ucore实验四：内核线程管理

笔记本：	My Notebook		
创建时间：	2022/7/29 19:25	**更新时间：**	2022/8/5 22:04
作者：	赵凌珂		
URL：	https://chyyuu.gitbooks.io/ucore_os_docs/content/lab4/lab4_2_1_exercises.html		

作者：赵凌珂

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，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）

---

alloc_proc函数如下：

```
static struct proc_struct * alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        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;
}
```

整个分配初始化函数的运行过程为：
1. 在堆上分配一块内存空间用来存放进程控制块
2. 初始化进程控制块内的各个参数
3. 返回分配的进程控制块

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。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。

完成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;
    // 首先分配一个PCB
    if ((proc = alloc_proc()) == NULL)
        goto fork_out;
    // 设置子进程的父进程
    proc->parent = current;
    // 分配内核栈
    if (setup_kstack(proc) != 0)
```

```
        goto bad_fork_cleanup_proc;
    // 将所有虚拟页数据复制过去
    if (copy_mm(clone_flags, proc) != 0)
        goto bad_fork_cleanup_kstack;
    // 复制线程的状态
    copy_thread(proc, stack, tf);
    // 将子进程的PCB添加进hash list或者list
    bool 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);
    // 设置新的子进程可执行
    wakeup_proc(proc);
    // 返回子进程的pid
    ret = proc->pid;

fork_out:
    return ret;

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

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。

在函数get_pid中，如果静态成员last_pid小于next_safe，则当前分配的last_pid一定是安全的，即唯一的PID。
但如果last_pid大于等于next_safe，或者last_pid的值超过MAX_PID，则当前的last_pid就不一定是唯一的PID，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。
之所以在该函数中维护一个合法的PID的区间，是为了优化时间效率。如果简单的暴力搜索，则需要搜索大部分PID和所有的线程，这会使该算法的时间消耗很大，因此使用PID区间来优化算法。

get_pid代码如下：

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

```
            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

请在实验报告中简要说明你对proc_run函数的分析。并回答如下问题：

- 在本实验的执行过程中，创建且运行了几个内核线程？
- 语句local_intr_save(intr_flag);....local_intr_restore(intr_flag);在这里有何作用?请说明理由

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。

---

pron_run函数：

```
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);
        {
            // 设置当前执行的进程
            current = proc;
            // 设置ring0的内核栈地址
            load_esp0(next->kstack + KSTACKSIZE);
            // 加载页目录表
            lcr3(next->cr3);
            // 切换上下文
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

具体执行过程如下：

1.让 current 指向 next 内核线程 initproc；

2.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 ；
3.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；
4.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 函数执行完“ret”指令后，就切换到 initproc 执行了。
