实验 5 报告文档

12/5-12/11 LAB5

练习 0：填写已有实验

本实验依赖实验 2/3/4。请把你做的实验 2/3/4 的代码填入本实验中代码中有“LAB2”/“LAB3”/“LAB4”的注释相应部分。注意：为了能够正确执行 lab5 的测试应用程序，可能需对已完成的实验 2/3/4 的代码进行进一步改进。

解答：

在 Lab5 当中，需要将我们在 Lab4 的 proc.c 当中实现的 `alloc_proc()、do_fork()` 函数的代码移植过来，并加以修改。

alloc_proc()函数的修改:

由于 Lab5 当中 `proc_struct` 结构又增添了 `wait_state , cptr, optr, yptr` 这四个成员变量，因此在 `alloc_proc()`函数当中也需要对他们进行初始化，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。这部分代码的修改如下：

```
Plain Text
// Lab4 Code ...

// Lab5 Update Code
proc->wait_state = 0;  //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
```

do_fork()函数的修改:

根据项目注释的提示，我们需要分别将 do_fork()函数的第 1 步和第 5 步加以修改。对于第 1 步，我们要将子进程的父节点设置为当前进程，还要确保当前进程的成员变量 wait_state 值为 0。这部分代码的修改如下：

```
Plain Text
proc->parent = current;
// update step 1:确保当前进程的 wait_state 是 0
assert(current->wait_state == 0);
```

对于第 5 步,除了将进程块插入到哈希链表和进程链表外，还要设置进程间的链接关系，这个功能通过调用 set_links()函数实现.接下来观察 set_links()函数的代码：

```
Plain Text
static void
set_links(struct proc_struct *proc) {
    list_add(&proc_list, &(proc->list_link));
    proc->yptr = NULL;
    if ((proc->optr = proc->parent->cptr) != NULL) {
        proc->optr->yptr = proc;
    }
    proc->parent->cptr = proc;
    nr_process ++;
}
```

可以看到，set_links()函数内部已经实现了将进程 proc 插入到进程链表和进程数加 1 的功能，故在 do_fork()函数当中无需再执行相应功能，这部分代码的修改如下：

```
Plain Text
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

    // update step 5: 删除原来的 nr_process++ 和 list_add(...)
    // 这是因为 set_links 函数内已经实现了这两个功能
    set_links(proc);   // 实现设置相关进程链接
}
local_intr_restore(intr_flag);//恢复中断
```

练习 1: 加载应用程序并执行（需要编码）

do_execve 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。

请在实验报告中简要说明你的设计实现过程。

- *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整个经过。*

解答：

编程设计与解析：

首先我们了解一下 do_execve 和 load_icode 这两个函数的功能：

do_execve 函数的主要功能和代码实现：

观察以下 do_execve 函数的代码:

```
Plain Text
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);
}
```

可以得知 do_execve 的工作流程，以及相应实现的功能如下:

- 首先调用 user_mem_check 函数来检查 name 的内存空间能否被用户态程序访问，

如果不行，直接 return -E_INVAL 返回错误。

- 然后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(mm 初始被赋值为 current->mm)，则通过调用函数 lcr3(boot_cr3)，将 cr3 页表基址指向 boot_cr3，从而设置页表为内核空间页表。
- 接下来进一步判断 mm 的引用计数减 1(通过 mm_count_dec(mm)函数实现)

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，通过调用 exit_mmap(mm) 和 put_pgdir(mm)这两个函数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 current->mm 置为 NULL。

通过以上流程，我们已经为 load_icode 做好了用户态内存空间清理的工作。之后 do_execve 就会调用 load_icode 来完成读 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等一系列复杂的操作，具体操作将会在接下来介绍。

load_icode 函数的主要功能：

由于 load_icode 函数的代码很长，考虑篇幅就不详细分析代码，而是概括 load_icode 的工作流程以及相应实现的功能。

- 调用 mm_create 函数来为当前进程的内存管理数据结构 mm 申请所需的内存空间，并对 mm 进行初始化；
- 调用 setup_pgdir 来申请一个页大小的内存空间，用来存储页目录表，并把描述 ucore 内核虚空间映射的内核页表（即 boot_pgdir 所指向的）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拟地址；
- 根据应用程序执行码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 格式的执行程序说明的各个段（TEXT 段、DATA 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，从而表明了用户进程的合法用户态虚拟地址空间；
- 调用 pgdir_alloc_page 函数，根据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；
- 给用户进程设置用户栈，为此调用 mm_map 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，并同样通过调用 pgdir_alloc_page 函数分配一定数量的物理内存，同时建立好栈的虚拟地址与物理地

址之间的映射关系(这通过 pgdir_alloc_page 函数内部调用的 page_insert 函数实现);

- 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;
- 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;

基于以上的分析，我们可以补充 load_icode 函数的第 6 步，为了让程序从中断返回时能正确进入用户态，需要设置好设置中断帧的相关参数，详细的设计思路如下所示:

首先，我们需要令中断帧 tf 的栈指针 sp 指向用户栈的栈顶，因此将 USTACKTOP 赋值给 tf->gpr.sp

然后，我们需要令 tf 的 epc 指向用户程序的入口点，便于接下来执行用户程序，而 tf 的成员变量 e_entry 的定义如下:

```
Plain Text
uint64_t e_entry;     // entry point if executable
```

因此设置 tf 的 epc 值为 elf->e_entry。

最后，设置好 tf 的特殊寄存器 sstatus 的值，需要正确设置其 SPP 位和 SPIE 位的值。

查看 SPP 位所表示的含义如下:

```
Plain Text
When a trap is taken, SPP is set to 0 if the trap originated from
user mode, or 1 otherwise.
When an SRET instruction is executed to return from the trap
handler,
the privilege level is set to user mode if the SPP bit is 0,
or supervisor mode if the SPP bit is 1; SPP is then set to 0.
```

概括下来就是，如果是用户态的程序，SPP 位需要置为 0，其它情况均设为 1.据此可以得知，在 load_icode 函数当中我们需要将 SPP 位设为 0.

查看 SPIE 位所表示的含义如下:

```
Plain Text
The SPIE bit indicates whether supervisor interrupts were enabled
prior to trapping into supervisor mode.
When a trap is taken into supervisor mode, SPIE is set to SIE, and
```

```
SIE is set to 0.
When an SRET instruction is executed, SIE is set to SPIE, then
SPIE is set to 1.
```

可见，当用户态程序发生中断时，SPIE 位需要存储当时 SIE 的值，这与它之前是什么值无关；而当程序中断处理完毕，返回用户态程序时，SPIE 位需置为 1。据此可知，我们应当设置 tf->sstatus 的 SPIE 位为 1.

最终，经过以上所有的分析，我们可以得到以下补充的 load_icode 函数的第 6 步的代码，如下所示：

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

问题 1.1 回答：

1. 调用 schedule 函数，寻找就绪态的用户态进程，然后再调用 proc_run 函数使其准备运行。之后触发中断从而转入到了系统调用的处理例程；

2. 之后进行正常的中断处理例程，在 trap()函数当中调用 trap_dispatch 函数，其中再调用 exception_handler 转到 syscall()函数，

3. syscall()函数根据系统调用号决定调用 sys_exec 函数，而在 sys_exec 函数中调用了 do_execve 函数来完成指定应用程序的加载；

4. 具体在 do_execve 中，进行了若干设置，包括退出当前进程的页表，换用内核的 PDT 等待。

5. 然后调用 load_icode 函数完成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；

6. 在完成了 do_execve 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 sp 已经被修改成了应用程序的入口处，因此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；

7. 开始执行应用程序的第一条代码；

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你的设计实现过程。

- 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。

解答：

编程设计与解析：

首先叙述一下 do_fock 函数，它是一个内核函数，用于父进程对子进程的复制。工作流程大致如下：

- do_fork 将创建进程控制块，之后分配 kernel stack，包括分配 memory 以及虚地址。
- 之后，do_fork 会调用 copy_mm 函数拷贝当前进程（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。
- 接下来，do_fork 会通过 copy_thread 函数设置 trapframe 和 context 的相关参数，之后便将新创建好的子进程放到进程队列中去，设置好链接关系（通过 set_links 函数实现），唤醒子进程，便可以等待执行。

其中，复制父进程内存的部分会调用 copy_mm()函数，而 copy_mm()再下一层调用 dup_mmap 函数完成新进程中的·。然后 dup_mmap 函数调用了 copy_range 函数。

接下来，使用 copy_range 函数在内存页级别上复制一段地址范围的内容。这个函数首先通过 get_pte 函数获取源页表中的页表项，然后在目标页表中获取或创建新的页表项，并为新的页分配内存。

最后，copy_range 确保源页和新页都成功分配，并准备调用进行最底层的复制操作，也就是我们需要完成的部分。这个过程主要有以下四步：

1. 通过刚开始获取的 page 即源 page 通过宏 page2kva 转换为源虚拟内存地址。
2. 按照同样的方法，将要复制过去的 n 个 page 转换为目的虚拟内存地址。
3. 通过 memcpy 将虚拟地址进行复制，复制其内容。
4. 最后调用 page_insert 函数建立 npage 相应的虚拟地址与物理地址之间的映射关

系

综合以上的分析，我们可以给出代码设计,详细的解析在注释当中：

```
Plain Text
// find src_kvaddr: the kernel virtual address of page
void* kva_src = page2kva(page);
// find dst_kvaddr: the kernel virtual address of npage
void* kva_dst = page2kva(npage);
//  Copy the content of the source page to the destination page
memcpy(kva_dst,kva_src,PGSIZE);
// build the map of phy addr of npage with the linear addr start
ret = page_insert(to,npage,start,perm);
```

问题 2.1 回答：

大致的思路是，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。接下来介绍详细一些的设计思路。

首先，Copy-on-write（简称 COW）的核心是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，这样的话我们在开始应该只需要分配一个相应的指针位，经过判断就可以进行读操作。

在实际中，发现实际上 dup_mmap 函数中有对 share 的设置，因此首先需要将 share 设为 1, 就表示可以共享。

接着，为了实现相应操作，应该在调用 copy_range 函数之前就 share 变量的设置进行修改以便后面识别。由于 dup_mmap 函数中会调用 copy_range 函数，所以我们在 copy_range 之前定义参数 share 为 1，保证 copy_range 读取的时候就意识到已经启动了共享机制。

完成这里的话，已经实现了读的共享，但是并没有对写做处理，因此需要对由于写了只能读的页面导致的页错误进行处理：当程序尝试修改只读的内存页面的时候，将触发 Page Fault 中断，这时候我们可以检测出是超出权限访问导致的中断，说明进程访问了共享的页面且要进行修改，因此内核此时需要重新为进程分配页面、拷贝页面内容、建立映射关系。这些在页面初始化函数中可以实现。

我们在本实验当中已经实现了 COW 机制，故详细的代码设计放到 Challenge 部分再介绍。

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 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？
- 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）

解答：

1. fork: 完成进程的拷贝，由 do_fork 函数完成。

- 调用过程:

用户态： `fork()` -> `sys_fork()` -> `do_fork()` -> `syscall(SYS_fork)` -> `ecall`

内核态： `syscall()` -> `sys_fork()` -> `do_fork(0, stack, tf)`

- 整体流程:

在 `do_fork` 函数中，首先通过 `alloc_proc` 分配一个新的 `proc_struct` 结构体来创建子进程，并将当前进程设置为其父进程。接着，使用 `setup_kstack` 为子进程分配一个独立的内核栈，然后根据 `clone_flag` 通过 `copy_mm` 函数复制或共享父进程的内存管理结构。`copy_thread` 被调用来在新进程的 `proc_struct` 中设置中断帧和上下文信息。之后，新进程被插入到进程哈希列表和进程链表中，通过 `wakeup_proc` 将其设置为可运行状态。最后，`do_fork` 返回新创建的子进程的进程标识符（PID）。

2. exec: 完成用户进程的创建并进入执行状态，由 do_execve 函数完成。

- 调用过程:

内核态： `kernel_execve()` -> `ebreak` -> `syscall()` -> `sys_exec()` -> `do_execve()`

- 整体流程:

在处理用户提供的程序时，代码首先进行合法性检查以确保程序名称是有效的。如果当前进程已有内存分配（即其内存管理结构 `mm` 非空），则执行清理操作，包括切换回内核页表，释放进程的内存映射和页目录表，并最终销毁进程的内存管理结构。随后，

通过 `load_icode` 函数加载用户的二进制文件，并将代码段加载到内存中。最后，使用 `set_proc_name` 函数为进程设置新的名称。

3. wait: 对子进程的内核栈和进程控制块所占内存空间进行回收，由 do_wait 函数完成。

- 调用过程:

用户态： `wait()` -> `sys_wait()` -> `syscall(SYS_wait)` -> `ecall`

内核态： `syscall()` -> `sys_wait()` -> `do_wait()`

- 整体流程:

在处理子进程退出时，首先对 `code_store` 指向的内存区域进行检查，确保其可访问。接着，函数遍历查找具有指定 PID 的子进程，如果该子进程的父进程是当前进程，设置 `haskid` 标志为 1。当 pid 为零时，会遍历所有子进程，寻找已经退出的子进程。一旦找到符合条件的子进程，程序跳转到 `found` 标签。如果存在子进程，当前进程的状态将设置为 `PROC_SLEEPING`，等待状态设置为 `WT_CHILD`，然后调用 `schedule()` 函数进行调度，以便系统选择另一个可运行的进程。如果当前进程被标记为 `PF_EXITING`，则调用 `do_exit` 函数来处理其退出，并跳转回 `repeat` 标签继续查找。在找到相应的子进程后，会检查这个子进程是否是特殊的系统进程，如空闲进程 `idleproc` 或初始化进程 `initproc`，如果是，则触发 `panic` 错误。最后，存储子进程的退出状态，并处理子进程的退出，以便释放其占用的资源。

4. exit: 完成当前进程执行退出过程中的部分资源回收，由 do_exit 函数完成。

- 调用过程:

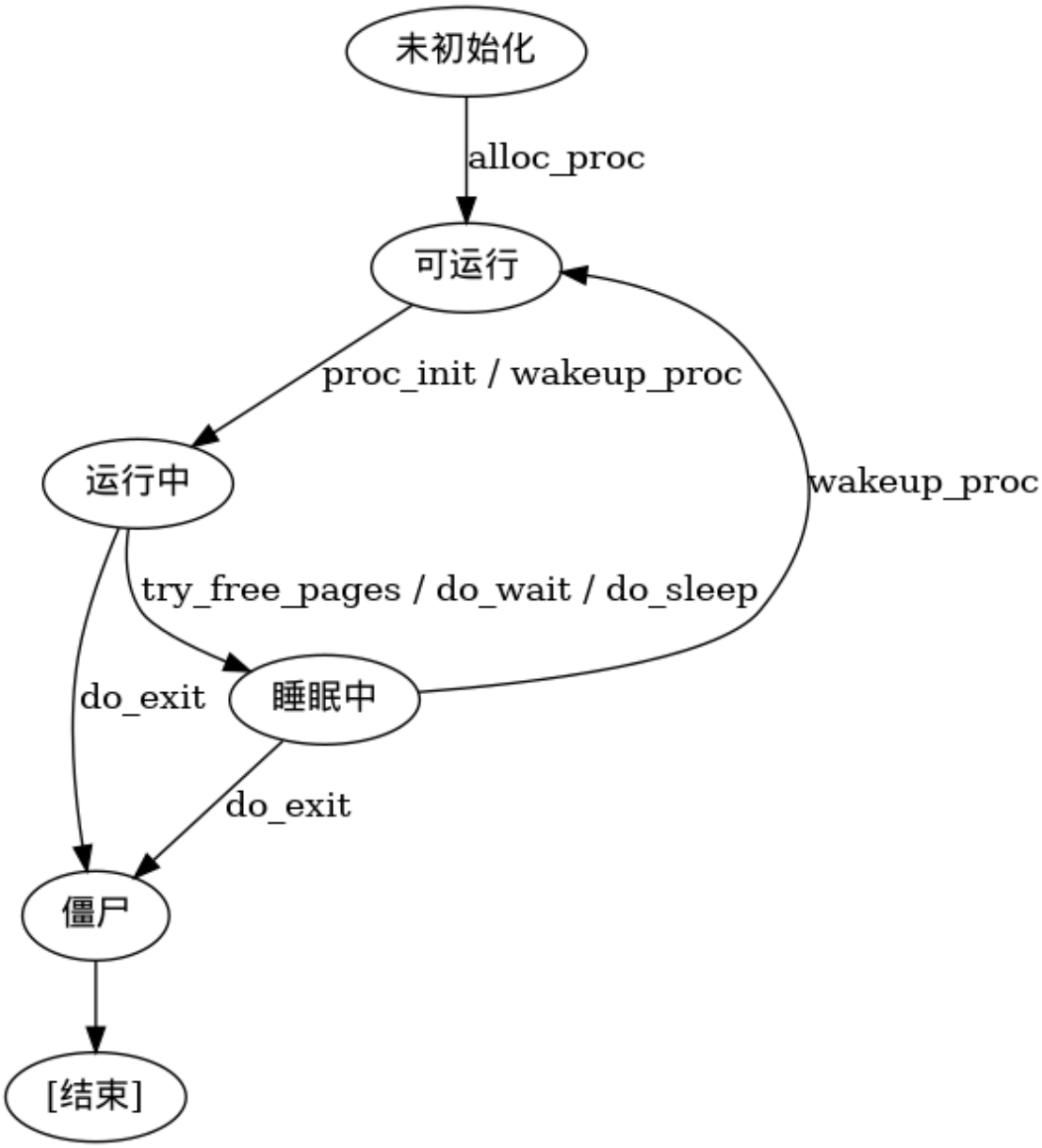
用户态： `exit()` -> `sys_exit()` -> `syscall(SYS_exit)` -> `ecall`

内核态： `syscall()` -> `sys_exit()` -> `do_exit()`

- 整体流程

当一个进程准备退出时，首先检查该进程是否是系统的空闲进程（`idleproc`）或初始化进程（`initproc`），如果是，则触发紧急错误（`panic`）。接着，函数获取当前进程的内存管理结构（`mm`），并减少其引用计数。如果引用计数降至零，表明没有其他进程共享这个内存管理结构，因此进行内存映射的清理、释放页目录表，并销毁内存管理结构。此后，将当前进程的 `mm` 指针设为 `NULL`。进程的状态被设置为 `PROC_ZOMBIE`，表示它已经执行完毕但尚未完全清理。如果当前进程的父进程正在等待其退出，那么父进程会被唤醒。随后，循环处理该进程的所有子进程，将它们的父进程改为初始化进程，并在必要时唤醒初始化进程，以便它可以回收这些子进程。最后，执行调度操作，让系统选择另一个进程来运行。

状态生命周期图如下:



扩展练习 Challenge

1. 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。

- 这是一个 big challenge.

2. 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？

Challenge 1 回答：

当进行内存访问时，CPU 会根据 PTE 上的读写位 PTE_V、PTE_W 来确定当前内存操作是否允许，如果不允许，则触发 Page Fault 中断。我们可以在 copy_range 函数中，将父进程中所有 PTE 中的 PTE_W 置为 0，这样便可以将父进程中所有空间都设置为只读。然后使子进程的 PTE 全部指向父进程中 PTE 存放的物理地址，这样便可以达到内存共享的目的。详细的代码设计如下：

dup_mmap 函数中：

```
Plain Text
bool share = 1;
```

在 copy_range 函数中：

设置父进程所有空间为只读的原因，是因为在之后的内存操作中，如果对这些空间进行写操作的话，程序就会触发 Page Fault 中断，那么 CPU 就可以在处理 Page Fault 中断的程序，即 vmm.c 中的 do_pgfault()函数中复制该内存，这也就是 COW 的写时复制。这部分详细代码设计如下：


```
Plain Text
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
{
    // 练习 2 部分的代码，未作任何改动，略去展示
}
```

在 do_pgfault 函数中：

当某个进程想写入一个共享内存时，由于 PTE 上的 PTE_W 为 0，所以会触发缺页中断处理程序。此时进程需要在缺页中断处理程序中复制该页内存，并设置该页内存所对应的 PTE_W 为 1。

此外需要注意的是，在执行缺页中断处理程序中的内存复制操作前，需要先检查该物

理页的引用次数。如果该引用次数已经为 1 了，则表明此时的物理页只有当前进程所使用，故可以直接设置该页内存所对应的 PTE_W 为 1 即可，不需要进行内存复制。

再之后的流程就与 Lab3 当中我们补充的 do_pgfault 部分基本一致，只是适当更改语句控制流，在其中加入一些安全检查工作而已。

此外，我们在整个 COW 的流程当中添加了相应有关 COW 的 print 语句，可以在用户程序输出的日志文件当中看到我们编写的有关 COW 的提示信息。

详细的代码设计如下：

```
Plain Text
if (*ptep == 0)
{ // 这部分代码没有改动
    if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
        cprintf("pgdir_alloc_page in do_pgfault failed\n");
        goto failed;
    }
}
else {
    struct Page *page = NULL;
    // 如果当前页错误的原因是写入了只读页面
    if (*ptep & PTE_V)
    {
        // 写时复制：复制一块内存给当前进程
        cprintf("\n\nCOW: ptep 0x%x, pte 0x%x\n",ptep, *ptep);
        // 原先所使用的只读物理页
        page = pte2page(*ptep);
        // 如果该物理页面被多个进程引用
        if(page_ref(page) > 1) {
            // 释放当前 PTE 的引用并分配一个新物理页
            struct Page* newPage = pgdir_alloc_page(mm->pgdir,
addr, perm);
            uintptr_t kva_src = page2kva(page);
            uintptr_t kva_dst = page2kva(newPage);
            // 拷贝数据内容
            memcpy(kva_dst, kva_src, PGSIZE);
        }
        // 如果该物理页面只被当前进程所引用,即 page_ref 等 1
        else {
            // 则可以直接执行 page_insert，保留当前物理页并重设其 PTE
权限。
            page_insert(mm->pgdir, page, addr, perm);
        }
```

```
    }
    else {
        if (swap_init_ok)
        {
            //分配一个内存页并将磁盘页的内容读入这个内存页
            if ((ret = swap_in(mm, addr, &page)) != 0) {
                cprintf("swap_in in do_pgfault failed\n");
                goto failed;
            }
            //建立 Page 的 phy addr 与线性 addr la 的映射
            page_insert(mm->pgdir,page,addr,perm);
        }
        else {
            cprintf("no swap_init_ok but ptep is %x, failed\n",
*ptep);
            goto failed;
        }
    }
    //设置页面可交换
    swap_map_swappable(mm, addr, page, 1);
    page->pra_vaddr = addr;
}
ret = 0;
failed:
    return ret;
```

Dirty COW 本地提权漏洞分析

Dirty COW 本地提权漏洞作用大致为：无特权的本地用户可以利用这个漏洞获得对只读内存映射的写访问权限，从而增加他们在系统上的特权。而在实际的漏洞利用当中，此漏洞允许具有本地系统帐户的攻击者修改磁盘上的二进制文件，绕过标准权限机制。

下面给出漏洞函数的代码:

```
Plain Text
long __get_user_pages(struct task_struct *tsk, struct mm_struct
*mm,
        unsigned long start, unsigned long nr_pages,
        unsigned int gup_flags, struct page **pages,
        struct vm_area_struct **vmas, int *nonblocking)
{
    // ....
  do {
retry:
        // 注意这里的进程调度
```

```
        cond_resched();
        // ......
        /* 查找虚拟地址的 page */
        page = follow_page_mask(vma, start, foll_flags,
&page_mask);
        if (!page) {
            /* 如果 page 找不到，则进行处理 */
            ret = faultin_page(tsk, vma, start, &foll_flags,
nonblocking);
            switch (ret) {
            case 0:
                goto retry;
            // ......
            }
        }
        if (page)
          // 加入 page 数组
    } while (nr_pages);
    // ...
}
```

代码执行流程如下：

1. 执行__get_user_pages 函数时，函数参数会携带一个 FOLL_WRITE 标记，用以指明当前操作是写入某个物理页。

2. 在 follow_page_mask 中，程序会找出特定的物理页。但大部分情况下第一次执行该函数时无法真正将该物理页的地址返回，因为可能存在缺页或者权限不够的情况（例如写入了一个只读页）。

3. 若 page 为 NULL，则之后会执行 faultin_page 函数对 follow_page_mask 的失败进行处理。包括但不限于分配新的页、修改页权限、页数据复制等等情况（上述说明的三种情况不一定会同时发生）。然后跳转至 retry 重新执行 follow_page_mask。

4. 经过几轮的循环后，当 faultin_page 函数再一次执行时，该函数会执行内存复制操作，以完成写时复制操作。同时 FOLL_WRITE 标记将会被抹去 ，之后跳转回 retry。因为 COW 已经执行完成，对于新的物理页无论是读还是写都没有问题，所以在下一次执行 follow_page_mask 函数时一定会返回该物理页，所以该标记已经失去了作用，可以被抹去。

5. 但此时需要注意的是，retry 下的第一条语句是 cond_resched 函数，它将会执行线程调度，执行其他线程。但倘若调度到的线程将之前新创建的物理页删除，则一旦重新调度回当前线程后，执行 follow_page_mask 返回的是之前的只读页。

6. 最后，该只读页被添加到 page 数组，并在接下来的操作中被成功修改。

以上即为 Dirty COW 漏洞的大致原理。

Challenge 2 回答：

在这次实验中，用户程序是通过在 `makefile` 的最后一步使用 `ld` 命令加载的，其中 `hello` 应用程序的执行代码 `obj/_user_hello.out` 被链接到了 `ucore kernel` 的末尾。同时，两个全局变量被用来记录 `hello` 应用程序的起始位置和大小，因此 `hello` 应用程序与 `ucore` 内核一同被 `bootloader` 加载到内存中。

与常见操作系统的加载方式相比，在大多数操作系统中，应用程序不会在系统启动时加载到内存中，只有当用户需要运行某个应用程序时，操作系统才会加载它。这种方法被称为延迟加载或按需加载。延迟加载的主要优点是有效管理系统资源，尤其是内存资源。如果操作系统在启动时就加载了所有可能需要的应用程序，内存资源将很快耗尽，通过实施延迟加载，操作系统确保只有真正需要运行的应用程序才会占用内存资源。但由于 ucore 没真正实现硬盘来储存用户软件，且其内核很小，出于简化教学的目的直接在系统启动时加载。