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：2015 年 05 月 27 日下午 1:00~3:00

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，并优先回答你会和用时较少的题目。
4. 请注意回答所有试题。本试卷有 7 个题目，共 18 页。
5. 考试完毕，必须将试题纸和答卷本一起交回。

一、（10分）在用do_execve启动一个用户态进程时，ucore需要完成很多准备工作，这些工作有的在内核态完成，有的在用户态完成。请判断下列事项是否是ucore在正常完成do_execve中所需要的，如果是，指出它完成于内核态还是用户态（通过修改trapframe，在iret时改变寄存器的过程被认为是在内核态完成）。

a. 初始化进程所使用的栈；
b. 在栈上准备argc和argv的内容；
c. 将argc和argv作为用户main函数的参数放到栈上；
d. 设置EIP为用户main函数的地址；
e. 设置系统调用的返回值。

二、（18分）ucore lite建立了一个文件系统very simple file system, 简称vsfs。

vsfs的用户操作包括
mkdir() - 创建一个新目录
creat() - 创建一个空文件
open(), write(), close() - 对文件写一个数据buffer,注意常规文件的最大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）。或者如果data block也满了，也会报错。
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

vsfs的硬盘组织：
superblock : 可用inode数量，可用data block数量
inode bitmap ： 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

vsfs的关键数据结构：
inode数据结构：

inode : 包含3个fields, 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]

数据块内容结构：
一般文件的内容的表示：只是包含单个字符的list，即占一个data block，比如['a'], ['b'] .....
目录内容的表示： 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

完整文件系统的例子：

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

表明： 此文件系统有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vsfs初始状态为:

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

请问接下来的连续6个状态变化的对应用户操作是啥？

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

2) inode bitmap  11100000
inodes       [d a:0 r:3] [d a:1 r:3] [f a:-1 r:1] [] [] [] [] []
```

```
data bitmap   11000000
data         [(.,0) (..,0) (c,1)] [(.,1) (..,0) (h,2)] [] [] [] [] [] []

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

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

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

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

三、（16 分）在 ucore 中 enum proc_state 定义包含以下四个值：
PROC_UNINIT, PROC_SLEEPING, PROC_RUNNABLE, PROC_ZOMBIE.
请解释每一种状态的含义，以及各状态之间可能的迁移.

四、（15分）假设在lab6测试stride scheduling的过程中，采用如下默认配置：BigStride为0x7FFFFFFF，CPU时间片为50ms，测试过程包含五个进程，其初始stride均为1，优先级分别为1、2、3、4、5，测试时间为10s。下面给出了五种修改上述配置的方式，试讨论：对于每一种改动，测试结果相比改动之前是否会发生明显的变化？如果是，结果会变得更接近于理想情况，还是远离理想情况？

a. BigStride改为120；
b. CPU时间片改为5ms；
c. 五个进程的初始stride改为100；
d. 五个进程的优先级设为2、4、6、8、10；
e. 测试时间延长到20s。

五、（10 分）生产者-消费者问题是指，一组生产者进程和一组消费者进程共享一个初始为空、大小为 2 的缓冲区，只有缓冲区没满时，生产者才能把消息放入到缓冲区，否则必须等待；只有缓冲区不空时，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息，或者一个消费者从中取出消息。
下面是生产者-消费者问题的一个实现和测试结果。请回答下面问题：
1）请用伪码给出信号量的 PV 操作实现。
2）这个实现正确吗？如果不正确，给出你的正确实现。

3）这两个测试用例能发现该实现中的可能错误吗？如果不能，请给出你的尽可能完整的测试用例。

```
==== producer-consumer.cpp =====
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <cstring>
#include <unistd.h>
#include <string>
#include <cstdlib>
#include <new>        // ::operator new[]

using namespace std;

#define BUFFER_SIZE 3
#define SLEEP_SPAN 5
#define WORK_SPAN 4

#define PRODUCER 0
#define CONSUMER 1

int iflag = 0;
int oflag = 0;
sem_t empty, full, mutex;
int empty_count, full_count;
int data_num = 0;
int num = 0;

int buffer[BUFFER_SIZE] = {};

int p_task_done = -1;
int c_task_done = -1;

struct arg_struct{
   arg_struct(int _id, int _start, int _work, string _indent): id(_id), start(_start), work(_work),
indent(_indent){}
   arg_struct(int _id): id(_id), start(0), work(0), indent(string("")){}
   int id;
   int start;
   int work;
   string indent;
};

void* producer(void* argv){
   arg_struct arg = *(arg_struct*)argv;
   int id = arg.id;
```

```
    const char* indent = arg.indent.c_str();

    sleep(arg.start);

    printf("%sSTART\n", indent);

    sem_wait(&mutex);
    printf("%saMUTEX\n", indent);

    sem_wait(&empty);
    printf("%saEMPTY\n", indent);

    printf("%sENTER\n", indent);

    int time = rand()%SLEEP_SPAN;
    sleep(arg.work);

    p_task_done++;
    printf("%sProd %d\n", indent, p_task_done);

    buffer[iflag] = p_task_done;

    if (empty_count == 0) printf("Error: Produce while no empty\n");
    iflag = (iflag + 1) % BUFFER_SIZE;
    empty_count--;
    full_count++;

    printf("%sEXIT\n", indent);

    sem_post(&mutex);
    printf("%srMUTEX\n", indent);

    sem_post(&full);
    printf("%srFULL\n", indent);

    return NULL;
}

void* consumer(void* argv){
    arg_struct arg = *(arg_struct*)argv;
    int id = arg.id;
    const char* indent = arg.indent.c_str();

    sleep(arg.start);

    printf("%sSTART\n", indent);
```

```
    sem_wait(&full);
    printf("%saFULL\n", indent);

    sem_wait(&mutex);
    printf("%saMUTEX\n", indent);

    printf("%sENTER\n", indent);

    sleep(arg.work);

    ++c_task_done;
    if (full_count == 0) printf("Error: Consume while no full\n");

    int tmp = buffer[oflag];
    printf("%sCons %d\n", indent, tmp);

    oflag = (oflag + 1) % BUFFER_SIZE;
    if (c_task_done != tmp) printf("Error: Consume data wrong\n");
    if (c_task_done > p_task_done) printf("Error: Over-consume!\n");

    full_count--;
    empty_count++;

    printf("%sEXIT\n", indent);

    sem_post(&mutex);
    printf("%srMUTEX\n", indent);

    sem_post(&empty);
    printf("%srEMPTY\n", indent);

    return NULL;
}

#define N 3
void testcase_producer_consumer(int ThreadNumber, int inst[2 * N][3]){
    pthread_t * p_consumer = new pthread_t[ThreadNumber];
    pthread_t * p_producer = new pthread_t[ThreadNumber];

    int c_count = 0, p_count = 0;

    printf("testcase_producer_consumer:\n");
    /* For managed creation of 'ThreadNumber' threads */
    int st_time = 0;
```

```
    /* Print the first line */
    int tmp_c = 0, tmp_p = 0;
    for (int i = 0; i < ThreadNumber; i++){
        if (inst[i][0] == PRODUCER){
            printf("P%d\t", tmp_p++);
        } else if (inst[i][0] == CONSUMER){
            printf("C%d\t", tmp_c++);
        }
    }
    printf("\n");

    /* Create Producers and Consumers according to $inst*/
    int rc;
    string indent("");
    for (int i = 0; i < ThreadNumber; i++){
        if (inst[i][0] == PRODUCER){
            rc = pthread_create(p_producer + p_count, NULL, producer, new arg_struct(p_count, inst[i][1],
inst[i][2], indent));
            if (rc) printf("ERROR\n");
            p_count++;
        } else if (inst[i][0] == CONSUMER){
            rc = pthread_create(p_consumer + c_count, NULL, consumer, new arg_struct(c_count, inst[i][1],
inst[i][2], indent));
            if (rc) printf("ERROR\n");
            c_count++;
        }
        indent += '\t';
    }

    /* wait until every thread finishes*/
    for (int i = 0; i < p_count; i++){
        pthread_join(p_producer[i], NULL);
    }
    for (int i = 0; i < c_count; i++){
        pthread_join(p_consumer[i], NULL);
    }
    delete[] p_producer;
    delete[] p_consumer;
}
int main(int argc, char** argv) {
    srand((unsigned)time(NULL));

    memset(buffer, 0, sizeof(int) * BUFFER_SIZE);

    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
```

```
    sem_init(&full, 0, 0);

    empty_count = BUFFER_SIZE;
    full_count = 0;

    /* For managed creation of 2 * N threads */
    int ThreadNumber = 2 * N ;
    int st_time = 0;
    int inst[2 * N][3] = {
        /* { Consumer or Producer to be create?,
         When does it start to work after being created?, st_stime += N means it starts N seconcds later
than the previous P/C
         How long does it work after it enters critical zone? } */
        {CONSUMER, st_time += 0, 2},
        {CONSUMER, st_time += 1, 2},
        {CONSUMER, st_time += 2, 2},
        {PRODUCER, st_time += 3, 2},
        {PRODUCER, st_time += 4, 2},
        {PRODUCER, st_time += 5, 2}
    };
    testcase_producer_consumer(ThreadNumber, inst);
    st_time = 0;

    int inst2[2 * N][3] = {
        {PRODUCER, st_time += 0, 2},
        {PRODUCER, st_time += 1, 2},
        {CONSUMER, st_time += 2, 2},
        {CONSUMER, st_time += 3, 2},
        {PRODUCER, st_time += 4, 2},
        {CONSUMER, st_time += 5, 2}
    };
    testcase_producer_consumer(ThreadNumber, inst2);
    return 0;
}
```

测试用例的执行输出结果:

```
xyong@ubuntu-xyong:~/work$ gcc producer-consumer.cpp -lpthread -lstdc++
xyong@ubuntu-xyong:~/work$ ./a.out
testcase_producer_consumer:
C0      C1      C2      P0      P1      P2
START
        START
                START
                        START
                        aMUTEX
                        aEMPTY
```

```
                    ENTER
                    Prod 0
                    EXIT
                    rMUTEX
                    rFULL
aFULL
aMUTEX
ENTER
                          START
Cons 0
EXIT
                          aMUTEX
                          aEMPTY
                          ENTER
rMUTEX
rEMPTY
                          Prod 1
                          EXIT
                          rMUTEX
                          rFULL
      aFULL
      aMUTEX
      ENTER
      Cons 1
      EXIT
      rMUTEX
      rEMPTY
                                START
                                aMUTEX
                                aEMPTY
                                ENTER
                                Prod 2
                                EXIT
                                rMUTEX
                                rFULL
             aFULL
             aMUTEX
             ENTER
             Cons 2
             EXIT
             rMUTEX
             rEMPTY

testcase_producer_consumer:
P0      P1      C0      C1      P2      C2
START
```

```
aMUTEX
aEMPTY
ENTER
      START
Prod 3
EXIT
rMUTEX
rFULL
      aMUTEX
      aEMPTY
      ENTER
            START
            aFULL
      Prod 4
      EXIT
      rMUTEX
      rFULL
            aMUTEX
            ENTER
                  START
                  aFULL
            Cons 3
            EXIT
            rMUTEX
            rEMPTY
                  aMUTEX
                  ENTER
                  Cons 4
                  EXIT
                  rMUTEX
                  rEMPTY
                        START
                        aMUTEX
                        aEMPTY
                        ENTER
                        Prod 5
                        EXIT
                        rMUTEX
                        rFULL
                              START
                              aFULL
                              aMUTEX
                              ENTER
                              Cons 5
                              EXIT
                              rMUTEX
```

```
                                    rEMPTY
xyong@ubuntu-xyong:~/work$
```

六、（16 分）下面是关于 ucore 中用户程序的生命历程的代码。请完成下面填空和代码补全。

1）在 sh 的命令行上输入 "args 1" 启动用户程序 args，则 sh 会调用 _(1a)_ 创建新进程并调用 _(1b)_ 将 args 加载到该进程的地址空间中.（回答系统调用名称即可）

2）将 args 从硬盘加载主要由 load_icode 完成，请补全以下代码.

```
// load_icode -  called by sys_exec-->do_execve

static int
load_icode(int fd, int argc, char **kargv) {
   /* LAB8:EXERCISE2 YOUR CODE  HINT:how to load the file with handler fd  in to process's memory? how to setup argc/argv?
    * MACROs or Functions:
    *  mm_create       - create a mm
    *  setup_pgdir     - setup pgdir in mm
    *  load_icode_read  - read raw data content of program file
    *  mm_map          - build new vma
    *  pgdir_alloc_page - allocate new memory for  TEXT/DATA/BSS/stack parts
    *  lcr3            - update Page Directory Addr Register -- CR3
    */
    /* (1) create a new mm for current process
    * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
    *    (3.1) read raw data content in file and resolve elfhdr
    *    (3.2) read raw data content in file and resolve proghdr based on info in elfhdr
    *    (3.3) call mm_map to build vma related to TEXT/DATA
    *    (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file
    *         and copy them into the new allocated pages
    *    (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
    * (4) call mm_map to setup user stack, and put parameters into user stack
    * (5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
    * (6) setup uargc and uargv in user stacks
    * (7) setup trapframe for user environment
    * (8) if up steps failed, you should cleanup the env.
    */
   assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);

   if (current->mm != NULL) {
      panic("load_icode: current->mm must be empty.\n");
   }

   int ret = -E_NO_MEM;
   struct mm_struct *mm;
```

```c
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    struct Page *page;

    struct elfhdr __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, elf, _(2a)_, 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }

    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm, phnum;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
        if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0) {
            goto bad_cleanup_mmap;
        }
        if (ph->p_type != ELF_PT_LOAD) {
            _(2b)_
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            continue ;
        }
        vm_flags = 0, perm = PTE_U;
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }
        off_t offset = ph->p_offset;
        size_t off, size;
```

```
uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

ret = -E_NO_MEM;

end = ph->p_va + ph->p_filesz;
while (start < end) {
    if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
        ret = -E_NO_MEM;
        goto bad_cleanup_mmap;
    }
    off = start - la, size = PGSIZE - off, la += PGSIZE;
    if (end < la) {
        size -= la - end;
    }
    if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0) {
        goto bad_cleanup_mmap;
    }
    start += size, offset += size;
}
end = ph->p_va + ph->p_memsz;

if (start < la) {
    /* ph->p_memsz == ph->p_filesz */
    if (start == end) {
        continue ;
    }
    off = start + PGSIZE - la, size = PGSIZE - off;
    if (end < la) {
        size -= la - end;
    }
    memset(page2kva(page) + off, 0, size);
    start += size;
    assert((end < la && start == end) || (end >= la && start == la));
}
while (start < end) {
    if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
        ret = -E_NO_MEM;
        goto bad_cleanup_mmap;
    }
    off = start - la, size = PGSIZE - off, la += PGSIZE;
    if (end < la) {
        size -= la - end;
    }
    memset(page2kva(page) + off, 0, size);
    start += size;
}
```

```
    }
    sysfile_close(fd);

    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, _(2c)_, USTACKSIZE, vm_flags, NULL)) != 0) {
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);

    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //setup argc, argv
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {
        argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    argv_size = 0;
    for (i = 0; i < argc; i ++) {
        uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
        _(2d)_
    }

    stacktop = (uintptr_t)uargv - sizeof(int);
    *(int *)stacktop = _(2e)_;

    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    tf->tf_cs = USER_CS;
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = stacktop;
    tf->tf_eip = elf->e_entry;
    tf->tf_eflags = FL_IF;
    ret = 0;
out:
    return ret;
bad_cleanup_mmap:
```

```
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

3）完成加载后会从内核态回到用户态. 请补全此时的用户栈图示.
(假定未写入部分全部初始化为零, 注意使用小尾端)

```
0xb0000000
         -------------
0xafffffffc |00 31 00 00|
         -------------
0xafffff8 |61 72 67 73|  // 'args'
         -------------
0xafffff4 |    (3a)  |
         -------------
0xafffff0 |    (3b)  |
         -------------
0xaffffec |    (3c)  |
         -------------
0xaffffe8 |    (3d)  |
         -------------
```

此时并不会直接进入 main 函数, 而是执行以下代码, 请简述其作用.

```
//////// user/libs/initcode.S //////////
.text
.globl _start
_start:
    movl $0x0, %ebp

    movl (%esp), %ebx
    lea 0x4(%esp), %ecx

    subl $0x20, %esp

    pushl %ecx
    pushl %ebx

    call umain
1:  jmp 1b
////////// user/libs/umain.c ///////////
#include <ulib.h>
#include <unistd.h>
```

```c
#include <file.h>
#include <stat.h>
int main(int argc, char *argv[]);

static int
initfd(int fd2, const char *path, uint32_t open_flags) {
    int fd1, ret;
    if ((fd1 = open(path, open_flags)) < 0) {
        return fd1;
    }
    if (fd1 != fd2) {
        close(fd2);
        ret = dup2(fd1, fd2);
        close(fd1);
    }
    return ret;
}
void
umain(int argc, char *argv[]) {
    int fd;
    if ((fd = initfd(0, "stdin:", O_RDONLY)) < 0) {
        warn("open <stdin> failed: %e.\n", fd);
    }
    if ((fd = initfd(1, "stdout:", O_WRONLY)) < 0) {
        warn("open <stdout> failed: %e.\n", fd);
    }
    int ret = main(argc, argv);
    exit(ret);
}
```

4) 虽然 main 函数以 "return 0;" 结束，但此后程序仍在用户态，经过 _(4a)_ 进入内核态. 参考 do_exit 代码，其主要完成了页表和文件描述符的释放、设置进程状态和返回值、唤醒等待中的父进程（如果有）、_(4b)_. (while 循环部分) do_exit 中该进程占用的内存并未完全释放，例如 _(4c)_. 它们将在 _(4d)_ 中被释放.

```c
int
do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }

    struct mm_struct *mm = current->mm;
    if (mm != NULL) {
```

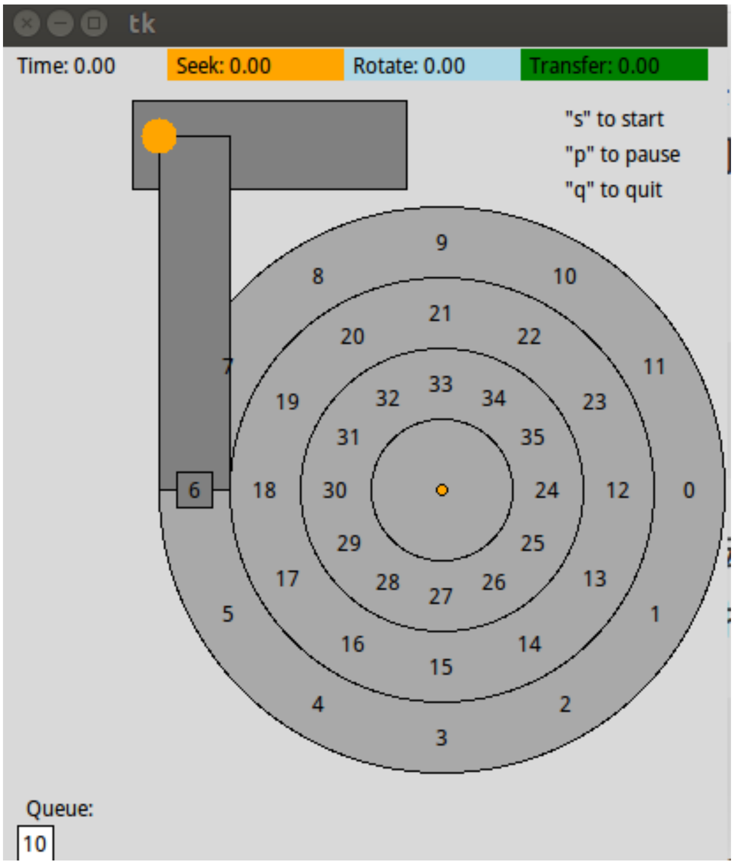
```
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    put_fs(current); //for LAB8
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;

    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);
        }
        while (current->cptr != NULL) {
            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
            proc->parent = initproc;
            initproc->cptr = proc;
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
        }
    }
    local_intr_restore(intr_flag);

    schedule();
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

七、（15分）一磁盘逆时针旋转，磁盘有3个磁道和一个磁头，每个磁道有12个扇区。最外侧磁道0包含扇区0~11, 中间侧磁道1包含扇区12~23,最内侧磁道包含扇区24~25. 如下图所示，可以看到磁头初始位置在外侧磁道的扇区6的中间位置，扇区10与扇区6在一个磁道上。



完成一次磁盘扇区的访问请求时间包括：
寻道时间(seek time)+旋转时间（rotational time）+传输时间（transfer time）。如，ucore发出访问请求序列为['10']，即只有一次对扇区10的访问请求，则磁盘花费的访问请求时间如下：

```
REQUESTS ['10']
Block:  10  Seek:  0  Rotate:105  Transfer: 30  Total: 135
TOTALS      Seek:  0  Rotate:105  Transfer: 30  Total: 135
```

表示寻道时间是0个时间单位，旋转时间是105个时间单位，传输时间是30个时间单位，总共的磁盘访问请求的时间是135.注意，相邻磁头移动一个磁道的时间是40个时间单位；从扇区6到扇区9，旋转了90度；而为了进行传输，需要从扇区9～10的中间位置开始，从扇区10～11的中间位置结束。所以需要再旋转15度，即旋转了105度，而每旋转1度花费1个时间单位，所以旋转花费了105个时间单位。

1)若采用FIFO磁盘调度策略，访问请求序列为['10', '12', '24', '1']，请按执行先后顺序列出完成每个磁盘请求的寻道时间(seek time)，旋转时间（rotational time），传输时间（transfer time）。
2)若采用SSFT磁盘调度策略，访问请求序列为['10', '12', '24', '1']，请按执行先后顺序列出完成每个磁盘请求的寻道时间(seek time)，旋转时间（rotational time），传输时间（transfer time）。