南开大学操作系统实验报告

学号：2110598 姓名：许宸

学号：2113384 姓名：刘新宇

学号：2112487 姓名：刘轩宇

实验题目 — Lab 4 进程管理

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

环境

软件环境

ubuntu22.04，QEMU-4.1.1

实验步骤与内容

（一） 练习 1：分配并初始化一个进程控制块

`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。

```
// 建立进程控制块
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;  // 设置进程状态为未初始化
    proc->pid = -1; // 进程 ID
    proc->runs = 0; // 进程运行次数
    proc->kstack = 0;   // 进程内核栈
    proc->need_resched = 0; // 是否需要重新调度
    proc->parent = NULL;    // 父进程
    proc->mm = NULL;    // 进程所用的虚拟内存
    memset(&(proc->context), 0, sizeof(struct context)); // 进程的上下文
    proc->tf = NULL; // 中断帧指针
    proc->cr3 = boot_cr3; // 页目录表地址 设为 内核页目录表基址
    proc->flags = 0; // 标志位
    memset(&(proc->name), 0, PROC_NAME_LEN); // 进程名
    }
    return proc;
}
```

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?

1. `struct context context`:
    - 该成员表示进程的上下文，包括各种寄存器的值，这些寄存器定义了进程的状态。
    - 这些寄存器包括 `ra`（返回地址）、`sp`（栈指针）以及 `s0` 到 `s11`（保存的寄存器）。
    - 在上下文切换期间，保存了当前执行进程的这些寄存器的值，并且会在切换到其他进程时将这些值恢复。使操作系统能够在不同进程之间进行切换，从而实现并发执行。
2. `struct trapframe *tf`:
    - 该成员表示当前中断或异常处理的陷阱帧。
    - 陷阱帧是一个数据结构，保存了处理中断或异常时处理器的状态。它包括各种寄存器的值、程序计数器以及其他在中断或异常发生时需要的信息。
    - 当发生中断或异常时，操作系统将当前进程的状态保存在其陷阱帧中。稍后，在处理完中断或异常并计划再次运行进程时，从陷阱帧中恢复保存的状态。

在提供的代码和操作系统内核的实现中:

- `struct context context` 用于保存和恢复通用寄存器以及上下文执行期间的执行上下文。
- `struct trapframe *tf` 用于存储处理中断或异常时处理器的状态，以便内核能够处理这些事件并稍后恢复中断的进程。

（二）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。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

```
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
     *   wakeup_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 wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid
    if ((proc = alloc_proc()) == NULL)  // 分配并初始化进程控制块
        goto fork_out;
    if (setup_kstack(proc) != 0)    // 分配并初始化内核栈
```

```
        goto bad_fork_cleanup_proc;
    if (copy_mm(clone_flags, proc) != 0)    // 根据 clone_flags 决定是复制还是共享内存管理系统（copy_mm 函数）
        goto bad_fork_cleanup_kstack;
    copy_thread(proc, stack, tf);   // 复制父进程的中断帧和上下文，设置子进程的中断帧和上下文
    proc->pid = get_pid();  // 分配进程 ID
    nr_process++;   // 进程数加一
    hash_proc(proc);    // 将进程控制块链接到哈希表中
    list_add_before(&proc_list, &proc->list_link);  // 将进程控制块链接到进程控制块链表中
    wakeup_proc(proc);  // 将进程状态设置为 PROC_RUNNABLE，表示进程可以运行
    ret = proc->pid;    // 返回子进程的进程 ID
    // 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。
fork_out:
    return ret;
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

ucore是否做到给每个新fork的线程一个唯一的id?

是的，ucore通过get_pid函数(通过递增last_pid并遍历进程链表来查找未被使用的ID，保证了分配的ID在有效范围内(1到MAX_PID-1))，为每个新的fork的线程分配一个唯一的id。

（三）练习 3：编写 proc_run 函数

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：

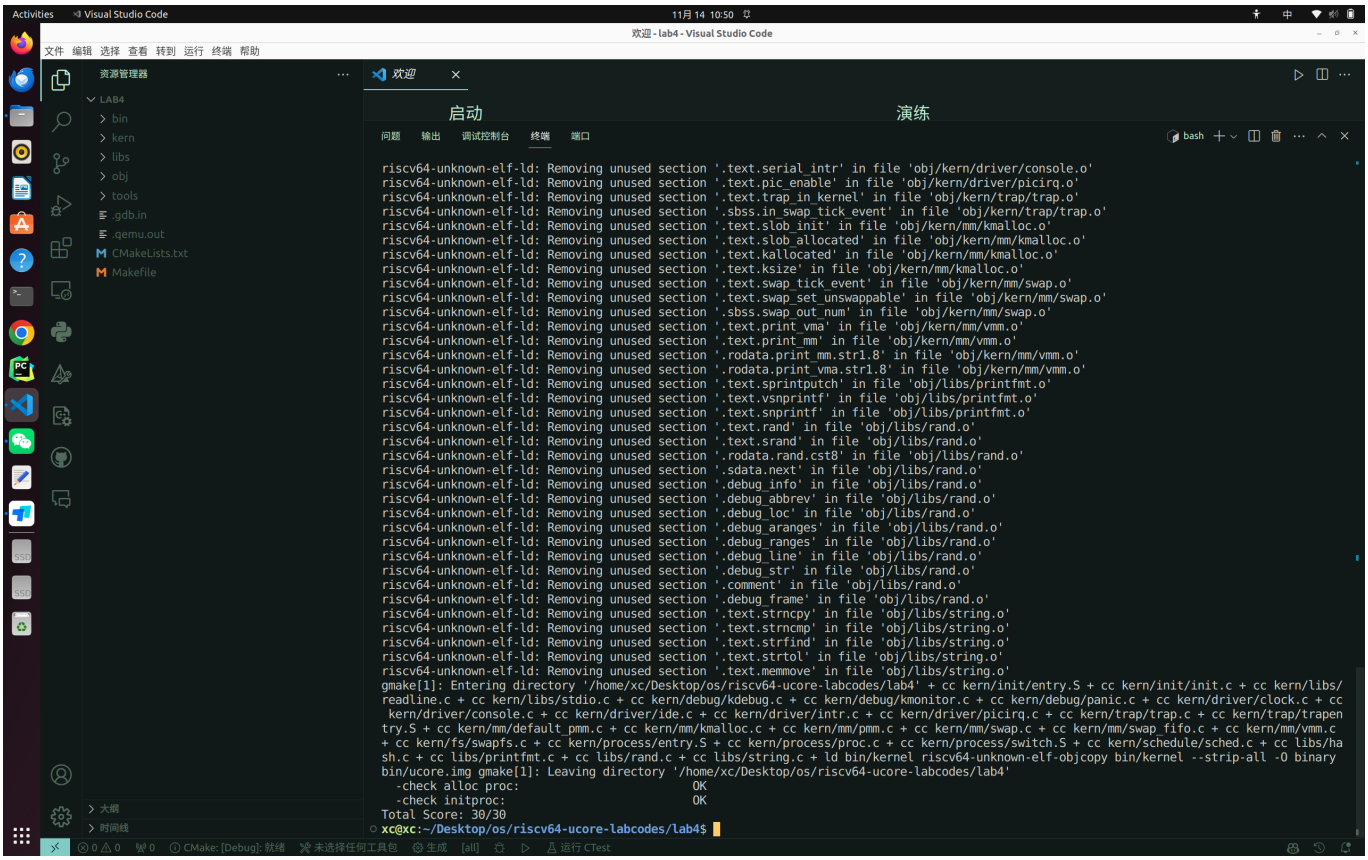
- 检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换。
- 禁用中断，使用/kern/sync/sync.h中定义好的宏local_intr_save(x)和local_intr_restore(x)来实现关、开中断。
- 切换当前进程为要运行的进程。
- 切换页表，以便使用新进程的地址空间。/libs/riscv.h中提供了lcr3(unsigned int cr3)函数，可实现修改CR3寄存器值的功能。
- 实现上下文切换。/kern/process中已经预先编写好了switch.S，其中定义了switch_to()函数。可实现两个进程的context切换。
- 允许中断。

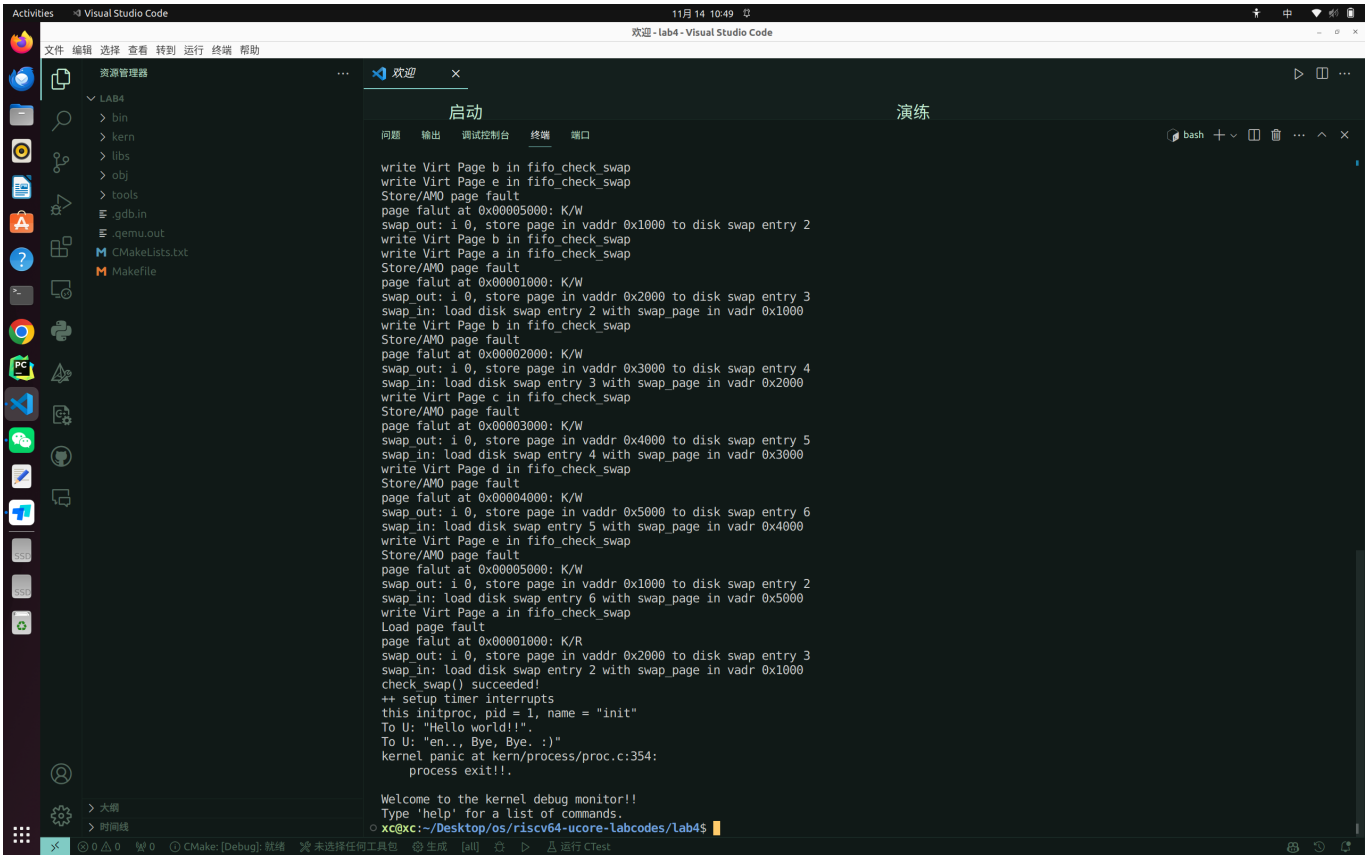
```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        // LAB4:EXERCISE3 YOUR CODE
        /*
```

```
        * 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;
        local_intr_save(intr_flag); // 关闭中断
        {
            current = proc; // 将当前进程换为 要切换到的进程
            // 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);
            lcr3(next->cr3); // 重新加载 cr3 寄存器(页目录表基址) 进行进程间的页表切换
            switch_to(&(prev->context), &(next->context)); // 调用 switch_to 进行上
下文的保存与切换
        }
        local_intr_restore(intr_flag);
    }
}
```

在本实验的执行过程中，创建且运行了几个内核线程?

创建了`idleproc`和`initproc`两个内核线程。





（四） 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?

1. `local_intr_save(x)`：

- 这个宏将当前中断状态保存在变量 `x` 中。
- `__intr_save()` 函数是一个内联函数，用于检查中断（异常）是否已启用（`sstatus` 寄存器中设置了 `SSTATUS_SIE` 位）。
- 如果中断已启用，它使用 `intr_disable()` 禁用中断，并返回1，表示中断原本是启用状态。
- 如果中断未启用，它返回0，表示中断原本已经禁用。
- 结果（1 或 0）存储在变量 `x` 中。

2. **`local_intr_restore(x)`**：

- 这个宏根据变量 `x` 中存储的值来恢复中断状态。
- `__intr_restore(bool flag)` 函数以保存的中断状态作为参数。
- 如果保存的状态（`flag`）为真（表示中断原本是启用状态），则使用 `intr_enable()` 启用中断。
- 如果保存的状态为假（表示中断原本已禁用），则不执行任何操作。

这些语句使用 `intr_disable` 和 `intr_enable` 函数来控制中断状态。

两个函数 `intr_enable` 和 `intr_disable`，用于在RISC-V架构上启用和禁用中断。

1. **`intr_enable` 函数**：

- 该函数使用 `set_csr(sstatus, SSTATUS_SIE)` 启用中断。
- `sstatus` 寄存器中的 `SIE` 位用于控制中断的全局使能，设置为1表示启用中断。

2. **`intr_disable` 函数**：

- 该函数使用 `clear_csr(sstatus, SSTATUS_SIE)` 禁用中断。
- `sstatus` 寄存器中的 `SIE` 位用于控制中断的全局使能，清除为0表示禁用中断。

这两个函数在处理器级别上提供了对中断的基本控制，允许系统在需要的时候启用或禁用中断。