实验五：用户进程管理

计31 张正 2013011418

一、实验目的

- 了解第一个用户进程创建过程
- 了解系统调用框架的实现机制
- 了解ucore如何实现系统调用sys_fork/sys_exec/sys_exit/sys_wait来进行进程管理

二、实验内容

实验4完成了内核线程，但到目前为止，所有的运行都在内核态执行。实验5将创建用户进程，让用户进程在用户态执行，且在需要ucore支持时，可通过系统调用来让ucore提供服务。为此需要构造出第一个用户进程，并通过系统调用sys_fork/sys_exec/sys_exit/sys_wait来支持运行不同的应用程序，完成对用户进程的执行过程的基本管理。

练习0：填写已有实验

本实验依赖实验1/2/3/4。请把你做的实验1/2/3/4的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行lab5的测试应用程序，可能需对已完成的实验1/2/3/4的代码进行进一步改进。

将之前的代码拷贝进去之后，根据更新的提示，更新了如下的代码

更新了lab1试验中填入的idt_init函数

```
void
idt_init(void) {
     /* LAB1 2013011418 : STEP 2 */
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);    //设置IDT
    }
SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);//设置相应中断门即可
    lidt(&idt_pd); //载入IDT表
}
```

更新了lab1中的时钟中断部分

```
 ticks ++;    //一次中断累加1
 if (ticks % TICK_NUM == 0) {
            assert(current != NULL);
            current->need_resched = 1;//时间片用完设置为需要调度
```

```
}
```
更新了lab4中练习一

```
        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));//初始化进程上下文
        proc->tf = NULL; //初始化中断帧，用于记录进程发生中断前的状态
        proc->cr3 = boot_cr3;         //因为是内核线程，所以CR3=boot_cr3
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0; //初始化进程等待状态
        proc->cptr = proc->optr = proc->yptr = NULL;//进程相关指针初始化
```

更新了lab4中的练习二

```
    if ((proc = alloc_proc()) == NULL) {//获得用户信息块
        goto fork_out;
    }
    proc->parent = current;//设置父进程
    assert(current->wait_state == 0); //确保当前进程为等待进程
    if (setup_kstack(proc) != 0) {//分配了2页内核栈
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);//设置中断帧
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);//将新进程加入hash_list
         set_links(proc);//设置进程的相关链接
    }
    local_intr_restore(intr_flag);//将新进程加入proc_list
```

```
wakeup_proc(proc);//唤醒进程，等待调度
ret = proc->pid;
```

练习1: 加载应用程序并执行

do_execv函数调用load_icode（位于kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好proc_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的trapframe内容。

在相应的lab5练习一的地方加入如下代码

```
tf->tf_cs = USER_CS;
tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;
tf->tf_eip = elf->e_entry;
tf->tf_eflags = FL_IF;//FL_IF为中断打开状态
```

- 请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被ucore选择占用CPU执行（RUNNING态）到具体执行应用程序第一条指令的整个经过

①调用mm_create函数来申请进程的内存管理数据结构mm所需内存空间，并对mm进行初始化；

②调用setup_pgdir来申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表（boot_pgdir所指）的内容拷贝到此新目录表中，最后让mm->pgdir指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间；

③根据应用程序执行码的起始位置来解析此ELF格式的执行程序，并调用mm_map函数根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把vma插入到mm结构中，从而表明了用户进程的合法用户态虚拟地址空间；

④调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；

⑤需要给用户进程设置用户栈，为此调用mm_mmap函数建立用户栈的vma结构，明确用户栈的位置在用户虚空间的顶端，大小为256个页，即1MB，并分配一定数量的物理内存且建立好栈的虚地址<-->物理地址映射关系；

⑥至此,进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被程序

的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；

⑦先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让CPU转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断；

至此，用户进程的用户环境已经搭建完毕。此时initproc将按产生系统调用的函数调用路径原路返回，执行中断返回指令“iret”后，将切换到用户进程程序的第一条语句位置_start处开始执行。

练习2: 父进程复制自己的内存空间给子进程

创建子进程的函数do_fork在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过copy_range函数（位于kern/mm/pmm.c中）实现的，请补充copy_range的实现，确保能够正确执行。

补充代码如下

```
void * kva_src = page2kva(page);//返回父进程的内核虚拟页地址
void * kva_dst = page2kva(npage);//返回子进程的内核虚拟页地址
memcpy(kva_dst, kva_src, PGSIZE);//复制父进程到子进程
ret = page_insert(to, npage, start, perm);//建立子进程页地址起始位置与物理地址的映射关系(prem是权限)
```

练习3: 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

请在实验报告中简要说明你对 fork/exec/wait/exit函数的分析。

①.fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。

②exit函数会把一个退出码error_code传递给ucore，ucore通过执行内核函数do_exit来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作，具体流程如下：

1. 如果current->mm != NULL，表示是用户进程，则开始回收此用户进程所占用的用户态虚拟内存空间；

a) 首先执行“lcr3(boot_cr3)”，切换到内核态的页表上，这样当前用户进程目前只能在内核虚拟地址空间执行了，这是为了确保后续释放用户态内存和进程页表的工作能够正常执行；

b) 如果当前进程控制块的成员变量mm的成员变量mm_count减1后为0，则开始回收用户进程所占的内存资源：

i. 调用exit_mmap函数释放current->mm->vma链表中每个vma描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后还把页表所占用的空间释放并把对应的页目录表项清空；

ii. 调用put_pgdir函数释放当前进程的页目录所占的内存；

iii. 调用mm_destroy函数释放mm中的vma所占内存，最后释放mm所占内存；

c) 此时设置current->mm为NULL，表示与当前进程相关的用户虚拟内存空间和对应的内存管理成员变量所占的内核虚拟内存空间已经回收完毕；

2. 这时，设置当前进程的执行状态current->state=PROC_ZOMBIE，当前进程的退出码current->exit_code=error_code。此时当前进程已经不能被调度了，需要此进程的父进程来做最后的回收工作（即回收描述此进程的内核栈和进程控制块）；

3. 如果当前进程的父进程current->parent处于等待子进程状态：

current->parent->wait_state==WT_CHILD，

则唤醒父进程，让父进程帮助自己完成最后的资源回收；

4. 如果当前进程还有子进程，则需要把这些子进程的父进程指针设置为内核线程initproc，且各个子进程指针需要插入到initproc的子进程链表中。如果某个子进程的执行状态是PROC_ZOMBIE，则需要唤醒initproc来完成对此子进程的最后回收工作。

5. 执行schedule()函数，选择新的进程执行。

③do_execve函数来完成用户进程的创建工作。此函数的主要工作流程如下：

（一）首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。由于此处的initproc是内核线程，所以mm为NULL，整个处理都不会做。

（二）接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。这里涉及到读ELF格式的文件，申请内存空间，建立用户态虚存空间，加载应用程序执行码等。load_icode函数完成了整个复杂的工作。

④wait函数等待任意子进程的结束通知，wait_pid函数等待进程id号为pid的子进程结束通知。这两个函数最终访问sys_wait系统调用接口让ucore来完成对子进程的最后回收工作，即回收子进程的内核栈和进程控制块所占内存空间，具体流程如下：

1. 如果pid!=0，表示只找一个进程id号为pid的退出状态的子进程，否则找任意一个处于退出状态的子进程；

2. 如果此子进程的执行状态不为PROC_ZOMBIE，表明此子进程还没有退出，则当前进程只好设置自己的执行状态为PROC_SLEEPING，睡眠原因为WT_CHILD，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤1处执行；

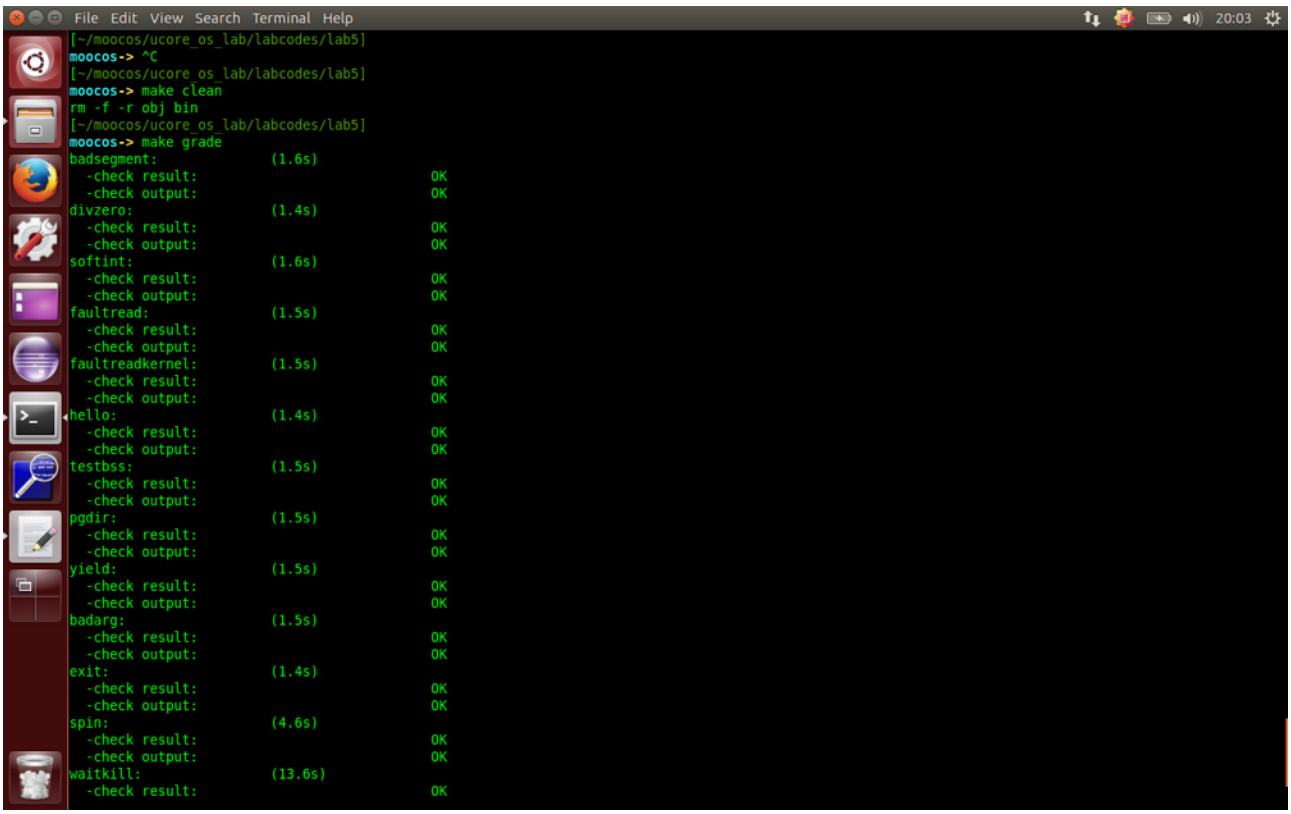
3. 如果此子进程的执行状态为PROC_ZOMBIE，表明此子进程处于退出状态，需要当前进程（即子进程的父进程）完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc_list和hash_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，消除了它所占用的所有资源。

• 请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的?

以上分析已给出。

• 请给出ucore中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）

最终运行结果为


File Edit View Search Terminal Help
20:03
[~/moocos/ucore_os_lab/labcodes/lab5]
moocos-> ^C
[~/moocos/ucore_os_lab/labcodes/lab5]
moocos-> make clean
rm -f -r obj bin
[~/moocos/ucore_os_lab/labcodes/lab5]
moocos-> make grade
badsegment: (1.6s)
-check result: OK
-check output: OK
divzero: (1.4s)
-check result: OK
-check output: OK
softint: (1.6s)
-check result: OK
-check output: OK
faultread: (1.5s)
-check result: OK
-check output: OK
faultreadkernel: (1.5s)
-check result: OK
-check output: OK
hello: (1.4s)
-check result: OK
-check output: OK
testbss: (1.5s)
-check result: OK
-check output: OK
pgdir: (1.5s)
-check result: OK
-check output: OK
yield: (1.5s)
-check result: OK
-check output: OK
badarg: (1.5s)
-check result: OK
-check output: OK
exit: (1.4s)
-check result: OK
-check output: OK
spin: (4.6s)
-check result: OK
-check output: OK
waitkill: (13.6s)
-check result: OK