

Synchronization: Basics

Introduction to Computer Systems

Today

- **Threads review**
- **Sharing**
- **Mutual exclusion**
- **Semaphores**
- **Other concurrency issues**
 - Races
 - Locking and Deadlocks

Process: Traditional View

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

Process context

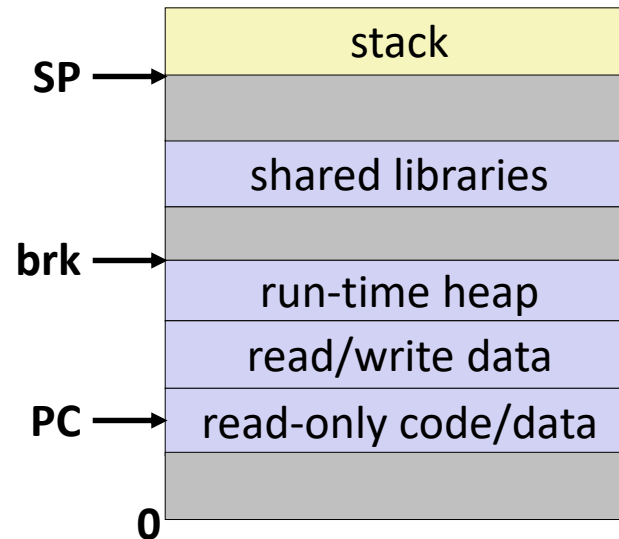
Program context:

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

Kernel context:

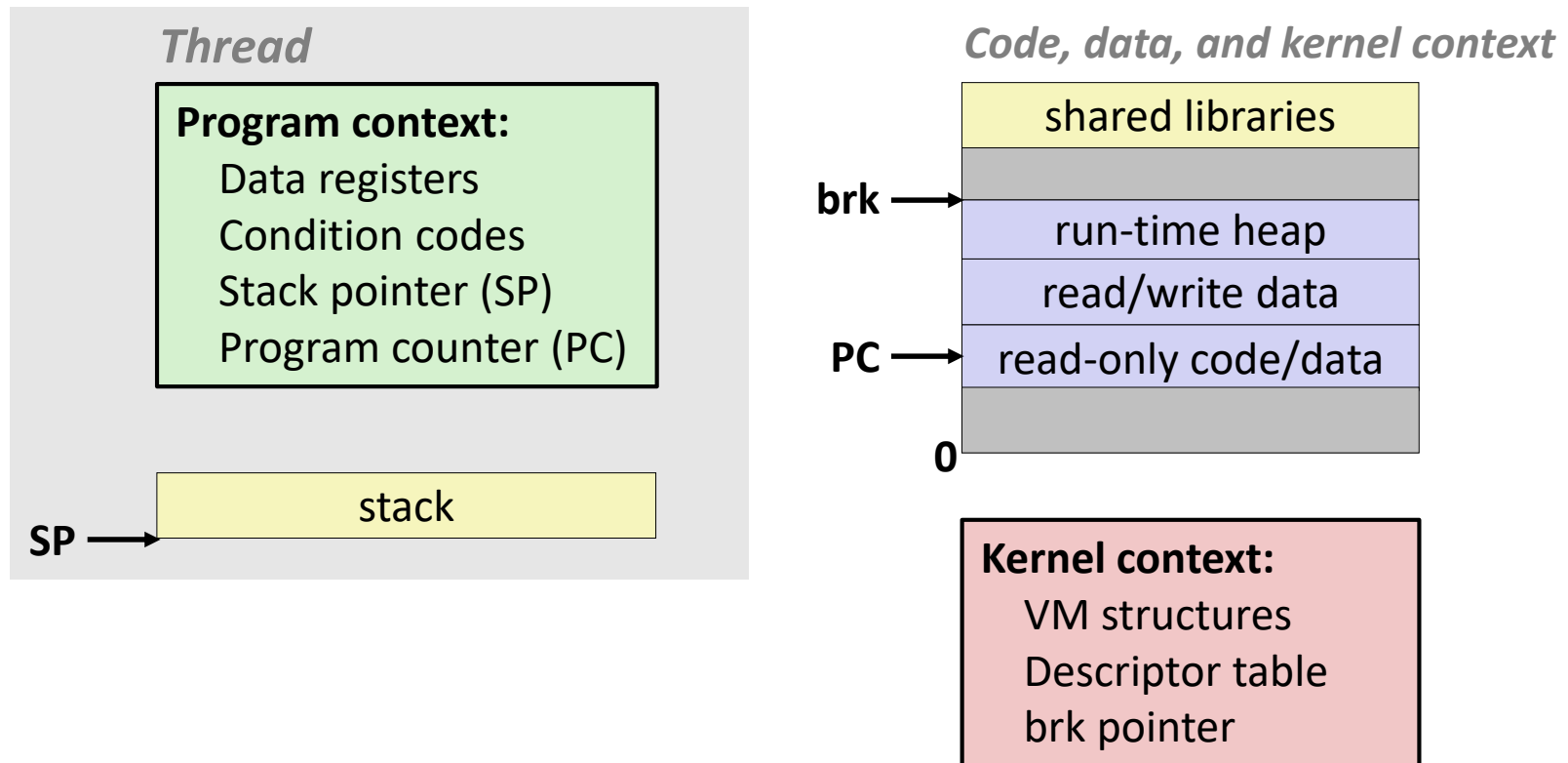
VM structures
Descriptor table
brk pointer

Code, data, and stack



Process: Alternative View

- Process = thread + code, data, and kernel context

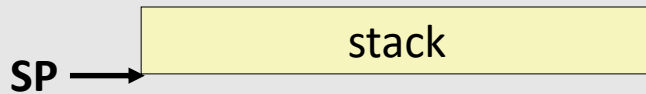


Process with Two Threads

Thread 1

Program context:

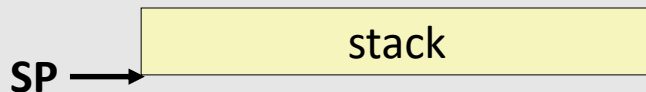
Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)



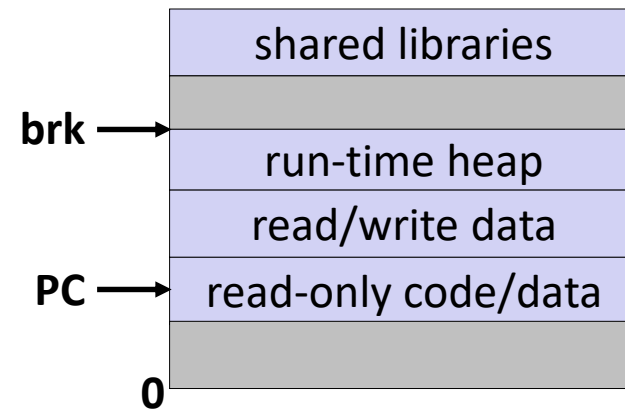
Thread 2

Program context:

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



Code, data, and kernel context



Kernel context:

VM structures
Descriptor table
brk pointer

Threads vs. Processes

■ Threads and processes: similarities

- Each has its own logical control flow
- Each can run concurrently with others
- Each is scheduled and context switched by the kernel

■ Threads and processes: differences

- Threads share code and data, processes (typically) do not
- Threads are less expensive than processes
 - Process control (creating and reaping) is more expensive than thread control
 - Context switches for processes more expensive than for threads

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**
- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**

Today

- Threads review
- **Sharing**
- Mutual exclusion
- Semaphores
- Other concurrency issues
 - Races
 - Locking and Deadlocks

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

Today

- Threads review
- Sharing
- **Mutual exclusion**
- Semaphores
- Other concurrency issues
 - Races
 - Locking and Deadlocks

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>	<pre> } <i>H_i : Head</i> </pre>
<pre> .L3: movq cnt(%rip), %rdx addq \$1, %rdx movq %rdx, cnt(%rip) </pre>	<pre> } <i>L_i : Load cnt</i> <i>U_i : Update cnt</i> <i>S_i : Store cnt</i> </pre>
<pre> addq \$1, %rax cmpq %rcx, %rax jne .L3 .L2: </pre>	<pre> } <i>T_i : 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 _i	%rdx ₁	%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?

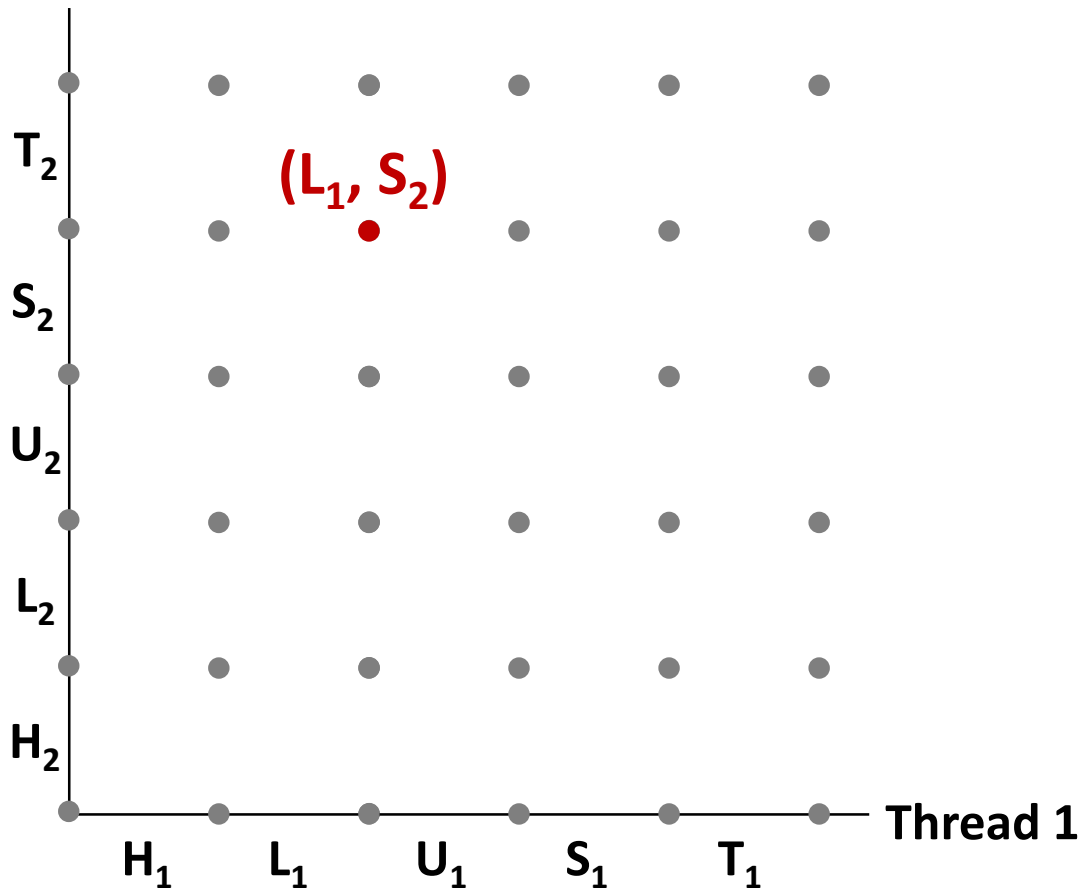
i (thread)	instr _i	%rdx ₁	%rdx ₂	cnt
1	H ₁			0
1	L ₁	0		
2	H ₂			
2	L ₂		0	
2	U ₂		1	
2	S ₂		1	1
1	U ₁	1		
1	S ₁	1		1
1	T ₁			1
2	T ₂			1

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.

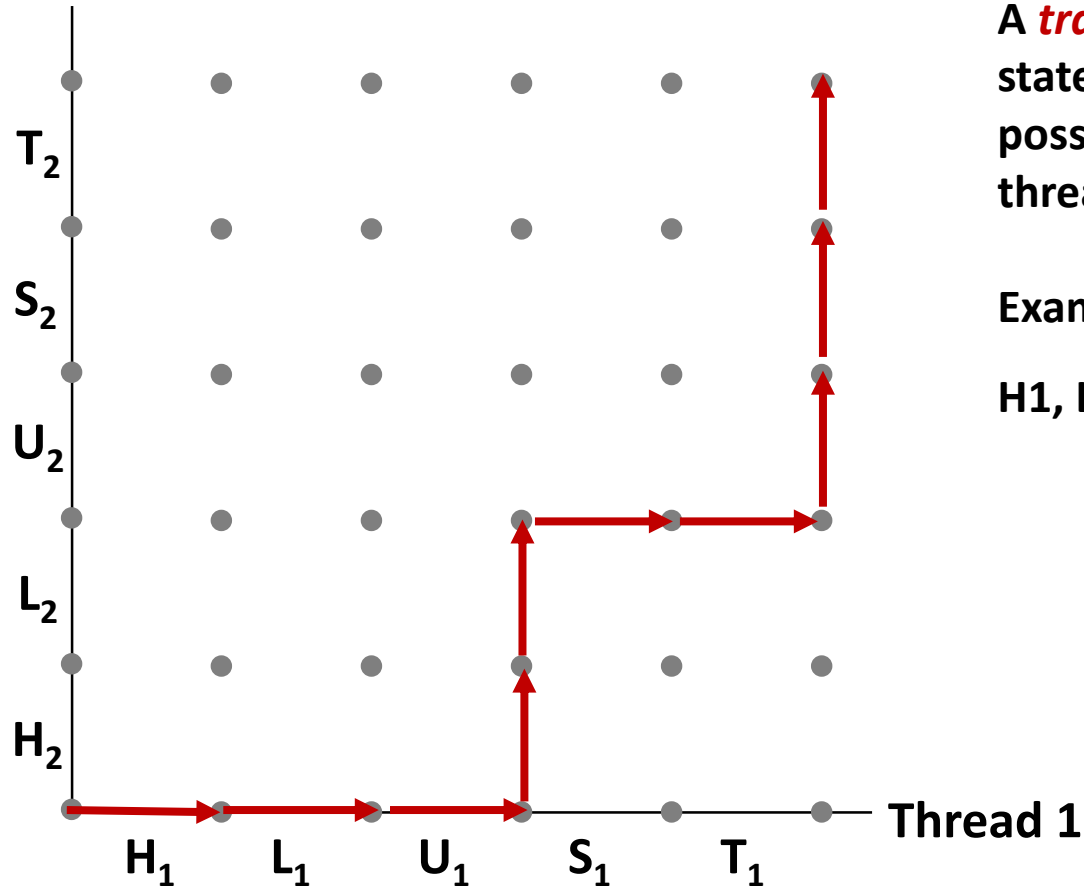
Each axis corresponds to the sequential order of instructions in a thread.

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

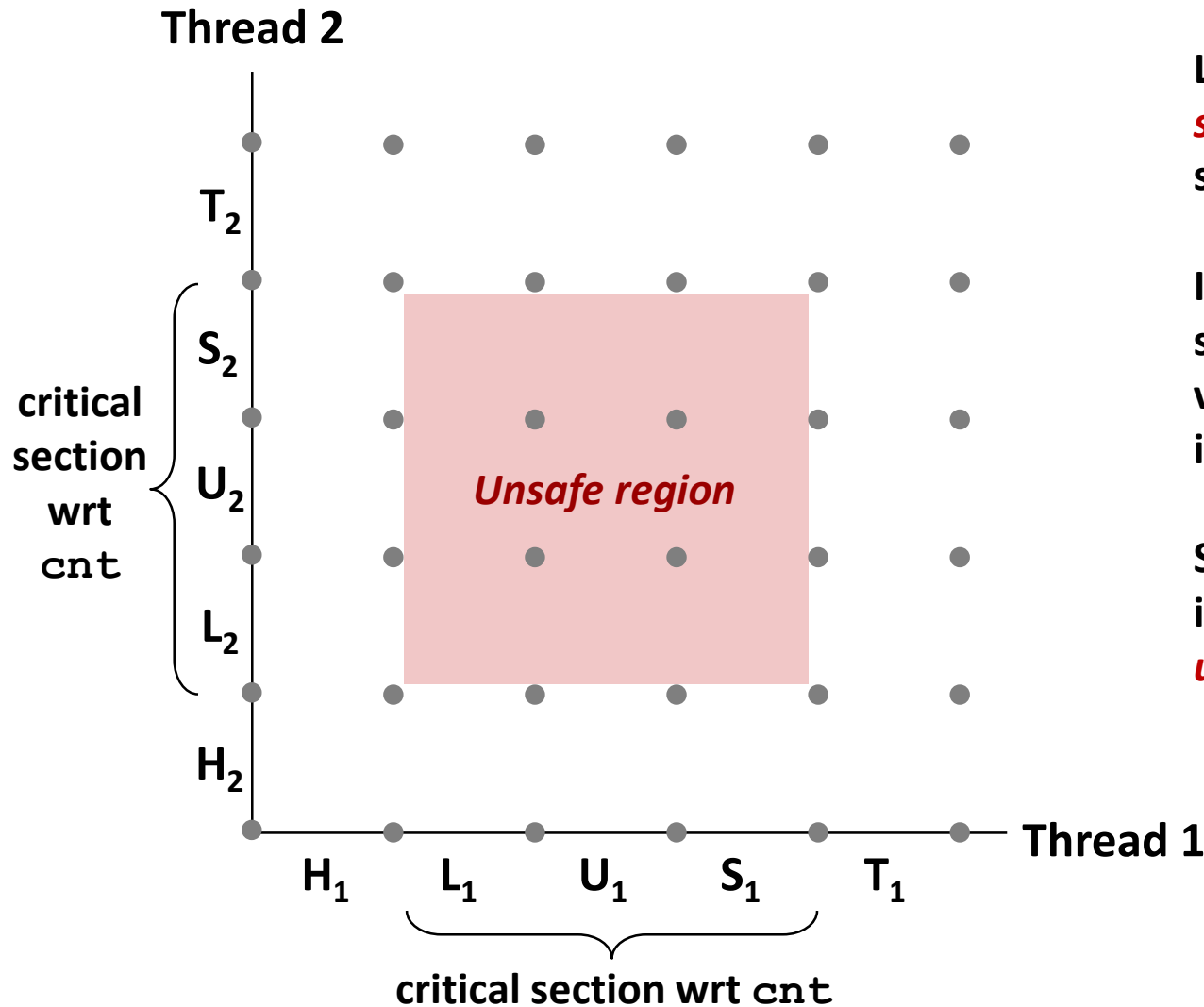


A **trajectory** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

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

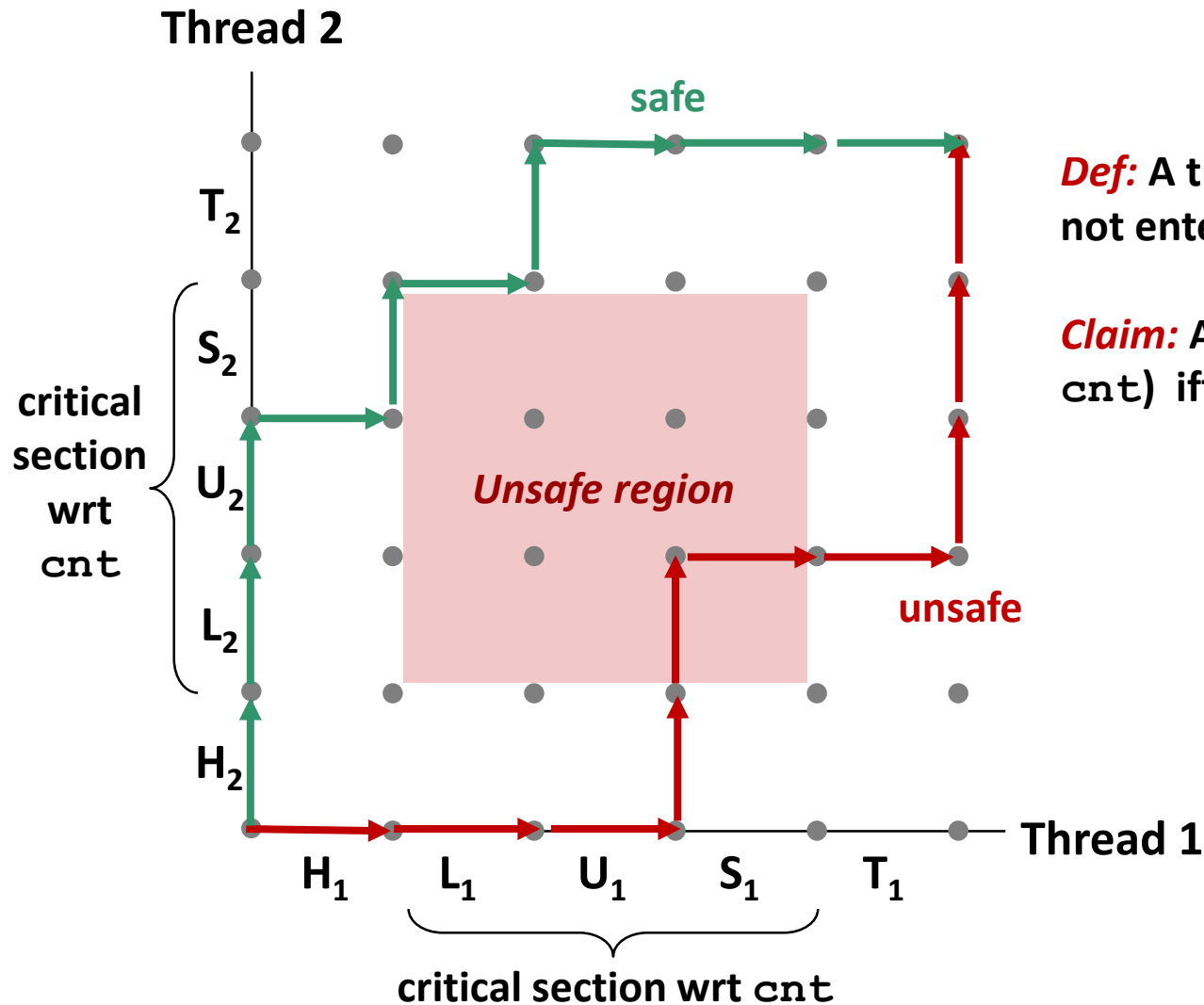


L, U, and S form a **critical section** with respect to the shared variable `cnt`

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 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)

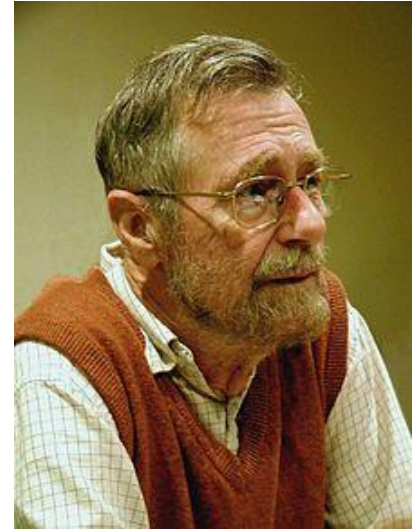
Today

- Threads review
- Sharing
- Mutual exclusion
- **Semaphores**
- Other concurrency issues
 - Races
 - Locking and Deadlocks

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: ($s \geq 0$)**

Semaphores (cont.)



Edsger Wybe Dijkstra
in 2002

- **Manipulated by P and V operations:**
 - $P(s)$: [**while** ($s == 0$) **wait()**; $s--$;]
 - Dutch for "Proberen" (test)
 - $V(s)$: [$s++$;]
 - Dutch for "Verhogen" (increment)
- **OS kernel guarantees that operations between brackets [] are executed indivisibly**
 - Only one P or V operation at a time can modify s .
 - When **while** loop in P terminates, only that P can decrement s

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);
}
```

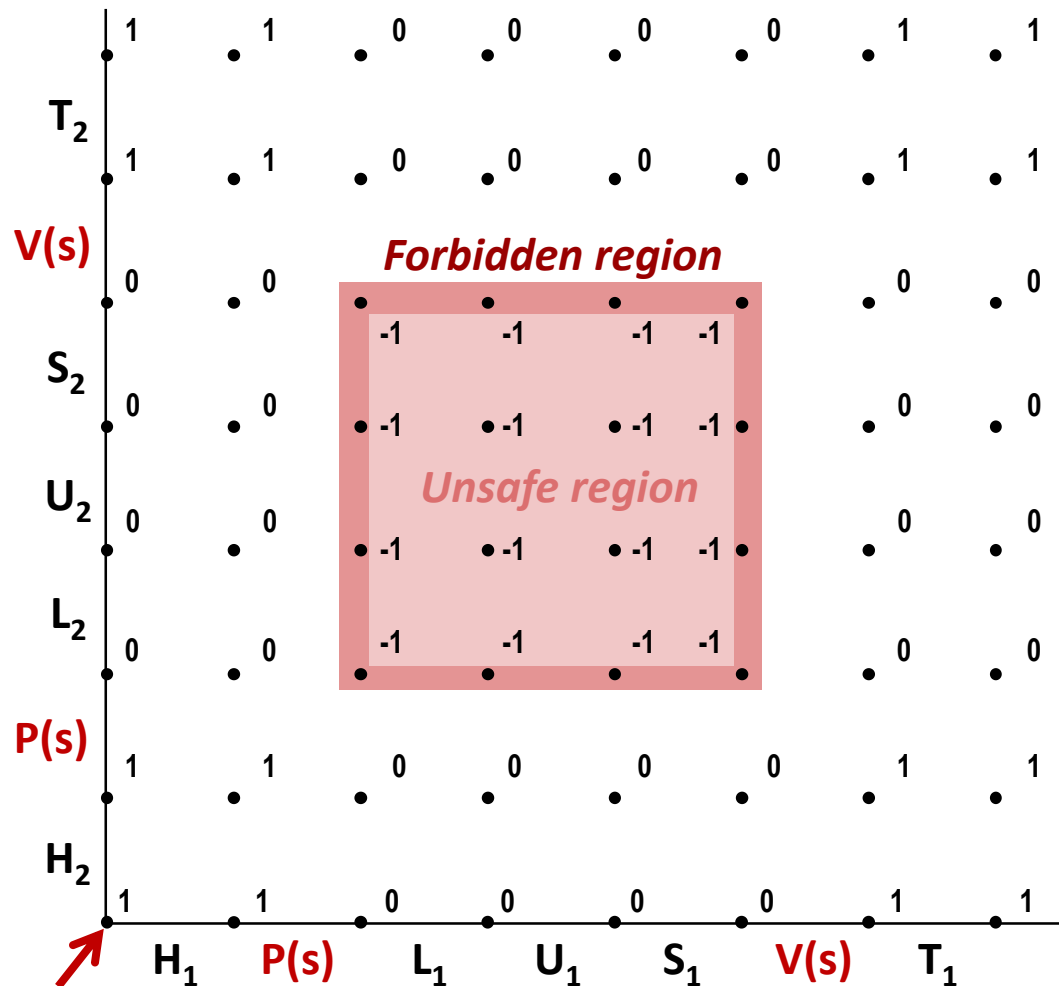
goodcnt.c

```
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



Initially
 $s = 1$

Thread 1

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.

Today

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
- **Other concurrency issues**
 - Races
 - Locking and Deadlocks

One worry: Races

- A **race** occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

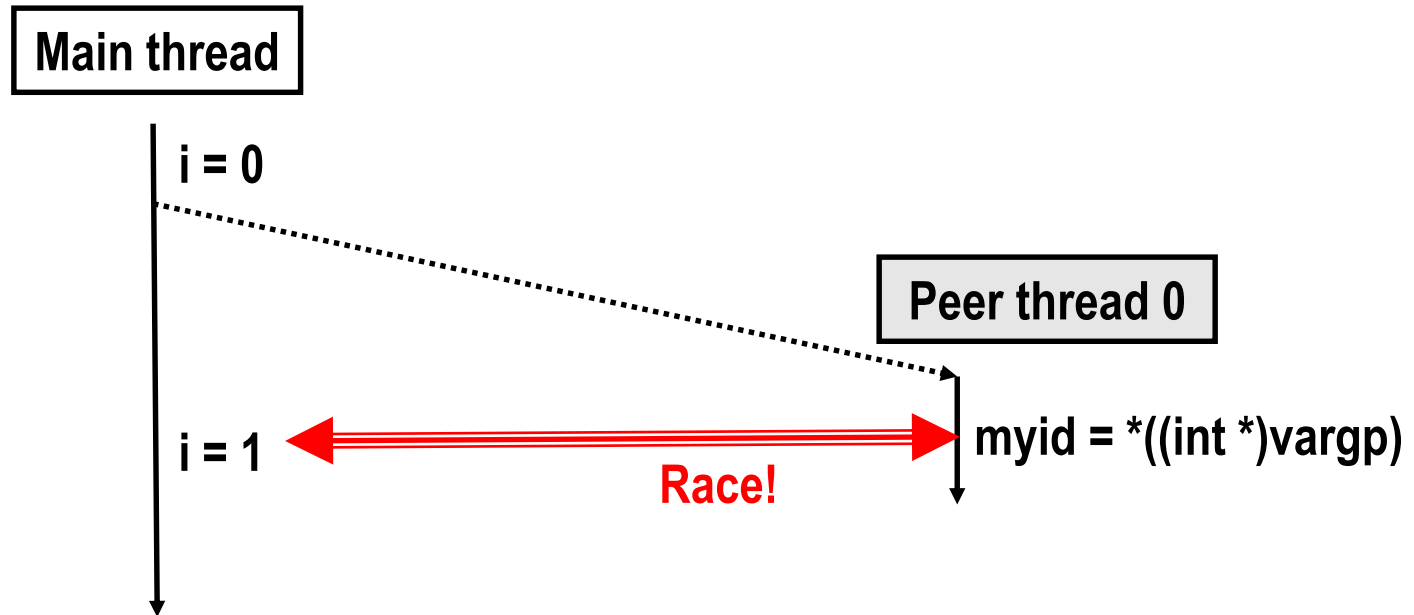
/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

N threads are sharing i

race.c

Race Illustration

```
for (i = 0; i < N; i++)  
    Pthread_create(&tid[i], NULL, thread, &i);
```



■ Race between increment of i in main thread and deref of `vargp` in peer thread:

- If deref happens while $i = 0$, then OK
- Otherwise, peer thread gets wrong id value

Race Elimination

```
/* Threaded program without the race */
int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

■ Avoid unintended sharing of state

Another worry: Deadlock

- Def: A process is *deadlocked* iff it is waiting for a condition that will never be true
- 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

```
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);
}
```

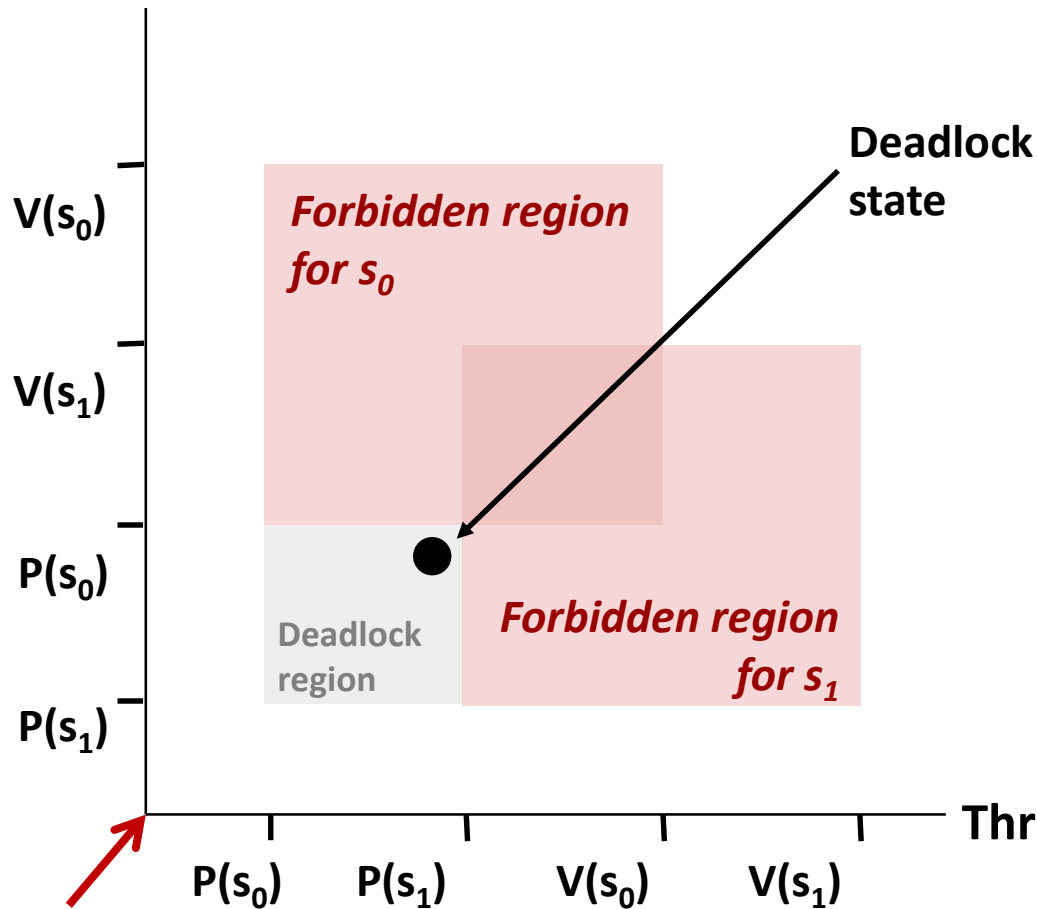
```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

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

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

Deadlock Visualized in Progress Graph

Thread 1



Locking introduces the potential for **deadlock**: waiting for a condition that will never be true

Any trajectory that enters the **deadlock region** will eventually reach the **deadlock state**, waiting for either S_0 or S_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often nondeterministic (race)

Avoiding Deadlock

Acquire shared resources in same order

```
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);
}
```

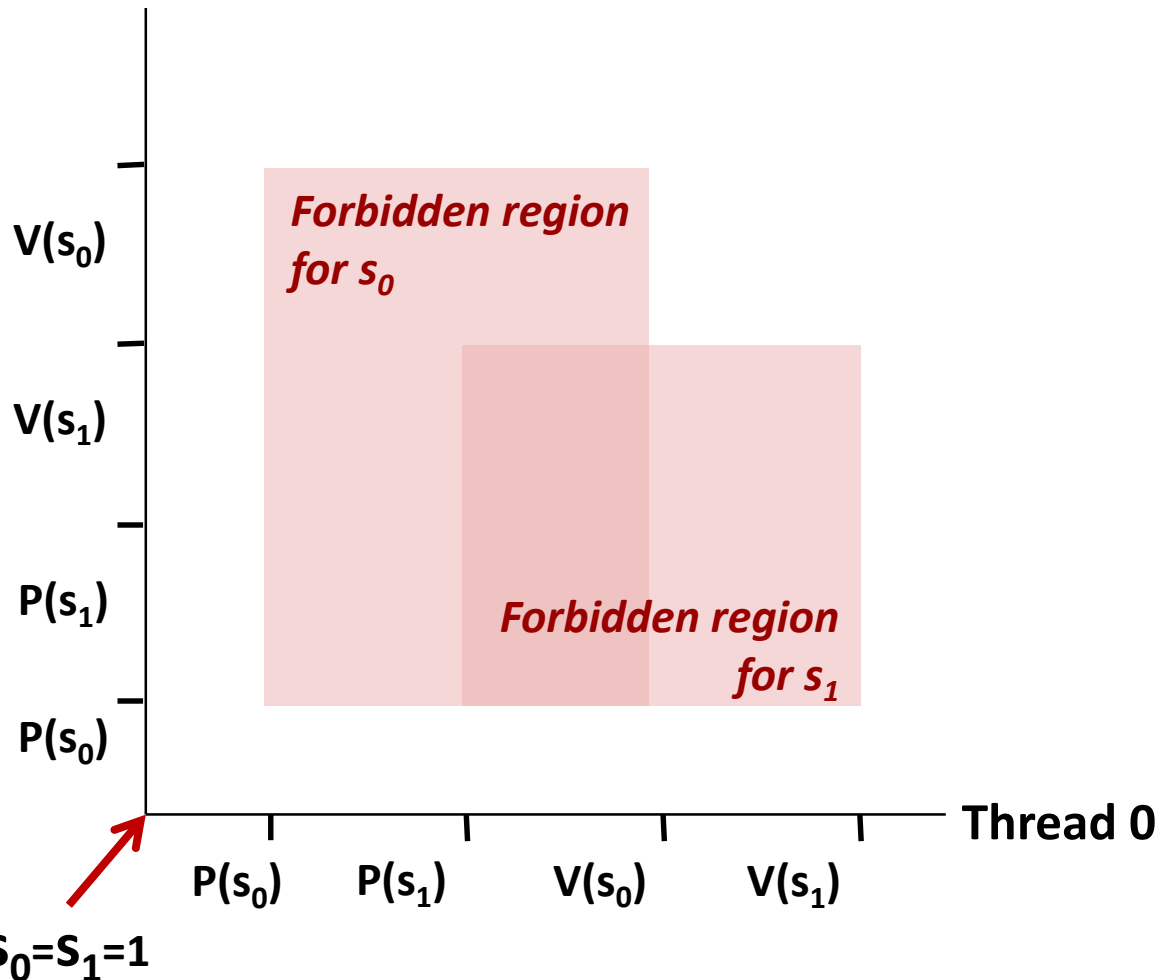
```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

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

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

Avoided Deadlock in Progress Graph

Thread 1



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

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.**