OS Lab5实验报告

组员：刘俊彤，刘玉菡，孙启森

实验目标

实验4完成了内核线程，但到目前为止，所有的运行都在内核态执行。实验5将创建用户进程，让用户进程在用户态执行，且在需要ucore支持时，可通过系统调用来让ucore提供服务。为此需要构造出第一个用户进程，并通过系统调用sys_fork/sys_exec/sys_exit/sys_wait来支持运行不同的应用程序，完成对用户进程的执行过程的基本管理。

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。

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

将sp设置为栈顶，epc设置为文件的入口地址，使得返回用户空间时从这个文件开始执行。sstatus的SPP位清零，代表异常来自用户态.这样中断返回后就会返回用户态。

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。

- 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。
- 在user_main中通过宏KERNEL_EXECVE，调用kernel_execve
- 在kernel_execve中执行ebreak，发生断点异常，然后到达CAUSE_BREAKPOINT处调用syscall
- 在syscall中根据参数，确定执行sys_exec，调用do_execve
- do_execve首先会检查并释放旧的内存空间进行初始化然后调用load_icode
- 在load_icode中将提供的二进制程序（ELF格式）加载到当前进程的内存空间中
- 然后设置中断帧，使得中断返回后能够进入用户态执行程序。

练习2：父进程复制自己的内存空间给子进程

第一题：补充copy_range的实现

调用过程：`copy_mm`-->`dup_mmap`-->`copy_range`

`copy_range`函数实现：

函数接受五个参数：

- `pde_t *to`：目标进程的页目录表指针
- `pde_t *from`：原进程的页目录表指针

- `uintptr_t start`：待复制内存的起始位置（用户空间）
- `uintptr_t end`：待复制内存的结束位置
- `bool share`：true表示共享页面，false表示创建页面的新副本。这里只使用了false即创建新副本的情况。

具体实现：

```
int copy_range(pde_t *to, pde_t *from, uintptr_t start, uintptr_t end,
               bool share) {
    assert(start % PGSIZE == 0 && end % PGSIZE == 0); // 保证页面对齐
    assert(USER_ACCESS(start, end)); // 保证页面位于用户空间
    do {
        pte_t *ptep = get_pte(from, start, 0), *nptep; // 找到原进程对应的页表项
        if (ptep == NULL) { // 不存在则跳过 start指向下一个页面
            start = ROUNDDOWN(start + PTSIZE, PTSIZE);
            continue;
        }
        if (*ptep & PTE_V) {  // 检查页表项是否有效 无效则跳过该页面
            if ((nptep = get_pte(to, start, 1)) == NULL) { // 为目标进程分配页表项
                return -E_NO_MEM;
            }
            uint32_t perm = (*ptep & PTE_USER); // 获取原程序的用户权限标志
            struct Page *page = pte2page(*ptep); // 获取原进程物理页
            struct Page *npage = alloc_page(); // 为目标进程分配物理页
            assert(page != NULL);
            assert(npage != NULL);
            int ret = 0;
            uintptr_t* src_kvaddr = page2kva(page); // 获取原页面的虚拟地址（内核空
间）
            uintptr_t* dst_kvaddr = page2kva(npage); // 获取目标页面的虚拟地址（内核空
间）
            memcpy(dst_kvaddr,src_kvaddr,PGSIZE); // 复制
            ret = page_insert(to,npage,start,perm); // 目标进程的页目录表中插入新页
            assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

`start`是原进程物理页对应的用户空间的虚拟地址，`src_kvaddr`是原进程物理页对应的内核空间的映射位置。需要`src_kvaddr`是因为内核运行在特权模式下，不能直接访问用户空间的虚拟地址，需要通过内核空间的映射来访问物理内存。

第二题：设计Copy on Write机制

- 当进程调用fork时，子进程继承父进程的内存映射，但将所有的页面都标记为只读和共享。这样父进程和子进程就可以共享物理页面
- 当一个进程尝试写入页面操作时，因为所有页面的权限都是只读的，这时就会触发一个Page Fault， 这里可以定义一种新的异常类型COW
- 当遇到COW异常时，对应的处理函数中要重新分配物理页，将原来页面中的内容复制到新的物理页中（也就是copy_range操作），并更新页表项使新页可写
- 将要写的对象从原页面改为新分配的这个可写的副本，就可以实现写操作，且这个资源改变对其他进程不可见。

练习3：理解进程执行fork/exec/wait/exit 的实现，以及系统调用的实现

1. `fork` 的执行流程分析

功能：`fork` 系统调用用于创建一个与当前进程（父进程）几乎完全相同的子进程。

执行流程：

- **用户态**：
  - 用户程序调用 `fork()`，触发系统调用中断（如 `syscall` 指令）。
- **内核态**：
  - 系统调用处理函数 `syscall()` 被触发。
  - 在 `syscall.c` 中，调用 `sys_fork`，传递当前进程的 `trapframe`。
  - `sys_fork` 调用 `do_fork` 来实际创建子进程。
- `do_fork` **函数（`proc.c`）**：
  - **分配和初始化**：调用 `alloc_proc()` 分配新的进程结构，初始化状态、PID 和内核栈。
  - **设置内核栈**：通过 `setup_kstack()` 为子进程分配内核栈空间。
  - **复制内存空间**：调用 `copy_mm()` 复制或共享父进程的内存，采用写时复制（COW）机制。
  - **复制线程上下文**：使用 `copy_thread()` 复制父进程的 `trapframe` 和上下文信息。
  - **分配 PID 并加入进程列表**：调用 `get_pid()` 分配唯一 PID，并将子进程加入进程哈希表和链表。
  - **唤醒子进程**：调用 `wakeup_proc(proc)` 将子进程状态设为 `PROC_RUNNABLE`。
  - **返回 PID**：`do_fork` 返回子进程的 PID 给 `sys_fork`，再返回给用户态。
- **用户态**：
  - `fork()` 返回给父进程子进程的 PID，子进程从 `fork()` 返回 `0`。

用户态与内核态的交错执行：

- 用户态通过系统调用触发内核态执行 `do_fork`。
- 内核态完成进程创建后，修改 `trapframe` 并使用 `sret` 指令返回用户态，传递返回值。

---

2. `exec` 的执行流程分析

功能：`exec` 系统调用用于用一个新的程序替换当前进程的地址空间和执行上下文。

执行流程：

- **用户态**：
  - 用户程序调用 `exec()`，触发系统调用中断。
- **内核态**：
  - 系统调用处理函数 `syscall()` 被触发。
  - 在 `syscall.c` 中，调用 `sys_exec`，传递程序名、长度、二进制代码及其大小。
  - `sys_exec` 调用 `do_execve` 进行程序替换。
- `do_execve` **函数（`proc.c`）**：
  - **参数校验**：检查程序名和代码段地址是否合法。
  - **释放当前内存**：如果有，调用 `exit_mmap()`、`put_pgdir()`、`mm_destroy()` 释放当前内存空间。

    - **加载新程序**：调用 `load_icode()` 加载 ELF 格式的可执行文件，设置新的页目录和用户栈，配置 `trapframe` 的入口地址（`epc`）。
    - **设置进程名称**：调用 `set_proc_name()` 更新进程名称。
    - **返回用户态**：加载成功返回 `0`，失败则调用 `do_exit()` 终止进程。
- **用户态**：
    - `exec()` 成功后，当前进程执行新程序的入口地址。

用户态与内核态的交错执行：

- 用户态通过 `exec()` 系统调用切换到内核态，执行 `do_execve`。
- 内核完成程序替换后，通过修改 `trapframe` 并使用 `sret` 指令返回用户态。

---

3. `wait` 的执行流程分析

功能：`wait` 系统调用使父进程等待其子进程结束，并回收子进程的资源。

执行流程：

- **用户态**：
    - 用户程序调用 `wait(pid, &status)`，触发系统调用中断。
- **内核态**：
    - 系统调用处理函数 `syscall()` 被触发。
    - 在 `syscall.c` 中，调用 `sys_wait`，传递子进程的 PID 和状态存储地址。
    - `sys_wait` 调用 `do_wait` 执行等待操作。
- `do_wait` **函数（`proc.c`）**：
    - **参数校验**：验证 `code_store` 指针合法性。
    - **查找子进程**：根据 PID 查找目标子进程，若 PID 为 `0` 则等待任意子进程。
    - 检查子进程状态：
        - 若子进程为 `PROC_ZOMBIE`，则回收资源：写入退出状态、移除进程列表、释放内核栈和进程结构。
        - 若子进程未退出，将当前进程设为 `PROC_SLEEPING` 并调用 `schedule()`。
    - **返回结果**：成功回收返回 `0`，无符合条件子进程返回 `-E_BAD_PROC`。
- **用户态**：
    - `wait()` 返回子进程的退出状态，供父进程处理。

用户态与内核态的交错执行：

- 用户态通过 `wait()` 系统调用切换到内核态，执行 `do_wait`。
- 若子进程未退出，内核将父进程设为 `PROC_SLEEPING` 并调度其他进程。
- 子进程退出后，内核唤醒父进程，继续执行 `do_wait` 并返回结果。

---

4. `exit` 的执行流程分析

功能：`exit` 系统调用用于终止当前进程，并释放其占用的资源。

执行流程：

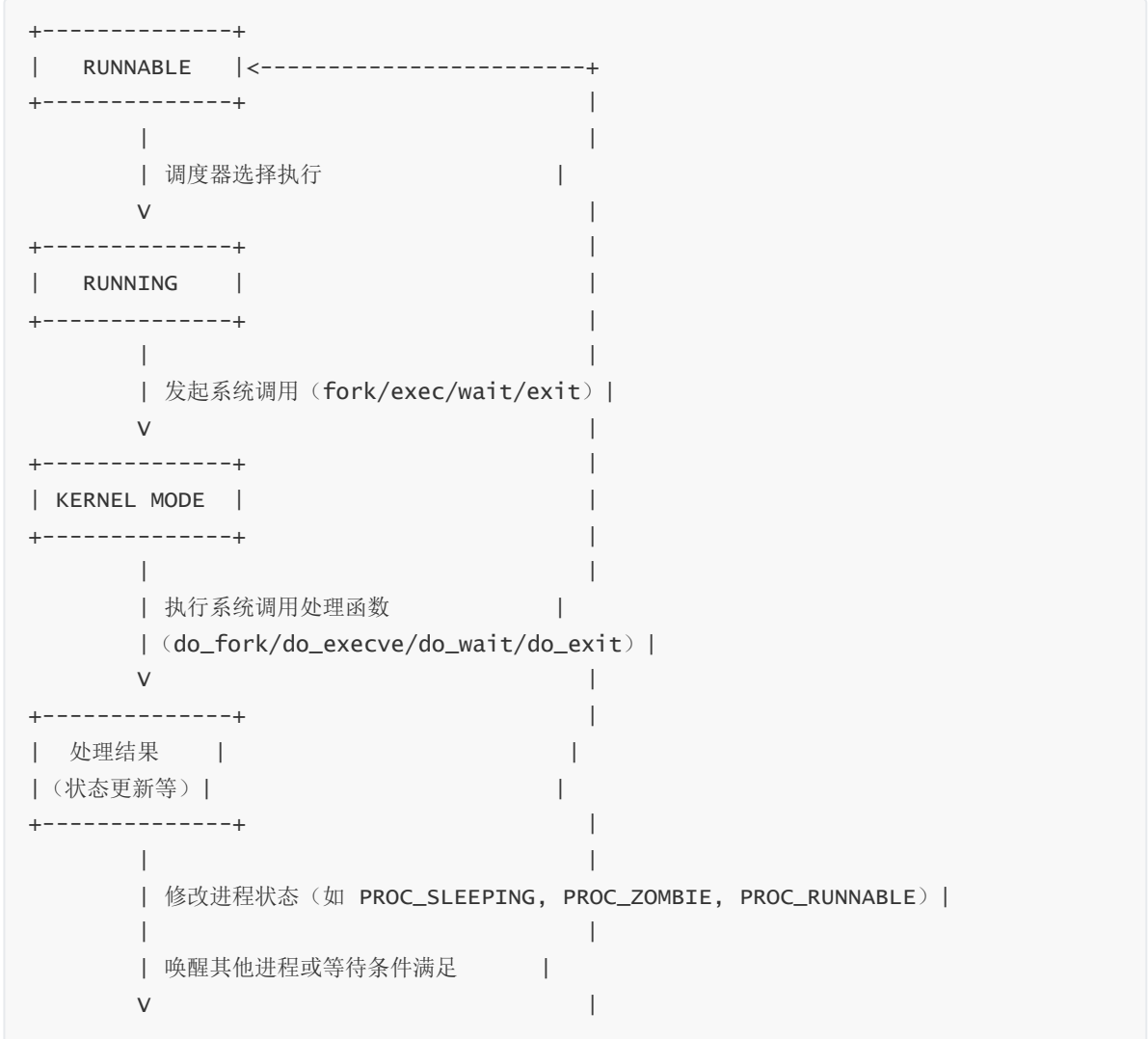
- **用户态**：

    - 用户程序调用 `exit(status)`，触发系统调用中断。
- **内核态**：
    - 系统调用处理函数 `syscall()` 被触发。
    - 在 `syscall.c` 中，调用 `sys_exit`，传递错误码。
    - `sys_exit` 调用 `do_exit` 执行退出操作。
- `do_exit` **函数（**`proc.c`**）**：
    - **检查特殊进程**：若当前进程为 `idleproc` 或 `initproc`，触发内核 panic。
    - **释放内存**：调用 `exit_mmap()`、`put_pgdir()`、`mm_destroy()` 释放内存空间。
    - **设置为僵尸状态**：更新进程状态为 `PROC_ZOMBIE` 并记录退出码。
    - **处理子进程**：将子进程的父进程指向 `initproc`，必要时唤醒 `initproc`。
    - **唤醒父进程**：若父进程在等待，调用 `wakeup_proc(parent)` 唤醒父进程。
    - **调度其他进程**：调用 `schedule()` 切换到其他可运行进程。
    - **终止执行**：进程标记为 `PROC_ZOMBIE` 后，不再返回用户态。
- **用户态**：
    - 进程退出，资源被回收，控制权转移给调度器选择的其他进程。

用户态与内核态的交错执行：

- 用户态通过 `exit()` 系统调用切换到内核态，执行 `do_exit`。
- 内核完成资源释放后，通过调度器切换到其他进程，当前进程不再执行。

用户态进程的执行状态生命周期图



状态说明：

- **RUNNABLE**：进程处于可运行状态，等待调度器选择执行。
- **RUNNING**：进程正在 CPU 上执行。
- **KERNEL MODE**：进程通过系统调用切换到内核态，执行内核操作。
- **SLEEPING**：进程因等待某条件（如子进程退出）被阻塞。
- **PROC_ZOMBIE**：进程已结束执行，但资源尚未被父进程回收。

状态变换触发事件：

- **RUNNABLE → RUNNING**：调度器选择该进程执行。
- **RUNNING → KERNEL MODE**：进程发起系统调用（`fork`、`exec`、`wait`、`exit`）。
- **KERNEL MODE → RUNNABLE/SLEEPING**：根据系统调用结果，进程状态更新为 `PROC_RUNNABLE` 或 `PROC_SLEEPING`。
- **SLEEPING → RUNNABLE**：被唤醒事件（如子进程退出）发生。
- **RUNNING → PROC_ZOMBIE**：进程调用 `exit`，进入僵尸状态。
- **PROC_ZOMBIE → RUNNABLE**：父进程通过 `wait` 回收资源，僵尸进程被删除。

重要知识点与操作系统原理对应关系

实验知识点	操作系统原理知识点	理解与关系
`do_fork` 实现进程创建	进程创建与进程控制块（PCB）	`do_fork` 分配并初始化新的 PCB，类似于操作系统中进程创建的机制。
写时复制（Copy-on-Write, COW）机制	虚拟内存管理与页表共享	COW 通过共享页表项并在写操作时复制页，实现高效内存利用，与操作系统的虚拟内存管理一致。
`do_execve` 实现程序替换	程序执行与地址空间管理	`do_execve` 替换进程的地址空间，类似于操作系统中程序执行的描述。
进程等待与回收（`do_wait`）	父子进程关系与资源回收	`do_wait` 实现父进程等待子进程退出并回收资源，符合操作系统中进程间关系管理。
进程退出与僵尸状态（`do_exit`）	进程终止与资源释放	`do_exit` 设置进程为僵尸状态并释放资源，与操作系统中进程终止的处理一致。
系统调用处理机制	系统调用接口与用户态/内核态切换	系统调用通过中断机制切换到内核态，执行内核函数，再返回用户态，符合操作系统原理。

扩展练习challenge的第二小问

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 ucore 实验中，用户程序是在内核初始化阶段被预先加载到内存中的。具体来说，通过 Makefile 中的 `ld` 命令，用户程序（例如 `hello` 应用程序）的执行码 `obj/__user_hello.out` 被静态链接到 ucore 内核的末尾。启动时，bootloader 将整个内核镜像，包括预先链接的用户程序，一同加载到内存中，并通过全局变量记录了用户程序的起始位置和大小。因此，用户程序与内核一起在系统启动时便驻留在内存中，并由内核的第二个线程 `init_proc` 立即执行。

与常用操作系统在加载用户程序的方式上的区别：一般操作系统采用按需加载（延迟加载）的策略，用户程序在运行时通过文件系统从磁盘动态读取并加载到内存中。这种动态加载方式能够有效管理系统资源，避免在启动时占用过多内存，仅在用户真正需要运行某个程序时才加载，从而提高内存利用效率。

但是在本次实验中，用户程序与内核一起被静态加载到内存中，而不是通过动态加载。因为实验环境主要关注于理解和实现进程管理的核心机制，而不涉及复杂的文件系统和动态加载机制。通过预先加载用户程序，简化了实验环境，让我们能够专注于进程创建、管理和系统调用的实现，而无需处理程序加载的细节。这种方法确保了用户程序能够在系统启动后立即执行，便于实验的顺利进行。