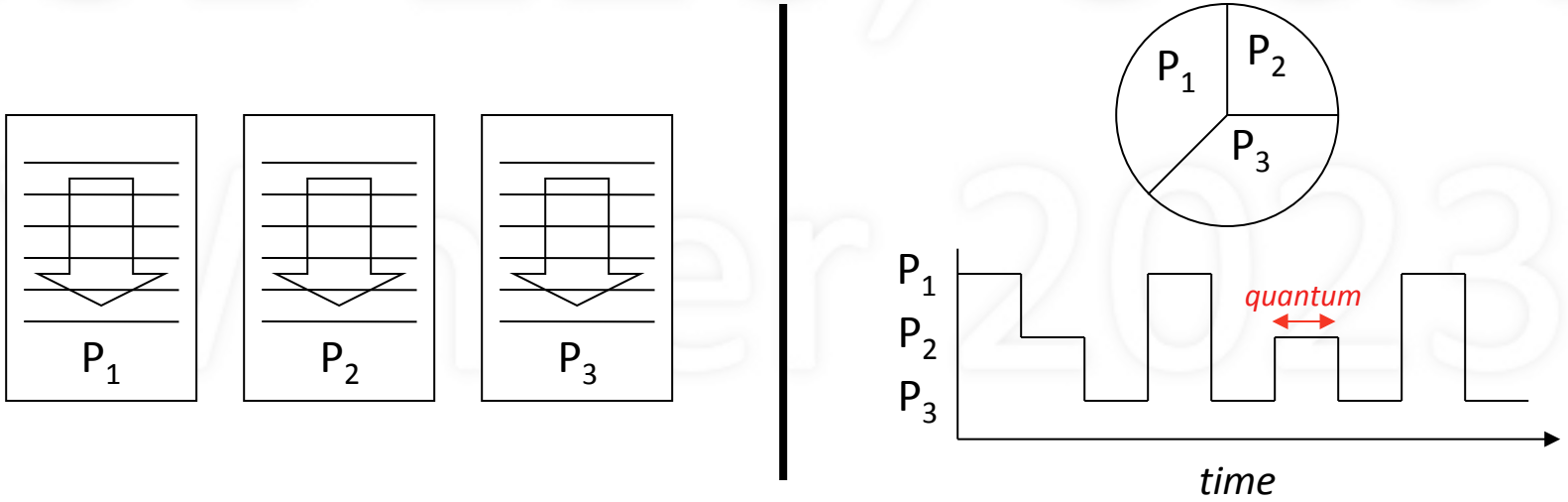
CSE 120: Principles of Operating Systems
Lecture 3: Timesharing

Prof. Joseph Pasquale
University of California, San Diego

January 18, 2023

Timesharing

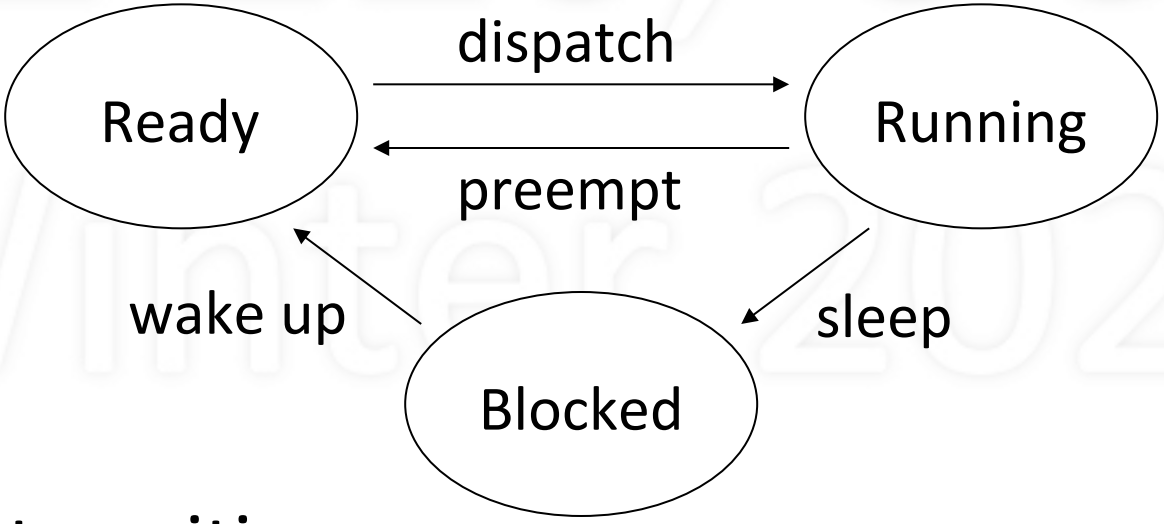


- Timesharing: multiplexing use of CPU over time
- Multiple processes, single CPU (uniprocessor)
- Conceptually, each process makes progress over time
- In reality, each periodically gets quantum of CPU time
- Illusion of parallel progress by rapidly switching CPU

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
  - Running: actually making progress, using CPU
  - Ready: able to make progress, but not using CPU
  - Blocked: not able to make progress, can't use CPU
- Kernel selects a ready process, lets it run
  - Eventually, the kernel gets back control
  - Selects another ready process to run, …

Process State Diagram



- State transitions
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready

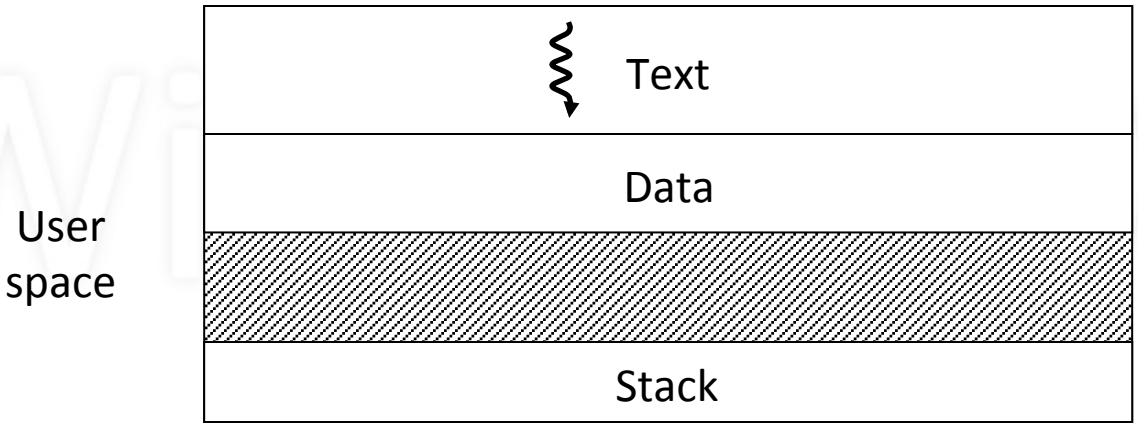
Logical vs. Physical Execution

Physical Execution	Logical Execution: Able to execute	Logical Execution: Not able to execute
Actually executing	Run	X
Not actually executing	Ready	Blocked

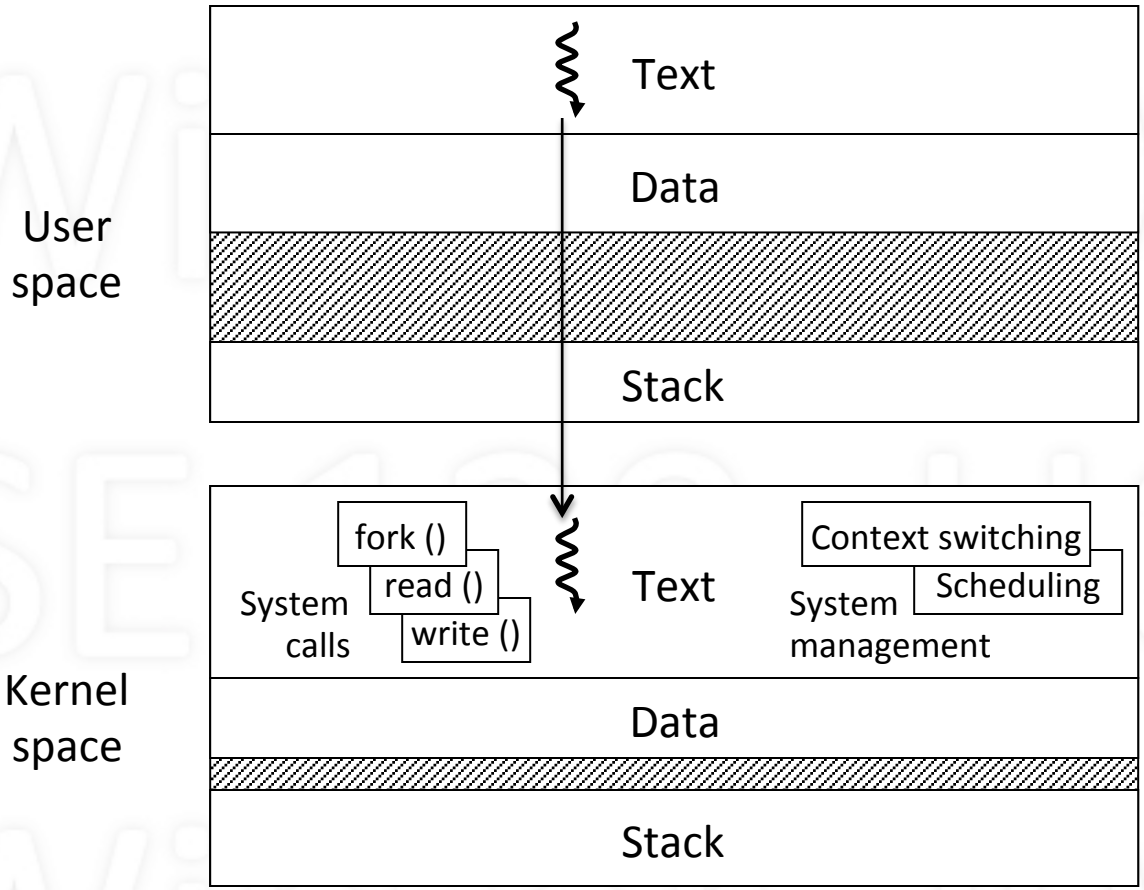
Process vs. Kernel

- Kernel: code that supports processes
  - system calls: fork ( ), exit ( ), read ( ), write ( ), …
  - management: context switching, scheduling, …
- When does the kernel run?
  - when system call or hardware interrupt occurs
- The kernel runs as part of the running process
  - due to that process having made a system call
  - in response to device issuing interrupt

Process Running in User Space



Process Running in Kernel Space



Kernel Maintains List of Processes

Process ID	State	Other info
1534	Ready	Saved context, …
34	Running	Memory areas used, …
487	Ready	Saved context, …
9	Blocked	Condition to unblock, …

- All processes: unique names (IDs) and states
- Other info kernel needs for managing system
  - contents of CPU contexts
  - areas of memory being used
  - reasons for being blocked

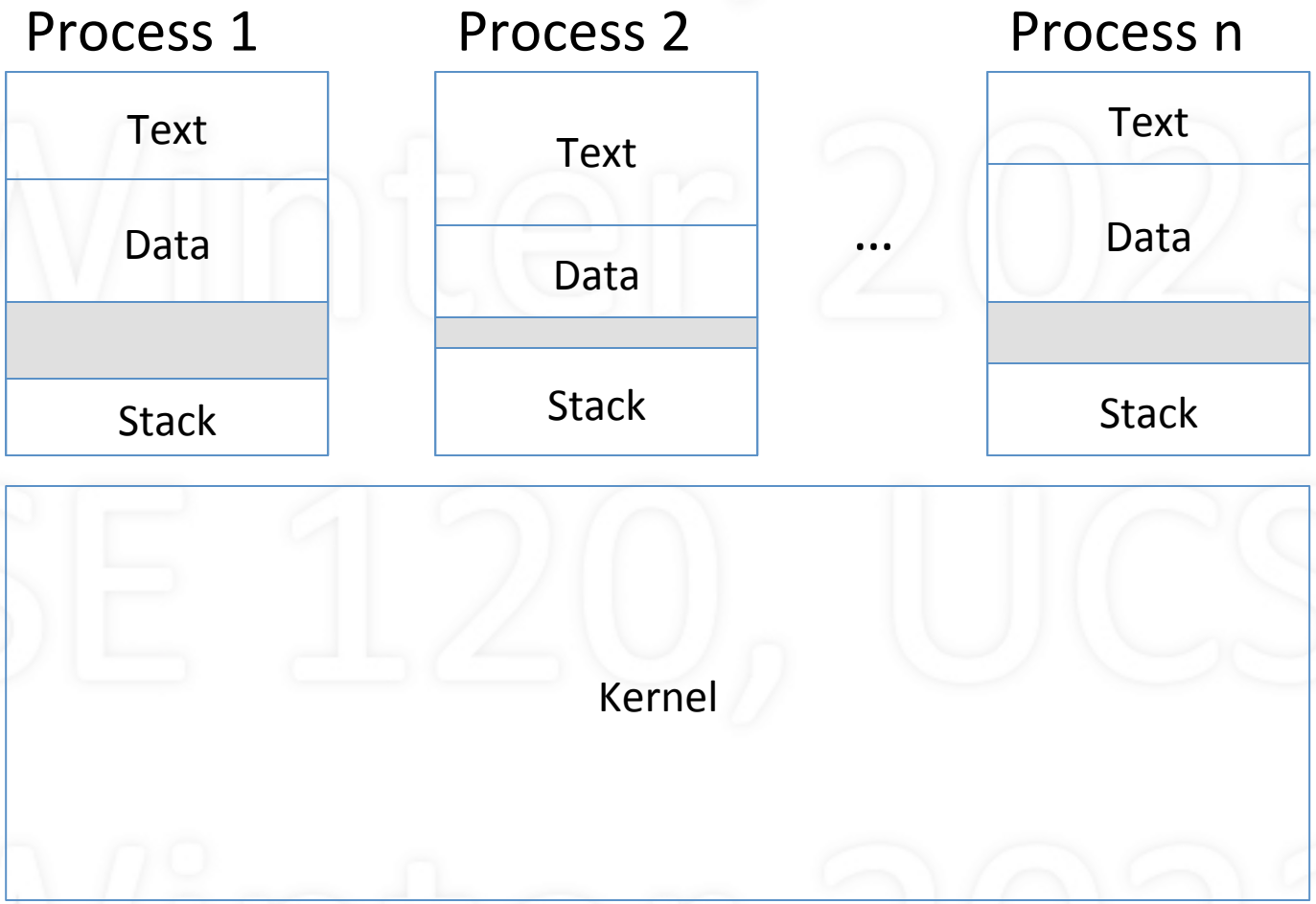
How Does Kernel Get Control

- Process can give up control voluntarily
  - Makes system call that blocks, e.g., read ()
  - System-call function calls yield () to give up CPU
  - Kernel selects a ready process, dispatches it
- Or, CPU is forcibly taken away: *preemption*
  - Interrupt generated when hardware timer expires
  - Interrupt forces control to go to kernel
  - While kernel running, resets timer for next time

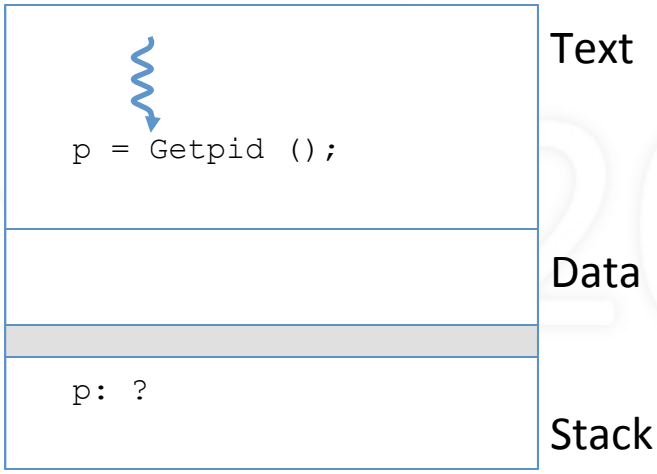
How a Context Switch Occurs

- Process makes system call or interrupt occurs
- What's done by hardware
  - Switch from user to kernel mode: amplifies power
  - Go to fixed kernel location: trap/interrupt handler
- What's done in software (in the kernel)
  - Save context of current process
  - Select a process that is ready; restore its context
  - RTI: return from interrupt/trap

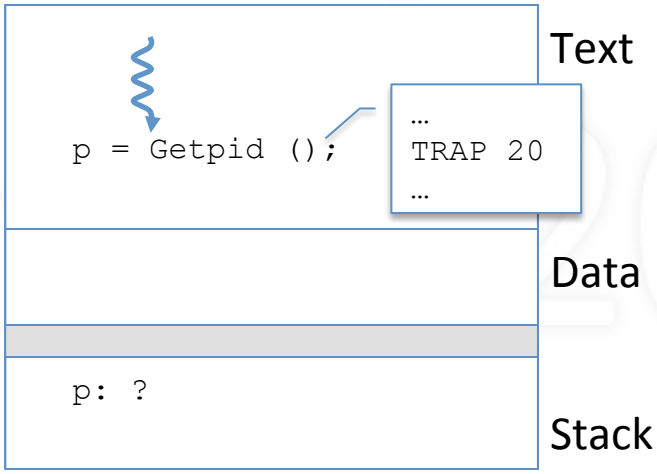
Example



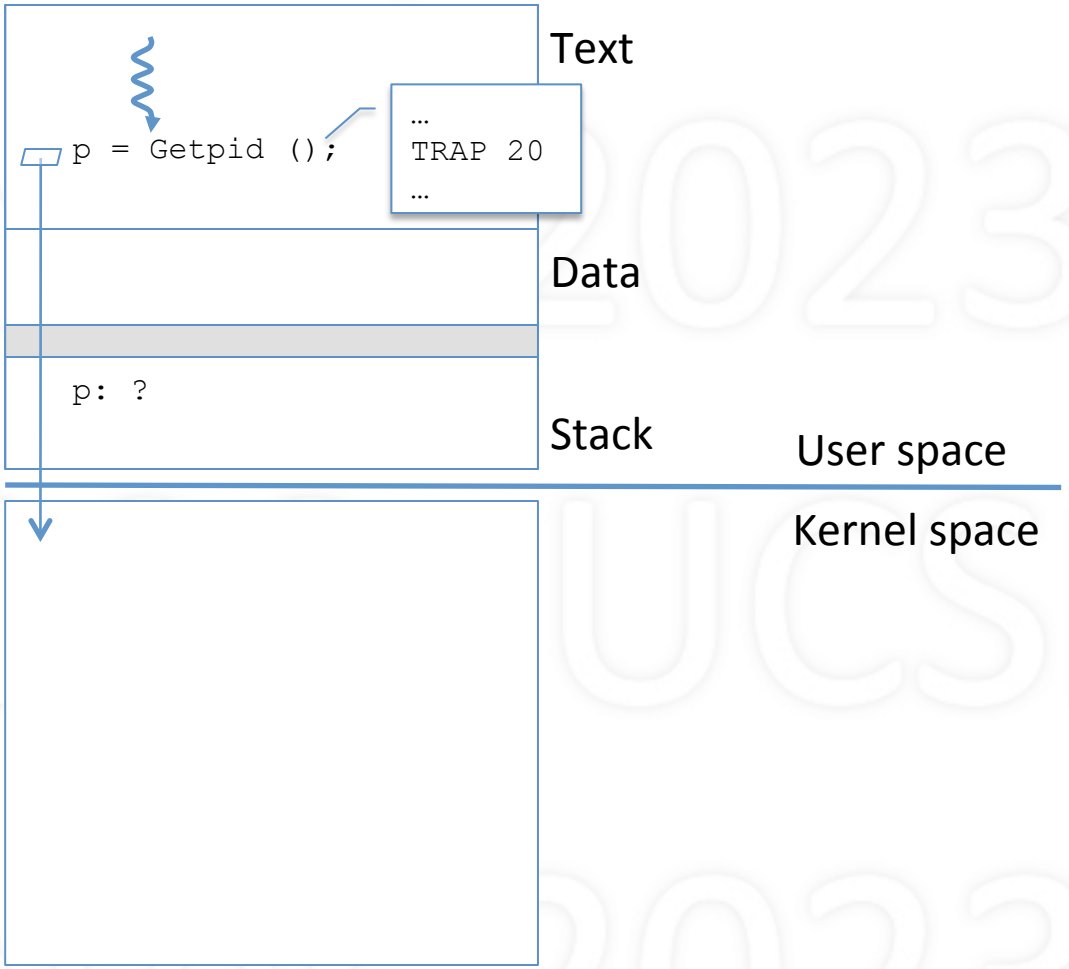
Process makes system call



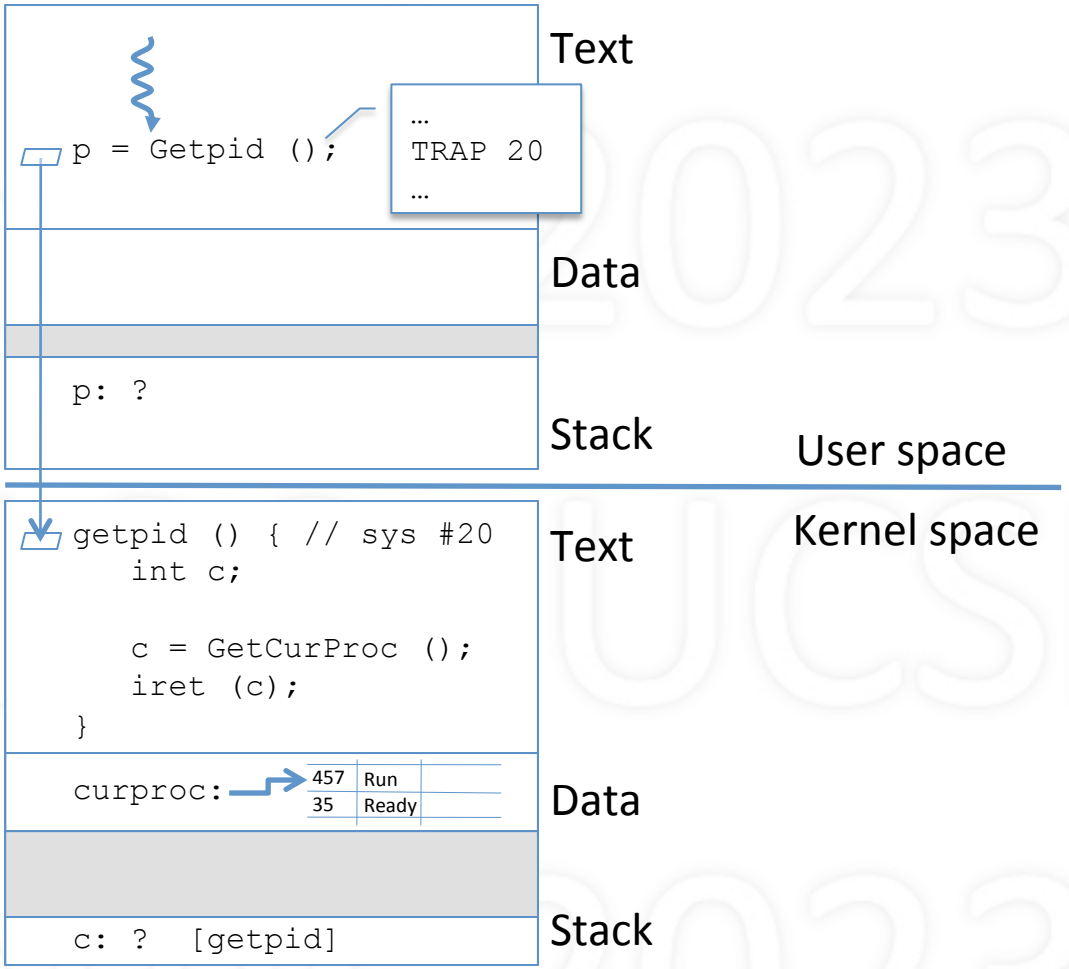
Translates into a TRAP



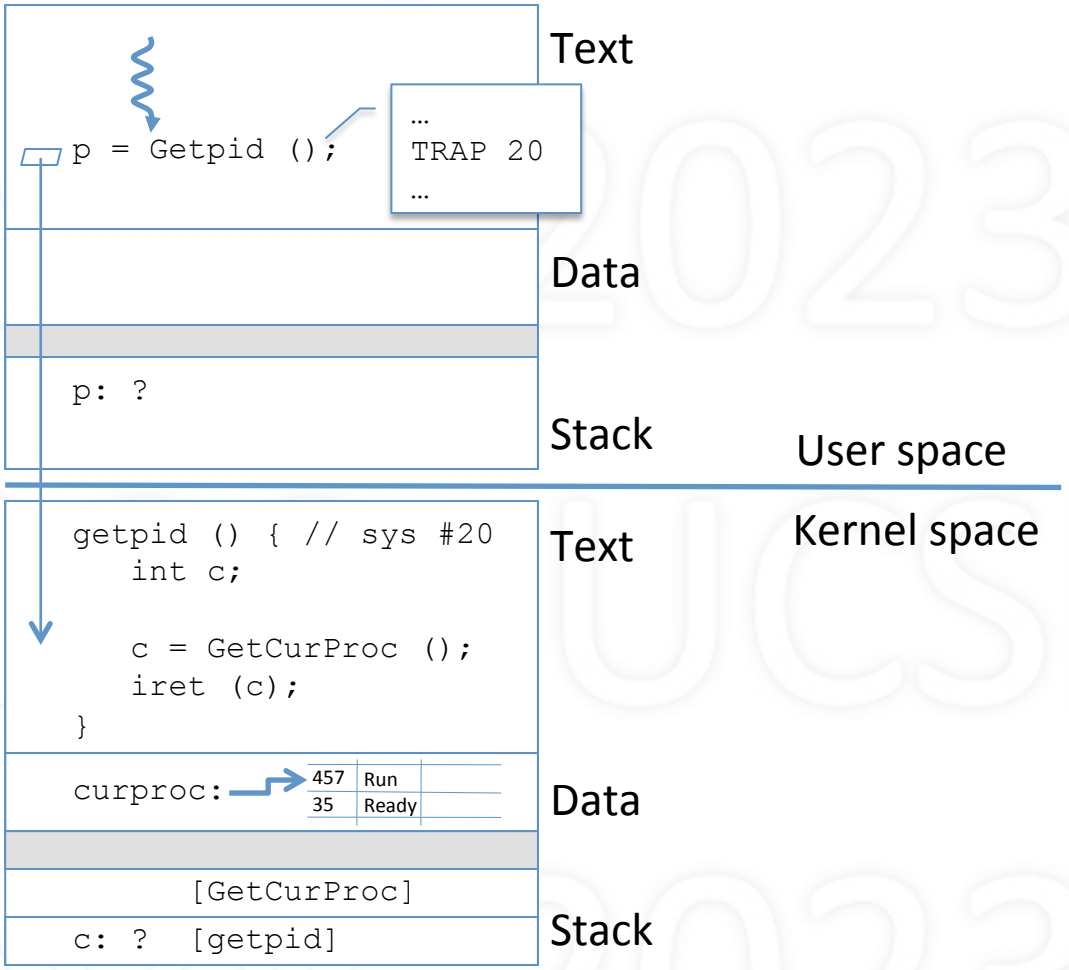
TRAP causes entry into kernel



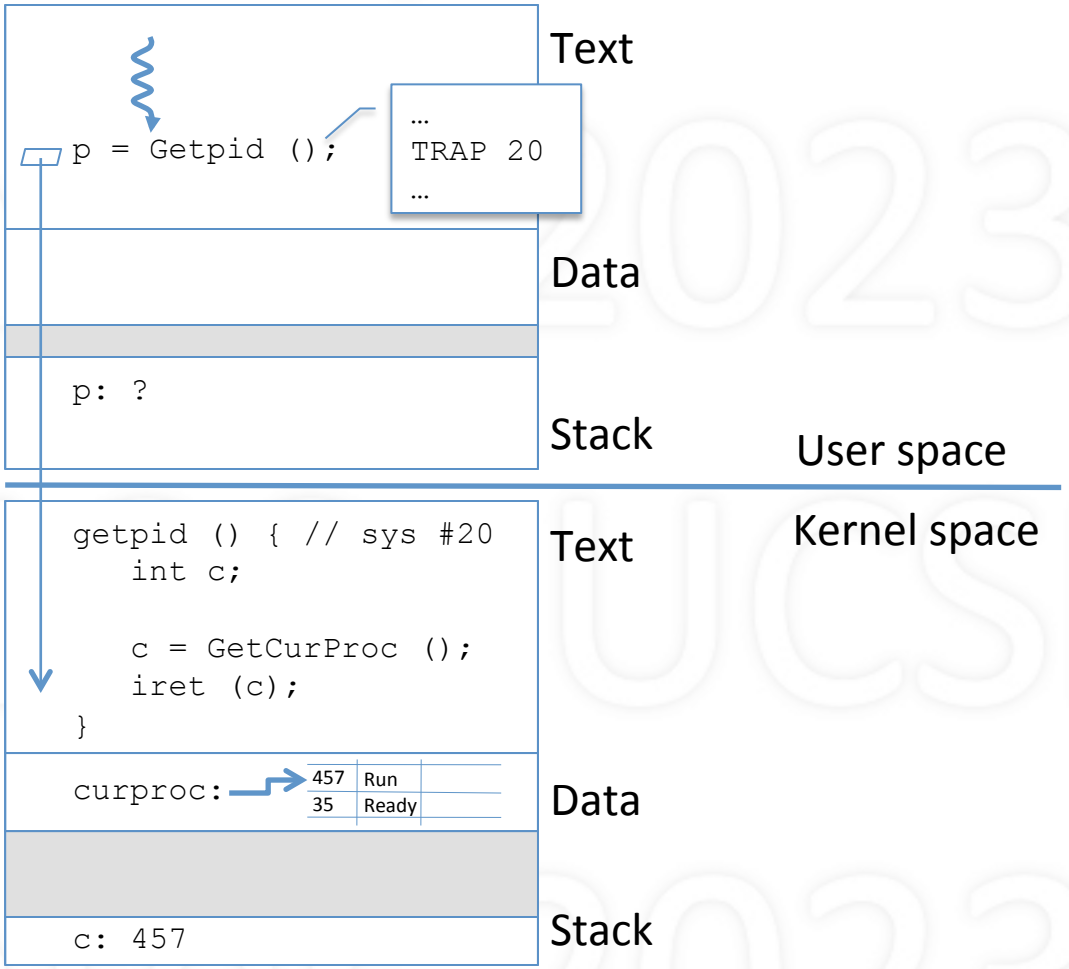
Jump to proper system function



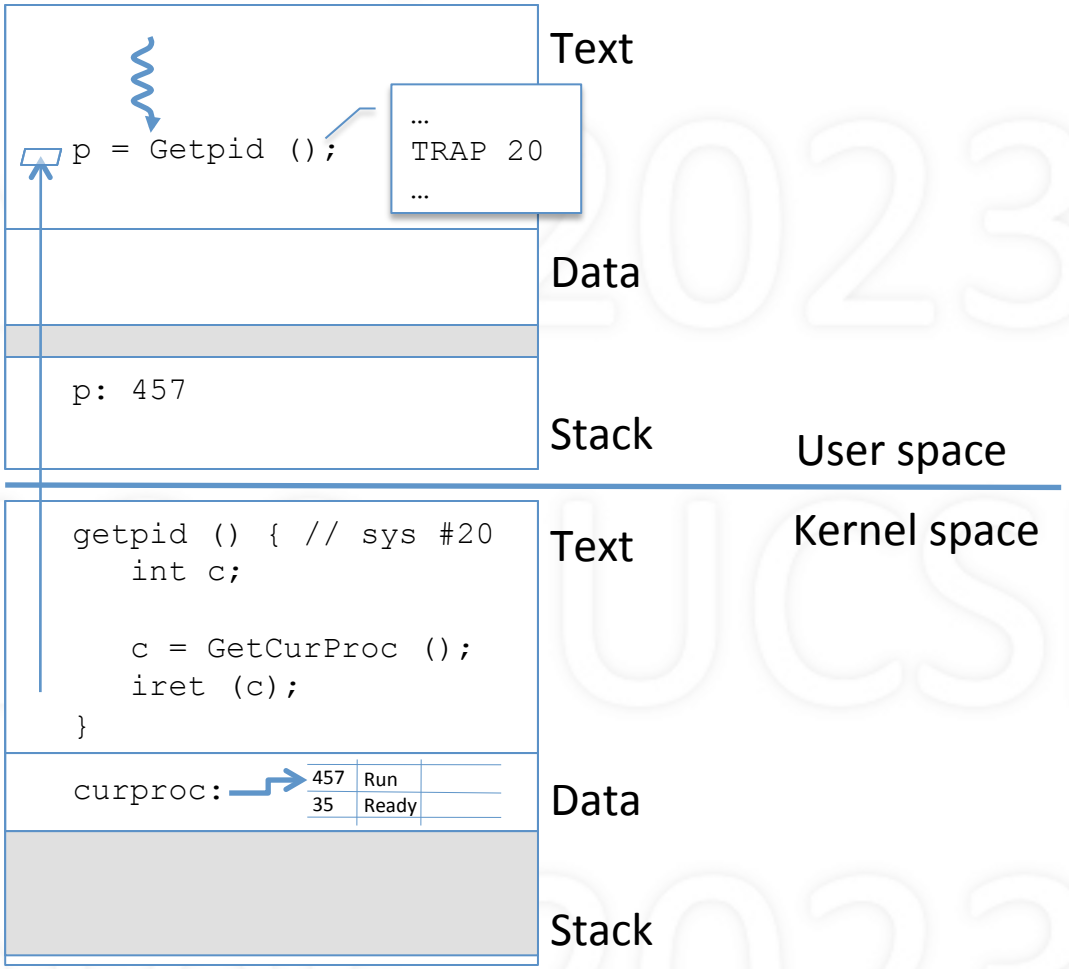
System function starts executing



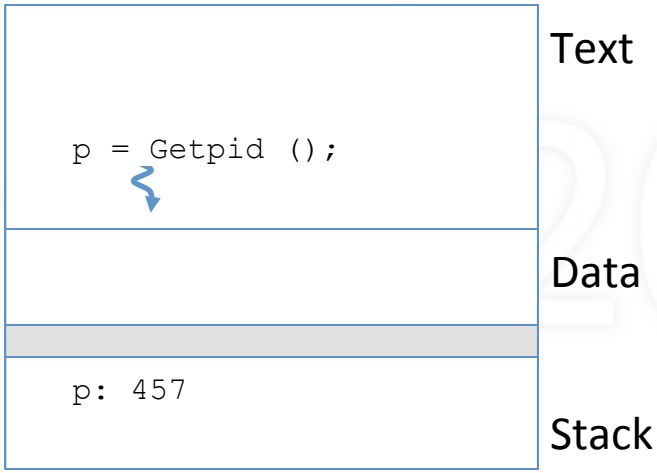
System function completes



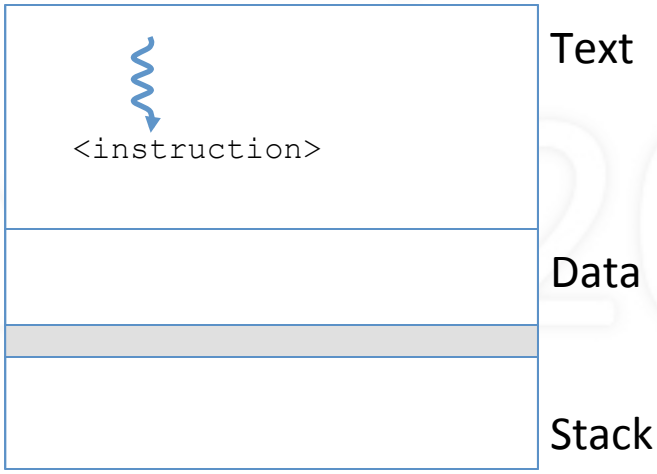
Return from kernel to process



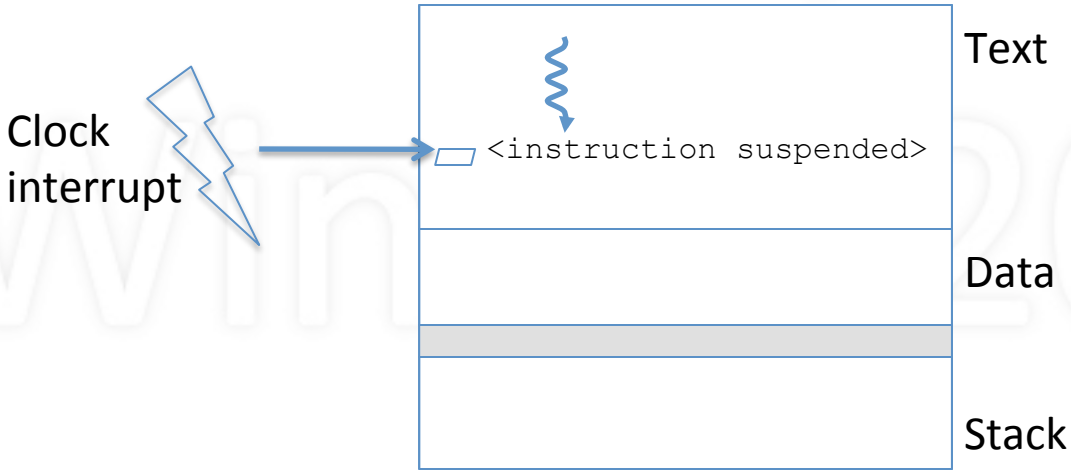
Process continues after system call



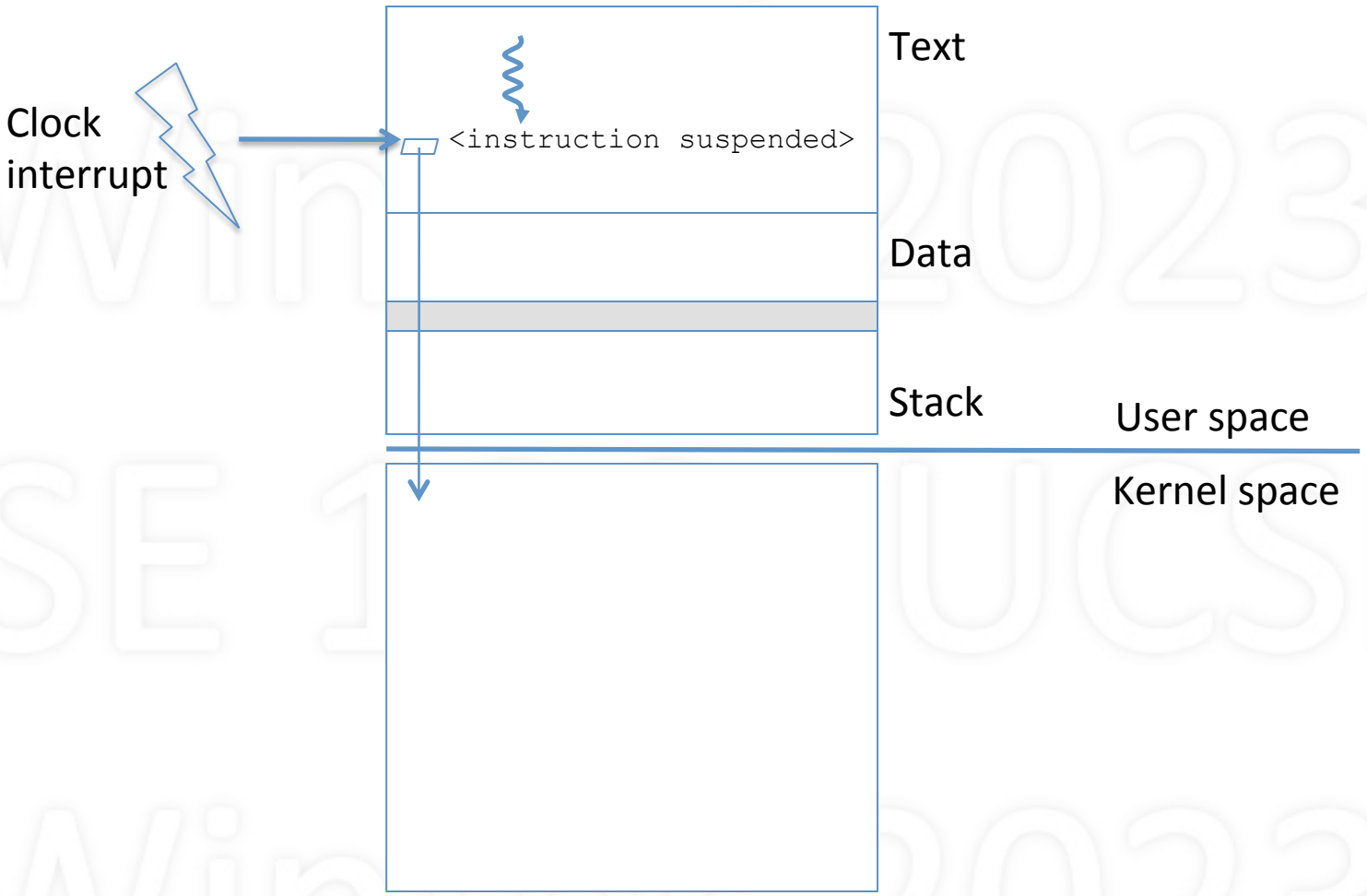
Kernel entry via clock interrupt



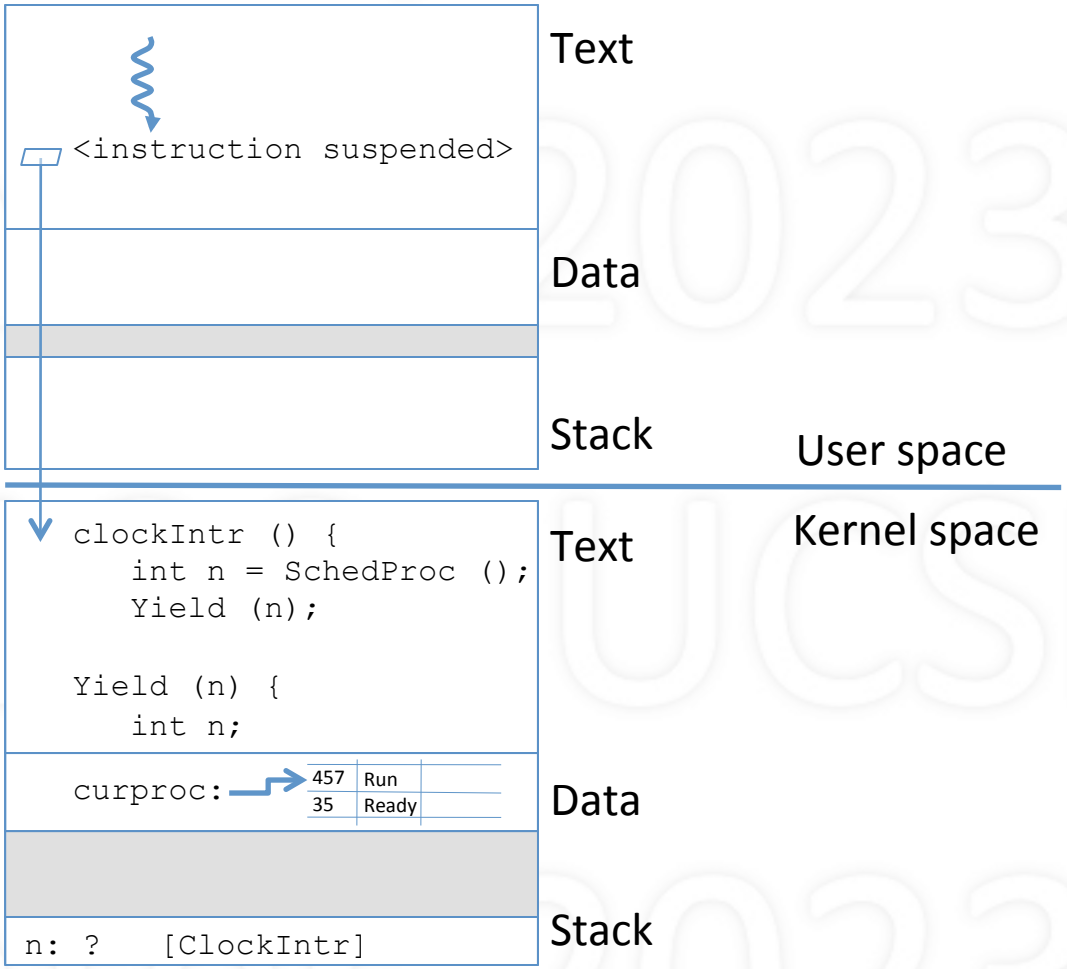
Asynchronous clock interrupt



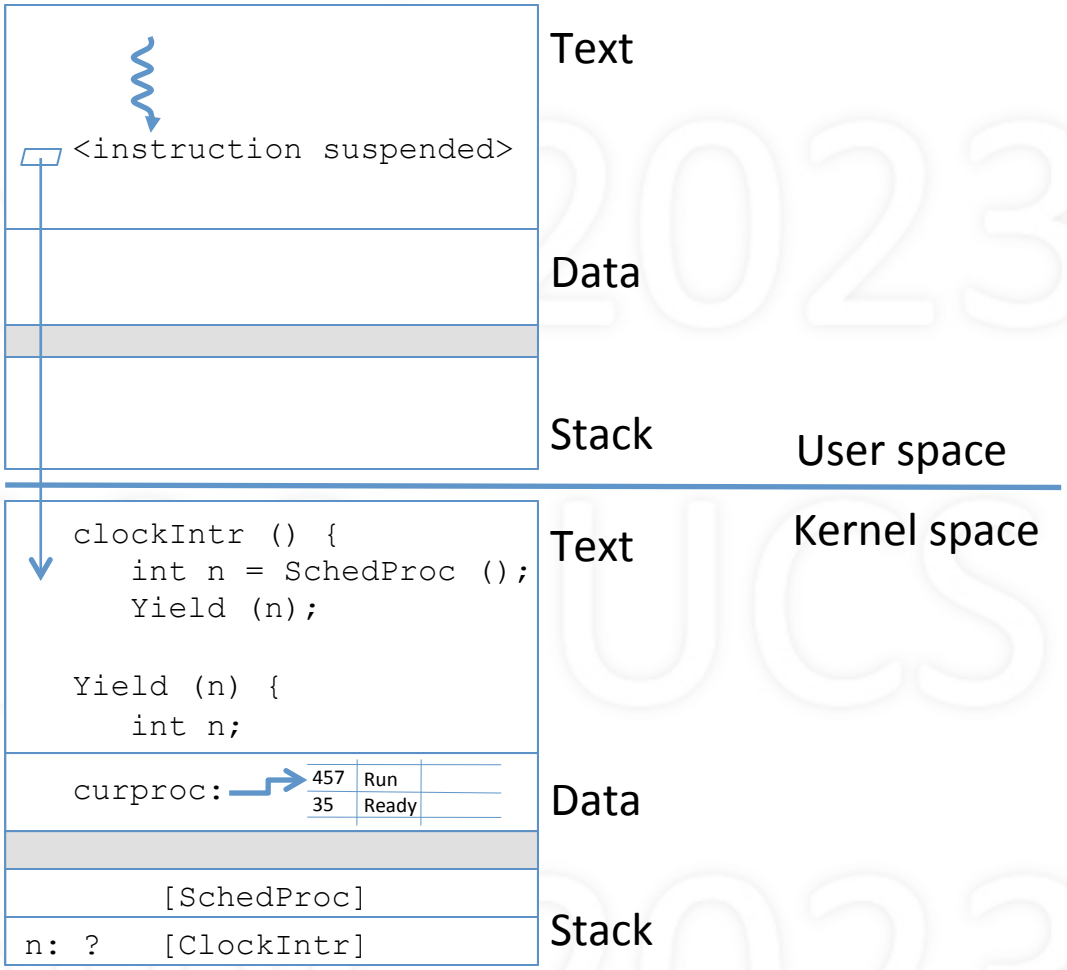
Causes forced kernel entry



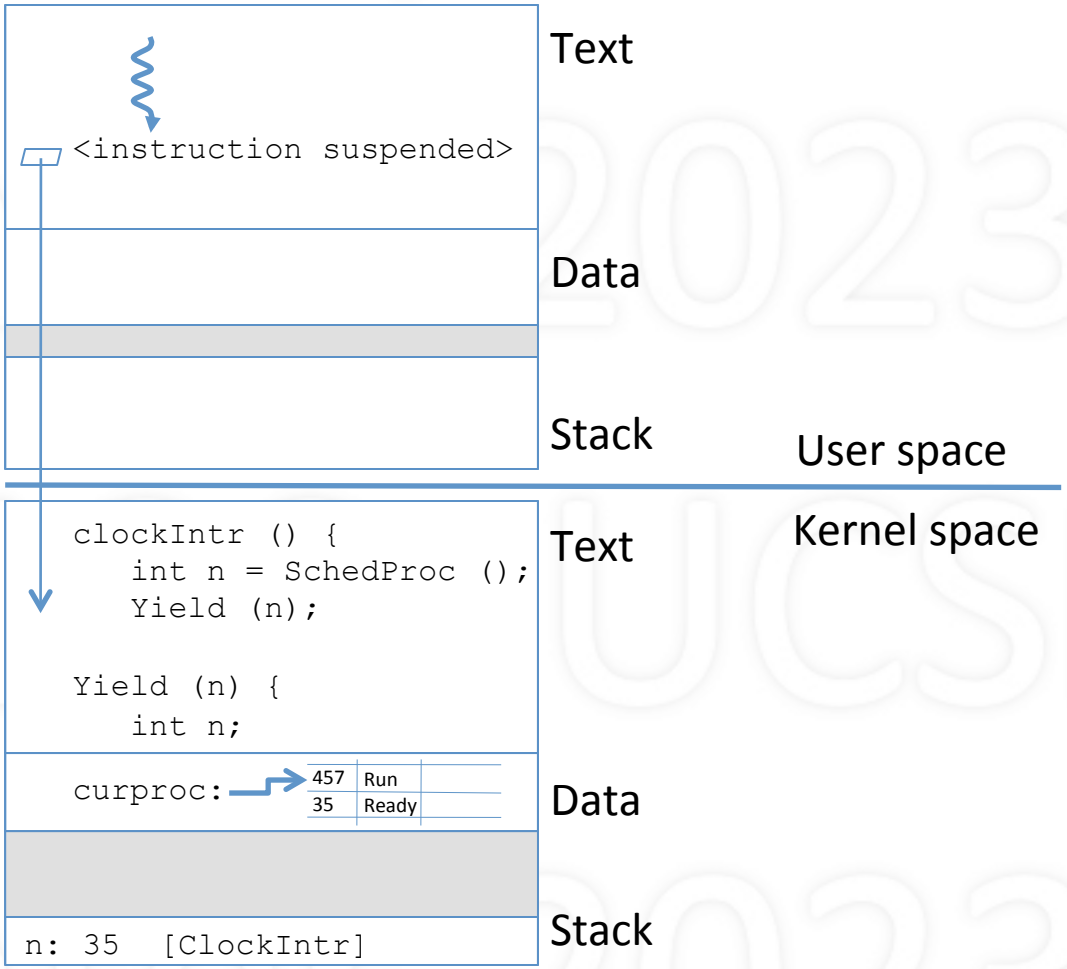
Clock interrupt handler executes



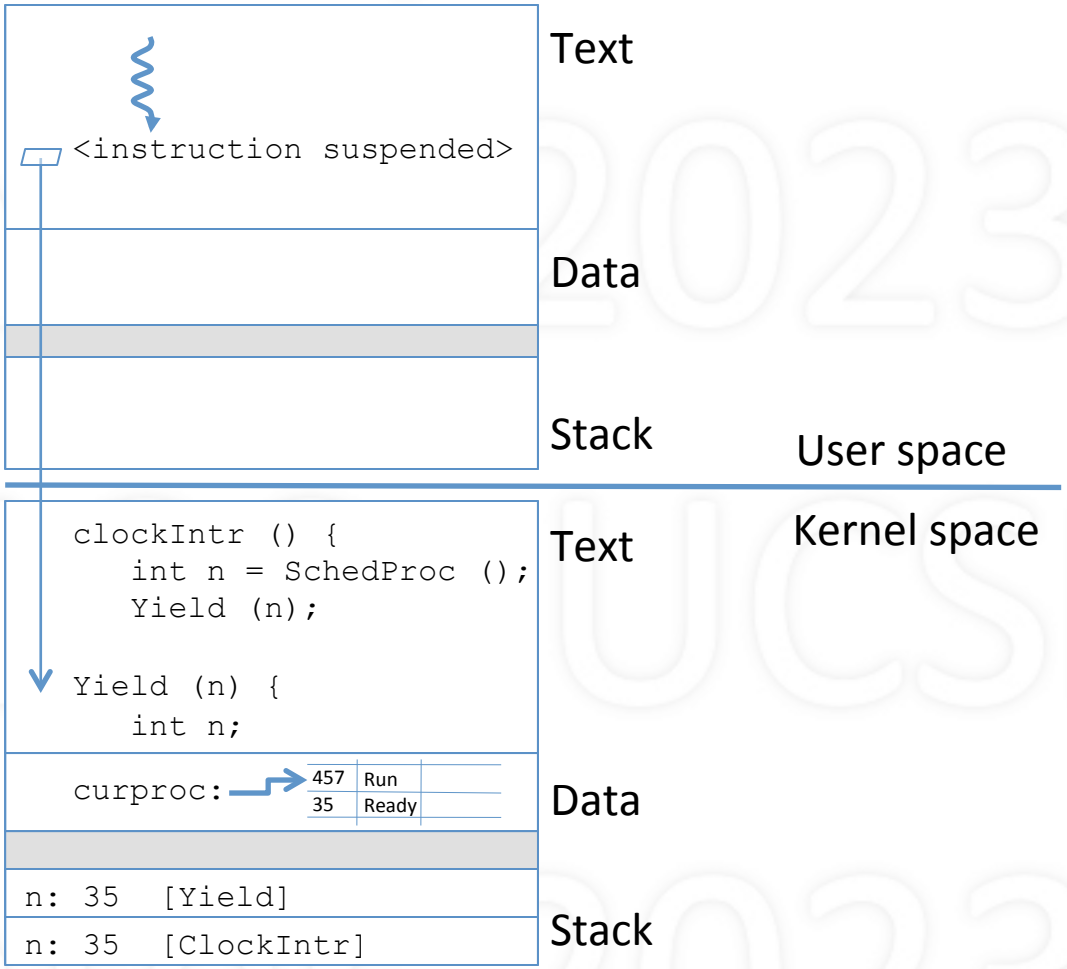
Decide which process to run next



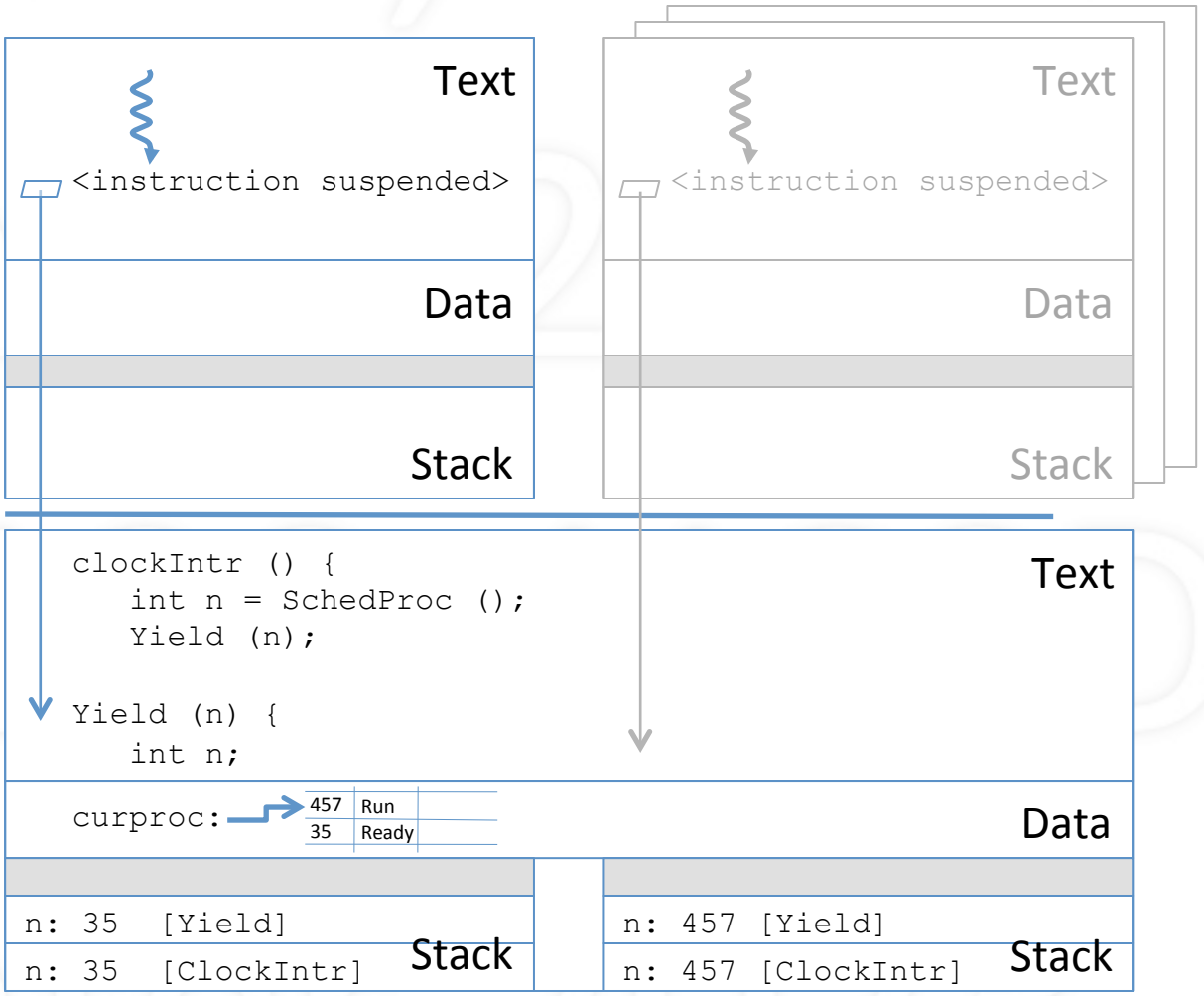
Yield to that process



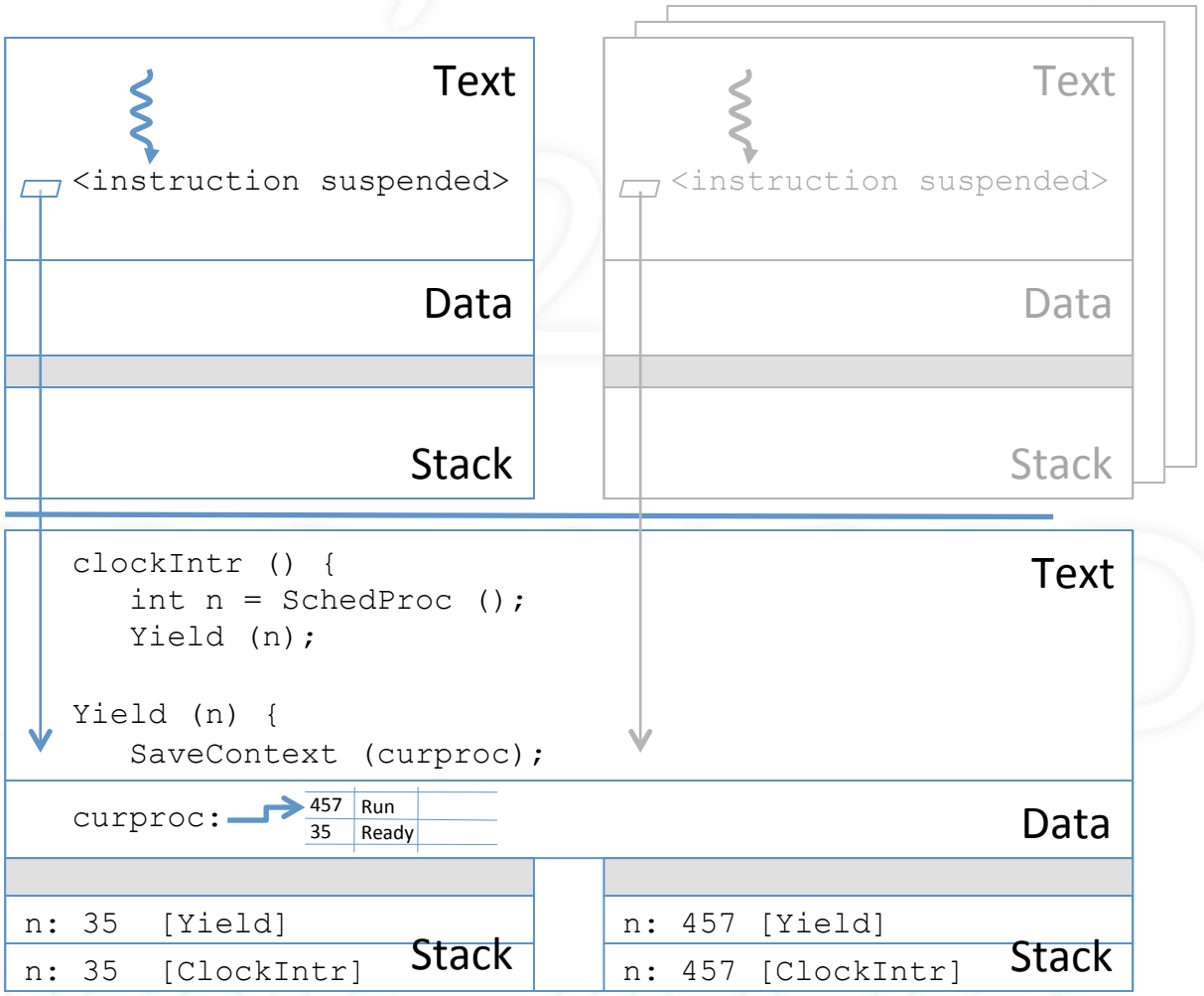
Call to yield



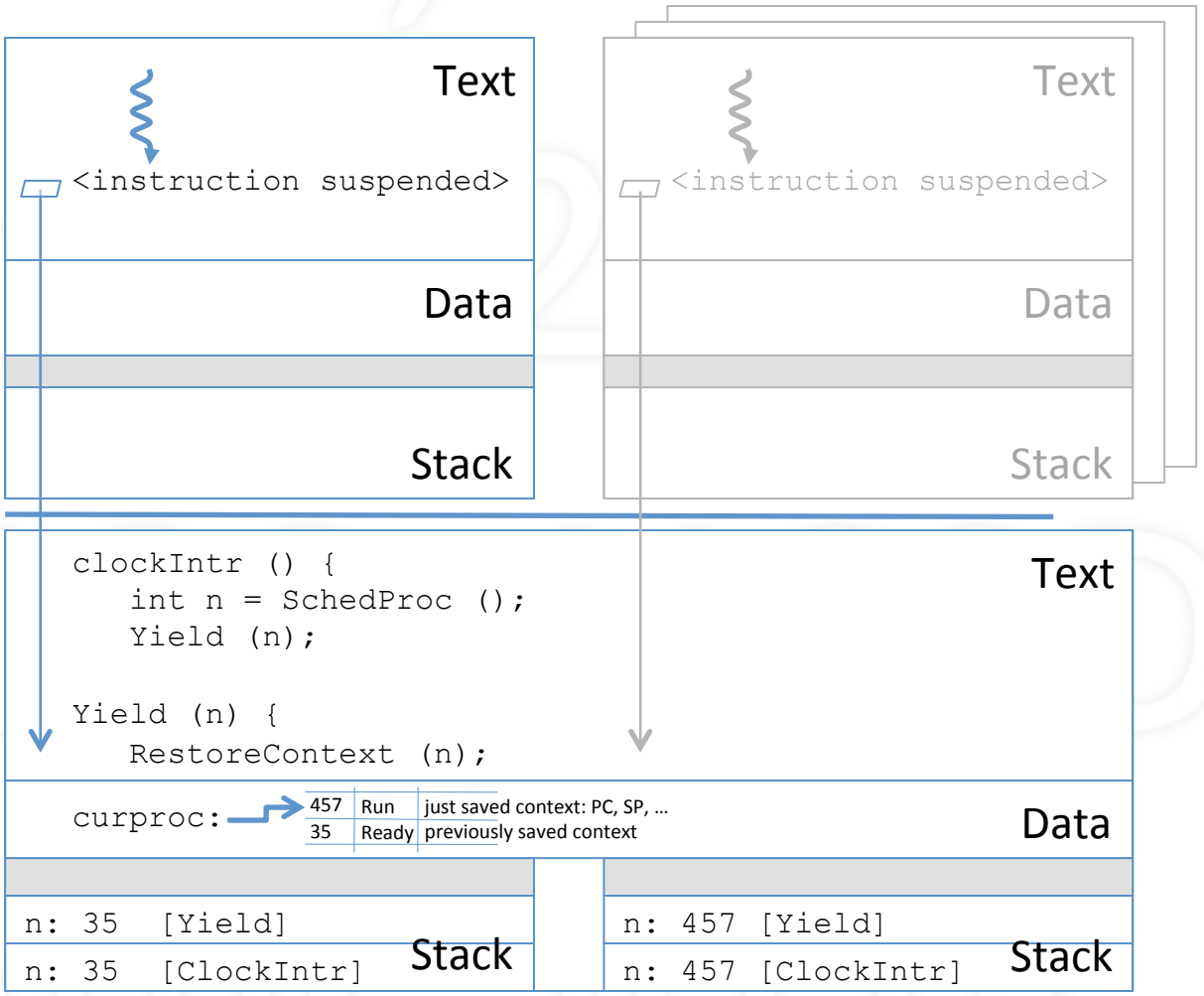
Yield magic



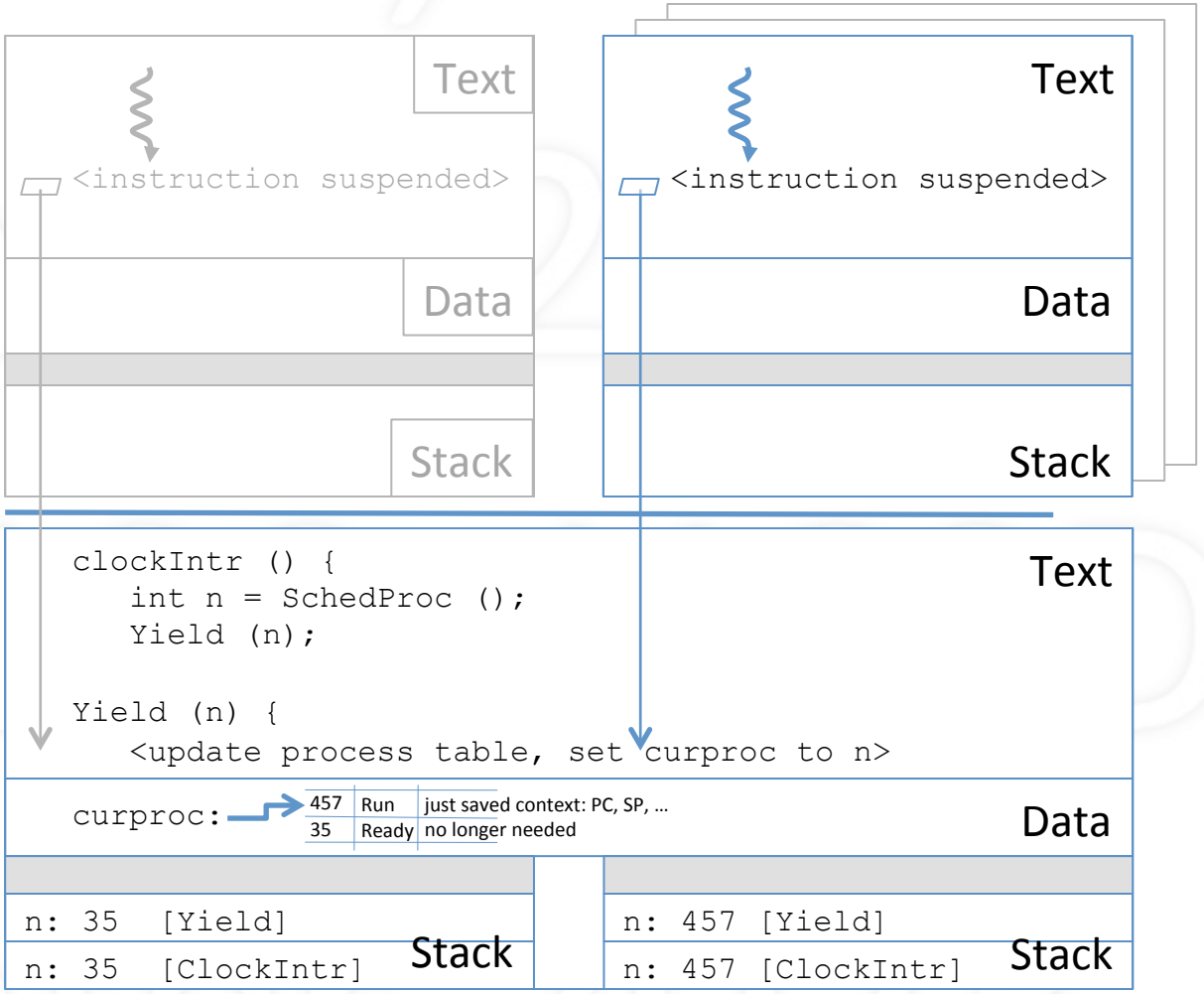
Save context



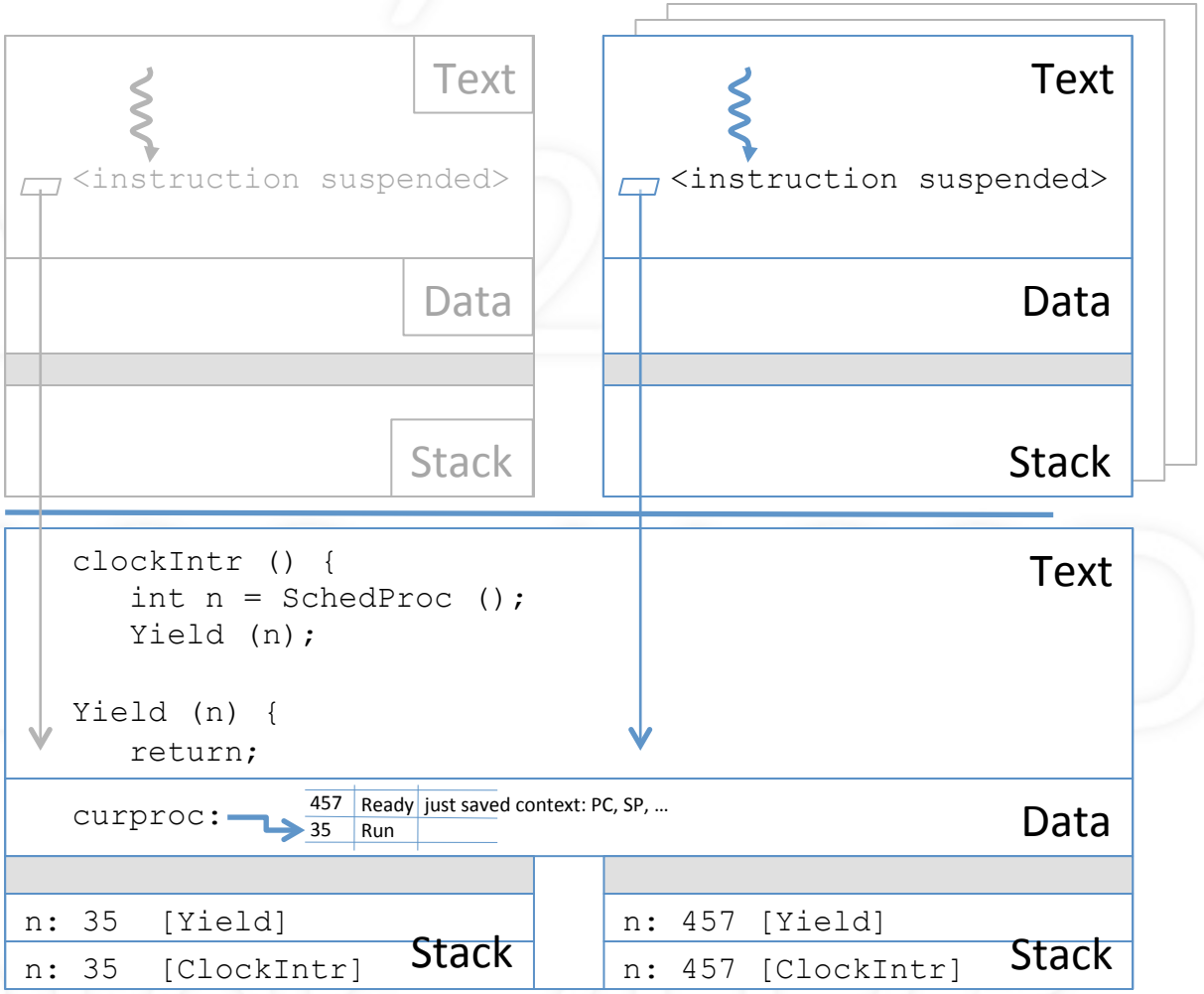
Restore context of next process



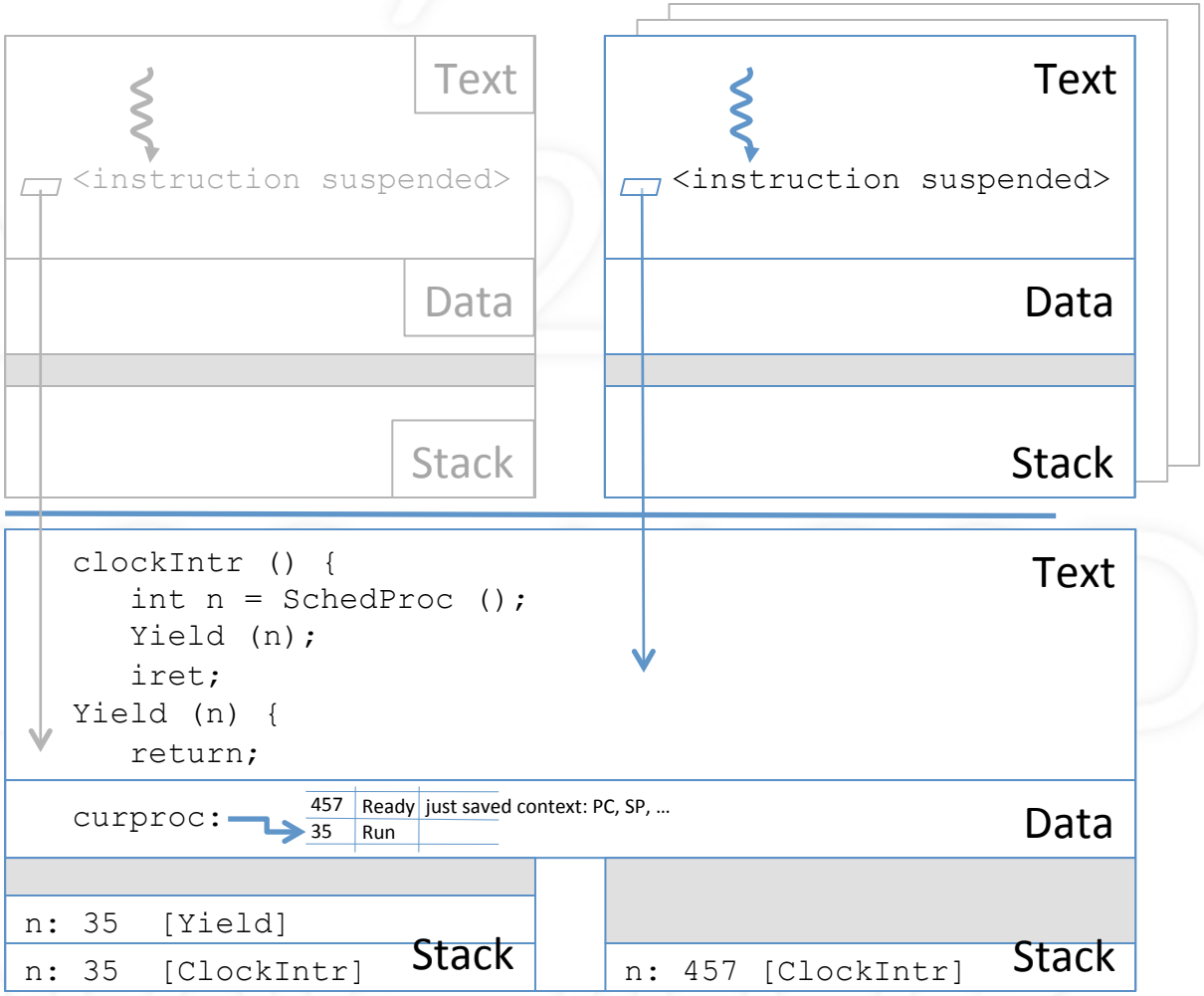
Next process almost running



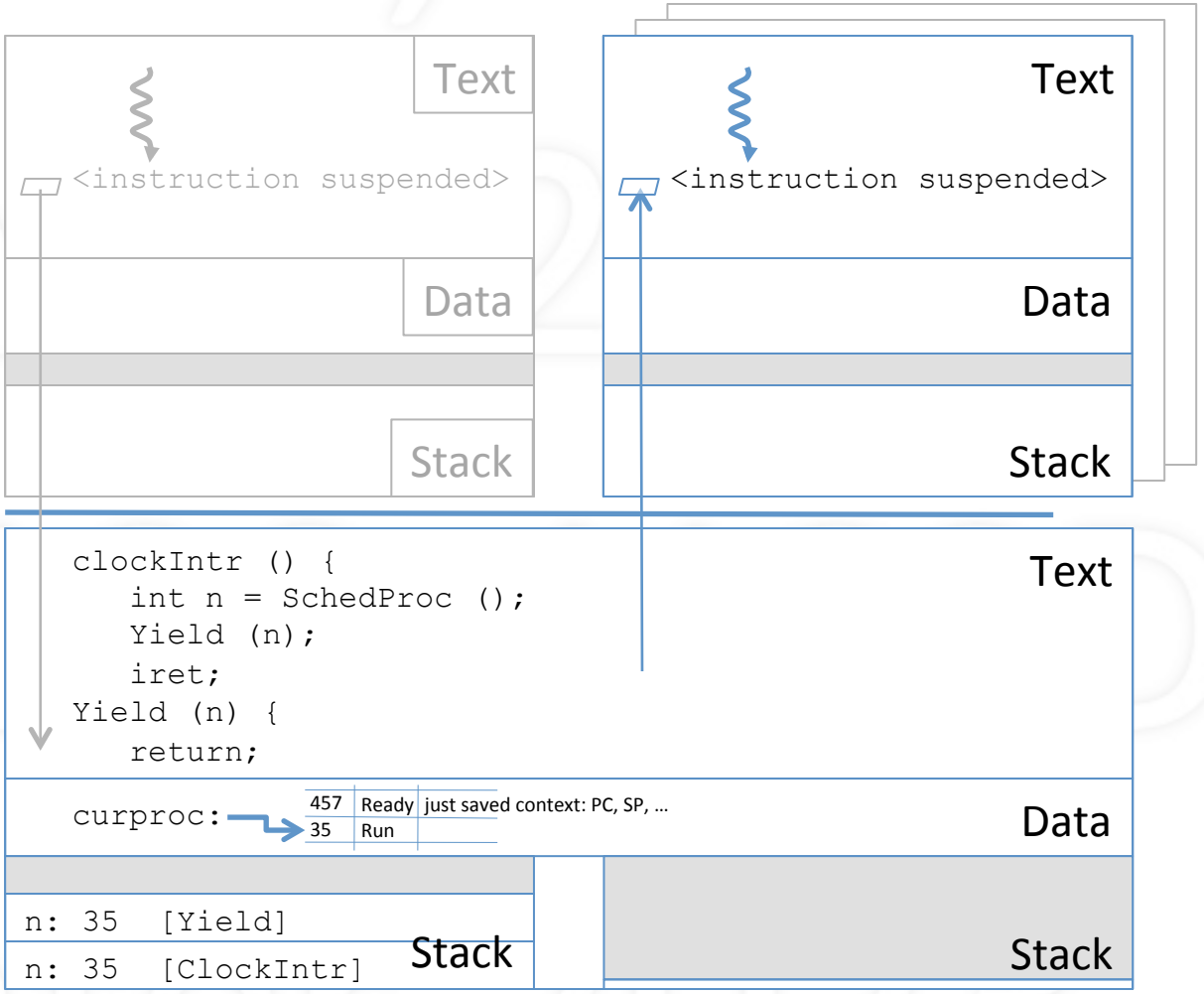
Update process table



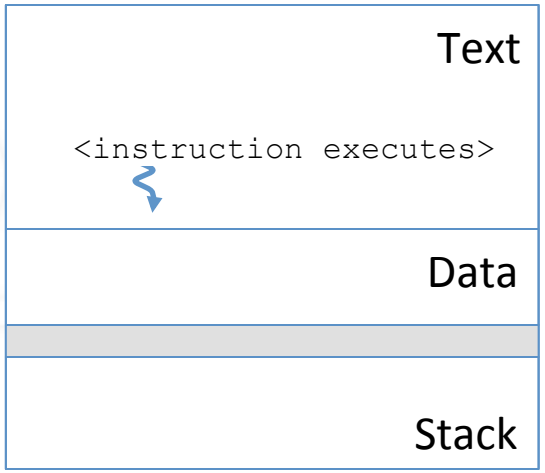
About to return from Yield



Return to process code

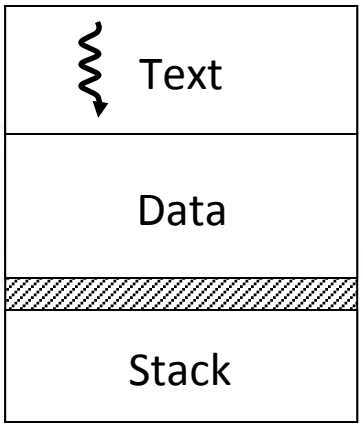


Resumed process now running

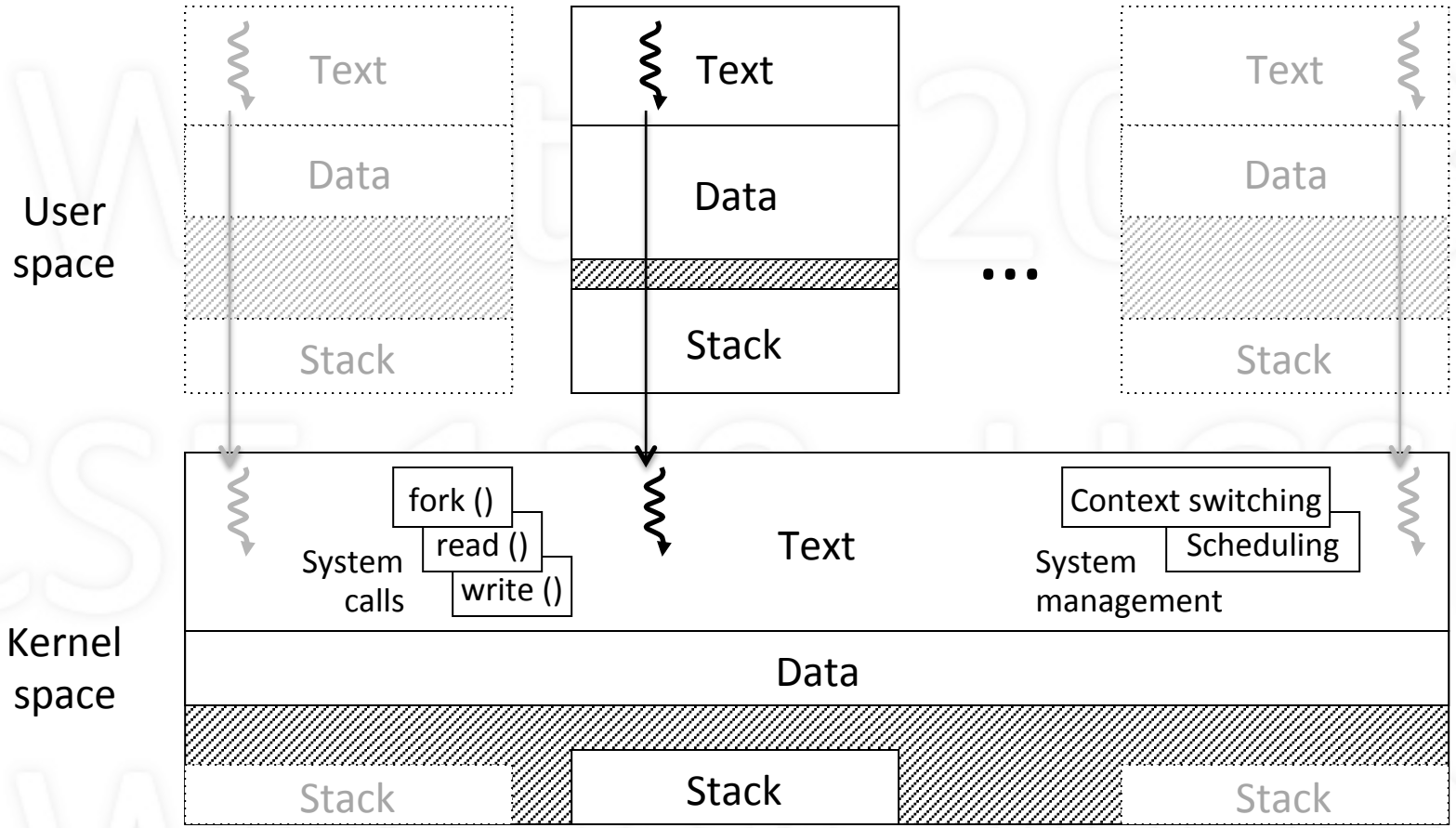


Process Running in User Space

User space



Process Running in Kernel Space



How to Get Parallelism in Process

- Process is a “program in execution”
  - assumed (so far) a single path of execution
  - in a memory composed of text, data, stack
- What if we want multiple paths of execution?
  - Single text, but multiple executions in parallel
  - Single data, any execution can see others’ updates
  - Need separate stacks: one per ongoing execution
- Multiple processes? No (separate memories)

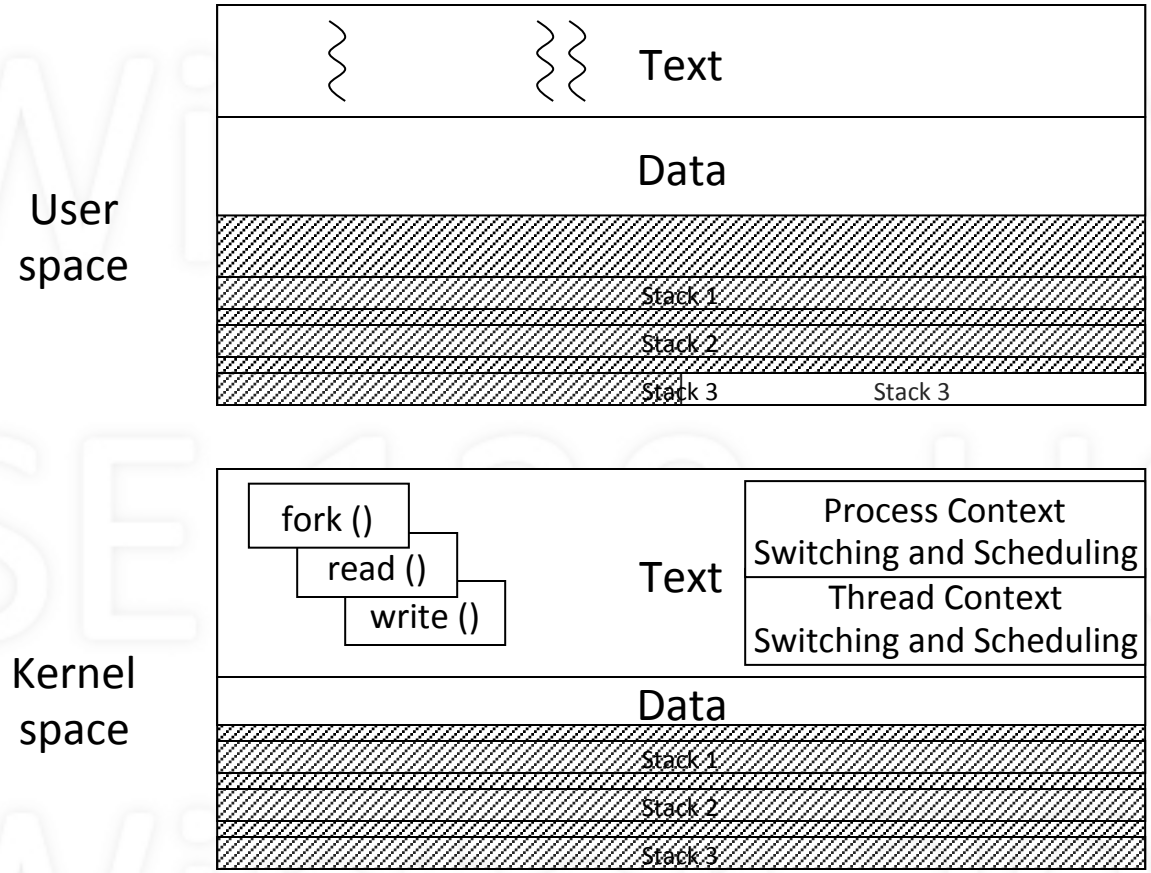
Threads

- Thread: single sequential path of execution
- Abstraction is independent of memory
  - Contrast to process: path of execution + memory
- A thread is part of a process
  - Lives in the memory of a process
  - Distinction allows multiple threads in a process
- To the user: unit of parallelism
- To the kernel: unit of schedulability

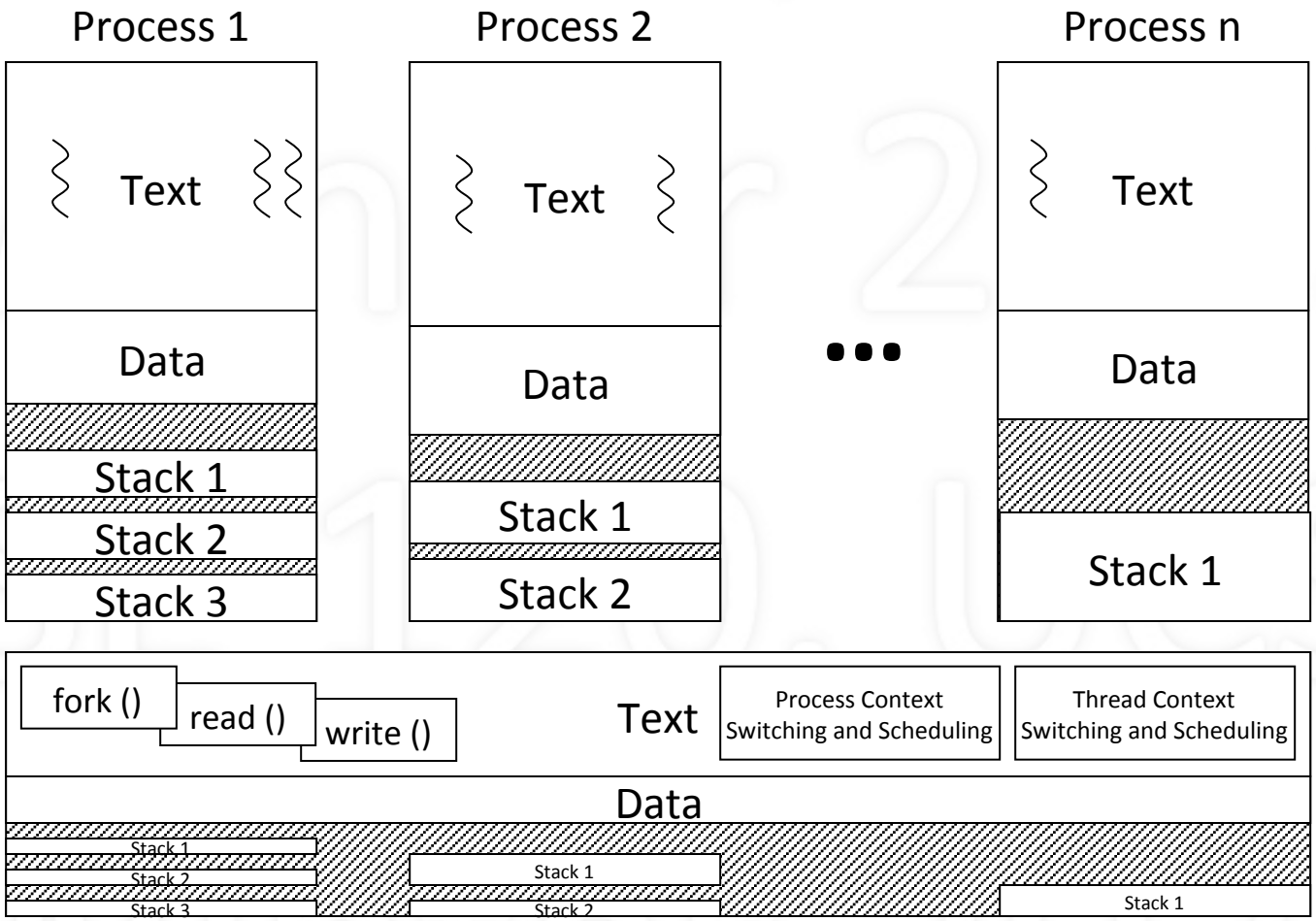
Implementing Threads

- Thread calls are system calls
  - ForkThread (): like process Fork () but for threads
  - Thread system call functions are in kernel
- Thread management functions are in kernel
  - Thread context switching
  - Thread scheduling
- Each thread requires user and kernel stacks
- Kernel can schedule threads on separate CPUs

Single Process, Multiple Threads



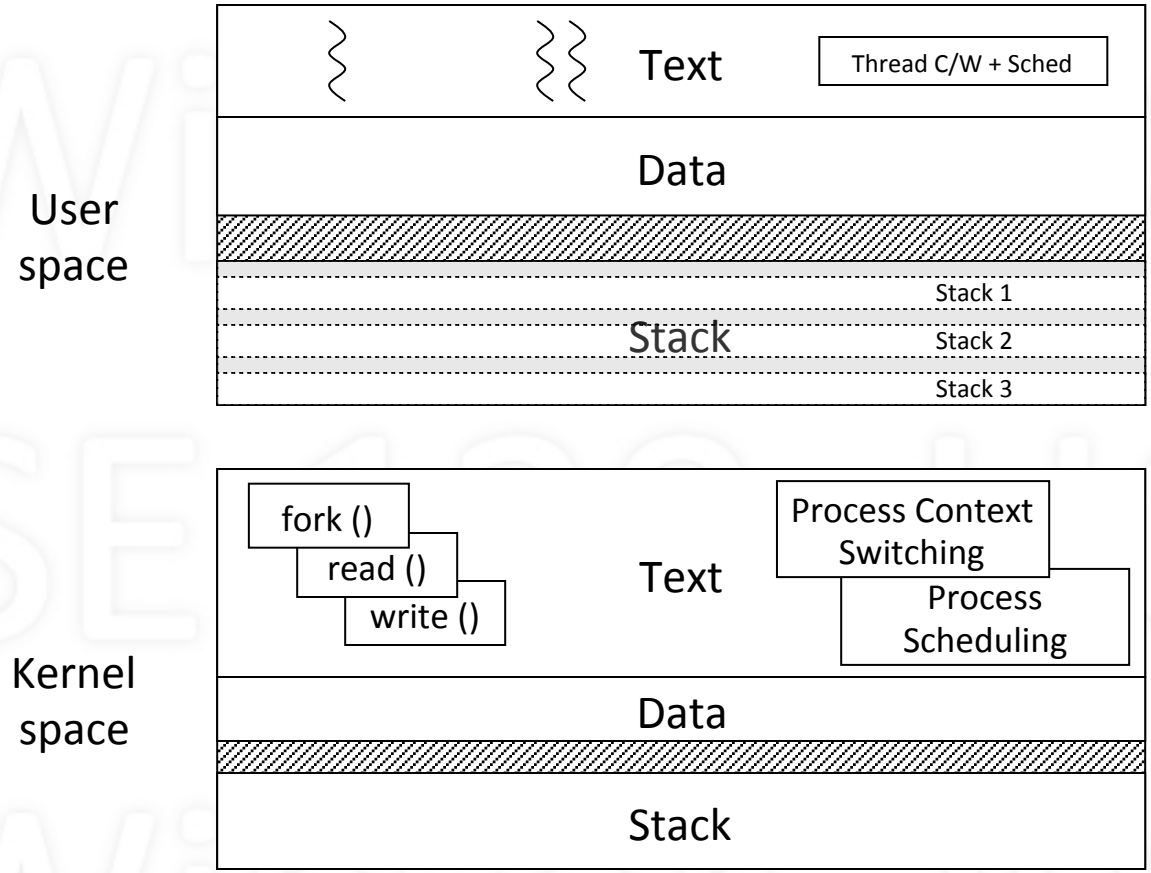
Many Processes with Threads



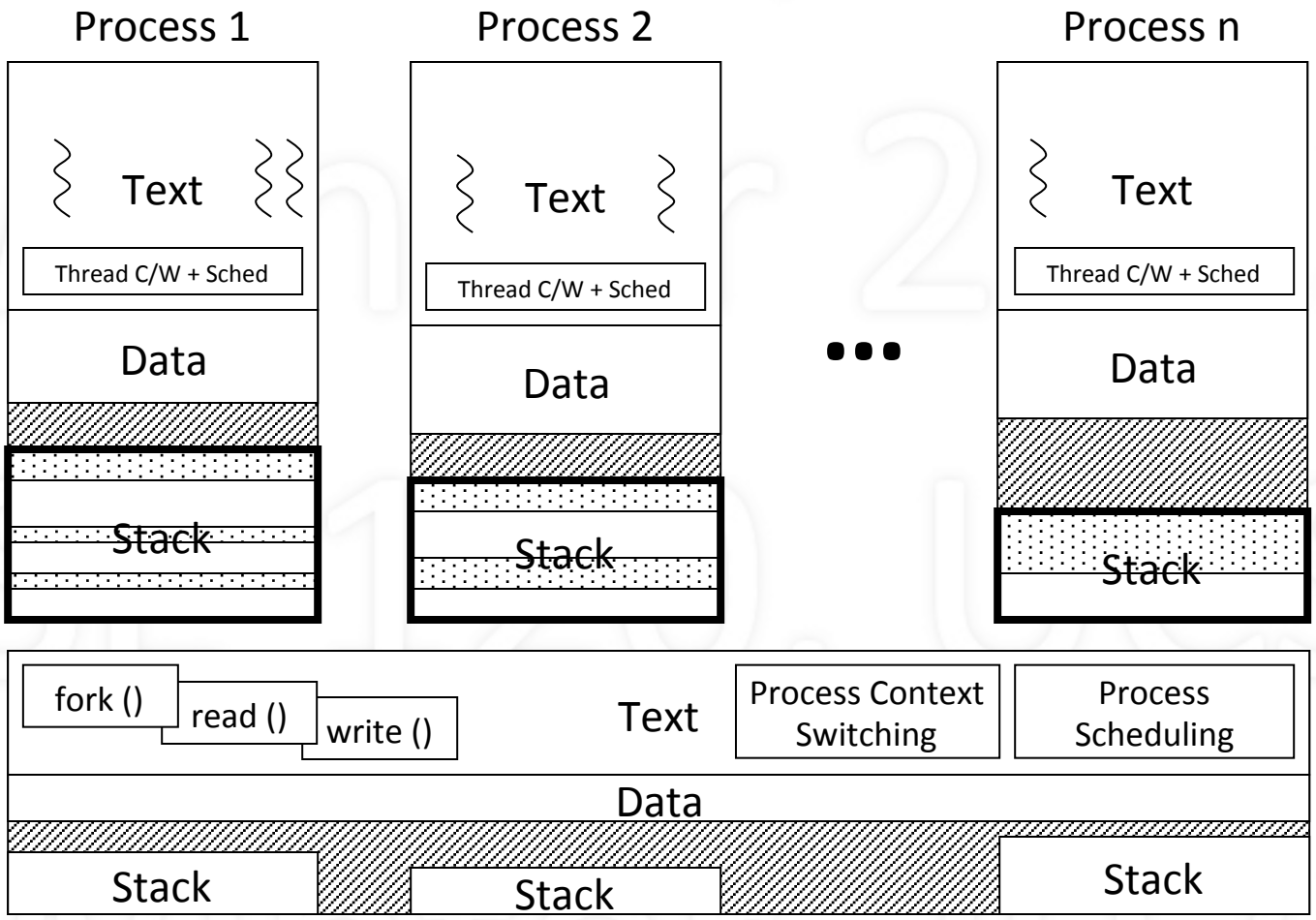
User-Level Threads

- Can support threads at user level
- Included via thread library
- Thread calls at user level
  - ForkThread (), YieldThread (), …
- Thread Management at user level
- Supports threads regardless of kernel support
- However, no true parallelism

User-Level Threads



User-Level Threads



Pros and Cons

- User-level threads
  - Portability: works on any kernel
  - Efficient: thread-switching occurs in user space
  - User can decide on scheduling policy
  - But no true parallelism (without special support)
- Kernel-level threads
  - Can achieve true parallelism
  - Overhead: thread switch requires kernel call

Thread Support vs. Execution

- Distinguish between
  - Where is thread abstraction supported?
  - Where is thread executing?
- User-level vs. kernel-level threads
  - Is thread *support part of* user or kernel code?
- Running in user space vs. kernel space
  - Is thread *running in* user or kernel space?
- Make sure you understand the distinction!

Summary

- Timesharing: rapidly switching the CPU
  - creates illusion of parallel progress of processes
- Process states: running, ready, blocked
  - distinguishes logical vs. physical execution
- Kernel: extension of all processes
  - gets control by process yielding or preemption
- Thread: single sequential path of execution
  - kernel threads vs. user threads

Textbook

- OSP: Chapter 3
- OSC: Chapters 3, 4, 12
  - Lecture-related: 3.1-3.3, 3.9, 4.1, 4.3, 4.8, 12.3.3
  - Recommended: 4.2, 4.4-4.7