

Lab5

马浩祎 卢艺晗 陆皓喆

练习0：填写已有实验

在已有的基础上，做一些修改：

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;
        proc->cptr = NULL;
        proc->optr = NULL;
        proc->yptr = NULL;
    }
    return proc;
}
```

在已有的函数上，添加 `wait_state`、`*cptr`、`*yptr`、`*optr` 的初始化操作，代码如下所示。

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; // 添加
```

```

assert(current->wait_state == 0);
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);
{
    int pid = get_pid();
    proc->pid = pid;
    hash_proc(proc);
    set_links(proc);
}
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;
}

```

在之前的基础上添加 `proc->parent = current`，将当前的进程设置为子进程的父进程。

`assert(current->wait_state == 0)` 确保了当前进程的等待状态为0。`set_links(proc)` 设置了进程间的关系。

练习1: 加载应用程序并执行（需要编码）

trapframe代码

问题

`do_exec` 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。你需要补充 `load_icode` 的第6步，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。

回答

代码如下所示：

```
tf->gpr.sp = USTACKTOP;
tf->epc = elf->e_entry;
tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
```

我们将 `sp` 寄存器设置为栈顶 `USTACKTOP`，然后将 `epc` 寄存器设置为文件的入口地址，将 `sstatus` 的 `SPP` 位清零，表示异常来自用户态；同时将 `sstatus` 的 `SPIE` 位清零，表示不启用中断。

执行过程

问题

请简要描述这个用户态进程被 `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。

回答

1. 首先，在 `init_main` 函数中，通过调用 `int pid = kernel_thread(user_main, NULL, 0)` 来调用 `do_fork` 函数，创建并唤醒进程，执行函数 `user_main`，此时线程状态已经变为 `PROC_RUNNABLE`，表明该线程开始运行；
2. 跳转到我们的 `user_main` 函数中，执行 `KERNEL_EXECVE(exit)`，相当于调用了 `kern_execve` 函数；
3. 在 `kernel_execve` 中执行到 `ebreak` 之后，发生断点异常，转到 `__alltraps`，转到 `trap`，再到 `trap_dispatch`，然后到 `exception_handler`，再到 `CAUSE_BREAKPOINT` 处，最后调用 `syscall` 函数

```
void
syscall(void) {
    struct trapframe *tf = current->tf;
    uint64_t arg[5];
    int num = tf->gpr.a0;
    if (num >= 0 && num < NUM_SYSCALLS) {
        if (syscalls[num] != NULL) {
            arg[0] = tf->gpr.a1;
            arg[1] = tf->gpr.a2;
            arg[2] = tf->gpr.a3;
            arg[3] = tf->gpr.a4;
            arg[4] = tf->gpr.a5;
            tf->gpr.a0 = syscalls[num](arg);
            return ;
        }
    }
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
        num, current->pid, current->name);
}
```

4. 在 `syscall` 中根据参数，确定执行 `sys_exec`，调用 `do_execve`

```
static int
sys_exec(uint64_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);
}
```

5. 在 `do_execve` 中调用 `load_icode`，加载文件

```
if ((ret = load_icode(binary, size)) != 0) {
    goto execve_exit;
}
```

6. 加载完毕后一路返回，直到 `__alltraps` 的末尾，接着执行 `__trapret` 后的内容，到 `sret`，表示退出S态，回到用户态执行，这时开始执行用户的应用程序

练习2: 父进程复制自己的内存空间给子进程（需要编码）

copy_range代码

问题

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。

回答

代码如下所示：

```
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));
    do {
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            struct Page *page = pte2page(*ptep);
            struct Page *npage = alloc_page();
            assert(page != NULL);
            assert(npage != NULL);
            int ret = 0;
            uintptr_t* src = page2kva(page); // 获取src源地址的内存虚拟地址
```

```
uintptr_t* dst = page2kva(npage); //获取dst目的地址的内核虚拟地址
memcpy(dst, src, PGSIZE); //拷贝内存，将src的内存复制到dst中
ret = page_insert(to, npage, start, perm); //最后将拷贝完的页插入到页表中即可

    assert(ret == 0);
}
start += PGSIZE;
} while (start != 0 && start < end);
return 0;
}
```

具体的实现：

1. 首先获取 `src` 源地址和 `dst` 目的地址的内核虚拟地址；
2. 拷贝内存，将 `src` 的内存复制到 `dst` 中；
3. 最后将拷贝完的页插入到页表中即可。

Copy on Write机制

问题

- 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。

回答

想要实现 `cow` 机制，其实只需要区分开读和写两个操作对页表的操作即可。如果只需要读的话，我们并不需要去修改页中的内容，只需要使用指针进行资源的访问即可。我们只需要在 `fork` 时，将父进程的所有页表项均设置为只读模式，然后在新进程中只复制栈和虚拟内存的页表，不去分配新的页；在子进程中执行时，如果子进程请求修改页中的内容，那么就会引发异常，此时我们再去分配空间的分配，将被访问的页表复制进去，再更新子线程的页表项即可。

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）

函数分析

问题

简要说明你对 `fork/exec/wait/exit` 函数的分析。

回答

fork函数

```
static int
sys_fork(uint64_t arg[]) {
    struct trapframe *tf = current->tf;
    uintptr_t stack = tf->gpr.sp;
    return do_fork(0, stack, tf);
}
```

sys_fork 函数调用 do_fork 函数。 do_fork 函数如下所示:

```
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; // 添加
    assert(current->wait_state == 0);
    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);
    {
        int pid = get_pid();
        proc->pid = pid;
        hash_proc(proc);
        set_links(proc);
    }
    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;
}

```

首先，为该进程的父进程赋值为当前的进程，`setup_kstack` 完成了内核栈空间的分配；`copy_mm` 完成了分配新的虚拟内存或与其他线程共享虚拟内存；`copy_thread` 获取了原线程的上下文与中断帧，并且设置了当前线程的上下文与中断帧；然后，为新进程获取新的 `pid` 进程号，并且赋值给该进程。然后将新线程插入哈希表和链表中，唤醒该新建的进程，返回该进程的 `pid` 号。

exec函数

```

static int
sys_exec(uint64_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);
}

```

`sys_exec` 函数调用了 `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);

    if (mm != NULL) {
        cputs("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);
}

```

该函数用于创建用户空间，加载用户程序。完成了当前线程的虚拟内存空间的回收，以及为当前线程分配新的虚拟内存空间，并加载了应用程序。

wait函数

```

static int
sys_wait(uint64_t arg[]) {
    int pid = (int)arg[0];
    int *store = (int *)arg[1];
    return do_wait(pid, store);
}

```

sys_wait 函数调用了 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;
    if (pid != 0) {
        proc = find_proc(pid);
        if (proc != NULL && proc->parent == current) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    else {
        proc = current->cptr;
        for (; proc != NULL; proc = proc->optr) {
            haskid = 1;
            if (proc->state == PROC_ZOMBIE) {
                goto found;
            }
        }
    }
    if (haskid) {
        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;

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);
        remove_links(proc);
    }
    local_intr_restore(intr_flag);
    put_kstack(proc);
    kfree(proc);
    return 0;
}

```

首先，查找状态为 PROC_ZOMBIE 的子线程；如果找到了，就将线程从哈希表和链表中删除，最后释放线程的资源。

如果查询到拥有子线程的线程，则设置线程状态并切换线程；如果线程已退出，则调用 `do_exit` 函数。

exit函数

```

static int
sys_exit(uint64_t arg[]) {
    int error_code = (int)arg[0];
    return do_exit(error_code);
}

```

`sys_exit` 函数调用了 `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) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {

```

```

        exit_mmap(mm);
        put_pgdir(mm);
        mm_destroy(mm);
    }
    current->mm = NULL;
}
current->state = PROC_ZOMBIE;
current->exit_code = error_code;
bool intr_flag;
struct proc_struct *proc;
local_intr_save(intr_flag);
{
    proc = current->parent;
    if (proc->wait_state == WT_CHILD) {
        wakeup_proc(proc);
    }
    while (current->cptr != NULL) {
        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();
panic("do_exit will not return!! %d.\n", current->pid);
}

```

具体代码逻辑：

- 如果当前线程的虚拟内存没有用于其他线程，则销毁该虚拟内存
- 如果用于其他的线程了，就将当前线程状态设为 `PROC_ZOMBIE`，唤醒该线程的父线程
- 完成 `exit` 后，调用 `schedule` 切换到其他线程

执行流程

问题

请分析 `fork/exec/wait/exit` 的执行流程。重点关注哪些操作是在用户态完成，哪些是在内核态完成？内核态与用户态程序是如何交错执行的？内核态执行结果是如何返回给用户程序的？

回答

- 系统调用的部分在内核态进行，用户程序的执行在用户态进行；
- 内核态通过系统调用结束后的 `sret` 指令来切换到用户态，用户态通过发起系统调用来产生 `ebreak` 异常，从而切换到内核态；
- 内核态执行的结果通过 `kernel_execve_ret` 函数将中断帧添加到线程的内核栈中，从而将结果返回给用户。

生命周期图

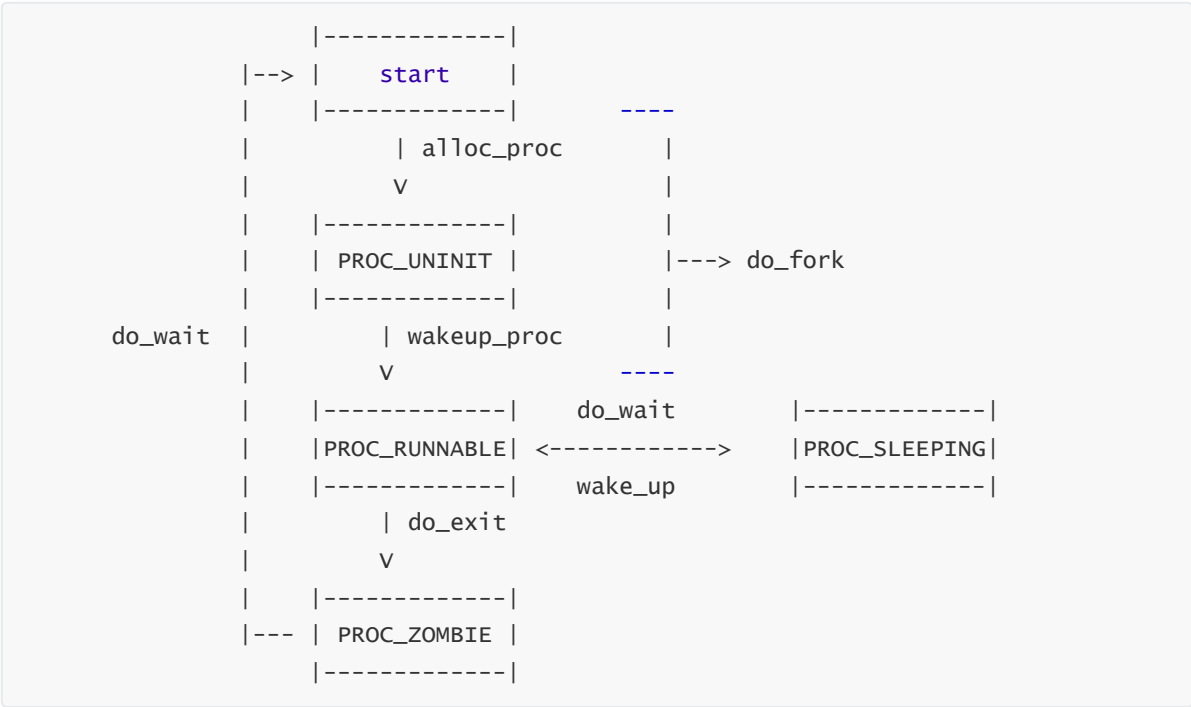
问题

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）

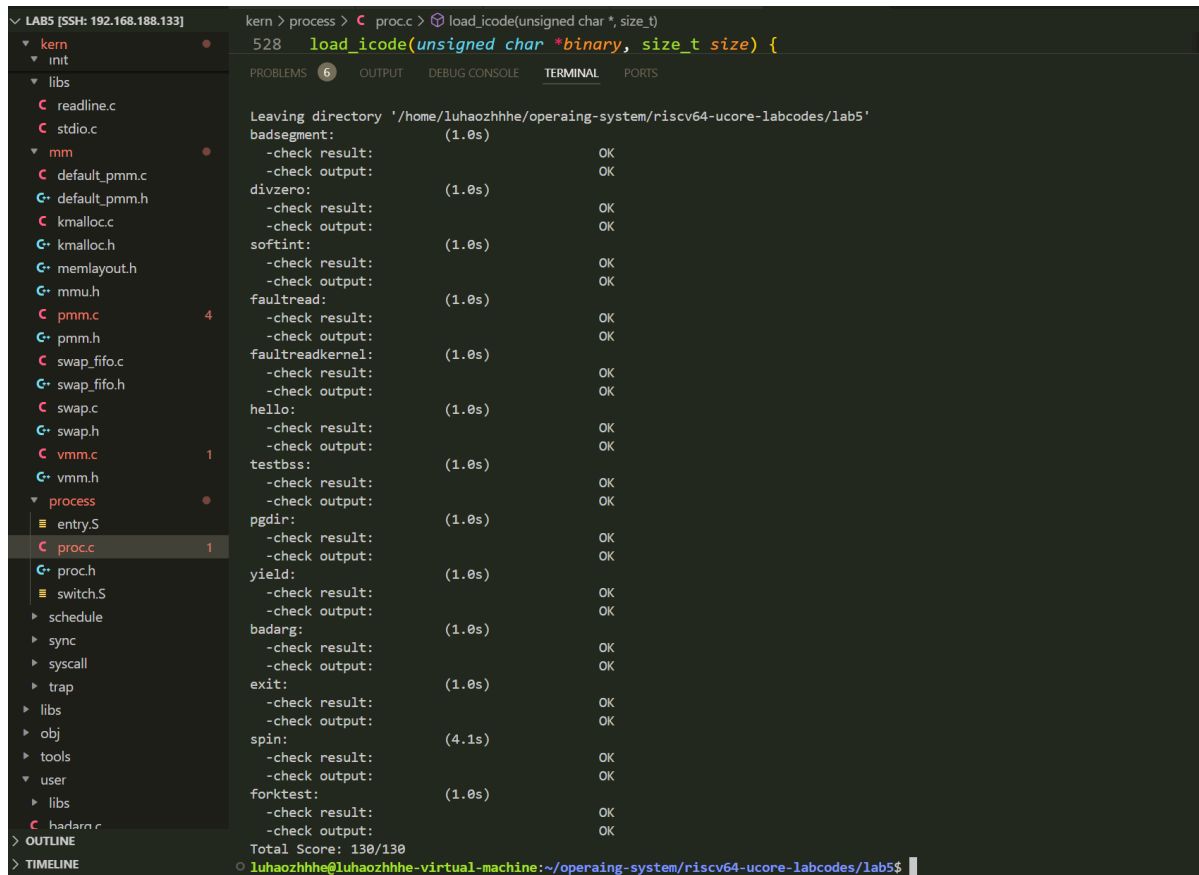
回答

生命周期图如下所示：

- 执行状态：start, PROC_UNINIT, PROC_RUNNABLE, PROC_ZOMBIE, PROC_SLEEPING等
- 函数调用：do_wait, alloc_proc, do_fork, wakeup_proc, wake_up, do_exit等



实验结果



```
kern > process > C proc.c > load_icode(unsigned char *, size_t)
528 load_icode(unsigned char *binary, size_t size) {
PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL PORTS
Leaving directory '/home/luhaozhhe/operating-system/riscv64-ucore-labcodes/lab5'
badsegment: (1.0s)
-check result: OK
-check output: OK
divzero: (1.0s)
-check result: OK
-check output: OK
softint: (1.0s)
-check result: OK
-check output: OK
faultread: (1.0s)
-check result: OK
-check output: OK
faultreadkernel: (1.0s)
-check result: OK
-check output: OK
hello: (1.0s)
-check result: OK
-check output: OK
testbss: (1.0s)
-check result: OK
-check output: OK
pgdir: (1.0s)
-check result: OK
-check output: OK
yield: (1.0s)
-check result: OK
-check output: OK
badarg: (1.0s)
-check result: OK
-check output: OK
exit: (1.0s)
-check result: OK
-check output: OK
spin: (4.1s)
-check result: OK
-check output: OK
forktest: (1.0s)
-check result: OK
-check output: OK
Total Score: 130/130
luhaozhhe@luhaozhhe-virtual-machine:~/operating-system/riscv64-ucore-labcodes/lab5$
```

make grade 得到 130/130 , 验证完毕!

扩展练习 Challenge

实现COW机制

我们修改了页面复制的函数copy_range:

```
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));
    do {
        pte_t *ptep = get_pte(from, start, 0), *nptep;
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {
            if ((nptep = get_pte(to, start, 1)) == NULL) {
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER);
            struct Page *page = pte2page(*ptep);
            int ret = 0;

            if(share)
```

```

    {
        // 物理页面共享，并设置两个PTE上的标志位为只读
        page_insert(from, page, start, perm & ~PTE_W);
        ret = page_insert(to, page, start, perm & ~PTE_W);
    }else{//原来的复制逻辑
        struct Page *npage = alloc_page();
        assert(page != NULL);
        assert(npage != NULL);
        uintptr_t* src = page2kva(page);
        uintptr_t* dst = page2kva(npage);
        memcpy(dst, src, PGSIZE);
        // 将目标页面地址设置到PTE中
        ret = page_insert(to, npage, start, perm);
    }
    assert(ret == 0);
}
start += PGSIZE;
} while (start != 0 && start < end);
return 0;
}

```

如果share的值为true，也就是启动共享机制，我们不需要像原来的代码那样进行页面的申请和复制，只需要通过调用 `page_insert` 函数，将源进程（from）中对应页面的权限重新设置为 `perm & ~PTE_W`，也就是在原权限基础上清除可写（`PTE_W`）标志位，使其变为只读，这样多个进程共享该物理页面时，就可以实现可读不可写的要求了。

然后，我们同样调用 `page_insert` 函数，将目标进程（to）中对应页面的权限也设置为 `perm & ~PTE_W`，达到两个进程共享同一个物理页面且都以只读方式访问的效果。只有当某个进程尝试对该共享页面进行写操作时（触发写时复制机制），才会去分配新的物理页面进行真正的复制操作，实现内存的高效利用以及数据的一致性保护。

如果share为false的时候，就说明我们不启用共享的机制，我们只需要执行原来的代码，完成页面的分配和内存的复制即可。

用户程序加载

（1）该用户程序的加载方式

在ucore的实现中，程序在`do_execve`函数中被显式加载到内存。调用`load_icode`函数，依次将ELF文件的文件头、程序头、程序段内容按页加载到内存空间中。在程序执行之前，所有需要的内容都已经加载到了内存中。

（2）常见操作系统（如Linux、Windows）的加载方式

传统操作系统广泛使用懒加载（Lazy Loading）机制，在`execve`中，只会加载文件头、程序头表，并分配内存区域，但此时并不会将程序段内容（代码段、数据段等）加载过来。

当程序开始执行时，以“按需加载”的方式，访问到某个虚拟地址时，若还没映射到物理地址，会通过缺页异常，从磁盘加载对应的页面内容，并更新页表。

(3) 比较

1. ucore的一次性加载简化了内存管理，更适合嵌入式系统和教学应用；
2. 懒加载的方式减少了初始内存占用；在执行过程中，仅加载执行路径中实际需要的段，减少了内存占用和磁盘 I/O。有助于提高整体性能。同时需要完善的页表和缺页异常处理机制来支持。