

Introduction to Computer Systems

15-213/18-243, fall 2009

22nd Lecture, Nov. 17

Instructors:

Roger B. Dannenberg and Greg Ganger

Today

- **Threads: review basics**
- **Synchronization**
- **Races, deadlocks, thread safety**

Process: Traditional View

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

Process context

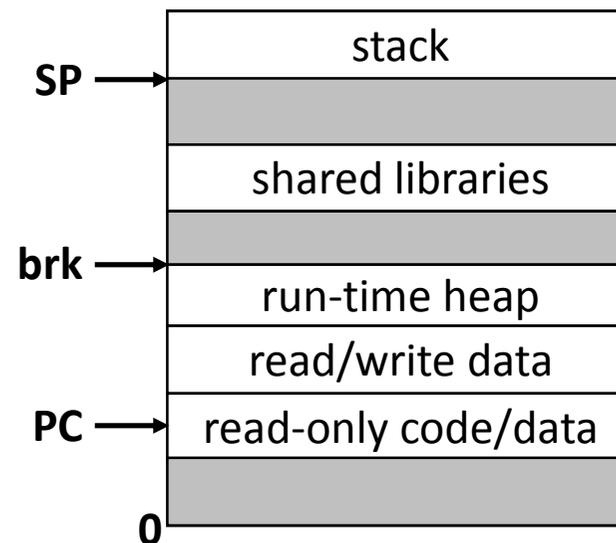
Program context:

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

Kernel context:

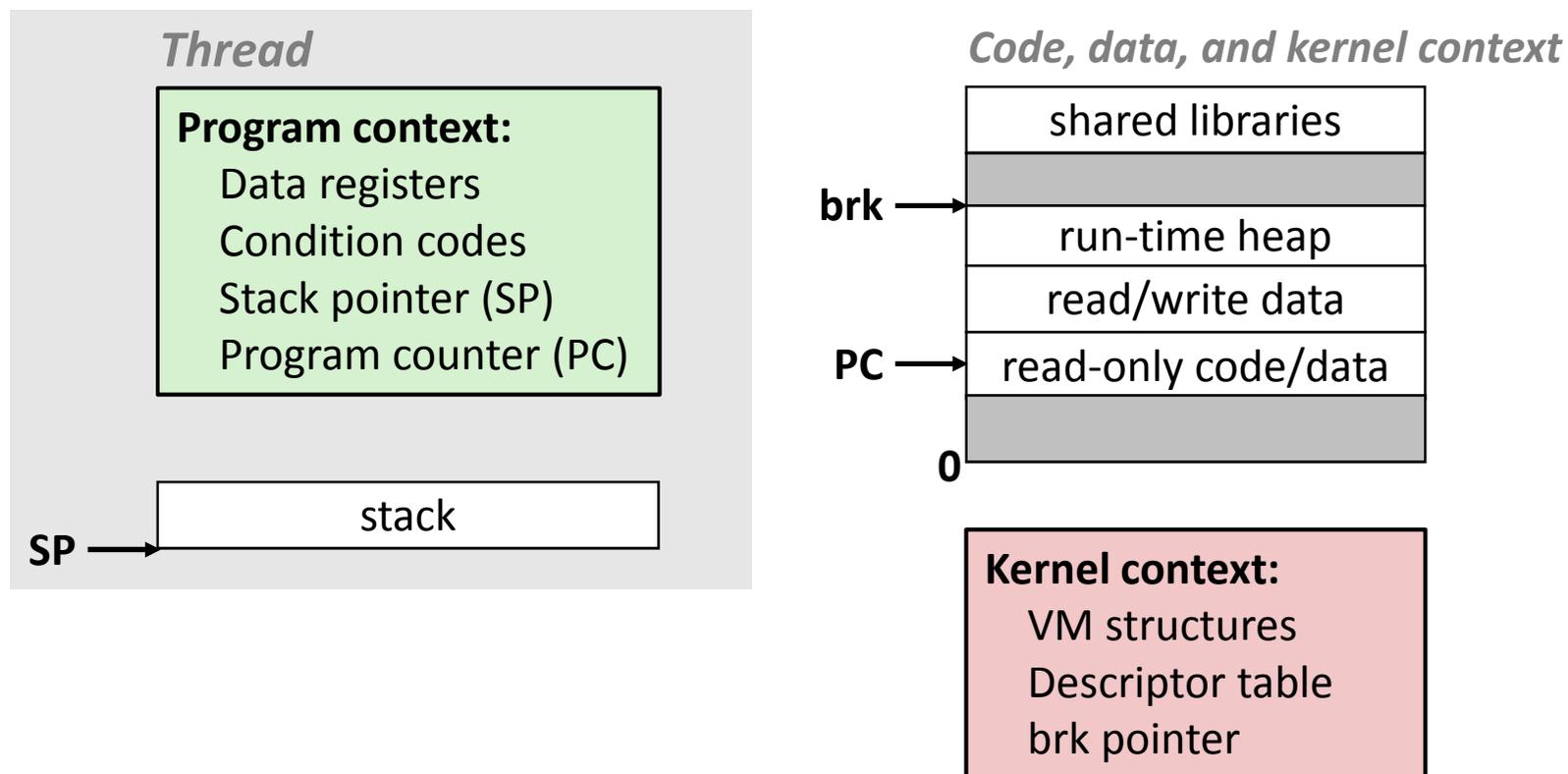
VM structures
Descriptor table
brk pointer

Code, data, and stack

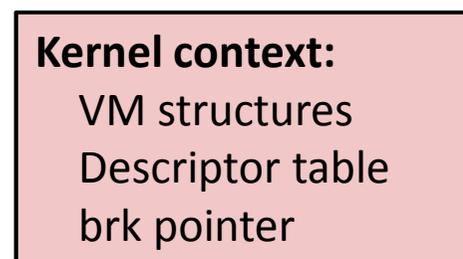
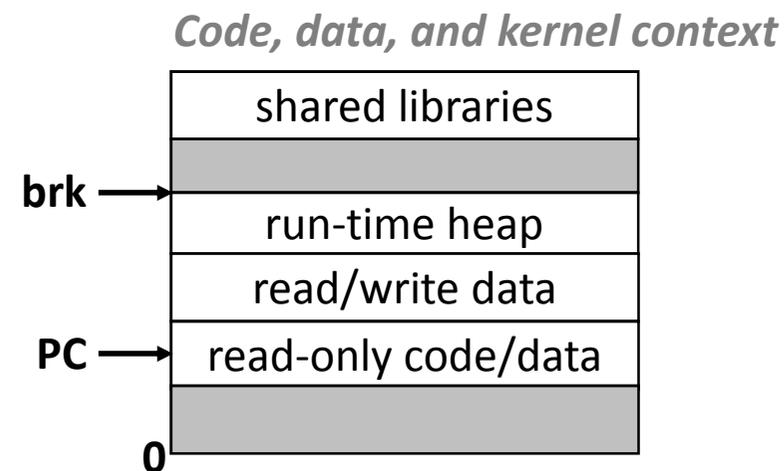
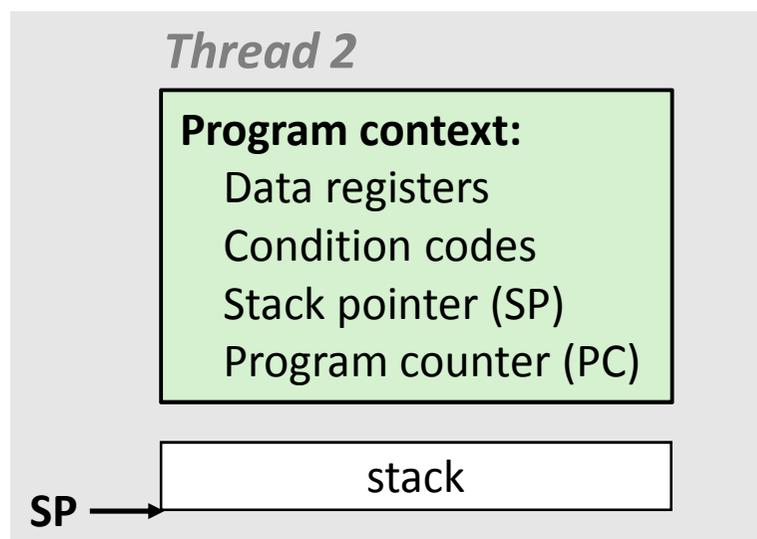
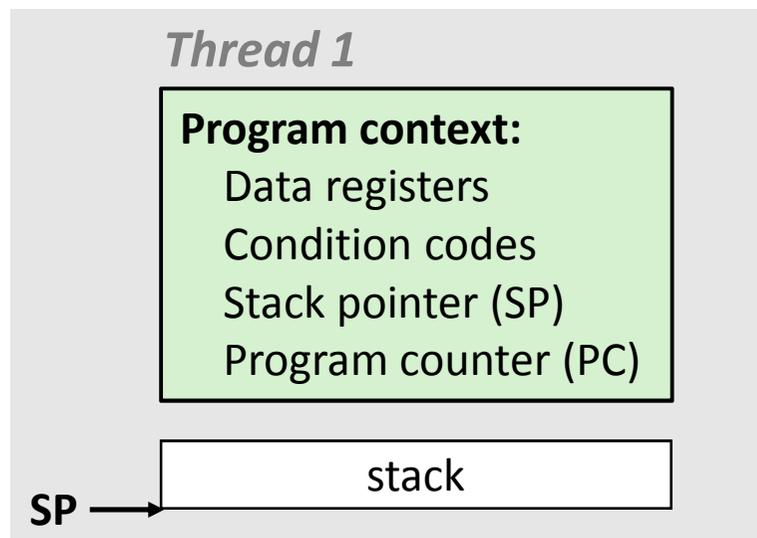


Process: Alternative View

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



Process with Two Threads



Threads vs. Processes

■ Threads and processes: similarities

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

■ Threads and processes: differences

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

Detaching Threads

■ Thread-based servers:

Use “detached” threads to avoid memory leaks

- At any point in time, a thread is either *joinable* or *detached*
- *Joinable* thread can be reaped and killed by other threads
 - must be reaped (with `pthread_join`) to free memory resources
- *Detached* thread cannot be reaped or killed by other threads
 - resources are automatically reaped on termination
- Default state is joinable
 - use `pthread_detach(pthread_self())` to make detached

■ Must be careful to avoid unintended sharing

- For example, what happens if we pass the address of `connfd` to the thread routine?
 - `pthread_create(&tid, NULL, thread, (void *)&connfd);`

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!**
 - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads

Today

- Threads: basics
- **Synchronization**
- Races, deadlocks, thread safety

Shared Variables in Threaded C Programs

- **Question: Which variables in a threaded C program are shared variables?**
 - 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 variables mapped to each memory instance?
 - 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, program counter, condition codes, and general purpose 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

■ *Mismatch between the conceptual and operation model is a source of confusion and errors*

Thread Accessing Another Thread'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 svar = 0;

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

Peer threads access main thread's stack indirectly through global ptr variable

Mapping Variables to Memory Instances

Global var: 1 instance (ptr [data])

Local vars: 1 instance (i_m , msgs.m)

```
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);
}
```

Local var: 2 instances (
 myid_{p0} [peer thread 0's stack],
 myid_{p1} [peer thread 1's stack]
)

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

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

Local static var: 1 instance (svar [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>svar</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`, `svar`, and `msgs` are shared
- `i` and `myid` are *not* shared

badcnt.c: Improper Synchronization

```

/* shared */
volatile unsigned int cnt = 0;
#define NITERS 100000000

int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                  count, NULL);
    Pthread_create(&tid2, NULL,
                  count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n",
              cnt);
    else
        printf("OK cnt=%d\n",
              cnt);
}

```

```

/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}

```

```

linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672

```

cnt should be
equal to 200,000,000.
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

Head (H_i)	{	.L9:	movl -4(%ebp),%eax	
			cmpl \$99999999,%eax	
			jle .L12	
			jmp .L10	
Load cnt (L_i)		.L12:	movl cnt,%eax	# Load
Update cnt (U_i)			leal 1(%eax),%edx	# Update
Store cnt (S_i)			movl %edx,cnt	# Store
Tail (T_i)	{	.L11:	movl -4(%ebp),%eax	
			leal 1(%eax),%edx	
			movl %edx,-4(%ebp)	
			jmp .L9	
		.L10:		

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

Key:
L oad
U pdate
S tore

OK

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

Key:
 L oad
 U pdate
 S tore

Oops!

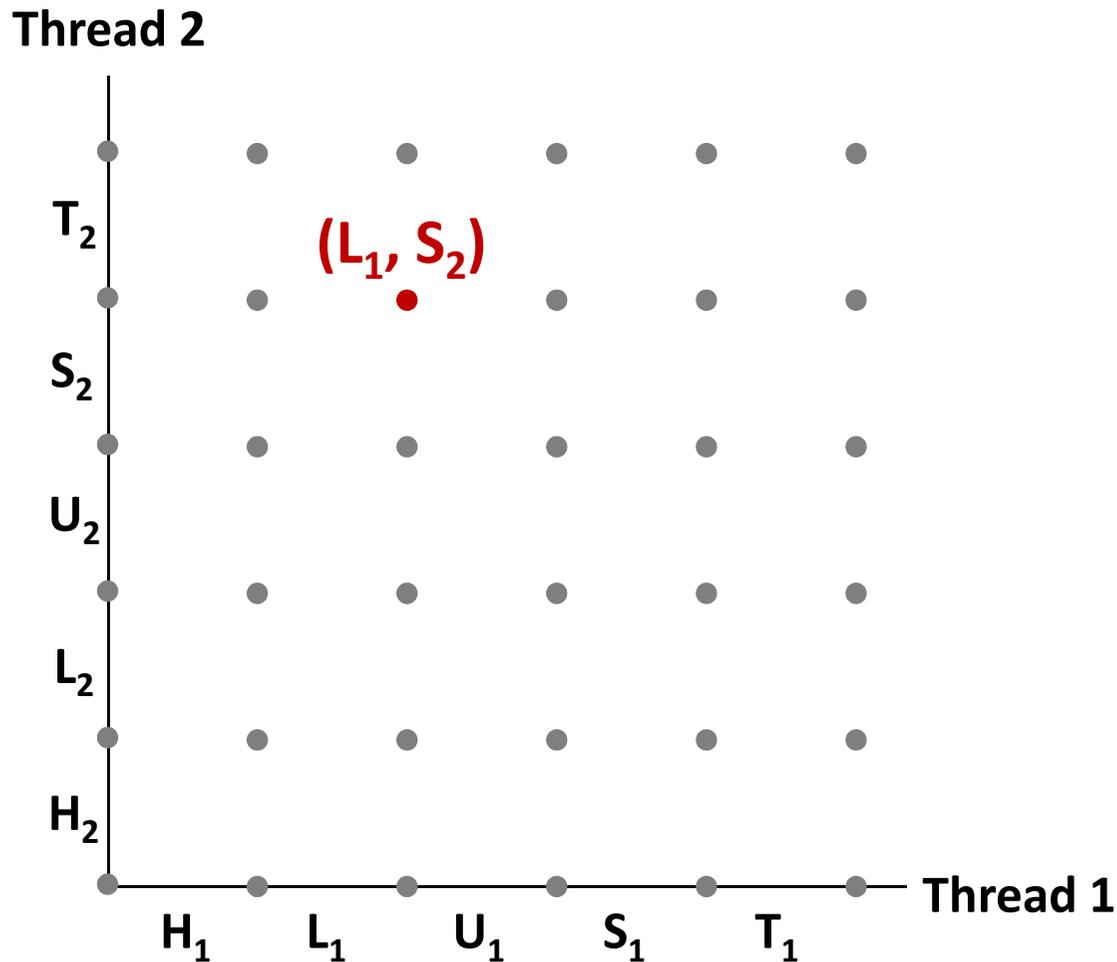
Concurrent Execution (cont)

- How about this ordering?

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

- We can analyze the behaviour using a *process graph*

Progress Graphs



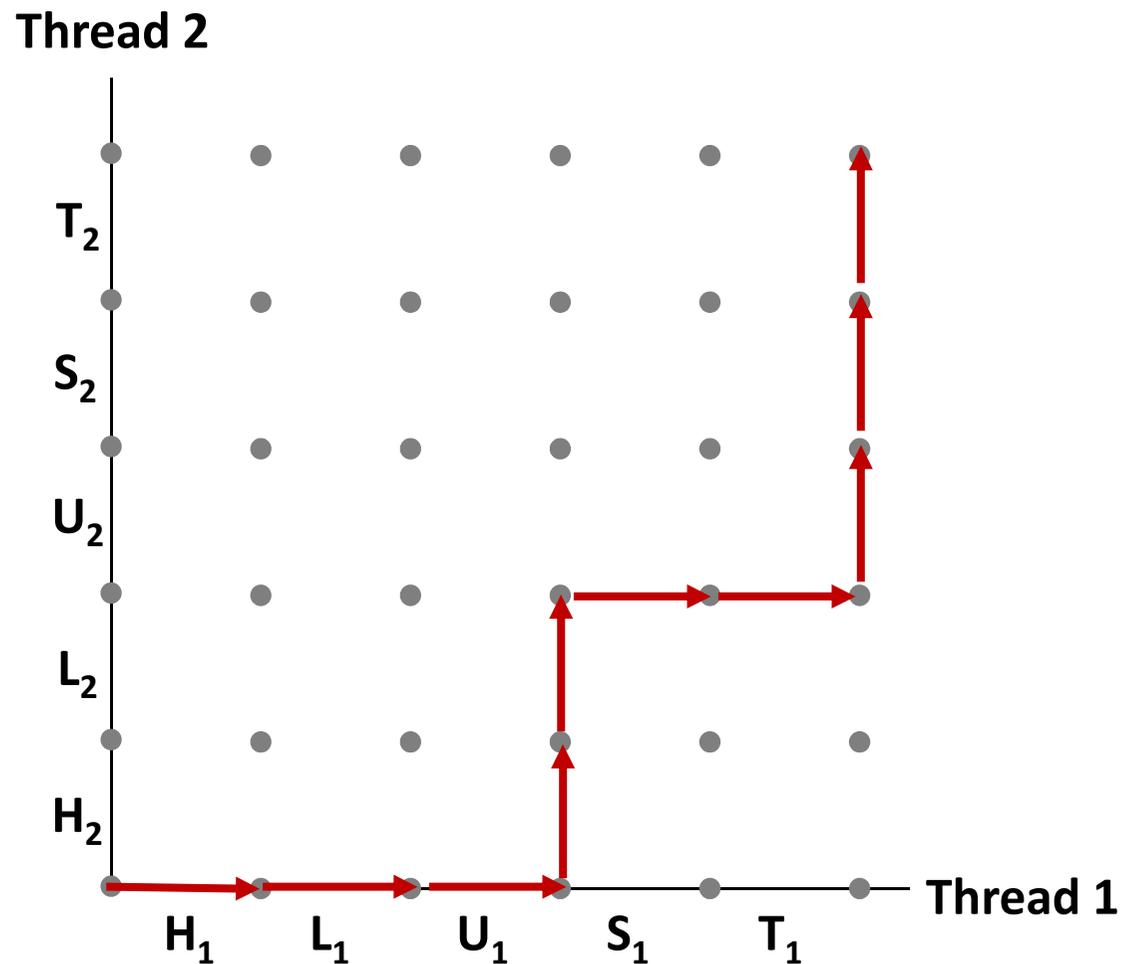
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

Each point corresponds to a possible *execution state* (Inst₁, Inst₂).

E.g., (L₁, S₂) denotes state where thread 1 has completed L₁ and thread 2 has completed S₂.

Trajectories in Progress Graphs

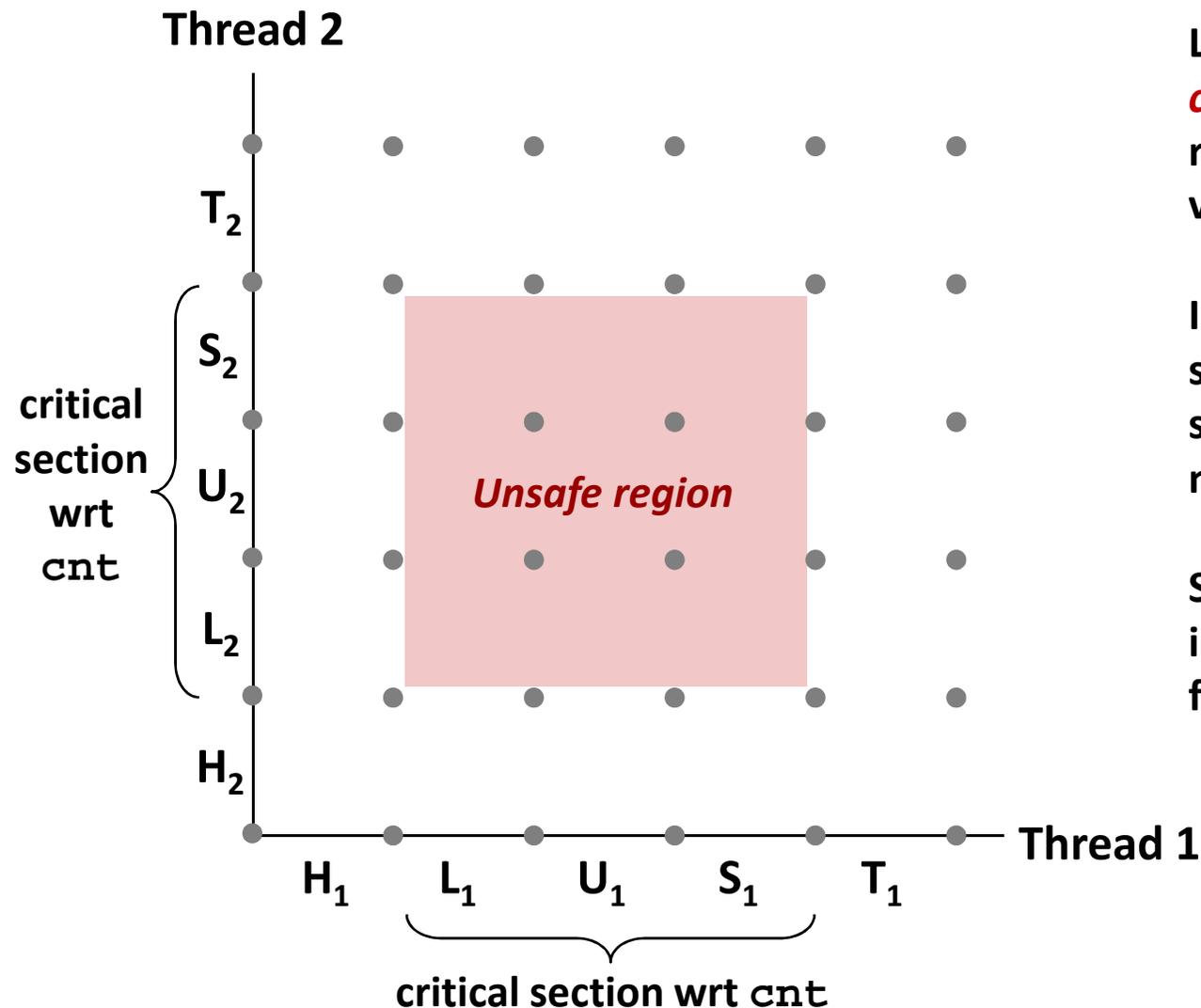


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

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

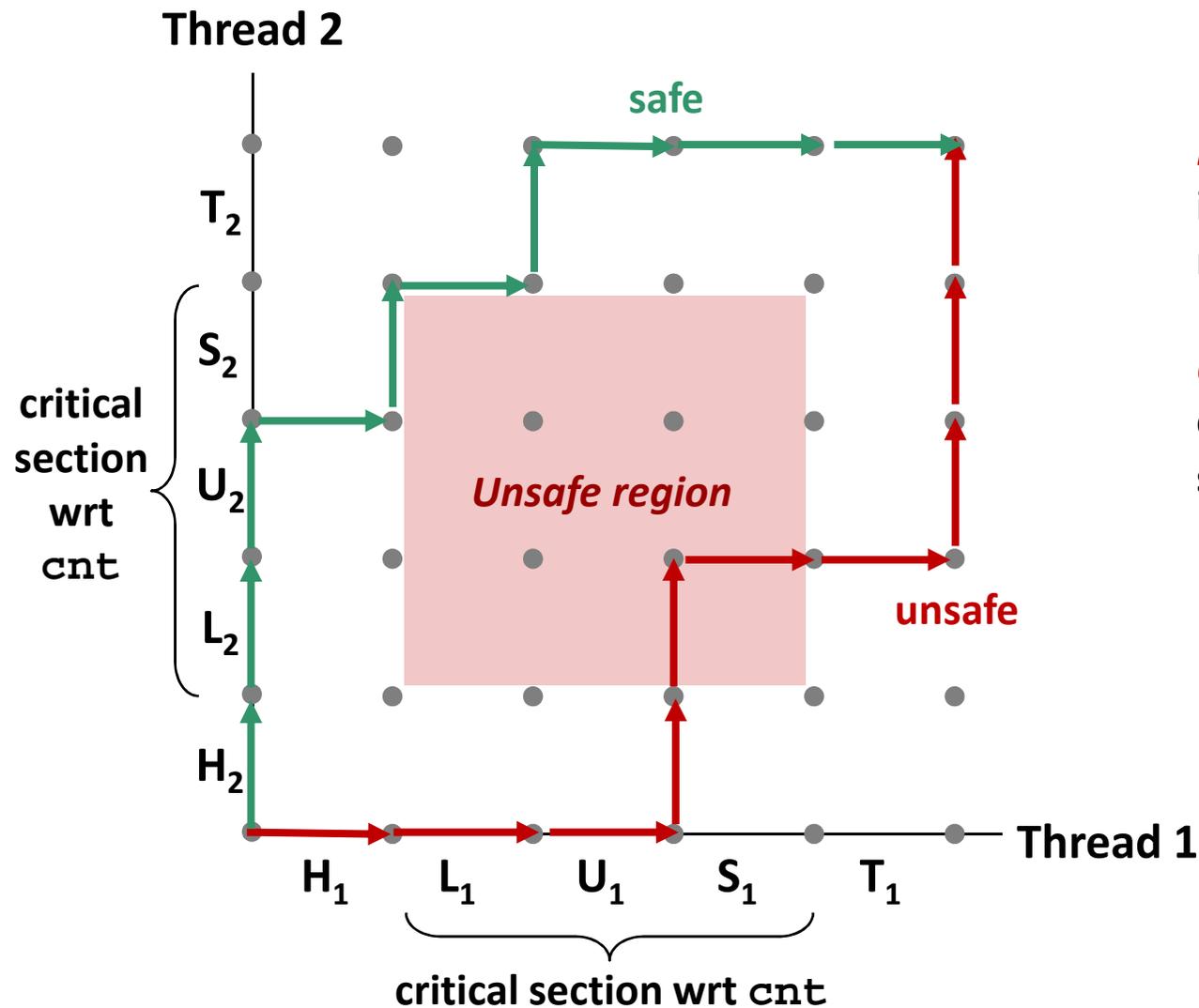


L , U , and S form a *critical section* with respect to the shared variable `cnt`

Instructions in critical sections (wrt to some shared variable) should not be interleaved

Sets of states where such interleaving occurs form *unsafe regions*

Critical Sections and Unsafe Regions



Definition: A trajectory is *safe* iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt cnt) iff it is safe

Semaphores

- **Question: How can we guarantee a safe trajectory?**
 - We must **synchronize** the threads so that they never enter an unsafe state.
- **Classic solution: Dijkstra's P and V operations on semaphores**
 - **Semaphore:** non-negative global integer synchronization variable
 - P(s): [`while (s == 0) wait(); s--;`]
 - Dutch for "Proberen" (test)
 - V(s): [`s++;`]
 - Dutch for "Verhogen" (increment)
 - OS 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: ($s \geq 0$)**

badcnt.c: Improper Synchronization

```
/* shared */
volatile unsigned int cnt = 0;
#define NITERS 100000000

int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                  count, NULL);
    Pthread_create(&tid2, NULL,
                  count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n",
              cnt);
    else
        printf("OK cnt=%d\n",
              cnt);
}
```

```
/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

How to fix using semaphores?

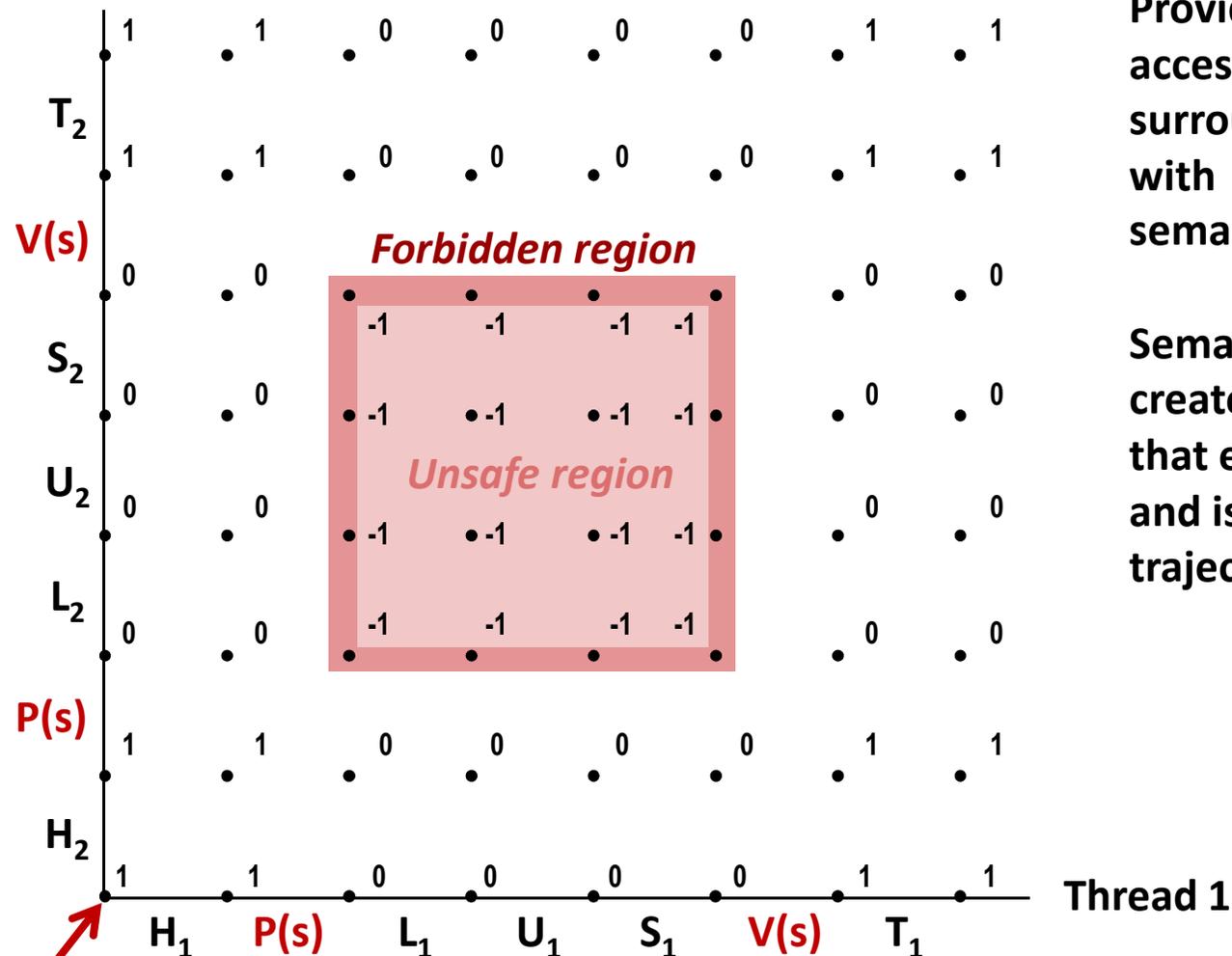
Safe Sharing with Semaphores

- One semaphore per shared variable
- Initially set to 1
- Here is how we would use P and V operations to synchronize the threads that update cnt

```
/* Semaphore s is initially 1 */  
  
/* Thread routine */  
void *count(void *arg)  
{  
    int i;  
  
    for (i=0; i<NITERS; i++) {  
        P(s);  
        cnt++;  
        V(s);  
    }  
    return NULL;  
}
```

Safe Sharing With Semaphores

Thread 2



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 is entered by any trajectory

Initially
 $s = 1$

Wrappers on POSIX Semaphores

```
/* Initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process */
void Sem_init(sem_t *sem, int pshared, unsigned int value) {
    if (sem_init(sem, pshared, value) < 0)
        unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
    if (sem_wait(sem))
        unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
    if (sem_post(sem))
        unix_error("V");
}
```

Sharing With POSIX Semaphores

```

/* properly sync'd counter program */
#include "csapp.h"
#define NITERS 10000000

volatile unsigned int cnt;
sem_t sem;          /* semaphore */

int main() {
    pthread_t tid1, tid2;

    Sem_init(&sem, 0, 1); /* sem=1 */

    /* create 2 threads and wait */
    ...

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

```

```

/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}

```

Warning:
It's really slow!

Today

- Threads: basics
- Synchronization
- **Races, deadlocks, thread safety**

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;
}
```

Where is
the race?

Race Elimination

- Make sure don't have unintended sharing of state

```
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = malloc(sizeof(int));
        *valp = i;
        Pthread_create(&tid[i], NULL, thread, valp);
    }
    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;
}
```

Another worry: Deadlock

- **Processes wait for 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 POSIX Semaphores

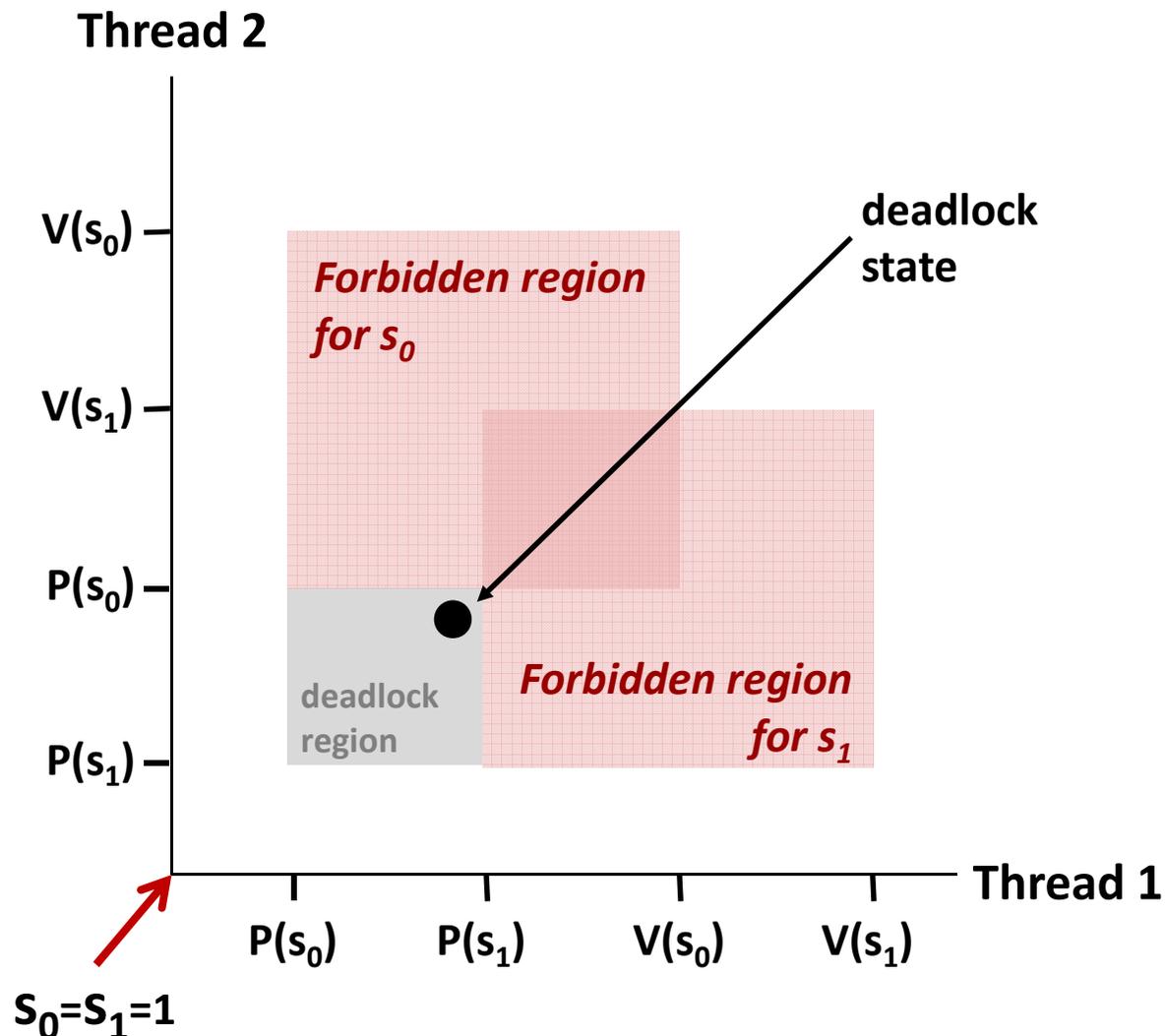
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1],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



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 non-deterministic

Avoiding Deadlock

Acquire shared resources in same order

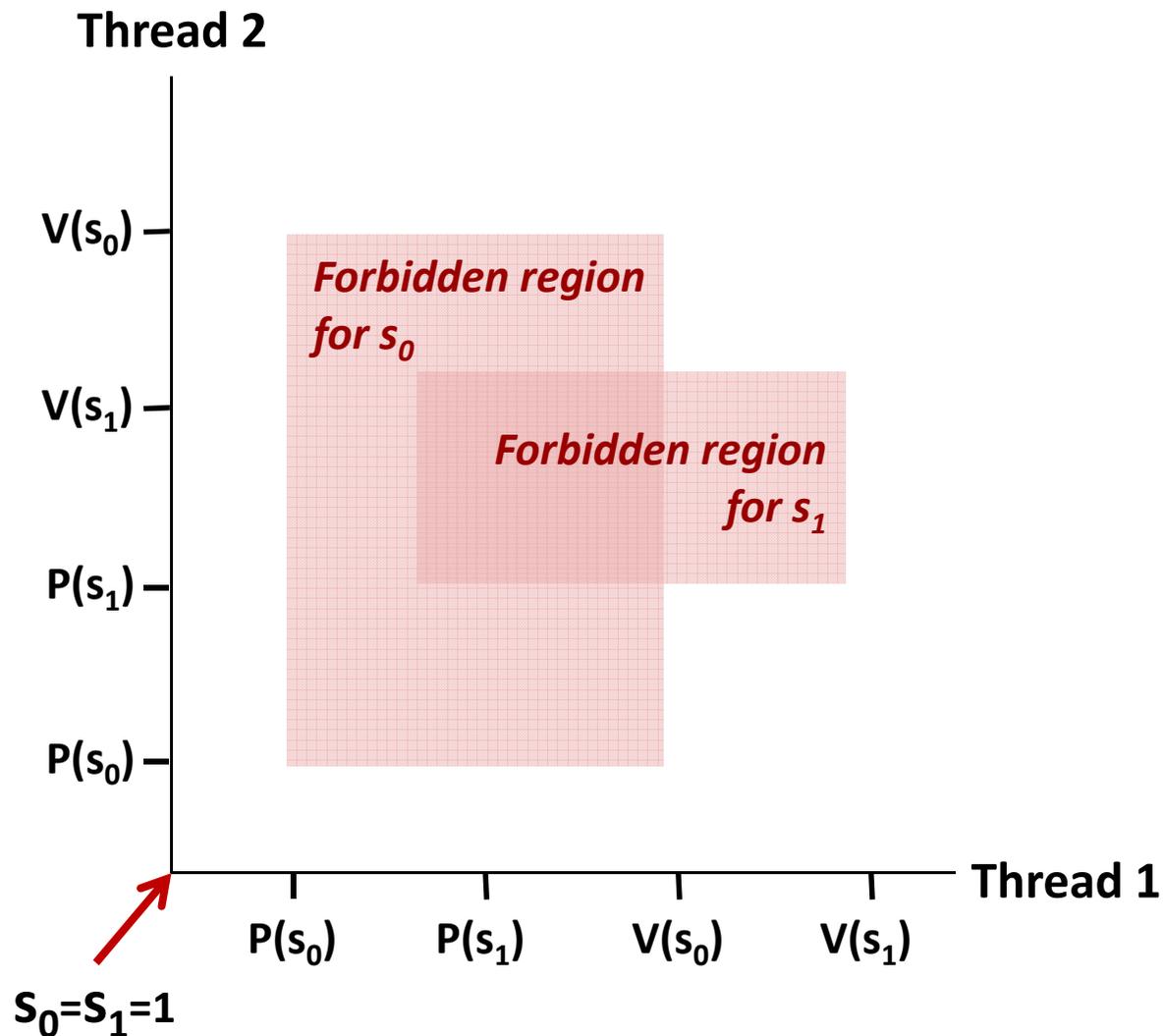
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1],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

Crucial concept: Thread Safety

- **Functions called from a thread (without external synchronization) must be *thread-safe***
 - Meaning: it must always produce correct results when called repeatedly from multiple concurrent threads
- **Some examples of thread-unsafe functions:**
 - Failing to protect shared variables
 - Relying on persistent state across invocations
 - Returning a pointer to a static variable
 - Calling 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
 - e.g., `badcnt` requires 0.5s, `goodcnt` requires 7.9s

Thread-Unsafe Functions (Class 2)

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

```
/* rand: return pseudo-random integer on 0..32767 */
static unsigned int next = 1;
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;
}
```

Making Thread-Safe RNG

- Pass state as part of argument
 - and, thereby, eliminate static state

```
/* rand - 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 must maintain seed

Thread-Unsafe Functions (Class 3)

- Returning a ptr to a static variable
- Fixes:
 - 1. Rewrite code so caller passes pointer to **struct**
 - Issue: Requires changes in caller and callee
 - 2. *Lock-and-copy*
 - Issue: Requires only simple changes in caller (and none in callee)
 - However, caller must free memory

```
struct hostent
*gethostbyname(char name)
{
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}
```

```
hostp = Malloc(...);
gethostbyname_r(name, hostp);
```

```
struct hostent
*gethostbyname_ts(char *name)
{
    struct hostent *q = Malloc(...);
    struct hostent *p;
    P(&mutex); /* lock */
    p = gethostbyname(name);
    *q = *p; /* copy */
    V(&mutex);
    return q;
}
```

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 😊

Thread-Safe Library Functions

- All functions in the Standard C Library (at the back of your K&R text) 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
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(none)
<code>localtime</code>	3	<code>localtime_r</code>
<code>rand</code>	2	<code>rand_r</code>

Threads Summary

- **Threads provide another mechanism for writing concurrent programs**
- **Threads are very popular**
 - Somewhat cheaper than processes
 - Easy to share data between threads
 - Make use of multiple cores for parallel algorithms
- **However, the ease of sharing has a cost:**
 - Easy to introduce subtle synchronization errors
 - Tread carefully with threads!
- **For more info:**
 - D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997