**Lab5 用户进程管理**

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，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。 请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU是如何让这个应用程序最终在用户态执行起来的。即 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调用mm\_create函数来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；

②调用setup\_pgdir来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（boot\_pgdir所指）的内容拷贝到此新目录表中，最后让mm->pgdir指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间；

③根据应用程序执行码的起始位置来解析此ELF格式的执行程序，并调用mm\_map函数根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把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；

④调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，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，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；

⑤需要给用户进程设置用户栈，为此调用mm\_mmap函数建立用户栈的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<–>物理地址映射关系；

⑥至此,进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被程序的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；

⑦先清空进程的中断帧，再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；

至此，用户进程的用户环境已经搭建完毕。此时initproc将按产生系统调用的函数调用路径原路返回，执行中断返回指令“iret”（位于trapentry.S的最后一句）后，将切换到用户进程hello的第一条语句位置\_start处（位于user/libs/initcode.S的第三句）开始执行。

#define KSTACKPAGE 2 // # of pages in kernel stack

#define KSTACKSIZE (KSTACKPAGE \* PGSIZE) // sizeof kernel stack

#define USERTOP 0xB0000000

#define USTACKTOP USERTOP

#define USTACKPAGE 256 // # of pages in user stack

#define USTACKSIZE (USTACKPAGE \* PGSIZE) // sizeof user stack

#define USERBASE 0x00200000

#define UTEXT 0x00800000 // where user programs generally begin

#define USTAB USERBASE

/\* LAB5:EXERCISE1 YOUR CODE

\* should set tf\_cs,tf\_ds,tf\_es,tf\_ss,tf\_esp,tf\_eip,tf\_eflags

\* NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel. So

\* tf\_cs should be USER\_CS segment (see memlayout.h)

\* tf\_ds=tf\_es=tf\_ss should be USER\_DS segment

\* tf\_esp should be the top addr of user stack (USTACKTOP)

\* tf\_eip should be the entry point of this binary program (elf->e\_entry)

\* tf\_eflags should be set to enable computer to produce Interrupt

\*/

tf->tf\_cs = USER\_CS;

tf->tf\_ds = tf->tf\_es = tf->tf\_ss = USER\_DS;

tf->tf\_esp = USTACKTOP;

tf->tf\_eip = elf->e\_entry;

tf->tf\_eflags = FL\_IF;//FL\_IF为中断打开状态

ret = 0;

out:

return ret;

bad\_cleanup\_mmap:

exit\_mmap(mm);

bad\_elf\_cleanup\_pgdir:

put\_pgdir(mm);

bad\_pgdir\_cleanup\_mm:

mm\_destroy(mm);

bad\_mm:

goto out;

}

vectors.S::vector128起始处:

pushl $0

pushl $128

......

trapentry.S::\_\_alltraps起始处:

pushl %ds

pushl %es

pushal

……

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。

do\_fork()---->copy\_mm()---->dup\_mmap()---->copy\_range()

int copy\_range(pde\_t \*to, pde\_t \*from, uintptr\_t start, uintptr\_t end, bool share) {

......

......

void \* kva\_src = page2kva(page);//返回父进程的内核虚拟页地址

void \* kva\_dst = page2kva(npage);//返回子进程的内核虚拟页地址

memcpy(kva\_dst, kva\_src, PGSIZE);//复制父进程到子进程

ret = page\_insert(to, npage, start, perm);//建立子进程页地址起始位置与物理地址的映射关系(prem是权限)

......

......

}

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在实现中是如何影响进程的执行状态的？

fork

首先当程序执行fork时，fork使用了系统调用SYS\_fork,而系统调用SYS\_fork则主要是由do\_fork和wakeup\_proc来完成的。

1、分配并初始化进程控制块(alloc\_proc 函数);

2、分配并初始化内核栈(setup\_stack 函数);

3、根据 clone\_flag标志复制或共享进程内存管理结构(copy\_mm 函数);

4、设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧和执行上下文(copy\_thread 函数);

5、把设置好的进程控制块放入hash\_list 和 proc\_list 两个全局进程链表中;

6、自此,进程已经准备好执行了,把进程状态设置为“就绪”态;

7、设置返回码为子进程的 id 号。

而wakeup\_proc函数主要是将进程的状态设置为等待，即proc->wait\_state = 0，

exec

当应用程序执行的时候，会调用SYS\_exec系统调用,而当ucore收到此系统调用的时候，则会使用do\_execve()函数来实现，因此这里我们主要介绍do\_execve()函数的功能，函数主要时完成用户进程的创建工作，同时使用户进程进入执行。

主要工作如下：

1、首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。

2、接下来是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。之后就是调用load\_icode从而使之准备好执行。

wait

  当执行wait功能的时候，会调用系统调用SYS\_wait，而该系统调用的功能则主要由do\_wait函数实现，完成对子进程的最后回收工作，即回收子进程的内核栈和进程控制块所占内存空间。

  具体的功能实现如下：

1、 如果 pid!=0，表示只找一个进程 id 号为 pid 的退出状态的子进程，否则找任意一个处于退出状态的子进程;

2、 如果此子进程的执行状态不为PROC\_ZOMBIE，表明此子进程还没有退出，则当前进程设置执行状态为PROC\_SLEEPING（睡眠），睡眠原因为WT\_CHILD(即等待子进程退出)，调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤 1 处执行;

3、 如果此子进程的执行状态为 PROC\_ZOMBIE，表明此子进程处于退出状态，需要当前进程(即子进程的父进程)完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc\_list和hash\_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。

exit

当执行exit功能的时候，会调用系统调用SYS\_exit，而该系统调用的功能主要是由do\_exit函数实现。具体过程如下：

1、先判断是否是用户进程，如果是，则开始回收此用户进程所占用的用户态虚拟内存空间;（具体的回收过程不作详细说明）

2、设置当前进程的中hi性状态为PROC\_ZOMBIE，然后设置当前进程的退出码为error\_code。表明此时这个进程已经无法再被调度了，只能等待父进程来完成最后的回收工作（主要是回收该子进程的内核栈、进程控制块）

3、如果当前父进程已经处于等待子进程的状态，即父进程的wait\_state被置为WT\_CHILD，则此时就可以唤醒父进程，让父进程来帮子进程完成最后的资源回收工作。

4、如果当前进程还有子进程,则需要把这些子进程的父进程指针设置为内核线程init,且各个子进程指针需要插入到init的子进程链表中。如果某个子进程的执行状态是 PROC\_ZOMBIE,则需要唤醒 init来完成对此子进程的最后回收工作。

5、执行schedule()调度函数，选择新的进程执行。

所以说该函数的功能简单的说就是，回收当前进程所占的大部分内存资源,并通知父进程完成最后的回收工作。

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） 执行：make grade。如果所显示的应用程序检测都输出ok，则基本正确。（使用的是qemu-1.0.1）

