# 5.实验四：内核线程管理

## 5.1．实验目的：

* 了解内核线程创建/执行的管理过程
* 了解内核线程的切换和基本调度过程

## 5.2.实验内容：

实验2/3完成了物理和虚拟内存管理，这给创建内核线程（内核线程是一种特殊的进程）打下了提供内存管理的基础。当一个程序加载到内存中运行时，首先通过ucore OS的内存管理子系统分配合适的空间，然后就需要考虑如何分时使用CPU来“并发”执行多个程序，让每个运行的程序（这里用线程或进程表示）“感到”它们各自拥有“自己”的CPU。

本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：

* 内核线程只运行在内核态
* 用户进程会在在用户态和内核态交替运行
* 所有内核线程共用ucore内核内存空间，不需为每个内核线程维护单独的内存空间
* 而用户进程需要维护各自的用户内存空间

相关原理介绍可看附录B：【原理】进程/线程的属性与特征解析。

### 5.2.1．练习

对实验报告的要求：

* 基于markdown格式来完成，以文本方式为主
* 填写各个基本练习中要求完成的报告内容
* 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
* 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）
* 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

#### 练习0：填写已有实验

本实验依赖实验1/2/3。请把你做的实验1/2/3的代码填入本实验中代码中有“LAB1”,“LAB2”,“LAB3”的注释相应部分。

#### 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，你需要完成这个初始化过程。

【提示】在alloc\_proc函数的实现中，需要初始化的proc\_struct结构中的成员变量至少包括：state/pid/runs/kstack/need\_resched/parent/mm/context/tf/cr3/flags/name。

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

* 请说明proc\_struct中struct context context和struct trapframe \*tf成员变量含义和在本实验中的作用是啥？（提示通过看代码和编程调试可以判断出来）

#### 练习2：为新创建的内核线程分配资源（需要编码）

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。你需要完成在kern/process/proc.c中的do\_fork函数中的处理过程。它的大致执行步骤包括：

* 调用alloc\_proc，首先获得一块用户信息块。
* 为进程分配一个内核栈。
* 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）
* 复制原进程上下文到新进程
* 将新进程添加到进程列表
* 唤醒新进程
* 返回新进程号

请在实验报告中简要说明你的设计实现过程。请回答如下问题：

* 请说明ucore是否做到给每个新fork的线程一个唯一的id？请说明你的分析和理由。

#### 练习3：阅读代码，理解 proc\_run 函数和它调用的函数如何完成进程切换的。（无编码工作）

请在实验报告中简要说明你对proc\_run函数的分析。并回答如下问题：

* 在本实验的执行过程中，创建且运行了几个内核线程？
* 语句local\_intr\_save(intr\_flag);....local\_intr\_restore(intr\_flag);在这里有何作用?请说明理由

完成代码编写后，编译并运行代码：make qemu

如果可以得到如 附录A所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

#### 扩展练习Challenge：实现支持任意大小的内存分配算法

这不是本实验的内容，其实是上一次实验内存的扩展，但考虑到现在的slab算法比较复杂，有必要实现一个比较简单的任意大小内存分配算法。可参考本实验中的slab如何调用基于页的内存分配算法（注意，不是要你关注slab的具体实现）来实现first-fit/best-fit/worst-fit/buddy等支持任意大小的内存分配算法。。

【注意】下面是相关的Linux实现文档，供参考

SLOB

<http://en.wikipedia.org/wiki/SLOB> <http://lwn.net/Articles/157944/>

SLAB

<https://www.ibm.com/developerworks/cn/linux/l-linux-slab-allocator/>

### 5.2.2．项目组成

├── boot

├── kern

│ ├── debug

│ ├── driver

│ ├── fs

│ ├── init

│ │ ├── init.c

│ │ └── ...

│ ├── libs

│ │ ├── rb\\_tree.c

│ │ ├── rb\\_tree.h

│ │ └── ...

│ ├── mm

│ │ ├── kmalloc.c

│ │ ├── kmalloc.h

│ │ ├── memlayout.h

│ │ ├── pmm.c

│ │ ├── pmm.h

│ │ ├── swap.c

│ │ ├── vmm.c

│ │ └── ...

│ ├── process

│ │ ├── entry.S

│ │ ├── proc.c

│ │ ├── proc.h

│ │ └── switch.S

│ ├── schedule

│ │ ├── sched.c

│ │ └── sched.h

│ ├── sync

│ │ └── sync.h

│ └── trap

│ ├── trapentry.S

│ └── ...

├── libs

│ ├── hash.c

│ ├── stdlib.h

│ ├── unistd.h

│ └── ...

├── Makefile

└── tools

相对与实验三，实验四中主要改动如下：

* kern/process/ （新增进程管理相关文件）
  + proc.[ch]：新增：实现进程、线程相关功能，包括：创建进程/线程，初始化进程/线程，处理进程/线程退出等功能
  + entry.S：新增：内核线程入口函数kernel\_thread\_entry的实现
  + switch.S：新增：上下文切换，利用堆栈保存、恢复进程上下文
* kern/init/
  + init.c：修改：完成进程系统初始化，并在内核初始化后切入idle进程
* kern/mm/ （基本上与本次实验没有太直接的联系，了解kmalloc和kfree如何使用即可）
  + kmalloc.[ch]：新增：定义和实现了新的kmalloc/kfree函数。具体实现是基于slab分配的简化算法 （只要求会调用这两个函数即可）
  + memlayout.h：增加slab物理内存分配相关的定义与宏 （可不用理会）。
  + pmm.[ch]：修改：在pmm.c中添加了调用kmalloc\_init函数,取消了老的kmalloc/kfree的实现；在pmm.h中取消了老的kmalloc/kfree的定义
  + swap.c：修改：取消了用于check的Line 185的执行
  + vmm.c：修改：调用新的kmalloc/kfree
* kern/trap/
  + trapentry.S：增加了汇编写的函数forkrets，用于do\_fork调用的返回处理。
* kern/schedule/
  + sched.[ch]：新增：实现FIFO策略的进程调度
* kern/libs
  + rb\_tree.[ch]：新增：实现红黑树，被slab分配的简化算法使用（可不用理会）

**编译执行**

编译并运行代码的命令如下：

make

make qemu

则可以得到如下的显示内容（仅供参考，不是标准答案输出）

(THU.CST) os is loading ...

Special kernel symbols:

entry 0xc010002a (phys)

etext 0xc010a708 (phys)

edata 0xc0127ae0 (phys)

end 0xc012ad58 (phys)

...

++ setup timer interrupts

this initproc, pid = 1, name = "init"

To U: "Hello world!!".

To U: "en.., Bye, Bye. :)"

kernel panic at kern/process/proc.c:354:

process exit!!.

Welcome to the kernel debug monitor!!

Type 'help' for a list of commands.

K> qemu: terminating on signal 2

## 5.3. 内核线程管理

### 5.3.1．实验执行流程概述

lab2和lab3完成了对内存的虚拟化，但整个控制流还是一条线串行执行。lab4将在此基础上进行CPU的虚拟化，即让ucore实现分时共享CPU，实现多条控制流能够并发执行。从某种程度上，我们可以把控制流看作是一个内核线程。本次实验将首先接触的是内核线程的管理。内核线程是一种特殊的进程，内核线程与用户进程的区别有两个：内核线程只运行在内核态而用户进程会在在用户态和内核态交替运行；所有内核线程直接使用共同的ucore内核内存空间，不需为每个内核线程维护单独的内存空间而用户进程需要维护各自的用户内存空间。从内存空间占用情况这个角度上看，我们可以把线程看作是一种共享内存空间的轻量级进程。

为了实现内核线程，需要设计管理线程的数据结构，即进程控制块（在这里也可叫做线程控制块）。如果要让内核线程运行，我们首先要创建内核线程对应的进程控制块，还需把这些进程控制块通过链表连在一起，便于随时进行插入，删除和查找操作等进程管理事务。这个链表就是进程控制块链表。然后在通过调度器（scheduler）来让不同的内核线程在不同的时间段占用CPU执行，实现对CPU的分时共享。那lab4中是如何一步一步实现这个过程的呢？

我们还是从lab4/kern/init/init.c中的kern\_init函数入手分析。在kern\_init函数中，当完成虚拟内存的初始化工作后，就调用了proc\_init函数，这个函数完成了idleproc内核线程和initproc内核线程的创建或复制工作，这也是本次实验要完成的练习。idleproc内核线程的工作就是不停地查询，看是否有其他内核线程可以执行了，如果有，马上让调度器选择那个内核线程执行（请参考cpu\_idle函数的实现）。所以idleproc内核线程是在ucore操作系统没有其他内核线程可执行的情况下才会被调用。接着就是调用kernel\_thread函数来创建initproc内核线程。initproc内核线程的工作就是显示“Hello World”，表明自己存在且能正常工作了。

调度器会在特定的调度点上执行调度，完成进程切换。在lab4中，这个调度点就一处，即在cpu\_idle函数中，此函数如果发现当前进程（也就是idleproc）的need\_resched置为1（在初始化idleproc的进程控制块时就置为1了），则调用schedule函数，完成进程调度和进程切换。进程调度的过程其实比较简单，就是在进程控制块链表中查找到一个“合适”的内核线程，所谓“合适”就是指内核线程处于“PROC\_RUNNABLE”状态。在接下来的switch\_to函数(在后续有详细分析，有一定难度，需深入了解一下)完成具体的进程切换过程。一旦切换成功，那么initproc内核线程就可以通过显示字符串来表明本次实验成功。

接下来将主要介绍了进程创建所需的重要数据结构--进程控制块 proc\_struct，以及ucore创建并执行内核线程idleproc和initproc的两种不同方式，特别是创建initproc的方式将被延续到实验五中，扩展为创建用户进程的主要方式。另外，还初步涉及了进程调度（实验六涉及并会扩展）和进程切换内容。

### 5.3.2．设计关键数据结构 -- 进程控制块

在实验四中，进程管理信息用struct proc\_struct表示，在*kern/process/proc.h*中定义如下：

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

};

下面重点解释一下几个比较重要的成员变量：

● mm：内存管理的信息，包括内存映射列表、页表指针等。mm成员变量在lab3中用于虚存管理。但在实际OS中，内核线程常驻内存，不需要考虑swap page问题，在lab5中涉及到了用户进程，才考虑进程用户内存空间的swap page问题，mm才会发挥作用。所以在lab4中mm对于内核线程就没有用了，这样内核线程的proc\_struct的成员变量\*mm=0是合理的。mm里有个很重要的项pgdir，记录的是该进程使用的一级页表的物理地址。由于\*mm=NULL，所以在proc\_struct数据结构中需要有一个代替pgdir项来记录页表起始地址，这就是proc\_struct数据结构中的cr3成员变量。

● state：进程所处的状态。

● parent：用户进程的父进程（创建它的进程）。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个内核线程idleproc。内核根据这个父子关系建立一个树形结构，用于维护一些特殊的操作，例如确定某个进程是否可以对另外一个进程进行某种操作等等。

● context：进程的上下文，用于进程切换（参见switch.S）。在 uCore中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等等）。使用 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。

● 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trapframe，uCore 在内核栈上维护了 tf 的链，可以参考trap.c::trap函数做进一步的了解。

● cr3: cr3 保存页表的物理地址，目的就是进程切换的时候方便直接使用 lcr3实现页表切换，避免每次都根据 mm 来计算 cr3。mm数据结构是用来实现用户空间的虚存管理的，但是内核线程没有用户空间，它执行的只是内核中的一小段代码（通常是一小段函数），所以它没有mm 结构，也就是NULL。当某个进程是一个普通用户态进程的时候，PCB 中的 cr3 就是 mm 中页表（pgdir）的物理地址；而当它是内核线程的时候，cr3 等于boot\_cr3。而boot\_cr3指向了uCore启动时建立好的饿内核虚拟空间的页目录表首地址。

● kstack: 每个线程都有一个内核栈，并且位于内核地址空间的不同位置。对于内核线程，该栈就是运行时的程序使用的栈；而对于普通进程，该栈是发生特权级改变的时候使保存被打断的硬件信息用的栈。uCore在创建进程时分配了 2 个连续的物理页（参见memlayout.h中KSTACKSIZE的定义）作为内核栈的空间。这个栈很小，所以内核中的代码应该尽可能的紧凑，并且避免在栈上分配大的数据结构，以免栈溢出，导致系统崩溃。kstack记录了分配给该进程/线程的内核栈的位置。主要作用有以下几点。首先，当内核准备从一个进程切换到另一个的时候，需要根据kstack 的值正确的设置好 tss （可以回顾一下在实验一中讲述的 tss 在中断处理过程中的作用），以便在进程切换以后再发生中断时能够使用正确的栈。其次，内核栈位于内核地址空间，并且是不共享的（每个线程都拥有自己的内核栈），因此不受到 mm 的管理，当进程退出的时候，内核能够根据 kstack 的值快速定位栈的位置并进行回收。uCore 的这种内核栈的设计借鉴的是 linux 的方法（但由于内存管理实现的差异，它实现的远不如 linux 的灵活），它使得每个线程的内核栈在不同的位置，这样从某种程度上方便调试，但同时也使得内核对栈溢出变得十分不敏感，因为一旦发生溢出，它极可能污染内核中其它的数据使得内核崩溃。如果能够通过页表，将所有进程的内核栈映射到固定的地址上去，能够避免这种问题，但又会使得进程切换过程中对栈的修改变得相当繁琐。感兴趣的同学可以参考 linux kernel 的代码对此进行尝试。

为了管理系统中所有的进程控制块，uCore维护了如下全局变量（位于*kern/process/proc.c*）：

● static struct proc \*current：当前占用CPU且处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证操作的原子性，目前至少需要屏蔽中断。可以参考 switch\_to 的实现。

● static struct proc \*initproc：本实验中，指向一个内核线程。本实验以后，此指针将指向第一个用户态进程。

● static list\_entry\_t hash\_list[HASH\_LIST\_SIZE]：所有进程控制块的哈希表，proc\_struct中的成员变量hash\_link将基于pid链接入这个哈希表中。

● list\_entry\_t proc\_list：所有进程控制块的双向线性列表，proc\_struct中的成员变量list\_link将链接入这个链表中。

### 5.3.3．创建并执行内核线程

建立进程控制块（proc.c中的alloc\_proc函数）后，现在就可以通过进程控制块来创建具体的进程/线程了。首先，考虑最简单的内核线程，它通常只是内核中的一小段代码或者函数，没有自己的“专属”空间。这是由于在uCore OS启动后，已经对整个内核内存空间进行了管理，通过设置页表建立了内核虚拟空间（即boot\_cr3指向的二级页表描述的空间）。所以uCore OS内核中的所有线程都不需要再建立各自的页表，只需共享这个内核虚拟空间就可以访问整个物理内存了。从这个角度看，内核线程被uCore OS内核这个大“内核进程”所管理。

#### 5.3.3.1. 创建第 0 个内核线程 idleproc

在init.c::kern\_init函数调用了proc.c::proc\_init函数。proc\_init函数启动了创建内核线程的步骤。首先当前的执行上下文（从kern\_init 启动至今）就可以看成是uCore内核（也可看做是内核进程）中的一个内核线程的上下文。为此，uCore通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第0个内核线程 -- idleproc。具体步骤如下：

首先调用alloc\_proc函数来通过kmalloc函数获得proc\_struct结构的一块内存块-，作为第0个进程控制块。并把proc进行初步初始化（即把proc\_struct中的各个成员变量清零）。但有些成员变量设置了特殊的值，比如：

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->cr3 = boot\_cr3; 使用内核页目录表的基址

...

上述三条语句中,第一条设置了进程的状态为“初始”态，这表示进程已经 “出生”了，正在获取资源茁壮成长中；第二条语句设置了进程的pid为-1，这表示进程的“身份证号”还没有办好；第三条语句表明由于该内核线程在内核中运行，故采用为uCore内核已经建立的页表，即设置为在uCore内核页表的起始地址boot\_cr3。后续实验中可进一步看出所有内核线程的内核虚地址空间（也包括物理地址空间）是相同的。既然内核线程共用一个映射内核空间的页表，这表示内核空间对所有内核线程都是“可见”的，所以更精确地说，这些内核线程都应该是从属于同一个唯一的“大内核进程”—uCore内核。

接下来，proc\_init函数对idleproc内核线程进行进一步初始化：

idleproc->pid = 0;

idleproc->state = PROC\_RUNNABLE;

idleproc->kstack = (uintptr\_t)bootstack;

idleproc->need\_resched = 1;

set\_proc\_name(idleproc, "idle");

需要注意前4条语句。第一条语句给了idleproc合法的身份证号--0，这名正言顺地表明了idleproc是第0个内核线程。通常可以通过pid的赋值来表示线程的创建和身份确定。“0”是第一个的表示方法是计算机领域所特有的，比如C语言定义的第一个数组元素的小标也是“0”。第二条语句改变了idleproc的状态，使得它从“出生”转到了“准备工作”，就差uCore调度它执行了。第三条语句设置了idleproc所使用的内核栈的起始地址。需要注意以后的其他线程的内核栈都需要通过分配获得，因为uCore启动时设置的内核栈直接分配给idleproc使用了。第四条很重要，因为uCore希望当前CPU应该做更有用的工作，而不是运行idleproc这个“无所事事”的内核线程，所以把idleproc->need\_resched设置为“1”，结合idleproc的执行主体--cpu\_idle函数的实现，可以清楚看出如果当前idleproc在执行，则只要此标志为1，马上就调用schedule函数要求调度器切换其他进程执行。

**关键数据结构和变量**

first\_fit分配算法需要维护一个查找有序（地址按从小到大排列）空闲块（以页为最小单位的连续地址空间）的数据结构，而双向链表是一个很好的选择。

libs/list.h定义了可挂接任意元素的通用双向链表结构和对应的操作，所以需要了解如何使用这个文件提供的各种函数，从而可以完成对双向链表的初始化/插入/删除等。

kern/mm/memlayout.h中定义了一个 free\_area\_t 数据结构，包含成员结构

list\_entry\_t free\_list; // the list header 空闲块双向链表的头

unsigned int nr\_free; // # of free pages in this free list 空闲块的总数（以页为单位）

显然，我们可以通过此数据结构来完成对空闲块的管理。而default\_pmm.c中定义的free\_area变量就是干这个事情的。

kern/mm/pmm.h中定义了一个通用的分配算法的函数列表，用pmm\_manager 表示。其中init函数就是用来初始化free\_area变量的, first\_fit分配算法可直接重用default\_init函数的实现。init\_memmap函数需要根据现有的内存情况构建空闲块列表的初始状态。何时应该执行这个函数呢？

通过分析代码，可以知道：

kern\_init --> pmm\_init-->page\_init-->init\_memmap--> pmm\_manager->init\_memmap

所以，default\_init\_memmap需要根据page\_init函数中传递过来的参数（某个连续地址的空闲块的起始页，页个数）来建立一个连续内存空闲块的双向链表。这里有一个假定page\_init函数是按地址从小到大的顺序传来的连续内存空闲块的。链表头是free\_area.free\_list，链表项是Page数据结构的base->page\_link。这样我们就依靠Page数据结构中的成员变量page\_link形成了连续内存空闲块列表。

**设计实现**

default\_init\_memmap函数讲根据每个物理页帧的情况来建立空闲页链表，且空闲页块应该是根据地址高低形成一个有序链表。根据上述变量的定义，default\_init\_memmap可大致实现如下：

default\_init\_memmap(struct Page \*base, size\_t n) {

struct Page \*p = base;

for (; p != base + n; p ++) {

p->flags = p->property = 0;

set\_page\_ref(p, 0);

}

base->property = n;

SetPageProperty(base);

nr\_free += n;

list\_add(&free\_list, &(base->page\_link));

}

如果要分配一个页，那要考虑哪些呢？这里就需要考虑实现default\_alloc\_pages函数，注意参数n表示要分配n个页。另外，需要注意实现时尽量多考虑一些边界情况，这样确保软件的鲁棒性。比如

if (n > nr\_free) {

return NULL;

}

这样可以确保分配不会超出范围。也可加一些 assert函数，在有错误出现时，能够迅速发现。比如 n应该大于0，我们就可以加上

assert(n \> 0);

这样在n<=0的情况下，ucore会迅速报错。firstfit需要从空闲链表头开始查找最小的地址，通过list\_next找到下一个空闲块元素，通过le2page宏可以更加链表元素获得对应的Page指针p。通过p->property可以了解此空闲块的大小。如果>=n，这就找到了！如果<n，则list\_next，继续查找。直到list\_next== &free\_list，这表示找完了一遍了。找到后，就要从新组织空闲块，然后把找到的page返回。所以default\_alloc\_pages可大致实现如下：

static struct Page \*

default\_alloc\_pages(size\_t n) {

if (n > nr\_free) {

return NULL;

}

struct Page \*page = NULL;

list\_entry\_t \*le = &free\_list;

while ((le = list\_next(le)) != &free\_list) {

struct Page \*p = le2page(le, page\_link);

if (p->property >= n) {

page = p;

break;

}

}

if (page != NULL) {

list\_del(&(page->page\_link));

if (page->property > n) {

struct Page \*p = page + n;

p->property = page->property - n;

list\_add(&free\_list, &(p->page\_link));

}

nr\_free -= n;

ClearPageProperty(page);

}

return page;

}

default\_free\_pages函数的实现其实是default\_alloc\_pages的逆过程，不过需要考虑空闲块的合并问题。这里就不再细讲了。注意，上诉代码只是参考设计，不是完整的正确设计。更详细的说明位于lab2/kernel/mm/default\_pmm.c的注释中。希望同学能够顺利完成本实验的第一部分。

#### 5.3.3.2. 创建第 1 个内核线程initproc

第0个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行cpu\_idle函数开始过退休生活了。所以uCore接下来还需创建其他进程来完成各种工作，但idleproc内核子线程自己不想做，于是就通过调用kernel\_thread函数创建了一个内核线程init\_main。在实验四中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看init\_main函数）。但在后续的实验中，init\_main的工作就是创建特定的其他内核线程或用户进程（实验五涉及）。下面我们来分析一下创建内核线程的函数kernel\_thread：

kernel\_thread(int (\*fn)(void \*), void \*arg, uint32\_t clone\_flags)

{

struct trapframe tf;

memset(&tf, 0, sizeof(struct trapframe));

tf.tf\_cs = KERNEL\_CS;

tf.tf\_ds = tf\_struct.tf\_es = tf\_struct.tf\_ss = KERNEL\_DS;

tf.tf\_regs.reg\_ebx = (uint32\_t)fn;

tf.tf\_regs.reg\_edx = (uint32\_t)arg;

tf.tf\_eip = (uint32\_t)kernel\_thread\_entry;

return do\_fork(clone\_flags | CLONE\_VM, 0, &tf);

}

注意，kernel\_thread函数采用了局部变量tf来放置保存内核线程的临时中断帧，并把中断帧的指针传递给do\_fork函数，而do\_fork函数会调用copy\_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间。

给中断帧分配完空间后，就需要构造新进程的中断帧，具体过程是：首先给tf进行清零初始化，并设置中断帧的代码段（tf.tf\_cs）和数据段(tf.tf\_ds/tf\_es/tf\_ss)为内核空间的段（KERNEL\_CS/KERNEL\_DS），这实际上也说明了initproc内核线程在内核空间中执行。而initproc内核线程从哪里开始执行呢？tf.tf\_eip的指出了是kernel\_thread\_entry（位于kern/process/entry.S中），kernel\_thread\_entry是entry.S中实现的汇编函数，它做的事情很简单：

kernel\_thread\_entry: # void kernel\_thread(void)

pushl %edx # push arg

call \*%ebx # call fn

pushl %eax # save the return value of fn(arg)

call do\_exit # call do\_exit to terminate current thread

从上可以看出，kernel\_thread\_entry函数主要为内核线程的主体fn函数做了一个准备开始和结束运行的“壳”，并把函数fn的参数arg（保存在edx寄存器中）压栈，然后调用fn函数，把函数返回值eax寄存器内容压栈，调用do\_exit函数退出线程执行。

do\_fork是创建线程的主要函数。kernel\_thread函数通过调用do\_fork函数最终完成了内核线程的创建工作。下面我们来分析一下do\_fork函数的实现（练习2）。do\_fork函数主要做了以下6件事情：

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。

这里需要注意的是，如果上述前3步执行没有成功，则需要做对应的出错处理，把相关已经占有的内存释放掉。copy\_mm函数目前只是把current->mm设置为NULL，这是由于目前在实验四中只能创建内核线程，proc->mm描述的是进程用户态空间的情况，所以目前mm还用不上。copy\_thread函数做的事情比较多，代码如下：

static void

copy\_thread(struct proc\_struct \*proc, uintptr\_t esp, struct trapframe \*tf) {

//在内核堆栈的顶部设置中断帧大小的一块栈空间

proc->tf = (struct trapframe \*)(proc->kstack + KSTACKSIZE) - 1;

\*(proc->tf) = \*tf; //拷贝在kernel\_thread函数建立的临时中断帧的初始值

proc->tf->tf\_regs.reg\_eax = 0;

//设置子进程/线程执行完do\_fork后的返回值

proc->tf->tf\_esp = esp; //设置中断帧中的栈指针esp

proc->tf->tf\_eflags |= FL\_IF; //使能中断

proc->context.eip = (uintptr\_t)forkret;

proc->context.esp = (uintptr\_t)(proc->tf);

}

此函数首先在内核堆栈的顶部设置中断帧大小的一块栈空间，并在此空间中拷贝在kernel\_thread函数建立的临时中断帧的初始值，并进一步设置中断帧中的栈指针esp和标志寄存器eflags，特别是eflags设置了FL\_IF标志，这表示此内核线程在执行过程中，能响应中断，打断当前的执行。执行到这步后，此进程的中断帧就建立好了，对于initproc而言，它的中断帧如下所示：

//所在地址位置

initproc->tf= (proc->kstack+KSTACKSIZE) – sizeof (struct trapframe);

//具体内容

initproc->tf.tf\_cs = KERNEL\_CS;

initproc->tf.tf\_ds = initproc->tf.tf\_es = initproc->tf.tf\_ss = KERNEL\_DS;

initproc->tf.tf\_regs.reg\_ebx = (uint32\_t)init\_main;

initproc->tf.tf\_regs.reg\_edx = (uint32\_t) ADDRESS of "Helloworld!!";

initproc->tf.tf\_eip = (uint32\_t)kernel\_thread\_entry;

initproc->tf.tf\_regs.reg\_eax = 0;

initproc->tf.tf\_esp = esp;

initproc->tf.tf\_eflags |= FL\_IF;

设置好中断帧后，最后就是设置initproc的进程上下文，（process context，也称执行现场）了。只有设置好执行现场后，一旦uCore调度器选择了initproc执行，就需要根据initproc->context中保存的执行现场来恢复initproc的执行。这里设置了initproc的执行现场中主要的两个信息：上次停止执行时的下一条指令地址context.eip和上次停止执行时的堆栈地址context.esp。其实initproc还没有执行过，所以这其实就是initproc实际执行的第一条指令地址和堆栈指针。可以看出，由于initproc的中断帧占用了实际给initproc分配的栈空间的顶部，所以initproc就只能把栈顶指针context.esp设置在initproc的中断帧的起始位置。根据context.eip的赋值，可以知道initproc实际开始执行的地方在forkret函数（主要完成do\_fork函数返回的处理工作）处。至此，initproc内核线程已经做好准备执行了

#### 5.3.3.3. 调度并执行内核线程 initproc

在uCore执行完proc\_init函数后，就创建好了两个内核线程：idleproc和initproc，这时uCore当前的执行现场就是idleproc，等到执行到init函数的最后一个函数cpu\_idle之前，uCore的所有初始化工作就结束了，idleproc将通过执行cpu\_idle函数让出CPU，给其它内核线程执行，具体过程如下：

void

cpu\_idle(void) {

while (1) {

if (current->need\_resched) {

schedule();

……

首先，判断当前内核线程idleproc的need\_resched是否不为0，回顾前面“创建第一个内核线程idleproc”中的描述，proc\_init函数在初始化idleproc中，就把idleproc->need\_resched置为1了，所以会马上调用schedule函数找其他处于“就绪”态的进程执行。

uCore在实验四中只实现了一个最简单的FIFO调度器，其核心就是schedule函数。它的执行逻辑很简单：

1．设置当前内核线程current->need\_resched为0； 2．在proc\_list队列中查找下一个处于“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。并通过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；
4. 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。

注意，在第二步设置任务状态段ts中特权态0下的栈顶指针esp0的目的是建立好内核线程或将来用户线程在执行特权态切换（从特权态0<-->特权态3，或从特权态3<-->特权态3）时能够正确定位处于特权态0时进程的内核栈的栈顶，而这个栈顶其实放了一个trapframe结构的内存空间。如果是在特权态3发生了中断/异常/系统调用，则CPU会从特权态3-->特权态0，且CPU从此栈顶（当前被打断进程的内核栈顶）开始压栈来保存被中断/异常/系统调用打断的用户态执行现场；如果是在特权态0发生了中断/异常/系统调用，则CPU会从从当前内核栈指针esp所指的位置开始压栈保存被中断/异常/系统调用打断的内核态执行现场。反之，当执行完对中断/异常/系统调用打断的处理后，最后会执行一个“iret”指令。在执行此指令之前，CPU的当前栈指针esp一定指向上次产生中断/异常/系统调用时CPU保存的被打断的指令地址CS和EIP，“iret”指令会根据ESP所指的保存的址CS和EIP恢复到上次被打断的地方继续执行。

在页表设置方面，由于idleproc和initproc都是共用一个内核页表boot\_cr3，所以此时第三步其实没用，但考虑到以后的进程有各自的页表，其起始地址各不相同，只有完成页表切换，才能确保新的进程能够正常执行。

第四步proc\_run函数调用switch\_to函数，参数是前一个进程和后一个进程的执行现场：process context。在上一节“设计进程控制块”中，描述了context结构包含的要保存和恢复的寄存器。我们再看看switch.S中的switch\_to函数的执行流程：

.globl switch\_to

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

# save from's registers

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

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

context

movl %esp, 4(%eax)

……

movl %ebp, 28(%eax)

# restore to's registers

movl 4(%esp), %eax # not 8(%esp): popped return address already

# eax now points to to

movl 28(%eax), %ebp

……

movl 4(%eax), %esp

pushl 0(%eax) # push TO’s context’s eip, so return addr = TO’s eip

ret # after ret, eip= TO’s eip

首先，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

context

在接下来的7条汇编指令完成了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。至此前一个进程的执行现场保存完毕。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的高地址的成员变量ebp开始，逐一把相关成员变量的值赋值给对应的寄存器，倒数第二条汇编指令“pushl 0(%eax)”其实把context中保存的下一个进程要执行的指令地址context.eip放到了堆栈顶，这样接下来执行最后一条指令“ret”时，会把栈顶的内容赋值给EIP寄存器，这样就切换到下一个进程执行了，即当前进程已经是下一个进程了。uCore会执行进程切换，让initproc执行。在对initproc进行初始化时，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。而forkret会调用位于kern/trap/trapentry.S中的forkrets函数执行，具体代码如下：

.globl \_\_trapret

\_\_trapret:

# restore registers from stack

popal

# restore %ds and %es

popl %es

popl %ds

# get rid of the trap number and error code

addl $0x8, %esp

iret

.globl forkrets

forkrets:

# set stack to this new process's trapframe

movl 4(%esp), %esp //把esp指向当前进程的中断帧

jmp \_\_trapret

可以看出，forkrets函数首先把esp指向当前进程的中断帧，从\_trapret开始执行到iret前，esp指向了current->tf.tf\_eip，而如果此时执行的是initproc，则current->tf.tf\_eip=kernel\_thread\_entry，initproc->tf.tf\_cs = KERNEL\_CS，所以当执行完iret后，就开始在内核中执行kernel\_thread\_entry函数了，而initproc->tf.tf\_regs.reg\_ebx = init\_main，所以在kernl\_thread\_entry中执行“call %ebx”后，就开始执行initproc的主体了。Initprocde的主体函数很简单就是输出一段字符串，然后就返回到kernel\_tread\_entry函数，并进一步调用do\_exit执行退出操作了。本来do\_exit应该完成一些资源回收工作等，但这些不是实验四涉及的，而是由后续的实验来完成。至此，实验四中的主要工作描述完毕。

## 3.4．实验报告要求

从git server网站上取得ucore\_lab后，进入目录labcodes/lab4，完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录labcodes/lab4下存放实验报告，实验报告文档命名为lab4-学生ID.md。推荐用**markdown**格式。对于lab4中编程任务，完成编写之后，再通过git push命令把代码同步回git server网站。最后请一定提前或按时提交到git server网站。

注意有“LAB4”的注释，代码中所有需要完成的地方（challenge除外）都有“LAB4”和“YOUR CODE”的注释，请在提交时特别注意保持注释，并将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

## 5.5. 附录A. 实验四的参考输出如下：

make qemu

(THU.CST) os is loading ...

Special kernel symbols:

entry 0xc010002c (phys)

etext 0xc010d0f7 (phys)

edata 0xc012dad0 (phys)

end 0xc0130e78 (phys)

Kernel executable memory footprint: 196KB

memory management: default\_pmm\_manager

e820map:

memory: 0009f400, [00000000, 0009f3ff], type = 1.

memory: 00000c00, [0009f400, 0009ffff], type = 2.

memory: 00010000, [000f0000, 000fffff], type = 2.

memory: 07efd000, [00100000, 07ffcfff], type = 1.

memory: 00003000, [07ffd000, 07ffffff], type = 2.

memory: 00040000, [fffc0000, ffffffff], type = 2.

check\_alloc\_page() succeeded!

check\_pgdir() succeeded!

check\_boot\_pgdir() succeeded!

-------------------- BEGIN --------------------

PDE(0e0) c0000000-f8000000 38000000 urw

|-- PTE(38000) c0000000-f8000000 38000000 -rw

PDE(001) fac00000-fb000000 00400000 -rw

|-- PTE(000e0) faf00000-fafe0000 000e0000 urw

|-- PTE(00001) fafeb000-fafec000 00001000 -rw

--------------------- END ---------------------

check\_slab() succeeded!

kmalloc\_init() succeeded!

check\_vma\_struct() succeeded!

page fault at 0x00000100: K/W [no page found].

check\_pgfault() succeeded!

check\_vmm() succeeded.

ide 0: 10000(sectors), 'QEMU HARDDISK'.

ide 1: 262144(sectors), 'QEMU HARDDISK'.

SWAP: manager = fifo swap manager

BEGIN check\_swap: count 1, total 31944

mm->sm\_priv c0130e64 in fifo\_init\_mm

setup Page Table for vaddr 0X1000, so alloc a page

setup Page Table vaddr 0~4MB OVER!

set up init env for check\_swap begin!

page fault at 0x00001000: K/W [no page found].

page fault at 0x00002000: K/W [no page found].

page fault at 0x00003000: K/W [no page found].

page fault at 0x00004000: K/W [no page found].

set up init env for check\_swap over!

write Virt Page c in fifo\_check\_swap

write Virt Page a in fifo\_check\_swap

write Virt Page d in fifo\_check\_swap

write Virt Page b in fifo\_check\_swap

write Virt Page e in fifo\_check\_swap

page fault at 0x00005000: K/W [no page found].

swap\_out: i 0, store page in vaddr 0x1000 to disk swap entry 2

write Virt Page b in fifo\_check\_swap

write Virt Page a in fifo\_check\_swap

page fault at 0x00001000: K/W [no page found].

swap\_out: i 0, store page in vaddr 0x2000 to disk swap entry 3

swap\_in: load disk swap entry 2 with swap\_page in vadr 0x1000

write Virt Page b in fifo\_check\_swap

page fault at 0x00002000: K/W [no page found].

swap\_out: i 0, store page in vaddr 0x3000 to disk swap entry 4

swap\_in: load disk swap entry 3 with swap\_page in vadr 0x2000

write Virt Page c in fifo\_check\_swap

page fault at 0x00003000: K/W [no page found].

swap\_out: i 0, store page in vaddr 0x4000 to disk swap entry 5

swap\_in: load disk swap entry 4 with swap\_page in vadr 0x3000

write Virt Page d in fifo\_check\_swap

page fault at 0x00004000: K/W [no page found].

swap\_out: i 0, store page in vaddr 0x5000 to disk swap entry 6

swap\_in: load disk swap entry 5 with swap\_page in vadr 0x4000

check\_swap() succeeded!

++ setup timer interrupts

this initproc, pid = 1, name = "init"

To U: "Hello world!!".

To U: "en.., Bye, Bye. :)"

kernel panic at kern/process/proc.c:316:

process exit!!.

Welcome to the kernel debug monitor!!

Type 'help' for a list of commands.

K>

## 3.6. 附录B.【原理】进程的属性与特征解析

物操作系统负责进程管理，即从程序加载到运行结束的全过程，这个程序运行过程将经历从“出生”到“死亡”的完整“生命”历程。所谓“进程”就是指这个程序运行的整个执行过程。为了记录、描述和管理程序执行的动态变化过程，需要有一个数据结构，这就是进程控制块。进程与进程控制块是一一对应的。为此，ucore需要建立合适的进程控制块数据结构，并基于进程控制块来完成对进程的管理。

为了让多个程序能够使用CPU执行任务，需要设计用于进程管理的内核数据结构“进程控制块”。但到底如何设计进程控制块，如何管理进程？如果对进程的属性和特征了解不够，则无法有效地设计进程控制块和实现进程管理。

再一次回到进程的定义：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。这里有四个关键词：程序、数据集合、执行和动态执行过程。从CPU的角度来看，所谓程序就是一段特定的指令机器码序列而已。CPU会一条一条地取出在内存中程序的指令并按照指令的含义执行各种功能；所谓数据集合就是使用的内存；所谓执行就是让CPU工作。这个数据集合和执行其实体现了进程对资源的占用。动态执行过程体现了程序执行的不同“生命”阶段：诞生、工作、休息/等待、死亡。如果这一段指令执行完毕，也就意味着进程结束了。从开始执行到执行结束是一个进程的全过程。那么操作系统需要管理进程的什么？如果计算机系统中只有一个进程，那操作系统的工作就简单了。进程管理就是管理进程执行的指令，进程占用的资源，进程执行的状态。这可归结为对一个进程内的管理工作。但实际上在计算机系统的内存中，可以放很多程序，这也就意味着操作系统需要管理多个进程，那么，为了协调各进程对系统资源的使用，进程管理还需要做一些与进程协调有关的其他管理工作，包括进程调度、进程间的数据共享、进程间执行的同步互斥关系（后续相关实验涉及）等。下面逐一进行解析。

#### 1. 资源管理

在计算机系统中，进程会占用内存和CPU，这都是有限的资源，如果不进行合理的管理，资源会耗尽或无法高效公平地使用，从而会导致计算机系统中的多个进程执行效率很低，甚至由于资源不够而无法正常执行。

对于用户进程而言，操作系统是它的“上帝”，操作系统给了用户进程可以运行所需的资源，最基本的资源就是内存和CPU。在实验二/三中涉及的内存管理方法和机制可直接应用到进程的内存资源管理中来。在有多个进程存在的情况下，对于CPU这种资源，则需要通过进程调度来合理选择一个进程，并进一步通过进程分派和进程切换让不同的进程分时复用CPU，执行各自的工作。对于无法剥夺的共享资源，如果资源管理不当，多个进程会出现死锁或饥饿现象。

#### 2. 进程状态管理

用户进程有不同的状态（可理解为“生命”的不同阶段），当操作系统把程序的放到内存中后，这个进程就“诞生”了，不过还没有开始执行，但已经消耗了内存资源，处于“创建”状态；当进程准备好各种资源，就等能够使用CPU时，进程处于“就绪”状态；当进程终于占用CPU，程序的指令被CPU一条一条执行的时候，这个进程就进入了“运行”状态，这时除了继续占用内存资源外，还占用了CPU资源；当进程由于等待某个资源而无法继续执行时，进程可放弃CPU使用，即释放CPU资源，进入“等待”状态；当程序指令执行完毕，由操作系统回收进程所占用的资源时，进程进入了“死亡”状态。

这些进程状态的转换时机需要操作系统管理起来，而且进程的创建和清除等服务必须由操作系统提供，而且在“运行”与“就绪”/“等待”状态之间的转换，涉及到保存和恢复进程的“执行现场”，也就是进程上下文，这是确保进程即使“断断续续”地执行，也能正确完成工作的必要保证。

#### 3. 进程与线程

一个进程拥有一个存放程序和数据的的虚拟地址空间以及其他资源。一个进程基于程序的指令流执行，其执行过程可能与其它进程的执行过程交替进行。因此，一个具有执行状态（运行态、就绪态等）的进程是一个被操作系统分配资源（比如分配内存）并调度（比如分时使用CPU）的单位。在大多数操作系统中，这两个特点是进程的主要本质特征。但这两个特征相对独立，操作系统可以把这两个特征分别进行管理。

这样可以把拥有资源所有权的单位通常仍称作进程，对资源的管理成为进程管理；把指令执行流的单位称为线程，对线程的管理就是线程调度和线程分派。对属于同一进程的所有线程而言，这些线程共享进程的虚拟地址空间和其他资源，但每个线程都有一个独立的栈，还有独立的线程运行上下文，用于包含表示线程执行现场的寄存器值等信息。

在多线程环境中，进程被定义成资源分配与保护的单位，与进程相关联的信息主要有存放进程映像的虚拟地址空间等。在一个进程中，可能有一个或多个线程，每个线程有线程执行状态（运行、就绪、等待等），保存上次运行时的线程上下文、线程的执行栈等。考虑到CPU有不同的特权模式，参照进程的分类，线程又可进一步细化为用户线程和内核线程。

到目前为止，我们就可以明确用户进程、内核进程（可把ucore看成一个内核进程）、用户线程、内核线程的区别了。从本质上看，线程就是一个特殊的不用拥有资源的轻量级进程，在ucore的调度和执行管理中，并没有区分线程和进程。且由于ucore内核中的所有内核线程共享一个内核地址空间和其他资源，所以这些内核线程从属于同一个唯一的内核进程，即ucore内核本身。理解了进程或线程的上述属性和特征，就可以进行进程/线程管理的设计与实现了。但是为了叙述上的简便，以下用户态的进程/线程统称为用户进程。