第 37 章

1. 计算以下几组请求的寻道，旋转和传输时间 -a 0; -a 6; -a 30; -a 7,30,8; -a 10,11,12,13

阅读 readme 文件，其中有一些参数

- -G 可以查看可视化内容
- -c 可以计算结果。
- -a 提供待访问的数组
- -S 将寻道速率改为不同值（第 2 题）
- -R 将旋转速率修改为不同值（第 3 题）
- -p 提供调度算法，默认 FIFO，可以替换为 SATF,SSTF 等（第 4 题）
- -o 引入磁道偏移（第 6 题）

出现 ModuleNotFoundError: No module named 'tkinter'错误，表示 python3 没有安装可视化的 tkinter 包。

此时不能使用 pip install tkinter 来试图安装，即使换源也会显示失败。

一定要使用 sudo apt-get install python3-tk 来安装 tkinter 包。

此外，还详细介绍了计算过程，运行模拟器并计算一些基本的寻道、旋转和传输时间。

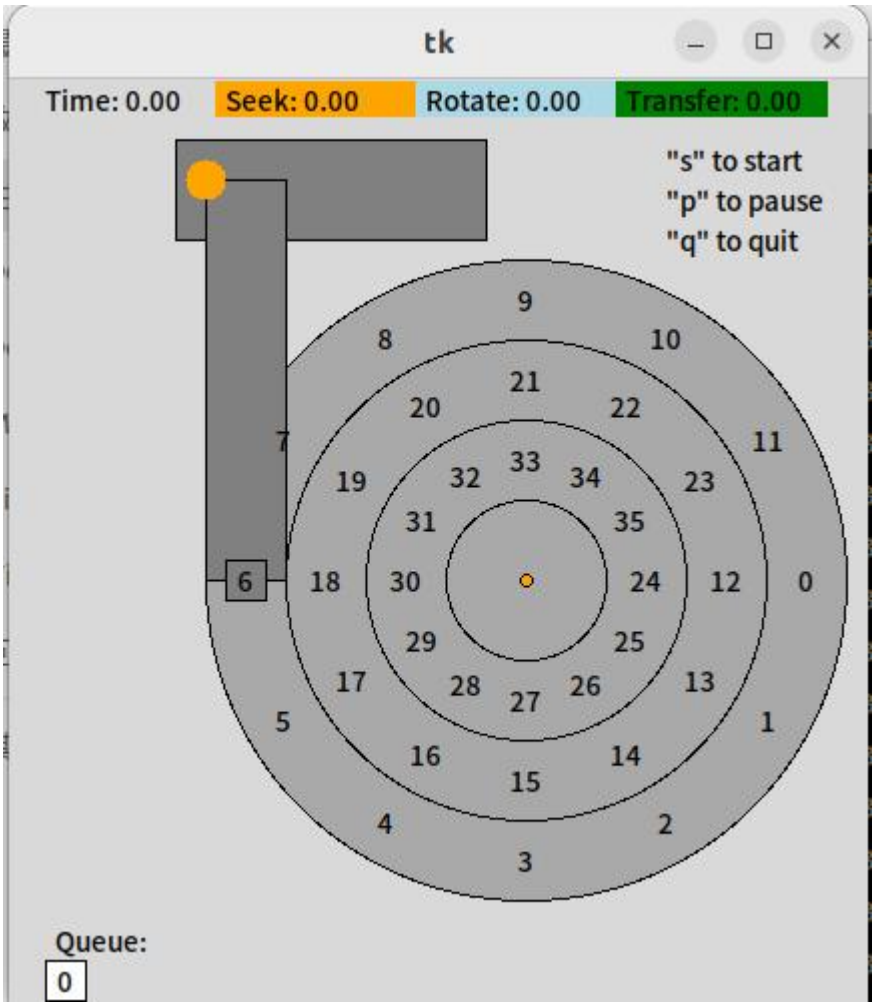
1）转速默认设置为每个时间单位 1 度。因此，要进行彻底的革命，需要 360 个时间单位。在默认磁盘中，每个磁道有 12 个扇区，这意味着每个扇区占用 30 度的旋转空间。

2）转移在扇区之间的中间点开始和结束，要读取一个扇区，需要 30 个时间单位（给定我们默认的旋转速度）

3）默认情况下，每条轨迹之间的距离为 40 个距离单位，默认搜索速率为每单位时间 1 个距离单位。因此，从外侧轨道到中间轨道的寻道需要 40 个时间单位。

磁盘均为逆时针旋转

（1） **-a 0;**



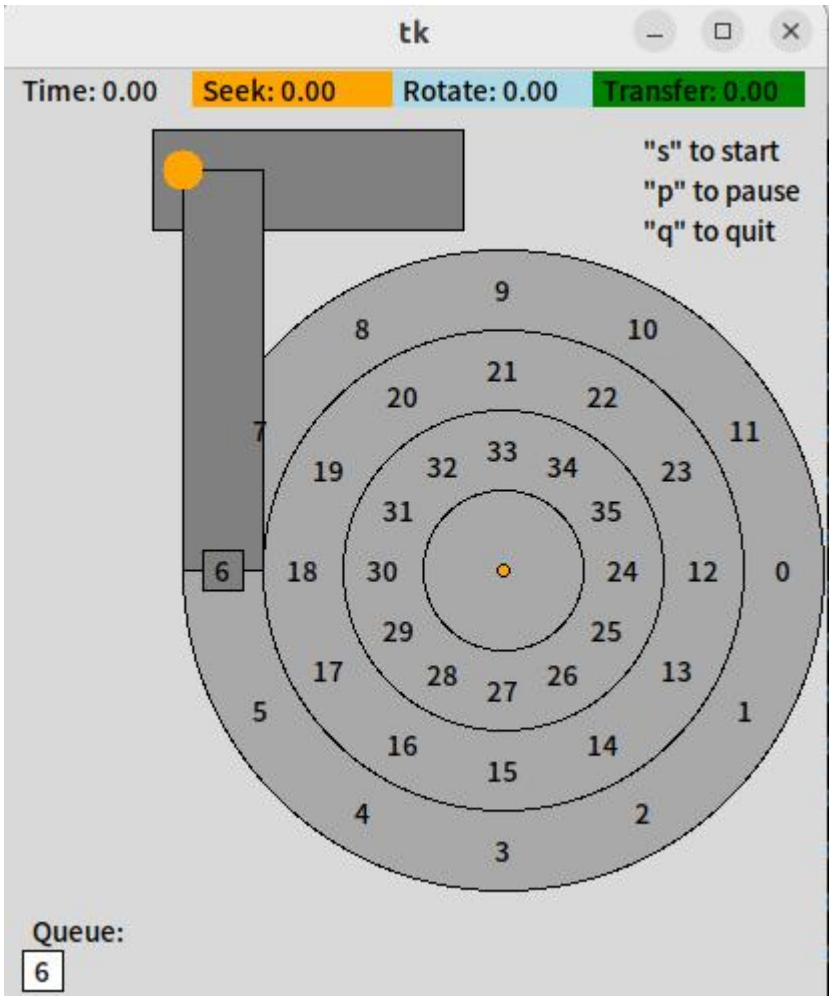
Block	T 寻道	T 旋转	T 传输	T 总
0	0	165	30	195

t 寻道=0（块位于同一条磁道上）
t 旋转=（12-6.5）*30=165
t 传输=30（11->0）
total=195

```
Block:   0  Seek:  0  Rotate:165  Transfer: 30  Total: 195

TOTALS      Seek:  0  Rotate:165  Transfer: 30  Total: 195
```

结果符合预期

（2） **-a 6;**



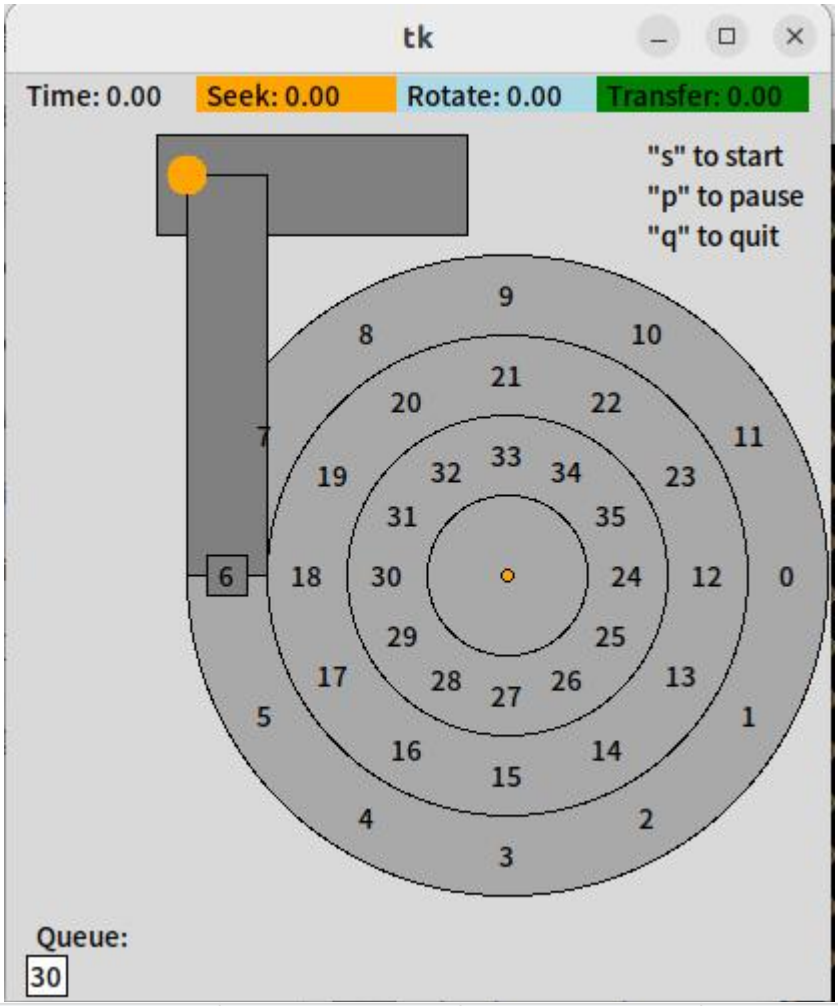
Block	T 寻道	T 旋转	T 传输	T 总
6	0	345	30	375

t 寻道=0（块位于同一条磁道上）
t 旋转=（（12-6.5）+6）*30=345
t 传输=30（6->7）
total=375

```
Block:   6  Seek:  0  Rotate:345  Transfer: 30  Total: 375

TOTALS      Seek:  0  Rotate:345  Transfer: 30  Total: 375
```

结果符合预期

（3） **-a 30;**



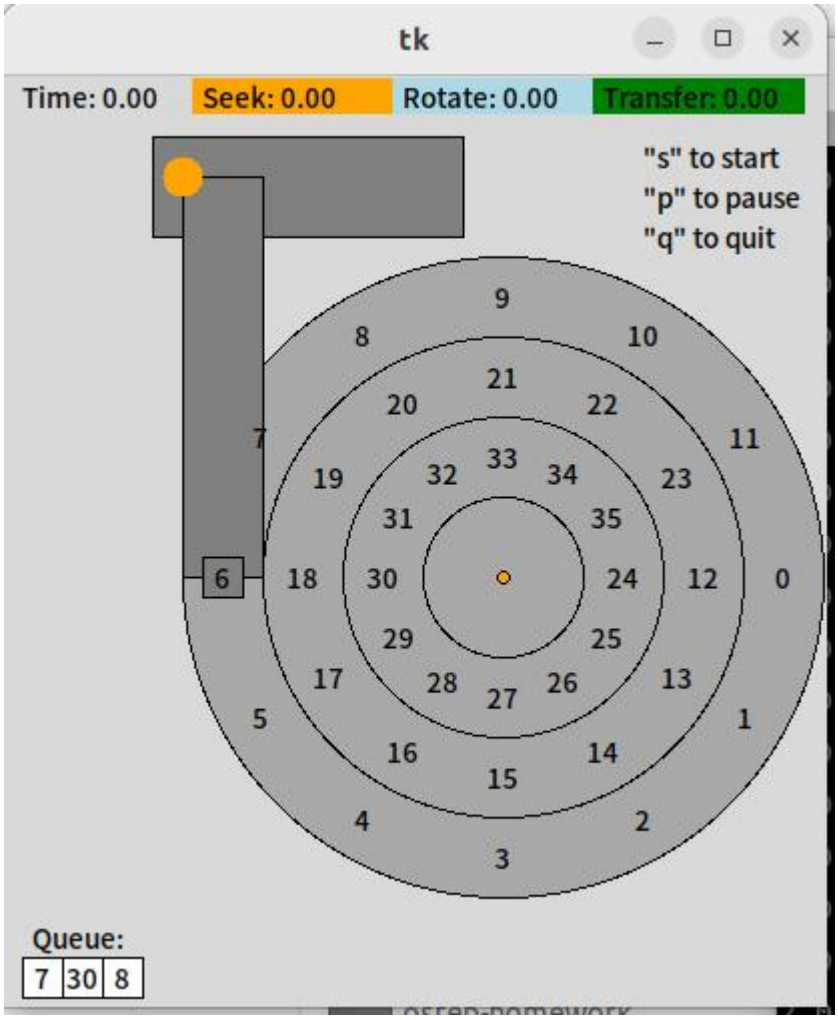
Block	T 寻道	T 旋转	T 传输	T 总
30	80	265	30	375

t 寻道=80（两次改变磁道）
t 旋转=（（12-6.5）+6）*30-80=345-80=265（30 对应的是 6，因为边寻道边旋转，所以需要减去寻道的时间）
t 传输=30（30->31）
total=80+265+30=375

```
Block:  30  Seek: 80  Rotate:265  Transfer: 30  Total: 375

TOTALS      Seek: 80  Rotate:265  Transfer: 30  Total: 375
```

结果符合预期

（4） **-a 7,30,8;**



Block	T 寻道	T 旋转	T 传输	T 总
7	0	15	30	45
30	80	220	30	330
8	80	310	30	420

这一问相比之前更复杂

7：

t 寻道=0

t 旋转=(7-6.5)*30=15

t 传输=30（7->8）

total=0+15+30=45

30：

t 寻道=80（两次改变磁道）

t 旋转=（（12-8）+6）*30-80=300-80=220（30 对应的是 6，因为边寻道边旋转，所以需要减去寻道的时间）

t 传输=30（30->31）

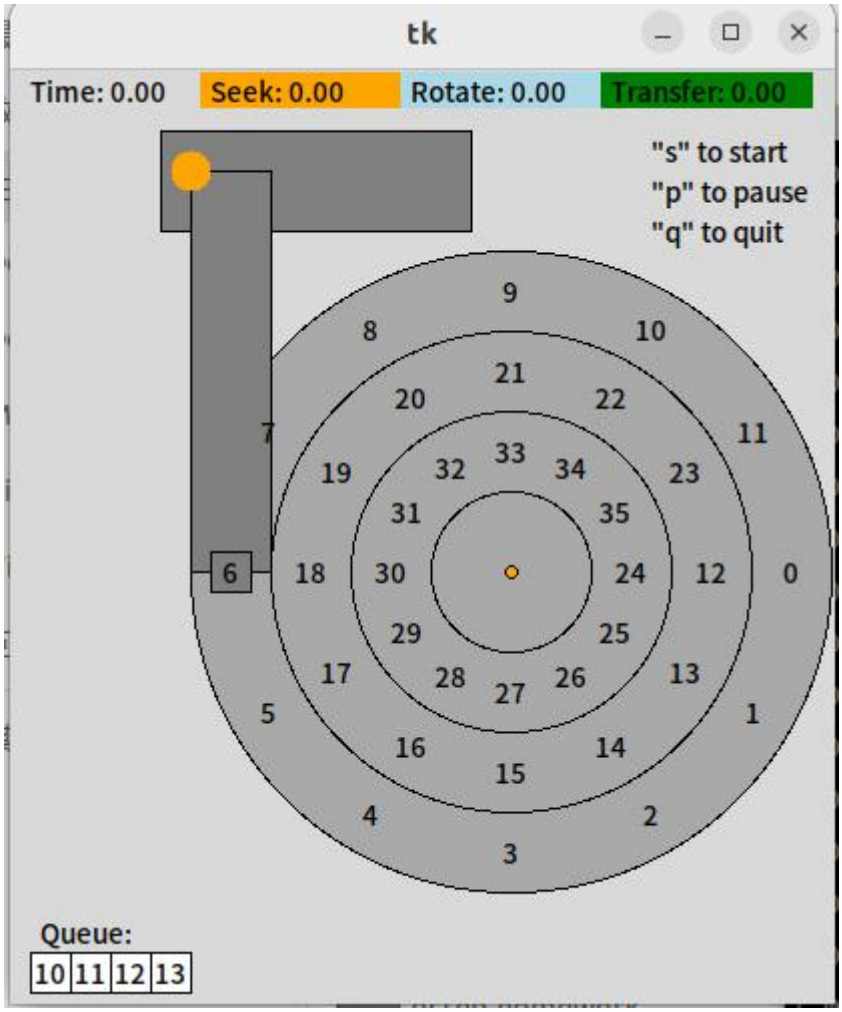
total=80+220+30=330

8：
t 寻道=80（两次改变磁道）
t 旋转=（8-7+12）*30-80=310（31 对应的是 7，边寻道边旋转反而超过了我们的目的地，最终计算的相当于又绕了一圈）
t 传输=30（8->9）
total=80+310+30=420

```
Block:   7  Seek:  0  Rotate: 15  Transfer: 30  Total:  45
Block:  30  Seek: 80  Rotate:220  Transfer: 30  Total: 330
Block:   8  Seek: 80  Rotate:310  Transfer: 30  Total: 420

TOTALS      Seek:160  Rotate:545  Transfer: 90  Total: 795
```

结果符合预期

（5） **-a 10,11,12,13**



Block	T 寻道	T 旋转	T 传输	T 总

Block	T 寻道	T 旋转	T 传输	T 总
10	0	105	30	135
11	0	0	30	30
12	40	320	30	390
13	0	0	30	30

这一问相对简单
10：
t 寻道=0
t 旋转=(10-6.5)*30=105
t 传输=30（10->11）
total=0+105+30=135
11:
t 寻道=0
t 旋转=0
t 传输=30（11->12）
total=0+0+30=30
12:
t 寻道=40
t 旋转=360-40=320(12 对应的是 0，本来 11 传输后指向了 0，但是经过寻道，相当于需要多绕一圈，再减去寻道的时间)
t 传输=30（12->13）
total=40+320+30=390
13:
t 寻道=0
t 旋转=0
t 传输=30（11->12）
total=0+0+30=30

```
Block:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Block:  11  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
Block:  12  Seek: 40  Rotate:320  Transfer: 30  Total: 390
Block:  13  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
```

结果符合预期

2. 执行上述相同请求,但将寻道速率更改为不同值:-S 2,-S 4, -S 8, -S 10,-S 40, -S 0.1 时间如何变化?
（1） -a 0；-a 6
对于这两个，不进行寻道，寻道时间为 0，所以不会变化
（2） 对于其他情况来说，可能会因为寻道时间的变化而出现多绕一圈的情况。

也有可能出现寻道时间减少不明显而保持旋转时间不变，总时间不变的情况。

对于-a 30

寻道速率 v	单次 T 寻道	T 寻道	T 旋转	T 传输	T 总	比较
1（原题）	40	80	265	30	375	
2	20	40	305	30	375	寻道时间短，不影响总时间
4	10	20	325	30	375	寻道时间短，不影响总时间
8	5	10	315	30	375	寻道时间短，不影响总时间
10	4	8	337	30	375	寻道时间短，不影响总时间
40	1	2	343	30	375	寻道时间短，不影响总时间
0.1	400	800	265	30	1095	寻道时间长，总时间增加

对于 30 这种情况，寻道时间的减少，不会导致我们多旋转来找到对应块，所以旋转时间+寻道时间保持不变，因此最终的时间不变

但是 v=0.1 的情况下，寻道时间大大增大，对旋转时间产生了影响，我们必须多耗费时间来到对应的磁道上，所以会导致我们多绕几圈，但是我们仍然需要满足寻道时间+旋转时间=345+360*n,所以在这里我们的旋转时间恰好仍然是 265，但是最终的时间为 800+265+30=1095

对于-a 7,30,8

寻道速率 v	单次 T 寻道	T 寻道	T 旋转	T 传输	T 总	比较
1（原题）	40	0+80+80=160	15+220+310=545	30+30+30=90	795	
2	20	0+40+40=80	15+260+350=625	30+30+30=90	795	寻道时间短，不影响总时间
4	10	0+20+20=40	15+280+10=305	30+30+30=90	435	寻道时间短，由于恰好避免一整圈，显著缩短总时间

寻道速率 v	单次 T 寻道	T 寻道	T 旋转	T 传输	T 总	比较
8	5	0+10+10=20	15+290+20=325	30+30+30=90	435	寻道时间短，由于恰好避免一整圈，显著缩短总时间
10	4	0+8+8=16	15+292+22=329	30+30+30=90	435	寻道时间短，由于恰好避免一整圈，显著缩短总时间
40	1	0+2+2=4	15+298+28=341	30+30+30=90	435	寻道时间短，由于恰好避免一整圈，显著缩短总时间
0.1	400	800+800=1600	15+220+310=545	30+30+30=90	2235	寻道时间长，总时间显著变长

在这里需要考虑三次的影响，对每次分别考虑

对于-a 10，11，12，13
这里也需要分别考虑吧这四次的影响

寻道速率 v	单次 T 寻道	T 寻道	T 旋转	T 传输	T 总	比较
1（原题）	40	0+0+40+0=40	105+0+320+0=425	30+30+30+30=120	585	

寻道速率 v	单次 T 寻道	T 寻道	T 旋转	T 传输	T 总	比较
2	20	0+0+20+0=20	105+0+340+0=445	30+30+30+30=120	585	寻道时间短，不影响总时间
4	10	0+0+10+0=10	105+0+350+0=455	30+30+30+30=120	585	寻道时间短，不影响总时间
8	5	0+0+5+0=5	105+0+355+0=460	30+30+30+30=120	585	寻道时间短，不影响总时间
10	4	0+0+4+0=4	105+0+356+0=461	30+30+30+30=120	585	寻道时间短，不影响总时间
40	1	0+0+1+0=1	105+0+359+0=464	30+30+30+30=120	585	寻道时间短，不影响总时间
0.1	400	0+0+400+0=400	105+0+320+0=425	30+30+30+30=120	945	寻道时间长，总时间显著变长

3. 同样的请求,但改变旋转速率:-R 0.1,-R 0.5,-R 0.01。时间如何变化?

随着旋转速率的降低，总的旋转时间和传输时间是在总体增大的，而且可能因为旋转时间的相对增大，从而少绕一圈，减少总时间。这个时候不需要再考虑寻道时间的区别，因为寻道时间会影响旋转时间，而旋转时间并不影响寻道时间。

例如-a 7，30，8 中的从 30 到 8 的过程中，旋转速率为 0.1 时，寻道完成后还未进入 8 的扇区，相比之前可以少绕一圈，只需要转 30 度，这时旋转时间为 30/0.1=300，比之前的 30+360-40=350 要短

对于-a 0：

旋转速率 v	旋转时间	传输时间
0.5	330	60
0.1	1650	300
0.01	16500	3000

对于-a 6：

旋转速率 v	旋转时间	传输时间
0.5	690	60
0.1	3450	300
0.01	34500	3000

对于-a 30:

旋转速率 v	旋转时间	传输时间
0.5	610	60
0.1	3370	300
0.01	34420	3000

对于-a 7,30,8:

旋转速率 v	旋转时间	传输时间
0.5	1250	180
0.1	3290	900
0.01	34340	9000

对于-a 10，11，12，13：

旋转速率 v	旋转时间	传输时间
0.5	890	240

旋转速率 v	旋转时间	传输时间
0.1	4610	1200
0.01	46460	12000

总结

对比图如下

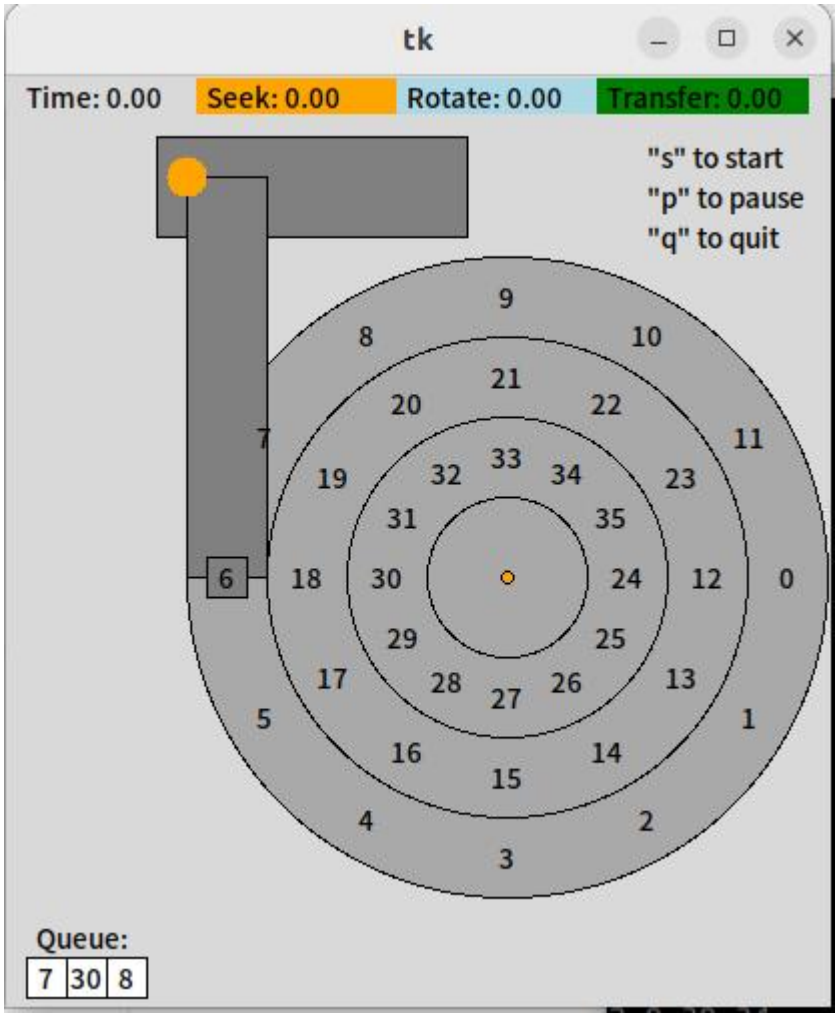
	-R 0.5	- R 0.1	-R 0.01
-a 0-	旋转时间增加	旋转时间增加	旋转时间增加
-a 6	旋转时间增加	旋转时间增加	旋转时间增加
-a 30	旋转时间增加	旋转时间增加	旋转时间增加
-a 7,30,8	只是旋转时间增加	少旋转一圈, 旋转时间减少	少旋转一圈，但旋转时间增多
-a 10,11,12,13	旋转时间增加	旋转时间增加	旋转时间增加

4. **你可能已经注意到,对于一些请求流,一些策略比 FIFO 更好。例如,对于请求流 -a 7,30,8 处理请求的顺序是什么?现在在相同的工作负载上运行最短寻道时间优先 (SSTF)调度程序(-p SSTF)。每个请求服务需要多长时间(寻道、旋转、传输)?**

对于请求流-a 7，30，8，如果按照 FIFO 的方式进行，处理顺序是 7，30，8，但是 SSTF 以及电梯算法 SCAN，最短定位时间优先都会是 7，8，30

由第一问得到的。使用 FIFO 算法的时间为 795

./disk.py -a 7,30,8 -p SSTF -G



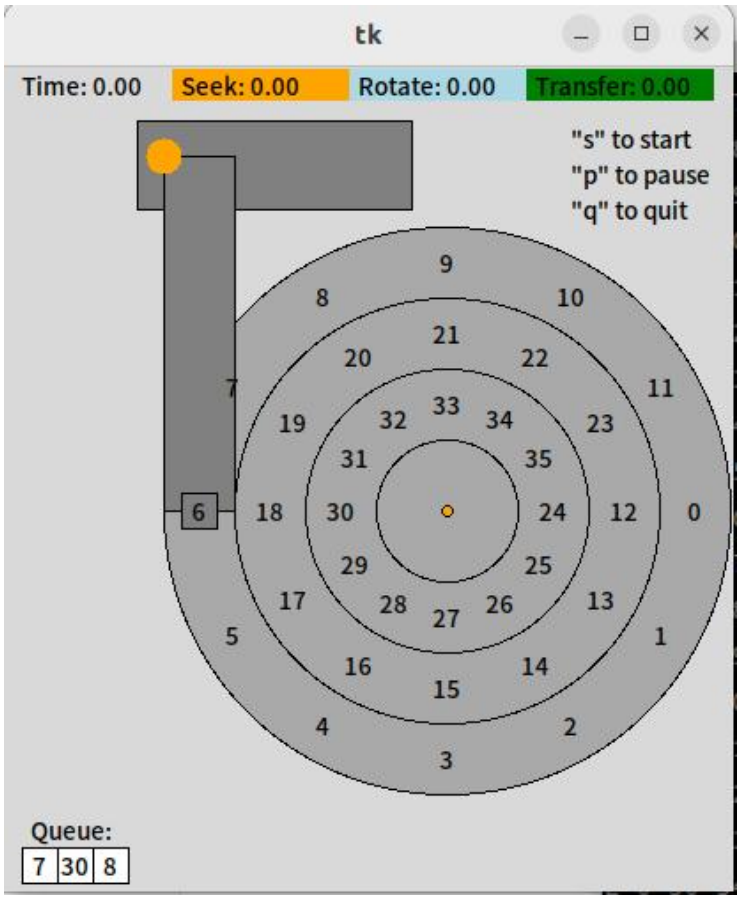

```
Block:   7  Seek:  0  Rotate: 15  Transfer: 30  Total:  45
Block:   8  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
Block:  30  Seek: 80  Rotate:190  Transfer: 30  Total: 300

TOTALS      Seek: 80  Rotate:205  Transfer: 90  Total: 375
```

7：
t 寻道=0
t 旋转=(7-6.5)*30=15
t 传输=30（7->8）
total=0+15+30=45
8：
t 寻道=0
t 旋转=0
t 传输=30（8->9）
total=0+0+30=30

30：
t 寻道=80（两次改变磁道）
t 旋转=（（12-9）+6）*30-80=270-80=190（30 对应的是 6，因为边寻道边旋转，所以需要减去寻道的时间）
t 传输=30（30->31）
total=80+190+30=300
总时间为 375，比之前的快了很多，这说明了合适的调度算法的重要性

5. 现在做同样的事情,但使用最短的访问时间优先(SATF)调度程序(-SATF)。 它是否对 -a 7,30,8 请求有影响? 找到 SATF 明显优于 SSTF 的一组请求。出现显著差异的条件是什么?

./disk.py -a 7,30,8 -p SATF -G



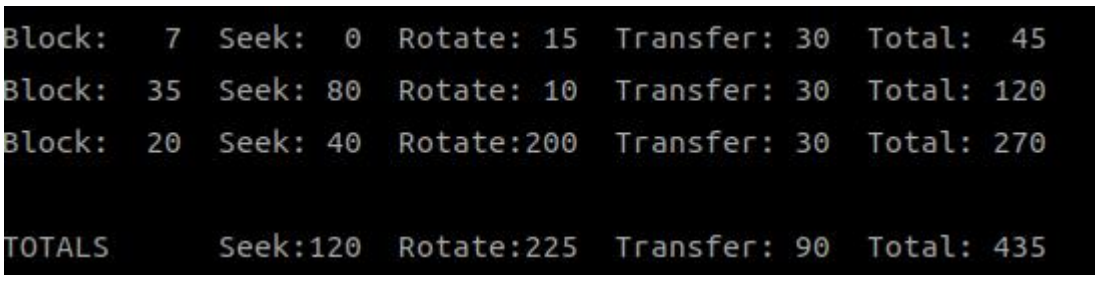

```
Block:   7  Seek:  0  Rotate: 15  Transfer: 30  Total:  45
Block:   8  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
Block:  30  Seek: 80  Rotate:190  Transfer: 30  Total: 300

TOTALS      Seek: 80  Rotate:205  Transfer: 90  Total: 375
```

7：
t 寻道=0
t 旋转=(7-6.5)*30=15
t 传输=30（7->8）
total=0+15+30=45
8：
t 寻道=0
t 旋转=0
t 传输=30（8->9）
total=0+0+30=30

30：
t 寻道=80（两次改变磁道）
t 旋转=（（12-9）+6）*30-80=270-80=190（30 对应的是 6，因为边寻道边旋转，所以需要减去寻道的时间）
t 传输=30（30->31）
total=80+190+30=300
总时间为 375

SATF 明显优于 SSTF：
也就是最短定位时间优先比最短寻道优先性能优异，比如构造 7，20，35 这个序列，
我们从 6.5 处开始，接着紧挨着执行 7，接下来如果按照寻道优先的话，会按照从外层到内层执行，这是会正好错过 20，需要多绕一圈，但是如果按照最短定位时间优先的话，会先进行 35，然后执行 20
./disk.py -a 7,20,35 -p SATF -G
./disk.py -a 7,20,35 -p SSTF -G

首先是 SATF



可以看到是按照 7->35->20 的顺序进行的

然后观察 SSTF

```
Block:   7  Seek:  0  Rotate: 15  Transfer: 30  Total:  45
Block:  20  Seek: 40  Rotate:320  Transfer: 30  Total: 390
Block:  35  Seek: 40  Rotate: 20  Transfer: 30  Total:  90

TOTALS      Seek: 80  Rotate:355  Transfer: 90  Total: 525
```

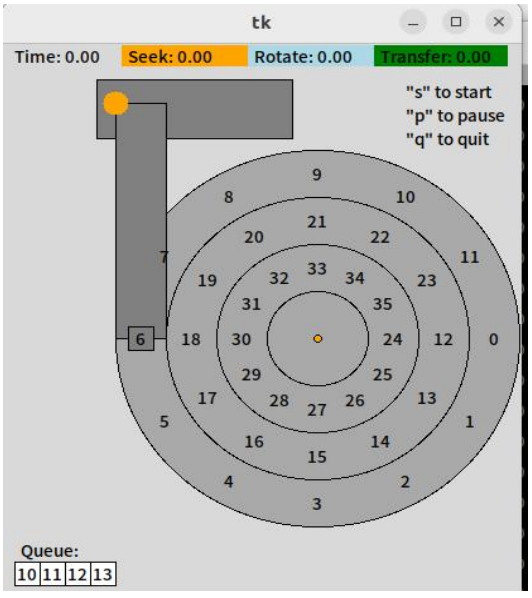
可以看到 SSAF 在这个序列上时间的优越性。

出现显著差异的条件：最短寻道时间优先会错过目标序列，导致多旋转，而最短定位时间优先的话，不会多旋转，这样会产生明显的差异

6. **你可能已经注意到,该磁盘没有特别好地处理请求流 -a 10,11,12,13。这是为什么？你可以引入一个磁道偏斜来解决这个问题(-o skew,其中 skew 是一个非负整数)?考虑到默认寻道速率,偏斜应该是多少,才能尽量减少这一组请求的总时间?对于不同的寻道速率(例如,-S 2,-S 4)呢?一般来说,考虑到寻道速率和扇区布局信息,你能否写出 一个公式来计算偏斜?**

没有处理好是因为 11->12，如果这几个块是连续的，直接进行传输就可以，但是由于不在一个磁道上，需要寻道，结果旋转了一圈才又到达对应的地方，但是如果磁道倾斜，使 11->12 时，寻道后直接进行传输就可以尽量避免时间的浪费

Block	T 寻道	T 旋转	T 传输	T 总
10	0	105	30	135
11	0	0	30	30
12	40	320	30	390
13	0	0	30	30



只需要让传输完 11，寻道之后再经过一点旋转就到达 12 处就好，这里磁道偏斜

只能取整数，1 不足以满足寻道时间的旋转，所以选择 2，此时寻道之后，只需要再旋转 20 度，就可以到达 12，进行传输，这样时间是最优的

```
Block:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Block:  11  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
Block:  12  Seek: 40  Rotate:320  Transfer: 30  Total: 390
Block:  13  Seek:  0  Rotate:  0  Transfer: 30  Total:  30

TOTALS      Seek: 40  Rotate:425  Transfer:120  Total: 585
```

我们采用指令./disk.py -a 10,11,12,13 -o 2 -c

```
Block:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
Block:  11  Seek:  0  Rotate:  0  Transfer: 30  Total:  30
Block:  12  Seek: 40  Rotate: 20  Transfer: 30  Total:  90
Block:  13  Seek:  0  Rotate:  0  Transfer: 30  Total:  30

TOTALS      Seek: 40  Rotate:125  Transfer:120  Total: 285
```

285 是我们的最短时间

按照以上的分析，磁道偏斜的角度应该大于寻道时间内旋转过的角度，以减少请求的总时间。假设寻道速率为 v，磁道间的距离为 s，旋转速率为 p，则寻道时间内旋转过的角度为 p*(s/v)。假设一个磁道有 n 个扇区，则一个扇区的角度为 360/n，偏斜磁道数设为 x，则有 p*(s/v) < (360/n)*x，x 应该取使不等式成立的最小值。

因此可以使用以下公式计算磁道偏斜：

$$x>pns/360v$$

x 向上取整

第 38 章

首先查看 readme 相关内容

- -s 种子 seed
- -D 磁盘个数，也就是书上的 N
- -C 大块大小，书上尝试了 2，但一般是 1
- -n 请求数量
- -S 请求大小
- -W 工作负载，选项为“rand”或者“seq”

- -w 写入占比，100 表示全写，0 表示全读
- -R 请求的范围
- -L RAID 的等级，提供 0，1，4，5
- -5 RAID5 的左对称（left- symmetric）和左不对称（left-asymmetric）布局，分别为“LS”或"LA"。
- -r 翻转标志
- -t 计算时间
- -c 查看答案

1. **使用模拟器执行一些基本的 RAID 映射测试。运行不同的级别（0、1、4、5），看看你是否可以找出一组请求的映射。 对于 RAID-5，看看你是否可以找出左对称（left- symmetric）和左不对称（left-asymmetric）布局之间的区别。 使用一些不同的随机种子，产生不同于上面的问题。**

1）RAID0 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

磁盘号=地址%磁盘数
偏移量=地址/磁盘数
执行指令：./raid.py -D 4 -n 5 -L 0 -R 16 -c
也就是四块磁盘，生成五组数据，使用 RAID0，范围是 0-15

```
13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 1, offset 3]

6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 2, offset 1]

8 1
LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 2]

12 1
LOGICAL READ from addr:12 size:4096
  read  [disk 0, offset 3]

7 1
LOGICAL READ from addr:7 size:4096
  read  [disk 3, offset 1]
```

13%4=1 13/4=3 6%4=2 6/4=1 8%4=0 8/4=2 12%4=0 12/4=3 7%4=3 7/4=2
符合规则
2）RAID1 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

磁盘号=2*地址%磁盘数 其副本磁盘号=2*地址%磁盘数 +1

偏移= 2*地址/磁盘数

执行指令：./raid.py -D 4 -n 5 -L 1 -R 8 -c

```
6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 1, offset 3]

3 1
LOGICAL READ from addr:3 size:4096
  read  [disk 3, offset 1]

4 1
LOGICAL READ from addr:4 size:4096
  read  [disk 0, offset 2]

6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 1, offset 3]

3 1
LOGICAL READ from addr:3 size:4096
  read  [disk 3, offset 1]
```

6*2%4=0 6*2/4=3 3*2%4=2 3*2/4=1 4*2%4=0 4*2/4=2

符合规则

3) RAID4 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

磁盘号=地址%（磁盘数-1）

偏移量=地址/（磁盘数-1）

执行指令：./raid.py -D 5 -n 5 -L 4 -R 16 -c

```
13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 1, offset 3]
6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 2, offset 1]
8 1
LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 2]
12 1
LOGICAL READ from addr:12 size:4096
  read  [disk 0, offset 3]
7 1
LOGICAL READ from addr:7 size:4096
  read  [disk 3, offset 1]
```

13%(5-1)=1 13/(5-1)=3 6%(5-1)=2 6/(5-1)=1 8%(5-1)=0 8/(5-1)=2
12%(5-1)=0 12/(5-1)=3 7%(5-1)=3 7/(5-1)=1
符合规则
4 RAID5 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

根据不同的布局直接找到对应地址
./raid.py -D 5 -n 20 -L 5 -R 20 -5 LS -W seq -c
./raid.py -D 5 -n 20 -L 5 -R 20 -5 LA -W seq -c
左对称布局：

Disk0	Disk1	Disk2	Disk3	DISK4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14

Disk0	Disk1	Disk2	Disk3	DISK4
P4	16	17	18	19

左不对称布局：

Disk0	Disk1	Disk2	Disk3	DISK4
0	1	2	3	P0
4	5	6	P1	7
8	9	P2	10	11
12	P3	13	14	15
P4	16	17	18	19

两种的区别在于数据块排列的格式，左对称分布中，数据块按照顺序分布在不同磁盘中，下一个数据块放在下一个磁盘上，直到最后一个磁盘。左不对称分布中，如果下一个块磁盘存放了校验块，就跳过下一个磁盘。

2. 与第一个问题一样，但这次使用 -C 来改变块的大小。大块的大小如何改变映射?

1）RAID0：./raid.py -D 4 -n 16 -L 0 -R 16 -W seq -C 8192 -c

四块磁盘，请求数量是 16，使用 RAID0，范围是 0-15，采用顺序，使用块 8192（也就是 2*4k）

./raid.py -D 4 -n 16 -L 0 -R 16 -W seq -C 8192 -c

Disk0	Disk1	Disk2	Disk3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

相当于读取时直接读取 0 和 1 块，接着读取 2 和 3

2）RAID1：./raid.py -D 4 -n 8 -L 1 -R 8 -W seq -C 8192 -c

Disk0	Disk1	Disk2	Disk3
0	0	2	2
1	1	3	3

Disk0	Disk1	Disk2	Disk3
4	4	6	6
5	5	7	7

相当于先读取 0 和 1，下一个磁盘备份，然后读取 2 和 3，下一个磁盘备份，接着的内容按上面的进行。

3）RAID4：./raid.py -D 5 -n 24 -L 4 -R 24 -W seq -C 8192 -c

Disk0	Disk1	Disk2	Disk3	DISK4
0	2	4	6	P0
1	3	5	7	P1
8	10	12	14	P2
9	11	13	15	P3
16	18	20	22	P4
17	19	21	23	P5

可以看到依旧是先在一个磁盘中放置两块内容

4）RAID5：./raid.py -D 5 -n 24 -L 5 -R 24 -W seq -C 8192 -5 LA -c

Disk0	Disk1	Disk2	Disk3	DISK4
0	2	4	6	P0
1	3	5	7	P1
8	10	12	P2	14
9	11	13	P3	15
16	18	P4	20	22
17	19	P5	21	23

可以发现大块大小没有改变各级的布局规则，相当于把多块内容看成一个整体，映射情况并没有发生变化。

3. 执行上述测试，但使用 r 标志来反转每个问题的性质。

-r 反转后，问题为给出磁盘号和磁盘偏移，计算地址。

已知映射关系以及 RAID 布局的情况下，只需要从布局中找到 RAID 中某个磁盘特定位置保存的地址，或使用映射公式计算。

RAID0：地址 = 磁盘数*偏移 + 磁盘号，据此就可以算出 1 中的地址。

RAID1：地址 = (磁盘数*偏移 + 磁盘号)/2

RAID4：地址 = (磁盘数-1)*偏移 + 磁盘号 (可能有 1 的偏差，因为不确定条带前面是否已有校验块)

RAID5：可以根据不同的布局直接找到地址。

1）RAID0 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

磁盘号=地址%磁盘数
偏移量=地址/磁盘数
执行指令：./raid.py -D 4 -n 5 -L 0 -R 16 -r -c

```
13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 1, offset 3]

6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 2, offset 1]

8 1
LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 2]

12 1
LOGICAL READ from addr:12 size:4096
  read  [disk 0, offset 3]

7 1
LOGICAL READ from addr:7 size:4096
  read  [disk 3, offset 1]
```

符合规则
2）RAID1 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

磁盘号=2*地址%磁盘数 其副本磁盘号=2*地址%磁盘数 +1

偏移= 2*地址/磁盘数

执行指令：./raid.py -D 4 -n 5 -L 1 -R 8 -r -c

```
6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 1, offset 3]

3 1
LOGICAL READ from addr:3 size:4096
  read  [disk 3, offset 1]

4 1
LOGICAL READ from addr:4 size:4096
  read  [disk 0, offset 2]

6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 1, offset 3]

3 1
LOGICAL READ from addr:3 size:4096
  read  [disk 3, offset 1]
```

符合规则

3) RAID4 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

磁盘号=地址%（磁盘数-1）

偏移量=地址/（磁盘数-1）

执行指令：./raid.py -D 5 -n 5 -L 4 -R 16 -r -c

```
13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 1, offset 3]
6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 2, offset 1]
8 1
LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 2]
12 1
LOGICAL READ from addr:12 size:4096
  read  [disk 0, offset 3]
7 1
LOGICAL READ from addr:7 size:4096
  read  [disk 3, offset 1]
```

符合规则

4 RAID5 映射关系：

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

根据不同的布局直接找到对应地址

./raid.py -D 5 -n 20 -L 5 -R 20 -5 LS -W seq -r -c

./raid.py -D 5 -n 20 -L 5 -R 20 -5 LA -W seq -r -c

左对称布局：

Disk0	Disk1	Disk2	Disk3	DISK4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

左不对称布局：

Disk0	Disk1	Disk2	Disk3	DISK4
0	1	2	3	P0
4	5	6	P1	7
8	9	P2	10	11
12	P3	13	14	15
P4	16	17	18	19

符合规则

4.现在使用反转标志，但用-S 标志增加每个请求的大小。尝试指定 8 KB、12 KB 和 16 KB 的大小，同时改变 RAID 级别。 当请求的大小增加时，底层 IO 模式会发生什么？请务必在顺序工作负载上尝试此操作（-W sequential）。 对于什么请求大小，RAID-4 和 RAID-5 的 I0 效率更高?

1）-S 8K

RAID0:

./raid.py -n 5 -L 0 -R 20 -r -S 8K -W seq

./raid.py -n 5 -L 0 -R 20 -r -S 8K -W seq -w 100

```
0 2
LOGICAL READ from addr:0 size:8192
  read  [disk 0, offset 0]
  read  [disk 1, offset 0]

2 2
LOGICAL READ from addr:2 size:8192
  read  [disk 2, offset 0]
  read  [disk 3, offset 0]

4 2
LOGICAL READ from addr:4 size:8192
  read  [disk 0, offset 1]
  read  [disk 1, offset 1]

6 2
LOGICAL READ from addr:6 size:8192
  read  [disk 2, offset 1]
  read  [disk 3, offset 1]

8 2
LOGICAL READ from addr:8 size:8192
  read  [disk 0, offset 2]
  read  [disk 1, offset 2]
```

```
0 2
LOGICAL WRITE to  addr:0 size:8192
  write [disk 0, offset 0]
  write [disk 1, offset 0]

2 2
LOGICAL WRITE to  addr:2 size:8192
  write [disk 2, offset 0]
  write [disk 3, offset 0]

4 2
LOGICAL WRITE to  addr:4 size:8192
  write [disk 0, offset 1]
  write [disk 1, offset 1]

6 2
LOGICAL WRITE to  addr:6 size:8192
  write [disk 2, offset 1]
  write [disk 3, offset 1]

8 2
LOGICAL WRITE to  addr:8 size:8192
  write [disk 0, offset 2]
  write [disk 1, offset 2]
```

读和写都需要两个操作完成

RAID1:

./raid.py -n 5 -L 1 -R 20 -r -S 8K -W seq
./raid.py -n 5 -L 1 -R 20 -r -S 8K -W seq -w 100

```
0 2
LOGICAL READ from addr:0 size:8192
  read  [disk 0, offset 0]
  read  [disk 2, offset 0]

2 2
LOGICAL READ from addr:2 size:8192
  read  [disk 1, offset 1]
  read  [disk 3, offset 1]

4 2
LOGICAL READ from addr:4 size:8192
  read  [disk 0, offset 2]
  read  [disk 2, offset 2]

6 2
LOGICAL READ from addr:6 size:8192
  read  [disk 1, offset 3]
  read  [disk 3, offset 3]

8 2
LOGICAL READ from addr:8 size:8192
  read  [disk 0, offset 4]
  read  [disk 2, offset 4]
```

```
0 2
LOGICAL WRITE to  addr:0 size:8192
  write [disk 0, offset 0]    write [disk 1, offset 0]
  write [disk 2, offset 0]    write [disk 3, offset 0]

2 2
LOGICAL WRITE to  addr:2 size:8192
  write [disk 0, offset 1]    write [disk 1, offset 1]
  write [disk 2, offset 1]    write [disk 3, offset 1]

4 2
LOGICAL WRITE to  addr:4 size:8192
  write [disk 0, offset 2]    write [disk 1, offset 2]
  write [disk 2, offset 2]    write [disk 3, offset 2]

6 2
LOGICAL WRITE to  addr:6 size:8192
  write [disk 0, offset 3]    write [disk 1, offset 3]
  write [disk 2, offset 3]    write [disk 3, offset 3]

8 2
LOGICAL WRITE to  addr:8 size:8192
  write [disk 0, offset 4]    write [disk 1, offset 4]
  write [disk 2, offset 4]    write [disk 3, offset 4]
```

考虑到镜像备份的存在，读操作进行两个操作，写操作进行四个操作
RAID4:
./raid.py -n 5 -L 4 -R 20 -r -S 8K -W seq
./raid.py -n 5 -L 4 -R 20 -r -S 8K -W seq -w 100

```
0 2
LOGICAL READ from addr:0 size:8192
  read  [disk 0, offset 0]    read  [disk 1, offset 0]
2 2
LOGICAL READ from addr:2 size:8192
  read  [disk 2, offset 0]    read  [disk 0, offset 1]
4 2
LOGICAL READ from addr:4 size:8192
  read  [disk 1, offset 1]    read  [disk 2, offset 1]
6 2
LOGICAL READ from addr:6 size:8192
  read  [disk 0, offset 2]    read  [disk 1, offset 2]
8 2
LOGICAL READ from addr:8 size:8192
  read  [disk 2, offset 2]    read  [disk 0, offset 3]
```

```
0 2
LOGICAL WRITE to  addr:0 size:8192
  read  [disk 2, offset 0]
  write [disk 0, offset 0]    write [disk 1, offset 0]    write [disk 3, offset 0]

2 2
LOGICAL WRITE to  addr:2 size:8192
  read  [disk 2, offset 0]    read  [disk 3, offset 0]
  write [disk 2, offset 0]    write [disk 3, offset 0]
  read  [disk 0, offset 1]    read  [disk 3, offset 1]
  write [disk 0, offset 1]    write [disk 3, offset 1]

4 2
LOGICAL WRITE to  addr:4 size:8192
  read  [disk 0, offset 1]
  write [disk 1, offset 1]    write [disk 2, offset 1]    write [disk 3, offset 1]

6 2
LOGICAL WRITE to  addr:6 size:8192
  read  [disk 2, offset 2]
  write [disk 0, offset 2]    write [disk 1, offset 2]    write [disk 3, offset 2]

8 2
LOGICAL WRITE to  addr:8 size:8192
  read  [disk 2, offset 2]    read  [disk 3, offset 2]
  write [disk 2, offset 2]    write [disk 3, offset 2]
  read  [disk 0, offset 3]    read  [disk 3, offset 3]
  write [disk 0, offset 3]    write [disk 3, offset 3]
```

读取需要两个操作

按照减法奇偶校验写入一个块，应该要先读地址对应的块和奇偶校验块，判断是否需要改变，然后再写入地址对应的块和校验块。请求大小为 8k，其实是写入地址和地址+1 的块。

若两个块不连续，则需要两次上述的操作，需要 8 个 I/O 操作；若这两个块是连续的（偏移量一样），此时由于这两个块共用同一个校验位，使用加法奇偶校验，只需要写入这两个块和校验位 P，由原来的 8 个 I/O 操作减少到 4 个。

RAID5:
./raid.py -n 5 -L 5 -R 20 -r -S 8K -W seq
./raid.py -n 5 -L 5 -R 20 -r -S 8K -W seq -w 100

```
0 2
LOGICAL READ from addr:0 size:8192
  read  [disk 0, offset 0]    read  [disk 1, offset 0]
2 2
LOGICAL READ from addr:2 size:8192
  read  [disk 2, offset 0]    read  [disk 3, offset 1]
4 2
LOGICAL READ from addr:4 size:8192
  read  [disk 0, offset 1]    read  [disk 1, offset 1]
6 2
LOGICAL READ from addr:6 size:8192
  read  [disk 2, offset 2]    read  [disk 3, offset 2]
8 2
LOGICAL READ from addr:8 size:8192
  read  [disk 0, offset 2]    read  [disk 1, offset 3]
```

```
0 2
LOGICAL WRITE to  addr:0 size:8192
  read  [disk 2, offset 0]
  write [disk 0, offset 0]    write [disk 1, offset 0]    write [disk 3, offset 0]

2 2
LOGICAL WRITE to  addr:2 size:8192
  read  [disk 2, offset 0]    read  [disk 3, offset 0]
  write [disk 2, offset 0]    write [disk 3, offset 0]
  read  [disk 3, offset 1]    read  [disk 2, offset 1]
  write [disk 3, offset 1]    write [disk 2, offset 1]

4 2
LOGICAL WRITE to  addr:4 size:8192
  read  [disk 3, offset 1]
  write [disk 0, offset 1]    write [disk 1, offset 1]    write [disk 2, offset 1]

6 2
LOGICAL WRITE to  addr:6 size:8192
  read  [disk 0, offset 2]
  write [disk 2, offset 2]    write [disk 3, offset 2]    write [disk 1, offset 2]

8 2
LOGICAL WRITE to  addr:8 size:8192
  read  [disk 0, offset 2]    read  [disk 1, offset 2]
  write [disk 0, offset 2]    write [disk 1, offset 2]
  read  [disk 1, offset 3]    read  [disk 0, offset 3]
  write [disk 1, offset 3]    write [disk 0, offset 3]
```

RAID5 与 RAID4 类似，不再分析

2）-S 12K
对于 RAID0 与 RAID1，只是需要多对一个块进行处理。因此与 8K 类似，RAID0

读写均需要 3 次 I/O 完成请求，RAID1 需要 3 次读操作完成读请求，6 次写操作完成写请求。
对于 RAID4 顺序读取需要 3 次完成，但是写入，也是分为在一个条带和不在一个条带。对于在同一个条带上写，只需要将三个块异或得到校验位，然后将四个块全部写入就可以，所以需要 4 次写操作；对于不在同一个条带上写，肯定会出现两个块在同一条带上，这两个块需要一次读，三次写，也就是 4 次 IO 操作，另一个需要读读写写，需要四次 IO 操作，所以一共需要 8 次 IO 操作。
RAID5 和 RAID4 类似
3）-S 16K
对于 RAID0 与 RAID1，再多对一个块进行处理。因此与 8K 类似，RAID0 读写均需要 4 次 I/O 完成请求，RAID1 需要 4 次读操作完成读请求，8 次写操作完成写请求。
对于 RAID4，顺序读需要四次读操作，对于写有两种情况：
（1）两个条带块数分别是 3，1
3 个块的条带需要 4 次写操作，一个块的条带需要读读写写，四次 IO 操作，一共 8 次。
（2）两个条带块数分别是 2，2
每个条带都需要一次读，三次写，所以一共 8 次 IO 操作。
对于 RAID5，与 RAID4 类似

总结：对于 4 个磁盘的情况下，请求块数越接近一个条带的块数，RAID4 和 RAID5 的写性能更好。即 RAID4/5 更适合接近一个条带块数的顺序写入，也就是全写入。在这种情况下，加法奇偶校验可以比减法奇偶校验使用更少的写操作完成请求，最好的情况下，可以使用全条带写入直接完成写入，而不需要读取数据块。

5. 使用模拟器的定时模式（-t）来估计 100 次随机读取到 RAID 的性能，同时改变 RAID 级别，使用 4 个磁盘。

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	$N/2$	$N-1$	$N-1$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$

1）RAID0：./raid.py -L 0 -t -n 100 -c

```
disk:0  busy: 100.00  I/Os:    28 (sequential:0 nearly:1 random:27)
disk:1  busy:  93.91  I/Os:    29 (sequential:0 nearly:6 random:23)
disk:2  busy:  87.92  I/Os:    24 (sequential:0 nearly:0 random:24)
disk:3  busy:  65.94  I/Os:    19 (sequential:0 nearly:1 random:18)

STAT totalTime 275.69999999999993
```

2）RAID1：./raid.py -L 1 -t -n 100 -c

```
disk:0  busy: 100.00  I/Os:    28 (sequential:0 nearly:1 random:27)
disk:1  busy:  86.98  I/Os:    24 (sequential:0 nearly:0 random:24)
disk:2  busy:  97.52  I/Os:    29 (sequential:0 nearly:3 random:26)
disk:3  busy:  65.23  I/Os:    19 (sequential:0 nearly:1 random:18)

STAT totalTime 278.7
```

3）RAID4：./raid.py -L 4 -t -n 100 -c

```
disk:0  busy:  78.48  I/Os:    30 (sequential:0 nearly:0 random:30)
disk:1  busy: 100.00  I/Os:    40 (sequential:0 nearly:3 random:37)
disk:2  busy:  76.46  I/Os:    30 (sequential:0 nearly:2 random:28)
disk:3  busy:   0.00  I/Os:     0 (sequential:0 nearly:0 random:0)

STAT totalTime 386.1000000000002
```

4）RAID5：./raid.py -L 5 -t -n 100 -c

```
disk:0  busy: 100.00  I/Os:    28 (sequential:0 nearly:1 random:27)
disk:1  busy:  95.84  I/Os:    29 (sequential:0 nearly:5 random:24)
disk:2  busy:  87.60  I/Os:    24 (sequential:0 nearly:0 random:24)
disk:3  busy:  65.70  I/Os:    19 (sequential:0 nearly:1 random:18)

STAT totalTime 276.7
```

基本上与课本上的内容相吻合

第 40 章

首先阅读 readme

- -s 随机种子 seed
- -i inode 数量
- -d 数据块数量
- -n 请求数量
- -r 打印设置
- -p 打印最终文件系统架构
- -c 计算答案

1. **用一些不同的随机种子（比如 17、18、19、20）运行模拟器，看看你是否能确定每次状态变化之间一定发生了哪些操作。**

不同操作的作用与对磁盘的影响：

- mkdir() - creates a new directory：修改 inode 位图，增加一个 inode 用来存放新目录元数据，向存放新目录的目录块中增加一个条目，修改 data 位图，增加一个数据块用于存放新目录的内容，更新相应 inode 中的引用计数
- creat() - creates a new (empty) file：修改 inode 位图，增加一个 inode 用来存放新文件元数据，向存放新文件的目录块中增加一个条目，更新相应 inode 中的引用计数
- open(), write(), close() - appends a block to a file: 修改 data 位图，增加一个数据块用于存放文件的新内容，修改 inode 中的数据块地址字段
- link() - creates a hard link to a file: 修改 inode，增加其中的引用计数，在保存链接的目录块中增加一个条目
- unlink() - unlinks a file (removing it if linkcnt==0): 修改 inode，减小其中的引用计数，在保存链接的目录块中删除一个条目，当引用计数减为 0 时，删除文件，释放 inode、数据块，修改 inode 位图、data 位图

每个 inode 都有三个字段：第一个字段表示文件类型（例如，f 表示常规文件，d 表示目录）;第二个指示哪个数据块属于一个文件（这里，文件只能是空的，数据块的地址设置为 -1，或者一个块的大小，它将有一个非负的地址）;第三个显示文件或目录的引用计数。例如，以下 inode 是一个常规文件，该文件为空（地址字段设置为 -1），并且在文件系统中只有一个链接：[f a:-1 r:1]

如果同一个文件分配了一个块（比如块 10），它将显示如下：[f a:10 r:1]

如果有人随后创建了指向此 inode 的硬链接，那么它将变为：　[f a:10 r:2]

最后，数据块可以保留用户数据或目录数据。如果填充目录数据，则块中的每个条目都采用（name， inumber） 的形式，其中 “name” 是文件或目录的名称，“inumber” 是文件的 inode 编号。因此，假设根 inode 为 0，则空的根目录如下所示：　[(.,0) (..,0)]

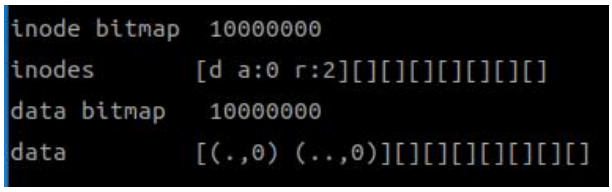
如果我们将单个文件“f”添加到根目录中，该文件已分配了 inode 编号 1，则根目录内容将变为：　[(.,0) (..,0) (f,1)]
如果数据块包含用户数据，则该数据块仅显示为块中的单个字符，例如“h”。如果它为空且未分配，则仅显示一对空括号 （[]）。
因此，整个文件系统描述如下：

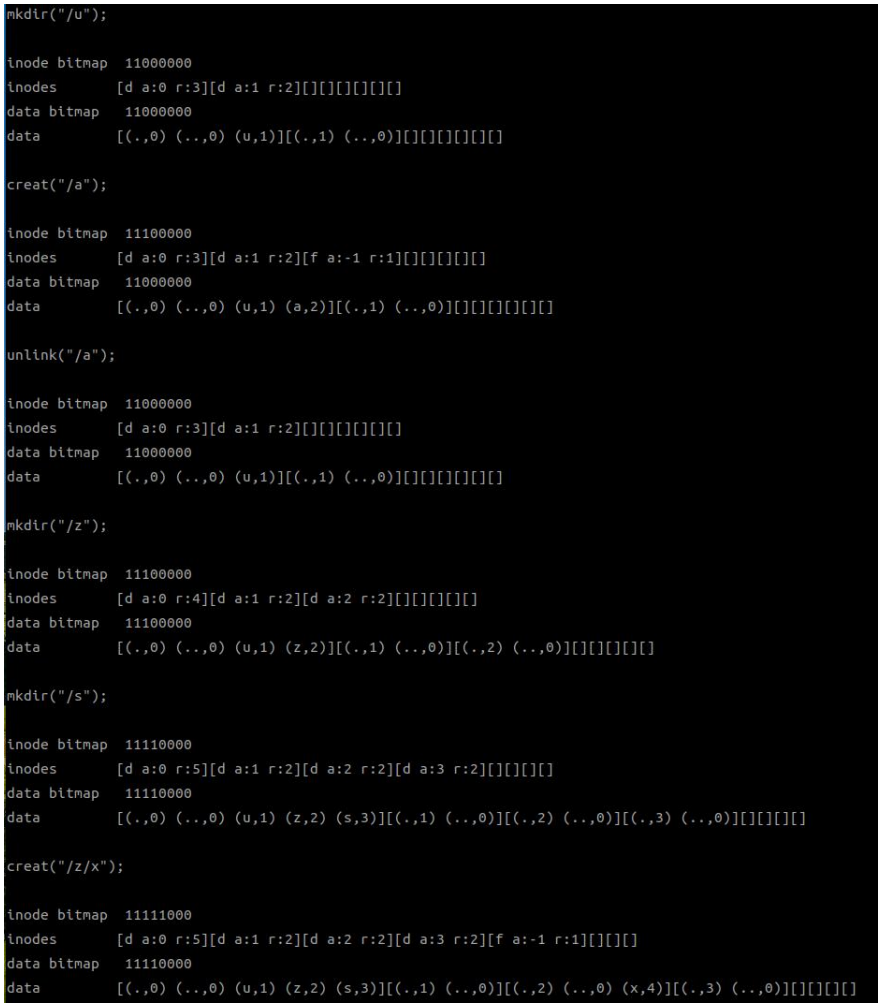
```
inode bitmap 11110000
inodes       [d a:0 r:6] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
data bitmap  11100000
data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] ...
```

此文件系统有 8 个 inode 和 8 个数据块。根目录包含三个条目（“.”和“..”除外），分别为“y”、“z”和“f”。通过查找 inode 1，我们可以看到“y”是一个常规文件（类型 f），分配给它的一个数据块（地址 1）。在该数据块 1 中是文件“y”的内容：即“u”。我们还可以看到，“z”是一个空的常规文件（地址字段设置为-1），而“f”（索引节点号 3）是一个目录，也是空的。您还可以从位图中看到，前四个 inode 位图条目以及前三个数据位图条目被标记为已分配。

这个文件系统有八个 inode 和八个数据块。根目录包含三个条目（“.”和“..”除外），分别为“y”、“z”和“f”。 通过查找 inode 1，我们可以看到“y”是一个常规文件（类型 f），分配给它的单个数据块（地址 1）。 在那个数据块 1 中是文件“y”的内容：即“u”。 我们还可以看到“z”是一个空的常规文件（地址字段设置为-1），而“f”（inode 编号 3）是一个目录，也是空的。 您还可以从位图中看到前四个 inode 位图条目被标记为已分配，以及前 三个数据位图条目。

1）seed 17：./vsfs.py -n 6 -s 17 -c





操作 1 同时修改了 inode 位图和 data 位图，所以是 mkdir()。查看 1 号 inode,发现新建了一个目录，其数据存放在 1 号数据块，在 0 号数据块中查看新增加的条目，指示新建的目录名为“u”，所以操作 1 是 mkdir(“/u”)

操作 2 只修改了 inode 位图，所以是 creat()。查看 2 号 inode,发现新建了一个文件，在 0 号数据块中查看新增加的条目，指示新建的文件名为“a”，所以操作 2 是 creat(“/a”)

操作 3 修改了 inode 位图，删除了 inode 和目录块中的条目，所以是 unlink()。发现删除的是 2 号 inode，所以操作 3 是 unlink(“/a”)

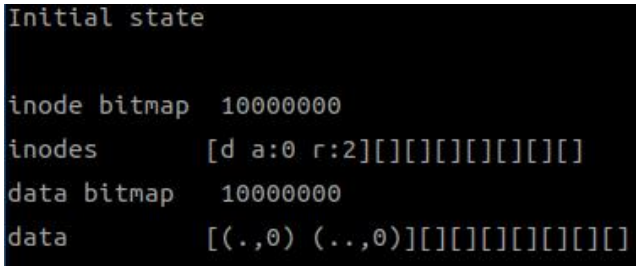
操作 4 同时修改了 inode 位图和 data 位图，所以是 mkdir()。查看 2 号 inode,发现新建了一个目录，其数据存放在 2 号数据块，在 0 号数据块中查看新增加的条目，指示新建的目录

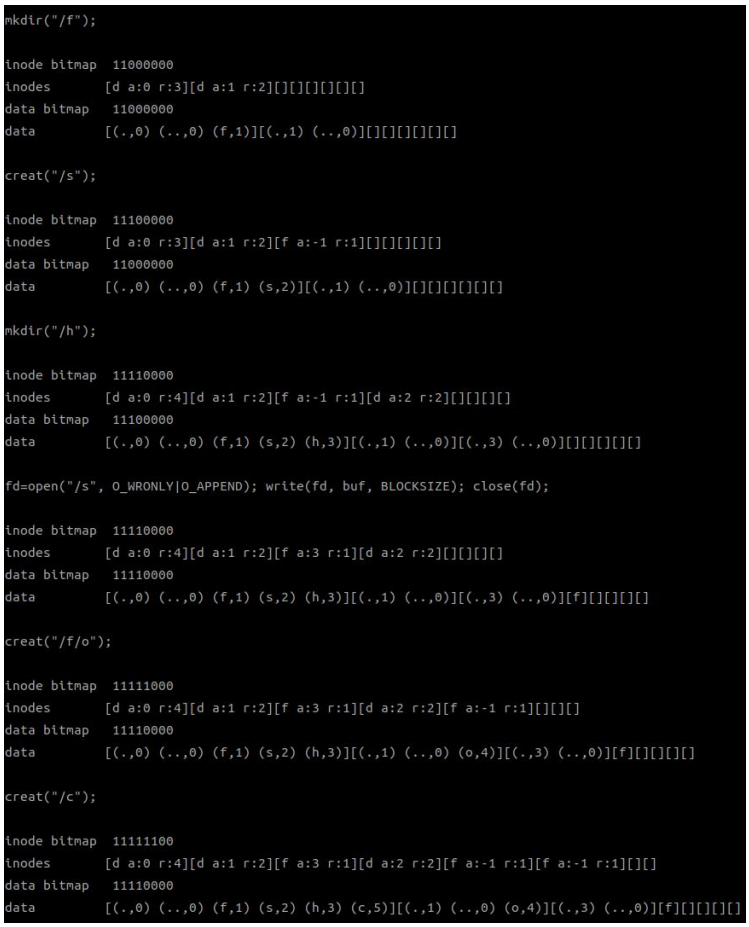
名为“z”，所以操作 4 是 mkdir(“/z”)

操作 5 同时修改了 inode 位图和 data 位图，所以是 mkdir()。查看 3 号 inode,发现新建了一个目录，其数据存放在 3 号数据块，在 0 号数据块中查看新增加的条目，指示新建的目录名为“s”，所以操作 5 是 mkdir(“/s”)

操作 6 只修改了 inode 位图，所以是 creat。查看 4 号 inode,发现新建了一个文件，在 3 号数据块（目录 z 的目录块）中查看新增加的条目，指示新建的文件名为“x”，所以操作 6 是 creat(“/z/x”)

2）seed 18：./vsfs.py -n 6 -s 18 -c





操作 1 同时修改了 inode 位图和数据位图，只有 mkdir 可以做到.查看 1 号 inode,发现新建了一个目录，其数据存放在 1 号数据块，在 0 号数据块中查看新增加的条目,可以看到 data 中多了条目(f,1)，所以新增目录为/f,所以为 mkdir(“/f”)

操作 2 只修改了 inode 位图，所以应该为 creat，查看 2 号 inode，发现增加了一个文件，在数据块 0 中可以看到新增了一个 s 文件，所以操作为 creat(“/s”)

操作 3 同时修改了 inode 位图和数据位图，只有 mkdir 可以做到，多了一个 3 号 inode，

发现新建了一个目录，查看数据块 0 可以看到多了一个（h,3），所以操作为 mkdir(“/h”)

操作 4 只改变了数据位图，修改了 2 号 inode（文件 s）中的地址字段，增加了 3 号数据块，所以操作 4 是 fd=open(“/s”,O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd)

操作 5 修改了 inode 位图，发现多了 4 号 inode，看到 1 号 data 块中多了（o，4），也就是增加了 o 文件，对应的是/f 目录，所以操作为 creat(“/f/o”)

操作 6 只修改了 inode 位图，发现多了 5 号 inode，看到 0 号数据块中多了（c,5）,也就是增加了 c 文件，所以操作为 creat(“/c”)

3）seed 19：./vsfs.py -n 6 -s 19 -c

```
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2][][][][][][][]
data bitmap   10000000
data          [(.,0) (..,0)][][][][][][][]
```

```
creat("/k");

inode bitmap  11000000
inodes        [d a:0 r:2][f a:-1 r:1][][][][][]
data bitmap   10000000
data          [(.,0) (..,0) (k,1)][][][][][][]

creat("/g");

inode bitmap  11100000
inodes        [d a:0 r:2][f a:-1 r:1][f a:-1 r:1][][][][]
data bitmap   10000000
data          [(.,0) (..,0) (k,1) (g,2)][][][][][][]

fd=open("/k", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap  11100000
inodes        [d a:0 r:2][f a:1 r:1][f a:-1 r:1][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (k,1) (g,2)][g][][][][][]

link("/k", "/b");

inode bitmap  11100000
inodes        [d a:0 r:2][f a:1 r:2][f a:-1 r:1][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (k,1) (g,2) (b,1)][g][][][][][]

link("/b", "/t");

inode bitmap  11100000
inodes        [d a:0 r:2][f a:1 r:3][f a:-1 r:1][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (k,1) (g,2) (b,1) (t,1)][g][][][][][]

unlink("/k");

inode bitmap  11100000
inodes        [d a:0 r:2][f a:1 r:2][f a:-1 r:1][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (g,2) (b,1) (t,1)][g][][][][][]
```

操作 1：inode 位图发现变化，所以应该是 creat，可以看到增加了 1 号 inode，0 号数据块中增加了(k,1)，所以操作是 creat(“/k”)

操作 2：inode 位图发现变化，所以应该是 creat,可以看到增加了 2 号 inode，0 号数据块增加了（g,2）,所以操作是 creat(“/g”)

操作 3：修改了 data 位图，修改了 1 号 inode（文件 k）中的地址字段，增加了 1 号数据块，所以操作 3 是 fd=open("/k",O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd)

操作 4：1 号 inode+1, 数据块 0 多了(b,1),而且文件标识也为 1, 所以操作是 link("/k", "/b")

操作 5：1 号 inode+1, 数据块 0 多了(t,1),而且文件标识也为 1, 所以操作是 link("/b", "/t")

操作 6：1 号 inode-1，数据块 0 少了(k,1),所以操作是 unlink("/k")

4）seed 20：./vsfs.py -n 6 -s 20 -c

```
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2][][][][][][][]
data bitmap   10000000
data          [(.,0) (..,0)][][][][][][][]
```

```
creat("/x");

inode bitmap  11000000
inodes        [d a:0 r:2][f a:-1 r:1][][][][][][]
data bitmap   10000000
data          [(.,0) (..,0) (x,1)][][][][][][][]

fd=open("/x", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap  11000000
inodes        [d a:0 r:2][f a:1 r:1][][][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (x,1)][x][][][][][][]

creat("/k");

inode bitmap  11100000
inodes        [d a:0 r:2][f a:1 r:1][f a:-1 r:1][][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (x,1) (k,2)][x][][][][][][]

creat("/y");

inode bitmap  11110000
inodes        [d a:0 r:2][f a:1 r:1][f a:-1 r:1][f a:-1 r:1][][][][]
data bitmap   11000000
data          [(.,0) (..,0) (x,1) (k,2) (y,3)][x][][][][][][]

unlink("/x");

inode bitmap  10110000
inodes        [d a:0 r:2][][f a:-1 r:1][f a:-1 r:1][][][][]
data bitmap   10000000
data          [(.,0) (..,0) (k,2) (y,3)][][][][][][][]

unlink("/y");

inode bitmap  10100000
inodes        [d a:0 r:2][][f a:-1 r:1][][][][][]
data bitmap   10000000
data          [(.,0) (..,0) (k,2)][][][][][][][]
```

操作 1 只修改了 inode 位图，所以是 creat()。查看 1 号 inode,发现新建了一个文件，在 0 号数据块中查看新增加的条目，指示新建的文件名为"x"，所以操作 1 是 creat("/x")

操作 2 修改了 data 位图，修改了 1 号 inode（文件 x）中的地址字段，增加了 1 号数据块，所以操作 2 是 fd=open("/x",O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd)

操作 3 只修改了 inode 位图，所以是 creat()。查看 2 号 inode,发现新建了一个文件，在 0 号数据块中查看新增加的条目，指示新建的文件名为"k"，所以操作 3 是 creat("/k")

操作 4 只修改了 inode 位图，所以是 creat()。查看 3 号 inode,发现新建了一个文件，在 0 号数据块中查看新增加的条目，指示新建的文件名为"y"，所以操作 4 是 creat("/y")

操作 5 修改了 inode 位图和 data 位图，删除了 inode、数据块和目录块中的条目，所以是 unlink()。发现删除的是 1 号 inode（文件 x），所以操作 5 是 unlink("/x")

操作 6 修改了 inode 位图，删除了 inode 和目录块中的条目，所以是 unlink()。发现删除的是 3 号 inode（文件 y），所以操作 6 是 unlink("/y")

2. 现在使用不同的随机种子（比如 21、22、23、24），但使用 -r 标志运行，这样做可以让你在显示操作时猜测状态的变化。关于 inode 和数据块分配算法，根据它们喜欢分配的块，你可以得出什么结论?

1） Seed21：./vsfs.py -n 6 -s 21 -r

Initial state

```
inode bitmap    10000000
inodes          [d a:0 r:2][][][][][][][]
data bitmap     10000000
data            [(.,0) (..,0)][][][][][][][]
```

```
mkdir("/o");
```

```
inode bitmap    11000000
inodes          [d a:0 r:3][d a:1 r:2][][][][][][]
data bitmap     11000000
data            [(.,0) (..,0) (o,1)][(.,1) (..,0)][][][][][][]
```

```
creat("/b");
```

```
inode bitmap    11100000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][][][][][]
data bitmap     11000000
data            [(.,0) (..,0) (o,1) (b,2)][(.,1) (..,0)][][][][][][]
```

```
creat("/o/q");
```

```
inode bitmap    11110000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:1][][][][]
data bitmap     11000000
data            [(.,0) (..,0) (o,1) (b,2)][(.,1) (..,0) (q,3)][][][][][][]
```

```
fd=open("/b", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);
```

inode bitmap 11110000
inodes [d a:0 r:3][d a:1 r:2][f a:2 r:1][f a:-1 r:1][][][][]
data bitmap 11100000
data [(.,0) (..,0) (o,1) (b,2)][(.,1) (..,0) (q,3)][m][][][][][]

fd=open("/o/q", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap 11110000
inodes [d a:0 r:3][d a:1 r:2][f a:2 r:1][f a:3 r:1][][][][]
data bitmap 11110000
data [(.,0) (..,0) (o,1) (b,2)][(.,1) (..,0) (q,3)][m][j][][][][]

creat("/o/j");

inode bitmap 11111000
inodes [d a:0 r:3][d a:1 r:2][f a:2 r:1][f a:3 r:1][f a:-1 r:1][][][]
data bitmap 11110000
data [(.,0) (..,0) (o,1) (b,2)][(.,1) (..,0) (q,3) (j,4)][m][j][][][][]

2） Seed22：./vsfs.py -n 6 -s 22 -r
Initial state

inode bitmap 10000000
inodes [d a:0 r:2][][][][][][][][]
data bitmap 10000000
data [(.,0) (..,0)][][][][][][][][]

creat("/z");

inode bitmap 11000000
inodes [d a:0 r:2][f a:-1 r:1][][][][][][][]
data bitmap 10000000
data [(.,0) (..,0) (z,1)][][][][][][][][]

fd=open("/z", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap 11000000
inodes [d a:0 r:2][f a:1 r:1][][][][][][][]
data bitmap 11000000
data [(.,0) (..,0) (z,1)][q][][][][][][][]

unlink("/z");

```
inode bitmap    10000000
inodes          [d a:0 r:2]□□□□□□□□□
data bitmap     10000000
data            [(.,0) (..,0)]□□□□□□□□□

creat("/y");

inode bitmap    11000000
inodes          [d a:0 r:2][f a:-1 r:1]□□□□□□□
data bitmap     10000000
data            [(.,0) (..,0) (y,1)]□□□□□□□□□

link("/y", "/s");

inode bitmap    11000000
inodes          [d a:0 r:2][f a:-1 r:2]□□□□□□□
data bitmap     10000000
data            [(.,0) (..,0) (y,1) (s,1)]□□□□□□□□□

creat("/e");

inode bitmap    11100000
inodes          [d a:0 r:2][f a:-1 r:2][f a:-1 r:1]□□□□□□
data bitmap     10000000
data            [(.,0) (..,0) (y,1) (s,1) (e,2)]□□□□□□□□□
```

3） Seed23：./vsfs.py -n 6 -s 23 -r

```
Initial state

inode bitmap    10000000
inodes          [d a:0 r:2]□□□□□□□□□
data bitmap     10000000
data            [(.,0) (..,0)]□□□□□□□□□

mkdir("/c");

inode bitmap    11000000
inodes          [d a:0 r:3][d a:1 r:2]□□□□□□□
data bitmap     11000000
data            [(.,0) (..,0) (c,1)][(.,1) (..,0)]□□□□□□□

creat("/c/t");

inode bitmap    11100000
```

inodes [d a:0 r:3][d a:1 r:2][f a:-1 r:1]□□□□□□□
data bitmap 11000000
data [(.,0) (..,0) (c,1)][(.,1) (..,0) (t,2)]□□□□□□□□

unlink("/c/t");

inode bitmap 11000000
inodes [d a:0 r:3][d a:1 r:2]□□□□□□□
data bitmap 11000000
data [(.,0) (..,0) (c,1)][(.,1) (..,0)]□□□□□□□□

creat("/c/q");

inode bitmap 11100000
inodes [d a:0 r:3][d a:1 r:2][f a:-1 r:1]□□□□□□□
data bitmap 11000000
data [(.,0) (..,0) (c,1)][(.,1) (..,0) (q,2)]□□□□□□□□

creat("/c/j");

inode bitmap 11110000
inodes [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:1]□□□□□
data bitmap 11000000
data [(.,0) (..,0) (c,1)][(.,1) (..,0) (q,2) (j,3)]□□□□□□□□

link("/c/q", "/c/h");

inode bitmap 11110000
inodes [d a:0 r:3][d a:1 r:2][f a:-1 r:2][f a:-1 r:1]□□□□□
data bitmap 11000000
data [(.,0) (..,0) (c,1)][(.,1) (..,0) (q,2) (j,3) (h,2)]□□□□□□□□

4） Seed24 ./vsfs.py -n 6 -s 24 -r
Initial state

inode bitmap 10000000
inodes [d a:0 r:2]□□□□□□□□□
data bitmap 10000000
data [(.,0) (..,0)]□□□□□□□□□

mkdir("/z");

inode bitmap 11000000
inodes [d a:0 r:3][d a:1 r:2]□□□□□□□

```
data bitmap     11000000
data            [(.,0) (..,0) (z,1)][(.,1) (..,0)][][][][][][]

creat("/z/t");

inode bitmap    11100000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][][][][][]
data bitmap     11000000
data            [(.,0) (..,0) (z,1)][(.,1) (..,0) (t,2)][][][][][][]

creat("/z/z");

inode bitmap    11110000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:1][f a:-1 r:1][][][][]
data bitmap     11000000
data            [(.,0) (..,0) (z,1)][(.,1) (..,0) (t,2) (z,3)][][][][][][]

fd=open("/z/z", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap    11110000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:2 r:1][][][][]
data bitmap     11100000
data            [(.,0) (..,0) (z,1)][(.,1) (..,0) (t,2) (z,3)][y][][][][][]

creat("/y");

inode bitmap    11111000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:2 r:1][f a:-1 r:1][][][]
data bitmap     11100000
data            [(.,0) (..,0) (z,1) (y,4)][(.,1) (..,0) (t,2) (z,3)][y][][][][][]

fd=open("/y", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap    11111000
inodes          [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:2 r:1][f a:3 r:1][][][]
data bitmap     11110000
data            [(.,0) (..,0) (z,1) (y,4)][(.,1) (..,0) (t,2) (z,3)][y][v][][][][]
```

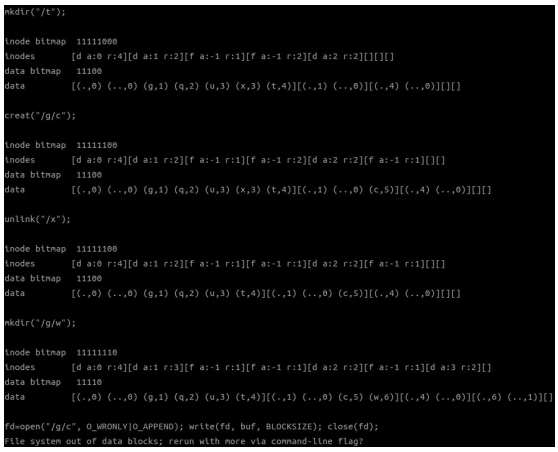
3. 现在将文件系统中的数据块数量减少到非常少（比如两个），并用 100 个左右的请求来运行模拟器。在这种高度约束的布局中，哪些类型的文件最终会出现在文件系统中？什么类型的操作会失败？

用命令进行模拟./vsfs.py -d 2 -c -n 100 -c

```
ARG seed 0
ARG numInodes 8
ARG numData 2
ARG numRequests 100
ARG reverse False
ARG printFinal False

Initial state

inode bitmap  10000000
inodes        [d a:0 r:2][][][][][][][]
data bitmap   10
data          [(.,0) (..,0)][]

mkdir("/g");
File system out of data blocks; rerun with more via command-line flag?
```

mkdir("/g");
File system out of data blocks; rerun with more via command-line flag?
看到只进行了 mkdir("/g")就已经报错，因为没有多余的数据块可以继续进行某些操作
我们可以尝试将数据块增加到 5 个 ./vsfs.py -d 5 -c -n 100 -c

```
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2][][][][][][][]
data bitmap   10000
data          [(.,0) (..,0)][][][][]

mkdir("/g");

inode bitmap  11000000
inodes        [d a:0 r:3][d a:1 r:2][][][][][][]
data bitmap   11000
data          [(.,0) (..,0) (g,1)][(.,1) (..,0)][][][]

creat("/q");

inode bitmap  11100000
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][][][][][]
data bitmap   11000
data          [(.,0) (..,0) (g,1) (q,2)][(.,1) (..,0)][][][]

creat("/u");

inode bitmap  11110000
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:1][][][][]
data bitmap   11000
data          [(.,0) (..,0) (g,1) (q,2) (u,3)][(.,1) (..,0)][][][]

link("/u", "/x");

inode bitmap  11110000
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:2][][][][]
data bitmap   11000
data          [(.,0) (..,0) (g,1) (q,2) (u,3) (x,3)][(.,1) (..,0)][][][]
```



发现可以进行的任务增加了许多

分析一下各个操作，mkdir 和 open,write,close 操作需要数据块，creat，link，unlink 操作不需要数据块，所以 mkdir 和 open,write,close 操作会导致错误，creat，link，unlink 操作则不会

4. 现在做同样的事情，但针对 inodes。只有非常少的 inode，什么类型的操作才能成功？哪些通常会失败？文件系统的最终状态可能是什么？

使用命令进行模拟./vsfs.py -i 5 -c -n 100

```
Initial state

inode bitmap  10000
inodes        [d a:0 r:2][][][][]
data bitmap   10000000
data          [(.,0) (..,0)][][][][][][][]

mkdir("/g");

inode bitmap  11000
inodes        [d a:0 r:3][d a:1 r:2][][][]
data bitmap   11000000
data          [(.,0) (..,0) (g,1)][(.,1) (..,0)][][][][][][]

creat("/q");

inode bitmap  11100
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][][]
data bitmap   11000000
data          [(.,0) (..,0) (g,1) (q,2)][(.,1) (..,0)][][][][][][]

creat("/u");

inode bitmap  11110
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:1][]
data bitmap   11000000
data          [(.,0) (..,0) (g,1) (q,2) (u,3)][(.,1) (..,0)][][][][][][]

link("/u", "/x");

inode bitmap  11110
inodes        [d a:0 r:3][d a:1 r:2][f a:-1 r:1][f a:-1 r:2][]
data bitmap   11000000
data          [(.,0) (..,0) (g,1) (q,2) (u,3) (x,3)][(.,1) (..,0)][][][][][][]

mkdir("/t");
File system out of inodes; rerun with more via command-line flag?
```

mkdir()和 creat()需要创建 inode，而 open(), write(), close() 和 link()、unlink()不需要 inode，所以 mkdir()和 creat()操作会导致失败，open(), write(), close() 、link()、unlink()操作不会。