操作系统实验报告

实验名称：物理内存和页表

小组成员：李娅琦 周思洁 周末

一、实验内容

练习1: 加载应用程序并执行（需要编码）

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。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。

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。

1. 设计实现过程

1. 设置栈指针

   在 load_icode 函数中，首先要设置 tf->gpr.sp（即上下文切换时保存的寄存器中的栈指针）为用户栈的顶部地址（USTACKTOP）。这个设置确保用户程序运行时能正确地访问栈。

2. 设置程序计数器（epc）

   接下来，要将 tf->epc 设置为 ELF 文件头中的入口点地址 (elf->e_entry)。这样，当用户程序开始执行时，处理器知道从哪里开始运行代码。

3. 设置处理器状态寄存器（status）

   最后配置 tf->status，即处理器的状态信息。这里主要涉及两个状态位：

- SPP（Supervisor Previous Privilege）：

  SPP 表示在发生中断或异常之前的特权级别。对于用户态程序，由于在异常发生前用户态是正常的执行状态，SPP 应设置为 0，以便在处理完中断后能够使用 sret 指令返回至用户模式。

- SPIE（Supervisor Previous Interrupt Enable）：

  SPIE 用于表示中断使能状态。在用户态运行的程序出于正常执行的考虑，应该允许中断，因此 SPIE 应设置为 1，以确保用户程序能够正常接收中断。

2. 代码实现：

```
    /* LAB5:EXERCISE1 YOUR CODE
     * should set tf->gpr.sp, tf->epc, tf->status
     * NOTICE: If we set trapframe correctly, then the user level process can
return to USER MODE from kernel. So
     *          tf->gpr.sp should be user stack top (the value of sp)
     *          tf->epc should be entry point of user program (the value of sepc)
     *          tf->status should be appropriate for user program (the value of
sstatus)
     *          hint: check meaning of SPP, SPIE in SSTATUS, use them by
SSTATUS_SPP, SSTATUS_SPIE(defined in risv.h)
     */
    tf->gpr.sp = USTACKTOP;
    // Set the entry point of the user program
    tf->epc = elf->e_entry;
    // Set the status register for the user program
    tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
```

3. 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。

1. 进程选择与切换（调度器选择进程）

- 调度器选择进程: 从就绪队列中选择一个 PROC_RUNNABLE 状态的进程。
- 进程切换: 通过上下文切换（Context Switch），将当前进程的状态保存（寄存器、堆栈等），然后加载目标进程的状态。
- 准备执行: 切换后的进程会被加载到 CPU 中，并开始执行，通常会先执行一些初始化工作，比如设置中断帧等。

2. 准备加载新的执行代码

- 清空用户态内存空间
- 判断进程是否有用户态内存（mm）
- 加载应用程序执行代码（load_icode）

3. 内存管理数据结构的创建

为每个进程创建和初始化内存管理数据结构（mm）：

- mm_create: 创建进程的内存管理结构，并为它分配所需的内存空间。
- setup_pgdir: 创建一个页目录表，并将内核虚拟地址空间映射到此页表中。页目录表管理着虚拟地址到物理地址的映射

4. 用户虚拟内存空间的建立

- 加载 ELF 文件的各个段: 使用 load_segment 函数将 ELF 文件中的代码段、数据段加载到内存中。
- 栈的映射: 为用户进程分配栈空间，通常栈从虚拟地址空间的高端开始。栈的大小和位置是固定的，栈顶通常设置为 USTACKTOP。

5. 设置进程的执行现场（Trapframe）

在进程切换到用户模式之前，内核需要设置进程的 中断帧（Trapframe），以确保在执行中断返回指令（如 iret）后，进程能够从用户模式开始执行。

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。

```
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_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);
            // alloc a page for process B
            struct Page *npage = alloc_page();
            assert(page != NULL);
            assert(npage != NULL);
            int ret = 0;
            /* LAB5:EXERCISE2 YOUR CODE
             * replicate content of page to npage, build the map of phy addr of
             * nage with the linear addr start
             *
             * Some Useful MACROs and DEFINEs, you can use them in below
             * implementation.
             * MACROs or Functions:
             *    page2kva(struct Page *page): return the kernel vritual addr of
             * memory which page managed (SEE pmm.h)
             *    page_insert: build the map of phy addr of an Page with the
             * linear addr la
             *    memcpy: typical memory copy function
             *
             * (1) find src_kvaddr: the kernel virtual address of page
             * (2) find dst_kvaddr: the kernel virtual address of npage
```

```
             * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
             * (4) build the map of phy addr of  nage with the linear addr start
             */

            void * kva_src = page2kva(page);
            void * kva_dst = page2kva(npage);

        memcpy(kva_dst, kva_src, PGSIZE);

        ret = page_insert(to, npage, start, perm);
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

- page2kva(page) 和 page2kva(npage)：这两个函数将物理页面（page 和npage）映射到内核虚拟地址空间
- 通过**memcpy**将虚拟地址进行复制，复制其内容。
- 最后使用前面的参数（to是目标进程的页目录地址，npage是页，start是起始地址，perm是提取出的页目录项ptep中的PTE_USER即用户级别权限相关的位）**调用page_insert函数。**

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。

由于时间原因，只对该机制做一个简要的设计： **关键步骤**

1. **资源初始化**：资源最初由一个使用者创建，并可以被多个使用者共享。
2. **读操作**：共享资源的多个使用者可以直接读取数据，资源的引用计数增加。
3. **写操作**：
    - 当某个使用者进行写操作时，首先检查该资源是否为共享状态。
    - 如果是共享的，则拷贝资源的内容，生成该使用者的私有副本，并允许修改副本。
4. **销毁资源**：当资源的引用计数为 0 时，销毁该资源。

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）

1. `fork` **函数:** 调用用于创建一个与当前进程几乎相同的子进程。子进程从父进程返回点开始执行

- **用户态**：
    - 用户调用 `fork()` 发起系统调用，传递创建新进程的请求
    - 用户程序根据返回值判断自己是父进程（返回非零 PID）还是子进程（返回 0）
- **内核态**：

    - `sys_fork` 调用 `do_fork` 进行实际的进程复制。内核会分配新的虚拟内存空间、栈空间，并初始化子进程的上下文（CPU 寄存器等）
    - 内核会将父进程的上下文和中断帧复制给子进程，并为子进程分配 PID。
    - 内核唤醒新线程并加入调度队列，等待执行
- **用户态返回**：
    - 当内核执行完成，`fork` 系统调用返回，父进程收到子进程的 PID，子进程收到 0

2. `exec` **函数**: 调用用来替换当前进程的程序，加载新的程序并执行

- **用户态**：
    - 用户调用 `exec()` 发起系统调用，传递新程序的路径和参数。
- **内核态**：
    - `sys_exec` 调用 `do_execve` 来加载新的程序。内核会回收当前进程的虚拟内存空间，并为当前进程分配新的虚拟内存。
    - 内核将新程序的代码和数据加载到进程的地址空间，更新进程的堆栈等资源，并进行程序的初始化。
    - 内核为进程准备新的堆栈和执行环境，传递命令行参数和环境变量。
- **用户态返回**：
    - 执行完成后，新的程序从其入口点开始执行，原有的程序代码被完全替换，进程的 PID 保持不变。

3. `wait` **函数**: 调用用于父进程等待子进程的退出，并获取其退出状态

- **用户态**：
    - 父进程调用 `wait()`，请求操作系统等待子进程结束。
- **内核态**：
    - `sys_wait` 调用 `do_wait` 来处理进程间的同步。内核检查是否有子进程退出。
    - 如果子进程尚未退出，父进程会被挂起，进入阻塞状态，直到子进程退出并返回其状态。
    - 如果子进程已经退出，内核会将其退出状态返回给父进程。
- **用户态返回**：
    - 父进程从 `wait()` 返回，获得子进程的退出状态，继续执行。

4. `exit` **函数**: 用于终止当前进程的执行，并释放其资源

- **用户态**：
    - 用户调用 `exit()` 以结束当前进程。
- **内核态**：
    - `sys_exit` 调用 `do_exit` 来处理进程退出。内核会回收进程的虚拟内存、关闭文件描述符并释放其他资源。
    - 内核将进程的状态标记为“僵尸”状态，等待父进程调用 `wait()` 获取退出状态。
    - 如果父进程存在，内核会唤醒父进程，通知其子进程已经结束。
- **用户态返回**：
    - 当前进程终止，操作系统会销毁进程的资源，进程表中的该条目被删除。

5. **执行流程分析**

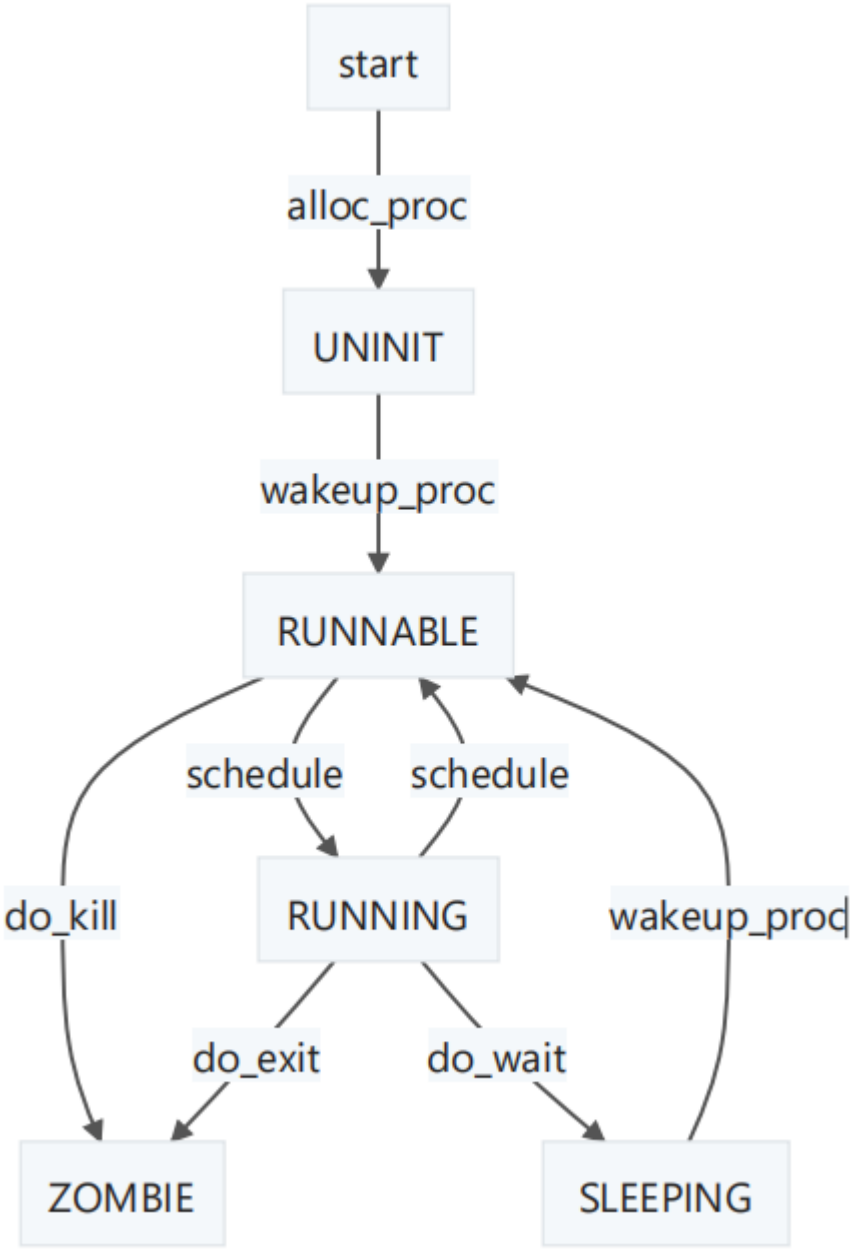
- **用户态与内核态的交替执行**：

    - **系统调用触发**：用户程序通过调用系统调用（`fork()`、`exec()`、`wait()`、`exit()`）发起内核操作。此时，CPU 从**用户态切换到内核态**，执行系统调用的相关处理

    - **内核态操作**：内核负责完成系统调用的实际操作，如进程复制、程序加载、资源回收等
    - **返回用户态**：当内核完成操作后，通过`sret`指令切换回用户态，继续执行用户程序。系统调用的返回值通过寄存器返回给用户程序

- **内核态执行结果如何返回给用户程序**：
    - 通过系统调用的返回值，内核将执行结果传递给用户程序
        - `fork()`返回子进程的 PID（父进程）或 0（子进程）
        - `exec()`返回 -1（失败），并设置`errno`
        - `wait()`返回子进程的退出状态
    - 这些结果通过寄存器（如`eax`寄存器）传递给用户程序

2. **第二个问题**



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？

1. COW机制中的用户程序何时被加载到内存中？

在使用COW机制时，用户程序的加载并不会在进程创建时立即进行，而是采用延迟加载的方式，只有在进程或其子进程实际访问某个内存页面时，内核才会将该页面加载到内存中。大致流程如下：

1. **进程创建时（fork或execve）：**
    - 当通过 `fork()` 创建一个新的子进程时，子进程会与父进程共享相同的内存页（代码段、数据段等）。这时，内存页的内容并没有复制到子进程，而是两个进程都指向相同的物理内存页。
    - 对于共享的内存页，操作系统会将页的访问权限设置为只读。这样，如果父进程或子进程尝试修改这些页，操作系统会通过页故障（page fault）机制触发相应的处理。
2. **写时复制（Copy on Write）：**
    - 当父进程或子进程试图修改共享的内存页时，发生页故障，操作系统会检测到该页是只读的，并触发COW机制。
    - 操作系统会为触发写操作的进程分配一个新的物理页，并将该页的内容从共享页复制到新的物理页中。然后，将新的物理页映射到该进程的虚拟地址空间中，并更新页表。
    - 此后，修改的进程就可以在新的物理页上进行写操作，而不会影响到另一个进程的内存。
3. **延迟加载：**
    - COW机制利用延迟加载的方式，避免了在进程创建时就进行大量的内存复制。在实际的内存访问时，只有被修改过的内存页才会被复制。
    - 这种方式与传统的内存分配方式不同，因为传统的进程复制通常会在 `fork()` 或 `execve()` 时立即为每个进程分配独立的内存页面。

2. 与常用操作系统加载方式的区别

传统加载方式：

1. **进程创建（fork）或加载（execve）时：**
    - 在常规的操作系统中，进程的内存空间在 `fork()` 或 `execve()` 时会进行完整的复制。也就是说，在进程创建时，内核会为子进程分配独立的物理内存页，复制父进程的内存内容（包括代码段、数据段、堆栈等）。这时，父进程和子进程的内存是完全独立的。
2. **内存复制：**
    - 在传统的加载过程中，内核会为每个进程分配独立的物理内存页面，这需要进行大量的内存复制，尤其在进程创建时。这种方法导致了较高的内存开销，尤其在进程只读取而不修改数据时，这些内存复制是不必要的。

COW加载方式：

1. **进程创建时：**
    - 在COW机制下，进程创建时，父进程和子进程共享相同的内存页面，直到其中一个进程修改这些页面。这种方式减少了内存复制的开销，避免了不必要的内存分配。
2. **内存复制：**
    - 只有当进程需要修改其内存中的某个页面时，操作系统才会进行真正的内存复制，这时会为该进程分配一个新的物理内存页。直到这时，才会将数据从共享页面复制到新的页面中。这种“懒加载”方式能够有效地减少内存的浪费。

COW机制与传统的内存复制机制的根本区别在于 **延迟复制** 和 **共享内存** 的使用。传统的复制机制认为每个进程应该拥有独立的内存空间，而COW则认为在很多情况下，进程并不需要修改所有的内存内容。因此，COW机制通过共享内存来避免不必要的内存复制，提高了内存的使用效率，尤其在大多数进程只是读取数据而不修改时。

3. 原因：

- **内存效率**：COW 通过延迟复制内存页，避免了不必要的内存复制，从而节省了内存和复制时间。
- **快速创建进程**：传统的内存复制会导致进程创建时的延迟，而COW机制通过共享内存、懒加载和按需复制的方式使进程创建更为迅速。
- **性能优化**：在进程大多数时间内不会修改自己的内存空间时，COW避免了多余的内存分配和复制，提高了系统的整体性能。