lab4 实验报告

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

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

进程是资源分配单位，线程是CPU调度单位。

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。

和前面一样，直接合并代码即可

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。

代码如下（按照提示，需要初始化的成员变量：state/pid/runs/kstack/need_resched/parent/mm/context/tf/cr3/flags/name）：

```
// 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) {
        proc->state =  (enum proc_state)PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
    }
```

```
    return proc;
}
```

请在实验报告中简要说明你的设计实现过程。

其实就是按照提示，初始化相应的成员变量

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）

  struct context 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。

  需要注意的是，与trapframe保存用户态内核态的上下文不同，context保存的是线程当前的上下文，可能是执行用户代码的上下文，也可能是执行内核代码的上下文。

  ```
  // Saved registers for kernel context switches.
  // Don't need to save all the %fs etc. segment registers,
  // because they are constant across kernel contexts.
  // Save all the regular registers so we don't need to care
  // which are caller save, but not the return register %eax.
  // (Not saving %eax just simplifies the switching code.)
  // The layout of context must match code in switch.S.
  struct context {
      uint32_t eip;
      uint32_t esp;
      uint32_t ebx;
      uint32_t ecx;
      uint32_t edx;
      uint32_t esi;
      uint32_t edi;
      uint32_t ebp;
  };
  ```

  struct trapframe *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。

- 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。

  进程之间通过进程调度来切换控制权，当某个`fork`出的新进程获取到了控制流后，首当其中执行的代码是`current->context->eip`所指向的代码，此时新进程仍处于内核态，但实际上我们想在用户态中执行代码，所以我们需要从内核态切换回用户态，也就是中断返回。此时会遇上两个问题：

- **新进程如何执行中断返回?** 这就是`proc->context.eip = (uintptr_t)forkret`的用处。`forkret`会使新进程正确的从中断处理例程中返回。
- **新进程中断返回至用户代码时的上下文为?** 这就是`proc_struct->tf`的用处。中断返回时，新进程会恢复保存的`trapframe`信息至各个寄存器中，然后开始执行用户代码。

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

do_fork函数代码如下：

```
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

    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

    copy_thread(proc, stack, tf);

    int intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
```

```
        list_add(&proc_list, &proc->list_link);
        nr_process++;
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);
    ret = proc->pid;
fork_out:
    return ret;

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

实现过程如下：

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

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。

  也就是get_pid函数是否能返回一个唯一的id。

  从代码可以看到，

  第一次被调用时，next_safe，last_pid刚开始赋值为8192，first time last_pid = 1，也就如果last_pid小于next_safe，那么是安全的。

  如果last_pid大于next_safe/MAX_PID，不一定是安全的，此时就需要遍历`proc_list`，重新对`last_pid`和`next_safe`进行设置，为下一次的`get_pid`调用打下基础。

  ```
  // get_pid - alloc a unique pid for process
  static int
  get_pid(void) {
      // MAX_PID = 8192
      static_assert(MAX_PID > MAX_PROCESS);
      struct proc_struct *proc;
      // proc 链表
      list_entry_t *list = &proc_list, *le;

      // next_safe = 8192, last_pid = 8192
      static int next_safe = MAX_PID, last_pid = MAX_PID;

      // first time last_pid = 1
      if (++ last_pid >= MAX_PID) {
          last_pid = 1;
  ```

```
            goto inside;
        }

        if (last_pid >= next_safe) {
        inside:
            next_safe = MAX_PID;
        repeat:
            le = list;
            while ((le = list_next(le)) != list) {
                proc = le2proc(le, list_link);
                // last_pid
                if (proc->pid == last_pid) {
                    if (++ last_pid >= next_safe) {
                        if (last_pid >= MAX_PID) {
                            last_pid = 1;
                        }
                        next_safe = MAX_PID;
                        goto repeat;
                    }
                }
                else if (proc->pid > last_pid && next_safe > proc->pid) {
                    next_safe = proc->pid;
                }
            }
        }
        return last_pid;
    }
```

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

请在实验报告中简要说明你对proc_run函数的分析。

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

`proc_run`代码如上所示，从init.c的kern_init->cpu_idle->schedule从proc_list找到需要执行的proc

主要步骤：

- current = proc，把当前proc设置为将要运行的proc

- load_esp0, 设置ts的ts_esp0（设置TSS中ring0的内核栈地址）为kstack + KSTACKSIZE
- lcr3加载cr3为将要运行的proc的cr3
- switch_to切换context，从当前运行的proc，切换为即将要运行的proc

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？

  两个，`idle_proc`，`init_proc`。

- 语句`local_intr_save(intr_flag);....local_intr_restore(intr_flag);`在这里有何作用？请说明理由

  保证两个语句中间的代码是原子操作不被打断，即disable_interrupt和enable_interrupt。