# Lecture 19: Threads

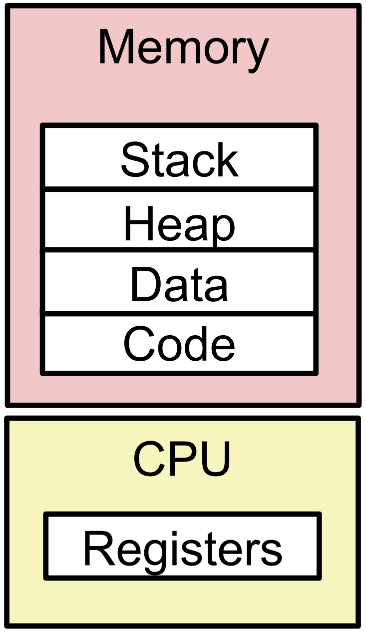
CS 105

April 3, 2019

# Processes

- Definition: A ***program*** is a file containing code + data that describes a computation
- Definition: A ***process*** is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*



# Traditional View of a Process

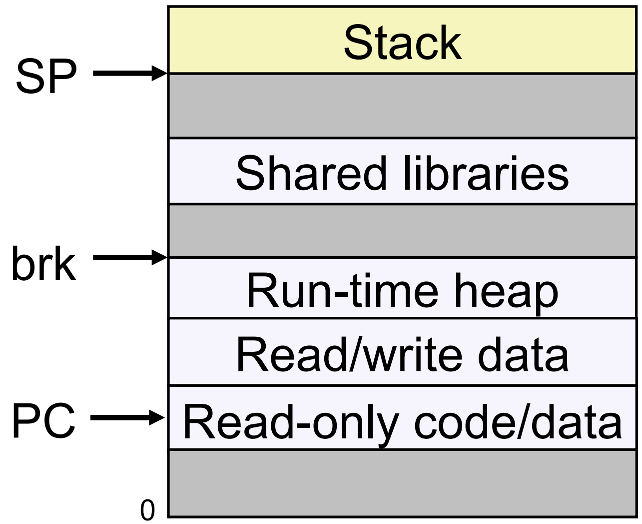
- Process = process context + code, data, and stack

**Process context**

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

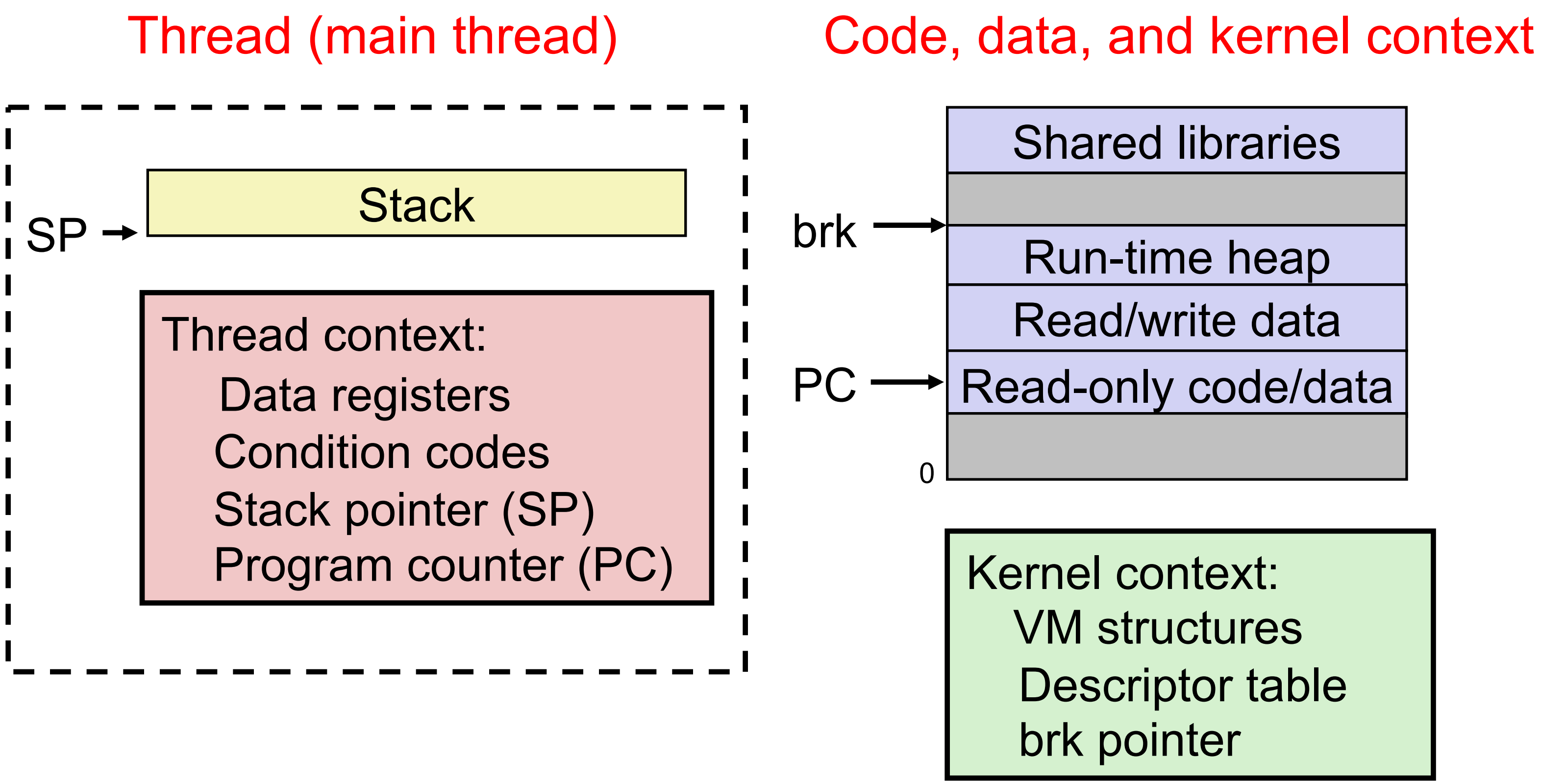
Kernel context:
- VM structures
- Descriptor table
- brk pointer

**Code, data, and stack**



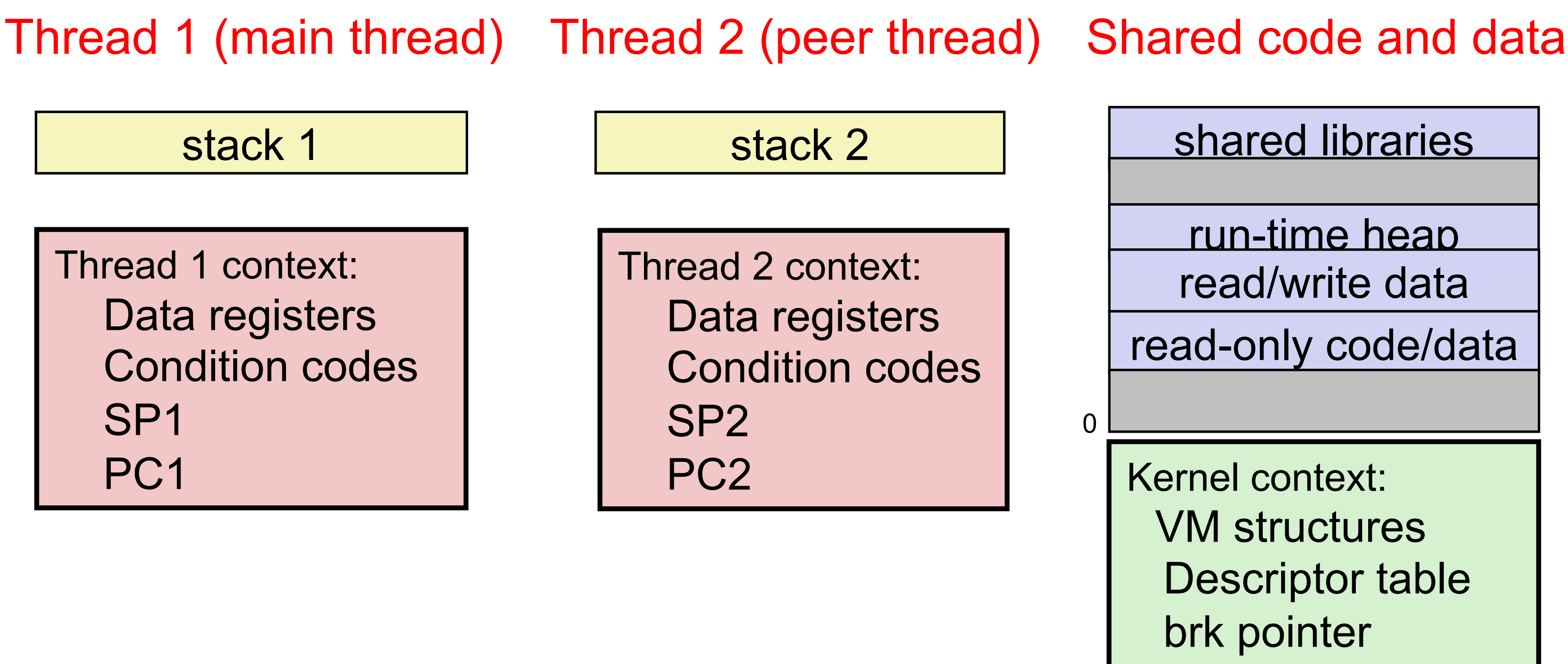
# Alternate View of a Process

- Process = 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 has its own stack for local variables
  - Each thread has its own thread id (TID)
  - Each thread shares the same code, data, and kernel context

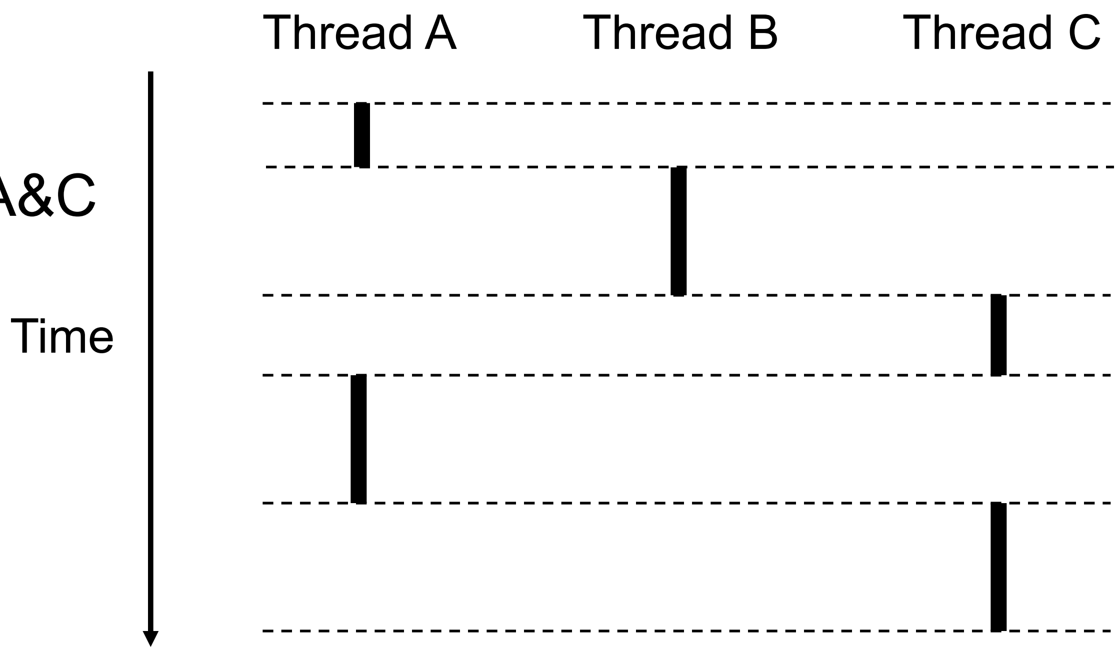


# Threads vs. Processes

- 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

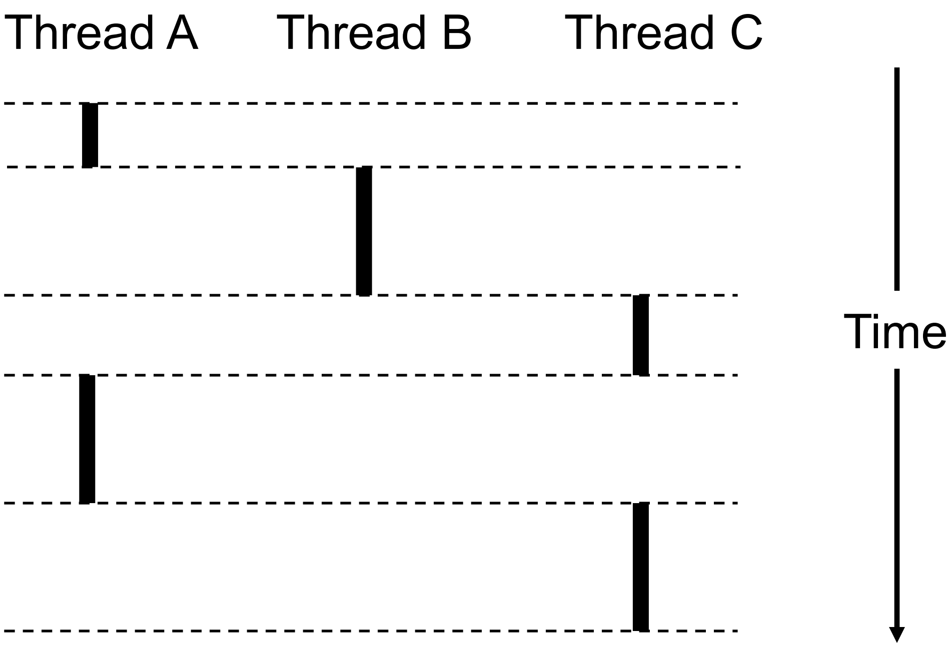
# Concurrent Threads

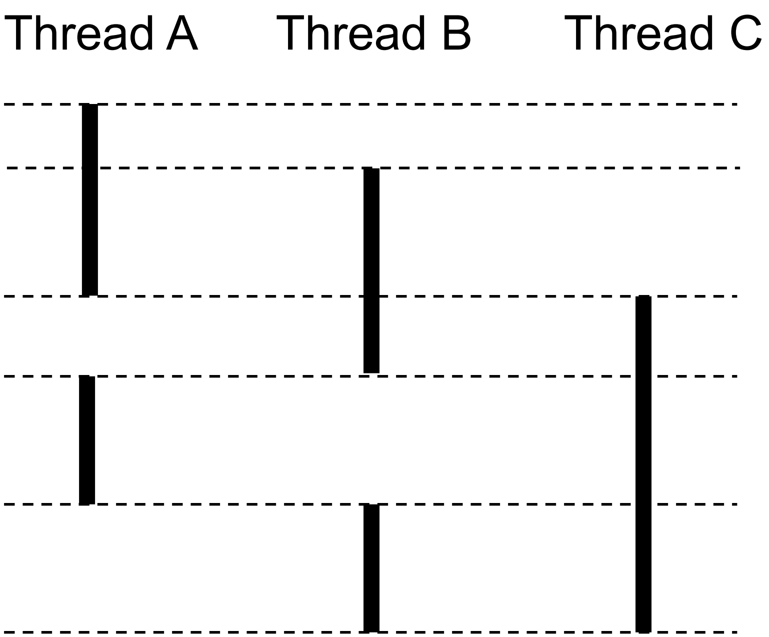
- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential
- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



# Concurrent Thread Execution

- Single Core Processor
  - Simulate parallelism by time slicing
- Multi-Core Processor
  - Can have true parallelism




Run 3 threads on 2 cores

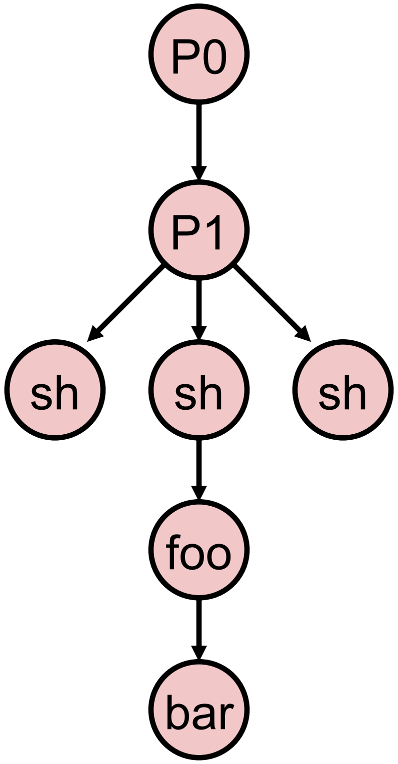
# Threads vs. Processes

- 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
- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

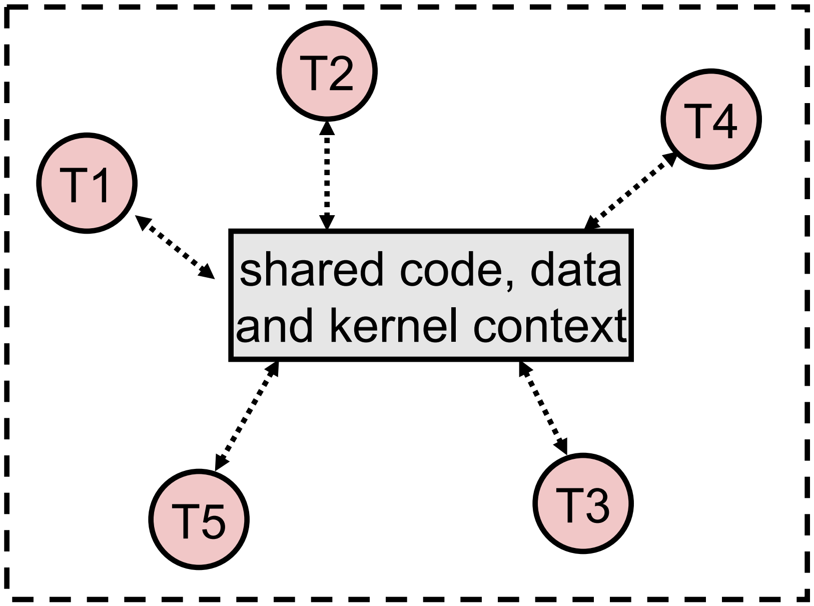
# Logical View of Threads

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

Process hierarchy




Threads associated with process foo



# 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] , `RET` [terminates current thread]

# The Pthreads "hello, world" Program

```c
/*
 * 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);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

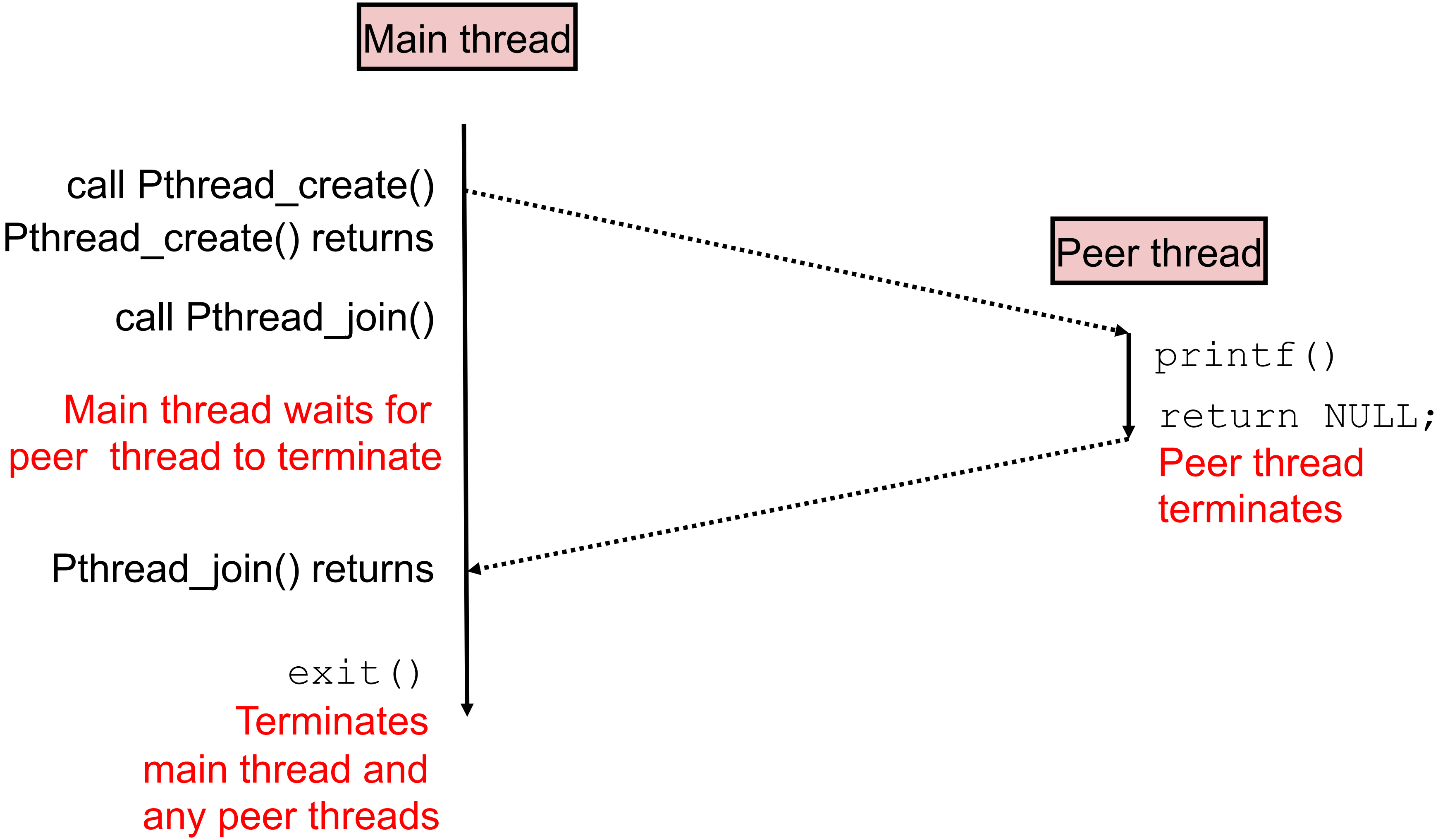
Thread arguments (void *p)

Return value (void **p)

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

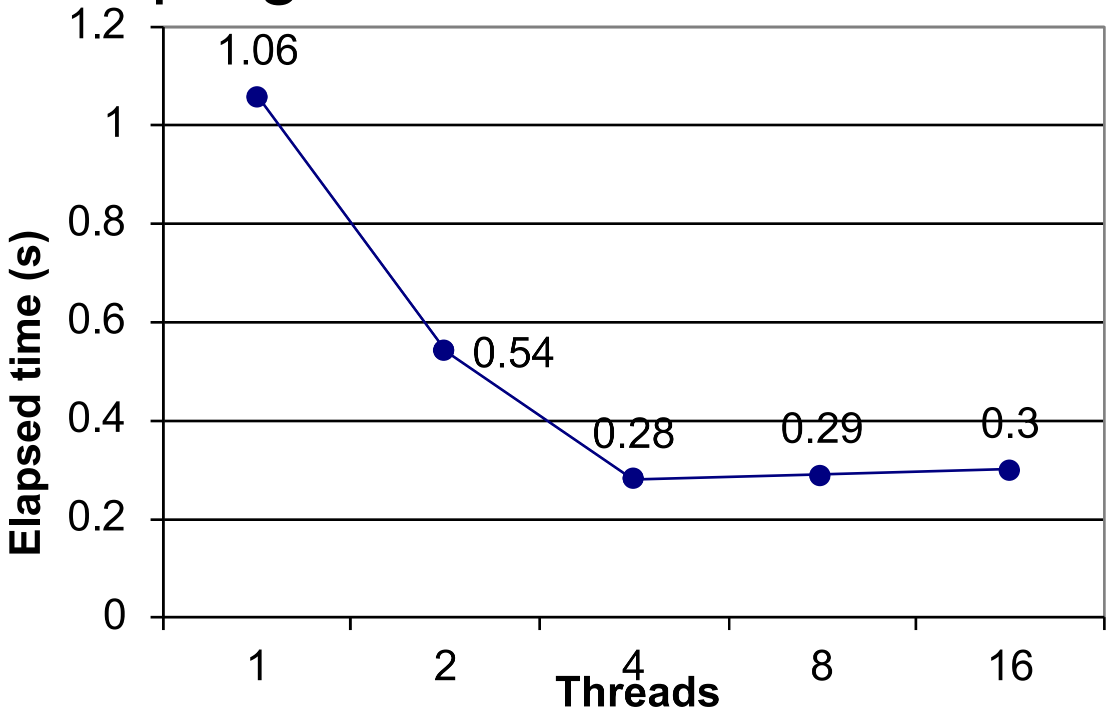
hello.c

# Execution of Threaded "hello, world"



# Using Threads for Parallelism

- on a multi-core system, the OS can schedule concurrent threads in parallel on multiple cores
- … so concurrent programs can run faster that sequential programs



# Shared Variables in Threaded 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 refer to 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 refer to each of these instances?

# 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*

# Example Program to Illustrate Sharing

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

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

```
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*

# 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

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

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

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

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

*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 |

- 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**

# Pros and Cons of Thread-Based Designs

+ Threads are more efficient than processes

+ Easy to share data structures between threads

  e.g., logging information, file cache

- Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low. But nonzero!

# 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;
}
```

```
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

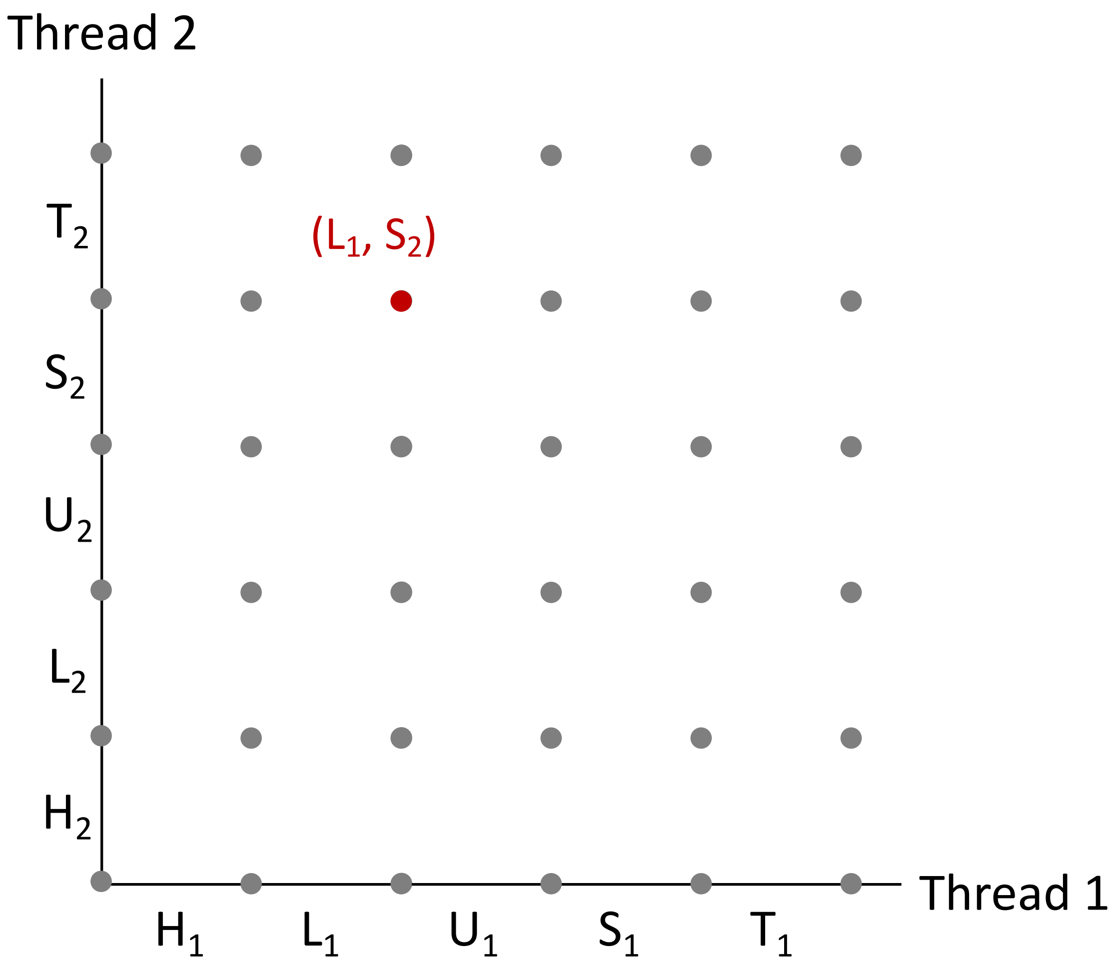
$L_i$ : Load `cnt`
$U_i$ : Update `cnt`
$S_i$ : Store `cnt`

$T_i$ : Tail

# Safe Schedules

- A schedule of instructions is safe if the resulting concurrent computation returns the correct answer
- Assume two threads executing routine `thread`. Which of the following schedules are safe?
  - $H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1$
  - $H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2$
  - $H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2$

# Progress Graphs



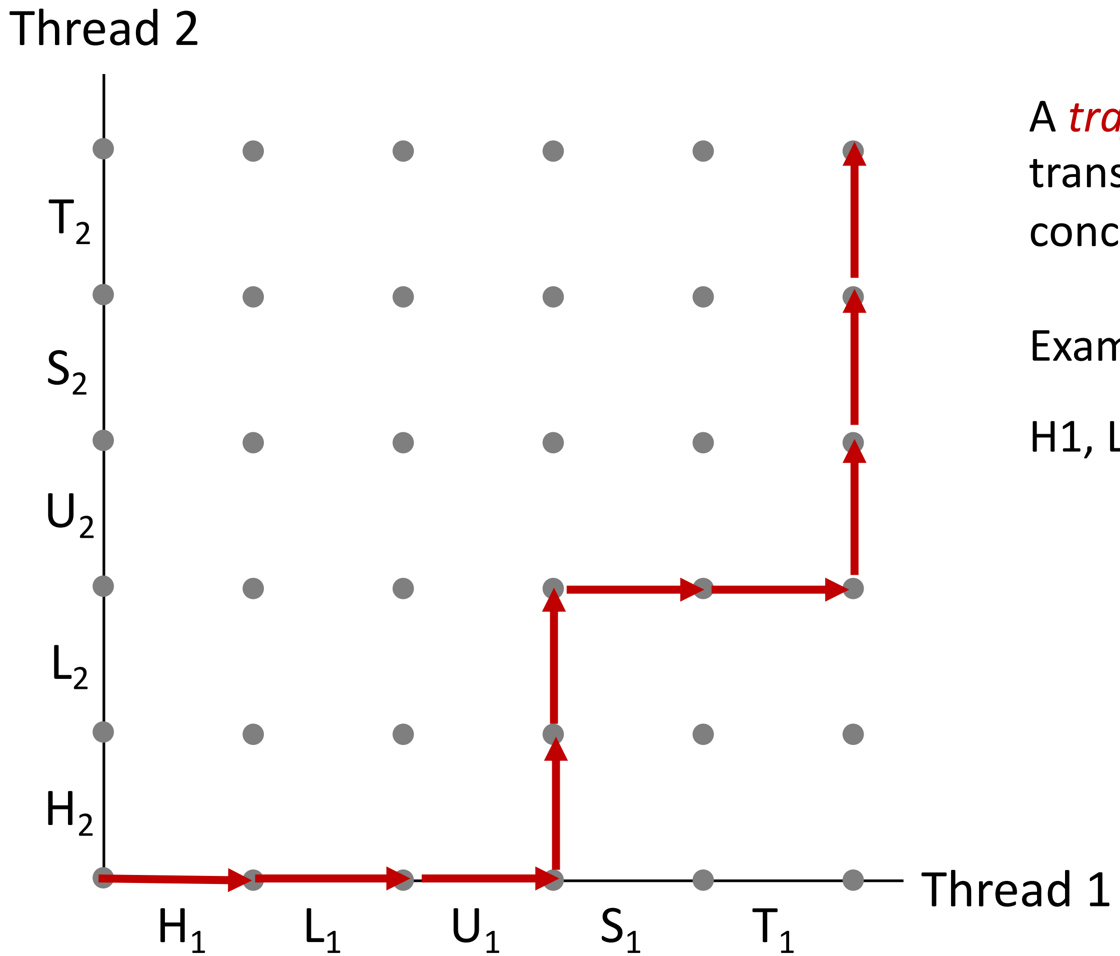

A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

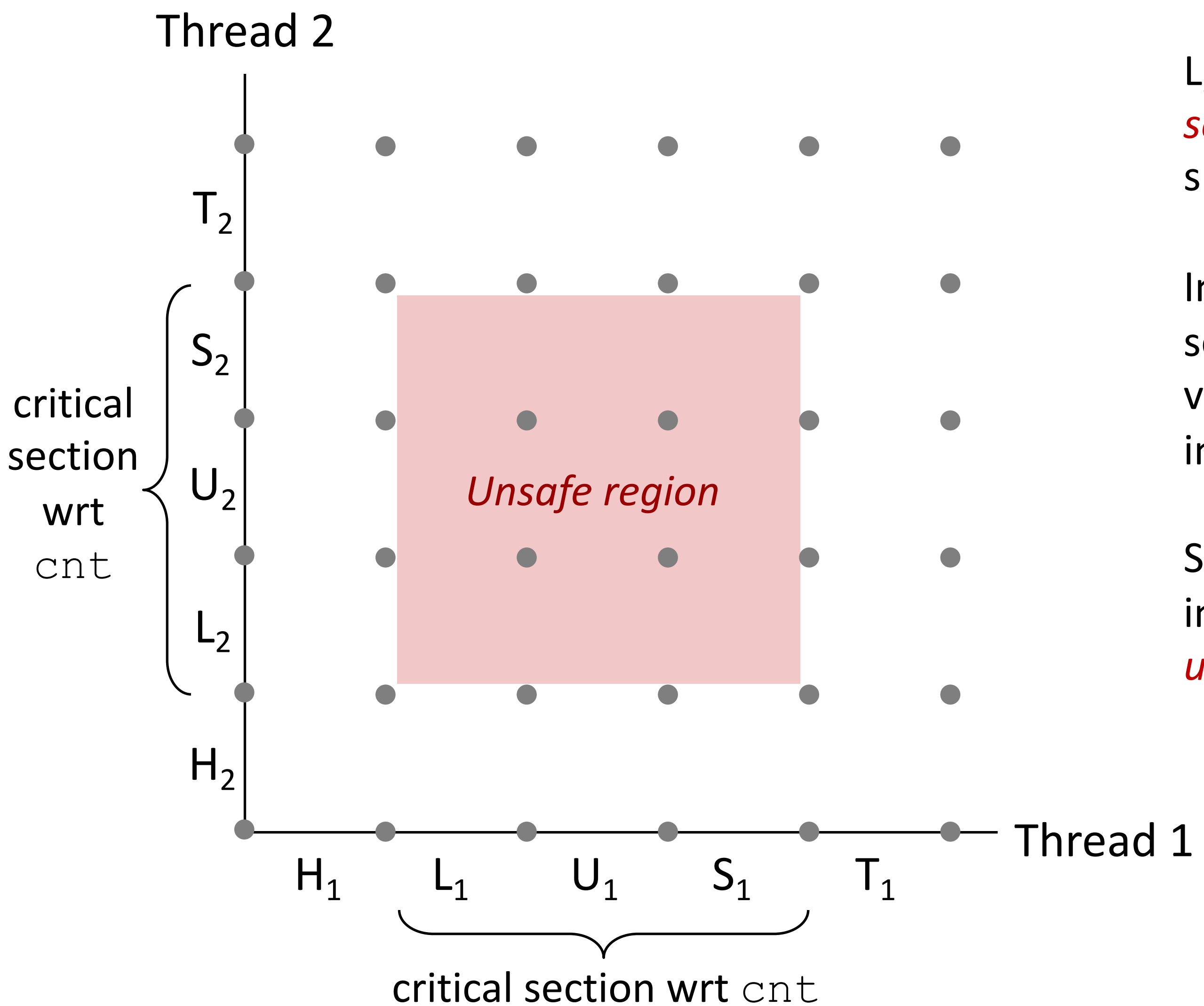



A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

# Critical Sections and Unsafe Regions

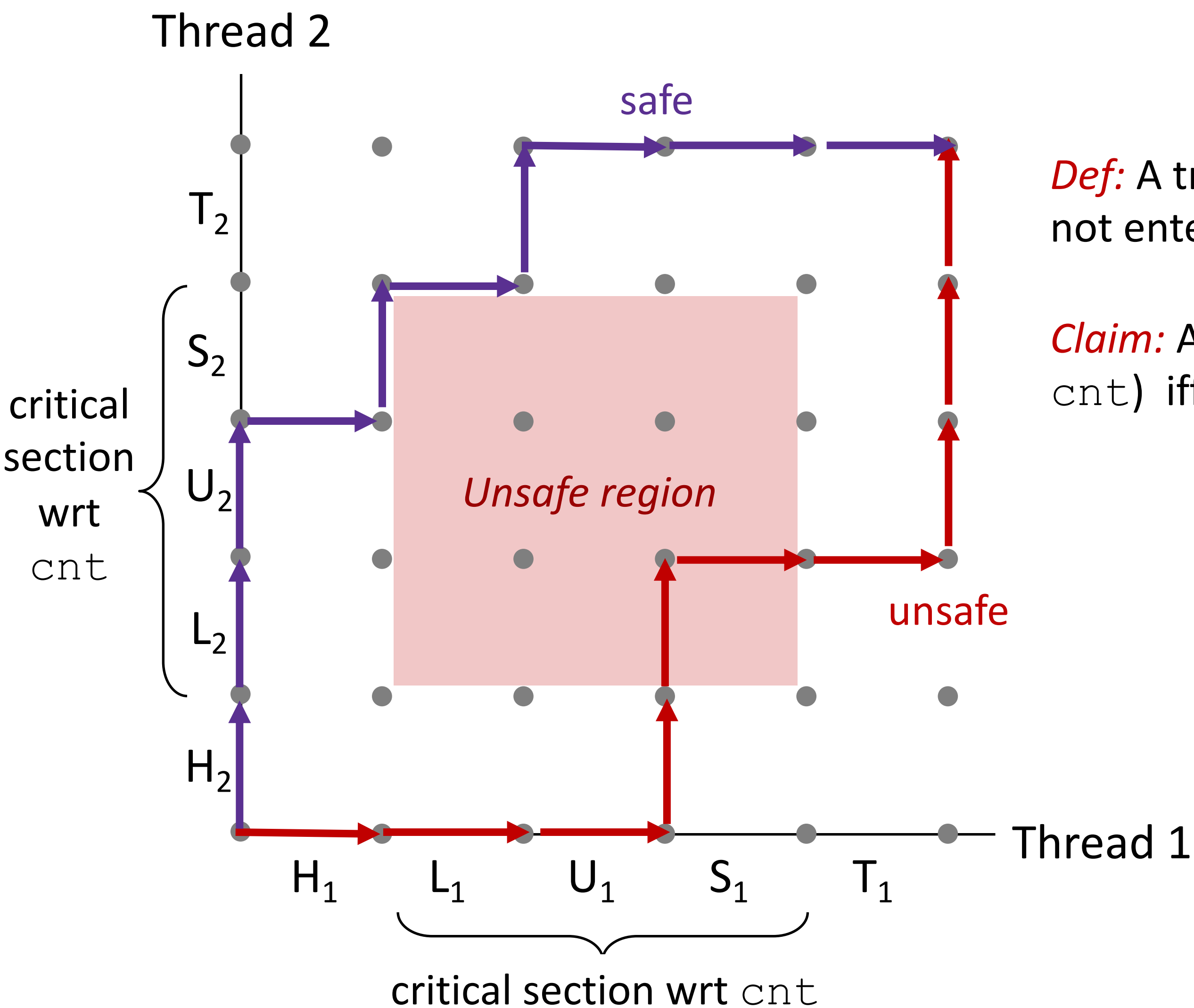



L, U, and S form a *critical section* with respect to the shared variable `cnt`

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

# 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.
- Possible solutions:
  - Locks
  - Semaphores
  - Condition variables