操作系统 Lab4

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，并请回答所给问题。

代码实现

```
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;
}
```

- 首先，我们调用kmalloc函数为改进程分配一个proc_struct结构体所需要的内存大小。
- 如果分配成功，即proc的值不为空，我们则需要进行该进程初始化的变量及其值分别为：
    - state设置为PROC_UNINIT，即其状态为未进行初始化；
    - pid设置为-1，表示进程pid的还未初始化；
    - runs设置为0，表示该进程被运行的次数为0；
    - kstack设置为0，代表还没有给该进程分配内核栈（因此内核栈的位置未知）；
    - need_resched设置为0，代表该进程不需要调用schedule来释放自己所占用的CPU资源；
    - parent设置为NULL，代表该进程没有父进程。
    - mm设置为NULL，代表还未给该进程准备好内存管理；
    - context在这里被使用memset进行了数据清零操作，代表为该进程准备了新的空白上下文；
    - tf设置为NULL，代表还未给该进程分配中断帧；
    - cr3设置为boot_cr3，代表该进程使用的是内核页目录表。
    - flags设置为0，代表该进程还未被设置任何标志；

    - name在这里被使用memset进行数据清零操作，方便后续设置新的进程名。
- 最后我们把该进程的指针返回。

问题回答

> 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?

- context用于保存进程上下文使用结构体struct context保存，其中包含了ra，sp，s0~s11共14个寄存器。我们不需要保存所有的寄存，因为我们巧妙地利用了编译器对于函数的处理。寄存器可以分为调用者保存（caller-saved）寄存器和被调用者保存（callee-saved）寄存器。因为线程切换在一个函数当中，所以编译器会自动帮助我们生成保存和恢复调用者保存寄存器的代码，在实际的进程切换过程中我们只需要保存被调用者保存寄存器。
- 比如在本实验中，当idle进程被CPU切换为init进程时，将idle进程的上下文就会保存在 `idleproc->context` 中，如果uCore调度器选择了idleproc执行，就要根据 `idleproc->context` 恢复现场，继续执行，具体的切换过程体现在switch.S中的switch_to函数中。
- tf保存了进程的中断帧。当进程从用户空间跳进内核空间的时候，进程的执行状态被保存在了中断帧中（注意这里需要保存的执行状态数量不同于上下文切换）。系统调用可能会改变用户寄存器的值，我们可以通过调整中断帧来使得系统调用返回特定的值。
- 具体在本次实验中，在我们进行进程切换的时候，switch_to函数运行的最后会返回到forket函数中，而这个时候我们会把此次中断的中断帧置于a0寄存器中，这样在执行__trapret函数的时候，就会利用这个中断帧返回到指定的位置、以特定的参数执行某个函数。

练习2：为新创建的内核线程分配资源（需要编程）

> 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，并回答给出的问题。

实现过程

```
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;
    //--------CODE STRAT--------
     proc = alloc_proc(); // 给当前进程分配一个进程结构体
     if (proc == NULL) {
        goto fork_out; // 分配失败，返回
    }
    proc -> parent = current; // current进程作为新进程的父进程
    if(setup_kstack(proc)!=0)// 为进程分配一个内核栈
    {
        goto bad_fork_cleanup_proc; // 分配失败，返回
    }
    if(copy_mm(clone_flags, proc)!=0)// 将父进程的内存管理信息复制到子进程
    {
        goto bad_fork_cleanup_kstack; // 拷贝失败，返回
    }
    copy_thread(proc, stack, tf); // 复制原进程上下文到新进程
    bool intr_flag;
    local_intr_save(intr_flag); // 屏蔽中断，intr_flag设为1
    {
        proc->pid = get_pid();// 获取当前进程的pid
        hash_proc(proc);// 建立映射
        nr_process++;// 进程数+1
        list_add(&proc_list,&(proc->list_link));// 将进程加入进程链表中
    }
    local_intr_restore(proc);// 恢复中断
    wakeup_proc(proc); // 唤醒新进程
    ret = proc -> pid; // 返回新进程号
    //--------CODE END--------
fork_out:
    return ret;
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

问题回答

> 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

- 对于这个问题，分配ID的代码是get_pid()，其代码如下

```
 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;
}
```

- `[last_pid, next_safe)` 指定了一段连续的未分配的pid区间。关于这段区间的维护流程是：
  - 如果当前遍历到的进程号等于last_pid，则说明区间的左端点需要更新，先给last_pid加一，然后检查是否有 `last_pid < next_safe` 成立，如果成立时，则继续遍历，如果不成立；则代表右边界需要重新更新为MAX_PID，这时如果`last_pid` 超过了MAX_PID，需要重新设置为1。
  - 如果当前遍历到的进程号不等于last_pid，并且处于所维护的区间之内，则需要更新右边界。扫描结束时，last_pid为当前进程列表中没有被占用过的pid，可以作新的pid，因此可以做到给每个新fork的线程一个唯一的id。

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。
>
> 请完成代码的编写，并回答给出的问题。

实现过程

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) { // 如果我们需要切换的进程不等于当前的进程
       bool intr_flag;
       struct proc_struct *prev = current, *next = proc;
       // prev代表前一个进程 next代表后一个进程
       local_intr_save(intr_flag); // 禁用中断
       {
         current = proc; // 更新current指针
         lcr3(next->cr3); // 更新cr3寄存器的值，也就是页目录表
         switch_to(&(prev->context), &(next->context)); // 进行上下文切换
       }
       local_intr_restore(intr_flag); // 恢复中断
    }
}
```

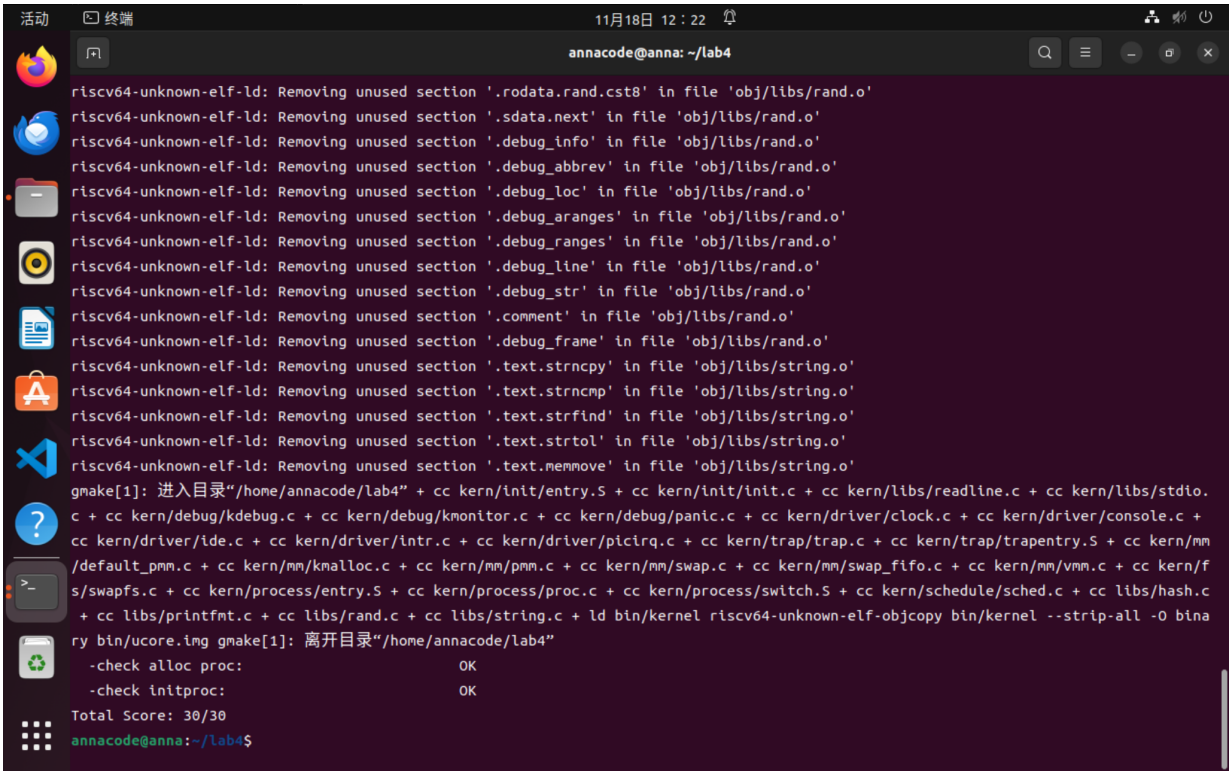
问题回答

> 在本实验的执行过程中，创建且运行了几个内核线程?

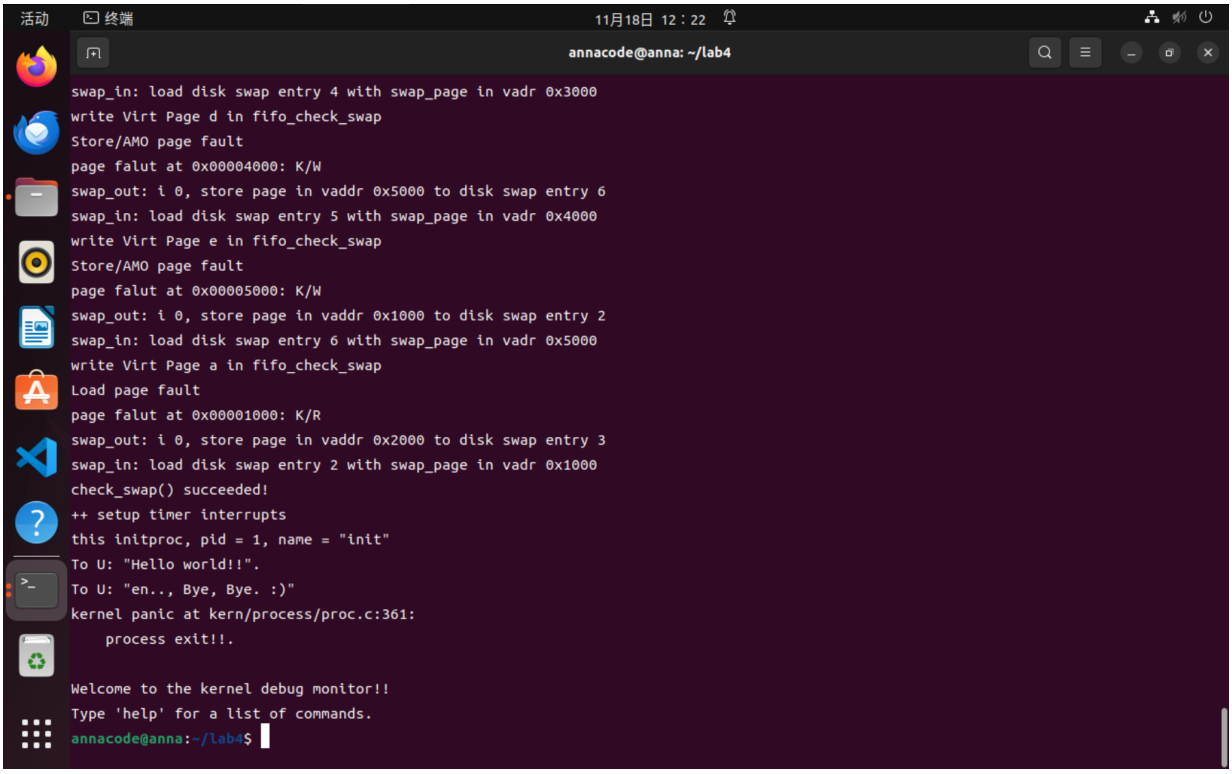
有两个内核线程： **1） 创建第0个内核线程idleproc。**在 init.c::kern_init 函数调用了 proc.c::proc_init 函数。proc_init 函数启动了创建内核线程的步骤。首先当前的执行上下文（从 kern_init 启动至今）就可以看成是 uCore 内核（也可看做是内核进程）中的一个内核线程的上下文。为此，uCore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第 0 个内核线程 – idleproc。 **2） 创建第 1 个内核线程 initproc。**第 0 个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行 cpu_idle 函数开始过退休生活了。所以 uCore 接下来还需创建其他进程来完成各种工作，但 idleproc 内核子线程自己不想做，于是就通过调用 kernel_thread 函数创建了一个内核线程 init_main。在Lab4中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看 init_main 函数）。但在后续的实验中，init_main 的工作就是创建特定的其他内核线程或用户进程。

运行截图

- make grade



- make qemu



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?

- 在Linux内核中，`local_intr_save(intr_flag)`和`local_intr_restore(intr_flag)`是用来实现开关中断的语句。
- `local_intr_save(intr_flag)`是一个宏，用于保存当前中断状态并禁用中断。它的实现通常会做以下两个步骤：

    1. `intr_flag`是一个本地变量，用于保存当前中断状态。在保存中断状态之前，它通常会先使用相关的寄存器或者操作，将当前的中断状态保存到`intr_flag`中。
    2. 然后，它会使用特定的指令（如`cli`指令）来禁用中断。这个指令会将中断屏蔽位设置为屏蔽中断的状态，从而实现禁用中断的效果。
- `local_intr_restore(intr_flag)`也是一个宏，用于根据之前保存的中断状态重新设置中断。它的实现通常包括以下两个步骤：
    1. 首先，它会使用特定的指令（如`sti`指令）来恢复中断。这个指令会将中断屏蔽位设置为允许中断的状态，从而实现中断的恢复。
    2. 然后，它会使用之前保存的中断状态（即`intr_flag`）将中断状态还原回之前的状态。这可能涉及使用相关的寄存器或操作，将保存的中断状态重新加载，使其生效。
- 总体而言，通过使用这两个宏，可以在必要时禁用中断，并在适当时机恢复中断，以实现临时的中断屏蔽和恢复的功能。