第12章 并发编程

同步：基础 Synchronization: Basics

100076202： 计算机系统导论

任课教师：

宿红毅 张艳 黎有琦 颜珂

原作者：

Randal E. **Bryant and** David R. O’Hallaron

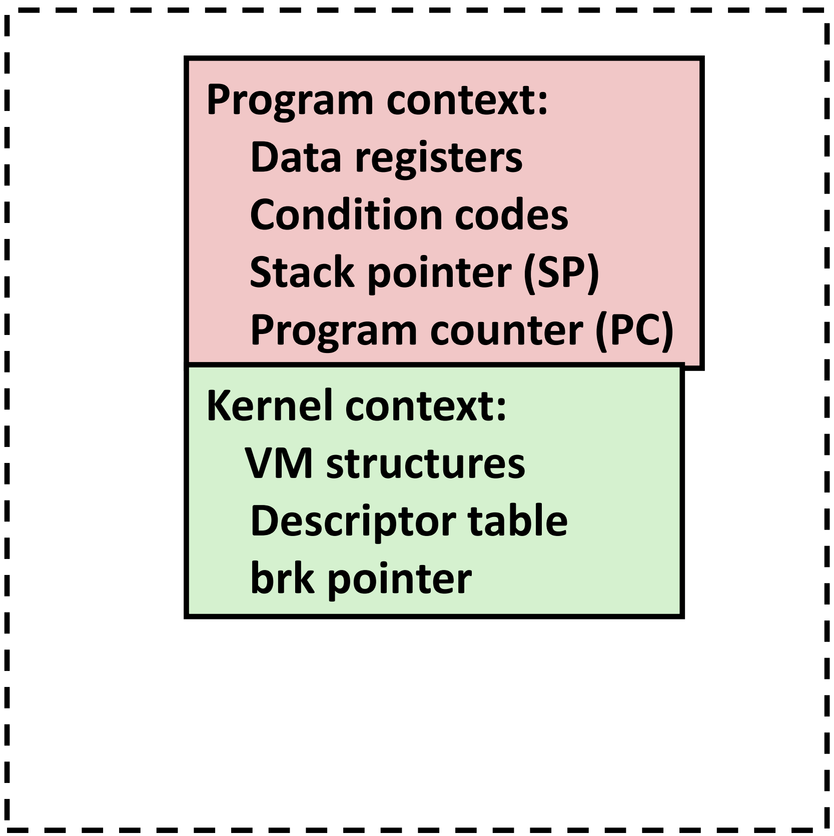
议题 Today

- **线程回顾 Threads review**
- 共享 Sharing
- 互斥 Mutual exclusion
- 信号量 Semaphores

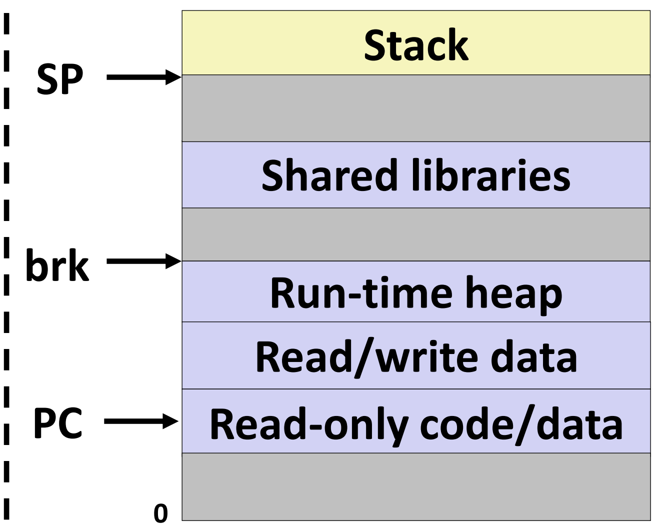
传统进程的视图 Traditional View of a Process

- 进程=进程上下文+代码、数据和栈 Process = process context + code, data, and stack

进程上下文
Process context

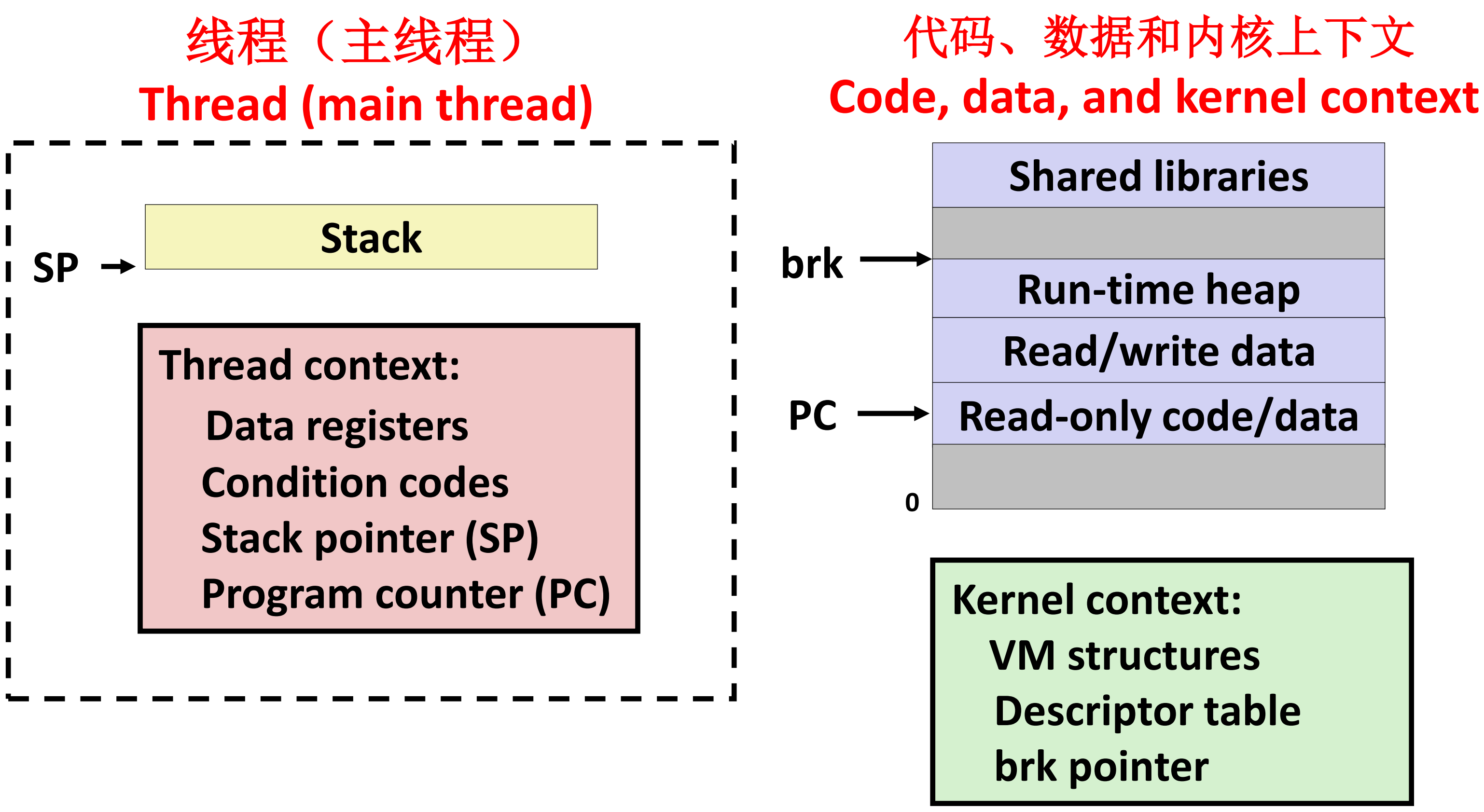


代码、数据和栈
Code, data, and stack



进程的替代视图 Alternate View of a Process

- 进程=线程+（代码、数据和内核上下文） **Process = thread + (code, data, and kernel context)**



一个进程有多个线程-多线程进程
A Process With Multiple Threads

- 多个线程可以与一个进程关联 **Multiple threads can be associated with a process**
  - 每个线程都有自己的逻辑控制流 Each thread has its own logical control flow
  - 每个线程共享相同的代码、数据和内核上下文 Each thread shares the same code, data, and kernel context
  - 每个线程都有自己的局部变量栈 Each thread has its own stack for local variables
    - 但不受其他线程的保护 but not protected from other threads
  - 每个线程都有自己的线程id（TID） Each thread has its own thread id (TID)

线程1（主线程）
Thread 1 (main thread)

stack 1

Thread 1 context:
Data registers
Condition codes
SP_1
PC_1

线程2（对等线程）
Thread 2 (peer thread)

stack 2

Thread 2 context:
Data registers
Condition codes
SP_2
PC_2

共享代码和数据
Shared code and data

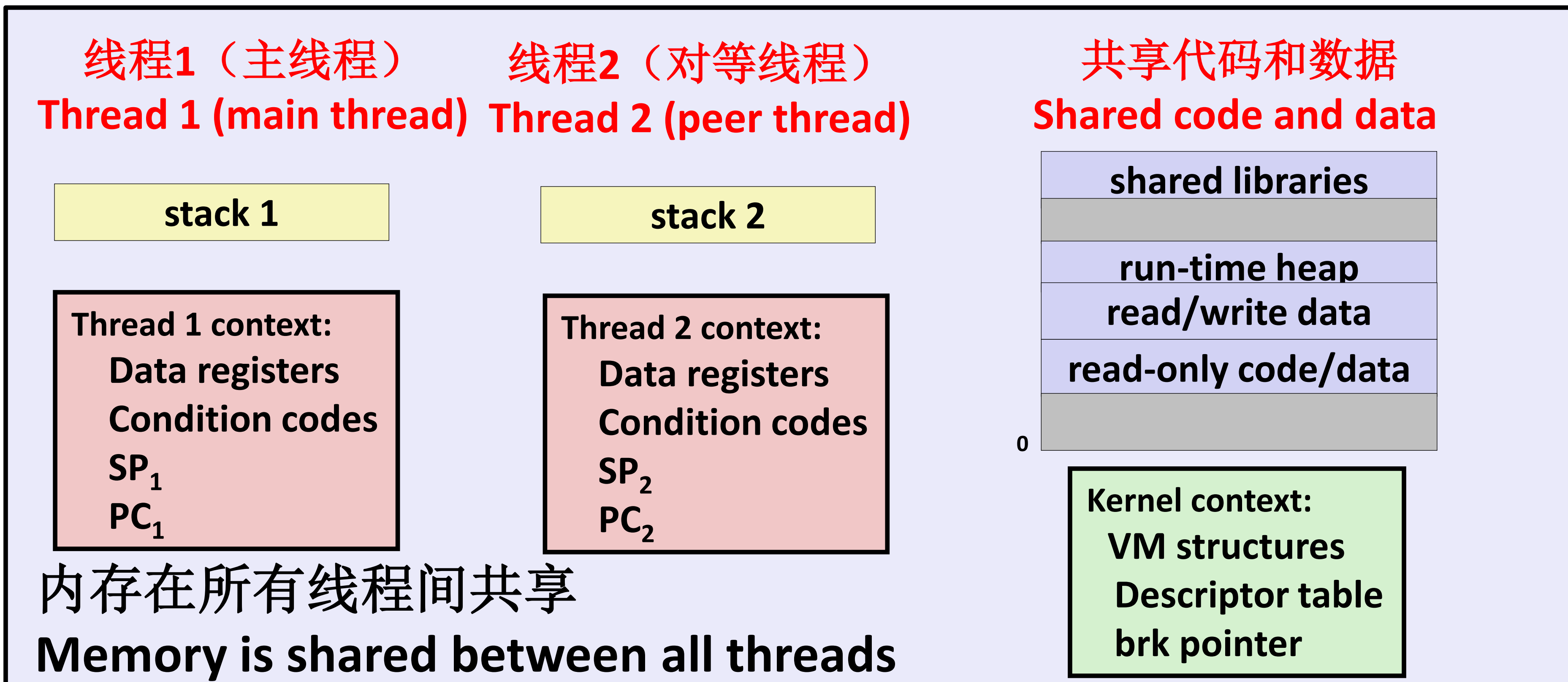
shared libraries

run-time heap
read/write data
read-only code/data
0

Kernel context:
VM structures
Descriptor table
brk pointer

不要让图片迷惑你！

Don’t let picture confuse you!



议题 Today

- 线程回顾 Threads review
- **共享 Sharing**
- 互斥 Mutual exclusion
- 信号量 Semaphores
- 生产者-消费者同步 Producer-Consumer Synchronization

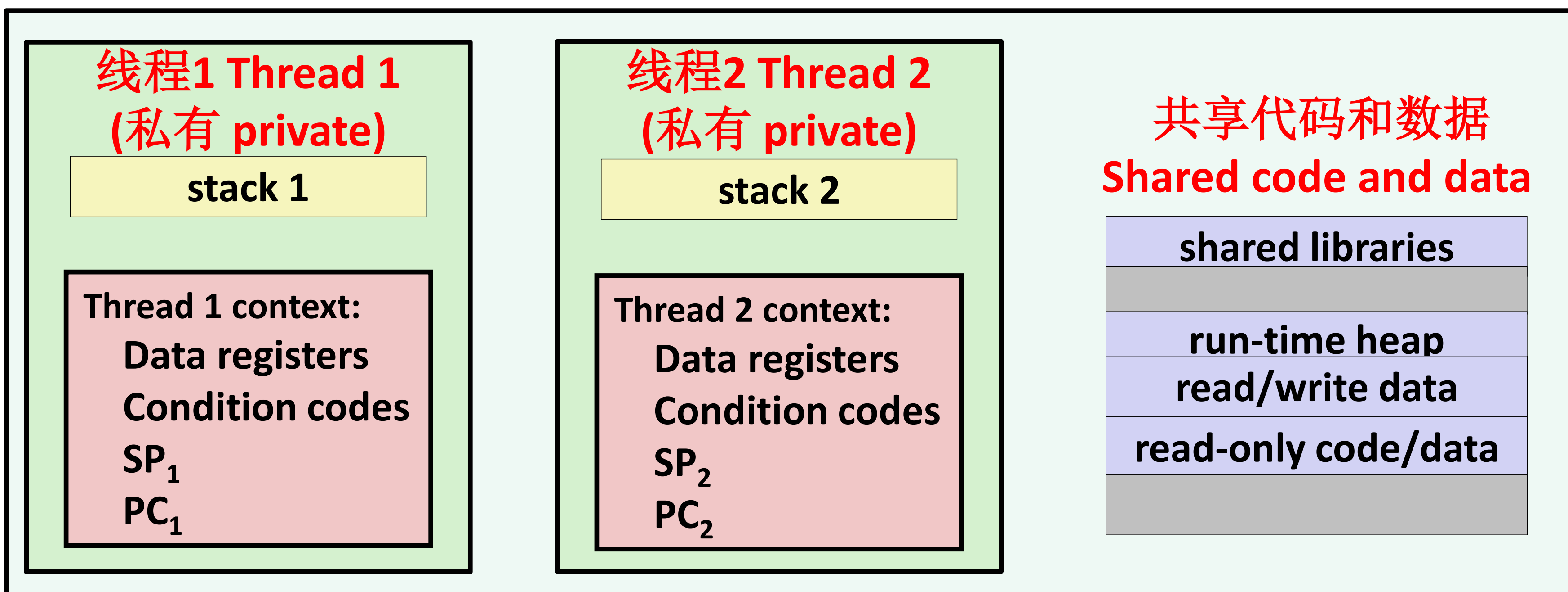
在线程化的C语言程序中共享变量 Shared Variables in Threaded C Programs

- 问题：线程化C程序中的哪些变量是共享的？ **Question: Which variables in a threaded C program are shared?**
  - 答案并不像“全局变量是共享的”和“栈变量是私有的”那么简单 The answer is not as simple as “*global variables are shared*” and “*stack variables are private*”
- ***定义：***当且仅当多个线程引用x的某个实例时，变量x是共享的 ***Def:*** **A variable x is *shared* if and only if multiple threads reference some instance of x.**

- 需要以下问题的答案： **Requires answers to the following questions:**
  - 线程的内存模型是什么？ What is the memory model for threads?
  - 变量实例如何映射到内存？ How are instances of variables mapped to memory?
  - 有多少个线程可以引用每个实例？ How many threads might reference each of these instances?

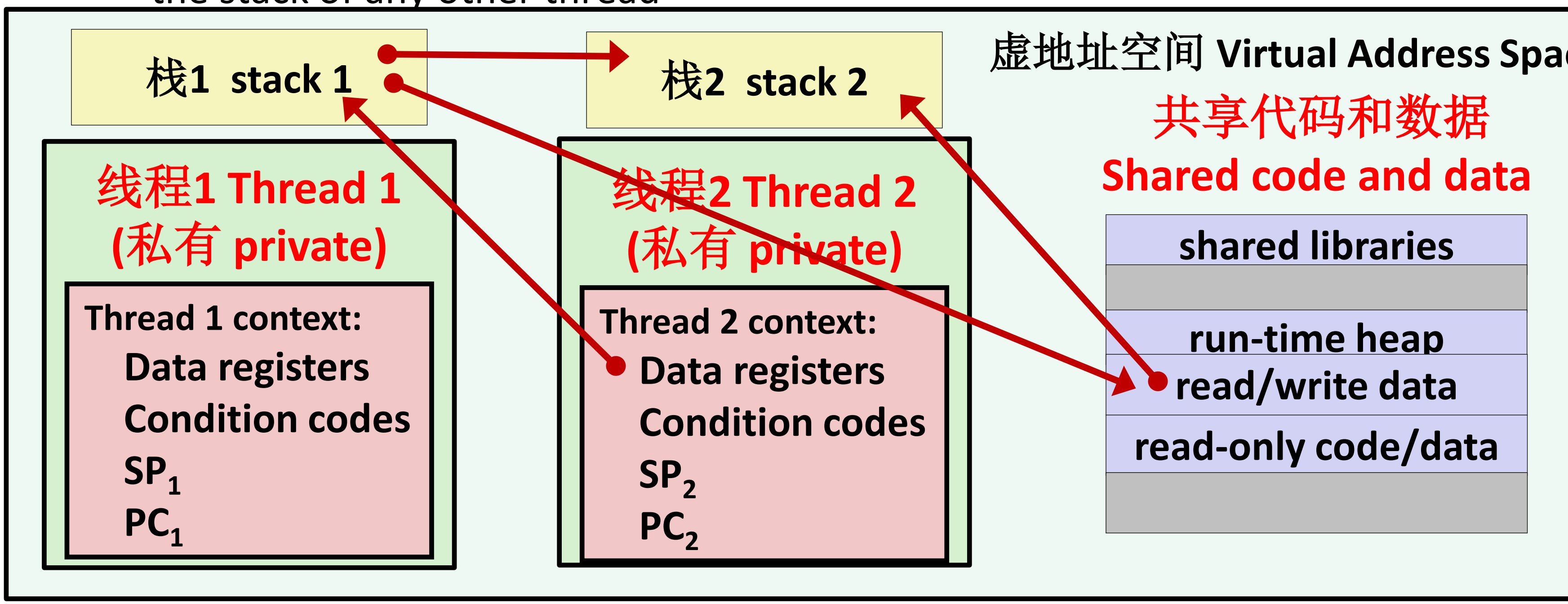
线程内存模型：概念上
Threads Memory Model: Conceptual

- 多个线程在单个进程的上下文中运行 **Multiple threads run within the context of a single process**
- 每个线程都有自己独立的线程上下文 **Each thread has its own separate thread context**
  - 线程ID、栈、栈指针、PC、条件码和GP寄存器 Thread ID, stack, stack pointer, PC, condition codes, and GP registers
- 所有线程共享剩余的进程上下文 **All threads share the remaining process context**
  - 进程虚拟地址空间的代码、数据、堆和共享库段 Code, data, heap, and shared library segments of the process virtual address space
  - 打开文件和安装的信号处理程序 Open files and installed handlers



线程内存模型：实际上
Threads Memory Model: Actual

- 未严格执行数据分离： **Separation of data is not strictly enforced:**
  - 寄存器值是真正独立和受保护的，但是… Register values are truly separate and protected, but…
  - 任何线程都可以读取和写入任何其他线程的栈 Any thread can read and write the stack of any other thread



概念模型和操作模型之间的不匹配是混淆和错误的根源

The mismatch between the conceptual and operation model is a source of confusion and errors

向线程传递参数 – 学究式方法
Passing an argument to a thread - Pedantic

```
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;
}
```

```
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");
}
```

向线程传递参数 – 学究式方法
Passing an argument to a thread - Pedantic

```
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;
}
```

- 使用malloc为每个线程分配堆内存存放参数 Use malloc to create a per thread heap allocated place in memory for the argument
- 记得在线程中释放内存！Remember to free in thread!
- 生产者-消费者模式 Producer-consumer pattern

向线程传递参数 – 另一种方法！
Passing an argument to a thread – Also OK!

```
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;
}
```

- 使用强制转换也可以，因为长整数大小小于等于无类型指针的大小 **Ok to Use cast since sizeof(long) <= sizeof(void*)**
- 强制转换不会改变位模式 **Cast does NOT change bits**

向线程传递参数 – 警告！
Passing an argument to a thread – WRONG!

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

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

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

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

- 取i的地址对所有的线程来说都指向同样的位置 `&i` points to same location for all threads!
- 产生数据竞争！ Creates a data race!

传递线程参数的三种方法
Three Ways to Pass Thread Arg

- 申请/释放空间 **Malloc/free**
  - 生产者申请空间，传递指针给pthread_create Producer malloc's space, passes pointer to pthread_create
  - 消费者释放指针空间 Consumer dereferences pointer
- 指向栈槽位 **Ptr to stack slot**
  - 生产者在pthread_create中传递生产者栈地址 Producer passes address to producer's stack in pthread_create
  - 消费者释放指针 Consumer dereferences pointer
- 强制转换成整数 **Cast of int**
  - 在pthread_create中生产者强制转换整数/长整数为地址 Producer casts an int/long to address in pthread_create
  - 消费者强制转换无类型指针参数回整数/长整数 Consumer casts void* argument back to int/long

示例程序说明共享
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;
}
```

对等线程间接通过全局ptr变量引用主线程的栈

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

一种通用方法传递单个参数给一个线程例程 ***A common way to pass a single argument to a thread routine***

在线程化的C语言程序中共享变量
Shared Variables in Threaded C Programs

- **问题：线程化C程序中的哪些变量是共享的？ Question: Which variables in a threaded C program are shared?**
  - 答案并不像“全局变量是共享的”和“栈变量是私有的”那么简单 The answer is not as simple as “*global variables are shared*” and “*stack variables are private*”
- ***定义：*当且仅当多个线程引用x的某个实例时，变量x是共享的 *Def:* A variable x is *shared* if and only if multiple threads reference some instance of x.**

- **需要以下问题的答案： Requires answers to the following questions:**
  - 线程的内存模型是什么？ What is the memory model for threads?
  - 变量实例如何映射到内存？ How are instances of variables mapped to memory?
  - 有多少个线程可以引用每个实例？ How many threads might reference each of these instances?

映射变量实例到内存
Mapping Variable Instances to Memory

- 全局变量 **Global variables**
  - *定义：* 在函数外部声明的变量 *Def:* Variable declared outside of a function
  - **虚拟内存仅包含任何全局变量的一个实例 Virtual memory contains exactly one instance of any global variable**
- 局部变量 **Local variables**
  - *定义：* 在函数内声明的没有静态属性的变量 *Def:* Variable declared inside function without `static` attribute
  - **每个线程栈包含每个局部变量的一个实例 Each thread stack contains one instance of each local variable**
- 局部静态变量 **Local static variables**
  - *定义：* 在函数内部声明的带有静态属性的变量 *Def:* Variable declared inside function with the `static` attribute
  - **虚拟内存只包含任何本地静态变量的一个实例 Virtual memory contains exactly one instance of any local static variable.**

映射变量实例到内存
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;
}
```

映射变量实例到内存
Mapping Variable Instances to Memory

全局变量：1个实例 ***Global var***: 1 instance (`ptr` [data])

局部变量：1个实例 ***Local vars***: 1 instance (`i.m`, `msgs.m`, `tid.m`)

```
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

局部变量：2个实例 ***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;
}
```

局部静态变量：1个实例
Local static var: 1 instance (`cnt` [data])

共享变量分析 Shared Variable Analysis

- 哪些变量是共享的？ **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

```
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;
}
```

共享变量分析 Shared Variable Analysis

- 哪些变量是共享的？ **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

- 答案：变量x是共享的，当且仅当多个线程引用最少一个x的实例，因此： **Answer: A variable `x` is shared iff multiple threads reference at least one instance of `x`. Thus:**
  - **`ptr`、`cnt`和`msgs`是共享的 `ptr`, `cnt`, and `msgs` are shared**
  - **`i`和`myid`不是共享的 `i` and `myid` are *not* shared**

同步线程 Synchronizing Threads

- 共享变量很方便。。。 **Shared variables are handy...**

- ……但会引入严重同步错误的可能性 **…but introduce the possibility of nasty *synchronization* errors.**

badcnt.c:不正确的同步
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;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt应该等于20,000 cnt should equal 20,000.

发生了什么错？ What went wrong?

计数循环的汇编代码
Assembly Code for Counter Loop

线程i中循环计数的C代码 C code for counter loop in thread i

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

线程i的汇编代码 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 循环头

L_i : Load cnt 装载cnt

U_i : Update cnt 更新cnt

S_i : Store cnt 存储cnt

T_i : Tail 循环尾

并发执行 Concurrent Execution

- *关键思想：* 一般来说，任何顺序一致的*指令交错执行都是可能的，但有些会产生意想不到的结果！ ***Key idea:*** **In general, any sequentially consistent* interleaving is possible, but some give an unexpected result!**
  - I_i表示线程i执行指令I　I_i denotes that thread i executes instruction I
  - %rdxi是线程i上下文中%rdx的内容　$\%rdx_i$ 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

**现在。实际上，在x86上，甚至可以进行非顺序一致的指令交错执行*

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

并发执行 Concurrent Execution

- *关键思想：*一般来说，任何顺序一致的*指令交错执行都是可能的，但有些会产生意想不到的结果！ ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i表示线程i执行指令I I_i denotes that thread i executes instruction I
  - %rdxi是线程i上下文中%rdx的内容 $\%rdx_i$ 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

线程1临界区 Thread 1 critical section

线程2临界区 Thread 2 critical section

OK

并发执行（续）
Concurrent Execution (cont)

- 不正确的顺序：两个线程递增计数器，但结果是**1**而不是**2**
  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

哎呀！ ***Oops!***

并发执行（续）
Concurrent Execution (cont)

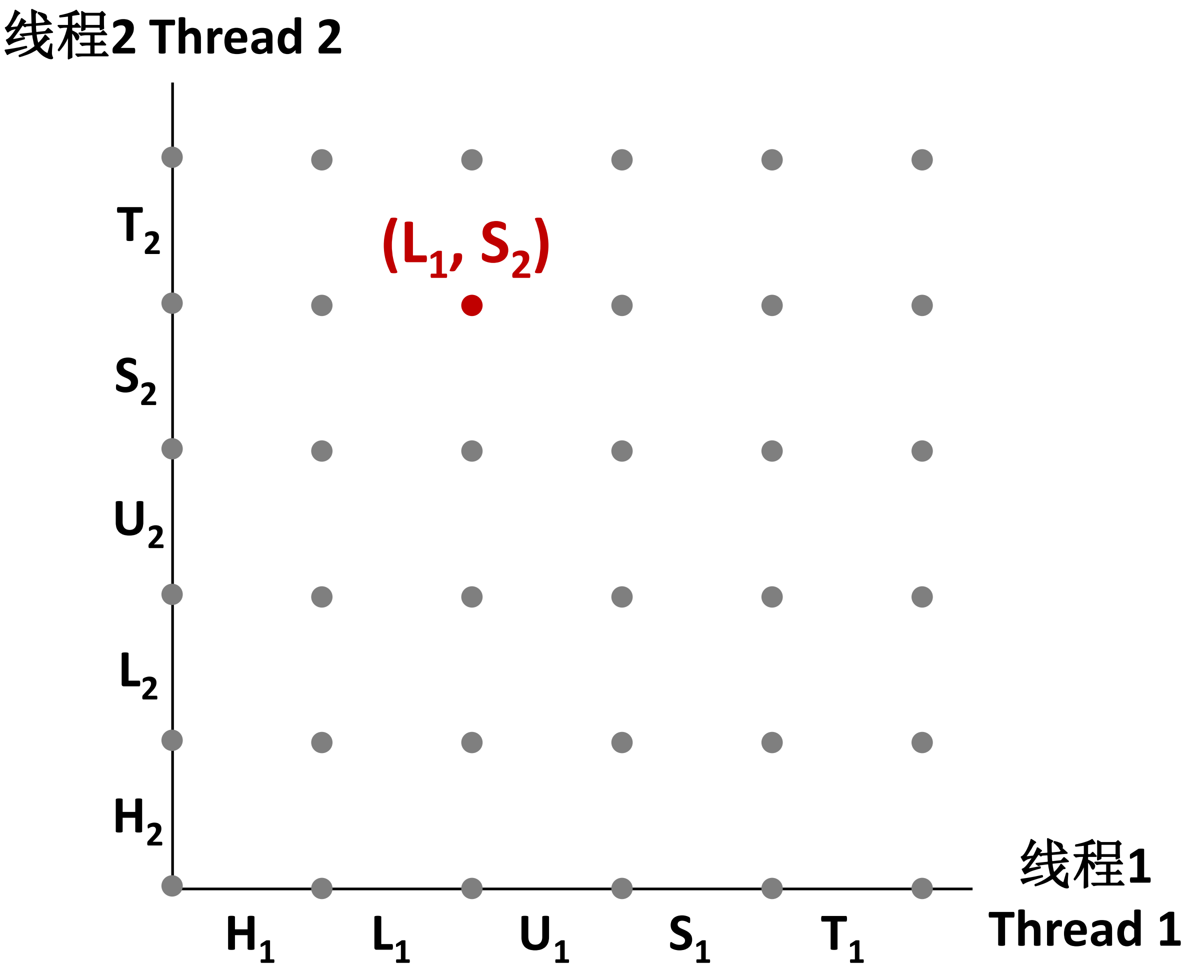
- 这个顺序会怎么样？ **How about this ordering?**

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

哎呀！ Oops!

- 我们可以使用进度图分析行为 **We can analyze the behavior using a *progress graph***

进度图 Progress Graphs



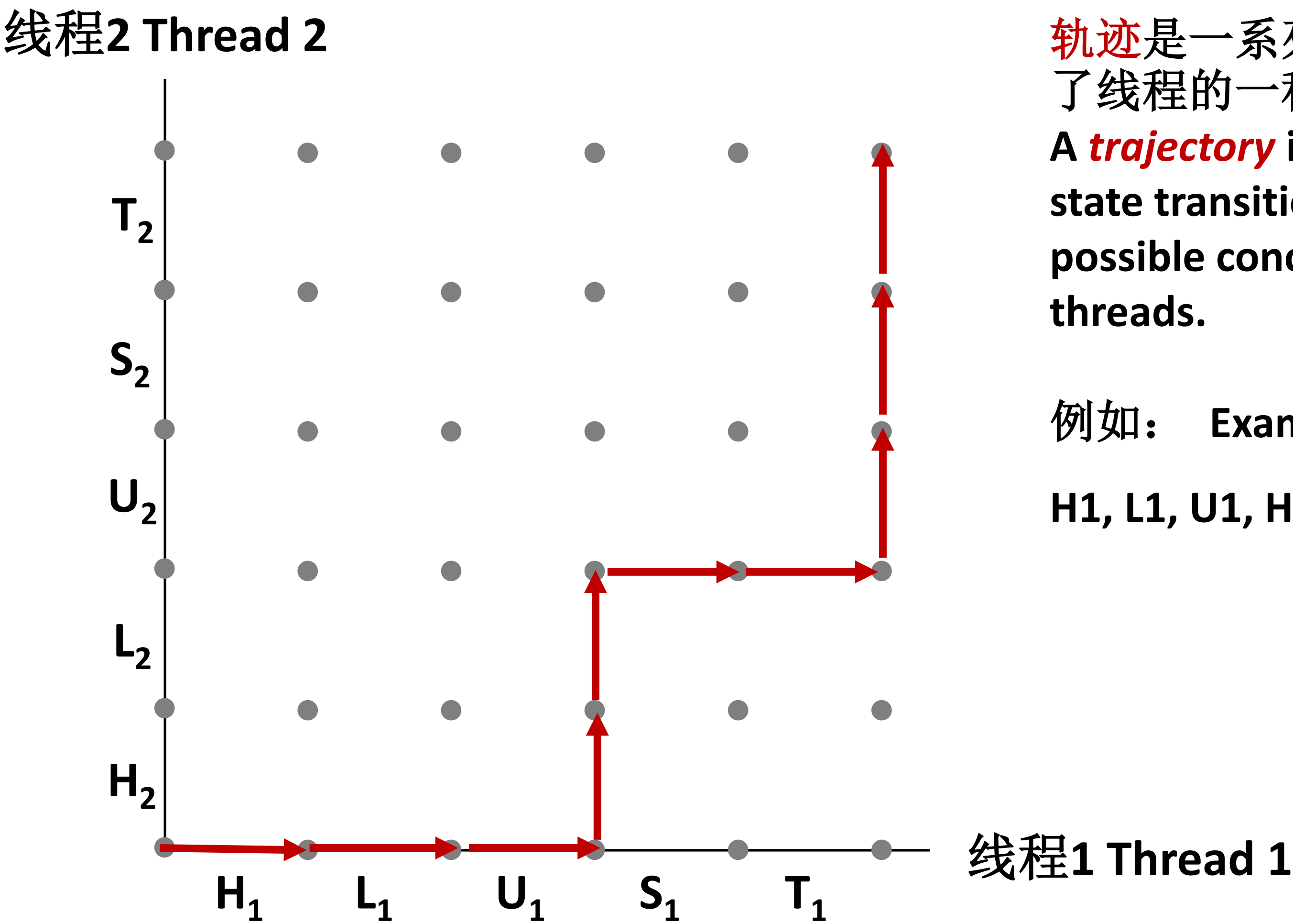
进度图描述了并发线程的离散执行状态空间 A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

每个轴对应于线程中的指令顺序 Each axis corresponds to the sequential order of instructions in a thread.

每个点对应于可能的执行状态 Each point corresponds to a possible ***execution state*** ($Inst_1$, $Inst_2$).

例如（L1，S2）表示状态，其中线程1已完成L1和线程2已完成S2 E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

进度图中的轨迹
Trajectories in Progress Graphs



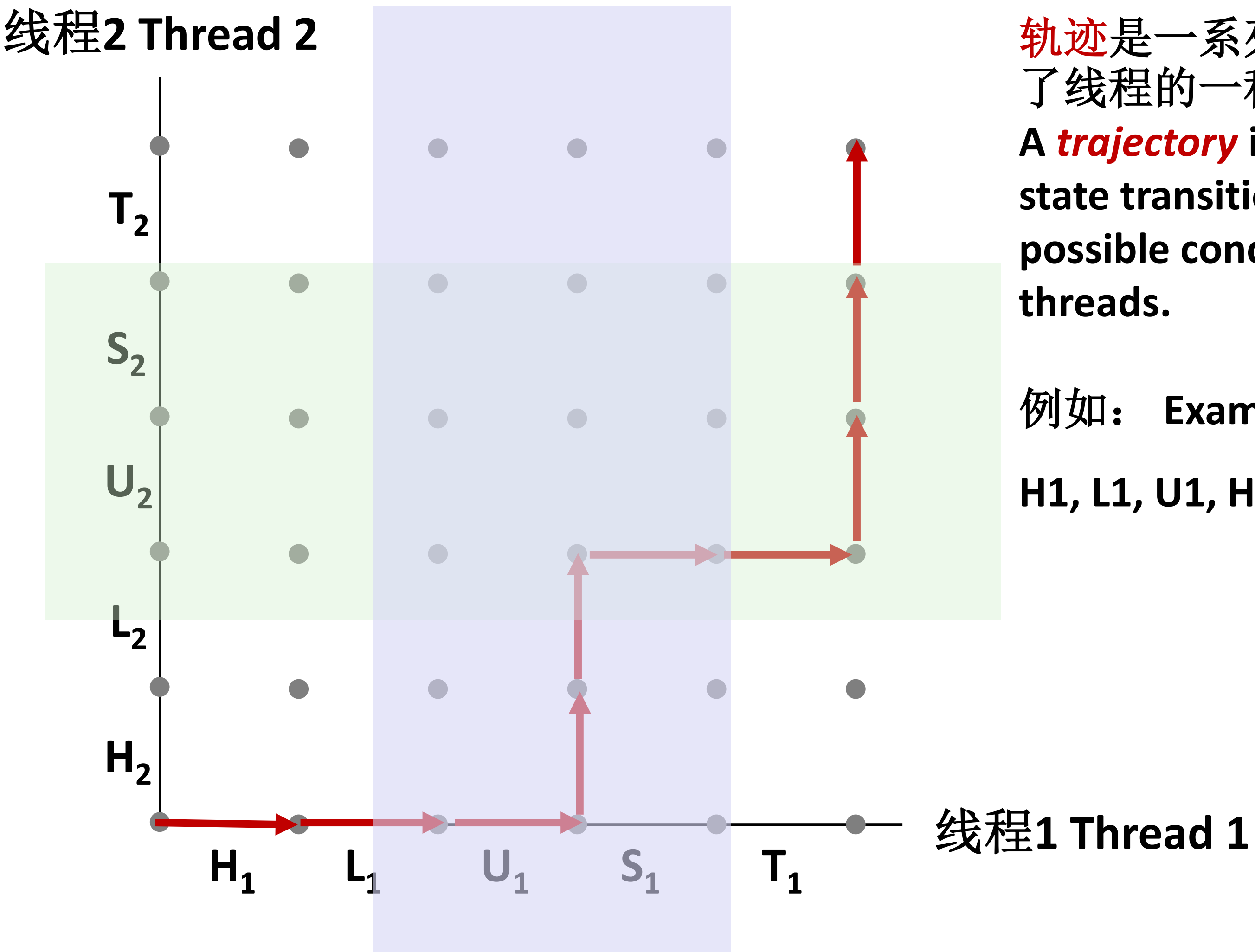
轨迹是一系列合法状态转换，描述了线程的一种可能并发执行。

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

例如： **Example:**

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

进度图中的轨迹
Trajectories in Progress Graphs



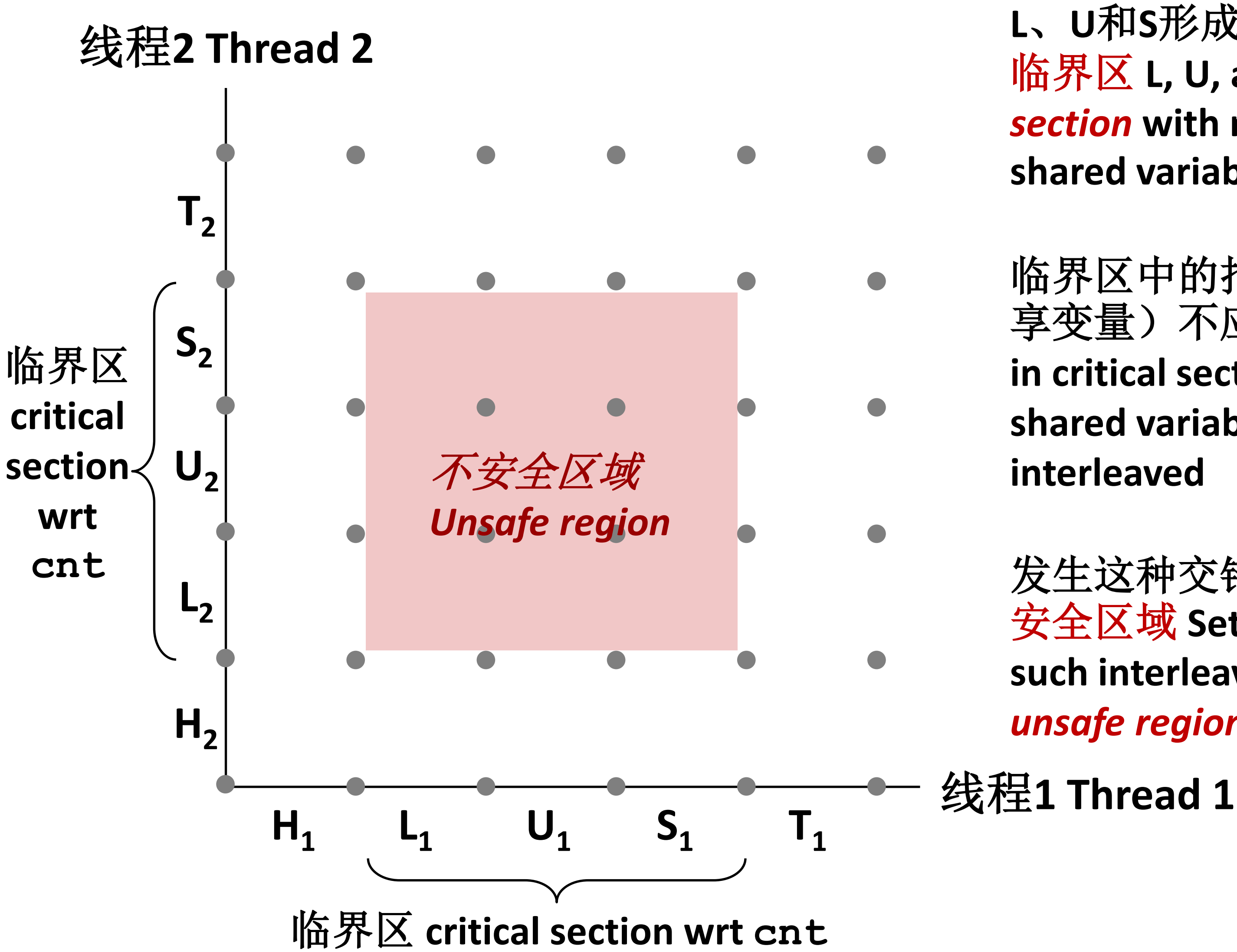
轨迹是一系列合法状态转换，描述了线程的一种可能并发执行。

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

例如： **Example:**

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

临界区和不安全区域
Critical Sections and Unsafe Regions

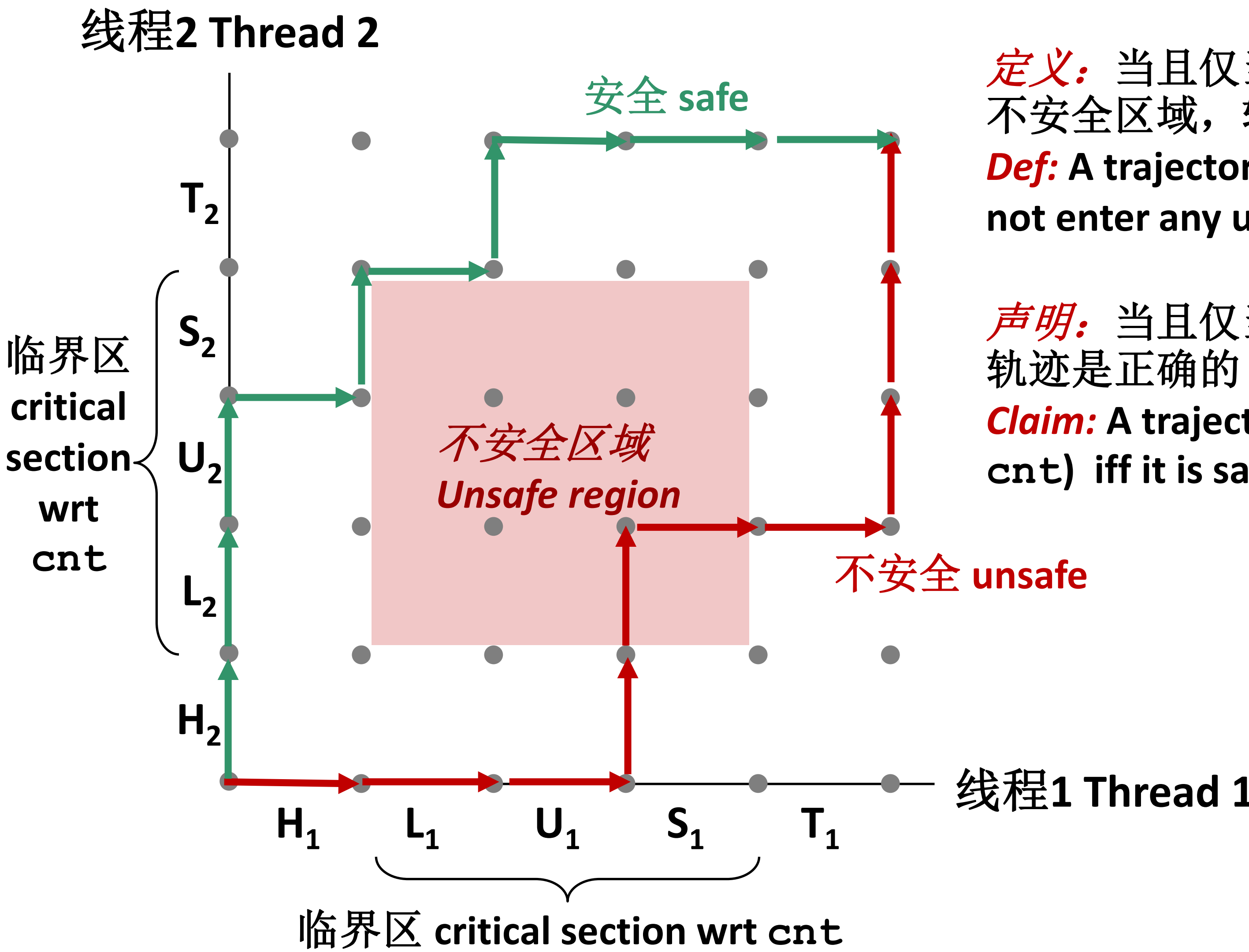


L、U和S形成关于共享变量cnt的临界区 L, U, and S form a *critical section* with respect to the shared variable `cnt`

临界区中的指令（写入一些共享变量）不应交错 Instructions in critical sections (wrt some shared variable) should not be interleaved

发生这种交错的状态集形成不安全区域 Sets of states where such interleaving occurs form *unsafe regions*

临界区和不安全区域
Critical Sections and Unsafe Regions



定义: 当且仅当轨迹不进入任何不安全区域，轨迹是安全的

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

声明: 当且仅当轨迹是安全的，轨迹是正确的（写入cnt）

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

badcnt.c: 不正确的同步

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;
}
```

Variable	main	thread1	thread2
cnt			
niters.m			
tid1.m			
i.1			
i.2			
niters.1			
niters.2			

badcnt.c: 不正确的同步

badcnt.c: Improper Synchronization

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

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

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

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

badcnt.c

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

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

    return NULL;
}
```

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

议题 Today

- 线程回顾 **Threads review**
- 共享 **Sharing**
- **互斥 Mutual exclusion**
- 信号量 Semaphores
- 生产者-消费者同步 **Producer-Consumer Synchronization**

执行互斥 Enforcing Mutual Exclusion

- *问题：*我们如何保证安全的轨迹？ ***Question:* How can we guarantee a safe trajectory?**

- 答：我们必须同步线程的执行，以便它们永远不会有不安全的轨迹 **Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.**
  - 即需要保证每个临界区的互斥访问 i.e., need to guarantee ***mutually exclusive access*** for each critical section.

- 经典解决方案： **Classic solution:**
  - 互斥锁（pthreads） Mutex (pthreads)
  - 信号量（Edsger Dijkstra） Semaphores (Edsger Dijkstra)

- 其他方法（超出我们的讨论范围） **Other approaches (out of our scope)**
  - 条件变量（pthreads） Condition variables (pthreads)
  - 监视器（Java） Monitors (Java)

互斥锁（mutex）
MUTual EXclusion (mutex)

- *互斥锁*：布尔型同步变量 ***Mutex*: boolean synchronization variable**
- **enum {locked = 0, unlocked = 1}**
- **lock(m)**
  - 如果互斥锁当前未锁定，请锁定它并返回 If the mutex is currently not locked, lock it and return
  - 否则，等待（挂起、休眠等）并重试 Otherwise, wait (spinning, yielding, etc) and retry
- **unlock(m)**
  - 将互斥锁状态更新为解锁 Update the mutex state to unlocked

互斥锁（mutex）
MUTual EXclusion (mutex)

- *互斥锁*：布尔型同步变量* **Mutex**: boolean synchronization variable *

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

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

  // [] –通过硬件/OS的魔力实现原子操作 atomic by the magic of hardware / OS

- **Lock(m):**

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

- **Unlock(m):**

  m->state = unlocked;

**现在。实际上，许多其他实现和设计选择（参见15-410、418等）。*

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

badcnt.c: 不正确的同步
badcnt.c: Improper Synchronization

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

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

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

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

badcnt.c

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

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

    return NULL;
}
```

如何使用同步解决此问题？

How can we fix this using synchronization?

goodmcnt.c:互斥锁同步
goodmcnt.c: Mutex Synchronization

- 为共享变量cnt定义并初始化互斥锁: 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
```

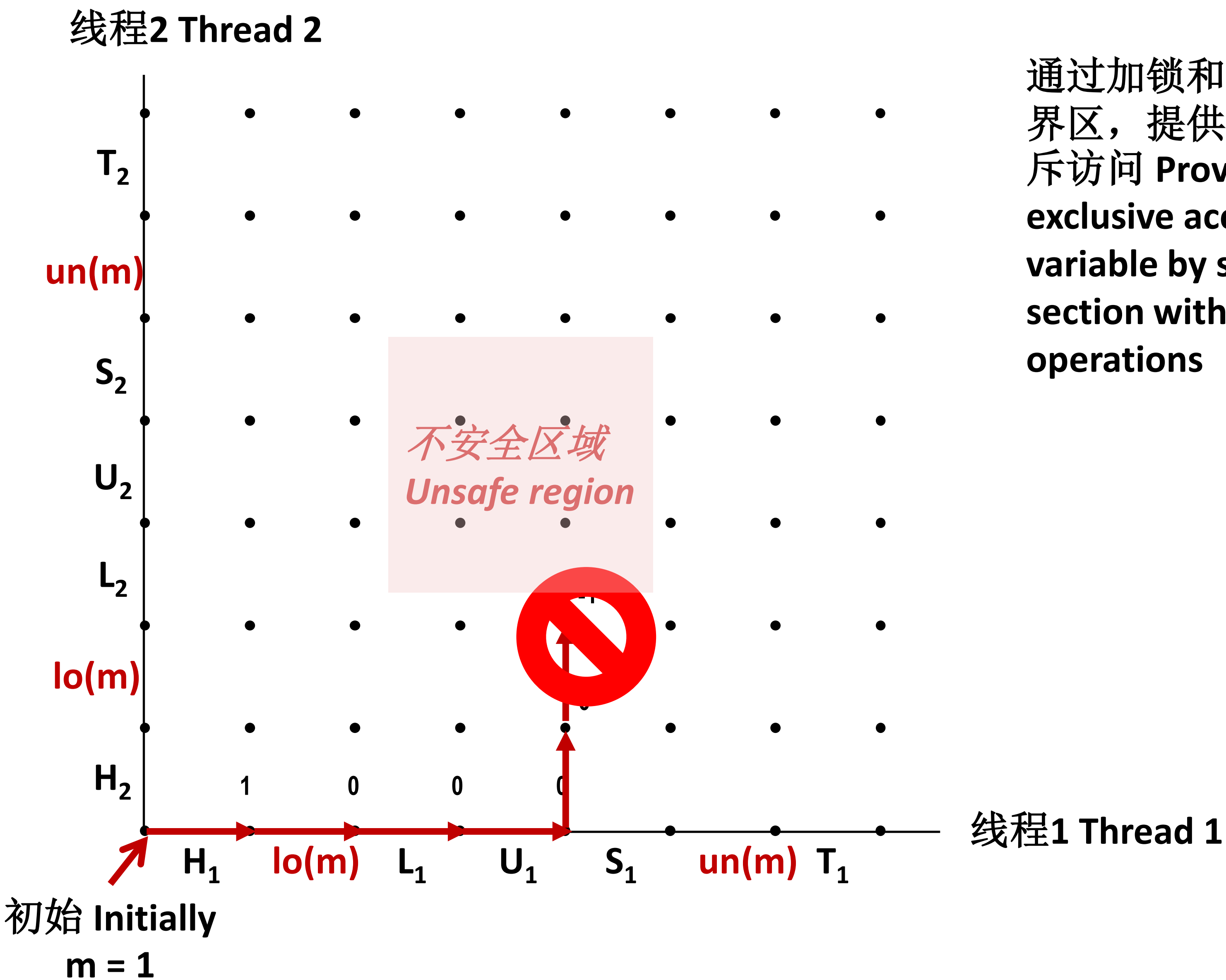
- 用加锁和解锁包围临界区: Surround critical section with *lock* and *unlock*:

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

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

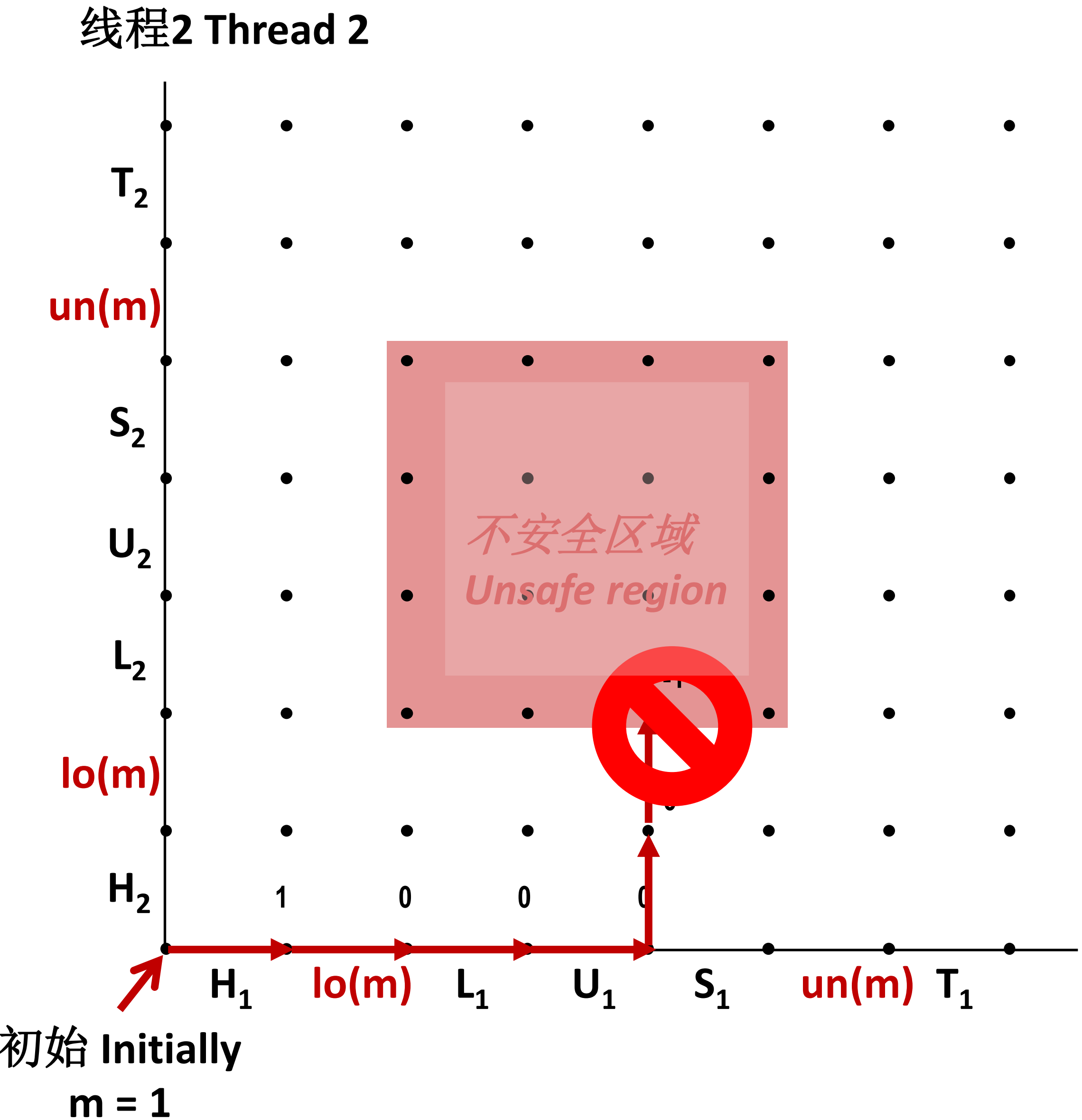
Function	badcnt	goodmcnt
Time (ms) niters = 10^6	12.0	214.0
减速 Slowdown	1.0	17.8

为什么互斥锁有效 Why Mutexes Work



通过加锁和解锁操作围绕临界区，提供对共享变量的互斥访问 Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

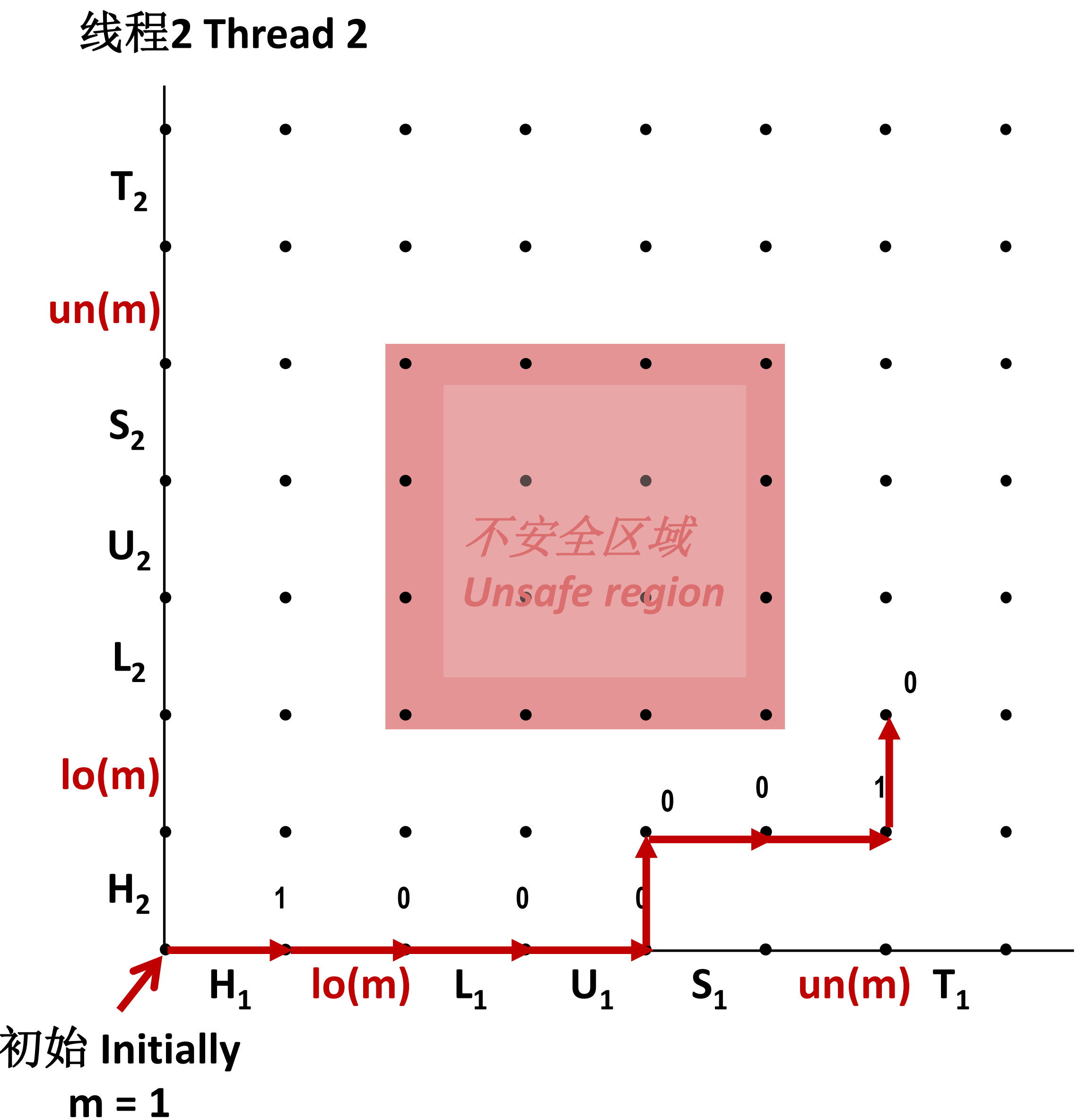
为什么互斥锁有效 Why Mutexes Work



通过加锁和解锁操作围绕临界区，提供对共享变量的互斥访问 **Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations**

互斥锁恒定大于等于零的特性创建了一个封闭不安全区域的禁区，任何轨迹都无法进入 **Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.**

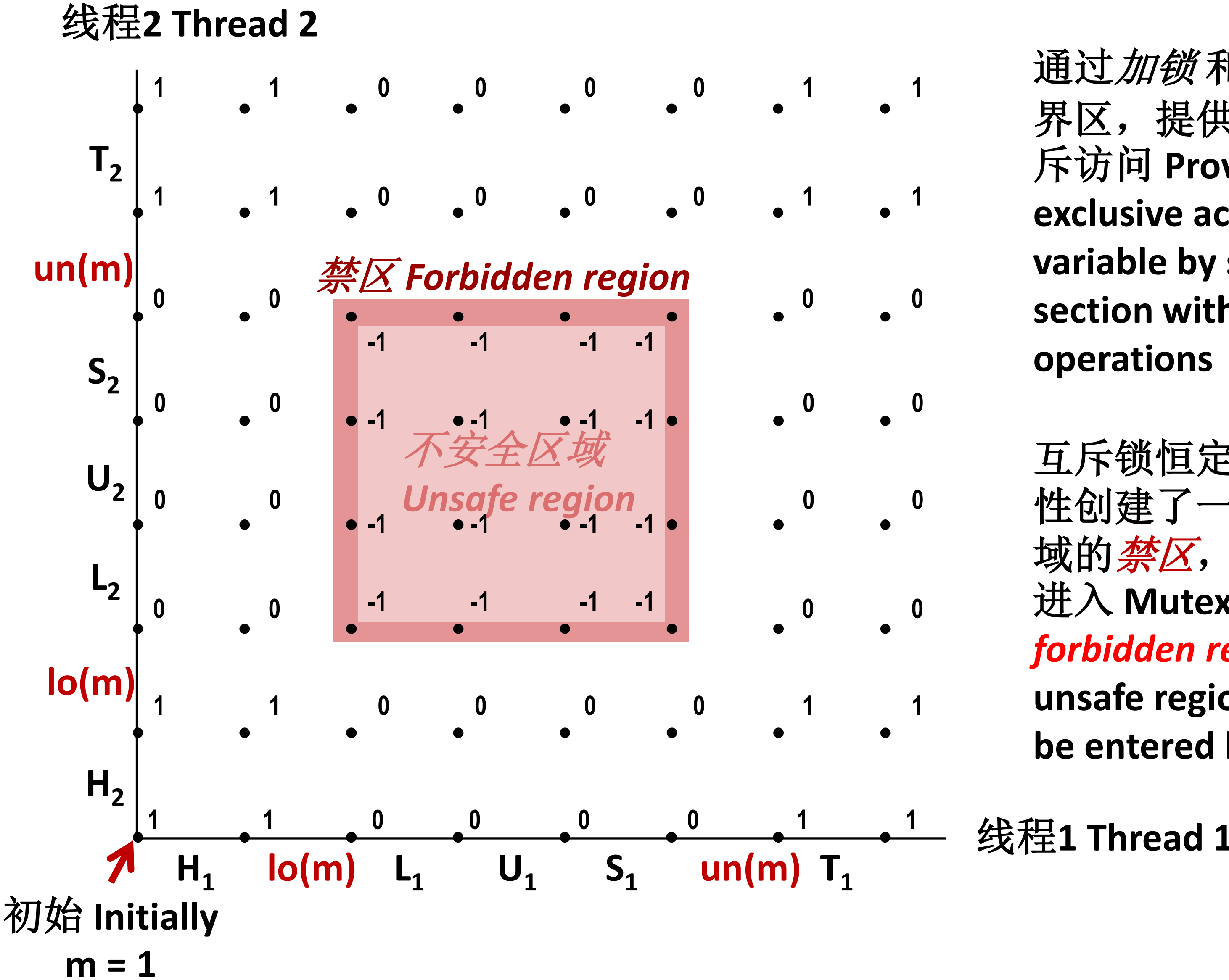
为什么互斥锁有效 Why Mutexes Work



通过*加锁*和*解锁*操作围绕临界区，提供对共享变量的互斥访问 **Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations**

互斥锁恒定大于等于零的特性创建了一个封闭不安全区域的*禁区*，任何轨迹都无法进入 **Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.**

为什么互斥锁有效 Why Mutexes Work



通过*加锁*和*解锁*操作围绕临界区，提供对共享变量的互斥访问 **Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations**

互斥锁恒定大于等于零的特性创建了一个封闭不安全区域的*禁区*，任何轨迹都无法进入 **Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.**

议题 Today

- 线程回顾/Threads review
- 共享 Sharing
- 互斥 Mutual exclusion
- **信号量 Semaphores**
- 生产者-消费者同步 Producer-Consumer Synchronization

信号量 Semaphores

- *信号量：*非负全局整数同步变量，由P和V操作操纵 ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s)**
  - 如果s为非零，则将s减1并立即返回 If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - 测试和减1操作以原子方式发生（不可分割） Test and decrement operations occur atomically (indivisibly)
  - 如果s为零，则挂起线程，直到s变为非零，并通过V操作重新启动线程 If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - 重新启动后，P操作将s减1并将控制权返回给调用者 After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - 将s递增1 Increment s by 1.
    - 增量操作以原子方式发生 Increment operation occurs atomically
  - 如果在P操作中有任何线程被阻塞，等待s变为非零，那么只重新启动其中一个线程，然后通过将s减1来完成P操作 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.
- 信号量恒定大于等于零： **Semaphore invariant: *(s >= 0)***

信号量 Semaphores

- *信号量：*非负全局整数同步变量 ***Semaphore:*** **non-negative global integer synchronization variable**

- **由P和V操作操纵 Manipulated by *P* and *V* operations:**
  - *P(s):* [`while (s == 0) wait(); s--;`]
    - 荷兰语单词“Proberen”（测试） Dutch for "Proberen" (test)
  - *V(s):* [`s++;`]
    - 荷兰语单词“Verhogen”（增加） Dutch for "Verhogen" (increment)

- **OS内核保证括号[]之间的操作不可分割地执行 OS kernel guarantees that operations between brackets [ ] are executed indivisibly**
    - 一次只能一个P或V操作修改s Only one *P* or *V* operation at a time can modify s.
    - 当P中的while循环终止时，只有该P操作可以减少s When `while` loop in *P* terminates, only that *P* can decrement `s`

- **信号量恒定大于等于零： Semaphore invariant: *(s >= 0)***

C语言信号量操作
C Semaphore Operations

Pthread函数 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包装器函数 CS:APP wrapper functions:

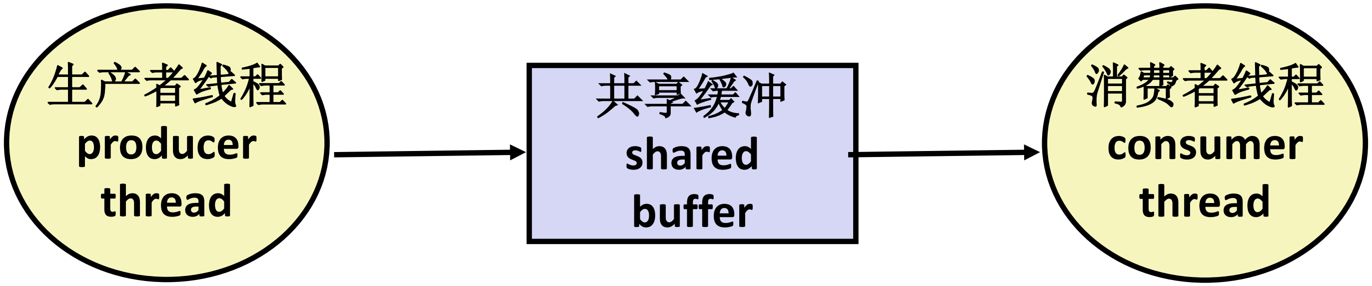
```
#include "csapp.h”

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

使用信号量协调共享资源的访问 Using Semaphores to Coordinate Access to Shared Resources

- 基本思想：线程使用信号量操作通知另一个线程某些条件已变为真 **Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true**
  - 使用计数信号量来跟踪资源状态 Use counting semaphores to keep track of resource state.
  - 使用二元信号量通知其他线程 Use binary semaphores to notify other threads.

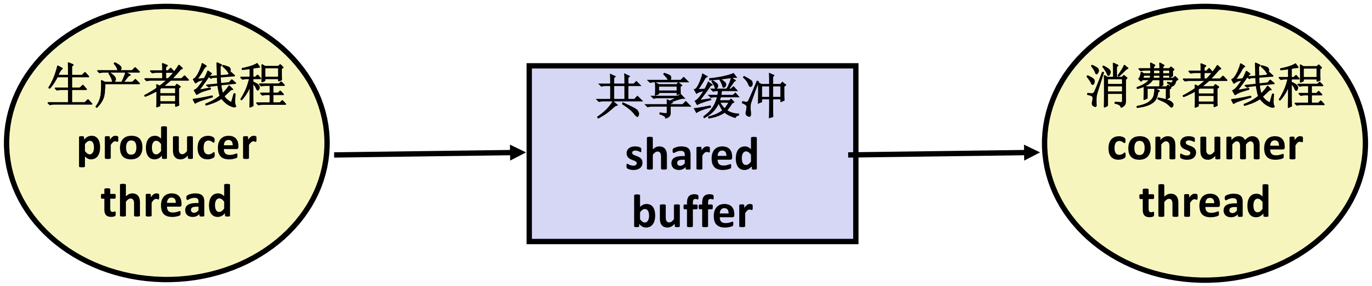
- 生产者-消费者问题 **The Producer-Consumer Problem**
  - 对进程之间的交互操作进行协调，一个进程产生信息，另一个进程使用这些消息 Mediating interactions between processes that generate information and that then make use of that information

生产者-消费者问题
Producer-Consumer Problem



- 通用同步模式： **Common synchronization pattern:**
  - 生产者等待空槽，将项目插入缓冲区，并通知消费者 Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
  - 消费者等待项目，将其从缓冲区中删除，并通知生产者 Consumer waits for ***item***, removes it from buffer, and notifies producer

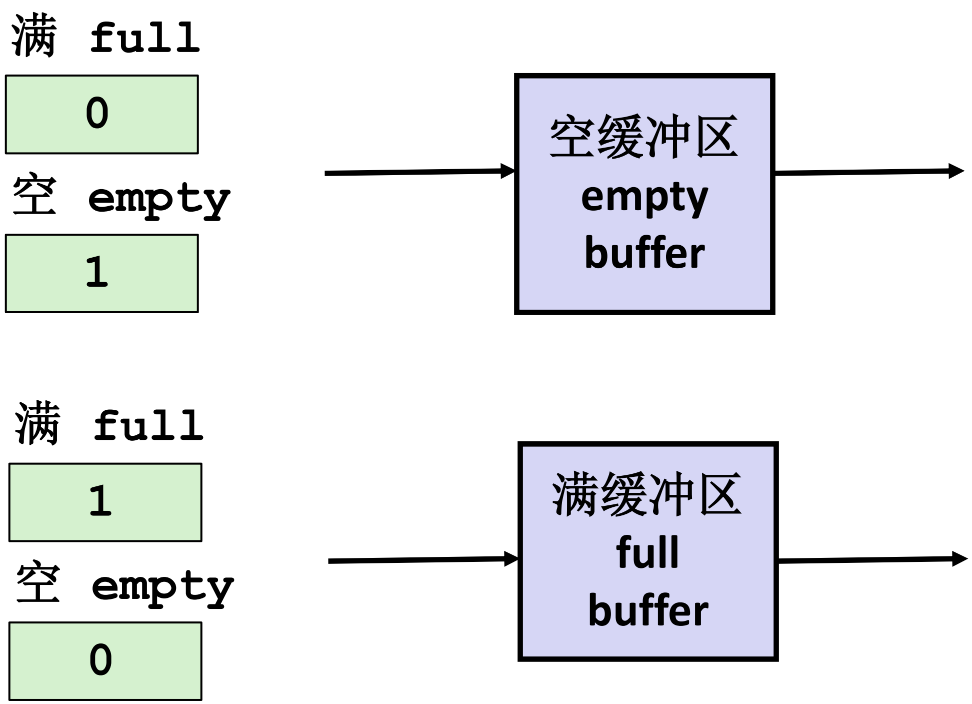
生产者-消费者问题
Producer-Consumer Problem



- **示例 Examples**
  - 多媒体处理： Multimedia processing:
    - 生产者创建视频帧，消费者对其进行渲染 Producer creates video frames, consumer renders them
  - 事件驱动的图形用户界面 Event-driven graphical user interfaces
    - 生产者检测鼠标点击、鼠标移动和键盘点击，并在缓冲区中插入相应的事件 Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - 消费者从缓冲区检索事件并绘制显示 Consumer retrieves events from buffer and paints the display

生产者和消费者之间有1个元素的缓冲区
Producer-Consumer on 1-element Buffer

- 维护两个信号量：缓冲区满full+缓冲区空 **Maintain two semaphores: `full` + `empty`**



生产者和消费者之间有1个元素的缓冲区
Producer-Consumer on 1-element Buffer

```c
#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;
```

```c
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;
}
```

生产者和消费者之间有1个元素的缓冲区
Producer-Consumer on 1-element Buffer

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

生产者线程 Producer Thread

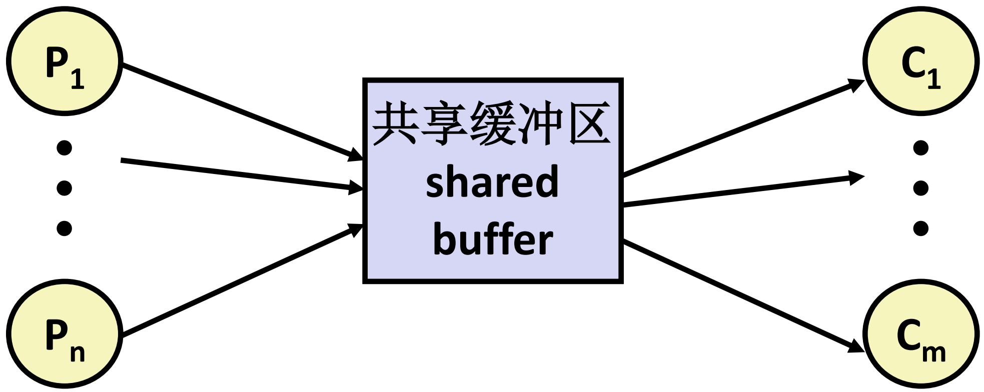
```
void *producer(void *arg) {
  int i, item;

  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;
}
```

消费者线程 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;
}
```

为何对一个条目的缓冲区使用2个信号量？
Why 2 Semaphores for 1-Entry Buffer?

- 考虑多个生产者和多个消费者 **Consider multiple producers & multiple consumers**



- 生产者将与每个人竞争以获得空缓冲区 **Producers will contend with each to get `empty`**
- 消费者将相互竞争以获得满缓冲区 **Consumers will contend with each other to get `full`**

生产者 Producers

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

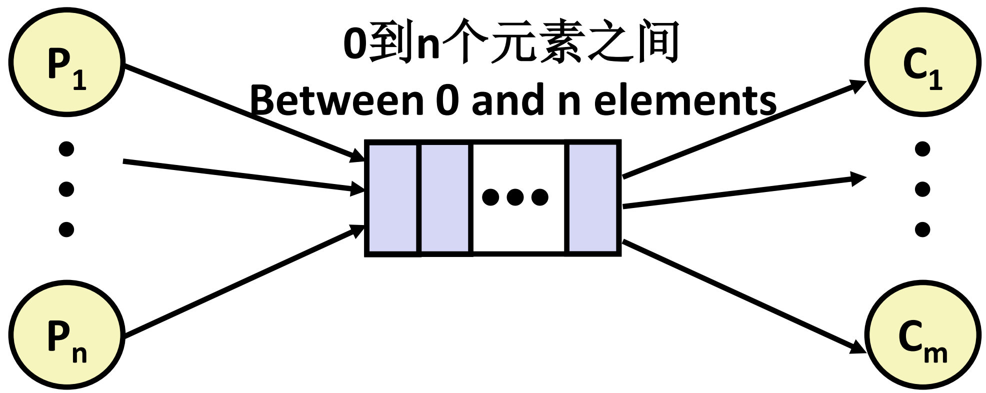
空 `empty`

满 `full`

消费者 Consumers

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

生产者和消费者之间有n个元素的缓冲区
Producer-Consumer on an *n*-element Buffer



- 使用名为sbuf的共享缓冲区包实现 **Implemented using a shared buffer package called `sbuf`.**

环形缓冲区（n=10）
Circular Buffer (n = 10)

- 将元素存储在大小为n的数组中 **Store elements in array of size n**
- 项目：缓冲区中的元素数 **items: number of elements in buffer**
- 空缓冲区： **Empty buffer:**
  - front = rear
- 非空缓冲区 **Nonempty buffer**
  - rear：最近插入的元素的索引 rear: index of most recently inserted element
  - front：(要删除的下一个元素的索引–1)mod n front: (index of next element to remove – 1) mod n
- 初始 **Initially:**

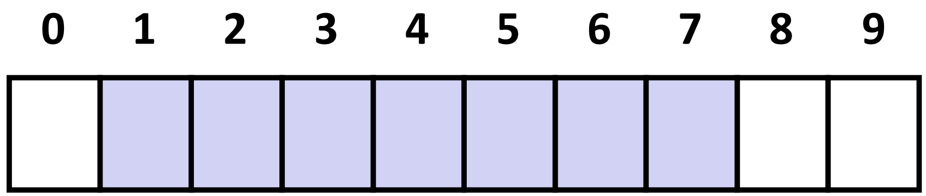
front	**0**
rear	**0**
项目 **items**	**0**

0	1	2	3	4	5	6	7	8	9

环形缓冲区操作（n=10）
Circular Buffer Operation (n = 10)

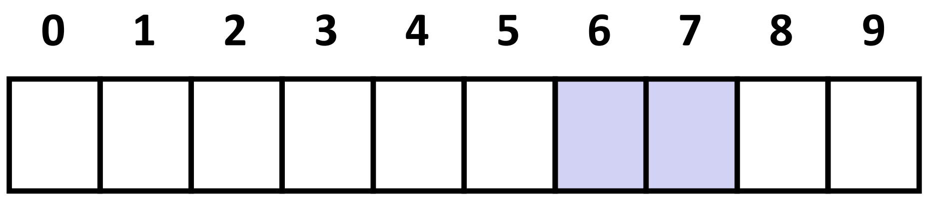
- 插入7个元素 Insert 7 elements

front	0
rear	7
items	7



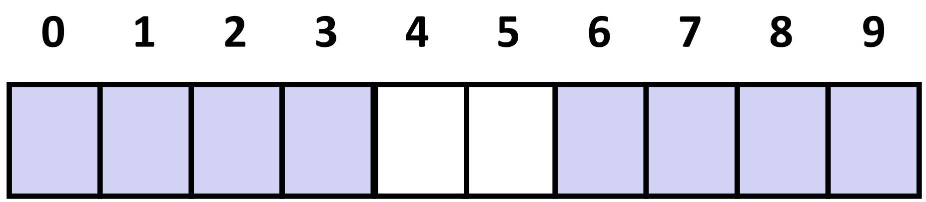
- 删除5个元素 Remove 5 elements

front	5
rear	7
items	2



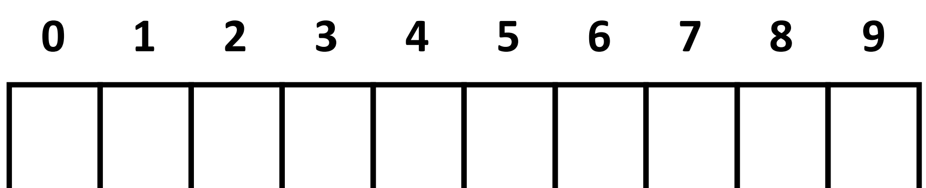
- 插入6个元素 Insert 6 elements

front	5
rear	3
items	8



- 删除8个元素 Remove 8 elements

front	3
rear	3
items	0



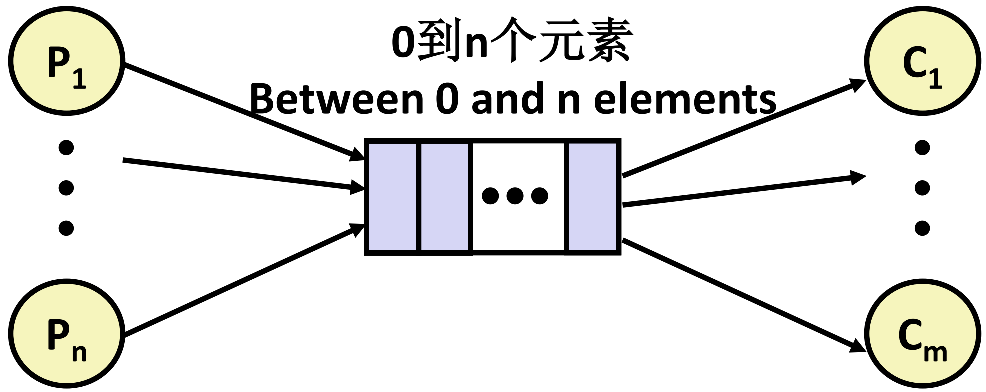
顺序环形缓冲区代码
Sequential Circular Buffer Code

```
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;
}
```

生产者和消费者之间有n个元素的缓冲区
Producer-Consumer on an *n*-element Buffer



- 需要一个互斥锁和两个计数信号量： **Requires a mutex and two counting semaphores:**
  - 互斥锁：执行对缓冲区和计数器进行互斥访问 `mutex`: enforces mutually exclusive access to the buffer and counters
  - 槽位数：统计缓冲区中的可用槽位 `slots`: counts the available slots in the buffer
  - 项目：统计缓冲区中的可用项目 `items`: counts the available items in the buffer
- 使用通用信号量 **Makes use of general semaphores**
  - 值范围从0到n Will range in value from 0 to n

sbuf包-声明 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          */
    pthread_mutex_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包-实现
sbuf Package - Implementation

初始化和释放共享缓冲区 **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->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包-实现
sbuf Package - Implementation

插入一个项目到共享缓冲区 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 */
    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   */
    V(&sp->items);                          /* Announce available item */
}
```

sbuf.c

sbuf包-实现
sbuf Package - Implementation

从共享缓冲区删除一个项目 **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 */
    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     */
    V(&sp->slots);                   /* Announce available slot */
    return item;
}
```

sbuf.c

演示 Demonstration

- **参见code目录中的程序produce-consume.c See program produce-consume.c in code directory**
- **10个条目的共享环形缓冲区 10-entry shared circular buffer**
- **5个生产者 5 producers**
  - 代理i生成从20*i到20*i–1的数字 Agent i generates numbers from 20*i to 20*i – 1.
  - 将它们放入缓冲区 Puts them in buffer
- **5个消费者 5 consumers**
  - 每个从缓冲区中检索20个元素 Each retrieves 20 elements from buffer
- **主程序 Main program**
  - 确保0到99之间的每个值检索一次 Makes sure each value between 0 and 99 retrieved once

小结 Summary

- 程序员需要一个线程如何共享变量的清晰模型。 **Programmers need a clear model of how variables are shared by threads.**

- 必须保护多个线程共享的变量，以确保互斥访问 **Variables shared by multiple threads must be protected to ensure mutually exclusive access.**

- 信号量是执行互斥的基本机制 **Semaphores are a fundamental mechanism for enforcing mutual exclusion.**