OS_Lab5_Experimental report

湖南大学信息科学与工程学院

计科 210X 甘晴void （学号 202108010XXX）

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

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**：预处理（继承与新增）

继承

本实验依赖实验1/2/3/4。请把你做的实验1/2/3的代码填入本实验中代码中有“LAB1”,“LAB2”,“LAB3”,“LAB4”的注释相应部分。

使用meld将实验1/2/3的代码中相应的部分填入实验四中的代码中：

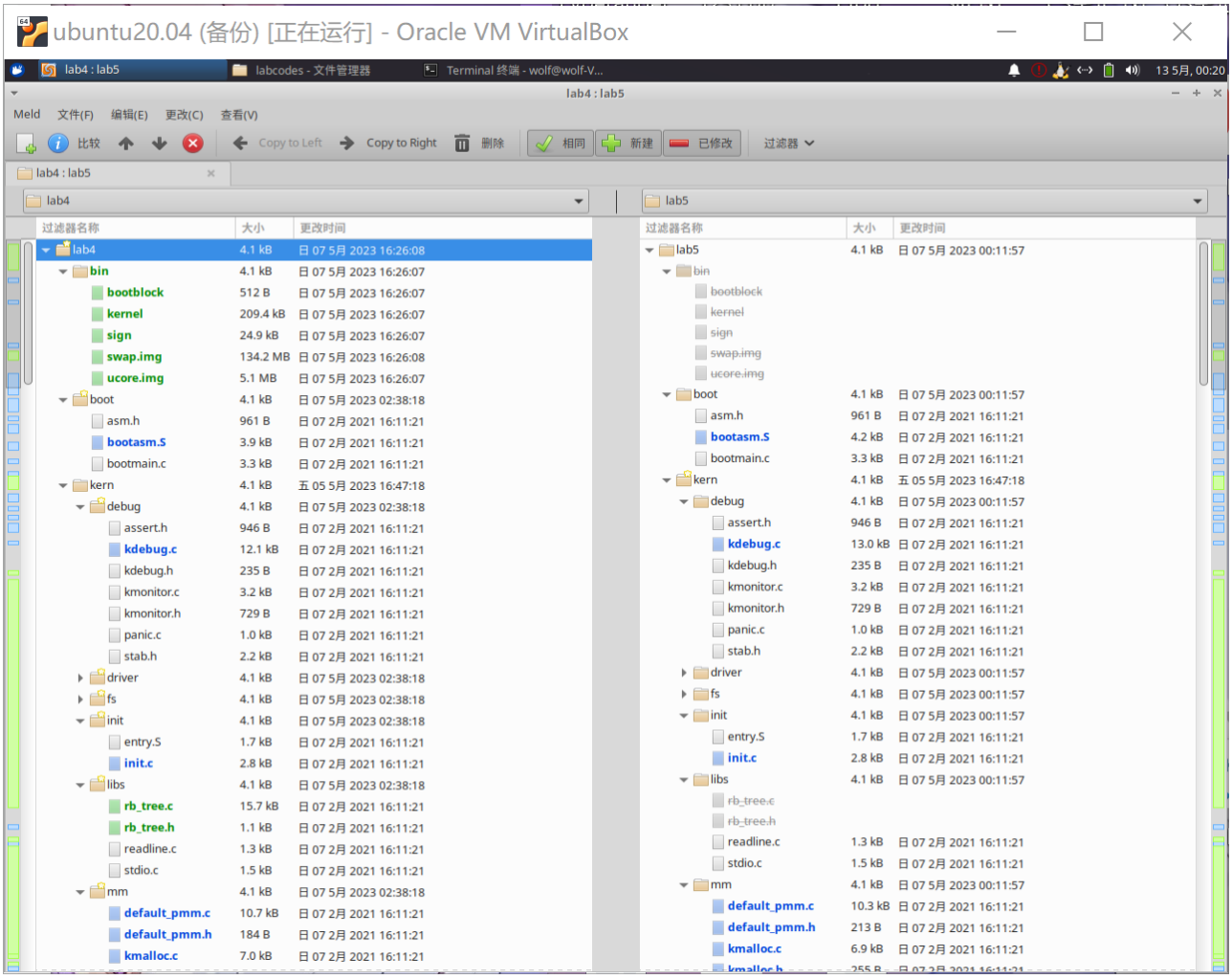
使用meld工具可以比较方便地查看Lab5与Lab4的差异，由于Lab1、Lab2、Lab3已经是被Lab4兼容了，所以不需要再做考虑。

其中，需要修改的部分为：

- proc.c
- default_pmm.c
- pmm.c

- swap_fifo.c
- vmm.c
- trap.c

我们对这些作出修改。



另外，根据实验指导书要求，本实验在此之外还需做如下改动：

1.proc_struct结构

为了能够管理进程，进程控制块中新增加了变量，记录等待状态和退出原因，并将相关进程通过链表链接起来。

- exit_code：记录进程的退出原因，这个值将传给等待的父进程
- wait_state：标记当前进程是否处于等待状态
- cptr：当前进程的子进程双向链表结点
- yptr/optr：当前进程的older sibling和younger sibling的双向链表结点

```
//路径：kern/process/proc.h（主体部分在Lab4中已经进行了说明）
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

    //以下是新增的部分
    int exit_code;                              // exit code (be
sent to parent proc)
    uint32_t wait_state;                        // waiting state
    struct proc_struct *cptr, *yptr, *optr;     // relations
between processes
};
```

2.alloc_proc

由于进程控制块增加了新的变量，分配进程控制块时也需要进行相应的初始化。只需要在alloc_proc添加，将等待状态设为0，链表节点设为NULL。（该段的主体实现是lab4）

```
//在Lab4练习1中实现的，这离需要新增初始化刚刚定义的几个变量
static struct proc_struct *alloc_proc(void) {
```

```
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;      //设置进程为未初始化状态
        proc->pid = -1;                 //未初始化的的进程id为-1
        proc->runs = 0;                 //初始化时间片
        proc->kstack = 0;               //内存栈的地址
        proc->need_resched = 0;         //是否需要调度：设为不需要
        proc->parent = NULL;            //父节点设为空
        proc->mm = NULL;                //虚拟内存设为空
        memset(&(proc->context), 0, sizeof(struct context));    //
上下文的初始化
        proc->tf = NULL;                //中断帧指针置为空
        proc->cr3 = boot_cr3;           //页目录设为内核页目录表的基址
        proc->flags = 0;                //标志位
        memset(proc->name, 0, PROC_NAME_LEN);   //进程名
        //以下两行是新增的代码（这四个指针的意义在前面讲了）
        proc->wait_state = 0;           //★PCB 进程控制块中新增的条目，
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
        //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
    }
    return proc;
}
```

3.do_fork

do_fork中进行了进程的复制，对新的进程控制块的设置也需要补充。确认等待状态为0，只有在wait状态时进程的wait_state才会被设置为等待，一旦被唤醒，在wakeup_proc就会重新设置为0，此时应该不在等待状态。调用set_links将新进程和相关进程建立联系。补充后的do_fork如下：

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe
*tf) {
    int ret = -E_NO_FREE_PROC;//尝试为进程分配内存
    struct proc_struct *proc;//定义新进程
    if (nr_process >= MAX_PROCESS) {//分配进程数大于 4096，返回
        goto fork_out;//返回
    }
    ret = -E_NO_MEM;//因内存不足而分配失败
```

```
    if((proc=alloc_proc())==NULL) {//调用 alloc_proc() 函数申请内存块，
如果失败，直接返回处理
        goto fork_out;//返回
    }
    assert(current->wait_state == 0);    //★改动1：确保当前进程正在等待
    proc->parent = current;//将子进程的父节点设置为当前进程
    if(setup_kstack(proc)) {//调用 setup_stack() 函数为进程分配一个内核
栈
        goto bad_fork_cleanup_proc;//返回
    }
    if(copy_mm(clone_flags,proc)) {//调用 copy_mm() 函数复制父进程的内存
信息到子进程
        goto bad_fork_cleanup_kstack;//返回
    }
    copy_thread(proc, stack, tf);//调用 copy_thread() 函数复制父进程的
中断帧和上下文信息
   //将新进程添加到进程的 hash 列表中
    bool intr_flag=0;
    local_intr_save(intr_flag);//屏蔽中断，intr_flag 置为 1
    {
        proc->pid = get_pid();//获取当前进程 PID
        hash_proc(proc);//建立 hash 映射
        set_links(proc);                    //★改动2：设置链表（替换了原来的
直接计数ticks++）
        //具体作用下面会解释，简单来说就是把fork出来的进程链接到旧进程上
        //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);//屏蔽中断，intr_flag 置为 1
    wakeup_proc(proc);//一切就绪，唤醒子进程
    ret=proc->pid;//返回子进程的 pid
fork_out: //已分配进程数大于 4096
    return ret;
bad_fork_cleanup_kstack: //分配内核栈失败
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

其中set_links()函数如下

```
//set_links会将进程加入进程链表，设置父进程的子进程为自己，找到自己的older
sibling进程，最后将进程数+1。
static void
set_links(struct proc_struct *proc) {
    list_add(&proc_list, &(proc->list_link));    //进程加入进程链表
    proc->yptr = NULL;                           //当前进程的 younger
sibling 为空
    if ((proc->optr = proc->parent->cptr) != NULL) {
        proc->optr->yptr = proc;                 //当前进程的 older
sibling 为当前进程
    }
    proc->parent->cptr = proc;                   //父进程的子进程为当前
进程
    nr_process ++;                               //进程数加一
}
```

4.idt_init

引入用户进程后，需要用户进程能够进行系统调用，即可以发起中断，进行特权级切换。系统调用的中断号是128，需要单独设置该中断向量的特权级为用户特权级，这样用户就可以通过该中断发起系统调用。

```
void idt_init(void) {
      extern uintptr_t __vectors[];
      int num=sizeof(idt)/sizeof(struct gatedesc);
      for(int i=0;i<num;i++){
            SETGATE(idt[i],1,GD_KTEXT,__vectors[i],DPL_KERNEL);
      }
      //改动：为T_SYSCALL设置用户态权限
      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
      lidt(&idt_pd);
}
```

设置一个特定中断号的中断门，专门用于用户进程访问系统调用。在上述代码中，可以看到在执行加载中断描述符表 lidt 指令前，专门设置了一个特殊的中断描述符 idt[T_SYSCALL]，它的特权级设置为 DPL_USER，中断向量处理地址在 `__vectors[T_SYSCALL]` 处。这样建立好这个中断描述符后，一旦用户进程执行 INT T_SYSCALL 后，由于此中断允许用户态进程产生（它的特权级设置为 DPL_USER），所以

CPU 就会从用户态切换到内核态，保存相关寄存器，并跳转到 `__vectors[T_SYSCALL]` 处开始执行，形成如下执行路径：

vector128(vectors.S)-->__alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)---->syscall(syscall.c)

5.trap_disptach

为了操作系统能正常进行进程调度，需要在时钟中断处，将进程的need_schedule设置为1，表示该进程时间配额已用完，需要调度运行其他程序。在trap调用trap_dispatch完成中断服务例程后，会判断这个值是否为1，然后调用need_schedule进行进程调度。

```
//trap_disptach
        case IRQ_OFFSET + IRQ_TIMER:
        ticks++;
        if(ticks%TICK_NUM==0) {
            assert(current != NULL);
            current->need_resched = 1;//★改动：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
        }
        break;
//trap中最后进行进程调度
void trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    // used for previous projects
    if (current == NULL) {
        trap_dispatch(tf);
    }
    else {
        // keep a trapframe chain in stack
        struct trapframe *otf = current->tf;
        current->tf = tf;

        bool in_kernel = trap_in_kernel(tf);            //是否是内核产生的中断

        trap_dispatch(tf);

        current->tf = otf;
        if (!in_kernel) {
            if (current->flags & PF_EXITING) {
```

```
                do_exit(-E_KILLED);
            }
            if (current->need_resched) {                //判断是否需
要调度
                schedule();
            }
        }
    }
}
```

练习1: 加载应用程序并执行（需要编码）

> ***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。

1.从Lab4更进一步

Lab4中已经实现了内核线程的创建，能够在内核态运行线程。在Lab5中需要实现用户进程的创建，并为用户进程提供一些系统调用，并对用户进程的执行进行基本的管理。

进程运行在用户态，有自己的地址空间。与内核相比，进程管理和内存管理这两个部分有很大的不同。

（1）进程管理

在进程管理方面，操作系统主要需要实现的有：

- 建立进程的页表和维护进程可访问空间
- 加载ELF格式的程序到进程控制块管理的内存中的方法
- 在进程复制（fork）过程中，把父进程的内存空间拷贝到子进程内存空间的技术

此外还需要实现与用户态进程生命周期管理相关的：

- 让进程放弃CPU而睡眠等待某事件
- 让父进程等待子进程结束
- 一个进程杀死另一个进程
- 给进程发消息
- 建立进程的关系链表

（2）内存管理

在内存管理方面，操作系统主要是需要维护进程的地址空间，即维护用户进程的页表，维护地址空间到物理内存的映射。不同的进程有各自的页表，即便不同进程的用户态虚拟地址相同，由于页表把虚拟页映射到了不同的物理页帧，不同进程的地址空间也不同，且相互隔离开。此外，在用户态内存空间和内核态内核空间之间需要拷贝数据，让CPU处在内核态才能完成对用户空间的读或写，为此需要设计专门的拷贝函数（copy_from_user和copy_to_user）完成。

（3）从内核线程到用户进程

在proc_init中，会建立第1个内核线程idle_proc，这个线程总是调度运行其他线程。然后proc_init会调用kernel_thread建立init_main线程，接着在init_main中将调用kernel_thread建立user_main线程。user_main仍然是一个内核线程，但他的任务是创建用户进程。在user_main中将调用KERNEL_EXECVE，从而调用kernel_execve来把某一具体程序(exit)的执行内容放入内存，覆盖user_main线程，此后就可以调度执行程序，该程序在用户态运行，此时也就完成了用户进程的创建。

具体代码如下：

```
//在user_main中调用KERNEL_EXECVE，覆盖掉user_main，创建用户进程
static int
user_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(exit);
#endif
    panic("user_main execve failed.\n");
}
```

2.加载应用程序

（1）产生中断

在user_main中，将调用KERNEL_EXECVE2加载用户程序，将该程序的内存空间替换掉当前线程的内存空间，将当前内核线程转变为要执行的进程。加载过程的第一步是由KERNEL_EXECVE2调用kernel_execve，发起系统调用。

```
static int
kernel_execve(const char *name, unsigned char *binary, size_t size)
{
    int ret, len = strlen(name);
    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL), "0" (SYS_exec), "d" (name), "c" (len),
"b" (binary), "D" (size)
        : "memory");
    return ret;
}

#define __KERNEL_EXECVE(name, binary, size) ({
     \
            cprintf("kernel_execve: pid = %d, name = \"%s\".\n",
     \
                    current->pid, name);
     \
            kernel_execve(name, binary, (size_t)(size));
     \
        })

#define KERNEL_EXECVE(x) ({
    \
            extern unsigned char
_binary_obj___user_##x##_out_start[],  \
                _binary_obj___user_##x##_out_size[];
     \
            __KERNEL_EXECVE(#x, _binary_obj___user_##x##_out_start,
    \
                            _binary_obj___user_##x##_out_size);
    \
```

```
        })
```

由于此时还没有建立文件系统，需要执行的用户程序是随ucore的kernel直接加载到内存中的，并使用全局变量记录了这段用户程序代码的起始位置和大小。从宏定义调用kernel_execve会将程序名，位置和大小都传入。kernel_execve将这些信息保存到指定的寄存器中，发起中断，进行系统调用，具体的细节在练习三中进行分析。中断进行系统调用时的调用顺序如下：

vector128(vectors.S)-->__alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)-->syscall(syscall.c)-->sys_exec（syscall.c）-->do_execve(proc.c)

最终系统调用将通过do_execve完成用户程序的加载。

```
//syscall.c，exec系统调用
static int
sys_exec(uint32_t arg[]) {
    const char *name = (const char *)arg[0];
    size_t len = (size_t)arg[1];
    unsigned char *binary = (unsigned char *)arg[2];
    size_t size = (size_t)arg[3];
    return do_execve(name, len, binary, size);
}
```

（2）.do_execve

接下来分析do_execve是怎样加载处于内存中的程序并建立好用户内存空间，并设置中断帧，完成用户进程创建并执行用户程序的。传入的参数为用户程序名和长度，用户程序代码位置和大小。do_execve完整的实现如下：

```
//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
int
do_execve(const char *name, size_t len, unsigned char *binary,
size_t size) {
    struct mm_struct *mm = current->mm;//获取当前进程的内存地址
    if (!user_mem_check(mm, (uintptr_t)name, len, 0)) {
        return -E_INVAL;
    }
    if (len > PROC_NAME_LEN) {
        len = PROC_NAME_LEN;
    }
```

```
    char local_name[PROC_NAME_LEN + 1];
    memset(local_name, 0, sizeof(local_name));
    memcpy(local_name, name, len);
     //为加载新的执行码做好用户态内存空间清空准备
    /*清空内存空间*/
    if (mm != NULL) {
        lcr3(boot_cr3);//设置页表为内核空间页表
        if (mm_count_dec(mm) == 0) {//如果没有进程再需要此进程所占用的内存空间
            exit_mmap(mm); //释放进程所占用户空间内存和进程页表本身所占空间
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;//把当前进程的 mm 内存管理指针为空
    }
    int ret;// 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。
    if ((ret = load_icode(binary, size)) != 0) {    //调用load_icode加载用户程序并完成后续工作
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

execve_exit:
    do_exit(ret);
    panic("already exit: %e.\n", ret);
}
```

进入do_execve后先进行程序名字长度的调整，然后就开始使用新进程覆盖原进程。首先，由于新进程将使用新的用户内存空间，原进程的内存空间需要进行清空。如果mm_struct为空，则原进程是内核线程，不需要处理。如果mm_struct不为空，设置页表为内核空间页表，将引用计数-1，如果引用计数为0则根据mm_struct记录的信息对原进程的内存空间进行释放。

```
//do_execve中将原内存空间清空
    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
```

```
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
//exit_mmap调用unmap_range,exit_range取消地址映射
void exit_mmap(struct mm_struct *mm) {
    assert(mm != NULL && mm_count(mm) == 0);
    pde_t *pgdir = mm->pgdir;
    list_entry_t *list = &(mm->mmap_list), *le = list;
    while ((le = list_next(le)) != list) {
        struct vma_struct *vma = le2vma(le, list_link);
        unmap_range(pgdir, vma->vm_start, vma->vm_end);
    }
    while ((le = list_next(le)) != list) {
        struct vma_struct *vma = le2vma(le, list_link);
        exit_range(pgdir, vma->vm_start, vma->vm_end);
    }
}
//put_pgdir释放页目录占用的内存空间
static void put_pgdir(struct mm_struct *mm) {
    free_page(kva2page(mm->pgdir));
}
//mm_destroy销毁mm_struct
void mm_destroy(struct mm_struct *mm) {
    assert(mm_count(mm) == 0);

    list_entry_t *list = &(mm->mmap_list), *le;
    while ((le = list_next(list)) != list) {
        list_del(le);
        kfree(le2vma(le, list_link));  //kfree vma
    }
    kfree(mm); //kfree mm
    mm=NULL;
}
```

接下来的工作就是加载elf格式的用户程序，申请新的用户内存空间，并设置中断帧，使用户进程最终可以运行。这些工作都是由load_icode函数完成的。

（3）.load_icode

该函数的功能主要分为 6 个部分，而我们需要填写的是第 6 个部分，就是伪造中断返回现场，使得系统调用返回之后可以正确跳转到需要运行的程序入口，并正常运行；而 1-5 部分则是一系列对用户内存空间的初始化，这部分将在 LAB8 的编码实现中具体体现，因此在本 LAB 中暂时不加具体说明；与 LAB1 的 challenge 类似的，第 6 个部分是在进行中断处理的栈（此时应当是内核栈）上伪造一个中断返回现场，使得中断返回的时候可以正确地切换到需要的执行程序入口处；在这个部分中需要对 tf 进行设置，不妨通过代码分析来确定这个 tf 变量究竟指到什么位置，该 tf 变量与 current->tf 的数值一致，而 current->tf 是在进行中断服务里程的 trap 函数中被设置为当前中断的中断帧，也就是说这个 tf 最终指向了当前系统调用 exec 产生的中断帧处；

load_icode完成了以下6个工作：

- 为新进程创建mm结构
- 创建新的页目录，并把内核页表复制到新创建的页目录，这样新进程能够正确映射内核空间
- 分配内存，从elf文件中复制代码和数据，初始化.bss段
- 建立用户栈空间
- 将新进程的mm结构设置为刚刚创建的mm
- 构造中断帧，使用户进程最终能够正确在用户态运行

①创建**mm**结构与新的页目录

前两个工作比较简单，只需要调用mm_create与setup_pgdir，完成mm结构的创建与新的页目录的创建，创建失败则需要将已创建的mm和页目录进行销毁。

```
static int load_icode(unsigned char *binary, size_t size) {
    if (current->mm != NULL) {//当前进程的内存为空
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;//记录错误信息：未分配内存
    struct mm_struct *mm;
    //创建一个mm_struct给用户程序使用
    if ((mm = mm_create()) == NULL) {//分配内存
        goto bad_mm;//分配失败，返回
    }
    //创建新的PDT，并把内核页表的内容复制到新的页目录
    if (setup_pgdir(mm) != 0) {//申请一个页目录表所需的空间
        goto bad_pgdir_cleanup_mm;//申请失败
    }
```

②创建虚拟内存空间**vma**

第三步主要是创建虚拟内存空间vma，根据elf文件头的信息复制代码段和数据段的数据，并将vma插入mm结构中，表示合法的用户虚拟空间。

```
    //(3) copy TEXT/DATA section, build BSS parts in binary to
memory space of process
    struct Page *page;
    //elf文件的ELF头部
    struct elfhdr *elf = (struct elfhdr *)binary;
    //确定elf文件的program section headers
    struct proghdr *ph = (struct proghdr *)(binary + elf-
>e_phoff);//获取段头部表的地址
    //确认为有效的elf文件
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;//读取的 ELF 文件不合法
        goto bad_elf_cleanup_pgdir;//返回
    }
    //开始创建虚拟空间并复制数据
    uint32_t vm_flags, perm;
    struct proghdr *ph_end = ph + elf->e_phnum;//段入口数目
    for (; ph < ph_end; ph ++) {
        //遍历每个program section headers（程序段）
        if (ph->p_type != ELF_PT_LOAD) {//当前段不能被加载
            continue ;                          //不是需要加载的段跳过
        }
        if (ph->p_filesz > ph->p_memsz) {//虚拟地址空间大小大于分配的物理
地址空间
            ret = -E_INVAL_ELF;                 //大小不正确
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {                //段大小为0，跳过
            continue ;
        }
        //调用mm_map进行vma的建立
        vm_flags = 0, perm = PTE_U;
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        //建立合法vma并插入mm结构维护的链表
```

```
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags,
NULL)) != 0) {
            goto bad_cleanup_mmap;
        }
        unsigned char *from = binary + ph->p_offset;
        size_t off, size;
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start,
PGSIZE);
        ret = -E_NO_MEM;
        //已建立了合法的vma，接下来分配物理内存
        end = ph->p_va + ph->p_filesz;
        //加载elf文件中的数据
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) ==
NULL) {       //分配页
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memcpy(page2kva(page) + off, from, size);
//数据复制
            start += size, from += size;
        }

        //设置.bss段
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            if (start == end) {
                continue ;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
//bss段清0
            start += size;
            assert((end < la && start == end) || (end >= la &&
start == la));
```

```
        }
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) ==
NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size);
//bss段清0
            start += size;
        }
    }
```

③建立用户栈并设立合法虚拟空间

接下来的第四步和第五步是建立用户栈，为用户栈设立合法虚拟空间，然后将已经设置好的mm设置为当前进程的mm。

```
    //建立用户栈，设置合法虚拟空间
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE,
vm_flags, NULL)) != 0) {
        goto bad_cleanup_mmap;
    }
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER)
!= NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE ,
PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE ,
PTE_USER) != NULL);
    assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE ,
PTE_USER) != NULL);
    //设置当前的mm，cr3
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));
```

★④构造用户进程中断帧

最后一步就是构造用户进程的中断帧，在load_icode、sys_exec函数返回，中断结束后，从中断帧恢复寄存器后回到用户态，降低特权级，能够执行用户进程的程序。中断帧中，cs，ds，ss，es寄存器设置为用户代码段和数据段的段寄存器，esp设置为用户栈的栈顶，eip设置为用户程序的入口，最后设置标志位，使用户进程可以被中断，这样中断帧就设置好了用户态下用户进程运行的环境。这一步是练习一中要求补全的部分，代码如下：

```
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    tf->tf_cs = USER_CS;//将 trapframe 的代码段设为 USER_CS
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;//将 trapframe 的数据段、附加段、堆栈段设为 USER_DS
    tf->tf_esp = USTACKTOP;//将 trapframe 的栈顶指针设为 USTACKTOP
    tf->tf_eip = elf->e_entry;//将 trapframe 的代码段指针设为 ELF 的入口地址 elf->e_entry
    tf->tf_eflags = FL_IF; //主要是打开中断
    ret = 0;//设置 ret 为 0，表示正常返回
```

★3.应用程序的运行

通过上述do_execve中的操作，原来的user_main已经被用户进程所替换掉了。此时处于RUNNABLE状态的是已经创建完成了的用户进程，系统调用已经完成，将按照调用的顺序一路返回到*_trapret*，从中断帧中恢复寄存器的值，通过*iret*回到用户进程*exit*的第一条语句（*initcode.S*中的start）开始执行。

★综上所述，一个用户进程创建到执行第一条指令的完整过程如下：

父进程通过fork系统调用创建子进程。通过do_fork进行进程资源的分配，创建出新的进程
fork返回0，子进程创建完成，等待调度。fork中将进程设置为RUNNABLE，该进程可以运行schedule函数进行调度，调用proc_run运行该进程
该进程调用kernel_execve，产生中断并进行exec系统调用
do_execve将当前进程替换为需要运行的用户进程，加载程序并设置好中断帧
从中断帧返回到用户态，根据中断帧中设置的eip，跳转执行用户程序的第一条指令

★问题

> 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。

分析在创建了用户态进程并且加载了应用程序之后，其占用 CPU 执行到具体执行应用程序的整个经过：

1. 在经过调度器占用了 CPU 的资源之后，用户态进程调用了 exec 系统调用，从而转入到了系统调用的处理例程；
2. 在经过了正常的中断处理例程之后，最终控制权转移到了 syscall.c 中的 syscall 函数，然后根据系统调用号转移给了 sys_exec 函数，在该函数中调用了上文中提及的 do_execve 函数来完成指定应用程序的加载；
3. 在do_execve中进行了若干设置，包括退出当前进程的页表，换用 kernel 的 PDT 之后，使用 load_icode 函数，完成了对整个用户线程内存空间的初始化，包括堆栈的设置以及将 ELF 可执行文件的加载，之后通过 current->tf 指针修改了当前系统调用的 trapframe，使得最终中断返回的时候能够切换到用户态，并且同时可以正确地将控制权转移到应用程序的入口处；
4. 在完成了 do_exec 函数之后，进行正常的中断返回的流程，由于中断处理例程的栈上面的 eip 已经被修改成了应用程序的入口处，而 CS 上的 CPL 是用户态，因此 iret 进行中断返回的时候会将堆栈切换到用户的栈，并且完成特权级的切换，并且跳转到要求的应用程序的入口处；
5. 接下来开始具体执行应用程序的第一条指令；

练习**2:** 父进程复制自己的内存空间给子进程（需要编码）

> 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*。

1.复制父进程的内存

在Lab4中已经分析过了do_fork函数对创建的进程的资源的分配，其中内存资源的分配是由copy_mm完成的，Lab4创建内核线程，因此没有进行内存的复制，在本实验中，copy_mm将为新进程分配内存空间，并将父进程的内存资源复制到新进程的内存空间。

调用流程如下：

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

接下来我们将分别研究这几个函数

(1)copy_mm

对于共享内存的线程或进程，不需要进行复制，根据clone_flags判断是共享时，可以直接返回父进程的mm。而不共享的情况下，首先创建一个mm_struct，调用setup_pgdir创建新的页目录，并将内核页目录复制到新的页目录，然后调用dup_mmap进行内存资源的复制。

```
static int
copy_mm(uint32_t clone_flags, struct proc_struct *proc) {
    struct mm_struct *mm, *oldmm = current->mm;
    /* current is a kernel thread */
    if (oldmm == NULL) {//当前进程地址空间为 NULL
        return 0;
    }
    if (clone_flags & CLONE_VM) {//可以共享地址空间
        mm = oldmm;//共享地址空间
        goto good_mm;
    }
    int ret = -E_NO_MEM;
    if ((mm = mm_create()) == NULL) {//创建地址空间未成功
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
    lock_mm(oldmm);//打开互斥锁,避免多个进程同时访问内存
    //定义在vmm.h中，进行上锁
    {
        ret = dup_mmap(mm, oldmm);      //★调用 dup_mmap 函数，进行内存资源的复制
    }
    unlock_mm(oldmm);//释放互斥锁
    if (ret != 0) {
        goto bad_dup_cleanup_mmap;
    }
```

```
good_mm:
    mm_count_inc(mm);//共享地址空间的进程数加一
    proc->mm = mm;//复制空间地址
    proc->cr3 = PADDR(mm->pgdir);//复制页表地址
    return 0;
bad_dup_cleanup_mmap:
    exit_mmap(mm);
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    return ret;
}
```

(2)dup_mmap

dup_mmap中，通过遍历mm_struct中的链表，给子进程分配所有父进程拥有的vma虚拟空间，将创建的vma插入mm中，并调用copy_range将父进程vma中的数据复制到子进程新创建的vma中。

```
//vmm.c中定义的dup_mmap
int
dup_mmap(struct mm_struct *to, struct mm_struct *from) {
    assert(to != NULL && from != NULL);//必须非空
    // mmap_list 为虚拟地址空间的首地址
    list_entry_t *list = &(from->mmap_list), *le = list;
    while ((le = list_prev(le)) != list) {//遍历所有段
        struct vma_struct *vma, *nvma;
        vma = le2vma(le, list_link);//获取某一段
        nvma = vma_create(vma->vm_start, vma->vm_end, vma-
>vm_flags);   //创建vma
        if (nvma == NULL) {
            return -E_NO_MEM;
        }
        insert_vma_struct(to, nvma);//向新进程插入新创建的段
//将新创建的vma插入mm
        //进行复制
        bool share = 0;//调用 copy_range 函数
        if (copy_range(to->pgdir, from->pgdir, vma->vm_start, vma-
>vm_end, share) != 0) {
            return -E_NO_MEM;
```

```
        }
    }
    return 0;
}
```

★(3)copy_range

copy_range中会把父进程的vma中的内容复制给子进程的vma。父进程和子进程的vma相同，但映射到的物理页不同。首先找到父进程的vma对应的页表项，从该页表项可以找到父进程的vma对应的物理页，然后为子进程创建页表项并分配新的一页，接下来将父进程vma物理页中的数据复制到子进程新分配出的一页，再调用page_insert将子进程的新页的页表项设置好，建立起虚拟地址到物理页的映射，这个vma的复制工作就完成了。具体的数据复制使用memcpy函数，需要传入虚拟地址，但此时处于内核态，因此还要把物理页的地址转换为该页对应内核的虚拟地址。这是练习二需要完成的部分，最终copy_range完整的实现如下：

```
int
copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
bool share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    //按页复制
    do {
        //找到父进程的页表项
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //建立新进程的页表项
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            //获取父进程的物理页
            struct Page *page = pte2page(*ptep);
            //为新进程分配物理页
            struct Page *npage=alloc_page();
            assert(page!=NULL);
            assert(npage!=NULL);
```

```
                int ret=0;
                //得到页的内核虚拟地址后使用memcopy复制
                //★以下为补全部分
                void *src_kva=page2kva(page);//返回父进程的内核虚拟页地址
                void *dst_kva=page2kva(npage);//返回子进程的内核虚拟页地址
                memcpy(dst_kva,src_kva,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)
                assert(ret == 0);
            }
            start += PGSIZE;
        } while (start != 0 && start < end);
        return 0;
}
```

补全部分代码的含义

1. 找到父进程指定的某一物理页对应的内核虚拟地址；
2. 找到需要拷贝过去的子进程的对应物理页对应的内核虚拟地址；
3. 将前者的内容拷贝到后者中去；
4. 为子进程当前分配这一物理页映射上对应的在子进程虚拟地址空间里的一个虚拟页；

2.Copy-on-Write机制

Copy on Write 是读时共享，写时复制机制。多个进程可以读同一部分数据，需要对数据进行写时再复制一份到自己的内存空间。具体的实现为，在fork时，直接将父进程的地址空间即虚拟地址复制给子进程，不分配实际的物理页给子进程，并将父进程所有的页都设置为只读。父子进程都可以读取该页，当父子进程写该页时，就会触发页访问异常，发生中断，调用中断服务例程，在中断服务例程中，将触发异常的虚拟地址所在的页复制，分配新的一页存放数据，这样父子进程写该部分数据时就各自可以拥有一份自己的数据。

大概的实现思路为：

复制父进程内存时直接将父进程的物理页映射到子进程的虚拟页，且父子进程的该页表项均修改为只读。（修改copy_range）
当父子进程需要写时，会触发页访问异常，在页访问异常中进行内存页的分配和复制（修改do_pgfault）

★问题

> 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。

接下来将说明如何实现 "Copy on Write" 机制，该机制的主要思想为使得进程执行 fork 系统调用进行复制的时候，父进程不会简单地将整个内存中的内容复制给子进程，而是暂时共享相同的物理内存页；而当其中一个进程需要对内存进行修改的时候，再额外创建一个自己私有的物理内存页，将共享的内容复制过去，然后在自己的内存页中进行修改；根据上述分析，主要对实验框架的修改应当主要有两个部分，一个部分在于进行 fork 操作的时候不直接复制内存，另外一个处理在于出现了内存页访问异常的时候，会将共享的内存页复制一份，然后在新的内存页进行修改，具体的修改部分如下：

- do fork 部分：在进行内存复制的部分，比如 copy_range 函数内部，不实际进行内存的复制，而是将子进程和父进程的虚拟页映射上同一个物理页面，然后在分别在这两个进程的虚拟页对应的 PTE 部分将这个页置成是不可写的，同时利用 PTE 中的保留位将这个页设置成共享的页面，这样的话如果应用程序试图写某一个共享页就会产生页访问异常，从而可以将控制权交给操作系统进行处理；
- page fault 部分：在 page fault 的 ISR 部分，新增加对当前的异常是否由于尝试写了某一个共享页面引起的，如果是的话，额外申请分配一个物理页面，然后将当前的共享页的内容复制过去，建立出错的线性地址与新创建的物理页面的映射关系，将 PTE 设置设置成非共享的；然后查询原先共享的物理页面是否还是由多个其它进程共享使用的，如果不是的话，就将对应的虚地址的 PTE 进行修改，删掉共享标记，恢复写标记；这样的话 page fault 返回之后就可以正常完成对虚拟内存（原想的共享内存）的写操作了；

上述实现有一个较小的缺陷，在于在 do fork 的时候需要修改所有的 PTE，会有一定的时间效率上的损失；可以考虑将共享的标记加在 PDE 上，然后一旦访问了这个 PDE 之后再将标记下传给对应的 PTE，这样的话就起到了标记延迟和潜在的标记合并的左右，有利于提升时间效率；

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（不需要编码）

> 请在实验报告中简要说明你对 *fork/exec/wait/exit*函数的分析。并回答如下问题：
>
> - 请分析*fork/exec/wait/exit*在实现中是如何影响进程的执行状态的？
> - 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）
>
> 执行：*make grade*。如果所显示的应用程序检测都输出*ok*，则基本正确。（使用的是*qemu-1.0.1*）

1.系统调用

用户进程在用户态下运行，不能执行特权指令，如果需要执行特权指令，只能通过系统调用切换到内核态，交给操作系统来完成。

用户库

为了简化应用程序进行系统调用方式，用户库中提供了对系统调用的封装。即只需要在程序中通过调用如exit，fork，wait等库函数，库函数将进行系统调用的发起。在ucore中，这部分封装放在user/libs/ulib.c中

```
void exit(int error_code) {
    sys_exit(error_code);
    cprintf("BUG: exit failed.\n");
    while (1);
}

int fork(void) {
    return sys_fork();
}

int wait(void) {
    return sys_wait(0, NULL);
}
......
```

最终这些库函数都会调用syscall.c中的syscall，只是传入的参数不同，在该函数中使用内联汇编直接发起中断，中断号为定义的T_SYSCALL（0x80），即系统调用为128号中断。进行中断调用时会向eax寄存器传入参数，这个参数表示发生了具体哪个系统调用，同时还可以根据需要传入最多5个参数，分别传入edx，ecx，ebx，edi和esi寄存器中。

```
libs/unistd.h:#define T_SYSCALL           0x80
static inline int
syscall(int num, ...) {
    va_list ap;
    va_start(ap, num);
    uint32_t a[MAX_ARGS];
    int i, ret;
    for (i = 0; i < MAX_ARGS; i ++) {
        a[i] = va_arg(ap, uint32_t);
    }
```

```
    va_end(ap);

    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a[0]),
          "c" (a[1]),
          "b" (a[2]),
          "D" (a[3]),
          "S" (a[4])
        : "cc", "memory");
    return ret;
}

int sys_exit(int error_code) {
    return syscall(SYS_exit, error_code);
}
```

系统中断

用户态下发起中断后，就可以跳转执行对应的中断服务例程，而中断服务例程的地址保存在idt表中，idt表在内核启动后的kern_init中调用idt_init进行初始化，这是在lab1中已经完成的。设置中断服务例程时，进入中断服务例程的特权级均设置为内核特权级，本实验中，系统调用由用户发起，因此需要单独设置中断表idt中128号中断描述符的特权级为用户特权级，这样用户就可以通过中断提升特权级，进行系统调用。

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i = 0;
    for (i = 0; i < (sizeof(idt) / sizeof(struct gatedesc)); i++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    //为T_SYSCALL设置用户态权限,类型为系统调用
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL],
DPL_USER);
    lidt(&idt_pd);
}
```

根据设置好的中断描述符就可以切换特权级，进入系统调用对应的中断服务例程了。在进行特权级切换及进入中断服务历程之前，首先在**alltraps**完成中断帧**trapframe**的建立，在执行完**trap**后会回到alltraps，在_trapret中从中断帧恢复寄存器的值，中断返回。

```
.globl vector128
vector128:
  pushl $0
  pushl $128
  jmp __alltraps
//建立trapframe
.globl __alltraps
__alltraps:
    # push registers to build a trap frame
    # therefore make the stack look like a struct trapframe
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal

    # load GD_KDATA into %ds and %es to set up data segments for
kernel
    movl $GD_KDATA, %eax
    movw %ax, %ds
    movw %ax, %es

    # push %esp to pass a pointer to the trapframe as an argument
to trap()
    pushl %esp

    # call trap(tf), where tf=%esp
    call trap
```

接下来将调用trap，并在trap中根据情况调用trap_dispatch，trap_dispatch中根据trapframe中的tf_trapno进行相应的处理，这个值是一开始就被压入栈中的中断号128，将调用syscall。

```
//trap_dispatch
switch (tf->tf_trapno) {
    ...
    case T_SYSCALL:
        syscall();
        break;
    ...
}
```

在syscall中，将根据发出中断调用时传入eax寄存器的值判断系统调用具体类型，调用对应的函数。

```
void
syscall(void) {
    struct trapframe *tf = current->tf;
    uint32_t arg[5];
    int num = tf->tf_regs.reg_eax;
    if (num >= 0 && num < NUM_SYSCALLS) {
        if (syscalls[num] != NULL) {
            arg[0] = tf->tf_regs.reg_edx;
            arg[1] = tf->tf_regs.reg_ecx;
            arg[2] = tf->tf_regs.reg_ebx;
            arg[3] = tf->tf_regs.reg_edi;
            arg[4] = tf->tf_regs.reg_esi;
            tf->tf_regs.reg_eax = syscalls[num](arg);
            return ;
        }
    }
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
            num, current->pid, current->name);
}
```

ucore一共提供了以下这些系统调用：

```
static int (*syscalls[])(uint32_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_yield]             sys_yield,
    [SYS_kill]              sys_kill,
    [SYS_getpid]            sys_getpid,
    [SYS_putc]              sys_putc,
    [SYS_pgdir]             sys_pgdir,
};
```

下面是表格化的总结

系统调用名	含义	具体完成服务的函数
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 pid	

而这些函数最终会调用do_fork，do_exit等函数完成需要完成的任务，然后返回值存放在eax寄存器中，一路返回到__trapret，从中断栈恢复寄存器的值，回到用户态，中断结束，继续正常运行进程。

2.fork

调用过程为：`fork->SYS_fork->do_fork+wakeup_proc`

fork用于创建新的进程。进程调用fork函数，将通过系统调用，创建一个与原进程相同的进程，该进程与原进程内存相同，执行相同的代码，但有自己的地址空间。对于父进程，fork返回子进程的pid，创建出的子进程从fork返回0。一次具体的fork调用从调用fork用户库函数开始，调用包装好的fork。

```
int fork(void) {
    return sys_fork();
}
```

包装好的用户库函数将进一步调用sys_fork，在sys_fork中将调用syscall，传入SYS_fork，即系统调用类型：

```
int
sys_fork(void) {
    return syscall(SYS_fork);
}
```

在syscall中将发起中断，传入相关参数，通过int指令发起128号中断。发生中断首先进行用户栈到特权栈的切换，在__alltraps函数中建立trapframe，然后call trap，进行中断处理，中断处理将使用中断号128，从中断表中进入对应的中断服务例程即syscall系统调用，在syscall中，根据传入的SYS_fork确定系统调用的具体类型，然后就将进入对应的系统调用函数：

```
static int (*syscalls[])(uint32_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_yield]             sys_yield,
    [SYS_kill]              sys_kill,
    [SYS_getpid]            sys_getpid,
    [SYS_putc]              sys_putc,
    [SYS_pgdir]             sys_pgdir,
};
```

最终将进入do_fork进行进程的复制，该函数在Lab4中已经完成。该函数将创建一个新的进程控制块管理新的进程，然后调用copy_mm，copy_thread等函数给新的进程分配资源，并复制父进程的内存，在copy_thread中复制父进程的上下文和中断帧时，设置中断帧的eax值为0，这样复制出的子进程在将来返回时将返回0，且eip设置为forkret，调度运行子进程时，会进行上下文切换进入forkret，然后从中断帧恢复寄存器，返回0。

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct
trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
//内核栈顶
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;                  //子进程返回0
    proc->tf->tf_esp = esp;                         //父进程的用户栈指针
    proc->tf->tf_eflags |= FL_IF;                   //设置能够响应中断
    proc->context.eip = (uintptr_t)forkret;         //返回
    proc->context.esp = (uintptr_t)(proc->tf);      //trapframe
}
```

而父进程将返回子进程的pid。

```
//do_fork返回
    ...
    ret=proc->pid;
fork_out:
    return ret;
    ...
```

完成子进程的创建工作之后，将从do_fork按调用顺序返回至**trapret**，从**trapframe**恢复状态，返回到用户库的**syscall**，最后返回用户程序调用**fork**处继续执行下一条语句。此时，父子进程同时存在，此后如果发生调度，子进程也将通过上下文切换，从**forkret**返回trapret，最后返回到用户程序的下一条语句。

完整的一次fork调用的调用顺序如下：

```
fork-->sys_fork-->syscall-->int 0x80发起128号中断-->__alltraps--
>trap_dispatch-->syscall-->sys_fork
-->do_fork
```

完成调用后，父进程状态不变，子进程创建成功，为可运行状态，等待调度。

3.exec

调用过程为：`SYS_exec->do_execve`

在本实验的ucore代码中，没有提供用户库包装的exec，如果编写类似fork的包装，调用情况与fork是完全相同的。本实验中，直接在user_main中使用宏定义发起中断，最终调用do_execve，将user_main替换为exit.c中的用户程序，调用顺序如下：

```
user_main-->KERNEL_EXECVE-->__KERNEL_EXECVE-->kernel_execve-->int 0x80发起128号中断-->__alltraps-->trap_dispatch-->syscall-->sys_exec-->do_execve
```

在do_execve调用的load_icode中，对中断帧进行了设置，将eip设置为了elf文件中给出的程序入口，即用户程序的入口，_start。接下来将调用umain，从umain进入exit.c的main中开始执行程序。

```
//_start
.text
.globl _start
_start:
    # set ebp for backtrace
    movl $0x0, %ebp

    # move down the esp register
    # since it may cause page fault in backtrace
    subl $0x20, %esp

    # call user-program function
    call umain
//umain
int main(void);
void
umain(void) {
    int ret = main();
    exit(ret);
}
```

通过exec（在本实验中是KERNEL_EXECVE），当前正在执行的进程可以发起系统调用，然后通过do_execve，创建一个新的进程，建立完全不同的地址空间，从elf文件中加载代码和数据信息，进行好加载工作后，设置中断帧，使中断返回时能返回到新的进程的程序入口，这样返回后就开始执行新的程序。通过这一系列工作，这个新的程序就可以将原来的程序替

换掉。看起来进程只是发生了一次系统调用，但系统调用结束后，进程的地址空间，代码，数据等已经完全被替换掉，然后开始正常运行新的程序。

其中do_execve()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）

4.wait

调用过程为：`SYS_wait->do_wait`

wait函数用于让当前进程等待他的子进程结束。ucore提供了用户库包装后的wait和waitpid，wait是使用默认参数的waitpid，即等待任意进程结束。这里对waitpid进行分析，waitpid调用的过程与fork类似：

```
waitpid-->sys_wait-->syscall-->int 0x80发起128号中断-->__alltraps-->trap_dispatch-->syscall-->sys_wait-->do_wait
```

最终将调用系统调用函数do_wait，do_wait中会寻找是否有子进程为僵尸态（PROC_ZOMBIE），如果没有则将运行schedule调度其他进程运行，当前进程睡眠（PROC_SLEEPING），当有子进程运行结束转变为僵尸态，这个进程将被唤醒，从进程链表删除子进程，并将子进程的进程控制块也释放，彻底结束子进程，然后返回。传入的参数为0则等待任意子进程结束，否则等待指定的子进程结束。

```
int
do_wait(int pid, int *code_store) {
    struct mm_struct *mm = current->mm;
    if (code_store != NULL) {                              //存放导致子进程退出的退出码
        if (!user_mem_check(mm, (uintptr_t)code_store, sizeof(int), 1)) {
            return -E_INVAL;
        }
    }
```

```
    struct proc_struct *proc;
    bool intr_flag, haskid;
repeat:
    haskid = 0;
    //如果pid! =0，则找到进程id为pid的处于退出状态的子进程
    if (pid != 0) {                                         //等待指定pid
的子进程
        proc = find_proc(pid);
        if (proc != NULL && proc->parent == current) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    else {
        //如果pid==0，则随意找一个处于退出状态的子进程
        proc = current->cptr;                               //等待任意子
进程
        for (; proc != NULL; proc = proc->optr) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    if (haskid) {//如果没找到，则父进程重新进入睡眠，并重复寻找的过程
        current->state = PROC_SLEEPING;                     //进入睡眠状
态
        current->wait_state = WT_CHILD;                     //等待状态-等
待子进程
        schedule();                                         //调度
        if (current->flags & PF_EXITING) {                  //如果当前进
程已经结束，do_exit
            do_exit(-E_KILLED);
        }
        goto repeat;
    }
    return -E_BAD_PROC;
//释放子进程的所有资源
found:
    if (proc == idleproc || proc == initproc) {
```

```
        panic("wait idleproc or initproc.\n");          //不可以等待
init_proc和idle_proc结束
    }
    if (code_store != NULL) {
        *code_store = proc->exit_code;
    }
    local_intr_save(intr_flag);
    {
        unhash_proc(proc);//将子进程从hash_list中删除
        remove_links(proc);//将子进程从proc_list中删除
    }
    local_intr_restore(intr_flag);
    put_kstack(proc);                                   //释放子进程
的内核堆栈
    kfree(proc);                                        //释放子进程
的进程控制块
    return 0;
}
```

调用waitpid，当前进程将等待子进程运行结束，未结束时，当前进程将进入睡眠状态，直到子进程结束。等到了子进程的结束，do_wait中会将子进程从进程链表删除，让子进程彻底结束。

do_wait()函数的总结如下

- 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。

5.exit

调用过程为：`SYS_exit->exit`

exit用于退出并结束当前进程，也已经进行了包装，用户程序可以直接调用。

```
void exit(int error_code) {
    sys_exit(error_code);
    cprintf("BUG: exit failed.\n");
    while (1);
}
```

调用过程和fork，waitpid类似，最终调用do_exit。

```
exit-->sys_exit-->syscall-->int 0x80发起128号中断-->__alltraps-->trap_dispatch-->syscall-->sys_exit-->do_exit
```

在do_exit中，该进程的内存资源将被释放，同时状态将被设置为PROC_ZOMBIE，最后从进程链表删除该进程由他的父进程来完成，因此会判断其父进程是否在等待，如果等待则将父进程唤醒。最后还要处理该进程的子进程，因为他结束后无法处理自己的子进程，就遍历链表将子进程全部设置为init_proc的子进程，让init_proc完成PROC_ZOMBIE状态的子进程最后的处理。

```
int
do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }
    //释放内存
    struct mm_struct *mm = current->mm;
    if (mm != NULL) {//如果该进程是用户进程
        lcr3(boot_cr3);//切换到内核态的页表
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);                                  //取消映射
            /*如果没有其他进程共享这个内存释放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*/
            put_pgdir(mm);//释放页目录占用的内存
```

```
            mm_destroy(mm);//释放mm占用的内存
        }
        current->mm = NULL;//虚拟内存空间回收完毕
    }
    current->state = PROC_ZOMBIE;//PROC_ZOMBIE状态
    current->exit_code = error_code;//等待父进程做最后的回收

    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);//如果父进程在等待子进程，则唤醒
        }
        while (current->cptr != NULL) {/*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。*/
            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
            proc->parent = initproc;
            initproc->cptr = proc;
            //如果子进程已经为PROC_ZOMBIE且init_proc在等待，唤醒init_proc
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
        }
    }
    local_intr_restore(intr_flag);

    schedule(); //调度其他进程
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

综上所述，调用exit会让当前进程结束，释放所有的内存资源，但这个进程将仍以PROC_ZOMBIE状态存在，等待父进程做最后的处理，并且该进程结束前也会把自己的子进程交给init_proc，确保自己的子进程也可以最终被彻底结束，然后就调用schedule，调度运行其他进程。

6.用户态进程的生命周期

用户态进程的生命周期可用下图表示：

一个进程首先由父进程fork产生，状态会由刚分配进程控制块的UNINIT状态转变为RUNNABLE状态，为就绪状态。当发生调度选中次进程时，调度程序调用proc_run切换到该进程，该进程进入运行态。此后子进程可以通过execve发起系统调用，将自己替换为用户程序，但进程状态不会发生改变。父进程可以通过wait发起系统调用，将自己转变为SLEEPING休眠态，等待子进程的完成。子进程运行结束后将变为ZOMBIE僵尸态，而父进程将被唤醒，进行子进程资源的回收。此后父进程回到运行态（或就绪态），而子进程已彻底结束。

★问题1 fork/exec/wait/exit影响进程的执行状态

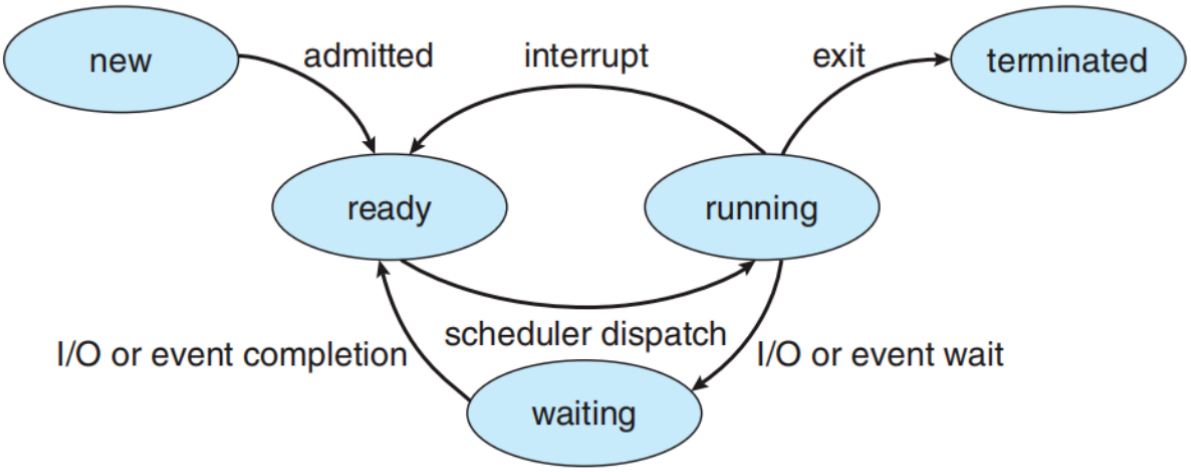
请分析 fork/exec/wait/exit 在实现中是如何影响进程的执行状态的？

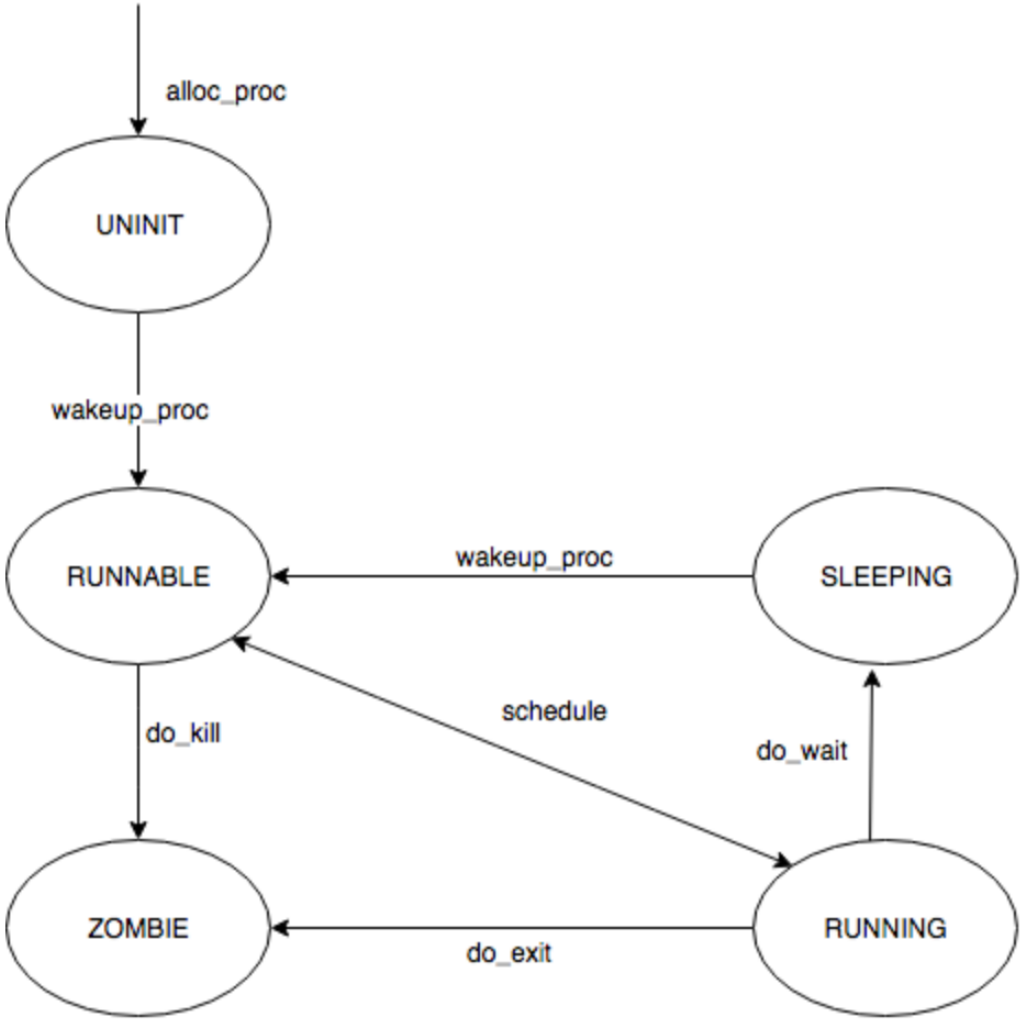
- fork 执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID。我们可以通过 fork 返回的值来判断当前进程是子进程还是父进程。fork 不会影响当前进程的执行状态，但是会将子进程的状态标记为 RUNNALB，使得可以在后续的调度中运行起来；
- exec 完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。exec 不会影响当前进程的执行状态，但是会修改当前进程中执行的程序；
- wait 是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。wait 系统调用取决于是否存在可以释放资源（ZOMBIE）的子进程，如果有的话不会发生状态的改变，如果没有的话会将当前进程置为 SLEEPING 态，等待执行了 exit 的子进程将其唤醒；
- exit 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。exit 会将当前进程的状态修改为 ZOMBIE 态，并且会将父进程唤醒（修改为RUNNABLE），然后主动让出 CPU 使用权；

★问题2 生命周期图

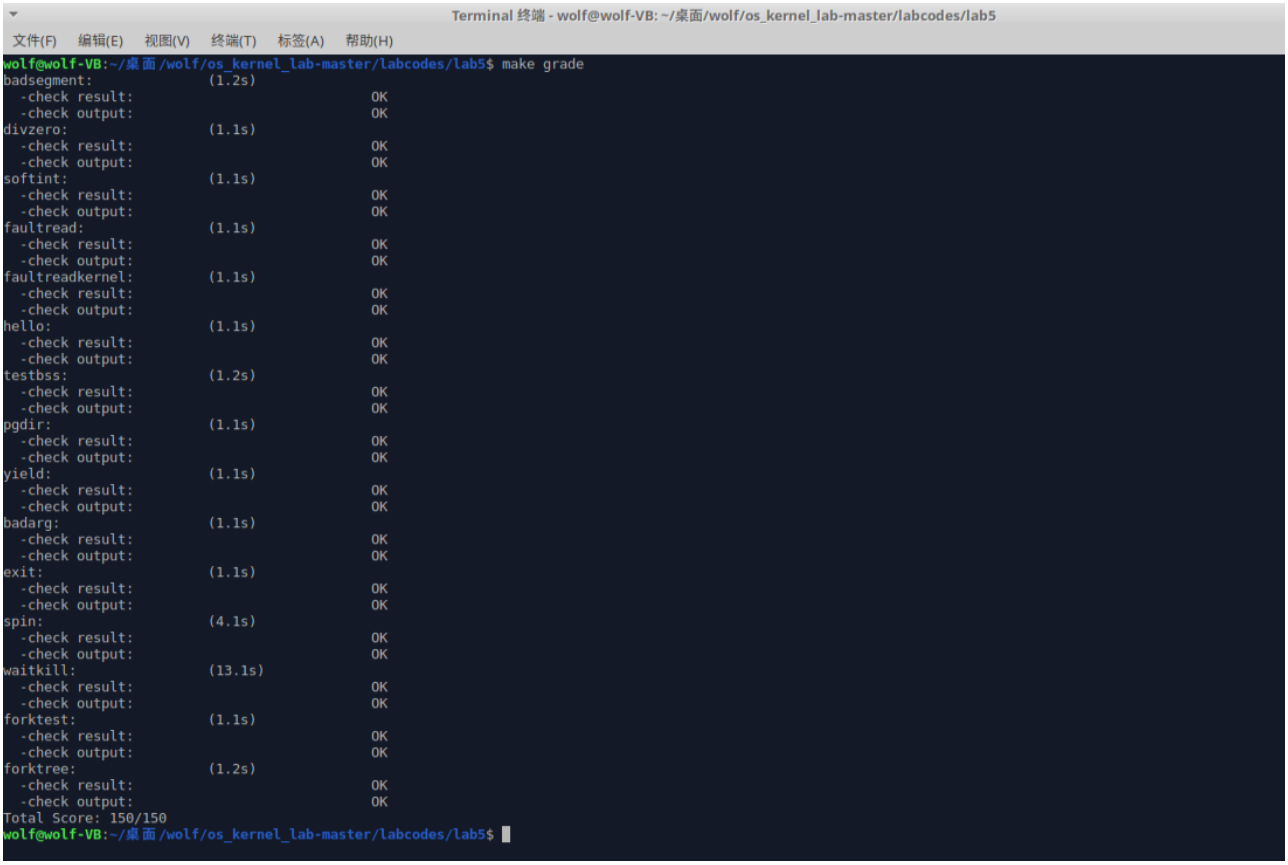
请给出 ucore 中一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）。（字符方式画即可）





实验结果

使用make grade执行后结果如下：



表示实验正确。

扩展练习 **Challenge**：实现 **Copy on Write**（**COW**）机制

> 给出实现源码,测试用例和设计报告（包括在*cow*情况下的各种状态转换（类似有限状态自动机）的说明）。
>
> 这个扩展练习涉及到本实验和上一个实验“虚拟内存管理”。在*ucore*操作系统中，当一个用户父进程创建自己的子进程时，父进程会把其申请的用户空间设置为只读，子进程可共享父进程占用的用户内存空间中的页面（这就是一个共享的资源）。当其中任何一个进程修改此用户内存空间中的某页面时，*ucore*会通过*page fault*异常获知该操作，并完成拷贝内存页面，使得两个进程都有各自的内存页面。这样一个进程所做的修改不会被另外一个进程可见了。请在*ucore*中实现这样的*COW*机制。
>
> 由于*COW*实现比较复杂，容易引入*bug*，请参考 *https://dirtycow.ninja/* 看看能否在*ucore*的*COW*实现中模拟这个错误和解决方案。需要有解释。

设置共享标志

在 vmm.c 中将 dup_mmap 中的 share 变量的值改为 1，启用共享：

```
int dup_mmap(struct mm_struct *to, struct mm_struct *from) {
        ...
        bool share = 1;
        ...
}
```

映射共享页面

在 pmm.c 中为 copy_range 添加对共享的处理，如果 share 为 1，那么将子进程的页面映射到父进程的页面。由于两个进程共享一个页面之后，无论任何一个进程修改页面，都会影响另外一个页面，所以需要子进程和父进程对于这个共享页面都保持只读。

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t
end, bool share) {
    ...
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);//获取父进程的虚拟地址
            struct Page *page = pte2page(*ptep);//获取父进程的物理页
            assert(page!=NULL);//断言：原页面必须非空
            int ret=0;//默认设置正常返回值
            if (share) {//启用共享
                // share page
                page_insert(from, page, start, perm & (~PTE_W));
                //启用共享，但修改权限（perm是原权限）
                ret = page_insert(to, page, start, perm &
(~PTE_W));//返回值，若成功插入则返回0
            } else {
                // alloc a page for process B
                struct Page *npage=alloc_page();//新建物理页
                assert(npage!=NULL);//断言：新创建页面必须非空
                uintptr_t src_kvaddr = page2kva(page);//通过物理页获取
page（原页）内核虚拟地址
                uintptr_t dst_kvaddr = page2kva(npage);//通过物理页获
取npage（新页）内核虚拟地址
                memcpy(dst_kvaddr, src_kvaddr, PGSIZE);//使用memcpy
函数进行拷贝
                ret = page_insert(to, npage, start, perm);//返回值，
若成功插入则返回0
```

```
            }
            assert(ret == 0);
        }
        ...
    return 0;
}
```

修改时拷贝

当程序尝试修改只读的内存页面的时候，将触发Page Fault中断，在错误代码中 P=1，W/R=1[OSDev]。因此，当错误代码最低两位都为 1 的时候，说明进程访问了共享的页面，内核需要重新分配页面、拷贝页面内容、建立映射关系：

```
//在vmm.c内
int do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t
addr) {
    ...
    if (*ptep == 0) {
        ...
    } else if (error_code & 3 == 3) {   // copy on write
        struct Page *page = pte2page(*ptep);//从页表项获取相应物理页的
Page构造体（找到物理址）
        struct Page *npage = pgdir_alloc_page(mm->pgdir, addr,
perm);//新建物理页的Page构造体
        uintptr_t src_kvaddr = page2kva(page);//通过物理页获取page（原
页）内核虚拟地址
        uintptr_t dst_kvaddr = page2kva(npage);//通过物理页获取npage
（新页）内核虚拟地址
        memcpy(dst_kvaddr, src_kvaddr, PGSIZE);//使用memcpy函数进行拷
贝
        //原型void *memcpy(void *destin, void *source, unsigned n),
PGSIZE为拷贝个数
    } else {
        ...
    }
    ...
}
```

实验总结

重要知识点

- 用户进程的创建过程
- 加载用户程序的过程
- fork对父进程内存资源的复制
- copy-on-write机制
- 系统调用

参考文献

实验理解与流程主要参考

https://blog.csdn.net/Aaron503/article/details/130453812?spm=1001.2014.3001.5501

challenge部分主要参考

https://github.com/AngelKitty/review_the_national_post-graduate_entrance_examination/blob/master/books_and_notes/professional_courses/operating_system/sources/ucore_os_lab/docs/lab_report/lab5/lab5%20%E5%AE%9E%E9%AA%8C%E6%8A%A5%E5%91%8A.md

【hidden】错误解决方法：

若出现!! error: missing 'check_slab() succeeded!'，这是由于lab4的challenge我们没有实现，此时

删除os_kernel_lab-master/labcodes/lab5/tools/grade.sh中340行左右的这一句

'check_slab() succeeded!'

即可

出现!! error: missing 'init check memory pass.'，

删除倒数的两个run_test -prog 'forktest' -check default_check中的

'init check memory pass.'即可