Concurrent Programming II

HPPS

David Marchant

Based on slides by:
Troels Henriksen, Randal E. Bryant and David R. O’Hallaron

Concurrent Programming is Hard!

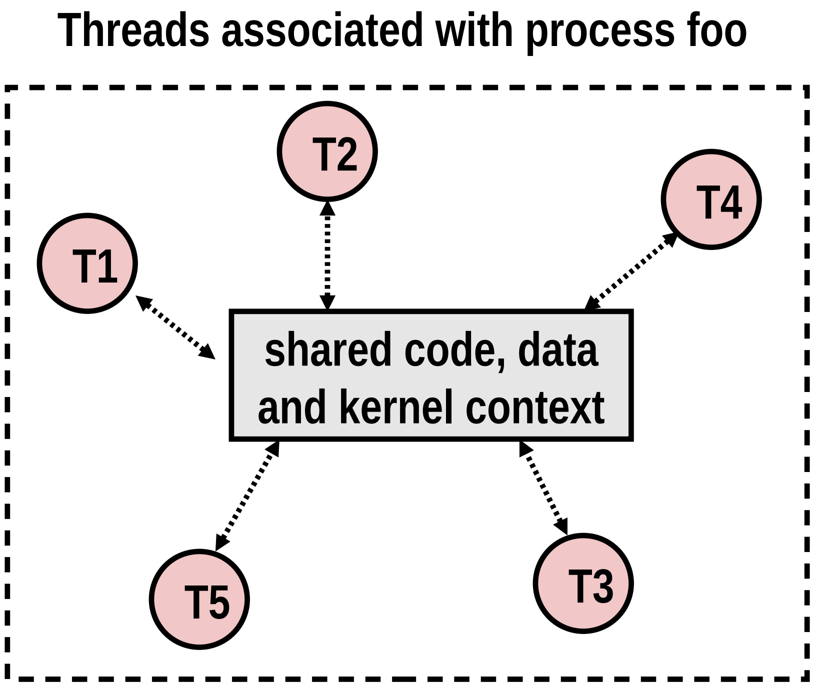
- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

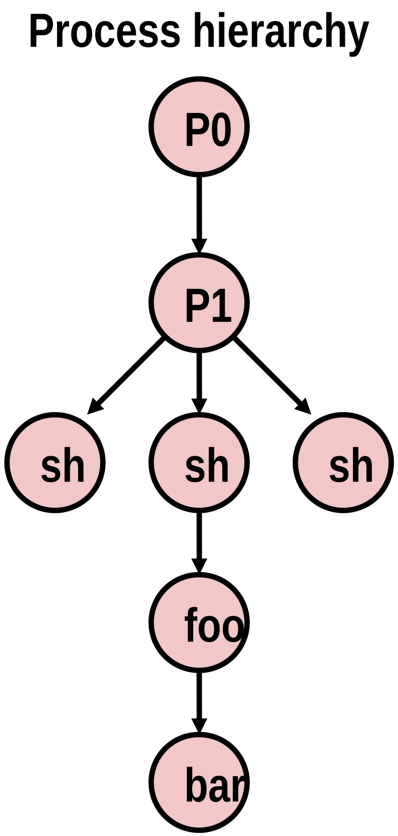
Concurrent Programming is Hard!

- **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.

Logical View of Threads

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





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

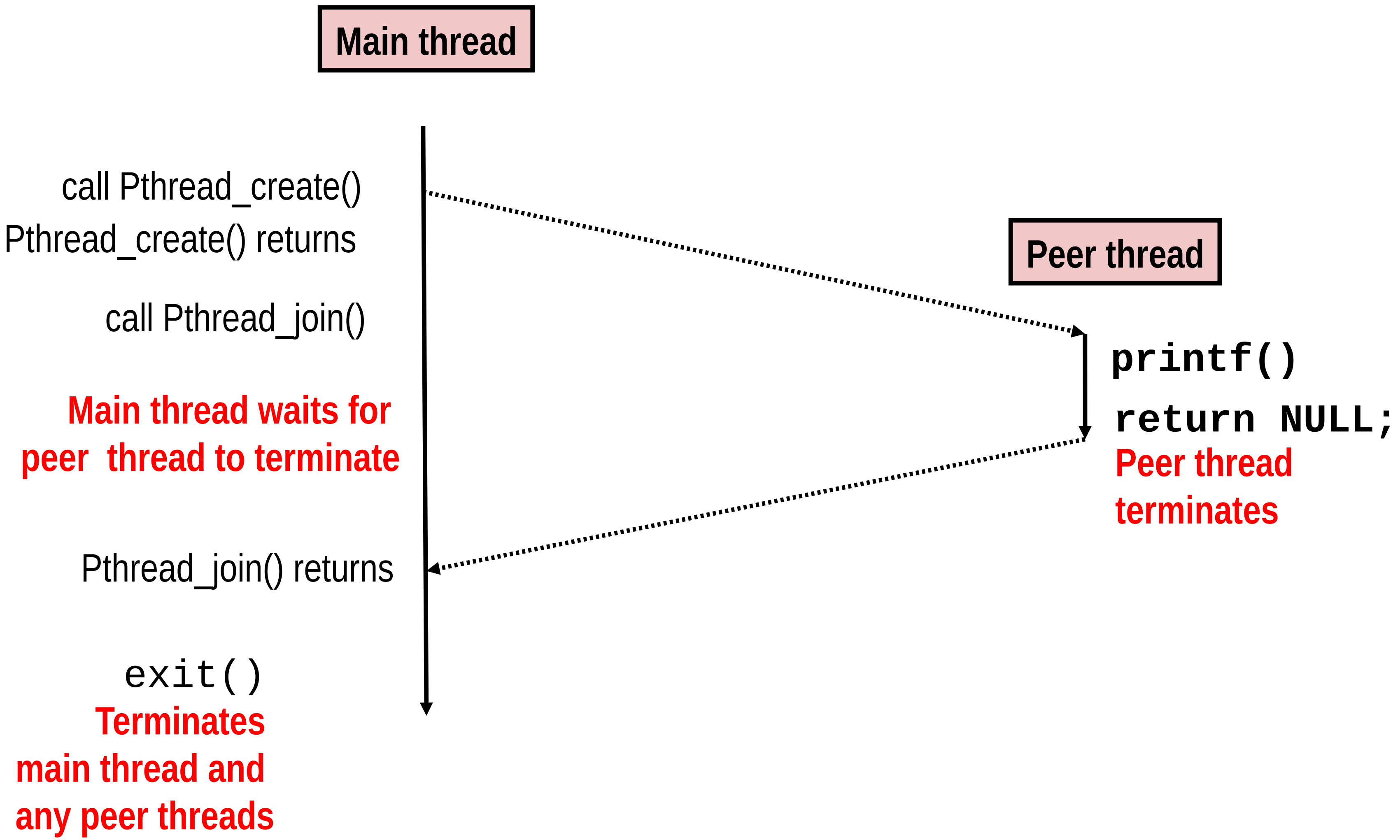
Thread arguments (void *p)

Return value (void **p)

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

hello.c

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 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.**

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**

Synchronizing Threads

- **Shared variables are handy...**

- **...but introduce the possibility of nasty *synchronization* errors.**

badcnt.c: Improper Synchronization

```c
/* 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

```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

```
    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

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!

One worry: Races

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

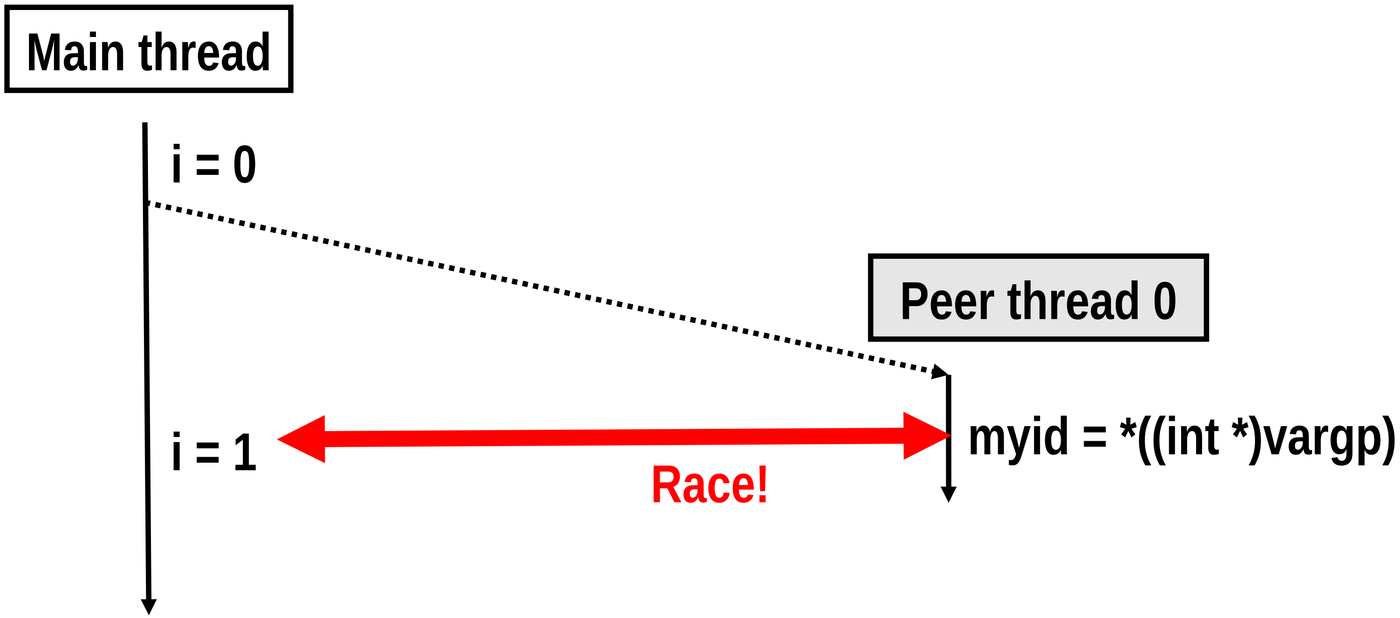
```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    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;
}
```

race.c

N threads are sharing i

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

Could this race really occur?

Main thread

```
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL,
                   thread,&i);
}
```

Peer thread

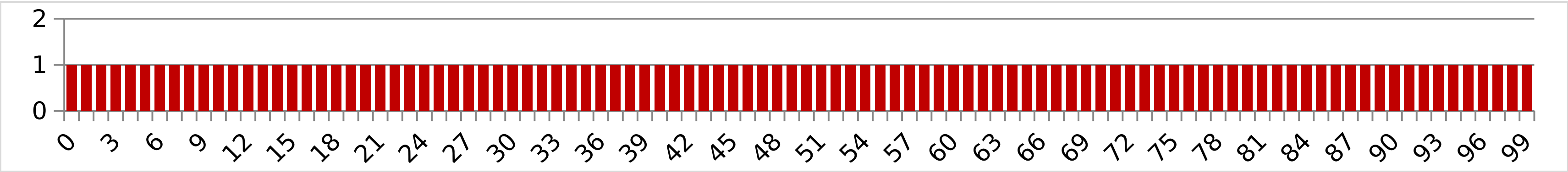
```
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    int i = *((int *)vargp);
    save_value(i);
    return NULL;
}
```

race.c

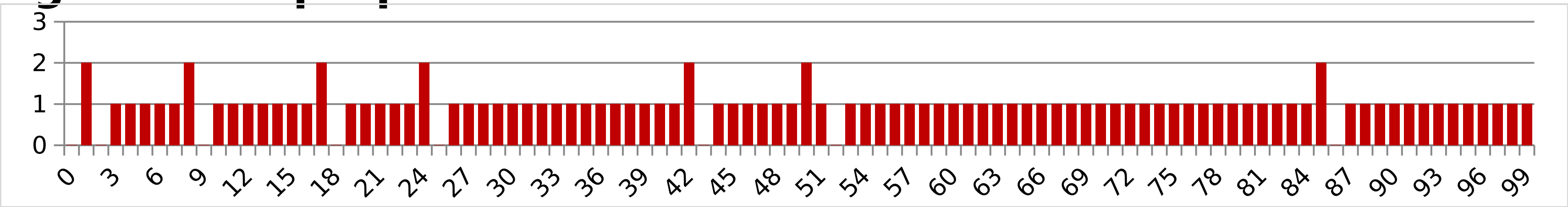
- **Race Test**
  - If no race, then each thread would get different value of i
  - Set of saved values would consist of one copy each of 0 through 99

Experimental Results

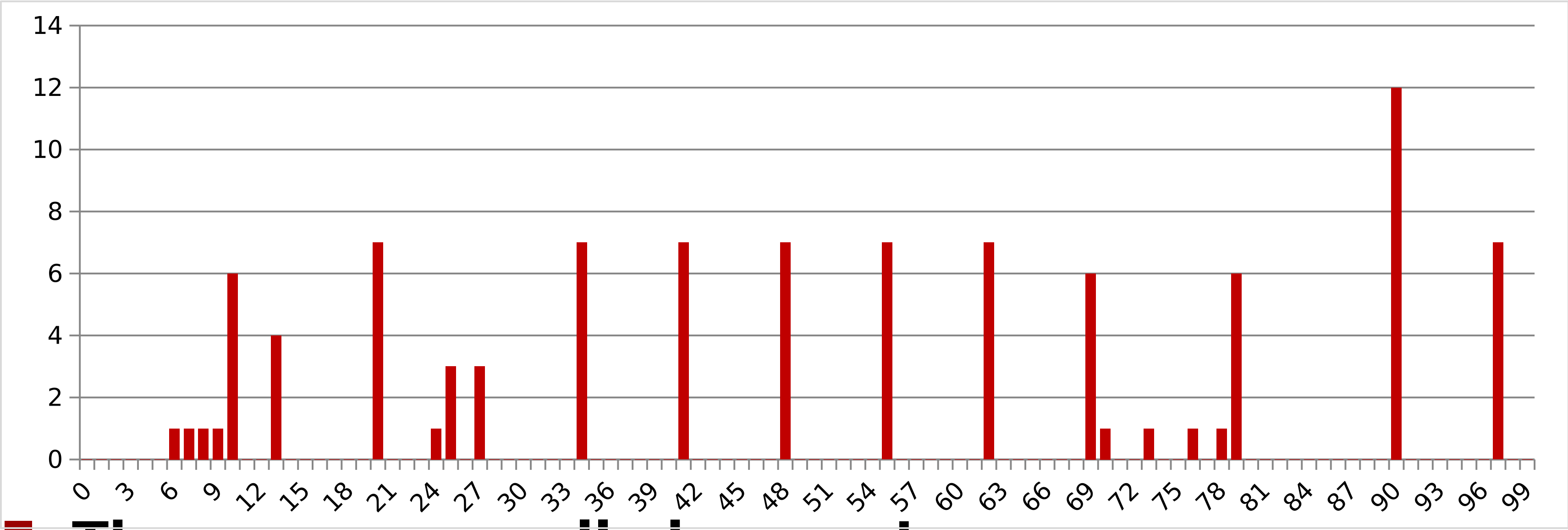
No Race



Single core laptop



Multicore device



- **The race can really happen!**

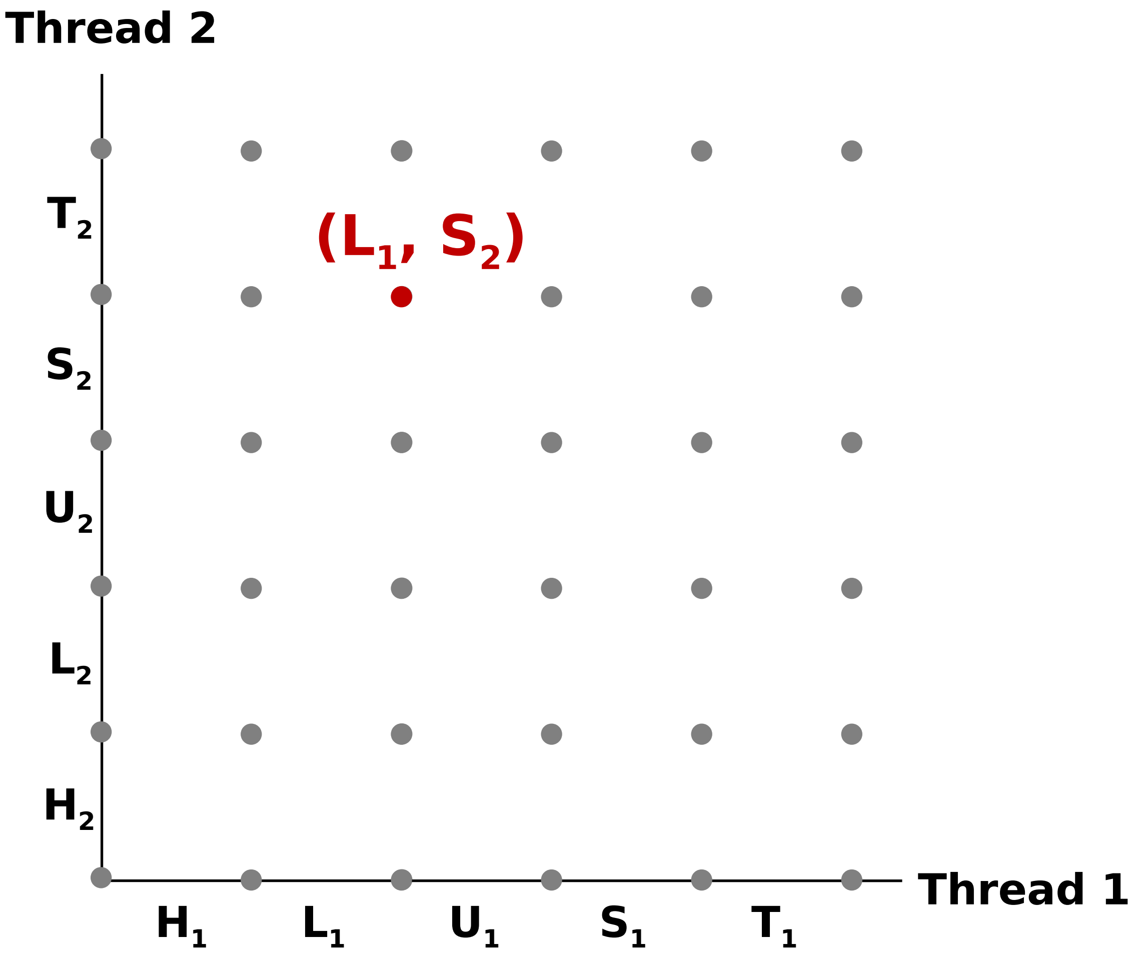
Race Elimination

```
/* 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;
}
```

norace.c

- **Avoid unintended sharing of state**

Progress Graphs



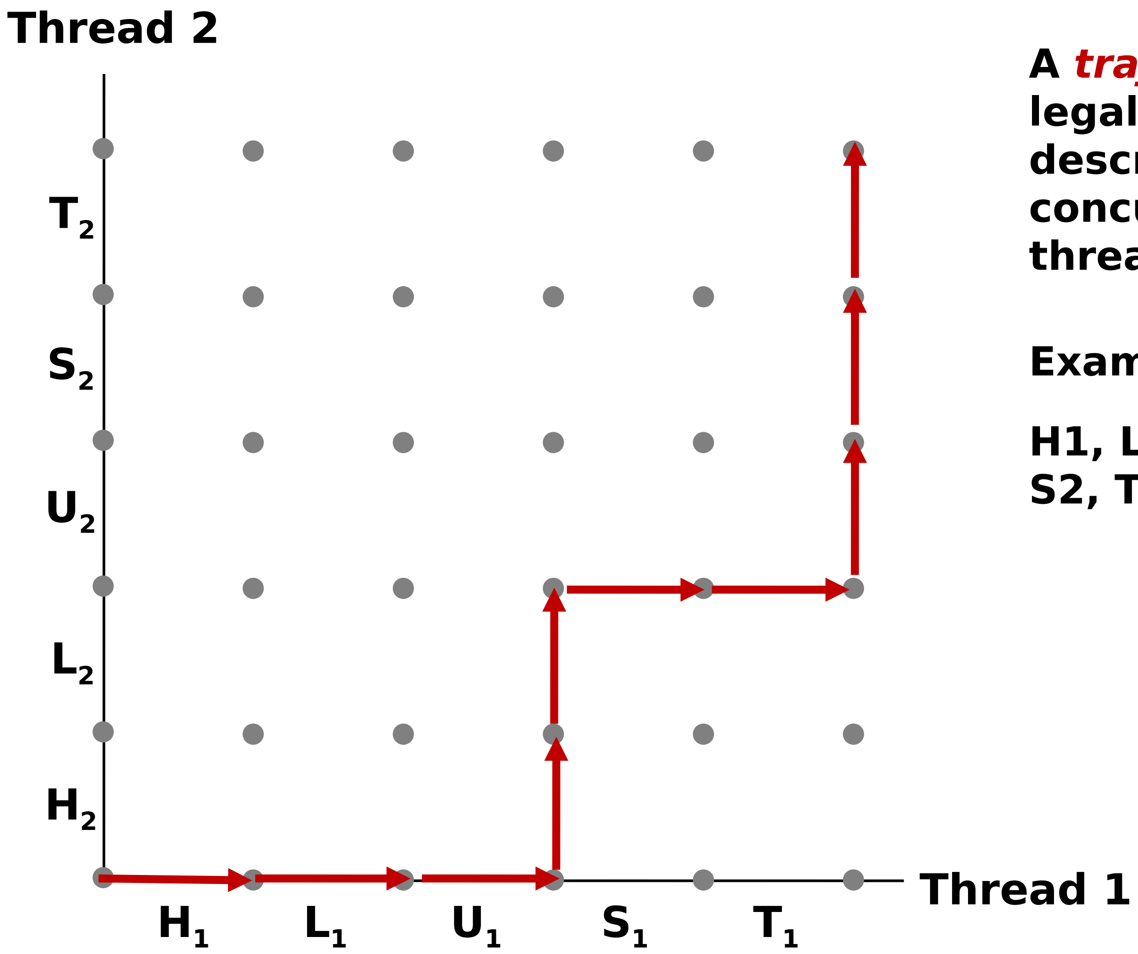
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

Enforcing Mutual Exclusion

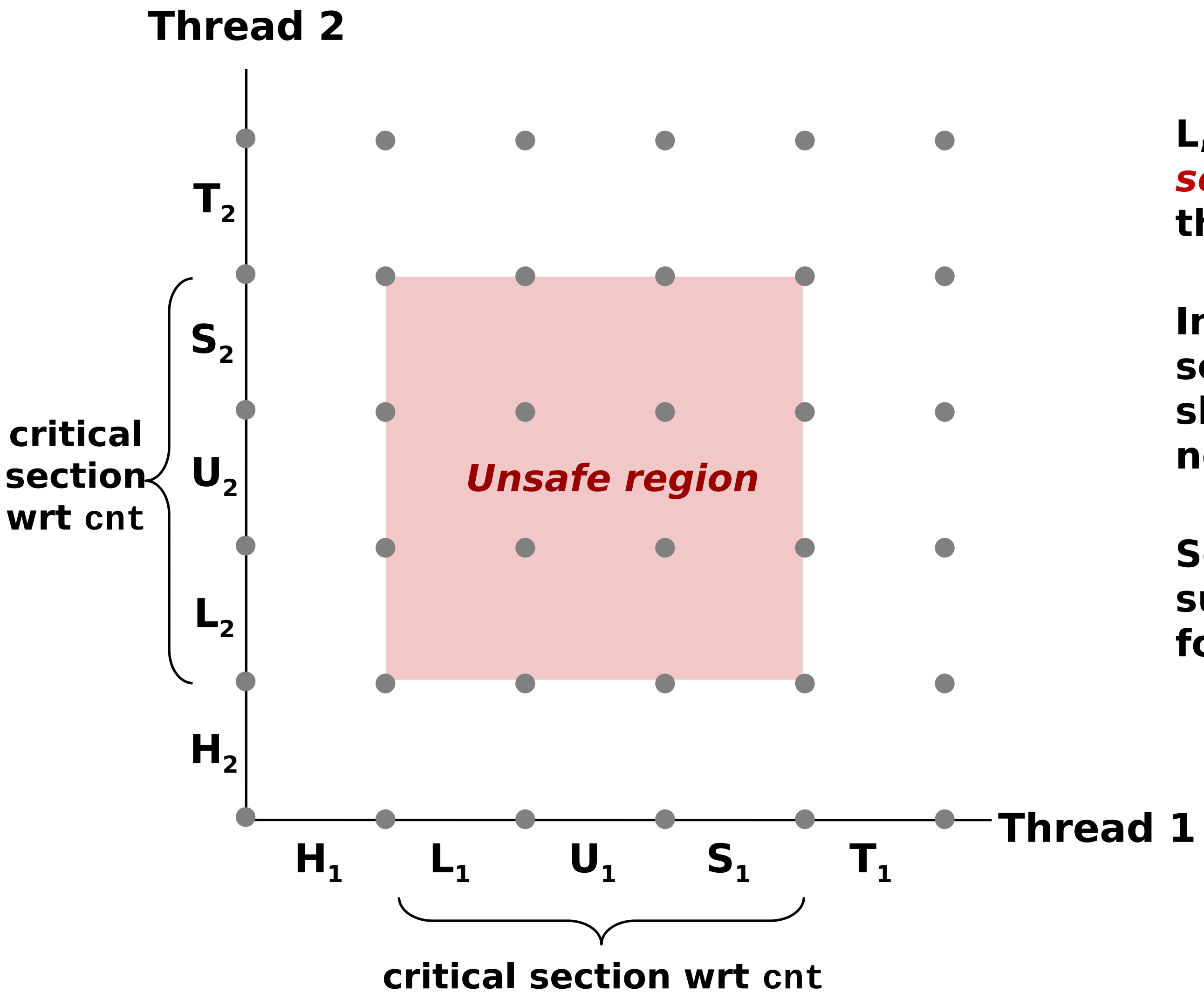
- ***Question:*** **How can we guarantee a safe trajectory?**

- **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)

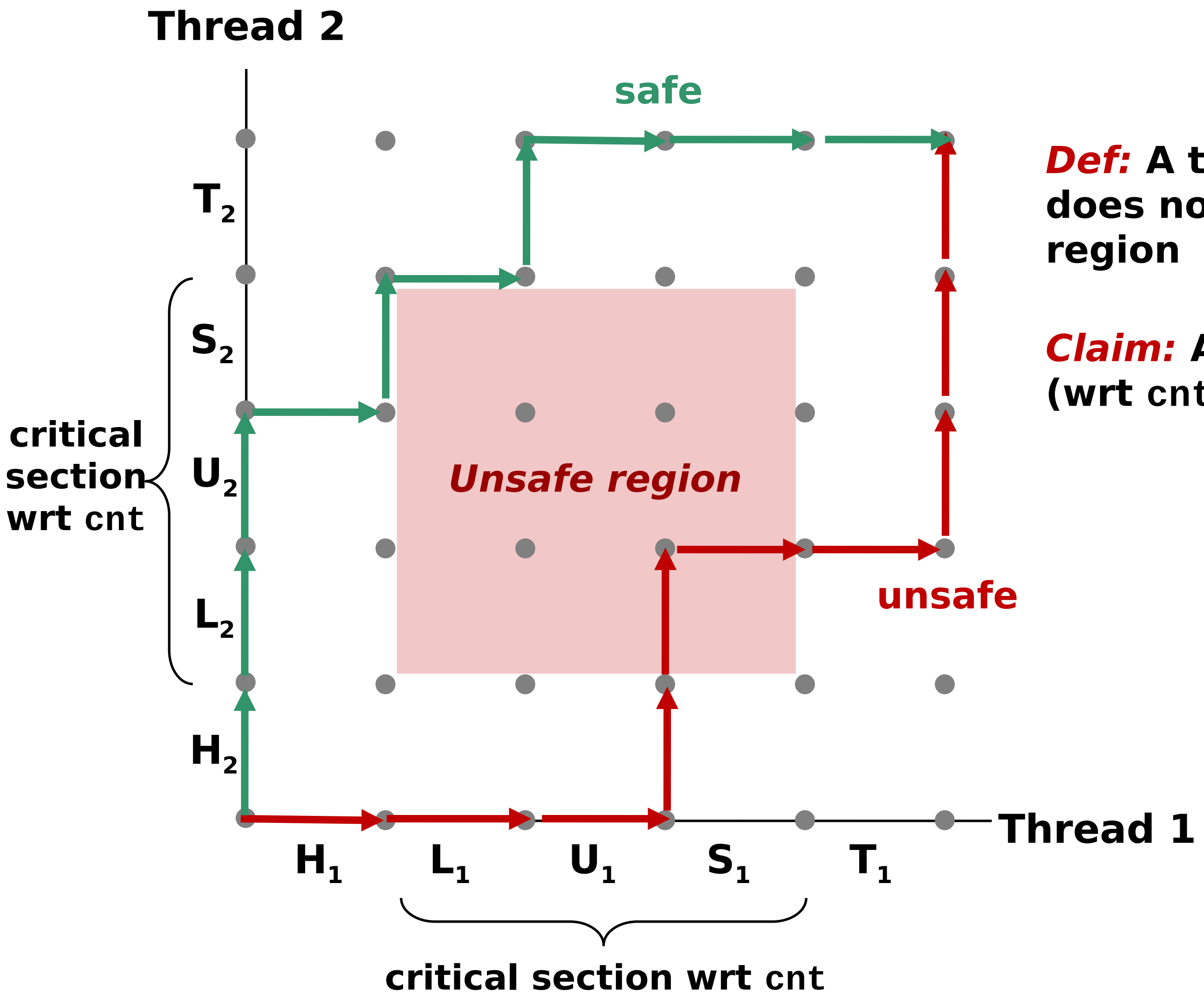
- **Other approaches**
  - Mutexes and condition variables from Pthreads
  - Monitors (Java) (boring languages are outside our scope)

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 >= 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

```c
/* 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

```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?

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*:**

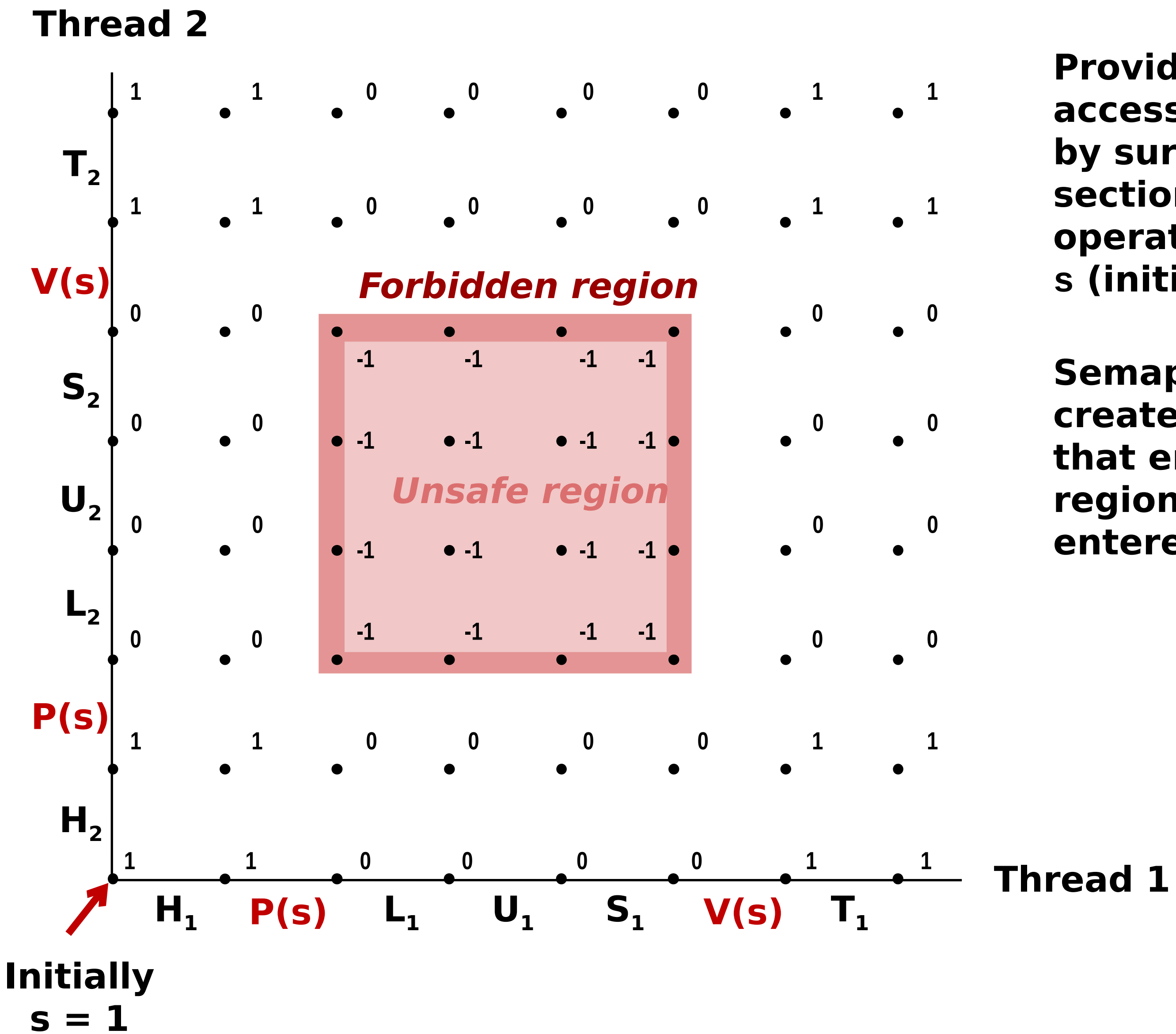
```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

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



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.

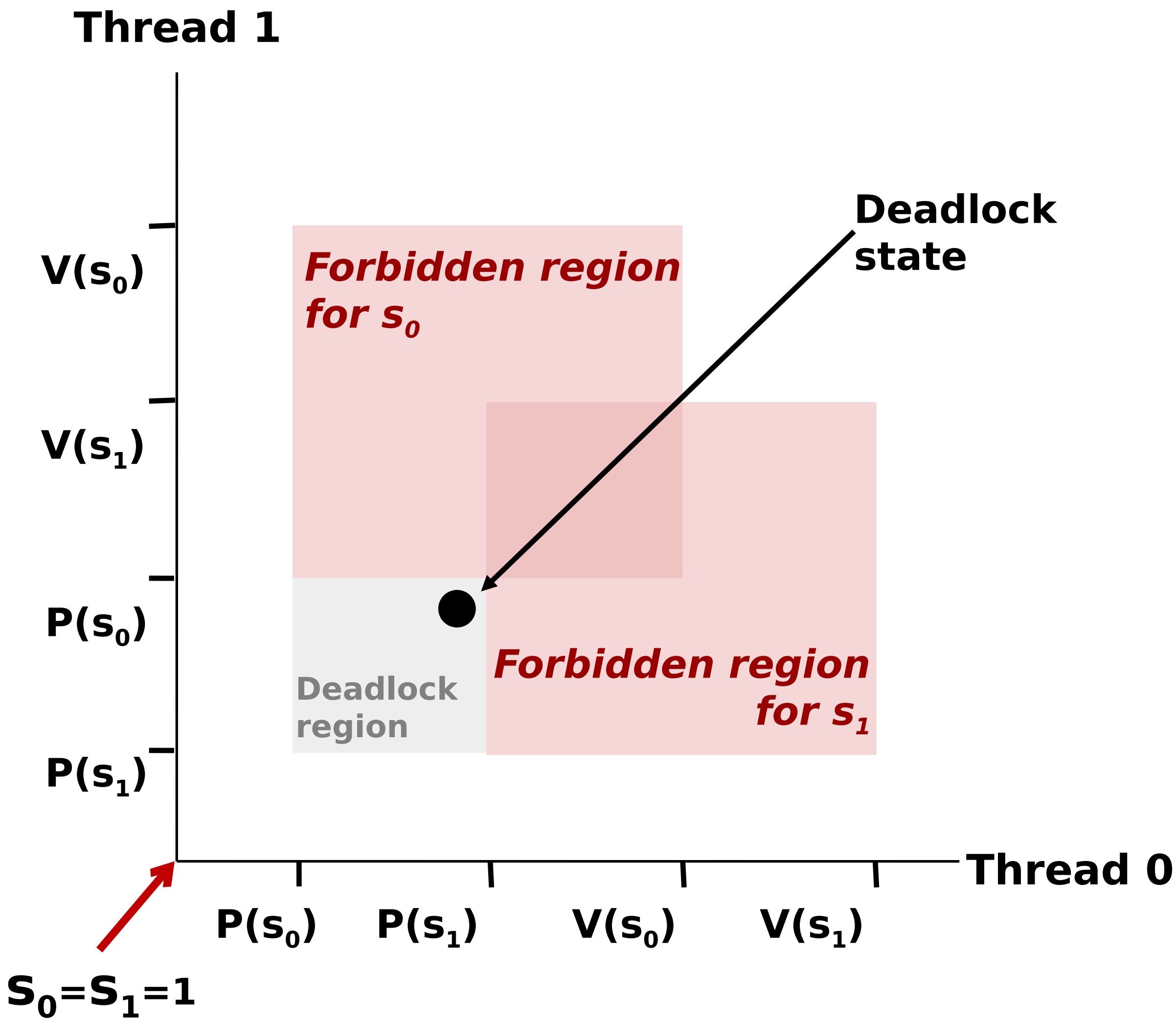
Deadlocking With Semaphores

```
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]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;

Deadlock Visualized in Progress Graph



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

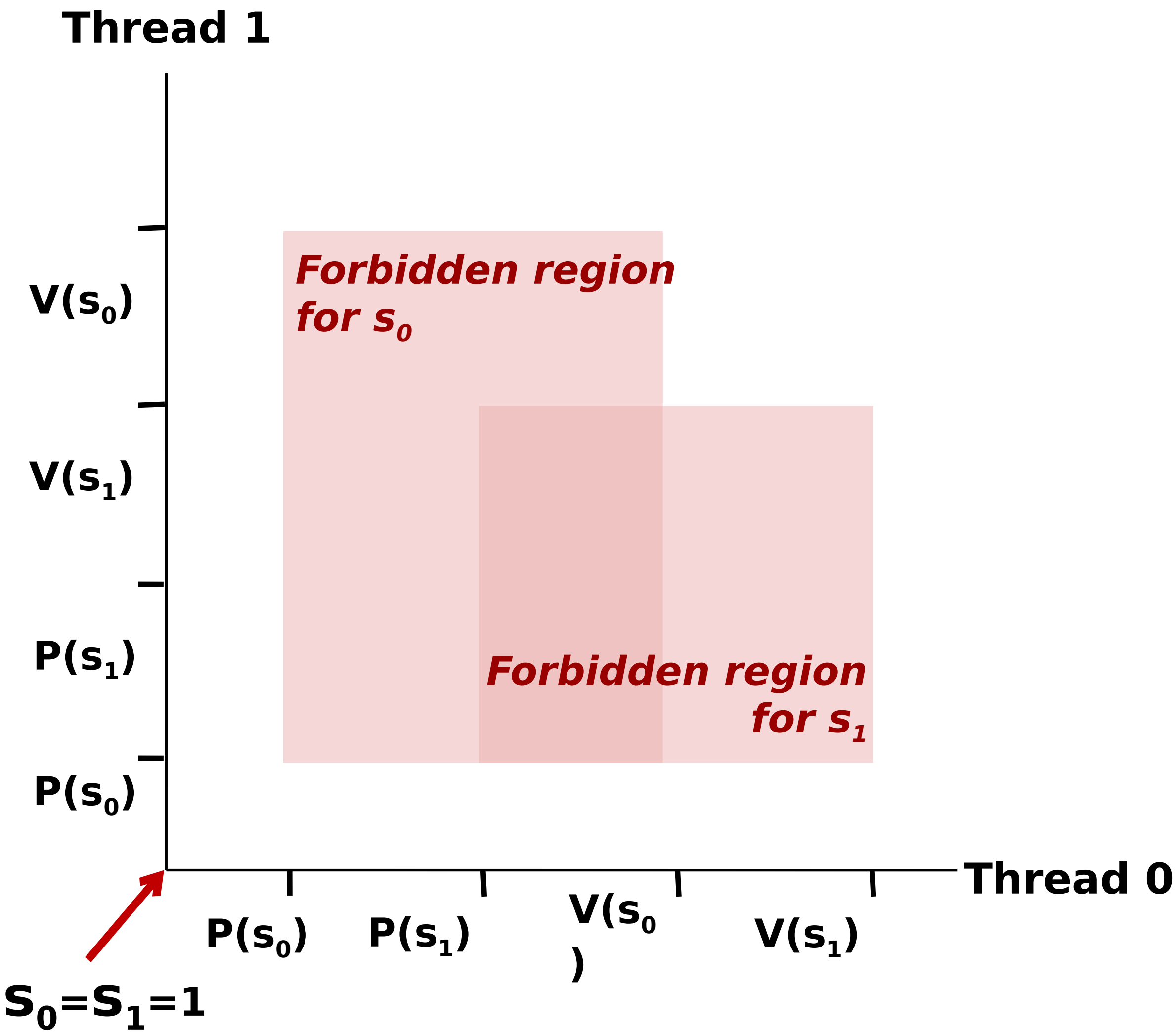
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], 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



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.