uCore实验五报告

黄郭斌 计科1502班 201507010206

一. 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。相关原理介绍可看附录B。

二. 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

三. 实验分析及结果

0. 练习零

> 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。

这里简单将我们需要修改的地方罗列如下：

- proc.c
- default_pmm.c
- pmm.c
- swap_fifo.c
- vmm.c
- trap.c

另外根据试验要求，我们需要对部分代码进行改进，这里讲需要改进的地方的代码和说明罗列如下：

alloc_proc函数

改进后的`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);
        proc->wait_state = 0;
        proc->cptr = proc->optr = proc->yptr = NULL;
    }
    return proc;
}12345678910111213141516171819201
```

比起改进之前多了这两行代码：

```
        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  12
```

这里解释proc的几个新指针：

```
parent:           proc->parent  (proc is children)
children:         proc->cptr    (proc is parent)
older sibling:    proc->optr    (proc is younger sibling)
younger sibling:  proc->yptr    (proc is older sibling)1234
```

就像注释所写的，这两行代码主要是初始化进程等待状态、和进程的相关指针，例如父进程、子进程、同胞等等。

因为这里涉及到了用户进程，自然需要涉及到调度的问题，所以进程等待状态和各种指针需要被初始化。

do_fork函数

改进后的`do_fork`函数如下：

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

    proc->parent = current;
    assert(current->wait_state == 0);//确保当前进程正在等待

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

```
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        set_links(proc);//将原来简单的计数改成来执行set_links函数，从而实现设置进程的相关链接

    }
    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;
}123456789101112131415161718192021222324252627282930313233343536373839404142434445
```

改动主要是上述代码中含注释的两行，第一行是为了确定当前的进程正在等待，第二行是将原来的计数换成了执行一个`set_links`函数，因为要涉及到进程的调度，所以简单的计数肯定是不行的。

idt_init函数

改进后的`idt_init`函数如下：

```
void idt_init(void) {
    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);
    }
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
    lidt(&idt_pd);
}123456789
```

相比于之前，多了这一行代码：

```
SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);////这里主要是设置相应的中断门1
```

trap_dispatch函数

改进后的部分函数如下：

```
ticks ++;
       if (ticks % TICK_NUM == 0) {
           assert(current != NULL);
           current->need_resched = 1;
       }
       break;123456
```

相比与原来主要是多了这一行代码

```
current->need_resched = 1;1
```

这里主要是将时间片设置为需要调度，说明当前进程的时间片已经用完了。

1. 练习一(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。 请在实验报告中简要说明你的设计实现过程。 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。

根据实验说明书，我们需要完善的函数是`load_icode`函数。

这里介绍下这个函数的功能：`load_icode`函数主要用来被`do_execve`调用,将执行程序加载到进程空间（执行程序本身已从磁盘读取到内存中）,这涉及到修改页表、分配用户栈等工作。 该函数主要完成的工作如下：

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. 根据可执行程序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格式执行程序的各个段(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，表明这些是用户进程的合法用户态虚拟地址空间;
4. 根据可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 init 已经被 exit 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;

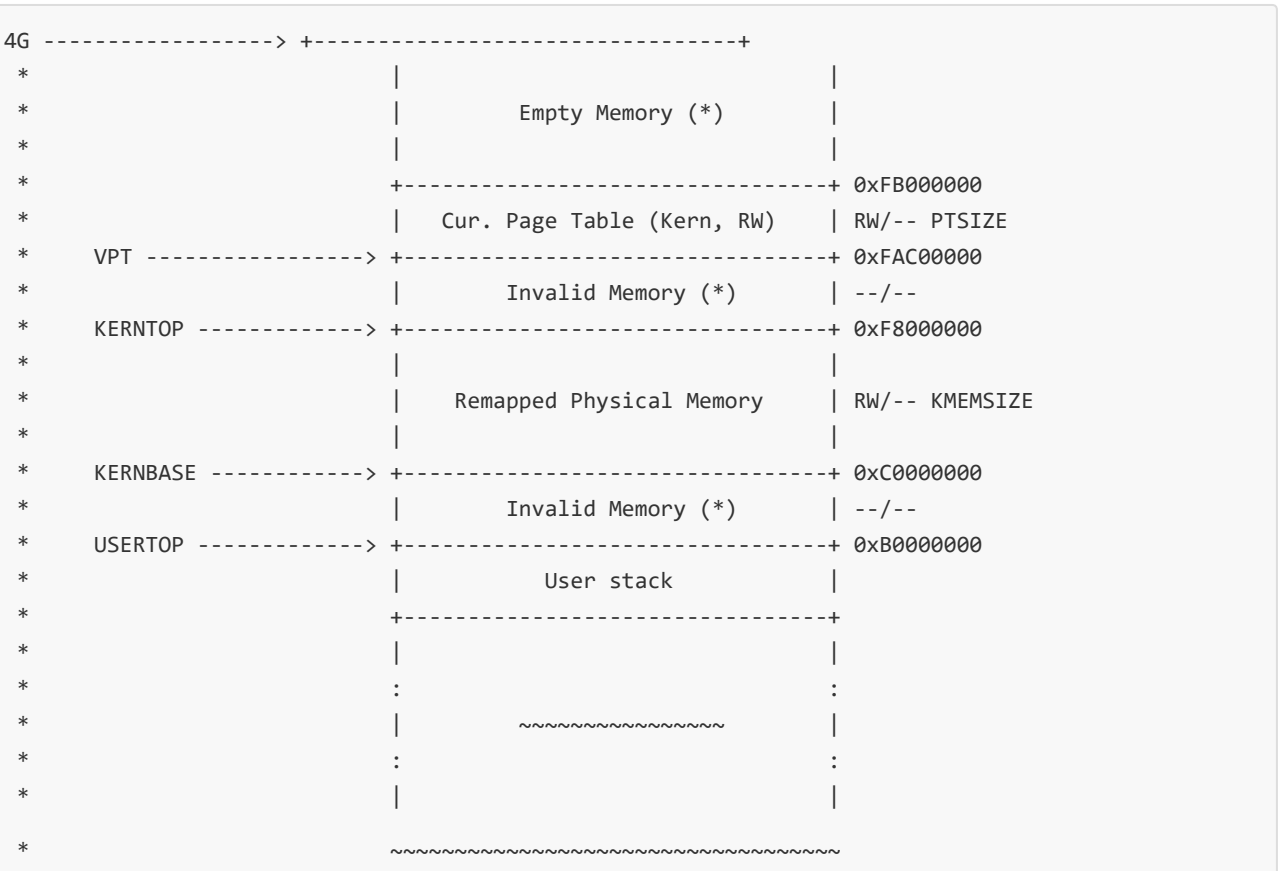
简单的说，该`load_icode` 函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。

而这里这个`do_execve`函数主要做的工作就是先回收自身所占用户空间，然后调用`load_icode`，用新的程序覆盖内存空间，形成一个执行新程序的新进程。

由于该完整函数太长，所以这里我只将我们补充的部分罗如下：

```
static int load_icode(unsigned char *binary, size_t size) {
    ......
    ......
    /* LAB5:EXERCISE1 YOUR CODE
     * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
     * NOTICE: If we set trapframe correctly, then the user level process can return to USER
MODE from kernel. So
     *          tf_cs should be USER_CS segment (see memlayout.h)
     *          tf_ds=tf_es=tf_ss should be USER_DS segment
     *          tf_esp should be the top addr of user stack (USTACKTOP)
     *          tf_eip should be the entry point of this binary program (elf->e_entry)
     *          tf_eflags should be set to enable computer to produce Interrupt
     */
    tf->tf_cs = USER_CS;
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = USTACKTOP;//0xB0000000
    tf->tf_eip = elf->e_entry;//
    tf->tf_eflags = FL_IF;
    ......
    ......
}123456789101112131415161718192O
```

根据注释这里我们主要完成的是`proc_struct`结构中tf结构体变量的设置，因为这里我们要设置好tf以便于从内核态切换到用户态然后执行程序，所以这里`tf_cs`即代码段设置为`USER_CS`、将`tf->tf_ds`、`tf->tf_es`、`tf->tf_ss`均设置为USER_DS。 至于之后的`tf_esp`和`tf_eip`的设置需要看这个图，这是一个完整的虚拟内存空间的分布图：



```
 *                            |       User Program & Heap       |
 *     UTEXT ---------------> +---------------------------------+ 0x00800000
 *                            |        Invalid Memory (*)       | --/--
 *                            |  - - - - - - - - - - - - - - -  |
 *                            |    User STAB Data (optional)    |
 *     USERBASE, USTAB------> +---------------------------------+ 0x00200000
 *                            |        Invalid Memory (*)       | --/--
 *     0 -------------------> +---------------------------------+
0x000000001234567891011121314151617181920212223242526272829303l
```

这样子就知道为啥要这样赋值了。 至于最后的`tf->tf_eflags = FL_IF`主要是打开中断。

2. 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。 请在实验报告中简要说明如何设计实现”Copy on Write 机制“， 给出概要设计， 鼓励给出详细设计。
>
> > Copy-on-write（简称COW）的基本概念是指如果有多个使用者对一个资源A（比如内存块）进行读操作， 则每个使用者只需获得一个指向同一个资源A的指针， 就可以该资源了。 若某使用者需要对这个资源A进行写操作， 系统会对该资源进行拷贝操作， 从而使得该”写操作”使用者获得一个该资源A的”私有”拷贝—资源B， 可对资源B进行写操作。 该”写操作”使用者对资源B的改变对于其他的使用者而言是不可见的， 因为其他使用者看到的还是资源A。

如题，这个工作的完整由`do_fork`函数完成，具体是调用`copy_range`函数，而这里我们的任务就是补全这个函数。 这个具体的调用过程是由`do_fork`函数调用`copy_mm`函数，然后`copy_mm`函数调用`dup_mmap`函数，最后由这个`dup_mmap`函数调用`copy_range`函数。

即

```
do_fork()---->copy_mm()---->dup_mmap()---->copy_range()1
```

这里我们需要填写以下部分:

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end, bool share) {
    ......
    ......
    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)
    ......
    ......
}12345678910
```

这里就是调用一个`memcpy`将父进程的内存直接复制给子进程即可。

3. 练习三(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)

- 我们逐个进行分析

- fork

首先当程序执行fork时，fork使用了系统调用`SYS_fork`,而系统调用`SYS_fork`则主要是由`do_fork`和`wakeup_proc`来完成的。`do_fork()`完成的工作在lab4的时候已经做过详细介绍，这里再简单说一下，主要是完成了以下工作：

- 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。

而`wakeup_proc`函数主要是将进程的状态设置为等待，即`proc->wait_state = 0`，此处不赘述。

- exec

当应用程序执行的时候，会调用`SYS_exec`系统调用,而当ucore收到此系统调用的时候，则会使用`do_execve()`函数来实现，因此这里我们主要介绍`do_execve()`函数的功能，函数主要时完成**用户进程**的创建工作，同时使用户进程进入执行。 主要工作如下：

- 1、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。
- 2、接下来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。之后就是调用`load_icode`从而使之准备好执行。（具体`load_icode`的功能在练习1已经介绍的很详细了，这里不赘述了）

- wait

当执行`wait`功能的时候，会调用系统调用`SYS_wait`，而该系统调用的功能则主要由`do_wait`函数实现，主要工作就是父进程如何完成对子进程的最后回收工作，具体的功能实现如下：

- 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，则当前进程设置执行状态为`PROC_SLEEPING`（睡眠），睡眠原因为`WT_CHILD`(即等待子进程退出)，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，它所占用的所有资源均已释放。

- exit

当执行`exit`功能的时候，会调用系统调用`SYS_exit`，而该系统调用的功能主要是由`do_exit`函数实现。具体过程如下：

- 1、先判断是否是用户进程，如果是，则开始回收此用户进程所占用的用户态虚拟内存空间;（具体的回收过程不作详细说明）

- 2、设置当前进程的中hi性状态为`PROC_ZOMBIE`，然后设置当前进程的退出码为`error_code`。表明此时这个进程已经无法再被调度了，只能等待父进程来完成最后的回收工作（主要是回收该子进程的内核栈、进程控制块）
- 3、如果当前父进程已经处于等待子进程的状态，即父进程的`wait_state`被置为`WT_CHILD`，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。
- 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`init`,且各个子进程指针需要插入到`init`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`init`来完成对此子进程的最后回收工作。
- 5、执行schedule()调度函数，选择新的进程执行。

所以说该函数的功能简单的说就是，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。

关于系统调用

首先罗列下目前ucore所有的系统调用如下表：

```
SYS_exit        : process exit,                           -->do_exit
SYS_fork        : create child process, dup mm            -->do_fork-->wakeup_proc
SYS_wait        : wait process                            -->do_wait
SYS_exec        : after fork, process execute a program   -->load a program and refresh the
mm
SYS_clone       : create child thread                     -->do_fork-->wakeup_proc
SYS_yield       : process flag itself need resecheduling, -->proc->need_sched=1, then
scheduler will rescheule this process
SYS_sleep       : process sleep                           -->do_sleep
SYS_kill        : kill process                            -->do_kill-->proc->flags |=
PF_EXITING
                                                                  -->wakeup_proc-->do_wait--
>do_exit
SYS_getpid      : get the process's pid12345678910
```

一般来说，用户进程只能执行一般的指令,无法执行特权指令。采用系统调用机制为用户进程提供一个获得操作系统服务的统一接口层，简化用户进程的实现。 根据之前的分析，应用程序调用的 exit/fork/wait/getpid 等库函数最终都会调用 syscall 函数,只是调用的参数不同而已（分别是 SYS_exit / SYS_fork / SYS_wait / SYS_getid ）

当应用程序调用系统函数时，一般执行`INT T_SYSCALL`指令后，CPU 根据操作系统建立的系统调用中断描述符，转入内核态，然后开始了操作系统系统调用的执行过程，在内核函数执行之前，会保留软件执行系统调用前的执行现场，然后保存当前进程的`tf`结构体中，之后操作系统就可以开始完成具体的系统调用服务，完成服务后，调用`IRET`返回用户态，并恢复现场。这样整个系统调用就执行完毕了。

四. 实验体会

通过本次实验,很好的理解了ucore是如何实现系统调用sys_fork/sys_exec/sys_exit/sys_wait来进行进程管理;了解了如何创建用户进程,fork的作用,父进程为了等待对子进程的回收而进行等待,最后还有如何处理进程的退出。在这个实验中,还对于用户态的执行状态生命周期有了更深刻的理解,特别是对状态生命周期图有了更深刻的印象。