Lab4

一、练习

练习 0：填写已有实验

本实验依赖实验 2/3。请把你做的实验 2/3 的代码填入本实验中代码中有“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）

```
// 初始化进程状态
proc->state = PROC_UNINIT;

// 初始化进程ID和运行次数
proc->pid = -1;
proc->runs = 0;

// 初始化内核栈、need_resched 和 parent
proc->kstack = 0;
proc->need_resched = 0;
proc->parent = NULL;

// 初始化内存管理字段
proc->mm = NULL;

// 初始化用于进程切换的上下文
memset(&(proc->context), 0, sizeof(struct context));

// 初始化陷阱帧
proc->tf = NULL;

// 初始化CR3寄存器和进程标志
proc->cr3 = boot_cr3;
proc->flags = 0;

// 初始化进程名称
memset(proc->name, 0, PROC_NAME_LEN + 1);
```

struct context context：储存进程当前状态，用于进程切换中上下文的保存与恢复。

需要注意的是，与 trapframe 所保存的用户态上下文不同，context 保存的是线程的当前上下文。这个上下文可能是执行用户代码时的上下文，也可能是执行内核代码时的上下文。

struct trapframe* tf ：用于保存中断或异常发生时的现场信息，包括被中断的指令地址、寄存器状态等。无论是用户程序在用户态通过系统调用进入内

核态，还是线程在内核态中被创建，内核态中的线程返回用户态所加载的上下文就是 struct trapframe* tf 。 所以当一个线程在内核态中建立，
则该新线程就必须伪造一个 trapframe 来返回用户态。

从用户态进入内核态会压入当时的用户态上下文 trapframe。
两者关系：以 kernel_thread 函数为例，尽管该函数设置了 proc- >trapframe，但在 fork 函数中的 copy_thread 函数里，程序还会设置 proc- >context。两个上下文看上去好像冗余，但实际上两者所分的工是不一样的。

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。因此，我们实际需要"fork"的东西就是 stack 和 trapframe。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。

```
//    ?. set ret vaule using child proc's pid
proc = alloc_proc();   //新创建的进程。
if (proc == NULL)
    goto fork_out;
proc->parent = current;

// 分配内核栈
if (setup_kstack(proc))
    goto bad_fork_cleanup_proc;

// 拷贝或者共享mm
if (copy_mm(clone_flags, proc))   //如果 clone_flags 为真，则共享内存结构；否则，复制内存结构。
    goto bad_fork_cleanup_kstack;

// 在新进程的内核栈顶部设置当前进程的上下文 context 和中断帧 tf
copy_thread(proc, stack, tf);

// 需要去使能中断
bool intr_flag;
local_intr_save(intr_flag);

// 获得当前进程的id
proc->pid = get_pid();
nr_process++;

// 新创建的进程插入到进程哈希列表中
hash_proc(proc);
list_add(&proc_list, &proc->list_link);

local_intr_restore(intr_flag);

// 当前进程设为 RUNNABLE
wakeup_proc(proc);

ret = proc->pid;  //返回新进程的 PID
```

练习 3：编写 proc_run 函数（需要编码）

proc_run 用于将指定的进程切换到 CPU 上运行。它的大致执行步骤包括：

- 检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换。
- 禁用中断。你可以使用 `/kern/sync/sync.h` 中定义好的宏 `local_intr_save(x)` 和 `local_intr_restore(x)` 来实现关、开中断。

- 切换当前进程为要运行的进程。
- 切换页表，以便使用新进程的地址空间。`/libs/riscv.h` 中提供了 `lcr3(unsigned int cr3)` 函数，可实现修改 CR3 寄存器值的功能。
- 实现上下文切换。`/kern/process` 中已经预先编写好了 `switch.S`，其中定义了 `switch_to()` 函数。可实现两个进程的 context 切换。
- 允许中断。

请回答如下问题：

在本实验的执行过程中，创建且运行了几个内核线程？

完成代码编写后，编译并运行代码：make qemu

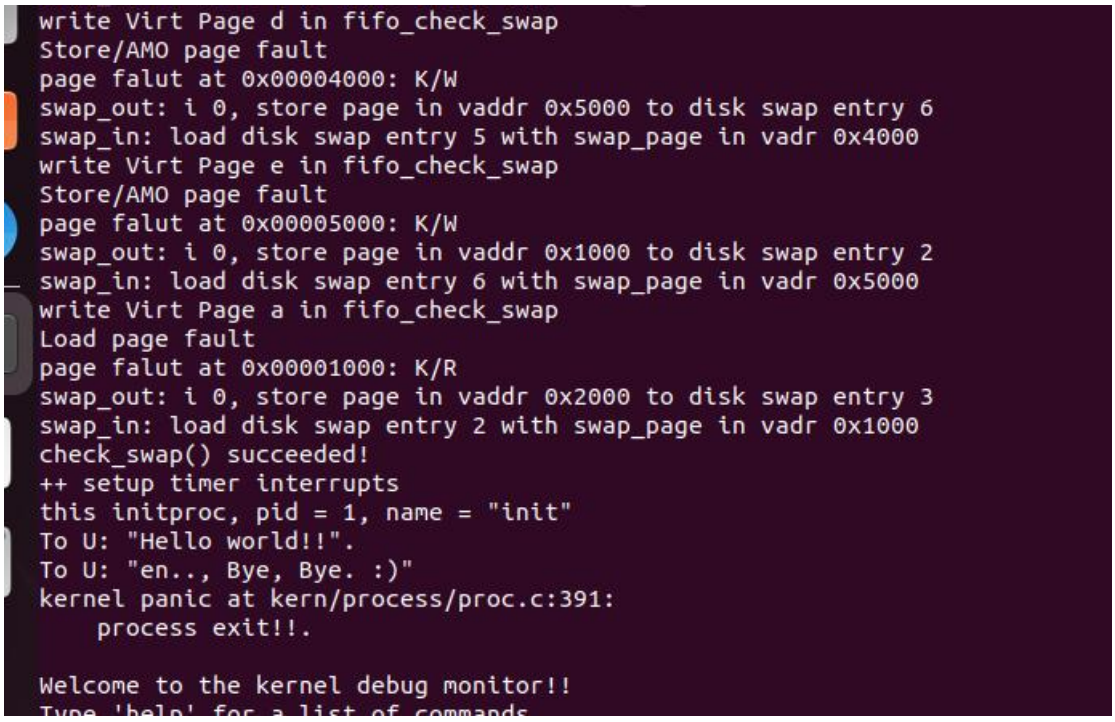
如果可以得到如 附录 A 所示的显示内容（仅供参考，不是标准答案输出），则基本正确。

```
void
proc_run(struct proc_struct *proc) {
    // 首先判断要切换到的进程是不是当前进程，若是则不需进行任何处理。
    if (proc != current) {
        // LAB4:EXERCISE3 2111642
        /*
        * Some Useful MACROs, Functions and DEFINEs, you can use them in below implementation.
        * MACROs or Functions:
        *   local_intr_save():        Disable interrupts
        *   local_intr_restore():     Enable Interrupts
        *   lcr3():                   Modify the value of CR3 register
        *   switch_to():              Context switching between two processes
        */

        // 调用local_intr_save和local_intr_restore函数禁用中断，避免在进程切换过程中出现中断。
        bool intr_flag;
        local_intr_save(intr_flag);

        struct proc_struct *prev = current;
        struct proc_struct *next = proc;

        // 将当前进程设为传入的进程
        current = proc;

        // 修改 esp 指针的值,设置任务状态段tss中的特权级0下的esp0指针为next内核线程的内核栈的栈顶
        load_esp0(next->kstack + KSTACKSIZE);

        // 修改页表项,重新加载 cr3 寄存器(页目录表基址) 进行进程间的页表切换
        lcr3(next->cr3);

        // 使用 switch_to 进行上下文切换。
        switch_to(&(prev->context), &(next->context));

        local_intr_restore(intr_flag);

    }
```

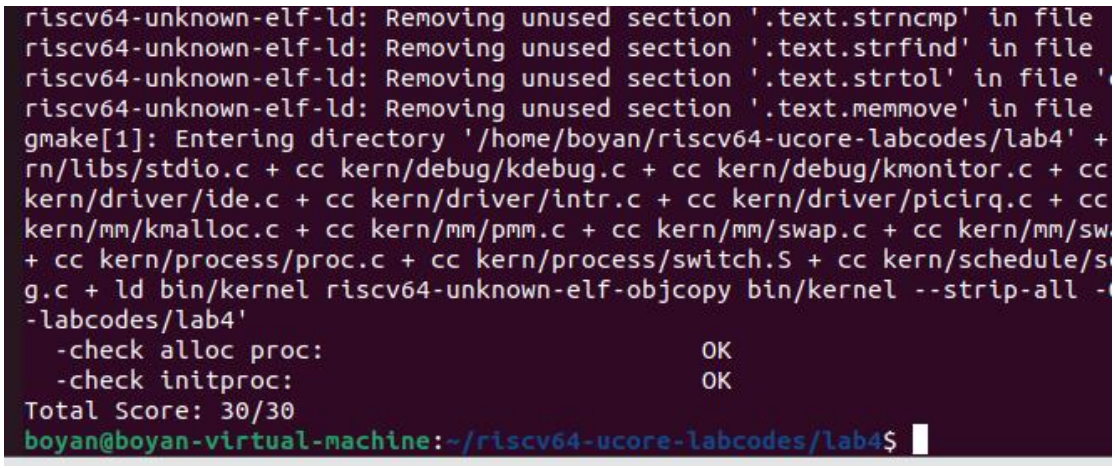
在 uCore 执行完 proc_init 函数后，就创建好了两个内核线程：idleproc 和 initproc，这时 uCore 当前的执行现场就是 idleproc，等到执行到 init 函数的最后一个函数 cpu_idle 之前，uCore 的所有初始化工作就结束了，idleproc 将通过执行 cpu_idle 函数让出 CPU，给其它内核线程执行

二、实验结果

make qemu



make grade



μCore 内存管理初始化流程

在 kernel.c/kern_init() 中调用 proc_init()函数

在 proc.c/proc_init()中:

初始化内核链表 proc_list

初始化进程控制块儿（proc_struct 结构体）idleproc

创建一个内核线程 init_main

μCore 创建进程、线程的流程分析

创建线程 ：

调用 do_fork() 函数

分配并初始化进程控制块儿（alloc_proc() 函数）

分配并初始化内核栈（setup_stack 函数，给内核栈分配一些页）

根据 clone_flag 标志复制或共享进程内存管理结构（copy_mm 函数）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）

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

自此，进程已经准备好执行了，把进程状态设置为“就绪”态

设置返回码为子进程的 id 号

μCore 进程、线程调度的流程分析

在 proc.c/cpu_idle() 中进行线程调度。内核的第一个进程 idleproc 会执行 cpu_idle 函数，并从中调用 schedule 函数，准备开始调度进程。

在 schedule() 函数中：

设置当前内核线程 current->need_resched 为 0

在 proc_list 队列中查找下一个处于“就绪”态的线程或进程 next

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

在 proc_run() 函数中：

让 current 指向 next 内核线程 initproc

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

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 函数执完“ret”指令后，就切换到 initproc 执行了

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创建并执行内核线程个进程了。