CS31: Introduction to Computer Systems

Week 13, Class 1
Threads
04/23/24

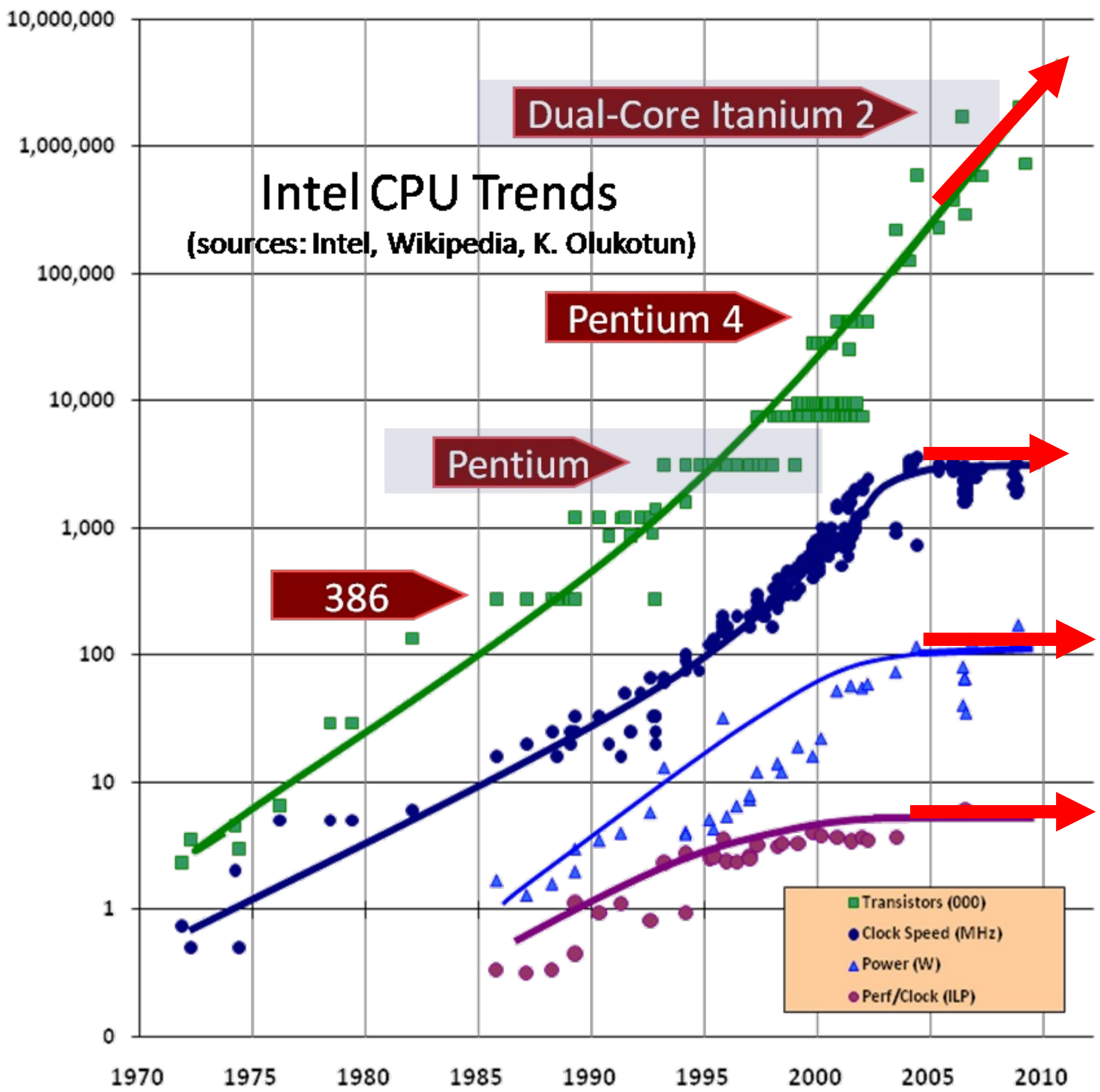
Dr. Sukrit Venkatagiri
Swarthmore College

Making Programs Run *Faster*

- We all like how fast computers are…
- 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 (if it’s safe to do so)
  - With help from the programmer

Processor Design Trends

- Transistors (*10^3)
- Clock Speed (MHZ)
- Power (W)
- ILP (IPC)



From Herb Sutter, Dr. Dobbs Journal

The “Multi-Core Era”

- 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
- Exciting: CPU capable of doing a lot more!
- Problem: up to programmer to take advantage of multiple cores
  - 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 parts of a process does the OS need to keep track of multiple (independent) copies of to run a process on multiple CPU cores in parallel?

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.)

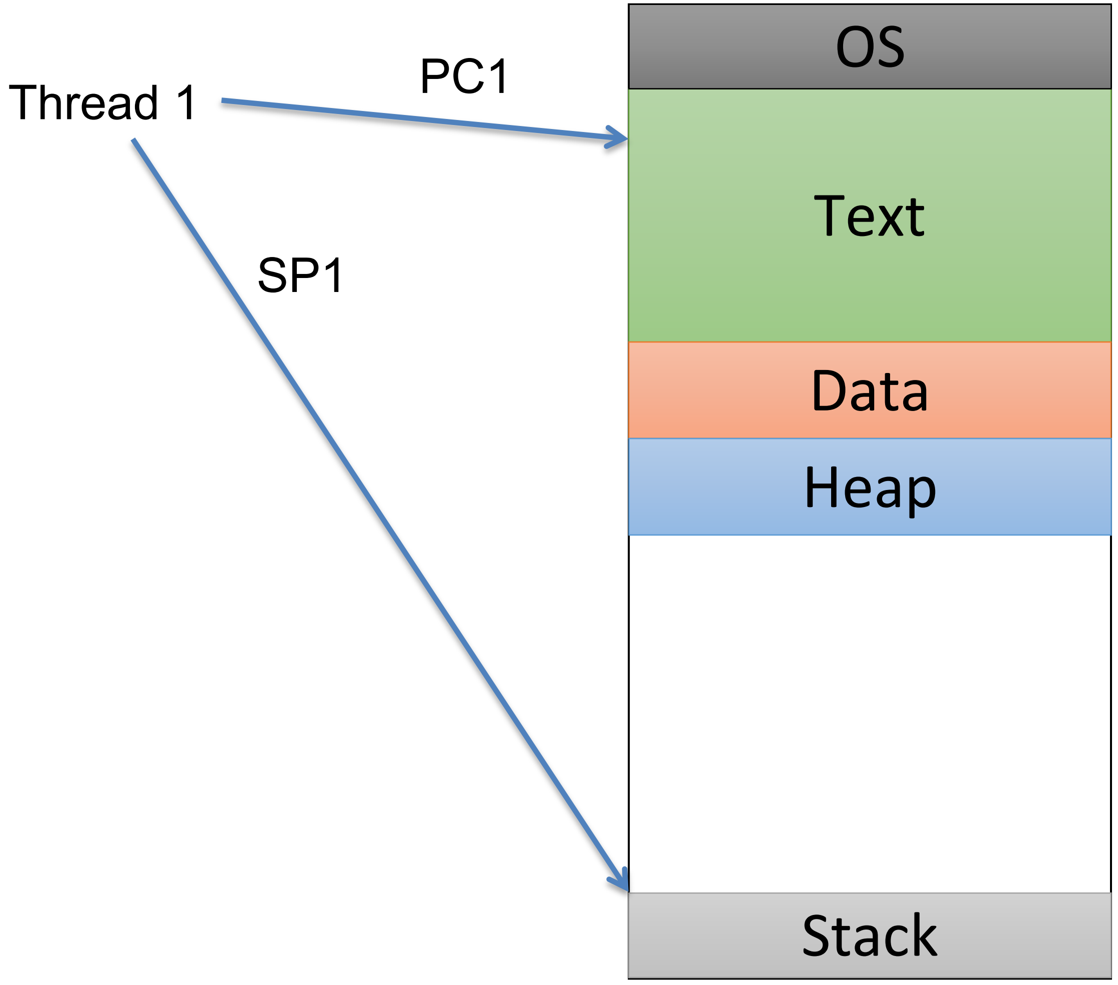
E. More than one of these (which?)

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, other registers)

- A thread is bound to a single process
  - Processes, however, can have multiple threads
  - Each process has at least one thread

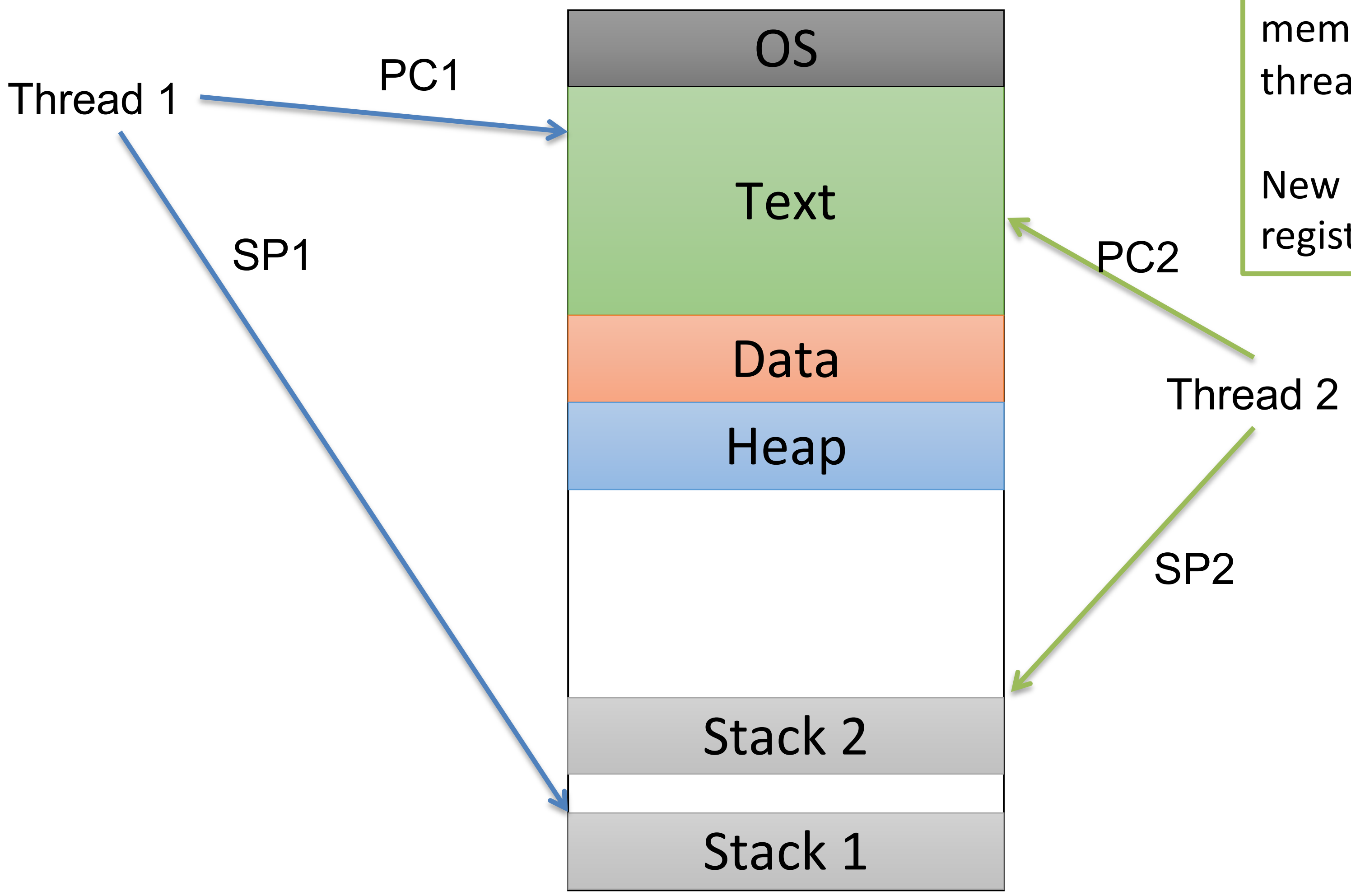
Threads



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



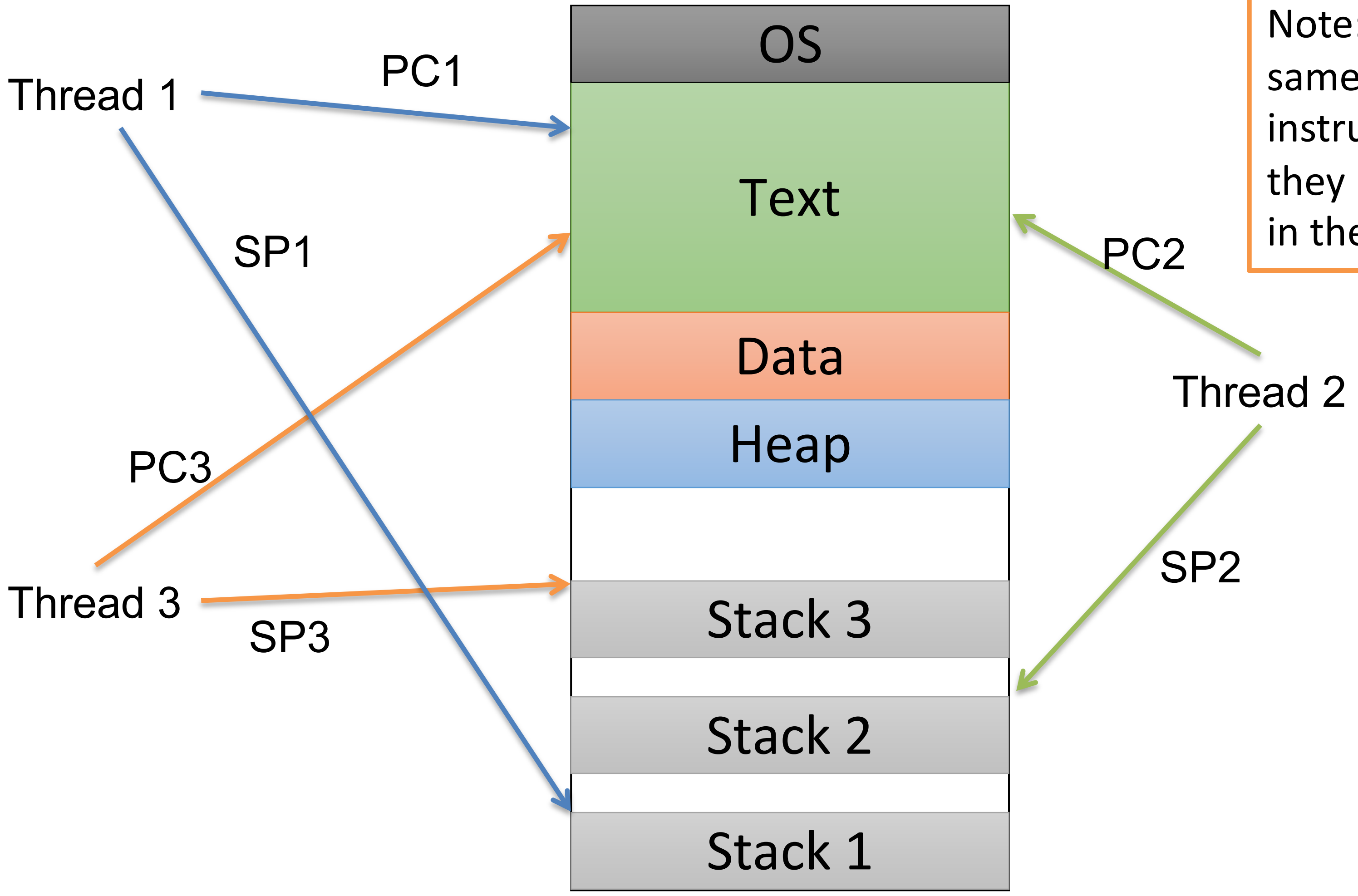
We can add a thread to the process. New threads share all memory (VAS) with other threads.

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 - LWP)
  - Low-overhead sharing between threads in same process (threads share page tables, access same memory)

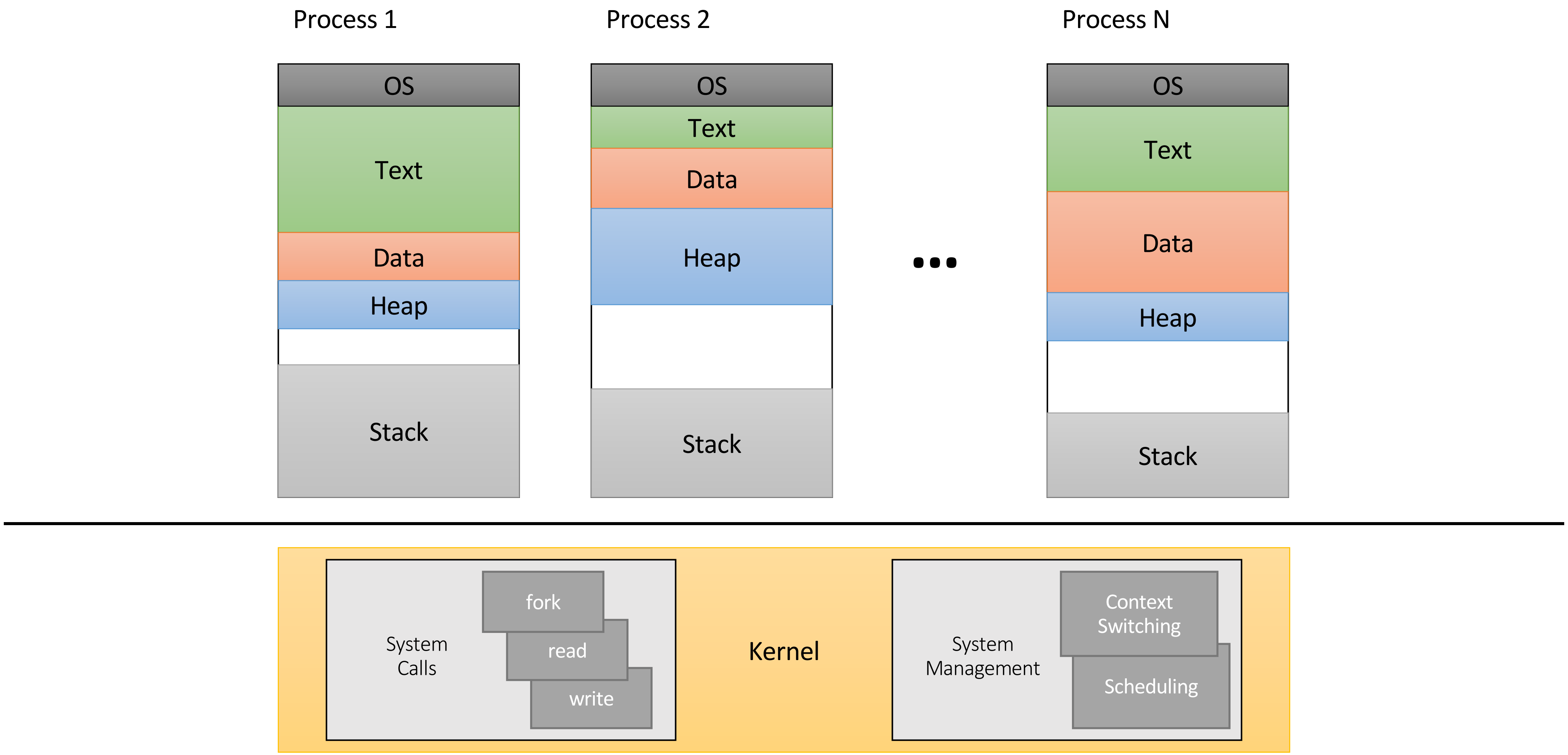
- Concurrency (multithreading) can be very useful

Concurrency?

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

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

Recall: Processes

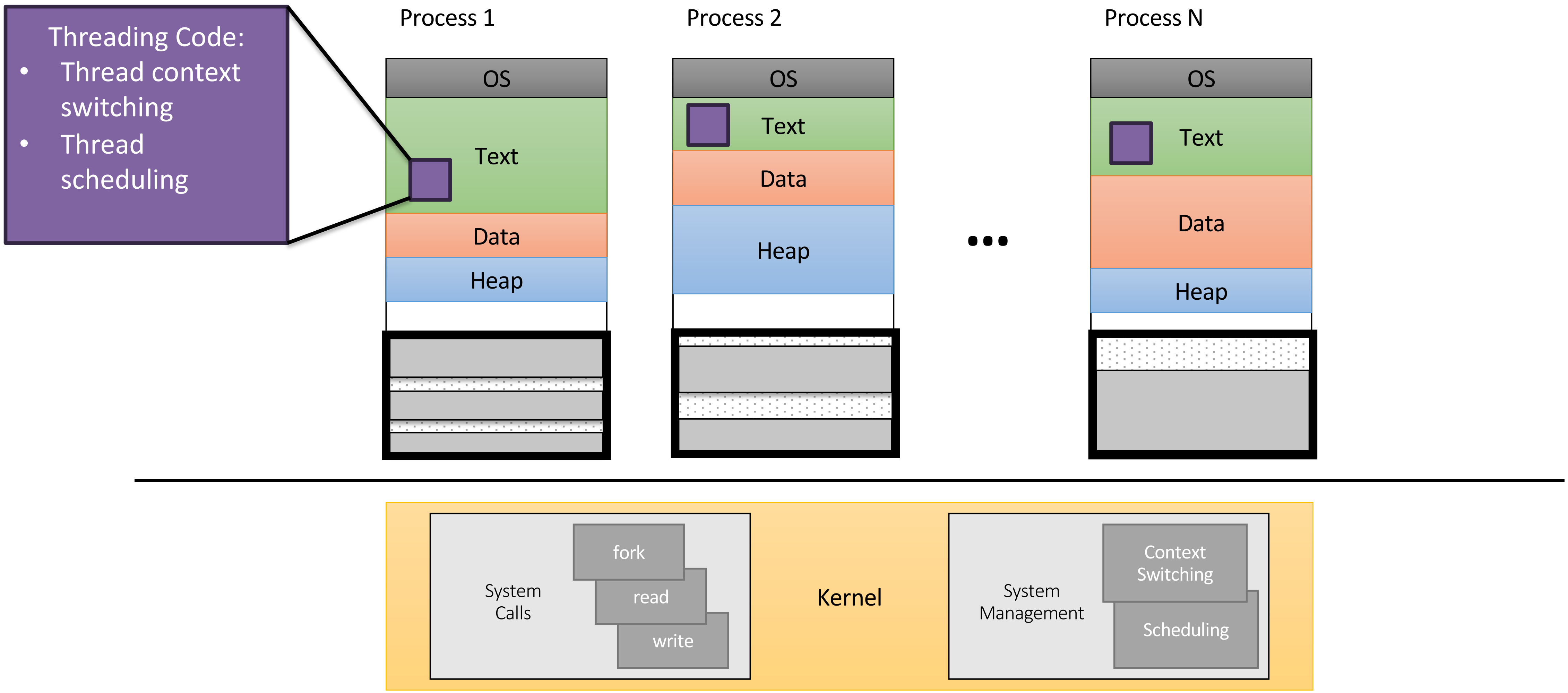


Scheduling Threads

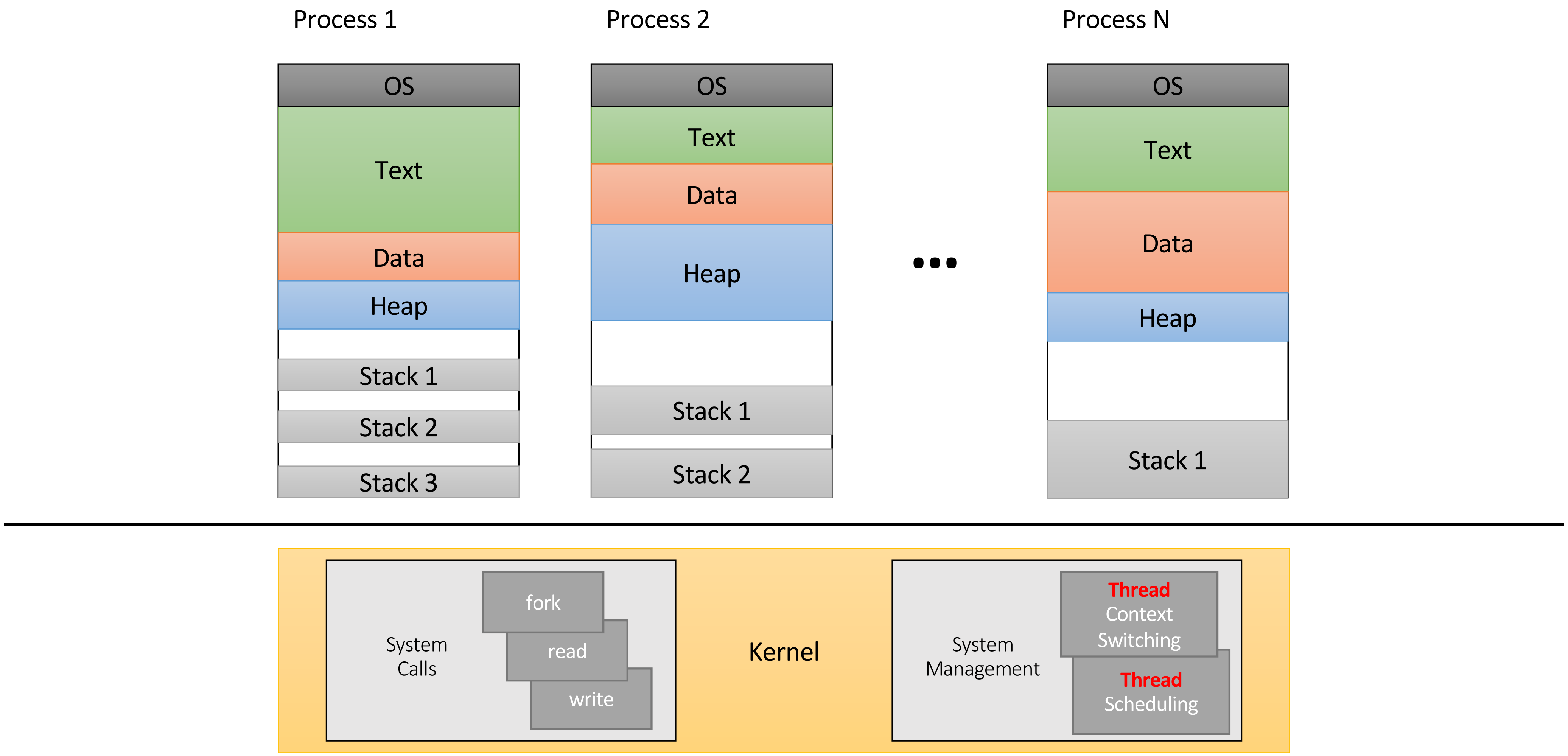
- We have two options
  1. Kernel explicitly selects among threads in a process
  2. Hide threads from the kernel, and have a user-level scheduler inside each multi-threaded process

- 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



Kernel-Level 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

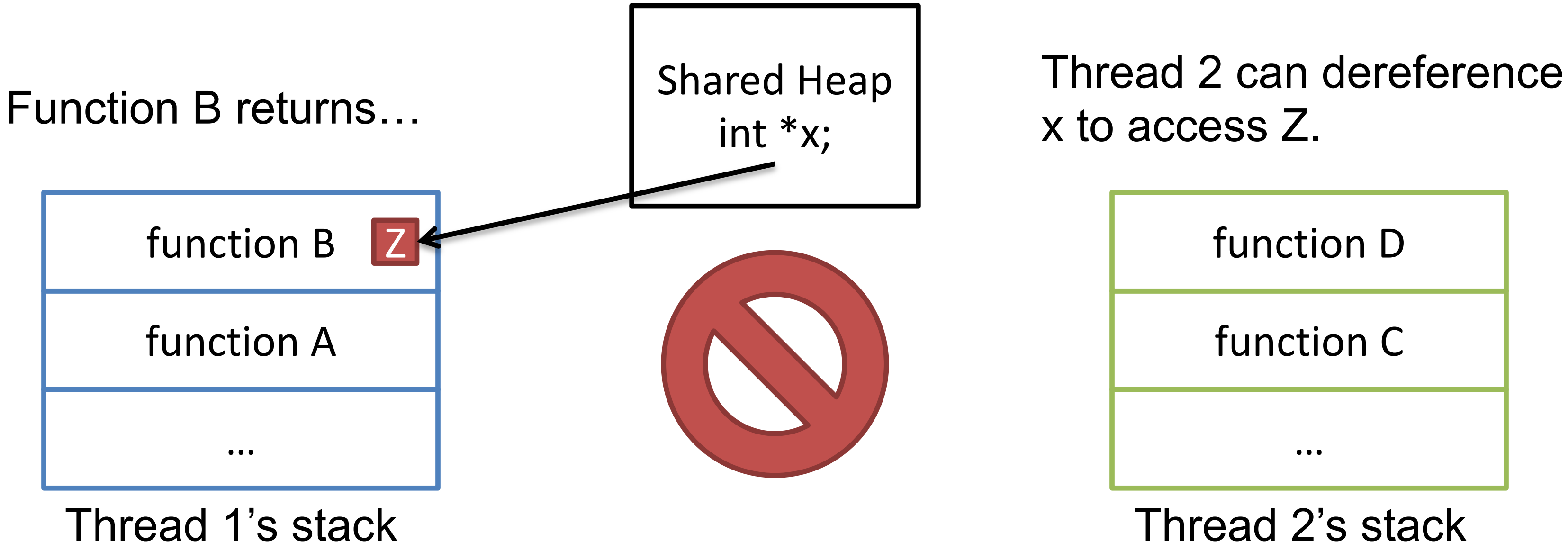
- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slower to create, manipulate, synchronize
    - Requires getting the OS involved, which means making system calls and changing context (relatively expensive)
- User-level threads
  - Faster to create, manipulate, synchronize
  - 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
  - <u>Never pass/share/store a pointer to a local variable on another thread's stack</u>

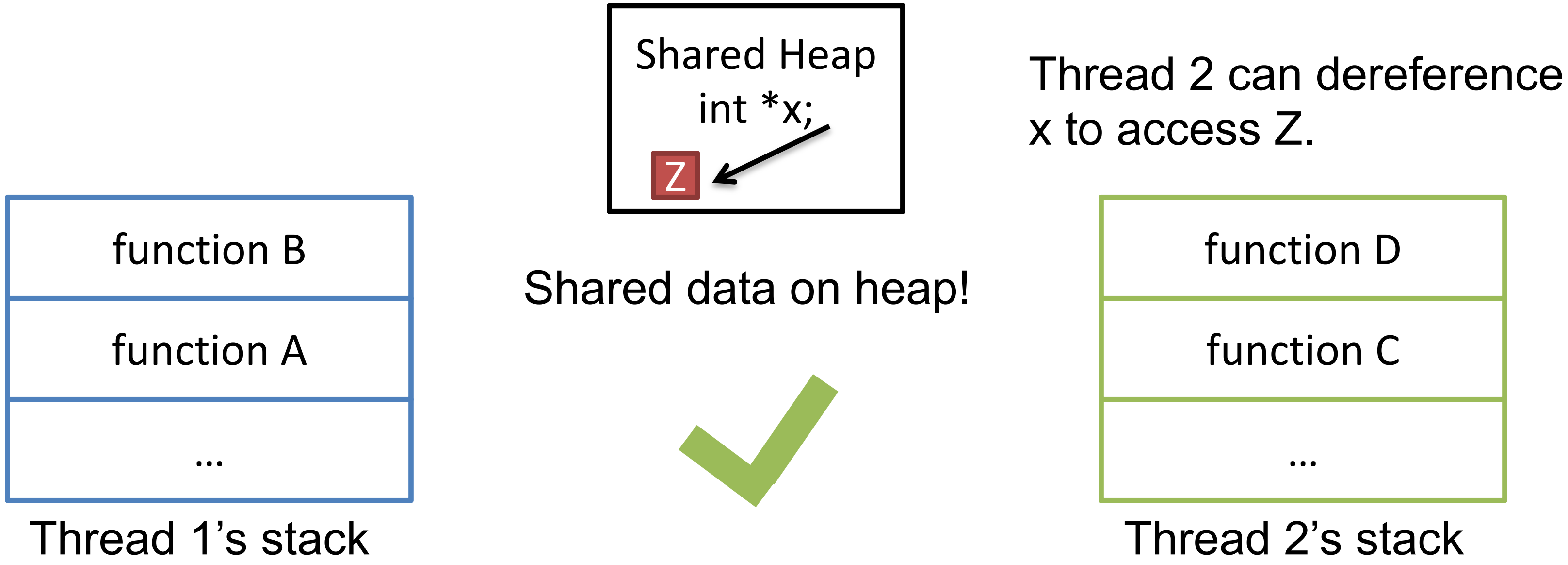
Threads & Sharing

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



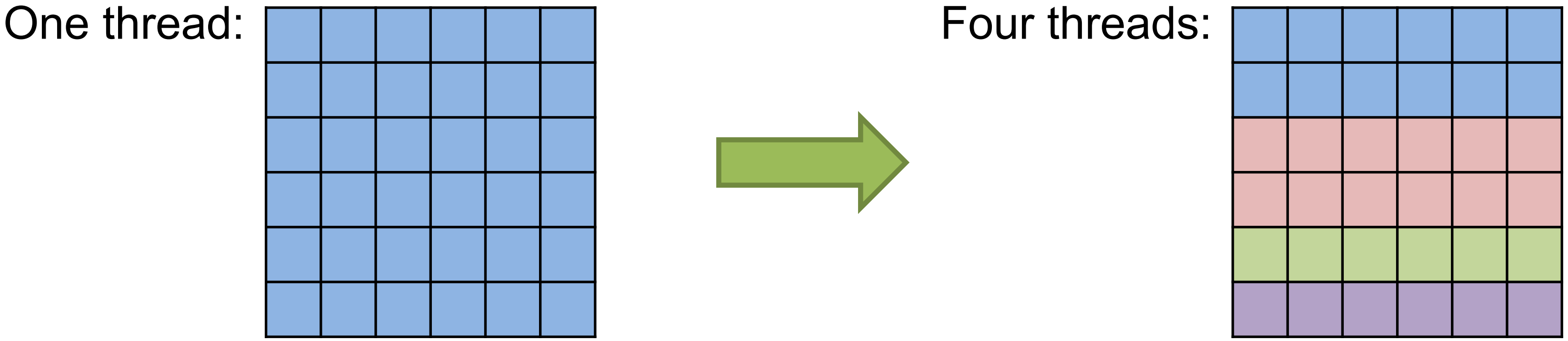
Threads & Sharing

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



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?

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)

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, resulting in 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!