

**本科生实验报告**

实验课程：操作系统原理实验

任课教师：刘宁

实验题目：从内核态到用户态

专业名称：计算机科学与技术

学生姓名：

学生学号：

实验地点：实验中心B202

实验时间：2025.6.10

**Section 1 实验概述**

在本次实验中，首先会学习并了解讨论保护模式下的特权级的相关内容。特权级保护是保护模式的特点之一，通过特权级保护，区分了内核态和用户态，从而限制用户态的代码对特权指令的使用或对资源的访问。但是，用户态的代码有时不得不使用一些特权指令，如输入输出等。因此，这就引申出了系统调用的概念和如何通过中断来实现系统调用。通过系统调用，可以实现从用户态到内核态转移，然后在内核态下执行特权指令等，执行完成后返回到用户态。

在实现了系统调用后，本次实验通过创建进程的三步来实现了进程的创建。并通过分页机制来实现进程之间的虚拟地址空间的隔离。

最后本次实验，还实现了fork/wait/exit的这些系统调用的实现.

**Section 2 预备知识与实验环境**

* 预备知识：x86汇编语言、IA-32处理器体系结构、操作系统中断处理、进程管理、保护模式特权级
* 实验环境：
  + 虚拟机版本/处理器型号：Virtualbox7.0.6、Ubuntu18.04
  + 代码编辑环境：gedit文本编辑器、gdb调试器、qemu
  + 代码编译工具：nasm

**Section 3 实验任务**

* **实验任务1：实现系统调用的方法**

1. 请解释为什么需要使用寄存器来传递系统调用的参数，以及是如何在执行int 0x80前在栈中找到参数并放入寄存器的

2. 分析在我们调用了 int 0x80 后：系统的栈发生了怎样的变化? esp 的值和在setup.cpp 中定义的变量 tss 有什么关系?此外还有哪些段寄存器发生了变化? 变化后的内容是什么?

3. 在进入 asm\_system\_call\_handler 的那一刻：栈顶的地址是什么? 栈中存放的内容是什么? 为什么存放的是这些内容?

4. 分析 asm\_system\_call\_handler 是如何找到中断向量号 index 对应的函数的

5. 分析在 asm\_system\_call handler 中执行 iret 后：哪些段寄存器发生了变化? 变化后的内容是什么? 这些内容来自于什么地方?

* **实验任务2：进程的创建和调度**

1. 分析如何在线程的基础上创建进程的PCB的(即分析进程创建的三个步骤)

2. 分析为什么 asm\_switch\_thread 在执行 iret 后会跳转到 load process

3. 请结合代码逻辑和gdb来分析是如何设置 ProcessStartStack 的内容，从而使得我们能够在 asm\_start\_process 中实现内核态到用户态的转移，即从特权级0转移到特权级3下，并使用 iret 指令成功启动进程的

4. 结合代码，分析在创建进程后，我们对 ProgramManager::schedule 作了哪些修改? 这样做的目的是什么?

5. 在进程的创建过程中，我们存在如下语句 :

int ProgramManager::executeProcess(const char \*filename, int priority)

{

...

//找到刚刚创建的PCB

PCB \*process =ListItem2PCB(allPrograms.back()， tagInAllList);

...

}

正如教程中所提到， “...但是，这样做是存在风险的，我们应该通过pid来找到刚刚创建的PCB...”

编写一个 ProgramManager 的成员函数 findProgramByPid :

PCB \*findProgramByPid(int pid);

并用上面这个函数替换指导书中提到的"存在风险的语句"，替换结果如下:

int ProgramManager::executeProcess(const char \*filename, int priority)

{ .

..

//找到刚刚创建的PCB

PCB \*process =findProgramByPid(pid);

...

}

* **实验任务3：fork的实现**

1. 请根据代码逻辑概括 fork 的实现的基本思路，分析如何解决"四个关键问题“的

2. 在 asm\_switch\_thread 切换到子进程的栈中时， esp 的地址是什么? 栈中保存的内容是什么?

3. 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从 fork 返回，根据gdb来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完 ProgramManager::fork 后的返回过程的异同

4. 请根据代码逻辑和gdb来解释子进程的 fork 返回值为什么是0，而父进程的 fork 返回值是子进程的pid

5. 请解释在 Programanager::schedule 中，是如何从一个进程的虚拟地址空间切换到另外一个进程的虚拟地址空间的

* **实验任务4：wait & exit 的实现**

1. 请结合代码逻辑和具体的实例来分析 exit 的执行过程。

2. 请解释进程退出后能够隐式调用 exit 的原因(从栈的角度分析)

3. 请结合代码逻辑和具体的实例来分析 wait 的执行过程。

4. 请对代码做出修改，实现回收僵尸进程的有效方法

**Section 4 实验步骤与实验结果**

------------------------- **实验任务1** -------------------------

* 任务要求：

1. 请解释为什么需要使用寄存器来传递系统调用的参数，以及是如何在执行int 0x80前在栈中找到参数并放入寄存器的

2. 请使用gdb来分析在我们调用了 int 0x80 后，系统的栈发生了怎样的变化? esp 的值和在setup.cpp 中定义的变量 tss 有什么关系? 此外还有哪些段寄存器发生了变化? 变化后的内容是什么?

3. 请使用gdb来分析在进入 asm\_system\_call\_handler 的那一刻，栈顶的地址是什么? 栈中存放的内容是什么? 为什么存放的是这些内容?

4. 请结合代码分析 asm\_system\_call\_handler 是如何找到中断向量号 index 对应的函数的

5. 请使用gdb来分析在 asm\_system\_call handler 中执行 iret 后，哪些段寄存器发生了变化? 变化后的内容是什么? 这些内容来自于什么地方?

* 思路分析：

首先我们需要复现指导书的系统调用的实现一节，系统调用主要是为了用户级程序能够使用到我们的内核态才能够调用的服务，也就是函数，所以我们的第一步就是需要实现特权级的转移，提高特权级之后，通过系统中断的方式来调用系统调用，在完成了系统调用之后，通过iret指令实现从高特权级到低特权级的转移.从而完成系统调用的功能.

* 实验步骤：

**Step1：设置系统调用的接口**

声明如下：

extern "C" int asm\_system\_call(int index, int first = 0, int second = 0, int third = 0, int forth = 0, int fifth = 0);

系统调用函数时，系统调用的参数通过5个寄存器来传递，所以，我们函数的的参数不可以超过5个，之所以使用寄存器来传递参数而不使用栈，这是因为如果使用栈来传递参数，asm\_system\_call的参数就会被保存在用户程序的栈中，也就是低特权级的栈中，但是用户程序使用系统调用时会进行特权级转移，系统调用发生后，我们从低特权级转移到高特权级，此时CPU会从TSS中加载高特权级的栈地址到esp寄存器中，保存参数的栈和现在期望取出函数参数而访问的栈并不是同一个栈，这样就会导致CPU找不到存放在之前低特权级的栈的参数，为了解决这个问题所以我们使用了寄存器来传递系统调用的参数.

所以有了下面的汇编语言来实现我们的asm\_system\_call函数：

asm\_system\_call:

push ebp

mov ebp, esp

push ebx

push ecx

push edx

push esi

push edi

push ds

push es

push fs

push gs

mov eax, [ebp + 2 \* 4]

mov ebx, [ebp + 3 \* 4]

mov ecx, [ebp + 4 \* 4]

mov edx, [ebp + 5 \* 4]

mov esi, [ebp + 6 \* 4]

mov edi, [ebp + 7 \* 4]

int 0x80

pop gs

pop fs

pop es

pop ds

pop edi

pop esi

pop edx

pop ecx

pop ebx

pop ebp

ret

在asm\_system\_call中通过调用0x80号中断，0x80中断处理函数会根据我们传入的index也就是eax寄存器中的系统调用号来调用不同的函数.所以我们先实现我们0x80中断处理函数asm\_system\_call\_handler

asm\_system\_call\_handler:

push ds

push es

push fs

push gs

pushad

push eax

; 栈段会从tss中自动加载

mov eax, DATA\_SELECTOR

mov ds, eax

mov es, eax

mov eax, VIDEO\_SELECTOR

mov gs, eax

pop eax

; 参数压栈

push edi

push esi

push edx

push ecx

push ebx

sti

call dword[system\_call\_table + eax \* 4]

cli

add esp, 5 \* 4

mov [ASM\_TEMP], eax

popad

pop gs

pop fs

pop es

pop ds

mov eax, [ASM\_TEMP]

iret

system\_call\_table是保存我们系统调用函数的地址表，其中系统调用号就是表中的索引，因此[system\_call\_table + eax \* 4]就是系统调用号对应的系统调用处理函数的地址，通过保存在eax中的系统调用号找到对应的系统调用函数，然后调用.

**Step2：实现一个系统调用管理类SystemService**

声明：

class SystemService

{

public:

SystemService();

void initialize();

// 设置系统调用，index=系统调用号，function=处理第index个系统调用函数的地址

bool setSystemCall(int index, int function);

};

// 第0个系统调用

int syscall\_0(int first, int second, int third, int forth, int fifth);

成员函数的实现：

初始化函数首先初始化了系统调用表，然后设置将asm\_system\_call\_handler设置为我们的0x80的中断处理函数，我们希望通过0x80中断从特权级3(用户态)到特权级0(内核态)转移，因此中断描述符中的代码段选择子的RPL=0。然后，我们需要设置中断描述符的DPL=3，否则会由于特权级保护导致用户程序无法使用这个中断.

int system\_call\_table[MAX\_SYSTEM\_CALL];

void SystemService::initialize()

{

memset((char \*)system\_call\_table, 0, sizeof(int) \* MAX\_SYSTEM\_CALL);

// 代码段的选择子默认是DPL=0的平坦模式代码段选择子，

// 但中断描述符的DPL=3，否则用户态程序无法使用该中断描述符

interruptManager.setInterruptDescriptor(0x80, (uint32)asm\_system\_call\_handler, 3);

}

设置系统调用函数，其实就是将function的地址放入到对应系统调用表中的index位置.

bool SystemService::setSystemCall(int index, int function)

{

system\_call\_table[index] = function;

return true;

}

**Step3：测试第一个系统调用**

首先定义0号系统调用函数：

int syscall\_0(int first, int second, int third, int forth, int fifth)

{

printf("systerm call 0: %d, %d, %d, %d, %d\n",

first, second, third, forth, fifth);

return first + second + third + forth + fifth;

}

然后在setup.cpp中设置这个系统调用并调用其测试之

extern "C" void setup\_kernel()

{

...

// 初始化系统调用

systemService.initialize();

systemService.setSystemCall(0, (int)syscall\_0);

int ret;

ret = asm\_system\_call(0);

printf("return value: %d\n", ret);

ret = asm\_system\_call(0, 123);

printf("return value: %d\n", ret);

ret = asm\_system\_call(0, 123, 324);

printf("return value: %d\n", ret);

ret = asm\_system\_call(0, 123, 324, 9248);

printf("return value: %d\n", ret);

ret = asm\_system\_call(0, 123, 324, 9248, 7);

printf("return value: %d\n", ret);

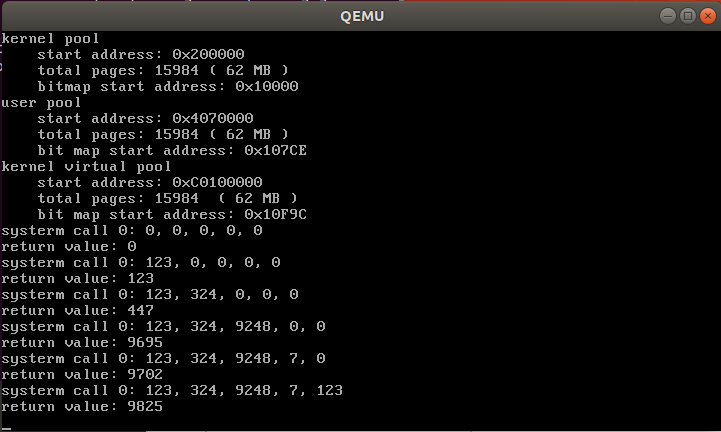
ret = asm\_system\_call(0, 123, 324, 9248, 7, 123);

printf("return value: %d\n", ret);

...

}

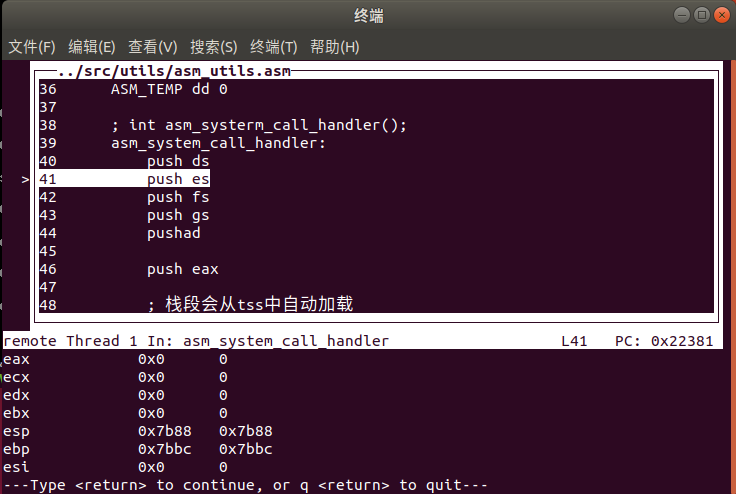
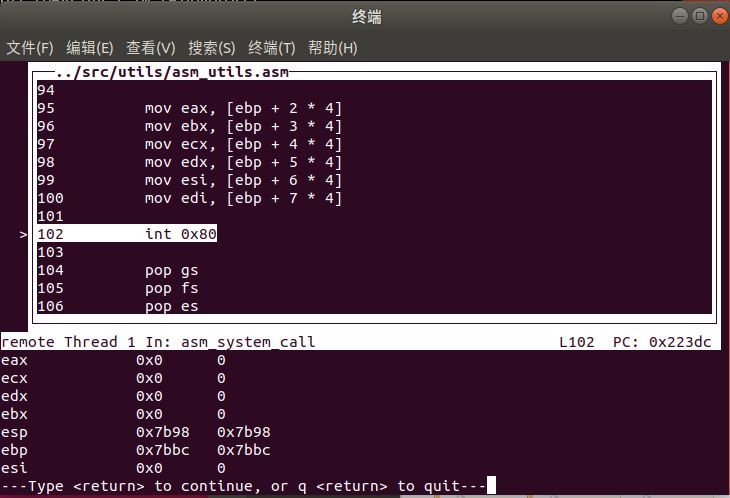
* 实验结果展示：通过执行前述代码，可得下图结果。



可以看到我们的成功的实现了系统调用，接着我们使用gdb来追踪系统调用的过程同时来回答下实验任务的5个问题

1.系统调用函数时，参数通过5个寄存器来传递，这是因为若使用栈来传递参数，asm\_system\_call的参数就会被保存在用户程序的栈中，也就是低特权级的栈中，但是系统调用发生后，用户程序使用系统调用时会进行从低特权级转移到高特权级的特权级转移，此时CPU会从TSS中加载高特权级的栈地址到esp寄存器中，保存参数的栈和现在期望取出函数参数而访问的栈并不是同一个栈，这样就会导致CPU找不到存放在之前低特权级的栈的参数，为了解决这个问题所以我们使用了寄存器来传递系统调用的参数.首先我们从栈中依次弹出低特权级栈中的参数到我们的寄存器即可，通过寄存器来传递到中断处理函数中.

2.我们启动gdb在int 0x80处设置断点，左边是调用前的栈指针，而右边是调用之后的变化

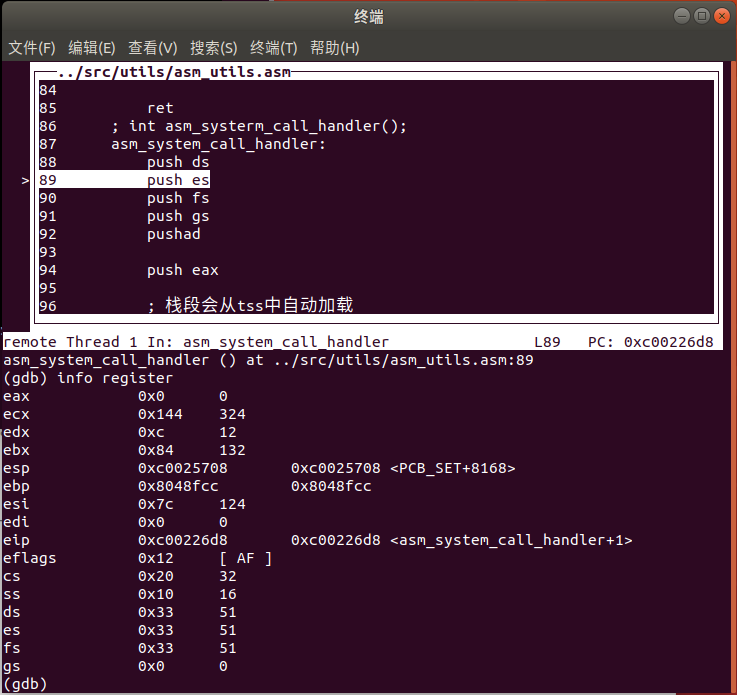
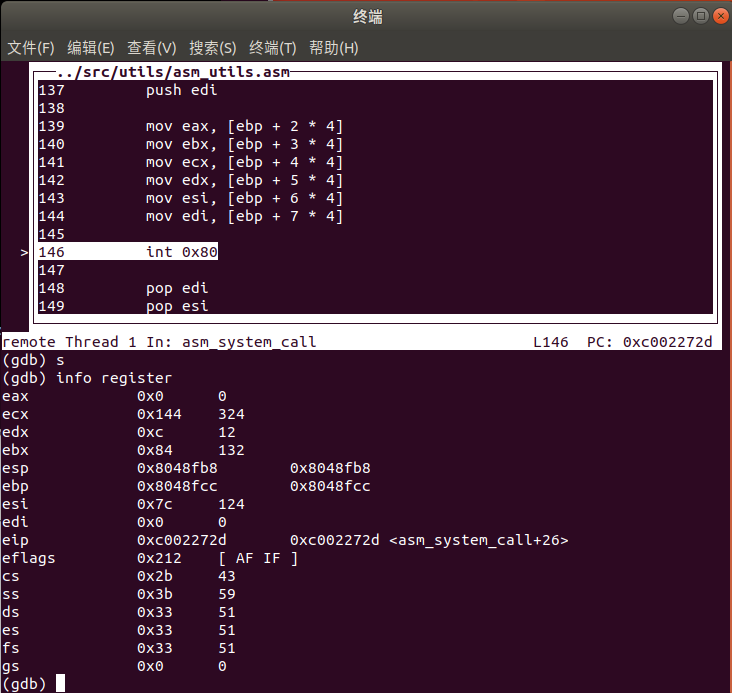


我们看到栈并没有发生变化，只是在本来的栈中又增长了16位寄存器，为什么呢？**这是因为我们本次实验任务我们只是实现了系统调用，并没有实现用户进程，也没有把内核空间提升到3GB以上，所以我们所写的setup的代码是在内核态下的，内核态拥有所有的权限，并不用进行特权级的变化就可以运行我们的代码，所以我们调用中断函数的时候栈不会发生变化.**

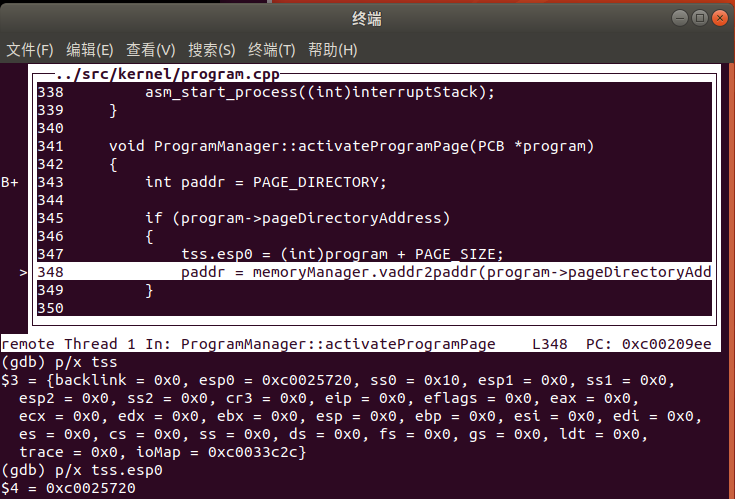
为了观察特权级的变化，我们将试验任务2中的测试例子来进行观察，试验任务2已经实现了用户进程的创建.我们再次执行前面的调试操作如下图所示：

可以看到在调用int 0x80之后我们的栈立马发生了变化，从0x8048fb8变为了0xc0025708，可以看到变化后的栈是0xc00开头的，说明切换后的栈是位于内核空间的特权级0的栈，说明调用之后我们进入到了内核态.

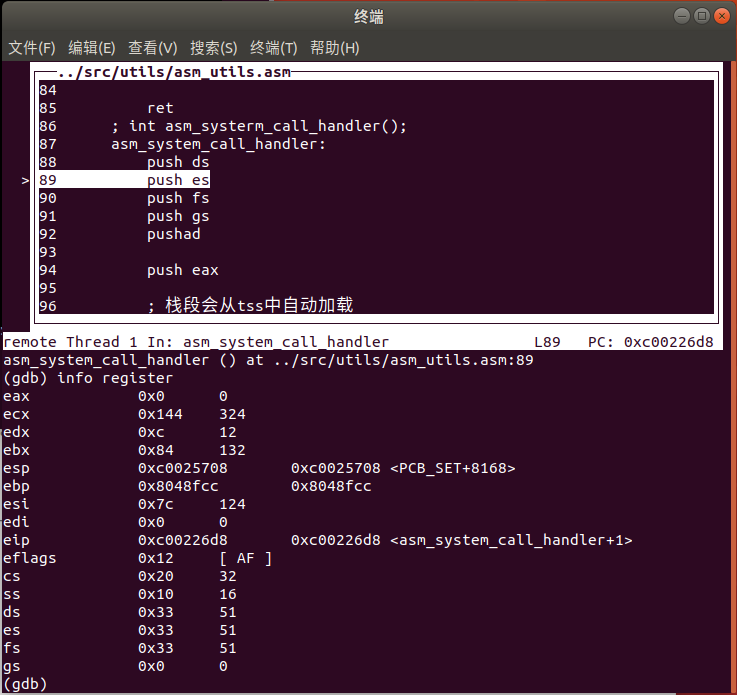
除此之外我们还可以看到我们的**段寄存器代码段选择子寄存器cs、栈段选择子ss寄存器的最后两位都从11（3）变成了00（0），这表明了我们从特权级3提升到了特权级0.**



这时候我们来观察下tss中的值，我们直接查看我们创建的tss实例模型，我们看到tss中0特权级栈的地址，和我们切换到内核态的栈是一样的.

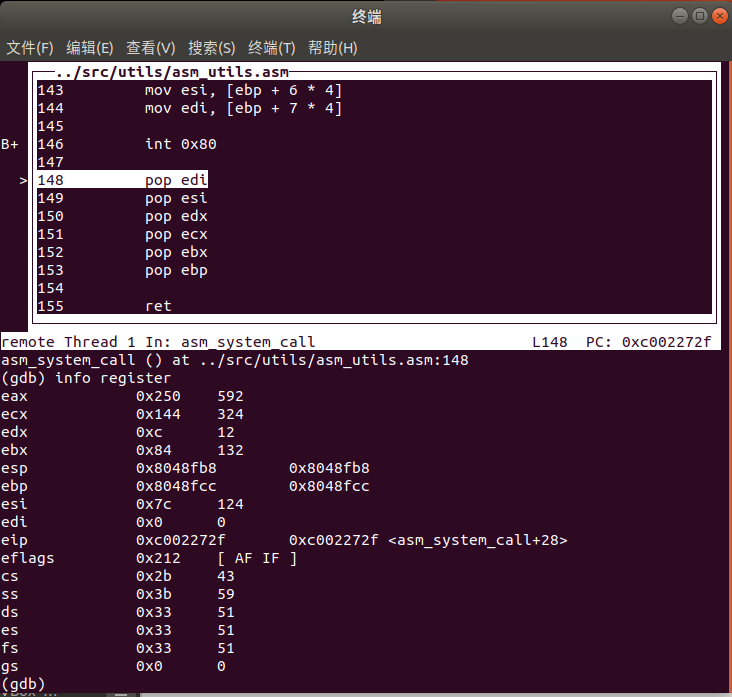
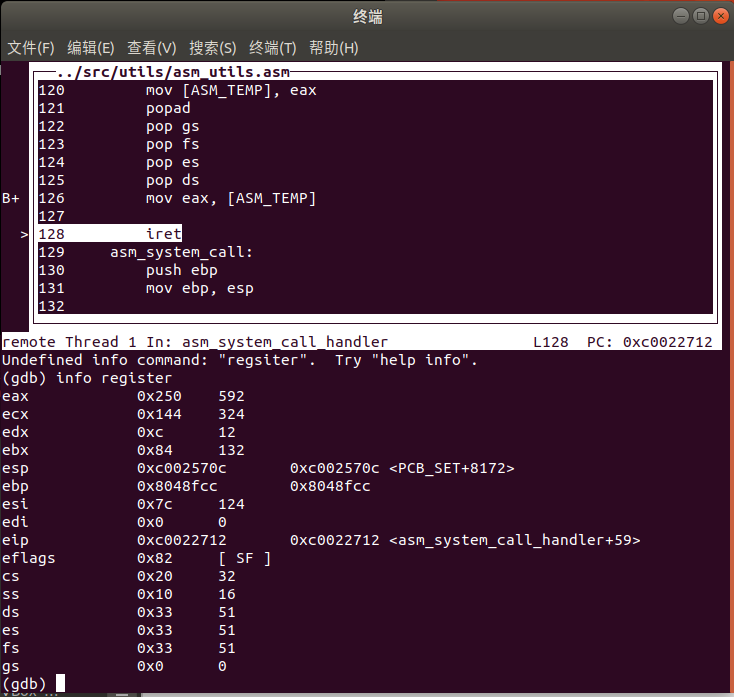
综上就说明了我们TSS的作用：**TSS的作用是向CPU提供进程特权级转移时的栈段选择子和栈指针。**当从低特权级向高特权级转移时，CPU首先会在TSS中找到高特权级栈的段选择子和栈指针，然后送入SS，ESP，此时的栈已经变成了TSS保存的高特权级的栈

3.前面已经讨论过，在进入 asm\_system\_call\_handler 的那一刻，我们的栈顶指针变成了0xc0025708，从用户栈变成了内核态的栈，这个**栈中存放着我们通过寄存器来传入的5个参数**，可以看到我们传入的参数0，324，12，132，0这五个参数都在栈中被我们pop到寄存器中了，**以及中断函数的返回地址**.



4.asm\_system\_call\_handler直接通过eax存放的index参数的值，因为我们的系统调用函数的地址表system\_call\_table是线性的，所以可以**通过[system\_call\_table + eax \* 4]直接计算系统调用号index对应的系统调用处理函数的地址**.

5.我们使用gdb在asm\_system\_call\_hanlder中的iret处打上断点，我们可以看到我们的**代码段寄存器cs，栈段寄存器ss都从高特权级0变为了低特权级的3（11）**，说明中断完成之后我们就会返回用户态.而**这些内容都是来自进入中断前保存在高特权级栈段寄存器的值，在调用iret之后就会进行恢复**.



------------------------- **实验任务2** -------------------------

* 任务要求：

1. 如何在线程的基础上创建进程的PCB的(即分析进程创建的三个步骤)

2. 在进程的PCB第一次被调度执行时，进程实际上并不是跳转到进程的第一条指令处，而是跳转到load process.请结合代码逻辑和gdb来分析为什么 asm\_switch\_thread 在执行 iret 后会跳转到 load process 。

3. 在跳转到 load process 后，我们巧妙地设置了 ProcessStartStack 的内容，然后在asm\_start\_process 中跳转到进程第一条指令处执行。请结合代码逻辑和gdb来分析我们是如何设置 ProcessStartStack 的内容，从而使得我们能够在 asm\_start\_process 中实现内核态到用户态的转移，即从特权级0转移到特权级3下，并使用 iret 指令成功启动进程的。

4. 结合代码，分析在创建进程后，我们对 ProgramManager::schedule 作了哪些修改? 这样做的目的是什么?

5. 在进程的创建过程中，我们存在如下语句 :

int ProgramManager::executeProcess(const char \*filename, int priority)

{

...

//找到刚刚创建的PCB

PCB \*process =ListItem2PCB(allPrograms.back()， tagInAllList);

...

}

正如教程中所提到， “...但是，这样做是存在风险的，我们应该通过pid来找到刚刚创建的PCB...”，现在，同学们需要编写一个 ProgramManager 的成员函数 findProgramByPid :

PCB \*findProgramByPid(int pid);

并用上面这个函数替换指导书中提到的"存在风险的语句"，替换结果如下:

int ProgramManager::executeProcess(const char \*filename, int priority){

...

//找到刚刚创建的PCB

PCB \*process =findProgramByPid(pid);

...

}

* 思路分析：

为了实现进程的实现，进程的调度，第一个进程，用户进程和内核线程最大的区别在于用户进程有自己的虚拟地址空间，而内核线程使用的是内核虚拟地址空间。同时为了避免虚拟地址的冲突，每个进程设置自己的页目录表和页表，这样即使对于相同的虚拟地址，我们的页目录表和页表不同，那么CPU在寻址过程中得到的物理地址也就会不同。以及为了实现资源的共享，每个进程的3~4GB的虚拟地址空间来作为共享的内核区域从而使得进程也能访问到内核的资源。

所以我们将会在线程的基础上为进程添加虚拟地址池、页目录表从而创建出进程，然后通过切换虚拟地址池、页目录表来实现进程的调度，然后创建本次实验的第一个进程测试我们的进程管理

* 实验步骤：

**Step1：创建进程前的准备，提升内核空间的地址到3GB**

我们需要将我们的内核变量提升到3GB以上的空间，也就是加上0xc0000000偏移，但是我们是通过分页机制将内核空间映射到0~1MB的空间的空间的，因此，为了能够让CPU能够正确寻址，我们需要在跳转到内核之前就开启分页机制，也就是在bootloader中完成这个过程.

修改后的bootloader如下：

...

**call open\_page\_mechanism**

mov eax, PAGE\_DIRECTORY

mov cr3, eax ; 放入页目录表地址

mov eax, cr0

or eax, 0x80000000

mov cr0, eax ; 置PG=1，开启分页机制

sgdt [pgdt]

add dword[pgdt + 2], 0xc0000000

lgdt [pgdt]

jmp dword CODE\_SELECTOR:KERNEL\_VIRTUAL\_ADDRESS ; 跳转到kernel

...

由于我们的内核提升到了3GB空间，所以我们要修改之前常量和变量，将其都提升到3GB以上.

boot.inc：

; \_\_\_\_\_\_\_\_\_\_kernel\_\_\_\_\_\_\_\_\_

KERNEL\_START\_SECTOR equ 6

KERNEL\_SECTOR\_COUNT equ 145

KERNEL\_START\_ADDRESS equ 0x20000

KERNEL\_VIRTUAL\_ADDRESS equ 0xc0020000

os\_constant.h中的地址也要提升到3GB空间：

#ifndef OS\_CONSTANT\_H

#define OS\_CONSTANT\_H

#define IDT\_START\_ADDRESS 0xc0008880

...

#define MEMORY\_SIZE\_ADDRESS 0xc0007c00

#define BITMAP\_START\_ADDRESS 0xc0010000

#define KERNEL\_VIRTUAL\_START 0xc0100000

...

#endif

显存访问地址也提升到3GB空间：

void STDIO::initialize()

{

screen = (uint8 \*)0xc00b8000;

}

**Step2：初始化TSS和用户段描述符**

首先向ProgramManager中加入存储代码段、数据段和栈段描述符的变量

class ProgramManager

{

public:

...

int USER\_CODE\_SELECTOR; // 用户代码段选择子

int USER\_DATA\_SELECTOR; // 用户数据段选择子

int USER\_STACK\_SELECTOR; // 用户栈段选择子

...

}

由于进程的运行环境需要用到TSS、特权级3下的平坦模式代码段和数据段描述符，我们现在来初始化这些内容，如下所示。

void ProgramManager::initialize()

{

allPrograms.initialize();

readyPrograms.initialize();

running = nullptr;

for (int i = 0; i < MAX\_PROGRAM\_AMOUNT; ++i)

{

PCB\_SET\_STATUS[i] = false;

}

// 初始化用户代码段、数据段和栈段

int selector;

selector = asm\_add\_global\_descriptor(USER\_CODE\_LOW, USER\_CODE\_HIGH);

USER\_CODE\_SELECTOR = (selector << 3) | 0x3;

selector = asm\_add\_global\_descriptor(USER\_DATA\_LOW, USER\_DATA\_HIGH);

USER\_DATA\_SELECTOR = (selector << 3) | 0x3;

selector = asm\_add\_global\_descriptor(USER\_STACK\_LOW, USER\_STACK\_HIGH);

USER\_STACK\_SELECTOR = (selector << 3) | 0x3;

initializeTSS();

}

红色代码部分，我们向GDT中新增如下所示的DPL为3用户代码段描述符，数据段描述符和栈段描述符，描述符的定义如下：

#define USER\_CODE\_LOW 0x0000ffff

#define USER\_CODE\_HIGH 0x00cff800

#define USER\_DATA\_LOW 0x0000ffff

#define USER\_DATA\_HIGH 0x00cff200

#define USER\_STACK\_LOW 0x00000000

#define USER\_STACK\_HIGH 0x0040f600

然后使用asm\_add\_global\_descriptor函数将上述段描述符放入GDT，并更新GDTR的内容，最后返回段描述符在GDT的位置.

首先，先读入GDTR的内容，然后找到GDT的起始地址和偏移地址，然后在GDT的末尾新增一个段描述符，然后计算段描述符在GDT的位置，并放入eax寄存器中，最后将GDT的界限增加8并更新GDTR。

; int asm\_add\_global\_descriptor(int low, int high);

asm\_add\_global\_descriptor:

push ebp

mov ebp, esp

push ebx

push esi

sgdt [ASM\_GDTR]

mov ebx, [ASM\_GDTR + 2] ; GDT地址

xor esi, esi

mov si, word[ASM\_GDTR] ; GDT界限

add esi, 1

mov eax, [ebp + 2 \* 4] ; low

mov dword [ebx + esi], eax

mov eax, [ebp + 3 \* 4] ; high

mov dword [ebx + esi + 4], eax

mov eax, esi

shr eax, 3

add word[ASM\_GDTR], 8

lgdt [ASM\_GDTR]

pop esi

pop ebx

pop ebp

ret

然后我们需要定义TSS结构体，注意这个结构体的内容必须要这样子设置.

struct TSS

{

public:

int backlink;

int esp0;

int ss0;

int esp1;

int ss1;

int esp2;

int ss2;

int cr3;

int eip;

int eflags;

int eax;

int ecx;

int edx;

int ebx;

int esp;

int ebp;

int esi;

int edi;

int es;

int cs;

int ss;

int ds;

int fs;

int gs;

int ldt;

int trace;

int ioMap;

};

然后我们定义我们的TSS初始化函数.TSS本质上也是一个描述符，CPU是在特权级转移的时候自动加载TSS的内容的，他的选择子被存放到TR寄存器，我们使用ltr将TSS的段选择子送入TR即可

void ProgramManager::initializeTSS()

{

int size = sizeof(TSS);

int address = (int)&tss;

memset((char \*)address, 0, size);

tss.ss0 = STACK\_SELECTOR; // 内核态堆栈段选择子

int low, high, limit;

limit = size - 1;

low = (address << 16) | (limit & 0xff);

// DPL = 0

high = (address & 0xff000000) | ((address & 0x00ff0000) >> 16) | ((limit & 0xff00) << 16) | 0x00008900;

int selector = asm\_add\_global\_descriptor(low, high);

// RPL = 0

asm\_ltr(selector << 3);

tss.ioMap = address + size;

}

**Step3：进程的创建**

首先修改下我们的PCB，由于进程都有着自己的虚拟地址空间，所以我们在PCB中增加些进程中的数据结构，虚拟地址池和页目录表.

struct PCB

{

...

int pageDirectoryAddress; // 页目录表地址

AddressPool userVirtual; // 用户程序虚拟地址池

};

我们进程的创建是通过executeProcess函数来实现的：

int ProgramManager::executeProcess(const char \*filename, int priority)

{

bool status = interruptManager.getInterruptStatus();

interruptManager.disableInterrupt();

// 在线程创建的基础上初步创建进程的PCB

int pid = executeThread((ThreadFunction)load\_process, (void \*)filename, filename, priority);

if (pid == -1)

{

interruptManager.setInterruptStatus(status);

return -1;

}

// 找到刚刚创建的PCB

PCB \*process = ListItem2PCB(allPrograms.back()， tagInAllList);

// 创建进程的页目录表

process->pageDirectoryAddress = createProcessPageDirectory();

if (!process->pageDirectoryAddress)

{

process->status = ThreadStatus::DEAD;

interruptManager.setInterruptStatus(status);

return -1;

}

// 创建进程的虚拟地址池

bool res = createUserVirtualPool(process);

if (!res)

{

process->status = ThreadStatus::DEAD;

interruptManager.setInterruptStatus(status);

return -1;

}

interruptManager.setInterruptStatus(status);

return pid;

}

首先我们在线程的基础上创建一个线程的PCB，然后在线程PCB的基础上，初始化PCB中的虚拟地址池和页目录表.

进程的页目录表我们是通过createProcessPageDirectory函数来创建和初始化的，如下所示：

首先，我们从内核中分配一页来存储进程的页目录表，然后我们将内核的第768~1022个页目录项复制到进程的页目录表的相同位置，令用户虚拟地址的3GB-4GB的空间指向内核空间，最后，我们将最后一个页目录项指向用户进程页目录表物理地址，这是为了我们在切换到用户进程后，我们也能够构造出页目录项和页表项的虚拟地址。

int ProgramManager::createProcessPageDirectory()

{

// 从内核地址池中分配一页存储用户进程的页目录表

int vaddr = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);

if (!vaddr)

{

//printf("can not create page from kernel\n");

return 0;

}

memset((char \*)vaddr, 0, PAGE\_SIZE);

// 复制内核目录项到虚拟地址的高1GB

int \*src = (int \*)(0xfffff000 + 0x300 \* 4);

int \*dst = (int \*)(vaddr + 0x300 \* 4);

for (int i = 0; i < 256; ++i)

{

dst[i] = src[i];

}

// 用户进程页目录表的最后一项指向用户进程页目录表本身

((int \*)vaddr)[1023] = memoryManager.vaddr2paddr(vaddr) | 0x7;

return vaddr;

}

进程的虚拟地址池通过createUserVirtualPool来实现，如下所示：

首先，我们计算这部分地址所占的页表的数量，从而计算出为了管理这部分地址所需的位图大小，然后，我们在内核空间中为进程分配位图所需的内存，最后，我们初始化用户虚拟地址池。

bool ProgramManager::createUserVirtualPool(PCB \*process)

{

int sourcesCount = (0xc0000000 - USER\_VADDR\_START) / PAGE\_SIZE;

int bitmapLength = ceil(sourcesCount, 8);

// 计算位图所占的页数

int pagesCount = ceil(bitmapLength, PAGE\_SIZE);

int start = memoryManager.allocatePages(AddressPoolType::KERNEL, pagesCount);

if (!start)

{

return false;

}

memset((char \*)start, 0, PAGE\_SIZE \* pagesCount);

(process->userVirtual).initialize((char \*)start, bitmapLength, USER\_VADDR\_START);

return true;

}

至此我们完成了进程的创建.这也是我们实验任务第一个问题的答案.

在创建用户进程的时候，我们是运行在特权级0下，但是我们调度的用户进程是运行在特权级3下的，但是CPU不允许我们从高特权级转移到低特权级，只能够通过iret指令强制将低特权级下的段选择子和栈送入段寄存器，所以为了能够成功的运行的我们的进程我们要实现load\_process函数实现进程的启动.

为了方便表示，我们首先定义一个ProgramStartStack类来表示启动进程之前栈放入的内容

struct ProcessStartStack{

int edi;

int esi;

int ebp;

int esp\_dummy;

int ebx;

int edx;

int ecx;

int eax;

int gs;

int fs;

int es;

int ds;

int eip;

int cs;

int eflags;

int esp;

int ss;

};

然后，我们需要在PCB的顶部预留出ProcessStartStack的空间，以便后续ProcessStartStack的加入.

int ProgramManager::executeThread(ThreadFunction function, void \*parameter, const char \*name, int priority){

...

// 线程栈

thread->stack = (int \*)((int)thread + PAGE\_SIZE - sizeof(ProcessStartStack));

...

}

当进程的PCB被首次加载到处理器执行时，CPU首先会进入load\_process来启动我们的进程，如下所示：

void load\_process(const char \*filename)

{

interruptManager.disableInterrupt();

PCB \*process = programManager.running;

ProcessStartStack \*interruptStack = (ProcessStartStack \*)((int)process + PAGE\_SIZE - sizeof(ProcessStartStack));

interruptStack->edi = 0;

interruptStack->esi = 0;

interruptStack->ebp = 0;

interruptStack->esp\_dummy = 0;

interruptStack->ebx = 0;

interruptStack->edx = 0;

interruptStack->ecx = 0;

interruptStack->eax = 0;

interruptStack->gs = 0;

interruptStack->fs = programManager.USER\_DATA\_SELECTOR;

interruptStack->es = programManager.USER\_DATA\_SELECTOR;

interruptStack->ds = programManager.USER\_DATA\_SELECTOR;

interruptStack->eip = (int)filename;

interruptStack->cs = programManager.USER\_CODE\_SELECTOR;

interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1);

interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1);

if (interruptStack->esp == 0)

{

printf("can not build process!\n");

process->status = ThreadStatus::DEAD;

asm\_halt();

}

interruptStack->esp += PAGE\_SIZE;

interruptStack->ss = programManager.USER\_STACK\_SELECTOR;

asm\_start\_process((int)interruptStack);

}

首先我们需要初始化启动进程需要的栈结构，我们首先要将用户进程低特权级的选择子放入栈中的fs，es，ds，cs，然后将我们希望的返回地址放入到eip中，这样在返回的时候CPU就可以直接跳转到eip所在的地址来执行函数.

然后，每一个特权级都有自己的栈，我们在进程虚拟地址空间中分配一页来作为进程的特权级3栈，进程的特权级0栈在进程的PCB中

最后，通过asm\_start\_process中断返回iret来启动进程，如下所示：

我们将ProcessStartStack的起始地址送入了esp，这个栈就是我们在load\_process中自己设置的栈，然后我们首先pop出栈中的所有寄存器的值、pop出栈中低特权级的选择子放入到寄存器gs、fs、es、ds，最后我们使用iret中断返回指令，特权级3的选择子被放入到段寄存器中，然后代码将会返回到eip中的地址继续执行，而eip就是我们进程函数的起始地址

; void asm\_start\_process(int stack);

asm\_start\_process:

;jmp $

mov eax, dword[esp+4]

mov esp, eax

popad

pop gs;

pop fs;

pop es;

pop ds;

iret

最后，在用户进程中，我们需要进程内存分配和释放，这部分内存是来源于用户虚拟空间和用户物理空间的，因此，我们需要对页内存分配和释放的函数稍作修改，使得我们可以分配和释放用户空间的页内存。

int MemoryManager::allocateVirtualPages(enum AddressPoolType type, const int count)

{

int start = -1;

if (type == AddressPoolType::KERNEL)

{

start = kernelVirtual.allocate(count);

} else if (type == AddressPoolType::USER){

start = programManager.running->userVirtual.allocate(count);

}

return (start == -1) ? 0 : start;

}

void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count)

{

if (type == AddressPoolType::KERNEL)

{

kernelVirtual.release(vaddr, count);

}

else if (type == AddressPoolType::UESR)

{

programManager.running->userVirtual.release(vaddr, count);

}

}

**Step4：进程的调度**

我们接下来完成进程的调度部份，进程的调度只需要在原先的线程调度的基础上，加入页目录表的切换和TSS特权级0栈的切换即可实现进程的切换.

void ProgramManager::schedule()

{

...

activateProgramPage(next);

...

}

void ProgramManager::activateProgramPage(PCB \*program)

{

int paddr = PAGE\_DIRECTORY;

if (program->pageDirectoryAddress)

{

tss.esp0 = (int)program + PAGE\_SIZE;

paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);

}

asm\_update\_cr3(paddr);

}

**Step5：创建第一个进程**

我们创建第一个进程，这个进程会调用之前的系统调用0，如下所示：我们创建了三个进程.

void first\_process()

{

asm\_system\_call(0, 132, 324, 12, 124);

asm\_halt();

}

void first\_thread(void \*arg)

{

printf("start process\n");

programManager.executeProcess((const char \*)first\_process, 1);

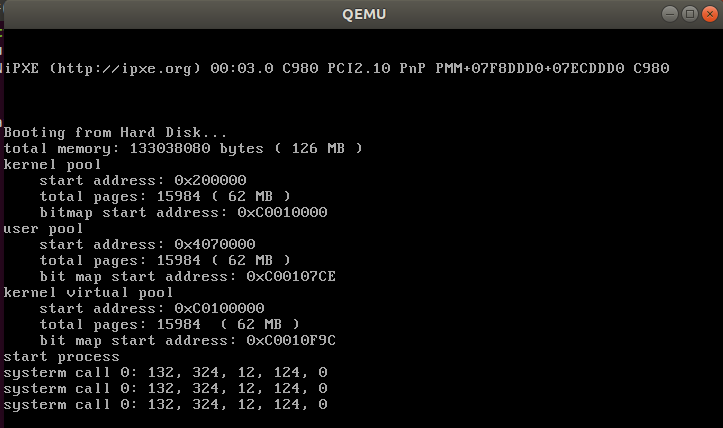
programManager.executeProcess((const char \*)first\_process, 1);

programManager.executeProcess((const char \*)first\_process, 1);

asm\_halt();

}

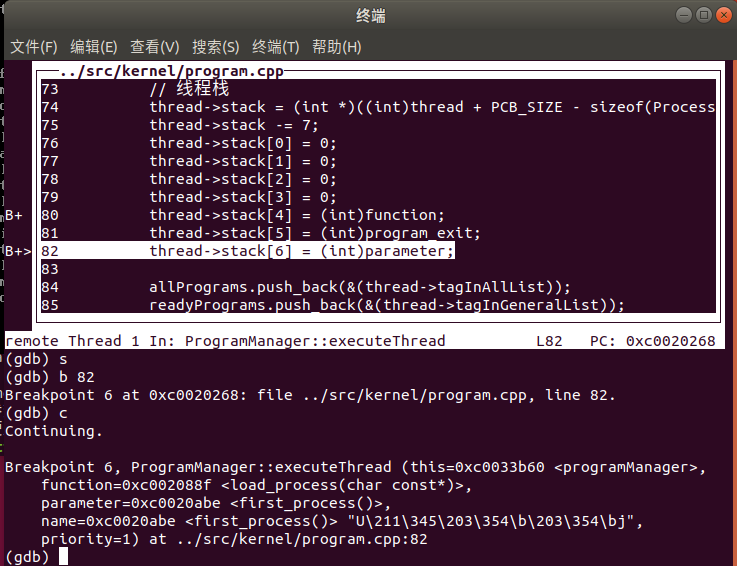
* 实验结果展示：



可以看到我们成功的实现了用户的进程，接下来将通过gdb来追踪我们创建进程的过程，同时回答实验任务中的5个问题

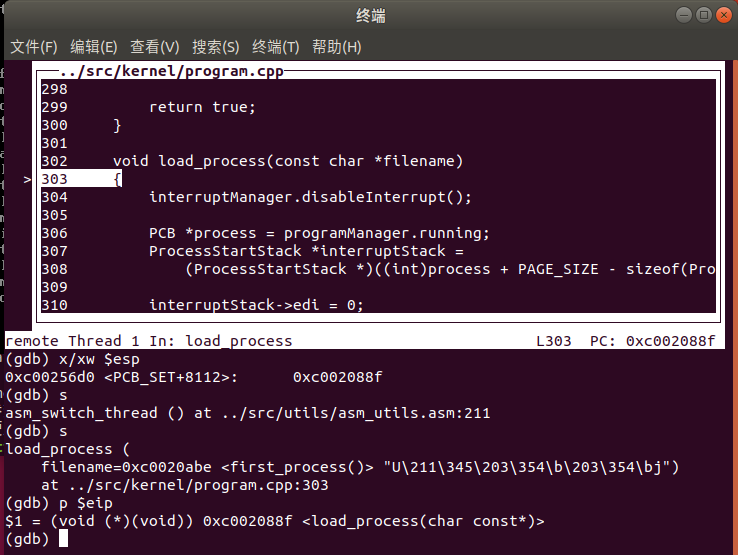
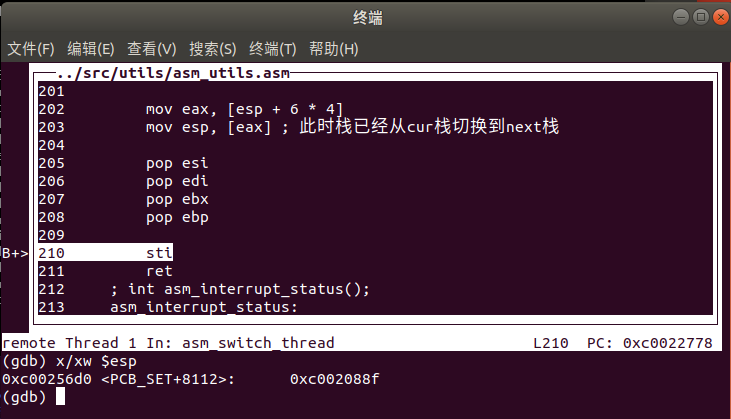
1.首先我们创建一个线程的PCB，然后通过createUserVirtualPool来创建进程的虚拟地址池以及createProcessPageDirectory来创建进程的页目录表，[详细的分析已在前面的实验步骤中给出.](#进程的创建详细分析)

2.我们在创建进程的PCB的时候我们将函数executeThread将我们传入的load\_process函数的地址放入到了进程栈的最顶部，然后在放入进程的返回时跳转的地址，然后再是进程结束返回的参数的地址.如下图所示，注意我们function的地址0xc002088f，后面会用到.

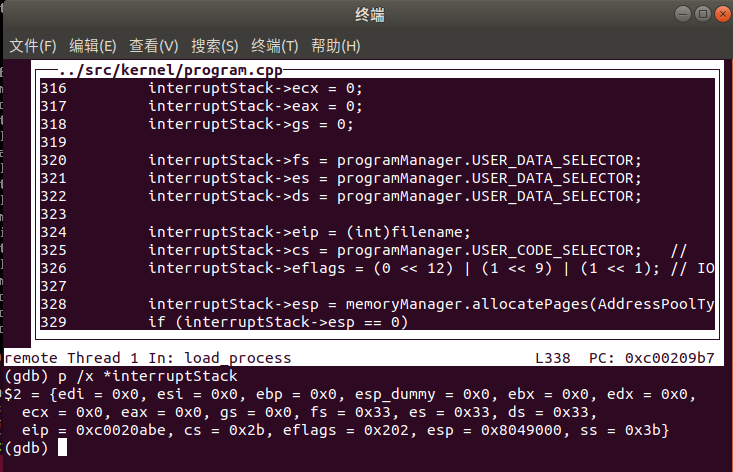


然后我们跳转到asm\_switch\_thread 中，首先asm\_switch\_thread将切换下一个进程的栈，然后我们将栈的前四个参数都pop到寄存器中，**此时我们的栈顶元素就是function的地址，也就是load\_process函数的地址，然后在执行 iret 后，function会被加载进eip，从而使得CPU跳转到这个函数中执行，**所以asm\_switch\_thread 在执行 iret 后会跳转到 load\_process

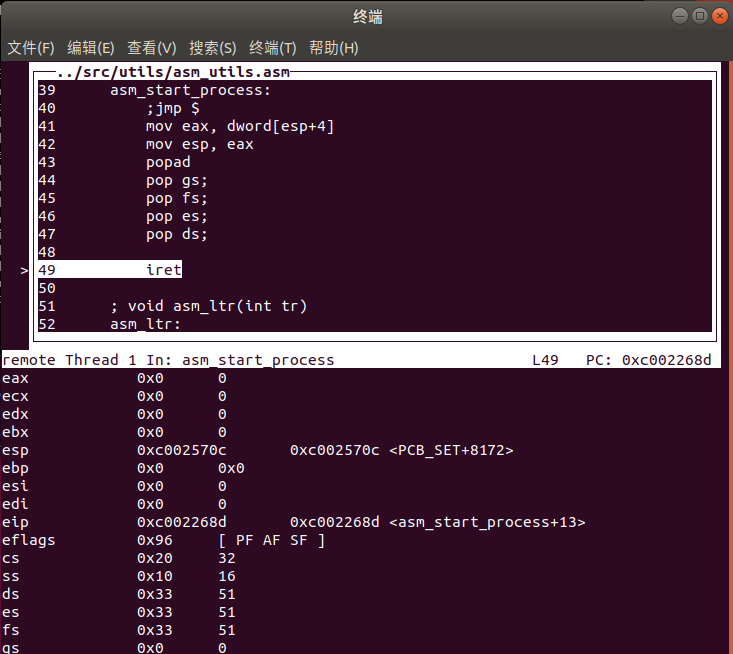
左图就是asm\_switch\_thread准备ret之前栈顶的地址，这个地址就是我们load\_process的地址.而右图就是执行ret之后跳转到load\_process函数处，此时的eip刚好是asm\_switch\_thread的栈顶地址.



3.load\_process中，我们设置一个用于启动进程的栈，因为此时我们是处于高特权级的栈，想要跳转到进程的低特权级的栈就必须利用iret指令来强制降低我们的特权级.所以我们准备一个用于启动进程的栈，栈中存放着我们的用户级的段选择子，可以看到我们将fs、es、ds都设置成了用户级的段选择子，cs也设置成了用户级的代码段选择子，同时将eip设置成我们进程执行函数的地址，如下图所示.



设置完ProcessStartStack之后，我们调用asm\_start\_procrss使用中断返回来启动我们的进程.如下图所示，首先我们先加载我们传入的ProcessStartStack，依次pop出其中的值到寄存器.最后执行iret中断返回后，特权级3的选择子被放入到段寄存器中，代码跳转到进程的起始处执行，CPU会跳转到 eip 指向的地址继续执行，而eip被我们巧妙的设置为了进程执行函数的地址，所以可以实现特权级的转换以及函数的跳转，从而启动我们的进程.



4.schedule增加了一步activateProgramPage(next)，代码如下：

void ProgramManager::activateProgramPage(PCB \*program)

{

int paddr = PAGE\_DIRECTORY;

if (program->pageDirectoryAddress)

{

tss.esp0 = (int)program + PAGE\_SIZE;

paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);

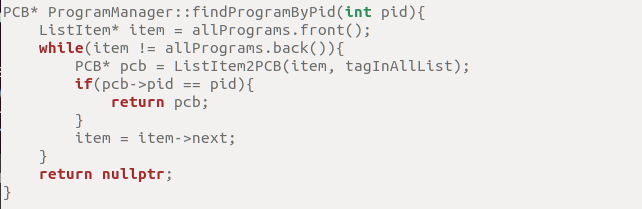
}

asm\_update\_cr3(paddr);

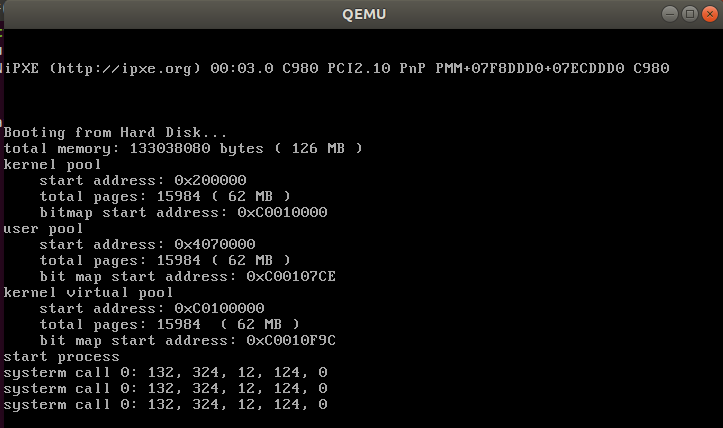
}

这样做是为了切换页目录表以及更新TSS中的特权级0的栈，我们知道进程有着自己的虚拟地址空间和特权级0的栈用于切换到内核态，所以实现进程的调度的时候，我们需要切换进程的页目录表以及TSS中的特权级0栈，然后再切换我们的进程，实现进程的调度.

5.为了解决PCB \*process =ListItem2PCB(allPrograms.back()， tagInAllList)语句带来的风险，我实现了下面的findProgramByPid函数，这个函数遍历allPrograms中的所有进程PCB的pid来寻找新创建的PCB.



然后将其替换原来的风险语句，再次make run启动测试，得到如下的结果：



可以看到我们使用pid来寻找进程，同样实现了进程的创建.

------------------------- **实验任务3** -------------------------

* 任务要求：

1. 请根据代码逻辑概括 fork 的实现的基本思路，并简要分析我们是如何解决"四个关键问题”的

2. 请根据gdb来分析子进程第一次被调度执行时，即在 asm\_switch\_thread 切换到子进程的栈中时，esp 的地址是什么? 栈中保存的内容是什么?

3. 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从 fork 返回，根据gdb来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完 ProgramManager::fork 后的返回过程的异同

4. 请根据代码逻辑和gdb来解释子进程的 fork 返回值为什么是0，而父进程的 fork 返回值是子进程的pid

5. 请解释在 Programanager::schedule 中，我们是如何从一个进程的虚拟地址空间切换到另外一个进程的虚拟地址空间的

* 思路分析：

fork函数用于将运行中的进程分成两个（几乎）完全一样的进程，每个进程会从fork的返回点开始执行。fork是一个系统调用，所以我们要在syscall声明同时并实现fork函数.

fork函数用于创建一个子进程，子进程是父进程的副本，所以我们需要将父进程的数据段，栈全都复制到子进程的PCB中。

* 实验步骤：

**Step1：首先实现fork系统调用和系统调用处理函数的定义和实现**

int fork() {

return asm\_system\_call(2);

}

int syscall\_fork() {

return programManager.fork();

}

然后在setup.cpp中设置fork系统调用

...

extern "C" void setup\_kernel()

{

...

// 设置2号系统调用

systemService.setSystemCall(2, (int)syscall\_fork);

...

进入内核态后，fork的实现通过ProgramManager::fork来实现的：

首先，fork是进程的系统调用，因此我们禁止内核线程调用，我们知道进程和线程的区别就是进程有页目录表，所以我们可以通过pageDirectoryAddress是否为0来判断当前执行的是线程还是进程。然后我们调用executeProcess来创建一个子进程，然后找到刚刚创建的子进程，调用copyProcess复制父进程的资源到子进程.

int ProgramManager::fork()

{

bool status = interruptManager.getInterruptStatus();

interruptManager.disableInterrupt();

// 禁止内核线程调用

PCB \*parent = this->running;

if (!parent->pageDirectoryAddress)

{

interruptManager.setInterruptStatus(status);

return -1;

}

// 创建子进程

int pid = executeProcess("", 0);

if (pid == -1)

{

interruptManager.setInterruptStatus(status);

return -1;

}

// 初始化子进程

PCB \*child = ListItem2PCB(this->allPrograms.back(), tagInAllList);

bool flag = copyProcess(parent, child);

if (!flag)

{

child->status = ProgramStatus::DEAD;

interruptManager.setInterruptStatus(status);

return -1;

}

interruptManager.setInterruptStatus(status);

return pid;

}

复制父进程的资源到子进程中是由copyProcess来完成的，详细设计如下：

首先我们我们实际上就是把在中断的那一刻保存的寄存器的内容复制到子进程的0特权级栈中，然后将eax寄存器的值设置为0，这是子进程的返回值.

bool ProgramManager::copyProcess(PCB \*parent, PCB \*child){

// 复制进程0级栈

ProcessStartStack \*childpss = (ProcessStartStack \*)((int)child + PAGE\_SIZE - sizeof(ProcessStartStack));

ProcessStartStack \*parentpss =(ProcessStartStack \*)((int)parent + PAGE\_SIZE - sizeof(ProcessStartStack));

memcpy(parentpss, childpss, sizeof(ProcessStartStack));

// 设置子进程的返回值为0

childpss->eax = 0;

接着我们初始化子进程的0特权级栈，我们将asm\_start\_process设置为0特权级栈的执行函数，当asm\_switch\_thread将子进程换上到CPU时，子进程能够从asm\_switch\_thread跳转到asm\_start\_proces处执行

// 准备执行asm\_switch\_thread的栈的内容

child->stack = (int \*)childpss - 7;

child->stack[0] = 0;

child->stack[1] = 0;

child->stack[2] = 0;

child->stack[3] = 0;

child->stack[4] = (int)asm\_start\_process;

child->stack[5] = 0; // asm\_start\_process 返回地址

child->stack[6] = (int)childpss; // asm\_start\_process 参数

接着，我们设置子进程的PCB，直接将父进程的虚拟地址池、bitmap、PCB复制到子进程的PCB中即可.

// 设置子进程的PCB

child->status = ProgramStatus::READY;

child->parentPid = parent->pid;

child->priority = parent->priority;

child->ticks = parent->ticks;

child->ticksPassedBy = parent->ticksPassedBy;

strcpy(parent->name, child->name);

// 复制用户虚拟地址池

int bitmapLength = parent->userVirtual.resources.length;

int bitmapBytes = ceil(bitmapLength, 8);

memcpy(parent->userVirtual.resources.bitmap, child->userVirtual.resources.bitmap, bitmapBytes);

然后，从内核中分配一页来作为数据复制的中转页，因为分页机制实现了地址隔离，父进程就无法将数据复制到具有相同虚拟地址的子进程中，因此，内核空间的中转页，首先在父进程的虚拟地址空间下将数据复制到中转页中，再切换到子进程的虚拟地址空间中，然后将中转页复制到子进程对应的位置，同时这是第4个关键问题的答案

// 从内核中分配一页作为中转页

char \*buffer = (char \*)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);

if (!buffer)

{

child->status = ProgramStatus::DEAD;

return false;

}

然后，我们先将父进程的页目录表复制到子进程中，我们首先检查目录项是否有对应的页表，然后在页目录项有指向的页表的前提下，我们从用户物理空间中分配一个物理页作为页目录项指向的页表，然后我们切换到子进程的虚拟地址空间，接着我们设置子进程页目录表的页目录项，然后初始化页目录项指向的页表，最后我们切换回父进程的虚拟地址空间

// 子进程页目录表物理地址

int childPageDirPaddr = memoryManager.vaddr2paddr(child->pageDirectoryAddress);

// 父进程页目录表物理地址

int parentPageDirPaddr = memoryManager.vaddr2paddr(parent->pageDirectoryAddress);

// 子进程页目录表指针(虚拟地址)

int \*childPageDir = (int \*)child->pageDirectoryAddress;

// 父进程页目录表指针(虚拟地址)

int \*parentPageDir = (int \*)parent->pageDirectoryAddress;

// 子进程页目录表初始化

memset((void \*)child->pageDirectoryAddress, 0, 768 \* 4);

// 复制页目录表

for (int i = 0; i < 768; ++i)

{

// 无对应页表

if (!(parentPageDir[i] & 0x1))

{

continue;

}

// 从用户物理地址池中分配一页，作为子进程的页目录项指向的页表

int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);

if (!paddr)

{

child->status = ProgramStatus::DEAD;

return false;

}

// 页目录项

int pde = parentPageDir[i];

// 构造页表的起始虚拟地址

int \*pageTableVaddr = (int \*)(0xffc00000 + (i << 12));

asm\_update\_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

childPageDir[i] = (pde & 0x00000fff) | paddr;

memset(pageTableVaddr, 0, PAGE\_SIZE);

asm\_update\_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间

}

然后处理完子进程的页目录表后，复制页表和物理页的数据，

首先，检查页表项指向的页表是否存在，在页表项指向的页表存在前提下，从用户物理地址池中分配一页作为子进程的页表项指向的物理页，然后复制父进程的物理页的数据到中转页中，接着切换到子进程的虚拟地址空间设置页表项，然后从中转页中复制数据到子进程的物理页中，最后，切换到父进程的虚拟地址空间。

// 复制页表和物理页

for (int i = 0; i < 768; ++i)

{

// 无对应页表

if (!(parentPageDir[i] & 0x1))

{

continue;

}

// 计算页表的虚拟地址

int \*pageTableVaddr = (int \*)(0xffc00000 + (i << 12));

// 复制物理页

for (int j = 0; j < 1024; ++j)

{

// 无对应物理页

if (!(pageTableVaddr[j] & 0x1))

{

continue;

}

// 从用户物理地址池中分配一页，作为子进程的页表项指向的物理页

int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);

if (!paddr){

child->status = ProgramStatus::DEAD;

return false;

}

// 构造物理页的起始虚拟地址

void \*pageVaddr = (void \*)((i << 22) + (j << 12));

// 页表项

int pte = pageTableVaddr[j];

// 复制出父进程物理页的内容到中转页

memcpy(pageVaddr, buffer, PAGE\_SIZE);

asm\_update\_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

pageTableVaddr[j] = (pte & 0x00000fff) | paddr;

// 从中转页中复制到子进程的物理页

memcpy(buffer, pageVaddr, PAGE\_SIZE);

asm\_update\_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间

}

}

最后，我们归还中转页，返回子进程的pid，这也是父进程fork的返回值

// 归还从内核分配的中转页

memoryManager.releasePages(AddressPoolType::KERNEL, (int)buffer, 1);

return true;

}

**Step2：我们编写一个进程测试一下fork函数**

void first\_process()

{

int pid = fork();

if (pid == -1)

{

printf("can not fork\n");

}

else

{

if (pid)

{

printf("I am father, fork reutrn: %d\n", pid);

}

else

{

printf("I am child, fork return: %d, my pid: %d\n", pid,

programManager.running->pid);

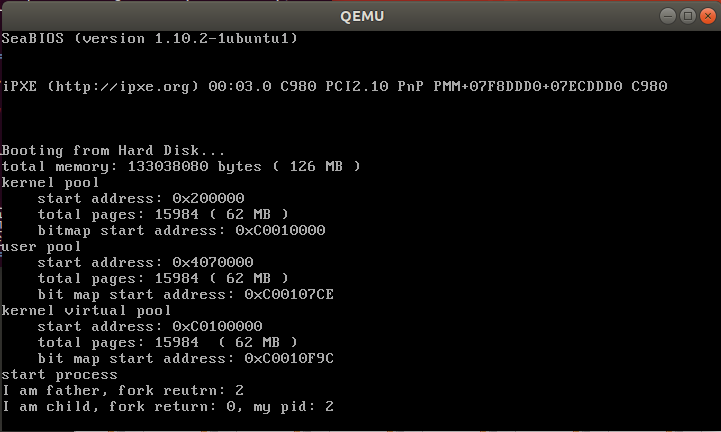
}

}

asm\_halt();

}

* 实验结果展示：通过执行前述代码，可得下图结果，



可以看到我们成功的实现了fork函数，然后我将通过gdb来追踪父进程和子进程同时来回答实验任务中的5个问题.

1.[fork函数的实现思路](#fork的实现)在实验步骤中已详细说明，简单来说就是创建一个新的PCB，然后通过共享的内核空间，将父进程的资源，栈、页目录表、数据等都复制到子进程的PCB中，然后令父子进程都从相同的返回点开始执行，同时下面概括了解决4个关键问题的解决方法

* **如何实现父子进程的代码段共享？**

使用了自己编写的系统调用来模拟一个进程，而函数的代码都是放在内核中的，同时在我们的设置下每一个进程3GB~4GB的空间是映射到内核的0-1MB空间来实现内核共享的，因此父子进程都可以访问内核空间的代码的，实现了代码的共享的

* **如何使得父子进程从相同的返回点开始执行？**

ProgramStartProcess中保存了父进程的eip，eip的内容也是asm\_system\_call\_handler的返回地址，我们通过asm\_start\_process来启动子进程，此时，asm\_start\_process的最后的iret会将上面说到的保存在0特权级栈的eip的内容送入到eip中，也就是将asm\_system\_call\_handler的返回地址送入到eip，然后子进程便可以从父进程的返回点处开始执行，这样子进程的返回顺序就变成了从asm\_system\_call\_handler到syscall\_fork，最终从fork返回。然后我们会回到用户态，也就是切换3特权级栈，3特权级栈保存了父进程在执行int 0x80后的逐步返回的返回地址，由于我们复制了父进程的3特权级栈到子进程的3特权级栈中，所以父子进程用户态的返回也是一样的。因此，父子进程的逐步返回的地址是相同的，从而实现了在执行fork后，父子进程从相同的点返回

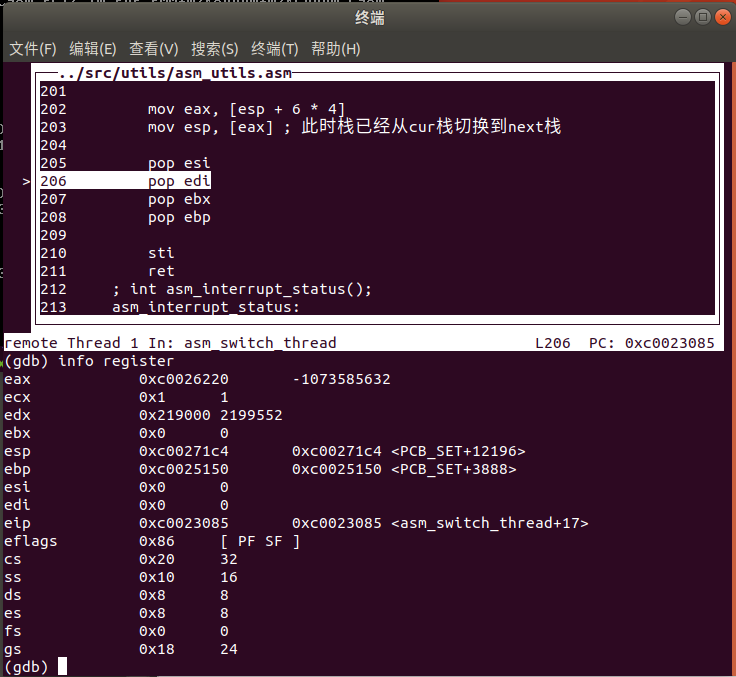
* **除代码段外，进程包含的资源有哪些？**

出了代码段之外，金宠的资源还包括0特权级栈，PCB、虚拟地址池、页目录表、页表及其指向的物理页

* **如何实现进程的资源在进程之间的复制？**

为了实现进程资源的复制，考虑到每个进程都共享着内核的空间，所以我们在内核空间创建一个中转页，首先在父进程的虚拟地址空间下将数据复制到中转页中，再切换到子进程的虚拟地址空间中，然后将中转页复制到子进程对应的位置

**2.**我们在 asm\_switch\_thread 切换到子进程的栈时设置断点，然后查看寄存器中的值，可以看到我们esp中的值为0xc00271c4



这个栈是父进程在copyProcess函数中为子进程的栈准备，内容如下：

// 准备执行asm\_switch\_thread的栈的内容

child->stack = (int \*)childpss - 7;

child->stack[0] = 0;

child->stack[1] = 0;

child->stack[2] = 0;

child->stack[3] = 0;

child->stack[4] = (int)asm\_start\_process;

child->stack[5] = 0; // asm\_start\_process 返回地址

child->stack[6] = (int)childpss;//asm\_start\_process参数

当子进程被调度执行时，子进程能够从asm\_switch\_thread跳转到asm\_start\_proces处执行，同时将为子进程创建好的0特权级栈childpss传入到asm\_start\_process进行启动.

3.接着我们使用gdb跟踪子进程的执行流程.

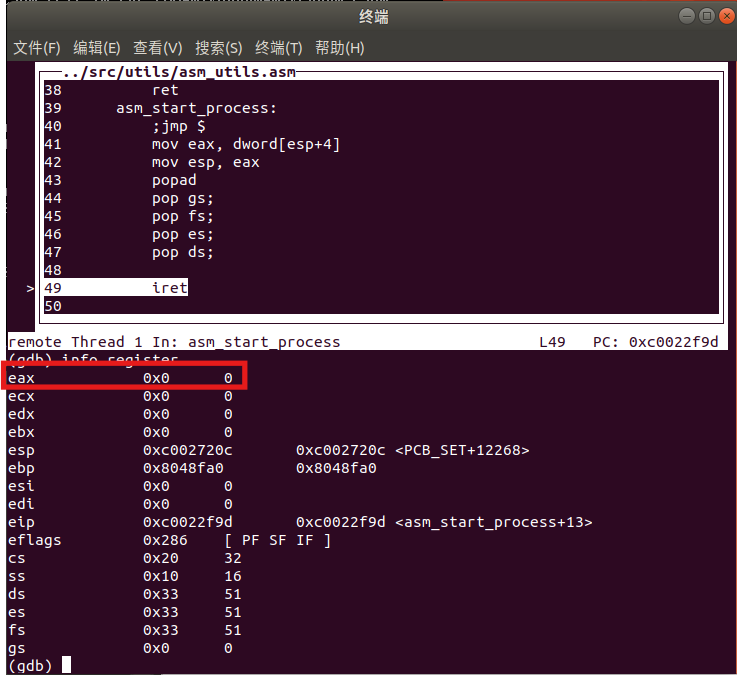
首先父进程在为子进程创建0特权级栈的时候将eax的值设置为了0，这点很重要，这决定了子进程的返回值.

// 设置子进程的返回值为0

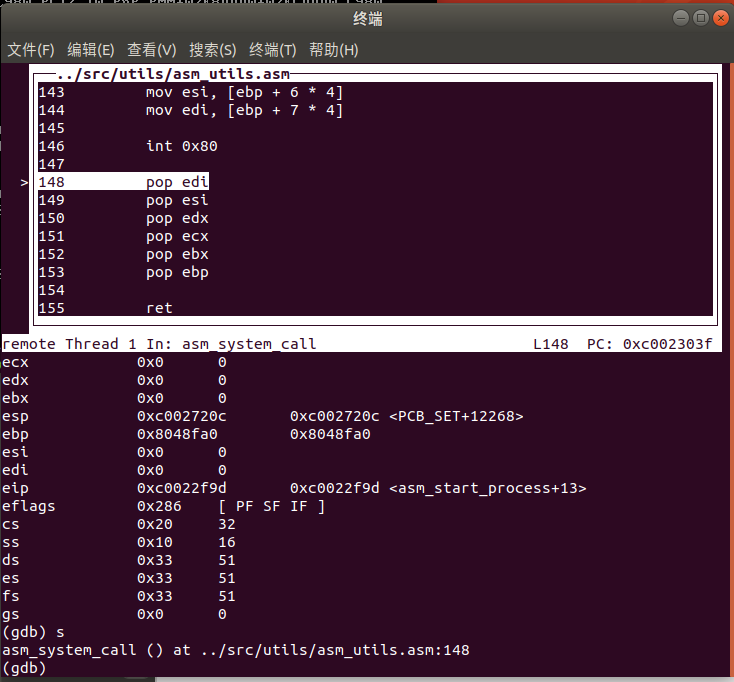
childpps->eax = 0;

由于父进程是执行int 0x80中断的时候调用fork函数的，那么父进程此时的栈就是调用int 0x80时刻的栈，也就是说eip是int 0x80的下一条指令.这也很重要.因为在创建子进程的时候，**父进程将自己此时候的栈复制到了子进程的栈中.也就是说子进程的栈中的eip是int 0x80的下一条指令.**

在asm\_switch\_thread执行ret之后，跳转到了asm\_start\_proces处，asm\_start\_proces函数将存放在childpss这个启动栈值全部都pop出来，然后执行iret实现进程的启动.段寄存器也都会切换到3特权级栈.

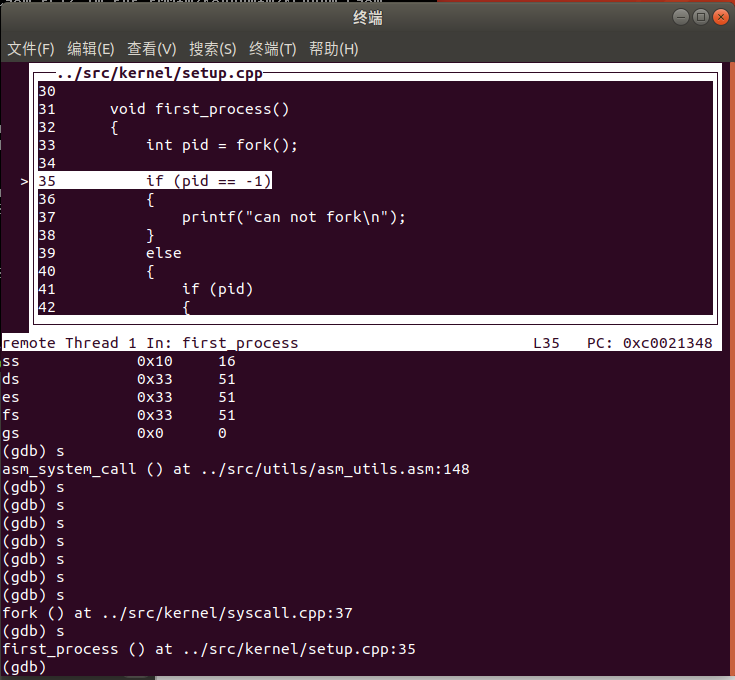
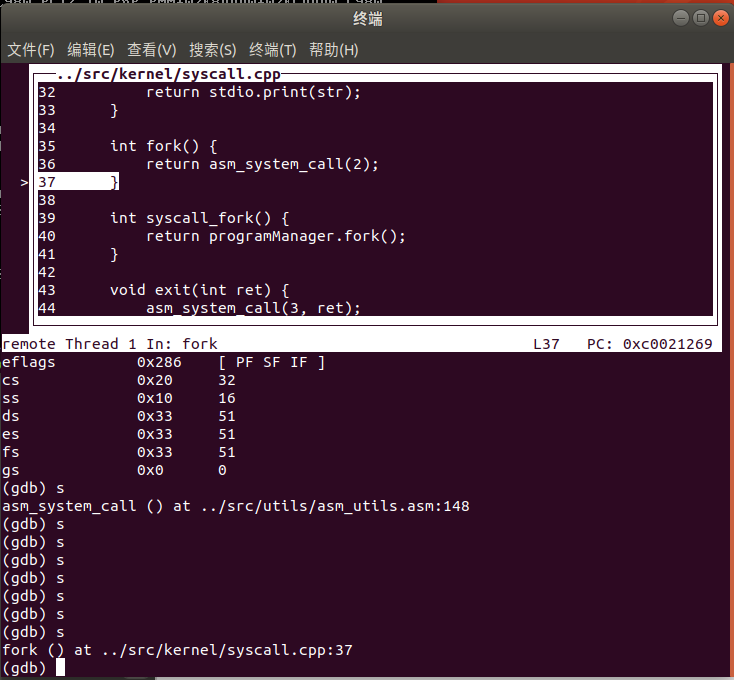


正如前面所说的，子进程栈中的eip是int 0x80的下一条指令的地址，所以在asm\_start\_proces执行iret返回的时候，子进程就自然而然的跳转到了int 0x80的下一条指令的位置，**注意此时我们的eax寄存器的值是从childpss中pop出来的，所以还是0，**如下图所示：此时的栈此时的段寄存器是内核态的.



然后继续执行，当执行ret之后，就会返回到调用系统调用的函数处，也就是fork函数.然后我们的返回值是存放在eax寄存器的值，这也是为什么我们子进程的返回值是0的原因.

下图我们的子进程依次返回到fork和first\_process处



4.子进程的返回值为0在问题3的解答中已经详细的解释过了，其实就是我们在创建子进程的栈的时候将eax寄存器设置为了0，而子进程启动运行的时候是从父进程中断点开始的，也就是int 0x80之后开始的，然后在之后我们都没有修改过eax寄存器中的值，而eax是用于函数返回的，所以子进程的返回值为0.

// 设置子进程的返回值为0

childpps->eax = 0;

而对于父进程来说，是按照正常的代码逻辑来执行的.在父进程中，父进程的fork函数为子进程创建了一个PCB，然后调用函数copyProcess复制自己的资源和栈到子进程中.在fork的最后返回了子进程的pid，然后一步一步的返回这个子进程的pid一直返回到first\_process中.所以我们的父进程返回的值是子进程的pid.

// 创建子进程

int pid = executeProcess("", 0);

5.schedule主要是通过activateProgramPage中的asm\_update\_cr3函数将我们输入的页目录表地址放入到CPU的cr3寄存器来实现进程之间虚拟地址空间的转换，我们只要切换了页目录表和页表，这样即使相同的虚拟地址通过映射之后也会映射到不同的物理地址，所以我们只需要改变cr3中的页目录表的地址就可以切换到了其他的地址空间，实现虚拟地址空间的转换.

------------------------- **实验任务4** -------------------------

* 任务要求：

1. 请结合代码逻辑和具体的实例来分析 exit 的执行过程。

2. 请解释进程退出后能够隐式调用 exit 的原因(从栈的角度分析)

3. 请结合代码逻辑和具体的实例来分析 wait 的执行过程。

4. 如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为 DEAD 开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法

* 思路分析：

exit和wait是两个系统调用，exit是进程或者线程主动结束自己的运行，而wait则是回收已结束的进程的PCB，也就是把标记为DEAD的进程的PCB进行回收，如果不进行回收的话，就会永久的占用内存，已经CPU的功能.

实现exit函数就是要释放掉除了PCB以外的所有进程或者线程的资源，包括页表、物理页、虚拟地址池、页目录表、位图等，然后再马上调度下一个进程进行执行.

实现wait，我们就需要设置一个循环直到找到一个DEAD进程，然后释放掉其PCB的空间，或者没有能够回收的进程，那么也会退出循环.

* 实验步骤：

1. **实现exit函数**

**Step1:首先实现exit系统调用和系统调用处理函数的定义和实现**

// 第3个系统调用, exit

void exit(int ret);

void syscall\_exit(int ret);

struct PCB

{

...

int retValue; // 返回值

};

**Step2：实现exit函数：**

首先我们将调用exit函数的进程的状态修改为DEAD，然后将传入的ret值放入PCB中的ret，然后我们依次释放掉进程所占用的物理页、页表、页目录表、虚拟地址池、位图等资源，最后我们调度下一个进程.

void ProgramManager::exit(int ret)

{

// 关中断

interruptManager.disableInterrupt();

// 第一步，标记PCB状态为`DEAD`并放入返回值。

PCB \*program = this->running;

program->retValue = ret;

program->status = ProgramStatus::DEAD;

int \*pageDir, \*page;

int paddr;

// 第二步，如果PCB标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池bitmap的空间。

if (program->pageDirectoryAddress)

{

pageDir = (int \*)program->pageDirectoryAddress;

for (int i = 0; i < 768; ++i)

{

if (!(pageDir[i] & 0x1))

{

continue;

}

page = (int \*)(0xffc00000 + (i << 12));

for (int j = 0; j < 1024; ++j)

{

if(!(page[j] & 0x1)) {

continue;

}

paddr = memoryManager.vaddr2paddr((i << 22) + (j << 12));

memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);

}

paddr = memoryManager.vaddr2paddr((int)page);

memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);

}

memoryManager.releasePages(AddressPoolType::KERNEL, (int)pageDir, 1);

int bitmapBytes = ceil(program->userVirtual.resources.length, 8);

int bitmapPages = ceil(bitmapBytes, PAGE\_SIZE);

memoryManager.releasePages(AddressPoolType::KERNEL,

(int)program->userVirtual.resources.bitmap, bitmapPages);

}

// 第三步，立即执行线程/进程调度。

schedule();

}

在线程的实现中，线程在退出的时候可以自动调用program\_exit返回，同样的在实现了exit后我们的进程也可以在退出的时候自动调用我们的exit函数实现退出，我们只要在进程的3特权级栈的顶部放入exit的地址和参数即可，当执行进程的函数退出后就会主动跳转到exit，这就是我们实验任务中第2个问题的答案。

为此，我们需要修改load\_process

void load\_process(const char \*filename)

{

...

// 设置进程返回地址

int \*userStack = (int \*)interruptStack->esp;

userStack -= 3;

userStack[0] = (int)exit;

userStack[1] = 0;

userStack[2] = 0;

interruptStack->esp = (int)userStack;

...

}

**Step3：测试exit函数**

在setup中编写了如下的函数测试我们的wait函数

void first\_process()

{

int pid = fork();

if (pid == -1)

{

printf("can not fork\n");

asm\_halt();

}

else

{

if (pid)

{

printf("I am father\n");

asm\_halt();

}

else

{

printf("I am child, exit\n");

}

}

}

1. **实现wait函数**

**Step1：：首先实现wait系统调用和系统调用处理函数的定义和实现**

int wait(int \*retval) {

return asm\_system\_call(4, (int)retval);

}

int syscall\_wait(int \*retval) {

return programManager.wait(retval);

}

**Step2：实现wait函数**

wait的实现实际上是由ProgramManager来完成的

首先，我们试图在allPrograms中找到一个状态为DEAD的子进程，allPrograms中包含了所有状态的所有进程和线程，如果我们找到了一个可回收的子进程，我们取出子进程的返回值放入到retval指向的变量中。然后取出子进程的pid，调用releasePCB来回收子进程的PCB，最后返回子进程的pid，如果没有找到符号要求的子进程，则返回-1，如果，存在子进程但子进程的状态不是DEAD，因此我们执行调度。主要到我们是死循环，当父进程调用wait后，如果存在子进程但子进程的状态不是DEAD，则父进程会被阻塞，即wait不会返回直到子进程结束，具体代码如下：

int ProgramManager::wait(int \*retval)

{

PCB \*child;

ListItem \*item;

bool interrupt, flag;

while (true)

{

interrupt = interruptManager.getInterruptStatus();

interruptManager.disableInterrupt();

item = this->allPrograms.head.next;

// 查找子进程

flag = true;

while (item)

{

child = ListItem2PCB(item, tagInAllList);

if (child->parentPid == this->running->pid)

{

flag = false;

if (child->status == ProgramStatus::DEAD)

{

break;

}

}

item = item->next;

}

if (item) // 找到一个可返回的子进程

{

if (retval)

{

\*retval = child->retValue;

}

int pid = child->pid;

releasePCB(child);

interruptManager.setInterruptStatus(interrupt);

return pid;

}

else

{

if (flag) // 子进程已经返回

{

interruptManager.setInterruptStatus(interrupt);

return -1;

}

else // 存在子进程，但子进程的状态不是DEAD

{

interruptManager.setInterruptStatus(interrupt);

schedule();

}

}

}

}

PCB的回收是通过releasePCB函数来实现的，实现如下：

void ProgramManager::releasePCB(PCB \*program)

{

int index = ((int)program - (int)PCB\_SET) / PCB\_SIZE;

PCB\_SET\_STATUS[index] = false;

this->allPrograms.erase(&(program->tagInAllList));

}

对于线程的回收，直接在调度的时候将状态为DEAD的线程的PCB回收即可.

void ProgramManager::schedule()

{

...

else if (running->status == ProgramStatus::DEAD)

{

// 回收线程，子进程留到父进程回收

if(!running->pageDirectoryAddress) {

releasePCB(running);

}

...

}

**Step3：测试wait函数**

在setup中编写如下的测试函数：

void first\_process(){

int pid = fork();

int retval;

if (pid)

{

pid = fork();

if (pid)

{

while ((pid = wait(&retval)) != -1)

{

printf("wait for a child process, pid: %d, return value: %d\n", pid, retval);

}

printf("all child process exit, programs: %d\n",

programManager.allPrograms.size());

asm\_halt();

}

else

{

uint32 tmp = 0xffffff;

while (tmp)

--tmp;

printf("exit, pid: %d\n", programManager.running->pid);

exit(123934);

}

}

else

{

uint32 tmp = 0xffffff;

while (tmp)

--tmp;

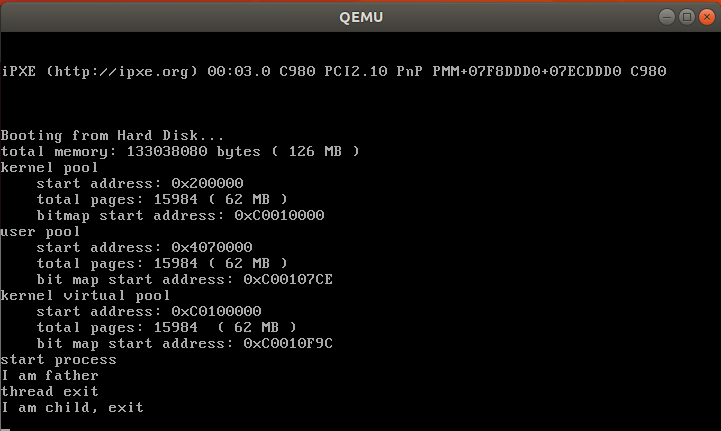
printf("exit, pid: %d\n", programManager.running->pid);

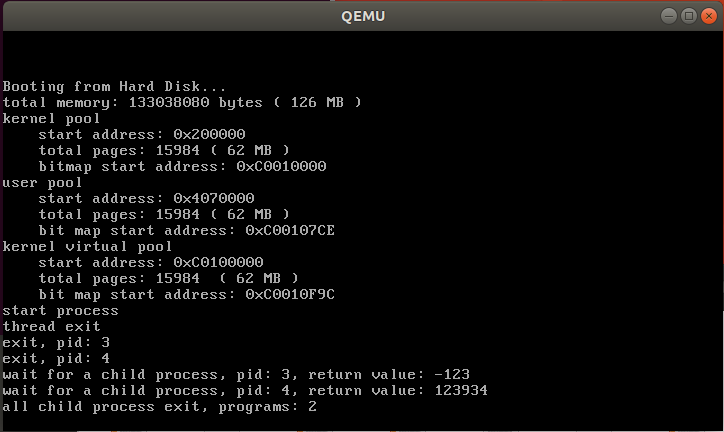
exit(-123);

}

}

* 实验结果展示：通过执行前述代码，可得下图结果，

exit:

wait:

可以看到我们成功的实现了exit和wait的系统调用，然后我来回答下实验任务中的4个问题：

1.exit可以看成是copyProcess的逆过程，他将PCB的状态改为DEAD，然后释放其占用的资源包括虚拟地址池、物理页、位图、页目录表等资源.

[exit的代码逻辑和执行过程](#exit的实现)在实验步骤中已详细说明了，这里结合我们的测试函数作为实例来说明exit的执行过程.

测试函数首先用fork函数产生一个子进程，然后如果是父进程那么就打印I am father，对于我们的子进程就调用exit主动退出进程.

2.进程退出后能够隐式调用exit ，是因为我们**在load\_process中，将exit的函数地址和参数放入了进程的3特权级栈的顶部**，**这样当执行进程的函数退出后就会主动的返回到栈顶的地址，也就是跳转到exit**，这样就实现了exit的隐式调用

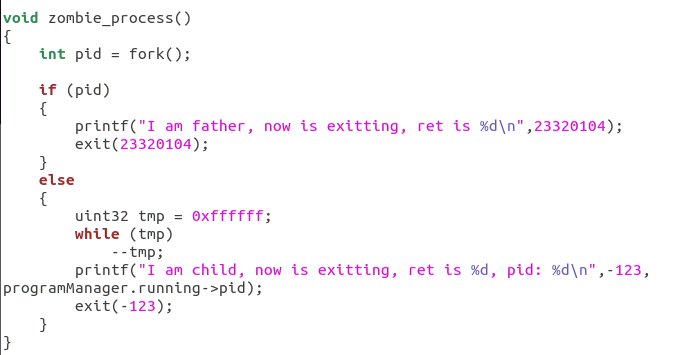
3.wait函数首先找到处于DEAD状态的PCB，然后通过releasePCB释放他的PCB内存，如果子进程还在运行，那么父进程就一直等待，等待子进程exit.

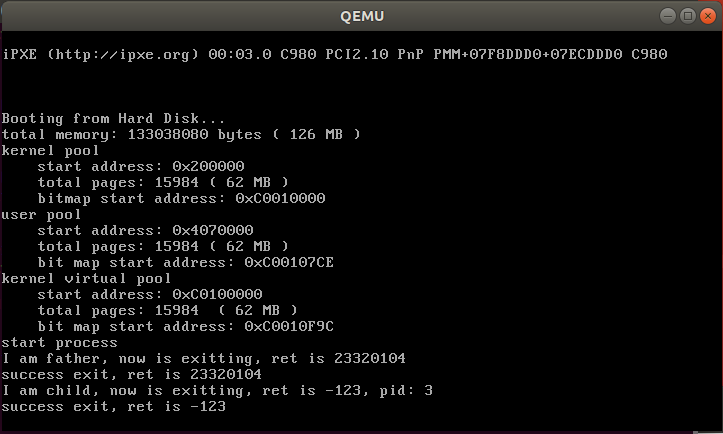
[wait的代码逻辑和执行过程](#wait的实现)在实验步骤中已详细说明了，这里结合我们的测试函数作为实例来说明wait的执行过程.

首先我们第一次fork，产生第一个子进程，然后代码进入判断pid是否为0来区分是子进程还是父进程，如果是父进程，那么再次调用fork产生第二个子进程，然后父进程执行wait，wait函数检查到所有子进程都没结束，立刻切换到下一个进程，切换到第一个子进程，第一个子进程从第一次fork处开始执行，（子进程是从调用fork处的返回点开始执行的），如果是子进程，那么直接exit主动结束子进程.然后切换到第二个子进程，第二个子进程是从第二个fork处开始执行的，同样的，第二个子进程主动exit退出进程.然后所以继续等待到两个子进程最后都执行完毕结束，开始回收，开始回收第一个，回收完之后判断还有一个，继续回收，最后退出wait，打印结束语句。

4.回收僵尸进程的办法

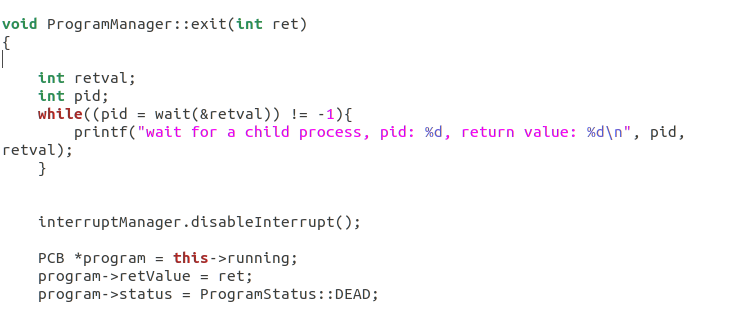
僵尸进程指的是父进程先于子进程退出，导致子进程退出后子进程的PCB没有进行回收，首先我们构造了一个僵尸进程的实例，代码和运行的结果如下：





可以看到我们的父进程23320104先于子进程-123退出了，这导致了子进程变为了僵尸进程.

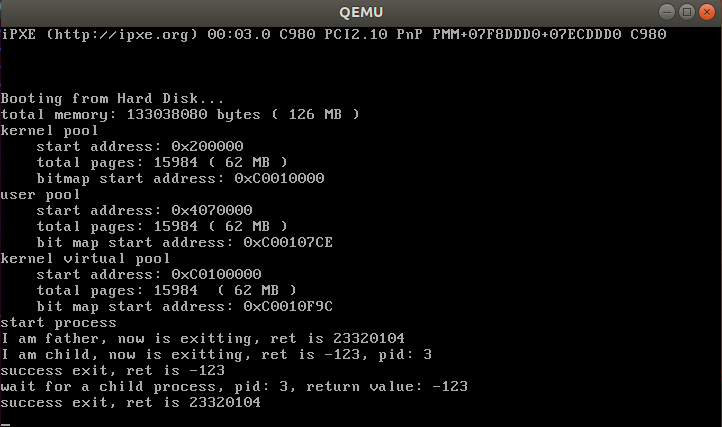
为了解决这个问题，我们可以修改下进程的退出函数exit，如果父进程要退出的话，再退出之前，先让父进程检查下是否还有子进程没回收或者还在运行，只有检查到所有的子进程都wait了之后，才能开始父进程后面的退出.对于判断父子进程的方法，我们可以检查退出进程的PCB中的parentpid是否为0来判断，实现代码如下：



代码的逻辑比较的简单，就是在进程退出的时候调用wait函数，如果所有的子进程的PCB都被回收的话，这个时候wait就会返回-1，所以这个就是我们退出while循环的标志.当我们退出while循环的时候这就说明该父进程的所有子进程的PCB都被回收了，这个时候就可以开始执行父进程的退出了.

如果是子进程要退出，那么wait函数里面的逻辑是能够直接返回-1的.这样也不会影响其他子进程的推出.

我们再次运行上面那个僵尸进程的例子：



可以看到我们有效的实现了僵尸进程的回收.可以看到father进程要退出的时候，并没有马上的exit成功，而是等待了他的子进程退出，并wait了他的PCB我们的父进程23320104才成功的退出.

**Section 5 实验总结与心得体会**

本次实验首先学习到了特权级的概念，然后操作系统根据特权级分成了内核态和用户态，同时为了使得用户态能够使用到内核态的一些功能，本次实验首先实现了系统调用.

在实现了系统调用的基础上，本次实验接着实现了用户的进程，用户进程于线程最大的不同就是进程有着属于自己的虚拟地址空间和，由于我们创建进程是属于内核态，为了实现用户态的进程，我们必须通过load\_process函数构造出一个低特权级的栈，实现高特权级向低特权级的转移，从而实现进程的启动.

实现完了进程之后，本次实验又学习到了一些与进程有关的系统调用和实现，如fork、exit、wait，fork函数用于进程创建一个和自己几乎一模一样的子进程，父子进程从相同的地址返回，而exit函数则实现了进程的退出，wait函数实现了PCB的回收.同时本次实验还提出了一种有效回收僵尸进程的办法.

最后，本次关于内核态和用户态的实验，不仅让我更加深刻的理解了操作系统保护模式下的内核态和用户态的区别，同时也更加深刻了解了系统调用的知识以及系统调用的实现方法，同时也对进程和线程的区别和联系有了更加清晰的认知.

**结语：**最后的最后，本次实验作为本学期操作系统的最后一个实验，回看本学习所学习的8个实验，收获和感想都颇多，从Linux虚拟机的配置到混合编程，到内存管理，再到进程管理和用户态和内核态的实现，跟着实验书的指导，我们一步步自己搭建了一个简易的操作系统内核，虽然简陋但是却基本实现了操作系统的功能.

实验指导书的内容很详尽，实验任务设置都也很循循善诱，只是很可惜本学期没有更多的时间去钻研实验指导书上的每一个过程和细节，希望在未来的时间里可以学习到更多有关于操作系统的知识.最后这里想要特别感谢感谢编写指导书的老师和学长的指导，以及老师和助教的帮助和辛勤付出！

**Section 6 附录：代码清单**

由于代码文件较多， 代码量较大， 为了篇幅的精简， 具体代码以附件的形式给出