Synchronization: Basics

CS230: System Programming
18th Lecture

Instructor:

Jongse Park

Today

- Sharing
- Mutual exclusion
- Semaphores

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*"
- 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?
- *Def:* A variable `x` is *shared* if and only if multiple threads reference some instance of `x`.

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 */

int main()
{
    int 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);
}
```

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

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

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])

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]
)

```
char **ptr;  /* global */

int main()
{
    int 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);
}
```

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

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

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

- 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

- Sharing
- Mutual exclusion
- Semaphores

badcnt.c: Improper Synchronization

```
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
  int niters = atoi(argv[1]);
  pthread_t tid1, tid2;

  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=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}
```

```
/* Thread routine */
void *thread(void *vargp)
{
  int i, niters = *((int *)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++;
```

Corresponding assembly code

```
    movl (%rdi),%ecx
    movl $0,%edx
    cmpl %ecx,%edx
    jge .L13
.L11:
    movl cnt(%rip),%eax
    incl %eax
    movl %eax,cnt(%rip)
    incl %edx
    cmpl %ecx,%edx
    jl .L11
.L13:
```

Head (H_i)

Load `cnt` (L_i)
Update `cnt` (U_i)
Store `cnt` (S_i)

Tail (T_i)

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
  - $\%eax_i$ is the content of %eax in thread i's context

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	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

Thread 2 critical section

Concurrent Execution (cont)

- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	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)

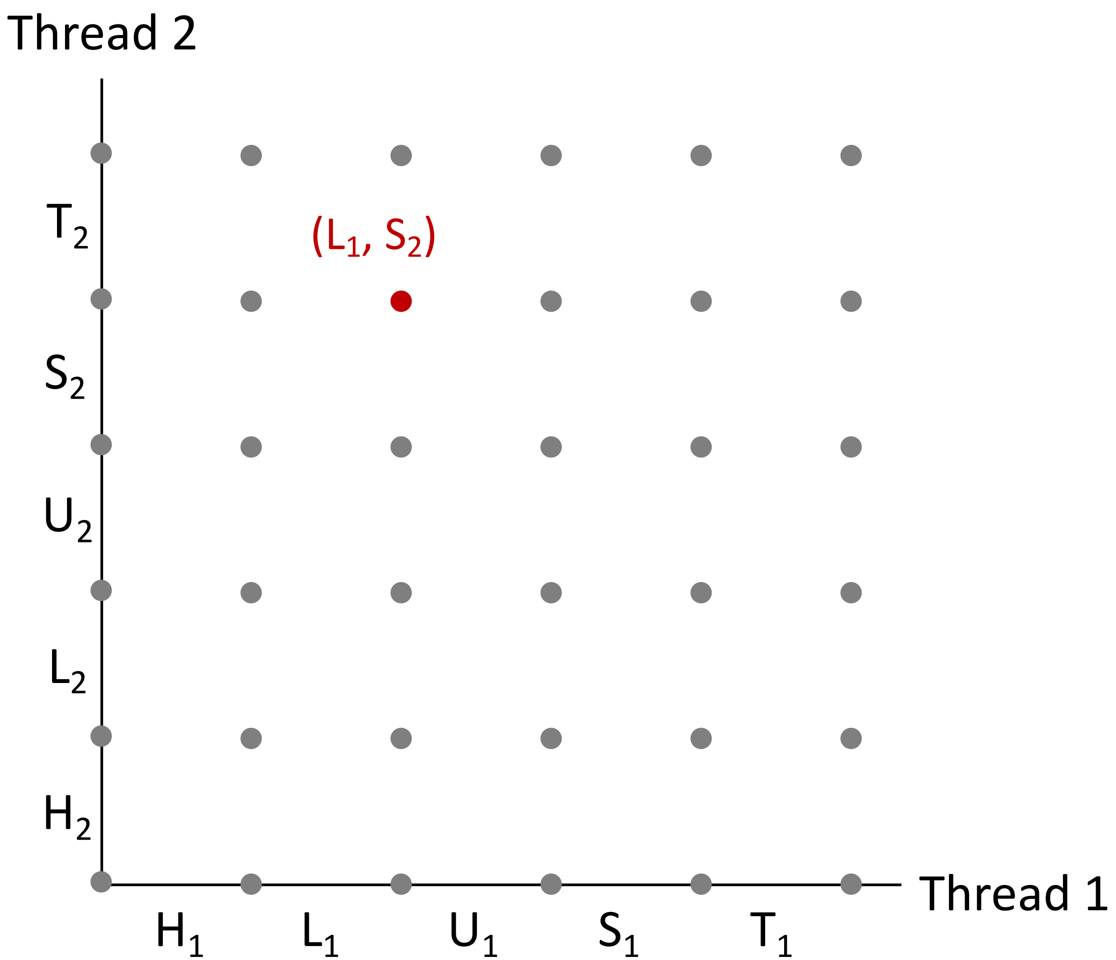
- How about this ordering?

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	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			
2	T_2			1

Oops!

- We can analyze the behavior using a *progress graph*

Progress Graphs



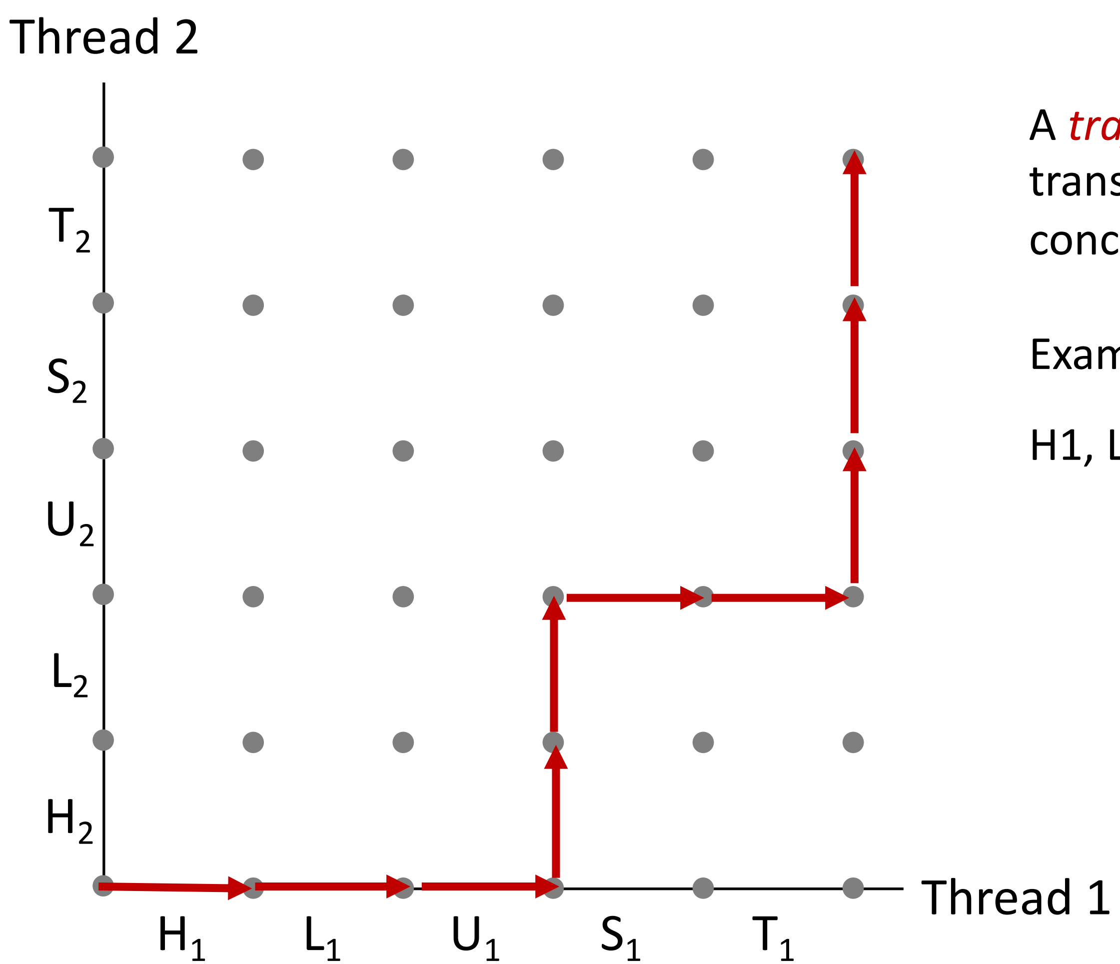
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

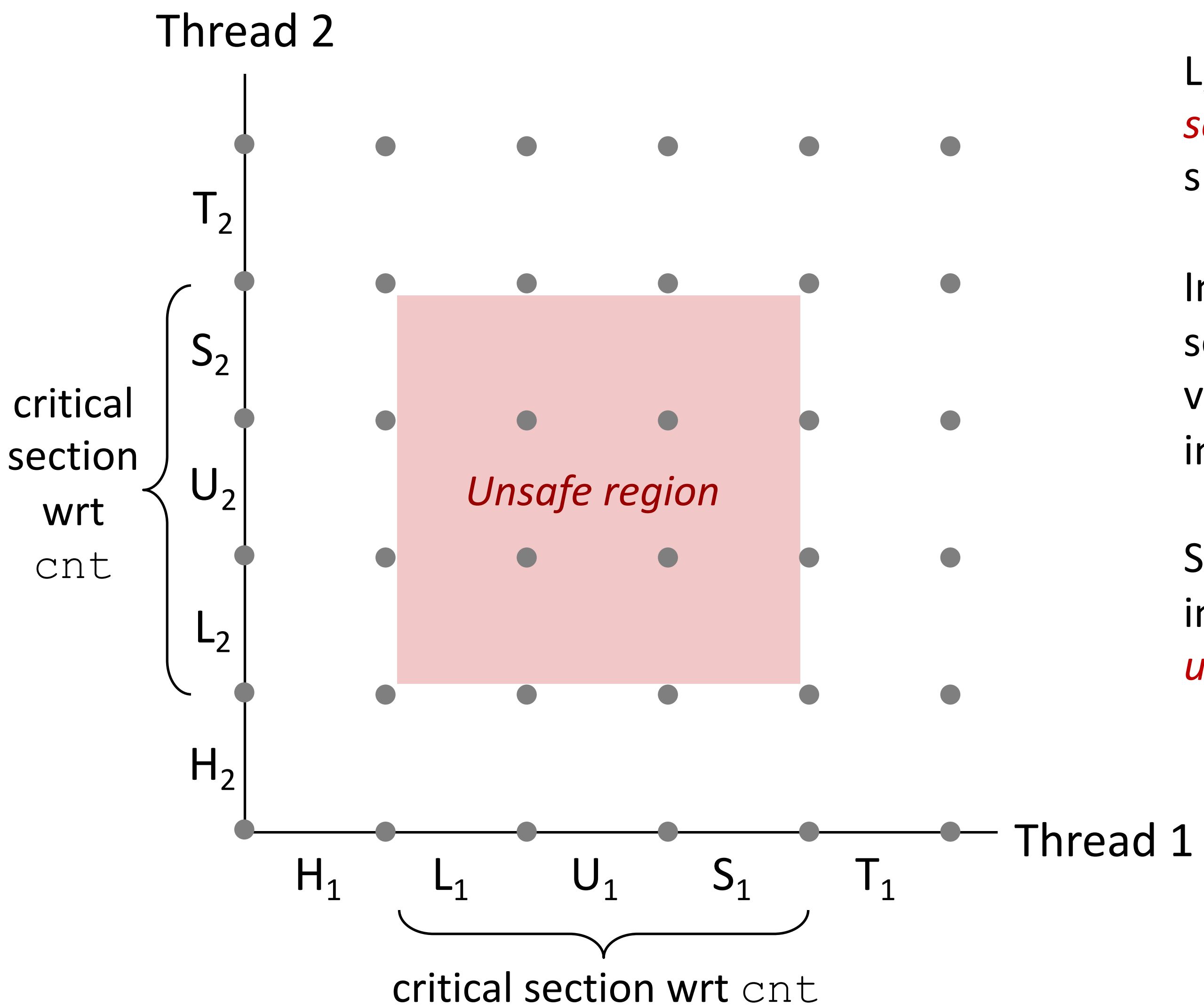


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Critical Sections and Unsafe Regions

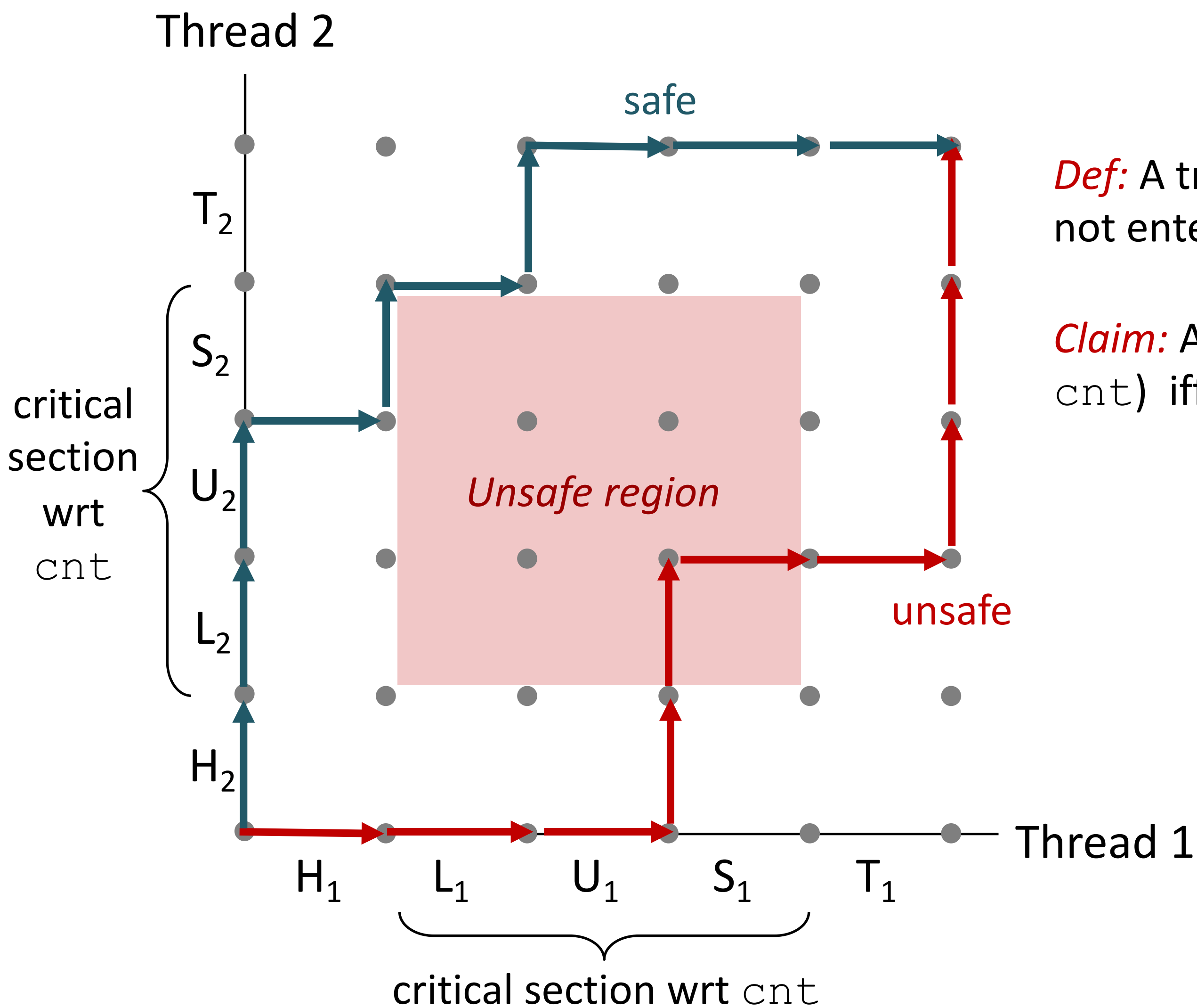


L, U, and S form a *critical section* with respect to the shared variable `cnt`

Instructions in critical sections (wrt to 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 never have an unsafe trajectory.
  - i.e., need to guarantee ***mutually exclusive access*** to critical regions
- Classic solution:
  - Semaphores (Edsger Dijkstra)
- Other approaches (out of our scope)
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Today

- Sharing
- Mutual exclusion
- Semaphores

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 *sem, 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

```
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
  int niters = atoi(argv[1]);
  pthread_t tid1, tid2;

  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=%d\n”, cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}
```

```
/* Thread routine */
void *thread(void *vargp)
{
  int i, niters = *((int *)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 int cnt = 0;       /* Counter */
sem_t mutex;                /* Semaphore that protects cnt */

sem_init(&mutex, 0, 1);     /* mutex = 1 */
```

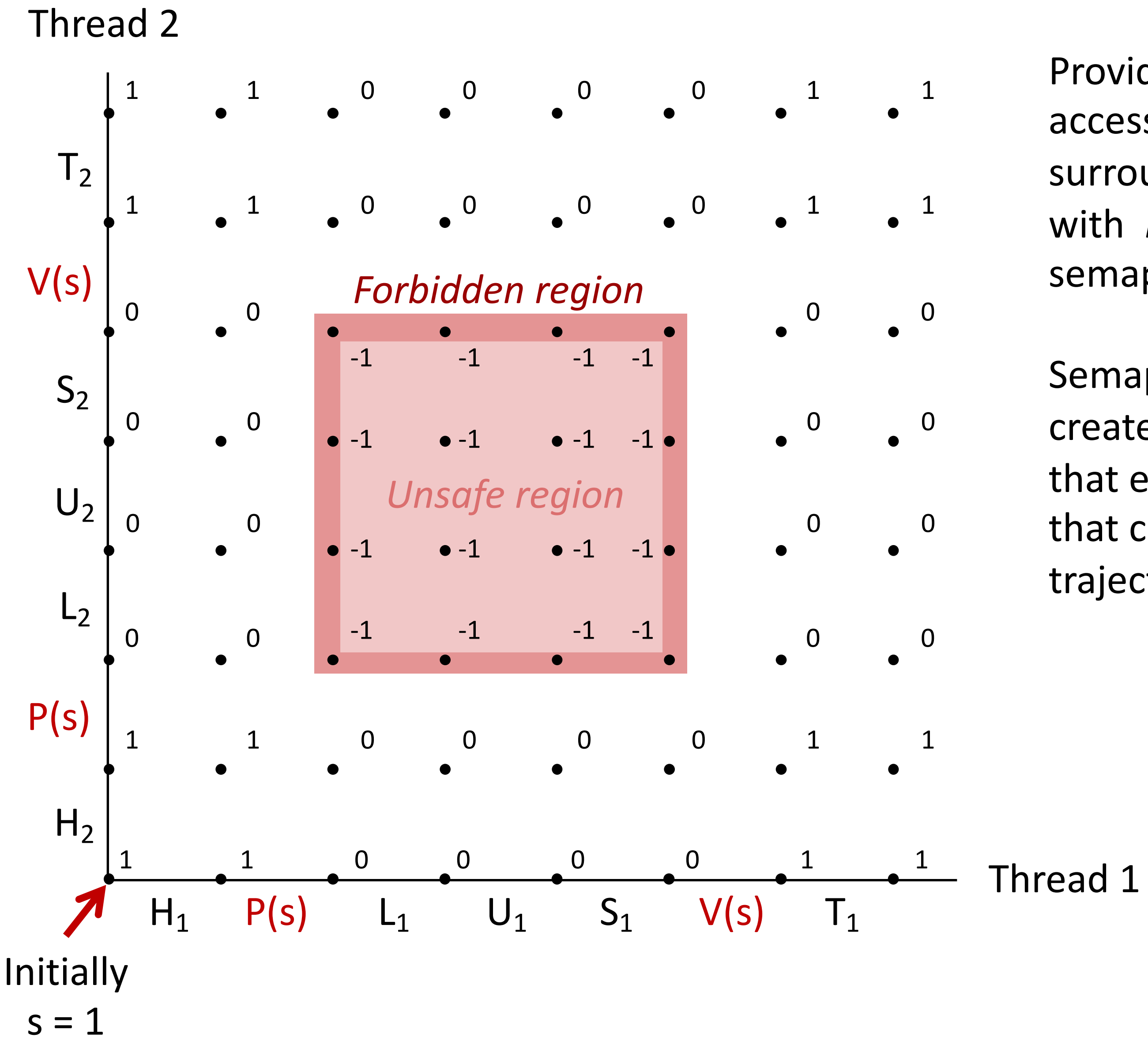
- **Surround** critical section with *P* and *V*:

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

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

Warning: It's much 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 that cannot be entered by any trajectory.

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.