CS 31: Introduction to Computer Systems

22: Race Conditions & Synchronization

April 23, 2020

SWARTHMORE COLLEGE

Recap

- To speed up a job, must divide it across multiple cores.

- Thread: abstraction for execution within process.
  - Threads share process memory.
  - Threads may need to communicate to achieve goal

- Thread communication:
  - To solve task (e.g., neighbor GOL cells)
  - To prevent bad interactions (synchronization)

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)

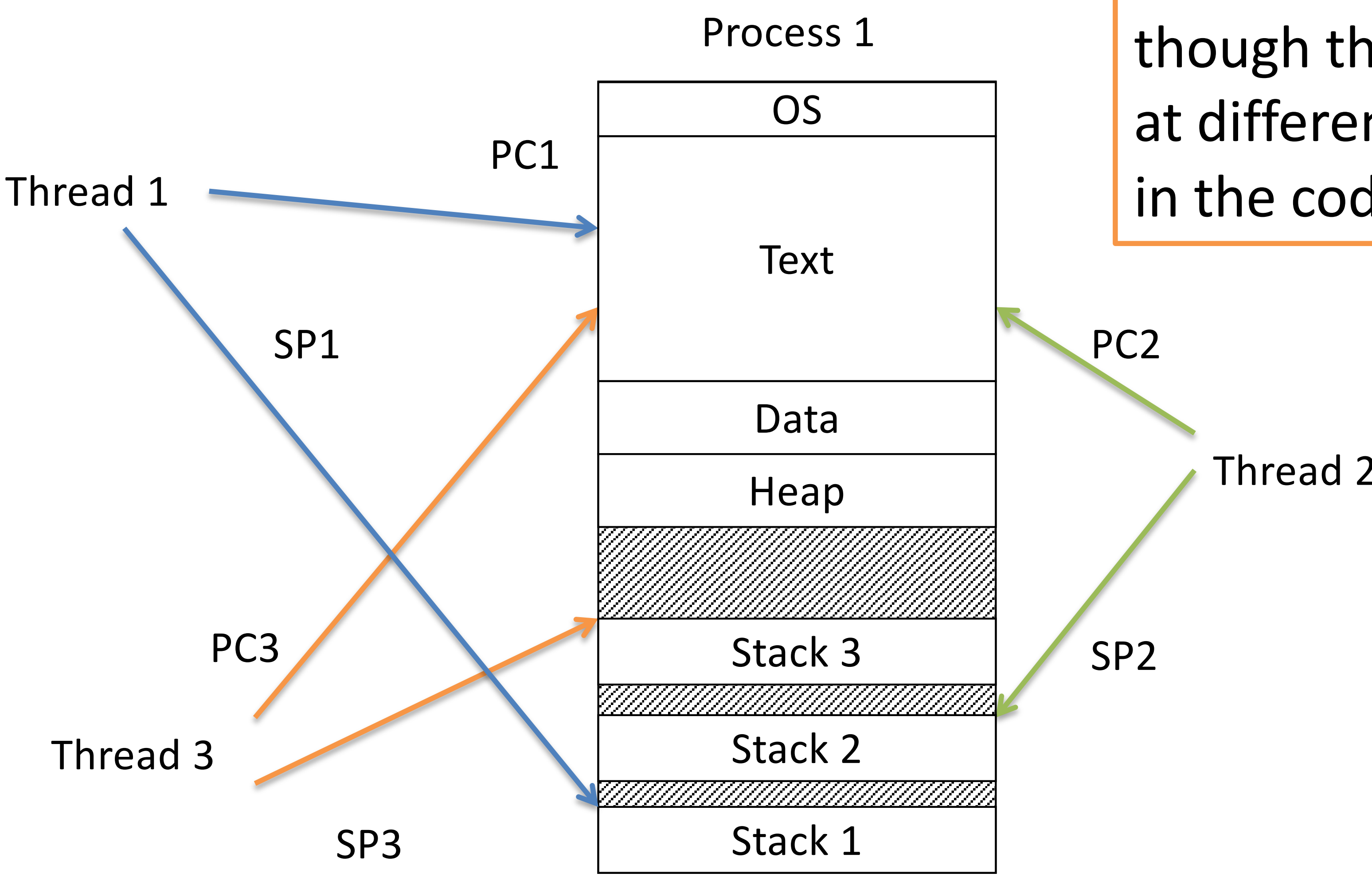
E. More than twice as fast(>2X) (rare: possible if you have more CPU + more memory)

Parallel Speedup

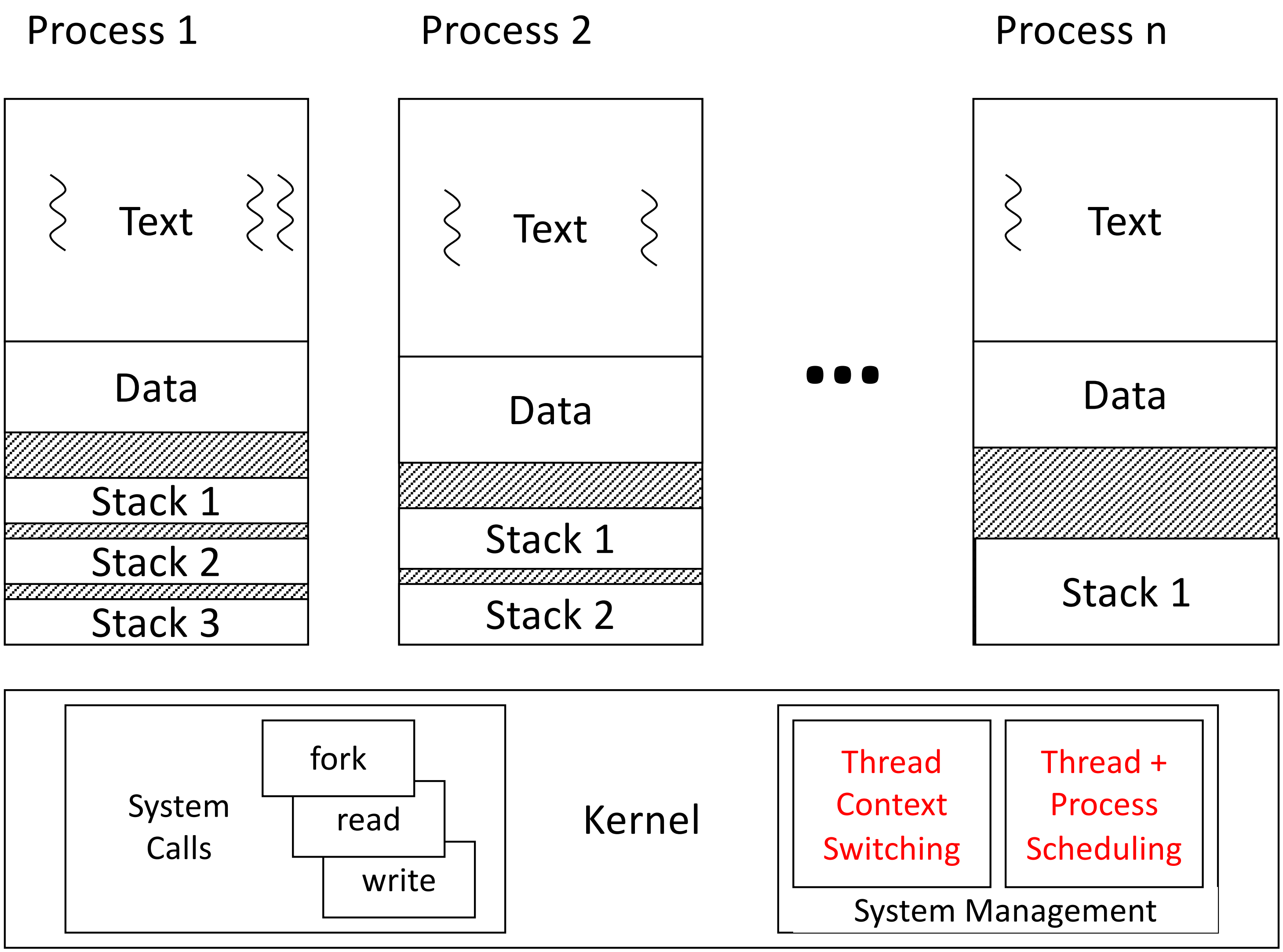
- Performance benefit of parallel threads depends on many factors:
  - algorithm divisibility
  - communication overhead
  - memory hierarchy and locality
  - implementation quality
- *For most programs*, more threads means more communication, diminishing returns.

Threads

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



Kernel-Level Threads



Kernel Context switching over threads

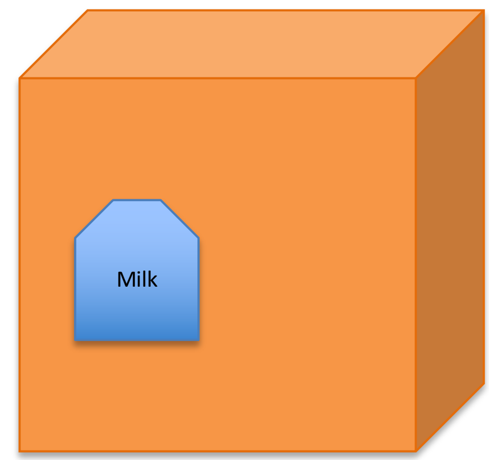
Each process has explicitly mapped regions for stacks

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)

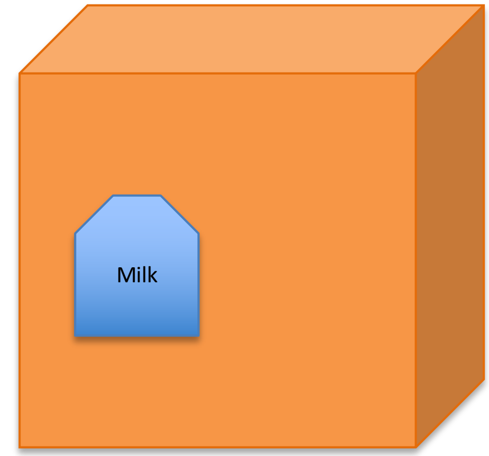
Time	You	Your Roommate
3.00	Arrive home	
3.05	Look in fridge, no milk	
3.10	Leave for the grocery store	
3.15		
3.20	Arrive at the grocery store	
3.25	Buy Milk	
3.30		
3.35	Arrive home, put milk in fridge	Arrive Home
3.40		Look in fridge, find milk
3.45		Cold Coffee (nom)



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?)

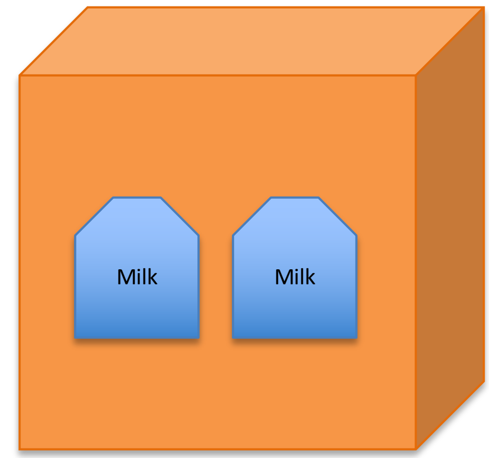
Time	You	Your Roommate
3.00	Arrive home	
3.05	Look in fridge, no milk	
3.10	Leave for the grocery store	
3.15		
3.20	Arrive at the grocery store	
3.25	Buy Milk	
3.30		
3.35	Arrive home, put milk in fridge	Arrive Home
3.40		Look in fridge, find milk
3.45		Cold Coffee (nom)



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!



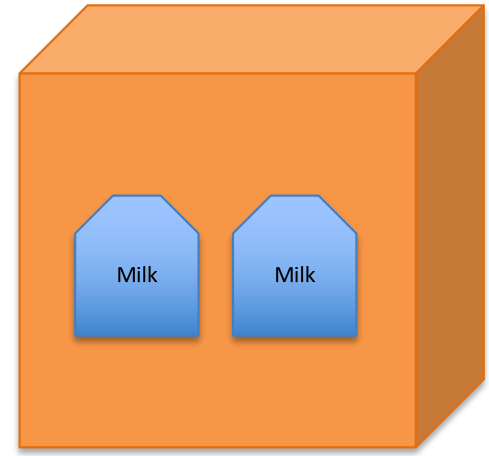
What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

Synchronization:

Threads get scheduled in an arbitrary manner:

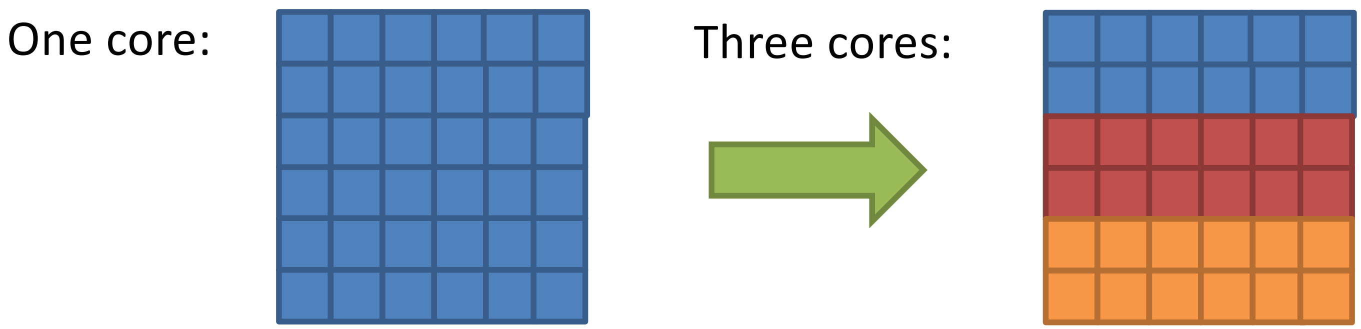
bad things may happen: ...or nothing may happen

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

Say T_0 runs first
Read \$1000 into b
Switch to T_1
Read \$1000 into b Debit by \$100 **Write \$900**

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);
}
```

CONTEXT SWITCH

Credit/Debit Problem: Race Condition

Thread T_0

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

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

  PrintReceipt (b);
}
```

Say T_0 runs first
Read $1000 into b
Switch to T_1
Read $1000 into b Debit by $100 Write $900

Thread T_1

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

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

  PrintReceipt (b);
}
```

Switch back to T_0
Read $1000 into b Credit $100 **Write $1100**

Bank gave you $100!

What went wrong?

"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!

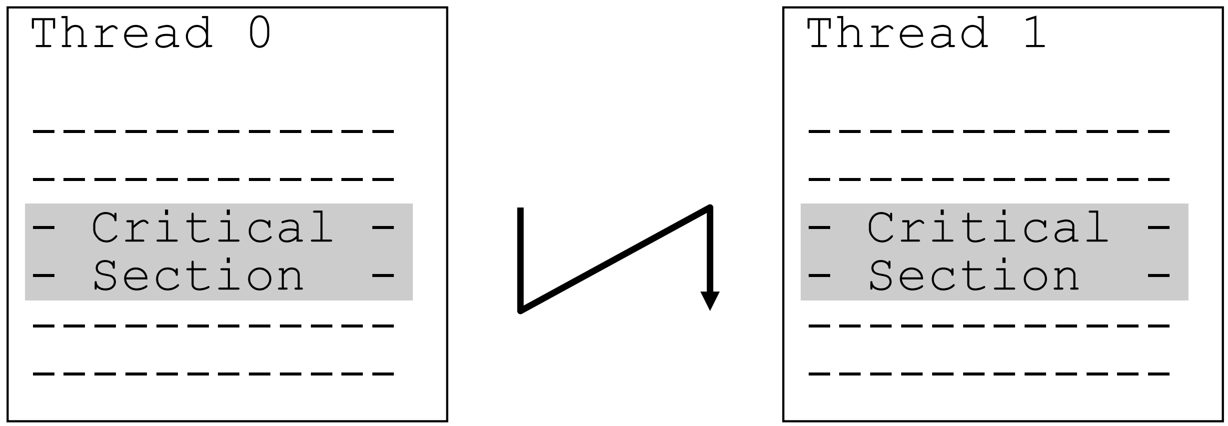
Thread T_1

```
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

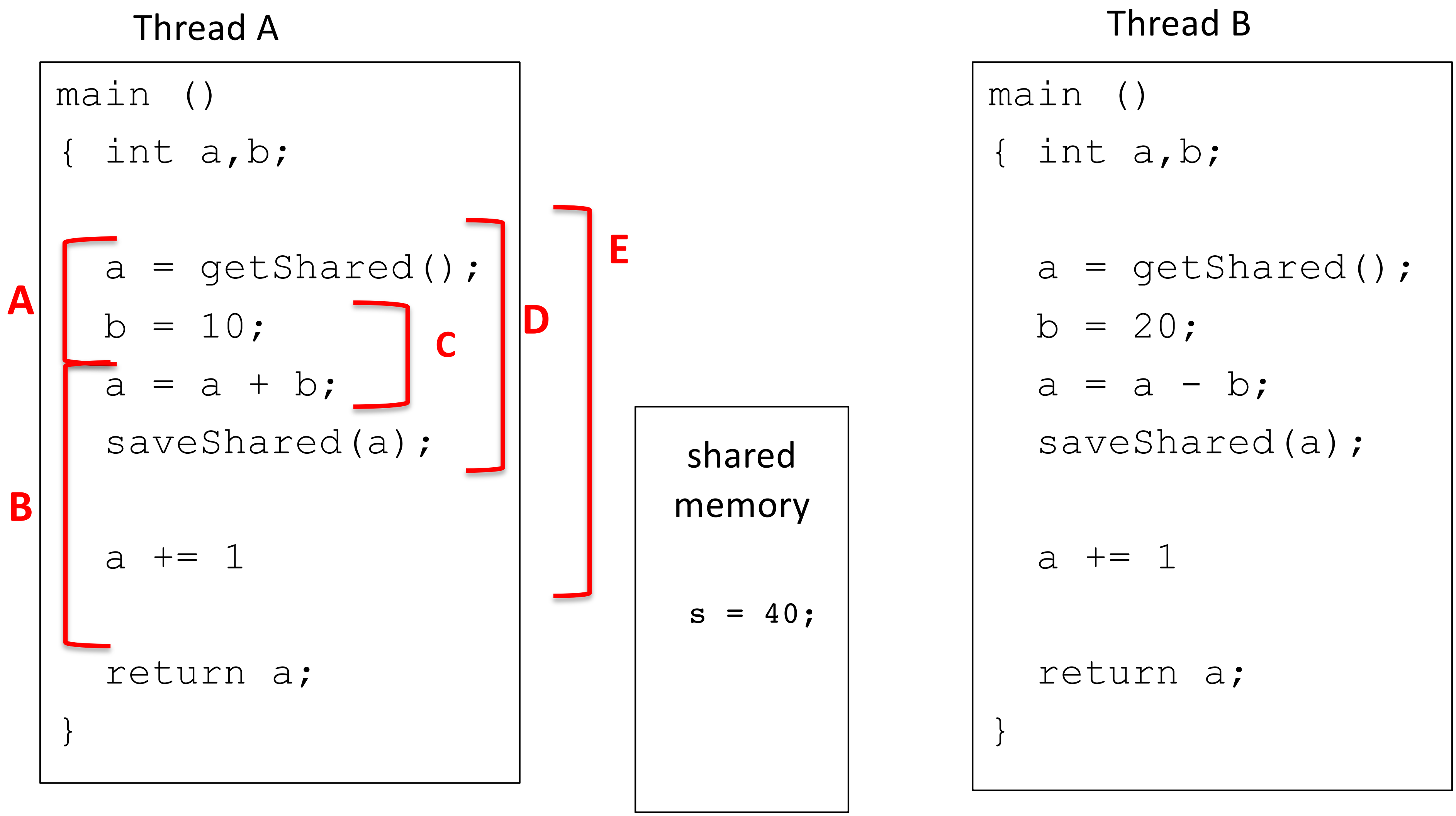
2. Use synchronization to enforce mutual exclusion
   – Only one thread active in a critical section

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?



Which code region is a critical section?

<u>read + modify + write of shared variable</u>

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);

  return a;
}
```

shared memory

```
s = 40;
```

Thread B

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

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

  return a;
}
```

If A runs first

Thread A

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

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

  return a;
}
```

Thread B

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

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

  return a;
}
```

shared memory

```
(s = 40)
s = 50
```

B runs after A Completes

Thread A

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

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

  return a;
}
```

Thread B

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

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

  return a;
}
```

shared memory

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

What about interleaving?

Thread A

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

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

  return a;
}
```

Thread B

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

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

  return a;
}
```

shared memory

```
s = 40;
```

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

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

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1

```
lock (mutex);
< critical section >
unlock (mutex);
```

Atomicity: run the entire instruction without interruption.

Mutual Exclusion with Semaphores

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

T_0

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1

```
lock (mutex);
< critical section >
unlock (mutex);
```

<u>Atomicity: run the entire instruction without interruption.</u>

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

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1

```
lock (mutex);
< critical section >
unlock (mutex);
```

<u>Atomicity: run the entire instruction without interruption.</u>

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 Executio

Mutual Exclusion with Semaphores

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

T_0

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1 (blocked)

```
lock (mutex);
< critical section >
unlock (mutex);
```

<u>Atomicity: run the entire instruction without interruption.</u>

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

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1 (blocked)

```
lock (mutex);
< critical section >
unlock (mutex);
```

<u>Atomicity: run the entire instruction without interruption.</u>

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

Mutual Exclusion with Semaphores

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

T_0

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1 (blocked)

```
lock (mutex);
< critical section >
unlock (mutex);
```

<u>Atomicity: run the entire instruction without interruption.</u>

T_0: Completes execution of critical section

```
Updates mutex value = 1. (release lock)
```

Atomic Executio

Mutual Exclusion with Semaphores

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

T_0

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1

```
lock (mutex);
< critical section >
unlock (mutex);
```

<u>Atomicity: run the entire instruction without interruption.</u>

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

```
lock (mutex);
< critical section >
unlock (mutex);
```

T_1

```
lock (mutex);
< critical section >
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.

Synchronization: More than Mutexes

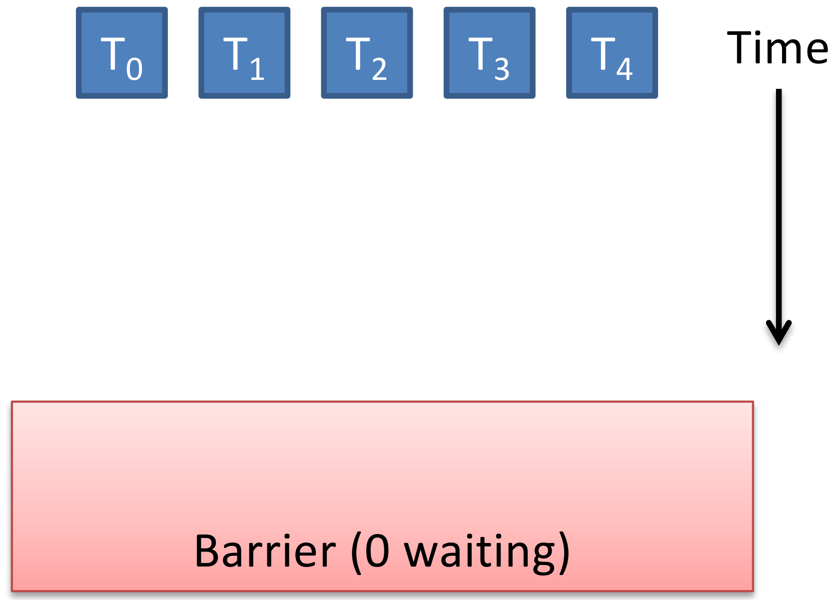
- “I want to block a thread until something specific happens.”
  - Condition variable: wait for a condition to be true
- “I want all my threads to sync up at the same point.”
  - Barrier: wait for everyone to catch up.

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

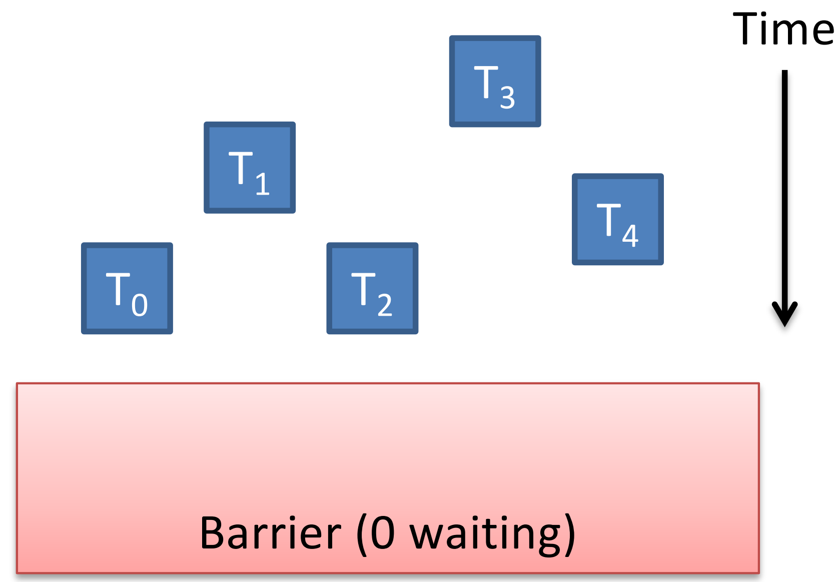
```
shared barrier b;

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)
  }
}
```

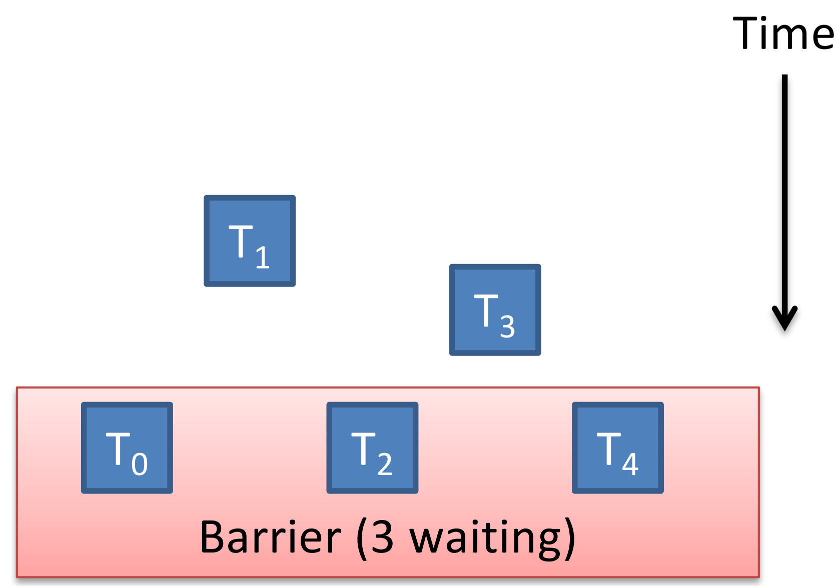
Threads make progress computing current round at different rates.



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)
  }
}
```

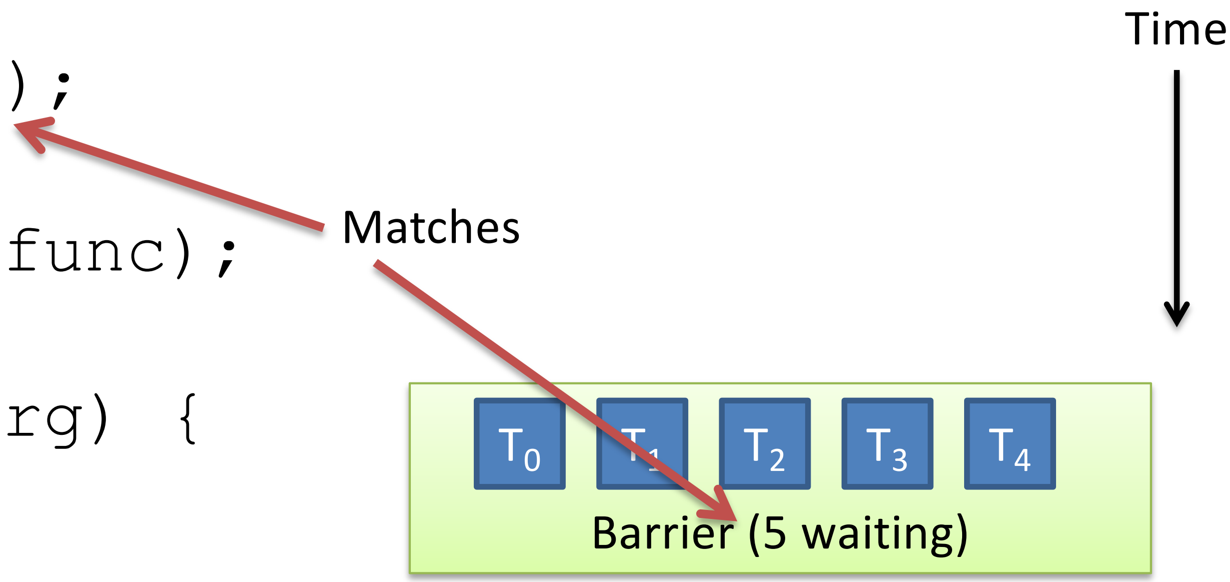
Threads that make it to barrier must wait for all others to get there.



Barrier Example, N Threads

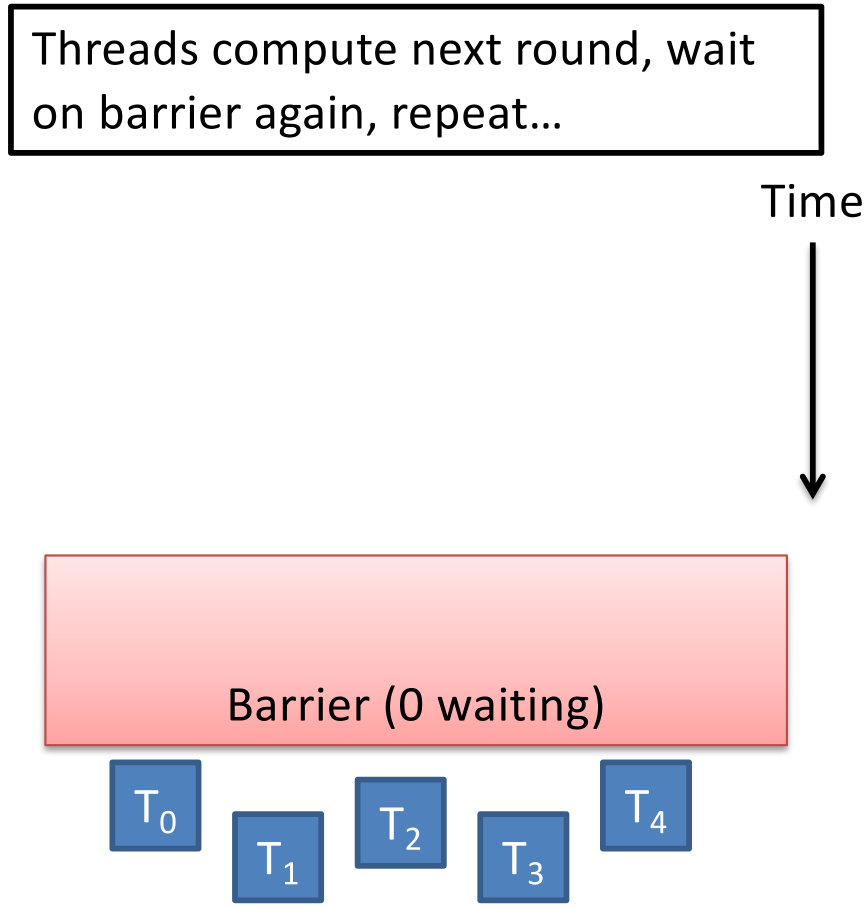
Barrier allows threads to pass when N threads reach it.

```
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)
  }
}
```



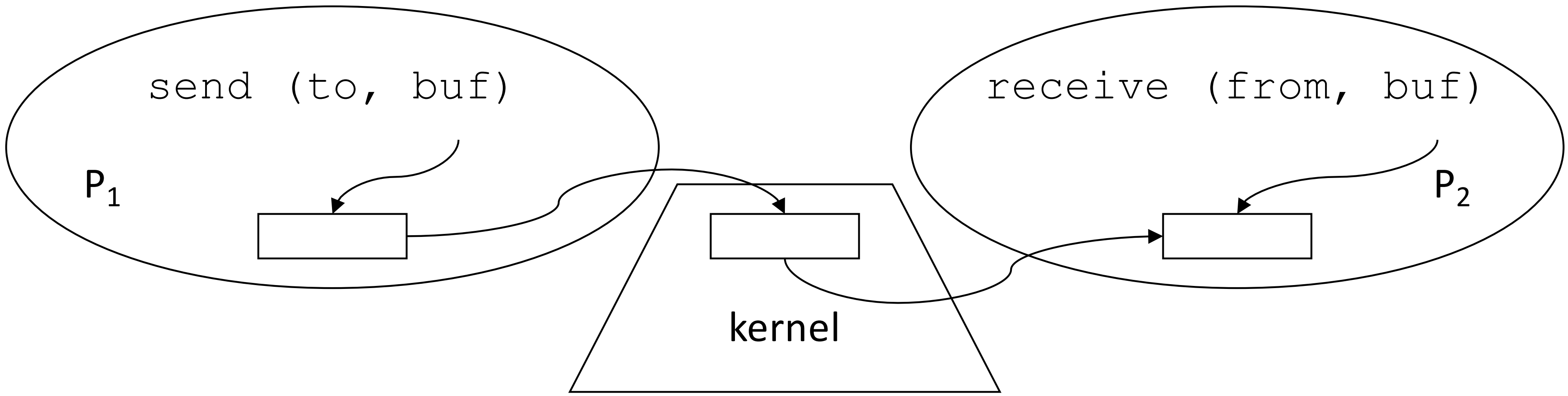
Synchronization: More than Mutexes

- "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

Additional Slides: Solution to the Race Condition

Solution with mutexes

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;
}
```

Using Locks

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;
}
```

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by:
Nobody

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by:
Thread A

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by:
Thread A

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by: Thread A

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by: Nobody

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by:
Thread B

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by:
Nobody

Using Locks

Thread A

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

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

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

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by:
Nobody

- No matter how we order threads or when we context switch, result will always be 30, like we expected (and probably wanted).