同步

Synchronization

多线程C程序中的共享变量

- **问题：在多线程的C程序中，哪些变量是共享的?**
  - 答案并不像“全局变量是共享的”和“栈变量是私有的”那样简单
- **定义：变量x是共享的，当且仅当多个线程引用了x的某个实例**
- **需要首先回答以下问题：**
  - 线程的内存模型是什么?
  - 变量实例是如何映射到内存的?
  - 每个这些实例可能被多少个线程引用?

线程内存模型

- **概念模型：**
  - 多个线程在单个进程的上下文中运行
  - 每个线程都有自己独立的线程上下文
    - 线程ID、栈、栈指针、程序计数器、条件码和通用寄存器
  - 所有线程共享进程上下文
    - 进程虚拟地址空间的代码、数据、堆和共享库段
    - 打开的文件和已初始化好的信号处理程序
- **在操作上，该模型并非被严格执行：**
  - 寄存器值是真正独立和受保护的，但是……
  - 任何线程都可以读取和写入任何其他线程的栈
- **概念模型与操作模型之间的不匹配是混淆和错误的源泉**

示例：共享变量

```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;
}
```

对等线程通过全局变量ptr间接引用主线程的栈。

将变量映射到内存

- 全局变量：
  - 定义：在函数外声明的变量
  - 虚拟内存中包含任何全局变量的确切一个实例
- 本地变量：
  - 定义：在函数内部声明的没有静态属性的变量
  - 每个线程栈包含每个局部变量的一个实例
- 本地静态变量：
  - 定义：在函数内部声明具有静态属性的变量
  - 虚拟内存中包含任何局部静态变量的确切一个实例。

同步

Synchronization

将变量映射到内存

全局变量: 1 个实例 (ptr [位于.data])　本地变量: 1 个实例 (i.m, msgs.m), .m表示主线程

```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

本地变量: 2 个实例 (
myid.p0 [位于对等线程0的栈],
myid.p1 [位于对等线程1的栈]
)

```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;
}
```

本地静态变量: 1 个实例(cnt [位于.data])

共享变量分析

- 哪些变量是共享的?

变量实例	被主线程引用	被对等线程0引用	被对等线程1引用
ptr	Y	Y	Y
cnt	N	Y	Y
i.m	Y	N	N
msgs.m	Y	Y	Y
myid.p0	N	Y	N
myid.p1	N	N	Y

- 答案：变量x是共享的当且仅当多个线程引用了x的至少一个实例。因此：
  - ptr、cnt 和 msgs 是共享的
  - i 和 myid 不是共享的

同步

Synchronization

线程同步

- 共享变量很方便，但也引入了出现同步错误的可能性

```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>
```

哪里出问题了？

计数循环中的汇编指令

线程i中的计数循环（C语言）

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

汇编语言

```
    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: 头

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

T_i: 尾

并发执行

- **关键问题：**通常情况下，任何顺序一致的交错都是可能的，但有些会产生意外的结果!
  - I_i 表示线程i执行指令I
  - $\%rdx_i$表示线程i上下文中%rdx的内容

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 临界区

Thread 2临界区

可行的执行顺序

并发执行（2）

- 错误的顺序：结果是1，而不是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

Thread 1 临界区

Thread 2临界区

出错

并发执行（2）

- 同样是错误的顺序

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

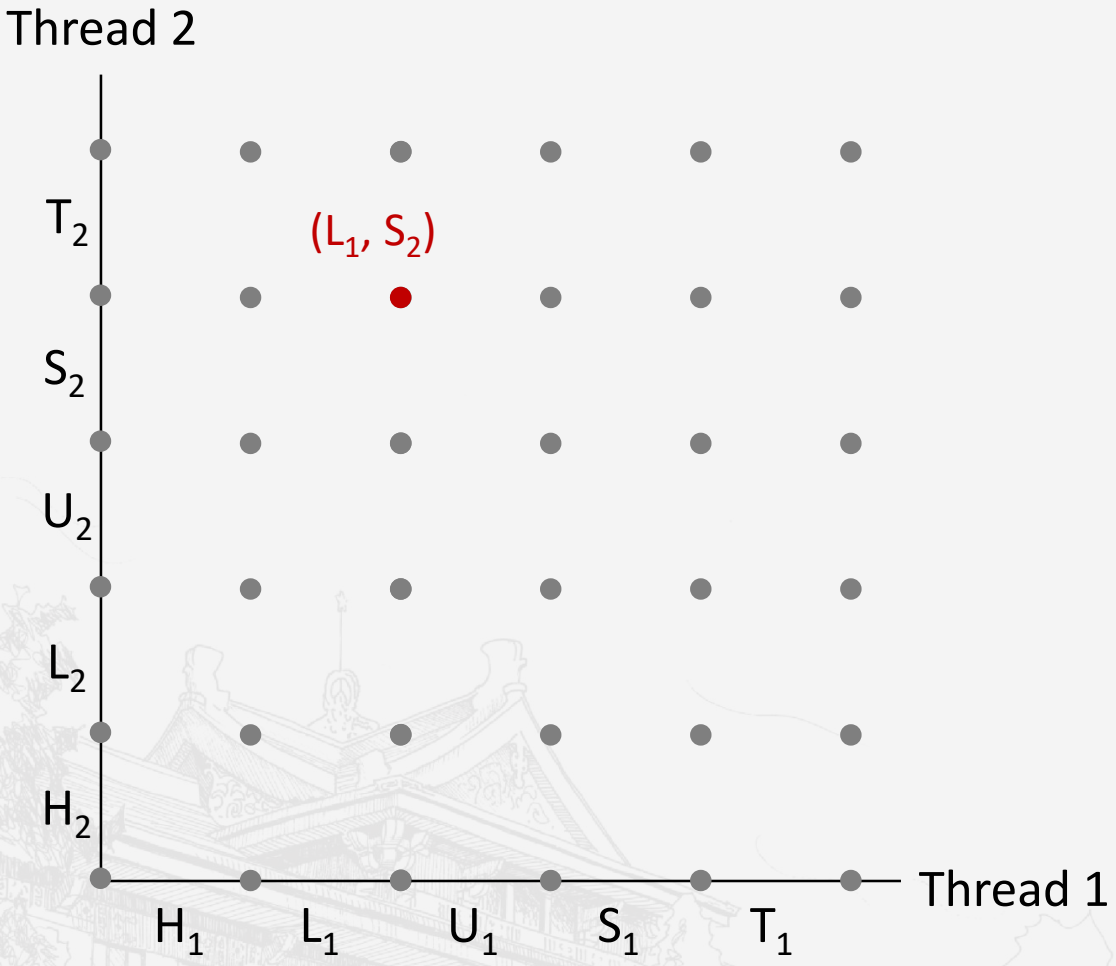
出错

Thread 1 临界区

Thread 2临界区

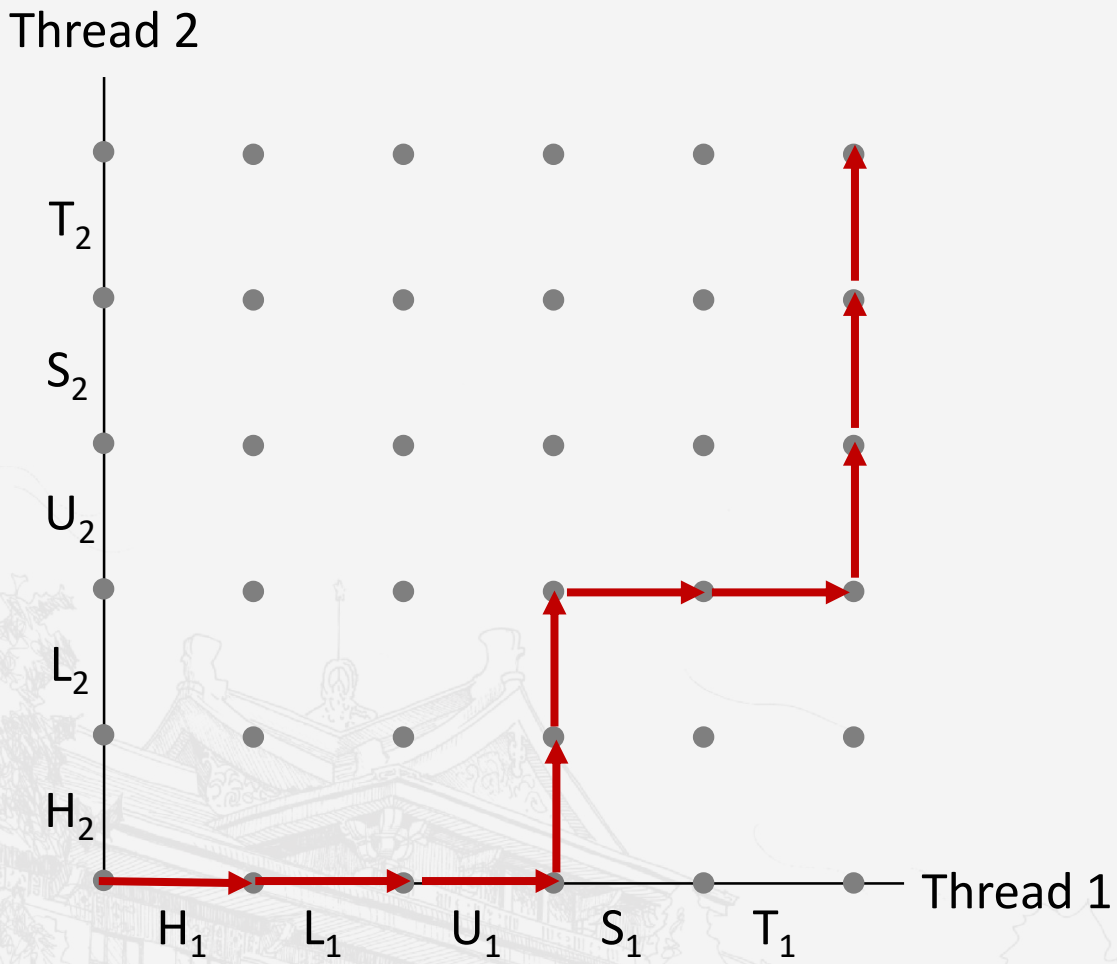
- 可以使用进程图进行分析

进程图



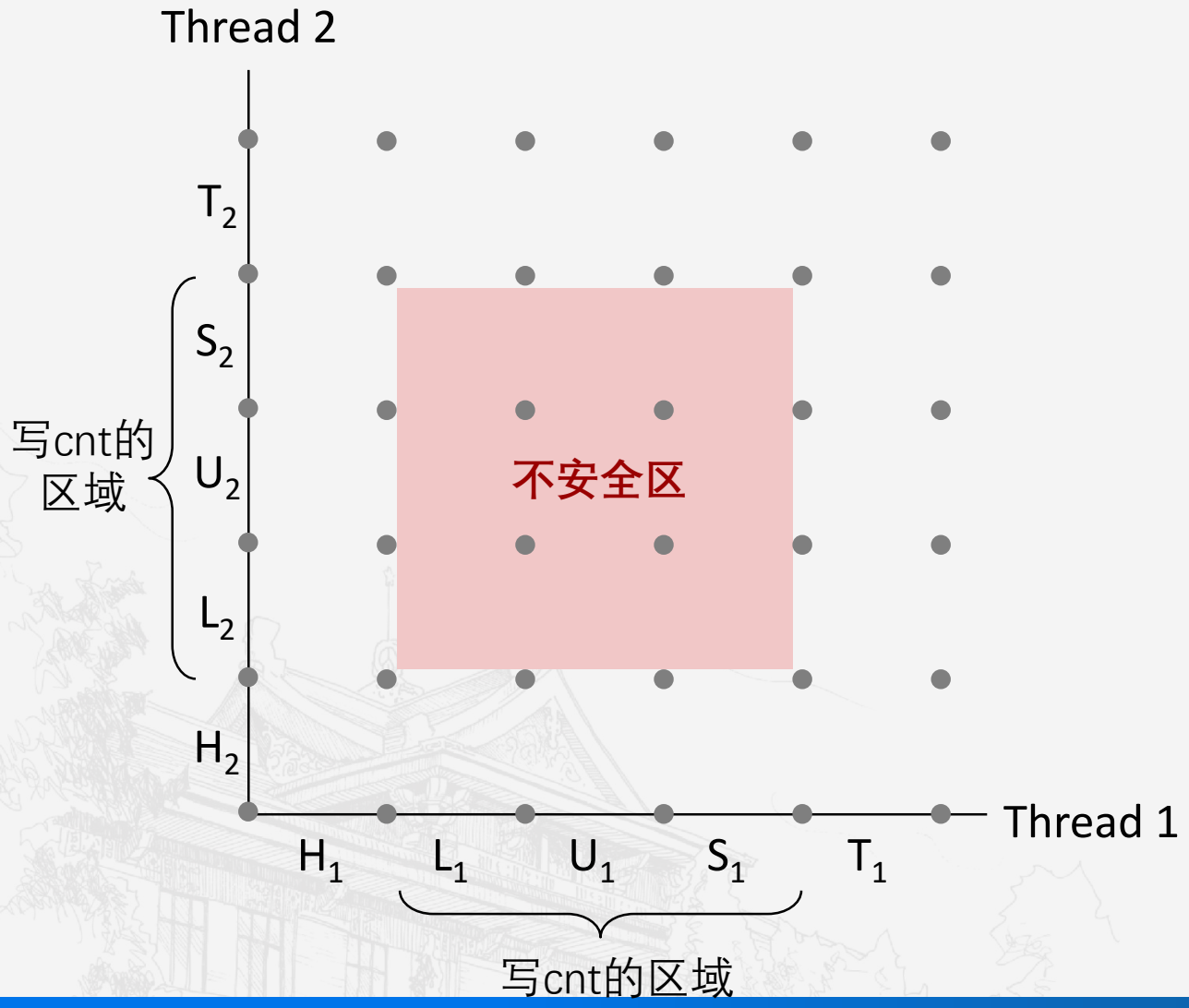
- 进度图描述了并发线程的离散执行状态空间。
- 每个轴对应于线程中指令的顺序。
- 每个点对应于一个可能的执行状态（Inst1，Inst2）。
- 例如：（L1，S2）表示线程1已完成L1，线程2已完成S2。

进程图中的轨迹



- 轨迹是描述线程并发执行中的一次可能的状态转换过程。
- 例如：H1，L1，U1，H2，L2，S1，T1，U2，S2，T2

临界区和不安全区

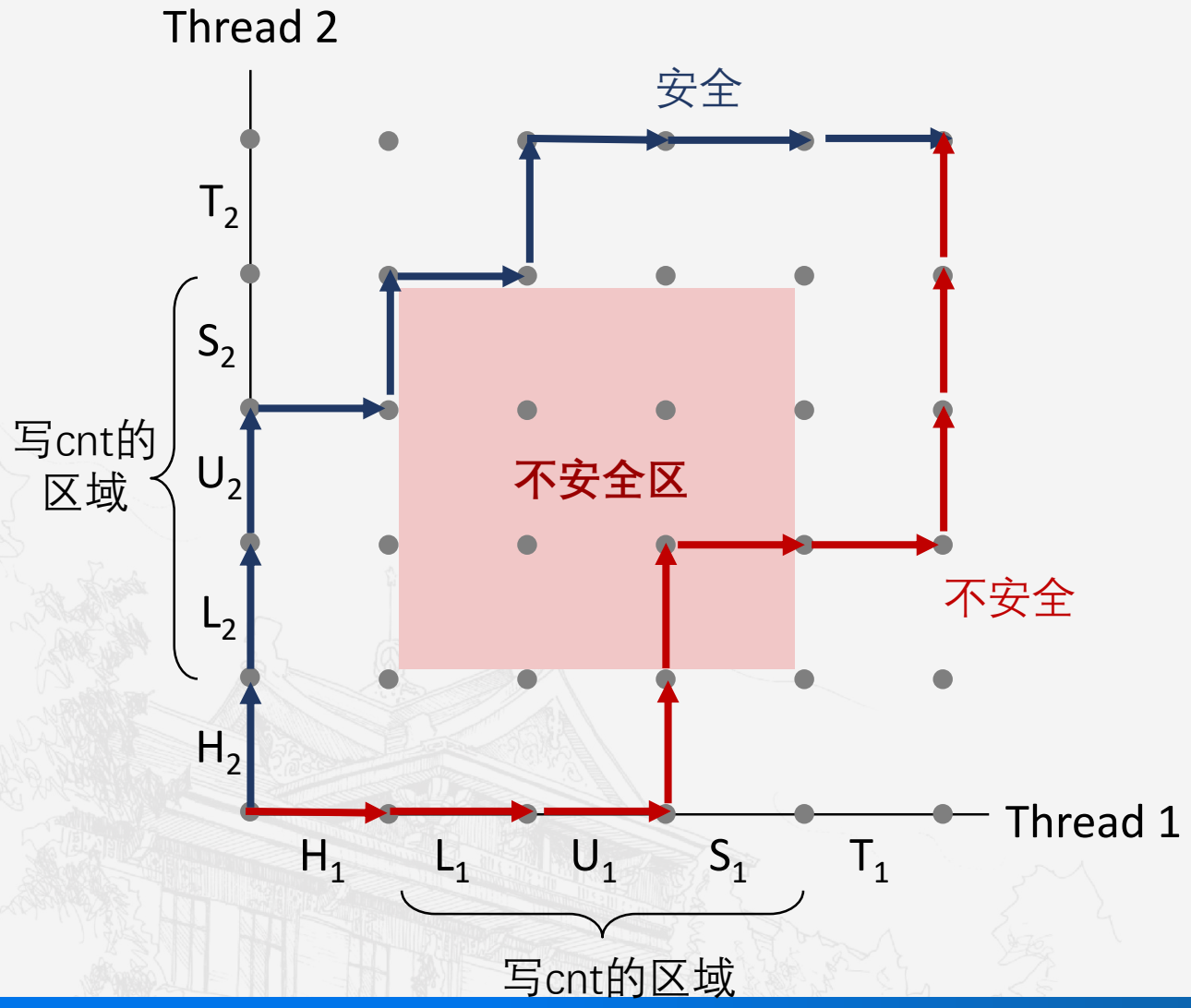


- L、U 和 S 相对于共享变量 cnt 形成了一个**临界区**
- 临界区中的指令（写某个共享变量）不应该交错执行
- 这种交错发生的状态集合构成了不安全区

临界资源：多个线程需要互斥访问或修改的共享资源。

临界区：涉及对临界资源的访问或修改的代码片段。

临界区和不安全区（2）



- 定义：如果轨迹不进入任何不安全的区域，则该轨迹是**安全**的。
- 如果轨迹是安全的（相对于 cnt），则程序结果是正确的。

强制互斥访问

- **问题：我们如何确保一个安全的轨迹?**
- 答案：必须**同步**线程的执行，以便它们永远不会产生不安全的轨迹
  - 即需要为每个临界区**保证互斥访问**。
- 经典解决方案：信号量（Dijkstra）
- 其他方法（不在我们的范围内）：
  - 互斥锁和条件变量（Pthreads）
  - 监视器（Java）

信号量（Semaphores）

- 信号量：非负整数的全局同步变量。通过P和V操作进行操作。
- P操作：
  - 如果s不为零，则将s减1并立即返回。
  - 测试和减少操作是原子性的（不可分割的）。
  - 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。
  - 重新启动后，P操作将减少s并将控制返回给调用者。
- V操作：
  - 将s增加1。
  - 增量操作是原子性的。
  - 如果有任何线程在P操作中被阻塞，等待s变为非零，则精确地重新启动其中一个线程，然后该线程通过减少s完成其P操作。
- 信号量的不变性：s >= 0

C语言信号量操作

Pthread库函数

```
#include <semaphore.h>
/*
    s 待初始化的信号量
    pshared 0 线程间共享信号量，1进程间共享信号量
    val 信号量的初值
*/
int sem_init(sem_t *s, int pshared, unsigned int val);


int sem_wait(sem_t *s);  /* P 操作 */
int sem_post(sem_t *s);  /* V 操作 */
```

不正确的同步

```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;
}
```

如何使用信号量机制解决代码中的问题?

使用信号量实现互斥

- 基本思想：
  - 将每个共享变量（或相关的一组共享变量）与一个唯一的信号量 mutex 关联起来，初始值为 1。
  - 将相应的临界区用 P(mutex) 和 V(mutex) 操作包围起来。
- 名词解释：
  - 二进制信号量（Binary semaphore）：其值始终为 0 或 1 的信号量
  - 互斥锁（Mutex）：用于互斥访问的二进制信号量
    - P 操作：“锁定”互斥锁
    - V 操作：“解锁”或“释放”互斥锁
    - “持有”互斥锁：被锁定且尚未解锁。
  - 计数信号量（Counting semaphore）：用作可用资源集合的计数器。

正确的同步

- 定义和初始化一个互斥锁（mutex）来保护共享变量 cnt:

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- 使用 P 和 V 操作包裹住临界区

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

goodcnt.c

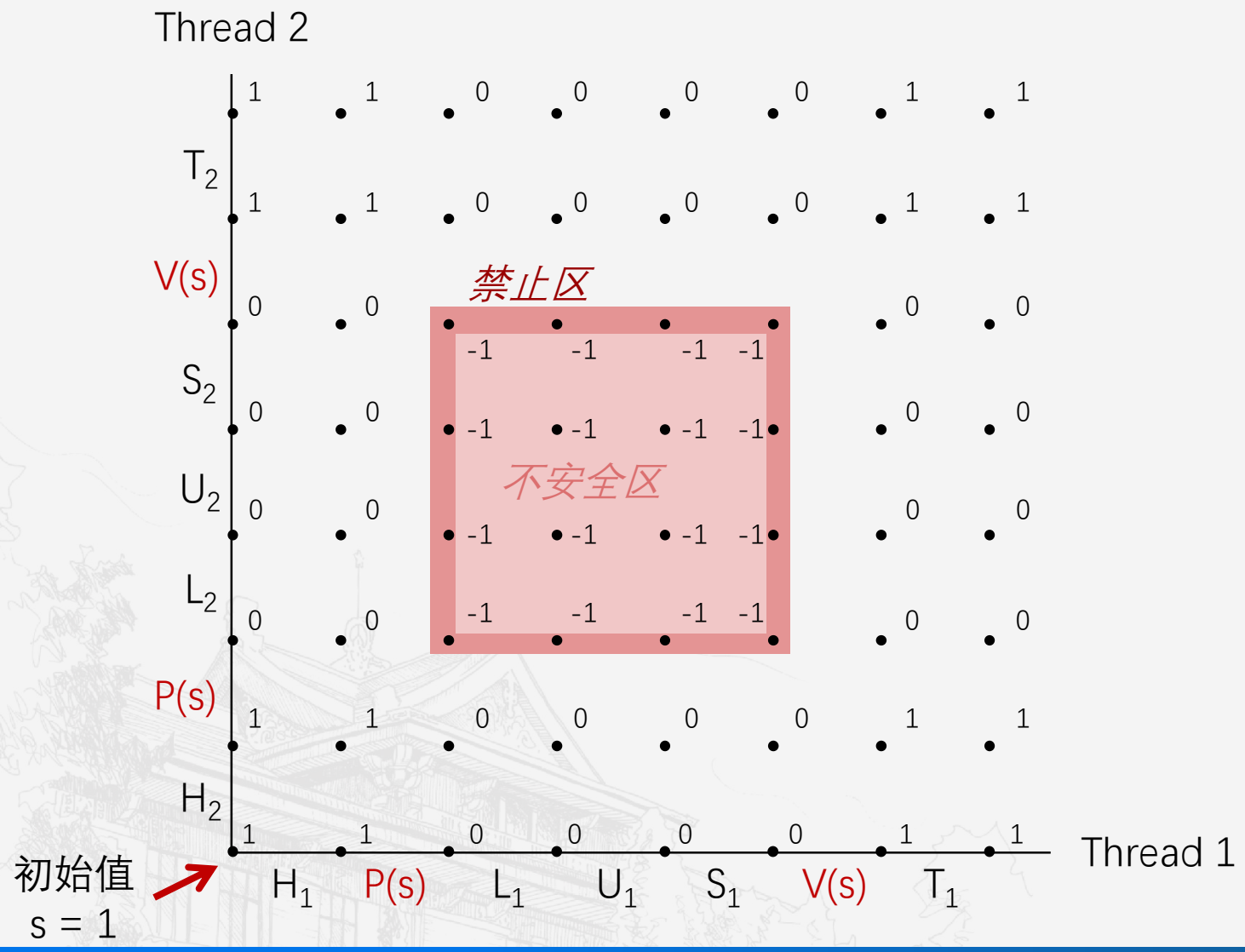
```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

与bancnt.c的代码相比，性能差几个数量级

同步

Synchronization

为什么互斥锁可以有效工作



- 通过在临界区周围使用信号量 s 的 P 和 V 操作来提供对共享变量的互斥访问（初始设置为 1）。
- 利用信号量不变性创造了一个禁止区域，该区域包围了不安全的区域，并且任何轨迹都无法进入该禁止区域。

总结

- 程序员需要一个清晰的模型来理解变量如何被线程共享
- 被多个线程共享的变量必须受保护，以确保互斥访问
- 信号量是强制互斥的基本机制