操作系统原理实验

综合实验（UCORE LAB8++）

实验准备

> 主机环境：macOS X 10.14.5 Mojave
>
> 虚拟机环境：Linux Ubuntu 14.04 LTS
>
> 虚拟机搭载软件：Parallels Desktop.app
>
> 命令行终端：Linux 下 Terminal
>
> 终端shell：zsh
>
> 之前的第一次 ucore 实验中，实验用虚拟机硬盘文件.vdi搭建的虚拟机环境存在兼容性不好，卡顿严重，分辨率也不高等问题。故从这次实验开始，在Github上将原实验项目clone下来，自己按照指导书第一章讲的环境配置，安装所需支持，配置实用工具进行实验。
>
> 为满足实验要求，将命令行用户名字段临时指定为姓名。（需要在zsh主题文件中修改显示。因为中文姓名在命令行终端会影响显示效果，故用我的中大NetID作为署名标记：许滨楠 - xubn。）

实验目的

> - 考察对操作系统的文件系统的设计实现了解；
> - 考察操作系统内存管理的虚存技术的掌握；
> - 考察操作系统进程调度算法的实现。

实验内容

> - 在前面 ucore 实验 lab1-lab7 的基础上，完成 ucore 文件系统（lab8）；
> - 在上述实验的基础上，修改 ucore 调度器为多级反馈队列调度算法，队列共设6个优先级，最高级的时间片为 q，并且每降低 1 级，其时间片为上一级时间片 * 2；
> - 在上述实验的基础上，修改虚拟存储中的页面置换算法为某种工作集页面置换算法，具体如下：
>   - 对每一新建进程分配 3 帧物理页面；
>   - 当需要页面置换时，选择缺页次数最少的进程中的页面置换到外存；
>   - 对进程中的页面置换算法用改进的 clock 页替换算法。

LAB8 练习

练习0

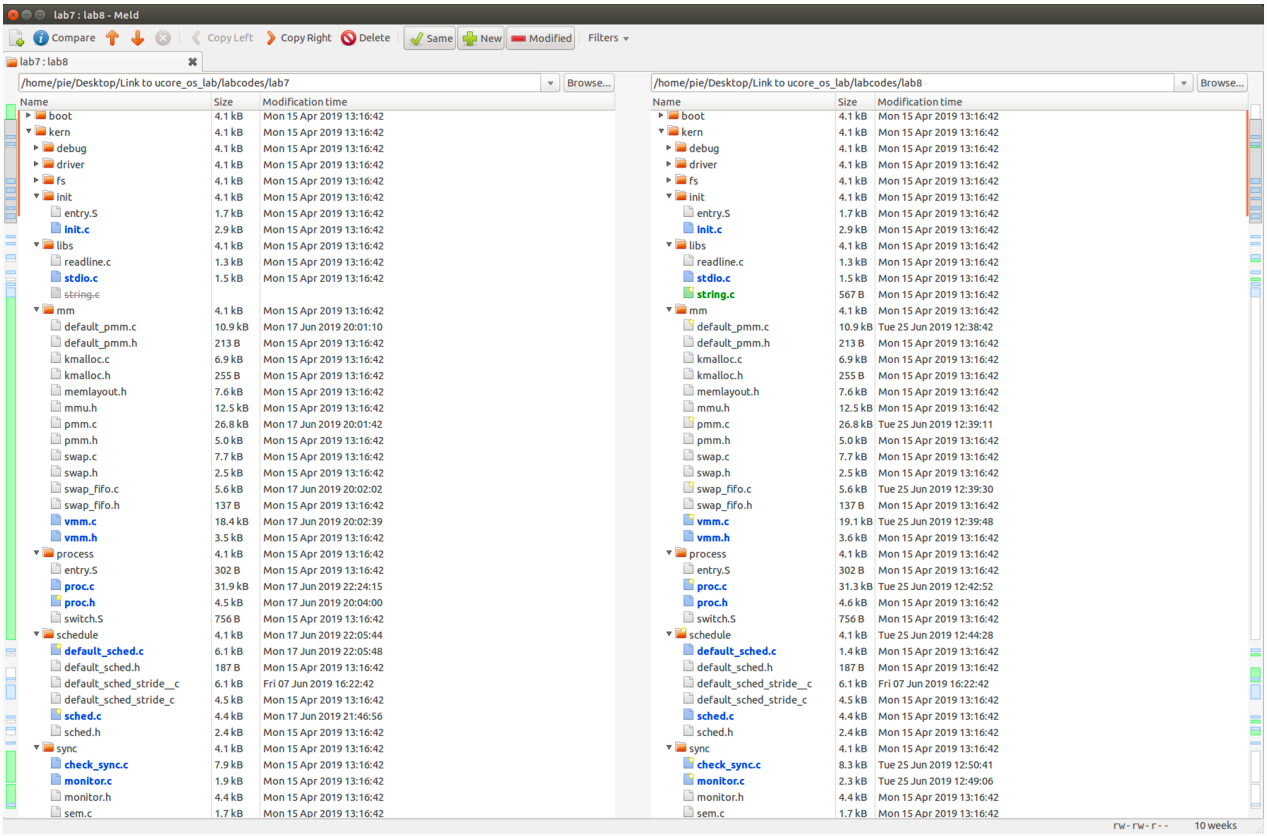
> 填写已有实验。

利用 meld 工具，将之前完成的 lab1/2/3/4/5/6/7 代码 merge 到 lab8 的代码中。因为之前已经将 lab1/2/3/4/5/6 的代码整合到 lab7 ，所以这次直接将 lab7 和 lab8 的代码在 meld 中进行目录比较，合并已经完成的代码。主要在之前编写过代码的文件中做比较，补充填入已完成的代码。由于之前在 merge 的时候错把一些新的 labcode 在与旧的 code merge 的时候去掉了，导致重要内容缺失，后续的测试总是出错，所以只需要把前面的 lab 练习中要求补充完整的函数 merge 到新的 lab 中。具体需要填写的文件和函数如下：

- Lab 1
  - kdebug.c : print_stackframe
- Lab 2
  - default_pmm.c : default_init
  - default_pmm.c : default_init_memmap
  - default_pmm.c : default_alloc_pages
  - default_pmm.c : default_free_pages
  - pmm.c : get_pte
  - pmm.c : page_remove_pte
- Lab 3
  - vmm.c : do_pgfault
  - swap_fifo.c : __fifo_map_swappable
  - swap_fifo.c : __fifo_swap_out_victim
- Lab 4 / 5
  - proc.c : alloc_proc
  - proc.c : do_fork
  - trap.c : idt_init
- Lab 6

    - kern/schedule/*
- Lab 7
    - 上个实验并没有完整的但是已经可以满分通过测试，将上个实验的代码内容全部补上即可

利用 meld 对照 Lab7 和 Lab8 代码，对 Lab8 源代码进行补充：



将前述内容，已经完成的练习代码补充到 lab8 初始代码中后，在命令行中使用 `make qemu` 命令可以验证编译是否通过来判断合并的基本正确性。利用 git 工具进行一次 commit，保存当前的工作状态到远程仓库备用。

练习1

> 完成读文件操作的实现。编写 sfs_inode.c 中 sfs_io_nolock 读文件中数据的实现代码。

了解打开文件的处理流程

读文件的时候需要系统内核态的权限，用户程序打开文件需要借助系统调用来实现。当用户程序执行 read 操作时，ucore 的执行过程从系统调用到逐个函数的调用，与要实现的 sfs_io_nolock 函数有关的调用链条如下：

/kern/syscall/syscall.c/sys_read —> /kern/fs/sysfile.c/sysfile_read —> /kern/fs/file.c/file_read —> /kern/fs/vfs/inode.h —> /kern/fs/vfs/inode.h #define vop_read —> /kern/fs/sfs/sfs_inode.c/sfs_read —> /kern/fs/vfs/sfs_inode.c/sfs_io

链条很长，先看到 sfs_io 函数：

```
static inline int
sfs_io(struct inode *node, struct iobuf *iob, bool write) {
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    struct sfs_inode *sin = vop_info(node, sfs_inode);
    int ret;
    lock_sin(sin);
    {
        size_t alen = iob->io_resid;
        ret = sfs_io_nolock(sfs, sin, iob->io_base, iob->io_offset, &alen, write);
        if (alen != 0) {
            iobuf_skip(iob, alen);
        }
    }
    unlock_sin(sin);
    return ret;
}
```

该函数接受三个参数依次代表文件的 i 节点、缓存、读写标志。先通过 i 节点找到文件的具体内容，包括其对应文件系统结构体和 i 节点结构体，接着就会进行原子的读取文件操作，通过调用 sfs_io_nolock 函数，传入之前获得的文件系统、i 节点、读写缓存、文件偏移、长度标识、读写标志位作为参数进行读写操作，接下来看到要实现的 sfs_io_nolock 函数：

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t offset, size_t *alenp, bool write) {
    struct sfs_disk_inode *din = sin->din;
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
    // calculate the Rd/Wr end position
    if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
        return -E_INVAL;
    }
    if (offset == endpos) {
        return 0;
    }
    if (endpos > SFS_MAX_FILE_SIZE) {
        endpos = SFS_MAX_FILE_SIZE;
    }
    if (!write) {
        if (offset >= din->size) {
            return 0;
        }
        if (endpos > din->size) {
            endpos = din->size;
        }
```

```
    }

    int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno, off_t offset);
    int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t nblks);
    if (write) {
        sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
    }
    else {
        sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
    }

    int ret = 0;
    size_t size, alen = 0;
    uint32_t ino;
    uint32_t blkno = offset / SFS_BLKSIZE;      // The NO. of Rd/Wr begin block
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/Wr blocks

//LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf, sfs_rblock,etc. read different kind of blocks in file
/*
 * (1) If offset isn't aligned with the first block, Rd/Wr some content from offset to the end of the first block
 *       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_BLKSIZE) of the last block
 *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
 */
out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}
```

从代码和注释中可以看出，这个函数，对由给定 inode 指定的文件资源进行访问，根据参数给定的偏移量作为操作长度进行读/写操作。

- 函数通过一系列的边界错误判定，排除了不合法的访问，保证访问操作的给定参数都合理且可以正常进行完毕；

- 函数申明了两个函数指针，用于根据读写标志位的传参情况选定对应的函数进行读写操作；
- *完成操作中变量的初始化之后，开始进行文件操作，其中分为几种情况进行讨论和定义操作：
    - *处理起始的没有对齐到块的部分，如果偏移量不是与第一块对齐的，则对从该偏移量开始到第一块结束的内容进行规定的读写操作，调用的是 sfs_rbuf 函数读写数据；
    - *处理对齐到块的部分，也就是以块为单位循环处理中间的部分，中间的整块读写调用 sfs_block_op函数；
    - *处理结尾的部分，如果结尾的位置没有和最后一块对齐，则最后一块的内容读/写到结束位置指示的块内位置为止，具体块内位置即为结束位置对块大小取模，结尾块也是调用 sfs_rbuf 函数进行读写。
- 最后通过判断文件长度信息，返回处理的字符长度值。

具体代码实现

练习 1 中主要需要实现的是上面的函数流程中 * 标的内容。其中用到的关键函数有：

- sfs_bmap_load_nolock 函数将 blkno 指示的索引对饮的一个 block 的索引值取出，存入相应的 i 节点存储单元 ino，如果成功则返回 0；
- sfs_block_op 函数指针调用 sfs_rwblock_nolock 函数完成对磁盘的具体读写操作；

对 sfs_io_nolock 函数中的练习部分补充内容如下：

```
// if offset isn't aligned with the first block
if ((blkoff = offset % SFS_BLKSIZE) != 0) {
    // if endpos and offset are in the same block, calculate the size of the first block to read
    size = nblks != 0 ? SFS_BLKSIZE - blkoff : endpos - offset;
    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;
    }
    alen += size;
    if (0 == nblks) {
        goto out;
    }
    buf += size;
    ++blkno;
    --nblks;
}

// operations on aligned blocks
size = SFS_BLKSIZE;
while (nblks != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
        goto out;
```

```
    }
    alen += size;
    buf += size;
    ++blkno;
    --nblks;
}

// if endpos isn't aligned with last block
if ((size = endpos % SFS_BLKSIZE) != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
        goto out;
    }
    alen += size;
}
```

分别对上一部分提到的三种不同情况的讨论进行操作，读写文件的头、中、尾三个部分并更新读写信息参量。

问题解答

> 给出设计实现 UNIX 的 PIPE 机制的概要设计方案。

PIPE 顾名思义是实现管道的机制，供进程通信使用。是一个存在于内存中的文件，可以参考 SFS 的设计，实现 PIPE。

首先初始化(STDIN, STDOUT) 的时候需要同时加入 PIPE 的初始化并且创建其索引节点，并为其分配缓存区。

对于数据的读写操作，PIPE 机制以 FIFO 的队列作为缓冲区的结构基础：

- 进程在写数据的时候，会判断缓冲区是否已满：
    - 如果缓冲区未满，则入队；
    - 如果缓冲区已满，阻塞进程；
- 进程在读数据的时候，会判断缓冲区是否为空：
    - 如果缓冲区非空，则可以将数据读出缓冲队列；
    - 如果缓冲区为空，阻塞进程；

添加相关的系统调用，暴露给用户进程使用，完成通信，主要要实现的功能有：

- 进程与管道 PIPE 的连接；
- 进程向管道 PIPE 中写入数据；
- 进程从管道 PIPE 中读出数据。

练习2

> 完成基于文件系统的执行程序机制的实现。

文件系统执行程序机制

练习中需要修改 proc.c 中的 load_icode 函数和其他相关函数 (alloc_proc, do_fork etc.)，实现基于文件系统的执行程序机制。基本的系统运行过程为从磁盘上读取可执行的文件，载入内存，然后由需要编写的 load_icode 函数完成内存空间的初始化。注释提示比较详尽，基本的流程大致如下：

- 1.为当前的用于进程创建一个新的内存管理结构 mm（因为原先该进程的 mm 在 do_execve 中被释放了）；
- 2.创建用户进程空间的页目录表 PDT，同时其页目录置为内核虚拟空间的地址；
- 3.将磁盘上的 ELF 文件中的各个二进制字段写入进程内存空间：
    - 3.1.从文件中读取数据内容并提取出 ELF 文件的 header；
    - 3.2.根据上一步骤读取到的 ELF header，获取磁盘上存储的 program header；
    - 3.3.调用 mm_map 为 TEXT/DATA 创建虚拟内存；
    - 3.4.调用 pgdir_alloc_page 为 TEXT/DATA 分配页，从文件中读取内容并将其拷贝到所分配的页中；
    - 3.5.调用 pgdir_alloc_page 为 BSS 分配页并初始置为0；
- 4.调用 mm_map 设置用户栈空间，将相关参数推入；
- 5.设置当前的进程内存空间、cr3 寄存器，利用 lcr3 宏定义重置 pgdir；
- 6.设置用户栈中的参数计数和具体参数；
- 7.设置用户环境的中断帧；
- 8.如果设置步骤失败，则需要清理环境。

具体代码实现

首先补充其他相关函数，需要根据注释添加一两行代码实现一些细节功能支持后续实验。

alloc_proc 函数中，需要完成进程控制块中与文件内容相关的初始化操作：

```
1 //LAB8:EXERCISE2 YOUR CODE HINT:need add some code to init fs in
  proc_struct, ...
2 proc->filesp = files_create();
```

do_fork 函数中，父进程的文件系统信息也要和其他信息一并复制到子进程中：

```
//LAB8:EXERCISE2 YOUR CODE HINT:how to copy the fs in parent's proc_struct?
int file_success = copy_files(clone_flags, proc);
if (file_success != 0) {
    goto bad_fork_cleanup_fs;
}
```

然后是对主要的函数 load_icode 按前述步骤进行补充：

```
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? */
    if (current->mm != NULL) {
        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) {
        goto bad_mm;
    }

    // 2.create a new PDT, and mm->pgdir= kernel virtual addr of PDT
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    // 3.copy TEXT/DATA/BSS parts in binary to memory space of process
    struct Page * page;
    //    3.1.read raw data content in file and resolve elfhdr
    struct elfhdr __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 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 ++) {
    //    3.2.read raw data content in file and resolve proghdr based on info in elfhdr
        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) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            continue ;
        }
    //    3.3.call mm_map to build vma related to TEXT/DATA
        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;
    //    3.4.callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file and copy them into the new allocated pages
        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;
        }
    //    3.5.callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
```

```
81              /* ph->p_memsz == ph->p_filesz */
82              if (start == end) {
83                  continue ;
84              }
85              off = start + PGSIZE - la, size = PGSIZE - off;
86              if (end < la) {
87                  size -= la - end;
88              }
89              memset(page2kva(page) + off, 0, size);
90              start += size;
91              assert((end < la && start == end) || (end >= la &&
start == la));
92          }
93          while (start < end) {
94              if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) ==
NULL) {
95                  ret = -E_NO_MEM;
96                  goto bad_cleanup_mmap;
97              }
98              off = start - la, size = PGSIZE - off, la += PGSIZE;
99              if (end < la) {
100                 size -= la - end;
101             }
102             memset(page2kva(page) + off, 0, size);
103             start += size;
104         }
105     }
106     sysfile_close(fd);
107
108     // 4.call mm_map to setup user stack, and put parameters into
user stack
109     vm_flags = VM_READ | VM_WRITE | VM_STACK;
110     if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE,
vm_flags, NULL)) != 0) {
111         goto bad_cleanup_mmap;
112     }
113     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE ,
PTE_USER) != NULL);
114     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE ,
PTE_USER) != NULL);
115     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE ,
PTE_USER) != NULL);
116     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE ,
PTE_USER) != NULL);
117
118     // 5.setup current process's mm, cr3, reset pgidr (using lcr3
MARCO)
119     mm_count_inc(mm);
120     current->mm = mm;
121     current->cr3 = PADDR(mm->pgdir);
122     lcr3(PADDR(mm->pgdir));
123
```

```
    // 6.setup uargc and uargv in user stacks
    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]);
        argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }
    stacktop = (uintptr_t)uargv - sizeof(int);
    *(int *)stacktop = argc;

    // 7.setup trapframe for user environment
    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;

    // 8.if up steps failed, you should cleanup the env.
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;
}
```

问题解答

> 给出基于 UNIX 的硬链接和软链接机制的概要设计方案。

通过在文件描述符中添加是否为硬链接的标志位，来实现硬链接的机制。当该标记位为特定值（比如1）则表示指定完成硬链接，同时在文件描述符中维护一个文件指针，指向链接的文件，同时会维护原文件的链接计数 nlinks（在引用文件的 inode 存储结构中）进行自增。在删除文件的时候进行判断，判断引用文件的链接计数，如果达到0，则不再有该文件的链接，此时对文件进行删除不会引发错误，可

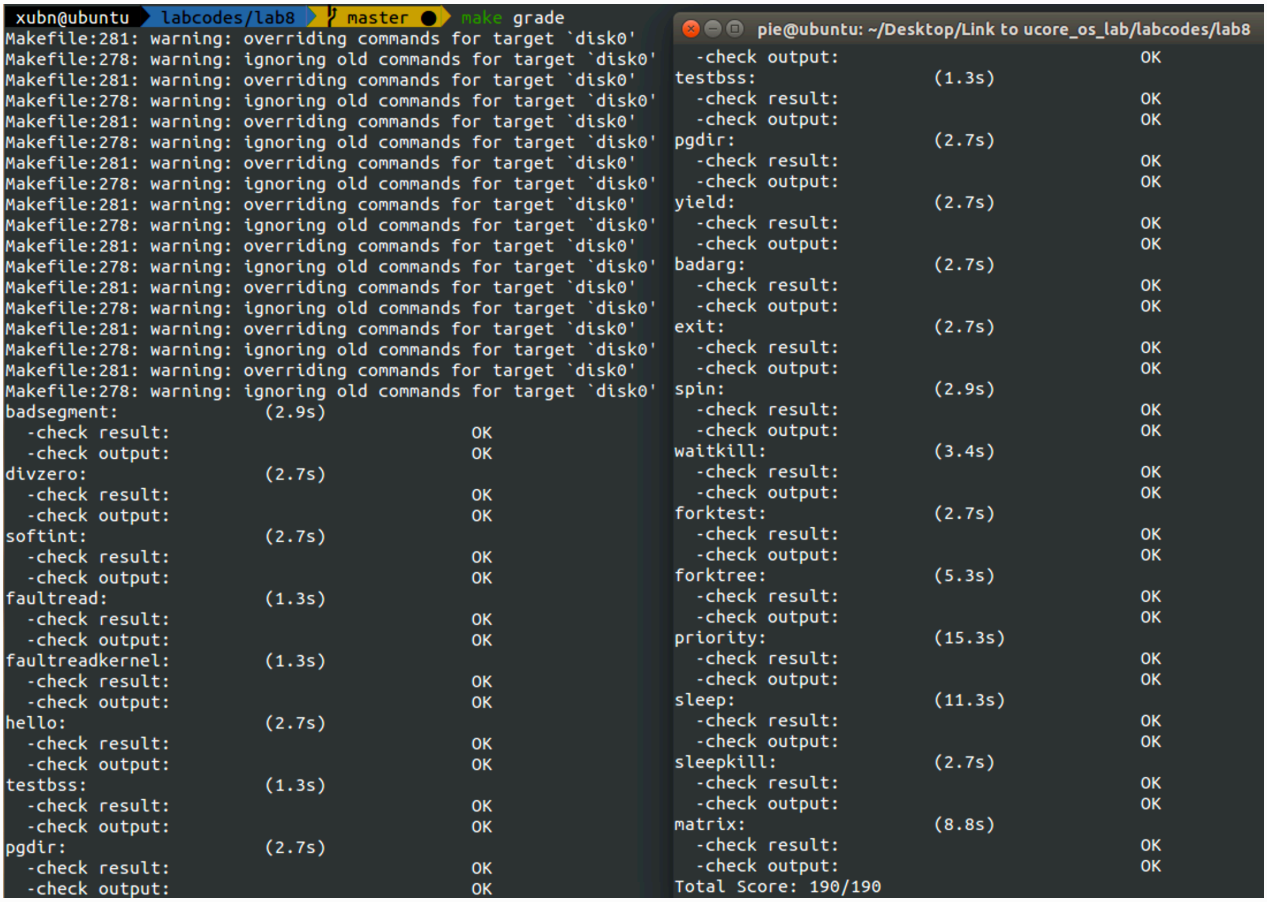
以直接完成删除。

而软链接则可以通过直接拷贝文件对应的索引节点 inode 中的数据信息进行实现，与创建普通文件无异地进行 inode 的创建，维护标志来表示软链接类型，删除软链接类型的时候，直接对其存储结构 inode 进行删除即可。

LAB 8 Test

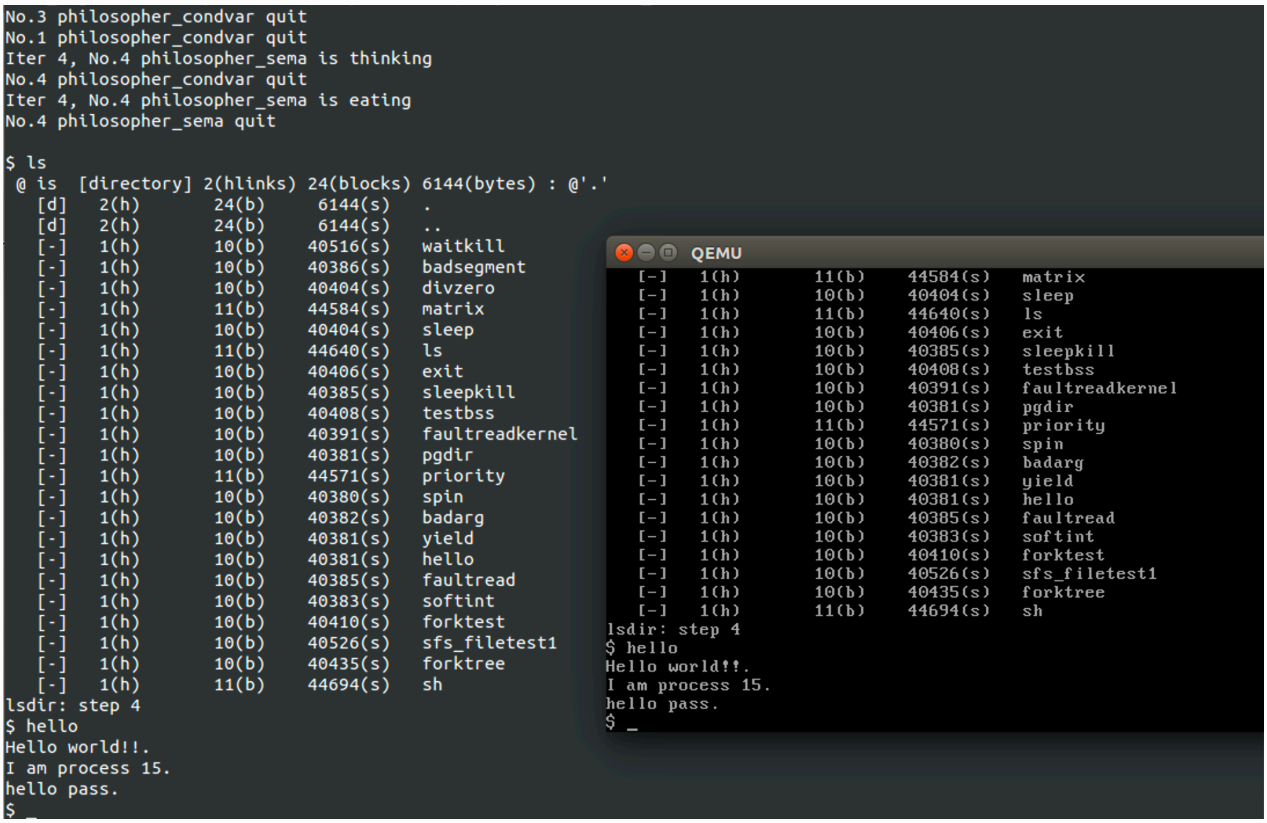
在完成上述练习的编码过程和问题回答之后，在实验环境下对完成的代码进行测试。

`make grade` 命令进行检查：



可以正常通过所有测试，LAB 8 练习中需要实现的功能基本正确实现。

`make qemu` 运行 ucore 环境进行测试，测试之前需要在 /kern/schedule/sched.h 中将时间片的最大值修改得大一些，方便后续测试，这里将 MAX_TIME_SLICE 的宏定义值改为 100：



`make qemu` 可以正常执行，在哲学家问题流程完成后，回车进入 shell 街面，可以通过 ls, hello 等命令进行文件相关操作。LAB 8 练习中需要实现的功能实现，与理论预期一致。

至此 LAB 8 完成。

多级反馈队列调度算法

> 在上述实验的基础上，修改 ucore 调度器为多级反馈队列调度算法，队列共设6个优先级，最高级的时间片为 q，并且每降低 1 级，其时间片为上一级时间片 * 2；

根据理论课中的学习和原先的初步了解，多级反馈队列调度算法的基本实现过程如下：

- 设置多个调度队列空间，即为"多级队列"的基本结构；
- 为每个队列设置不同的调度优先级，以及每个队列中进程被调度时可以占用 CPU 的时间片的数量，优先级越低，可以占用的时间片越多，在题目要求的实现中，每降低一个优先级，时间片增加为上一级的两倍；
- 每个进程子啊第一次就绪的时候，会进入最高级的队列进行等待；
- 调度过程中，优先调度最高级别队列中的进程，当最高级别的队列为空，则进入下一级别的队列选择进程进行调度，以此类推；
- 如果当前进程可以占用的时间片用完了，而其还没有完成执行，则进行相应的老化操作：降低其优先级，增加其下次调度的可用时间片，将其推入下一优先级别的队列进行等待，继续按照前述规则进行逐个优先级的调度；
- 对每个进程应用以上规则进行调度，直至其完成执行或者已经降到最低一级——在题目要求中为第六级，不再进行降级，若还没执行完则在最低一级队列循环地调度和等待。

在 LAB 6 中我们进行过调度算法的更改和替换，操作比较简单，只需要创建一个新的调度算法源文件，比如此处在 /kern/schedule/ 目录下创建一个 modified_sched_multi.c 文件，仿照原先的 default_sched.c 或是之前实现的 default_sched_stride.c 进行接口函数编写，将包装有一套函数的结构体全局化，暴露给上层调用即可，为了尽可能少地进行改动，暴露的结构体名称不变，然后将原来调度算法的接口包装过程注释掉使其失效，就可以完成调度算法的替换了。

另外还需要以下细小变动：

- 对 sched.h 中定义的 run_queue 结构体进行一些添加，使其能支持多级队列的结构；

```
#define MULTI_NUM 6
struct run_queue {
    // ……
    // For multi feedback queue
    list_entry_t multi_list[MULTI_NUM];
};
```

- 对 proc.h 中定义的 proc_struct 结构体进行一些添加，使其能够记录保存该进程所处的调度优先级别；

```
struct proc_struct {
    // ……
    int multi_level; // FOR multi feedback queue
};
```

主要实现算法的文件 modified_sched_multi.c 中代码如下：

```
#include <defs.h>
#include <list.h>
#include <proc.h>
#include <assert.h>
#include <default_sched.h>

static void
multi_init(struct run_queue *rq) {
    int i;
    for (i = 0; i < MULTI_NUM; ++i) { // MULTI_NUM 为定义在 sched.h 的宏定义，为 6
        list_init(&(rq->multi_list[i])); // 初始化六个优先调度队列
    }
    rq->proc_num = 0;
}

static void // 入队操作需要维护进程的级别和时间片长度
multi_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    int level = proc->multi_level;
    if (proc->time_slice == 0 && level < (MULTI_NUM-1))
        level ++;
    proc->multi_level = level;
    list_add_before(&(rq->multi_list[level]), &(proc->run_link));
```

```
23      if (proc->time_slice == 0 || proc->time_slice > (rq->max_time_slice << level))
24          proc->time_slice = (rq->max_time_slice << level);
25      proc->rq = rq;
26      rq->proc_num ++;
27  }
28
29  static void // 出队操作与原先实现的调度算法无异
30  multi_dequeue(struct run_queue *rq, struct proc_struct *proc) {
31      assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
32      list_del_init(&(proc->run_link));
33      rq->proc_num --;
34  }
35
36  static struct proc_struct * // 从最高级的队列向下选择
37  multi_pick_next(struct run_queue *rq) {
38      int i;
39      for (i = 0; i < MULTI_NUM; i++) {
40          if (!list_empty(&(rq->multi_list[i]))) {
41              return le2proc(list_next(&(rq->multi_list[i])), run_link);
42          }
43      }
44      return NULL;
45  }
46
47  static void // 与原算法无异
48  multi_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
49      if (proc->time_slice > 0) {
50          proc->time_slice --;
51      }
52      if (proc->time_slice == 0) {
53          proc->need_resched = 1;
54      }
55  }
56
57  struct sched_class default_sched_class = { // 进行包装，将函数对应填入并写好名字即可
58      .name = "multi_feedback_queue_scheduler",
59      .init = multi_init,
60      .enqueue = multi_enqueue,
61      .dequeue = multi_dequeue,
62      .pick_next = multi_pick_next,
63      .proc_tick = multi_proc_tick,
64  };
```

创建一个新的目录，保留其他部分源代码，只对上述提到的文件进行修改，然后在命令行终端运行 `make qemu` 看到调度算法可以正常工作，实验要求基本完成。

实验总结

分析与区别

- LAB8 练习 1 实现与答案基本一致。原先自己的代码实现中忽视了每次读写操作后对相关参量的更新，DEBUG 过程中改进，参考了答案。
- LAB8 练习 2 实现与答案基本一致。根据注释提示的 api 可以完成编写，后在对照答案的过程中规范了一些变量和跳转位置的命名。

重要知识点

- 文件管理系统；
- 系统调用的使用；
- 宏定义实现接口的结构；
- 进程空间、虚拟空间的分配和管理；
- ELF 文件相关知识；
- 调度算法原理和在 ucore 中的具体实现；

补充知识点

- 文件系统具体 mount 挂载的过程；
- 更多调度算法的实现；