

# Synchronization: Basics

CSE251: System Programming  
24<sup>th</sup> Lecture, Jun. 3, 2019

**Instructor:**

Hyungon Moon

# 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 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])

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

**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?

<i>Variable instance</i>	<i>Referenced by main thread?</i>	<i>Referenced by peer thread 0?</i>	<i>Referenced by peer thread 1?</i>
<code>ptr</code>	yes	yes	yes
<code>cnt</code>	no	yes	yes
<code>i.m</code>	yes	no	no
<code>msgs.m</code>	yes	yes	yes
<code>myid.p0</code>	no	yes	no
<code>myid.p1</code>	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

```
/* 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 $i$

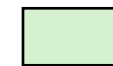
<pre>movq    (%rdi), %rcx testq   %rcx,%rcx jle     .L2 movl    \$0, %eax</pre>	$H_i$ : Head
<pre>.L3: movq     cnt(%rip), %rdx addq     \$1, %rdx movq     %rdx, cnt(%rip)</pre>	$L_i$ : Load cnt $U_i$ : Update cnt $S_i$ : Store cnt
<pre>addq     \$1, %rax cmpq     %rcx, %rax jne      .L3 .L2:</pre>	$T_i$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

# 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!*

# Concurrent Execution (cont)

## ■ How about this ordering?

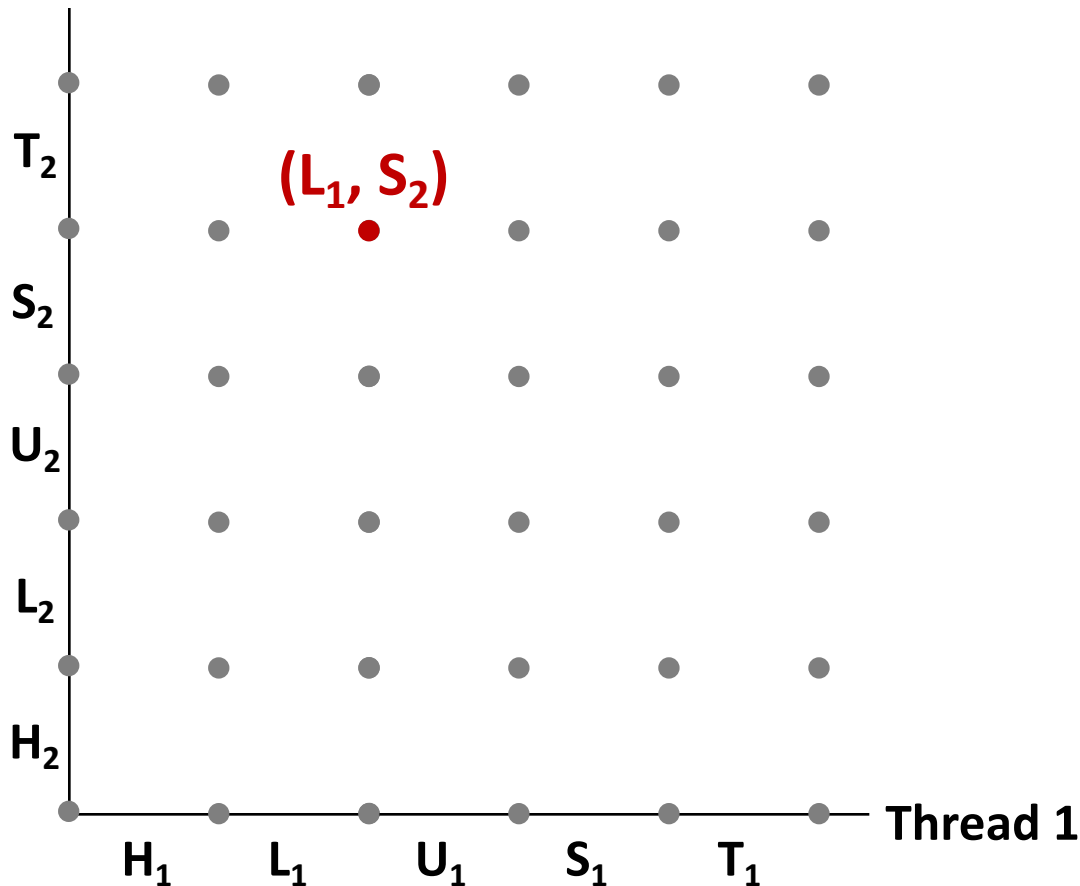
i (thread)	instr <sub>i</sub>	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	H <sub>1</sub>			0
1	L <sub>1</sub>	0		
2	H <sub>2</sub>			
2	L <sub>2</sub>		0	
2	U <sub>2</sub>		1	
2	S <sub>2</sub>		1	1
1	U <sub>1</sub>	1		
1	S <sub>1</sub>	1		1
1	T <sub>1</sub>			1
2	T <sub>2</sub>			1

*Oops!*

## ■ We can analyze the behavior using a *progress graph*

# Progress Graphs

Thread 2



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

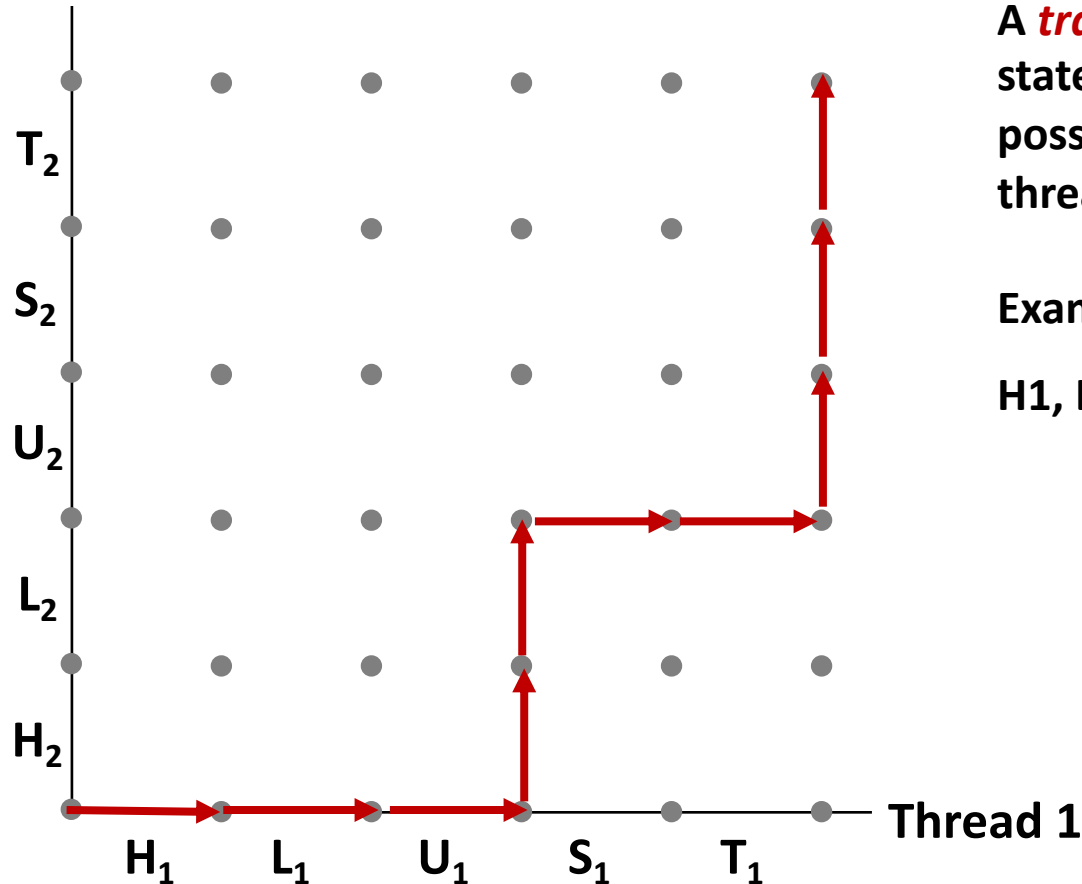
Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible **execution state** ( $\text{Inst}_1, \text{Inst}_2$ ).

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

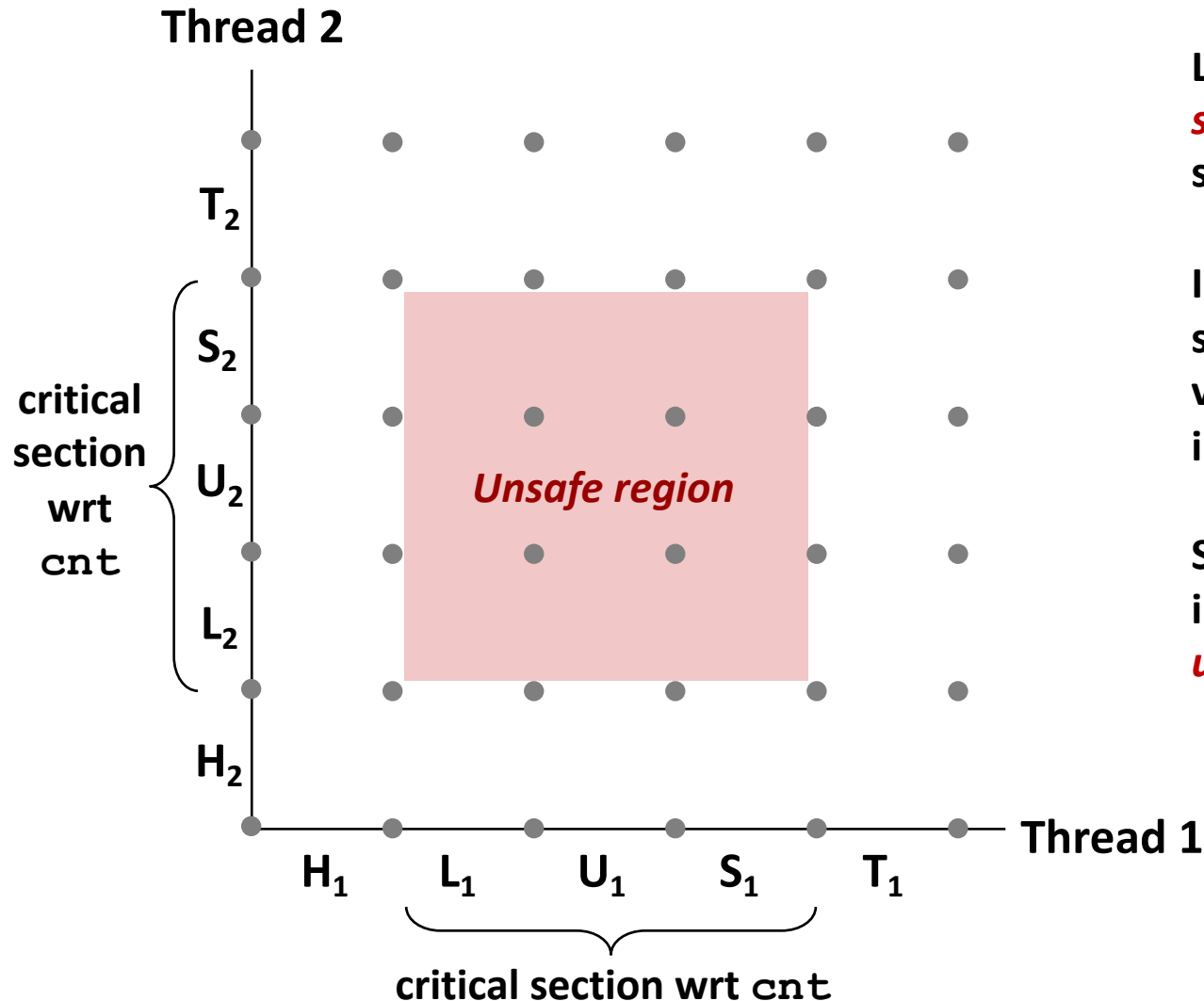


A **trajectory** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

# Critical Sections and Unsafe Regions



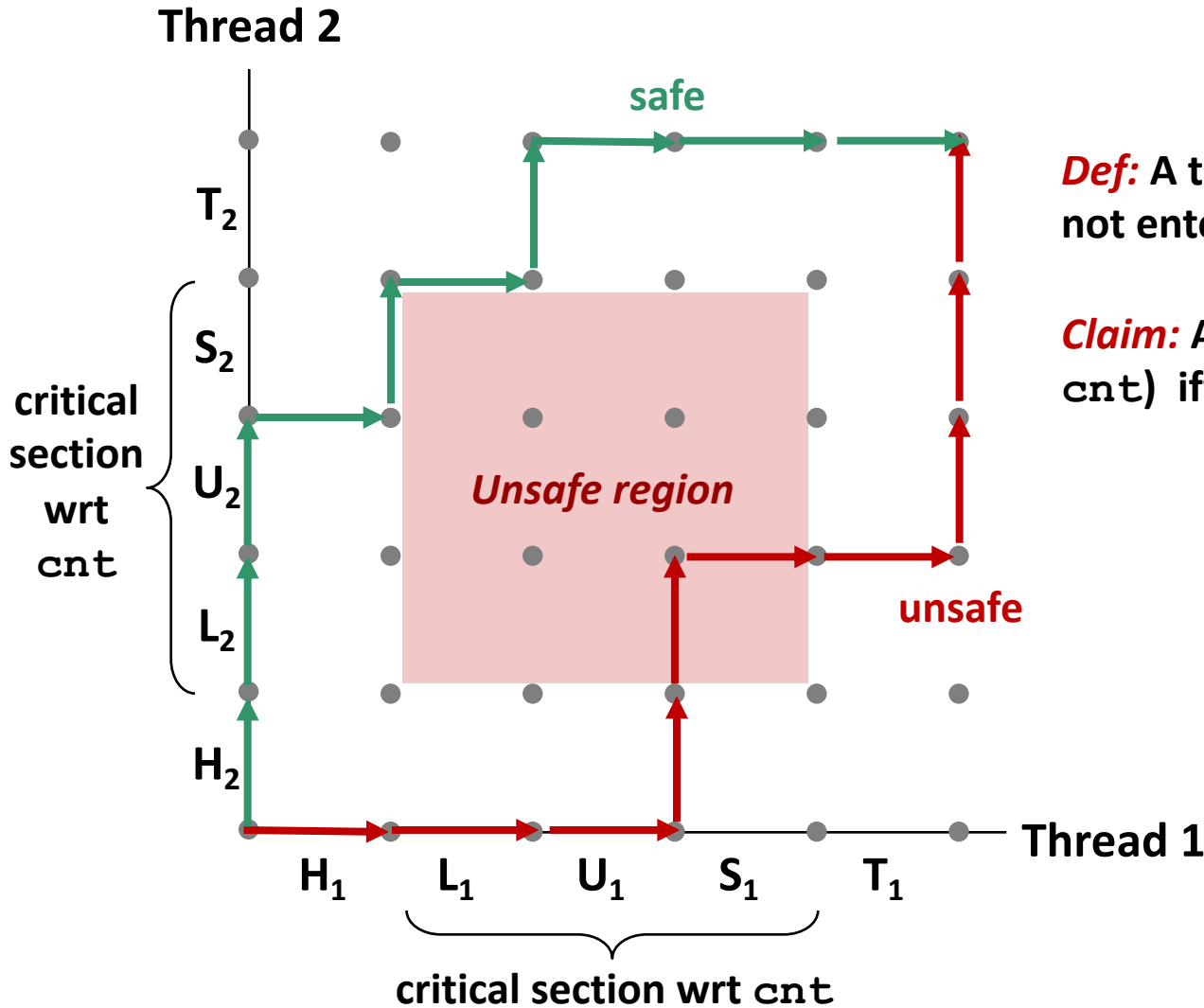
L, U, and S form a **critical section** with respect to the shared variable `cnt`

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  $\text{cnt}$ ) 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)
- **Other approaches (out of our scope)**
  - Mutex and 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: ( $s \geq 0$ )**

# 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 */
```

# 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 semaphores?

# 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);
}
```

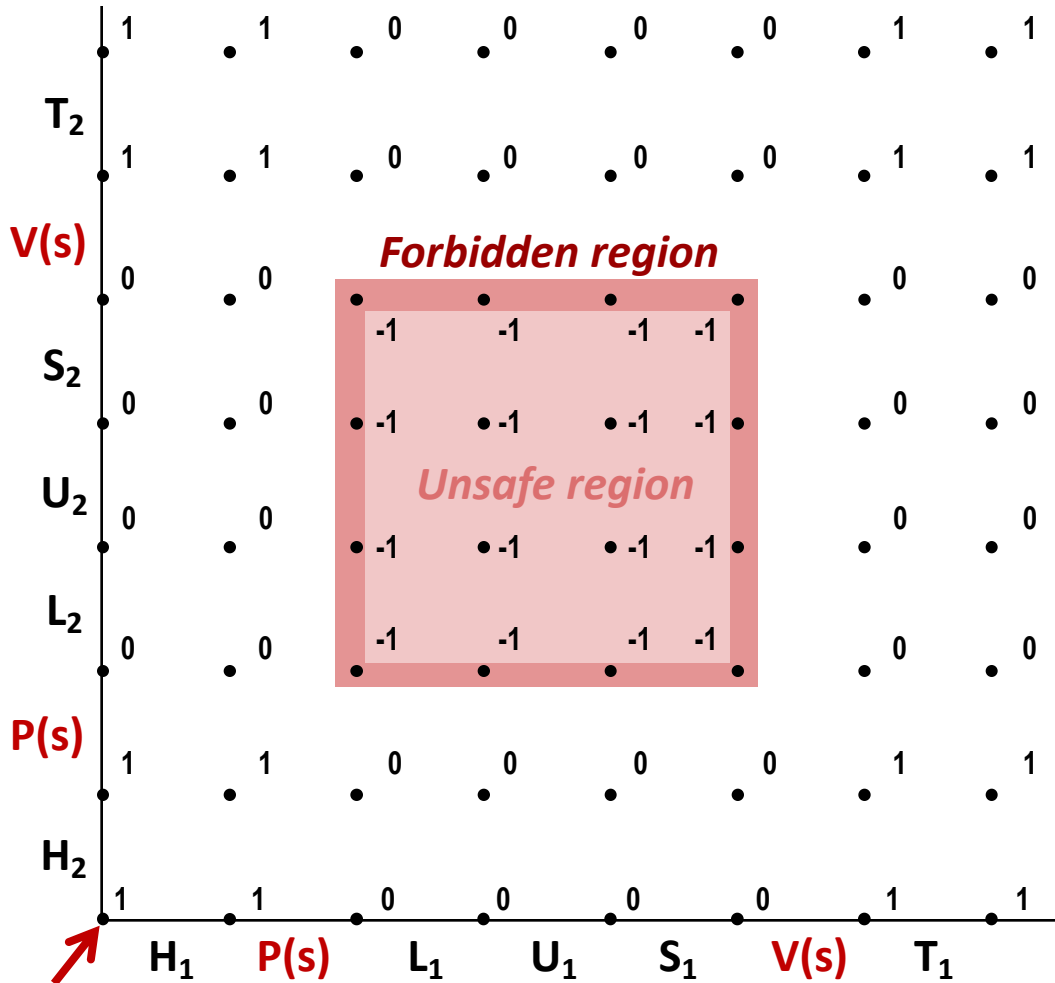
goodcnt.c

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

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

# 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.

Initially

$s = 1$

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.**