Synchronization: Basics
同步基础

100076202： 计算机系统导论

任课教师：
计卫星　　宿红毅　　张艳

原作者：
Randal E. Bryant and David R. O'Hallaron

多线程C程序中的共享变量/Shared Variables in Threaded C Programs

- **问题：C程序中的哪个变量是共享的？/Question: Which variables in a threaded C program are shared?**
  - 答案不像“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：*只有多个线程引用一个变量x的某个实例时x才是共享的*/Def:* A variable x is *shared* if and only if multiple threads reference some instance of x.**
- **需要回答以下问题/Requires answers to the following questions:**
  - 线程使用的内存模型是什么？/What is the memory model for threads?
  - 变量实例是如何映射到内存的？/How are instances of variables mapped to memory?
  - 多少个线程会引用这些实例？/How many threads might reference each of these instances?

线程内存模型/Threads Memory Model

- **概念模型/Conceptual model:**
  - 多个线程在一个进程上下文下运行/Multiple threads run within the context of a single process
  - 每个线程有自己独立的线程上下文/Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - 所有线程共享其余的进程上下文/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
- **实际操作上并没有严格强制使用该模型/Operationally, this model is not strictly enforced:**
  - 寄存器值是真正分离和保护的，但是/Register values are truly separate and protected, but…
  - 任何线程可以读写其他线程的堆栈/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很多问题产生的根源/The mismatch between the conceptual and operation model is a source of confusion and errors

关于共享的示例程序/Example Program to Illustrate Sharing

```
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

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

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

其他线程通过全局ptr变量间接引用主线程栈/Peer threads reference main thread's stack indirectly through global ptr variable

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

- 全局变量/Global variables
  - *定义/Def:* 声明在函数外面的变量/Variable declared outside of a function
  - **全局变量在虚拟内存中只有一个实例/Virtual memory contains exactly one instance of any global variable**
- 局部变量/Local variables
  - *定义/Def:* 在函数内声明的非static变量/Variable declared inside function without `static` attribute
  - **每个线程栈有一个局部变量实例/Each thread stack contains one instance of each local variable**
- 局部静态变量/Local static variables
  - *定义/Def:* 在函数内声明的static变量/Variable declared inside function with the `static` attribute
  - **虚拟内存中只有一个局部静态变量实例/Virtual memory contains exactly one instance of any local static variable.**

将变量实例映射到内存/Mapping Variable Instances to Memory

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

Local vars: 1 instance (`i.m`, `msgs.m`)

```
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]
)

```
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])

共享变量分析/Shared Variable Analysis

- 哪些是共享变量？/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：一个变量x至少有一个实例被多个线程引用就是共享的/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

线程同步/Synchronizing Threads

- **共享变量很容易/Shared variables are handy...**
- **但是可能会引入很多同步错误/...but introduce the possibility of nasty *synchronization* errors.**

badcnt.c: Improper Synchronization/不正确地同步

```
/* 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;
}
```

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

cnt should equal 20,000.

What went wrong?

循环计数的汇编代码/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

并发执行/Concurrent Execution

- *关键点/Key idea:* 任意符合顺序一致性的交叉执行是可能的，但是有一些得到意外的结果/In **general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i 表示线程i在执行指令I / I_i denotes that thread i executes instruction I
  - %rdx$_i$是%rdx在线程i上下文下的内容/ %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

🟩	**Thread 1 临界区/critical section**
🟪	**Thread 2 临界区/critical section**

并发执行/ Concurrent Execution (cont)

- 不正确的顺序：两个线程增加计数器，但是结果是1而不是2/Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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!

并发执行/ Concurrent Execution (cont)

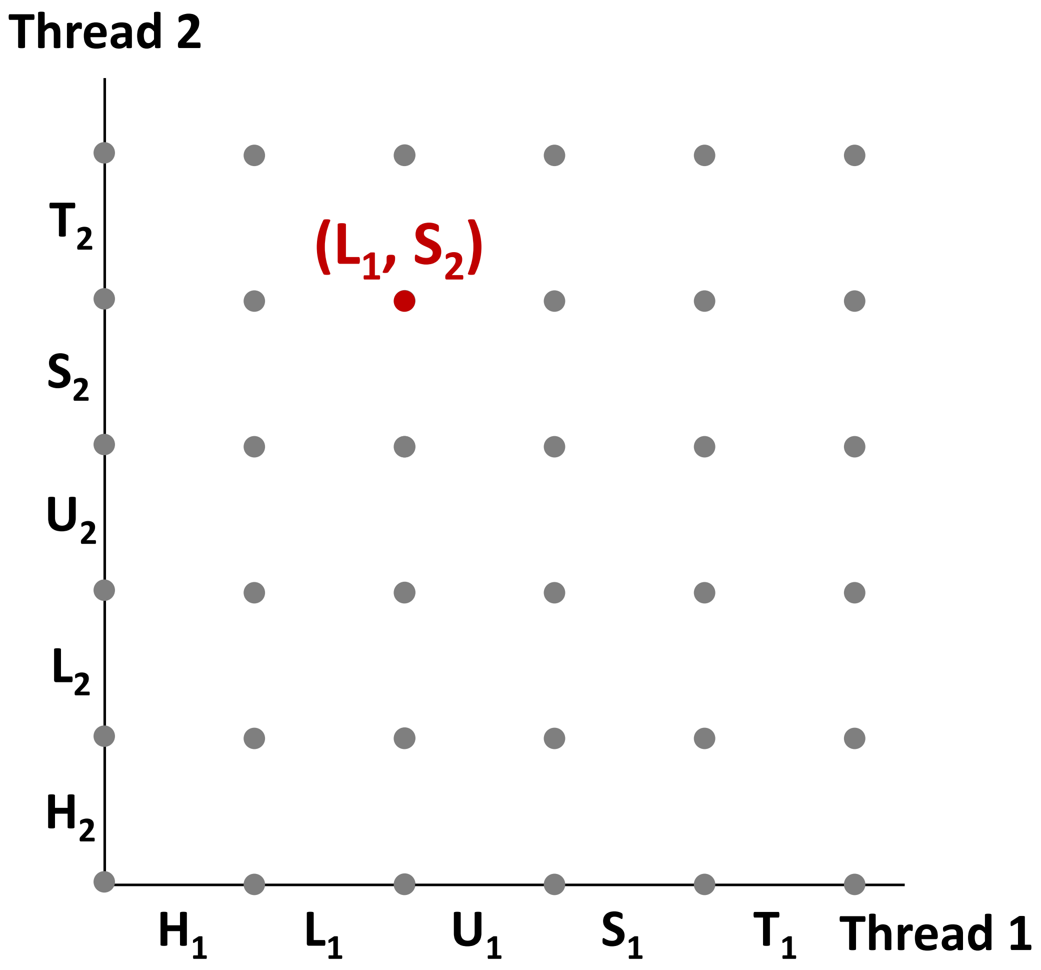
- 变为这个顺序呢？/How about this ordering?

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

Oops!

- 我们可以用进度图来分析其行为/We can analyze the behavior using a *progress graph*

进度图/Progress Graphs



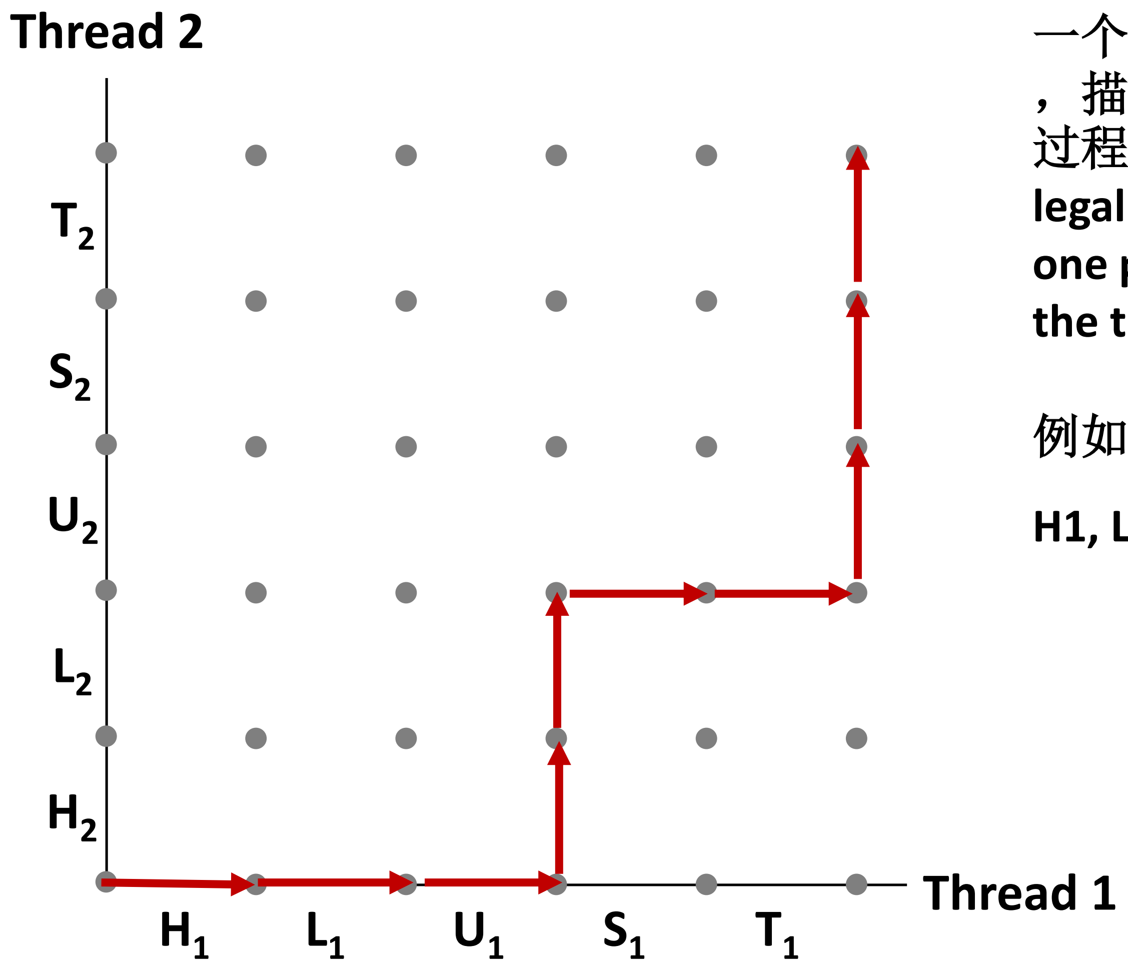
一个进度图描述了并发线程离散的执行状态空间/A *progress graph* depicts the discrete *execution state space* of concurrent threads.

每个轴对应每个线程中指令的执行顺序/Each axis corresponds to the sequential order of instructions in a thread.

每个点对应一个可能的执行状态（$Inst_1$, $Inst_2$ ）/ Each point corresponds to a possible *execution state* ($Inst_1$, $Inst_2$).

例如， (L_1, S_2) 表示线程已经完成了L_1 ，线程2已经完成了S_2 /E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

进度图轨迹/Trajectories in Progress Graphs

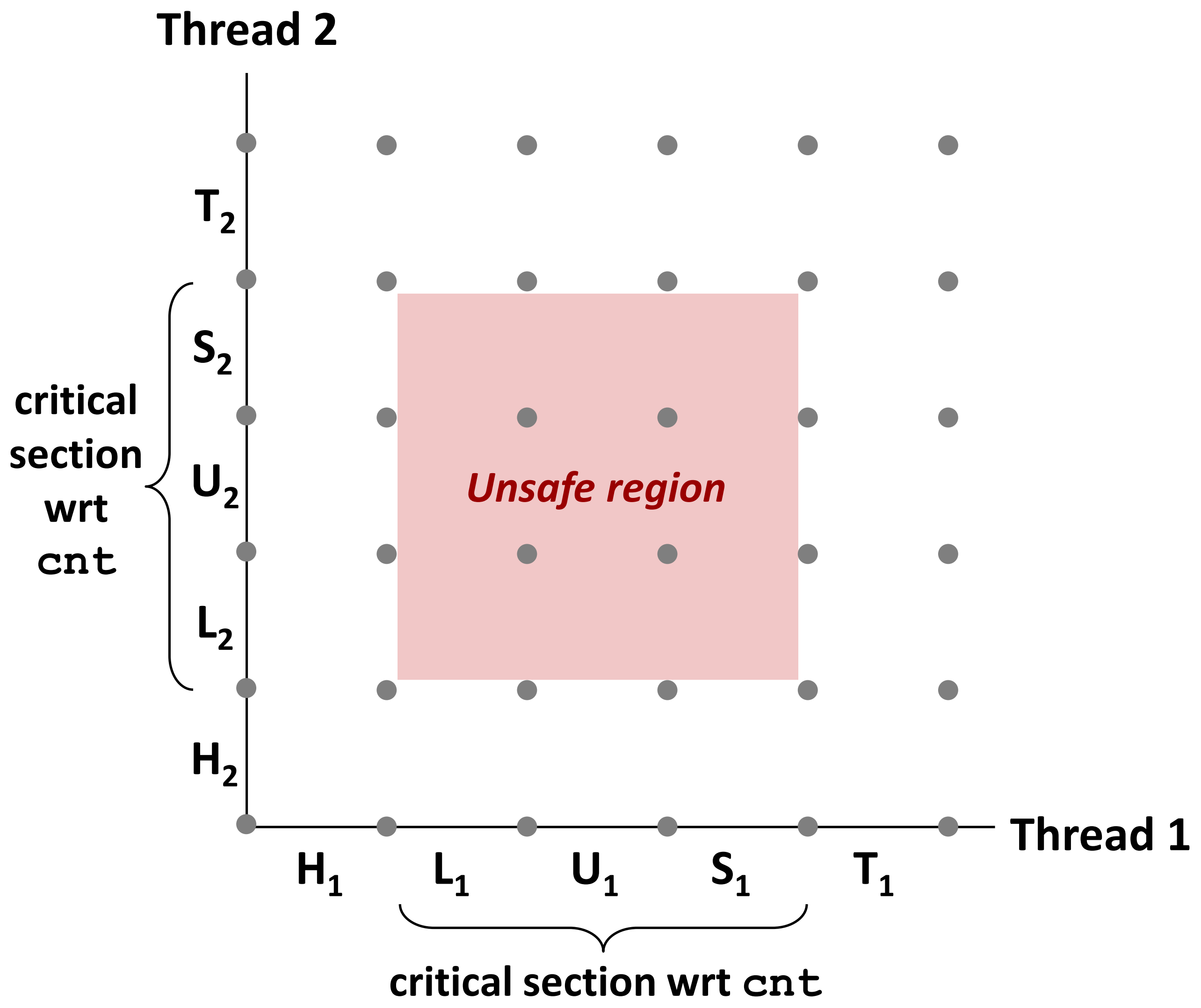


一个轨迹是一系列合法的状态转换，描述了线程一个可能的并发执行过程/A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

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

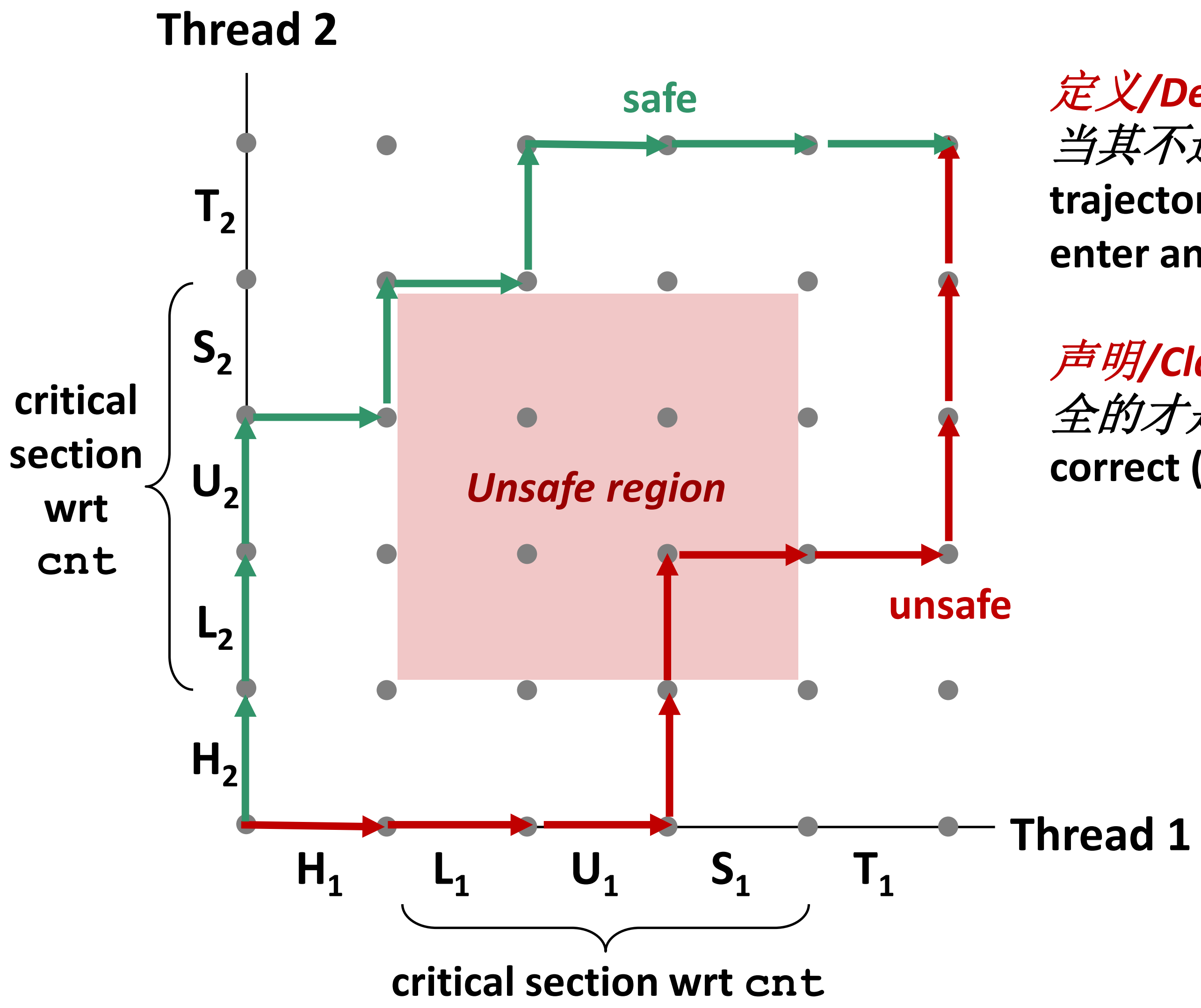


L、U和S形成一个关于共享变量cnt的临界区/L, U, and S form a *critical section* with respect to the shared variable `cnt`

临界区中的指令（依据某些共享变量）不应该交叉执行/Instructions in critical sections (wrt some shared variable) should not be interleaved

那些可能出现这种交叉的状态集就是不安全区域/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/Def: 一个轨迹是安全的只有当其不进入任何不安全区域/A trajectory is *safe* iff it does not enter any unsafe region

*声明/Claim: 只有当一个轨迹是安全的才是正确的*A trajectory is correct (wrt `cnt`) iff it is safe

强制互斥/Enforcing Mutual Exclusion

- *问题：我们如何保证一个安全的轨迹？/Question:* **How can we guarantee a safe trajectory?**

- 答案：我们必须同步线程执行不让其出现不安全的轨迹/**Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.**
  - 例如，需要每个临界区的互斥访问/i.e., need to guarantee ***mutually exclusive access*** for each critical section.
- 经典方案/**Classic solution:**
  - 信号量/Semaphores (Edsger Dijkstra)

- 其他方案（不在这里讨论）**Other approaches (out of our scope)**
  - 互斥和条件变量（Pthreads）/Mutex and condition variables (Pthreads)
  - Monitors (Java)

信号量/Semaphores

- *信号量：/Semaphore:* **非负全局整型同步变量，由P和V操作/non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s)**
  - 如果s非0，则减1之后立即返回/If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - 测试和自减操作原子性完成/Test and decrement operations occur atomically (indivisibly)
  - 如果s是0，则挂起线程直到s变为非0并由v操作重启动/If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - 重启动后，P操作自减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):***
  - 自增1/Increment *s* by 1.
    - 自增操作原子性完成/Increment operation occurs atomically
  - 如果有任意线程在P操作中阻塞等待s变为非0，则重启其中一个线程，完成对应的p操作并递减s/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*.

- **信号量不变式/Semaphore invariant: *(s >= 0)***

信号量操作/C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

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

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

CS:APP wrapper functions:

```
#include "csapp.h”

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

badcnt.c: Improper Synchronization/不正确同步

```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;
}
```

如何使用信号量修复？
/How can we fix this using semaphores?

使用信号量实现互斥/Using Semaphores for Mutual Exclusion

- **基本思路/Basic idea:**
  - 对每个共享变量关联一个信号量，初识值为1/Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
  - 在临界区周围使用P操作和V操作/Surround corresponding critical sections with *P(mutex)* and *V(mutex)* operations.
- **术语/Terminology:**
  - *二值信号量/Binary semaphore*: 信号量的值为0或者1/semaphore whose value is always 0 or 1
  - *互斥/Mutex:* 二值信号量用于互斥/binary semaphore used for mutual exclusion
    - P操作：对mutex加锁/P operation: “locking” the mutex
    - V操作：对mutex解锁或者释放/V operation: “unlocking” or “releasing” the mutex
    - *持有mutex：已加锁并且没有解锁/*“Holding” a mutex: locked and not yet unlocked.
  - *信号量计数：Counting semaphore*:/对一组可用资源当做计数器使用/ used as a counter for set of available resources.

`goodcnt.c`: Proper Synchronization/不正确同步

- 对共享变量cnt定义和初始化一个mutex/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 */
```

- 使用P和V包围临界区/Surround critical section with *P* and *V*:

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

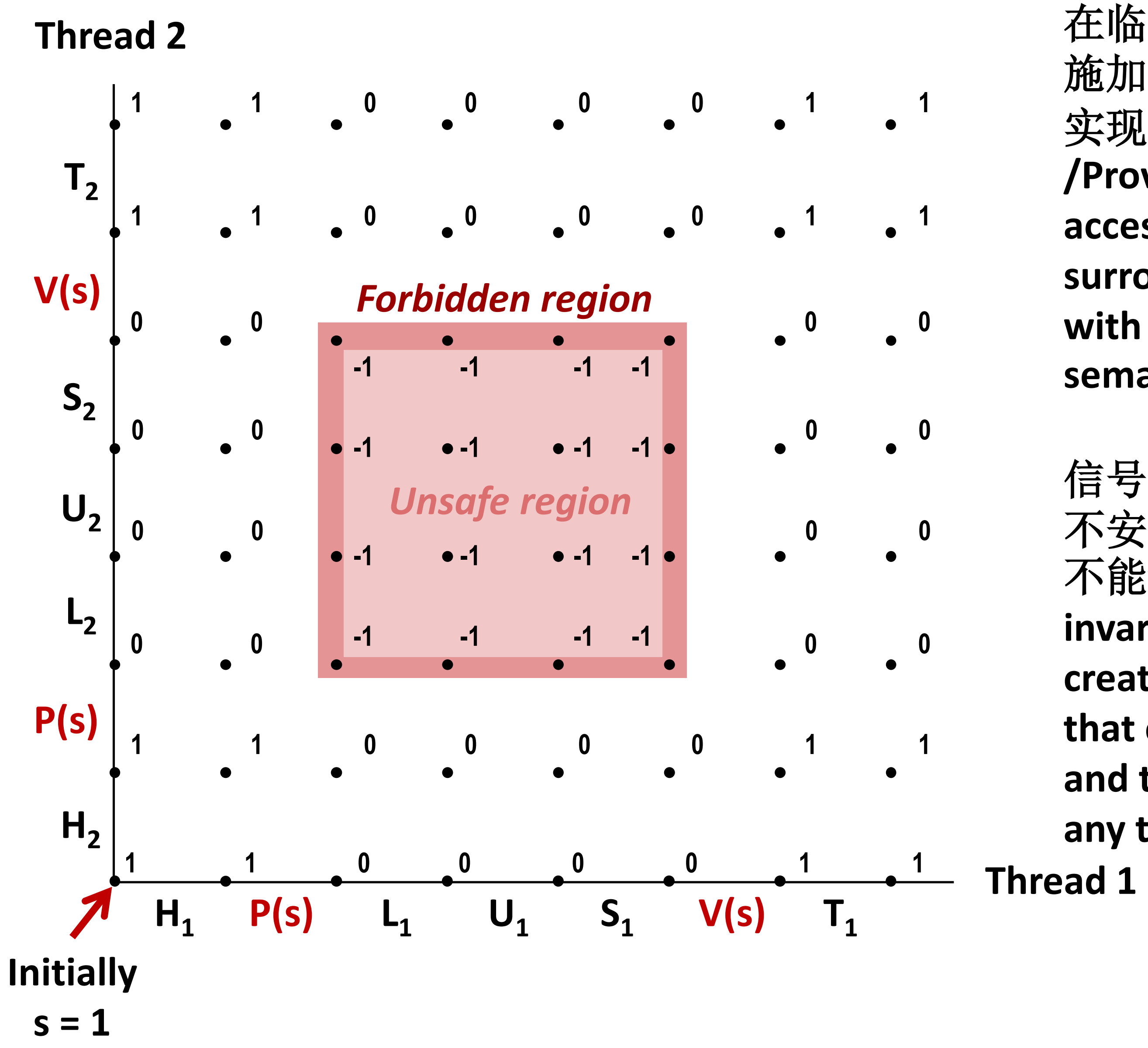
goodcnt.c

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

Warning: It's orders of magnitude slower than `badcnt.c`.

警告：可能会比`badcnt.c`慢一个数量级

为什么Mutexes 可以正确工作/Why Mutexes Work



在临界区周围使用P和V操作施加在初始化为1的信号量上实现对共享变量的互斥访问/Provide mutually exclusive access to shared variable by surrounding critical section with *P* and *V* operations on semaphore s (initially set to 1)

信号量不变式创建了一个由不安全区域包围且任何轨迹不能进入的禁区/Semaphore invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

总结/Summary

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

- 多个线程共享的变量需要保护以实现互斥访问 /Variables shared by multiple threads must be protected to ensure mutually exclusive access.

- 信号量是实现强制互斥的一种重要机制/Semaphores are a fundamental mechanism for enforcing mutual exclusion.