一、第三章的计算有点多（ＰＶ操作，银行家算法，低级调度算法）

ＰＶ操作

操作系统的进程管理中，PV是重点和难点

信号量：信号量是个数据结构。

struct semaphore{int value;pcb \*blockqueue;}mutex;

其中value是信号量的值；blockqueue是等待使用该信号量的进程排成的队列的对手指针。

p操作：当一个进程对信号量mutex执行p操作时，执行两个动作：

mutex.valu–； //申请一个资源

if (mutex.value<0) //申请失败

sleep(); //本进程进入该信号量等待队列睡眠

v操作：当一个进程对信号量mutex执行v操作时，执行两个动作：

mutex.value++； //释放一个资源

if (mutex.value>=0) //如果有进程在等待信号量

wakeup(); //从该信号量的等待队列中唤醒一个进程

注：操作系统会保证PV操作的原子性，也就是说当一个进程执行PV操作，检测信号量时，不受中断。

看一下PV操作实现的功能：

实现进程之间的互斥； 实现进程之间的同步；（接下来的两个例题是互斥与同步的典型）

区别：互斥是为了保证资源一次只能由一个进程使用，互斥进程彼此在逻辑上是完全无关的，它们的运的运行不具有时间次序的特征。而同步是为了实现进程通信，同步进程之间具有合作关系，在执行时间上须按照一定顺序协同进行。

**1.互斥：**进出教室问题：有一个变量count，初值为0，一个学生进入教室则count++，出教室则count–-。

mutex = 1;

IN: OUT:

p(mutex); p(mutex);

count++; count–;

v(mutex); v(mutex);

过程：一个学生进入教室执行IN，p操作，mutex.value = 0；假设在进行count++之前遇到了中断，而中断之后跳回来时正好这个学生又在出教室，那么这时候就会执行OUT，mutex.value = -1，该OUT进程进入睡眠，返回IN进程，count = 1，v操作，mutex.value = 0（说明有等待使用count的进程）；唤醒OUT进程，count = 0，v操作，mutex.value = 1。

注意上面划线部分的假设。PV操作在这就是为了保证这种竞争情况的发生。

**2.同步：**桌上有一空盘，允许存放一只水果。爸爸可向盘中放苹果，也可向盘中放桔子，儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定当盘空时一次只能放一只水果供吃者取用，请用P、V原语实现爸爸、儿子、女儿三个并发进程的同步。

分析：在本题中，爸爸、儿子、女儿共用一个盘子，盘中一次只能放一个水果。当盘子为空时，爸爸可将一个水果放入果盘中。若放入果盘中的是桔子，则允许儿子吃，女儿必须等待；若放入果盘中的是苹果，则允许女儿吃，儿子必须等待。本题实际上是生产者-消费者问题的一种变形。这里，生产者放入缓冲区的产品有两类，消费者也有两类，每类消费者只消费其中固定的一类产品。

解：在本题中，应设置三个信号量S、So、Sa，信号量S表示盘子是否为空，其初值为l；信号量So表示盘中是否有桔子，其初值为0；信号量Sa表示盘中是否有苹果，其初值为0。同步描述如下：

int S＝1;int Sa＝0;int So＝0;

main()

{cobegin

father();

son();

daughter();

coend

｝

father()

{while(1)

{P(S);

将水果放入盘中;

if（放入的是桔子）V(So);

else V(Sa);

}

}

son()

{while(1)

{P(So);

从盘中取出桔子;

V(S);

吃桔子;

｝

}

daughter()

{while(1)

{P(Sa);

从盘中取出苹果;

V(S);

吃苹果;

｝

｝

一些问题：为什么要设计三个信号量？因为这里盘子的状态有三种情况。所以在PV操作用在同步的时候，资源都多少种状态，就应该有多少个信号量（高并发的不一定好，需要更多的信号量，这样消耗系统的资源就更多）。还有，有没有留意到，每一次“吃”的操作都是在V操作之后进行，这是为什么呢？这是因为V操作是释放资源的一个操作，当然是越早释放对系统越有利啊。

哲学家就餐问题

五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一把叉子；

解一：

设fork[5]为5 个信号量，初值为均1

设信号量S ，用于封锁第5个哲学家，初值为4。

Philosopheri：

while (1)

{ 思考;

P(S);

P(fork[i]);

P(fork[(i+1) % 5]);

进食;

V(fork[i]);

V(fork[(i+1) % 5]);

V(S);

}

解二：

设fork[5]为5 个信号量，初值为均1。

Philosopher1：

while (1)

{ 思考;

P(fork[1]);

P(fork[2]);

进食;

V(fork[2]);

V(fork[1]);

}

Philosopher2：

while (1)

{ 思考;

P(fork[3]);

P(fork[2]);

进食;

V(fork[2]);

V(fork[3]);

}

银行家算法

* 练习：有三类资源A(17)、B(5)、C(20)。有5个进程P1～P5。T0时刻系统状态如下：

|  |  |  |
| --- | --- | --- |
|  | * 最大需求 | * 已分配 |
| * P1 | * 5 5 9 | * 2 1 2 |
| * P2 | * 5 3 6 | * 4 0 2 |
| * P3 | * 4 0 11 | * 4 0 5 |
| * P4 | * 4 2 5 | * 2 0 4 |
| * P5 | * 4 2 4 | * 3 1 4 |

问(1)T0时刻是否为安全状态，给出安全系列。

(2)T0时刻，P2: Request(0,3,4)，能否分配，为什么？

(3)在(2)的基础上P4:Request(2,0,1)，能否分配，为什么？

(4)在(3)的基础上P1:Request(0,2,0)，能否分配，为什么？

解析：（1）（求安全系列：说的通俗点就是把所有剩下的资源分给进程，但是要看剩下的资源数是否能满足进程的需求量，满足了就分给它，等它结束后释放它所拥有的所有资源，再继续往下分，直到所有的进程都运行成功。）

1>先求剩余量**A(17)、B(5)、C(20) ————**三个资源的各自的总量

**Work=2 3 3————**分给每个进程后三个资源所剩下的量{这样子算的：

A（17）-2-4-4-2-3=2，即总量减去已分配里边竖起来相加的和等于剩余}

2>再求每个进程T0时刻的所需量：Need=最大雪球—已分配

综上可得到如图：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 最大需求 | 已分配 | **Need** |
| P1 | 5 5 9 | 2 1 2 | **3 4 7** |
| P2 | 5 3 6 | 4 0 2 | **1 3 4** |
| P3 | 4 0 11 | 4 0 5 | **0 0 6** |
| P4 | 4 2 5 | 2 0 4 | **2 2 1** |
| P5 | 4 2 4 | 3 1 4 | **1 1 0** |

观察Need的那些值，**Work=2 3 3，**work能够need的有P4跟P5，二者选哪个都可以，比如就选择P4，即我们把work的值分给P4，让它运行完成，P4释放所有资源，则work的值就变为了 work=4 3 7（P4的所有资源都要释放哦），再继续找满足条件的进程以此类推，可得到安全系列:{P4,P2,P5,P3,P1}(**第一问在答题时只需写出一个安全系列就行了，且安全系列并不确定)**

（2）在(2)的基础上P2: Request(0,3,4)（P2的need（=1 3 4））

第一次比较：Request(0,3,4)<= need（=1 3 4）

第二次比较：Request(0,3,4) >=Available (=2 3 3) 因为

Request(0,3,4) >=Available (=2 3 3)

所以不能分配。{ Available (=2 3 3)就是work =2 3 3，不同的表达而已，但是第二问就是要这样子写才规范，Request(0,3,4)表示对三种资源的需求，所剩资源必须完全满足需求才可以分配，否则不行}

（3）P4:Request(2,0,1)

因为P2不能分得资源，推迟了，则现在所剩仍是Available (=2 3 3)，

Request(2,0,1)<=Need（=2 2 1）(第一条件满足了)

Request(2,0,1)<=Available (=2 3 3);（第二条件也满足），接下来还得进行安全性算法检查，先假定可以分配，若分配后得到我work（0 3 2），看这时的（0 3 2 ）是否能够让新的形成的序列达到安全，若是安全就分配，不安全就推迟，经计算是可以形成安全系列{P4，P5，P3，P2，P1}则可以分配。

（4）在(3)的基础上P1:Request(0,2,0)

在(3)的基础上，即P4运行完成释放了所有资源，这时候work的值发生了变化

Work=4 3 7（它是原来的work加上已分配得到的），

Request(0,2,0)<= Need（=3 4 7）;

Request(0,2,0) <= Available (=4 3 7),都满足，但是此时的work是上次分配后剩下的，即work（=0 3 2）,若再分配（0 2 0 ）出去，剩下的work（0 1 2）就不能再满足任何的进程需要了，所以不可以可以分配。

低级调度算法

低级调度算法老师提到的有三种｛ＦＣＦＳ：先来先服务；ＳＪＦ：短作业优先；ＨＰＦ（ＨＲＲＮ）：高响应比优先｝（缩写由来**FCFS：First Come First Serve；**

**SJ(P)F： Shortest Job (Process) First；　　　HPF：Highest Priority First**）

1. 先来先服务：顾名思义，先进来的先运行，运行完了其他后进来的再运行
2. 短作业优先：进程必须是到达之后，看其还需多少服务时间完成，所有到达的进程比较，所需服务时间少的先运行，一次类推不断的比较选出可以运行的，直到所有的都运行完了为止；
3. 高响应比优先（就是优先权）：(优先权=等待时间+要求服务时间)/要求服务时间；(或者优先权=响应时间/要求等待时间)，响应时间=要求等待时间+等待时间；
4. 周转时间=完成时间-到达时间 带权周转时间=周转时间/服务时间

用一个例题来具体说明下：



ＦＣＦＳ：先来先服务算法：



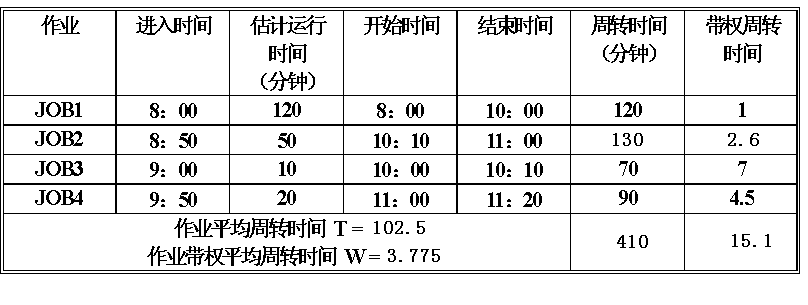
ＳＪＦ：短作业优先算法：

个



1. 根据每个进程进入的时间可知，在8:00的时候只有job1到达了固其开始时间是8:00，job1开始运行50分钟后，即8；50时刻，job2到达了，（因为一般都是非抢断方式的，所以必须让运行着的进程结束后再比较其他的进程），job1继续运行，十分钟后即9:00job3达到，直到10:00，job1运行完毕，期间其余进程都已经到达，则开始比较所有达到的进程的估计运行时间，job2（50分钟）>job4 （20）>job3(10)，则10:00的时候，job3开始运行，**一次类推就可以算出每个的开始时间与结束时间。**
2. **周转时间：**就是要用进程结束的时间减掉它到达的时间，如job2，结束时间11:20减去到达时间8:50，得到周转时间150分钟。
3. **带权周转时间：**用周转时间除于估计运行时间，如job2周转时间150分钟，除于估计运行时间50分钟，得到3

ＨＰＦ（ＨＲＲＮ）：高响应比优先



1. 在job1运行完成后，算其余进程的响应比，等待时间都是用相同的现在的时间减去进程各自到达的时间，job2的等待时间是10:00-8：50=70（分钟），则响应比R2=（70+50）/50=2.4，R3=7，R4=1.5；R3最大则先运行job2.以此类推得到上表中的开时时间和结束时间。周转时间和带全周转时间跟上边SJF的算法一样。

**一般这三种调度算法会比较来用，可能会出这样子的吧。时间片算法好像老师没有提到（要是谁记得老师提到了告诉我哈，我再弄下），至于实时调度算法老师说不会考。**

**第四章**

**第一种计算是分区的计算**

·动态分区分配算法：

**（1）FF——首次适应法(first-fit)：**要求空闲分区链以首地址递增的次序链接。方式：查找大小适合的空闲区，分配出去，下次从剩下的链表的头部开始查找

**（2）NF——下次匹配法(next-fit) （循环首次适应算法 ） ：同样**要求空闲分区链以首地址递增的次序链接。方式：每次从上次结束的地方开始查找

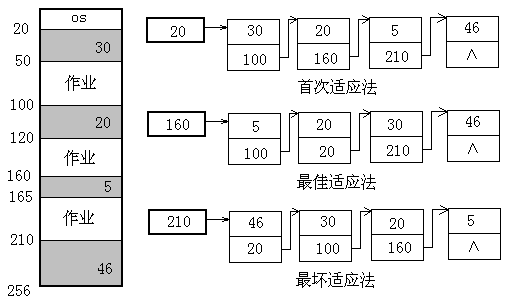
（3）NF——最佳适应法（best-fit ）：注意它要求空闲分区链以大小递增的方式链接。方式：每次都从头开始找起

（4）WF——最坏适配法（worst fit）：要求空闲分区链以大小递减的方式链接。方式每次都从头开始查找。（与最佳的对应着看）

（5）BF——快速适应算法（quick fit）：（不好意思，这个只能意会不能言传，别担心，不会考的）

例题：

**有作业序列：作业A要求18K；作业B要求25K，作业C要求30K。系统中空闲区按（首次，最佳，最坏）三种算法组成的空闲区队列：**

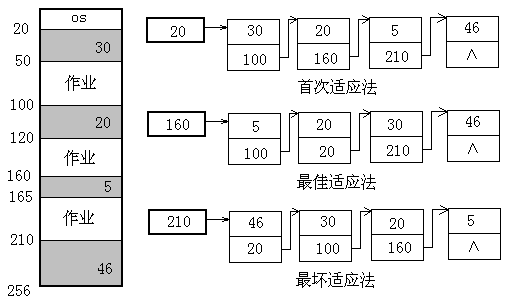


首地址 灰色的区域为空闲容量 上边这三个是链表，只解释首次适应法哈：链表按首地址递增的方式排列，即20,100,160,210.（30,20,5,4这几个数是空闲区的大小），接下来就用那三个作业的大小与空闲区比较，A作业要求18K，首地址是20 的空闲区有30K的容量，它满足条件，则把她分配给作业A，再找作业B的25K，没有能满足条件的，查找失败。其他两个队列也这么用，然后比较三种算法，哪种比较适合这个作业序列：

* **经分析可知：最佳适应法对这个作业序列是合适的，而其它两种对该作业序列是不合适的。**

再赠送道练习题：

有作业序列：作业A要求21K；作业B要求30K，作业C要求25K。



算法有多种，虽然比较的时候一般是两三种，但是这两三种都不一定适合当时的作业序列哈。

**第二种计算是页面置换的计算**

有两类，第一类：逻辑地址到物理地址的转换

页号和页内地址的计算

若给定一个逻辑地址空间中的地址为A，页面大小为L，则页号P和页内地址d可按下式求得：

P = A div L

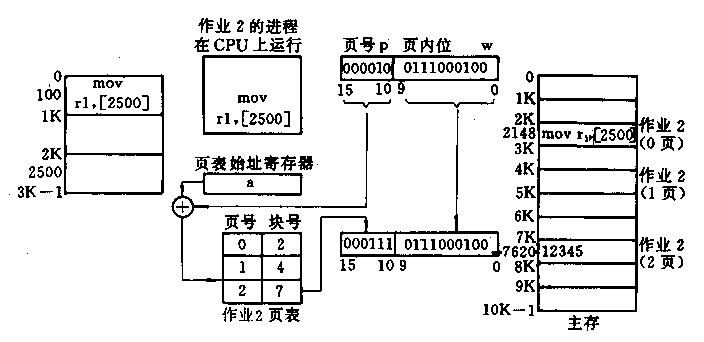
d = A mod L

其中，div指整除，mod指取余

例：系统页面大小为1KB，设A=2170B，则由上式可得P=2，d=112。

上边这是辅助运行公式，接下来是例题  
例

**设页长为1K，程序地址字长为16位，用户程序空间和页表如图。**



这个图这样看：先看作业2的进程在CPU上运行，这句话下边的框框mov r1 2500，别的不管只要2500,也就是逻辑地址A=2500，页面大小题目要求中给出是1K，即L=1024，用开始给出的两个公式，**简单的用就是A / L = 2500/1024=2余452,即叶号是2，页内地址是452，**把它们转换成二进制就是这个框框边上的那个二进制，至于这些二进制的**位数划分**就是看页长1K=1024B，即2的10方，所以得到页内地址占用10位，其余的位数是页号位。

现在我们去找逻辑地址，看图的下边那个写着页号块号的框框，我们已经算出页号是2，找到它，它对应的块号是7，它边上的二进制框框是块号+页内地址，现在看那个内存的框框，它是按1KB分的块，块号是首地址，找到7K的地方，7K=1024\*7=7168，再用452+7168=7620，即是物理地址。

第二类，页面置换算法

最佳页面算法(OPT, optimal)

先进先出页面置换算法(FIFO)

最近最久未使用置换算法（LRU）

轮转算法(clock)

最不经常使用（LFU）

例题：

**某程序在内存中分配三个块，访问页的走向为4，3，2，1，4，3，5，4，3，2，1，5，按FIFO、 LRU、OPT算法分别计算缺页次数。 假设开始时所有页均不在内存。**

这个例题就不详细解答了，弄起来太麻烦了，我附加了一份PPT先自己看下，再看下书应该可以理解了，要是不明白下午外包讲起走。

第五章 磁盘寻道算法

磁盘的访问时间可能不会考吧，还是一起弄下吧，以防万一。

访问时间=寻道时间+旋转延迟+数据传输；

公式：Ta=Ts+1/2r+b/rN=(m\*n+s)+ 1/2r+b/rN;

m一般是常量0.2，会给出，n是磁道数，s是启动磁壁的时间也一般会给出，Ts是寻道时间。

至于r是磁盘每秒的转数，N是一条磁道上的字节数。b是每次写入或读出的总字节数。

磁盘寻道算法

**先来先服务算法：（按出现的先后顺序寻道）**

例题：

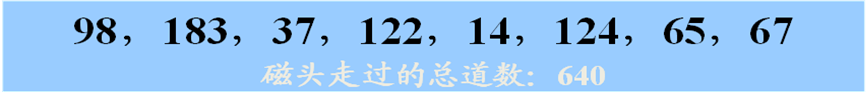
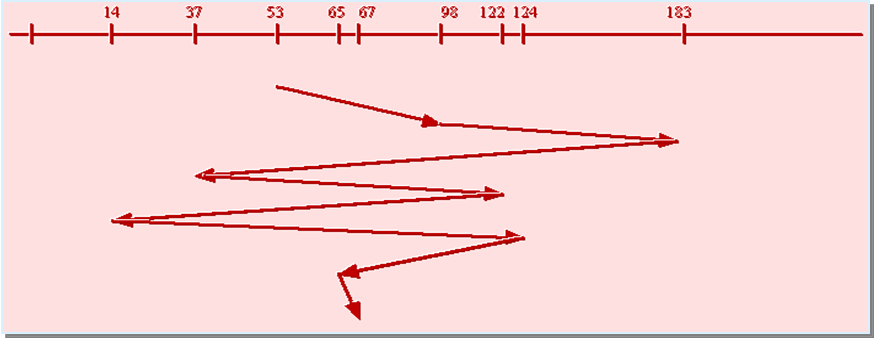
**假设磁盘访问序列：98，183，37，122，14，124，65，67**

**读写头起始位置：53**

**安排磁头服务序列**

**计算磁头移动总距离（道数）**

**图解如下**



**最短寻道时间优先算法-SSTF（按离当前道数最近的方式寻找）**

同样是上边的问题

图解：

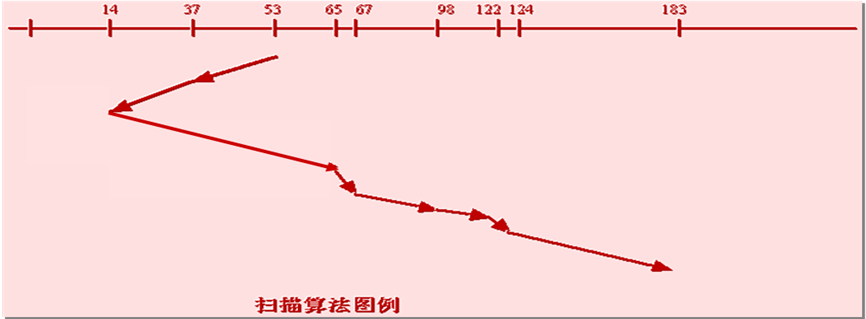
.**扫描算法（电梯算法）（SCAN）（要先看最初寻道的方向是怎么样的，沿着原有的方向寻道，到头后转向，进行反方向寻道）**



一般会给出寻道是按磁道号增加找还是减少找

同样是上边的例题，**假设**是开始的时候寻道**方向是按道号减小的方向则：**

**图解：**



**单向扫描调度（循环扫描）算法（CSCAN）**

**（它也是按同一方向寻找，不同的是它找到头之后要调回到最开始的欲访问的磁道上，按相同方向找，也就是它的方向是不变的）**

**图解：**

**假设这次按照增大的方向找：**



从图可以看出来，它找到183号道后跳转到14号道，再按增大的方向找起。

以上仅供参考，如有疑问请及时联系我哦，谢谢。

**西南石油大学软件工程2008级 王林 为你分享**