CS 31: Introduction to Computer Systems

21: Parallel Programming

April 21, 2019

SWARTHMORE COLLEGE

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory
- Into desirable conveniences: illusions
  - Simple, easy-to-use resources
  - Multiple/unlimited number of processors
  - Large/unlimited amount of memory

Kernel provides common functions

- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in central place (kernel)
  - Called by programs (system calls)
  - Or accessed implicitly
- What should functions be?
  - How many programs should benefit?
  - Might kernel get too big?

Process Management: Summary

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
  - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a context switch
- Processes communicate either with message passing or shared memory

Process vs. Kernel

- Is the kernel itself a process?
  - No, it supports processes and devices

- OS only runs when necessary…
  - as an extension of a process making system call
  - in response to a device issuing an interrupt

Process vs. Kernel

- Is the kernel itself a process?
  - No, it supports processes and devices

- OS only runs when necessary…
  - as an extension of a process making system call
  - in response to a device issuing an interrupt

Process vs. Kernel

- The kernel is the code that supports processes
  - System calls: fork ( ), exit ( ), read ( ), write ( ), …
  - System management: context switching, scheduling, memory management

Kernel Execution

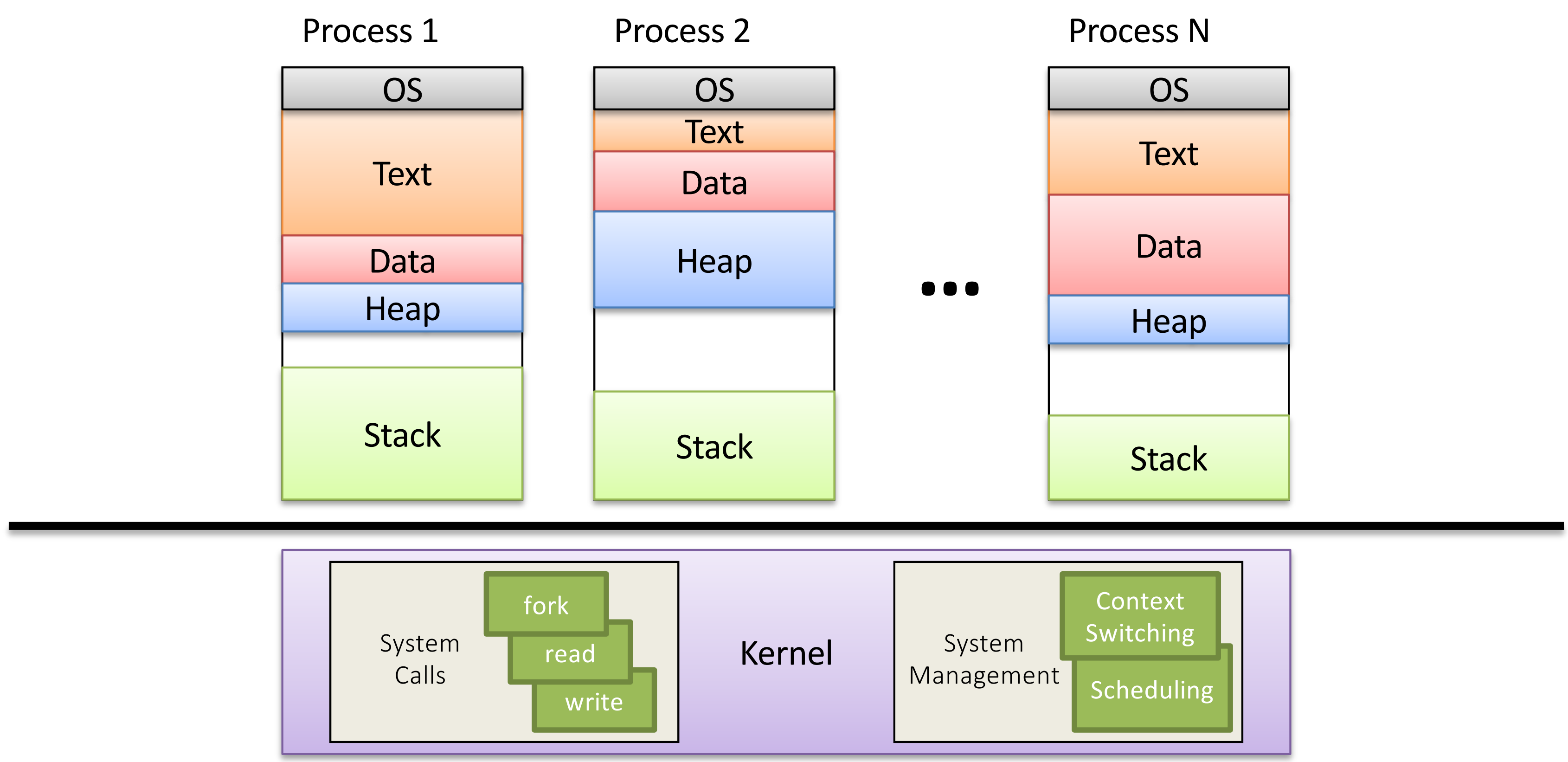
- Great, the OS is going to somehow give us these nice abstractions.

- So…how / when should the kernel execute to make all this stuff happen?

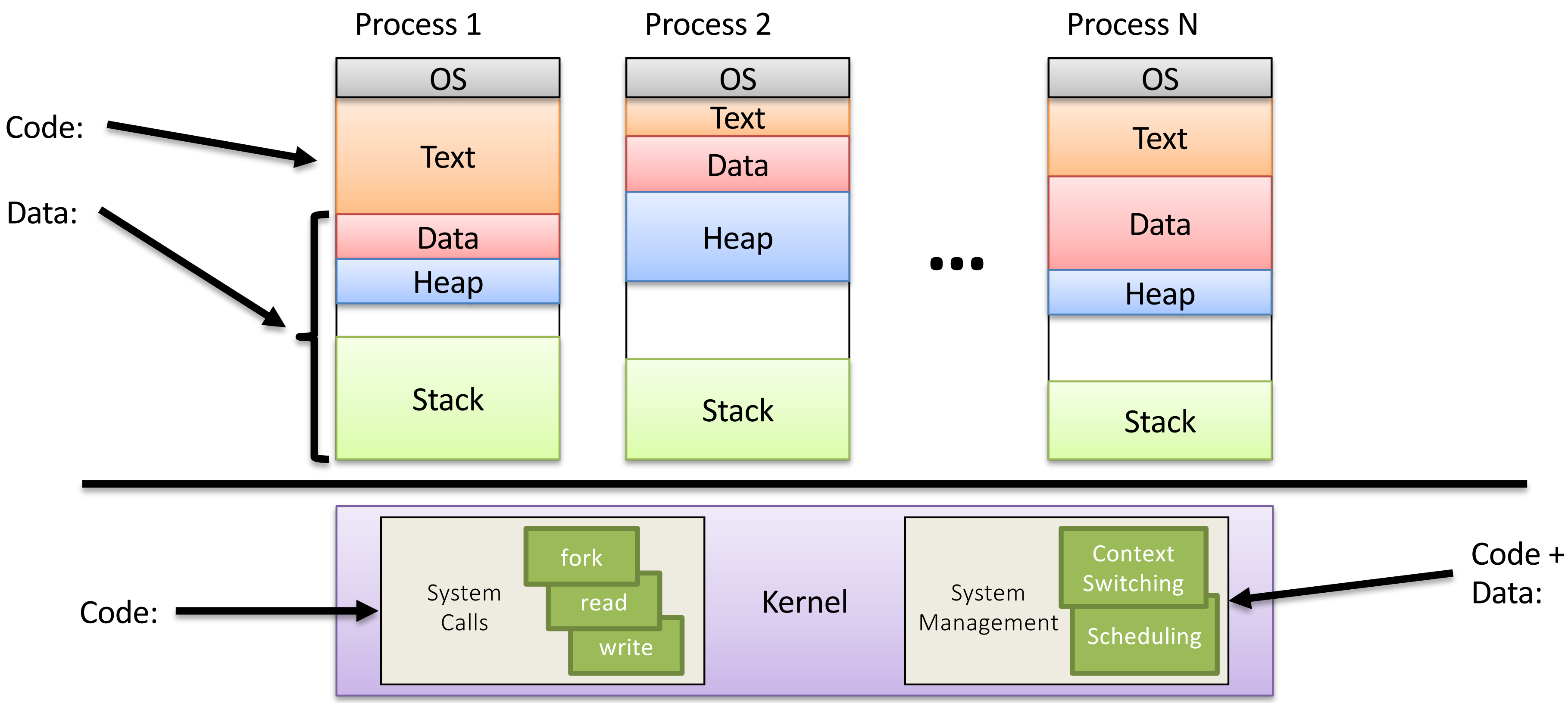
Process vs. Kernel

- The kernel is the code that supports processes
  - System calls: fork ( ), exit ( ), read ( ), write ( ), …
  - System management: context switching, scheduling, memory management

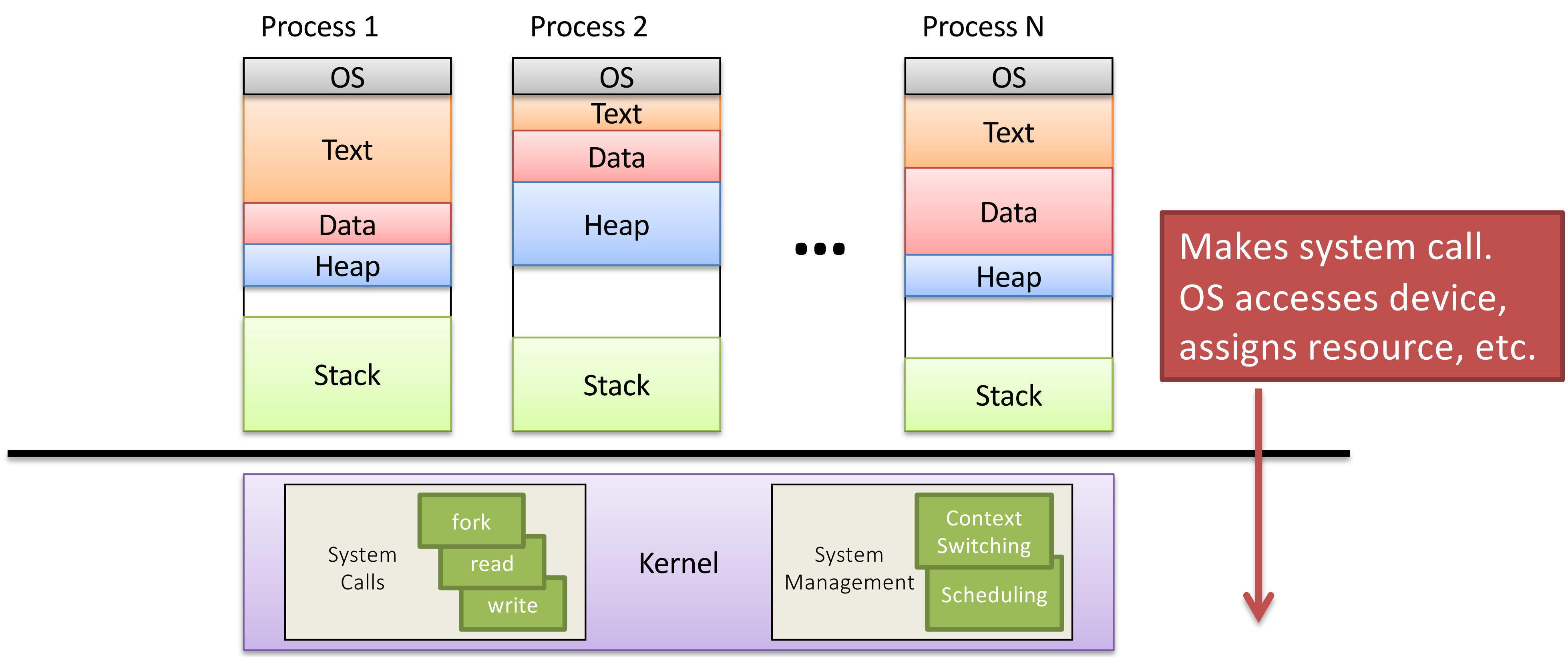
Kernel vs. Userspace: Model



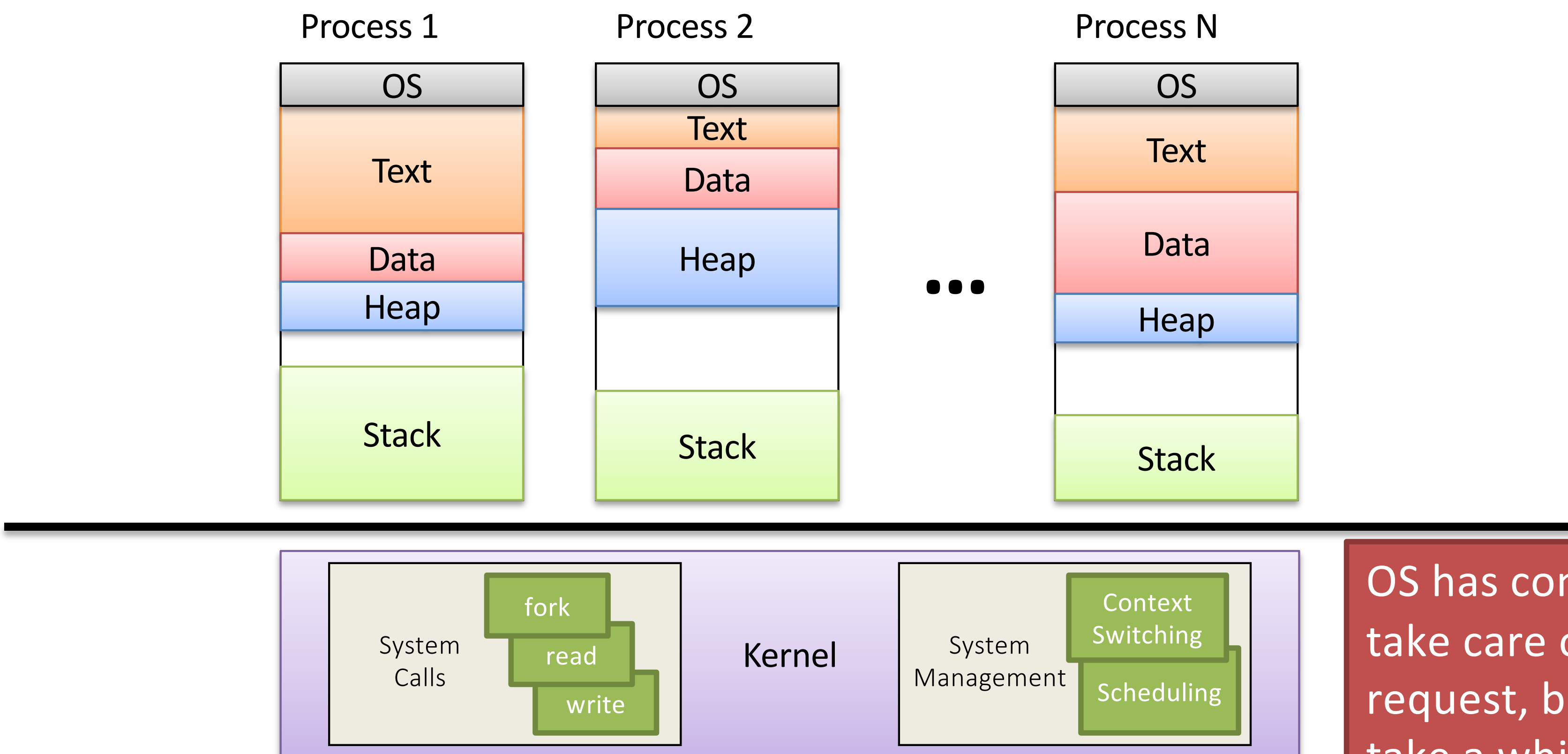
Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



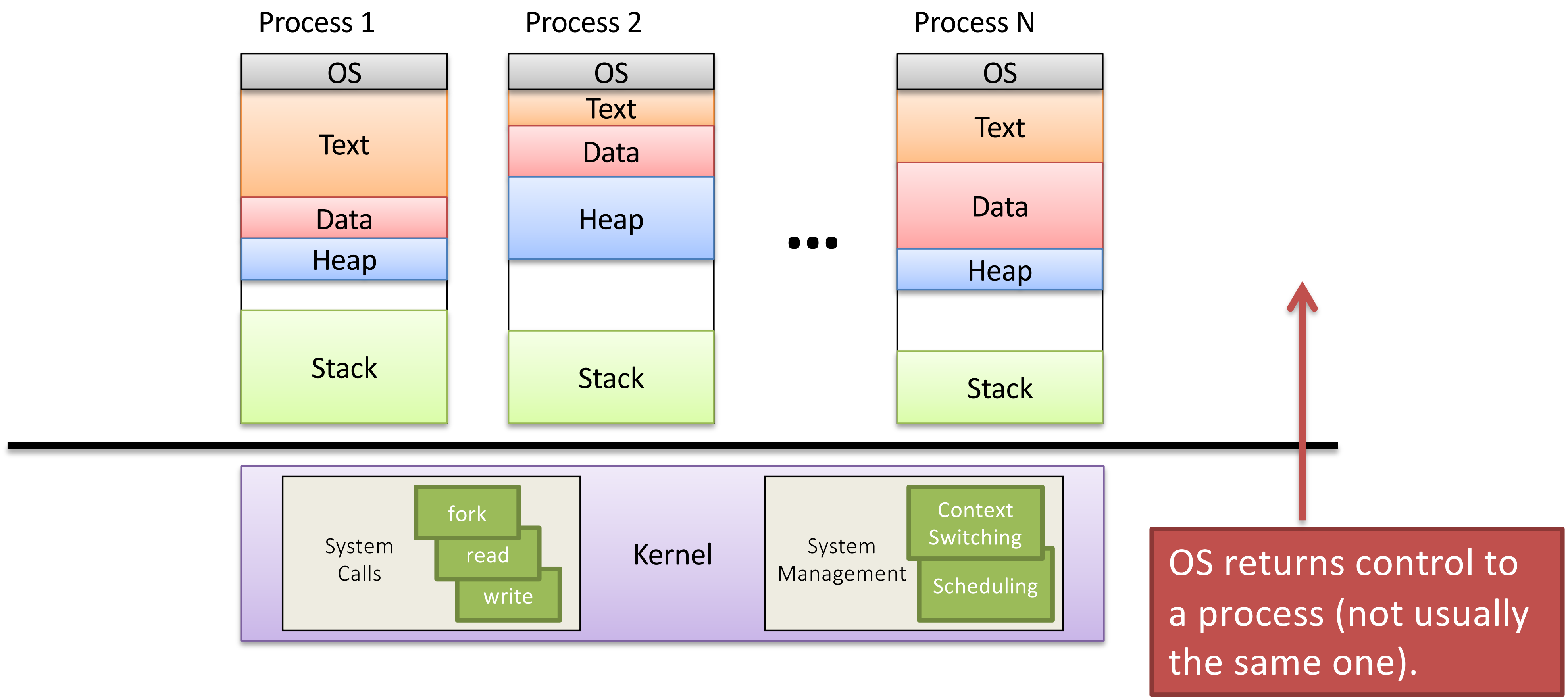
Kernel vs. Userspace: Model



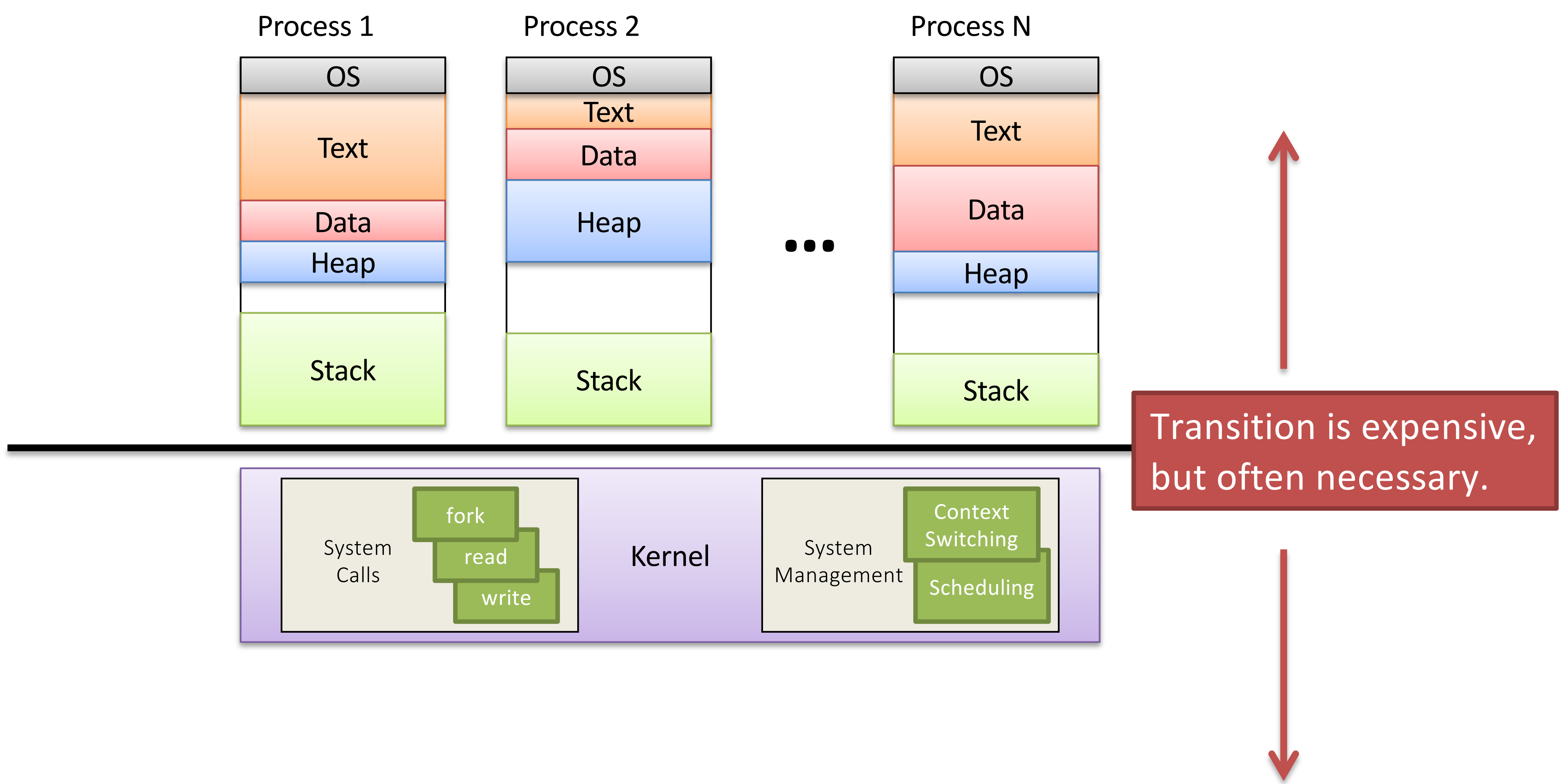
OS has control. It will take care of process's request, but it might take a while.

It can context switch (and usually does at this point).

Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)

Control over the CPU

- To context switch processes, kernel must get control:

1. Running process can give up control voluntarily
   - To block, call yield () to give up CPU
   - Process makes a blocking system call, e.g., read ()
   - Control goes to kernel, which dispatches new process

2. CPU is forcibly taken away: preemption

CPU Preemption

1. While kernel is running, set a hardware timer.

2. When timer expires, a hardware interrupt is generated. (device asking for attention)

3. Interrupt pauses process on CPU, forces control to go to OS kernel.

4. OS is free to perform a context switch.

Summary

- Processes cycled off and on CPU rapidly
  - Mechanism: context switch
  - Policy: CPU scheduling
- Processes created by `fork()` ing
- Other functions to manage processes:
  - exec(): replace address space with new program
  - exit(): terminate process
  - wait(): reap child process, get status info
- Signals one mechanism to notify a process of something

Processor Design Trends

Transistors (*10^3)

Clock Speed (MHZ)

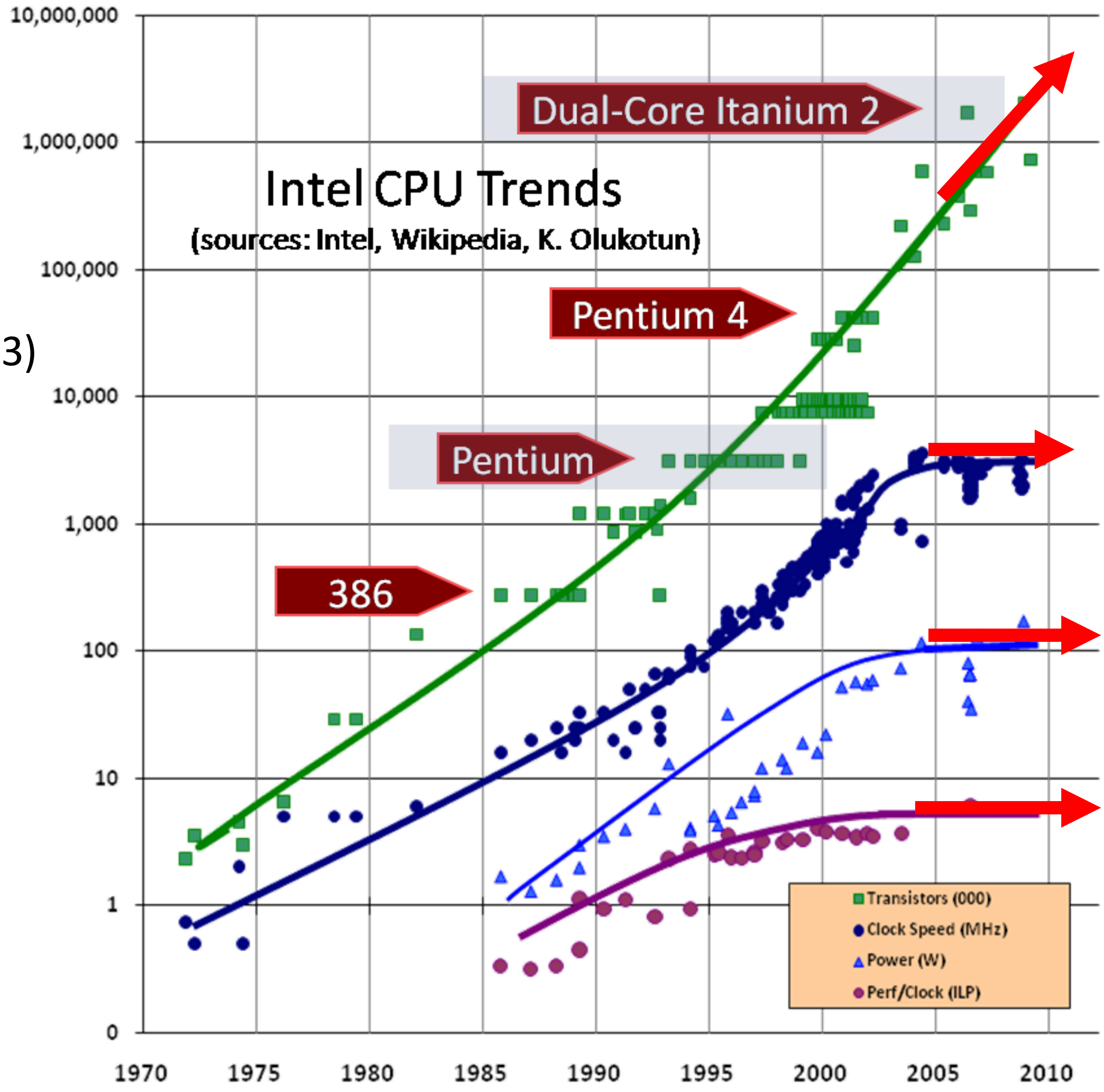
Power (W)

ILP (IPC)

Instruction Level Parallelism

From Herb Sutter,

Dr. Dobbs Journal



Making Programs Run *Faster*

- In the "old days" (1980's - 2005):
  - Algorithm too slow? Wait for HW to catch up.
- Modern CPUs exploit parallelism for speed:
  - Executes multiple instructions at once
  - Reorders instructions on the fly
- Today, can't make a single core go much faster.
  - Limits on clock speed, heat, energy consumption
- Use extra transistors to put multiple CPU cores on the chip.
- Programmer's job to speed-up computation
  - Humans bad at thinking in parallel

Parallel Abstraction

- To speed up a job, must divide it across multiple cores.
- A process contains both execution information and memory/resources.
- What if we want to separate the execution information to give us parallelism in our programs?

Which components of a process might we replicate to take advantage of multiple CPU cores?

A. The entire address space (memory)

B. Parts of the address space (memory)

C. OS resources (open files, etc.)

D. Execution state (PC, registers, etc.)

E. More than one of these (which?)

Which components of a process might we replicate to take advantage of multiple CPU cores?

A. The entire address space (memory – not duplicated)

B. Parts of the address space (memory - stack)

C. OS resources (open files, etc – not duplicated.)

D. Execution state (PC, registers, etc.)

E. More than one of these (which?)

Don’t duplicate shared resources,
duplicate resources where we need a private copy per thread:
like execution state, and stack

Threads

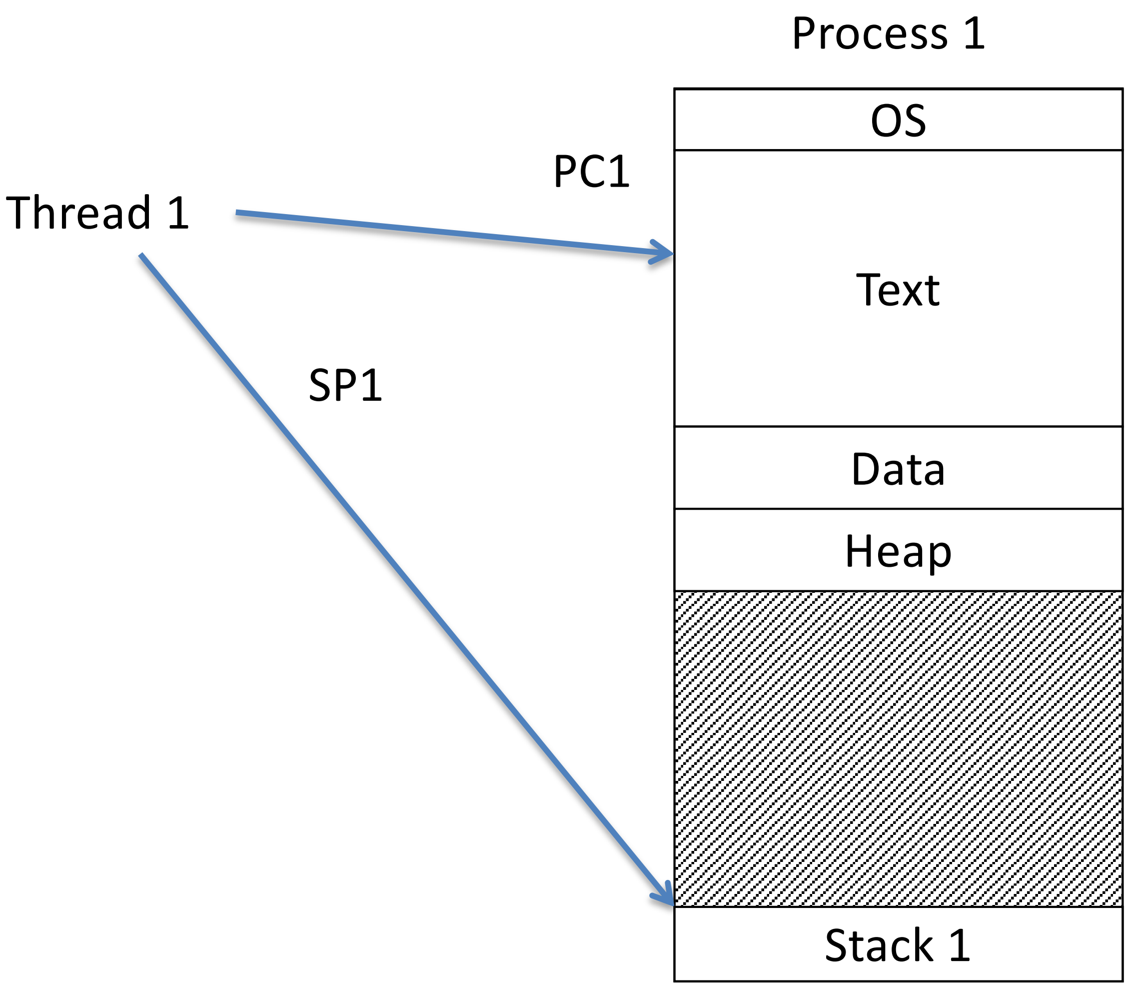
- Modern OSes separate the concepts of processes and threads.
  - The process defines the address space and general process attributes (e.g., open files)
  - The thread defines a sequential execution stream within a process (PC, SP, registers)

- A thread is bound to a single process
  - Processes, however, can have multiple threads
  - Each process has at least one thread (e.g. main)

Processes versus Threads

- A process defines the address space, text, resources, etc.,
- A thread defines a single sequential execution stream within a process (PC, stack, registers).
- Threads extract the thread of control information from the process
- Threads are bound to a single process.
- Each process may have multiple threads of control within it.
  - The address space of a process is shared among all its threads
  - No system calls are required to cooperate among threads

Threads



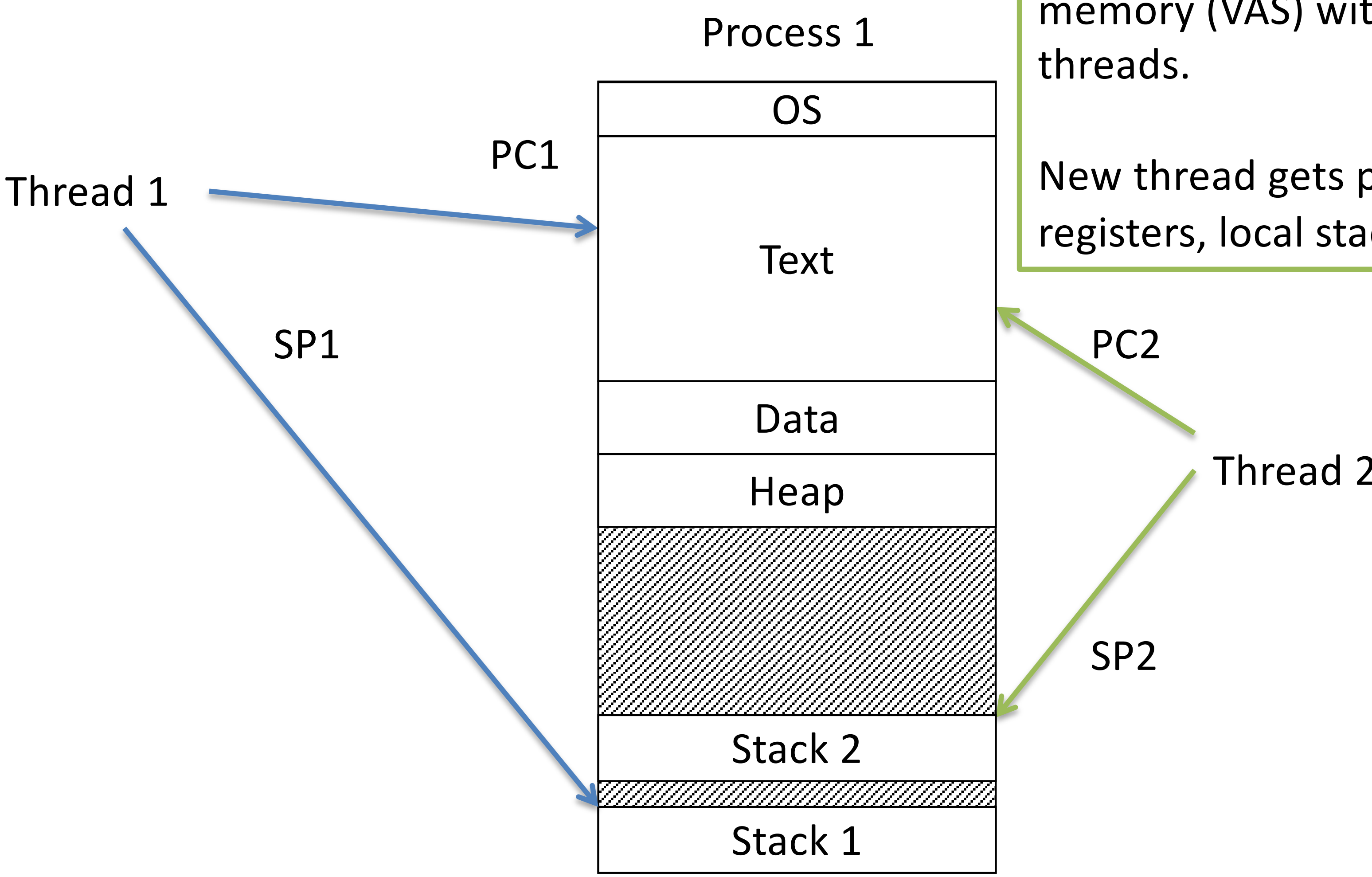
This is the picture we've been using all along:

A process with a single thread, which has execution state (registers) and a stack.

Threads

We can add a thread to the process. New threads share all memory (VAS) with other threads.

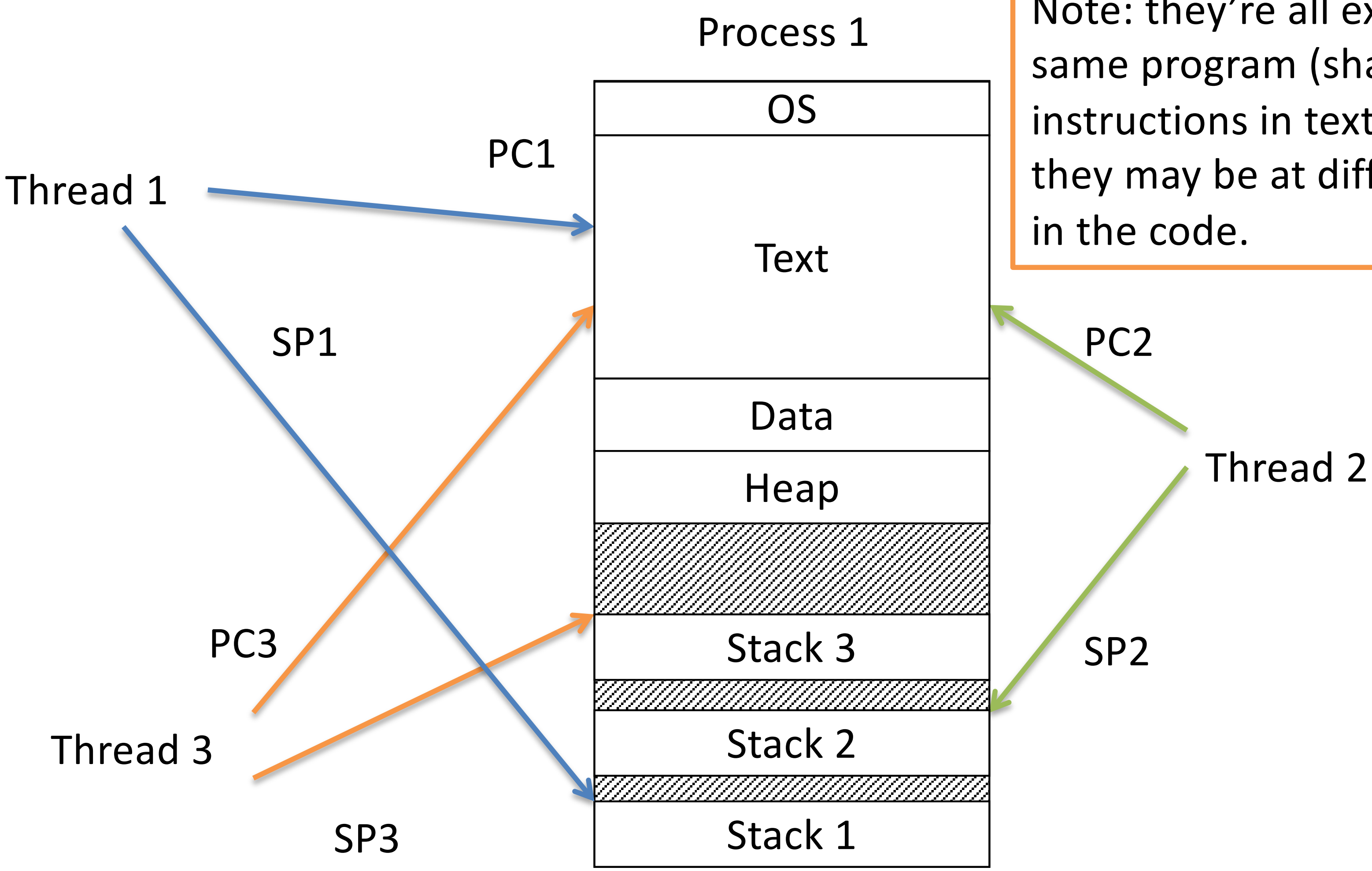
New thread gets private registers, local stack.



Threads

A third thread added.

Note: they're all executing the same program (shared instructions in text), though they may be at different points in the code.



Why Use Threads?

- Separating threads and processes makes it easier to support parallel applications:
  - Creating multiple paths of execution does not require creating new processes (less state to store, initialize – Light Weight Process)
  - Low-overhead sharing between threads in same process (threads share page tables, access same memory)

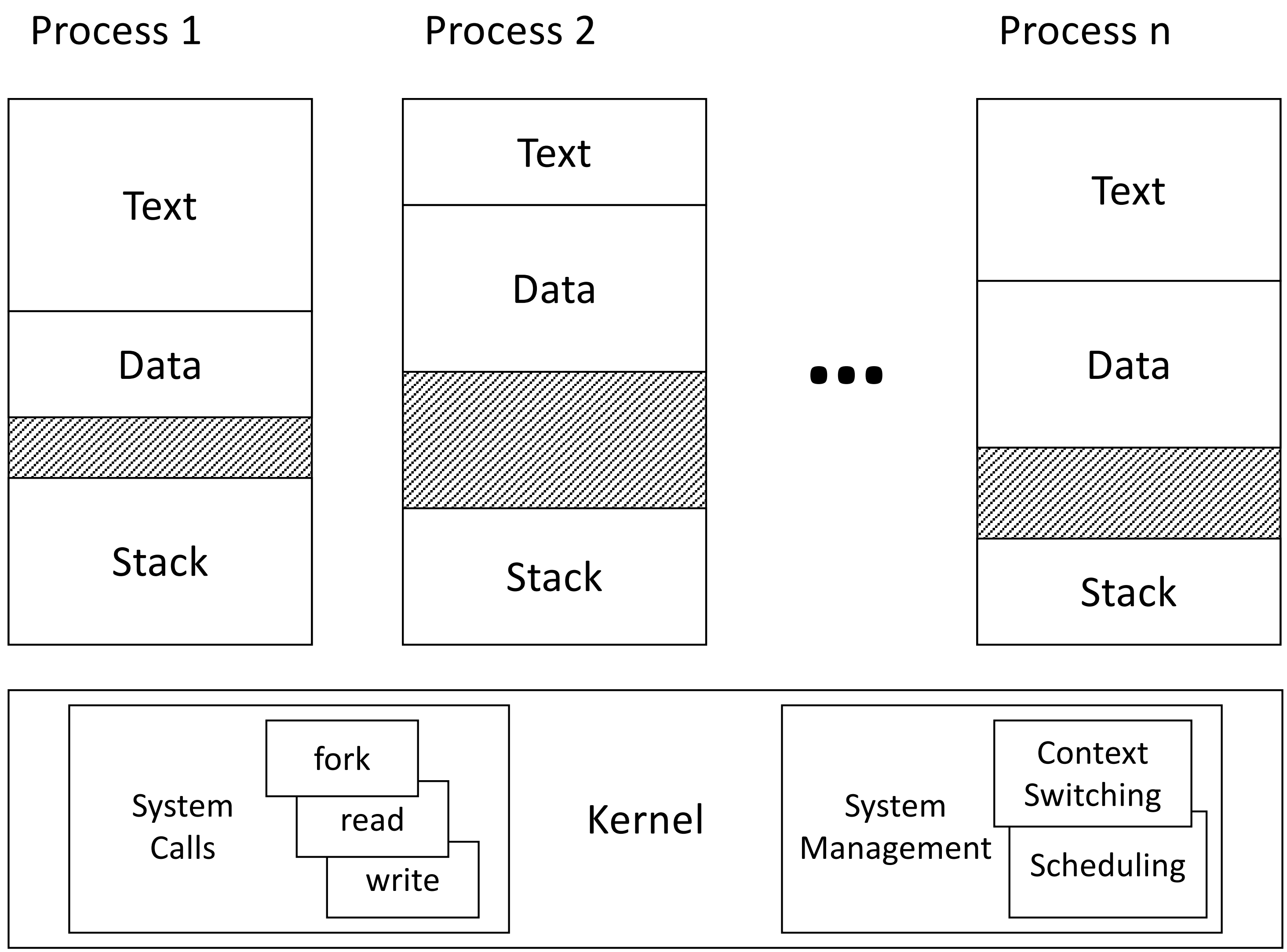
- Concurrency (multithreading) can be very useful

Concurrency?

- Several computations or threads of control are executing simultaneously, and potentially interacting with each other.

- We can multitask! Why does that help?
  - Taking advantage of multiple CPUs / cores
  - Overlapping I/O with computation
  - Improving program structure

Recall: Processes

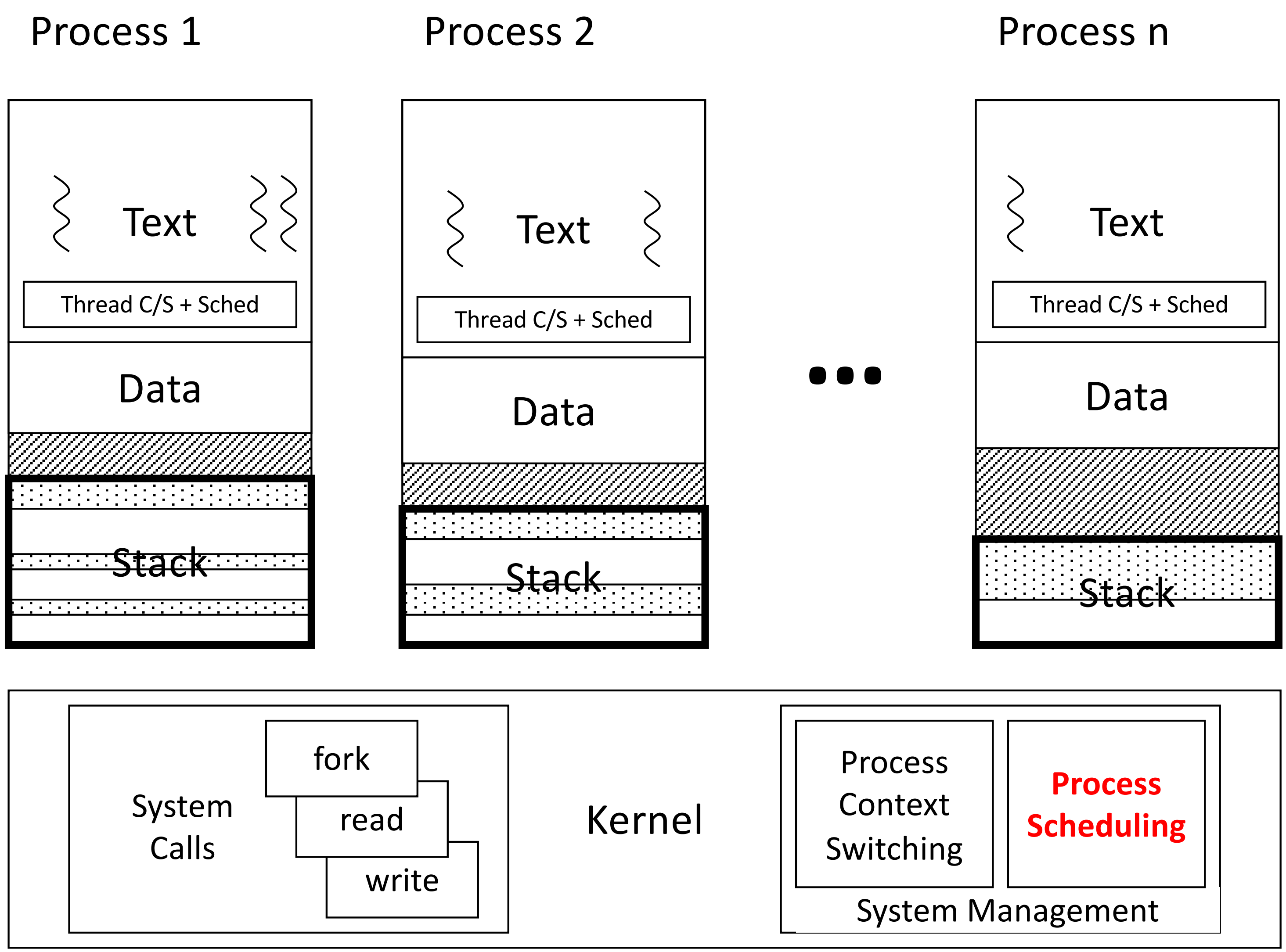


Scheduling Threads

- We have basically two options
    1. Kernel explicitly selects among threads in a process
    2. Hide threads from the kernel, and have a user-level scheduler inside each multi-threaded process

- Why do we care?
    - Think about the overhead of switching between threads
    - Who decides which thread in a process should go first?
    - What about blocking system calls?

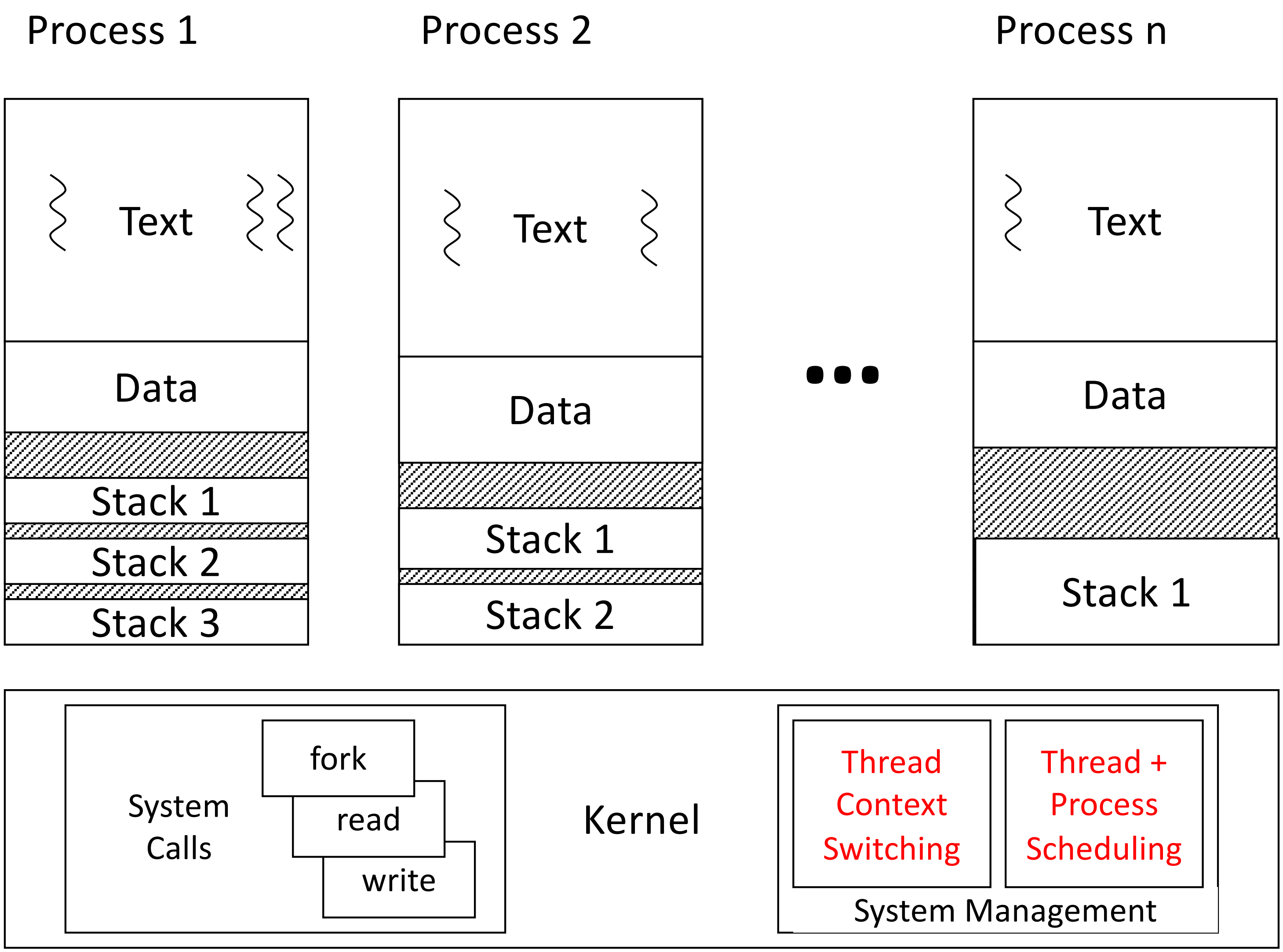
User-Level Threads



Library divides stack region

Threads are invisible to the kernel

Kernel-Level Threads



Kernel Context switching over threads

Each process has explicitly mapped regions for stacks

If you call `thread_create()` on a modern OS (Linux/Mac/Windows), which type of thread would you expect to receive? (Why? Which would you pick?)

A. Kernel threads

B. User threads

C. Some other sort of threads

If you call `thread_create()` on a modern OS (Linux/Mac/Windows), which type of thread would you expect to receive? (Why? Which would you pick?)

A. Kernel threads

B. User threads

C. Some other sort of threads

Kernel vs. User Threads

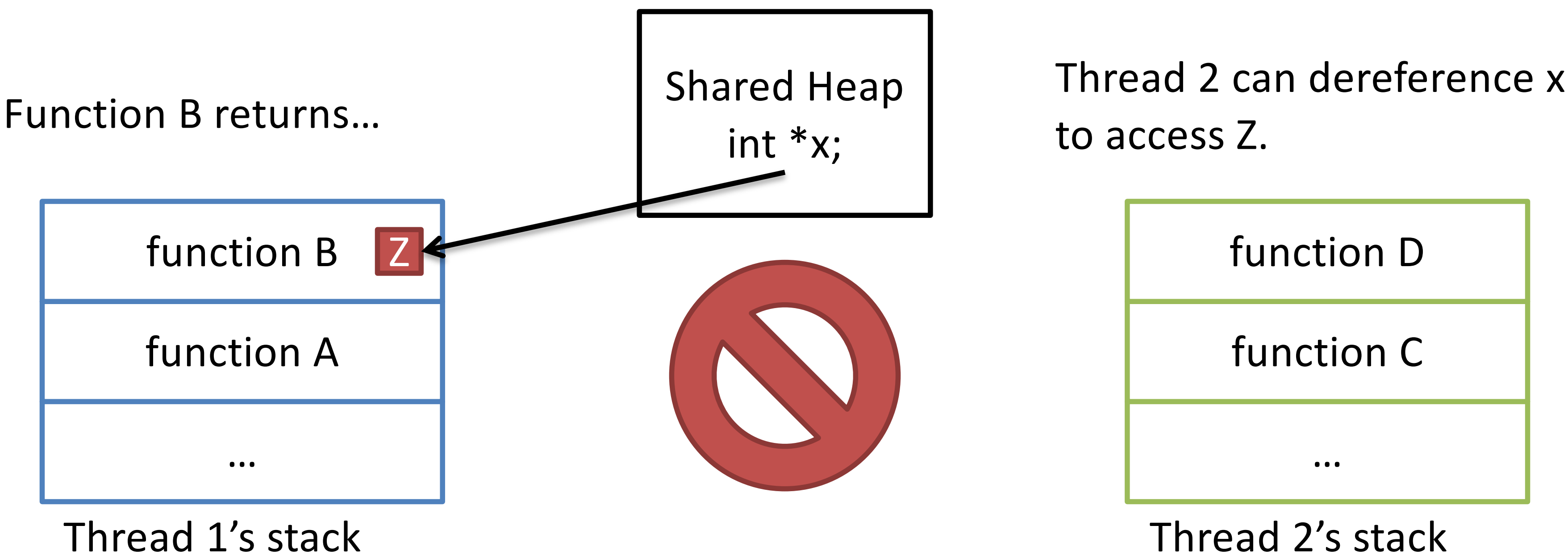
- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slower to create, manipulate, synchronize
    - Requires getting the OS involved, which means changing context (relatively expensive)
- User-level threads
  - Faster to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
    - If one thread makes a syscall, all of them get blocked because the OS doesn't distinguish.

Threads & Sharing

- Code (text) shared by all threads in process
- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete
- Local variables should not be shared
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack!!

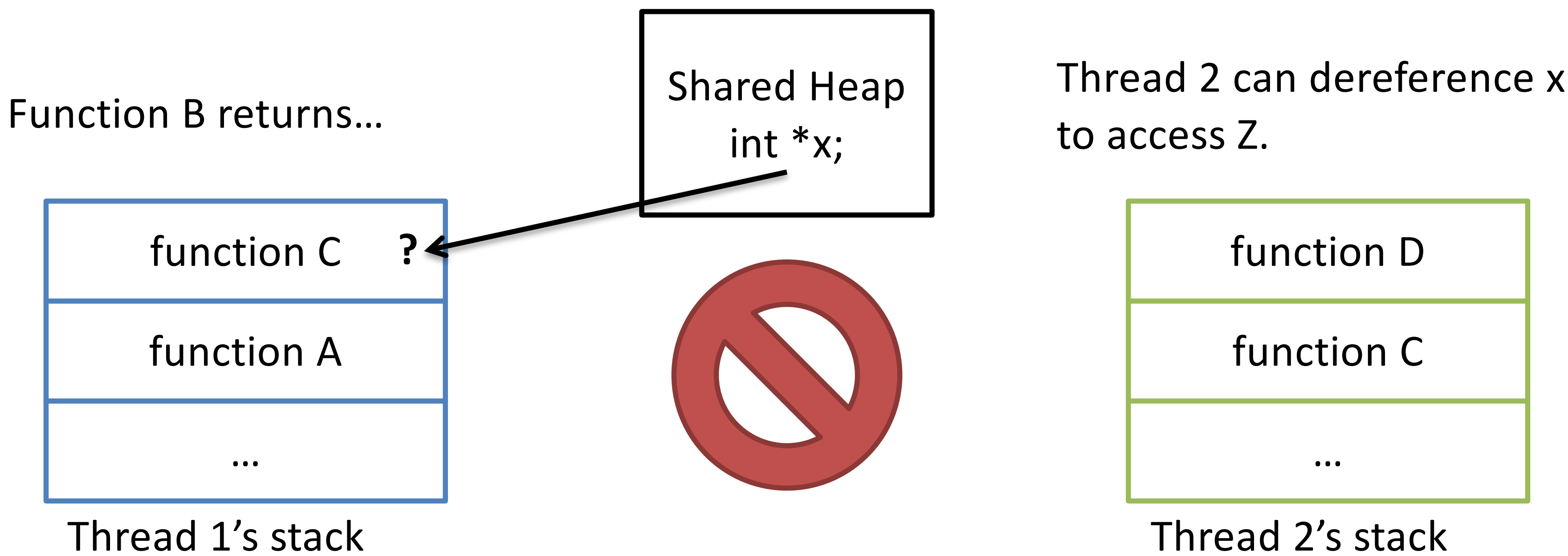
Threads & Sharing

- Local variables should not be shared
  - Refer to data on the stack
  - Each thread has its own stack
  - <u>Never pass/share/store a pointer to a local variable on another thread's stack</u>



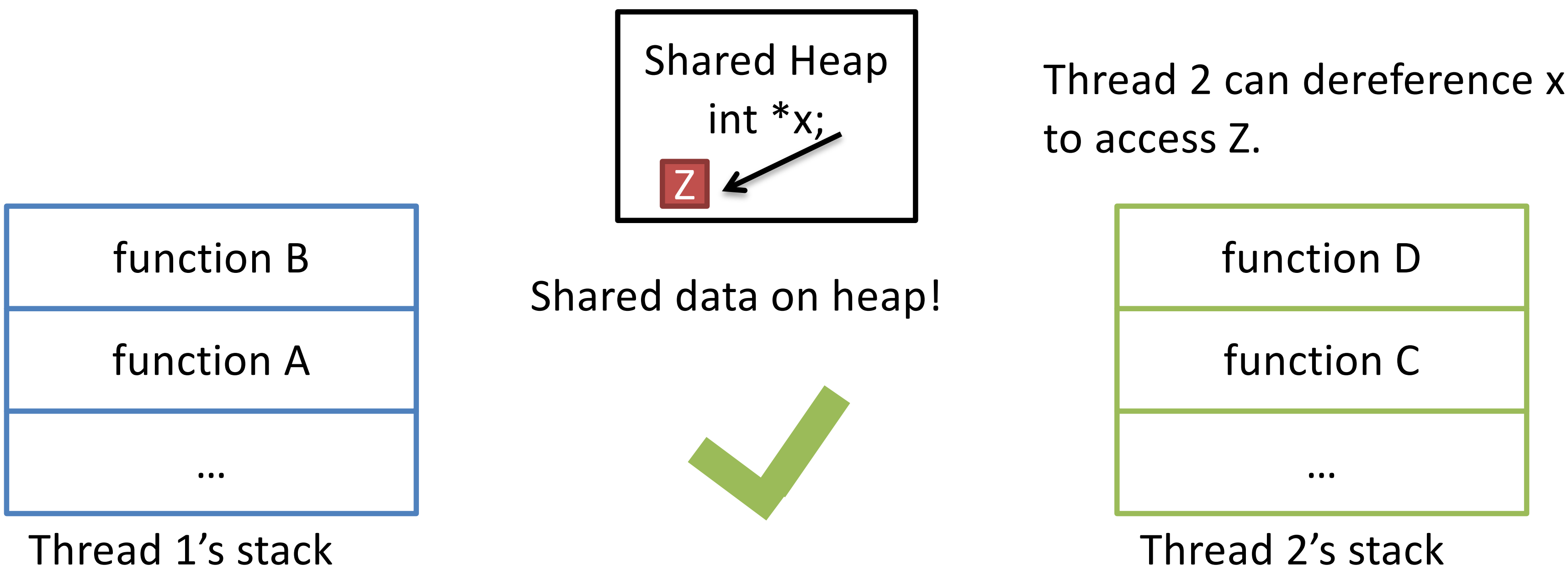
Threads & Sharing

- Local variables should not be shared
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack



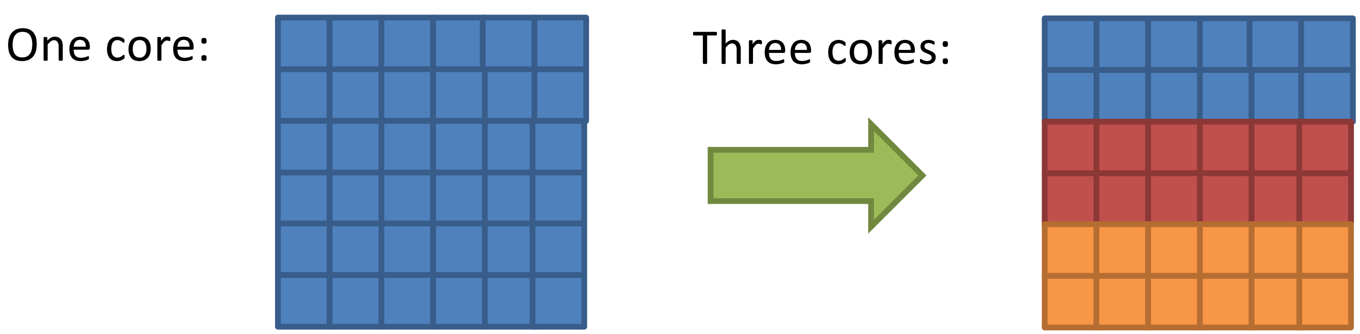
Threads & Sharing

- Local variables should not be shared
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack



Thread-level Parallelism

- Speed up application by assigning portions to CPUs/cores that process in parallel
- Requires:
  - partitioning responsibilities (e.g., parallel algorithm)
  - managing their interaction
- Example: game of life (next lab)



If one CPU core can run a program at a rate of X, how quickly will the program run on two cores? Why?

A. Slower than one core (<X)
B. The same speed (X)
C. Faster than one core, but not double (X-2X)
D. Twice as fast (2X)
E. More than twice as fast(>2X)

If one CPU core can run a program at a rate of X, how quickly will the program run on two cores? Why?

A. Slower than one core (<X) (if we try to parallelize serial applications!)

B. The same speed (X) (some applications are not parallelizable)

C. Faster than one core, but not double (X-2X): most of the time: (some communication overhead to coordinate/synchronization of the threads)

D. Twice as fast (2X)(class of problems called embarrassingly parallel programs. E.g. protein folding, SETI)

E. More than twice as fast(>2X) (rare: possible if you have more CPU + more memory)

Parallel Speedup

- Performance benefit of parallel threads depends on many factors:
  - algorithm divisibility
  - communication overhead
  - memory hierarchy and locality
  - implementation quality
- *For most programs*, more threads means more communication, diminishing returns.

Summary

- Physical limits to how much faster we can make a single core run.
  - Use transistors to provide more cores.
  - Parallelize applications to take advantage.
- OS abstraction: thread
  - Shares most of the address space with other threads in same process
  - Gets private execution context (registers) + stack
- Coordinating threads is challenging!