

# Concurrent Programming

HPPS

Troels Henriksen


**Based on slides by:**

Randal E. Bryant and David R. O'Hallaron

# Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

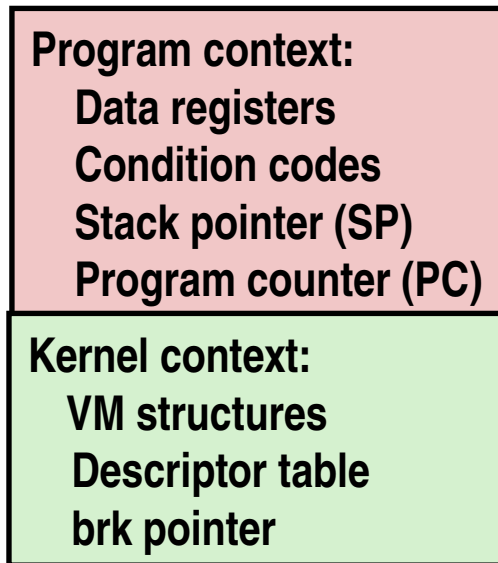
# Concurrent Programming is Hard!

- **Classical problem classes of concurrent programs:**
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness:*** external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line
- **Many aspects of concurrent programming are beyond the scope of our course..**
  - but, not all 
  - We'll cover some of these aspects in the next few lectures.

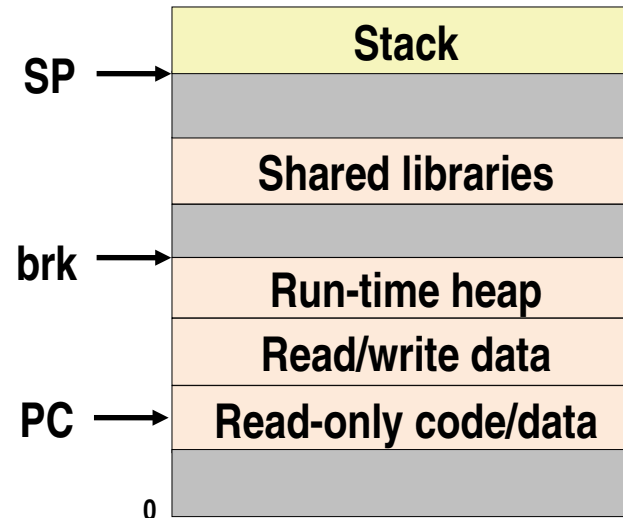
# Traditional View of a Process

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

## Process context



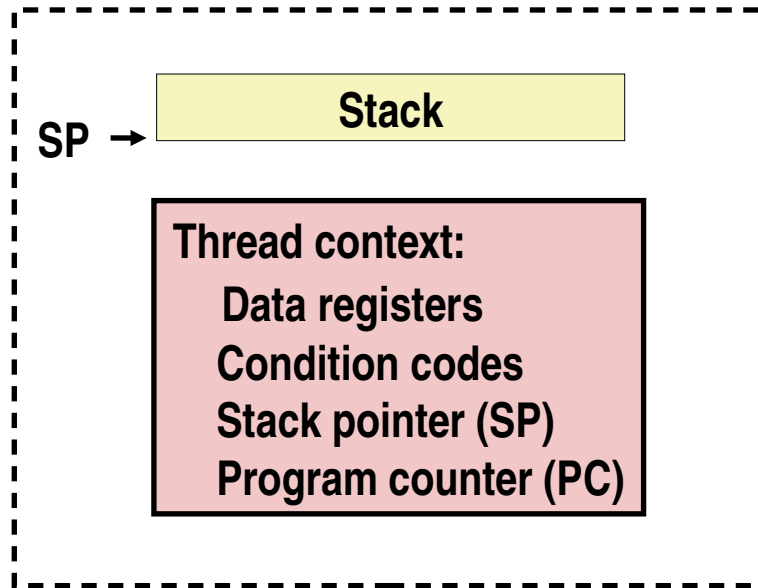
## Code, data, and stack



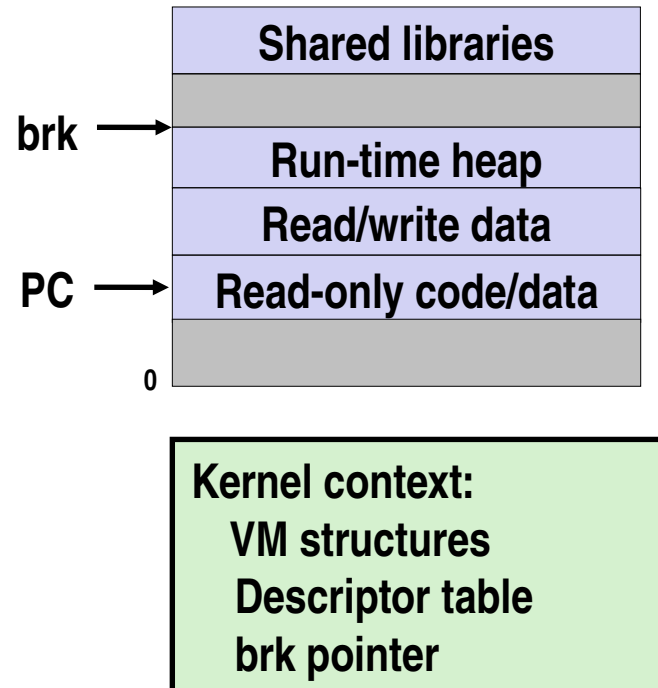
# Alternate View of a Process

- 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)

**Thread 1 (main thread)**

**Thread 2 (peer thread)**

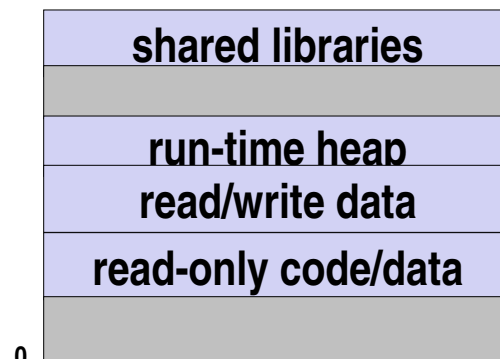
**Shared code and data**

**stack 1**

**stack 2**

Thread 1 context:  
 Data registers  
 Condition codes  
 SP1  
 PC1

Thread 2 context:  
 Data registers  
 Condition codes  
 SP2  
 PC2

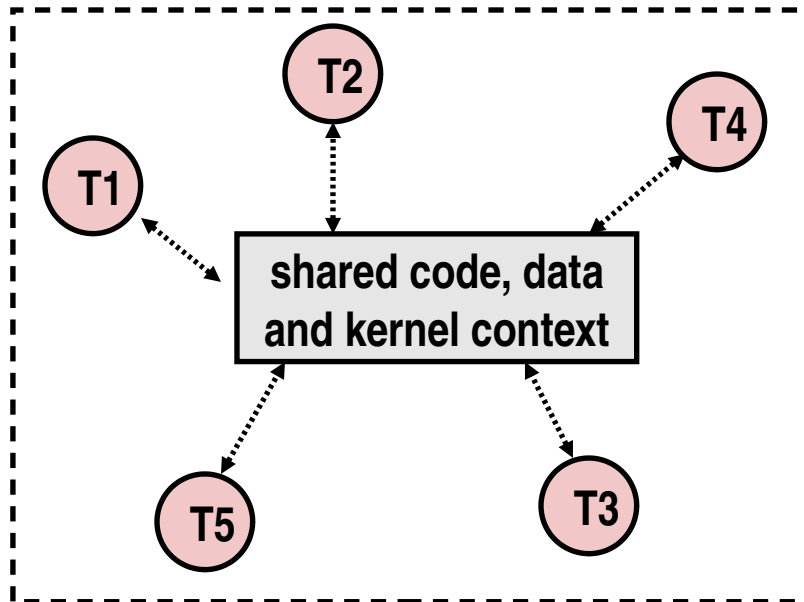


Kernel context:  
 VM structures  
 Descriptor table  
 brk pointer

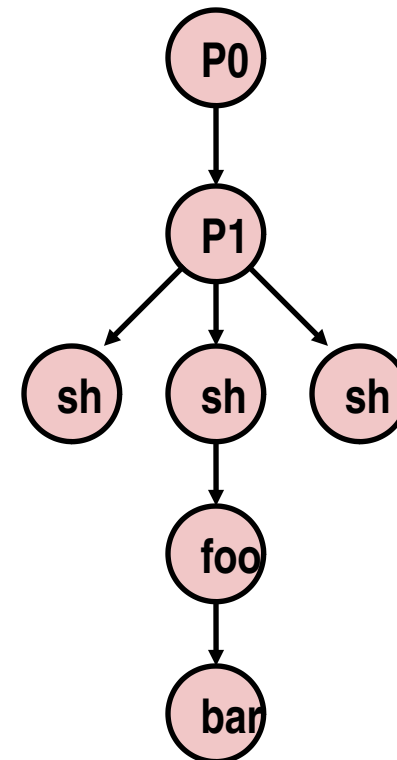
# Logical View of Threads

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

Threads associated with process foo



Process hierarchy



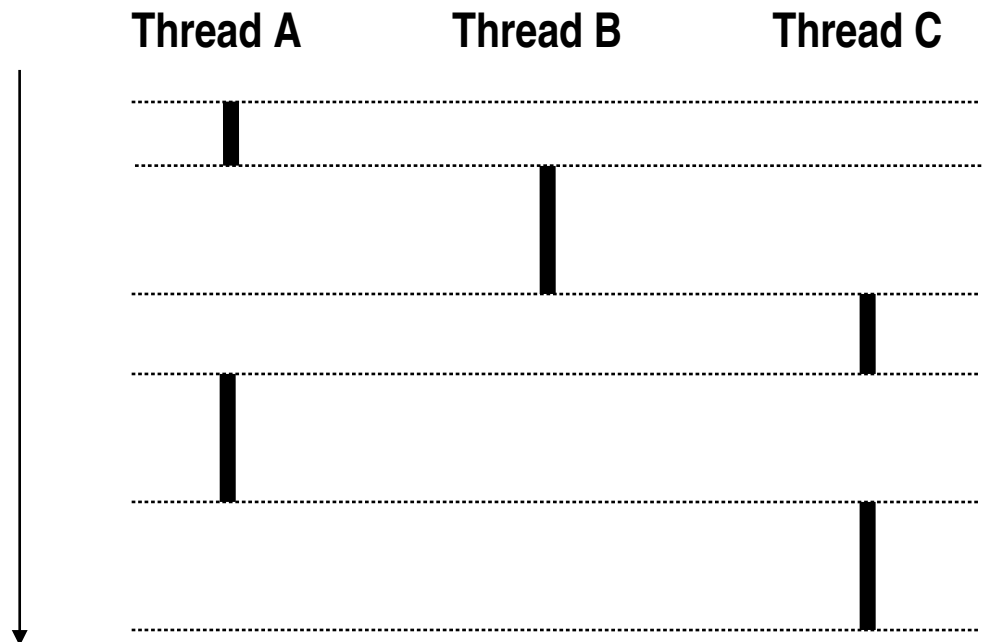
# Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

- **Examples:**

- Concurrent: A & B, A&C
- Sequential: B & C

Time

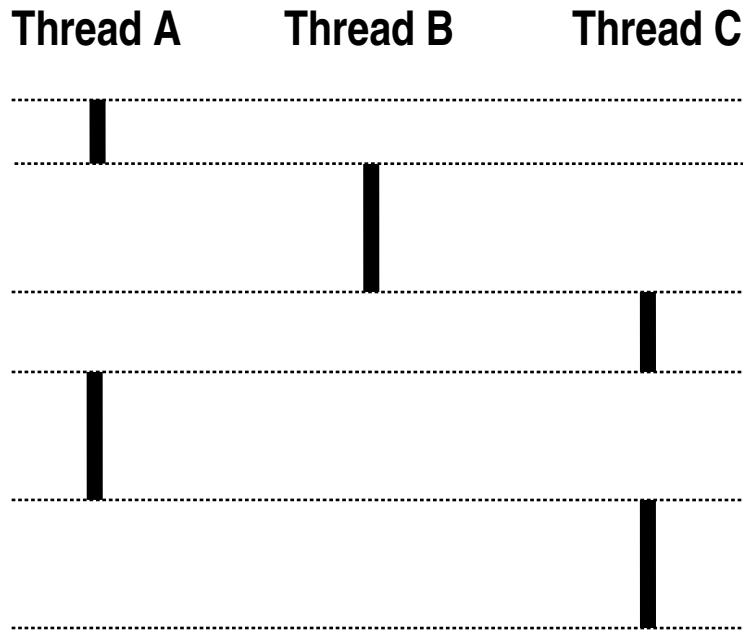




# Concurrent Thread Execution

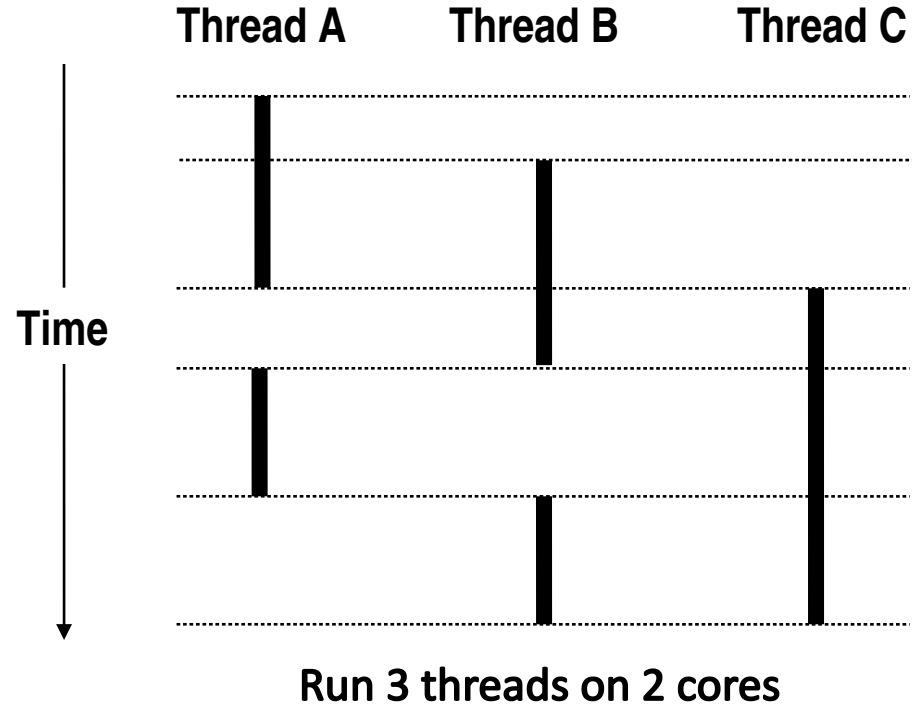
## ■ Single Core Processor

- Simulate parallelism by time slicing



## ■ Multi-Core Processor

- Can have true parallelism



# 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
    - *Much* larger difference on non-Unices.

# 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()` [terminates current thread]
    - `exit()` [terminates all threads]
  - 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);
}
```

hello.c

Thread ID

Thread attributes  
(usually NULL)

Thread routine

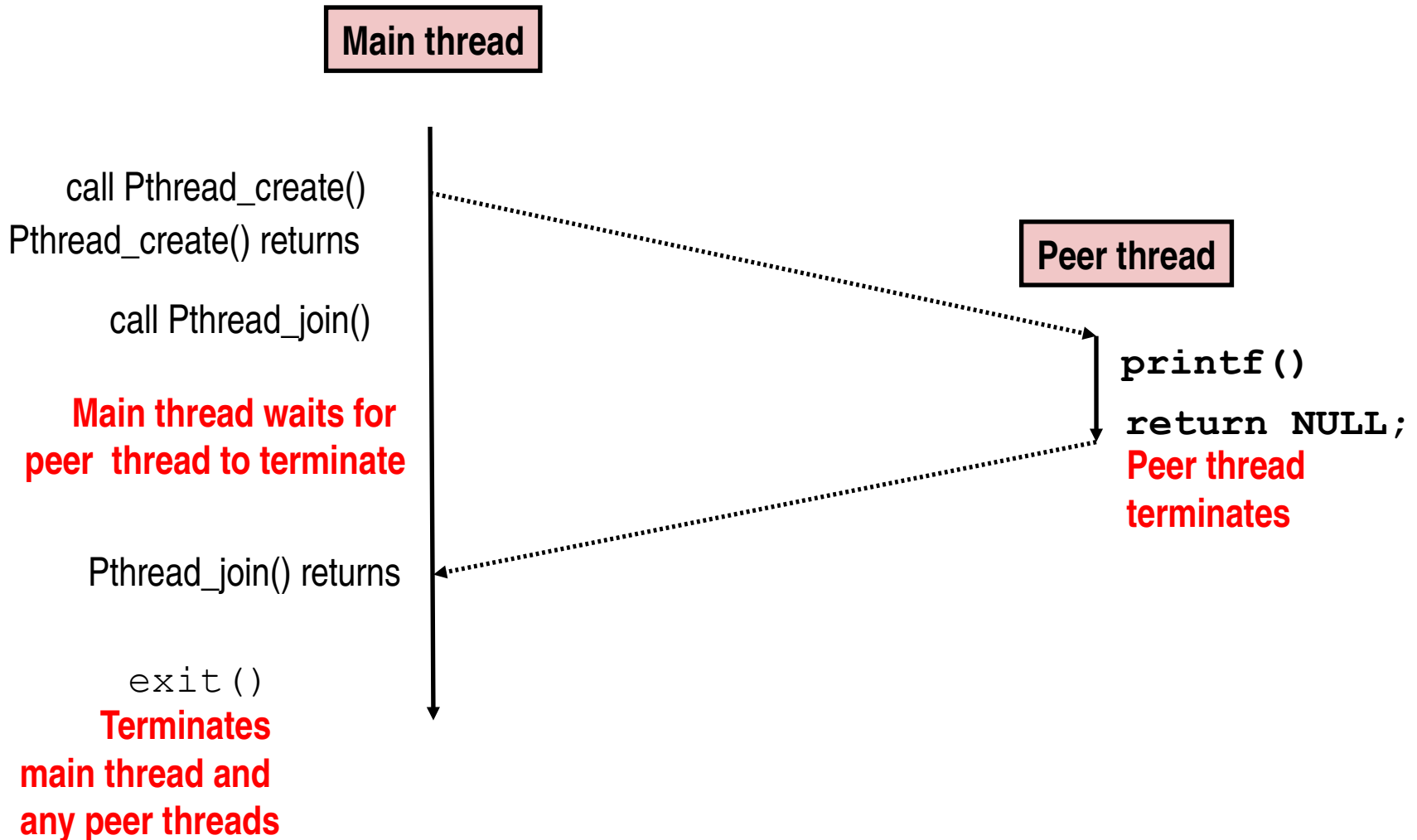
Thread arguments  
(void \*p)

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

hello.c

Return value  
(void \*\*p)

# Execution of Threaded “hello, world”



# Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**
  - ...take with a grain of salt.
- **– 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!

# Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared among threads?
  - 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 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

## ■ 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.**

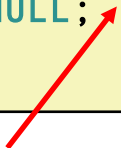
# Shared Variable Analysis

## ■ 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                 | yes                        | no                           | no                           |
| msgs              | yes                        | yes                          | yes                          |
| myid.p0           | no                         | yes                          | no                           |
| myid.p1           | no                         | no                           | yes                          |

```
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*

```
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

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

```

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  $i$*

|  |   |
|--|---|
| <pre> movq    (%rdi), %rcx testq   %rcx,%rcx jle     .L2 movl     \$0, %eax </pre>     | <pre> } <i>H<sub>i</sub> : Head</i> </pre>  |
| <pre> .L3: movq     cnt(%rip), %rdx addq     \$1, %rdx movq     %rdx, cnt(%rip) </pre> | <pre> } <i>L<sub>i</sub> : Load cnt</i> <i>U<sub>i</sub> : Update cnt</i> <i>S<sub>i</sub> : Store cnt</i> </pre> |
| <pre> addq     \$1, %rax cmpq     %rcx, %rax jne     .L3 .L2: </pre>                   | <pre> } <i>T<sub>i</sub> : Tail</i> </pre>  |

# 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

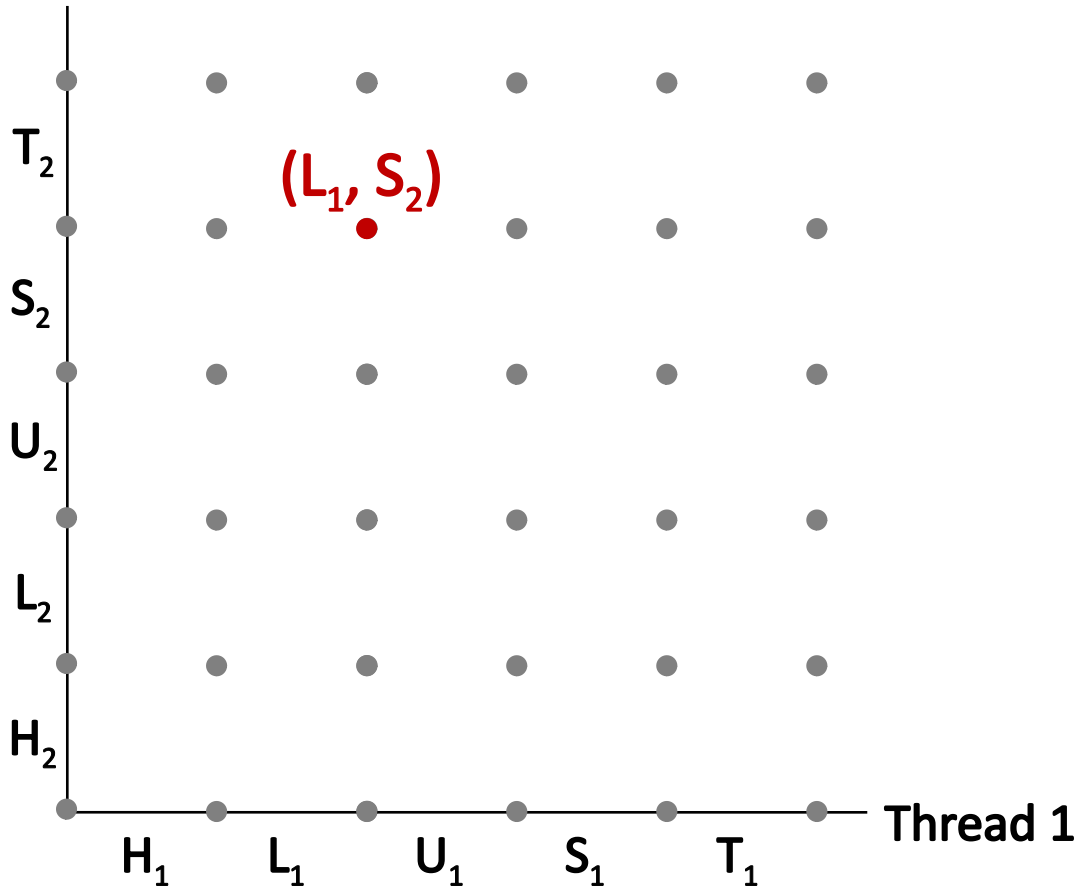
| i (thread) | instr <sub>i</sub> | %rdx <sub>1</sub> | %rdx <sub>2</sub> | cnt |
|------------|--------------------|-------------------|-------------------|-----|
| 1          | H <sub>1</sub>     | -                 | -                 | 0   |
| 1          | L <sub>1</sub>     | 0                 | -                 | 0   |
| 1          | U <sub>1</sub>     | 1                 | -                 | 0   |
| 2          | H <sub>2</sub>     | -                 | -                 | 0   |
| 2          | L <sub>2</sub>     | -                 | 0                 | 0   |
| 1          | S <sub>1</sub>     | 1                 | -                 | 1   |
| 1          | T <sub>1</sub>     | 1                 | -                 | 1   |
| 2          | U <sub>2</sub>     | -                 | 1                 | 1   |
| 2          | S <sub>2</sub>     | -                 | 1                 | 1   |
| 2          | T <sub>2</sub>     | -                 | 1                 | 1   |

*Oops!*



# Progress Graphs

Thread 2



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

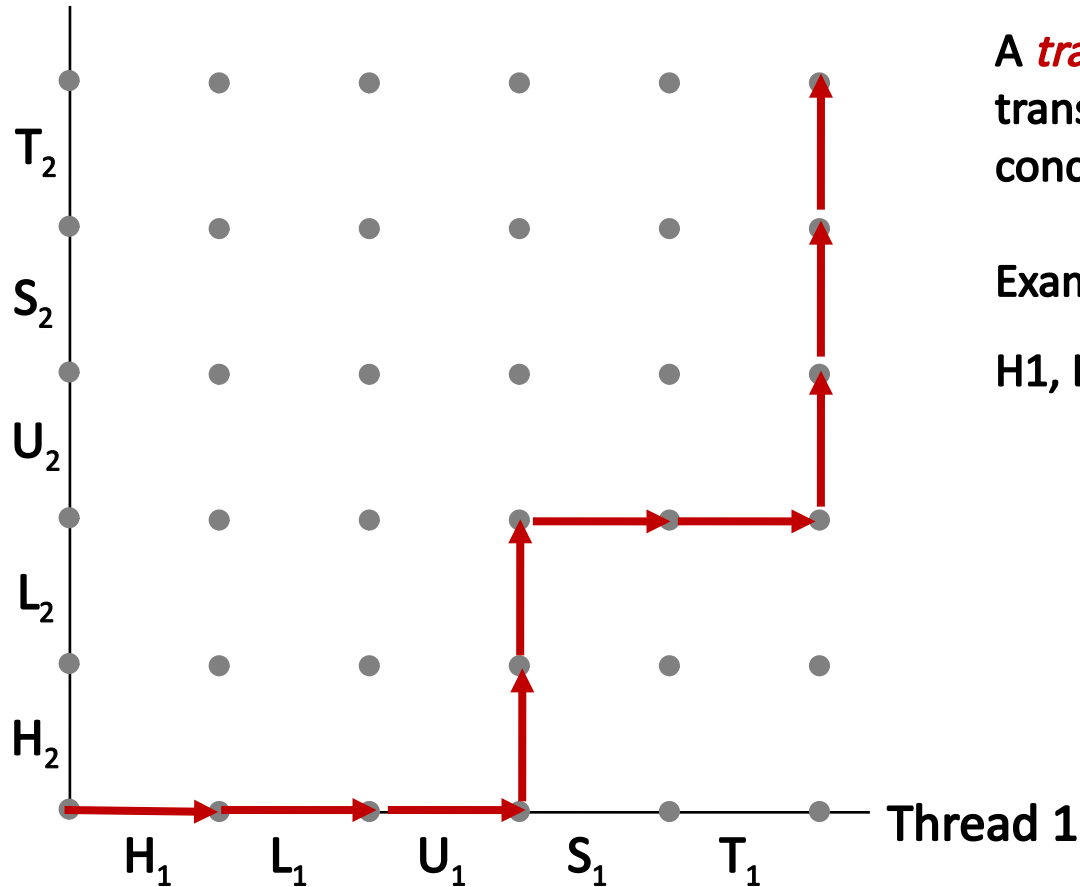
Each axis corresponds to the sequential order of instructions in a thread.

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

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

Thread 2



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

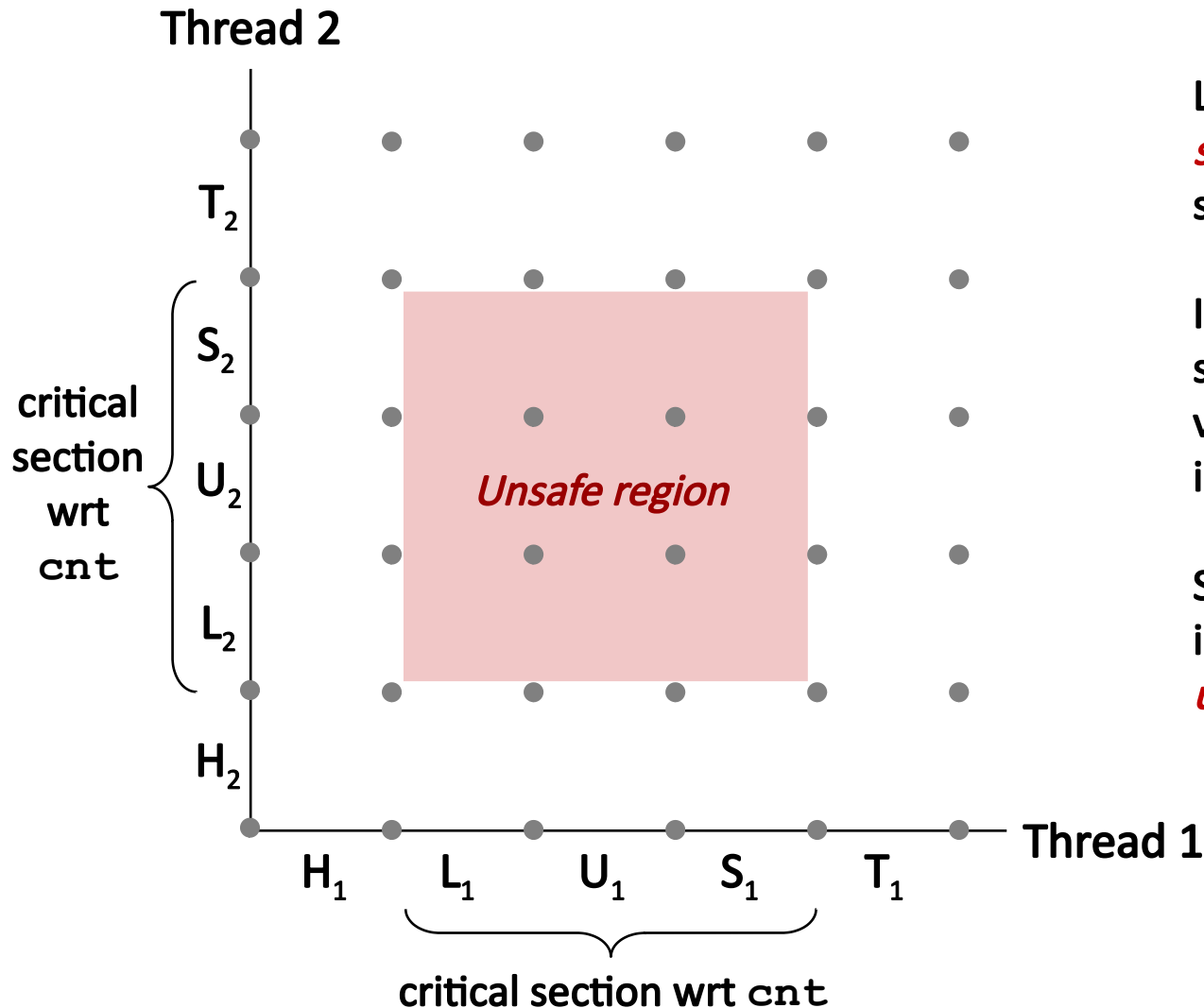
Example:

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

# 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)
- **Other approaches**
  - Mutexes and condition variables from Pthreads
  - Monitors (Java) (boring languages are outside our scope)

# Critical Sections and Unsafe Regions

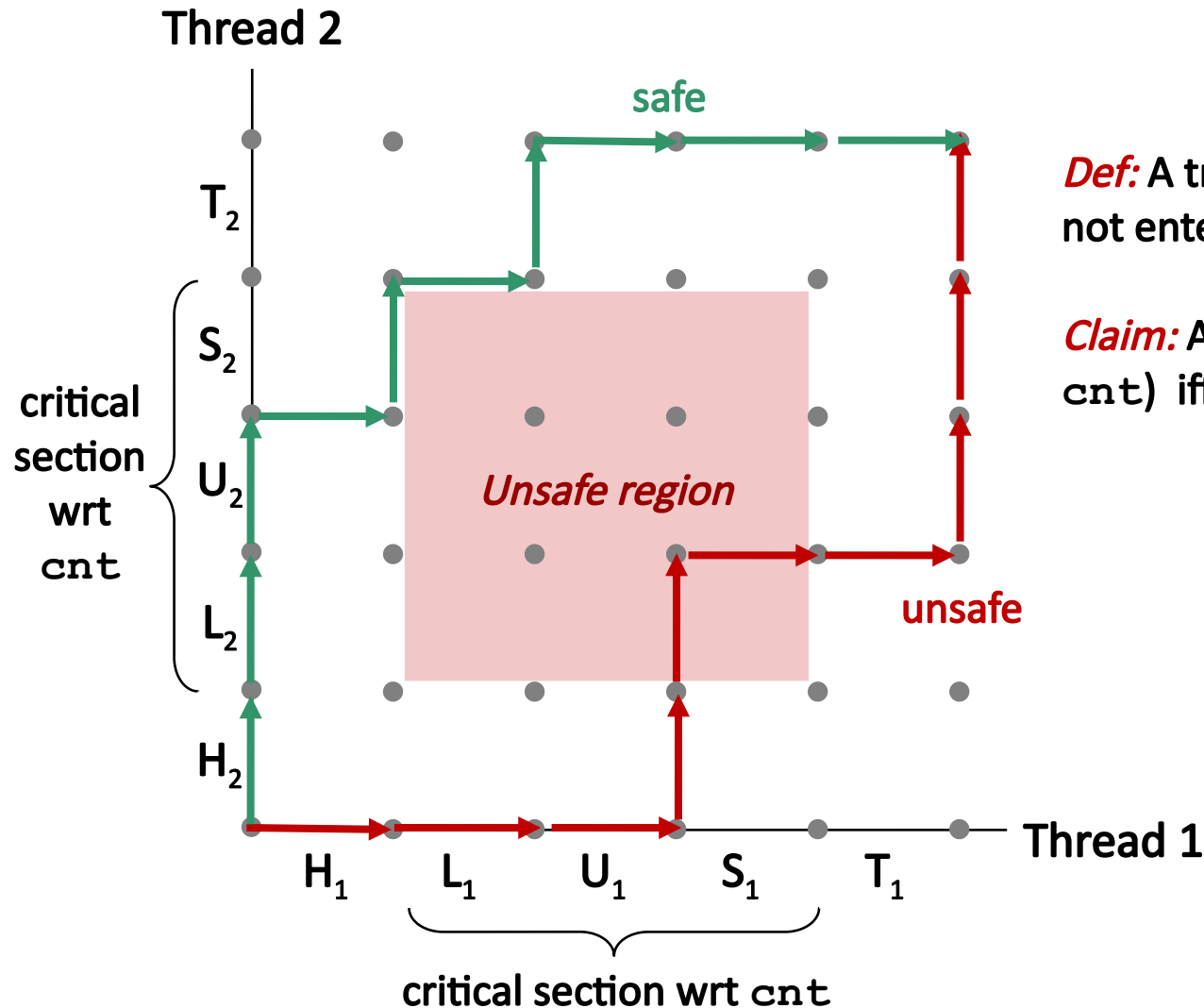


$L$ ,  $U$ , and  $S$  form a *critical section* with respect to the shared variable `cnt`

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

# Semaphores

- ***Semaphore***: non-negative global integer synchronization variable. Manipulated by *P (passering)* and *V (vrijgave)* 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: ( $s \geq 0$ )**

# 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 */
```

# 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);
}
```

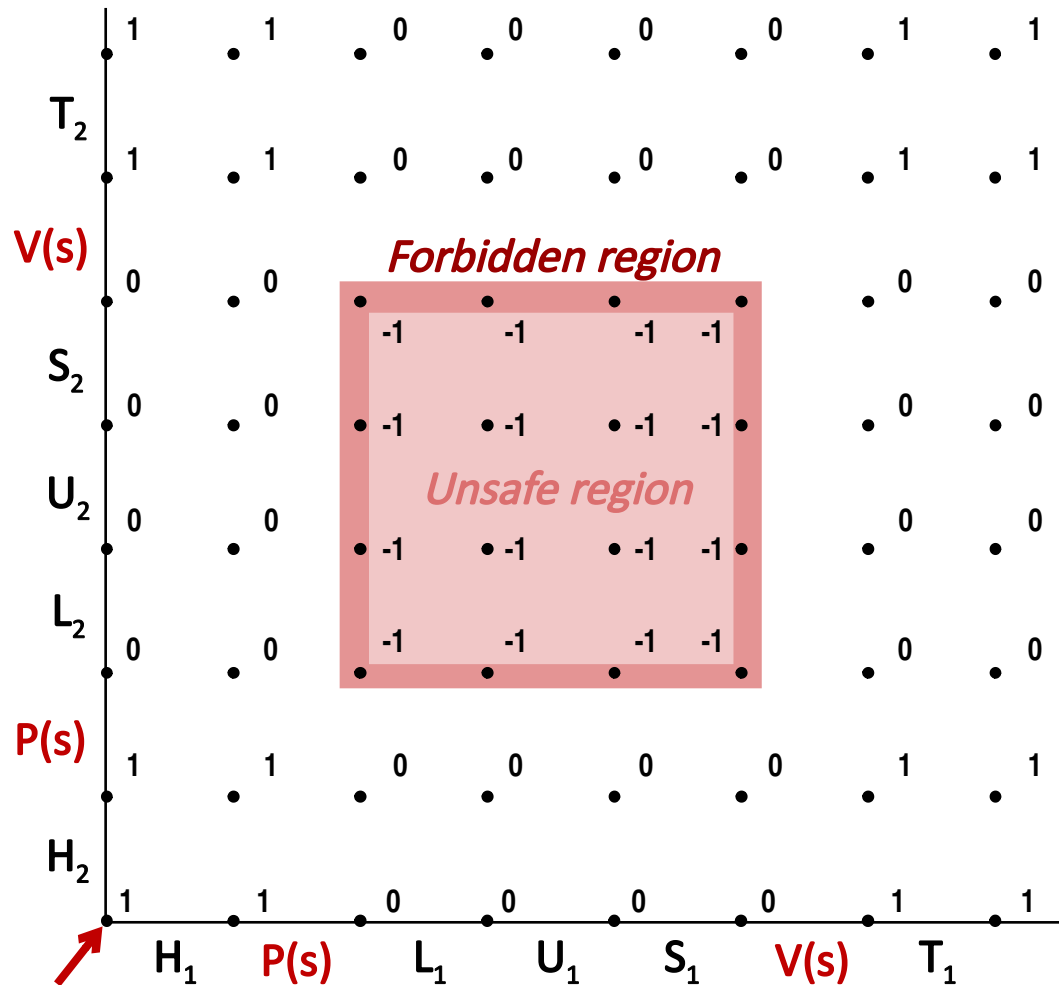
goodcnt.c

```
$ ./goodcnt 10000
OK cnt=20000
$ ./goodcnt 10000
OK cnt=20000
$
```

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

# Why Mutexes Work

Thread 2



Provide mutually exclusive access to shared variable by surrounding critical section with  $P$  and  $V$  operations on semaphore  $s$  (initially set to 1)

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

Thread 1

# 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.**