操作系统实验报告

实验名称：用户程序 实验地点：图书馆323

组号：56 小组成员：周钰宸 王志远 齐明杰

一、实验目的

- 了解第一个用户进程创建过程
- 了解系统调用框架的实现机制
- 了解ucore如何实现系统调用sys_fork/sys_exec/sys_exit/sys_wait来进行进程管理

二、实验过程

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。

1.1 问题1

回答：

首先我们对位于`kern/process/proc.c`中的`load_icode`函数进行续写，完成第六步设置中断帧的过程。确保要运行的应用程序能正确加载到进程空间，中断返回后进入**U mode**并从应用程序起始地址开始执行。

为了做到这个，我们需要设置三步：

- 设置tf的gpr.sp即存储栈顶指针为**用户栈的顶部地址（`USTACKTOP`）**，以便在用户程序运行时可以正确地访问栈。
- 设置tf的epc存储**发生异常或中断时的程序计数器的值为elf的e_entry。**elf是在load_icode的前面通过`struct elfhdr *elf = (struct elfhdr *)binary;`方式定义的二进制ELF文件的文件头，**elf的e_entry就是用户程序的入口点地址了**，以便在用户程序运行时可以从正确的位置开始执行。
- 设置tf的status即存储处理器的状态信息。其中用到了两个状态位SSTATUS的SPP和SPIE：
  - **SPP**:用于表示处理器在发生异常或中断之前的特权级别。它可以有两个可能的值：0表示处理器在发生异常或中断之前处于用户模式（User Mode），1表示处理器在发生异常或中断之前处于特权模式（Supervisor Mode）。又因为统调用 sys_exec 之后， 我们在 trap 返回的时候调用了 sret 指令。sret指令会根据SPP的值回到中断前的状态。**因为我们如果是在用户态可能发生中断而来到特权态处理，为了最终通过sret返回U Mode，所以SPP应该为0。**
  - **SPIE**：用于表示处理器在发生异常或中断之前的中断使能状态。它可以有两个可能的值：0表示处理器在发生异常或中断之前中断被禁用，1表示处理器在发生异常或中断之前中断被启用。**为了保证用户态能够正常触发中断，因此应该启用中断，即SPIE应该为1。**

```
/* LAB5:EXERCISE1 YOUR CODE */
    // Set the user stack top
    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;
```

1.2 问题2

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。

回答：

在一个用户进程被ucore内核选择，实际上这个选择指的是当**操作系统的调度器从就绪进程队列中选择了一 个就绪进程后**，通过执行**进程切换**，就让这个被选上的就绪进程执行了。不过还需要做一些准备，包括设置中断帧等，具体如下：

1. **准备加载新的执行码**
    - **清空用户态内存空间准备加载新的执行码：具体由do_execve实现。对mm进行一些判断。**
        - **mm不为空**：**说明是用户进程，设置页表为内核空间页表**，目的是转入内核态。
        - **mm引用数为1**：意味着只有当前进程在使用这块内存。如果这个进程结束了，那么它所占用的用户空间内存和进程页表本身所占空间就没有其他进程会再使用，因此这些空间应该被释放，以便可以被其他进程使用。**此时释放进程所占用户空间内存和进程页表本身所占空间，可以有效地管理和利用有限的内存资源。**
2. **加载应用程序执行码与建立用户环境：包括读取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。主要由load_icode实现。**
    - **mm_create**:申请进程的内存管理数据结构mm所需内存空间，并对mm进行初始化。
    - **setup_pgdir:**申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表的内容拷贝到此新目录表中。**能够正确映射内核虚空间。**
    - **解析此ELF格式的执行程序**，然后调用**mm_map**根据执行程序说明的各个段的起始位置和大小建立对应的vma结构，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。**
    - **分配物理内存空间，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。
    - **调用mm_map设置用户栈，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，并分配一定数量的物理内存且建立好栈的虚地址和物理地址映射关系。
3. **更新用户进程的虚拟内存空间**：把mm->pgdir赋值到cr3寄存器中，更新了用户进程的虚拟内存空间。
4. **建立进程的执行现场**：清空进程的中断帧，重新设置进程的中断帧，使得在执行中断返回指令“iret”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。

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*。

2.1 问题1

回答：

copy_range函数的调用过程：do_fork()---->copy_mm()---->dup_mmap()---->copy_range()。

其中：

- do_fork函数调用的copy_mm函数在LAB4中没有实现，其他的过程和lab4一样，都是创建一个进程，并放入CPU中调度。**本次我们要重点实现如何完成从父进程把内存复制到子进程中。**
- copy_mm：使用互斥锁（用于避免多个进程同时访问内存）。进下一层的调用dup_mmap函数。
- dup_mmap接受两个参数，前一个mm是待复制的内存，而复制的源内容在oldmm（父进程）内容中。只是完成了新进程中的段创建。**具体行为最终落到copy_range中。**

最后到了copy_range中，**用于在内存页级别上复制一段地址范围的内容。**首先，它通过`get_pte`函数获取源页表中的页表项，并检查其有效性。然后它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，它确保源页和新页都成功分配，**并准备进行复制操作，也就是我们需要完成的。主要分为四步：**

- 首先通过刚开始获取的page即源page通过宏page2kva**转换为源虚拟内存地址。**
- 同样的将要复制过去的n个page**转换为目的虚拟内存地址。**
- 通过**memcpy**将虚拟地址进行复制，复制其内容。
- 最后我们再使用前面的参数（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。**

```
1 void *src_kvaddr = page2kva(page); // Get the kernel virtual address of the
  source page.
2             void *dst_kvaddr = page2kva(npage); // Get the kernel virtual
  address of the destination page.
3 
4             memcpy(dst_kvaddr, src_kvaddr, PGSIZE); // Copy the content of
  the source page to the destination page.
5 
6             ret = page_insert(to, npage, start, perm); // Insert the
  destination page into the page table of the target process.
```

2.2 问题2

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。

回答：

出于时间原因，这里只对Copy On Write机制设计一个简单思路。我们认为实现COW机制可以分为以下几个步骤：

1. **资源共享**：当多个任务读取同一资源时，它们共享对该资源的访问，而不是复制资源。**具体来说可以通过copy_range的share参数实现，在copy_range中要根据share参数判断一下是应该直接dup还是共享。**

2. **检测写操作**：当一个任务试图写入共享资源时，系统需要捕获这个操作。通过内存保护硬件实现，当任务试图写入一个标记为只读的内存区域时，硬件触发一个异常。**可以通过定义一个新的trap类型，然后到trap.c的exception_handler中对应处理。**
3. **资源复制**：当系统捕获到写操作时，它会分配新的内存或磁盘空间，并将原始资源的内容复制到新分配的空间中。**可以调用copy_range实现。**
4. **更新指针**：系统将试图写入资源的任务的**指针**更新为指向新复制的资源，然后允许写操作继续进行。这样写操作只影响新复制的资源，而不影响其他任务看到的原始资源。

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）

回答：

3.1 问题1

1. **fork**：完成进程的拷贝。由do_fork函数完成。

   调用过程：

   ```
   fork->SYS_fork->do_fork+wakeup_proc
   ```

   进程调用`fork`系统调用，进入正常的中断处理机制，最终交由`syscall`函数进行处理，在`syscall`函数中，根据系统调用，交由`sys_fork`函数处理，该函数进一步调用了`do_fork`函数。

   整体流程：

   - 首先检查当前总进程数目是否到达限制，如果到达限制，那么返回`E_NO_FREE_PROC`；
   - 调用`alloc_proc`来创建并初始化一个进程控制块；
   - 调用`setup_kstack`为内核进程（线程）建立栈空间、分配内核栈；
   - 调用`copy_mm`拷贝或者共享内存空间；
   - 调用`copy_thread`复制父进程的中断帧和上下文信息；
   - 调用`get_pid()`为进程分配一个PID；
   - 将进程控制块加入哈希表和链表，并实现相关进程的链接；
   - 最后返回进程的PID
2. **exec**：完成用户进程的创建工作，同时使用户进程进入执行。由do_exec函数完成。

   调用过程：

   ```
   SYS_exec->do_execve
   ```

   整体流程：

   - 检查进程名称的地址和长度是否合法，如果合法，那么将名称暂时保存在函数栈中，否则返回`E_INVAL`；
   - 将cr3页表基址指向内核页表，然后实现对进程的内存管理区域的释放；
   - 调用`load_icode`将代码加载进内存并建立新的内存映射关系，如果加载错误，那么调用`panic`报错；
   - 调用`set_proc_name`重新设置进程名称。

3. **wait：**完成对子进程的内核栈和进程控制块所占内存空间的回收。由do_wait函数完成。

   调用过程：

   ```
   SYS_wait->do_wait
   ```

   整体流程：

   - 首先检查用于保存返回码的`code_store`指针地址位于合法的范围内；
   - 根据PID找到需要等待的子进程PCB，循环询问正在等待的子进程的状态，直到有子进程状态变为`ZOMBIE`：
     - 如果没有需要等待的子进程，那么返回`E_BAD_PROC`；
     - 如果子进程正在可执行状态中，那么将当前进程休眠，在被唤醒后再次尝试；
     - 如果子进程处于僵尸状态，那么释放该子进程剩余的资源，即完成回收工作。
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：

   ```
   SYS_exit->exit
   ```

   整体流程：

   - 释放进程的虚拟内存空间；
   - 设置当期进程状态为`PROC_ZOMBIE`即标记为僵尸进程
   - 如果父进程处于等待当期进程退出的状态，则将父进程唤醒；
   - 如果当前进程有子进程，则将子进程设置为`initproc`的子进程，并完成子进程中处于僵尸状态的进程的最后的回收工作
   - 主动调用调度函数进行调度，选择新的进程去执行。

用户态与内核态操作分析：

1. **fork:**
   - 用户态：父进程调用 `fork()` 系统调用。
   - 内核态：内核复制父进程的所有资源（内存、文件描述符等），创建一个新的子进程。
   - 用户态：子进程从 `fork` 调用返回，得到一个新的进程ID（PID），父进程也从 `fork` 调用返回，得到子进程的PID。
2. **exec:**
   - 用户态：进程调用 `exec` 系统调用，加载并执行新的程序。
   - 内核态：内核加载新程序的代码和数据，并进行一些必要的初始化。
   - 用户态：新程序开始执行，原来的程序替换为新程序。
3. **wait:**
   - 用户态：父进程调用 `wait` 或 `waitpid` 系统调用等待子进程的退出。
   - 内核态：如果子进程已经退出，内核返回子进程的退出状态给父进程；如果子进程尚未退出，父进程被阻塞，等待子进程退出。
   - 用户态：父进程得到子进程的退出状态，可以进行相应的处理。
4. **exit:**
   - 用户态：进程调用 `exit` 系统调用，通知内核准备退出。
   - 内核态：内核清理进程资源，包括释放内存、关闭文件等。
   - 用户态：进程退出，返回到父进程。

在这个过程中，用户态和内核态的切换主要发生在系统调用的边界上。当进程执行系统调用时，会从用户态切换到内核态，内核执行相应的操作，然后再切换回用户态，将执行结果返回给用户程序。例如，`fork`、`exec`、`wait`、`exit`等**系统调用都会引起用户态到内核态的切换。**

结果的返回通常是通过一些指定的寄存器或内存区域传递给用户程序。

3.2 问题2

```
          创建进程      +---wait()---------RUNNING----------------+
            |          |                    ^ |                   |
            |          |                    | |                   |
      alloc_page()     |                proc_run()            exit()
            |          |                    | |                   |
            V          |                    | v                   v
         UNINIT ------wakeup_proc()--> RUNNABLE --exit()--> ZOMBIE--父进程调用wait()-->
                       |                    ^                   ^
                       |                    |                   |
                       |                子进程调用exit()        exit()
                       |                    |                   |
                       |                    |                   |
                      +---wait()------>SLEEPING----------------+

```

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?

回答：由于我们只做了第二问，所以回答第二问的非编码问题。

- **用户程序被加载的时间**：本次实验中实际上通过make file中的make文件里面最后一步的ld命令加载的，hello 应用程序的执行码 obj/__user_hello.out 连接在了 ucore kernel 的末尾。并且通过两个全局变量记录了hello应用程序的起始位置和大小。**所以它是和和 ucore 内核一起被 bootloader 加载到内存里中的。**

  而对于一般的应用程序，会在需要时被加载到内存中。这个加载的过程通常是动态的，而不是一开始就将整个程序加载到内存中。

- **与常见操作系统加载的区别以及原因**：
  - **区别**：在常见的操作系统中应用程序并不是在系统启动时就被加载到内存中。相反，当用户需要运行某个应用程序时，操作系统才会将其加载到内存中。这种方式被称为**延迟加载**或**按需加载**。

延迟加载的优点是，它可以有效地管理系统资源，特别是内存。如果操作系统在启动时就将所有可能需要的应用程序都加载到内存中，那么很快就会耗尽内存资源。而通过延迟加载，操作系统可以确保只有真正需要运行的应用程序才会占用内存资源。

而本次实验中hello应用程序是和ucore一起被加载到内存中，而不是一种动态的加载方式，而是静态的在系统启动时就被加载到内存中。

- **原因：因为我们的hello应用程序是要紧跟着内核的第二个线程init_proc执行的，所以它其实在系统一启动就执行了。**而**不是后面通过调度选择它**来执行，由于我们本次实验不涉及到不同用户态应用程序的调度也没有实现，我们不能在后期动态加载这个程序，所以就和ucore内核一起在启动时就加载了。**方便启动时不使用调度而直接执行hello。**

  除此之外，在于苗学长讨论后，我们意识到另外一个可能出现问题的地方就在于ucore的内核空间实在是太小了，如果最开始将大部分的文件一口气上传，可能会导致ucore内核空间不足溢出等非常严重的问题。

三、与参考答案的对比

```
  -check result:                             OK
  -check output:                             OK
spin:                      (4.6s)
  -check result:                             OK
  -check output:                             OK
forktest:                  (1.3s)
  -check result:                             OK
  -check output:                             OK
Total Score: 130/130
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/riscv64-u
```

make grade通过，满分。

由于本实验的代码逻辑较为固定，因此我们小组所完成的代码与参考答案差别不大。

四、实验中的知识点

4.1 用户进程与内核进程

这一点上课老师介绍过，在pdf中也说明的很清楚，如果一两句话概括的话，**我觉得就是首先不能让所有的用户进程都对内核空间这么重要的地方进行操作**，所以要保护起来；另外也是我们出于实际考虑，考**虑到不是所有的程序员都能够做到对内核OS代码的合理运用（即“上帝”）**，即一些错误溢出问题可能会比较严重，因此有必要将内核空间保护起来。**由此诞生了分别在用户环境和内核环境运行的用户进程和内核线程。**