Lab 5 用户程序

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。

- 简要说明你的设计实现过程

```
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    //使用 kmalloc 分配了一个 struct proc_struct 类型大小的内存空间，返回值是一个指向新分配内存的指针 proc。
    if (proc != NULL) {
    proc->state = PROC_UNINIT;
    //初始化进程的状态 state 为 PROC_UNINIT，表示进程尚未初始化。
    proc->pid = -1;
        //表示还未分配有效的进程标识符。
    proc->runs = 0;
        //表示该进程尚未执行。
    proc->kstack = 0;
        //表示尚未分配内核栈
    proc->need_resched = 0;
        //表示当前进程不需要调度。
    proc->parent = NULL;
        //表示当前进程没有父进程。
    proc->mm = NULL;
        //表示该进程尚未分配内存管理结构。
    memset(&(proc->context), 0, sizeof(struct context));
        //使用 memset 将进程的 context（上下文）结构体清零。context 保存进程的 CPU 状态（如寄存器值、堆栈指针等）。
    proc->tf = NULL;
        //将进程的 tf（Trap Frame）指针初始化为 NULL
    proc->cr3 = boot_cr3;
    //将进程的 cr3 寄存器初始化为 boot_cr3。cr3 通常用于保存当前进程的页表基地址，boot_cr3 很可能是内核引导阶段使用的页表基地址。
    proc->flags = 0;
    memset(proc->name, 0, PROC_NAME_LEN);
    //使用 memset 将进程的 name（进程名）数组初始化为全零。PROC_NAME_LEN 是进程名的最大长度。

    //LAB5 YOUR CODE : (update LAB4 steps)
    proc->wait_state = 0;  // 进程等待状态初始化为0
    proc->cptr = proc->yptr = proc->optr = NULL;  // 进程间指针初始化为NULL
    }
    //初始化三个进程间指针 cptr（子进程指针）、yptr（兄弟进程指针）、optr（父进程指针）为 NULL。这些指针很可能用于表示进程之间的层级关系或链表结构。
    return proc;
    // 函数返回指向新创建并初始化的进程结构体的指针 proc
}
```

- 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. 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
2. 在`user_main`中通过宏`KERNEL_EXECVE`，调用`kernel_execve`
3. 在`kernel_execve`中执行`ebreak`，发生断点异常，转到`__alltraps`，转到`trap`，再到`trap_dispatch`，然后到`exception_handler`，最后到`CAUSE_BREAKPOINT`处
4. 在`CAUSE_BREAKPOINT`处调用`syscall`
5. 在`syscall`中根据参数，确定执行`sys_exec`，调用`do_execve`
6. 在`do_execve`中调用`load_icode`，加载文件
7. 加载完毕后一路返回，直到`__alltraps`的末尾，接着执行`__trapret`后的内容，到`sret`，表示退出S态，回到用户态执行，这时开始执行用户的应用程序

练习2 父进程复制自己的内存空间给子进程

> 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。

首先我们先查看copy_range函数的相关参数含义：

- `pde_t *to`：目标进程的页目录（Page Directory）。
- `pde_t *from`：源进程的页目录（Page Directory）。
- `uintptr_t start`：要复制的内存起始地址。
- `uintptr_t end`：要复制的内存结束地址。
- `bool share`：是否启用共享机制，若启用，则使用写时复制（COW）策略。

然后我们看非共享机制，即完整拷贝：

```
else {
    struct Page *npage = alloc_page();
    assert(npage != NULL);
    cprintf("alloc a new page 0x%x\n", page2kva(npage));
    void* src_kvaddr = page2kva(page);
    void* dst_kvaddr = page2kva(npage);
    memcpy(dst_kvaddr, src_kvaddr, PGSIZE);
    ret = page_insert(to, npage, start, perm);
}
```

没有启用共享机制，会为目标进程分配一个新的物理页面（`alloc_page`）。然后，将源进程物理页的内容复制到新分配的页面中。接着，将目标进程的虚拟地址映射到新的物理页面。

`memcpy`函数用来将源页面的内容复制到目标页面。复制操作完成后，新的物理页面将被插入到目标进程的页表中。

```
if(share) {
    cprintf("Sharing the page 0x%x\n", page2kva(page));
    page_insert(from, page, start, perm & ~PTE_W);
    ret = page_insert(to, page, start, perm & ~PTE_W);
}
```

如果启用了共享（写时复制，COW），则将源进程的页表项中的写权限移除（`perm & ~PTE_W`），使得源进程对该页面只读。然后将该页面插入到目标进程的页表中，映射到相同的物理页面，并保持只读权限。

这样，当目标进程尝试写入该页面时，会触发页面故障（page fault），然后可以在故障处理程序中将该页复制到目标进程的私有空间。

> 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。

写时复制（Copy on Write，COW）是一种内存优化技术，广泛应用于多进程系统中，尤其是用于进程的 **fork** 操作，以减少不必要的内存复制。当父进程和子进程都共享同一内存页面时，只有在进程试图修改这些页面时，才会为该进程分配新的内存页面,接下来我们简单说说其设计思想：

1. 进程创建与内存映射

在操作系统中，进程创建通常通过 `fork()` 系统调用来实现。当父进程调用 `fork()` 创建子进程时，操作系统会将父进程的页表复制给子进程，但是这并不意味着为子进程分配新的物理页面。子进程的页表将指向父进程的物理页面，并且通过写时复制机制（COW），两个进程共享相同的物理内存。

1.1 页表项设置为只读

- 在 `copy_range()` 中调用 `page_insert(from, page, start, perm & ~PTE_W)` 和 `page_insert(to, page, start, perm & ~PTE_W)`，通过将页表项的写权限去掉（即使用 `~PTE_W`），确保父子进程的对应页表项为只读。这样，如果子进程或父进程修改该页，就会触发页面错误（Page Fault）。
- 共享页面的引用计数被增加了，以确保当某个进程退出时，不会立即释放物理页面，只有当所有使用该页面的进程都退出时，页面才会被释放。

2. 写时复制（COW）

当进程试图修改一个共享的页面时，会触发 **页错误（Page Fault）**，因为该页面被标记为只读。操作系统会处理这个异常，分配一个新的页面并完成写时复制操作。

2.1 触发写时复制异常

- 当进程尝试写入一个只读页面时，会触发一个 `CAUSE_STORE_PAGE_FAULT` 异常，操作系统会将控制转到异常处理程序 `do_pgfault`。

2.2 异常处理：判断是否是写时复制

- 在 `do_pgfault` 中，首先会检查触发页面错误的页表项 `*ptep` 的有效性（即 `*ptep & PTE_V` 是否为真）。如果为真，说明该错误是由于写时复制导致的，那么可以继续进行写时复制的后续处理。

2.3 申请新的页面

- 操作系统会使用 `pgdir_alloc_page(mm->pgdir, addr, perm)` 为触发写时复制的进程申请一个新的页面，并为该页面建立虚拟地址到新物理页面的映射。
- 同时，更新该页表项的权限，将其设置为可写（`PTE_W`），确保进程可以修改新的物理页面。

2.4 复制原始页面内容

- 使用 `memcpy()` 将原始页面的内容复制到新的物理页面中。这样，新的页面包含了原页面的内容，但由于是私有页面，只有当前进程可以修改它。

2.5 处理原始页面

- `pgdir_alloc_page()` 会将原始页面的引用计数减一。如果该页面的引用计数变为 1，且触发了 `CAUSE_STORE_PAGE_FAULT` 异常，说明该页面是共享页面，且已被 COW 复制，剩余的所有进程现在拥有各自的私有副本。
- 在这种情况下，原始页面的权限将被更新为可写权限，这意味着原始页面不再是共享页面，而是一个私有副本。

3. 内存释放

当进程退出或者不再需要某个页面时，操作系统将清空该进程的页表项，并减少该物理页面的引用计数。然而，物理页面不会立即被释放，只有当物理页面的引用计数变为 0 时，才会真正释放该页面。

3.1 页表项清除

- 在进程退出时，操作系统会清空进程页表中对应页面的映射，并减少该物理页面的引用计数。

3.2 引用计数变为 0

- 当一个页面的引用计数变为 0，即没有任何进程再使用这个页面时，操作系统会释放该页面。释放操作通常包括返回该页面到内存池中，以便后续重新分配。

我们最终运行结果如下：

```
  -check output:                          OK
testbss:                (1.3s)
  -check result:                          OK
  -check output:                          OK
pgdir:                  (1.1s)
  -check result:                          OK
  -check output:                          OK
yield:                  (1.2s)
  -check result:                          OK
  -check output:                          OK
badarg:                 (1.2s)
  -check result:                          OK
  -check output:                          OK
exit:                   (1.4s)
  -check result:                          OK
  -check output:                          OK
spin:                   (5.7s)
  -check result:                          OK
  -check output:                          OK
forktest:               (1.3s)
  -check result:                          OK
  -check output:                          OK
Total Score: 130/130
wz@wz-virtual-machine:~/riscv64-ucore-labcodes/lab5$
```

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）

fork/exec/wait/exit的执行流程

fork

用户态： `fork()` -> `sys_fork()` -> `syscall(SYS_fork)` -> `ecall` （产生一个`trap`） -> 内核态

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

在`do_fork`函数中，首先检查是否允许创建新进程（如最大进程数限制），接着调用`alloc_proc`分配并初始化子进程结构，再调用`setup_kstack`为子进程分配内核栈，接下来调用`copy_mm`根据`clone_flags`复制或共享父进程的内存空间，接着调用`copy_thread`复制父进程的上下文信息和中断帧给子进程，然后给子进程分配唯一的进程`ID`，并将子进程添加到`hash_proc`和`proc_list`中，设置其状态为`PROC_RUNNABLE`（可运行），最后返回子进程的`PID`。

exec

内核态：`kernel_execve()` -> `ebreak` -> `syscall()` -> `sys_exec()` -> `do_execve(name, len, binary, size)`

在`do_execve`函数中，首先验证用户传入的程序名和长度是否合法，合法则将用户传入的进程名`name`复制到内核栈上的`local_name`中。接着检查`mm`是否非空，如果非空，说明需要清理现有的用户地址空间（具体操作包括切换到内核页表，清理虚拟内存映射，释放页表的物理内存，销毁`mm`），接下来调用`load_icode`函数，将传入的二进制文件`binary`加载到内存，并设置新的入口点，然后调用`set_proc_name`更新当前进程的名称为`local_name`，最后返回。

wait

用户态： `wait()` -> `sys_wait()` -> `syscall(SYS_wait)` -> `ecall` （产生一个`trap`）-> 内核态

内核态： `syscall()` -> `sys_wait()` -> `do_wait(pid, store)`

在`do_wait`函数中，首先验证`code_store`指针是否指向父进程的合法用户态内存，是则查看给定`PID`是否为`0`。若不为`0`，通过`find_proc`函数查找指定`PID`的进程，如果找到，并且目标进程是当前进程的直接子进程，设置`haskid`标志为`1`，如果子进程已进入僵尸状态（`PROC_ZOMBIE`），跳转到`found`。若给定`PID`为`0`，遍历当前进程的子进程链表，查找任意一个进入僵尸状态的子进程，找到就跳转到`found`。

如果父进程有子进程，但没有僵尸子进程，将父进程的状态设置为`PROC_SLEEPING`，等待状态设置为`WT_CHILD`，调用`schedule`暂停父进程，切换到其他可运行进程，如果父进程被标记为退出（`PF_EXITING`），调用`do_exit`强制退出。

如果父进程有僵尸子进程，跳转`found`后，先检查子进程是不是`idleproc`或`initproc`，是则触发`panic`。再检查`code_store`是否非空，非空则将子进程的退出码写入父进程提供的内存地址。最后清理僵尸子进程的所有资源并返回。

exit

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

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

在`do_exit`函数中，首先检查当前进程是不是`idleproc`或`initproc`，是则触发`panic`。接着检查当前进程的`mm`是否非空，如果非空，说明当前进程分配了用户态内存资源，需要释放（具体操作包括切换到内核页表，释放进程映射的虚拟内存区域，释放页目录，销毁`mm`）。接下来将当前进程的状态设置为`PROC_ZOMBIE`，并将`error_code`保存到进程的`exit_code`字段中，以供父进程查询。然后使用`current->parent`找到父进程。如果父进程的等待状态是`WT_CHILD`，调用`wakeup_proc`唤醒父进程。接着遍历当前进程的子进程链表，将子进程添加到`initproc`的子进程链表中，如果被移交的子进程已经是僵尸状态，且`initproc`正在等待子进程退出，则唤醒`initproc`。最后调用`schedule`切换到其他可运行的进程。一般来说，`do_exit`函数不会返回。

内核态与用户态程序如何交错执行

fork

进入内核态：用户程序调用`fork()`，通过系统调用进入内核。

内核分配进程结构，复制内存，返回子进程 PID。

返回用户态：子进程和父进程分别在各自上下文中继续执行，从`fork`返回。

exec

进入内核态：用户程序调用`exec()`，通过系统调用进入内核。

内核清理内存，加载新程序。

返回用户态：不返回旧程序，直接切换到新程序的入口点，运行新程序的代码。

wait

进入内核态：用户程序调用`wait()`，通过系统调用进入内核。

内核检查子进程状态，阻塞等待，回收资源。

返回用户态：子进程退出后，内核将退出状态返回给用户态的父进程。

exit

进入内核态：用户程序调用`exit()`，通过系统调用进入内核。

内核释放资源，通知父进程，进入僵尸状态。

不返回用户态：进程退出后直接切换到其他进程。

内核结果返回机制

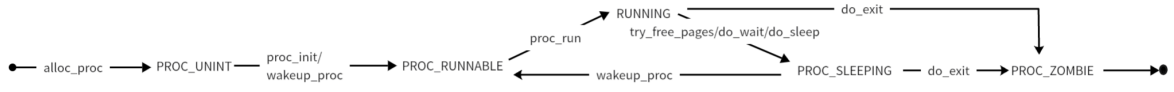
1. **寄存器返回**：

   系统调用完成后，内核将返回值（如 `fork` 的子进程 PID ）存入通用寄存器，然后返回用户态。

2. **用户地址空间写入**：

   对于一些需要返回大量数据的调用（如 `wait` 的退出状态），内核会直接写入用户地址空间。

ucore中一个用户态进程的执行状态生命周期图



扩展练习 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？

分析Lab5的Makefile，可以发现，在构建内核目标（`kernel`）时，用户程序（`USER_BINS`）会被打包到内核中，以一种特殊的方式嵌入到内核二进制中，并定义好了起始位置和大小，最后内核和嵌入的用户程序被打包到镜像文件`ucore.img`中，一起被bootloader加载到物理内存中。具体来说，在`user_main()`函数中`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。

而在常用的操作系统中，用户程序通常存储在文件系统中，操作系统启动时并不会将用户程序加载到内存中。只有当用户执行程序时，操作系统通过系统调用（如`execve`）按需从文件系统中加载用户程序。

产生这种区别的原因可能是，ucore在仿真器上运行，不涉及复杂的文件系统和动态加载机制，无法在运行时从外部存储加载用户程序，用户程序需要通过内核直接管理和调度，与内核同时加载是一种适应性选择。并且ucore的内存和存储资源有限，直接将用户程序与内核打包可以减少系统复杂度。

实验中的知识点

系统调用（system call)

操作系统应当提供给用户程序一些接口，让用户程序使用操作系统提供的服务。这些接口就是**系统调用**。用户程序在用户态运行(U mode), 系统调用在内核态执行(S mode)。这里有一个CPU的特权级切换的过程, 要用到`ecall`指令从U mode进入S mode。用`ecall`从U mode进入S mode之后，对应的处理需要内核系统调用的代码来完成。

对于用户进程的管理，有四个系统调用比较重要。

`sys_fork()`：把当前的进程复制一份，创建一个子进程，原先的进程是父进程。接下来两个进程都会收到`sys_fork()`的返回值，如果返回0说明当前位于子进程中，返回一个非0的值（子进程的PID）说明当前位于父进程中。然后就可以根据返回值的不同，在两个进程里进行不同的处理。

`sys_exec()`：在当前的进程下，停止原先正在运行的程序，开始执行一个新程序。PID不变，但是内存空间要重新分配，执行的机器代码发生了改变。我们可以用`fork()`和`exec()`配合，在当前程序不停止的情况下，开始执行另一个程序。

`sys_exit()`：退出当前的进程。

`sys_wait()`：挂起当前的进程，等到特定条件满足的时候再继续执行。