Lab4：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

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

在ucore的调度和执行管理中，**对线程和进程做了统一的处理**。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。

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）

我们根据指导书给出的提示，在alloc_proc函数中也给出提示，根据这两个提示进行编写，编写的初始化代码如下：

```
1  static struct proc_struct *
2  alloc_proc(void) {
3      struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
4      if (proc != NULL) {
5      //LAB4:EXERCISE1 2210751
6      /*
7       * below fields in proc_struct need to be initialized
8       *       enum proc_state state;                      // Process state
9       *       int pid;                                    // Process ID
10      *       int runs;                                   // the running times
    of Proces
11      *       uintptr_t kstack;                           // Process kernel
    stack
12      *       volatile bool need_resched;                 // bool value: need
    to be rescheduled to release CPU?
13      *       struct proc_struct *parent;                 // the parent process
14      *       struct mm_struct *mm;                       // Process's memory
    management field
15      *       struct context context;                     // Switch here to run
    process
16      *       struct trapframe *tf;                       // Trap frame for
    current interrupt
17      *       uintptr_t cr3;                              // CR3 register: the
    base addr of Page Directroy Table(PDT)
18      *       uint32_t flags;                             // Process flag
19      *       char name[PROC_NAME_LEN + 1];               // Process name
20      */
21      proc->state = PROC_UNINIT; //状态设置为未初始化
22      proc->pid = -1; //ID都默认为-1
23      proc->runs = 0; //运行次数默认为0
24      proc->kstack = 0; //除了0号进程全需要后续分配
25      proc->need_resched = 0; //不需要切换线程
26      proc->parent = NULL; //没有父进程
27      proc->mm = NULL; //一开始创建未分配内存
```

```
28      memset(&(proc->context), 0, sizeof(struct context));//将上下文变量全部赋值为
    0，清空
29      proc->tf = NULL; //初始化没有中断帧
30      proc->cr3 = boot_cr3; //内核线程的cr3为boot_cr3，即页目录为内核页目录表
31      proc->flags = 0; //标志位设置为0
32      memset(proc->name, 0, PROC_NAME_LEN+1); //将线程名变量全部赋值为0，清空
33
34      }
35      return proc;
36  }
```

---

proc_struct中的成员变量`struct context context`：该变量表示该进程的上下文，用于进程切换时进行保存，通过我们看switch.S代码看出，这部分主要存储几个“被调用寄存器的信息”。观察`proc_run`函数看出，在将指定的进程切换到CPU上进行执行时，需要调用`switch_to`函数，将原进程的寄存器状态保存，以便下次切换读出，保证了进程的正确切换

proc_struct中的成员变量`struct trapframe *tf`：该变量表示进程的中断帧，保存了32个通用寄存器及、程序计数器、状态寄存器、异常原因寄存器等的信息。它是在进程进入内核模式时进行信息的保存，内核中断处理完成后，进程便会根据tf中的信息恢复进程状态。在本实验中，利用`s0`、`s1`寄存器传递线程执行的函数指针和函数参数；利用`epc`寄存器恢复到要执行的位置，执行了`init_main`，还在创建子进程时，将子进程的`tf.a0`寄存器设置为0，作为子进程的标识。

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。

我们根据练习2给出的大致执行步骤，再通过研究相关代码后，编写出 `do_fork` 函数

```
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 2210751
    /*
     * 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

    //调用alloc_proc，首先获得一块用户信息块
    proc = alloc_proc();
```

```
37      //分配失败
38      if(!proc)
39      {
40          goto fork_out;
41      }
42
43      //设置当前进程为新进程的父进程,为了copy_mm
44      proc->parent = current;
45
46      //为进程分配一个内核栈。
47      if(setup_kstack(proc))
48      {
49          goto bad_fork_cleanup_kstack;//跳转进行清理
50      }
51
52      //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）
53      //复制进程的内存布局信息，以确保新进程拥有与原进程相同的内存环境
54      if(copy_mm(clone_flags,proc))
55      {
56          goto bad_fork_cleanup_proc;//失败则进行清理
57      }
58
59      //复制原进程上下文到新进程
60      copy_thread(proc,stack,tf);
61
62      //get_pid中的全局变量需要原子性的更改，禁止中断
63      bool intr_flag;
64      local_intr_save(intr_flag);
65
66      //为新进程分配一个的进程号(唯一)
67      proc->pid = get_pid();
68
69      //将新进程添加到进程列表，并允许中断
70      hash_proc(proc);
71      list_add(&proc_list,&(proc->list_link));//将proc->list_link加到proc_list后
72      nr_process ++;//更新进程数量计数器
73      local_intr_restore(intr_flag);
74
75      //唤醒新进程
76      wakeup_proc(proc);
77
78      //返回新进程号
79      ret = proc->pid;
80
81  fork_out:
82      return ret;
83
```

```
84 bad_fork_cleanup_kstack:
85     put_kstack(proc);
86 bad_fork_cleanup_proc:
87     kfree(proc);
88     goto fork_out;
89 }
```

请说明ucore是否做到给每个新fork的线程一个唯一的id：

ucore做到了给每个新fork的线程一个唯一的id。我们观察get_pid函数，定义了两个静态变量 `next_safe` 和 `last_pid` ，这两个变量用于追踪下一个安全的PID和最后一个分配的PID。该函数主要流程为：首先判断last_pid如果小于 `next_safe` ，则分配的 `last_pid` 一定是唯一的。若 `last_pid` 大等于 `next_safe` 或大于 `MAX_PID` ，则需要进一步遍历proc_list重新设置 `last_pid` 和 `next_safe` ，以便下一次函数循环时正常分配。即分配唯一的id依靠如下：1.遍历检查避免重复id 2.循环回收：如果大于MAX_PID则置1，利用较小的PID 3.在遍历检查时，记录比当前 `last_pid` 更大的最小 `proc->pid` 作为 `next_safe` ，会缩小后续搜索范围，加速分配。这样通过遍历检查和循环回收的机制保证了每个新fork的线程有唯一的id。 `get_pid` 函数代码如下：

```
1  // get_pid - alloc a unique pid for process
2  static int
3  get_pid(void) {
4      static_assert(MAX_PID > MAX_PROCESS);//确保足够大
5      struct proc_struct *proc;
6      list_entry_t *list = &proc_list, *le;
7      static int next_safe = MAX_PID, last_pid = MAX_PID;//last_pid从MAX_PID开始使
   用
8      if (++ last_pid >= MAX_PID) {
9          last_pid = 1;
10         goto inside;
11     }
12     //last_pid比next_safe大，寻找一个安全的PID
13     if (last_pid >= next_safe) {
14     inside:
15         next_safe = MAX_PID;
16     repeat:
17         le = list;
18         //遍历检查每一个进程PID
19         while ((le = list_next(le)) != list) {
20             proc = le2proc(le, list_link);
21             //如果发现已有进程的 PID 与当前 last_pid 冲突：将 last_pid 自增并重新检
   查，直到找到一个未被占用的 PID。
22             if (proc->pid == last_pid) {
23                 if (++ last_pid >= next_safe) {
```

```
                    //如果超过最大值 MAX_PID，从 1 开始重新分配（避免溢出）
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            //如果发现进程的 PID 大于当前 last_pid，更新 next_safe
            else if (proc->pid > last_pid && next_safe > proc->pid) {
                next_safe = proc->pid;
            }
        }
    }
    return last_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。

我们根据练习3给出的大致执行步骤，再通过研究相关代码后，编写出 `proc_run` 函数。

```
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
```

```
10          *   lcr3():                 Modify the value of CR3 register
11          *   switch_to():            Context switching between two processes
12          */
13         bool intr_flag;
14         struct proc_struct *prev = current, *next = proc;
15         local_intr_save(intr_flag);
16         {
17             current = proc;
18             lcr3(proc->cr3);
19             switch_to(&(prev->context), &(next->context));
20         }
21         local_intr_restore(intr_flag);
22
23     }
24 }
```

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？

答：两个

1.第0个内核线程---idleproc，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

2.第1个内核线程---initproc，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。

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？

答：我们前往 `/kern/sync/sync.h` 文件分析源代码。

禁用中断： 在需要保护临界区（例如操作共享资源或执行重要的内存管理操作）时，调用 local_intr_save(x) 宏，它会调用 __intr_save() ，保存当前中断状态，并禁用中断。 这样做可以确保这段代码在执行时不会被中断打断。

恢复中断： 在关键操作执行完毕后，调用 local_intr_restore(x) 宏，它会调用 __intr_restore(x) ，恢复之前保存的中断状态。通过这种方式，操作系统保证在禁用中断期间，不会影响系统的正常中断响应。

1. `local_intr_save(intr_flag)`：

- 该语句的作用是保存当前中断状态，并禁用中断。

实现禁用中断的过程（分析源代码）：

首先，if (read_csr(sstatus) & SSTATUS_SIE)

通过调用 `read_csr(sstatus)` 读取当前的 `sstatus` 寄存器，并检查当前 SIE 位是否被置位。

如果 `SIE` 位为 1，表示当前中断是启用的，那么将调用intr_disable()函数将中断禁用并返回 `1`。

如果 `SIE` 位为 0，表示当前中断已经禁用，函数直接返回 `0`，不进行任何操作。

2. `local_intr_restore(intr_flag)`:

- 该语句的作用是恢复之前保存的中断状态。

实现禁用中断的过程（分析源代码）：

首先， if (flag)

通过flag判定当前是否应该恢复中断

如果flag为1，那么将调用intr_enable()函数，然后结束本函数。

如果flag为0，直接结束本函数。