# CPT104_四类典型：OS计算一遍过

## CPU调度 CPU scheduling

- cpu执行时间Burst Time：是一个进程在CPU上执行指令的时间长度。
- 周转时间Turnaround Time (TT): 流程从首次进入就绪状态到完成所花费的总时间

  周转时间 = 退出时间 - 到达时间
- 等待时间Waiting Time (WT): 进程/线程在就绪状态下等待 CPU 所花费的总时间。

  等待时间 = 周转时间 – 突发时间
- 响应时间Response time: 进程首次获取（进入） CPU 的时间

## 非RTOS调度

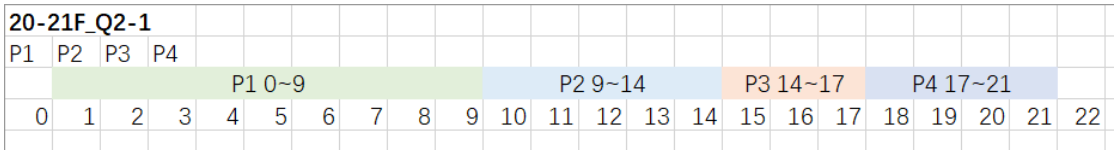
### 先到先得调度（First-Come First-Served, FCFS）

最朴素的调度方法，非抢占式。可以使用队列实现，**进程进入就绪队列时排到队尾，每当CPU空闲，从队首取出一个PCB运行。**平均等待时间长，性能差。

**20-21F_Q2-1**

| Process | Arrival time | Burst time |
|---|---|---|
| P1 | 0 | 9 |
| P2 | 1 | 5 |
| P3 | 2 | 3 |
| P4 | 3 | 4 |

**绘制甘特图（Gantt chart）**




| Process | Start time | End time |
|---|---|---|
| P1 | 0 | 9 |
| P2 | 9 | 14 |
| P3 | 14 | 17 |

| Process | Start time | End time |
| --- | --- | --- |
| P4 | 17 | 21 |

- **平均周转时间（average turnaround time）**

TT=End time-Arrival time

ATT=(TT1+TT2+TT3+TT4)/4=((9-0)+(14-1)+(17-2)+(21-3))/4=13.75ms

- **平均等待时间（average waiting time）**

WT=End time-Arrival time-Burst time

AWT=(WT1+WT2+WT3+WT4)/4=((9-0-9)+(14-1-5)+(17-2-3)+(21-3-4))/4=8.5ms

若已计算出ATT，AWT=ATT-avg(Burst time)

## 最短作业优先调度（Shortest-Job-First, SJF）

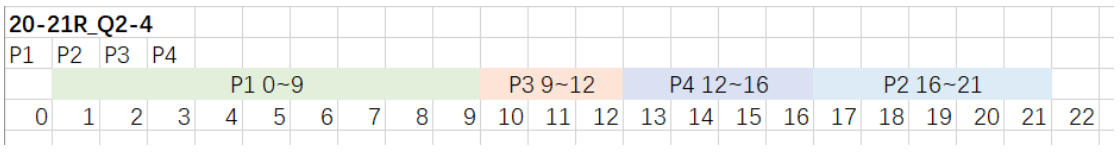
将Burst time短的进程安排在前面，且可以是**非抢占式或抢占式**的，默认前者，后者也被称为**最短剩余时间优先（SRTF）**。

优势：最大限度地减少平均等待时间和平均响应时间（最优，但现实中不知道进程长度，无法在短期CPU调度上实现）

劣势：难以预测未来每个进程的脉冲时间；会导致脉冲时间长的进程饿死

**20-21R_Q2-4**

| Process | Arrival time | Burst time |
| --- | --- | --- |
| P1 | 0 | 9 |
| P2 | 1 | 5 |
| P3 | 2 | 3 |
| P4 | 3 | 4 |




ATT=13ms

AWT=7.75ms

## 估计未来的CPU运行时间

使用过去运行的长度进行估计：下一个 = 所有过去运行时间的指数平均（exponential averaging）。

$$\tau_{n+1} = at_n + (1-a)\tau_n$$

一般a=0.5；若a=1，仅与上一次时间相关；若a=0，历史记录无作用（$\tau_{n+1}=\tau_n$）。其余情况下，a越大，受上一次影响越大；a越小，受历史平均影响越大。

**Assessment I**：predicted burst time using exponential averaging for the **fifth process;**

predicted burst time for the first process = **10,** first four processes is **2, 4, 6** and **8** units respectively, given **a = 0.5**.

pre1=10

pre2 = 2/2 +10/2 =6

pre3 = 4/2 + 6/2 =5

pre4 = 6/2 +5/2 =5.5

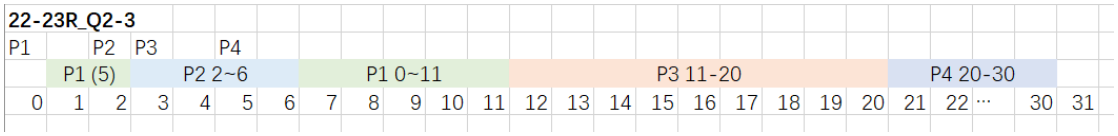
pre5 = 8/2 +5.5/2 =**6.75**

## 最短剩余时间优先调度（Shortest-Remaining-Time-First, SRTF）

**抢占式调度**，如果**新进程到达的突发时间比当前进程的剩余时间短**，则安排新进程。从而进一步减少平均等待时间和平均响应时间。

**22-23R_Q2-3**

| Process | Arrival time | Burst time |
|---|---|---|
| P1 | 0 | 7 |
| P2 | 2 | 4 |
| P3 | 3 | 9 |
| P4 | 5 | 10 |




ATT=14.25

AWT=14.25-7.5=6.75

## 循环调度（Round-Robin, RR）

每个进程都获得一小部分 CPU 时间（时间片time quantum/ time-slice），通常为 10-100 毫秒。在此段时间过后（elapsed），若进程未结束，此进程将被抢占（preempted）并添加入等待队列（ready queue）的末尾。对于**新进程，也会加入等待队列的末尾**，等待滚动至自己后运行。

1. 未完成进程：当一个进程的时间片用完且未完成时，该进程会被加入到轮询队列的末尾，等待下一次调度。
2. 新进程：当一个新的进程被创建并且准备好执行时，也会被加入到轮询队列的末尾。
3. 完成进程：立即取队首的进程开始执行

当时间片较大时，RR效率约等于FCFS；当时间片较小时，上下文切换不应占时间片q的比例太高，否则切换开销太大。

**W4_Tutorial:**

Time quantum = 4 ms

| Process | Arrival time | Burst time |
|---|---|---|
| P1 | - | 8 |
| P2 | - | 4 |
| P3 | - | 3 |

W4_Tutorial: Time quantum = 4 ms

| | P1 (4) | | | | P2 4~8 | | | | P3 8~11 | | | P1 0~15 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| | P1 | | | | P2 | | | | P3 | | | P1 | | | | |
| | P2 | | | | P3 | | | | P1 | | | | | | | |
| | P3 | | | | P1 | | | | | | | | | | | |

AWT=(7+4+8)/3=6.33ms

## 优先级调度（Priority scheduling）

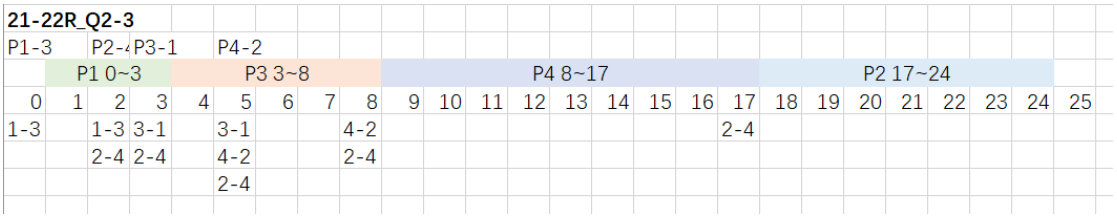
可以是**抢占式或非抢占式**的，有到达时间默认抢占式，每个进程都有对应优先级，最高优先级的进程会分配到CPU，相同级别按FCFS顺序调度。实际上SJF就是一种特殊的优先级调度。此方法会导致饥饿（starvation），解决方案是**老化（aging）**，逐渐增加等待很长时间的进程的优先级。

默认数字越小优先级越高。

**21-22R_Q2-3**

| Process | Arrival time | Burst time | Priority number |
|---|---|---|---|
| P1 | 0 | 3 | 3 |

| Process | Arrival time | Burst time | Priority number |
| --- | --- | --- | --- |
| P2 | 2 | 7 | 4 |
| P3 | 3 | 5 | 1 |
| P4 | 5 | 9 | 2 |




AWT=(3-0-3 +24-2-7 +8-3-5 +17-5-9)/4=4.5ms

## RTOS调度

**特征**：时间限制以周期period和截止日期deadline的形式出现。

周期period：是定期重复任务的迭代之间的时间量。

截止时间deadline：是必须完成操作的最长时间限制的约束。

- 中断延迟（Interrupt latency）（interrupt response time中断响应时间）：在当前中断任务上执行的最后一条指令与中断处理程序启动之间经过的时间。
- 派发延迟（Dispatch latency）：派发程序停止一个进程并启动另一个进程运行所需的时间。保持低派发延迟就是提供抢占式内核。

  降低派发延迟的最有效技术：提供抢占式内核（Provide preemptive kernels）

## 单调速率调度（Rate-monotonic）

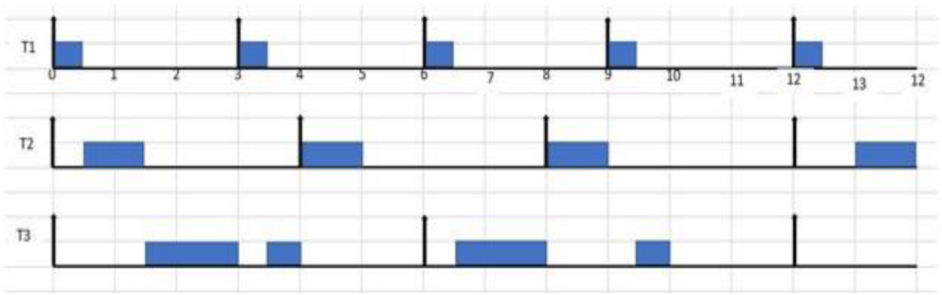
对周期性（periodic）任务的**抢占式**的，静态优先级的调度。优先级由进程的速率决定，越大越优先。

- **任务速率**（Rate of a periodic task） =1/p

  $0 \le t \le d \le p$

  周期period p；截止时间deadline d；处理时间processing time t

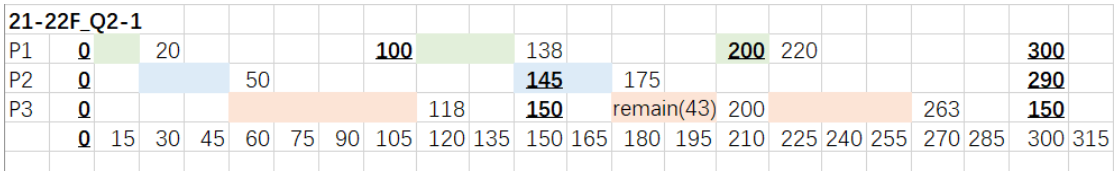
The timing diagram



**21-22F_Q2-1**（另有**20-21F_Q2-5**）

| Process | Execution time, t | Period, p | 计算任务速率 |
|---|---|---|---|
| P1 | 20 | 100 | 1/100 |
| P2 | 30 | 145 | 1/145 |
| P3 | 68 | 150 | 1/150 |

timing diagram:（这张图可能有点抽象）




## 计算CPU利用率

$CPUutilization = \sum_{i=0}^{i} \frac{ExeTime_i}{Period_i}$如果p0~pi是所有进程

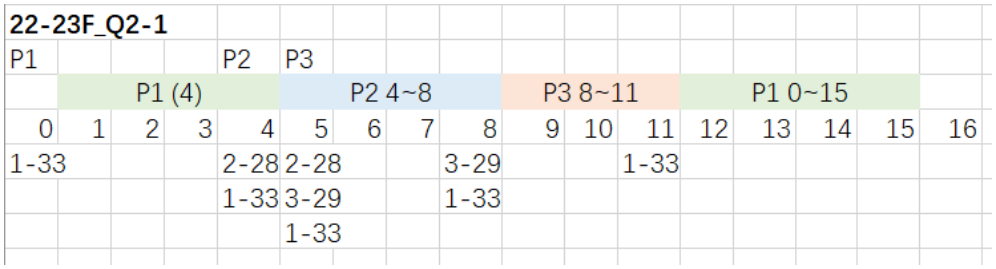
utilization=20/100+30/145+68/150=0.86

## 最早截止期限优先调度（Earliest-Deadline-First, EDF）

抢占式调度，根据截止期限动态调整优先级；**截止期限越近=优先级越高。**任务不一定是周期性的。

**22-23F_Q2-1**

| Process | Arrival time | Execution time | Deadline |
|---|---|---|---|
| P1 | 0 | 8 | 33 |
| P2 | 4 | 4 | 28 |
| P3 | 5 | 3 | 29 |




## 比例份额调度（Proportional Share）

调度器尝试保证每个作业都获取一定百分比的 **CPU** 时间，来取代对周转时间或是响应时间的优化。

假设有两个进程 A 和 B，且 A 有 75 张票证，B 仅有 25 张票证。因此，我们想要做的是另 A 获得 CPU 的 **75%**，而 B 保有剩下的 **25%**。

抽奖调度每隔一段时间（例如，一个时间片）便抽一次奖来概率地（即不确定性）实现这一点。抽一次奖是直接的：调度器一定知道总共有多少张票证（在我们的例子中，共有 **100**张）。调度器然后选取一张中奖票证。假设 A 持有的票证为 **0** 到 **74**，B 持有的票证为 **75** 到 **99**，中奖票证就决定是 A 运行还是 B 运行。然后调度器加载中奖进程的机器状态并运行它。

# 内存管理Memory management

## 页置换Page replace

在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。此时如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。

### 先进先出（First-In, First-Out, FIFO）

**22-23F_Q2-2**（另有**20-21F_Q2-3**，**21-22R_Q2-1**）

*Calculate the **number of page faults** for the following sequence of page references (each element in the sequence represents a page number) using the **First-In, First-Out (FIFO)** algorithm with **frame size of 3**.*

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

| 22-23F_Q2-2 | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 2 | 1 | 5 | 2 | 1 | 6 | 2 | 5 | 6 | 3 | 1 | 3 | 6 | 1 | 2 | 4 | 3 | |
| FIFO | | 1 | 2 | 3 | 3 | 3 | 5 | 5 | 1 | 6 | 2 | 5 | 5 | 3 | 1 | 1 | 6 | 6 | 2 | 4 | 3 | |
| | | | 1 | 2 | 2 | 2 | 3 | 3 | 5 | 1 | 6 | 2 | 2 | 5 | 3 | 3 | 1 | 1 | 6 | 2 | 4 | |
| | | | | 1 | 1 | 1 | 2 | 2 | 3 | 5 | 1 | 6 | 6 | 2 | 5 | 5 | 3 | 3 | 1 | 6 | 2 | |
| | | F | F | F | | | F | | F | F | F | F | | F | F | | F | | F | F | F | fault=14 |
| LRU | | 1 | 2 | 3 | 2 | 1 | 5 | 2 | 1 | 6 | 2 | 5 | 6 | 3 | 1 | 3 | 6 | 1 | 2 | 4 | 3 | |
| | | | 1 | 2 | 3 | 2 | 1 | 5 | 5 | 1 | 6 | 2 | 5 | 6 | 3 | 1 | 3 | 6 | 1 | 2 | 4 | |
| | | | | 1 | 1 | 3 | 2 | 1 | 2 | 5 | 1 | 6 | 2 | 5 | 6 | 6 | 1 | 3 | 6 | 1 | 2 | |
| | | F | F | F | | | F | | | F | F | F | | F | F | | | | F | F | F | fault=12 |
| | | | | | | | | | better | | | | | | | | better | | | | | |

**Belady现象**：采用FIFO算法时，可能出现分配的物理页面数增加，缺页次数反而升高的异常现象。

### 最近最少使用算法（Least Recently Used, LRU）

**21-22F_Q2-2**（另有**22-23R_Q2-1**，**20-21R_Q2-1**）

示例同上，区别仅在命中时是否调换队列顺序。

### 有效存储访问时间(effective memory access time EAT)

页表缓存（Translation Lookaside Buffer, TLB）存放将虚拟地址映射至物理地址的标签页表条目，如果命中可以快速取得对应页表物理地址，然后再访问内存；若未命中，需要进入内存的（多级）页表中寻找，耗时较长，然后再访问内存。

- 未缺页$EAT = A(E + M) + (1 - A)(E + 2 * M)$

> 缺页$EAT = (1-p) * [A(E+M) + (1-A)(E+2*M)] + p*F$
>
> TLB查找时间=E；内存访问时间=M；命中率=A；未命中率=1-A；缺页错误率=p；处理缺页时间=F
>
> *如果有多级列表，如二级页表，需要改成*$(1-A)(E+(2+1)M)$

**22-23F_Q2-4**（另有**22-23R_Q2-2，20-21R_Q2-2**）

*A paging scheme uses a **Translation Lookaside Buffer (TLB)**. **A TLB access takes 20 ns**, and **a main memory access takes 50 ns**. What is the **effective access time** (in ns) if the Translation Lookaside buffer **TLB hit ratio is 70%** and there is no page fault?*

$EAT = 0.7 * (20+50) + (1-0.7) * (20+2*50) = 85ns$

也有可能反着问你，命中率达到多少时可以让EAT小于120ns？解方程即可

## 页大小计算

**W11_Tutorial:**

*Consider a simple paging system with the following parameters:*

- 2^31 bytes of addressable physical memory;
- page size of 2^10 bytes;
- 2^26 bytes of logical address space.

(a) How many bits are in a logical address? 26
(b) How many bytes in a frame? $2^{10}$（帧大小等于页大小）
(c) How many bits in the physical address specify the frame? 31-10=21（总物理内存能放下$2^{21}$个帧）
(d) How many entries in the page table? 26-10=16（逻辑空间的大小就是页数*页大小，逻辑空间是通过页表和其他内存管理机制映射到物理内存的）
(e) How many bits in each page table entry (assume each page table entry includes a valid/invalid bit). 21+1=22（帧数+脏位）

**20-21F_Q2-2**

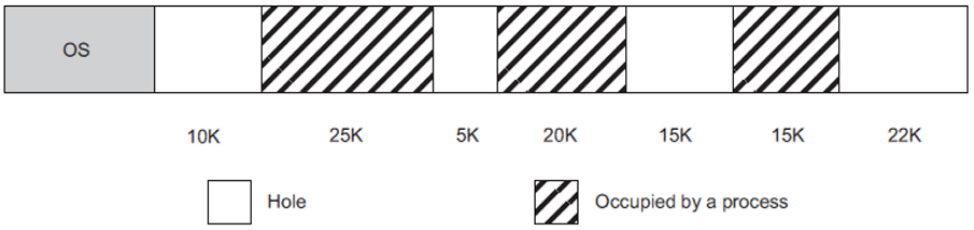
*There is a system with **64 pages** of **512 bytes page size** and a **physical memory of 32 frames**.* How many bits are required in the logical and physical address? (6')

- logical address：页码=log(64)=6bit，页面偏移=log(512)=9bit，逻辑地址=页码+页偏移=15bit
- physical address：帧码=log(32)=5bit，页面偏移=log(512)=9bit，物理地址=帧码+页偏移=14bit

## 内存分配

**21-22R_Q2-2**

*Consider the memory allocation scenario as shown in the figure below. Allocate memory for additional requests of **4K** and **10K** (in this order).* Compare the memory allocation, using **first-fit**, **best-fit**, and **worst-fit** allocation methods, in terms of internal fragmentation. (12')




- **first-fit 从内存列表的开始处开始，检查每个空闲块，直到找到一个足够大的空闲块来满足请求的大小。**

  4k in Hole=10k; 10k in Hole=15k

  Internal fragmentation =10-4+15-10=11k

- **best-fit 遍历整个内存列表，查找最接近请求大小的空闲块。**

  4k in Hole=5k; 10k in Hole=10k

  Internal fragmentation =5-4+10-10=1k

- **worst-fit 遍历整个内存列表，查找最大的空闲块，即使这个空闲块远大于请求的大小。**

  4k in Hole=22k; 10k in Hole=15k

  Internal fragmentation =22-4+15-10=23k

**20-21R_Q2-3**（另有**21-22F_Q2-5**）

*Three processes P1, P2, and P3 of size 19900, 19990, and 19888 bytes, respectively, need space in memory.*

- If partitions of equal size, that is, 20000 bytes, are allocated to P1, P2, and P3, will there be any **fragmentation** in this allocation? If, yes, then what is the size of the space left? (4')

  There will be Internal fragmentation 3*20000-19900-19990-19888=222bytes

- Can a process of 200 bytes be accommodated? (3')

  No, the left space of 222bytes is not contimuous, which do not have a contiuous space of 200bytes. So it cannot be accommodated.

## 寻址计算

**22-23F_Q2-5**

*A program has been divided into five modules. Their lengths and base addresses are stored in the segment table, as depicted in the following space:*

| Segment number | Length | Base address |
|---|---|---|
| 0 | 300 | 4000 |
| 1 | 600 | 1000 |
| 2 | 500 | 2700 |
| 3 | 900 | 1800 |
| 4 | 1000 | 2500 |

What will be the physical memory address for the following logical addresses? Show the physical memory mapping for the segments. (6')

| Segment number s | Offset d |
|---|---|
| 1 | 550 |
| 3 | 908 |
| 4 | 670 |

Segment存放偏移量offset，Length=limit代表栈顶，Base address=基地址

- Seg1：offset=550<600，合法

  物理地址=基址+偏移量=1000+550=1550

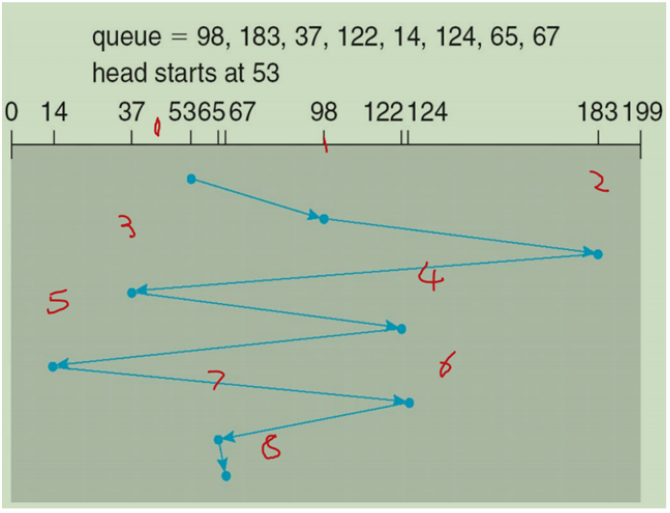
- Seg3：offset=908>900，非法

  由于偏移量超出段长度，这是一个无效的地址，因此无法转换为物理地址。

# 磁盘调度Disk scheduling

## 先来先服务（First-Come First-Served, FCFS）

磁头朴素地按队列顺序访问磁道。

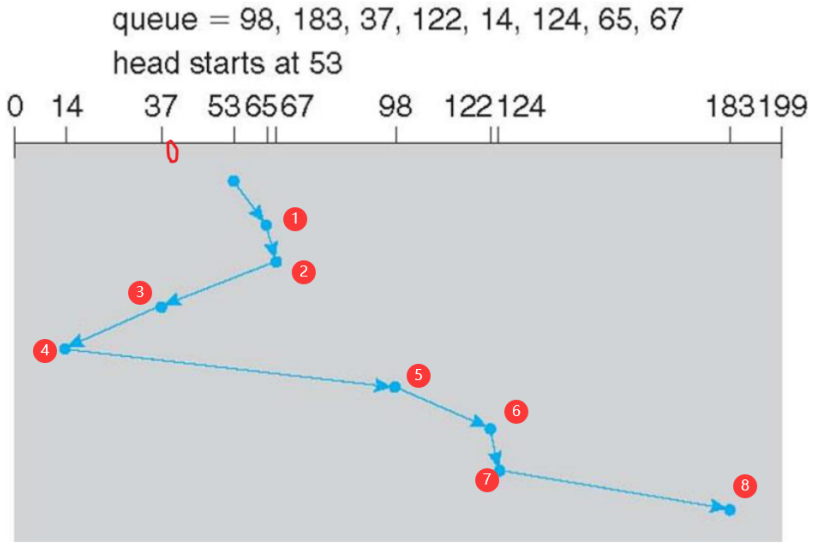



**Head movement**: = |53 − 98| + |98 − 183| + |183 − 37| + |37 − 122| + |122 − 14| + |14 − 124| + |124 − 65| + |65 − 67| = 640

见**22-23R_Q2-4**

## 最短寻道时间优先(Shortest Seek Time First, SSTF)

优先处理与当前磁头最近的磁道。 可以保证每次寻道时间最短，但是不能保证总的寻道时间最短。可能产生**饥饿**现象，磁头有可能在一个小区域内来回得移动。

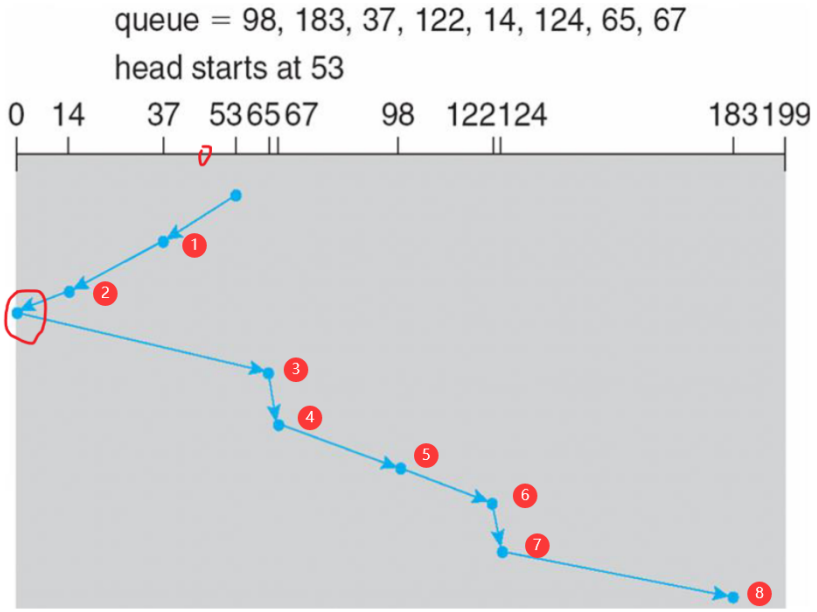



见**22-23F_Q2-3**（另见**20-21F_Q2-4**，**21-22R_Q2-4**）

## 扫描调度SCAN（Elevator）

从起始点开始，向磁盘的一端运动，经过有需求的柱面时处理请求，直到**遇到磁盘的另一端**，磁头运动方向翻转，继续处理请求。就像大楼里的电梯。

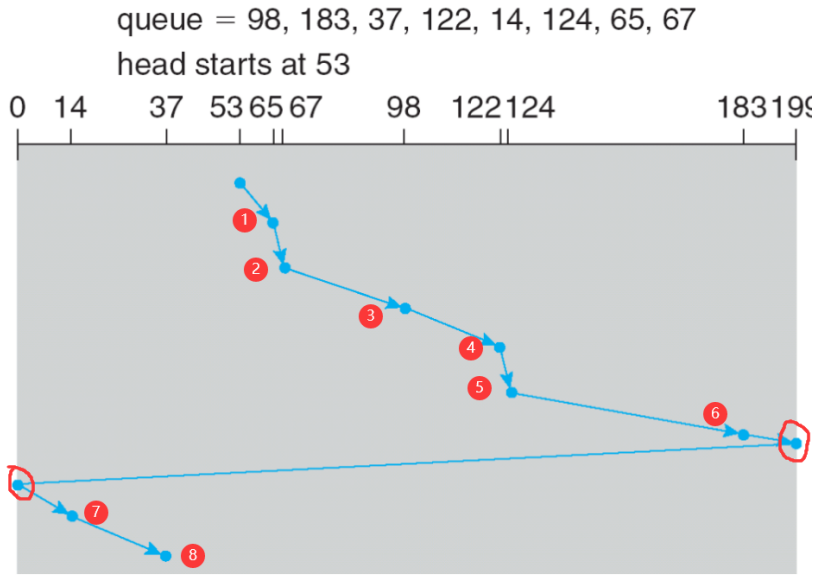
但由于起始方向不同可能会导致经过柱面数不同，最好两个方向都算一遍，取最小。




见**21-22F_Q2-3**

## 循环扫描（Circular-SCAN, C-SCAN）

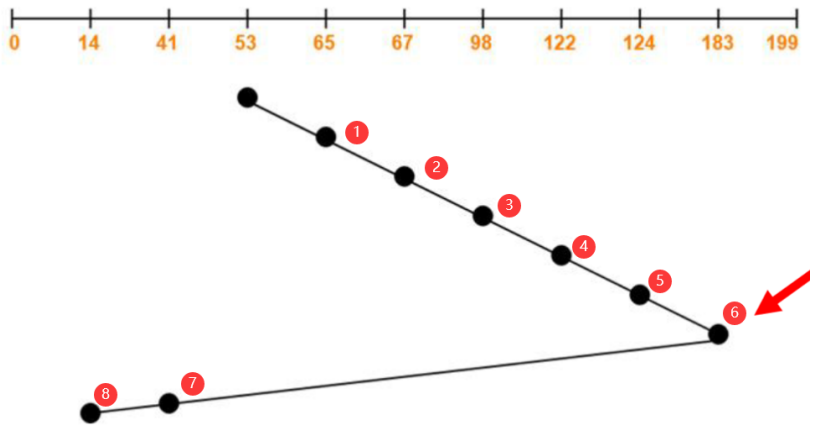
同上，但当磁头到达另一端时，立即回到磁盘的开头（速度比寻道快，但不好量化，故没出过题），不在回程处理任何请求。




## LOOK调度

相较于SCAN，磁头只会移动到一个方向的最远请求为止，然后调转方向。

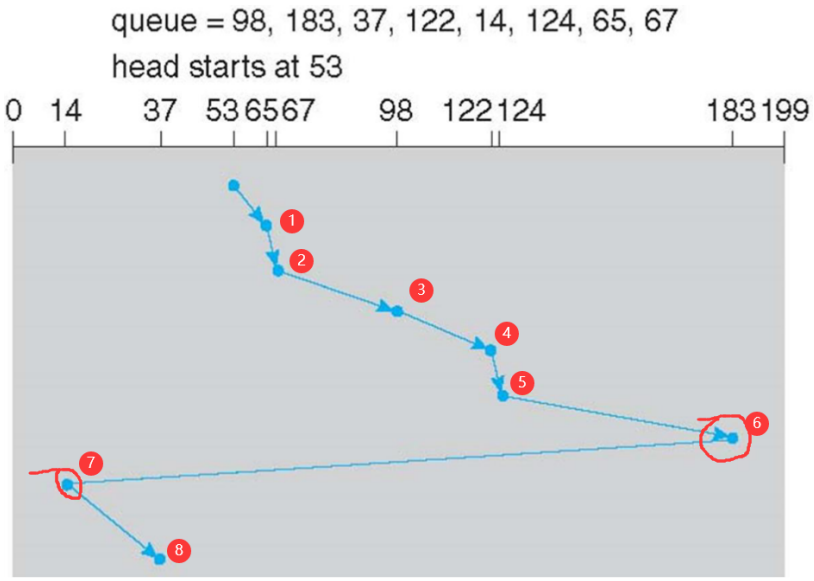
但由于起始方向不同可能会导致经过柱面数不同，最好两个方向都算一遍，取最小。




见**20-21R_Q2-5**

## C-LOOK调度（Circular-LOOK）

同上，但当磁头到达一端最远请求时，立即回到另一端最远的请求（速度比寻道快，但不好量化，故没出过题），不在回程处理任何请求。




## 资源分配Resource allocation

### **22-23F_Q3**（另见**21-22F_Q3**，**20-21F_Q3**，**22-23R_Q3**，**20-21R_Q3**）

22-23F_Q3

| Process | Max | | | Allocation | | |
|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R1 | R2 | R3 |
| P1 | 0 | 0 | 1 | 0 | 0 | 1 |
| P2 | 1 | 7 | 5 | 1 | 0 | 0 |
| P3 | 2 | 3 | 5 | 1 | 3 | 5 |
| P4 | 0 | 6 | 5 | 0 | 6 | 3 |

Available或直接给你，或给总资源数R按下式计算

Available=Recource Vector R - sum(Allocation)

| Process | Need=Max-Allocation | | |
|---|---|---|---|
| | R1 | R2 | R3 |
| P1 | 0 | 0 | 0 |
| P2 | 0 | 7 | 5 |
| P3 | 1 | 0 | 0 |
| P4 | 0 | 0 | 2 |

1 计算需求

| Allocate Process | Recource Available R1 | R2 | R3 | |
|---|---|---|---|---|
| | 1 | 5 | 2 | 如果有额外request，在Available减去即可 |
| P1 | 1 | 5 | 3 | Ava_{n+1}=Ava_{n}+Alloc_{P} |
| P3 | 2 | 8 | 8 | |
| P2 | 3 | 8 | 8 | |
| P4 | 3 | 14 | 11 | 所有进程都能安排→Safe state |

2 计算可用

3 计算分配

- Is this system currently in a safe or unsafe state? Why? Explain. (6')

  安全序列：P1,P3,P2,P4；由于找到了安全序列，系统当前处于安全状态。

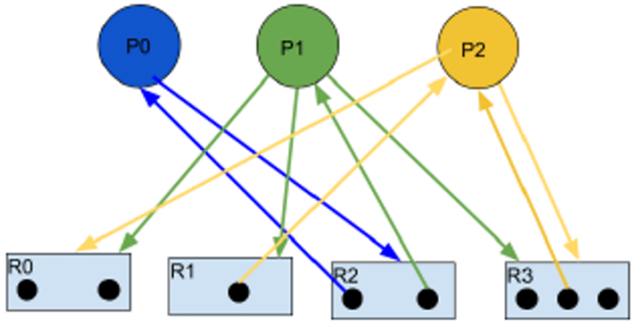
- If a request from P2 arrives for (0, 5, 0), can that request be safely granted immediately? Explain the answer. (6')

  Available变为[1,0,2]；类似地，找到安全序列：P1,P3,P2,P4；

  由于找到了安全序列，P2的请求（0,5,0）可以被安全地立即授予。

## 资源分配图（Resource Allocation Graph, RAG）

**21-22R_Q3**




Convert the RAG to the matrix representation (Allocation, Need and Available). (7')

- 从每个资源中的小黑点可以发现每个资源的实例数
- 从进程指向资源的代表进程的需求
- 从资源实例指向进程的代表进程已被分配的实例数量

| 21-22R_Q3 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Process | Allocation | | | | Need | | | |
| | R0 | R1 | R2 | R3 | R0 | R1 | R2 | R3 |
| P0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| P1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| P2 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| | | | | | | | | |
| | Recource Vector | | | | | | | |
| | R0 | R1 | R2 | R3 | | | | |
| | 2 | 1 | 2 | 3 | | | | |