Operating System

2/2563

Course Outline

- Introduction
- Kernel
- Structure
- Concurrency
- Synchronization + Advance Synchronization
- Scheduling
- Address Translation
- Caching + Virtual Memory

Tools

- C#

References

- Thomas Anderson and Michael Dahlin, Operating Systems Principles & Practice Volume I, II, III, IV, 2nd edition, Recursive Books, 2015
- Any Operating Systems Books

Introduction

2/2563

Roles of the Operating System

What is an Operating System?

Operating System Evaluation

- Reliability and Availability

- Reliability and Availability
- Security
- Portability
  - AVM, API, HAL
- Performance
  - Overhead, efficiency
  - Fairness, response time, throughput
  - Performance predictability
- Adoption

Design Tradeoffs

- Must balance between the 5s

- Examples
  - Preserves legacy API → Portability ↑, reliable ↓, secure ↓
  - Breaking an abstraction → Performance ↑, Portability ↓, Reliability ↓

Computer Performance Over Time

	1981	1997	2014	Factor (2014/1981)
Uniprocessor speed (MIPS)	1	200	2500	2.5K
CPUs per computer	1	1	10+	10+
Processor MIPS/$	$100K	$25	$0.20	500K
DRAM Capacity (MiB)/$	0.002	2	1K	500K
Disk Capacity (GiB)/$	0.003	7	25K	10M
Home Internet	300 bps	256 Kbps	20 Mbps	100K
Machine room network	10 Mbps (shared)	100 Mbps (switched)	10 Gbps (switched)	1000
Ratio of users to computers	100:1	1:1	1:several	100+

From Thomas Anderson and Michael Dahlin, Operating Systems Principles & Practice Volume I, 2nd edition, Recursive Books, 2015

Early Operating Systems: Computers Very Expensive

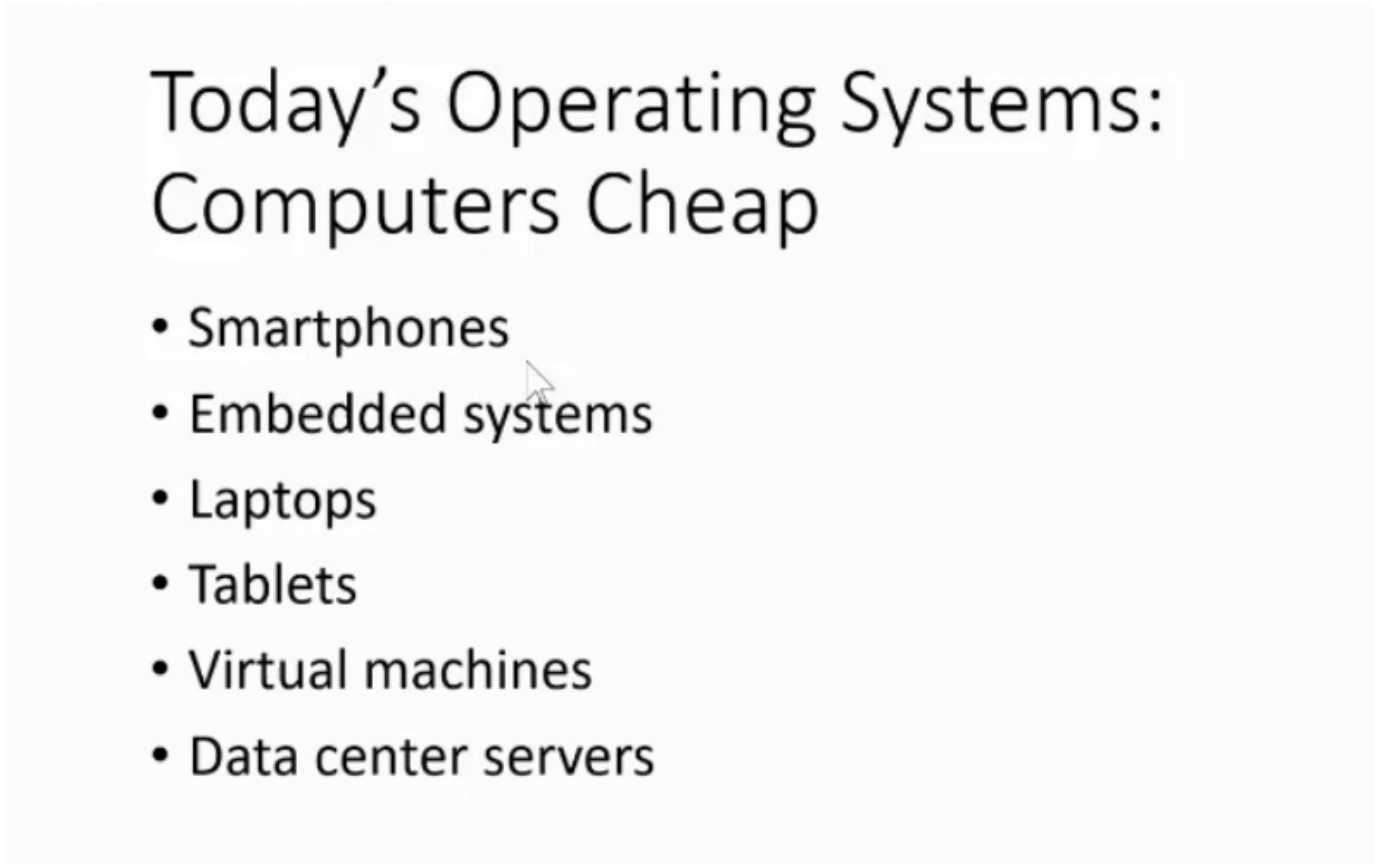
- One application at a time
  - Had complete control of hardware
  - OS was runtime library
  - Users would stand in line to use the computer
- Batch systems
  - Keep CPU busy by having a queue of jobs
  - OS would load next job while current one runs
  - Users would submit jobs, and wait, and wait, and

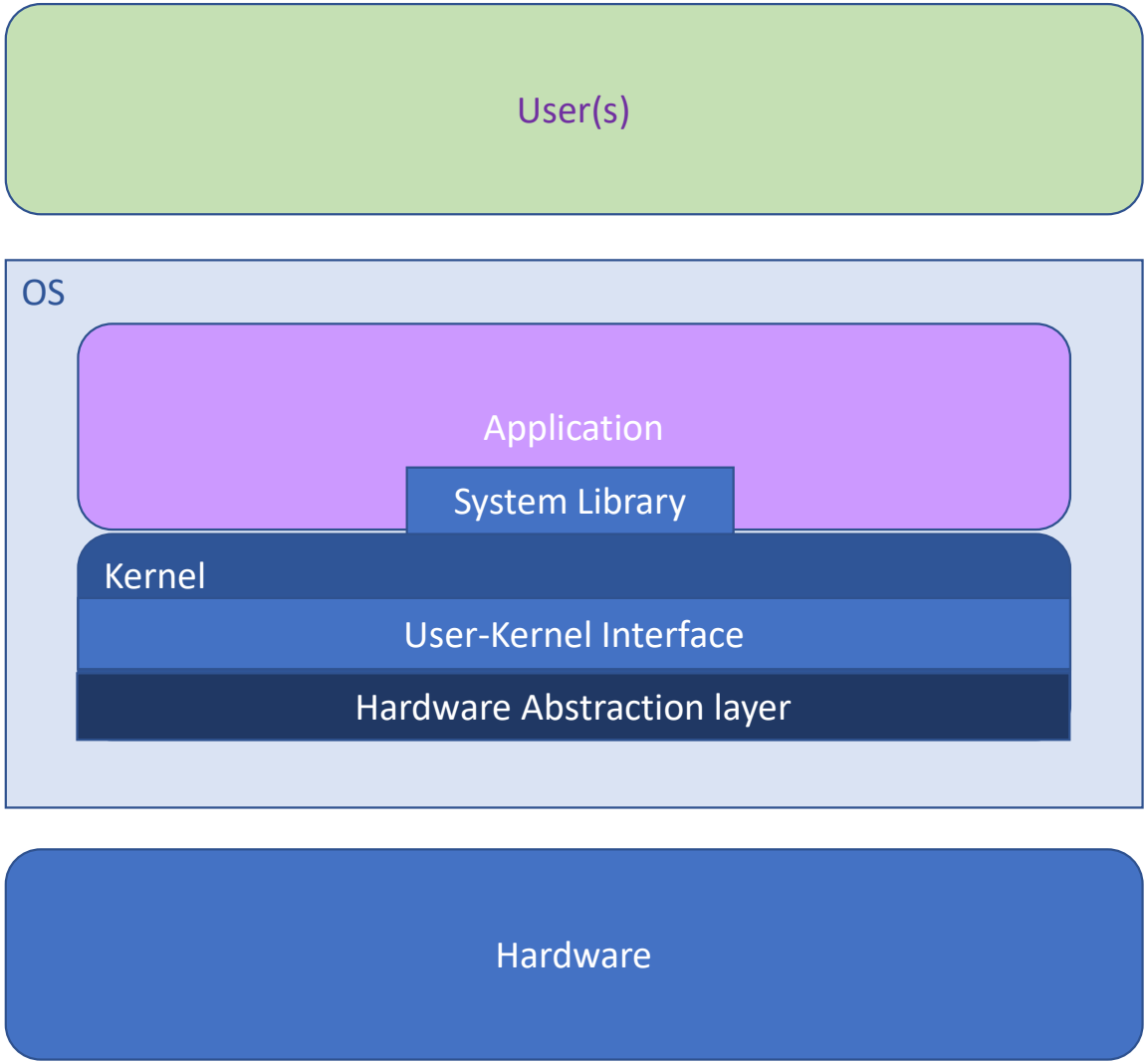
Time-Sharing Operating Systems: Computers and People Expensive

Time-Sharing Operating Systems:
Computers and People Expensive

- Multiple users on computer at same time
  - Multiprogramming: run multiple programs at same time
  - Interactive performance: try to complete everyone's tasks quickly
  - As computers became cheaper, more important to optimize for user time, not computer time

Today’s Operating Systems: Computers Cheap




User(s)
OS
Application
System Library
Kernel
User-Kernel Interface
Hardware Abstraction layer
Hardware

Tomorrow's Operating Systems

- Giant-scale data centers
- Increasing numbers of processors per computer
- Increasing numbers of computers per user
- Very large scale storage

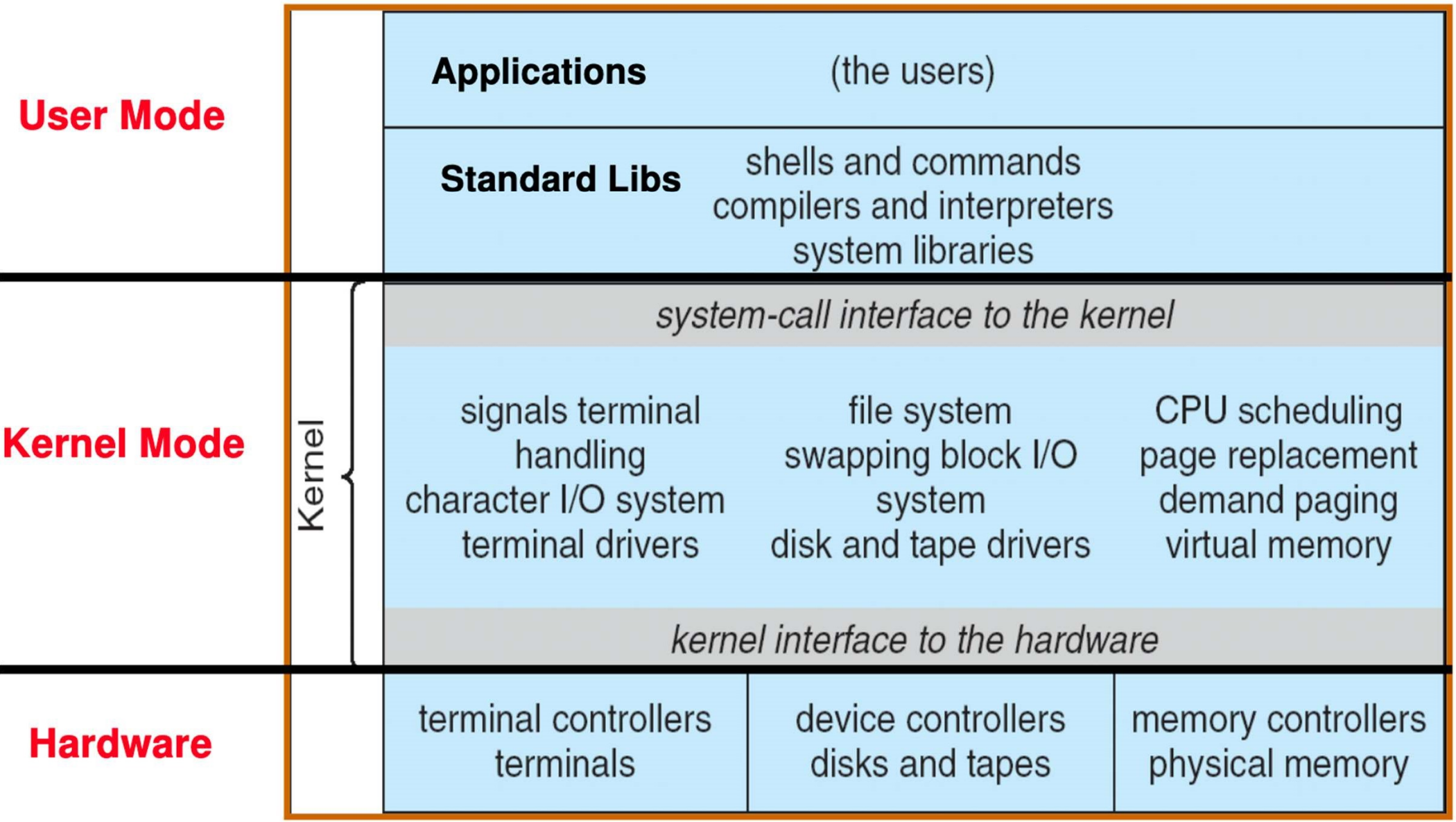
The Kernel Abstraction

Slides adopted from CSE 451 class at UW, CS162 class at Berkeley and CSE 421/521 class at UB

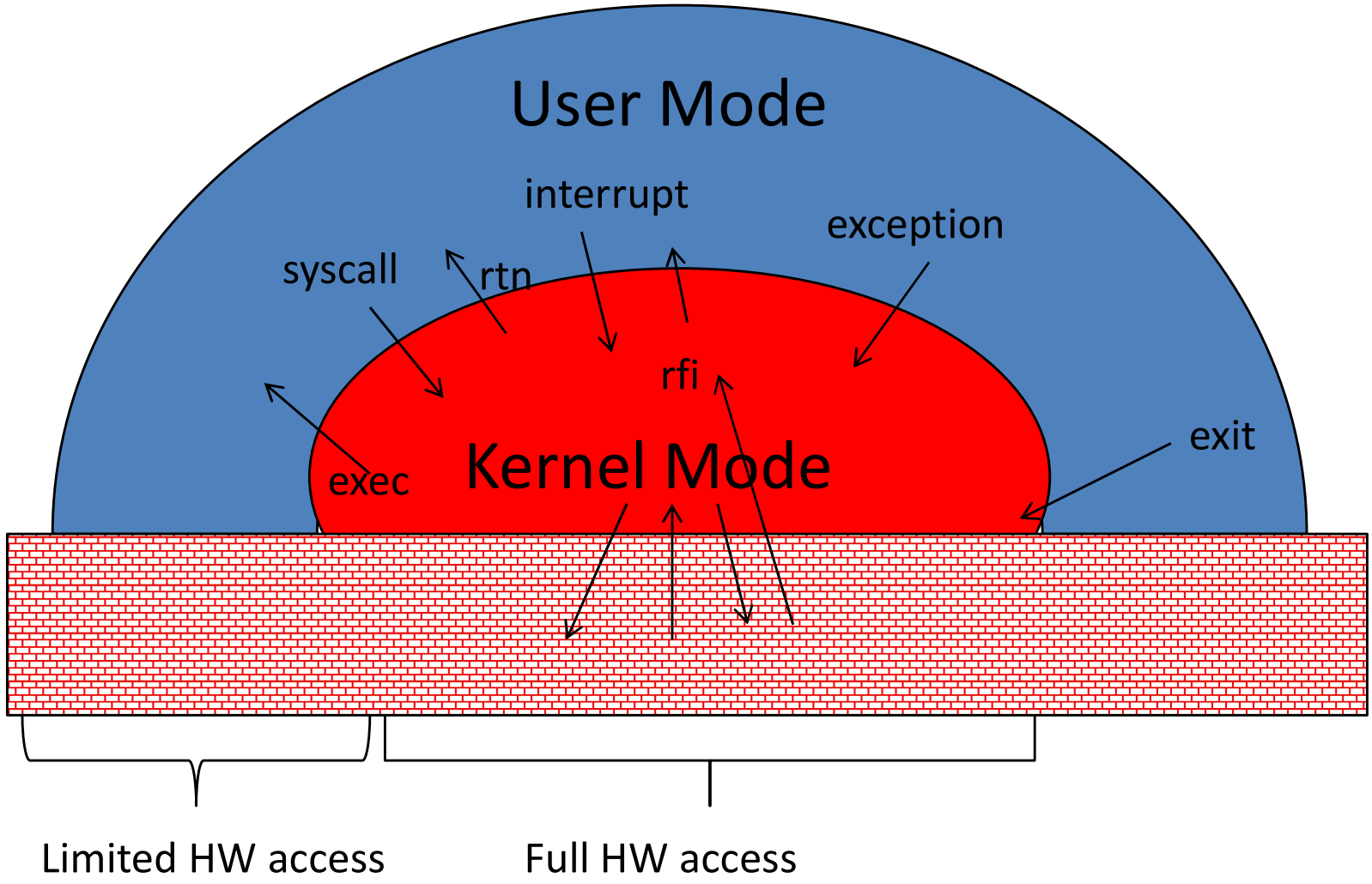
What is an OS…

- 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



User/Kernel (Privileged) Mode



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

Protection: WHY?

เวลา 10 นาที

Protection: How? (HW/SW)

เวลา 10 นาที

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

Privileged instructions

- Examples?

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

User Mode

- Application program
  - Running in process

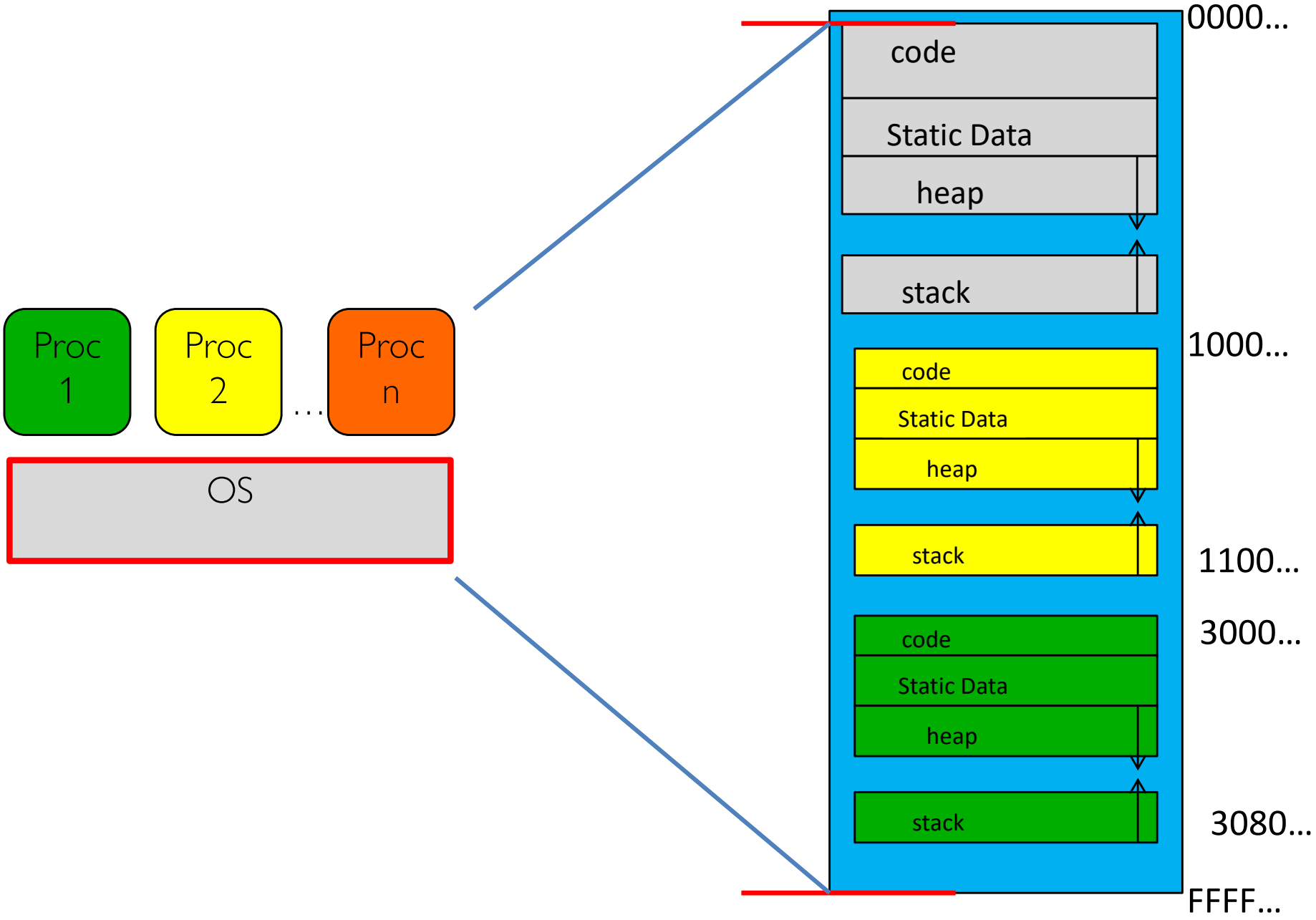
Virtual Machine:VM

- 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
    - Write the program for the OS rather the HW

Kernel mode & User mode

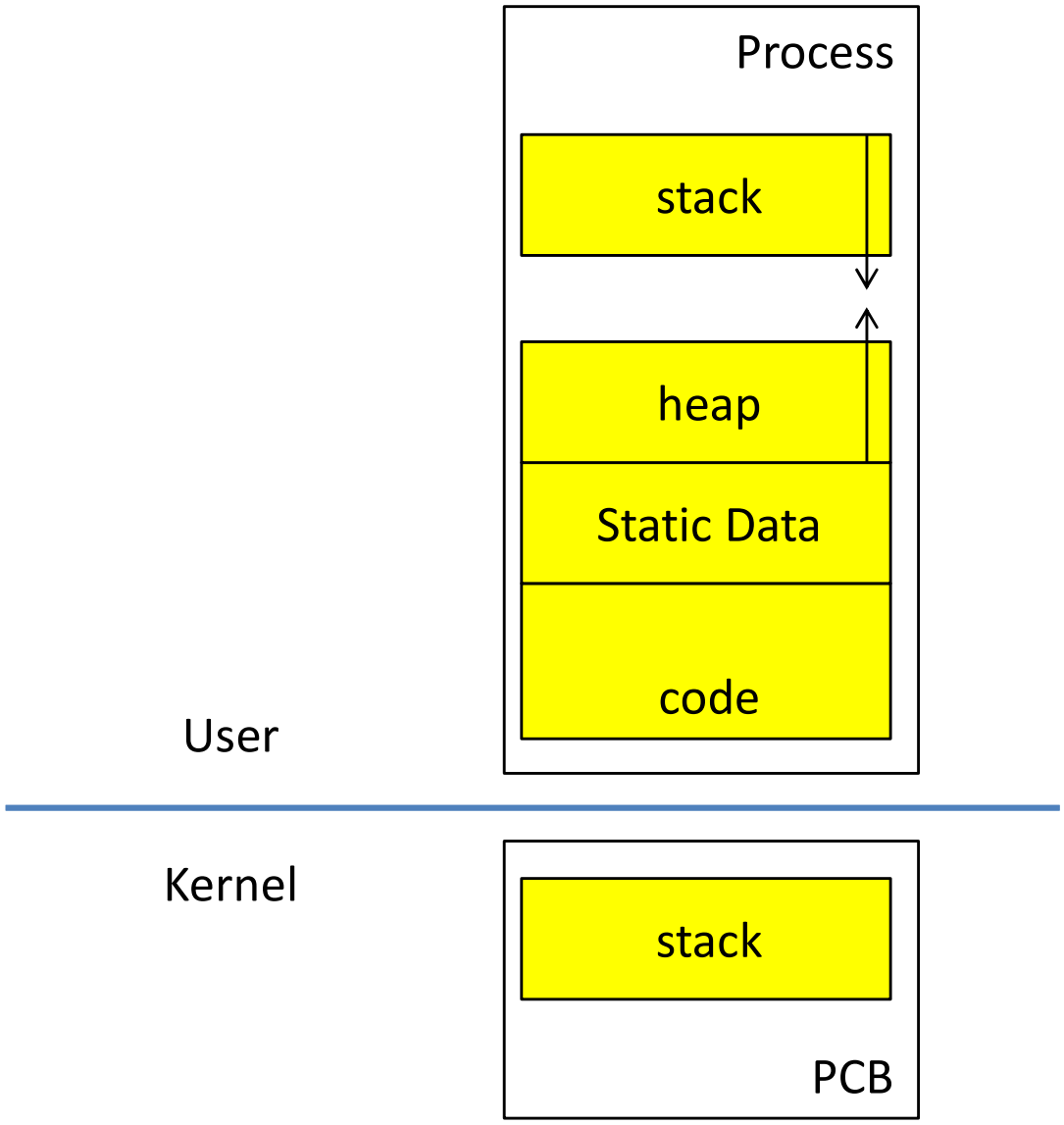


Process Abstraction

- 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

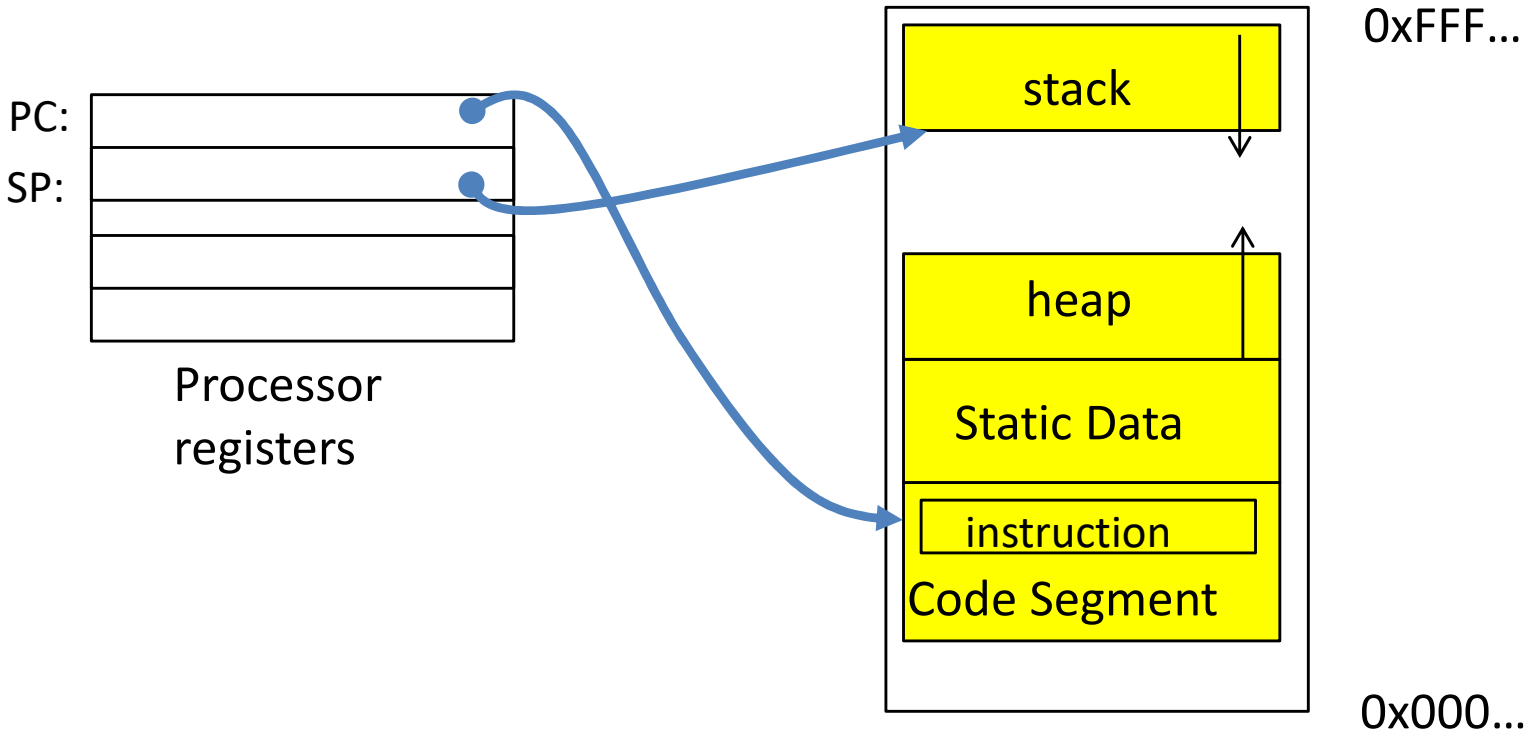
- 2 parts
  - PCB in kernel
  - Others in user



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



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

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

- 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

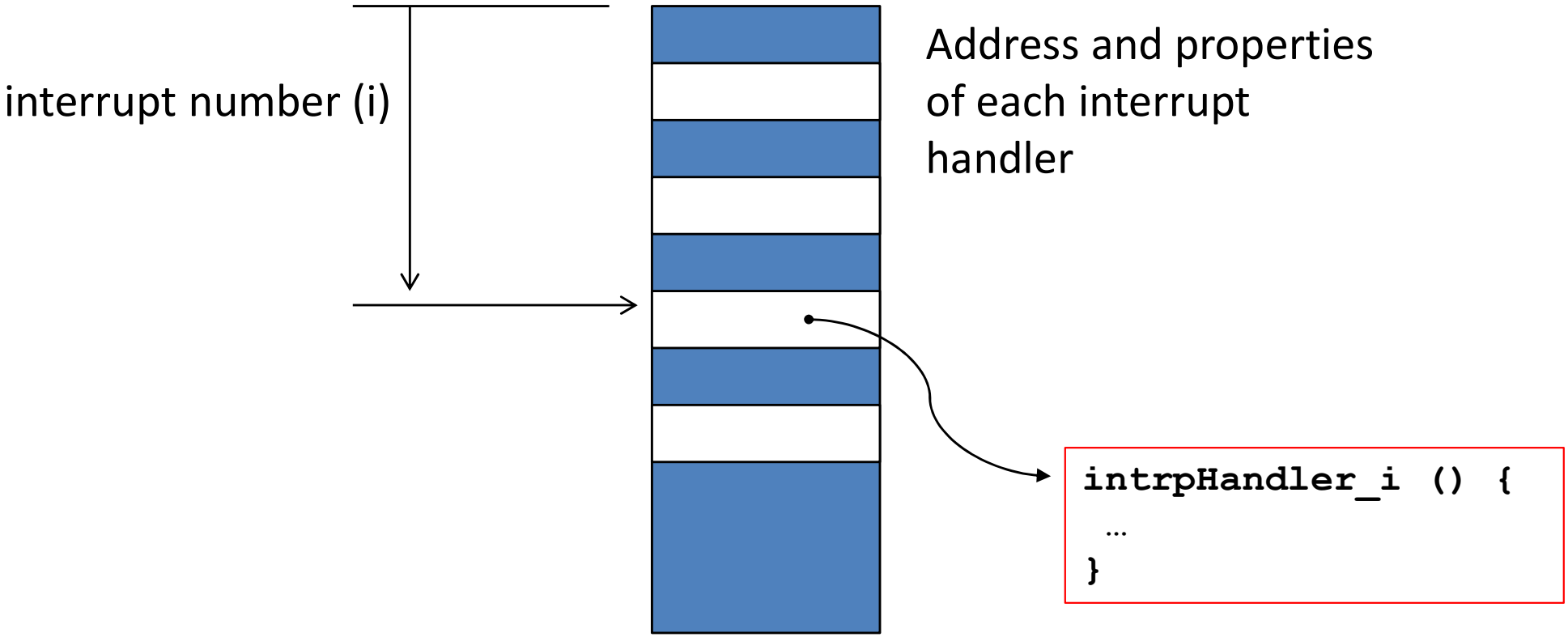
- How do device interrupts work?
  - Where does the CPU run after an interrupt?
  - What stack does it use?
  - Is the work the CPU had been doing before the interrupt lost forever?
  - If not, how does the CPU know how to resume that work?

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***

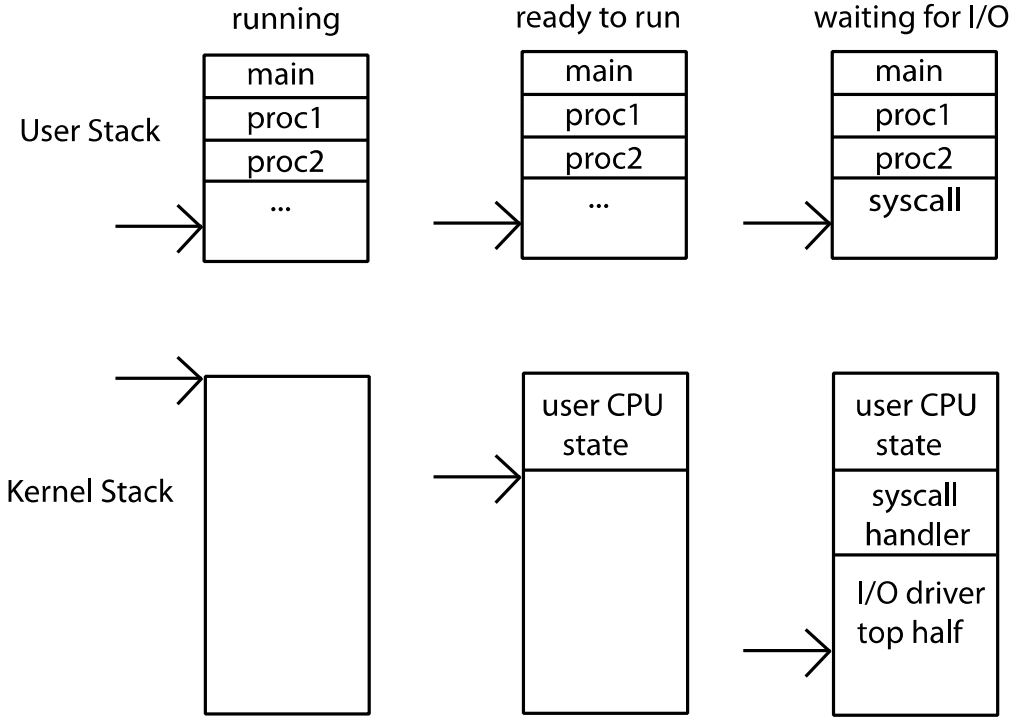


The Kernel Stack

- 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



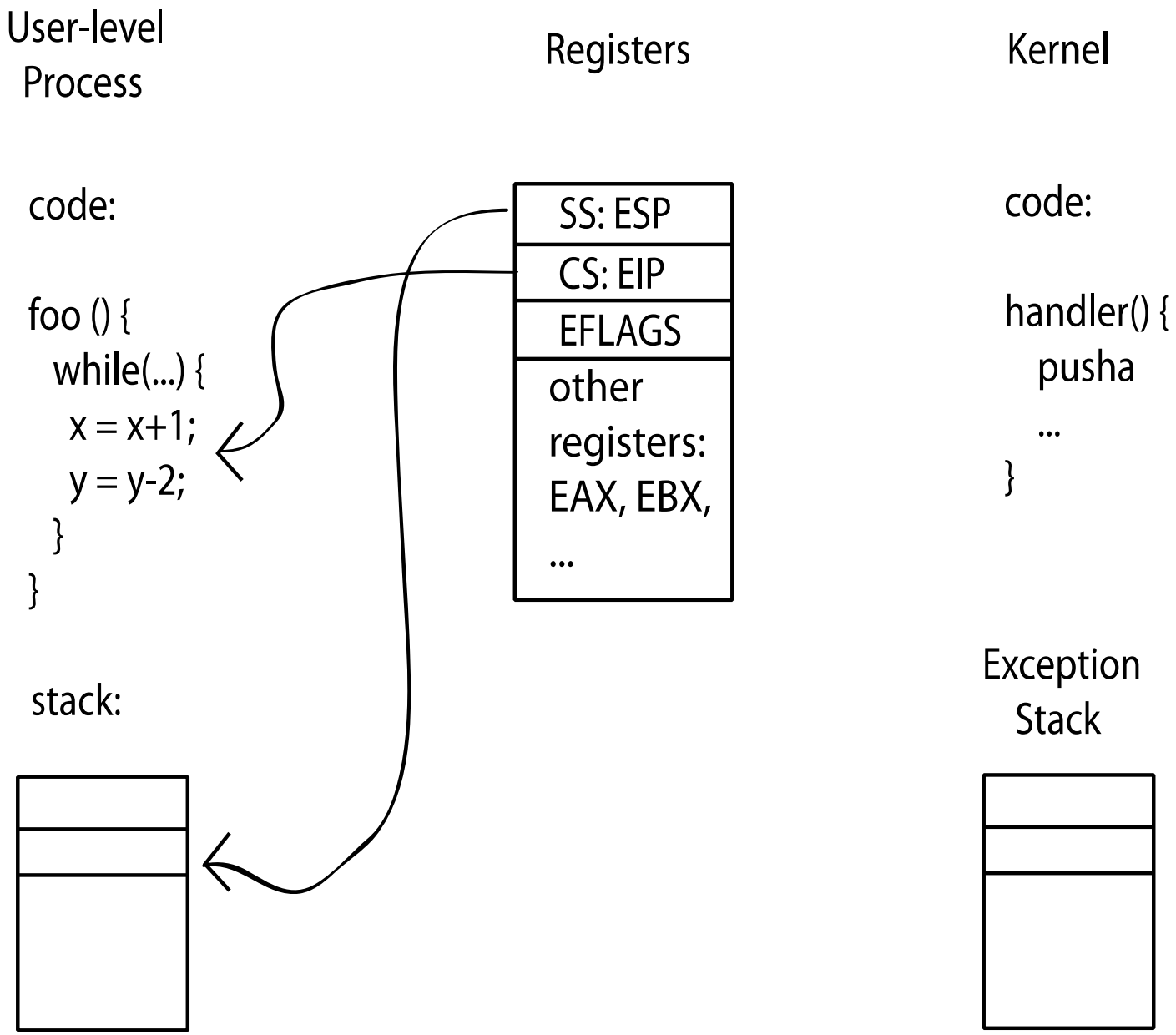
Interrupt Stack

- 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?

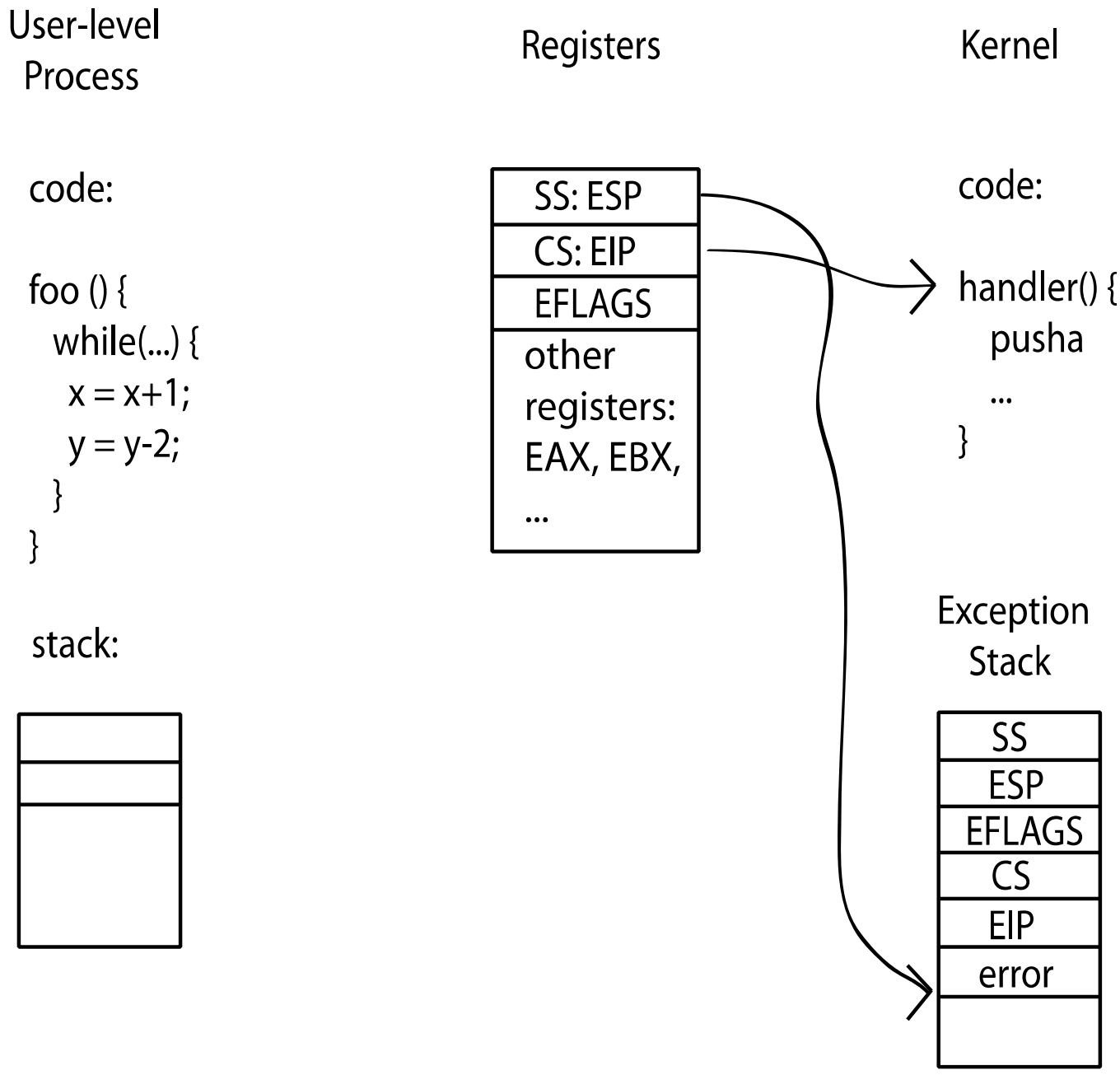
Case Study: x86 Interrupt

- 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

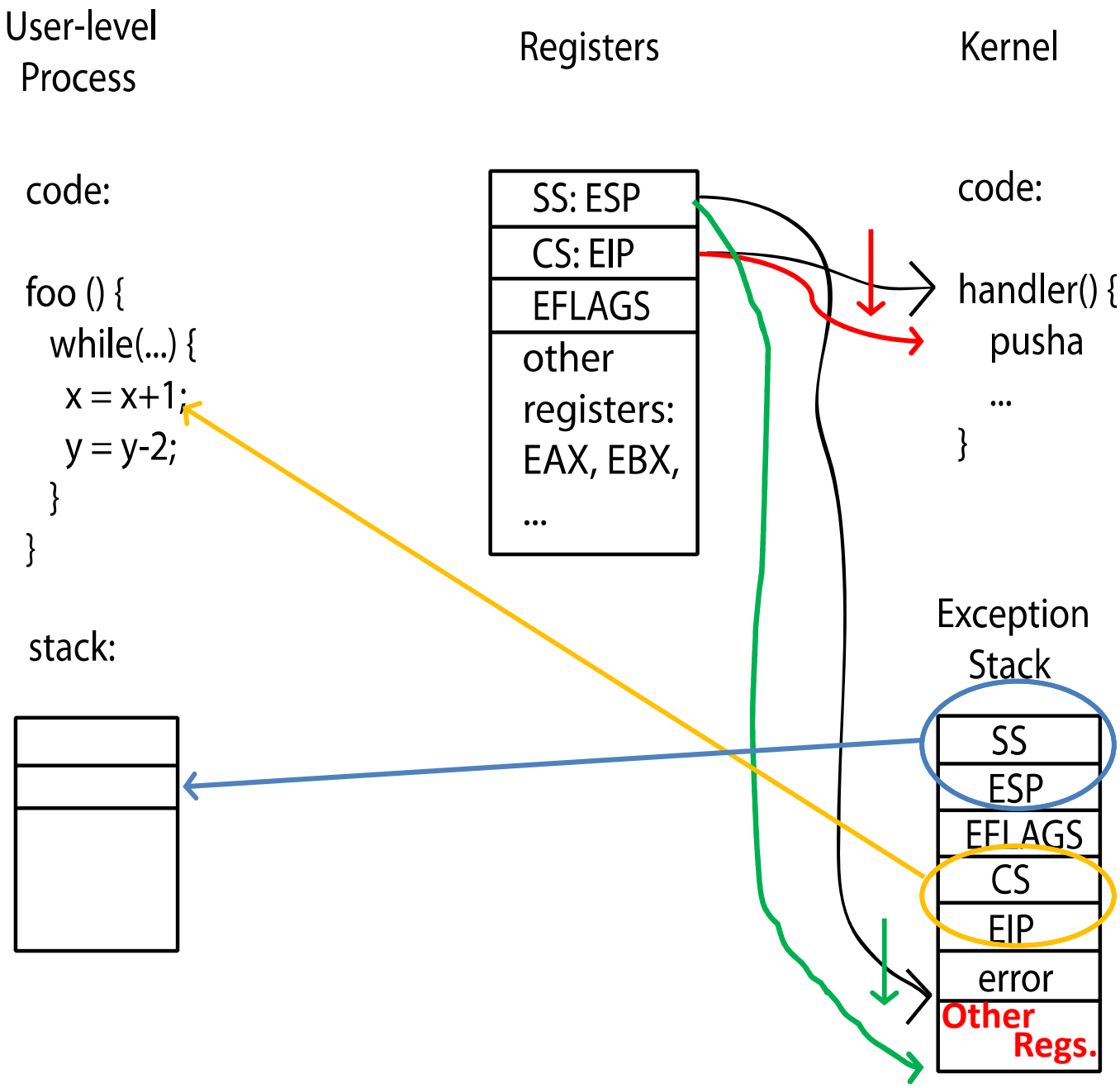
Before Interrupt



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

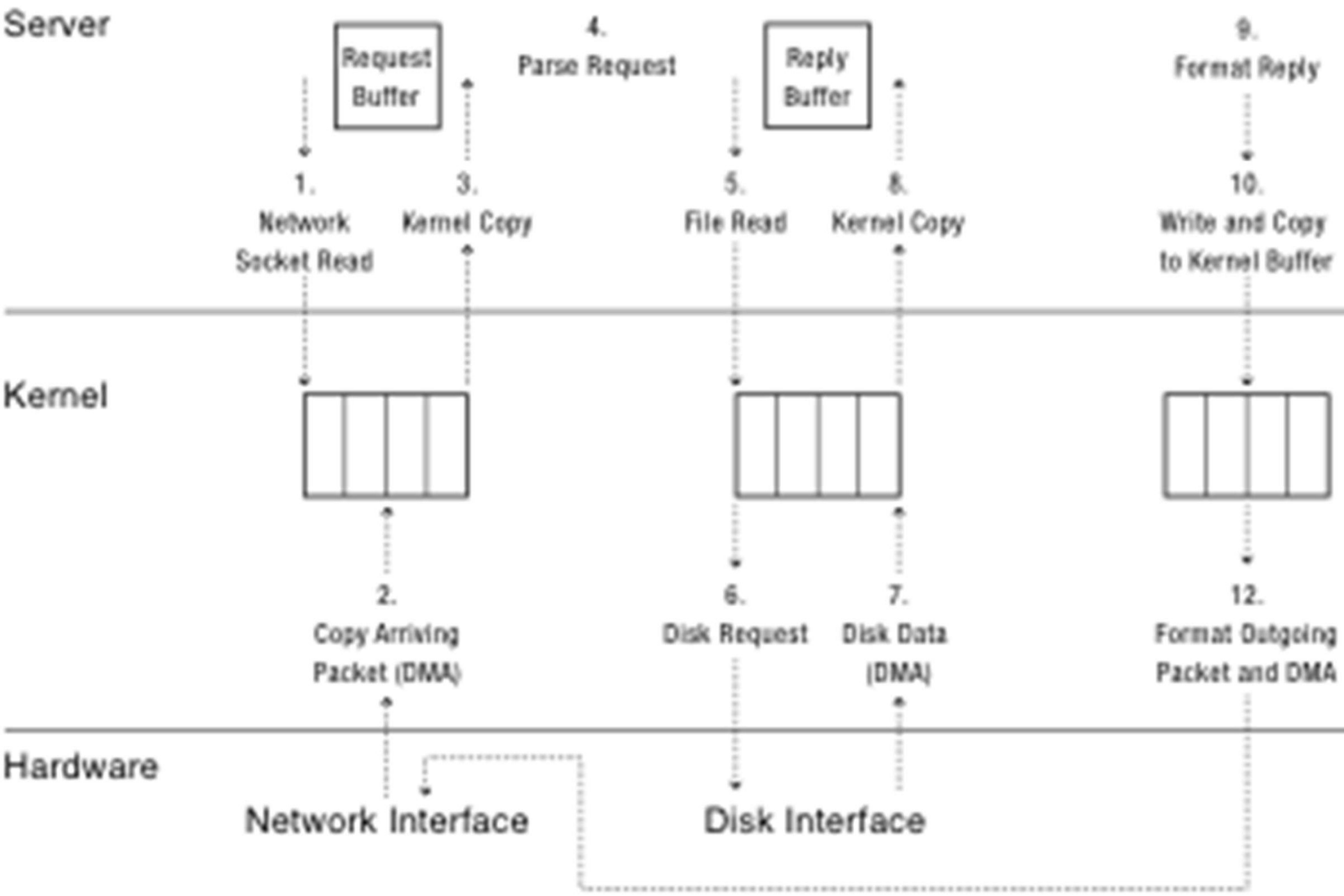
Hardware support: Interrupt Control

- 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

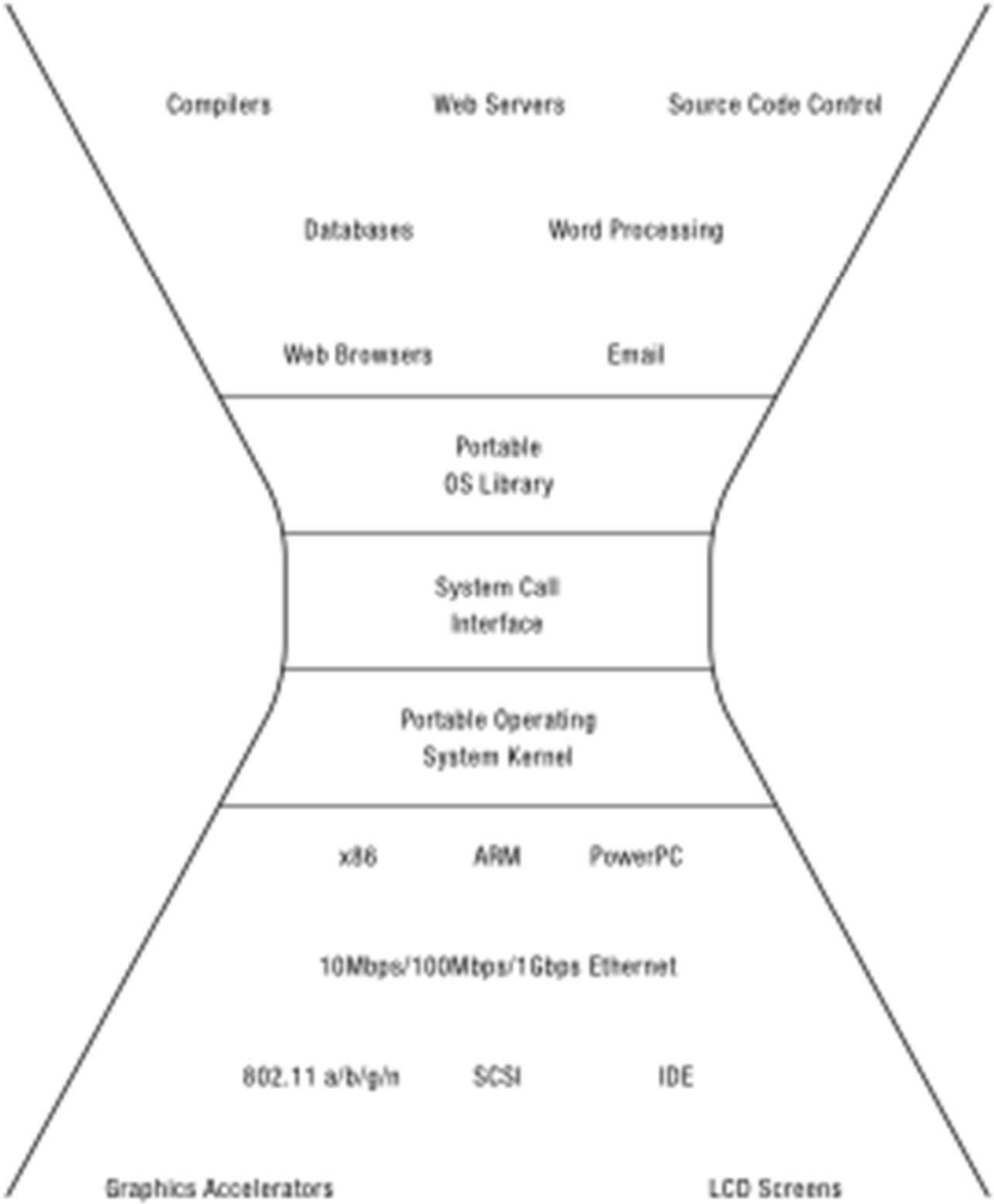
- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory – carefully checking locations!
  - Protect kernel from malicious code evading checks
-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


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

The Process
and
Programming Interface


Compilers
Web Servers
Source Code Control
Databases
Word Processing
Web Browsers
Email
Portable
OS Library
System Call
Interface
Portable Operating
System Kernel
x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
802.11 a/b/g/n
SCSI
IDE
Graphics Accelerators
LCD Screens

Main Points

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Communicating between processes
  - pipe, dup, select, connect
- Example: implementing a shell

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program

  cc –c sourcefile1.c

  cc –c sourcefile2.c

  ln –o program sourcefile1.o sourcefile2.o

Question

- If the shell runs at user-level, what system calls does it make to run each of the programs?
  - Ex: cc, ln

Windows CreateProcess

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''
  - Inform the scheduler that the new process is ready to run

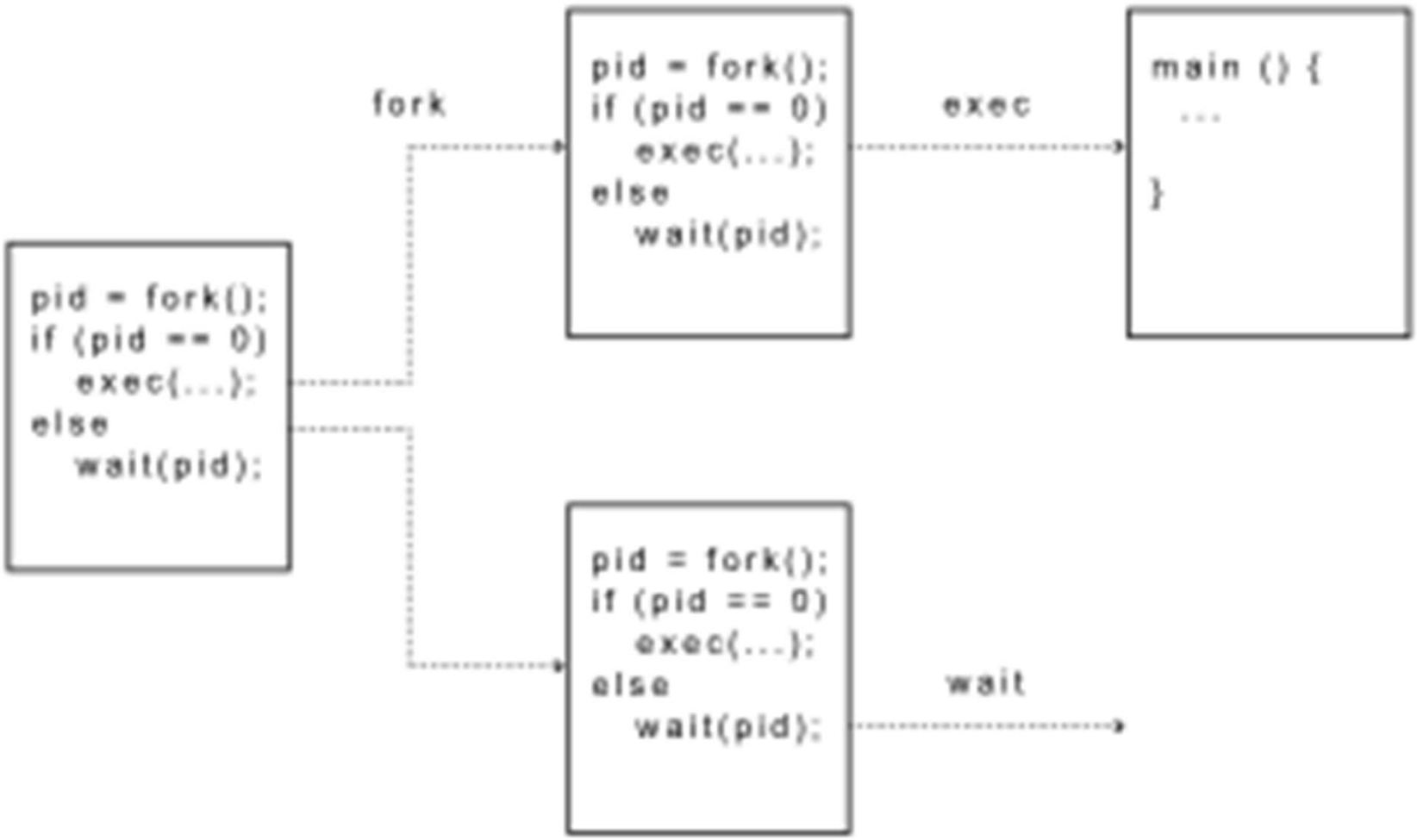
Windows CreateProcess API (simplified)

```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Pointer to PROCESS_INFORMATION structure
)
```

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {          // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                       // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

Implementing UNIX exec

- Steps to implement UNIX fork
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''

UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
  - To garbage collect the open file descriptor

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - …

Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
    create(name);   // can create fail?
fd = open(name);   // does the file exist?
```

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // create a child process
    if (child_pid == 0) {
       exec(prog, args);       // I'm the child process.  Run program
      // NOT REACHED
    } else {
       wait(child_pid);       // I'm the parent, wait for child
       return 0;
    }
}
```

In Unix

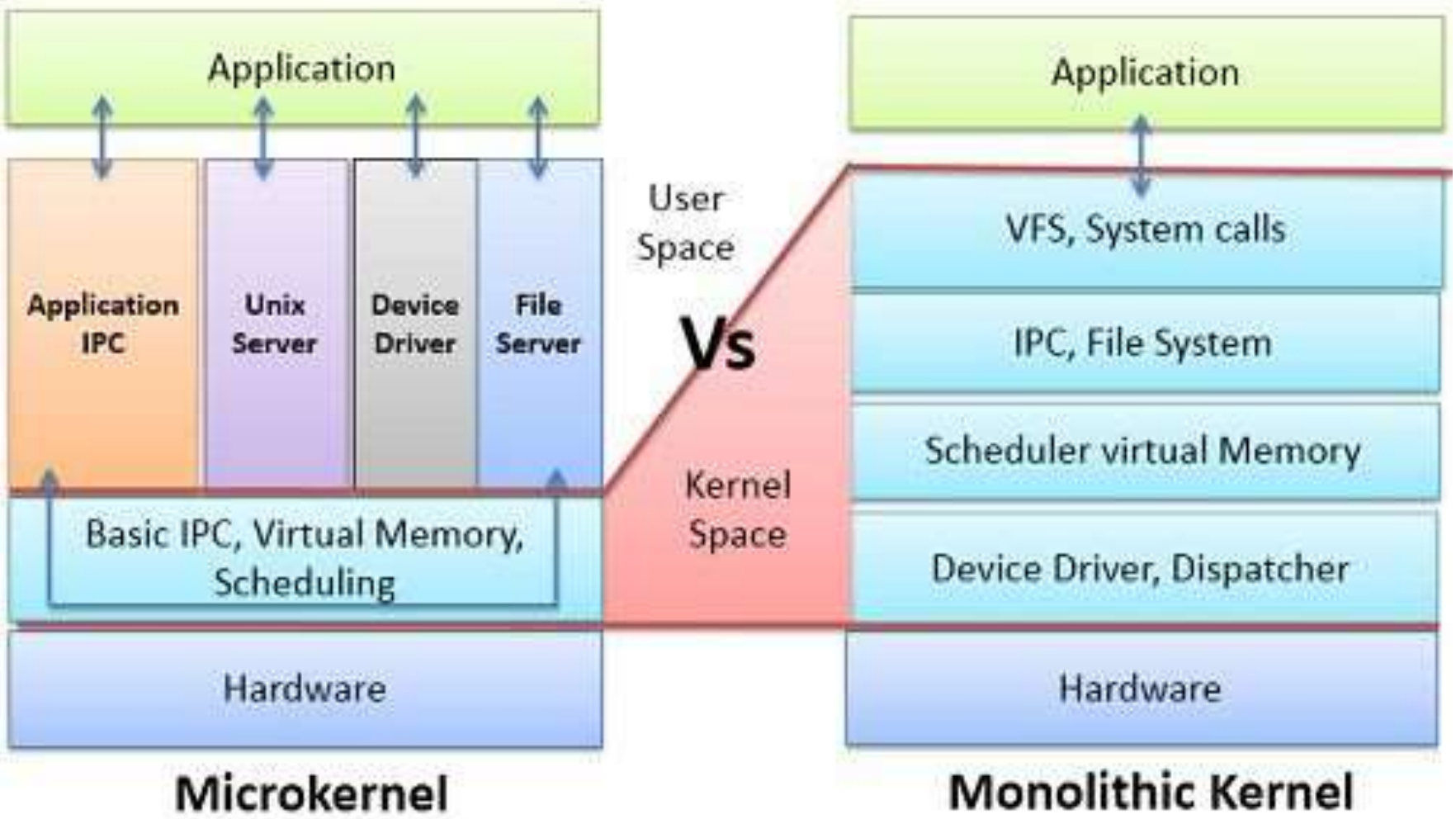
- A program can be a file of commands
- A program can send its output to a file
- A program can read its input from a file
- The output of one program can be the input to another program

Interprocess Communication

- Producer-consumer
  - Output of one program is accepted as input of another program
    - One-way communication
    - Pipe
  - Client-server
    - Two-way communication
    - Server implements specialize task
      - Print serve
  - File system
    - Write data to a file then read file as an input
    - Reader and writer are not need to running at the same time

Operating system structure

- Monolithic kernel
- Microkernel



Source:https://techdifferences.com/difference-between-microkernel-and-monolithic-kernel.html

Concurrency

Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks, system maintenance
- Humans are not very good at keeping track of multiple things happening simultaneously
- Threads are an abstraction to help bridge this gap

Why Concurrency?

- Servers
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

Definitions

- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain

Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

Possible Executions

Thread Operations

- thread_create(thread, func, args)
  - Create a new thread to run func(args)
- thread_yield()
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- thread_exit
  - Quit thread and clean up, wake up joiner if any

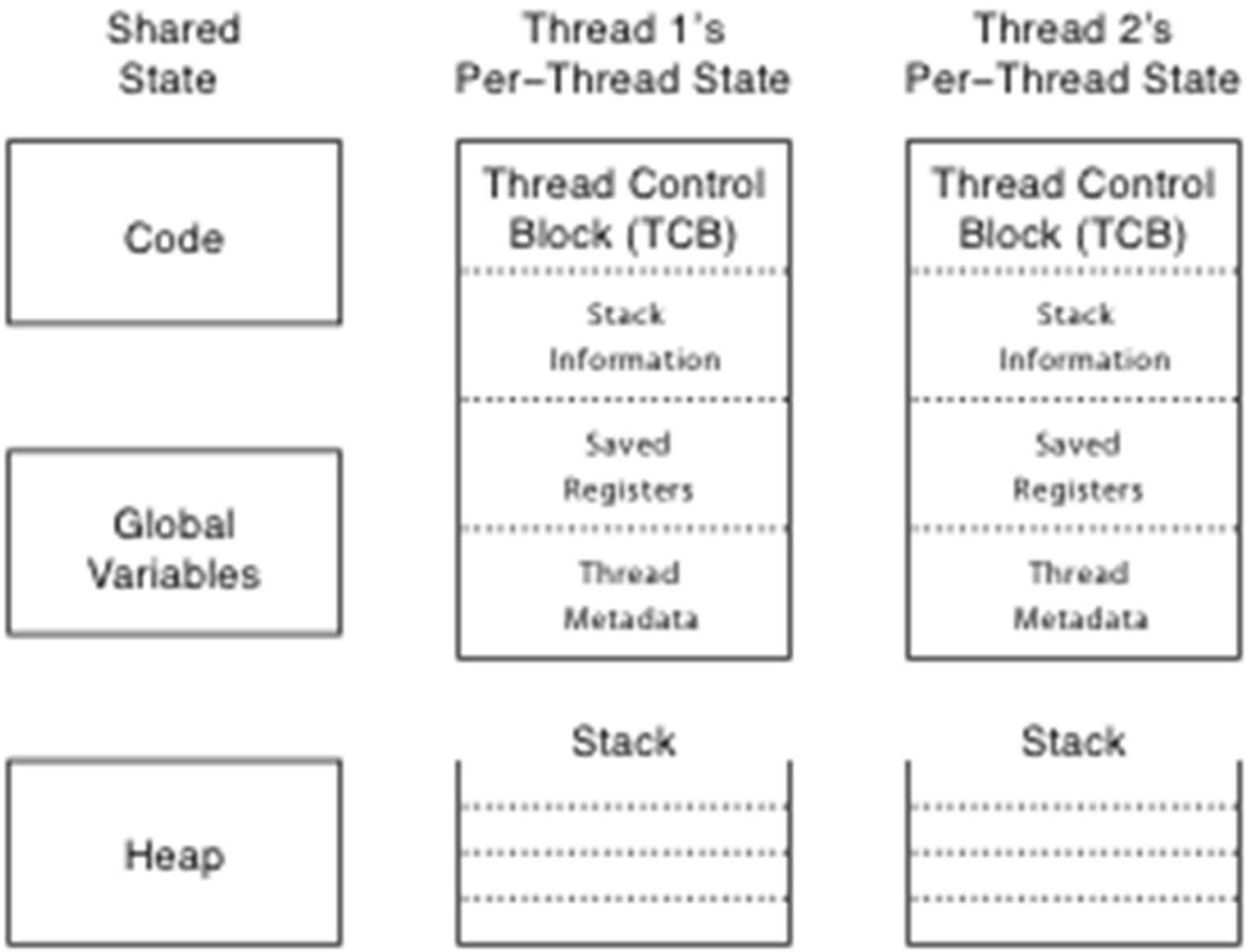
Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
   for (i = 0; i < NTHREADS; i++)  thread_create(&threads[i], &go, i);
   for (i = 0; i < NTHREADS; i++) {
      exitValue = thread_join(threads[i]);
      printf("Thread %d returned with %ld\n", i, exitValue);
   }
   printf("Main thread done.\n");
}
void go (int n) {
   printf("Hello from thread %d\n", n);
   thread_exit(100 + n);
   // REACHED?
}
```

Fork/Join Concurrency

- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

Thread Data Structures

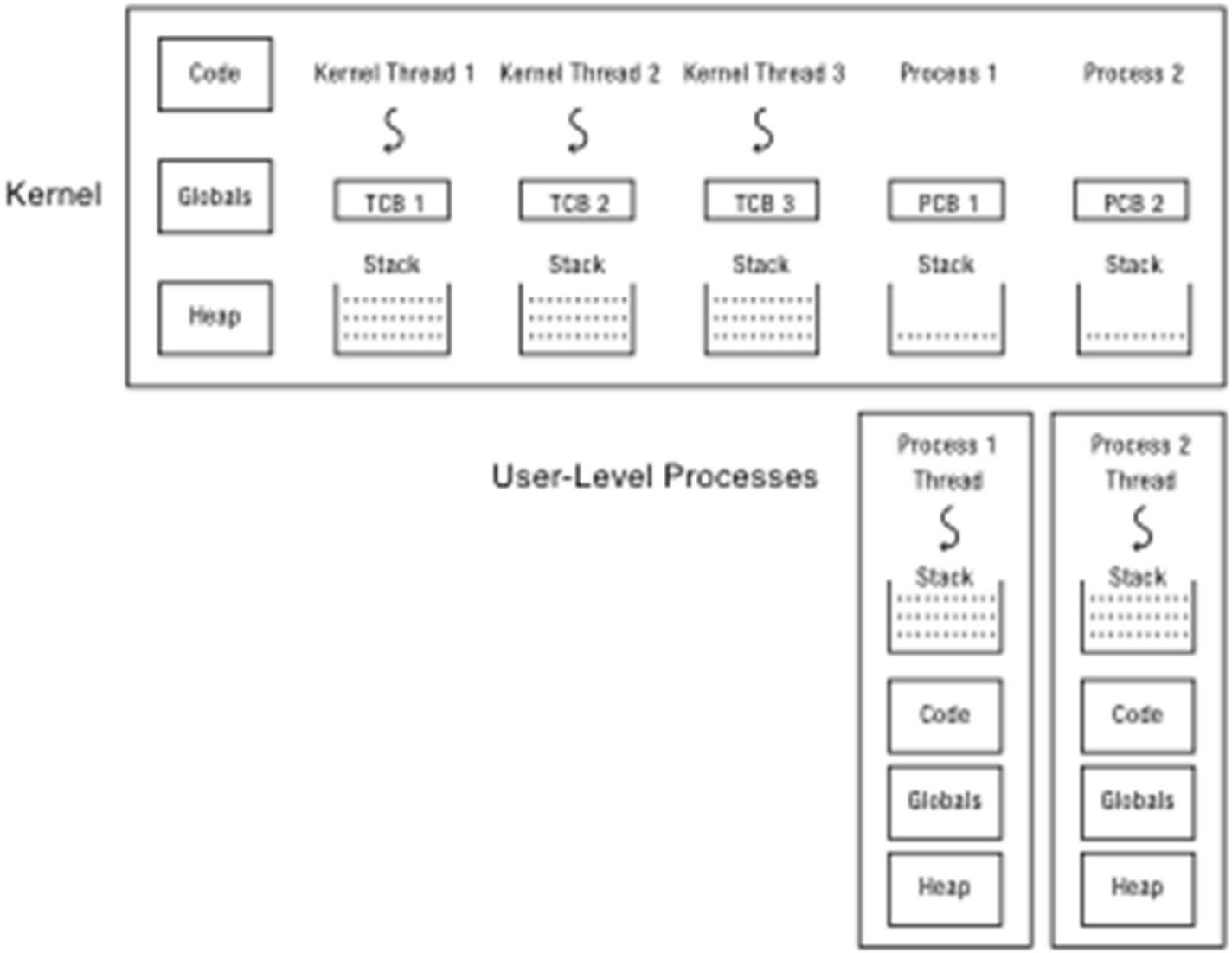


Thread Lifecycle

Implementing Threads: Roadmap

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads (Linux, MacOS)
  - Kernel thread operations available via syscall
- User-level threads
  - Thread operations without system calls

Multithreaded OS Kernel



Implementing threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)
- stub(func, args): OS/161 mips_threadstart
  - Call (*func)(args)
  - If return, call thread_exit()

Thread Stack

- What if a thread puts too many procedures on its stack?
  - What happens in Java?
  - What happens in the Linux kernel?
  - What happens in OS/161?
  - What *should* happen?

Thread Context Switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

OS/161 switchframe_switch

```
/* a0: old thread stack pointer
 * a1: new thread stack pointer */

/* Allocate stack space for 10 registers. */
  addi sp, sp, -40

  /* Save the registers */
  sw   ra, 36(sp)
  sw   gp, 32(sp)
  sw   s8, 28(sp)
  sw   s6, 24(sp)
  sw   s5, 20(sp)
  sw   s4, 16(sp)
  sw   s3, 12(sp)
  sw   s2, 8(sp)
  sw   s1, 4(sp)
  sw   s0, 0(sp)

  /* Store old stack pointer in old thread */
  sw   sp, 0(a0)

 /* Get new stack pointer from new thread */
  lw   sp, 0(a1)
  nop         /* delay slot for load */

/* Now, restore the registers */
  lw   s0, 0(sp)
  lw   s1, 4(sp)
  lw   s2, 8(sp)
  lw   s3, 12(sp)
  lw   s4, 16(sp)
  lw   s5, 20(sp)
  lw   s6, 24(sp)
  lw   s8, 28(sp)
  lw   gp, 32(sp)
  lw   ra, 36(sp)
  nop               /* delay slot for load */

  /* and return. */
  j ra
  addi sp, sp, 40      /* in delay slot */
```

x86 switch_threads

```
# Save caller’s register state
#  NOTE: %eax, etc. are ephemeral
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old
    thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)

# Change stack pointer to new
    thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

A Subtlety

- Thread_create puts new thread on ready list
- When it first runs, some thread calls switchframe
  - Saves old thread state to stack
  - Restores new thread state from stack
- Set up new thread's stack as if it had saved its state in switchframe
  - "returns" to stub at base of stack to run func

Two Threads Call Yield

Thread 1's instructions	Thread 2's instructions	Processor's instructions
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

Involuntary Thread/Process Switch

- Timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (OS/161)
  - End of interrupt handler calls switch()
  - When resumed, return from handler resumes kernel thread or user process
  - Thus, processor context is saved/restored twice (once by interrupt handler, once by thread switch)

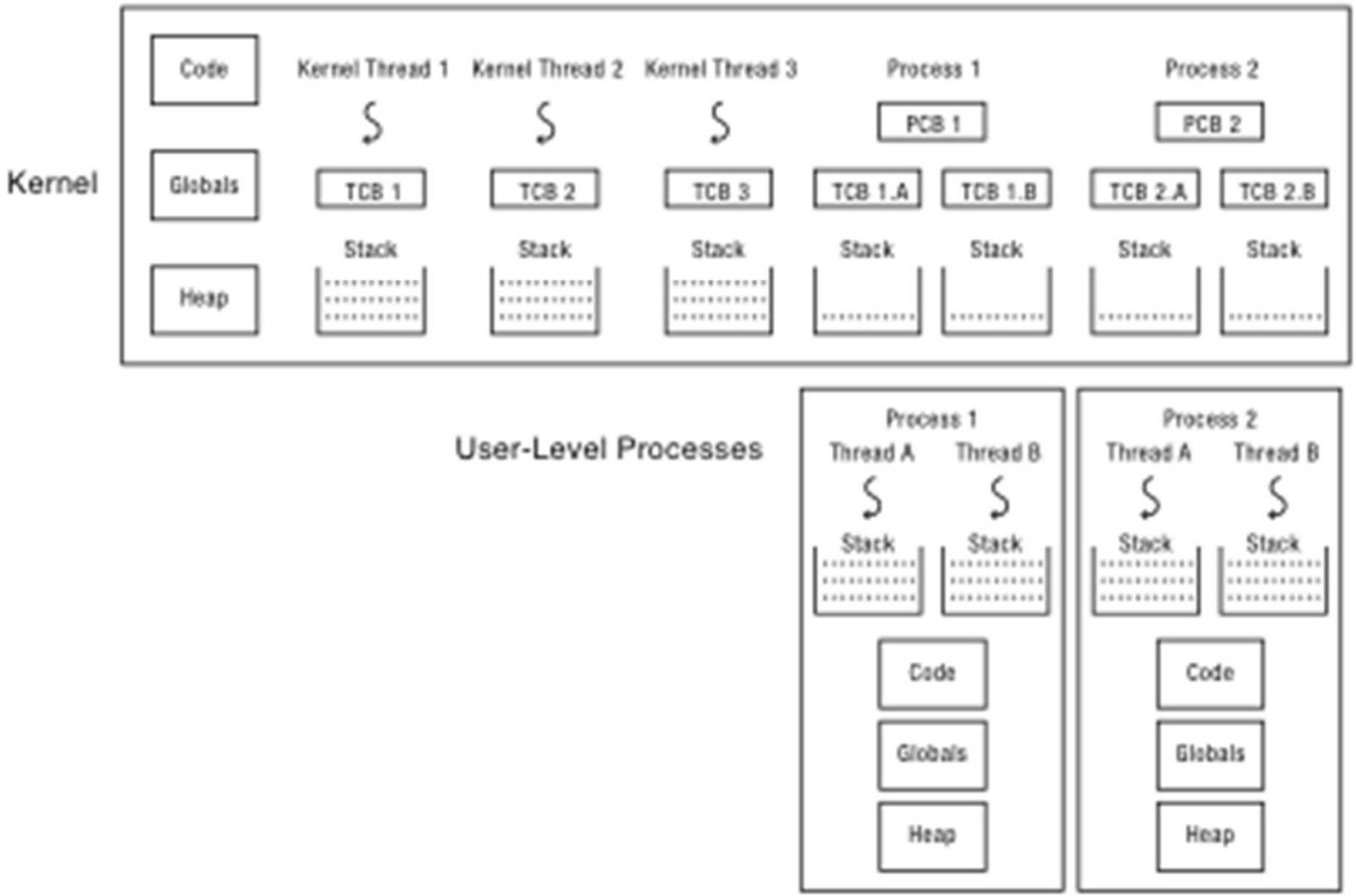
Faster Thread/Process Switch

- What happens on a timer (or other) interrupt?
  - Interrupt handler saves state of interrupted thread
  - Decides to run a new thread
  - Throw away current state of interrupt handler!
  - Instead, set saved stack pointer to trapframe
  - Restore state of new thread
  - On resume, pops trapframe to restore interrupted thread

Multithreaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

Multithreaded User Processes (Take 1)



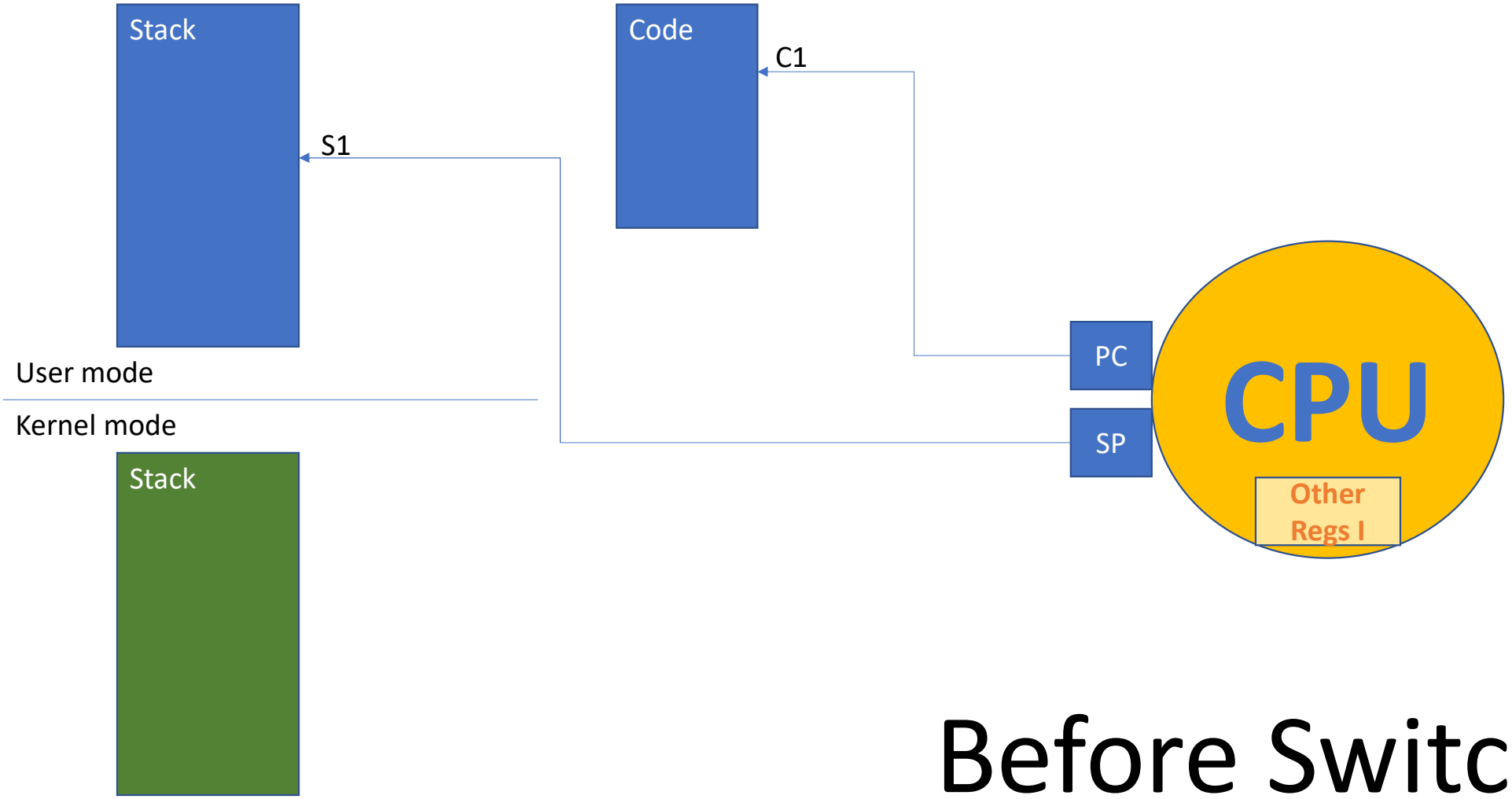
Multithreaded User Processes (Take 2)

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

Multithreaded User Processes (Take 3)

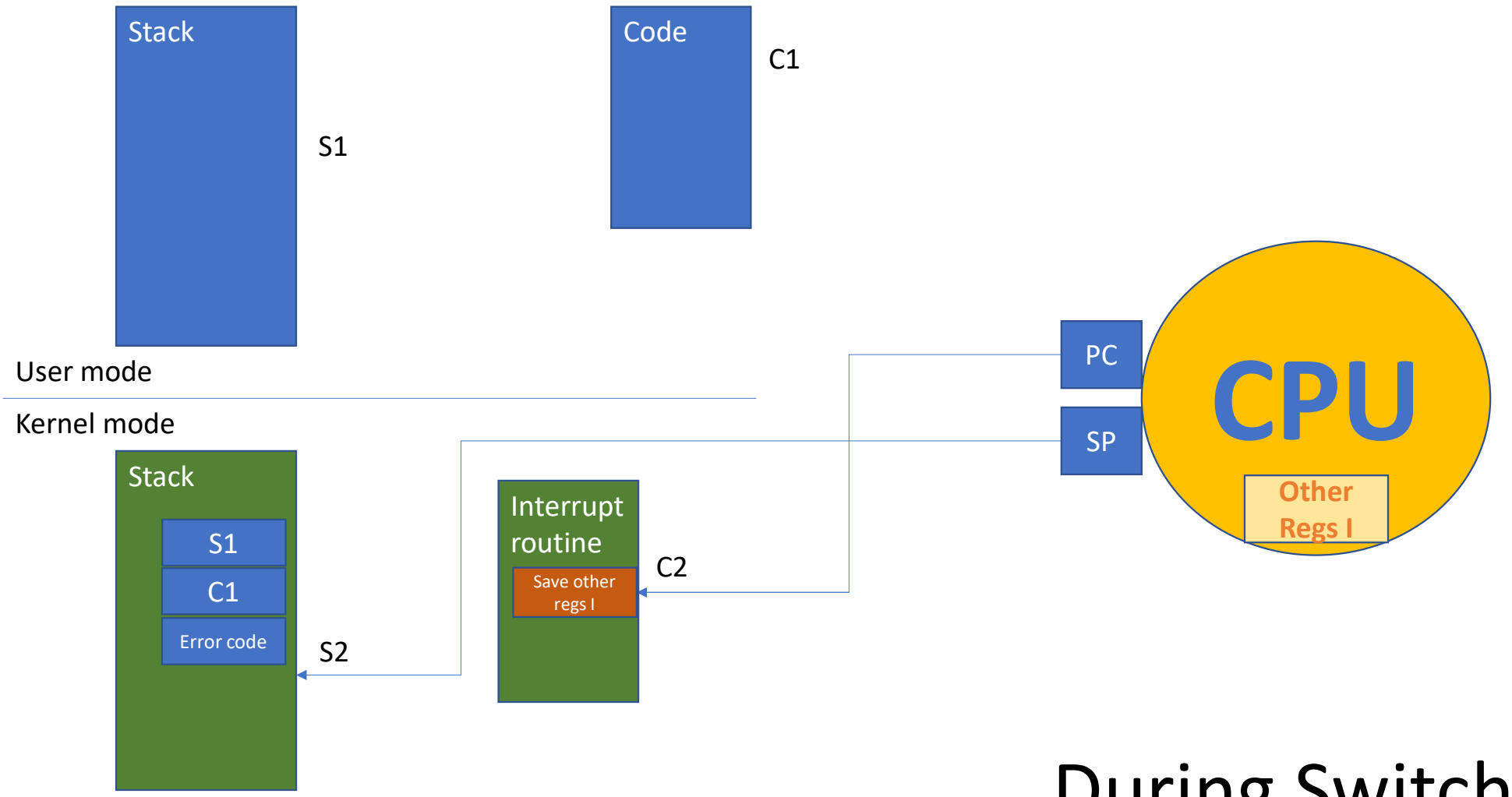
- Scheduler activations (Windows 8)
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
- Upcall whenever kernel needs a user-level scheduling decision
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel

Context Switch Conclusion

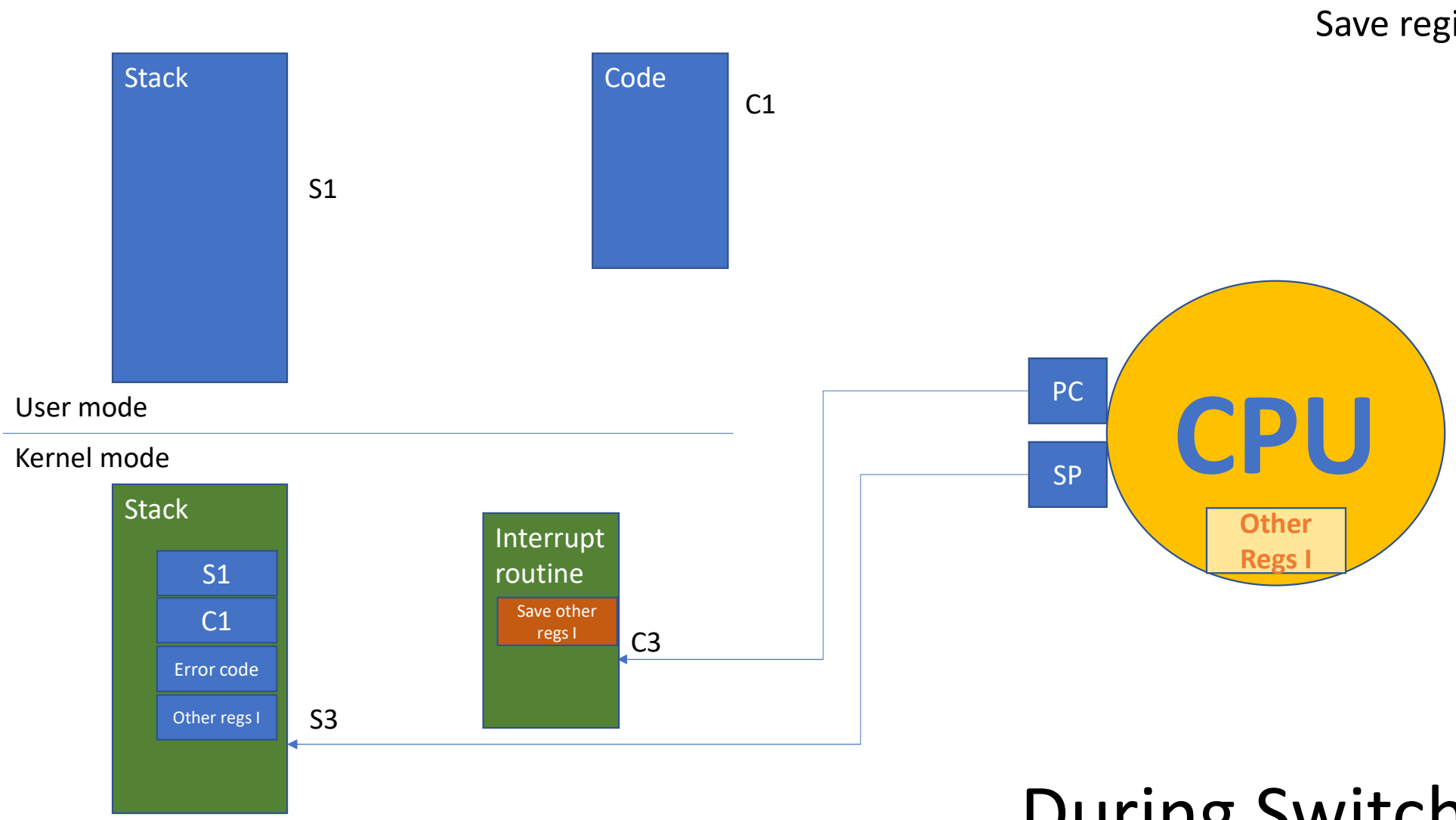


Before Switch

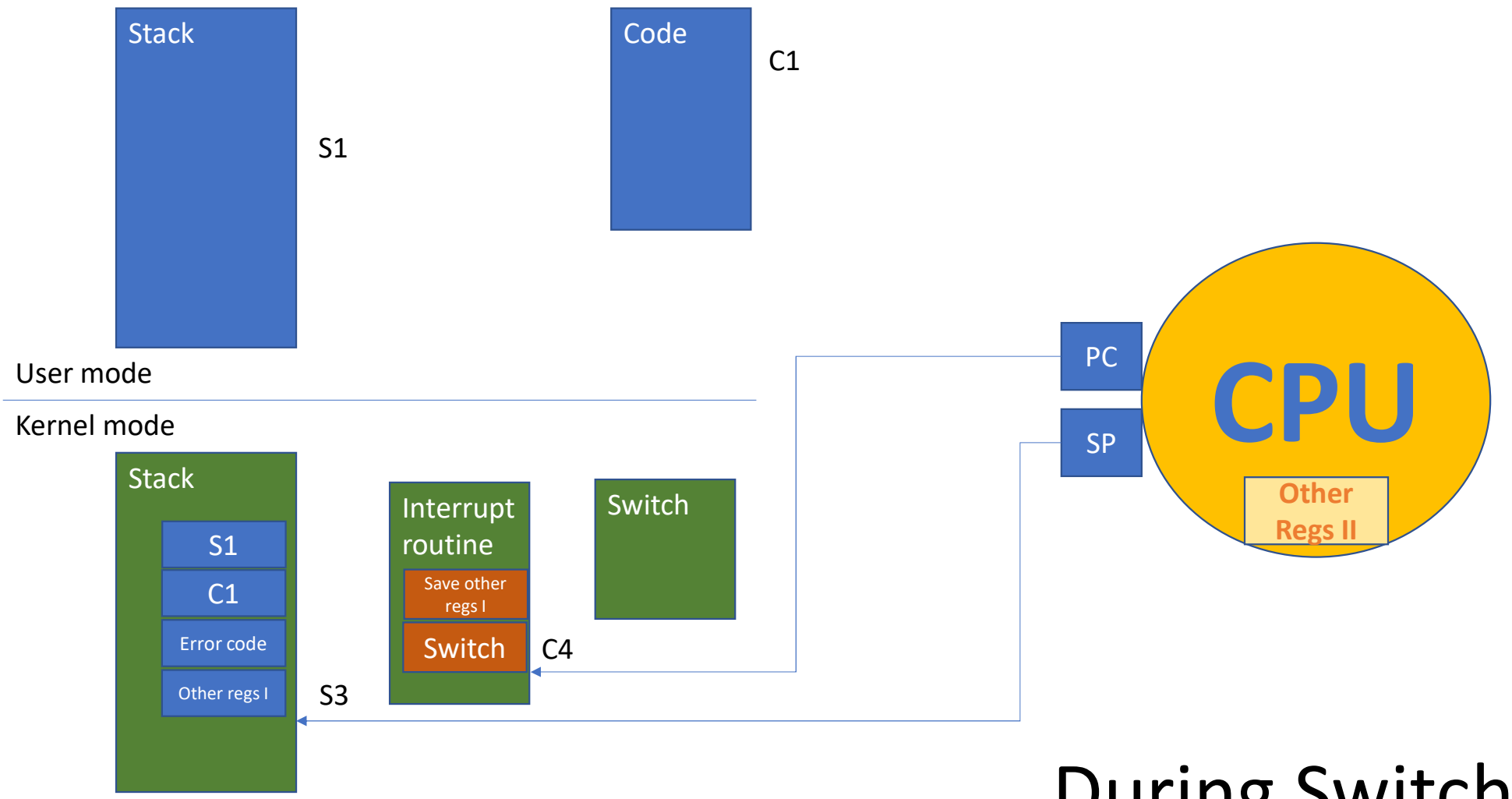
Interrupt/System call



During Switch 1

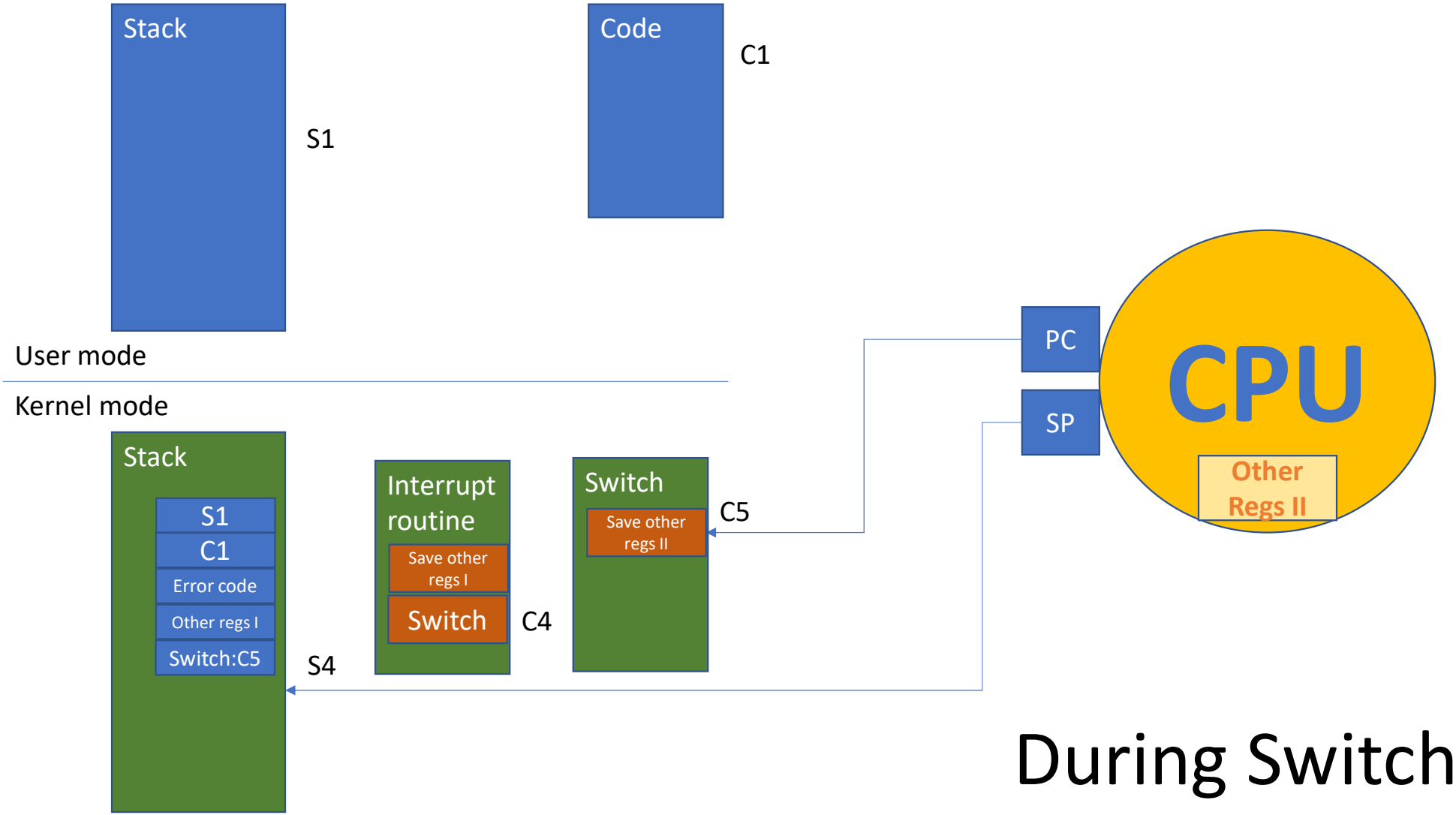

Save registers
Stack
S1
Code
C1
User mode
Kernel mode
PC
SP
CPU
Other Regs I
Stack
S1
C1
Error code
Other regs I
S3
Interrupt routine
Save other regs I
C3
During Switch 2

Call Switch() System call



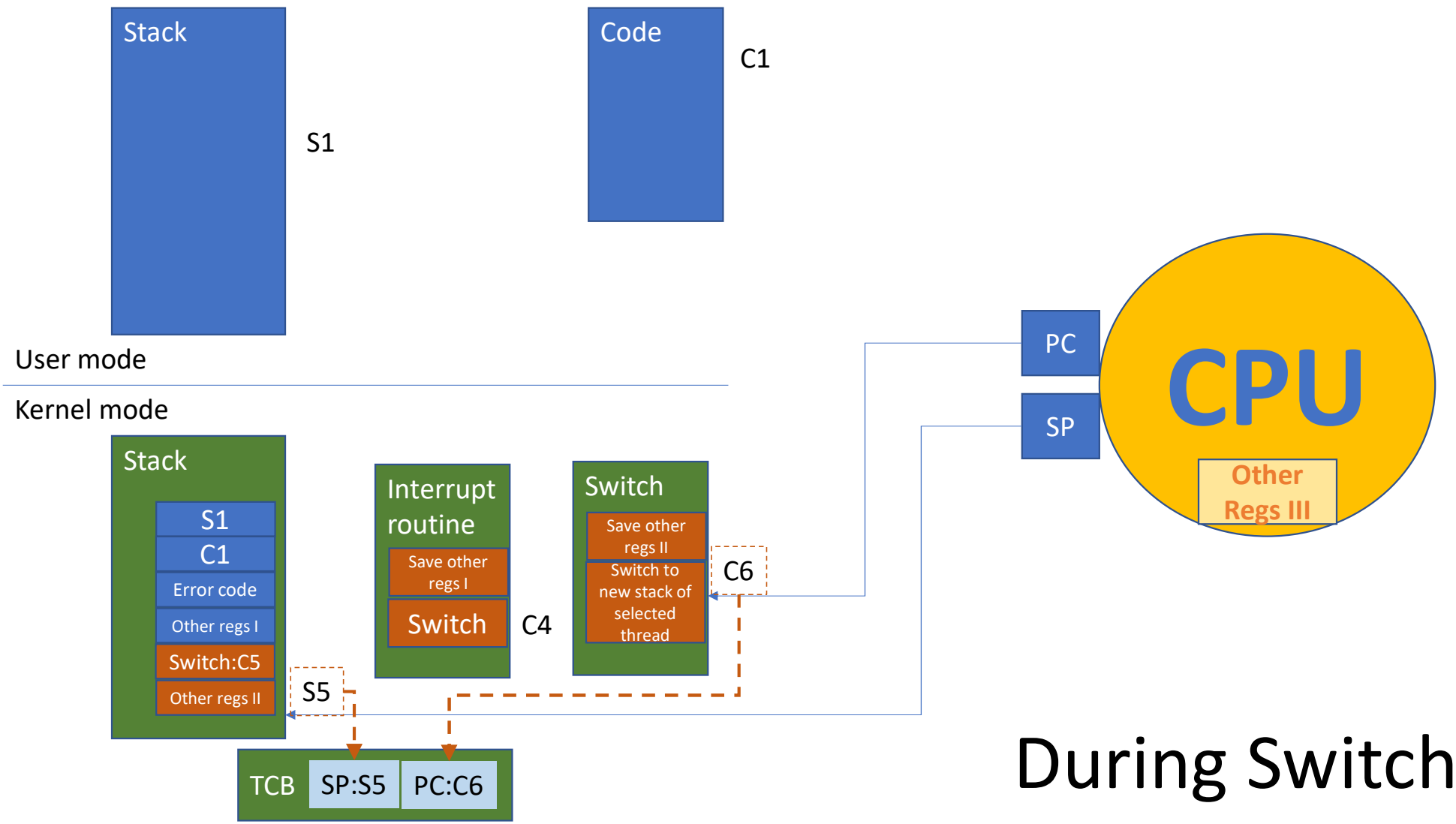
During Switch 3

Save Registers



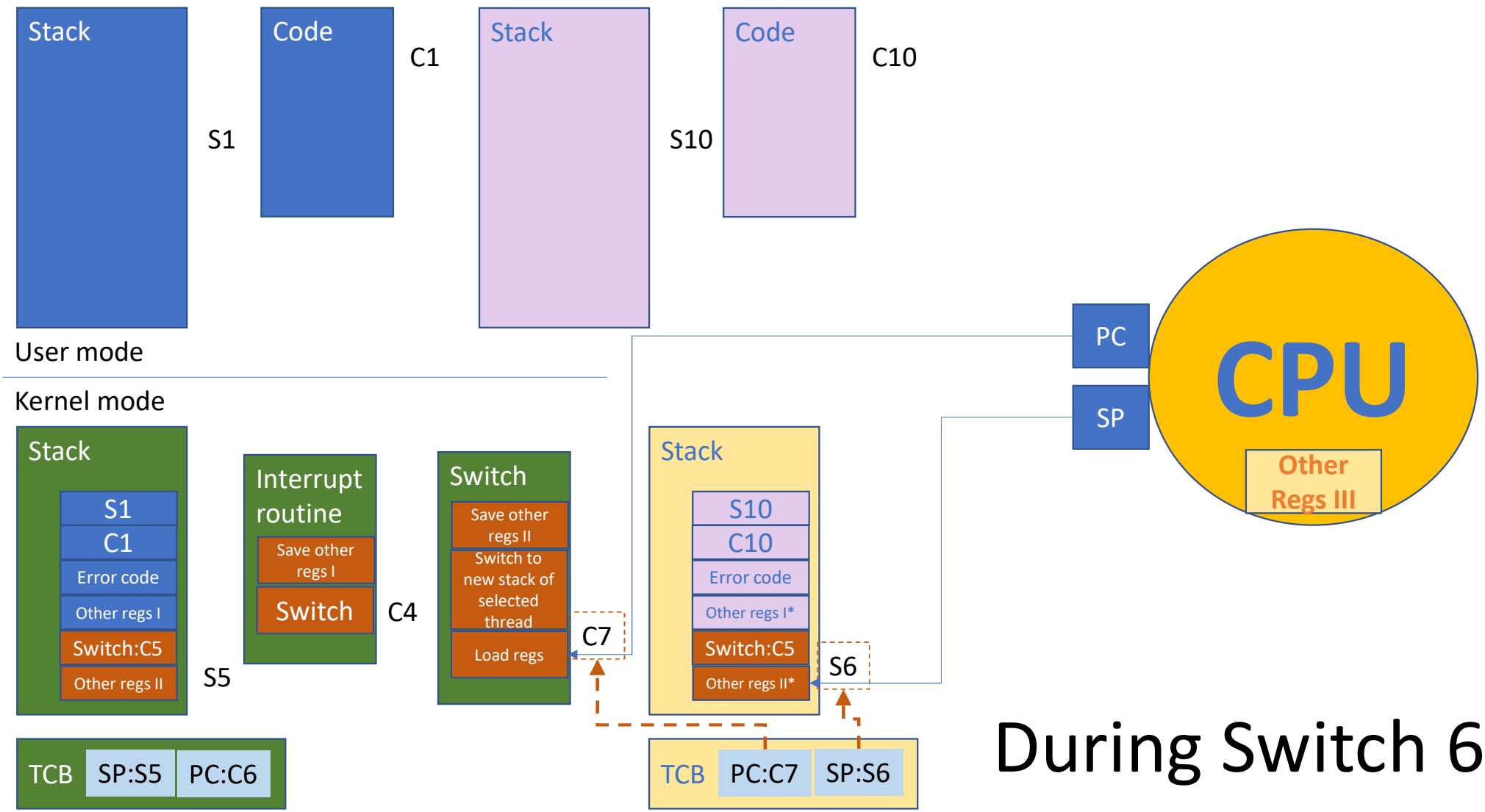
During Switch 4

Switch to new stack

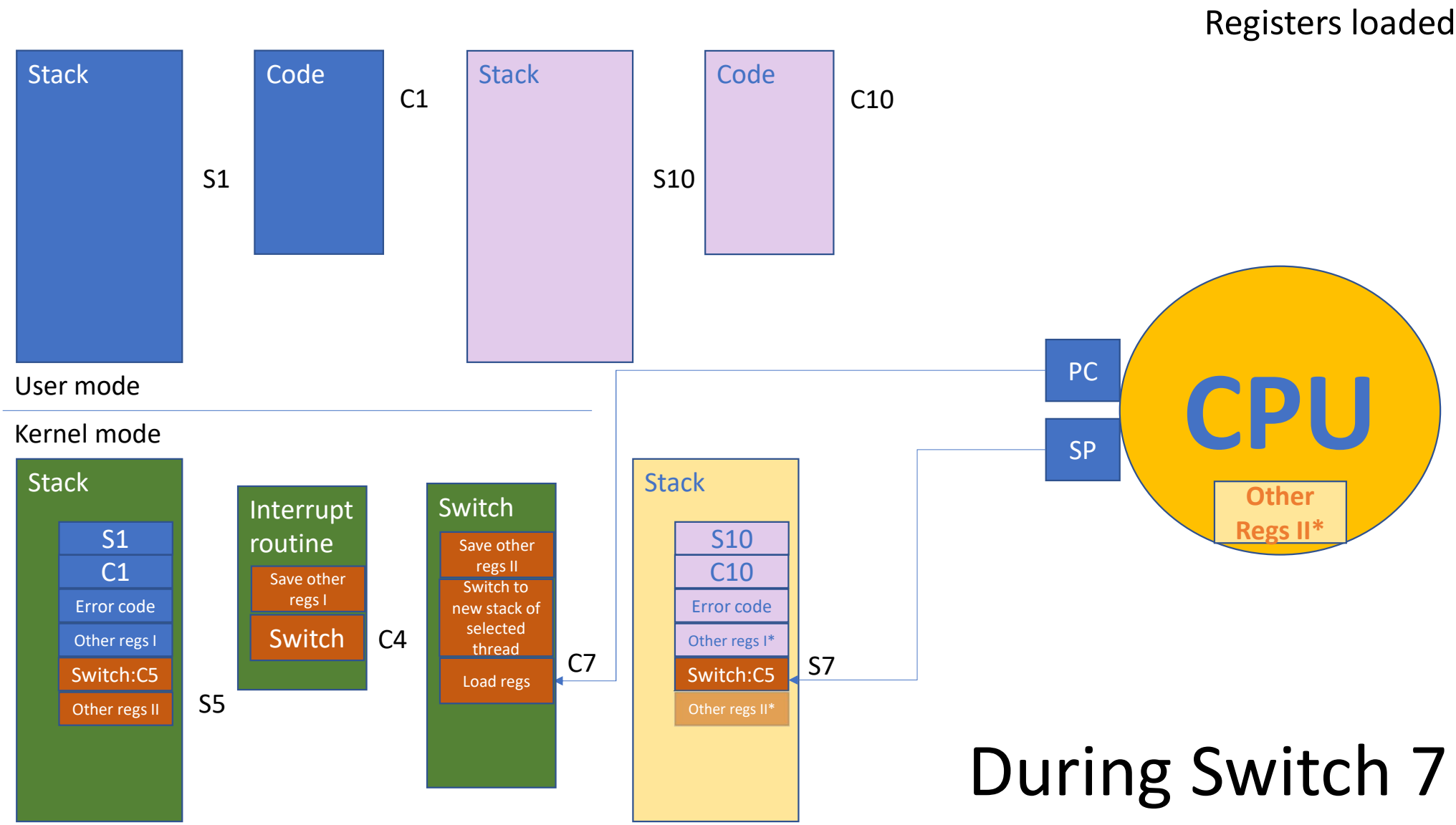


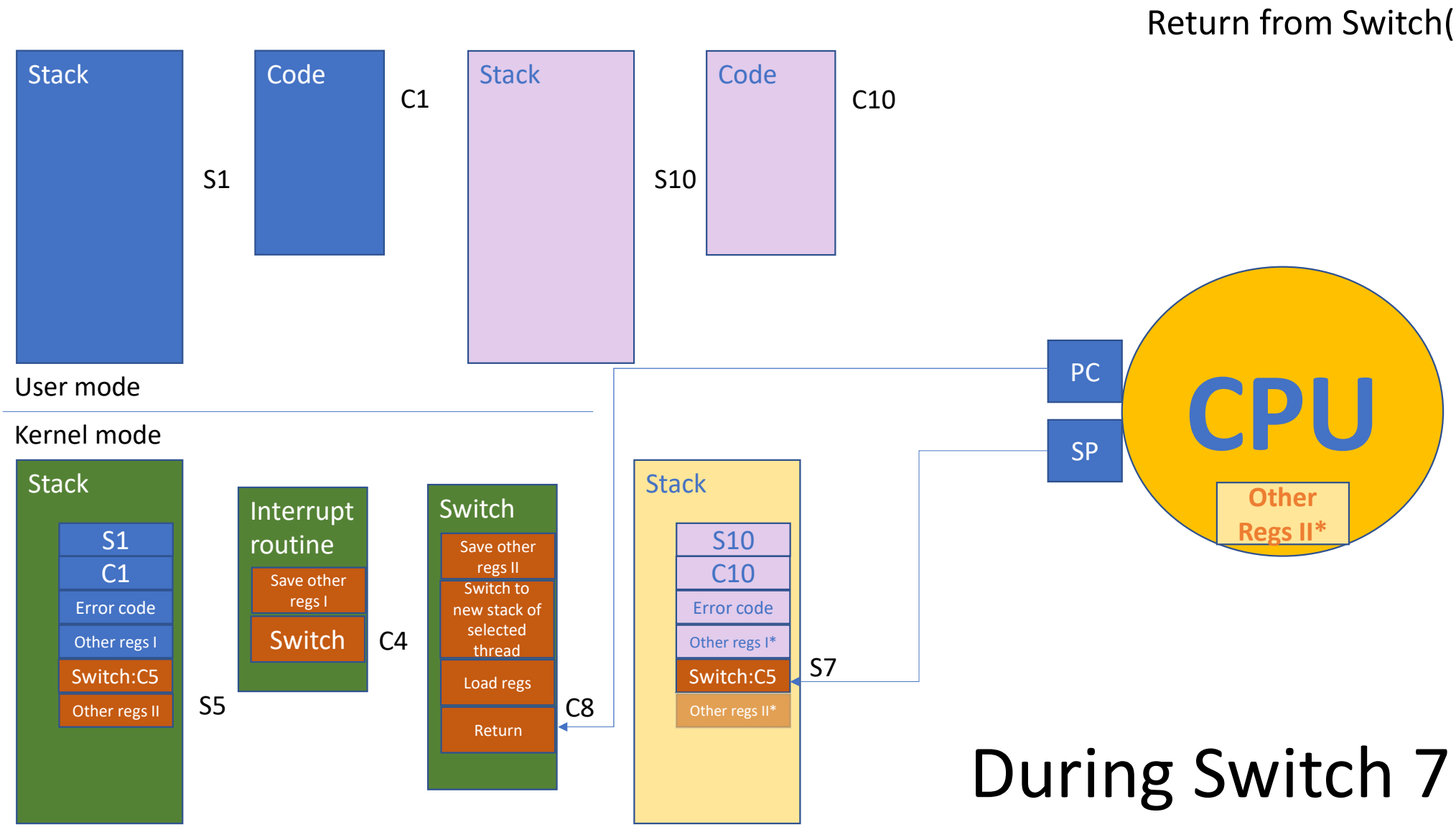
During Switch 5

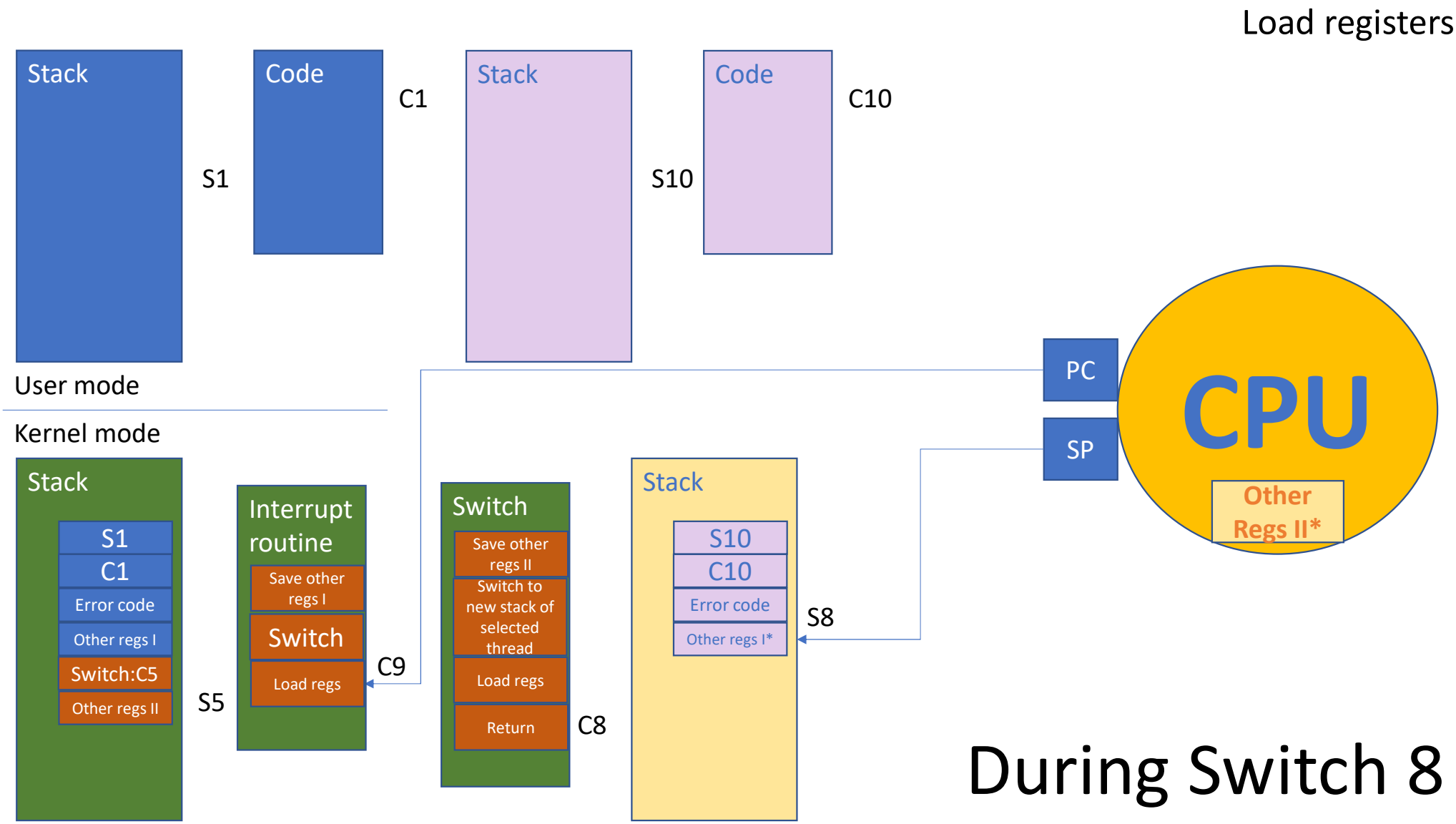
Load regs from new stack

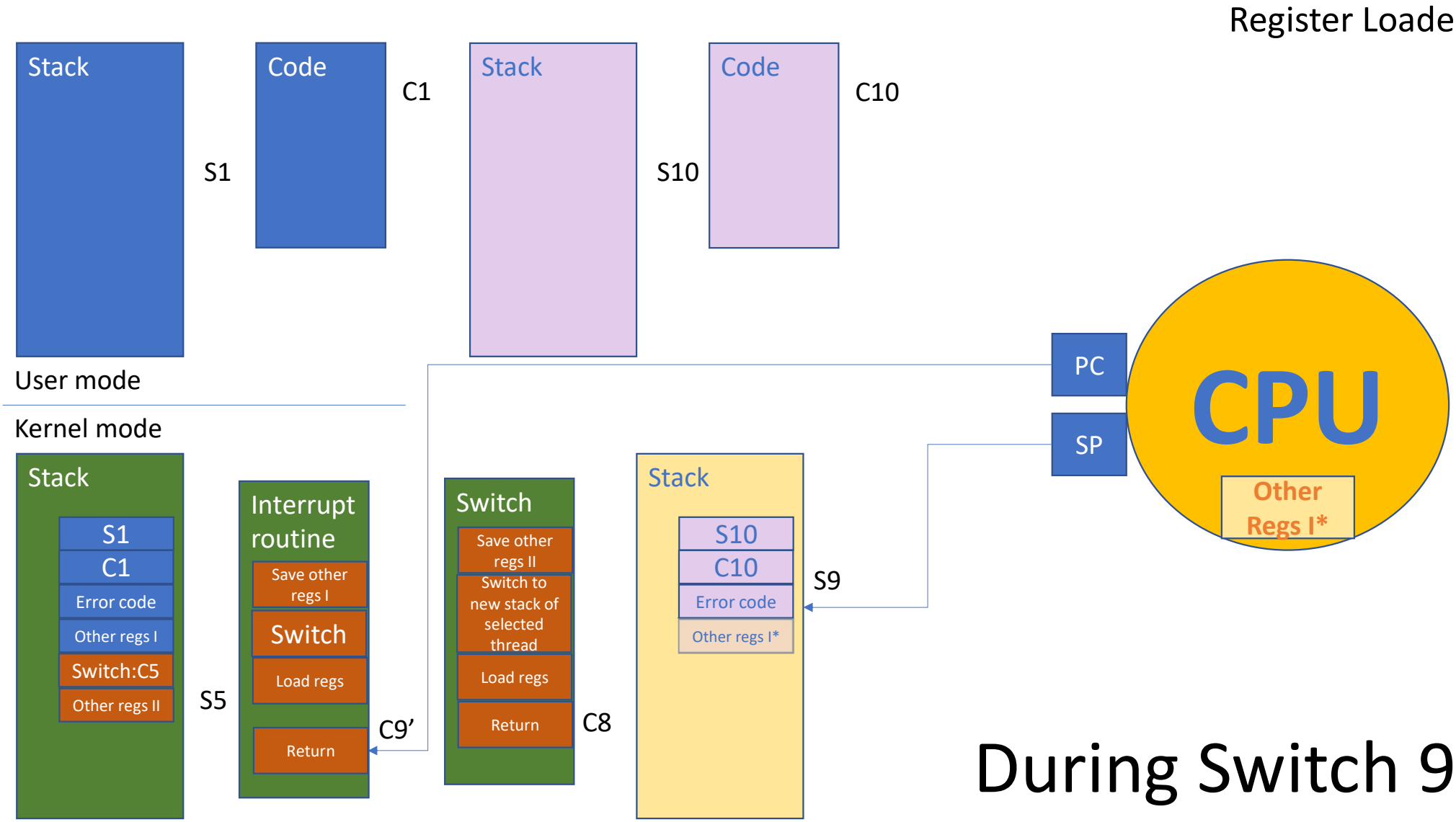


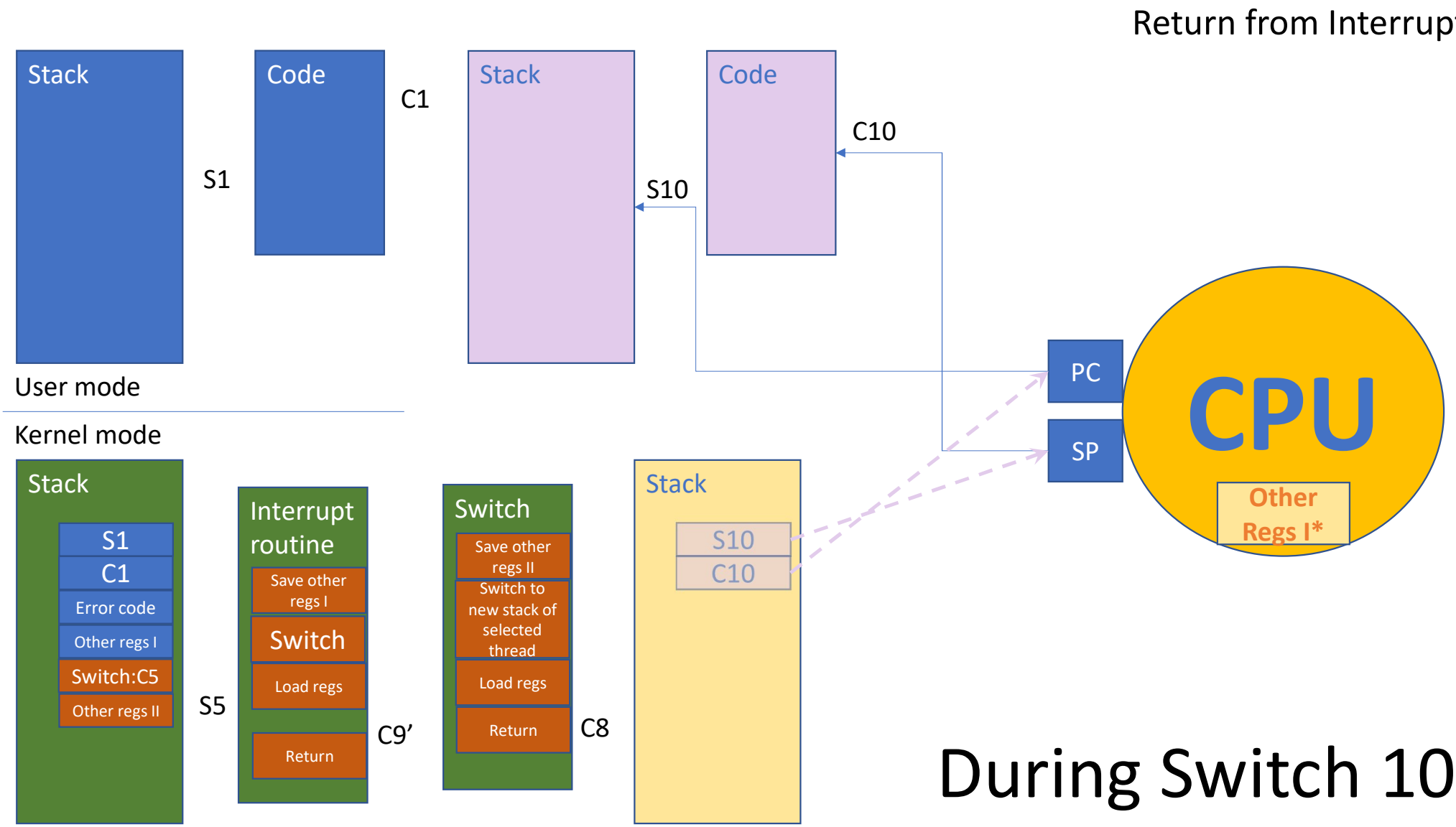
During Switch 6


Registers loaded
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs II*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
C4
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
C7
Stack
S10
C10
Error code
Other regs I*
Switch:C5
Other regs II*
S7
During Switch 7


Return from Switch()
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs II*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
C4
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
Error code
Other regs I*
Switch:C5
Other regs II*
S7
During Switch 7


Load registers
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs II*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
Load regs
C9
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
Error code
Other regs I*
S8
During Switch 8


Register Loaded
Stack
S1
Code
C1
Stack
S10
Code
C10
User mode
Kernel mode
PC
SP
CPU
Other Regs I*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
Load regs
Return
C9'
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
Error code
Other regs I*
S9
During Switch 9


Return from Interrupt
Stack
Code
C1
S1
Stack
Code
C10
S10
User mode
Kernel mode
PC
SP
CPU
Other Regs I*
Stack
S1
C1
Error code
Other regs I
Switch:C5
Other regs II
S5
Interrupt routine
Save other regs I
Switch
Load regs
Return
C9'
Switch
Save other regs II
Switch to new stack of selected thread
Load regs
Return
C8
Stack
S10
C10
During Switch 10

Synchronization

The Dark Side of Concurrency

With interleaved executions, the order in which processes execute at runtime is *nondeterministic*.

- depends on the exact order and timing of process arrivals
- depends on exact timing of asynchronous devices (disk, clock)
- depends on scheduling policies

Some schedule interleavings may lead to incorrect behavior.

- Open the bay doors *before* you release the bomb.
- Two people can't wash dishes in the same sink at the same time.

The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic

Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while (!pInitialized)
    ;
q = someFunction(p);
if (q != someFunction(p))
    panic
```

Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
  - If variables can spontaneously change, most compiler optimizations become impossible
- Why do CPUs reorder instructions?
  - Write buffering: allow next instruction to execute while write is being completed

Fix: **memory barrier**

  - Instruction to compiler/CPU
  - All ops before barrier complete before barrier returns
  - No op after barrier starts until barrier returns

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if (!note)
    if (!milk) {
        leave note
        buy milk
        remove note
    }
```

Too Much Milk, Try #2

Thread A

```
leave note A
if (!note B) {
    if (!milk)
        buy milk
}
remove note A
```

Thread B

```
leave note B
if (!noteA) {
    if (!milk)
        buy milk
}
remove note B
```

Too Much Milk, Try #3

Thread A

```
leave note A
while (note B) // X
    do nothing;
if (!milk)
    buy milk;
remove note A
```

Thread B

```
leave note B
if (!noteA) {   // Y
    if (!milk)
        buy milk
}
remove note B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

Lessons

- Solution is complicated
  - "obvious" code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson's algorithm

Roadmap

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

Locks

- Lock::acquire
  - wait until lock is free, then take it
- Lock::release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
    buy milk
lock.release();
```

Lock Example: Malloc/Free

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}
```

```
void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Will this code work?

```
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = …
    p->field2 = …
    return p;
}
```

Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Condition Variable Design Pattern

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
     }

    // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state

    // If testSharedState is now true
    cv.signal(&lock);


    // Read/write shared state
    lock.release();
}
```

C#

- Lock

```
static object _Lock = new object();

lock (_Lock)
{
   …
}
```

- Condition Variable

```
lock (_Lock)
{
   …
   Monitor.Wait(_Lock);
   or
   Monitor.Pulse(_Lock);
   Monitor.PulseALL(_Lock);
   …
}
```

Synchronization

Part II

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

Homework Solution: Bounded Buffer

```
get() {
  lock(_key)
  {
     while (front == tail) {
        monitor.wait(_key);
     }
     item = buf[front % MAX];
     front++;
     monitor.signal(_key);
   }
    return item;
}
```

```
put(item) {
   lock.(_key);
   {
      while ((tail – front) == MAX) {
         monitor.wait(_key);
      }
      buf[tail % MAX] = item;
      tail++;
      monitor.signal(_key);
    }
}
```

Initially: front = tail = 0; MAX is buffer capacity

empty/full are condition variables

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= tail
  - front + MAX >= tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness

Pre/Post Conditions

```
methodThatWaits() {
   lock.acquire();
   // Pre-condition: State is consistent

   // Read/write shared state

   while (!testSharedState()) {
       cv.wait(&lock);
    }
   // WARNING: shared state may
   // have changed!  But
  // testSharedState is TRUE
  // and pre-condition is true

  // Read/write shared state
   lock.release();
}
```

```
methodThatSignals() {
   lock.acquire();
   // Pre-condition: State is consistent

   // Read/write shared state

   // If testSharedState is now true
   cv.signal(&lock);

   // NO WARNING: signal keeps lock

   // Read/write shared state
   lock.release();
}
```

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait()) {
    condition.Wait(lock);
}
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - Overhead of creating threads, if not needed
  - Lock contention: only one thread at a time can hold a given lock
  - Shared data protected by a lock may ping back and forth between cores
  - False sharing: communication between cores even for data that is not shared

Reducing Lock Contention

- Fine-grained locking
  - Partition object into subsets, each protected by its own lock
  - Example: hash table buckets
- Per-processor data structures
  - Partition object so that most/all accesses are made by one processor
  - Example: per-processor heap
- Ownership/Staged architecture
  - Only one thread at a time accesses shared data
  - Example: pipeline of threads

Synchronization without Lock

Communicating Sequential Processes (CSP/Google Go)

- A thread per shared object
  - Only thread allowed to touch object's data
  - To call a method on the object, send thread a message with method name, arguments
  - Thread waits in a loop, get msg, do operation
- No memory races!

Locks/CVs vs. CSP

- Create a lock on shared data

  = create a single thread to operate on data
- Call a method on a shared object

  = send a message/wait for reply
- Wait for a condition

  = queue an operation that can’t be completed just yet
- Signal a condition

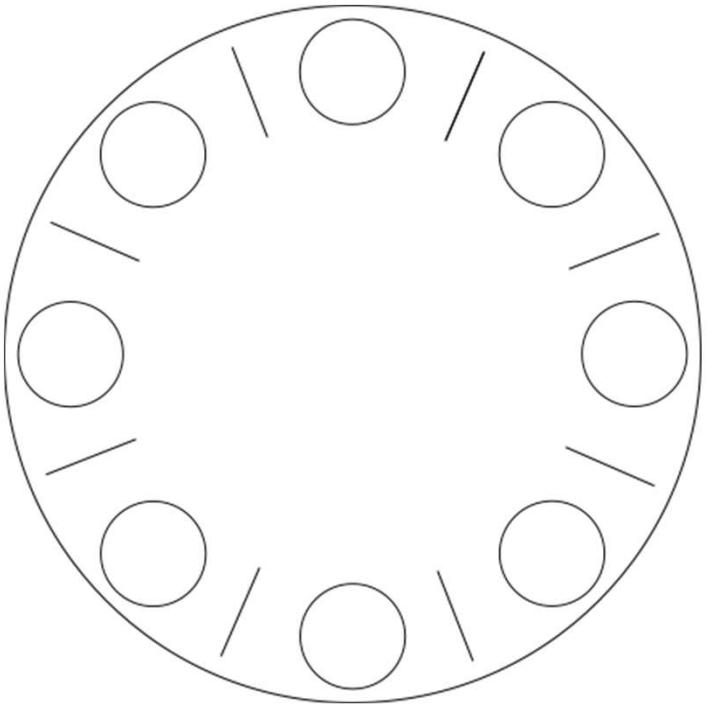
  = perform a queued operation, now enabled

Multi-Object Synchronization

Multi-Object Programs

- What happens when we try to synchronize across multiple objects in a large program?
  - Each object with its own lock, condition variables
- Performance
- Semantics/correctness
- Deadlock
- Eliminating locks

Dining Lawyers



Each lawyer needs two chopsticks to eat.
Each grabs chopstick on the right first.

Deadlock Definition

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

Example: two locks

Thread A

```
lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();
```

Thread B

```
lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```

Two locks and a condition variable

Thread A

```
lock1.acquire();
…
lock2.acquire();
while (need to wait) {
    condition.wait(lock2);
}
lock2.release();
…
lock1.release();
```

Thread B

```
lock1.acquire();
…
lock2.acquire();
…
condition.signal(lock2);
…
lock2.release();
…
lock1.release();
```

Necessary Conditions for Deadlock

- Limited access to resources
  - If infinite resources, no deadlock!
- No preemption
  - If resources are virtual, can break deadlock
- Multiple independent requests
  - “wait while holding”
- Circular chain of requests

Preventing Deadlock

- Exploit or limit program behavior
  - Limit program from doing anything that might lead to deadlock
- Predict the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior

- Provide enough resources
  - How many chopsticks are enough?
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup
- Eliminate circular waiting
  - Lock ordering: always acquire locks in a fixed order
  - Example: move file from one directory to another

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

Example:

```
semaphore  smp = new semaphore;
smp.v();                    smp's value = 1

smp.p();                    try to reduce smp by 1 (acquire lock)
ShR = 100;                  Working with shared resource(s)
smp.v();                    try to increse smp by 1 (release lock)
```

Scheduling

Main Points

- Scheduling policy: what to do next, when there are multiple threads ready to run
  - Or multiple packets to send, or web requests to serve, or …
- Definitions
  - response time, throughput, predictability
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling
- Queueing theory
  - Can you predict/improve a system's response time?

Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, …
- Latency/response time
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?

More Definitions

- Workload
  - Set of tasks for system to perform
- Preemptive scheduler
  - If we can take resources away from a running task
- Work-conserving
  - Resource is used whenever there is a task to run
- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered

First In First Out (FIFO)

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: memcached
  - Facebook cache of friend lists, …

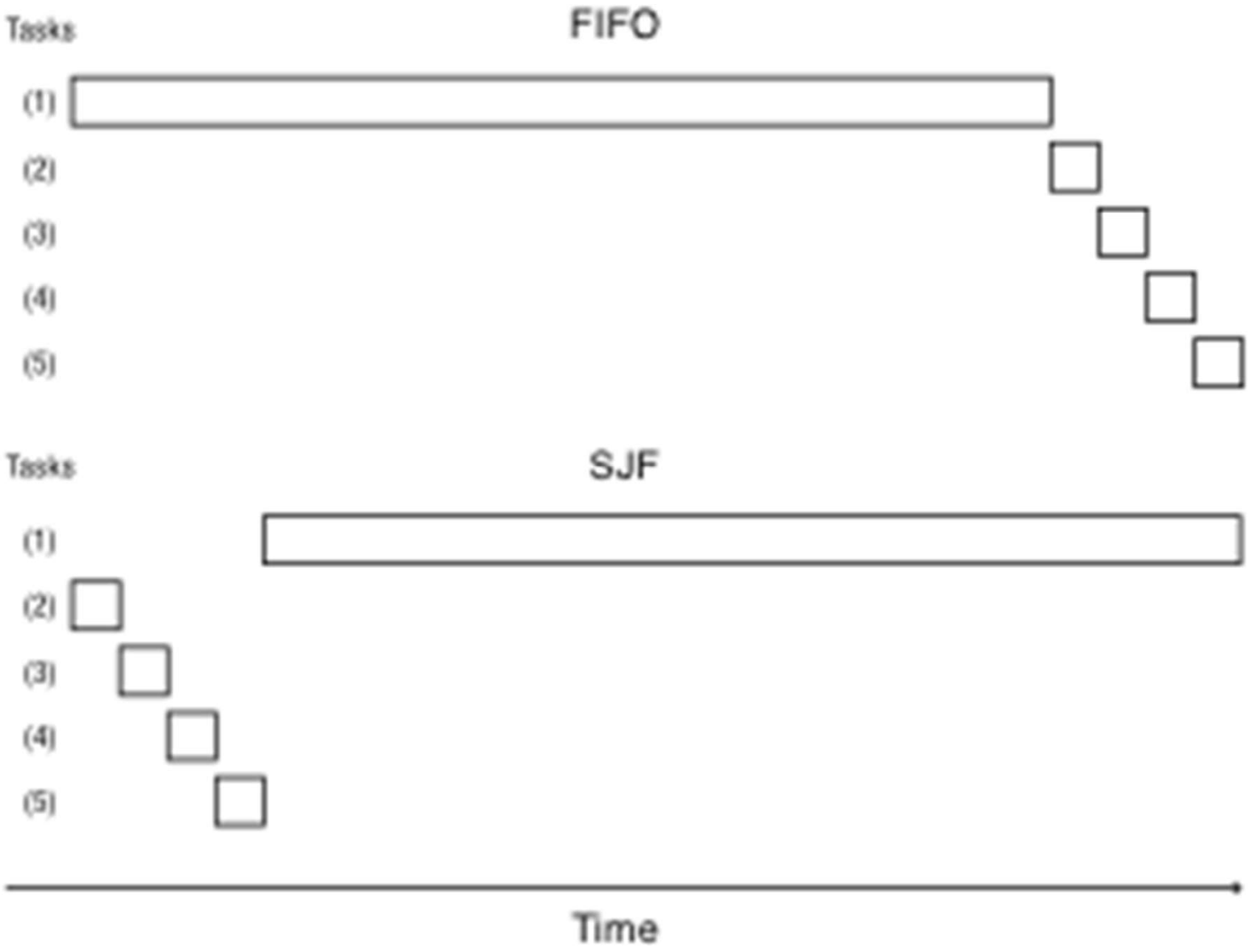
- On what workloads is FIFO particularly bad?

Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

FIFO vs. SJF



Question

- Claim: SJF is optimal for average response time
  - Why?

- Does SJF have any downsides?

Question

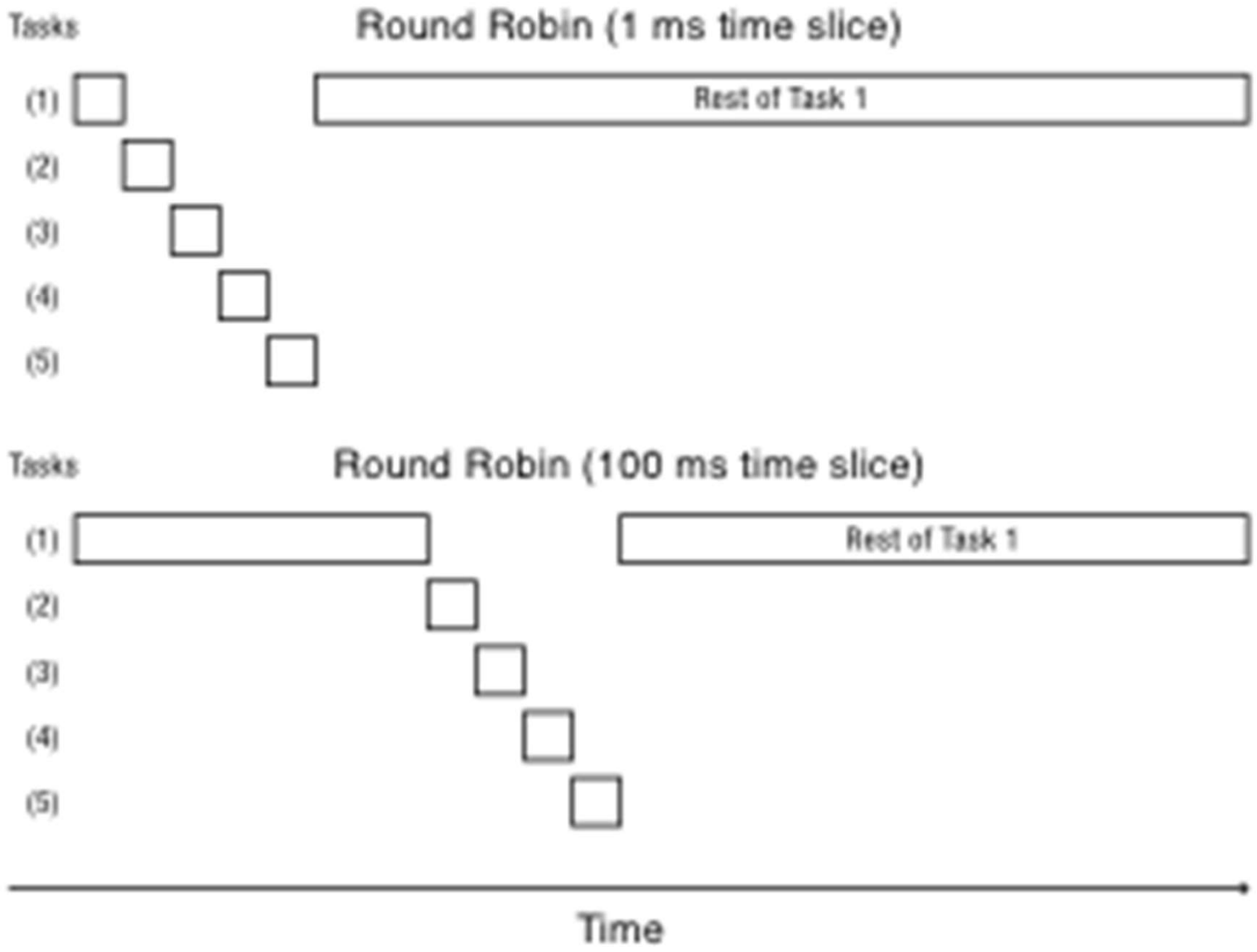
- Is FIFO ever optimal?

- Pessimal?

Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

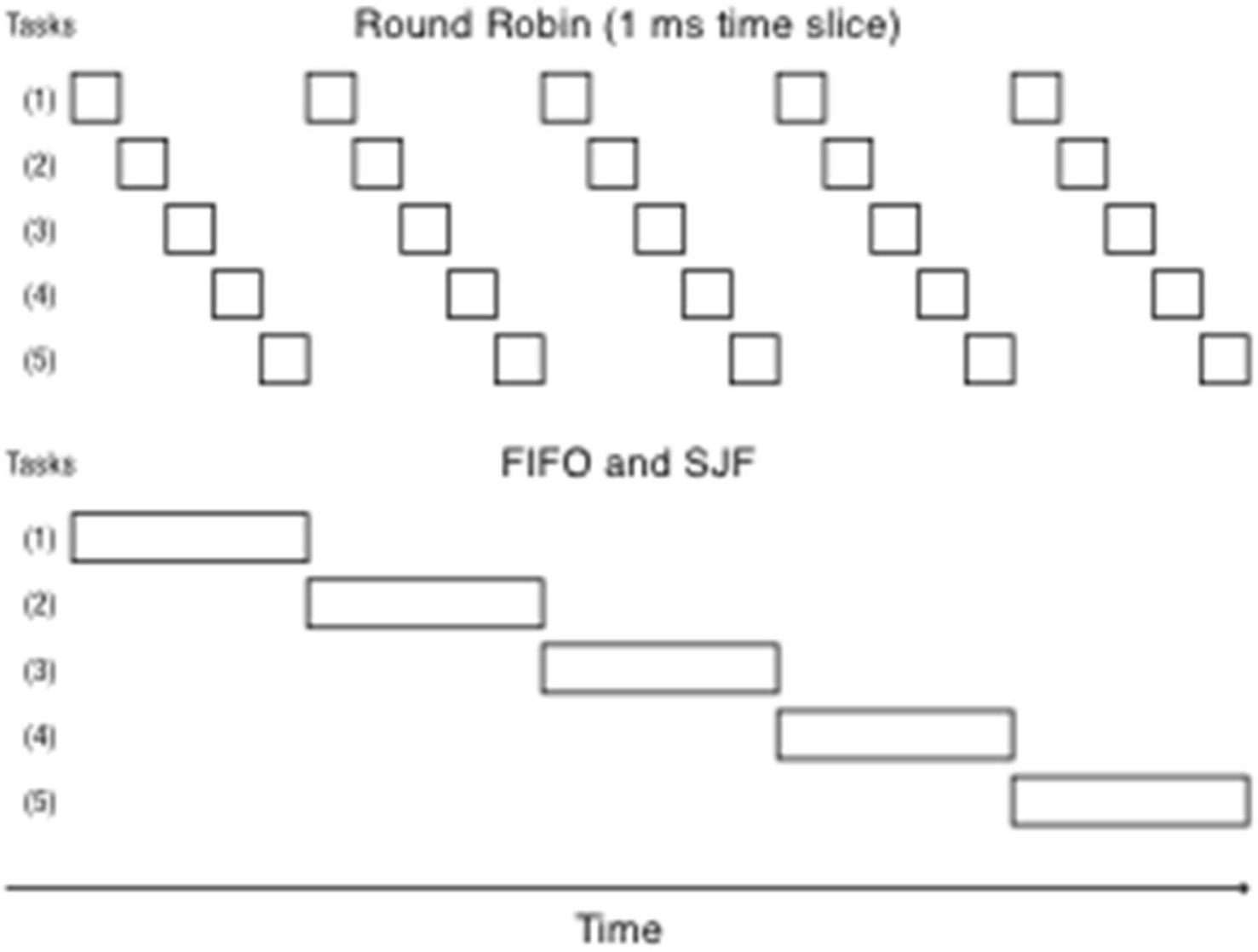
Round Robin


Tasks
Round Robin (1 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Tasks
Round Robin (100 ms time slice)
(1)
Rest of Task 1
(2)
(3)
(4)
(5)
Time

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?

Round Robin vs. FIFO

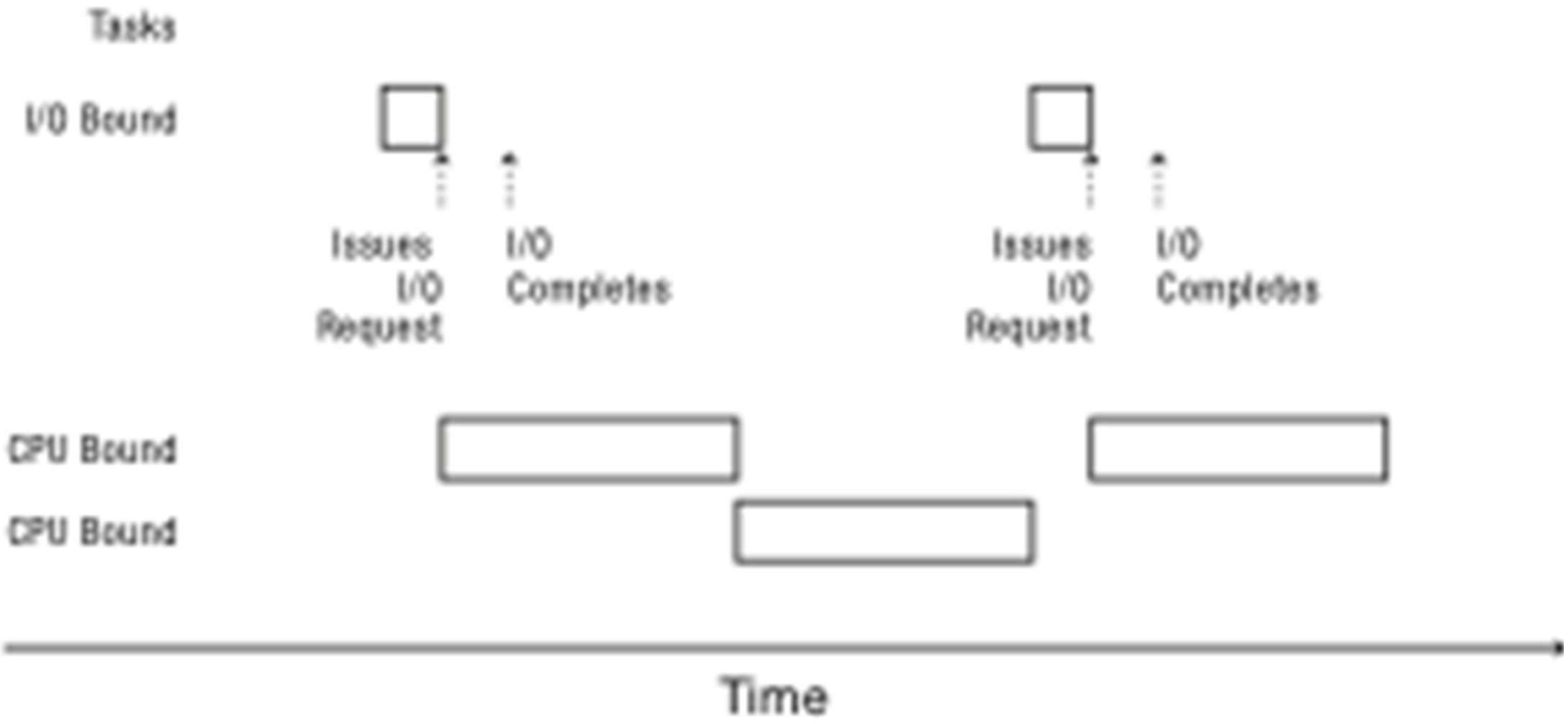


Round Robin = Fairness?

- Is Round Robin always fair?

- What is fair?
  - FIFO?
  - Equal share of the CPU?
  - What if some tasks don't need their full share?
  - Minimize worst case divergence?
    - Time task would take if no one else was running
    - Time task takes under scheduling algorithm

Mixed Workload



Max-Min Fairness

- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned
- One approach: maximize the minimum allocation given to a task
  - If any task needs less than an equal share, schedule the smallest of these first
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

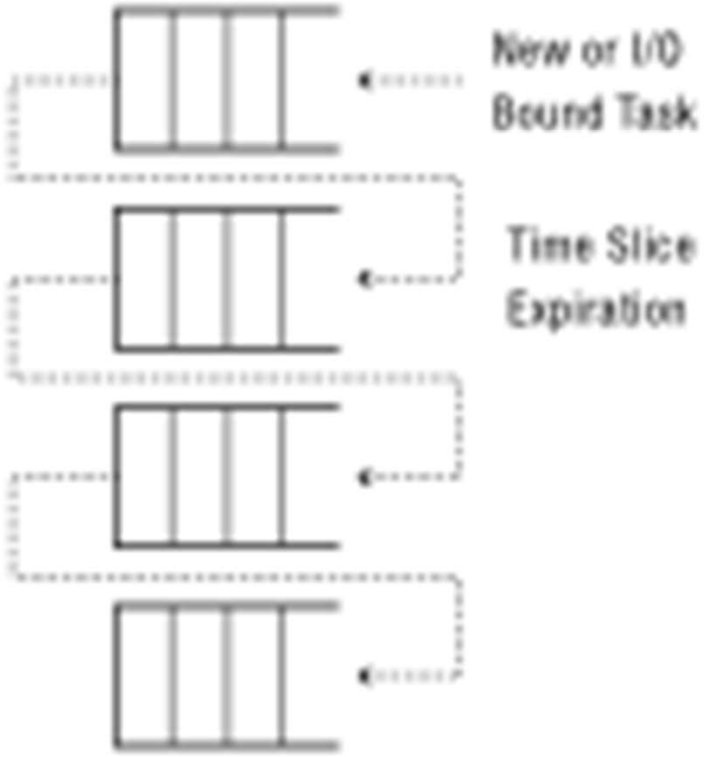
MFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

MFQ

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



Uniprocessor Summary (1)

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Uniprocessor Summary (2)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.

Uniprocessor Summary (3)

- Max-Min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

Question

- If we want to use MFQ on a multiprocessor system
  - Is there any problem?
    - If yes, list the primary problems and their causes
    - Suggest methods to solve or minimize these problems

Address Translation

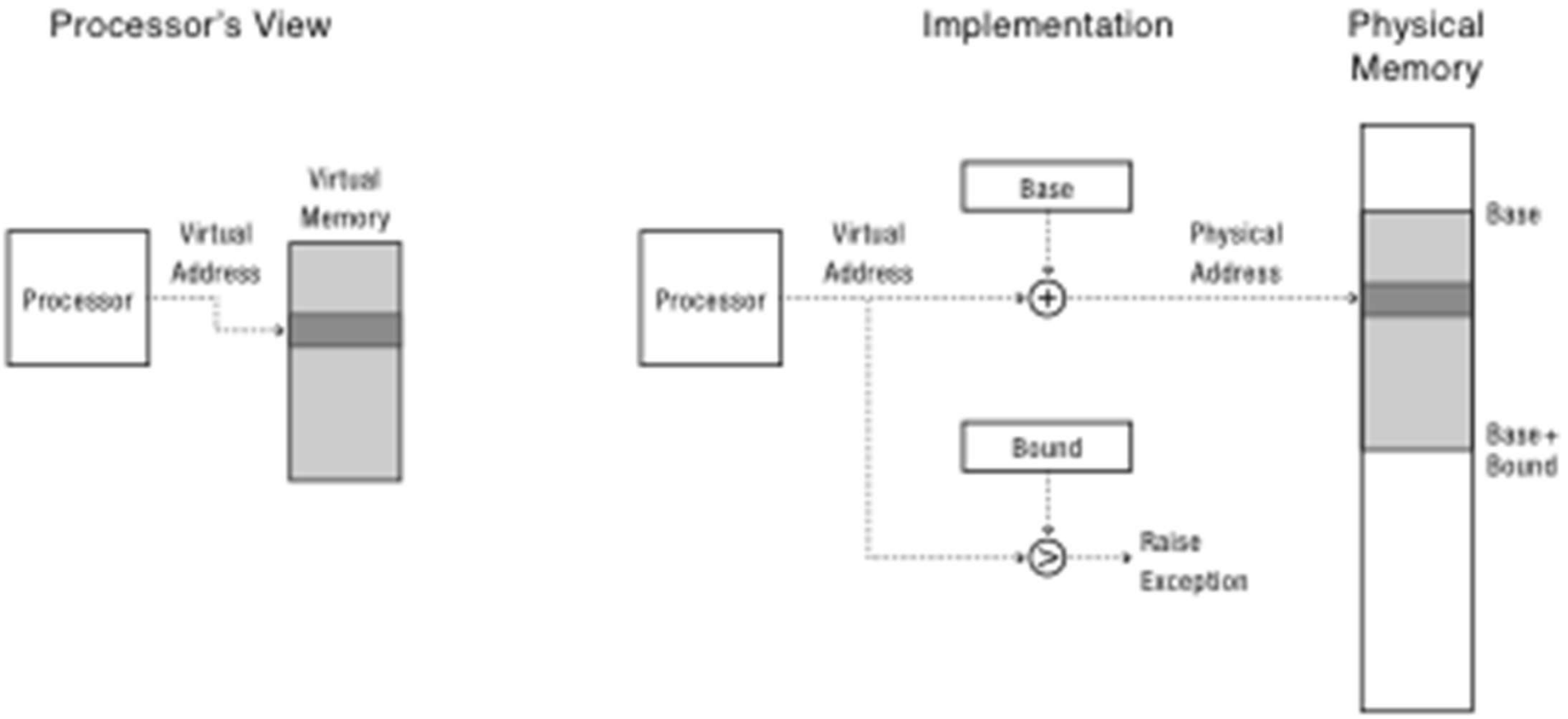
Main Points

- Address Translation Concept
  - How do we convert a virtual address to a physical address?
- Flexible Address Translation
  - Base and bound
  - Segmentation
  - Paging
  - Multilevel translation
- Efficient Address Translation
  - Translation Lookaside Buffers
  - Virtually and physically addressed caches

Address Translation Goals

- Memory protection
- Memory sharing
  - Shared libraries, interprocess communication
- Sparse addresses
  - Multiple regions of dynamic allocation (heaps/stacks)
- Efficiency
  - Memory placement
  - Runtime lookup
  - Compact translation tables

Virtually Addressed Base and Bounds



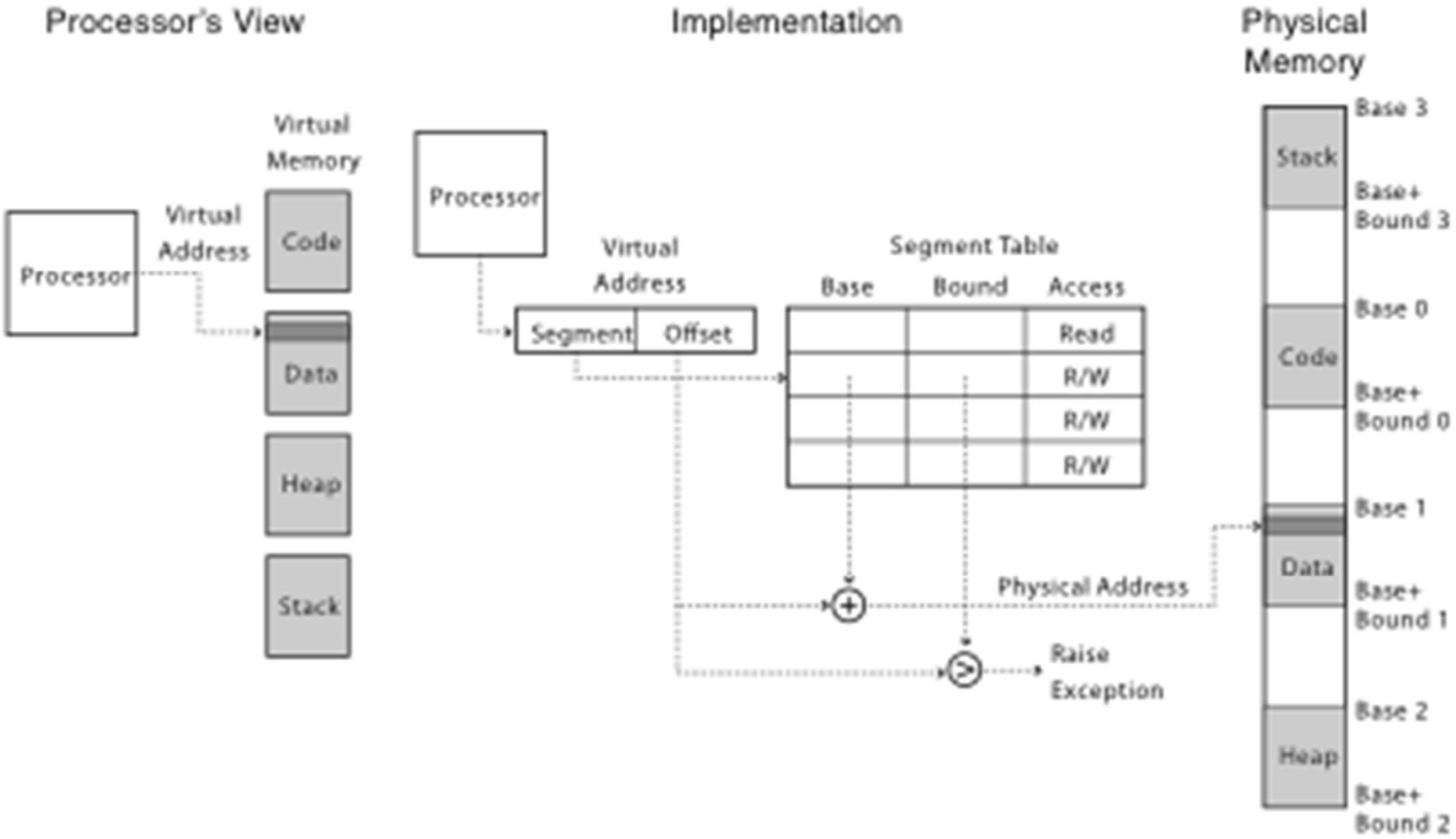
Virtually Addressed Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Safe
  - Can relocate in physical memory without changing process
- Cons?
  - Can’t keep program from accidentally overwriting its own code
  - Can’t share code/data with other processes
  - Can’t grow stack/heap as needed

Segmentation

- Segment is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions

Segmentation



2 bit segment #
12 bit offset

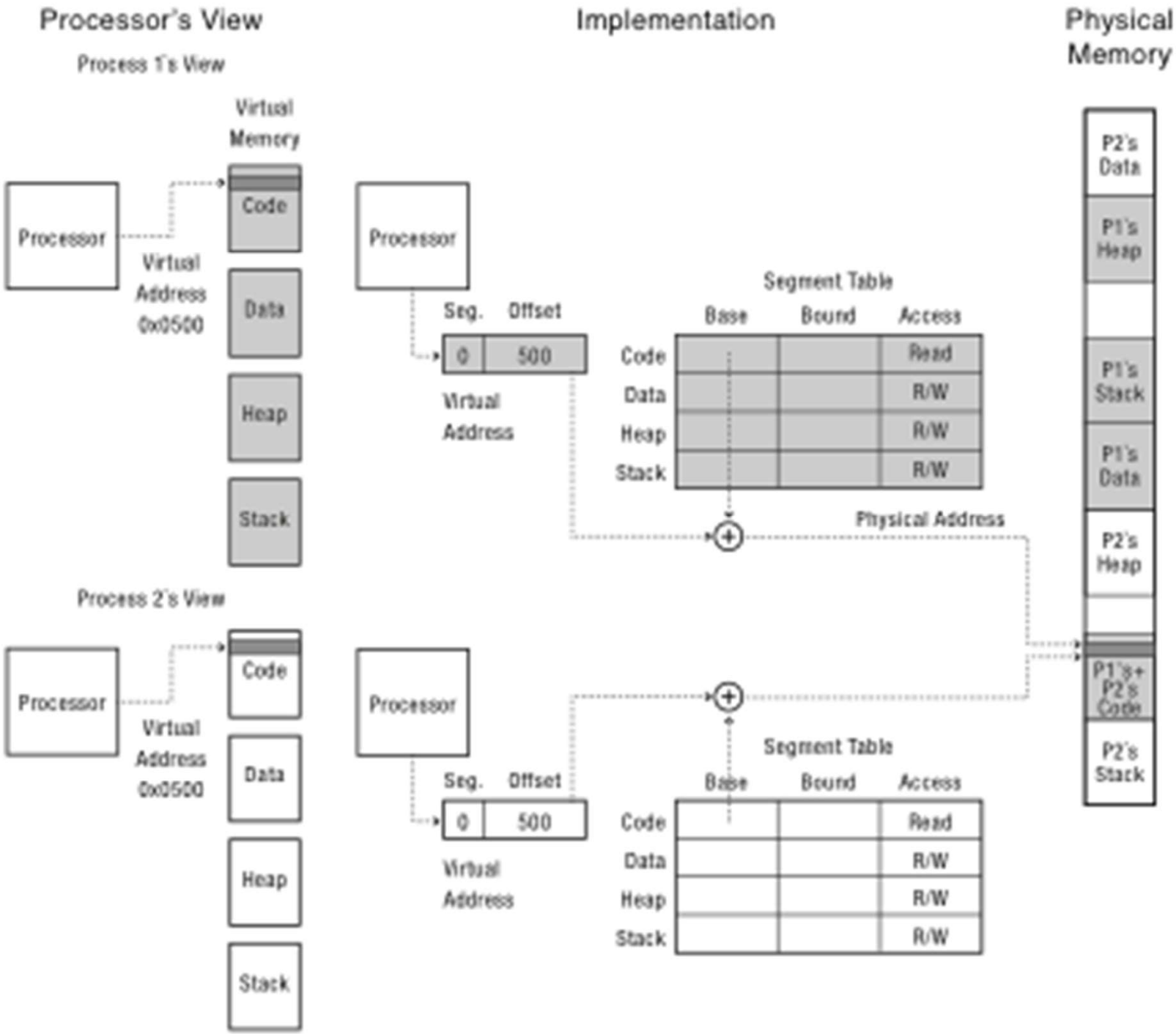
	Segment start	length
code	0x4000	0x700
data	0	0x500
heap	-	-
stack	0x2000	0x1000

Virtual Memory

main: 240	store #1108, r2
244	store pc+8, r31
248	jump 360
24c	
…	
strlen: 360	loadbyte (r2), r3
…	…
420	jump (r31)
…	
x: 1108	a b c \0
…	

Physical Memory

x: 108	a b c \0
…	
main: 4240	store #1108, r2
4244	store pc+8, r31
4248	jump 360
424c	
…	…
strlen: 4360	loadbyte (r2),r3
…	
4420	jump (r31)
…	


Processor's View
Implementation
Physical Memory
Process 1's View
Virtual Memory
Processor
Virtual Address 0x0500
Code
Data
Heap
Stack
Segment Table
Base
Bound
Access
Seg.
Offset
0
500
Virtual Address
Code
Data
Heap
Stack
Read
R/W
R/W
R/W
Physical Address
P2's Data
P1's Heap
P1's Stack
P1's Data
P2's Heap
P1's+ P2's Code
P2's Stack
Process 2's View

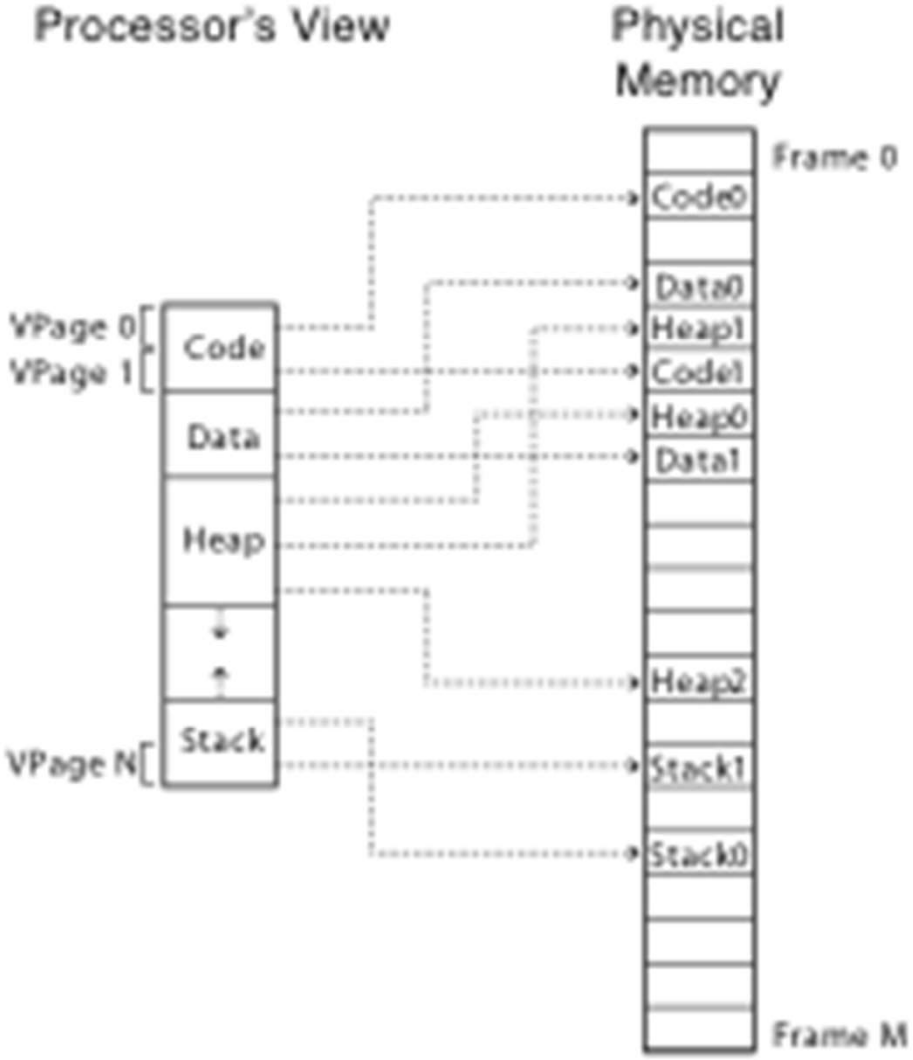
Segmentation

- Pros?
  - Can share code/data segments between processes
  - Can protect code segment from being overwritten
  - Can transparently grow stack/heap as needed
  - Can detect if need to copy-on-write
- Cons?
  - Complex memory management
    - Need to find chunk of a particular size
  - May need to rearrange memory from time to time to make room for new segment or growing segment
    - External fragmentation: wasted space between chunks

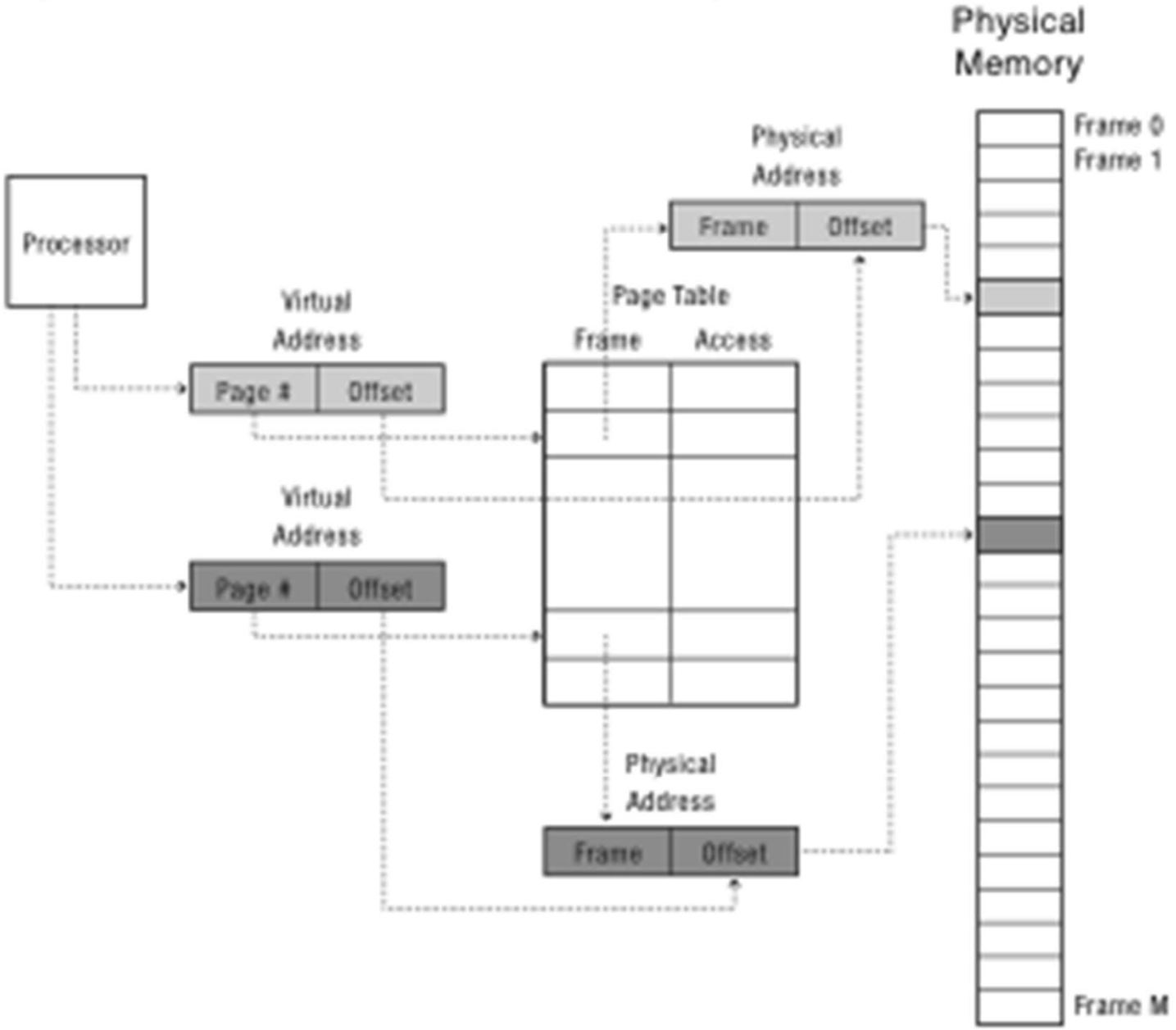
Paged Translation

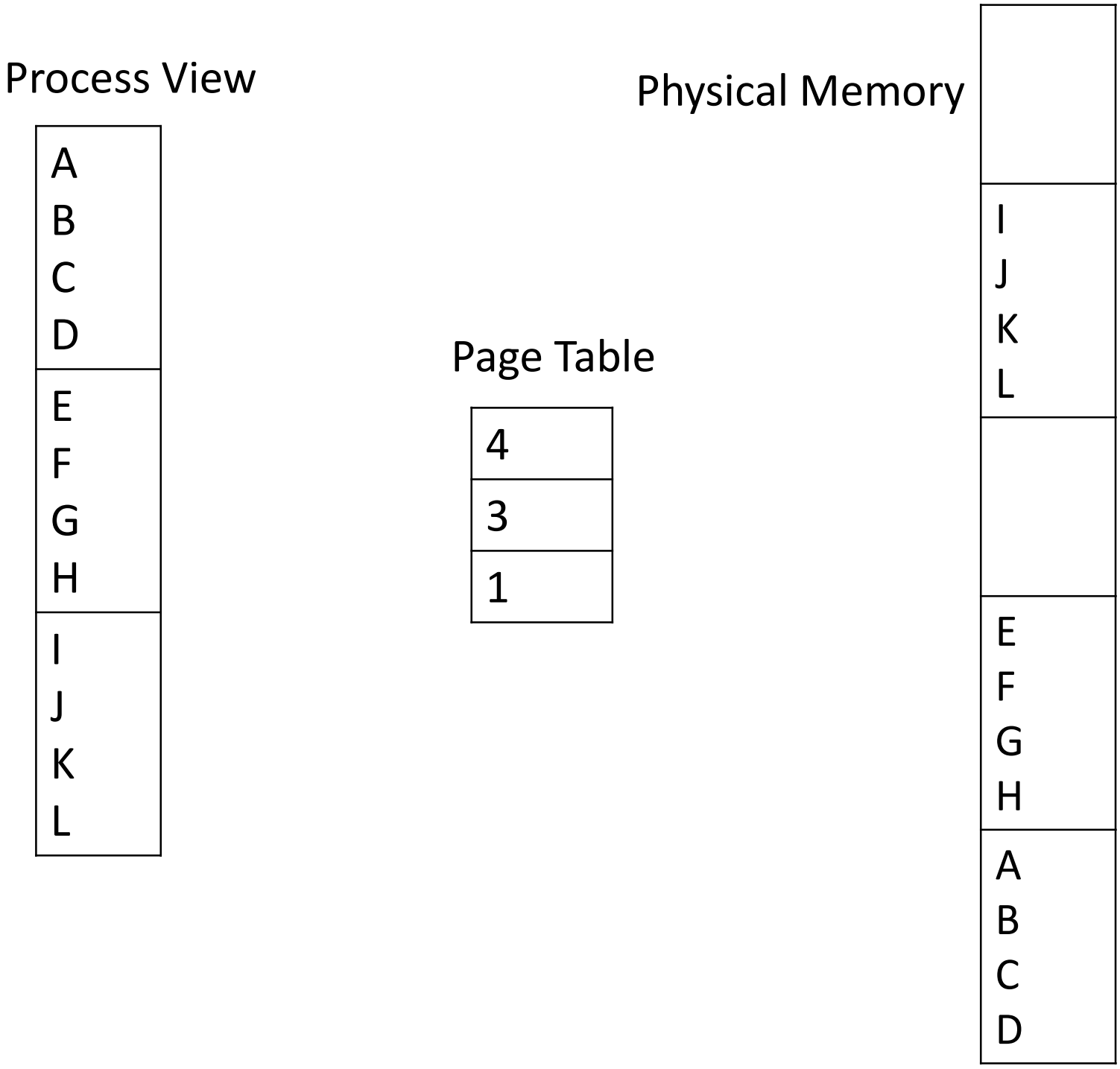
- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length

Paged Translation (Abstract)



Paged Translation (Implementation)




Process View
A
B
C
D
E
F
G
H
I
J
K
L
Page Table
4
3
1
Physical Memory
I
J
K
L
E
F
G
H
A
B
C
D

Large Address Spaces

- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

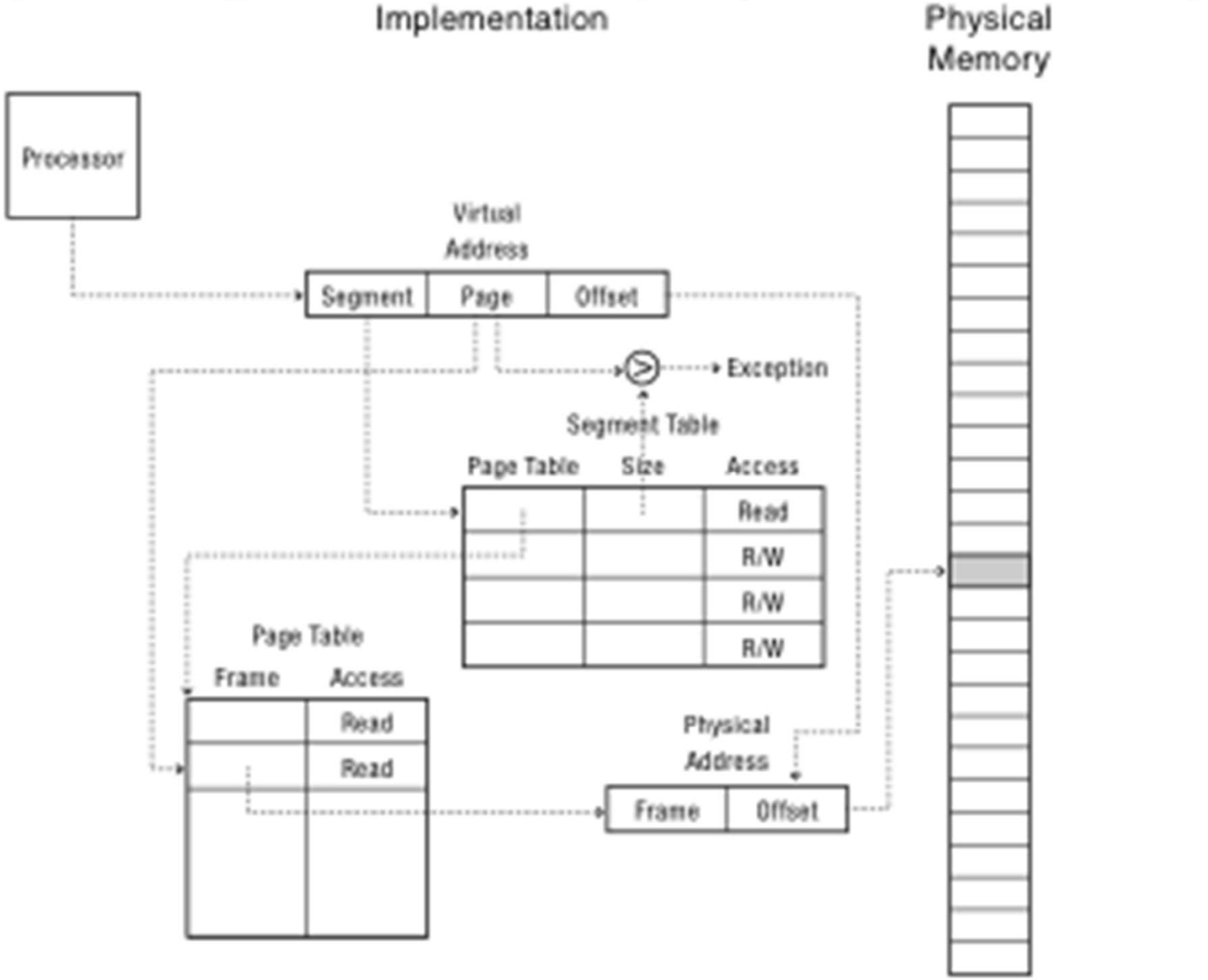
Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- Fixed-size page as lowest level unit of allocation
  - Efficient memory allocation (compared to segments)
  - Efficient for sparse addresses (compared to paging)
  - Efficient disk transfers (fixed size units)
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
  - Variable granularity for protection/sharing

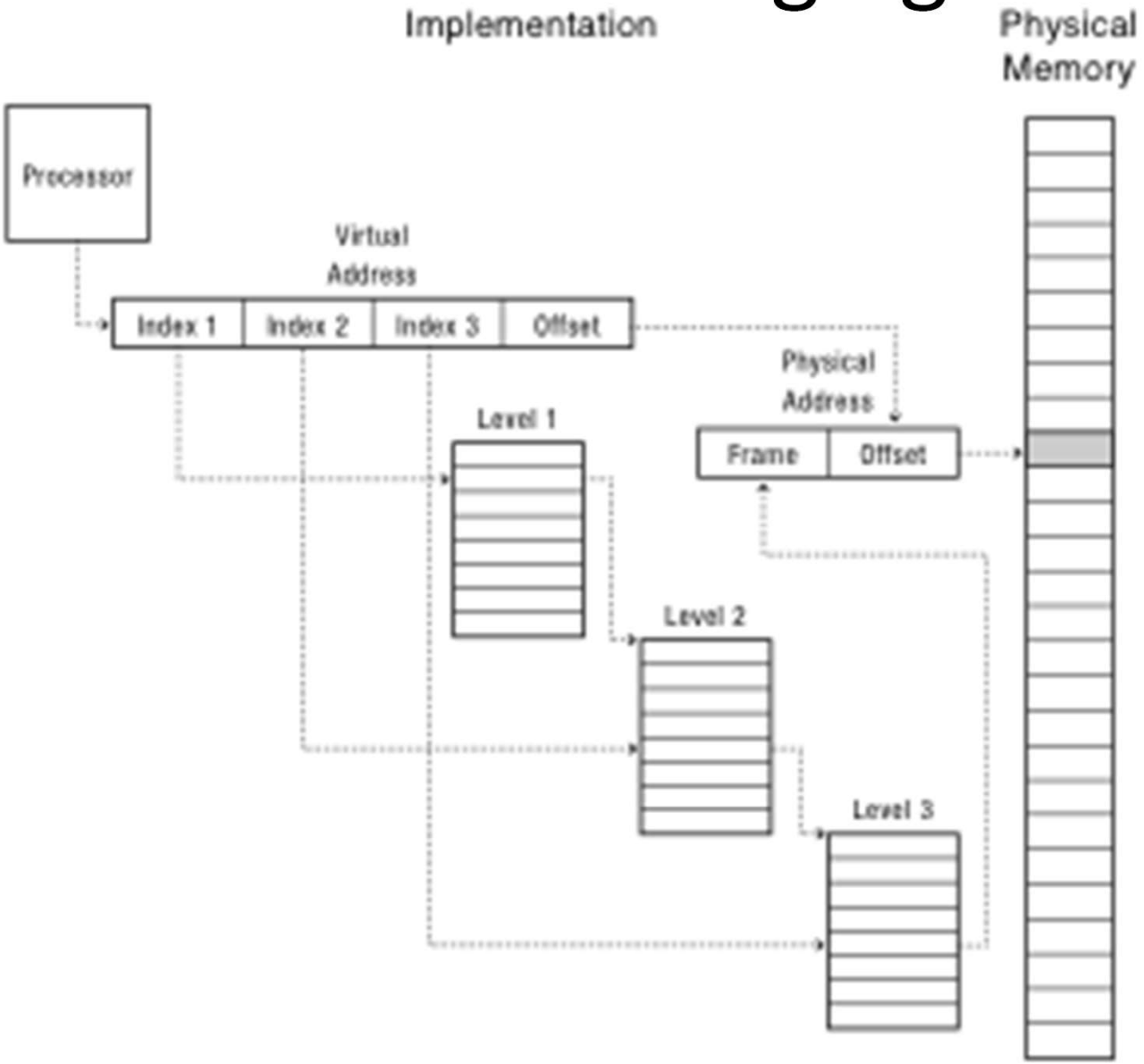
Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Paged Segmentation (Implementation)



Multilevel Paging



Multilevel Translation

- Pros:
  - Allocate/fill only page table entries that are in use
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Space overhead: one pointer per virtual page
  - Two (or more) lookups per memory reference

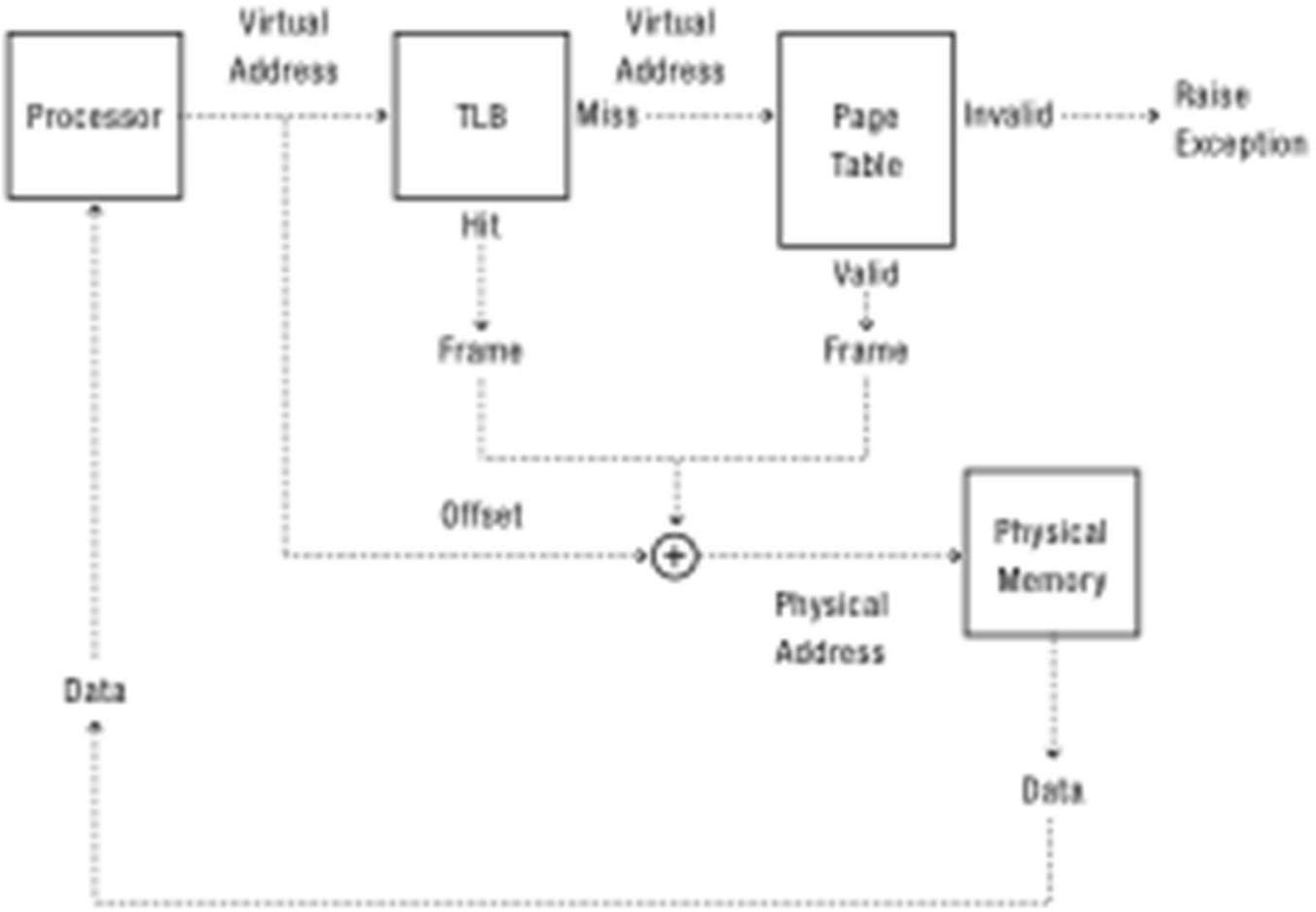
Efficient Address Translation

- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table
- Cost of translation =

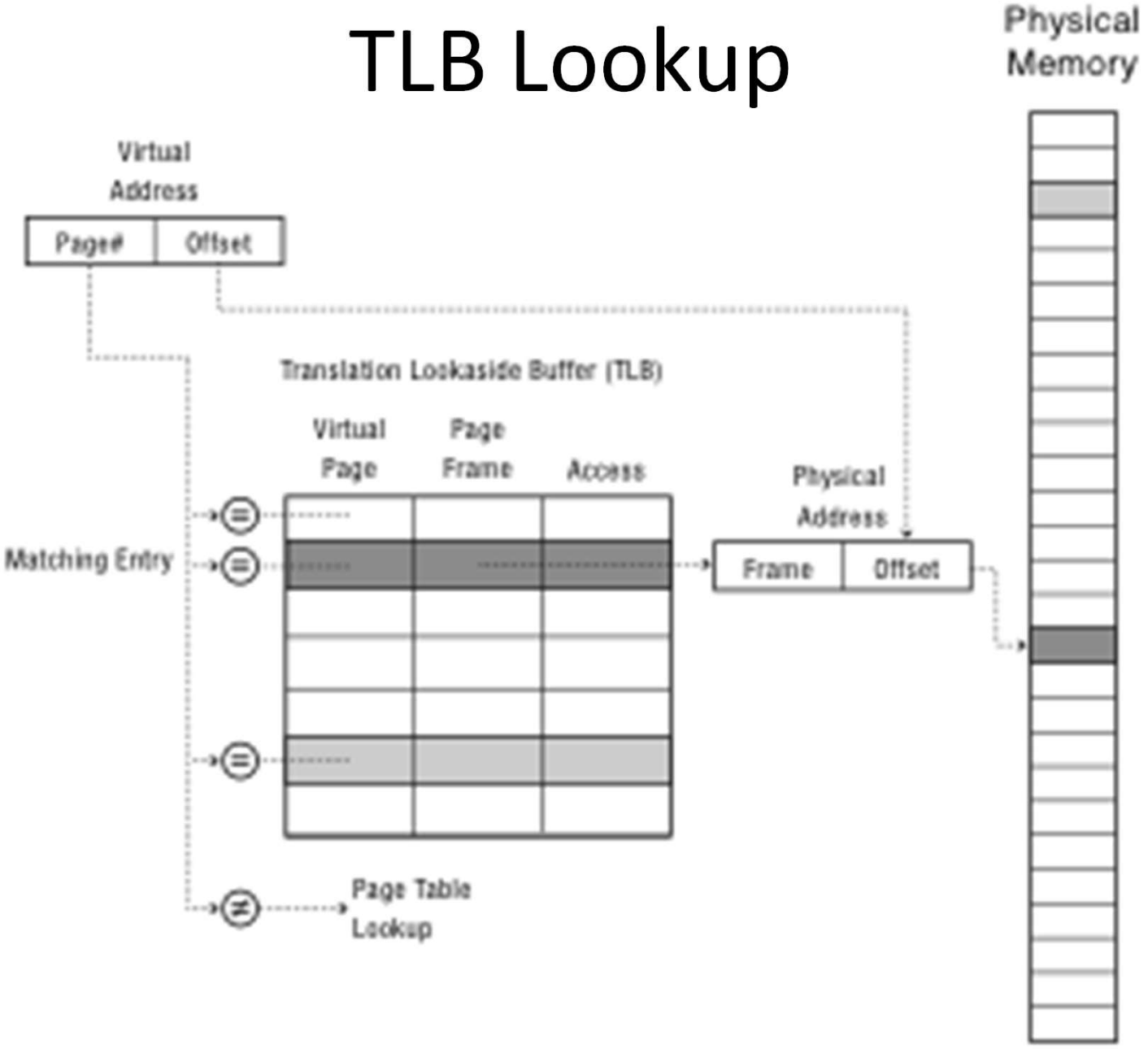
  Cost of TLB lookup +

  Prob(TLB miss) * cost of page table lookup

TLB and Page Table Translation



TLB Lookup


Physical Memory
Virtual Address
Page#
Offset
Translation Lookaside Buffer (TLB)
Virtual Page
Page Frame
Access
Physical Address
Matching Entry
Frame
Offset
Page Table Lookup

Address Translation Uses

- Process isolation
  - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand