# 实验项目：Ucore综合实验

姓名：**张伟焜** 学号：**17343155** 邮箱：[zhangwk8@mail2.sysu.edu.cn](mailto:zhangwk8@mail2.sysu.edu.cn)

院系：数据科学与计算机学院 专业：17级软件工程 指导教师：张永东

**【实验题目】**

Ucore综合实验

**【实验目的】**

考察对操作系统的文件系统的设计实现了解；

考察操作系统进程调度算法的实现。

考察操作系统内存管理的虚存技术的掌握（选做，加分题）。

**【实验要求】**

1. 在前面ucore实验lab1-lab7的基础上，完成ucore文件系统(参见ucore\_os\_docs.pdf中的lab8及相关视频)；

2. 在上述实验的基础上，修改ucore调度器为采用多级反馈队列调度算法的，队列共设6个优先级（6个队列），最高级的时间片为q(使用原RR算法中的时间片），并且每降低1级，其时间片为上一级时间片乘2（参见理论课）；

3. （选做，加分题）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：

对每一用exec新建进程分配3帧物理页面；

当需要页面置换时，选择最近一段时间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。

在一段时间（如1000个时间片）后将所有进程缺页次数清零，然后重新计数。

**【实验方案】**

实验环境：老师提供的虚拟机（Virtual box），无特殊硬件要求

实验思路：根据实验指导，先了解理论知识，再进行实验

**【实验过程】**

**------------------------------实验内容一-----------------------------**

**练习0：填写已有实验。**

使用meld软件将ucore启动实验的代码导入。

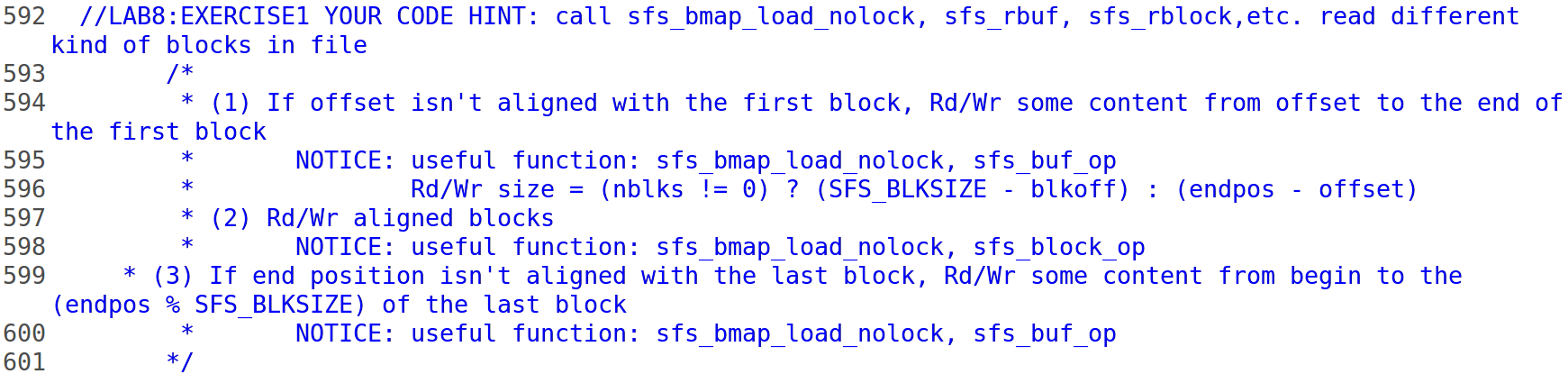
注意要点击标星文件进行对比，将之前实验完成的函数复制过来，不要将整个文件进行覆盖。之前实验修改的内容主要在kdebug.c trap.c pmm.c default\_pmm.c vmm.c swap\_fifo.c proc.c等。

**练习1：完成读文件操作的实现。**

*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。*

首先在sfs\_inode.c中查看注释sfs\_io\_nolock读文件中数据的实现代码。

查看该函数的注释：



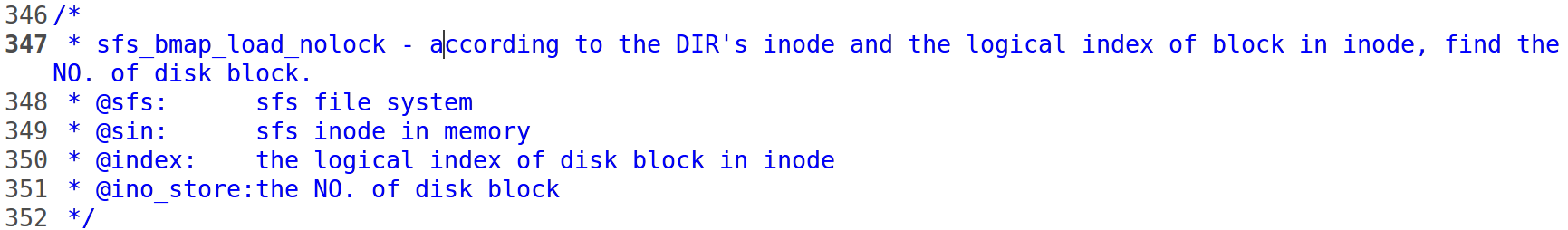
根据注释及相关代码得知，sfs\_io\_nolock函数的功能是从指定偏移量，对指定的文件进行指定长度的读或者写操作。

在Ucore中，用户调用read或者write函数，进一步触发系统调用sys\_read和sys\_write进入操作系统。最终调用到sfs\_io\_nolock函数。

在sfs\_io\_nolock中，首先，要判断被需要读/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，如果不是，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。

（相关函数为sfs\_bmap\_load\_nolock 和 sfs\_buf\_op。）

sfs\_bmap\_load\_nolock：



sfs\_bmap\_load\_nolock通过目录索引节点以及节点中的逻辑索引定位到磁盘上数据块的编号。

sfs\_buf\_op：

int(\*sfs\_buf\_op)(struct sfs\_fs \*sfs, void \*buf, size\_t len, uint32\_t blkno, off\_t offset);

这是一个函数指针，对应buf操作。Buf操作向磁盘指定位置写入一定长度的内容。

为了实现文件操作，我们接下来需要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。还要获取当前这个数据块对应到的磁盘上的数据块的编号；将数据写入到磁盘中；维护已经读写成功的数据长度信息。

//读取第一部分数据

if ((blkoff = offset % SFS\_BLKSIZE) != 0) {

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 (nblks == 0) {

goto out;

}

buf += size, blkno++, nblks--;

}

接下来读取第二部分数据。该部分数据都是一块一块的，大小为size。调用sfs\_bmap\_load\_nolock和sfs\_buf\_op函数对磁盘进行操作。

//读取第二部分数据。将其分为大小为size的块，然后一次读一块直至读完。

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--;

}

最后，读取第三部分数据。该部分的数据可能没有占满一整块。

//读取第三部分的数据

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;

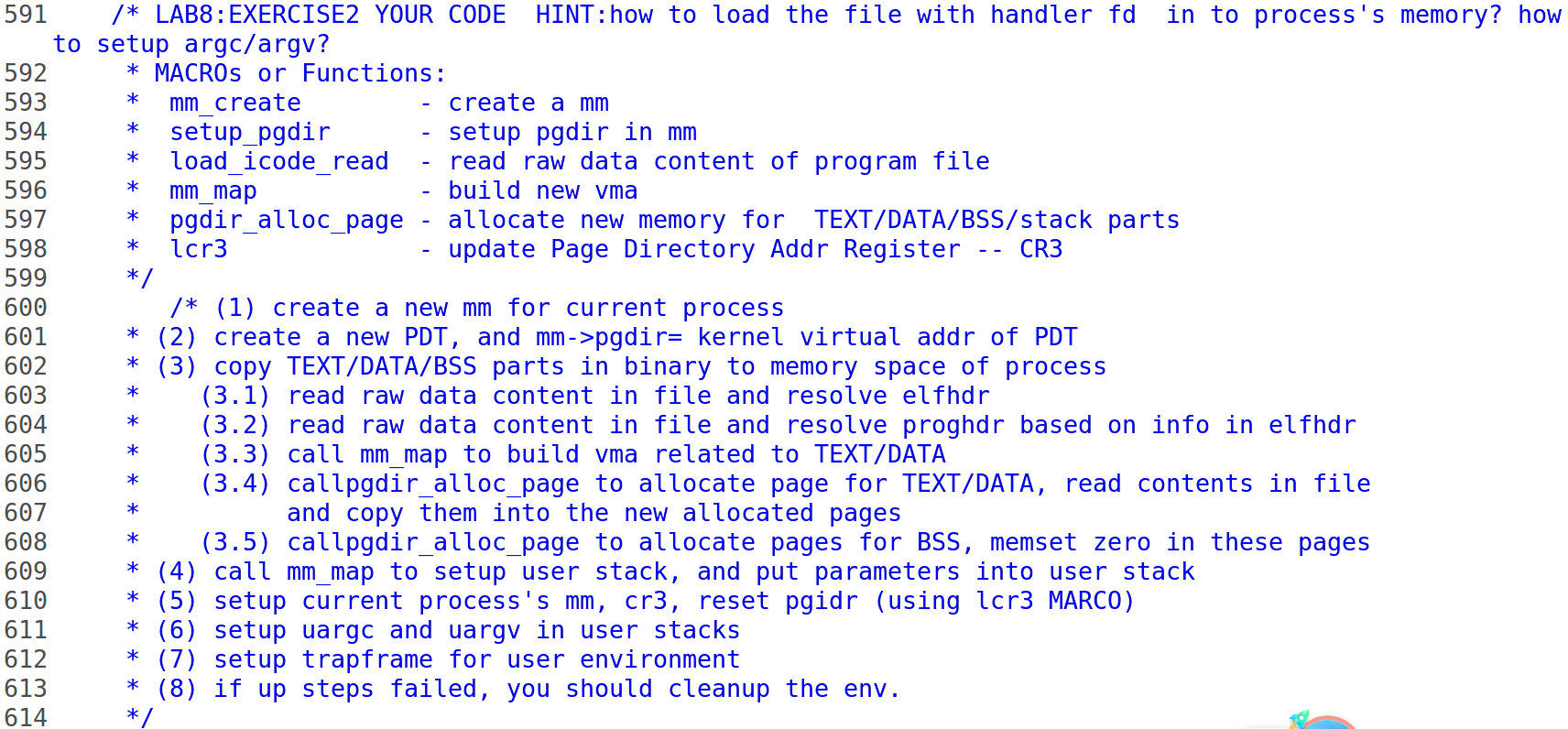
}

**练习2：完成基于文件系统的执行程序机制的实现。**

*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的其他放置在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。*

之前lab中，用户文件在启动操作系统的时候与操作系统文件一起导入了内存。在lab8，我们要用load\_icode\_read函数从新磁盘读取文件（即系统调用）。

查看load\_icode的代码注释：



实现load\_icode函数：

static int

load\_icode(int fd, int argc, char \*\*kargv) {

assert(argc >= 0 && argc <= EXEC\_MAX\_ARG\_NUM);

//(1)建立内存管理器

if (current->mm != NULL) { //要求当前内存管理器为空

panic("load\_icode: current->mm must be empty.\n");

}

int ret = -E\_NO\_MEM; // E\_NO\_MEM代表因为存储设备产生的请求错误

struct mm\_struct \*mm; //建立内存管理器

if ((mm = mm\_create()) == NULL) {

goto bad\_mm;

}

//(2)建立页目录

if (setup\_pgdir(mm) != 0) {

goto bad\_pgdir\_cleanup\_mm;

}

struct Page \*page; //页表的建立

//(3)从文件加载程序到内存

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++) {

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;

}

vm\_flags = 0, perm = PTE\_U; //建立虚拟地址与物理地址之间的映射

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;

//复制数据段和代码段

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;

}

//建立BSS段

end = ph->p\_va + ph->p\_memsz;

if (start < la) {

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);

//(4)建立相应的虚拟内存映射表

vm\_flags = VM\_READ | VM\_WRITE | VM\_STACK;

if ((ret = mm\_map(mm, USTACKTOP - USTACKSIZE, 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);

//(5)设置用户栈

mm\_count\_inc(mm);

current->mm = mm;

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

lcr3(PADDR(mm->pgdir));

//(6)处理用户栈中传入的参数，其中argc对应参数个数，uargv[]对应参数的具体内容的地址

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)设置进程的中断帧

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)错误处理部分

out:

return ret;

bad\_cleanup\_mmap:

exit\_mmap(mm);

bad\_elf\_cleanup\_pgdir:

put\_pgdir(mm);

bad\_pgdir\_cleanup\_mm:

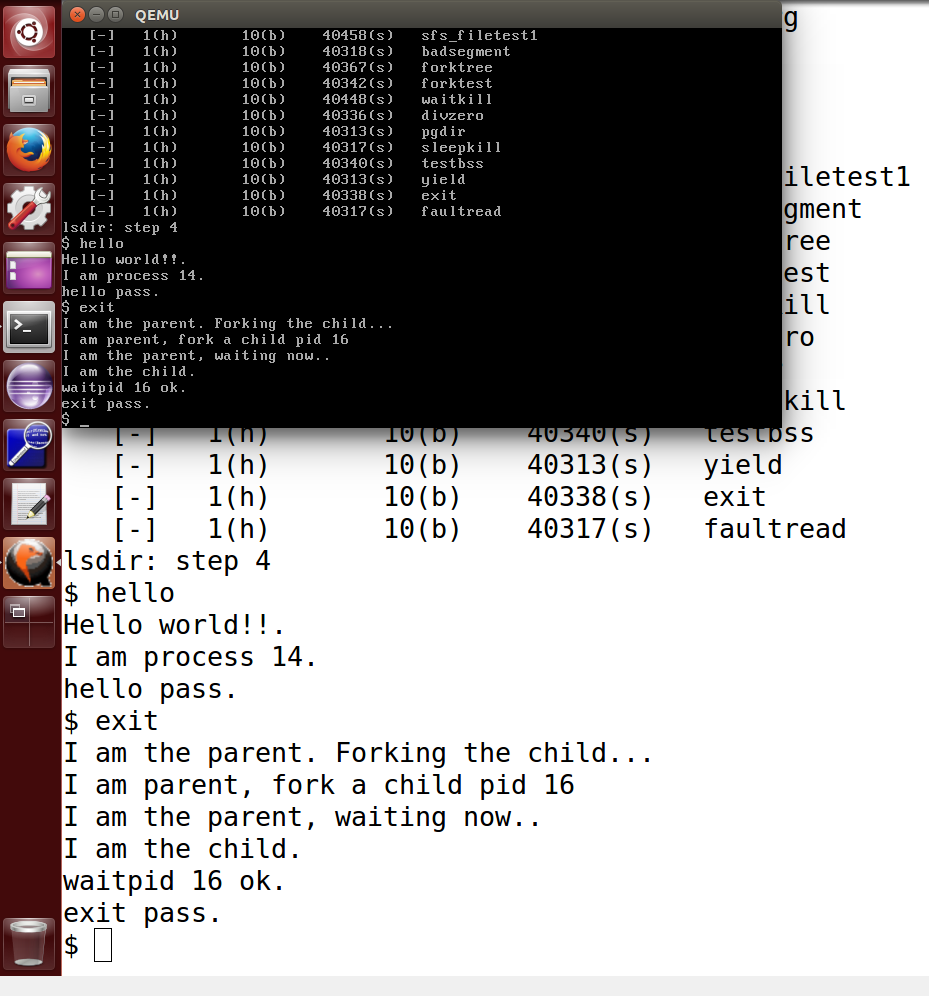
mm\_destroy(mm);

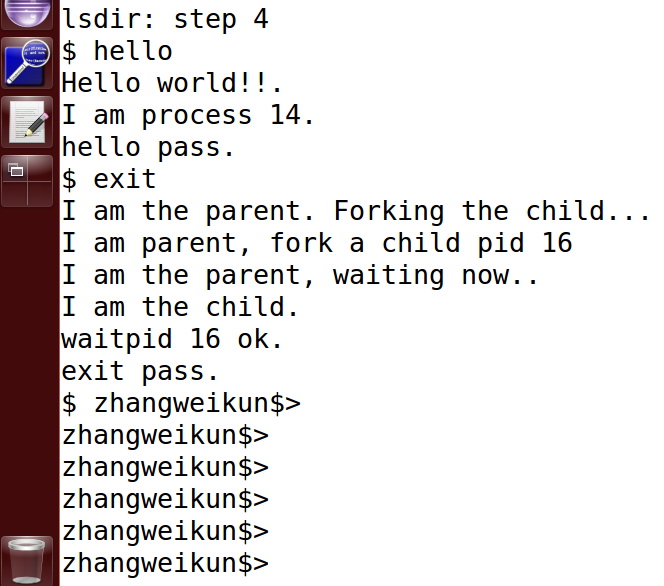
bad\_mm:

goto out;

}

**执行make qemu。(执行ls 并运行hello 和 exit)**





**------------------------------实验内容二-----------------------------**

*在上述实验的基础上，修改ucore调度器为采用多级反馈队列调度算法的，队列共设6个优先级（6个队列），最高级的时间片为q(使用原RR算法中的时间片），并且每降低1级，其时间片为上一级时间片乘2（参见理论课）；*

这里回顾实验七（lab6）中的“多级反馈队列调度算法”的设计问题。

**首先给出多级反馈队列调度算法的描述。**

a) 进程在进入待调度的队列等待时，首先进入优先级最高的Q1等待。

b) 首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q1,Q2,Q3三个队列，当且仅当在Q1中没有进程等待时才去调度Q2，同理，只有Q1,Q2都为空时才会去调度Q3。

c) 对于同一个队列中的各个进程，按照FCFS分配时间片调度。比如Q1队列的时间片为N，那么Q1中的作业在经历了N个时间片后若还没有完成，则进入Q2队列等待，若Q2的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。

d) 在最后一个队列QN中的各个进程，按照时间片轮转分配时间片调度。

e) 在低优先级的队列中的进程在运行时，又有新到达的作业，此时须立即把正在运行的进程放回当前队列的队尾，然后把处理机分给高优先级进程。换而言之，任何时刻，只有当第1~i-1队列全部为空时，才会去执行第i队列的进程（抢占式）。特别说明，当再度运行到当前队列的该进程时，仅分配上次还未完成的时间片，不再分配该队列对应的完整时间片。

**接下来给出具体实现。**

在proc\_struct中设置N个多级反馈队列入口。队列编号越大优先级越低，优先级越低的队列上进程的时间片越大，具体可以设置为上一个队列的两倍。

为了记录进程所在的队列，我们在PCB中增加条目来进行记录。

首先，对所有优先级队列进行初始化。将进程加入到就绪进程集合时，观察该进程剩余的时间片，如果为0，就降一级；如果不为0，则不降级。

对同一个优先级的队列内的进程使用时间片轮转算法。

选择下一个执行进程时，优先看较高优先级的队列中是否存在任务，如果不存在才在较低优先级的队列中寻找进程去执行。

从就绪进程集合中删除某一个进程也要在对应队列中删除。

处理时间中断的函数仍然不需要改变。（与RR相同）

综上，该调度算法主要修改init()、enqueue()、dequeue()、pick\_next()、proc\_tick() 函数。

0）准备工作。首先将proc.c 和proc.h中的”stride”全部替换为”MFQ”

1）实现多级反馈队列，需要把之前的运行队列从一个改为6个。

struct run\_queue {

list\_entry\_t run\_list[6];

unsigned int proc\_num;

int max\_time\_slice;

// For LAB6 ONLY

skew\_heap\_entry\_t \*lab6\_run\_pool;

};

2）init() 函数。该函数初始化6个队列，此时进程数为0。

static void

MFQ\_init(struct run\_queue \*rq) {

int i = 0;

for (i = 0; i < 6; i++)

list\_init(&(rq->run\_list[i]));

rq->proc\_num = 0;

}

3）enqueue()函数。将进程放入前，首先判断进程的时间片是否用完，如果用完，则优先级降低+1，时间片x2；否则不进行操作，把该进程加入对应优先级队列。

static void

MFQ\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {

assert(list\_empty(&(proc->run\_link)));

if (proc->time\_slice == 0 && proc->lab6\_priority != 5) {

++(proc->lab6\_priority);

}

list\_add\_before(&(rq->run\_list[proc->lab6\_priority]), &(proc->run\_link));

proc->time\_slice = (rq->max\_time\_slice << proc->lab6\_priority);

proc->rq = rq;

rq->proc\_num++;

}

4）dequeue()函数。

static void

MFQ\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc) {

assert(!list\_empty(&(proc->run\_link)) && proc->rq == rq);

list\_del\_init(&(proc->run\_link));

rq->proc\_num--;

}

5）pick\_next()函数。该函数选出下一个运行的进程。从优先级最高的开始，如果高优先级的队列是空的，就选择低优先级的进程运行。（饥饿处理）优先级越高的进程被选中的概率越大，低优先级的进程有可能被调度。

static struct proc\_struct \*

MFQ\_pick\_next(struct run\_queue \*rq) {

int p = rand() % (32 + 16 + 8 + 4 + 2 + 1);

int priority;

if (p >= 0 && p < 32) {

priority = 0;

}

else if (p >= 32 && p < 48) {

priority = 1;

}

else if (p >= 48 && p < 56) {

priority = 2;

}

else if (p >= 56 && p < 60) {

priority = 3;

}

else if (p >= 60 && p < 62) {

priority = 4;

}

else if (p >= 62 && p < 63) {

priority = 5;

}

list\_entry\_t \*le = list\_next(&(rq->run\_list[priority]));

if (le != &(rq->run\_list[priority])) {

return le2proc(le, run\_link);

}

else {

int i = 0;

for (i = 0; i < 5; ++i) {

le = list\_next(&(rq->run\_list[i]));

if (le != &(rq->run\_list[i])) return le2proc(le, run\_link);

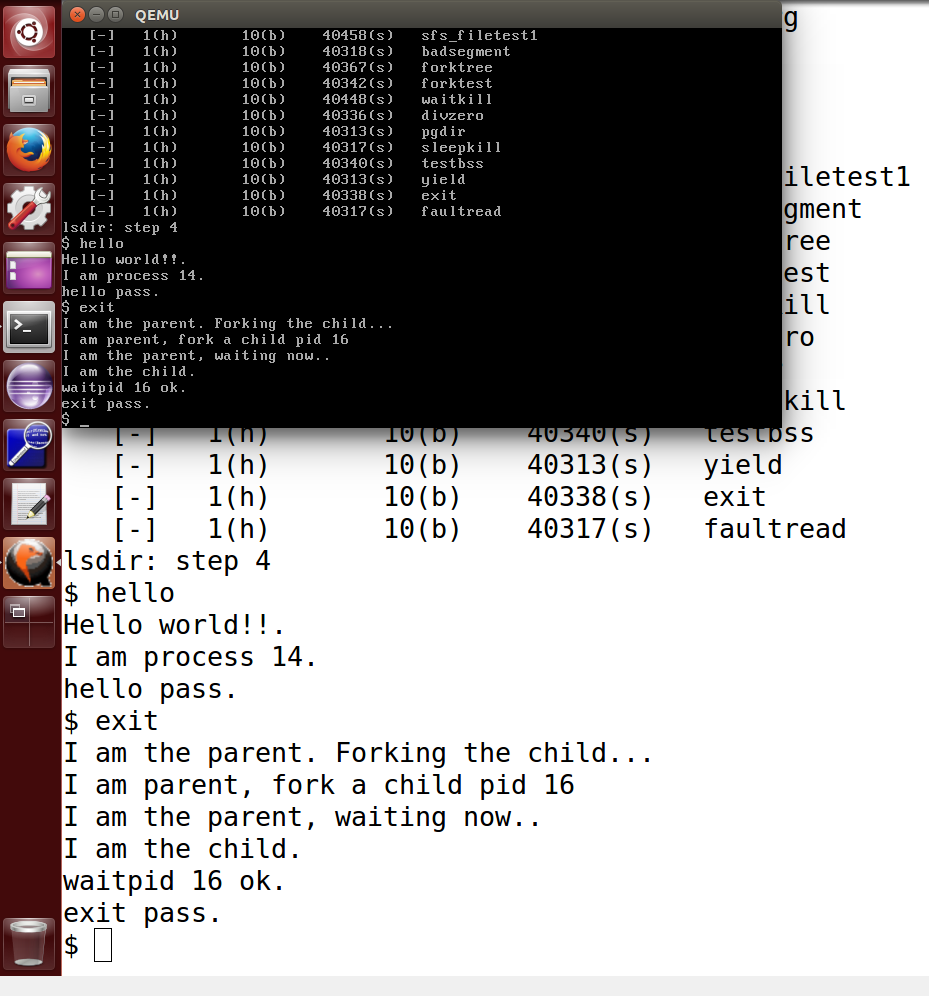
}

}

return NULL;

}

**执行make qemu。(执行ls 并运行hello 和 exit)**



**实验运行成功**

**【实验总结】**

**完成实验后，请分析ucore\_lab中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别。**

文件系统部分由于结合注释进行编码，所以与答案近似，功能完整。

多级反馈队列调度算法实验中，没有参考答案，为独立完成。

**列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）**

1.文件系统

2.多级反馈队列调度方案的设计

对应了OS原理中的：

1.文件系统的实现

2.进程的调度算法

3.多级反馈队列调度方案的原理与概念

**理论知识是基础；**

**实验知识是理论知识的实际应用与实践。**

**列出你认为OS原理中很重要，但在实验中没有对应上的知识点**

1.原子事务中的基于日志的恢复

2.没有分析管程

3.个人没有完成选作部分的页面置换算法

**心得体会**

本次实验涉及文件系统的实现及相关操作和进程调度算法的问题，文件系统比较复杂，不仅涉及的函数较多，还有很多层的调用，实现过程中也遇到了不少的困难。

多级队列调度算法的实现过程中，我结合了之前调度实验。首先复习理论课上的相关知识，然后结合实验7的思考进行设计，最后完成编码。

在调度算法实现过程中，主要遇到了进程饥饿问题。在理论课上学习到的解决方法是设置老化时间，但在实现过程中，需要在每个时钟周期对队列进行遍历。在舍友的指导下学会了另一种解决饥饿的方法：优先级越高的进程被选中的概率越大，低优先级的进程有可能被调度，不需要修改时钟操作，只需修改pick\_next函数。

比较遗憾的是，临近期末，时间比较紧张，没能抽时间完成第三部分。之后有时间一定会去试试第三部分，挑战下自己。

最后，本学期的这门实验课就要结束了。我想在此感谢老师的认真指导，感谢TA们的作业批改，你们辛苦了！

**【参考文献】**

*《**操作系统实验指导(清华大学)陈渝、向勇编著》*