Lab4

练习1 分配并初始化一个进程控制块（需要编码）

alloc\_proc函数（位于kern/process/proc.c中）负责分配并返回一个新的struct proc\_struct结构，用于存储新建立的内核线程的管理信息。ucore需要对这个结构进行最基本的初始化，完成这个初始化过程。

关键数据结构 struct proc\_struct

struct proc\_struct {

enum proc\_state state; // Process state

int pid; // Process ID

int runs; // the running times of Proces

uintptr\_t kstack; // Process kernel stack

volatile bool need\_resched; // need to be rescheduled to release CPU?

struct proc\_struct \*parent; // the parent process

struct mm\_struct \*mm; // Process's memory management field

struct context context; // Switch here to run process

struct trapframe \*tf; // Trap frame for current interrupt

uintptr\_t cr3; // the base addr of Page Directroy Table(PDT)

uint32\_t flags; // Process flag

char name[PROC\_NAME\_LEN + 1]; // Process name

list\_entry\_t list\_link; // Process link list

list\_entry\_t hash\_link; // Process hash list

};

下面对参数进行简单的讲解

mm：内存管理的信息，包括内存映射列表、页表指针等。

state：进程所处的状态。

parent：用户进程的父进程（创建它的进程）。

kstack：记录了分配给该进程/线程的内核桟的位置。

need\_resched：是否需要调度

context：进程的上下文，用于进程切换

tf：中断帧的指针

cr3: cr3 保存页表的物理地址

代码填写

根据题目中的提示填写代码

static struct proc\_struct \*

alloc\_proc(void) {

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

if (proc != NULL) {

//LAB4:EXERCISE1 YOUR CODE

/\*

\* below fields in proc\_struct need to be initialized

\* enum proc\_state state; // Process state

\* int pid; // Process ID

\* int runs; // the running times of Proces

\* uintptr\_t kstack; // Process kernel stack

\* volatile bool need\_resched; // bool value: need to be rescheduled to release CPU?

\* struct proc\_struct \*parent; // the parent process

\* struct mm\_struct \*mm; // Process's memory management field

\* struct context context; // Switch here to run process

\* struct trapframe \*tf; // Trap frame for current interrupt

\* uintptr\_t cr3; // CR3 register: the base addr of Page Directroy Table(PDT)

\* uint32\_t flags; // Process flag

\* char name[PROC\_NAME\_LEN + 1]; // Process name

\*/

proc->state = PROC\_UNINIT; //设置进程为“初始”态

proc->pid = -1; //设置进程pid的未初始化值

proc->runs = 0;//初始化时间片

proc->kstack = 0;//内核栈的地址

proc->need\_resched = 0;//是否需要调度

proc->parent = NULL;//父节点为空

proc->mm = NULL; //内存管理初始化

memset(&(proc->context), 0, sizeof(struct context));//进程上下文初始化

proc->tf = NULL; //中断帧指针置为空，总是能够指向中断前的trapframe

proc->cr3 = boot\_cr3;//设置内核页目录表的基址

proc->flags = 0; //标志位初始化

memset(proc->name, 0, PROC\_NAME\_LEN); //进程名初始化

}

return proc;

}

context和\*tf的作用分析

①context：进程的上下文，用于进程切换。起到的作用就是保存了现场。在 ucore中，所有的进程在内核中也是相对独立的，因此context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context进行上下文切换的函数是在kern/process/switch.S中定义switch\_to。

② tf：中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，ucore内核允许嵌套中断。因此为了保证嵌套中断发生时tf 总是能够指向当前的tf，ucore 在内核栈上维护了 tf 的链。

练习2 为新创建的内核线程分配资源

创建一个内核线程需要分配和设置好很多资源。kernel\_thread函数通过调用do\_fork函数完成具体内核线程的创建工作。do\_kernel函数会调用alloc\_proc函数来分配并初始化一个进程控制块，但alloc\_proc只是找到了一小块内存用以记录进程的必要信息，并没有实际分配这些资源。ucore一般通过do\_fork实际创建新的内核线程。do\_fork的作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。在这个过程中，需要给新内核线程分配资源，并且复制原进程的状态。完成在kern/process/proc.c中的do\_fork函数中的处理过程。

执行步骤

①调用alloc\_proc，首先获得一块用户信息块。

②为进程分配一个内核栈。

③复制原进程的内存管理信息到新进程（但内核线程不必做此事）

④复制原进程上下文到新进程

⑤将新进程添加到进程列表

⑥唤醒新进程

⑦返回新进程号（设置子进程号为返回值）

代码填写

根据代码提示填写代码如下，同时标注了对于代码的认识理解

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;

//LAB4:EXERCISE2 YOUR CODE

/\*

\* Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.

\* MACROs or Functions:

\* alloc\_proc: create a proc struct and init fields (lab4:exercise1)

\* setup\_kstack: alloc pages with size KSTACKPAGE as process kernel stack

\* copy\_mm: process "proc" duplicate OR share process "current"'s mm according clone\_flags

\* if clone\_flags & CLONE\_VM, then "share" ; else "duplicate"

\* copy\_thread: setup the trapframe on the process's kernel stack top and

\* setup the kernel entry point and stack of process

\* hash\_proc: add proc into proc hash\_list

\* get\_pid: alloc a unique pid for process

\* wakup\_proc: set proc->state = PROC\_RUNNABLE

\* VARIABLES:

\* proc\_list: the process set's list

\* nr\_process: the number of process set

\*/

// 1. call alloc\_proc to allocate a proc\_struct

// 2. call setup\_kstack to allocate a kernel stack for child process

// 3. call copy\_mm to dup OR share mm according clone\_flag

// 4. call copy\_thread to setup tf & context in proc\_struct

// 5. insert proc\_struct into hash\_list && proc\_list

// 6. call wakup\_proc to make the new child process RUNNABLE

// 7. set ret vaule using child proc's pid

//第一步：申请内存块，如果失败，直接返回处理

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

goto fork\_out;

}

//将子进程的父节点设置为当前进程

proc->parent = current;

//第二步：为进程分配一个内核栈

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

goto bad\_fork\_cleanup\_proc;

}

//第三步：复制父进程的内存信息到子进程

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

goto bad\_fork\_cleanup\_kstack;

}

//第四步：复制父进程相关寄存器信息（上下文）

copy\_thread(proc, stack, tf);

//第五步：将新进程添加到进程列表（此过程需要加保护锁）

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

proc->pid = get\_pid();

//建立散列映射方便查找

hash\_proc(proc);

//将进程链节点加入进程列表

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

//进程数+1

nr\_process ++;

}

local\_intr\_restore(intr\_flag);

//第六步：一切准备就绪，唤醒子进程

wakeup\_proc(proc);

//第七步：别忘了设置返回的子进程号

ret = proc->pid;

fork\_out:

return ret;

bad\_fork\_cleanup\_kstack:

put\_kstack(proc);

bad\_fork\_cleanup\_proc:

kfree(proc);

goto fork\_out;

}

在使用 fork 或 clone 系统调用时产生的进程均会由内核分配一个新的唯一的PID值。

具体来说，就是在分配PID时，设置一个保护锁，暂时不允许中断，保证了ID的唯一性。上述操作真正完成了资源分配的工作，与第一步中的工作有着明显的区别。do\_fork只是创建当前进程的副本，他们执行的上下文，寄存器，代码都是一样的。

练习3 理解proc\_run和它调用的函数如何完成进程切换的

schedule代码分析

在分析 proc\_run 函数之前，我们先分析调度函数 schedule() 。

schedule()代码如下：

void

schedule(void) {

bool intr\_flag;

list\_entry\_t \*le, \*last;

struct proc\_struct \*next = NULL;

local\_intr\_save(intr\_flag);

{

current->need\_resched = 0;

last = (current == idleproc) ? &proc\_list : &(current->list\_link);

le = last;

do {

if ((le = list\_next(le)) != &proc\_list) {

next = le2proc(le, list\_link);

if (next->state == PROC\_RUNNABLE) {

break;

}

}

} while (le != last);

if (next == NULL || next->state != PROC\_RUNNABLE) {

next = idleproc;

}

next->runs ++;

if (next != current) {

proc\_run(next);

}

}

local\_intr\_restore(intr\_flag);

}

简单介绍一下schedule的执行过程：

①设置当前内核线程current->need\_resched为0；

②proc\_list队列存储着所有状态的进程/线程，在其中查找下一个处于“就绪”态的线程或进程next；

③找到这样的进程后，就调用proc\_run函数，保存当前进程current的执行现场（进程上下文），恢复新进程的执行现场，完成进程切换。

至此，新的进程next就开始执行了。由于在proc10中只有两个内核线程，且idleproc要让出CPU给initproc执行，我们可以看到schedule函数通过查找proc\_list进程队列，只能找到一个处于“就绪”态的initproc内核线程。

proc\_run代码分析

void proc\_run(struct proc\_struct \*proc) {

if (proc != current) {

bool intr\_flag;

struct proc\_struct \*prev = current, \*next = proc;

local\_intr\_save(intr\_flag);

{

current = proc;

load\_esp0(next->kstack + KSTACKSIZE);

lcr3(next->cr3);

switch\_to(&(prev->context), &(next->context));

}

local\_intr\_restore(intr\_flag);

}

}

通过proc\_run和进一步的switch\_to函数完成两个执行现场的切换，具体流程如下：

①让current指向next内核线程initproc；

②设置任务状态段ts中特权态0下的栈顶指针esp0为next内核线程initproc的内核栈的栈顶，即next->kstack + KSTACKSIZE ；

③设置CR3寄存器的值为next内核线程initproc的页目录表起始地址next->cr3，这实际上是完成进程间的页表切换；

由switch\_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当switch\_to函数执行完“ret”指令后，就切换到initproc执行了。

switch\_to函数分析

switch\_to函数的执行流程：

.globl switch\_to

switch\_to: # switch\_to(from, to)

# 保存前一个进程的执行现场，前两条汇编指令（如下所示）保存了进程在返回switch\_to函数后的指令地址到context.eip中

movl 4(%esp), %eax # eax points to from

popl 0(%eax) # esp--> return address, so save return addr in FROM’s

# 保存前一个进程的其他7个寄存器到context中的相应成员变量中。

movl %esp, 4(%eax)

movl %ebx, 8(%eax)

movl %ecx, 12(%eax)

movl %edx, 16(%eax)

movl %esi, 20(%eax)

movl %edi, 24(%eax)

movl %ebp, 28(%eax)

#再往后是恢复向一个进程的执行现场，这其实就是上述保存过程的逆执行过程，即从 context 的高地址的域 ebp 开始，逐一把相关域的值赋值给对应的寄存器。

设置了initproc->context.eip = (uintptr\_t)forkret，这样，当执行switch\_to函数并返回后，initproc将执行其实际上的执行入口地址forkret。