练习1: 加载应用程序并执行（需要编码）

\* 实现思路：

1. 由于最终是在用户态下运行的，所以需要将段寄存器初始化为用户态的代码段、数据段、堆栈段；

2. esp 应当指向先前的步骤中创建的用户栈的栈顶；

3. eip 应当指向 ELF 可执行文件加载到内存之后的入口处；

4. eflags 中应当初始化为中断使能，注意 eflags 的第 1 位是恒为 1 的；

5. 设置 ret 为 0，表示正常返回；

练习2: 父进程复制自己的内存空间给子进程（需要编码）

\* 实现思路：

copy\_range 函数的具体执行流程是遍历父进程指定的某一段内存空间中的每一个虚拟页，如果这个虚拟页是存在的话，为子进程对应的同一个地址（但是页目录表是不一样的，因此不是一个内存空间）也申请分配一个物理页，然后将前者中的所有内容复制到后者中去，然后为子进程的这个物理页和对应的虚拟地址（事实上是线性地址）建立映射关系；而在本练习中需要完成的内容就是内存的复制和映射的建立，具体流程如下：

1. 找到父进程指定的某一物理页对应的内核虚拟地址；

2. 找到需要拷贝过去的子进程的对应物理页对应的内核虚拟地址；

3. 将前者的内容拷贝到后者中去；

4. 为子进程当前分配这一物理页映射上对应的在子进程虚拟地址空间里的一个虚拟页；

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）

请分析fork/exec/wait/exit在实现中是如何影响进程的执行状态的？

**fork**

调用过程为：fork->SYS\_fork->do\_fork+wakeup\_proc

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。do\_fork() 完成的工作在练习 2 及 lab4 中已经做过详细介绍，这里再简单说一下，主要是完成了以下工作：

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，为内核进程（线程）建立栈空间（ 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、为进程分配一个 PID（ get\_pid() 函数）;

6、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;

7、自此，进程已经准备好执行了，把进程状态设置为“就绪”态;

8、设置返回码为子进程的 PID 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**

调用过程为：SYS\_exec->do\_execve

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。（具体 load\_icode 的功能在练习 1 已经介绍的很详细了，这里不赘述了）

**wait**

调用过程为：SYS\_wait->do\_wait

我们可以看看 do\_wait 函数的实现过程：

/\*

file\_path:kern/process/proc.c

\*/

// do\_wait - wait one OR any children with PROC\_ZOMBIE state, and free memory space of kernel stack

// - proc struct of this child.

// NOTE: only after do\_wait function, all resources of the child proces are free.

int do\_wait(int pid, int \*code\_store) {

struct mm\_struct \*mm = current->mm;

if (code\_store != NULL) {

if (!user\_mem\_check(mm, (uintptr\_t)code\_store, sizeof(int), 1)) {

return -E\_INVAL;

}

}

struct proc\_struct \*proc;

bool intr\_flag, haskid;

repeat:

haskid = 0;

//如果pid！=0，则找到进程id为pid的处于退出状态的子进程

if (pid != 0) {

proc = find\_proc(pid);

if (proc != NULL && proc->parent == current) {

haskid = 1;

if (proc->state == PROC\_ZOMBIE) {

goto found; //找到进程

}

}

}

else {

//如果pid==0，则随意找一个处于退出状态的子进程

proc = current->cptr;

for (; proc != NULL; proc = proc->optr) {

haskid = 1;

if (proc->state == PROC\_ZOMBIE) {

goto found;

}

}

}

if (haskid) {//如果没找到，则父进程重新进入睡眠，并重复寻找的过程

current->state = PROC\_SLEEPING;

current->wait\_state = WT\_CHILD;

schedule();

if (current->flags & PF\_EXITING) {

do\_exit(-E\_KILLED);

}

goto repeat;

}

return -E\_BAD\_PROC;

//释放子进程的所有资源

found:

if (proc == idleproc || proc == initproc) {

panic("wait idleproc or initproc.\n");

}

if (code\_store != NULL) {

\*code\_store = proc->exit\_code;

}

local\_intr\_save(intr\_flag);

{

unhash\_proc(proc);//将子进程从hash\_list中删除

remove\_links(proc);//将子进程从proc\_list中删除

}

local\_intr\_restore(intr\_flag);

put\_kstack(proc); //释放子进程的内核堆栈

kfree(proc); //释放子进程的进程控制块

return 0;

}

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，主要工作就是父进程如何完成对子进程的最后回收工作，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**

调用过程为：SYS\_exit->exit

我们可以看看 do\_exit 函数的实现过程：

/\*

file\_path:kern/process/proc.c

\*/

// do\_exit - called by sys\_exit

// 1. call exit\_mmap & put\_pgdir & mm\_destroy to free the almost all memory space of process

// 2. set process' state as PROC\_ZOMBIE, then call wakeup\_proc(parent) to ask parent reclaim itself.

// 3. call scheduler to switch to other process

int do\_exit(int error\_code) {

if (current == idleproc) {

panic("idleproc exit.\n");

}

if (current == initproc) {

panic("initproc exit.\n");

}

struct mm\_struct \*mm = current->mm;

if (mm != NULL) { //如果该进程是用户进程

lcr3(boot\_cr3); //切换到内核态的页表

if (mm\_count\_dec(mm) == 0){

exit\_mmap(mm);

/\*如果没有其他进程共享这个内存释放current->mm->vma链表中每个vma描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后还把页表所占用的空间释放并把对应的页目录表项清空\*/

put\_pgdir(mm); //释放页目录占用的内存

mm\_destroy(mm); //释放mm占用的内存

}

current->mm = NULL; //虚拟内存空间回收完毕

}

current->state = PROC\_ZOMBIE; //僵死状态

current->exit\_code = error\_code;//等待父进程做最后的回收

bool intr\_flag;

struct proc\_struct \*proc;

local\_intr\_save(intr\_flag);

{

proc = current->parent;

if (proc->wait\_state == WT\_CHILD) {

wakeup\_proc(proc); //如果父进程在等待子进程，则唤醒

}

while (current->cptr != NULL) {

/\*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initproc，且各个子进程指针需要插入到initproc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initproc来完成对此子进程的最后回收工作。\*/

proc = current->cptr;

current->cptr = proc->optr;

proc->yptr = NULL;

if ((proc->optr = initproc->cptr) != NULL) {

initproc->cptr->yptr = proc;

}

proc->parent = initproc;

initproc->cptr = proc;

if (proc->state == PROC\_ZOMBIE) {

if (initproc->wait\_state == WT\_CHILD) {

wakeup\_proc(initproc);

}

}

}

}

local\_intr\_restore(intr\_flag);

schedule(); //选择新的进程执行

panic("do\_exit will not return!! %d.\n", current->pid);

}

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。

请分析 fork/exec/wait/exit 在实现中是如何影响进程的执行状态的？

fork 执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID。我们可以通过 fork 返回的值来判断当前进程是子进程还是父进程。fork 不会影响当前进程的执行状态，但是会将子进程的状态标记为 RUNNALB，使得可以在后续的调度中运行起来；

exec 完成用户进程的创建工作。首先为加载新的执行码做好用户态内存空间清空准备。接下来的一步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。exec 不会影响当前进程的执行状态，但是会修改当前进程中执行的程序；

wait 是等待任意子进程的结束通知。wait\_pid 函数等待进程 id 号为 pid 的子进程结束通知。这两个函数最终访问 sys\_wait 系统调用接口让 ucore 来完成对子进程的最后回收工作。wait 系统调用取决于是否存在可以释放资源（ZOMBIE）的子进程，如果有的话不会发生状态的改变，如果没有的话会将当前进程置为 SLEEPING 态，等待执行了 exit 的子进程将其唤醒；

exit 会把一个退出码 error\_code 传递给 ucore，ucore 通过执行内核函数 do\_exit 来完成对当前进程的退出处理，主要工作简单地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。exit 会将当前进程的状态修改为 ZOMBIE 态，并且会将父进程唤醒（修改为RUNNABLE），然后主动让出 CPU 使用权；

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）

流程图如下：

