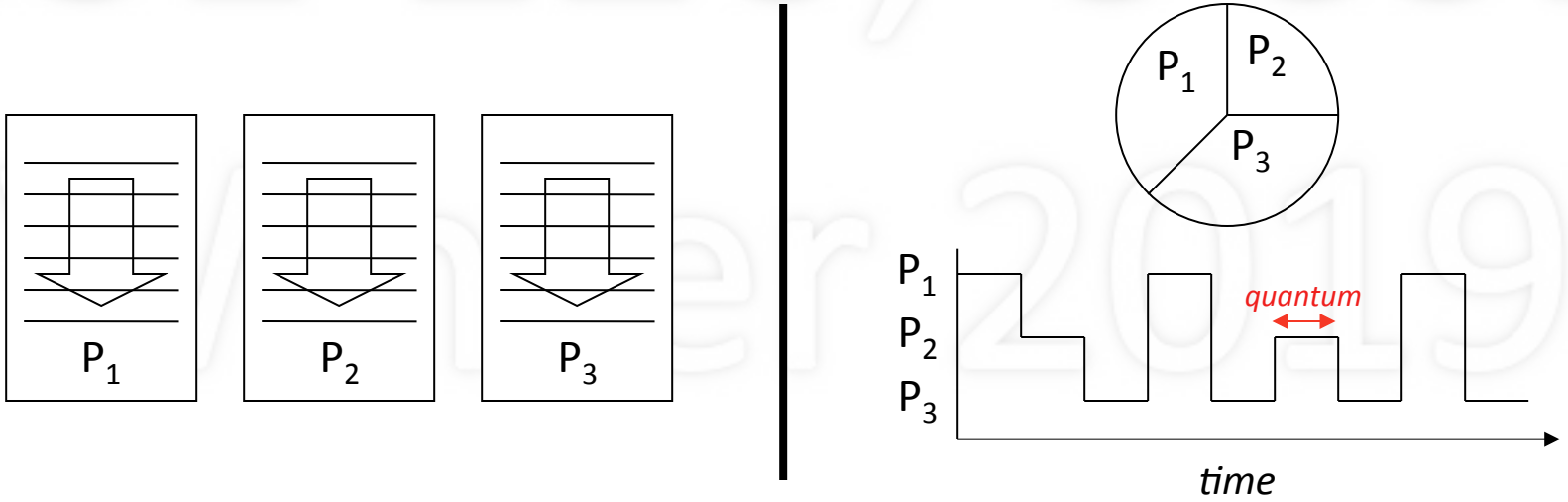
CSE 120: Principles of Operating Systems
Lecture 3: Timesharing

Prof. Joseph Pasquale
University of California, San Diego

January 16, 2018

Timesharing

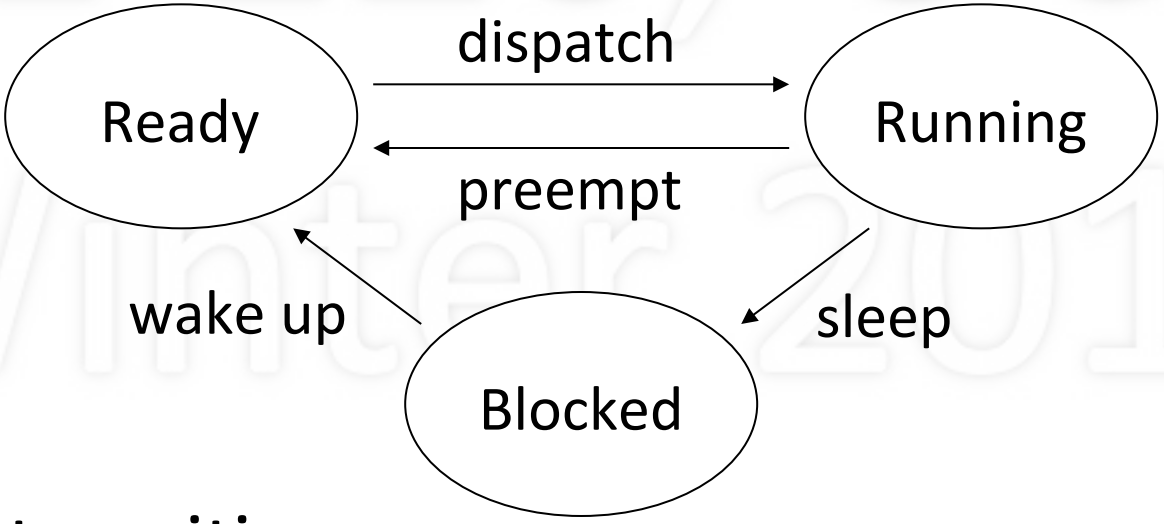


- Timesharing: multiplexing use of CPU over time
- Multiple processes, single CPU (uniprocessor)
- Conceptually, each process makes progress over time
- In reality, each periodically gets quantum of CPU time
- Illusion of parallel progress by rapidly switching CPU

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

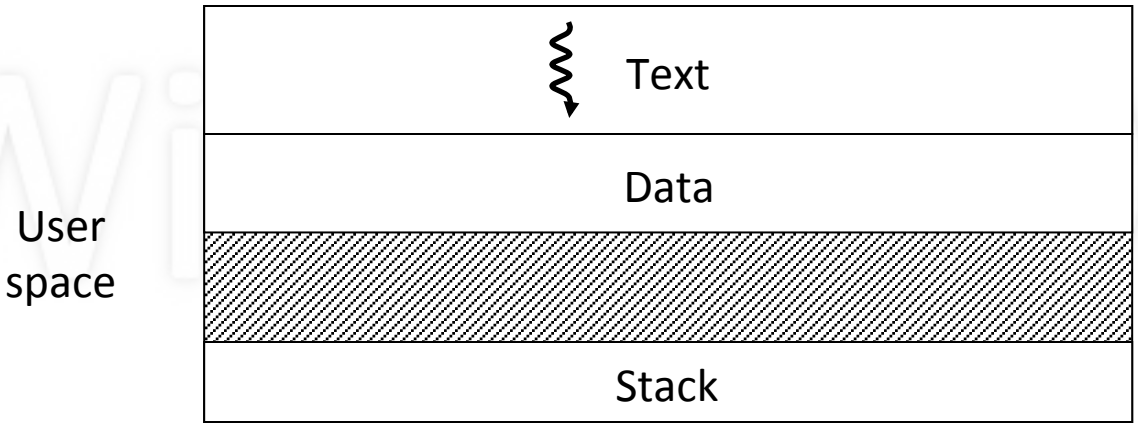
Logical vs. Physical Execution

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

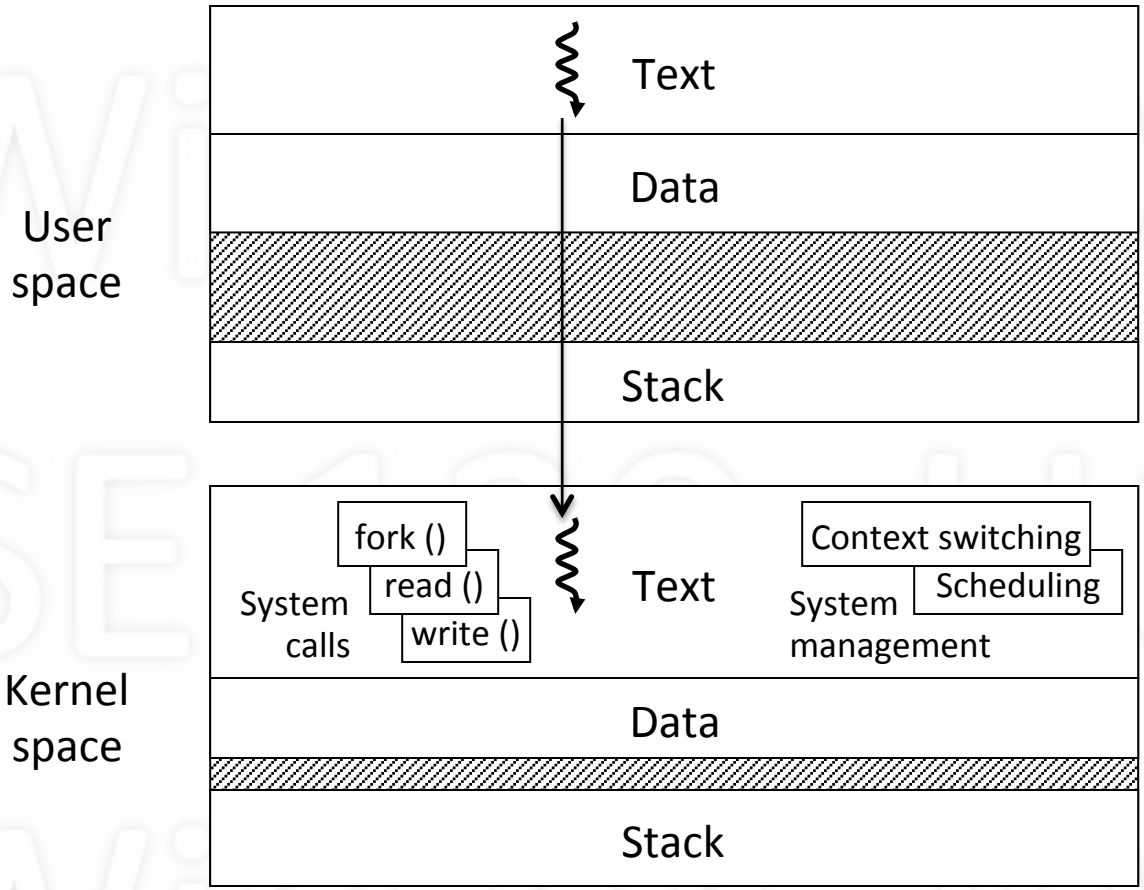
Process vs. Kernel

- 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

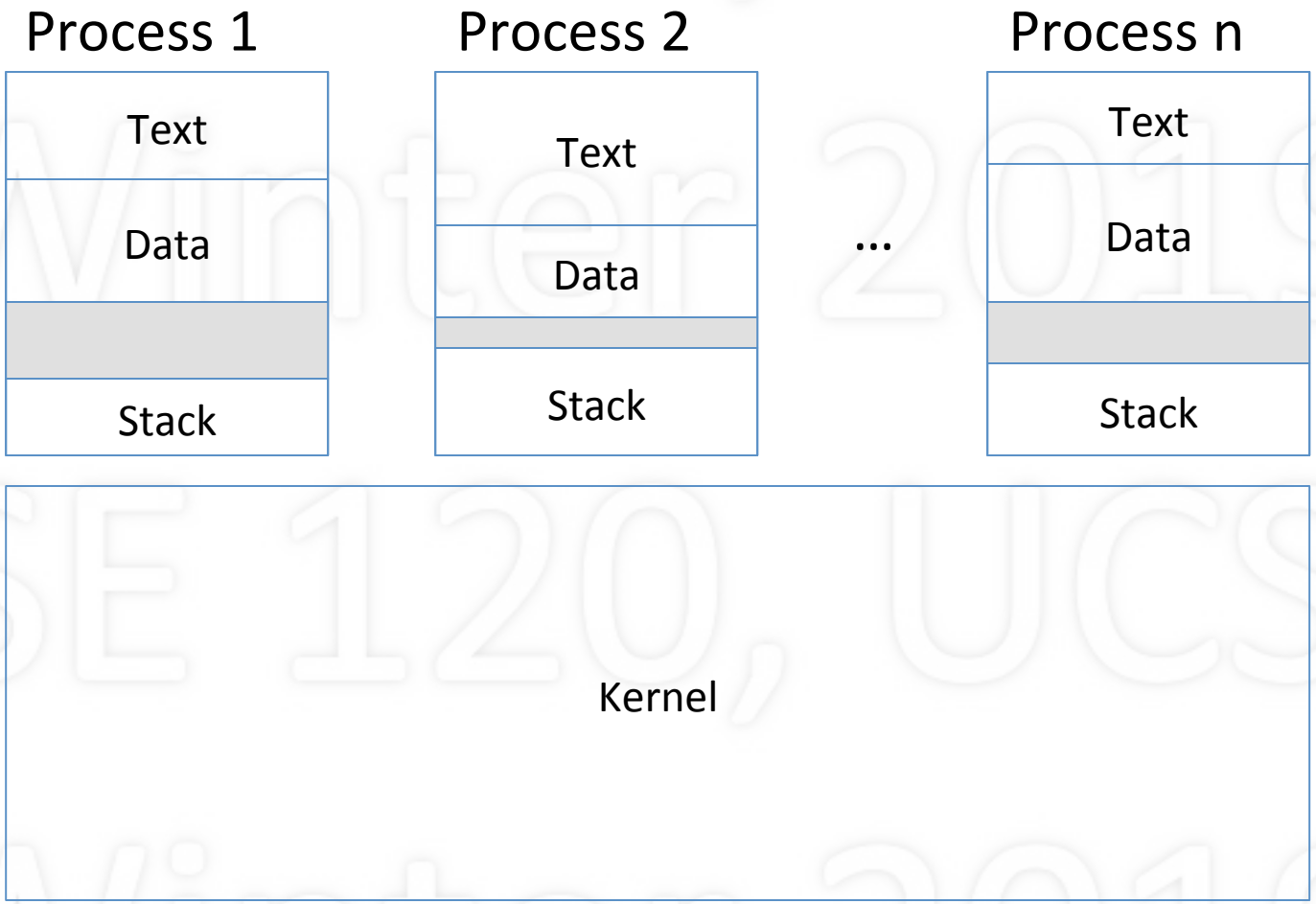
How Does Kernel Get Control

- 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

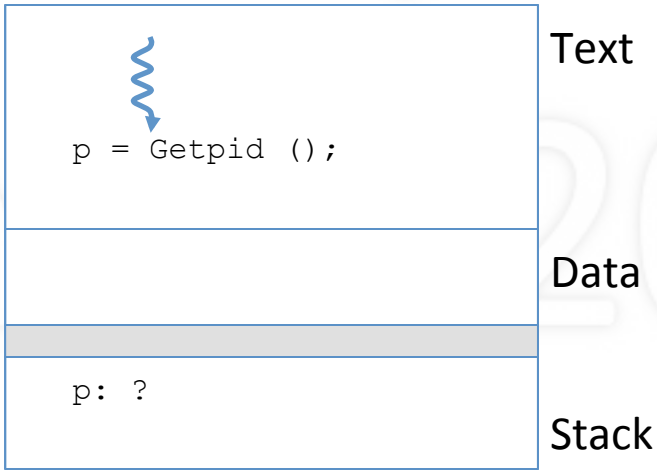
How a Context Switch Occurs

- 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

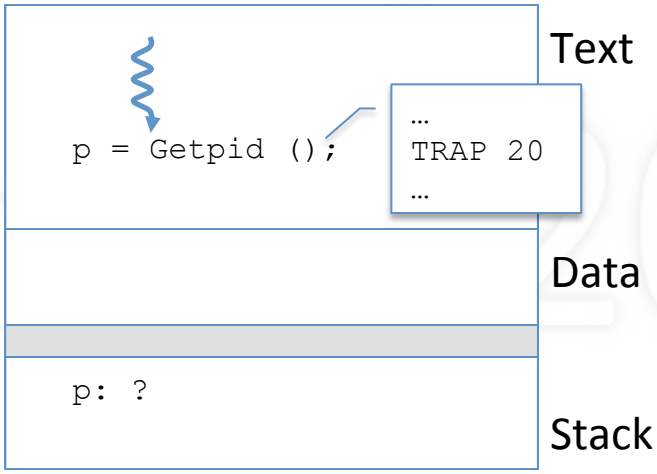
Example



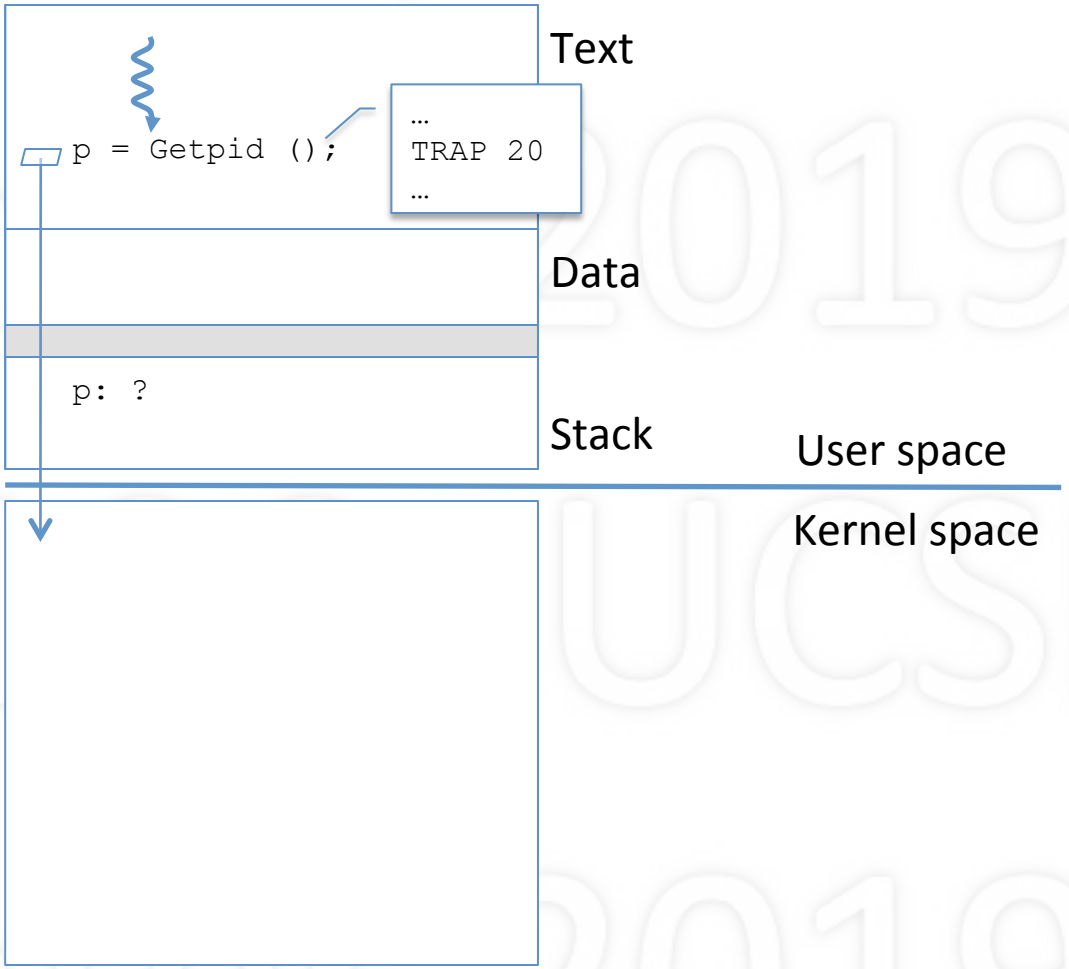
Process makes system call



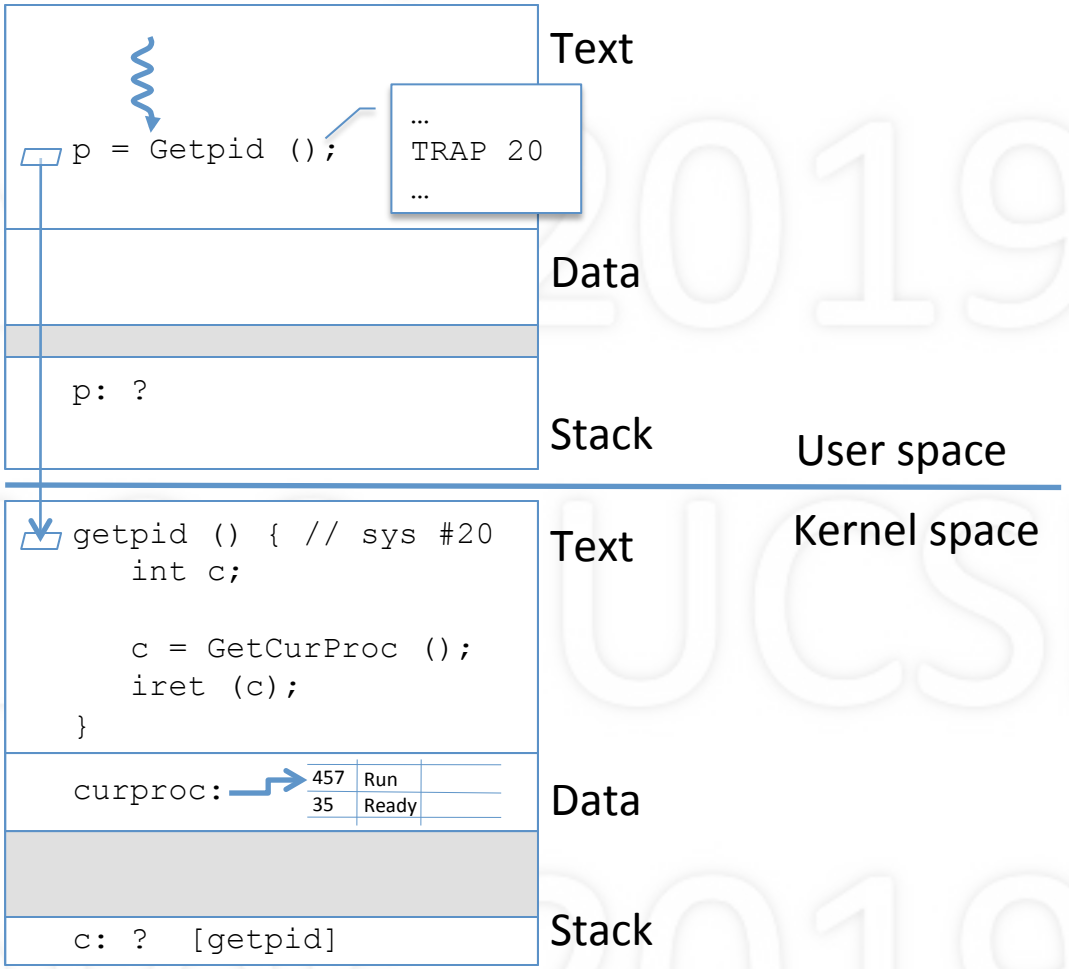
Translates into a TRAP



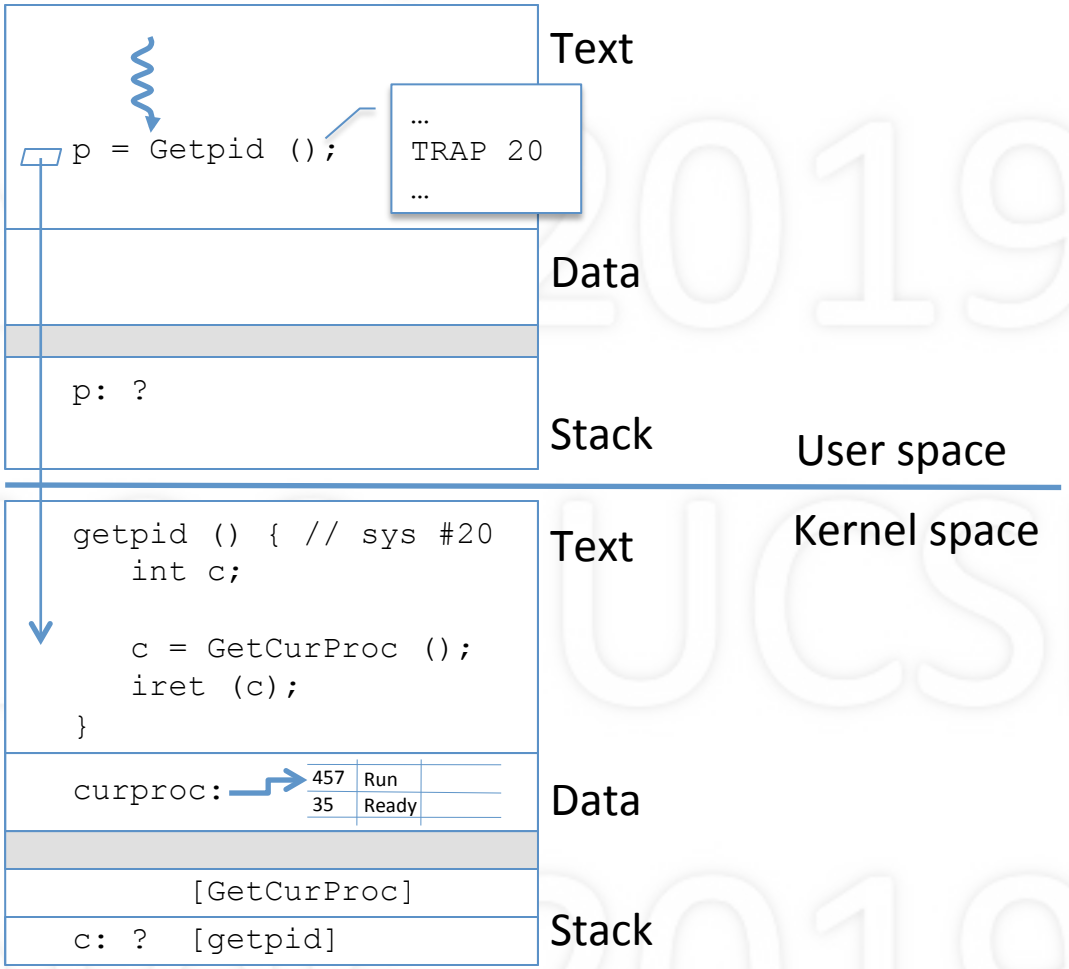
TRAP causes entry into kernel



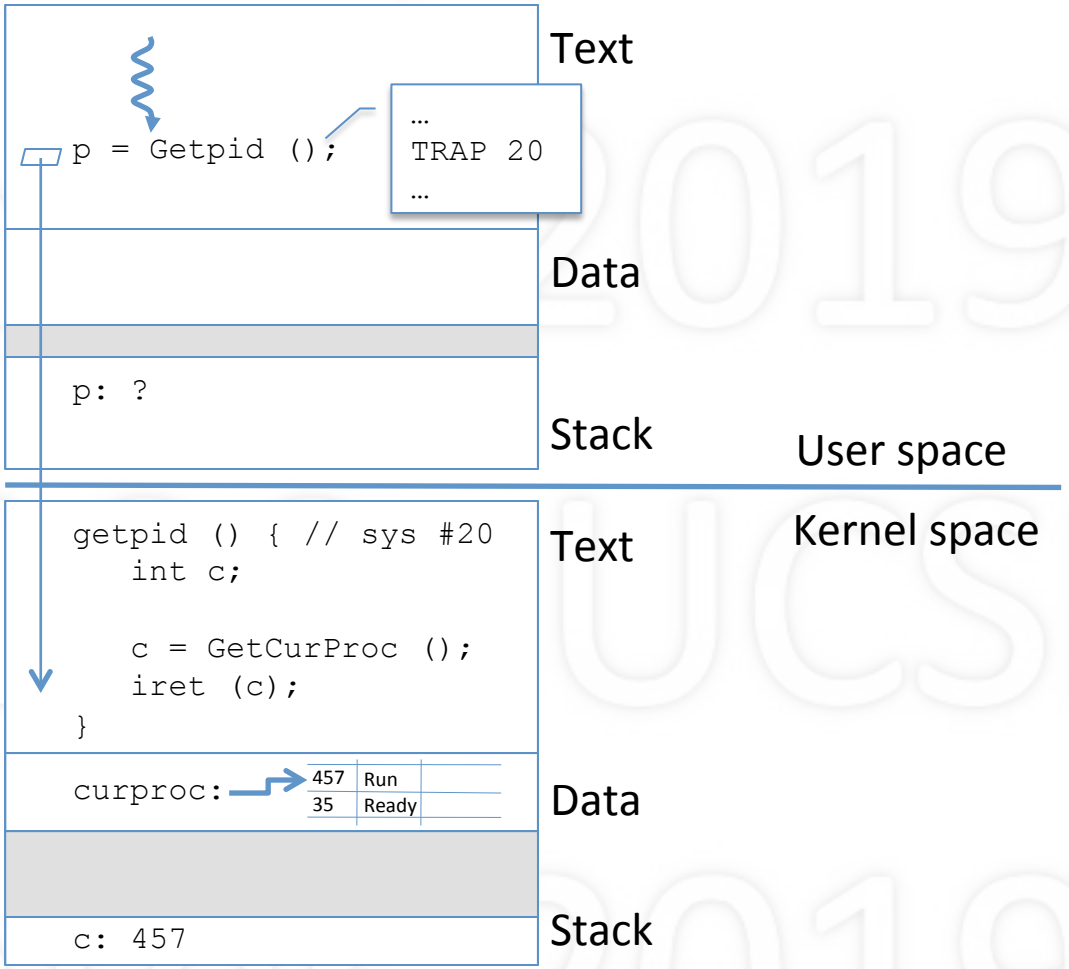
Jump to proper system function



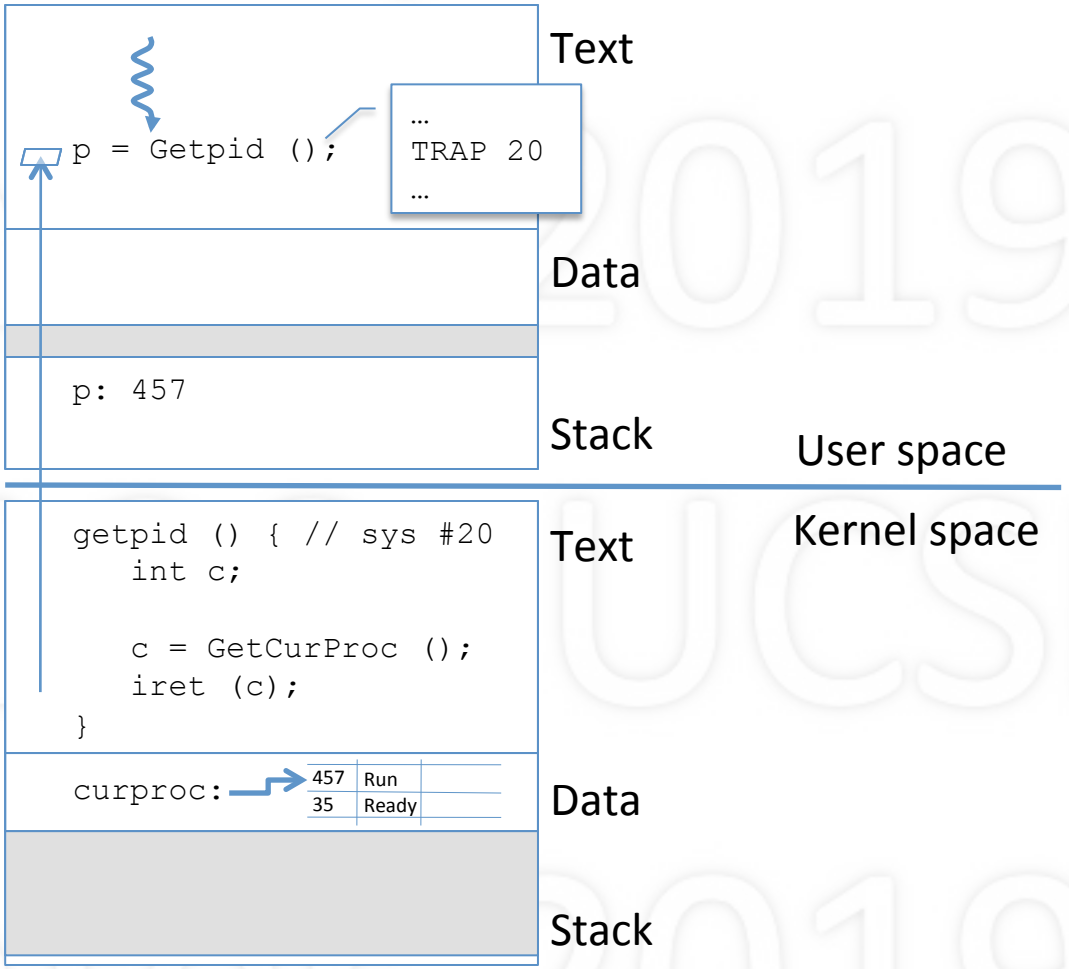
System function starts executing



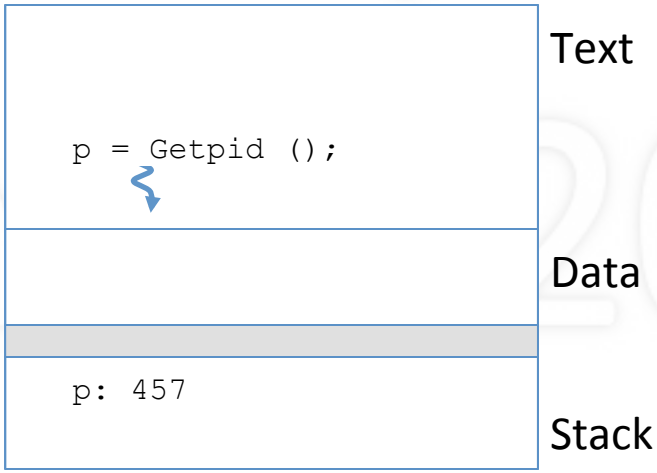
System function completes



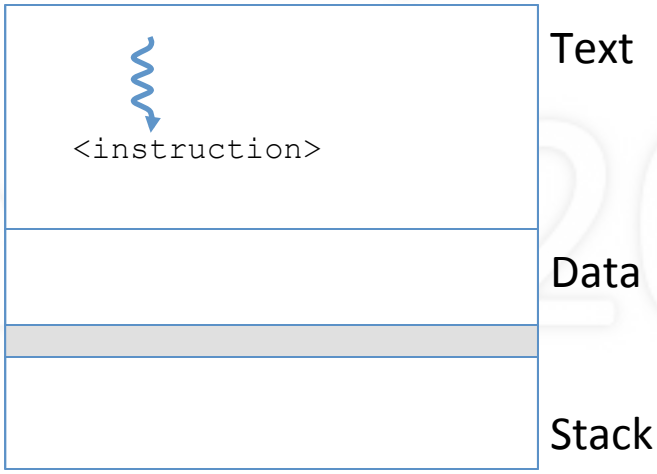
Return from kernel to process



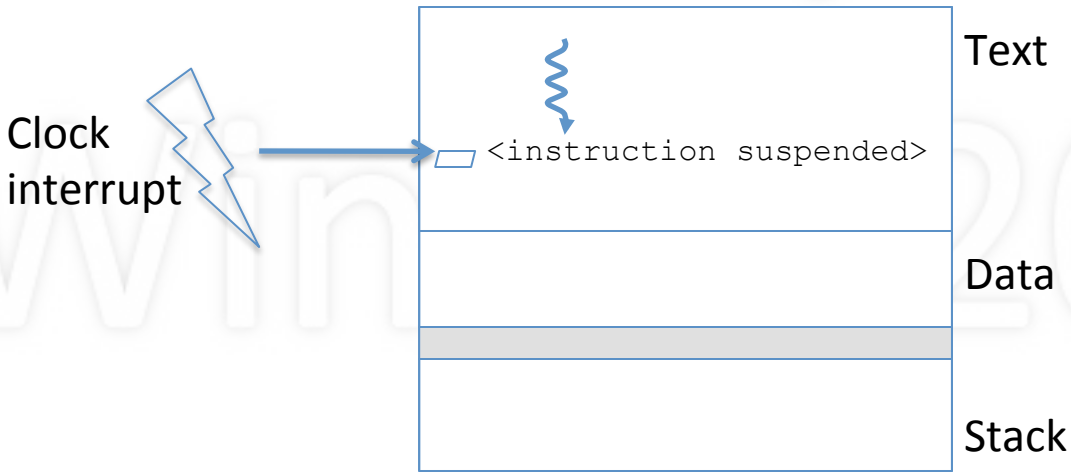
Process continues after system call



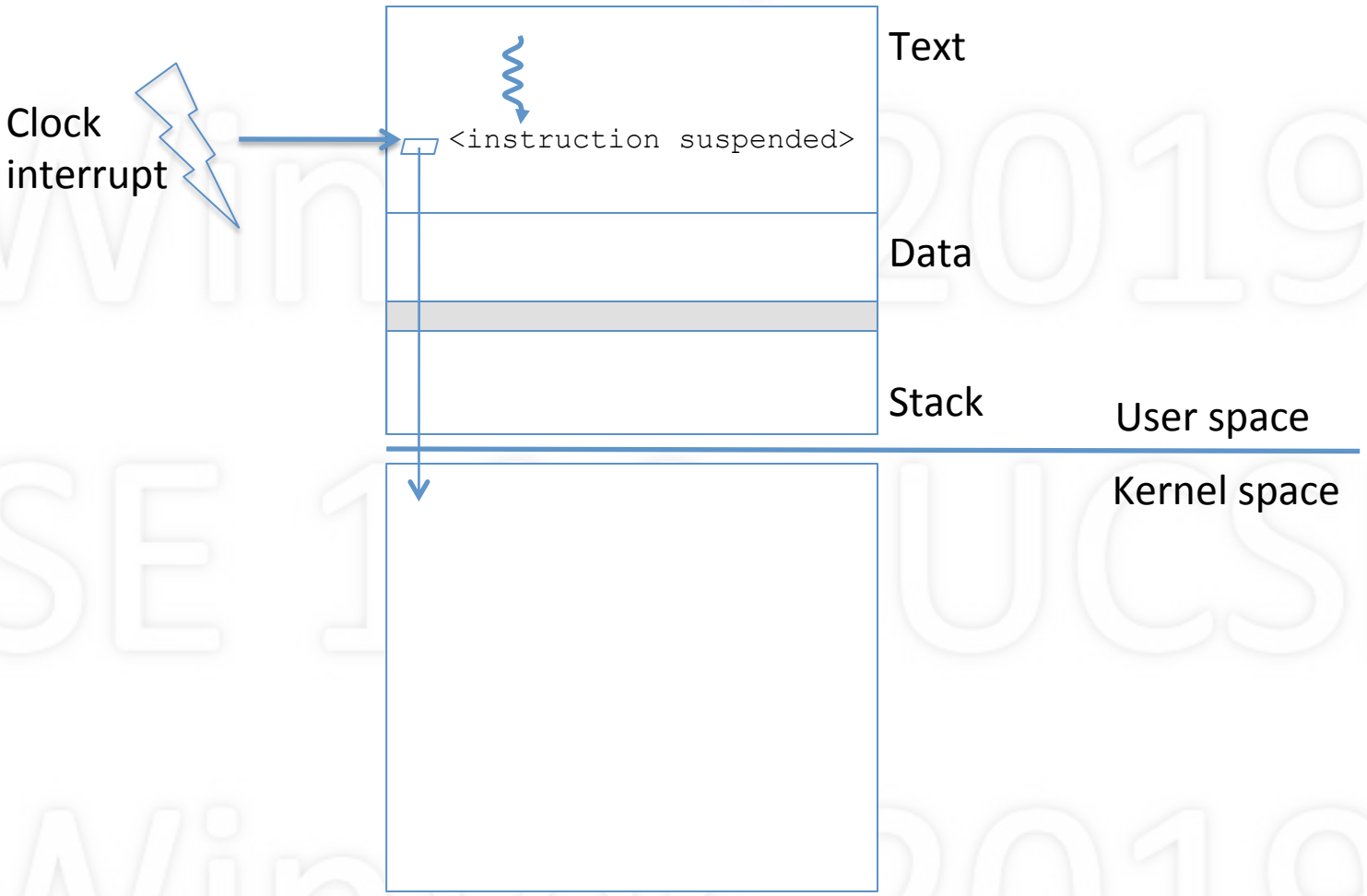
Kernel entry via clock interrupt



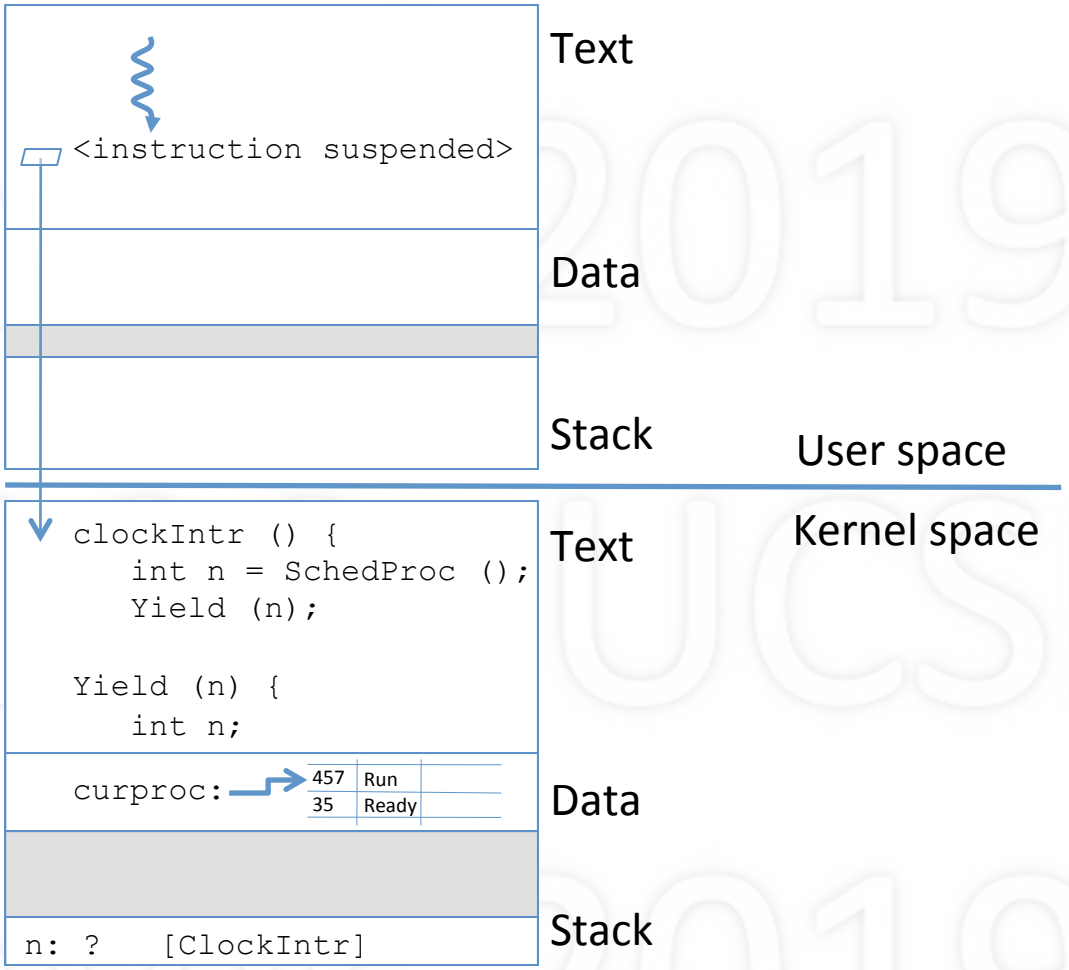
Asynchronous clock interrupt



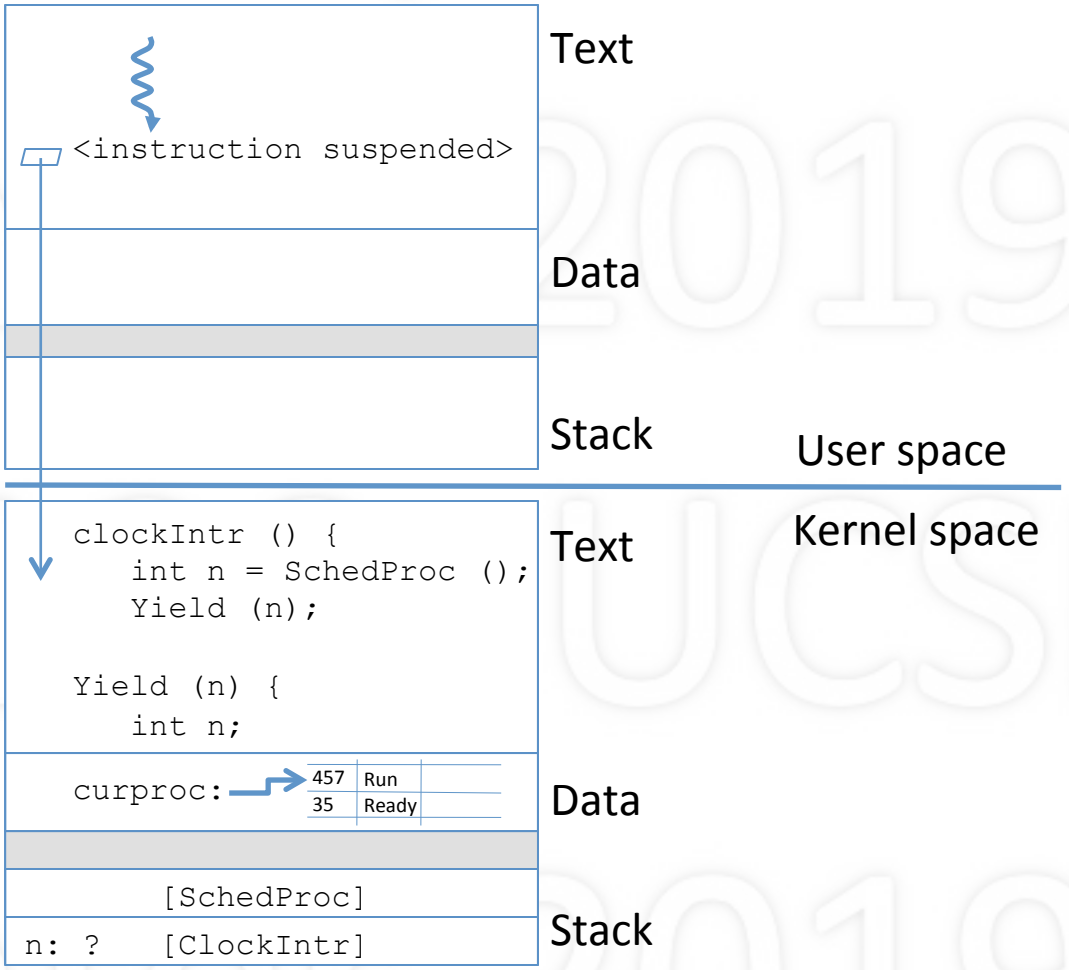
Causes forced kernel entry



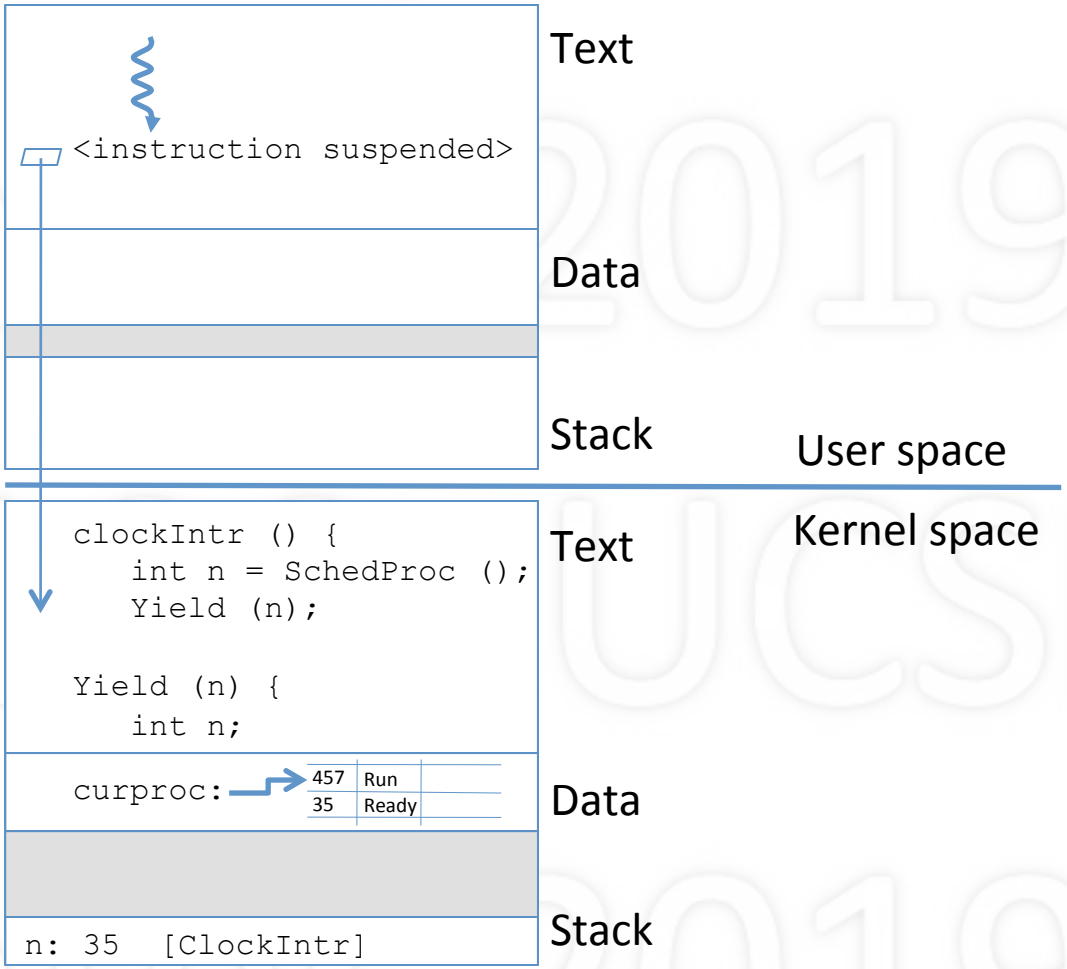
Clock interrupt handler executes



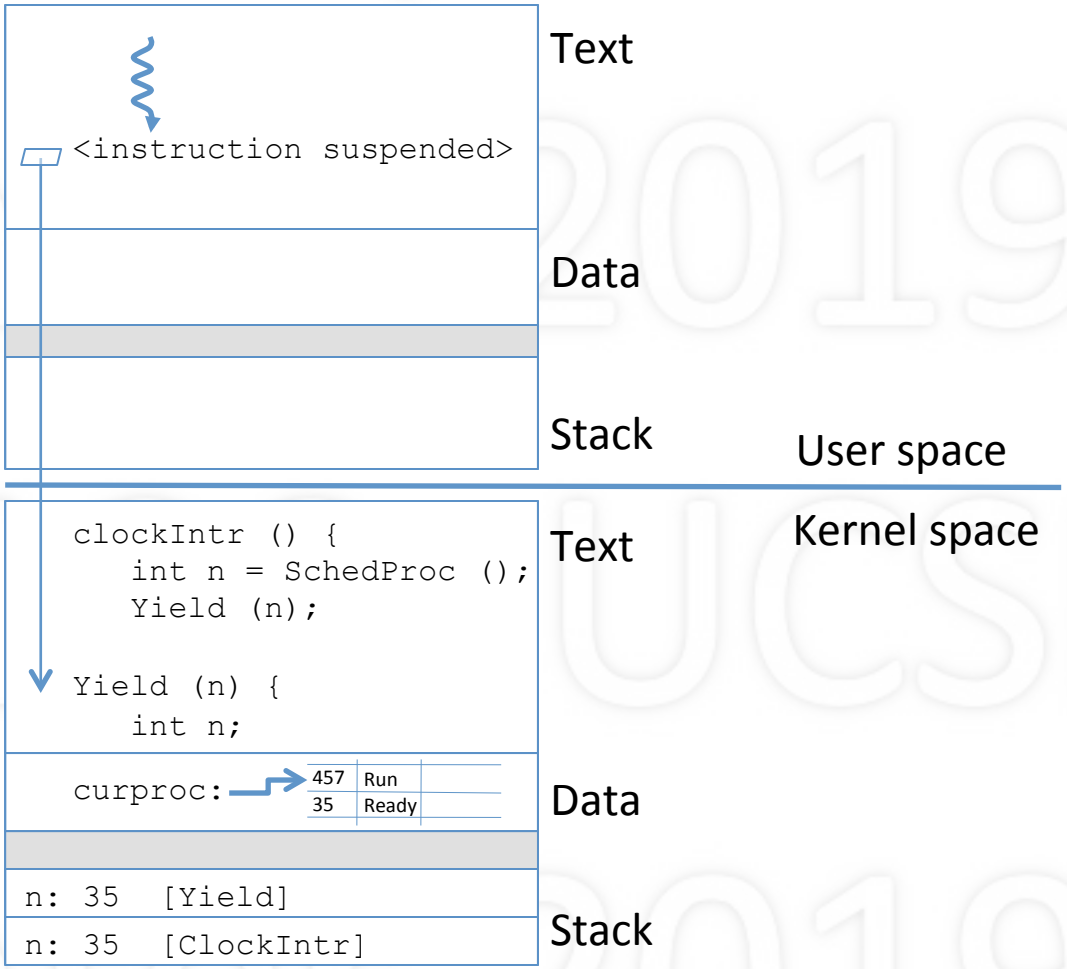
Decide which process to run next



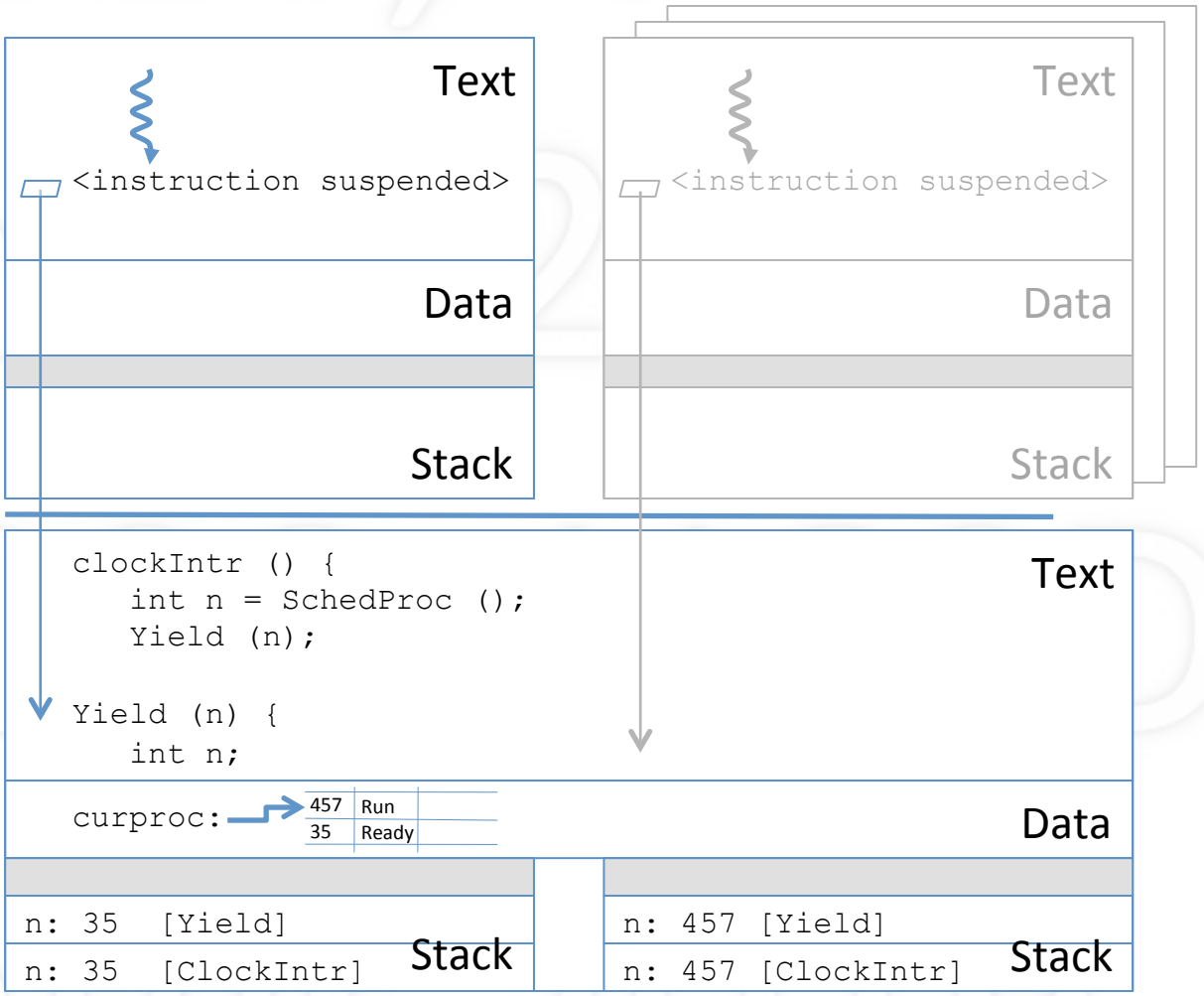
Yield to that process



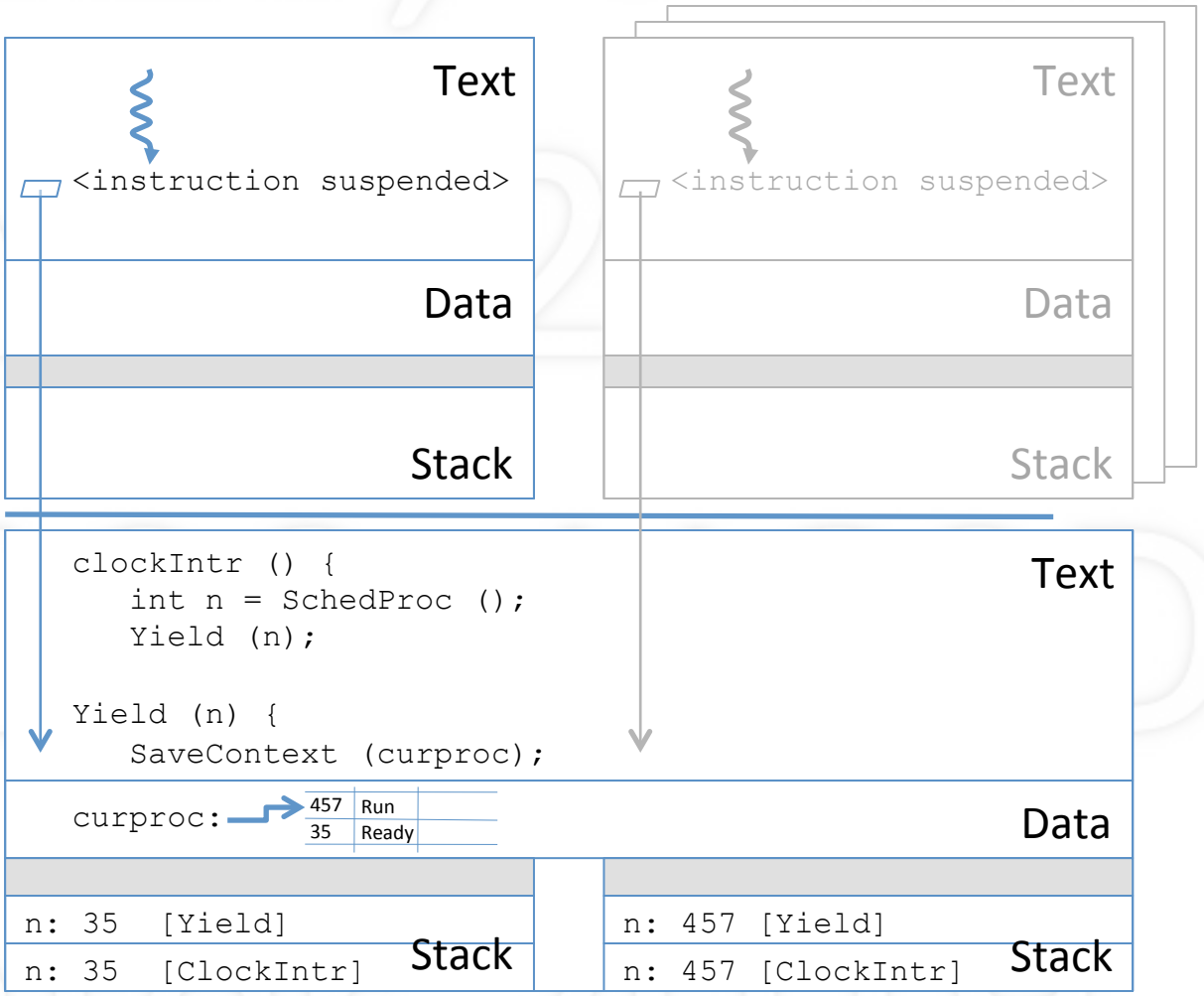
Call to yield



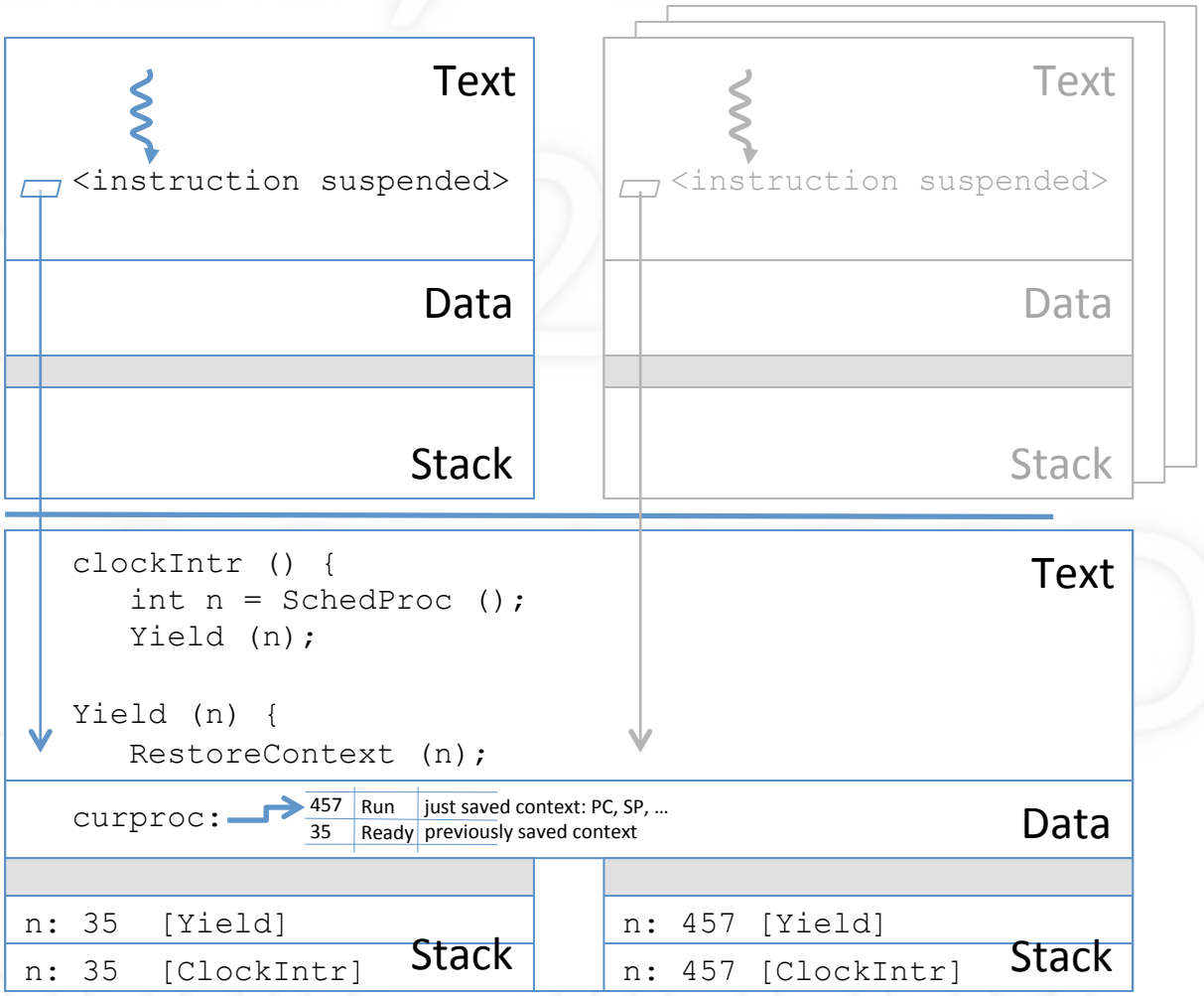
Yield magic



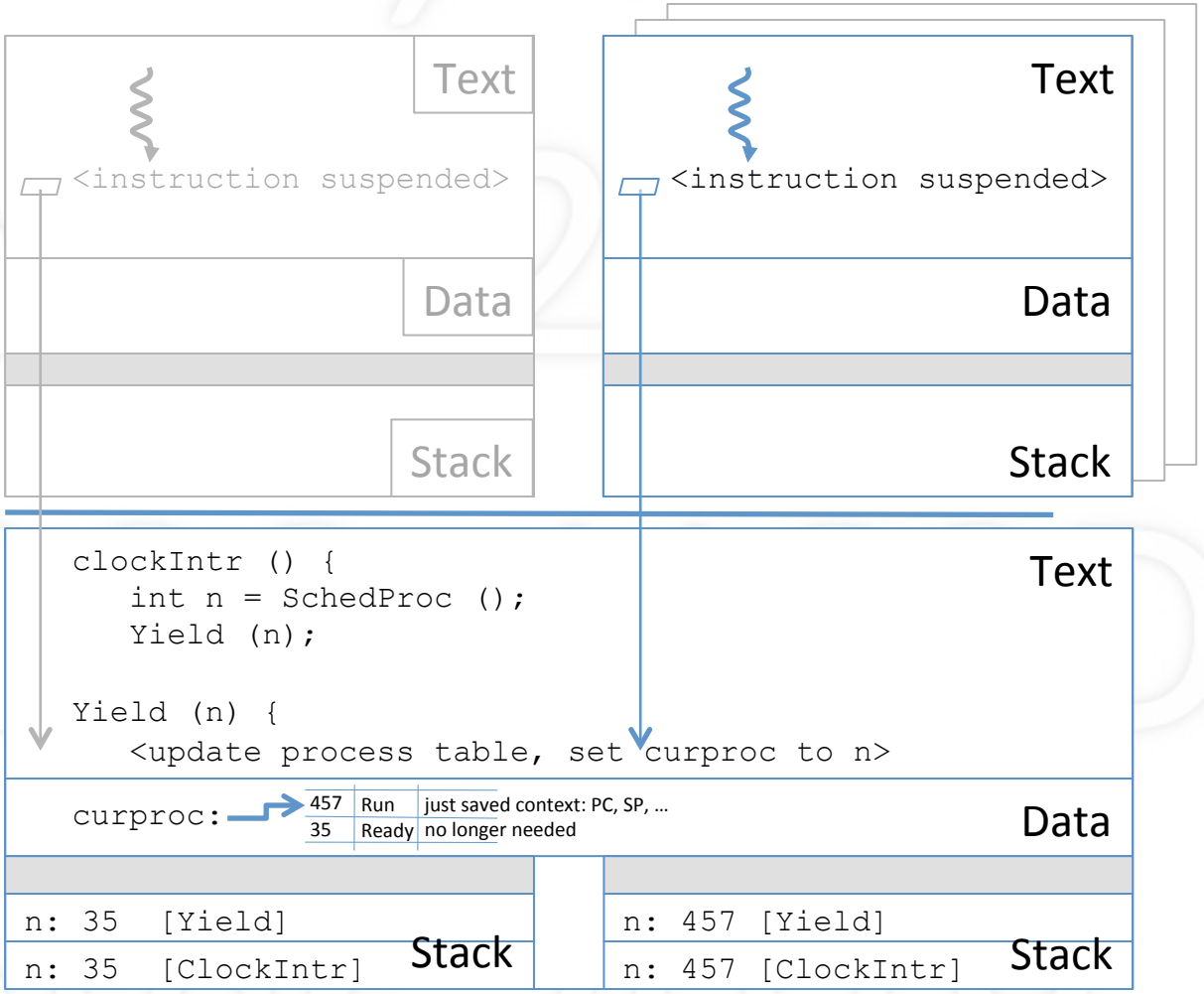
Save context



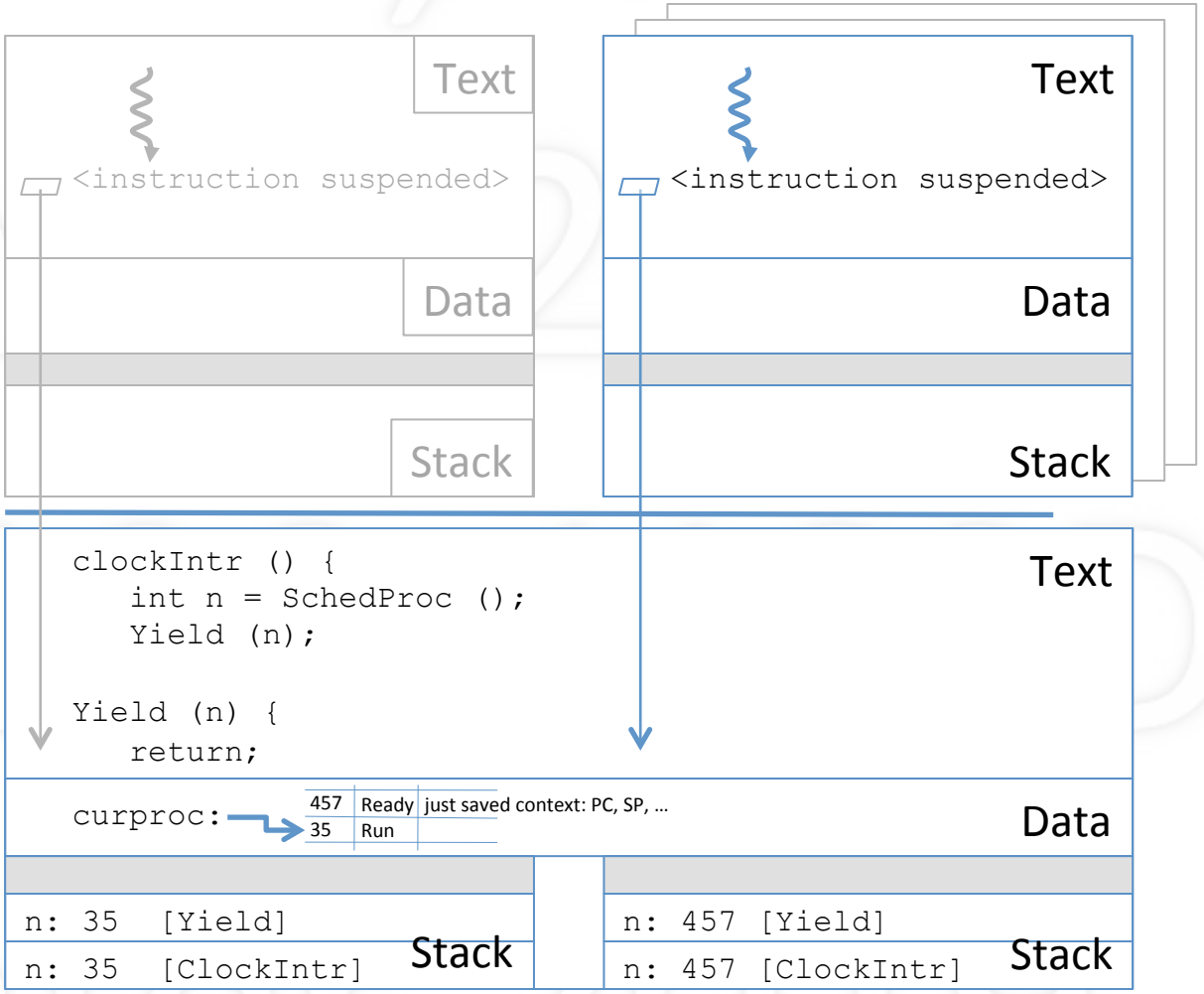
Restore context of next process



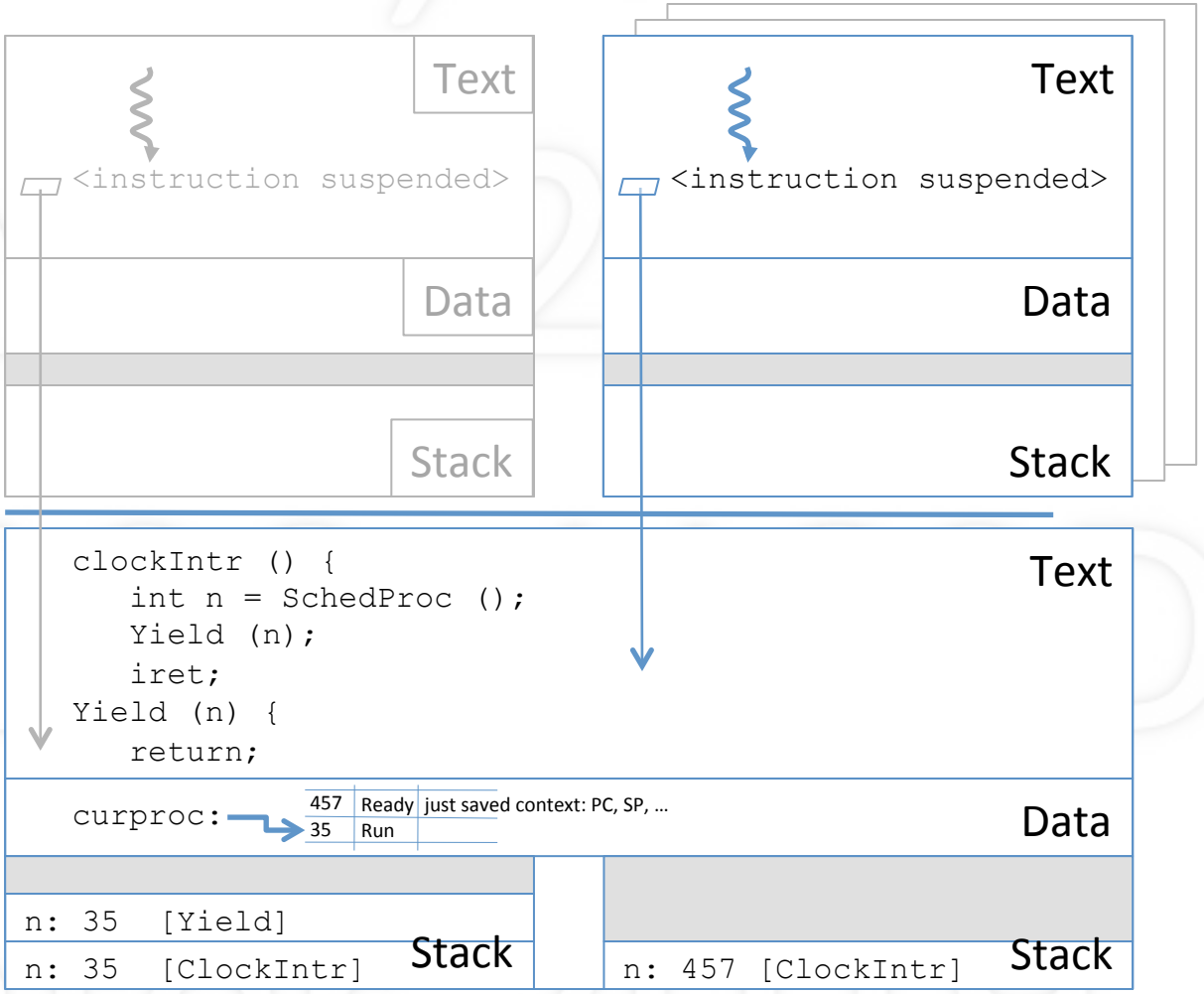
Next process almost running



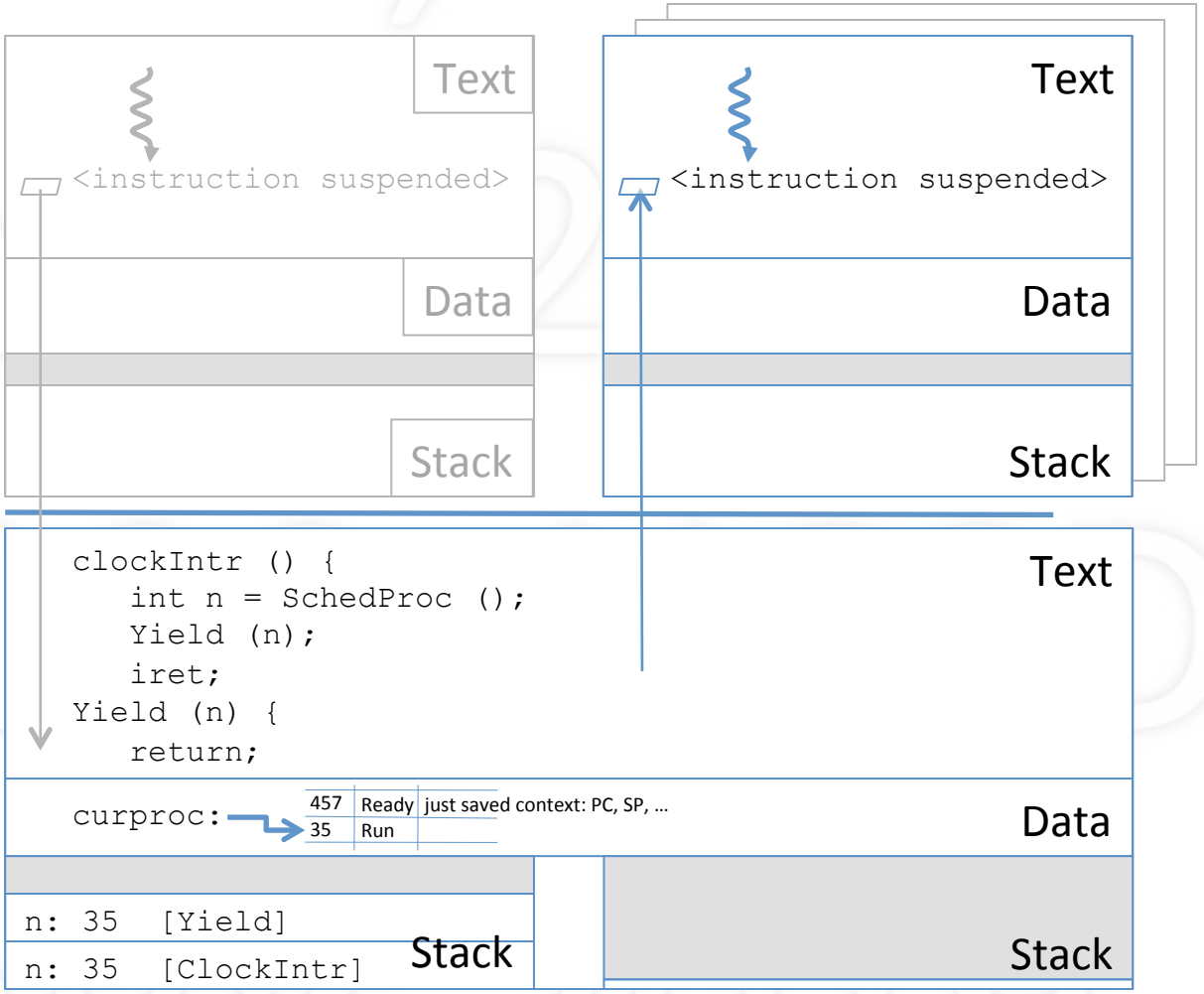
Update process table



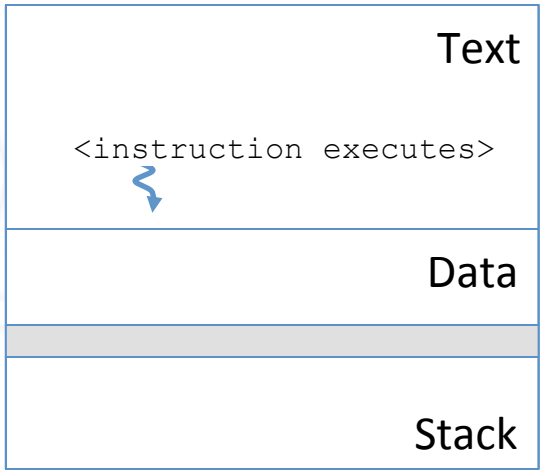
About to return from Yield



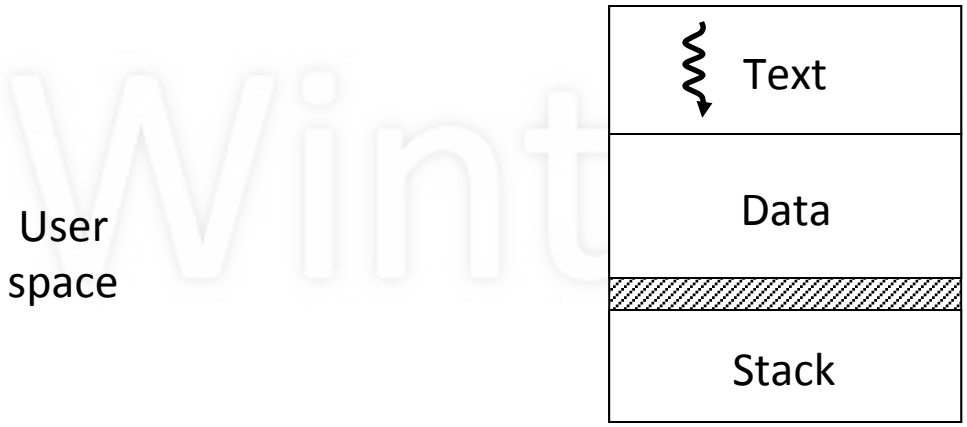
Return to process code



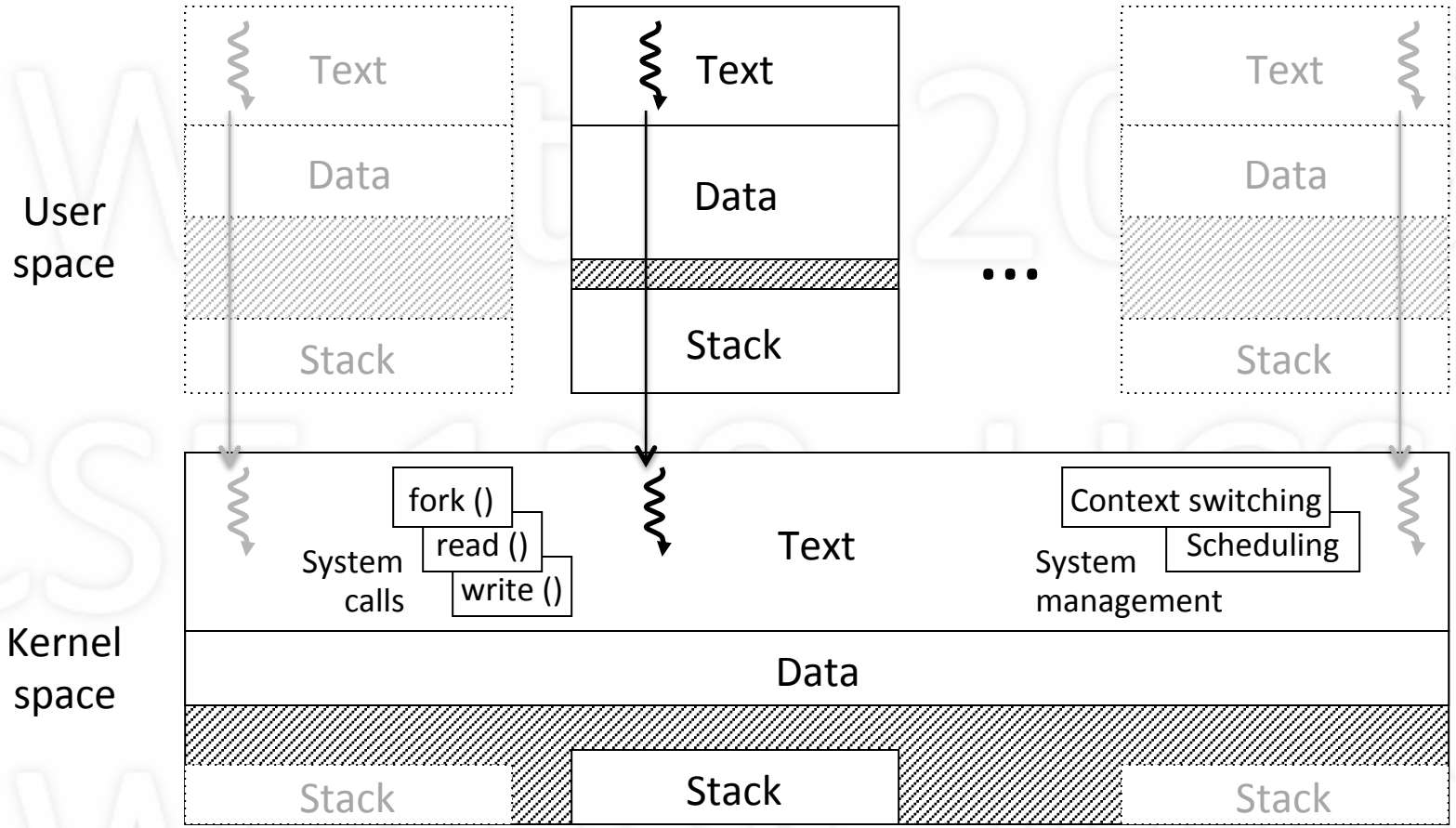
Resumed process now running



Process Running in User Space



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)

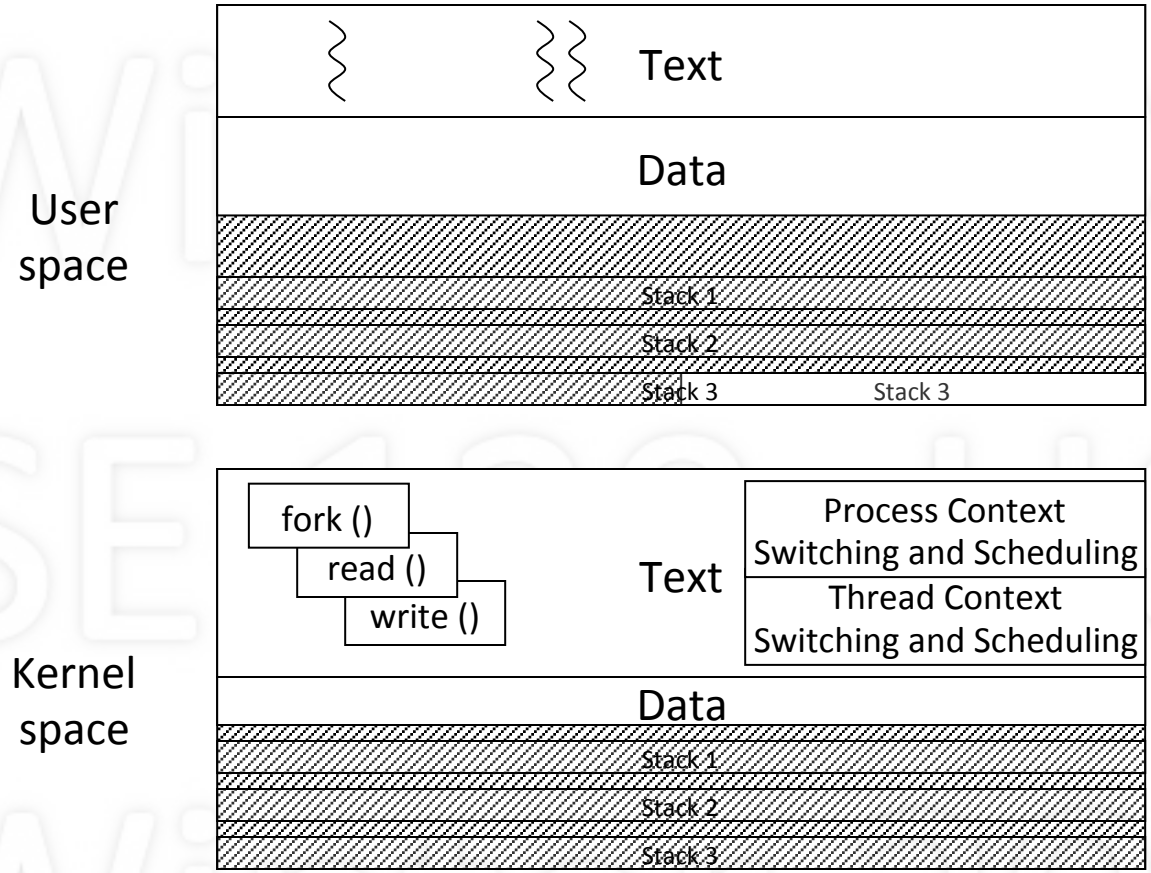
Threads

- 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

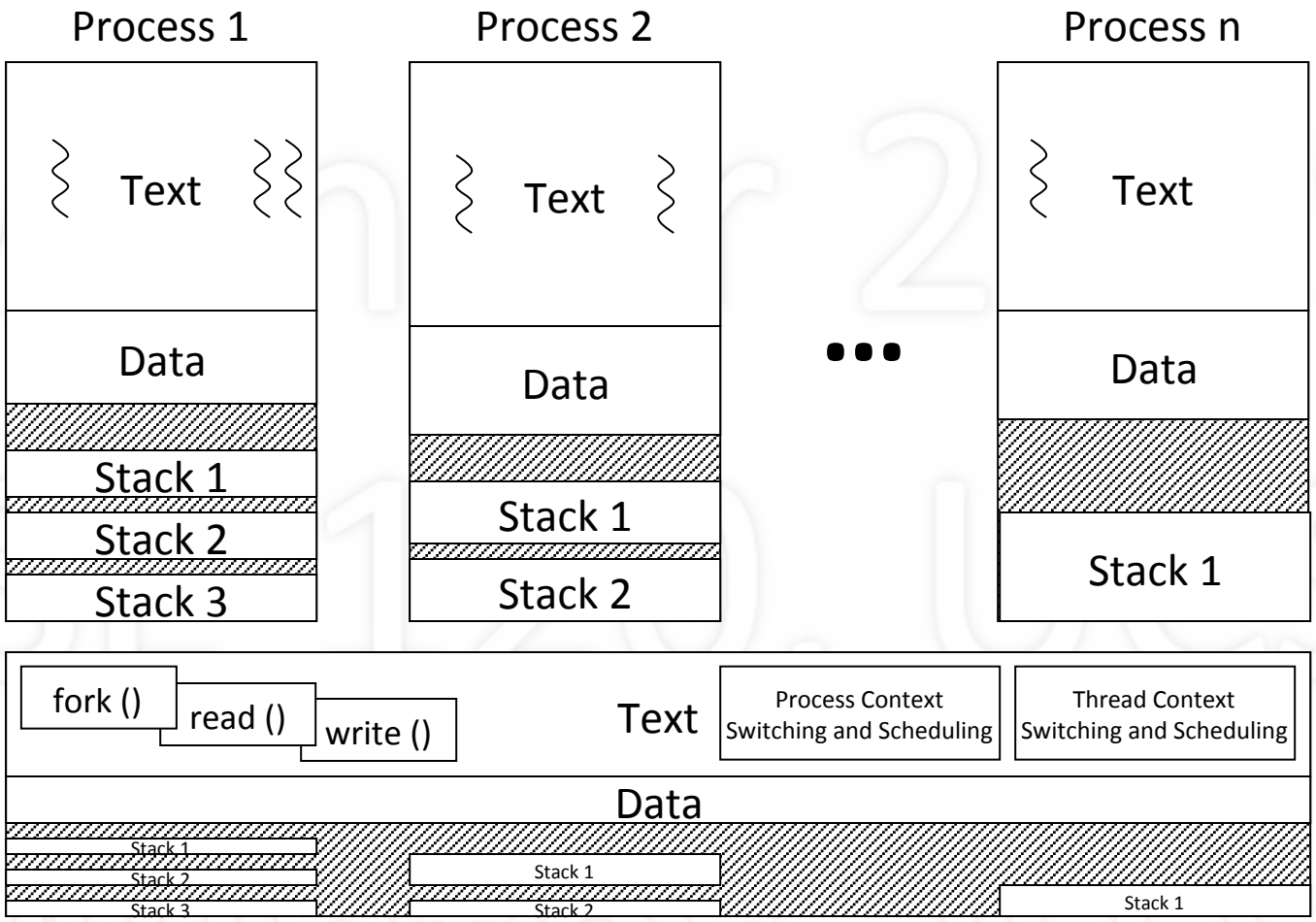
Implementing Threads

- 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



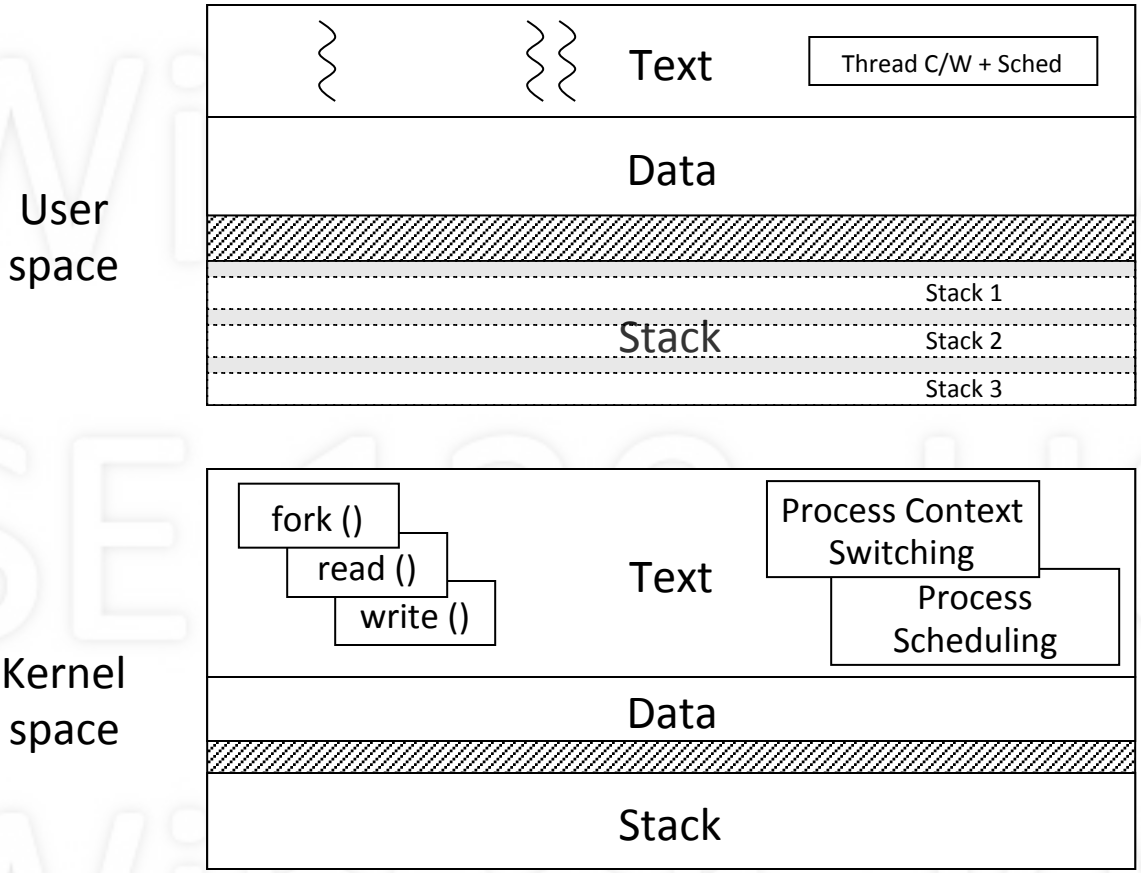
Many Processes with Threads



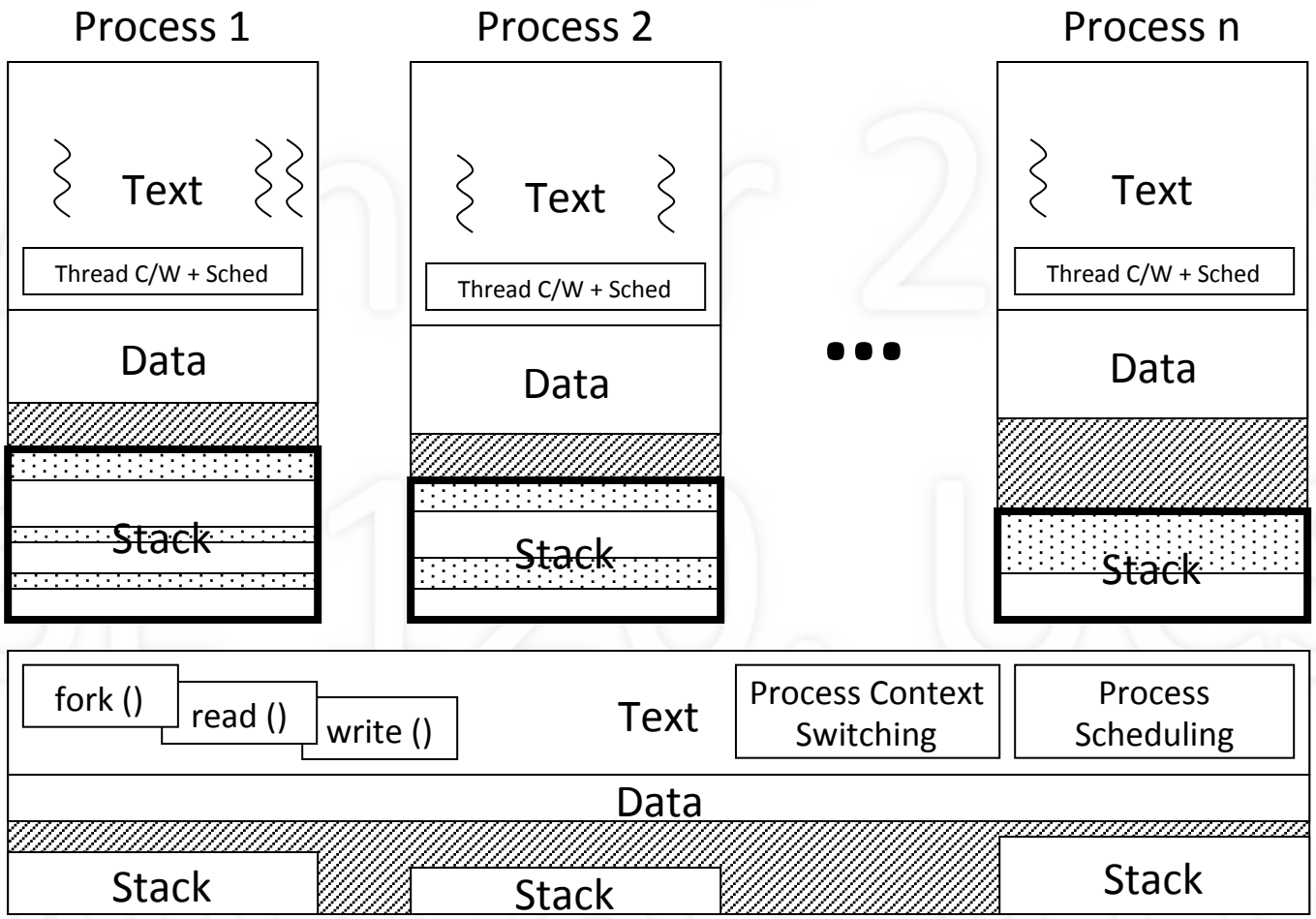
User-Level Threads

- 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

- Chapters 3, 4, 13 (processes, threads)
  - Lecture-related: 3.1-3.3, 3.9, 4.1, 4.3, 4.8, 12.3.3
  - Recommended: 4.2, 4.4-4.7

Review & Research

- What is the difficulty in supporting multiple processes when there is a single CPU?**
- What would it mean to say multiple processes are actually running simultaneously?***
- What would it mean for it to seem to a human that multiple processes are running simultaneously despite there being one CPU? ** What might be a test for this?***

R&R

- What is meant by “timesharing”?*
- How is timesharing implemented?
- What is meant to say “a process is: (a) Running?”; (b) Ready?” (c) Blocked?”
- How are the Running and Ready states similar?*
- How do the Running and Ready states differ?*
- How are the Ready and Blocked states similar?*
- How do the Ready and Blocked states different?*

R&R

- When the kernel decides which process to run next, does it look only at Ready processes, or only Blocked processes, or both, and why?*
- What is a state transition?*
- What is meant by the following state transitions: (a) Dispatch? (b) Preempt? (c) Sleep? (d) Wakeup?

R&R

- What does the word “preemption” mean in general (its dictionary meaning)?
- What does the word “preemption” mean in the context of operating systems?*
- Why is there no transition from the Ready state to the Blocked state?**
- Why is there no transition from the Blocked state to the Running state?**

R&R

- What is the difference between “logical execution” and “physical execution”?*
- Why does it not make sense to have a process that is physically executing but not logically executing?**

R&R

- What are four examples of system calls?
- What are two examples of system management functions?
- Why are the functions for system management placed inside the kernel?*
- Why are the functions for system calls placed inside the kernel?*

R&R

- When does the kernel run?
- Why does the kernel run only at these times? ***
- Is the kernel a process?
- What is a justification for the view as to whether the kernel is a process?**
- Can you offer a reason for why it might be good to adopt the alternative view?***

R&R

- What is the difference between User Space and Kernel Space?*
- When a process runs in User Space, what kind of code is it executing?*
- When a process runs in Kernel Space, what kind of code is it executing?*
- Why make a distinction between User Space and Kernel Space?**

R&R

- How does a process go (change where it is executing) from User Space to Kernel Space? **
- How does a process go (change where it is executing) from Kernel Space to User Space? ***
- What kind of information is found in the kernel's text area?* data area?* stack area?**

R&R

- What is kept in the Process Table?
- Can you justify why each item of information you mentioned is kept in the Process Table: Why is each needed, and what kernel decisions depend on it?**

R&R

- What are the two ways that the kernel can get control, i.e., gets the CPU and starts running?
- How does a process give up control voluntarily?
- Can a process give up control without directly calling yield?*
- How can the CPU be forcibly taken away from a process?* Who takes it away?**

R&R

- What happens when a hardware interrupt occurs, as far as the kernel is concerned?*
- How can the kernel ensure that it will eventually get to run?**

R&R

- What are the steps, in both hardware and software, for doing a context switch?
- What is meant by “user mode”?**
- What is meant by “kernel mode”?**
- If a process is running in user mode, what code is it allowed to execute?*
- Can a process run in kernel mode, and explain why or why not?**

R&R

- When switching from user to kernel mode, what is meant by “amplifies power”?***
- Upon executing the TRAP instruction, where does the CPU start executing?**
- If a hardware interrupt occurs, where does the CPU start executing?**

R&R

- When the kernel selects the next process to run, what state must that process be in?**
- What if there are no processes in those states, what process should run next?***
- What does the RTI machine instruction do, and how is it different from a normal RET (return) machine instruction?***

R&R

- What distinguishes a system call from a normal procedure call?**
- What is the TRAP instruction do?*
- How is it that the TRAP instruction appears in a user program (assuming the programmer does not insert it)?**

R&R

- In slides 13 and 14, why is the kernel not shown?***
- In slide 15, what is meant by “TRAP 20”?*
- Why is the kernel shown in slide 15, i.e., why does it appear?***
- In slide 16, how is it that the function “getpid” starts being executed (why that function rather than some other one)?**

R&R

- In slide 16, what is the purpose of curproc?**
- Why is curproc in the Data section of the kernel (why can't it be in the Stack section)?**
- What data structure does curproc point into?*
- What does it mean that curproc points to 457?**
- Can you explain the contents of the kernel's Stack area?**

R&R

- In slide 17, why did the kernel stack grow?*
- In slide 18, why did the kernel stack shrink?*
- In slide 18, what caused c to be set to 457?*
- In slide 19, why did the kernel stack shrink?*
- In slide 20, why did the kernel seem to disappear?***

R&R

- In slide 21, what is meant by “instruction suspended”?**
- In slide 22, why does the kernel appear?**
- In slide 23, what is clockIntr, and how is it that it starts executing?**
- In slide 24, what is SchedProc?**

R&R

- In slide 25, what is the result of calling SchedProc, and why?**
- Could the result have been any different?***
- In slide 25, why is Yield called?**
- In slide 27, why do the memory areas appear on the right side, and why are they in shadow?***
- In slide 27, why are 2 kernel stacks shown?**

R&R

- In slide 27, can you explain the contents of the right-side kernel stack (and in particular, why do the two 457's appear)?**
- In slide 29, can you explain the changes in the data structure in the kernel's Data area?*

R&R

- In slide 30, why did the left side go into shadow while the right side came out of shadow?**
- Given your previous answer, why is kernel stack on the left side NOT in shadow?***
- In slide 31, what caused the change in curproc?*
- In slide 34, what disappeared, and why?***

R&R

- In slide 35, how many processes are there?
- How many threads are there?
- In slide 36, how many processes are there?
- Why are there 3 stacks in the kernel?*

R&R

- In slide 37, what is meant by “multiple paths of execution” within a single process?**
- How is a single process with two paths of execution different from two processes that each have a single path of execution?***
- Why does each path of execution require its own stack?***

R&R

- What is a thread?
- How is a thread different from a process?*
- What is meant by a thread being a “unit of parallelism” to the user?***
- What is meant by a thread being a “unit of schedulability” to the kernel?***

R&R

- What is meant by “user-level threads”?*
- What is meant by “kernel-level threads”?*
- What is the primary difference between user-level threads and kernel-level threads?**
- What is thread management?

R&R

- In slide 40, how many processes are there?
- How many threads are there?*
- Why are there 3 stacks in User space?**
- Why is there only 1 stack in Kernel space?**

R&R

- In slide 41, how many processes are there?
- How many threads does each process have?*
- For each process, how many stacks do they have in user space, and why?**
- For each process, how many stacks do they have in kernel space, and why?**

R&R

- In slide 41, why does each process have its own copy of the thread Context-Switching/ Scheduling Code (CSSC)?**
- Why is the difference between thread CSSC and process CSSC?***
- Why is the process CSSC in the kernel?**

R&R

- In slide 42, how many processes are there?
- How many threads does each process have?*
- For each process, how many stacks do they have in user space, and why?**
- For each process, how many stacks do they have in kernel space, and why?**
- Why is process CSSC and thread CSSC all in the kernel?***

R&R

- In slide 45, how many processes are there?
- How many threads does each process have?*
- For each process, how many stacks do they have in user space, and why?**
- For each process, how many stacks do they have in kernel space, and why?**

R&R

- In a user-level thread system, where is the thread abstraction supported?**
- In a kernel-level thread system, where is the thread abstraction supported?**
- In a user-level thread system, can threads execute in: user space? kernel space? both?**
- In a kernel-level thread system, can threads execute in: user space? kernel space? both?**

R&R

- What are the advantages of user-level threads over kernel-level threads?*
- What are the advantages of kernel-level threads over user-level threads?*