



中山大學
SUN YAT-SEN UNIVERSITY

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: 第七章 从内核态到用户态

专业名称: 信息与计算科学

学生姓名: 郑鸿鑫

学生学号: 22336313

实验地点: 实验中心 D503

实验时间: 2024/6/13

Section 1 实验概述

在本次实验中，我们首先会简单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，我们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等。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，我们介绍了系统调用的概念和通过中断实现系统调用的方法。通过系统调用，我们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，执行完成后返回到用户态。在实现了系统调用后，我们通过三个步骤创建进程。在这里，我们需要重点理解如何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，本次实验将介绍 `fork/wait/exit` 的一种简洁的实现思路。

Section 2 预备知识与实验环境

- 预备知识：x86 汇编语言程序设计、IA-32 处理器体系结构，LBA 方式读写硬盘和 CHS 方式读写硬盘的相关知识。

- 实验环境：

- 虚拟机版本/处理器型号：

- 11th Gen Intel® Core™ i5-11320H @ 3.20GHz × 2

- 代码编辑环境：VS Code

- 代码编译工具：g++

- 重要三方库信息：Linux 内核版本号：linux-5.10.210

- Ubuntu 版本号：Ubuntu 18.04.6LTS，Busybox 版本号：

- Busybox_1_33_0

Section 3 实验任务

- 实验任务 1：实现系统调用的方法
- 实验任务 2：进程的创建与调度
- 实验任务 3：fork 的实现

● 实验任务 4: wait&exit 的实现

Section 4 实验步骤与实验结果

----- 实验任务 1 -----

● 任务要求:

复现指导书中“系统调用的实现”一节，并回答以下问题。

1. 请解释为什么需要使用寄存器来传递系统调用的参数，以及我们是如何在执行 `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？

● 实验步骤:

■ 子任务 1:

- a. 解释为什么需要用寄存器来传递系统调用的参数:

答：在使用系统调用时会发生特权级转换，传递系统调用的参数如果用栈来进行传递，则参数会被保存在用户程序的低特权级的栈。但是系统调用执行后，我们会从低特权级转到高特权级，CPU 会将高特权级的栈地址加载到 `esp` 寄存器中，供 C 语言编译后使用，但是我们的参数此时保存在低特权级的栈中，CPU 无法找到我们的参数。为了解决这个问题，我们使用寄存器来传递系统调用的参数。

- b. 解释如何在中断指令之前找到参数并放入寄存器:

为了在执行中断指令前找到参数，使用如下汇编代码:

```
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]

```

首先将 `ebp` 入栈，这用于保存函数的上下文以便返回时重新恢复。然后将 `esp` 的栈指针的值赋给 `ebp`，这标志着新的栈帧的开始。然后将寄存器依次入栈，这仍然是为了保存函数局部的参数，这些寄存器包含了函数的局部变量、参数以及段寄存器的状态。压栈是为了保护现场，确保中断处理程序可以安全地使用这些寄存器，而不会影响原函数的执行。后续赋值语句是从当前栈帧中复制参数到寄存器。每次地址需要乘以 4 是因为一个 `int` 型占据 4 个字节。这样我们就实现将参数放入寄存器中并保存了现场，可以去执行 `int 0x80` 中断了。

■ 子任务 2:

1. 设置断点并运行到中断指令执行前。
2. 使用 `info register` 指令查看寄存器尤其是 `esp` 寄存器的值。
3. 使用 `info f` 指令查看栈帧信息。
4. 再运行到中断指令执行后,再次查看 `esp` 寄存器的值和栈帧信息。

（对问题的解答与分析放在实验结果展示子任务 2 小标题下）

■ 子任务 3:

1. 设置断点并运行到函数 `asm_system_call_handler`，查看栈顶地址。
2. 使用 `x/5xw $esp` 指令查看栈顶内容。

（栈顶内容的解释放在实验结果展示子任务 3 小标题下）

■ 子任务 4:

代码如下:

```

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

```

push ds ... push gs: 这些指令将段寄存器的当前值压入栈以保存它们的状态

pushad: 这个指令将所有通用寄存器 (eax, ecx, ..., edi) 的值压入栈, 保存当前的上下文。

push eax: 再次将 eax 寄存器的值压入栈。由于 eax 通常用于传递系统调用号, 这个步骤确保了在调用系统调用处理函数之前, 系统调用号的安全。

mov eax, DATA_SELECTOR ... mov es, eax: 这些指令设置数据段寄存器, 使其指向全局描述符表 (GDT) 中定义的正确段。

mov eax, VIDEO_SELECTOR ... mov gs, eax: 这里设置了 gs 段寄存器, 通常用于访问视频内存或图形界面。

pop eax: 将之前压入栈的 eax 值恢复, 即系统调用号。

参数压栈: push edi ... push ebx: 将函数参数压入栈。这些参数将被传递给系统调用处理函数。

sti: 开启中断。在执行系统调用处理函数期间, 允许其他中断发生。

call dword[system_call_table + eax * 4]: 通过计算 system_call_table 数组中相应索引的位置来调用系统调用处理函数。eax 中的系统调用号乘以 4 (因为每个函数指针占用 4 字节) 得到偏移量。这个调用会跳转到正确的系统调用处理函数。通过这样, 我们可以找到中断向量号对应的系统调用的函数的地址。

■ 子任务 5:

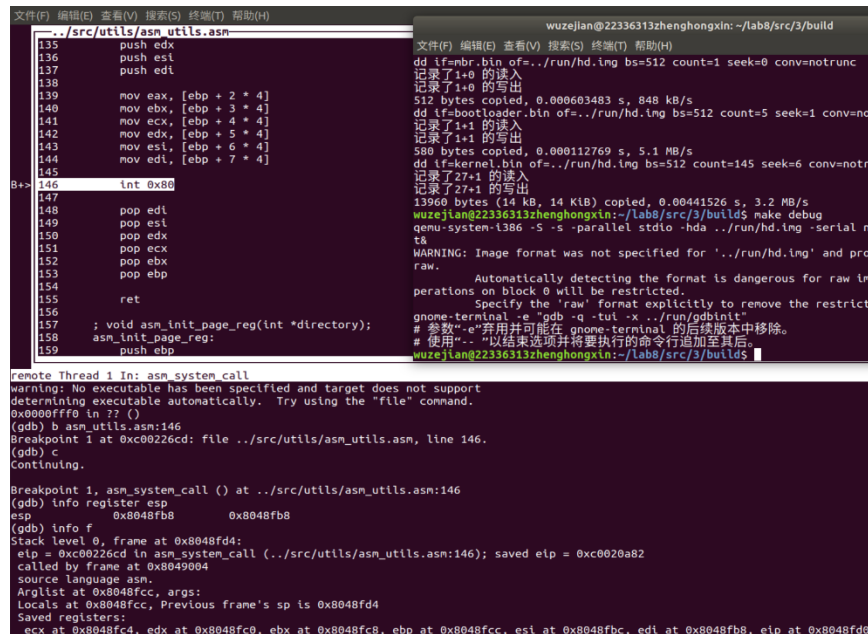
1. 设置断点跳转到执行 `iret` 以后。
2. 使用 `info register` 指令查看寄存器的值。

(对寄存器值变化的分析放在实验结果展示子任务 5 小标题下)

● 实验结果展示:

子任务 2:

1. 在中断调用指令执行之前查看 `esp` 寄存器和栈帧的值:



The screenshot shows a debugger window with two panes. The left pane displays assembly code from `../src/utlis/asm_utils.asm`, with line 146 (`int 0x80`) highlighted. The right pane shows the command prompt of a virtual machine, displaying the execution of `dd` commands to write bootloaders and kernels to a hard disk image, along with a warning about the image format. Below the panes, the GDB console shows the following output:

```
remote Thread 1 In: asm_system_call
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) b asm_utils.asm:146
Breakpoint 1 at 0xc00226cd: file ../src/utlis/asm_utils.asm, line 146.
(gdb) c
Continuing.

Breakpoint 1, asm_system_call () at ../src/utlis/asm_utils.asm:146
(gdb) info register esp
esp                0x8048fb8      0x8048fb8
(gdb) info f
Stack level 0, frame at 0x8048fd4:
eip = 0xc00226cd in asm_system_call (../src/utlis/asm_utils.asm:146); saved eip = 0xc0020a82
called by frame at 0x8049004
source language asm.
Arglist at 0x8048fcc, args:
Locals at 0x8048fc0, Previous frame's sp is 0x8048fd4
Saved registers:
ecx at 0x8048fc4, edx at 0x8048fc0, ebx at 0x8048fc8, ebp at 0x8048fcc, esi at 0x8048fbc, edi at 0x8048fb8, eip at 0x8048fd0
```

2. 在中断调用指令执行之后查看 `esp` 寄存器和栈帧的值:

```
../src/utils/asm_utils.asm
77
78     add word[ASM_GDTR], 8
79     lgdt [ASM_GDTR]
80
81     pop esi
82     pop ebx
83     pop ebp
84
85     ret
86     ; int asm_system_call_handler();
87     asm_system_call_handler:
B+> 88     push ds
89     push es
90     push fs
91     push gs
92     pushad
93
94     push eax
95
96     ; 栈段会从tss中自动加载
97
98     mov eax, DATA_SELECTOR
99     mov ds, eax
100    mov es, eax
101

remote Thread 1 In: asm_system_call_handler
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) b asm_utils.asm:88
Breakpoint 1 at 0xc0022677: file ../src/utils/asm_utils.asm, line 88.
(gdb) c
Continuing.

Breakpoint 1, asm_system_call_handler () at ../src/utils/asm_utils.asm:88
(gdb) info register esp
esp             0xc002568c      0xc002568c <PCB_SET+8172>
(gdb) info f
Stack level 0, frame at 0xc0025690:
eip = 0xc0022677 in asm_system_call_handler (../src/utils/asm_utils.asm:88); saved eip = 0xc00226cf
called by frame at 0x8048fd4
source language asm.
Arglist at 0xc0025688, args:
Locals at 0xc0025688, Previous frame's sp is 0xc0025690
Saved registers:
eip at 0xc002568c
```

可以看到，在中断指令执行前后，栈顶指针寄存器 `esp` 和栈帧都发生了变化，这是因为栈顶指针已经从低特权级的栈顶地址变化为了高特权级的栈顶的地址了。用户进程可分配的虚拟地址是从 `0x8048000` 开始分配所以看到执行中断指令之前栈顶的地址是 `0x8048fb8`，说明此时的栈仍然是低特权级的用户空间的栈。执行中断调用指令后，栈顶的地址变换为了 `0xc002568c`，说明已经进入了内核虚拟地址空间，切换为高特权级的栈。

对于 `esp` 值的变化与 `setup.cpp` 中的变量 `tss` 的关系：

查看变量 `tss` 的信息如下：

```
(gdb) p/x tss
$2 = {backlink = 0x0, esp0 = 0xc00256a0, ss0 = 0x10, esp1 = 0x0, ss1 = 0x0,
      esp2 = 0x0, ss2 = 0x0, cr3 = 0x0, eip = 0x0, eflags = 0x0, eax = 0x0,
      ecx = 0x0, edx = 0x0, ebx = 0x0, esp = 0x0, ebp = 0x0, esi = 0x0, edi = 0x0,
      es = 0x0, cs = 0x0, ss = 0x0, ds = 0x0, fs = 0x0, gs = 0x0, ldt = 0x0,
      trace = 0x0, ioMap = 0xc0033bac}
```

可以看到，`tss` 中的 `esp0` 存储了 `0xc00256a0`，为特权级 0 的栈顶的地址。`ss0` 为 `0x10`，为特权级 0 的段选择子。在系统调用发生时，CPU 会从 `tss` 中加载高特权级栈顶的 `esp0` 和 `ss0` 送入 `esp` 和 `ss` 寄存器，这就是为什么系统调用后 `esp` 发生了改变。

对于其他发生变换的段寄存器：

在中断指令执行之前查看寄存器的值：

```
(gdb) info register
eax            0x0          0
ecx            0x144        324
edx            0xc          12
ebx            0x84         132
esp            0x8048fb8     0x8048fb8
ebp            0x8048fcc     0x8048fcc
esi            0x7c         124
edi            0x0          0
eip            0xc00226cd    0xc00226cd <asm_system_call+26>
eflags         0x212        [ AF IF ]
cs             0x2b         43
ss             0x3b         59
ds             0x33         51
es             0x33         51
fs             0x33         51
gs             0x0          0
```

在中断指令执行之后再次查看他们的值：

```
(gdb) info register
eax            0x0          0
ecx            0x144        324
edx            0xc          12
ebx            0x84         132
esp            0xc0025688    0xc0025688 <PCB_SET+8168>
ebp            0x8048fcc     0x8048fcc
esi            0x7c         124
edi            0x0          0
eip            0xc0022678    0xc0022678 <asm_system_call_handler+1>
eflags         0x12         [ AF ]
cs             0x20         32
ss             0x10         16
ds             0x33         51
es             0x33         51
fs             0x33         51
gs             0x0          0
```

可以看到发生变化的寄存器有 esp（刚才已经分析过），cs 和 ss，这是因为他们的值来源于高特权级的栈地址和段选择子，所以都随着系统调用发生了变化。

子任务 3：

1. 进入函数 asm_system_call_handler 后栈顶地址如下：

```
(gdb) info register esp
esp            0xc002568c    0xc002568c <PCB_SET+8172>
```

2. 查看栈顶内容如下：

```
(gdb) x/5xw 0xc002568c
0xc002568c <PCB_SET+8172>: 0xc00226cf 0x0000002b 0x00000212 0x08048fb8
0xc002569c <PCB_SET+8188>: 0x0000003b
```

关于栈顶内容的解释：

这是由于在特权级转换的时候，CPU 将中断前的 ss,esp,eflags,cs,eip 等寄存器依次压入特权级 0 的栈中。可以对照 int 0x80 指令前寄存器的值得到验证：


```
(gdb) info register
eax          0x0          0
ecx          0x144        324
edx          0xc          12
ebx          0x84         132
esp          0x8048fb8     0x8048fb8
ebp          0x8048fcc     0x8048fcc
esi          0x7c         124
edi          0x0          0
eip          0xc00226cd     0xc00226cd <asm_system_call+26>
eflags       0x212        [ AF IF ]
cs           0x2b         43
ss           0x3b         59
ds           0x33         51
es           0x33         51
fs           0x33         51
gs           0x0          0
```

验证得出结论与分析一致。

子任务 5:

查看执行 `iret` 后寄存器的值如下:

```
(gdb) info register
eax          0x250        592
ecx          0x144        324
edx          0xc          12
ebx          0x84         132
esp          0x8048fb8     0x8048fb8
ebp          0x8048fcc     0x8048fcc
esi          0x7c         124
edi          0x0          0
eip          0xc00226cf     0xc00226cf <asm_system_call+28>
eflags       0x212        [ AF IF ]
cs           0x2b         43
ss           0x3b         59
ds           0x33         51
es           0x33         51
fs           0x33         51
gs           0x0          0
```

对照上图和执行中断指令之前的图可以发现:

段寄存器发生了变化, 变回了中断调用之前的值。这是因为当高特权级向低特权级转移时, 我们并不需要给出低特权级栈的信息。因为 CPU 是禁止高特权级向低特权级转移的, 只有一种情况除外, 就是中断返回或任务返回, 汇编命令是 `iret` 和 `retf`。因此, CPU 默认高特权级向低特权级转移的情况是中断或调用处理完成后返回。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` 可以被恢复。

----- 实验任务 2 -----

● 任务要求:

复现“进程的实现”“进程的调度”“第一个进程”三节, 并回答以下问题

1. 请结合代码分析我们是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. 进程的创建过程中，我们使用了以下代码：

```
1 int ProgramManager::executeProcess(const char *filename, int priority)
2 {
3     ...
4     //找到刚刚创建的PCB
5     PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
6     ...
7 }
8
```

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`：

```
1 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：

```
1 int ProgramManager::executeProcess(const char *filename, int priority)
2 {
3     ...
4     //找到刚刚创建的PCB
5     PCB *process = findProgramByPid(pid);
6     ...
7 }
```

自行测试通过后，说一说你的实现思路，并保存结果截图。

● 实验步骤：

■ 子任务 1

对进程的创建分析如下：

进程和线程使用了 PCB 来保存其基本信息，如 pid，栈等。但是，进程和线程的区别在于进程有自己的虚拟地址空间和相应的分页机制，也就是虚拟地址池和页目录表。我们先在 PCB 中新增如下内容。

```
struct PCB
{
    ...
    int pageDirectoryAddress;           // 页目录表地址
    AddressPool userVirtual;           // 用户程序虚拟地址池
};
```

然后按照以下三步来分析进程的创建：

1. 创建进程的 PCB
2. 初始化进程的页目录表
3. 初始化进程的虚拟地址池

```
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 = ProgramStatus::DEAD;
        interruptManager.setInterruptStatus(status);
        return -1;
    }
    // 创建进程的虚拟地址池
    bool res = createUserVirtualPool(process);
    if (!res)
    {
        process->status = ProgramStatus::DEAD;
        interruptManager.setInterruptStatus(status);
        return -1;
    }
}
```

```

    }
    interruptManager.setInterruptStatus(status);
    return pid;
}

```

对于第一步创建进程的 PCB，我们像创建一个线程一样创建进程的 PCB。这里，创建线程的参数并不是进程的起始地址，而是加载进程的函数，这个我们后面再分析。如果进程创建失败，我们只要简单地将其标记为 DEAD 即可。

对于第二步初始化进程的页目录表，我们为进程创建页目录表。页目录表的创建和初始化通过 ProgramManager::createProcessPageDirectory 来实现，这个函数的代码如下所示：

```

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;
}

```

我们先从内核中分配一页来存储进程的页目录表，为了使进程能够访问内核资源，根据之前的约定，我们令用户虚拟地址的 3GB-4GB 的空间指向内核空间。为此，我们需要将内核的第 768~1022 个页目录项复制到进程的页目录表的相同位置。值得注意的是，我们需要构造出页目录表的第 768~1022 个页目录项分别在内核页目录表 and 用户进程中的虚拟地址。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。

我们接着分析第三步，初始化进程的虚拟地址池: 这里，我们将用户进程的可分配的虚拟地址的定义在 `USER_VADDR_START` 和 `3GB` 之间，这是仿照 `linux` 的做法。然后，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。接着我们在内核空间中为进程分配位图所需的内存。至此，进程的创建已经完成。

■ 子任务 2

1. 设置断点运行到创建一个新进程时创建 `PCB` 的代码处
2. 单步进入该函数，观察进入的函数。
3. 分析参数和 `esp` 的值。

(详细的解释和问题的分析放在实验结果子任务 2 小标题下)

■ 子任务 3

1. 代码分析 `ProgressStartStack` 内容的设置。

```
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; // 用户模
式平坦模式
interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1); // IOPL,
IF = 1 开中断, MBS = 1 默认
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;
interruptStack->ss = programManager.USER_STACK_SELECTOR;
```

由于 CPU 默认高特权级向低特权级转移的情况是中断或调用处理完成后返回。低特权级栈的信息在进入中断前被保存在高特权级栈中。因此这里我们想要强调的是 ss 和 esp 在 load_progress 中是如何被设置的。由代码中高亮的地方我们可以看到 esp 被赋值为用户地址池中分配的一页，而 ss 被赋值为用户级即特权级 3 的栈的段选择子。还有一点是 eip 被赋值为(int)filename,即进程函数的地址。

2. 我们结合 gdb 的调试分析来验证我们的代码实现是否正确。

(gdb 操作，分析及其结果解释放在实验结果展示子任务 3 小标题下)

■ 子任务 4

我们对 schedule 函数修改后的代码如下所示：

```
void ProgramManager::schedule()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();
    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }
    else
    {
        //printf("amount of ready programs: %d\n",
readyPrograms.size());
    }
    if (running->status == ProgramStatus::RUNNING)
    {
        running->status = ProgramStatus::READY;
        running->ticks = running->priority * 10;
        readyPrograms.push_back(&(running->tagInGenerallist));
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
    ListItem *item = readyPrograms.front();
```

```

PCB *next = ListItem2PCB(item, tagInGeneralList);
PCB *cur = running;
next->status = ProgramStatus::RUNNING;
running = next;
readyPrograms.pop_front();
activateProgramPage(next);
asm_switch_thread(cur, next);
interruptManager.setInterruptStatus(status);
}

```

我们对其进行的修改体现在高亮的地方，我们在原来的线程调度的基础上增加了切换页目录表和更新 TSS 中的特权级 0 的栈这两项。其中函数 `activateProgramPage` 函数的功能是激活下一个要运行的进程的页目录表，它的具体代码如下所示：

```

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);
}

```

`int paddr = PAGE_DIRECTORY;`：初始化页目录表的物理地址为默认值。
`if (program->pageDirectoryAddress)`：检查进程是否有自己的页目录表。
`tss.esp0 = (int)program + PAGE_SIZE;`：设置 TSS 中的 `esp0` 字段，指向进程的特权级 0 栈的栈顶。这是为进程切换后可能发生的中断或异常准备的。
`paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);`：如果进程有自己的页目录表，则将页目录表的物理地址更新为 `paddr`。
`asm_update_cr3(paddr);`：最后，更新 CR3 寄存器，使 CPU 使用新的页目录表，完成虚拟地址空间的切换。

- 做这样改进的目的是：

每个进程都有自己的虚拟地址空间，通过切换页目录表，可以确保进程使用其独立的地址空间。这有助于隔离不同进程的内存，防止它们相互影响。更新 TSS 中的特权级 0 栈是为了在进程切换时能够正确地处理中断和异常。这确

保了在进程被调度执行时，如果发生中断或异常，CPU 能够使用正确的栈来保存状态。

■ 子任务 5

1. 先在类 Program Manager 中添加函数 findProgramByPid 的声明。

```
PCB* findProgramByPid(int pid);
```

2. 然后在 program.cpp 中实现该函数，主要思路是遍历链表，逐个与 pid 进行比对，然后查找得到结果。

具体代码如下：

```
PCB*ProgramManager:: findProgramByPid(int pid){
    ListItem*cur = &allPrograms.head;
    while(cur != nullptr){
        PCB*temp = ListItem2PCB(cur,tagInAllList);
        if (temp -> pid == pid ) return temp;
        else cur = cur -> next;
    }
    return nullptr;
}
```

3. 将原来具有风险的语句注释掉，然后利用刚刚实现的方法替换进去。

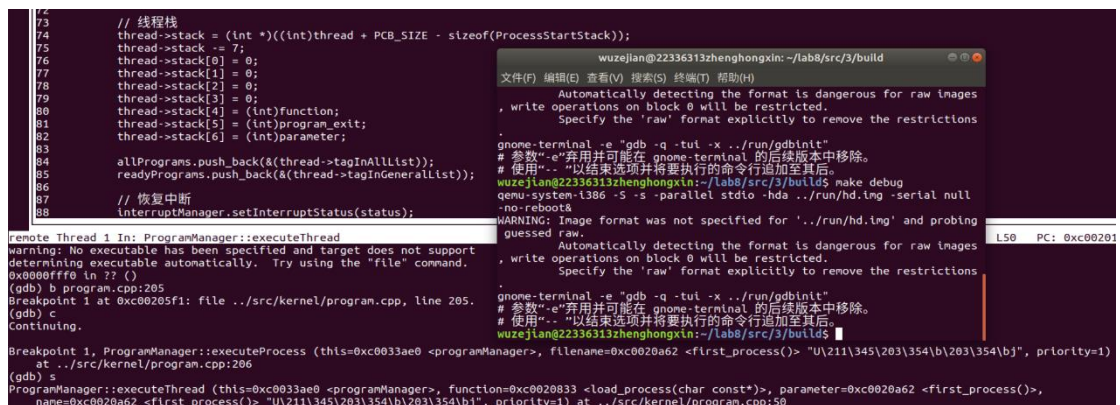
```
// 找到刚刚创建的PCB
//PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
PCB*process = findProgramByPid(pid);
```

4. 重新编译运行代码，观察结果是否正常。

● 实验结果展示：

子任务 2：

1. 单步进入此函数：



```
73 // 线程栈
74 thread->stack = (int *)((int)thread + PCB_SIZE - sizeof(ProcessStartStack));
75 thread->stack -= 7;
76 thread->stack[0] = 0;
77 thread->stack[1] = 0;
78 thread->stack[2] = 0;
79 thread->stack[3] = 0;
80 thread->stack[4] = (int)function;
81 thread->stack[5] = (int)program_exit;
82 thread->stack[6] = (int)parameter;
83
84 allPrograms.push_back(&(thread->tagInAllList));
85 readyPrograms.push_back(&(thread->tagInGeneralList));
86
87 // 恢复中断
88 InterruptManager.setInterruptStatus(status);

remote Thread 1 In: ProgramManager::executeThread
warning: no executable has been specified and target does not support
determining executable automatically. Try using the 'file' command.
0x0000ffff in ?? ()
(gdb) b program.cpp:205
Breakpoint 1 at 0xc00205f1: file ../src/kernel/program.cpp, line 205.
(gdb) c
Continuing.

Breakpoint 1, ProgramManager::executeThread (this=0xc0033ae0 <programManager>, filename=0xc0020a62 <first_process>) at ../src/kernel/program.cpp:206
(gdb) s
ProgramManager::executeThread (this=0xc0033ae0 <programManager>, function=0xc0020833 <load_process(char const*)>, parameter=0xc0020a62 <first_process>, name=0xc0020a62 <first_process>) at ../src/kernel/program.cpp:58
```

发现参数 function 被存储在 thread 的 stack 中。因此我们继续执行该函数进入了 asm_switch-thread 函数。

2. 执行的语句如下:

```
202      mov eax, [esp + 6 * 4]
203      mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
```

这说明 esp 中的值已经被赋值为 load_process 函数的地址, 即在函数执行 iret 后, 代码会跳转到 load_process。我们可以通过 gdb 来分析这一点, 首先查看 executeThread 中 function 中的值:

```
(gdb) p/x function
$1 = 0xc0020a8d
```

然后再看 asm_switich_thread 函数执行 ret 后, 栈顶的值:

```
(gdb) info register esp
esp          0xc0024650          0xc0024650 <PCB_SET+4016>
(gdb) p/x *0xc0024650
$2 = 0xc0020a8d
```

可以看到这两个值都是相同的, 为 0xc0020a8d。验证了我们说的函数 asm_switich_thread 执行 ret 后会切换到 load_process。

子任务 3:

1. 进入 load_process 函数后查看 esp 和 ss 寄存器的值如下所示

```
321      interruptStack->esp += PAGE_SIZE;
322      interruptStack->ss = program;
323
324      asm_start_process((int)interruptStack->esp, interruptStack->ss);
325  }
326
327  void ProgramManager::activateThread(tg)
328  {
329      int paddr = PAGE_DIRECTOR;
330
331      if (program->pageDirector)
332      {
333          tss.esp0 = (int)program;
334          paddr = memoryManager->getPhysicalAddress(program);
335      }
336
337      asm_update_cr3(paddr);
338  }

remote Thread 1 In: load_process
warning: No executable has been specified
determining executable automatically. Try:
0x0000ffff in ?? ()
(gdb) b program.cpp:324
Breakpoint 1 at 0xc002095b: file ../src/kernel/asm_start_process.c
(gdb) c
Continuing.
warning: Source file is more recent than target.
Breakpoint 1, load_process (filename=0xc0020a62 <first_process()> "U\211\345\203\354\b\203
(gdb) p/x interruptStack->esp
$1 = 0x8049000
(gdb) p/x interruptStack->ss
$2 = 0x3b
```

2. 再在 load_process 中查看 interruptStack 的值, 然后单步进入函数 asm_start_process 中来查看赋值后 esp 中的值

```
Breakpoint 1, load_process (filename=0xc0020a62 <first_process()> "U\211\345\203\354\b\203\354\bj") at ../src/kernel/program.cpp:325
(gdb) p/x interruptStack
$1 = 0xc002565c
(gdb) s
asm_start_process () at ../src/utls/asm_utls.asm:41
(gdb) n
(gdb) n
asm_start_process () at ../src/utls/asm_utls.asm:43
(gdb) info register esp
Undefined info command: "register esp". Try "help info".
(gdb) fs cmd
Focus set to cmd window.
(gdb) info register esp
esp      0xc002565c      0xc002565c <PCB_SET+8124>
```

可以看到二者的值相同，都是 0xc002565c，所以这就是为什么在 iret 指令执行后可以实现从内核态到用户态的转移，即从特权级 0 转移到特权级 3 中，而且由于上文提到了 eip 中存储着进程函数的地址，所以 iret 执行后可以启动进程。

子任务 5:

修改“有风险的实现方法”后，编译运行结果如下：

```
wuzejian@22336313zhenghongxin:~/lab8/src/3/build$ make run
qemu-system-i386 -hda ../run/hd.ing -serial null -parallel stdio -no-reboot
WARNING: Image format was not specified for '../run/hd.ing' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

QEMU

iPXE (http://ipxe.org) 00:03.0 C980 FC12.10 PnP PMM+07F8DD00+07ECDD00 C980

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x2000000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
```

----- 实验任务 3 -----

● 任务要求:

复现“fork”一小节的内容，并回答以下问题：

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“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

● 实验步骤:

■ 子任务 1

- 关于 fork 的基本思路的概括:

在用户程序中调用 fork 系统调用时，操作系统内核会复制发起调用的父进程的资源，创建一个新的子进程。这个子进程几乎复制了父进程的所有属性，包括代码段、数据段、堆和栈，但拥有独立的进程 ID。fork 调用在父进程中返回新创建的子进程的 PID，在子进程中返回 0，若出错则在两者中都返回负值。通过这种方式，fork 实现了一次调用但两次返回的机制，允许父进程和子进程从 fork 调用之后的代码点继续独立执行，从而支持多任务和并发操作。

- 如何解决“四个关键问题”的分析:

首先四个关键问题为:

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?

对于第一个关键问题，由于函数的代码段都位于相同的地址空间，所以父子进程实现彼此代码段之间的共享。再通过如下代码:

```
// 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));
```

我们实现把 ProcessStartStack 的内容复制到子进程的 0 级栈中，子进程通过 asm_start_process 来启动，其中 ProcessStartStack 中保存了父进程的

eip, 指向了 asm_system_call_handler 的返回地址, 所以在子进程中执行 iret 后会返回到这个地址。使得子进程开始执行的代码正是父进程的返回点。接着, 子进程继续执行, 通过 fork 等函数返回。父进程的 3 级栈中保存着系统调用的返回地址。通过将父进程的 3 级栈复制给子进程, 就可以实现父子进程返回到相同的地址。这便实现了第二个关键问题, 使子进程和父进程从相同的返回点开始执行。对于第三个关键问题, 除代码段外, 进程还包括以下资源: PCB, 虚拟地址池, 页目录表, 页表以及其指向的物理页等。

接下来, 我们来研究 fork 实现最关键的部分——资源的复制, 也就是第四个关键问题, 通过以下代码实现:

```
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;
    // 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
    // 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);
    // 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;
    }
    // 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;
        }
        // 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
        }
    }
    // 归还从内核分配的中转页

```



```
memoryManager.releasePages(AddressPoolType::KERNEL, (int)buffer,
1);
return true;
}
```

这段代码通过 `ProgramManager::copyProcess` 函数实现了父进程到子进程的资源复制过程，首先复制父进程的栈信息和 PCB 配置到子进程，然后在子进程中设置适当的栈帧以准备执行，并利用分配的中转页在物理页级别上复制页目录和页表项，最终确保子进程拥有独立的地址空间和资源副本，而父子进程的代码段共享，以完成 `fork` 操作。

■ 子任务 2

1. 启动 gdb 调试，设置断点跳转到 `asm_switch_thread` 函数切换到子进程的栈的位置。
2. 使用 `info register esp` 指令查看 `esp` 的值。
3. 使用 `x/3xw` 指令来查看栈顶的三个元素。

（具体结果与问题分析解释放在实验结果展示的子任务 2 小标题下）

■ 子任务 3

1. 启动 gdb 调试，首先在调用 `fork` 处步入。
2. 然后逐级深入的找到系统调用 `ProgramManager::fork`。
3. 然后逐级返回，分析子进程和父进程在这一过程中的寄存器信息的异同。

（详细流程和分析放在实验结果展示的子任务 3 小标题下）

■ 子任务 4

1. 在子任务 3 的 gdb 调试过程中我们已经回答了为什么父进程 `fork` 的返回值为子进程的 `pid`，下面再分析为什么子进程的 `fork` 返回值是 0。
2. 观察代码可以看到:

```
// 初始化子进程
PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
bool flag = copyProcess(parent, child);
// 设置子进程的返回值为 0
childpss->eax = 0;
```

在初始化子进程的时候我们对于子进程的 `eax` 设置默认为 0，而且如果这个子进程没有创建新的“子孙”进程，那么他的 `eax` 将一直保持是 0。所以我们可以知道，从子进程被建立后没有再创建新的进程了，所以 `eax` 中的值一直保

持是 0。在 C/C++ 函数的调用规则中，返回值被存储 `eax` 寄存器中，这就是为什么子进程的返回值是 0，根本原因在于其没有新的子进程。

■ 子任务 5

在实验任务中的子任务 4 中我们分析了线程调度的改进。我们继续看到这个函数的代码，从一个进程的虚拟空间切换到另一个进程的虚拟空间是依靠函数 `activateProgramPage()` 来实现的，具体代码如下：

```
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);
}
```

其中高亮部分是赋值了一个 `int` 变量，给其赋值为下一个进程的页目录表地址，然后更新 `cr3` 寄存器以实现虚拟地址空间的切换。

● 实验结果展示：

子任务 2

查看子进程栈顶元素和查看 `asm_start_process` 函数的起始地址：

```
210     sti
B+> 211     ret
212     ; int asm_interrupt_status();
213     asm_interrupt_status:
214     xor eax, eax
215     pushfd
216     pop eax
217     and eax, 0x200
218     ret
219
220     ; void asm_disable_interrupt();
221     asm_disable_interrupt:
222     cli
223     ret
224     ; void asm_init_page_reg(int *director

remote Thread 1 In: asm_switch_thread
(gdb) fs cmd
Focus set to cmd window.
(gdb) i r esp
esp             0xc0025d90      0xc0025d90 <PCB_SET+8112>
(gdb) c
Continuing.

Breakpoint 1, asm_switch_thread () at ../src/utils/asm_utils.asm:211
(gdb) i r esp
esp             0xc0024ce4      0xc0024ce4 <PCB_SET+3844>
(gdb) c
Continuing.

Breakpoint 1, asm_switch_thread () at ../src/utils/asm_utils.asm:211
(gdb) i r esp
esp             0xc0026d90      0xc0026d90 <PCB_SET+12208>
(gdb) x/3xw 0xc0026d90
0xc0026d90 <PCB_SET+12208>:  0xc0022c20      0x00000000      0xc0026d9c
(gdb) p asm_start_process
$1 = {<text variable, no debug info>} 0xc0022c20 <asm_start_process>
```

可以看到这两个地址是相同的，都是 0xc0022c20，这说明在执行 ret 后，函数会跳转到 asm_start_process 的起始地址。

在进入函数 asm_switch_process 后执行两步后，下一条指令的地址就是栈顶的第 3 个元素，也就是 0xc0026dbc，这个地址将指向子进程的 0 级栈，我们再设置断点跳转到 copyProcess 函数中，打印 childpss 的地址，得到结果如下：

```
(gdb) p childpss
$1 = (ProcessStartStack *) 0xc0026d9c <PCB_SET+12220>
```

与上面栈顶的第三个元素一致，说明子进程的栈中已经从父进程复制了 0 级栈。

子任务 3

在 fork 处使用步入调试。

```
(gdb) b first_process
Breakpoint 1 at 0xc0020fc9: file ../src/kernel/setup.cpp, line 32.
(gdb) c
Continuing.

Breakpoint 1, first_process () at ../src/kernel/setup.cpp:32
```

fork 先调用 asm_system_call 。

```
35     int fork() {
> 36         return asm_system_call(2);
37     }
38
39     int syscall_fork() {
40         return programManager.fork();
41     }^?
42
43
44
45
46
47
48
49

remote Thread 1 In: fork
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) b first_proces
Function "first_proces" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b first_process
Breakpoint 1 at 0xc0020fc9: file ../src/kernel/setup.cpp, line 32.
(gdb) c
Continuing.

Breakpoint 1, first_process () at ../src/kernel/setup.cpp:32
(gdb) n
(gdb) s
```

asm_system_call 中再调用了 asm_system_handler 找到第 2 个系统调用 syscall_fork,并调用之。最后 syscall_fork 会调用 ProgramManager::fork, 这便是逐级调用的流程。在 ProgramManager::fork 函数中, 代码执行会一分为二, 所以返回会有两次。

下面先分析父进程:

在 ProgramManager::fork 函数 return 后, 子进程的 pid 会被存储在 eax 寄存器中, 接着父进程返回到调用的地方也就是 asm_system_call_handler, 用变量 ASM_TEMP 来保证 eax 的值不发生改变。

```
118         add esp, 5 * 4
119
120         mov [ASM_TEMP], eax
121         popad
122         pop gs
123         pop fs
124         pop es
125         pop ds
126         mov eax, [ASM_TEMP]
127
128         iret
```

然后再返回上一层调用 asm_system_call。此时我们使用 info register 查看父进程的寄存器信息如下:

```

eax      0x2      2
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048f98  0x8048f98
ebp      0x8048fac  0x8048fac
esi      0x0      0
edi      0x0      0
eip      0xc0022ccf 0xc0022ccf <asm_system_call+28>
eflags   0x216    [ PF AF IF ]
cs       0x2b     43
ss       0x3b     59
ds       0x33     51
es       0x33     51
fs       0x33     51
gs       0x0      0

```

最后父进程返回到最外层 fork，eax 返回子进程的 pid。

对于子进程，同理按照刚才逐层返回的思想跳转到 asm_system_call 中与父进程一致的返回点，同样使用 info register 指令查看子进程的寄存器信息。

```

(gdb) i r
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048f98  0x8048f98
ebp      0x8048fac  0x8048fac
esi      0x0      0
edi      0x0      0
eip      0xc0022ccf 0xc0022ccf <asm_system_call+28>
eflags   0x216    [ PF AF IF ]
cs       0x2b     43
ss       0x3b     59
ds       0x33     51
es       0x33     51
fs       0x33     51
gs       0x0      0

```

对照两个进程的寄存器信息，不难发现，只有 eax 寄存器存在差异。这是因为子进程分返回不需要经过 ProgramManager::fork，而是直接跳转到了 asm_system_call 中。所以他的 eax 中的值为 0（原因在子任务 4 中会给出），父进程的 eax 存储的值为子进程的 pid。

----- 实验任务 4 -----

● 任务要求：

参考指导书中“wait”和“exit”两节的内容，实现 wait 函数和 exit 函数，回答以下问题：

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。(tips: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。

● 实验步骤：

◆ 子任务 1

exit 是一个系统调用，用于进程和线程的主动结束。我们参考以下代码来分析 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();
}

```

重要步骤已用高亮标注，首先将 PCB 的状态置为 DEAD，然后根据页目录表进行判断，判断 PCB 标识的为线程还是进程，如果是进程，则需要释放其对应的资源，否则不做处理。最后再进程/线程调度，运行新的进程/线程。

再编写简单的测试样例来测试 exit:

```

void first_process()
{
    int pid = fork();
    if (pid == -1)
    {
        printf("can not fork\n");
        asm_halt();
    }
    else
    {
        if (pid)
        {
            printf("father halt\n");
            asm_halt();
        }
        else
        {
            printf("child exit\n");
            exit(22336313);
        }
    }
}
}

```

◆ 子任务 2

从栈的角度分析进程退出后可以隐式调用 exit()的原因:

根据 C/C++ 函数的调用规则，我们只要在进程的特权级 3 的栈的顶部放入 `exit` 函数的地址和对应的参数。所以需要在 `load_process` 中修改代码如下：

```
// 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;
interruptStack->ss = programManager.USER_STACK_SELECTOR;
asm_start_process((int)interruptStack);
```

其中，`userStack[0]` 是 `exit` 的函数地址，`userStack[1]` 是 `exit` 的返回地址，`userStack[2]` 是 `exit` 的参数。这样便可实现隐式的调用 `exit(0)`。

◆ 子任务 3

在 `exit` 中我们将识别进程与线程，如果是进程则会将其除 PCB 外的所有已分配的资源，但还没有清除 PCB。PCB 需要依靠父进程调用 `wait` 来回收，用来等待子进程执行完成并回收 PCB。

`Wait` 函数有一个参数为子进程的返回值，而 `wait` 自身的返回值为子进程的 `pid`。具体代码实现如下：

```
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 (flag)
        {
            continue;
        }
        *retval = child->pid;
        delete child;
    }
}
```

```

        }
    }
    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();
    }
}
}
}

```

代码实现了 wait 系统调用的过程，其中父进程通过遍历所有子进程的控制块（PCB）来查找状态为终止（DEAD）的子进程。一旦找到已终止的子进程，它将该子进程的返回值赋给提供的 retval 指针，清理子进程的资源，并返回子进程的 PID。如果在遍历过程中没有发现已终止的子进程，wait 调用将导致当前进程进入等待状态，直到至少有一个子进程结束。这个实现确保了父进程能够准确地收集子进程的退出状态，同时保持系统资源的有效利用。

下面对 wait 编写测试样例

```

void first_process()
{
    int pid = fork();
    int retval;

```

```

if(pid){
    pid = fork();
    if(pid){
        while((pid = wait(&retval)) != -1)    printf("wait :
pid:%d,return value:%d\n ",pid,retval);
        printf("all child
exit:program: %d\n",programManager.allPrograms.size());
        asm_halt();
    }else{
        uint32 tmp = 0xffffffff;
        while(tmp) --tmp;
        printf("exit,pid:%d\n",programManager.running->pid);
        exit(6313);
    }
}
else{
    uint32 tmp = 0xffffffff;
    while(tmp) --tmp;
    printf("exit,pid:%d\n",programManager.running->pid);
    exit(2233);
}
}

```

◆ 子任务 4

孤儿进程的处理：在父进程意外退出时，操作系统中的初始化进程（通常 PID 为 1 的 init 进程）会自动接管所有成为孤儿的子进程。Init 进程会调用 wait 来回收这些孤儿进程，确保它们不会成为僵尸进程。**僵尸进程的回收：**为了避免僵尸进程占用系统资源，可以采取以下措施：**修改调度器：**在调度器的 schedule 函数中添加检查，如果发现有状态为 DEAD 的进程，立即调用 releasePCB 函数来回收其资源。

对于线程没有被父线程回收，我们是在 schedule 函数中用以下语句回收：

```

else if (running->status == ProgramStatus::DEAD)
{
    releasePCB(running);
}

```

那么对于没有父进程回收的子进程，我们也可以做类似的处理，将进程的资源释放掉，具体代码如下：

```

else if (running->status == ProgramStatus::DEAD)
{
    if (!running->pageDirectoryAddress){
        printf("release thread PCB with pid %d\n",running->pid);
        releasePCB(running);
    }
}

```

```

    }
    else if(running->pid && findProgramByPid(running->parentPid)-
>status == ProgramStatus::DEAD){
        printf("release zombie process PCB with pid %d\n",running-
>pid);
        releasePCB(running);
    }
}

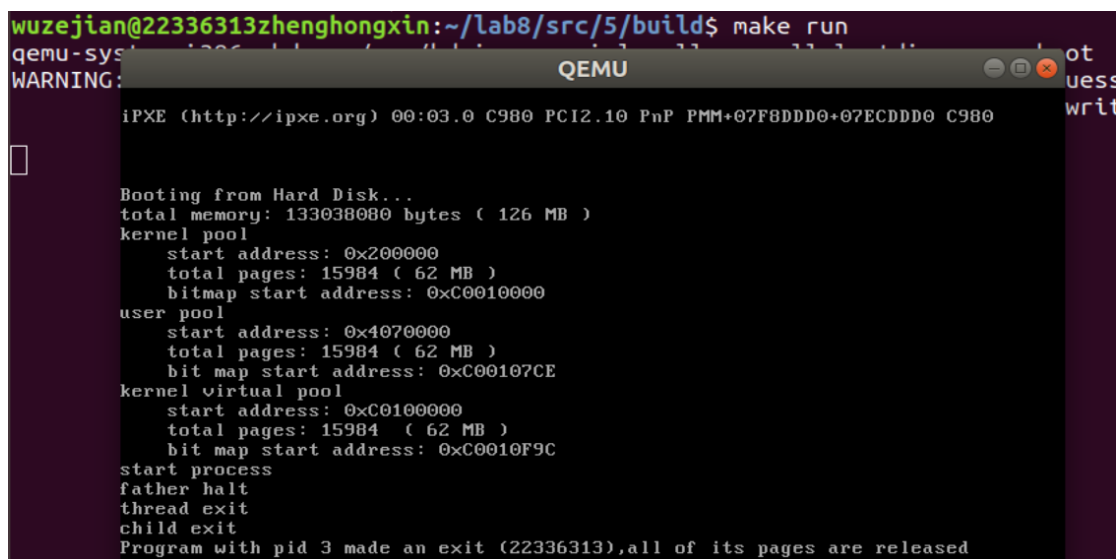
```

由于 PCB 初始化的时候，parentpid 默认为 0，只有在执行 fork 时，parentpid 才会被更新为父进程的 pid，所以依靠这点来判断其是否为某个父进程的子进程。然后我们再通过 pid 来查找父进程，查看父进程是否状态为 DEAD，如果是的话，则说明子进程已经是僵尸进程。

● 实验结果展示：

子任务 1：

测试 exit 的简单测试样例结果展示：



```

wuzajian@22336313zhenghongxin:~/lab8/src/5/build$ make run
qemu-system-i386 -hda /home/wuzajian/lab8/src/5/build/disk.img -smp 1 -m 128M -nographic
WARNING: Please do not use the -nographic option with the -usb option.
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DD0+07ECDD0 C980

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
father halt
thread exit
child exit
Program with pid 3 made an exit (22336313), all of its pages are released

```

子任务 3：

测试 wait 的简单测试样例结果展示：

```
wuzejian@2233631zhenghongxin:~/lab8/src/6/build$ make build
g++ -g -Wall -ma
el/setup.cpp ../
../src/utils/std
ld -o kernel.o
20000 -e enter_k
objcopy -O binar
dd if=mbr.bin of
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied
dd if=bootloader
记录了1+1 的读入
记录了1+1 的写出
580 bytes copied
thread exit
exit,pid:3
Program with pid 3 made an exit (2233),all of its pages are released
exit,pid:4
Program with pid 4 made an exit (6313),all of its pages are released
wait : pid:3,return value:2233
wait : pid:4,return value:6313
all child exit:program: 2
wuzejian@2233631
qemu-system-i386
```

```
wuzejian@22336313zhenghongxin:~/lab8/src/7/build$ make build
g++ -g -Wall -march
el/setup.cpp ../src
/utls/address_pool
ld -o kernel.o -mel
xt 0xc0020000 -e en
objcopy -O binary k
dd if=mbr.bin of=..
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0
dd if=bootloader.bi
记录了1+1 的读入
记录了1+1 的写出
580 bytes copied, 0
dd if=kernel.bin of=
记录了35+1 的读入
记录了35+1 的写出
18088 bytes (18 kB)
wuzejian@22336313zh
qemu-system-i386 -h
```

```
wuzejian@22336313zhenghongxin:~/lab8/src/7/build$ make build
g++ -g -Wall -m...
el/setup.cpp ../
/utils/address_...
ld -o kernel.o -iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980
xt 0xc0020000 -e
objcopy -O binar...
dd if=mbr.bin of=...
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied
dd if=bootloader...
记录了1+1 的读入
记录了1+1 的写出
580 bytes copied
dd if=kernel.bin of=...
记录了35+1 的读入
记录了35+1 的写出
18416 bytes (18...
wuzejian@22336313$
qemu-system-i386
WARNING: Image format was not specified for './run/hd.img' and probing guessed r
```

Section 5 实验总结与心得体会

通过本次实验，我深入理解了操作系统中用户态与内核态的概念及其重要性。学习了如何通过系统调用实现用户态到内核态的转换，并掌握了系统调用的底层实现机制。实验中，我实践了进程的创建和管理，包括进程控制块（PCB）的初始化、页目录表的配置，以及调度器的实现。此外，我还探索了 fork、wait 和 exit 系统调用的工作方式，理解了它们在进程生命周期管理中的作用。解决孤儿进程和僵尸进程的问题让我认识到了进程管理的复杂性以及资源回收的重要性。

实验过程中，我遇到了不少挑战，但通过不断尝试和调试，我加深了对操作系统原理的理解，并提高了我的编程和问题解决能力。使用 GDB 调试器进行系统调用和进程调度的调试，让我更加熟悉了底层调试技术。总之，这次实验不仅巩固了我的理论知识，也锻炼了我的实践技能，是一次宝贵的学习经历。

Section 6 附录：参考资料清单

1. [SYSU-2023-Spring-Operating-System: 中山大学 2023 学年春季操作系统课程 - Gitee.com](#)
2. [GDB 常用命令大全 GDB 命令详细解释 gdb 指令-CSDN 博客](#)
3. [lab8 • NelsonCheung/SYSU-2023-Spring-Operating-System - 码云 - 开源中国 \(gitee.com\)](#)

4. [Linux 下的 exit 函数和 wait 函数 试述 linux 中 wait 函数和 exit 函数的作用和它们之间的联系? -CSDN 博客](#)