Synchronization: Basics

15-213/14-513/15-513: Introduction to Computer Systems
24th Lecture, November 29, 2022

Instructors:

Dave Andersen (15-213)

Zack Weinberg (15-213)

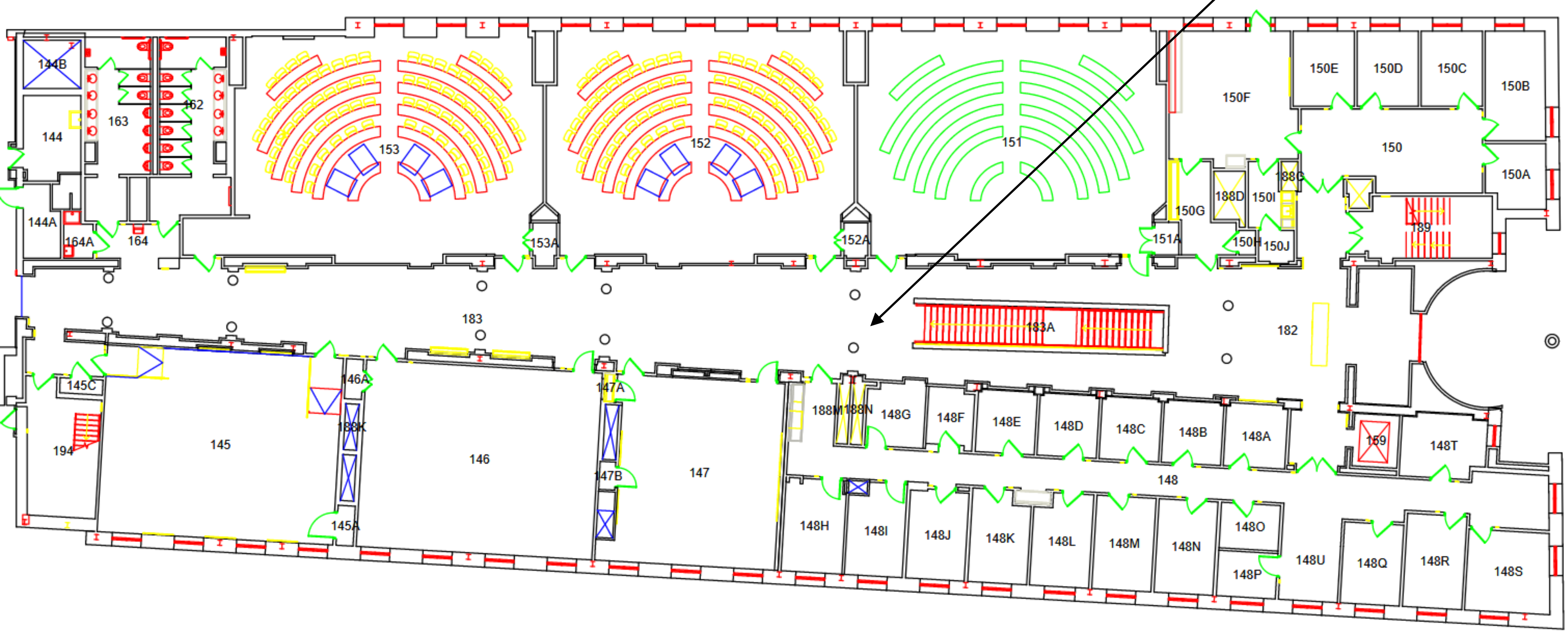
Brian Railing (15-513)

David Varodayan (14-513)

Final Exam Logistics

- Friday, 16 December, 5:30–8:30pm (Pgh time)
- Go to Posner Hall first floor main corridor
- We will meet you and direct you to rooms

Meet TAs here



More Final Exam Logistics

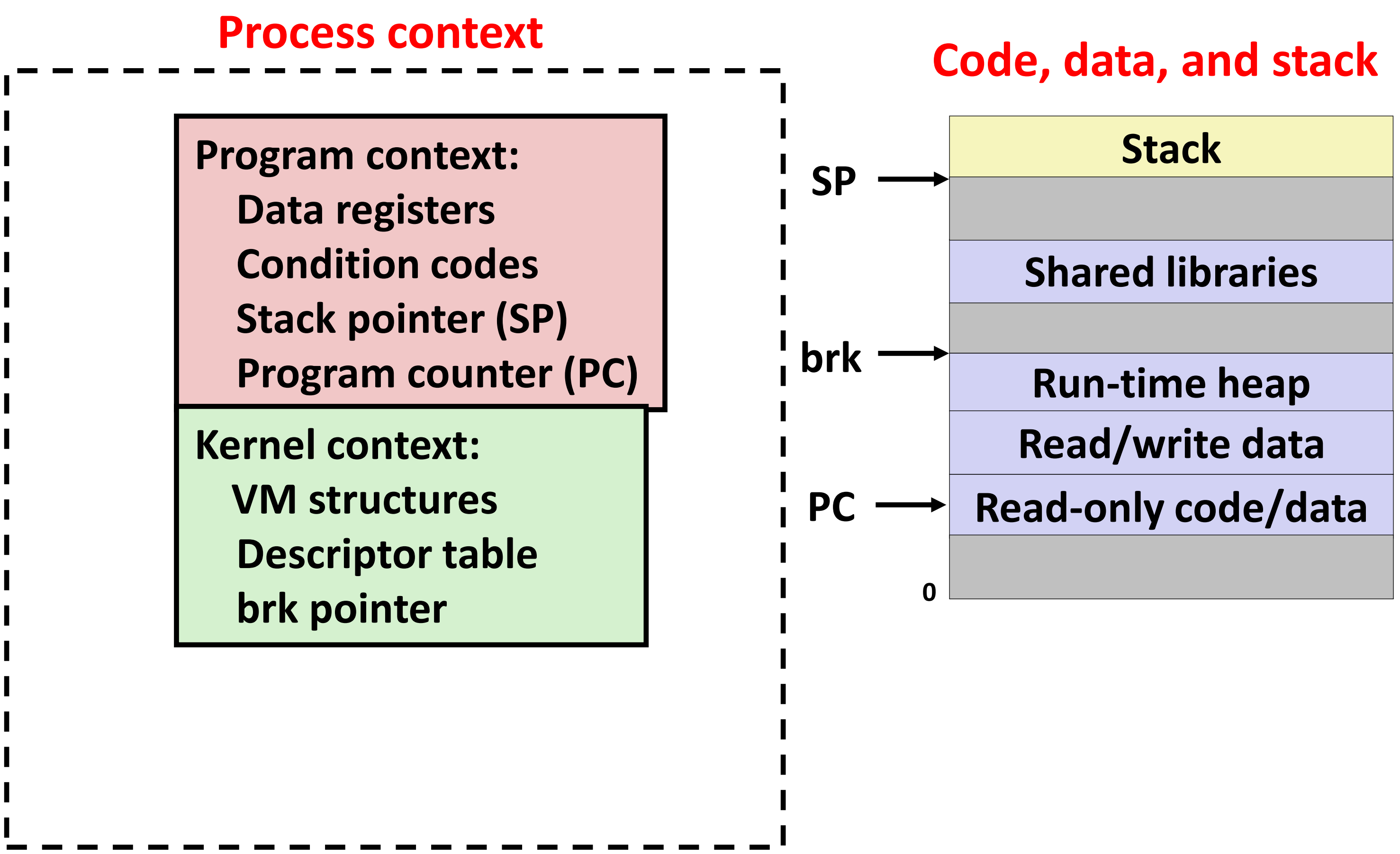
- **Make-up final exam session Monday 19 December**
  - Location and time TBD
- **Final exam review session: not yet scheduled**
- **If you have disability accommodations**
  - Make sure they're on file with the disabilities office
  - Also fill out the form below
  - You will take the exam at the Disability Resources Testing Center (5136 Margaret Morrison Street); do not go to Posner
- **Need any sort of adjustment to exam logistics?**
  - https://forms.gle/UVutWayszmxM89JP9
- **More details:**
  - https://piazza.com/class/l6ff8gpm6nt247/post/1950
  - https://www.cs.cmu.edu/~213/exams.html

Today

- **Threads review**
- **Sharing and Data Races**
- **Fixing Data Races**
  - Mutexes
  - Semaphores
  - Atomic memory operations

Traditional View of a Process

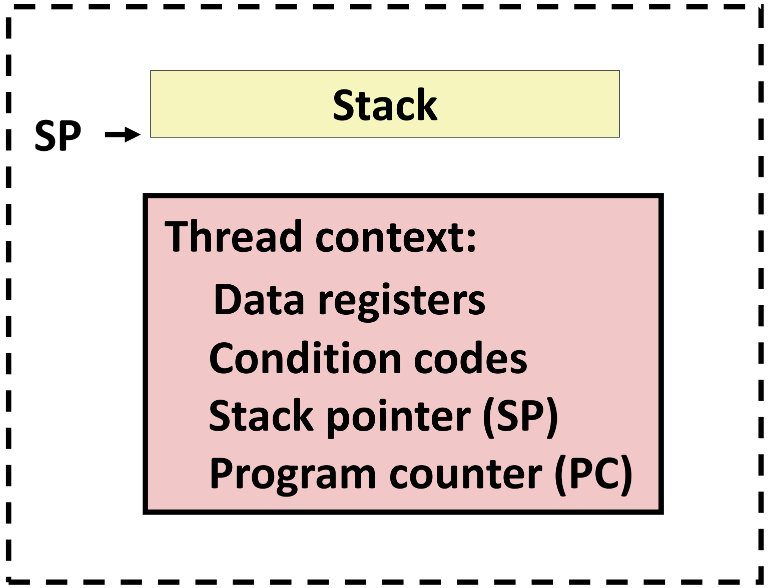
- Process = process context + code, data, and stack



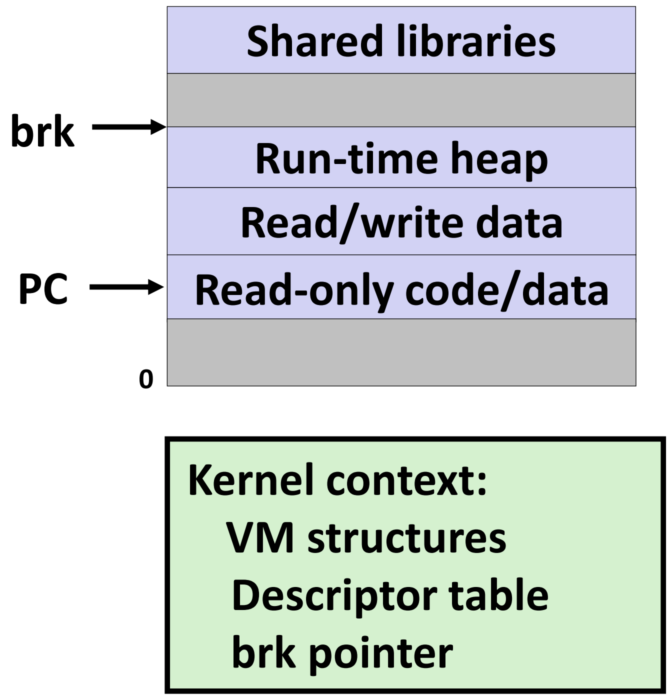
Alternate View of a Process

- Process = thread + (code, data, and kernel context)

Thread (main thread)

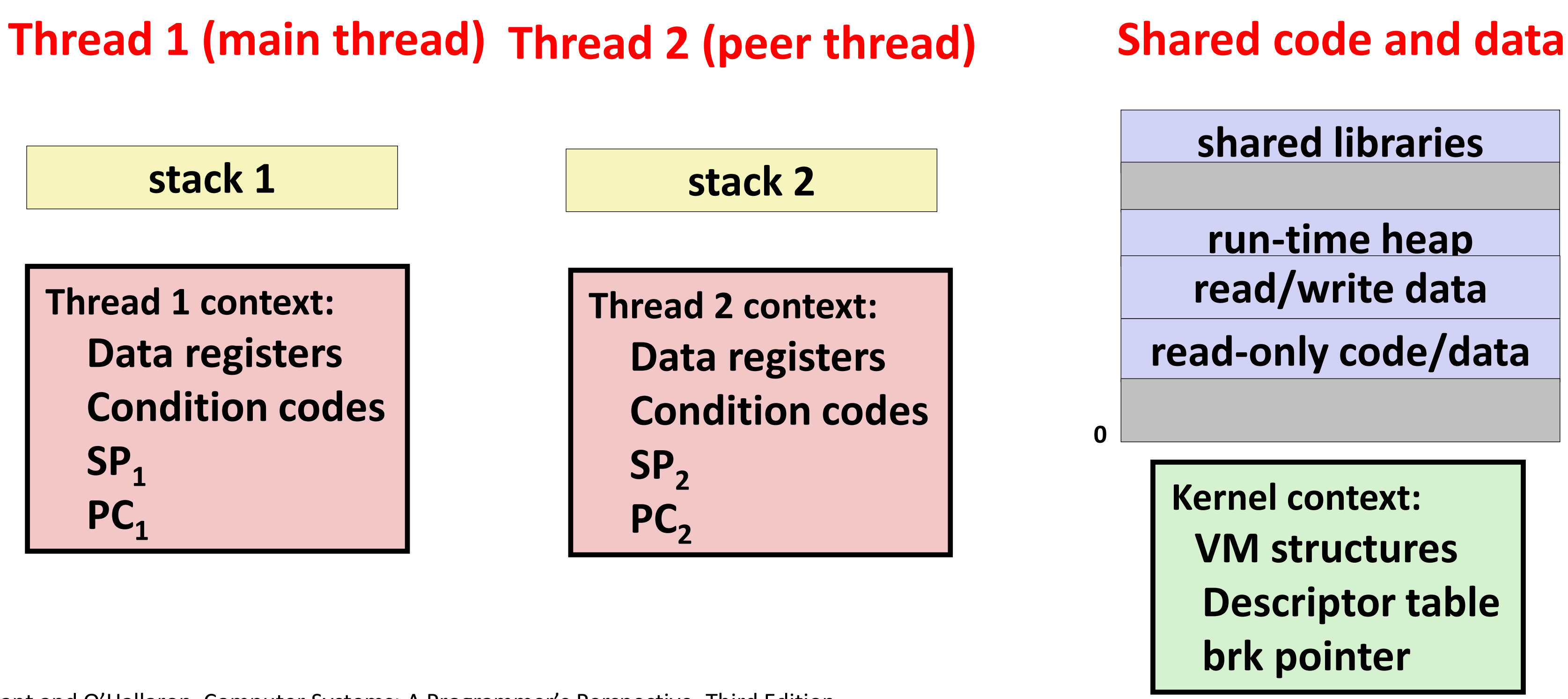


Code, data, and kernel context

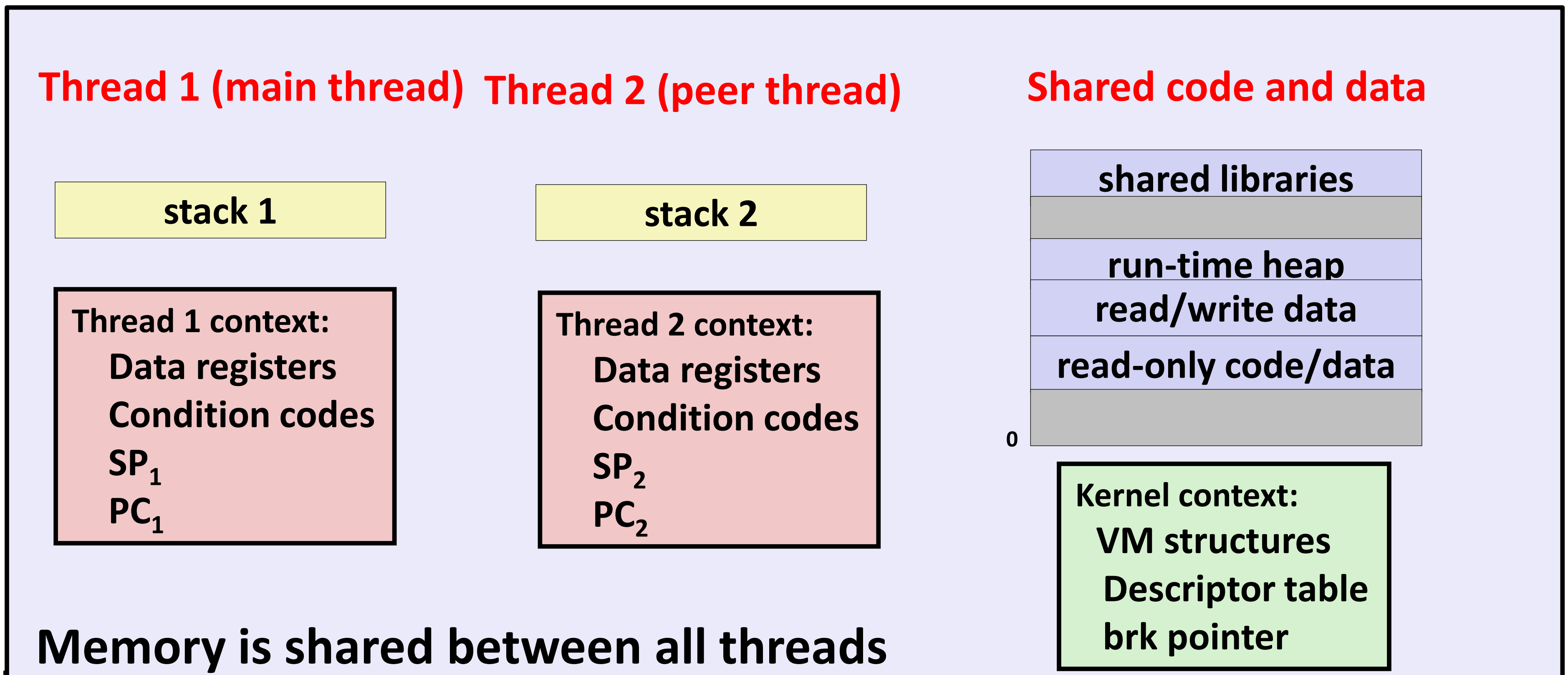


A Process With Multiple Threads

- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)



Don’t let picture confuse you!



Today

- Threads review
- **Sharing and Data Races**
- Fixing Data Races
  - Mutexes
  - Semaphores
  - Atomic memory operations

Shared Variables in Threaded C Programs

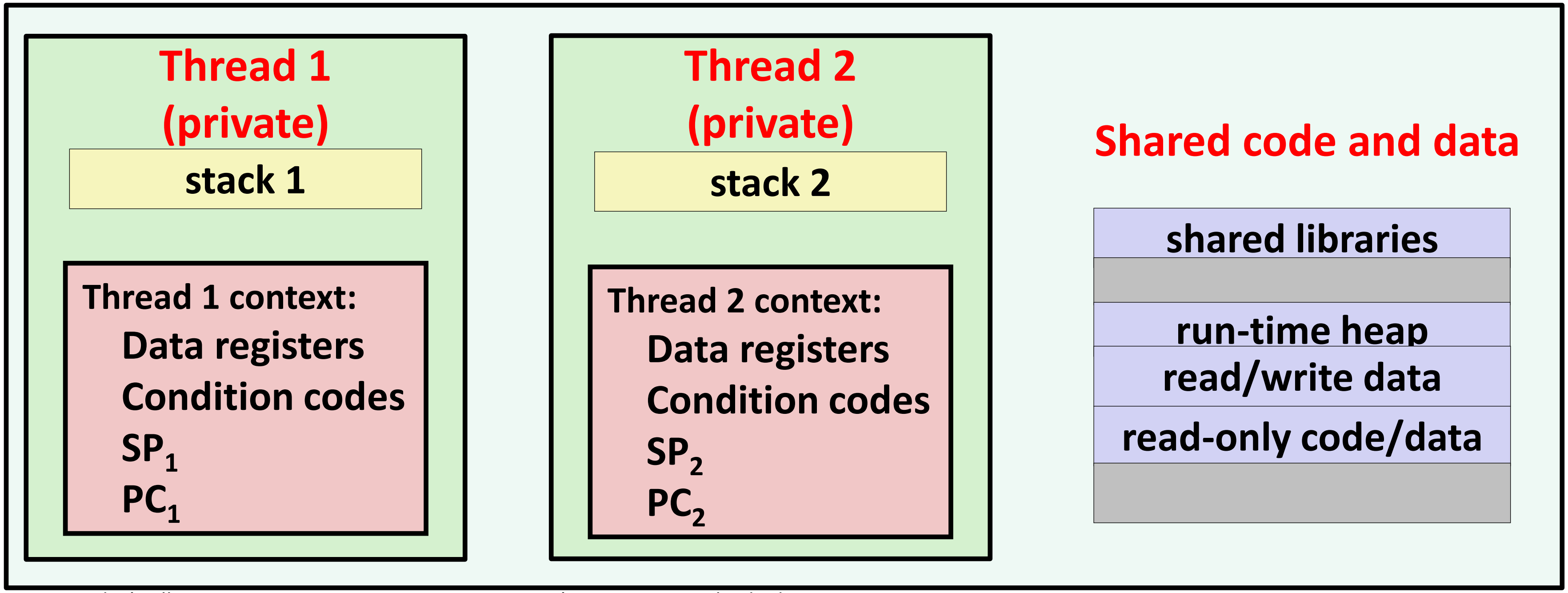
- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- ***Def:* A variable `x` is *shared* if and only if multiple threads reference some instance of `x`.**

- **Requires answers to the following questions:**
  - What is the memory model for threads?
  - How are instances of variables mapped to memory?
  - How many threads might reference each of these instances?

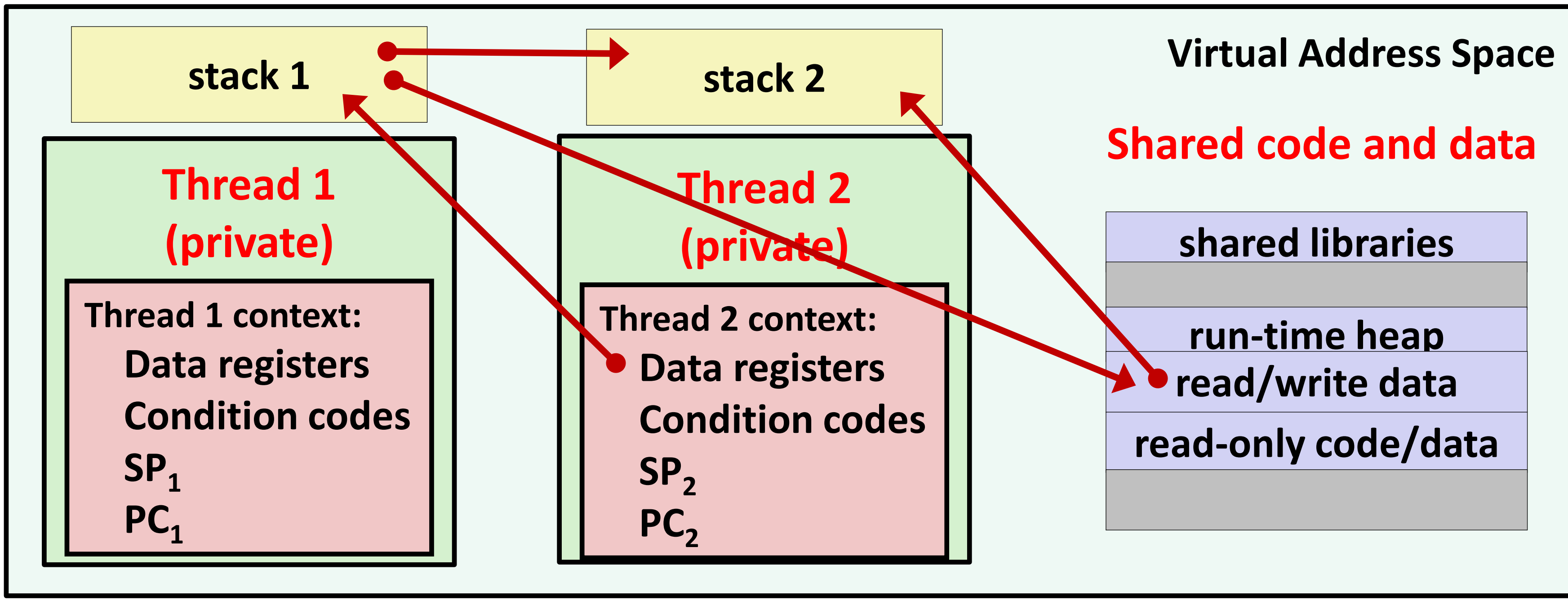
Threads Memory Model: Conceptual

- **Multiple threads run within the context of a single process**
- **Each thread has its own separate thread context**
  - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
- **All threads share the remaining process context**
  - Code, data, heap, and shared library segments of the process virtual address space
  - Open files and installed handlers



Threads Memory Model: Actual

- **Separation of data is not strictly enforced:**
  - Register values are truly separate and protected, but…
  - Any thread can read and write the stack of any other thread



The mismatch between the conceptual and operation model is a source of confusion and errors

Three Ways to Pass Thread Arg

- **Malloc/free**
  - Producer malloc's space, passes pointer to pthread_create
  - Consumer dereferences pointer, frees space
  - Always works; necessary for passing large amounts of data
- **Cast of int**
  - Producer casts an int/long to void*, passes to pthread_create
  - Consumer casts void* argument back to int/long
  - Works for small amounts of data (one number)
- **INCORRECT: Pointer to stack slot**
  - Producer passes address to producer's stack in pthread_create
  - Consumer dereferences pointer
  - Why is this unsafe?

Passing an argument to a thread

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++)
     Pthread_create(&tids[i],
                    NULL,
                    thread,
                    &hist[i]);
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   *(int *)vargp += 1;
   return NULL;
}
```

- Each thread receives a *unique pointer*

```
void check(void) {
   for (int i=0; i<N; i++) {
     if (hist[i] != 1) {
       printf("Failed at %d\n", i);
       exit(-1);
     }
   }
   printf("OK\n");
}
```

Passing an argument to a thread – Also OK

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
    long i;
    pthread_t tids[N];

    for (i = 0; i < N; i++)
      Pthread_create(&tids[i],
                     NULL,
                     thread,
                     (void *)i);
    for (i = 0; i < N; i++)
      Pthread_join(tids[i], NULL);
    check();
}
```

```
void *thread(void *vargp)
{
    hist[(long)vargp] += 1;
    return NULL;
}
```

- Each thread receives a *unique array index*
- Casting from long to void* and back is safe

Passing an argument to a thread – Also OK

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++)
     long* p = Malloc(sizeof(long));
     *p = i;
     Pthread_create(&tids[i],
                    NULL,
                    thread,
                    p);
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   hist[*(long *)vargp] += 1;
   free(vargp);
   return NULL;
}
```

- Each thread receives a *unique array index*
- Malloc in parent, free in thread
- Necessary if passing structs

Passing an argument to a thread – WRONG!

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++)
     Pthread_create(&tids[i],
                    NULL,
                    thread,
                    &i);
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   hist[*(long *)vargp] += 1;
   return NULL;
}
```

- **Each thread receives *the same pointer*, to `i` in main**
- **Data race: each thread *may or may not* read a unique array index from `i` in main**

Shared Variables in Threaded C Programs

- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- ***Def:* A variable `x` is *shared* if and only if multiple threads reference some instance of `x`.**

- **Requires answers to the following questions:**
  - What is the memory model for threads?
  - How are instances of variables mapped to memory?
  - How many threads might reference each of these instances?

Mapping Variable Instances to Memory

- **Global variables**
  - Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**
- **Local automatic variables**
  - Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**
- **Local static variables**
  - Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable.**
- **`errno` is special**
  - Declared outside a function, but **each thread stack contains one instance**

Mapping Variable Instances to Memory

```
char **ptr;  /* global var */

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
    return NULL;
}
```

Mapping Variable Instances to Memory

Global var: 1 instance (`ptr` [data])

Local auto vars: 1 instance (`i.m, msgs.m, tid.m`)

```
char **ptr;  /* global var */

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

Local auto var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
          myid, ptr[myid], ++cnt);
    return NULL;
}
```

Local static var: 1 instance (`cnt` [data])

Shared Variable Analysis

- **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL, thread,(void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

Shared Variable Analysis

- **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

- **Answer: A variable `x` is shared iff multiple threads reference at least one instance of `x`. Thus:**
  - `ptr`, `cnt`, and `msgs` are shared
  - `i` and `myid` are *not* shared

Synchronizing Threads

- Shared variables are handy…
- …but you risk *data races* and *synchronization errors*.

```
static unsigned long cnt = 0;

void *incr_thread(void *arg) {
  unsigned long i;
  unsigned long niters =
     (unsigned long) arg;

  for (i = 0; i < niters; i++) {
    cnt++;
  }
}
```

```
int main(int argc, char **argv) {
  unsigned long niters =
    strtoul(argv[1], NULL, 10);

  pthread_t t1, t2;
  Pthread_create(&t1, NULL,
                 incr_thread,
                 (void *)niters);
  Pthread_create(&t2, NULL,
                 incr_thread,
                 (void *)niters);
  Pthread_join(&t1, NULL);
  Pthread_join(&t2, NULL);
  if (cnt != 2*niters) {
    printf("FAIL: cnt=%lu not %lu\n",
           cnt, 2*niters;
    return 1;
  } else {
    printf("OK: cnt=%lu\n", cnt);
    return 0;
  }
}
```

Coding demo 1:
Counting to 20,000 incorrectly
(with threads)

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i : Head

L_i : Load cnt
U_i : Update cnt
S_i : Store cnt

T_i : Tail

Concurrent Execution

- ***Key idea:*** **Any interleaving of instructions is possible, and some give an unexpected result!**
  - I_i denotes that thread i executes instruction I
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

Thread 1 critical section

Thread 2 critical section

OK

Concurrent Execution (cont)

- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

Concurrent Execution (cont)

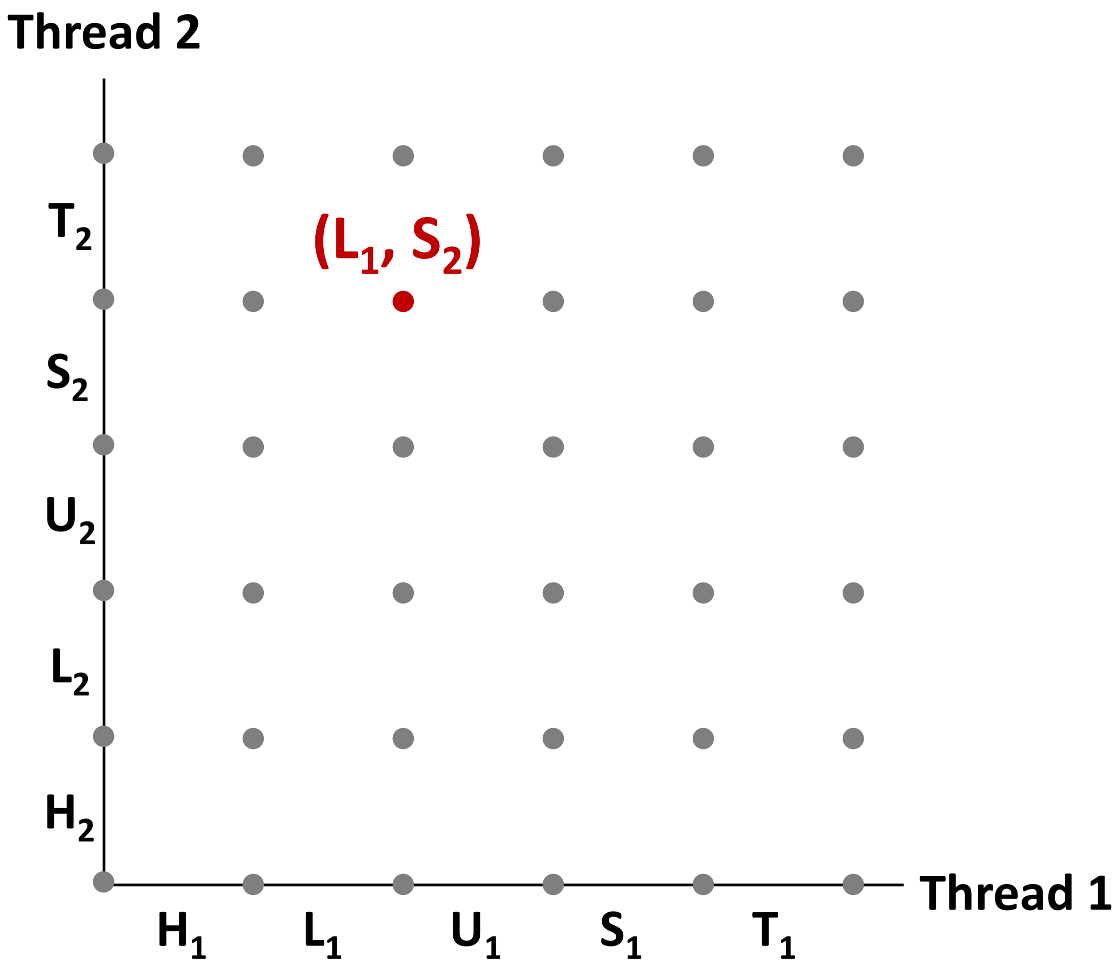
- How about this ordering?

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			1
2	T_2			1

Oops!

- We can analyze the behavior using a *progress graph*

Progress Graphs



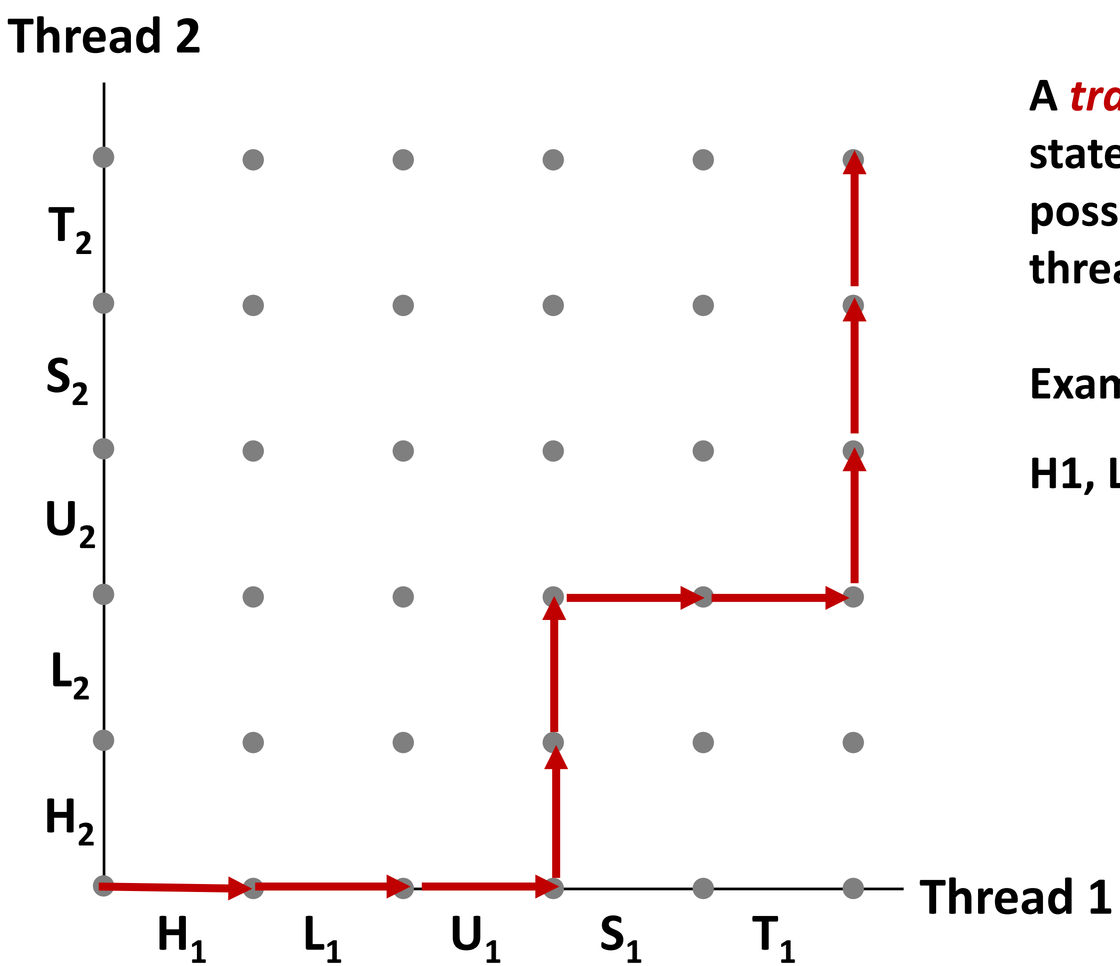
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible ***execution state*** ($Inst_1$, $Inst_2$).

E.g., **(L_1, S_2)** denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs

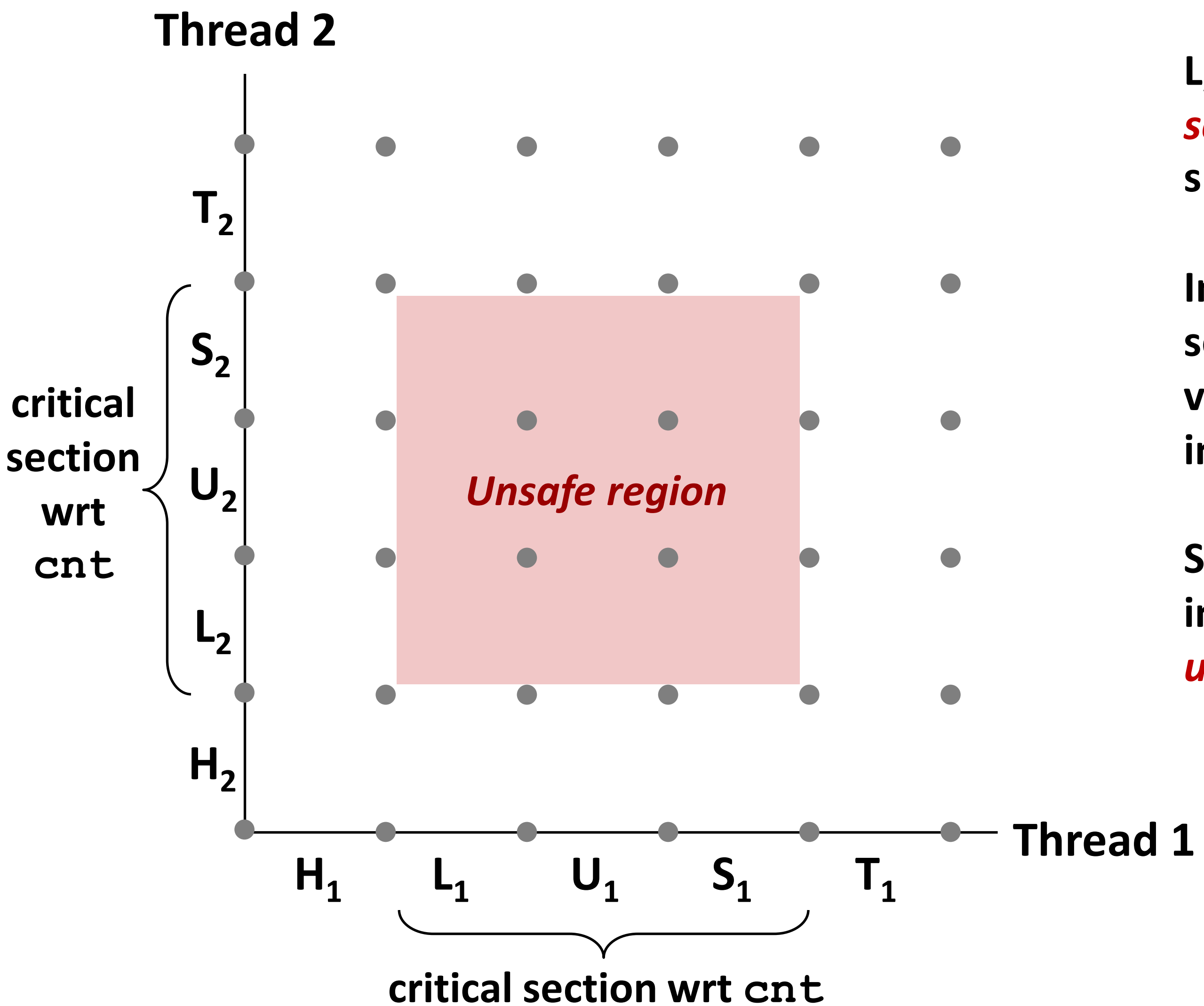


A ***trajectory*** **is a sequence of legal state transitions that describes one possible concurrent execution of the threads.**

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

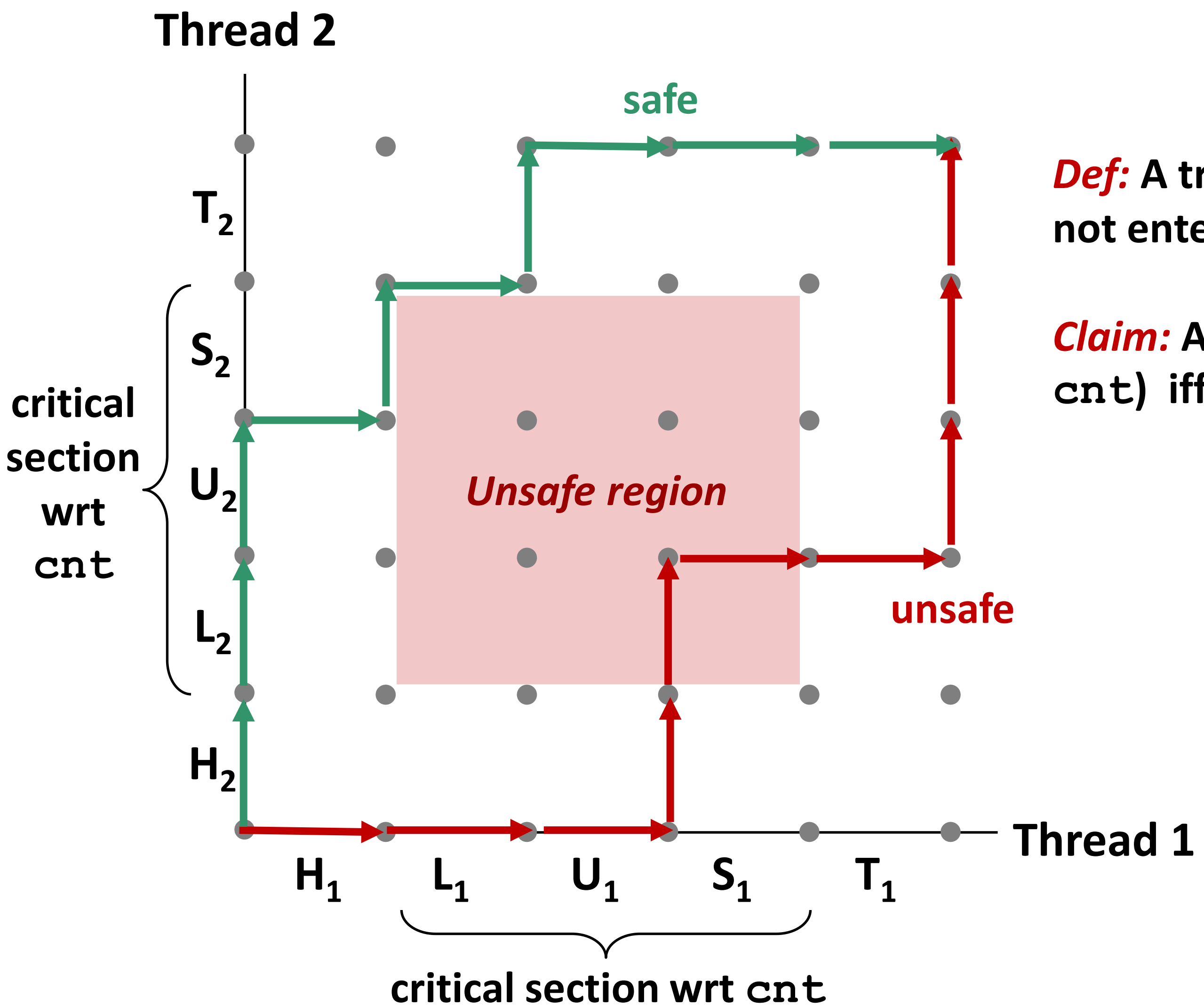


L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

Instructions in critical sections (wrt some shared variable) should not be interleaved

Sets of states where such interleaving occurs form ***unsafe regions***

Critical Sections and Unsafe Regions



Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt `cnt`) iff it is safe

Quiz time!

https://canvas.cmu.edu/courses/30386/quizzes/86859

Today

- Threads review
- Sharing and Data Races
- **Fixing Data Races**
  - Mutexes
  - Semaphores
  - Atomic memory operations

Enforcing Mutual Exclusion

- *Question:* **How can we guarantee a safe trajectory?**

- **Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.**
  - Need to guarantee ***mutually exclusive access*** to each critical section.

```
static unsigned long cnt = 0;
static pthread_mutex_t lock =
  PTHREAD_MUTEX_INITIALIZER;

void *incr_thread(void *arg) {
  unsigned long i;
  unsigned long niters =
    (unsigned long) arg;

  for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&lock);
    cnt++;
    pthread_mutex_unlock(&lock);
  }
}
```

Coding demo 2:
Counting to 20,000 correctly
(with threads and a mutex)

MUTual EXclusion (mutex)

- **Mutex: opaque object which is either *locked* or *unlocked***
  - Boolean value, but cannot do math on it
  - Starts out unlocked
  - Two operations:
- **lock(m)**
  - If the mutex is currently not locked, lock it and return
  - Otherwise, wait until it becomes unlocked, then retry
- **unlock(m)**
  - Can only be called when mutex is locked, by the code that locked it
  - Change mutex to unlocked

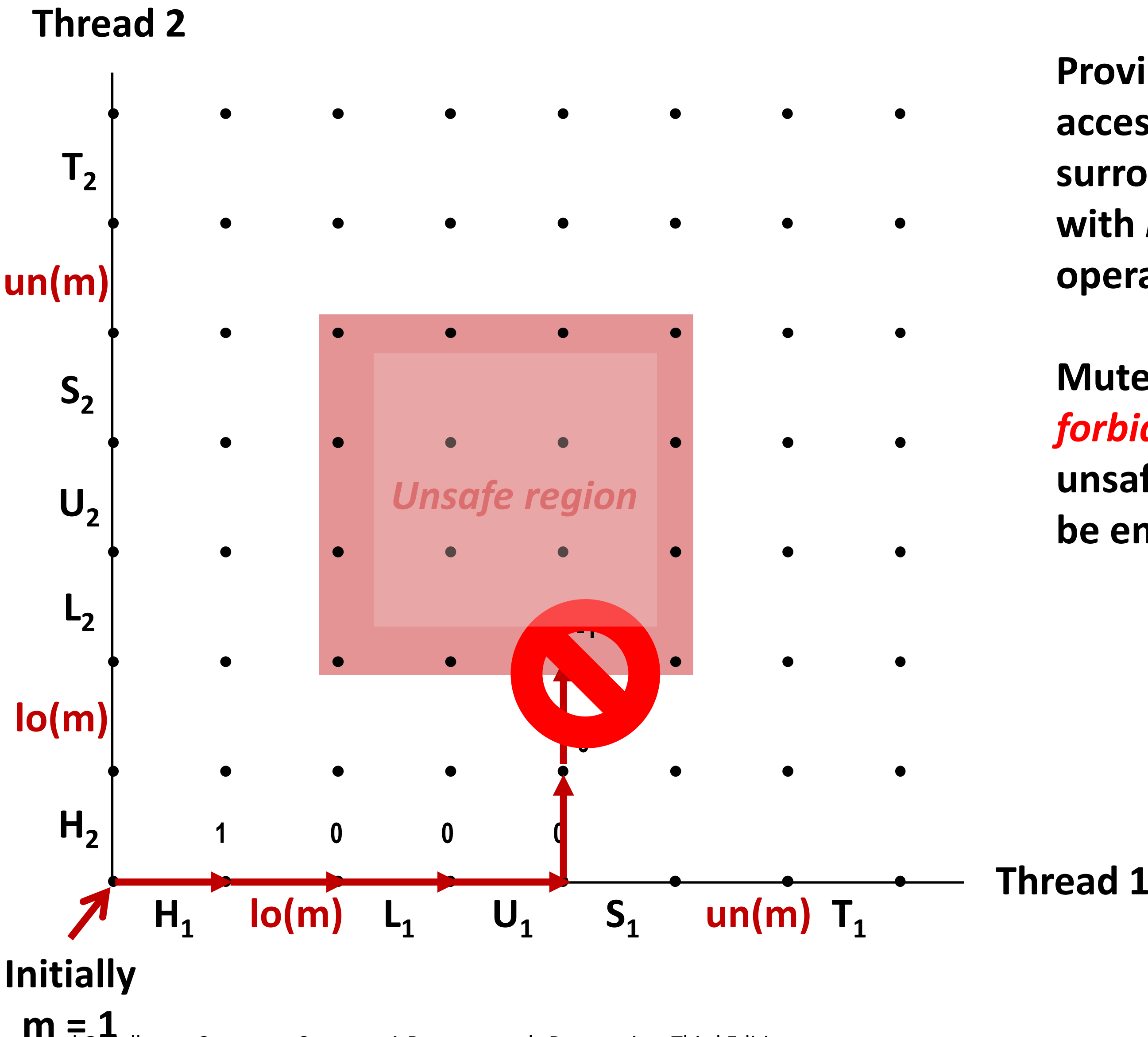
Mutex implementation (partial)

```
/**
 * void pthread_mutex_lock(pthread_mutex_t *mtx)
 * Lock the mutex pointed to by MTX.  If it is already locked,
 * first sleep until it becomes unlocked.
 */
pthread_mutex_lock:
        call    gettid            // current thread ID now in %eax
        mov     $1, %edx          // increment
   lock xadd    %edx, MUTEX_CONTENDERS(%rdi)
        // %edx now holds _previous_ value of mtx->contenders
        test    %edx, %edx
        jne     .Lcontended

        // The lock was unlocked, and now we hold it.
        mov     %eax, MUTEX_HOLDER(%rdi)
        ret

.Lcontended:
        // Sleep until another thread calls pthread_mutex_unlock
        // (30 more machine instructions and a system call)
```

Just one of many ways to implement (discussed in 15-410, -418, etc)

All require assistance from the CPU (special instructions)

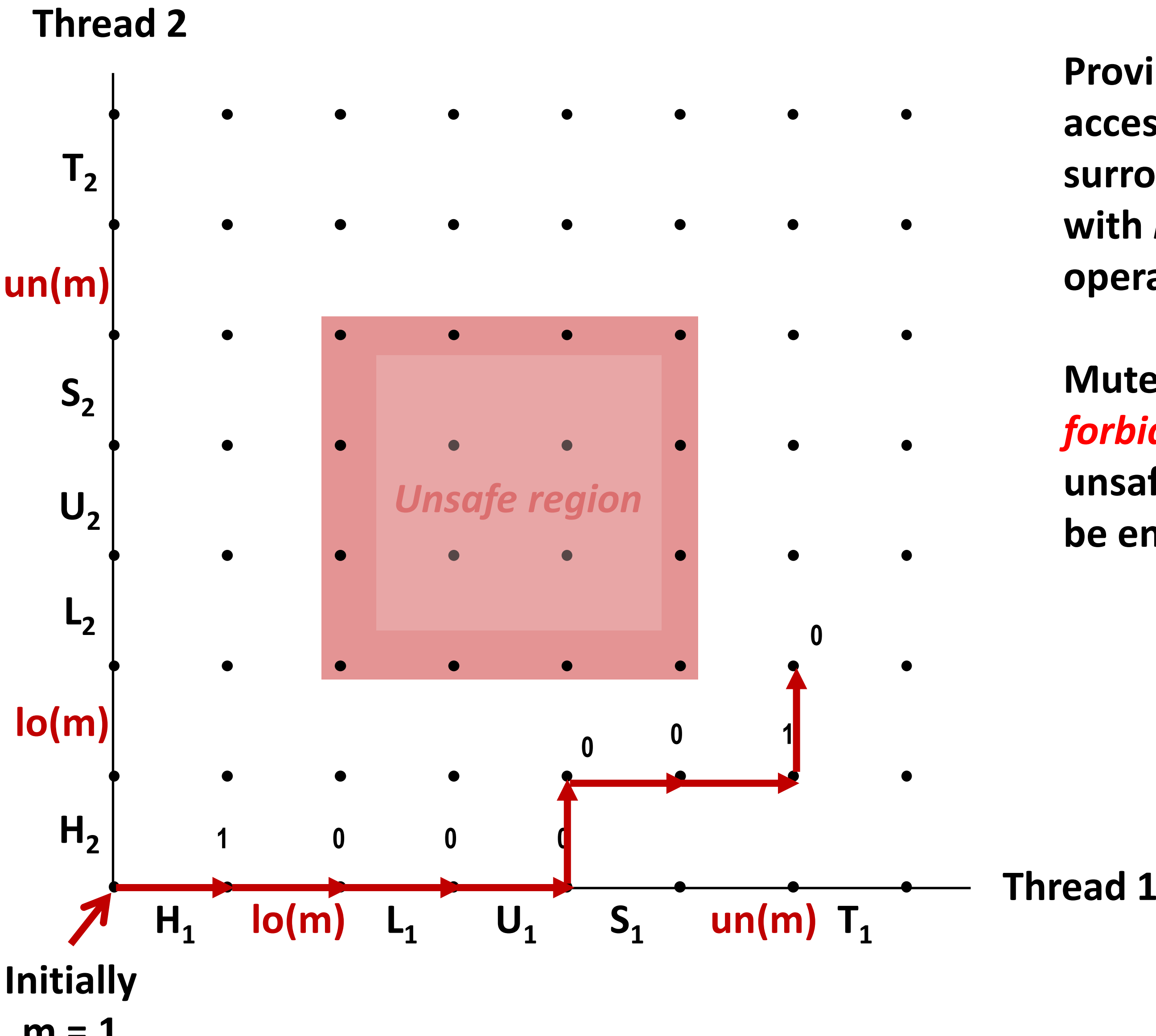
Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

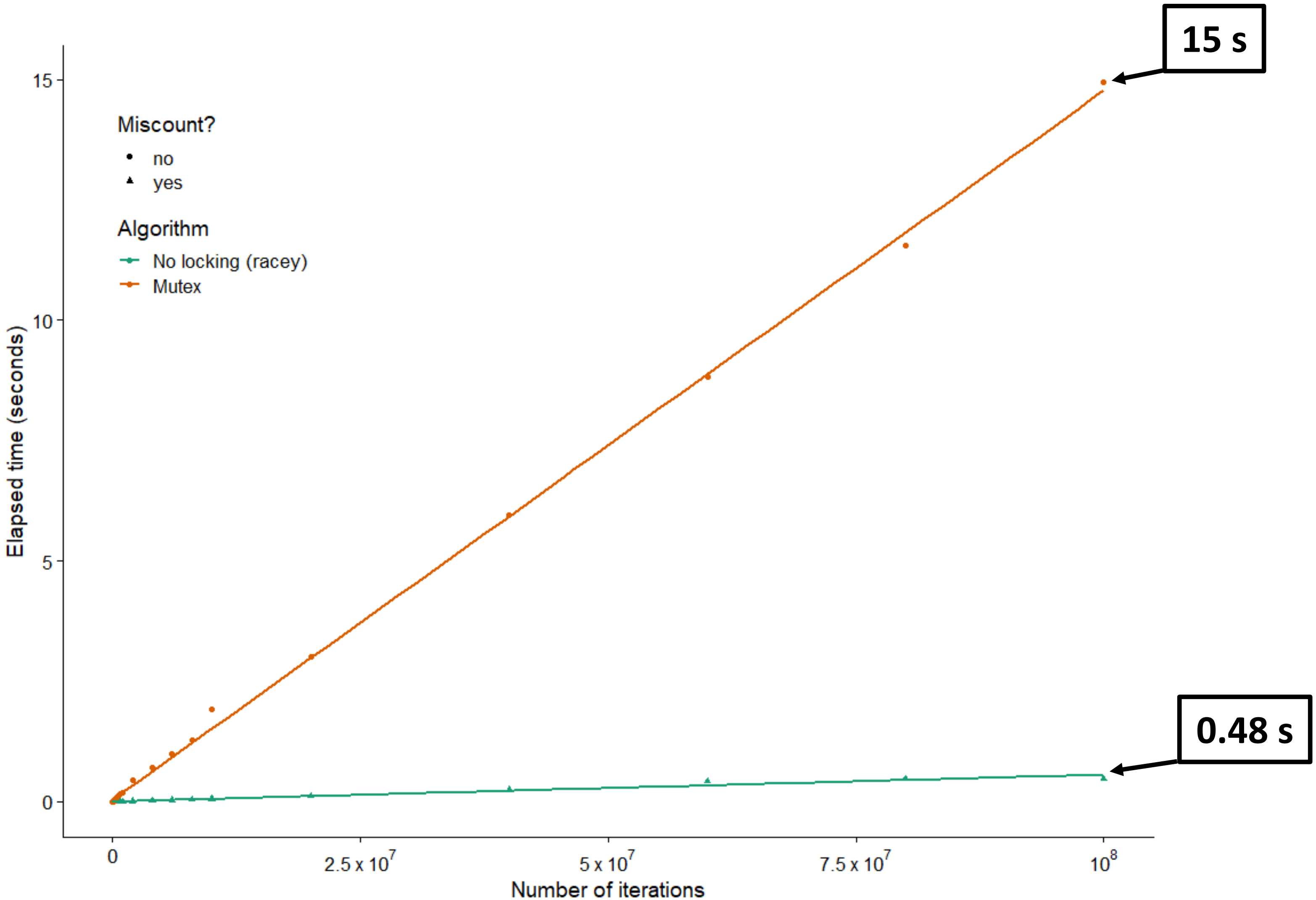
Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

The Cost of Mutexes



Today

- Threads review
- Sharing and Data Races
- Fixing Data Races
  - Mutexes
  - Semaphores
  - Atomic memory operations

```
static unsigned long cnt = 0;
static sem_t lock;

void *incr_thread(void *arg) {
  unsigned long i;
  unsigned long niters =
    (unsigned long) arg;

  for (i = 0; i < niters; i++) {
    sem_wait(&lock);
    cnt++;
    sem_post(&lock);
  }
}
```

```
int main(int argc, char **argv) {
  unsigned long niters =
    strtoul(argv[1], NULL, 10);
  sem_init(&lock, 0, 1);
  // ...
}
```

Coding demo 3:
Counting to 20,000 correctly
(with threads and a semaphore)

Semaphores

- ***Semaphore:*** **generalization of mutex**
  - Unsigned integer value, but cannot do math on it.
  - Created with some value >= 0
  - Two operations:

- **P(s) [“Prolaag,” Dutch shorthand for “try to reduce”]**
  - If *s* is zero, wait for a *V* operation to happen.
  - Then subtract 1 from *s* and return.

- **V(s) [“Verhogen,” Dutch for “increase”]**
  - Add 1 to *s*.
  - If there are any threads waiting inside a *P* operation, resume *one* of them

- **Unlike mutexes, no requirement to call P before calling V**

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */

int sem_init(sem_t *s, int pshared, unsigned int val);
```

Share among processes? (normally you want to pass zero, see manpage for details)

Initial semaphore value

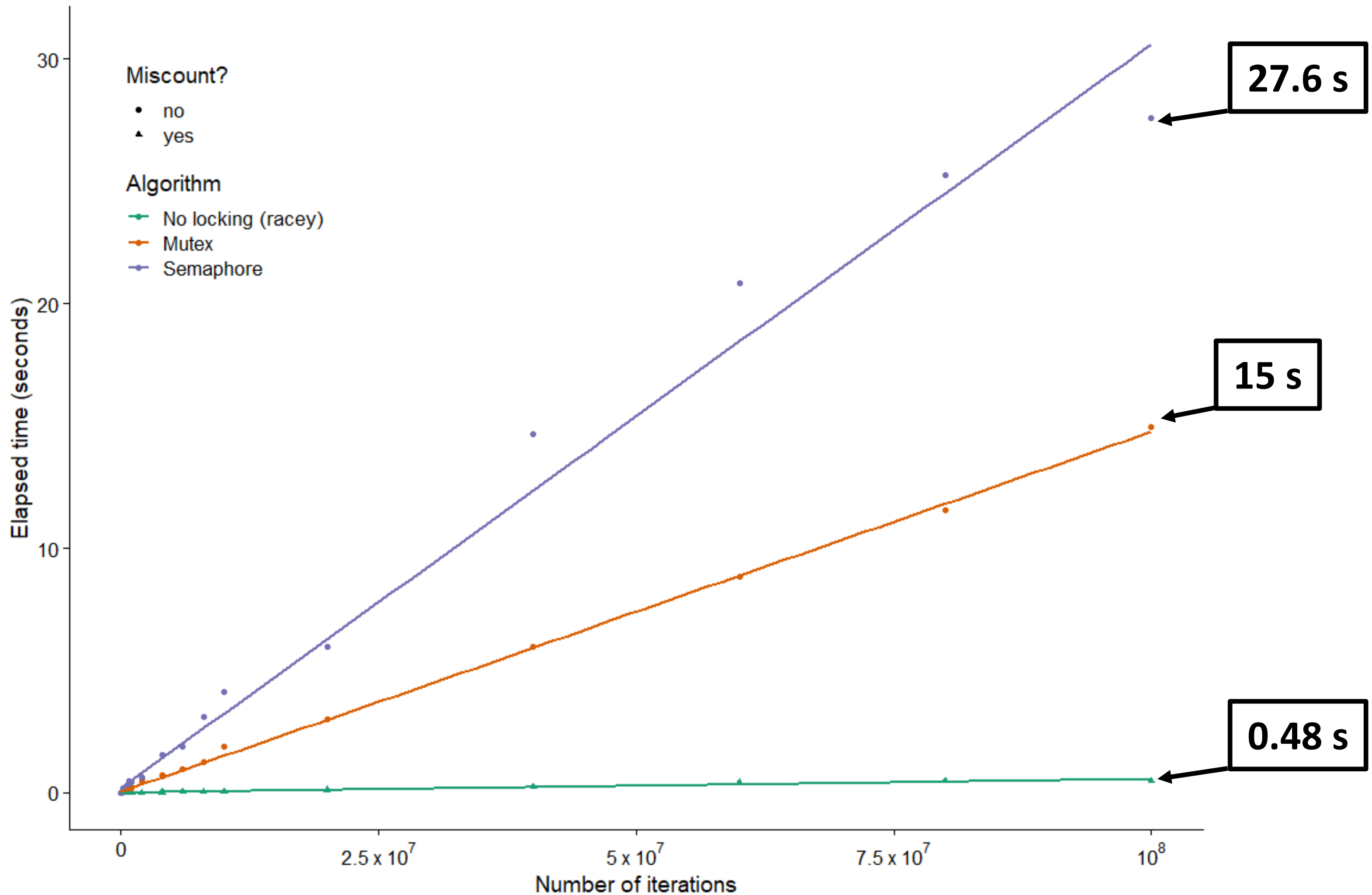
Semaphore implementation (partial)

```
/**
 * void sem_wait(sem_t *sem)
 * Decrement the count of the semaphore pointed to by SEM.  If this
 * would make the count negative, first sleep until it is possible to
 * decrement the count without making it negative.
 */
sem_wait:
        mov       $-1, %edx           // decrement
   lock xadd      %edx, SEM_COUNT(%rdi)
        // %edx now holds _previous_ value of sem->count
        test      %edx, %edx
        jle       .Lclosed
        // The semaphore was open.
        ret

.Lclosed:
        // Sleep until another thread calls sem_post
        // (30 more machine instructions and a system call)
```

Suspiciously similar to a mutex, huh?

(This implementation makes sem_post do most of the work)

The cost of semaphores



Today

- Threads review
- Sharing and Data Races
- **Fixing Data Races**
  - Mutexes
  - Semaphores
  - Atomic memory operations

Atomic memory operations

- **Special hardware instructions**
  - "Test and set," "compare and swap", "exchange and add", …
  - Do a read-modify-write on memory; hardware prevents data races
  - Used to implement mutexes, semaphores, etc.
- **Not going to get into details, but…**
  - Wouldn't it be nice if we could use them directly?
  - Especially when we just want to increment a counter?

```
static _Atomic unsigned long cnt = 0;

void *incr_thread(void *arg) {
  unsigned long i;
  unsigned long niters =
    (unsigned long) arg;
  for (i = 0; i < niters; i++) {
    cnt++;
  }
}
```

Coding demo 4:
Counting to 20,000 correctly
(with threads and C2011 atomics)

Assembly Code for Counter Loop

C code

```
for (i = 0; i < niters; i++)
    cnt++;
```

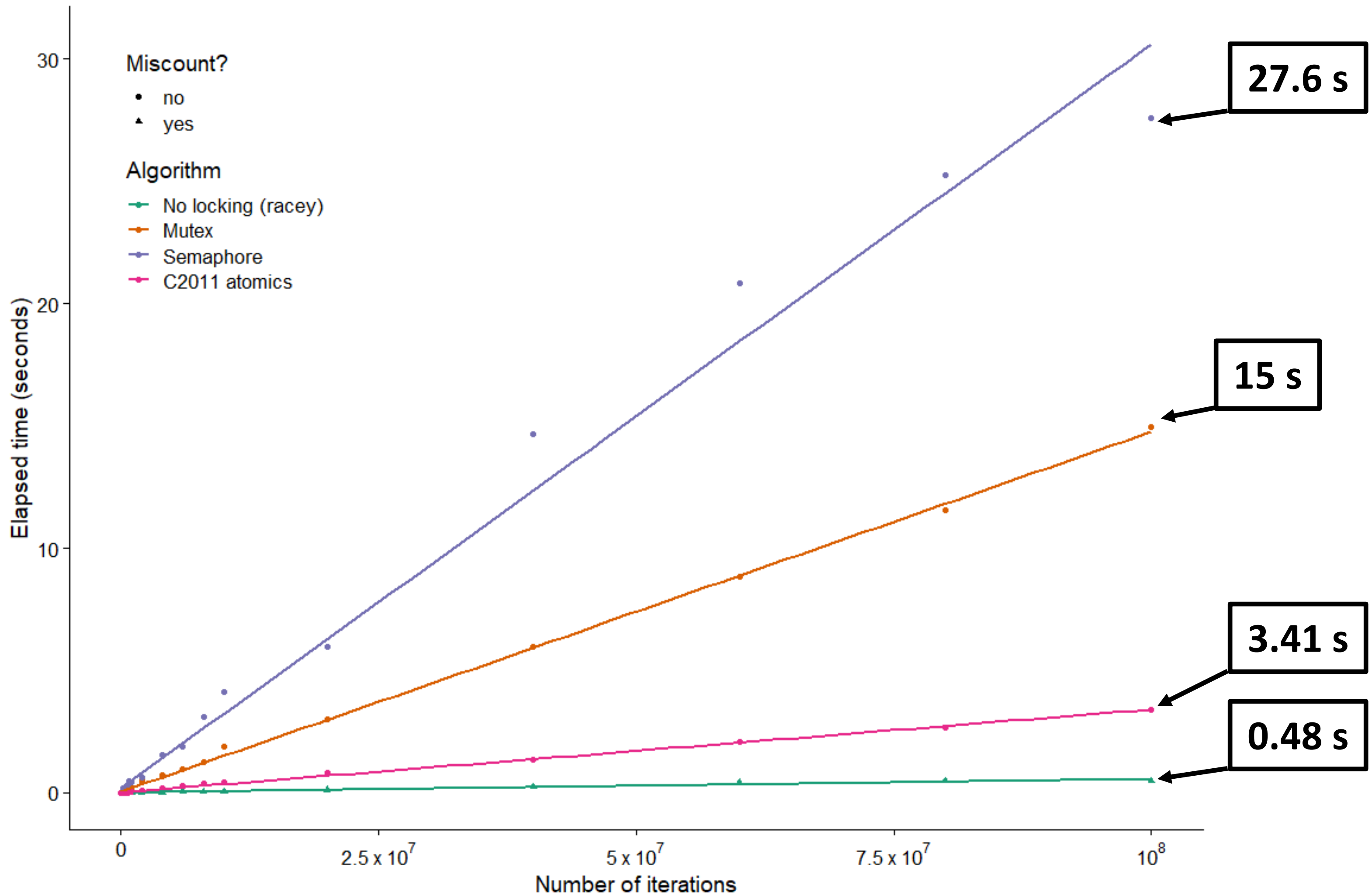
Assembly (unsigned long)

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

Assembly (_Atomic unsigned long)

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    lock addq $1, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

The cost of atomic memory operations



Summary

- **Access shared variables with care to avoid data races.**
  - Crucial to understand which variables are shared in the first place
  - Avoid sharing, if you can
  - Avoid writing from multiple threads, if you can
- **Mutexes help, but…**
  - They're slow
  - (Next time: They can cause problems as well as solve them)
- **Don't use a semaphore when a mutex will do**
  - They're even slower
  - (Next time: When is a semaphore actually useful?)
- **Atomic memory ops are handy, but…**
  - The hardware might not provide the operation you need
  - (Later courses: Tricky to use correctly)