



# Synchronization: Basic

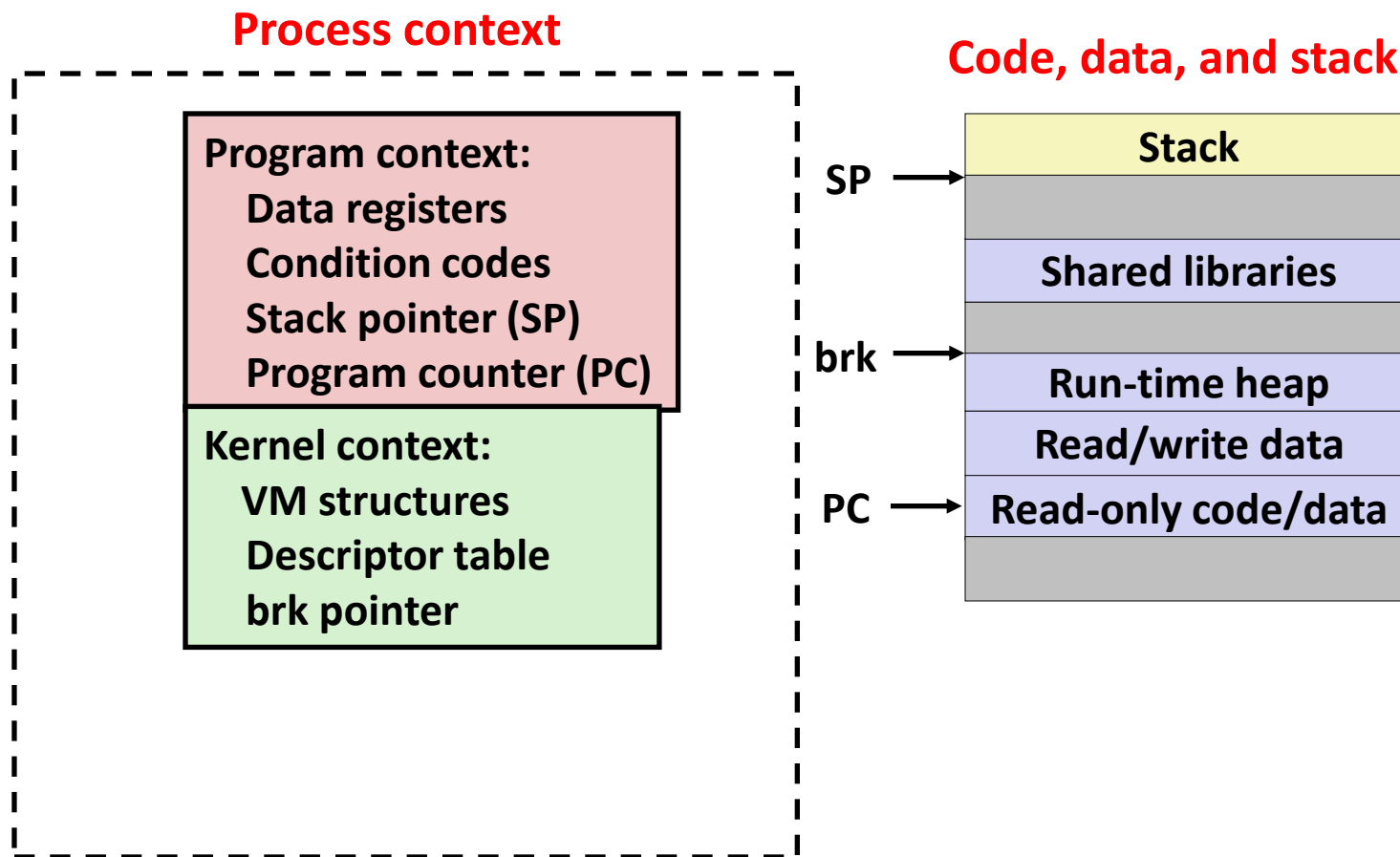
18-213/18-613: Introduction to Computer Systems  
23<sup>rd</sup> Lecture, November 21, 2024

# Today

- **Recap: Threads, races, and deadlocks**
- **Sharing** CSAPP 12.4
- **Mutual exclusion** CSAPP 12.5
- **Semaphores** CSAPP 12.5
- **Producer-Consumer Synchronization** CSAPP 12.5

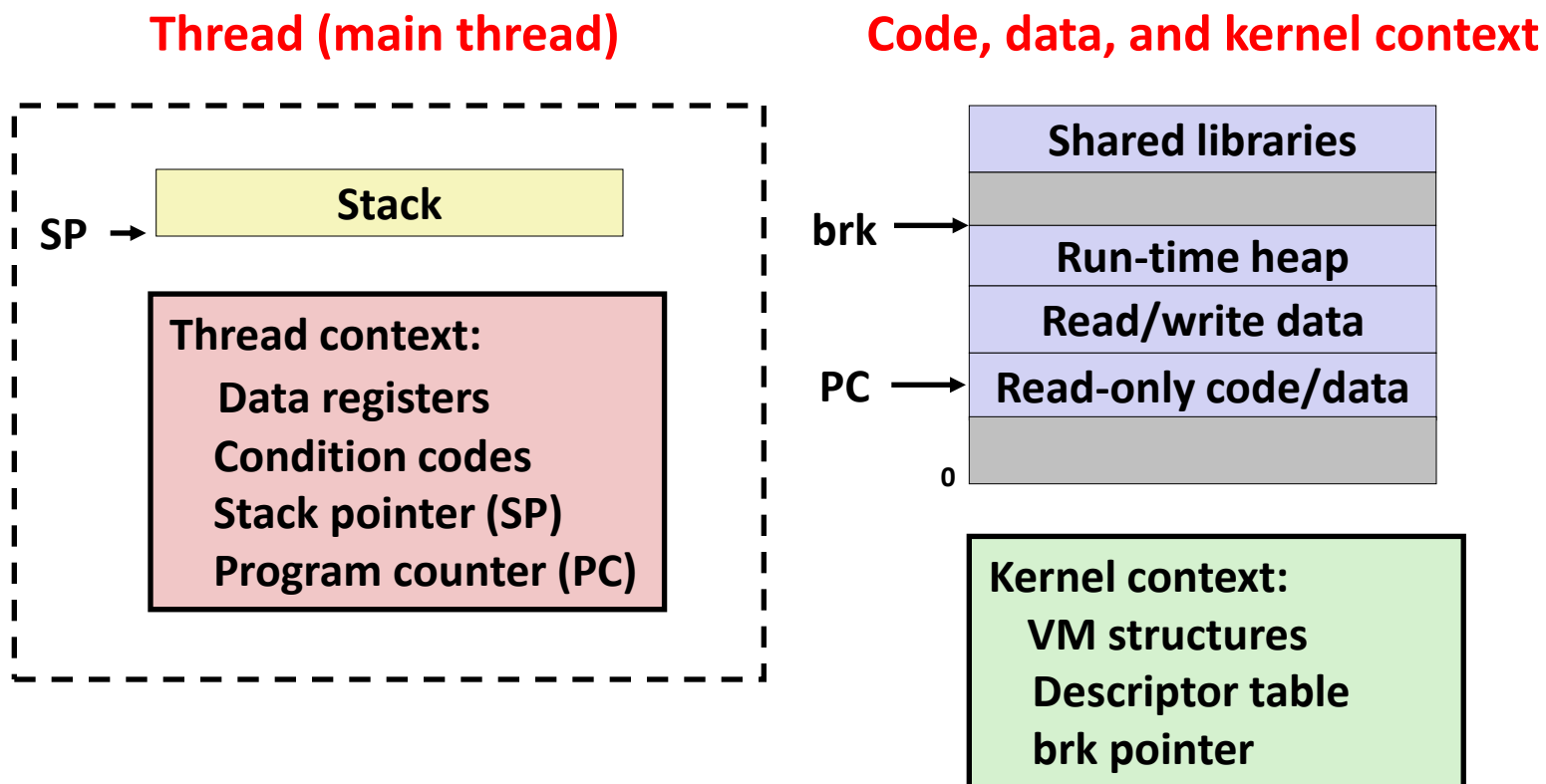
# Traditional View of a Process

- 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)

## Thread 1 (main thread)

## Thread 2 (peer thread)

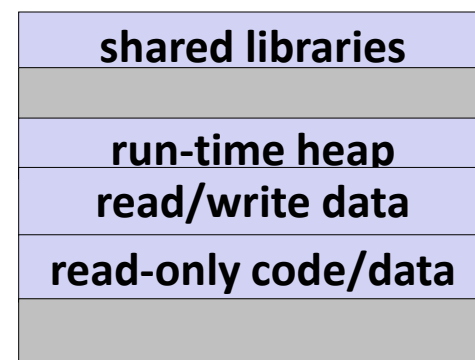
## Shared code and data

stack 1

stack 2

Thread 1 context:  
 Data registers  
 Condition codes  
 $SP_1$   
 $PC_1$

Thread 2 context:  
 Data registers  
 Condition codes  
 $SP_2$   
 $PC_2$

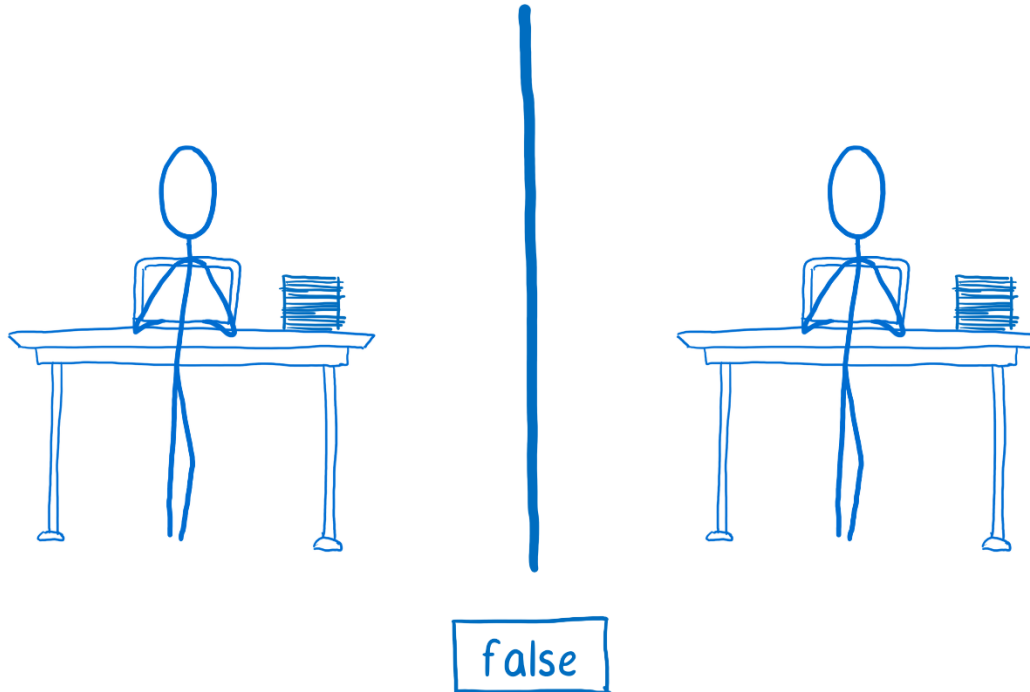


Kernel context:  
 VM structures  
 Descriptor table  
 brk pointer

# Race conditions

- **Event A can happen either before or after event B**
- **The program behaves differently depending on which one happens first**
  - Races are not necessarily bugs!
  - Only if one of the possible behaviors is incorrect

# Race condition example

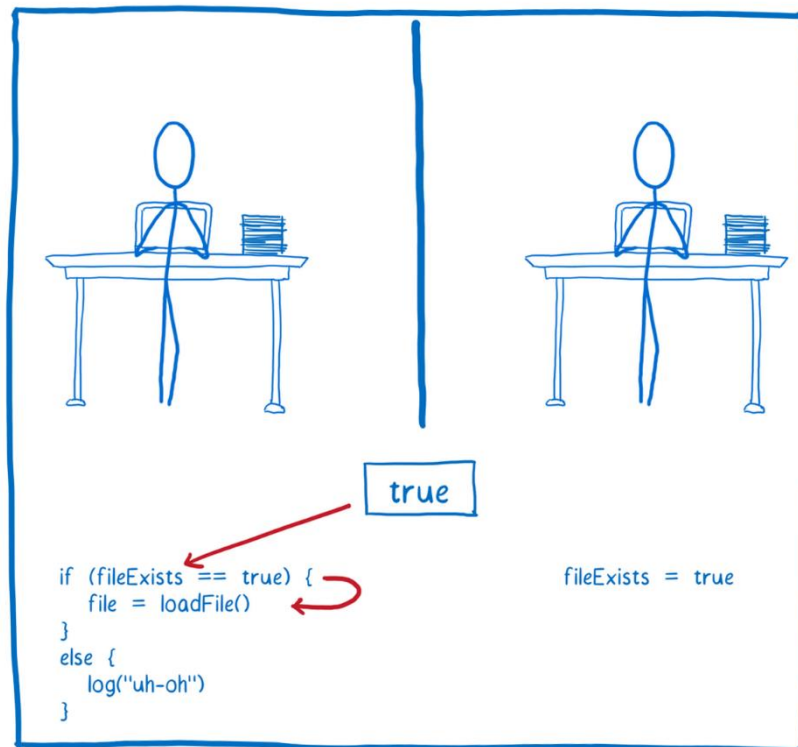
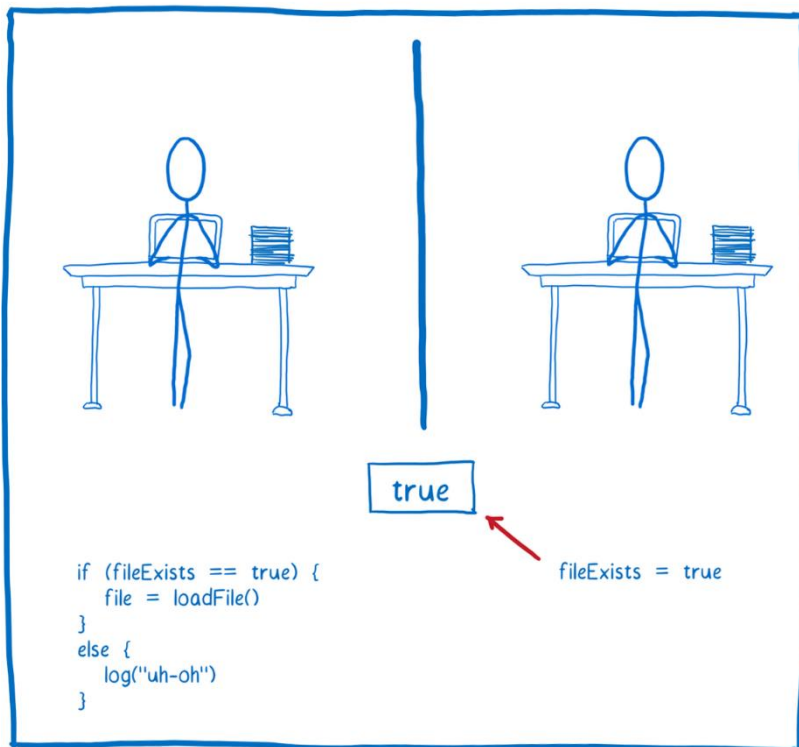


```
if (fileExists == true) {  
    file = loadFile()  
}  
else {  
    log("uh-oh")  
}
```

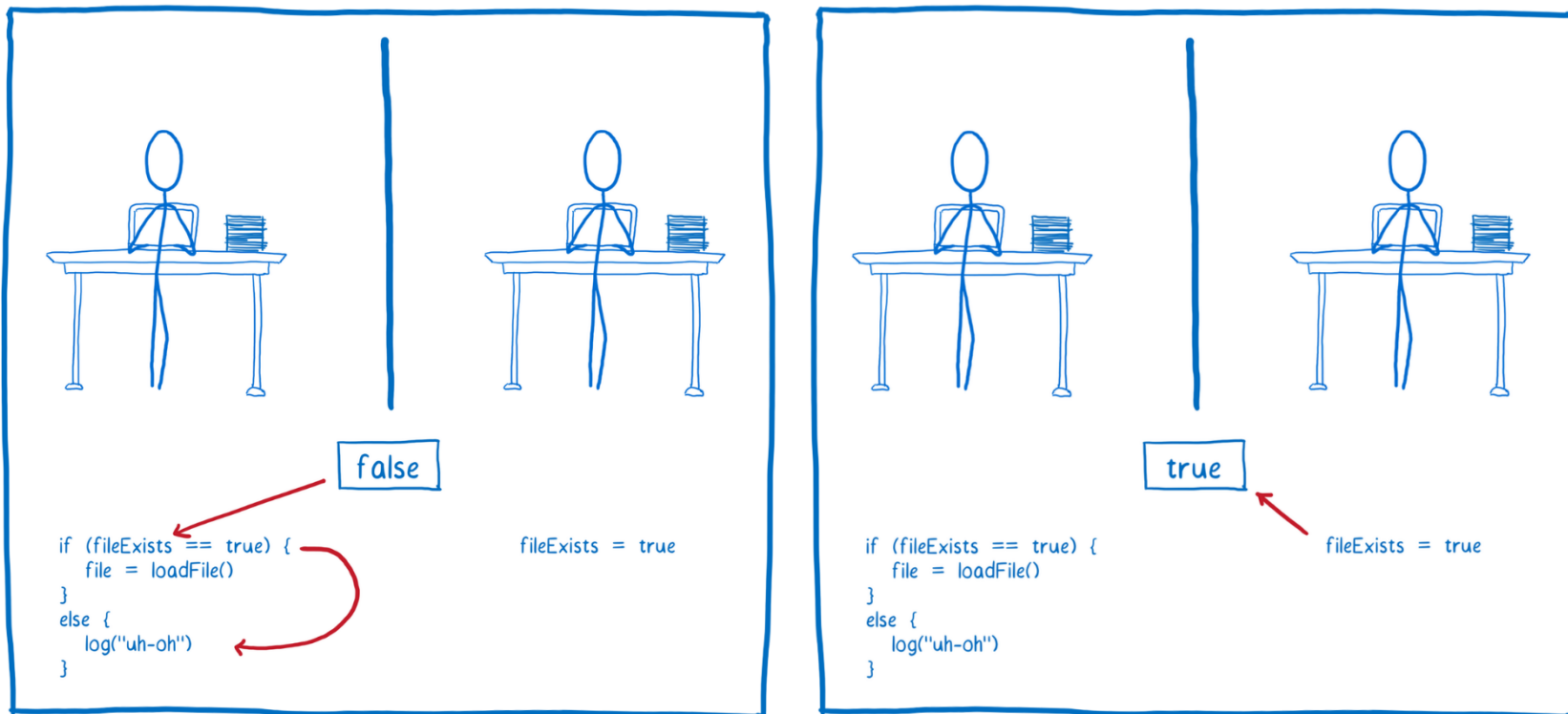
fileExists = true



# Race condition example



# Race condition example



# More race condition examples

- File is deleted, in between when a program checks whether the file exists, and when it opens the file (“time-of-check to time-of-use” race)
- Child exits before parent can add it to the job list (tsh)
- Child thread reads variable after parent has changed it (previous lecture)
- Two threads update the same variable simultaneously (later in this lecture)

# Deadlock

- **Whenever two or more threads/processes/... are stuck waiting for each other to do something**
- **In real life:**
  - Alice cannot put the groceries down until Bob opens the door
  - Bob cannot open the door until Alice hands him the keys
  - Alice cannot hand Bob the keys because she is holding the groceries
- **In programming:**
  - Client is waiting for server to send a message before it closes the connection
  - Server is waiting for client to close the connection before it sends the message (server has a bug)
- **Deadlock is *always* a bug**

# Today

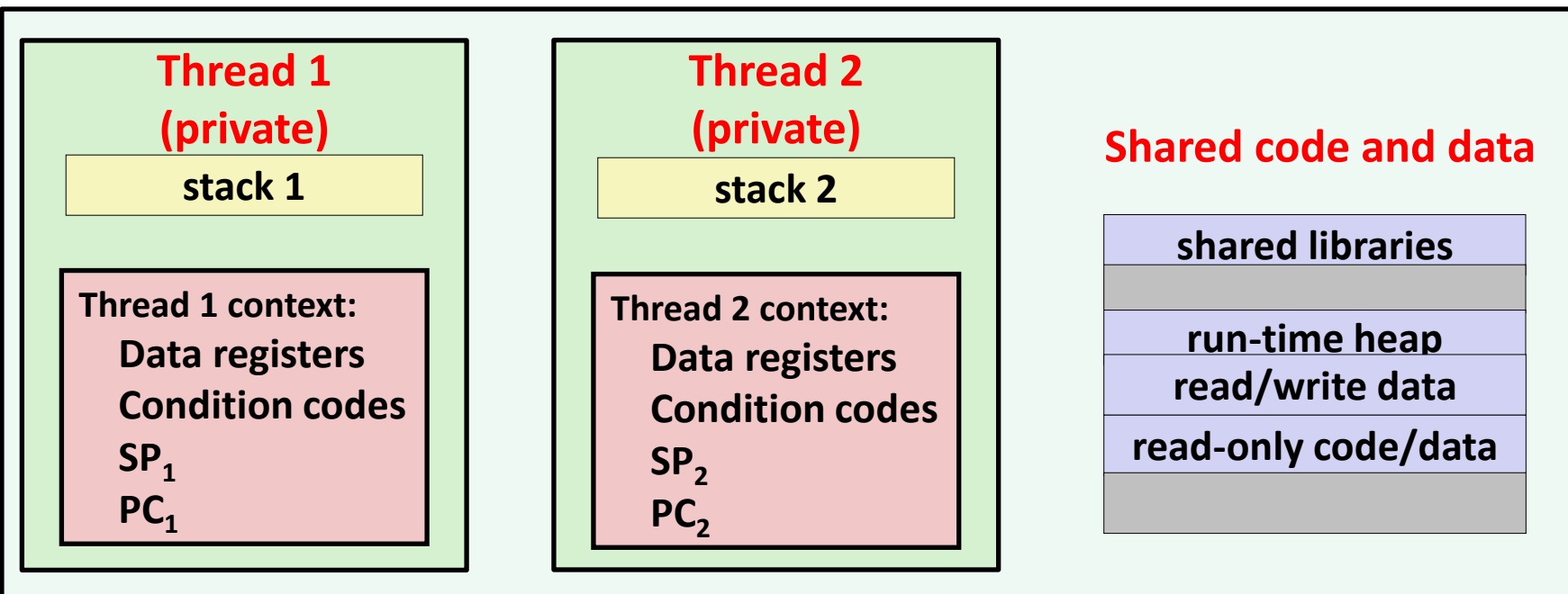
- Recap: Threads, races, and deadlocks
- **Sharing**
- Mutual exclusion
- Semaphores
- Producer-Consumer Synchronization

# 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?

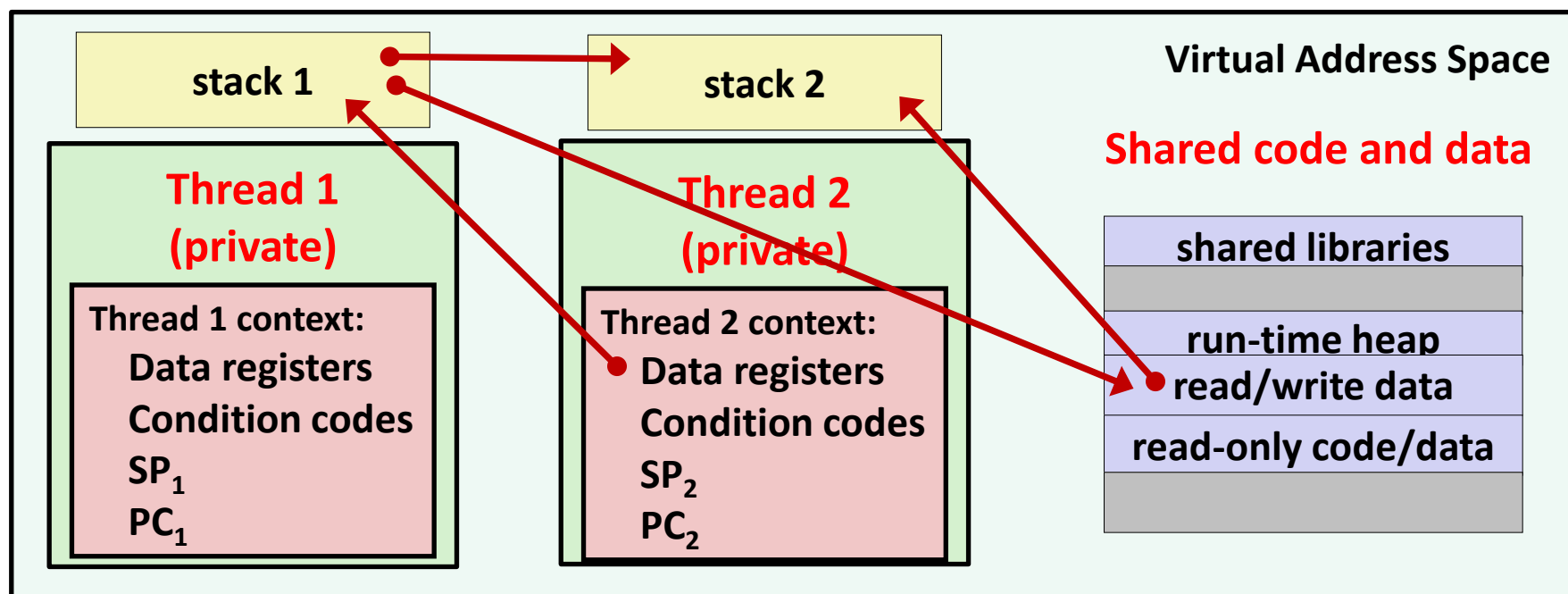
# Threads Memory Model: Conceptual

- 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`)

Notation:  
instance of  
`msgs` in `main`

```
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?

<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>cnt</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

```
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?

<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>cnt</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`, `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 j, niters =
        *((long *)vargp);

    for (j = 0; j < niters; j++)
        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 (j = 0; j < niters; j++)
    cnt++;
```

## Asm code for thread $i$

<pre>movq  (%rdi), %rcx testq %rcx,%rcx jle   .L2 movl  \$0, %eax</pre>	} $H_i$ : Head
<pre>----- .L3: movq  cnt(%rip), %rdx addq  \$1, %rdx movq  %rdx, cnt(%rip)</pre>	} $L_i$ : Load cnt $U_i$ : Update cnt $S_i$ : Store cnt
<pre>----- addq  \$1, %rax cmpq  %rcx, %rax jne   .L3 .L2:</pre>	} $T_i$ : 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

*Note: One of many possible interleavings*

**OK**

*\*For now. In reality, on x86 even non-sequentially consistent interleavings are possible*

# 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 <sub>i</sub>	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	H <sub>1</sub>	-	-	0
1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
2	H <sub>2</sub>	-	-	0
2	L <sub>2</sub>	-	0	0
1	S <sub>1</sub>	1	-	1
1	T <sub>1</sub>	1	-	1
2	U <sub>2</sub>	-	1	1
2	S <sub>2</sub>	-	1	1
2	T <sub>2</sub>	-	1	1

**Oops!**

*(badcnt will print "BOOM!")*

# Concurrent Execution (cont)

- How about this ordering?

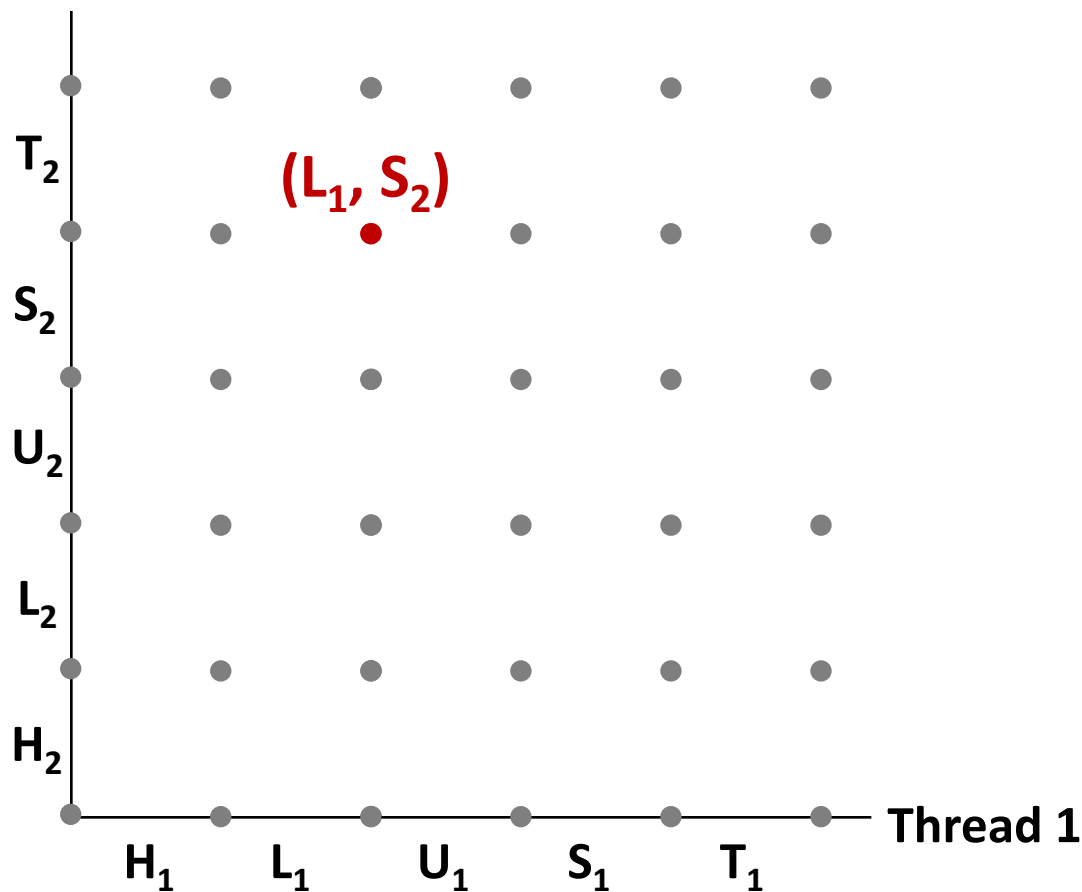
i (thread)	instr <sub>i</sub>	%rdx <sub>1</sub>	%rdx <sub>2</sub>	cnt
1	H <sub>1</sub>			0
1	L <sub>1</sub>	0		
2	H <sub>2</sub>			
2	L <sub>2</sub>		0	
2	U <sub>2</sub>		1	
2	S <sub>2</sub>		1	1
1	U <sub>1</sub>	1		
1	S <sub>1</sub>	1		1
1	T <sub>1</sub>			1
2	T <sub>2</sub>			1

*Oops again!*

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

# Progress Graphs

Thread 2



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

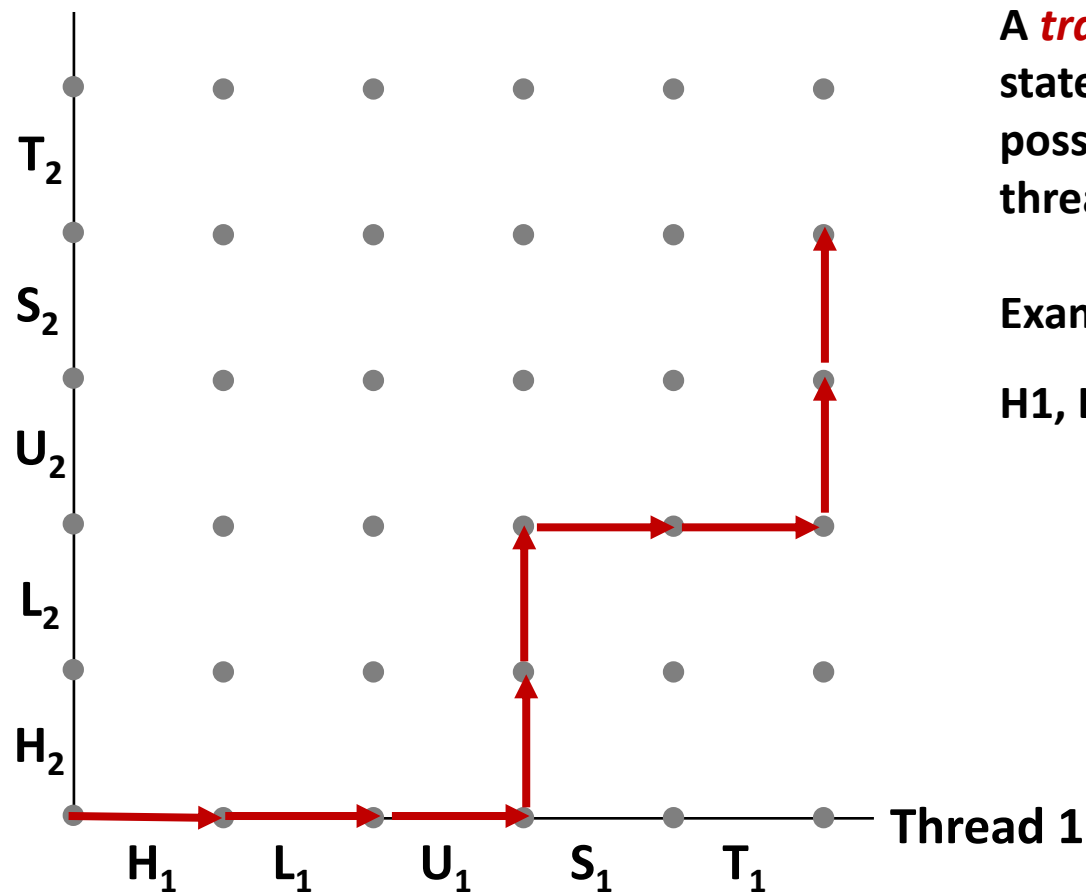
Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* (Inst<sub>1</sub>, Inst<sub>2</sub>).

E.g., (L<sub>1</sub>, S<sub>2</sub>) denotes state where thread 1 has completed L<sub>1</sub> and thread 2 has completed S<sub>2</sub>.

# Trajectories in Progress Graphs

Thread 2



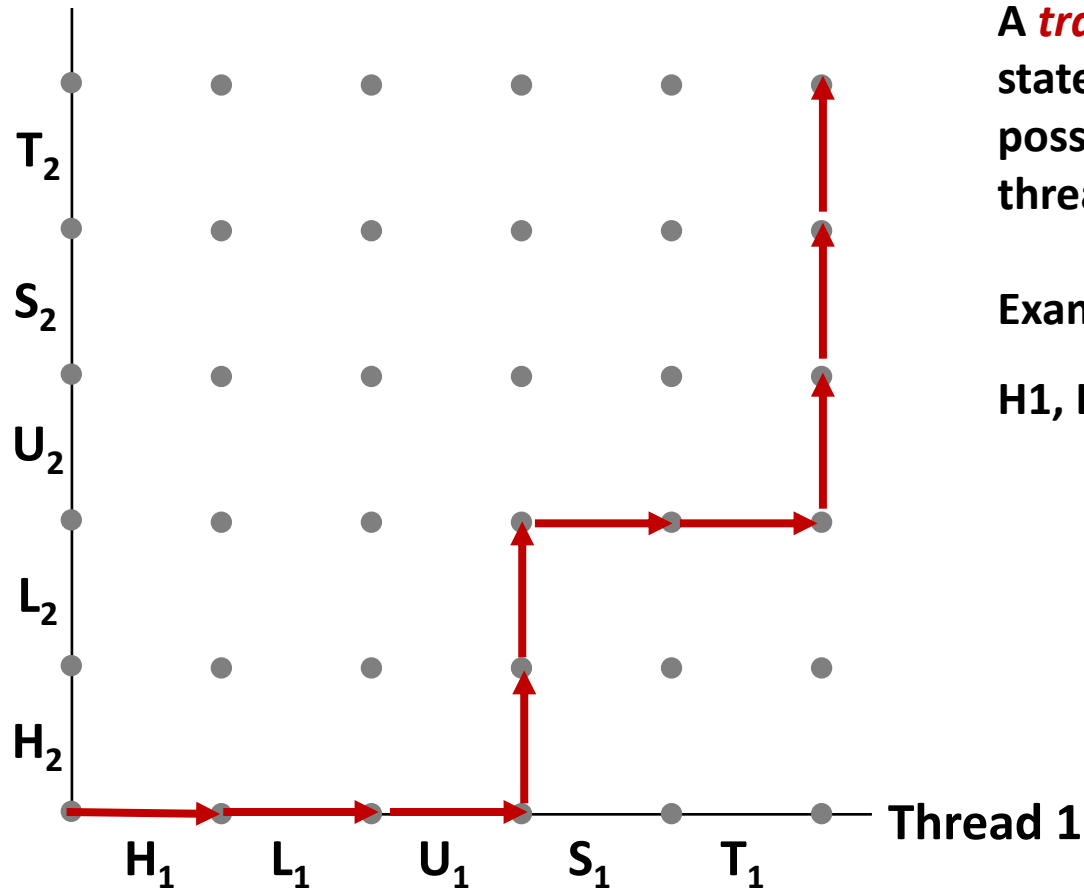
A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2$

# Trajectories in Progress Graphs

Thread 2

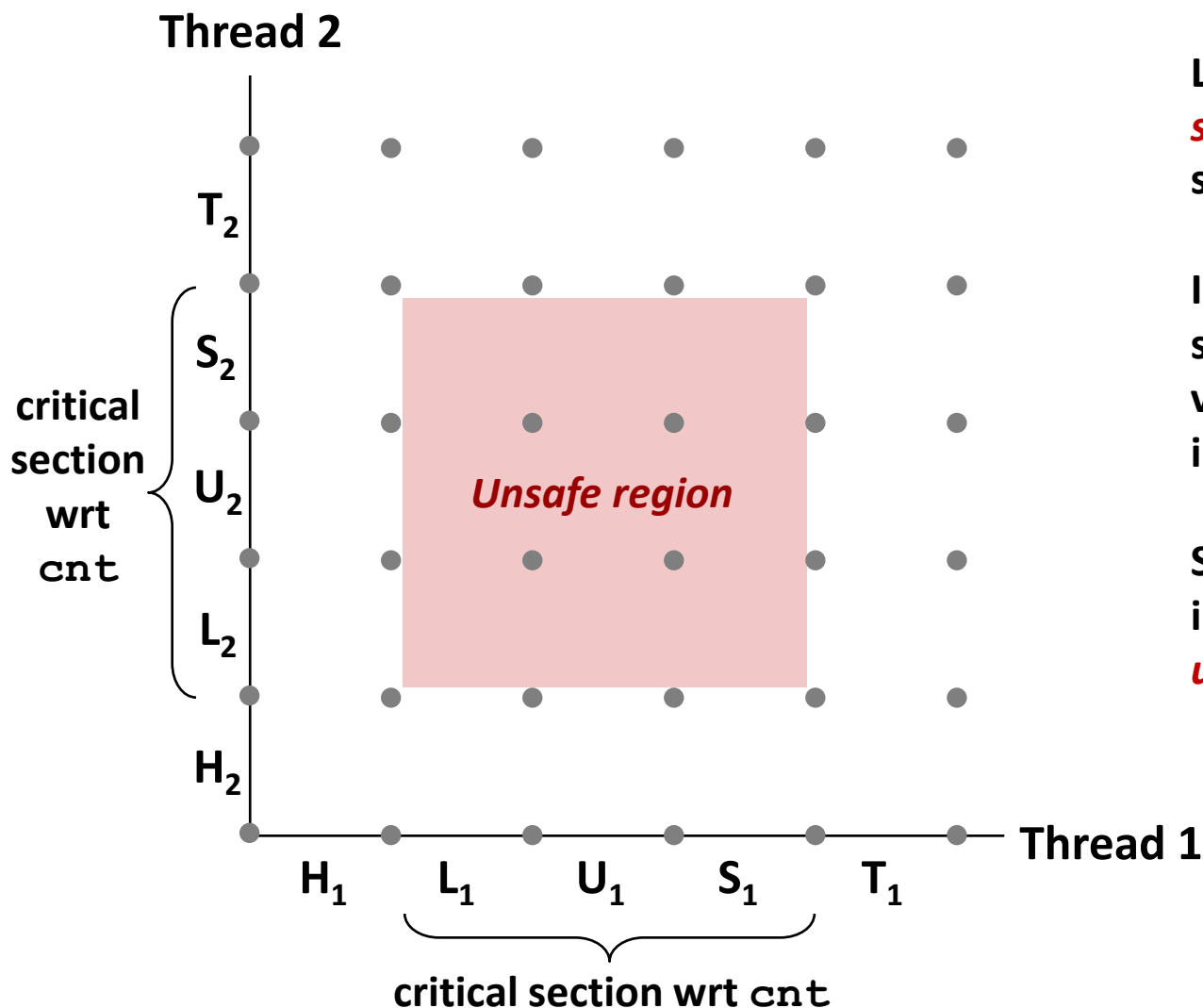


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

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



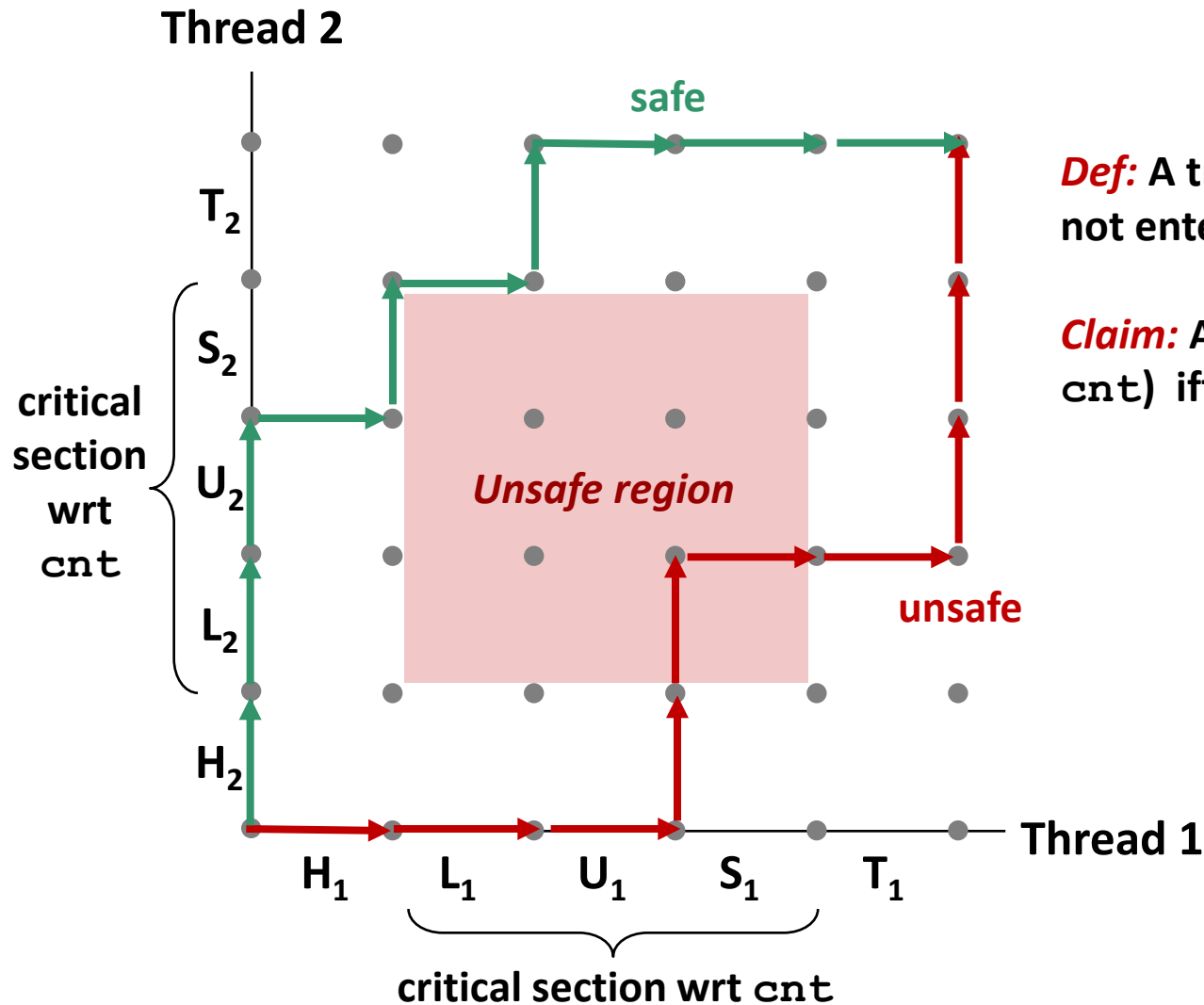
L, U, and S form a **critical section** with respect to the shared variable `cnt`

Instructions in critical sections (wrt 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 j, niters =
        *((long *)vargp);

    for (j = 0; j < niters; j++)
        cnt++;

    return NULL;
}

```

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

# Today

- Threads review
- Sharing
- **Mutual exclusion**
- Semaphores
- Producer-Consumer Synchronization

# 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:**
  - Mutex (pthreads)
  - Semaphores (Edsger Dijkstra)
- **Other approaches (out of our scope)**
  - Condition variables (pthreads)
  - Monitors (Java)

# MUTual EXclusion (mutex)

- ***Mutex***: boolean synchronization variable
- `enum {locked = 0, unlocked = 1}`
- **lock(m)**
  - If the mutex is currently not locked, lock it and return
  - Otherwise, wait (spinning, yielding, etc) and retry
- **unlock(m)**
  - Update the mutex state to unlocked

# MUTual EXclusion (mutex)

- ***Mutex***: boolean synchronization variable \*

- **Swap(\*a, b)**

```
[t = *a; *a = b; return t;]
```

```
// Notation: what's inside the brackets [ ] is indivisible (a.k.a. atomic)
```

```
//           by the magic of hardware / OS
```

- **Lock(m):**

```
while (swap(&m->state, locked) == locked) ;
```

- **Unlock(m):**

```
m->state = unlocked;
```

*\*For now. In reality, many other implementations and design choices (c.f., 15-410, 418, etc).*

# 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 j, niters =
        *((long *)vargp);

    for (j = 0; j < niters; j++)
        cnt++;

    return NULL;
}
```

How can we fix this using synchronization?

# goodmct.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);
}
```

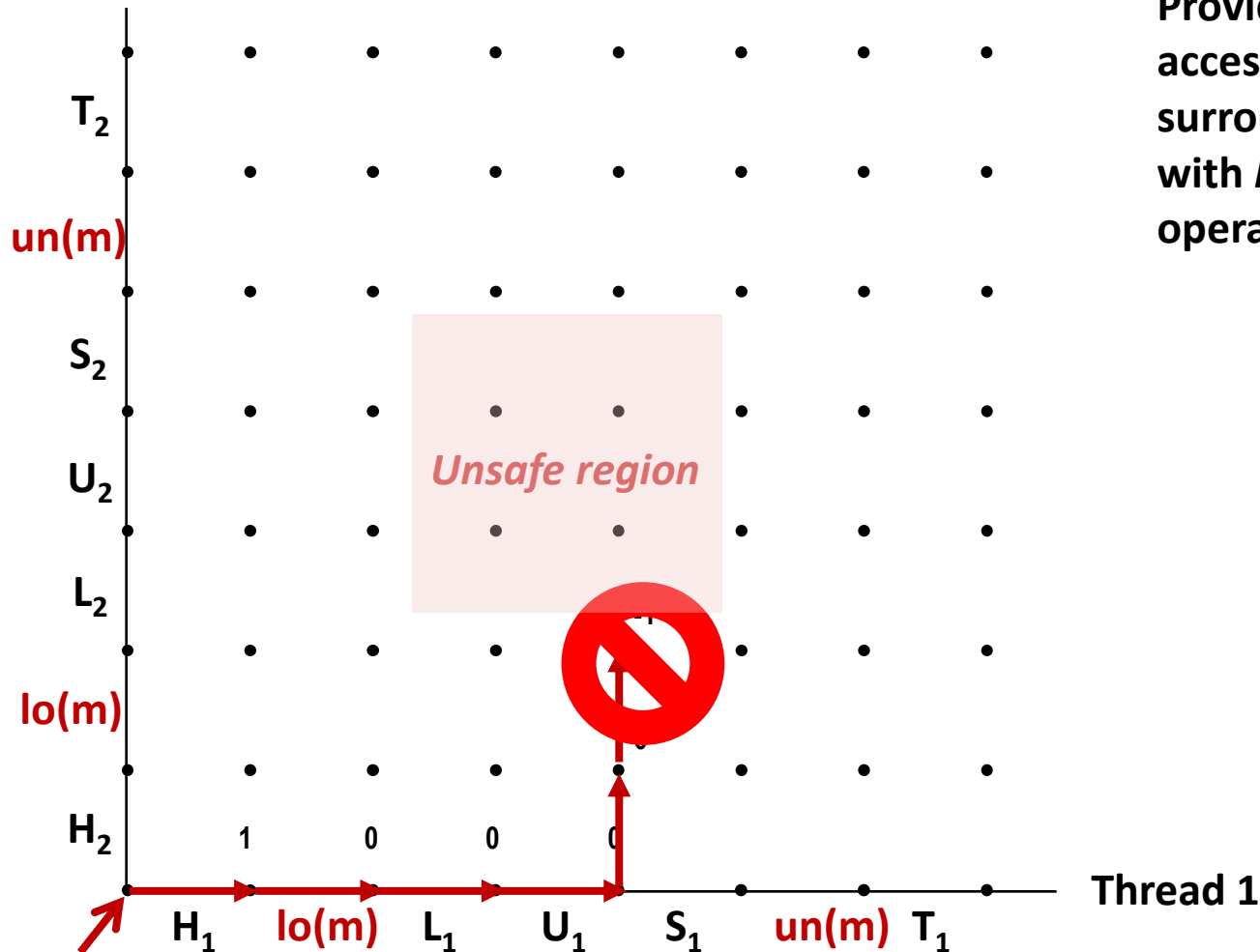
```
linux> ./goodmct 10000
OK cnt=20000
linux> ./goodmct 10000
OK cnt=20000
```

Function	badcnt	goodmct
Time (ms) niters = 10 <sup>6</sup>	12.0	214.0
Slowdown	1.0	17.8



# Why Mutexes Work

Thread 2



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Initially

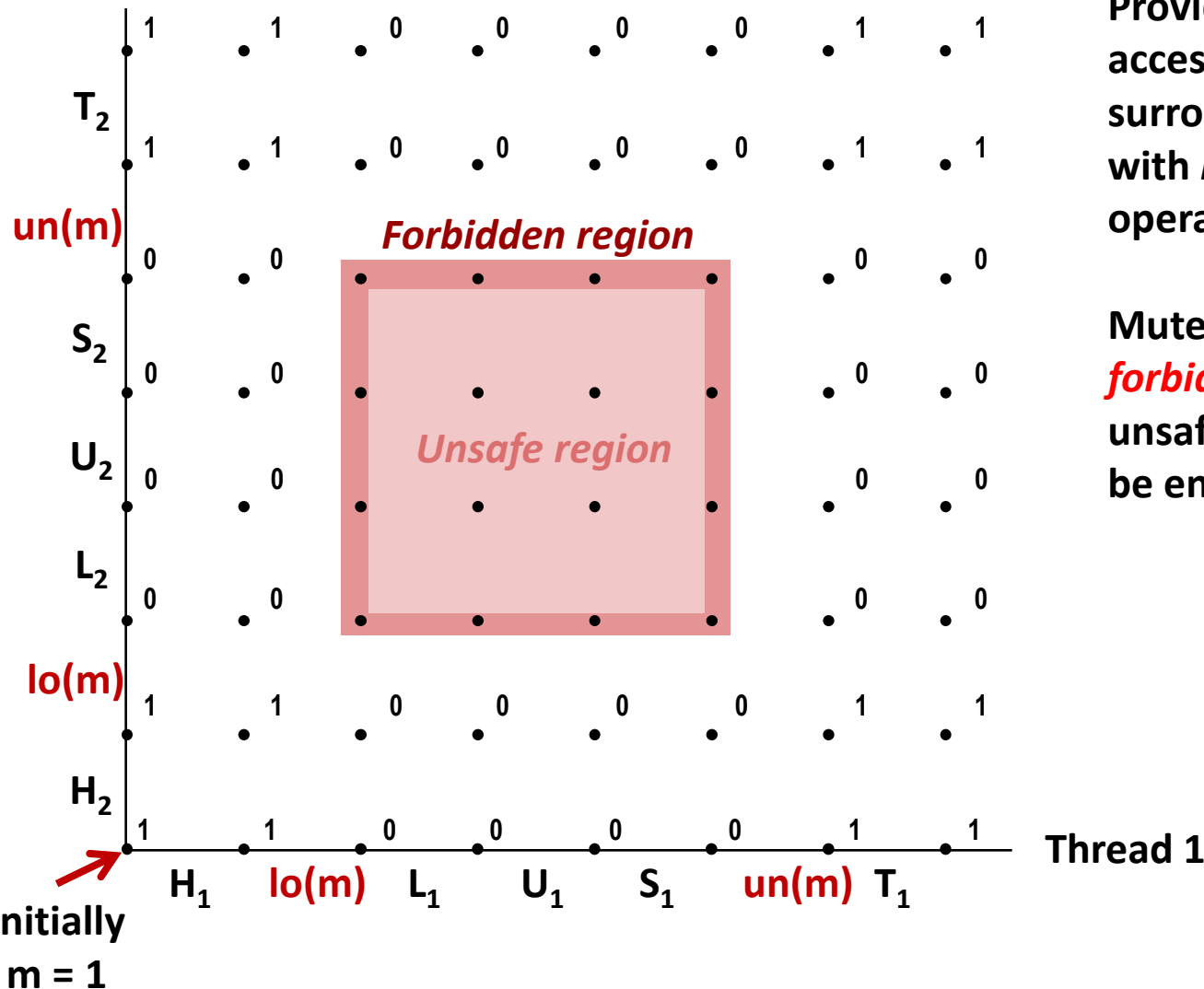
$m = 1$





# Why Mutexes Work

Thread 2



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Mutex invariant creates a **forbidden region** that encloses unsafe region and that cannot be entered by any trajectory.

Thread 1

# Today

- Threads review
- Sharing
- Mutual exclusion
- **Semaphores**
- **Producer-Consumer Synchronization**

# 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:  $s \geq 0$**

# 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/atomically**
  - 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:  $s \geq 0$**

# 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"

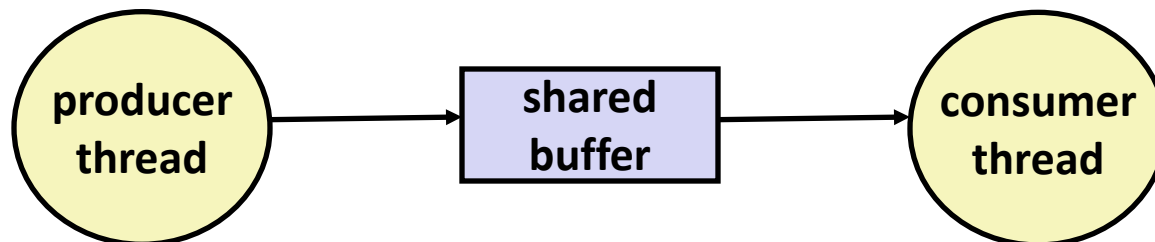
void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```



# 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.
  - Use binary semaphores to notify other threads.
- **The Producer-Consumer Problem**
  - Mediating interactions between processes that generate information and that then make use of that information

# Producer-Consumer Problem



## ■ Common synchronization pattern:

- Producer waits for empty *slot*, inserts item in buffer, and notifies consumer
- Consumer waits for *item*, removes it from buffer, and notifies producer

## ■ Examples

- Multimedia processing:
  - Producer creates 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 1-element Buffer

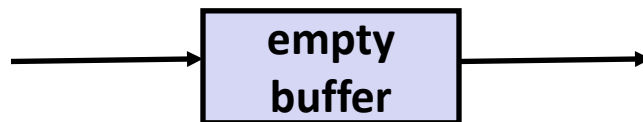
- Maintain two semaphores: `full` + `empty`

`full`

0

`empty`

1

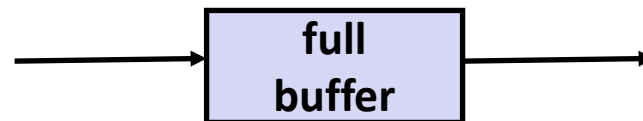


`full`

1

`empty`

0



# Producer-Consumer on 1-element Buffer

```
#include "csapp.h"

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;
```

```
int main(int argc, char** argv) {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* Initialize the semaphores */ Initial
    Sem_init(&shared.empty, 0, 1); value
    Sem_init(&shared.full, 0, 0);

    /* Create threads and wait */
    Pthread_create(&tid_producer, NULL,
                  producer, NULL);
    Pthread_create(&tid_consumer, NULL,
                  consumer, NULL);

    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);

    return 0;
}
```

# Producer-Consumer on 1-element Buffer

Initially: `empty==1, full==0`

## Producer Thread

```
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Produce item */
        item = i;
        printf("produced %d\n",
            item);

        /* Write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

## Consumer Thread

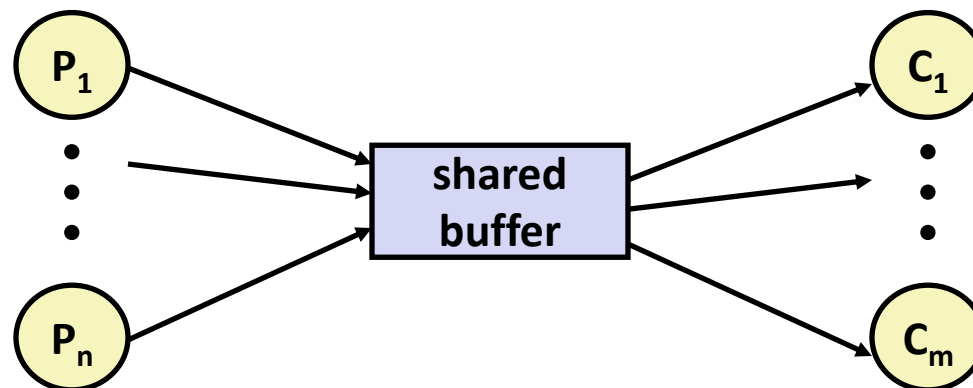
```
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* Consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```

# Why 2 Semaphores for 1-Entry Buffer?

- Consider multiple producers & multiple consumers



- Producers will contend with each other to get **empty**
- Consumers will contend with each other to get **full**

## Producers

```
P(&shared.empty);
shared.buf = item;
V(&shared.full);
```

empty



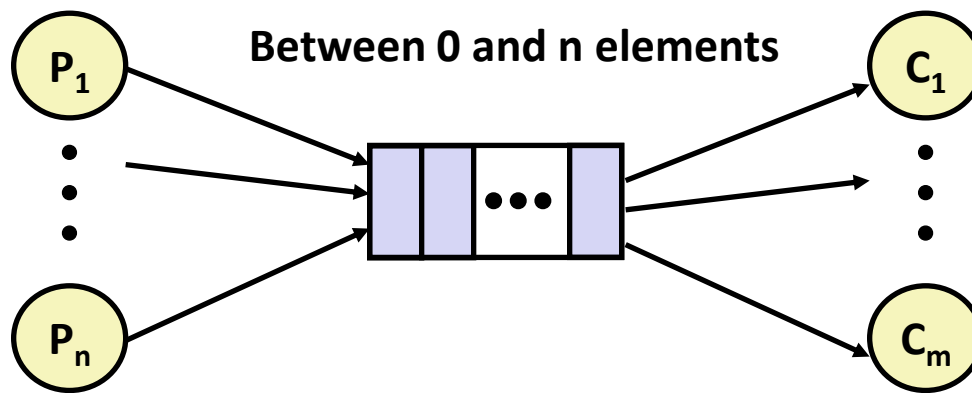
full



## Consumers

```
P(&shared.full);
item = shared.buf;
V(&shared.empty);
```

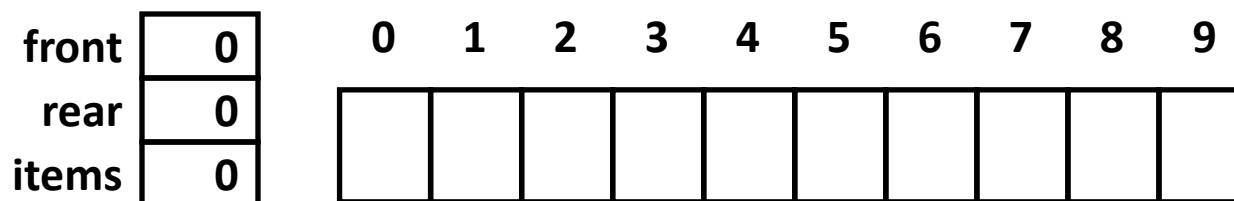
# Producer-Consumer on an $n$ -element Buffer



- Implemented using a shared buffer package called `sbuf`.

# Circular Buffer (n = 10)

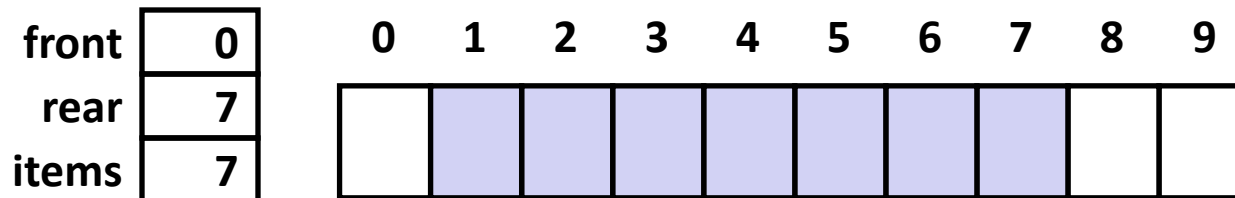
- Store elements in array of size n
- items: number of elements in buffer
- Empty buffer:
  - front = rear
- Nonempty buffer
  - rear: index of most recently inserted element
  - front: (index of next element to remove – 1) mod n
- Initially:



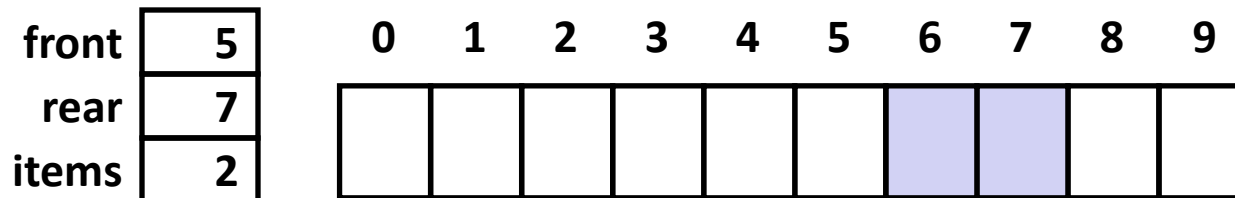


# Circular Buffer Operation (n = 10)

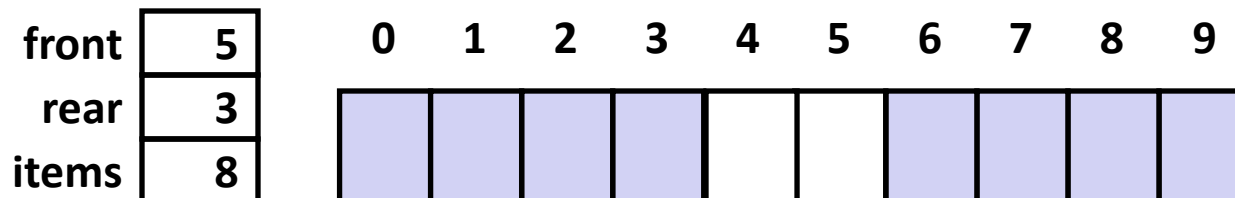
## ■ Insert 7 elements



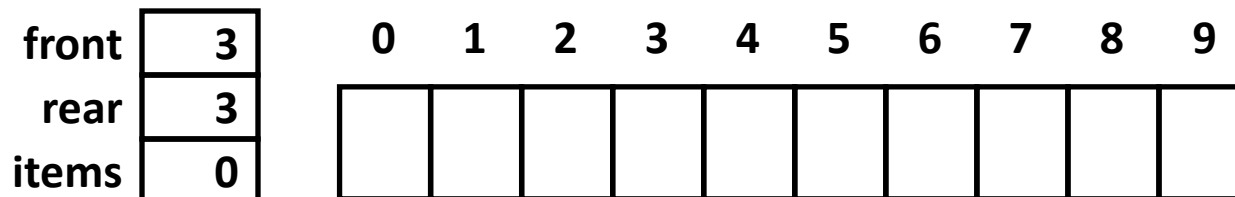
## ■ Remove 5 elements



## ■ Insert 6 elements



## ■ Remove 8 elements



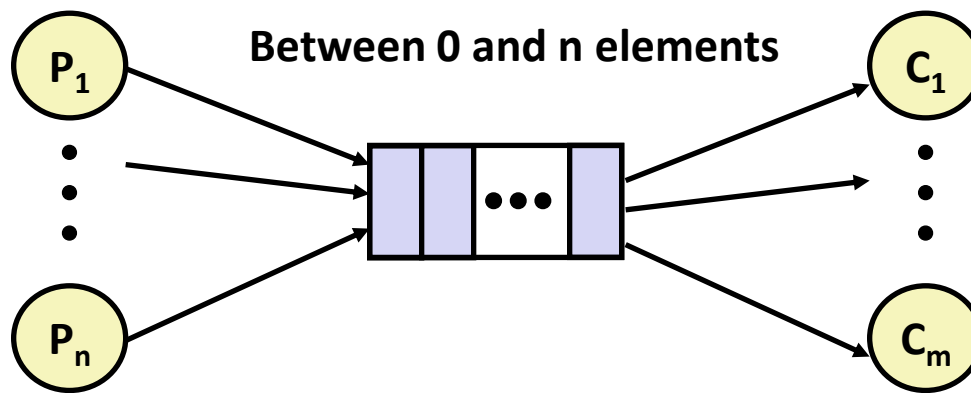
# Sequential Circular Buffer Code

```
init(int v)
{
    items = front = rear = 0;
}

insert(int v)
{
    if (items >= n)
        error();
    if (++rear >= n) rear = 0;
    buf[rear] = v;
    items++;
}

int remove()
{
    if (items == 0)
        error();
    if (++front >= n) front = 0;
    int v = buf[front];
    items--;
    return v;
}
```

# Producer-Consumer on an $n$ -element Buffer



- **Requires a mutex and two counting semaphores:**
  - `mutex`: enforces mutually exclusive access to the buffer and counters
  - `slots`: counts the available slots in the buffer
  - `items`: counts the available items in the buffer
- **Makes use of general semaphores**
  - Will range in value from 0 to  $n$

# sbuf Package - Declarations

```
#include "csapp.h"

typedef struct {
    int *buf;          /* Buffer array */
    int n;            /* Maximum number of slots */
    int front;        /* buf[front+1 (mod n)] is first item */
    int rear;         /* buf[rear] is 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.h

# sbuf Package - Implementation

## Initializing and deinitializing 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 zero items */
}

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

sbuf.c

# sbuf Package - Implementation

Inserting an item into a shared buffer:

```
/* Insert 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          */
    if (++sp->rear >= sp->n) /* Increment index (mod n) */
        sp->rear = 0;
    sp->buf[sp->rear] = item; /* Insert the item          */
    V(&sp->mutex);          /* Unlock the buffer        */
    V(&sp->items);          /* Announce available item */
}
```

sbuf.c

# sbuf Package - Implementation

## Removing an item from a shared buffer:

```
/* 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 */
    if (++sp->front >= sp->n) /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front]; /* Remove the item */
    V(&sp->mutex);          /* Unlock the buffer */
    V(&sp->slots);          /* Announce available slot */
    return item;
}
```

sbuf.c

# 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**
  - E.g., using mutex lock and unlock, semaphore P and V
- **Semaphores are a fundamental mechanism for enforcing mutual exclusion**
  - And can also support producer-consumer synchronization