# CS 31: Introduction to Computer Systems

## 23 OS Processes and Parellelism

04-17-2025

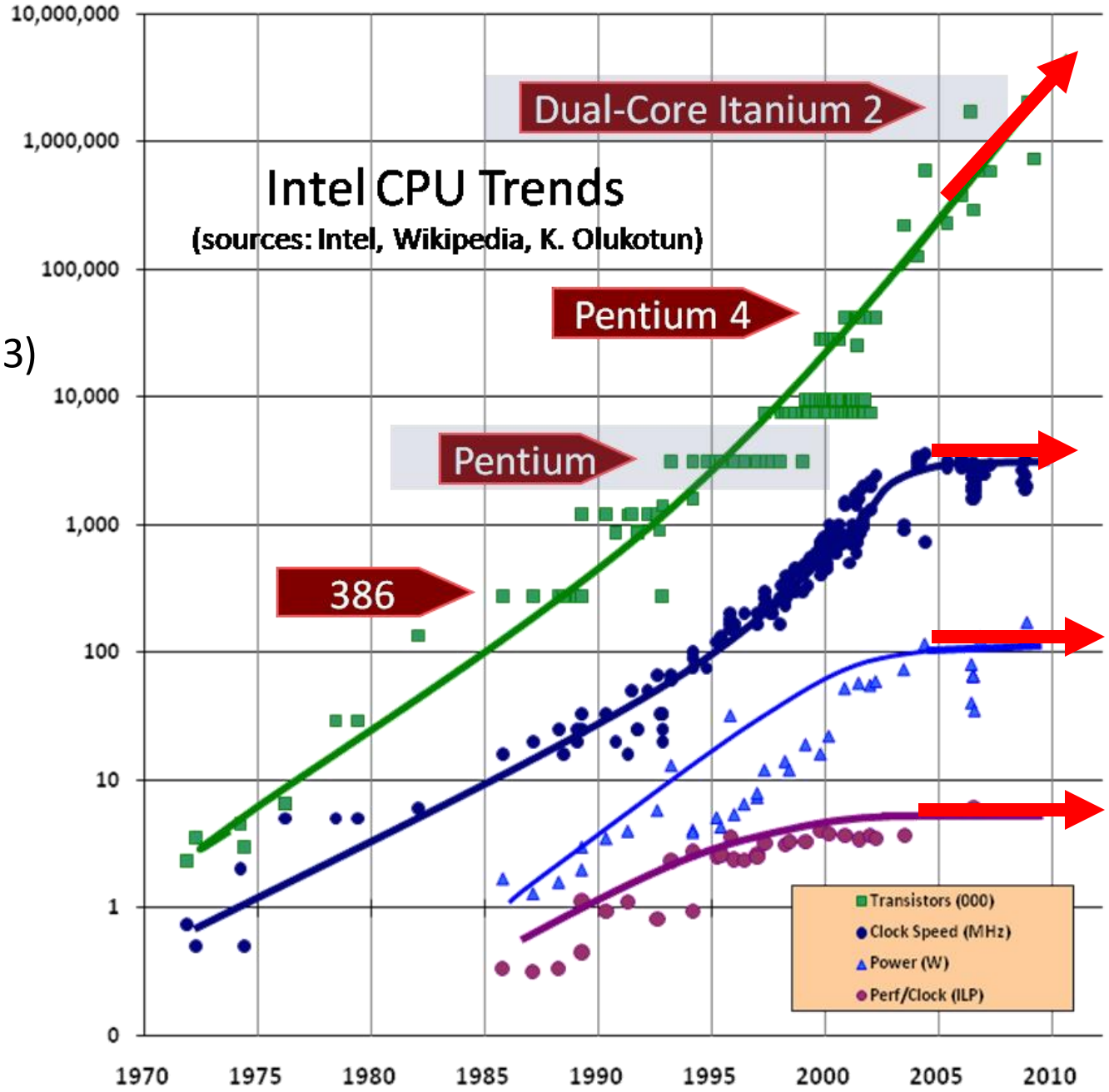
# Reading Quiz

# Processor Design Trends

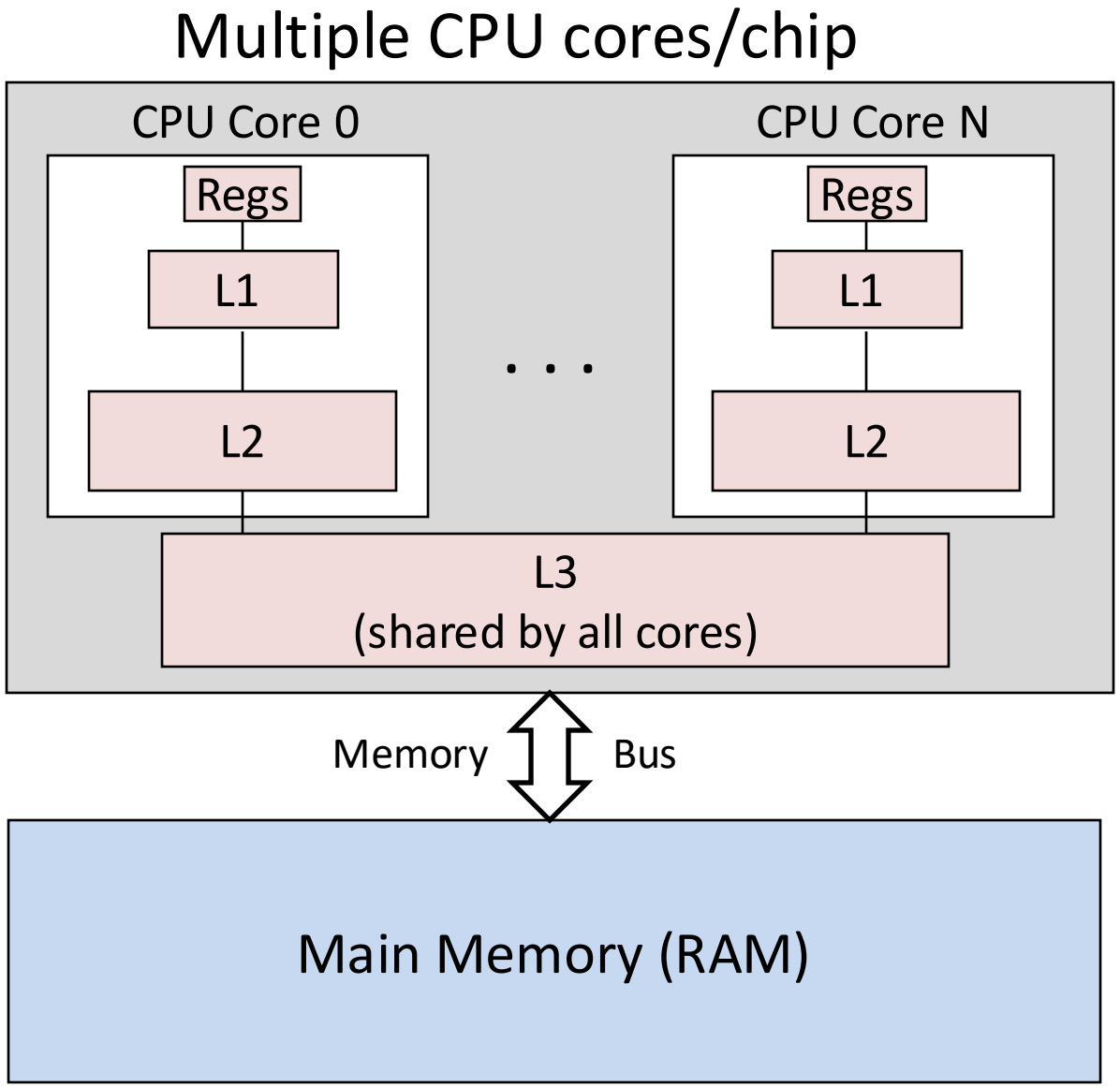
- Transistors (*10^3)
- Clock Speed (MHZ)
- Power (W)
- ILP (IPC)
  Instruction Level Parallelism

From Herb Sutter,

Dr. Dobbs Journal



# Today's Processors are Multi-core




All CPU cores share same Main Memory

OS manages all cores and memory

RAM contains processes' VM pages

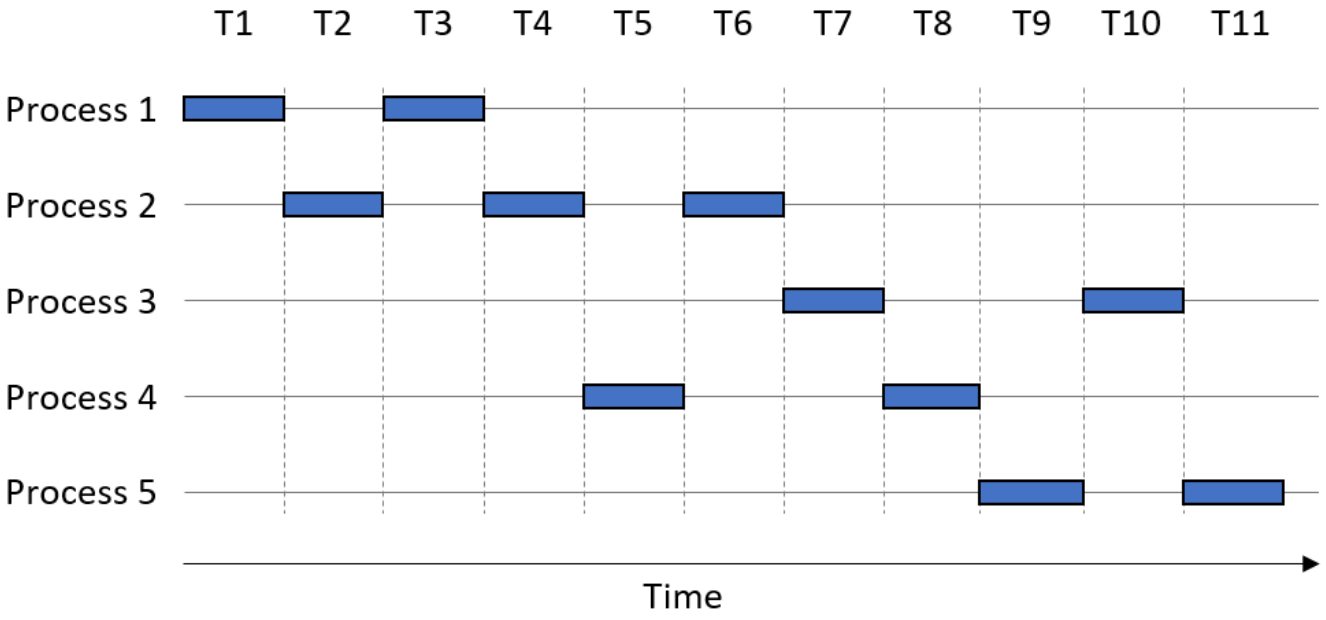
# Impact of multicore systems on process execution

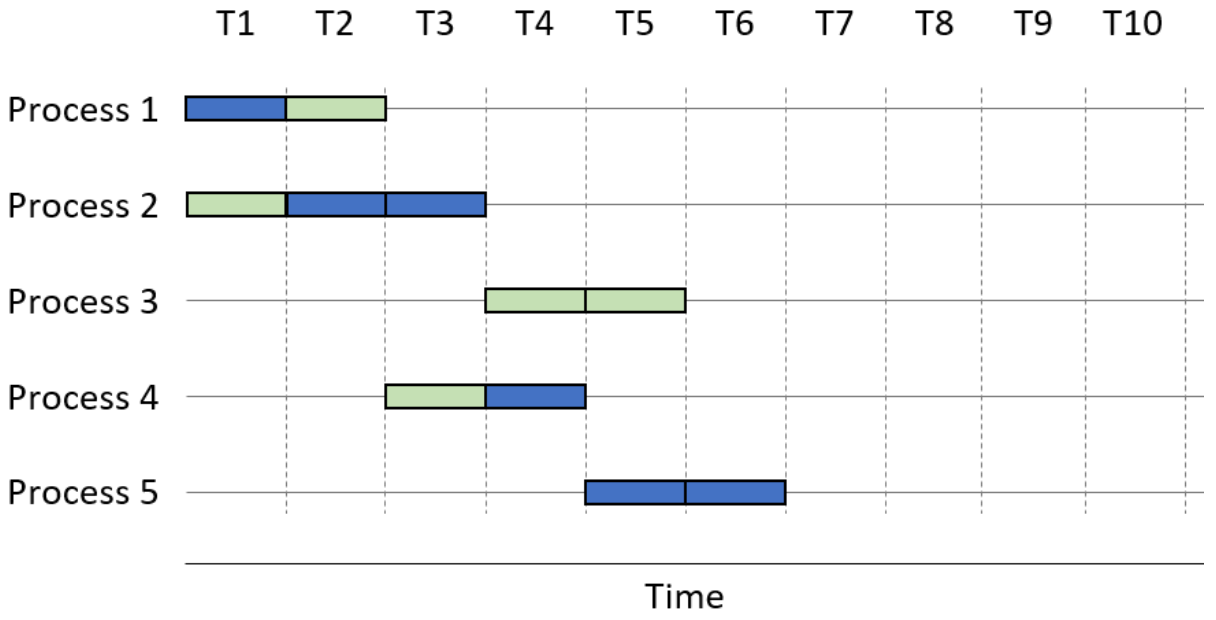
An example of five processes running on a single-core CPU, via context switching.

The mechanism behind multiprogramming determines how the OS swaps one process running on the CPU with another.

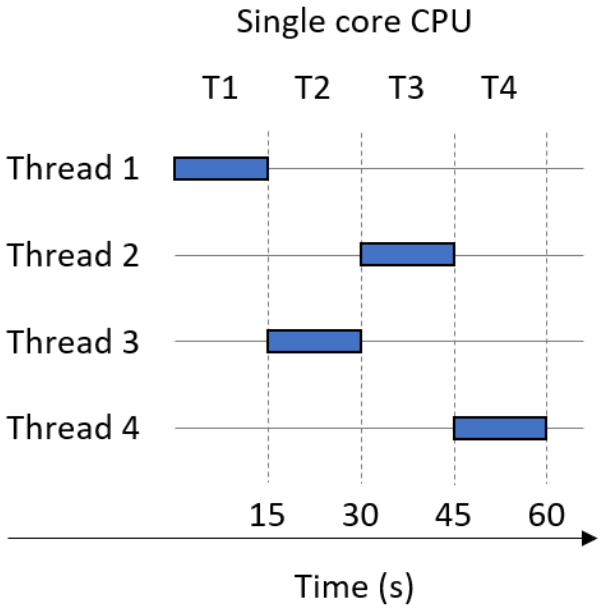
The policy aspect of multiprogramming governs scheduling the CPU, or picking which process from a set of candidate processes gets to use the CPU next and for how long.

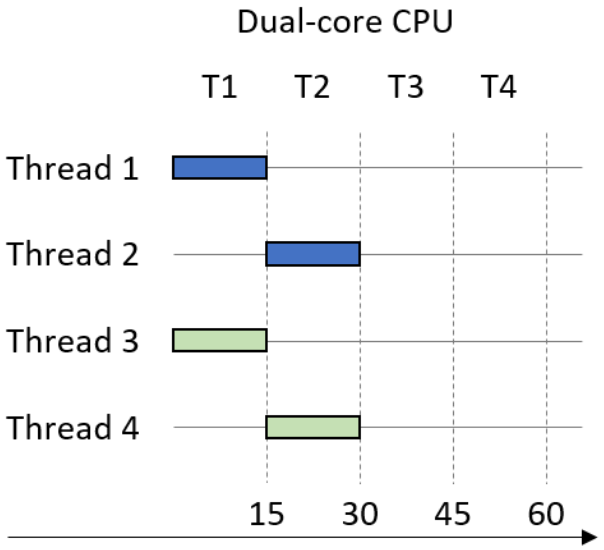
A multicore CPU enables the OS to schedule a different process to each available core, allowing processes to execute simultaneously. *The simultaneous execution of instructions from processes running on multiple cores is referred to as parallel execution.*

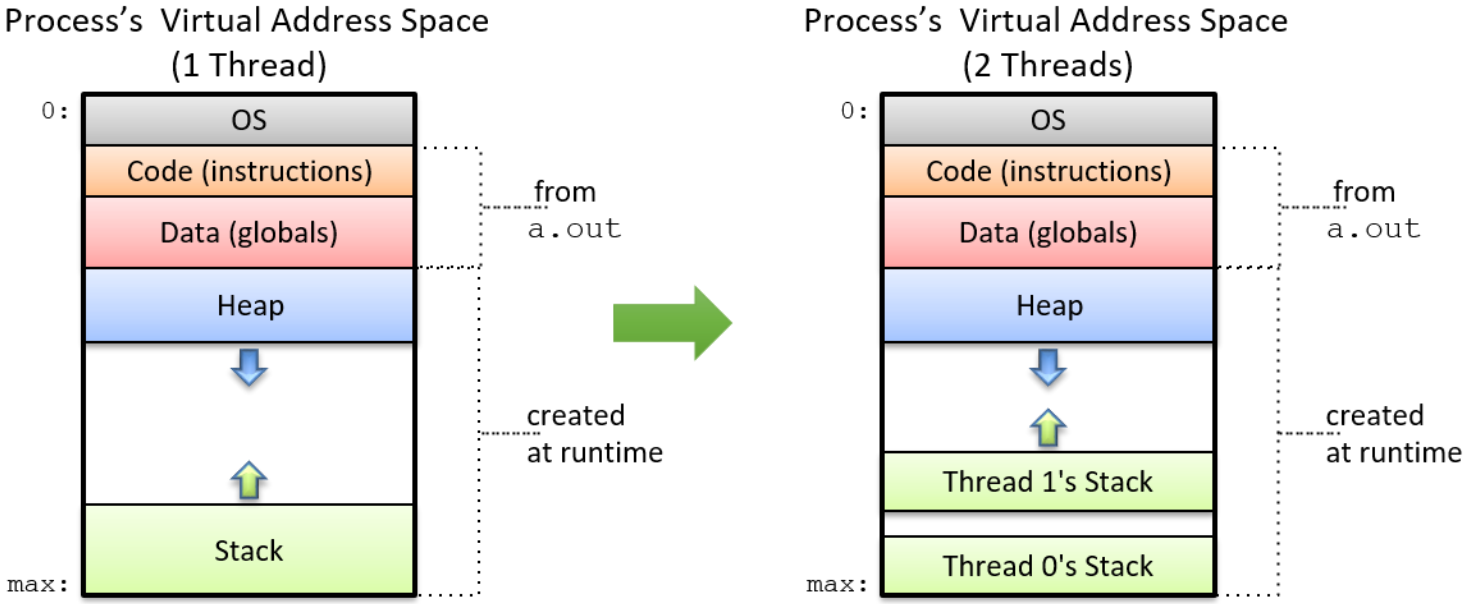


# Expediting process execution with threads




- 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

# 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 – 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?)

# 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

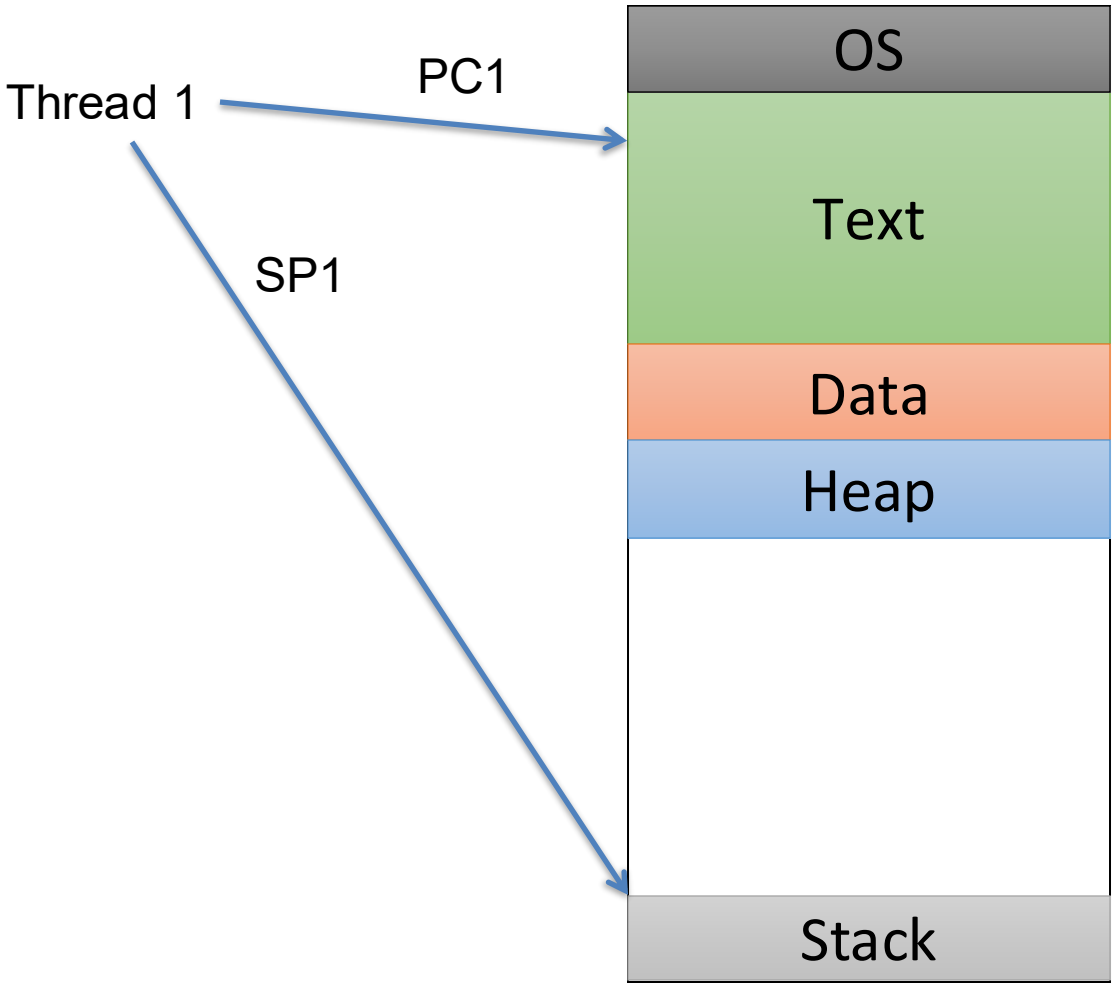
- Modern OSes separate the concepts of processes and threads.
  - The process defines the address space and general process attributes (e.g., open files)
  - The thread defines a sequential execution stream within a process (PC, SP, registers)

- 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

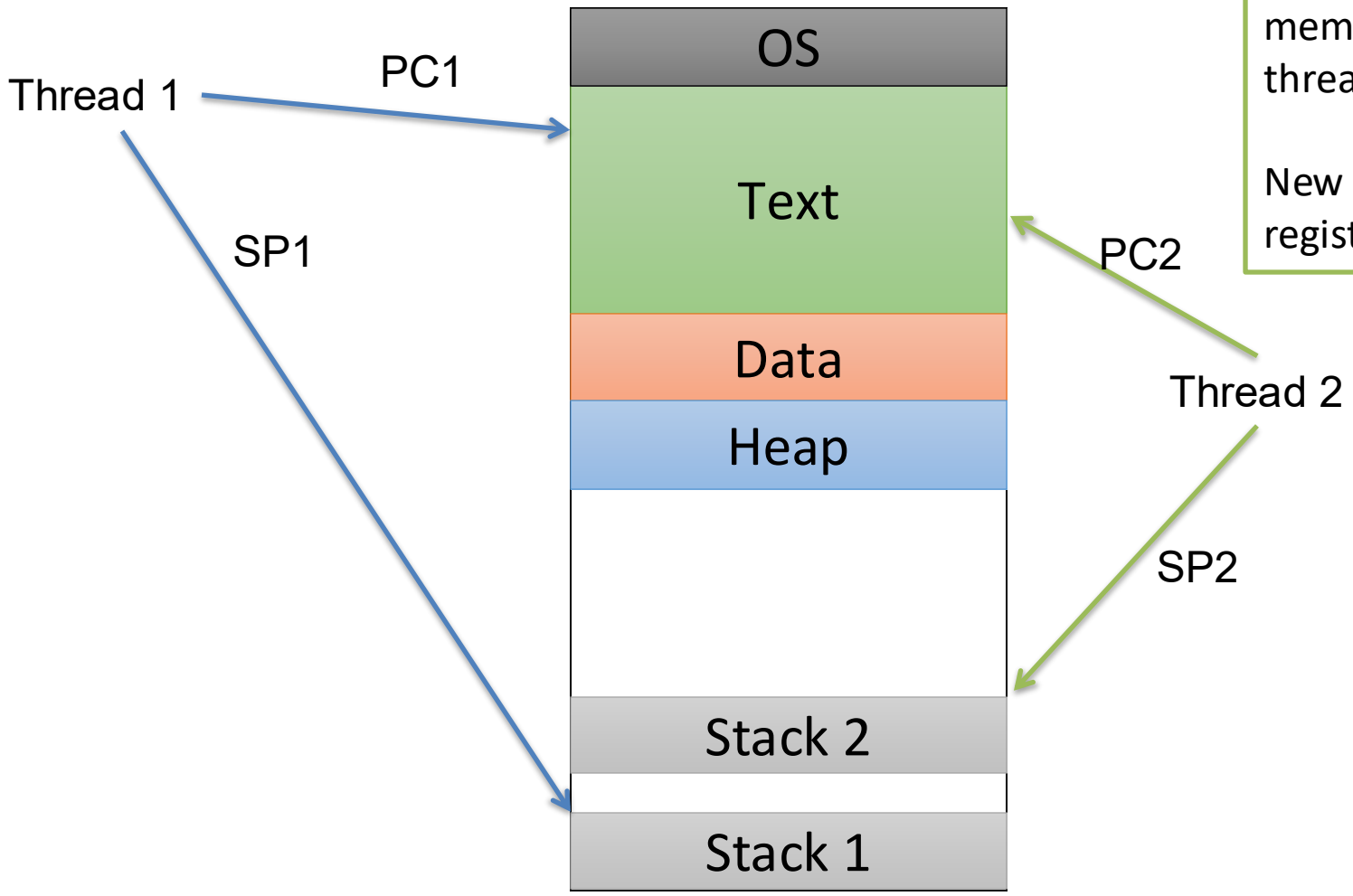
# Threads




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

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

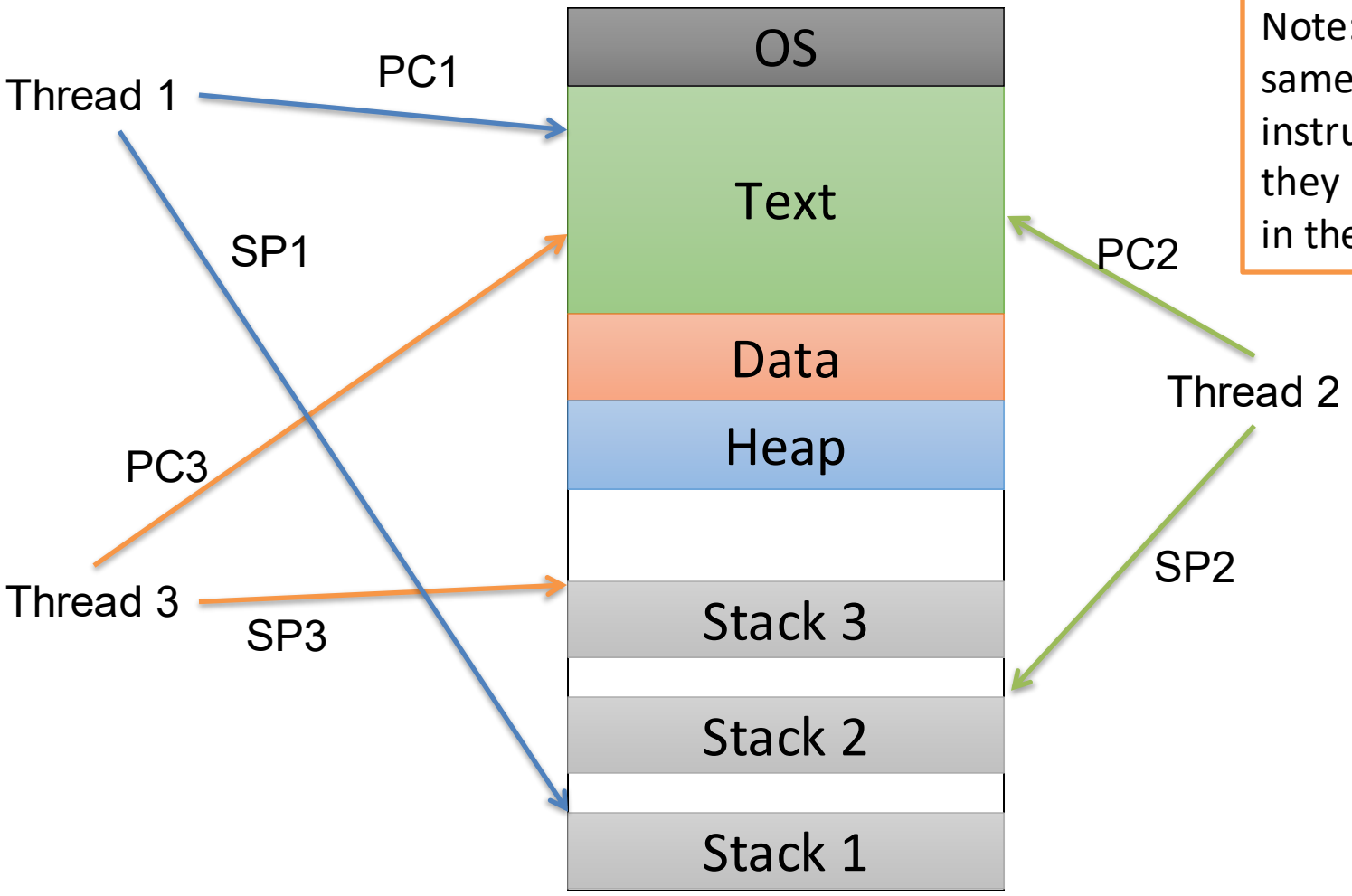
# Threads




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.

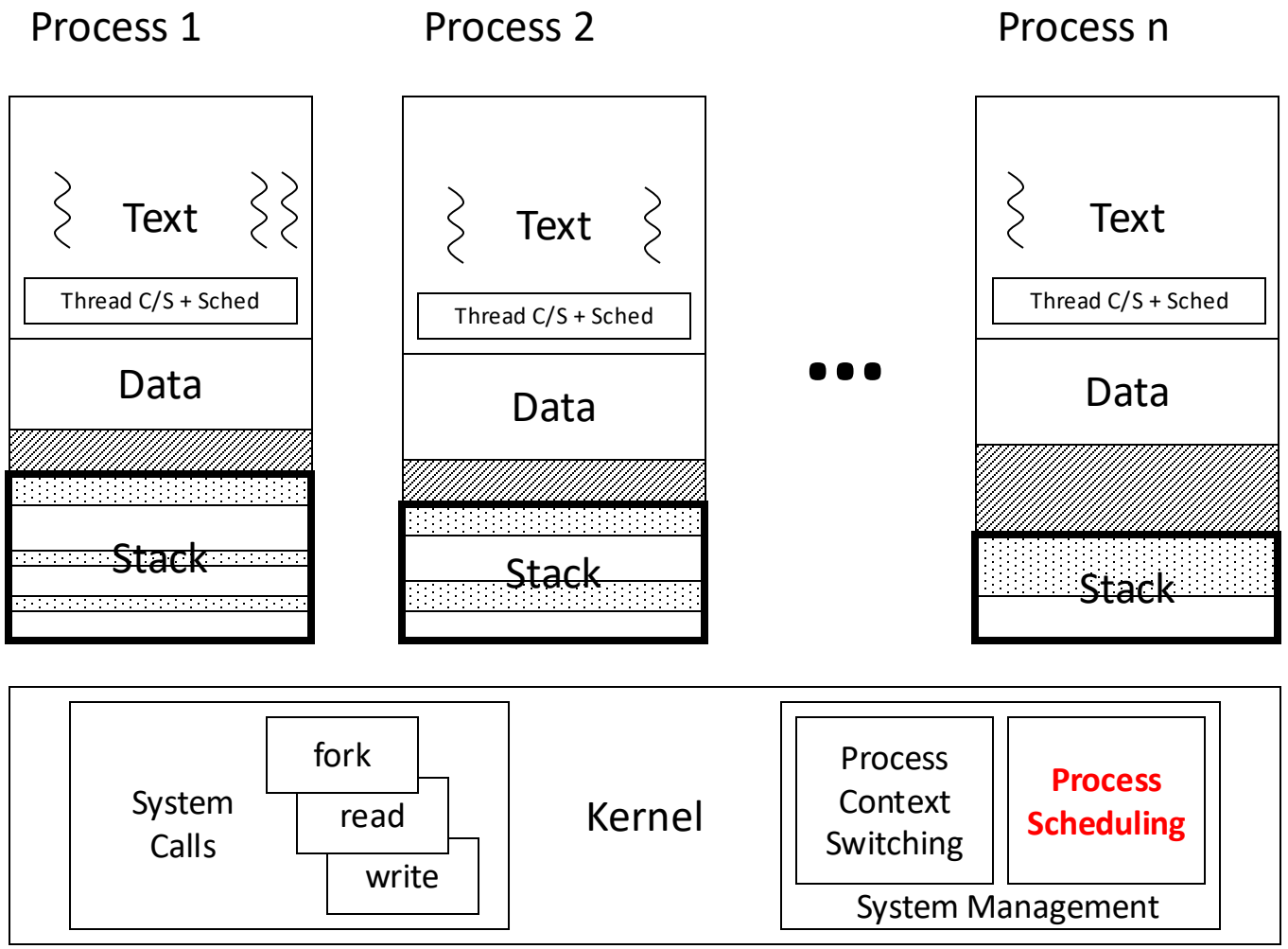
- We can multitask!  Why does that help?
  - Taking advantage of multiple CPUs / cores
  - Overlapping I/O with computation
  - Improving program structure

# 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?

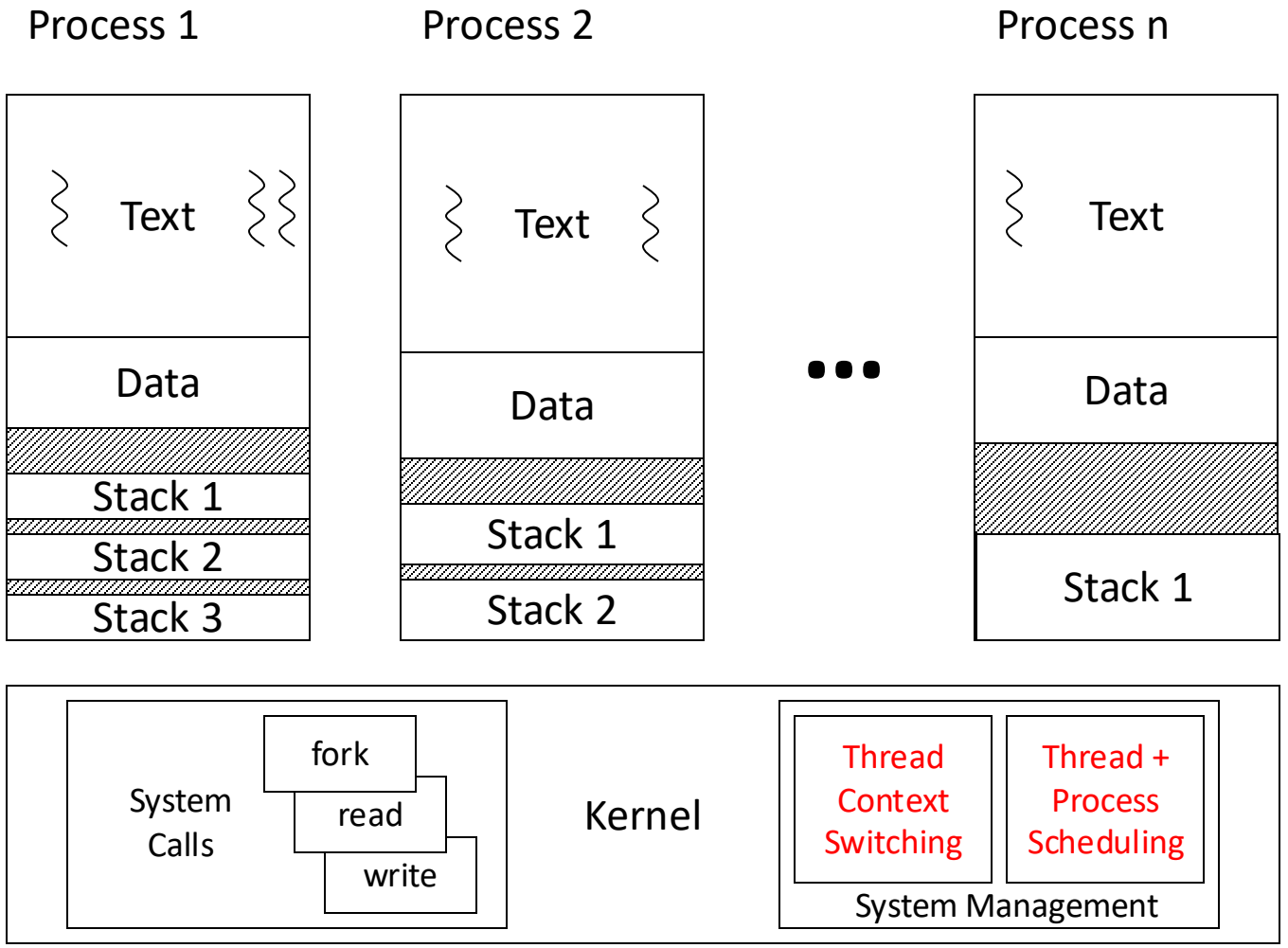
# User-Level Threads




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

# Kernel vs. User Threads

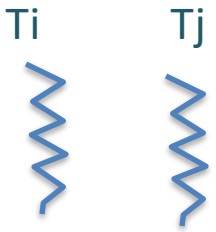
- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slower to create, manipulate, synchronize
    - Requires getting the OS involved, which means changing context (relatively expensive)
- User-level threads
  - Faster to create, manipulate, synchronize
  - Not integrated with OS (uninformed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

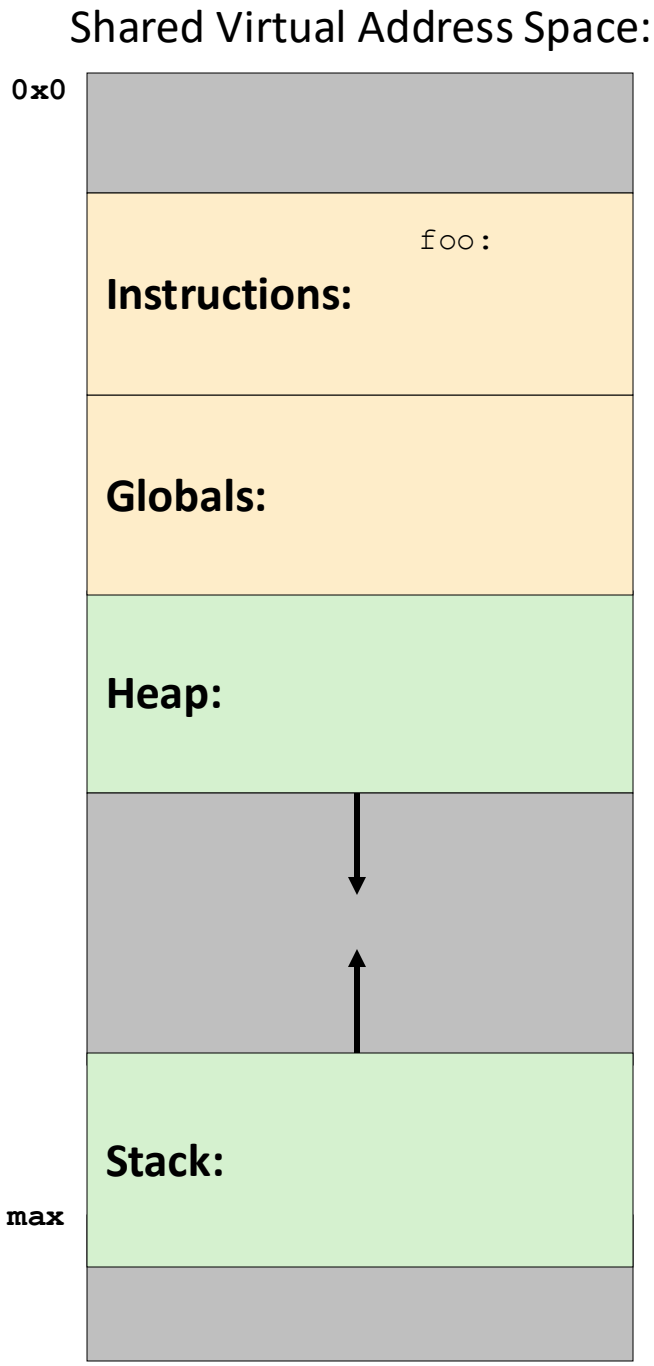
# Threads & Sharing

- 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!!

# Example

```
static int x;

int foo(int *p) {
    int y;

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




If threads Ti and Tj both execute function *foo* code:

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

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

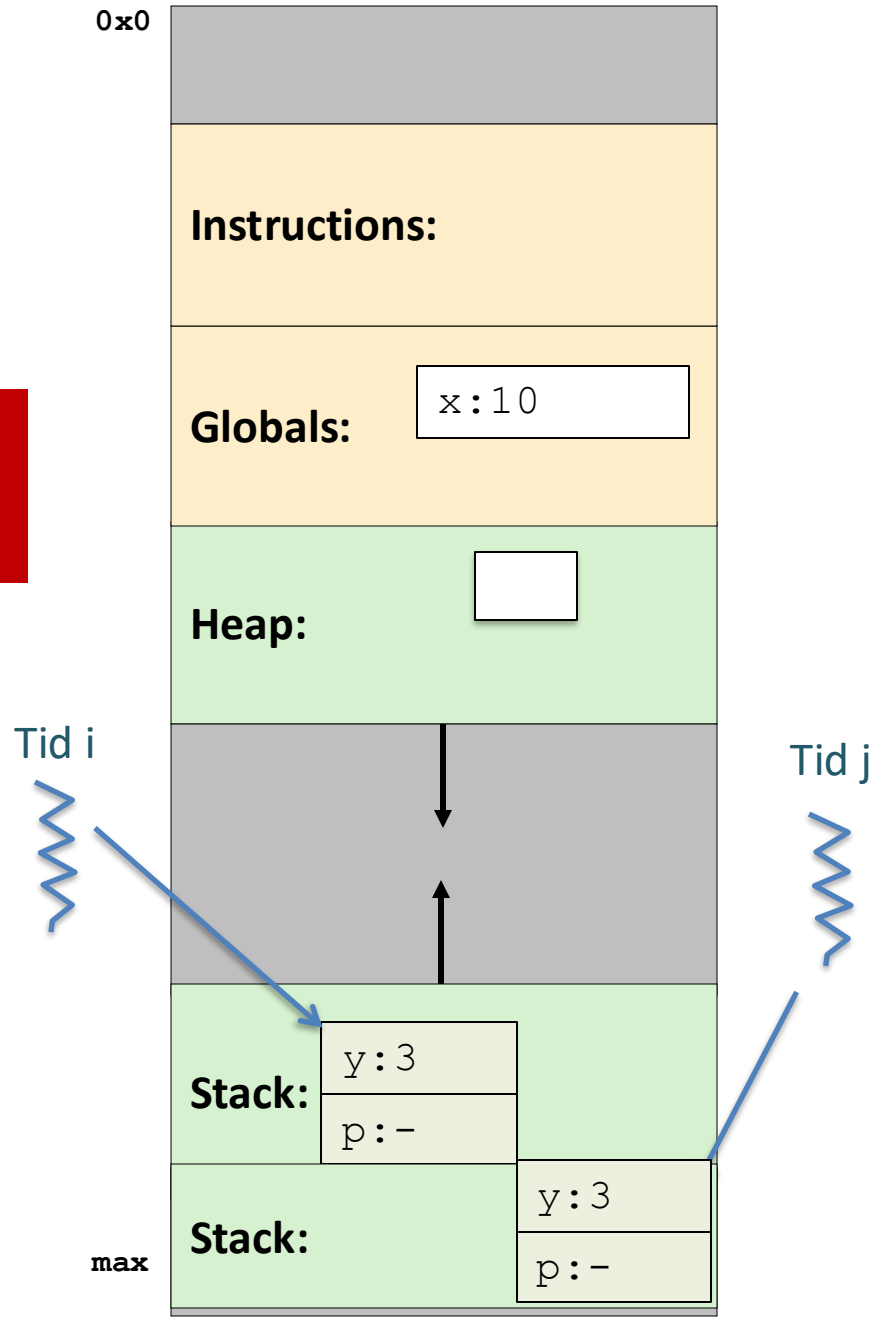
# Example

```
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

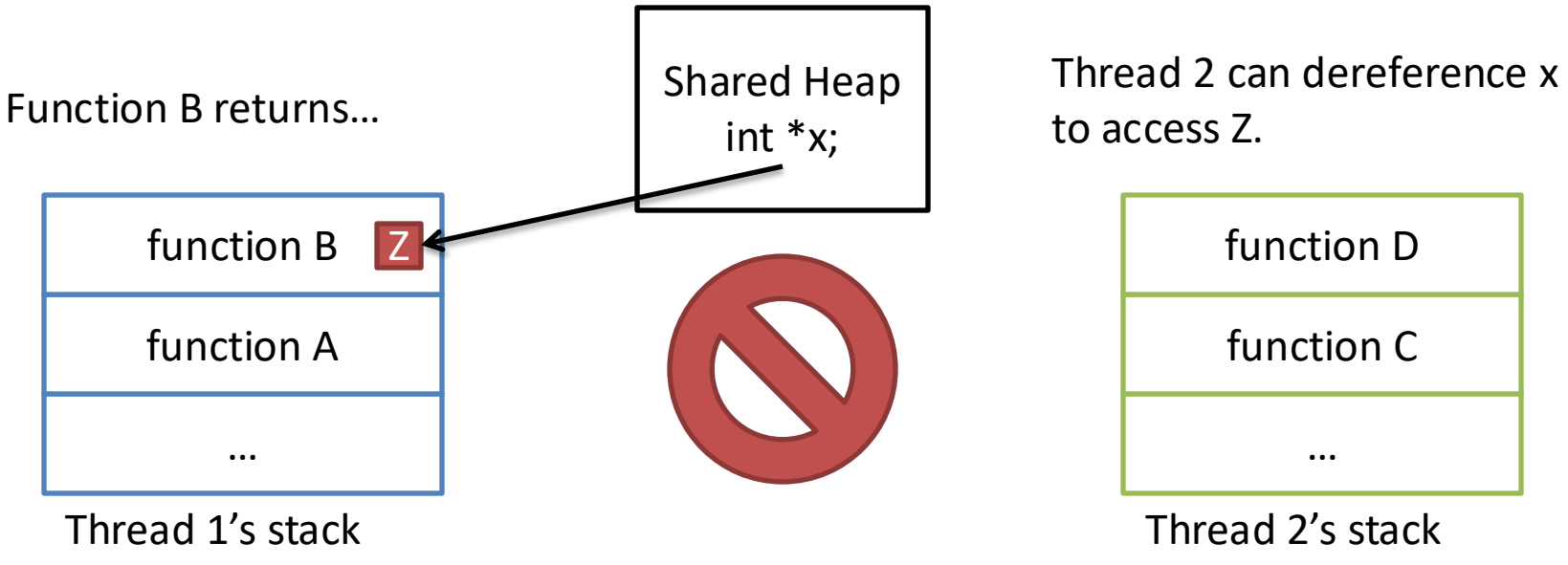
x is in global memory and is shared by every thread

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



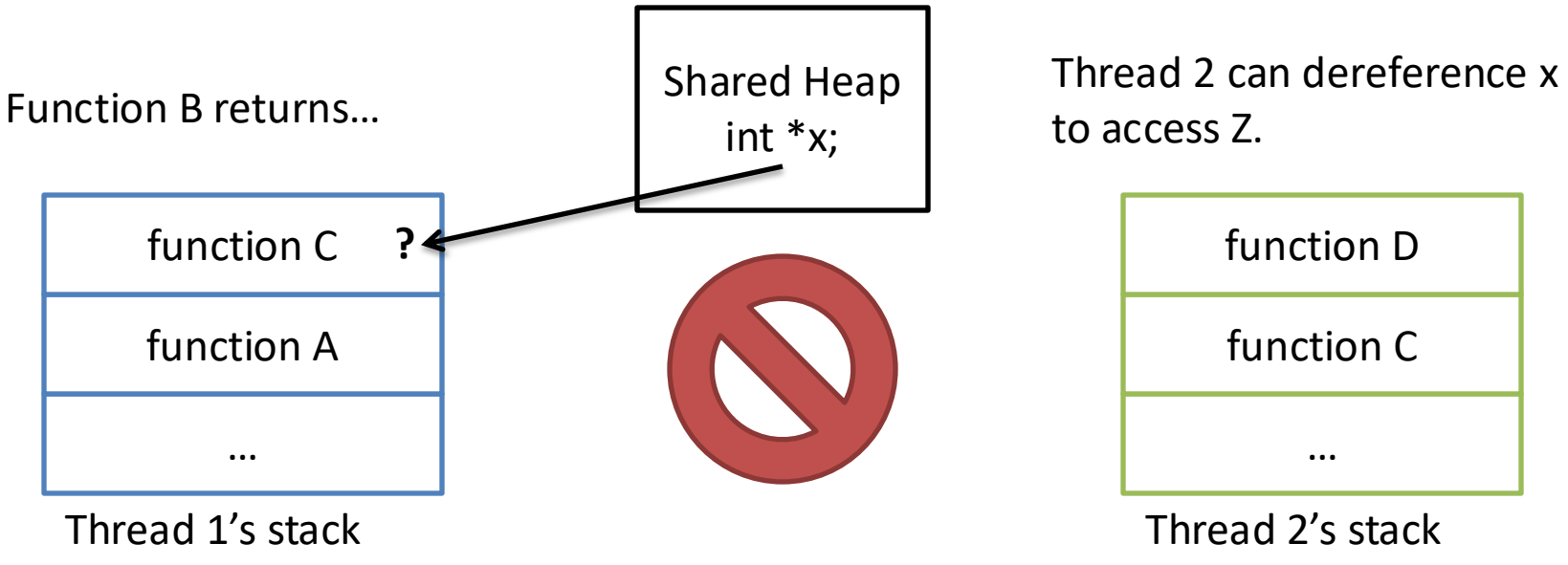
# Threads & Sharing

- 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



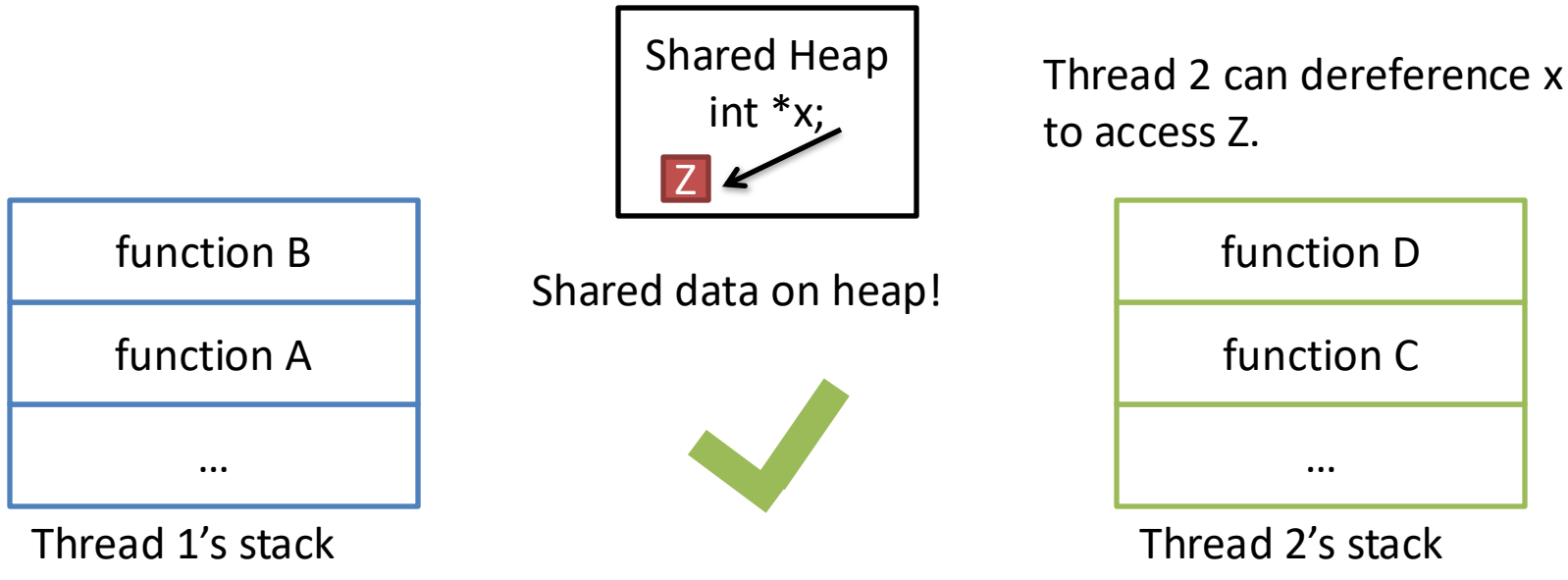
# Threads & Sharing

- 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



# 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



# Pthreads Programming

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 OSes

  (e.g.) 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`

# 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 thread, void **retval);
```

Terminating a thread:

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

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

# void *

```
int pthread_create(…,  void *args);
```

`void *:` a pointer to any type (a generic pointer) can be used for return value and parameter types only

- all memory addresses take up the same number of bytes

```
char *cptr; int *ptr;   // store 8 byte addresses
```

- can pass the address of any type as a void *

```
pthread_create( …, &x);  // addr of an int
pthread_create(…, &ch);  //addr of a char
```

cannot de-reference a void * pointer directly

```
*args = 6; // store 6 in 1 byte? 2 bytes? 4 bytes?
```

- re-cast first before dereference!

```
*((int *) args) = 6; // store 6 in an int (4 bytes)
```

# Example: hello.c with 2 threads

```
#include “pthreads.h”

void *hello(void *arg); //thread’s "main" function

int main() {
  pthread_t tid[2];//two thread ids

  pthread_create(&tid[0], NULL, hello, NULL);
  pthread_create(&tid[1], NULL, hello, NULL);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  exit(0);
}
void *hello(void *arg) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

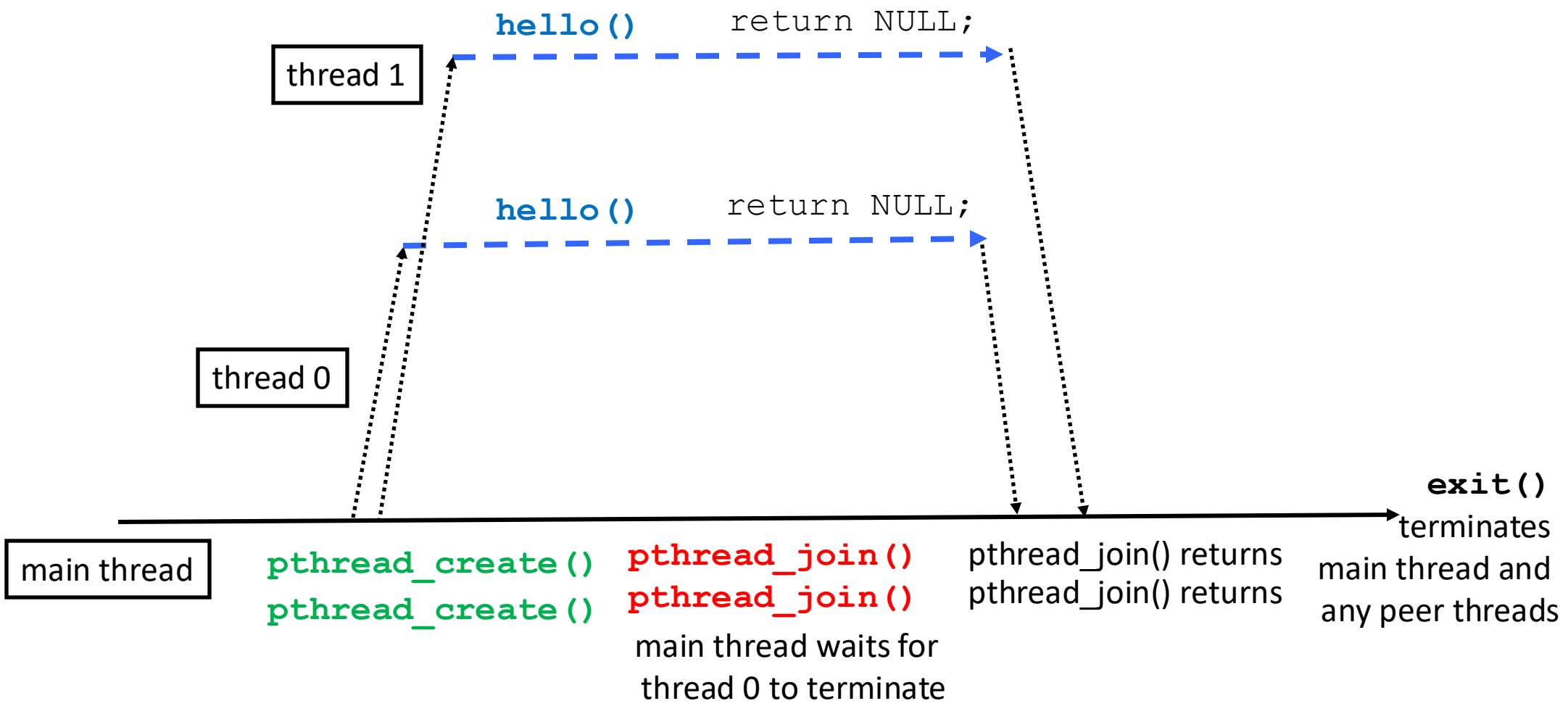
Thread arguments (void *p)

return value (void **p) "pass by pointer parameter"

pass NULL (or 0) if you don't need return value (or if thread’s main returns NULL)

Note: not showing error return value handling

# Concurrent Execution



# Example: hello.c with N threads

- Code on slides is subset of full code
  - Minus error detection and handling for space
  - Minus other non-thread important code

- You can copy and try out the full code from here:

```
$ cd ~/cs31/WeeklyLabs $ mkdir week12
$ cd week12
$ pwd /home/you/cs31/WeeklyLabs/week12
$ cp ~chaganti/public/cs31/s25/week12/* ./
```

```
[week12]$ ./hello 3
Hello! I am thread 0
Hello! I am thread 2
Hello! I am thread 1
Goodbye! I was thread 2
Goodbye! I was thread 0
Goodbye! I was thread 1
count = 1283598
Main thread done
[week12]$ ./hello 3
Hello! I am thread 0
Hello! I am thread 1
Hello! I am thread 2
Goodbye! I was thread 2
Goodbye! I was thread 1
Goodbye! I was thread 0
count = 1123889
```

# hello.c: main function

```
static unsigned long long count = 0;  // global variable

int main(int argc, char *argv) {
   pthread_t *tids;           // thread ids
   int ntids, i, *tid_args;  //arguments passed to thread funcs.
```

initial var. declaration

# hello.c: main function

```
static unsigned long long count = 0;  // global variable

int main(int argc, char *argv) {
   pthread_t *tids;           // thread ids
   int ntids, i, *tid_args;  //arguments passed to thread funcs.

   ntids = atoi(argv[1]);
   tids = (pthread_t *) malloc( sizeof(pthread_t) * ntids);
   tid_args = (int *) malloc( sizeof(int) * ntids);
```

initial var. declaration

allocate memory for pointer vars.

# hello.c: main function

```
static unsigned long long count = 0;  // global variable

int main(int argc, char *argv) {
   pthread_t *tids;           // thread ids
   int ntids, i, *tid_args;  //arguments passed to thread funcs.

   ntids = atoi(argv[1]);
   tids = (pthread_t *) malloc( sizeof(pthread_t) * ntids);
   tid_args = (int *) malloc( sizeof(int) * ntids);

   for (i=0; i < ntids; i++) { //loop through num. threads
         tid_args[i] = i;      //input argument for each thread.
         pthread_create(&tids[i], 0, thread_hello, &tid_args[i]);
      }   //create the thread, with tid, func ptr, and input args.

   for (i=0; i < ntids; i++) {
          pthread_join(tids[i], 0);
      }
```

initial var. declaration

allocate memory for pointer vars.

create threads

join equivalent of waiting to reap threads

# Hello.c: main function:

pthread_create:

```
ret = pthread_create(&tids[i], NULL,
        thread_hello, &tid_args[i])
```

- function pointer argument (`thread_hello`)
  - name of the function that the spawned thread will start executing
  - generic function pointer type:

    ```
    void *thread_main_func(void *arg);
    ```
- args argument (`&tid_args[i]`)

  `void *`: pass a pointer to any type: int, float, struct, array, …

pthread_join: like wait in fork-wait

- tid argument: which *pthreads* thread to wait for to exit

# hello.c: thread main function

```
static unsigned long long count =0;  // global variable

void *thread_hello(void *arg) {


  int myid, i;
  myid = *((int *)arg);


  printf("hello I'm thread %d with pthread_id %lu\n",
      myid, pthread_self());

  for(i = 0; i < 1000000; i++) {

    count += i;
  }

  printf("goodbye I'm thread %d\n",myid);
  return (void *)0;  // recast 0 to return type (void *)
}
```

thread get own copy of local vars and params on its stack

getting arg's value:
arg's type is void *
but we know its type for a specific implementation, re-cast to correct type

returns the ID of the thread in which it is invoked

all threads can access global variable (shared access)

# Some runs with 4 threads:

result with 4 threads should be `1999998000000`

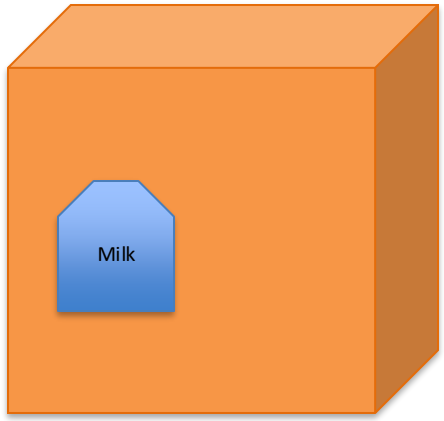
```
./hello 4
count = 793900079488
./hello 4
count = 539879105421
./hello 4
count = 509829883618
./hello 4
count = 581580128846
```

# Synchronization

- Synchronize: to (arrange events to) happen such that two events do not overwrite each other's work.
- Thread synchronization
  - When one thread has to wait for another
  - Events in threads that occur "at the same time"
- Uses of synchronization
  - Prevent race conditions
  - Wait for resources to become available (only one thread has access at any time - deadlocks)

# Synchronization: Too Much Milk (TMM)

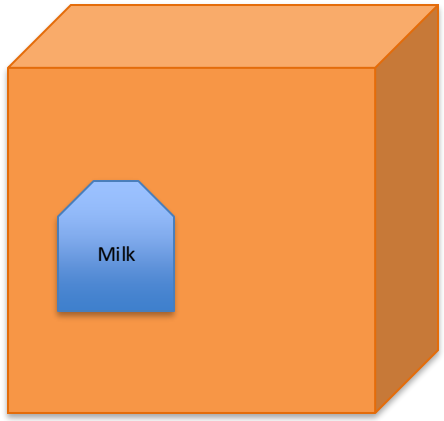
| Time | You | Your Roommate |
|---|---|---|
| 3.00 | Arrive home | |
| 3.05 | Look in fridge, no milk | |
| 3.10 | Leave for the grocery store | |
| 3.15 | | |
| 3.20 | Arrive at the grocery store | |
| 3.25 | Buy Milk | |
| 3.30 | | |
| 3.35 | Arrive home, put milk in fridge | Arrive Home |
| 3.40 | | Look in fridge, find milk |
| 3.45 | | Cold Coffee (nom) |




What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

# How many cartons of milk can we have in this scenario? (Can we ensure this somehow?)

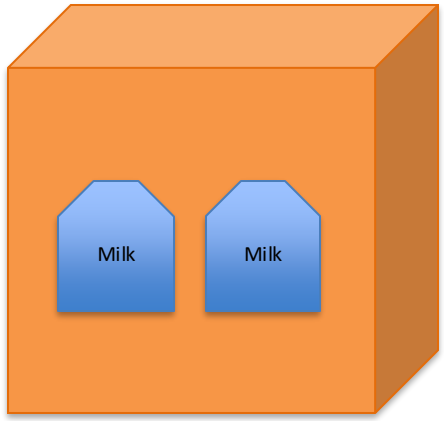
| Time | You | Your Roommate |
|---|---|---|
| 3.00 | Arrive home | |
| 3.05 | Look in fridge, no milk | |
| 3.10 | Leave for the grocery store | |
| 3.15 | | |
| 3.20 | Arrive at the grocery store | |
| 3.25 | Buy Milk | |
| 3.30 | | |
| 3.35 | Arrive home, put milk in fridge | Arrive Home |
| 3.40 | | Look in fridge, find milk |
| 3.45 | | Cold Coffee (nom) |




A. One carton (you)
B. Two cartons
C. No cartons
D. Something else

# Synchronization:
# Too Much Milk (TMM): One possible scenario

| Time | You | Your Roommate |
|---|---|---|
| 3.00 | Arrive home | |
| 3.05 | Look in fridge, no milk | |
| 3.10 | Leave for grocery | Arrive Home |
| 3.15 | | Look in fridge, no milk |
| 3.20 | Arrive at grocery | Leave for grocery |
| 3.25 | Buy Milk | |
| 3.30 | | Arrive at grocery |
| 3.35 | Arrive home, put milk in fridge | |
| 3.40 | | Arrive home, put milk in fridge |
| 3.45 | | Oh No! |



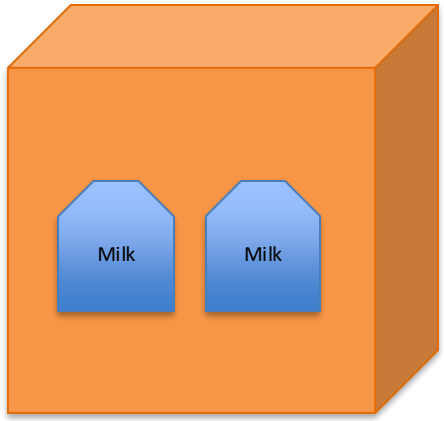

What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

# Synchronization:

Threads get scheduled in an arbitrary manner:

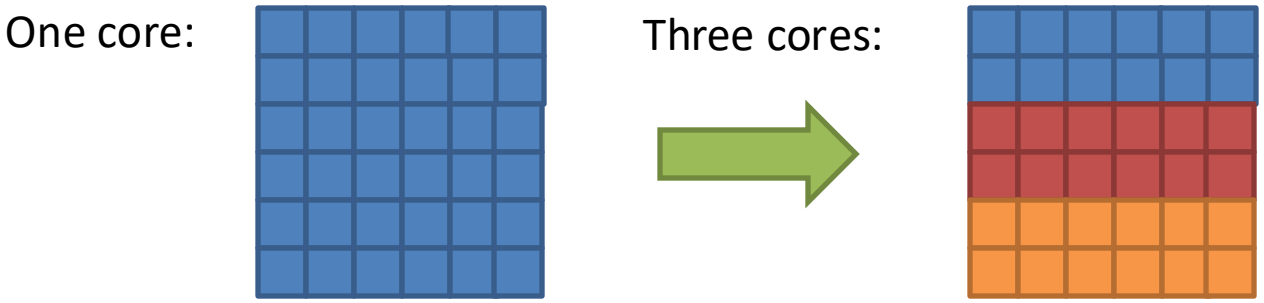
bad things may happen: ...or nothing may happen

| Time | You | Your Roommate |
|---|---|---|
| 3.00 | Arrive home | |
| 3.05 | Look in fridge, no milk | |
| 3.10 | Leave for grocery | Arrive Home |
| 3.15 | | Look in fridge, no milk |
| 3.20 | Arrive at grocery | Leave for grocery |
| 3.25 | Buy Milk | |
| 3.30 | | Arrive at grocery |
| 3.35 | Arrive home, put milk in fridge | |
| 3.40 | | Arrive home, put milk in fridge |
| 3.45 | | Oh No! |




What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

# Synchronization Example




- Coordination required:
  - Which thread goes first?
  - Threads in different regions must work together to compute new value for boundary cells.
  - Threads might not run at the same speed (depends on the OS scheduler). Can't let one region get too far ahead.
  - Context switches can happen at any time!

# Thread Ordering

(Why threads require care. Humans aren't good at reasoning about this.)

- As a programmer you have *no idea* when threads will run. The OS schedules them, and the schedule will vary across runs.
- It might decide to context switch from one thread to another *at any time*.
- Your code must be prepared for this!
  - Ask yourself: "Would something bad happen if we context switched here?"
- hard to debug this problem if it is not reproducible

# Example: The Credit/Debit Problem

- Say you have $1000 in your bank account
  - You deposit $100
  - You also withdraw $100
- How much should be in your account?
- What if your deposit and withdrawal occur at the same time, at different ATMs?

# Credit/Debit Problem: Race Condition

**Thread $T_0$**

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

**Thread $T_1$**

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

# Credit/Debit Problem: Race Condition

| Say $T_0$ runs first |
| --- |
| **Read $1000 into b** |

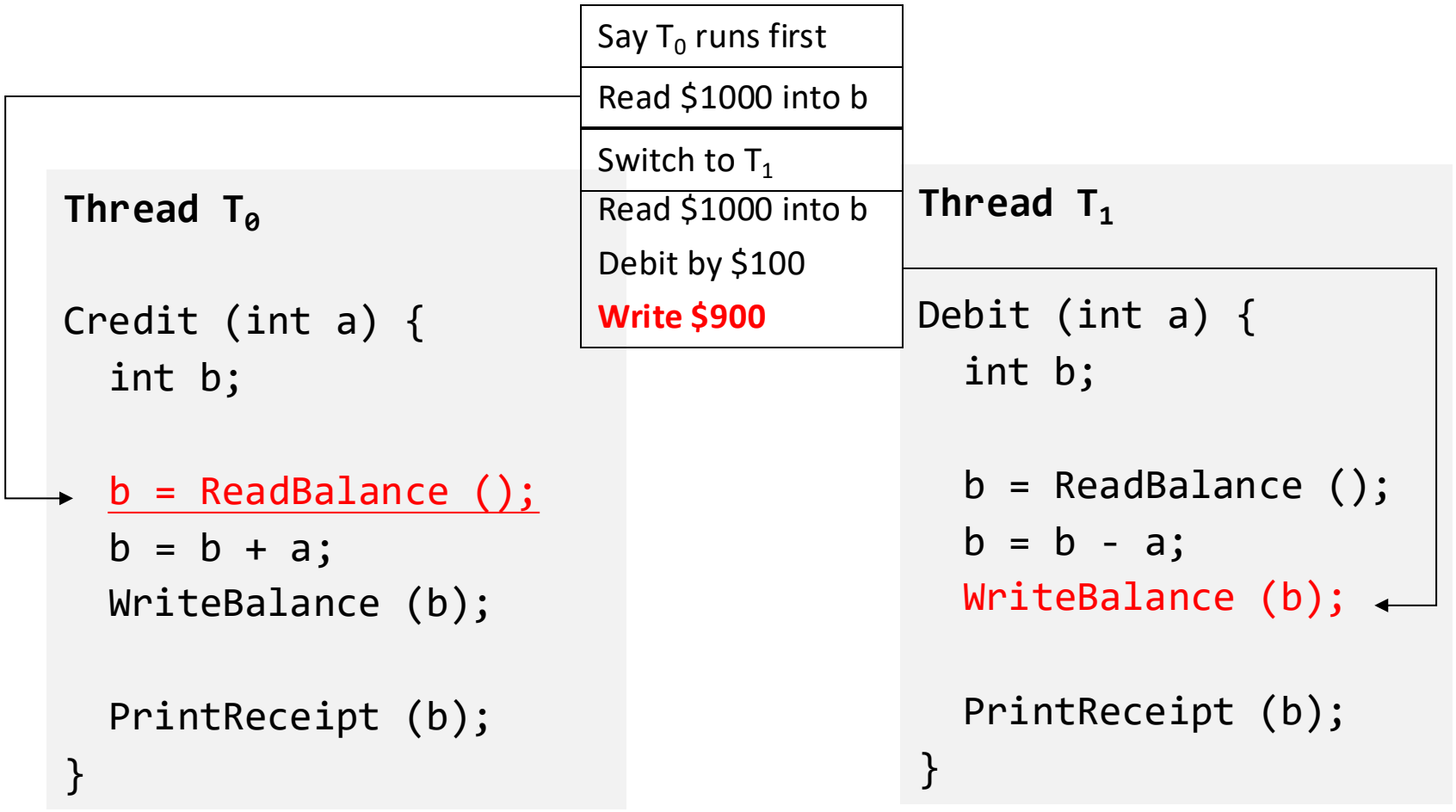
**Thread $T_0$**

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

**Thread $T_1$**

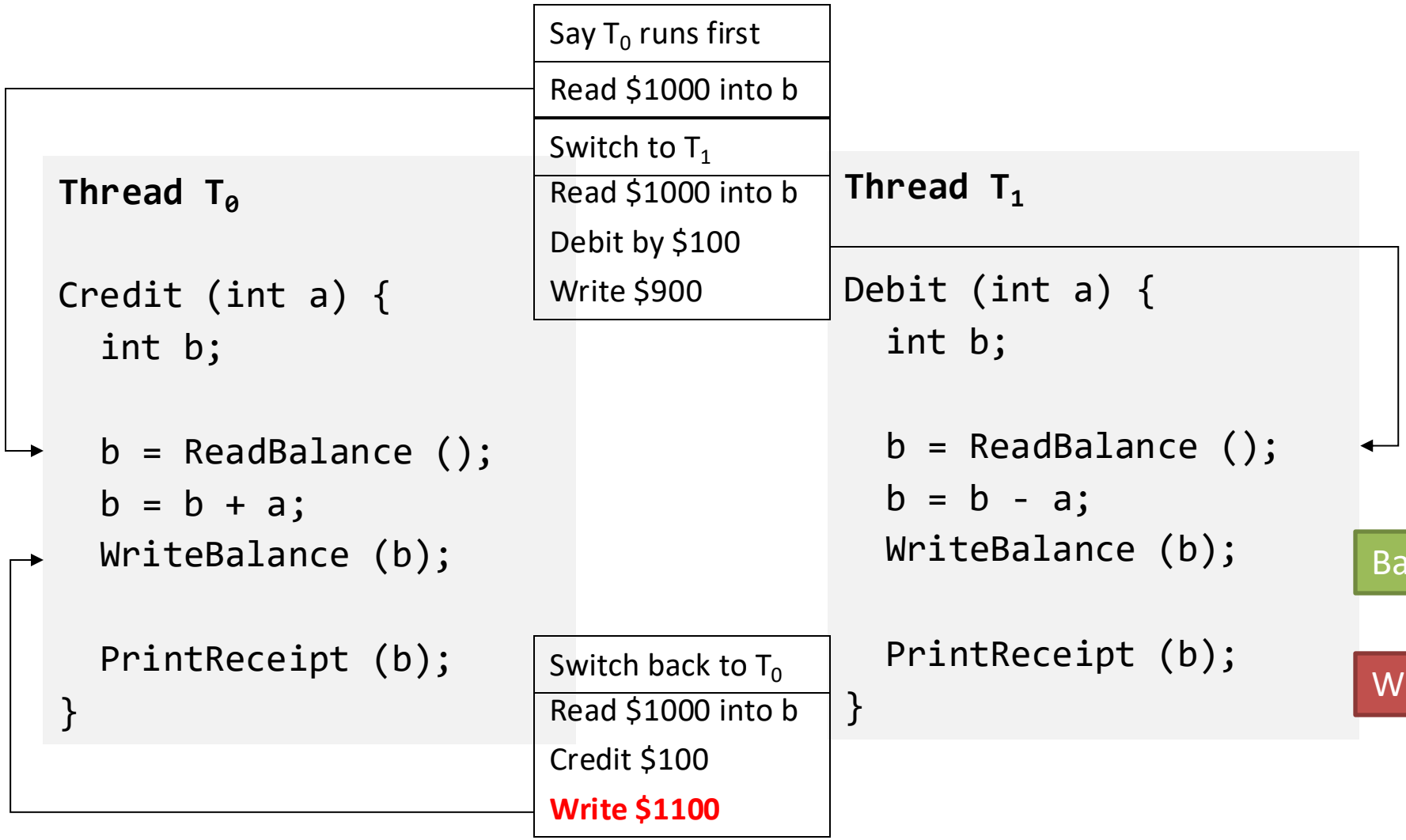
```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

# Credit/Debit Problem: Race Condition




**CONTEXT SWITCH**

# Credit/Debit Problem: Race Condition



# "Critical Section"

**Thread $T_0$**

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Danger Will Robinson!

Badness if context switch here!

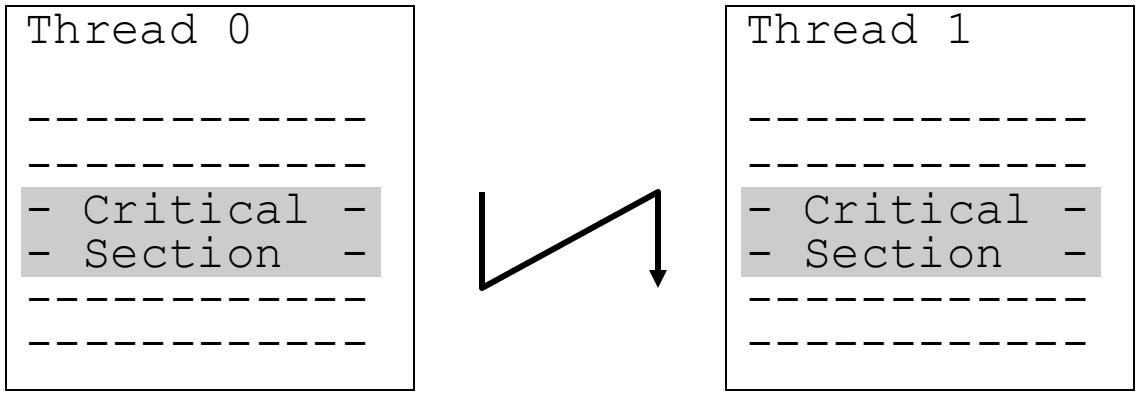
**Thread $T_1$**

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Bank gave you $100!

What went wrong?

# To Avoid Race Conditions




1. Identify critical sections

2. Use synchronization to enforce mutual exclusion
   - Only one thread active in a critical section

# Critical Section and Atomicity

- Sections of code executed by multiple threads
  - Access shared variables, often making local copy
  - Places where order of execution or thread interleaving will affect the outcome
  - Follows: read + modify + write of shared variable

- Must run atomically with respect to each other
  - Atomicity: runs as an entire instruction or not at all. Cannot be divided into smaller parts.

# Which code region is a critical section?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

A

B

C

D

E

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

# Which code region is a critical section?

read + modify + write of shared variable

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

D

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

Large enough for correctness + Small enough to minimize slow down

# Which values might the shared `s` variable hold after both threads finish?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

# If A runs first

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

(s = 40)
s = 50

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

# B runs after A Completes

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

```
(s = 50)
 s = 30;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

# What about interleaving?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

One of the threads will overwrite the other's changes.

# Is there a race condition?

Suppose `count` is a global variable, multiple threads increment it:

```
count++;
```

A. Yes, there's a race condition (`count++` is a critical section).
B. No, there's no race condition (`count++` is not a critical section).
C. Cannot be determined

How about if compiler implements it as:

```
movq (%rdx), %rax    // read count value
addq $1, %rax        // modify value
movq %rax, (%rdx)    // write count
```

How about if compiler implements it as:

```
incq (%rdx)          // increment value
```

# Atomicity

- The implementation of acquiring/releasing critical section must be atomic.
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes “all or nothing”
- How do we make them atomic?
  - Atomic instructions (e.g., test-and-set, compare-and-swap)
  - Allows us to build “semaphore” OS abstraction

# Four Rules for Mutual Exclusion

1. No two threads can be inside their critical sections at the same time (one of many but not more than one).
2. No thread outside its critical section may prevent others from entering their critical sections.
3. No thread should have to wait forever to enter its critical section. (Starvation)
4. No assumptions can be made about speeds or number of CPU's.

Railroad Semaphore

- Help trains figure out which track to be on at any given time.

Railroad Semaphore

- Help trains figure out which track to be on at any given time.

O.S. Semaphore:

- Construct that the OS provides to processes.
- Make system calls to modify their value

# Mutual Exclusion with Semaphores

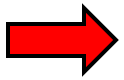
```
mutex = 1; //lock and unlock mutex atomically.
```

| $T_0$ | $T_1$ |
| --- | --- |
| `lock (mutex);` | `lock (mutex);` |
| `< critical section >` | `< critical section >` |
| `unlock (mutex);` | `unlock (mutex);` |

Atomicity: run the entire instruction without interruption.

# Mutual Exclusion with Semaphores

```
mutex = 1; //unlocked.
```



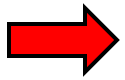
| $T_0$ | $T_1$ |
|---|---|
| lock (mutex); | lock (mutex); |
| < critical section > | < critical section > |
| unlock (mutex); | unlock (mutex); |

Atomicity: run the entire instruction without interruption.

$T_0$: Wants to execute the critical section
$T_0$: Reads the value of mutex,
Changes the value of mutex = 0 (acquires lock)
Enters critical section.

# Mutual Exclusion with Semaphores

```
mutex = 0; //locked.
```



| $T_0$ | $T_1$ |
|---|---|
| `lock (mutex);` | `lock (mutex);` |
| `< critical section >` | `< critical section >` |
| `unlock (mutex);` | `unlock (mutex);` |

Atomicity: run the entire instruction without interruption.

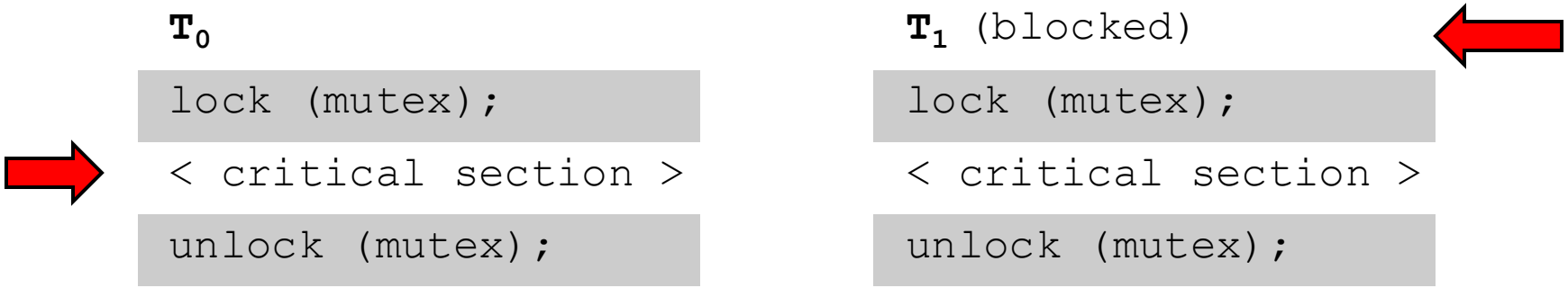
$T_0$: Wants to execute the critical section

$T_0$: Reads the value of mutex,
Changes the value of mutex = 0 (acquires lock)
Enters critical section.

Atomic Execution

# Mutual Exclusion with Semaphores

```
mutex = 0; //locked.
```

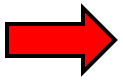



Atomicity: run the entire instruction without interruption.

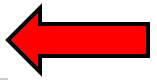
$T_0$: In the critical section
$T_1$: Wants to enter the critical section.
Reads the value of mutex (mutex = 0)
Cannot enter critical section.
Blocked.

# Mutual Exclusion with Semaphores

```
mutex = 0; //locked.
```



| $T_0$ | $T_1$ (blocked) |
| --- | --- |
| lock (mutex); | lock (mutex); |
| < critical section > | < critical section > |
| unlock (mutex); | unlock (mutex); |

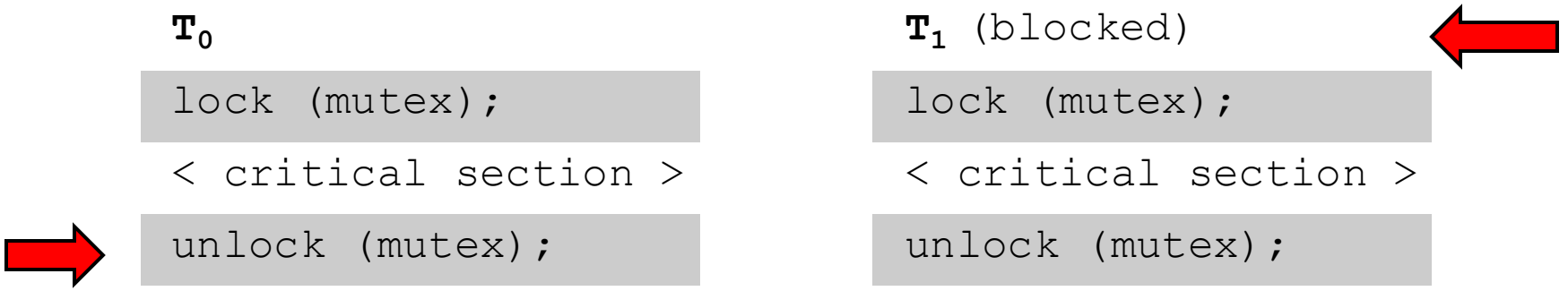


Atomicity: run the entire instruction without interruption.

$T_0$: Completes execution of critical section
Updates mutex value = 1. (release lock)

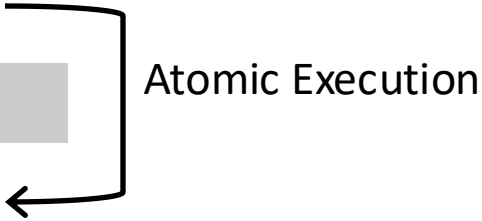
# Mutual Exclusion with Semaphores

```
mutex = 1; //unlocked.
```




Atomicity: run the entire instruction without interruption.

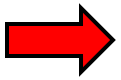
$T_0$: Completes execution of critical section
Updates mutex value = 1. (release lock)

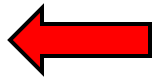


# Mutual Exclusion with Semaphores

```
mutex = 1; //locked.
```

| $T_0$ | $T_1$ |
| --- | --- |
| lock (mutex); | lock (mutex); |
| < critical section > | < critical section > |
| unlock (mutex); | unlock (mutex); |





Atomicity: run the entire instruction without interruption.

$T_1$: Can now acquire lock atomically and
Enter the critical section

# Mutual Exclusion with Semaphores

```
mutex = 1; //lock and unlock mutex atomically.
```

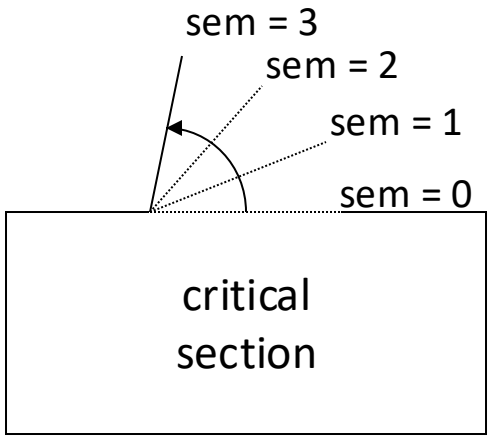
| $T_0$ | $T_1$ |
|---|---|
| `lock (mutex);` | `lock (mutex);` |
| `< critical section >` | `< critical section >` |
| `unlock (mutex);` | `unlock (mutex);` |

- Use a “mutex” semaphore initialized to 1
- Only one thread can enter critical section at a time.
- Simple, works for any number of threads

Atomicity: runs as an entire instruction or not at all.

# Semaphores

- Semaphore: OS synchronization variable
  - Has integer value
  - List of waiting threads
- Works like a gate
- If sem > 0, gate is open
  - Value equals number of threads that can enter
- Else, gate is closed
  - Possibly with waiting threads



# Semaphores

- Associated with each semaphore is a queue of waiting threads
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

# Semaphore Operations

```
sem s = n;                                          // declare and initialize

wait (sem s)                                        // Executes atomically(*)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s)                                      // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

(*) With help from special hardware instructions.

# Semaphore Operations

```
sem s = n;                                          // declare and initialize

wait (sem s)                                        // Executes atomically(*)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s)                                      // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

Based on what you know about semaphores, should a process be able to check beforehand whether wait(s) will cause it to block?

A. Yes, it should be able to check.

B. No, it should not be able to check.

# Semaphore Operations

```
sem s = n;                                          // declare and initialize

wait (sem s)                                        // Executes atomically(*)
    decrement s;
    if s < 0:
        block thread (and associate with s);

signal (sem s)                                      // Executes atomically(*)
    increment s;
    if blocked threads:
        unblock (any) one of them;
```

- No other operations allowed
- In particular, semaphore's value can't be tested!
  - No thread can tell the value of s

# Synchronization: More than Mutexes

- "I want to block a thread until something specific happens."
  - Condition variable: wait for a condition to be true
- "I want all my threads to sync up at the same point."
  - Barrier: wait for everyone to catch up.

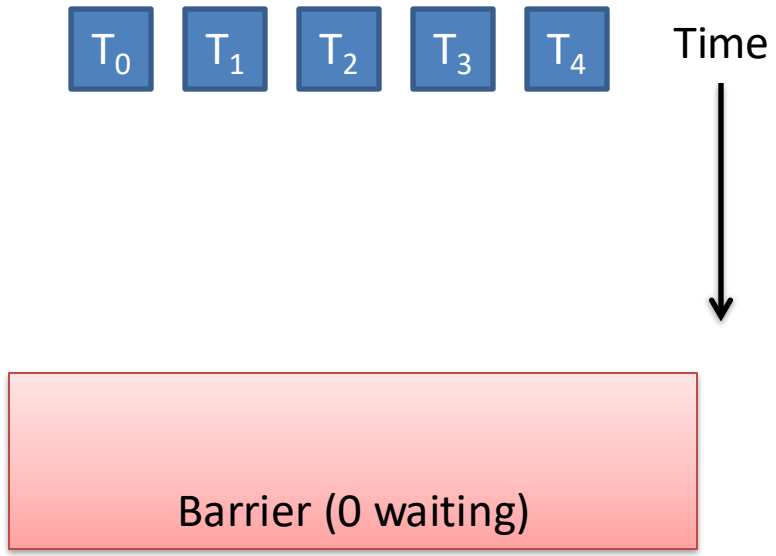
# Barriers

- Used to coordinate threads, but also other forms of concurrent execution.

- Often found in simulations that have discrete rounds.  (e.g., game of life)

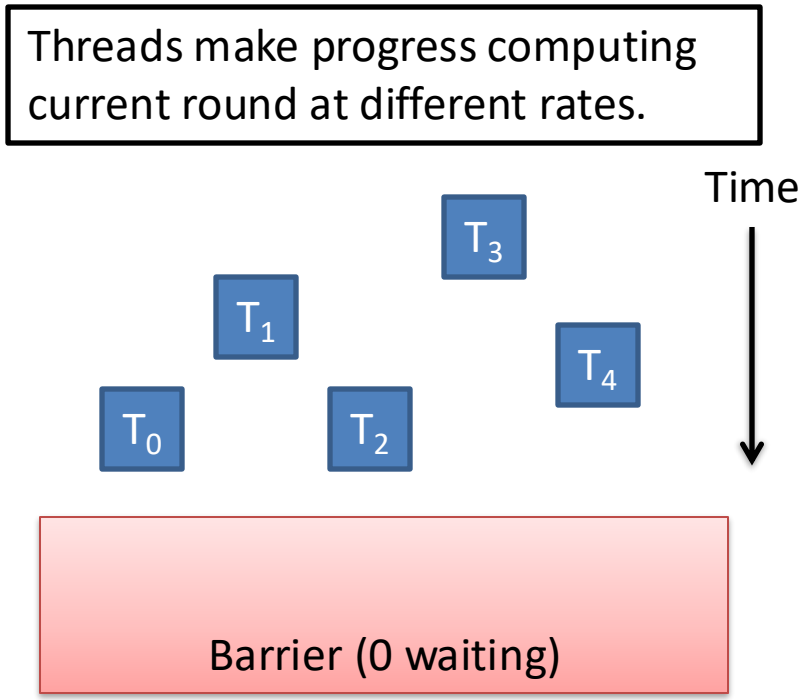
# Barrier Example, N Threads

```
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
  while (…) {
    compute_sim_round()
    barrier_wait(&b)
  }
}
```



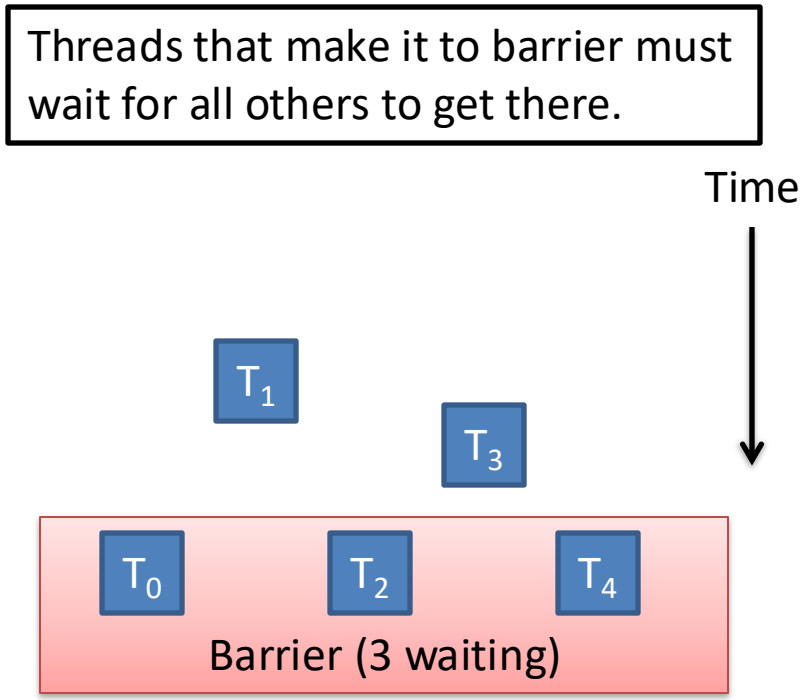
# Barrier Example, N Threads

```
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
  while (…) {
    compute_sim_round()
    barrier_wait(&b)
  }
}
```



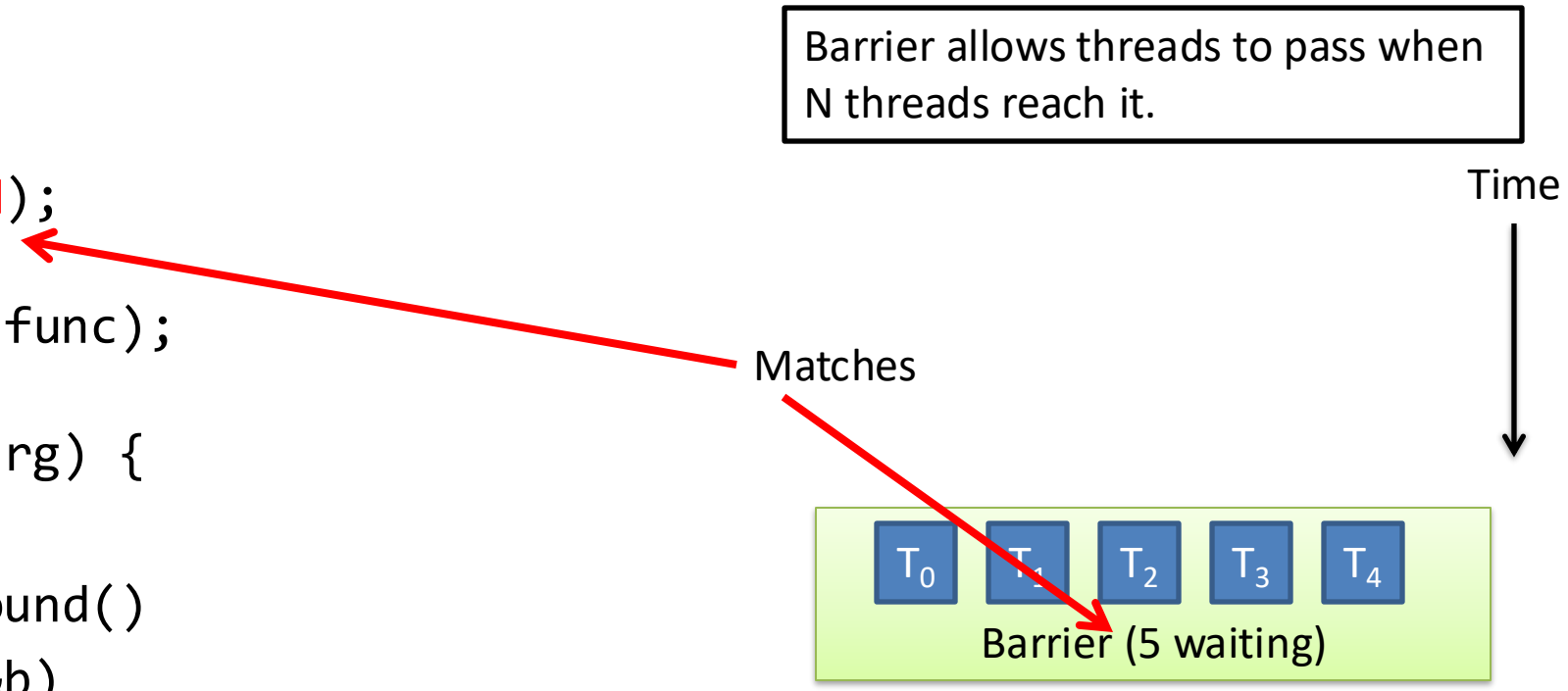
# Barrier Example, N Threads

```
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
  while (…) {
    compute_sim_round()
    barrier_wait(&b)
  }
}
```



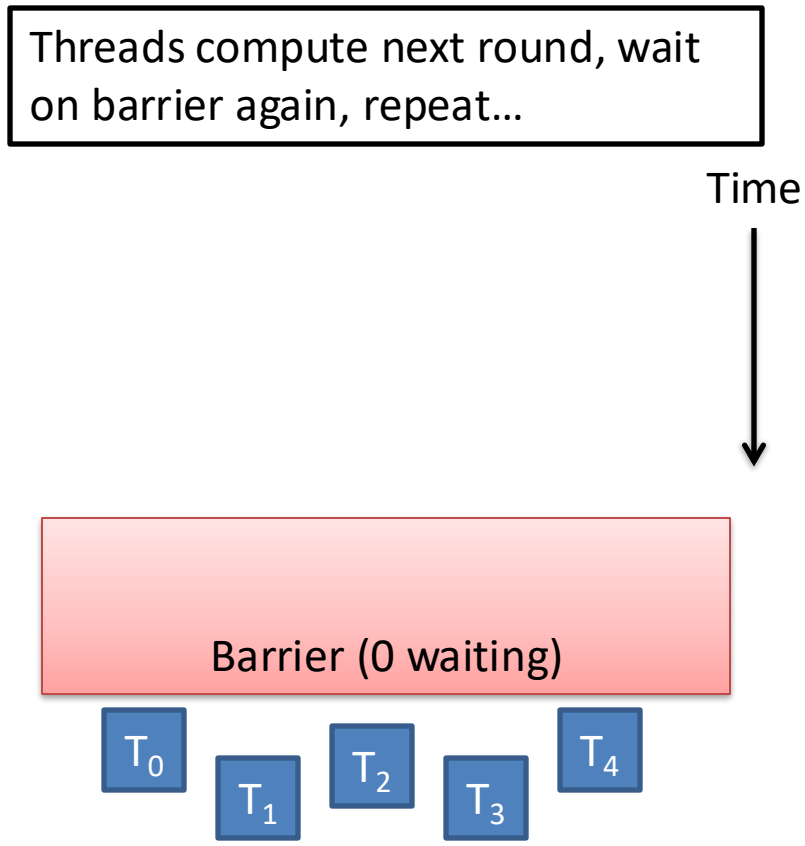
# Barrier Example, N Threads

```
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
  while (…) {
    compute_sim_round()
    barrier_wait(&b)
  }
}
```



# Barrier Example, N Threads

```
shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
  while (…) {
    compute_sim_round()
    barrier_wait(&b)
  }
}
```



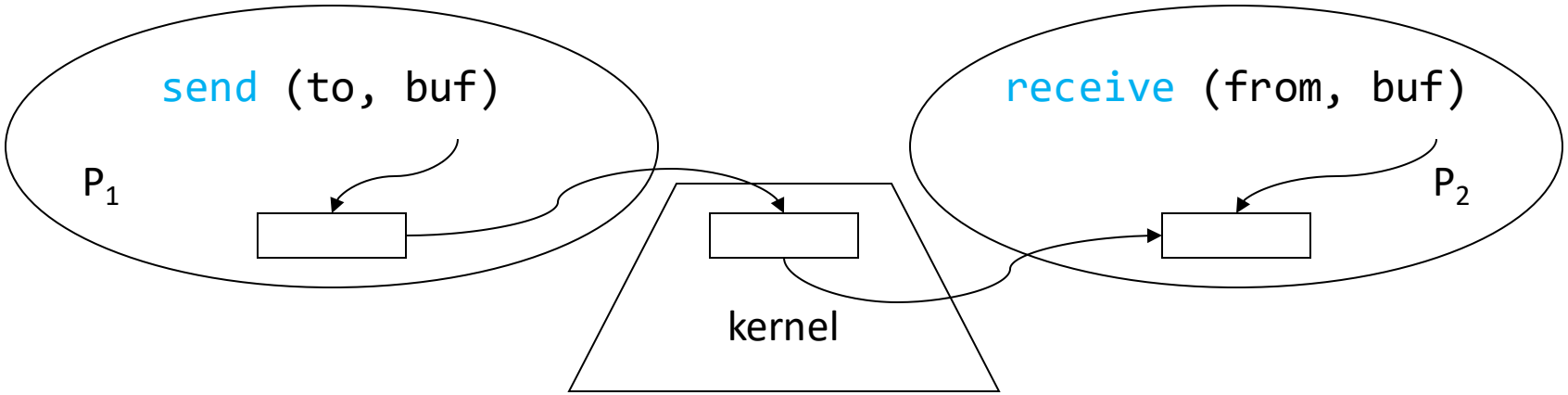
# Synchronization: More than Mutexes

- I want all my threads to sync up at the same point.
  - Barrier: wait for everyone to catch up.

- I want to block a thread until something specific happens.
  - Condition variable: wait for a condition to be true

- I want my threads to share a critical section when they're reading, *but still safely write*.
  - Readers/writers lock: distinguish how lock is used

# Synchronization: Beyond Mutexes
# Message Passing




- Operating system mechanism for IPC
  - `send (destination, message_buffer)`
  - `receive (source, message_buffer)`
- Data transfer: in to and out of kernel message buffers
- Synchronization: can't receive until message is sent

# Summary

- We have no idea when OS will schedule or context switch our threads.
  - Code must be prepared, tough to reason about.
- Threads often must synchronize
  - To safely communicate / transfer data, without races
- Synchronization primitives help programmers
  - Kernel-level semaphores: limit # of threads that can do something, provides atomicity
  - User-level locks: built upon semaphore, provides mutual exclusion (usually part of thread library)