**Lab 5**

**练习1: 加载应用程序并执行（需要编码）**

**do\_execv函数调用load\_icode（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好proc\_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。**

**（1）请在实验报告中简要说明你的设计实现过程。**

**首先我们了解一下两个函数的功能：**

do\_execv函数的主要功能和实现： 完成用户进程的创建工作。为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。由于此处的initproc是内核线程，所以mm为NULL，整个处理都不会做。加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。这里涉及到读ELF格式的文件，申请内存空间，建立用户态虚存空间，加载应用程序执行码等。load\_icode函数完成了整个复杂的工作。

//主要目的在于清理原来进程的内存空间，为新进程执行准备好空间和资源

int do\_execve(const char \*name, size\_t len, unsigned char \*binary, size\_t size)

{

struct mm\_struct \*mm = current->mm;

if (!user\_mem\_check(mm, (uintptr\_t)name, len, 0)) {

return -E\_INVAL;

}

if (len > PROC\_NAME\_LEN) {

len = PROC\_NAME\_LEN;

}

char local\_name[PROC\_NAME\_LEN + 1];

memset(local\_name, 0, sizeof(local\_name));

memcpy(local\_name, name, len);

//如果mm不为NULL，则不执行该过程

if (mm != NULL)

{

//将cr3页表基址指向boot\_cr3,即内核页表

lcr3(boot\_cr3);

if (mm\_count\_dec(mm) == 0)

{

//下面三步实现将进程的内存管理区域清空

exit\_mmap(mm);

put\_pgdir(mm);

mm\_destroy(mm);

}

current->mm = NULL;

}

int ret;

//填入新的内容，load\_icode会将执行程序加载，建立新的内存映射关系，从而完成新的执行

if ((ret = load\_icode(binary, size)) != 0) {

goto execve\_exit;

}

//给进程新的名字

set\_proc\_name(current, local\_name);

return 0;

execve\_exit:

do\_exit(ret);

panic("already exit: %e.\n", ret);

}

至此完成了用户环境的搭建。此时initproc将按产生系统调用的函数调用路径原路返回，执行中断返回指令iret后，将切换到用户进程程序的第一条语句位置\_start处开始执行。

在这里需要补充的是proc\_struct结构中tf结构体变量的设置，从而实现tf从内核态切换到用户态然后执行程序。

如何初始化：

1、由于最终是在用户态下运行的，所以需要将段寄存器初始化为用户态的代码段、数据段、堆栈段；

2、esp应当指向先前的步骤中创建的用户栈的栈顶；

3、eip应当指向ELF可执行文件加载到内存之后的入口处；

4、eflags中应当初始化为中断使能，注意eflags的第1位是恒为1的；

5、设置ret为0，表示正常返回；

代码如下：

tf->tf\_cs = USER\_CS;

tf->tf\_ds = tf->tf\_es = tf->tf\_ss = USER\_DS;

tf->tf\_esp = USTACKTOP;//0xB0000000

tf->tf\_eip = elf->e\_entry;

tf->tf\_eflags = FL\_IF;//FL\_IF为中断打开状态

ret = 0;

**（2）请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。**

1、调用schedule函数，调度器占用了CPU的资源之后，用户态进程调用了exec系统调用，从而转入到了系统调用的处理例程；

2、之后进行正常的中断处理例程，然后控制权转移到了syscall.c中的syscall函数，然后根据系统调用号转移给了sys\_exec函数，在该函数中调用了do\_execve函数来完成指定应用程序的加载；

3、在do\_execve中进行了若干设置，包括推出当前进程的页表，换用内核的PDT，调用load\_icode函数完成对整个用户线程内存空间的初始化，包括堆栈的设置以及将ELF可执行文件的加载，之后通过current->tf指针修改了当前系统调用的trapframe，使得最终中断返回的时候能够切换到用户态，并且同时可以正确地将控制权转移到应用程序的入口处；

4、在完成了do\_exec函数之后，进行正常的中断返回的流程，由于中断处理例程的栈上面的eip已经被修改成了应用程序的入口处，而CS上的CPL是用户态，因此iret进行中断返回的时候会将堆栈切换到用户的栈，并且完成特权级的切换，并且跳转到要求的应用程序的入口处；

5、开始执行应用程序的第一条代码。

**练习2: 父进程复制自己的内存空间给子进程（需要编码）**

**创建子进程的函数do\_fork在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过copy\_range函数（位于kern/mm/pmm.c中）实现的，请补充copy\_range的实现，确保能够正确执行。**

**请在实验报告中简要说明如何设计实现”Copy on Write 机制“，给出概要设计，鼓励给出详细设计。**

首先，Copy on Write 是在复制一个对象的时候并不是真正的把原先的对象复制到内存的另外一个位置上，而是在新对象的内存映射表中设置一个指针，指向源对象的位置，并把那块内存的Copy-On-Write位设置为1。通俗来说一下这样做的好处：如果复制的对象只是对内容进行"读"操作，那其实不需要真正复制，这个指向源对象的指针就能完成任务，这样便节省了复制的时间并且节省了内存。但是问题在于，如果复制的对象需要对内容进行写的话，单单一个指针可能满足不了要求，因为这样对内容的修改会影响其他进程的正确执行，所以就需要将这块区域复制一下，当然不需要全部复制，只需要将需要修改的部分区域复制即可，这样做大大节约了内存并提高效率。

因为如果设置原先的内容为只可读，则在对这段内容进行写操作时候便会引发Page Fault，这时候我们便知道这段内容是需要去写的，在Page Fault中进行相应处理即可。也就是说利用Page Fault来实现权限的判断，或者说是真正复制的标志。

基于原理和之前的用户进程创建、复制、运行等机制进行分析，设计思想：

设置一个标记位，用来标记某块内存是否共享，实际上dup\_mmap函数中有对share的设置，因此首先需要将share设为1,表示可以共享。

在pmm.c中为copy\_range添加对共享页的处理，如果share为1，那么将子进程的页面映射到父进程的页面即可。由于两个进程共享一个页面之后，无论任何一个进程修改页面，都会影响另外一个页面，所以需要子进程和父进程对于这个共享页面都保持只读。

当程序尝试修改只读的内存页面的时候，将触发Page Fault中断，这时候我们可以检测出是超出权限访问导致的中断，说明进程访问了共享的页面且要进行修改，因此内核此时需要重新为进程分配页面、拷贝页面内容、建立映射关系

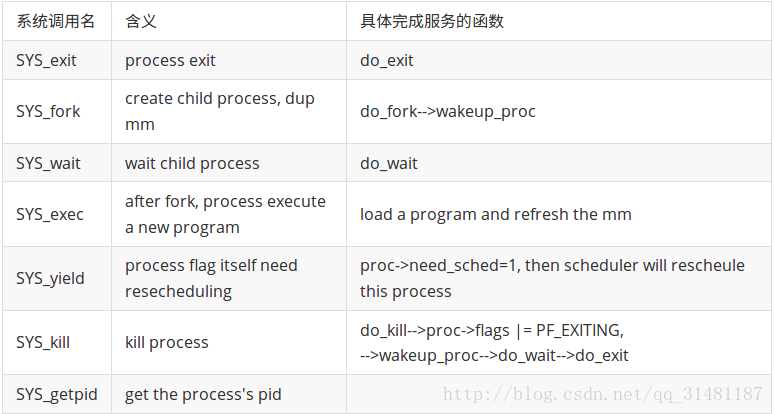
基本设计思想即为此，基于此，可以实现较为简单的"Copy on Write"机制。

int copy\_range(pde\_t \*to, pde\_t \*from, uintptr\_t start, uintptr\_t end, bool share) {  
    ......  
    ......      
    void \* kva\_src = page2kva(page);//返回父进程的内核虚拟页地址    
    void \* kva\_dst = page2kva(npage);//返回子进程的内核虚拟页地址    
    memcpy(kva\_dst, kva\_src, PGSIZE);//复制父进程到子进程    
    ret = page\_insert(to, npage, start, perm);//建立子进程页地址起始位置与物理地址的映射关系(prem是权限)    
    ......  
    ......  
}

**练习3: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现（不需要编码）**

**请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。并回答如下问题：**

**（1）请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的？**



1、fork将创建新的子线程，将子线程的状态由UNINIT态变为RUNNABLE态，不改变父进程的状态

2、exec完成用户进程的创建工作，同时使用户进程进入执行，不改变进程状态

3、wait完成子进程资源回收，如果有已经结束的子进程或者没有子进程，那么调用会立刻结束，不影响进程状态；否则，进程需要等待子进程结束，进程从RUNNIG态变为SLEEPING态。

4、exit完成对资源的回收，进程从RUNNIG态变为ZOMBIE态。

**（2）请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或**

**函数调用）。（字符方式画即可）**

**执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。（使用的是qemu-1.0.1）**

执行状态生命周期图：

创建进程 +---wait()-------RUNNING-------------+

| | ^| |

| | | | |

alloc\_page() | proc\_run() exit()

| | | | |

V | | v v

UNINIT ---wakeup\_proc()-> RUNNABLE --exit()--> ZOMBIE--父进程调用wait()-->

| ^ ^

| | |

| 子进程调用exit() exit()

| | |

| | |

+---wait()----->SLEEPING----------------+

**Lab 6**

**练习1: 使用 Round Robin 调度算法（不需要编码）**

**完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。**

**请在实验报告中完成：**

1. **请理解并分析sched\_calss中各个函数指针的用法，并接合Round Robin 调度算法描述ucore的调度执行过程**

我们来逐个函数的分析，从而了解Round Robin调度算法的原理。

struct sched\_class {

// the name of sched\_class

const char \*name;

// Init the run queue

void (\*init)(struct run\_queue \*rq);

// put the proc into runqueue, and this function must be called with rq\_lock

void (\*enqueue)(struct run\_queue \*rq, struct proc\_struct \*proc);

// get the proc out runqueue, and this function must be called with rq\_lock

void (\*dequeue)(struct run\_queue \*rq, struct proc\_struct \*proc);

// choose the next runnable task

struct proc\_struct \*(\*pick\_next)(struct run\_queue \*rq);

// dealer of the time-tick

void (\*proc\_tick)(struct run\_queue \*rq, struct proc\_struct \*proc);

/\* for SMP support in the future

\* load\_balance

\* void (\*load\_balance)(struct rq\* rq);

\* get some proc from this rq, used in load\_balance,

\* return value is the num of gotten proc

\* int (\*get\_proc)(struct rq\* rq, struct proc\* procs\_moved[]);

\*/

}；

RR算法

static void RR\_init(struct run\_queue \*rq) {

list\_init(&(rq->run\_list)); *//初始化运行队列*

rq->proc\_num = 0;

}

1.RR\_init函数，完成了对进程队列的初始化。

struct run\_queue {

list\_entry\_t run\_list; //进程队列

unsigned int proc\_num; //进程数量

int max\_time\_slice; //最大时间片长度（RR）

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

//在stride调度算法中，为了“斜堆”数据结构创建的一种特殊进程队列，本质就是进程队列。

}；

2.RR\_enqueue函数，

static void RR\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc) {

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

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

if (proc->time\_slice == 0 || proc->time\_slice > rq->max\_time\_slice) {

proc->time\_slice = rq->max\_time\_slice;

}

proc->rq = rq;

rq->proc\_num ++;

}

首先，它把进程的进程控制块指针放入到rq队列末尾，且如果进程控制块的时间片为0，则需要把它重置为max\_time\_slice。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。然后在依次调整rq和rq的进程数目加一。

3.RR\_dequeue函数

static void

RR\_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 --;

}

即简单的把就绪进程队列rq的进程控制块指针的队列元素删除，然后使就绪进程个数的proc\_num减一。

4.pick\_next函数。

static struct proc\_struct \*RR\_pick\_next(struct run\_queue \*rq) {

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

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

return le2proc(le, run\_link);

}

return NULL;

}

选取函数，即选取就绪进程队列rq中的队头队列元素，并把队列元素转换成进程控制块指针，即置为当前占用CPU的程序。

5. proc\_tick函数

static void RR\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc) {

if (proc->time\_slice > 0) {

proc->time\_slice --;

}

if (proc->time\_slice == 0) {

proc->need\_resched = 1;

}

}

观察代码，即每一次时间片到时的时候，当前执行进程的时间片time\_slice便减一。如果time\_slice降到零，则设置此进程成员变量need\_resched标识为1，这样在下一次中断来后执行trap函数时，会由于当前进程程成员变量need\_resched标识为1而执行schedule函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行。

之后是一个对象化，提供一个类的实现。

6.初始化信息

struct sched\_class default\_sched\_class = {

.name = "RR\_scheduler",

.init = RR\_init,

.enqueue = RR\_enqueue,

.dequeue = RR\_dequeue,

.pick\_next = RR\_pick\_next,

.proc\_tick = RR\_proc\_tick,

}；

初始化绑定

void sched\_init(void) {

list\_init(&timer\_list);

sched\_class = &default\_sched\_class;

rq = &\_\_rq;

rq->max\_time\_slice = MAX\_TIME\_SLICE;

sched\_class->init(rq);

cprintf("sched class: %s\n", sched\_class->name);

}

1. **请在实验报告中简要说明如何设计实现“多级反馈队列调度算法”，给出概要设计，鼓励给出详细设计**

假设进程一共有4个调度优先级，分别为0、1、2、3，其中0位最高优先级，3位最低优先级。为了支持4个不同的优先级，在运行队列中开4个队列，分别命名为rq -> run\_list[0..3]。除此之外，在proc\_struct中加入priority成员表示该进程现在所处的优先级，初始化为0。

1、MLFQ\_init：进程队列初始化，和RR算法实现一样，不同之处在于需要初始化4个队列，分别对应0、1、2、3。

2、MLFQ\_enqueue(struct run\_queue \*rq, struct proc\_struct \*proc)：判断proc进程的时间片proc -> time\_slice是否为0，如果为0，则proc -> priority += 1，否则不变。根据proc加入到对应优先级的列表中去。时间片的长度也和优先级有关，低优先级的时间片长度设置为高优先级的两倍。

3、MLFQ\_dequeue(struct run\_queue \*rq, struct proc\_struct \*proc)：将proc进程从相应的优先级运行队列中删除。

4、MLFQ\_pick\_next(struct run\_queue \*rq)：为了避免优先级较低的进程出现饥饿现象，对每个优先级设置一定的选中概率，高优先级是低优先级选中概率的两倍，然后选出一个优先级，找到这个优先级中的第一个进程返回。

5、MLFQ\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc)：和RR算法相似。