2022-5

试题有五道大题：选择(10 分)、名词解释(20 分)、简答(20 分)、计算题(24 分)、应用题(26 分)。用中文答题。

名词解释

1 Mutual exclusion 互斥

以某种手段确保当一个进程在使用一个共享变量或文件时，其他进程不能做同样的操作。

进程互斥是多道程序系统中进程间存在的一种源于资源共享的制约关系，也称间接制约关系，主要是由被共享资源的使用性质所决定的。

2 Process 进程

一个进程就是一个正在执行程序的实例，包括程序计数器、寄存器和变量的当前值。

3 Thread 线程

线程是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

4 Operating System 操作系统

操作系统是一种运行在内核态的软件。

操作系统是一组主管并控制计算机操作、运用和运行硬件、软件资源和提供公共服务来组织用户交互的相互关联的系统软件程序。

5 Race Conditions 竞争条件

两个或多个进程读写某些共享数据时，最后的结果取决于进程运行的精确时序。

6 Deadlock 死锁

两个或两个以上的进程在执行过程中，由于竞争资源或者彼此通信而造成的一种阻塞的现象，若无外力作用，两个进程都将永远的阻塞下去，无法再进行有效的工作。

7 System Calls 系统调用

系统调用是操作系统提供给用户的一种服务，程序设计人员在编写程序的时候可以用来请求操作系统的服务。系统内核通过包装一些能够实现特定功能的特殊硬件指令和硬件状态，即为内核函数，通过一组称为系统调用(system call)的接口呈现给用户。

8 Multiprogramming 多道程序设计

多道程序设计是在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制之下，相互穿插地运行。从宏观上看是并行的，多道程序都处于运行

中，并且都没有运行结束；从微观上看是串行的，各道程序轮流使用 CPU，交替执行。

9 Physical Address 物理地址

在存储器里以字节为单位存储信息，为正确地存放或取得信息，每一个字节单元给以一个唯一的存储器地址，称为物理地址。

10 Critical Region 临界区域

对共享资源进行访问的程序片段。当有线程进入临界区域时，其他线程或是进程必须等待。

11 Busy Waiting 忙等待

一个进程进入临界区之后，其他进程因无法满足竞争条件而循环探测竞争条件。连续测试一个变量直到某个值出现为止。

12 Buffering 缓冲

I/O 软件的另一个问题是缓冲。数据离开一个设备之后通常并不能直接存放到其最终的目的地，必须先预先放置到输出缓冲区之中，从而消除缓冲区填满速率和缓冲区清空速率之间的相互影响，以避免缓冲区欠载。缓冲涉及大量的复制工作，并且经常对 I/O 性能有重大影响。

13 I-nodes 索引节点

索引节点是一种数据结构。每个索引节点保存了文件系统中的一个文件系统对象的元信息数据，但不包括数据内容或者文件名。

14 Monitors 管程

代表共享资源的数据结构以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序共同构成了一个操作系统的资源管理模块，我们称之为管程。

一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作，这组操作能同步进程和改变管程中的数据。

15 Virtual Address 虚拟地址

CPU 启动保护模式后，程序运行在虚拟地址空间中，此时程序访问存储器所使用的逻辑地址称为虚拟地址。

16 Interrupt 中断

由于系统出现了某种需要处理的紧急情况，CPU 暂停正在执行的程序，转而去执行另一段特殊程序来处理的出现的紧急事务，处理结束后 CPU 自动返回到原先暂停的程序中去继续执行，这种执行过程由于外界的的原因被中间打断的情况成为中断。

17 Semaphore 信号量

信号量是 E.W.Dijkstra 在 1965 年提出的一种方法，他使用一个整形变量来

累计唤醒次数，供以后使用。在他的建议中引入了一个新的变量类型，称作信号量。一个信号量的取值可以为 0（表示没有保存下来的唤醒操作）或者为正值（表示有一个或多个唤醒操作）

18 Device driver　设备驱动程序

专门与控制器对话，发出命令并接收响应的软件，称为设备驱动程序。

19 Relocation　重定位

重定位就是把程序的逻辑地址空间变换成内存中的实际物理地址空间的过程。它是实现多道程序在内存中同时运行的基础。

20 Atomic action　原子操作

原子操作是指不会被线程调度机制打断的一个或一系列操作，这种操作一旦开始，就一直运行到结束。

21 Device independence　设备独立性

可使应用程序独立于具体使用的物理设备。设备独立性的具体实现：在应用程序中，使用逻辑设备名来请求使用某类设备，而在系统实际执行时，必须将逻辑设备名映射成物理设备名使用。

22 Avoiding Locks　避免锁

读-复制-更新，通过 RCU（读-复制-更新）将更新过程中的移除和再分配过程分离开来，从而不需要锁住任何东西。

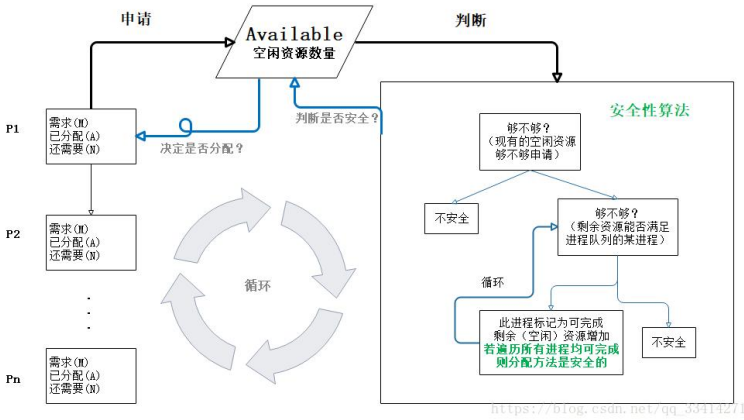
23 Read-Copy-Update　读-复制-更新

读-复制-更新将更新过程中的移除和再分配过程分离开。对于被 RCU 保护的共享数据，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后在适当的时机把指向原来数据的指针替换为新的被修改的数据。这个时机就是所有引用该数据的 CPU 都退出对共享数据的访问。

计算题：主要包括银行家算法、死锁检测算法、页面调度算法、磁臂调度算法、作业调度算法、地址定位、互斥算法

银行家算法

用于避免死锁。当一个进程申请使用资源的时候，银行家算法通过先 试探 分配给该进程资源，然后通过安全性算法判断分配后的系统是否处于安全状态，若不安全则试探分配作废，让该进程继续等待。



首先是银行家算法中的进程：
包含进程 Pi 的需求资源数量（也是最大需求资源数量，MAX）
已分配给该进程的资源 A（Allocation）
还需要的资源数量 N（Need=M-A）
Available 为空闲资源数量，即资源池（注意：资源池的剩余资源数量+已分配给所有进程的资源数量=系统中的资源总量）

假设资源 P1 申请资源，银行家算法先试探的分配给它（当然先要看看当前资源池中的资源数量够不够），若申请的资源数量小于等于 Available，然后接着判断分配给 P1 后剩余的资源，能不能使进程队列的某个进程执行完毕，若没有进程可执行完毕，则系统处于不安全状态（即此时没有一个进程能够完成并释放资源，随时间推移，系统终将处于死锁状态）。

若有进程可执行完毕，则假设回收已分配给它的资源（剩余资源数量增加），把这个进程标记为可完成，并继续判断队列中的其它进程，若所有进程都可执行完毕，则系统处于安全状态，并根据可完成进程的分配顺序生成安全序列（如{P0，P3，P2，P1}表示将申请后的剩余资源 Work 先分配给 P0－>回收（Work+已分配给 P0 的 A0=Work）－>分配给 P3－>回收（Work+A3=Work）－>分配给 P2－>・・・・・・满足所有进程）。

如此就可避免系统存在潜在死锁的风险。
Eg：

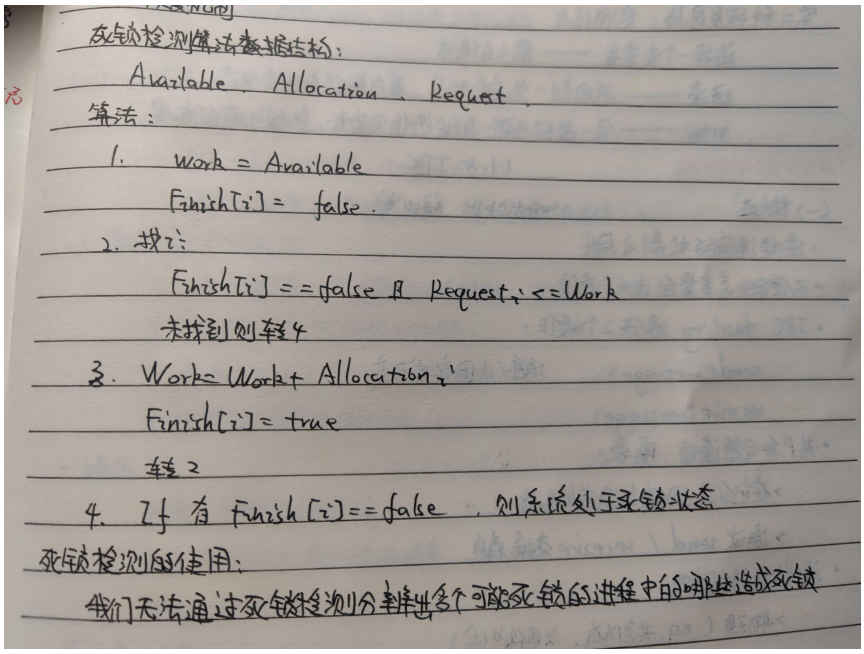
Process	Allocation	Need	Available
P0	0032	0012	1622
P1	1000	1750	
P2	1354	2356	
P3	0332	0652	
P4	0014	0656	

（1）该状态是否安全？ （2）若进程 P2 提出请求 Request（1，2，2，2）后，系统能否将资源分配给它？
（1）利用安全性算法对上面的状态进行分析（见下表），找到了一个安全序列{P0, P3, P4, P1, P2}，故系统是安全的。

	Work	Need	Allocation	Work+Allocation	Finish
P0	1 6 2 2	0 0 1 2	0 0 3 2	1 6 5 4	true
P3	1 6 5 4	0 6 5 2	0 3 3 2	1 9 8 6	true
P4	1 9 8 6	0 6 5 6	0 0 1 4	1 9 9 10	true
P1	1 9 9 10	1 7 5 0	1 0 0 0	2 9 9 10	true
P2	2 9 9 10	2 3 5 6	1 3 5 4	3 12 14 14	true

（2）P2 发出请求向量 Request(1,2,2,2)，系统按银行家算法进行检查：
①Request2(1,2,2,2)<=Need2(2,3,5,6)　满足 P2 此要求<=P2 的总要求
②Request2(1,2,2,2)<=Available(1,6,2,2)　P2 此要求<=剩余资源
③系统假定可为 P2 分配资源，并修改 Available，Allocation2 和 Need2：
Available=(0,4,0,0)
Allocation2=(2,5,7,6)
Need2=(1,1,3,4)
此时再进行安全性检查，发现 Available=(0,4,0,0) 不能满足任何一个进程，所以判定系统进入不安全状态，即不能分配给 P2 相应的 Request(1,2,2,2)。

死锁检测算法



页面调度算法
1. 最优页面置换算法

1. 最优页面置换算法

- 基本思路：当一个缺页中断发生时，对于保存在内存当中的每一个逻辑页面，计算在它的下一次访问之前，还需等待多长时间，从中选择等待时间最长的那个，作为被置换的页面。
- 这只是一种理想情况，在实际系统中是无法实现的，因为操作系统无从知道每一个页面要等待多长时间以后才会再次被访问。

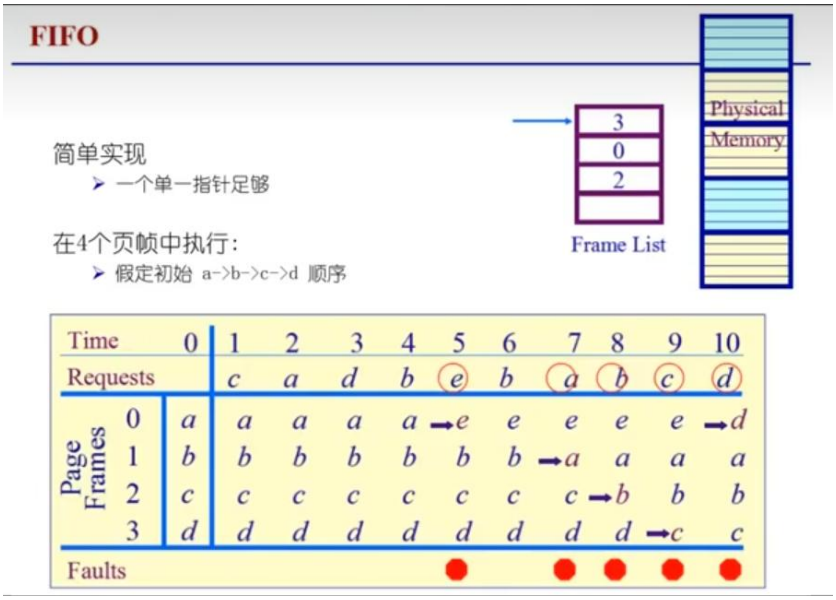
可用作其他算法的性能评价的依据（在一个模拟器上运行某个程序，并记录每一次的页面访问情况，在第二遍运行时即可使用最优算法 ）。

2. 先进先出算法 FIFO

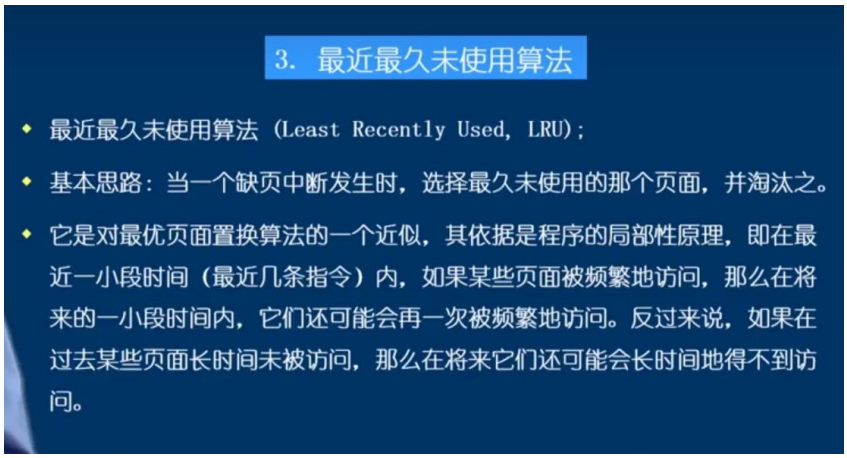
2. 先进先出算法

- 先进先出算法（First-In First-Out， FIFO）；
- 基本思路：选择在内存中驻留时间最长的页面并淘汰之。具体来说，系统维护着一个链表，记录了所有位于内存当中的逻辑页面。从链表的排列顺序来看，链首页面的驻留时间最长，链尾页面的驻留时间最短。当发生一个缺页中断时，把链首页面淘汰出局，并把新的页面添加到链表的末尾。

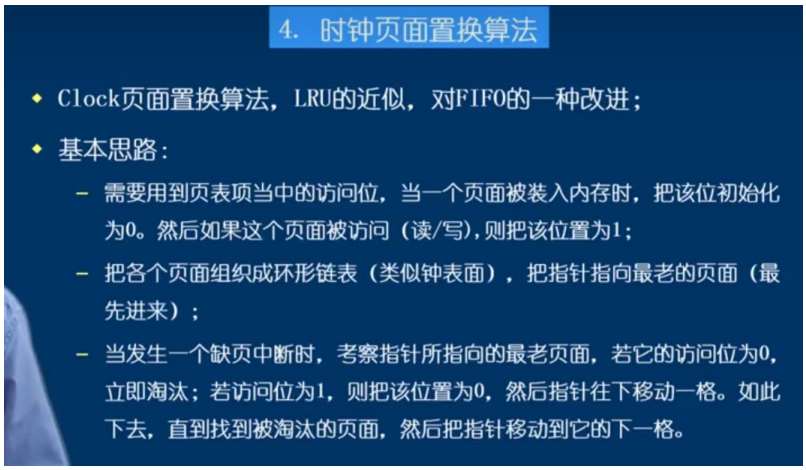
性能较差，调出的页面有可能是经常要访问的页面，并且有Belady现象。FIFO算法很少单独使用。

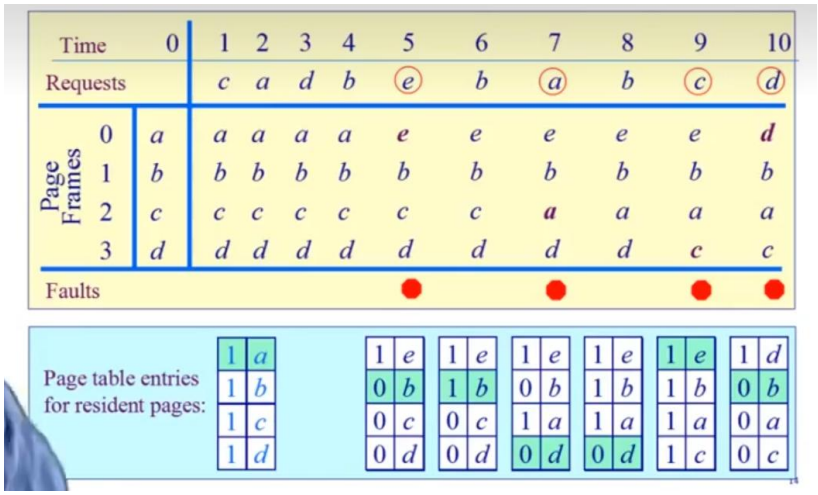


3. 最近最久未使用算法 LRU

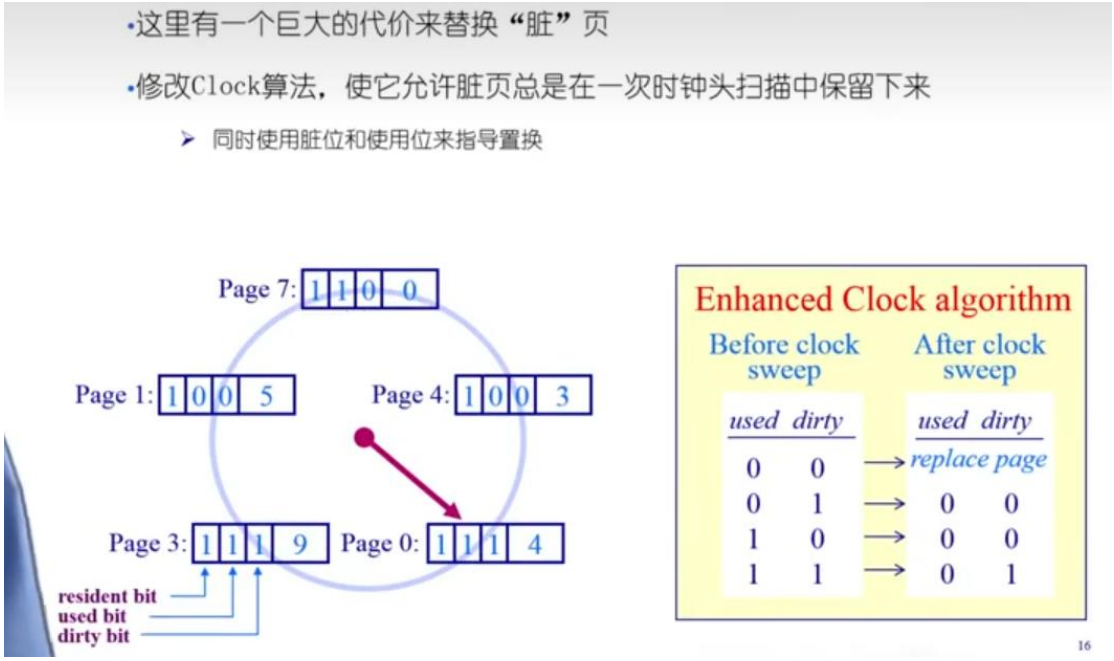


4. 时钟页面置换算法



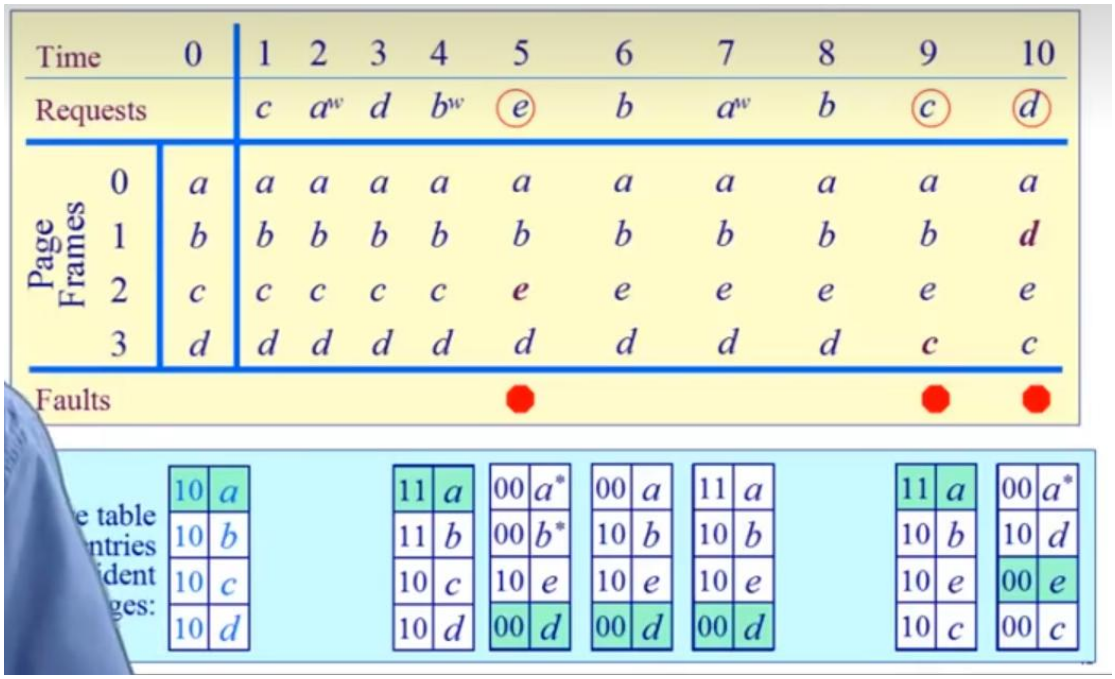


5. 二次机会法



二次机会法多设置了一个脏位。在脏位为 0 的情况下，如果访问位是 0，就淘汰这页；如果访问位是 1，就给它第二次机会，并选择下一个页面。当一个页面得到第二次机会时，它的

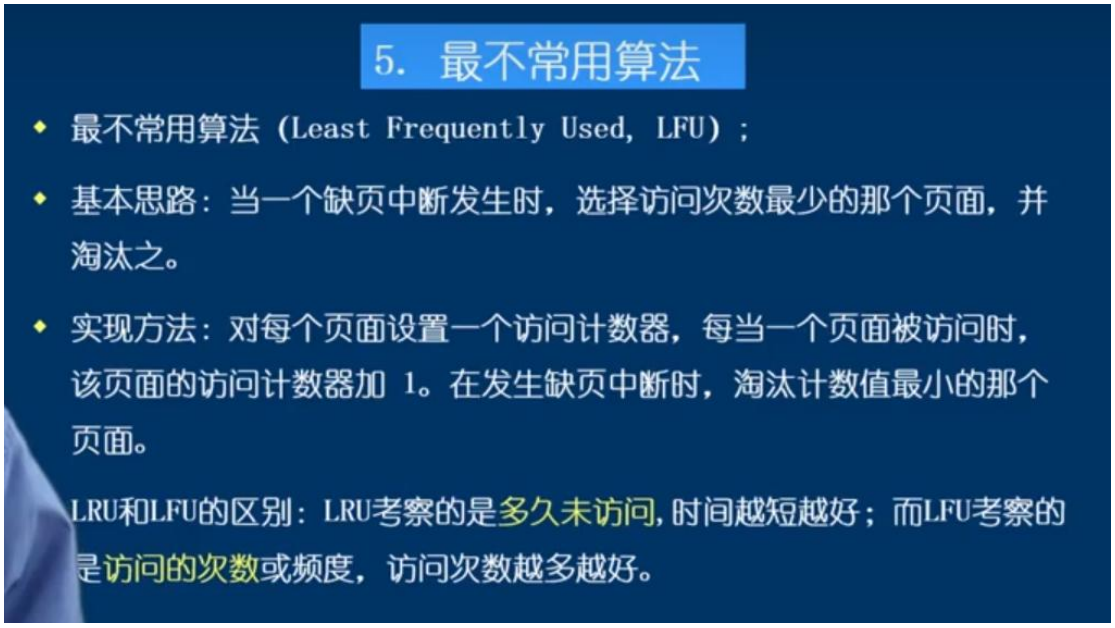
访问位清为 0，它的到达时间就置为当前时间。如果某页被访问，则访问位置为 1。如果某页被写操作，访问位与脏位都置 1。



Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a^w	d	b^w	e	b	a^w	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	d
	2	c	c	c	c	c	e	e	e	e	e	e
	3	d	d	d	d	d	d	d	d	d	c	c
Faults							●				●	●

e table ntries dent ges:	10 a		11 a		00 a^*		00 a		11 a		11 a	00 a^*
	10 b		11 b		00 b^*		10 b		10 b		10 b	10 d
	10 c		10 c		10 e		10 e		10 e		10 e	00 e
	10 d		10 d		00 d		00 d		00 d		10 c	00 c

6. 最不常用法



执行在4个页帧中：

- 假设最初的访问次数：a->8　b-> 5 c-> 6　d->2

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	e	e	e	e	e	d
	1	b	b	b	b	b	b	b	a	a	a	a
	2	c	c	c	c	c	c	c	c	b	b	b
	3	d	d	d	d	d	d	d	d	d	c	c
Faults												

全局页面置换算法

7. 工作集模型

工作集：一个进程当前正在使用的逻辑页面集合，

可以用一个二元函数W(t, Δ)来表示：

- t是当前的执行时刻；
- Δ 称为工作集窗口（working-set window），即一个定长的页面访问的时间窗口；
- W(t, Δ)＝在当前时刻 t 之前的 Δ时间窗口当中的所有页面所组成的集合（随着 t 的变化，该集合也在不断地变化）；
- | W(t, Δ) | 指工作集的大小，即页面数目。

页面访问顺序：

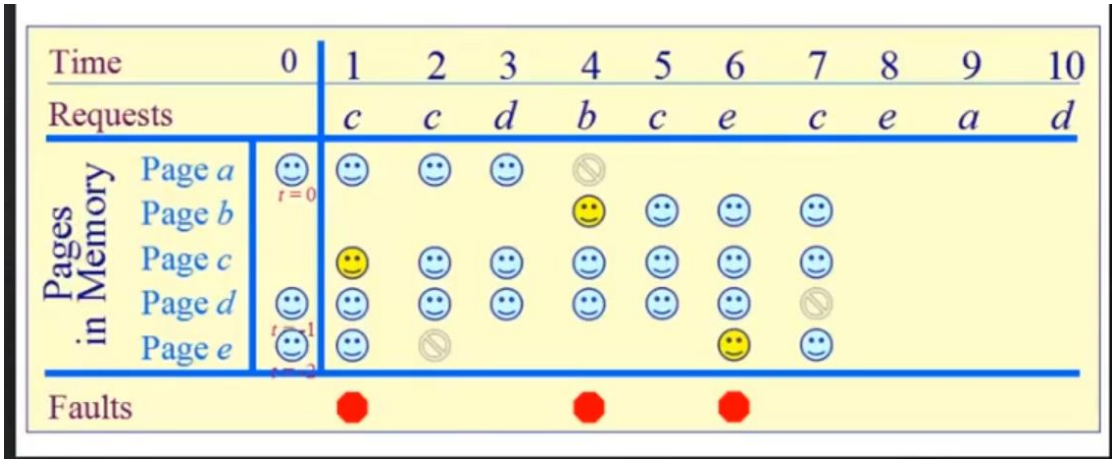
2 6 1 5 7 7 7 7 5 1　6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4　1 3 2 7

如果 Δ 时间窗口的长度为10，那么：

$W(t_1, \Delta) = \{1, 2, 5, 6, 7\}$

$W(t_2, \Delta) = \{3, 4\}$

下例中限定工作集窗口大小为4（工作集中可有重复页）



8. 缺页率页面置换算法

缺页率页面置换算法

- 可变分配策略：常驻集大小可变。例如：每个进程在刚开始运行的时候，先根据程序大小给它分配一定数目的物理页面，然后在进程运行过程中，再动态地调整常驻集的大小。
  - 可采用全局页面置换的方式，当发生一个缺页中断时，被置换的页面可以是在其它进程当中，各个并发进程竞争地使用物理页面。
  - 优缺点：性能较好，但增加了系统开销。
  - 具体实现：可以使用缺页率算法(PFF，page fault frequency)来动态调整常驻集的大小。

缺页率表示“缺页次数 / 内存访问次数”（比率）或“缺页的平均时间间隔的倒数”。影响缺页率的因素：

- 页面置换算法
- 分配给进程的物理页面数目
- 页面本身的大小
- 程序的编写方法

一个交替的工作集计算明确的试图最小化页缺失

- 当缺页率高的时候-增加工作集
- 当缺页率低的时候-减少工作集

算法：

保持追踪缺失发生概率

当缺失发生时，从上次页缺失起计算这个时间记录这个时间，t_{last}是上次的页缺失的时间。

如果发生页缺失之间的时间是“大”的，之后减少工作集如果。

如果 $t_{current} - t_{last} > T$，之后从内存中移除所有在[$t_{last}$, $t_{current}$]时间内没有被引用的页。

如果这个发生页缺失的时间是“小”的，之后增加工作集。

如果$t_{current} - t_{last} < T$，之后增加缺失页到工作集中。

Example: window size = 2

- 如果 $t_{current} - t_{last} > 2$，从工作集中移除没有在[$t_{last}$, $t_{current}$] 被引用的页面
- 如果 $t_{current} - t_{last} < 2$，仅增加缺失页到工作集中

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		c	c	d	b	c	e	c	e	a	d
Pages in Memory: Page a	☺	☺	☺	☺	⊘					☺	☺
Page b					☺	☺	☺	☺	☺	⊘	
Page c		☺	☺	☺	☺	☺	☺	☺	☺	☺	☺
Page d	☺	☺	☺	☺	☺	☺	☺	☺	☺	⊘	☺
Page e	☺	☺	☺	☺	⊘		☺	☺	☺	☺	☺
Faults		●			●		●			●	●
$t_{cur} - t_{last}$		1			3		2			3	1

磁臂调度算法

Ta = 访问时间

Ts = 寻道时间

Tr = 旋转延迟

T = 传输时间

b = 传输的比特数

N = 磁道上的比特数

r = 磁盘转数

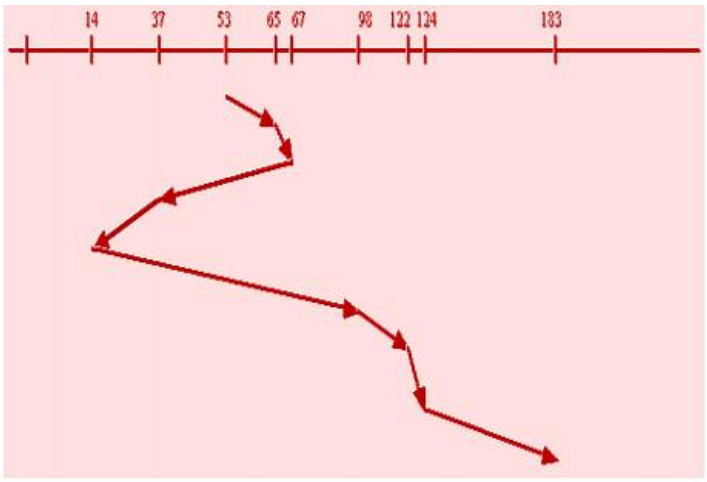
$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

$$= + +$$

Tr Tt

1/r 表示旋转一周的时间

1. 最短寻道时间优先 SSF (Shortest Seek First)
总是处理与磁头距离最近的请求以使寻道时间最小化。
磁盘访问序列：
98，183，37，122，14，124，65，67
读写头起始位置：53，
那么磁臂移动的轨迹为：



2. 电梯算法 (Elevator algorithm)

当设备无访问请求时，磁头不动；当有访问请求时，磁头按一个方向移动，在移动过程中对遇到的访问请求进行服务，然后判断该方向上是否还有访问请求，如果有则继续扫描；否则改变移动方向，并为经过的访问请求服务，如此反复。

举例：

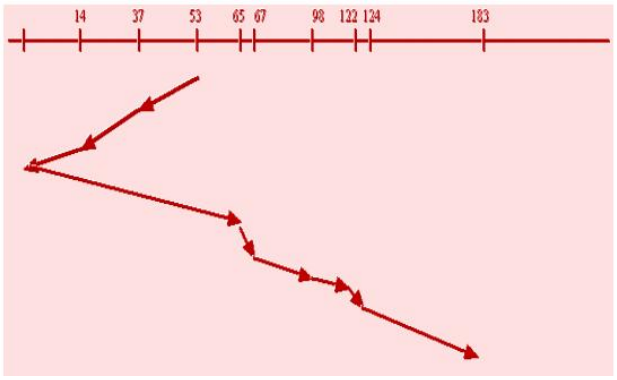
假设磁盘访问序列：

磁盘访问序列：

98，183，37，122，14，124，65，67

读写头起始位置：53，

那么磁臂移动的轨迹为：



作业调度算法

单道批处理系统中，有四个作业，到达时间和所需运行时间如下表所示，按照先来先服务（FCFS），短作业优先（SJF），最高响应比优先(HRN）算法计算各个进程的开始运行时间，完成时间，周转时间，带权周转时间和所有作业的平均周转时间，平均带权周转时间。

作业	到达时间/h	运行时间/h	开始时间	完成时间	周转时间	带权周转时间	平均周转时间	平均带权周转时间
1	8.0	2.0						
2	8.6	0.6						
3	8.8	0.2						
4	9.0	0.5						

这里先贴出几个式子：

名称	式子
响应比	(等待时间 + 运行时间) / 运行时间
周转时间	完成时间 - 到达时间
带权周转时间	周转时间 / 运行时间
完成时间	开始时间 + 运行时间

（一）批处理系统中的调度

1. 先来先服务（FCFS）

按照作业到达时间的先后来调度作业运行，最先到达的作业会先投入运行，因为作业 1 到达时间最早所以最先运行作业 1

2. 短作业优先（SJF）

短作业优先会先运行运行时间短的作业，比如上面的作业中，运行时间最短的是作业 3，有人会问那是不是先运行作业 3 呢？答案是不对的，因为作业 3 的到达时间比较晚，在作业 3 到达之前作业 1 是第一个到达的，所以还是先运行作业 1，作业 1 的完成时间是 10.0，这时所有的作业都已经到达了，按照短作业优先算法，接下来运行作业 3，然后是作业 4，最后是作业 2

3. 最短剩余时间优先（SRT）

抢占式的 SJF，选剩余运行时间最短的那个程序来运行

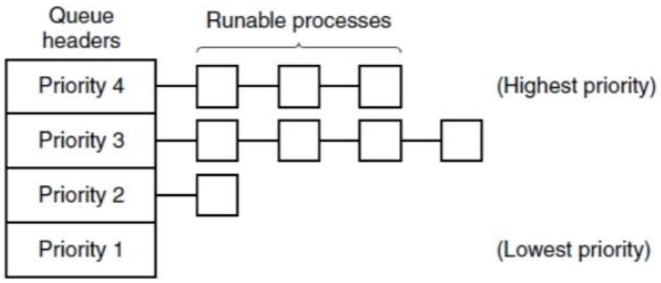
（二）交互式系统中的调度

1. 轮转法(Round Robin)

将系统中所有的就绪进程按照 FCFS 原则，排成一个队列。每次调度时将 CPU 分派给队首进程，让其执行一个时间片。时间片的长度从几个 ms 到几百 ms。在一个时间片结束时，发生时钟中断。调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。

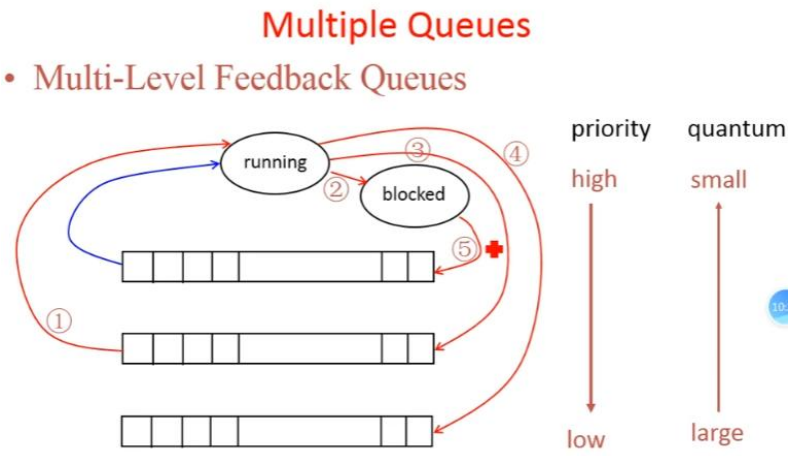
进程可以未使用完一个时间片，就出让 CPU（如阻塞）。

2. 优先级算法（Priority Scheduling）



静态优先级从一而终不改变；动态优先级会改变。

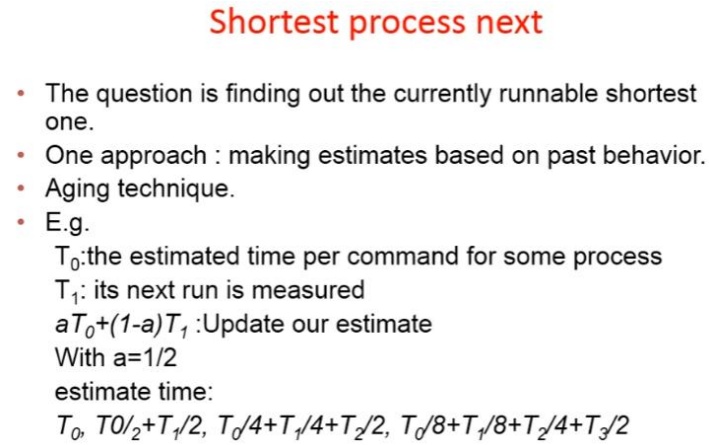
3. 多级队列（Multiple Queues）



最高级别队列按轮巡的方式执行，在一段固定时间后，如果最高级别队列中仍有未完成的进程，则下放到下一个级别的最末尾，此时最高级别队列中无进程，故使用轮巡的方式执行下一级队列的队列中所有进程。一旦有新的最高级别的进程插入到最高级队列，则重新回去执行最高级别队列的进程。

时间片从高级到低级越来越大

4. 最短进程优先



根据算出来的估计时间来选择最短进程

地址重定位

地址重定位就是操作系统将逻辑地址转变为物理地址的过程

十进制虚拟地址 20000

[解法]

20000 转换为二进制分别为：0100 1110 0010 0000

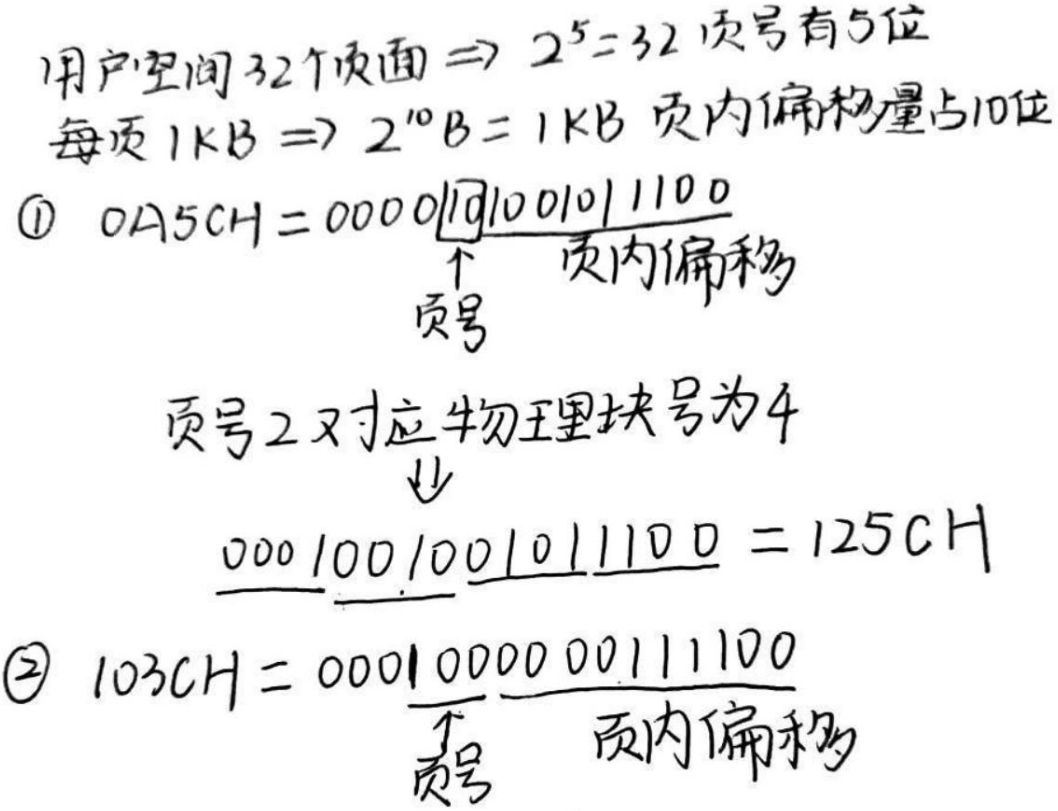
（1）使用 4KB 页面

4K=4*1K=2^2*2^10=2^12,页内地址范围 0----2^12-1,即需要 12 位表示业内偏移量，对应 1110 0010 0000，剩余部分 0100 即为页号对应部分。

虚拟页号=0100=2^2=4，

偏移量=1110 0010 0000= 2^11+2^10+2^9+2^5=2048+1024+512+32=3616.

某虚拟存储器的用户空间共有 32 个页面，每页 1KB，主存 16KB。假定某时刻系统为用户的第 0、1、2、3 页分配的物理块号为 5、10、4、7，而该用户作业的长度为 6 页，试将十六进制的虚拟地址 0A5C、103C 转换成物理地址。



互斥算法

1.单标记法：

设置一个标记 turn 表示谦让，访问临界资源前，检查是否被谦让，每次访问完，将临界资源谦让出去：

```
//检查
while(turn==1);
//访问
TODO
//谦让
turn=1;
```

2.双标记先检查法：

双标记指的是 flag[ ]=1 表示想访问，其中 flag[0]=1 表示 P0 想用；flag[1]=1 表示 P1 想用。每次先检查后上锁。

```
while(flag[1]==1);//检查
flag[0]=1;//上锁
TODO//访问
flag[0]=0;//开锁
```

3.双标记后检查法：

每次先上锁后检查。

```
flag[0]=1;//上锁
```

while(flag[1]==1);//检查
TODO//访问
flag[0]=0;//开锁
4.PeterSon 算法：
综合单标记法和双标记法（三个标记），有 turn 表示谦让，flag[]=1 表示想访问，其中 flag[0]=1 表示 P0 想用；flag[1]=1 表示 P1 想用。
//表示想用，再表示谦让
flag[0]=1；
turn=1;
//检查
while(flag[1]==1&&turn=1);//他想用，且我谦让了
//访问
TODO
//表示不想用
flag[0]=0;
5.TestAndSet
将检查和上锁一气呵成，bolck 表示上锁。
TestAndSet(bool &lock)//检查和上锁一气呵成
{
bool old=lock;//先检查
lock=true；//后上锁
return old;
}

while(TestAndSet(bool &lock)==true);
//访问
TODO
//开锁
lock=false；
6.整形信号量
用整形数 S 表示资源数。（先检查后上锁）
```
 wait(S)
 {
 while(S<=0);//检查
 S--;//占用资源
 TODO//访问
 }
 signal(S)
 {
  S++;//释放资源
}
```

8.记录型信号量
记录可以将进程挂起，实现让权等待。（先上锁后检查）

```
typedef Struct{
int value;//资源剩余数
Struct process *L//进程
}semaphore;//信号量

P(semaphore S){
S.value--;//占用资源
if(S.value<0)//检查,如果没资源则挂起
    block(S.L);
}
V(semaphore S){
S.value++;//释放资源
//检查是否还有阻塞的进程
if(S.value<=0)//因为 value<=0 表明释放后仍没有空闲资源，即一定还有进程被阻塞，所以要唤醒
    weakup(S.L);
}
```

同步先 V 后 P；互斥先 P 后 V

应用题：用信号量及管程解决同步互斥问题

信号量

进入临界区的进程或线程若只是做读操作，则没必要只限制 1 个线程或进程去执行，需要更高级的同步互斥手段去完成这一机制，所以引入信号量

信号量

抽象数据类型

- 一个整形（sem），两个原子操作
- P()：sem减1，如果 sem<0，等待，否则继续
- V()：sem加1，如果 sem<=0，唤醒一个等待的P

信号量是整数

信号量是被保护的变量

- 初始化完成后，唯一改变一个信号量的值的办法是通过P()和V()
- 操作必须是原子

P 操作可能会使进程阻塞挂起
V 操作会唤醒进程

使用硬件原语

禁用中断

原子指令(test-and-set)

类似锁

例如：使用‘禁用中断’

```
classSemaphore {
int sem;
WaitQueue q;
}
```

```
Semaphore::P() {
  sem--;
  if (sem < 0) {
    Add this thread t to q;
    block(p);
  }
}
```

```
Semaphore::V() {
  sem++;
  if (sem<=0) {
    Remove a thread t from q;
    wakeup(t);
  }
}
```


两种类型信号量

（一）二进制信号量：只能是 0 或 1，可以用来模拟锁机制

实现互斥

```
mutex = new Semaphore(1);
```

```
mutex->P();
  …
Critical Section;
  …
mutex->V();
```

实现同步

初始值设为 0

```
condition = new Semaphore(0);
```

Thread A

```
…
condition->P();
  …
```

Thread B

```
…
condition->V();
  …
```

（二）计数信号量：任意非负数

更复杂的同步互斥不能用简单的二进制信号量来解决，如生产者-消费者问题

1.buffer 有限

2.（互斥）生产者写数据时，消费者不能取数据；允许多个生产者（消费者）向 buffer 写（读）数据

3.（同步）buffer 为空时，消费者应该睡眠，等到生产者向 buffer 里写完数据后才唤醒消费者；当 buffer 为满时，生产者要等消费者取后才可以往里写

正确性要求

- 在任何一个时间只能有一个线程操作缓冲区（互斥）
- 当缓冲区为空，消费者必须等待生产者（调度/同步约束）
- 当缓存区满，生产者必须等待消费者（调度/同步约束）

每个约束用一个单独的信号量

- 二进制信号量互斥
- 一般信号量fullBuffers
- 一般信号量emptyBuffers

2 个一般信号量用于实现同步
mutex 初始化为 1，用来实现互斥
fullBuffers 初始化为 0，说明最开始时 buffer 里 1 个也没有
emptyBuffers 初始化为 n，表示可以往 buffer 里塞多少个数据

```
Class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
```

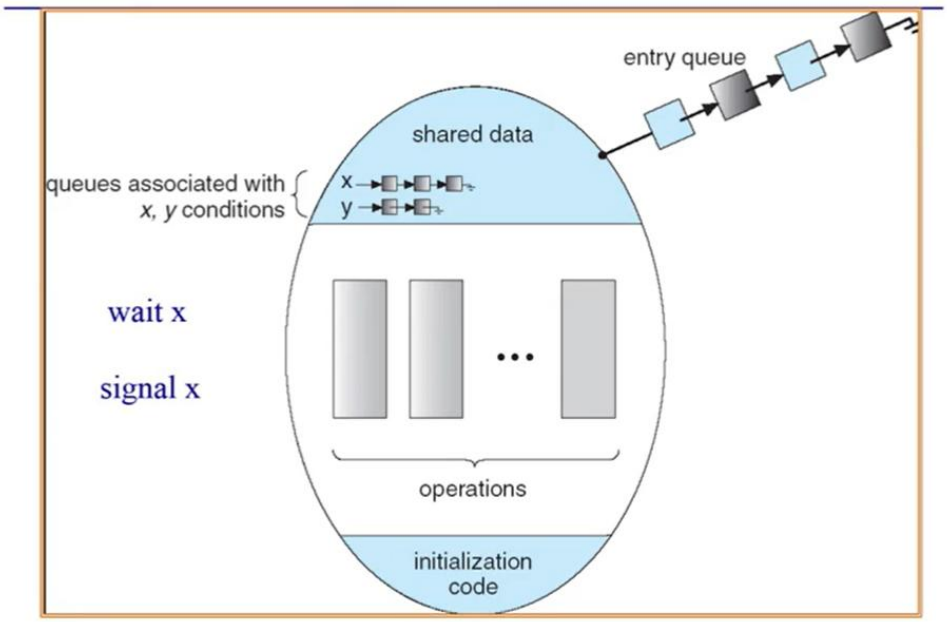
```
BoundedBuffer::Deposit(c) {
    emptyBuffers->P();
    mutex->P();
    Add c to the buffer;
    mutex->V();
    fullBuffers->V();
}
```

```
BoundedBuffer::Remove(c) {
    fullBuffers->P();
    mutex->P();
    Remove c from buffer;
    mutex->V();
    emptyBuffers->V();
}
```


管程

管程是包含了一系列共享变量以及针对这些变量的操作的函数的一个组合（模块）

- 一个锁：指定临界区
- 0或者多个条件变量：等待/通知信号量用于管理并发访问共享数据



entry queue：往管程里送进程的队列，进入管程是互斥的，要先获得 lock 才能进去，取不到就只能在这个队列中
条件变量 x,y：有 2 个等待队列，挂着需要等待 x 或 y 的线程。当某个线程需要等待 x 或 y 时，就要先释放锁，再挂到 x 或 y 的等待队列上去
wait x：让进程去等待 x（挂在 x 队列）
signal x：唤醒 x，使得挂在 x 上的进程有机会能继续执行
条件变量的实现

```
Class Condition {
   int numWaiting = 0;
   WaitQueue q;
}
```

```
Condition::Wait(lock){
   numWaiting++;
   Add this thread t  to q;
   release(lock);
   schedule(); //need mutex
   require(lock);
}
```

```
Condition::Signal(){
   if (numWaiting > 0) {
      Remove a thread t from q;
      wakeup(t); //need mutex
      numWaiting--;
   }
}
```

numWaiting：等待条件的线程的个数
sem：信号量的个数
PV 操作一定会有加减操作，但是 Signal 不一定会做减操作
问：Wait 里为什么要先 release 再 require 锁？

Deposit 里先 require 之后才会有可能调用 wait，此时如果 wait 里没有 release 锁，那么其他线程都不会进入到管程中（这锁你拿了也没用，先还了，等你条件满足了[被唤醒]，再请求拿锁吧）

管程实现消费者-生产者

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

```
BoundedBuffer::Deposit(c) {
    lock->Acquire();
    while (count == n)
        notFull.Wait(&lock);
    Add c to the buffer;
    count++;
    notEmpty.Signal();
    lock->Release();
}
```

```
BoundedBuffer::Remove(c) {
    lock->Acquire();
    while (count == 0)
        notEmpty.Wait(&lock);
    Remove c from buffer;
    count--;
    notFull.Signal();
    lock->Release();
}
```

lock->Acquire()、lock->Release()的位置不同于信号量的（信号量的是紧紧靠在 buffer 上下的）：原因是管程的定义为只允许一个线程进入

当一个线程在管程中执行的话，唤醒了另一个线程后，是先执行唤醒的还是先执行被唤醒的？有以下 2 种想法：

Hansen：先执行唤醒方

Hoare：先执行被唤醒方

先执行被唤醒方还是唤醒方由硬件实施，不是靠代码

下例生产者为被唤醒方

```
Hansen-style :Deposit(){
  lock->acquire();
  while (count == n) {
      notFull.wait(&lock);
  }
  Add  thing;
  count++;
  notEmpty.signal();
  lock->release();
}
```

```
Hoare-style: Deposit(){
  lock->acquire();
  if (count == n) {
      notFull.wait(&lock);
  }
  Add thing;
  count++;
  notEmpty.signal();
  lock->release();
}
```

Hansen：当有线程做了 signal 操作时，会接着往下执行，有可能有多个等待在条件变量上的线程被唤醒，它们会去抢 CPU，但只有 1 个 CPU，当被唤醒的线程去执行时，可能 count 不为 n 了，所以要用 while 再做 1 次确认

Hoare：做完 signal 操作后就把 CPU 交给了被唤醒的线程，这时候只有 1 个线程被唤醒（不会有多个，因为只能唤醒 1 个），此时 count 一定不为 n，因为 count 小于 n 时才做 signal 操作，使用被唤醒后，count<n 这个操作依然满足，不会被破坏

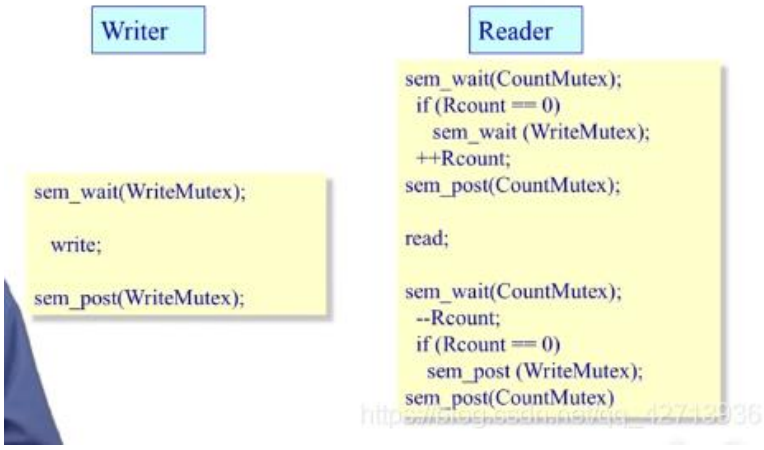
读者-写者问题

- 动机
  - 共享数据的访问
- 两种类型使用者
  - 读者：不需要修改数据
  - 写者：读取和修改数据
- 问题的约束
  - 允许同一时间有多个读者，但在任何时候只有一个写者
  - 当没有写者是读者才能访问数据
  - 当没有读者和写者时写者才能访问数据
  - 在任何时候只能有一个线程可以操作共享变量

不允许多个写者同时操作（会导致数据的不一致性）；允许多个读者同时操作
不允许读者和写者同时操作

- 共享数据
  - 数据集
  - 信号量CountMutex初始化为1
  - 信号量WriteMutex初始化为1
  - 整数 Rcount 初始化为0

当有读者读操作时，写者必须要等待，直到读者全部读完之后，写者才能去写
写者做写操作，读者和其他写者都必须要等待，直到写者做完写操作
Rcount：读者的个数（因为写的时候只能有 1 个写者操作，所以没必要记写者个数）
CountMutex：保证对 Rcount 的读或写是互斥操作 （同一时间只有一个进程能访问 Rcount）
WriteMutex：写者去操作也是互斥的 （用于满足读写互斥，写写互斥）
读者优先：当有读者正在读时，来了一个写者，之后又来了一个读者，则后来的这个读者可以先跳过等待的写者。读操作对数据没有破坏，则可以跳过等待的写者去完成相应的操作
writer：保证只有 1 个写者可以进去操作
reader：保证一旦有写者再写，则读者不能进去读，一旦有读者再读，写者也不能进去写
sem_wait() = P 操作
sem_post() = V 操作



对 Rcount++和 Rcount－要进行互斥的保护，所以加 CountMutex 的 PV 操作

写者优先：只要有写者来，读者就不能读，要先写
读者在读之前，要确保没有写者，此种写者包括：1、正在执行写操作的写者，2、在等待队列中的写者。只要上述 2 种写者存在，读者就不能执行读操作
读者

```
AR = 0;           // # of active readers
AW = 0;           // # of active writers
WR = 0;           // # of waiting readers
WW = 0;           // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```

```
Public Database::Read() {
    //Wait until no writers;
    StartRead();
    read database;
    //check out – wake up
waiting writers;
    DoneRead();
}
```

```
Private Database::StartRead() {
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}
```

```
Private Database::DoneRead() {
lock.Acquire();
    AR--;
    if (AR ==0 && WW > 0) {
        okToWrite.signal();
    }
lock.Release();
}
```

有写者在写或者等待：那我也等　　没人在读且有写者在等待：唤醒写者

写者

```
AR = 0;           // # of active readers
AW = 0;           // # of active writers
WR = 0;           // # of waiting readers
WW = 0;           // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```

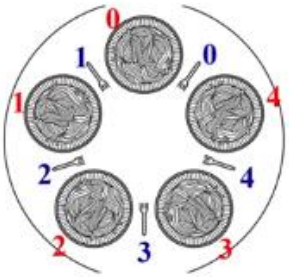
```
Public Database::Write() {
    //Wait until no readers/writers;
    StartWrite();
    write database;
    //check out – wake up waiting
readers/writers;
    DoneWrite();
}
```

```
Private Database::StartWrite() {
lock.Acquire();
    while ((AW+AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
lock.Release();
}
```

```
Private Database::DoneWrite() {
lock.Acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    }
    else if (WR > 0) {
        okToRead.broadcast();
    }
lock.Release();
}
```

有人在读或者写：我等等　　有写者在等：叫他　没写者但是有读者在等：叫醒他

哲学家吃饭



问题描述：（1965年由Dijkstra首先提出并解决）5个哲学家围绕一张圆桌而坐，桌子上放着5支叉子，每两个哲学家之间放一支；哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两支叉子，思考时则同时将两支叉子放回原处。如何保证哲学家们的动作有序进行？如：不出现有人永远拿不到叉子；

- 共享数据
  - Bowl of rice (data set)
  - Semaphore fork [5] initialized to 1

take_fork(i) : P(fork[i])　　put_fork(i) : V(fork[i])

5 个叉子 fork[5]使用的是信号量，初始化为 1，拿起为 P，放下为 V
1 个哲学家代表 1 个线程

思路(2) 计算机程序怎么来解决这个问题？

指导原则：不能浪费CPU时间；进程间相互通信。

S1　思考中…
S2　进入饥饿状态；
S3　如果左邻居或右邻居正在进餐，进程进入阻塞态；
　　否则转S4
S4　拿起两把叉子；
S5　吃面条…
S6　放下左边的叉子，看看左邻居现在能否进餐
　　（饥饿状态、两把叉子都在），若能则唤醒之；
S7　放下右边的叉子，看看右邻居现在能否进餐，
　　（饥饿状态、两把叉子都在），若能，唤醒之；
S8　新的一天又开始了，转S1

1. 必须有一个数据结构，来描述每个哲学家的当前状态；

```
#define   N          5              // 哲学家个数
#define   LEFT       i              // 第i个哲学家的左邻居
#define   RIGHT      (i+1)%N        // 第i个哲学家的右邻居
#define   THINKING   0              // 思考状态
#define   HUNGRY     1              // 饥饿状态
#define   EATING     2              // 进餐状态
int   state[N];                     // 记录每个人的状态
```

2. 该状态是一个临界资源，对它的访问应该互斥地进行

```
semaphore   mutex;                  // 互斥信号量，初值1
```

3. 一个哲学家吃饱后，可能要唤醒邻居，存在着同步关系

```
semaphore   s[N];                   // 同步信号量，初值0
```

函数philosopher的定义

```
void   philosopher(int  i)          // i的取值：0到N－1
{
    while(TRUE)                     // 封闭式循环
    {
S1  ——→ think( );                   // 思考中…
S2 -S4 ——→ take_forks(i);           // 拿到两把叉子或被阻塞
S5  ——→ eat( );                     // 吃面条中…
S6 -S7 ——→ put_forks(i);            // 把两把叉子放回原处
    }
}
```


函数take_forks的定义

```
// 功能：要么拿到两把叉子，要么被阻塞起来。
void   take_forks(int  i)                  // i的取值：0到N－1
{
    P(mutex);                              // 进入临界区
    state[i] = HUNGRY;                     // 我饿了！
    test_take_left_right_forks(i);         // 试图拿两把叉子
    V(mutex);                              // 退出临界区
    P(s[i]);                               // 没有叉子便阻塞
}
```


函数test_take_left_right_forks的定义

```
void  test_take_left_right_forks(int  i)    //i：0到N－1
{
    if(state[i]        ==  HUNGRY &&        // i：我自己，or 其他人
       state[LEFT]     !=  EATING &&
       state[RIGHT]  !=  EATING)
    {
       state[i] = EATING;                   // 两把叉子到手
       V(s[i]);                             //通知第i人可以吃饭了
    }
}
```

函数put_forks的定义

```
// 功能：  把两把叉子放回原处，并在需要的时候，
// 去唤醒左邻右舍。
void   put_forks(int  i)                    // i的取值：0到N－1
{
    state[i] = THINKING;                     // 交出两把叉子
    test_take_left_right_forks(LEFT);       // 看左邻居能否进餐
    test_take_left_right_forks(RIGHT);      // 看右邻居能否进餐

}
```


函数test_take_left_right_forks的定义

```
void  test_take_left_right_forks(int  i)    //i：0到N－1
{
    if(state[i]        ==  HUNGRY &&       // i：我自己，or 其他人
        state[LEFT]     !=  EATING &&
        state[RIGHT]  !=  EATING)
    {
        state[i] = EATING;                  // 两把叉子到手
        V(s[i]);                            //通知第i人可以吃饭了
    }
}
```


设系统磁盘只有一个移动磁头，磁道由外向内编号为：0、1、2、……、199； 当前磁头位置处于第 102 磁道，当前移动方向由外向内。设有磁道的 I/O 请求序列：**75、132、55、110、105、130、85、31、68、46、95、16、100** 每个请求读/写磁道上的 1 个扇区；按照算法要求写出给定 I/O 请求序列的调度序列，并计算磁头的移动量。

（1）最短寻道优先;

（2）电梯算法.

每一个位（bit）存储一个 1 位的二进制码，一个字节（B，bytes）由 8 位组成。而字（Word）通常为 16、32 或 64 个位组成。

1KB=1024B，1MB=1024KB，1GB=1024（2^10）MB

B KB MB GB

1 秒(s)=1000 毫秒(ms)
ps、ns、us、ms、s 皮秒 纳秒 微秒 毫秒 秒

互斥锁（英語：Mutual exclusion，缩写 Mutex）是一种用于多线程编程中，防止两条线程同时对同一公共资源进行读写的机制。该目的通过将代码切片成一个一个的临界区域（critical p）达成。临界区域指的是一块对公共资源进行存取的代码。

信号量(Semaphore)，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用，可以认为 mutex 是 0-1 信号量；