LAB8 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：物理内存与虚拟内存管理

专业名称：计算机科学与技术

学生姓名：马福泉

学生学号：　23336179

实验地点：实验中心 B202

实验时间：　2025.6.12

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：汇编语言基础，计算机体系结构，操作系统基础，调试工具的使用（如 gbd）
- 实验环境：
  - 虚拟机版本/处理器型号：VMware-Ubuntu22.04.5/i386（32 位）
  - 代码编辑环境： 编辑器：VSCode
    插件：C/C++插件，汇编插件
  - 代码编译工具： 编译器：gcc
    工具链：binutils、make、qemu、nasm 等
    调试工具：gdb
  - 重要三方库信息：GNU binutils：工具集（如 as、ld）
    gdb：调试工具
    QEMU、Bochs：模拟器用于测试 等

Section 3 实验任务

- 实验任务 1：实现系统调用的方法

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?

- 实验任务 2：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. 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);
    ...
    }
```

自行测试通过后，说一说你的实现思路，并保存结果截图 。

- 实验任务 3：fork 的实现

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。

- 实验任务 4：wait & exit 的实现

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。

Section 4 实验步骤与实验结果

---------- **实验任务 1** ----------

- 任务要求：实现系统调用的方法
  复现指导书中“系统调用的实现”一节，并回答问题。
- 实验步骤：

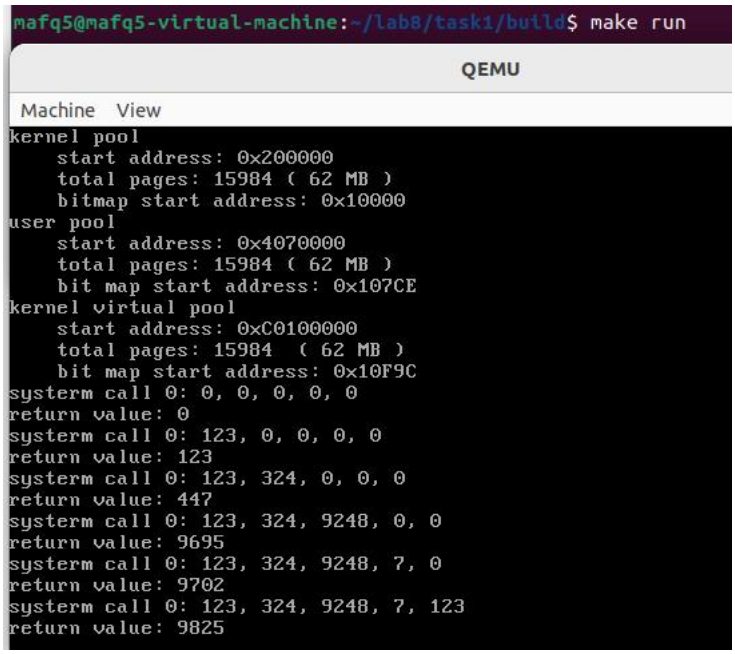
1. 阅读实验指导书，把代码复制到本地，运行复现

```
make clean
make build
make run
make debug
```

2. 使用 gdb 分析并回答问题

- 实验结果展示：

1. 复现结果如下：



2. 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。

(1) 特权级切换会导致栈切换。用户态的参数在用户栈中，而内核态使用内核栈，如果通过栈传递参数，内核代码无法访问用户栈中的参数。而寄存器在特权级切换时会被自动保存，可以安全地在不同特权级间传递数据。

(2) 这些代码通过 ebp 基址寻址，从调用者的栈帧中获取参数并存入寄存器。

```
mov eax, [ebp + 2 * 4]  ; 系统调用号(index)
mov ebx, [ebp + 3 * 4]  ; 第一个参数(first)
mov ecx, [ebp + 4 * 4]  ; 第二个参数(second)
mov edx, [ebp + 5 * 4]  ; 第三个参数(third)
mov esi, [ebp + 6 * 4]  ; 第四个参数(forth)
mov edi, [ebp + 7 * 4]  ; 第五个参数(fifth)
```

3. 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?

(1) 我们总结一下系统调用的流程，明确我们需要在什么地方设置断点。

- 用户程序调用 asm_system_call 函数
- asm_system_call 设置参数并触发 int 0x80 中断
- CPU 切换到内核态，保存上下文并跳转到 asm_system_call_handler
- 处理系统调用
- 恢复上下文并返回用户态

因此，我们需要在 asm_system_call 函数、int 0x80 中断、asm_system_call_handler 函数前后设置断点，查看栈的内容。

（2）启动

```
make clean
make
make debug
list
```

（3）设置断点并开始分析

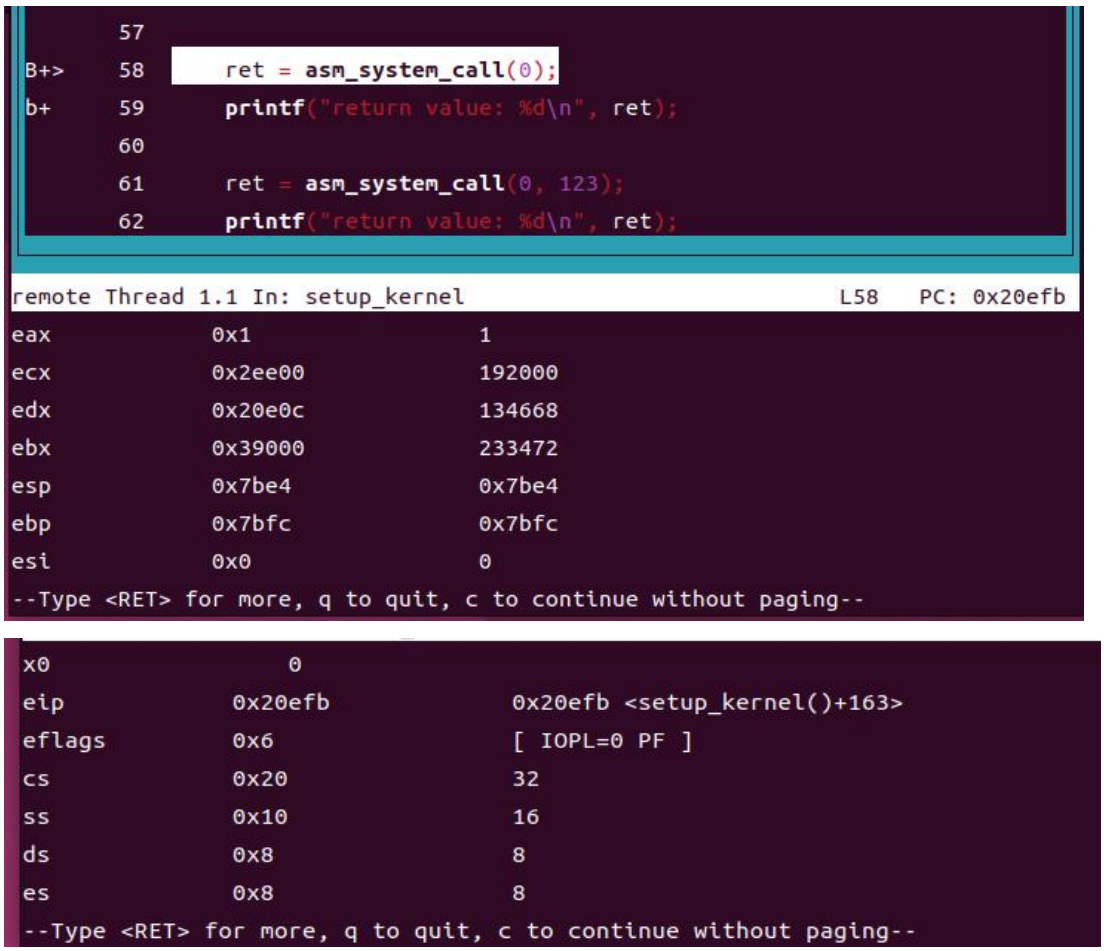
```
break setup.cpp:56
break setup.cpp:58
...
break setup.cpp:72
break setup.cpp:74
#我们仅以第一次和最后一次系统调用为例设计断点说明
```

```
../src/kernel/setup.cpp
b+     58      ret = asm_system_call(0);
b+     59      printf("return value: %d\n", ret);
       60
b+     61      ret = asm_system_call(0, 123);
       62      printf("return value: %d\n", ret);
       63
       64      ret = asm_system_call(0, 123, 324);
       65      printf("return value: %d\n", ret);
       66
       67      ret = asm_system_call(0, 123, 324, 9248);
       68      printf("return value: %d\n", ret);
       69
       70      ret = asm_system_call(0, 123, 324, 9248, 7);
       71      printf("return value: %d\n", ret);
       72
b+     73      ret = asm_system_call(0, 123, 324, 9248, 7, 123);
b+     74      printf("return value: %d\n", ret);
       75
       76      // 创建第一个线程
b+     77      int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
```

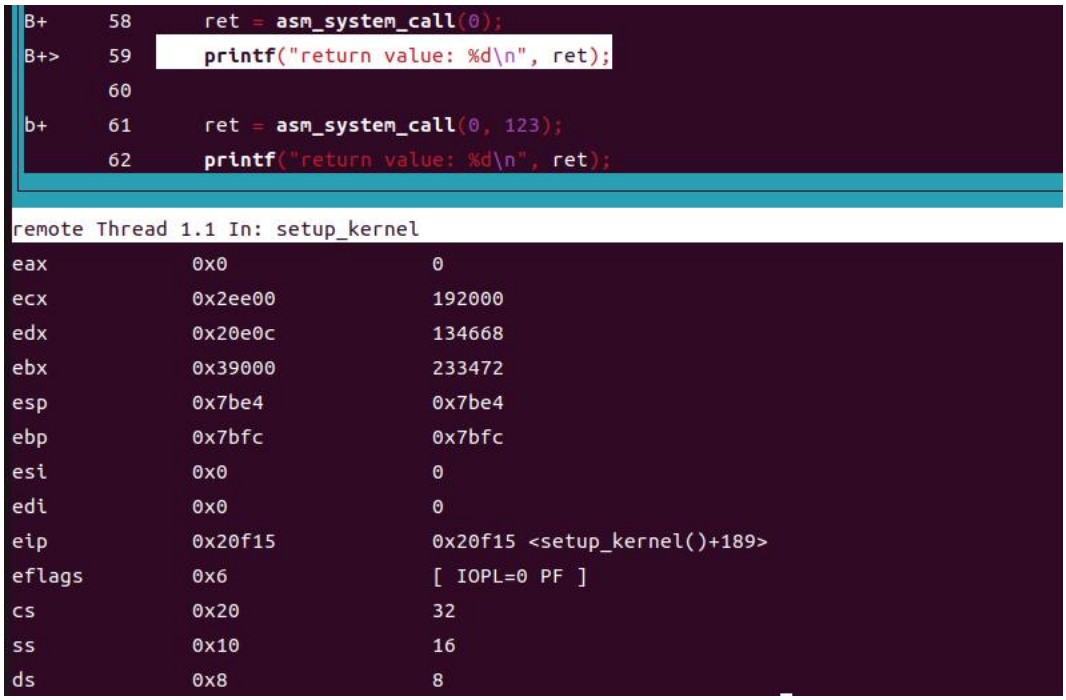
（4） 查看结果并分析

```
# 运行并查看状态
c
info registers
info registers cs ds es fs gs ss
x/10x $esp
c
```

断点 1：第 58 行（调用前）



断点 2：第 59 行（调用后）



断点 3：第 73 行（调用前）

```
B+>     73      ret = asm_system_call(0, 123, 324, 9248, 7, 123);
b+      74      printf("return value: %d\n", ret);

remote Thread 1.1 In: setup_kernel
eax            0x13                19
ecx            0xb8f9f             757663
edx            0x13                19
ebx            0x39000             233472
esp            0x7be4              0x7be4
ebp            0x7bfc              0x7bfc
esi            0x0                 0
edi            0x0                 0
eip            0x20feb             0x20feb <setup_kernel()+403>
eflags         0x6                 [ IOPL=0 PF ]
cs             0x20                32
ss             0x10                16
ds             0x8                 8
```

断点 4：第 74 行（调用后）

```
B+      73      ret = asm_system_call(0, 123, 324, 9248, 7, 123);
B+>     74      printf("return value: %d\n", ret);
        75
        76      // 创建第一个线程
        77      int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);

remote Thread 1.1 In: setup_kernel
eax            0x2661              9825
ecx            0xb8f9f             757663
edx            0x13                19
ebx            0x39000             233472
esp            0x7be4              0x7be4
ebp            0x7bfc              0x7bfc
esi            0x0                 0
edi            0x0                 0
eip            0x2100b             0x2100b <setup_kernel()+435>
eflags         0x6                 [ IOPL=0 PF ]
cs             0x20                32
ss             0x10                16
ds             0x8                 8
```

结果如上图：

- 特权级变化:

调用前: CPL = 0（内核态，因为 setup_kernel 本身运行在内核态）
int 0x80: 由于中断描述符 DPL=3，允许从用户态调用
处理中: CPL = 0（内核态处理系统调用）
返回后: CPL = 0（返回到内核态）

- 我们也可以进入系统调用内部进行查看,如下：

```
s#单步执行
b asm_syscall_0
b asm_system_call
b asm_system_call_handler
```

- ESP 与 TSS 关系：当前情况：ESP 保持在内核栈，不使用 TSS。如果有用户

态调用：ESP 会切换到 TSS.esp0 指向的内核栈

- 断点 1：第 58 行（调用前）

ESP = 0x7be4(内核栈位置)
EBP = 0x7bfc(setup_kernel 的栈帧)
EAX = 10
CS=0x20, DS=ES=FS=GS=SS=0x8

- 断点 2：第 59 行（调用后）

ESP = 恢复到调用前值
EAX = 0 (返回值)
ret = 0
其他寄存器恢复到调用前状态

- 断点 3：第 73 行（调用前）

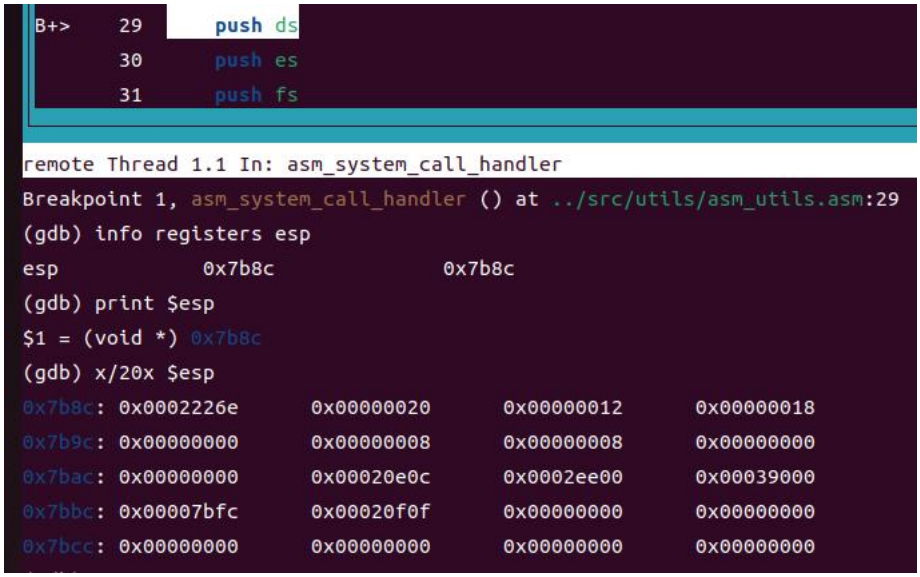
系统调用号: 0
参数 1: 123
参数 2: 324
参数 3: 9248
参数 4: 7
参数 5: 123

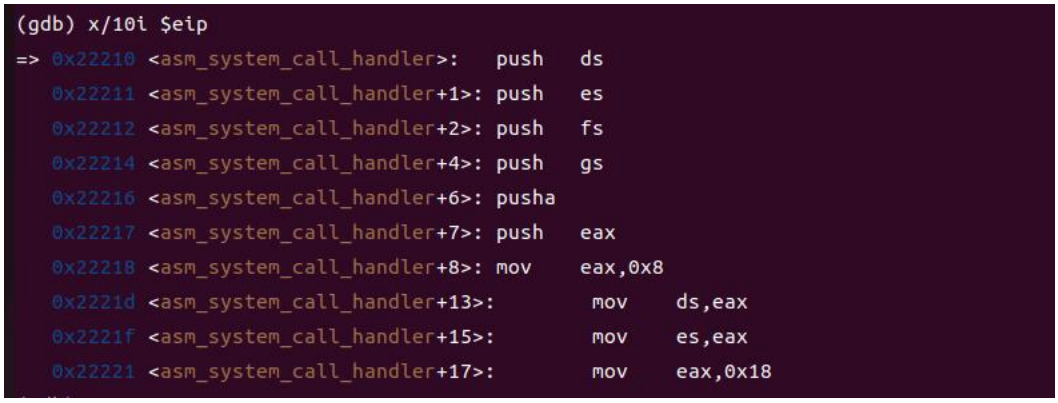
- 断点 4：第 74 行（调用后）

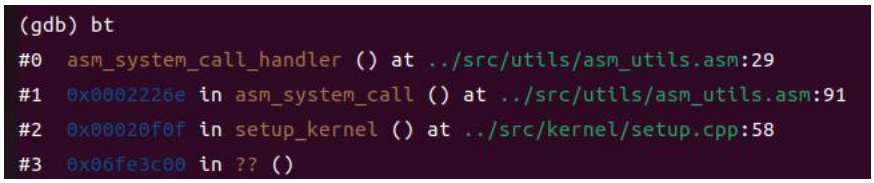
EAX = 9825 (123+324+9248+7+123)
ret = 9825
ESP 恢复到调用前

4. 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?

```
# 设置断点在 asm_system_call_handler 入口
break asm_system_call_handler
c
# 查看栈顶地址
info registers esp
print $esp
# 查看栈中内容(从栈顶开始的 20 个字)
x/20x $esp
# 查看栈顶的具体内容含义
x/10i $eip          # 查看当前指令
info registers      # 查看所有寄存器状态
bt                  # 查看调用栈
```







- 栈顶地址为 0x7b8c
- 栈顶内容为

偏移(ESP+n)	内容说明
ESP+0	原 EIP (返回地址)
ESP+4	原 CS
ESP+8	原 EFLAGS
ESP+12	保存的 GS
ESP+16	保存的 FS
ESP+20	保存的 ES
ESP+24	保存的 DS
ESP+28	保存的 EDI
ESP+32	保存的 ESI
ESP+36	保存的 EDX
ESP+40	保存的 ECX
ESP+44	保存的 EBX

| ESP+48 | 保存的 EBP
| ESP+52 | 函数返回地址
| ESP+56 | 系统调用号
| ESP+60 | 参数 1

- 为什么存放的是这些内容?

CPU 自动保存的中断上下文:当 int 0x80 中断触发时，CPU 自动执行：

```
push EFLAGS    ; 保存标志寄存器
push CS        ; 保存代码段选择子
push EIP       ; 保存返回地址(int 0x80 的下一条指令)
```

asm_system_call 中手动保存的上下文:

```
; 保存调用者上下文
push ebp       ; 保存栈帧指针
push ebx       ; 保存通用寄存器
push ecx
push edx
push esi
push edi
push ds        ; 保存段寄存器
push es
push fs
push gs
```

5. 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？

- 系统调用注册过程

在 setup_kernel() 函数中，我们可以看到系统调用的注册：

```
// 初始化系统调用
systemService.initialize();
// 设置 0 号系统调用
systemService.setSystemCall(0, (int)syscall_0);
```

这里将 syscall_0 函数注册为 0 号系统调用。

- SystemService 的实现机制

SystemService 类应该维护一个函数指针表来存储系统调用函数：

3. asm_system_call_handler 的查找逻辑

asm_system_call_handler 的实现逻辑应该是：

```
class SystemService {
private:
    // 系统调用函数指针表
    int (*systemCallTable[MAX_SYSTEM_CALL])(int, int, int, int, int);

public:
    void setSystemCall(int index, int functionAddress) {
        // 将函数地址转换为函数指针并存储在表中
```

```
            systemCallTable[index] = (int(*)(int, int, int, int,
int))functionAddress;
        }

        // 执行系统调用的核心函数
        int handleSystemCall(int index, int arg1, int arg2, int arg3, int arg4,
int arg5) {
            if (index >= 0 && index < MAX_SYSTEM_CALL &&
systemCallTable[index] != nullptr) {
                return systemCallTable[index](arg1, arg2, arg3, arg4, arg5);
            }
            return -1; // 无效的系统调用号
        }
    };
```

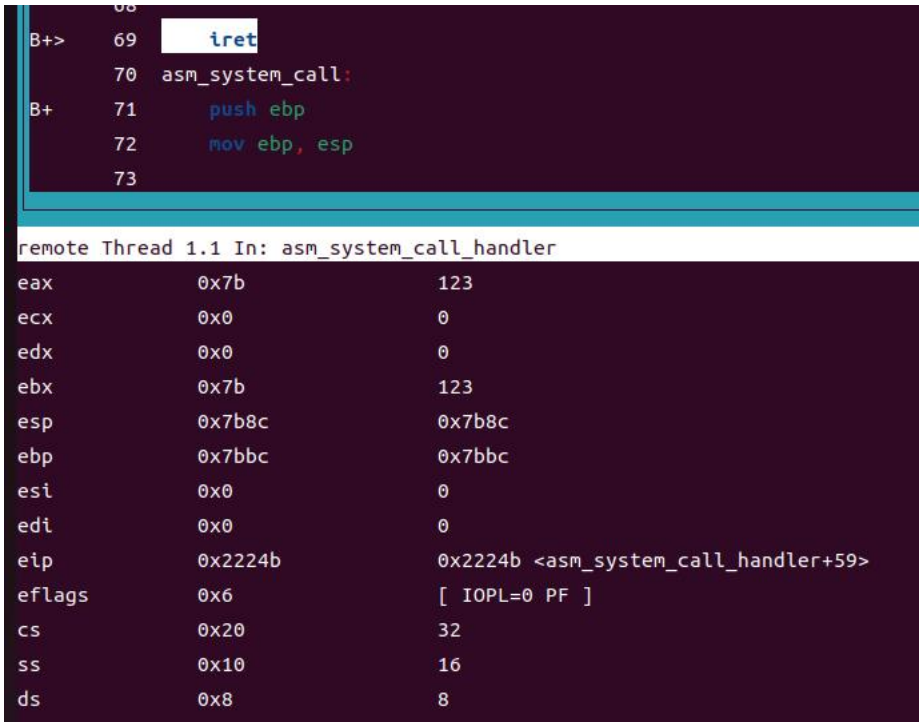
- 完整的函数查找流程

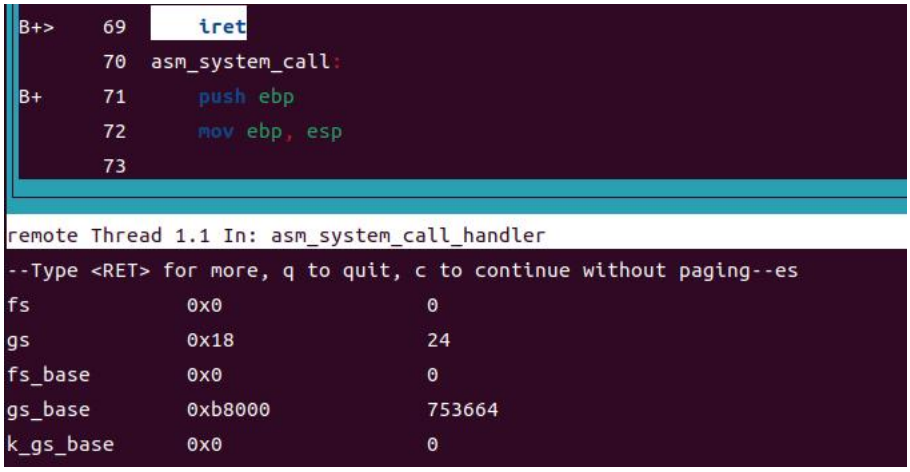
用户调用: asm_system_call(0, 123, 324)
↓
汇编函数设置寄存器:
EAX = 0 (系统调用号)
EBX = 123 (参数 1)
ECX = 324 (参数 2)
↓
触发中断: int 0x80
↓
CPU 跳转到: asm_system_call_handler
↓
汇编处理器调用: systemService.handleSystemCall(0, 123, 324, 0, 0, 0)
↓
C++函数查表: systemCallTable[0] -> syscall_0 函数指针
↓
执行函数: syscall_0(123, 324, 0, 0, 0)
↓
返回结果

6. 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？

```
break asm_system_call_handler
c
b 69
b 70
info registers
c
info registers
# 查看栈顶内容(iret 会从栈中恢复)
x/10x $esp
```

执行 iret 前：





执行 iret 后：

```
B+>    71      push ebp
       72      mov ebp, esp
       73
       74      push ebx
       75      push ecx
       76      push edx

remote Thread 1.1 In: asm_system_call
eax            0x10                16
ecx            0x2ee00             192000
edx            0x10                16
ebx            0x39000             233472
esp            0x7bc0              0x7bc0
ebp            0x7bfc              0x7bfc
esi            0x0                 0
edi            0x0                 0
eip            0x2224c             0x2224c <asm_system_call>
eflags         0x12                [ IOPL=0 AF ]
cs             0x20                32
ss             0x10                16
ds             0x8                 8
```

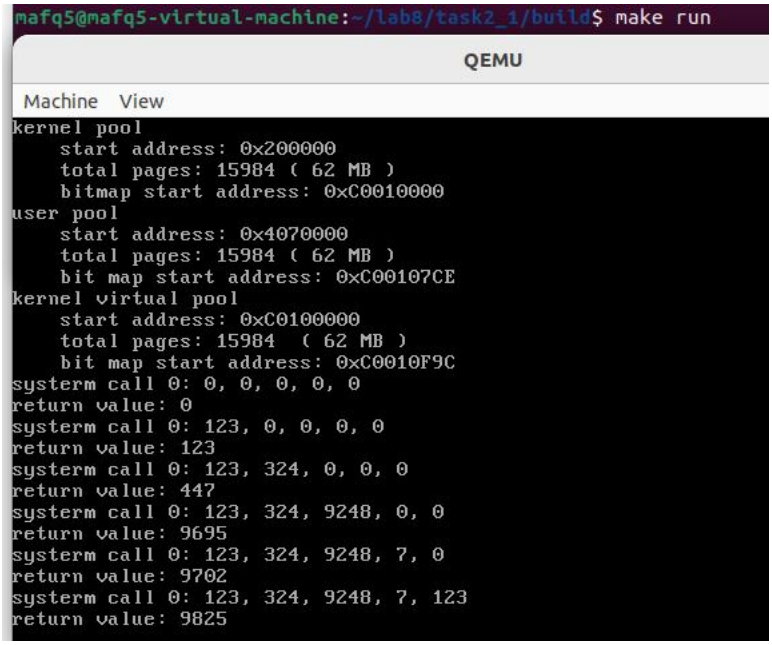
```
B+>    71      push ebp
       72      mov ebp, esp
       73
       74      push ebx
       75      push ecx
       76      push edx

remote Thread 1.1 In: asm_system_call
--Type <RET> for more, q to quit, c to continue without paging
fs             0x0                 0
gs             0x18                24
fs_base        0x0                 0
gs_base        0xb8000             753664
k_gs_base      0x0                 0
```

返回用户态时（即执行 `iret` 时），栈指针会恢复为用户态栈指针
段寄存器并未发现变化。

---------- 实验任务 2 ----------

- 任务要求：阅读实验指导书，把代码复制到本地，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。
- 实验步骤以及结果展示

1. 复现“进程的实现”，“进程的调度”，“第一个进程”,结果如下：



2. 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)

进程创建的三个步骤分析

- 基于线程创建进程的基础 PCB

```
在 executeProcess 函数中，首先调用 executeThread 创建基础 PCB：
int pid = executeThread((ThreadFunction)load_process,
                (void *)filename, filename, priority);
```

这里将 load_process 函数作为线程入口点，文件名作为参数。executeThread 函数会：

分配一个 4KB 的 PCB 页面

初始化基本的线程属性（名称、状态、优先级、ticks 等）

设置线程栈

将 PCB 添加到就绪队列

- 创建进程的页目录表

通过 createProcessPageDirectory 函数创建独立的虚拟地址空间：

process->pageDirectoryAddress = createProcessPageDirectory();

该函数的关键操作：

从内核地址池分配一页用于存储页目录表

将页目录表清零

复制内核目录项到高 1GB：将内核空间的页目录项复制到用户进程的页目录表

中，确保进程可以访问内核空间

设置自映射：页目录表的最后一项指向自身，用于页表管理

```
// 复制内核目录项到虚拟地址的高 1GB
int *src = (int *)(0xfffff000 + 0x300 * 4);  // 内核页目录表的 768 项开始
int *dst = (int *)(vaddr + 0x300 * 4);       // 新页目录表的 768 项开始
for (int i = 0; i < 256; ++i)
{
    dst[i] = src[i];  // 复制 256 个目录项(高 1GB)
}

// 自映射:最后一项指向页目录表自身
((int *)vaddr)[1023] = memoryManager.vaddr2paddr(vaddr) | 0x7;
```

- 创建进程的用户虚拟地址池

通过 createUserVirtualPool 函数为进程创建独立的虚拟地址管理：

```
bool res = createUserVirtualPool(process);
```

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 ret 后会跳转到 load process。

分析：

- ESP（栈指针寄存器）

作用：指向当前栈顶

关键性：asm_switch_thread 通过切换 ESP 来实现线程切换

预期变化：从当前线程栈 → 目标线程栈

- EIP（指令指针寄存器）

作用：指向下一条要执行的指令

关键性：决定程序跳转到哪里执行

预期变化：执行 ret 后跳转到栈顶保存的返回地址

- EAX 寄存器

在切换中的作用：临时保存 PCB 指针

预期值：指向目标线程的 PCB 地址

- 进程创建时的栈帧构造

当调用 programManager.executeProcess() 创建新进程时，系统需要为这个进程构造一个初始的执行环境,这包括在进程的内核栈上构造一个"假的"中断栈帧

- 中断返回机制

asm_switch_thread 最终通过 iret 指令恢复进程执行

iret 会从栈上恢复 CS:EIP、SS:ESP、EFLAGS 等寄存器

这些值决定了进程恢复执行的位置

- load_process 的作用

load_process 是一个包装函数，负责：

设置进程的执行环境

跳转到真正的进程入口点

处理进程执行完毕后的清理工作

- 进程 PCB 初始化时，EIP 被设置为 load_process 的地址

load_process 作为中介，负责最终跳转到真正的进程入口点

这种设计允许系统在进程启动时进行必要的环境设置和资源管理

- 设置断点进入

```
# 在关键位置设置断点
(gdb) b asm_switch_thread
(gdb) b setup.cpp:76
(gdb) c
```

- 检查调用前的状态

```
# 到达 setup.cpp:76 断点时
(gdb) print firstThread
# 查看 PCB 结构
(gdb) x/1x firstThread       # PCB 中保存的 ESP 值
(gdb) x/20x *(void**)firstThread  # 查看保存的栈内容
# 当前状态
(gdb) info registers esp eip eax
```

```
(gdb) print firstThread
$1 = (PCB *) 0x100000
(gdb) print/x firstThread
$2 = 0x100000
```

```
(gdb) info registers esp eip eax
esp            0x7be4             0x7be4
eip            0xc0021454         0xc0021454 <setup_kernel()+207>
eax            0x0                0
```

```
(gdb) x/1x firstThread
0x100000:       Cannot access memory at address 0x100000
(gdb) x/20x *(void**)firstThread
Cannot access memory at address 0x100000
```

esp：栈指针寄存器的值是 0x7be4，指向当前线程的栈顶。

eip：指令指针寄存器的值是 0xc0021454，指向 setup_kernel 函数的第 207 行。这表明程序正在执行 setup_kernel 函数中的代码。

eax：通用寄存器的值是 0x0，表示没有特定的值被存储在这个寄存器中。

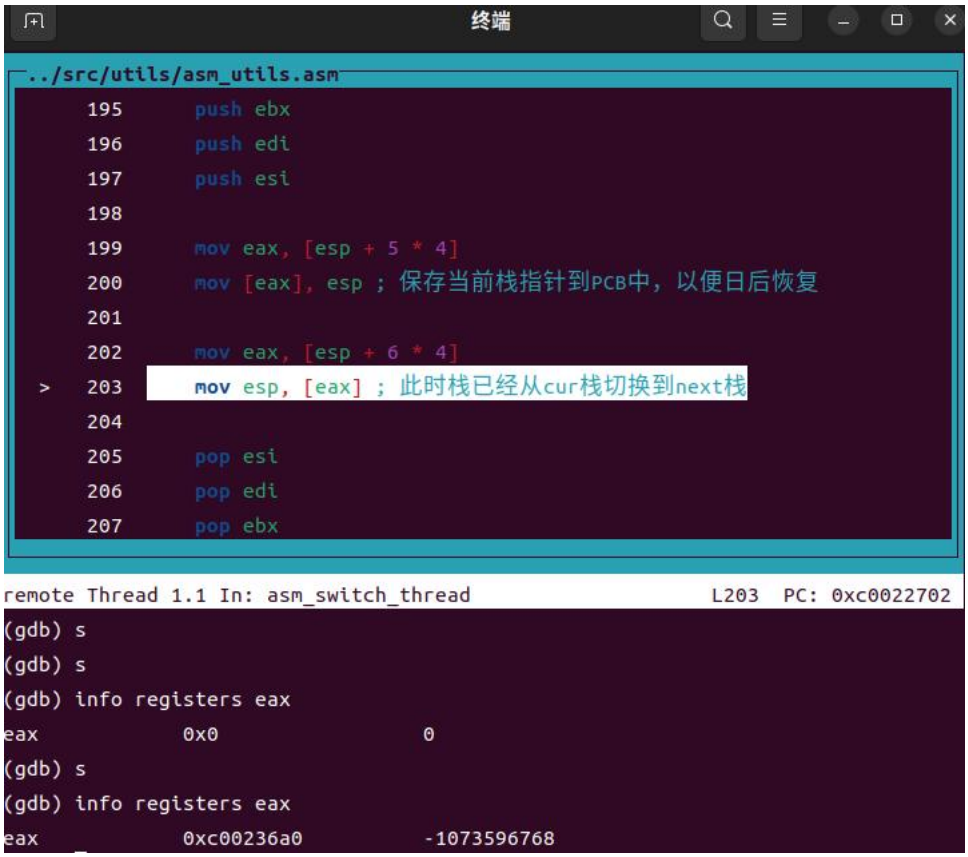
firstThread 指向的地址 0x100000 无法访问，这可能是因为该地址没有被正确初始化或者不在当前进程的内存映射范围内。程序正在执行 setup_kernel 函数，栈指针和指令指针的值表明程序的执行状态正常，但 eax 寄存器没有存储特定的值。

- 进入 asm_switch_thread 分析

```
(gdb) s  # 进入函数
(gdb) x/6x $esp
# 执行到 mov eax, [esp + 5 * 4]
(gdb) stepi
(gdb) info registers eax
# eax 现在包含当前 PCB 指针(0, 因为传入的 cur 是 0)
执行到 mov eax, [esp + 6 * 4]
(gdb) stepi
(gdb) info registers eax
# eax 现在包含 firstThread 指针
```

```
 >   199     mov eax, [esp + 5 * 4]
     200     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
     201
remote Thread 1.1 In: asm_switch_thread                    L199  PC: 0xc0
(gdb) s
(gdb) s
(gdb) x/6x $esp
0x7bc0: 0x00000000      0x00000000      0x00039000      0x00007bfc
0x7bd0: 0xc002149f      0x00000000
(gdb) info registers eax
eax            0xc00236a0          -1073596768
```

```
 >   200     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复
     201
remote Thread 1.1 In: asm_switch_thread                    L200  PC: 0
0x7bc0: 0x00000000      0x00000000      0x00039000      0x00007bfc
0x7bd0: 0xc002149f      0x00000000
(gdb) info registers eax
eax            0xc00236a0          -1073596768
(gdb) s
(gdb) info registers eax
eax            0x0                 0
```



保存当前线程状态：

mov [eax], esp：这条指令将当前的栈指针（esp）保存到由 eax 寄存器指向的内存地址中。这个地址是当前线程的进程控制块（PCB）中的栈指针位置。这样做的目的是为了在线程切换回来时能够恢复到正确的栈状态。

此时，eax 寄存器包含了当前线程 PCB 的地址，而 esp 寄存器指向当前线程的栈顶。

加载下一个线程状态：

mov eax, [esp + 5 * 4]：这条指令从栈中加载下一个要执行的线程的 PCB 地址到 eax 寄存器中。这个地址是作为函数参数传入的，位于栈帧中。

然后，执行 mov [eax], esp：将当前的栈指针（esp）保存到下一个线程的 PCB 中，为即将进行的线程切换做准备。

切换到下一个线程的栈：

mov esp, [eax]：这条指令将下一个线程的栈指针从其 PCB 中加载到 esp 寄存器中。这样，栈指针就指向了下一个线程的栈，完成了栈的切换。

此时，eax 寄存器被清零（mov eax, 0），准备后续操作。

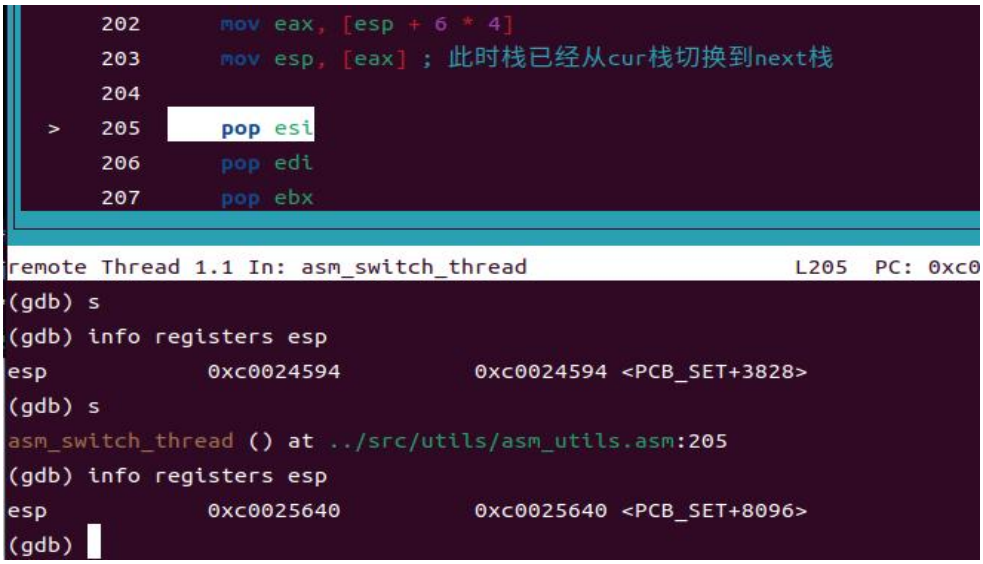
- 关键的栈切换分析

```
# 执行栈切换指令前
(gdb) x/1x $eax  # 查看 firstThread PCB 中保存的 ESP
# 其他内核为新线程分配的栈地址
# 执行 mov esp, [eax] - 这是关键！
(gdb) s
(gdb) info registers esp
# ESP 现在指向新线程的栈
```

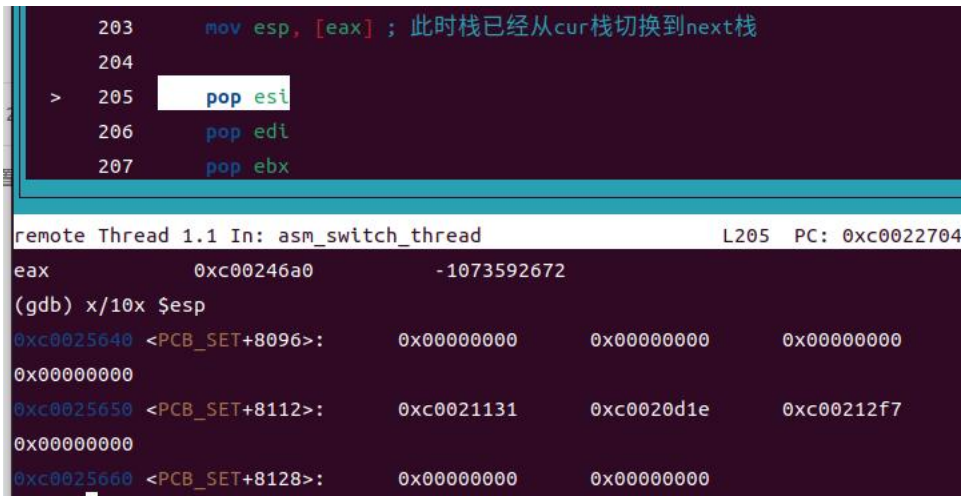
```
# 查看新栈的内容
(gdb) x/10x $esp
# 执行完所有 pop 操作后查看返回地址
(gdb) stepi  # pop esi
(gdb) stepi  # pop edi
(gdb) stepi  # pop ebx
(gdb) stepi  # pop ebp
(gdb) x/1x $esp  # 现在 ESP 指向返回地址
# 检查这个地址是什么函数
(gdb) info symbol 0x******
```

查看 firstThread PCB 中保存的 ESP

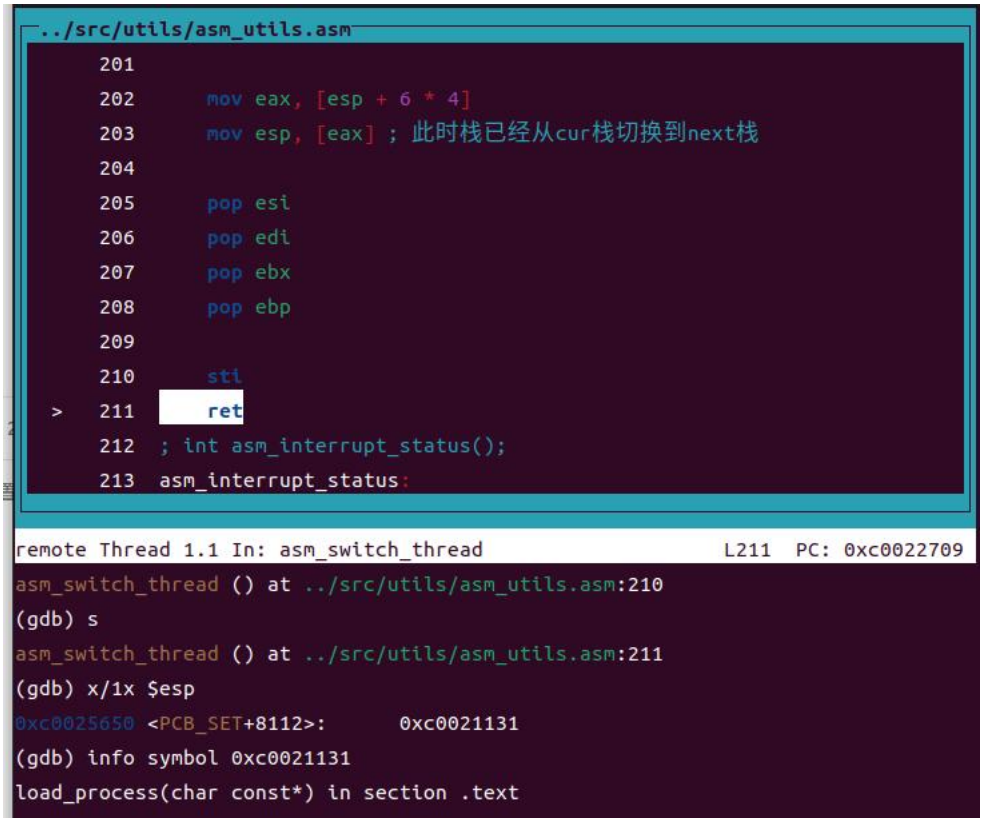
执行 mov esp, [eax] - 这是关键!
切换前后线程的栈地址：



查看新栈的内容 ：



执行完所有 pop 操作后查看返回地址并且检查这个地址是什么函数：



通过这个详细的调试过程，你会清楚地看到：
asm_switch_thread 本身工作正常，它正确地切换了栈
问题在于新线程的栈是人工构造的，返回地址指向 load_process
这是操作系统的设计选择，通过 load_process 来统一处理新进程的启动流程
这种设计允许系统在进程真正开始执行前进行必要的初始化工作。

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。

- 关键断点设置:

```
# 断点 1:ProcessStartStack 设置完成后
break load_process

# 断点 2:asm_start_process 入口
break asm_start_process

# 启动
C
s
```

- 分析 ProcessStartStack 设置:

```
在 load_process 断点处:
# 查看 ProcessStartStack 内容
print interruptStack
x/15w interruptStack
info registers
c
```

eip: 确认跳转目标
cs/ss: 确认特权级设置(低 2 位=3 表示用户态)
eflags: 确认中断开启状态
esp: 确认用户栈设置

- 分析 asm_start_process 执行

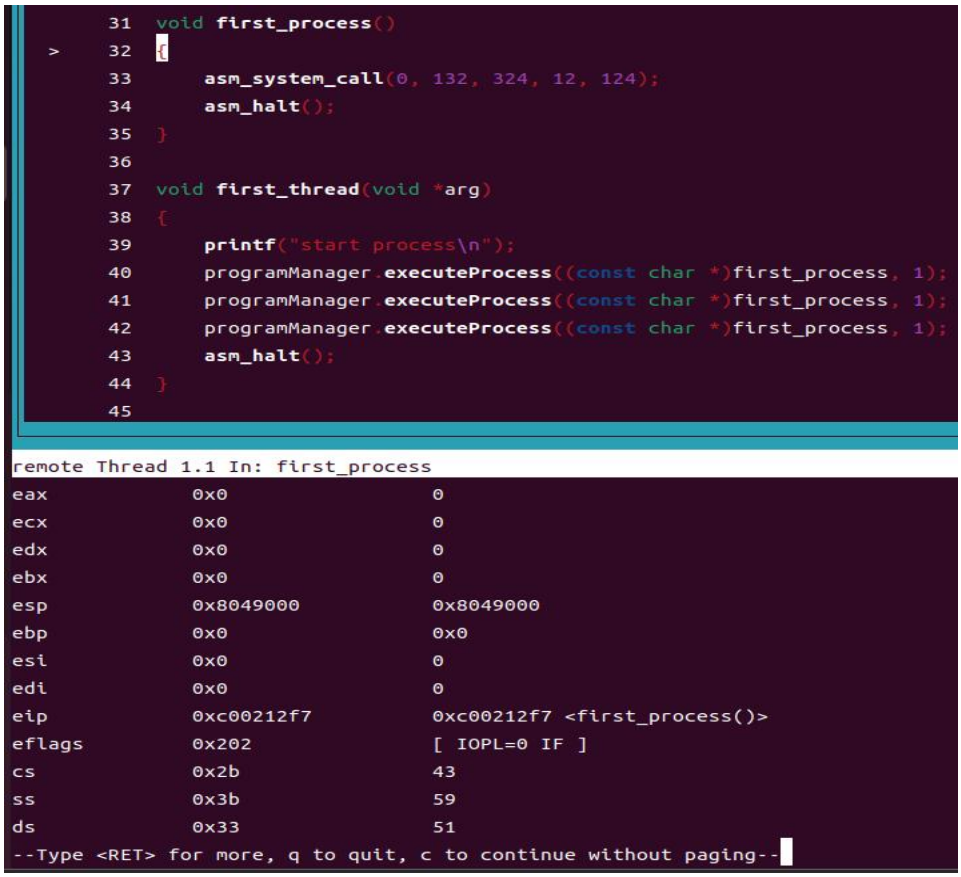
```
在 asm_start_process 断点处:
# 当前状态(内核态)
print/x $cs & 3    # 当前特权级, 应该=0
print/x $esp       # 当前栈指针

# 执行 mov esp, eax (切换到 ProcessStartStack)
stepi 2
print/x $esp       # 新栈指针

# 执行到 iret 前
s
info registers

# 执行 iret
S
Info registers
```

```
      289  void load_process(const char *filename)
B+>   290  {
      291      interruptManager.disableInterrupt();
      292
      293      PCB *process = programManager.running;
      294      ProcessStartStack *interruptStack =
```

```
remote Thread 1.1 In: load_process
eax            0xc00246a0          -1073592672
ecx            0x1                 1
edx            0x200000            2097152
ebx            0x0                 0
esp            0xc0025654          0xc0025654 <PCB_SET+8116>
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0xc0021131          0xc0021131 <load_process(char const*)>
eflags         0x282               [ IOPL=0 IF SF ]
cs             0x20                32
ss             0x10                16
ds             0x8                 8
```

```
B+    41      mov eax, dword[esp+4]
      42      mov esp, eax
      43      popad
      44      pop gs;
      45      pop fs;
      46      pop es;
      47      pop ds;
      48
  >   49      iret
      50
      51  ; void asm_ltr(int tr)
```

```
remote Thread 1.1 In: asm_start_process
eax            0x0                 0
ecx            0x0                 0
edx            0x0                 0
ebx            0x0                 0
esp            0xc002568c          0xc002568c <PCB_SET+8172>
ebp            0x0                 0x0
esi            0x0                 0
edi            0x0                 0
eip            0xc002261d          0xc002261d <asm_start_process+13>
eflags         0x92                [ IOPL=0 SF AF ]
cs             0x20                32
ss             0x10                16
ds             0x33                51
```



分析：

- ProcessStartStack 结构

从 process.h 可以看到：

```
struct ProcessStartStack
{
    int edi, esi, ebp, esp_dummy;
    int ebx, edx, ecx, eax;
    int gs, fs, es, ds;
    int eip, cs, eflags, esp, ss;
};
```

- load_process 函数设置

在 program.cpp 的 load_process 函数中：

```
void load_process(const char *filename)
{
    // ...设置各种寄存器值
    interruptStack->eip = (int)filename;  // 用户程序入口
    interruptStack->cs = programManager.USER_CODE_SELECTOR;   // 用户代码段
    interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1); // 开中断
    interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1) + PAGE_SIZE;
    interruptStack->ss = programManager.USER_STACK_SELECTOR;  // 用户栈段
```

```
    asm_start_process((int)interruptStack);
}
```

- asm_start_process 实现

在 asm_utils.asm 中：

```
asm_start_process:
    mov eax, dword[esp+4]  ; 获取 ProcessStartStack 地址
    mov esp, eax           ; 切换栈指针到 ProcessStartStack
    popad                  ; 恢复通用寄存器
    pop gs; pop fs; pop es; pop ds;  ; 恢复段寄存器
    iret                   ; 中断返回, 实现特权级转换
```

ProcessStartStack 模拟了中断栈帧，iret 指令从栈中弹出 EIP、CS、EFLAGS、ESP、SS 来实现特权级转换。

特权级转换：iret 前 cs 为 0x20,CPL=0，iret 后,cs 为 0x43,CPL=3。能清楚看到 ProcessStartStack 如何巧妙地利用 iret 指令的栈帧格式，实现从 Ring 0 到 Ring 3 的无缝转换。

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？

- 添加了页表切换机制

```
void ProgramManager::schedule()
{
    // ...existing code...

    // 关键修改:在切换到下一个进程前激活进程页表
    activateProgramPage(next);

    asm_switch_thread(cur, next);
    // ...existing code...
}
```

- 新增的 activateProgramPage 函数

```
void ProgramManager::activateProgramPage(PCB *program)
{
    int paddr = PAGE_DIRECTORY;

    if (program->pageDirectoryAddress)
    {
        // 设置 TSS 中的内核栈指针
        tss.esp0 = (int)program + PAGE_SIZE;
        // 切换到进程的页目录表
        paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);
    }

    // 更新 CR3 寄存器, 切换页表
    asm_update_cr3(paddr);
}
```

- ProcessStartStack 的巧妙设置，前面已经讨论
- asm_start_process 的实现

```
asm_start_process:
    mov eax, dword[esp+4]  ; 获取 ProcessStartStack 地址
    mov esp, eax           ; 切换到 ProcessStartStack
    popad                  ; 恢复通用寄存器
    pop gs                 ; 恢复段寄存器
    pop fs
    pop es
    pop ds
    iret                   ; 关键:实现特权级 0→3 的转换
```

- 修改效果：

(1) 实现虚拟内存隔离

每个进程拥有独立的页目录表

通过 activateProgramPage() 切换页表，确保进程间内存隔离

使用 CR3 寄存器切换实现地址空间隔离

(2) 支持用户态进程

创建用户态段描述符（DPL=3）

设置 ProcessStartStack 支持特权级切换

使用 iret 指令从内核态（Ring 0）切换到用户态（Ring 3）

(3) 3 建立完整的进程上下文

ProcessStartStack 包含了 iret 所需的完整上下文

正确设置 EFLAGS、CS、SS 等关键寄存器

为用户进程分配独立的栈空间

(4) TSS 支持

设置 tss.esp0 为进程的内核栈

支持用户态到内核态的切换（系统调用、中断）

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 :

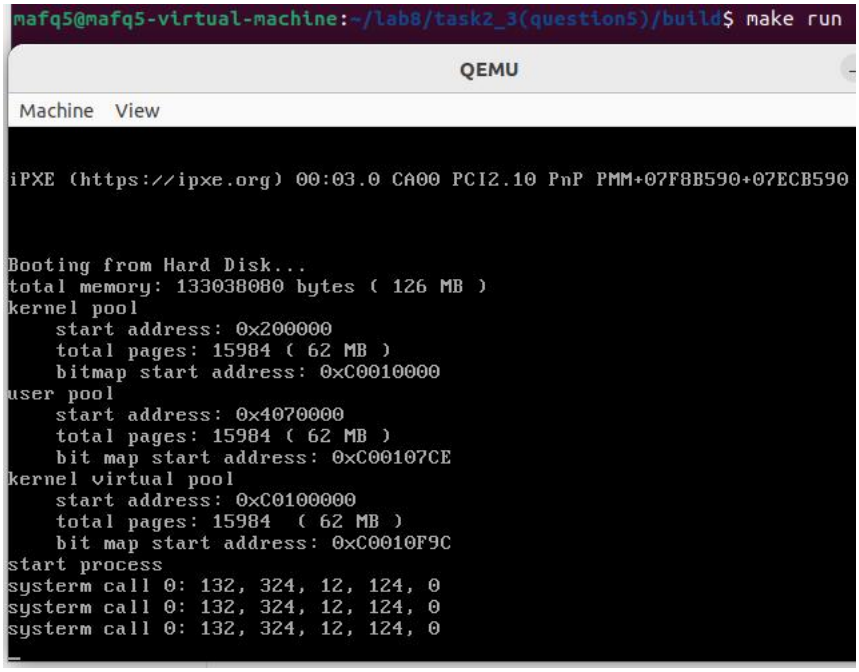
```
PCB *findProgramByPid(int pid);
```

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);
    ...
    }
```

- 自行测试通过后，说一说你的实现思路，并保存结果截图 。



实现了 findProgramByPid 函数

```
    PCB *ProgramManager::findProgramByPid(int pid)
    {
        // 检查 pid 是否在有效范围内
        if (pid < 0 || pid >= MAX_PROGRAM_AMOUNT)
        {
            return nullptr;
        }

        // 检查该 pid 对应的 PCB 是否已分配
        if (!PCB_SET_STATUS[pid])
        {
            return nullptr;
        }

        // 计算 PCB 地址
        PCB *pcb = (PCB *)((int)PCB_SET + PCB_SIZE * pid);

        // 验证 PCB 的 pid 字段是否匹配
        if (pcb->pid != pid)
        {
            return nullptr;
        }
```

```
    return pcb;
}
```

在头文件中添加了函数声明

```
class ProgramManager
{
    // ...existing code...

    // 根据 pid 查找对应的 PCB
    PCB *findProgramByPid(int pid);
};
```

3 在 executeProcess 中使用了安全的查找方式

```
int ProgramManager::executeProcess(const char *filename, int priority)
{
    // ...existing code...

    // 找到刚刚创建的 PCB
    PCB *process = findProgramByPid(pid);  // 使用安全的查找方式
    if (!process)
    {
        interruptManager.setInterruptStatus(status);
        return -1;
    }

    // ...existing code...
}
```

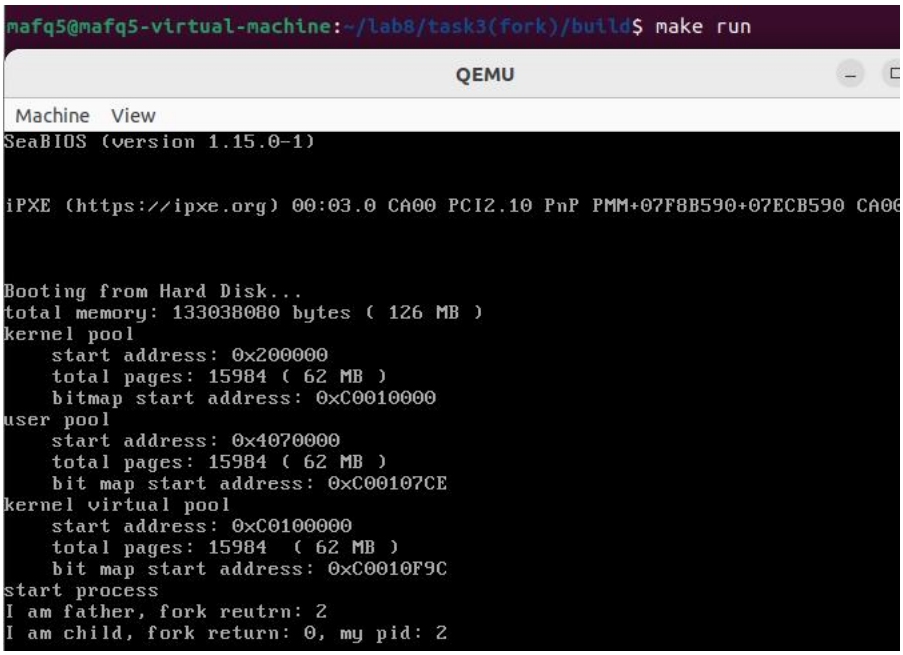
- 修改效果：

线程安全：避免了在多线程环境下链表操作的竞态条件
错误检查：添加了完整的边界检查和状态验证
可靠性：通过 pid 直接定位，避免了链表结构变化导致的问题
一致性：确保找到的 PCB 确实是刚创建的那个进程

---------- 实验任务 3 ----------

- 任务要求：fork 的实现，复现“fork”一小节的内容，并回答问题:

复现“fork”一小节的内容:



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。

- fork 函数采用了以下基本思路：

禁用中断保护：防止 fork 过程被打断
验证调用者：确保只有用户进程能调用 fork（内核线程不能）
创建空子进程：调用 executeProcess("", 0) 创建基础的 PCB 结构
完整复制父进程：通过 copyProcess 复制所有必要的进程信息
恢复中断并返回：父进程返回子进程 PID，子进程返回 0

- 四个关键问题的解决方案

(1) 如何让子进程返回 0，父进程返回子进程 PID？

解决方案：在 copyProcess 函数中巧妙设置子进程的 eax 寄存器：

```
    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;  // 关键:子进程返回 0

        // ...
    }
```

父进程：fork() 函数正常返回子进程的 PID

子进程：通过设置 ProcessStartStack 中的 eax = 0，当子进程恢复执行时，fork() 系统调用返回 0

(2) 如何复制进程的虚拟地址空间？

解决方案：分三个层次完整复制整个虚拟地址空间：

a) 复制页目录表结构

```
    // 复制页目录表
    for (int i = 0; i < 768; ++i)  // 只复制用户空间(0-3GB)
    {
        if (!(parentPageDir[i] & 0x1))
            continue;

        // 为子进程分配新的页表
        int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER,
1);
        int pde = parentPageDir[i];

        asm_update_cr3(childPageDirPaddr); // 切换到子进程地址空间
        childPageDir[i] = (pde & 0x00000fff) | paddr;
        memset(pageTableVaddr, 0, PAGE_SIZE);
        asm_update_cr3(parentPageDirPaddr); // 切换回父进程
    }
```

b) 复制页表和物理页内容

```
    // 复制页表和物理页
    for (int i = 0; i < 768; ++i)
    {
        for (int j = 0; j < 1024; ++j)
        {
            if (!(pageTableVaddr[j] & 0x1))
                continue;

            // 分配新物理页
            int paddr =
memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);

            // 使用中转页复制内容
            void *pageVaddr = (void *)((i << 22) + (j << 12));
            memcpy(pageVaddr, buffer, PAGE_SIZE);  // 父进程页→中转页

            asm_update_cr3(childPageDirPaddr);
            pageTableVaddr[j] = (pte & 0x00000fff) | paddr;
            memcpy(buffer, pageVaddr, PAGE_SIZE);  // 中转页→子进程页
            asm_update_cr3(parentPageDirPaddr);
        }
    }
```

(3)　　如何处理写时复制（Copy-on-Write）？

当前实现：采用了 立即完全复制 的策略，而非写时复制：

优点：实现简单，逻辑清晰，避免了复杂的页面错误处理

缺点：内存使用效率较低，fork 操作较慢

如果要实现 COW，需要：

```
// 每个物理页都立即分配和复制
int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
memcpy(pageVaddr, buffer, PAGE_SIZE);  // 立即复制内容
```

将父子进程的页面都标记为只读

在页面错误中断中检测写操作

动态分配新页面并复制内容

4. 如何正确设置子进程的执行上下文？

解决方案：精确复制和设置子进程的执行环境：

a) 复制进程控制信息

```
// 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);
```

b) 设置正确的栈结构

```
// 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;  // 子进程从这里开始执行
child->stack[5] = 0;                       // 返回地址
child->stack[6] = (int)childpss;           // 参数：ProcessStartStack
```

c) 复制虚拟地址池

```
// 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);
```

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？

```
启动调试并设置断点
(gdb) b ProgramManager::copyProcess
(gdb) b asm_switch_thread
(gdb) c
```

```
# 2. 在 copyProcess 中查看子进程栈设置
(gdb) c
(gdb) next 10   # 执行到栈设置完成
(gdb) print/x child
(gdb) print/x child->stack

# 3. 在调度切换时查看栈状态
(gdb) c   # 到 asm_switch_thread
(gdb) s 3    # 执行到栈切换指令
(gdb) info registers esp
(gdb) x/7x $esp
```

- 从 GDB 输出可以看到：

ESP 地址: 0xc025d48
栈中保存的内容（从 ESP 开始）：
ESP+0 (0xc025d48): 0x00000000 - esi 寄存器
ESP+4 (0xc025d4c): 0x00000000 - edi 寄存器
ESP+8 (0xc025d50): 0x00000000 - ebx 寄存器
ESP+12 (0xc025d54): 0xc0025d84 - ebp 寄存器
ESP+16 (0xc025d58): 0xc0020d04 - asm_start_process 函数地址
ESP+20 (0xc025d5c): 0xc0024e00 - 返回地址
ESP+24 (0xc025d60): 0xc0023e00 - asm_start_process 的参数(childpss)
这个结果完全符合 copyProcess 函数中的栈设置代码：

```
// 准备执行 asm_switch_thread 的栈的内容
child->stack = (int *)childpss - 7;
child->stack[0] = 0;                          // esi = 0x00000000
child->stack[1] = 0;                          // edi = 0x00000000
child->stack[2] = 0;                          // ebx = 0x00000000
child->stack[3] = 0;                          // ebp = 0x00000000
child->stack[4] = (int)asm_start_process; // 返回地址 = 0xc0020d04
child->stack[5] = 0;                          // 返回地址
child->stack[6] = (int)childpss;          // 参数 = childpss 地址
```

当 asm_switch_thread 执行 mov esp, [eax]时，ESP 被设置为子进程的栈指针 0xc025d48，栈中的内容正是为了让子进程从 asm_start_process 开始执行而精心构造的。

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。

1. 设置调试断点
设置关键断点

```
(gdb) b ProgramManager::schedule          # 调度器入口
(gdb) b asm_switch_thread                 # 线程切换
(gdb) b asm_start_process                 # 进程启动
```

```
(gdb) b syscall_fork                    # fork 系统调用
(gdb) b first_process                   # 用户进程入口
(gdb) c
```

2. 子进程第一次被调度执行

在调度器中捕获子进程:

```
(gdb) p/x running->pid                  # 查看当前进程 PID
(gdb) p/x next->pid                     # 查看即将调度的进程 PID
(gdb) x/8x next->stack                  # 查看子进程栈内容
```

3. 子进程栈切换分析

```
(gdb) s
(gdb) info registers esp eip            # 查看栈指针和指令指针
(gdb) x/16x $esp                        # 查看栈内容
 4. asm_start_process 执行分析
(gdb) s
(gdb) x/20x $esp                        # 查看 ProcessStartStack 内容
(gdb) info registers                    # 查看所有寄存器状态
5. 子进程从 fork 返回
在 first_process 中:
(gdb) stepi
(gdb) p/x $eax                          # fork 返回值
(gdb) x/i $eip                          # 当前指令
```

父进程返回过程:

(1) 直接返回: 从`syscall_fork`直接返回到用户态

(2) 返回值: `eax = 子进程 PID` (非 0 值)

(3) 无栈切换 继续在当前栈上执行

(4) 无段寄存器变化: 保持原有段寄存器值

(5) 页目录不变: 继续使用原页目录表

子进程返回过程:

(1) 间接返回: 通过调度器→`asm_start_process`→`iret`返回

(2) 返回值: `eax = 0` (在`copyProcess`中预设)

(3) 完整栈切换: 从内核栈切换到用户栈

(4) 段寄存器重新加载: 从`ProcessStartStack`中恢复

(5) 页目录切换:使用独立的页目录表

父进程是同步返回，子进程是异步返回(通过调度机制)。

3. 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

fork 返回值的设置机制如下：

- 父进程返回值：子进程 PID

在 ProgramManager::fork() 中:

```
int ProgramManager::fork()
{
    int pid = executeProcess("", 0);   // 创建子进程，返回 PID
    // ... 复制进程 ...
```

```
        return pid;   // 直接返回子进程 PID
    }
```

父进程通过系统调用直接返回，eax 寄存器保存子进程 PID。

- 子进程返回值：0

```
    在 copyProcess 中预设:
    bool ProgramManager::copyProcess(PCB *parent, PCB *child)
    {
        ProcessStartStack *childpps = ...;
        // 关键:预设子进程 fork 返回值为 0
        childpps->eax = 0;
        // ...
    }
```

子进程被调度时恢复:

- 差异：

父进程: 系统调用同步返回，eax 直接保存子进程 PID

子进程: 创建时预设返回值 0，调度时通过 ProcessStartStack 恢复 eax=0

这实现了 fork 的经典语义：一次调用，两次返回 - 父进程返回子 PID，子进程返回 0。

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。

- 切换过程

保存当前进程状态：通过 asm_switch_thread 保存当前进程的栈指针和寄存器

切换页目录表：通过 asm_update_cr3 将新进程的页目录表物理地址加载到 CR3

更新 TSS：设置新进程的内核栈地址到 tss.esp0

恢复新进程状态：通过 asm_switch_thread 恢复新进程的栈指针和寄存器

其中虚拟地址空间切换的关键步骤

- 进程调度准备

ProgramManager::schedule 方法中：

```
    ListItem *item = readyPrograms.front();
    PCB *next = ListItem2PCB(item, tagInGeneralList);
    PCB *cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.pop_front();
```

- 页目录表切换 - 关键步骤

activateProgramPage(next);

这是虚拟地址空间切换的核心，通过 ProgramManager::activateProgramPage 方法实现：

```
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);
}
```

- CR3 寄存器更新

最关键的操作是调用 asm_update_cr3：

```
asm_update_cr3:
    push eax
    mov eax, dword[esp+8]
    mov cr3, eax            ; 将新的页目录表物理地址加载到 CR3 寄存器
    pop eax
    ret
```

- 上下文切换

通过 asm_switch_thread 完成栈和寄存器的切换。

```
asm_switch_thread(cur, next);
```

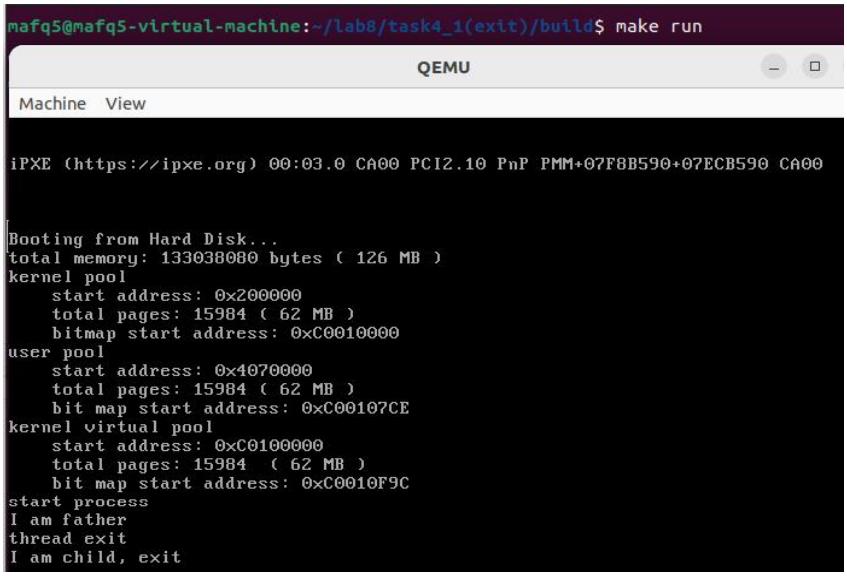
---------- 实验任务 4 ----------

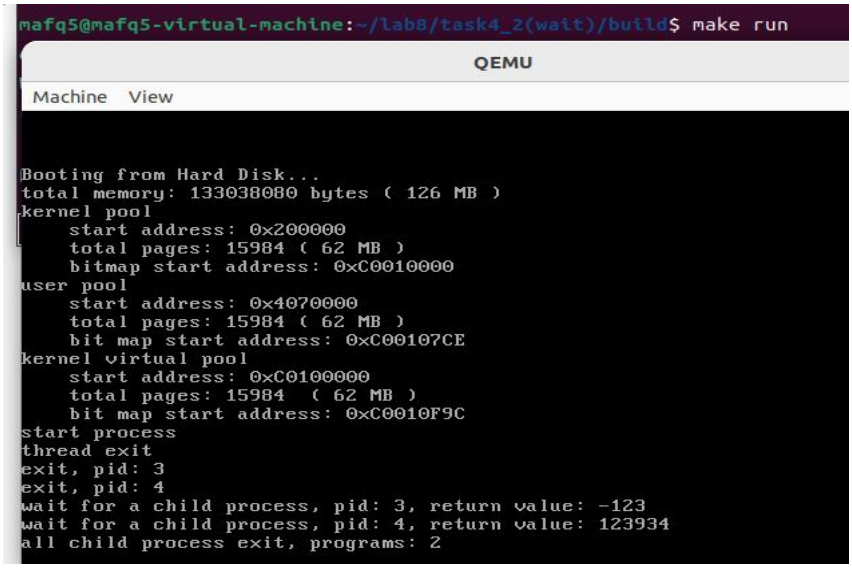
- 任务要求：wait & exit 的实现

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

- 实验步骤和实验结果展示:

1. 复现 wait 函数和 exit 函数





2. 请结合代码逻辑和具体的实例来分析 exit 的执行过程。

exit 系统调用的执行流程

1. 系统调用触发

在 setup.cpp 中，子进程执行到：

```
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");
                // 隐式调用 exit(0) - 在 load_process 中设置的返回地址
            }
        }
    }
```

2. 用户态到内核态转换

```
    // syscall.cpp
    void exit(int ret) {
        asm_system_call(3, ret);   // 3 号系统调用，参数为返回值
    }
    通过 int 0x80 中断进入内核态，调用 syscall_exit(ret)：
    void syscall_exit(int ret) {
        programManager.exit(ret);
    }
```

3. 内核态 exit 处理 (ProgramManager::exit)

```
    第一步：标记进程状态
    void ProgramManager::exit(int ret)
    {
        // 关中断
        interruptManager.disableInterrupt();

        // 标记 PCB 状态为 DEAD 并保存返回值
        PCB *program = this->running;
        program->retValue = ret;
        program->status = ProgramStatus::DEAD;
    第二步：释放进程资源（如果是进程）
        if (program->pageDirectoryAddress)   // 是进程而非线程
        {
            // 释放用户物理页
            pageDir = (int *)program->pageDirectoryAddress;
            for (int i = 0; i < 768; ++i) {   // 遍历用户空间页目录项
                if (!(pageDir[i] & 0x1)) continue;

                page = (int *)(0xffc00000 + (i << 12));   // 页表虚拟地址

                // 释放页表中的所有物理页
                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);
                }
```

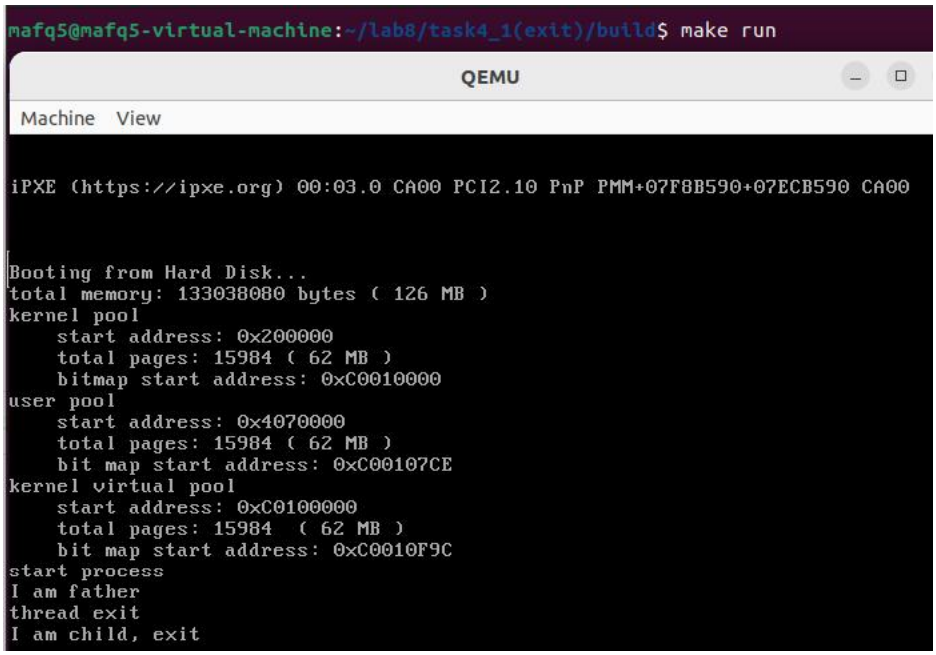
```
                // 释放页表本身
                paddr = memoryManager.vaddr2paddr((int)page);
                memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);
            }

            // 释放页目录表
            memoryManager.releasePages(AddressPoolType::KERNEL, (int)pageDir, 1);

            // 释放虚拟地址池 bitmap
            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:立即调度
        // 立即执行线程/进程调度
        schedule();
    }
```

4. 调度器处理死亡进程

在 ProgramManager::schedule() 中：

```
    void ProgramManager::schedule()
    {
        // ...
        if (running->status == ProgramStatus::RUNNING) {
            // 正常运行的进程加入就绪队列
            running->status = ProgramStatus::READY;
            running->ticks = running->priority * 10;
            readyPrograms.push_back(&(running->tagInGeneralList));
        }
        else if (running->status == ProgramStatus::DEAD) {
            // 死亡进程直接释放 PCB
            releasePCB(running);
        }

        // 切换到下一个就绪进程
        ListItem *item = readyPrograms.front();
        PCB *next = ListItem2PCB(item, tagInGeneralList);
        // ...
    }
```

如图：



first_thread 启动，输出 "start process"

first_process (父进程) 执行 fork()，输出 "I am father"

子进程 被创建并调度执行，输出 "I am child, exit"，然后调用 exit(0)

子进程 执行 exit 系统调用，释放所有资源后被销毁

second_thread 被调度执行，输出 "thread exit"，然后调用 exit(0)

second_thread 也被销毁

3. 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)

进程隐式调用 `exit` 的原因是：在创建进程时预设了栈中的返回地址，利用函数调用的返回机制，通过预设栈中返回地址实现进程的自动清理退出

- 在 `load_process` 函数中：

```
// 设置进程返回地址
int *userStack = (int *)interruptStack->esp;
userStack -= 3;
userStack[0] = (int)exit;      // 返回地址设为 exit 函数
userStack[1] = 0;                // exit 的参数 (返回值 0)
userStack[2] = 0;                // 栈对齐

interruptStack->esp = (int)userStack;
```

- 栈结构

```
高地址
+-----------------+
|        0        | ← 栈对齐
+-----------------+
```

```
|         0              | ← exit 参数
+------------------+
|    exit 函数地址      | ← 返回地址
+------------------+
|    进程代码执行       |← ESP
+------------------+
低地址
```

- 执行机制

进程代码（如 `first_process`）正常执行完毕

函数执行 `ret` 指令返回

CPU 从栈顶弹出返回地址 - 正是预设的 `exit` 函数地址

程序自动跳转到 `exit(0)` 执行

触发 3 号系统调用完成进程退出

4. 请结合代码逻辑和具体的实例来分析 wait 的执行过程。

- wait 系统调用执行流程

1. 用户态调用

```
// setup.cpp - first_process()
while ((pid = wait(&retval)) != -1) {
    printf("wait for a child process, pid: %d, return value: %d\n", pid, retval);
}
```

2. 系统调用接口

```
// syscall.cpp
int wait(int *retval) {
    return asm_system_call(4, (int)retval);   // 4 号系统调用
}

int syscall_wait(int *retval) {
    return programManager.wait(retval);
}
```

3. 内核态 wait 实现

```
// program.cpp
int ProgramManager::wait(int *retval)
{
    while (true) {
        // 关中断
        interruptManager.disableInterrupt();

        // 遍历所有进程查找子进程
        item = this->allPrograms.head.next;
```

```
            flag = true;
            while (item) {
                child = ListItem2PCB(item, tagInAllList);
                if (child->parentPid == this->running->pid) {
                    flag = false;
                    if (child->status == ProgramStatus::DEAD) {
                        break;   // 找到死亡子进程
                    }
                }
                item = item->next;
            }

            if (item) {   // 找到死亡子进程
                if (retval) *retval = child->retValue;
                int pid = child->pid;
                releasePCB(child);   // 回收子进程 PCB
                return pid;
            } else {
                if (flag) return -1;   // 无子进程
                else schedule();         // 有子进程但未死亡，调度其他进程
            }
        }
    }
```

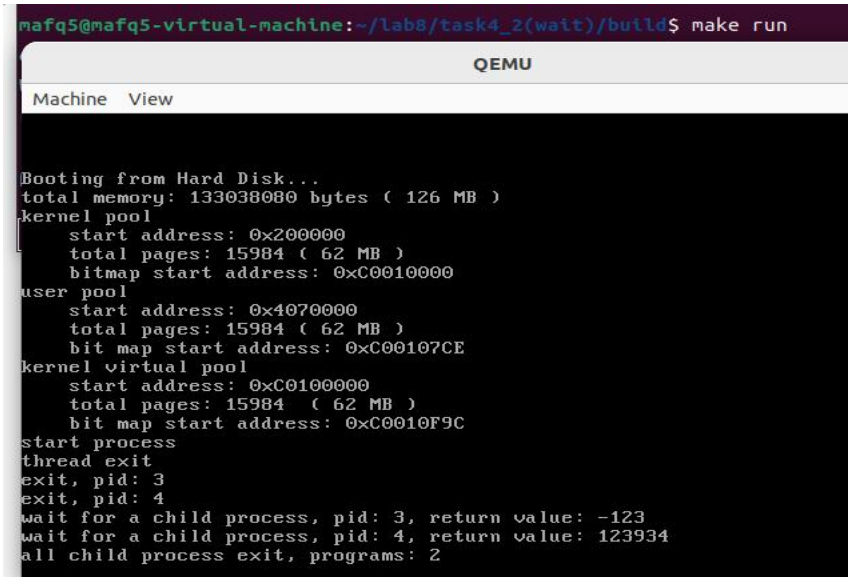
- 关键机制：

子进程状态检查：通过 child->parentPid == this->running->pid 识别子进程

阻塞等待：如果子进程存在但未死亡，调用 schedule() 让出 CPU

资源回收：找到死亡子进程后，通过 releasePCB(child) 回收 PCB

返回值传递：通过 child->retValue 获取子进程的退出码



- 实例分析如图，wait 的具体执行过程：

第一次 wait 调用：

父进程查找子进程，发现 pid=4 已死亡 (DEAD 状态)

获取返回值 -123，回收 PCB

返回 pid=4, retval=-123

第二次 wait 调用：

父进程查找子进程，发现 pid=5 已死亡

获取返回值 123934，回收 PCB

返回 pid=5, retval=123934

第三次 wait 调用：

遍历所有进程，未找到任何子进程

flag = true，返回 -1 表示无更多子进程

5. 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，实现回收孤儿进程和僵尸进程的有效方法。

孤儿进程处理机制

1. init 进程作为收养者

```
// setup.cpp - 创建 init 进程(PID=1)
void init_process()
{
    int retval;
    int pid;
    while (true)
    {
        while ((pid = wait(&retval)) != -1)
        {
            printf("recycle orphan process, pid: %d, return value: %d\n", pid, retval);
        }
    }
}
```

2. 进程退出时的孤儿收养

```
// program.cpp - exit 函数中的孤儿处理
void ProgramManager::exit(int ret)
{
    // ...
```

```
        ListItem *item = this->allPrograms.head.next;
        PCB *pcb;
        bool parentFound = false;
        bool parentDead = false;

        while (item)
        {
            pcb = ListItem2PCB(item, tagInAllList);

            // 1. 将当前进程的所有子进程交给 init 进程收养
            if (pcb->parentPid == program->pid)
            {
                pcb->parentPid = 1;   // 改为 init 进程的子进程
                printf("Process %d is now orphan, adopted by init process\n", pcb->pid);
            }

            // 2. 检查当前进程是否已经是孤儿
            if (program->parentPid != 0 && pcb->pid == program->parentPid)
            {
                parentFound = true;
                if (pcb->status == ProgramStatus::DEAD)
                {
                    parentDead = true;
                }
            }

            item = item->next;
        }

        // 如果当前进程是孤儿，也交给 init 收养
        if (program->parentPid != 0 && (!parentFound || parentDead))
        {
            program->parentPid = 1;
            printf("Process %d is orphan, changing parent to init process\n", program->pid);
        }
        // ...
    }
```

僵尸进程回收机制

1. 调度器中的僵尸进程处理

```
    // program.cpp - schedule 函数
    void ProgramManager::schedule()
    {
        // ...
```

```
        if (running->status == ProgramStatus::DEAD)
        {
            // 只回收线程，进程留给父进程回收
            if(!running->pageDirectoryAddress) {
                releasePCB(running);   // 立即回收线程
            }
            // 进程变为僵尸状态，等待父进程回收
        }
        // ...
    }
```

2. wait 系统调用回收僵尸子进程

```
    int ProgramManager::wait(int *retval)
    {
        PCB *child;
        ListItem *item;
        bool flag;
        int lastPid = -1;

        while (true)
        {
            interruptManager.disableInterrupt();

            item = this->allPrograms.head.next;
            flag = true;   // 假设没有子进程

            // 扫描所有进程，回收僵尸子进程
            while (item)
            {
                child = ListItem2PCB(item, tagInAllList);
                item = item->next;   // 提前获取下一个，防止当前被释放

                if (child->parentPid == this->running->pid)
                {
                    flag = false;   // 有子进程
                    if (child->status == ProgramStatus::DEAD)
                    {
                        // 回收僵尸子进程
                        if (retval)
                        {
                            *retval = child->retValue;
                        }
                        lastPid = child->pid;
                        releasePCB(child);   // 释放 PCB 和所有资源
                        printf("recycle zombie process: %d\n", lastPid);
```

```
                    }
                }
            }

            if (flag)
            {
                // 没有子进程，返回-1
                interruptManager.setInterruptStatus(interrupt);
                return lastPid != -1 ? lastPid : -1;
            }
            else if (lastPid == -1)
            {
                // 有子进程但都还活着，调度等待
                interruptManager.setInterruptStatus(interrupt);
                schedule();
            }
            else
            {
                // 回收了至少一个子进程
                interruptManager.setInterruptStatus(interrupt);
                return lastPid;
            }
        }
    }
    init 进程(PID=1) - 永久运行，专门回收孤儿进程
    ├── first_process
    ├── second_process
    └── 所有孤儿进程 (动态收养)
```

关键设计

- init 进程的特殊作用

  PID=1，系统启动时创建

  无限循环调用 wait()回收孤儿进程

  永不退出，确保孤儿进程能被及时回收

- 双重收养机制

  进程退出时主动将子进程交给 init

  检测到自己是孤儿时主动投靠 init

- 僵尸进程的延迟回收

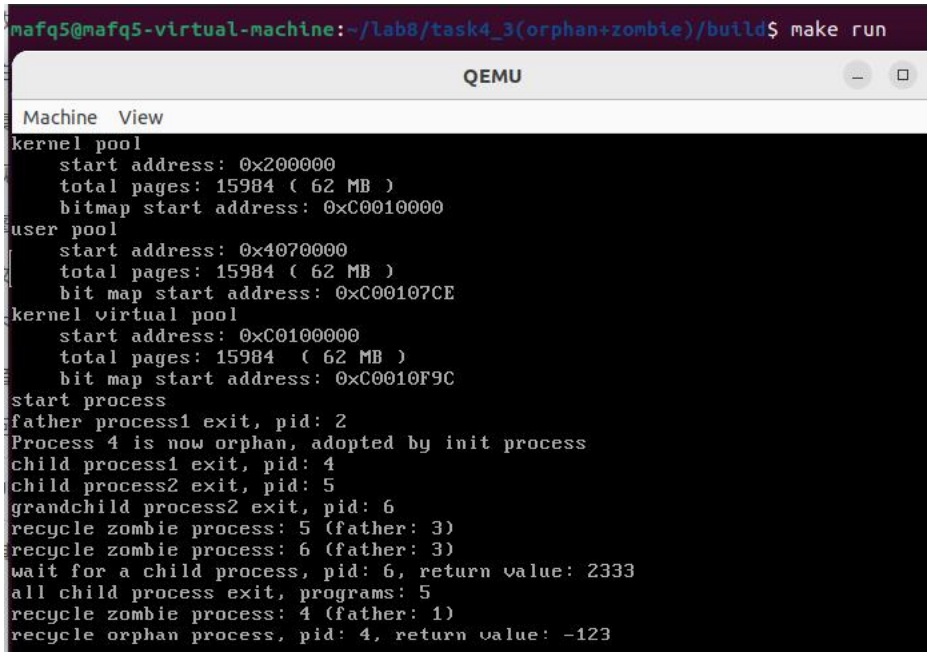
  进程死亡后不立即释放 PCB

  保留退出状态供父进程获取

通过 wait()系统调用完成最终回收

- 资源回收的完整性

```
void ProgramManager::releasePCB(PCB *program)
{
    int index = ((int)program - (int)PCB_SET) / PCB_SIZE;
    PCB_SET_STATUS[index] = false;
    this->allPrograms.erase(&(program->tagInAllList));
}
```



如图：

- 孤儿进程处理

父进程(pid=2)先退出

子进程(pid=4)被 init 进程收养

最终由 init 进程回收孤儿进程

- 僵尸进程处理

子进程 5、6 变成僵尸进程

父进程 3 通过 wait()回收僵尸子进程

正确获取退出状态(2333)

- 系统资源管理

所有进程最终被正确回收

无进程泄漏，系统稳定运行

Section 5 实验总结与心得体会

（1）在使用 gdb 进行 debug 的时候，指令‘s’固然可以单步执行，但是在遇到循环的时候，特别是几百上千次的循环，使用 s 会一遍一遍地执行循环，浪
费大量的时间，这时候我们可以采用指令‘next’，它是一种“进阶版”的单步执行，可以跳过循环；

（2）用 gdb 进行 debug 的过程中，如果我们光看 debug 的窗口，我们很容易会不知道程序运行到什么程度了，这时可以观看 qemu 的输出来了解进度。

比如我们看到 qemu 已经输出父进程函数里的信息了，我们就可以确认此时已经
进入了子进程中。只有这样，我们才能知道现在看的寄存器存储的是父进程的信息还是子进程的信息。

（3）

```
启动 GDB
  gdb <executable>
运行程序
  run
  start
停止程序
  stop
  kill
设置断点
  break <location>
  b <location>
  tbreak <location>
  cbreak <location>
查看和检查程序状态
  list
  list <filename>
  info locals
  info args
  info registers
  x <expression>
  print <expression>
单步执行
  step
  next
  stepi
```

```
    nexti
跳过函数
    finish
    return
继续执行
    continue
    c
查看调用栈
    backtrace 或 bt
    frame <framenumber>
改变程序执行
    call <function>
    set var <variable> = <value>
管理断点
    disable <breakpoint>
    enable <breakpoint>
    delete <breakpoint>
    info breakpoints
退出 GDB
    quit
    q
```

Section 6 附录：参考资料清单

1. 部分代码 debug 与指令除了参考实验指导书，还参考了大语言模型
2. 参考文章，博客部分如下：

操作系统实验——用户态与内核态资源-CSDN 文库_资源-CSDN 文库

CPU 用户态切换到内核态的触发机制详解_进入内核态的操作-CSDN 博客