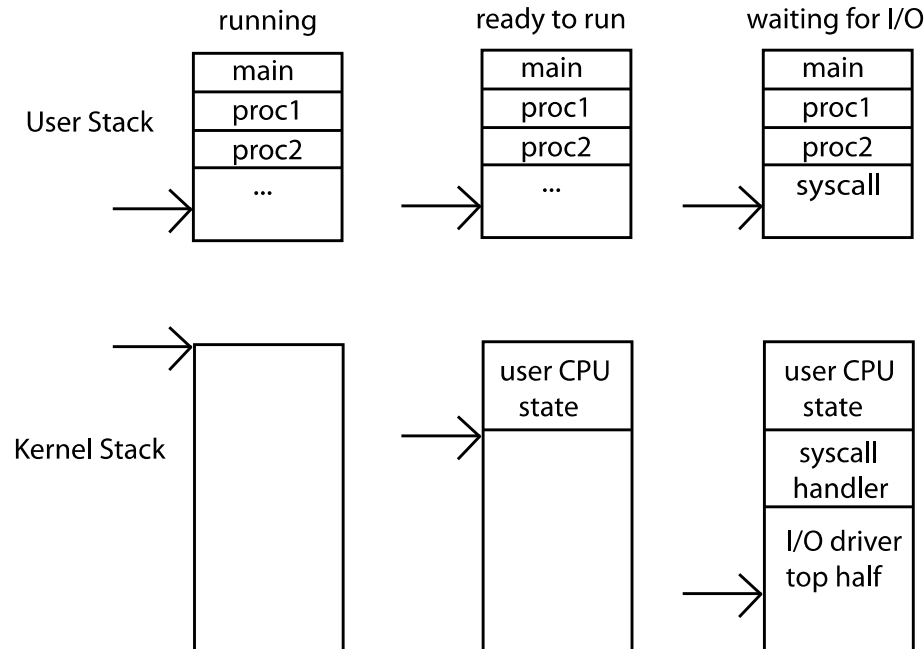


# The Kernel Stack

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

# 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

User-level  
Process

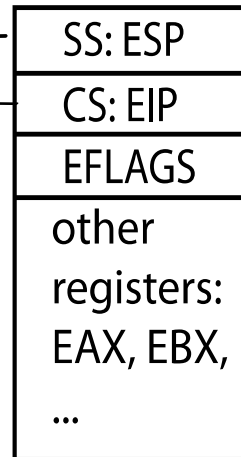
code:

```
foo () {  
  while(...) {  
    x = x+1;  
    y = y-2;  
  }  
}
```

stack:



Registers



Kernel

code:

```
handler() {  
  pusha  
  ...  
}
```

Exception  
Stack



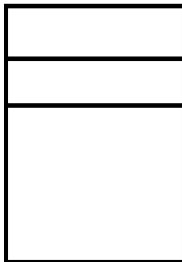
# During Interrupt

User-level  
Process

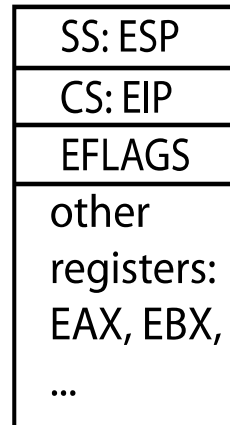
code:

```
foo () {  
  while(...) {  
    x = x+1;  
    y = y-2;  
  }  
}
```

stack:



Registers

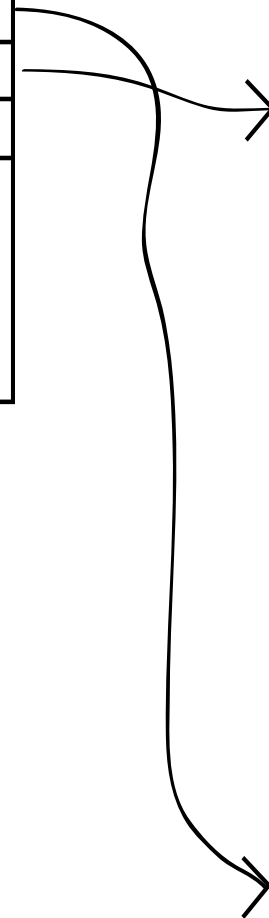
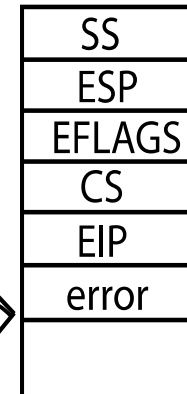


Kernel

code:

```
handler() {  
  pusha  
  ...  
}
```

Exception  
Stack



# After Interrupt

User-level  
Process

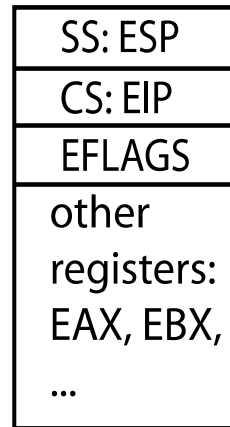
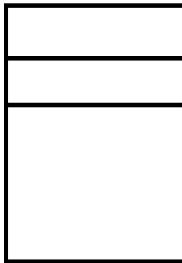
Registers

Kernel

code:

```
foo () {  
  while(...) {  
    x = x+1;  
    y = y-2;  
  }  
}
```

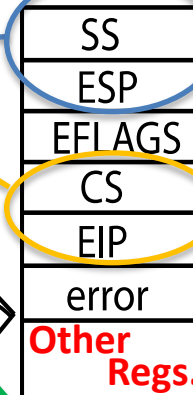
stack:



code:

```
handler() {  
  pusha  
  ...  
}
```

Exception  
Stack



# 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 interrupts
    - 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
- 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!

BREAK



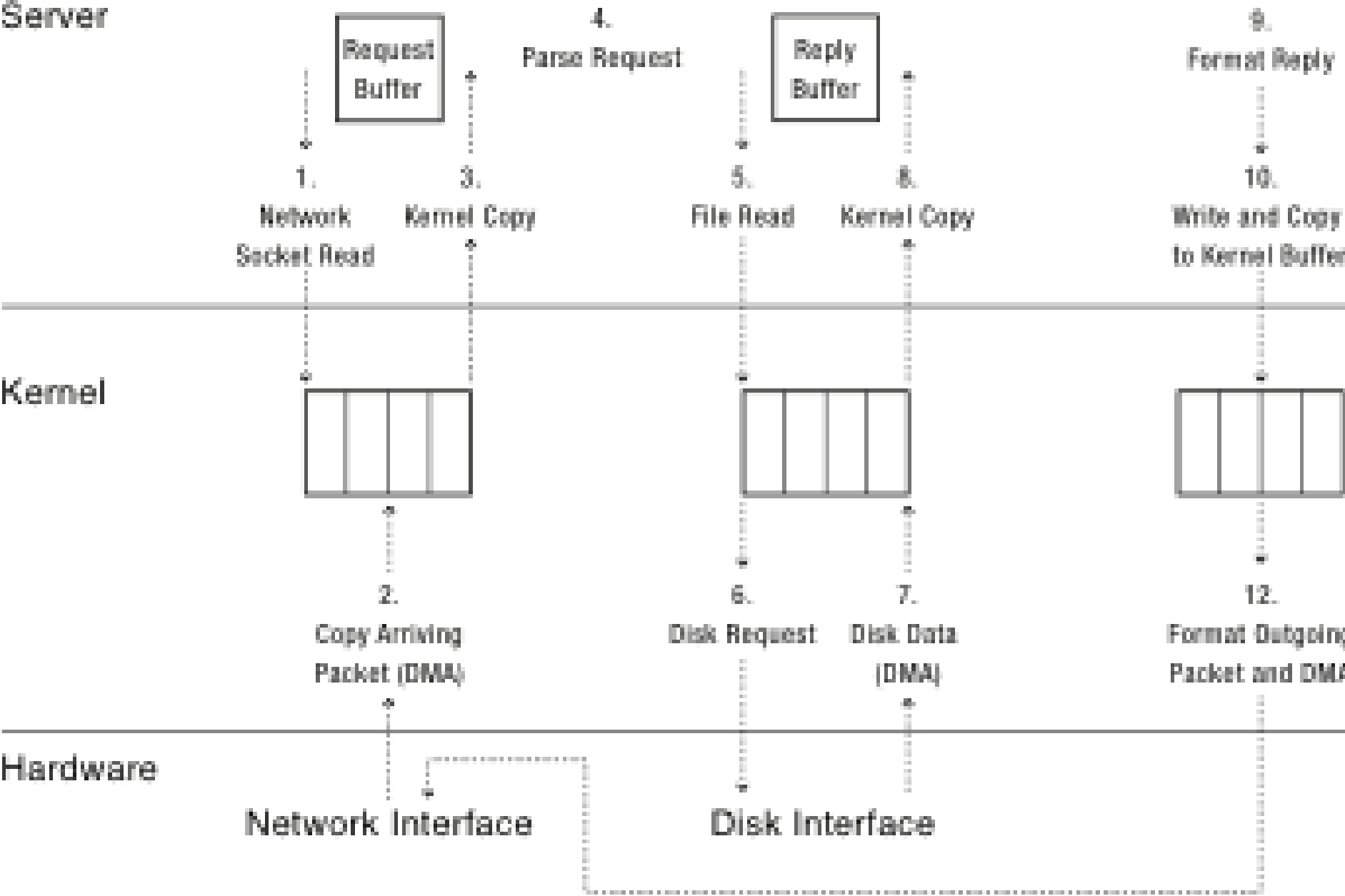
# Kernel System Call Handler

- **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!

Server

Kernel

Hardware



# 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