

# ECE 350

## Real-time Operating Systems



# Lecture 2: OS Concepts

---

Prof. Seyed Majid Zahedi

<https://ece.uwaterloo.ca/~smzahedi>

# Outline

---

- Brief history of OS's
- Four fundamental OS concepts
  - Thread
  - Address space
  - Process
  - Dual-mode operation/protection

# Very Brief History of OS

---

- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, ... Multics



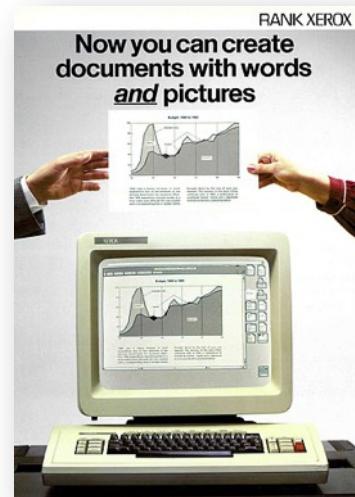
“I think there is a world market for maybe five computers.” – Thomas Watson, chairman of IBM, 1943

Thomas Watson was often called “the world's greatest salesman” by the time of his death in 1956

# Very Brief History of OS

---

- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, ... Multics
  - Hardware cheaper, humans expensive
    - PCs, workstations, rise of GUIs
  - Hardware very cheap, humans very expensive
    - Ubiquitous devices, widespread networking



# Very Brief History of OS

---

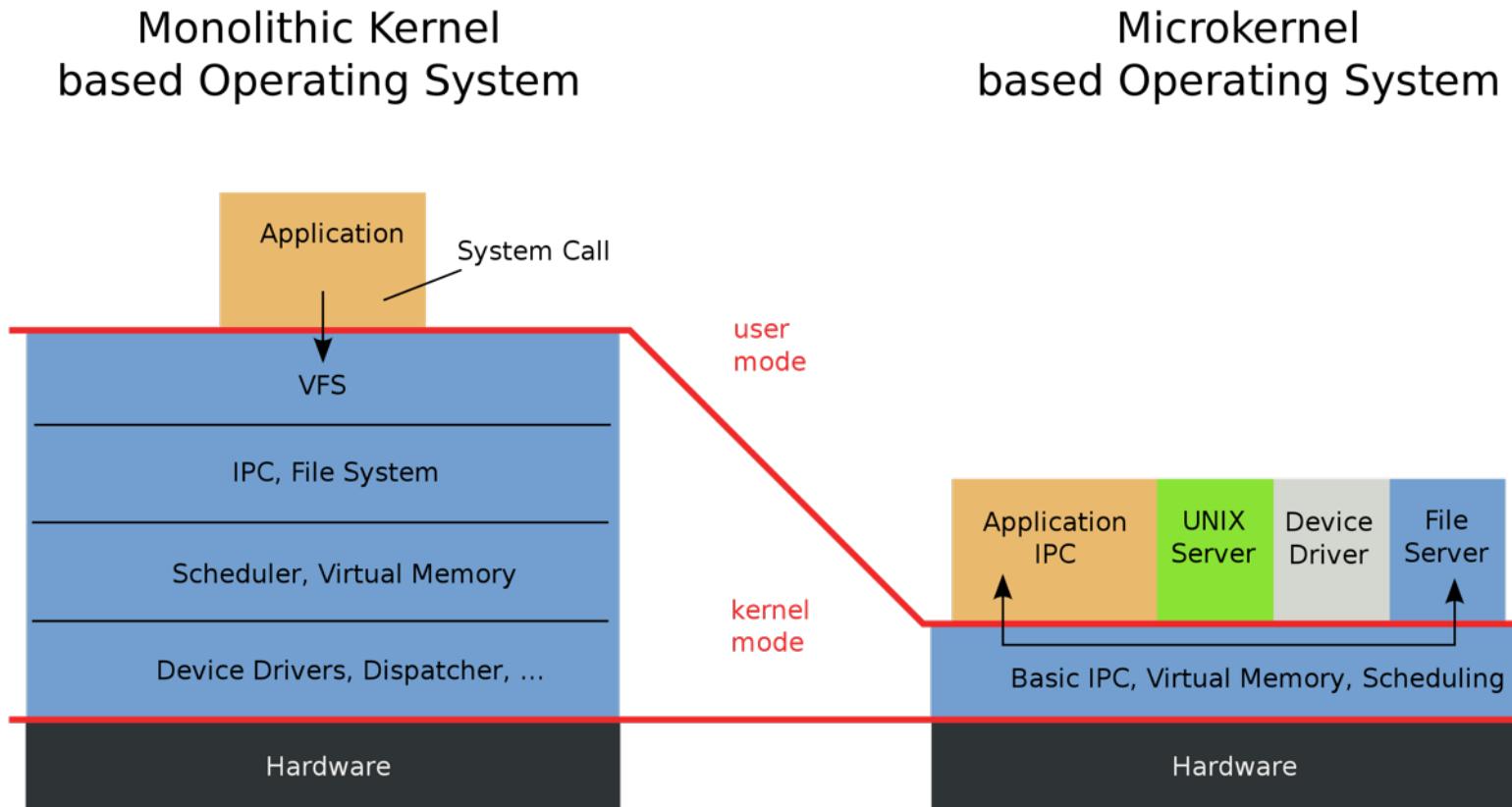
- Several distinct phases:
  - Hardware expensive, humans cheap
    - Eniac, ... Multics
  - Hardware cheaper, humans expensive
    - PCs, workstations, rise of GUIs
  - Hardware very cheap, humans very expensive
    - Ubiquitous devices, widespread networking
- Rapid change in hardware leads to changing OS
  - Batch  $\Rightarrow$  multiprogramming  $\Rightarrow$  timesharing  $\Rightarrow$  GUI  $\Rightarrow$  ubiquitous devices
  - Gradual migration of features into smaller machines
- Today
  - Small OS: 100K lines / Large: 20M lines (10M browser!)
  - 100-1000 people-years

# OS Archaeology

---

- Due to high cost of building OS from scratch, most modern OS's have long lineage
- Multics  $\Rightarrow$  AT&T Unix  $\Rightarrow$  BSD Unix  $\Rightarrow$  Ultrix, SunOS, NetBSD,...
- Mach (micro-kernel) + BSD  $\Rightarrow$  NextStep  $\Rightarrow$  XNU  $\Rightarrow$  Apple OS X, iPhone iOS
- MINIX  $\Rightarrow$  Linux  $\Rightarrow$  Android, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,...
- CP/M  $\Rightarrow$  QDOS  $\Rightarrow$  MS-DOS  $\Rightarrow$  Windows 3.1  $\Rightarrow$  NT  $\Rightarrow$  95  $\Rightarrow$  98  $\Rightarrow$  2000  $\Rightarrow$  XP  $\Rightarrow$  Vista  $\Rightarrow$  7  $\Rightarrow$  8  $\Rightarrow$  10  $\Rightarrow$  ...

# Monolithic vs Microkernel OS



# Influence of Microkernels

---

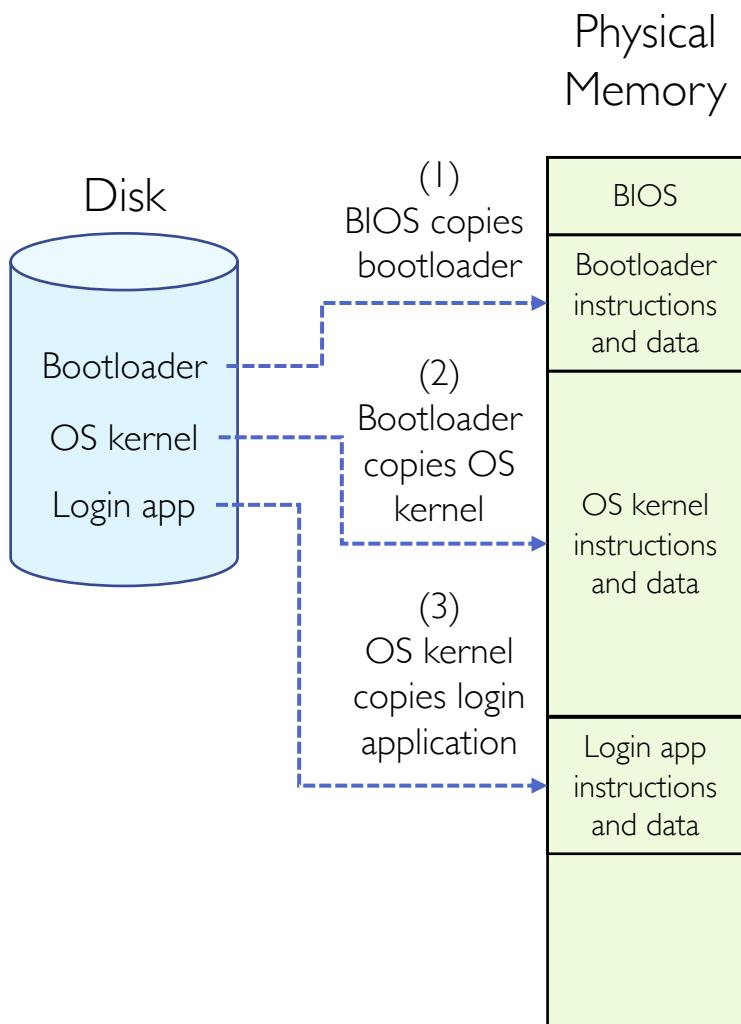
- Microkernels provide better modularity, security, and fault tolerance, but they introduce higher communication overhead
  - Too many context switches
- Many OSes provide some services externally, like microkernels
  - OS X and Linux: Windowing (graphics and UI)
- Some currently monolithic OSes started as microkernels
  - Windows family originally had microkernel design
  - OS X is hybrid of Mach microkernel and FreeBSD monolithic kernel

# Today: Four Fundamental OS Concepts

---

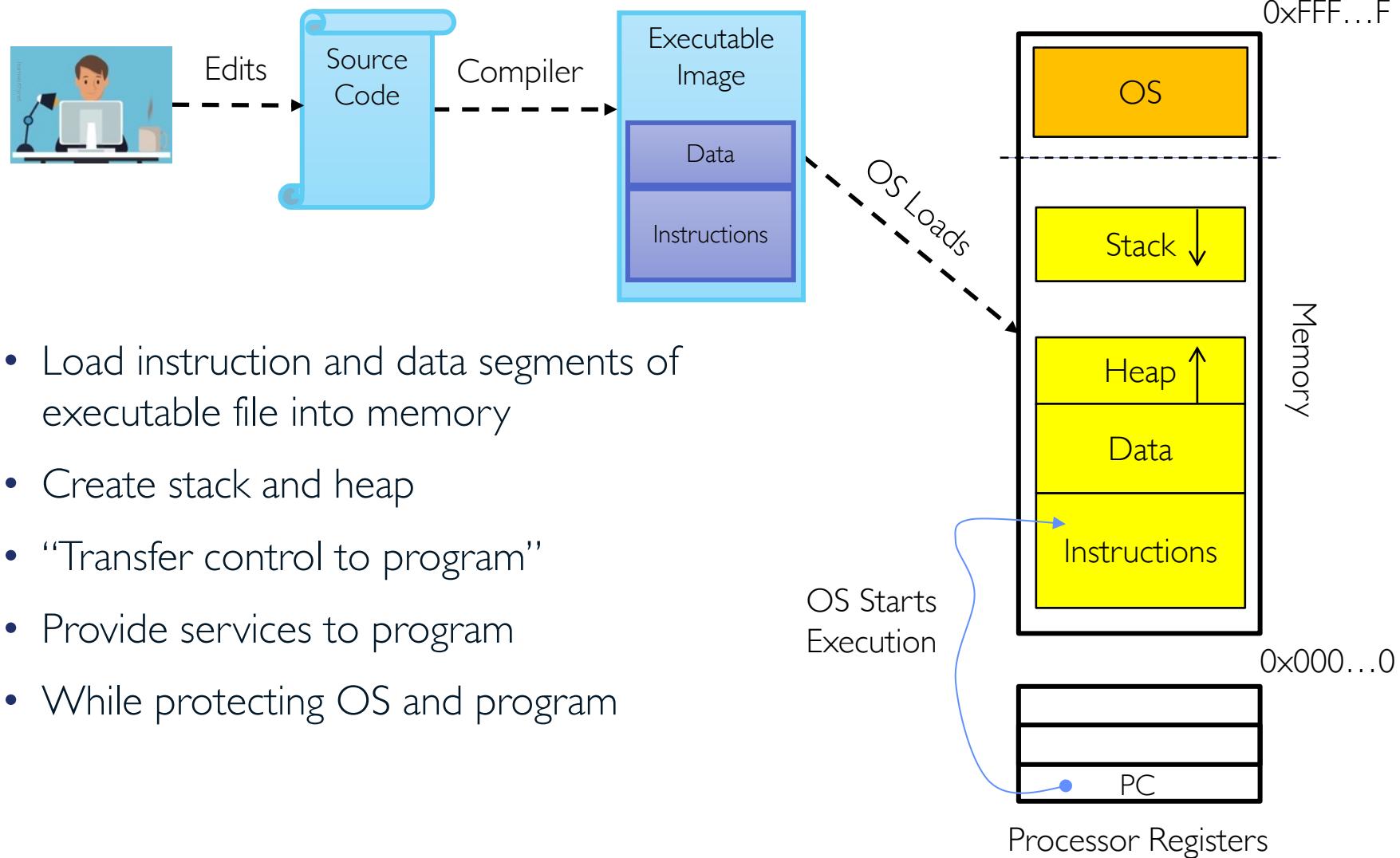
- Thread
  - Single unique execution context which fully describes program state
  - Program counter, registers, execution flags, stack
- Address space (with translation)
  - Address space which is distinct from machine's physical memory addresses
- Process
  - Instance of executing program consisting of address space and 1+ threads
- Dual-mode operation/protection
  - Only "system" can access certain resources
  - OS and hardware are protected from user programs
  - User programs are isolated from one another by controlling translation from program virtual addresses to machine physical addresses

# Booting OS

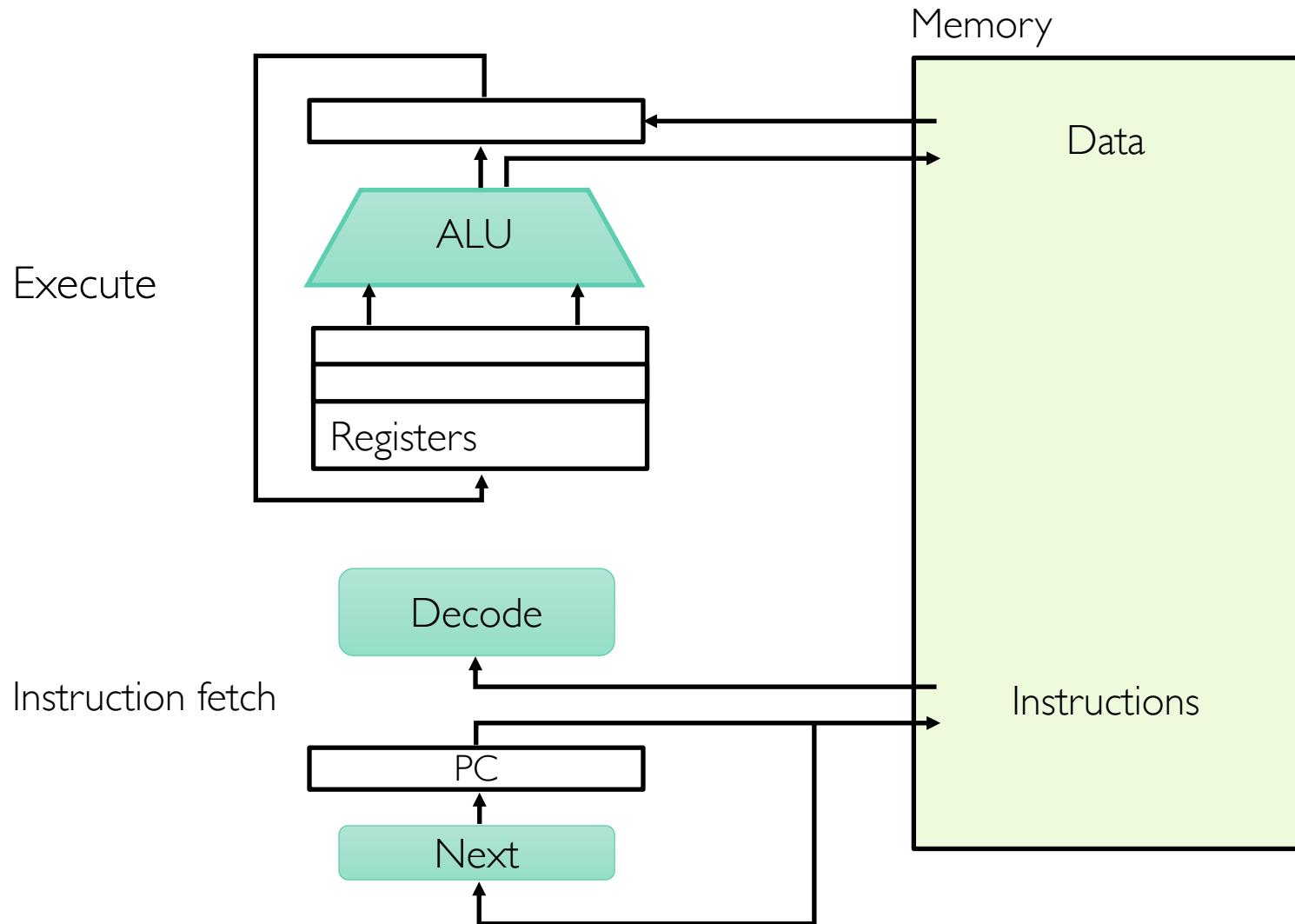


- In most x86 systems, BIOS is stored on Boot ROM
  - Expensive and writing to it is slow
- Why not storing kernel on Boot ROM?
  - Hard to update (OS updates are frequent)
- Why does BIOS load bootloader not OS?
  - Might have multiple OSes installed
  - BIOS needs to read raw bytes from disk, whereas bootloader needs to know how to read from filesystem

# OS Bottom Line: Run Programs

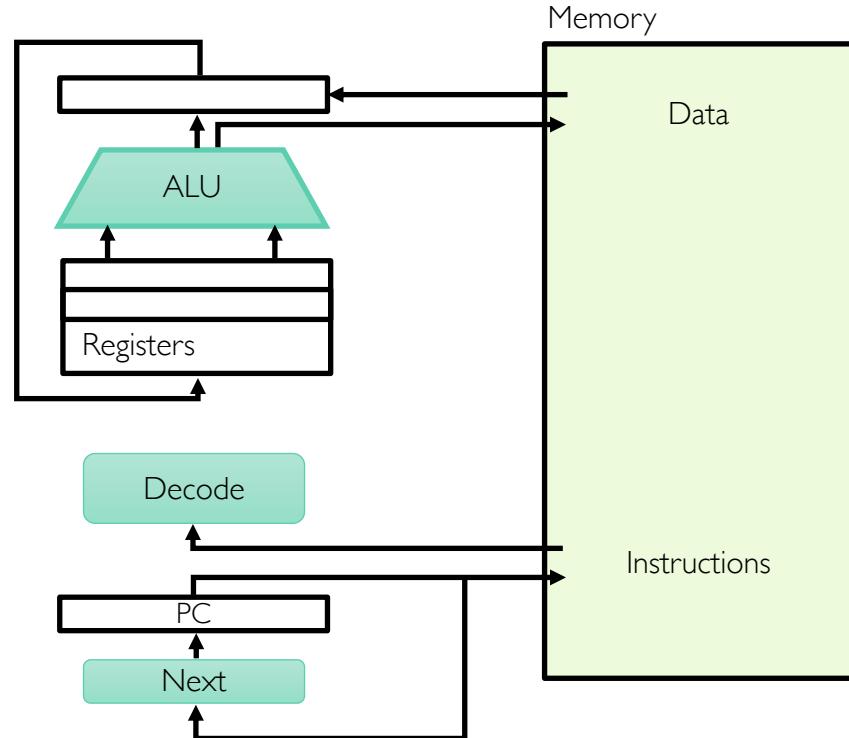


# Instruction Cycle: Fetch, Decode, Execute



# What Happens During Program Execution?

- Execution sequence:
  - Fetch instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/memory
  - $PC \leftarrow Next(PC)$
  - Repeat



Next instruction or jump to new address ...

# Thread (1<sup>st</sup> OS Concept)

---

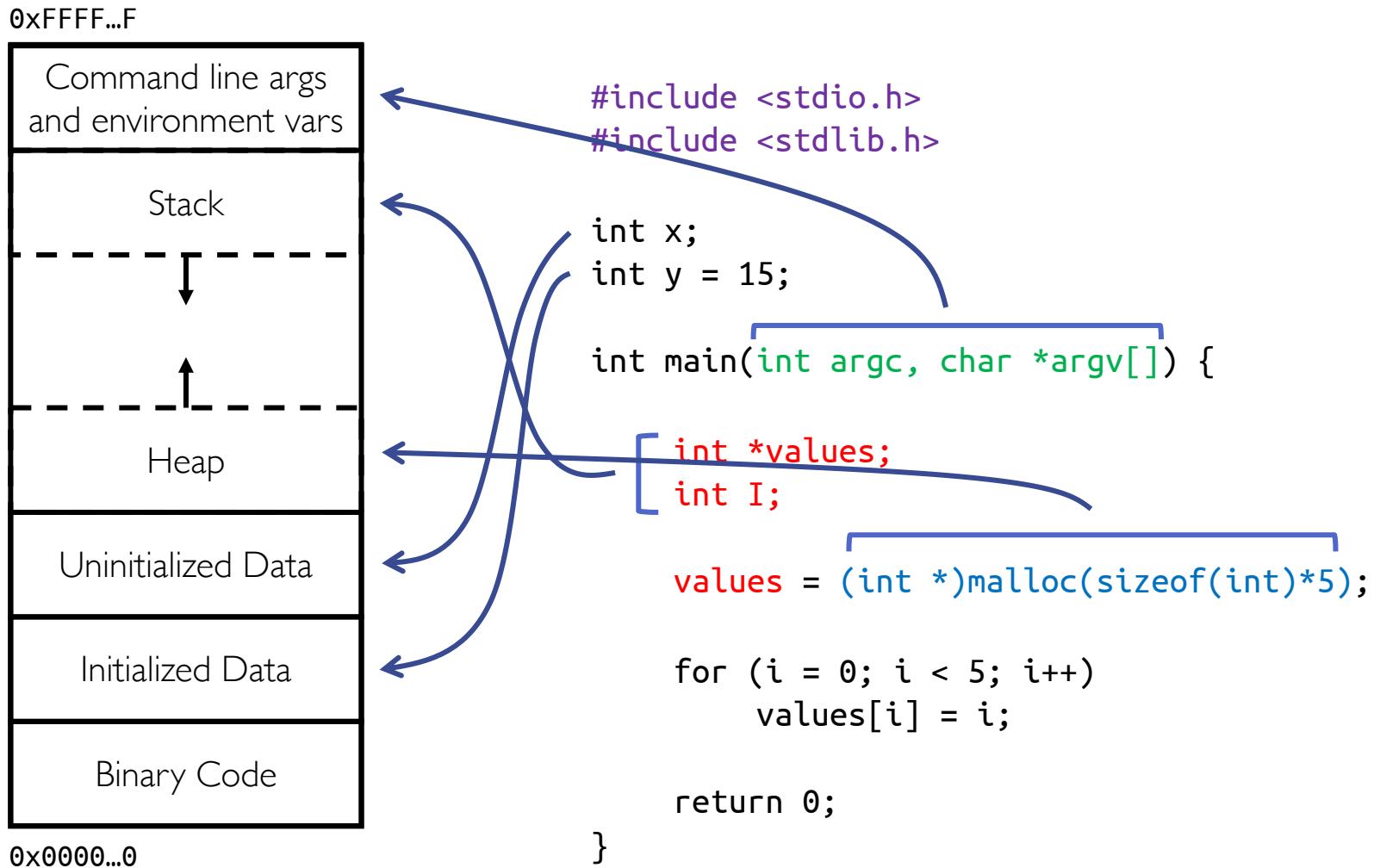
- Thread is single **unique** execution context
  - Program counter (PC), registers, execution flags, stack
- Thread is executing on processor when it resides in processor's registers
- Registers hold root state of thread (the rest is "in memory")
- Registers are defined by **instruction set architecture (ISA)** or by compiler
  - Stack pointer (SP) holds address of top of stack
    - Other conventions: frame pointer, heap pointer, data
  - PC register holds the address of executing instruction in the thread

# Address Space (2nd OS Concept)

---

- **Address space**: set of accessible addresses and their state
- **Physical memory**: data storage medium
- **Physical addresses**: addresses available on physical memory
  - For 4GB of memory:  $2^{32}$  ~ 4 billion addresses
- **Virtual addresses**: addresses generated by program
  - For 64-bit processor:  $2^{64} > 18$  quintillion ( $10^{18}$ ) addresses

# Virtual Address Space Layout of C Programs



# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
ext:  
      A(1);
```

Stack  
Pointer →

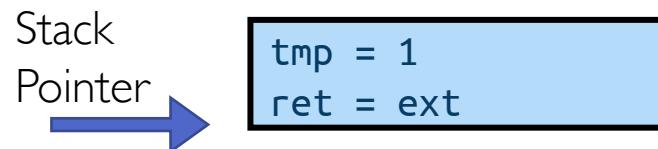
```
tmp = 1  
ret = ext
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```

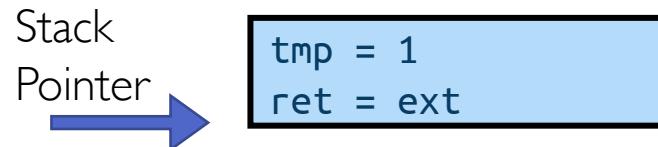


- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:     printf(tmp);  
A4: }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```

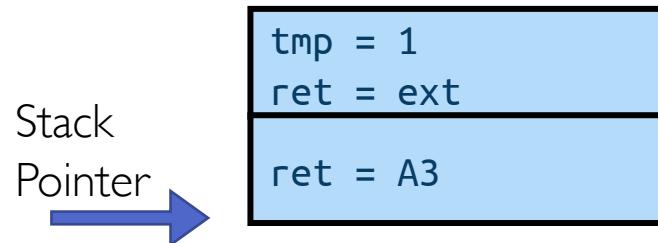


- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```

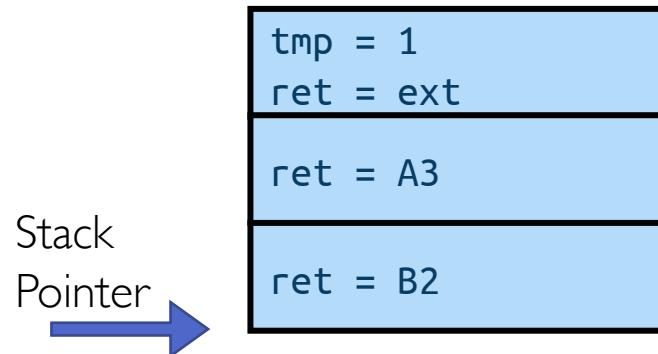


- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

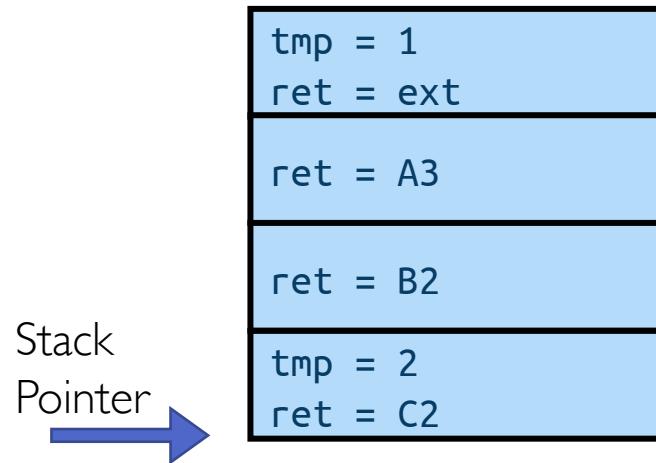
```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
  
B0: B() {  
B1:     C();  
B2: }  
  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
  
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

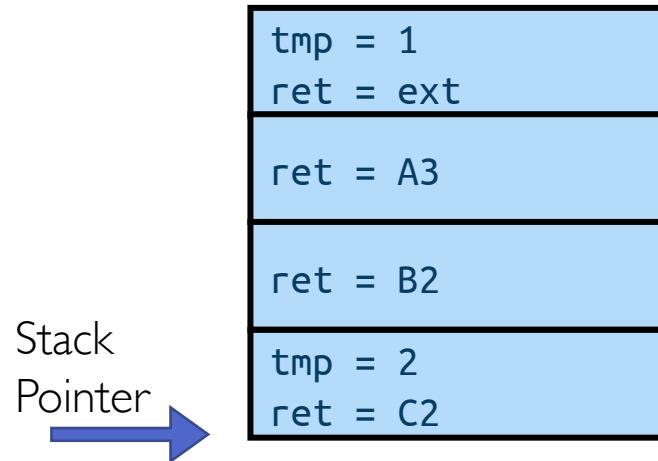
```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:     printf(tmp);  
A4: }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```

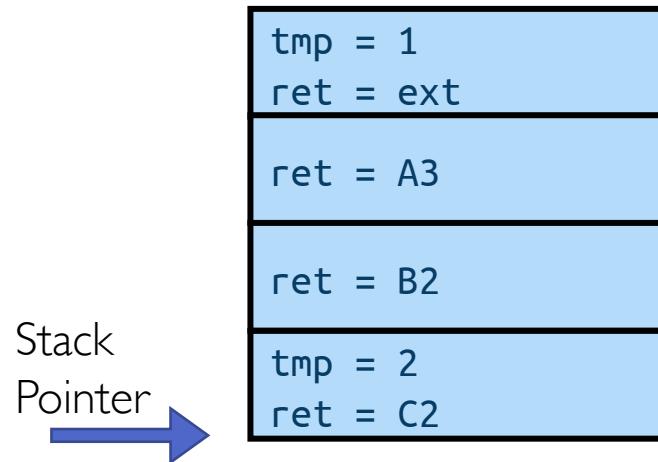


> 2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }   
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```



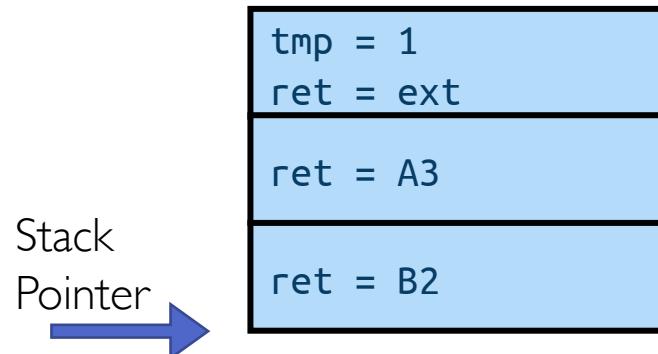
> 2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
  
B0: B() {  
B1:     C();  
B2: }  
  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
  
ext:
```



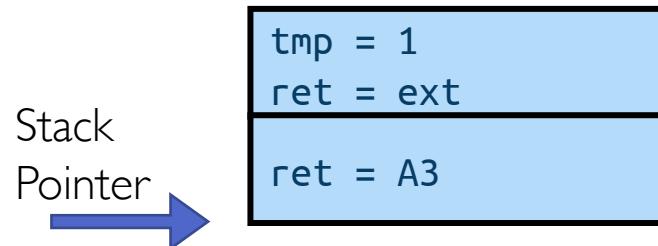
> 2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```



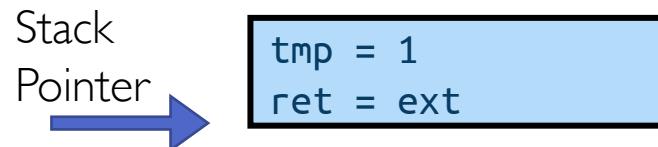
> 2

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```



> 21

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }   
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext:
```

Stack  
Pointer 

```
tmp = 1  
ret = ext
```

> 21

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Stack Example

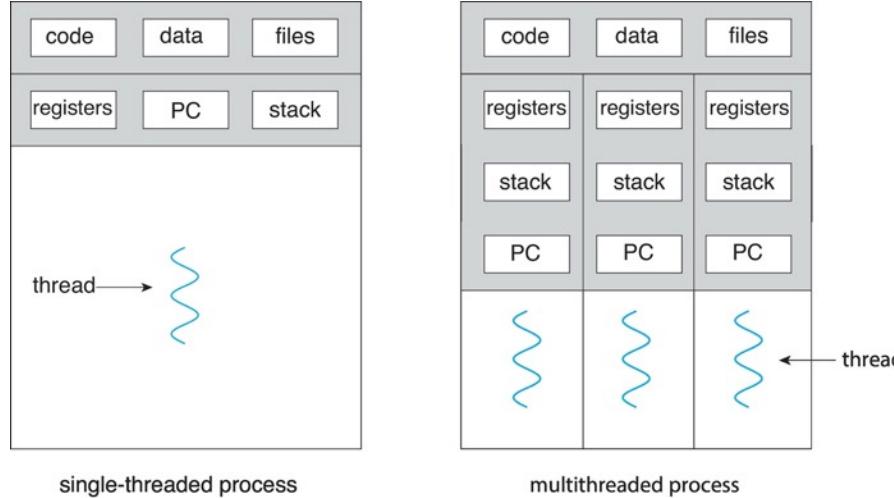
---

```
A0: A(int tmp) {  
A1:     if (tmp<2)  
A2:         B();  
A3:         printf(tmp);  
A4:     }  
B0: B() {  
B1:     C();  
B2: }  
C0: C() {  
C1:     A(2);  
C2: }  
        A(1);  
ext: 
```

> 21

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

# Process (3<sup>rd</sup> OS Concept)



- Process: execution environment with **restricted rights**
  - Address space with one or more threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, ...
- Why processes?
  - Fundamental tradeoff between protection and efficiency
    - Communication easier within a process
    - Communication harder between processes

# What Does it Take to Create Process?

---

- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space (more on this later)
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - With Unix `fork()`, child process gets copy of parent's memory and I/O state
  - Originally very expensive
  - Much less expensive with "copy on write" (more on this later)
- Copy I/O state (file handles, etc.)
  - Medium expense

# Multithreaded Processes

---

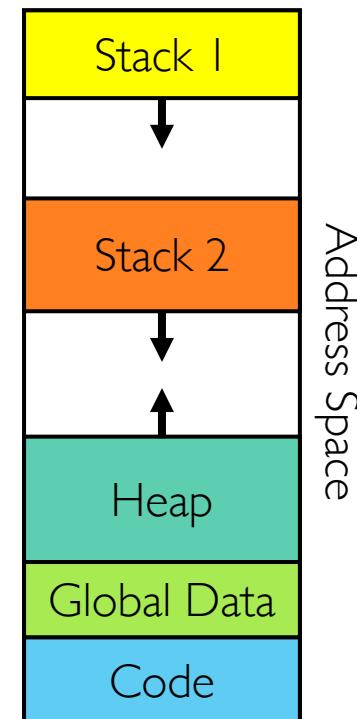


- Threads encapsulate **concurrency** and are **active** components
- Address spaces encapsulate **protection** and are **passive** part
  - Keeps buggy program from trashing system
- Why have multiple threads per address space?
  - Processes are expensive to start, switch between, and communicate between

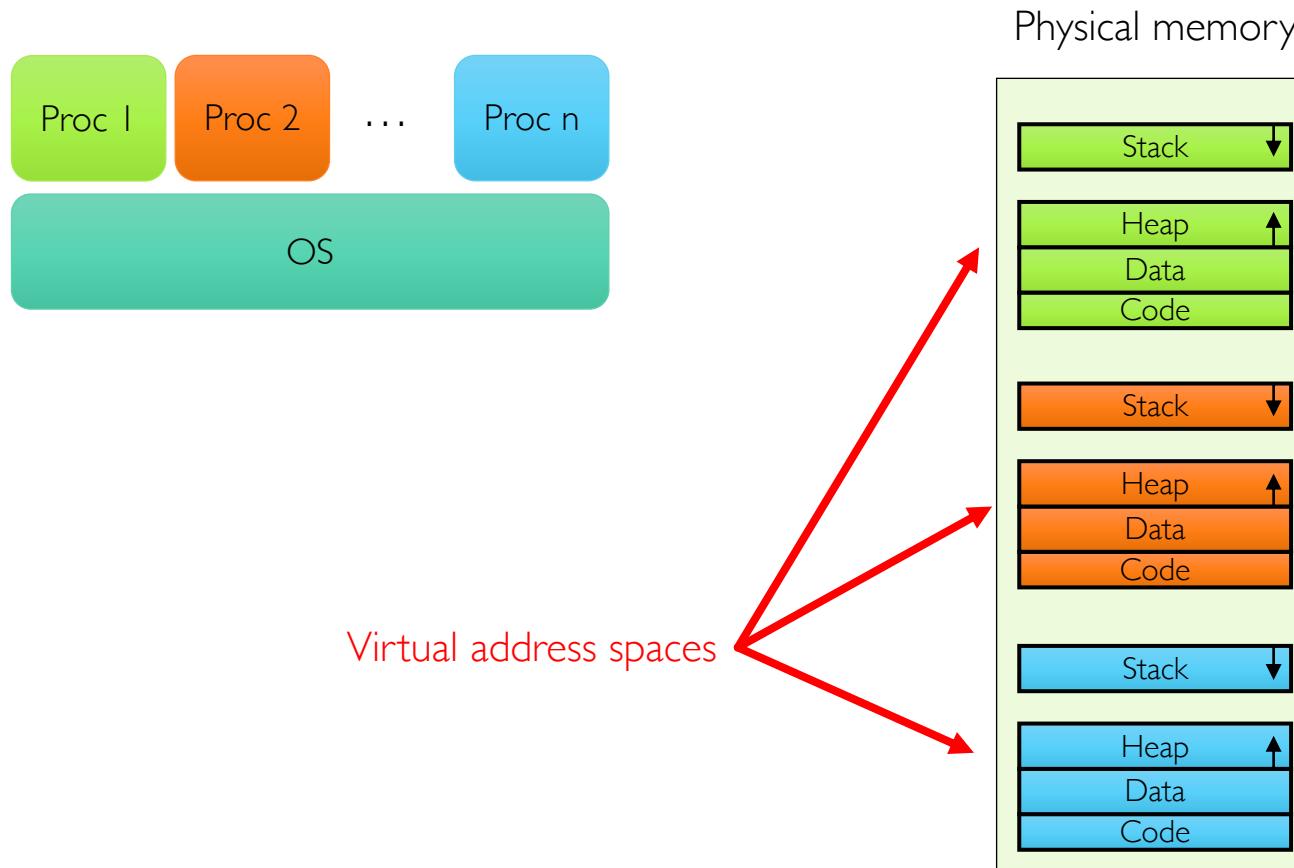
# Memory Footprint of Multiple Threads

---

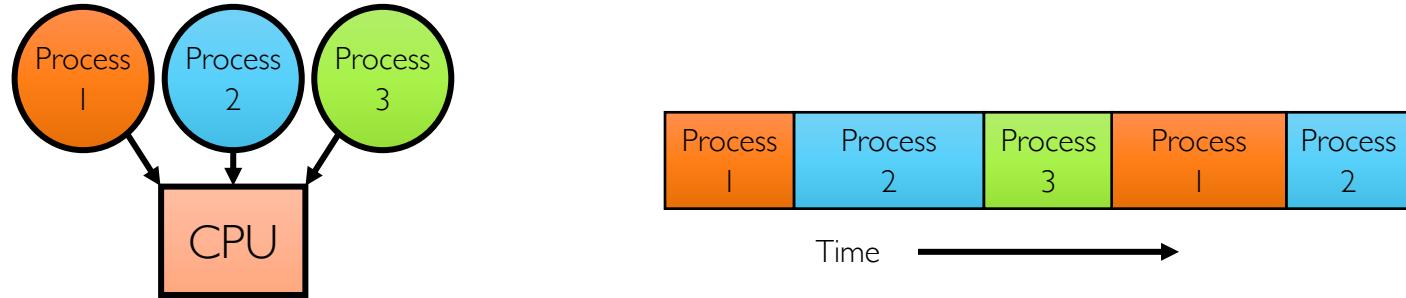
- How do we position stacks relative to each other?
- What maximum size should we choose for stacks?
  - 8KB for **kernel-level** stacks in Linux on x86
  - Less need for tight space constraint for user-level stacks
- What happens if threads violate this?
  - "... program termination and/or corrupted data"
- How might you catch violations?
  - Place guard values at top and bottom of each stack
  - Check values on every context switch



# Multiprogramming: Running Multiple Processes



# Time Sharing



- **Illusion:** Infinite number of processors
  - Each thread runs on dedicated virtual processor
- **Reality:** few processors, multiple threads running at variable speed
- How can we give illusion of infinite number of processors?
  - Multiplex in time!
- How do we **switch** from one process to next?
  - Save PC, SP, and registers in current PCB
  - Load PC, SP, and registers from new PCB
- What **triggers** switch?
  - Timer, voluntary yield, I/O interrupts, ...

# How Do We Multiplex Processes?

---

- **Scheduling:** OS decides which process uses CPU time
  - Only one process is “running” on each CPU at any time
  - Scheduler could give more time to *important* processes
- **Protection:** OS divides non-CPU resources among processes
  - E.g., give each process their own address space
  - E.g., multiplex I/O through system calls

# Scheduling

---

- Kernel scheduler decides which processes/threads receive CPU
- There are variety of scheduling policies for ...
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ...
- Kernel scheduler maintains data structure containing PCBs



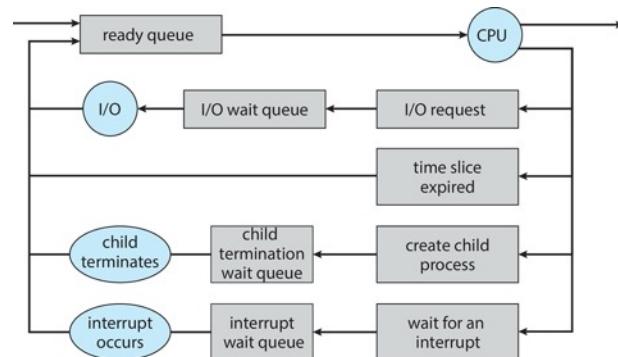
```
if (readyProcesses(PCBs)) {  
    nextPCB = selectProcess(PCBs);  
    run(nextPCB);  
} else {  
    run_idle_process();  
}
```

# Ready Queue

---

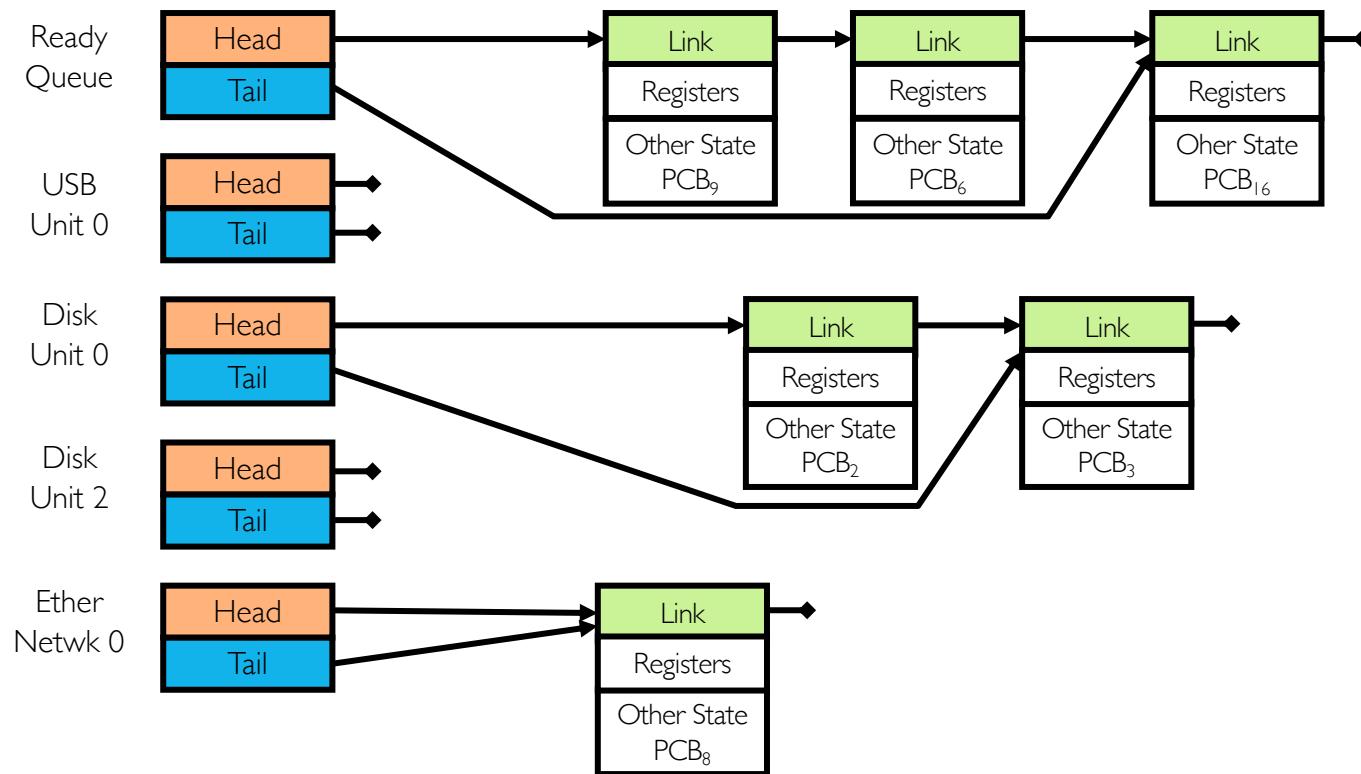


- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are **scheduling** decisions
  - Many algorithms possible (more on this in a few weeks)



# Ready Queue And I/O Device Queues

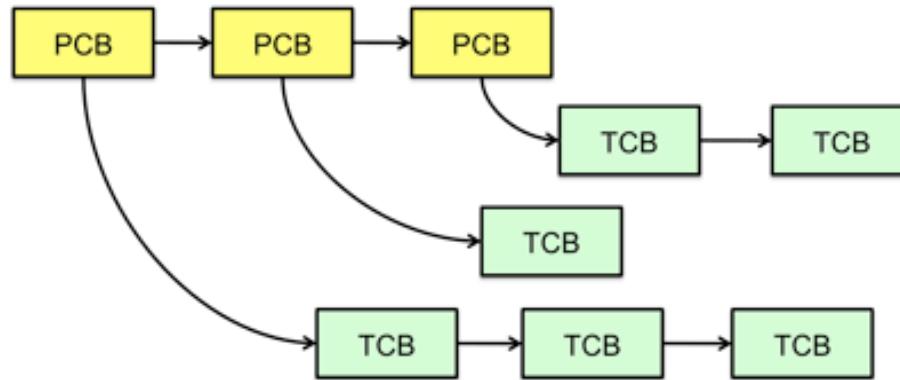
- Process not running  $\Rightarrow$  PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have different scheduler policy



# Multithreaded Processes

---

- PCBs could point to multiple TCBs



- Switching threads within one block is simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables

# Protection

---



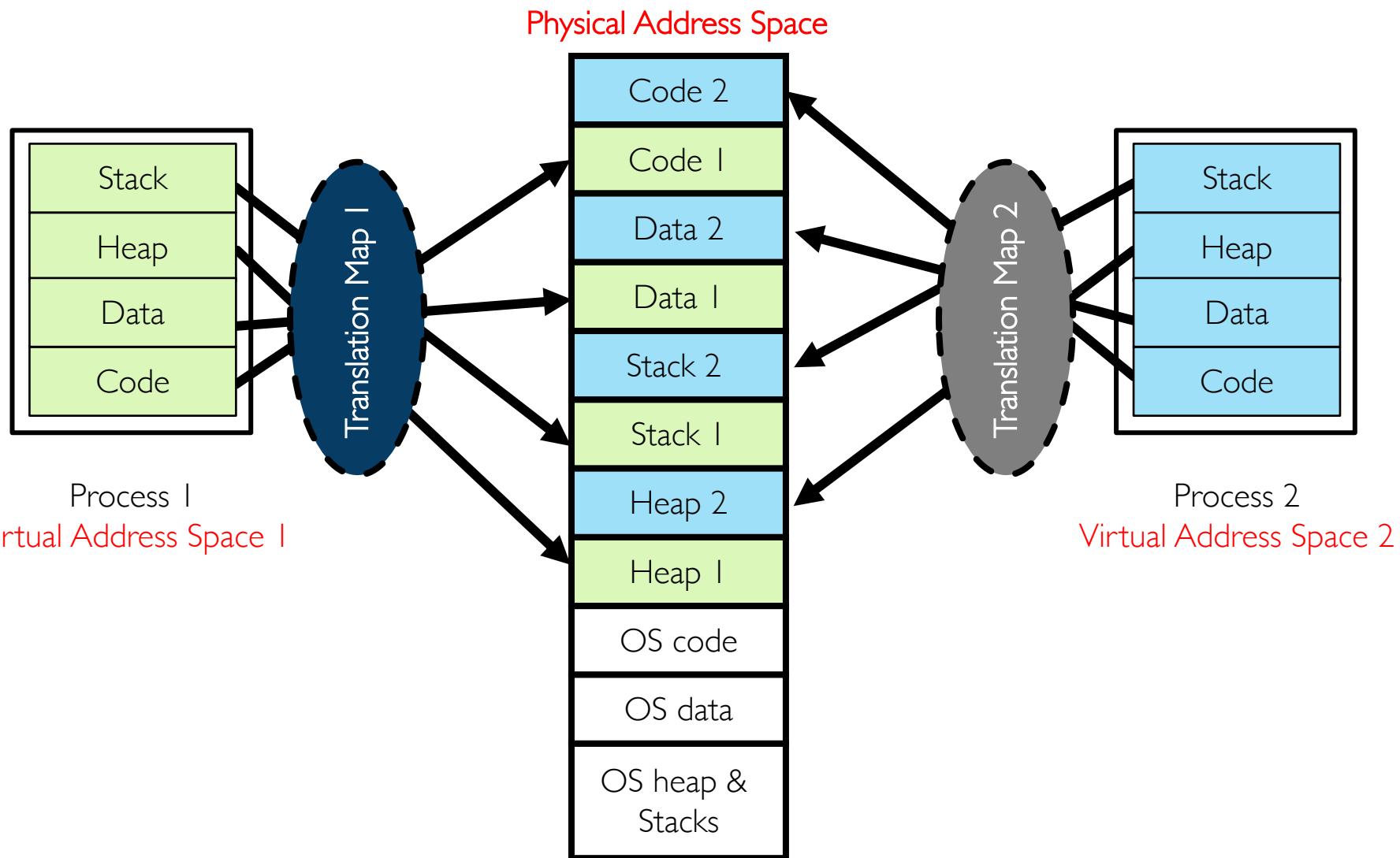
- OS must protect itself from user programs
  - Reliability: prevent OS from crashing
  - Security: limit scope of what processes can do
  - Privacy: limit data each process can access
  - Fairness: enforce appropriate share of HW
- It must protect user programs from one another
- Main method is to limit translation from virtual to physical address space

# How to Protect Processes from One Another?

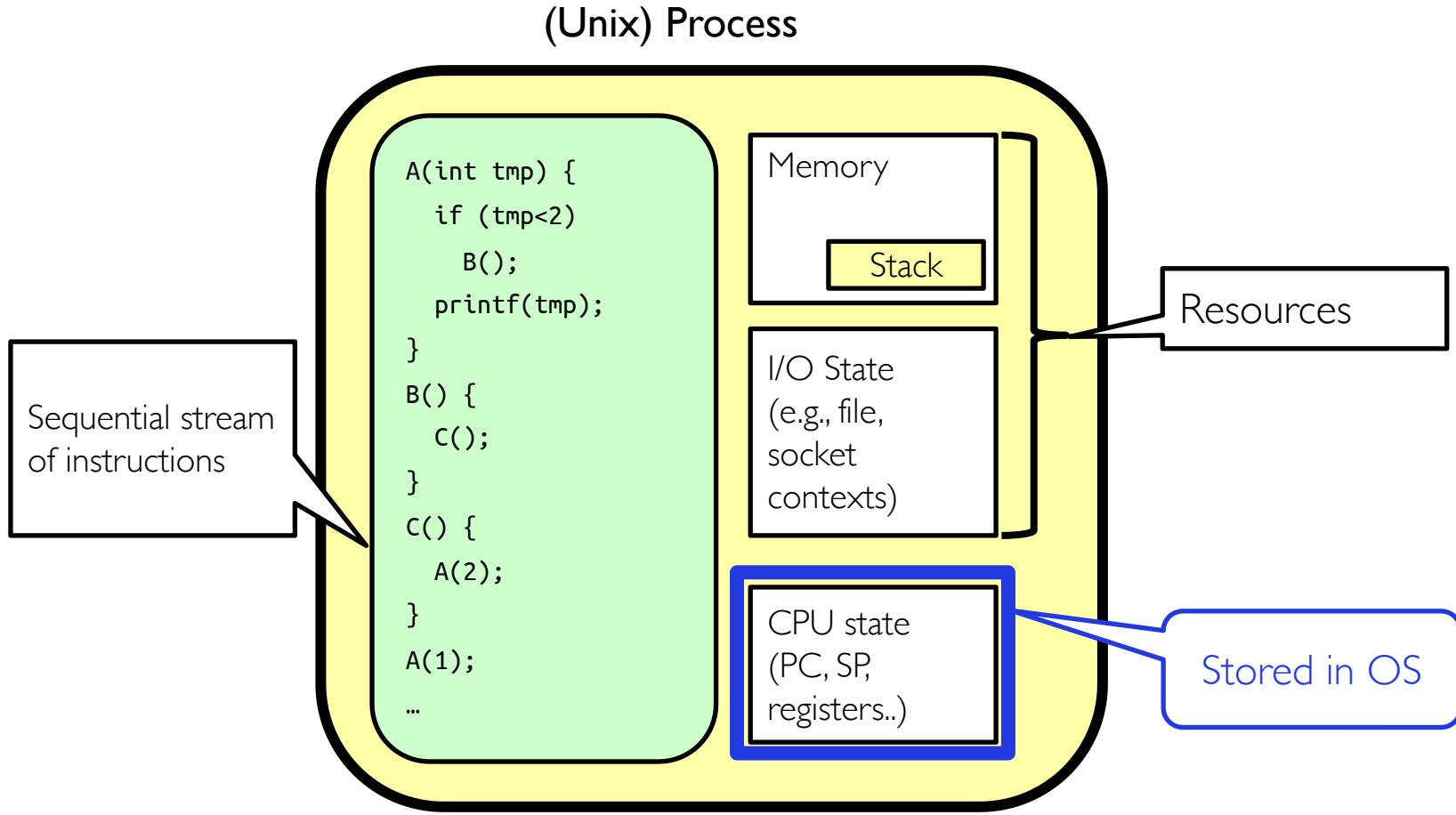
---

- Protection of memory
  - Every process does not have access to all memory
- Protection of I/O devices
  - Every process does not have access to every device
- Protection of access to processor
  - Preemptive switching from process to process
  - Use of timer
  - Must not be possible to disable timer from user code

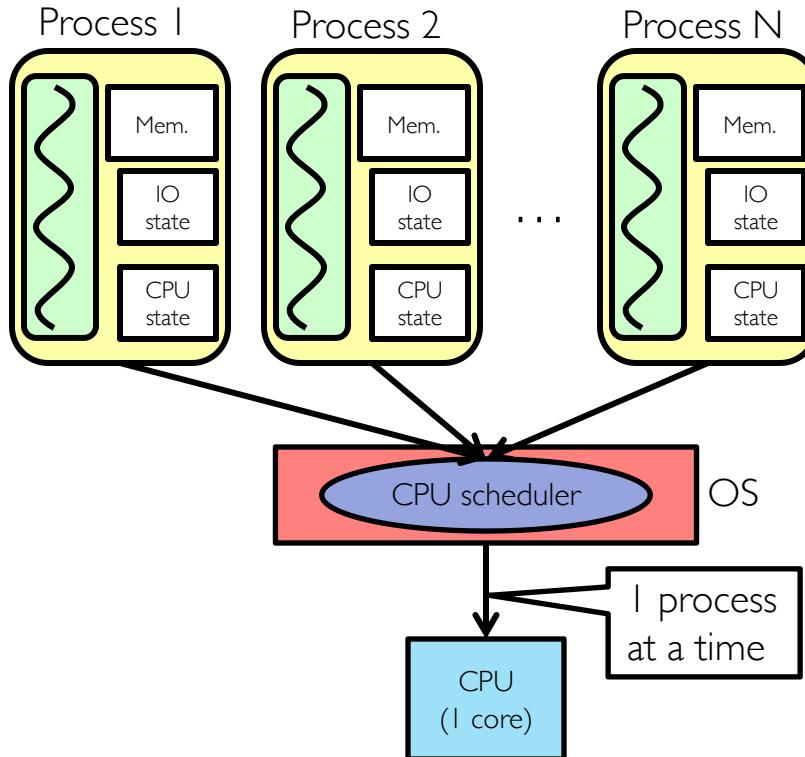
# Address Translation Maps: Illusion of Separate Address Space



# Putting it Together: Process

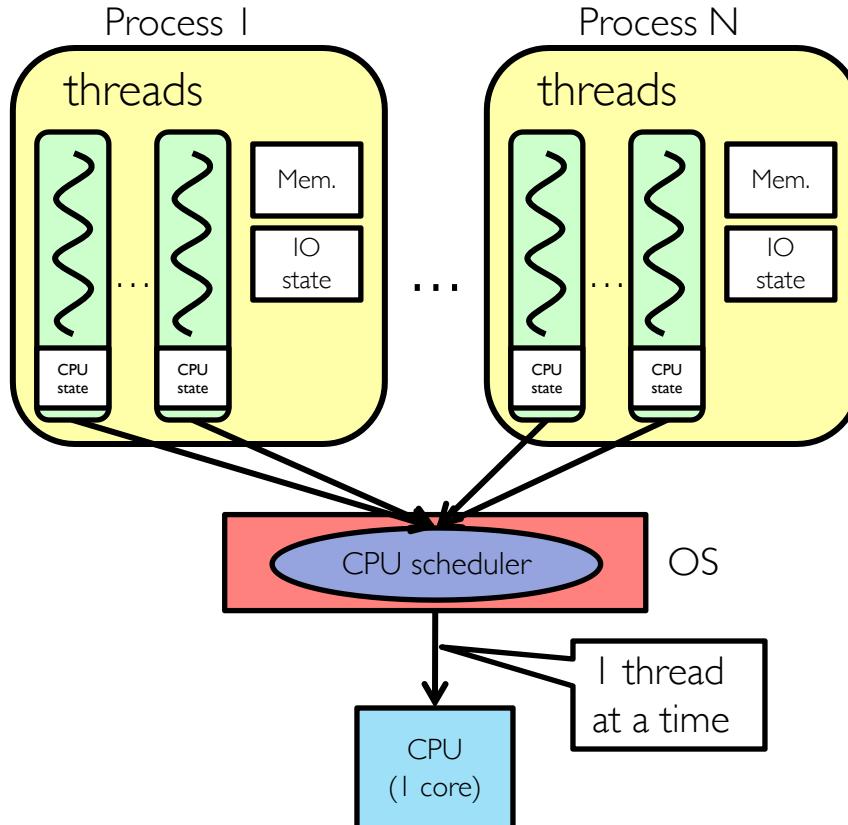


# Putting it Together: Processes



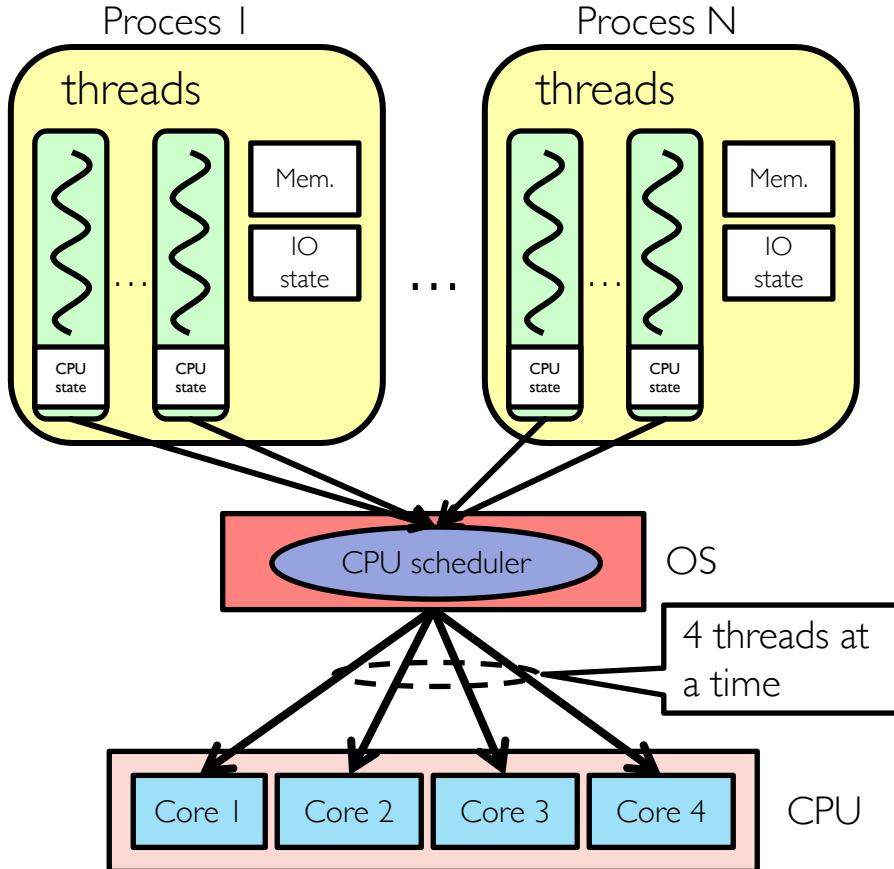
- Switch overhead: **high**
  - CPU state: **low**
  - Memory/IO state: **high**
- Process creation: **high**
- Protection
  - CPU: **yes**
  - Memory/IO: **yes**
- Sharing overhead: **high**  
(involves at least one context switch)

# Putting it Together: Threads



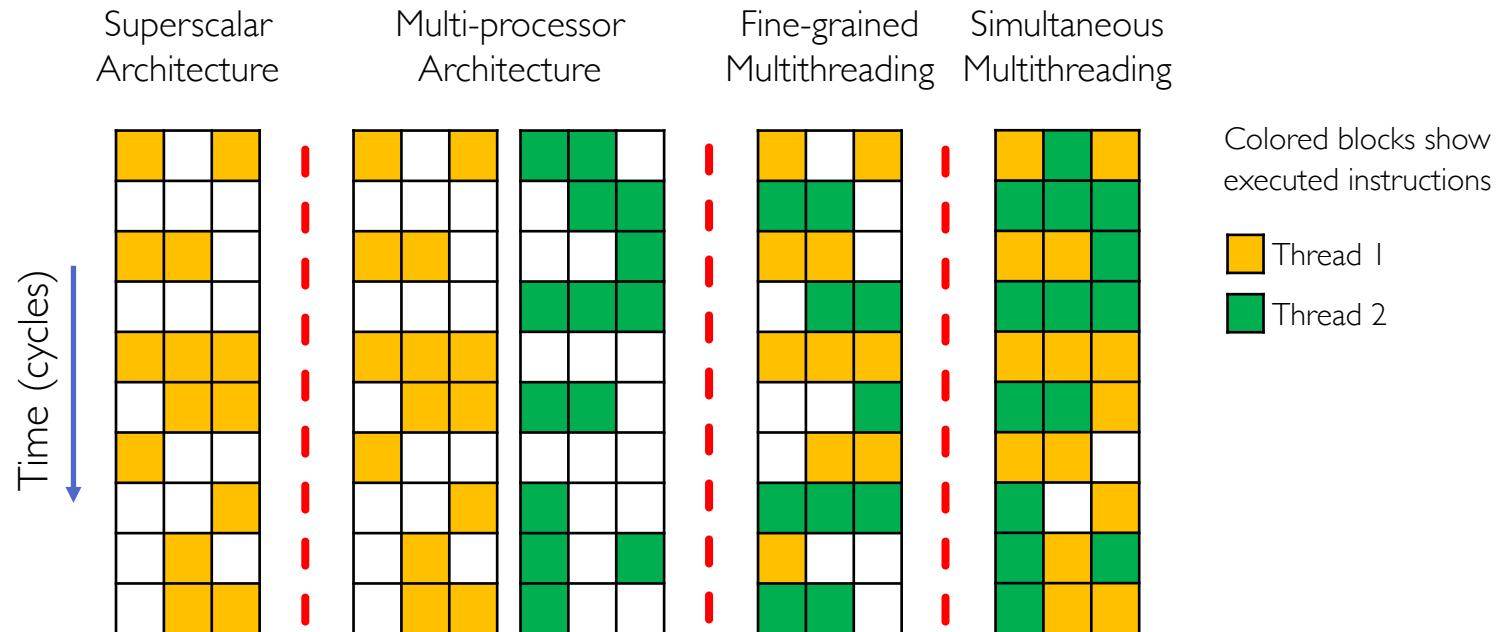
- Switch overhead: **medium**
  - CPU state: **low**
- Thread creation: **medium**
- Protection
  - CPU: **yes**
  - Memory/IO: **no**
- Sharing overhead: **low(ish)**  
(thread switch overhead low)

# Putting it Together: Multi-cores



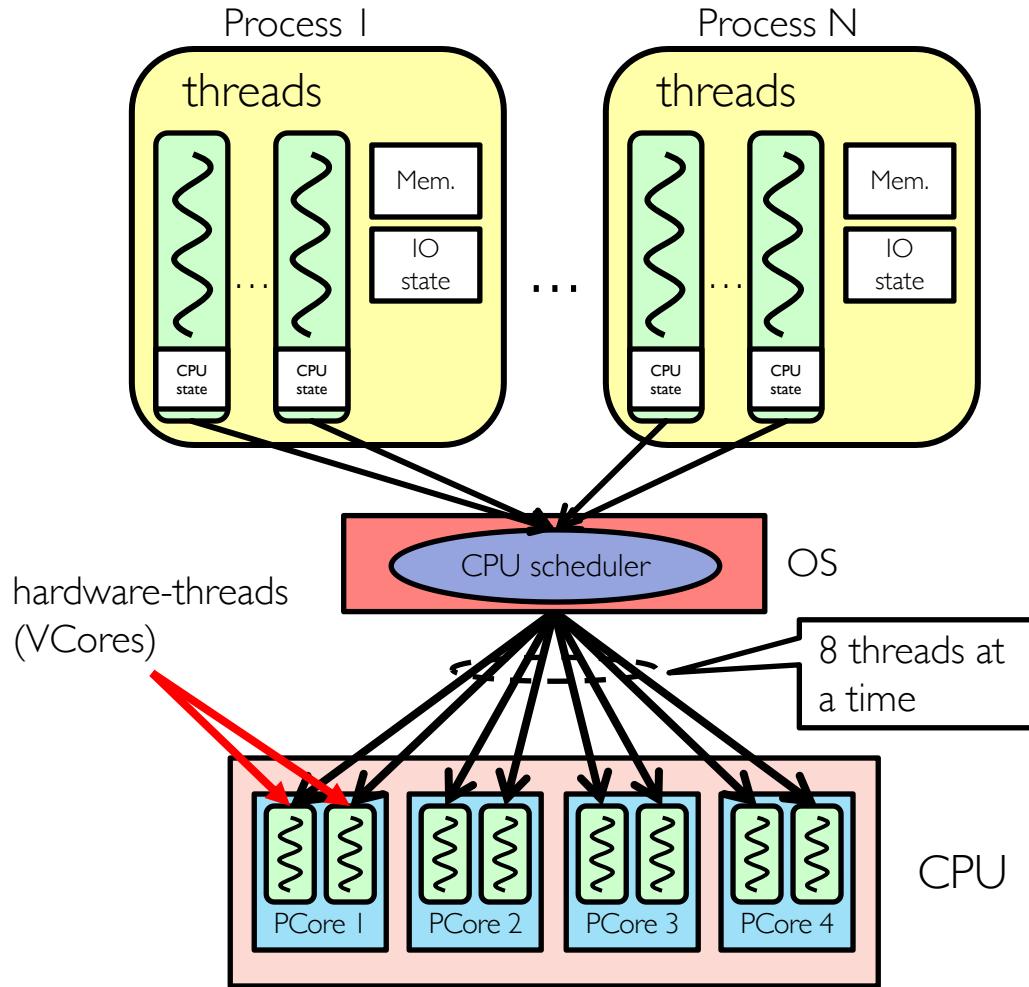
- Switch overhead: **low** (only CPU state)
- Thread creation: **low**
- Protection
  - CPU: **yes**
  - Memory/IO: **no**
- Sharing overhead: **low** (thread switch overhead **low**, may not need to switch at all!)

# Hyperthreading



- Superscalar processors can execute multiple instructions that are independent
- Multiprocessors can execute multiple independent threads
- Fine-grained multithreading executes two independent threads by switches between them
- Hyperthreading duplicates register state to make second (hardware) “thread” (virtual core)
  - From OS’s point of view, virtual cores are separate CPUs
  - OS can schedule as many threads at a time as there are virtual cores (but, sub-linear speedup!)
  - See: <http://www.cs.washington.edu/research/smt/index.html>

# Putting it Together: Hyperthreading



- Switch overhead between hardware-threads: **very-low** (done in hardware)
- Contention for ALUs/FPUs may **hurt** performance

# Dual-mode Operation (4<sup>th</sup> OS Concept)

---

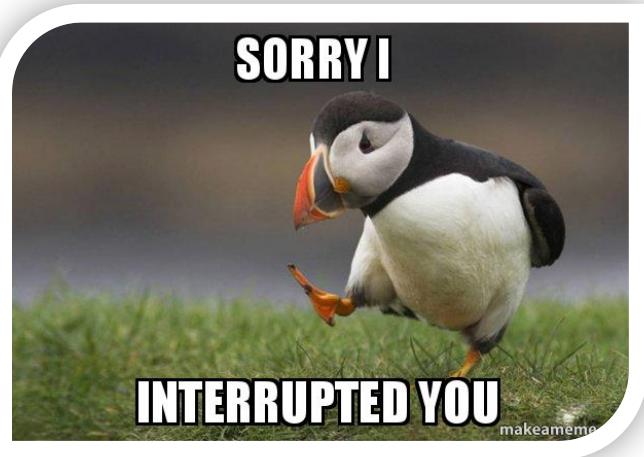
- Hardware provides at least two modes
  - Kernel mode (or “supervisor” or “protected”)
  - User mode, which is how normal programs are executed
- How can hardware support dual-mode operation?
  - Single bit of state (user/system mode bit)
  - Certain operations/actions only permitted in system/kernel mode
    - In user mode they fail or trap
  - User to kernel transition sets system mode AND saves user PC
    - OS code carefully puts aside user state then performs necessary actions
  - Kernel to user transition clears system mode AND restores user PC
    - E.g., `rfi`: return-from-interrupt

# Three Types of Mode Transfer

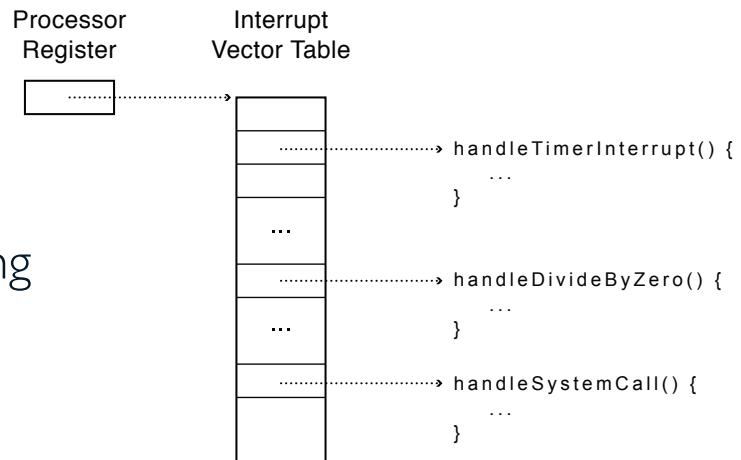
---

- Interrupt
  - External asynchronous event triggers context switch, e.g., Timer, I/O device
    - Independent of user process
- Syscall
  - Process requests system service, e.g., **exit**
    - Like function call, but outside process
  - Process does not have address of system function to call
    - Like a Remote Procedure Call (RPC) – for later
  - OS marshalls syscall id and args in registers and exec syscall
- Trap or exception
  - Internal synchronous event in process triggers context switch, e.g., protection violation (segmentation fault), divide by zero, ...
- All 3 are **UNPROGRAMMED CONTROL TRANSFER**
- How do we get address of unprogrammed control transfer?

# Interrupt Vector

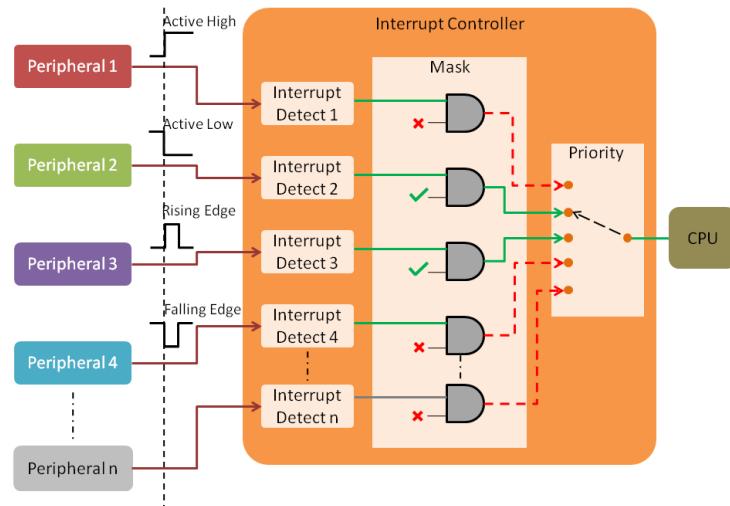


- Table set up by OS pointing to code to run on different events



# Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority picks highest enabled interrupt
  - Software interrupt set/cleared by SW
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) cannot be disabled



# Interrupt Masking

---

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - E.g., when determining next process/thread to run
  - On x86
    - `cli`: disable interrupts
    - `sti`: enable interrupts
    - Only applies to current CPU (on a multicore)
- We will need this to implement synchronization  
(more on this later)

# 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 (copy before check)
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

# Implementing Safe Mode Transfers

---

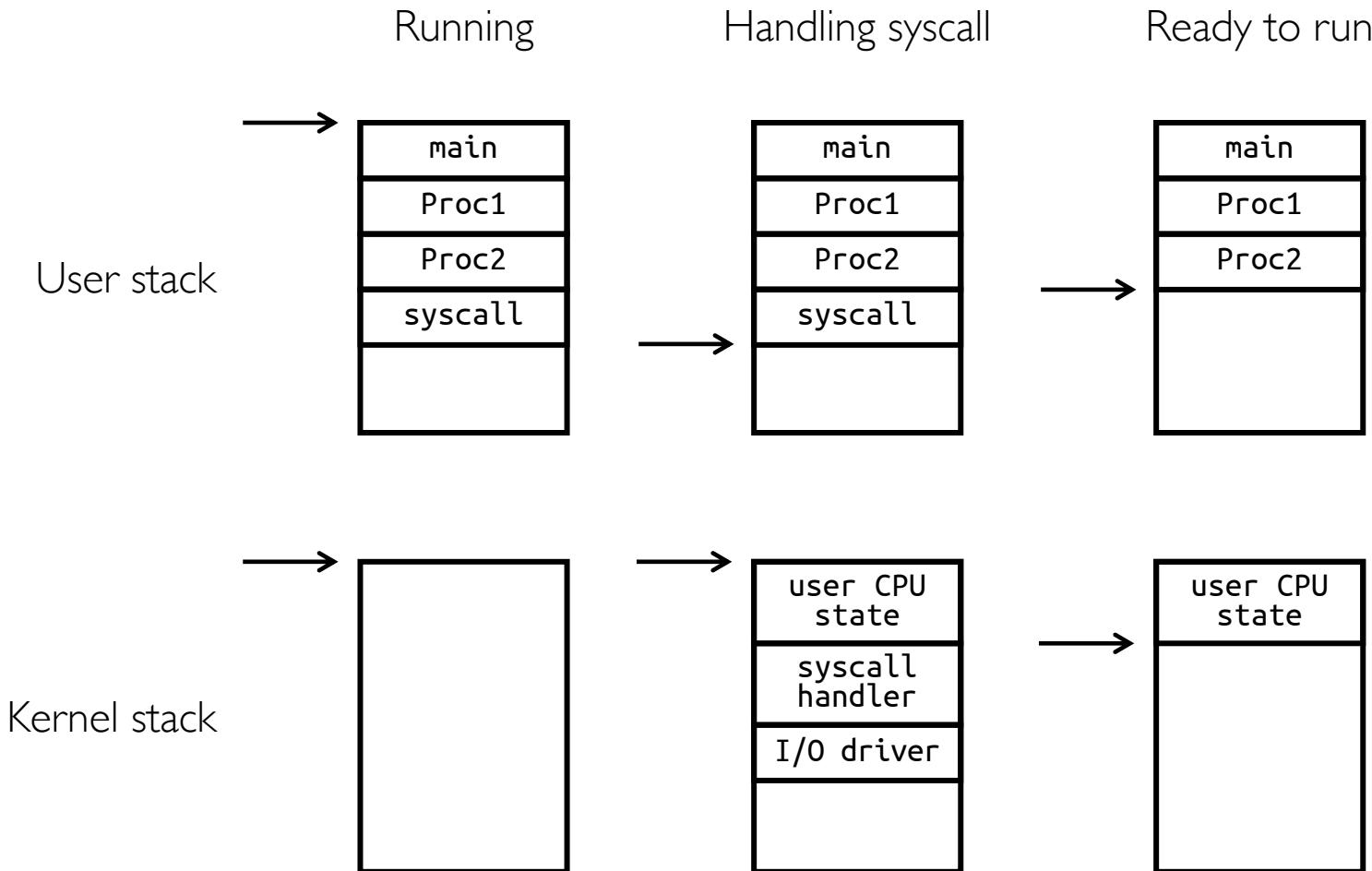
- It should be impossible for buggy or malicious user program to cause kernel to corrupt itself
  - Controlled transfer into kernel (e.g., interrupt vector table)
  - Separate kernel stack
- Carefully constructed kernel code should pack up user process state and set it aside
  - Details depend on the machine architecture

# Need for Separate Kernel Stacks

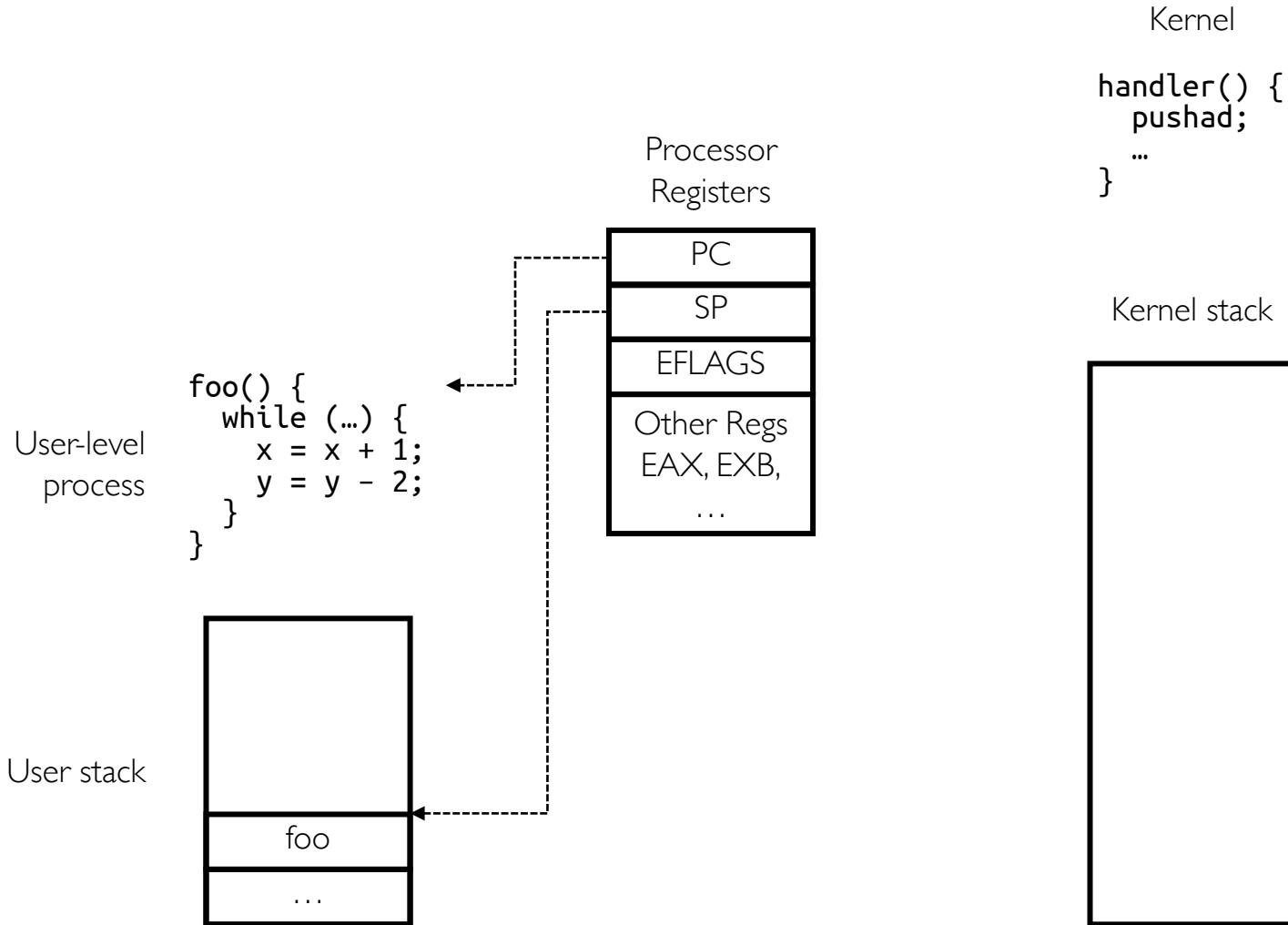
---

- Kernel cannot put anything on user stack (Why?)
  - Reliability: what if user program's SP is not valid?
  - Security: what if other threads in user process change kernel's return address?
- Two-stack model
  - Kernel keeps separate stack for each thread in kernel memory (in addition to user stack in user memory)

# Two-Stack Model

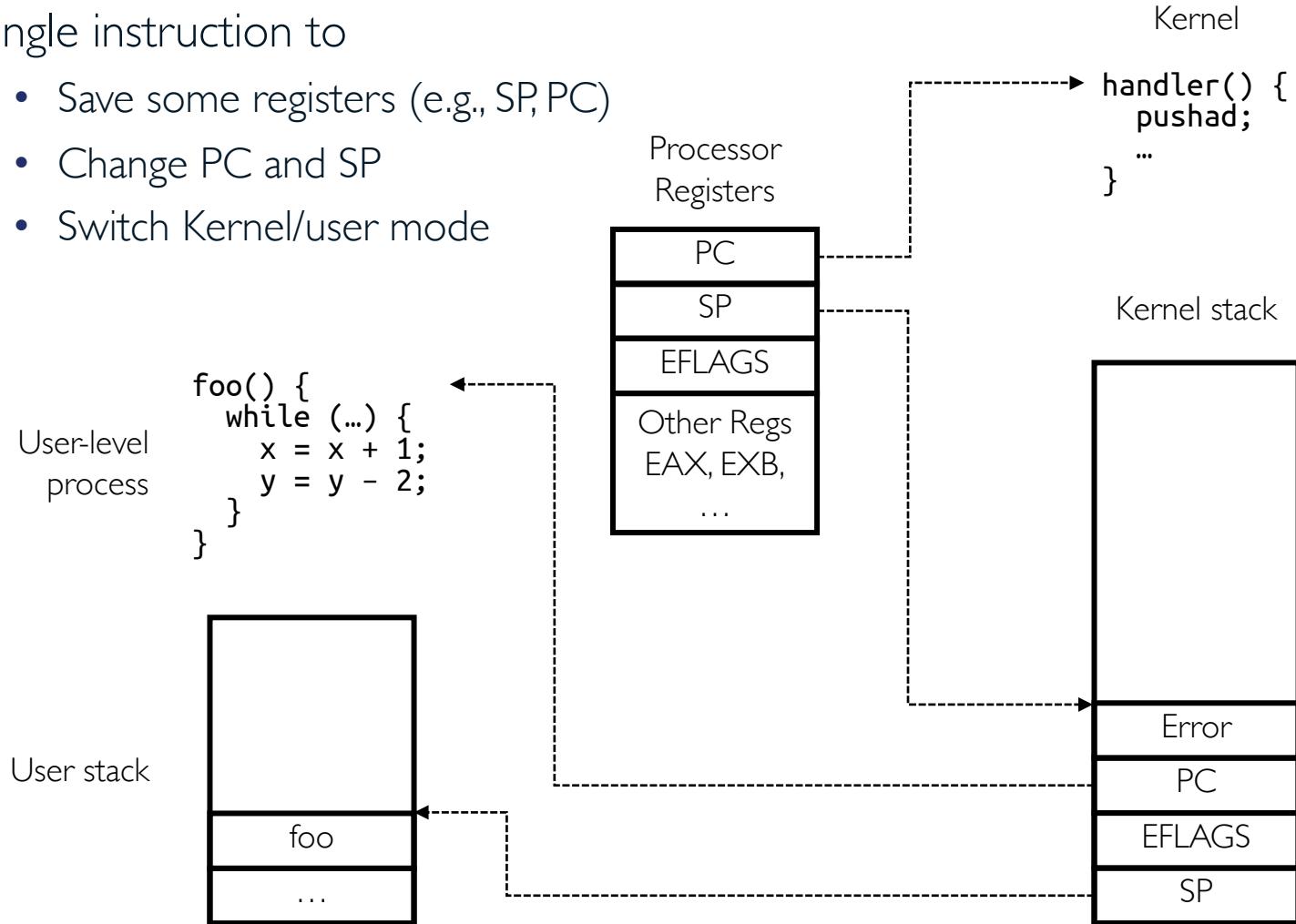


# Atomic Transfer of Control



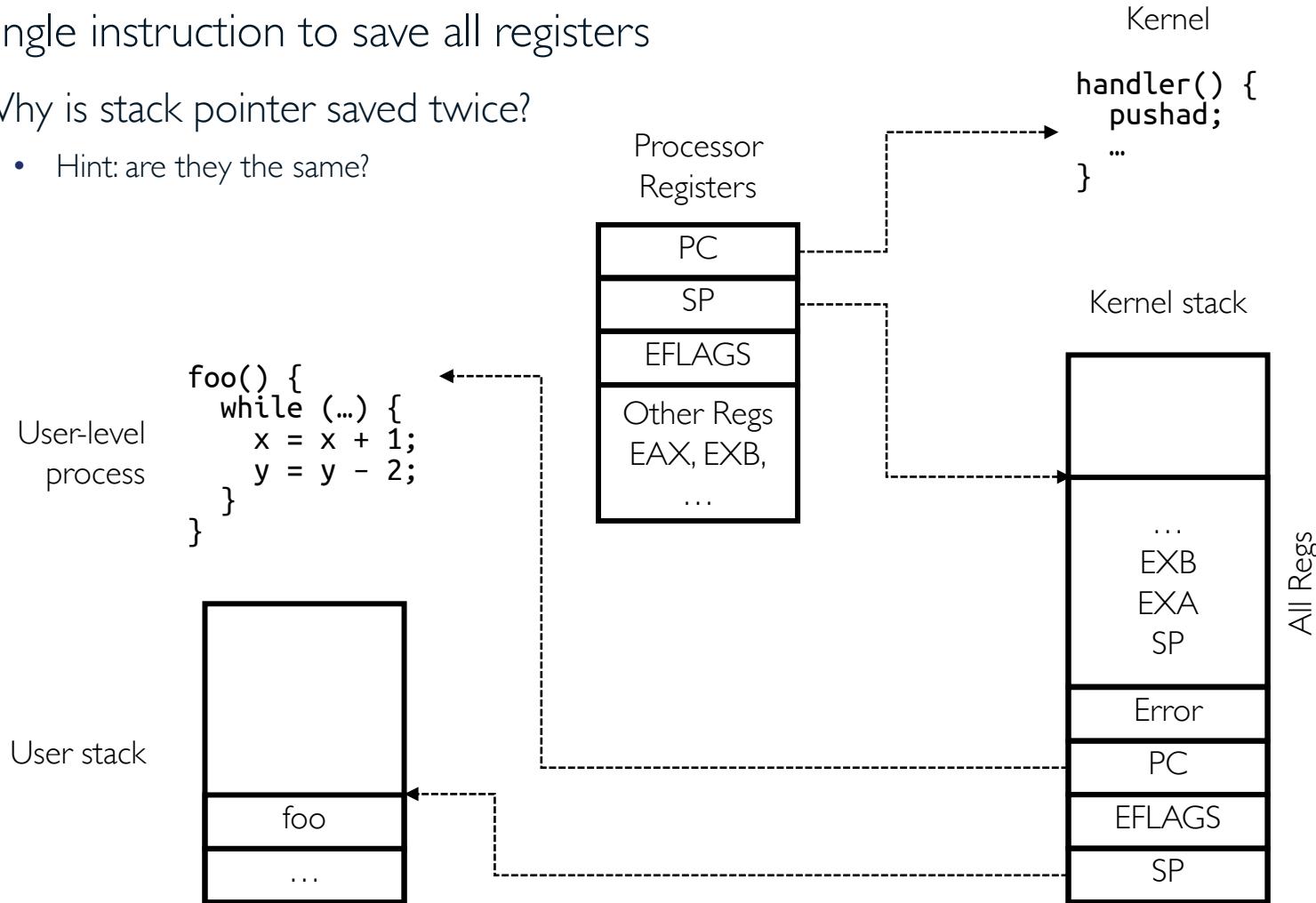
# Atomic Transfer of Control (cont.)

- Single instruction to
  - Save some registers (e.g., SP, PC)
  - Change PC and SP
  - Switch Kernel/user mode



# Atomic Transfer of Control (cont.)

- Single instruction to save all registers
- Why is stack pointer saved twice?
  - Hint: are they the same?



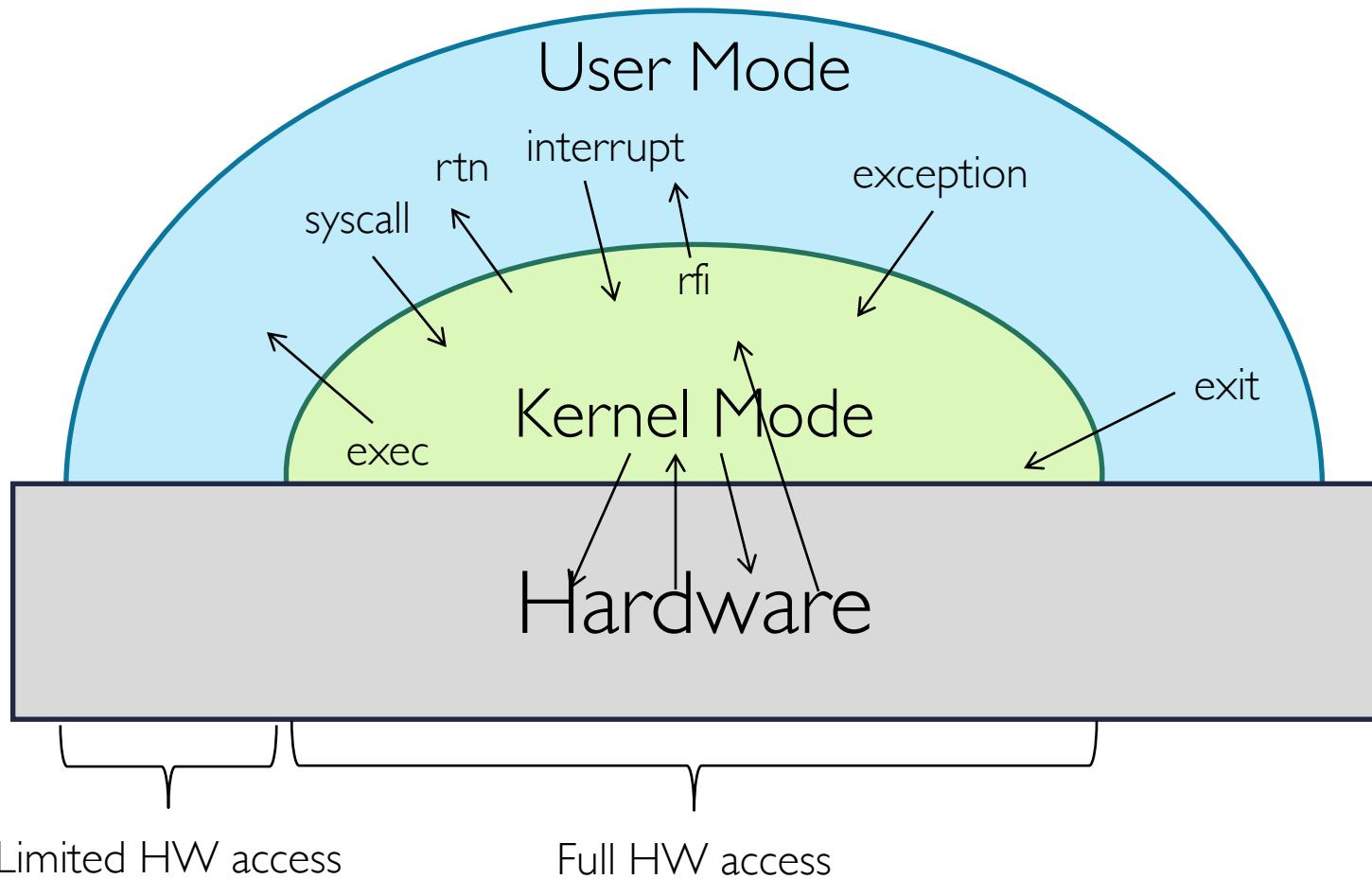
# Kernel to User Mode Switch

---

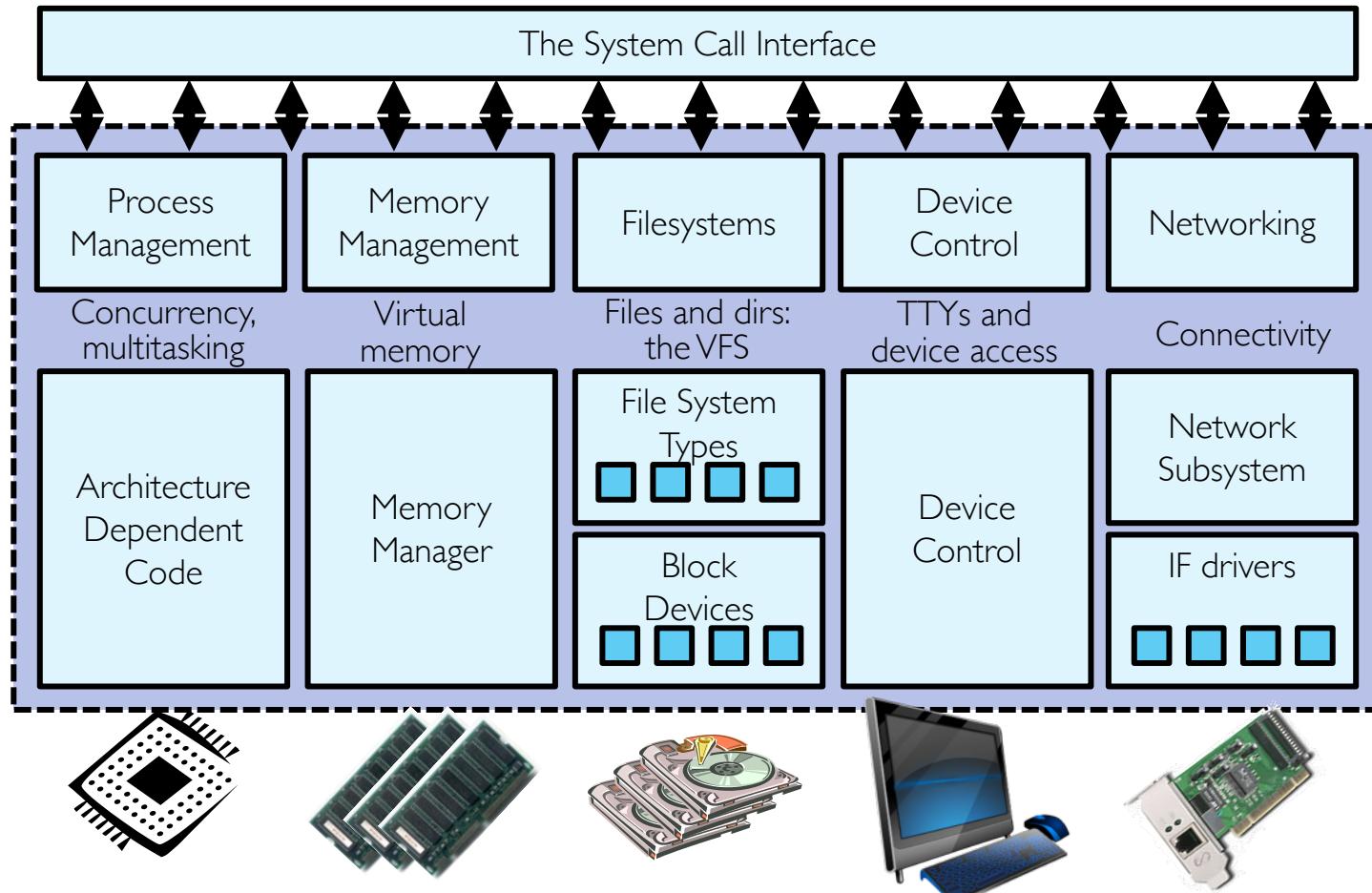
- New process/new thread start
  - Jump to first instruction in program/thread
- Return from interrupt, exception, system call
  - Resume suspended execution
- Process/thread context switch
  - Resume some other process
- User-level *upcall* (UNIX *signal*)
  - Asynchronous notification to user program
    - Preemptive user-level threads
    - Asynchronous I/O notification
    - Interprocess communication
    - User-level exception handling
    - User-level resource allocation

# Example: User/Kernel Mode Transfers

---



# System Call Interface: Access Point to Hardware Resources

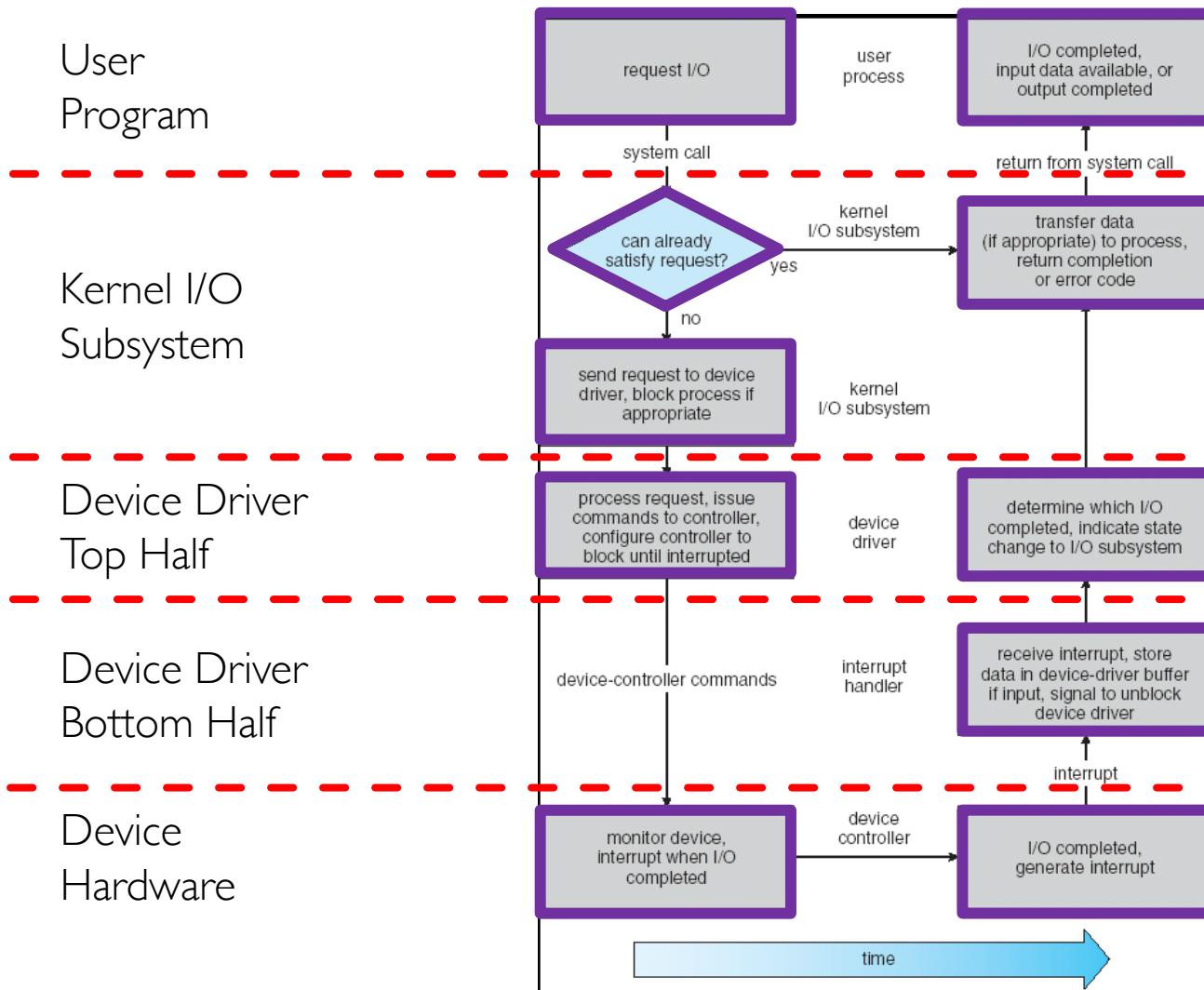


# Device Drivers

---

- Device-specific code in kernel that interacts directly with device hardware
  - Supports standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with `ioctl()` syscall
- Device drivers are typically divided into two pieces
  - Top half: accessed in call path from system calls
    - implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, etc.
    - This is kernel's interface to device driver
    - Top half will start I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

# Life Cycle of an I/O Request

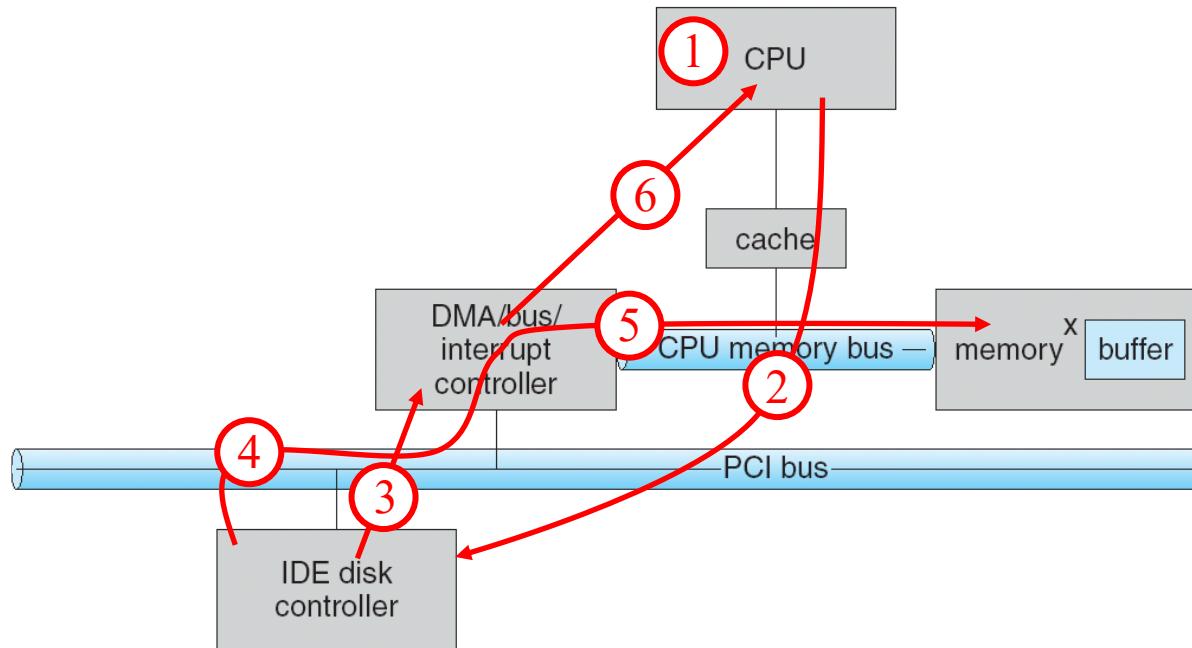


# I/O Data Transfer

---

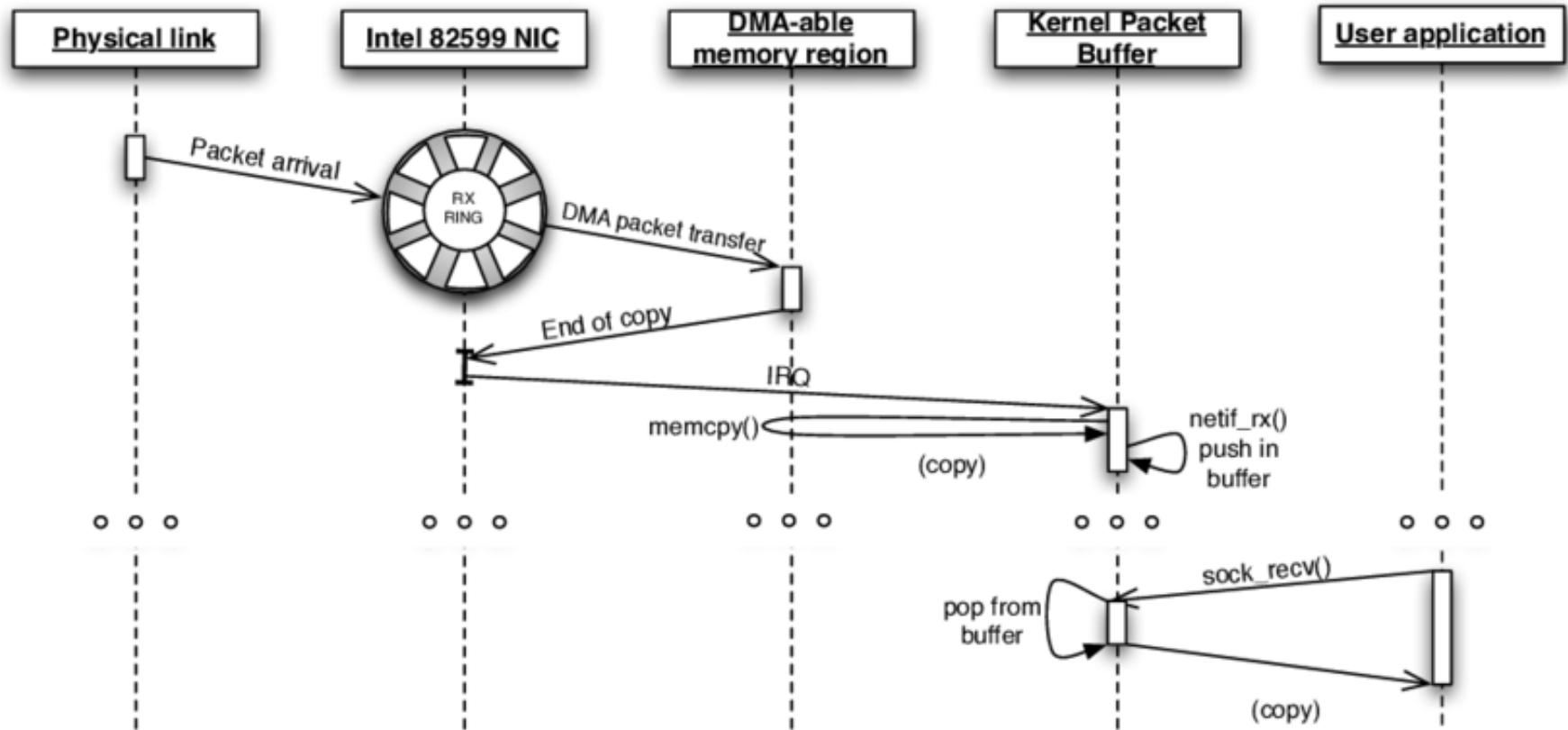
- Programmed I/O
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct memory access (DMA)
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

# DMA Transfer



1. Device driver is told to transfer disk data to buffer at address  $x$
2. Device driver tells disk controller to transfer  $C$  bytes from disk to buffer at address  $x$
3. Disk controller initiates DMA transfer
4. Disk controller send each byte to DMA controller
5. DMA controller transfers bytes to buffer  $x$ , increasing address and decreasing  $C$
6. When  $C = 0$ , DMA interrupts CPU to signal transfer completion

# DMA Example: Network Stack in Linux Kernels before 2.6



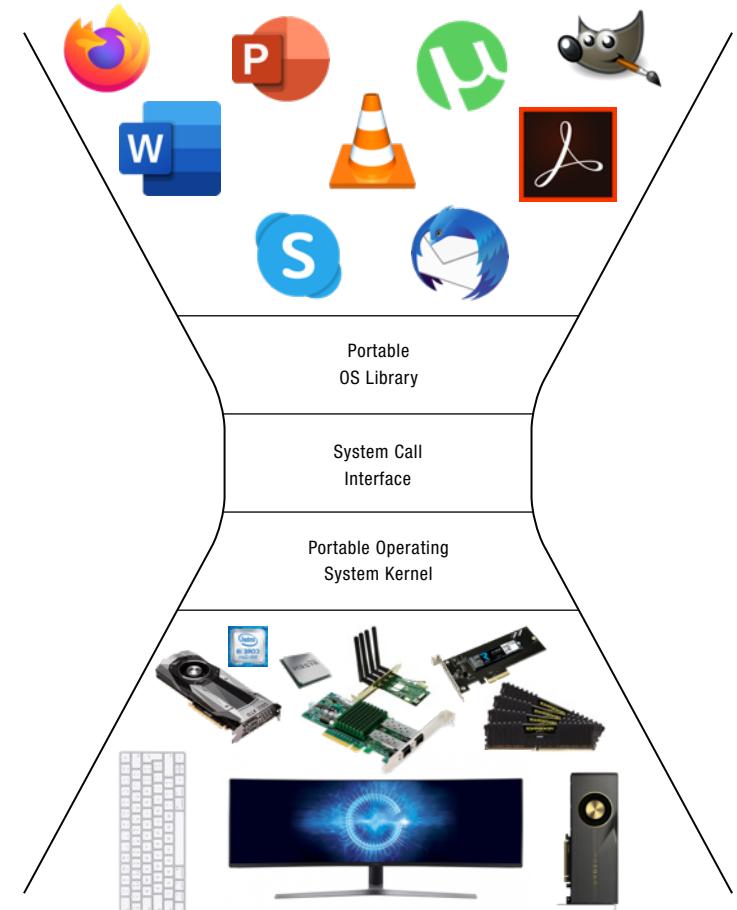
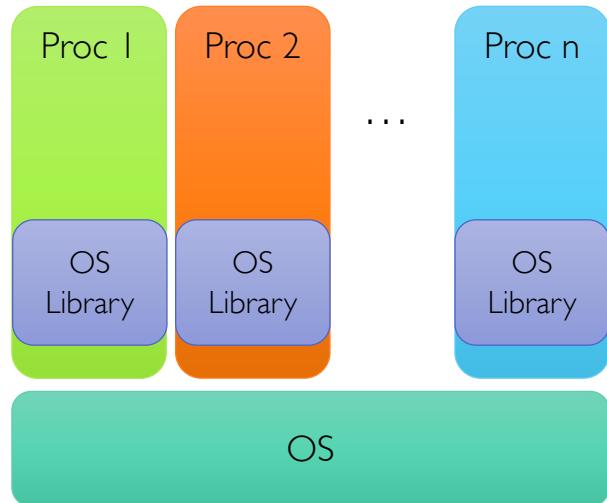
# How Does Kernel Provide Services?

---



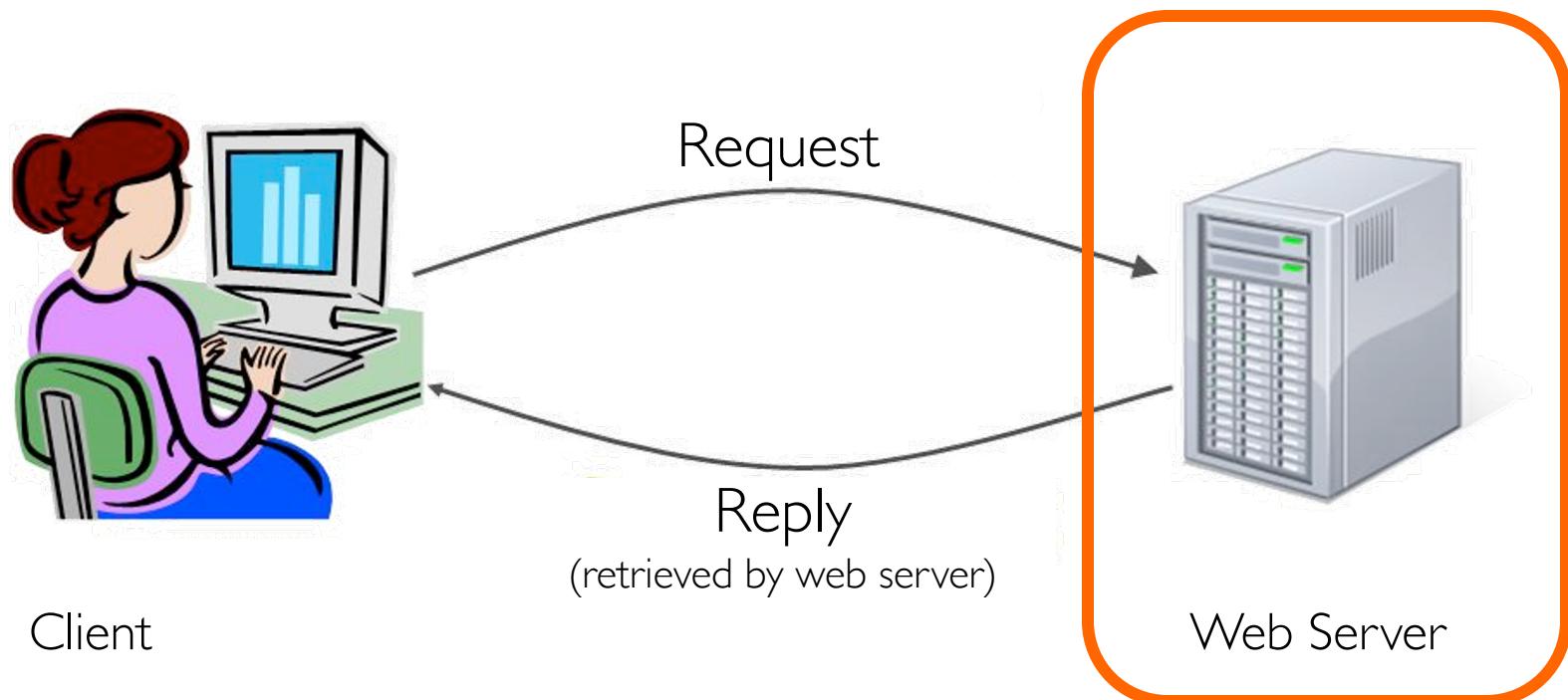
- You said that applications request services from OS via syscall, but ...
  - I've been writing all sorts of applications, and I never ever saw a "syscall" !!!
- That's right!
- It was buried in the programming language runtime library (e.g., libc.a)
  - ... Layering

# OS Run-time Library

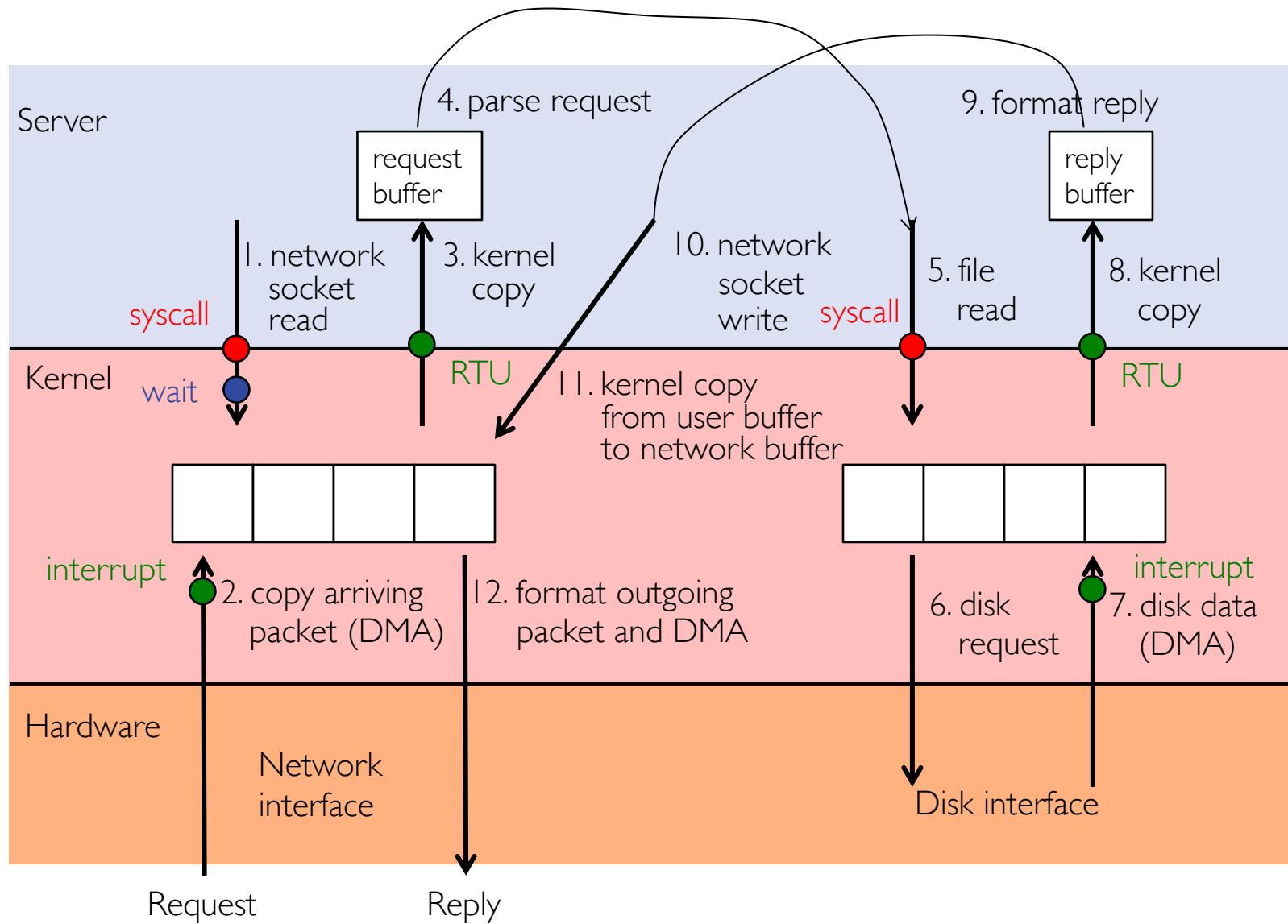


# Putting it Together: Web Server

---



# Putting it Together: Web Server (cont.)



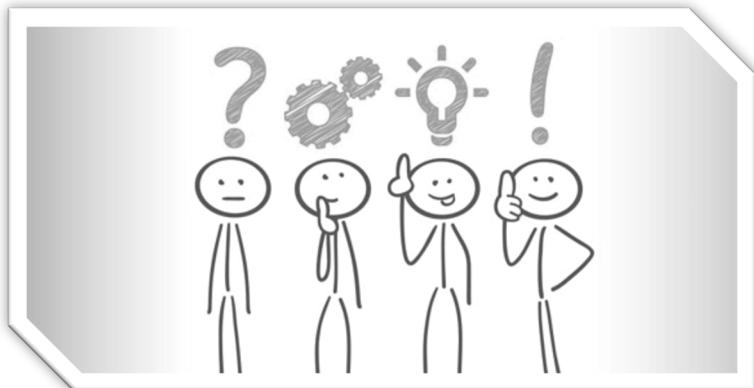
# Summary: Four Fundamental OS Concepts

---

- Thread
  - Single unique execution context which fully describes program state
  - Program counter, registers, execution flags, stack
- Address space (with translation)
  - Address space which is distinct from machine's physical memory addresses
- Process
  - Instance of executing program consisting of address space and 1+ threads
- Dual-mode operation/protection
  - Only "system" can access certain resources
  - OS and hardware are protected from user programs
  - User programs are isolated from one another by controlling translation from program virtual addresses to machine physical addresses

# Questions?

---



# Acknowledgment

---

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny