Synchronization

CSE4100: Multicore Programming

Sungyong Park (PhD)
Data-Intensive Computing and Systems Laboratory (DISCOS)
https://discos.sogang.ac.kr
Office: R908A, E-mail: parksy@sogang.ac.kr

Shared Variables in Threaded C Programs

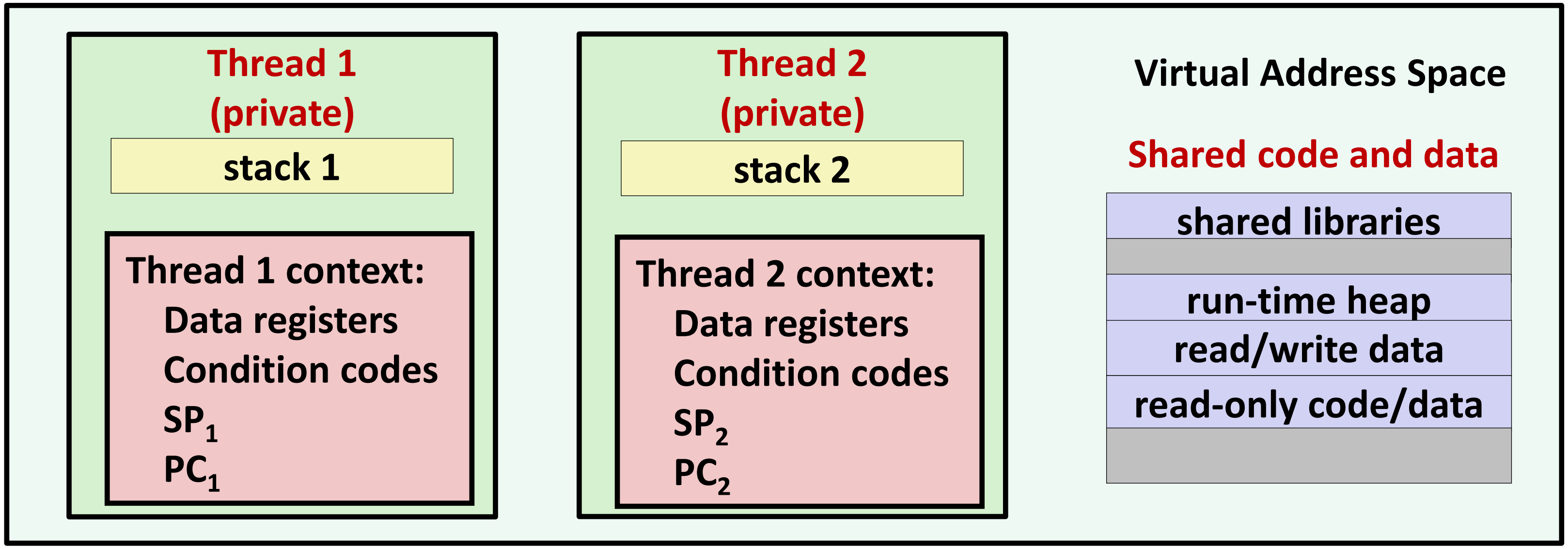
- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- **A variable `x` is *shared* if and only if (iff) 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?

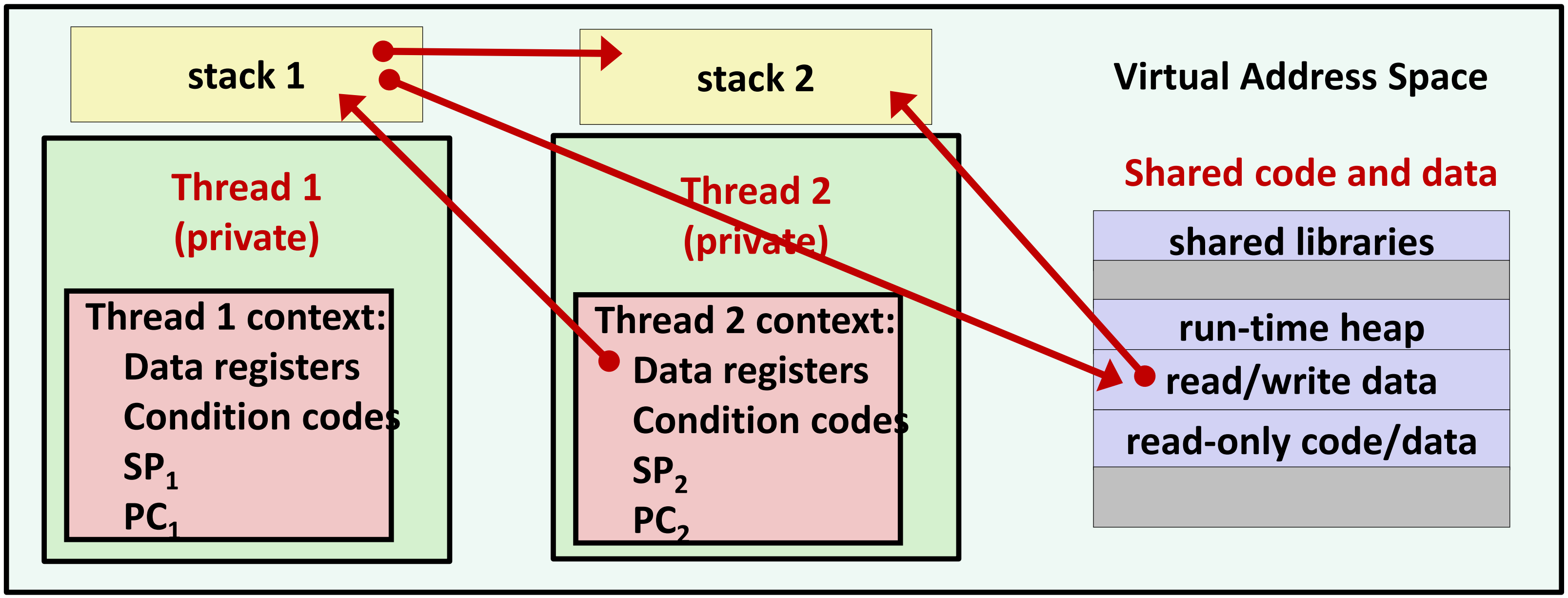
Threads Memory Model: Conceptual

- **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!

Three Ways to Pass Thread Argument

- **Malloc / Free**
  - Producer malloc's space, passes pointer to `pthread_create`
  - Consumer dereferences pointer, frees space
  - Always works; necessary for passing large amounts of data

- **Cast of int**
  - Producer casts an int/long to void*, passes to `pthread_create`
  - Consumer casts void* argument back to int/long
  - Works for small amounts of data (one number)

- **INCORRECT: Pointer to stack slot**
  - Producer passes address to producer's stack in `pthread_create`
  - Consumer dereferences pointer
  - Why is this unsafe?

Passing an Argument to a Thread (Case 1)

```
int hist[N] = {0};

int main(int argc, char *argv[])
{
   long i;
   pthread_t tids[N];

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

   for ( i = 0; i < N; i++ )
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   *(int *)vargp += 1;
   return NULL;
}
```

```
void check(void) {
   for ( int i = 0; i < N; i++ ) {
     if ( hist[i] != 1 ) {
       printf("Failed at %d\n", i);
       exit(-1);
     }
   }
   printf("OK\n");
}
```

- Each thread receives a *unique pointer*

Passing an Argument to a Thread (Case 2)

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for (i = 0; i < N; i++)
     Pthread_create(&tids[i],
                    NULL,
                    thread,
                    (void *)i);
   for (i = 0; i < N; i++)
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   hist[(long)vargp] += 1;
   return NULL;
}
```

- Each thread receives a *unique array index*
- Casting from long to void* and back is safe

Passing an Argument to a Thread (Case 3)

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

   for ( i = 0; i < N; i++ )
     long* p = Malloc(sizeof(long));
     *p = i;
     Pthread_create(&tids[i], NULL,
                    thread, p);

   for ( i = 0; i < N; i++ )
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   hist[*(long *)vargp] += 1;
   free(vargp);
   return NULL;
}
```

- Each thread receives a *unique array index*
- Malloc in parent, free in thread
- Necessary if passing structs

Passing an Argument to a Thread – WRONG!

```
int hist[N] = {0};

int main(int argc, char *argv[]) {
   long i;
   pthread_t tids[N];

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

   for ( i = 0; i < N; i++ )
     Pthread_join(tids[i], NULL);
   check();
}
```

```
void *thread(void *vargp)
{
   hist[*(long *)vargp] += 1;
   return NULL;
}
```

- **Each thread receives *the same pointer*, to *`i`* in main**
- **Data race: each thread *may or may not* read a unique array index from *`i`* in main**

Mapping Variable Instances to Memory

- **Global variables**
  - Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**
- **Local automatic variables**
  - Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**
- **Local static variables**
  - Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable**
- **`errno` is special**
  - Declared outside a function, but **each thread stack contains one instance**

Mapping Variable Instances to Memory (Cont)

Global var: 1 instance (`ptr` [data])

Local var: 1 instance (`i.m, msgs.m`)

```
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 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	→ *global*
`cnt`	no	yes	yes	→ *static*
`i.m`	yes	no	no	
`msgs.m`	yes	yes	yes	→ *global via `ptr`*
`myid.p0`	no	yes	no	
`myid.p1`	no	no	yes	

- **Answer: A variable `x` is shared if and only if one of its instances is referenced by more than one thread**
  - `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;
    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 be equal to 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++;
```

Assembly 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 (Case 1)

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - $\mathbf{I}_i$ denotes that thread ***i*** executes instruction **I**
  - **%rdx**$_i$ is the content of **%rdx** in thread ***i***'s context

Thread 1 critical section (green) | Thread 2 critical section (purple)

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 (Case 2)

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

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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 (Case 3)

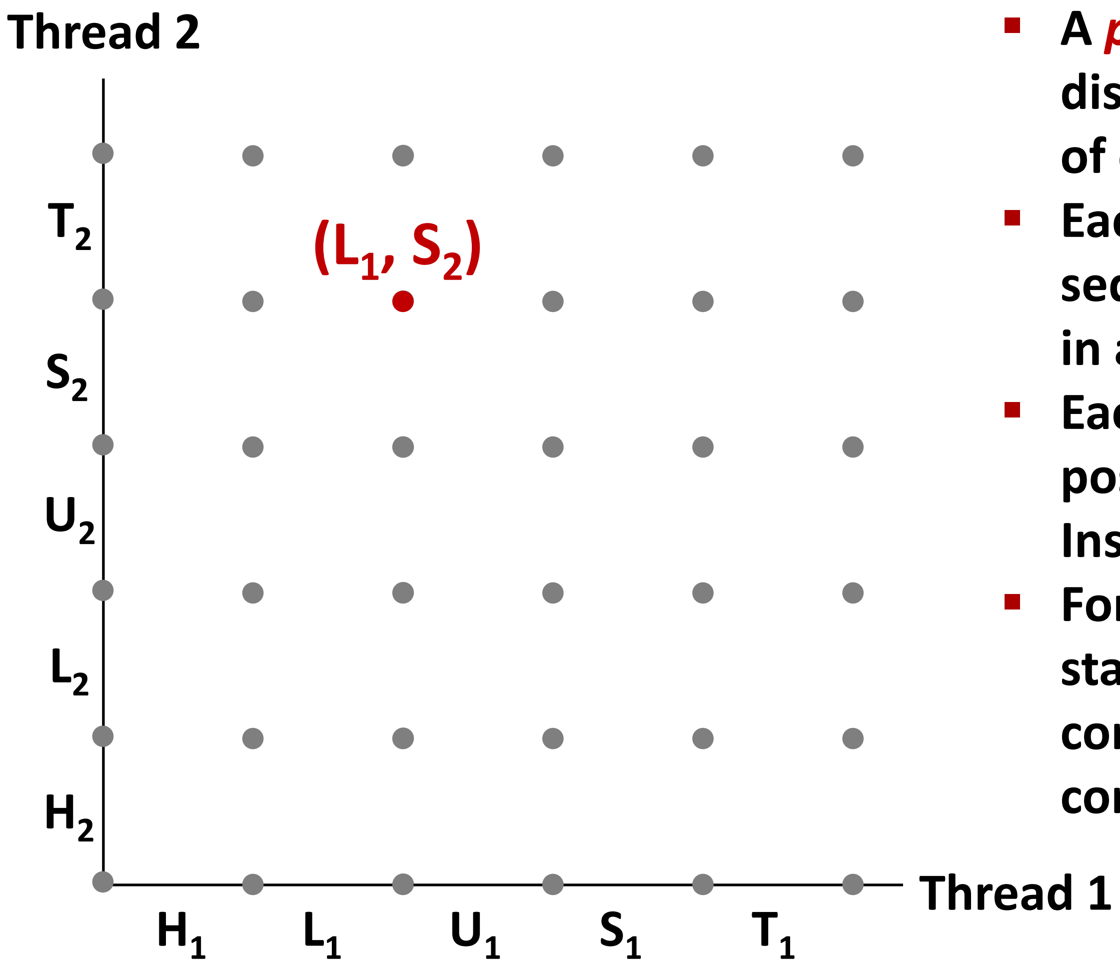
- How about this ordering?

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!

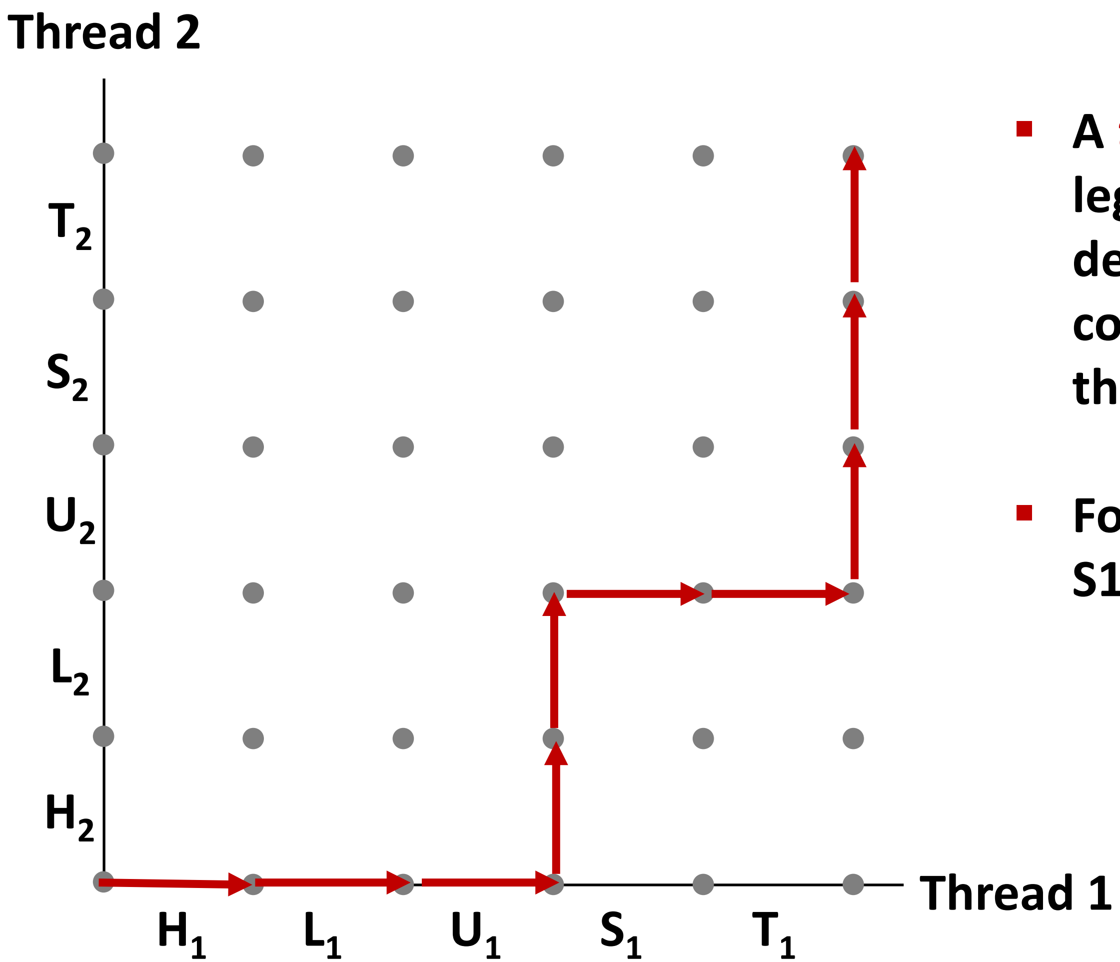
- We can analyze the behavior using a *progress graph*

Progress Graphs



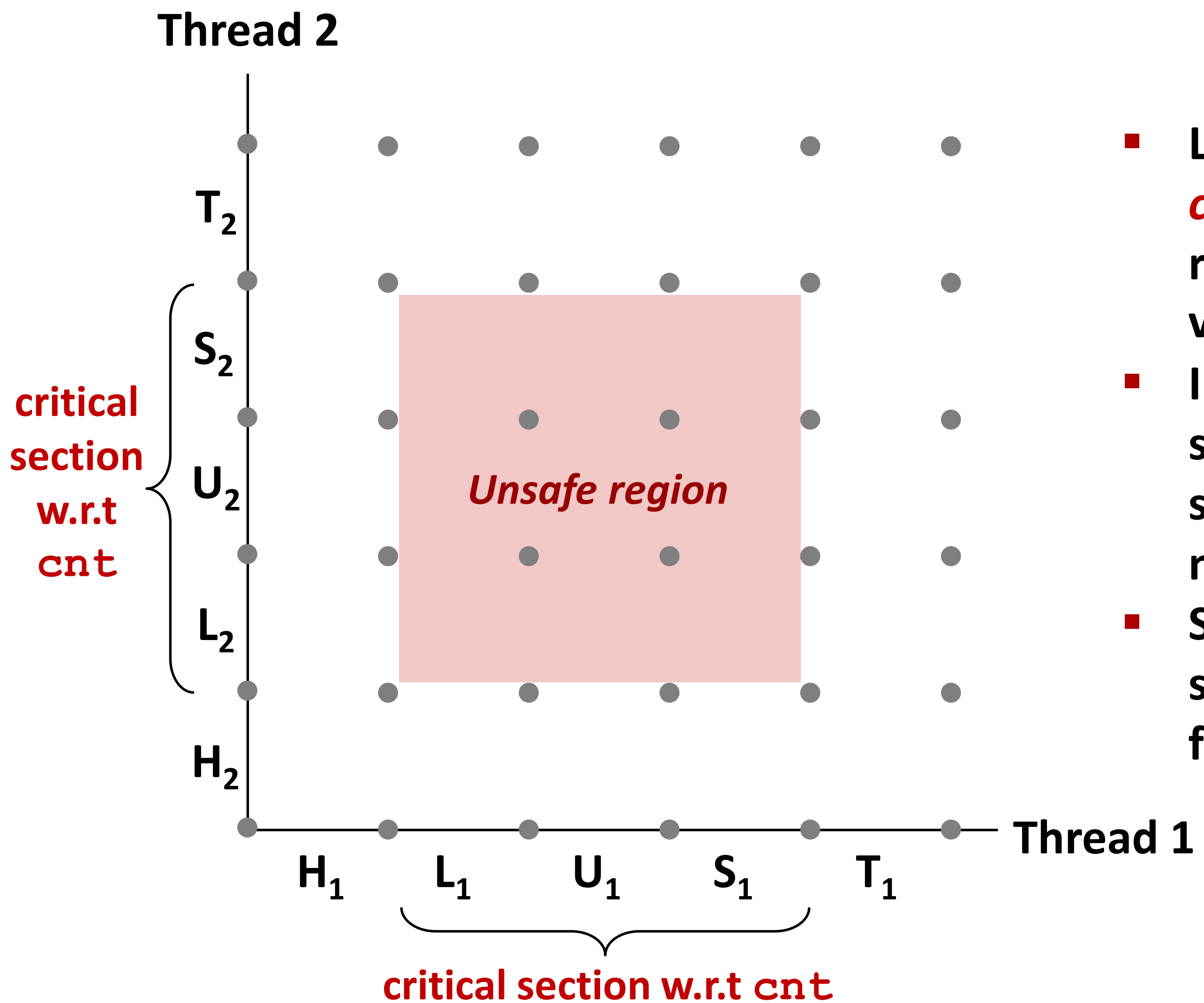
- A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads
- Each axis corresponds to the sequential order of instructions in a thread
- Each point corresponds to a possible ***execution state*** ($Inst_1$, $Inst_2$)
- For example, (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2

Trajectories in Progress Graphs



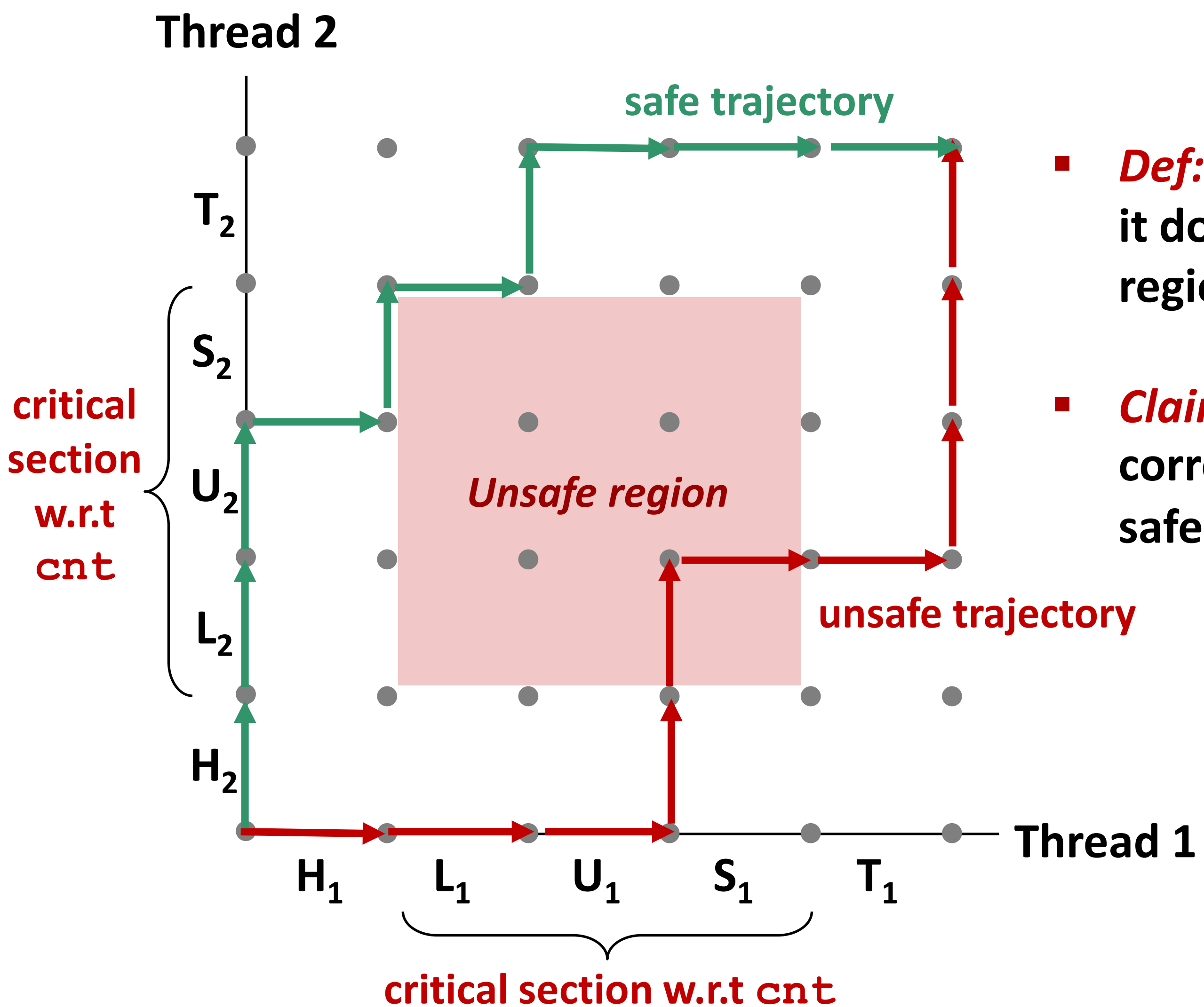
- A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads
- For example, H1, L1, U1, H2, L2, S1, T1, U2, S2, T2 is a trajectory

Critical Sections and Unsafe Regions



- L, U, and S form a *critical section* with respect to the shared variable `cnt`
- Instructions in critical sections (w.r.t some shared variable) should not be interleaved
- Sets of states where such interleaving occurs form *unsafe regions*

Critical Sections and Unsafe Regions (Cont)



- *Def:* A trajectory is *safe* iff it does not enter any unsafe region
- *Claim:* A trajectory is correct (w.r.t `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) : wait(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 the 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) : signal(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* (order or execution is implementation dependent but possibly the 1st thread on the queue), which then completes its ***P*** operation by decrementing ***s***
- **Semaphore invariant: *(s >= 0)***

Semaphore Operations in C

■ 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 */
```

Fix Improper Synchronization (badcnt.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

```
/* 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>
```

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 section (CS) with ***P(mutex)*** and ***V(mutex)*** operation

```
mutex = 1

P(mutex)
CS
V(mutex)
```

- **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*

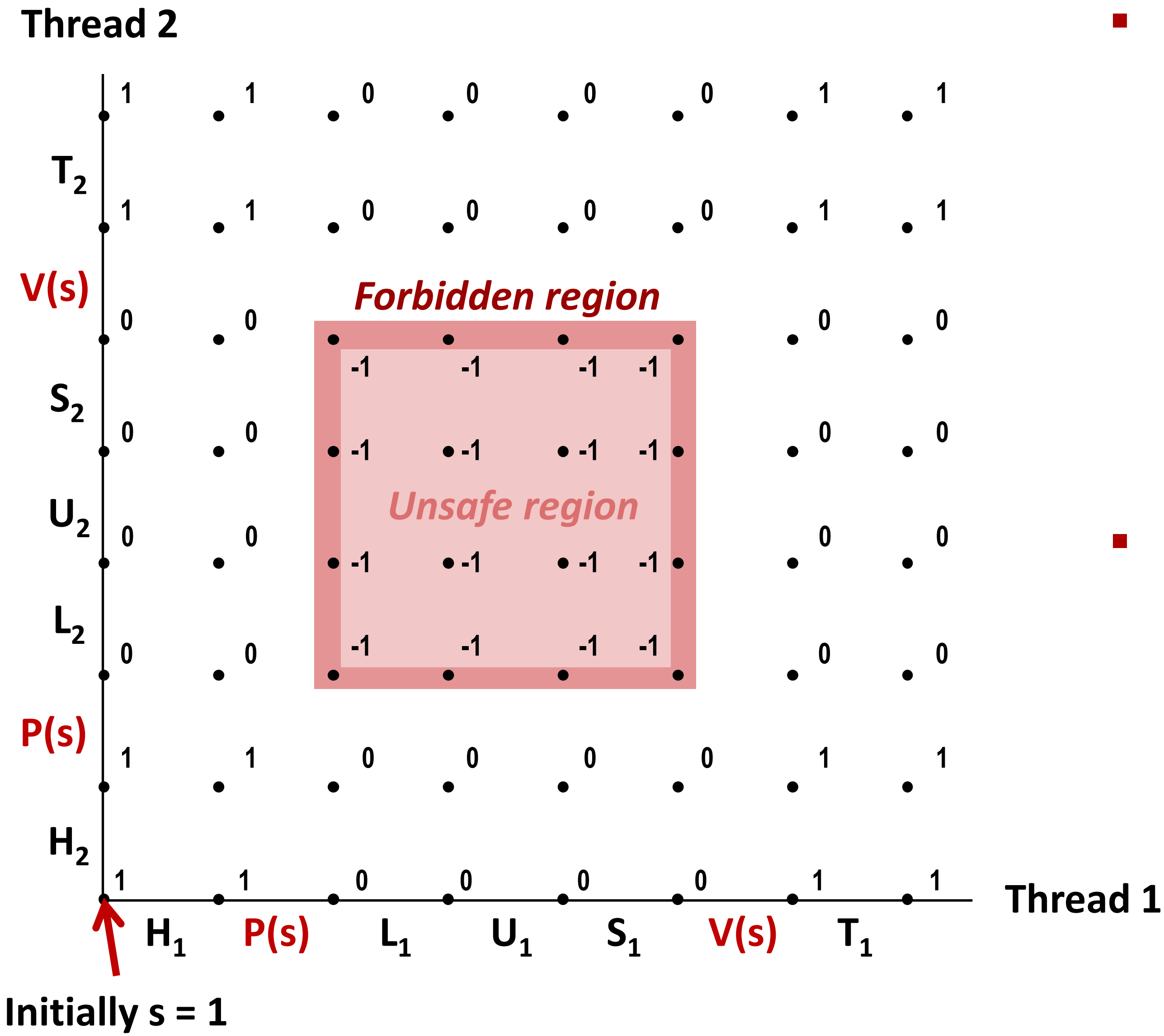
```
for (i = 0; i < niters; i++) {
     P(&mutex);
     cnt++;
     V(&mutex);
 }
```

goodcnt.c

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

Warning: It is 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

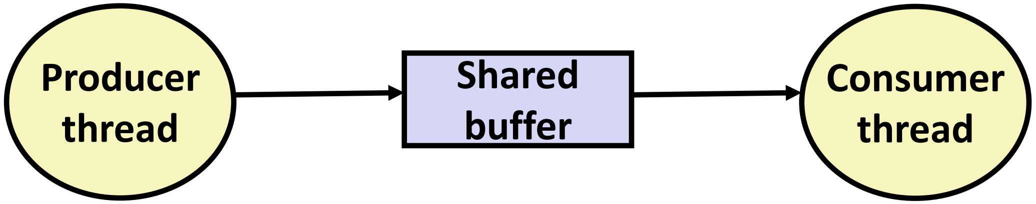
Using Semaphores to Coordinate Access to Shared Resources

- **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
  - Use counting semaphores to keep track of resource state and to notify other threads
  - Use *mutex* to protect access to resource

- **Two classic examples**
  - The ***Producer-Consumer*** Problem
  - The ***Readers-Writers*** Problem

Producer-Consumer Problem

- **Common synchronization pattern**
  - Producer waits for an empty ***slot***, inserts an item into the buffer, and notifies consumer
  - Consumer waits for an ***item***, removes it from the buffer, and notifies producer



- **Examples**
  - Multimedia processing:
    - Producer creates MPEG video frames, consumer renders them
  - Event-driven graphical user interfaces:
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - Consumer retrieves events from buffer and paints the display

Producer-Consumer on an *n*-Element Buffer

- **Requires a *mutex* and two counting semaphores**
  - **`mutex`**: enforces mutually exclusive access to the buffer
  - **`slots`**: counts the available slots in the buffer
  - **`items`**: counts the available items in the buffer
- **Implemented using a shared buffer package called `sbuf`**

```
#include "csapp.h"

typedef struct {
    int *buf;          /* Buffer array */
    int n;             /* Maximum number of slots */
    int front;         /* buf[(front+1)%n] is the first item */
    int rear;          /* buf[rear%n] is the last item */
    sem_t mutex;       /* Protects accesses to buf */
    sem_t slots;       /* Counts available slots */
    sem_t items;       /* Counts available items */
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int  sbuf_remove(sbuf_t *sp);
```

sbuf Package – Initialization

- Initializing and de-initializing a shared buffer

```
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
  sp->buf = Calloc(n, sizeof(int));
  sp->n = n;                       /* Buffer holds max of n items */
  sp->front = sp->rear = 0;        /* Empty buffer iff front == rear */
  Sem_init(&sp->mutex, 0, 1);  /* Binary semaphore for locking */
  Sem_init(&sp->slots, 0, n);  /* Initially, buf has n empty slots */
  Sem_init(&sp->items, 0, 0);  /* Initially, buf has 0 items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```
sbuf.c

sbuf Package – Insert / Remove

- Inserting / removing an item into / from a shared buffer

```
/* Insert an item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)]
       = item;                              /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}
```

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item =
      sp->buf[(++sp->front)%(sp->n)]; /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```

Readers-Writers Problem

- **Generalization of the mutual exclusion problem**
- **Problem statement**
  - *Reader* threads only read the object
  - *Writer* threads modify the object
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object

Can new reader/writer run?

Currently running

	Reader	***Writer***
Reader	**O**	**X**
Writer	**X**	**X**

- **Occurs frequently in real systems, e.g.,**
  - Online airline reservation system
  - Multithreaded caching Web proxy

Variants of Readers-Writers

- ***First readers-writers problem*** **(favors readers)**
  - No reader should be kept waiting unless a writer has already been granted permission to use the object
  - A reader that arrives after a waiting writer gets priority over the writer
- ***Second readers-writers problem*** **(favors writers)**
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting
- ***Starvation*** **(where a thread waits indefinitely) is possible in both cases**

Solution to First Readers-Writers Problem

Readers

```
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Initially = 1 */

void reader(void)
{
    while ( 1 ) {
      P(&mutex);
      readcnt++;
      if ( readcnt == 1 ) /* First in */
           P(&w);
      V(&mutex);

      /* CS: Reading happens */

      P(&mutex);
      readcnt--;
      if ( readcnt == 0 ) /* Last out */
           V(&w);
      V(&mutex);
    }
}
```

Writers

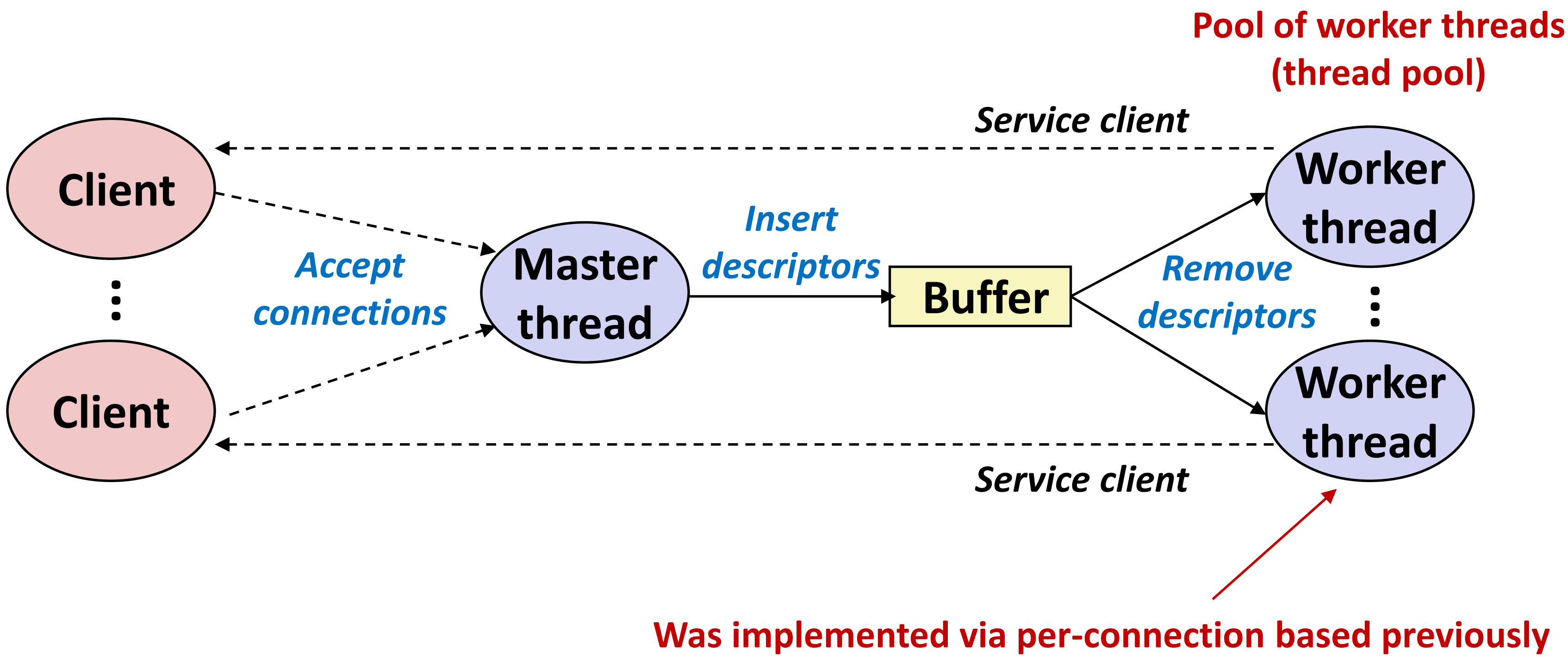
```
void writer(void)
{
    while ( 1 ) {
      P(&w);

      /* CS: Writing happens */

      V(&w);
    }
}
```

rw1.c

What is the order of execution for the following readers/writers?

(1) R_1->W_1->R_2->W_2
(2) W_1->R_1->R_2->W_2

Overview of Pre-threaded Concurrent Server



Pre-threaded Concurrent Server

```
sbuf_t sbuf; /* Shared buffer of connected descriptors */

int main(int argc, char **argv) {
    int i, listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    for ( i = 0; i < NTHREADS; i++ ) /* Create a pool of worker threads */
        Pthread_create(&tid, NULL, thread, NULL);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
}
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    while ( 1 ) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buf */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}
```

echoservert_pre.c

Pre-threaded Concurrent Server (Cont)

```
static int byte_cnt;  /* Byte counter */
static sem_t mutex;   /* and the mutex that protects it */
static void init_echo_cnt(void)
{
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
void echo_cnt(int connfd)
{
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;

    Pthread_once(&once, init_echo_cnt);
    Rio_readinitb(&rio, connfd);
    while ( ( n = Rio_readlineb(&rio, buf, MAXLINE) ) != 0 ) {
        P(&mutex);
        byte_cnt += n;
        printf("thread %d received %d (%d total) bytes on fd %d\n",
               (int) pthread_self(), n, byte_cnt, connfd);
        V(&mutex);
        Rio_writen(connfd, buf, n);
    }
}
```

echo_cnt.c

Crucial Concept: Thread Safety

- **Functions called from a thread must be *thread-safe***

- **A function is *thread-safe* iff it will always produce correct results when called repeatedly from multiple concurrent threads**

- **Classes of thread-unsafe functions:**
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions ☺

Thread-Unsafe Functions (Class 1)

- **Failing to protect shared variables**
  - Fix: Use *P* and *V* semaphore operations
  - Example: `goodcnt.c`
  - Issue: Synchronization operations will slow down code

Thread-Unsafe Functions (Class 2)

- **Relying on persistent state across multiple function invocations**
  - Example: Random number generator that relies on static state

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Thread-Safe Random Number Generator

- **Pass state as part of argument**
  - and, thereby, eliminate global state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- **Consequence: programmer using `rand_r` must maintain seed**

Thread-Unsafe Functions (Class 3)

- **Returning a pointer to a static variable**
- **Fix 1. Rewrite function so caller passes address of variable to store result**
  - Requires changes in caller and callee
- **Fix 2. Lock-and-copy**
  - Requires simple changes in caller (and none in callee)
  - However, caller must free memory

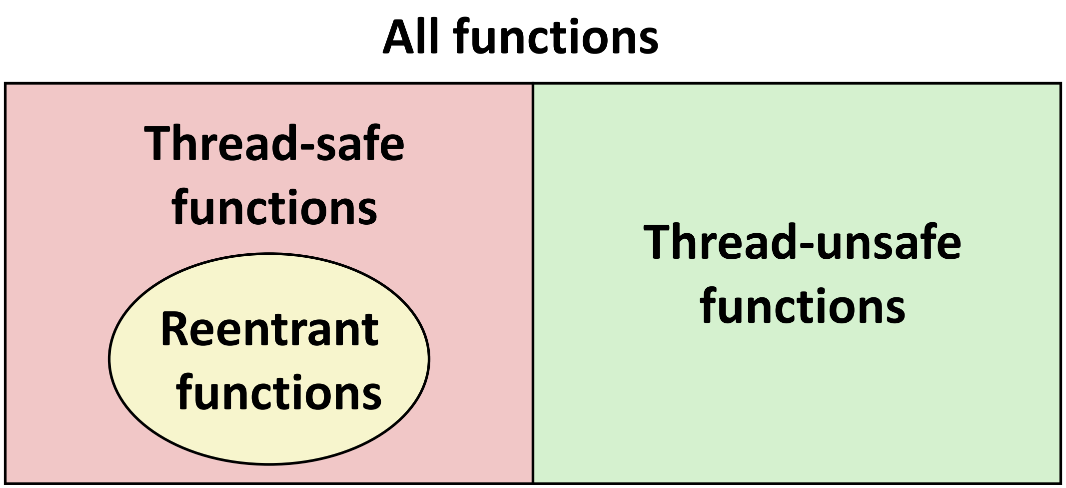
```
/* lock-and-copy version */

char *ctime_ts(const time_t *timep,
               char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);

    return privatep;
}
```

Thread-Unsafe Functions (Class 4)

- **Calling thread-unsafe functions**
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions ☺

Reentrant Functions

- **A function is *reentrant* iff it accesses no shared variables when called by multiple threads**
  - Important subset of thread-safe functions
    - Require no synchronization operations
    - Only way to make a Class 2 function thread-safe is to make it *reentrant* (e.g., `rand_r`)



Examples

- Not thread-safe, not re-entrant

```
int tmp;
int add(int a) {
     tmp = a;
     return tmp + 10;
}
```

- Not thread-safe, re-entrant

```
int tmp;
int add(int a) {
     tmp = a;
     return a + 10;
}
```

- Thread-safe, not re-entrant

```
thread_local int tmp;
int add(int a) {
     tmp = a;
     return tmp + 10;
}
```

- Thread-safe, re-entrant

```
int add(int a) {
    return a + 10;
}
```

Thread-Safe Library Functions

- **All functions in the Standard C Library are thread-safe**
  - Examples: `malloc`, `free`, `printf`, `scanf`
- **Most Unix system calls are thread-safe, with a few exceptions:**

Thread-unsafe function	Class	Reentrant version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	`(none)`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

One Worry: Races

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

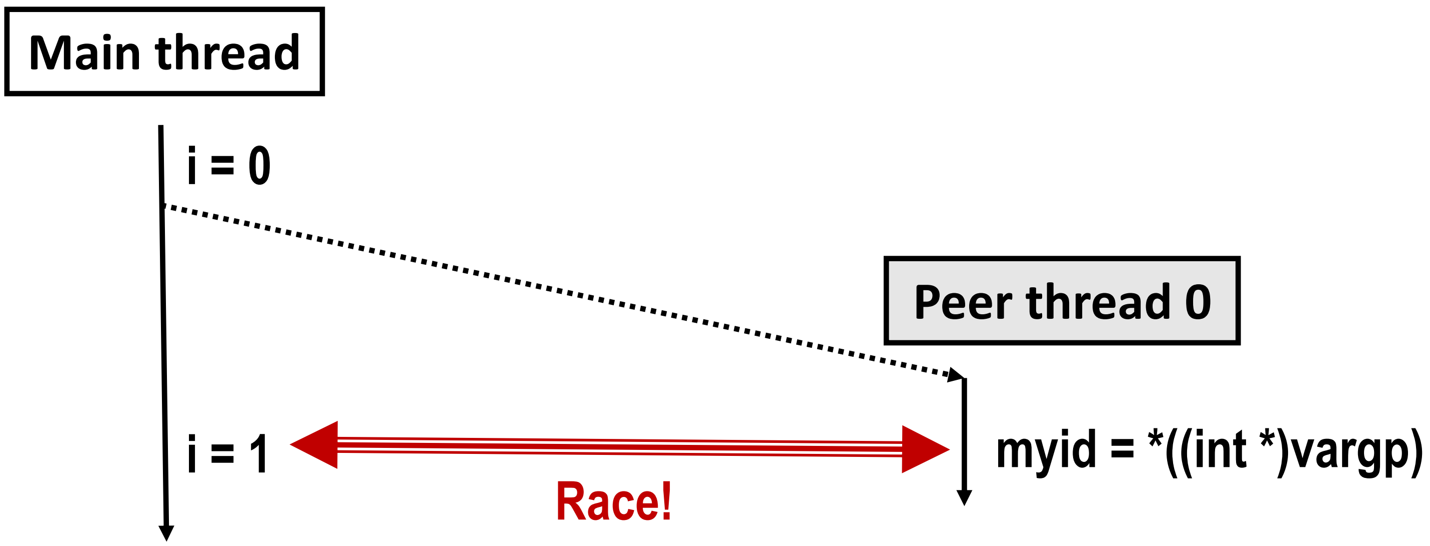
```
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;

    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);
}

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

N threads are sharing *i*

race.c

Why Race?

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



- **Race between increment of *i* in main thread and dereference of *vargp* in peer thread**
  - If dereference 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);
}
```

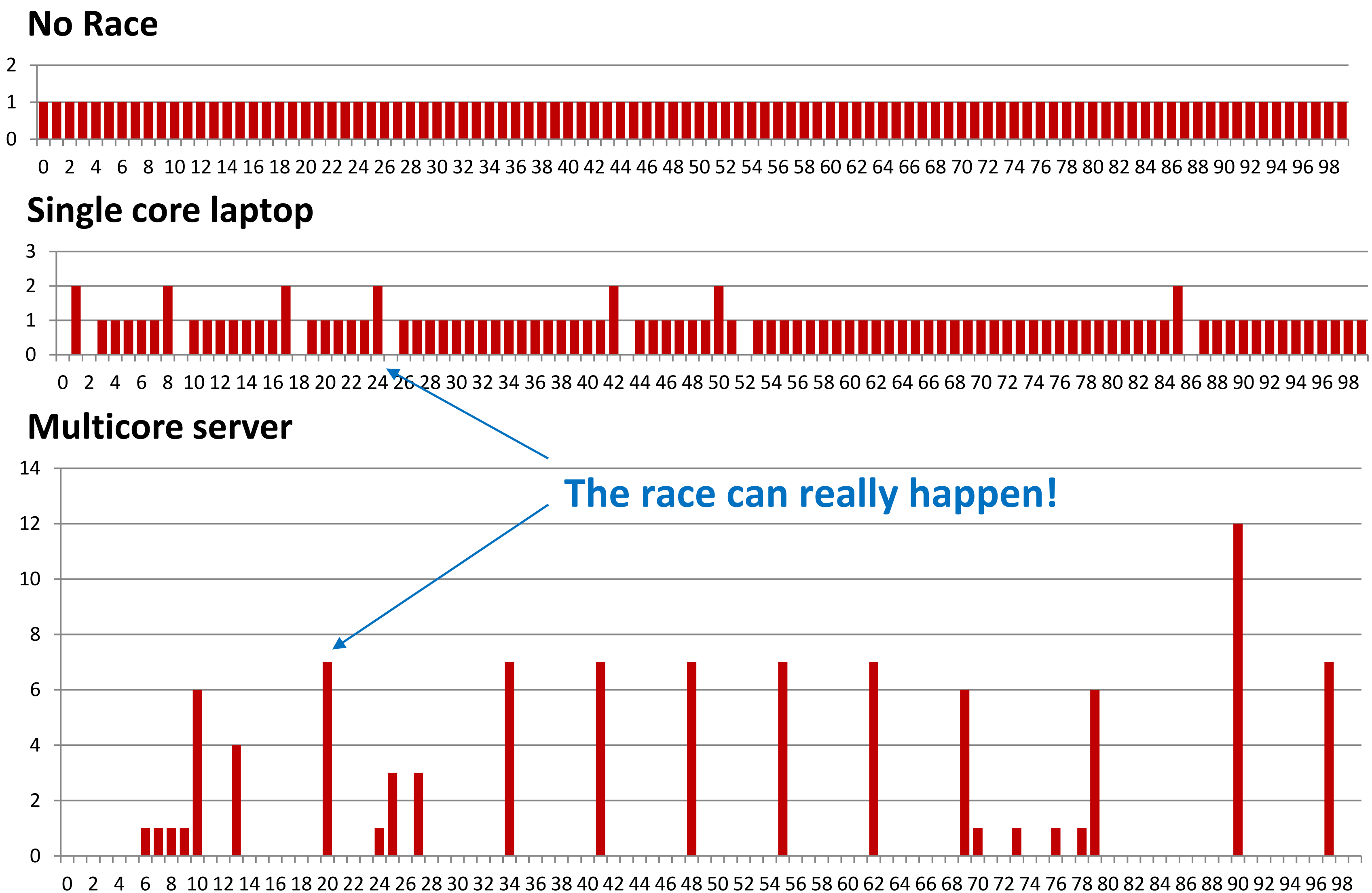
Peer thread

```
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



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);
}

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

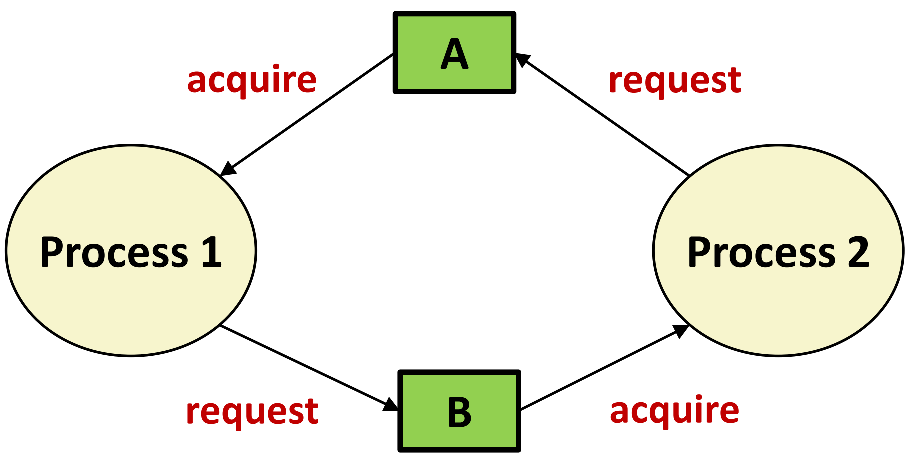
Avoid unintended sharing of state

norace.c

Another Worry: Deadlock

- 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!



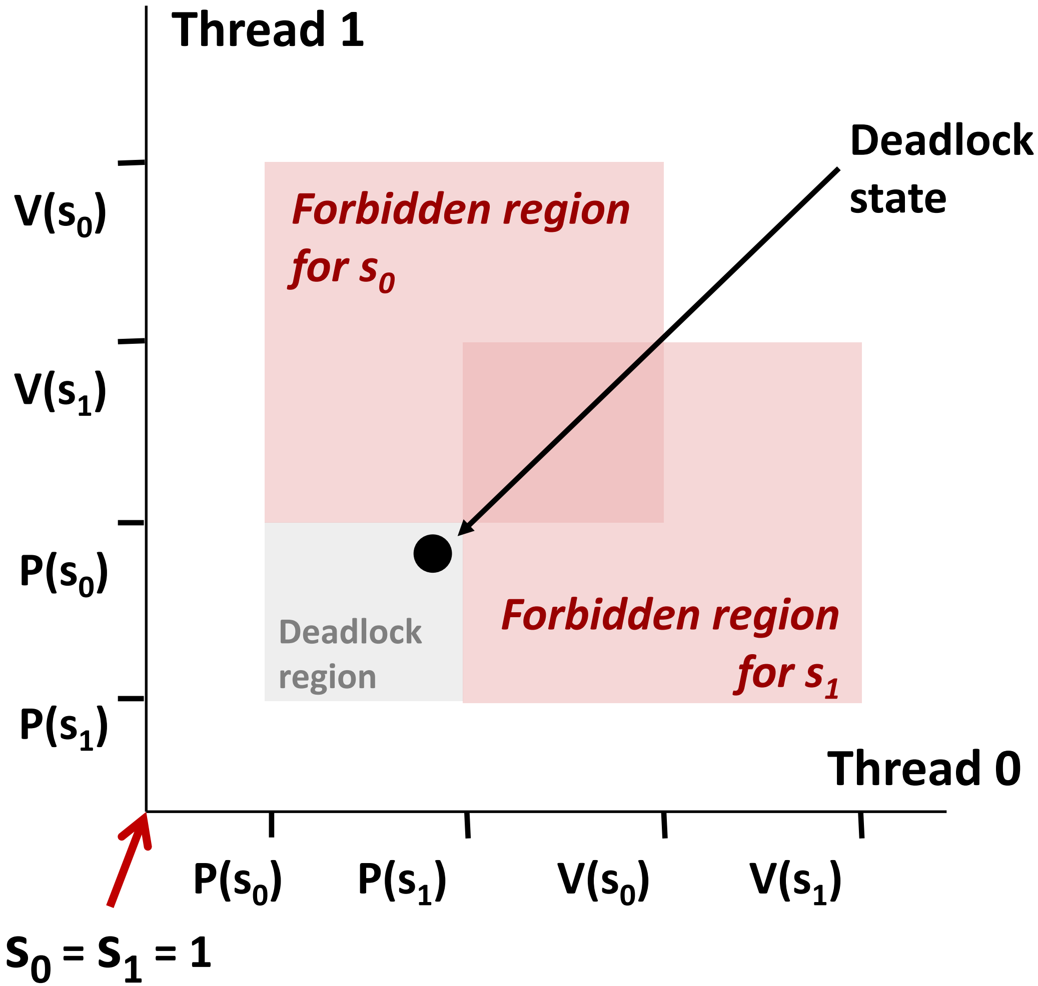
Deadlock with Semaphores

```
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

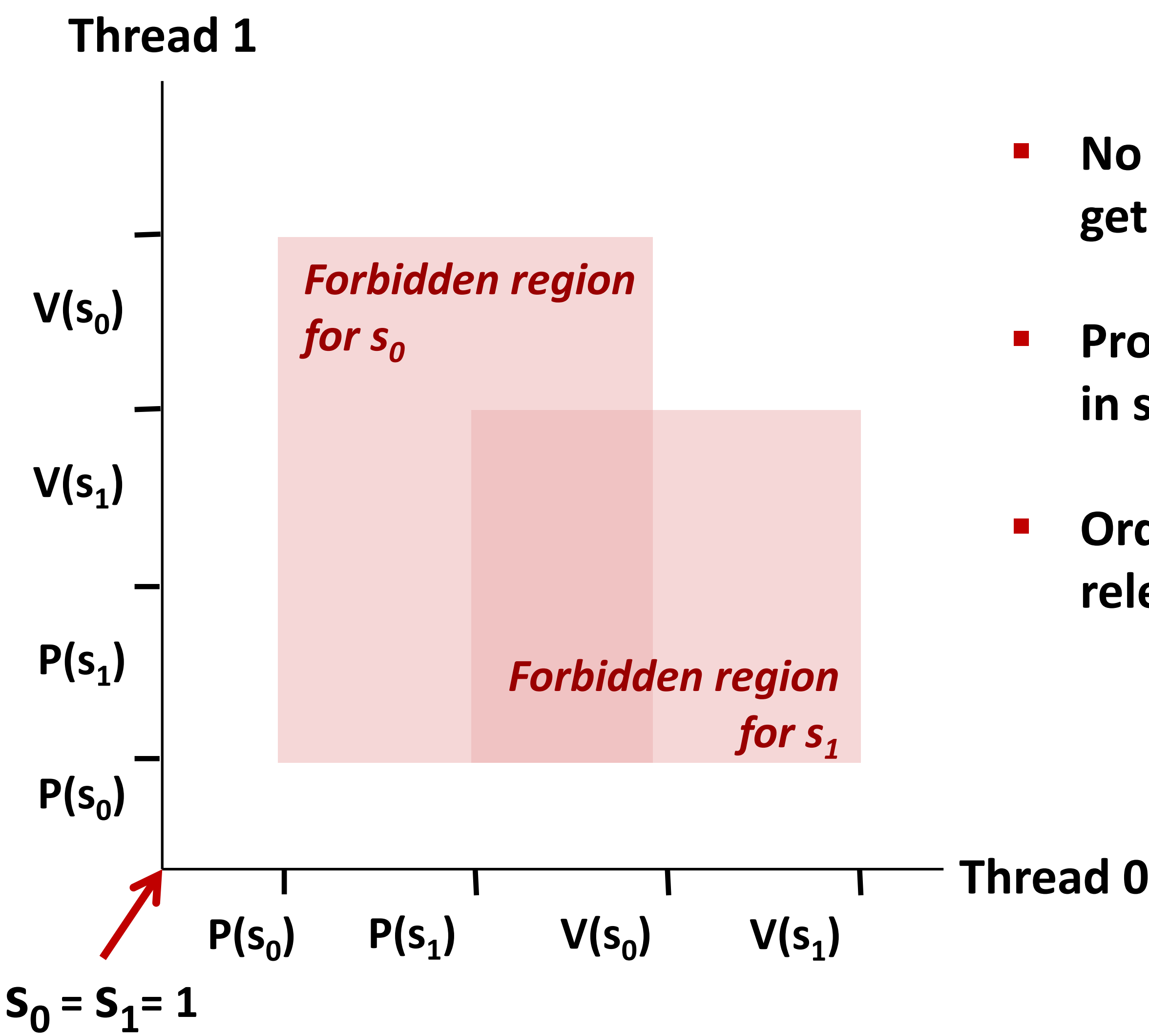
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

Acquire shared resources in same order

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