ucoreOS操作系统实验——Lab8

问题发现与改进

练习0

终于最后一次cv了真好。此外在 `proc.c` 中更改

```
static struct proc_struct *alloc_proc(void) {
    // 初始化 PCB 下的 fs(进程相关的文件信息)
    proc->filesp = NULL;
}
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    // 使用 copy_files()函数复制父进程的fs到子进程中
    if (copy_files(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
}
```

练习一

> 完成读文件操作的实现（需要编码）
>
> 首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，编写在sfs_inode.c中sfs_io_nolock读文件中数据的实现代码

打开文件的执行流程：

- 用户进程使用 `read(fd, data, len);` 读取磁盘上的文件
- 系统调用read->sys_read->syscall，进入内核态
- 通过ISR调用到sys_read内核函数，并进一步调用sysfile_read内核函数，进入文件系统抽象层
- 检查长度是否为0以及文件是否可读
- 分配buffer，即调用kmalloc分配4096字节的buffer空间，之后通过file_read每次读取buffer大小循环读文件，再通过调用copy_to_user函数将读到的内容拷贝到用户的内存空间中，直到指定len长度读取完毕，最后返回用户程序，用户程序收到读文件的内容
- 在file_read中，通过文件描述符查找到相应文件对应内存中的inode信息，然后转交给vop_read进行读取处理，实际上就是转交给sfs_read，调用sfs_io再进一步调用sfs_io_nolock

`sfs_io_nolock` 函数功能是针对指定的文件（文件对应的内存中的inode信息已经给出），从指定偏移量进行指定长度的读或者写操作

- 一系列的边界检查，访问是否合法
- 将读写操作实用的函数指针同一，针对整块操作
- 之后是我们要完成的部分，根据操作不落在整块数据块的各种情况进行分别处理

```
//LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf, sfs_rblock,etc. read different kind of b
  /*
   * (1) If offset isn't aligned with the first block, Rd/Wr some content from offset to the end of the first
   *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
   *               Rd/Wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset)
   * (2) Rd/Wr aligned blocks
   *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op
   * (3) If end position isn't aligned with the last block, Rd/Wr some content from begin to the (endpos % SFS
   *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
  */
  // 判断被需要读/写的区域所覆盖的数据块中的第一块是否是完全被覆盖的
  // 不是则需要调用非整块数据块进行读或写的函数来完成相应操作
  if (offset % SFS_BLKSIZE != 0 || endpos / SFS_BLKSIZE == offset / SFS_BLKSIZE)
  {
      blkoff = offset % SFS_BLKSIZE; // 计算出在第一块数据块中进行读或写操作的偏移量
      // 计算出在第一块数据块中进行读或写操作需要的数据长度
      // 若为同一块 则 size 为 endpos - offset
      // 若不为同一块 则 size 为 SFS_BLKSIZE - blkoff(偏移) 为 第一块要读的大小
      size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
      // 获取当前这个数据块对应到的磁盘上的数据块的编号
      if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0)
          goto out;
      if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0)
          goto out; // 将数据写入到磁盘中
      alen += size; // 维护已经读写成功的数据长度信息，真实长度
      buf += size;  // 维护缓冲区的偏移量
  }
  // 中间对齐
  uint32_t my_nblks = nblks;
  if (offset % SFS_BLKSIZE != 0 && my_nblks > 0) my_nblks --;
  if (my_nblks > 0) { // 判断是否存在被需要读写的区域完全覆盖的数据块
  // 如果存在，首先获取这些数据块对应到磁盘上的数据块的编号
      if ((ret = sfs_bmap_load_nolock(sfs, sin, (offset % SFS_BLKSIZE == 0) ? blkno: blkno + 1, &ino)) != 0)
          goto out;
      if ((ret = sfs_block_op(sfs, buf, ino, my_nblks)) != 0)
          goto out; // 将这些磁盘上的这些数据块进行读或写操作
      size = SFS_BLKSIZE * my_nblks;
      alen += size; // 维护已经读写成功的数据长度信息，真实长度
      buf += size; // 维护缓冲区的偏移量
  }
  // 判断需要读写的最后一个数据块是否被完全覆盖
  //  (这里还需要确保这个数据块不是第一块数据块，因为第一块数据块已经操作过了)
  if (endpos % SFS_BLKSIZE != 0 && endpos / SFS_BLKSIZE != offset / SFS_BLKSIZE) {
      size = endpos % SFS_BLKSIZE; // 确定在这数据块中需要读写的长度
      if ((ret = sfs_bmap_load_nolock(sfs, sin, endpos / SFS_BLKSIZE, &ino) == 0) != 0)
          goto out; // 获取该数据块对应到磁盘上的数据块的编号
      if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0)
          goto out; // 进行非整块的读或者写操作      You, a few seconds ago • Uncommitted changes
      alen += size;
      buf += size;
```

问题回答

请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，鼓励给出详细设计方案

- 可以考虑在磁盘上保留一部分空间或是一个特定的文件作为pipe机制的缓冲区
    - 当某两个进程之间要求建立管道，假设进程A的标准输出是进程B的标准输入，就可以在这两个进程的PCB上新增成员变量来记录进程的这种属性，同时生成一个临时的文件，将其在进程A、B中打开

- 当进程A使用标准输出进行write系统调用时，通过PCB中的变量得知，需要将这些标准输出的数据输出到先前创建的临时文件中
- 当进程B使用标准输入的时候进行read系统调用的时候，通过PCB中的变量给出信息，需要从上述临时文件中读取数据

练习二

> 改写proc.c中的load_icode函数和其他相关函数，实现基于文件系统的执行程序机制。执行：make qemu。如果能看看到sh用户程序的执行界面，则基本成功了。如果在sh用户界面上可以执行”ls”,”hello”等其他放置在sfs文件系统中的其他执行程序，则可以认为本实验基本成功

这次要完成的主要是具有从磁盘读取可执行文件并加载到内存功能的 `load_icode` ，和lab5中的区别是lab5仅将原先就加载到了内核内存空间中的ELF可执行文件加载到用户内存空间中，并没有涉及磁盘读取，而且也没有考虑给需要执行的应用程度传递操作的可能性

那么根据lab5中的help_comment， `load_icode` 实现流程：

- 给要执行的用户进程创建一个新的内存管理结构mm，因为原来的mm已经在 `do_execve` 中被释放了
- 创建用户内存空间的新的页目录表PDT
- 将磁盘上的ELF文件的TEXT/DATA/BSS段正确地加载到用户空间中
  - 从磁盘中读取elf文件的header
  - 根据elfheader中的信息，获取到磁盘上的program header
  - 对于每个program header
    - 为TEXT/DATA段在用户内存空间上的保存分配物理内存页，同时建立物理页和虚拟页的映射关系
    - 从磁盘上读取TEXT/DATA段，并且复制到用户内存空间上去；
    - 根据program header得知是否需要创建BBS段，如果是，则分配相应的内存空间，并且全部初始化成0，并且建立物理页和虚拟页的映射关系
  - 将用户栈的虚拟空间设置为合法，并且为栈顶部分先分配4个物理页，建立好映射关系
  - 切换到用户地址空间；
  - 设置好用户栈上的信息，即需要传递给执行程序的参数
  - 设置好中断帧

实现：

```
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 setu
     * 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.
    */
    if (current->mm != NULL)
    { // 判断当前进程的mm是否已经被释放掉了
        panic("load_icode: current->mm must be empty.\n");
    }
    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    // (1) create a new mm for current process
    if ((mm = mm_create()) == NULL) {
        // 为进程创建一个新的mm
        goto bad_mm;
    }
    // (2) create a new PDT
    if ((ret = setup_pgdir(mm)) != 0) {
        // 进行页表项的设置
        goto bad_pgdir_cleanup_mm;
    }
    // (3) copy TEXT/DATA/BSS section
    // (3.1) resolve elf header
    struct elfhdr elf, *elfp = &elf;
    off_t offset = 0;
    // 从磁盘上读取出ELF可执行文件的elf-header
    load_icode_read(fd, (void *) elfp, sizeof(struct elfhdr), offset);
    offset += sizeof(struct elfhdr);
    if (elfp->e_magic != ELF_MAGIC) {
        // 判断该ELF文件是否合法
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    struct proghdr ph, *php = &ph;
    uint32_t vm_flags, perm;
    struct Page *page;
    for (int i = 0; i < elfp->e_phnum; ++ i) {
        // 根据elf-header中的信息，找到每一个program header
        // (3.2) resolve prog header
        // 读取program header
        load_icode_read(fd, (void *) php, sizeof(struct proghdr), elfp->e_phoff + i * sizeof(struct proghdr));
        if (php->p_type != ELF_PT_LOAD) {
            continue;
        }
        if (php->p_filesz > php->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (php->p_filesz == 0) {
            continue;
        }
        // (3.3) build vma
        vm_flags = 0, perm = PTE_U;
        if (php->p_flags & ELF_PF_X) vm_flags |= VM_EXEC; // 根据ELF文件中的信息，对各个段的权限进行设置
        if (php->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (php->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        if ((ret = mm_map(mm, php->p_va, php->p_memsz, vm_flags, NULL)) != 0)
        { // 将这些段的虚拟内存地址设置为合法的
            goto bad_cleanup_mmap;
        }
        // (3.4) allocate pages for TEXT/DATA sections
        offset = php->p_offset;
        size_t off, size;
```

```
    uintptr_t start = php->p_va, end = php->p_va + php->p_filesz, la = ROUNDDOWN(start, PGSIZE);
    ret = -E_NO_MEM;
    while (start < end) {
        if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
            // 为TEXT/DATA段逐页分配物理内存空间
            goto bad_cleanup_mmap;
        }
        off = start - la, size = PGSIZE - off, la += PGSIZE;
        if (end < la) {
            size -= la - end;
        }
        load_icode_read(fd, page2kva(page) + off, size, offset);
        // 将磁盘上的TEXT/DATA段读入到分配好的内存空间中去
        //memcpy(page2kva(page) + off, page2kva(buff_page), size);
        start += size, offset += size;
    }

    // (3.5) allocate pages for BSS
    end = php->p_va + php->p_memsz;
    if (start < la) {
    // 如果存在BSS段，并且先前的TEXT/DATA段分配的最后一页没有被完全占用，则剩余的部分被BSS段占用，因此进行清零初始化
        if (start == end) {
            continue;
        }
        off = start + PGSIZE - la, size = PGSIZE - off;
        if (end < la) {
            size -= la - end;
        }
        memset(page2kva(page) + off, 0, size); // init all BSS data with 0
        start += size;
        assert((end < la && start == end) || (end >= la && start == la));
    }
    while (start < end) {  // 如果BSS段还需要更多的内存空间的话，进一步进行分配
        if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) { // 为BSS段分配新的物理内存页
            goto bad_cleanup_mmap;
        }
        off = start - la, size = PGSIZE - off, la += PGSIZE;
        if (end < la) {
            size -= la - end;
        }
        memset(page2kva(page), 0, size); // 将分配到的空间清零初始化
        start += size;
    }

}
sysfile_close(fd); // 关闭传入的文件，之后的操作中已经不需要读文件了

// (4) setup user stack
vm_flags = VM_READ | VM_WRITE | VM_STACK; // 设置用户栈的权限
if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) != 0)
{ // 将用户栈所在的虚拟内存区域设置为合法的
    goto bad_cleanup_mmap;
}
// setup args
uint32_t stacktop = USTACKTOP;
uint32_t argsize = 0;
for (int j = 0; j < argc; ++ j)
{  // 确定传入给应用程序的参数具体应当占用多少空间，即算出所有参数加起来的长度
    argsize += (1 + strlen(kargv[j])); // 包括 '\0'
}
argsize = (argsize / sizeof(long) + 1) * sizeof(long); //alignment
argsize += (2 + argc) * sizeof(long);
// 用户栈顶 减去所有参数加起来的长度 再 4字节对齐 找到 真正存放参数的栈的位置
```

```
    // 用户栈顶 减去所有参数加起来的长度 再 4字节对齐 找到 真正存放参数的栈的位置
    stacktop = USTACKTOP - argsize;
    uint32_t pagen = argsize / PGSIZE + 4;
    for (int j = 1; j <= 4; ++ j) { // 首先给栈顶分配四个物理页
        assert(pgdir_alloc_page(mm->pgdir, USTACKTOP - PGSIZE * j, PTE_USER) != NULL);
    }

    // for convinience, setup mm (5)
    // 切换到用户的内存空间，这样的话后文中在栈上设置参数部分的操作将简化
    // 具体因为空间不足而导致的分配物理页的操作已经交由page fault处理了，是完全透明的
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    // (6) setup args in user stack
    uint32_t now_pos = stacktop, argvp;
    *((uint32_t *) now_pos) = argc; // 设置好argc参数（压入栈）
    now_pos += 4;
    *((uint32_t *) now_pos) = argvp = now_pos + 4; // 设置argv数组的位置
    now_pos += 4;
    now_pos += argc * 4;
    for (int j = 0; j < argc; ++ j) {
        argsize = strlen(kargv[j]) + 1;  // 将argv[j]指向的数据拷贝到用户栈中
        memcpy((void *) now_pos, kargv[j], argsize);
        *((uint32_t *) (argvp + j * 4)) = now_pos; // 设置好用户栈中argv[j]的数值
        now_pos += argsize;
    }

    // (7) setup tf
    struct trapframe *tf = current->tf; // 设置中断帧
    memset(tf, 0, sizeof(struct trapframe));
    tf->tf_cs = USER_CS; // 需要返回到用户态，因此使用用户态的数据段和代码段的选择子
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = stacktop; //设置栈顶
    tf->tf_eip = elfp->e_entry; // 将返回地址设置为用户程序的入口
    tf->tf_eflags = 0x2 | FL_IF; // 允许中断      You, a few seconds ago • Uncommitted changes
    ret = 0;

out:
    return ret;
bad_cleanup_mmap: // 进行加载失败的一系列清理操作
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

问题回答

> 请在实验报告中给出设计实现基于”UNIX的硬链接和软链接机制“的概要设方案，鼓励给出详细设计方案

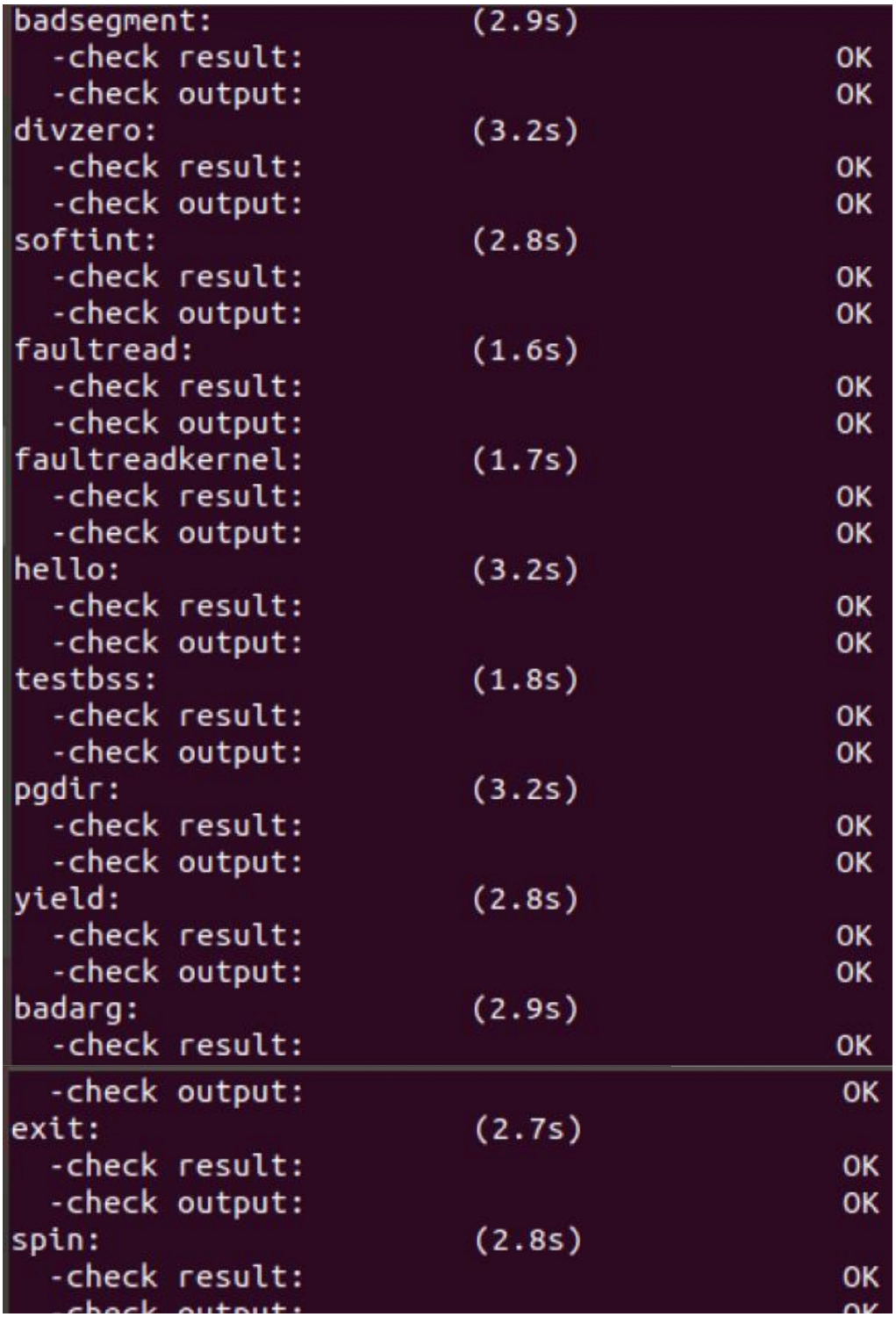
保存在磁盘上的inode信息均包含一个nlinks变量用于表示当前文件的被链接的次数

- 如果在磁盘上创建文件A的软链接B，首先将B当成正常的文件创建inode，然后将TYPE域设置为链接，然后使用剩余的域中的一个，指向A的inode位置，然后再额外使用一个位来标记当前的链接是软链接还是硬链接
- 当读写操作等系统调用访问B时，判断B是否为一个链接，则实际是将对B指向的文件A（已知A的inode位置）进行操作

- 当删除一个软链接B的时候，直接将其在磁盘上的inode删掉即可

- 如果在磁盘上的文件A创建一个硬链接B，那么在按照软链接的方法创建完B之后，还需要将A中的被链接的计数加1；
- 访问硬链接的方式与访问软链接一致
- 当删除一个硬链接B的时候，除了需要删除掉B的inode之外，还需要将B指向的文件A的被链接计数减1，如果减到了0，则将A删除

实验结果



```
  -check output:                                  OK
waitkill:                   (4.1s)
  -check result:                                  OK
  -check output:                                  OK
forktest:                   (2.8s)
  -check result:                                  OK
  -check output:                                  OK
forktree:                   (5.8s)
  -check result:                                  OK
  -check output:                                  OK
priority:                   (15.7s)
  -check result:                                  OK
  -check output:                                  OK
sleep:                      (11.6s)
  -check result:                                  OK
  -check output:                                  OK
sleepkill:                  (2.7s)
  -check result:                                  OK
  -check output:                                  OK
matrix:                     (15.6s)
  -check result:                                  OK
  -check output:                                  OK
Total Score: 190/190
```

ls：

```
lsdir: step 4
 @ is  [directory] 2(hlinks) 24(blocks) 6144(bytes) : @'.'
   [d]   2(h)       24(b)     6144(s)   .
   [d]   2(h)       24(b)     6144(s)   ..
   [-]   1(h)       10(b)    40152(s)   divzero
   [-]   1(h)       10(b)    40184(s)   forktree
   [-]   1(h)       10(b)    40156(s)   testbss
   [-]   1(h)       10(b)    40136(s)   sleepkill
   [-]   1(h)       10(b)    40132(s)   hello
   [-]   1(h)       10(b)    40136(s)   faultread
   [-]   1(h)       10(b)    40128(s)   spin
   [-]   1(h)       10(b)    40124(s)   pgdir
   [-]   1(h)       10(b)    40132(s)   softint
   [-]   1(h)       12(b)    48536(s)   sh
   [-]   1(h)       10(b)    40136(s)   badsegment
   [-]   1(h)       10(b)    40152(s)   sleep
   [-]   1(h)       10(b)    40236(s)   matrix
   [-]   1(h)       10(b)    40160(s)   forktest
   [-]   1(h)       10(b)    40156(s)   exit
   [-]   1(h)       10(b)    40140(s)   faultreadkernel
   [-]   1(h)       10(b)    40224(s)   priority
   [-]   1(h)       10(b)    40132(s)   yield
   [-]   1(h)       10(b)    40264(s)   waitkill
   [-]   1(h)       10(b)    40292(s)   ls
   [-]   1(h)       10(b)    40132(s)   badarg
   [-]   1(h)       10(b)    40276(s)   sfs_filetest1
lsdir: step 4
$
```

hello：


Hello world!!.
I am process 15.
hello pass.
$