# lab5实验报告

## 练习零

- 在`alloc_proc`中添加额外的初始化：

```
proc->wait_state = 0;
proc->cptr = NULL; // Child Pointer 表示当前进程的子进程
proc->optr = NULL; // Older Sibling Pointer 表示当前进程的上一个兄弟进程
proc->yptr = NULL; // Younger Sibling Pointer 表示当前进程的下一个兄弟进程
```

- 在`do_fork`中修改代码如下：

```
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;
```

## 练习一

### 代码

将`sp`设置为栈顶，`epc`设置为文件的入口地址，`sstatus`的`SPP`位清零，代表异常来自用户态，之后需要返回用户态；`SPIE`位清零，表示不启用中断。

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

## 执行过程

1. 在 `init_main` 中通过 `kernel_thread` 调用 `do_fork` 创建并唤醒线程，使其执行函数 `user_main`，这时该线程状态已经为 `PROC_RUNNABLE`，表明该线程开始运行
2. 在 `user_main` 中通过宏 `KERNEL_EXECVE`，调用 `kernel_execve`
3. 在 `kernel_execve` 中执行 `ebreak`，发生断点异常，转到 `__alltraps`，转到 `trap`，再到 `trap_dispatch`，然后到 `exception_handler`，最后到 `CAUSE_BREAKPOINT` 处
4. 在 `CAUSE_BREAKPOINT` 处调用 `syscall`
5. 在 `syscall` 中根据参数，确定执行 `sys_exec`，调用 `do_execve`
6. 在 `do_execve` 中调用 `load_icode`，加载文件
7. 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

# 练习二

## 代码

首先获取源地址和目的地址对应的内核虚拟地址，然后拷贝内存，最后将拷贝完成的页插入到页表中。

```
uintptr_t* src = page2kva(page);
uintptr_t* dst = page2kva(npage);
memcpy(dst, src, PGSIZE);
ret = page_insert(to, npage, start, perm);
```

## COW设计

- 在 `fork` 时，将父线程的所有页表项设置为只读，在新线程的结构中只复制栈和虚拟内存的页表，不为其分配新的页
- 切换到子线程执行时，如果子线程需要修改一页的内容，会访问页表，由于该页不允许被修改，所以会引发异常
- 异常处理部分，遇到该类异常，重新分配一块空间，将访问的页面复制进去，更新子线程的页表项

# 练习三

## 函数分析

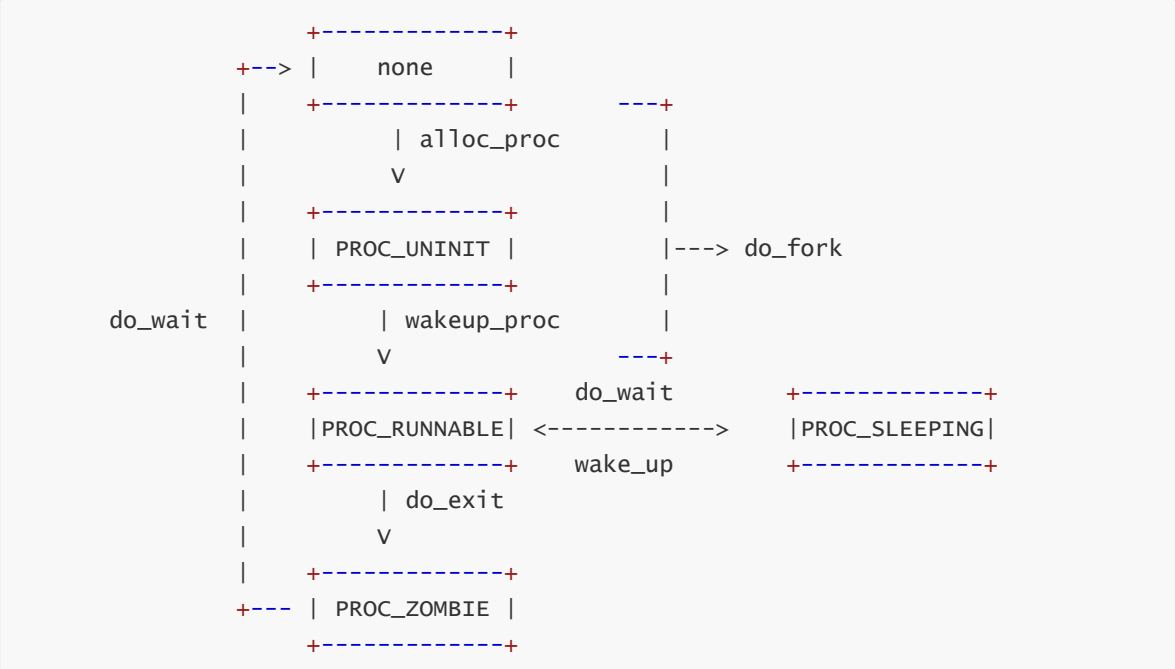
1. `fork`：通过发起系统调用执行 `do_fork` 函数。用于创建并唤醒线程，可以通过 `sys_fork` 或者 `kernel_thread` 调用。
    - 初始化一个新线程
    - 为新线程分配内核栈空间
    - 为新线程分配新的虚拟内存或与其他线程共享虚拟内存
    - 获取原线程的上下文与中断帧，设置当前线程的上下文与中断帧
    - 将新线程插入哈希表和链表中
    - 唤醒新线程
    - 返回线程 `id`

2. `exec`：通过发起系统调用执行`do_execve`函数。用于创建用户空间，加载用户程序，可以通过`sys_exec`调用。
    - 回收当前线程的虚拟内存空间
    - 为当前线程分配新的虚拟内存空间并加载应用程序
3. `wait`：通过发起系统调用执行`do_wait`函数。用于等待线程完成，可以通过`sys_wait`或者`init_main`调用。
    - 查找状态为`PROC_ZOMBIE`的子线程；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`
    - 将线程从哈希表和链表中删除
    - 释放线程资源
4. `exit`：通过发起系统调用执行`do_exit`函数。用于退出线程，可以通过`sys_exit`、`trap`、`do_execve`、`do_wait`调用。具体执行内容：
    - 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
    - 将当前线程状态设为`PROC_ZOMBIE`，唤醒该线程的父线程
    - 调用`schedule`切换到其他线程

## 执行流程

- 系统调用部分在内核态进行，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`指令切换到用户态，用户态通过发起系统调用产生`ebreak`异常切换到内核态
- 内核态执行的结果通过`kernel_execve_ret`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




# 扩展练习

COW的主体部分均放在`cow.c`文件中。

## COW实现

### 代码

创建失败时执行，这两项与`proc.c`中相同：

```
static int
setup_pgdir(struct mm_struct *mm) {
    struct Page *page;
    if ((page = alloc_page()) == NULL) {
        return -E_NO_MEM;
    }
    pde_t *pgdir = page2kva(page);
    memcpy(pgdir, boot_pgdir, PGSIZE);

    mm->pgdir = pgdir;
    return 0;
}

static void
put_pgdir(struct mm_struct *mm) {
    free_page(kva2page(mm->pgdir));
}
```

`do_pgfault`中添加判断页表项权限：

```
// 判断页表项权限，如果有效但是不可写，跳转到COW
if ((ptep = get_pte(mm->pgdir, addr, 0)) != NULL) {
    if((*ptep & PTE_V) & ~(*ptep & PTE_W)) {
        return cow_pgfault(mm, error_code, addr);
    }
}
```

将`do_fork`函数中的`copy_mm`改为`cow_copy_mm`：

```
// if(copy_mm(clone_flags, proc) != 0) {
//     goto bad_fork_cleanup_kstack;
// }
if(cow_copy_mm(proc) != 0) {
    goto bad_fork_cleanup_kstack;
}
```

复制虚拟内存空间：

```
int
cow_copy_mm(struct proc_struct *proc) {
    struct mm_struct *mm, *oldmm = current->mm;

    /* current is a kernel thread */
    if (oldmm == NULL) {
        return 0;
    }
    int ret = 0;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
```

```
    lock_mm(oldmm);
    {
        ret = cow_copy_mmap(mm, oldmm);
    }
    unlock_mm(oldmm);

    if (ret != 0) {
        goto bad_dup_cleanup_mmap;
    }

good_mm:
    mm_count_inc(mm);
    proc->mm = mm;
    proc->cr3 = PADDR(mm->pgdir);
    return 0;
bad_dup_cleanup_mmap:
    exit_mmap(mm);
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    return ret;
}
```

只复制`mm`与`vma`，将页表项均指向原来的页：

```
int
cow_copy_mmap(struct mm_struct *to, struct mm_struct *from) {
    assert(to != NULL && from != NULL);
    list_entry_t *list = &(from->mmap_list), *le = list;
    while ((le = list_prev(le)) != list) {
        struct vma_struct *vma, *nvma;
        vma = le2vma(le, list_link);
        nvma = vma_create(vma->vm_start, vma->vm_end, vma->vm_flags);
        if (nvma == NULL) {
            return -E_NO_MEM;
        }
        insert_vma_struct(to, nvma);
        if (cow_copy_range(to->pgdir, from->pgdir, vma->vm_start, vma->vm_end)
!= 0) {
            return -E_NO_MEM;
        }
    }
    return 0;
}
```

设置页表项指向：

```
int cow_copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0);
    assert(USER_ACCESS(start, end));
    do {
        pte_t *ptep = get_pte(from, start, 0);
        if (ptep == NULL) {
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
```

```
        }
        if (*ptep & PTE_V) {
            *ptep &= ~PTE_W;
            uint32_t perm = (*ptep & PTE_USER & ~PTE_W);
            struct Page *page = pte2page(*ptep);
            assert(page != NULL);
            int ret = 0;
            ret = page_insert(to, page, start, perm);
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

实现COW的缺页异常处理：

```
int
cow_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
    int ret = 0;
    pte_t *ptep = NULL;
    ptep = get_pte(mm->pgdir, addr, 0);
    uint32_t perm = (*ptep & PTE_USER) | PTE_W;
    struct Page *page = pte2page(*ptep);
    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);
    uintptr_t start = ROUNDDOWN(addr, PGSIZE);
    *ptep = 0;
    ret = page_insert(mm->pgdir, npage, start, perm);
    ptep = get_pte(mm->pgdir, addr, 0);
    return ret;
}
```

至此，`make qemu`与`make grade`均能正常运行并通过。

## 错误复现

在`user/exit.c`文件中添加

```
uintptr_t* p = 0x800588;
cprintf("*p = 0x%x\n", *p);
*p = 0x222;
cprintf("*p = 0x%x\n", *p);
```

如果使用原本的策略，执行`make qemu`会提示：

```
*p = 0x5171101
Store/AMO page fault
kernel panic at kern/fs/swapfs.c:20:
    invalid swap_entry_t = 2013281b.
```

但是如果使用COW策略，则会正常运行：

```
*p = 0x5171101
Store/AMO page fault
COW page fault at 0x800000
Store/AMO page fault
COW page fault at 0x7ffff000
*p = 0x222
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:479:
    initproc exit.
```

这表明我们在`qemu`模拟出的磁盘上的`0x800588`处写入了数据`0x222`。

## 用户程序加载

该用户程序在操作系统加载时一起加载到内存里。我们平时使用的程序在操作系统启动时还位于磁盘中，只有当我们需要运行该程序时才会被加载到内存里。

原因是在`Makefile`里执行了`ld`命令，把执行程序的代码连接在了内核代码的末尾。