Threads

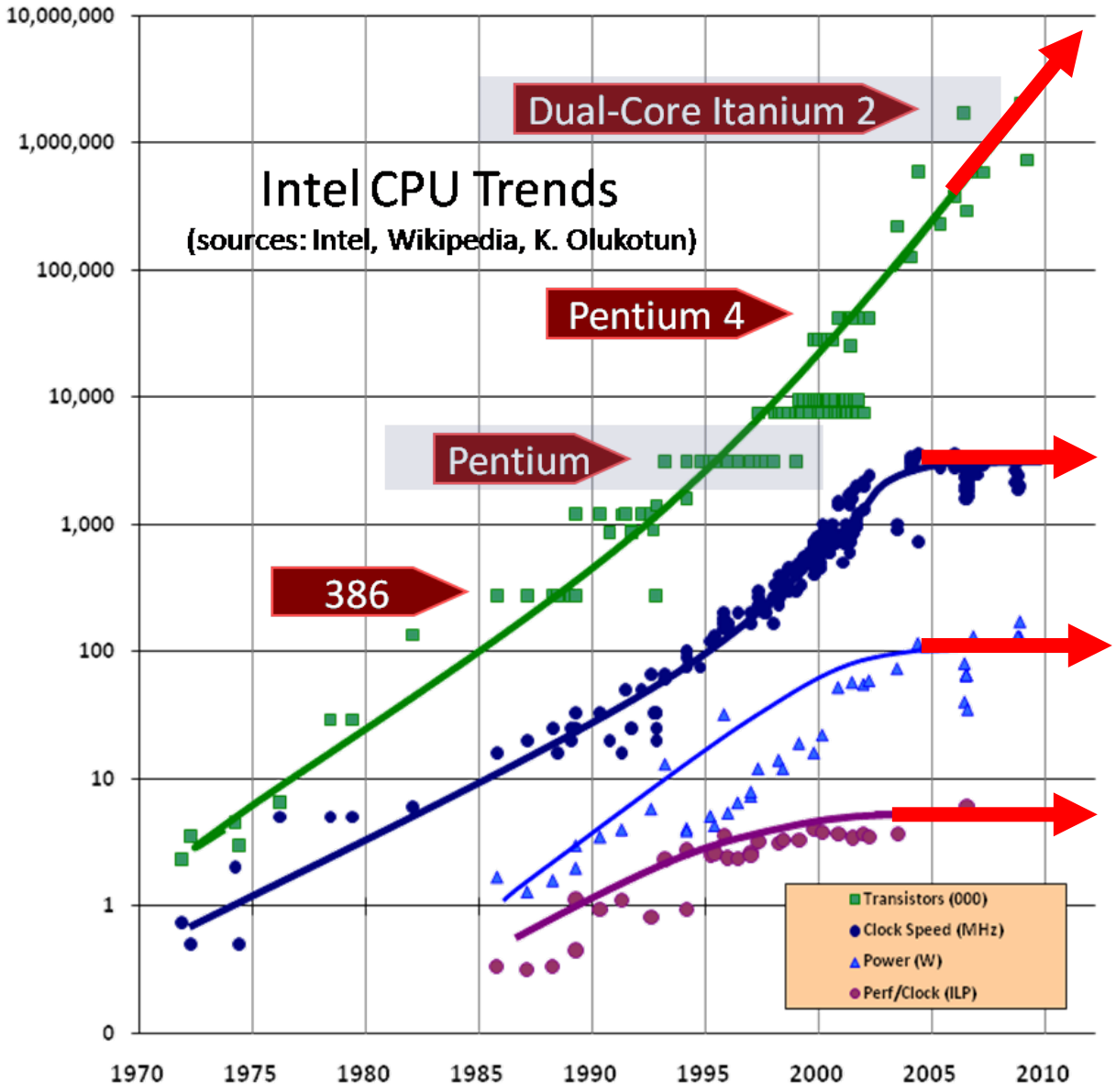
11/15/16

CS31 teaches you…

- How a computer runs a program.
  - How the hardware performs computations
  - How the compiler translates your code
  - How the operating system connects hardware and software
- The implications for you as a programmer
  - Pipelining instructions
  - Caching
  - Virtual memory
  - Process switching
  - Support for Parallel programming (threads)

Why do we care about parallel?

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



Moore's Law

- Circuit density (number of transistors in a fixed area) doubles roughly every two years.

- This used to mean that clock speed doubled too.
  - All your programs run twice as fast for free.
  - Problem: heat

- For now, circuit density is still increasing. How can we make use of it?

The “Multi-Core Era”

- We can’t make a single core go much faster.
- We can use the extra transistors to put multiple CPU cores on the chip.
- This is exciting: CPUs can do a lot more!
- Problem: it’s now up to the programmer to take advantage of multiple cores.
  - Humans are bad at thinking in parallel…

Parallel Abstraction

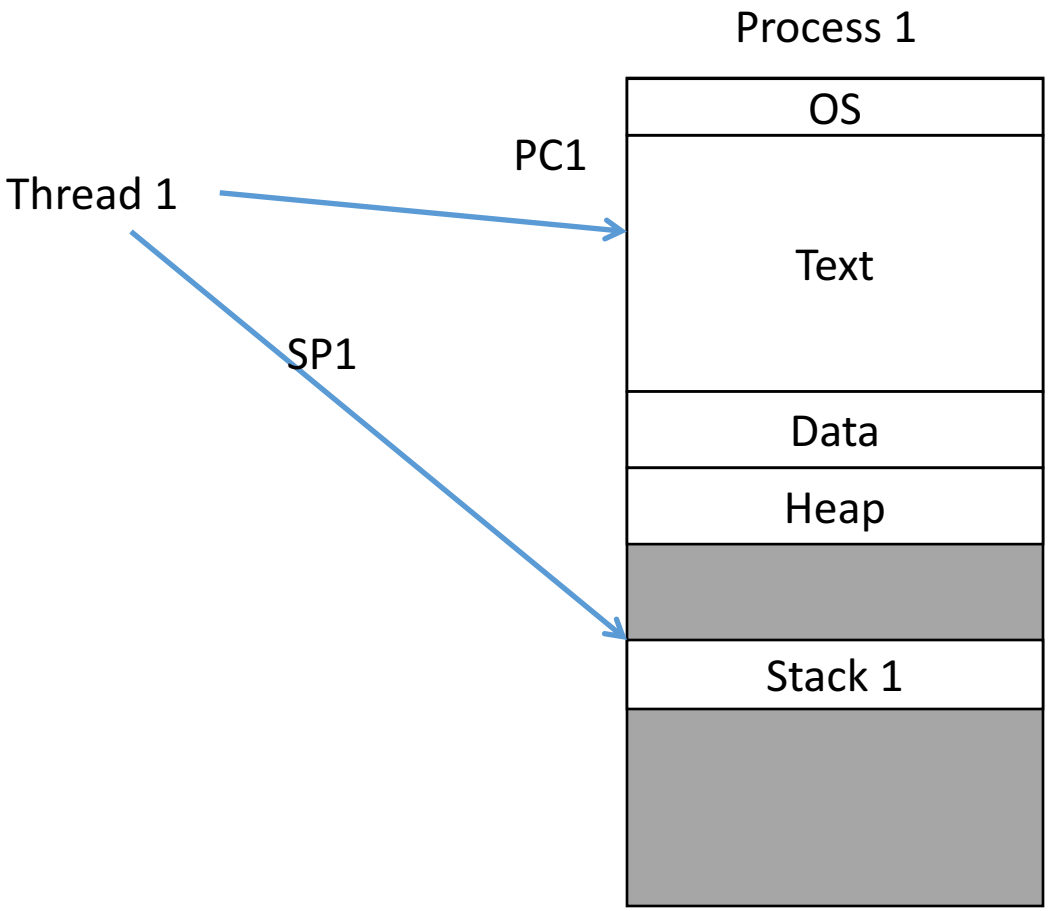
- To speed up a job, you have to divide it across multiple cores.
- A process contains both execution information and memory/resources.
- What if we want to separate the execution information to give us parallelism within a process?

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.

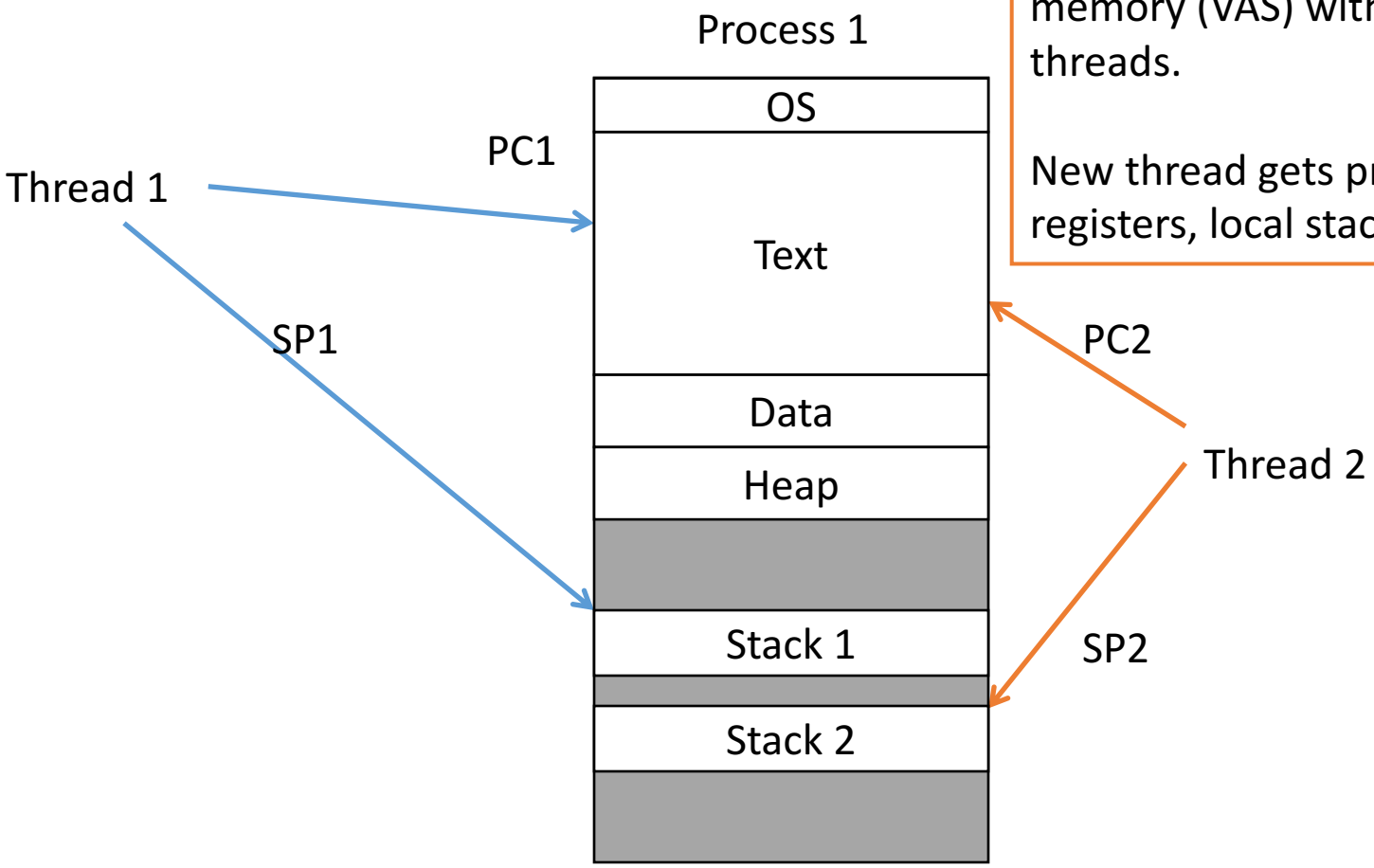
Threads



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

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

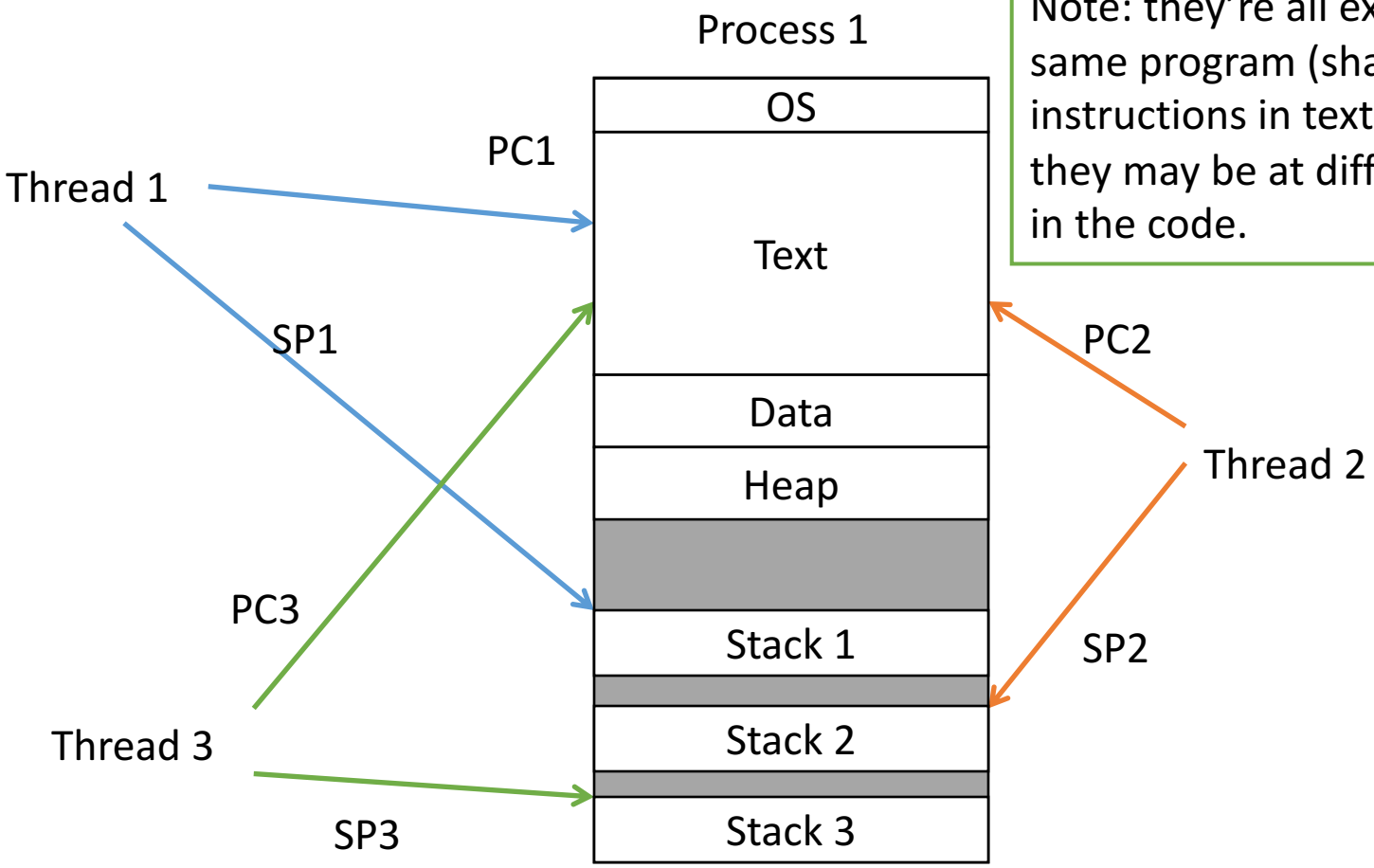
Threads



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

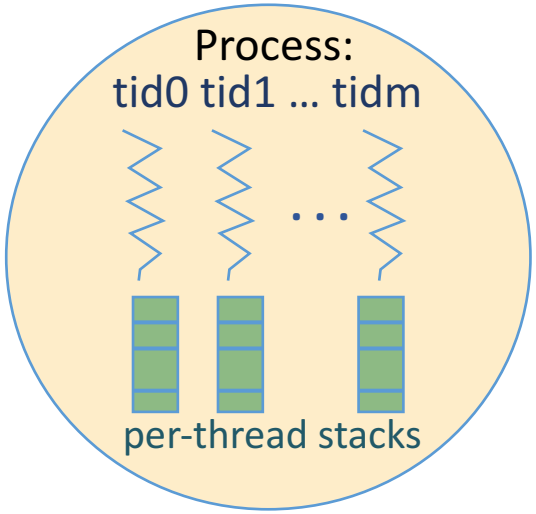


A third thread added.

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

Threads

- Private: tid, copy of registers, execution stack
- Shared: everything else in the process



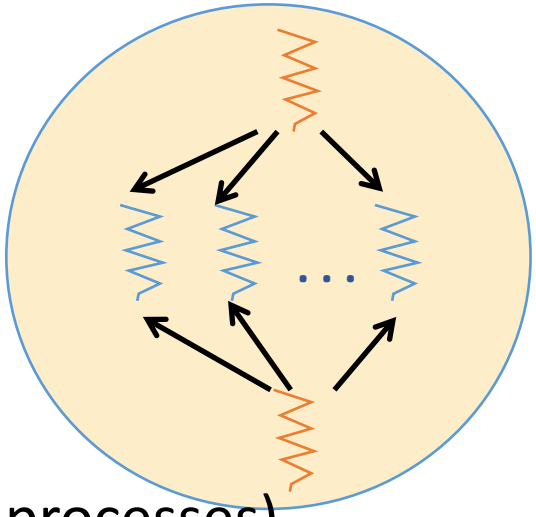
+ Sharing is easy
+ Sharing is cheap
  no data copy from one Pi's address space to another Pj's address space
+ Thread create faster than process
+ OS can schedule on multiple CPUs
  + Parallelism
– Coordination/Synchronization
  – How to not muck-up each other's state
– Can't use threads in distributed systems (when cooperating Pis are on different computers)

Programming Threads

Every Process has 1 thread of execution

- The single main thread executes from beginning

An example threaded program's execution:

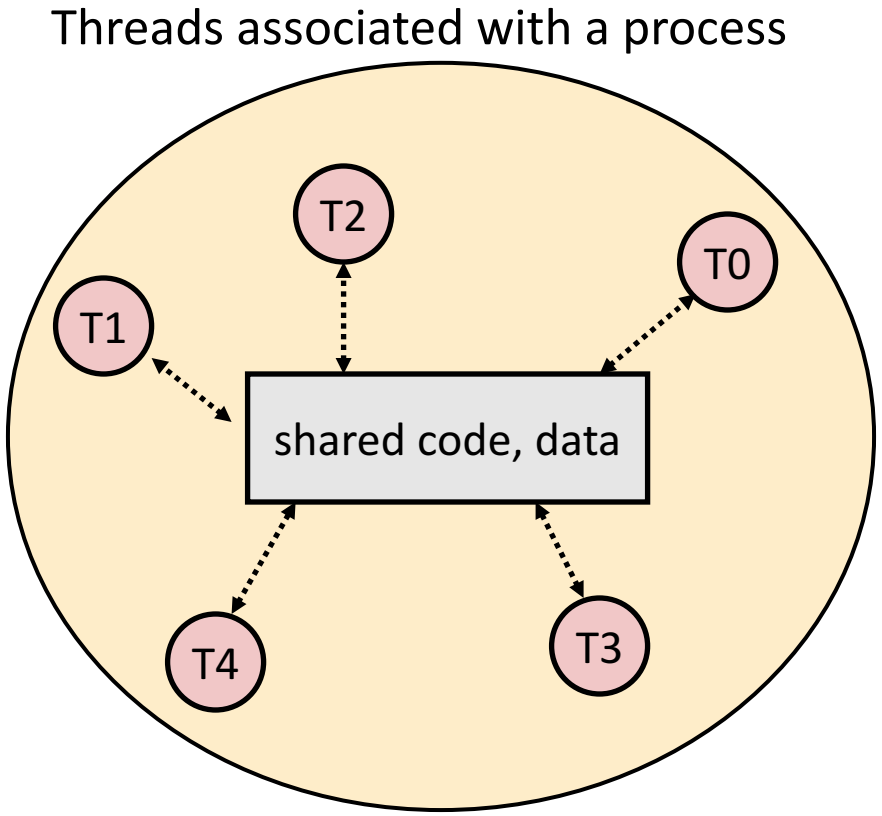


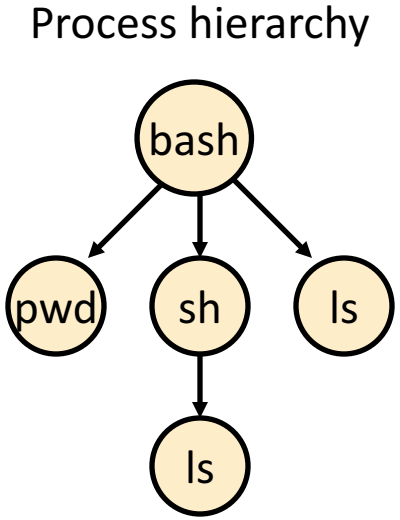
1. Main thread often initializes shared state
2. Then spawns multiple threads
3. Set of threads execute concurrently to perform some task
4. Main thread may do a join, to wait for other threads to exit (like wait & reaping processes)
5. Main thread may do some final sequential processing (like write results to a file)

Logical View of Threads

- Threads form a pool of peers w/in a process

  (Unlike processes which form a tree hierarchy)



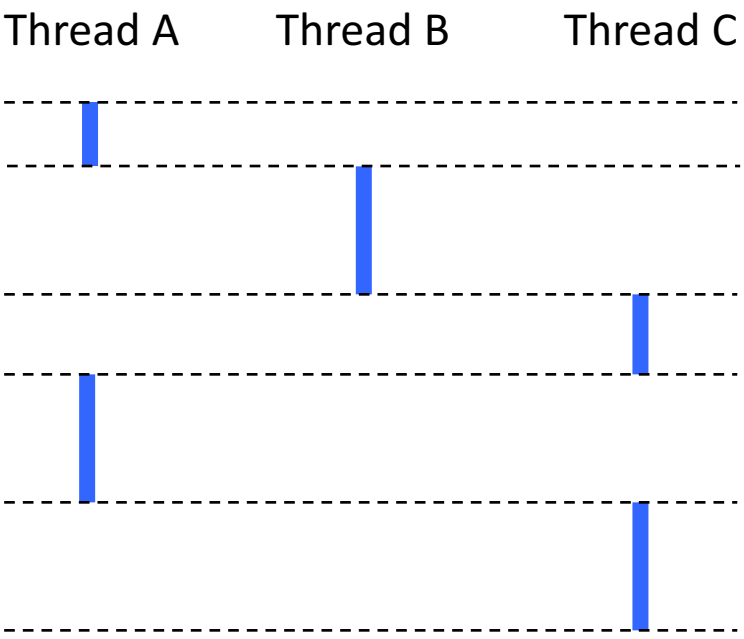


Thread Concurrency

Threads' Execution Control Flows Overlap

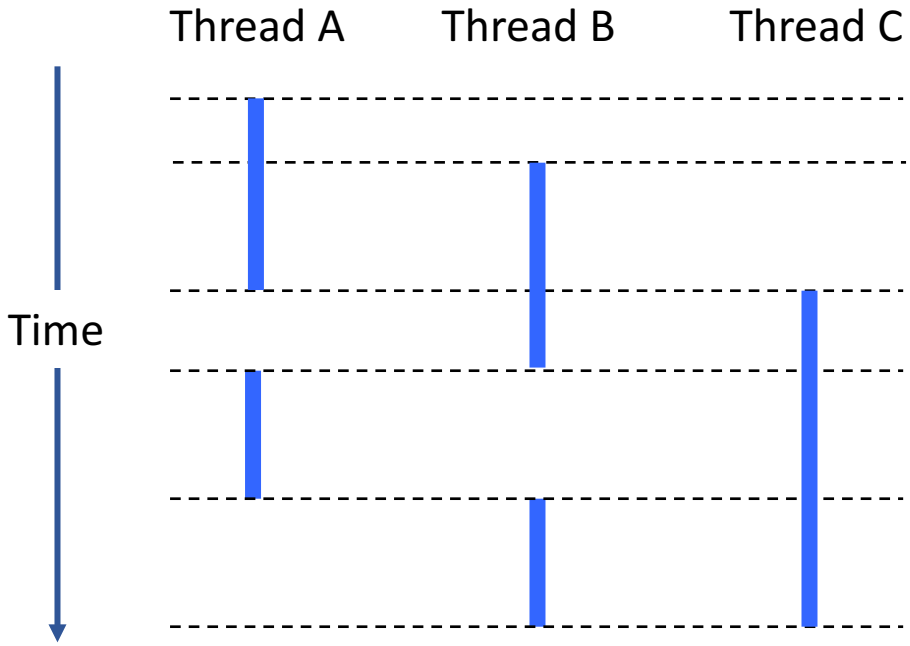
Single Core Processor

Simulate by time slicing



Multi-Core Processor

True concurrency



Run on multiple cores

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

Why use threads over processes?

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).

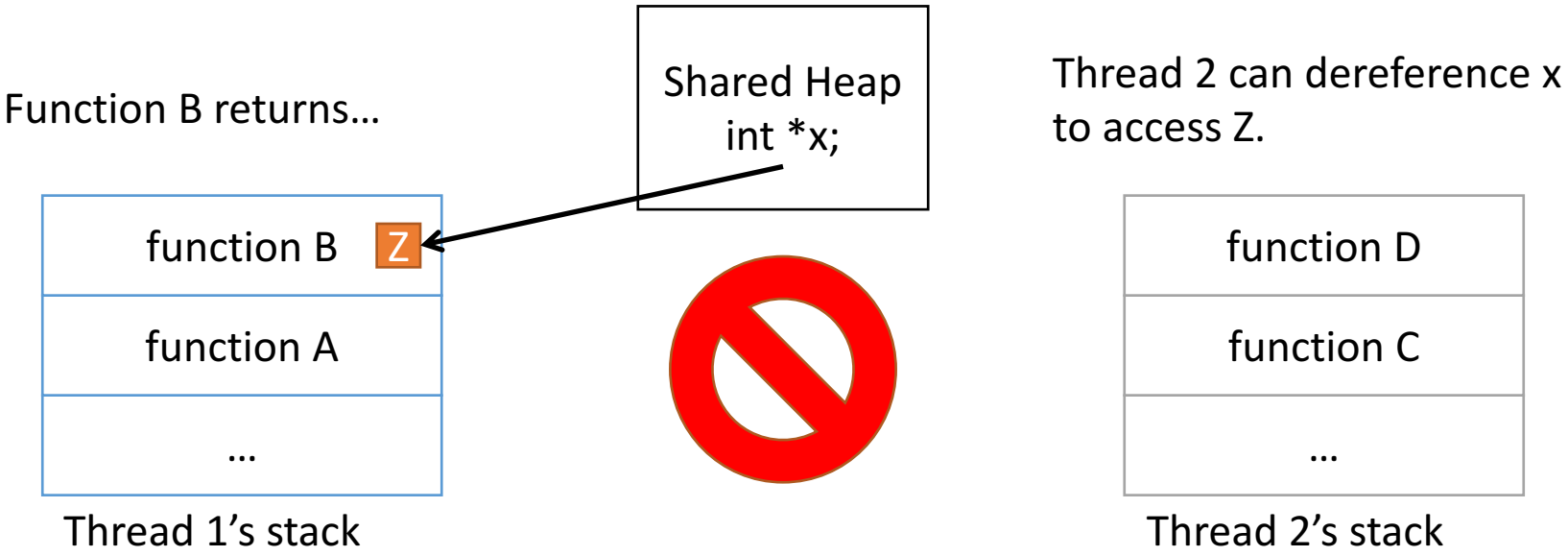
- Low-overhead sharing between threads in same process (threads share page tables, access same memory).

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 can BUT 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

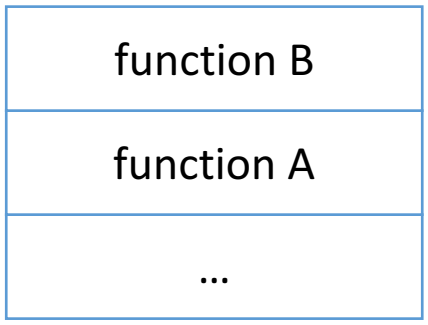
Threads & Sharing

- 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

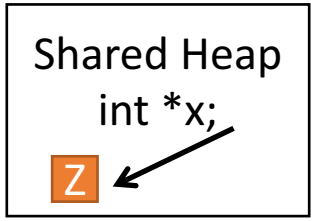


Threads & Sharing

- 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

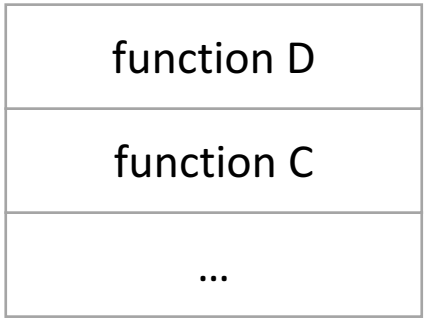


Thread 1's stack



Shared data on heap!

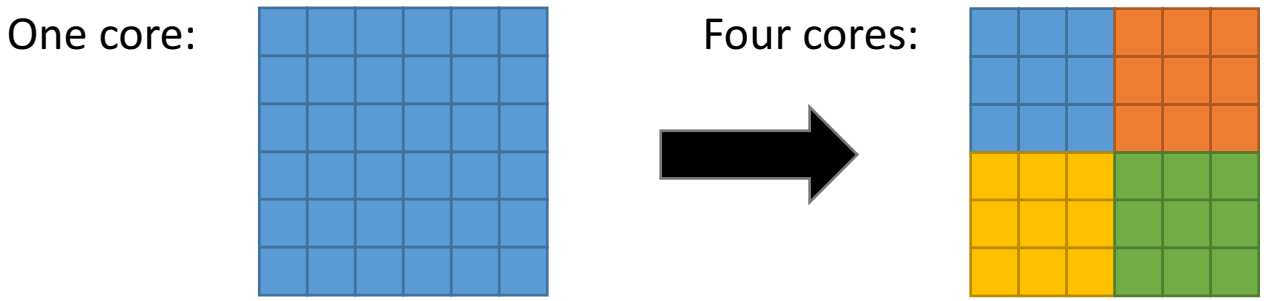
Thread 2 can dereference x to access Z.



Thread 2'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: processing an array



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

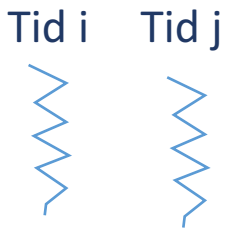
A. Slower than one core (<X)
B. The same speed (X)
C. Faster than one core, but not double (X-2X)
D. Twice as fast (2X)
E. More than twice as fast(>2X)

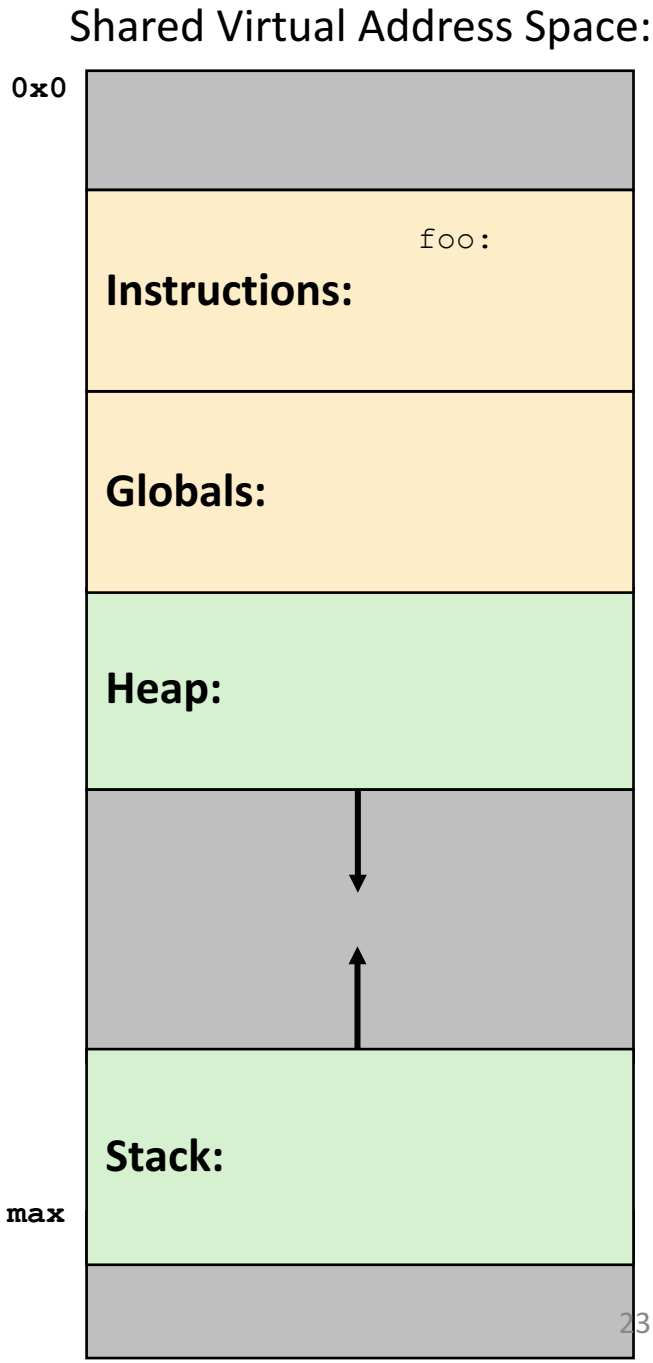
Parallel Speedup

- 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, diminishing returns.

Example

```
static int x;

int foo(int *p) {
    int y;

    y = 3;
    y = *p;
    *p = 7;
    x += y;
}
```





If threads i and j both execute function foo code:

Q1: which variables do they each get own copy of? which do they share?

Q2: which stmts can affect values seen by the other thread?

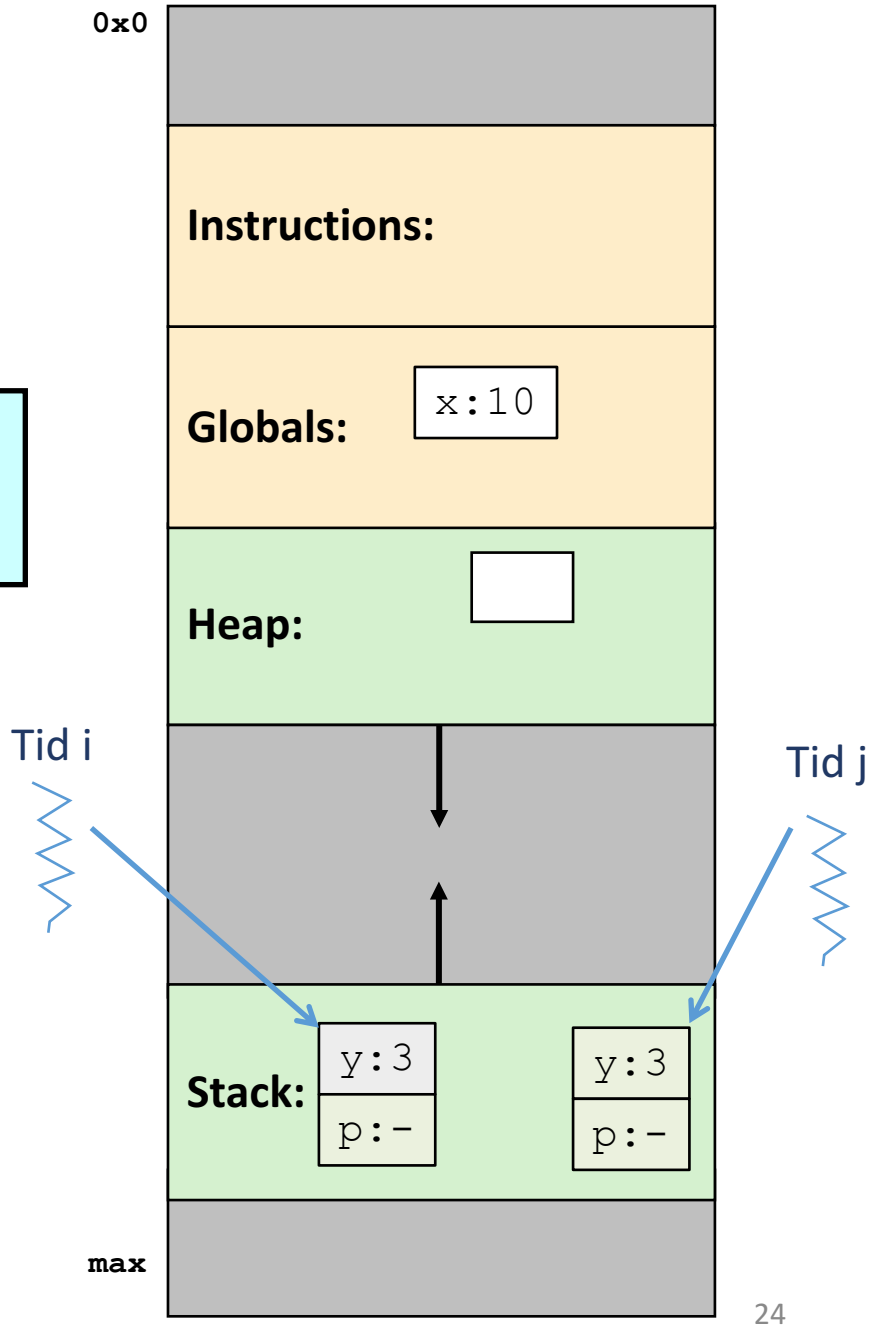
Example

```
static int x;

int foo(int *p) {
    int y;

    y = 3;
    y = *p;
    *p = 7;
    x += y;
}
```

Each tid gets its own copy of y on its stack

x is in global memory and is shared by every thread

p is parameter, each tid gets its own copy of p. However, p could point an int storage location: on the stack, or in global mem, or on the heap, or even in another's stack frame



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!