

2018 级《操作系统》期末

By dhy && selia

以下答案不保证正确，仅供参考。

一、存储管理1

1

进程的地址空间为 4 GB，共 2^{32} 字节。

2

页目录的逻辑地址为 $0xMN000000 + (0xMN000000 \gg 10)$ 。

页表的 4 MB 空间是全 4 GB 空间的第 $(0xMN000000 \gg 12)$ 页，这部分空间对应到页表中的第 $(0xMN000000 \gg 12)$ 项，页表项 4 B，故页目录起始地址相对页表起始地址的偏移为 $(0xMN000000 \gg 12 \ll 2) = (0xMN000000 \gg 10)$ ，完整逻辑地址为 $0xMN000000 + (0xMN000000 \gg 10)$ 。

自映射页表项的逻辑地址为 $0xMN000000 + (0xMN000000 \gg 10) + (0xMN000000 \gg 20)$ 。

页目录是 4 MB 页表中的第 $(0xMN000000 \gg 12 \gg 10) = (0xMN000000 \gg 22)$ 个页面，每个页表项 4 B，故页目录这一项相对页目录起始地址的偏移为 $(0xMN000000 \gg 22 \ll 2) = (0xMN000000 \gg 20)$ 。页目录的起始地址由上一问计算出，故该自映射页表项的逻辑地址为 $0xMN000000 + (0xMN000000 \gg 10) + (0xMN000000 \gg 20)$ 。

3

页目录物理地址为 $0x00200000$ ，大端存储。

逻辑地址格式：一级页表(10)-二级页表(10)-页内偏移(12)

物理地址格式：页框号(20)-页内偏移(12)

页表项(PTE)格式：页框号(20)-标志位(12)，标志位 0 为有效位，标志位 1 为读写位。

```
1 Load [0x00001034]
2 PDN = 0x000, PTN = 0x001, OFF = 0x034
3 1: *(Pde *)0x00200000 = 0x00100007, Valid, PTEEntry = 0x00100000;
4 2: *(Pte *)0x00100001 = 0x00002067, Valid, PGEntry = 0x00002000;
5 3: *(Byte*)0x00002034 = 0x12;
6 Ok, 0x12
7
8 Store [0x00C07665]
9 PDN = 0x003, PTN = 0x007, OFF = 0x665
10 1: *(Pde *)0x0020000C = 0x00103007, Valid, PTEEntry = 0x00103000;
11 2: *(Pte *)0x0010301C = 0xEEFF0067, Valid, PGEntry = 0xEEFF0000;
12 3: *(Byte*)0xEEFF0665
13 Ok
14
15 Store [0x00C005FF]
```

```

16 PDN = 0x003, PTN = 0x000, OFF = 0x5FF
17 1: *(Pde *)0x0020000C = 0x00103007, Valid, PTEEntry = 0x00103000;
18 2: *(Pte *)0x00103000 = 0x11220005, Valid, Not Writeable
19 Error
20
21 Load [0x00C03012]
22 PDN = 0x003, PTN = 0x003, OFF = 0x012
23 1: *(Pde *)0x0020000C = 0x00103007, Valid, PTEEntry = 0x00103000;
24 2: *(Pte *)0x0010300C = 0x00000007, Valid, PGEntry = 0x00000000;
25 3: *(Byte*)0x00000012 = 0x20;
26 Ok, 0x20
27
28 Load [0xFF80078F]
29 PDN = 0x3FE, PTN = 0x000, OFF = 0x78F
30 1: *(Pde *)0x00200FF8 = 0x001FE007, Valid, PTEEntry = 0x001FE000;
31 2: *(Pte *)0x001FE000 = 0x04150000, Invalid.
32 Error
33
34 Load [0xFFFFF00B]
35 PDN = 0x3FF, PTN = 0x3FF, OFF = 0x00B
36 1: *(Pde *)0x00200FFC = 0x001FF007, Valid, PTEEntry = 0x001FF000;
37 2: *(Pte *)0x001FFFFC = 0x00103067, Valid, PGEntry = 0x00103000;
38 3: *(Byte*)0x0010300B = 0xCC;
39 Ok, 0xCC

```

答案:

```

1 Load [0x00001034]: 0x12
2 Store [0x00C07665]: OK
3 Store [0x00C005FF]: Error
4 Load [0x00C03012]: 0x20
5 Load [0xFF80078F]: Error
6 Load [0xFFFFF00B]: 0xCC

```

二、存储管理2

1

页面走向: 5 4 3 2 4 5 4 1 5 2 5 4 5 2 1

```

1 OPT算法:
2 5: 5 - - ; Miss
3 4: 5 4 - ; Miss
4 3: 5 4 3 ; Miss
5 2: 5 4 2 ; Miss, 3 is out
6 4: 5 4 2 ; Hit
7 5: 5 4 2 ; Hit
8 4: 5 4 2 ; Hit
9 1: 5 1 2 ; Miss, 4 is out
10 5: 5 1 2 ; Hit
11 2: 5 1 2 ; Hit
12 5: 5 1 2 ; Hit
13 4: 5 4 2 ; Miss, 1 is out
14 5: 5 4 2 ; Hit
15 2: 5 4 2 ; Hit
16 1: 5 4 1 ; Miss, x is out

```

```

17 Total 7 misses.
18
19 FIFO算法：
20 5: 5 - - ; Miss
21 4: 4 5 - ; Miss
22 3: 3 4 5 ; Miss
23 2: 2 3 4 ; Miss, 5 is out
24 4: 2 3 4 ; Hit
25 5: 5 2 3 ; Miss, 4 is out
26 4: 4 5 2 ; Miss, 3 is out
27 1: 1 4 5 ; Miss, 2 is out
28 5: 1 4 5 ; Hit
29 2: 2 1 4 ; Miss, 5 is out
30 5: 5 2 1 ; Miss, 4 is out
31 4: 4 5 2 ; Miss, 1 is out
32 5: 4 5 2 ; Hit
33 2: 4 5 2 ; Hit
34 1: 1 4 5 ; Miss, 2 is out
35 Total 11 misses.
36
37 LRU算法：
38 5: 5 - - ; Miss
39 4: 4 5 - ; Miss
40 3: 3 4 5 ; Miss
41 2: 2 3 4 ; Miss, 5 is out
42 4: 4 2 3 ; Hit
43 5: 5 4 2 ; Miss, 3 is out
44 4: 4 5 2 ; Hit
45 1: 1 4 5 ; Miss, 2 is out
46 5: 5 1 4 ; Hit
47 2: 2 5 1 ; Miss, 4 is out
48 5: 5 2 1 ; Hit
49 4: 4 5 2 ; Miss, 1 is out
50 5: 5 4 2 ; Hit
51 2: 2 5 4 ; Hit
52 1: 1 2 5 ; Miss, 4 is out
53 Total 9 misses.

```

2

进程平均大小 1 MB , 页表项大小 8 B , 设页面大小为 p 字节, 则分页开销为: $\frac{2^{20} \cdot 8}{p} + \frac{p}{2}$ 。

使开销最小则 $p^2 = 2^{24}$, $p = 2^{12}$ 。页面大小为 4 KB 即 4096 字节。

给进程大小求最优页面大小, 直接背公式吧, [传送门 Coekjan's blog](#)

设进程平均大小为 s , 页面大小为 p , 页表项大小 e , 则分页开销为: $\frac{se}{p} + \frac{p}{2}$ 。(s 和 e 已知, 求 p)

使开销最小则需令 $\frac{se}{p} = \frac{p}{2}$, 即页面大小 $p = \sqrt{2se}$ 。

公式的含义: 前一项 $\frac{s}{p} \cdot e$ 表示进程空间所需页表项的占用空间 (页表项数量乘以大小), 后一

项 $\frac{p}{2}$ 表示进程的最后一页平均有一半是内碎片。

三、存储管理3

作业 X 请求 12 KB 内存，作业 Y 请求 30 KB 内存，作业 Z 请求 9 KB 内存。

到达顺序：X 到达、C 结束、D 结束、Y 到达、E 结束、Z 到达。

起始状态：

	StartAddr	SecSize	Usage
1	0	10K	A
2	10K	10K	Free
3	20K	12K	B
4	32K	2K	Free
5	34K	6K	C
6	40K	20K	Free
7	60K	24K	D
8	84K	8K	E
9	92K	18K	Free
10	110K	5K	F
11	115K	13K	G

采用 First Fit：

1	X start, requires 12 KB;		
2	StartAddr	SecSize	Usage
3	0	10K	A
4	10K	10K	Free
5	20K	12K	B
6	32K	2K	Free
7	34K	6K	C
8	40K	12K	X
9	52K	8K	Free
10	60K	24K	D
11	84K	8K	E
12	92K	18K	Free
13	110K	5K	F
14	115K	13K	G
15	C end;		
16	D end;		
17	StartAddr	SecSize	Usage
18	0	10K	A
19	10K	10K	Free
20	20K	12K	B
21	32K	8K	Free
22	40K	12K	X
23	52K	32K	Free
24	84K	8K	E
25	92K	18K	Free
26	110K	5K	F
27	115K	13K	G
28	Y start, requires 30KB;		
29	StartAddr	SecSize	Usage
30	0	10K	A
31	10K	10K	Free
32	20K	12K	B
33	32K	8K	Free
34	40K	12K	X

```

35 52K      30K      Y
36 82K      2K       Free
37 84K      8K       E
38 92K      18K      Free
39 110K     5K       F
40 115K     13K      G
41 E end;
42 StartAddr SecSize Usage
43 0          10K      A
44 10K       10K      Free
45 20K       12K      B
46 32K       8K       Free
47 40K       12K      X
48 52K       30K      Y
49 82K       28K      Free
50 110K      5K       F
51 115K      13K      G
52 Z start, requires 9KB;
53 StartAddr SecSize Usage
54 0          10K      A
55 10K       9K       Z
56 19K       1K       Free
57 20K       12K      B
58 32K       8K       Free
59 40K       12K      X
60 52K       30K      Y
61 82K       28K      Free
62 110K      5K       F
63 115K      13K      G

```

First Fit 最终内存分配表：

分区号	起始地址	分区大小	占用情况
1	0	10 K	作业 A
2	10 K	9 K	作业 Z
3	19 K	1 K	空闲
4	20 K	12 K	作业 B
5	32 K	8 K	空闲
6	40 K	12 K	作业 X
7	52 K	30 K	作业 Y
8	82 K	28 K	空闲
9	110 K	5 K	作业 F
10	115 K	13 K	作业 G

Best Fit 算法：

```

1 X start, requires 12KB;
2 StartAddr SecSize Usage

```

```

3 0 10K A
4 10K 10K Free
5 20K 12K B
6 32K 2K Free
7 34K 6K C
8 40K 20K Free
9 60K 24K D
10 84K 8K E
11 92K 12K X
12 104K 6K Free
13 110K 5K F
14 115K 13K G
15 C end;
16 D end;
17 StartAddr SecSize Usage
18 0 10K A
19 10K 10K Free
20 20K 12K B
21 32K 52K Free
22 84K 8K E
23 92K 12K X
24 104K 6K Free
25 110K 5K F
26 115K 13K G
27 Y start, requires 30KB;
28 StartAddr SecSize Usage
29 0 10K A
30 10K 10K Free
31 20K 12K B
32 32K 30K Y
33 62K 12K Free
34 84K 8K E
35 92K 12K X
36 104K 6K Free
37 110K 5K F
38 115K 13K G
39 E end;
40 StartAddr SecSize Usage
41 0 10K A
42 10K 10K Free
43 20K 12K B
44 32K 30K Y
45 62K 20K Free
46 92K 12K X
47 104K 6K Free
48 110K 5K F
49 115K 13K G
50 Z start, requires 9KB;
51 StartAddr SecSize Usage
52 0 10K A
53 10K 9K Z
54 19K 1K Free
55 20K 12K B
56 32K 30K Y
57 62K 20K Free
58 92K 12K X
59 104K 6K Free
60 110K 5K F

```

Best Fit 最终内存分配表:

分区号	起始地址	分区大小	占用情况
1	0	10 K	作业 A
2	10 K	9 K	作业 Z
3	19 K	1 K	空闲
4	20 K	12 K	作业 B
5	32 K	30 K	作业 Y
6	62 K	20 K	空闲
7	92 K	12 K	作业 X
8	104 K	6 K	空闲
9	110 K	5 K	作业 F
10	115 K	13 K	作业 G

四、磁盘管理

磁盘访问序列: `10 22 20 2 40 6 38`, 磁头初始位于 20, 正向, 移动 1 个柱面 4 ms, 共 51 柱面。

1

电梯调度算法处理请求, 访问顺序:

`1 | 20 -> 22 -> 38 -> 40 -> (50) -> 10 -> 6 -> 2`

寻道总时间: 从 20 先移动到 50 再反向移动到 2,

$$4 \times ((50 - 20) + (50 - 2)) = 4 \times 78 = 312\text{ms}.$$

2

1	Time	Pos	Queue
2	0	20	<code>10, 22, 2, 40, 6, 38</code>
3	8	22	<code>10, 2, 40, 6, 38</code>
4	72	38	<code>10, 2, 40, 6, 50, 1</code>
5	80	40	<code>10, 2, 6, 50, 1</code>
6	120	50	<code>10, 2, 6, 1, 10</code>
7	280	10	<code>2, 6, 1</code>
8	296	6	<code>2, 1</code>
9	312	2	<code>1</code>
10	316	1	<code>Null</code>

访问柱面次序:

`1 | 20 -> 22 -> 38 -> 40 -> 50 -> 10 -> 6 -> 2 -> 1`

寻道总时间：316ms。

五、进程同步与互斥1

```
1 Semaphore empty = 50; // 仓库空位数
2 Semaphore itemA = 0, itemB = 0; // A 产品和 B 产品的数量
3 Semaphore mutex = 1; // 控制对仓库的互斥访问
4
5 main() {
6     cobegin
7         ProducerA();
8         ProducerB();
9         for (int i = 0; i < 5; i++) {
10             ConsumerA();
11             ConsumerB();
12         }
13     coend
14 }
15
16 ProducerA() {
17     while (true) {
18         P(empty);
19         ProductA a = produceA();
20         P(mutex);
21         put(a);
22         V(mutex);
23         V(itemA);
24     }
25 }
26
27 ProducerB() {
28     while (true) {
29         P(empty);
30         ProductB b = produceB();
31         P(mutex);
32         put(b);
33         V(mutex);
34         V(itemB);
35     }
36 }
37
38 ConsumerA() {
39     while (true) {
40         P(itemA);
41         P(mutex);
42         ProductA a = takeA();
43         V(mutex);
44         consume(a);
45         V(empty);
46     }
47 }
48
49 ConsumerB() {
50     while (true) {
51         P(itemA);
52         P(mutex);
53         ProductB b = takeB();
```

```
54     V(mutex);
55     consume(b);
56     V(empty);
57 }
58 }
```

六、进程同步与互斥2

Version 1 摘自《The Little Book of Semaphore》

部分变量名有改动。

```
1 int passengers = 0;
2 Semaphore mutex = 1;
3 Semaphore multiplex = 50;
4 Semaphore bus = 0;
5 Semaphore allAboard = 0;
6
7 main() {
8     cobegin
9         Bus();
10        Passenger();
11    coend
12 }
13
14 Bus() {
15     P(mutex);
16     if (passengers > 0) {
17         V(bus); // 通知第一名乘客上车
18         P(allAboard);
19     }
20     V(mutex);
21     depart();
22 }
23
24 Passenger() {
25     P(multiplex);
26     P(mutex);
27     passengers++;
28     V(mutex);
29     P(bus);
30     V(multiplex);
31
32     boardBus();
33
34     passengers--;
35     if (passengers == 0) {
36         V(allAboard);
37     } else {
38         V(bus); // 接力棒，上车的乘客唤醒下一个乘客上车
39     }
40 }
```

Version 2 另一方法摘自《The Little Book of Semaphore》

部分变量名有改动。

```

1 int waiting = 0;
2 Semaphore mutex = 1;
3 Semaphore bus = 0;
4 Semaphore boarded = 0;
5
6 main() {
7     cobegin
8         Bus();
9         Passenger();
10    coend
11 }
12
13 Bus() {
14     P(mutex);
15     n = min(waiting, 50);
16     for (int i = 0; i < n; i++) {
17         V(bus);
18         P(boarded);
19     }
20
21     waiting = max(waiting - 50, 0);
22     V(mutex);
23     depart();
24 }
25
26 Passenger() {
27     P(mutex);
28     waiting++;
29     V(mutex);
30
31     P(bus);
32     boardBus();
33     V(boarded);
34 }
```

Version 3 自己写的

```

1 const int MAXN = 50;
2 int waitCount = 0; // 当前排队等待的人数
3 Semaphore mutexWaitCount = 1; // 对 waitCount 的互斥访问
4 Semaphore mutex = 1; // 对上车的互斥访问
5 Semaphore wait = 0; // 乘客排队阻塞，等待 bus 唤醒
6 Semaphore board = 0; // 乘客告知司机已上车
7
8 main() {
9     cobegin
10        Passenger();
11        Bus();
12    coend
13 }
14
15 Passenger() {
16     // 1. 先开始排队
17     P(mutexWaitCount);
18     waitCount++;
19     V(mutexWaitCount);
20 }
```

```

21     // 2. 等待被 Bus 唤醒
22     P(wait);
23     // 3. 开始上车
24     P(mutexWaitCount);
25     waitCount--;
26     V(mutexWaitCount);
27     P(mutex);
28     boardBus();
29     V(mutex);
30     V(board);
31 }
32
33 Bus() {
34     // 1. 如果无人等待则出发
35     P(mutexWaitCount);
36     if (waitCount == 0) {
37         V(mutexWaitCount);
38         depart();
39     } else {
39         // 2. 确定上车的人数, 依次唤醒
40         int n = waitCount <= MAXN ? waitCount : MAXN;
41         V(mutexWaitCount);
42         for (int i = 0; i < n; i++) {
43             V(wait);
44         }
45         // 3. 等待当前的全部乘客上车
46         for (int i = 0; i < n; i++) {
47             P(board);
48         }
49         // 4. 发车
50         depart();
51     }
52 }
53 }
```

Version 4 来自网络

```

1 #define N 50           //一辆车满载50人
2 Semaphore mutex = mutex2 = 1; // 用于保护waiting变量
3 Semaphore board = 1; // 用于保护上车
4 Semaphore queue = 0; // 用于排队等待一辆车, 初始队伍0人
5 int waiting = 0;      // 表示等待即将到来的车人数, 初始等车的人为0
6 int should_go_this_time=0;
7 // 乘客进程
8 void Passenger(){
9     P(mutex); // 保护waiting变量
10    waiting++; // 到来的时候必须要等待
11    V(mutex);
12    P(queue); // 占用一个排队名额, queue是负数时, |queue|=排队人数
13    // 此时被下面的bus唤醒, 接触排队阻塞, 继续
14    // 上面解决了这个人应该等待
15
16    P(mutex2);
17    boardBus(); // 巴士到达, 开始上车
18    waiting--; // 等待的人减少
19    should_go_this_time--;
20    V(mutex2);
21    // 若上车动作较慢, 还不能走
```

```

22     if(should_go_this_time == 0)
23         v(ready); // 告诉司机可以走了
24 }
25
26 // 巴士进程
27 void Bus(){
28     if(waiting == 0)
29         depart();
30     else{
31         // 确定了n之后，新来的同学也不能上车了
32         P(board); //开始保护上车进程
33         n = min(N, waiting); // 最多上50个
34         should_go_this_time = n;
35         for(int i = 0; i<n; i++){ //开始上车
36             V(queue); // 唤醒排队的n个人继续进程
37         }
38         V(board);
39
40         P(ready); // 等待所有人上完车
41         depart(); // 如果waiting==0, n==0, 直接就走了
42     }
43 }
```

七、死锁

1

初始状态

1	Process	Allocated				Need				---> OK
		R1	R2	R3	R4	R1	R2	R3	R4	
3	P1	0	1	2	0	0	0	0	0	--> OK
4	P2	1	0	0	0	0	8	5	0	
5	P3	1	3	5	4	1	0	0	2	
6	P4	0	2	3	2	0	0	2	0	--> OK
7	P5	0	0	1	4	0	6	4	2	
8	Free					0	3	2	0	

P1 不再需要资源，可正常结束，结束后变为

1	Process	Allocated				Need				---> OK
		R1	R2	R3	R4	R1	R2	R3	R4	
3	P2	1	0	0	0	0	8	5	0	
4	P3	1	3	5	4	1	0	0	2	
5	P4	0	2	3	2	0	0	2	0	--> OK
6	P5	0	0	1	4	0	6	4	2	
7	Free					0	4	4	0	

此时可用资源满足 P4 需求，故将资源分配给 P4，而后 P4 正常结束，系统状态变为

	Process	Allocated				Need			
		R1	R2	R3	R4	R1	R2	R3	R4
3	P2	1	0	0	0	0	8	5	0
4	P3	1	3	5	4	1	0	0	2
5	P5	0	0	1	4	0	6	4	2
6	Free					0	6	7	2

可用资源满足 P5 需求，可分配给 P5，而后 P5 正常结束；

	Process	Allocated				Need			
		R1	R2	R3	R4	R1	R2	R3	R4
3	P2	1	0	0	0	0	8	5	0
4	P3	1	3	5	4	1	0	0	2
5	Free					0	6	8	6

此时现有的资源不能满足 P2 和 P3 的需求，两个进程均在等待资源，故发生死锁。

2

由上一问的分析，增加 1 个 R1 资源即可破除死锁：

	Process	Allocated				Need			
		R1	R2	R3	R4	R1	R2	R3	R4
3	P2	1	0	0	0	0	8	5	0
4	P3	1	3	5	4	1	0	0	2
5	Free					1	6	8	6

P3 得以进行；

	Process	Allocated				Need			
		R1	R2	R3	R4	R1	R2	R3	R4
3	P2	1	0	0	0	0	8	5	0
4	Free					2	9	13	10

P2 也能进行。死锁被破除。

故最小的代价即为增加一个 R1 资源。

八、文件系统

1

每个文件有 10 个直接索引，该部分大小为 $10 \times 2 = 20\text{KB}$ ；1 个一级间接索引，每个索引块中含有 $\frac{2\text{KB}}{4\text{B}} = 512$ 个数据块地址，该部分大小为 $512 \times 2 = 1024\text{KB}$ ；1 个二级间接索引，二级间接索引数据块中有 512 个地址，指向 512 个索引块；每个索引块有 512 个地址，总共有 $512 \times 512 = 262144$ 个数据块，该部分大小为 $262144 \times 2 = 524288\text{KB}$ 。

故该文件系统中文件最大为 $524288 + 1024 + 20 = 525332\text{KB}$ 。

注：间接索引的索引块不算“文件大小”

2

大小为 2 KB 和大小为 1.5 KB 的文件均只需要 1 个直接索引。只考虑存储了文件数据的磁盘块（即不考虑 inode 的大小），则 2 KB 文件刚好占用 1 整个数据块，而 1.5 KB 的文件占用了 75% 的数据块空间。磁盘利用率为 $\frac{1}{2} \times 100\% + \frac{1}{2} \times 75\% = 87.5\%$ 。

若数据块大小改为 1 KB，则 2 KB 的文件需要 2 整个数据块，1.5 KB 的文件占用 2 个数据块，其中一个完全利用而另一个块利用率为 50%，总利用率为

$$\frac{1}{2} \times \frac{1}{2}(100\% + 100\%) + \frac{1}{2} \times \frac{1}{2}(100\% + 50\%) = 87.5\%.$$

3

答：由第 1 问计算过程分析知，随着文件大小的增大，文件数据的存储方式从直接索引逐渐转向一级、二级索引，对于大文件而言需要通过二级索引来读取。索引层级越多意味着访问磁盘的次数越多，故性能有所下降。

解决措施：对于文件中访问频繁的部分尽可能放置于直接索引或一级索引访问到的区域，利用局部性原理减少访问磁盘的次数；针对磁盘的访问引入高速缓存（Cache），优化磁盘的调度算法减少磁头移动等。

（主观题，答案不保证正确）

以下内容摘自 PPT：

提高文件系统性能的方法：目录项(FCB)分解、当前目录、磁盘碎片整理、块高速缓存、磁盘调度、提前读取、合理分配磁盘空间、信息的优化分布、RAID 技术等。