Synchronization: Basics

15-213/14-513/15-513: Introduction to Computer Systems
24th Lecture, July 28, 2022

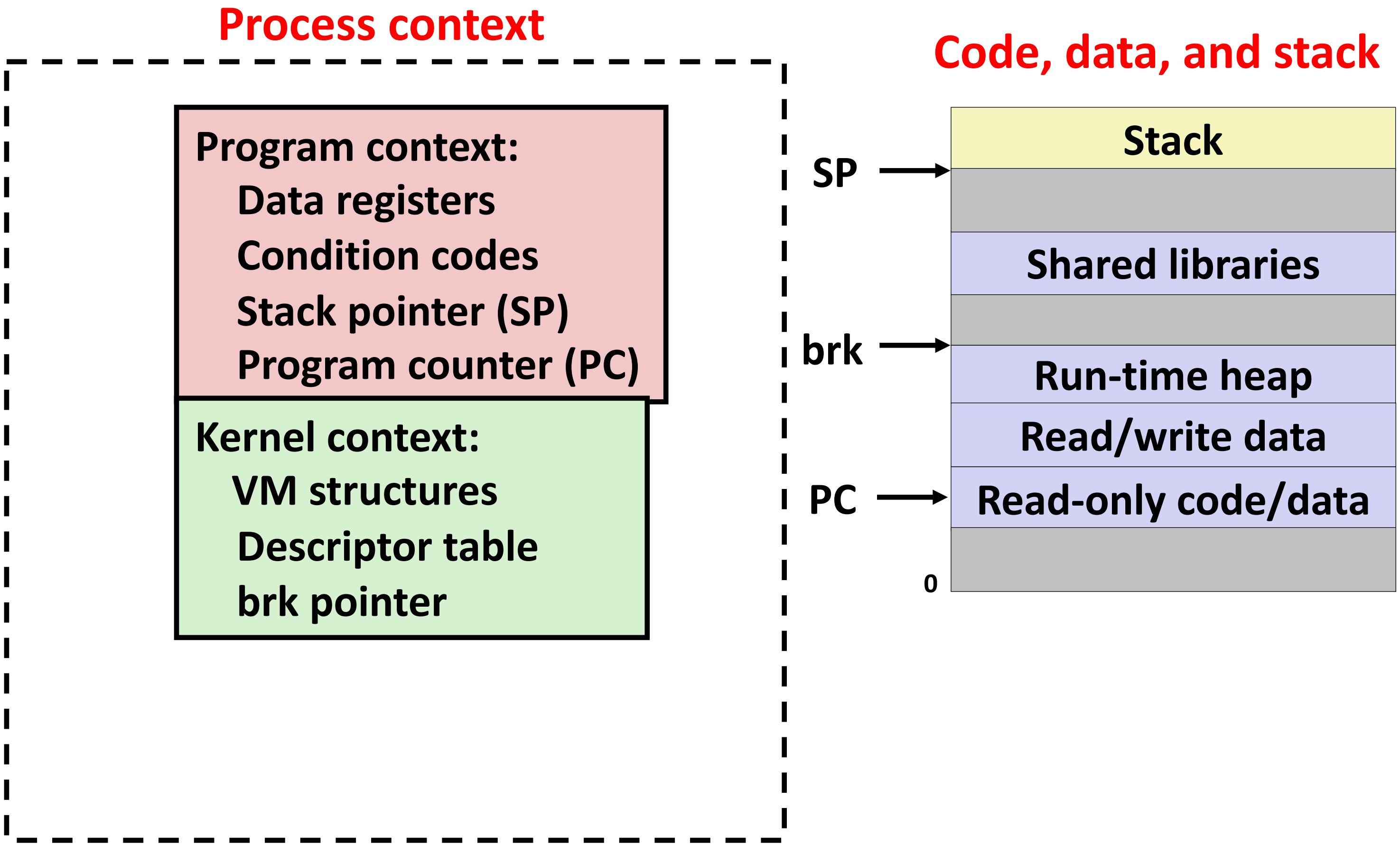
Instructor: Abi Kim

Today

- **Threads**
- Sharing
- Mutual exclusion

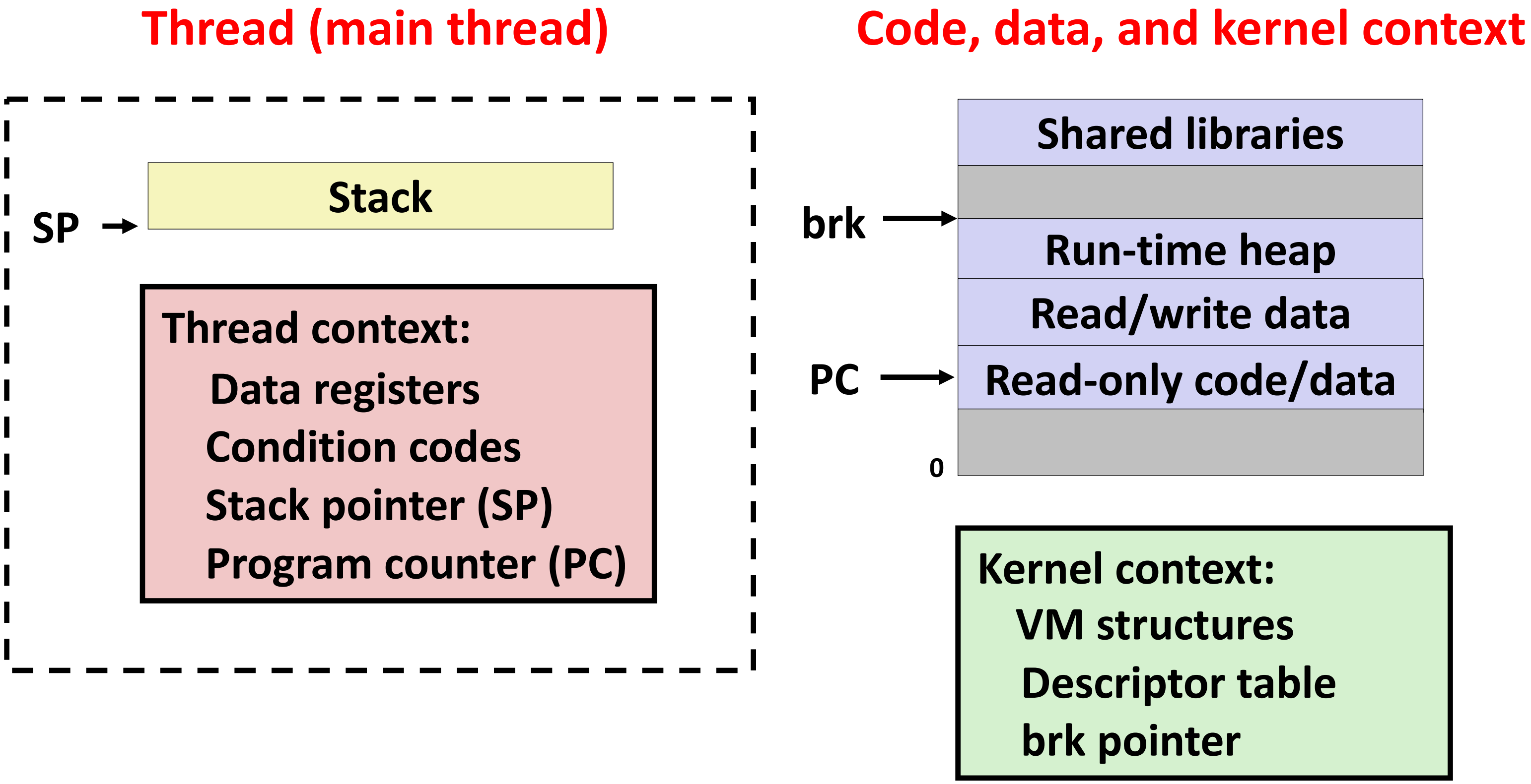
Traditional View of a Process

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



Alternate View of a Process

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

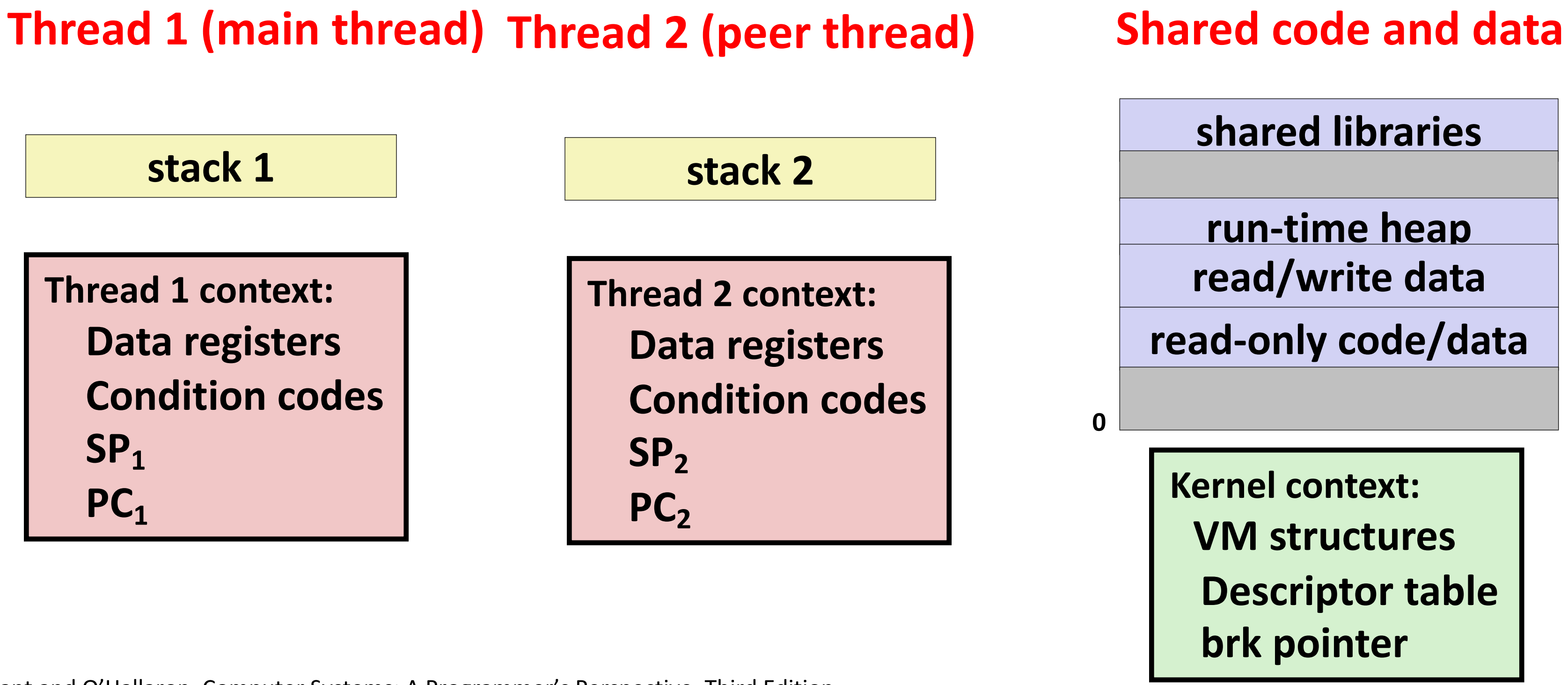


Thread context:
Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

Kernel context:
VM structures
Descriptor table
brk pointer

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)



Thread 1 (main thread)

stack 1

Thread 1 context:
Data registers
Condition codes
SP$_1$
PC$_1$

Thread 2 (peer thread)

stack 2

Thread 2 context:
Data registers
Condition codes
SP$_2$
PC$_2$

Shared code and data

shared libraries

run-time heap

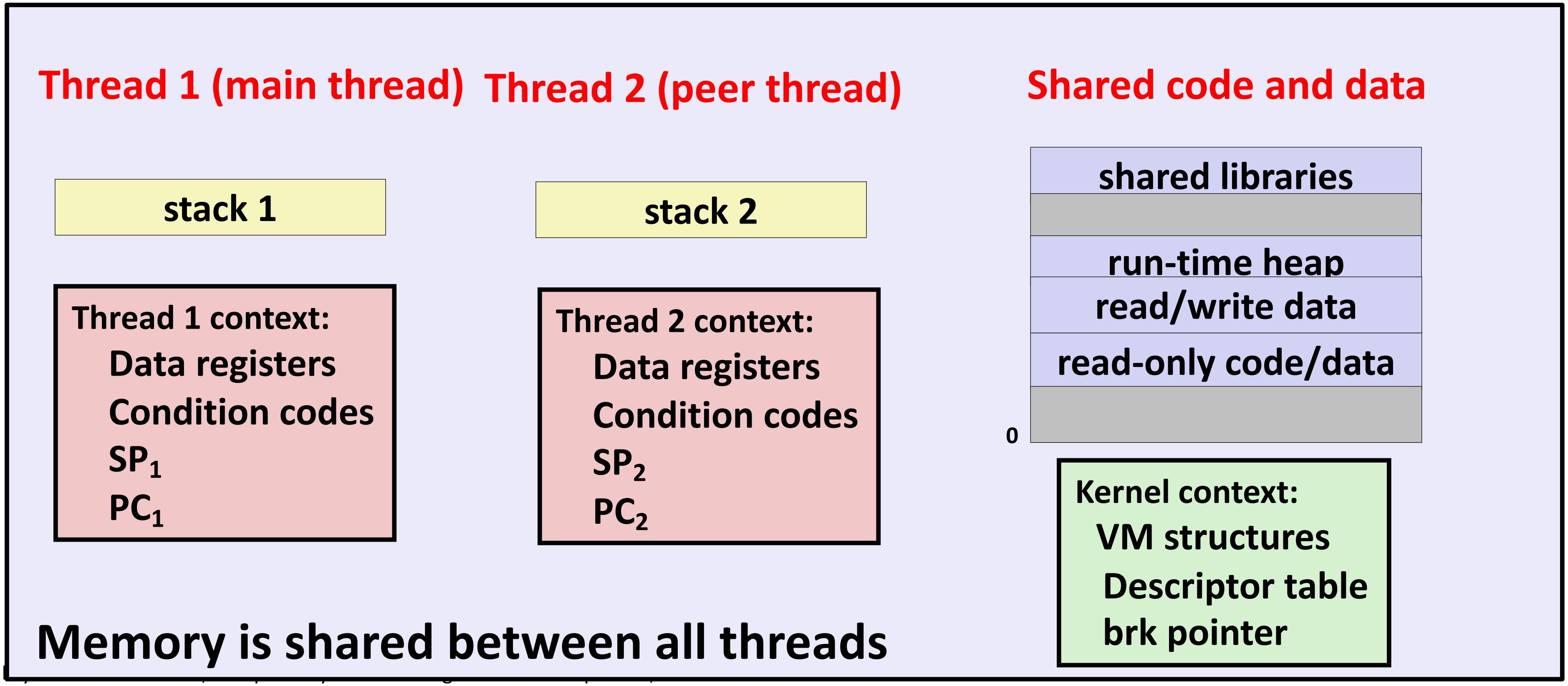
read/write data

read-only code/data

0

Kernel context:
VM structures
Descriptor table
brk pointer

Don’t let the picture confuse you!



Thread 1 (main thread)

stack 1

Thread 1 context:
- Data registers
- Condition codes
- SP$_1$
- PC$_1$

Thread 2 (peer thread)

stack 2

Thread 2 context:
- Data registers
- Condition codes
- SP$_2$
- PC$_2$

Shared code and data

shared libraries

run-time heap

read/write data

read-only code/data

0

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Memory is shared between all threads

Benefits of Threads

- Threads have lighter overhead
- Easier to share memory in concurrent programs using threads
- Threads are faster due to multi-core CPUs allowing multiple threads to execute at once

Today

- Threads review
- **Sharing**
- Mutual exclusion
- Semaphores

Shared Variables in Threaded C Programs

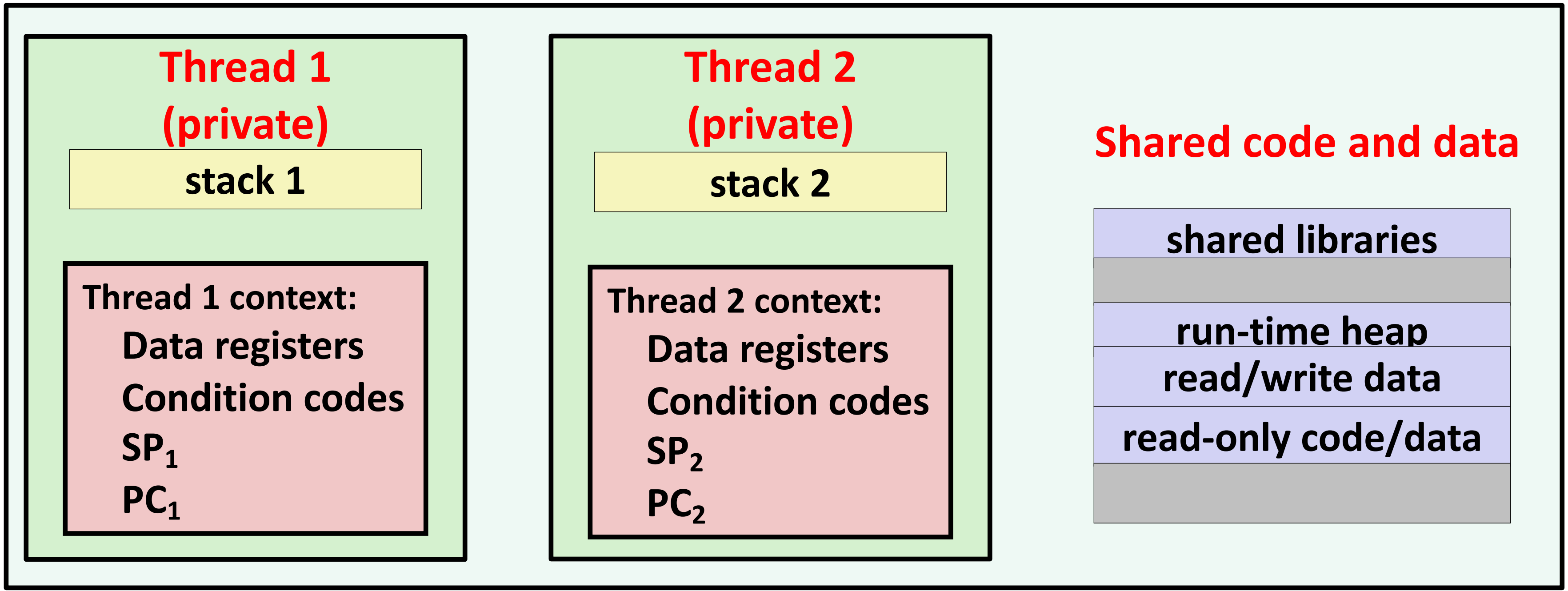
- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- ***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?

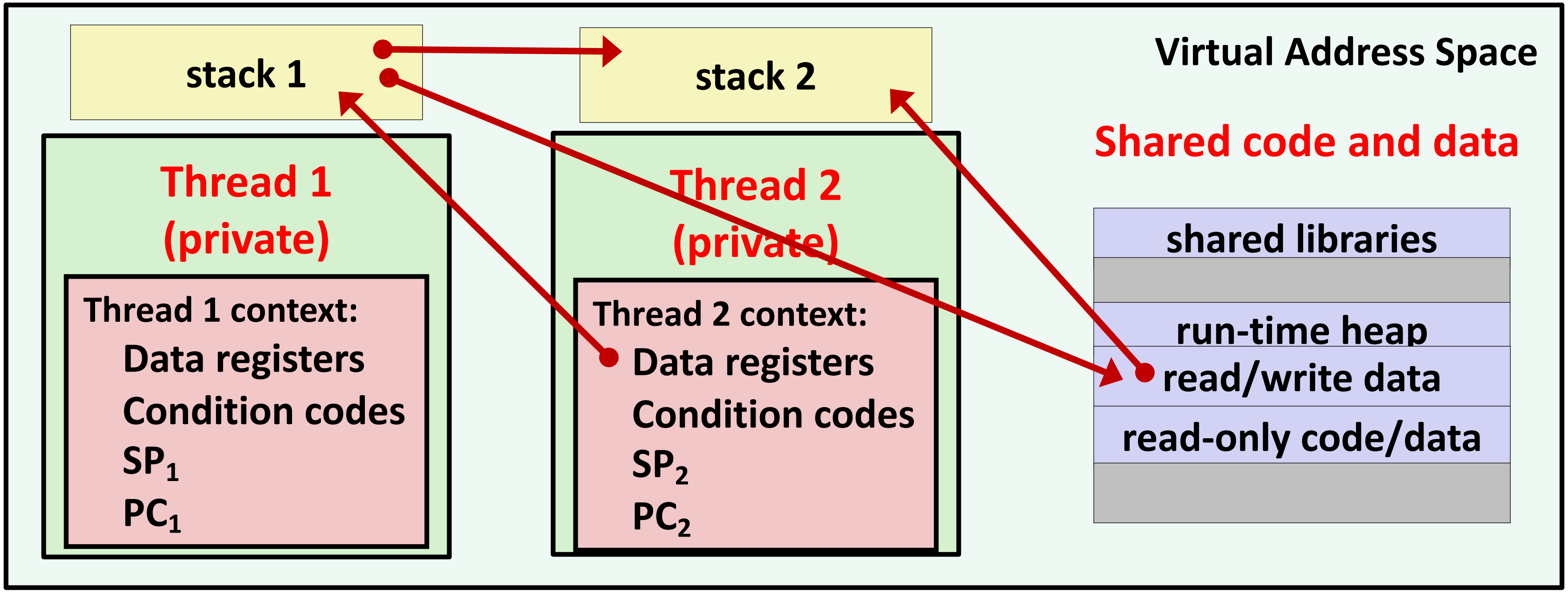
Threads Memory Model: Conceptual

- 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

Passing an argument to a thread - Pedantic

```
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,
                     (void *)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;
}
```

```
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 - Pedantic

```
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,
                     (void *)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;
}
```

- Use malloc to create a per thread heap allocated place in memory for the argument
- Remember to free in thread!
- Producer-consumer pattern

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;
}
```

- Ok to Use cast since sizeof(long) <= sizeof(void*)
- Cast does NOT change bits

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

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

- `&i` points to same location for all threads!
- Creates a data race!

Three Ways to Pass Thread Arg

- **Malloc/free**
  - Producer malloc’s space, passes pointer to pthread_create
  - Consumer dereferences pointer
- **Ptr to stack slot**
  - Producer passes address to producer’s stack in pthread_create
  - Consumer dereferences pointer
- **Cast of int**
  - Producer casts an int/long to address in pthread_create
  - Consumer casts void* argument back to int/long

Example Program to Illustrate Sharing

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

int main(int argc, 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;
}
```

Peer threads reference main thread's stack indirectly through global ptr variable

A common way to pass a single argument to a thread routine

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**
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**

- **Local variables**
  - *Def:* Variable declared inside function without **`static`** attribute
  - **Each thread stack contains one instance of each local variable**

- **Local static variables**
  - *Def:* Variable declared inside function with the **`static`** attribute
  - **Virtual memory contains exactly one instance of any local static variable.**

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 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 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 introduce the possibility of nasty *synchronization* errors.**

`badcnt.c`: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

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

    return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

`cnt` should equal 20,000.

What went wrong?

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:* **In general, any sequentially consistent* interleaving is possible, but 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

OK

**For now. In reality, on x86 even non-sequentially consistent interleavings are possible*

Concurrent Execution

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but 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)

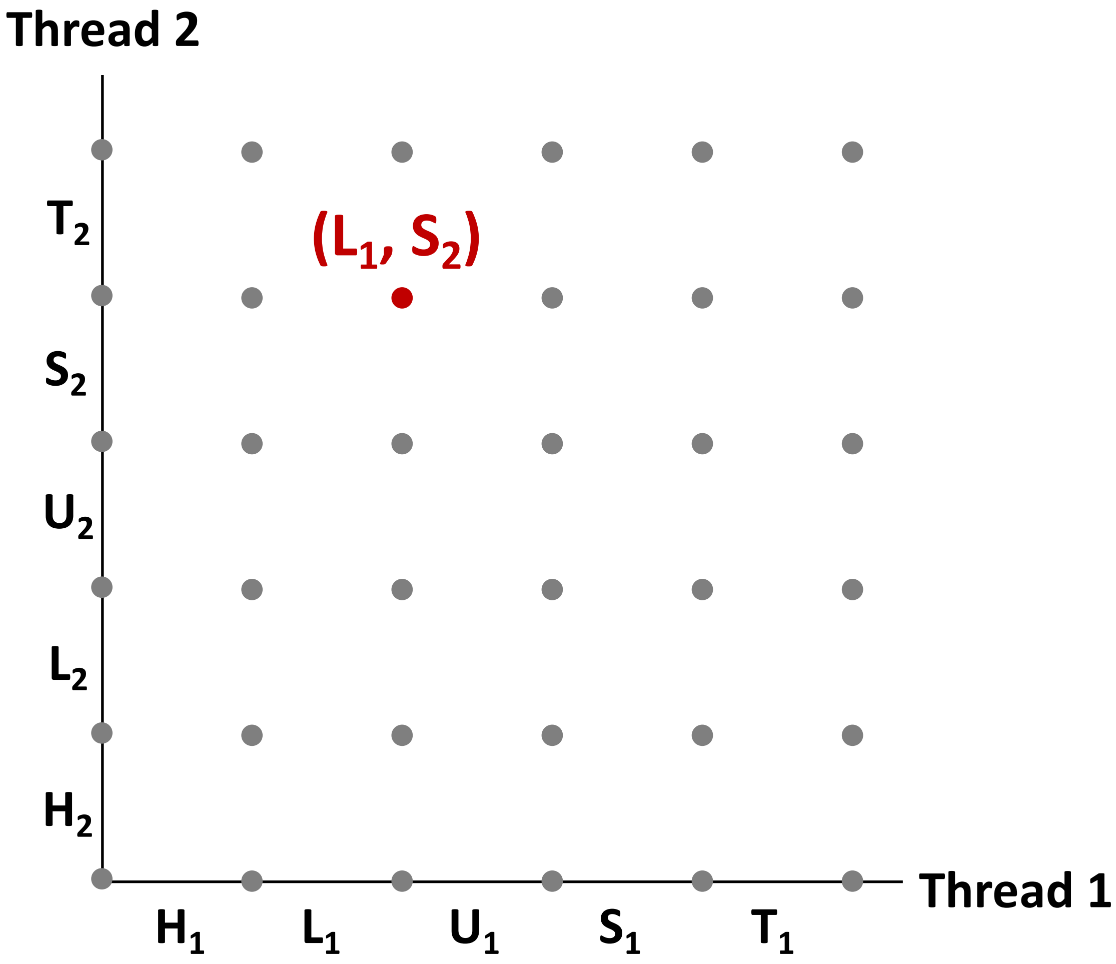
- **How about this ordering?**

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



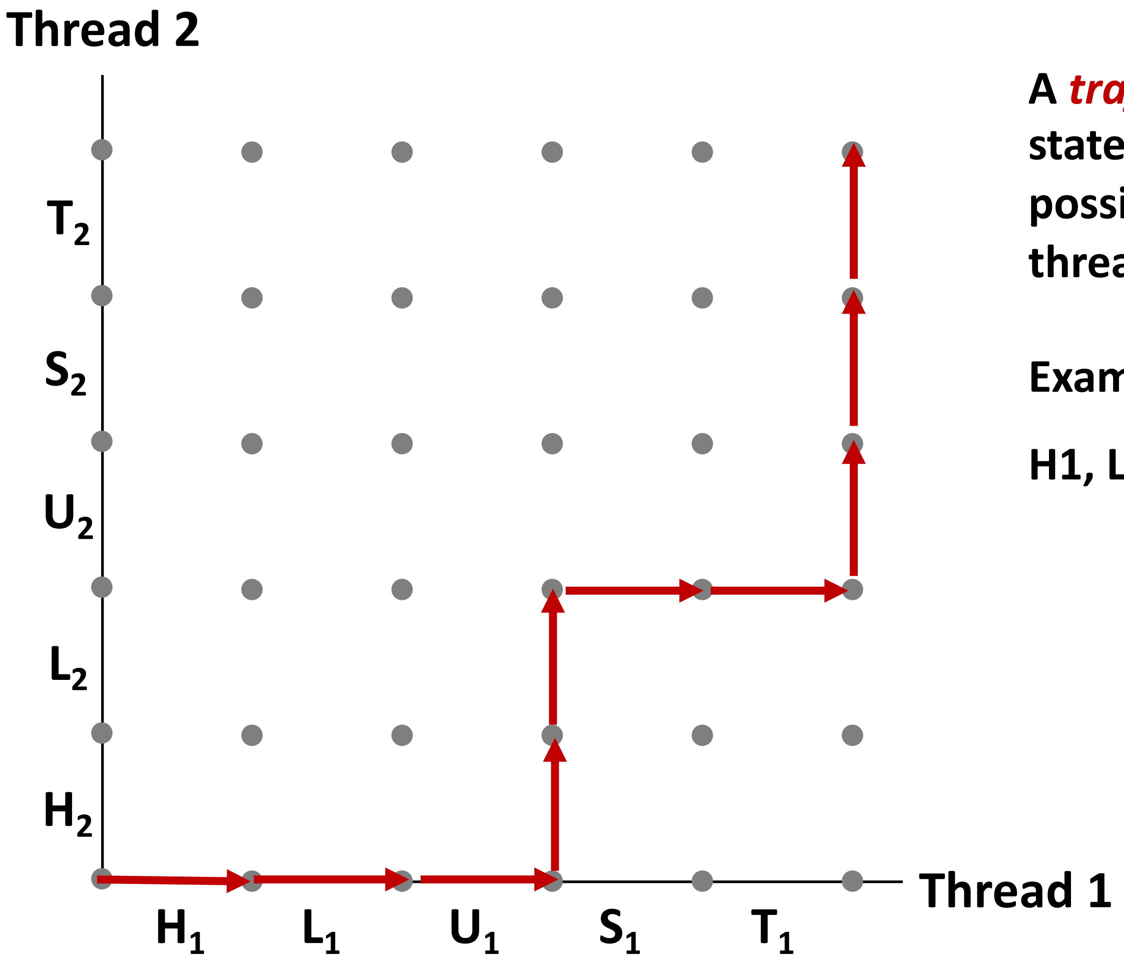
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

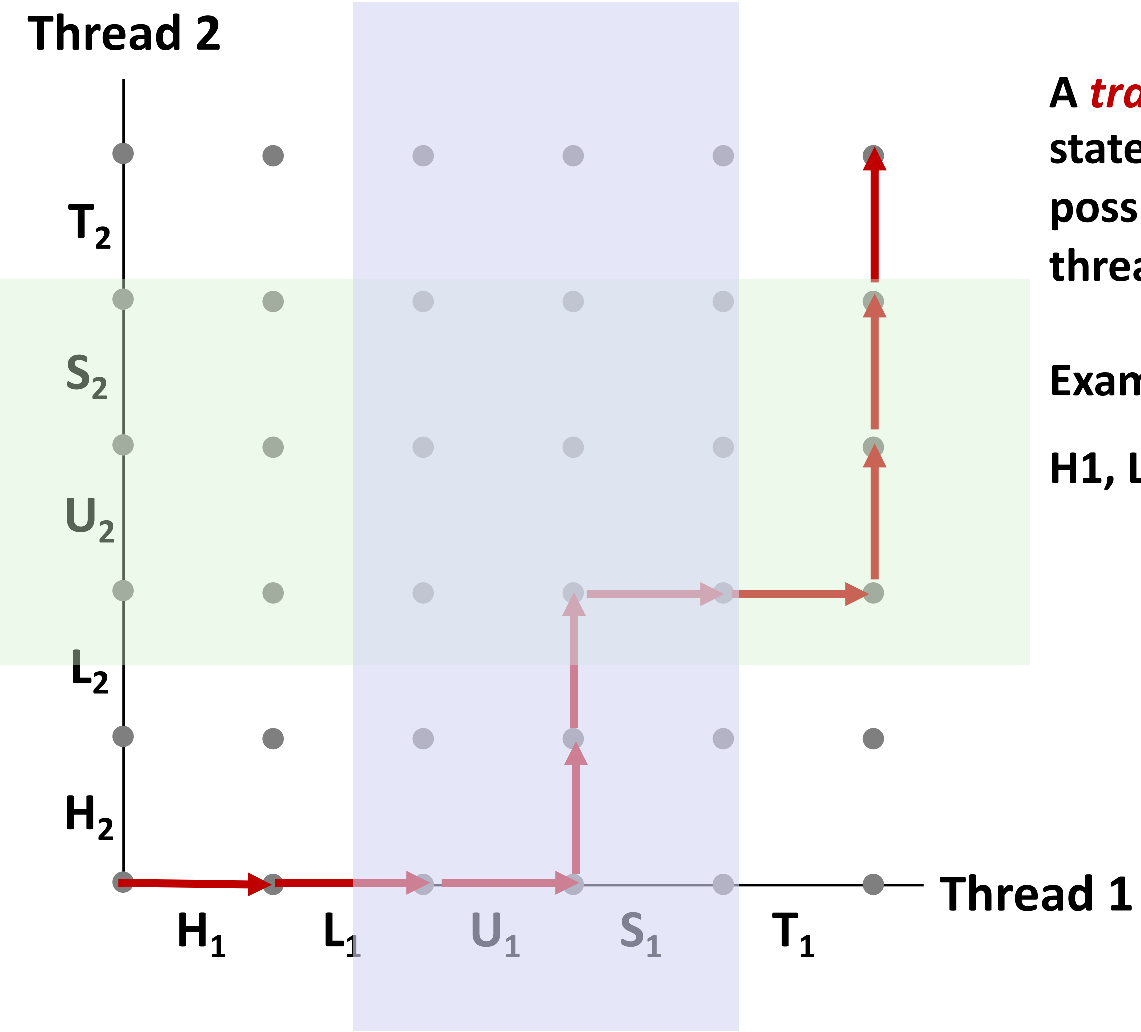


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

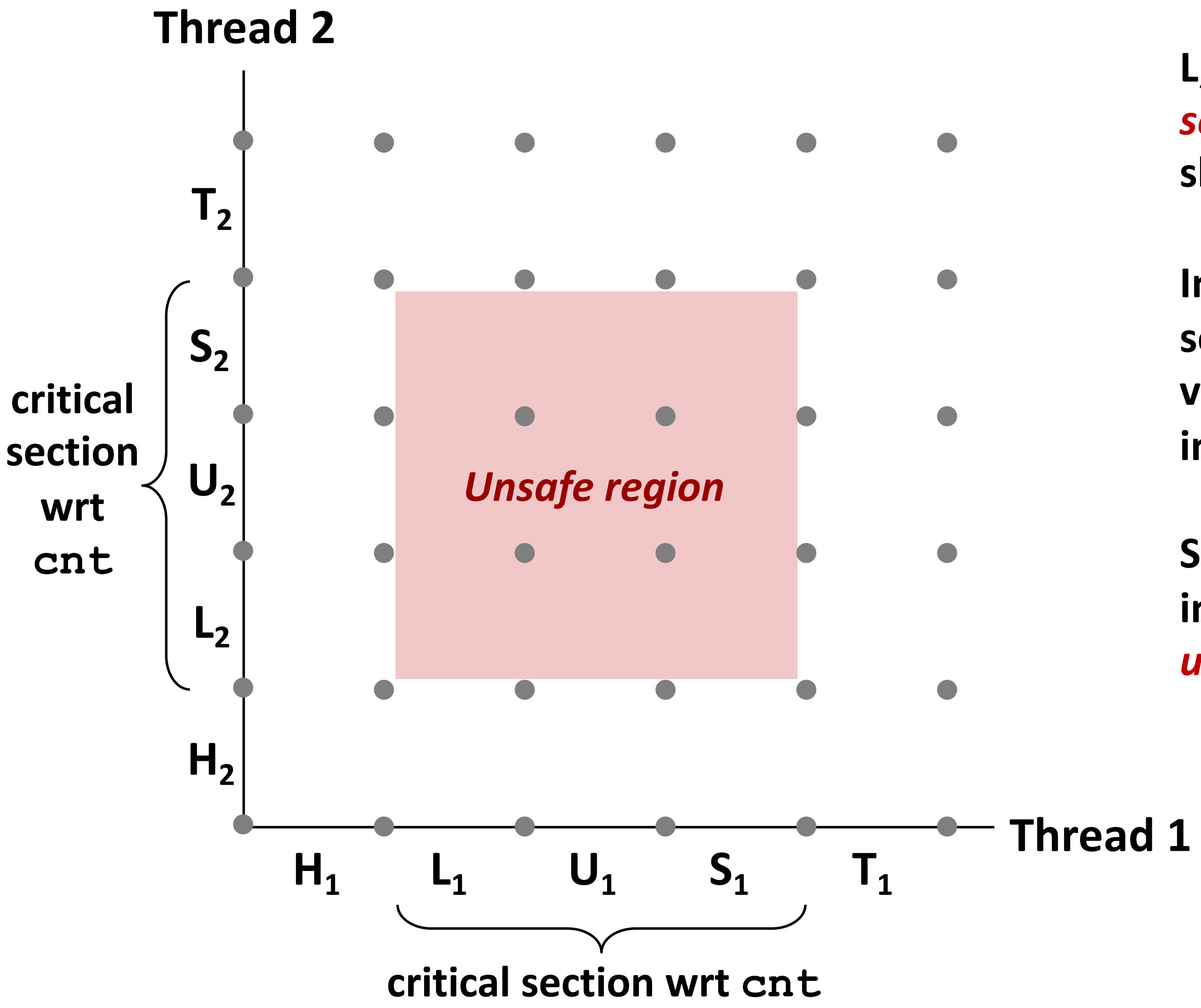


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Critical Sections and Unsafe Regions

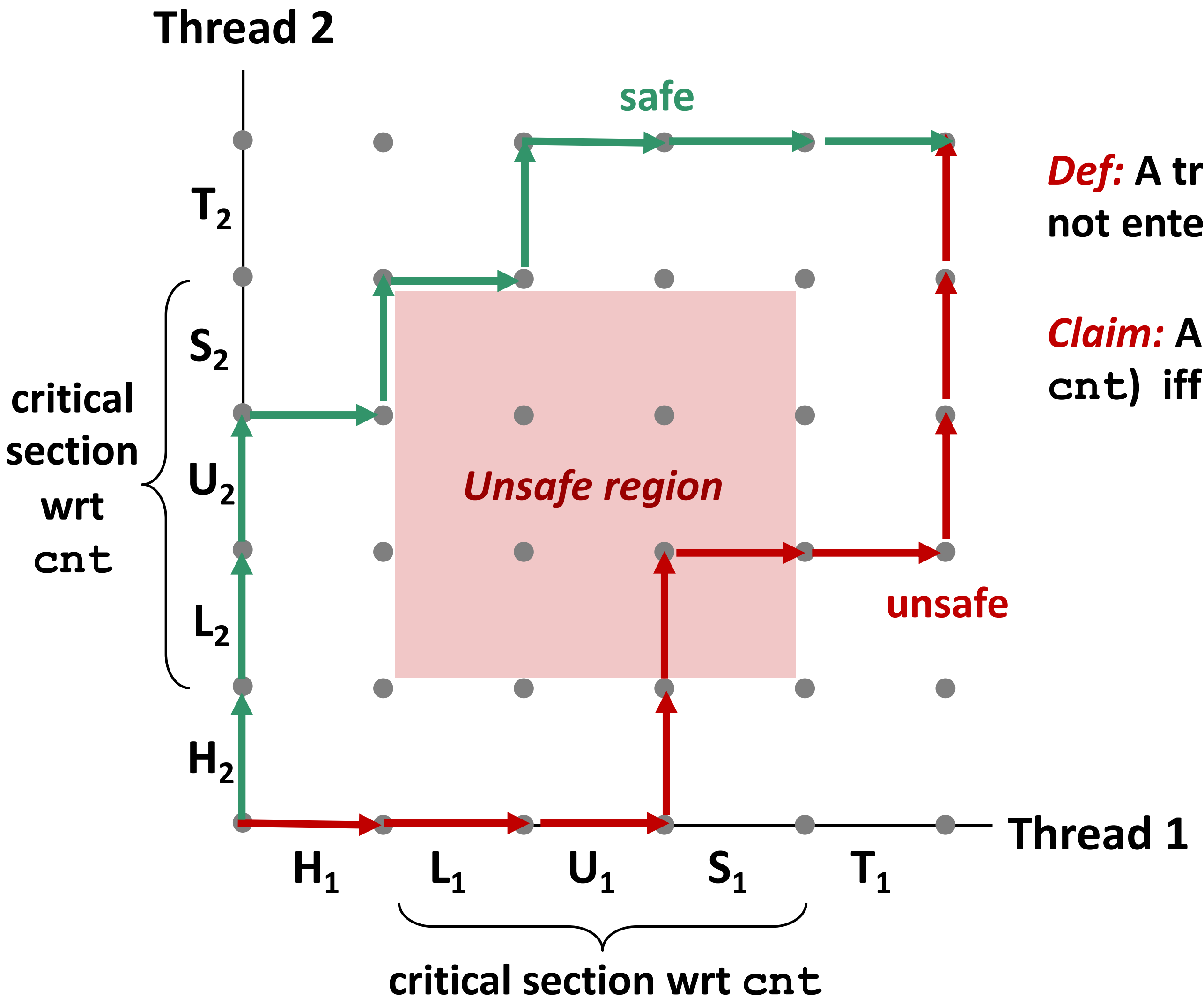


L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

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**

`badcnt.c`: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

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

    return NULL;
```

Variable	main	thread1	thread2
cnt			
niters.m			
tid1.m			
i.1			
i.2			
niters.1			
niters.2			

`badcnt.c`: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
               *((long *)vargp);

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

    return NULL;
```

Variable	main	thread1	thread2
cnt	yes*	yes	yes
niters.m	yes	no	no
tid1.m	yes	no	no
i.1	no	yes	no
i.2	no	no	yes
niters.1	no	yes	no
niters.2	no	no	yes

Today

- Threads review
- Sharing
- **Mutual exclusion**

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.**
  - i.e., need to guarantee ***mutually exclusive access*** for each critical section.

- **Classic solution:**
  - Mutex (pthreads)
  - Semaphores (Edsger Dijkstra)

- **Other approaches (out of our scope)**
  - Condition variables (pthreads)
  - Monitors (Java)

Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s)**
  - If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - Test and decrement operations occur atomically (indivisibly)
  - If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - Increment *s* by 1.
    - Increment operation occurs atomically
  - If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing *s*.

- **Semaphore invariant: *(s >= 0)***

Semaphores

- *Semaphore:* **non-negative global integer synchronization variable**

- **Manipulated by *P* and *V* operations:**
  - *P(s):* [`while (s == 0) wait(); s--;`]
    - Dutch for "Proberen" (test)
  - *V(s):* [`s++;`]
    - Dutch for "Verhogen" (increment)

- **OS kernel guarantees that operations between brackets [ ] are executed indivisibly**
    - Only one *P* or *V* operation at a time can modify s.
    - When `while` loop in *P* terminates, only that *P* can decrement `s`

- **Semaphore invariant: *(s >= 0)***

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

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

CS:APP wrapper functions:

```
#include "csapp.h”

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

`badcnt.c`: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
               *((long *)vargp);

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

    return NULL;
}
```

How can we fix this using synchronization?

MUTual EXclusion (mutex)

- ***Mutex*: boolean synchronization variable**
- **enum {locked = 0, unlocked = 1}**
- **lock(m)**
  - If the mutex is currently not locked, lock it and return
  - Otherwise, wait (spinning, yielding, etc) and retry
- **unlock(m)**
  - Update the mutex state to unlocked

Using Semaphores for Mutual Exclusion

Basic idea:

- Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
- Surround corresponding critical sections with *P(mutex)* and *V(mutex)* operations.

Terminology:

- *Binary semaphore*: semaphore whose value is always 0 or 1
- *Mutex:* binary semaphore used for mutual exclusion
  - P operation: "locking" the mutex
  - V operation: "unlocking" or "releasing" the mutex
  - *"Holding"* a mutex: locked and not yet unlocked.
- *Counting semaphore*: used as a counter for set of available resources.

`goodcnt.c`: Proper Synchronization

Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

sem_init(&mutex, 0, 1); /* mutex = 1 */
```

Surround critical section with *P* and *V*:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Warning: It's orders of magnitude slower than `badcnt.c`.

`goodcnt.c`: Proper Synchronization

Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

sem_init(&mutex, 0, 1); /* mutex = 1 */
```

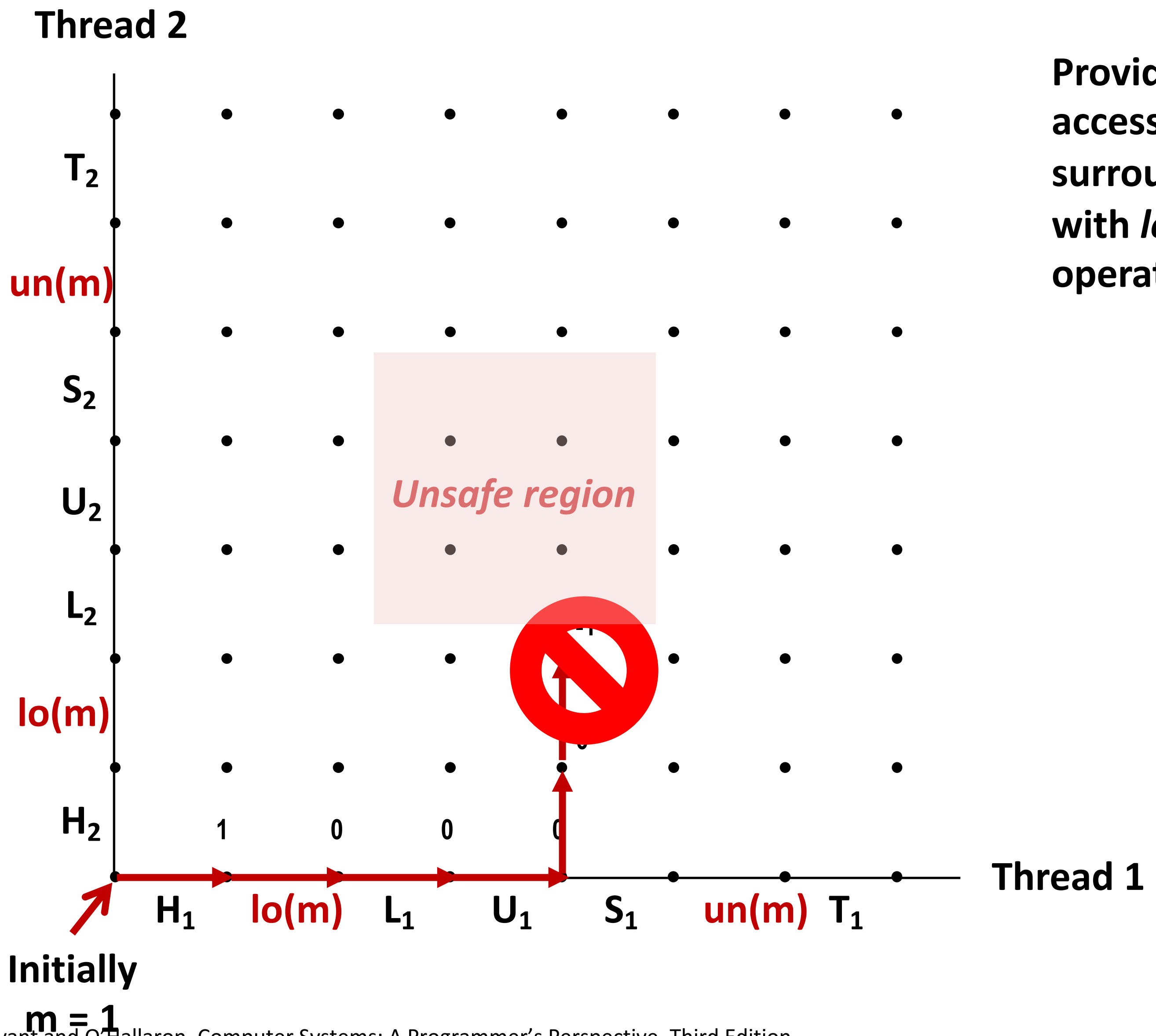
Surround critical section with *P* and *V*:

	OK cnt=2000000	BOOM! cnt=1036525	Slowdown
real	0m0.138s	0m0.007s	20X
user	0m0.120s	0m0.008s	15X
sys	0m0.108s	0m0.000s	NaN

And slower means much slower!

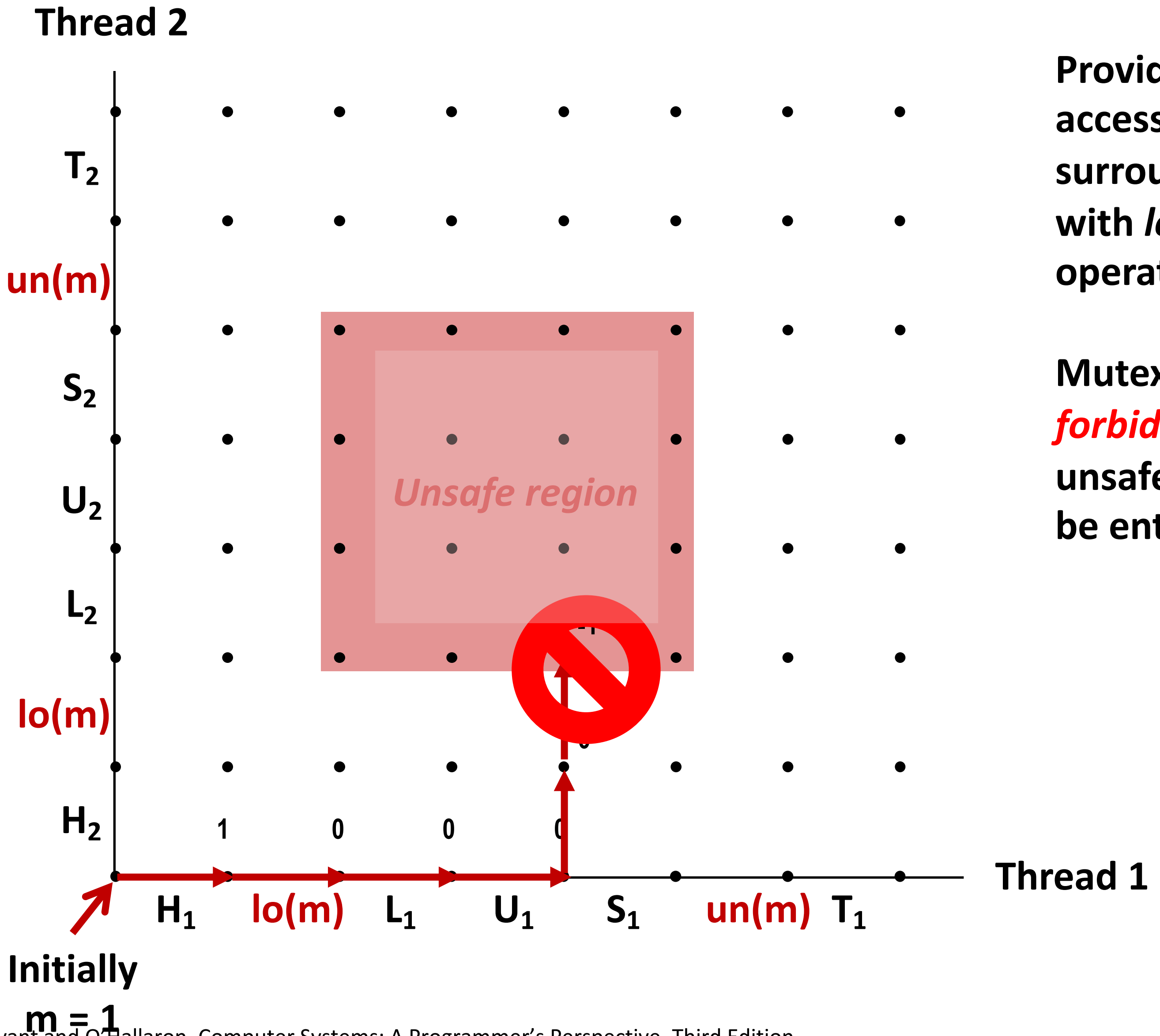
ver

Why Mutexes Work



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

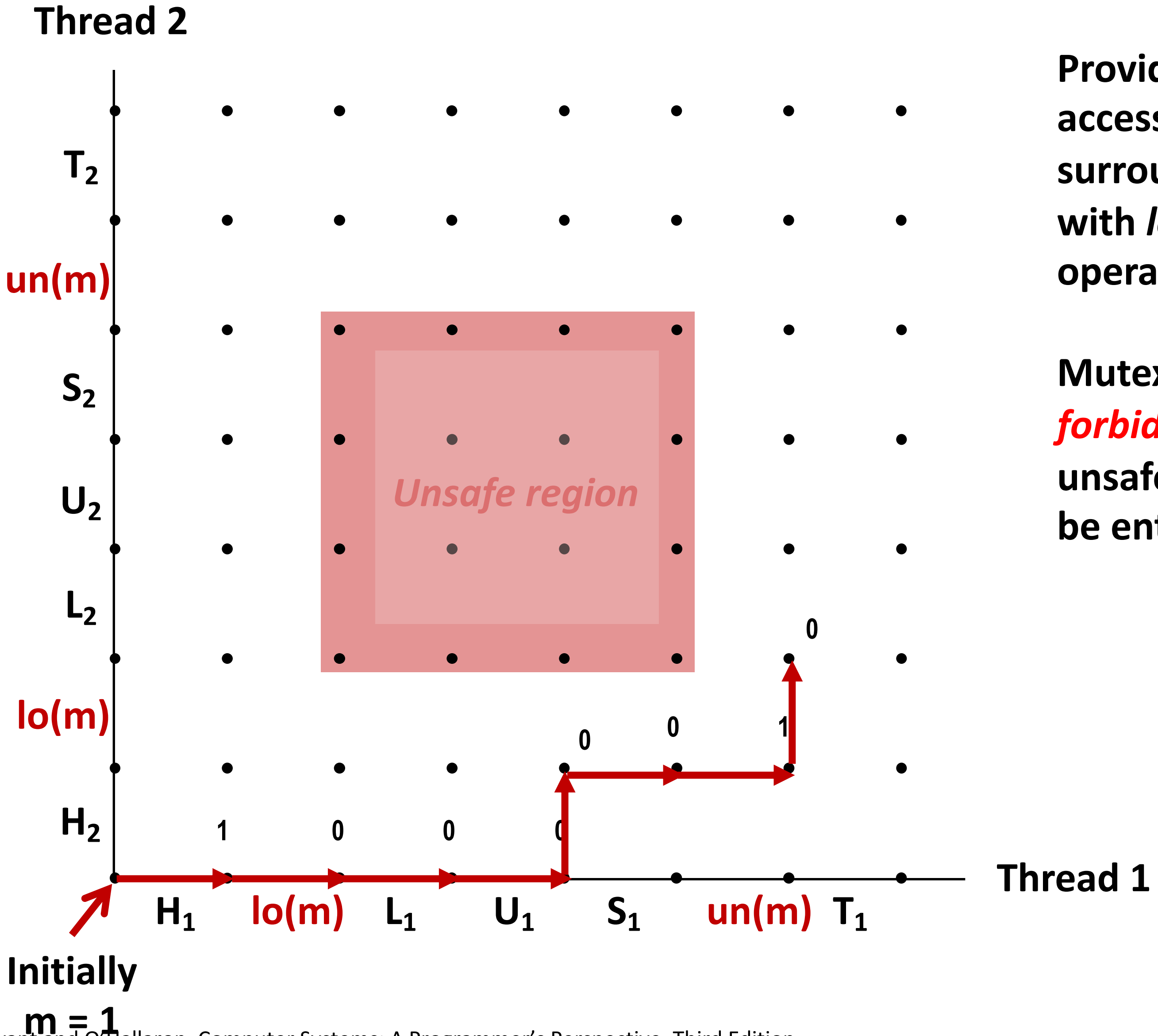
Why Mutexes Work



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.

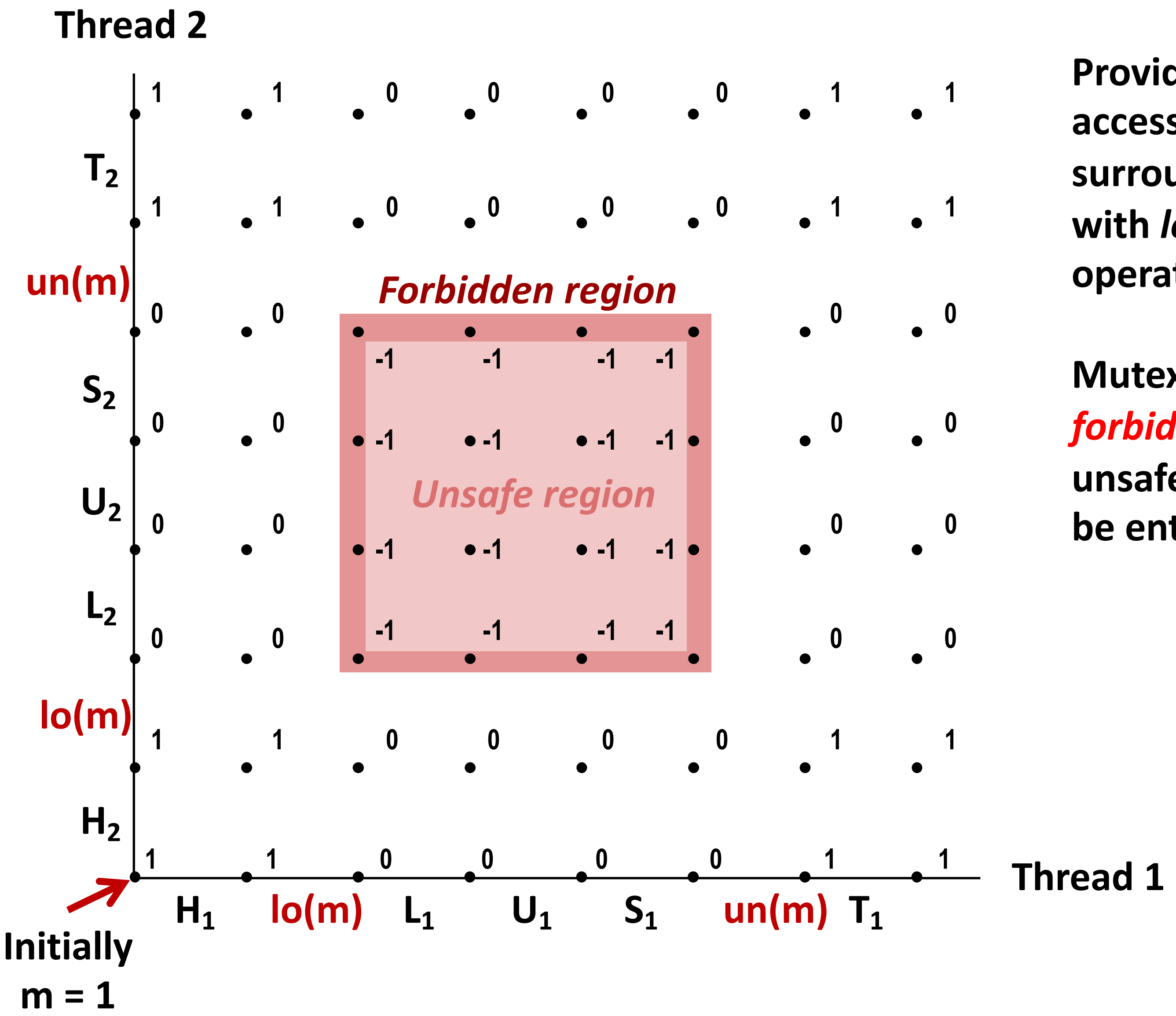
Why Mutexes Work



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.

Why Mutexes Work



1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
0	0	-1	-1	-1	-1	0	0
0	0	-1	-1	-1	-1	0	0
0	0	-1	-1	-1	-1	0	0
0	0	-1	-1	-1	-1	0	0
1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1

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.

Summary

- Programmers need a clear model of how variables are shared by threads.
- Variables shared by multiple threads must be protected to ensure mutually exclusive access.
- Semaphores are a fundamental mechanism for enforcing mutual exclusion.

Appendix: Binary Semaphores vs. Mutexes (not test material, just fyi!)

- **Binary semaphore: semaphore initialized with a value of 1**
- **Both binary semaphores and mutexes can be used to guarantee mutual exclusion**
- **Main difference is ownership**
  - Mutexes must be unlocked by the thread who owned them previously
  - Binary semaphores can be signaled/incremented (V) by a thread who did not decrement (P) them
- **As long as you use binary semaphores in the following way in all threads, they can be used as a mutex**

```
P(&sem);
// critical section
V(&sem);
```

They are also implemented differently but that's out of the scope for this class… (covered in 15-410)