lab5 实验报告

练习0：填写已有实验

对已完成的实验代码进行进一步改进。

在`alloc_proc`中添加额外的初始化：

```
proc->wait_state = 0;
proc->cptr = NULL; // 表示当前进程的子进程
proc->optr = NULL; // 表示当前进程的上一个兄弟进程
proc->yptr = NULL; // 表示当前进程的下一个兄弟进程
```

在do_fork中修改代码如下：

```
if((proc = alloc_proc()) == NULL)
{
    goto fork_out;
}
proc->parent = current;
assert(current->wait_state == 0);
if(setup_kstack(proc) != 0)
{
    goto bad_fork_cleanup_proc;
}
;
if(copy_mm(clone_flags, proc) != 0)
{
    goto bad_fork_cleanup_kstack;
}
copy_thread(proc, stack, tf);
bool intr_flag;
local_intr_save(intr_flag);
{
    int pid = get_pid();
    proc->pid = pid;
    hash_proc(proc);
    set_links(proc);
}
local_intr_restore(intr_flag);
wakeup_proc(proc);
ret = proc->pid;
```

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。

- 按照指导书与代码注释中的提示
  - 设置`sp`为用户栈栈顶
  - 设置`epc`为用户进程的入口地址
  - 将`sstatus`的`SPP`位置零，说明异常来自用户态，处理完成后需要返回用户态；`SPIE`位置零，表示不启用中断

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

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`中调用了`kernel_thread`其中调用`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`，开始运行。
2. `user_main`通过宏`KERNEL_EXECVE`，调用`kernel_execve`。
3. `kernel_execve`中因为此时处于特权态，通过将`a7`寄存器置为10并调用`ebreak`发生断点，告诉`ucore`这个中断需要转发给`syscall()`。断点异常会转到`__alltraps`，转到`trap`，再到`trap_dispatch`，然后到`exception_handler`，最后到`CAUSE_BREAKPOINT`处。
4. `CAUSE_BREAKPOINT`中会调用`syscall`，根据参数进行系统调用，这里会执行`sys_exec`，其中调用`do_execve`。
5. `do_execve`中调用`load_icode`，把新的程序加载到当前进程里。
6. 加载完毕后一路返回，到`__alltraps`后会执行之后的代码`sret`，退出S态回到U态继续执行，之后就开始执行用户应用程序。

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。**

调用关系：`do_fork()---->copy_mm()创建mm---->dup_mmap()复制---->copy_range()`，

dup_mmap遍历父进程，创建子进程后调用copy_range实现复制。

copy_range遍历父进程的指定内存空间的虚拟页，虚拟页存在就为子进程对应的同一个地址分配物理页（页目录表不同），将父进程的内容复制到子进程，并为子进程的物理页和虚拟地址建立映射。

```
uintptr_t* src_kvaddr = page2kva(page);//获取源页面的内核虚拟地址
uintptr_t* dst_kvaddr = page2kva(npage);//获取目标页面的内核虚拟地址
memcpy(dst_kvaddr, *src_kvaddr, PGSIZE);//将源页面的内容复制到目标页面
ret = page_insert(to, npage, start, perm);//将目标页的物理地址与目标进程的线性地址（start）映射
```

`Copy on Write`机制：

COW机制是一种延迟复制策略，在fork系统调用实现复制是，它允许多个进程暂时共享同一块内存页，对内存页有只读操作，直到其中一个进程对该内存页进行修改时，操作系统会创建该内存页的副本，让进行写操作的进程对副本的操作是其他进程不可见的，避免不必要的内存复制。

在do_fork函数中，COW 的核心实现是在 `copy_mm` 函数中进行的，`copy_mm` 会标记所有内存页为只读（通常是通过设置页表的权限位）。此时，父子进程共享同一内存页，当进程尝试修改这些共享页时，CPU 会触发页错误（page fault），操作系统的内存管理单元会捕获该错误，并通过 `copy_on_write` 机制将该内存页复制到进程的私有空间，从而保证每个进程都有自己的内存副本。

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

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（包括执行状态，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

用户态：fork() -> sys_fork() -> syscall(SYS_fork) -> ecall -> 内核态

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

do_fork中，调用 alloc_proc 分配一个 proc_struct，并设置父进程。调用 setup_kstack 为 子进程分配一个内核栈，调用 copy_mm 根据 clone_flag 复制或共享 mm，调用 copy_thread 在 proc_struct 中设置 tf 和上下文，将 proc_struct 插入 hash_list 和 proc_list，调用 wakeup_proc 使新的子进程变为可运行状态，使用子进程的 pid 设置返回值。

2、exec

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

检查用户提供的程序名称是否合法。 如果当前进程的 mm 不为空，说明当前进程占用了内存，进行相关清理操作，包括切换到内核页表、释 放进程的内存映射、释放页目录表、销毁进程的内存管理结构等。调用 load_icode 函数加载用户提供的二进制文件，将其代码段加载到内存中。 使用 set_proc_name 函数设置进程的名称。

3、wait

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

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

首先进行内存检查，确保 code_store 指向的内存区域可访问。遍历查找具有给定PID的子进程，若找 到且该子进程的父进程是当前进程，将 haskid 标志设置为1。如果 pid 为零，将循环遍历所有子进程，查找已经退出的子进程。如果找到，跳转到标签 found。如果存在子进程，将当前进程的状态设置为 PROC_SLEEPING，等待状态设置为 WT_CHILD，然后调用调度器 schedule() 来选择新的可运行进 程。如果当前进程被标记为PF_EXITING，则调用 do_exit 以处理退出,跳转到标签 repeat 继续执行。 找到后检查子进程是否是空闲进程 idleproc或初始化进程 initproc，如果是则触发 panic。存储子 进程的退出状态，处理子进程退出并释放资源。

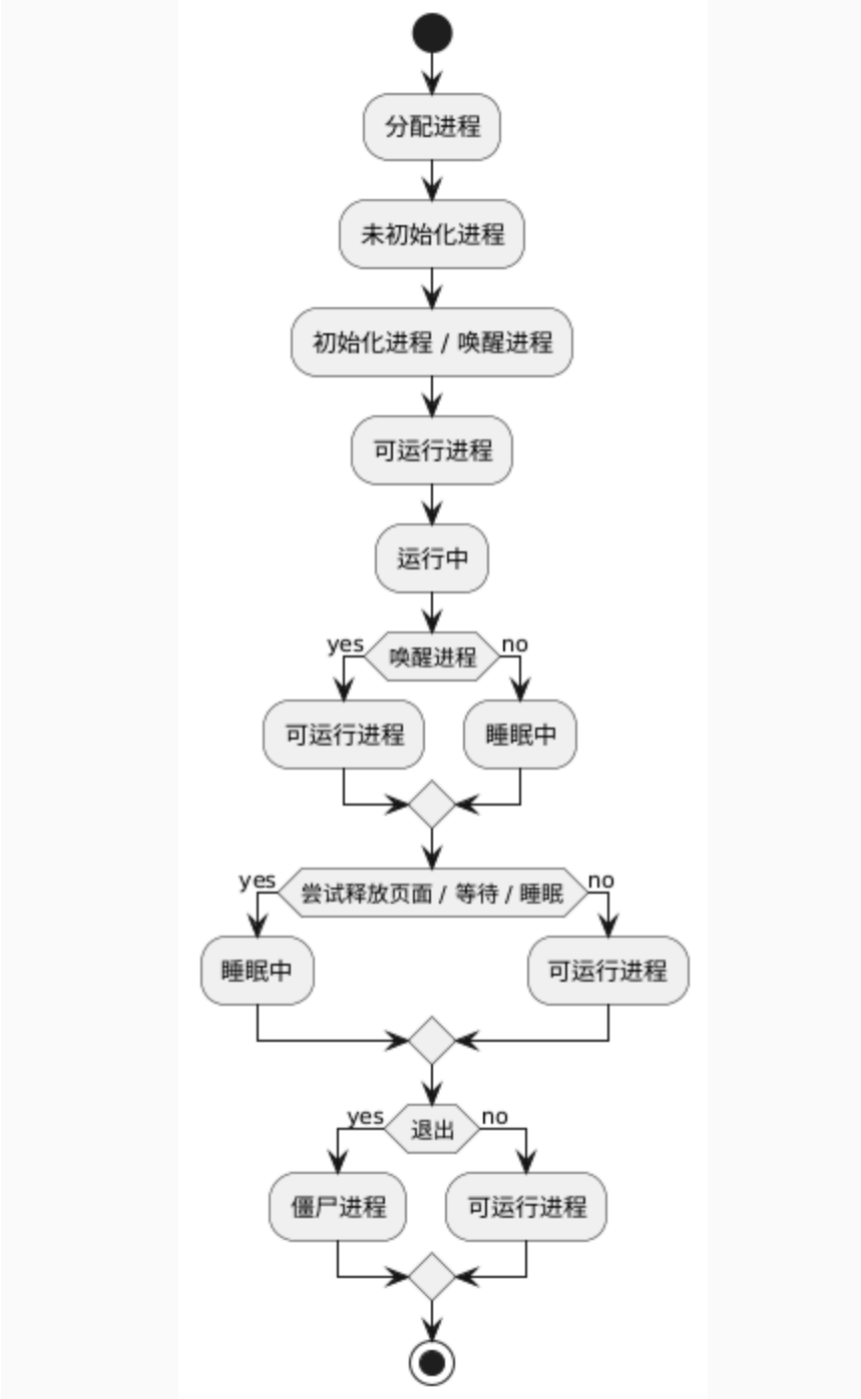
4、exit

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

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

检查当前进程是否是idleproc或initproc，若是则 panic 。获取内存管理结构，减少对内存管理结构的引用计数，如果引用计数降为零，代表没有其他进程共享该内存管理结构，那么清理映射并释放页目录表，最后销毁内存管理结构。最后，将当前进程的 mm 指针设置为 NULL 。将进程的状态设置为 PROC_ZOMBIE ，表示进程已经退出。如果父进程正在等待子进程退出，则唤醒当前进程的父进程。然后，通过循环处理当前进程的所有子进程，将它们的状态设置为 PROC_ZOMBIE ，并将其重新连接到初始 化进程的子进程链表上。如果初始化进程也正在等待子进程退出，那么也唤醒初始化进程。最后进行调度。

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？

在本次实验中，用户程序在编译时被链接到内核中，并定义好了起始位置和大小，然后在 user_main() 函数 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。实现了通过一个内核进程直接将整段文件直接加载内存中。

而在我们常用的操作系统中，用户程序通常是存储在外部存储设备上的独立文件。当需要执行某个程序时，操作系统会从磁盘等存储介质上动态地加载这个程序到内存中。

这里我们之所以采用这种加载方式的原因是 ucore 没实现硬盘和文件系统，出于简化和教学性质的考虑，将用户程序编译到内核中减少了实现的复杂度。