

操作系统实验4

# **从内核态到用户态**

实验课程: 操作系统原理实验

# 实验名称: 从内核态到用户态

专业名称: 计算机科学与技术

学生姓名: 钟旺烜

学生学号: 23336342

实验地点: 实验楼B203

实验成绩:

报告时间: 2025年6月11日

**Section 1 实验概述**

## **实验任务1：系统调用**

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

* 展现系统调用执行结果的正确性，结果截图并并说说你的实现思路。
* 请根据gdb来分析执行系统调用后的栈的变化情况。
* 请根据gdb来说明TSS在系统调用执行过程中的作用。

**实验任务2：Fork的奥秘** 实现fork函数，并回答以下问题:

* 请根据代码逻辑和执行结果来分析fork实现的基本思路。
* 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。
* 请根据代码逻辑和gdb来解释fork是如何保证子进程的fork返回值是0，而父进程的fork返回值是子进程的pid。

## **实验任务3：哼哈二将 wait & exit**

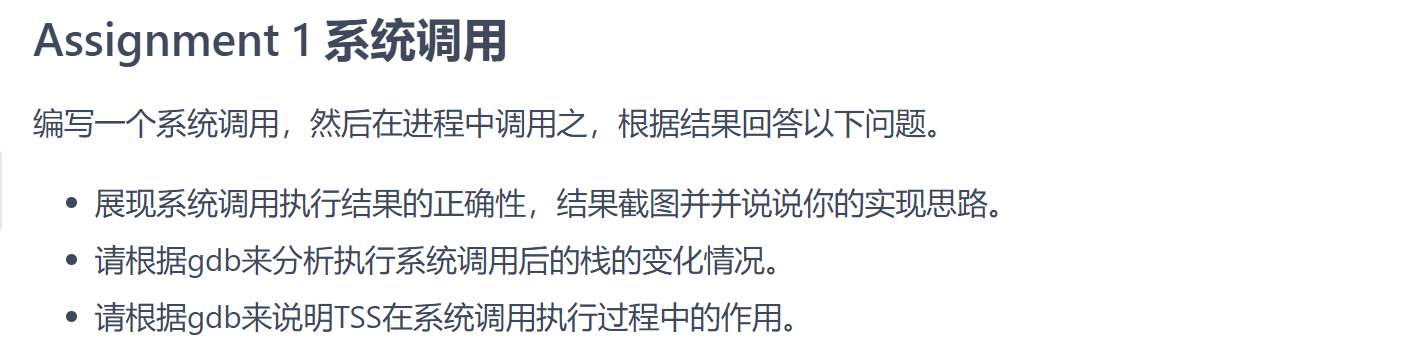
实现wait函数和exit函数，并回答以下问题。

* 请结合代码逻辑和具体的实例来分析exit的执行过程。
* 请分析进程退出后能够隐式地调用exit和此时的exit返回值是0的原因。
* 请结合代码逻辑和具体的实例来分析wait的执行过程。
* 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。

**Section 2 实验步骤与实验结果**

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

任务要求：



思路分析：

实验指导中已经给出了实现系统调用以及创建进程的代码，将代码复制到虚拟机上运行即可。

在此次实验任务中，我编写了一个使用printf输出字符的系统调用，命名为系统调用syscall\_1，其系统调用号为1。可以在进程中使用asm\_system\_call()函数调用系统调用号为1的函数，以int类型传递字符串参数，便可实现printf函数输出字符的功能。

本次实验主要任务为使用gdb探究系统调用前后栈的变化情况以及TSS在系统调用执行过程中的作用。

实验步骤：

首先实现能够在进程中使用printf打印字符的系统调用函数syscall\_1()。以实验指导中实现系统调用函数syscall\_0()为例，syscall\_1()的函数设计代码如下所示：代码保存在setup.cpp中

void syscall\_1(const char \*str)

{

printf("%s\n",str);

}

随后，使用以下代码将此系统调用函数加载进系统调用，设置其系统调用号为1。 systemService.setSystemCall(1, (int)syscall\_1);

至此，我成功的添加了一个新的系统调用。

我们修改第一个进程函数的代码，调用新添加的系统调用，修改后第一个进程的代码如下：

void first\_process()

{

asm\_system\_call(1,(int)"Hellon,World!");

asm\_halt();

}

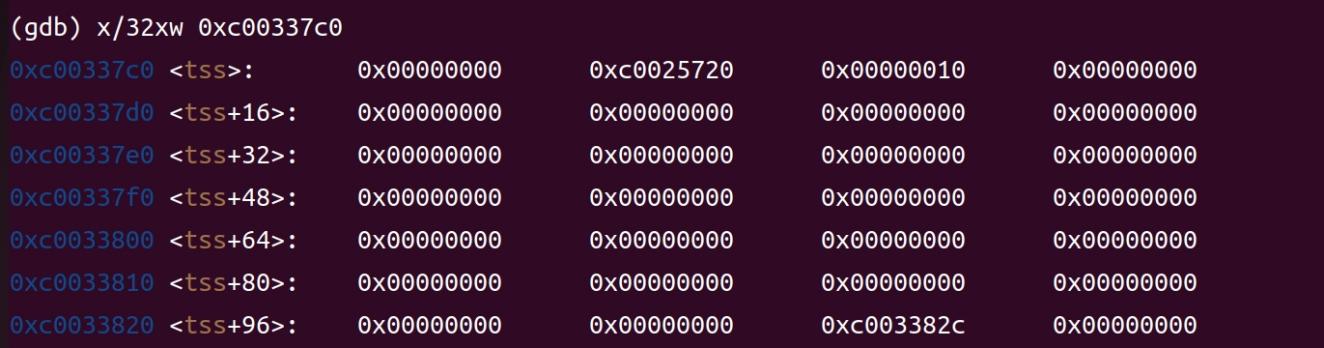
第一个线程函数与实验指导中的一样，创建三个first\_process()

在虚拟机中使用terminal输入make相关指令对操作系统进行编译运行即可观察到三个进程进行系统调用的结果。

下面是使用gdb调试观察执行系统调用前后栈的变化情况以及TSS在系统调用执行中的作用的内容:

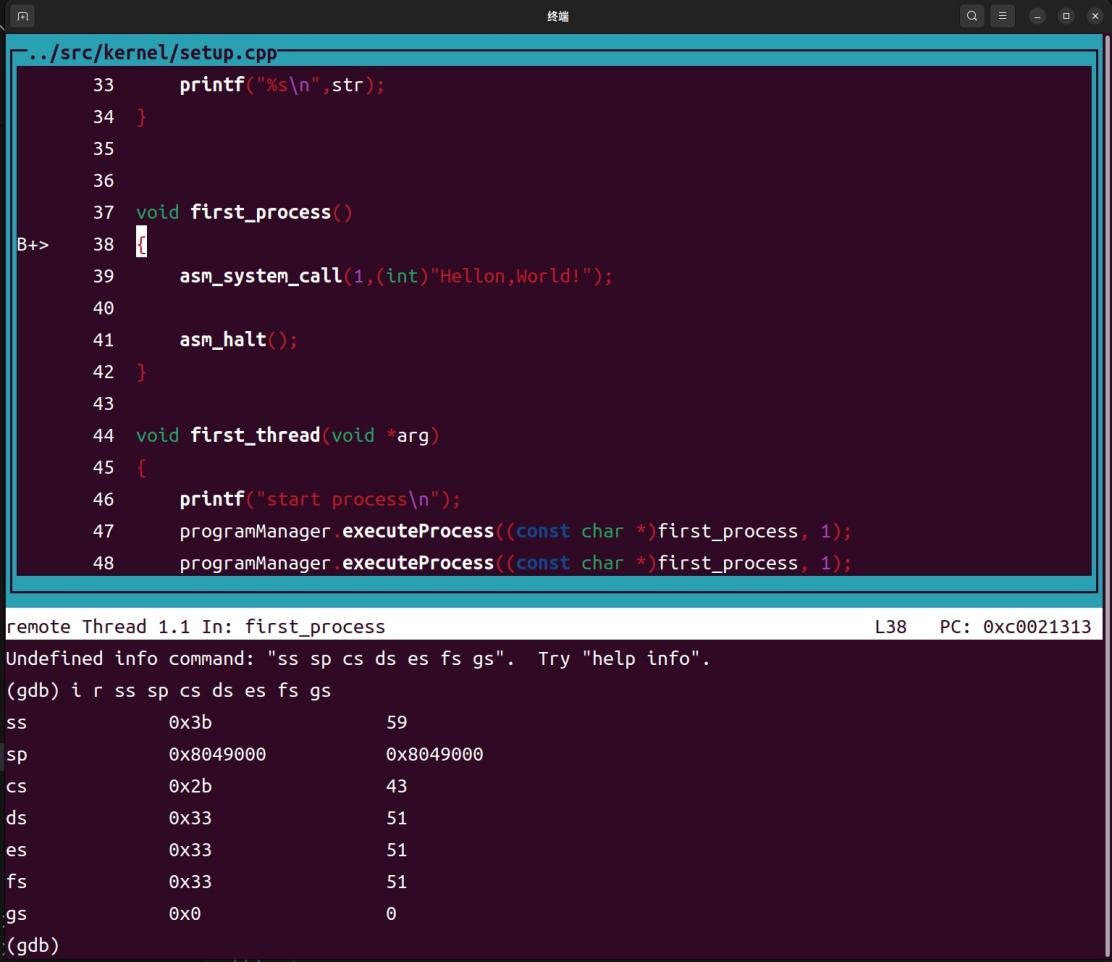
首先，当我们的进程放上处理器上运行前，首先会执行activateProgramPage()函数，此函数会对TSS的esp0进行更新，根据代码可知，更新后的esp0会指向到当前进程PCB的顶部，由此可知每一个进程的esp0，也就是特权级0的栈实际上保存在PCB中，接下来我们可以看到，当进入系统调用后，CPU会从TSS中读取esp0和ss0的值放入esp和ss寄存器中。实现特权级转换时，特权级的栈的切换。

在加载第一个进程进入系统调用前，下面是TSS中存储的内容：



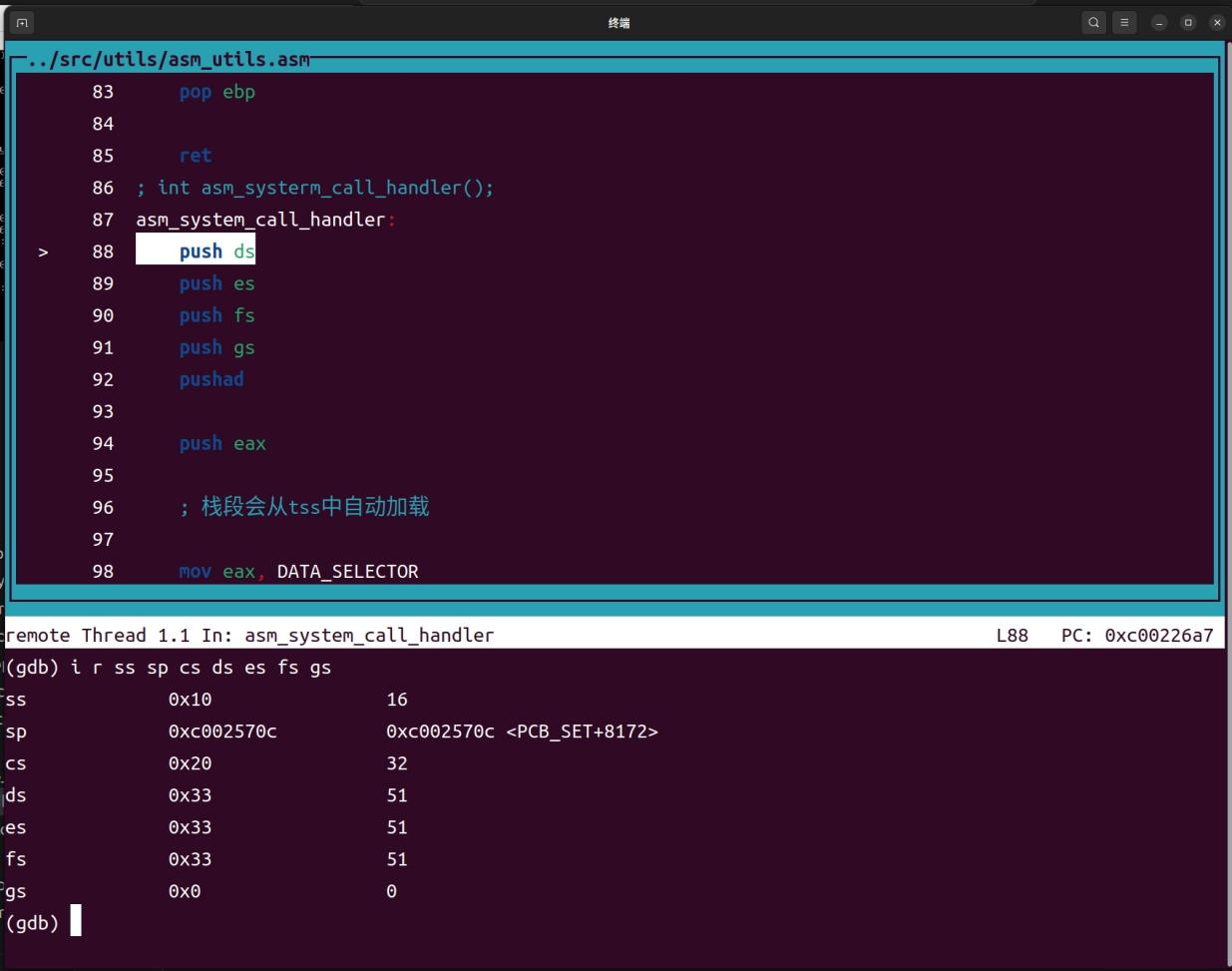
可以看到，esp0的值为0xc0025720,这正好是第一个进程所在的PCB的顶部的地址。而ss0的值是我们在初始化TSS设置好的STACK\_SELECTOR，即特权级0下的栈段选择子，值为0x10。

随后在进入系统调用前，下面是栈esp，以及各个段选择子对应储存的值：

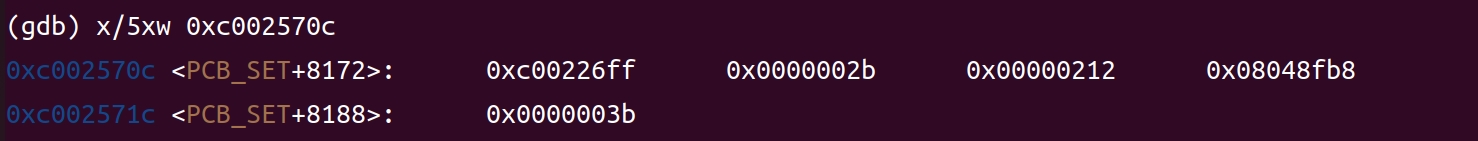


可以看到，现在处理器正处于特权级3(用户态)状态下。此时sp指针（也就是esp指针）正指向在加载进程时进程虚拟地址空间中分配一页创建的特权级3栈，其地址便是虚拟用户起始地址USER\_VADDR\_START = 0x8048000分配的第一页的页顶处。而各个段选择子储存的值也为特权级3(用户态)状态下各个段选择子信息。

下面当进入系统调用函数后，栈esp，以及各个段选择子对应储存的值如下所示：



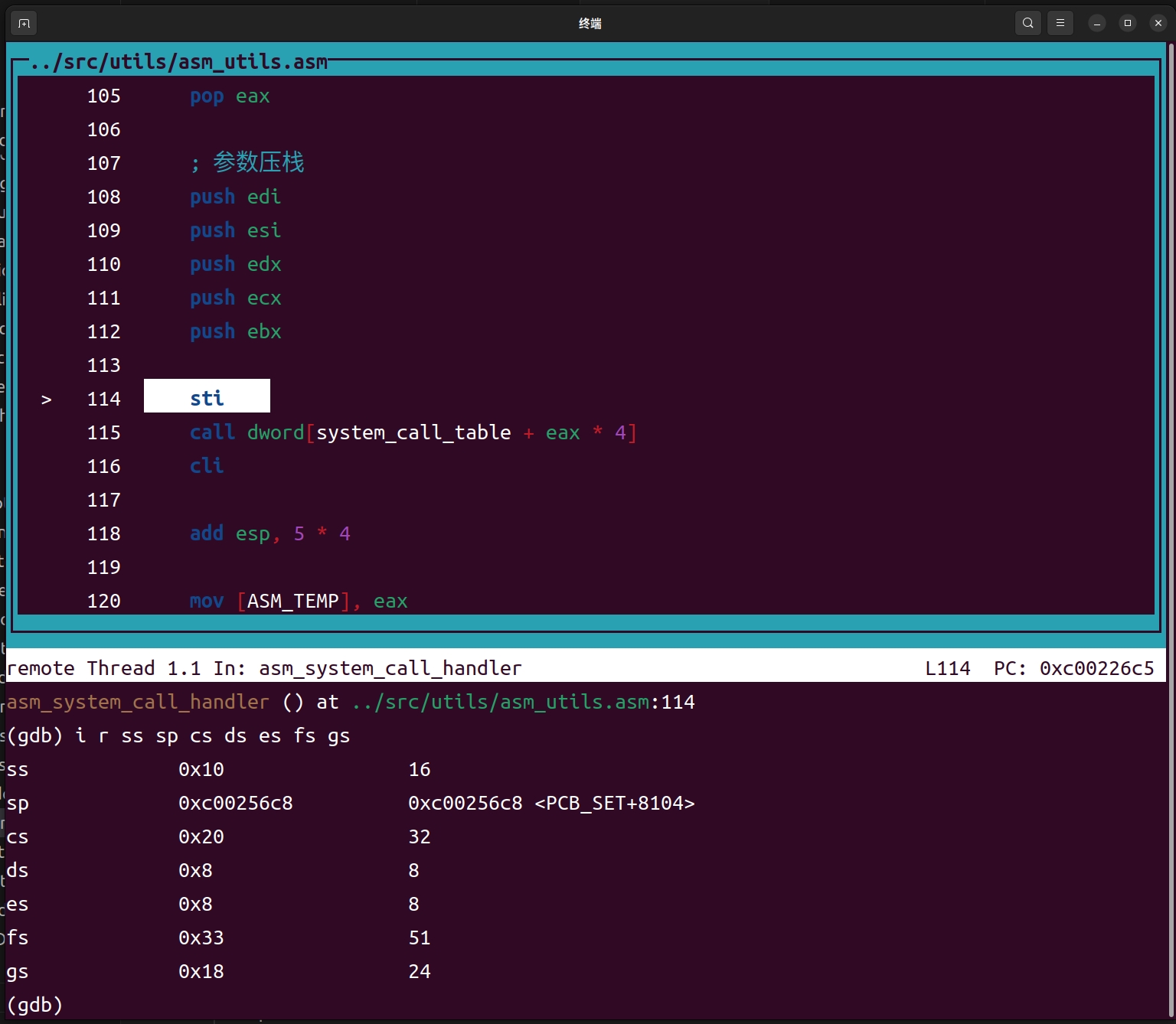
可以看到，当进入asm\_system\_call\_handler函数时，我们成功通过0x80中断从特权级3(用户态)到特权级0(内核态)转移，此时，CPU自动读取TTS中esp0和ss0的内容，并将其加载到(e)sp和ss寄存器上。将中断发生前的SS，ESP，EFLAGS、CS、EIP依次压入高特权级栈。所以在gdb的调试结果中我们可以看到：(e)sp的值为0xc002570c，正好为0xc0025720 - (5 \* 4) 的结果，通过使用gdb查看地址0xc002570c - 0xc0025720 储存的值，可以看到：



与实验指导中所说的一致，中断发生前的SS，ESP，EFLAGS、CS、EIP依次被压入了高特权级栈中。

我们发现，在进入asm\_system\_call\_handler函数时，CPU并不会自动更新ds,fs,es,gs段寄存器中的值，因此我们要手动将特权级为0的对应段选择子信息加载到ds,fs,es,gs段寄存器中，加载完毕后，正式根据系统调用号进行对应的系统调用函数：

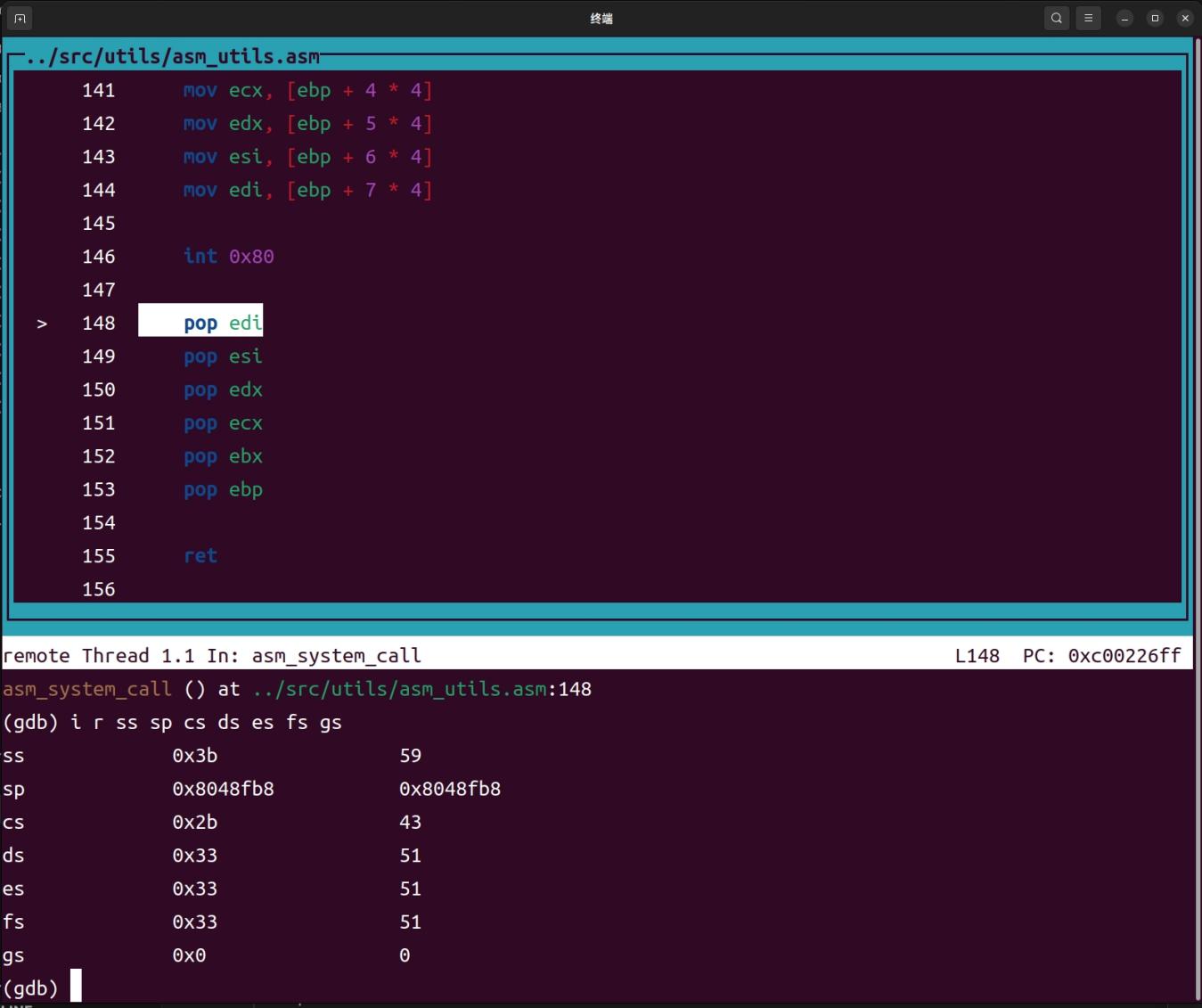
栈esp，以及各个段选择子对应储存的值如下所示



至此，我们完成了从特权级3(用户态)到特权级0(内核态)转移后栈esp，以及各个段选择子的设置，可以在特权级0(内核态)下运行系统调用函数。

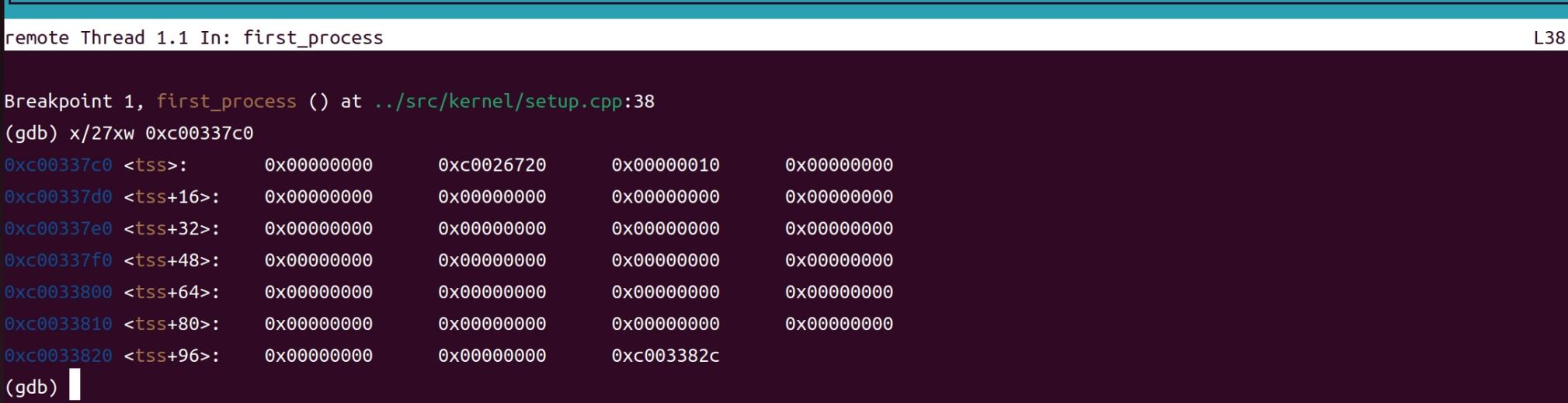
在执行完系统调用函数，恢复现场，执行iret指令实现中断返回。低特权级栈的信息在进入中断前被保存在高特权级栈中，因此执行iret后，低特权级栈的SS和ESP变可以被恢复。

在退出中断后，栈esp，以及各个段选择子对应储存的值如下所示



可以看到栈esp，以及各个段选择子对应储存的值重新置为特权级3(用户态)下对应的值。

在下一个进程被放入处理器运行时，TSS中esp0进行相应的更新，更新为新的进程的PCB顶部地址值，如下图所示：



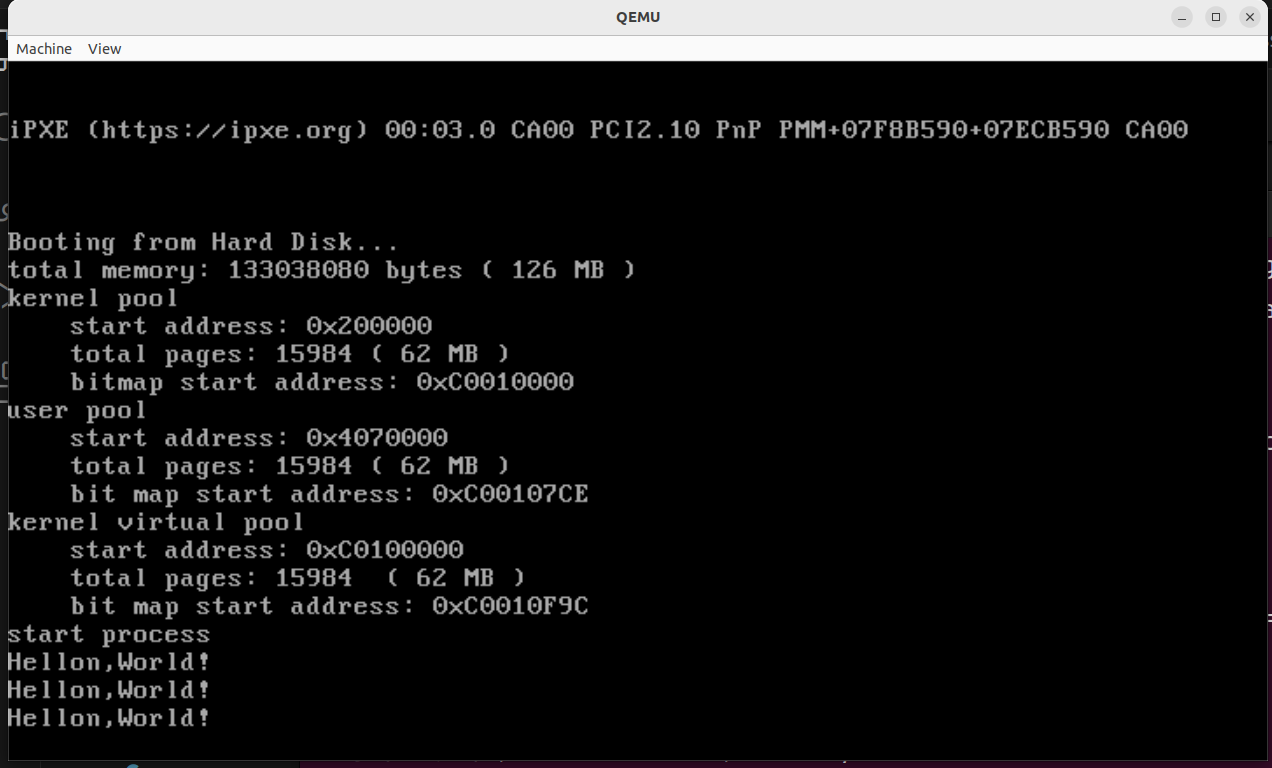
可以看到esp0的值被更新为0xc0026720,正好为第二个进程PCB的顶部地址（地址最大）

以上，便是系统调用前后栈的变化情况。从中也可以看到TSS在系统调用执行中的作用为保存将要运行的进程的esp0和ss0，以便在进程运行途中进行系统调用时，CPU能够通过TSS正确的载入esp0和ss0到esp和ss中。

实验结果展示：通过执行前述代码，可得下图结果。

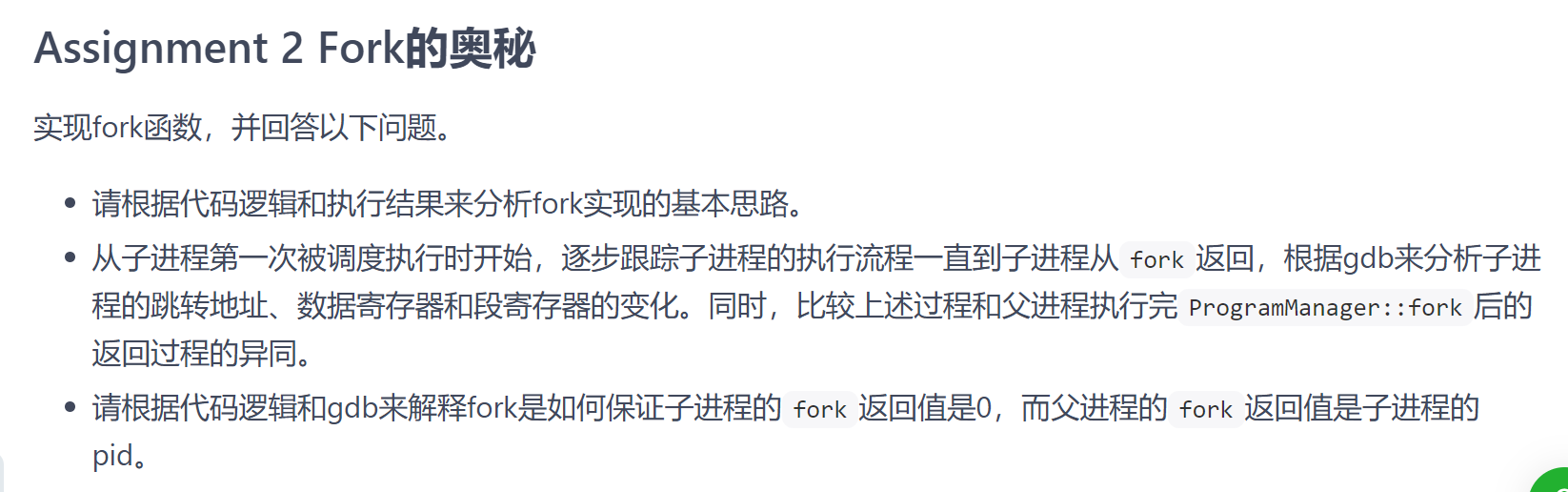
下面是我实现的系统调用函数的运行结果：

可以看到成功调用了printf函数在在qemu显示屏上打印了相应的字符



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

任务要求：



思路分析：

实验指导中给出了fork实现的4个问题。

1. 如何实现父子进程的代码段共享？
2. 如何使得父子进程从相同的返回点开始执行？
3. 除代码段外，进程包含的资源有哪些？
4. 如何实现进程的资源在进程之间的复制？

Fork实现的相关代码已经在实验指导中给出，将代码复制到虚拟机上进行编译运行即可得到运行结果，结合给出的代码的执行流程和执行结果分析fork的实现的基本思路。

使用gdb来调试运行操作系统，使用info register指令观察子进程在第一次被调度执行后一直到从fork()返回间跳转地址、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。

最后根据代码逻辑和gdb来解释fork是如何保证子进程的fork返回值是0，而父进程的fork返回值是子进程的pid。

实验步骤：

首先分析fork实现的基本思路：

在父进程使用fork()函数后，先进入系统调用，实现从特权级0(内核态)到特权级3(用户态)转移，随后调用函数programManager.fork()，进行子进程的创建操作。

fork()的主要实现在programManager.fork()函数中，下面我们分析programManager.fork()的代码来分析fork实现的基本思路。函数的代码如下:

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;

}

可以看到，ProgramManager::fork()函数先判断当前在处理器上运行的PCB是否是进程，因为线程是没有子进程的，不能进行fork()操作。判断的依据为当前运行的PCB的页目录地址是否为零。因为我们知道，只有在创建进程的时候才会给页目录地址分配页，而创建线程时并没有对页目录地址赋值。所以当PCB的页目录地址不为零时，说明当前的PCB属于进程，可以进行fork()操作。反之，说明为进程，终止fork()操作。

在确定当前PCB属于进程后，调用ProgramManager::executeProcess来创建一个子进程，随后，找到刚刚创建的子进程，然后调用ProgramManager::copyProcess来复制父进程的资源到子进程中。

ProgramManager::copyProcess是实现子进程创建最关键的函数，在此函数中，会先后完成以下复制资源的操作：

1、中断的那一刻保存的父进程0特权栈的内容复制到子进程的0特权级栈中。

2、设置子进程的PCB、复制父进程的管理虚拟地址池的bitmap到子进程的管理虚拟地址池的bitmap。

3、将父进程的页目录表复制到子进程中

4、复制父进程页表和物理页的数据到对应的子进程的页表和物理页

进行完以上操作后，我们便完成了子进程的创建工作，下面我们来结合对应的代码段逐一分析这四个步骤。

因为在实验指导中，对以上四步的操作流程以及原理以及有了比较清晰的解释，下面只对每一步操作的细节进行说明。

1. 中断的那一刻保存的父进程0特权栈的内容复制到子进程的0特权级栈中。相关代码如下:

// 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

childpps->eax = 0;

// 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

在0特权栈下，父进程所在的PCB顶部正好为0特权栈的底部，在一个ProcessStartStack大小的栈空间里保存了中断开始前对应于ProgramStartStack的内容，因此将其内容复制到子进程所在的对应的0特权栈的位置，便成功的实现了将在运行子进程时其所保持的状态同父进程进入fork()系统调用前的状态的一致性，并且在父进程的eip中正好保存的是父进程从int 0x80中断函数放回的地址，因此子进程在运行后可以直接在int 0x80中断函数放回后运行，实现了父子进程运行状态的同步。

1. 设置子进程的PCB、复制父进程的管理虚拟地址池的bitmap到子进程的管理虚拟地址池的bitmap，下面是实现的相关代码：

// 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);

这一步操作为子进程的PCB的特征值进行对应的设置，注意要设置子进程对应父进程的pid。随后对父进程bitmap进行复制，实现用户虚拟地址池的创建。

3、将父进程的页目录表复制到子进程中

4、复制父进程页表和物理页的数据到对应的子进程的页表和物理页

3和4的操作过程类似，都要使用到中转页。中转页承担了将父进程的页面数据传递给子进程的对应页面的作用，因为分页机制的实现，相同的虚拟地址对应的物理地址并不相同，而子进程还没有进行虚拟页和物理页的分配操作，如果直接使用父进程的虚拟地址进行页面复制，会访问同一块物理地址，进而不能进行页面的复制操作。

下面是申请中转页的代码：

// 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;

}

下面是分别进行页目录表复制和页表和物理页复制的代码：

// 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

}

// 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;

}

// 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

}

}

3、4步骤的流程可以概括为：

检查父进程中页表项或页面是否存在-->若存在，则进行页目录项和页面的复制

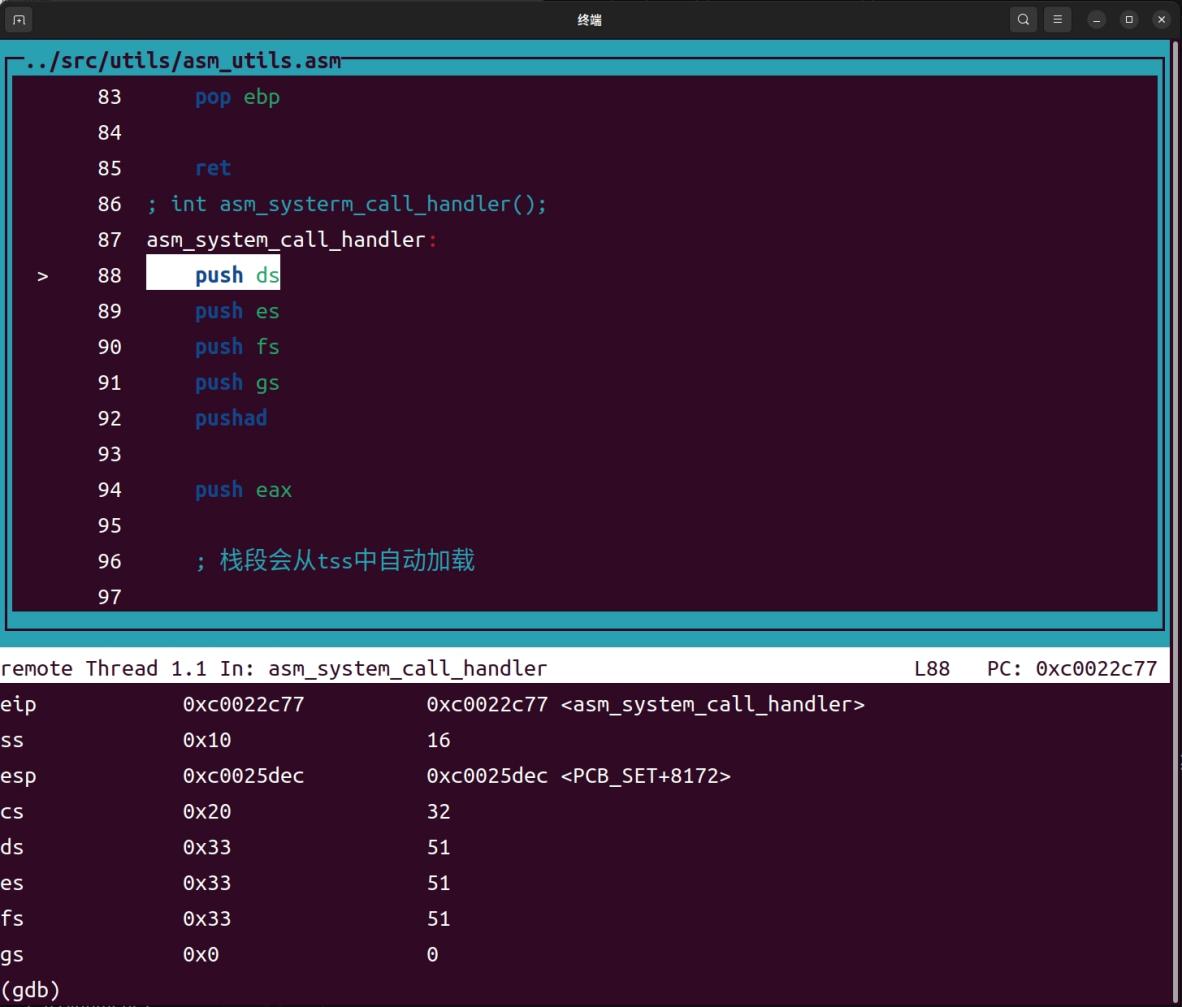
注意在进行页面复制的时候，要先将父进程页面的内容复制到中转页，再在子进程上进行页面的对应创建和复制。

注意到asm\_update\_cr3()的使用，我们知道在开启二级分页机制后，在对虚拟地址进行访问时CPU会根据保存在cr3寄存器中的页目录表地址来进行对应的物理地址寻址操作，因此在进行复制操作的时候要随时注意当前访问地址操作使用的是父进程还是子进程的页目录表地址，防止CPU的MMU单元的地址转换得到我们不想要的结果。

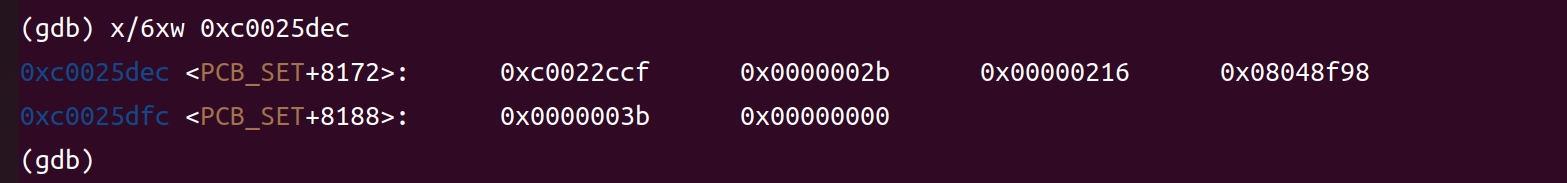
以上便是将父进程资源复制到子进程的关键操作，其他的细枝末节在实验指导中都有说明，这里就不做赘述了。

至此，我们便实现了fork()函数，下面是使用gdb调试系统来对fork()过程中父子进程的eip，数据寄存器和段寄存器所保存内容的监视与对比。

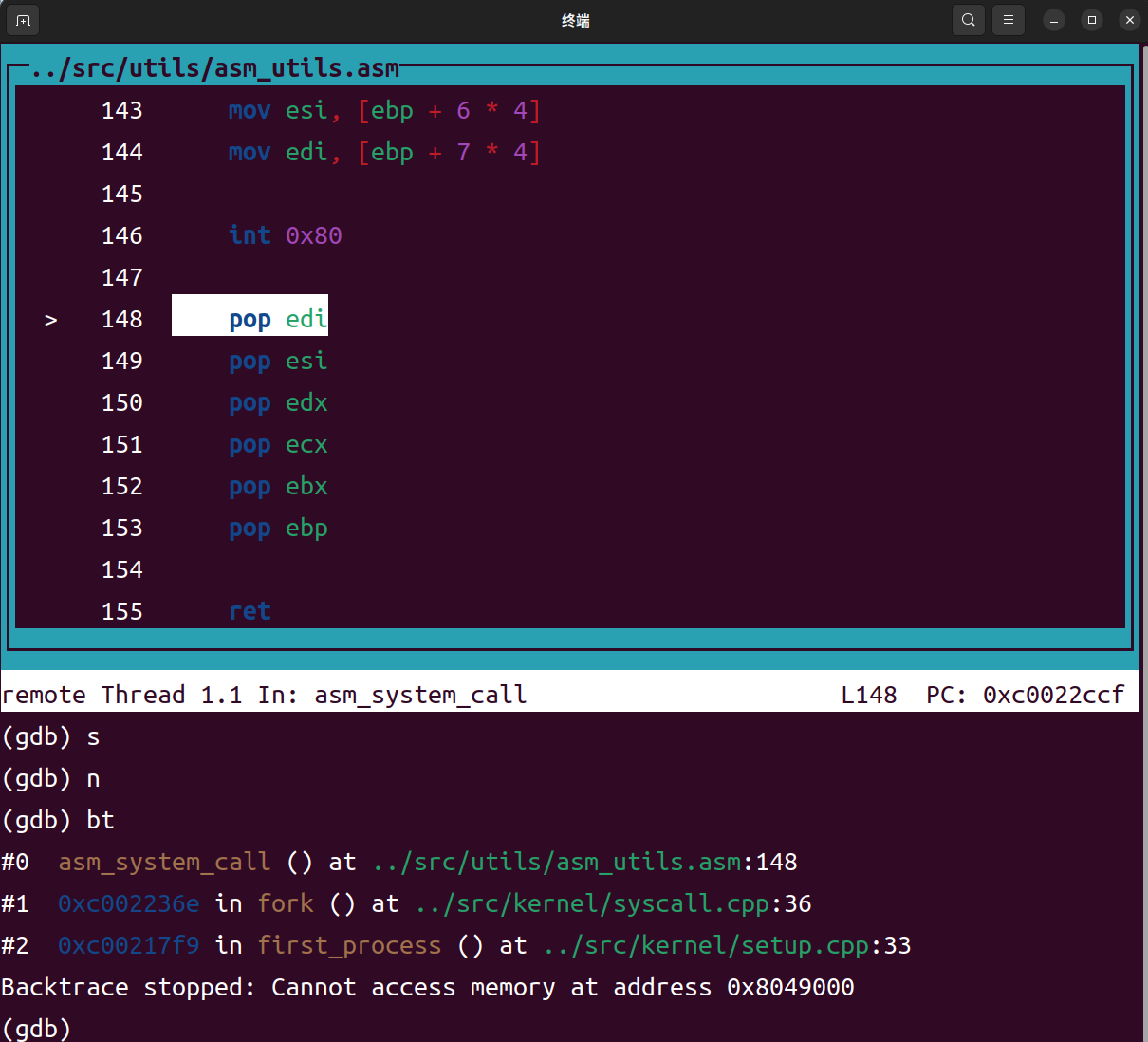
首先，在父进程进入中断后，其eip，数据寄存器和段寄存器所保存内容如下图所示：



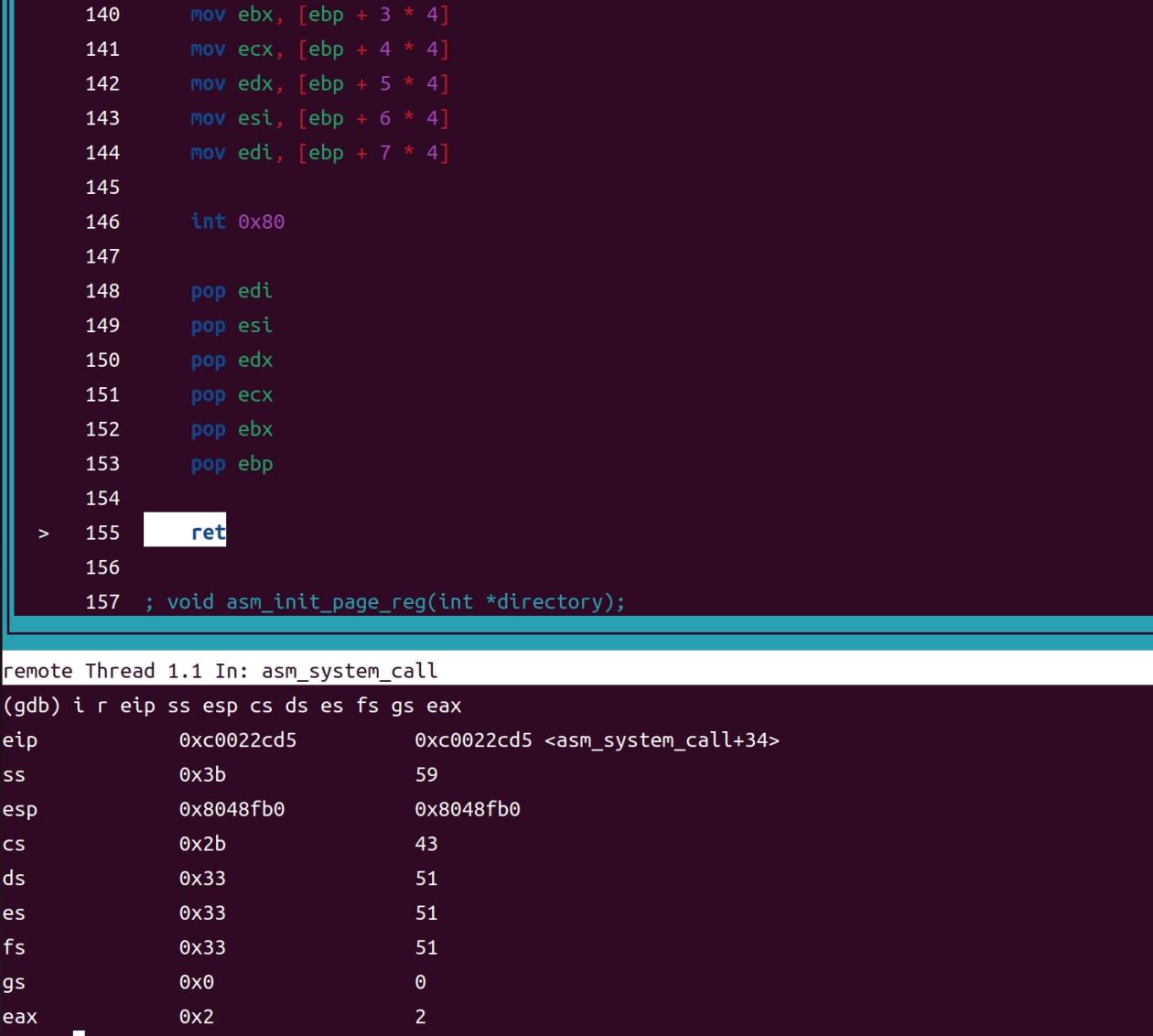
次数和我们进行系统调用时段寄存器还未我们手动更新前所得到的结果基本一致。注意到，此时父进程的中断函数的放回地址eip的值实际上已经保存到了特权级0的栈中，栈中保存的值如下所示：



根据CPU将中断发生前的SS，ESP，EFLAGS、CS、EIP依次压入高特权级栈，可以知道此时父进程的放回地址为0xc0022ccf，这个地址正好是执行完int 0x80中断指令的的位置，因此在执行完fork的系统调用函数后，放回的地址便为退出int 0x80中断指令的的位置。

接下来，我们来看父进程执行完ProgramManager::fork后的返回过程的函数调用栈的信息为：

可以看到，父进程在asm\_system\_call函数放回到fork()函数，再返回到first\_program主进程函数中。次数，父进程其eip，数据寄存器和段寄存器所保存内容如下图所示



我们主要关注到asm汇编函数放回值所放置的寄存器中eax的值为2，即子进程的pid

下面我们来看子进程在第一次被调度执行时开始的情况：

我们知道子进程的PCB设置的载入函数为asm\_start\_process函数，因此子进程被第一次被调度执行后，直接运行asm\_start\_process，加载子进程的ProcessStartStack到esp中，可以看到加载完毕后，子进程的函数调用变为下图gdb中所示结果

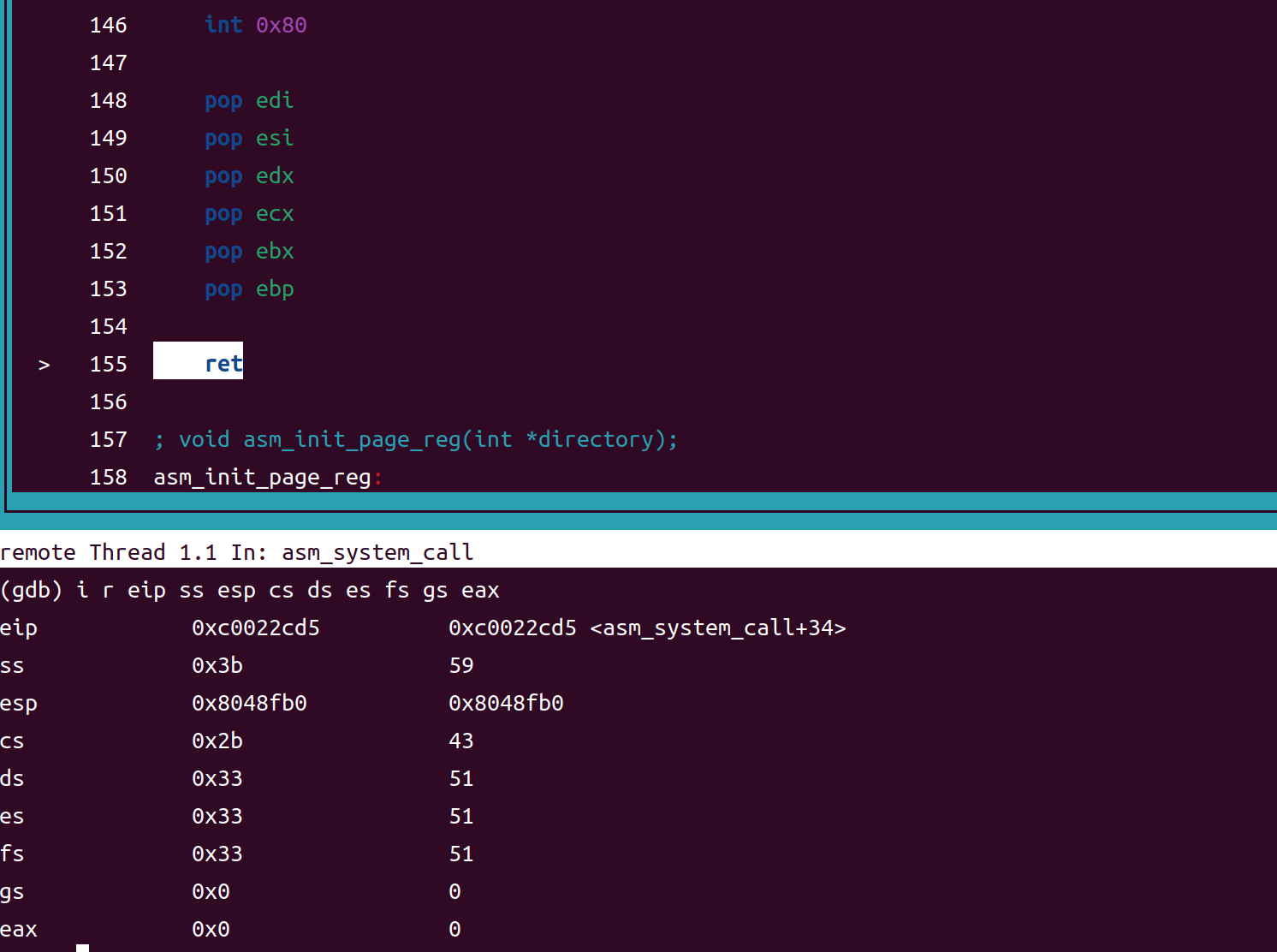


可以看到，当子进程从asm\_start\_process函数返回后，直接返回到了asm\_system\_call函数在int 0x80中断结束处的地址，这同我们复制父进程特权级0的栈中所保存的eip中的返回地址一致。



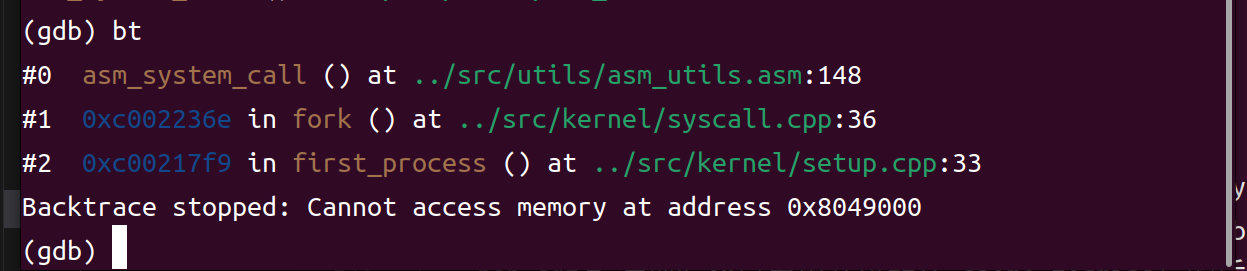
因此，子进程直接跳过了调用ProgramManager::fork系统调用函数前所有父进程进行的操作，实现了子进程同父进程的状态同步，使得父子进程从相同的返回点开始执行。

同时我们可以注意到，因为我们在复制父进程特权级0的栈到子进程的特权级0的栈的时候，我们将eax寄存器的值设置为了0，因此所以asm汇编函数返回的值都为0，如下图所示：



在子进程返回到asm\_system\_call函数后，因为子进程的3特权级栈同父进程的3特权级栈一致，而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。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。

子进程在返回asm\_system\_call函数后的函数调用栈如下图所示



至此，我们下面总结一下子进程的跳转地址、数据寄存器和段寄存器的变化和比较上述过程和父进程执行完ProgramManager::fork后的返回过程的异同：

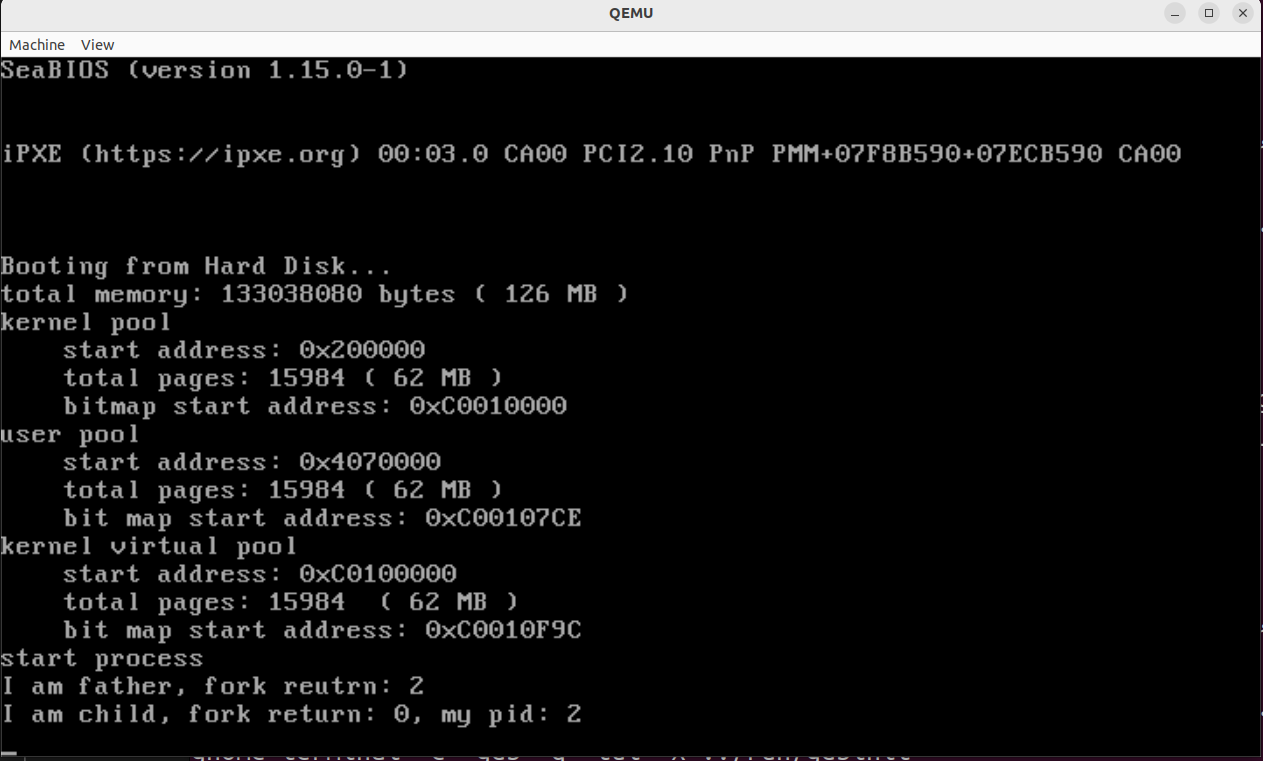
子进程在第一次被调度执行时开始直接进入asm\_start\_process函数，加载子进程的ProcessStartStack到esp中后，得到了父进程中断结束的返回地址，因此从asm\_start\_process函数返回后直接进入asm\_system\_call中，在此过程后，子进程的跳转地址、数据寄存器和段寄存器的设置同父进程一致，除了eax寄存器中所保存的函数返回值被我们手动设置为了0，确保子进程在fork()函数的返回值为0。

通过同一过程中父进程对比，可以看到在ProgramManager::fork后的返回过程中，除了eax的值不一样外，其他跳转地址、数据寄存器和段寄存器的值完全一致。

fork是如何保证子进程的fork返回值是0，而父进程的fork返回值是子进程的pid的问题也得以解决，因为子进程的fork返回值保存的eax寄存器中的值被人为修改为了0，而父进程得到的返回值就是ProgramManager::fork执行后返回的结果，即子进程的pid。

实验结果展示：通过执行前述代码，可得下图结果。

下面是运行fork操作后的展示结果：



可以看到fork操作正确执行

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

任务要求：



思路分析：

实验指导中已经给出了实现wait函数和exit函数的相关代码以及具体的实例，下面将结合代码逻辑和实例的运行结果来同时分析wait和exit的执行过程。

本次实验中，实现了ProgramManager::findProgramByPid(int pid)函数，解决了实验指导中所说的存在风险的寻找对应pid值得PCB的方法：

// 找到刚刚创建的PCB

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

替换这个提到的"存在风险的语句"，替换结果如下。

PCB \*process = findProgramByPid(pid);

同时，修改了ProgramManager中的executeProcess()函数和schedule(),实现了parentPid的初始值赋值以及状态为DEAD的父进程（或普通进程）PCB的释放，并添加了代码段实现僵尸进程PCB的释放，从而实现了将僵尸进程回收的功能。

实验步骤：

首先直接使用实验指导中给出的实例，实例代码如下所示：

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);

}

}

void second\_thread(void \*arg)

{

printf("thread exit\n");

exit(0);

}

void first\_thread(void \*arg)

{

printf("start process\n");

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

programManager.executeThread(second\_thread, nullptr, "second", 1);

asm\_halt();

}

通过分析实例代码，可以看到，线程一首先将创建进程一，再创建一个线程二到处理器上运行。

对于进程一，其在函数内进行了两次fork操作，两个子进程在创建后经过一段时间的延迟后exit(),而父进程通过wait()等待子进程的退出，并得到其子进程的返回值。

对于线程二，其在加载后直接进行exit()操作，来测试exit()函数如何处理线程的返回。

下面通过exit()的代码来分析exit()的执行过程：

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();

}

可以看到，当线程或进程进行exit()操作后，其进行以下三步操作：

第一步，标记PCB状态为`DEAD`并放入返回值。将PCB状态设置为`DEAD`为后续操作系统回收进程或线程已经结束的PCB提供了标识，返回值的放入可以得到线程或进程结束后的返回值。

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。按照创建进程时分配内存的逆过程释放进程所分配的内存以及管理内存的虚拟地址池。

第三步，立即执行线程/进程调度。进行调度后，将其他就绪的线程/进程放到处理器上执行。

以上便是exit()的执行过程。下面分析进程退出后能够隐式地调用exit和此时的exit返回值是0的原因。

我们知道，在一个函数返回时，会调用当前函数所处特权级栈的栈顶前一项的函数地址（如果有的话）作为当前函数的返回地址。因此当我们将3特权级(用户)栈的栈顶留出一部分空间，使进程执行函数前3特权级(用户)栈的栈顶的前一项为exit()函数的地址的话，我们便可以实现进程退出后能够隐式地调用exit，而且函数地址的栈上面的两项分别对应此调用函数的返回地址和调用的参数，我们将其全置为0，便可以实现exit返回值是0。

下面是实验指导中有关实现进程退出后能够隐式地调用exit和此时的exit返回值是0操作而对load\_process(const char \*filename)函数对应部分代码进行修改后的结果：

void load\_process(const char \*filename)

{

...

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

if (interruptStack->esp == 0)

{

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

process->status = ProgramStatus::DEAD;

asm\_halt();

}

interruptStack->esp += PAGE\_SIZE;

// 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;

...

}

正如我们所分析的那样，我们在进行load process操作时，往3特权级(用户)栈的栈顶处预留一小片空间，放入exit函数的地址exit函数的放回地址和接收参数，来实现进程函数返回时能调用exit()的功能。

下面通过wait()的代码来分析wait()的执行过程:

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();

}

}

}

}

wait()操作的实现过程其实很简单，当父进程使用wait()操作时，首先从allPrograms遍历所有的线程和进程，找到属于父进程的子进程，如果子进程状态为DEAD，当父进程想要知道子进程的放回值时，可以将返回值赋值给wait()函数的retval参数，随后将子进程的PCB释放，再返回子进程的pid。如果父进程的子进程已经返回，或者此“父”进程其实没有子进程，那么返回-1。如果存在子进程但此时子进程并没有exit()，则进入schedule()；调度其他线程和进程先执行。当再一次执行父进程时，父进程仍然从wait()函数中进入新一轮allPrograms的遍历，直到所有状态为DEAD子进程被释放后退出。

至此我们便解释了wait()的执行过程，下面我们来解释如何处理僵尸进程。

首先我们来看僵尸进程的特征：1、进程状态为DEAD；2、是子进程；3、父进程的状态为DEAD并且其PCB已经被释放。要实现僵尸进程PCB的释放，我们要完成对PCB是否为僵尸进程的PCB的判断逻辑。

首先，进程状态已经在PCB中储存，因此通过PCB可以轻松判断此PCB所代表进程的状态。

其次，对于如何判断一个进程为子进程，我们注意到，父进程(或普通进程)与子进程的PCB最大区别特征为parentPid的值。我们知道，当子进程在创建时，其parentPid被赋予父进程的Pid值，而父进程的parentPid为被置值。为了方便通过parentPid的值来判断进程是否为子进程，我们在创建进程时将进程的parentPid初始化为-1。

下面是实现PCB中parentPid的初始化修改函数ProgramManager::executeProcess(const char \*filename, int priority)的代码段：

这样，我们便可以判断该进程是否是子进程。

现在我们来解决判断父进程是否状态为DEAD并且其PCB已经被释放

// 创建进程的页目录表

process->pageDirectoryAddress = createProcessPageDirectory();

process->parentPid = -1;

首先因为实验指导提供的代码并没有实现父进程PCB的释放，因此下面修改schedule()函数来实现父进程（或普通进程）PCB的释放。我们上面已经实现了parentPid的初始化，所以parentPid的值为-1便说明此进程为父进程（或普通进程）。

下面是schedule()中修改的代码段：

...

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);

}

else

{

if(running->parentPid == -1)

{

releasePCB(running);

}

}

}

...

在成功实现父进程（或普通进程）PCB的释放后，下面我们通过子进程PCB中保存的parentPid到保存所有PCB的allPrograms中寻找其父进程是否存在。

我们可以通过ProgramManager::findProgramByPid(int pid)函数来实现从allPrograms中寻找对应Pid的进程PCB

下面为ProgramManager::findProgramByPid(int pid)函数的实现代码：

PCB \*ProgramManager::findProgramByPid(int pid)

{

PCB \*temp;

ListItem \*item;

bool interrupt;

item = this->allPrograms.head.next;

interrupt = interruptManager.getInterruptStatus();

interruptManager.disableInterrupt();

while (item)

{

temp = ListItem2PCB(item, tagInAllList);

if(temp->pid == pid)

{

interruptManager.setInterruptStatus(interrupt);

return temp;

}

item = item->next;

}

interruptManager.setInterruptStatus(interrupt);

return nullptr;

}

可以看到，如果在allPrograms中寻找不到对应Pid的PCB，那么放回空指针nullptr。

现在，我们便可以通过findProgramByPid()函数和parentPid来查看父进程PCB是否存在，如果findProgramByPid(parentPid)的返回结果为nullptr，说明父进程的PCB已经被释放。

以上我们解决了判断进程是否为僵尸进程的三个条件，现在我们在schedule函数中添加对应的逻辑来进行僵尸进程的判断和释放。添加的代码段如下所示：

...

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);

}

else

{

if(running->parentPid == -1)

{

releasePCB(running);

}

ListItem \*item = this->allPrograms.head.next;

while (item)

{

PCB \*temp = ListItem2PCB(item, tagInAllList);

if(temp->pageDirectoryAddress)

{

if(temp->status == ProgramStatus::DEAD && temp->parentPid != -1 &&

findProgramByPid(temp->parentPid) == nullptr)

{

printf("release zombie process: %d,return val: %d\n", temp->pid,

temp->retValue);

releasePCB(temp);

}

}

item = item->next;

}

}

}

...

以上我们便实现了回收僵尸进程的PCB。

下面我们设计一个void second\_process()，来测试我们的僵尸进程释放函数的正确性：

void second\_process()

{

int pid = fork();

if (pid)

{

pid = fork();

if (pid)

{

printf("parent process return, pid %d\n",programManager.running->pid);

return;

}

else

{

uint32 tmp = 0xffffff;

while (tmp)

--tmp;

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

exit(12345);

}

}

else

{

uint32 tmp = 0xffffff;

while (tmp)

--tmp;

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

exit(-67890);

}

}

在second\_process()中，我们fork()两次，在两个子进程退出之前，我们先将父进程返回，并且在schedule()过程中，父进程的PCB被释放。接着在子进程的状态变为DEAD后，schedule()中添加的回收僵尸进程的代码段将会在一个状态为DEAD的进程出现时调用，来释放存在的僵尸进程。

我们修改第一线程来执行我们设计的second\_process来查看僵尸进程释放的结果。

下面是对第一线程修改后的代码：

void first\_thread(void \*arg)

{

printf("start process\n");

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

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

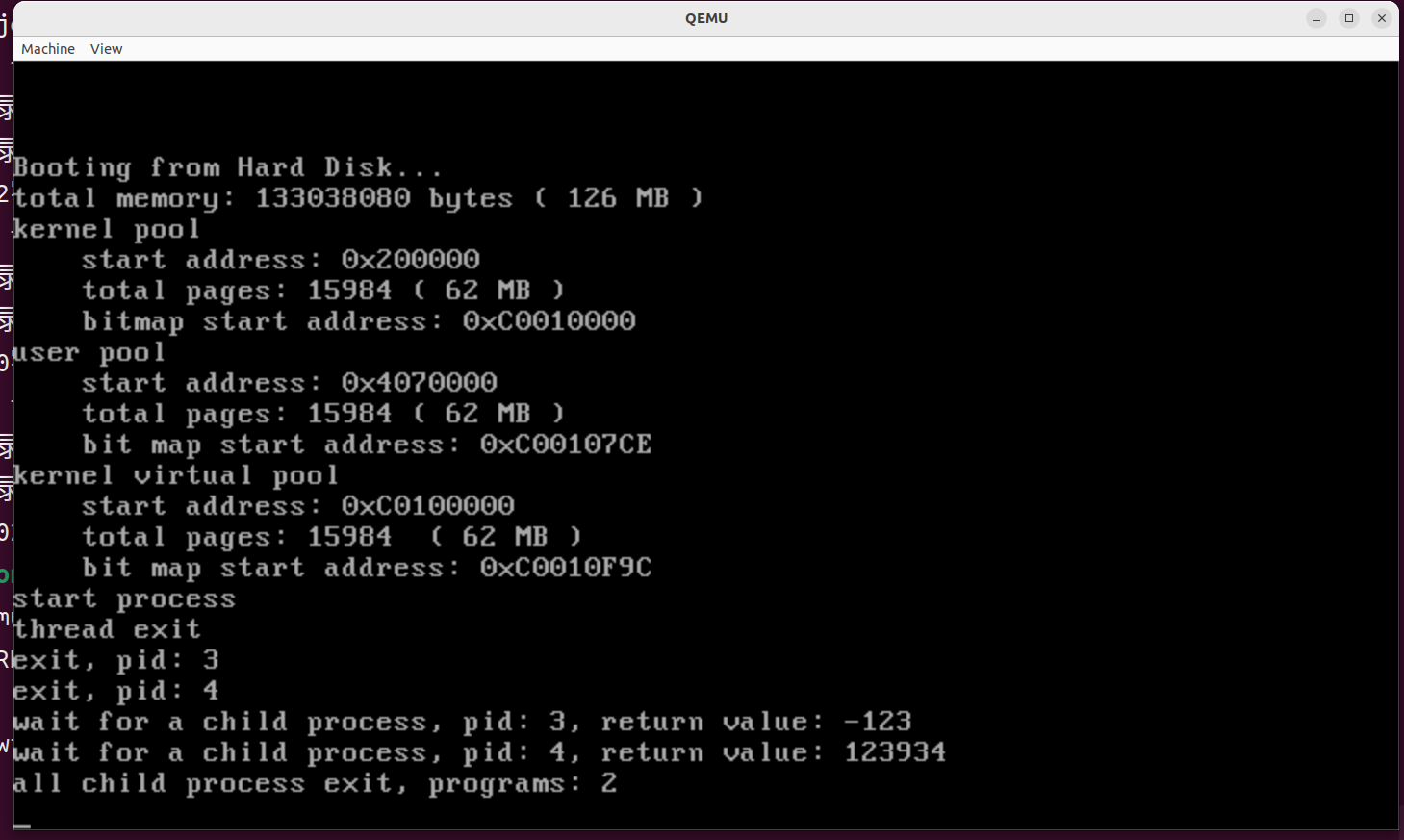
programManager.executeThread(second\_thread, nullptr, "second", 1);

}

下面我们在虚拟机中编译运行操作系统，查看测试结果。

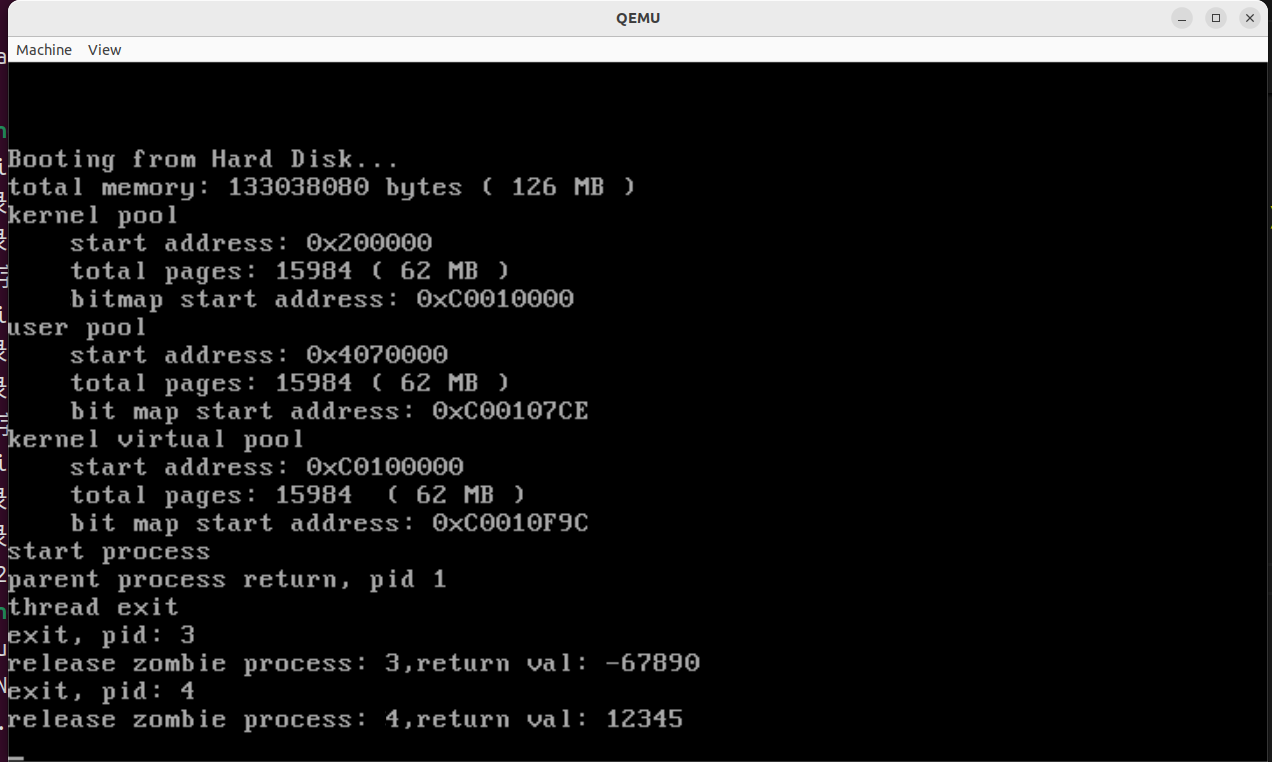
实验结果展示：通过执行前述代码，可得下图结果。

首先是实验指导实例的运行结果：



可以看到wait()和exit()函数都正确执行。

下面是测试僵尸进程释放函数的实例运行结果



可以看到在父进程放回后，子进程在变为僵尸进程后其PCB被成功释放。

**Section 3 实验总结与心得体会**

1、通过对特权级的了解，我对于CPU通过如何特权级实现特权代码以及数据的保护有了进一步认识，并了解了TSS的使用操作逻辑以及功能，对特权级的设计有了更深的理解。

2、通过实现系统调用，我学会了如何通过中断实现从特权级3(用户态)到特权级0(内核态)转移，让用户进程调用内核操作实现用户所不能实现的操作以及服务。在实现过程中知晓了系统调用的创建规则以及注意事项。

3、通过学习进程的实现，我知晓了实现进程所需要进行的各项准备以及明白了线程和进程创建的相似与不同之处，通过实现在进程中使用系统调度，我对进程在特权级转换方面需要注意实现的各项操作，如特权栈的设计以及TSS的更新有了更加深刻的认识。

4、通过实现创建子进程fork()函数，我对如何确保父子进程从fork()返回处同步运行，以及要将父进程的什么资源以何种方式复制给子进程进而实现父子进程的运行数据一致性还有差异性有了深刻的认识。对子进程的特权栈存放数值的特别安排顺序所实现的特别用处有了全面的了解。

5、exit()和wait()函数的实现，我对进程的退出以及父进程通过wait()函数实现等待子进程运行结束得到其返回结果的实现有了全面的认识。知道了通过在进程的特权级3(用户态)栈顶存放exit()函数的地址来实现进程放回后隐式调用exit()函数来实现进程数据的释放。

6、通过实现僵尸进程的释放功能，我明白了可以通过三个方面来判断一个进程是否为僵尸进程：进程状态为DEAD；进程为子进程；进程的父进程的PCB已经被释放。在实验指导代码的基础上，我添加了将状态为DEAD的父进程（或普通进程）的PCB释放的功能，并且通过将初始进程的parentPid初始值置为-1来方便判断进程是否为子进程。以上代码的修改使我对父子进程的实现以及相互之间的关系有了深刻的认识。

**Section 4 对实验的改进建议和意见**

1、可以提供更多实现系统调用的任务以及教程。

**Section 5 附录：参考资料清单**

<https://gitee.com/apshuang/sysu-2025-spring-operating-system/tree/master/lab8>

<https://github.com/linggm3/SYSU_Operate-System-Lab/tree/main>