lab5 用户进程管理

练习0 修改代码

用meld合并的时候
proc.c
default_pmm.c
pmm.c
swap_fifo.c
vmm.c
trap.c
改进后的alloc_proc函数：

```
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; //PCB新增的条目，初始化进程等待状态
        proc->cptr = proc->optr = proc->yptr = NULL;//设置指针
    }
    return proc;
}
```

增加的代码：

```
proc->wait_state = 0; //初始化进程等待状态
proc->cptr = proc->optr = proc->yptr = NULL; //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。其中的wait_state是进程控制块中新增的条目。避免之后由于未定义或未初始化导致管理用户进程时出现错误。
指针解释如下：
process relations
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)

改进的do_fork函数：

```
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); //确保进程在等待
    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); //设置进程链接
    }
    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;
}
```

新增加代码：

```
assert(current->wait_state == 0); //确保进程在等待
set_links(proc); //设置进程链接
```

第一行代码需要确保当前进程正在等待，我们在alloc_proc中初始化wait_state为0。do_fork函数整体未较大改动，主要修改部分为将子进程的父进程设置为 current process，并且确保current process 的 wait_state 为0，因此我们可以用一个assert()实现该功能。还有就是插入新进程到进程哈希表和进程链表时，设置好相关进程的链接。设置链接的函数为 set_links ，这里较为坑的地方在于set_links函数中已经实现了将进程插入链表并将进程总数加1，因此需要删掉lab4中这两句代码。

查看set_links函数:

```
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
}
```

set_links函数的作用就是设置当前进程的process relations。

改进 idt_init 函数:

```
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);
}
```

多增加代码：

```
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
```

增添功能为：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。设置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“INTT_SYSCALL”后，由于此中断允许用户态进程产生（注意它的特权级设置为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

改进trap_dispatch函数:

```
ticks ++;
        if (ticks % TICK_NUM == 0) {
```

```
            assert(current != NULL);
            current->need_resched = 1;
        }
        break;
```

增加代码:

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

主要修改地方在于 当分配给进程的时间片用完时，设置进程为需要被调度

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。

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。
load_icode函数

```
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;
    //(1) create a new mm for current process
    if ((mm = mm_create()) == NULL) { //分配内存
        goto bad_mm; //分配失败，返回
    }
    //(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
```

```
    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
    }
    //(3) copy TEXT/DATA section, build BSS parts in binary to memory space of process
    struct Page *page;
    //(3.1) get the file header of the bianry program (ELF format)
    struct elfhdr *elf = (struct elfhdr *)binary;
    //(3.2) get the entry of the program section headers of the bianry program (ELF format)
    struct proghdr *ph = (struct proghdr *)(binary + elf->e_phoff); //获取段头部表的地址
    //(3.3) This program is valid?
    if (elf->e_magic != ELF_MAGIC) { //读取的ELF文件不合法
        ret = -E_INVAL_ELF; //ELF文件不合法错误
        goto bad_elf_cleanup_pgdir; //返回
    }

    uint32_t vm_flags, perm;
    struct proghdr *ph_end = ph + elf->e_phnum;//段入口数目
    for (; ph < ph_end; ph ++) { //遍历每一个程序段
    //(3.4) find every program section headers
        if (ph->p_type != ELF_PT_LOAD) { //当前段不能被加载
            continue ; //continue
        }
        //虚拟地址空间大小大于分配的物理地址空间
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) { //当前段大小为0
            continue ;
        }
    //(3.5) call mm_map fun to setup the new vma ( ph->p_va, ph->p_memsz)
        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;
        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;

    //(3.6) alloc memory, and  copy the contents of every program section (from,
from+end) to process's memory (la, la+end)
        end = ph->p_va + ph->p_filesz;
    //(3.6.1) copy TEXT/DATA section of bianry program
        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;
            }
            memcpy(page2kva(page) + off, from, size);
            start += size, from += size;
        }

      //(3.6.2) build BSS section of binary program
        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);
            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);
            start += size;
        }
    }
    //(4) build user stack memory
    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);

    //(5) set current process's mm, sr3, and set CR3 reg = physical addr of Page Directory
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

    //(6) setup trapframe for user environment
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    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;
```

```
    ret = 0;
out:
    return ret;
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

总结一下就是：

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;

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。

查看do_execve函数：

```
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);

    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    int ret;
    if ((ret = load_icode(binary, size)) != 0) {
        goto execve_exit;
    }
    set_proc_name(current, local_name);
    return 0;

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

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。

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。
函数调用过程：
do_fork()---->copy_mm()---->dup_mmap()---->copy_range()
查看do_fork函数：

```
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) { //分配内存失败
        goto fork_out; //返回
    }
    proc->parent = current; //设置父进程名字
    if (setup_kstack(proc) != 0) {//分配内核栈
        goto bad_fork_cleanup_proc; //返回
    }
    if (copy_mm(clone_flags, proc) != 0) { //复制父进程内存信息
        goto bad_fork_cleanup_kstack; //返回
    }
    copy_thread(proc, stack, tf); //复制中断帧和上下文信息
    bool intr_flag;
    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
        list_add(&proc_list,&(proc->list_link));//加入进程链表
        nr_process ++;  //进程数加一
    }
    local_intr_restore(intr_flag); //恢复中断
    wakeup_proc(proc); //唤醒新进程
    ret = proc->pid; //返回当前进程的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;
}
```

再查看copy_mm函数：本函数在lab4中没有实现:

```
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
    {
        ret = dup_mmap(mm, oldmm); //调用dup_mmap函数
    }
    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;
}
```

查看dup_mmap函数：

```
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);
        if (nvma == NULL) {
            return -E_NO_MEM;
        }
```

```
            insert_vma_struct(to, nvma); //向新进程插入新创建的段
            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;
    }
```

实现copy_range函数：

```
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));
    // copy content by page unit.
    do {
        //call get_pte to find process A's pte according to the addr start
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue ;
        }
        //call get_pte to find process B's pte according to the addr start. If pte is NULL, just
alloc a PT
        if (*ptep & PTE_P) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
        uint32_t perm = (*ptep & PTE_USER);
        //get page from ptep
        struct Page *page = pte2page(*ptep);
        // alloc a page for process B
        struct Page *npage=alloc_page();
        assert(page!=NULL);
        assert(npage!=NULL);
```

```
            int ret=0;
            //返回父进程的内核虚拟页地址
            void * kva_src = page2kva(page);
            //返回子进程的内核虚拟页地址
            void * kva_dst = page2kva(npage);
           //复制父进程到子进程
            memcpy(kva_dst, kva_src, PGSIZE);
            //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)
            ret = page_insert(to, npage, start, perm);
            assert(ret == 0);
            }
            start += PGSIZE;
        } while (start != 0 && start < end);
        return 0;
    }
```

copy_range函数就是调用一个memcpy将父进程的内存直接复制给子进程即可.
思考题：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。
在创建子进程时，将父进程的PDE直接赋值给子进程的PDE，但是需要将允许写入的标志位置0；当子进程需要进行写操作时，再次出发中断调用do_pgfault()，此时应给子进程新建PTE，并取代原先PDE中的项，然后才能写入。

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。并回答如下问题：

1 请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的?
2 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）
fork：调用过程：
fork->SYS_fork->do_fork+wakeup_proc
练习2已经介绍了do_fork函数，主要工作：

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。
wakeup_proc函数主要是将进程的状态设置为等待。
exec：调用过程
SYS_exec->do_execve
do_execve函数：

```
int
do_execve(const char *name, size_t len, unsigned char *binary, size_t size) {
    struct mm_struct *mm = current->mm;
    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
    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) {                goto execve_exit;
    }
    set_proc_name(current, local_name);
      return 0;
execve_exit:
   do_exit(ret);
   panic("already exit: %e.\n", ret);
}
```

主要工作：

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。
wait：调用过程
SYS_wait->do_wait
do_wait函数：

```
int
do_wait(int pid, int *code_store) {
   struct mm_struct *mm = current->mm;
   if (code_store != NULL) {
      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) {
      proc = find_proc(pid);
```

```
        if (proc != NULL && proc->parent == current) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found; //找到进程
            }
        }
    }
    else {
     //如果pid==0，则随意找一个处于退出状态的子进程
        proc = current->cptr;
        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;
        current->wait_state = WT_CHILD;
        schedule();
        if (current->flags & PF_EXITING) {
            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");
    }
    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;
}
```

主要工作：

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：调用过程：
SYS_exit->exit
do_exit函数:

```
int
do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }
    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);
/*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; //僵死状态
    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;
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
```

```
        }
    }
    local_intr_restore(intr_flag);
    schedule(); //选择新的进程执行
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

主要工作：

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。
思考题：
1 请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的?
①fork：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：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。
③execve：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。
④wait：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
2 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）。