



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验 4

从内核态到用户态

实验课程: 操作系统原理实验

实验名称: 从内核态到用户态

专业名称: 计算机科学与技术

学生姓名: 钟旺烜

学生学号: 23336342

实验地点: 实验楼 B203

实验成绩:

报告时间: 2025 年 6 月 11 日

Section 1 实验概述

●实验任务 1：系统调用

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

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

●实验任务 2：Fork 的奥秘

实现 fork 函数，并回答以下问题：

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

●实验任务 3：哼哈二将 wait & exit

实现 wait 函数和 exit 函数，并回答以下问题。

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

Section 2 实验步骤与实验结果

----- 实验任务 1 -----

●任务要求：

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。

●思路分析：

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

在此次实验任务中，我编写了一个使用 `printf` 输出字符的系统调用，命名为系统调用 `syscall_1`，其系统调用号为 1。可以在进程中使用 `asm_system_call()` 函数调用系统调用号为 1 的函数，以 `int` 类型传递字符串参数，便可实现 `printf` 函数输出字符的功能。

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

●实验步骤：

首先实现能够在进程中使用 `printf` 打印字符的系统调用函数 `syscall_1()`。以实验指导中实现系统调用函数 `syscall_0()` 为例，`syscall_1()` 的函数设计代码如下所示：代码保存在 `setup.cpp` 中

```
void syscall_1(const char *str)
{
    printf("%s\n",str);
}
```

随后，使用以下代码将此系统调用函数加载进系统调用，设置其系统调用号为 1。

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

至此，我成功的添加了一个新的系统调用。

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

```
void first_process()
{
    asm_system_call(1,(int)"Hellon,World!");

    asm_halt();
}
```

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

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

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

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

在加载第一个进程进入系统调用前，下面是 TSS 中存储的内容：

```
(gdb) x/32xw 0xc00337c0
0xc00337c0 <tss>: 0x00000000 0xc0025720 0x00000010 0x00000000
0xc00337d0 <tss+16>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc00337e0 <tss+32>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc00337f0 <tss+48>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0033800 <tss+64>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0033810 <tss+80>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0033820 <tss+96>: 0x00000000 0x00000000 0xc003382c 0x00000000
```

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

随后在进入系统调用前，下面是栈 esp，以及各个段选择子对应储存的值：

```
37 void first_process()
38 {
39     asm_system_call(1,(int)"Hellon,World!");
40
41     asm_halt();
42 }
43
44 void first_thread(void *arg)
45 {
46     printf("start process\n");
47     programManager.executeProcess((const char *)first_process, 1);
48     programManager.executeProcess((const char *)first_process, 1);
}

remote Thread 1.1 In: first_process L38 PC: 0xc0021313
Undefined info command: "ss sp cs ds es fs gs". Try "help info".
(gdb) i r ss sp cs ds es fs gs
ss      0x3b      59
sp      0x8049000 0x8049000
cs      0x2b      43
ds      0x33      51
es      0x33      51
fs      0x33      51
gs      0x0       0
```

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

下面当进入系统调用函数后，栈 esp，以及各个段选择子对应储存的值如下所示：

```
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
97
98     mov eax, DATA_SELECTOR

remote Thread 1.1 In: asm_system_call_handler L88 PC: 0xc00226a7
(gdb) i r ss sp cs ds es fs gs
ss      0x10      16
sp      0xc002570c 0xc002570c <PCB_SET+8172>
cs      0x20      32
ds      0x33      51
es      0x33      51
fs      0x33      51
gs      0x0       0
```

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

```
(gdb) x/5xw 0xc002570c
0xc002570c <PCB_SET+8172>:    0xc00226ff    0x0000002b    0x000000212    0x08048fb8
0xc002571c <PCB_SET+8188>:    0x0000003b
```

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

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

栈 `esp`，以及各个段选择子对应储存的值如下所示

```
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

remote Thread 1.1 In: asm_system_call_handler L114 PC: 0xc00226c5
asm_system_call_handler () at ../src/utils/asm_utils.asm:114
(gdb) i r ss sp cs ds es fs gs
ss      0x10      16
sp      0xc00256c8 0xc00256c8 <PCB_SET+8104>
cs      0x20      32
ds      0x8       8
es      0x8       8
fs      0x33      51
gs      0x18      24
```

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

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

在退出中断后，栈 `esp`，以及各个段选择子对应储存的值如下所示


```
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
156

remote Thread 1.1 In: asm_system_call L148 PC: 0xc00226ff
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) i r ss sp cs ds es fs gs
ss          0x3b          59
sp          0x8048fb8     0x8048fb8
cs          0x2b          43
ds          0x33          51
es          0x33          51
fs          0x33          51
gs          0x0           0
(gdb)
```

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

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

```
(gdb) x/27xw 0xc00337c0
0xc00337c0 <tss>: 0x00000000 0xc0026720 0x00000010 0x00000000
0xc00337d0 <tss+16>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc00337e0 <tss+32>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc00337f0 <tss+48>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0033800 <tss+64>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0033810 <tss+80>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0033820 <tss+96>: 0x00000000 0x00000000 0xc003382c
(gdb)
```

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

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

●实验结果展示: 通过执行前述代码, 可得下图结果。

下面是我实现的系统调用函数的运行结果:

可以看到成功调用了 printf 函数在在 qemu 显示屏上打印了相应的字符

```
Machine View
QEMU

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

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
Hellow,World!
Hellow,World!
Hellow,World!
```

----- 实验任务 2 -----

●任务要求:

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。

●思路分析:

实验指导中给出了 `fork` 实现的 4 个问题。

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

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

使用 `gdb` 来调试运行操作系统，使用 `info register` 指令观察子进程在第一次被调度执行后一直到从 `fork()` 返回间跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完

`ProgramManager::fork` 后的返回过程的异同。

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

●实验步骤:

首先分析 `fork` 实现的基本思路:

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

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

```
int ProgramManager::fork()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    // 禁止内核线程调用
    PCB *parent = this->running;
    if (!parent->pageDirectoryAddress)
    {
        interruptManager.setInterruptStatus(status);
        return -1;
    }

    // 创建子进程
    int pid = executeProcess("", 0);
    if (pid == -1)
    {
        interruptManager.setInterruptStatus(status);
        return -1;
    }

    // 初始化子进程
    PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
    bool flag = copyProcess(parent, child);

    if (!flag)
    {
        child->status = ProgramStatus::DEAD;
        interruptManager.setInterruptStatus(status);
        return -1;
    }

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


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

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

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

- 1、中断的那一刻保存的父进程0特权栈的内容复制到子进程的0特权级栈中。
- 2、设置子进程的PCB、复制父进程的管理虚拟地址池的bitmap到子进程的管理虚拟地址池的bitmap。
- 3、将父进程的页目录表复制到子进程中
- 4、复制父进程页表和物理页的数据到对应的子进程的页表和物理页

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

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

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

```
// 复制进程0级栈
ProcessStartStack *childpss =
    (ProcessStartStack *)((int)child + PAGE_SIZE - sizeof(ProcessStartStack));
ProcessStartStack *parentpss =
    (ProcessStartStack *)((int)parent + PAGE_SIZE - sizeof(ProcessStartStack));
memcpy(parentpss, childpss, sizeof(ProcessStartStack));
// 设置子进程的返回值为0
childpss->eax = 0;

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

在0特权栈下，父进程所在的PCB顶部正好为0特权栈的底部，在一个`ProcessStartStack`大小

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

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

```
// 设置子进程的 PCB
child->status = ProgramStatus::READY;
child->parentPid = parent->pid;
child->priority = parent->priority;
child->ticks = parent->ticks;
child->ticksPassedBy = parent->ticksPassedBy;
strcpy(parent->name, child->name);

// 复制用户虚拟地址池
int bitmapLength = parent->userVirtual.resources.length;
int bitmapBytes = ceil(bitmapLength, 8);
memcpy(parent->userVirtual.resources.bitmap, child->userVirtual.resources.bitmap,
bitmapBytes);
```

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

3、将父进程的页目录表复制到子进程中

4、复制父进程页表和物理页的数据到对应的子进程的页表和物理页

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

下面是申请中转页的代码：

```
// 从内核中分配一页作为中转页
char *buffer = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
if (!buffer)
{
    child->status = ProgramStatus::DEAD;
    return false;
}
```

下面是分别进行页目录表复制和页表和物理页复制的代码：

```

// 子进程页目录表物理地址
int childPageDirPaddr = memoryManager.vaddr2paddr(child->pageDirectoryAddress);
// 父进程页目录表物理地址
int parentPageDirPaddr = memoryManager.vaddr2paddr(parent->pageDirectoryAddress);
// 子进程页目录表指针(虚拟地址)
int *childPageDir = (int *)child->pageDirectoryAddress;
// 父进程页目录表指针(虚拟地址)
int *parentPageDir = (int *)parent->pageDirectoryAddress;

// 子进程页目录表初始化
memset((void *)child->pageDirectoryAddress, 0, 768 * 4);

// 复制页目录表
for (int i = 0; i < 768; ++i)
{
    // 无对应页表
    if (!(parentPageDir[i] & 0x1))
    {
        continue;
    }

    // 从用户物理地址池中分配一页，作为子进程的页目录项指向的页表
    int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    if (!paddr)
    {
        child->status = ProgramStatus::DEAD;
        return false;
    }
    // 页目录项
    int pde = parentPageDir[i];
    // 构造页表的起始虚拟地址
    int *pageTableVaddr = (int *)(0xffc00000 + (i << 12));

    asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

    childPageDir[i] = (pde & 0x00000fff) | paddr;
    memset(pageTableVaddr, 0, PAGE_SIZE);

    asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
}
// 复制页表和物理页
for (int i = 0; i < 768; ++i)
{
    // 无对应页表
    if (!(parentPageDir[i] & 0x1))
    {
        continue;
    }

```

```

        // 从用户物理地址池中分配一页，作为子进程的页表项指向的物理页
        int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
        if (!paddr)
        {
            child->status = ProgramStatus::DEAD;
            return false;
        }

        // 构造物理页的起始虚拟地址
        void *pageVaddr = (void *)((i << 22) + (j << 12));
        // 页表项
        int pte = pageTableVaddr[j];
        // 复制出父进程物理页的内容到中转页
        memcpy(pageVaddr, buffer, PAGE_SIZE);

        asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

        pageTableVaddr[j] = (pte & 0x00000fff) | paddr;
        // 从中转页中复制到子进程的物理页
        memcpy(buffer, pageVaddr, PAGE_SIZE);

        asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
    }
}

```

3、4 步骤的流程可以概括为：

检查父进程中页表项或页面是否存在-->若存在，则进行页目录项和页面的复制

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

注意到 `asm_update_cr3()` 的使用，我们知道在开启二级分页机制后，在对虚拟地址进行访问时 CPU 会根据保存在 `cr3` 寄存器中的页目录表地址来进行对应的物理地址寻址操作，因此在进行复制操作的时候要随时注意当前访问地址操作使用的是父进程还是子进程的页目录表地址，防止 CPU 的 MMU 单元的地址转换得到我们不想要的结果。

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

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

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

```

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
97
remote Thread 1.1 In: asm_system_call_handler L88 PC: 0xc0022c77
eip      0xc0022c77      0xc0022c77 <asm_system_call_handler>
ss       0x10           16
esp      0xc0025dec      0xc0025dec <PCB_SET+8172>
cs       0x20           32
ds       0x33           51
es       0x33           51
fs       0x33           51
gs       0x0            0
(gdb)

```

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

```

(gdb) x/6xw 0xc0025dec
0xc0025dec <PCB_SET+8172>: 0xc0022ccf 0x0000002b 0x00000216 0x08048f98
0xc0025dfc <PCB_SET+8188>: 0x0000003b 0x00000000

```

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

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

```

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 1.1 In: asm_system_call L148 PC: 0xc0022ccf
(gdb) s
(gdb) n
(gdb) bt
#0 asm_system_call () at ../src/utils/asm_utils.asm:148
#1 0xc002236e in fork () at ../src/kernel/syscall.cpp:36
#2 0xc00217f9 in first_process () at ../src/kernel/setup.cpp:33
Backtrace stopped: Cannot access memory at address 0x8049000
(gdb)

```

可以看到，父进程在 `asm_system_call` 函数放回到 `fork()` 函数，再返回到 `first_program` 主进程函数

中。次数，父进程其 `eip`，数据寄存器和段寄存器所保存内容如下图所示

```
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
156
157 ; void asm_init_page_reg(int *directory);

remote Thread 1.1 In: asm_system_call
(gdb) i r eip ss esp cs ds es fs gs eax
eip      0xc0022cd5      0xc0022cd5 <asm_system_call+34>
ss       0x3b          59
esp      0x8048fb0      0x8048fb0
cs       0x2b          43
ds       0x33          51
es       0x33          51
fs       0x33          51
gs       0x0           0
eax      0x2           2
```

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

下面我们来看子进程在第一次被调度执行时开始的情况：

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

```
(gdb) bt
#0  asm_start_process () at ../src/utils/asm_utils.asm:49
#1  0xc0022ccf in asm_system_call () at ../src/utils/asm_utils.asm:146
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

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

```
145
146     int 0x80
147
148     pop edi
```

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

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

```
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
156
157 ; void asm_init_page_reg(int *directory);
158 asm_init_page_reg:

remote Thread 1.1 In: asm_system_call
(gdb) i r eip ss esp cs ds es fs gs eax
eip      0xc0022cd5      0xc0022cd5 <asm_system_call+34>
ss       0x3b          59
esp      0x8048fb0      0x8048fb0
cs       0x2b          43
ds       0x33          51
es       0x33          51
fs       0x33          51
gs       0x0           0
eax      0x0           0
```

在子进程返回到 `asm_system_call` 函数后，因为子进程的 3 特权级栈同父进程的 3 特权级栈一致，而 3 特权级栈保存了父进程在执行 `int 0x80` 后的逐步返回的返回地址。因此，父子进程的逐步返回的地址是相同的，从而实现了在执行 `fork` 后，父子进程从相同的点返回。

子进程在返回 `asm_system_call` 函数后的函数调用栈如下图所示

```
(gdb) bt
#0  asm_system_call () at ../src/utils/asm_utils.asm:148
#1  0xc002236e in fork () at ../src/kernel/syscall.cpp:36
#2  0xc00217f9 in first_process () at ../src/kernel/setup.cpp:33
Backtrace stopped: Cannot access memory at address 0x8049000
(gdb)
```

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

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

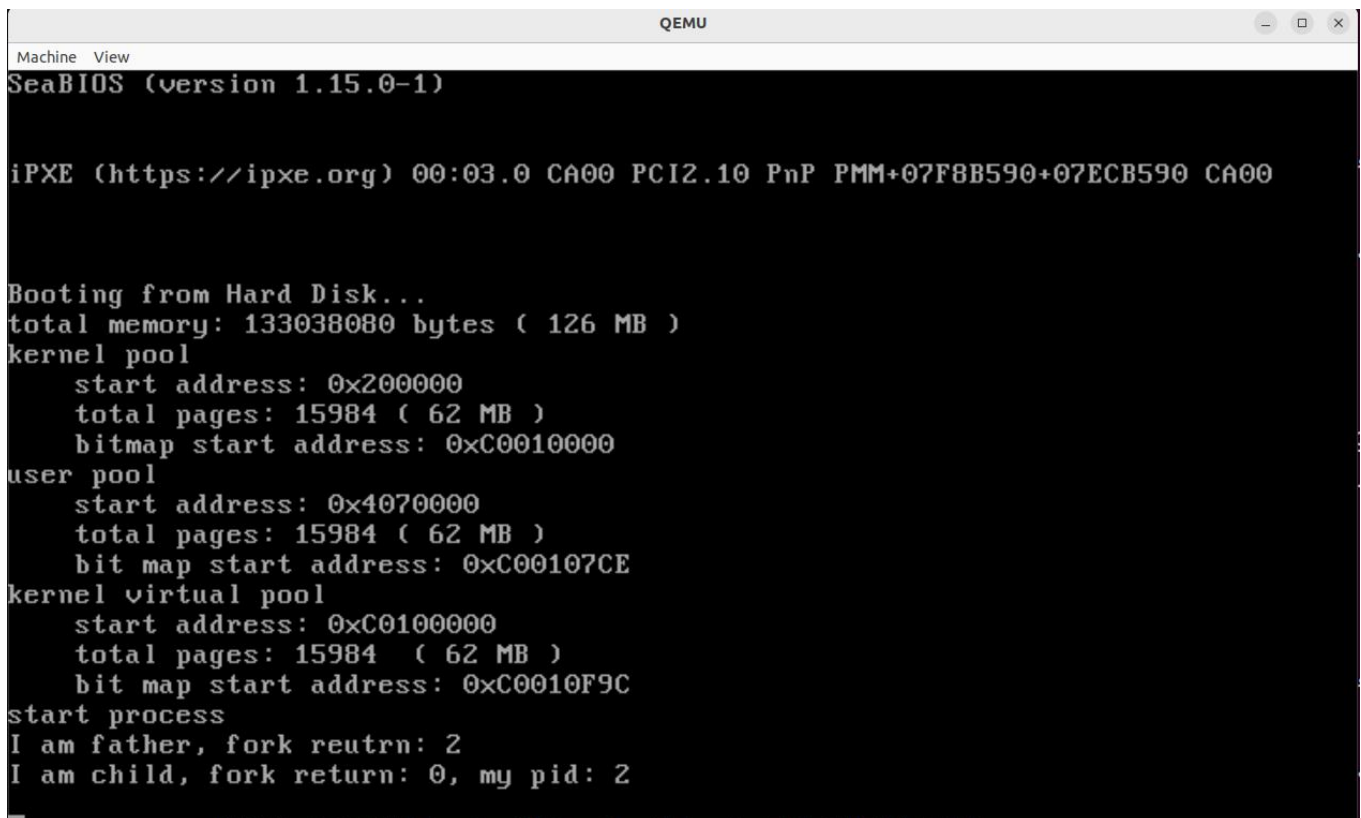
通过同一过程中父进程对比，可以看到在 `ProgramManager::fork` 后的返回过程中，除了 `eax` 的值不一样外，其他跳转地址、数据寄存器和段寄存器的值完全一致。

`fork` 是如何保证子进程的 `fork` 返回值是 0，而父进程的 `fork` 返回值是子进程的 `pid` 的问题也得以解决，因为子进程的 `fork` 返回值保存的 `eax` 寄存器中的值被人为修改为了 0，而父进程得到的返回值

就是 `ProgramManager::fork` 执行后返回的结果，即子进程的 `pid`。

●实验结果展示：通过执行前述代码，可得下图结果。

下面是运行 `fork` 操作后的展示结果：



```
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

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 return: 2
I am child, fork return: 0, my pid: 2
```

可以看到 `fork` 操作正确执行

----- 实验任务 3 -----

●任务要求：

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。

●思路分析：

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

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

// 找到刚刚创建的 PCB

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

替换这个提到的"存在风险的语句", 替换结果如下。

```
PCB *process = findProgramByPid(pid);
```

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

●实验步骤:

首先直接使用实验指导中给出的实例, 实例代码如下所示:

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

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

            printf("all child process exit, programs: %d\n", programManager.allPrograms.
                size());
            asm_halt();
        }
        else
        {
            uint32 tmp = 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();
}

```

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

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

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

下面通过 `exit()` 的代码来分析 `exit()` 的执行过程：

```

void ProgramManager::exit(int ret){
    // 关中断
    interruptManager.disableInterrupt();

    // 第一步，标记 PCB 状态为`DEAD`并放入返回值。
    PCB *program = this->running;
    program->retValue = ret;
    program->status = ProgramStatus::DEAD;

    int *pageDir, *page;
    int paddr;

    // 第二步，如果 PCB 标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池 bitmap
    的空间。
    if (program->pageDirectoryAddress)
    {
        pageDir = (int *)program->pageDirectoryAddress;
        for (int i = 0; i < 768; ++i)
        {
            if (!(pageDir[i] & 0x1))
            {
                continue;
            }

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

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

```



```

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

        paddr = memoryManager.vaddr2paddr((i << 22) + (j << 12));
        memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);
    }

    paddr = memoryManager.vaddr2paddr((int)page);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);
}

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

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

memoryManager.releasePages(AddressPoolType::KERNEL,
                           (int)program->userVirtual.resources.bitmap,
                           bitmapPages);

}

// 第三步，立即执行线程/进程调度。
schedule();
}

```

可以看到，当线程或进程进行 `exit()` 操作后，其进行以下三步操作：

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

第二步，如果 PCB 标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池 `bitmap` 的空间。按照创建进程时分配内存的逆过程释放进程所分配的内存以及管理内存的虚拟地址池。

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

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

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

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

```

void load_process(const char *filename)
{
    ...
    interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1);
}

```

```

    if (interruptStack->esp == 0)
    {
        printf("can not build process!\n");
        process->status = ProgramStatus::DEAD;
        asm_halt();
    }
    interruptStack->esp += PAGE_SIZE;

    // 设置进程返回地址
    int *userStack = (int *)interruptStack->esp;
    userStack -= 3;
    userStack[0] = (int)exit;
    userStack[1] = 0;
    userStack[2] = 0;

    interruptStack->esp = (int)userStack;

    ...
}

```

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

下面通过 wait()的代码来分析 wait()的执行过程:

```

int ProgramManager::wait(int *retval)
{
    PCB *child;
    ListItem *item;
    bool interrupt, flag;

    while (true)
    {
        interrupt = interruptManager.getInterruptStatus();
        interruptManager.disableInterrupt();

        item = this->allPrograms.head.next;

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

```

```

    }
}
item = item->next;
}

if (item) // 找到一个可返回的子进程
{
    if (retval)
    {
        *retval = child->retValue;
    }

    int pid = child->pid;
    releasePCB(child);
    interruptManager.setInterruptStatus(interrupt);
    return pid;
}
else
{
    if (flag) // 子进程已经返回
    {
        interruptManager.setInterruptStatus(interrupt);
        return -1;
    }
    else // 存在子进程，但子进程的状态不是 DEAD
    {
        interruptManager.setInterruptStatus(interrupt);
        schedule();
    }
}
}
}
}

```

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

至此我们便解释了 wait()的执行过程，下面我们来解释如何处理僵尸进程。

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

的判断逻辑。

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

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

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

```
// 创建进程的页目录表
process->pageDirectoryAddress = createProcessPageDirectory();
process->parentPid = -1;
```

这样，我们便可以判断该进程是否是子进程。

现在我们来解决判断父进程是否状态为 `DEAD` 并且其 PCB 已经被释放

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

下面是 `schedule()`中修改的代码段：

```
...
else if (running->status == ProgramStatus::DEAD)
{
    // 回收线程，子进程留到父进程回收
    if(!running->pageDirectoryAddress) {
        releasePCB(running);
    }
    else
    {
        if(running->parentPid == -1)
        {
            releasePCB(running);
        }
    }
}
...
```

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

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

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

```

PCB *ProgramManager::findProgramByPid(int pid)
{
    PCB *temp;
    ListItem *item;
    bool interrupt;

    item = this->allPrograms.head.next;

    interrupt = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    while (item)
    {
        temp = ListItem2PCB(item, tagInAllList);
        if(temp->pid == pid)
        {
            interruptManager.setInterruptStatus(interrupt);
            return temp;
        }
        item = item->next;
    }
    interruptManager.setInterruptStatus(interrupt);
    return nullptr;
}

```

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

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

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

```

...
else if (running->status == ProgramStatus::DEAD)
{
    // 回收线程，子进程留到父进程回收
    if(!running->pageDirectoryAddress) {
        releasePCB(running);
    }
    else
    {
        if(running->parentPid == -1)
        {
            releasePCB(running);
        }
        ListItem *item = this->allPrograms.head.next;
        while (item)
        {
            PCB *temp = ListItem2PCB(item, tagInAllList);

```



```

        if(temp->pageDirectoryAddress)
        {
            if(temp->status == ProgramStatus::DEAD && temp->parentPid != -1 &&
                findProgramByPid(temp->parentPid) == nullptr)
            {
                printf("release zombie process: %d,return val: %d\n", temp->pid,
                    temp->retValue);
                releasePCB(temp);
            }
        }
        item = item->next;
    }
}
...

```

以上我们便实现了回收僵尸进程的 PCB。

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

```

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

    if (pid)
    {
        pid = fork();
        if (pid)
        {
            printf("parent process return, pid %d\n",programManager.running->pid);
            return;
        }
        else
        {
            uint32 tmp = 0xffffffff;
            while (tmp)
                --tmp;
            printf("exit, pid: %d\n", programManager.running->pid);
            exit(12345);
        }
    }
    else
    {
        uint32 tmp = 0xffffffff;
        while (tmp)
            --tmp;
        printf("exit, pid: %d\n", programManager.running->pid);
        exit(-67890);
    }
}

```

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

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

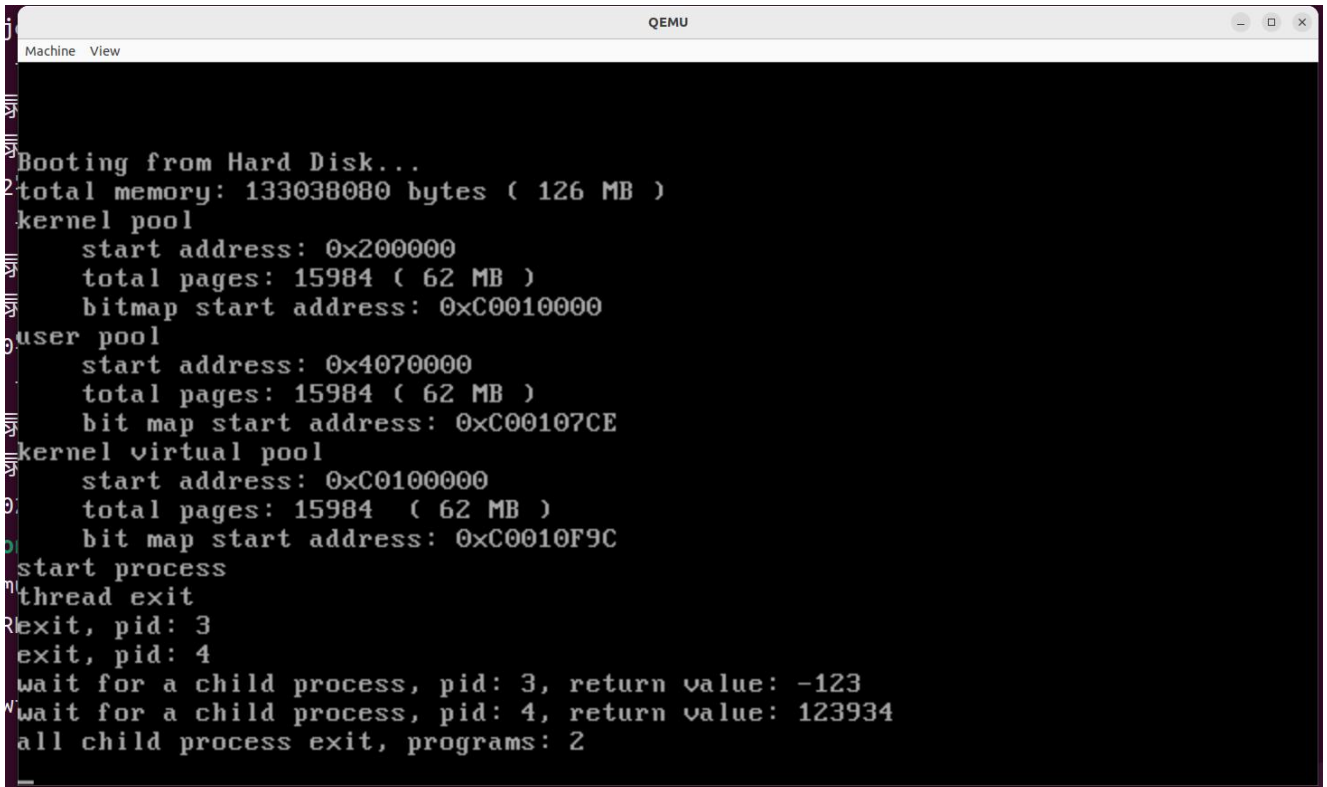
下面是对第一线程修改后的代码：

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

下面我们在虚拟机中编译运行操作系统，查看测试结果。

●实验结果展示：通过执行前述代码，可得下图结果。

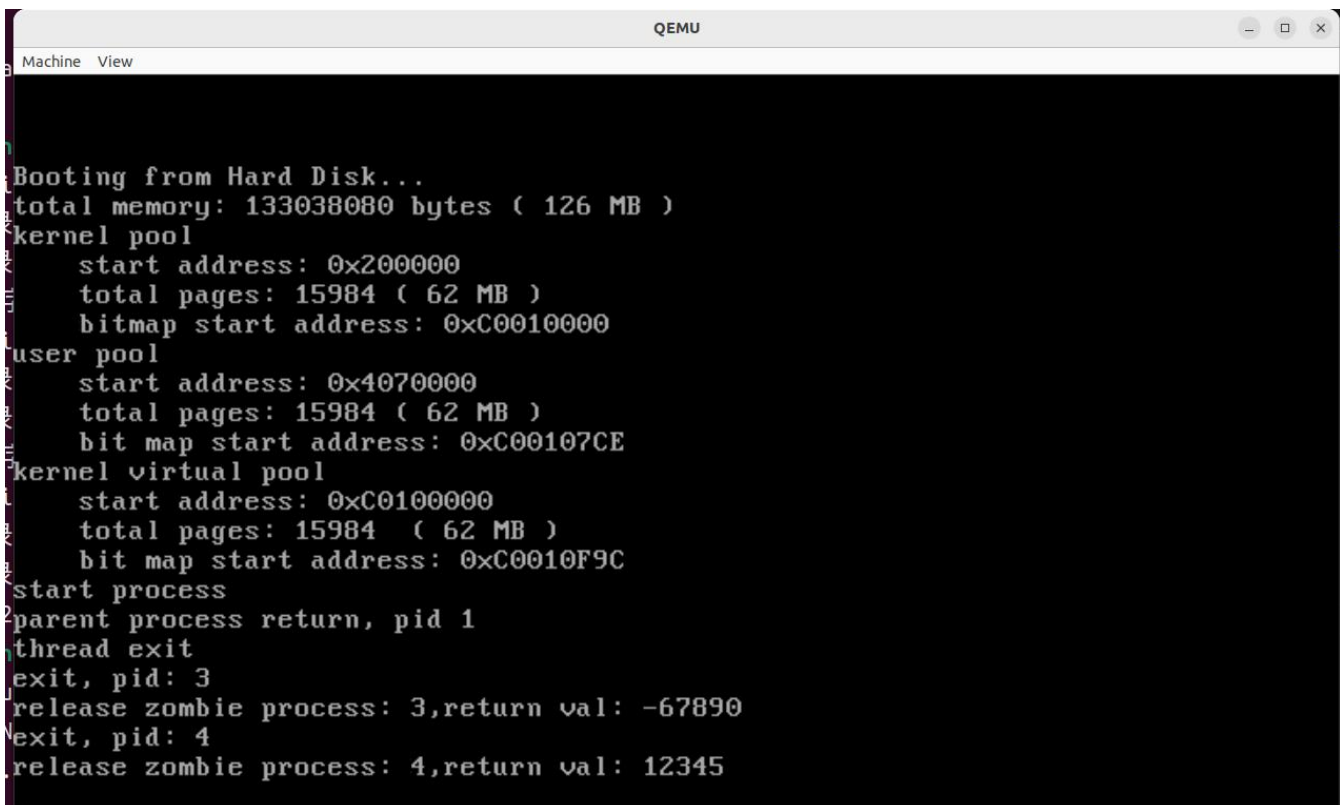
首先是实验指导实例的运行结果：



```
QEMU
Machine View
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
Rexit, 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
```

可以看到 `wait()` 和 `exit()` 函数都正确执行。

下面是测试僵尸进程释放函数的实例运行结果



```
Machine View
QEMU

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
parent process return, pid 1
thread exit
exit, pid: 3
release zombie process: 3,return val: -67890
exit, pid: 4
release zombie process: 4,return val: 12345
```

可以看到在父进程放回后，子进程在变为僵尸进程后其 PCB 被成功释放。

Section 3 实验总结与心得体会

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

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

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

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

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

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

Section 4 对实验的改进建议和意见

- 1、可以提供更多实现系统调用的任务以及教程。

Section 5 附录：参考资料清单

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

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