哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

操作系统

Operating Systems

刘川意　副教授

哈尔滨工业大学（深圳）

2019年10月

Module 3: 同步（ Synchronization）

1. 信号的介绍 （Introduction to Signals）
   CSAPP3e: 8.5

2. 同步（Synchronization）

   CSAPP3e: 12.4-12.7

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*"

- *Definition:* 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 model（概念模型）:
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments of the process virtual address space
    - Open files and installed handlers
- Operationally, this model is not strictly enforced:
  - Register（寄存器）values are truly separate and protected, but virtual memory is always shared
  - Any thread can read and write the stack of any other thread

The mismatch between the conceptual（概念）and operation（操作）model is a source of confusion and errors

Example Program to Illustrate Sharing

```c
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

```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

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`)

```c
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]
)

```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;
}
```

Local static var: 1 instance (`cnt` [data])

Shared Variable Analysis

- Which variables are shared?

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

- Answer: A variable `x` is shared iff multiple threads reference at least one instance of `x`. Thus:
  - **`ptr`, `cnt`, and `msgs` are shared**
  - **`i` and `myid` are *not* shared**

badcnt.c: Improper Synchronization

```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

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

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

    return NULL;
}
```

```
[zs_cao@localhost conc]$ ./badcnt 10000
OK cnt=20000
[zs_cao@localhost conc]$ ./badcnt 10000
BOOM! cnt=17302
[zs_cao@localhost conc]$ ./badcnt 10000
OK cnt=20000
```

线程并发执行的问题

Assembly Code for Counter Loop

- 编译:来自CSAPP第7章链接
  - gcc –s badcnt.c –o badcnt.s
  - vim badcnt.s

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

```
        movq    %rdi, -24(%rbp)
        movq    -24(%rbp), %rax
        movq    (%rax), %rax
        movq    %rax, -8(%rbp)
        movq    $0, -16(%rbp)
        jmp     .L6
.L7:
        movq    cnt(%rip), %rax
        addq    $1, %rax
        movq    %rax, cnt(%rip)
        addq    $1, -16(%rbp)
.L6:
        movq    -16(%rbp), %rax
        cmpq    -8(%rbp), %rax
        jl      .L7
        movl    $0, %eax
        popq    %rbp
```

H_i : Head

L_i : Load `cnt`

U_i : Update `cnt`

S_i : Store `cnt`

T_i : Tail

Assembly Code for Counter Loop

- 汇编：来自CSAPP第7章 链接
  - gcc –c badcnt.s –o badcnt.o
  - objdump -dx badcnt.o

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

```
00000000000000ed <thread>:
  ed:   55                      push   %rbp
  ee:   48 89 e5                mov    %rsp,%rbp
  f1:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
  f5:   48 8b 45 e8             mov    -0x18(%rbp),%rax
  f9:   48 8b 00                mov    (%rax),%rax
  fc:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 100:   48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
 107:   00
 108:   eb 17                   jmp    121 <thread+0x34>
 10a:   48 8b 05 00 00 00 00    mov    0x0(%rip),%rax        # 111
<thread+0x24>
                        10d: R_X86_64_PC32      cnt-0x4
 111:   48 83 c0 01             add    $0x1,%rax
 115:   48 89 05 00 00 00 00    mov    %rax,0x0(%rip)        # 11c
<thread+0x2f>
                        118: R_X86_64_PC32      cnt-0x4
 11c:   48 83 45 f0 01          addq   $0x1,-0x10(%rbp)
 121:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 125:   48 3b 45 f8             cmp    -0x8(%rbp),%rax
 129:   7c df                   jl     10a <thread+0x1d>
 12b:   b8 00 00 00 00          mov    $0x0,%eax
 130:   5d                      pop    %rbp
 131:   c3                      retq
```

H_i L_i U_i S_i T_i

Assembly Code for Counter Loop

- 链接:来自CSAPP第7章 链接
  - gcc –o badcnt.c –o badcnt -lpthread
  - objdump -d badcnt

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

怎么计算？

```
0000000000000957 <thread>:
 957:   55                      push   %rbp
 958:   48 89 e5                mov    %rsp,%rbp
 95b:   48 89 7d e8             mov    %rdi,-0x18(%rbp)
 95f:   48 8b 45 e8             mov    -0x18(%rbp),%rax
 963:   48 8b 00                mov    (%rax),%rax
 966:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 96a:   48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
 971:   00
 972:   eb 17                   jmp    98b <thread+0x34>
 974:   48 8b 05 b5 06 20 00    mov    0x2006b5(%rip),%rax
  # 201030 <cnt>
 97b:   48 83 c0 01             add    $0x1,%rax
 97f:   48 89 05 aa 06 20 00    mov    %rax,0x2006aa(%rip)
  # 201030 <cnt>
 986:   48 83 45 f0 01          addq   $0x1,-0x10(%rbp)
 98b:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 98f:   48 3b 45 f8             cmp    -0x8(%rbp),%rax
 993:   7c df                   jl     974 <thread+0x1d>
 995:   b8 00 00 00 00          mov    $0x0,%eax
 99a:   5d                      pop    %rbp
 99b:   c3                      retq
```

H_i

L_i

U_i

S_i

T_i

Concurrent Execution(并发执行)

- *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

OK

Thread 1 critical section

Thread 2 critical section

Concurrent Execution (cont)

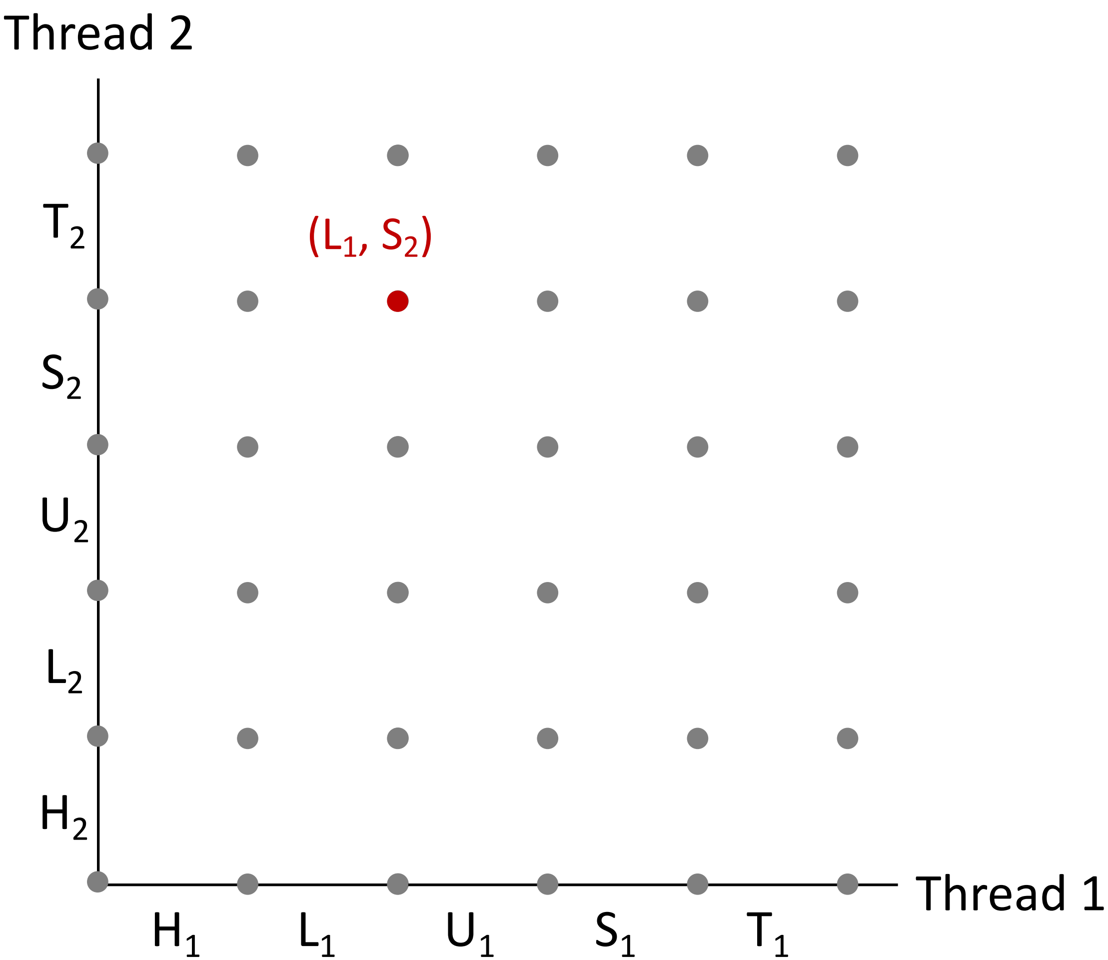
- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	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

S1应该在L2之前执行

Oops!

Progress Graphs（进度图）



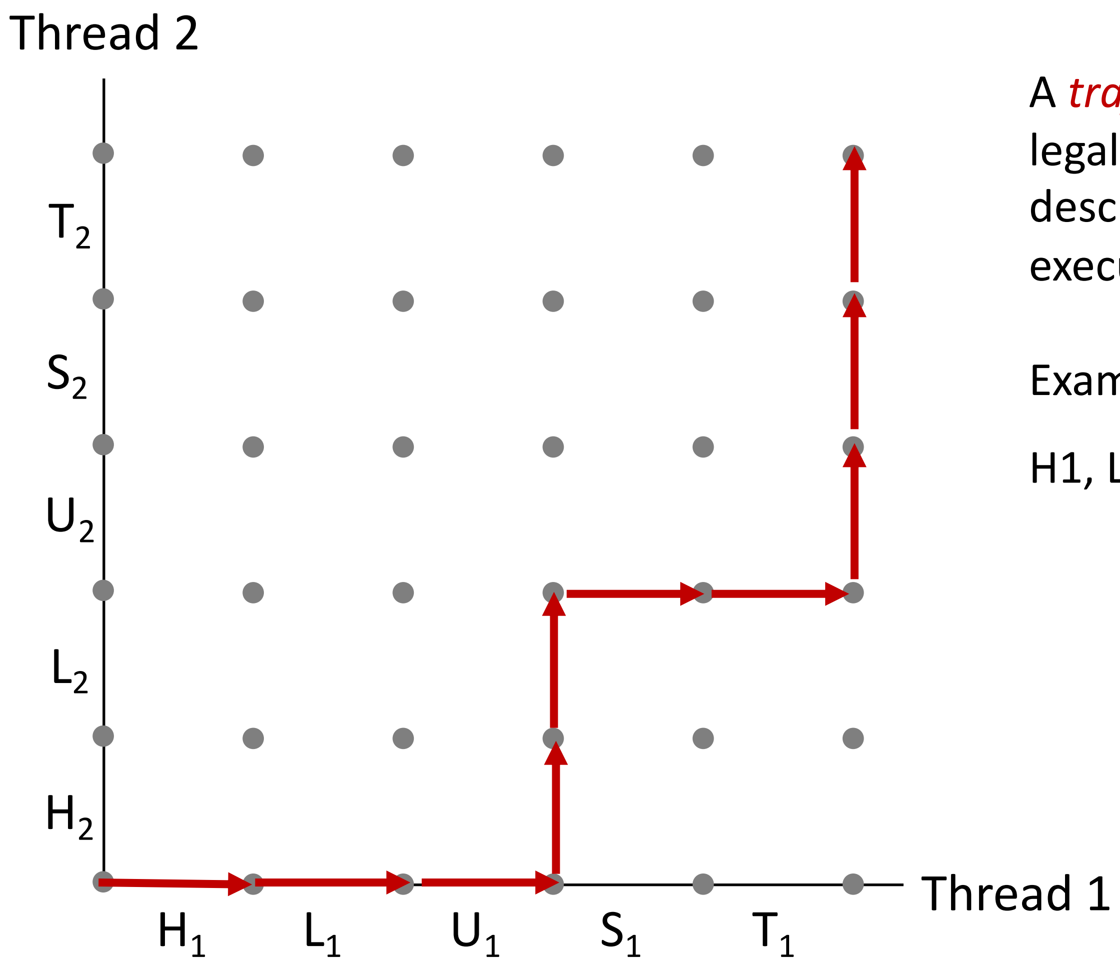
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

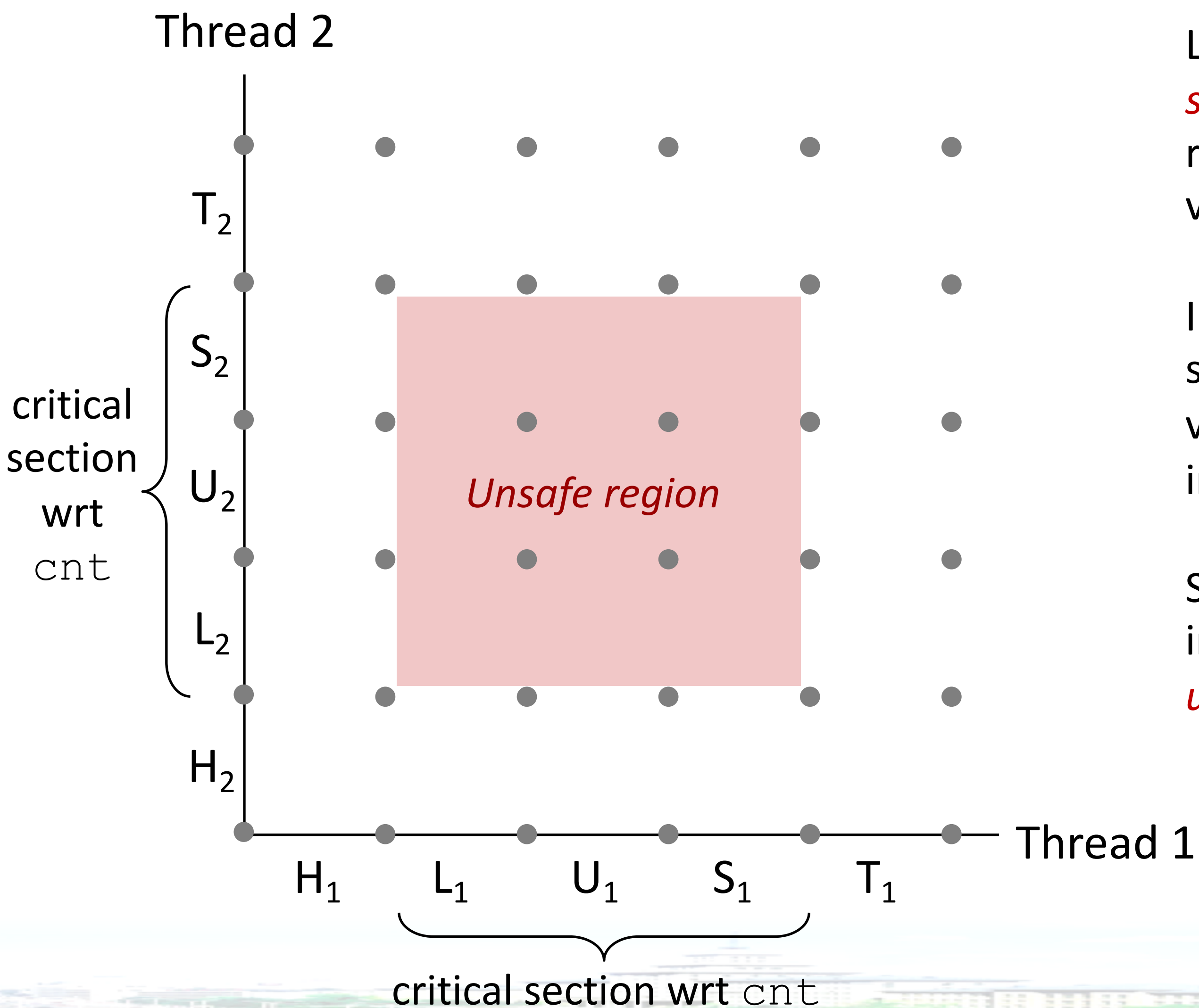


A *trajectory（轨道）* is a sequence of legal state transitions（转换） that describes one possible concurrent execution of the threads.

Example:

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

Critical Sections and Unsafe Regions

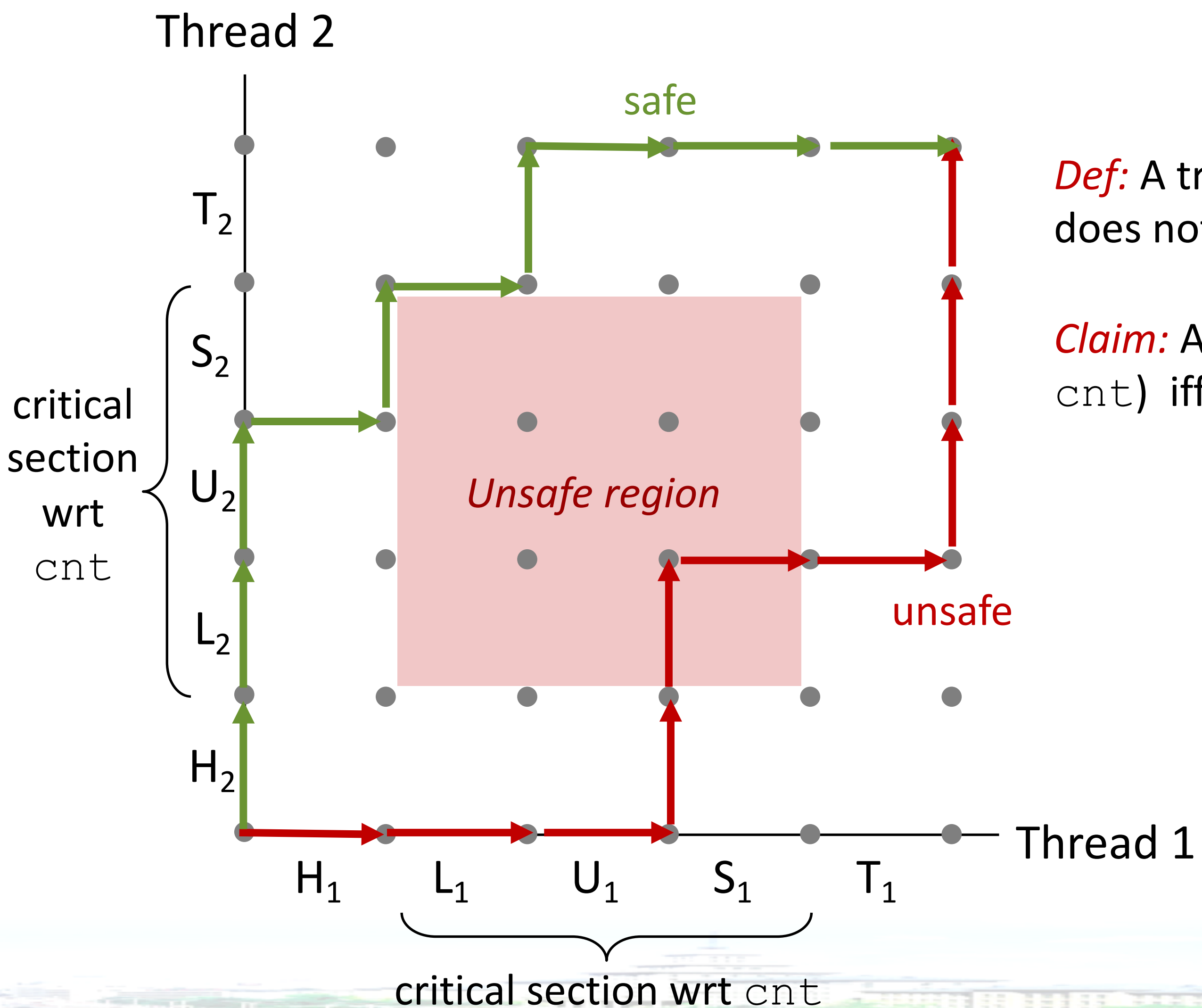


L, U, and S form a *critical section（临界区）*with respect to the shared variable `cnt`

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（轨道）is *safe* iff it does not enter any unsafe region

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

Enforcing Mutual Exclusion

- *Question:* How can we guarantee a safe trajectory?

- Answer: We must ***synchronize***（*同步*）the execution of the threads so that they can never have an unsafe trajectory.
  - i.e., need to guarantee ***mutually exclusive access（互斥地访问）*** for each critical section.

- Classic solution:
  - Semaphores（信号量）(Edsger Dijkstra)

- Other approaches (out of our scope)
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Semaphores（信号量）

- ***Semaphore:*** non-negative global integer synchronization variable. Manipulated by *P* and *V* operations. Semaphore invariant: *(s >= 0)*
- 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*.

```
P(s):
     if s>0 s--;
     if s==0 do block;
V(s):
     s++;
     if any threads blocked
  by P operation then wakeup
```

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, int pshared, unsigned int val);}
 /* s为指向信号量结构的一个指针；
    pshared不为0时此信号量在进程间共享，否则只能为当前进程的所有线程共享；
    val给出了信号量的初始值。
    成功返回0，错误返回-1
*/
int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS:APP wrapper functions:

```
void P(sem_t *sem)
{
  if (sem_wait(sem) < 0)
    unix_error("P error");
}
```

goodcnt.c

```
void V(sem_t *sem)
{
  if (sem_post(sem) < 0)
    unix_error("V error");
}
```

goodcnt.c

badcnt.c: Improper Synchronization

```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

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

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

    return NULL;
}
```

How can we fix this using semaphores?

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
  - Surround corresponding critical sections (临界区)with *P(mutex)* and *V(mutex)* operations.

- Terminology:
  - *Binary semaphore（二元信号量）*: semaphore whose value is always 0 or 1
  - *Mutex（互斥）:* 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（计数信号量）*: 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*:

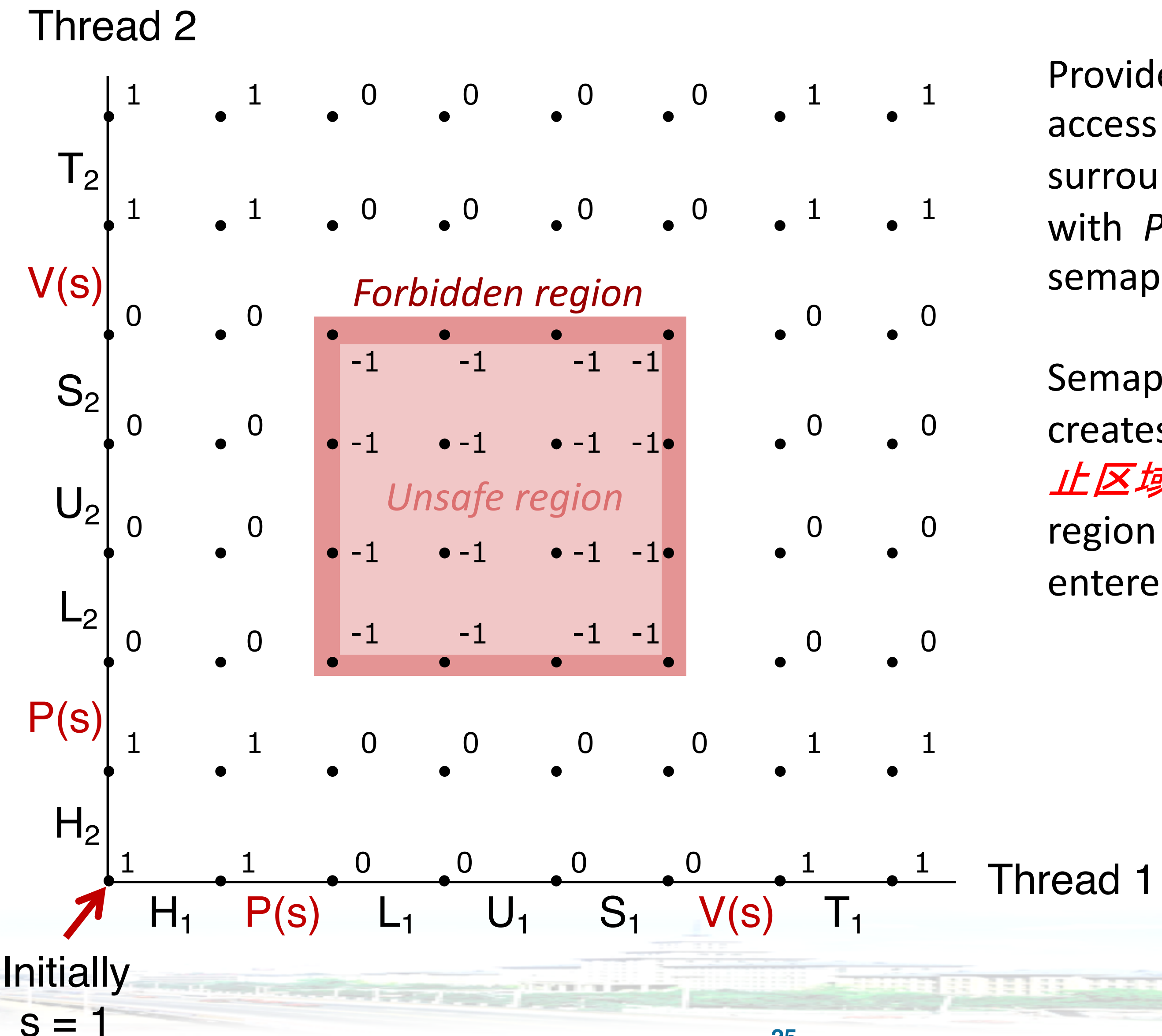
```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

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

Warning: It's 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（禁止区域）*that encloses unsafe region and that cannot be entered by any trajectory.

Summary

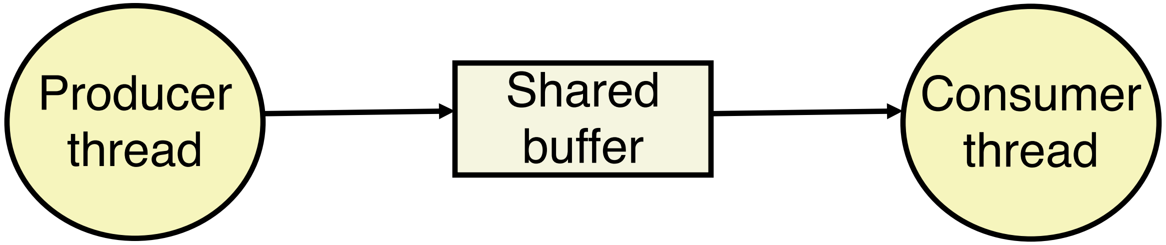
- Programmers need a clear model of how variables are shared by threads.
- Variables shared by multiple threads must be protected to ensure mutually exclusive access.
- Semaphores are a fundamental mechanism for enforcing mutual exclusion.
- 使用信号量的函数执行时间较未使用信号量的时间要长
  - 以goodcnt和badcnt为例，参数为100000时，goodcnt运行时长为badcnt时长的17倍

```
[zs_cao@localhost conc]$ time ./badcnt 1000000
OK cnt=2000000

real    0m0.013s
user    0m0.004s
sys     0m0.008s
[zs_cao@localhost conc]$ time ./goodcnt 1000000
OK cnt=2000000

real    0m0.225s
user    0m0.187s
sys     0m0.145s
[zs_cao@localhost conc]$
```

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 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 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 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`.

sbuf Package - Declarations

```
#include "csapp.h"

typedef struct {
    int *buf;          /* Buffer array */
    int n;             /* Maximum number of slots */
    int front;         /* buf[(front+1)%n] is first item */
    int rear;          /* buf[rear%n] 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);     /* 初始化信号量 */
void sbuf_deinit(sbuf_t *sp);          /* 释放buf */
void sbuf_insert(sbuf_t *sp, int item);  /*生产者线程*/
int sbuf_remove(sbuf_t *sp);          /* 消费者线程 */
```

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

sbuf.c