同步：进阶

Synchronization: Adnvanced

同步：进阶

Synchronization: Anvanced

信号量

- **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**

使用信号量实现互斥操作

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
- 将对共享变量的每次访问都用 P(mutex) 和 V(mutex) 操作包围起来

```
mutex = 1

P(mutex)
cnt++
V(mutex)
```

同步：进阶

Synchronization: Anvanced

利用信号量协调对共享资源的访问

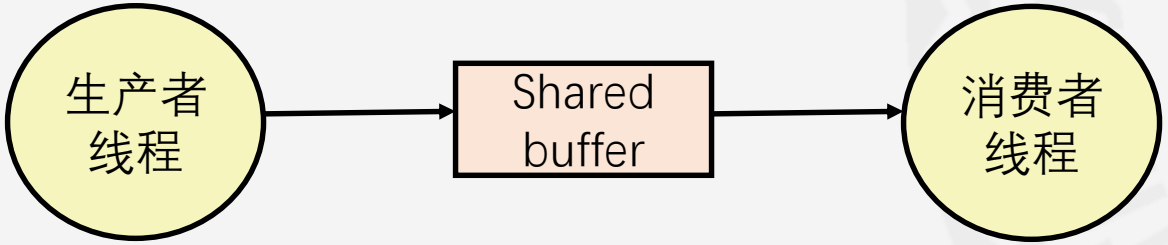
- **基本思想：线程使用信号量操作来通知另一个线程某个条件已经变为真**
  - 使用计数信号量来跟踪资源状态并通知其他线程。
  - 使用互斥锁来保护对资源的访问。

- 两个经典的例子：
  - 生产者-消费者问题
  - 读者-写者问题

同步：进阶

Synchronization: Anvanced

生产者-消费者问题



- **常见的同步模式：**
  - 生产者等待空槽位，将数据插入缓冲区，并通知消费者。
  - 消费者等待项目，从缓冲区中移除数据，并通知生产者。
- **示例：**
  - 多媒体处理：生产者创建 MPEG 视频帧，消费者将其渲染。
  - 事件驱动的图形用户界面：
    - 生产者检测鼠标点击、鼠标移动和键盘按键，并将相应的事件插入缓冲区。
    - 消费者从缓冲区检索事件并绘制显示。

大小为n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（互斥锁）：确保对缓冲区的互斥访问
  - slots（计数信号量）：计算缓冲区中可用的插槽数
  - items（计数信号量）：计算缓冲区中可用的项目数
- 使用一个名为 sbuf 的共享缓冲区包实现

同步：进阶

Synchronization: Anvanced

sbuf包声明部分

```c
#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);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
```

sbuf.h

sbuf包实现部分

初始化和销毁共享缓冲区

```c
/* 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包实现部分（2）

向共享缓冲区插入数据

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    Sem_wait(&sp->slots);                 /* Wait for available slot */
    Sem_wait(&sp->mutex);                  /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;  /* Insert the item */
    Sem_post(&sp->mutex);                  /* Unlock the buffer */
    Sem_post (&sp->items);                 /* Announce available item */
}
```

sbuf.c

sbuf包实现部分（3）

从共享缓冲区中删除数据

```c
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    Sem_wait(&sp->items);                    /* Wait for available item */
    Sem_wait(&sp->mutex);                    /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)]; /* Remove the item */
    Sem_post(&sp->mutex);                    /* Unlock the buffer */
    Sem_post(&sp->slots);                    /* Announce available slot */
    return item;
}
```

sbuf.c

同步：进阶

Synchronization: Anvanced

注意事项

```
void sbuf_insert(sbuf_t *sp, int item)
{
    Sem_wait(&sp->slots);
    Sem_wait(&sp->mutex);
    sp->buf[(++sp->rear)%(sp->n)] = item;
    Sem_post(&sp->mutex);
    Sem_post (&sp->items);
}
```

```
int sbuf_remove(sbuf_t *sp)
{
    int item;
    Sem_wait(&sp->items);
    Sem_wait(&sp->mutex);
    item = sp->buf[(++sp->front)%(sp->n)];
    Sem_post(&sp->mutex);
    Sem_post(&sp->slots);
    return item;
}
```

- 先申请计数信号量，再申请互斥锁，顺序不能颠倒
  - 如果颠倒，则先进入临界区，再申请资源信号量，如果申请不到计数信号量（=0），会阻塞线程，会发生死锁
- PV操作必须配对出现

同步：进阶

Synchronization: Anvanced

读者-写者问题

- 是互斥问题的一种更泛化的形式
- 问题描述：
  - 读线程只读取对象
  - 写线程修改对象
  - 写线程必须对对象具有独占访问权
  - 对读者的数量没有限制
- 在真实系统中经常发生，例如：
  - 在线航空订票系统
  - 多线程缓存 Web 代理

读者-写者问题的各种变种

- 读者优先：
  - 除非已经授予写者使用对象的许可，否则不应该让任何读者等待
  - 在等待的写者之后到达的读者优先于写者
- 写者优先
  - 一旦写者准备好写，它会尽快进行写操作
  - 在写者之后到达的读者必须等待，即使写者也在等待
- 以上两种情况下都可能发生饥饿
  - 饥饿：线程无限期的等待

同步：进阶

Synchronization: Anvanced

读者优先方案

读者

```c
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Initially = 1 */

void reader(void)
{
    while (1) {
        Sem_wait(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            Sem_wait(&w);
        Sem_post(&mutex);

        /* Critical section */
        /* Reading happens  */
        Sem_wait(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            Sem_post(&w);
        Sem_post(&mutex);
    }
}
```

写者

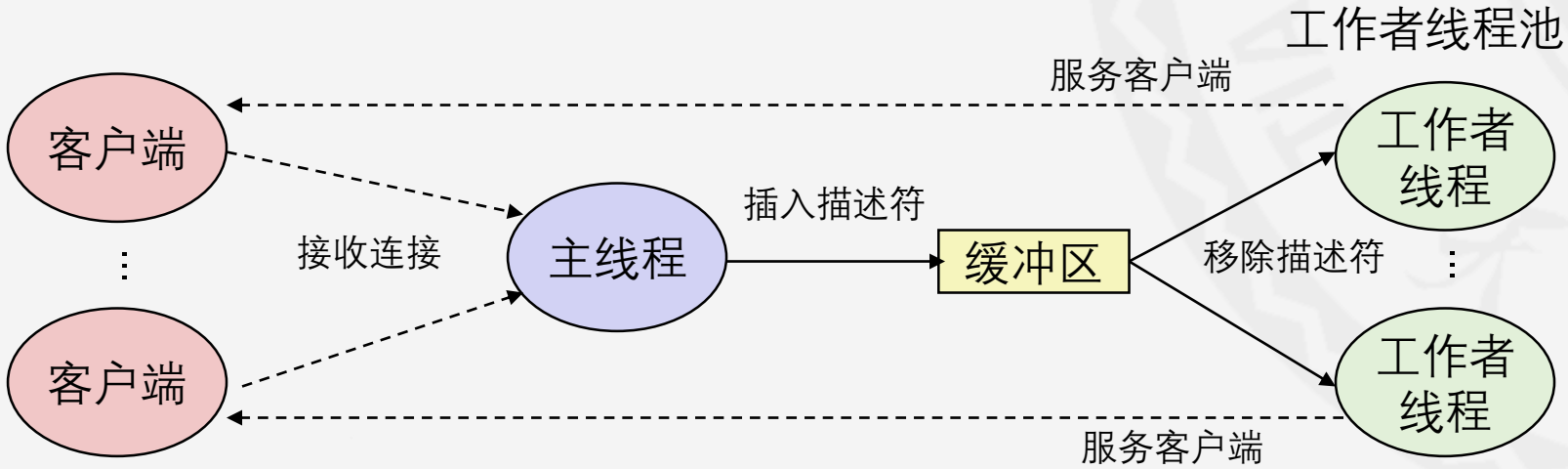
```c
void writer(void)
{
    while (1) {
        P(&w);

        /* Critical section */
        /* Writing happens  */

        V(&w);
    }
}
```

rw1.c

基于预线程化的并发服务



- 一组现有的线程不断地取出和处理来自有限缓冲区中的已连接描述符，并提供服务

基于预线程化的并发服务（2）

```c
sbuf_t sbuf; /* Shared buffer of connected descriptors */

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++)  /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
}
```

echoservert_pre.c

基于预线程化的并发服务（2）

工作者线程

```
void *thread(void *vargp)
{
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buf */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}
```

echoservert_pre.c

基于预线程化的并发服务（3）

echo_cnt初始化

```c
static int byte_cnt;  /* Byte counter */
static sem_t mutex;   /* and the mutex that protects it */

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

echo_cnt.c

基于预线程化的并发服务（3）

工作者线程所执行的echo_cnt服务

```c
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);  // 保证init_echo_cnt仅被执行一次
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        Sem_wait(&mutex);
        byte_cnt += n;
        printf("thread %d received %d (%d total) bytes on fd %d\n",
               (int) pthread_self(), n, byte_cnt, connfd);
        Sem_post(&mutex);
        Rio_writen(connfd, buf, n);
    }
}
```

echo_cnt.c

重要概念：线程安全

- 函数从线程中调用时必须是线程安全的。
- 定义：当一个函数从多个并发线程中重复调用时，始终能产生正确结果，则该函数是线程安全的。
- 线程不安全函数的种类：
  - 类型1：未保护共享变量的函数
  - 类型2：在多次调用之间保持状态的函数
  - 类型3：返回指向静态变量的指针的函数
  - 类型4：调用线程不安全函数的函数

同步：进阶

Synchronization: Anvanced

类型1：未保护共享变量

- 修正方法：使用P和V信号量操作
- 引入的问题：同步操作会降低代码执行速度

同步：进阶

Synchronization: Anvanced

类型2：在多次调用之间保持状态的函数

- 依赖于多次函数调用之间的持久状态
- 示例：依赖静态状态的随机数生成器

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
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

- 将状态作为参数的一部分传递，从而消除全局状态

```
/* rand_r - return pseudo-random integer on 0..32767 */

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

- 结果：使用rand_r函数的程序员必须维护种子

同步：进阶

Synchronization: Anvanced

类型3：返回指向静态变量的指针的函数

- 修复方法1：重写函数，使调用者传递存储结果的变量地址
  - 需要对调用者和被调用者进行更改
- 修复方法2：锁定并复制
  - 需要在调用者中进行简单更改（而不需要在被调用者中进行任何更改）
  - 然而，调用者必须释放内存。

```
/* 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;
}
```

类型4：调用线程不安全函数的函数

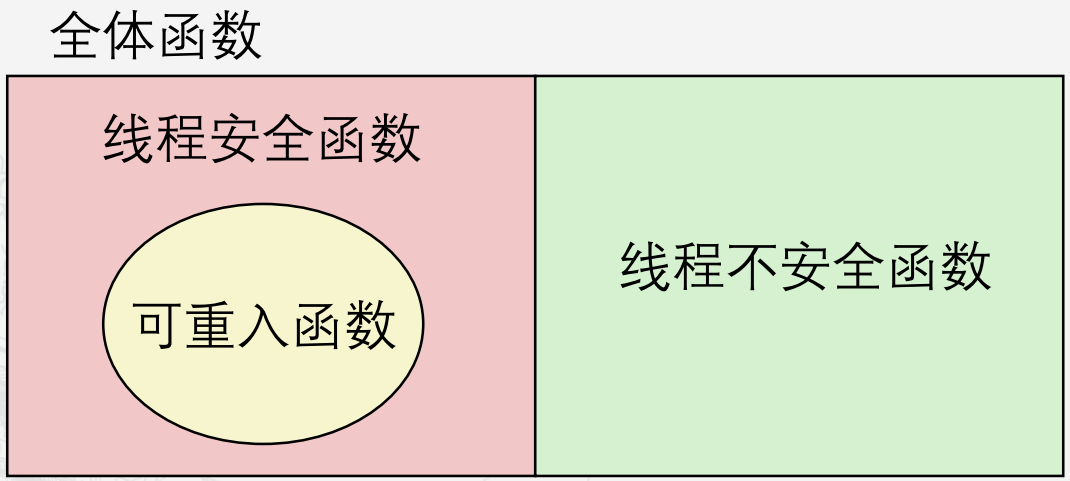
- 调用一个线程不安全函数会使调用它的整个函数也变得线程不安全。
- 修复：修改函数，使其只调用线程安全函数。

同步：进阶

Synchronization: Anvanced

可重入函数

- 定义：当一个函数在被多个线程调用时不访问任何共享变量，则该函数是可重入的。
  - 可重入函数是线程安全函数的重要子集。
    - 不需要同步操作
    - 使类型2函数线程安全的唯一方法是使其可重入（例如，rand_r函数）



同步：进阶

Synchronization: Anvanced

线程安全函数

- 标准C库中的所有函数都是线程安全的
  - 例如：malloc、free、printf、scanf
  - 线程安全和异步信号安全是有区别的
- 大多数Unix系统调用都是线程安全的，但也有一些例外情况：

线程不安全函数	种类	可重入的版本
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

同步：进阶

Synchronization: Anvanced

竞争条件

- 竞争条件发生在程序的正确性取决于一个线程在另一个线程到达点y之前到达点x的情况下。

```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 个线程共享 变量 i

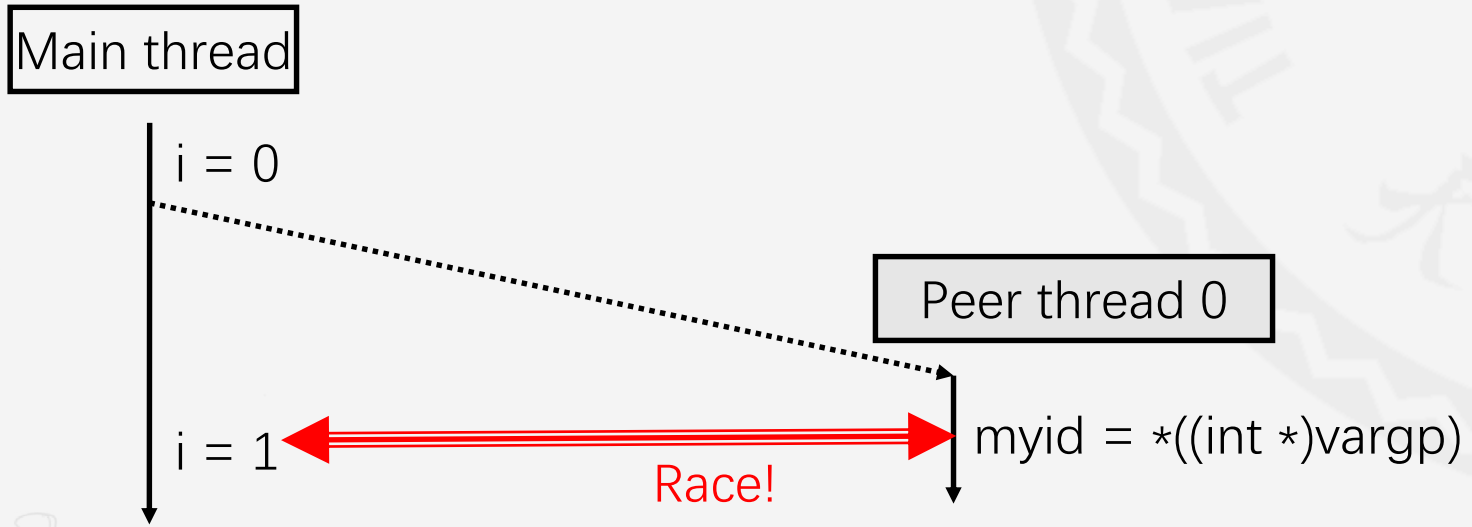
race.c

同步：进阶

Synchronization: Anvanced

竞争示例

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



- 在主线程中对i进行增量操作和在对等线程中对vargp进行解引用之间的竞争：
  - 如果解引用发生时i = 0，则没问题；
  - 否则，对等线程会得到错误的id值

同步：进阶

Synchronization: Anvanced

竞争真的可能发生吗?

主线程

```
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL,
                   thread,&i);
}
```

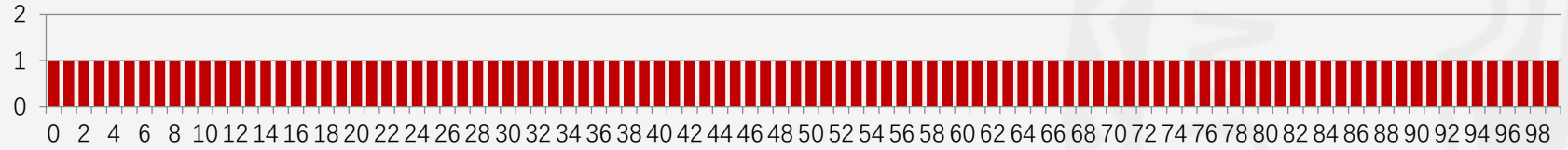
对等线程

```
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    int i = *((int *)vargp);
    save_value(i);
    return NULL;
}
```

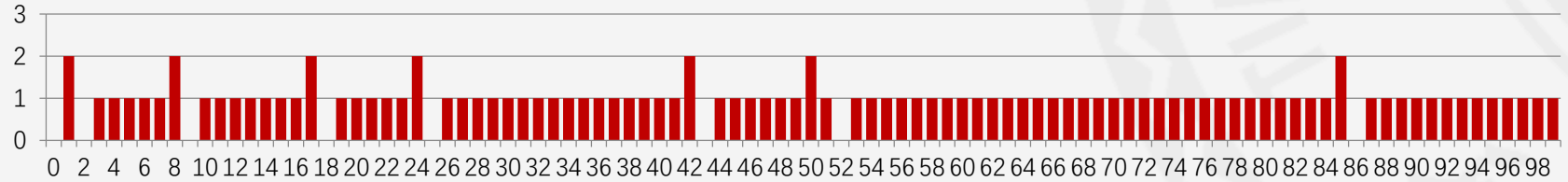
race.c

- 竞争测试
  - 如果没有竞争，那么每个线程会得到不同的i值
  - 保存的值集合将包括从0到99的每个值的一份副本

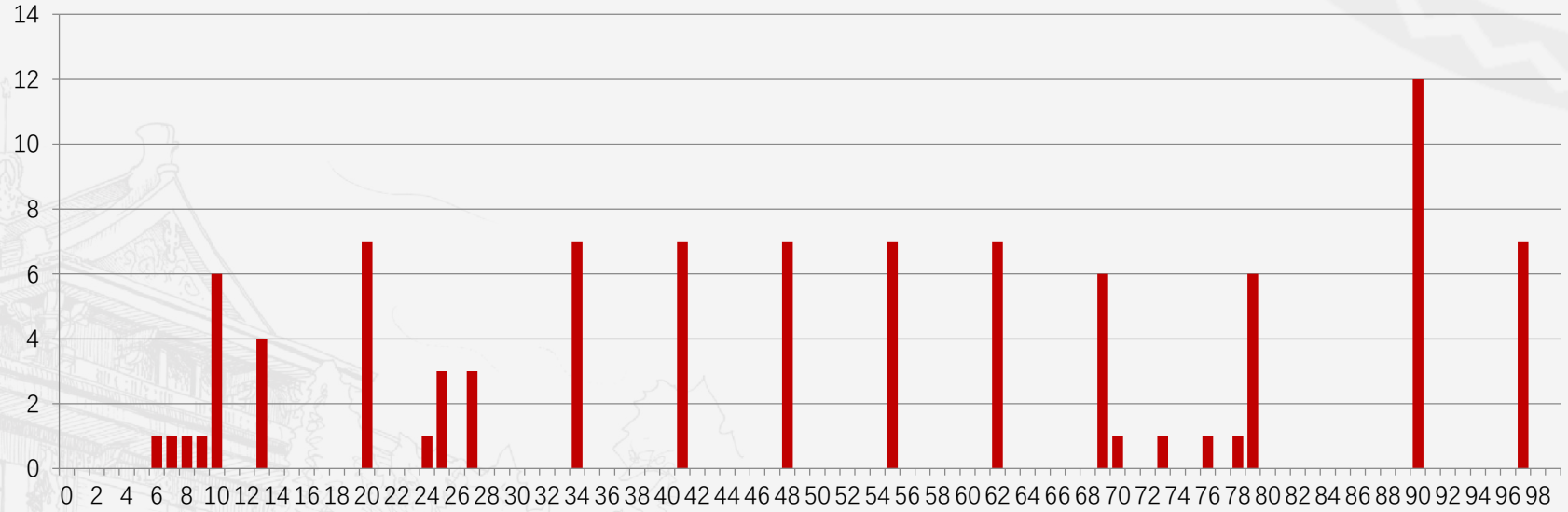
理想情况：无竞争出现



单核



多核



同步：进阶

Synchronization: Anvanced

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);
}
```

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

同步：进阶

Synchronization: Anvanced

死锁

- 定义：当一个进程正在等待一个永远不会成立的条件时，该进程就发生了死锁。

- 典型场景：
  - 进程1和进程2需要两个资源（A和B）才能继续执行。
  - 进程1获取资源A，等待资源B。
  - 进程2获取资源B，等待资源A。
  - 结果是两者将永远等待下去！

同步：进阶

Synchronization: Anvanced

使用信号量时出现的死锁问题

```c
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```c
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

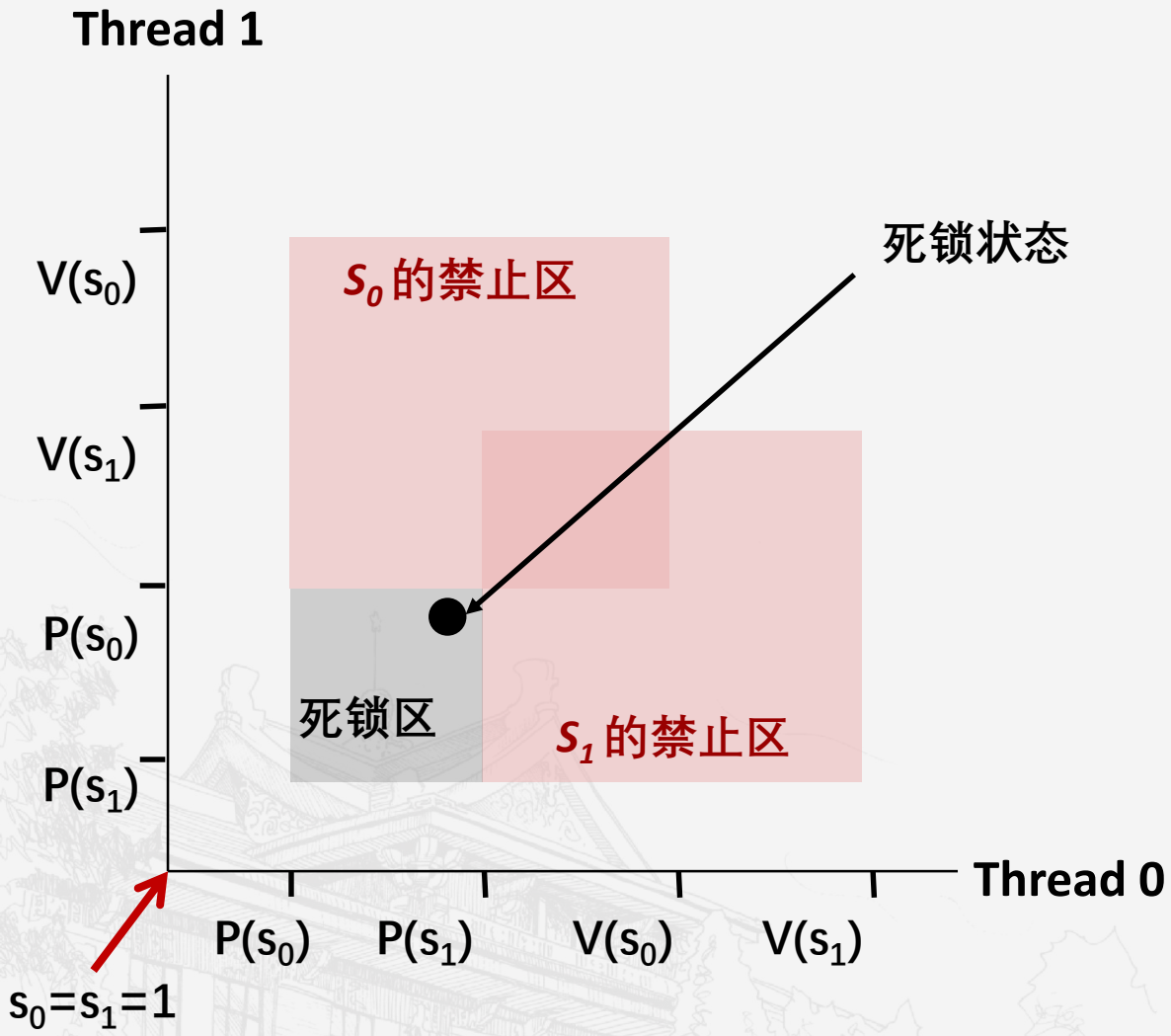
Tid[0]:
P(s_0);
P(s_1);
cnt++;
V(s_0);
V(s_1);

Tid[1]:
P(s_1);
P(s_0);
cnt++;
V(s_1);
V(s_0);

同步：进阶

Synchronization: Anvanced

使用进程图可视化死锁程序



- 加锁引入了死锁的潜在可能性：
- 即等待一个永远不会成立的条件。
- 任何进入死锁区域的轨迹最终都会达到死锁状态，等待s0或s1变为非零。
- 其他轨迹则会幸运地避开死锁区域。
- 不幸的事实：死锁通常是非确定性的（竞争）。

同步：进阶

Synchronization: Anvanced

```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

避免死锁

以相同的顺序获取共享资源

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

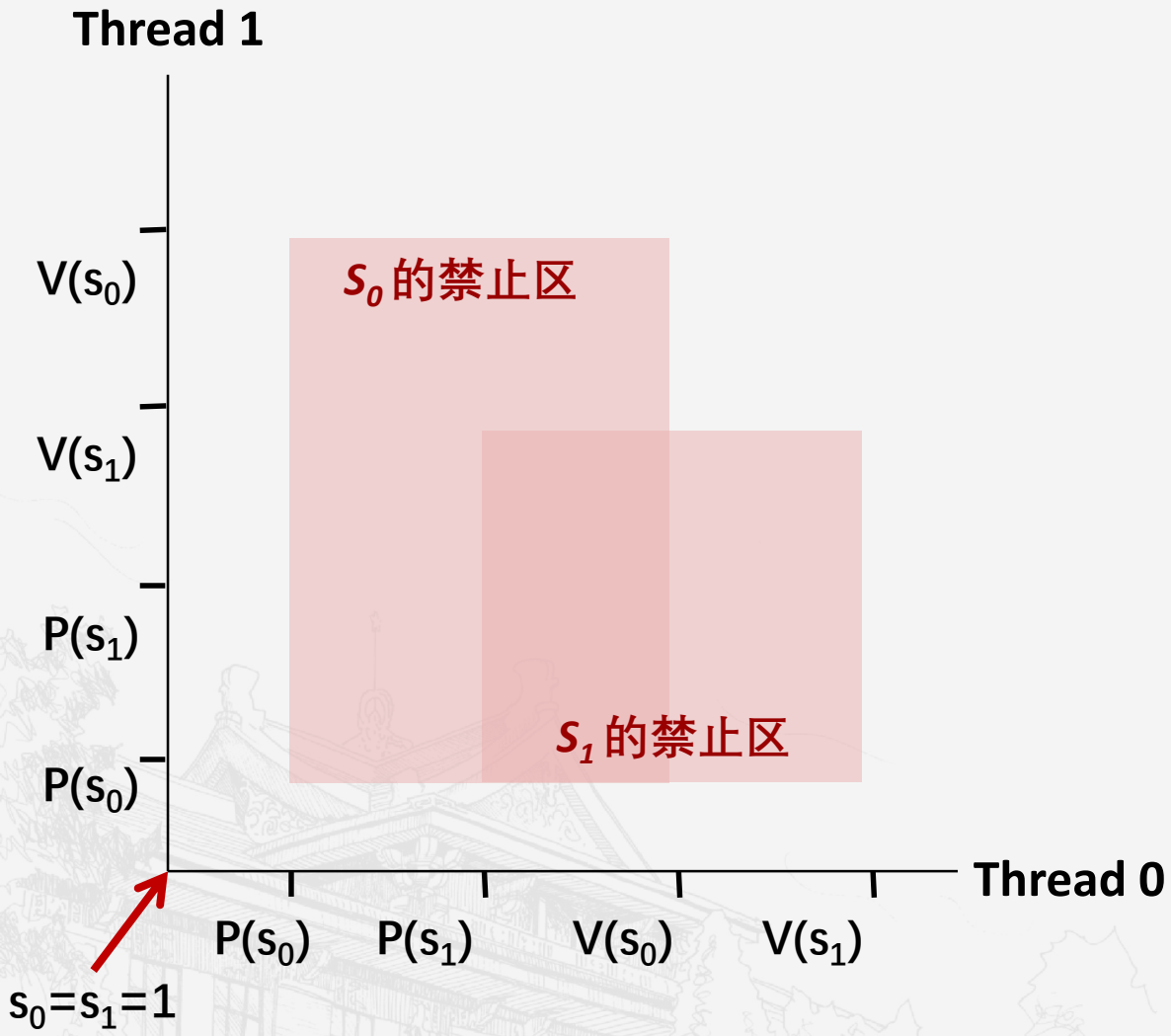
```
Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s1);
```

```
Tid[1]:
P(s0);
P(s1);
cnt++;
V(s1);
V(s0);
```

同步：进阶

Synchronization: Anvanced

使用进程图可视化避免死锁程序



- 程序运行轨迹没有被卡住的可能性
- 进程以相同的顺序获取锁
- 此时，锁的释放顺序无关紧要