练习0：填写已有实验

填写代码中需要更新代码的部分主要在两个函数中：

1. `alloc_proc` 函数中，需要添加对PCB中在LAB5中新增成员变量的初始化，涉及的成员变量为：`wait_state`、`cptr`、`yptr`、`optr`，具体代码如下：

```
// 把proc进行初步初始化（即把proc_struct中的各个成员变量清零）
proc->state = PROC_UNINIT;                                  // 设置进程状态为未初始化
proc->pid = -1;                                             // 设置进程ID为-1（还未分配）
proc->cr3 = boot_cr3;                                       // 设置为在uCore内核页表的起始地址
boot_cr3,内核线程共用一个映射内核空间的页表

// lab5 update
proc->state = 0;
proc->cptr = NULL;          // 初始化 cptr 为 NULL，表示没有子进程
proc->yptr = NULL;          // 初始化 yptr 为 NULL，表示没有“年轻”兄弟进程
proc->optr = NULL;          // 初始化 optr 为 NULL，表示没有“老”兄弟进程

// 所有内核线程的内核虚地址空间（也包括物理地址空间）是相同的。既然内核线程共用一个映射内核空间的页表，这表示内核空间对所有内核线程都是“可见”的，所以更精确地说，这些内核线程都应该是从属于同一个唯一的“大内核进程”—uCore内核。
proc->runs = 0;                                             // 设置进程运行次数为0
proc->kstack = 0;                                           // 设置内核栈地址为0（还未分配）
proc->need_resched = 0;                                     // 设置不需要重新调度
proc->parent = NULL;                                        // 设置父进程为空
proc->mm = NULL;                                            // 设置内存管理字段为空
memset(&(proc->context), 0, sizeof(struct context));        // 初始化上下文信息为0
proc->tf = NULL;                                            // 设置trapframe为空
proc->flags = 0;                                            // 设置进程标志为0
memset(proc->name, 0, PROC_NAME_LEN+1);                       // 初始化进程名为0
```

1. `do_fork` 函数中，需要添加设置当前进程的wait_state成员为0，而且还需要设置进程间的关系链接，其中设置进程间的关系链接利用到的函数是 `set_links`，具体代码如下：

```
// 1. 调用alloc_proc来分配一个proc_struct
if ((proc = alloc_proc()) == NULL)
    goto fork_out;

// 更新1：当前进程的wait_state是0
current->wait_state = 0;

// 2. 调用setup_kstack为子进程分配内核栈
proc->parent = current; // 设置子进程的父进程为当前进程
if (setup_kstack(proc))
    goto bad_fork_cleanup_kstack;
// 3. 调用copy_mm根据clone_flag来复制或共享内存,copy_mm函数目前只是把current->mm设置为NULL，这是由于目前在实验四中只能创建内核线程
// proc->mm描述的是进程用户态空间的情况，所以目前mm还用不上。
if (copy_mm(clone_flags, proc))
    goto bad_fork_cleanup_proc;
//前3步执行没有成功，则需要做对应的出错处理，把相关已经占有的内存释放掉。
```

```
// 4. 调用copy_thread来设置子进程的tf和context,在新创建的进程内核栈上专门给进程的中断帧分配一块空间。
copy_thread(proc, stack, tf);
// 5. 将新进程添加到进程列表和哈希表中
bool intr_flag;
local_intr_save(intr_flag); // 禁用中断,这些操作组合起来形成一个临界区，必须作为一个原子操作执行,确保数据结构（如哈希表和进程列表）的完整性和一致性
{
    proc->pid = get_pid();                    // 为子进程分配一个唯一的进程ID
    hash_proc(proc);                          // 将新进程添加到哈希表中
    //list_add(&proc_list, &(proc->list_link)); //这步已经在set_links函数里面实现了
    // 更新2：设置进程间的关系链接
    set_links(proc);
}
local_intr_restore(intr_flag); // 恢复中断
// 6. 调用wakeup_proc使新的子进程变为可运行状态
wakeup_proc(proc);
// 7. 使用子进程的pid作为返回值
ret = proc->pid;
```

练习一：加载应用程序并执行

在 `do_execve` 中，我们需要将用户态的程序加载到内核态，然后执行。

`load_icode` 函数主要流程：

1. 创建一个新的 `mm_struct`。
2. 创建一个新的 PDT，将 `mm` 的 `pgdir` 设置为这个 PDT 的虚拟地址。
3. 读取 ELF 格式，检验其合法性，循环读取每一个程序段，将需要加载的段加载到内存中，设置相应段的权限。之后初始化 BSS 段，将其清零。
4. 设置用户栈。
5. 设置当前进程的 `mm`，`cr3`，设置 `satp` 寄存器。
6. 设置 `trapframe`，将 `gpr.sp` 指向用户栈顶，将 `epc` 设置为 ELF 文件的入口地址，设置 `sstatus` 寄存器，将 `SSTATUS_SPP` 位置 0，表示退出当前中断后进入用户态，将 `SSTATUS_SPIE` 位置 1，表示退出当前中断后开启中断。

```
//(6) setup trapframe for user environment
struct trapframe *tf = current->tf;
// Keep sstatus
uintptr_t sstatus = tf->status;
memset(tf, 0, sizeof(struct trapframe));
tf->gpr.sp = USTACKTOP;
tf->epc = elf->e_entry;
// Set SPP to 0 so that we return to user mode
// Set SPIE to 1 so that we can handle interrupts
tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
```

用户态进程被选择到具体执行的详细过程

从 `proc_init` （创建了 `idle` 第0个内核线程然后调用 `kernel_thread` 创建了 `init` 第一个内核线程）然后到 `cpu_idle` 会不断的去查询 `need_resched` 标志位，调用 `schedule` 函数进行调度，在 `schedule` 找到第一个可以调度的线程后调用 `proc_run` 函数来切换到新进程， `proc_run` 调用了 `lcr3` 和 `swicth``_``to` 。然后返回到 `kernel_thread_entry` 去执行指定的函数，这里面就是去执行了 `initproc` 线程的主函数 `init_main` 。

上面这段与lab4一致，接下来有所区别，因为这里 `init_main` 函数不再是lab4中的只有几个 `print` 了。

在 `init_main` 中再次调用了 `kernel_thread` 去创建我们想去执行的程序，这里是 `user_main` 这个函数（ `user_main` 函数会在缺省的时候调用 `KERNEL_EXECVE(exit)` ， `KERNEL_EXECVE` 这个宏最后会去调用 `kernel_execve` 函数， `kernel_execve` 函数会通过内联汇编调用，使用 `ebreak` 指令来产生断点中断结合通过设置 `a7` 寄存器的值为10说明这不是一个普通的断点中断，让在trap.c文件中处理断点的时候去调用 `syscall` 函数和函数。后者在trapertry.S中定义；另一方面在 `kernel_execve` 函数的内联汇编中将 `sysexec` 宏放入了a0寄存器，让其执行一个系统调用，这个系统调用对应的函数就是 `sys_exec` 函数，其调用 `do_execve` 函数， `do_execve` 函数中调用 `load_icode` 函数来加载新的二进制程序到内存中，这个函数也设置了中断帧的一些内容，SPP设置为0，让其可以后面直接返回到用户态。）创建完 `user_main` 的那个线程后不会立刻调用执行，而且执行 `init_main` 函数后面的逻辑，这里会调用 `do_wait` 函数，来释放。对于当前进程来说，其实只有一个 `usermain` 是他子进程，而且这个子进程的状态也不是僵尸进程，所以会将当前进程状态设置为sleeping，等待状态设为 `WT_CHILD` ，并调用schedule函数进行调度，此时调度就会选中 `user_main` 进行执行。首先会按照上文括号中的过程加载这个二进制程序，会把exit应用程序执行码覆盖到 `usermain` 的用户虚拟内存空间来创建的，然后去执行 `kernel_execve_ret` 函数最后通过sret退出到用户态，然后开始执行 `initcode.S` 中代码，然后开始执行 `umian` 函数，其中会先执行用户态进程 `exit.c` 文件中的主体函数，然后执行 `exit` 函数进行退出。在 `exit` 的主体 `mian` 函数中会先调用 `fork` 函数fork出来一个子进程，然后在 `wait` 函数中进行了进程切换然后重新执行一遍，后续就是父进程将fork出来的子进程回收。然后父进程自己也进行了退出一直回到 `initproc` 这个内核线程中，然后通过 `do_exit` 函数的一个panic来结束这次实验。

练习二：父进程复制自己的内存空间给子进程

在 `copy_range` 中实现了将父进程的内存空间复制给子进程的功能。逐个内存页进行复制，首先找到父进程的页表项，然后创建一个子进程新的页表项，设置对应的权限，然后将父进程的页表项对应的内存页复制到子进程的页表项对应的内存页中，然后将子进程的页表项加入到子进程的页表中。

```
void *src_kvaddr = page2kva(page); // 父进程的内存页的 kernel addr
void *dst_kvaddr = page2kva(npage); // 子进程的内存页的 kernel addr
memcpy(dst_kvaddr, src_kvaddr, PGSIZE); // 复制内存页
ret = page_insert(to, npage, start, perm); // 将子进程的页表项加入到子进程的页表中
```

如何实现 Copy on Write 机制

要实现 Copy on Write 机制，可以在复制父进程的内存空间给子进程时，不复制整个内存页，而是只复制页表项，然后将父进程和子进程的页表项的权限都设置为只读。这样两个进程都可以访问同一个内存页，当其中一个进程要写入时，会触发缺页异常，然后在缺页异常处理函数中，复制整个内存页，设置可写入权限，这时就可以写入了。当某个共享页面只剩下一个进程时，就可以将其权限设置为可写。

更加详细的实现在后文Challenge1中实现。

练习三：理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

fork

用户态：

`fork()` -> `sys_fork()` -> `syscall(SYS_fork)` -> `ecall` -> 内核态

内核态：

```
syscall()` → `sys_fork()` → `do_fork(0, stack, tf)
```

`do_fork` 中，调用 `alloc_proc` 分配一个 `proc_struct`，并设置父进程。调用 `setup_kstack` 为子进程分配一个内核栈，调用 `copy_mm` 根据 `clone_flag` 复制或共享 `mm`，调用 `copy_thread` 在 `proc_struct` 中设置 `tf` 和上下文，将 `proc_struct` 插入 `hash_list` 和 `proc_list`，调用 `wakeup_proc` 使新的子进程变为可运行状态，使用子进程的 pid 设置返回值

exec

内核态：

```
kernel_execve()` → `ebreak` → `syscall()` → `sys_exec()` → `do_execve()
```

检查用户提供的程序名称是否合法。

如果当前进程的 mm 不为空，说明当前进程占用了内存，进行相关清理操作，包括切换到内核页表、释放进程的内存映射、释放页目录表、销毁进程的内存管理结构等。

调用 `load_icode` 函数加载用户提供的二进制文件，将其代码段加载到内存中。

使用 `set_proc_name` 函数设置进程的名称。

wait

用户态：

`wait()` -> `sys_wait()` -> `syscall(SYS_wait)` -> `ecall` -> 内核态

内核态：

```
syscall()` → `sys_wait()` → `do_wait()
```

首先进行内存检查，确保 `code_store` 指向的内存区域可访问。遍历查找具有给定PID的子进程，若找到且该子进程的父进程是当前进程，将 haskid 标志设置为1。如果 pid 为零，将循环遍历所有子进程，查找已经退出的子进程。如果找到，跳转到标签 found。如果存在子进程，将当前进程的状态设置为 `PROC_SLEEPING`，等待状态设置为 `WT_CHILD`，然后调用调度器 `schedule()` 来选择新的可运行进程。如果当前进程被标记为 `PF_EXITING`，则调用 `do_exit` 以处理退出,跳转到标签 repeat 继续执行。

找到后检查子进程是否是空闲进程 `idleproc` 或初始化进程 `initproc`，如果是则触发 panic。存储子进程的退出状态，处理子进程退出并释放资源。

exit

用户态：

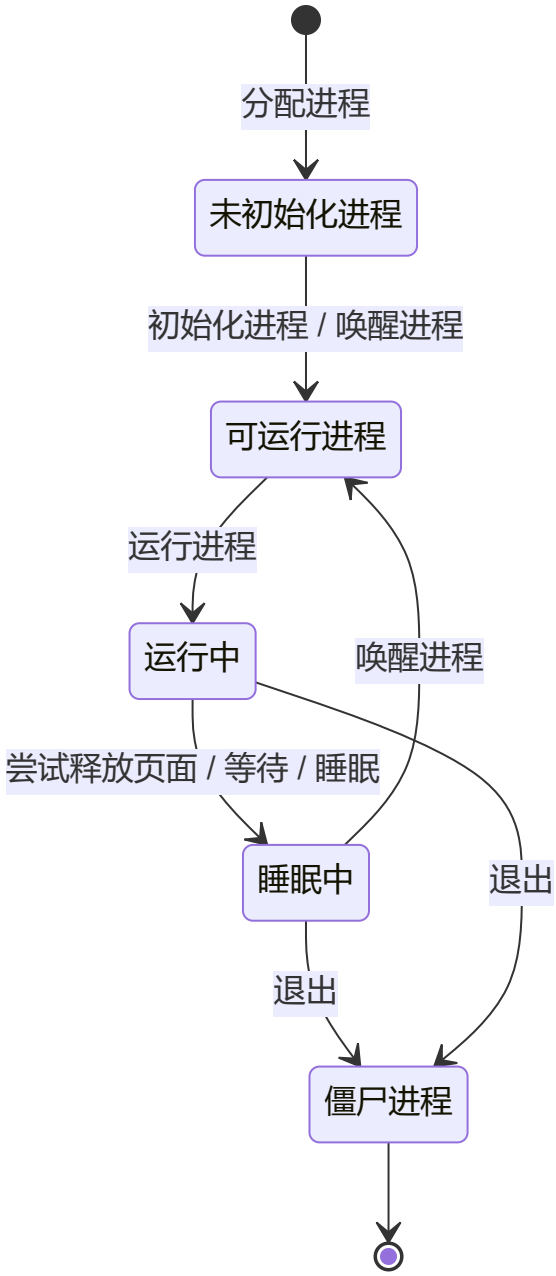
`exit()` -> `sys_exit()` -> `syscall(SYS_exit)` -> `ecall` -> 内核态

内核态：

```
syscall()` → `sys_exit()` → `do_exit()
```

检查当前进程是否是idleproc或initproc，若是则 `panic` 。获取内存管理结构，减少对内存管理结构的引用计数，如果引用计数降为零，代表没有其他进程共享该内存管理结构，那么清理映射并释放页目录表，最后销毁内存管理结构。最后，将当前进程的 `mm` 指针设置为 `NULL` 。将进程的状态设置为 `PROC_ZOMBIE` ，表示进程已经退出。如果父进程正在等待子进程退出，则唤醒当前进程的父进程。然后，通过循环处理当前进程的所有子进程，将它们的状态设置为 `PROC_ZOMBIE` ，并将其重新连接到初始化进程的子进程链表上。如果初始化进程也正在等待子进程退出，那么也唤醒初始化进程。最后，进行调度。

给出ucore中一个用户态进程的执行状态生命周期图



扩展练习 Challenge1：实现 Copy on Write 机制

实现源码

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) {
                ...
        /* 实现cow1：设置共享标志
         * 将dup_mmap中的share变量的值改为1，启用共享
         */
        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_V) {
            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);
            assert(page≠NULL);
            int ret=0;
            if (share) {
                 /* 实现cow2：映射共享页面
                  * 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。
                  */
                // share page
                page_insert(from, page, start, perm & (~PTE_W));
                ret = page_insert(to, page, start, perm & (~PTE_W));
            } else {
                // alloc a page for process B
                struct Page *npage=alloc_page();
                assert(npage≠NULL);
                uintptr_t src_kvaddr = page2kva(page);
                uintptr_t dst_kvaddr = page2kva(npage);
                memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
                ret = page_insert(to, npage, start, perm);
            }
            assert(ret == 0);
        }
                ...
```

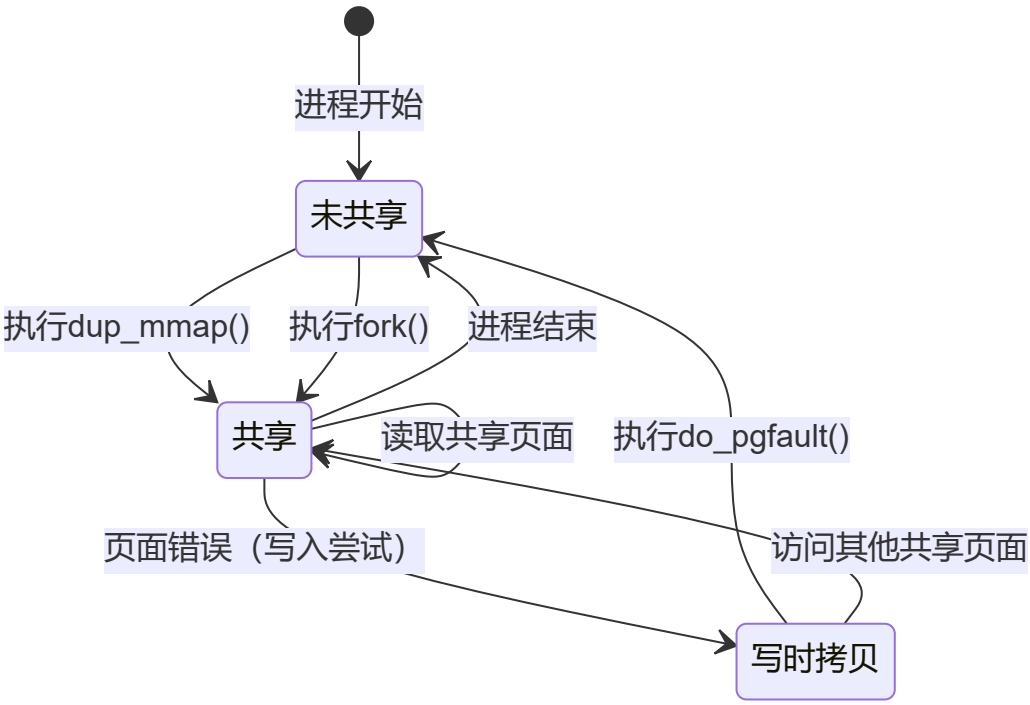
```
    return 0;
}
```

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。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：

```
int do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
        ...
    if (*ptep == 0) {
        ...
    } else if ((*ptep & PTE_V) && (error_code & 3 == 3)) {
        /* 实现cow3：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。
         * 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。
         */
        // copy on write
        struct Page *page = pte2page(*ptep);
        struct Page *npage = pgdir_alloc_page(mm→pgdir, addr, perm);
        uintptr_t src_kvaddr = page2kva(page);
        uintptr_t dst_kvaddr = page2kva(npage);
        memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
    } else {
                ...
           }
        ...
}
```

在cow情况下各种状态转换的时序图



初始状态为未共享状态，此时进程拥有自己的独立内存空间。此阶段没有激活共享或写时拷贝机制。

通过`dup_mmap()`转换到共享状态，当调用 fork 操作时，会调用`dup_mmap`函数，将`share`变量设为`1`。这表示父进程和子进程之间的内存页面进行了共享。但是，这些页面被设置为只读，以防止意外修改。

在共享状态下，父子进程共享相同的物理页面，这些页面对两个进程都是只读的。

如果父进程或子进程尝试写入共享页面，会发生页面错误（因为页面是只读的）。此时的错误代码低两位被设置为`1`，表明尝试对只读页面进行写入。此时，内核需要用`do_pgfault()`为试图写入的进程重新分配一个新页面，并将原页面的内容拷贝到新页面上。然后，更新页面表，使得试图写入的进程的页面映射指向这个新分配的页面。这样，原页面仍然与另一进程共享，而当前进程则有了一个可写的独立副本。

写时拷贝完成后，进程返回到未共享状态，拥有自己的独立、可写的内存空间。

执行结果

执行Make Grade，可以看见Copy On Write正确执行：

```
labcodes/lab5'
badsegment:              (s)
  -check result:                             OK
  -check output:                             OK
divzero:                 (s)
  -check result:                             OK
  -check output:                             OK
softint:                 (s)
  -check result:                             OK
  -check output:                             OK
faultread:               (s)
  -check result:                             OK
  -check output:                             OK
faultreadkernel:         (s)
  -check result:                             OK
  -check output:                             OK
hello:                   (s)
  -check result:                             OK
  -check output:                             OK
testbss:                 (s)
  -check result:                             OK
  -check output:                             OK
pgdir:                   (s)
  -check result:                             OK
  -check output:                             OK
yield:                   (s)
  -check result:                             OK
  -check output:                             OK
badarg:                  (s)
  -check result:                             OK
  -check output:                             OK
exit:                    (s)
  -check result:                             OK
  -check output:                             OK
spin:                    (s)
  -check result:                             OK
  -check output:                             OK
forktest:                (s)
  -check result:                             OK
  -check output:                             OK
Total Score: 130/130
```

扩展练习 Challenge2：说明该用户程序是何时被预先加载到内存中的？与我们常用操作系统的加载有何区别，原因是什么？

在本次实验中，用户程序在编译时被链接到内核中，并定义好了起始位置和大小，然后在 `user_main()` 函数 `KERNEL_EXECVE` 宏调用 `kernel_execve()` 函数，从而调用 `load_icode()` 函数将用户程序加载到内存中。实现了通过一个内核进程直接将整段文件直接加载内存中。

而在我们常用的操作系统中，用户程序通常是存储在外部存储设备上的独立文件。当需要执行某个程序时，操作系统会从磁盘等存储介质上动态地加载这个程序到内存中。

这里我们之所以采用这种加载方式的原因是 ucore 没实现硬盘和文件系统，出于简化和教学性质的考虑，将用户程序编译到内核中减少了实现的复杂度。