**Lab5**

**练习0：填写已有实验**

本实验依赖实验1/2/3/4。请把你做的实验1/2/3/4的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行lab5的测试应用程序，可能需对已完成的实验1/2/3/4的代码进行进一步改进。

答：

改进的alloc\_proc:

static struct proc\_struct \*

alloc\_proc(void) {

struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct));

if (proc != NULL) {

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->runs = 0;

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

memset(&(proc->context), 0, sizeof(struct context));

proc->tf = NULL;

proc->cr3 = boot\_cr3;

proc->flags = 0;

memset(proc->name, 0, PROC\_NAME\_LEN);

proc->wait\_state = 0;

proc->cptr = proc->optr = proc->yptr = NULL;

}

return proc;

}

改进的do\_fork:

Int do\_fork(uint32\_t clone\_flags, uintptr\_t stack, struct trapframe \*tf) {

int ret = -E\_NO\_FREE\_PROC;

struct proc\_struct \*proc;

if (nr\_process >= MAX\_PROCESS) {

goto fork\_out;

}

ret = -E\_NO\_MEM;

if ((proc = alloc\_proc()) == NULL) {

goto fork\_out;

}

proc->parent = current;

assert(current->wait\_state == 0);

if (setup\_kstack(proc) != 0) {

goto bad\_fork\_cleanup\_proc;

}

if (copy\_mm(clone\_flags, proc) != 0) {

goto bad\_fork\_cleanup\_kstack;

}

copy\_thread(proc, stack, tf);

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

proc->pid = get\_pid();

hash\_proc(proc);

set\_links(proc);

}

local\_intr\_restore(intr\_flag);

wakeup\_proc(proc);

ret = proc->pid;

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

改进的idt\_init:

void idt\_init(void) {

extern uintptr\_t \_\_vectors[];

int i;

for (i = 0;i<sizeof(idt)/sizeof(struct gatedesc);i ++) {

SETGATE(idt[i],0,GD\_KTEXT,\_\_vectors[i], DPL\_KERNEL);

}

SETGATE(idt[T\_SYSCALL], 1, GD\_KTEXT, \_\_vectors[T\_SYSCALL], DPL\_USER);

lidt(&idt\_pd);

}

改进的trap\_dispatch:

ticks ++;

if (ticks % TICK\_NUM == 0) {

assert(current != NULL);

current->need\_resched = 1;

}

break;

**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。

答：

（1）、调用 mm\_create 函数来申请进程的内存管理数据结构 mm 所需内存空间,并对 mm 进行初始化;

（2）、调用 setup\_pgdir来申请一个页目录表所需的一个页大小的内存空间,并把描述ucore内核虚空间映射的内核页表(boot\_pgdir所指)的内容拷贝到此新目录表中,最后让mm->pgdir指向此页目录表,这就是进程新的页目录表了,且能够正确映射内核虚空间;

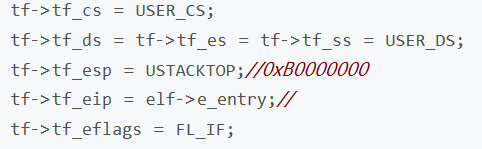
（3）、根据可执行程序的起始位置来解析此 ELF 格式的执行程序，并调用 mm\_map函数根据 ELF格式执行程序的各个段(代码段、数据段、BSS段等)的起始位置和大小建立对应的vma结构，并把vma 插入到 mm结构中，表明这些是用户进程的合法用户态虚拟地址空间;

（4）.根据可执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址,并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了;

（5）.需要给用户进程设置用户栈,为此调用 mm\_mmap 函数建立用户栈的 vma 结构,明确用户栈的位置在用户虚空间的顶端,大小为 256 个页,即1MB,并分配一定数量的物理内存且建立好栈的虚地址<-->物理地址映射关系;

（6）.至此,进程内的内存管理 vma 和 mm 数据结构已经建立完成,于是把 mm->pgdir 赋值到 cr3 寄存器中,即更新了用户进程的虚拟内存空间,此时的 init 已经被 exit 的代码和数据覆盖,成为了第一个用户进程,但此时这个用户进程的执行现场还没建立好;

（7）.先清空进程的中断帧,再重新设置进程的中断帧,使得在执行中断返回指令iret后,能够让 CPU转到用户态特权级,并回到用户态内存空间,使用用户态的代码段、数据段和堆栈,且能够跳转到用户进程的第一条指令执行,并确保在用户态能够响应中断;



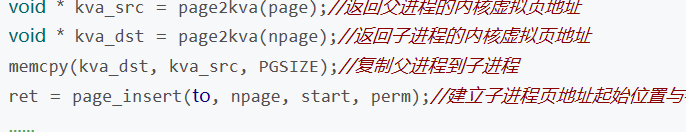
1. 请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过。

答:给用户进程建立一个能够让用户进程正常运行的用户环境。先回收自身所占用户空间，然后用新的程序覆盖内存空间，形成一个执行新程序的新进程。

**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。



Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作，则每个使用

者只需获得一个指向同一个资源A的指针，就可以该资源了。若某使用者需要对这个资源A进行写操作，系统会对该资

源进行拷贝操作，从而使得该“写操作”使用者获得一个该资源A的“私有”拷贝—资源B，可对资源B进行写操作。该“写操

作”使用者对资源B的改变对于其他的使用者而言是不可见的，因为其他使用者看到的还是资源A。

**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？

答：

（1）、分配并初始化进程控制块(alloc\_proc 函数);

（2）、分配并初始化内核栈(setup\_stack 函数);

（3）、根据 clone\_flag标志复制或共享进程内存管理结构(copy\_mm 函数);

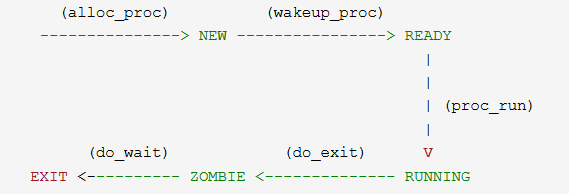
（4）、设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文(copy\_thread 函数);

（5）、把设置好的进程控制块放入hash\_list 和 proc\_list 两个全局进程链表中;

（6）、自此,进程已经准备好执行了,把进程状态设置为“就绪”态;

（7）、设置返回码为子进程的 id 号。

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）

答： 

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）

**Lab6**

**练习0：填写已有实验**

本实验依赖实验1/2/3/4/5。请把你做的实验2/3/4/5的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”“LAB5”的注释 相应部分。并确保编译通过。注意：为了能够正确执行lab6的测试应用程序，可能需对已完成的实验1/2/3/4/5的代码进行进一步改进。

答：

更改的alloc——proc：

static struct proc\_struct \*

alloc\_proc(void) {

struct proc\_struct \*proc = kmalloc(sizeof(struct proc\_struct));

if (proc != NULL) {

proc->state = PROC\_UNINIT;

proc->pid = -1;

proc->runs = 0;

proc->kstack = 0;

proc->need\_resched = 0;

proc->parent = NULL;

proc->mm = NULL;

memset(&(proc->context), 0, sizeof(struct context));

proc->tf = NULL;

proc->cr3 = boot\_cr3;

proc->flags = 0;

memset(proc->name, 0, PROC\_NAME\_LEN);

proc->wait\_state = 0;

proc->cptr = proc->optr = proc->yptr = NULL;

proc->rq = NULL;

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

proc->time\_slice = 0;

proc->lab6\_run\_pool.left = proc->lab6\_run\_pool.right = proc->lab6\_run\_pool.parent = NULL;

proc->lab6\_stride = 0;

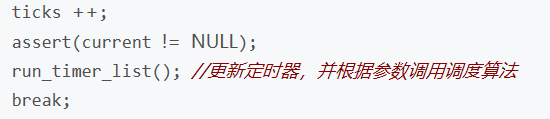
proc->lab6\_priority = 0;

}

return proc;

}

Trap——dispatch：



**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ucore的调度执行过程
2. 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

答;首先是RR\_init函数完成了对进程队列的初始化。

RR\_enqueue,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。

RR\_pick\_next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。

**练习2: 实现 Stride Scheduling 调度算法（需要编码）**

首先需要换掉RR调度器的实现，即用default\_sched\_stride\_c覆盖default\_sched.c。然后根据此文件和后续文档对Stride度器

的相关描述，完成Stride调度算法的实现。

后面的实验文档部分给出了Stride调度算法的大体描述。这里给出Stride调度算法的一些相关的资料（目前网上中文的资料比较欠缺）。

strid-shed paper location1

strid-shed paper location2

也可GOOGLE “Stride Scheduling” 来查找相关资料

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。如果只是priority.c过不去，可执行 make run-

priority 命令来单独调试它。大致执行结果可看附录。（ 使用的是 qemu-1.0.1 ）。

请在实验报告中简要说明你的设计实现过程。

答：

1、为每个runnable的进程设置一个当前状态stride，表示该进程当前的调度权。另外定义其对应的pass值，表示对应进程在调度后，stride 需要进行的累加值。

2、每次需要调度时，从当前 runnable 态的进程中选择 stride最小的进程调度。对于获得调度的进程P，将对应的stride加上其对应的步长pass（只与进程的优先权有关系）。

3、在一段固定的时间之后，回到步骤2，重新调度当前stride最小的进程

#include <defs.h>

#include <list.h>

#include <proc.h>

#include <assert.h>

#include <default\_sched.h>

#define USE\_SKEW\_HEAP 1

static int

proc\_stride\_comp\_f(void \*a, void \*b)

{

struct proc\_struct \*p = le2proc(a, lab6\_run\_pool);

struct proc\_struct \*q = le2proc(b, lab6\_run\_pool);

int32\_t c = p->lab6\_stride - q->lab6\_stride;

if (c > 0) return 1;

else if (c == 0) return 0;

else return -1;

}

static void

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

/\* LAB6: YOUR CODE \*/

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

rq->lab6\_run\_pool = NULL;

rq->proc\_num = 0;

}

static void

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

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

rq->lab6\_run\_pool = skew\_heap\_insert(rq->lab6\_run\_pool, &(proc->lab6\_run\_pool), proc\_stride\_comp\_f);

#else

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

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

#endif

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 ++;

}

static void

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

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

rq->lab6\_run\_pool =

skew\_heap\_remove(rq->lab6\_run\_pool, &(proc->lab6\_run\_pool), proc\_stride\_comp\_f);

#else

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

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

#endif

rq->proc\_num --;

}

static struct proc\_struct \*

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

/\* LAB6: YOUR CODE \*/

#if USE\_SKEW\_HEAP

if (rq->lab6\_run\_pool == NULL) return NULL;

struct proc\_struct \*p = le2proc(rq->lab6\_run\_pool, lab6\_run\_pool);

#else

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

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

return NULL;

struct proc\_struct \*p = le2proc(le, run\_link);

le = list\_next(le);

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

{

struct proc\_struct \*q = le2proc(le, run\_link);

if ((int32\_t)(p->lab6\_stride - q->lab6\_stride) > 0)

p = q;

le = list\_next(le);

}

#endif

if (p->lab6\_priority == 0)

p->lab6\_stride += BIG\_STRIDE;

else p->lab6\_stride += BIG\_STRIDE / p->lab6\_priority;

return p;

}

static void

stride\_proc\_tick(struct run\_queue \*rq, struct proc\_struct \*proc) {

/\* LAB6: YOUR CODE \*/

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

proc->time\_slice --;

}

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

proc->need\_resched = 1;

}

}

struct sched\_class default\_sched\_class = {

.name = "stride\_scheduler",

.init = stride\_init,

.enqueue = stride\_enqueue,

.dequeue = stride\_dequeue,

.pick\_next = stride\_pick\_next,

.proc\_tick = stride\_proc\_tick,

};