Systems Programming

Synchronization: Basics

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.

Today

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

Shared Variables in Threaded C Programs

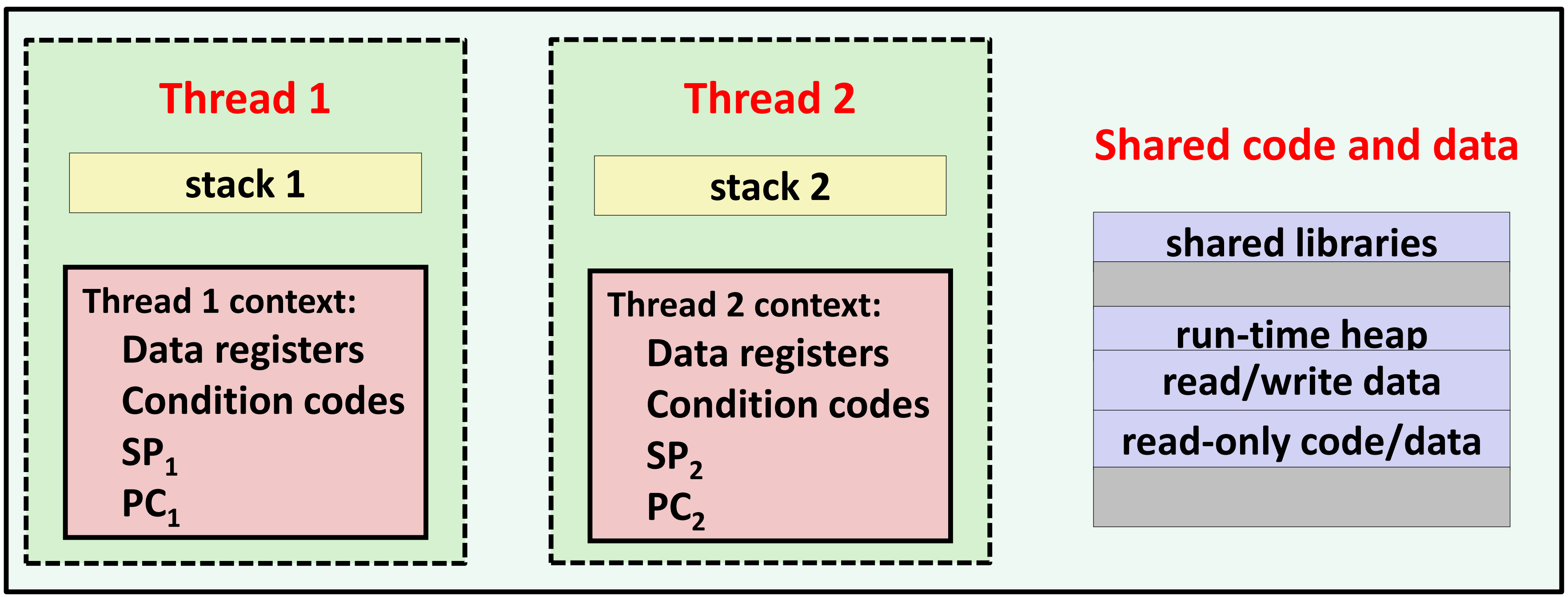
- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

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

- **Requires to answer 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?

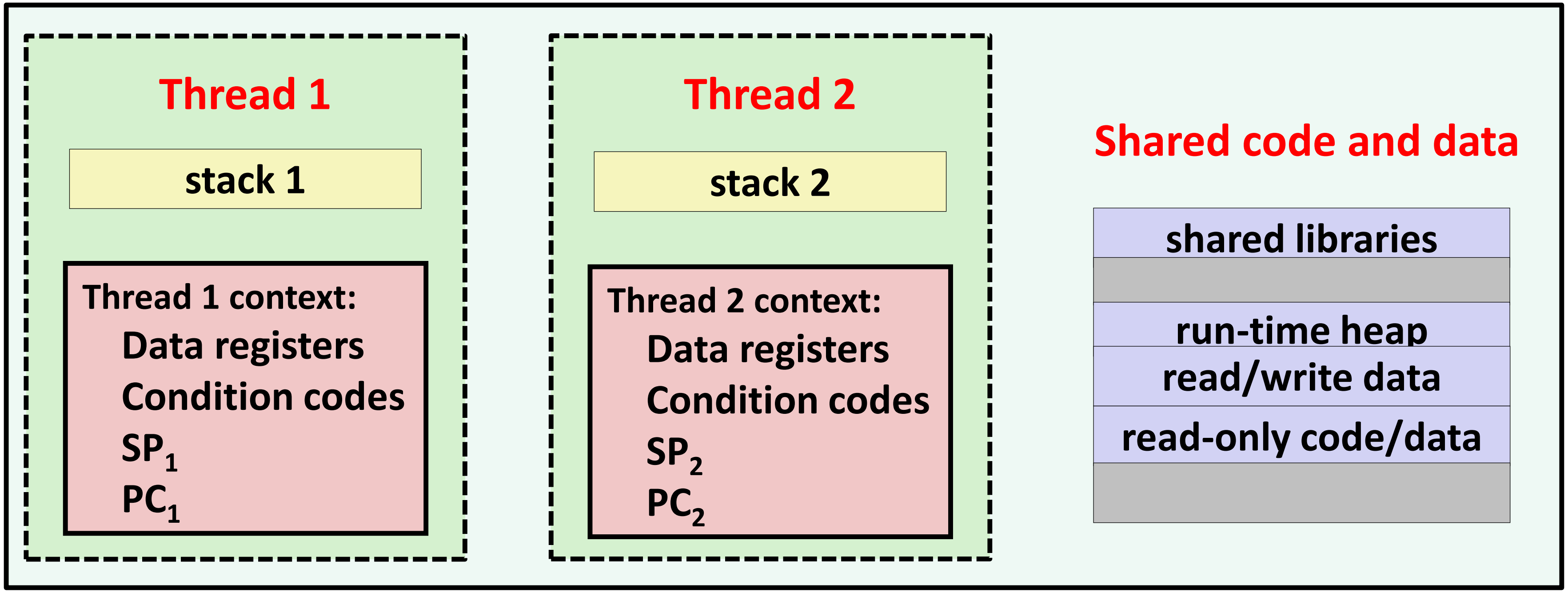
Threads Memory Model: Conceptual

- **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:**
  - Any thread can read and write entire memory or other threads



Passing an argument to a thread: #1. Pass by reference (individual storage)

```c
// #1. Pass through malloc/free
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,
                     (void *)p);
   }
   for (i = 0; i < N; i++)
     pthread_join(tids[i], NULL);
   check();
}
```

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

```c
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");
}
```

Passing an argument to a thread:
#1. Pass by reference (individual storage)

```
// #1. Pass through malloc/free
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,
                     (void *)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;
}
```

- **Use malloc to create a per thread heap allocated place in memory for the argument**
- **Remember to free in thread!**
- **Producer-consumer pattern**

Passing an argument to a thread:
#2. Pass by value

```
// #2. Pass through int casting
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;
}
```

- Good to use cast since sizeof(long) <= sizeof(void*)
- Cast does NOT change bits

Passing an argument to a thread:
#3. Pass by reference (same storage)

```c
// #3. Pass a variable's reference
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();
}
```

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

- **This is WRONG!**
- `&i` points to same location for all threads!
- Creates **a data race!**

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;
}
```

Reference "main thread's stack" indirectly through global ptr variable

A common way to pass a single argument to a thread routine

Mapping Variables 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

```
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

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

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

Mapping Variable Instances to Memory

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

Local vars: 1 instance [main thread's stack]

```
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.0` [thread 0's stack],
`myid.1` [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 thread 0?*	*Referenced by thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i`	yes	no	no
`msgs`	yes	yes	yes
`myid.0`	no	yes	no
`myid.1`	no	no	yes

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL, thread,(void *)i);
    pthread_exit(NULL);}
```

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

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

Shared Variable Analysis

- **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i`	yes	no	no
`msgs`	yes	yes	yes
`myid.0`	no	yes	no
`myid.1`	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 */
long niters = 10000;

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL,
                   thread, NULL);
    pthread_create(&tid2, NULL,
                   thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt == (2 * niters))
        printf("OK cnt=%ld\n", cnt);
    else
        printf("BOOM! cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i;
    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
$./badcnt
OK cnt=20000

$./badcnt
BOOM! cnt=13051
```

`cnt` should be 20,000.

What went wrong?

Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i : Head

L_i : Load cnt
U_i : Update cnt
S_i : Store cnt

T_i : Tail

Concurrent Execution

- **There can be many possible sequential orderings**
  - 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

Time

OK

**In reality, CPU may perform non-sequential execution orders (e.g., speculative execution), but we "safely" ignore such non-sequential issues.*

Concurrent Execution

- **There can be many possible sequential orderings**
  - 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

Time

Thread 1 critical section

Thread 2 critical section

OK

Concurrent Execution: Incorrect ordering #1

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

Time

Concurrent Execution: Incorrect ordering #2

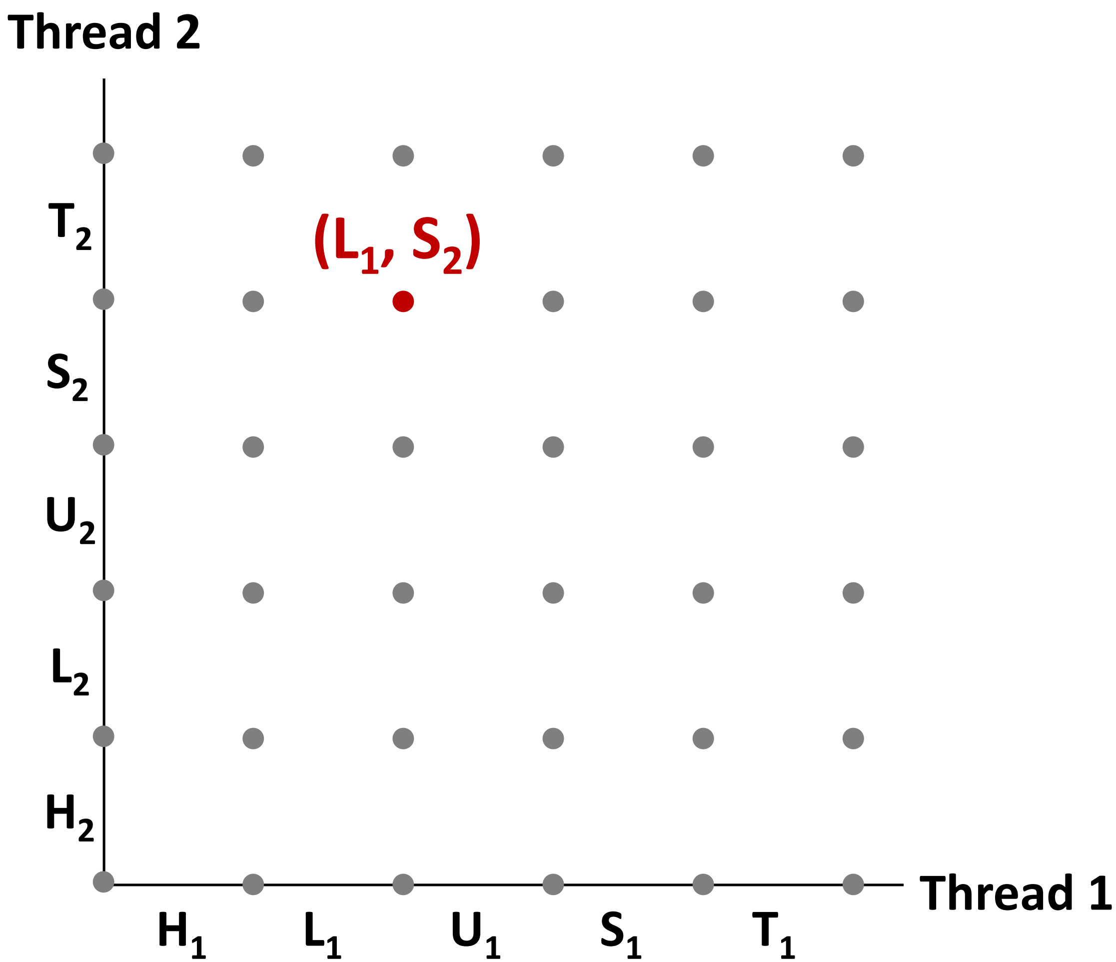
i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			1
2	T_2			1

Time

Oops!

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

Progress Graphs



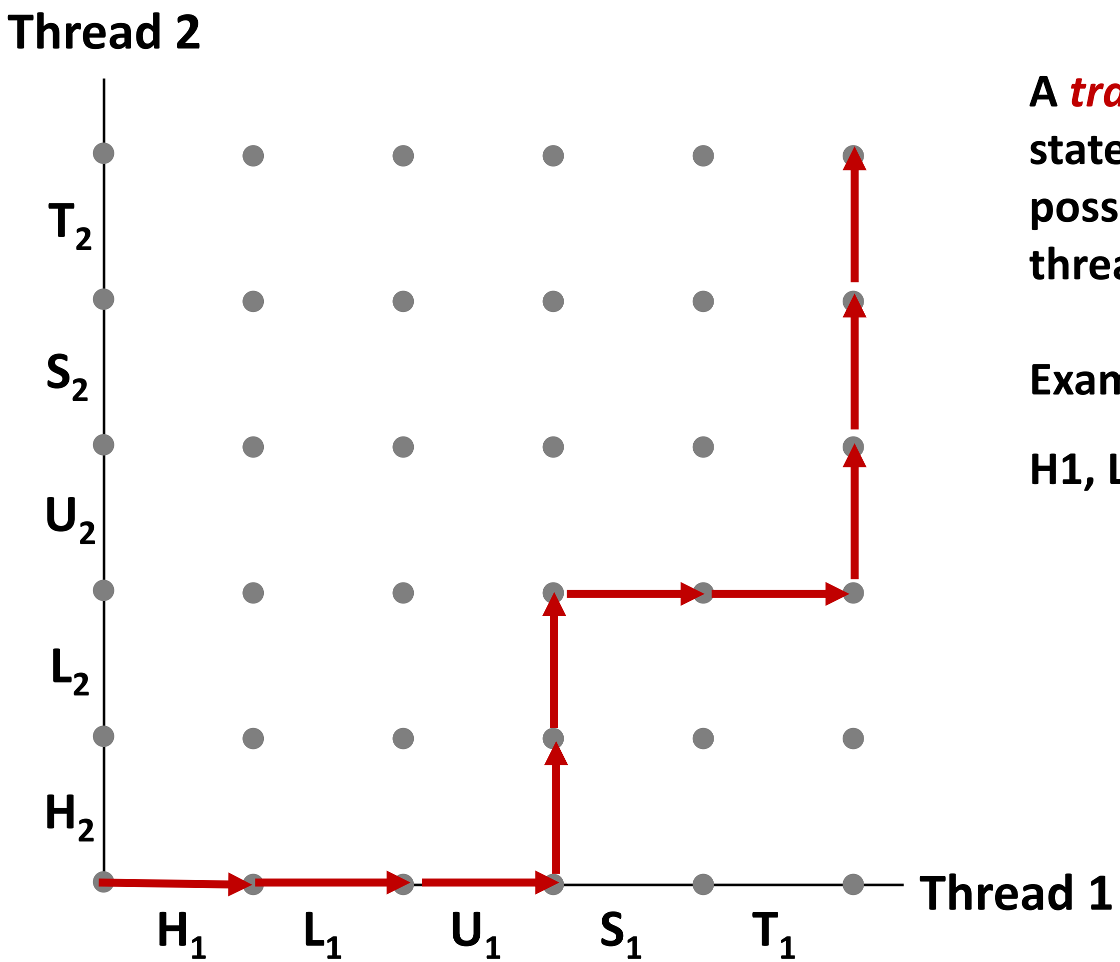
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

Each point corresponds to a possible ***execution state*** ($Inst_1$, $Inst_2$).

E.g., **(L_1, S_2)** denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs

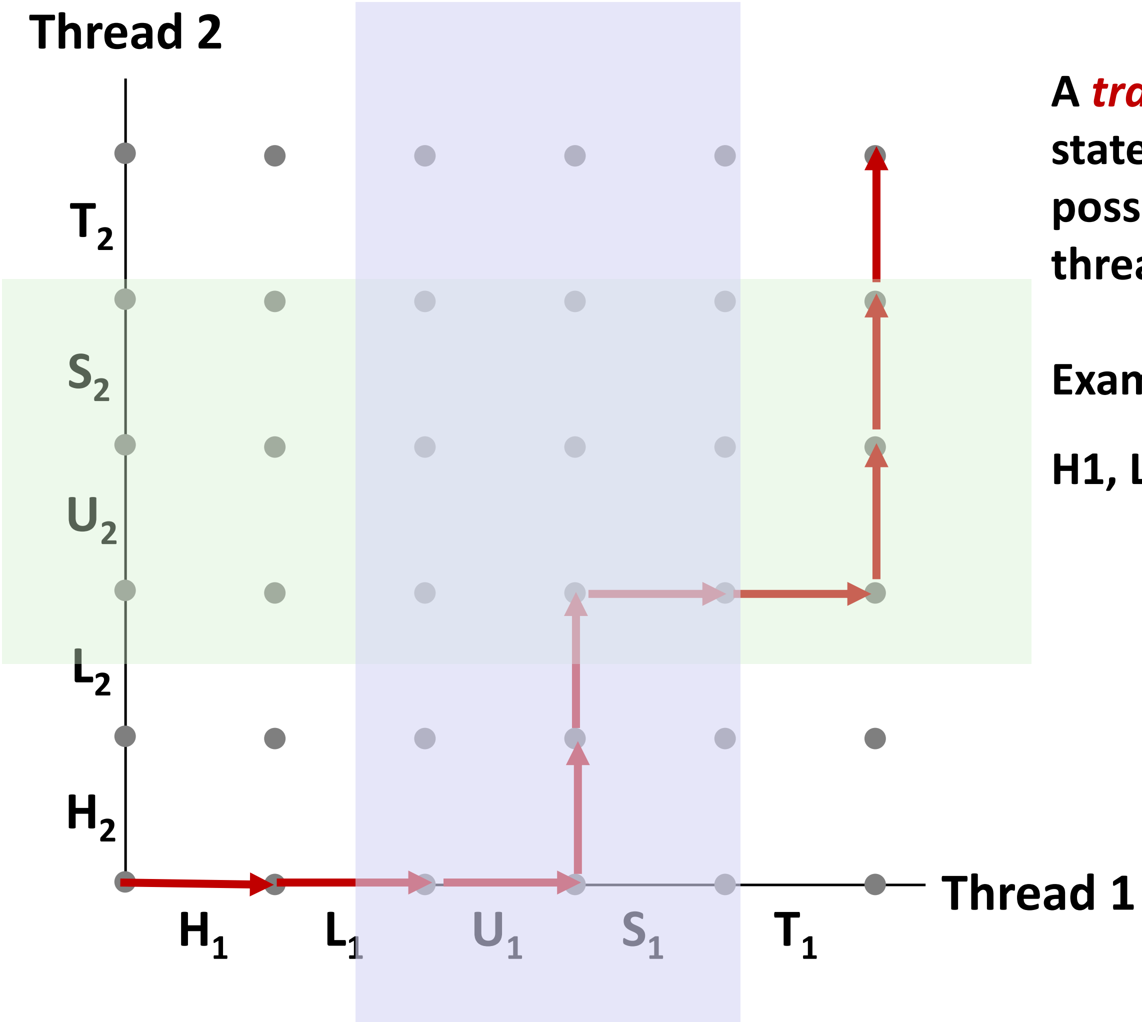


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

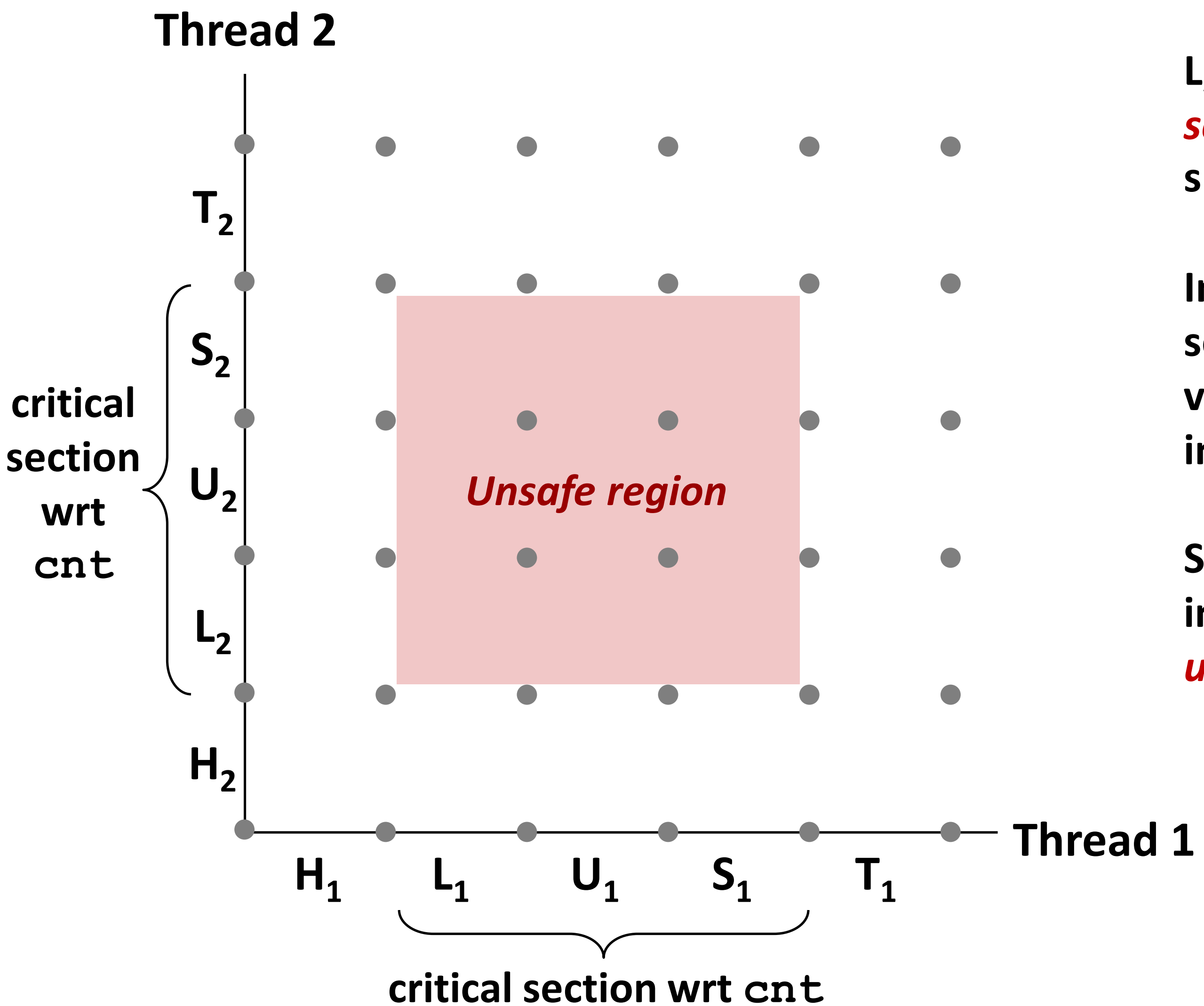


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Critical Sections and Unsafe Regions

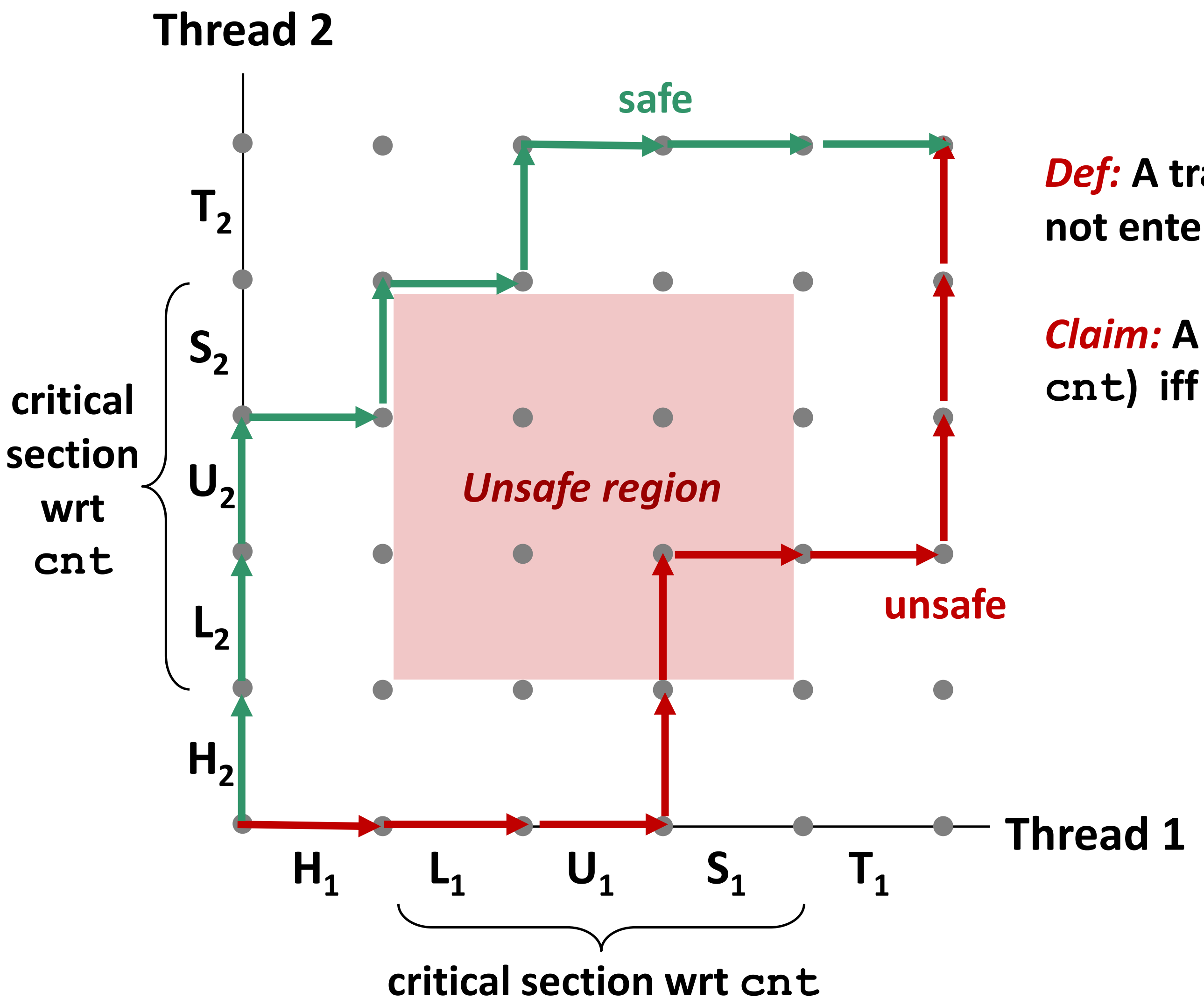


L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

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

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

Critical Sections and Unsafe Regions



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

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

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.

- **Solutions:**
  - Mutex (pthreads)
  - Semaphores (Edsger Dijkstra)

MUTual EXclusion (mutex): Concept

- ***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): Implementation

- ***Mutex***: **boolean synchronization variable** *
- **test_and_set(*p)**

  [old = *p;

  ***ptr = LOCKED;**

  return old;]

  // [] – atomic by the magic of hardware / OS

- **Lock(m):**

  while (test_and_set(&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

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */
long niters = 10000;

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL,
                   thread, NULL);
    pthread_create(&tid2, NULL,
                   thread, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt == (2 * niters))
        printf("OK cnt=%ld\n", cnt);
    else
        printf("BOOM! cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i;
    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

How can we fix this using synchronization?

goodmcnt.c: Mutex Synchronization

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

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

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

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

goodmcnt.c

```
$ ./goodmcnt 10000
OK cnt=20000

$ ./goodmcnt 10000
OK cnt=20000
```

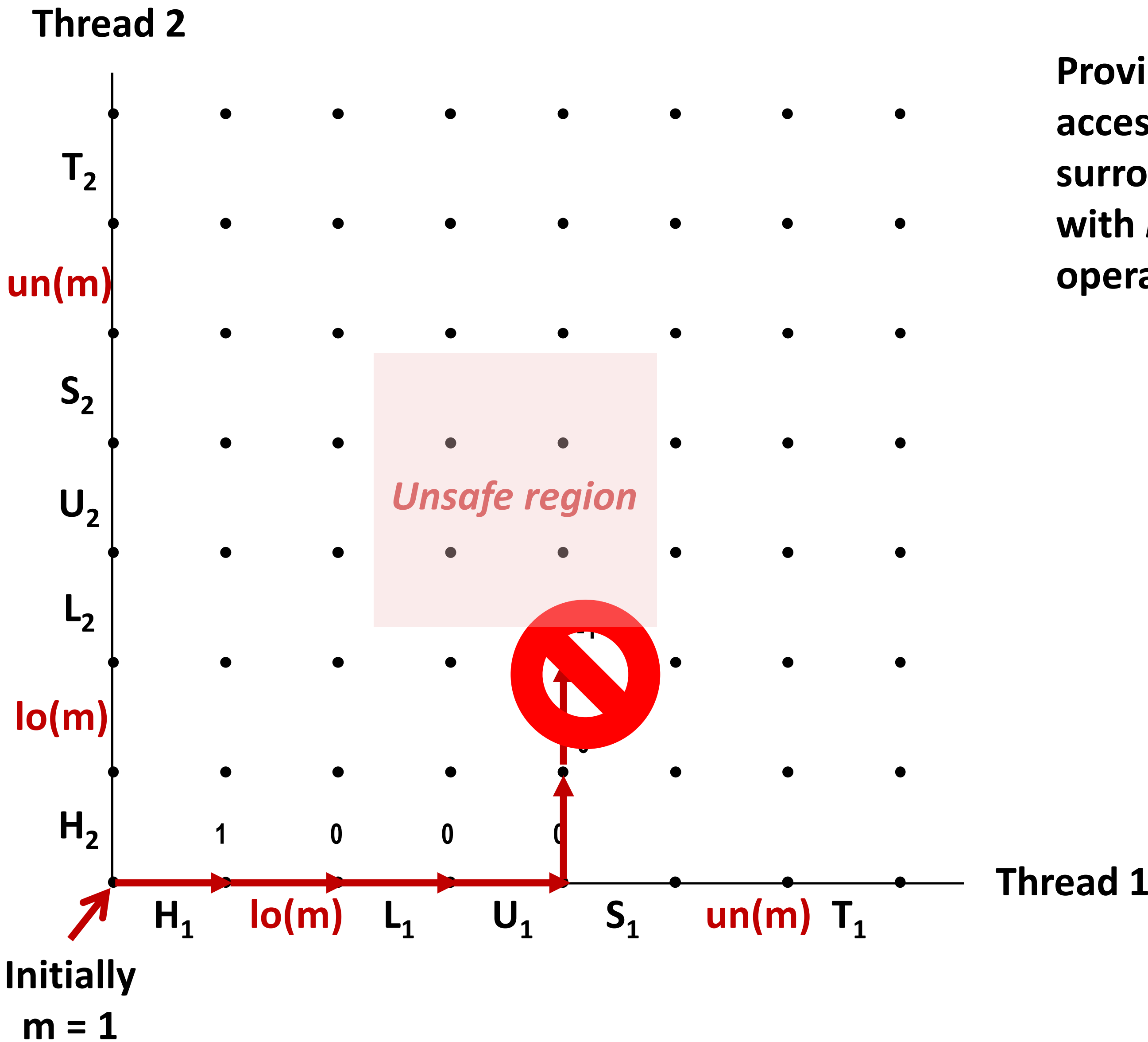
Function	badcnt	goodmcnt
Time (ms) niters = 10^6	12.0	214.0
Slowdown	1.0	17.8

Implementation Details of Mutex

```
#include <stdatomic.h>


#define lock(m) while (atomic_flag_test_and_set(m))
#define unlock(m) atomic_flag_clear(m)


volatile  atomic_flag   mutex;


void *thread(void *vagrp) {
        lock(&mutex);
        // Begin of critical section
        cnt++;
        // End of critical section
        unlock(&mutex);
}
```

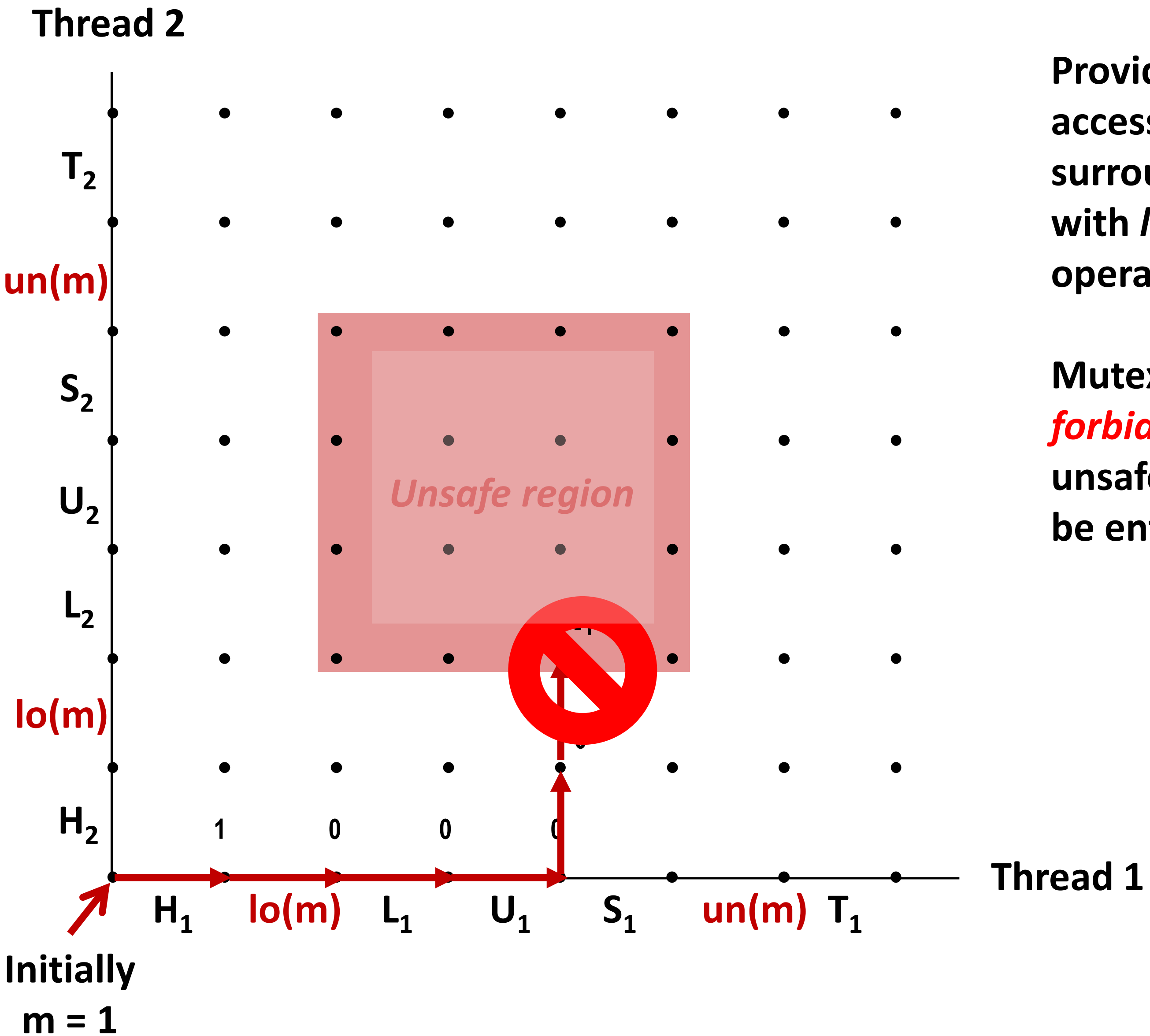
NOTE: The implementation of atomic_*() functions is architecture dependent.

Why Mutexes Work



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

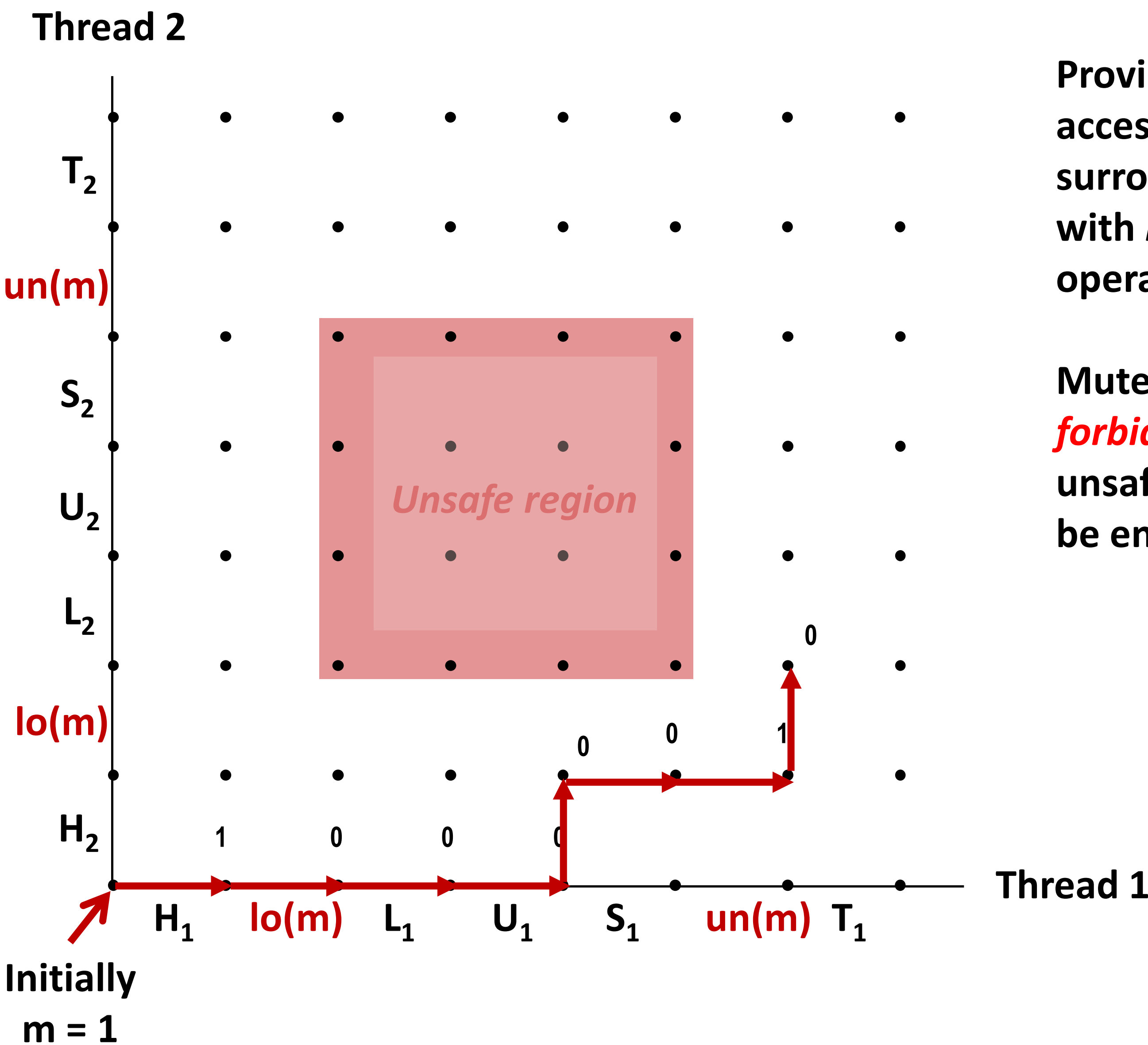
Why Mutexes Work



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.

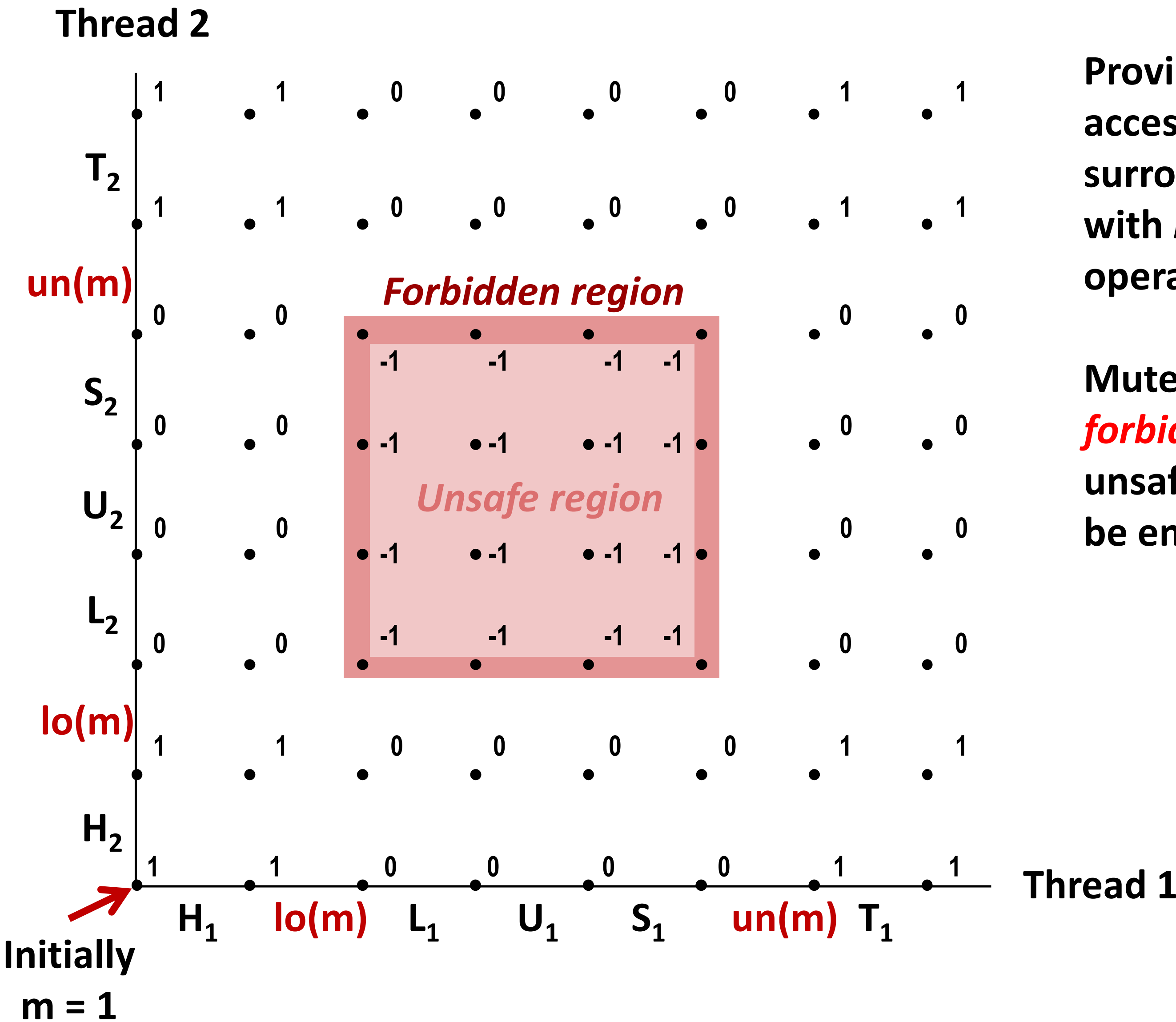
Why Mutexes Work



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.

Why Mutexes Work



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.

Today

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

Semaphores: Concept

- ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *wait* and *post* operations.**
- **wait(s)**
  - So called "P" operation: "Proberen" (test) in Dutch
  - If *s* is nonzero, then **decrement *s* by 1** and return immediately.
    - Test and decrement operations occur atomically (indivisibly)
  - If *s* is zero, then the thread is suspended until *s* becomes non-zero
  - Once s becomes non-zero, decrements *s* and then returns.
- ***post(s)***
  - So called "V" operation: "Verhogen" (increment) in Dutch
  - **Increment *s* by 1**.
    - Increment operation occurs atomically
  - If there are any threads blocked in a wait operation waiting for *s* to become non-zero, then post wakes up exactly one of those threads
- **Semaphore invariant: *(s >= 0)***

Semaphores: Implementation

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

- **Manipulated by *wait* and *post* operations:**
  - **wait(s)**

    [`while (s == 0) wait(); s--;`]
  - **post(s)**

    [`s++;`]
- **Need to guarantee that operations between brackets [ ] are atomic**
  - Only one *wait* or post operation can modify s.
  - When wait breaks out of a `while` loop, only that wait can decrement `s`.
  - **Q. How do you guarantee this?**
- **Semaphore invariant: *(s >= 0)***

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* wait(s) */
int sem_post(sem_t *s);  /* post(s) */
```

Implementation Details of Semaphores (1)

```
typedef volatile struct {
  volatile atomic_int val;
  volatile atomic_flag mut;
} mysem_t;
mysem_t sem;

#define lock(m) while (atomic_flag_test_and_set(m))
#define unlock(m) atomic_flag_clear(m)

int wait(mysem_t * s) {
  lock(&s->mut);
  while (atomic_load(&s->val) <= 0);
  atomic_fetch_sub(&s->val, 1);
  unlock(&s->mut);
  return 0;
} // wait

int post(mysem_t * s) {
  return atomic_fetch_add(&s->val, 1);
} // signal
```

https://gist.github.com/mepcotterell/6f0a779befe388ab822764255e3776ae

Implementation Details of Semaphores (2)

```
#define PING "ping"
#define PONG "pong"

void critical(const char * str) {
  size_t len = strlen(str);
  for (size_t i = 0; i < len; ++i) {
    printf("%c", str[i]);
  } // for
  printf("\n");
} // str

void *thread(void * p) {
  char * msg = (char *) p;
  for (;;) {
    wait(&sem);
    critical(msg);
    post(&sem);
  }
}
```

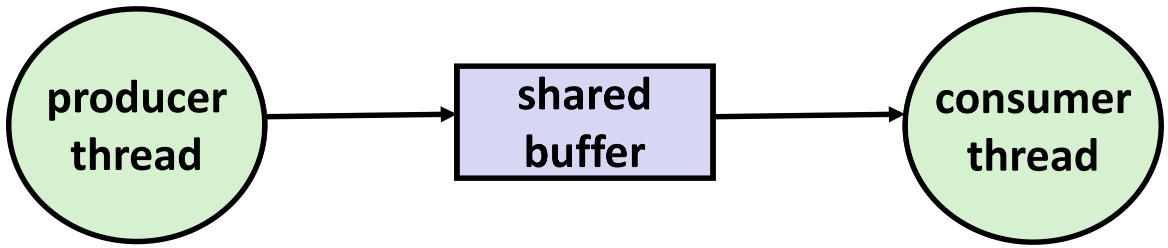
```
int main() {
  setvbuf(stdout, NULL, _IONBF, 0);
  assert(ATOMIC_INT_LOCK_FREE == 2);
  atomic_init(&sem.val, 1);
  pthread_t ping;
  pthread_t pong;
  pthread_create(&ping, NULL, pingpong, PING);
  pthread_create(&pong, NULL, pingpong, PONG);
  for(;;);
  return 0;
} // main
```

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

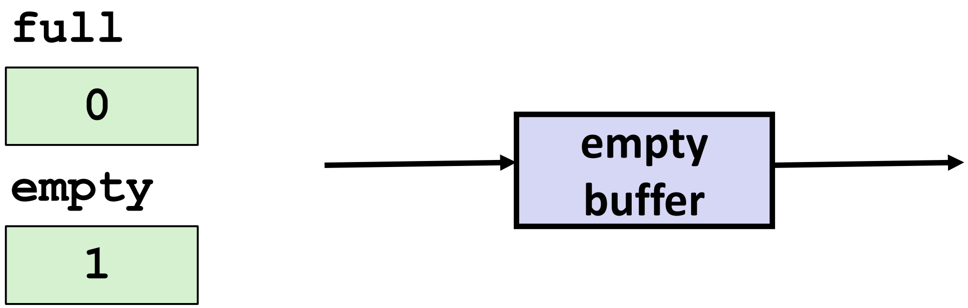


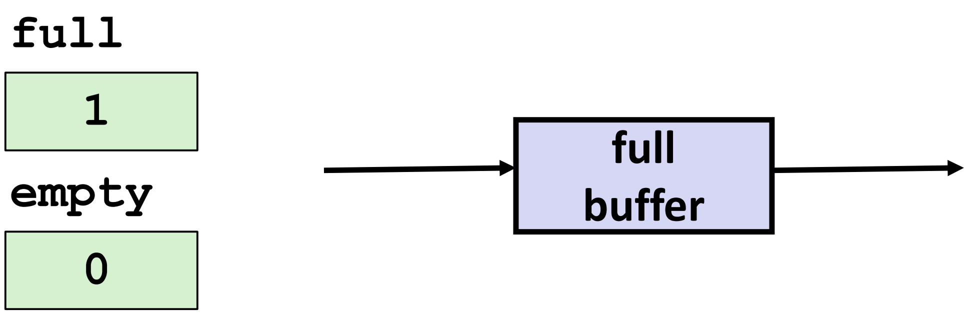
- **Common synchronization pattern:**
  - Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - Consumer waits for ***item***, removes it from buffer, and notifies producer

- **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`**





Producer-Consumer on 1-element Buffers

```
#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 */
  Sem_init(&shared.empty, 0, 1);
  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 */
    wait(&shared.empty);
    shared.buf = item;
    post(&shared.full);
  }
  return NULL;
}
```

Consumer Thread

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

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

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

- **Q. Can you program without any synchronization primitives while preserving the semantics?**

Producer-Consumer without synchronization Primitives

Initially: p = c = 0
p == c means empty
p != c means full

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 */
    while(p != c);
    shared.buf = item;
    p = !p;
}
  return NULL;
}
```

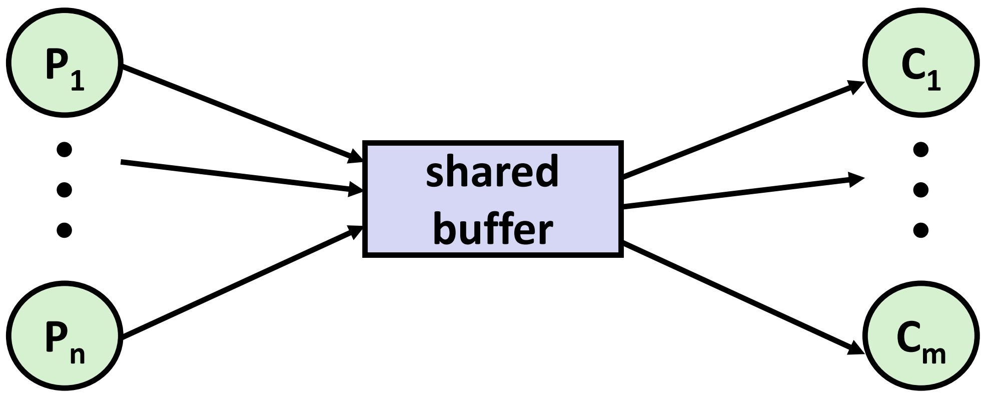
Consumer Thread

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

  for (i=0; i<NITERS; i++) {
    /* Read item from buf */
    while(p == c);
    item = shared.buf;
    c = !c;

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

Data race Vs Race condition

Why 2 Semaphores for 1-Entry Buffer?

- Consider multiple producers & multiple consumers



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

Producers

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

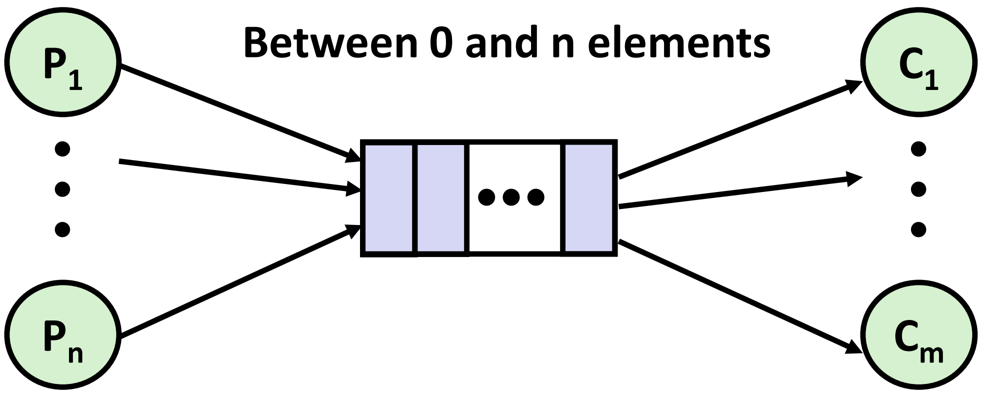
`empty`

`full`

Consumers

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

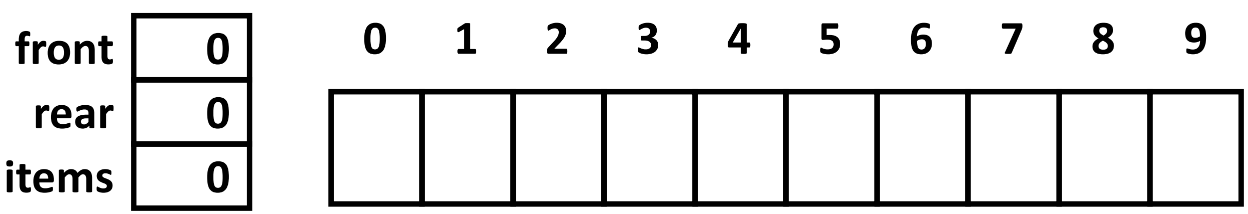
Producer-Consumer on an *n*-element Buffer



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

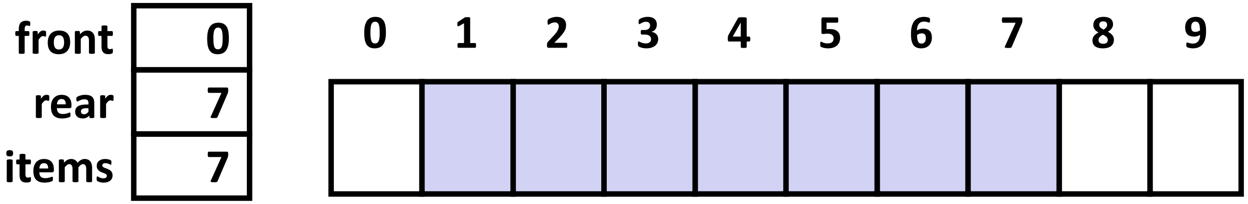
Circular Buffer (n = 10)

- **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:**

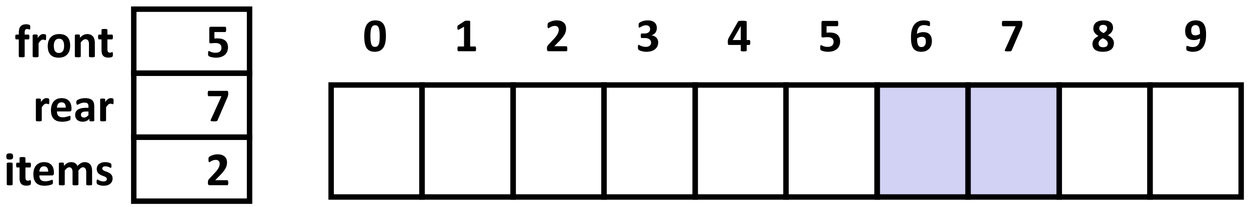


Circular Buffer Operation (n = 10)

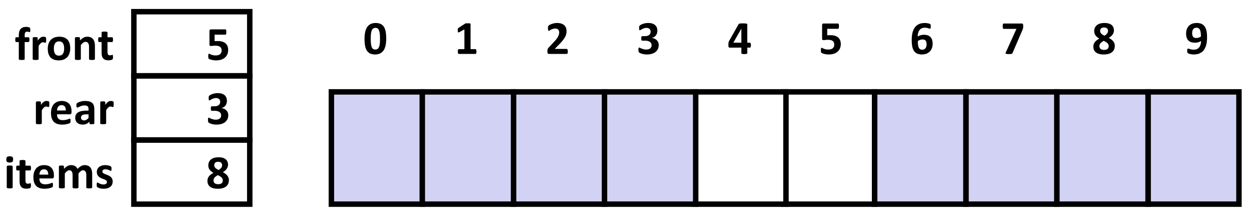
- Insert 7 elements



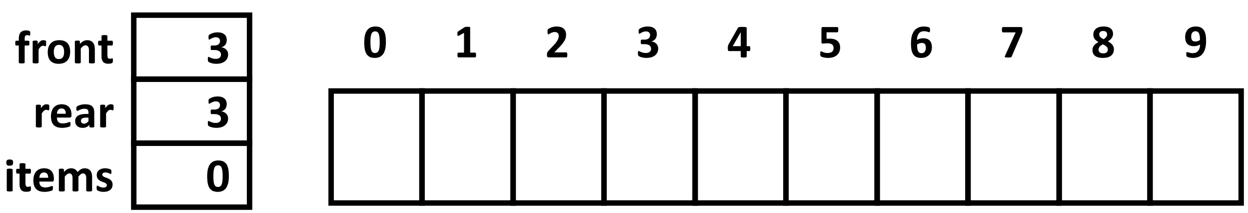
- Remove 5 elements



- Insert 6 elements



- Remove 8 elements



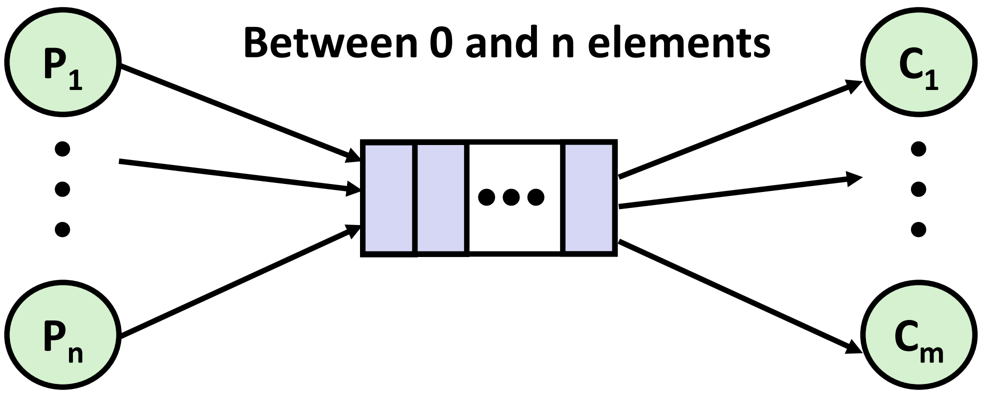
Sequential Circular Buffer Code

```
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
  - **`empty_slots`**: counts the **empty slots** in the buffer
  - **`full_slots`**: counts the **inserted items** in the buffer
- **Makes use of general semaphores**
  - Will range in value from 0 to n

sbuf Package - Declarations

```
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             */
    pthread_mutex_t mutex; /* Protects accesses to buf   */
    sem_t empty_slots;   /* # of empty slots                 */
    sem_t full_slots;  /*  # of inserted 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 */
    pthread_mutex_init(&sp->mutex, NULL); /* lock */
    sem_init(&sp->empty_slots, 0, n); /* Initially n empty slots */
    sem_init(&sp->full_slots, 0, 0); /* Initially 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)
{
    wait(&sp->empty_slots);          /* Wait for empty slot      */
    pthread_mutex_lock(&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          */
    pthread_mutex_unlock(&sp->mutex); /* Unlock the buffer   */
    post(&sp->full_slots);           /* Notify item is inserted */
}
```

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;
    wait(&sp->empty_slots);       /* Wait for available item */
    pthread_mutex_lock(&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         */
    pthread_mutex_unlock(&sp->mutex); /* Unlock the buffer   */
    post(&sp->full_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.

- Semaphores are a fundamental mechanism for enforcing mutual exclusion.