Agenda

Introduction to threads

- Parallel computing
- Multi-core

What is a thread

Programming with threads (pthreads)

Summary: Process

What is a process?

Advantages of processes:

- Abstraction: hardware easier to use for programmer/user
    - Maintains “lone view” of the system
    - Implements time sharing: each process gets small time slices of CPU
    - Implements virtual memory: simplifies programming model
- Efficiency:
    - Multiprogramming: 1+ running process at a time
    - Maximize use of hardware (disk/CPU)

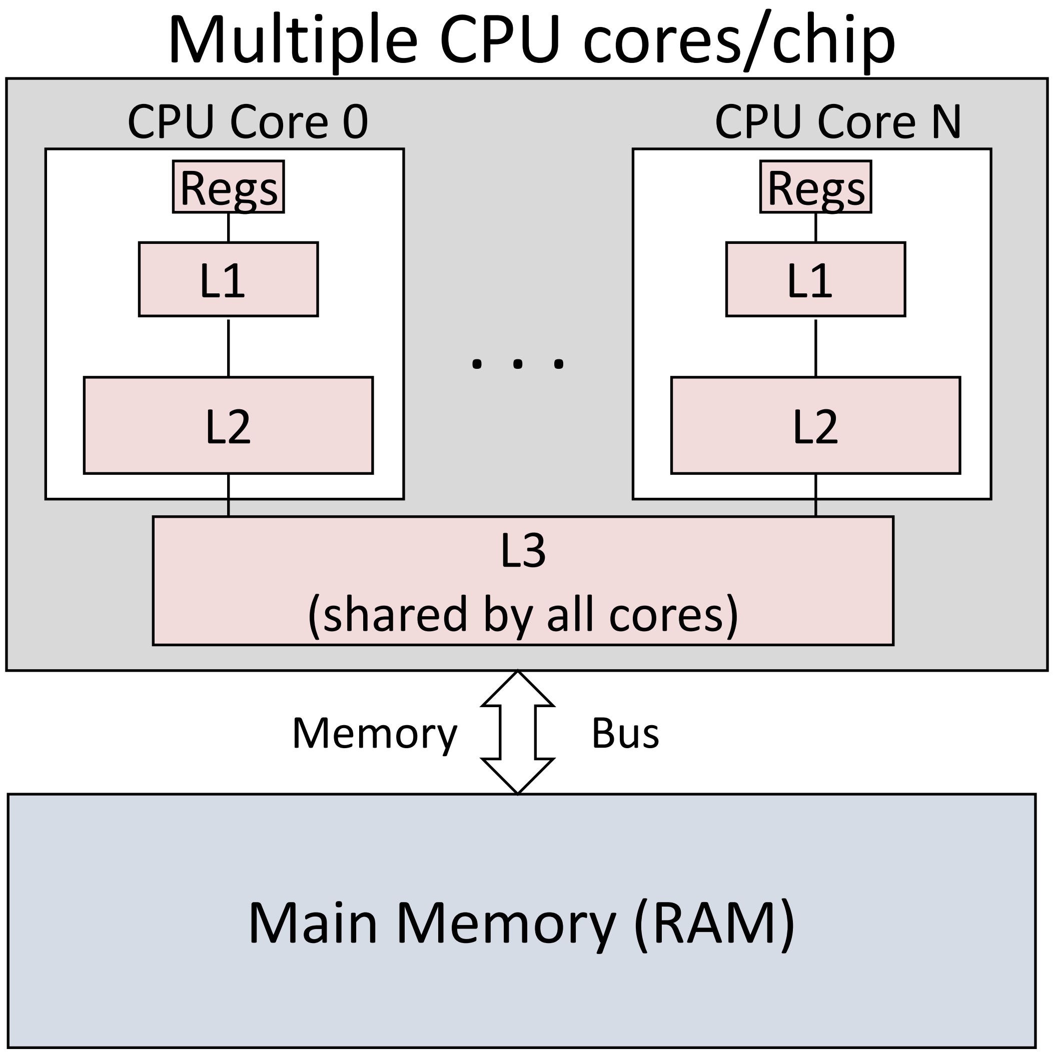
Threads

- Multiple processes running on a single CPU give illusion of concurrency on a single machine

- Modern CPUs have multiple cores -> true concurrency!

- Threads allow a single process to execute segments of code concurrently across cores
  - Unlike processes, all threads share the same virtual memory space
  - Like processes, the OS handles scheduling of threads across multiple cores

Modern processors are Multi-core



All CPU cores share same Main Memory

OS manages all cores and memory

RAM contains process' VM pages

Diagram by Tia Newhall and Andy Danner

Processes and Multicore

- OS schedules multiple processes to run simultaneously on separate CPU cores
  - On a 4-core processor, 4 concurrent processes can run at the same time, one per core
  - Get 4 times better throughput! (e.g. number of processes completed per hour)

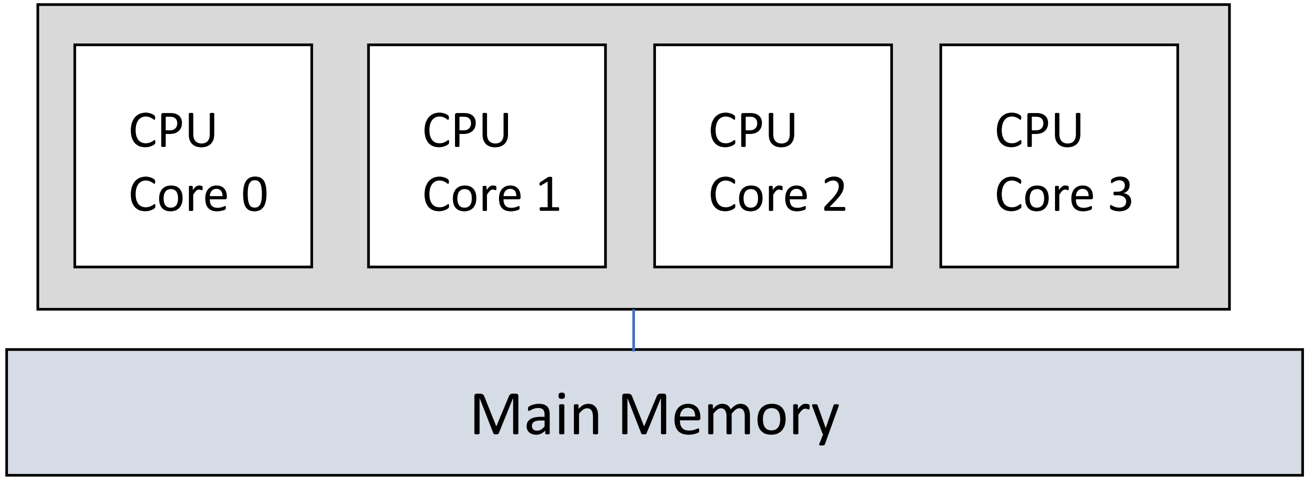


Diagram by Tia Newhall and Andy Danner

Parallel Computing

- Splits problem processing into segments that can be run concurrently

- Part of Making Programs Run Faster
  - Been around for decades for scientific computing
    - 1960s and 1970s first parallel machines
  - Prior to ~2005 most chips single core (1 CPU)
  - Now all (almost all) are multi-core
    - Every computer today is a parallel computer (N CPUs)

- "Era of Big Data"
  - Many more large computational problems across more disciplines
  - Increasingly require large parallel and distributed solutions

Why Multi-core?

Because Moore's Law hit the Power/Heat Wall

Historically:

- Moore's Law: the number transistors/chip doubles about every 2 years

  Transistor is building block on CPU (logic gates implemented with a few transistors)

  Twice transistors means ~twice improvement

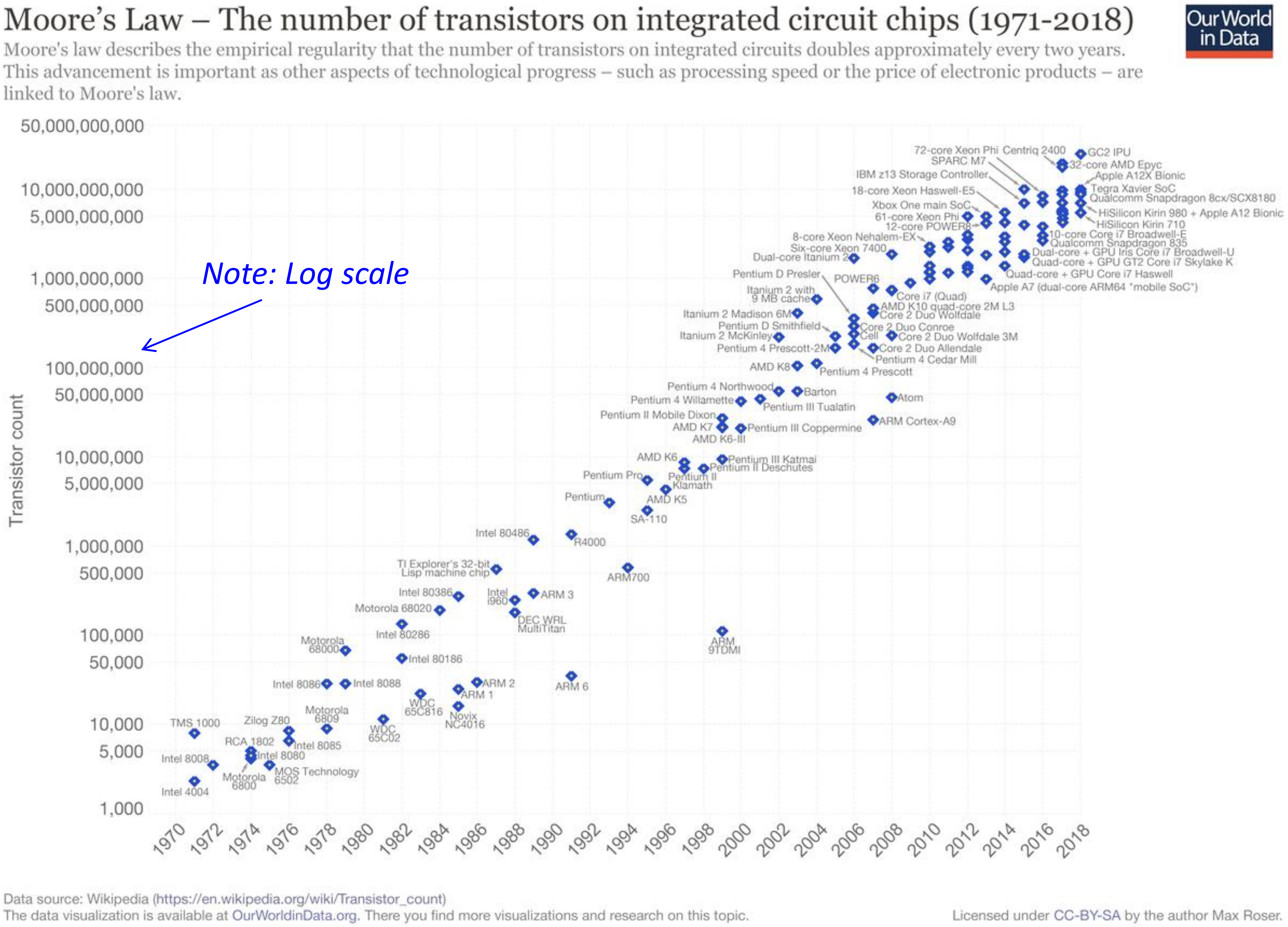
- Dennard Scaling: power use of these smaller transistors scales to area

  Increase CPU clock speed every 2 years without increasing power!

Why Multicore?

Moore's Law: still going strong!

The number of transistors/chip doubles about every 2 years!



Note: Log scale

How to Double CPU performance

How to get 2x faster the old way (before 2005): ILP

Instruction Level Parallelism in single CPU core

Implicit parallelism (+ hidden from programmer)

- need faster clock cycle to drive:

but end of Dennard Scaling around this time

too much heat: hit the heat/power wall

How the new way: Multi-core

Multiple, “simple” CPU cores on chip

Same clock speed each generation of chip

Explicit Parallelism to make single program run faster

- programmer and OS need to explicitly use them

Parallel Computing

Speed up a program by dividing it up into parts, each part runs in parallel on multiple CPUs

++ *10 hours on 1 CPU ➔ 6 mins on 100 CPUs!!! (maybe)*

- OS needs to implement abstractions to support (Threads)

- To schedule on multiple CPUs to get parallel execution
- To allow parallel components to share execution state

- Programmer must explicitly use

- Need a language/library for expressing parallelism
- Need to come up with parallelization strategy
  Parallel algorithm & assess efficiency of parallelization

Not always easy or even possible

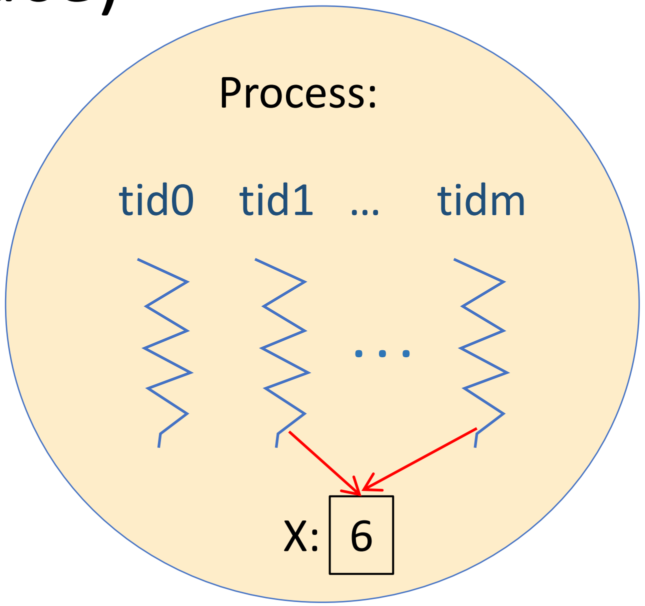
OS Abstraction: Thread

Thread: a stream of execution in a process

Multi-threaded process: multiple concurrent streams of execution in a single process

(all execute in a single shared virtual address space)

NOTE: writing a multi-threaded application is easier than a multi-process application because threads all have access to the same view of memory (processes need special mechanisms – pipes, shared memory, sockets – to coordinate)

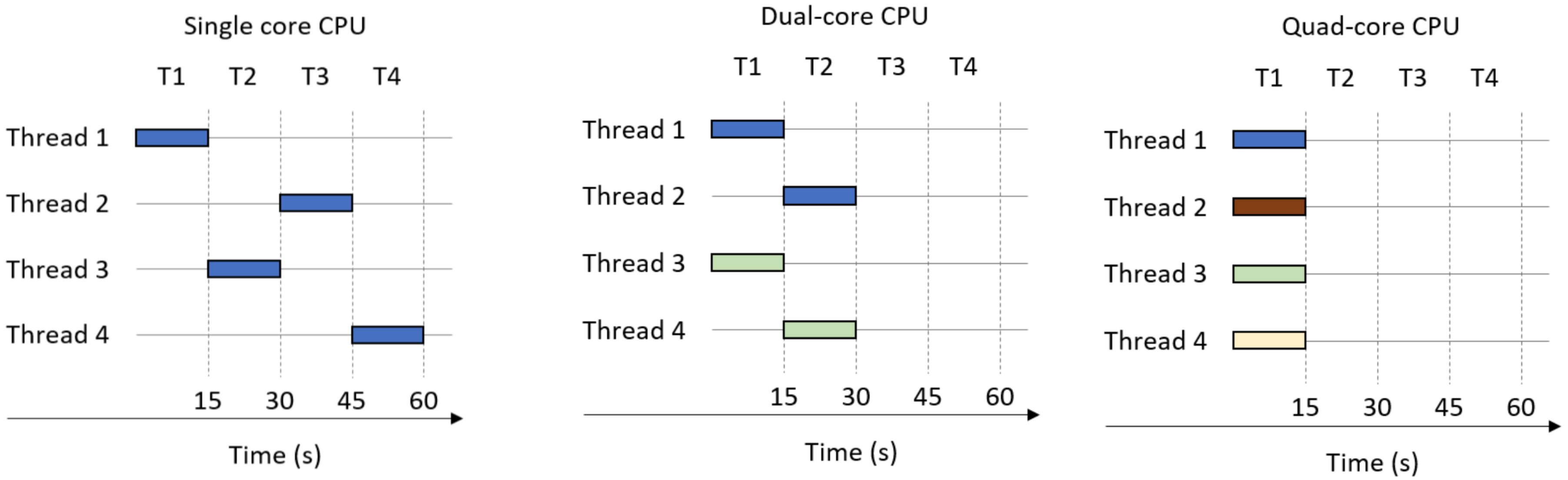


Threads Example

A **thread** is a stream of execution within a process

```
void scalar_multiply(int * array, long length, int s) {
  for (int i = 0; i < length; i++) {
    array[i] = array[i] * s;
  }
}
```

1. Create *t* threads.
2. Assign each thread a subset of the input array (i.e., *N*/*t* elements).
3. Instruct each thread to multiply the elements in its array subset by s



OS Abstraction: Thread

Threads are all part of a single Process:

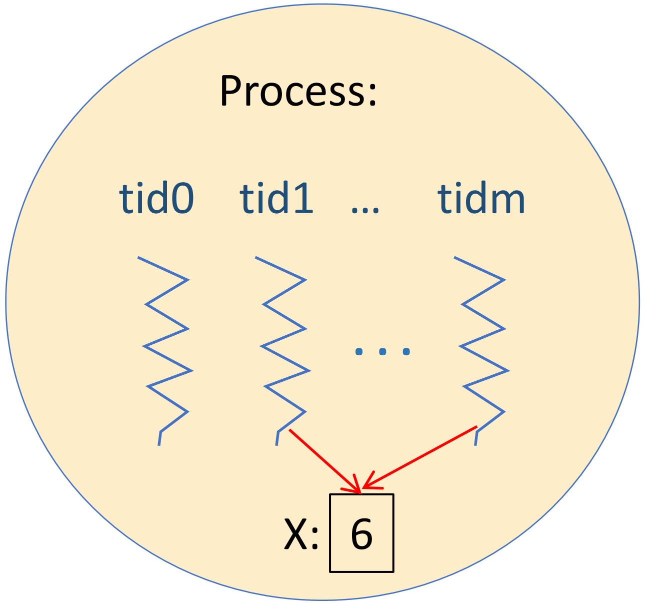
- Communicate using Shared Address Space

  thread 1: set x to 6 (write x)

  thread m: get value of x (read x)

- OS can schedule each thread to run on a different CPU on multi-core (can run simultaneously)

  + parallel execution of threads



Single Process Execution State

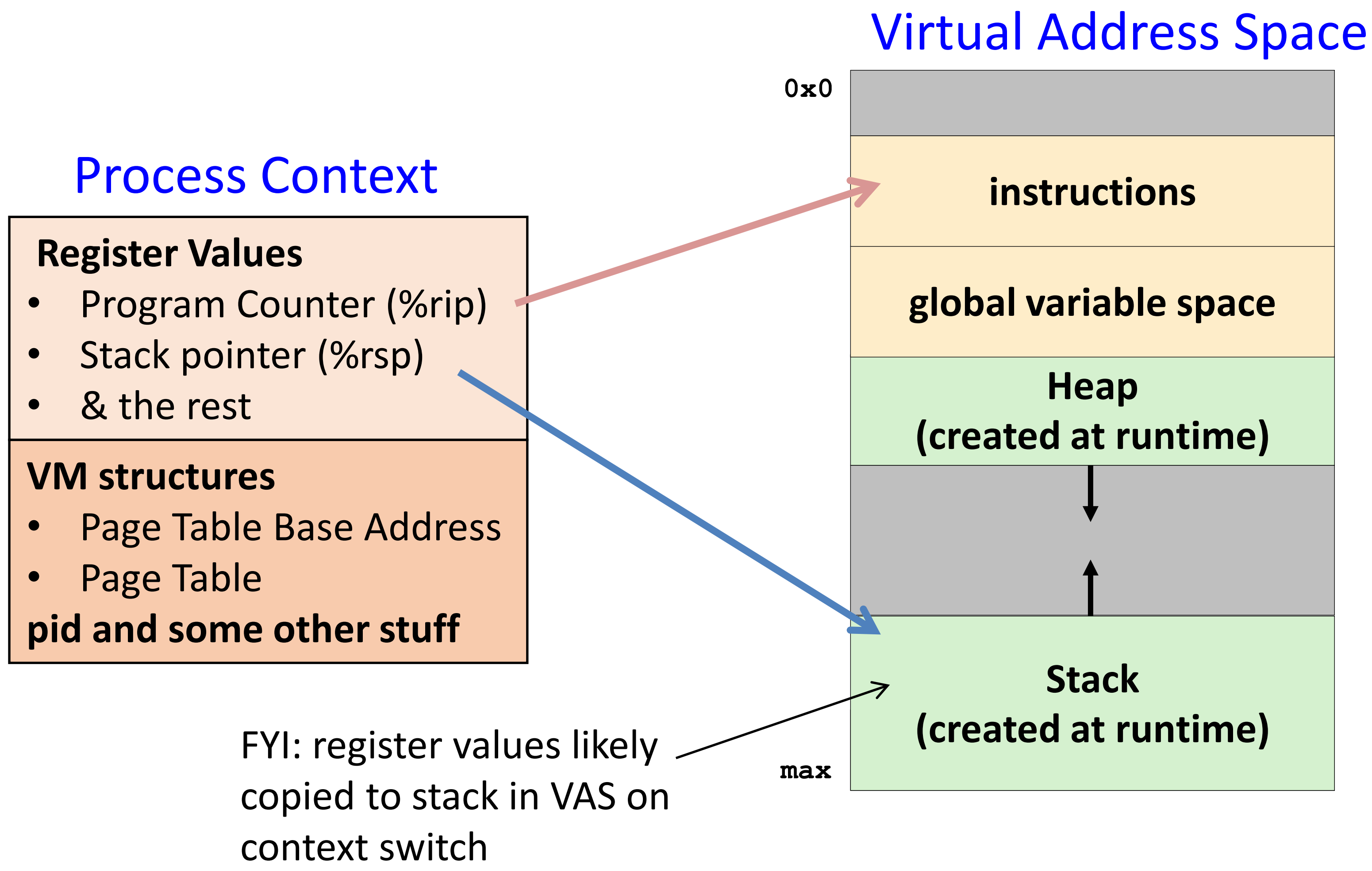


Diagram by Tia Newhall and Andy Danner

Multi-Threaded Processes

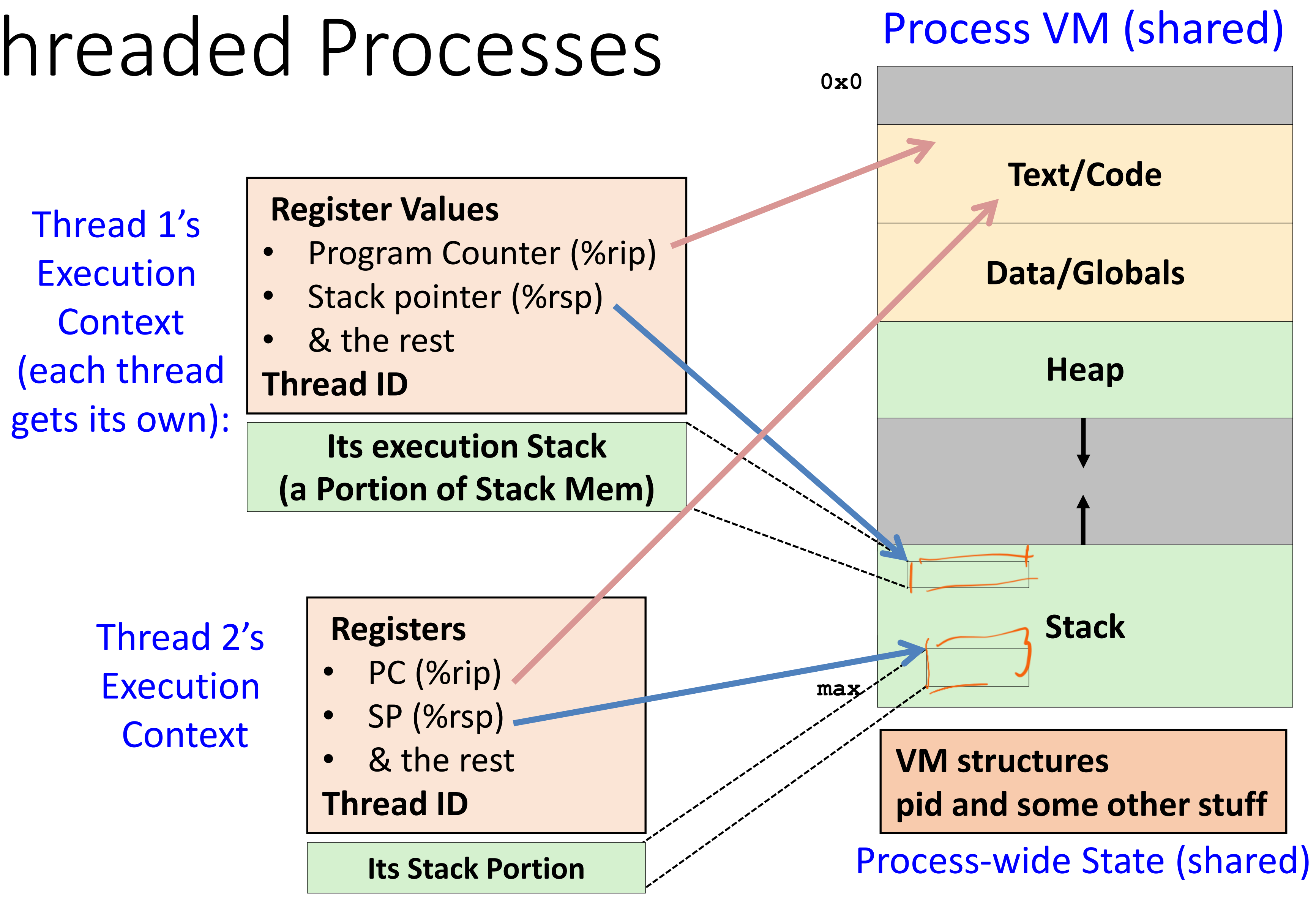
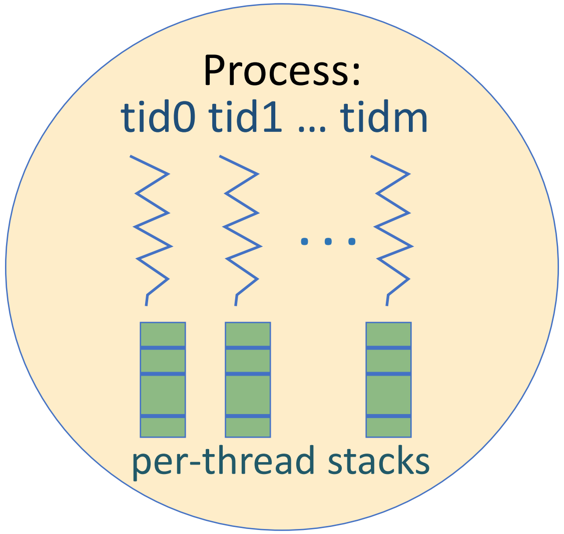


Diagram by Tia Newhall and Andy Danner

Threads

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



+ Sharing is easy
+ Sharing is cheap

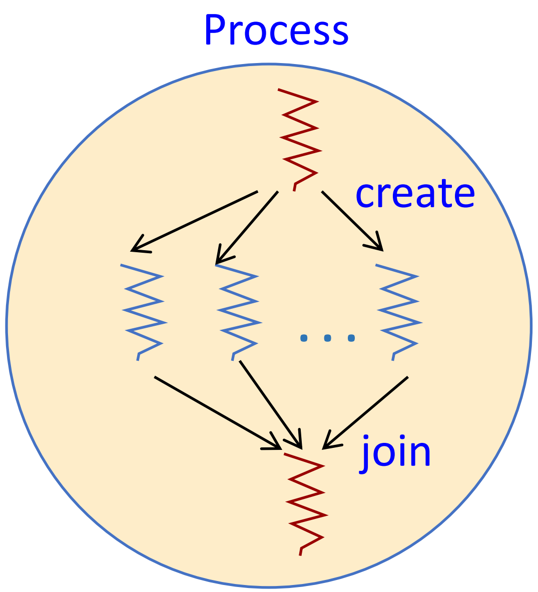
  no data copy from one Pi's virtual address space
  to another Pj's virtual address space
+ Thread create & CXS faster than process

  All share same virtual address space, page table
+ OS can schedule on multiple CPUs of multicore
  + Parallelism
- Coordination/Synchronization
  - How to not muck-up each other's state
- Can only use on systems with shared physical memory
  (can't if cooperating Pis are on different computers)

Programming Threads

Every Process has 1 thread of execution

- The single main thread executes from beginning



A common threaded execution model:

1. Main thread often initializes shared state first
2. Then spawns/creates 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 (~`wait` to reap exited threads)
5. Main thread may do some final sequential processing (like write results to a file)

Demo: HelloThread

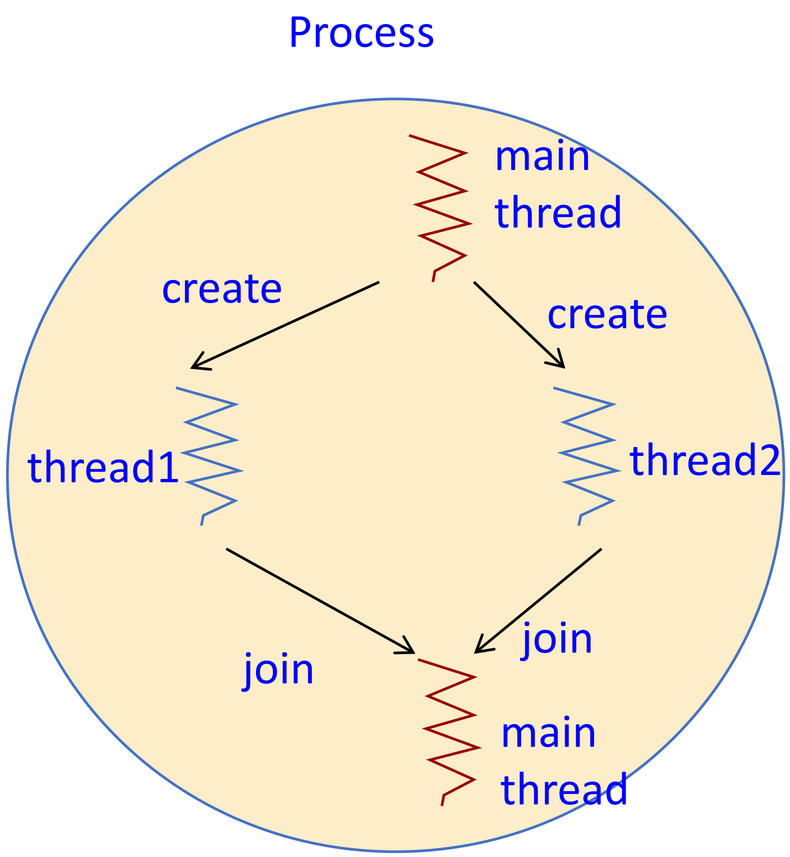
```c
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1 = 1, id2 = 2;
  long* retval1 = NULL, *retval2 = NULL;
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, HelloWorld, &id1);
  pthread_create(&thread2, NULL, HelloWorld, &id2);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

```
gcc thread-hello.c -lpthread
```

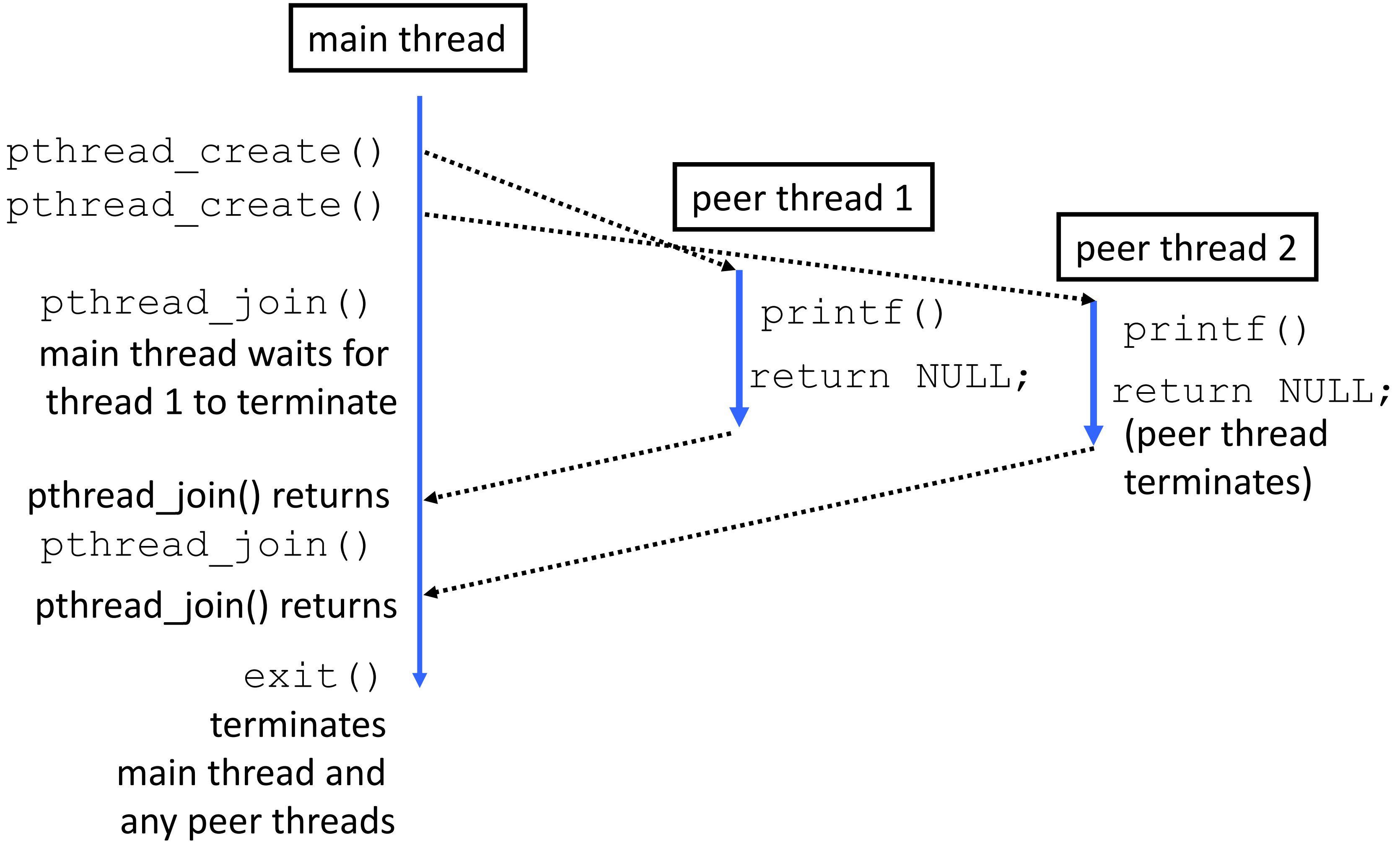
Visualizing HelloThread

Which lines of code are executed by which thread?

- main thread executes lines 12-16
- each thread executes lines 6-8
- main thread waits for thread 1 (line 17)
- main thread waits for thread 2 (line 18)
- main thread returns and exits (line 19)



Visualizing HelloThread: concurrent execution



Common pthread functions

Creating a thread (starts running start_func w/passed args):

```
int pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*start_func)(void *),
                   void *args);
```

Joining (reaping) a thread (caller waits for thread to exit):

```
int pthread_join(pthread_t thrd, void **retval);
```

Terminating a thread:

```
void pthread_exit(void *retval)
```

(or just `return` from thread's main function)

Pthreads

PThreads: The POSIX threading interface

- The **P**ortable **O**perating **S**ystem **I**nterface for UNI**X**
- A standard Interface to OS utilities

  system calls have same prototype & semantics on all Oss

  (ex) POSIX compliant code on Solaris will compile on Linux

Pthreads library contains functions for:

- Creating threads (and thread exit)
- Synchronizing threads
  - Coordinating their access to shared state

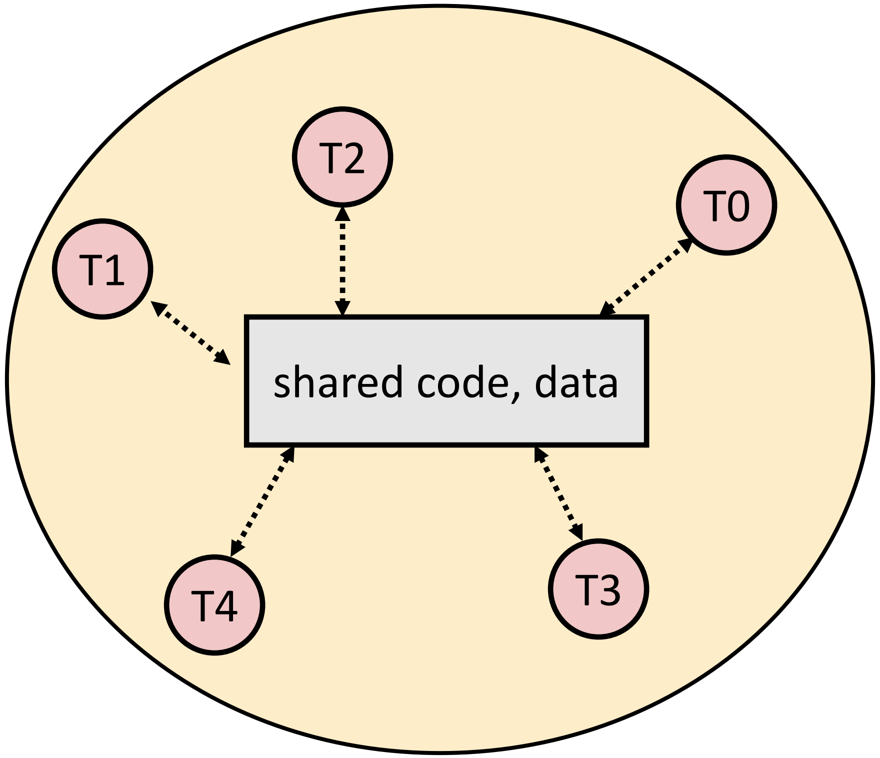
To compile: `gcc myprog.c -lpthread`

Logical View of Threads

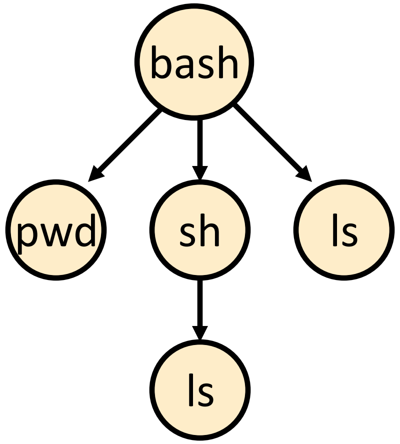
- Threads form a pool of peers w/in a process

  (Unlike processes which form a tree hierarchy)

Threads within one process



Process hierarchy



Draw a stack diagram: HelloThread

```c
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1 = 1, id2 = 2;
  long* retval1 = NULL, retval2 = NULL;
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, HelloWorld, &id1);
  pthread_create(&thread2, NULL, HelloWorld, &id2);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

Visualizing Thread execution: HelloThread

```
void *HelloWorld(void *id) {
  long *myid = (long *) id;
  printf("Hello world! I am thread %ld\n", *myid);
  return NULL;
}

int main(int argc, char **argv) {
  long id1 = 1, id2 = 2;
  long* retval1 = NULL, retval2 = NULL;
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, HelloWorld, &id1);
  pthread_create(&thread2, NULL, HelloWorld, &id2);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

Summary: Comparing fork() vs threads

Fork()

- Each process has its own memory
- Programmer must decide how to split work and coordinate processes
  - External mechanisms: shmem, pipes, sockets, wait, signals
- Concurrent code has no guarantees on the order it runs

Threads

- Each thread shares memory in same process
- Programmer must decide how to split work and coordinate threads
  - Built-in mechanisms: mutex, barrier, wait
- Concurrent code has no guarantees on the order it runs

Exercise: Design a multi-threaded program

Parallel Compute Min

- Large array of size N of int values
- M threads   (assume N >> M), and N %M is 0

Some Questions to Consider:

1. What part of total operation does each thread do?

2. How does a thread know which work it will do?

3. Do threads need to coordinate their actions in any way?

4. What global/shared state do you need? What local state?

Design

1. What part of total operation does each thread do?

2. How does a thread know which work it will do?

3. Do threads need to coordinate their actions in any way?

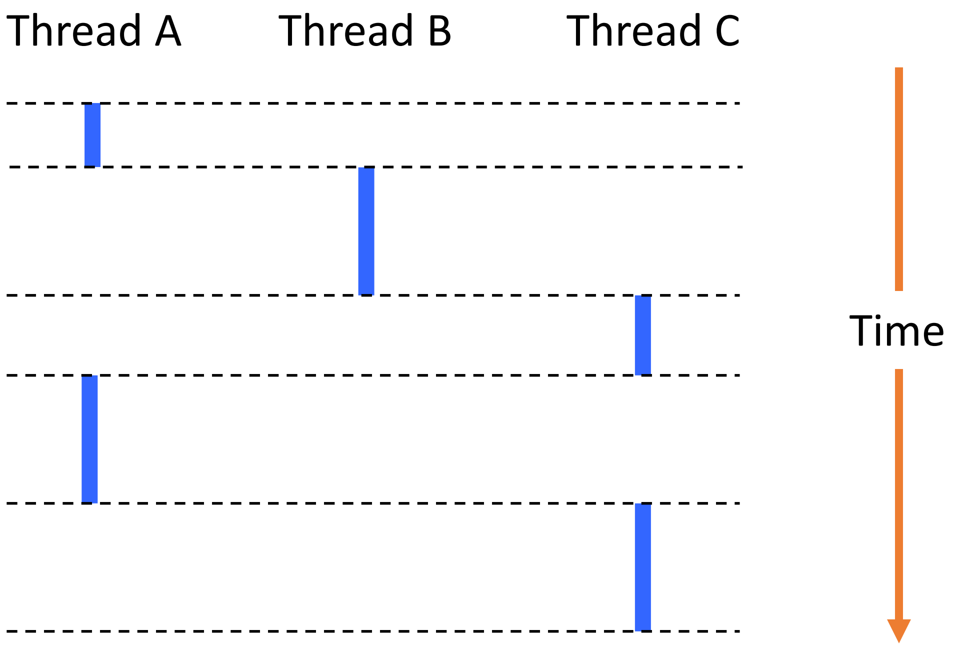
4. What global/shared state do you need? What local state?

Thread Concurrency

Threads' Execution Control Flows Overlap

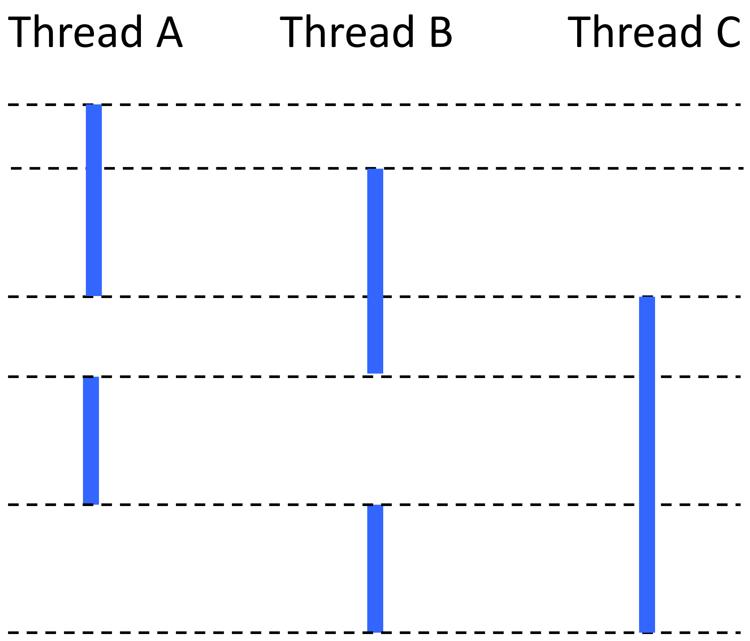
<u>Single Core Processor</u>

Concurrent w/time slicing



<u>Multi-Core Processor</u>

True parallel execution



Run on multiple cores

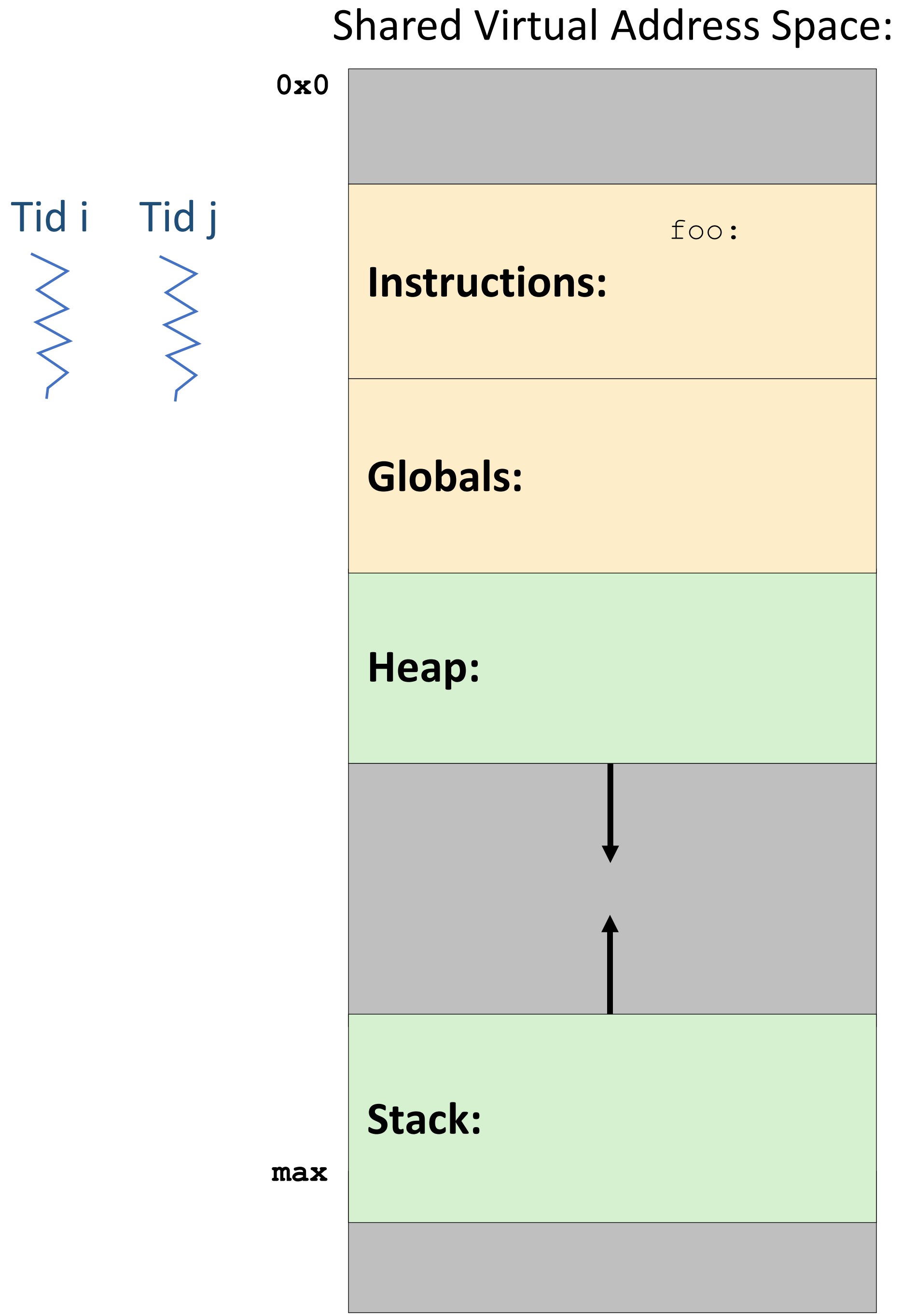
Exercise

```
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?



Exercise: Draw a stack diagram

```
static int x;

int foo(int *p) {
   int y;

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

int main() {
 int pvalue = 35;
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, foo, &pvalue);
 pthread_create(&thread2, NULL, foo, &pvalue);
 pthread_join(thread1, NULL);
 pthread_join(thread2, NULL);
 return 0;
}
```

Stack Diagram: Possibility 1

```
static int x = 0;

int foo(int *p) {
   int y;

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

int main() {
 int pvalue = 35;
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, foo, &pvalue);
 pthread_create(&thread2, NULL, foo, &pvalue);
 pthread_join(thread1, NULL);
 pthread_join(thread2, NULL);
 return 0;
}
```

Stack Diagram: Possibility 2

```
static int x = 0;

int foo(int *p) {
   int y;

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

int main() {
 int pvalue = 35;
 pthread_t thread1, thread2;
 pthread_create(&thread1, NULL, foo, &pvalue);
 pthread_create(&thread2, NULL, foo, &pvalue);
 pthread_join(thread1, NULL);
 pthread_join(thread2, NULL);
 return 0;
}
```