

## 实验题目：

从内核态到用户态

## 实验要求：

- DDL：2021年06月17号 23:59
- 提交的内容：将**3个assignment的代码和实验报告**放到**压缩包**中，命名为“lab8-姓名-学号”，并交到课程网站上[\[http://course.dds-sysu.tech/course/3/homework\]](http://course.dds-sysu.tech/course/3/homework)
- 材料的代码放置在 `src` 目录下。

1. 实验不限语言，C/C++/Rust都可以。
2. 实验不限平台，Windows、Linux和MacOS等都可以。
3. 实验不限CPU，ARM/Intel/Risc-V都可以。

## 实验内容：

### Assignment 1 系统调用

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

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

### Assignment 2 Fork的奥秘

实现fork函数，并回答以下问题。

- 请根据代码逻辑和执行结果来分析fork实现的基本思路。
- 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从 `fork` 返回，根据gdb来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完 `ProgramManager::fork` 后的返回过程的异同。
- 请根据代码逻辑和gdb来解释fork是如何保证子进程的 `fork` 返回值是0，而父进程的 `fork` 返回值是子进程的pid。

### Assignment 3 哼哈二将 wait & exit

实现wait函数和exit函数，并回答以下问题。

- 请结合代码逻辑和具体的实例来分析exit的执行过程。
- 请分析进程退出后能够隐式地调用exit和此时的exit返回值是0的原因。
- 请结合代码逻辑和具体的实例来分析wait的执行过程。
- 如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为 `DEAD` 开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法。

## 实验步骤：

### assignment1系统调用的实现：

系统调用的过程简单来说就是调用软中断、中断处理和中断返回。

#### (1) 首先声明一个系统调用的入口函数：

```
extern "C" int asm_system_call(int index, int first = 0, int second = 0, int third = 0, int forth = 0, int fifth = 0);
```

用汇编实现：

```
asm_system_call:
    push ebp
    mov ebp, esp

    push ebx
    push ecx
    push edx
    push esi
    push edi
    push ds
    push es
    push fs
    push gs

    mov eax, [ebp + 2 * 4]
    mov ebx, [ebp + 3 * 4]
    mov ecx, [ebp + 4 * 4]
    mov edx, [ebp + 5 * 4]
    mov esi, [ebp + 6 * 4]
    mov edi, [ebp + 7 * 4]

    int 0x80

    pop gs
    pop fs
    pop es
    pop ds
    pop edi
    pop esi
    pop edx
    pop ecx
    pop ebx
    pop ebp

    ret
```

用 `int 0x80` 调用中断实现系统调用。

## (2) 创建一个系统管理器，用于管理系统调用表。

声明在 `syscall.h` 中。

```
#ifndef SYSCALL_H
#define SYSCALL_H

#include "os_constant.h"

class SystemService
{
public:
    SystemService();
    void initialize();
    // 设置系统调用，index=系统调用号，function=处理第index个系统调用函数的地址
    bool setSystemCall(int index, int function);
};

// 第0个系统调用
int syscall_0(int first, int second, int third, int forth, int fifth);

#endif
```

实现：

```
#include "syscall.h"
#include "interrupt.h"
#include "stdlib.h"
#include "asm_utils.h"
#include "os_modules.h"

int system_call_table[MAX_SYSTEM_CALL];

SystemService::SystemService() {
    initialize();
}

void SystemService::initialize()
{
    memset((char *)system_call_table, 0, sizeof(int) * MAX_SYSTEM_CALL);
    // 代码段选择子默认是DPL=0的平坦模式代码段选择子，DPL=3，否则用户态程序无法使用该中断描述符
    interruptManager.setInterruptDescriptor(0x80,
    (uint32)asm_system_call_handler, 3);
}

bool systemService::setSystemCall(int index, int function)
{
    system_call_table[index] = function;
    return true;
}
```

其中，`system_call_table` 为系统调用表，每一个元素存放系统调用函数的地址。系统管理器先对系统调用表进行初始化，函数 `setSystemCall` 实现将函数 `function` 放进系统调用号为 `index` 的系统调用表。

(3) 在 `setup.cpp` 中写第一个系统调用函数，测试系统调用功能的实现。

```
int syscall_0(int first, int second, int third, int forth, int fifth)
{
    printf("system call 0: %d, %d, %d, %d, %d\n",
           first, second, third, forth, fifth);
    return first + second + third + forth + fifth;
}
```

然后在 `setup.cpp` 中的 `setup_kernel` 中，将第0个系统调用函数放入第0号系统调用表。然后分别调用5次，每次传进数量不同的参数。

```
extern "C" void setup_kernel()
{
    ...
    // 初始化系统调用
    systemService.initialize();
    systemService.setSystemCall(0, (int)syscall_0);

    int ret;

    ret = asm_system_call(0);
    printf("return value: %d\n", ret);

    ret = asm_system_call(0, 123);
    printf("return value: %d\n", ret);

    ret = asm_system_call(0, 123, 324);
    printf("return value: %d\n", ret);

    ret = asm_system_call(0, 123, 324, 9248);
    printf("return value: %d\n", ret);

    ret = asm_system_call(0, 123, 324, 9248, 7);
    printf("return value: %d\n", ret);

    ret = asm_system_call(0, 123, 324, 9248, 7, 123);
    printf("return value: %d\n", ret);

    ...
}
```

编译运行：

```
QEMU
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0x10000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
system call 0: 0, 0, 0, 0, 0
return value: 0
system call 0: 123, 0, 0, 0, 0
return value: 123
system call 0: 123, 324, 0, 0, 0
return value: 447
system call 0: 123, 324, 9248, 0, 0
return value: 9695
system call 0: 123, 324, 9248, 7, 0
return value: 9702
system call 0: 123, 324, 9248, 7, 123
return value: 9825
```

可以看到，到这里已经实现了系统调用。五次调用，由于传进去的参数不一样，最终的输出结果也不一样。

#### (4) 在进程中进行系统调用，实现内核态和用户态的分离。


将内核空间扩展到3GB，其他对应的地址也提升到3GB。如下图：

```
QEMU
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
system call 0: 0, 0, 0, 0, 0
return value: 0
system call 0: 123, 0, 0, 0, 0
return value: 123
system call 0: 123, 324, 0, 0, 0
return value: 447
system call 0: 123, 324, 9248, 0, 0
return value: 9695
system call 0: 123, 324, 9248, 7, 0
return value: 9702
system call 0: 123, 324, 9248, 7, 123
return value: 9825
```

#### (5) 初始化TSS和用户段描述符。

每个任务有一个任务状态段TSS，用于保存任务的有关信息，在任务内变换特权级和任务切换时，要用到这些信息。TSS是内存中的一个结构体，它在内存中的结构如下：

31	15	0	
I/O Map Base Address		Reserved	T 100
Reserved		LDT Segment Selector	96
Reserved		GS	92
Reserved		FS	88
Reserved		DS	84
Reserved		SS	80
Reserved		CS	76
Reserved		ES	72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved		SS2	24
ESP2			20
Reserved		SS1	16
ESP1			12
Reserved		SS0	8
ESP0			4
Reserved		Previous Task Link	0

 Reserved bits. Set to 0.

抽象成结构体如下：

```

#ifndef TSS_H
#define TSS_H

struct TSS
{
public:
    int backlink;
    int esp0;
    int ss0;
    int esp1;
    int ss1;
    int esp2;
    int ss2;
    int cr3;
    int eip;
    int eflags;
    int eax;
    int ecx;
    int edx;
    int ebx;
    int esp;
    int ebp;
    int esi;
    int edi;
    int es;

```

```

int cs;
int ss;
int ds;
int fs;
int gs;
int ldt;
int trace;
int ioMap;
};
#endif

```

当从低特权级向高特权级转移时，CPU首先会在TSS中找到高特权级栈的段选择子和栈指针，然后送入SS，ESP。此时，栈发生变化，此时的栈已经变成了TSS保存的高特权级的栈。接着，中断发生前的SS、ESP、EFLAGS、CS、EIP被依次压入了高特权级栈。

CPU是在特权级转移的时候自动加载TSS的内容的，那CPU是如何知道TSS在哪的呢？CPU通过tr寄存器知道TSS的相关信息，比如TSS段的基址，大小和属性。因此，CPU通过 tr 寄存器来确定 TSS 的位置的。

可以通过 `ltr` 指令跟上TSS段描述符的选择子来加载TSS段。该指令是特权指令，只能在特权级为0的情况下使用。

TSS的初始化如下：

```

void ProgramManager::initializeTSS()
{
    int size = sizeof(TSS);
    int address = (int)&tss;

    memset((char *)address, 0, size);
    tss.ss0 = STACK_SELECTOR; // 内核态堆栈段选择子

    int low, high, limit;

    limit = size - 1;
    low = (address << 16) | (limit & 0xff);
    // DPL = 0
    high = (address & 0xff000000) | ((address & 0x00ff0000) >> 16) | ((limit & 0xff00) << 16) | 0x00008900;

    int selector = asm_add_global_descriptor(low, high);
    // RPL = 0
    asm_ltr(selector << 3);
    tss.ioMap = address + size;
}

```

在这里只对 `TSS::ss0` 进行复制，`TSS::esp0` 会在进程切换时更新。

其中，`STACK_SELECTOR` 是特权级0下的栈段选择子，我们在bootloader中放入了SS的选择子。

```
#define STACK_SELECTOR 0x10
```

在 `ProgramManager` 中加入存储3个代码段、数据段和栈段描述符的变量。

```

class ProgramManager
{
public:
    List allPrograms;           // 所有状态的线程/进程的队列
    List readyPrograms;         // 处于ready(就绪态)的线程/进程的队列
    PCB *running;               // 当前执行的线程
    int USER_CODE_SELECTOR;     // 用户代码段选择子
    int USER_DATA_SELECTOR;     // 用户数据段选择子
    int USER_STACK_SELECTOR;    // 用户栈段选择子
    ...
}

```

向GDT中新增用户代码段描述符，数据段描述符和栈段描述符。加入的3个描述符的DPL为3.

```

#define USER_CODE_LOW  0x0000ffff
#define USER_CODE_HIGH 0x00cfff800

#define USER_DATA_LOW  0x0000ffff
#define USER_DATA_HIGH 0x00cfff200

#define USER_STACK_LOW  0x00000000
#define USER_STACK_HIGH 0x0040f600

```

将这个几个段描述符送入GDT:

```

; int asm_add_global_descriptor(int low, int high);
asm_add_global_descriptor:
    push ebp
    mov ebp, esp

    push ebx
    push esi

    sgdt [ASM_GDTR]
    mov ebx, [ASM_GDTR + 2] ; GDT地址
    xor esi, esi
    mov si, word[ASM_GDTR] ; GDT界限
    add esi, 1

    mov eax, [ebp + 2 * 4] ; low
    mov dword [ebx + esi], eax
    mov eax, [ebp + 3 * 4] ; high
    mov dword [ebx + esi + 4], eax

    mov eax, esi
    shr eax, 3

    add word[ASM_GDTR], 8
    lgdt [ASM_GDTR]

    pop esi
    pop ebx
    pop ebp

    ret

```



初始化TSS、特权级3下的平坦模式代码段和数据段描述符：

```
void ProgramManager::initialize()
{
    allPrograms.initialize();
    readyPrograms.initialize();
    running = nullptr;

    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        PCB_SET_STATUS[i] = false;
    }

    // 初始化用户代码段、数据段和栈段
    int selector;

    selector = asm_add_global_descriptor(USER_CODE_LOW, USER_CODE_HIGH);
    USER_CODE_SELECTOR = (selector << 3) | 0x3;

    selector = asm_add_global_descriptor(USER_DATA_LOW, USER_DATA_HIGH);
    USER_DATA_SELECTOR = (selector << 3) | 0x3;

    selector = asm_add_global_descriptor(USER_STACK_LOW, USER_STACK_HIGH);
    USER_STACK_SELECTOR = (selector << 3) | 0x3;

    initializeTSS();
}
```

## (6) 进程的创建

进程的创建分为3步。

- 创建进程的PCB。
- 初始化进程的页目录表。
- 初始化进程的虚拟地址池。

进程和线程都使用PCB描述，但是进程比线程多了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。于是在进程PCB中加入：

```
int pageDirectoryAddress;           // 页目录表地址
AddressPool userVirtual;            // 用户程序虚拟地址池
```

为进程创建虚拟地址池：

```
bool ProgramManager::createUserVirtualPool(PCB *process)
{
    int sourcesCount = (0xc0000000 - USER_VADDR_START) / PAGE_SIZE;
    int bitmapLength = ceil(sourcesCount, 8);

    // 计算位图所占的页数
    int pageCount = ceil(bitmapLength, PAGE_SIZE);

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

    if (!start)
```

```

{
    return false;
}

memset((char *)start, 0, PAGE_SIZE * pageCount);
(process->userVirtual).initialize((char *)start, bitmapLength,
USER_VADDR_START);

return true;
}

```

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,

```
#define USER_VADDR_START 0x8048000
```

通常情况下，CPU不允许我们从高特权级向低特权级转移。实现高特权级向低特权级转移的唯一办法就是通过中断返回。我们可以通过 `iret` 指令强制将低特权级下的段选择子和栈送入段寄存器，从而实现了从高特权级别向低特权级转移，然后跳转到用户进程所在地址处执行。因此，在启动进程之前，需要将进程需要的段选择子等信息放入栈中。

因此，为了方便保存信息，在 `include/process.h` 定义一个类 `ProgramStartStack` 来表示启动进程之前栈放入的内容，并在PCB的顶部预留出 `ProcessStartStack` 的空间。

```

#ifndef PROCESS_H
#define PROCESS_H
struct ProcessStartStack
{
    int edi;
    int esi;
    int ebp;
    int esp_dummy;
    int ebx;
    int edx;
    int ecx;
    int eax;

    int gs;
    int fs;
    int es;
    int ds;

    int eip;
    int cs;
    int eflags;
    int esp;
    int ss;
};

#endif

```

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

```

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

```

```

{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVirtual.allocate(count);
    } else if (type == AddressPoolType::USER){
        start = programManager.running->userVirtual.allocate(count);
    }

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

void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int
vaddr, const int count)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelVirtual.release(vaddr, count);
    }
    else if (type == AddressPoolType::UESR)
    {
        programManager.running->userVirtual.release(vaddr, count);
    }
}

```

## (7) 进程的调度

只需在线程调度的基础上加上

- 切换页目录表。
- 更新TSS中的特权级0的栈。

切换页目录表:

```

void ProgramManager::schedule()
{
    ...

    activateProgramPage(next);

    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}

```

更新TSS中的特权级0的栈:

```

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);
}

```

## (8) 创建第一个进程

该进程会调用之前写好的0号系统调用。在第一个线程中执行：

```

void first_thread(void *arg)
{
    printf("start process\n");
    programManager.executeProcess((const char *)first_process, 1);
    programManager.executeProcess((const char *)first_process, 1);
    programManager.executeProcess((const char *)first_process, 1);
    asm_halt();
}

```

运行结果：

```

QEMU

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 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
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0

```

## (9) 第二个系统调用, “Hello World!”

为了能在first\_process()中打出 `Hello world!`，我写了第二个系统调用。

```

void syscall_1(){
    printf("Hello world!\n");
    return;
}

```

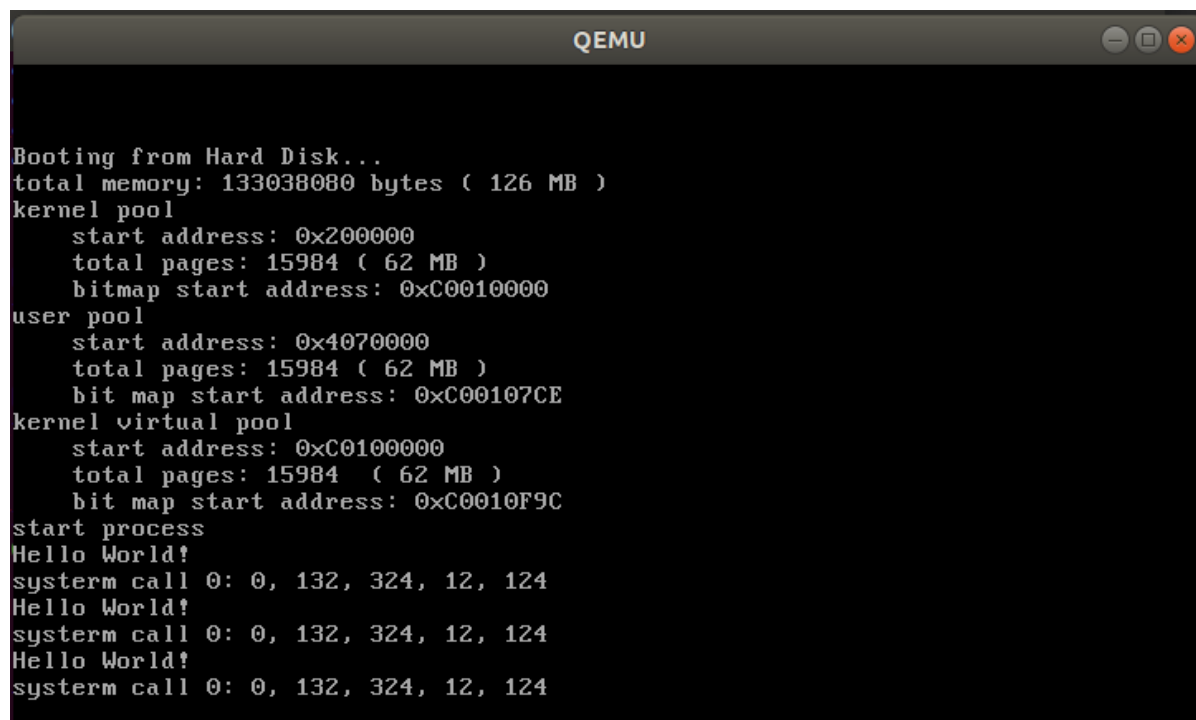
然后，将该系统调用函数设置为1号系统调用：

```
systemService.setSystemCall(1, (int)syscall_1);
```

然后在进程中使用1号系统调用打印 Hello world!：

```
void first_process()
{
    //printf("Hello World!\n");
    asm_system_call(1);
    asm_system_call(0, 0, 132, 324, 12, 124);
    asm_halt();
}
```

结果：

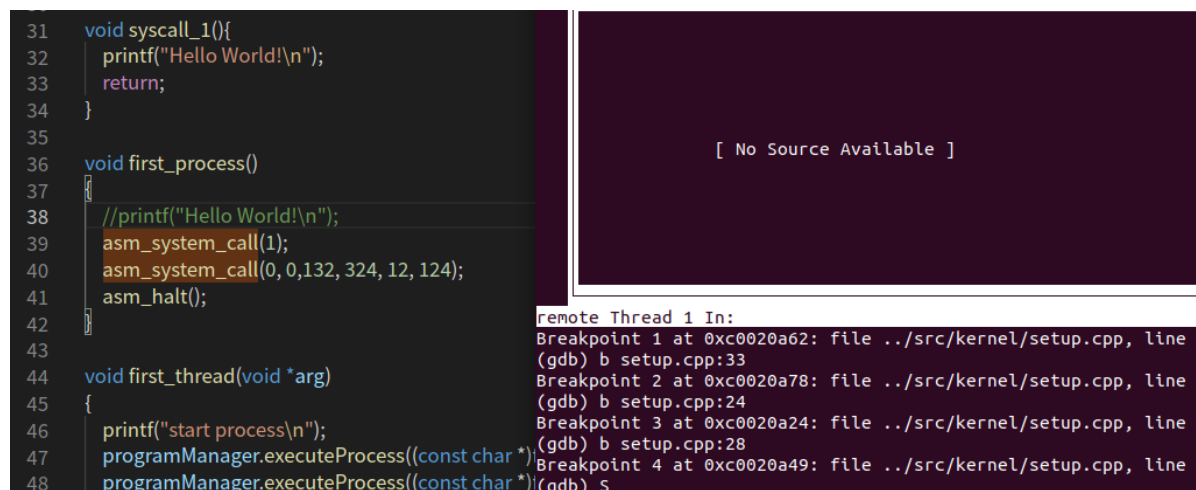
A screenshot of a QEMU terminal window. The title bar says 'QEMU'. The terminal output shows the boot process: 'Booting from Hard Disk...', memory allocation for kernel and user pools, and the start of a process. The process prints 'Hello World!' and then makes a system call with arguments 0, 132, 324, 12, 124. This sequence is repeated three times.

```
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
Hello World!
system call 0: 0, 132, 324, 12, 124
Hello World!
system call 0: 0, 132, 324, 12, 124
Hello World!
system call 0: 0, 132, 324, 12, 124
```

可以看到，在进程中，我们通过系统调用输出了“Hello World! ”。

## (10) 用gdb分析执行系统调用后的栈的变化情况。

如下图示，我们在setup.cpp:39和setup.cpp:40处和系统调用函数内部设置了断点。

A screenshot of the GDB interface. On the left, the source code for 'first\_process()' and 'first\_thread()' is shown. Line 39 and 40 in 'first\_process()' are highlighted. On the right, a panel shows '[ No Source Available ]'. At the bottom, a list of breakpoints is displayed, including breakpoints in 'setup.cpp' and 'syscall\_1'.

```
31 void syscall_1(){
32     printf("Hello World!\n");
33     return;
34 }
35
36 void first_process()
37 {
38     //printf("Hello World!\n");
39     asm_system_call(1);
40     asm_system_call(0, 0, 132, 324, 12, 124);
41     asm_halt();
42 }
43
44 void first_thread(void *arg)
45 {
46     printf("start process\n");
47     programManager.executeProcess((const char *)
48     programManager.executeProcess((const char *)
```

remote Thread 1 In:

- Breakpoint 1 at 0xc0020a62: file ../src/kernel/setup.cpp, line (gdb) b setup.cpp:33
- Breakpoint 2 at 0xc0020a78: file ../src/kernel/setup.cpp, line (gdb) b setup.cpp:24
- Breakpoint 3 at 0xc0020a24: file ../src/kernel/setup.cpp, line (gdb) b setup.cpp:28
- Breakpoint 4 at 0xc0020a49: file ../src/kernel/setup.cpp, line (gdb) s

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/setup.cpp
34     }
35
36     void first_process()
37     {
38         //printf("Hello World!\n");
39         asm system call(1);
B+ 40         asm system call(0, 0, 132, 324, 12, 124);
B+> 41         asm_halt();
42     }
43
44     void first_thread(void *arg)
45     {
46         printf("start process\n");

remote Thread 1 In: first_process L40 PC: 0xc0020a98
(gdb) bt
#0  first_process () at ../src/kernel/setup.cpp:39
Backtrace stopped: Cannot access memory at address 0x8049000
(gdb) c
Continuing.

Breakpoint 5, first_process () at ../src/kernel/setup.cpp:40
(gdb)
```

当执行到第40行时，执行 `gdb bt` 查看堆栈情况。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
125     pop ds
126     mov eax, [ASM_TEMP]
127
128     iret
129     asm system call:
> 130     push ebp
131     mov ebp, esp
132
133     push ebx
134     push ecx
135     push edx
136     push esi
137     push edi

remote Thread 1 In: asm system call L130 PC: 0xc00226f3
(gdb) bt
#0  asm_system_call () at ../src/utils/asm_utils.asm:130
#1  0xc0020ab2 in first_process () at ../src/kernel/setup.cpp:40
Backtrace stopped: Cannot access memory at address 0x8049000
#0  asm_system_call () at ../src/utils/asm_utils.asm:130
#1  0xc0020ab2 in first_process () at ../src/kernel/setup.cpp:40
Backtrace stopped: Cannot access memory at address 0x8049000
(gdb)
```

可以看到使用0号系统调用 (`setup.cpp:40`) 时，栈的信息是 `first_process()` --> `asm_system_call()`。

继续执行时，可以看到程序在 `asm_system_call()` 的 `int 0x80` 处进入了 `asm_system_call_handler`，栈的信息变为 `asm_system_call()` --> `asm_system_call_handler()`。  
`asm_system_call_handler` 将 `ds`，`es`，`fs`，`gs` 和其他数据寄存器压栈。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
83      pop ebp
84
85      ret
86      ; int asm_system_call_handler();
B+ 87      asm system_call_handler:
> 88      push ds
89      push es
90      push fs
91      push gs
92      pushad
93
94      push eax
95

remote Thread 1 In: asm system_call_handler L88 PC: 0xc00226b7
Continuing.

Breakpoint 6, asm_system_call_handler () at ../src/utils/asm_utils.asm:88
(gdb) bt
#0  asm_system_call_handler () at ../src/utils/asm_utils.asm:88
#1  0xc002270f in asm_system_call () at ../src/utils/asm_utils.asm:146
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) █
```

然后，运行到 `asm_system_call_handler` 中的 `call dword` 时，跳转到系统调用函数处执行。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
109     push esi
110     push edx
111     push ecx
112     push ebx
113
114     sti
> 115     call dword[system_call_table + eax * 4]
116     cli
117
118     add esp, 5 * 4
119
120     mov [ASM_TEMP], eax
121     popad

remote Thread 1 In: asm system_call_handler L115 PC: 0xc00226d6
asm_system_call_handler () at ../src/utils/asm_utils.asm:98
asm_system_call_handler () at ../src/utils/asm_utils.asm:108
asm_system_call_handler () at ../src/utils/asm_utils.asm:109
asm_system_call_handler () at ../src/utils/asm_utils.asm:110
asm_system_call_handler () at ../src/utils/asm_utils.asm:111
asm_system_call_handler () at ../src/utils/asm_utils.asm:112
asm_system_call_handler () at ../src/utils/asm_utils.asm:114
(gdb) ss
```

查看栈信息，可以看到栈信息为 `asm_system_call_handler()` --> `syscall_1()`。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/setup.cpp
30     }
31
32     void syscall_1(){
33
34         //printf(char*(c));
> 35         printf("Hello World!\n");
36         return;
37     }
38
39     void first_process()
40     {
41         //printf("Hello World!\n");
42
remote Thread 1 In: syscall_1                                L35   PC: 0xc0020a68
asm_system_call_handler () at ../src/utils/asm_utils.asm:114
warning: Source file is more recent than executable.
syscall_1 () at ../src/kernel/setup.cpp:35
(gdb) bt
#0  syscall_1 () at ../src/kernel/setup.cpp:35
#1  0xc00226dd in asm_system_call_handler () at ../src/utils/asm_utils.asm:115
#2  0x00000000 in ?? ()
(gdb)
```

执行完后，函数返回 `asm_system_call_handler`，然后 `asm_system_call_handler` 返回上一级的 `asm_system_call`，此时再查看栈情况，可以看到和第一次查看时的情况是一致的。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
143     mov esi, [ebp + 6 * 4]
144     mov edi, [ebp + 7 * 4]
145
146     int 0x80
147
> 148     pop edi
149     pop esi
150     pop edx
151     pop ecx
152     pop ebx
153     pop ebp
154
155     ret

remote Thread 1 In: asm system call                            L148  PC: 0xc002270f
asm_system_call_handler () at ../src/utils/asm_utils.asm:125
asm_system_call_handler () at ../src/utils/asm_utils.asm:126
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) bt
#0  asm_system_call () at ../src/utils/asm_utils.asm:148
#1  0xc0020a95 in first_process () at ../src/kernel/setup.cpp:43
Backtrace stopped: Cannot access memory at address 0x8049000
(gdb)
```

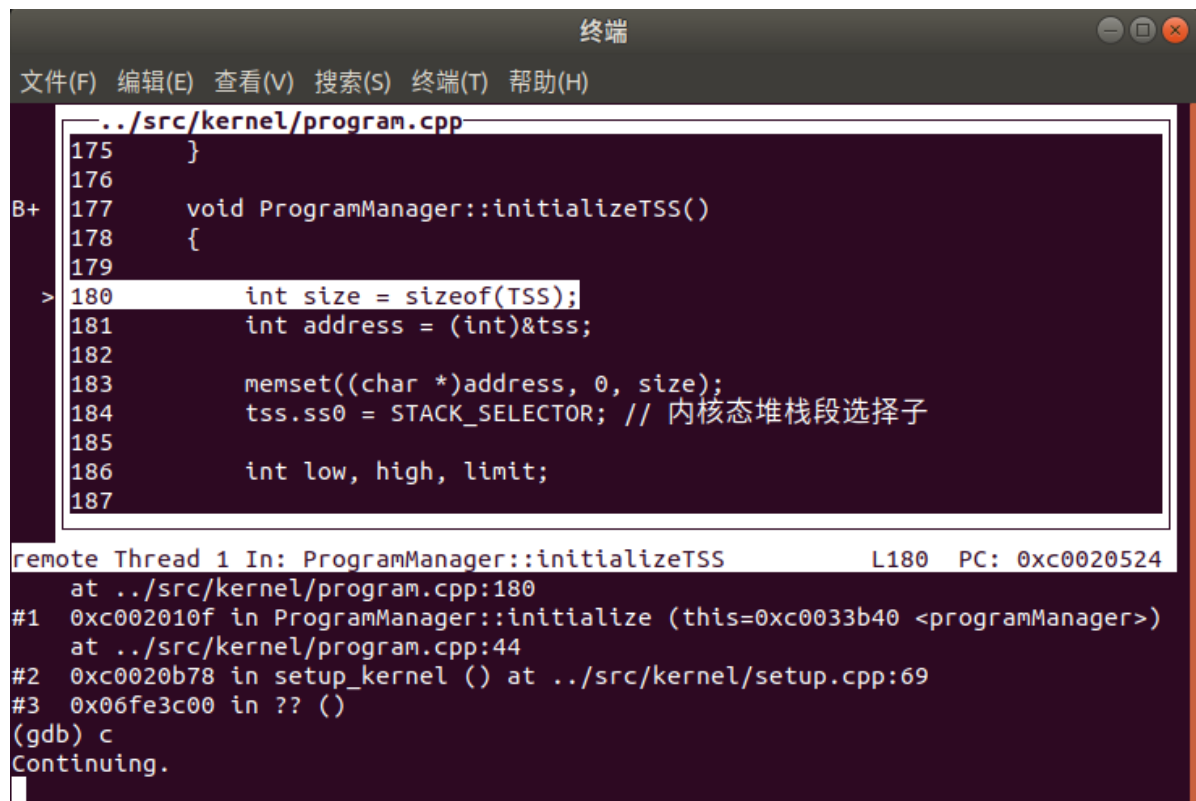
即 `first_process()` --> `asm_system_call()`。

因此系统调用的过程就是由进程发出系统调用，然后进入 `asm_system_call`，然后由 `asm_system_call` 进入 `asm_system_call_handler`，`asm_system_call_handler` 保存当前的栈信息，然后进入具体的系统调用函数。



### (11) 用gdb来说明TSS在系统调用执行过程中的作用。

首先可以看到，在 `setup_kernel()` 中的进程管理器 `programManager.initialize()` 中会首先对TSS进行初始化。



The screenshot shows a GDB terminal window with the title "终端". The menu bar includes "文件(F)", "编辑(E)", "查看(V)", "搜索(S)", "终端(T)", and "帮助(H)". The main window displays the source code for `../src/kernel/program.cpp`. The code is as follows:

```
175     }
176
177     void ProgramManager::initializeTSS()
178     {
179
180         int size = sizeof(TSS);
181         int address = (int)&tss;
182
183         memset((char *)address, 0, size);
184         tss.ss0 = STACK_SELECTOR; // 内核态堆栈段选择子
185
186         int low, high, limit;
187     }
```

The GDB prompt is at line 180. Below the code, the GDB output shows the current thread and program counter:

```
remote Thread 1 In: ProgramManager::initializeTSS      L180  PC: 0xc0020524
   at ../src/kernel/program.cpp:180
#1  0xc002010f in ProgramManager::initialize (this=0xc0033b40 <programManager>)
   at ../src/kernel/program.cpp:44
#2  0xc0020b78 in setup_kernel () at ../src/kernel/setup.cpp:69
#3  0x06fe3c00 in ?? ()
(gdb) c
Continuing.
```

初始化函数中将 `TSS::ss0` 初始化为特权级0下的栈段选择子 `STACK_SELECTOR`。`TSS::esp0` 会在进程切换时更新。

在进程切换时，会调用进程管理器的 `schedule()` 函数，然后在 `activateProgramPage` 中更新TSS中的特权级0的栈。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
29     ASM_GDTR dw 0
30         dd 0
31     ASM_TEMP dd 0
32     ; void asm_update_cr3(int address)
33     asm_update_cr3:
> 34     push eax
35         mov eax, dword[esp+8]
36         mov cr3, eax
37         pop eax
38         ret
39     asm_start_process:
40         ; jmp $
41         mov eax, dword[esp+4]

remote Thread 1 In: asm_update_cr3 L34 PC: 0xc0022656
#0  asm_update_cr3 () at ../src/utils/asm_utils.asm:34
#1  0xc00209b7 in ProgramManager::activateProgramPage (
    this=0xc0033b40 <programManager>, program=0xc0024700 <PCB_SET+4096>)
    at ../src/kernel/program.cpp:338
#2  0xc00203f1 in ProgramManager::schedule (this=0xc0033b40 <programManager>)
    at ../src/kernel/program.cpp:126
#3  0xc002117b in c_time_interrupt_handler () at ../src/kernel/interrupt.cpp:99
#4  0xc002277a in asm_time_interrupt_handler ()
    at ../src/utils/asm_utils.asm:241
#5  0x00000018 in ?? ()
#6  0xc002041c in ProgramManager::schedule (this=0x0)
    at ../src/kernel/program.cpp:131
---Type <return> to continue, or q <return> to quit---
```

之后CPU会在TSS中找到高特权级栈的段选择子和栈指针，从而完成进程的切换。

## assignment2 fork函数的奥秘

(1) 为实现fork做准备。

a、在PCB中加入父进程pid这个属性：

```
struct PCB
{
    ...
    int parentPid;           // 父进程pid
};
```

b、在 include/syscall.h 中加入fork系统调用和系统调用处理函数的定义：

```
#ifndef SYSCALL_H
#define SYSCALL_H

...

// 第2个系统调用，fork
int fork();
int syscall_fork();

#endif
```

c、在 src/kernel/setup.cpp 中设置这个系统调用：

```
systemService.setSystemCall(2, (int)syscall_fork);
```

d、实现fork系统调用：

```
int fork() {  
    return asm_system_call(2);  
}  
  
int syscall_fork() {  
    return programManager.fork();  
}
```

```
int ProgramManager::fork()  
{  
    bool status = interruptManager.getInterruptStatus();  
    interruptManager.disableInterrupt();  
  
    // 禁止内核线程调用  
    PCB *parent = this->running;  
    if (!parent->pageDirectoryAddress)  
    {  
        interruptManager.setInterruptStatus(status);  
        return -1;  
    }  
  
    // 创建子进程  
    int pid = executeProcess("", 0);  
    if (pid == -1)  
    {  
        interruptManager.setInterruptStatus(status);  
        return -1;  
    }  
  
    // 初始化子进程  
    PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);  
    bool flag = copyProcess(parent, child);  
  
    if (!flag)  
    {  
        child->status = ProgramStatus::DEAD;  
        interruptManager.setInterruptStatus(status);  
        return -1;  
    }  
  
    interruptManager.setInterruptStatus(status);  
    return pid;  
}
```

(2) 复制进程，实现函数 `bool ProgramManager::copyProcess(PCB *parent, PCB *child)`

a、实现父子进程从相同的返回点开始执行。把在中断的那一刻保存的寄存器的内容复制到子进程的0特权级栈中。

```

ProcessStartStack *childpss =
    (ProcessStartStack *)((int)child + PAGE_SIZE -
sizeof(ProcessStartStack));
ProcessStartStack *parentpss =
    (ProcessStartStack *)((int)parent + PAGE_SIZE -
sizeof(ProcessStartStack));
memcpy(parentpss, childpss, sizeof(ProcessStartStack));

```

b、设置子进程的返回值为0。

```
childpss->eax = 0;
```

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

```

// 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);

```

d、使用中转页进行资源的复制。具体做法是从内核中分配一页来作为数据复制的中转页，然后将父进程的页目录表复制到子进程中，然后复制页表和物理页的数据，最后归还中转页。

```

// 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;

```

### (3) 调用fork函数。

```

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

    if (pid == -1)
    {
        printf("can not fork\n");
    }
    else
    {
        if (pid)
        {
            printf("I am father, fork reutrnr: %d\n", pid);
        }
        else
        {
            printf("I am child, fork return: %d, my pid: %d\n", pid,
programManager.running->pid);
        }
    }
}

```

### (4) 编译运行:

```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD 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
I am father, fork reutrn: 2
I am child, fork return: 0, my pid: 2
```

可以看到父进程和子进程都运行了 `first_process` 中的代码，从输出语句可以看出他们的pid不一样。因此成功实现了fork函数。

#### (5) 跟踪进程的执行流程

下面进行子进程的跟踪。

我在 `setup.cpp:40` 处设置了一个断点，直接进入 `first_process` 的 `fork()` 函数，然后在gdb中设置调试子进程 `set follow-fork-mode child`，然后又在 `asm_utils.asm:39`，即 `asm_start_process` 入口处设置了一个断点。然后按 `c` 继续执行，直到子进程开始执行。

子进程从 `asm_start_process` 开始执行。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utls/asm_utils.asm
36      mov cr3, eax
37      pop eax
38      ret
B+ 39      asm_start_process:
40      ; jmp $
> 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

remote Thread 1 In: asm_start_process L41 PC: 0xc0022c30
eax      0xc0025e00      -1073586688
ecx      0x1            1
edx      0x219000      2199552
ebx      0x0            0
esp      0xc0026db4      0xc0026db4 <PCB_SET+12212>
ebp      0x0            0x0
esi      0x0            0
--Type <return> to continue, or q <return> to quit--S
```

然后执行一系列的pop把之前压栈的信息弹出。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utls/asm_utls.asm
47      pop ds;
48
> 49      iret
50
51      ; void asm_ltr(int tr)
52      asm_ltr:

0xc0022c39 <asm_start_process+9>      pop      fs
0xc0022c3b <asm_start_process+11>     pop      es
0xc0022c3c <asm_start_process+12>     pop      ds
> 0xc0022c3d <asm_start_process+13>   iret
0xc0022c3e <asm_ltr>                  ltr      WORD PTR [esp+0x4]
0xc0022c43 <asm_ltr+5>                ret

remote Thread 1 In: asm_start_process L49 PC: 0xc0022c3d
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx          0x0      0
esp          0xc0026dec 0xc0026dec <PCB_SET+12268>
ebp          0x8048fac 0x8048fac
esi          0x0      0
---Type <return> to continue, or q <return> to quit---
```

```
remote Thread 1 In: asm_start_process L49 PC: 0xc0022c3d
edi          0x0      0
eip          0xc0022c3d 0xc0022c3d <asm_start_process+13>
eflags       0x282    [ SF IF ]
cs           0x20     32
ss           0x10     16
ds           0x33     51
es           0x33     51
---Type <return> to continue, or q <return> to quit---
```

可以看到此时的数据寄存器都为0。然后，子进程回到 `asm_system_call`，继续执行下面的语句。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utls/asm_utls.asm
B+ 146      int 0x80
147
> 148      pop edi
149      pop esi
150      pop edx
151      pop ecx

0xc0022cda <asm_system_call+23> mov     edi,DWORD PTR [ebp+0x1c]
0xc0022cdd <asm_system_call+26> int     0x80
> 0xc0022cdf <asm_system_call+28> pop     edi
0xc0022ce0 <asm_system_call+29> pop     esi
0xc0022ce1 <asm_system_call+30> pop     edx
0xc0022ce2 <asm_system_call+31> pop     ecx

remote Thread 1 In: asm_system_call L148 PC: 0xc0022cdf
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx          0x0      0
esp          0x8048f98 0x8048f98
ebp          0x8048fac 0x8048fac
esi          0x0      0
---Type <return> to continue, or q <return> to quit---
```



```

remote Thread 1 In: asm_system_call L148 PC: 0xc0022cdf
edi      0x0      0
eip      0xc0022cdf 0xc0022cdf <asm_system_call+28>
eflags   0x216    [ PF AF IF ]
cs       0x2b     43
ss       0x3b     59
ds       0x33     51
es       0x33     51
---Type <return> to continue, or q <return> to quit---

```

此时的cs和ss寄存器发生了变化。而其他没有发生变化。cs 为代码段寄存器，ss 为栈段寄存器，而这里函数进行了返回，因此cs和ss发生改变。

然后，子进程返回 fork()。注意到此时的数据寄存器值为0，段寄存器没有改变。

终端

```

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/syscall.cpp
35     int fork() {
36         return asm_system_call(2);
> 37     }
38
39     int syscall_fork() {
40         return programManager.fork();

0xc0020f69 <fork()+21>      call  0xc0022cc3 <asm_system_call>
0xc0020f6e <fork()+26>      add    esp,0x20
> 0xc0020f71 <fork()+29>    leave
0xc0020f72 <fork()+30>      ret
0xc0020f73 <syscall_fork()>  push  ebp
0xc0020f74 <syscall_fork()+1> mov   ebp,esp

remote Thread 1 In: fork L37 PC: 0xc0020f71
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048fd4 0x8048fd4
ebp      0x8048fdc 0x8048fdc
esi      0x0      0
---Type <return> to continue, or q <return> to quit---

```

```

remote Thread 1 In: fork L37 PC: 0xc0020f71
edi      0x0      0
eip      0xc0020f71 0xc0020f71 <fork()+29>
eflags   0x206    [ PF IF ]
cs       0x2b     43
ss       0x3b     59
ds       0x33     51
es       0x33     51
---Type <return> to continue, or q <return> to quit---

```

上面的eax为0，因此子进程的返回值，即pid为0。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/setup.cpp
B+ 40      int pid = fork();
    41
> 42      if (pid == -1)
    43      {
    44          printf("can not fork\n");
    45      }

B+ 0xc0020fcf <first_process()+6> call    0xc0020f54 <fork(>
    0xc0020fd4 <first_process()+11> mov     DWORD PTR [ebp-0xc],eax
> 0xc0020fd7 <first_process()+14> cmp     DWORD PTR [ebp-0xc],0xffffffff
    0xc0020fdb <first_process()+18> jne     0xc0020fef <first_process()+38>
    0xc0020fdd <first_process()+20> sub     esp,0xc
    0xc0020fe0 <first_process()+23> push   0xc0022e6c

remote Thread 1 In: first process                                L42    PC: 0xc0020fd7
eax                0x0      0
ecx                0x0      0
edx                0x0      0
ebx                0x0      0
esp                0x8048fe4  0x8048fe4
ebp                0x8048ffc  0x8048ffc
esi                0x0      0
---Type <return> to continue, or q <return> to quit---

remote Thread 1 In: first process                                L42    PC: 0xc0020fd7
edi                0x0      0
eip                0xc0020fd7  0xc0020fd7 <first_process()+14>
eflags             0x206     [ PF IF ]
cs                 0x2b      43
ss                 0x3b      59
ds                 0x33      51
es                 0x33      51
---Type <return> to continue, or q <return> to quit---
```

然后，子进程回到 `first_process` 执行 `printf` 语句。然后结束。

### 下面进行父进程的跟踪。

我在 `setup.cpp:40`，即父进程的 `int pid = fork();` 设置了断点，然后一步步调试，跟踪 `fork()` 的执行。因为没有设置跟踪子进程的模式，所以默认情况下跟踪父进程。

调用 `fork()` 后，函数进入 `asm_system_call`，该函数将系统调用的参数保存在地特权级的栈中。然后，我们从低特权级转移到高特权级，CPU 从 TSS 中加载高特权级的栈地址到 `esp` 寄存器中。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
125     pop ds
126     mov eax, [ASM_TEMP]
127
128     iret
129     asm system_call:
> 130     push ebp
131     mov ebp, esp
132
133     push ebx
134     push ecx
135     push edx
136     push esi
137     push edi

remote Thread 1 In: asm system_call L130 PC: 0xc0022cb3
edx      0x0      0
ebx      0x0      0
esp      0x8048fb0  0x8048fb0
ebp      0x8048fdc  0x8048fdc
esi      0x0      0
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

即将调用中断 `int 0x80`，系统会根据寄存器 `eax` 中的值调用相应的系统调用函数。如下图，`eax` 为 2，因此系统调用的是第 2 号系统调用函数。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
137     push edi
138
139     mov eax, [ebp + 2 * 4]
140     mov ebx, [ebp + 3 * 4]
141     mov ecx, [ebp + 4 * 4]
142     mov edx, [ebp + 5 * 4]
143     mov esi, [ebp + 6 * 4]
144     mov edi, [ebp + 7 * 4]
145
> 146     int 0x80
147
148     pop edi
149     pop esi

remote Thread 1 In: asm system_call L146 PC: 0xc0022ccd
eax      0x2      2
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048f98  0x8048f98
ebp      0x8048fac  0x8048fac
esi      0x0      0
---Type <return> to continue, or q <return> to quit---
```

然后，父进程进入 `asm_system_call_handler`，然后在里面的 `call dword[system_call_table + eax * 4]`，进入系统调用函数。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
84
85     ret
86     ; int asm_system_call_handler();
87     asm_system_call_handler:
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

remote Thread 1 In: asm_system_call_handler L89 PC: 0xc0022c78
eax      0x2      2
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xc0025dc8 0xc0025dc8 <PCB_SET+8168>
ebp      0x8048fac 0x8048fac
esi      0x0      0
---Type <return> to continue, or q <return> to quit---
```

[system\_call\_table + eax \* 4] 就是系统调用号对应的系统调用处理函数的地址。然后进入系统调用函数。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
106
107     ; 参数压栈
108     push edi
109     push esi
110     push edx
111     push ecx
112     push ebx
113
114     sti
> 115     call dword[system_call_table + eax * 4]
116     cli
117
118     add esp, 5 * 4

remote Thread 1 In: asm_system_call_handler L115 PC: 0xc0022c96
edx      0x0      0
ebx      0x0      0
esp      0xc0025d88 0xc0025d88 <PCB_SET+8104>
ebp      0x8048fac 0x8048fac
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

然后父进程进入 `syscall_fork()`，再由 `syscall_fork()` 进入 `programManager::fork()`。

执行完 `programManager::fork()` 后，父进程返回 `syscall_fork()`。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/syscall.cpp
36     return asm_system_call(2);
37 }
38
39 int syscall_fork() {
40     return programManager.fork();
> 41 }^?
42
43
44
45
46
47
48

remote Thread 1 In: syscall_fork L41 PC: 0xc0020f89
eax      0x2      2
ecx      0xc0010f9c -1073672292
edx      0x0      0
ebx      0x0      0
esp      0xc0025d78 0xc0025d78 <PCB_SET+8088>
ebp      0xc0025d80 0xc0025d80 <PCB_SET+8096>
---Type <return> to continue, or q <return> to quit---
```

接着返回 `asm_system_call_handler` 函数

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
114     sti
115     call dword[system_call_table + eax * 4]
> 116     cli
117
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]

remote Thread 1 In: asm system call handler L116 PC: 0xc0022c9d
esp      0xc0025d78 0xc0025d78 <PCB_SET+8088>
ebp      0xc0025d80 0xc0025d80 <PCB_SET+8096>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) s
asm_system_call_handler () at ../src/utils/asm_utils.asm:116
(gdb) ss
```

然后，返回到 `asm_system_call` 函数。查看此时寄存器的值：

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
146         int 0x80
147
B+> 148     pop edi
149         pop esi
150         pop edx
151         pop ecx
152         pop ebx
153         pop ebp
154
155         ret
156
157         ; void asm_init_page_reg(int *directory);
158         asm_init_page_reg:

remote Thread 1 In: asm_system_call                                L148  PC: 0xc0022ccf
eax         0x2          2
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0x8048f98     0x8048f98
ebp         0x8048fac     0x8048fac
---Type <return> to continue, or q <return> to quit---
```

然后，返回 `fork()`。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/syscall.cpp
32         return stdio.print(str);
33     }
34
35     int fork() {
36         return asm_system_call(2);
> 37     }
38
39     int syscall_fork() {
40         return programManager.fork();
41     }^?
42
43
44

remote Thread 1 In: fork                                          L37   PC: 0xc0020f71
eax         0x2          2
ecx         0x0          0
edx         0x0          0
ebx         0x0          0
esp         0x8048fd4     0x8048fd4
ebp         0x8048fdc     0x8048fdc
---Type <return> to continue, or q <return> to quit---
```

可以看到此时的 `eax` 寄存器值为2，因此返回值为2。

最后父进程返回到 `first_process` 继续执行下面的语句。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/setup.cpp
37
38     void first_process()
39     {
40         int pid = fork();
41
42         if (pid == -1)
43         {
44             printf("can not fork\n");
45         }
46         else
47         {
48             if (pid)
49             {
remote Thread 1 In: first_process                                L42    PC: 0xc0020fd7
esp                0x8048fd4                0x8048fd4
ebp                0x8048fdc                0x8048fdc
---Type <return> to continue, or q <return> to quit---
Quit
(gdb) s
first_process () at ../src/kernel/setup.cpp:42
(gdb)
```

父子进程执行完 `ProgramManager::fork` 后的返回过程的异同:

执行完 `programManager::fork` 后, 父进程一步一步地返回上一步调用它的函数, 即从

`programManager::fork()` --> `syscall_fork()` --> `asm_system_call_handler` -->

`asm_system_call` --> `fork()` --> `first_process`. 最后执行 `first_process` 的 `printf` 语句结束。

而子进程则是在执行完 `programManager::fork` 后, 进入 `asm_start_process`, 然后和父进程一样回到 `asm_system_call` 的 `int 0x80` 语句后面, 继续执行下面的语句。然后和父进程一样从

`asm_system_call` 返回 `fork()`, 再返回 `first_process` 执行 `printf` 语句。

**fork的返回值:**

1、父进程在 `programManager::fork` 中返回创建的子进程的 `pid`, 然后将该值一直保存在 `eax` 中, 逐步返回至 `first_process`. 因此, 父进程返回的是子进程的 `pid`。

2、在创建子进程进行资源的复制时, 在 `ProgramManager::fork()` 中的 `copyProcess` 进行资源的复制时, 我们手动将子进程的 `eax` 设置成了 0, 子进程在 `asm_start_process` 中通过 `popad` 指令把 0 放在 `eax` 中, 然后逐步返回至 `first_process`. 因此, 子进程返回的是 0。

## Assignment 3 哼哈二将 wait & exit

(1) `exit` 的执行过程:

线程由 `exit()` 函数进入 `asm_system_call`.

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/syscall.cpp
39     int syscall_fork() {
40         return programManager.fork();
41     }
42
43     void exit(int ret) {
> 44         asm system call(3, ret);
45     }
46
47     void syscall_exit(int ret) {
48         programManager.exit(ret);
49     }^?
50
51

remote Thread 1 In: exit                                L44    PC: 0xc002115b
Breakpoint 1 at 0xc0021248: file ../src/kernel/setup.cpp, line 57.
(gdb) c
Continuing.

Breakpoint 1, second_thread (arg=0x0) at ../src/kernel/setup.cpp:57
(gdb) s
exit (ret=0) at ../src/kernel/syscall.cpp:44
(gdb)
```

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
125     pop ds
126     mov eax, [ASM_TEMP]
127
128     iret
129     asm system call:
> 130     push ebp
131     mov ebp, esp
132
133     push ebx
134     push ecx
135     push edx
136     push esi
137     push edi

remote Thread 1 In: asm system call                    L130   PC: 0xc0022f13
Continuing.

Breakpoint 1, second_thread (arg=0x0) at ../src/kernel/setup.cpp:57
(gdb) s
exit (ret=0) at ../src/kernel/syscall.cpp:44
(gdb) s
asm_system_call () at ../src/utils/asm_utils.asm:130
(gdb) █
```

然后从 `asm_system_call` 中的 `int 0x80` 进入 `asm_system_call_handler`。



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/asm_utils.asm
84
85         ret
86         ; int asm_system_call_handler();
87         asm_system_call_handler:
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

remote Thread 1 In: asm system call handler          L89   PC: 0xc0022ed8
Breakpoint 1, second_thread (arg=0x0) at ../src/kernel/setup.cpp:57
(gdb) s
exit (ret=0) at ../src/kernel/syscall.cpp:44
(gdb) s
asm_system_call () at ../src/utils/asm_utils.asm:130
(gdb) s
asm_system_call_handler () at ../src/utils/asm_utils.asm:89
(gdb) |
```

然后, 跳转到 `syscall_exit`, 再进入 `programManager.exit()`, 进入如下操作:

1. 标记PCB状态为 `DEAD` 并放入返回值。
2. 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。否则不做处理。
3. 立即执行线程/进程调度。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/syscall.cpp
43         void exit(int ret) {
44             asm_system_call(3, ret);
45         }
46
47         void syscall_exit(int ret) {
> 48             programManager.exit(ret);
49         }^?
50
51
52
53
54
55

remote Thread 1 In: syscall exit          L48   PC: 0xc002117c
asm_system_call_handler () at ../src/utils/asm_utils.asm:108
asm_system_call_handler () at ../src/utils/asm_utils.asm:109
asm_system_call_handler () at ../src/utils/asm_utils.asm:110
asm_system_call_handler () at ../src/utils/asm_utils.asm:111
asm_system_call_handler () at ../src/utils/asm_utils.asm:112
asm_system_call_handler () at ../src/utils/asm_utils.asm:114
syscall_exit (ret=0) at ../src/kernel/syscall.cpp:48
(gdb) |
```

最后, 通过 `programManager.exit()` 进入 `schedule()` 函数进行线程的调度。

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/program.cpp
127
128     activateProgramPage(next);
129
130     asm_switch_thread(cur, next);
131
132     interruptManager.setInterruptStatus(status);
133 }
134
135 void program_exit()
136 {
137     PCB *thread = programManager.running;
138     thread->status = ProgramStatus::DEAD;
139
remote Thread 1 In: ProgramManager::schedule          L96   PC: 0xc00202db
asm_interrupt_status () at ../src/utils/asm_utils.asm:217
InterruptManager::getInterruptStatus (this=0xc00344e4 <interruptManager>)
    at ../src/kernel/interrupt.cpp:116
ProgramManager::schedule (this=0xc0034500 <programManager>)
    at ../src/kernel/program.cpp:96
(gdb) b program.cpp:130
Breakpoint 3 at 0xc00203f4: file ../src/kernel/program.cpp, line 130.
(gdb)
```

然后通过 `asm_switch_thread` 函数更新下一个执行的线程/进程。然后由线程调度函数进行调度。

### (2) 进程退出后能够隐式地调用`exit`和此时的`exit`返回值是0的原因:

在 `load_process` 中, 我们在进程的3特权级栈中的栈顶处 `userStack[0]` 放入`exit`的地址, CPU会认为 `userStack[1]` 是`exit`的返回地址, `userStack[2]` 是`exit`的参数。当我们从0特权级返回特权级时, CPU获取 `userStack[0]` 的内容, 进程会跳转到 `exit` 函数执行。

### (3) `wait`的执行过程:

和前面的系统调用函数一样, `wait`的进口是显式调用 `wait()`, 然后跳转到 `asm_system_call`, 进一步跳转到 `programManager.wait(retval)` 执行。

`programManager.wait(retval)` 中, 父进程一直在循环检测是否有已死的子进程, 如果有则对其PCB进行释放, 如果有子进程, 但是子进程还没死, 则被阻塞。

通过`while`找到死的子进程或者没有找到死的子进程:

```
while (item)
{
    child = ListItem2PCB(item, tagInAllList);
    if (child->parentPid == this->running->pid)
    {
        flag = false;
        if (child->status == ProgramStatus::DEAD)
        {
            break;
        }
    }
    item = item->next;
}
```

如果找到死的子进程, 则将子进程的PCB进行释放, 然后返回子进程的pid。

```

int pid = child->pid;
releasePCB(child);
interruptManager.setInterruptStatus(interrupt);
return pid;

```

如果子进程已经返回，则什么也不干，返回-1.

```

interruptManager.setInterruptStatus(interrupt);
return -1;

```

如果，子进程没有返回，但是还没死，则执行调度，但是调度后该函数不会返回，而是循环检测，直到回收所有子进程的PCB。

```

interruptManager.setInterruptStatus(interrupt);
schedule();

```

在下面的 `setup.cpp` 中，父进程分别创建了两个子进程。子进程被创建后，执行输出语句，然后退出。

```

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

    if (pid)
    {
        pid = fork();
        if (pid)
        {
            while ((pid = wait(&retval)) != -1)
            {
                printf("wait for a child process, pid: %d, return value: %d\n",
                    pid, retval);
            }

            printf("all child process exit, programs: %d\n",
                programManager.allPrograms.size());

            asm_halt();
        }
        else
        {
            uint32 tmp = 0xffffffff;
            while (tmp)
                --tmp;
            printf("exit, pid: %d\n", programManager.running->pid);
            exit(123934);
        }
    }
    else
    {
        uint32 tmp = 0xffffffff;
        while (tmp)
            --tmp;
        printf("exit, pid: %d\n", programManager.running->pid);
        exit(-123);
    }
}

```

```

    }
}

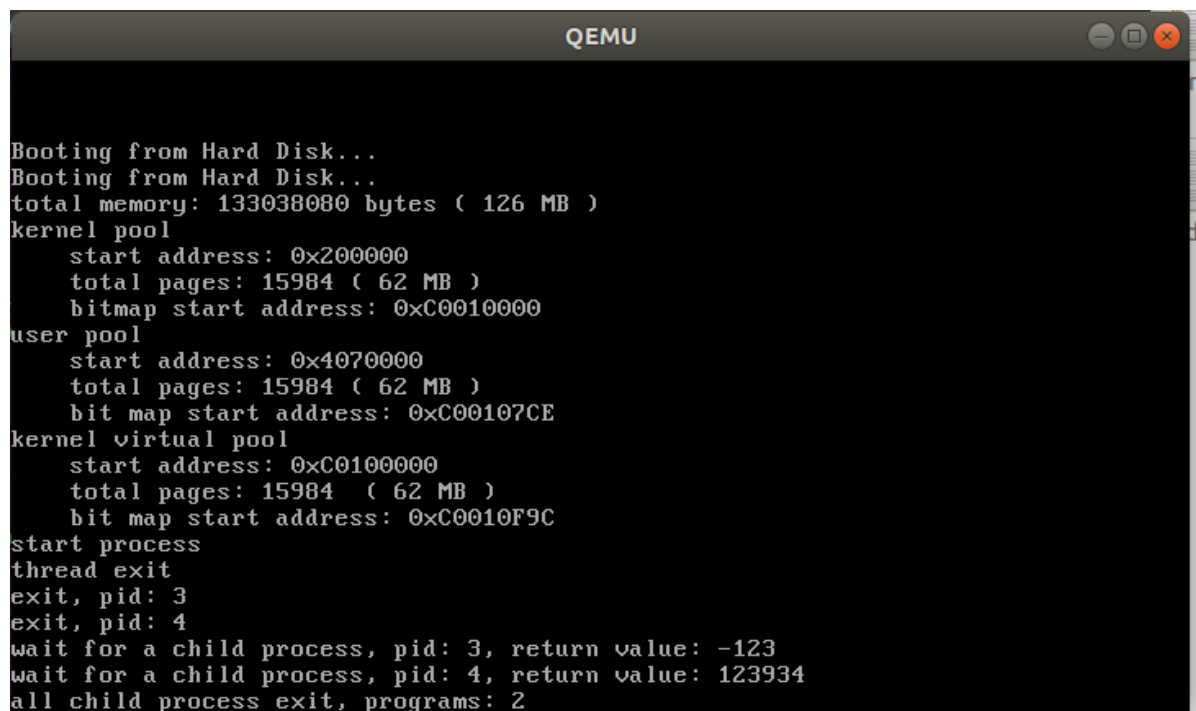
void second_thread(void *arg)
{
    printf("thread exit\n");
    //exit(0);
}

void first_thread(void *arg)
{
    printf("start process\n");
    programManager.executeProcess((const char *)first_process, 1);
    programManager.executeThread(second_thread, nullptr, "second", 1);
    asm_halt();
}

extern "C" void setup_kernel()
{
    ...
    // 设置4号系统调用
    systemService.setSystemCall(4, (int)syscall_wait);
    ...
}

```

查看运行结果：



The image shows a QEMU terminal window with a black background and white text. The text displays the boot process of a system, including memory pool initialization and process execution. The output is as follows:

```

QEMU

Booting from Hard Disk...
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
thread exit
exit, pid: 3
exit, pid: 4
wait for a child process, pid: 3, return value: -123
wait for a child process, pid: 4, return value: 123934
all child process exit, programs: 2

```

父进程等两个进程结束后，才执行两次wait操作，把子进程的PCB回收。然后输出相关语句。

#### (4) 实现回收僵尸进程的有效方法:

创建第一个进程专门用于回收其他被父进程遗弃的僵尸子进程。

1. 如果父进程先于子进程结束，那么子进程的父进程自动改为第一个进程进程。
2. 如果第一个进程的子进程结束，则第一个进程会自动回收其子进程的资源而不是让它变成僵尸进程。

为实现父进程先于子进程结束，那么子进程的父进程自动改为第一个进程进程，我在 `exit` 中做了如下修改：

```
void ProgramManager::exit(int ret)
{
    PCB *child;
    ListItem *item;
    interruptManager.disableInterrupt();

    item = this->allPrograms.head.next;

    // 查找子进程
    while (item)
    {
        child = ListItem2PCB(item, tagInAllList);
        if (child->parentPid == this->running->pid)
        {
            if (child->status != ProgramStatus::DEAD)
            {
                child->parentPid = FIRST_PROCESS_PID;
            }
        }
        item = item->next;
    }

    ...
}
```

让每个进程结束前都检查一下是否有未结束的子进程，如果有，则将子进程的 `parentPid` 变为 `FIRST_PROCESS_PID`。

其中 `FIRST_PROCESS_PID` 定义在 `os_constant.h` 中，为第一个进程的pid，为1。

```
#define FIRST_PROCESS_PID 1
```

实现如下，在 `setup.cpp` 中实现第一个进程的功能：

```
void first_process()
{
    int pid ;
    int retval;
    PCB *parent = programManager.running;
    int first_process_pid = parent->pid;
    printf("first_process_pid is %d.\n",first_process_pid);
    while(true){
        while ((pid = wait(&retval)) != -1)
        {
            ;
        }
    }
}
```

```

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

    //printf("all child process exit, programs: %d\n",
programManager.allPrograms.size());
}
asm_halt();
}

```

第一个进程永远不会结束，它会一直循环检测是否有其他僵尸进程，如果有，并且僵尸进程运行完，则将它们回收。

```

void second_process()
{
    int pid = fork();
    if( pid ){
        printf("I am going to exit, good bye my dear child!\n");
        exit(1);
    }
    if( !pid )
    {
        printf("I am child. I am going to be dead!\n");
        exit(2);
    }
}

```

然后创建第二个进程，在第二个进程中，利用fork创建一个子进程，然后父进程先 `exit`，比子进程先一步离开。然后子进程再执行 `exit`，变成没有父进程的，已执行完的僵尸进程。

```

void second_thread(void *arg)
{
    printf("thread exit\n");
    //exit(0);
}

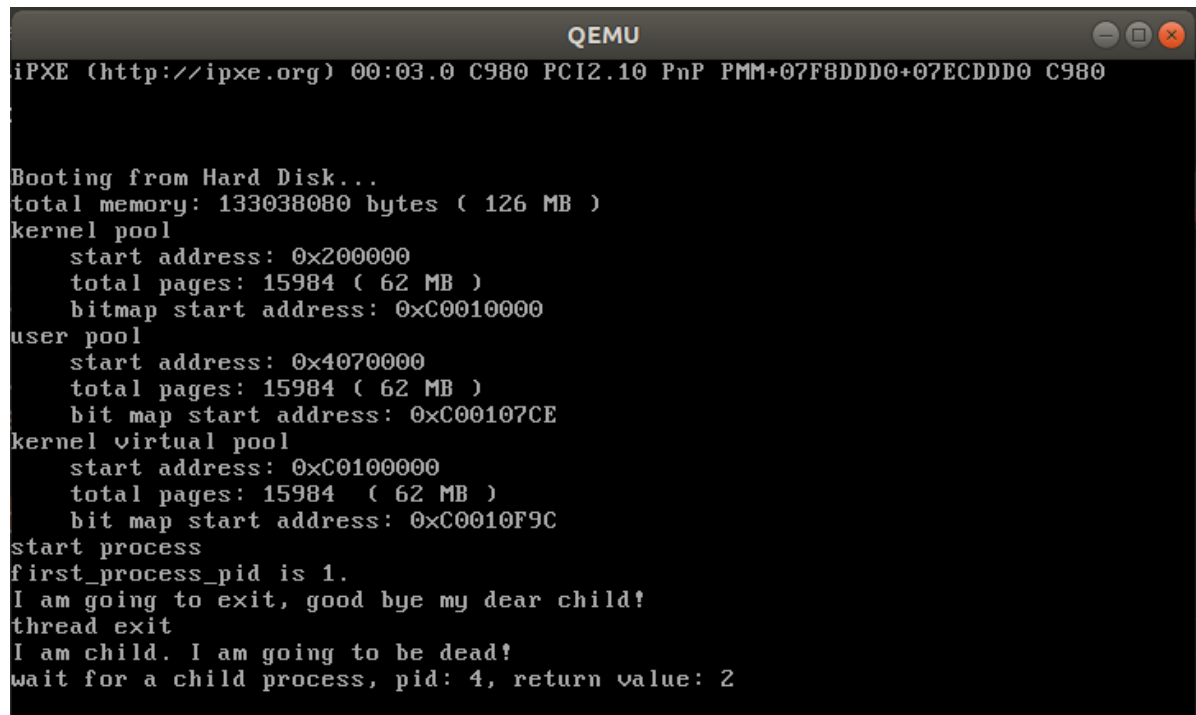
void first_thread(void *arg)
{
    printf("start process\n");
    int pid = programManager.executeProcess((const char *)first_process, 1);
    //printf("first_process pid is:%d. \n");
    programManager.executeProcess((const char *)second_process, 1);
    programManager.executeThread(second_thread, nullptr, "second", 1);
    asm_halt();
}

extern "C" void setup_kernel()
{
    ...
    int pid = programManager.executeThread(first_thread, nullptr, "first
thread", 1);
    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }
}

```

```
}  
...  
}
```

编译运行，查看结果：

A screenshot of a QEMU terminal window. The title bar says 'QEMU'. The terminal output shows the boot process of a virtual machine. It starts with 'iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980'. Then it says 'Booting from Hard Disk...'. It reports 'total memory: 133038080 bytes ( 126 MB )'. It then shows memory pool information for 'kernel pool' and 'user pool'. The 'kernel pool' has a start address of 0x200000, 15984 pages (62 MB), and a bitmap start address of 0xC0010000. The 'user pool' has a start address of 0x4070000, 15984 pages (62 MB), and a bit map start address of 0xC00107CE. It also shows 'kernel virtual pool' with a start address of 0xC0100000, 15984 pages (62 MB), and a bit map start address of 0xC0010F9C. The process then starts, and the first process pid is 1. It says 'I am going to exit, good bye my dear child!' and 'thread exit'. Then it says 'I am child. I am going to be dead!' and 'wait for a child process, pid: 4, return value: 2'.

从输出的语句顺序可以看出，第二个进程先离开，但是子进程还没结束。所以该进程在执行 `exit` 时会把将子进程的 `parentPid` 变为 `FIRST_PROCESS_PID`。

然后第二个线程执行。

然后子进程才输出语句，然后执行 `exit`，返回值为2. 此时子进程才结束。

然后，第一个进程检测到一个已经结束了的子进程，所以将它进行回收，故输出语句 `wait for a child process, pid: 4, return value: 2.`

至此，我们实现了僵尸进程的回收。

## 实验感想：

1、TSS在任务（进程）切换时起着重要的作用，通过它保存CPU中各寄存器的值，实现任务的切换。CPU通过tr寄存器找到TSS的信息。

2、操作系统不希望用户进程访问内核数据，所以需要给指令、数据附上一个特权级的属性，让程序受限制。而特权级分为0，1，2，3四种，用户态是最低等级的3 特权级，内核态就是最高等级的0 特权级。处理器在访问数据或跳转到代码时，需要进行特权级检查。

3、用fork创建的进程中，子进程是父进程的副本。父子进程共享代码段，但对于数据段、栈段等其他资源，父子进程并不共享。

4、对进程的回收，这里实现了两种方法，一种子进程在父进程结束之前结束，子进程由父进程回收；另一种是，父进程在子进程结束之前结束了，则子进程由第一个进程回收。

