[读书笔记]CSAPP：28[VB]同步

深度人工dazed
阿巴阿巴阿巴

关注他

5 人赞同了该文章

视频地址：

【精校中英字幕】2015 CMU 15-213 CSAPP 深入理解计算机系统 课程视频_哔哩哔哩 (゜- ゜)つロ 干杯~...
www.bilibili.com/video/BV1iW411d7hd?p=22

2015CMU 15-213 CSAPP 深入理解计算机系统 课程视频含英文字幕（精校字幕...
www.bilibili.com/video/BV1XW411A7fB?p...

课件地址：

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/24-sync-basic.pdf
www.cs.cmu.edu/afs/cs/academic/class/15213-f15/ww...

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/25-sync-advanced.pdf
www.cs.cmu.edu/afs/cs/academic/class/15213-f15/ww...

本章对应于书中的12.4, 12.5和12.7。

- 只有资源共享才会产生同步问题，如果线程没有共享任何资源，就没有同步问题。判断一个变量是否是共享的，首先判断该变量在内存的实例个数，然后看该变量的一个实例是否被多个线程引用。对共享变量进行操作时要进行同步。
- 多线程程序要画出进度图来分析
- 互斥锁主要是用来保护共享变量的，只要有多个线程访问共享变量，就加上互斥锁。如果要对资源进行调度，就使用计数信息量。
- **多线程代码**：首先判断哪些变量是共享变量，加上互斥锁。如果有多个互斥锁，画出进度图，避免出现死锁。
- 由于同步的开销很大，所以尽量写成可重入的形式。
- 多进程程序中，需要注意对信号进行阻塞；多线程程序中，需要注意对共享变量进行同步。
- 多线程程序要注意：共享变量，竞争，死锁，线程安全。

---

1 多线程程序的共享变量

我们以下面的程序为例来说明多线程共享变量的问题

code/conc/sharing.c

```c
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr;  /* Global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```


code/conc/sharing.c

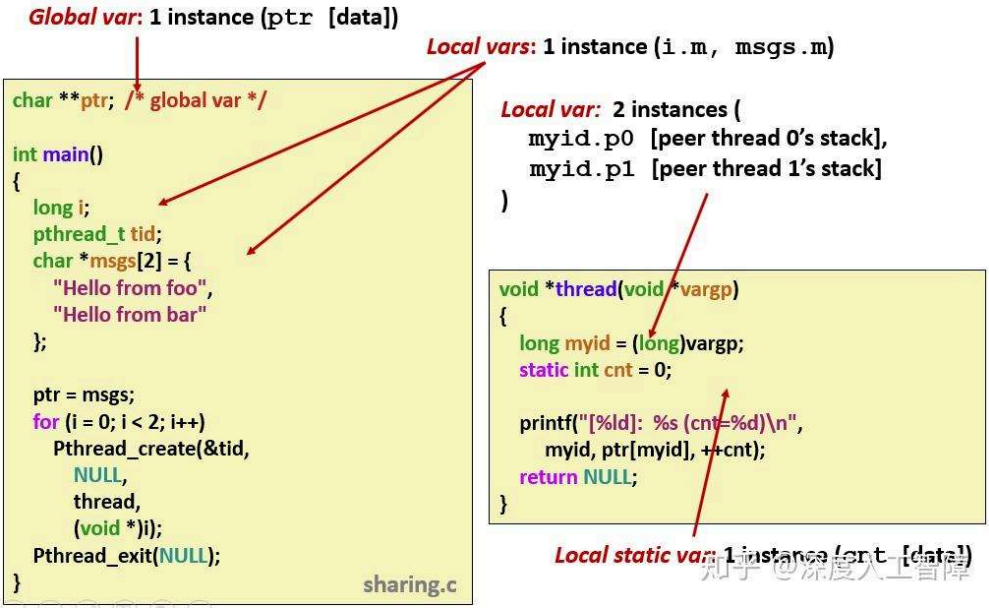
一组并发的线程运行在一个进程上下文中，每个线程具有自己的线程上下文，包括TID、栈、栈指针、程序计数器、条件码和通用目的寄存器值，这就说明每个进程可以通过自己的程序计数器运行自己的代码，而通过栈可以保存自己的局部变量，这些栈是保存在虚拟地址空间中的栈区域。而进程的其余部分在所有线程中是共享的，包括整个用户虚拟地址空间，其中有代码、读写数据、堆、共享代码库代码数据区域、打开的文件描述符和信号处理程序等等。

但是并不会完全遵守以上的线程内存模型，比如我们这里要求每个线程有自己独立的栈，但是这里并不会对其他线程的访问进行限制，比如上面代码在线程例程中通过全局变量ptr来访问主线程中栈的局部变量 `msgs` ，所以要十分小心这个线程内存模型。

根据链接这一章可知，未初始化静态变量、以及初始化为0的全局变量或静态变量的符号 `.bss` 节中，初始化过的全局变量或静态变量的符号在 `.data` 节中，这两个数据节在生成可执行目标文件时，都被分配到了数据段中，所以全局变量和静态变量的实例是保存在进程的虚拟地址空间中的数据段中的，所有线程都能访问到，且虚拟内存空间值包含一个实例。而局部非静态变量是保存在栈或寄存器中的，每个线程都有自己的栈和寄存器值，所以局部非静态变量的实例是保存在线程的栈或寄存器中的，通常只有自己能访问到。

而判断一个变量是否是共享的，就看该变量的一个实例是否被多个线程引用。

- `ptr` ： 它是未初始化全局变量，实例保存在 `.bss` 节中，所有线程都可以访问。在主线程和两个对等线程中都有引用过，所以是共享的。
- `cnt` ： 它是初始化了的静态变量，实例保存在 `.data` 节中，所有线程都可以访问。在两个对等线程中都有引用过，所以是共享的。
- `i.m` ： 主线程中的变量 `i` 是局部变量，保存在主线程中的栈，没有被其他线程引用过，所以是不共享的。
- `msgs.m` ： 主线程中的变量 `msgs` 是局部变量，保存在主线程中的栈。由于全局变量 `ptr` 指向了 `msgs` ，而两个对等线程通过 `ptr` 间接地引用了 `msgs` 的数据，所以是共享的。
- `myid.0` 和 `myid.1` ： 在两个对等线程中的局部变量 `myid` ，它们各自保存在对等线程自己的栈中，也不存在间接引用，所以它们是不共享的。



2 用信号量同步线程

由于共享变量的存在，可能会引入**同步错误（Synchronization Error）**。

code/conc/badcnt.c

```c
/* WARNING: This code is buggy! */
#include "csapp.h"

void *thread(void *vargp);  /* Thread routine prototype */

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

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

    /* Check input argument */
    if (argc != 2) {
        printf("usage: %s <niters>\n", argv[0]);
        exit(0);
    }
    niters = atoi(argv[1]);

    /* Create threads and wait for them to finish */
    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);
}

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

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

    return NULL;
}
```

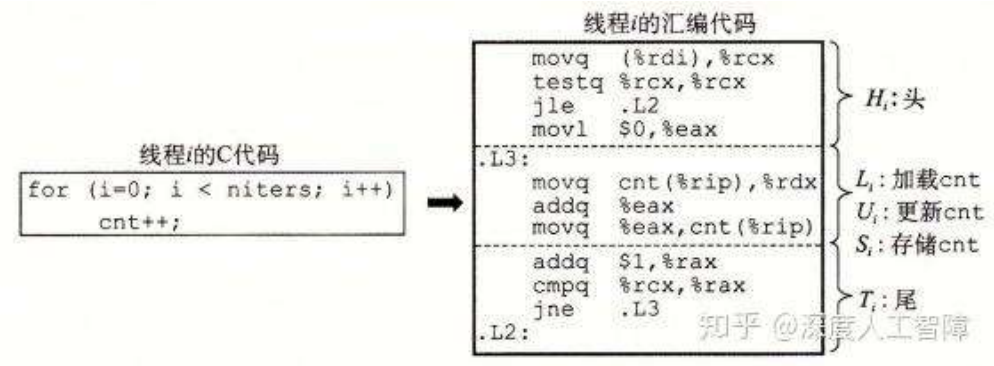

code/conc/badcnt.c

注意：由于我们这里在线程例程中不会对 `niters` 进行修改，所以可以直接将 `niters` 的地址传入 `pthread_create` 函数而不用担心竞争。

我们运行该程序时，希望 `cnt` 的值为 `2*niters` ，但是运行出来的结果却是

```
linux>  ./badcnt 1000000
BOOM! cnt=1445085

linux>  ./badcnt 1000000
BOOM! cnt=1915220

linux>  ./badcnt 1000000
BOOM! cnt=1404746
```


我们观察在线程例程中的汇编



其中， %rcx 保存 niters 的值， %rax 保存 i 的值， %rdx 保存 cnt 的值。我们知道，在单核线程并发时，是由内核调度这两个线程的逻辑流进行交替运行的，而上图的中间部分是两个逻辑流对共享变量 cnt 的操作，所以当其中一个线程运行到中间部分又被内核调度到另一个线程时，就可能会出错。这两个线程的逻辑流具有一些能得到正确结果的执行顺序，也有一些会得到错误结果的执行顺序，比如

步骤	线程	指令	$\%rdx_1$	$\%rdx_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

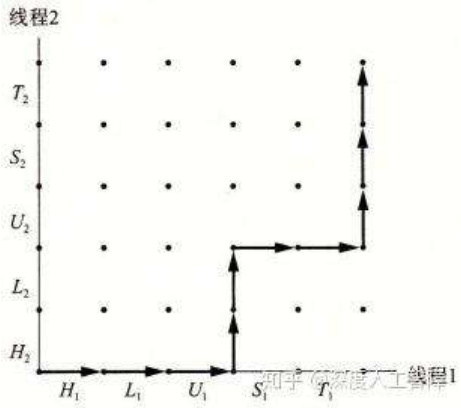
a）正确的顺序

步骤	线程	指令	$\%rdx_1$	$\%rdx_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

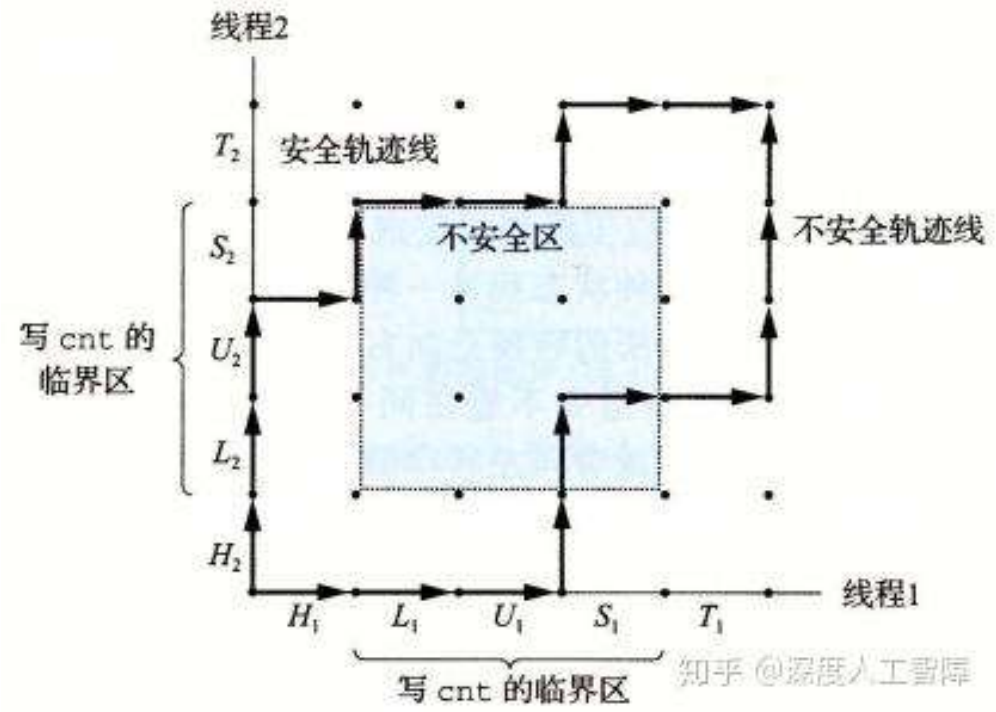
b）不正确的顺序

我们可以通过**进度图**[+]**（Process Graph）**来探讨指令执行的顺序。首先有n个并发线程就创建n维笛卡尔积空间，每个维度对应一个线程的进度，根据线程的指令顺序对每一维进行划分，则在该笛卡尔积空间中的某一条轨迹就是并发线程的指令执行顺序。**注意：**

- 图的原点是所有并发线程还未开始执行的初始状态
- 两条指令不能同时执行，所以不存在斜线
- 程序不允许反向运行，所以轨迹总是向右上方增长的



如上图所示就是根据汇编指令[+]得到的一条指令执行顺序的轨迹。其中我们将操作共享变量 cnt 的指令 (L_i,U_i,S_i) 当做**临界区（Critical Section）**，我们希望临界区不和其他线程的临界区交替执行，来使得线程拥有对共享变量的**互斥访问（Mutually Exclusive Access）**。我们可以将两个线程的临界区围起来构成一个**不安全区（Unsafe Region）**，只要有一条轨迹穿过不安全区，说明两个线程对共享变量 cnt 的操作是交替执行的，可能就会出现错误。我们将穿过不安全区的轨迹称为**不安全轨迹（Unsafe Trajectory）**，而其他的轨迹就是**安全轨迹（Safe Trajectory）**。



为了保证可以获得安全轨迹，可以通过信号量的方法同步线程来获得安全轨迹。

2.1 信号量

为了同步线程[+]，提出了一种特殊类型变量**信号量（Semaphore）** `s`，它是具有非负整数的全局变量，只能通过两个系统调用函数进行操作：

- **`P(s)`：**
  - 如果 `s` 是非零的，就对其减1并立即返回（原子操作）
  - 如果 `s` 是0，则将当前线程挂起，直到被 `V(s)` 函数加1且重启线程， `P(s)` 才会再对 `s` 减1并返回。
- **`V(s)`：** 将 `s` 加1（原子操作），如果有任何线程被P挂起，该函数会重启其中任意一个线程，此时控制权又回到了那个线程的P中。

P和V这样的定义是为了保证了信号量 `s` 不会变成负值，才能够对线程进行同步，这种属性称为**信号量不变性（Semaphore Invariant）**。这里有一些信号量的函数

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);  //P
int sem_post(sem_t *s);  //V
```

其中，每个信号量都要通过 `sem_init` 函数进行初始化，且初始化值为 `value`，然后通过 `sem_wait` 和 `sem_post` 函数来调用P和V。我们这里提供这两个的封装

```
#include "csapp.h"
int P(sem_t *s);
int V(sem_t *s);
```

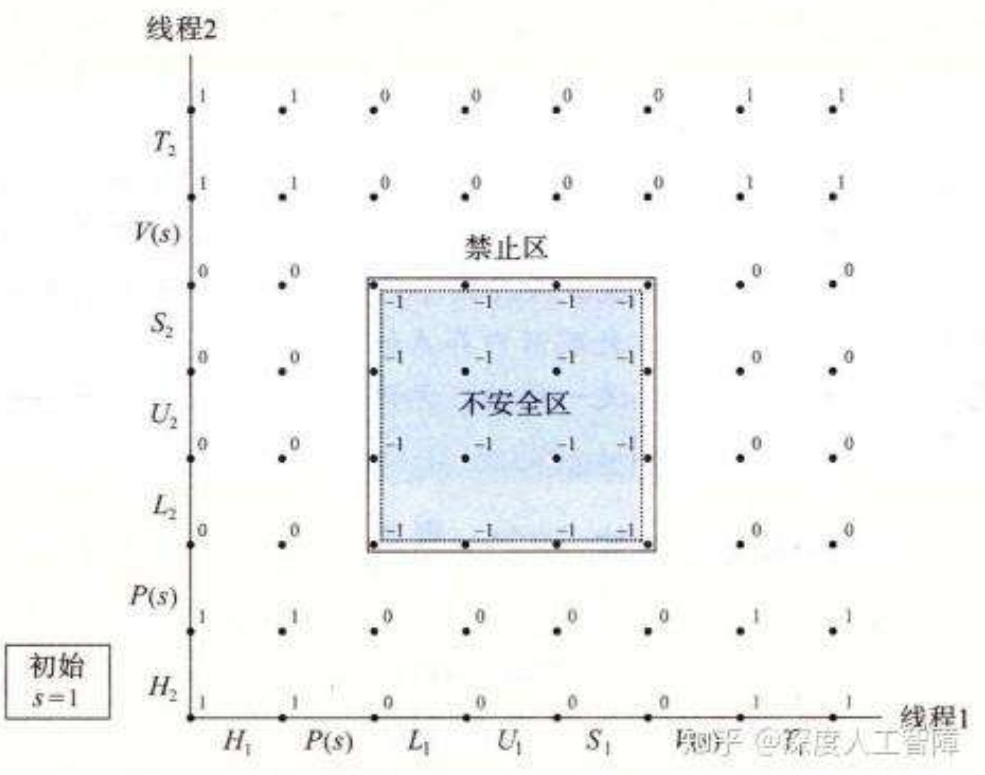
使用信号量来确保对共享变量的互斥访问，**基本思想**是将共享变量和一个信号量联系起来，然后使用P和V将对应的临界区包围起来。比如我们将原来的代码改为

```
sem_t mutex;  //信号量
//主线程中初始化信号量
sem_init(&mutex, 0, 1);
```

```
//对等线程例程中
for(i=0; i<niters; i++){
  P(&mutex);
  cnt++;
  V(&mutex);
}
```

这里我们声明了一个信号量 `mutex` ，然后在线程例程中用P和V将 `cnt++` 围住了。首先，当还没有线程执行线程例程时， `mutex` 被初始化为1，当有一个线程A在执行线程例程时，会首先执行 `P(&mutex)` ，此时由于 `mutex` 不是0，所以会将其减1，然后执行 `cnt++` ，此时 `mutex` 就为0，如果在执行 `cnt++` 指令的中间有另一个线程B也要执行线程例程时，在 `P(&mutex)` 中由于当前 `mutex` 为0，则P会将线程B挂起，直到线程A执行到 `V(&mutex)` 时，会将 `mutex` 加1，然后重启线程B，此时线程B又会执行 `P(&mutex)` ，此时由于 `mutex` 为1，所以也会顺利执行。

更形式化来说，我们将这种用来保护共享变量的信号称为**二元信号量（Binary Semaphore）**，因为它的值总是0或1。以提供互斥为目的的二元信号量也称为**互斥锁（Mutex）**，则执行P操作就称为**上锁**，执行V操作就称为**解锁**，一个线程对互斥锁上锁了还未解锁，就称该线程**占用**这个互斥锁。我们通过互斥锁将线程的临界区包围起来，则会构成一个**禁止区（Forbidden Region）**，在禁止区内由于信号量都为-1，想要操作共享变量就要通过P对信号量减1，此时就会将当前线程挂起，直到占用的线程解锁



图中每个点的值为信号量的值

2.2 利用信息量来调度共享资源

除了用二元信息量来提供互斥锁以外，还能用来作为一组可用资源的计数器，称为**计数信号量**，用来调度对共享资源的访问，经典的模型有**生产者-消费者问题（Producer-Consumer Problem）**和**读者-写者问题（Reader-Writer Problem）**。

2.2.1 生产者-消费者问题

我们考虑这样一类问题，假设目前我们去一个商场购买口罩，我们作为消费者可以自己去购买，如果口罩不够了，则消费者就等待生产者生产出更多的口罩放在商场供我们购买，如果生产者一不小心生产了太多口罩，则商场放不下了，就要等商场有空的区域时再生产。这就是一个生产者-消费者问题。

在这里，我们将生产者和消费者看成是生产者线程和消费者线程，而商场看成是一个具有有限空间的缓冲区，生产者线程会不断生成新的**项目（Item）**，然后将其插入缓冲区中，而消费者线程会不断从缓冲区中取出这些项目去消费。

由于缓冲区在这两个线程中是共享变量，从中取出项目和插入项目都要更新共享变量，所以要对缓冲区设置一个互斥锁，保证只有一个线程能访问缓冲区。其次，我们还需要通过计数信号量对缓冲区的访问进行调度，当缓冲区有空间来插入项目时，生产者线程才尝试插入项目，当缓冲区中有项目时，消费者线程才尝试从中取出项目。我们可以首先定义缓冲区的数据结构

code/conc/sbuf.h

```
1  typedef struct {
2      int *buf;          /* Buffer array */
3      int n;             /* Maximum number of slots */
4      int front;         /* buf[(front+1)%n] is first item */
5      int rear;          /* buf[rear%n] is last item */
6      sem_t mutex;       /* Protects accesses to buf */
7      sem_t slots;       /* Counts available slots */
8      sem_t items;       /* Counts available items */
9  } sbuf_t;
```


code/conc/sbuf.h

其中， `buf` 是一个动态分配的缓冲区， `n` 是缓冲区的大小， `front` 和 `rear` 是第一个项目和最后一个项目的索引，这里将 `buf` 实现为循环的数组。 `mutex` 是访问缓冲区的互斥锁， `slots` 是用来调度生产者线程的计数信息量， `items` 是用来调度消费者线程的计数信息量。

注意：计数信息量 `slot` 和 `items` 是用来判断能否执行生产者线程或消费者线程，如果可以，则会对 `buf` 进行访问，由于 `buf` 是共享变量，所以它的访问需要 `mutex` 互斥锁，所以这里会有三个信息量。

```
#include "csapp.h"
#include "sbuf.h"

/* 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 zero data items */
}
```


首先，我们需要初始化该缓冲区，其中 `sp->mutex` 要初始化为二院信息量， `sp->slots` 要初始化为 `n` ，表示有 `n` 个空间供生产者线程使用， `sp->items` 要初始化为0，表示现在还没有项目供消费者线程消费。

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

生产者线程可以调用 sbuf_insert 函数来将将项目插入到缓冲区中。首先需要通过 sp->slots 查看是否有空闲的区域，如果 sp->slots 为0，说明没有空闲的区域，则 P 会将生产者线程挂起，直到 sp->slots 大于0且该线程被 V 重启。然后需要通过 sp->mutex 来获得缓冲区的互斥锁，然后才能将 item 插入缓冲区中。随后通过 V 将 sp->mutex 解锁，然后用 V 将 sp->items 加1，如果刚好有消费者由于 sp->items 为0而挂起，则该 V 会随机重启一个消费者线程。

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


消费者线程可以调用 sbuf_remove 函数来从缓冲区中获得项目。首先通过 sp->items 查看缓冲区中是否存在项目，如果 sp->items 为0表示没有项目供消费者线程消费，此时消费者线程会挂起，直到被生产者线程的 P(&sp->items) 重启。然后通过 sp->mutex 来获得缓冲区的互斥锁，然后才能从缓冲区中获得项目 item 。随后通过 V 将 sp->mutex 解锁，然后用 V 将 sp->slots 加1，如果刚好有生产者线程由于 sp->slots 为0而挂起，则该 V 会随机重启一个生产者线程。

最终我们可以通过以下函数来释放缓冲区

```
/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```


2.2.2 读者-写者问题

当一组并发的线程要访问一个共享对象时，有些线程只读对象称为**读者线程**，而其他对象只写对象称为**写者线程**，写者线程必须有对对象的独占访问，而读者线程可以和其他读者线程共享对象。根据读者和写者的优先级，可以**分成两类问题**：

- **第一类**：读者优先级高于写者，除非写者已占用互斥锁，否则读者都不会一直等待，在等待的写者之后到达的读者具有更高的优先级。
- **第二类**：写者优先级高于读者，一旦写者准备好写时，它会尽快执行写操作，在写者之后到达的读者必须等待。

这两种情况都会出现饥饿问题。这里提供一个第一类读者-写者问题的代码

```
/* Global variables */
int readcnt;    /* Initially = 0 */
sem_t mutex, w; /* Both initially = 1 */
```


这里定义了一个全局共享变量 readcnt 来统计读者线程的数目，由于不同读线程要对其进行访问，所以需要一个互斥锁 mutex 来控制对 readcnt 的访问，然后还有一个 w 互斥锁来控制对对象的访问。

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

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

        V(&w);
    }
}
```

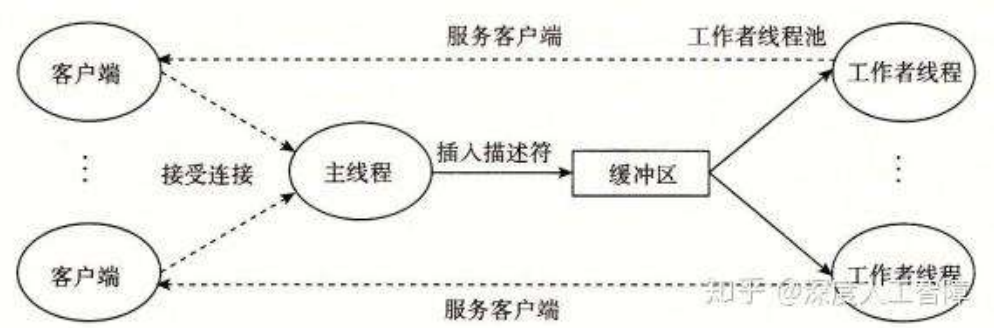

在写者线程中，会通过用 `P(&w)` 和 `V(&w)` 将临界区包围住，从而根据互斥锁 `w` 的状态来访问对象，此时保证了一次只能有一个写者线程能访问对象。

```
void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Critical section */
        /* Reading happens  */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```


在读者线程中，首先要通过 `readcnt` 来统计其中有多少个读者线程，因为是读者优先的，所以只有进入的读者数目 `readcnt` 为0时，才能让写者线程去访问对象，所以这里每个读者线程进来和出去时都要修改 `readcnt` ，而 `readcnt` 是共享变量，所以需要通过 `mutex` 来对其上锁。其次，我们只有当 `readcnt` 为1时才执行 `P(&w)` 对 `w` 进行上锁，当有写者线程尝试访问对象时，会被挂起，而当有其他读者线程进入时，由于 `readcnt` 被修改了，所以不会访问到 `P(&w)` ，也就不会被挂起，此时后续的读者线程就能不断进入临界区访问对象。而只有当 `readcnt` 为0时才执行 `V(&w)` 对 `w` 解锁，此时挂起的写者线程就能被重启。

2.3 基于预线程化的并发服务器

我们之前实现的基于线程的并发服务器，为每个客户端都生成一个线程进行处理，其实十分耗费资源。我们可以基于生产者-消费者问题来构建一个**预线程化（Prethreading）**的并发服务器，它将主线程看成是生产者，不断接收客户端发送的连接请求，并将已连接描述符放在缓冲区中，而提前创建的一系列线程就是消费者，不断从缓冲区中取出已连接描述符与客户端通信



code/conc/echoservert-pre.c

```
1   #include "csapp.h"
2   #include "sbuf.h"
3   #define NTHREADS  4
4   #define SBUFSIZE  16
5
6   void echo_cnt(int connfd);
7   void *thread(void *vargp);
8
9   sbuf_t sbuf; /* Shared buffer of connected descriptors */
10
11  int main(int argc, char **argv)
12  {
13      int i, listenfd, connfd;
14      socklen_t clientlen;
15      struct sockaddr_storage clientaddr;
16      pthread_t tid;
17
18      if (argc != 2) {
19          fprintf(stderr, "usage: %s <port>\n", argv[0]);
20          exit(0);
21      }
22      listenfd = Open_listenfd(argv[1]);
23
24      sbuf_init(&sbuf, SBUFSIZE);
25      for (i = 0; i < NTHREADS; i++)  /* Create worker threads */
26          Pthread_create(&tid, NULL, thread, NULL);
27
28      while (1) {
29          clientlen = sizeof(struct sockaddr_storage);
30          connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
31          sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
32      }
33  }
34
35  void *thread(void *vargp)
36  {
37      Pthread_detach(pthread_self());
38      while (1) {
39          int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
40          echo_cnt(connfd);                /* Service client */
41          Close(connfd);
42      }
43  }
```


code/conc/echoservert-pre.c

第24行首先对缓冲区进行初始化，然后生成了 NTHREADS 个消费者线程，在线程例程中，会通过 pthread_detach 函数来分离消费者线程，然后死循环通过 sbuf_remove 函数来从缓冲区中获得已连接描述符 connfd ，当 pthread_create 函数返回时，这些消费者线程就开始运行了，此时由于 sbuf->items 为0，所以这些消费者线程会被 P(&sbuf->items) 挂起。然后在生产者线程中通过 accept 函数来获得已连接描述符 connfd ，并通过 sbuf_insert 函数将其插入到缓冲区中，此时 sbuf->items 就不为0了，那些被挂起的消费者线程就会一个个被 V(&sbuf->items) 重启，然后将获得的 connfd 传入 echo_cnt 函数来与客户端通信。这里的对等线程的数目是我们一开始创建的

那些线程，而不是等到获得 connfd 时才创建，所以称为预线程化，可以自己控制对等线程的数目。

code/conc/echo-cnt.c

```c
#include "csapp.h"

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;
}

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("server received %d (%d total) bytes on fd %d\n",
               n, byte_cnt, connfd);
        V(&mutex);
        Rio_writen(connfd, buf, n);
    }
}
```


code/conc/echo-cnt.c

这个 echo_cnt 函数会通过全局变量 byte_cnt 来统计服务器总共从客户端获得的字节数，由于不同的消费者线程都会调用 echo_cnt 函数，所以会有多个线程来访问 byte_cnt 共享变量，所以我们对 byte_cnt 的访问需要使用互斥锁。而我们这里将互斥锁和 byte_cnt 的初始化过程放在了子线程中，通过 pthread_once 函数来保证该初始化函数 init_echo_cnt 函数只被执行一次。

3 其他并发问题

可以发现，线程由于会对共享变量进行操作，所以我们尝试使用同步技术来解决操作共享资源时出现的问题，接下来会介绍几个典型问题。

3.1 线程安全

当且仅当一个函数被多个并发线程反复调用还会一直产生正确结果时，才称为**线程安全的（Thread-Safe）**，我们可以根据线程不安全的原因分成以下几个类：

- **第一类：不保护共享变量的函数：**当我们没有对共享变量进行保护，而它的操作也是非原子的时，就可能会造成错误。
  - **解决方法：**引入互斥锁，使用 P 和 V 将对共享变量访问的部分围起来，但程序速度会变慢。
- **第二类：跨多个函数调用依赖于持久状态：**比如以下伪随机数[+]生成器，在单个线程程序中，我们可以通过 srand 函数定义相同的随机种子来得到可重复的随机数序列，但是在多个线程中调用 rand 函数时，由于 rand 函数中依赖于共享的前一个状态 next_seed ，此时 next_seed 可能会被其他线程修改而无法获得相同的随机数序列。
  - **解决方法：**重写函数，将状态作为参数的一部分传递，避免被其他线程修改。

code/conc/rand.c

```
1  unsigned next_seed = 1;
2
3  /* rand - return pseudorandom integer in the range 0..32767 */
4  unsigned rand(void)
5  {
6      next_seed = next_seed*1103515245 + 12543;
7      return (unsigned)(next_seed>>16) % 32768;
8  }
9
10 /* srand - set the initial seed for rand() */
11 void srand(unsigned new_seed)
12 {
13     next_seed = new_seed;
14 }
```


code/conc/rand.c

- **第三类：返回指向静态变量的指针的函数：**静态变量保存在数据区，在进程中只有一个实例，是所有线程共享的，当你返回一个静态变量的指针时，是指向一个内存地址，而在别的线程中可能也会通过该静态变量对该内存地址的内容进行修改。
  - **解决方法：**
    - 重写函数，让调用者传递保存结果的地址
    - 使用**加锁-复制（Lock-and-Copy）**技术，将线程不安全函数与一个互斥锁联系起来，用 `P` 和 `V` 函数将该不安全函数的调用部分包围起来，并将其结果保存在一个局部变量中。**问题：**需要引入额外的同步操作，会降低程序的速度，其次是有一些函数会返回十分复杂的数据结构，比如 `getaddrinfo`，此时就需要深拷贝才能复制整个结构数据。

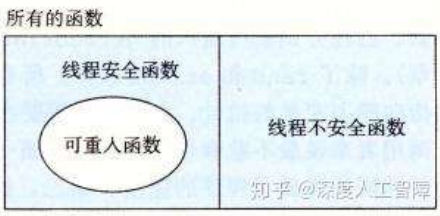
code/conc/ctime-ts.c

```
1  char *ctime_ts(const time_t *timep, char *privatep)
2  {
3      char *sharedp;
4
5      P(&mutex);
6      sharedp = ctime(timep);
7      strcpy(privatep, sharedp); /* Copy string from shared to private */
8      V(&mutex);
9      return privatep;
10 }
```


code/conc/ctime-ts.c

- **第四类：调用线程不安全函数的函数：**如果该线程不安全函数属于第二类，就要修改函数的源码，如果是第一类或第三类，可以直接通过互斥锁的方式保证函数时线程安全的。

Linux系统提供的大多数线程不安全函数都有对应的线程安全版本，一般是以 `_r` 结尾的。



我们可以更进一步得到线程安全函数中的一类**可重入函数（Reentrant Function）**，当他们被多个线程引用时，不会引用任何共享数据。可将其分成两类：

- **显示可重入的（Explicitly Reentrant）：**函数参数全都是值传递，且函数内的数据应用都是局部变量，不含有全局变量或静态变量。
- **隐式可重入的（Implicitly Reentrant）：**函数参数可以是引用传递的，如果调用线程传递一个非共享数据的指针，它就是可重入的。

可重入函数包括显示可重入和隐式可重入，说明可重入函数可能也依赖于调用线程的属性，比如传递一个非共享数据的指针到隐式可重入函数，保证其是可重入的，这类函数就不需要任何同步技术来保证多线程结果的正确。由于同步的开销很大，所以尽量写成可重入的形式。

上面第二类线程不安全函数，只能通过修改函数使其是线程安全的，并且由于无法使用互斥锁，所以只能将其改为可重入的

code/conc/rand-r.c

```
1  /* rand_r - return a pseudorandom integer on 0..32767 */
2  int rand_r(unsigned int *nextp)
3  {
4      *nextp = *nextp * 1103515245 + 12345;
5      return (unsigned int)(*nextp / 65536) % 32768;
6  }
```


code/conc/rand-r.c

注意：可重入函数是线程安全函数的真子集[+]，所以可重入函数一定是线程安全函数，而线程安全函数不一定是可重入函数，可能是用互斥锁来实现线程安全的。可重入函数由于不用同步操作，所以通常速度会更快。

3.2 竞争

多线程程序要求对进度图中任何可行的轨迹都正确工作，而程序员可能会下意识假设了某种特殊的轨迹，比如一个程序的正确性依赖于线程A要在线程B执行到y之前先执行到x，此时就会出现**竞争（Race）**。

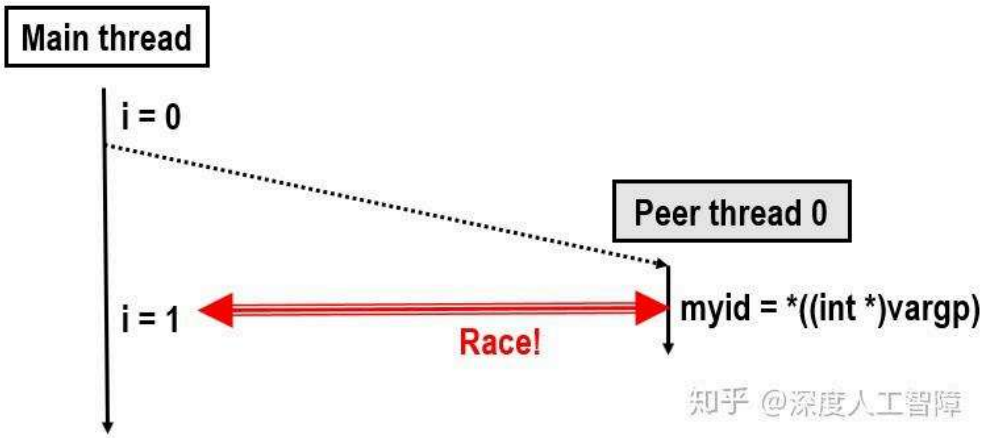
这个例子之前也说过

code/conc/race.c

```
1  /* WARNING: This code is buggy! */
2  #include "csapp.h"
3  #define N 4
4
5  void *thread(void *vargp);
6
7  int main()
8  {
9      pthread_t tid[N];
10     int i;
11
12     for (i = 0; i < N; i++)
13         Pthread_create(&tid[i], NULL, thread, &i);
14     for (i = 0; i < N; i++)
15         Pthread_join(tid[i], NULL);
16     exit(0);
17 }
18
19 /* Thread routine */
20 void *thread(void *vargp)
21 {
22     int myid = *((int *)vargp);
23     printf("Hello from thread %d\n", myid);
24     return NULL;
25 }
```


code/conc/race.c

该程序中，第13行创建线程时传入的参数为 `&i` ，而在线程例程中通过 `*((int*)vargp)` 将其转化为 `int` ，此时就会出现竞争。因为主线程中的局部变量 `i` 保存在主线程的栈中，而传入的线程例程的参数是 `i` 的地址，在前一个线程在第22行读取该地址中的 `i` 之前，第12行修改了该地址内的数据，就会造成竞争。这里假设了第22行在第12行之前执行。



正确的方法是申请一个独立的空间

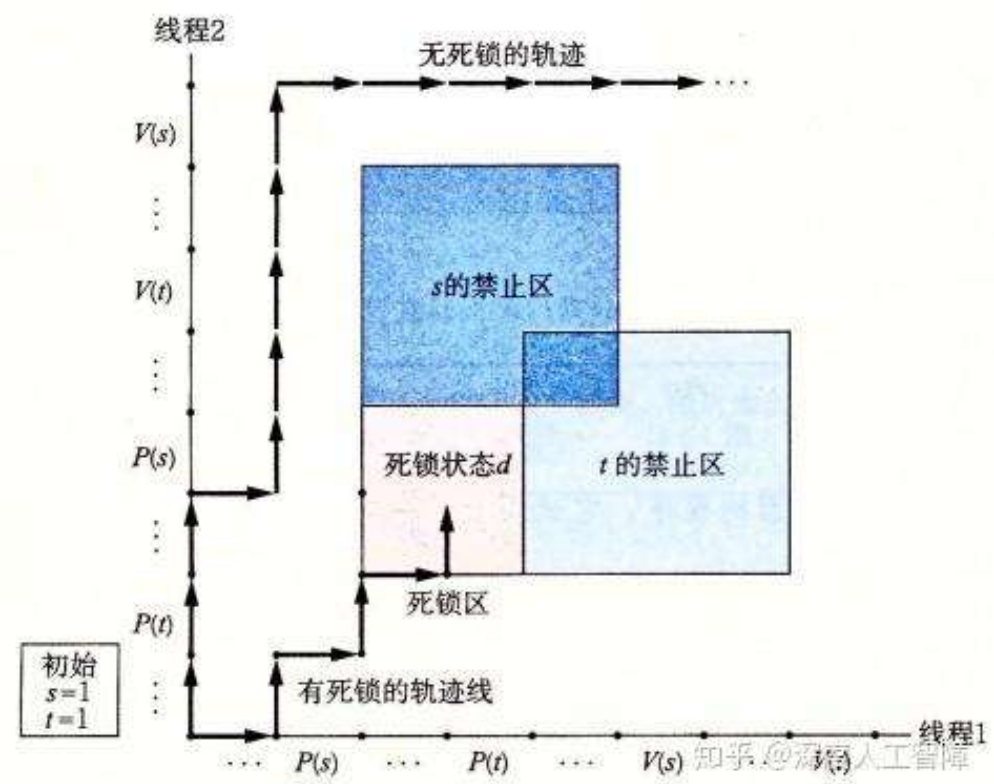
code/conc/norace.c

```
1   #include "csapp.h"
2   #define N 4
3
4   void *thread(void *vargp);
5
6   int main()
7   {
8       pthread_t tid[N];
9       int i, *ptr;
10
11      for (i = 0; i < N; i++) {
12          ptr = Malloc(sizeof(int));
13          *ptr = i;
14          Pthread_create(&tid[i], NULL, thread, ptr);
15      }
16      for (i = 0; i < N; i++)
17          Pthread_join(tid[i], NULL);
18      exit(0);
19  }
20
21  /* Thread routine */
22  void *thread(void *vargp)
23  {
24      int myid = *((int *)vargp);
25      Free(vargp);
26      printf("Hello from thread %d\n", myid);
27      return NULL;
28  }
```


code/conc/norace.c

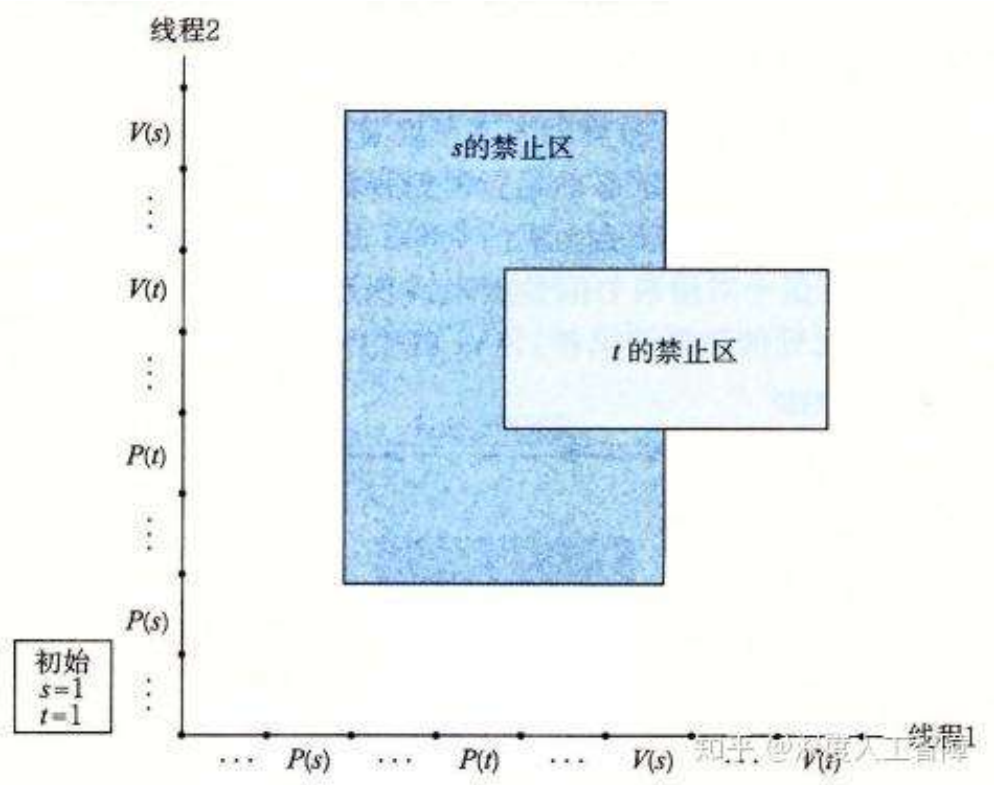
3.3 死锁

死锁（Deadlock）就是一组线程被阻塞，等待一个永远不会为真的条件。比如下图是一个使用了两个互斥锁的进度图



我们可以发现，当轨迹进入图中所示的**死锁区（Deadlock）**时，线程1占用互斥锁 s ，线程2占用互斥锁 t ，而轨迹要从死锁区出来，要么线程1占用互斥锁 s ，要么线程2占用互斥锁 t ，这两个都是不可能为真的条件，所以就陷入了死锁状态。所以轨迹可以进入死锁区，但是无法从死锁区出来。

可以发现是由于P和V的操作顺序不当，使得两个互斥锁的禁止区出现了重叠，导致死锁区的出现，这种原因导致的死锁常用的解决方法是定义一个**互斥锁加锁顺序规则**，给定所有互斥操作的一个全序，如果每个线程都是以一种顺序获得互斥锁并以相反的顺序释放锁，则该程序时无死锁的。也就是对于有多个互斥锁的情况，我们在所有线程中按照相同顺序来上锁，在解锁时用相反顺序。



注意：要注意互斥锁的初始状态。