作业4

Chp 37

T1

-a 0

T寻道=0（在同一条磁道上）
T旋转=165（6到0之前：180-15=165）
T传输=30

T总=T寻道+T旋转+T传输=195
模拟结果符合预期。

-a 6

T寻道=0（在同一条磁道上）
T旋转=345（6到6之前：360-15=345）
T传输=30

T总=T寻道+T旋转+T传输=375
模拟结果符合预期。

-a 30

T寻道=2*40=80（实际上在寻道的时候同时旋转也在进行）
T旋转=345-80=265（角度相当于是6到6和7的中间：360-15=345，再减去寻道的时候不计入这一部分）
T传输=30
T总=T寻道+T旋转+T传输=375

（与上一个实际上是一样的，这是因为寻道的同时也在旋转，实际上寻道时间被吞并了）
模拟结果符合预期。

-a 7，30，8

Block	T寻道	T旋转	T传输	T总
7	0	15	30	45
30	80	300-80	30	330
8	80	390-80	30	420

T总=T寻道+T旋转+T传输=795
模拟结果符合预期。

-a 10，11，12，13

这里模拟的应该是一个顺序读取情况。

Block	T寻道	T旋转	T传输	T总
10	0	105	30	135
11	0	0	30	30
12	80	360-80	30	390
13	0	0	30	30

T总=T寻道+T旋转+T传输=585
模拟结果符合预期。

T2

关于`-a 0`和`-a 6`的分析

当寻道时间为零时，这两种情况不会产生任何变化和影响。因为此时磁头不需要移动，直接进行数据读取操作。

★总体而言，可能出现以下几种情况：

1. 当寻道时间显著缩短时，可能避免磁盘多旋转一圈的情况，从而使得总时间以360度的整数倍大幅减少；
2. 若寻道时间减少幅度较小，则可能仍然受到旋转时间的"吞并"效应影响，导致总时间保持不变；
3. 当寻道速度v降低时，寻道时间可能呈指数级增长，进而对整体性能产生显著影响。

针对`-a 30`参数的分析

当寻道速度v=2时，单次寻道时间T=40/2=20ms。跨越两道所需时间为40ms。根据前述分析，此时寻道时间仍会被旋转时间所"吞并"。类似地，对于v=4、8、10、40等情况，虽然寻道时间进一步缩短，但仍会被"吞并"。

当v=0.1时，单次寻道时间T=40/0.1=400ms。跨越两道需要800ms。此时寻道时间已经大到无法被"吞并"，必须单独计算，同时还会影响旋转时间。根据关系式（寻道时间+旋转时间）=n×360+345，当n=2时，旋转时间为265ms。

各速度下的详细时间参数如下表所示：

寻道速率v	单次T寻道	T寻道	T旋转	T传输	T总	备注
1（基准）	40	80	265	30	375	
2	20	40	305	30	375	寻道时间缩短，总时间不变
4	10	20	325	30	375	寻道时间缩短，总时间不变
8	5	10	315	30	375	寻道时间缩短，总时间不变
10	4	8	337	30	375	寻道时间缩短，总时间不变
40	1	2	343	30	375	寻道时间缩短，总时间不变
0.1	400	800	265	30	1095	寻道时间延长，总时间增加

针对`-a 7, 30, 8`参数的分析

分析方法与前述类似，但需要考虑三次寻道操作的影响。值得注意的是，在v=0.1时，程序模拟可能出现寻道时间1601ms，旋转时间544ms的特殊情况，但这不影响最终结论。

各速度下的详细时间参数如下表所示：

寻道速率v	单次T寻道	T寻道	T旋转	T传输	T总	备注
1（基准）	40	160	545	90	795	
2	20	80	625	90	795	寻道时间缩短，总时间不变
4	10	40	305	90	435	寻道时间缩短，避免整圈旋转
8	5	20	325	90	435	寻道时间缩短，避免整圈旋转
10	4	16	329	90	435	寻道时间缩短，避免整圈旋转
40	1	4	341	90	435	寻道时间缩短，避免整圈旋转
0.1	400	1600	545	90	2235	寻道时间延长，总时间增加

针对`-a 10, 11, 12, 13`参数的分析

分析方法同上，需要考察四次寻道操作的影响。在v=0.1时，程序模拟可能出现寻道时间401ms，旋转时间424ms的特殊情况，但这不影响最终结论。

各速度下的详细时间参数如下表所示：

寻道速率v	单次T寻道	T寻道	T旋转	T传输	T总	备注
1（基准）	40	40	425	120	585	
2	20	20	445	120	585	寻道时间缩短，总时间不变

寻道速率v	单次T寻道	T寻道	T旋转	T传输	T总	备注
4	10	10	455	120	585	寻道时间缩短，总时间不变
8	5	5	460	120	585	寻道时间缩短，总时间不变
10	4	4	461	120	585	寻道时间缩短，总时间不变
40	1	1	464	120	585	寻道时间缩短，总时间不变
0.1	400	400	425	120	945	寻道时间延长，总时间增加

T3

磁盘访问时间分析：旋转速率的影响

基本原理说明

当访问同一磁道时，只需按比例增加旋转等待时间即可。而在跨磁道访问时，由于旋转时间相对增加（相当于寻道时间相对减少），可能避免磁盘多旋转一圈，从而显著降低总访问时间。

以`-a 7,30,8`为例，当旋转速率为0.1时，从30号扇区移动到8号扇区后，磁头只需旋转30度即可到达目标位置，无需完整旋转一圈。此时旋转时间为30/0.1=300ms，相比标准速率下的(30+360)/1=390ms反而更短。

具体案例分析

单扇区访问（-a 0）

访问0号扇区需要等待磁盘旋转165度：

- 旋转速率0.1：旋转时间1650ms，传输时间300ms
- 旋转速率0.5：旋转时间330ms，传输时间60ms
- 旋转速率0.01：旋转时间16500ms，传输时间3000ms

单扇区访问（-a 6）

与-a 0类似，但旋转角度为345度：

- 旋转速率0.1：旋转时间3450ms，传输时间300ms
- 旋转速率0.5：旋转时间690ms，传输时间60ms
- 旋转速率0.01：旋转时间34500ms，传输时间3000ms

跨磁道访问（-a 30）

包含80ms寻道时间：

- 旋转速率0.1：旋转时间3370ms，传输时间300ms
- 旋转速率0.5：旋转时间610ms，传输时间60ms

- 旋转速率0.01：旋转时间34420ms，传输时间3000ms

复杂跨磁道访问（-a 7,30,8）

包含160ms固定寻道时间：

- 旋转速率0.1：旋转时间3290ms，传输时间900ms
- 旋转速率0.5：旋转时间1250ms，传输时间180ms
- 旋转速率0.01：旋转时间34340ms，传输时间9000ms

连续扇区访问（-a 10,11,12,13）

包含40ms寻道时间：

- 旋转速率0.1：旋转时间4610ms，传输时间1200ms
- 旋转速率0.5：旋转时间890ms，传输时间240ms
- 旋转速率0.01：旋转时间46460ms，传输时间12000ms

性能影响总结

虽然降低旋转速率会显著增加旋转和传输时间，但由于寻道速度较快，在某些情况下（如跨磁道访问）可能通过减少旋转圈数来优化总时间。不过整体而言，旋转速率的降低会大幅增加访问延迟。

对比分析表

测试用例	R=0.1影响	R=0.5影响	R=0.01影响
-a 0	旋转时间大幅增加	旋转时间增加	旋转时间剧增
-a 6	旋转时间大幅增加	旋转时间增加	旋转时间剧增
-a 30	旋转时间大幅增加	旋转时间增加	旋转时间剧增
-a 7,30,8	减少旋转圈数，总时间优化	旋转时间增加	减少旋转圈数但总时间仍增加
-a 10,11,12,13	旋转时间大幅增加	旋转时间增加	旋转时间剧增

注：表格中的"剧增"表示时间增长幅度特别显著

T4

对于请求流-a 7,30,8，FIFO处理请求的顺序为7,30,8，会白白多了一次寻道和一圈的旋转；而最短寻道优先策略SSTF、电梯算法SCAN、最短定位时间优先SPTF的顺序都会是7,8,30。

由第一题我们知道使用后FIFO策略的时间为795

执行如下指令

```
python3 ./disk.py -a 7,30,8 -p SSTF -G
```

最终答案为375，几乎比之前的快了一倍。

T5

磁盘调度算法对比分析：SATF vs SSTF

案例1：`-a 7,30,8`的处理

当采用最短定位时间优先(SATF)策略处理该序列时，从旋转和寻道的综合角度分析，最优访问顺序确定为7→8→30。这一顺序无需调整，因为：

1. 首先访问7号扇区（最外层磁道）
2. 随后访问相邻的8号扇区（寻道时间最短）
3. 最后处理30号扇区（中间磁道）

执行命令及结果：

```
python3 ./disk.py -a 7,30,8 -p SATF -G
```

输出结果显示该顺序已是最优解，无需调整。

案例2：构造的对比序列`-a 7,20,35`

该序列的构造原理：

1. 初始位置在6号扇区
2. 首先访问相邻的7号扇区（最外层）
3. 精心设置中间磁道的20号扇区位置
4. 在SSTF策略下会陷入"最大代价陷阱"

SSTF策略的缺陷

执行命令：

```
python3 ./disk.py -a 7,20,35 -p SSTF -G
```

调度顺序：7→20→35
问题分析：

- 从7号移动到中间磁道的20号时，恰好错过目标位置
- 需要等待几乎完整的旋转周期（约360度）
- 之后才能访问35号扇区
- 总处理时间显著增加

SATF策略的优势

执行命令：

```
python3 ./disk.py -a 7,20,35 -p SATF -G
```

调度顺序：7→35→20
优势体现：

1. 从7号直接访问35号扇区（旋转定位最优）
2. 最后处理20号扇区
3. 完美避免完整旋转等待
4. 总处理时间大幅缩短

性能对比结论

指标	SATF策略	SSTF策略	优势幅度
寻道次数	2次	2次	持平
旋转等待	最优角度	近完整旋转	显著优势
总处理时间	最短	延长约360度旋转	30-40%

该案例充分证明：

1. SATF在综合考量旋转和寻道时更具优势
2. 特定序列下性能差异可达旋转周期的量级
3. 合理构造测试序列能有效验证算法优劣性
4. 实际系统中应采用SATF等综合优化算法

T6

磁道偏斜优化原理及计算

问题根源分析

当处理跨磁道请求序列$10, 11, 12, 13$时：

1. **同磁道访问**：10和11号扇区位于同一磁道（外层）
2. **跨磁道寻道**：处理完成后需移动到中间磁道访问12和13
3. **时间损耗**：寻道过程中磁盘持续旋转，导致目标扇区转离磁头位置，需等待完整旋转周期

优化原理

通过**磁道偏斜设计**，使目标磁道的起始扇区位置相对原磁道偏移特定角度。当完成寻道操作时，目标扇区刚好旋转至磁头下方，消除额外等待时间。

磁道偏斜计算公式推导

设参数：

- 旋转速率：p（度/单位时间）
- 寻道速率：v（磁道/单位时间）
- 相邻磁道间距：s（单位磁道数）
- 每磁道扇区数：n（个）
- 单个扇区角度：$\theta = \frac{360}{n}$（度）

核心不等式：

$$\text{磁道偏斜角度} > \text{寻道时间内的旋转角度}$$

计算步骤：

1. **相邻磁道寻道时间**：$T = \frac{s}{v}$（单位时间）
2. **寻道过程旋转角度**：$\alpha = p \times T = \frac{p \cdot s}{v}$（度）
3. **最小扇区偏斜数**：$x = \lceil \frac{\alpha}{\theta} \rceil = \lceil \frac{p \cdot s}{v \cdot \theta} \rceil$

Chp 38

T1

(1) RAID-0

```
./raid.py -D 4 -n 5 -L 0 -R 16

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]
```

(2) RAID-1

解答程序似乎在两个可选择的磁盘块中选择了和**offset**奇偶性相同的磁盘块

```
./raid.py -D 4 -n 5 -L 1 -R 8
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]
```

(3) RAID-4

```
./raid.py -D 5 -n 5 -L 4 -R 16
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]
```

(4) RAID-5

```
./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
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

区别：
RAID5左对称布局和左不对称布局的区别主要在数据块的分布上，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。

T2

RAID-0

```
./raid.py -D 4 -n 5 -R 16 -C 8192 -L 0

13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 2, offset 3]

6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 3, offset 0]

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 2, offset 2]

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

映射情况如下：

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

RAID-1

```
./raid.py -D 4 -n 5 -R 16 -C 8192 -L 1
13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 1, offset 7]

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

8 1
LOGICAL READ from addr:8 size:4096
  read  [disk 0, offset 4]

12 1
LOGICAL READ from addr:12 size:4096
  read  [disk 0, offset 6]

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

映射情况如下：

Disk0	Disk1	Disk2	Disk3
0	0	2	2
1	1	3	3
4	4	6	6
5	5	7	7

RAID-4

```
./raid.py -D 4 -n 5 -R 16 -C 8192 -L 4

13 1
LOGICAL READ from addr:13 size:4096
  read  [disk 0, offset 5]
6 1
LOGICAL READ from addr:6 size:4096
  read  [disk 0, offset 2]
8 1
LOGICAL READ from addr:8 size:4096
  read  [disk 1, offset 2]
12 1
LOGICAL READ from addr:12 size:4096
  read  [disk 0, offset 4]
7 1
LOGICAL READ from addr:7 size:4096
  read  [disk 0, offset 3]
```

映射情况如下：

Disk0	Disk1	Disk2	Disk3
0	2	4	P
1	3	5	P
6	8	10	P
7	9	11	P

RAID-5

左对称布局映射情况如下：

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

Disk0	Disk1	Disk2	Disk3
17	P	13	15

左不对称布局映射情况如下:

Disk0	Disk1	Disk2	Disk3
0	2	4	P
1	3	5	P
6	8	P	10
7	9	P	11
12	P	14	16
13	P	15	17

T3

RAID-0

```
./raid.py -D 4 -n 5 -L 0 -R 16 -r

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]
```

RAID-1

```
./raid.py -D 4 -n 5 -L 1 -R 8 -r

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]
```

RAID-4

```
./raid.py -D 5 -n 5 -L 4 -R 16 -r

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]
```

RAID-5

分别执行如下指令

```
python3 ./raid.py -D 5 -n 20 -L 5 -R 20 -5 LS -W seq -c
python3 ./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
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

T4

本题要求改变请求的大小，并使用-r反转，即观察不同大小的请求磁盘I/O的情况以及不同RAID级别下磁盘I/O的情况。

-S 8K

RAID0：

分别执行如下指令，观察读写请求

```
python3 ./raid.py -n 5 -L 0 -R 20 -r -S 8K -W seq
python3 ./raid.py -n 5 -L 0 -R 20 -r -S 8K -W seq -w 100
```

以读为例，如下：

```
0 2
LOGICAL OPERATION is ?
  read  [disk 0, offset 0]
  read  [disk 1, offset 0]

2 2
LOGICAL OPERATION is ?
  read  [disk 2, offset 0]
  read  [disk 3, offset 0]

4 2
LOGICAL OPERATION is ?
  read  [disk 0, offset 1]
  read  [disk 1, offset 1]

6 2
LOGICAL OPERATION is ?
  read  [disk 2, offset 1]
  read  [disk 3, offset 1]

8 2
LOGICAL OPERATION is ?
  read  [disk 0, offset 2]
  read  [disk 1, offset 2]
```

由于一次只能执行一个读操作，显然需要两次完成。

RAID1：

分别执行如下指令，观察读写请求

```
python3 ./raid.py -n 5 -L 1 -R 20 -r -S 8K -W seq
python3 ./raid.py -n 5 -L 1 -R 20 -r -S 8K -W seq -w 100
```

以写为例，如下：

```
0 2
LOGICAL OPERATION is ?
  write [disk 0, offset 0]    write [disk 1, offset 0]
  write [disk 2, offset 0]    write [disk 3, offset 0]

2 2
LOGICAL OPERATION is ?
  write [disk 0, offset 1]    write [disk 1, offset 1]
  write [disk 2, offset 1]    write [disk 3, offset 1]

4 2
LOGICAL OPERATION is ?
  write [disk 0, offset 2]    write [disk 1, offset 2]
  write [disk 2, offset 2]    write [disk 3, offset 2]

6 2
LOGICAL OPERATION is ?
  write [disk 0, offset 3]    write [disk 1, offset 3]
  write [disk 2, offset 3]    write [disk 3, offset 3]

8 2
LOGICAL OPERATION is ?
  write [disk 0, offset 4]    write [disk 1, offset 4]
  write [disk 2, offset 4]    write [disk 3, offset 4]
```

由于有镜像的存在，每次写入两个块，同时考虑镜像块，故有4次操作。

RAID4：

分别执行如下指令，观察读写请求

```
python3 ./raid.py -n 5 -L 4 -R 20 -r -S 8K -W seq
python3 ./raid.py -n 5 -L 4 -R 20 -r -S 8K -W seq -w 100
```

以写为例，如下：

```
0 2
LOGICAL OPERATION is ?
  read  [disk 2, offset 0]
  write [disk 0, offset 0]    write [disk 1, offset 0]    write
[disk 3, offset 0]

2 2
LOGICAL OPERATION is ?
  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 OPERATION is ?
  read  [disk 0, offset 1]
  write [disk 1, offset 1]    write [disk 2, offset 1]    write
[disk 3, offset 1]

6 2
LOGICAL OPERATION is ?
  read  [disk 2, offset 2]
  write [disk 0, offset 2]    write [disk 1, offset 2]    write
[disk 3, offset 2]

8 2
LOGICAL OPERATION is ?
  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]
```

正常情况下写入一个块，应该要先读地址对应的块，再读奇偶校验块，判断是否需要改变，然后再写入地址对应的块和校验块即可。

请求大小为8k的话，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），奇妙的事情发生了，此时由于这两个块共用同一个校验位，故只需要写入这两个快和校验位P，由原来的8个I/O操作减少到4个。

RAID5：

分别执行如下指令，观察读写请求

```
python3 ./raid.py -n 5 -L 5 -R 20 -r -S 8K -W seq
python3 ./raid.py -n 5 -L 5 -R 20 -r -S 8K -W seq -w 100
```

以写为例，如下：

```
0 2
LOGICAL OPERATION is ?
  read  [disk 2, offset 0]
  write [disk 0, offset 0]    write [disk 1, offset 0]    write
[disk 3, offset 0]

2 2
LOGICAL OPERATION is ?
  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 OPERATION is ?
  read  [disk 3, offset 1]
  write [disk 0, offset 1]    write [disk 1, offset 1]    write
[disk 2, offset 1]

6 2
LOGICAL OPERATION is ?
  read  [disk 0, offset 2]
  write [disk 2, offset 2]    write [disk 3, offset 2]    write
[disk 1, offset 2]

8 2
LOGICAL OPERATION is ?
  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基本相似。

-S 12K

分别执行如下指令完成测试

```
python3 ./raid.py -n 5 -L 0 -R 20 -r -S 12K -W seq
python3 ./raid.py -n 5 -L 0 -R 20 -r -S 12K -W seq -w 100
python3 ./raid.py -n 5 -L 1 -R 20 -r -S 12K -W seq
python3 ./raid.py -n 5 -L 1 -R 20 -r -S 12K -W seq -w 100
python3 ./raid.py -n 5 -L 4 -R 20 -r -S 12K -W seq
python3 ./raid.py -n 5 -L 4 -R 20 -r -S 12K -W seq -w 100
python3 ./raid.py -n 5 -L 5 -R 20 -r -S 12K -W seq
python3 ./raid.py -n 5 -L 5 -R 20 -r -S 12K -W seq -w 100
```

篇幅所限，测试结果不再赘述。总结如下：

请求大小修改为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。

RAID4的随机读和顺序读也与8K类似，需要3次读完成。随机写有所不同，如果在同一个条带上进行写，那只需要三个块进行异或，然后一次将包括校验块在内的四个块全部写入，故需要4次写。如果有2个块在同一条带上，那么这两个块的写入可以采取加法奇偶校验(4次写操作)，另一个单独在其他条带的块不论采用哪种方式，都需要4次写操作，故一共8次写操作。

RAID4顺序写时，写操作数明显减少了，因为每次请求都是对一个条带上的三个块进行写请求，可以采用全条带写入，即直接将三个块异或，然后全部和奇偶校验块一起写入。

RAID5的情况与RAID4是基本相同。

-S 16K

分别执行如下指令完成测试

```
python3 ./raid.py -n 5 -L 0 -R 20 -r -S 16K -W seq
python3 ./raid.py -n 5 -L 0 -R 20 -r -S 16K -W seq -w 100
python3 ./raid.py -n 5 -L 1 -R 20 -r -S 16K -W seq
python3 ./raid.py -n 5 -L 1 -R 20 -r -S 16K -W seq -w 100
python3 ./raid.py -n 5 -L 4 -R 20 -r -S 16K -W seq
python3 ./raid.py -n 5 -L 4 -R 20 -r -S 16K -W seq -w 100
python3 ./raid.py -n 5 -L 5 -R 20 -r -S 16K -W seq
python3 ./raid.py -n 5 -L 5 -R 20 -r -S 16K -W seq -w 100
```

篇幅所限，测试结果不再赘述。总结如下：

对于16K的请求，RAID0与RAID1的情况没有发生变化，只是需要多处理一个块。RAID0完成请求需要4次I/O。RAID1完成读需要4次读操作，完成写需要8次写操作。

RAID4的随机读和顺序读4次读操作完成。随机写时，有以下2种情况：

情况1：一个请求分布在两个条带上，两个条带上的块数分别为3,1，

情况2：一个请求分布在两个条带上，两个条带上的块数分别为2,2。

考虑3,1的情况，3个块在同一个条带上可以使用全条带写入(4次写)，剩下一个块4次写单独处理，共8次写。另一种2,2的情况，每一个条带上采用加法奇偶校验，各需要4次写，故也需要8次写。

对于顺序写，情况是与随机写相同的，因为请求大小比1个条带的数据块要多。因此顺序写也是以上的两种模式。

RAID5与RAID4基本相同。

总结

综合以上的所有分析，对于4个磁盘的情况下，请求块数越接近(小于等于)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。因为在这种情况下，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。

T5

具体性能书上有详细的表格：

表 38.10 RAID 容量、可靠性和性能

	RAID-0	RAID-1	RAID-4	RAID-5
容量	N	$N/2$	$N-1$	$N-1$
可靠性	0	1（肯定）		
		$N/2$（如果走运）		
吞吐量				
顺序读	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
顺序写	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
随机读	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
随机写	$N \cdot R$	$(N/2) \cdot R$	$1/2 \cdot R$	$N/4 \cdot R$

下面分别执行指令进行模拟

RAID0：

```
python3 ./raid.py -L 0 -t -n 100 -c
```

结果如下：

```
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
```

RAID1：

```
python3 ./raid.py -L 1 -t -n 100 -c
```

结果如下：

```
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
```

RAID4：

```
python3 ./raid.py -L 4 -t -n 100 -c
```

结果如下：

```
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
```

RAID5：

```
python3 ./raid.py -L 5 -t -n 100 -c
```

结果如下：

```
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
```

总结：

随机读写、顺序读写均与教材表格总结符合得较好。

Chp 40

T1

(1)

```
./vsfs.py -s 17
```

操作如下：

```
mkdir("/u");
creat("/a");
unlink("/a");
mkdir("/z");
mkdir("/s");
creat("/z/x");
link("/z/x", "/u/b");
unlink("/u/b");
fd=open("/z/x", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE);
close(fd);
creat("/u/b");
```

注意`link("/z/x", "/u/b");`的操作是给前者创建了一个名为后者的链接

(2)

```
./vsfs.py -s 18
```

操作如下:

```
mkdir("/f");
creat("/s");
mkdir("/h");
fd=open("/s", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE);
close(fd);
creat("/f/o");
creat("/c");
unlink("/c");
fd=open("/f/o", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE);
close(fd);
unlink("/s");
unlink("/f/o");
```

(3)

```
./vsfs.py -s 19
```

操作如下：

```
creat("/k");
creat("/g");
fd=open("/k", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE);
close(fd);
link("/k", "/b");
link("/b", "/t");
unlink("/k");
mkdir("/r");
mkdir("/p");
mkdir("/r/d");
link("/g", "/s");
```

(4)

```
./vsfs.py -s 20
```

操作如下：

```
creat("/x");
fd=open("/x", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE);
close(fd);
creat("/k");
creat("/y");
unlink("/x");
unlink("/y");
unlink("/k");
creat("/p");
fd=open("/p", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE);
close(fd);
link("/p", "/s");
```

T2

(1)

```
./vsfs.py -s 21 -r -n 6
```

```
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)]
[][][][][][][]
```

(2)

```
./vsfs.py -s 22 -r -n 6
```

```
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)

```
./vsfs.py -s 23 -r -n 6
```

```
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)

```
./vsfs.py -s 24 -r -n 6
```

```
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][][][][
[]
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][][][][][]
```

结论

关于 inode 和数据块分配算法，程序会优先分配索引最小的 inode 块和数据块。

T3

(1)

执行命令：

```
./vsfs.py -d 2 -c -n 100
```

结果：

```
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?
```

原因分析：
由于数据块太少，而无法创建新的文件夹。

(2)

最终会出现在文件系统的文件类型：
根目录，空文件，硬链接
示例如下：

```
MINGW64:/c/Users/75990/[
75990@Smile MINGW64 ~/Desktop/Operating System/Homework/ostep-homework-master/file-implementation
$ python3 ./vsfs.py -d 2 -c -n 100 -s 12
ARG seed 12
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)][]

creat("/p");

inode bitmap  11000000
inodes        [d a:0 r:2][f a:-1 r:1][][][][][][][]
data bitmap   10
data          [(.,0) (..,0) (p,1)][]

link("/p", "/m");

inode bitmap  11000000
inodes        [d a:0 r:2][f a:-1 r:2][][][][][][][]
data bitmap   10
data          [(.,0) (..,0) (p,1) (m,1)][]

creat("/b");

inode bitmap  11100000
inodes        [d a:0 r:2][f a:-1 r:2][f a:-1 r:1][][][][][][]
data bitmap   10
data          [(.,0) (..,0) (p,1) (m,1) (b,2)][]
```

会失败的操作类型：

1. 写入操作（open/ write/ close）
2. 多余的 mkdir()

T4

因为mkdir()和creat()需要inode，而open(), write(), close() 、link()、unlink()不需要inode，所以mkdir()和creat()操作会失败，open(), write(), close() 、link()、unlink()操作不会失败。

文件系统可能的最终状态：

1. 存在一些硬链接
2. 除根目录外只有极少数的目录和文件
3. 有的文件有数据块