Concurrent Programming: *Synchronizing threads:* 1. *The Mutual Exclusion Problem*

Synchronizing Threads

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

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;
}
```

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;
}
```

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

cnt should equal 20,000.

What went wrong?

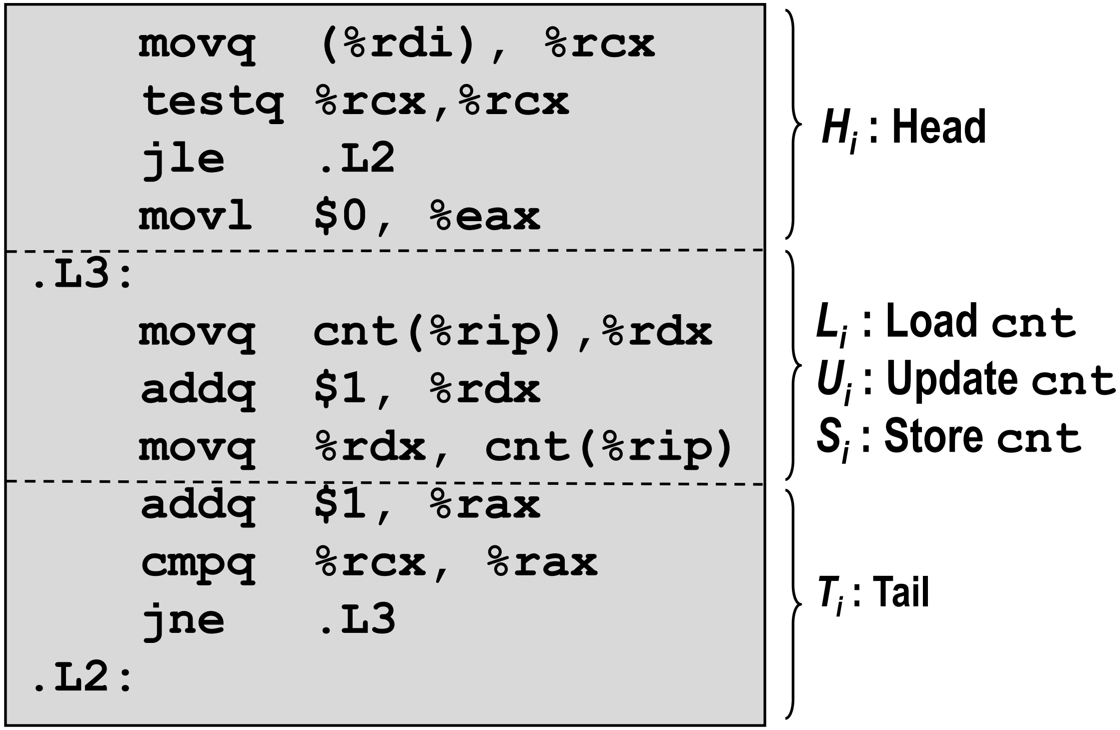
(would also fail: ./badcnt 1)

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

🟩 **Thread 1 critical section**

🟪 **Thread 2 critical section**

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	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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	$\%\text{rdx}_1$	$\%\text{rdx}_2$	cnt
1	H_1			
1	L_1			
2	H_2			
2	L_2			
2	U_2			
2	S_2			
1	U_1			
1	S_1			
1	T_1			
2	T_2			

Concurrent Execution (cont)

- How about this ordering?

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

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			
2	U_2			
2	S_2			
1	U_1			
1	S_1			
1	T_1			
2	T_2			

Concurrent Execution (cont)

- How about this ordering?

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2			
2	S_2			
1	U_1			
1	S_1			
1	T_1			
2	T_2			

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	U$_1$			
1	S$_1$			
1	T$_1$			
2	T$_2$			

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	U_1			
1	S_1			
1	T_1			
2	T_2			

Concurrent Execution (cont)

- **How about this ordering?**

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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	S_1			
1	T_1			
2	T_2			

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	T_1			
2	T_2			

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	T_1			
2	T_2			

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			
2	T_2			

Concurrent Execution (cont)

- How about this ordering?

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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			

Concurrent Execution (cont)

- How about this ordering?

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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

Concurrent Execution (cont)

- **How about this ordering?**

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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!

Concurrent Execution (cont)

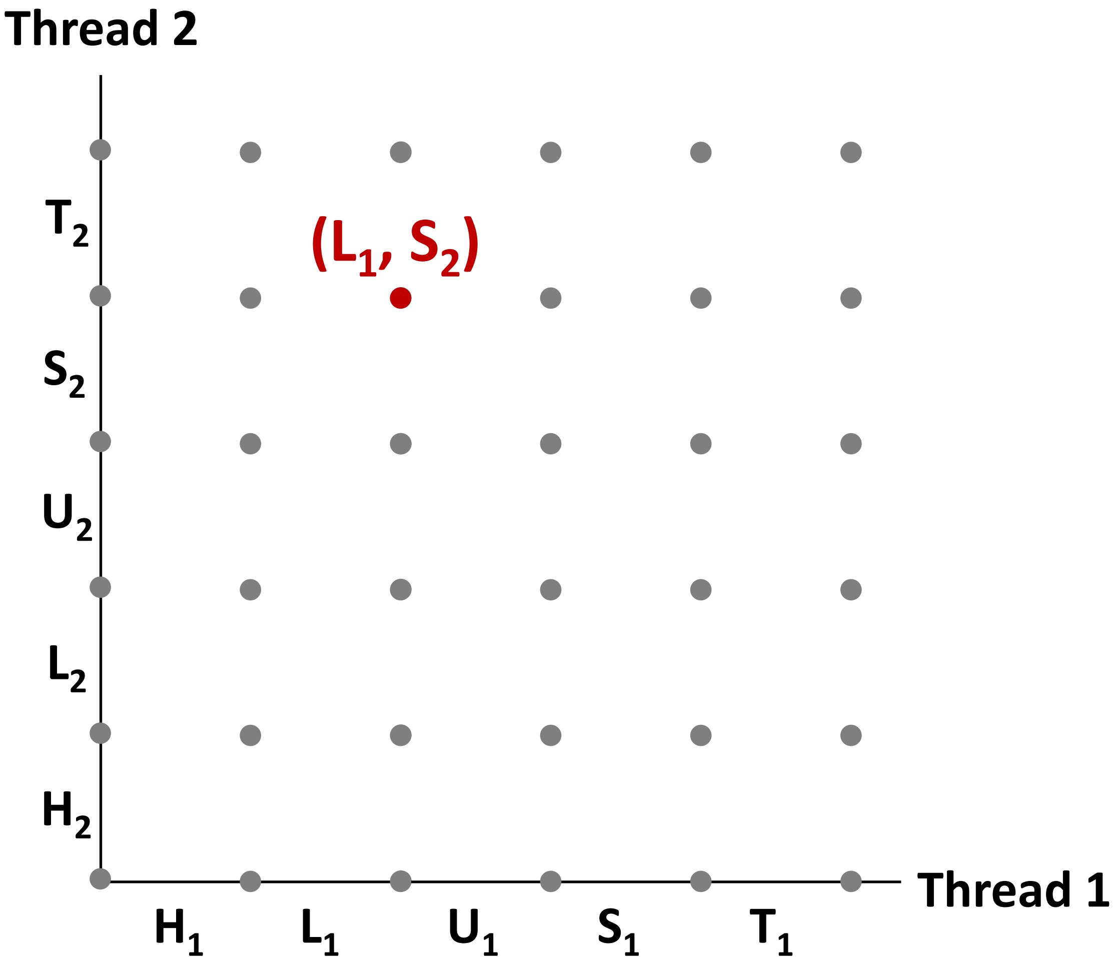
- How about this ordering?

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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



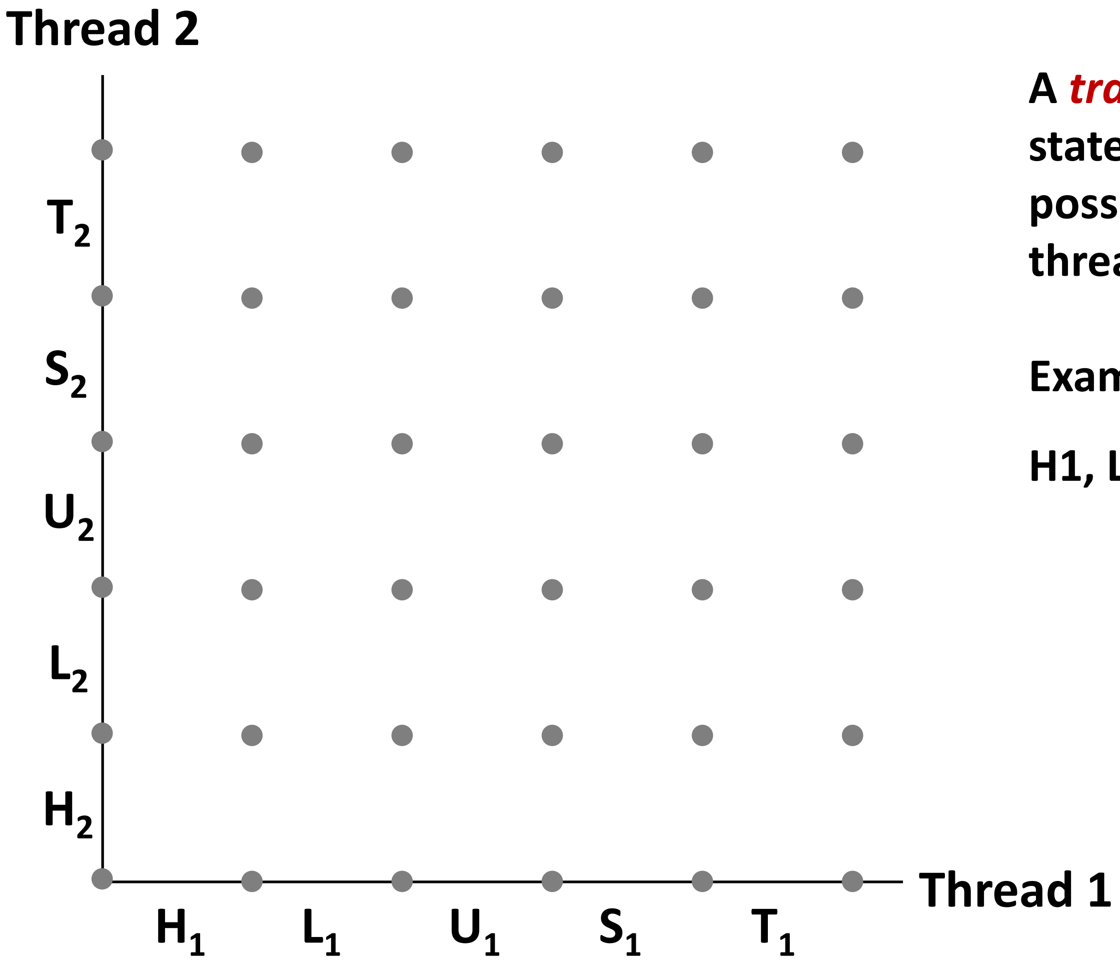
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

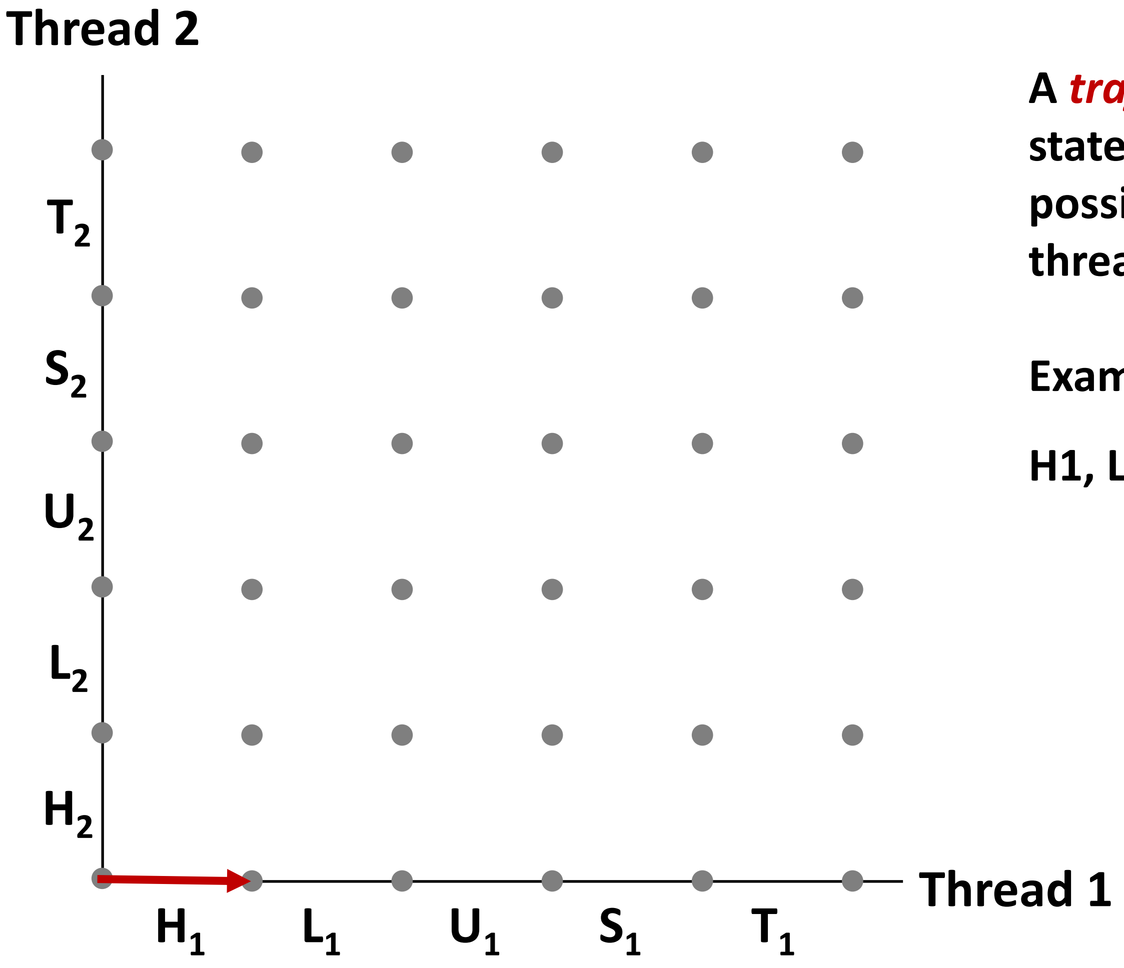


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

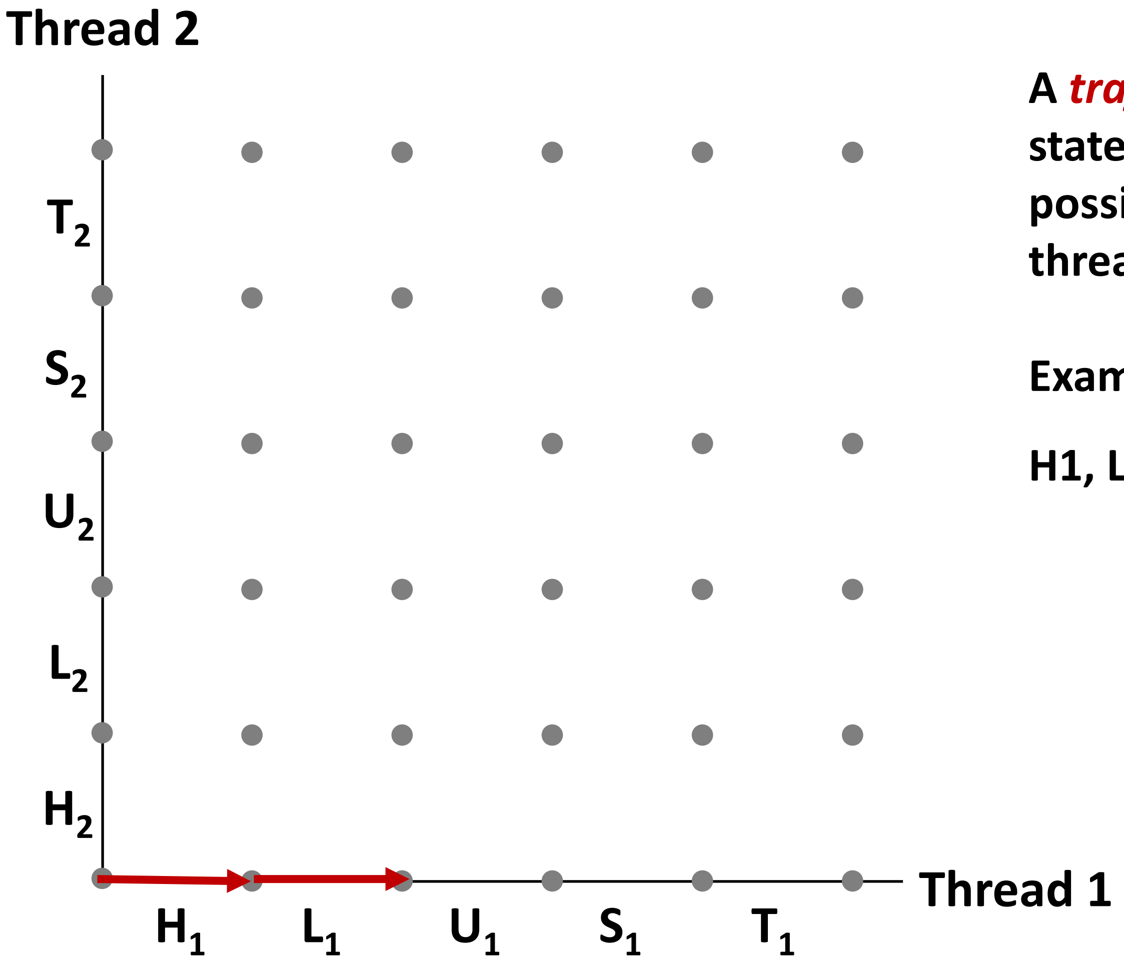


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

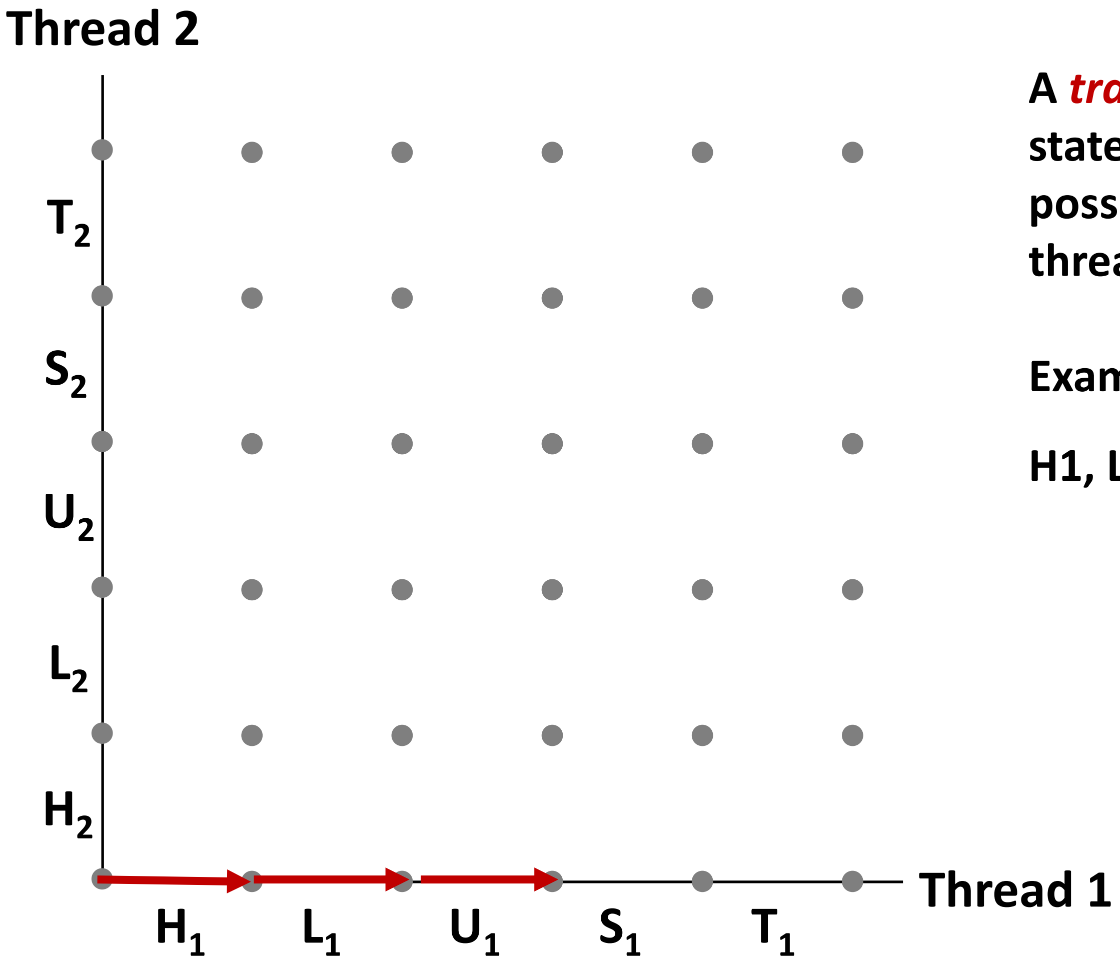


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

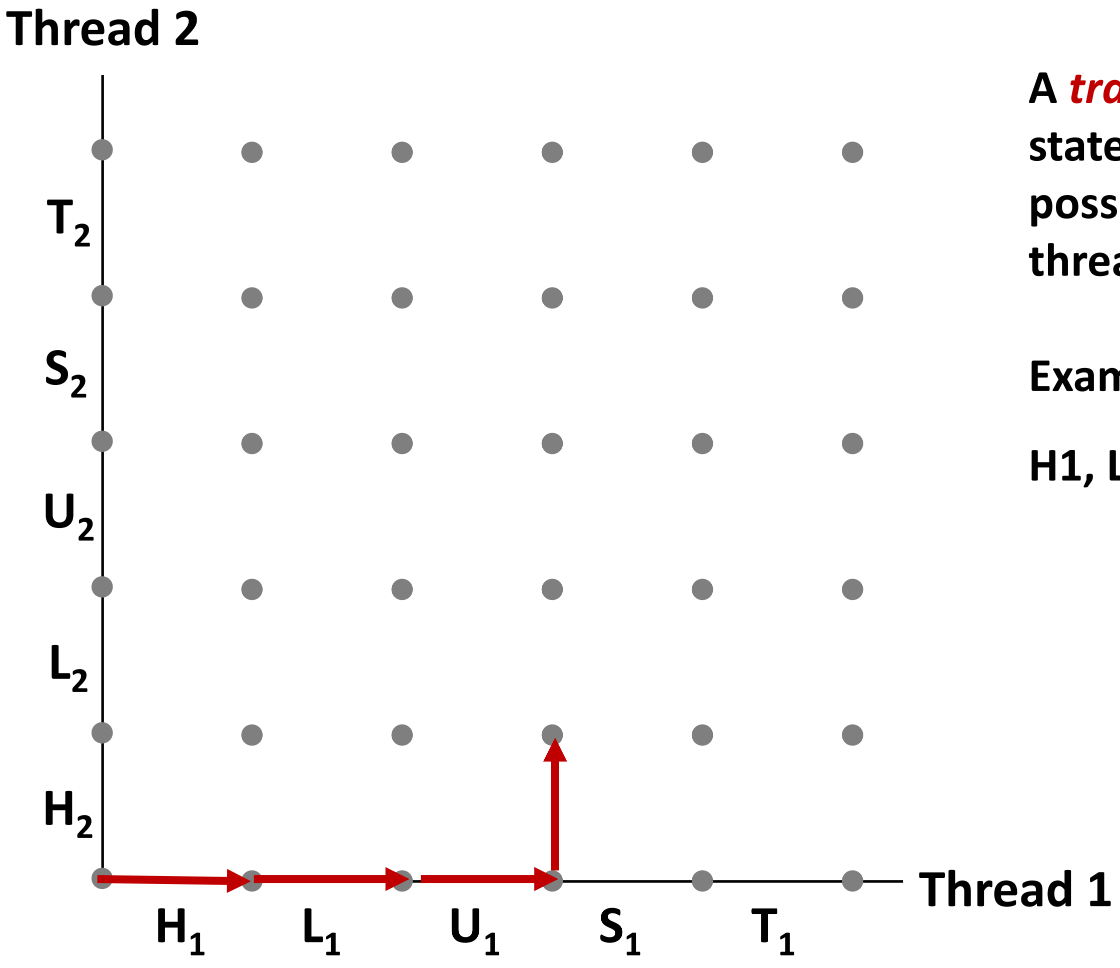


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

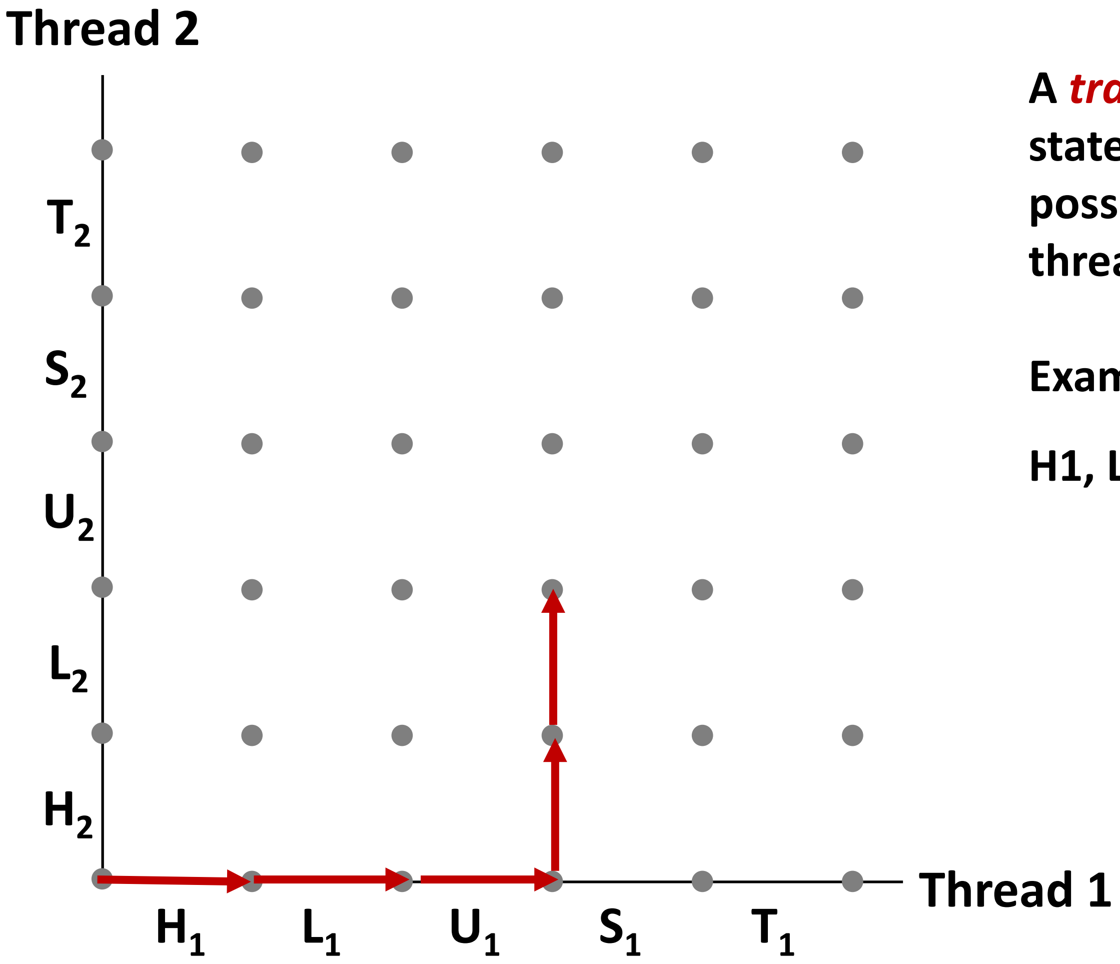


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

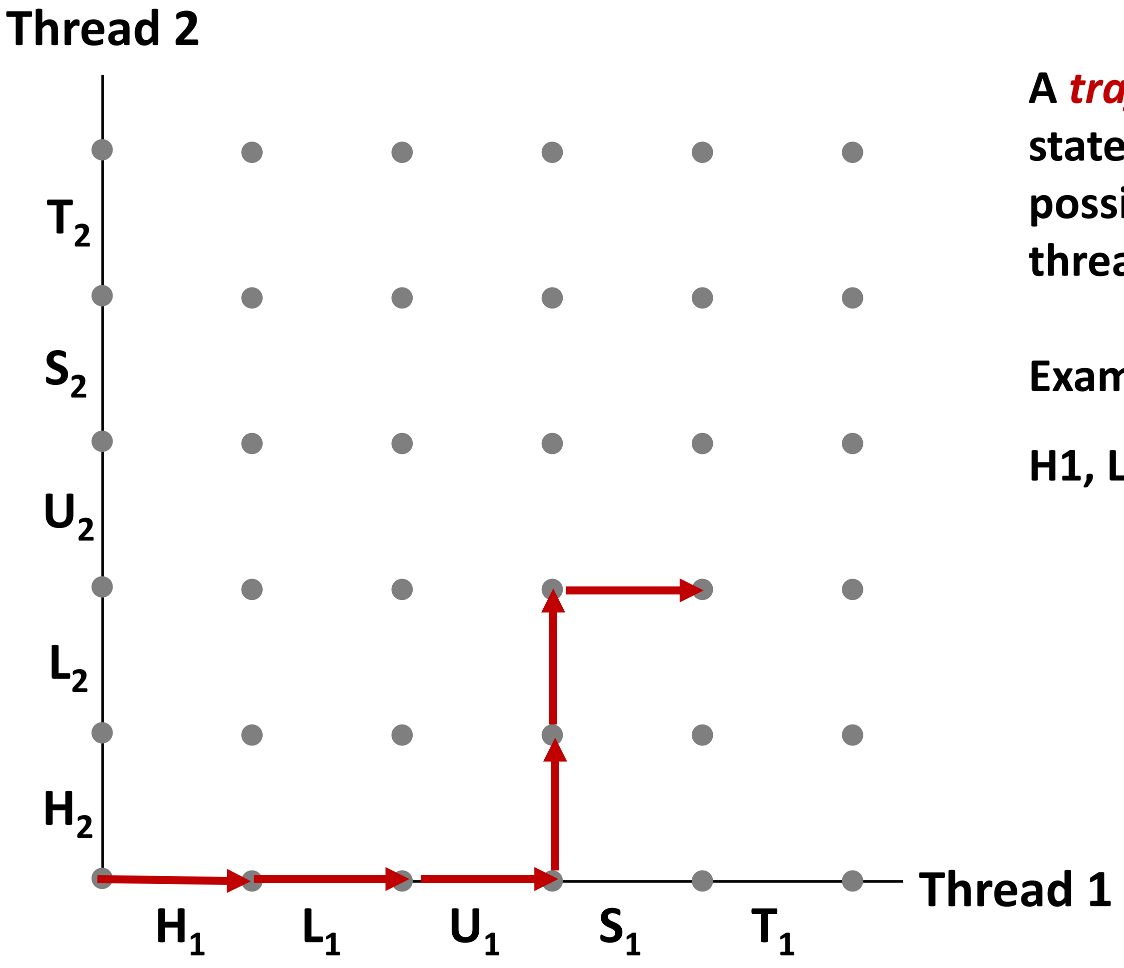


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

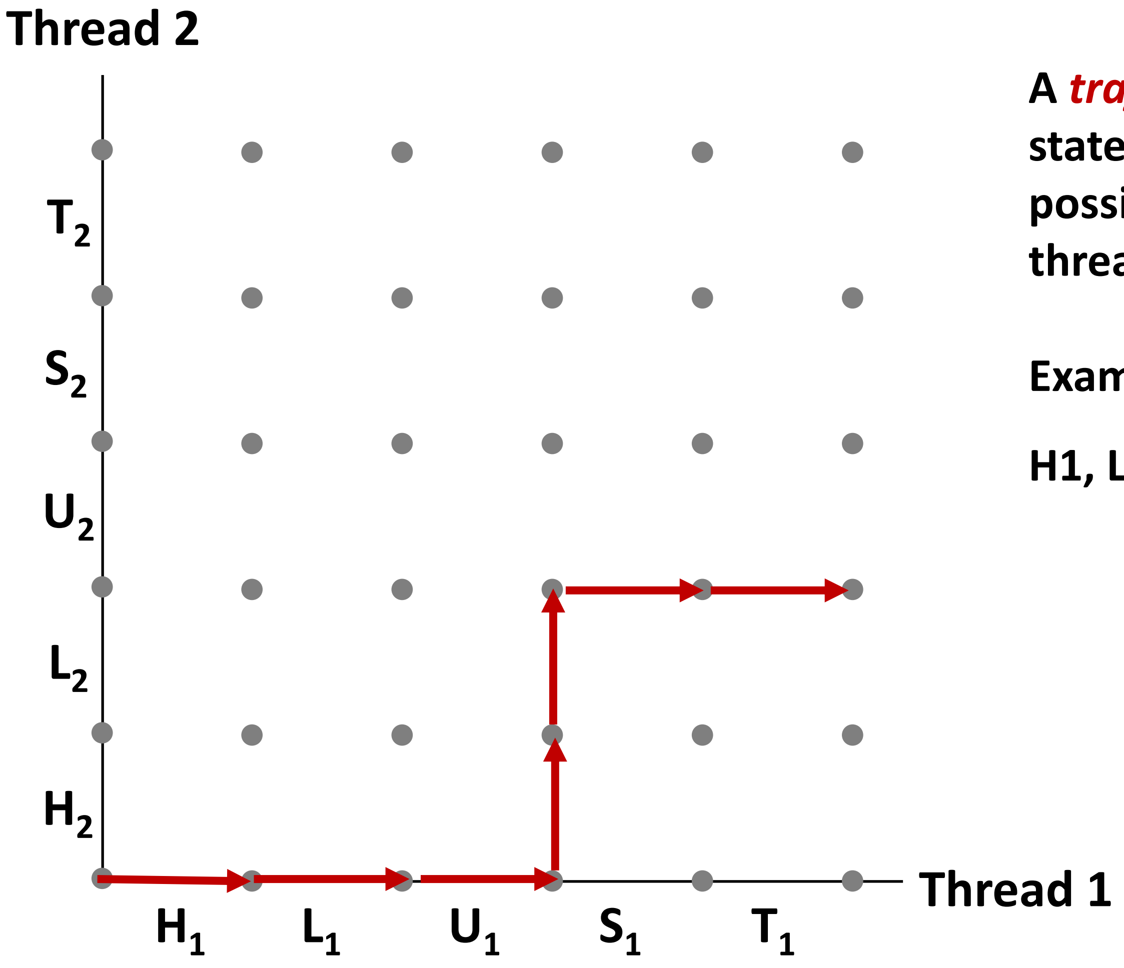


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

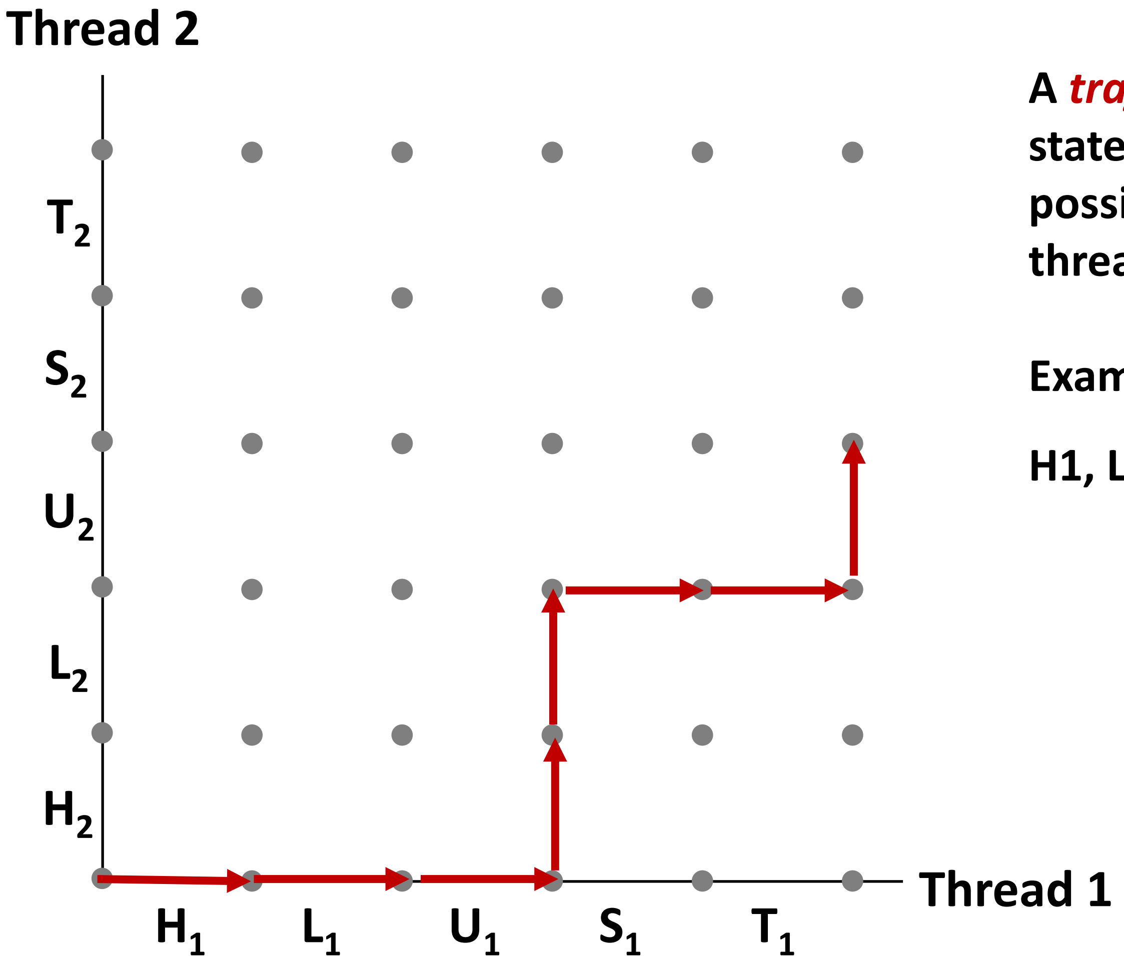


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

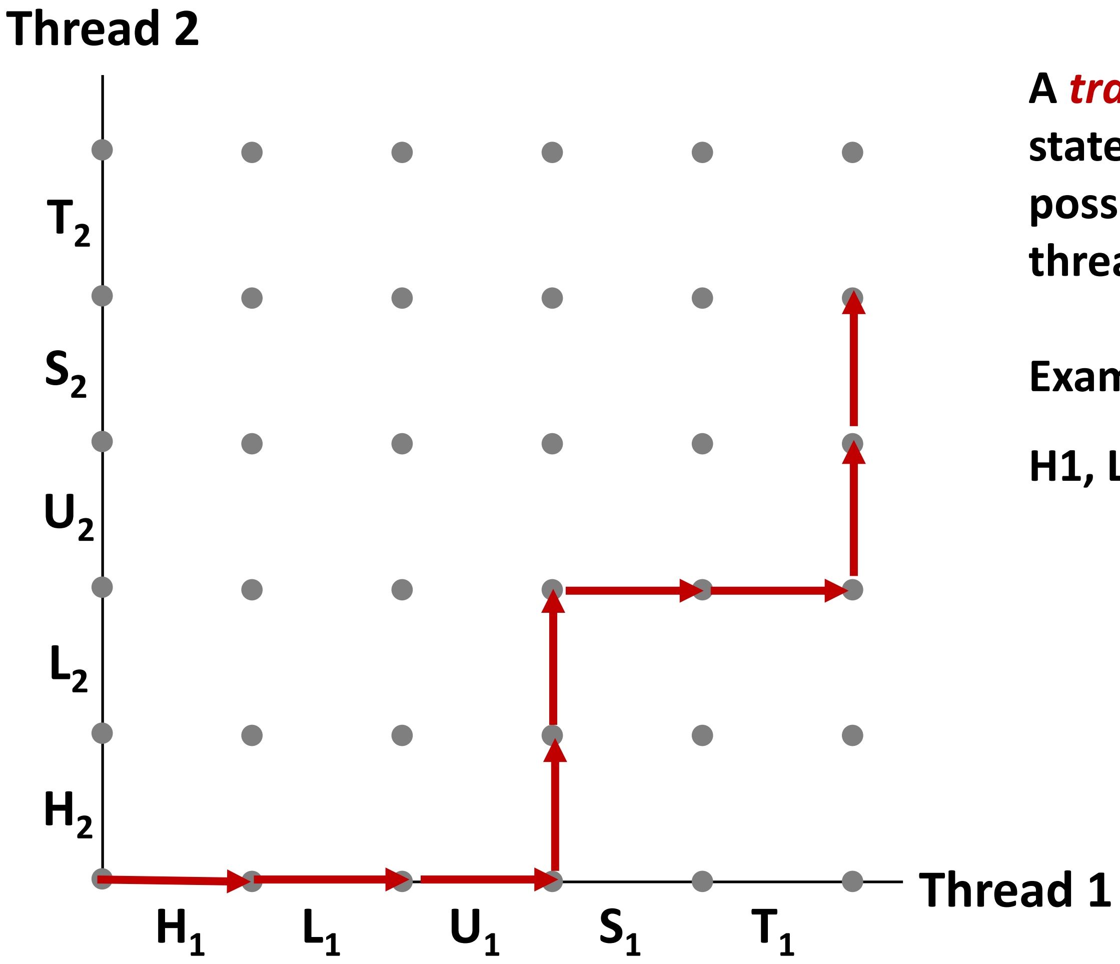


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

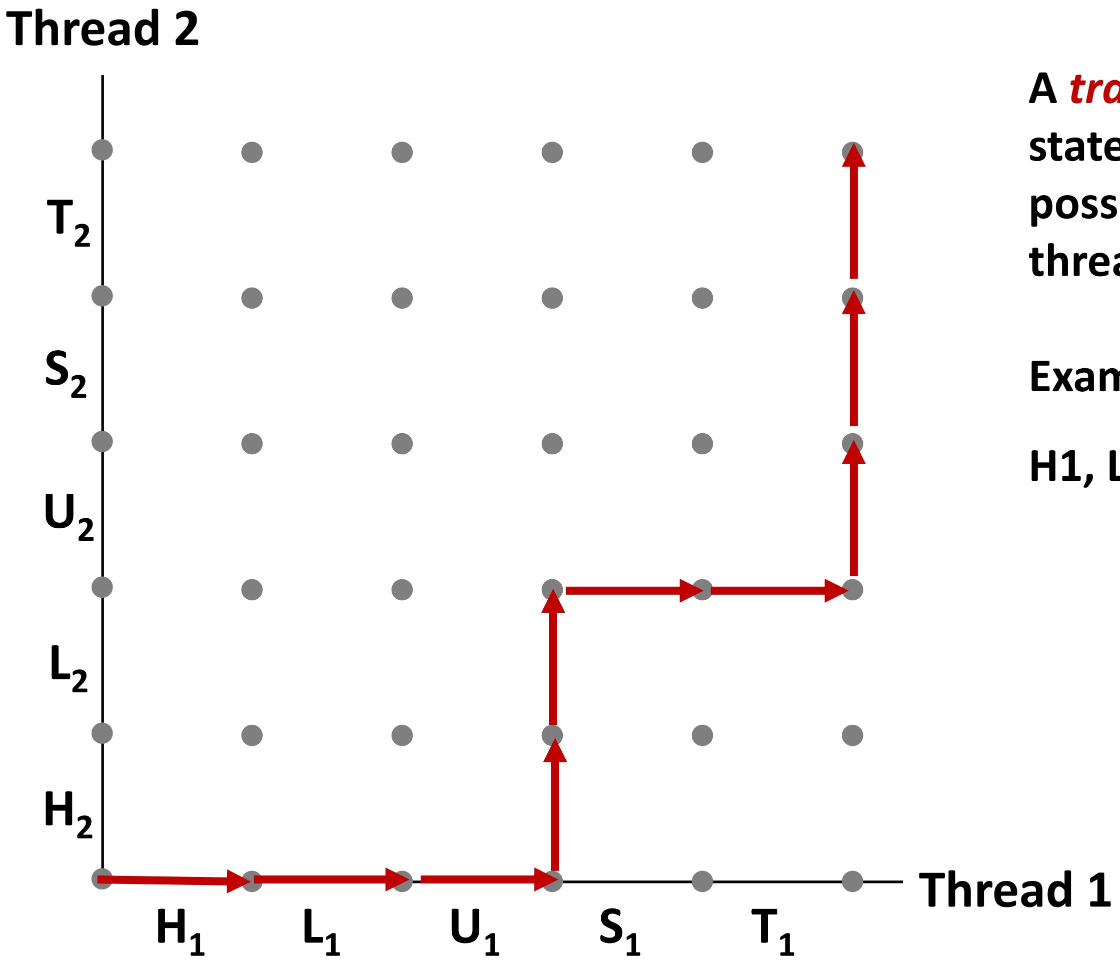


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

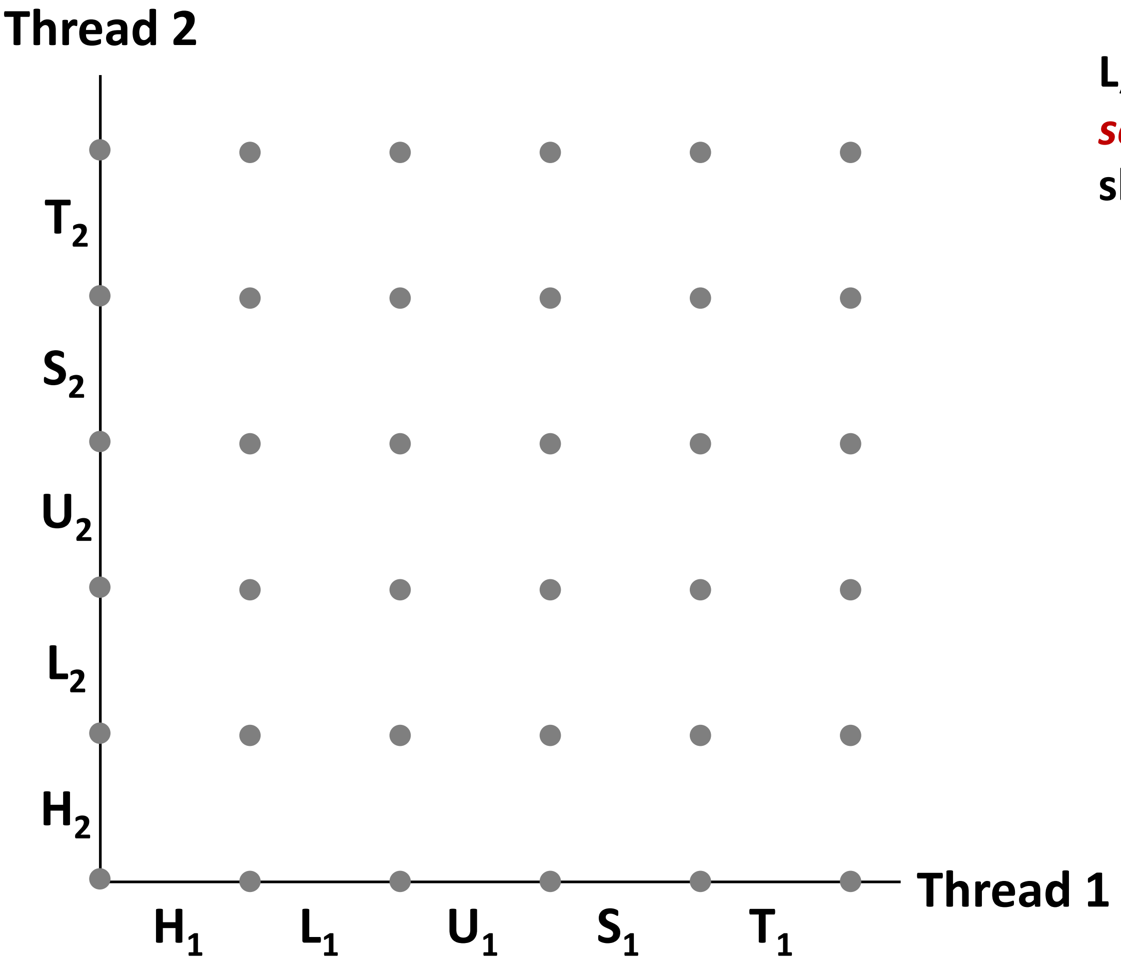


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

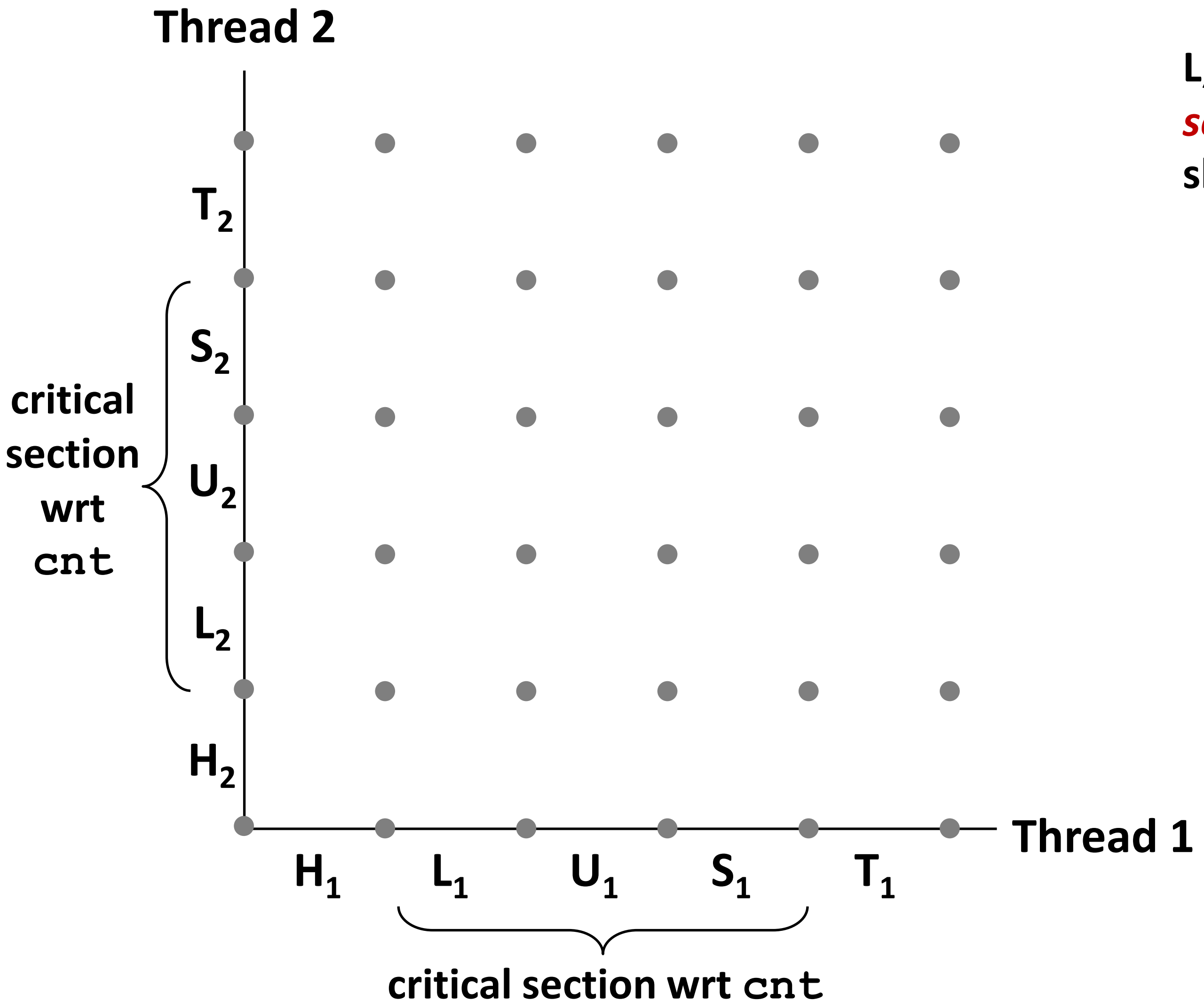
H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions



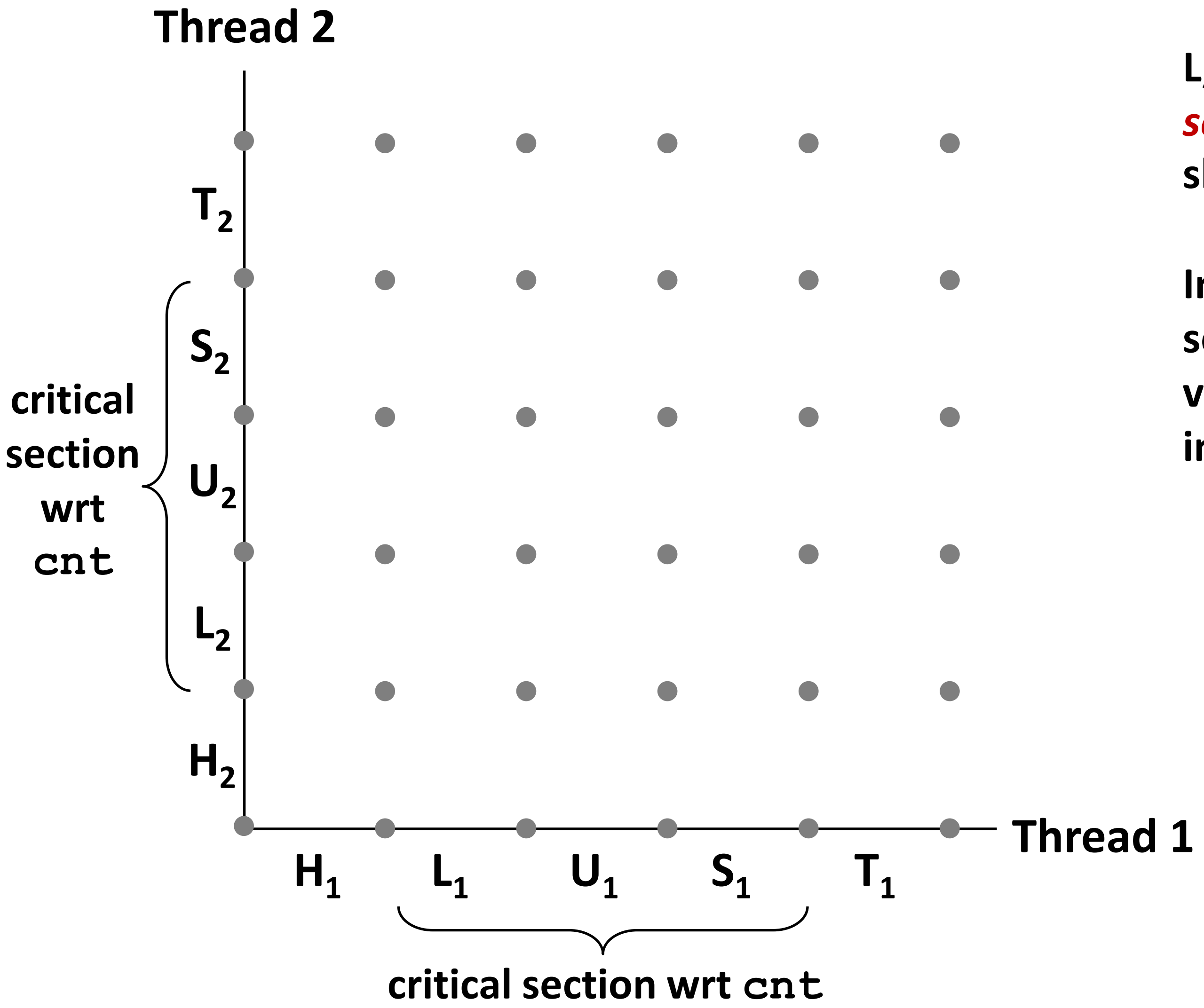
L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

Critical Sections and Unsafe Regions



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

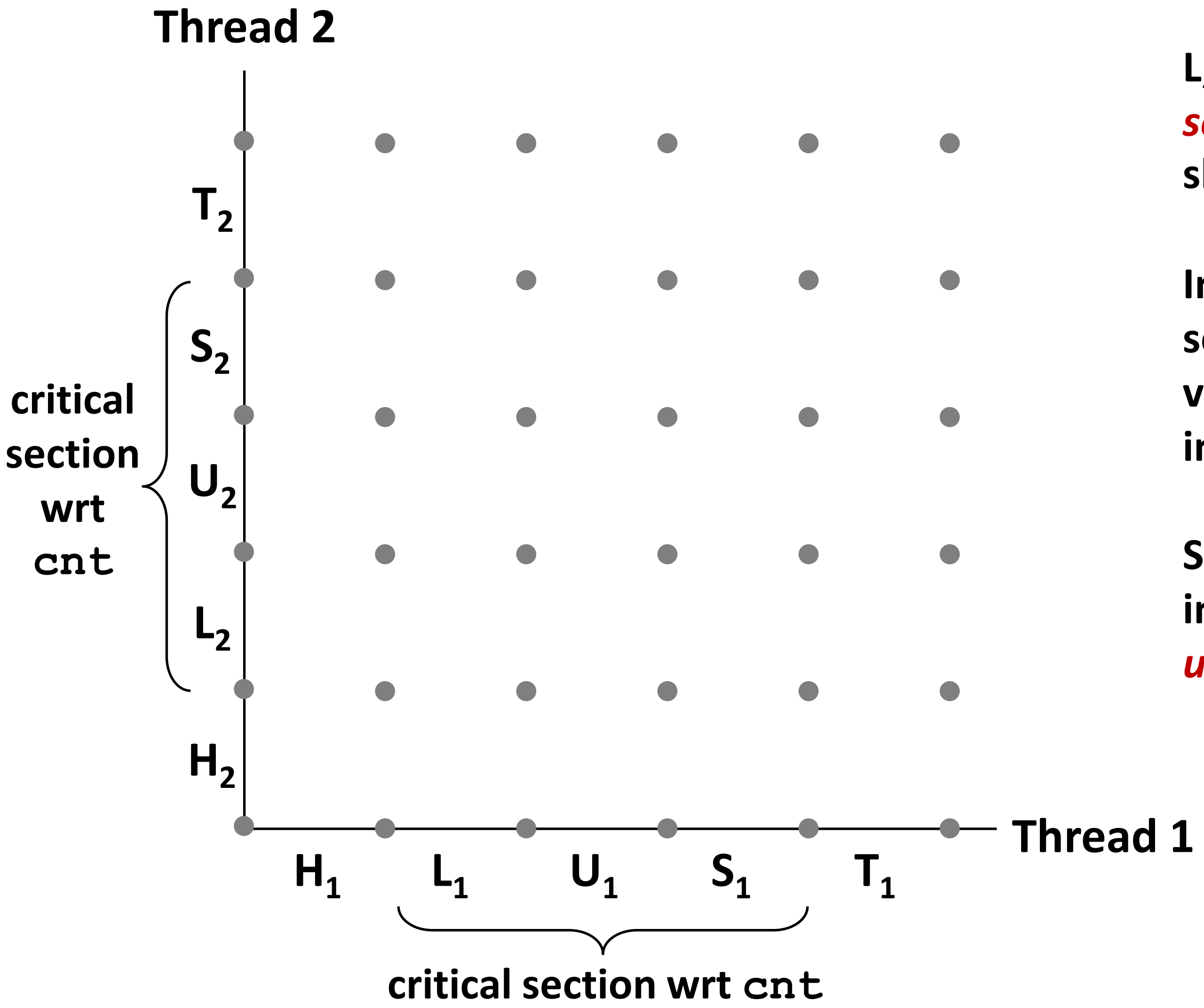
Critical Sections and Unsafe Regions



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

Critical Sections and Unsafe Regions

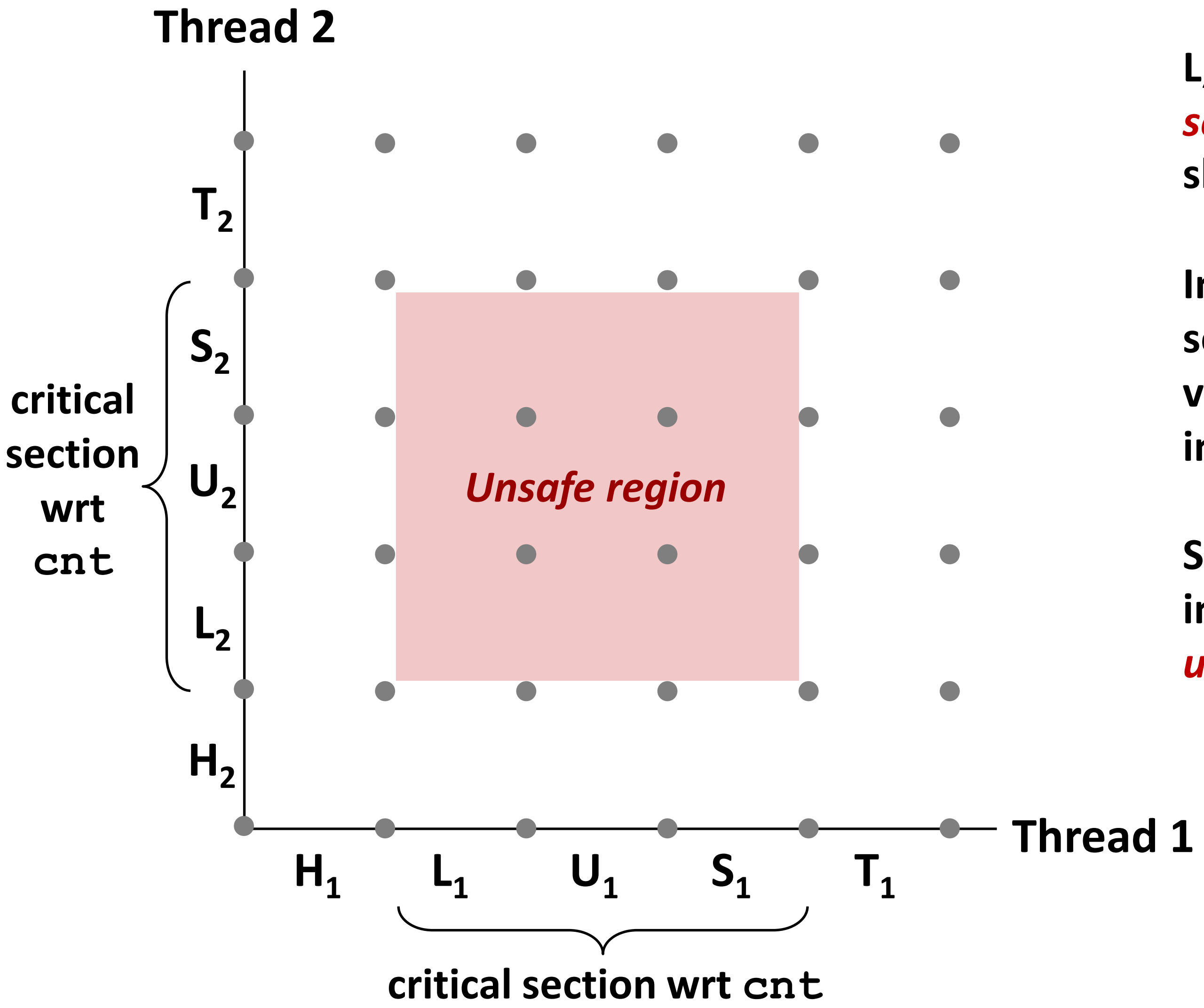


L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

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

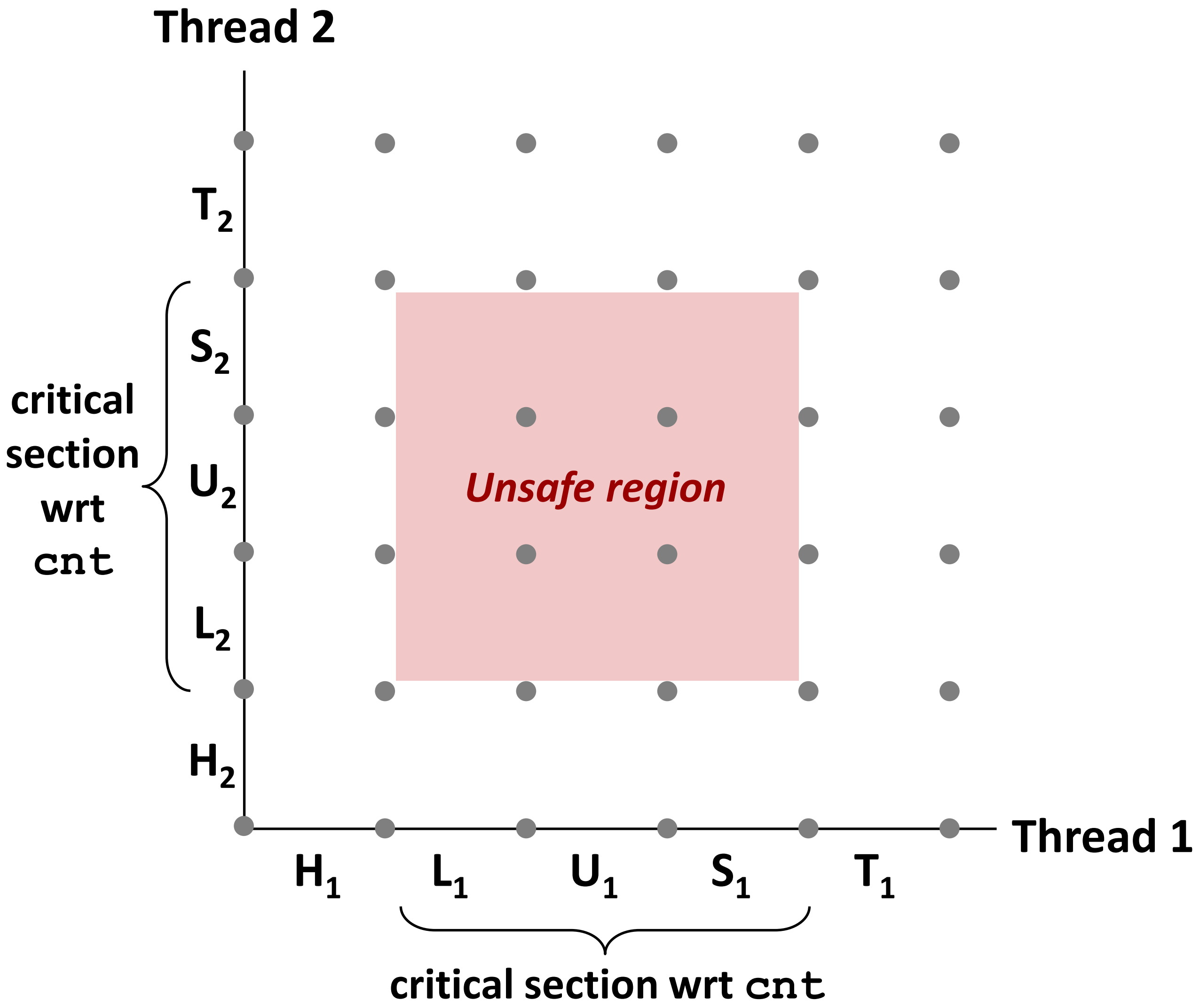


L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

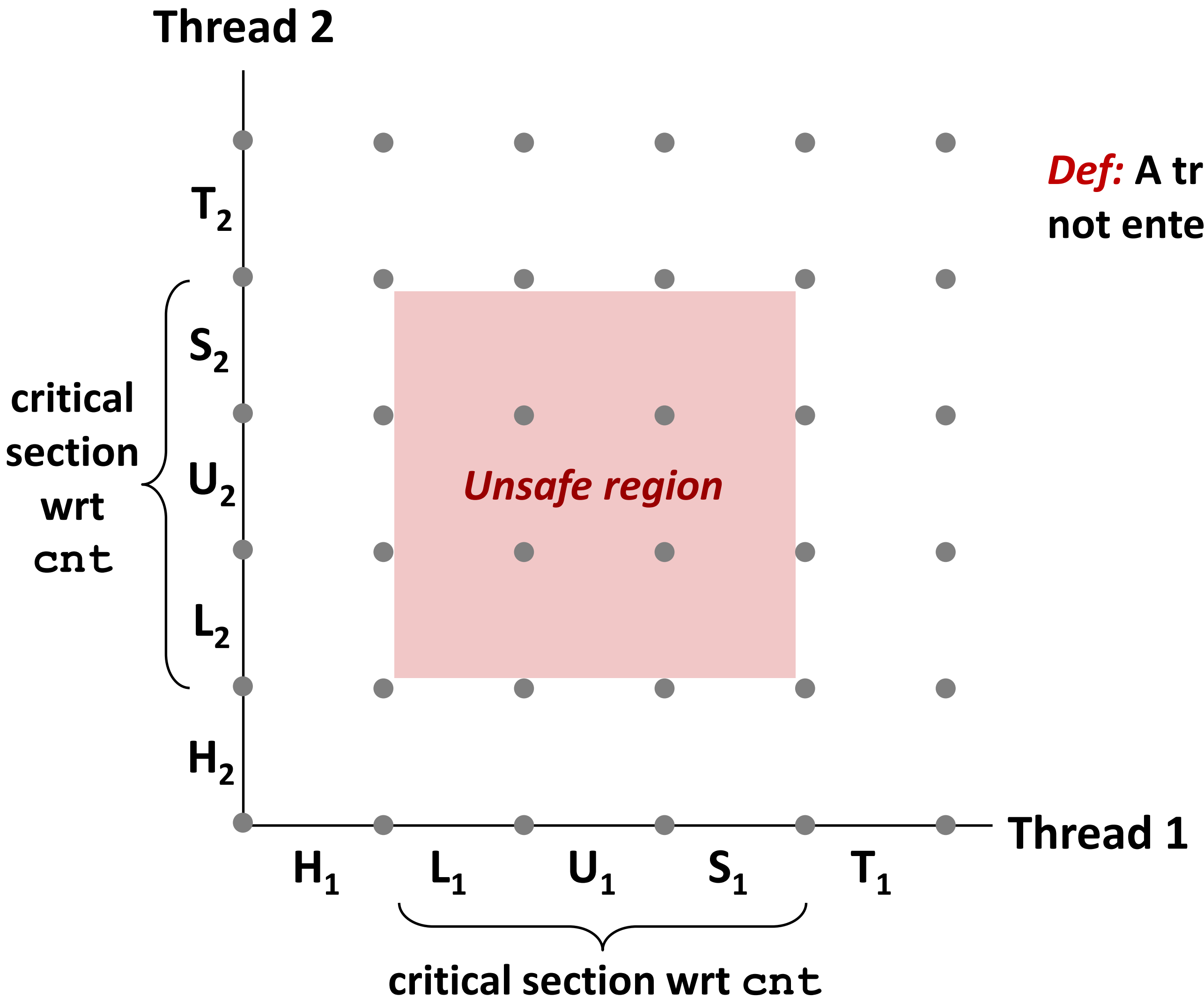
Instructions in critical sections (wrt some shared variable) should not be interleaved

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

Critical Sections and Unsafe Regions

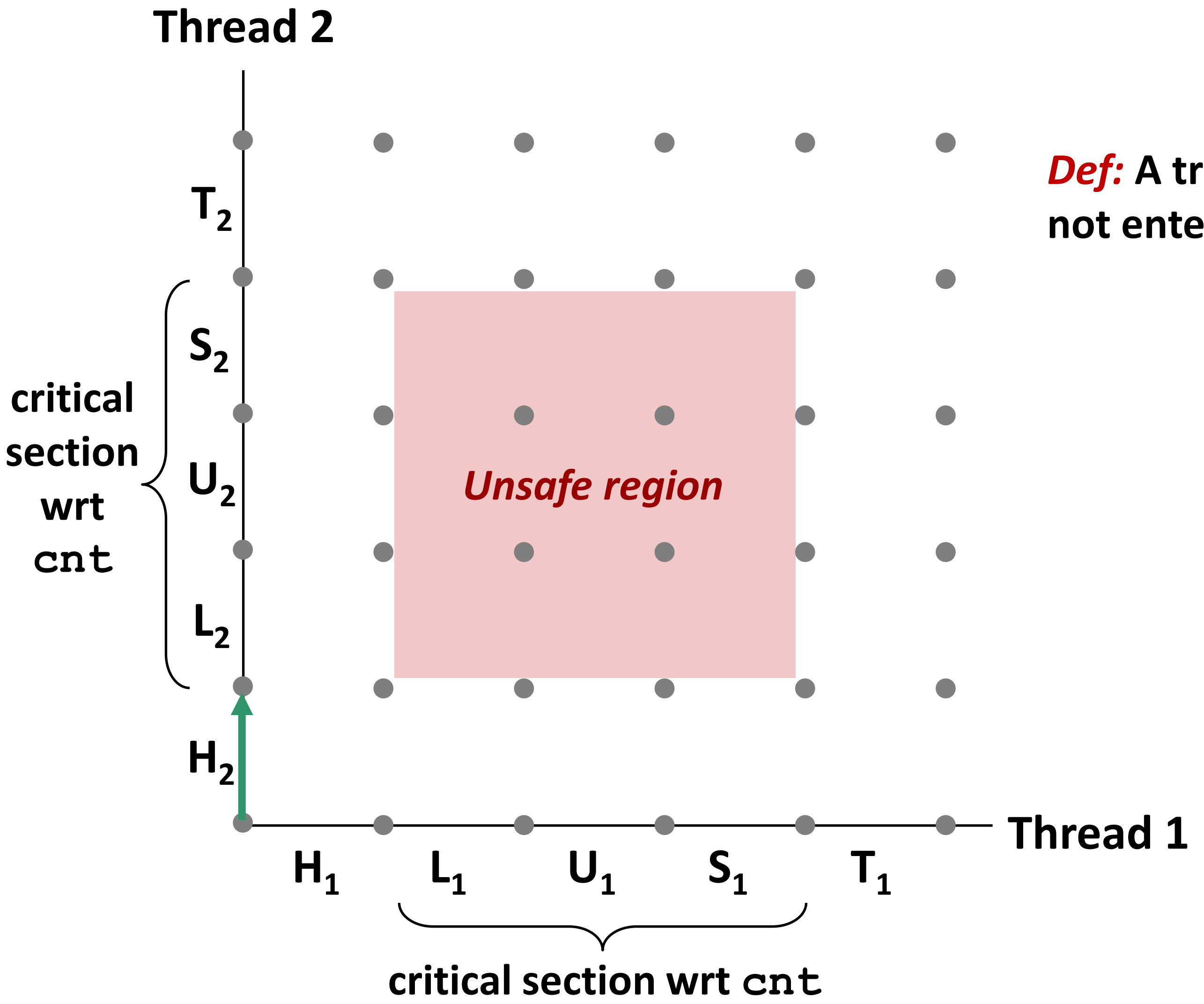


Critical Sections and Unsafe Regions



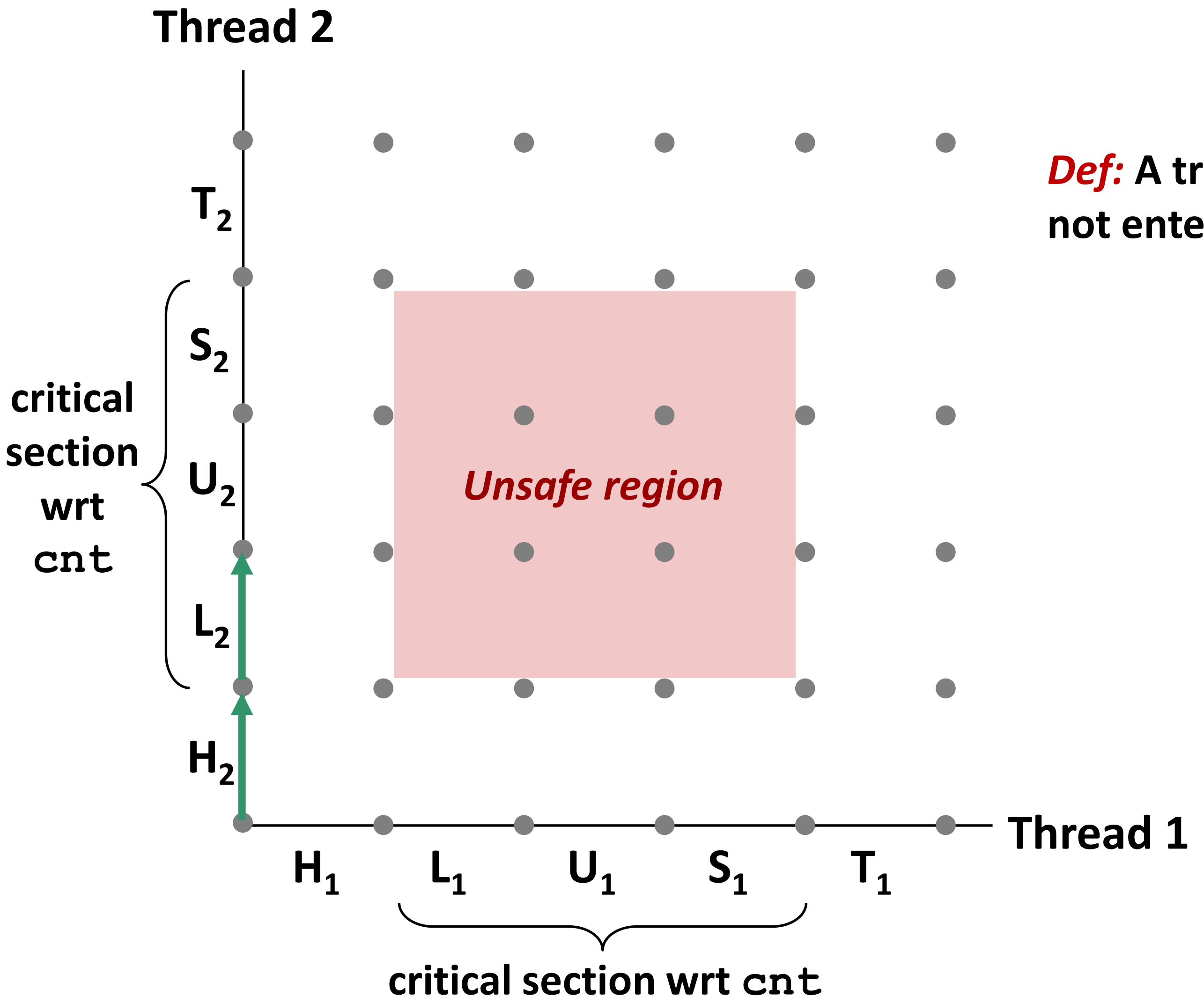
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



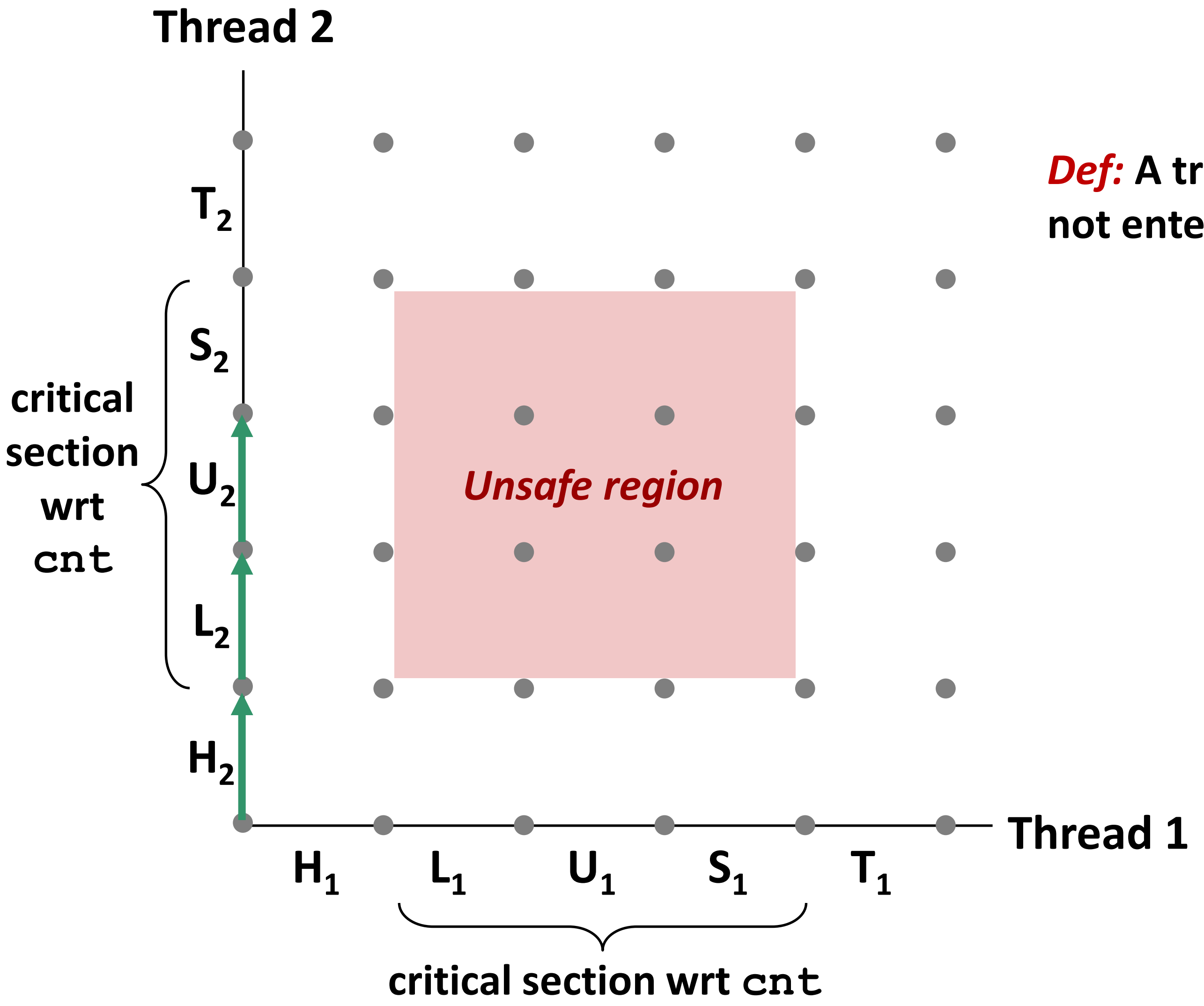
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



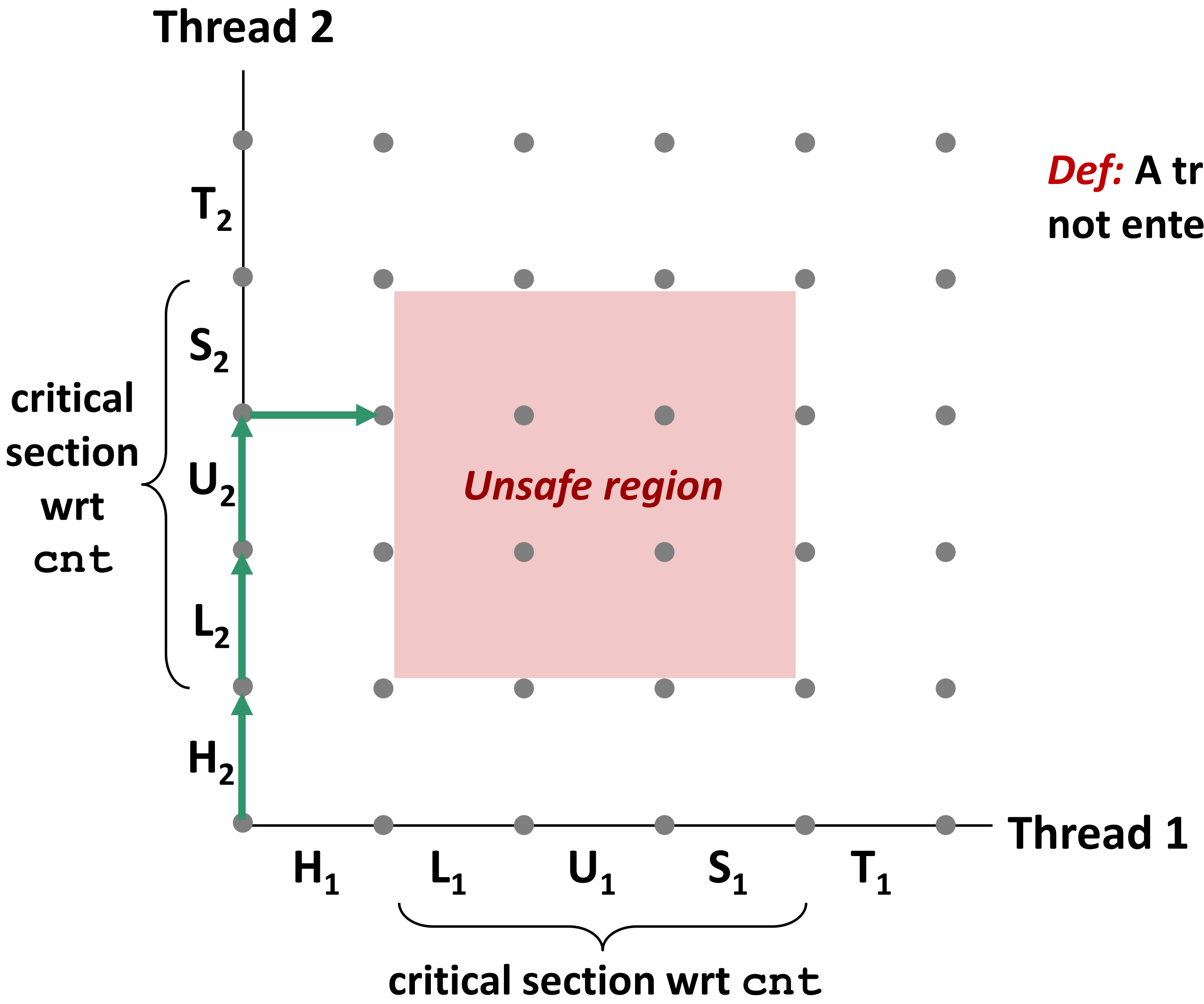
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



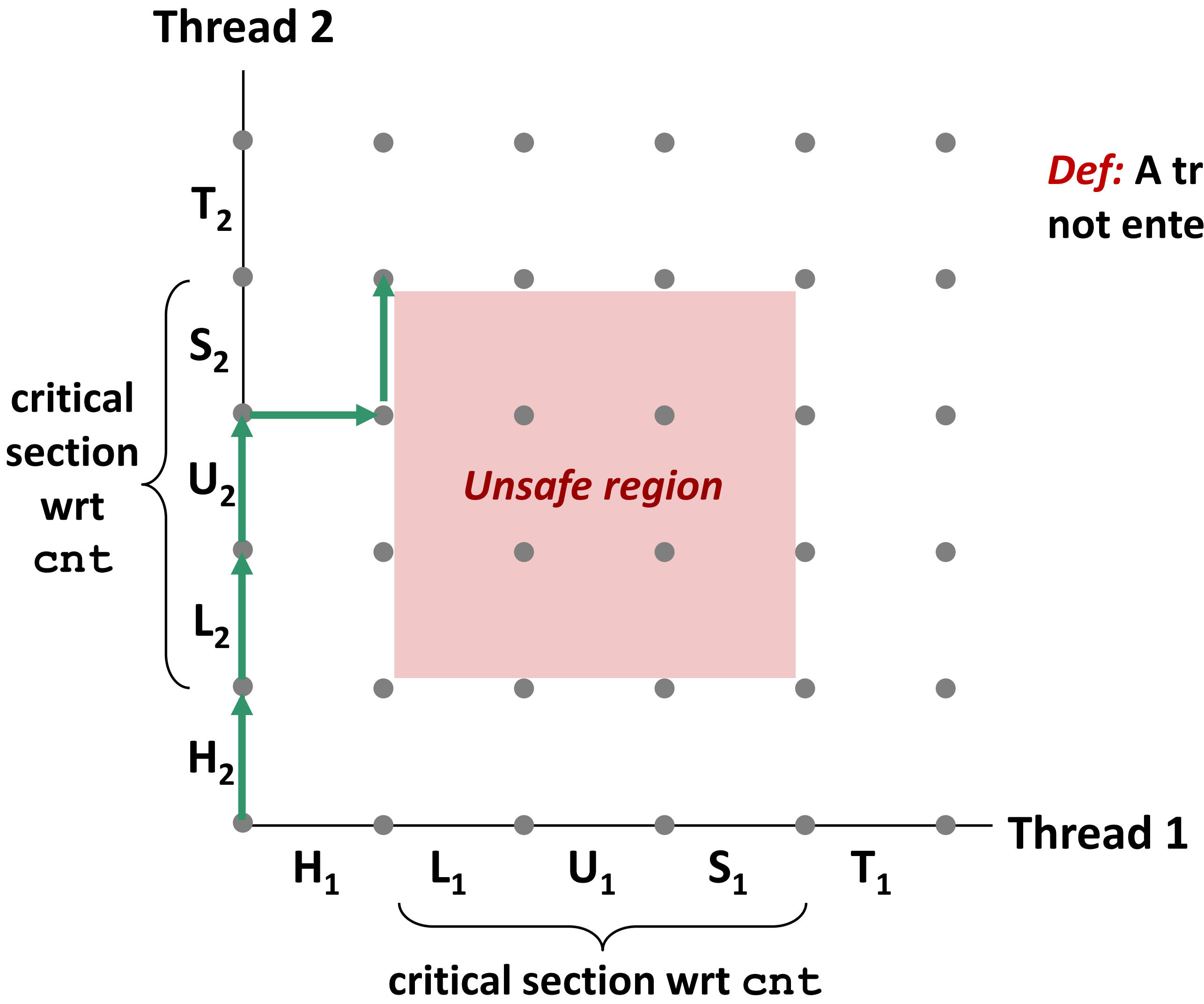
Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



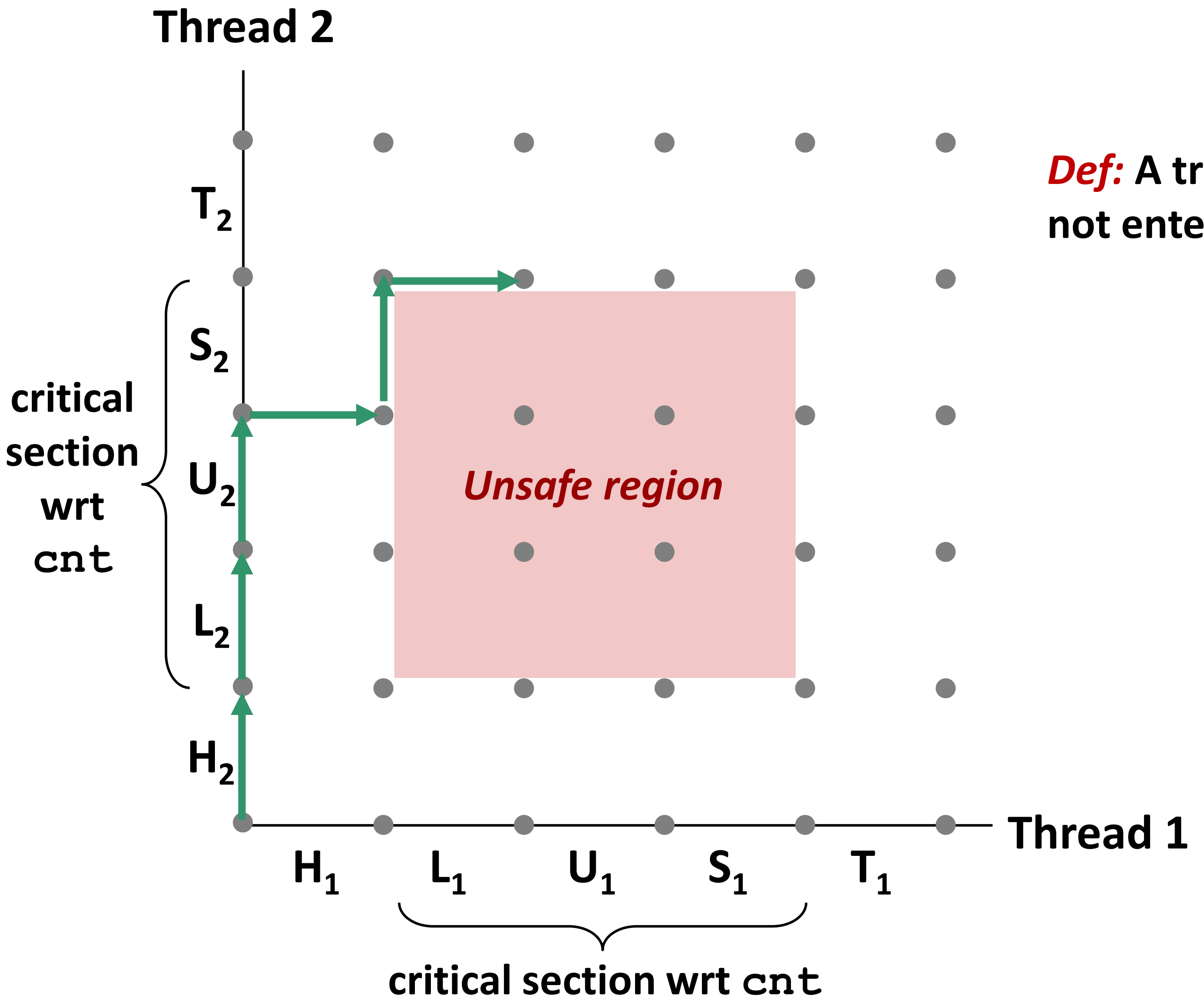
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



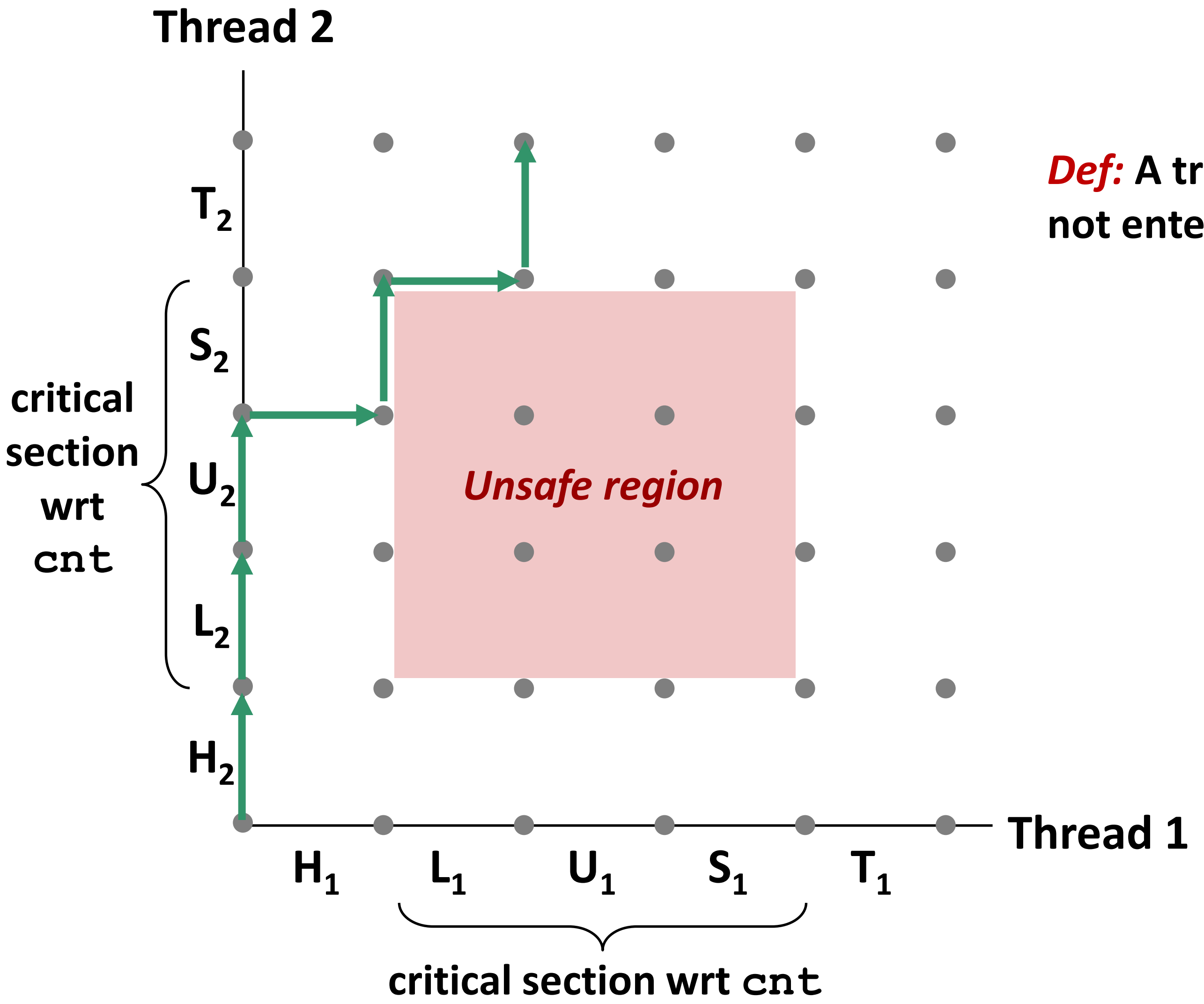
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



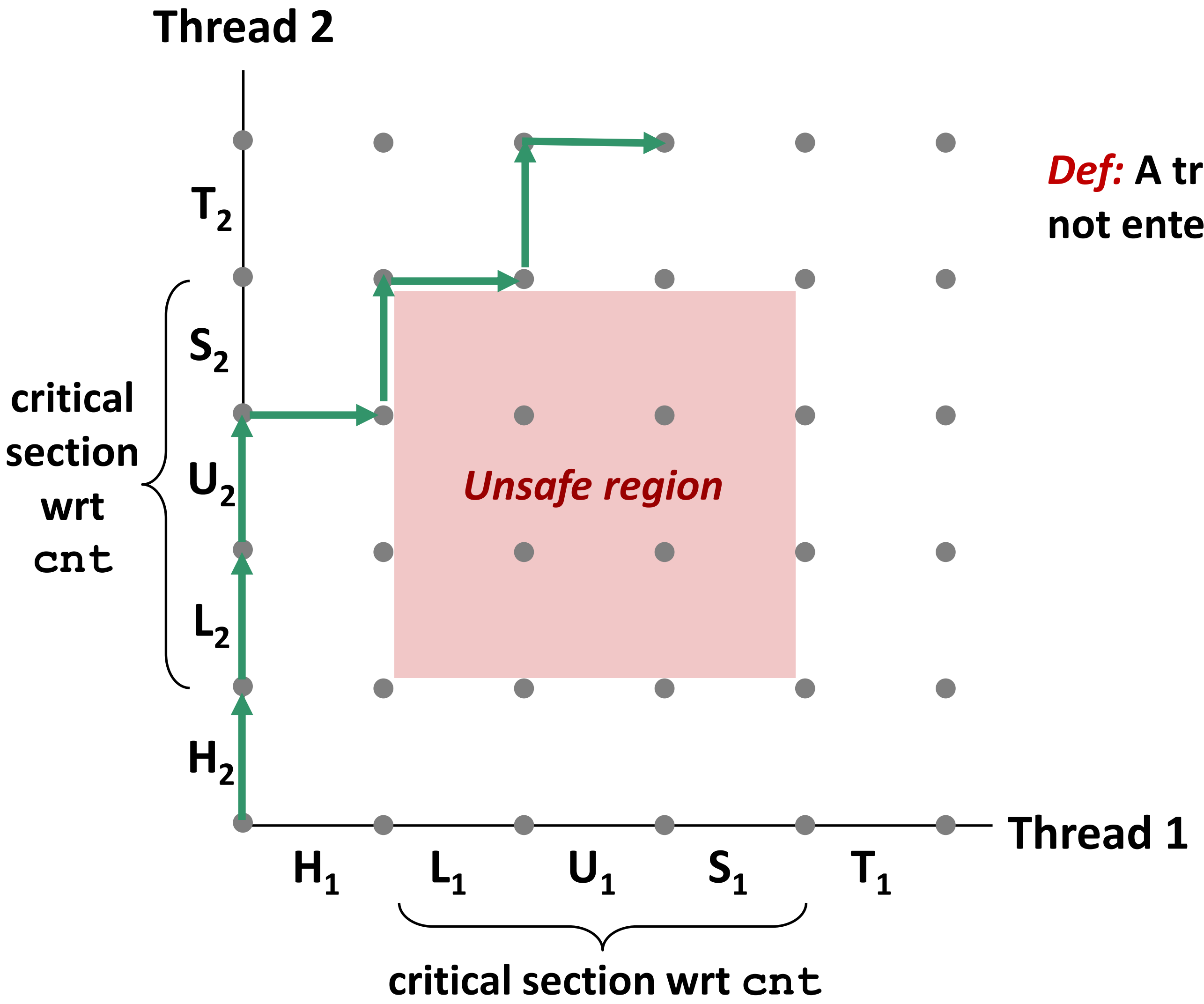
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



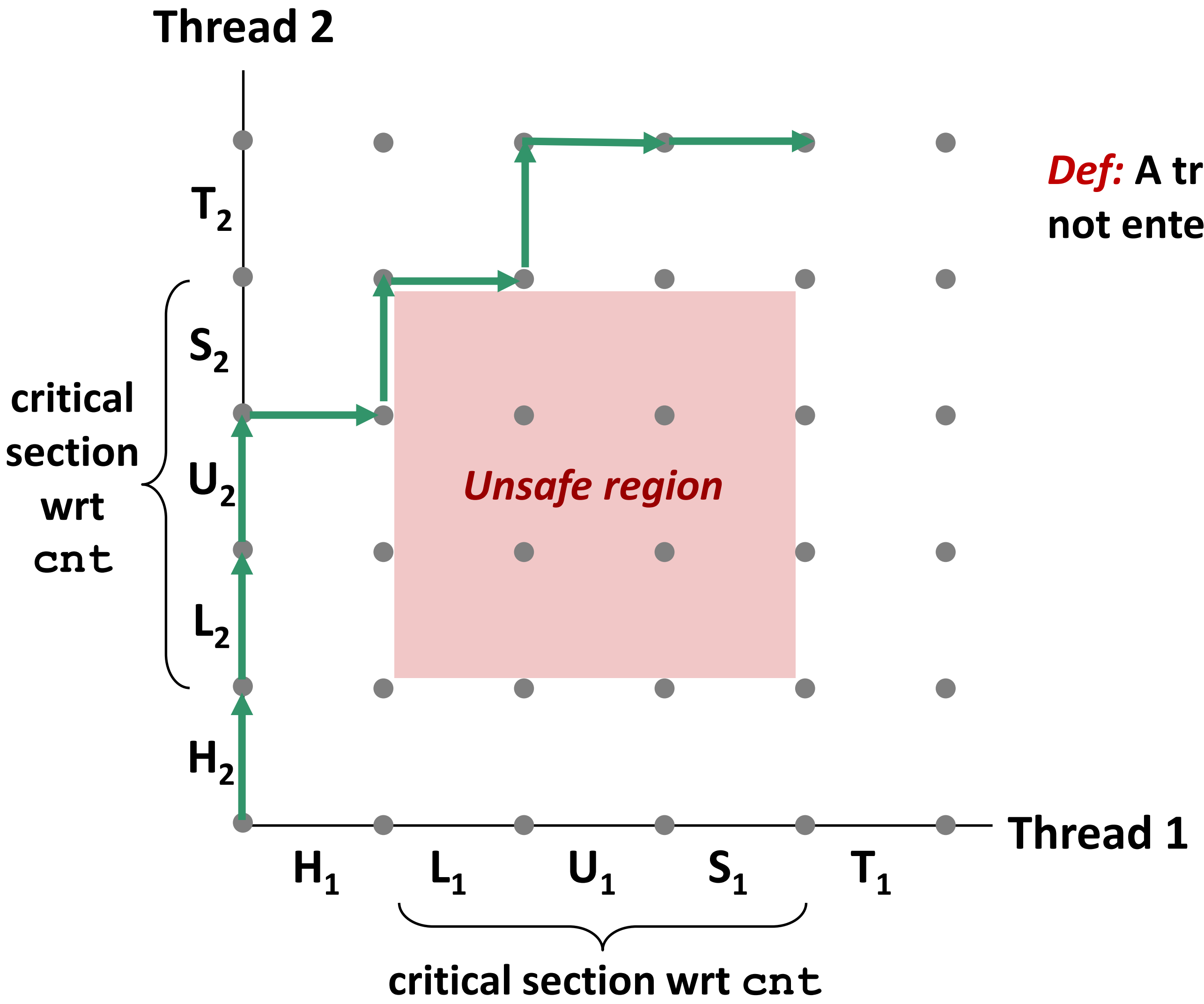
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



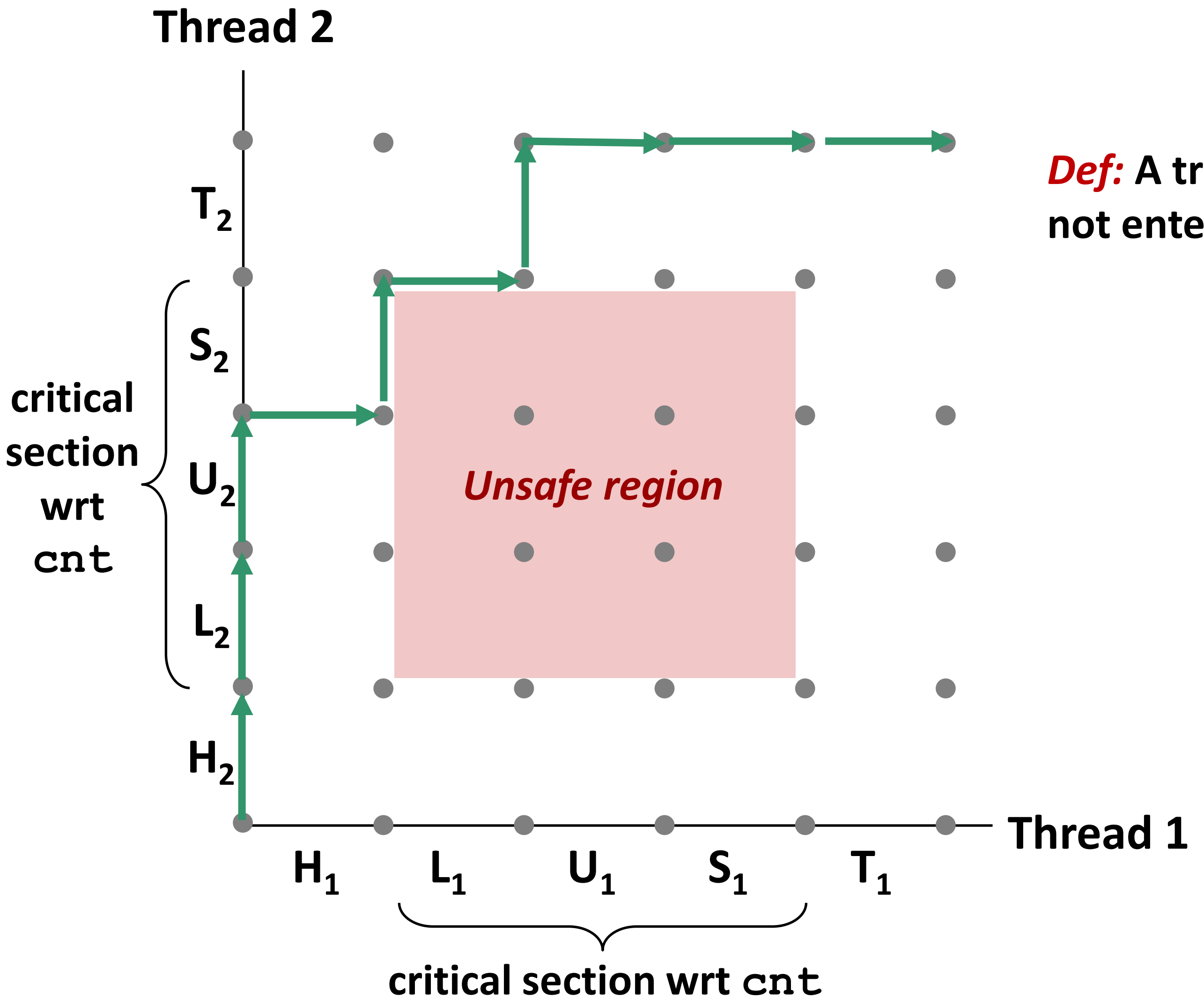
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



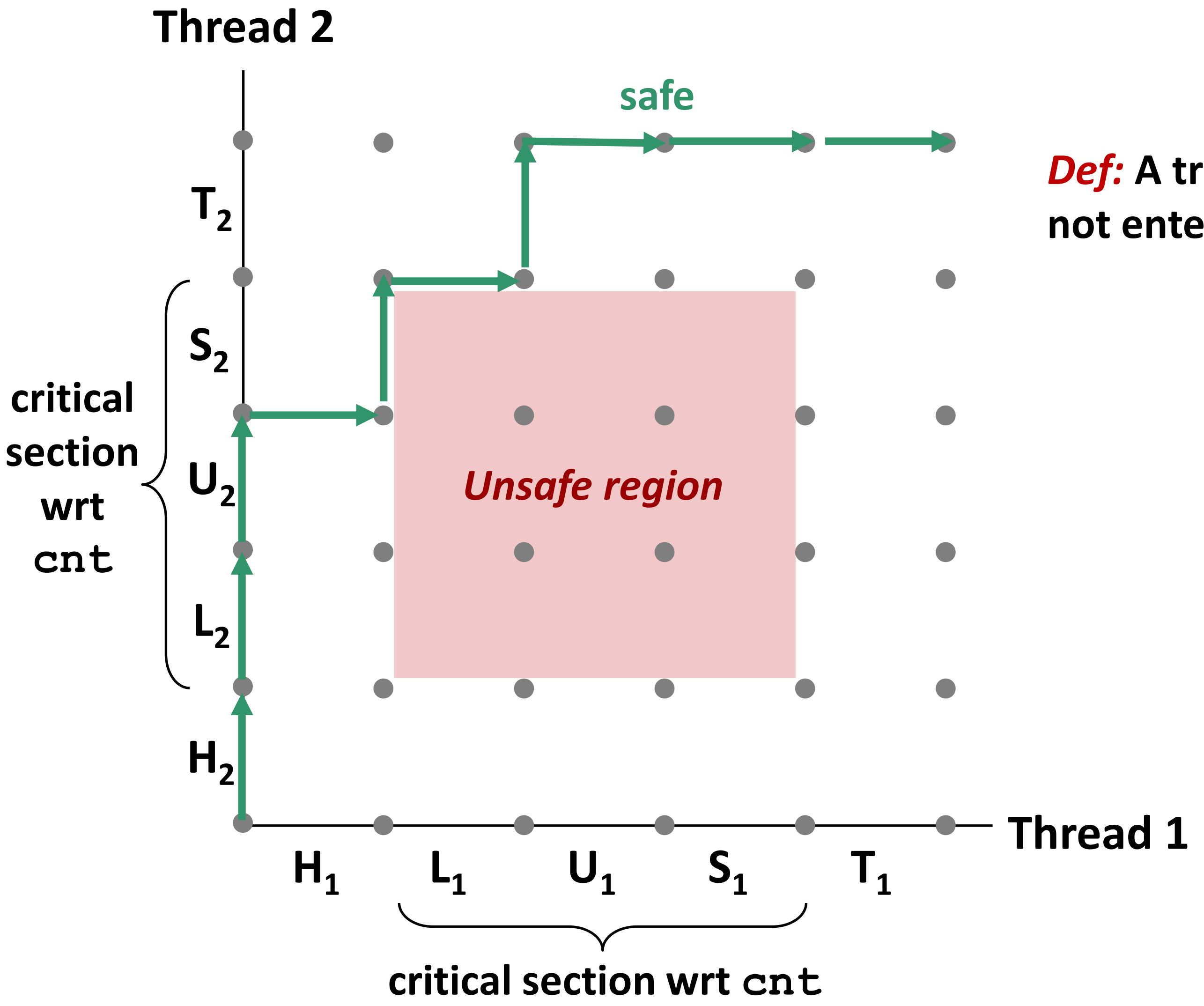
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



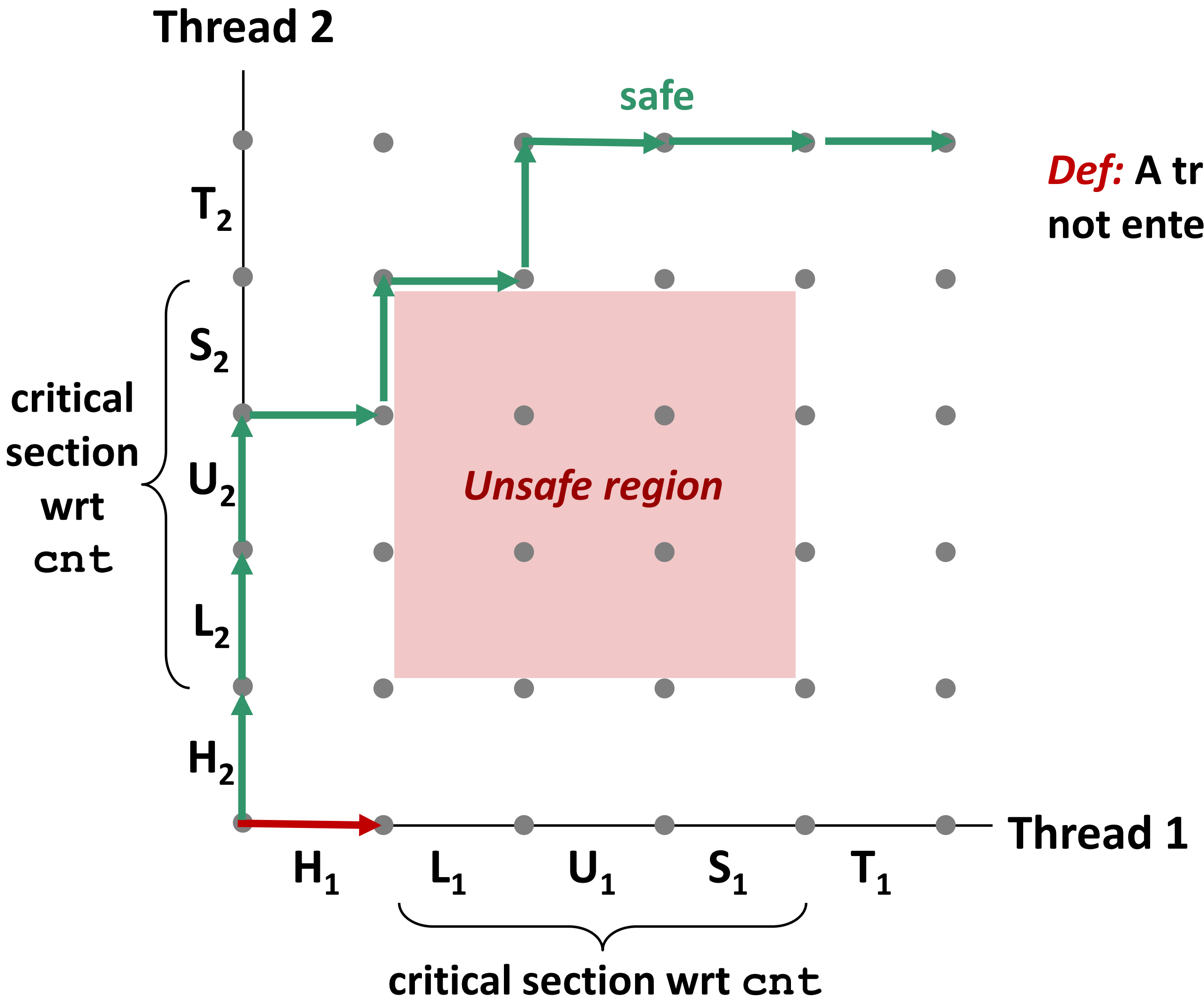
Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



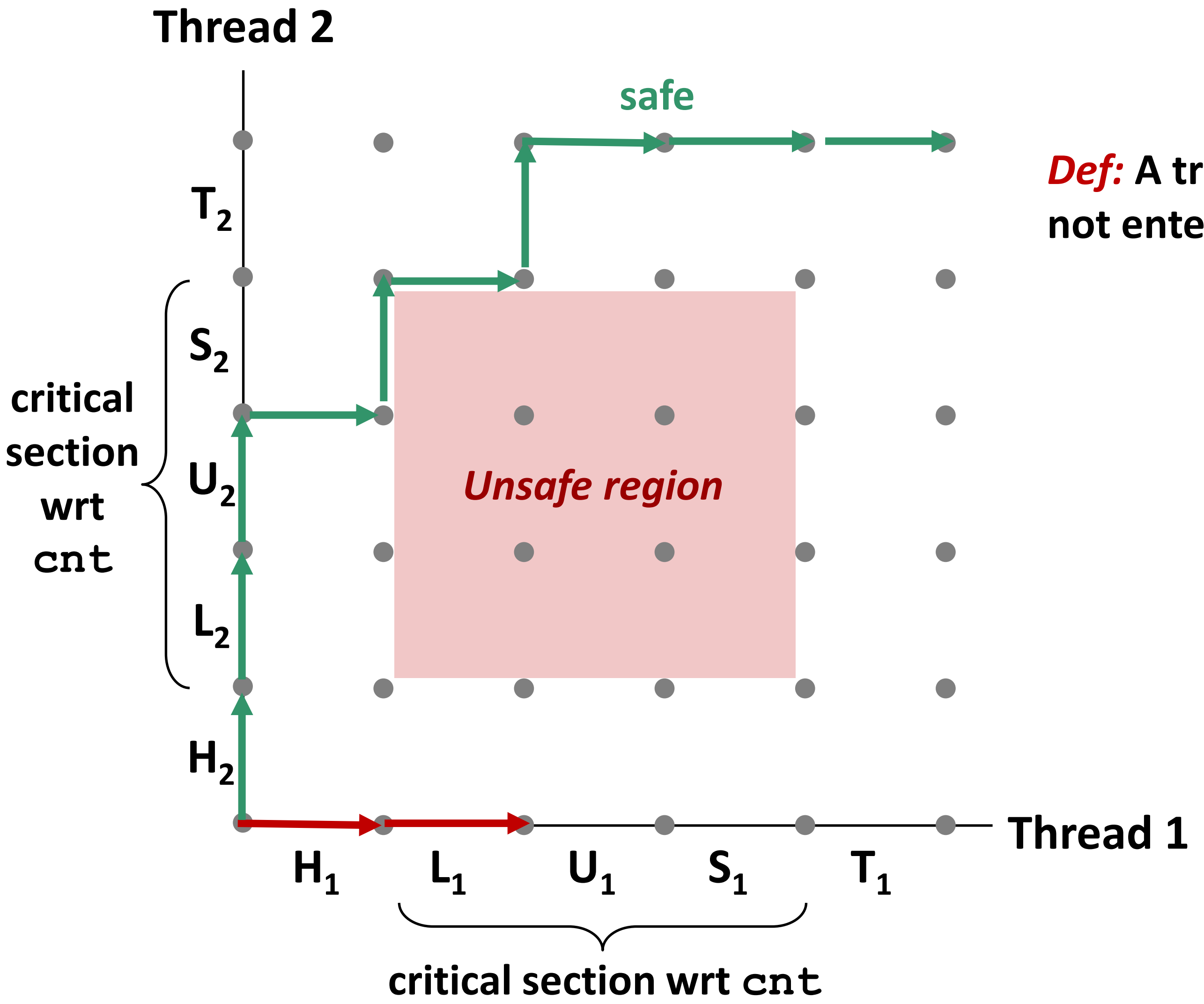
Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



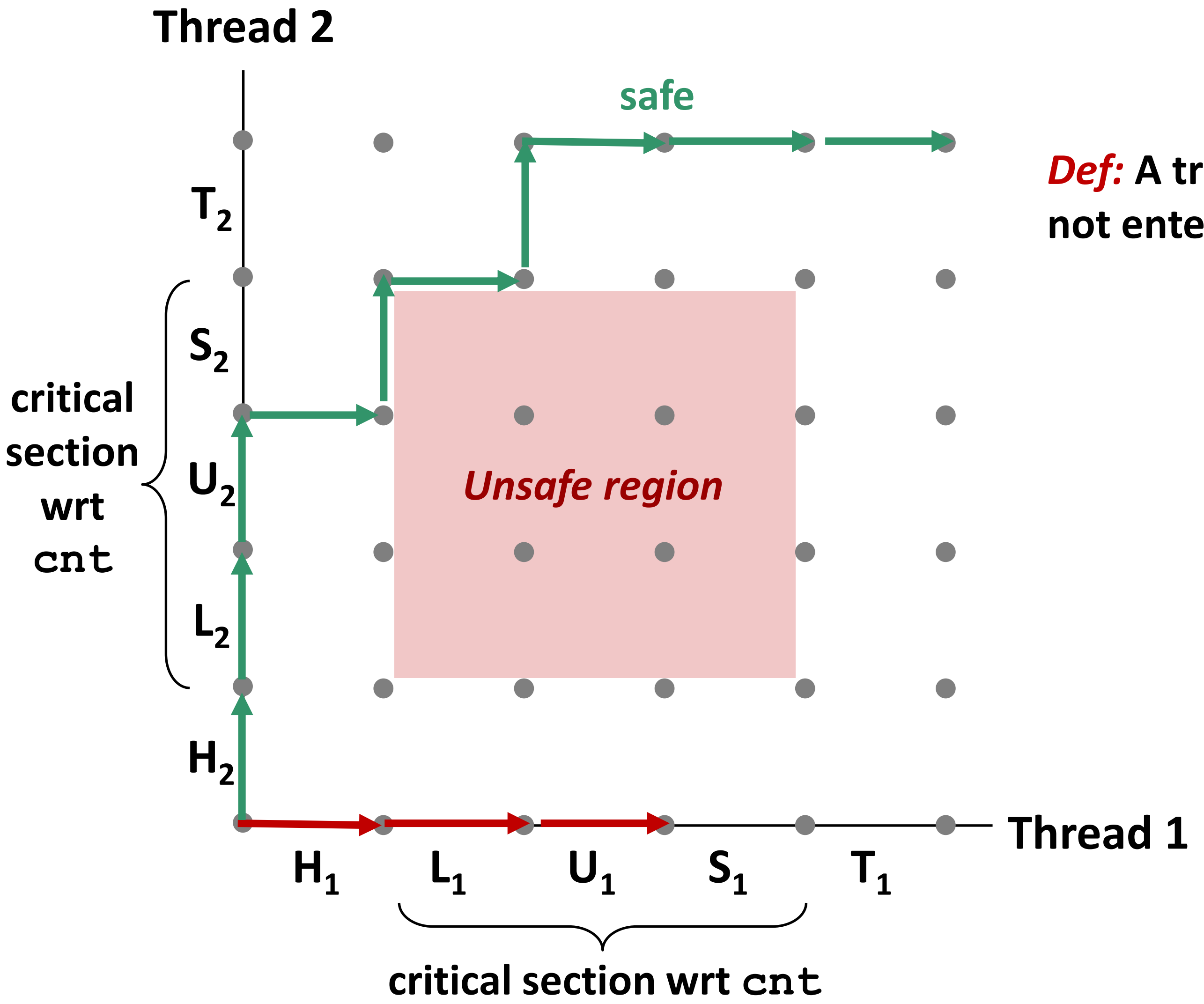
Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



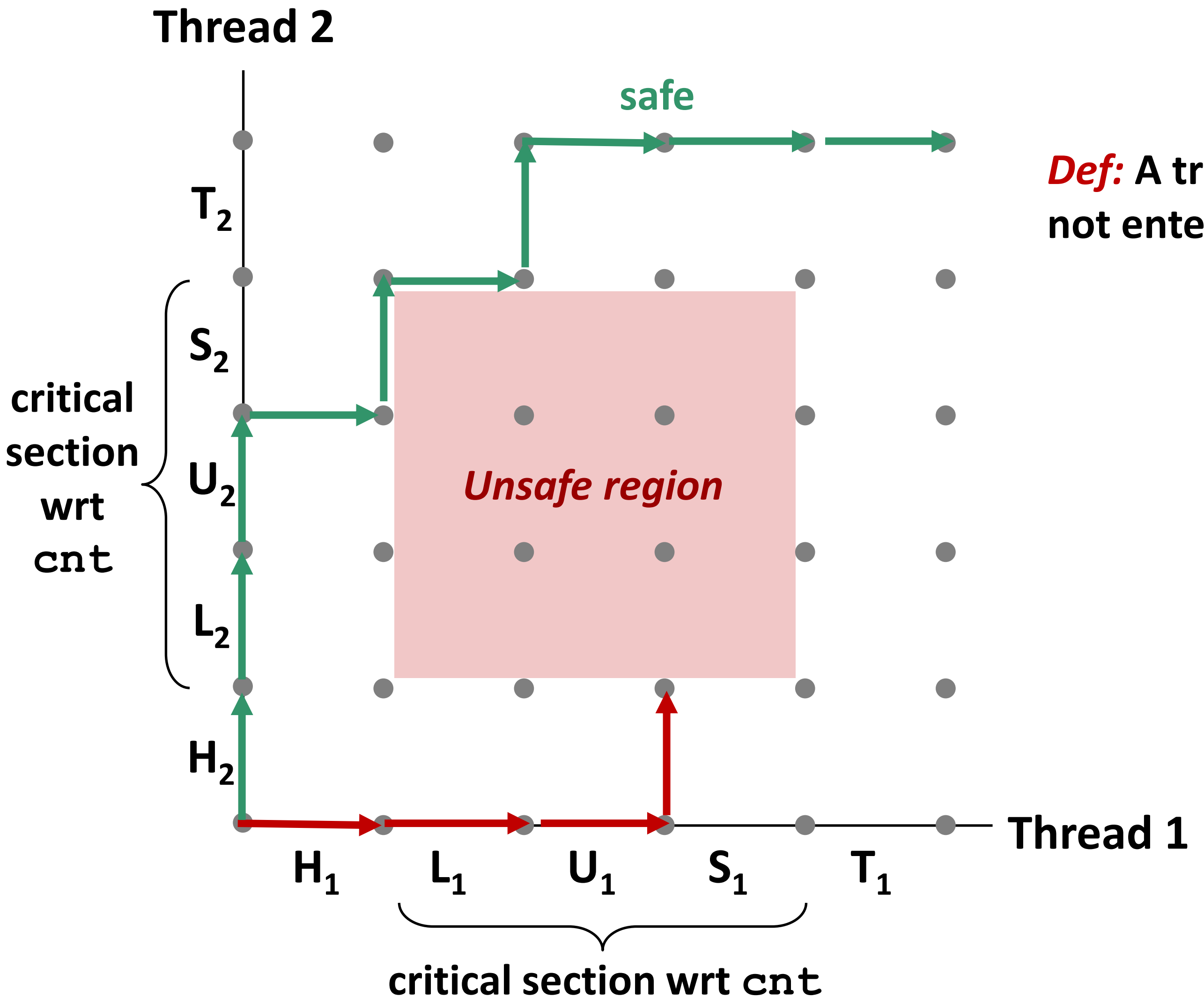
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



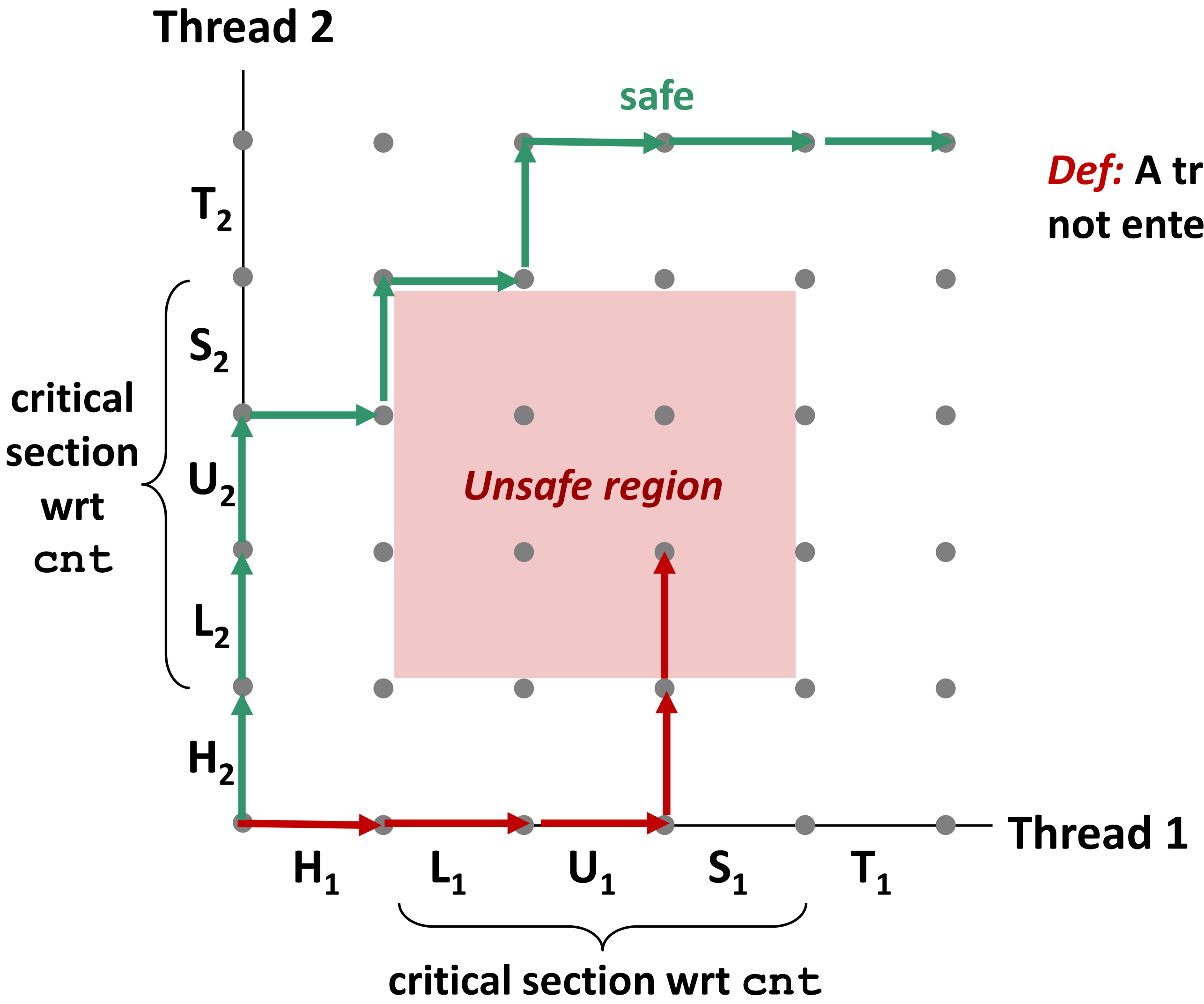
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



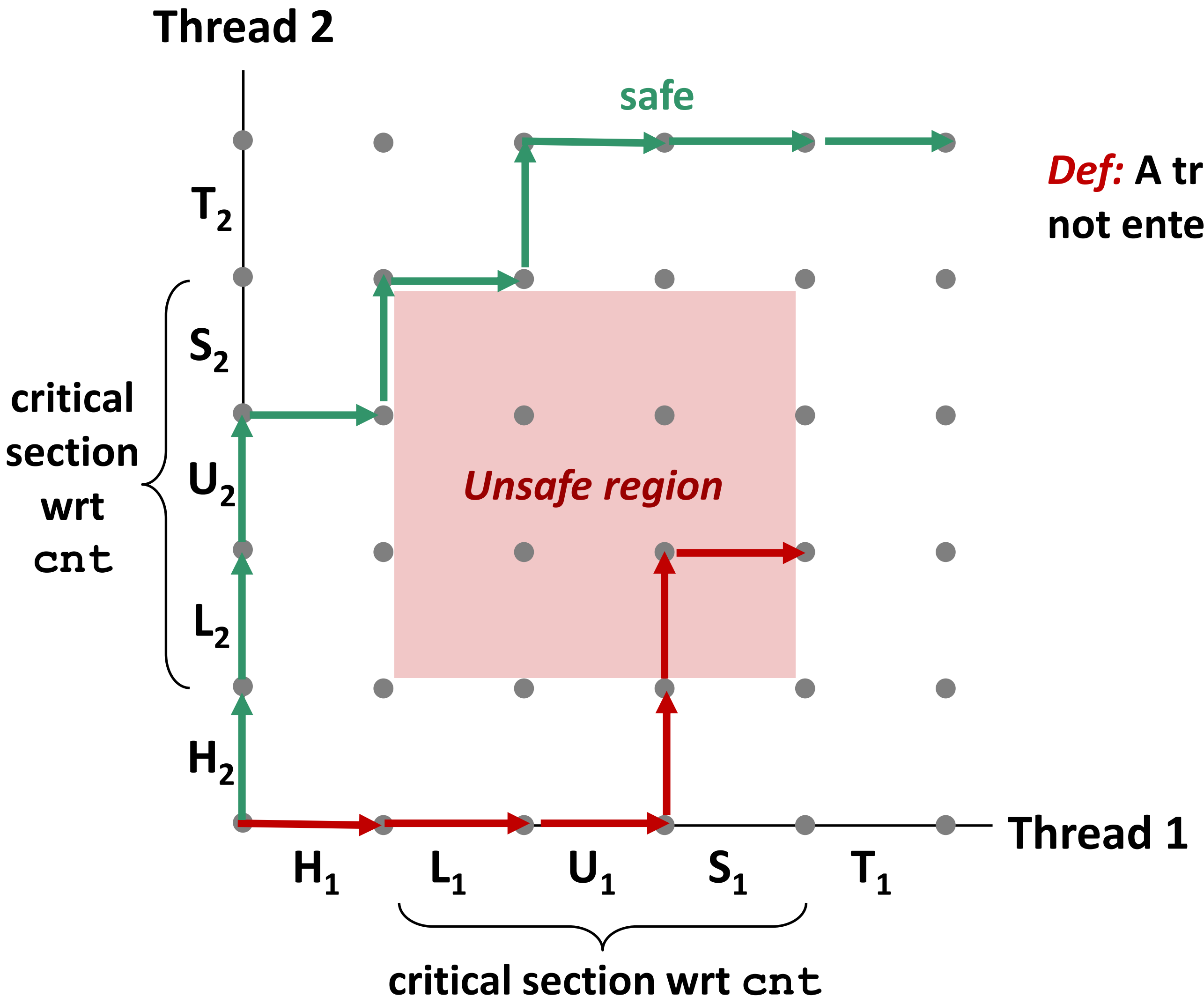
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



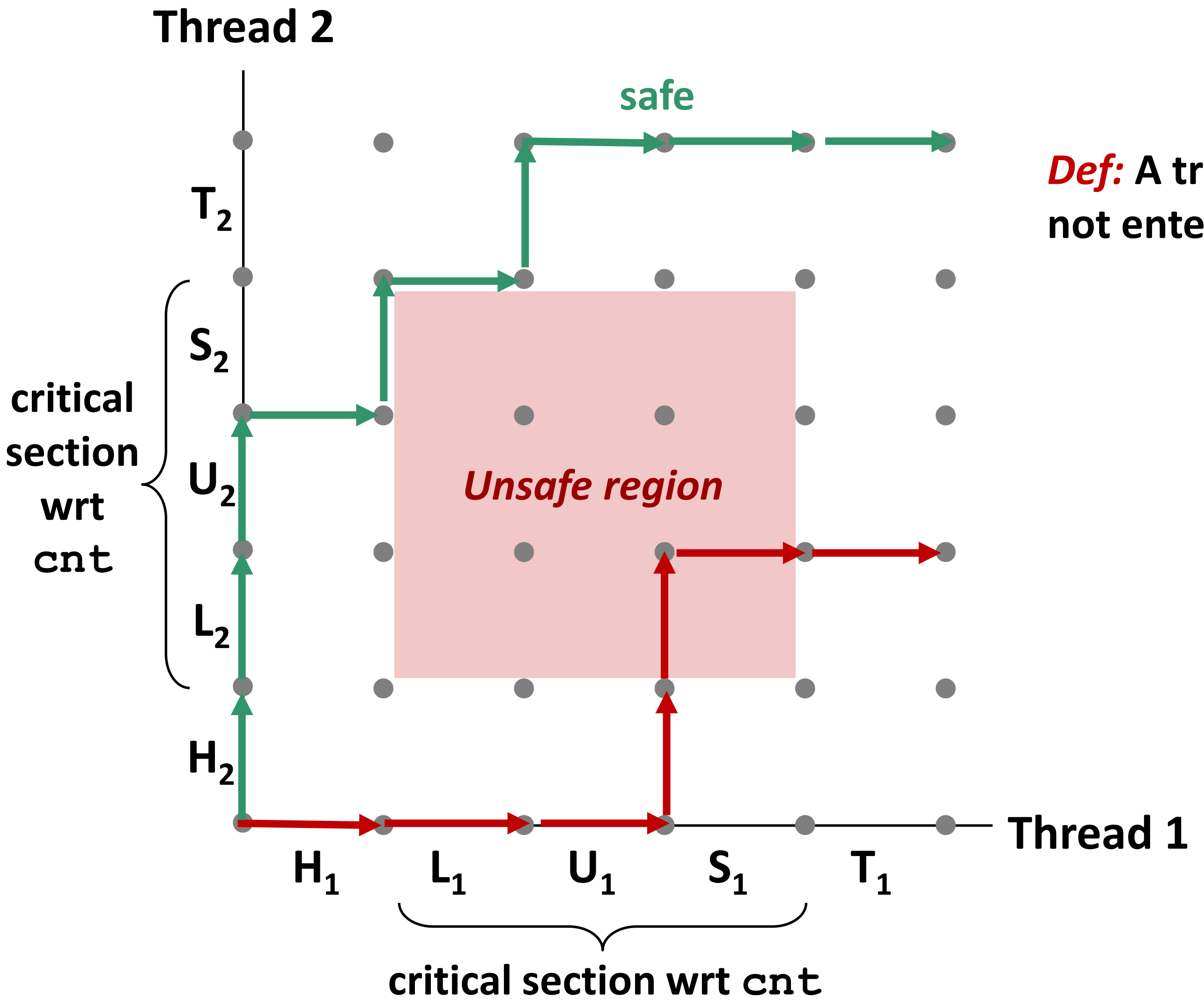
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



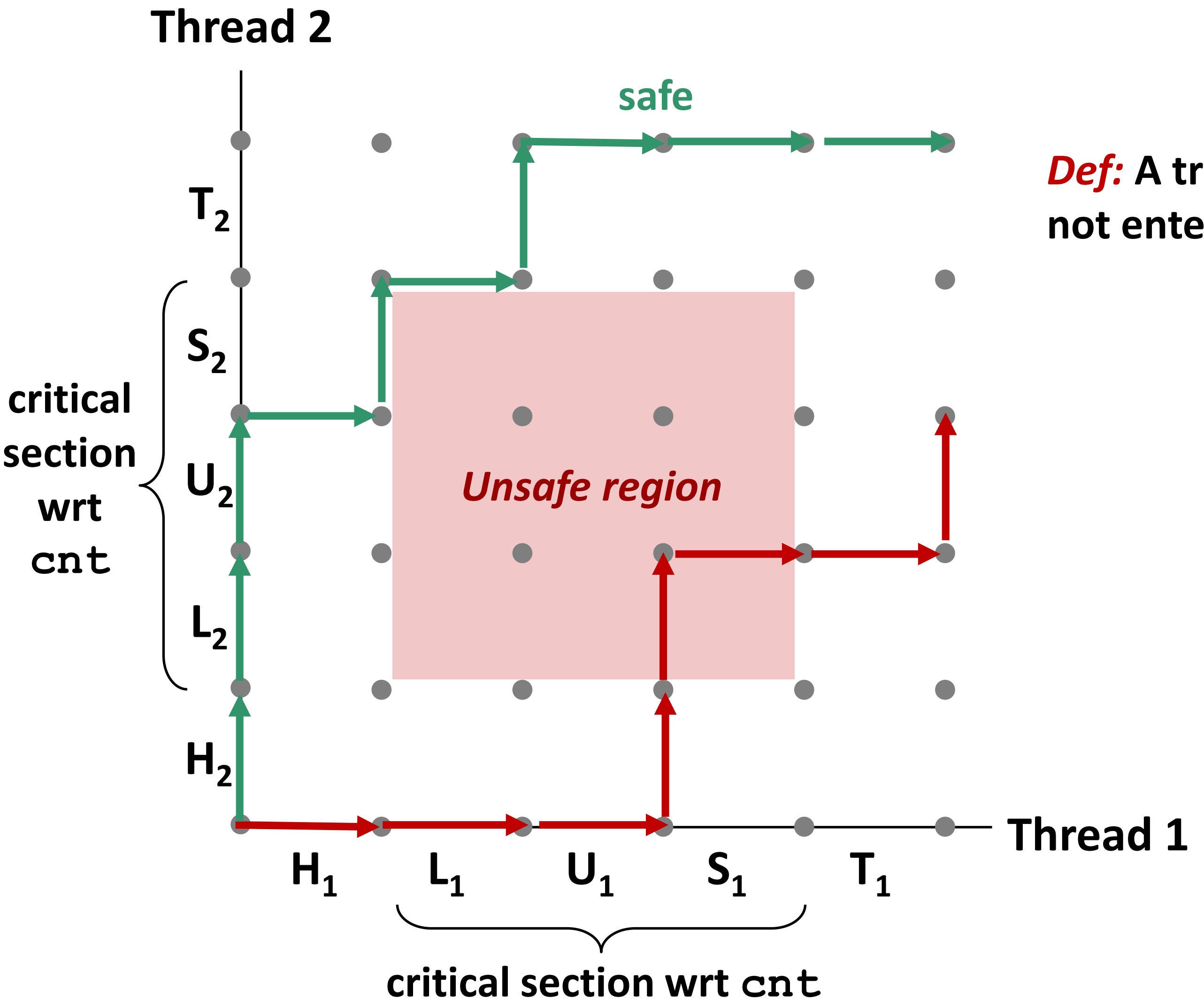
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



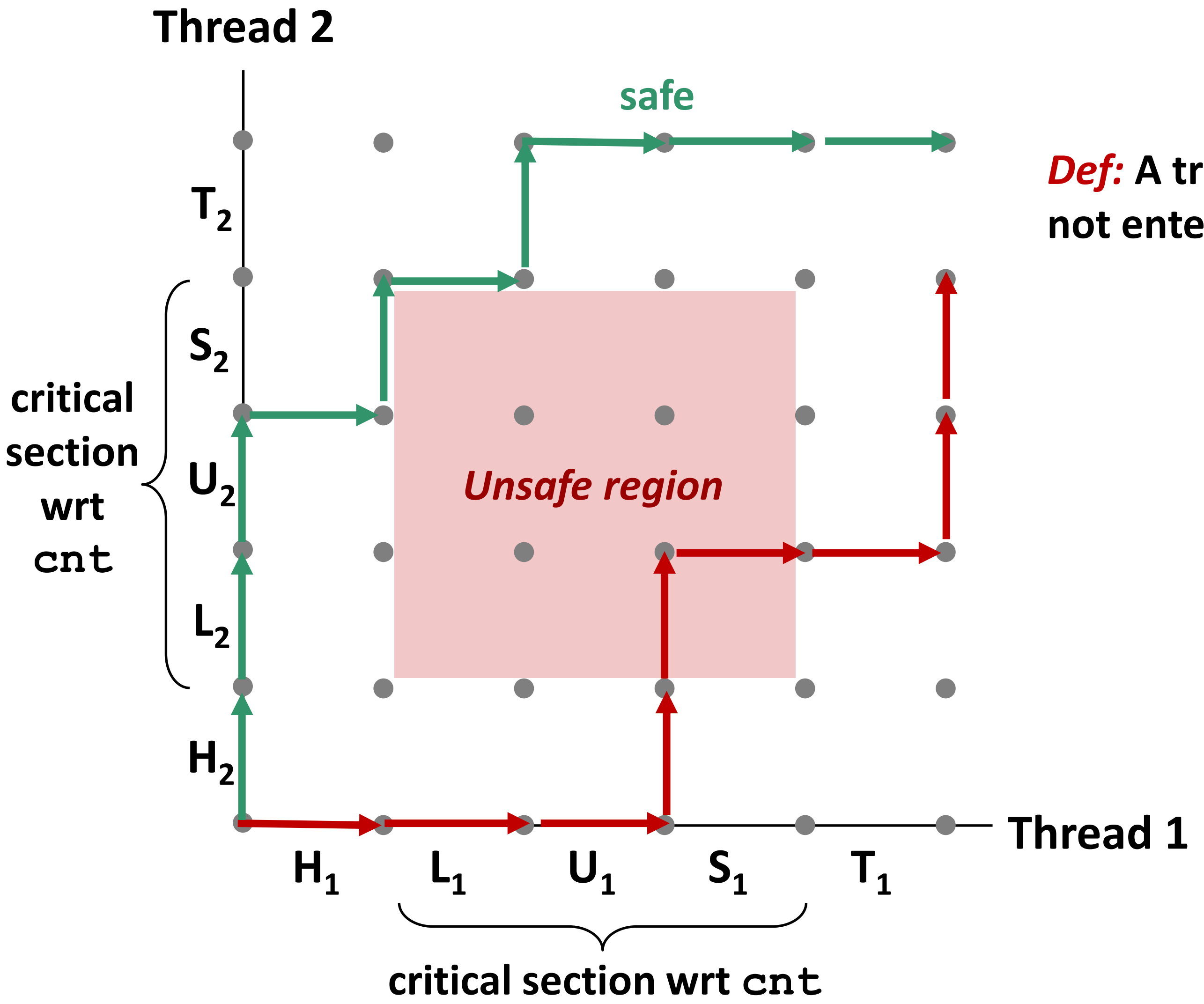
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



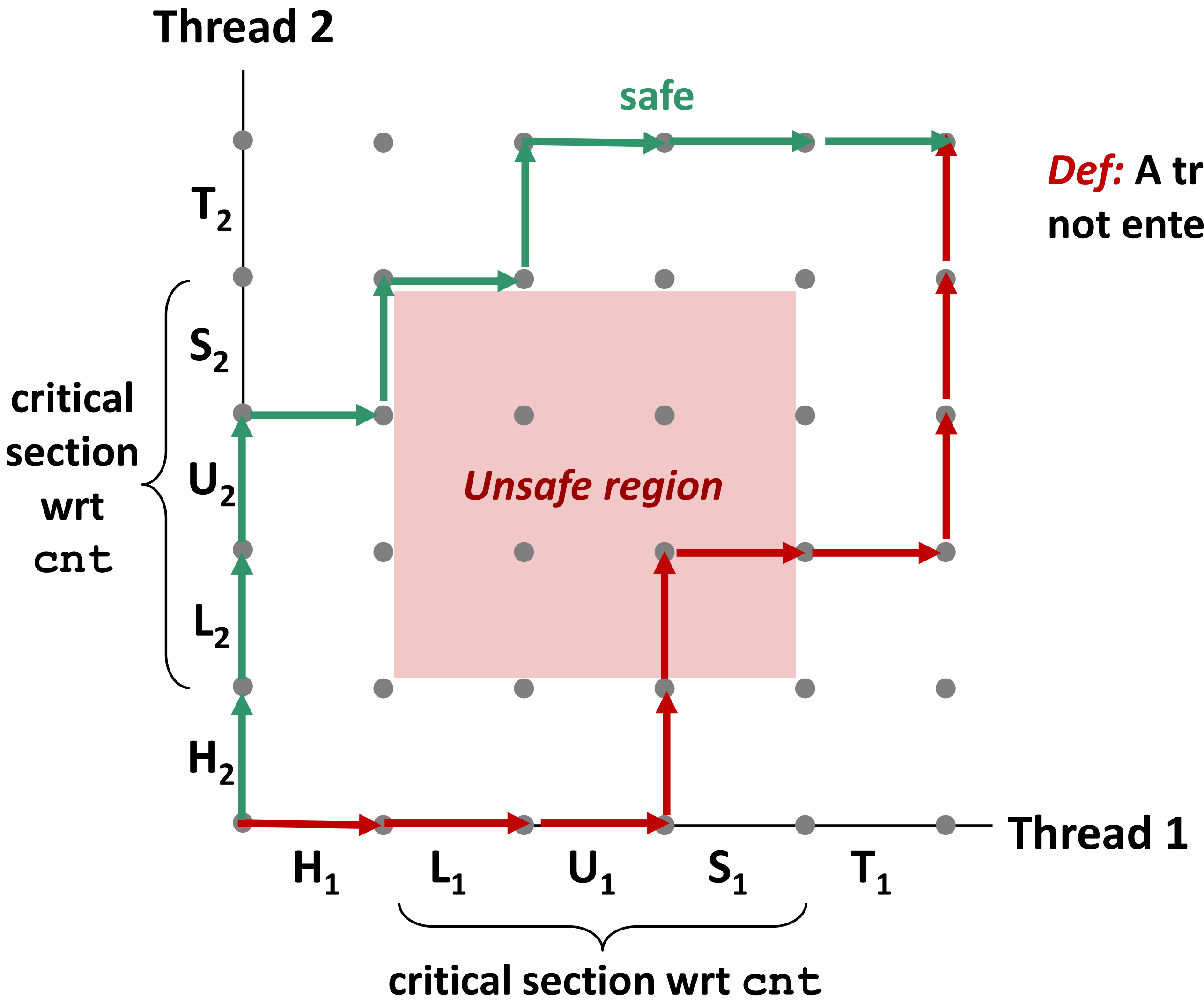
Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



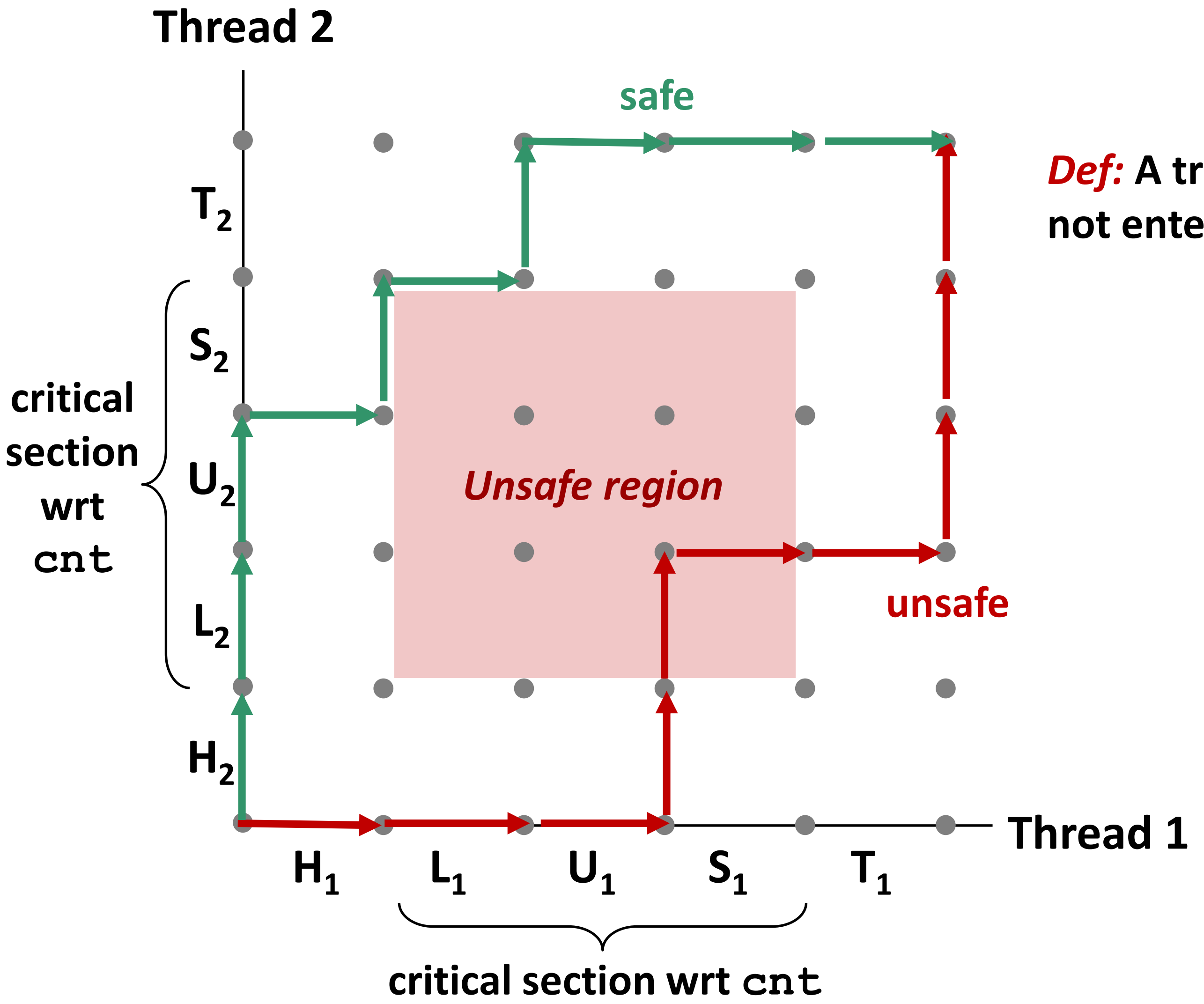
Def: A trajectory is *safe* iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



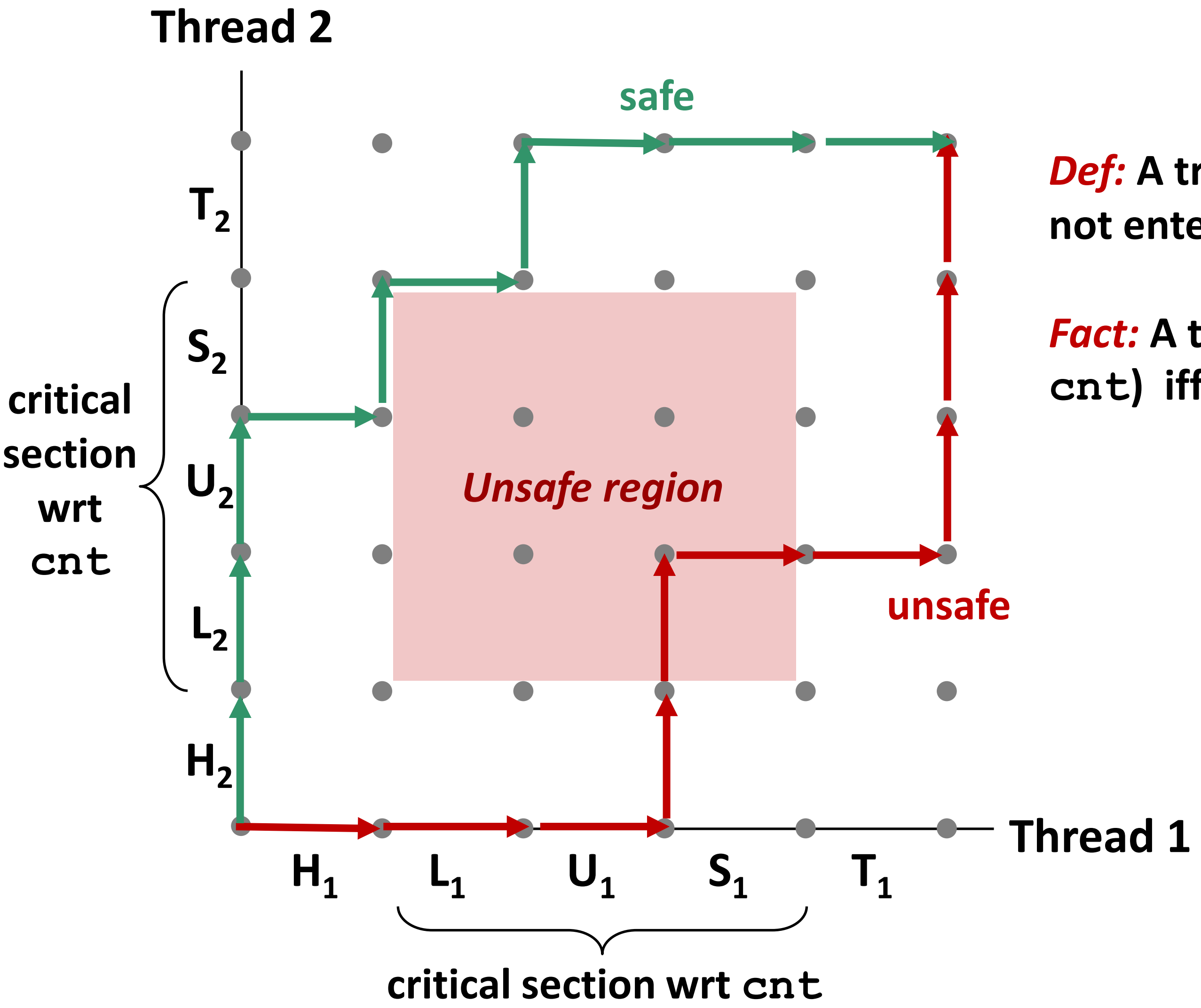
Def: A trajectory is ***safe*** iff it does not enter any unsafe region

Critical Sections and Unsafe Regions



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

Critical Sections and Unsafe Regions



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

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

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: Semaphores (Edsger Dijkstra, 1962)**

- **Other approaches (out of our scope)**
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)
  - C++11 atomic variables