

# 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：

```
592 //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf, sfs_rblock,etc. read different
    kind of blocks in file
593 /*
594  * (1) If offset isn't aligned with the first block, Rd/Wr some content from offset to the end of
    the first block
595  *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
596  *      Rd/Wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset)
597  * (2) Rd/Wr aligned blocks
598  *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op
599  * (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
600  *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
601  */
```

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:

```
346 /*
347 * sfs_bmap_load_nolock - according to the DIR's inode and the logical index of block in inode, find the
    NO. of disk block.
348 * @sfs:      sfs file system
349 * @sin:      sfs inode in memory
350 * @index:    the logical index of disk block in inode
351 * @ino_store: the NO. of disk block
352 */
```

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

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：

```
591  /* LAB8:EXERCISE2 YOUR CODE HINT:how to load the file with handler fd in to process's memory? how
    to setup argc/argv?
592  * MACROs or Functions:
593  * mm_create      - create a mm
594  * setup_pgdir    - setup pgdir in mm
595  * load_icode_read - read raw data content of program file
596  * mm_map         - build new vma
597  * pgdir_alloc_page - allocate new memory for TEXT/DATA/BSS/stack parts
598  * lcr3           - update Page Directory Addr Register -- CR3
599  */
600  /* (1) create a new mm for current process
601  * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
602  * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
603  * (3.1) read raw data content in file and resolve elfhdr
604  * (3.2) read raw data content in file and resolve proghdr based on info in elfhdr
605  * (3.3) call mm_map to build vma related to TEXT/DATA
606  * (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file
607  * and copy them into the new allocated pages
608  * (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
609  * (4) call mm_map to setup user stack, and put parameters into user stack
610  * (5) setup current process's mm, cr3, reset pgdir (using lcr3 MARCO)
611  * (6) setup uargc and uargv in user stacks
612  * (7) setup trapframe for user environment
613  * (8) if up steps failed, you should cleanup the env.
614  */
```

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_INVALID 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_INVALID 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 += strlen(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 += strlen(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;
```

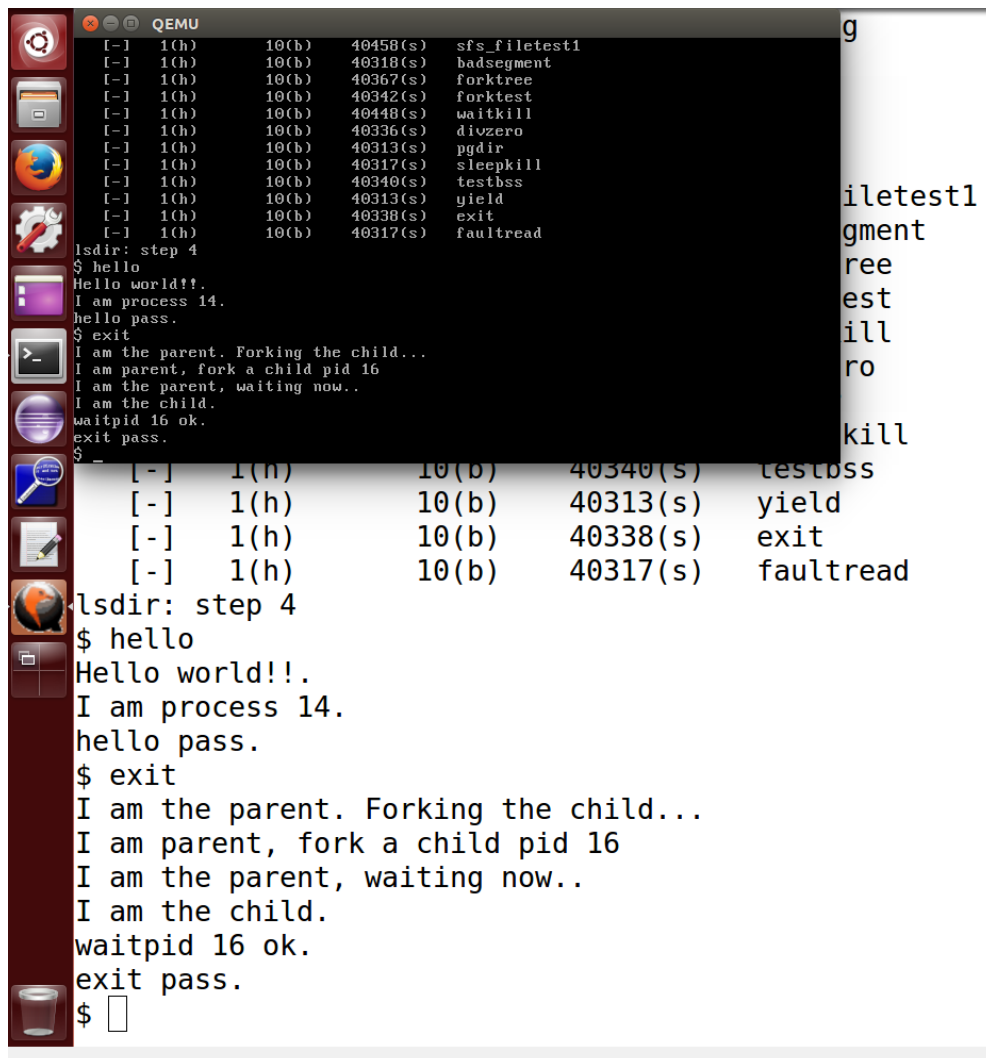


```

// (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)




```

QEMU
[-] 1(h) 10(b) 40458(s) sfs_filetest1
[-] 1(h) 10(b) 40318(s) badsegment
[-] 1(h) 10(b) 40367(s) forktree
[-] 1(h) 10(b) 40342(s) forkttest
[-] 1(h) 10(b) 40448(s) waitkill
[-] 1(h) 10(b) 40336(s) divzero
[-] 1(h) 10(b) 40313(s) pgdir
[-] 1(h) 10(b) 40317(s) sleepkill
[-] 1(h) 10(b) 40340(s) testbss
[-] 1(h) 10(b) 40313(s) yield
[-] 1(h) 10(b) 40338(s) exit
[-] 1(h) 10(b) 40317(s) faultread

lsdir: step 4
$ hello
Hello world!!.
I am process 14.
hello pass.
$ exit
I am the parent. Forking the child...
I am parent, fork a child pid 16
I am the parent, waiting now..
I am the child.
waitpid 16 ok.
exit pass.
$
[-] 1(h) 10(b) 40340(s) testbss
[-] 1(h) 10(b) 40313(s) yield
[-] 1(h) 10(b) 40338(s) exit
[-] 1(h) 10(b) 40317(s) faultread

lsdir: step 4
$ hello
Hello world!!.
I am process 14.
hello pass.
$ exit
I am the parent. Forking the child...
I am parent, fork a child pid 16
I am the parent, waiting now..
I am the child.
waitpid 16 ok.
exit pass.
$

```



```
lsdir: step 4
$ hello
Hello world!!.
I am process 14.
hello pass.
$ exit
I am the parent. Forking the child...
I am parent, fork a child pid 16
I am the parent, waiting now..
I am the child.
waitpid 16 ok.
exit pass.
$ zhangweikun$>
zhangweikun$>
zhangweikun$>
zhangweikun$>
zhangweikun$>
zhangweikun$>
```

## -----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。

- 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。
- 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。
- 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  $\times 2$ ；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--;
}

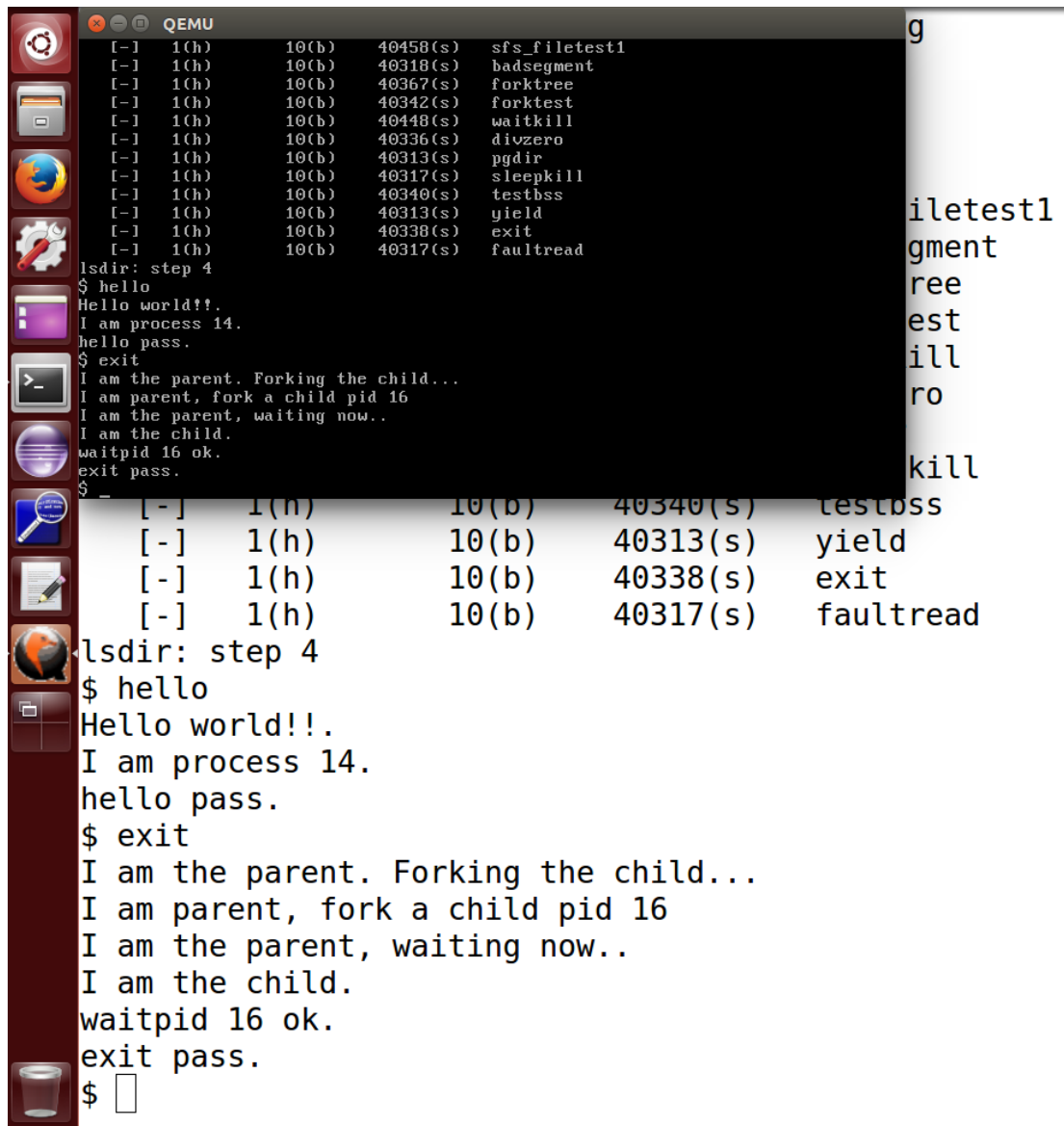
```

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)



```
QEMU
[ - ] 1(h) 10(b) 40458(s) sfs_filetest1
[ - ] 1(h) 10(b) 40318(s) badsegment
[ - ] 1(h) 10(b) 40367(s) forktree
[ - ] 1(h) 10(b) 40342(s) forktest
[ - ] 1(h) 10(b) 40448(s) waitkill
[ - ] 1(h) 10(b) 40336(s) divzero
[ - ] 1(h) 10(b) 40313(s) pgdir
[ - ] 1(h) 10(b) 40317(s) sleepkill
[ - ] 1(h) 10(b) 40340(s) testbss
[ - ] 1(h) 10(b) 40313(s) yield
[ - ] 1(h) 10(b) 40338(s) exit
[ - ] 1(h) 10(b) 40317(s) faultread

lsdir: step 4
$ hello
Hello world!!
I am process 14.
hello pass.
$ exit
I am the parent. Forking the child...
I am parent, fork a child pid 16
I am the parent, waiting now..
I am the child.
waitpid 16 ok.
exit pass.
$ -
[ - ] 1(h) 10(b) 40340(s) testbss
[ - ] 1(h) 10(b) 40313(s) yield
[ - ] 1(h) 10(b) 40338(s) exit
[ - ] 1(h) 10(b) 40317(s) faultread

lsdir: step 4
$ hello
Hello world!!
I am process 14.
hello pass.
$ exit
I am the parent. Forking the child...
I am parent, fork a child pid 16
I am the parent, waiting now..
I am the child.
waitpid 16 ok.
exit pass.
$
```

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》