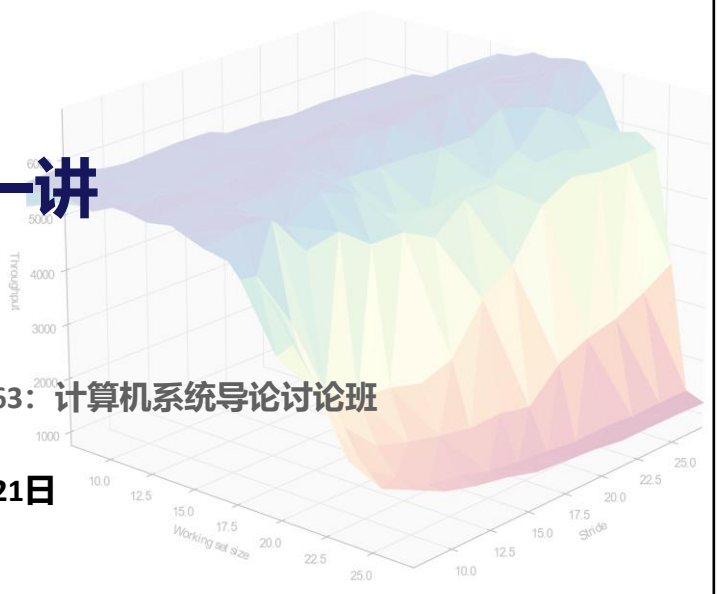
第十一讲



PKU 04832363：计算机系统导论讨论班

王畅

2021年12月21日

Activity 1

轮流回答问题

进程和线程

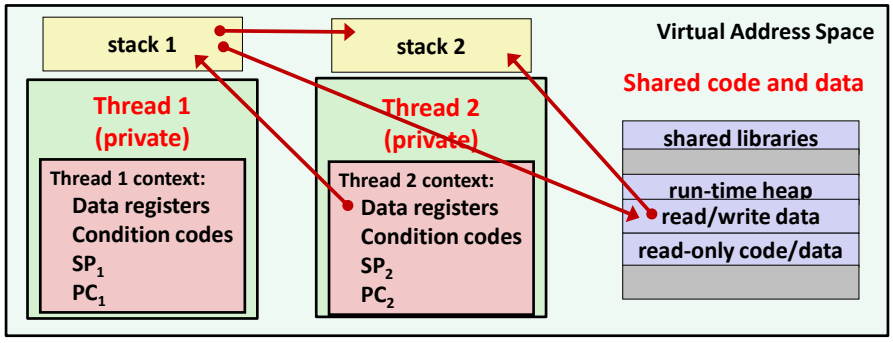
- **动机：网络服务器**
  - 比较基于进程、线程和I/O多路复用的服务器
  - 最后一个了解即可，但要读过
- **进程：运行中的程序实例， CPU+ 内存的状态（进程上下文）**
  - 地址空间相互独立，难以实现共享，切换开销大
- **线程：“轻量级”进程，某个进程上下文的一组控制流（仅CPU 的状态）**
  - 共享地址空间等进程上下文，切换开销小
  - 高度的共享引起竞争、死锁等同步问题，在线程模型中这个问题比进程的时候要更多更大。

3

I/O多路复用的服务器的本质是一个状态机。在不同的体系结构中，线程实现的方式不同（有一些是用户级的，有一些可能是内核级的，有一些可能是混合机制）。在POSIX中是用户级的。

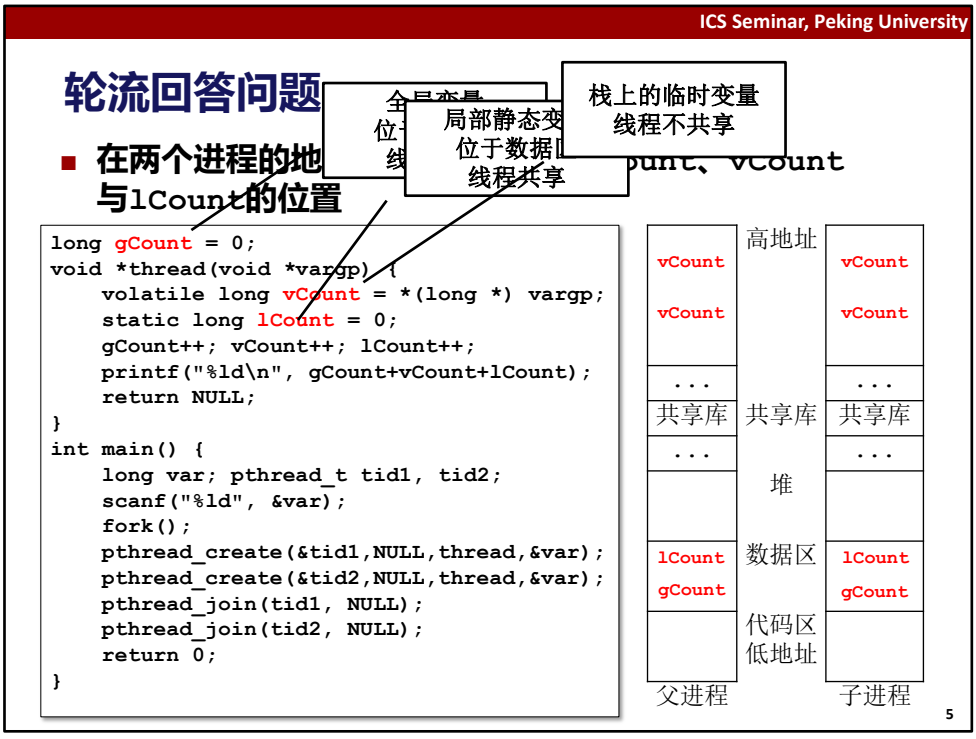
哪些内容共享

- **栈是私有的，但不是严格私有的（会带来问题）**
- **共享变量：当且仅当超过1个对等线程引用**
  - 和全局变量、静态变量、局部变量没有必然关系
  - 结合地址空间来理解！



4

如果某个对等线程持有对方栈的指针，那么栈就不是严格私有的。这里，只要能通过指针的方式访问到，那么就是引用了。而全局变量也可能只有一个对等线程引用。因此都是需要看具体代码的。



答案已经在上面示出。请注意volatile的局部变量是在栈上保存的。另外，书上所说的“自动变量”就是auto关键字，但是在C语言中auto的含义和C++不同（C++中一般是自动类型推导的意思），它是volatile的反义，一般不加特别修饰的局部变量都是自动的。

轮流回答问题

- **对等线程1引用了哪些变量?**

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    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);
}
void *thread(void *vargp) {
    long myid = (long) vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```


请注意，间接通过指针引用对方栈内的内容也算引用。所以引用了`myid`（自己的那个）、`cnt`、`ptr`和`msg`（但没有引用`i`，因为它是传值的！）。

POSIX线程

- **线程的大多数接口功能和进程系统调用相似**
  - 区别：所有的线程都是同级关系，因此它们彼此都能互相创建、互相回收（**对等**）

功能	线程	进程	区别
创建	pthread_create	fork	创建者和被创建的关系；线程可以通过参数指定逻辑流的入口，进程需要根据返回值进行设定
取id	pthread_self	getpid	
终止	pthread_exit/pthread_cancel	exit	任一线程执行exit会终止所有对等线程；线程间提供了直接终止某个对等线程的接口，而进程需要通过信号机制
回收	pthread_join	wait(pid)	前者只能等待特定线程终止
分离	pthread_detach		结束时能自动释放资源，不用专门阻塞其他某个线程；如果我们不在乎某个线程的终止状态和返回值，就可以没有负担地分离它；分离后不能被回收和杀死

7

Windows的线程API和模型和Unix是不同的。

轮流回答问题

- **下面程序中，Pthread_detach函数的作用是**
  - A. 使主线程阻塞以等待线程thread结束。
  - B. 线程thread运行结束后会自动释放所有资源。
  - C. 线程thread运行后主动释放CPU给其他线程。
  - D. 线程thread运行后成为僵尸线程。

```c
#include "csapp.h"
void *thread (void *arg) {
    printf("Hello World");
    Pthread_detach(pthread_self());
}
int main(void) {
    pthread_t tid;
    int sta;
    sta = Pthread_create(&tid, NULL, thread, NULL);
    if (sta == 0)
        printf("Oops, I can not create thread\n");
    exit(NULL);
}
```

8

B。注意如果不detach就要设法显式回收，否则造成内存泄漏。

考点1：竞争问题

- **线程执行的顺序是不确定的**
  - 对比/类比进程、信号处理程序。不同：**对等**!
- **原因一：共享变量**
- **原因二：原子性问题（导致更多问题）**
  - 一行操作大多数都不是原子的
  - 会在中间被打断，导致出现错误
  - 典型例子为全局变量`i`：`i++`翻译为`mov(load), add(use), mov(store)`
- **分析方法：进度图上的格点路径**
  - 和ECF中进程树的分析方法有什么不同？为什么？
  - 禁止区和不安全区

9

大体上说，原因一的分析方法就是各条指令之间执行顺序可以不同（仍然是拓扑排序，只不过这时是有多个连通分量的图）；原因二的分析方法就是指令内部还有细节，可能被打断（这时分析就复杂一些，需要比较灵活）。

和ECF中进程树的分析方法不同的地方是线程是对等的，进程有父子的区别（树vs森林）。但思想是一样的。

进度图一般用于一对线程，其中的执行曲线只能向上或者向右。禁止区是一个“闭集”含边界，是指进入就必然保证出错的位置；不安全区是一个“开集”，是指在其中就已经出错的位置。

轮流回答问题

```
        movq    (%rdi), %rcx          ┐
        testq   %rcx, %rcx            │ Hi
        jle     .L2                   │
        movl    $0, %eax              ┘
.L3:
        movq    count(%rip), %rdx     Li
        addq    $2, %rdx              Ui
        movq    %rdx, count(%rip)     Si
        addq    $1, %rax              ┐
        cmpq    %rcx, %rax            │ Ti
        jne     .L3                   ┘
.L2:
```

上面用 H_i, L_i, U_i, S_i, T_i $(i = 1, 2)$ 划分了指令的几个部分，下标代表执行的线程。在下列指令顺序对应的轨迹线中，哪一个是安全轨迹线？

A. $H_1 \to H_2 \to L_2 \to L_1 \to U_2 \to U_1 \to S_1 \to S_2 \to T_1 \to T_2$

B. $H_1 \to L_1 \to U_1 \to H_2 \to L_2 \to S_1 \to T_1 \to U_2 \to S_2 \to T_2$

C. $H_2 \to L_2 \to U_2 \to H_1 \to S_2 \to L_1 \to T_2 \to U_1 \to S_1 \to T_1$

D. $H_2 \to L_2 \to H_1 \to L_1 \to U_1 \to U_2 \to S_2 \to T_2 \to S_1 \to T_1$

10

C。这里的不安全问题就是LUS必须一步执行到位，否则出错（一个线程的S必须在另一个线程L之前）。所以如果存在L_i，U_i被打断给另一边的L_i，U_i，就不是安全的。A、B、D都出现了这样的问题。

轮流回答问题

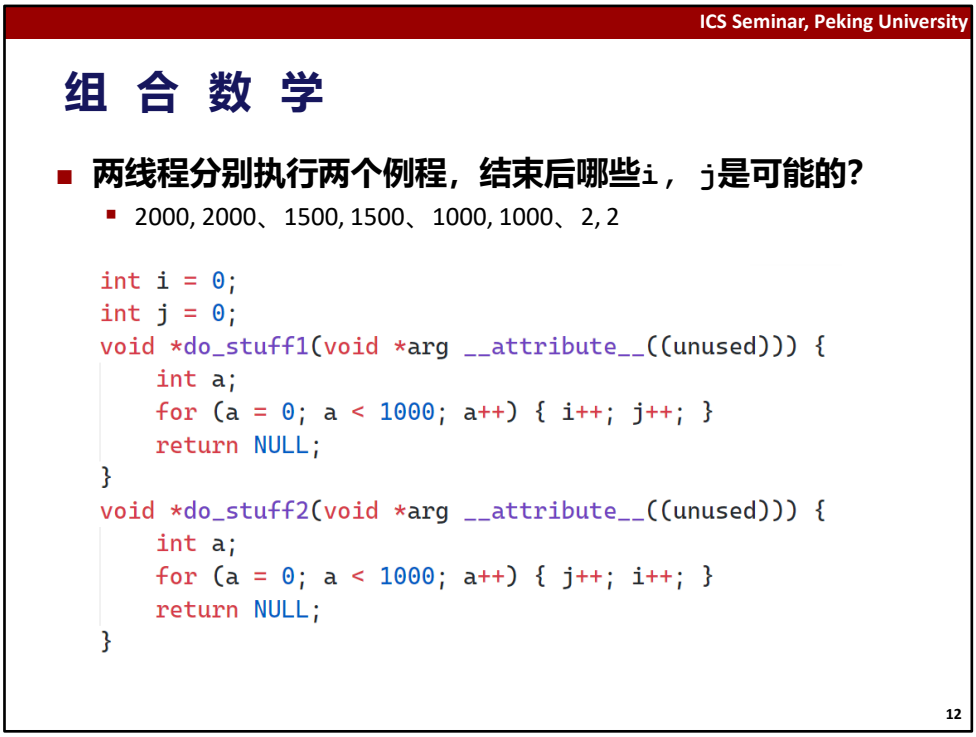
- 不考虑原子性问题，一个可能的输出结果是2 1 2 2，解释如何得到之

```
long foo = 0, bar = 0;

void *thread(void *vargp) {
    foo++;
    bar++;
    printf("%ld %ld ", foo, bar);
    fflush(stdout);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```


线程1将foo、bar改为1以后被线程2打断，线程2将foo改为2以后被线程1打断，线程1输出了2 1，线程2将bar改为2，并输出了2 2。这里注意先从简单的情况考虑，暂时不要考虑原子性问题。



这个题严格讲是错题，因为由于编译器和体系结构的原因，实际上可能少加非常多次。具体做起来分析如下：

2000, 2000当然是可能的（就是没有原子性问题的情况）

两个线程实际上执行了如下的

i++, j++, i++, j++, …

j++, i++, j++, i++, …

的序列。为什么会出现不足值的情况？就是说某个进程在执行load i, i++, store i的间隙中，另一个线程对i的操作都是全部无效的。我们可以构造1500, 1500的情形，比如线程1读出i后被2打断，2进行了500次j++和i++，此时i=500, j=500。然后1写回，得到i=1, j=500。此时1再读j，又被打断，2进行了500次的j++和i++，得到i=501, j=1000。然后1写回，得到i=501, j=501。这时2已经结束，1还有i++和j++各999个，所以得到1500, 1500。（官方答案构造的例子疑有误）

让我们来考虑1的某个i++被打断的情形，我们知道最坏情况是2完成了n次i++，然后都无效了，这中间至少会有n-1次的j++被完成，而

补上1的打断的那个i++，我们发现损失的自增最多不超过有效的自增数目。所以2，2不可能。另外注意到开头的i++和j++至少有一个完全有效（也不会被用来抵消别的损失），结尾的也是这样（反证即可），所以i+j至少是2000-1+2=2001。所以1000，1000不可能。但是1000，1001是可能的，官方答案在这里疑有误。

PV操作

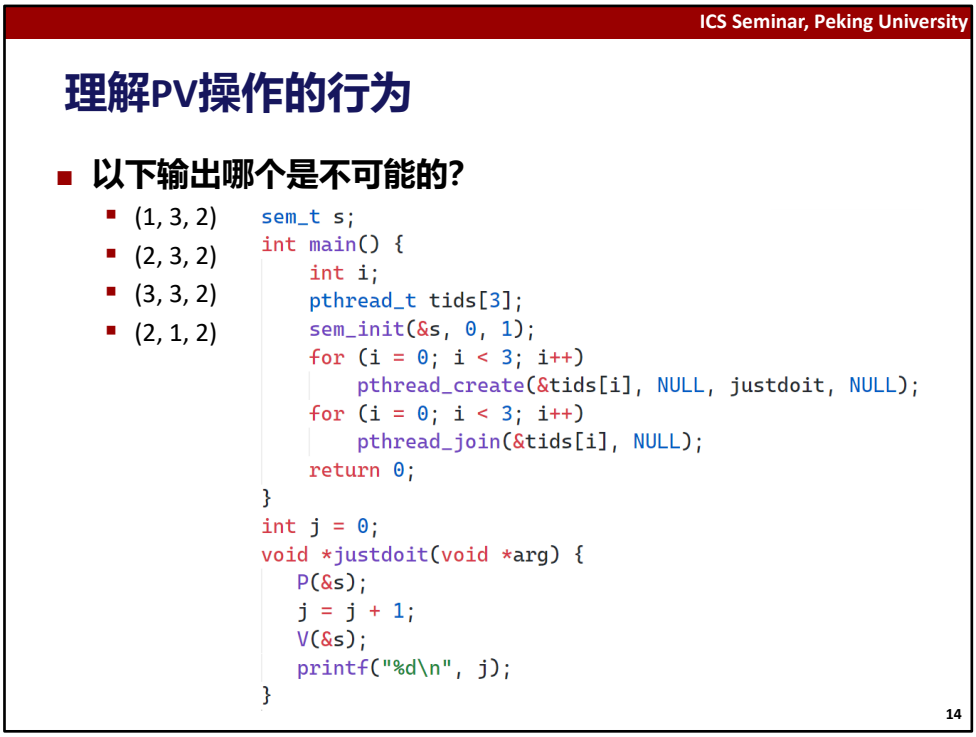
- **并发中的同步问题有多种方法解决**
  - 信号量+PV操作是一种重要的原语机制，但还有其他方法
  - 荷兰语（Dijkstra），passeren（pass, wait）、vrijgeven（release, signal）
- **PV操作**
  - 机理非常简单
  - 必须是原子操作，实现也有多种办法
  - `P(&s) while(s == 0) { wait(); } s--;`
  - `V(&s) s++;`

- **理解与使用**
  - 信号量可以理解为资源计数，一般用时要求非负
  - P理解为：等待非负；V理解为：加一
  - `sem_init`函数

13

信号量的本质原理是保证非负，这样就不会进入禁止区。

PV操作有两种理解，第一种是`wait`和`signal`，类似于通信（在同步时可以采用这种思考方式）；第二种就是资源计数（在互斥时可以采用这种方式，但是这和同步其实也是一样的）。`sem_init`的第二个参数要注意，它一般都是`0`，表示信号量是线程级共享（而不是进程级共享），第三个参数是初始化值。



D。信号量的操作导致j的增加是原子的，也就是说实际上是3个j++和3个printf（分两步）拓扑排序。这里我们需要假设printf是先读取j，然后再输出，所以结果不一定是单调递增的。但无论如何，最后的输出结果一定会有一个是3，因此D不可能。

考点2：PV同步机制设计

- **第一步：识别互斥关系**
  - 同时只能提供给一个进程使用
  - 访问临界资源需要上锁和解锁过程（0/1信号量或mutex）
  - 互斥PV在同一进程中成对出现
- **第二步：识别同步关系（可画图）**
  - 协调两个进程之间的工作次序。例如，A向缓冲区写数据，缓冲区中写满数据只后，B才能做读操作
  - ①一个节点运行完成后，对该节点后继的所有节点都进行一次V操作；②一个节点运行前，对它前驱的节点进行一次P操作
  - 同步PV在一对进程中成对出现（有一点点像信号）
- **每个临界资源一个信号量，每对同步关系一般要两个信号量（也可能更多或者更少）**
  - 掌握经典同步问题的解决方案

15

同步关系就是“先穿袜子再穿鞋”（先后顺序），互斥关系就是“不能两个人同时进入一个卫生间隔板”（独占资源）。大体上也是不需要画图的，但是需要从题目的实际描述中识别出来。

除了互斥关系和同步关系外，由于编程需要得到的那些共享变量也需要保护，所以会有更多需要（一个典型例子是读者写者问题）。

几个经典同步问题

- **生产者消费者问题**
  - 互斥：对缓冲区的访问
  - 同步：满了就不能放，空了就不能取
  - 所以需要三个信号量
  - 看P706页实现。思考有一些PV操作能不能交换？比如24、25行的P操作能不能交换？27、28行的V操作能不能交换？
- **读者写者问题**
  - 有两种，读者和写者谁优先。以第一类为例
  - 互斥：对读者数目的访问和修改
  - 互斥：读者到0时写者才能进去
  - 看P707页实现，特别注意对于第二个互斥锁的写法！
  - 第二类类似，但是读者要先请求无写者的互斥锁
  - 可能导致**饥饿**问题

16

P操作不可以交换，一般要求同步在前，互斥在后，否则可能发生死锁（可以自己构造一下情况）。此外，加锁和释放的顺序最好相反，确保无死锁（对于二元信号量）。一般来说，单方面修改V的顺序是没有问题的，因为V不会导致阻塞。

死锁问题

- **一个比较复杂的问题**
  - 指多个进程循环等待它方占有的资源而无限期僵持下去的局面（重点考察P操作是不是有成对反序的状况）
  - 在进度图中表现为一个区域，进入之后就会被禁止区全部挡住

- **二元信号量（mutex）解锁按相反顺序可确保无死锁**
  - 更深入的内容例如“死锁定理”可以在操作系统课上学到
  - 许多现代操作系统并不尝试避免死锁，而是采用死锁检测和解除的算法（否则太影响性能）

17

发生死锁的四个必要条件：互斥，某种资源一次只允许一个线程访问，即该资源一旦分配给某个线程，其他线程就不能再访问，直到该线程访问结束。占有且等待，一个线程本身占有资源，同时还有资源未得到满足，正在等待其他进程释放该资源。不可抢占，别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。循环等待，存在一个线程链，使得每个进程都占有下一个进程所需的至少一种资源。

这里比较重要的是互斥和循环等待。

例子

- **哲学家就餐问题**
  - 假设有五位哲学家围坐在一张圆形餐桌旁，做以下两件事情之一：吃饭，或者思考。吃东西的时候，他们就停止思考，思考的时候也停止吃东西。餐桌上有五碗意大利面，每位哲学家之间各有一只餐叉。因为用一只餐叉很难吃到意大利面，所以假设哲学家必须用两只餐叉吃东西。他们只能使用自己左右手边的那两只餐叉。
- **会导致死锁的解决方案**
  - 如果左边的叉子可用，就拿起来。
  - 哲学家等待右边的叉子可用。如果右边的叉子可用，就拿起来。
  - 如果两个叉子都已经拿起来，开始吃意大利面。
  - 吃完后先放下左边的叉子。然后放下右边的叉子。
  - 开始思考（进入一个循环）。

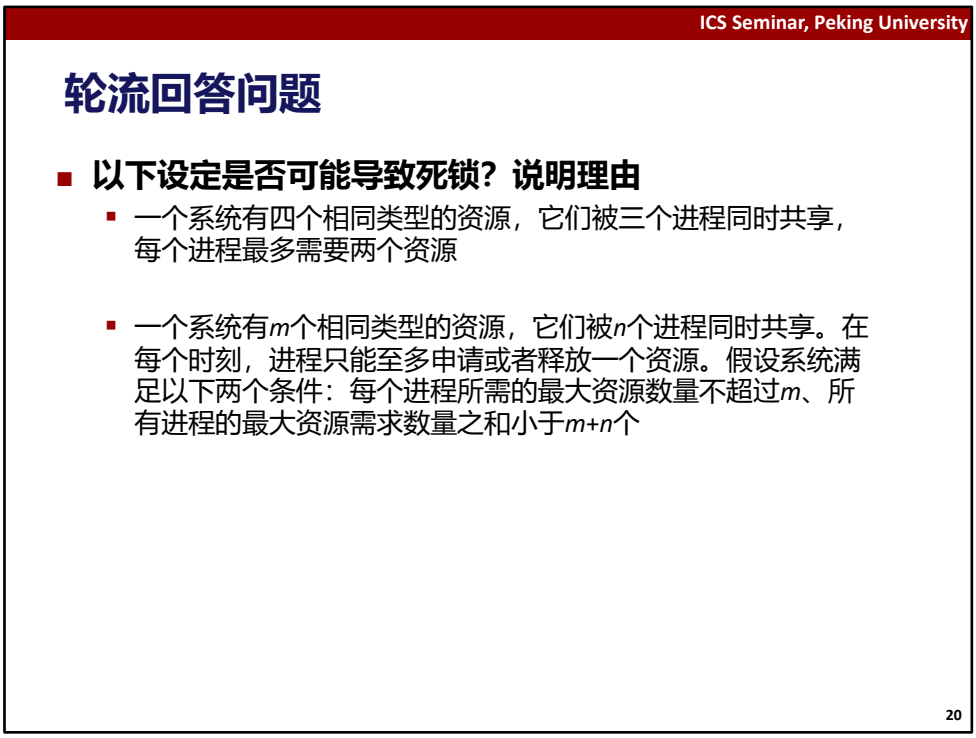
哲学家就餐问题的解决方案一般稍有复杂，超出了本课程的范围（一个自然的想法是分优先级）。可自行查阅。

轮流回答问题

- **信号量都初始化为1。以下哪两个线程并发时一定无死锁?**

进程 1	进程 2	进程 3
P(a)	P(d)	P(d)
P(d)	P(a)	P(c)
P(c)	P(c)	P(b)
P(b)	P(b)	P(a)
V(c)	V(d)	V(c)
V(b)	P(d)	V(b)
V(d)	V(a)	V(a)
V(a)	V(b)	V(d)
	V(c)	
	V(d)	

1，2和1，3都存在a，d的反序，会出现死锁。2，3的死锁出现在c，d的反序，2第二次要P(d)时还持有c的锁，但是3在获得c的锁之前不会释放d。所以都是可能发生死锁的。



都不会。假设1出现死锁，那么4个资源耗尽，而且每个进程都想要请求资源，但至少有一个进程有2个资源，矛盾。类似，假设2出现死锁，则全部资源都被消耗，而且每个进程都在请求资源，这样所需求的资源不少于m+n个，矛盾。

线程安全

- **是指被任意多个并发线程同时调用不会出现问题的例程**
- **以下都是不安全的**
  - 不保护共享变量
  - 状态在多个调用下相互关联（随机数生成器）
  - 返回指向静态变量的指针
  - 调用不安全的函数
- **可重入函数（一定线程安全）**
  - 严格定义之前讲过
  - 技术上就是不引用任何共享变量
  - 一般来说比不可重入的高效（请注意如果不需要同步则另论）
  - 不可重入：printf“灵魂出窍”的例子

21

If a function is reentrant with respect to multiple threads, we say that it is thread-safe. This doesn't tell us, however, whether the function is reentrant with respect to signal handlers. We say that a function that is safe to be reentered from an asynchronous signal handler is async-signal safe. 也就是说，（不论何时都）可重入的函数包含于信号安全函数，信号安全函数包含于线程安全函数。

可重入有两种，显式和隐式。伪随机数生成器的一种改写方法是采用传递指针的方式来处理共享问题。这样并不能确保可重入，就是说它不是显式可重入的，而是隐式可重入的，也就是说，如果传入一个局部变量的指针来表示随机数发生器的状态，那么这就是可重入的、安全的；但如果那个指针指向共享变量，那当然还是不行的。

printf不可重入当然是因为有全局缓冲区的缘故。如果stdout有锁，那么printf是线程安全的。这里说的灵魂出窍，是指这时它仍然不是信号安全的，如果用锁保护这个全局缓冲区，printf执行到一半（已经取得锁）被中断之后，信号处理程序再次执行printf时需要这个

锁，会构成死锁（因为只有执行完信号处理程序后才可能释放这个锁）。

轮流回答问题

- **判断以下说法的正确性：**
  - 如果一个函数的所有参数都是值传递的且无返回值，该函数一定是可重入的。
  - 函数的可重入版本一定比不可重入版本高效。
  - 可重入函数一定是线程安全的。

22

1不对，因为可以访问共享变量。2不对，如果不需要同步则不一定。3是对的，可重入函数真包含于线程安全的函数（举一个不可重入但线程安全的函数的例子？）。

Activity 2

问题求解

同步部分2，习题13

- **是否有类似的经典问题可以参考?**
  - 没有
- **有没有互斥关系?**
  - 没有
- **有没有同步关系?**
  - 停车之后才能开门
  - 关门之后才能开车

- **两个信号量：开车信号量（初始化为0）和开门信号量（初始化为0）**

同步部分2，习题13解答

P(开车信号量)
启动车辆
正常行驶
到站停车
V(开门信号量)

关门
V(开车信号量)
报站名或维持秩序
P(开门信号量)
到站开门

狒狒过峡谷

- **双读者问题**
  - 一个主修人类学、辅修计算机科学的学生参加了一个研究课题，调查是否可以教会非洲狒狒理解死锁。他找到一处很深的峡谷，在上边固定了一根横跨峡谷的绳索，这样狒狒就可以攀住绳索越过峡谷。
  - 同一时刻，只要朝着相同的方向就可以有几只狒狒通过。但如果向东和向西的狒狒同时攀在绳索上那么会产生死锁（狒狒会被卡在中间），由于它们无法在绳索上从另一只的背上翻过去。如果一只狒狒想越过峡谷，它必须看当前是否有别的狒狒正在逆向通行。利用信号量编写一个避免死锁的程序来解决该问题。不考虑连续东（西）行的狒狒会使得西（东）行的狒狒无限制地等待的问题。

狒狒过峡谷

- **是否有类似的经典问题可以参考?**
  - 读者写者问题
- **有没有互斥关系?**
  - 绳子按**方向**的使用权（不是狒狒）
- **有没有同步关系?**
  - 没有
- **至少一个信号量：绳子的互斥**
  - 是否还需要信号量？考虑到同向狒狒可以不断过来，绳子使用权要根据狒狒数目加锁和释放
  - 需要两个计数量，所以还要两个信号量

27

一般来说更多信号量是需要具体写下来之后才意识到的。

类比经典问题

```
void reader()
{
    while(true)
    {
        P(mutex);
        rc = rc + 1;
        if (rc == 1)
            P(w);
        V(mutex);
        /* reading... */
        P(mutex);
        rc = rc - 1;
        if (rc == 0)
            V(w);
        V(mutex);
    }
}
```

```
void writer()
{
    while(true)
    {
        P(w);
        /* writing... */
        V(w);
    }
}
```

w作为0-1互斥信号量来实现读写互斥
第一个读者和写者竞争这个信号量

解答

```
void Westward()
{
    P(W);
    if (CW == 0)
        P(mutex);
    CW = CW + 1;
    V(W);

    cross();

    P(W);
    CW = CW - 1;
    if (CW == 0)
        V(mutex);
    V(W);
}
```

```
void Eastward()
{
    P(E);
    if (CE == 0)
        P(mutex);
    CE = CE + 1;
    V(E);

    cross();

    P(E);
    CE = CE - 1;
    if (CE == 0)
        V(mutex);
    V(E);
}
```

同步部分2，习题19

- **有没有经典问题可以参考?**
  - 是二重生产者消费者问题
- **有没有互斥关系?**
  - 两个缓冲区均互斥
- **有没有同步关系?**
  - 两个缓冲区的空和满

- **六个信号量，两个互斥，两组同步（空/满）。空信号量初始化为容量（4、8），满信号量初始化为0**

同步部分2，习题19解答

从磁盘读入一个记录 P(b1_empty) P(b1_mutex) 将记录放入Buff1 V(b1_mutex) V(b1_full)	P(b1_full) P(b1_mutex) 从Buff1中取出一个记录 V(b1_mutex) V(b1_empty) P(b2_empty) P(b2_mutex) 将记录放入Buff2 V(b2_mutex) V(b2_full)	P(b2_full) P(b2_mutex) 从Buff2中取出一个记录 V(b2_mutex) V(b2_empty)

31

务必注意P的顺序。V的顺序一般不是非常要紧。

PV大题

- **如果感觉还是不熟练的话**
  - 可以去看操作系统相应的PPT
  - 搜索关键词“操作系统 PV大题”
- **但期末考试不一定考这些**
  - 有可能考竞争（如2018）
  - PV大题操统一般还会反复考

any questions?

Thanks & 感谢观看