lab4 report

练习0：填写已有实验

使用`meld`填写已有实验。

练习1：分配并初始化一个进程控制块(需要编码)

数据结构

proc_struct

```
struct proc_struct {
    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
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
};
```

实现思路

在`alloc_proc`函数中完成PCB的基本初始化。PCB的结构和成员变量如上。成员可以分为以下几种：

- 进程标识: pid, name, parent
- 状态: state, runs, flags, need_resched
- 资源: kstack, mm, cr3
- 保护现场: context, tf
- 队列信息: list_link, hash_link

具体步骤

把proc进行初步初始化(即把proc_struct中的各个成员变量清零)。但有些成员变量设置了特殊的值。

```c
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
    //LAB4:EXERCISE1 2014011336
        proc->state = PROC_UNINIT;  //设置进程为“初始”态
        proc->pid = -1;              //设置进程pid的未初始化值
        proc->cr3 = boot_cr3;        //使用内核页目录表的基址
        proc->run = 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->flags = 0;
        menset(proc->name,0,PROC_NAME_LEN);
    }
    return proc;
}
```

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。实际上用于**保存上一个进程的寄存器**。
  - 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。
- **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。
  - 进程切换时，产生中断，此时tf保存现场。切换完成后，根据下一个进程的tf恢复中断。

练习2：为新创建的内核线程分配资源(需要编码)

相关宏定义

```
    /*
     * 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
     */
```

实现思路

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。

具体步骤

1. 分配并初始化进程控制块(alloc_proc函数);
2. 分配并初始化内核栈(setup_stack函数);
3. 根据clone_flag标志复制或共享进程内存管理结构(copy_mm函数);
4. 设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文 (copy_thread函数);
5. 把设置好的进程控制块放入hash_list和proc_list两个全局进程链表中;
6. 自此，进程已经准备好执行了，把进程状态设置为“就绪”态;
7. 设置返回码为子进程的id号。

这里需要注意：

- 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。
- 还需要设置proc->parent = current
- 为保证与其他进程同步互斥(get_pid(), 插入进程队列, nr_process ++)，需要加锁，这里禁止了中断。由于目前ucore只实现了对单处理器的支持，所以通过这种方式，就可简单地支撑互斥操作了。

```
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 2014011336
    //    1. call alloc_proc to allocate a proc_struct
    proc = alloc_proc();
    if(proc==NULL){
        goto fork_out;
    }

    proc->parent = current;

    //    2. call setup_kstack to allocate a kernel stack for child process
    if(setup_kstack(proc)!=0){
        goto bad_fork_cleanup_proc;
    }

    //    3. call copy_mm to dup OR share mm according clone_flag
    if(copy_mm(clone_flags,proc)!=0){
        goto bad_fork_cleanup_kstack;
    }

    //    4. call copy_thread to setup tf & context in proc_struct
    copy_thread(proc,stack,tf);

    //    5. insert proc_struct into hash_list && proc_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);

    //    6. call wakeup_proc to make the new child process RUNNABLE
    wakeup_proc(proc);

    //    7. set ret vaule using child proc's pid
    ret = proc->pid;

fork_out:
    return ret;
```

```
50
51  bad_fork_cleanup_kstack:
52      put_kstack(proc);
53  bad_fork_cleanup_proc:
54      kfree(proc);
55      goto fork_out;
56  }
```

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。

做到了。要保证id的唯一性，需要在修改id时保证进程**同步互斥**。由于目前ucore只实现了对单处理器的支持，所以通过**禁止中断**的方式，就可简单地支撑互斥操作了。

根据实验指导书`8.3.3`，在单处理器情况下，可以通过开关中断实现对临界区的互斥保护，需要互斥的临界区代码的一般写法为：

```
1  local_intr_save(intr_flag);
2  {
3    临界区代码
4  }
5  local_intr_restore(intr_flag);
6  ……
7
8  关中断：local_intr_save --> __intr_save --> intr_disable --> cli
9  开中断：local_intr_restore--> __intr_restore --> intr_enable --> sti
```

在多处理器情况下，这种方法是无法实现互斥的，因为屏蔽了一个CPU的中断，只能阻止本CPU上的进程不会被中断或调度，并不意味着其他CPU上执行的进程不能执行临界区的代码。所以，开关中断只对单处理器下的互斥操作起作用。

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)

proc_run

```
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);
            //(1)
            switch_to(&(prev->context), &(next->context));
            //(2)
        }
        local_intr_restore(intr_flag);
    }
}
```

若需要切换进程prev->next, (prev≠next):

1. 禁止中断，防止切换过程被打断，防止再次被调度。
2. 设置当前PCB为next的PCB
3. 设置任务状态段ts中特权态0下的栈顶指针esp0为next的内核栈的栈顶，即next->kstack + KSTACKSIZE
4. 设置CR3寄存器的值为next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
5. 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next执行了。

说明：

- 对于一般的进程切换，会从上一个进程的(1)处切换到下一个进程的(2)处。
- 对于fork()刚创建的进程，由于设置了`context.eip = (uintptr_t)forkret`，切换后会执行forkret，forkret 调用 forkrets，forkrets 假装是从中断中返回。根据 tf 的内容从中断返回后，就设置好了 fork 好的子进程的上下文，并开始执行了。

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：

1. idleproc：第0号内核线程，代表ucore os的管理工作，完成内核中各个子系统的初始化，之后立即调度，执行其他进程。
2. initproc：用于完成实验的功能而调度的内核线程。

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

禁止中断，防止切换过程被打断，防止再次被调度。详见练习2最后。

与参考答案的区别

练习1

基本相同。参考答案使用`memset`为context成员初始化，然而似乎不用这样特地分配空间。

练习2

刚开始忽略了parent的初始化，而且没有在get_pid()等要求互斥的地方禁止中断。查看参考答案和查找实验指导书后理解了禁止中断的写法和含义。

知识点

列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

- 进程创建。对PCB的初始化，特别是tf和context。
- 进程切换。分为刚fork好的进程和其他类型。刚fork好的进程伪造了中断现场，假装从中断返回，执行第一条指令。

列出你认为OS原理中很重要，但在实验中没有对应上的知识点

- 整个进程切换的流程在实验中没有得到很好的展示。如何从中断开始，到恢复中断结束。经过自己学习后掌握。
- 进程创建时tf和context的初始化在实验中只是略微提了一句，没有作为实验内容。
- 建议可以说明一下`context.eip/esp`和`tf.tf_eip/esp`有什么不同。