

CSC 252: Computer Organization

Spring 2026: Lecture 23

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

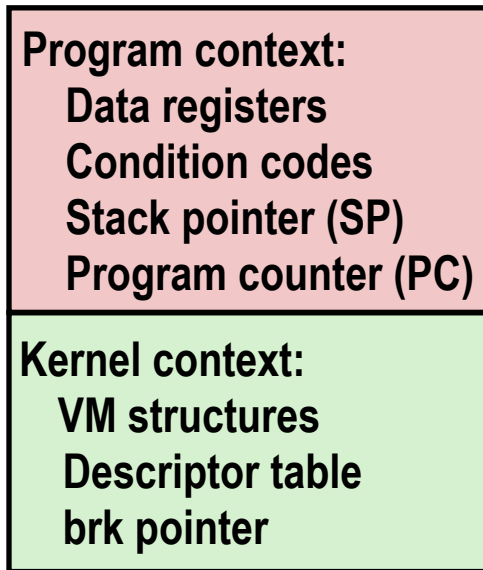
Today

- From process to threads
 - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
 - Single core
 - Multi-core
 - Hyper-threading
 - Cache coherence

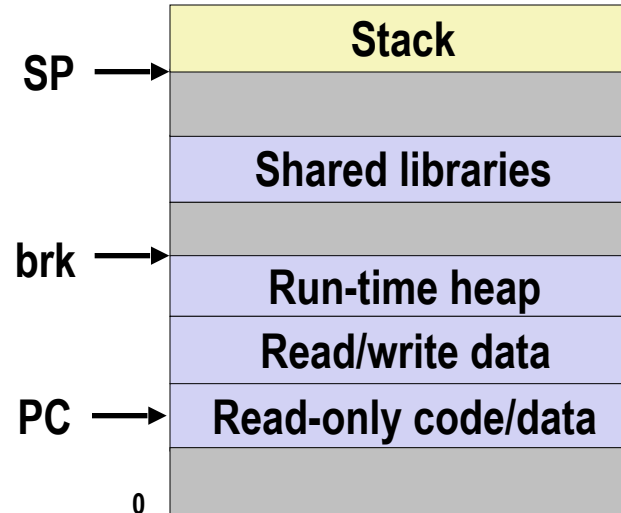
Programmers View of A Process

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

Process context



Code, data, and stack



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)

Thread 2 (peer thread)

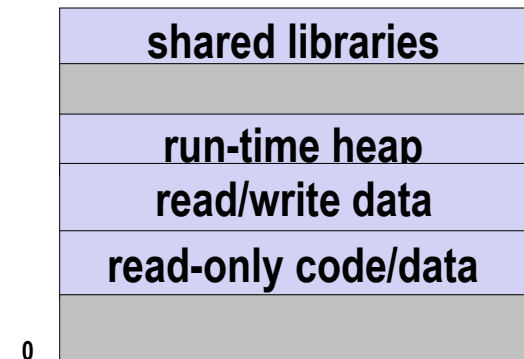
Shared code and data

stack 1

stack 2

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Thread 2 context:
Data registers
Condition codes
SP2
PC2

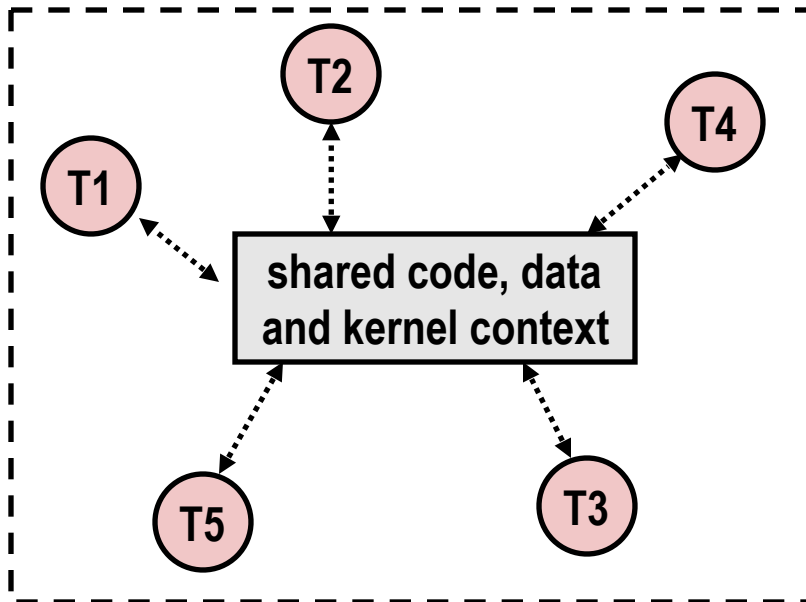


Kernel context:
VM structures
Descriptor table
brk pointer

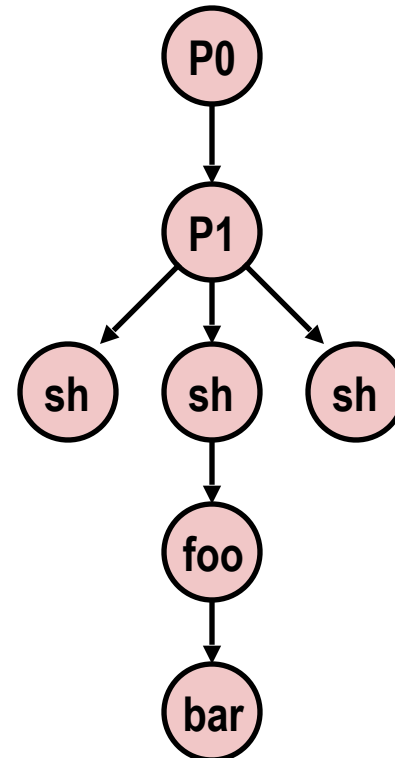
Logical View of Threads

- Threads associated with process form a pool of peers
 - Unlike processes which form a tree hierarchy

Threads associated with process foo

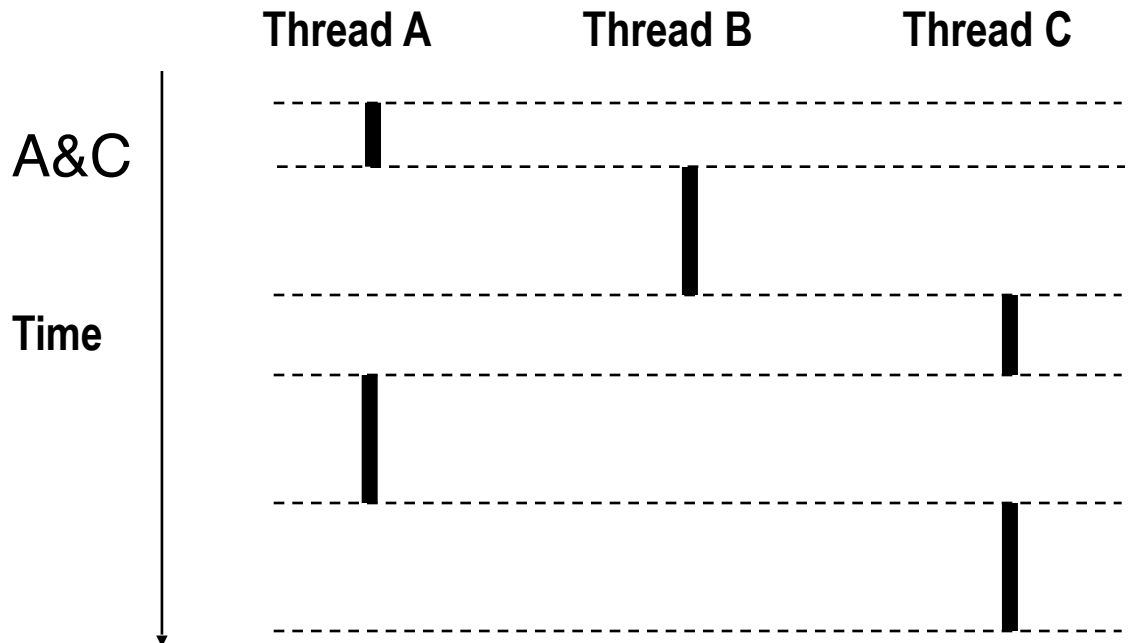


Process hierarchy



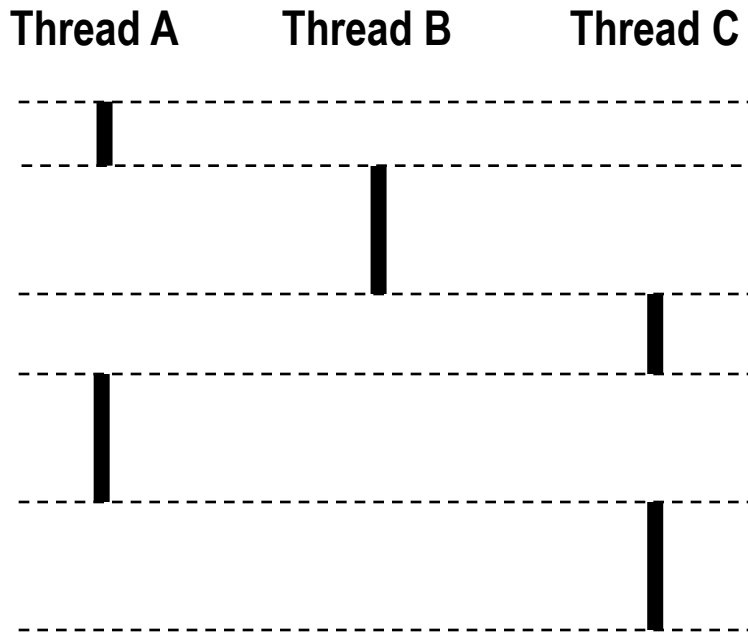
Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential
- Examples:
 - Concurrent: A & B, A&C
 - Sequential: B & C

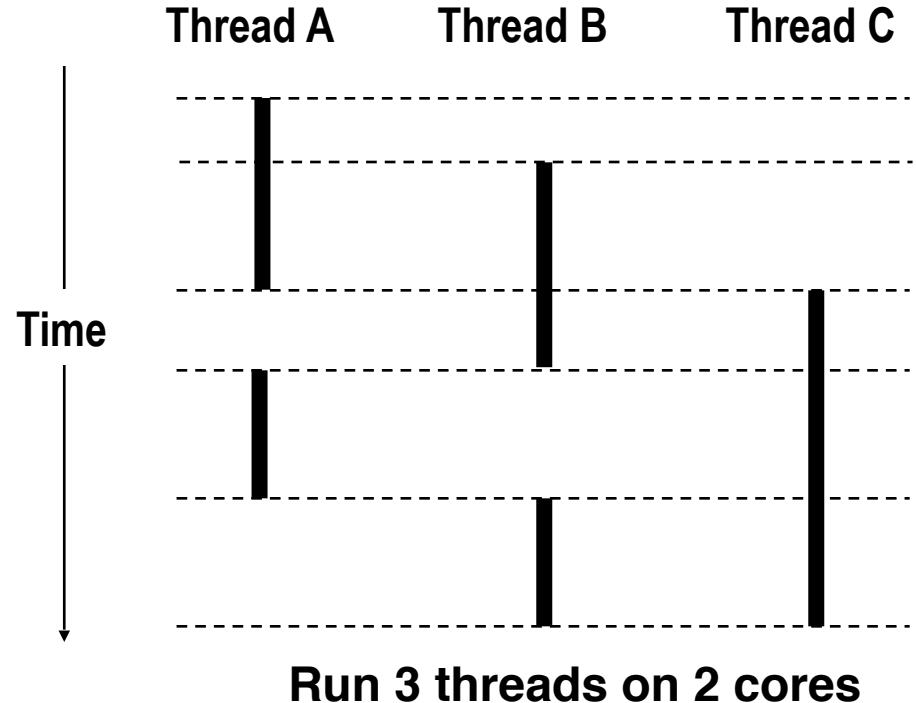


Concurrent Thread Execution

- Single Core Processor
 - Simulate parallelism by time slicing



- Multi Core Processor
 - Threads can have true parallelisms



Threads vs. Processes

- How threads and processes are similar
 - Each has its own logical control flow
 - Each can run concurrently with others (possibly on different cores)
 - Each is context switched, controlled by kernel
- How threads and processes are different
 - Threads share all code and data (except local stacks)
 - Processes (typically) do not
 - Threads are less expensive than processes
 - Space: threads share the same virtual address space except stacks, but processes have their own virtual address space
 - Process control (creating and reaping) twice as expensive
 - Typical Linux numbers:
 - ~20K cycles to create and reap a process
 - ~10K cycles (or less) to create and reap a thread

Today

- From process to threads
 - Basic thread execution model
- **Multi-threading programming**
- Hardware support of threads
 - Single core
 - Multi-core
 - Cache coherence

Posix Threads (Pthreads) Interface

- *Pthreads*: Standard interface for ~60 functions that manipulate threads from C programs
 - Creating and reaping threads
 - `pthread_create()`
 - `pthread_join()`
 - Determining your thread ID
 - `pthread_self()`
 - Terminating threads
 - `pthread_cancel()`
 - `pthread_exit()`
 - `exit()` [terminates all threads] , `return()` [terminates current thread]
 - Synchronizing access to shared variables
 - `pthread_mutex_init`
 - `pthread_mutex_[un]lock`

The Pthreads "hello, world" Program

```
/*  
 * hello.c - Pthreads "hello, world" program  
 */  
#include "csapp.h"  
void *thread(void *vargp);
```

```
int main()  
{  
    pthread_t tid;  
    Pthread_create(&tid, NULL, thread, NULL);  
    Pthread_join(tid, NULL);  
    exit(0);  
}
```

Thread ID

Thread attributes
(usually NULL)

Thread routine

Thread arguments
(void *p)

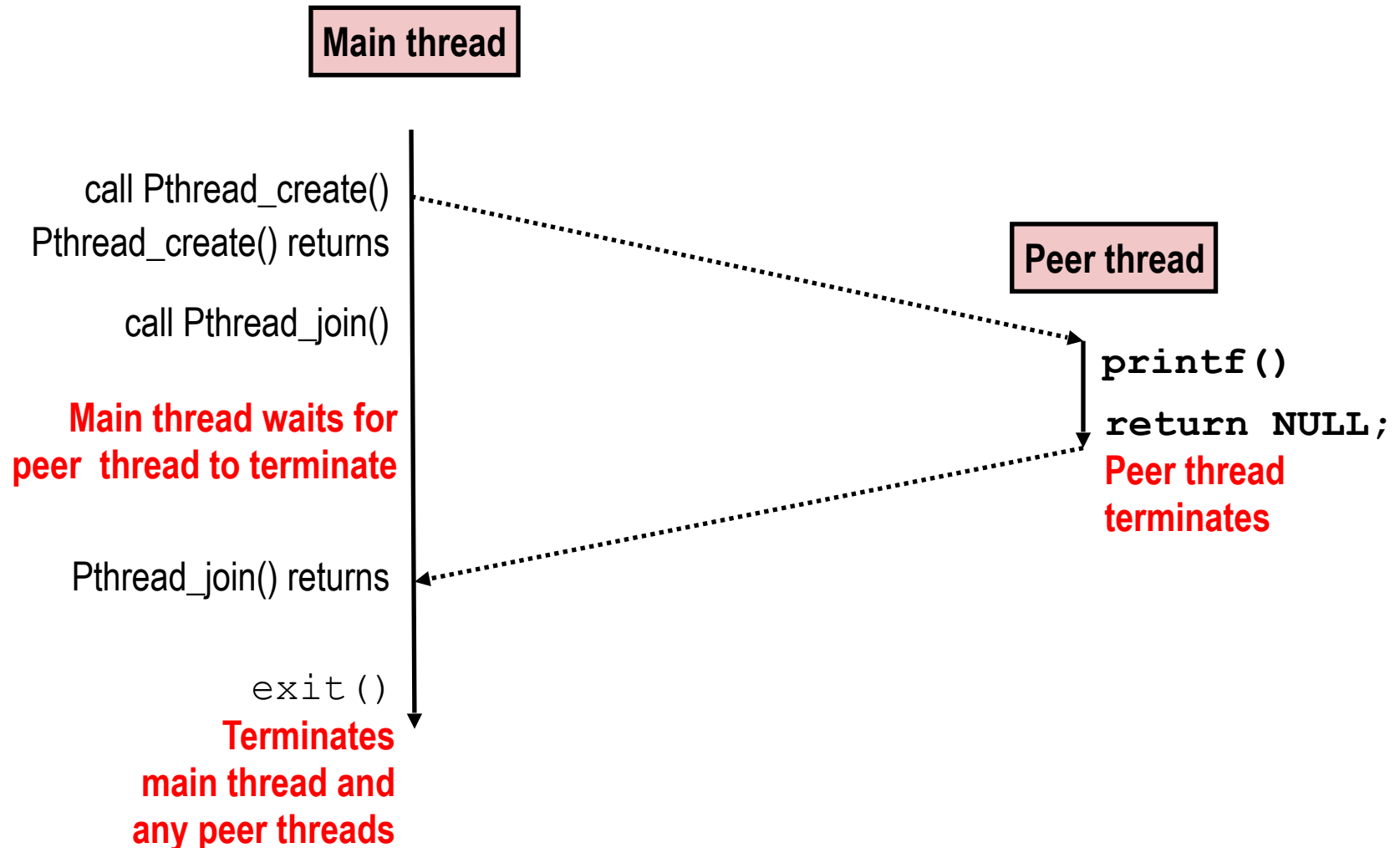
hello.c

```
void* thread(void *vargp) /* thread routine */  
{  
    printf("Hello, world!\n");  
    return NULL;  
}
```

Return value
(void **p)

hello.c

Execution of Threaded “hello, world”



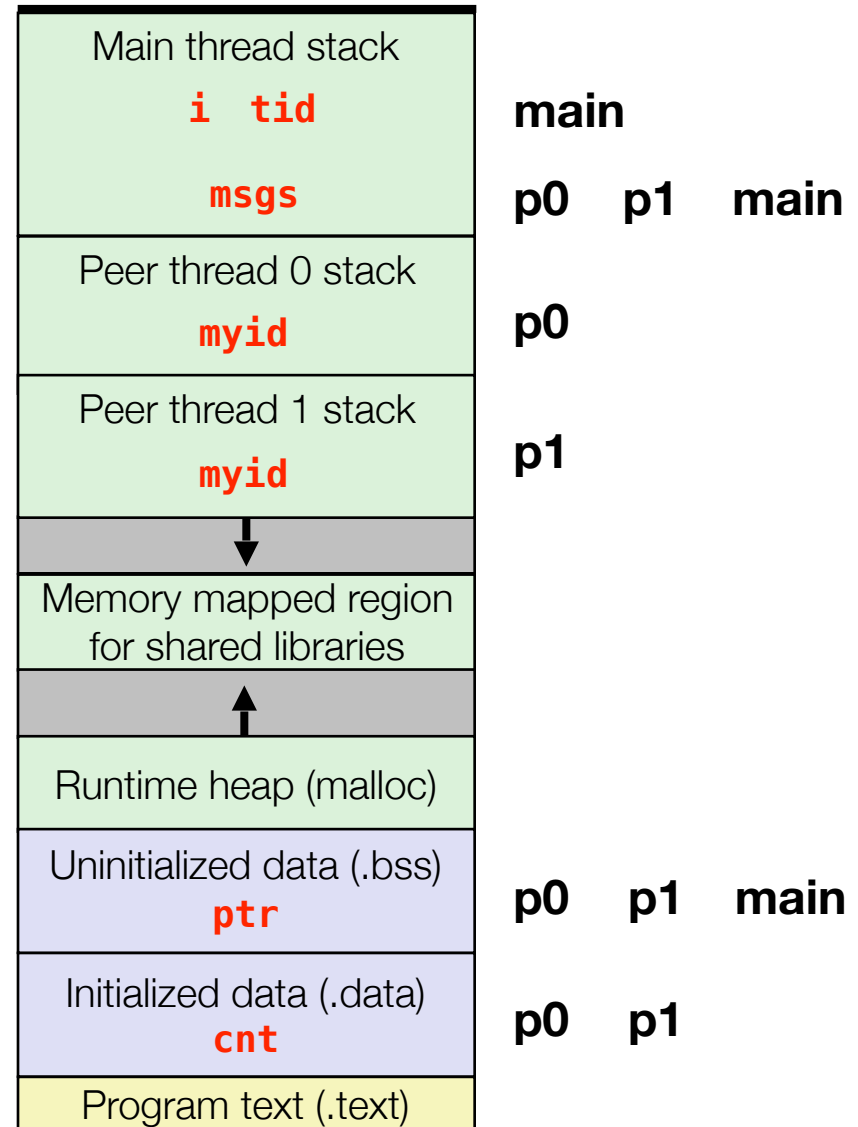
Example Program to Illustrate Sharing

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

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    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



Threads Memory Model

- Conceptual model:
 - 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
- Operationally, this model 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

Synchronizing Threads

- Shared variables are handy...
- ...but introduce the possibility of nasty *synchronization* errors.

Improper Synchronization

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

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;
    long niters = 10000;

    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 * 10000))
        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;
}
```

```
linux> ./badcnt
OK cnt=20000
```

```
linux> ./badcnt
BOOM! cnt=13051
```

cnt should be 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

<pre>movq (%rdi), %rcx testq %rcx, %rcx jle .L2 movl \$0, %eax</pre>	}	H_i : Head
<pre>.L3: movq cnt(%rip), %rdx addq \$1, %rdx movq %rdx, cnt(%rip)</pre>		
<pre>addq \$1, %rax cmpq %rcx, %rax jne .L3</pre>	}	L_i : Load cnt U_i : Update cnt S_i : Store cnt
<pre>.L2:</pre>		
	}	T_i : Tail

Concurrent Execution

- **Key observation:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

i (thread)	instr _i	%rdx ₁	%rdx ₂	cnt (shared)
1	L ₁	0	-	0
1	U ₁	1	-	0
1	S ₁	1	-	1
2	L ₂	-	1	1
2	U ₂	-	2	1
2	S ₂	-	2	2

Thread 1
critical section

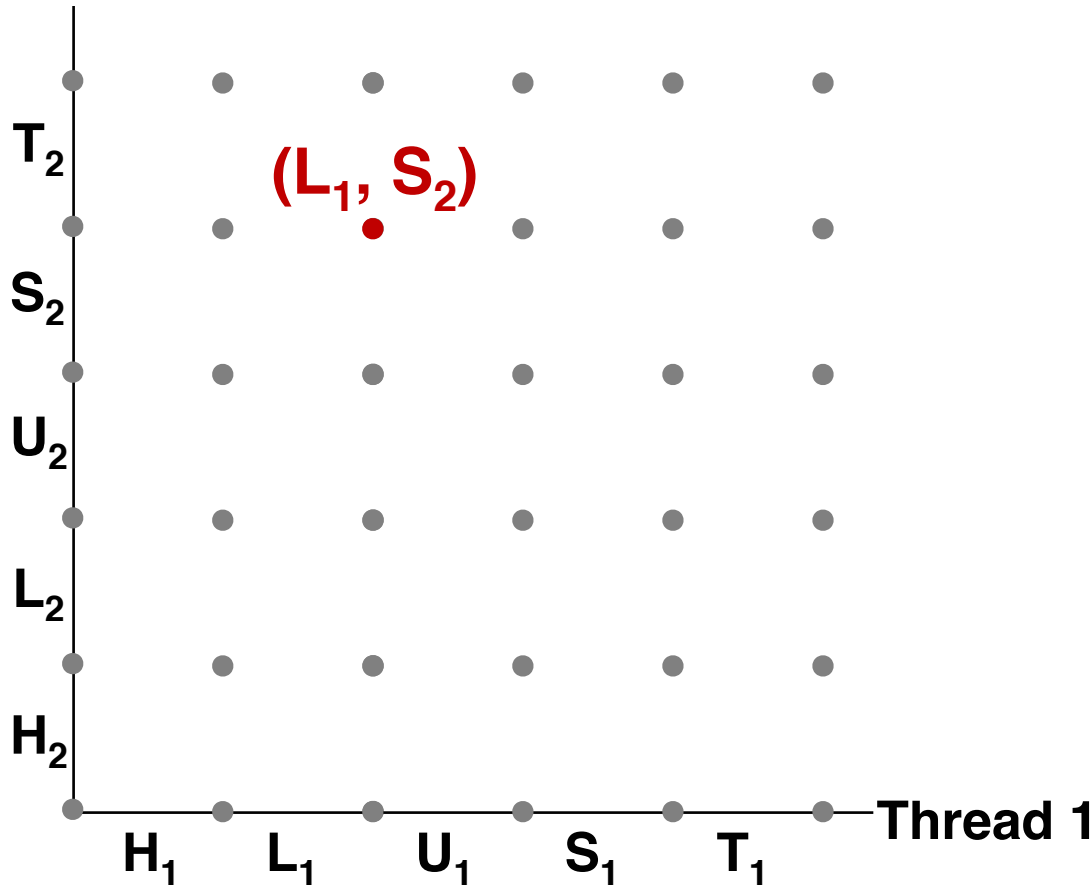
Thread 2
critical section

The legend consists of two entries. The first entry shows a light green rectangle followed by the text 'Thread 1' and 'critical section' on two lines. The second entry shows a light blue rectangle followed by the text 'Thread 2' and 'critical section' on two lines.

L_i	movq	cnt(%rip), %rdx
U_i	addq	\$1, %rdx
S_i	movq	%rdx, cnt(%rip)

Progress Graphs

Thread 2



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

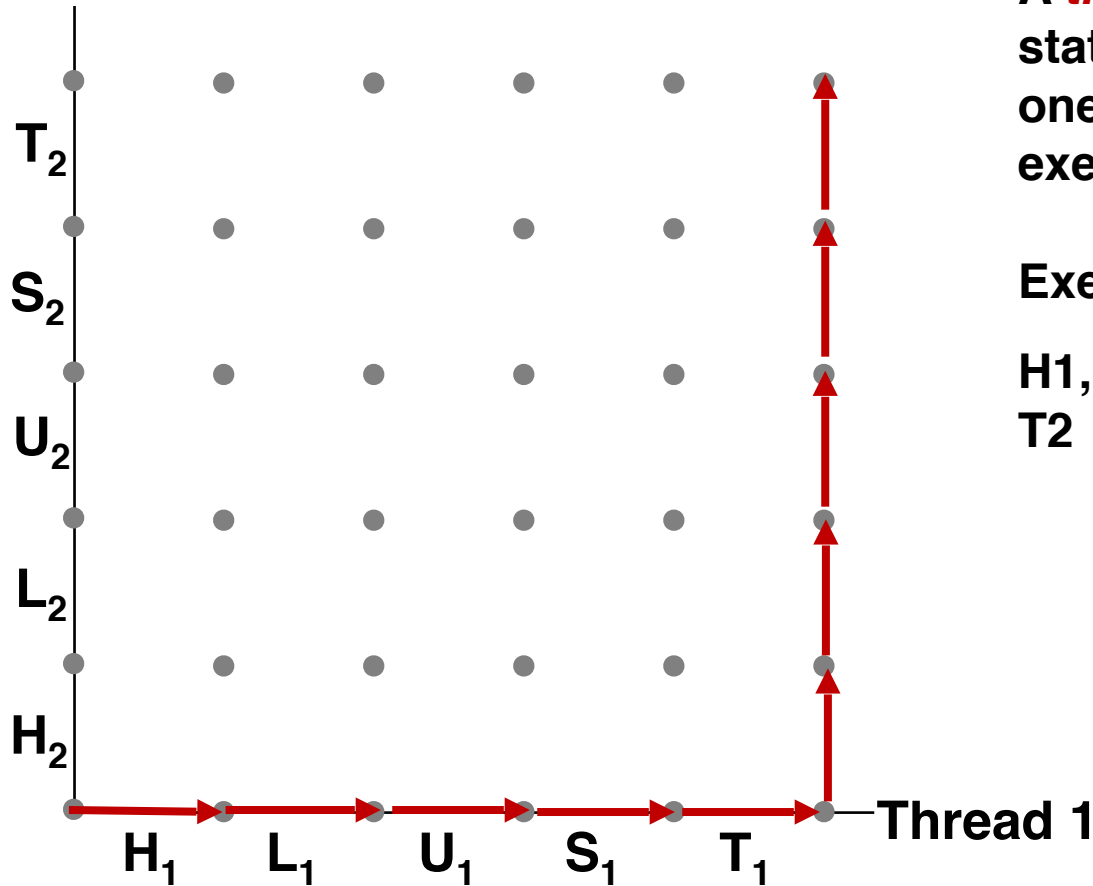
Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible **execution state** (Inst₁, Inst₂).

E.g., (L₁, S₂) denotes state where thread 1 has completed L₁ and thread 2 has completed S₂.

Trajectories in Progress Graphs

Thread 2



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

Execution Ordering:

$H_1, L_1, U_1, S_1, T_1, H_2, L_2, U_2, S_2, T_2$

Concurrent Execution (cont)

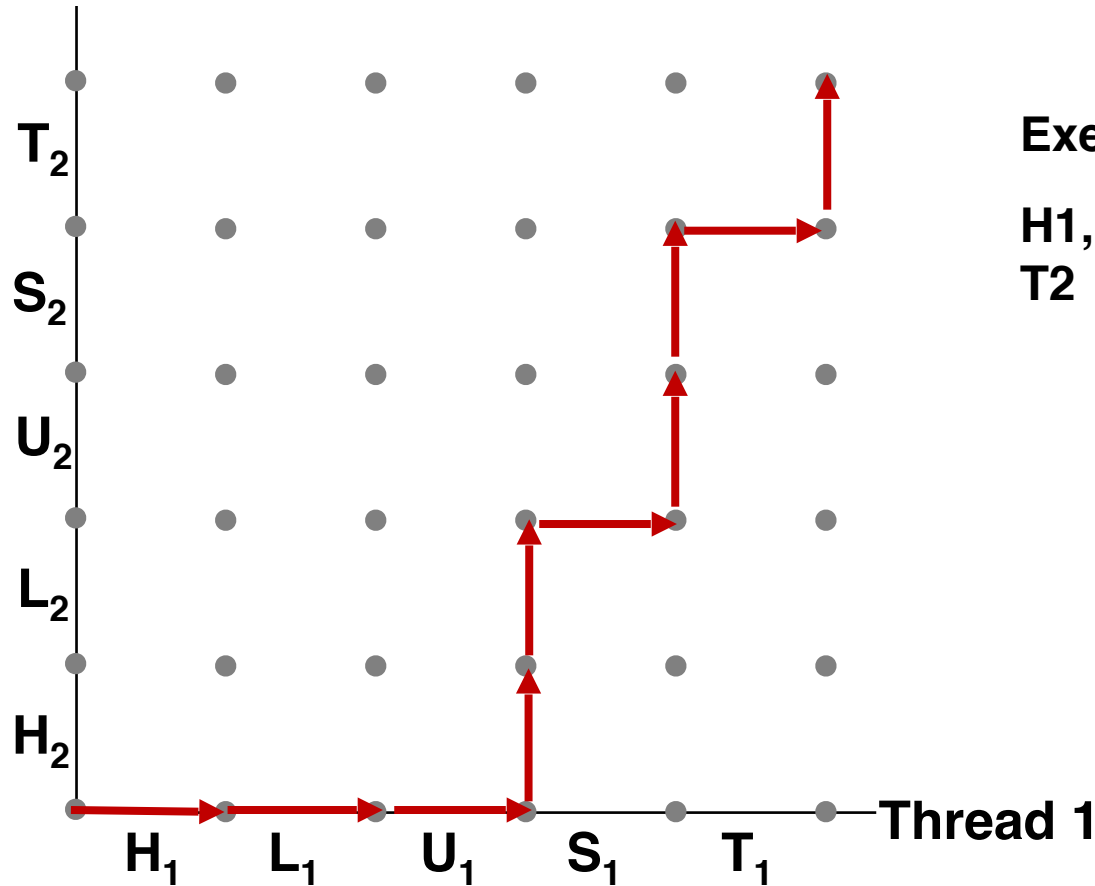
- A legal (feasible) but undesired ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	instr_i	%rdx₁	%rdx₂	cnt (shared)
1	L₁	0	-	0
1	U₁	1	-	0
2	L₂	-	0	0
1	S₁	1	-	1
2	U₂	-	1	1
2	S₂	-	1	1

L_i	movq	cnt(%rip), %rdx
U_i	addq	\$1, %rdx
S_i	movq	%rdx, cnt(%rip)

Trajectories in Progress Graphs

Thread 2



Execution Ordering:

$H_1, L_1, U_1, H_2, L_2, S_1, U_2, S_2, T_1, T_2$

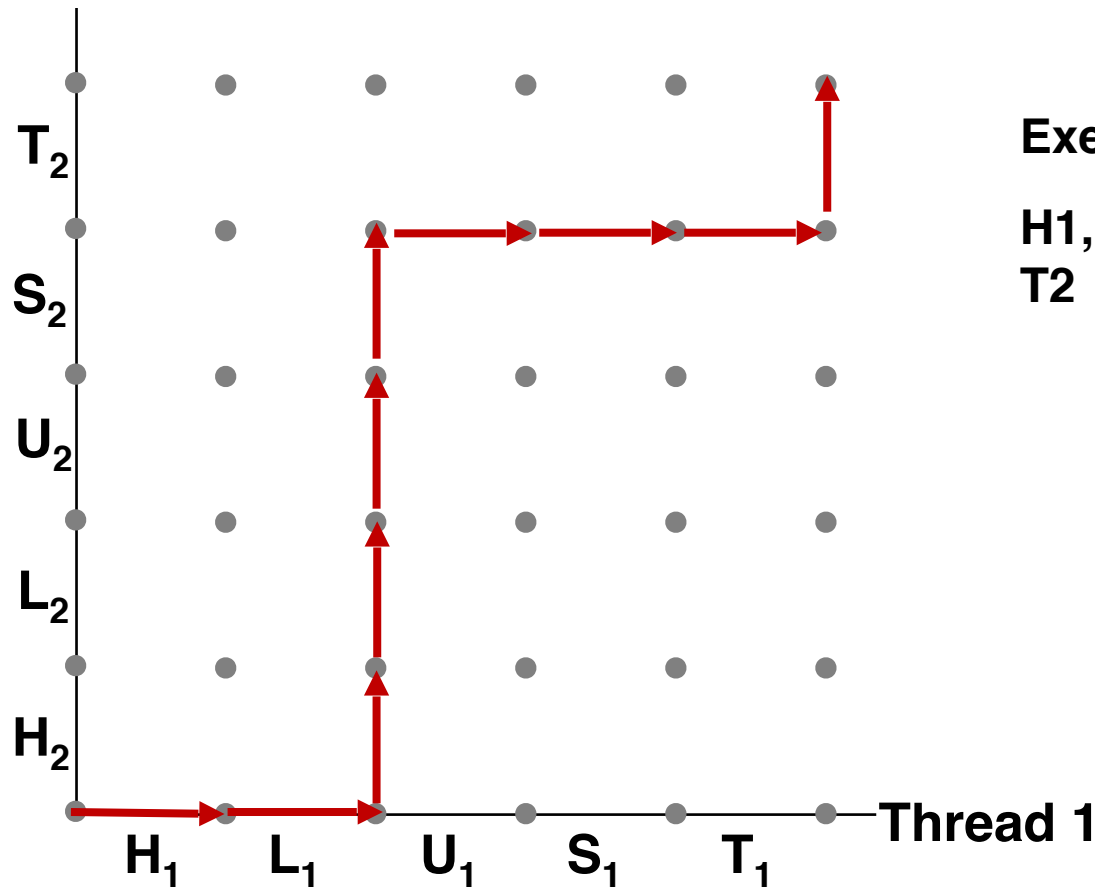
Concurrent Execution (cont)

i (thread)	instr _i	%rdx ₁	%rdx ₂	cnt (shared)
1	L ₁	0		
2	L ₂		0	
2	U ₂		1	
2	S ₂		1	1
1	U ₁	1		
1	S ₁	1		1

L_i	movq	cnt(%rip), %rdx
U_i	addq	\$1, %rdx
S_i	movq	%rdx, cnt(%rip)

Trajectories in Progress Graphs

Thread 2



Execution Ordering:

$H_1, L_1, H_2, L_2, U_2, S_2, U_1, S_1, T_1, T_2$

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

critical
section
wrt cnt



<pre>movq (%rdi), %rcx testq %rcx, %rcx jle .L2 movl \$0, %eax</pre>	H_i : Head
<pre>.L3: movq cnt(%rip), %rdx addq \$1, %rdx movq %rdx, cnt(%rip)</pre>	L_i : Load cnt U_i : Update cnt S_i : Store cnt
<pre>addq \$1, %rax cmpq %rcx, %rax jne .L3 .L2:</pre>	T_i : Tail

Critical Section

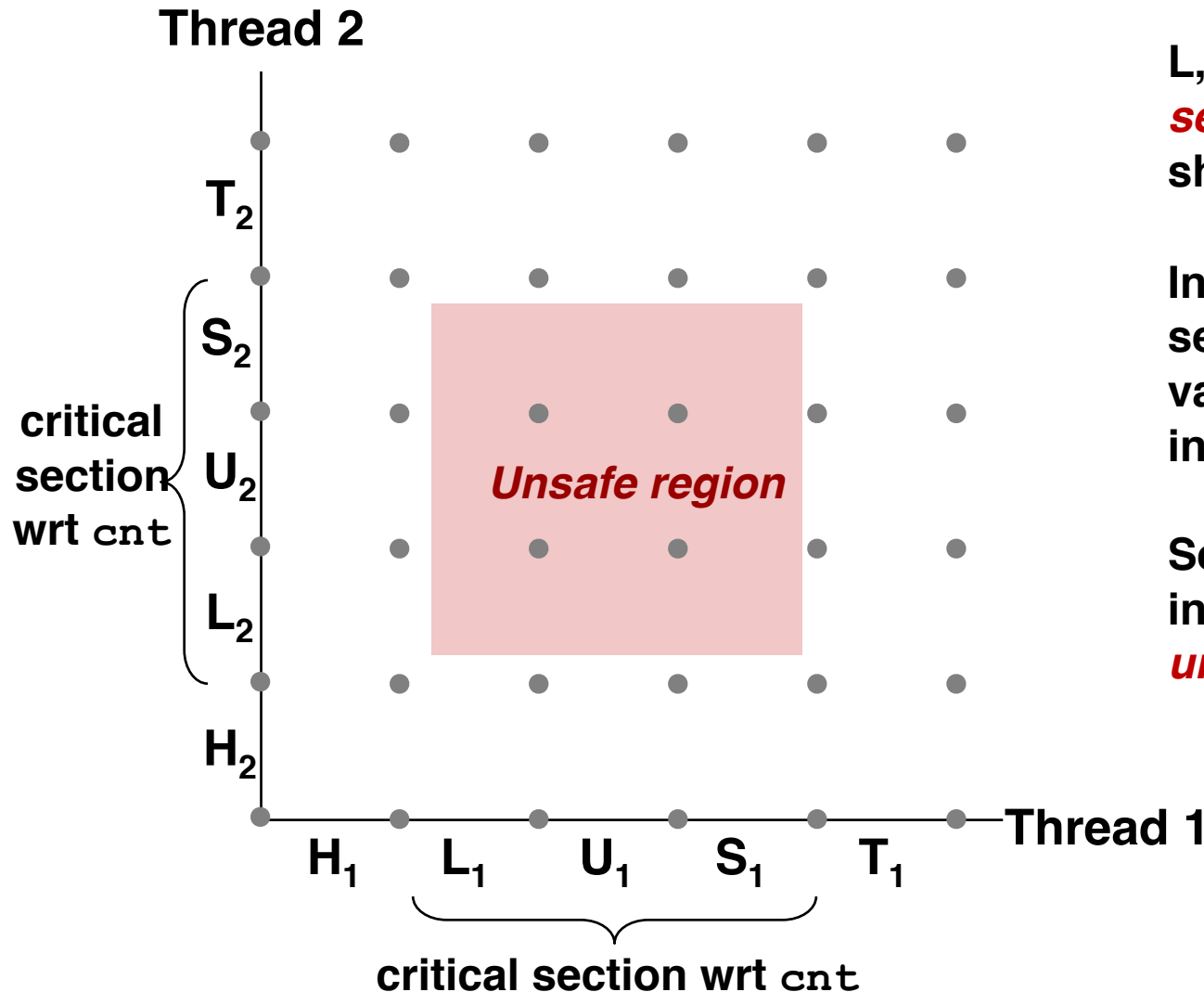
- Code section (a sequence of instructions) where no more than one thread should be executing concurrently.
 - Critical section refers to code, but its intention is to protect data!
- Threads need to have **mutually exclusive** access to critical section. That is, the execution of the critical section must be **atomic**: instructions in a CS either are executed entirely without interruption or not executed at all.

critical
section
wrt cnt



<pre>movq (%rdi), %rcx testq %rcx, %rcx jle .L2 movl \$0, %eax</pre>	} H_i : Head
<pre>----- .L3: movq cnt(%rip), %rdx addq \$1, %rdx movq %rdx, cnt(%rip)</pre>	} L_i : Load cnt U_i : Update cnt S_i : Store cnt
<pre>----- addq \$1, %rax cmpq %rcx, %rax jne .L3 .L2:</pre>	} T_i : Tail

Critical Sections and Unsafe Regions

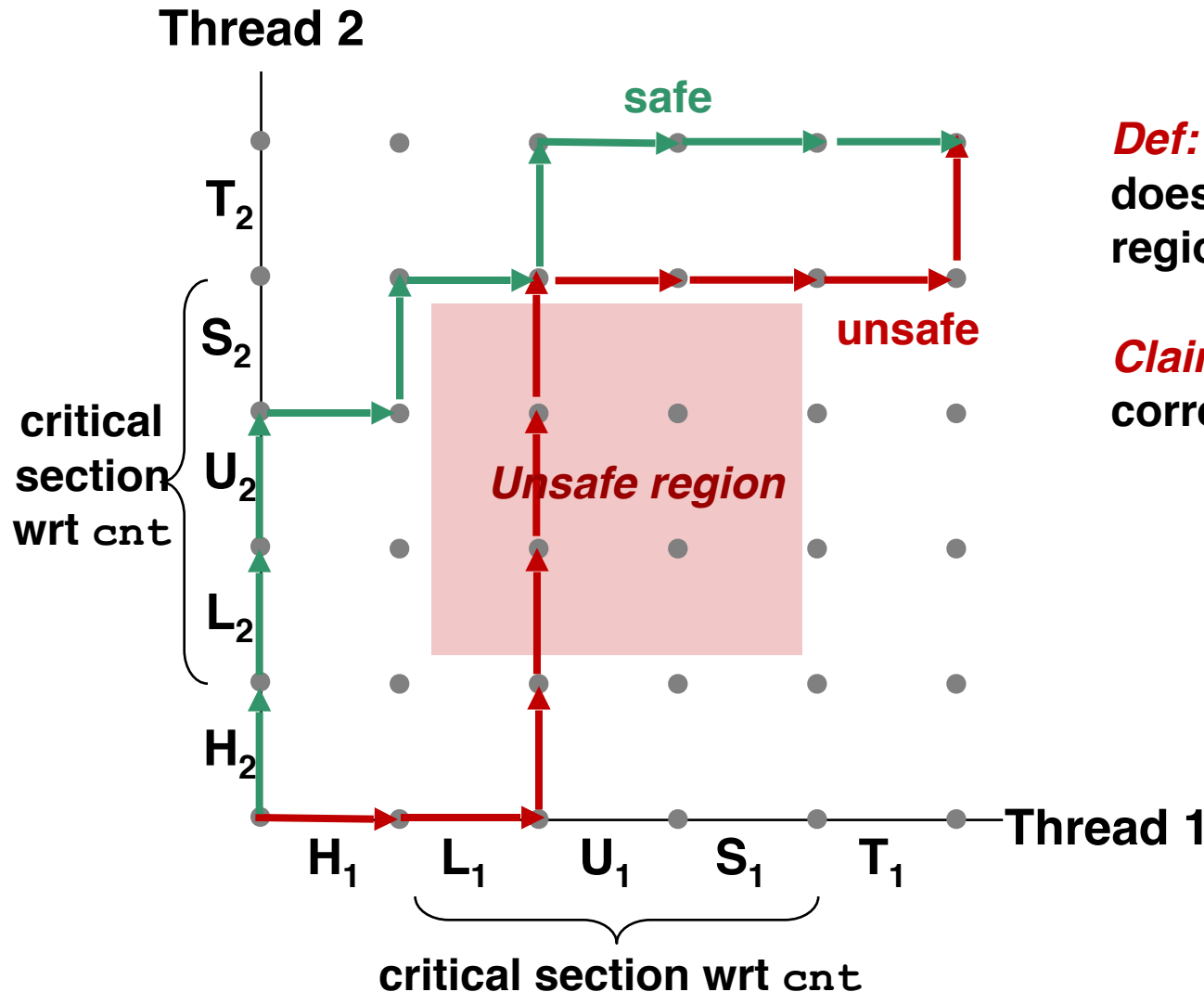


L, U, and S form a **critical section** with respect to the shared variable `cnt`

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

Enforcing Mutual Exclusion

- We must *coordinate/synchronize* the execution of the threads
 - i.e., need to guarantee ***mutually exclusive access*** for each critical section.
- Classic solution:
 - Semaphores/mutex (Edsger Dijkstra)
- Other approaches
 - Condition variables
 - Monitors (Java)
 - 254/258 discusses these

Using Semaphores for Mutual Exclusion

- Basic idea:

- Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
- Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
- Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
- No more than one thread can be in the critical section at a time.

- Terminology

- **Binary semaphore** is also called **mutex** (i.e., the semaphore value could only be 0 or 1)
- Think of P operation as “**locking**”, and V as “**unlocking**”.

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`.

Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.
 - Checking mutex value and setting its value must be an atomic unit: they either are performed entirely or not performed at all.
 - on x86: the atomic test-and-set instruction.

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

goodcnt.c

```
function Lock(boolean *lock) {  
    while (test_and_set(lock) == 1);  
}
```

Deadlock

- Def: A process/thread is *deadlocked* if and only if it is waiting for a condition that will never be true
- General to concurrent/parallel programming (threads, processes)
- Typical Scenario
 - Processes 1 and 2 needs two resources (A and B) to proceed
 - Process 1 acquires A, waits for B
 - Process 2 acquires B, waits for A
 - Both will wait forever!

Deadlocking With Semaphores

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}

int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

Tid[0]:
P(s₀);
P(s₁);
cnt++;
V(s₀);
V(s₁);

Tid[1]:
P(s₁);
P(s₀);
cnt++;
V(s₁);
V(s₀);

Avoiding Deadlock

Acquire shared resources in same order

```
Tid[0]:  
P(s0);  
P(s1);  
cnt++;  
V(s0);  
V(s1);
```

```
Tid[1]:  
P(s1);  
P(s0);  
cnt++;  
V(s1);  
V(s0);
```



```
Tid[0]:  
P(s0);  
P(s1);  
cnt++;  
V(s0);  
V(s1);
```

```
Tid[1]:  
P(s0);  
P(s1);  
cnt++;  
V(s1);  
V(s0);
```