清华大学本科生考试试题专用纸

考试课程：**操作系统（A 卷）**　　时间：2017 年 05 月 19 日下午 12:45~15:05

系别：_____ 班级：_____ 学号：_____ 姓名：

答卷注意事项：
1. 答题前，请先在试题纸和答卷本上写明 A 卷或 B 卷、系别、班级、学号和姓名。
2. 在答卷本上答题时，要写明题号，不必抄题。
3. 答题时，要书写清楚和整洁。
4. 请注意回答所有试题。本试卷有 15 个题目，共 5 页。
5. 考试完毕，必须将试题纸和答卷本一起交回。

一、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。

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。

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__)。

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）。

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。

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

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__)。

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。

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。

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它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。

二、问答题（70 分）

10.（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;
-----------------------------
```

11.（10 分）下面是一类管程机制的实现伪代码。

```
0 IMPLEMENTATION:
1    monitor mt {
```

```
2     -----variable in monitor----------
3     semaphore mutex;
4     semaphore next;
5     int next_count;
6     condvar {int count, sempahore sem}   cv[N];
7     other shared variables in mt;
8     ----condvar wait implementation----
9     cond_wait (cv) {
10          cv.count ++;
11          if(mt.next_count>0)
12              V(mt.next)
13          else
14              V(mt.mutex);
15          P(cv.sem);
16          cv.count --;
17     }
18    ----condvar signal implementation----
19     cond_signal(cv) {
20          if(cv.count>0) {
21              mt.next_count ++;
22              V(cv.sem);
23              P(mt.next);
24              mt.next_count--;
25          }
26     }
27    ----routine examples in monitor----
28    Routines_in_mt () {
29        P(mt.mutex);
30        real bodies of routines, may access shared variables, call cond_wait OR cond_signal
31        if(next_count>0)
32              V(mt.next);
33        else
34              V(mt.mutex);
35    }
```

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。

12.（20 分）理发店理有 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。

13.（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。

14.（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。

15.（ 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)] [] [] [] []