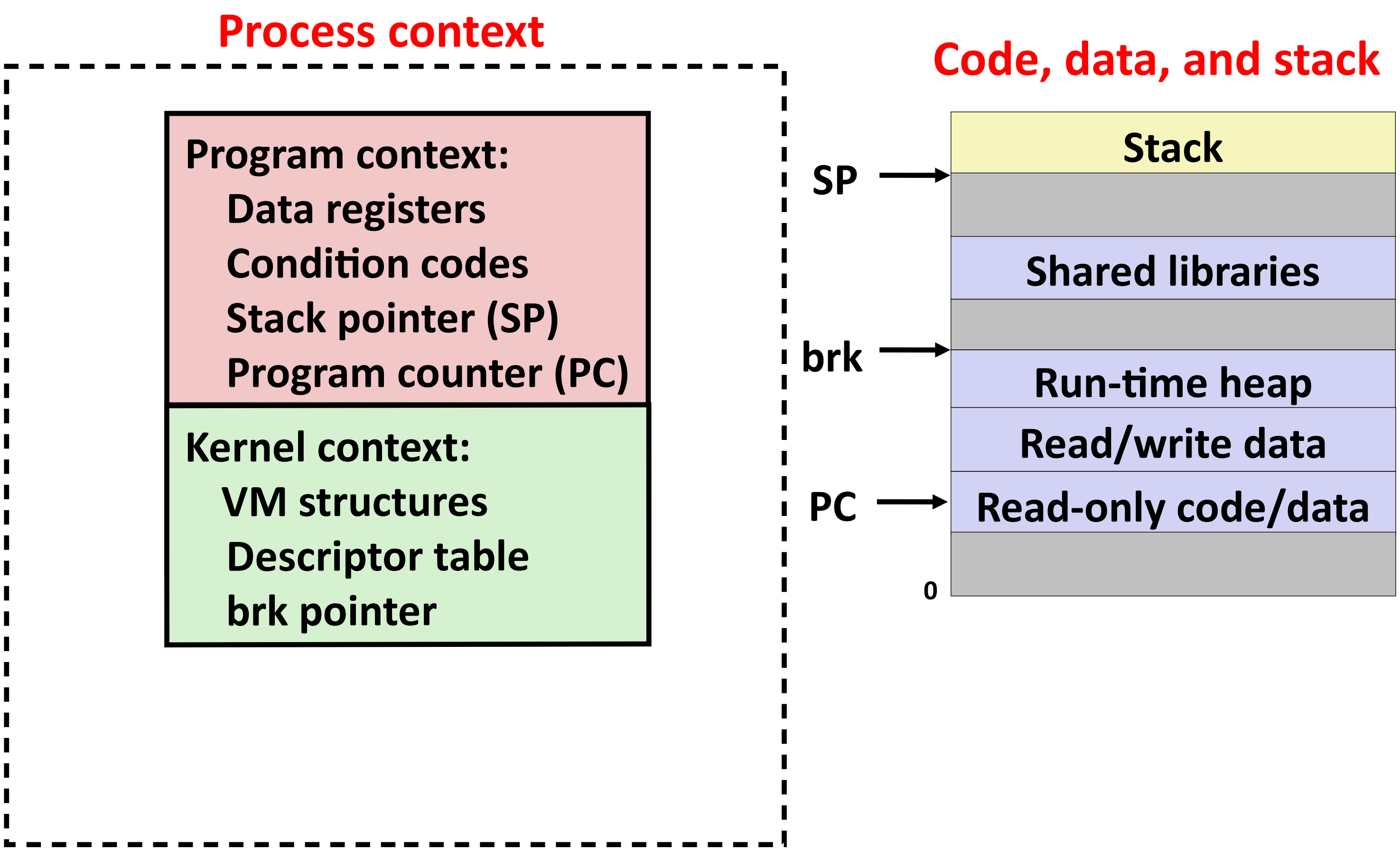
Today

- **Threads review**
- Sharing
- Mutual exclusion
- Semaphores

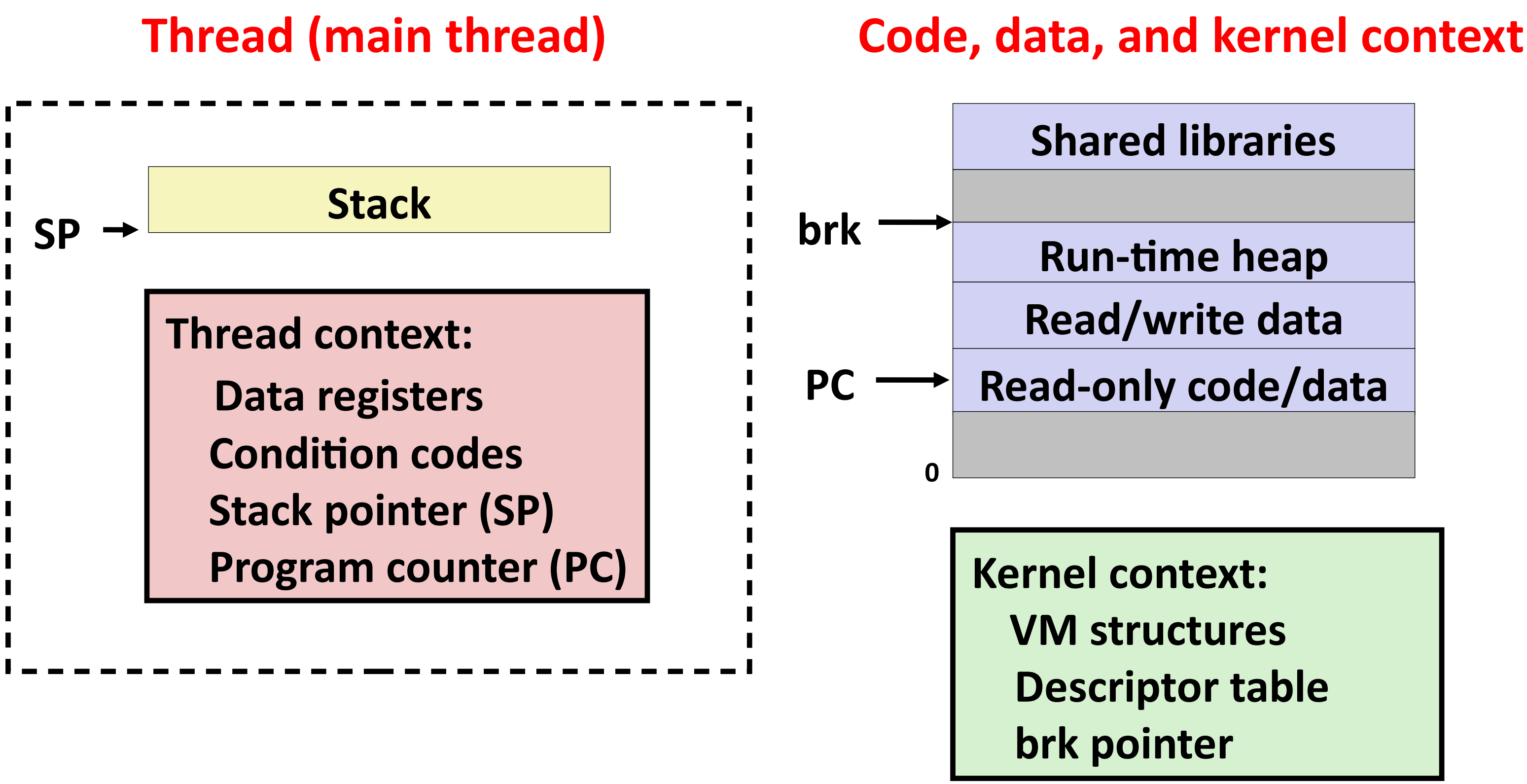
Traditional View of a Process

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



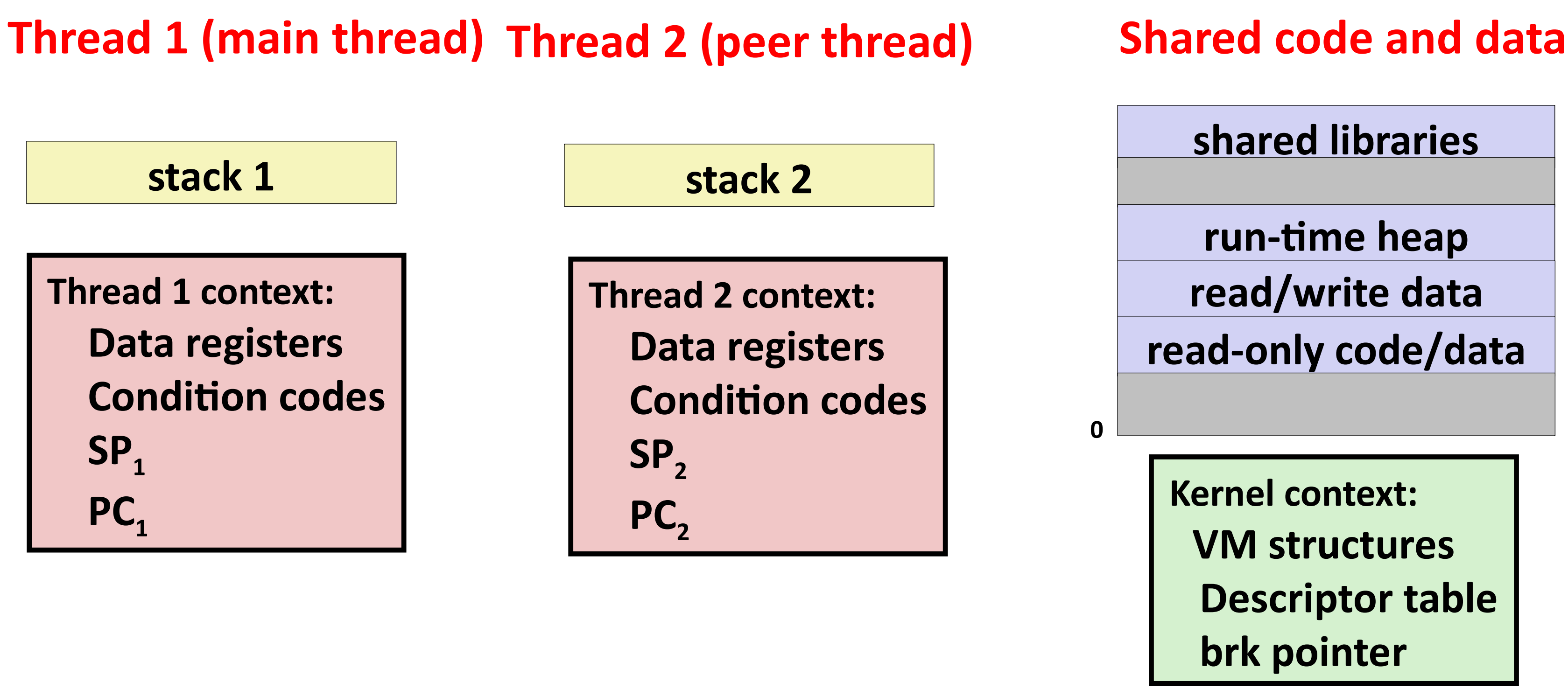
Alternate View of a Process

- 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 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)



Shared Variables in Threaded C Programs

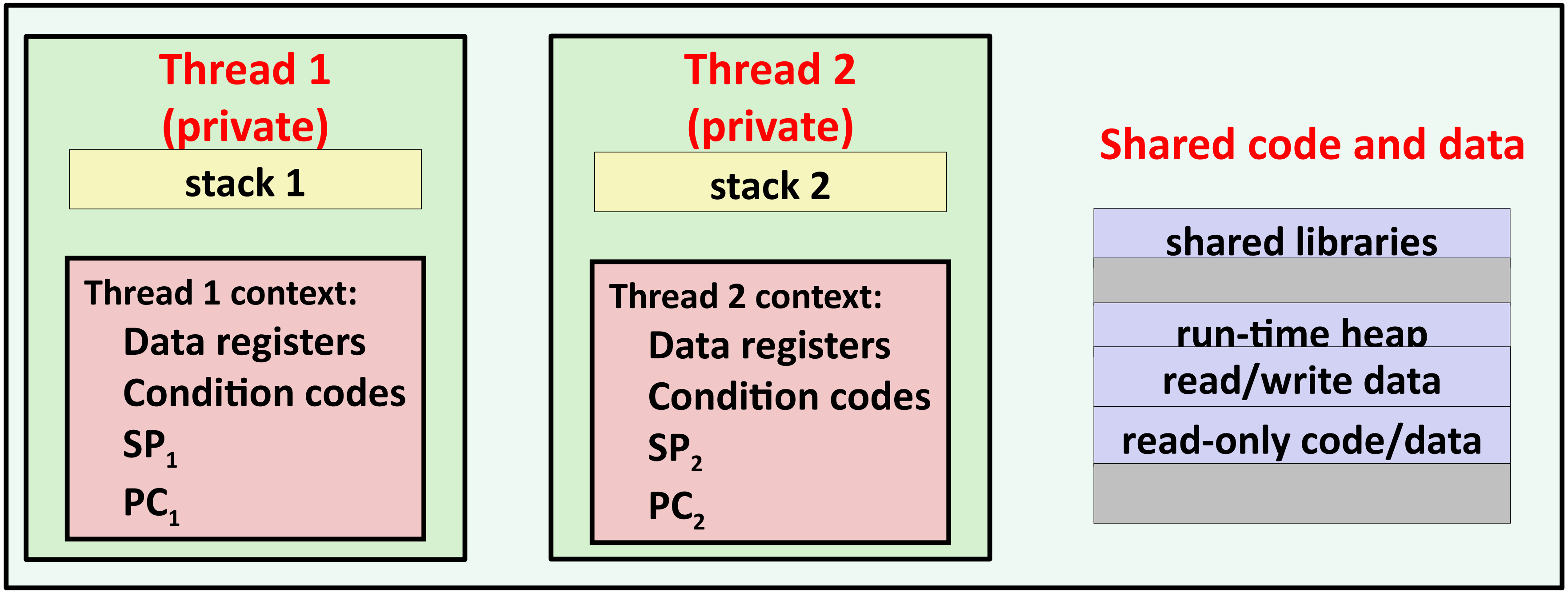
- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- ***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?

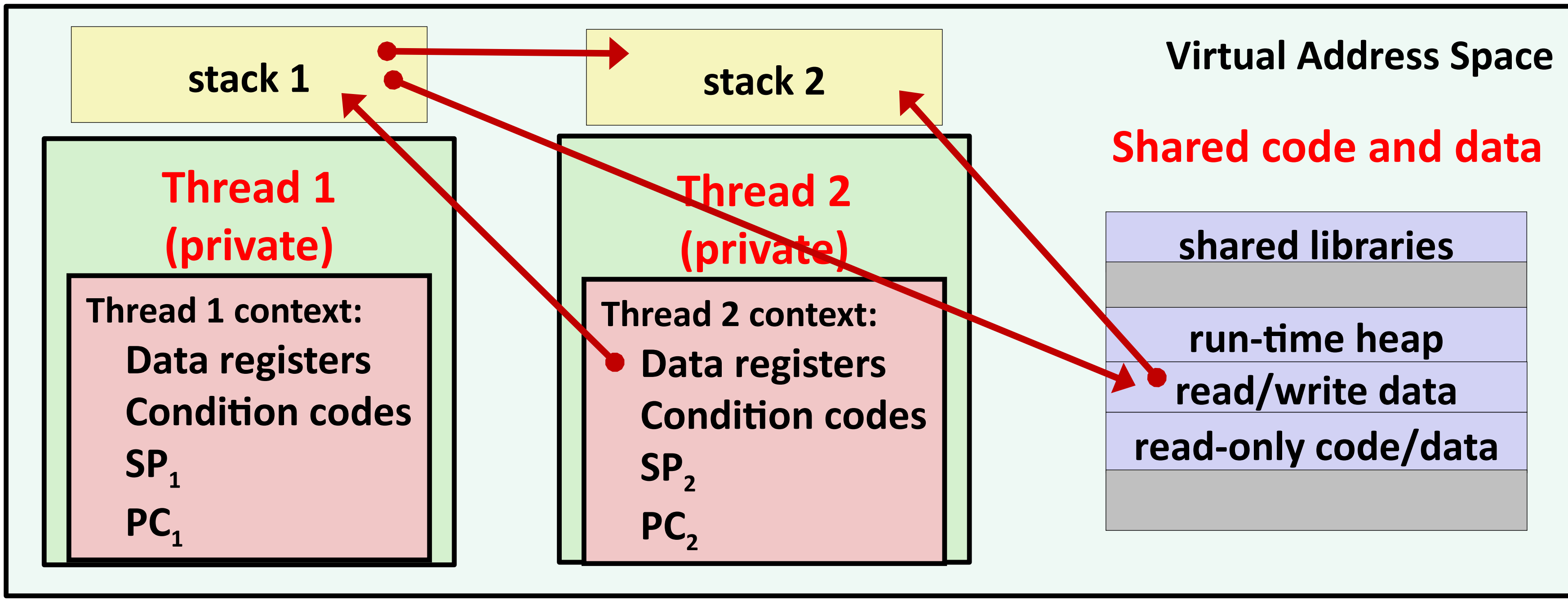
Threads Memory Model: Conceptual

- 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



Threads Memory Model: Actual

- **Separation of data 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(int argc, char *argv[])
{
  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

A common, but inelegant way to pass a single argument to a thread routine

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(int main, char *argv[])
{
  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

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  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);
}
```

```
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]:  %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
  return NULL;
}
```

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) {
  pthread_t tid1, tid2;

  long 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;
  long 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 (movq, testq, jle, movl)
- L_i : Load cnt
- U_i : Update cnt
- S_i : Store cnt
- T_i : Tail (addq, cmpq, jne)

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

OK

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

OK

Thread 1 critical section (green rows)

Thread 2 critical section (blue rows)

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)

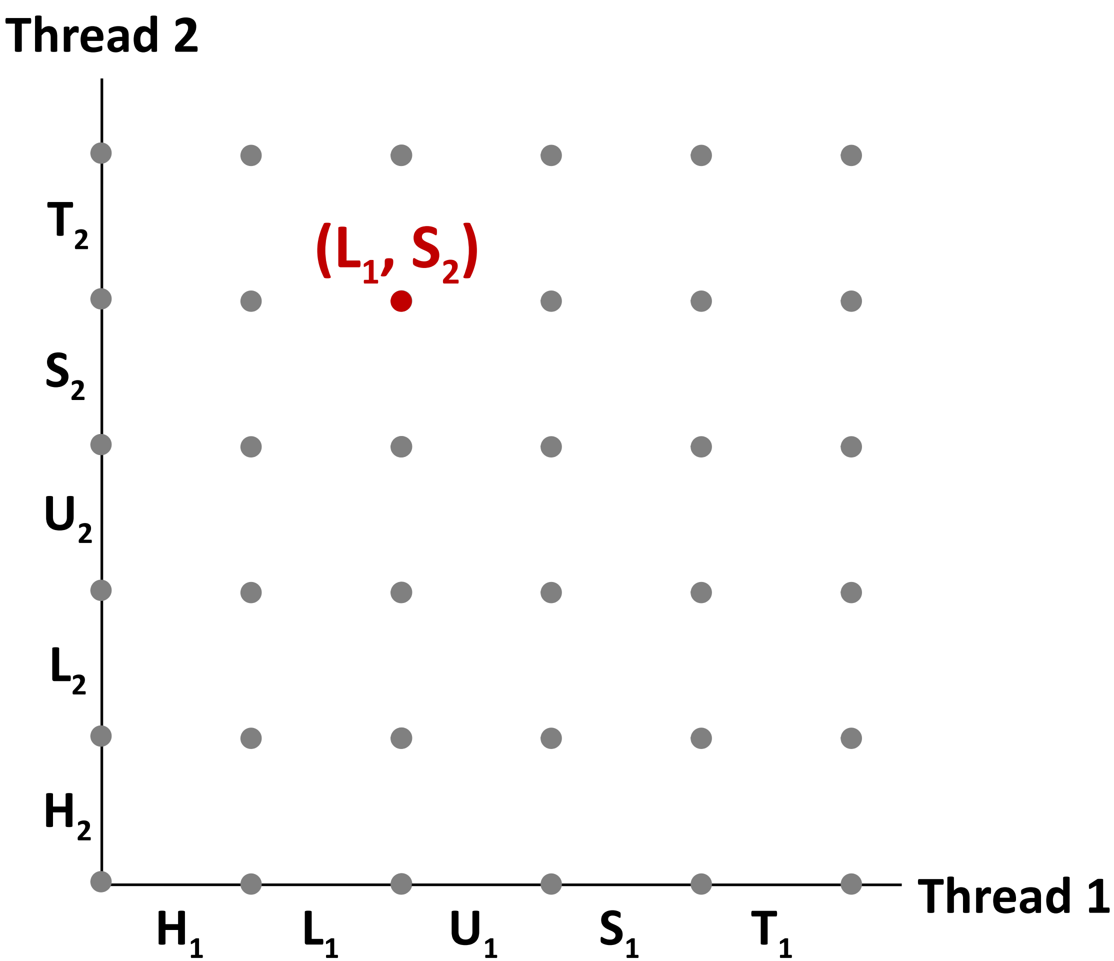
- How about this ordering?

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



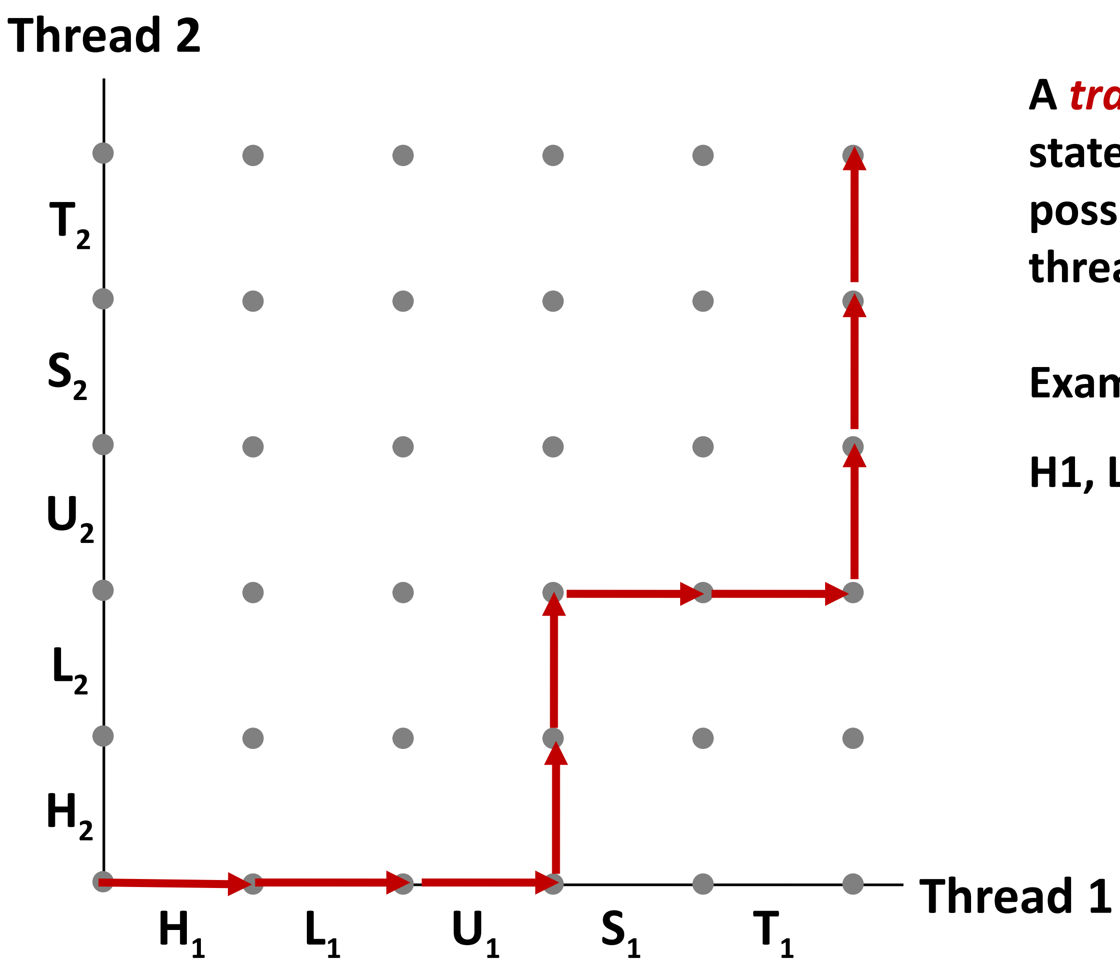
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

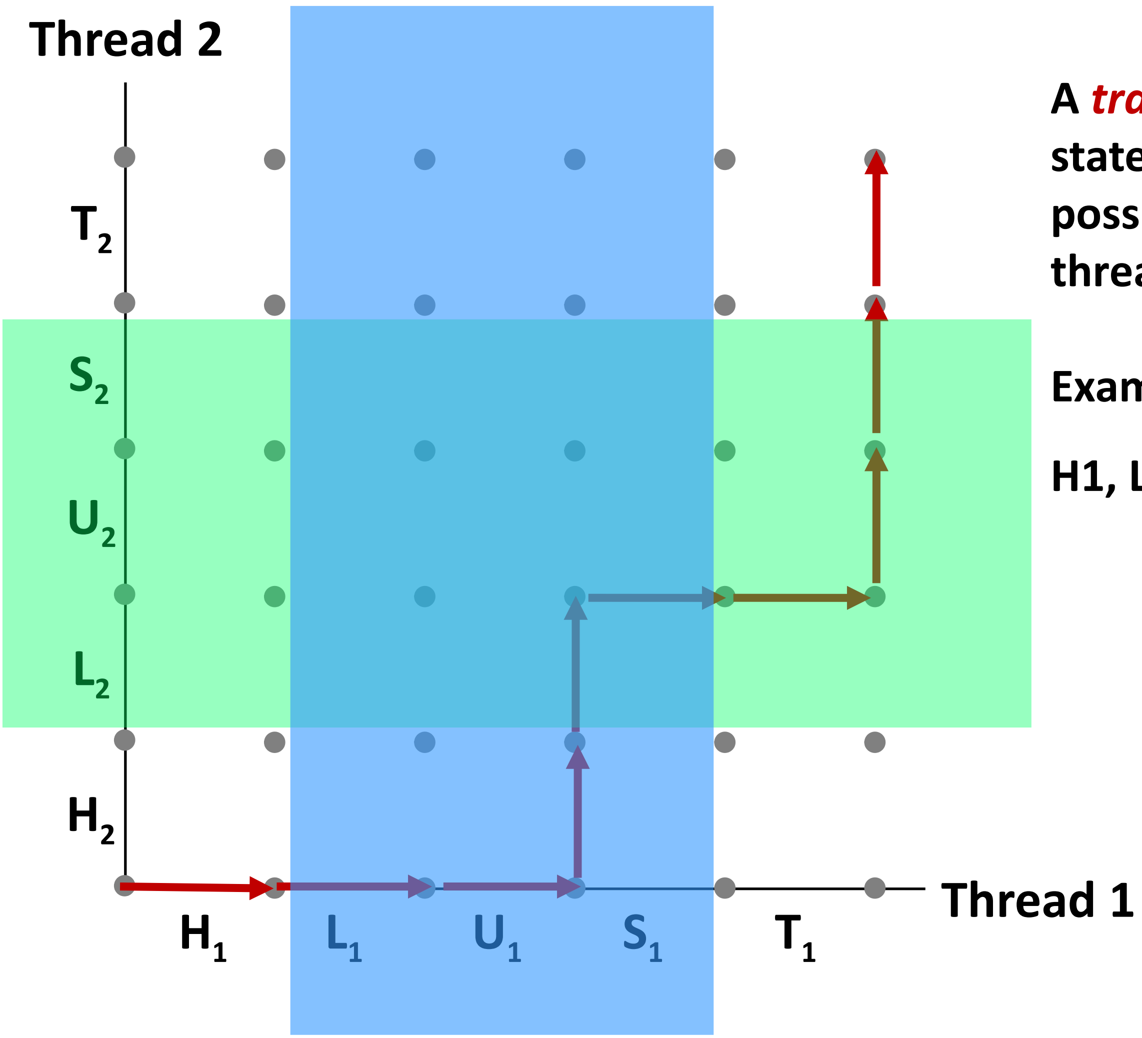


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

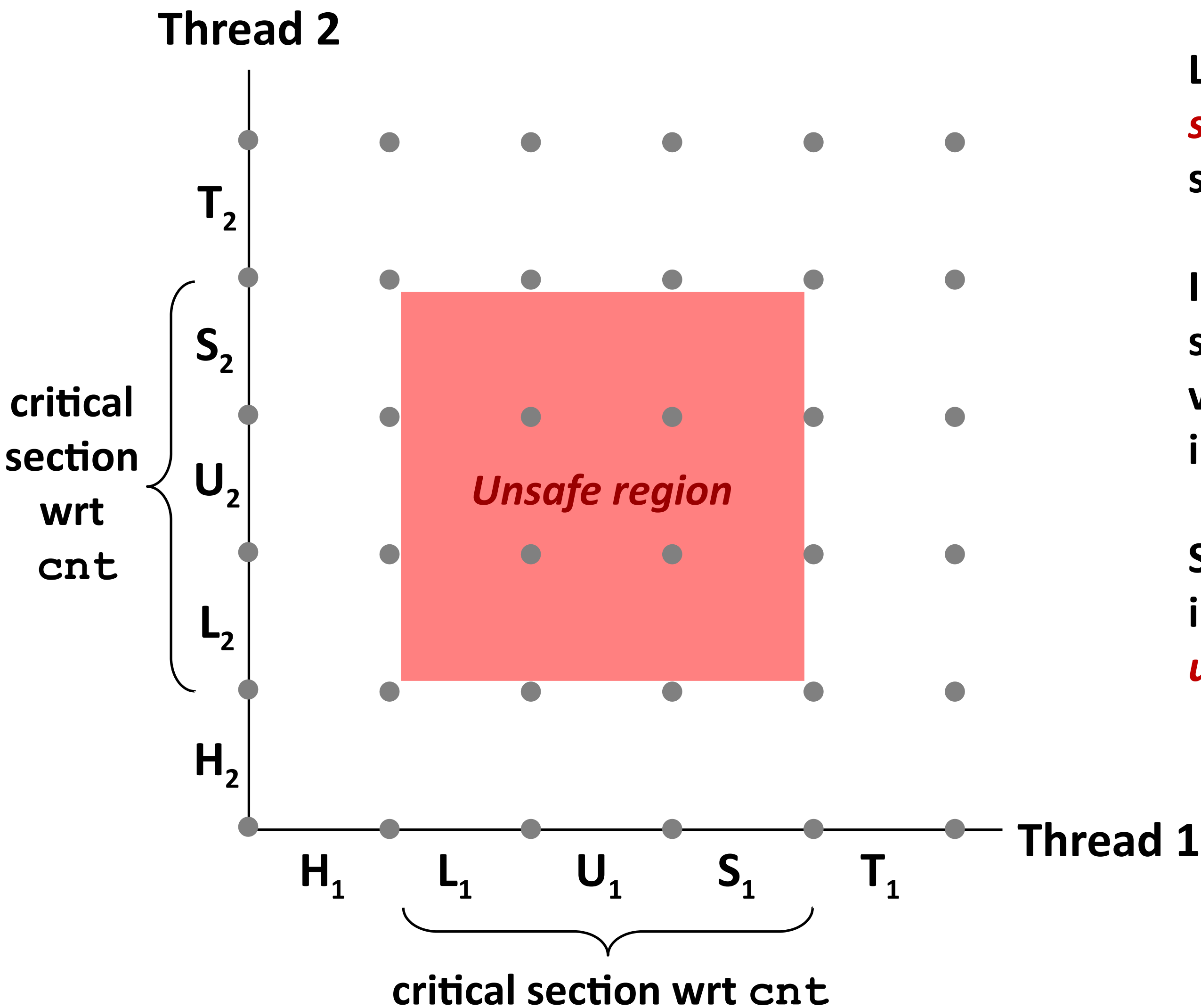


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

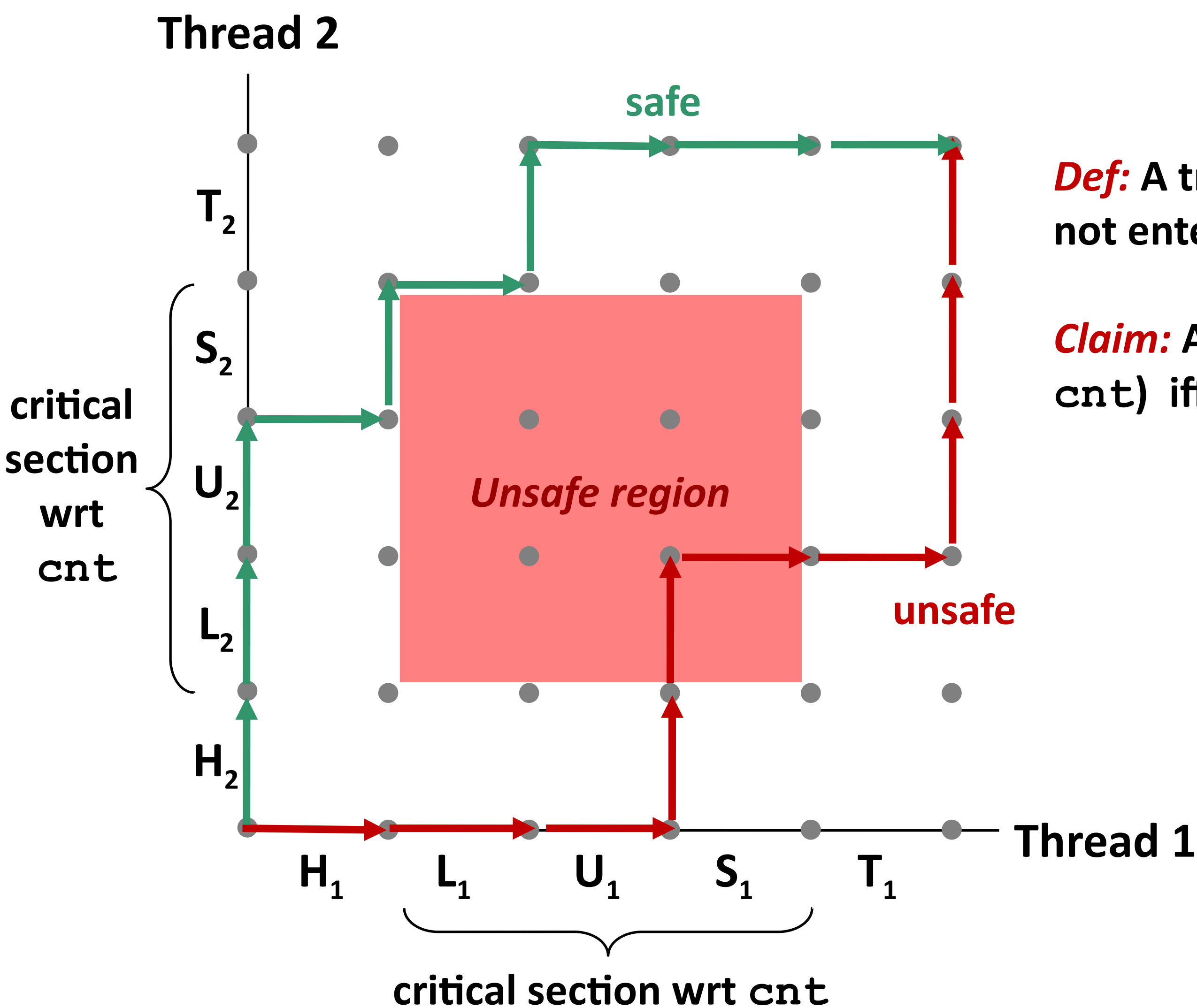
H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

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

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;
}
```

Variable	main	thread1	thread2
cnt	yes*	yes	yes
niters.m	yes	no	no
tid1.m	yes	no	no
i.1	no	yes	no
i.2	no	no	yes
niters.1	no	yes	no
niters.2	no	no	yes

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)

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 >= 0)***

Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable**

- **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`

- **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

```
/* 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);
}
```

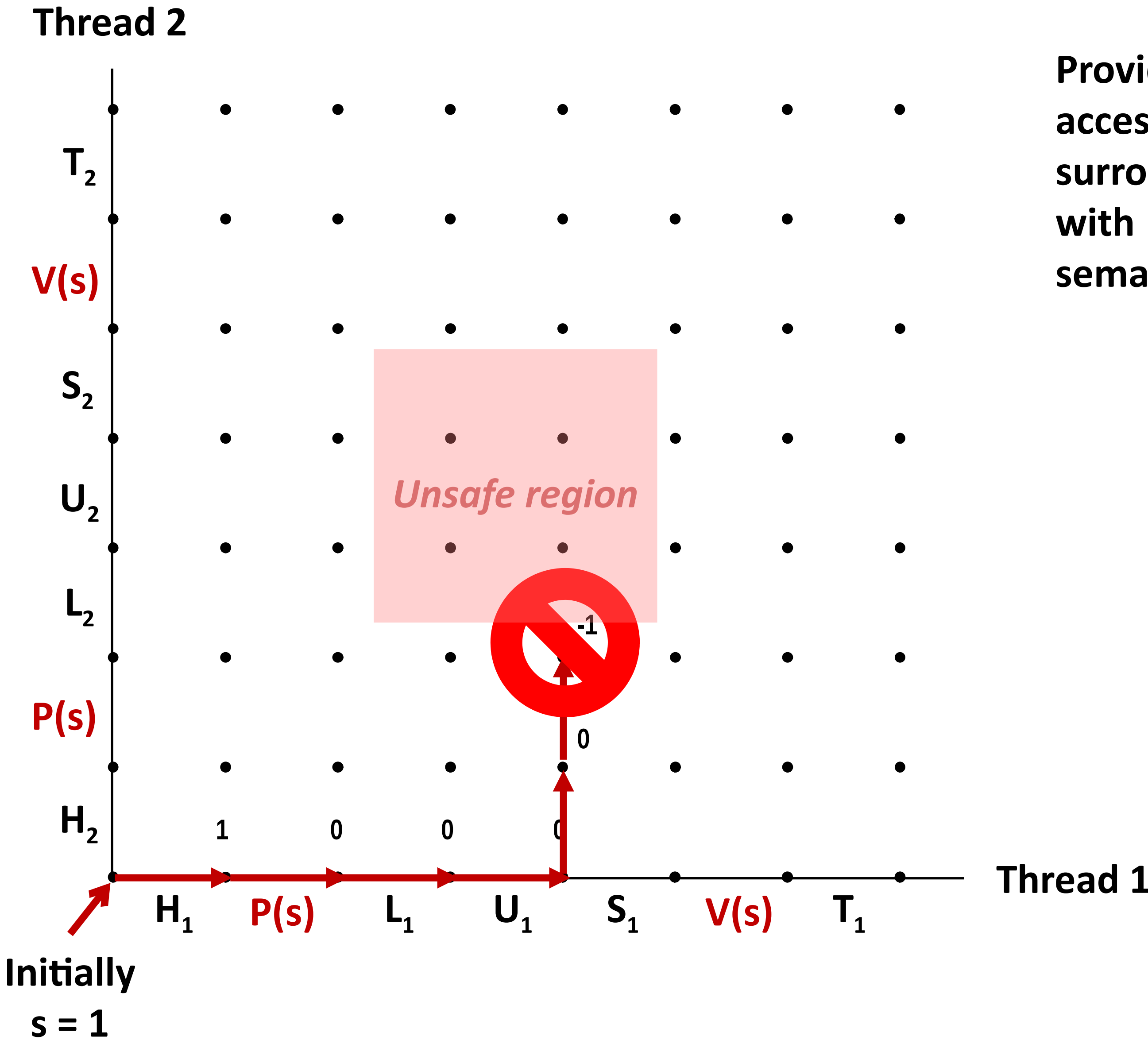
goodcnt.c

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

Function	badcnt	goodcnt
Time (ms) niters = 10^6	12	450
Slowdown	1.0	37.5

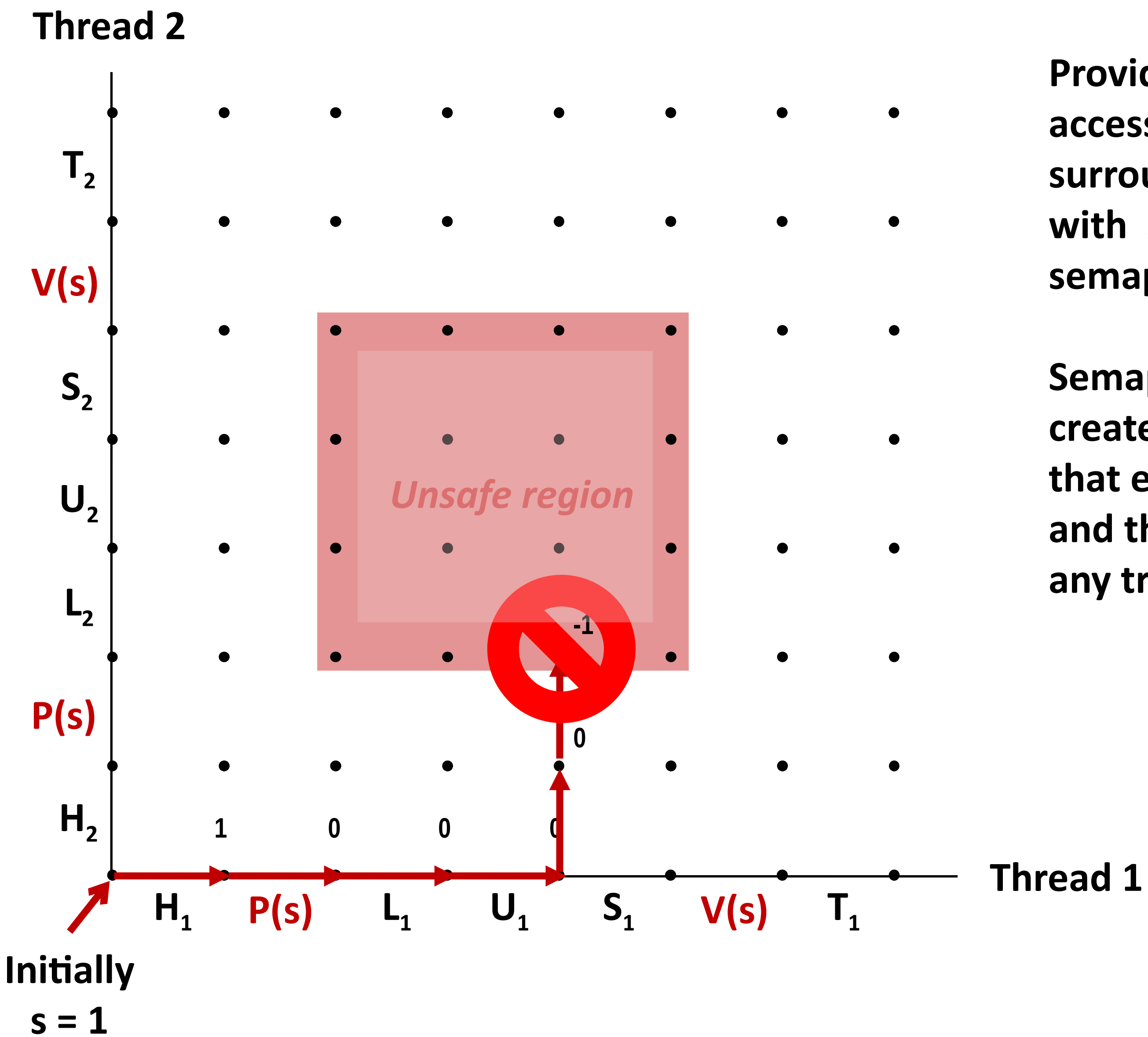
Warning: It's orders of magnitude slower than badcnt.c.

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)

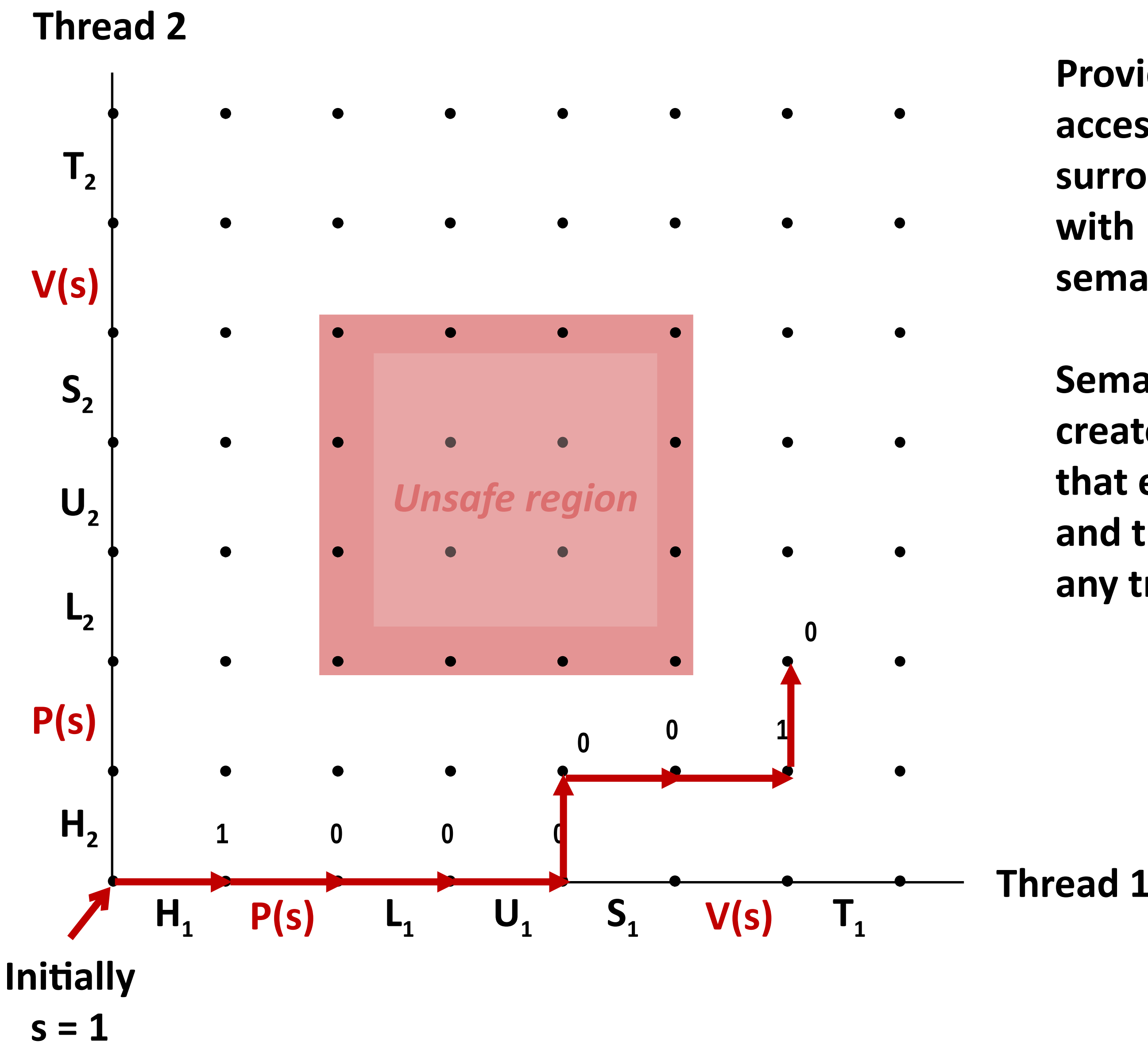
Why Mutexes Work



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.

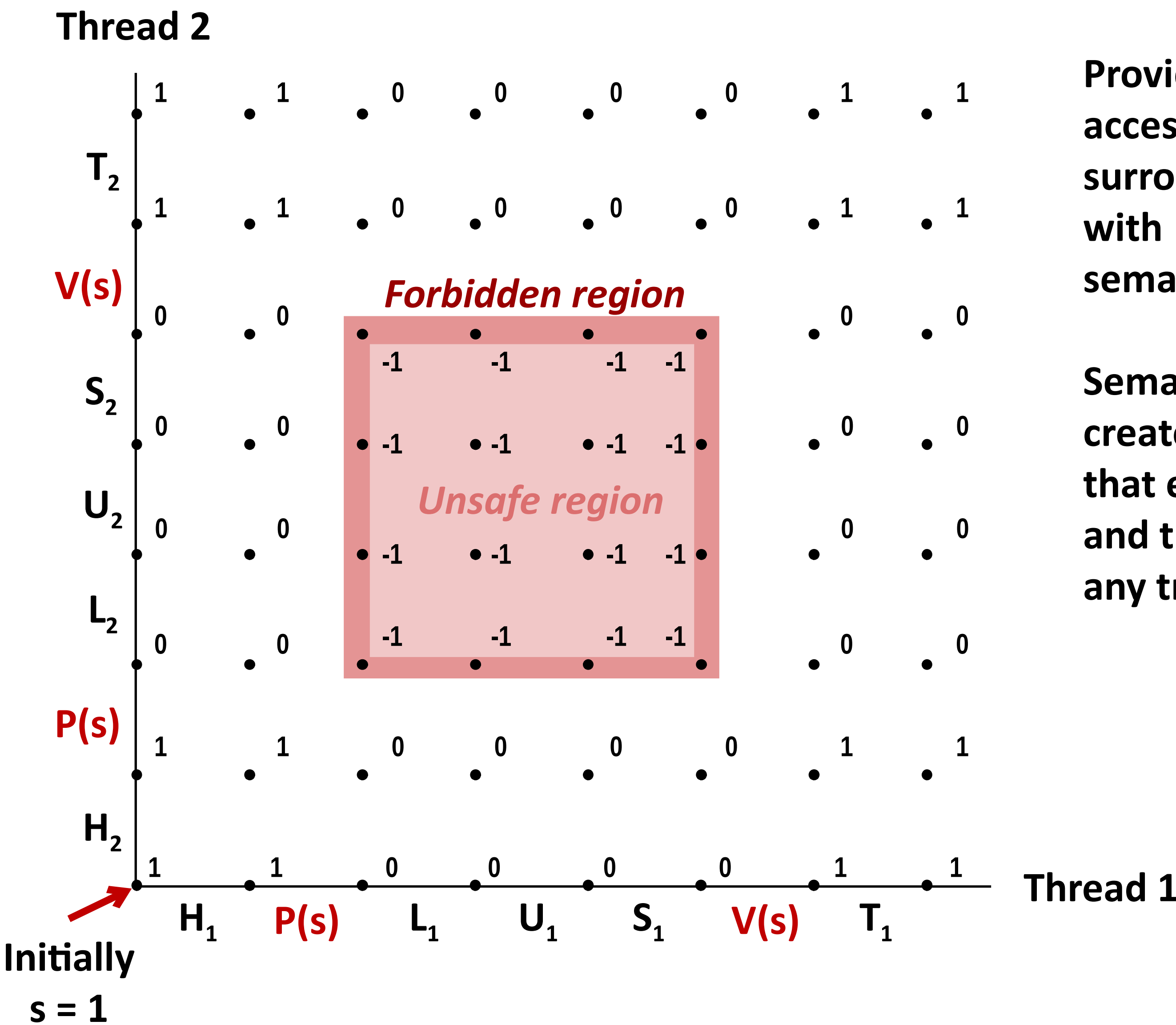
Why Mutexes Work



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.

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.

Binary Semaphores

- **Mutex is special case of semaphore**
  - Value either 0 or 1
- **Pthreads provides `pthread_mutex_t`**
  - Operations: lock, unlock
- **Recommended over general semaphores when appropriate**

goodmcnt.c: Mutex Synchronization

- Define and initialize a mutex for the shared variable cnt:

```
volatile long cnt = 0;  /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

- Surround critical section with *lock* and *unlock*:

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

goodcnt.c

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

Function	badcnt	goodcnt	goodmcnt
Time (ms) niters = 10^6	12	450	214
Slowdown	1.0	37.5	17.8

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.**