实验四：内核线程管理

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

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 的内存管理分配合适的空间，然后就需要考虑如何使用 CPU 来“并发”执行多个程序。

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。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。

2.1 练习

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 结构中的域至少包括：state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name。

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

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

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/

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，实验四主要增加的文件如上表红色部分所示，主要修改的文件如上表紫色部分所示。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
```

则可以得到如附录A所示的显示内容（仅供参考，不是标准答案输出）

3 内核线程管理

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。

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域：

- mm：内存管理的信息，包括内存映射列表、页表指针等。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。如果是内核线程的进程控制块，则 mm=NULL，因为它重用了 ucore 内核的页表。
- 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 中的域 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 中的域 list_link 将链接入这个链表中。

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，没有用户空间。而由于在操作系统启动后，已经对整个核心内存空间进行了管理，通过设置页表建立了核心虚拟空间（即 boot_cr3 指向的二级页表描述的空间）。所以内核中的所有线程都不需要再建立各自的页表，只需共享这个核心虚拟空间就可以访问整个物理内存了。

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—proc，这就是第 0 个进程控制块了，并把 proc 进行初步初始化（即把 proc_struct 中的各个域清零）。但有些域设置了特殊的值：

```
练习 1      //设置进程为“初始”态
练习 1      //进程的 pid 还没设置好
练习 1      //进程在内核中使用的内核页表的起始地址
```

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进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，这表示所有这些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。

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 "Hello world!!";
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。

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 中的相应域中。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 的高地址的域 ebp 开始，逐一把相关域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。

4 实验报告要求

从网站上下载 lab4.zip 后，解压得到本文档和代码目录 lab4，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab4-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

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。

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>
```

附录B: 【原理】进程的属性与特征解析

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。