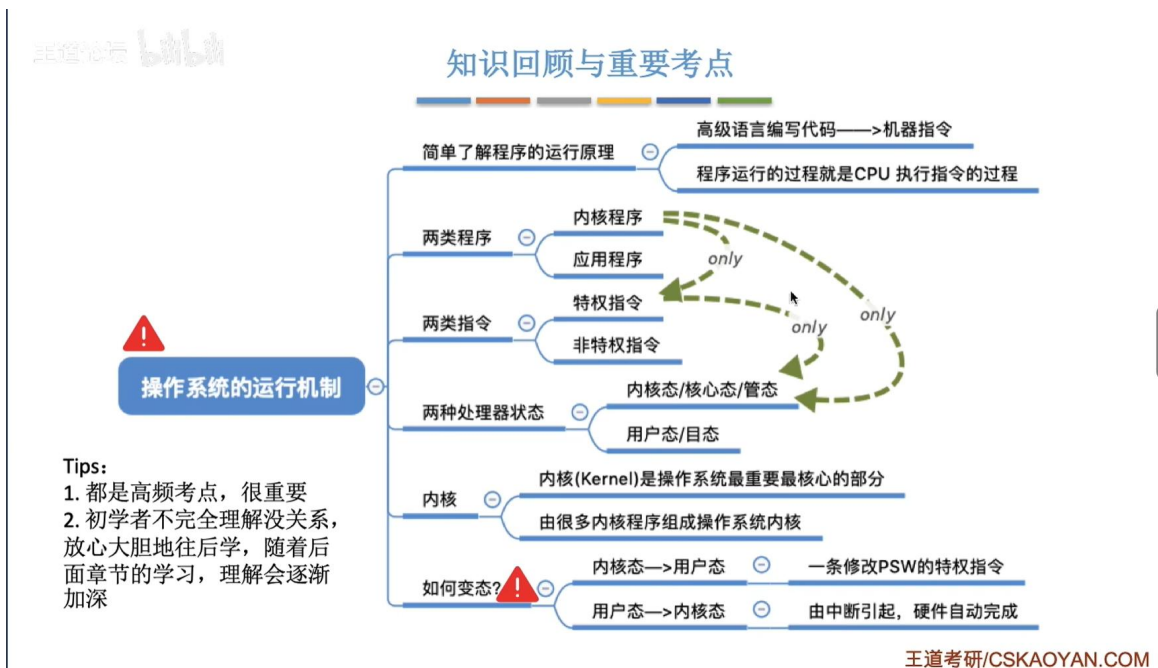


1.3.1 操作系统的运行机制

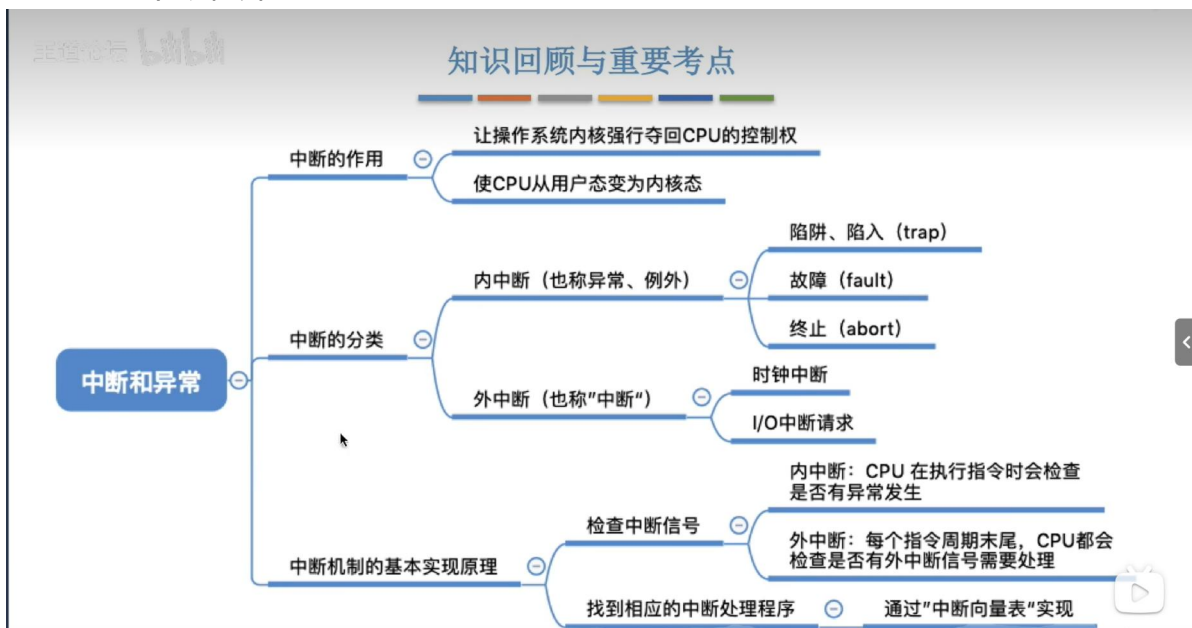


内核态（核心态，管态）和用户态（目态）：cpu的某种状态，内核态可以用特权指令。用户态只能用非特权指令。

psw：程序状态字program status word，存储0/1，表示用户态/内核态

内核态用户态的切换：内核-用户：执行一条特权指令，使psw更改，cpu变成用户态。用户-内核：引发中断，然后硬件自动完成变态。

1.3.2 中断和异常



trap：应用程序可以调用trap指令，引发一个内部中断信号。trap是非特权指令。

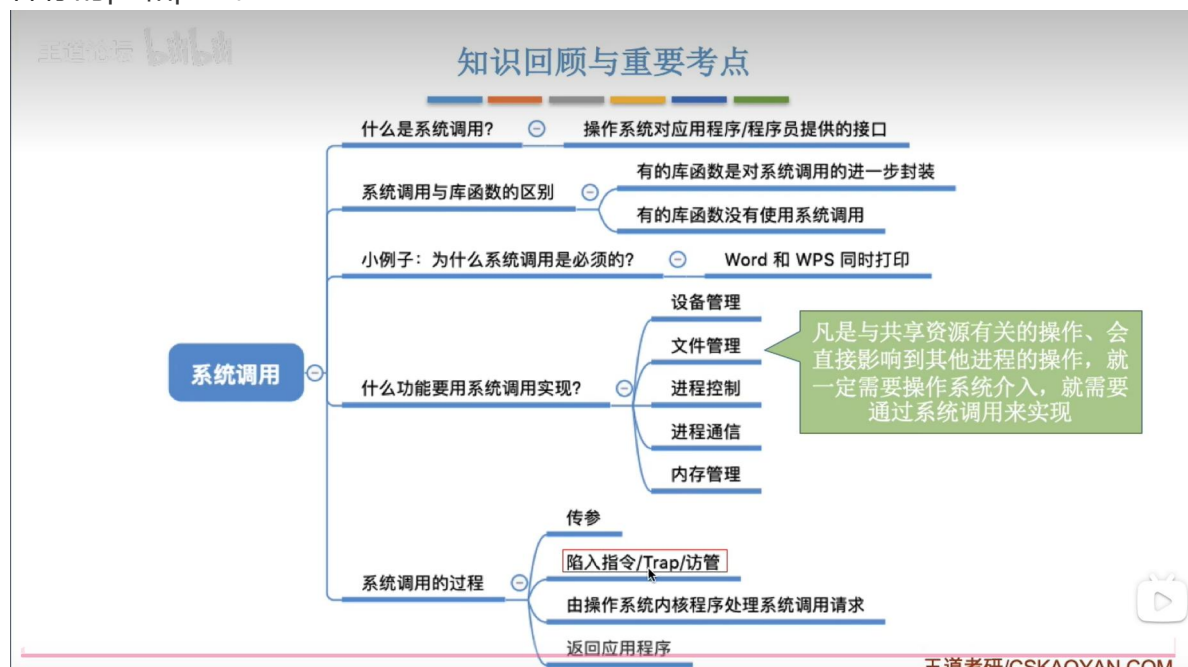
supervisor call: 是一种特殊的指令, 用于在用户程序和操作系统内核之间进行交互。当用户程序需要执行一些特权操作, 比如访问硬件设备、进行文件I/O、创建新进程等, 它可以通过发起SVC来请求操作系统内核的支持

仅有中断可以使操作系统夺回cpu使用权。

中断处理的过程: interrupt handler:

硬件部分: 当进程处理器发现中断信号时, 先把即将执行的指令的pc和psw存入control stack中, 然后处理器加载新的pc值(中断处理程序)。

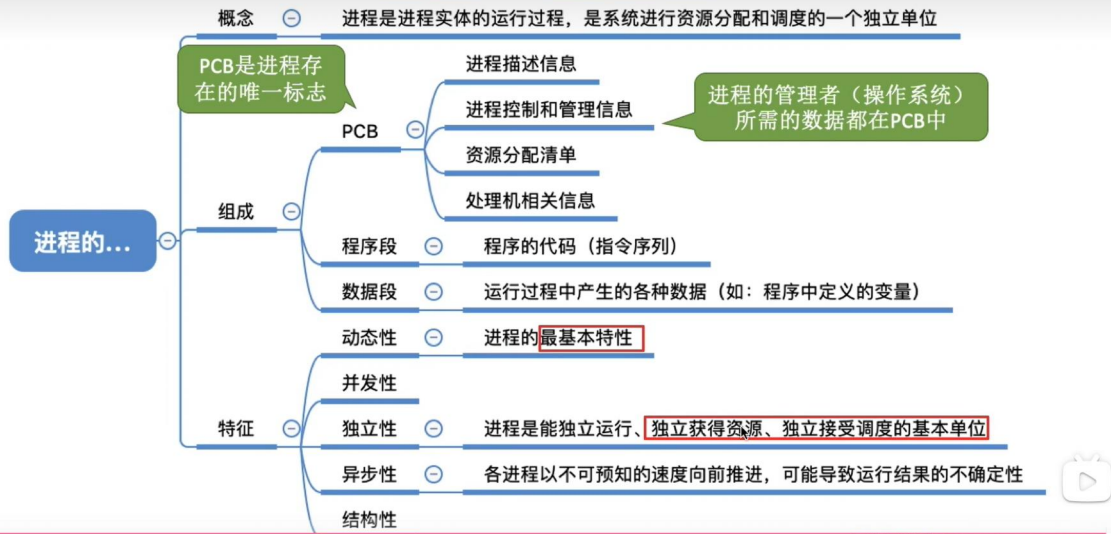
软件部分: 存储进程的剩余部分, 进程中断, 处理中断, 恢复软件存的进程, 恢复硬件存的pc和psw。



2 进程

进程的概念

知识回顾与重要考点



王道考研/CSKAOYAN.COM



18:37 / 19:05



已关闭弹幕

发送

1080P 高清

选集

倍速



进程的状态和转换

27人正在看

知识回顾与重要考点

运行状态 ○ CPU ✓ 其他所需资源 ✓

就绪状态 ○ CPU ✗ 其他所需资源 ✓

阻塞状态 ○ CPU ✗ 其他所需资源 ✗

创建状态 ○ 操作系统为新进程分配资源、创建PCB

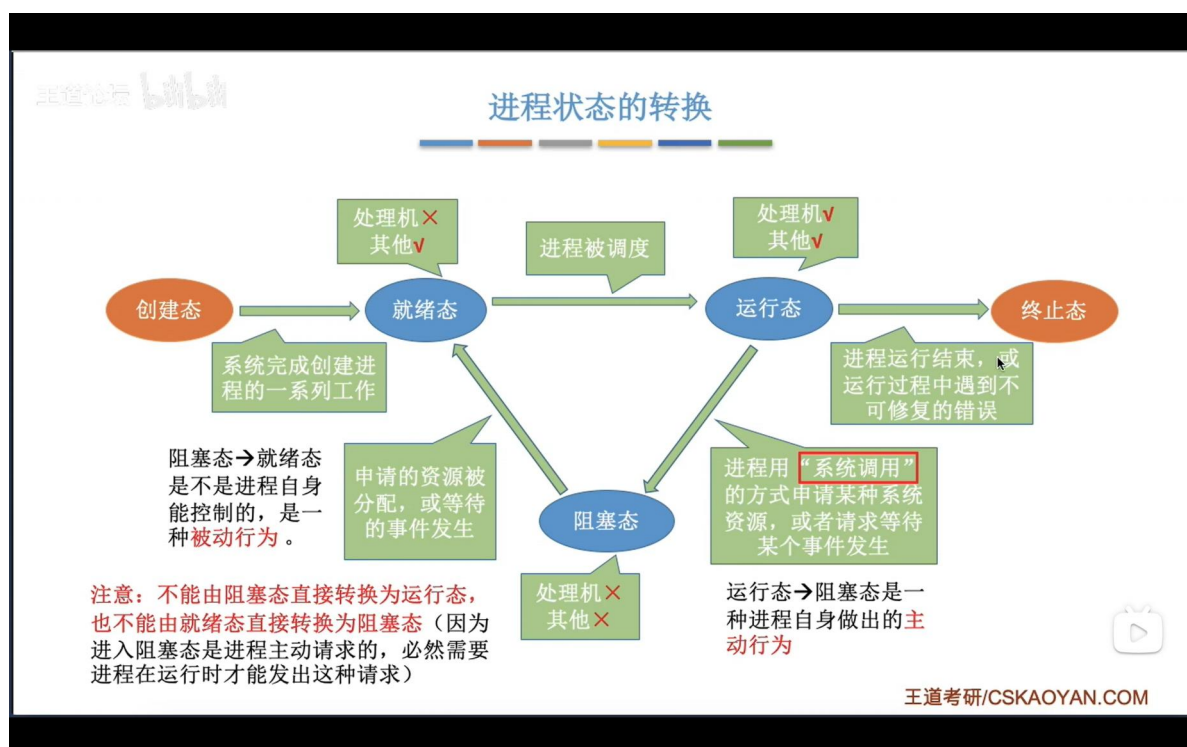
终止状态 ○ 操作系统回收进程的资源、撤销PCB

进程的...

进程状态间的转换

- 就绪态→运行态 ○ 进程被调度
- 运行态→就绪态 ○ 时间片到，或CPU被其他高优先级的进程抢占
- 运行态→阻塞态 ○ 等待系统资源分配，或等待某事件发生（主动行为）
- 阻塞态→就绪态 ○ 资源分配到位，等待的事件发生（被动行为）
- 创建态→就绪态 ○ 系统完成创建进程相关的工作
- 运行态→终止态 ○ 进程运行结束，或运行过程中遇到不可修复的错误

王道考研/CSKAOYAN.COM



进程控制

原语：原语就是实现一个不会被打断的函数。不会被打断是通过两个特权指令：关中断和开中断。先关在开。

进程通信

三种方式

1. 共享空间，共享空间需要是互斥的访问
2. 消息传递：直接传递和间接传递
通过两个原语send & receive实现。

23人正在看

消息传递（直接通信方式）

消息头包括：
发送进程ID、
接受进程ID、
消息长度等格
式化的信息

消息头
消息体

进程P 发送原语, `send(Q, msg)`

进程Q 接收原语, `receive(P, &msg)`

操作系统内核的
进程Q的PCB

内存

进程Q的消息队列

王道考研/CSKAOYAN.COM

消息传递（间接通信方式）

进程P 发送原语, `send(A, msg)`
往信箱A发送消息 `msg`

进程Q 接收原语, `receive(A, &msg)`
从信箱A接受消息

信箱A 信箱B

内存

可以多个进程往同一个信箱
`send`消息, 也可以多个进程
从同一个信箱中`receive`消息

间接通信方式, 以“信箱”作为中间实体进行消息传递。

王道考研/CSKAOYAN.COM

3.管道通信

特点：1.单向 2.FIFO，消息传递和共享空间不是FIFO的 3.单独一片内存区域 4.内存写满时，写进程将阻塞，管道空时，读进程阻塞。

3线程

线程的概念

一个进程可能有多个线程。线程是执行的最小单位。

线程也有对应的状态，就绪态-执行态。（阻塞态不按顺序）。

线程的特点：

1.共享进程资源

- 2.线程切换快，不用切换进程
- 3.TCB,线程控制块，记录线程基本信息

引入线程：进一步提升os的并发度，原本进程并发，现在线程并发。

线程的实现方式：ULT user-level-thread用户级线程，类似代码+while循环的方式，让用户设计线程，被称为ULT。并且cpu执行这个ULT时切换到用户态。

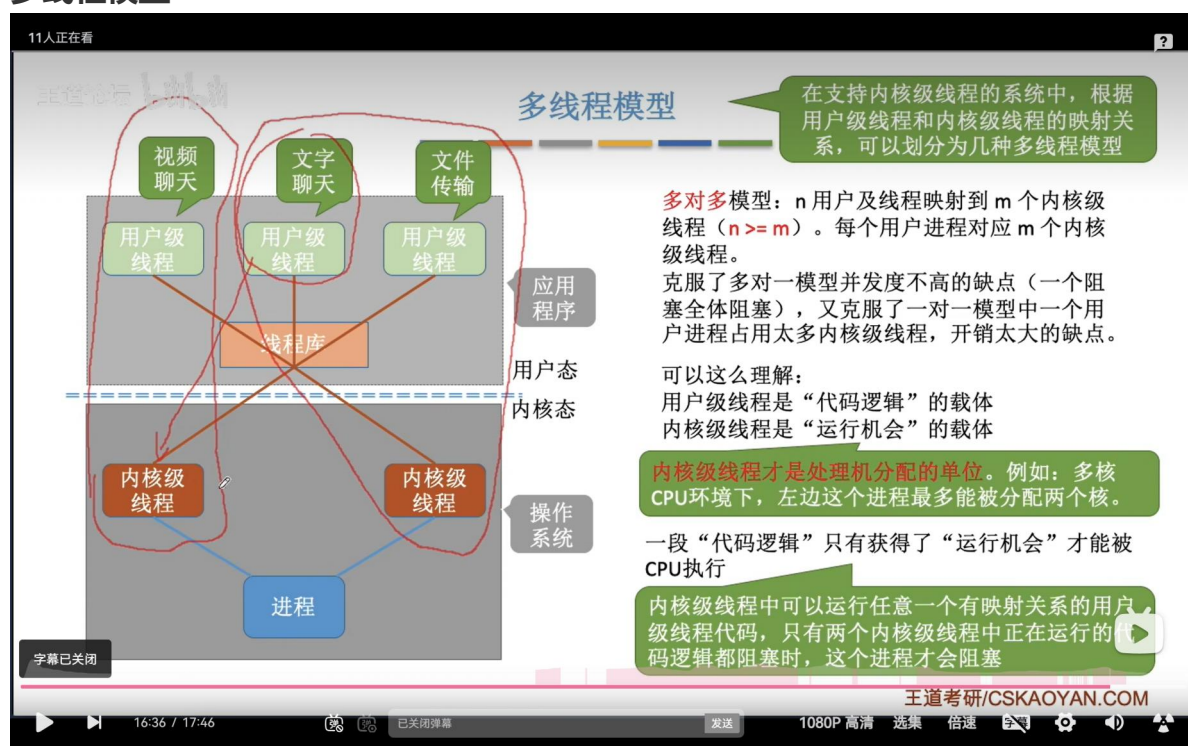
KLT

：内核级线程，kernel-level thread，一个内核级线程对应一个ULT，也就是说，线程切换需要cpu变态。

KLT的好处是：os调度的基本单位是thread，如果是多核cpu可以并行。

KLT的坏处是：线程切换麻烦。

多线程模型



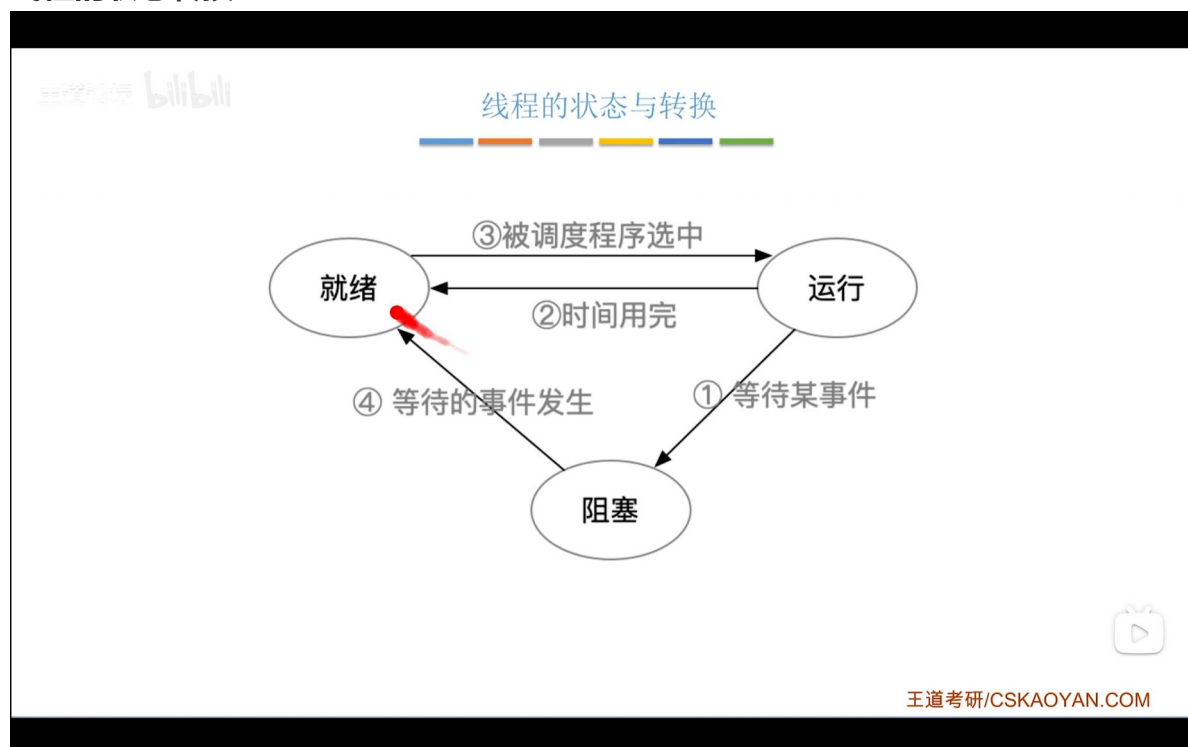
多对多模型：一个线程库给多个核用，一个线程库可以分开运行。

线程表

TCB: thread control table 线程控制块

Thread table:存储一个proc中的多个线程的TCB

线程的状态转换



调度和分派

Scheduler-Dispatcher

调度是：选择调度策略，即哪个进程先运行；以及选择运行的进程。

分派是：运行的具体过程，包括进程切换时上下文的保存，cpu的状态转换，地址切换到用户程序并运行。

处理机调度

1.高级调度：在外存的作业后备队列中，选取job。job是一个进程的集合，job是一个任务，调度job就是决定执行哪个job。

2.中级调度：选取挂起队列中的进程。

3.低级调度：调度进程的PCB，从**就绪队列**中选取一个进程。

我认为调度可以改名为选取。

挂起和阻塞

1.挂起：挂起是指把一个进程移出内存，并把PCB放入挂起队列中，需要运行进程时取回进程数据。

2.阻塞：阻塞的进程依旧在内存中，阻塞进程的PCB放置在阻塞队列中。一旦阻塞的事件完成，进程被移入就绪队列中。

调度的方式

1.剥夺调度方式（抢占式）

2.非剥夺式：等待proc主动让出cpu，

调度算法的评价指标

1.CPU利用率：CPU运行时间/周转时间

2.周转时间：turnaround, job完成时间-job开始时间，

平均周转时间 = 周转时间 / job数量

带权周转时间=周转时间/实际运行时间

响应时间=首次开始运行时间-arrival time

等待时间=周转时间-运行时间

调度算法

1.FCFS-serve

2.SJF-short job first, 短-运行时间短

3.HRRN=high response ratio Next高响应比优先, 响应比: 等待时间+剩余运行时间/剩余运行时间, HRRN是**非抢占式**

4.优先级

5.时间片轮转(roundrobin)

6.多级反馈(multilevel feedback), 抢占式。时间片+优先级 + n个队列 + FCFS

12人正在看

多级反馈队列调度算法

例题: 各进程到达就绪队列的时间、需要的运行时间如下表所示。使用**多级反馈队列**调度算法, 分析进程运行的过程。

进程	到达时间	运行时间
P1	0	8
P2	1	4
P3	5	1

P1(1) → P2(1)

优先级: 高 → 低

时间片: 小 → 大 (1, 2, 4)

第1级队列 → 使用CPU → 完成

第2级队列 → 使用CPU → 完成

第3级队列

设置多级就绪队列, 各级队列**优先级从高到低, 时间片从小到大**
新进程到达时**先进入第1级队列**, 按**FCFS**原则排队等待被分配时间片。若用完时间片进程还未结束, 则进程**进入下一级队列队尾**。如果此时**已经在最下级的队列**, 则**重新放回最下级队列队尾**
只有第 **k** 级队列为空时, 才会为 **k+1** 级队头的进程分配时间片

王道考研/CSKAQYAN.COM

越靠后的队列时间片越大。越靠前的队列越早被运行。

按先后顺序插入到队列中

7.多级队列调度, n个优先级队列, 越靠前的队列优先级越高。但是队列是按进程类型分类的, 每个队列可以再采取各自的调度算法。

8.SRTF: short remaining time first, 最短运行时间优先, 抢占式的SJF。

9.SPN:shortest process next, 同SJF。

临界区问题

critical section problem.当进程使用共享资源时, 称进程处于临界区(CS)。CS状态下是不允许调度的, 也就是说, CS状态下的进程不允许切换状态。并且CS状态下的进程必须: **互斥+无饥饿**

同步和互斥

1. 同步：进程之间有某种合作，这种合作体现在进程执行的顺序，原本是独立平行执行，现在是按某种顺序执行。
2. 互斥
3. 软件方法：单标志法，双标志先检查，双标志后检查，peterson。turn, flag, flag+turn, turn做谦让。

王道考研

Peterson 算法

算法思想：结合双标志法、单标志法的思想。如果双方都争着想进入临界区，那可以让进程尝试“孔融让梨”（谦让）。做一个有礼貌的进程。

```
bool flag[2];           //表示进入临界区意愿的数组，初始值都是false
int turn = 0;           //turn 表示优先让哪个进程进入临界区

P0 进程:
flag[0] = true;         ①
turn = 1;               ②
while (flag[1] && turn==1); ③
critical section;       ④
flag[0] = false;        ⑤
remainder section;

P1 进程:
flag[1] = true;         ⑥ //表示自己进入临界区
turn = 0;               ⑦ //可以优先让对方进入临界区
while (flag[0] && turn==0); ⑧ //对方想进，且最后一次是自己“让梨”，那自己就循环等待
critical section;       ⑨
flag[1] = false;        ⑩ //访问完临界区，表示自己已经不想访问临界区了
remainder section;
```

背后的含义：“表达意愿”
表达“谦让”

进入区: 1. 主动争取; 2. 主动谦让; 3. 检查对方是否也想使用，且最后一次是不是自己说了“客气话”

谁最后说了“客气话”，谁就失去了行动的优先权。
Eg: 过年了，某阿姨给你发压岁钱。

场景一
阿姨：乖，收下阿姨的心意~
你：不用了阿姨，您的心意我领了
阿姨：对阿姨来说你还是个孩子，你就收下吧
结局...

王道考研/CSKAOYAN.COM

4. 硬件方法：关中断+cs访问+开中断；TestAndSet(和锁差不多);
Swap(exchange);

TestAndSet指令

简称 TS 指令，也有地方称为 TestAndSetLock 指令，或 TSL 指令

TSL 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用C语言描述的逻辑

```
//布尔型共享变量 lock 表示当前临界区是否被加锁
//true 表示已加锁, false 表示未加锁
bool TestAndSet (bool *lock){
    bool old;
    old = *lock; //old用来存放lock 原来的值
    *lock = true; //无论之前是否已加锁, 都将lock设为true
    return old; //返回lock原来的值
}
```

```
//以下是使用 TSL 指令实现互斥的算法逻辑
while (TestAndSet (&lock)); //“上锁”并“检查”
临界区代码段...
lock = false; //“解锁”
剩余区代码段...
```

Swap指令

有的地方也叫 Exchange 指令，或简称 XCHG 指令。

Swap 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用C语言描述的逻辑

```
//Swap 指令的作用是交换两个变量的值
Swap (bool *a, bool *b) {
    bool temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
//以下是用 Swap 指令实现互斥的算法逻辑
//lock 表示当前临界区是否被加锁
bool old = true;
while (old == true)
    Swap (&lock, &old);
临界区代码段...
lock = false;
剩余区代码段...
```

逻辑上来看 Swap 和 TSL 并无太大区别，都是先记录下此时临界区是否已经被上锁（记录在 old 变量上），再将上锁标记 lock 设置为 true，最后检查 old，如果 old 为 false 则说明之前没有别的进程对临界区上锁，则可跳出循环，进入临界区。

信号量（必考）Semaphores

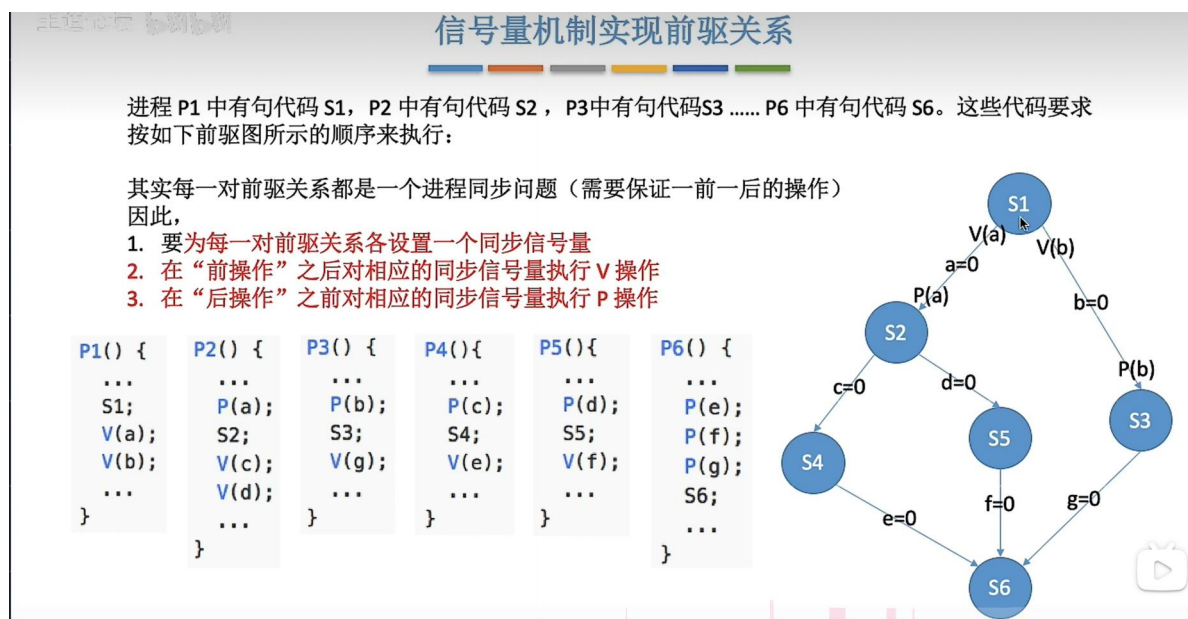
1. 信号量：

```
1 struct semaphore{
2     int value;
3     struct queue q;
4 }
5 void wait(s:semaphore){
6     s.count--;
7     if s.count<0
8     {queue.append(proc);}
9     block proc;
10 }
11 void signal(s){
12     s.count++;
13     if (s.count<0){//如果队列中还有在等待的话
14         queue.popout(proc);
15         ready proc;
16     }
17 }
```

s.count:当count<=0时，count的绝对值表示queue中有多少proc在等待。当count>0时，count表示还有多少资源可以使用。

2. 信号量可以实现同步、互斥和前驱

```
1 互斥：初始化s为1
2    wait();cs;signal();
3 同步：proc2必须在proc1执行完signal之后才能执行，这里假设s初始化为0
4    Proc1:
5        代码1;
6        代码2;
7        signal(s);
8        代码3
9    Proc2:
10       wait(s);
11       代码4;
12 前驱：
```



图中V等价为signal，P等价为wait

生产者消费者问题

问题简介：生产者生产某样东西，输入到buffer中，然后消费者获取。

条件：1.生产者生产必须互斥，否则可能会生产到同一个buffer中

2.buffer有限，当buffer满时，不允许生产。当buffer空时，不允许消费。

3.消费也需要互斥。

实现方法：1.信号量Mutex 2.信号量Full 3.信号量empty

如何实现

生产者、消费者共享一个初始为空、大小为n的缓冲区。
只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。
只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。
缓冲区是临界资源，各进程必须互斥地访问。

```
semaphore mutex = 1; //互斥信号量，实现对缓冲区的互斥访问
semaphore empty = n; //同步信号量，表示空闲缓冲区的数量
semaphore full = 0; //同步信号量，表示产品的数量，也即非空缓冲区的数量
```

实现互斥是在同一进程中进行一对PV操作

```
producer () {
    while(1) {
        生产一个产品;
        P(empty); //消耗一个空闲缓冲区
        P(mutex);
        把产品放入缓冲区;
        V(mutex);
        V(full); //增加一个产品
    }
}
```

```
consumer () {
    while(1) {
        P(full); //消耗一个产品（非空缓冲区）
        P(mutex);
        从缓冲区取出一个产品;
        V(mutex);
        V(empty); //增加一个空闲缓冲区
        使用产品;
    }
}
```

注意持有信号量的顺序。full和empty都不能放在mutex中，否则如果empty=0时，占用了mutex，直接死锁了。

如果将empty和full同时放入mutex，那也一样。

```
Initialization: S.count=1; //mutual excl.
                N.count=0; //full spaces
                E.count=k; //empty spaces
```

Producer:	Consumer:
While (1)	While (1)
{ produce v;	{ wait(N);
wait(E);	wait(S);
wait(S);	w:=take();
append(v);	signal(S);
signal(S);	signal(E);
signal(N);	consume(w);
}	}
append(v):	take():
b[in]:=v;	w:=b[out];
in++;	out++;
mod k;	mod k;
	return w;

多消费者多生产者：父母和儿女问题

如何实现

问题：可不可以不用互斥信号量？

```
semaphore mutex = 1; //实现互斥访问盘子（缓冲区）
semaphore apple = 0; //盘子中有几个苹果
semaphore orange = 0; //盘子中有几个橘子
semaphore plate = 1; //盘子中还可以放多少个水果
```

```
dad () {
    while(1) {
        准备一个苹果;
        P(plate);
        把苹果放入盘子;
        V(apple);
    }
}

mom () {
    while(1) {
        准备一个橘子;
        P(plate);
        把橘子放入盘子;
        V(orange);
    }
}

doughter () {
    while(1) {
        P(apple);
        从盘中取出苹果;
        V(plate);
        吃掉苹果;
    }
}

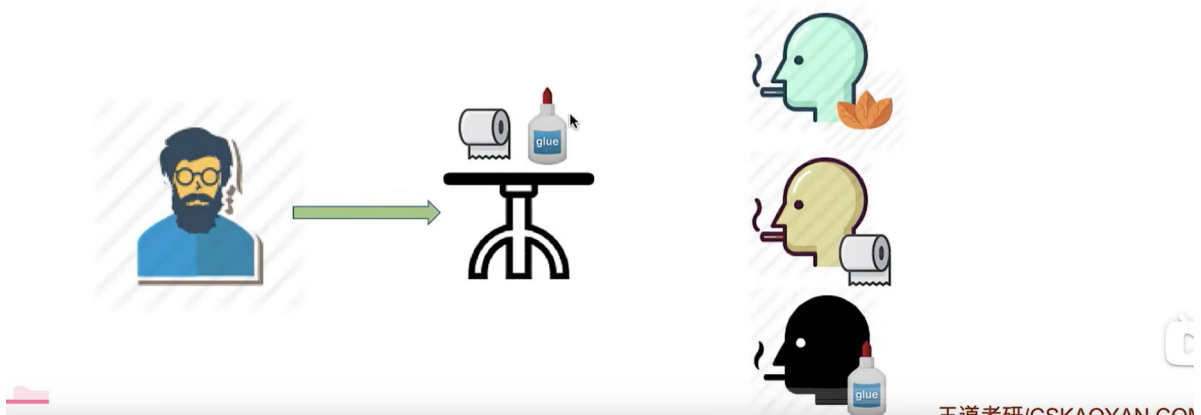
son () {
    while(1) {
        P(orange);
        从盘中取出橘子;
        V(plate);
        吃掉橘子;
    }
}
```

分析：刚开始，儿子、女儿进程即使上处理机运行也会被阻塞。如果刚开始是父亲进程先上处理机运行，则：父亲 P(plate)，可以访问盘子→母亲 P(plate)，阻塞等待盘子→父亲放入苹果 V(apple)，女儿进程被唤醒，其他进程即使运行也都会阻塞，暂时不可能访问临界资源（盘子）→女儿 P(apple)，访问盘子，V(plate)，等待盘子的母亲进程被唤醒→母亲进程访问盘子（其他进程暂时都无法进入临界区）→.....

吸烟者问题

问题描述

假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但是要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草、第二个拥有纸、第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放桌子上，拥有剩下那种材料的抽烟者卷一根烟并抽掉它，并给供应者进程一个信号告诉完成了，供应者就会放另外两种材料再桌上，这个过程一直重复（让三个抽烟者轮流地抽烟）

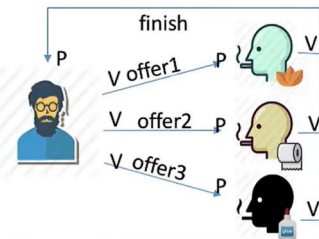


如何实现



是否需要设置一个专门的互斥信号量？

缓冲区大小为1，同一时刻，四个同步信号量中至多有一个的值为1



```
provider () {  
    while(1) {  
        if(i==0) {  
            将组合一放桌上;  
            V(offer1);  
        } else if(i==1) {  
            将组合二放桌上;  
            V(offer2);  
        } else if(i==2) {  
            将组合三放桌上;  
            V(offer3);  
        }  
        i = (i+1)%3;  
        P(finish);  
    }  
}
```

```
semaphore offer1 = 0; //桌上组合一的数量  
semaphore offer2 = 0; //桌上组合二的数量  
semaphore offer3 = 0; //桌上组合三的数量  
semaphore finish = 0; //抽烟是否完成  
int i = 0; //用于实现“三个抽烟者轮流抽烟”
```

```
smoker1 () {  
    while(1) {  
        P(offer1);  
        从桌上拿走组合一; 卷烟; 抽掉;  
        V(finish);  
    }  
}
```

```
smoker2 () {  
    while(1) {  
        P(offer2);  
        从桌上拿走组合二; 卷烟; 抽掉;  
        V(finish);  
    }  
}
```

```
smoker3 () {  
    while(1) {  
        P(offer3);  
        从桌上拿走组合三; 卷烟; 抽掉;  
        V(finish);  
    }  
}
```

小总结：同步互斥问题说白了就是先V后P问题，通过不断地VP顺序使进程按自己想要的顺序运行。

读写问题

问题描述：1.读者可以同时读文本 2.写者要互斥的写文本 3.如果写者要写文本，读者不能读文本。 4.写者如果要写，就必须尽快安排，不能让其饿死。

实现：

1,2：一个lock，读者共享这个lock，写者每个都要这个lock

```
1 struct semaphore {  
2     int value;  
3 }  
4 semaphore lock1 = 1; //只有一个Lock1会导致reader不同时，原因是，如果两个同时都是count==0要求lock1时。
```

```

5 semaphore lock2 = 1;
6 semaphore lock3 = 1; //只有两个lock时，如果读者获取了lock1，并且读者
  很多，那写者会饿死。
7 int count;//global
8 void reader()
9 {
10     P(lock3);
11     P(lock2);
12     if count==0 P(lock1);
13     count++;
14     V(lock2);
15     V(lock3);
16     read;
17     P(lock2);//count--需要lock2的原因是，否则两个reader的count也会
  同时为0，直接把lock2变成大于1了。
18     count--;
19     if count==0 V(lock1);
20     V(lock2);
21 }
22 void writer()
23 {
24     P(lock3);
25     //lock3可以让read不能无限的占有lock1，假设读者1-读者2-写者1-读者
  3，
26     //首先读者1，读者2会依次进入read状态，
27     //这时读者3不能更改count，然后read过程结束后，最终会释放lock1，
28     //写者就可以运行。
29     P(lock1);
30     write;
31     V(lock1);
32     V(lock3);
33 }
34

```

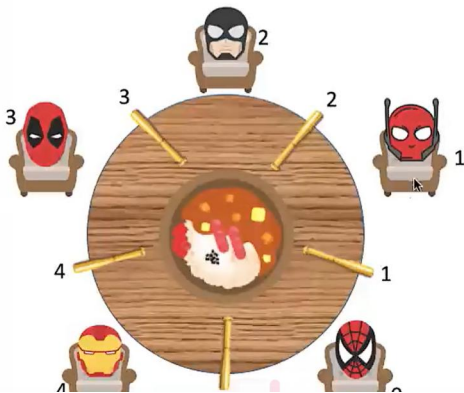
哲学家进食

哲学家问题就是拿筷子问题+避免死锁。

死锁问题

如何实现

一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子，桌子的中间是一碗米饭。哲学家们倾注毕生的精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿时，才试图拿起左、右两根筷子（一根一根地拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。



让奇数号拿左边筷子，偶数号拿右边。

如何防止死锁的发生呢？

①可以对哲学家进程施加一些限制条件，比如最多允许四个哲学家同时进餐。这样可以保证至少有一个哲学家是可以拿到左右两只筷子的

②要求奇数号哲学家先拿左边的筷子，然后再拿右边的筷子，而偶数号哲学家刚好相反。用这种方法可以保证如果相邻的两个奇偶号哲学家都想吃饭，那么只会有其中一个可以拿起第一只筷子，另一个会直接阻塞。这样避免了占有一支后再等待另一只的情况。

管程

我的理解是，把生产者消费者问题打包成一个数据结构，数据结构中包含了insert和takeout。

然后这个数据结构中自然实现了互斥和同步。

死锁

1. 死锁，饥饿，死循环：死锁指hold and wait,饥饿指永不被执行，死循环指进程代码写错了。
2. 预防死锁：1.破坏互斥 2.破坏hold and wait 3.抢占式 4.破坏循环等待
3. 避免死锁：银行家算法banker algorithm

安全序列：对于多个进程P1,P2,...，每个进程要求不同的资源，如果存在一个序列使得所有的进程都能够平安无事的使用资源释放资源，并且最终完成的话，那么这就是一个安全序列。

存在安全序列的状态叫做安全状态。

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

(d) P3 runs to completion

Safe sequence

P2
↓
P1
↓
P3
↓
P4

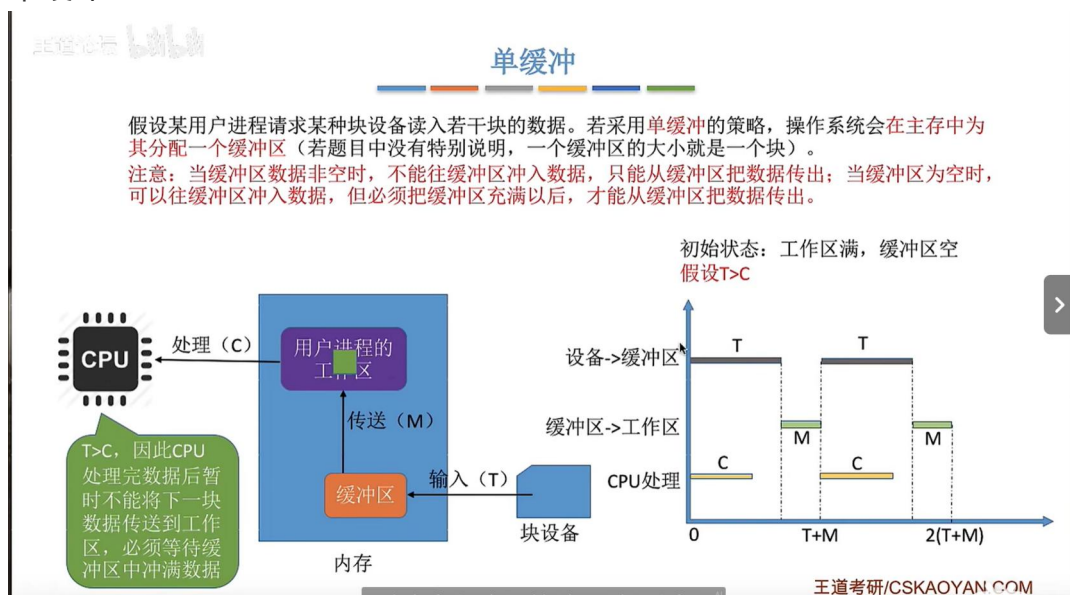
4. 死锁检测，死锁检测和找安全序列的区别是一样的。如果找不到安全序列就说明死锁了。

检测到死锁后，操作系统一般会采取1.挂起某个进程，释放其资源供其他进程 2.回退，让多个进程回退到没有抢占资源的时候 3.撤销，撤销某些进程，这方法太严重了。

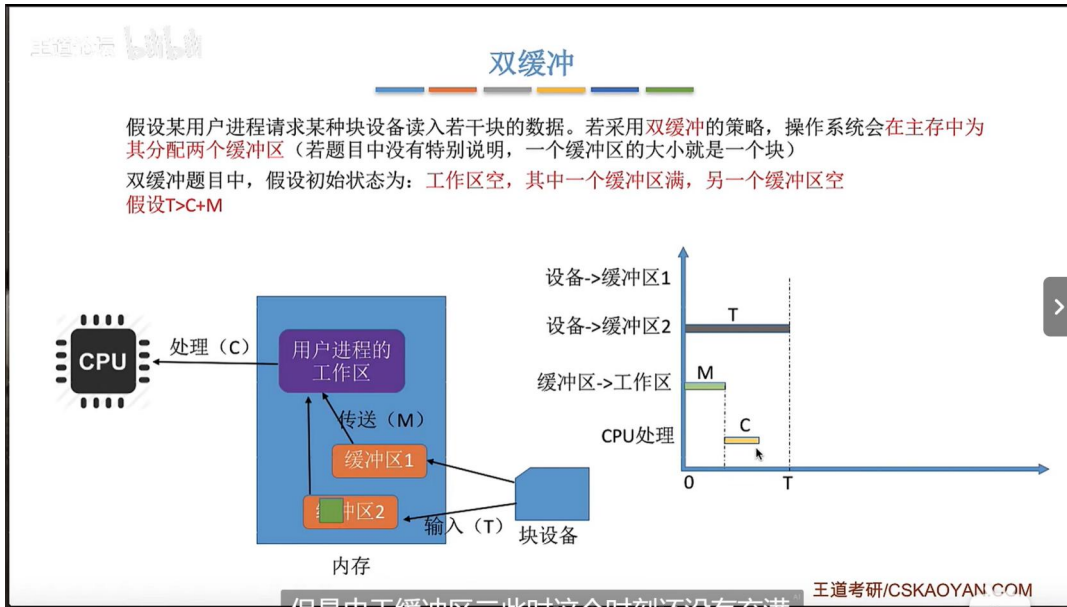
IO

缓冲：主要是要会画这个缓冲的过程图，首先是IO传到缓冲区，然后缓冲区才能传到工作区，工作区再传到CPU。

1. 单缓冲：



2. 双缓冲:

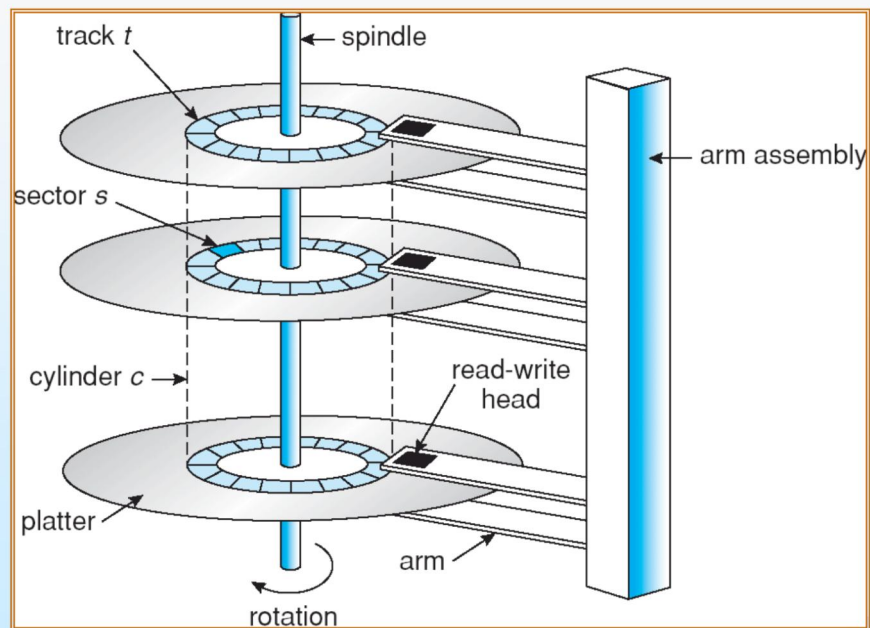


Disk

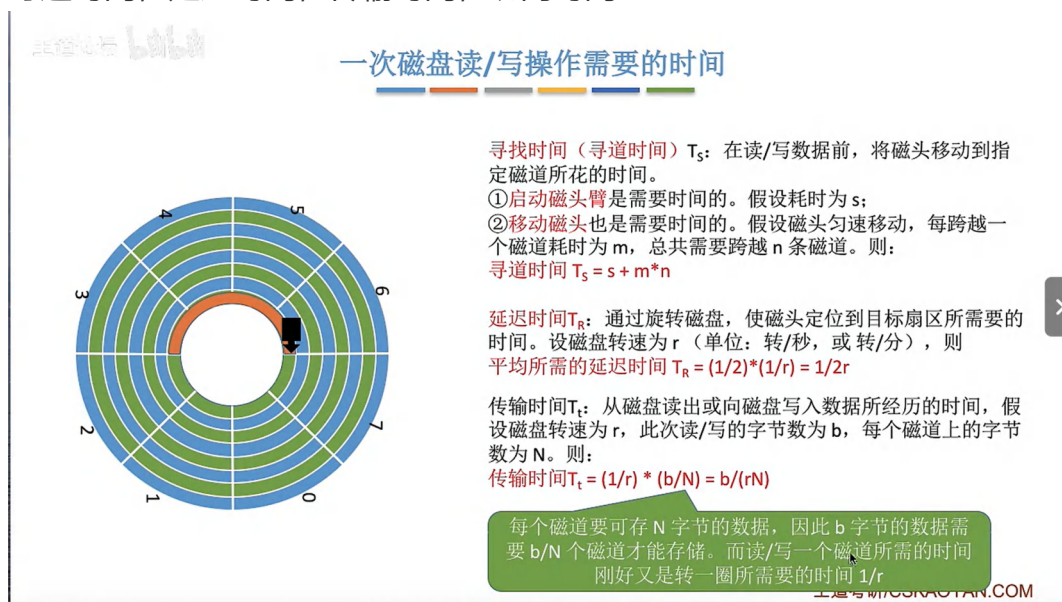
1. 柱面: cylinder
2. 盘面: platter
3. 盘区: sector



Moving-head Disk Mechanism



4. 寻道时间, 延迟时间, 传输时间, 访问时间:



disk调度策略

1. FCFS: 先来先服务
2. SSTF: 最短寻道时间优先
3. SCAN: 回到头，去到尾
4. C-SCAN: 不撞南墙不回头
5. C-LOOK: 撞到南墙前就回头
6. N-STEP-SCAN: FCFS-SCAN
7. FSCAN: 两个queue，一个用SCAN，一个存新请求

6, 7可以避免粘连问题, arm-stickness

disk格式化:

1. 物理格式化: 给磁盘分区，并且添加一些检验文件是否完整的校验部分。一个扇区分为头部，尾部，和数据部。头部尾部就放这种检验的东西。
2. 逻辑格式化: 创建文件系统，创建一个根目录和维系目录的一堆东西。初始化存储空间的数据结构。
3. BOOT Block: 引导块，用以初始化系统，开启os，初始化寄存器等。

note8考点:

1. 操作系统是如何管理空闲内存的: 链表
2. 交换: backing store是什么?
3. 页表是如何记录页的帧的, LA如何变成PA。LA如何记录。valid位, modified位, contro位。
4. 多级页表如何根据LA查找页号。
5. TLB计算平均访问时间。

6. 段表和页表的不同：段表记录<开始地址,段长度>。
7. 段页表如何找到PA。

note9考点：

1. 缺页中断机制
2. 缺页策略：FIFO,LRU,CLOCK,LFU,MFU
3. 页面缓存
4. 内存分配策略：局部/全局，可变/固定，固定和全局不能等同
5. 工作集如何帮助全局分配。

note10考点：

1. 文件内部数据存储：(记录式only)，如何存储；顺序，索引，顺序索引，哈希。
2. RWX，111表示owner，110表示group，001表示public。
3. FCB，file control block。

note11考点：

1. disk属性之：file allocation policy：continugous,linked,indexed.
2. FAT项：文件名，起始块，长度
3. 不同policy的FAT差异：c和l都是2，indexed是文件名，索引表块。
4. 空闲空间存储：bit法，链表法，索引法。
5. open file table，进程和os的oft区别。
6. 硬链接软连接：索引节点|符号链。