哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第三章 并发与同步：信号量

夏文 副教授
哈尔滨工业大学（深圳）
2021年秋季

Email：xiawen@hit.edu.cn

本章主要内容

- 信号的介绍 （Introduction to Signals）
- 同步（Synchronization）
- 并发编程 （ Concurrent Programming ）

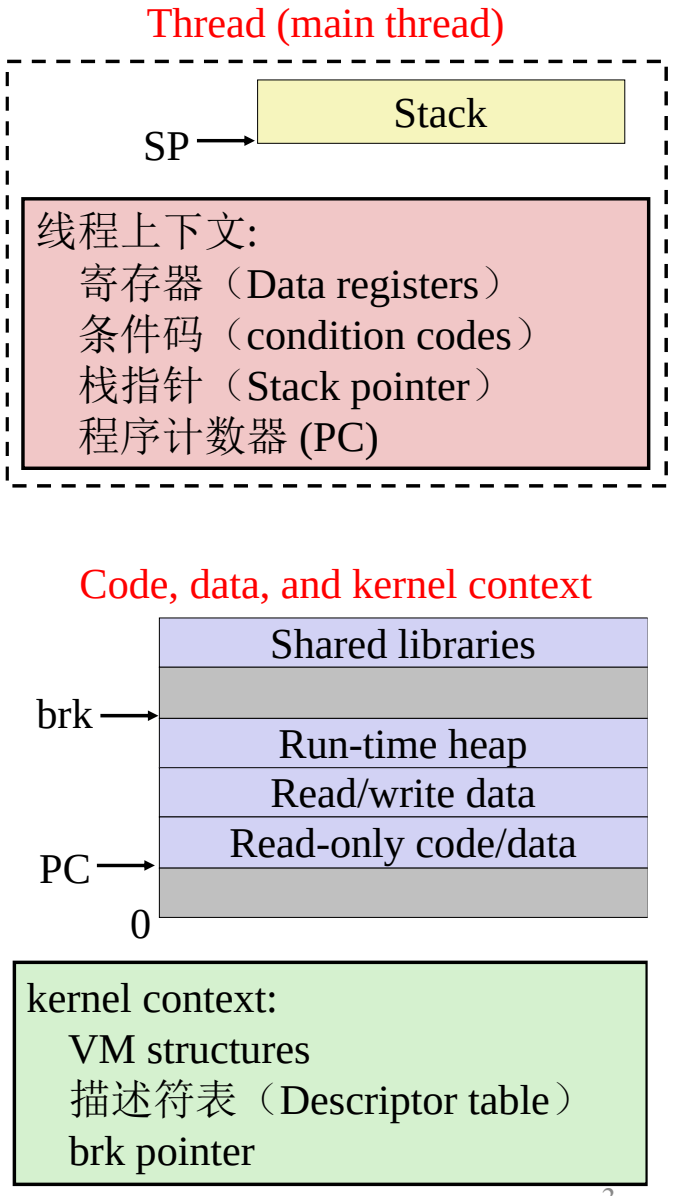
线程化C语言程序中的共享变量

- 问题：线程化的C程序中哪些变量是共享的？
  - 问题的答案不是像“全局变量是共享的，栈变量是私有的”那么简单。
- 定义：一个变量x是共享的，当且仅当它的一个实例被一个以上的线程引用。

- 我们需要解答以下几个问题：
  - 线程的基础内存模型是什么？
  - 变量实例是如何映射到内存的？
  - 有多少线程引用这些实例？

线程内存模型

- 概念模型：
  - 一组并发线程运行在一个进程的上下文中
  - 每个线程都有它独立的线程上下文，包括线程ID、栈、栈指针、程序计数器、条件码和通用目的寄存器值。
  - 所有线程共享进程上下文的剩余部分
    - ✓代码、数据、堆以及共享库代码和数据区域
    - ✓打开文件和已安装的信号处理程序
- 从实际操作的角度来说，这个模型不是那么严格的：
  - 寄存器值其实是不被共享的，但虚拟内存总是共享的
  - 一个线程可以读写另一个线程的栈
- 概念模型和实际操作模型之间的区别是困惑和错误的来源之一



示例


```
6. char **ptr;  /* global var */

33.int main()
34.{
35.     long i;
36.     pthread_t tid;
37.     char *msgs[2] = {
38.         "Hello from foo",
39.         "Hello from bar"
40.     };

41.     ptr = msgs;
42.     for (i = 0; i < 2; i++)
43.         pthread_create(&tid,
44.             NULL,
45.             thread,
46.             (void *)i);
47.     pthread_exit(NULL);
48.}
```

sharing.c

```
7. void *thread(void *vargp)
8. {
9.      long myid = (long)vargp;
10.     static int cnt = 0;

11.     printf("[%ld]:  %s (cnt=%d)\n",
12.          myid, ptr[myid], ++cnt);
13.     return NULL;
14.}
```

对等线程通过全局变量ptr间接引用主线程的栈内容

将变量映射到内存

- 全局变量
  - Def：定义在函数之外的变量
  - 虚拟内存只包含每个全局变量的一个实例
- 本地自动变量
  - Def: 定义在函数内部但没有static属性的变量
  - 每个线程都包含它自己的所有本地自动变量的实例
- 本地静态变量
  - Def: 定义在函数内部并且有static属性的变量
  - 虚拟内存只包含一个本地静态变量的实例

将变量映射到内存

全局变量：1 个实例 (ptr [data])

```
6. char **ptr;  /* global var */

33.int main()
34.{
35.     long i;
36.     pthread_t tid;
37.     char *msgs[2] = {
38.         "Hello from foo",
39.         "Hello from bar"
40.     };
41.     ptr = msgs;
42.     for (i = 0; i < 2; i++)
43.         pthread_create(&tid,
44.             NULL,
45.             thread,
46.             (void *)i);
47.     pthread_exit(NULL);
48.}
```

sharing.c

本地自动变量：1 个实例
(i.m, msgs.m)

本地自动变量：2 个实例
(myid.p0 [对等线程0的栈],
myid.p1 [对等线程1的栈])

```
7. void *thread(void *vargp)
8. {
9.     long myid = (long)vargp;
10.    static int cnt = 0;

11.    printf("[%ld]:  %s (cnt=%d)\n",
12.          myid, ptr[myid], ++cnt);
13.    return NULL;
14.}
```

本地静态变量：1 个实例 (cnt [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P0引用?	被对等线程P1引用?	说明
ptr	✔	✔	✔	由主线程写入并由对等线程读取的全局变量。
cnt	✘	✔	✔	一个静态变量，在内存中只有一个实例，由两个对等线程读取和写入。
i.m	✔	✘	✘	存储在主线程堆栈中的局部自动变量
msgs.m	✔	✔	✔	一个局部自动变量，存储在主线程的堆栈上，两个对等线程通过ptr间接引用。
myid.p0	✘	✔	✘	局部自动变量的实例位于对等线程0的堆栈上
myid.p1	✘	✘	✔	局部自动变量的实例位于对等线程1的堆栈上

- Answer：一个变量*x*是共享的，当且仅当它的一个实例被一个以上的线程引用。因此：
  - ptr, cnt, 和 msgs 是共享的
  - i 和 myid 不是共享的

badcnt.c: 不正确的同步


```
7. /* 全局共享变量 */
8. volatile long cnt = 0; /* Counter
17.int main(int argc, char **argv){
18.     long niters;
19.     pthread_t tid1, tid2;
20.     niters = atoi(argv[1]);
21.     pthread_create(&tid1, NULL,
22.         thread, &niters);
23.     pthread_create(&tid2, NULL,
24.         thread, &niters);
25.     pthread_join(tid1, NULL);
26.     pthread_join(tid2, NULL);
27.     /* Check result */
28.     if (cnt != (2 * niters))
29.         printf("BOOM! cnt=%ld\n", cnt);
30.     else
31.         printf("OK cnt=%ld\n", cnt);
32.     exit(0);
33.}
```

线程并发执行的问题

```
38./* Thread routine */
39.void *thread(void *vargp)
40.{
41.     long i, niters = *((long *)vargp);
42.
43.     for (i = 0; i < niters; i++)
44.         cnt++;
45.
46.     return NULL;
47.}
```

```
[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
```

badcnt.c

计数器循环的汇编代码

- 编译：
  - gcc –s badcnt.c –o badcnt.s
  - vim badcnt.s

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

H_i : Head

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

T_i : Tail

```
 94          movq    %rdi, -24(%rbp)
 95          movq    -24(%rbp), %rax
 96          movq    (%rax), %rax
 97          movq    %rax, -8(%rbp)
 98          movq    $0, -16(%rbp)
 99          jmp     .L6
100 .L7:
101          movq    cnt(%rip), %rax
102          addq    $1, %rax
103          movq    %rax, cnt(%rip)
104          addq    $1, -16(%rbp)
105 .L6:
106          movq    -16(%rbp), %rax
107          cmpq    -8(%rbp), %rax
108          jl      .L7
109          movl    $0, %eax
110          popq    %rbp
```

计数器循环的汇编代码

- 汇编：
  - 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

计数器循环的汇编代码

- 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++;
```

```
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 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
```

怎么计算？

H_i

L_i
U_i
S_i

T_i

并发执行

- Key idea: 一般而言，没有办法预测操作系统是否将为线程选择一个正确的指令顺序
  - I_i 表示线程*i*执行指令I
  - $\%rdx_i$ 是%rdx在线程*i*的上下文中的值

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

OK

并发执行(cont)

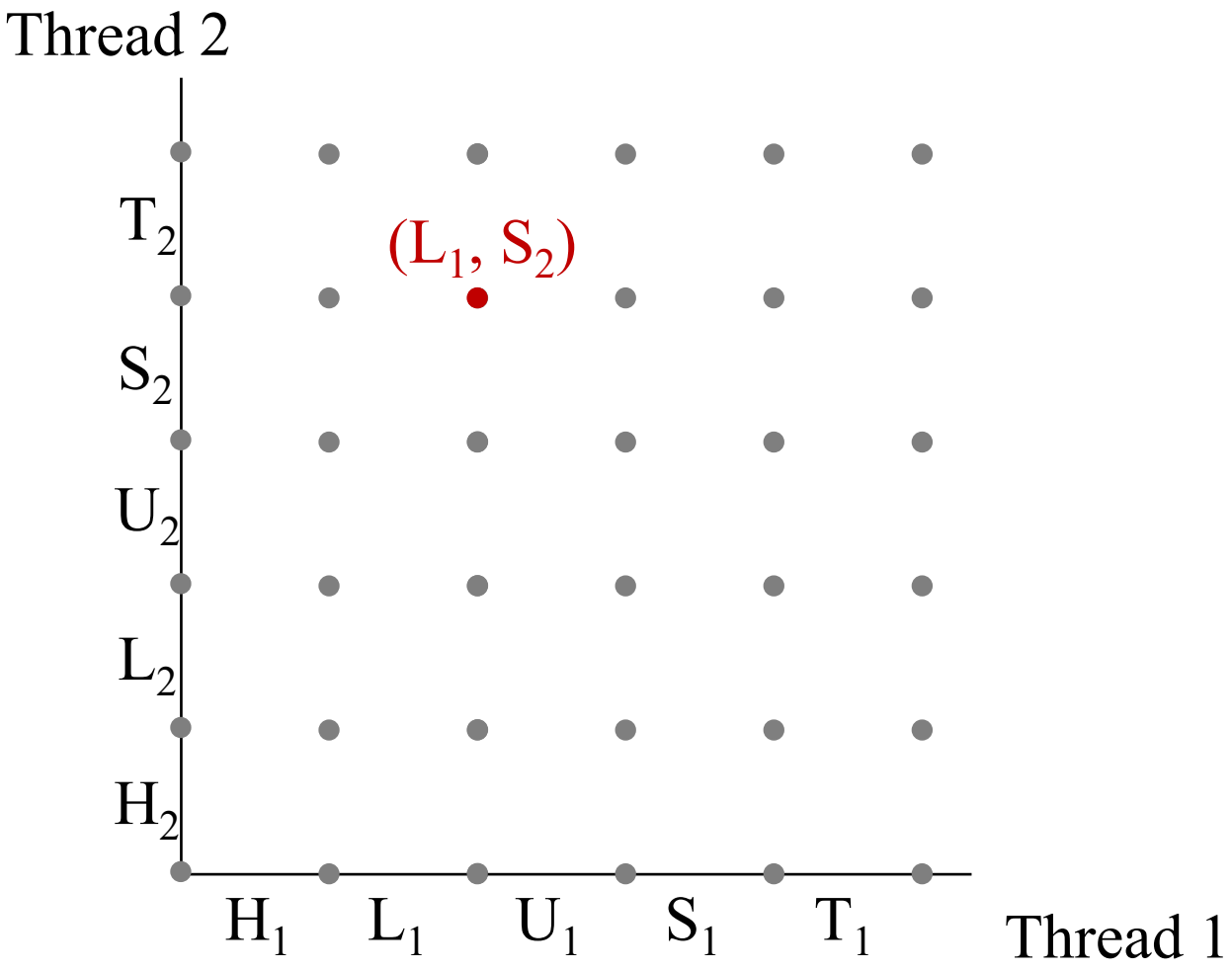
- 错误的顺序：两个线程都更新了counter，但最终结果是1而不是2

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

S1应该在L2之前执行

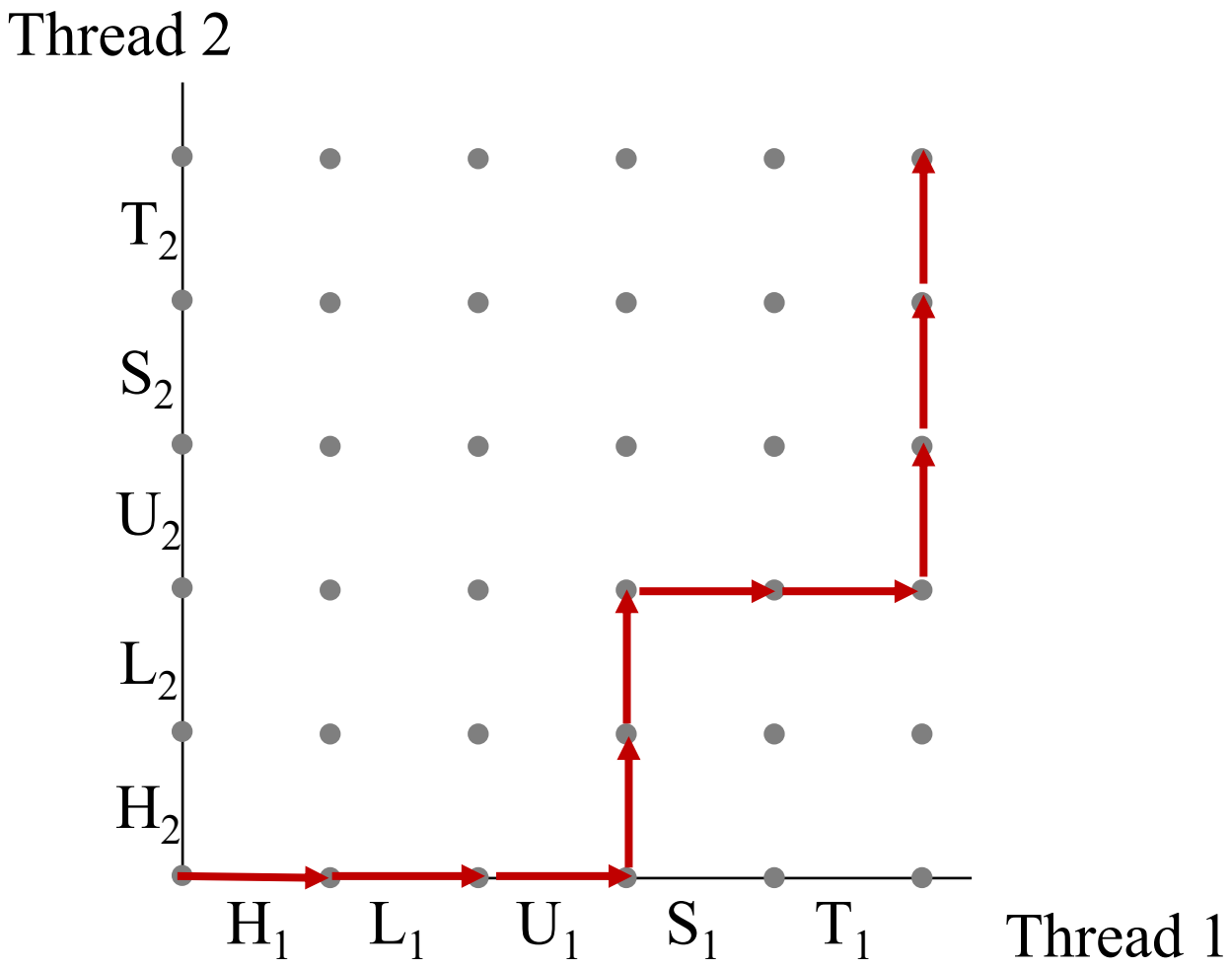
Oops!

Progress Graphs（进度图）



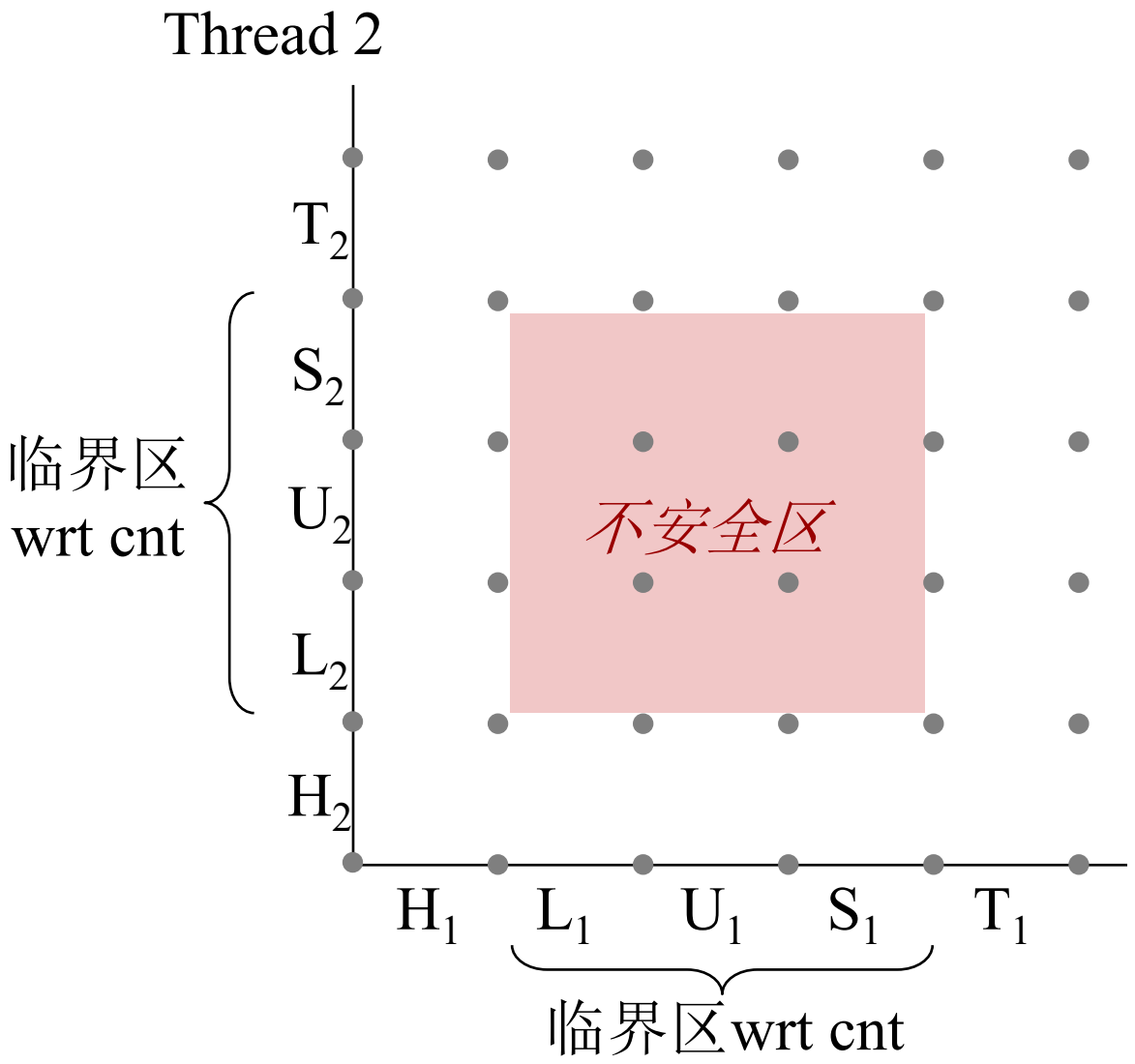
- 一个进度图描述的是并发线程的离散执行状态空间
- 每个坐标轴对应了一个线程的指令执行顺序
- 每个点对应一个可能的执行状态(Inst1, Inst2)
- 例如：(L1, S2) 表示线程1完成了L1，并且线程2 完成了S2

进度图中的轨迹



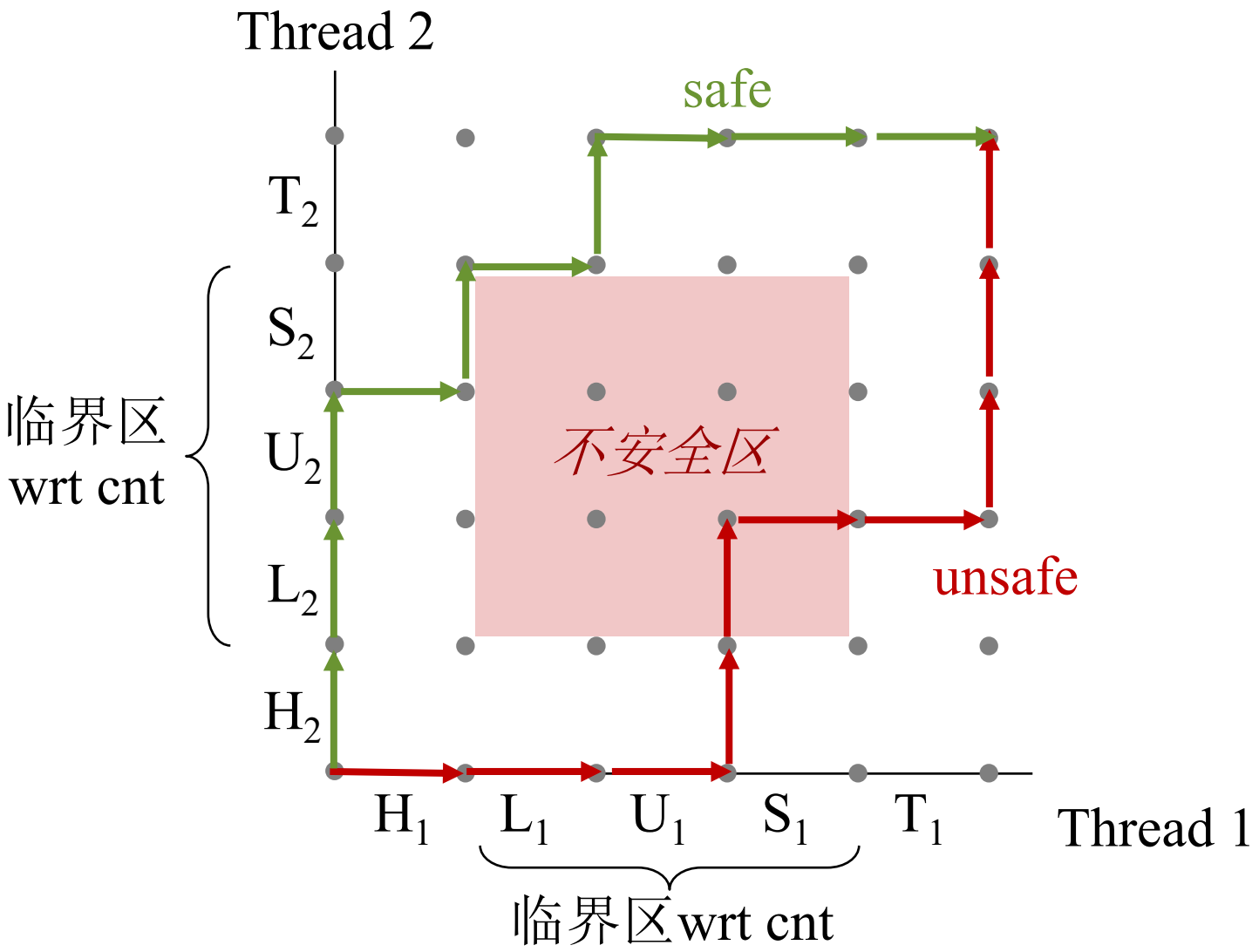
- 轨迹（ trajectory ）是一系列合法的状态转换，它描述了线程的一个可能的并发执行。
- Example:
  - H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

临界区和不安全区



- 操作共享变量cnt的指令L, U, 和 S 组成了一个关于共享变量cnt的临界区（ critical section ）
- 临界区内的指令不应和其他临界区内的指令交替执行
- 两个临界区的交集形成的状态空间称为不安全区（ unsafe regions ）

临界区和不安全区



- Def: 一个轨迹是安全的当且仅当它没有进入不安全区
- Claim: 一个轨迹是正确的当且仅当它是安全的

强制互斥

- Question: 如何确保轨迹正确?
- Answer: 我们必须同步线程的执行，使它们总是有一个安全的轨迹。也就是说，我们要保证互斥地访问每个临界区
- 经典方法:
  - Semaphores（信号量） (Edsger Dijkstra)
- 其他方法(不在我们的讨论范围)
  - 锁和条件变量(Pthreads)
  - Monitors (Java)

Semaphores（信号量）

- 信号量s：具有非负整数值的全局变量，被P和V操作处理
- P(s)
  - 如果s非零，那么P将s减1并立即返回
    - ✓此为原子操作
  - 如果s为零，那么就挂起这个线程，直到s变为非零，等待一个V操作重启这个线程。
  - 重启后，P操作将s减1，将控制权转移给调用者

```
1. P(s):
2.      if s>0 s--;
3.      if s==0 do block;
```

- V(s)
  - 将s加1
    - ✓ 此操作为原子操作
  - 如果有任何线程阻塞在P操作等待s变成非零，那么V操作将会重启一个线程，然后将s减1来完成它的P操作

```
1. V(s):
2.      s++;
3.      if any threads blocked by P
   operation then wakeup
```

C语言信号量操作

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:

```
94.void P(sem_t *sem)
95.{
96.  if (sem_wait(sem) < 0)
97.    unix_error("P error");
98.}                          goodcnt.c
```

```
100.void V(sem_t *sem)
101.{
102.  if (sem_post(sem) < 0)
103.    unix_error("V error");
104.}                          goodcnt.c
```

badcnt.c: 不正确的同步


```
7. /* 全局共享变量 */
8. volatile long cnt = 0; /* Counter */

17.int main(int argc, char **argv)
18.{
19.    long niters;
20.    pthread_t tid1, tid2;
21.    niters = atoi(argv[1]);
22.    pthread_create(&tid1, NULL,
23.        thread, &niters);
24.    pthread_create(&tid2, NULL,
25.        thread, &niters);
26.    pthread_join(tid1, NULL);
27.    pthread_join(tid2, NULL);
28.    /* Check result */
29.    if (cnt != (2 * niters))
30.        printf("BOOM! cnt=%ld\n", cnt);
31.    else
32.        printf("OK cnt=%ld\n", cnt);
33.    exit(0);
34.}
```

badcnt.c

```
38./* Thread routine */
39.void *thread(void *vargp)
40.{
41.    long i, niters = *((long *)vargp);
42.
43.    for (i = 0; i < niters; i++)
44.        cnt++;
45.
46.    return NULL;
47.}
```

如何利用信号量来修改它?

用信号量实现互斥

- 基本思想:
  - 将每个共享变量（或者一组相关的共享变量）与一个信号量（初始为1）联系起来
  - 用P和V操作将相应的临界区包围起来
- 术语:
  - Binary semaphore（二元信号量）: 值总是0或1的信号量
  - Mutex（互斥锁）: 以提供互斥为目的的二元信号量
    - ✓P 操作: 对互斥锁加锁
    - ✓V 操作: 对互斥锁解锁
    - ✓占有互斥锁: 对互斥锁加锁并且还没有解锁
  - Counting semaphore（计数信号量）:一个被用作一组可用资源的计数器的信号量

goodcnt.c: 正确的同步

- 为共享变量cnt定义和初始化一个互斥锁:

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

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

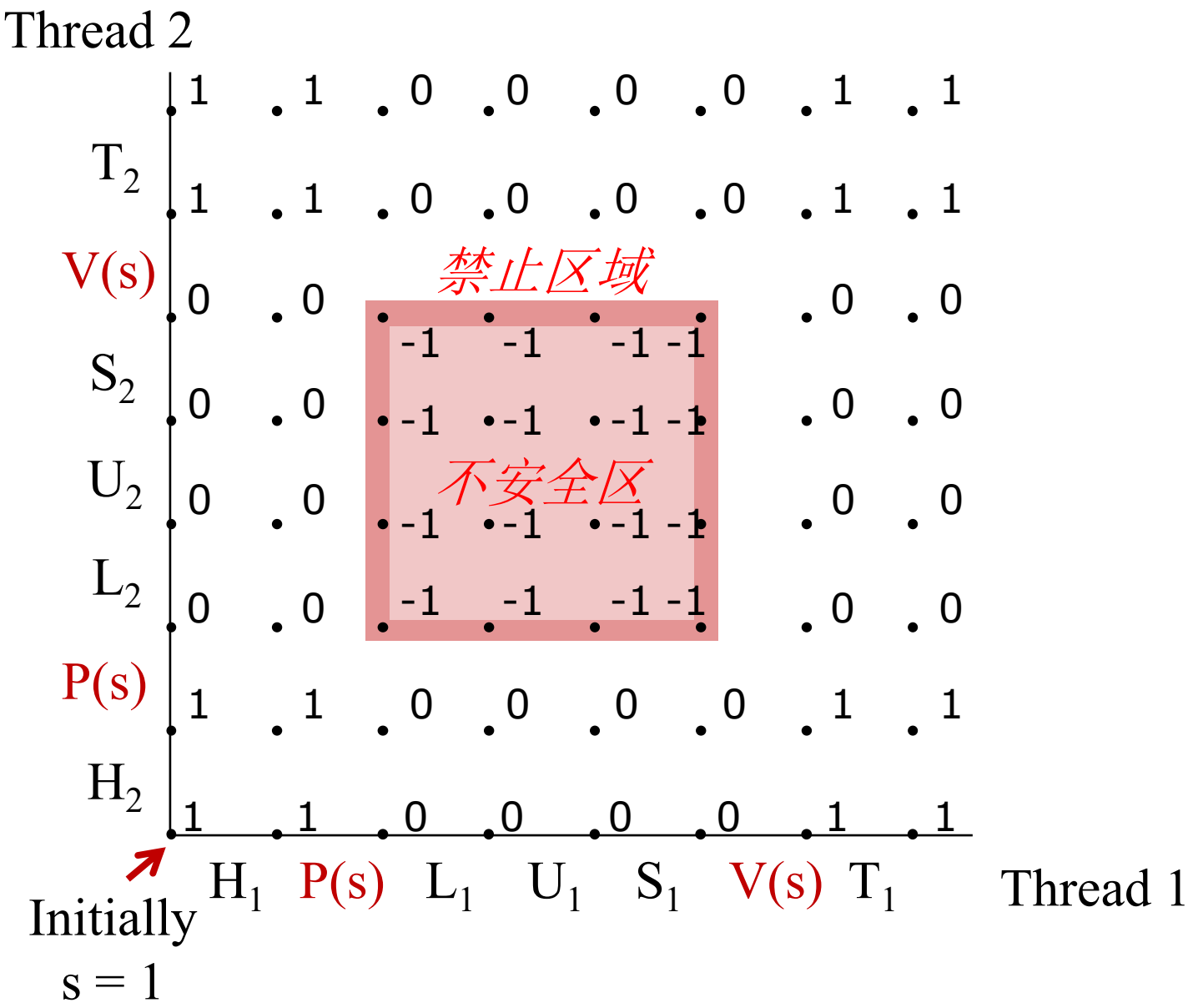
- 用P和V操作将临界区包围:

```
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>
```

为什么互斥锁有效



- 用信号量s（初始化为1）的P、V操作将临界区包围起来，以此来互斥地访问共享变量
- 信号量创建了一个禁止区域（forbidden region），它完全包括了不安全区，因此没有轨迹可以经过不安全区

小结（互斥）

- 需要一个关于线程如何共享变量的清晰模型
- 必须保护多线程共享的变量，以确保互斥访问
- 信号量是执行互斥的基本机制
- 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运行时长约为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]$
```

进程之间的制约关系

- 1、间接制约：
  - （1）临界资源：一次仅允许一个进程使用的资源称为临界资源。
    - ✓硬件：如输入机、打印机、磁带机等
    - ✓软件：如公用变量、数据、表格、队列等
  - （2）互斥：
    - ✓对某个系统资源，一个进程正在使用它，另外一个想用它的进程就必须等待，而不能同时使用；

进程A

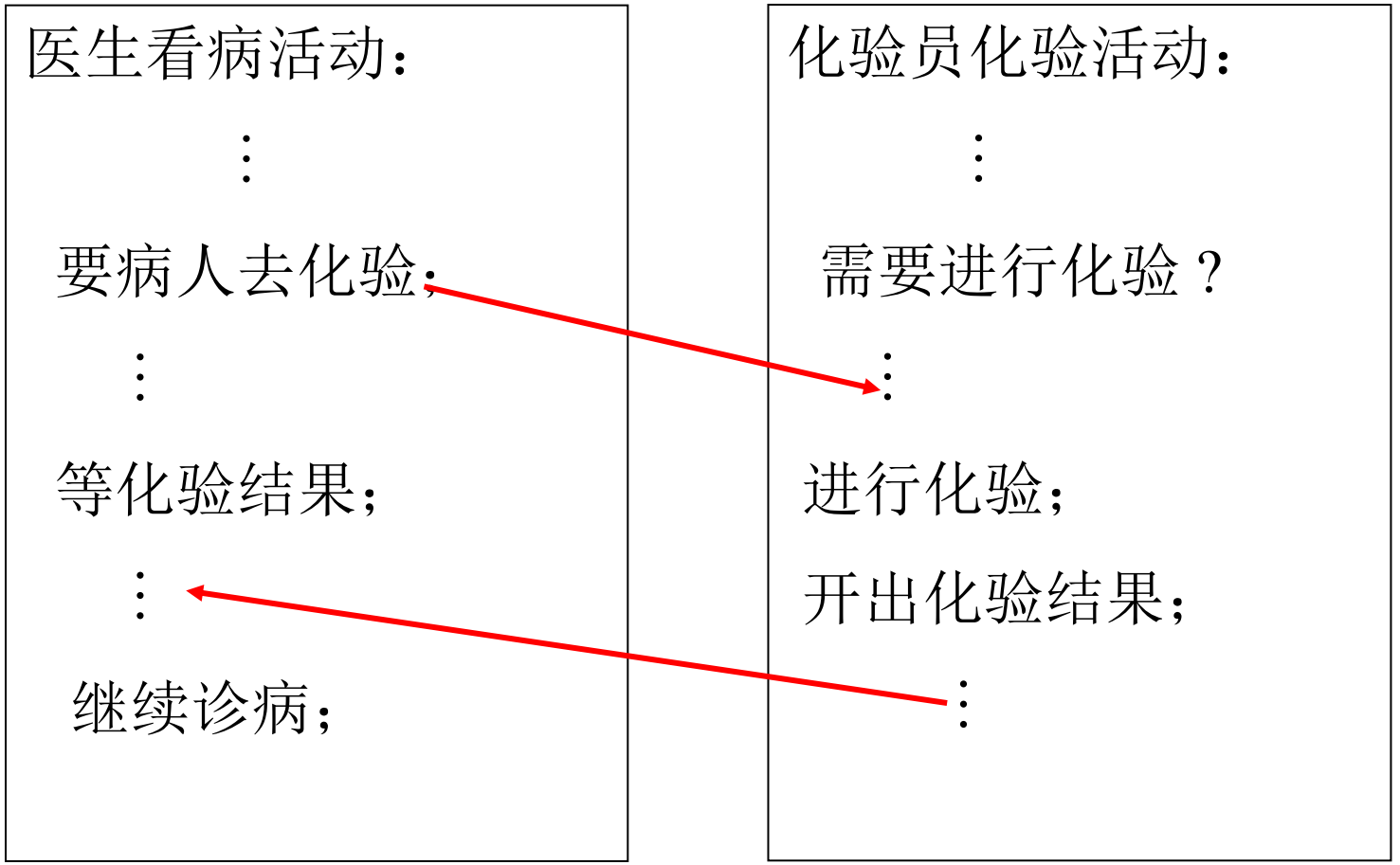
```
⋮
x := x+1;
⋮
```

进程B

```
⋮
x := x+1;
⋮
```

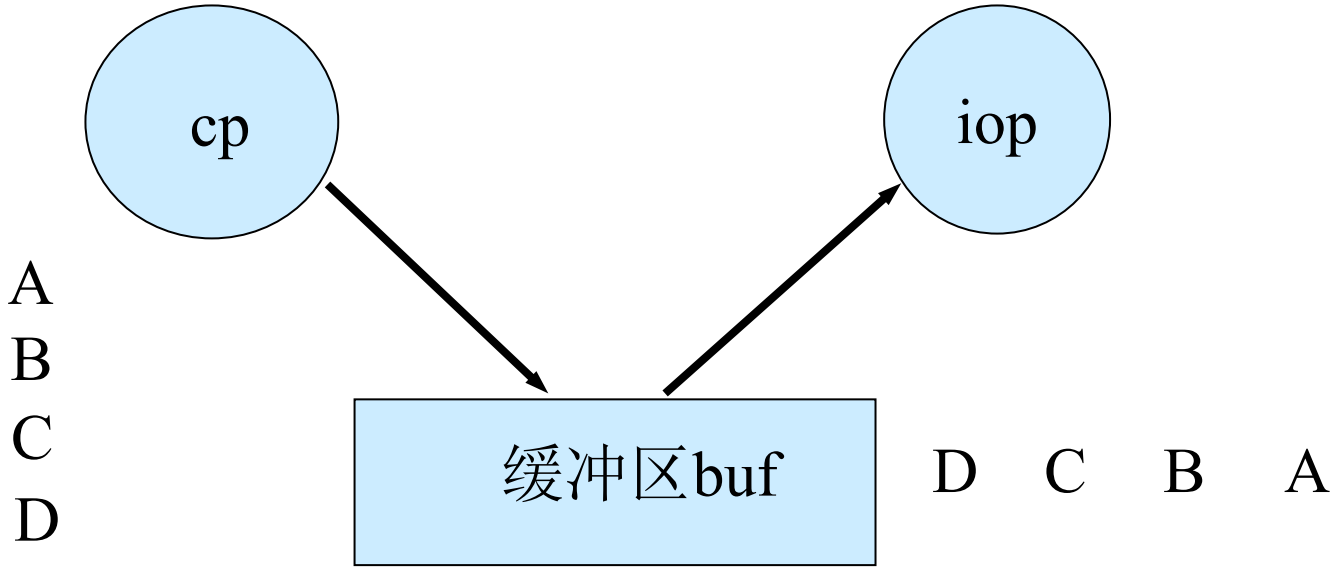
进程之间的制约关系

- 2、直接制约：相互协作，保证先后顺序（同步）
  - 例：病人就诊：



进程之间的制约关系

- 2、直接制约：相互协作 （同步）
  - 共享缓冲区的计算进程与打印进程的相互协作
  - 计算进程 cp和打印进程 iop公用一个单缓冲



关于同步的概念

- 1、概念：
  - 并发进程在一些关键点上可能需要互相等待或互通消息，
  - 这种相互制约的等待或互通消息称为进程同步。
- 2、同步机制应遵循的准则
  - 空闲让进：其他进程均不处于临界区；
  - 忙则等待：已有进程处于其临界区；
  - 有限等待：等待进入临界区的进程不能"死等"；
  - 让权等待：不能进入临界区的进程，应释放CPU（如转换到等待状态）

关于同步的概念

- 3、同步与互斥：
  - 互斥（**间接制约**）是指某一资源同时只允许一个访问者对其进行访问，具有**唯一性和排它性**。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。
  - 同步（**直接制约**）是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的**有序访问**。 少数情况下，同步允许多个访问者同时访问资源。
- 4、信号与信号量
  - 信号：（signal）是一种处理异步事件的方式，类似于软中断。
  - 信号量：（Semaphore）进程间通信处理同步互斥的机制。

利用信号量来调度共享资源(实现同步)

- 基本思想：一个线程用一个信号量来通知另一个线程，某个条件已经为真了
  - 用计数信号量跟踪资源状态和通知其他线程
  - 用互斥锁保护对资源的访问
- 两个经典例子：
  - 生产者-消费者问题（典型同步问题）
  - 读者-写者问题 （典型互斥问题）

生产者-消费者问题

- 同步模式:
  - 生产者等待空槽位（slot），将生成的新项目（item）加入缓冲区，并通知消费者
  - 消费者等待项目，将它从缓冲区移除，并通知生产者

- Examples
  - 多媒体处理：
    - ✓生产者创建MEPG视频帧，消费者渲染它们
  - 事件驱动的图形用户界面
    - ✓生产者检测鼠标点击、鼠标移动和键盘敲击，并将相应的事件插入到缓冲区
    - ✓消费者从缓冲区取走事件并绘制相应的图形界面

N元素缓冲区上的生产者-消费者问题

- 需要一个互斥锁和两个计数信号量:
  - mutex: 确保互斥地访问缓冲区
  - slots: 计数缓冲区中可用的槽位
  - items: 计数缓冲区中可用的元素

- 用一个叫做sbuf的共享缓冲区包来实现

sbuf 包

sbuf.h

```
#include "csapp.h"

typedef struct {
    int *buf;          /* 缓冲区数组 */
    int n;             /* 最大槽位数 */
    int front;         /* buf[(front+1)%n] 是第一个项目 */
    int rear;          /* buf[rear%n] 是最后一个项目 */
    sem_t mutex;       /* 确保互斥地访问缓冲区 */
    sem_t slots;       /* 计数缓冲区中可用的槽位 */
    sem_t items;       /* 计数缓冲区中可用的元素 */
} 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 包 – 实现

- 初始化和释放一个共享缓冲区:

```
/* 创建一个空的，有限的，共享的，先进先出的，有n个槽位的缓冲区 */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                          /* Buffer 最多有n个项目 */
    sp->front = sp->rear = 0;           /* 当且仅当front == rear时buffer为空 */
    Sem_init(&sp->mutex, 0, 1);         /* 二元信号量 */
    Sem_init(&sp->slots, 0, n);         /* 最开始有n个空槽位 */
    Sem_init(&sp->items, 0, 0);         /* 最开始有0个项目 */
}

/* 清理缓冲区 */
void sbuf_deinit(sbuf_t *sp)
{
    free(sp->buf);
}
```

sbuf.c

sbuf 包 – 实现

- 插入一个项目:

```
/* 在缓冲区尾插入一个项目 */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* 等待可用槽位*/
    P(&sp->mutex);                           /* 给buffer加锁*/
    sp->buf[(++sp->rear)%(sp->n)] = item;    /* 插入项目*/
    V(&sp->mutex);                           /* 解锁buffer*/
    V(&sp->items);                          /* 宣告这里有可用的项目
*/
}                                                    sbuf.c
```

sbuf 包 – 实现

- 移除一个项目:

```
/* 移除并返回第一个项目 */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                                  /* 等待可用项目*/
    P(&sp->mutex);                                    /* 给buffer加锁*/
    item = sp->buf[(++sp->front)%(sp->n)];  /* 移除项目 */
    V(&sp->mutex);                                    /* 解锁buffer*/
    V(&sp->slots);                                  /* 宣告这里有可用的槽位 */
    return item;
}
```

sbuf.c

读者-写者问题

- 该类问题属于典型互斥问题
- 问题具体说明:
  - 读者线程只读共享对象
  - 写者线程只修改共享对象
  - 写者必须互斥地访问共享对象
  - 可以有无限多个读者共享这个对象
- 在现实系统中很常见，例如,
  - 在线航空预定系统

读者-写者问题的变种

- 第一类读者-写者问题（读者优先）
  - 要求不让读者等待，除非已经把共享对象的使用权限给了一个写者
  - 一个后来的读者也比前面的写者优先级高
- 第二类读者-写者问题（写者优先）
  - 当一个写者准备好要写，它就会尽可能快地完成写操作
  - 在一个写者后到达的读者必须等待，即使这个写者也在等待
- 这两种情况都有可能会导致饥饿(starvation)现象（即一个线程无限期阻塞）

第一类读者-写者问题的解答

Readers:

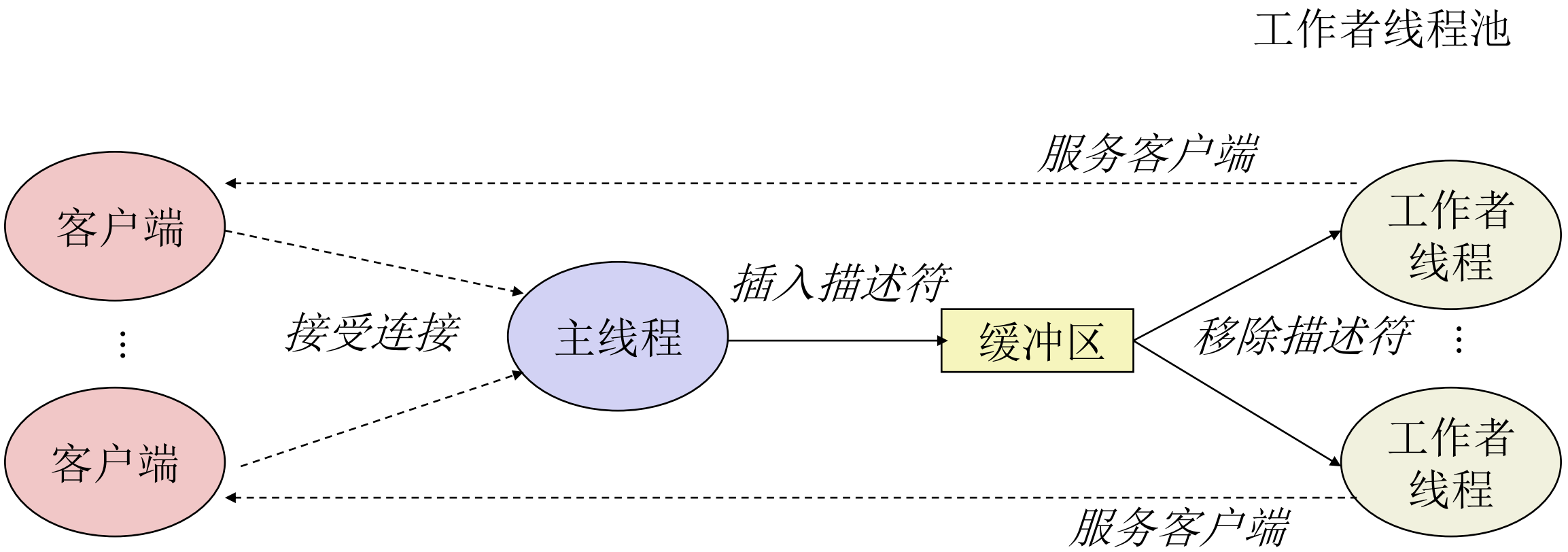
```
int readcnt;     /* Initially = 0 */
sem_t mutex, w; /* Initially = 1 */
void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* 第一个进入 */
            P(&w);
        V(&mutex);

        /* 这里放临界区代码 */
        /* 读 */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* 最后一个离开 */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```
void writer(void)
{
    while (1) {
        P(&w);

        /* 这里放临界区代码 */
        /* 写 */

        V(&w);
    }
}
```

rw1.c

综合：基于预线程化的并发服务器



预线程化的并发服务器

echoservert_pre.c

```
sbuf_t sbuf; /* 描述符的共享缓冲区 */

int main(int argc, char **argv)
{
    int i, listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = open_listenfd(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    for (i = 0; i < NTHREADS; i++)  /* 创建工作者线程 */
      pthread_create(&tid, NULL, thread, NULL);
    while (1) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = accept(listenfd, (SA *) &clientaddr, &clientlen);
      sbuf_insert(&sbuf, connfd); /* 将connfd插入缓冲区 */
    }
}
```

预线程化的并发服务器

- 工作者线程例程:

```
void *thread(void *vargp)
{
    pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf); /* 从缓冲区移除connfd */
        echo_cnt(connfd);                /* 服务客户端 */
        close(connfd);
    }
}
```

echoservert_pre.c

预线程化的并发服务器

- echo_cnt 初始化例程:

```
static int byte_cnt;   /* Byte counter */
static sem_t mutex;    /* byte_cnt的锁 */

static void init_echo_cnt(void)
{
    sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
```

echo_cnt.c

预线程化的并发服务器

- 工作者线程例程:

```
Static int byte_cnt; //记录接收字节数
Static sem_t mutex;
void echo_cnt(int connfd)
{
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;

    pthread_once(&once, init_echo_cnt);
    rio_readinitb(&rio, connfd);
    while((n = rio_readlineb(&rio, buf, MAXLINE)) != 0) {
       P(&mutex);
       byte_cnt += n;
       printf("thread %d received %d (%d total) bytes on fd %d\n",
              (int) pthread_self(), n, byte_cnt, connfd);
       V(&mutex);
       rio_writen(connfd, buf, n);
    }
}
```

echo_cnt.c

Thread Safety (线程安全)

- 线程调用的函数必须是线程安全的
- Def: 一个函数被称为线程安全的，当且仅当被多个并发线程反复调用时，它会一直产生正确的结果
- 四类线程不安全函数:
  - Class 1: 不保护共享变量的函数
  - Class 2: 保持跨越多个调用的状态的函数
  - Class 3: 返回指向静态变量的指针的函数
  - Class 4: 调用线程不安全函数的函数

线程不安全函数(Class 1)

- 不保护共享变量
  - 改正: 用信号量PV操作
  - Example: goodcnt.c
  - 存在问题: 同步操作将减慢程序的执行

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

线程不安全函数(Class 2)

- 依赖于跨多个函数的持久状态
  - Example: 随机数生成器，当前调用的结果依赖于前次调用的中间结果

```
static unsigned int next = 1;
/* rand: 返回取值范围为 0..32767 的伪随机数 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

线程安全的随机数生成器

- 在参数中传递状态信息
  - 以此来消除掉全局状态
  - 传入值不使用共享变量

```
/* rand_r -返回取值范围为 0..32767 的伪随机数 */

int rand_r(int *nextp)
{
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- 缺点：程序员现在还要被迫修改调用程序中的代码

线程不安全函数(Class 3)

- 返回一个指向静态变量的指针，比如ctime()
  - 正在被一个线程使用的结果可能会被另一个线程悄悄地覆盖了
- Fix 1. 重写函数，使得调用者传递存放结果的变量的地址
  - 需要修改调用者和被调用者的源代码
- Fix 2. 加锁-复制（Lock-and-copy）
  - 只需对调用者做一些小的修改，并且不需要修改被调用者
  - 然而，调用者必须要释放分配的内存（privatep）

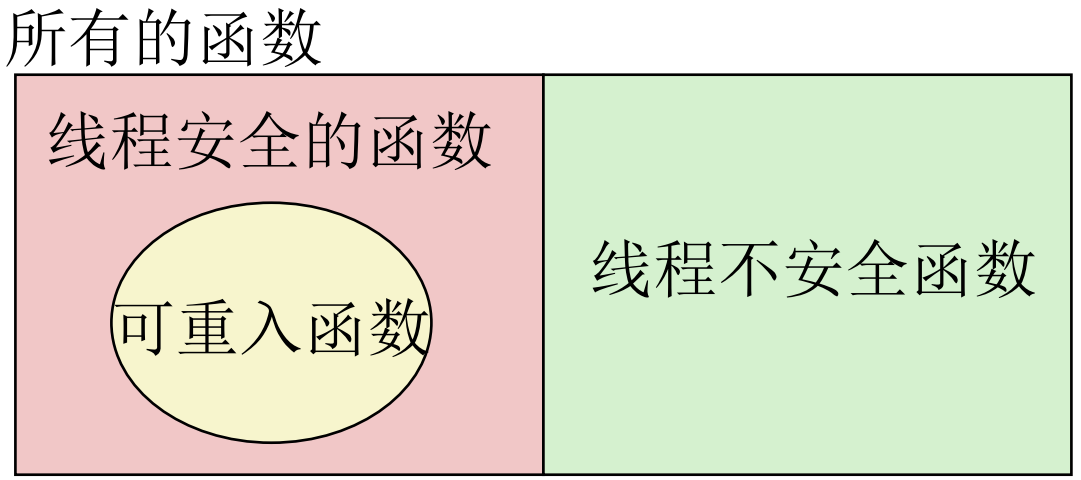
```
/* lock-and-copy version */
char *ctime_ts(const time_t *timep,
               char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

线程不安全函数(Class 4)

- 调用线程不安全函数
  - 调用一个线程不安全函数可能会使这个调用者的整个函数都变得线程不安全
  - Fix: 如果调用的是第2类函数，那么就只能改写这个函数；如果调用的是第1和第3类函数，可以用互斥锁加以保护

可重入函数

- Def: 一个函数是可重入的，当且仅当它被多个线程调用时，不会引入任何共享数据
  - 是线程安全函数的子集
    - ✓不需要同步操作
    - ✓使第2类不安全函数变安全的唯一方法就是将其变为可重入函数



线程安全的库函数

- 大多数Linux函数，包括定义在标准C库中的函数都是线程安全的
  - 例如: malloc, free, printf, scanf
- 只有一小部分是例外，例如:

Thread-unsafe function	Class	Reentrant version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r
rand	2	rand_r

竞争（换一种角度看同步互斥问题）

- 由于两个或多个进程使用不能被同时访问的资源，使得这些进程的结果因为时间上的先后而出现问题，这时就发生了竞争（race）。

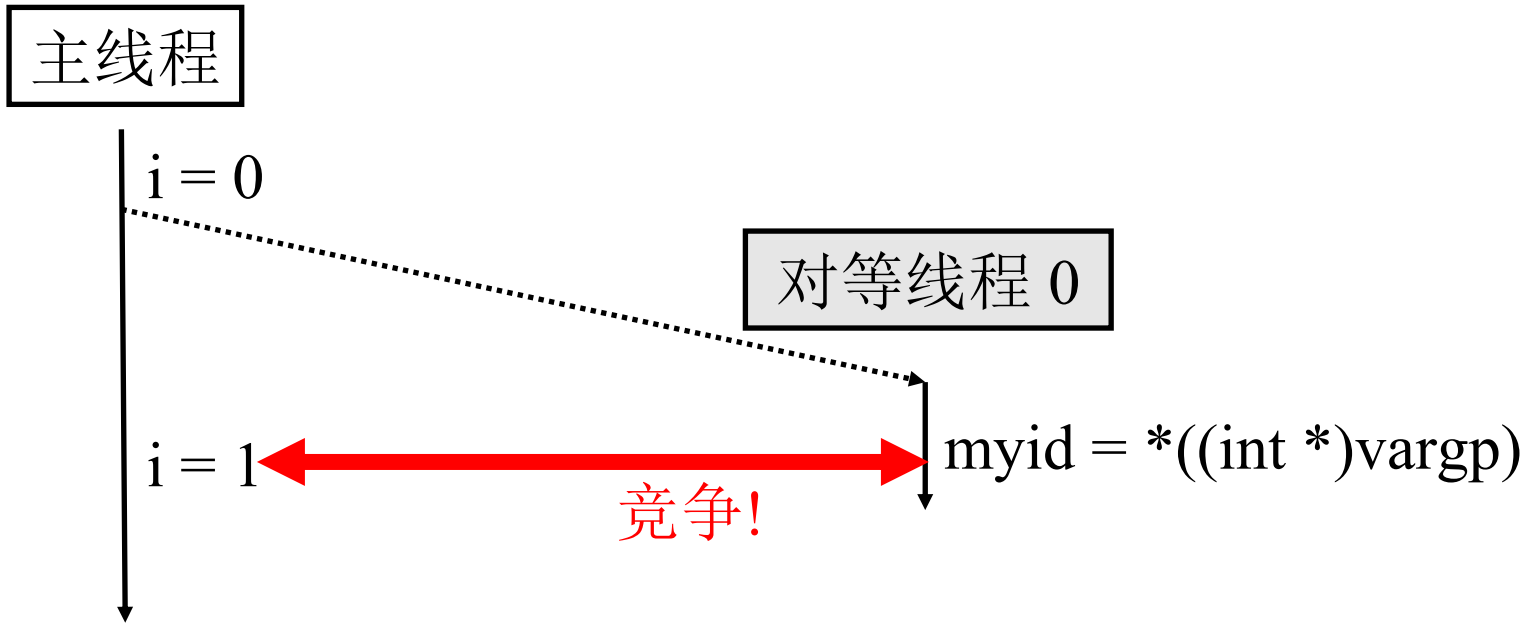
```c
/* A threaded program with a race */
int main()
{
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    exit(0);
}
/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

N个线程正在共享i

race.c

竞争

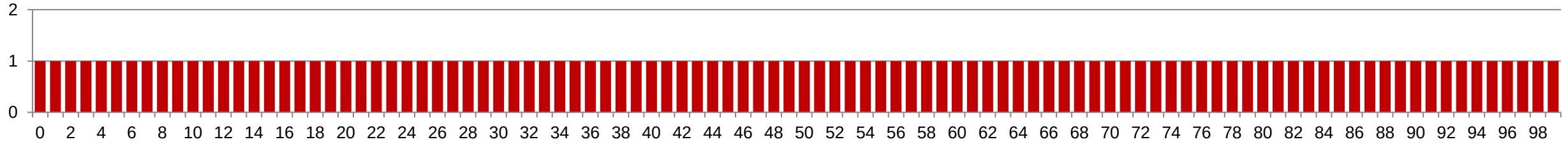

```
for (i = 0; i < N; i++)
    pthread_create(&tid[i], NULL, thread, &i);
```



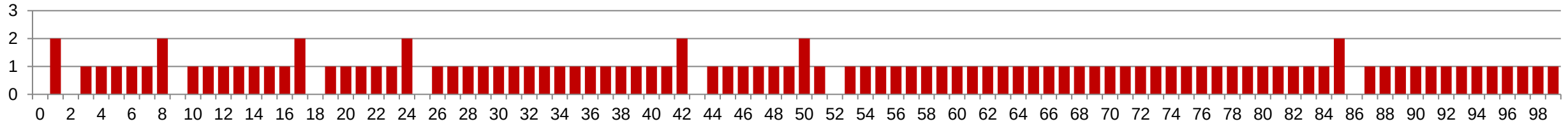
- 主线程中 i++ 和对等线程中读取 *vargp 产生竞争:
  - 如果在 i=0 时读取，那么结果就是正确的
  - 否则，对等线程会得到错误的id

实验结果

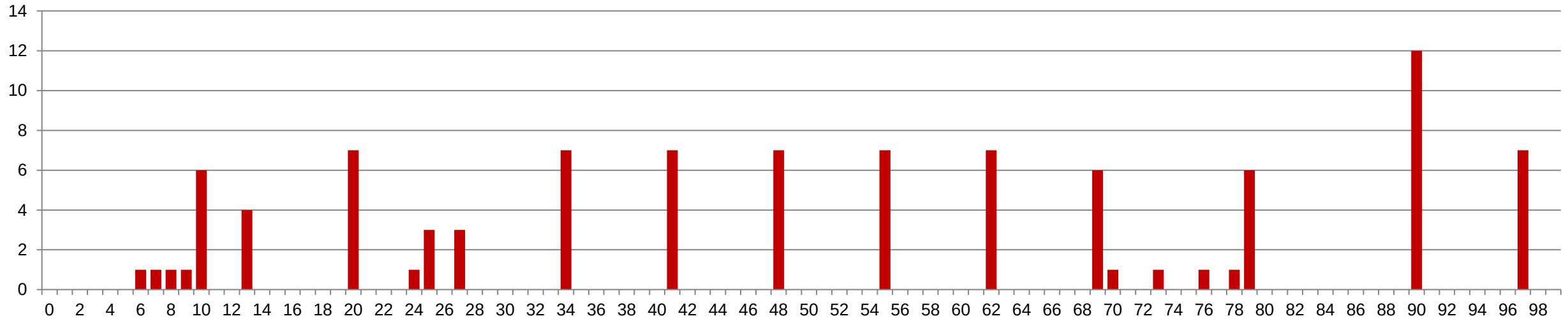
没有竞争



单核处理器



多核处理器



竞争确实会发生！

消除竞争


```c
/* Threaded program without the race */
int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = malloc(sizeof(int));
        *ptr = i;
        pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

避免意外的共享状态

norace.c

例题

- 1. 进程P1和P2均包含并发执行的线程，部分伪代码描述如下所示。下列选项中，需要互斥执行的操作是（ ）

A. a=1与a=2

B. a=x与b=x

C. x+=1与x+=2

D. x+=1与x+=3

```
//进程P1
int x=0;
Thread1( ) {
  int a;
  a=1;
  x+=1;
}
Thread2( ) {
  int a;
  a=2;
  x+=2;
}
```

```
//进程P2
int x=0;
Thread3( ) {
  int a;
  a=x;
  x+=3;
}
Thread4( ) {
  int b;
  b=x;
  x+=4;
}
```

例题

- 2. 设与某资源关联的信号量（K）初值为3，当前值为1。若M表示该资源的可用个数，N表示等待该资源的进程数，则M、N分别是（）

A. 0、1　　B. 1、0

C. 1、2　　D. 2、0

例题

- 3. 民航售票系统，n个售票处

```
/*Process Pi ,i=1,2,...,n*/
/*按订票要求找到数据库中的共享数据x[k]*/
/*x[k]存放某月某日某次航班的现有票数*/

R=x[k]; /*现有票数*/
if(R>=1){
        R--;
        x[k]=R;
        输出一张机票;
        }
else
        显示“票已售完”;
```

共享变量的修改存在冲突，请用信号量实现互斥

解题思路（读者写者问题）

- 使用信号量解决线程同步问题的步骤：
- （1）确定线程：
  - 包括线程的数量、线程的工作内容。
- （2）确定线程同步互斥关系：
  - 根据线程间是竞争临界资源还是相互合作处理上的前后关系，来确定线程间是互斥还是同步。
- （3）确定信号量：
  - 根据线程间的同步互斥关系确定信号量个数、含义、初始值，各线程需要对信号量进行的PV操作。
- （4）用类程序语言描述算法。

例题

- 4. 因为新冠疫情影响，某博物馆限制最多可容纳100人参观，有一个出入口，该出入口一次仅允许一个人通过。参观者的活动描述如下：

```
cobegin
参观者线程 i :
{
      …
      进门
      …
      参观
      …
      出门
      …
}
coend
```

请添加必要的信号量和P、V操作，以实现上述过程中的互斥与同步。要求写出完整的过程，说明信号量的含义并赋值。

解题思路（生产者消费者问题）

- 确定线程：线程数量及工作内容；
- 确定线程间的关系：
  - （1）互斥：多个线程间互斥使用同一个缓冲池；
  - （2）同步：当缓冲池空时，消费者必须阻塞等待；
    - ✓当缓冲池满时，生产者必须阻塞等待。
- 设置信号量：
  - Mutex：用于访问缓冲池时的互斥，初值是1
  - Full：“满缓冲”数目，初值为0；
  - Empty：“空缓冲"数目，初值为N。full+empty=N

Hope you enjoyed the OS course!