Lab 4

彭钰钊 2110756　姜涵 2113630　王健行 2111065

一、实验要求：

- 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
- 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

二、知识点整理：

内核线程与用户进程的区别

- 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

进程与线程

- 程序：源代码经过编译器变成的可执行文件
- 进程：程序装载进内存开始执行，包含程序内容，也包含“正在运行”的特性
- 线程：“正在运行”的部分(一个程序可以对应一个线程或多个线程)

> 这些线程之间往往具有相同的代码，共享一块内存，但是却有不同的CPU执行状态。相比于线程，进程更多的作为一个资源管理的实体（因为操作系统分配网络等资源时往往是基于进程的），这样线程就作为可以被调度的最小单元，给了调度器更多的调度可能

为什么需要进程：

（1）便于调度（否则所有的代码可能需要在操作系统编译的时候就打包在一块，安装软件将变成一件非常难的事情，这显然对于用户使用计算机是不利的）

（2）使用进程的概念有助于各个进程同时的利用CPU的各个核心，这是单进程系统往往做不到的。

（3）时间片轮转

三、实验练习：

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）

要分配并初始化一个进程控制块（PCB），首先需要了解一下我们是如何定义进程控制块的——`kern/process/proc.h`中我们定义了PCB的结构体`proc_struct`：

```
struct proc_struct {
    enum proc_state state;          // Process state 进程状态
    int pid;                        // Process ID 进程PID
    int runs;                       // the running times of Process 进程运行时间
    uintptr_t kstack;               // Process kernel stack 进程的内核栈
    volatile bool need_resched;     // bool value: 是否调度，1-调度；0-不调度
    struct proc_struct *parent;     // the parent process 父进程
    struct mm_struct *mm;           // Process's memory management field 进程内存调度
    struct context context;         // Switch here to run process 进程上下文
    struct trapframe *tf;           // Trap frame for current interrupt 中断帧指针
    uintptr_t cr3;                   // CR3 register: (PDT) 页表基址
    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
};
```

我们值得注意的几个成员变量是state、pid和cr3：

- state：表示进程状态，在我们本次实验中的定义如下：

  ```
  // process's state in his life cycle
  enum proc_state {
      PROC_UNINIT = 0,  // uninitialized 未初始化状态
      PROC_SLEEPING,    // sleeping 休眠状态
      PROC_RUNNABLE,    // runnable(maybe running) 运行状态
      PROC_ZOMBIE,      // almost dead, and wait parent proc to reclaim his resource
      // 僵尸状态
  };
  ```

- pid：进程唯一标识符，关于这个变量我们在下一问中详细说明
- cr3：记录当前进程页表基址

现在我们对于该函数的设计（见代码注释）和实现如下所示：

```
// alloc_proc - alloc a proc_struct and init all fields of proc_struct
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 Process
     *       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
     */
        // 参考实验指导手册：alloc_proc函数获取一块内存作为第0个进程控制块，并初始化（大体是将成员变量清零）
        proc->state = PROC_UNINIT;  // 设置进程为“未初始化”状态——即第0个内核线程（空闲进程idleproc）
        proc->pid = -1;  // 设置进程PID为未初始化值，即-1
        proc->runs = 0;  // 根据提示可知该成员变量表示进程的运行时间，初始化为0
        proc->kstack = NULL;  // 进程内核栈初始化为空【kstack记录了分配给该进程/线程的内核栈的位置】
        proc->need_resched = 0;  // 是否需要重新调度以释放 CPU？当然了，我们现在处于未初始化状态，不需要进行调度
        proc->parent = NULL;  // 父进程控制块指针，第0个进程控制块诶，它是始祖！
        proc->mm = NULL;  // 进程的内存管理字段:参见lab3练习一分析；对于内核进程而言，不存在虚拟内存管理
        memset(&(proc->context), 0, sizeof(struct context));  // 上下文，现在是源头，当然为空，发生切换时修改
        proc->tf = NULL;  // 进程中断帧，初始化为空，发生中断时修改
        proc->cr3 = boot_cr3;  // 页表基址初始化——在pmm_init中初始化页表基址，实际上是satp寄存器【X86历史残留，有点想改但是由于涉及文件相对较多，万一没修改完全就寄了，索性放弃】
        proc->flags = 0;  // 进程标志位，初始化为空
        memset(&(proc->name), 0, sizeof(struct context));  // 进程名初始化为空
    }
    return proc;
}
```

- `struct context context`是用于保存进程上下文的结构体，在该结构体中存储的是与进程上下文相关的寄存器值：

```
struct context {
    uintptr_t ra;  // 返回地址
    uintptr_t sp;  // 栈指针
    uintptr_t s0;  // 以下均为保存寄存器
    uintptr_t s1;
    uintptr_t s2;
    uintptr_t s3;
    uintptr_t s4;
    uintptr_t s5;
    uintptr_t s6;
```

```
    uintptr_t s7;
    uintptr_t s8;
    uintptr_t s9;
    uintptr_t s10;
    uintptr_t s11;
};
```

- `struct trapframe *tf`是进程中断帧的指针，总是指向内核栈的某个位置：
  - 当进程从用户态转移到内核态时，中断帧记录了进程在被中断前的状态,比如部分寄存器的值。
  - 当内核需要跳回用户态时，需要调整中断帧以恢复让进程继续执行的各寄存器值。

```
struct trapframe {
    struct pushregs gpr;  // 通用寄存器
    uintptr_t status;  // 状态
    uintptr_t epc;  // pc值
    uintptr_t badvaddr;  // 发生错误的地址
    uintptr_t cause;  // 错误原因
};
```

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。

关于do_fork函数的设计（见代码注释）与实现如下：

```
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's proc->tf
 */
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;  // 错误码：没有空闲进程
    struct proc_struct *proc;
```

```
    if (nr_process >= MAX_PROCESS) {  // 如果进程数量已达到上限，返回错误码，此时错误码表示没有空闲进程
        goto fork_out;
    }
    ret = -E_NO_MEM;  // 错误码：没有可分配内存
    //LAB4:EXERCISE2 YOUR CODE
    /*
     * 一些有用的 宏、函数和定义，你可以在下面的实现中使用它们。
     * 宏或函数：
     * alloc_proc：创建一个 proc 结构和初始化字段（lab4:exercise1）
     * setup_kstack：分配大小为 KSTACKPAGE 的页面作为进程内核栈
     * copy_mm：根据 clone_flags 复制或共享进程 "proc "的 mm，如果 clone_flags & CLONE_VM，则 "共享"；否则 "复制"。
     * copy_thread：在进程的内核堆栈顶部设置陷阱框架，并设置进程的内核入口点和栈
     * hash_proc: 将进程添加到进程 hash_list 中
     * get_pid：为进程分配唯一的 pid
     * wakeup_proc: 设置 proc->state = PROC_RUNNABLE
     * 变量：
     * proc_list：进程列表
     * nr_process：进程的数量
     */
    //    1. call alloc_proc to allocate a proc_struct 调用 alloc_proc 分配一个 proc_struct
    // 分析练习1中我们实现的进程分配函数，当返回值为NULL时是由于kmalloc(sizeof(struct proc_struct));的返回值为NULL
    // 而kmalloc函数是用于分配内存的函数，其返回值为NULL表示内存分配失败，此时错误码应该时表示内存问题，与我们前面ret = -E_NO_MEM;  // 错误码：没有可分配内存的设置一致，直接返回错误码
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;  // 子进程的父进程是当前进程
    //    2. call setup_kstack to allocate a kernel stack for child process 调用 setup_kstack 为子进程分配内核栈
    if (setup_kstack(proc) == -E_NO_MEM) {  // 检查进程内核栈分配是否成功（实际上复制了父进程的内核栈），如果返回-E_NO_MEM表示由于内存不足分配失败，我们需要处理已分配的子进程
        goto bad_fork_cleanup_proc;
    }
    //    3. call copy_mm to dup OR share mm according clone_flag 调用 copy_mm，根据 clone_flag 复制或共享 mm
    if (copy_mm(clone_flags, proc) != 0) {  // 本次实验中没有具体实现该函数功能，仅仅使用assert做判断模拟该函数错误情况，如果没有错误返回值为0，有错误那么我们需要释放初始化的子进程内核栈
        goto bad_fork_cleanup_kstack;
    }
    //    4. call copy_thread to setup tf & context in proc_struct 调用 copy_thread 在 proc_struct 中设置 tf 和 context
    copy_thread(proc, stack, tf);  // stack父节点的用户栈指针。果 stack==0，则表示fork一个内核线程。那么和esp没啥区别了吧，另外在risc-v的代码里看到X86遗迹真的好丑陋，应该是sp寄存器
    //    5. insert proc_struct into hash_list && proc_list 将 proc_struct 插入 hash_list && proc_list
    // hash_proc(proc);
    // list_add(&proc_list, &(proc->list_link));
    //【太天真了！会这么简单么？孩子，你太天真了！】
    // 现在让我们思考这样一种情况：当进程正在插入链表时，一个新的进程一脚踹开之前的进程，那么我们的前一个进程没插入，这...这不对吧？【竞争？】
```

```
    // 那么我们有什么办法可以避免这种问题么？好吧，这个问题意识到了，解决办法源于下一问
    // 去sync.h看看：
    // #define local_intr_save(x)      do { x = __intr_save(); } while (0)
    // #define local_intr_restore(x)   __intr_restore(x);
    /*
        【GPT】
        这是两个宏定义，用于在代码中禁用和恢复中断。这些宏的目的是保护临界区，防止在执行关键代码段时被中断打断。
        这两个宏的组合允许程序员在执行一段关键代码时禁用中断，以防止中断干扰。
        在关键代码执行完成后，通过调用 local_intr_restore 宏，恢复之前保存的中断状态，保持系统的正确性和可靠性。
        这样的操作通常在实现操作系统内核或驱动程序等系统级代码时会经常使用。

        另外一个神奇的问题：这个do-while是啥？？？简单说是程序健壮性，好！从来没有考虑的东西，下面是更加详细的解释
        【GPT】
        * 语法要求：
        do-while 结构确保宏定义始终是一个语句块，因此无论在何处使用它，都不会受到语法限制。
        在C语言中，do-while 是一种循环结构，它需要一个语句块作为循环体。通过使用 do-while (0)，即使宏在其他语法结构中被嵌套，也能够保证语法的正确性。
        这样的嵌套在实际应用中可能很少见，但是为了确保宏定义的通用性和安全性，这是一个良好的实践。

        * 强制使用花括号 {}：
        在C语言中，如果 do-while 循环体只有一条语句，可以不使用花括号。
        然而，这可能导致在某些上下文中产生意外的结果。
        通过使用 do-while (0)，宏定义始终需要花括号，从而避免了由于宏在某些上下文中被当作单个语句而导致的问题。
    */
    bool interrupt_flag;  // 判断是否禁用中断
    local_intr_save(interrupt_flag);  // copy_thread函数中tf的实参 是tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~SSTATUS_SIE;那么调用该函数将会禁用中断
    {  // 没别的意思，就是让下面这部分看起来更舒服，这是禁用中断后执行的一块代码
        proc->pid = get_pid();  // 获取当前pid
        hash_proc(proc);
        list_add(&proc_list, &proc->list_link);  // 这才是正确的打开方式（bushi）
        nr_process++;  // 更新进程数
    }
    local_intr_restore(interrupt_flag);  // 恢复之前的中断状态；有借有还呢
    //    6. call wakeup_proc to make the new child process RUNNABLE 调用 wakeup_proc 使新的子进程可运行
    wakeup_proc(proc);
    //    7. set ret vaule using child proc's pid 使用子进程的 pid 设置 ret vaule
    ret = proc->pid;

    fork_out:
        return ret;  // 正确情况下返回子进程pid，否则返回错误码

    bad_fork_cleanup_kstack:
        put_kstack(proc);
    bad_fork_cleanup_proc:
        kfree(proc);
        goto fork_out;
}
```

要回答ucore是否做到给每个新fork的线程一个唯一的id，我们来看看get_pid函数的实现：

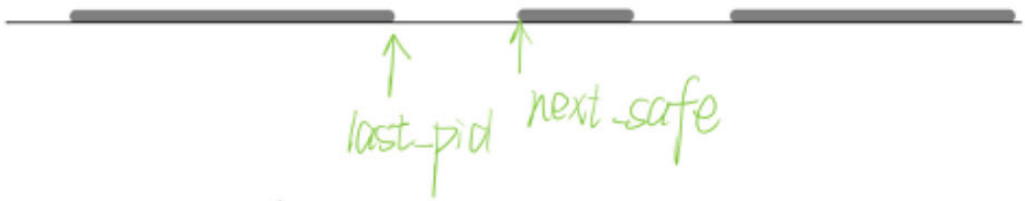
```
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);  // 用于在编译时检查 MAX_PID 是否大于 MAX_PROCESS，以确保 PID 的范围足够大以覆盖所有可能的进程。
    struct proc_struct *proc;  // 遍历进程链表时会使用的指针
    list_entry_t *list = &proc_list, *le;  // 遍历进程链表
    static int next_safe = MAX_PID, last_pid = MAX_PID;  // next_safe 用于保存下一个安全的 PID，last_pid 用于保存上一个分配的 PID。
    if (++ last_pid >= MAX_PID) {  // 如果递增后的 last_pid 超过或等于 MAX_PID，则将其重置为1，然后跳转到 inside 标签。
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe) {  // 如果 last_pid 大于等于 next_safe，则执行以下操作：
    inside:
        next_safe = MAX_PID;  // 将 next_safe 重置为 MAX_PID
    repeat:
        le = list;
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) {  //  如果当前进程的 PID 等于 last_pid，表示该 PID 已经被使用。
                if (++ last_pid >= next_safe) {  // 如果递增后的 last_pid 超过或等于 next_safe，则执行以下操作：
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;  // 跳转到 repeat，重新检查进程链表
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid) {  // 如果当前进程的 PID 大于 last_pid 且小于 next_safe，则更新 next_safe 为当前进程的 PID
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

我们定义PID最初的可用区间为[0,MAX_PID)，其中MAX_PID=2*MAX_PROCESS，即我们可用的PID数目是线程数的两倍，这保证了我们在使用的线程一定会有一个可用的PID。在该函数中我们定义了两个变量next_safe（可用PID区间右边界）和last_pid（可用PID区间左边界），即我们每次分配PID时都是在区间[last_pid,next_safe)中去依次递增，并且不断的调整该区间：

- 当我们的变量last_pid增加到区间最右端点后，表示我们不能继续通过递增来获取pid，此时我们将该变量赋值为1，则我们现在将在区间[1,MAX_PID)之间遍历进程链表寻找可用pid
- 我们在遍历的过程中找到第一个与可用区间左端点相同的的pid，表示当前pid已被使用，然后判断last_pid+1是否超出区间右端点next_safe，如果超出则先判断是否与第一步类似，如果类似则做相同的处理，否则，将右端点移至最大值MAX_PID重复遍历过程；那么如果为超出右端点，则直接返回last_pid，此时这个变量存储的就是我们要分配的pid

- 我们现在已经解释了当我们找到第一个与可用区间左端点相同的的pid时的处理方式，那么我们在未找到该pid的时候该如何处理？判断当前pid是否在可用区间[last_pid,next_safe)内，如果当前pid在该区间内则更新右端点为当前pid
- 以上是我们在最初的时候的操作以及分配PID一轮之后的操作，那么其他时候将需要判断我们是否存在可用区间，即可用区间非空——左端点小于右端点，如果为空，则按照我们上面第二步和第三步重新寻找可用区间；否则，直接返回last_pid，此时这个变量存储的就是我们要分配的pid

最初始的状态：我们此时的可用PID区间是[1, MAX_PID)，当我们运行一段时间后这个区间可能会变成下面这样



几个分支的示意：

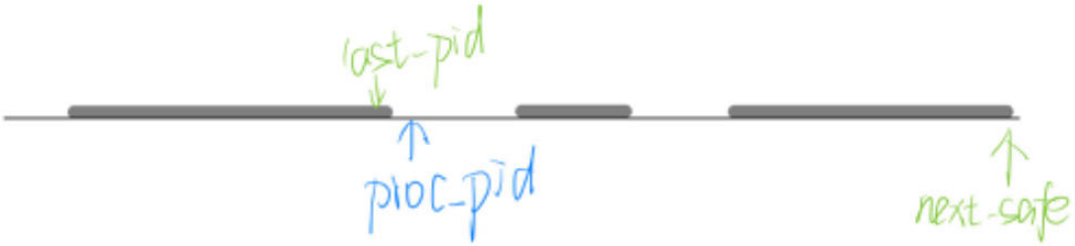
i. ++last_pid ≥ MAX_PID：

a. 最开始的情况

b. 一轮分配

ii. ++last_pid ≥ next_safe

iii. proc_pid 在区间内



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？

为了厘清该实验中ucore切换进程的原理，我们观察程序，从`proc_init()`函数开始。

在`proc_init`函数中，我们调用`alloc_proc()`函数，为`idleproc`进程分配了PCB块并进行了初始化操作，把`idleproc`进程的PCB的`need_resched`设置为1，并让current指向该PCB。

随后使用`kernel_thread`函数创建了`init`这一内核线程。在`kernel_thread`中，先设置了`trapframe`，对中断帧（记录了进程在被中断前的状态）的内容进行了巧妙的设置(后续解释)，调用`do_fork`函数，把设置好的中断帧tf作为参数，用于创建新进程。

`do_fork` 中使用 `alloc_proc` ，为`init`这一进程分配了PCB。`do_fork`的设计与实现在练习2中进行了详细说明。为了设置`init`新进程的`trapframe`、内核栈和`上下文（context）`,调用了`copy_thread`函数。该函数让新进程的`ra`寄存器(返回地址)指向`forkret`函数，`sp`寄存器指向新进程的`trapframe`，随后设置新线程的pid，再调用`wakeup_proc`将进程状态设置为`PROC_RUNNABLE`。随后设置了进程名，通过断言保证了idle进程和init进程的pid。至此`proc_init`函数执行工作完成，**创建了两个内核线程（idle线程与init线程）。**

在init.c中，kernel初始化时，一系列的初始化函数执行结束后，cpu_idle函数开始执行。该函数内容如下：

```
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

结合上述内容，我们知道`current`进程是`idleproc`进程。上面我们提到，该进程的PCB的`need_resched`为1，因此会执行`schedule`函数。`schedule`则会先把`idleproc`进程的`need_resched`设置为0，然后寻找`PROC_RUNNABLE`的进程，并且使用`proc_run`执行这一进程。上面我们提到，在`do_fork`函数中，我们把`init`进程状态设置为`PROC_RUNNABLE`，所以会用`proc_run`执行`init`进程。因此，**idle线程与init线程都进行了运行，本实验的执行过程中，创建且运行了2个内核线程。**

`proc_run`的具体实现如下：

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
```

```
void
proc_run(struct proc_struct proc) {
    if (proc != current) {
        // LAB4:EXERCISE3 YOUR CODE
        /*
        * 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
        */
        bool intr_flag;
        struct proc_struct *prev = current, next = proc;
        //禁用中断，保护进程切换不会被中断，以免进程切换时其他进程再进行调度
        local_intr_save(intr_flag);{
            //进程切换
            current = proc;    //让current指向next内核线程initproc
            lcr3(next->cr3);   //cr3寄存器改为需要运行进程的页目录表
            switch_to(&(prev->context), &(next->context));//上下文切换
switch.s
        }
        //开中断
        local_intr_restore(intr_flag);
    }
}
```

我们可以看到，该函数先检查要切换到的目标进程是不是当前运行的进程，若相同则不执行操作，否则先禁用中断，保证进程切换过程可以顺利进行。在切换过程中，让`current`指针指向要切换的目标进程，`cr3`寄存器改为需要运行进程的页目录表，随后调用`switch_to`函数完成进程的上下文切换。上下文切换结束后，恢复允许中断，完成了进程的切换。我们再来看上下文切换的原理。

```
.globl switch_to
switch_to:
    # save from's registers
    STORE ra, 0*REGBYTES(a0)
    STORE sp, 1*REGBYTES(a0)
    STORE s0, 2*REGBYTES(a0)
    STORE s1, 3*REGBYTES(a0)
    STORE s2, 4*REGBYTES(a0)
    STORE s3, 5*REGBYTES(a0)
    STORE s4, 6*REGBYTES(a0)
    STORE s5, 7*REGBYTES(a0)
    STORE s6, 8*REGBYTES(a0)
    STORE s7, 9*REGBYTES(a0)
    STORE s8, 10*REGBYTES(a0)
    STORE s9, 11*REGBYTES(a0)
    STORE s10, 12*REGBYTES(a0)
    STORE s11, 13*REGBYTES(a0)
    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
```

```
    LOAD s3, 5*REGBYTES(a1)
    LOAD s4, 6*REGBYTES(a1)
    LOAD s5, 7*REGBYTES(a1)
    LOAD s6, 8*REGBYTES(a1)
    LOAD s7, 9*REGBYTES(a1)
    LOAD s8, 10*REGBYTES(a1)
    LOAD s9, 11*REGBYTES(a1)
    LOAD s10, 12*REGBYTES(a1)
    LOAD s11, 13*REGBYTES(a1)
    ret
```

我们看到，在上下文切换时，`switch_to`函数首先保存先前执行进程的ra，sp和多个通用寄存器的值到`prev`线程的上下文中，随后在`next`进程的上下文中找到这些寄存器的值，加载到CPU的寄存器中。随后执行`ret`，返回到之前设置的`ra`的地址。

我们在创建这个`next`进程（在此为`init`进程）时，使用`copy_thread`函数把进程控制块的`ra`设置成了设定成了`forkret`函数的地址。

```
// proc.c中copy_thread函数
proc->context.ra = (uintptr_t)forkret;
//  proc.c中
static void forkret(void) { forkrets(current->tf);}
# trapentry.S中
    .globl forkrets
forkrets:
    # set stack to this new process's trapframe
    move sp, a0
    j __trapret
    .globl __trapret
__trapret:
    RESTORE_ALL
    # go back from supervisor call
    sret
```

我们看到，`switch_to`交换上下文结束后，由于返回地址是`forkret`函数，因此与执行`move sp, a0`。这里把`a0`寄存器(函数参数，即`current->tf`)的值给到sp，就是该`next`进程（`init`进程）的中断帧给到`sp`。随后执行了`RESTORE_ALL`，恢复了`trapframe`中的所有的上下文，之后使用`sret`指令返回 pc对应的地址。

```
 // proc.c中kernel_thread函数
 tf.epc = (uintptr_t)kernel_thread_entry;
```

我们可以看到，恢复tf上下文后，返回到`kernel_thread_entry`的地址。

```
# entry.S中
.text
.globl kernel_thread_entry
kernel_thread_entry: # void kernel_thread(void)
move a0, s1
jalr s0
jal do_exit
  // proc.c中kernel_thread函数
  int kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
  ……
  tf.gpr.s0 = (uintptr_t)fn; // s0 寄存器保存函数地址
```

```
    tf.gpr.s1 = (uintptr_t)arg; // s1 寄存器保存函数参数地址
    ……
    }
    // proc.c中的的void proc_init(函数
    int pid = kernel_thread(init_main, "Hello world!!", 0);
```

`kernel_thread_entry`中，将此前设置在`tf`中的参数位置`s1`转移到`a0`，并且跳转到`s0`对应的地址处，即`kernel_thread`调用时传入的`fn`，此处即为`init_main`函数的地址，所以恢复tf后，会到`init_main`函数的地址，参数为`"Hello world!!"`

至此，成功完成了上下文的切换，`init`线程正确执行目标函数。

最终运行结果：

```
  -check alloc proc:                         OK
  -check initproc:                           OK
Total Score: 30/30
yuzhao-peng@yuzhaopeng-virtual-machine:~/NKU-OS/riscv64-ucore-labcodes/lab4$
```

扩展练习 Challenge：

- 说明语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`是如何实现开关中断的?

语句`local_intr_save(intr_flag);`和`local_intr_restore(intr_flag);`通过“屏蔽中断（interrupt masking）”技术来实现开关中断。在这种技术中，中断被分为不同的优先级。处理器可以根据中断的优先级来决定哪些中断应该被屏蔽（禁用），哪些中断可以被处理。当中断被屏蔽时，处理器将不会响应该中断的信号。

- 函数`local_intr_save(intr_flag);`:将当前处理器的中断状态保存到一个变量（如`intr_flag`）中，然后将处理器的中断状态设置为禁止状态。这样即使有中断信号到来，处理器也不会响应任何中断请求，就可以防止在某些关键的代码段执行期间被中断打断。
- 函数`local_intr_restore(intr_flag);`：将之前保存的中断状态（即`intr_flag`）恢复到处理器中，从而允许中断再次被处理。

通过在关键代码段的开头调用`local_intr_save(intr_flag);`来禁止中断，在代码段的末尾调用`local_intr_restore(intr_flag);`来恢复中断状态，可以确保在关键代码段执行期间不会被中断干扰。

本次实验中的具体实现(在proc_run函数中,分析见下面代码注释):

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;//定义一个bool类型的变量intr_flag，用于保存当前中断状态
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag); //调用local_intr_save(intr_flag)函数，将当前中断状态保存到intr_flag中，并禁止中断。
        {//进入一个代码块，使用花括号({})括起来的部分
            current = proc;
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
    local_intr_restore(intr_flag);//根据intr_flag中保存的状态恢复中断。
    }
}
```

其中local_intr_save和local_intr_restore在sync.h中定义实现，具体见以下代码注释：

```c
//一个同步机制相关的头文件，定义了一些用于中断保存和恢复的宏和内联函数
#ifndef __KERN_SYNC_SYNC_H__
#define __KERN_SYNC_SYNC_H__

#include <defs.h>
#include <intr.h>
#include <riscv.h>

static inline bool __intr_save(void) {//用于保存中断状态并禁用中断
    if (read_csr(sstatus) & SSTATUS_SIE) {//首先检查当前的 sstatus 寄存器的值是否设置了 SIE（中断使能）位
        intr_disable();
        return 1;
    }//如果是，则禁用中断，并返回1，表示中断被保存
    return 0;
}//否则，返回0，表示中断未保存。

static inline void __intr_restore(bool flag) {//用于恢复中断状态
    if (flag) {//如果之前的中断被保存了（即传入的 flag 为真），则调用 intr_enable 函数重新启用中断。
        intr_enable();
    }
}

//local_intr_save 宏用于保存中断状态，并将保存的结果存储在给定的变量 x 中。
#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
#define local_intr_restore(x) __intr_restore(x);
//local_intr_restore 宏用于恢复中断状态。
#endif /* !__KERN_SYNC_SYNC_H__ */
```