张慕晖的博客

LUX ET VERITAS

2018-05-22

《操作系统》2017年期末考试分析

试题来自2017年春季学期操作系统课期末考试。

填空题（30分）

同学们认真完成了从lab0 ~ lab8的所有实验，在实验实践过程中了解和学到了很多知识。下面是他们从最开始到近期的实验心得，请补充完整。

1

lab0: 小强发现完成实验需要在Linux下操作很多命令行工具，于是他认真学习了 lab0中的知识，了解到Linux中在命令行模式下可以通过执行命令(**1.1**)来显示当前目录的文件，如果编写的程序有语法错误，编译器(**1.2**)会报错，根据错误信息，可以修改程序，并可以通过硬件模拟器工具(**1.3**)来执行 ucore 操作系统。

1. ls
2. gcc
3. qemu

2

lab1: 小晔在bootloader的代码中添加了一条打印语句，但发现编译生成lab项目出错，原来在ucore中只要bootloader的执行代码段+数据段的长度超过了(**2.1**)字节，就无法形成合法有效的bootloader。开始写实验报告时，本来准备提交MS Word文档格式的实验报告，但仔细看过实验报告的提交要求，原来实验指导书中明确要求同学用(**2.2**)文档格式来提交实验报告，小晔之前没学过这个文档格式，不过上网一查，花很短时间就掌握了编写方法，迅速完成了lab1。

1. ~~446~~510

2. Markdown

根据https://zh.wikipedia.org/wiki/%E4%B8%BB%E5%BC%95%E5%AF%BC%E8%AE%B0%E5%BD%95，主引导扇区中代码区的大小最多为446字节。

2018.5.24 UPD：经过tsz同学的提醒，我查证了一下`lab1/tools/sign.c`（用于生成磁盘主引导扇区的代码），发现里面是这么写的：

```
char buf[512];
memset(buf, 0, sizeof(buf));
FILE *ifp = fopen(argv[1], "rb");
int size = fread(buf, 1, st.st_size, ifp);
if (size != st.st_size) {
    fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
    return -1;
}
fclose(ifp);
buf[510] = 0x55;
buf[511] = 0xAA;
```

所以虽然维基说的没问题，但是ucore没有管什么"标准MBR分区表规划"，直接把`0x55AA`之外的510字节全用上了。

3

lab2: 小晖需要了解x86的内存大小与布局，页机制，页表结构等。硬件模拟器提供了 128MB的内存，并设定一个页目录项（PDE）占用(**3.1**)个Byte，一个页表项（PTE）占用(**3.2**)个 Byte。在lab2中可通过(**3.3**)和(**3.4**)两种方式获取系统内存大小，并且由于空闲的RAM空间不连续，所以bootloader简化处理，从物理内存地址(**3.5**)起始填充ucore os kernel的代码段和数据段。在ucore建立完页表并进入页模式后，ucore代码段的起始物理地址对应的虚拟地址为(**3.6**)。

1. 4
2. 4
3. BIOS中断调用
4. 直接探测
5. 0x00100000
6. 0xC0100000

这块不愧是令人比较困惑。我可能需要复习一下ucore的内存映射方式了。

4

lab3: 小彤发现ucore在完成页机制建立后，内核某内存单元的虚拟地址va为 0xC2345678，且此时硬件模拟器模拟的的cr3寄存器的值为0x221000，则此va对应的页目录表的起始物理地址是(**4.1**)，此va对应的PDE的物理地址

是(**4.2**)。如果一个页（4KB/页）被置换到了硬盘某8个连续扇区（0.5KB/扇区），该页对应的页表项（PTE）的最低位--present 位应该为(**4.3**)，表示虚实地址映射关系不存在，而原来用来表示页帧号的高(**4.4**)位，恰好可以用来表示此页在硬盘上的起始扇区的位置（其从第几个扇区开始）。

1. 0x00221000
2. 0x00221C20
3. 0
4. 20

该va对应的页号和偏移量：

- 0xC2345678 = b11000010001101000101011001111000
- 页目录号 = b1100001000 = 0x308
- 页表号 = b1101000101 = 0x345
- 偏移量 = b011001111000 = 0x678

PDE地址 = CR3 + 4*页目录号 = 0x00221000 + 0xC20 = 0x00221C20

5

lab4: 小颖在理解进程管理中，仔细分析了ucore源码中的进程控制块数据结构(**5.1**)，且其中的关键域（也称field，字段）数据结构(**5.2**)用于保存被中断打断的运行现场，关键域数据结构(**5.3**)用于进行进程/线程上下文切换的保存与回复。

1. `proc_struct`
2. `trap_frame`
3. `context`

6

lab5: 小辰对用户进程的创建有了更深入的了解：用户进程在用户态下执行时，CS 段寄存器最低第两位的值为(**6.1**)。当ucore os kernel建立完毕第一个用户进程的执行环境后，通过执行x86机器指令(**6.2**)后，将从内核态切换到用户态，且将从用户进程的第一条指令处继续执行。当用户进程执行sys_exit系统调用后，ucore会回收当前进程所占的大部分资源，并把当前进程的状态设置为(**6.3**)。

1. 3
2. IRET
3. ZOMBIE

用户态下，CS段的RPL的值应该为3.

7

lab6: 小磊通过阅读代码，了解了ucore的调度框架和RR调度算法等，体会到调度本质上体现了对(**7.1**)资源的抢占，操作系统通过(**7.2**)来避免用户态进程长期运行，并获得控制权。

1. 处理机执行能力（时间片？）
2. 时钟中断

之所以1的回答是"处理机执行能力"，主要是因为第15讲里说，"处理机调度是管理处理机执行能力的资源"。我觉得答CPU之类的也可以。

8

lab7: 小航发现课本中阐述的同步互斥原理对实现细节简化了很多。在ucore中，通过利用x86机器指令(**8.1**)简洁地实现了入临界区代码，通过利用x86指令(**8.2**)简洁地实现了出临界区代码。通过分析ucore中管程的数据结构，可知道ucore中的管程机制是基于(**8.3**)机制和(**8.4**)机制来实现的。

1. CLI
2. STI
3. 信号量
4. 等待队列

有时候分不清楚CLI和STI。事实上，CLI的意思是"Clean IF"，即将IF置零，屏蔽中断；STI的意思是"Set IF"，即将IF置1，不屏蔽中断。姑且这样记一下吧。

9

lab8: 小行了解到ucore中的文件系统架构包含四类主要的数据结构， (**9.1**)：它主要从文件系统的全局角度描述特定文件系统的全局信息。 (**9.2**)：它主要从文件系统中单个文件的角度描述了文件的各种属性和数据所在位置。 (**9.3**)：它主要从文件系统的文件路径的角度描述了文件路径中的特定目录。 (**9.4**)： 它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。

1. `struct sfs_fs` （`struct sfs_super`？）
2. `struct sfs_inode`
3. ？？
4. `struct File`

Lab8中的各种结构好多，完全不知道该回答什么啊。

2018.5.24 UPD：今天tsz同学提醒我，这个问题是实验指导书中的原话：

```
从ucore操作系统不同的角度来看，ucore中的文件系统架构包含四类主要的数据结构，它们分别是：
* 超级块（SuperBlock）：它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个OS空间。
* 索引节点（inode）：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整个OS空间。
* 目录项（dentry）：它主要从文件系统的文件路径的角度描述了文件路径中的特定目录。它的作用范围是整
```

个OS空间。
* 文件（file）：它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

问答题（70分）

10. 同步互斥（10分）

下面列出的n个线程互斥机制的伪代码实现有误，请指出错误处，给出错误原因分析，描述错误会带来的后果（即给出反例：无法正确执行n线程有效互斥运行行为的执行序列）。最后请修正错误，使得伪代码正确。

```
INITIALIZATION:
    shared int num[n];
    for (j=0; j < n; j++) {
        num[j] = j;
    }

------------------------------
ENTRY PROTOCOL (for Thread i):
    num[i] = MAX(num[0], ..., num[n-1]) + 1;
    for (j=0; j < n; j++) {
        if ((num[j] > 0) && ((num[j] < num[i]) || (num[j] == num[i]) && (j < i))) {
            while (num[j] > 0) {}
        }
    }

----------------------------
EXIT PROTOCOL (for Thread i):
    num[i] = 0;
----------------------------
```

这道题和2016年期末的26.2题几乎一模一样，唯一的区别是此处初始化时将`num[j]`初始化为`j`，而不是`0`。这一点显然违背了"空闲则入"的原则：假如线程`n-1`想要进入临界区，它必须等待编号比它小的线程全部进入过临界区，这可能会导致饥饿，所以还是应该初始化为`0`。因此，在线程i执行ENTRY PROTOCOL之前，`num[i]`必然为0。

在我的理解中，共享数组`num[n]`有两个含义：

- `num[i] > 0`表示线程i正在等待或已经进入临界区；`num[i] = 0`表示线程i离开了临界区且并没有等待进入临界区
- `num[i] > 0`时，表示线程i等待的优先级，数字越大，优先级越低

基于以上的讨论，我认为，`num`数组中非0的值必须是互不相同的。从ENTRY PROTOCOL的实现可以看出，如果`num[i] = MAX(num[0], ..., num[n-1]) + 1`这一操作是原子的，则上述结论显然；如果这一操作不是原子的，则

可能会出现两个线程i和j的优先级相同的情况。不妨设i<j，且其他线程的`num`均为0。假如线程i在`num[i]`没有完成赋值之前被打断，切换到线程j，则j会发现其他线程的`num`均为0，于是进入临界区。之后切换回线程i，i检查时虽然发现`num[i] == num[j]`，但由于`i < j`，于是也进入临界区，破坏了"忙则等待"要求。

假如能够保证`num[i] = MAX(num[0], ..., num[n-1]) + 1`这一操作是原子的，则上述实现是正确的，且可以删去`(num[j] == num[i]) && (j < i)`这一判断条件。在检查条件过程中被打断并不会影响算法的正确性，因为，即使已经被检查过的线程的优先级发生了变化，它也只可能变成0（它不再进入临界区，没有影响）或者优先级比当前线程变得更大（它退出临界区之后又想重新进入，需要取max），不需要重新进行等待。

但是现在的问题是怎么实现取max操作的原子性。如果直接加个互斥锁，不免太过智熄。（那我们还实现软件方法的N线程互斥干啥……）那就直接加个共享变量作为自旋锁好了，而且需要保证赋值过程是原子的。……虽然这样也完全没有意义，难道要直接改成Eisenberg & McGuire算法么……

```
INITIALIZATION:
    shared int num[n];
    shared bool choose[n];
    for (j = 0; j < n; j++) {
        num[j] = 0;
    }
    mutex = 0;

------------------------------
ENTRY PROTOCOL (for Thread i):
    num[i] = MAX(num[0], ..., num[n-1]) + 1;  // do this atomically
    for (j = 0; j < n; j++) {
        if ((num[j] > 0) && ((num[j] < num[i]) || (num[j] == num[i]) && (j < i))) {
            while (num[j] > 0) {}
        }
    }

-----------------------------
EXIT PROTOCOL (for Thread i):
    num[i] = 0;
----------------------------
```

2018.5.24 UPD：tsz同学给出了一种想法，我还没有仔细思考过它的正确性：

```
------------------------------
ENTRY PROTOCOL (for Thread i):
    num[i] = MAX(num[0], ..., num[n-1]) + 1;  // do this atomically
    for (j = 0; j < n; j++) {
        if ((num[j] > 0) && ((num[j] < num[i]) || (num[j] == num[i]) && (j < i)) ||
flag[j] && (j < i)) {
            while (num[j] > 0) {}
        }
```

```
    }
```

11. 管程（10分）

下面是一类管程机制的实现伪代码。

```
IMPLEMENTATION:
monitor mt {
    -----variable in monitor-----------
    semaphore mutex;
    semaphore next;
    int next_count;
    condvar {int count, sempahore sem} cv[N];
    other shared variables in mt;
    ----condvar wait implementation----
    cond_wait (cv) {
        cv.count ++;
        if(mt.next_count>0)
            V(mt.next);
        else
            V(mt.mutex);
        P(cv.sem);
        cv.count --;
    }
    ----condvar signal implementation----
    cond_signal(cv) {
        if(cv.count>0) {
            mt.next_count ++;
            V(cv.sem);
            P(mt.next);
            mt.next_count--;
        }
    }
    ----routine examples in monitor----
    Routines_in_mt () {
        P(mt.mutex);
        real bodies of routines, may access shared variables, call cond_wait OR
cond_signal
        if(next_count>0)
        V(mt.next);
        else
        V(mt.mutex);
    }
}
```

在上述伪码中，如果有3个线程a,b,c需要访问管程，并会使用管程中的2个条件变量 cv[0],cv[1]。请问cv[i]->count含义是什么？cv[i]->count是否可能<0, 是否可能>1？请说明原因，并给出相应的3个线程同步互斥执行实例和简要解释。请问 mt->next_count含义是什么？mt->next_count是否可能<0, 是否可能>1？请说明原因，并给出相应的3个线程同步互斥执行过程实例和简要解释。

这道题和2016年期末的27题一模一样，连笔误都一样，不解释了。

12. 理发师问题（20分）

理发店里有m位理发师、m把理发椅和n把供等候理发的顾客坐的椅子。理发师为一位顾客理完发后，查看是否有顾客等待，如有则唤醒一位为其理发；如果没有顾客，理发师便在理发椅上睡觉。一个新顾客到来时，首先查看理发师在干什么，如果理发师在理发椅上睡觉，他必须叫醒理发师，然后理发师理发，顾客被理发；如果理发师正在理发，则新顾客会在有空椅子可坐时坐下来等待，否则就会离开。请用信号量机制实现理发师问题的正确且高效的同步与互斥活动：请说明所定义的信号量的含义和初始值，描述需要进行互斥处理的各种行为，描述需要进行同步处理的各种行为；要求用类C语言的伪代码实现，并给出必要的简明代码注释。

感觉这个的初步实现到处都是，比如http://whatbeg.com/2017/03/06/semaphore.html#问题8：理发师问题。但是我现在都要困死了，实在思考不了这种高思维含量的东西。

2018.5.24 UPD：今天tsz同学给出了一种做法：

```
// Customer
int Customer() {
    if (waitCnt > n)
        return FAIL;
    waitLock.P();
    waitCnt++;
    waitLock.V();
    waitList.P();

    // haircut

    waitLock.P();
    waitCnt--;
    waitLock.V();
    return SUCCEED;
}

// Barber
void Barber() {
    while (true) {
        while (waitCnt == 0);
        waitList.V();
    }
```

```
24  }
```

但我还没有仔细想过这个做法的正确性。

13. SPN算法（8分）

请给出平均周转时间的定义，请给出短进程优先算法的描述，请证明：短进程优先算法具有最小平均周转时间。

- 周转时间：进程从初始化到结束（包括等待）的总时间
- 平均周转时间：所有进程周转时间的平均数
- 短进程优先（SPN）算法：总是选择就绪队列中执行时间最短的进程占用CPU进入运行状态

证明：
假设就绪队列中共有N个进程，它们的执行时间分别为$t_1, t_2, \ldots, t_N$ 。不妨设$t_1 \leq t_2 \leq \ldots \leq t_N$ 。则SPN算法的总周转时间为

$$T = t_1 + (t_1 + t_2) + (t_1 + t_2 + t_3) + \ldots + (t_1 + t_2 + \ldots + t_N) = N * t_1 + (N-1) * t_2 + \ldots + t_N$$

假设我们交换了第i和j（i<j）个进程的执行顺序，则此时，总周转时间会变为

$$T' = N * t_1 + (N-1) * t_2 + \ldots + (N-i+1) * t_j + \ldots + (N-j+1) * t_i + t_N$$

$$T' - T = (N-i+1) * t_j + (N-j+1) * t_i - (N-i+1) * t_i - (N-j+1) * t_j = (i-j) * (t_i$$

由于任何进程执行顺序都可以通过对顺序排列的进程进行若干次交换而得到，上述证明表明，任何其他执行顺序得到的平均周转时间都不可能比SPN算法更优。因此，SPN算法具有最小平均周转时间。

14. LFU算法（14分）

LFU是最近最不常用页面置换算法(Least Frequently Used)，小白听到两个LFU定义的说法，有些糊涂：

1. 采用LFU算法的OS在碰到进程访问的物理内存不够时，换出进程执行期内被访问次数最少的内存页，当此页被换出后，其访问次数n会被记录下来，当此页被再次访问并被换入时，此页的访问次数为n+1。
2. 采用LFU算法的OS在碰到进程访问的物理内存不够时，换出进程执行期内被访问次数最少的内存页，当此页被换出后，其访问次数清零，当此页被再次访问并被换入时，此页的访问次数为1。

请问你认为那种LFU的定义是正确的？请分别回答第一种/第二种LFU定义是否有Belady 异常现象。如没有，请给出证明，如有，请给出会引起Belady异常现象的的页数/页帧数设置以及访问序列。

呃，我不会啊……但我认为做法1显然不太可取。LFU算法比较严重的一个问题是计数器的劣化（这个名字是我随便起的）：之前被大量访问，但以后不再被使用的页不容易被换出。为了解决这个问题，计数器可以定期右移之类的。现在页计数器根本不会减小，怕不是要出事……不过被换出的页大概被访问次数是很少的，所以我也不知道1有什么用。

2大概有Belady现象。1不知道。看来需要仔细研究一下Piazza上给出的例子了。

2018.5.24 UPD:
今天zp同学提醒我，Piazza上有一个帖子讨论了这一内容。我们一般说的LFU的定义是第2种，而非第1种；该帖子指出，第1种定义下LFU不会出现Belady问题（虽然没有给出证明），而第2种定义下LFU会出现Belady问题，并举出了例子。

令访问序列为[0 0 1 1 1 2 2 0 0 2 2 3 1 3 1 3 1 3 1 3 1 3 1 ... （之后无限循环3 1 3 1）]。在有2个物理页帧的情况下，访问过程是这样的（括号里是访问计数）：

访存	物理页a	物理页b	缺页	换出
0	0(1)	-	0	-
0	0(2)	-	-	-
1	0(2)	1(1)	1	-
1	0(2)	1(2)	-	-
1	0(2)	1(3)	-	-
2	2(1)	1(3)	2	0
2	2(2)	1(3)	-	-
0	0(1)	1(3)	0	2
0	0(2)	1(3)	-	-
2	2(1)	1(3)	2	0
2	2(2)	1(3)	-	-
3	3(1)	1(3)	3	2
1	3(1)	1(4)	-	-
3	3(2)	1(4)	-	-
1	3(2)	1(5)	-	-
3	3(3)	1(5)	-	-
1	3(3)	1(6)	-	-

可以看出，一共只会缺页6次，在之后的循环过程中不会缺页。

但是，如果物理页帧数量增加到3，访问过程会变成这样：

访存	物理页a	物理页b	物理页c	缺页	换出
0	0(1)	-	-	0	-
0	0(2)	-	-	-	-
1	0(2)	1(1)	-	1	-
1	0(2)	1(2)	-	-	-
1	0(2)	1(3)	-	-	-
2	0(2)	1(3)	2(1)	2	-
2	0(2)	1(3)	2(2)	-	-
0	0(3)	1(3)	2(2)	-	-
0	0(4)	1(3)	2(2)	-	-
2	0(4)	1(3)	2(4)	-	-
2	0(4)	1(3)	2(4)	-	-
3	0(4)	3(1)	2(4)	3	1
1	0(4)	1(1)	2(4)	1	3
3	0(4)	3(1)	2(4)	3	1
1	0(4)	1(1)	2(4)	1	3
3	0(4)	3(1)	2(4)	3	1
1	0(4)	1(1)	2(4)	1	3

可以看出，现在已经缺页9次了，而且之后每访问一次都会发生缺页。

上述讨论可以说明一般的LFU算法有Belady问题。那么为什么修改过的（有记忆的）LFU算法可以没有Belady问题呢？stackoverflow上是这么说的：

> http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-358.pdf section 1.3 defines the stack algorithm and finishes by working through an example of this for LFU. Basically you can maintain a stack as you follow through a trace of memory fetches such that the top i entries of the stack are the entries that will be held in memory if you have capacity for i entries in your memory. Since you can maintain such a stack a larger

> memory must always hold all of the entries kept in core for any smaller memory and so Belady's anomaly is not possible.
> Of course this assumes an exact implementation of LFU with counters of infinite capacity.

上面的内容大概是说，修改过的LFU算法实际上相当于维护了一个很大的访问次数栈，栈中的页按访问次数排序，顶端的N个页驻留在内存中。因此，它本质上是一种栈算法，所以不存在Belady问题。具体证明我现在没时间去看了，欢迎大家自己去看论文。

15. 小明文件系统（**8**分）

小明为更好理解lab8，设计了一个简化文件系统Xiao Miang File System, 简称 xmfs。

xmfs的系统调用接口包括：

- mkdir() - 创建一个新目录
- creat() - 创建一个空文件
- open(), write(), close() - 打开文件，写文件，关闭文件
- link() - 对文件创建一个硬链接（ hard link）
- unlink() - 对文件取消一个硬链接 (如果文件的链接数为0，则删除文件）
- **注意**：通过 write()对文件写一个数据buffer时，常规文件的最大size是一个 data block，所以第二次写（写文件的语义是在上次写的位置后再写一个data block）会报错（文件大小满了）。如果data block 也满了，也会报错。

xmfs在硬盘上的总体组织结构如下：

- superblock：记录可用inode数量，可用data block数量
- inode bitmap：已用/空闲inode的分配图（基于bitmap）
- inodes：inode的存储区域
- data bitmap：data block的分配图（基于bitmap）
- data：data block的存储区域

xmfs的关键数据结构--inode数据结构如下：
inode：包含3个fields（file type，data block addr of file content，reference count）,用list表示：

- file type: f -> 常规文件： regular file, d -> 目录文件： directory
- data block addr of file content: -1 -> file is empty
- reference count: file/directory 的引用计数，注意directory的引用计数是指在此目录中的inode的个数
- **注意**：比如，刚创建的一个空文件inode：[f a:-1 r:1]，一个有1个硬链接的文件inode：[f a:10 r:2]

xmfs的关键数据结构--数据块（data block）结构如下：

- 一般文件的内容表示：只是包含单个字符的 list，即占一个 data block，比如['a'], ['b'] .....
- 目录的内容表示：多个两元组（name, inode_number）形成的list，比如，根目录 [(.,0) (..,0)]，或者包含了一个'f'文件的根目录[(.,0) (..,0) (f,1)]。
- **注意**：一个目录的目录项的个数是有限的。block.maxUsed = 32
- **注意**：data block 的个数是有限的,为fs.numData
- **注意**：inode 的个数是有限的,为fs.numInodes

完整xmfs文件系统的参考实例：

fs.ibitmap: inode bitmap 11110000
fs.inodes: [d a:0 r:5] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
fs.dbitmap: data bitmap 11100000
fs.data: [(.,0) (..,0) (y,1) (z,2) (x,3)] [u] [(.,3) (..,0)] [] ...

对上述xmfs参考实例的解释：有8个inode空间,8个data blocks.其中，根目录包含5个目录项，“.”，“..”，“y”，“z”，“x”。而“y”是常规文件,并有文件内容，包含一个data block，文件内容为“u”。“z”是一个空的常规文件。“x”是一个目录文件，是空目录。

如果xmfs初始状态为：
inode bitmap 10000000
inodes [d a:0 r:2] [] [] [] [] [] [] []
data bitmap 10000000
data [(.,0) (..,0)] [] [] [] [] [] [] []

在执行了系统调用mkdir("/t")后，**xmfs的当前状态为：**
inode bitmap 11000000
inodes [d a:0 r:3] [d a:1 r:2] [] [] [] [] [] []
data bitmap 11000000
data [(.,0) (..,0) (t,1)] [(.,1) (..,0)] [] [] [] [] [] []

请问接下来的4个状态变化所对应系统调用是什么？ 要求回答格式象上面“mkdir("/t")”一样。

（1）
inode bitmap 11100000
inodes [d a:0 r:4] [d a:1 r:2] [f a:-1 r:1] [] [] [] [] []
data bitmap 11000000
data [(.,0) (..,0) (t,1) (y,2)] [(.,1) (..,0)] [] [] [] [] [] []

（2）
inode bitmap 11100000
inodes [d a:0 r:4] [d a:1 r:3] [f a:-1 r:2] [] [] [] [] []
data bitmap 11000000
data [(.,0) (..,0) (t,1) (y,2)] [(.,1) (..,0) (c,2)] [] [] [] [] [] []

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

（4）
inode bitmap 11110000
inodes [d a:0 r:5] [d a:1 r:3] [f a:2 r:2] [d a:3 r:2] [] [] [] []

data bitmap 11110000
data [(.,0) (..,0) (t,1) (y,2) (v,3)] [(.,1) (..,0) (c,2)] [o] [(.,3) (..,0)] [] [] [] []

---

1. create("/y")
2. link("/y", "/t/c")
3. fd=open("/y"), write(fd), close(fd)
4. mkdir("/v")

这道题的形式非常类似于MOOC上的期末试题中的第20题。

#OS

下一篇
《操作系统》2015年期末考试分析

上一篇
《操作系统》MOOC期末考试题分析

标签云

归档

十二月 2018 (18)
十一月 2018 (22)
十月 2018 (31)
九月 2018 (63)
八月 2018 (69)
七月 2018 (15)
六月 2018 (3)
五月 2018 (19)
四月 2018 (27)
二月 2018 (2)
一月 2018 (7)
十二月 2017 (9)
七月 2017 (11)
四月 2017 (1)
三月 2017 (2)
一月 2017 (1)

近期文章

为什么sigmoid和softmax需要和cross entropy一起计算
Leetcode 956. Tallest Billboard（DP）
Leetcode 955. Delete Columns to Make Sorted II（贪心）
Leetcode 954. Array of Doubled Pairs（贪心）
Leetcode 953. Verifying an Alien Dictionary（hash），及周赛（114）总结

友情链接

wenj
ssh