



本科生实验报告

实验课程 中山大学 2021 学年春季操作系统课程

实验名称 从内核态到用户态

专业名称 计算机科学与技术（超算）

学生姓名 黄玟瑜

学生学号 19335074

任课教师 陈鹏飞

实验地点

实验成绩

二〇二一年六月十六日

目录

1	Assignment 1: 系统调用	1
1.1	编写系统调用	1
1.2	GDB 分析	3
1.3	TSS 的作用	7
2	Assignment 2: Fork 的奥秘	8
2.1	分析 fork 实现的基本思路	8
2.2	GDB 追踪	12
2.3	fork 返回值解释	16
3	Assignment 3: 哼哈二将 wait & exit	19
3.1	分析 exit 执行过程	19
3.2	分析进程退出后隐式地调用 exit 和此时 exit 返回值是 0 的原因	22
3.3	分析 wait 执行过程	22
3.4	回收僵尸进程的方法	23
4	总结	27

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。

编写系统调用

编写系统调用函数如下，系统调用号为 7，函数功能为：

对输入的两个参数，若两个参数相加结果为 7 则打印学号姓名，否则不输出，返回值为两数之和。

```
1  #include "asm_utils.h"
2  #include "interrupt.h"
3  #include "stdio.h"
4  #include "program.h"
5  #include "thread.h"
6  #include "sync.h"
7  #include "memory.h"
8  #include "syscall.h"
9  // 屏幕IO处理器
10 STDIO stdio;
11 // 中断管理器
12 InterruptManager interruptManager;
13 // 程序管理器
14 ProgramManager programManager;
15 // 内存管理器
16 MemoryManager memoryManager;
17 // 系统调用
18 SystemService systemService;
19 int syscall_0(int first, int second, int third, int forth, int fifth)
20 {
21     printf("system call 0: %d, %d, %d, %d, %d\n",
22           first, second, third, forth, fifth);
23     return first + second + third + forth + fifth;
24 }
25 int syscall_7(int a, int b)
26 {
27     if(a + b == 7)
28         printf("system call 7: 19335074 wenny\n");
29     return a + b;
```

```

30 }
31 void first_thread(void *arg)
32 {
33     asm_halt();
34 }
35 extern "C" void setup_kernel()
36 {
37     // 中断管理器
38     interruptManager.initialize();
39     interruptManager.enableTimeInterrupt();
40     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler
        );
41     // 输出管理器
42     stdio.initialize();
43     // 进程/线程管理器
44     programManager.initialize();
45     // 内存管理器
46     memoryManager.openPageMechanism();
47     memoryManager.initialize();
48     // 初始化系统调用
49     systemService.initialize();
50     // 设置0号系统调用
51     systemService.setSystemCall(0, (int)syscall_0);
52     // 设置7号系统调用
53     systemService.setSystemCall(7, (int)syscall_7);
54     int ret;
55     ret = asm_system_call(7, 7);
56     printf("return value: %d\n", ret);
57     ret = asm_system_call(7, 3, 4);
58     printf("return value: %d\n", ret);
59     ret = asm_system_call(7, 3, 6);
60     printf("return value: %d\n", ret);
61     ret = asm_system_call(7, 7, -1);
62     printf("return value: %d\n", ret);
63
64     // 创建第一个线程
65     int pid = programManager.executeThread(first_thread, nullptr, "first
        thread", 1);
66     if (pid == -1)
67     {
68         printf("can not execute thread\n");
69         asm_halt();
70     }
71     ListItem *item = programManager.readyPrograms.front();
72     PCB *firstThread = ListItem2PCB(item, tagInGeneralList);

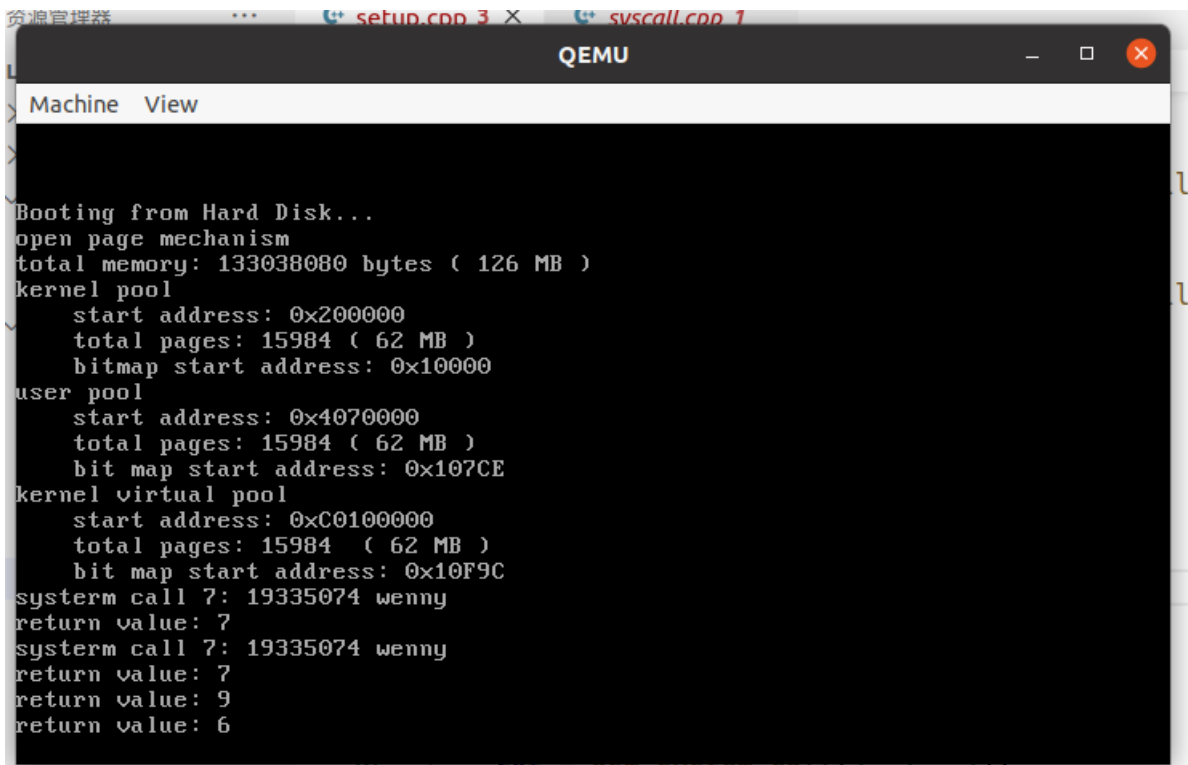
```

```

73     firstThread->status = RUNNING;
74     programManager.readyPrograms.pop_front();
75     programManager.running = firstThread;
76     asm_switch_thread(0, firstThread);
77     asm_halt();
78 }

```

结果如下所示：



```

Machine View
Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
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 7: 19335074 wenny
return value: 7
system call 7: 19335074 wenny
return value: 7
return value: 9
return value: 6

```

在主程序中依次输出了 4 次调用的返回值。

前面两种情况两数之和为 7，因此打印了学号姓名，后面两种情况两数之和不等于 7，因此不做打印。

GDB 分析

启动 GDB，在 `asm_system_call` 和 `asm_system_call_handler` 处设置断点。
运行。

在主程序中调用了 `asm_system_call`，因此运行后会在 `asm_system_call` 处停顿，保护现场完毕后，将系统调用号和系统调用的 5 个参数因此放入寄存器中，如下图所示：

（实际上只有 2 个参数，第 3、4、5 个参数默认为 0）

```

Register group: general
eax      0x7      7
ecx      0x0      0
edx      0x0      0
ebx      0x7      7
esp      0x7b98    0x7b98
ebp      0x7bbc    0x7bbc
esi      0x0      0
edi      0x0      0
eip      0x223ac   0x223ac <asm_system_call+32>
eflags   0x12     [ IOPL=0 AF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

../src/utils/asm_utils.asm
86      mov ecx, [ebp + 4 * 4]
87      mov edx, [ebp + 5 * 4]
88      mov esi, [ebp + 6 * 4]
89      mov edi, [ebp + 7 * 4]
90
>91      int 0x80
92
93      pop gs
94      pop fs
95      pop es
96      pop ds
97      pop edi
98      pop esi

```

由于设置了 80 号中断的中断描述符的 DPL=3，因此用户程序可以直接使用这个中断。此时 esp 为 0x7b98。

```

Register group: general
eax      0x7      7
ecx      0x0      0
edx      0x0      0
ebx      0x7      7
esp      0x7b88    0x7b88
ebp      0x7bbc    0x7bbc
esi      0x0      0
edi      0x0      0
eip      0x22351   0x22351 <asm_system_call_handler+1>
eflags   0x12     [ IOPL=0 AF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

../src/utils/asm_utils.asm
25
26      ASM_TEMP dd 0
27      ; int asm_system_call_handler();
28      asm_system_call_handler:
29      push ds
>30      push es
31      push fs
32      push gs
33      pushad
34
35      push eax
36
QEMU      ; ^&^ &^ $^ $^ ss^$^ (^ %^ %^ (^
remote Thread 1.1 In: asm_system_call_handler

```

调用中断后，转到 80 号中断的中断处理函数 asm_system_call_handler，此时 esp 被置为 0x7b8c。(ds 入栈后变为 0x7b88)

```

Register group: general
eax      0x7      7
ecx      0x0      0
edx      0x0      0
ebx      0x7      7
esp      0x7b48   0x7b48
ebp      0x7bbc   0x7bbc
esi      0x0      0
edi      0x0      0
eip      0x2236f  0x2236f <asm_system_call_handler+31>
eflags   0x212    [ IOPL=0 IF AF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

../src/utls/asm_utils.asm
47
48      ; ^^ &^ %^ &^
49      push edi
50      push esi
51      push edx
52      push ecx
53      push ebx
54
55      sti
>56      call dword[system_call_table + eax * 4]
57      cli
58
59      add esp, 5 * 4

remote Thread 1.1 In: asm system call handler

```

```

Register group: general
eax      0x7      7
ecx      0x0      0
edx      0x0      0
ebx      0x7      7
esp      0x7b44   0x7b44
ebp      0x7bbc   0x7bbc
esi      0x0      0
edi      0x0      0
eip      0x2055c  0x2055c <syscall_7(int, int)>
eflags   0x212    [ IOPL=0 IF AF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

../src/kernel/setup.cpp
18      // ^^ ^ (^ ^
19      SystemService systemService;
20
QEMU    int syscall_0(int first, int second, int third, int forth, int fifth)
22      {
23          printf("system call 0: %d, %d, %d, %d, %d\n",
24              first, second, third, forth, fifth);
25          return first + second + third + forth + fifth;
26      }
27
28      int syscall_7(int a, int b)
>29      {
30          if(a + b == 7)

```

remote Thread 1.1 In: syscall_7

保护现场完毕后，此时的栈为内核栈，将 5 个参数依次入栈，调用系统调用处理的函数，即 `syscall_7`，此时返回地址也会入栈，从栈顶向下依次为返回地址、各个参数，符合函数调用规则。

```
Register group: general
eax      0x7      7
ecx      0x0      0
edx      0x7      7
ebx      0x7      7
esp      0x7b48   0x7b48
ebp      0x7bbc   0x7bbc
esi      0x0      0
edi      0x0      0
eip      0x22376   0x22376 <asm_system_call_handler+38>
eflags   0x202    [ IOPL=0 IF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8

./src/utils/asm_utils.asm
47
48      ; ^^ &^ %^ &^
49      push edi
50      push esi
51      push edx
52      push ecx
53      push ebx
54
55      sti
56      call dword[system_call_table + eax * 4]
>57      cli
58
59      add esp, 5 * 4
```

函数处理完后，返回值在 `eax` 中，将 `eax` 中的返回值暂存在 `[ASM_TEMP]` 中，恢复现场，中断返回。

此时输出结果如下：

```
0x39000 233472
QEMU [Paused]
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
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 7: 19335074 wenny
1
1 pangler ( ) at ./src/utils/asm_utils.asm:57
```


TSS 的作用

在本次实验中 TSS 的作用仅在调用 80 号中断后，进入 80 号中断的中断处理函数 `asm_system_call_handler` 时栈指针 `esp` 以及段选择子被自动加载（段选择子被加载没看出来）。

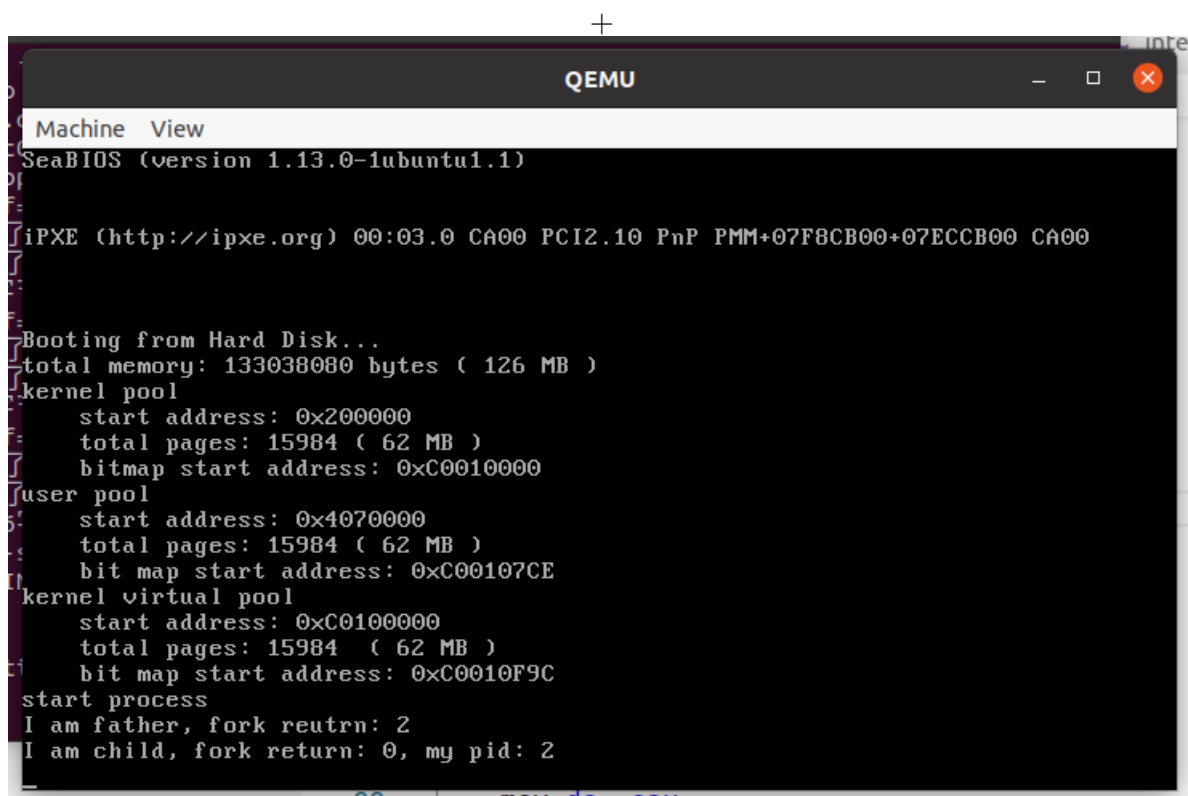
TSS 的主要作用在于保存各个特权级的栈和段选择子的信息，不同特权级的栈是相互独立的，因此在不同特权级切换时 TSS 可用于正确的恢复现场，避免了任务被错误执行。

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。

复现结果如下：



```
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 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 reutrn: 2
I am child, fork return: 0, my pid: 2
```

分析 fork 实现的基本思路

我们期望 fork 函数将运行中的进程分成两个（几乎）完全一样的进程，每个进程会从 fork 的返回点开始执行。

在第一个进程中调用 fork 函数，fork 调用 2 号系统调用，2 号系统调用已和 syscall_fork 绑定，syscall_fork 将调用 programManager.fork()：

```

1 int fork() {
2     return asm_system_call(2);
3 }
4
5 int syscall_fork() {
6     return programManager.fork();
7 }

```

在 programManager.fork() 中将真正的创建新的子进程。其基本思路为：创建子进程（通过函数 ProgramManager::executeProcess）、初始化子进程（通过函数 ProgramManager::copyProcess 将父进程状态信息复制到子进程中）。

```

1 int ProgramManager::fork()
2 {
3     bool status = interruptManager.getInterruptStatus();
4     interruptManager.disableInterrupt();
5
6     // 禁止内核线程调用
7     PCB *parent = this->running;
8     if (!parent->pageDirectoryAddress)
9     {
10         interruptManager.setInterruptStatus(status);
11         return -1;
12     }
13
14     // 创建子进程
15     int pid = executeProcess("", 0);
16     if (pid == -1)
17     {
18         interruptManager.setInterruptStatus(status);
19         return -1;
20     }
21
22     // 初始化子进程
23     PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
24     bool flag = copyProcess(parent, child);
25
26     if (!flag)
27     {
28         child->status = ProgramStatus::DEAD;
29         interruptManager.setInterruptStatus(status);
30         return -1;
31     }
32
33     interruptManager.setInterruptStatus(status);

```

```

34     return pid;
35 }

```

在这之前先判断是否为内核线程调用，若是则禁止其调用。

在创建内核线程 (ProgramManager::executeThread) 时，为内核线程申请 PCB 后并未为其创建页目录表，因此其页目录表默认为空，而在创建用户进程时 (ProgramManager::executeProcess)，创建进程的基础线程的 PCB 后为进程创建了页目录表 (ProgramManager::createProcessPageDirectory)，因此用户进程的页目录表不为空。通过页目录表的有无可判断区分内核线程和用户进程。

随后开始创建子进程，filename 为空，优先级为 0

```

1  int ProgramManager::executeProcess(const char *filename, int priority)
2  {
3      bool status = interruptManager.getInterruptStatus();
4      interruptManager.disableInterrupt();
5
6      // 在线程创建的基础上初步创建进程的PCB
7      int pid = executeThread((ThreadFunction)load_process,
8                              (void *)filename, filename, priority);
9      if (pid == -1)
10     {
11         interruptManager.setInterruptStatus(status);
12         return -1;
13     }
14
15     // 找到刚刚创建的PCB
16     PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
17
18     // 创建进程的页目录表
19     process->pageDirectoryAddress = createProcessPageDirectory();
20     //printf("%x\n", process->pageDirectoryAddress);
21
22     if (!process->pageDirectoryAddress)
23     {
24         process->status = ProgramStatus::DEAD;
25         interruptManager.setInterruptStatus(status);
26         return -1;
27     }
28
29     // 创建进程的虚拟地址池
30     bool res = createUserVirtualPool(process);
31
32     if (!res)
33     {
34         process->status = ProgramStatus::DEAD;

```

```

35         interruptManager.setInterruptStatus(status);
36         return -1;
37     }
38
39     interruptManager.setInterruptStatus(status);
40
41     return pid;
42 }

```

创建子进程的过程为：在线程创建的基础上初步创建进程的 PCB、创建进程的页目录表、创建进程的虚拟地址池。

创建子进程实际上只是创建了子进程的 PCB 并对 ProgramStartStack 进行了初始化，在子进程正式准备完毕之前，中断已被禁止，因此不会未准备好的线程 PCB 不会被换上处理器上去执行。

在 load_process 中将对 ProgramStartStack 进行初始化，指导说明很详细，故不再复述，最后将通过 asm_start_process 转到进程的起始处执行。

详细过程如下：

asm_start_process 中传入进程的 ProcessStartStack 的起始地址。

```

1  asm_start_process((int)interruptStack);

```

ProcessStartStack 的结构：

```

1  struct ProcessStartStack
2  {
3      int edi;
4      int esi;
5      int ebp;
6      int esp_dummy;
7      int ebx;
8      int edx;
9      int ecx;
10     int eax;
11
12     int gs;
13     int fs;
14     int es;
15     int ds;
16
17     int eip;
18     int cs;
19     int eflags;
20     int esp;
21     int ss;
22 };

```

以下第 4-5 行将 ProcessStartStack 的起始地址送入 esp, popad 依次恢复 8 个基本寄存器 (对应以上 3-10 行), 随后依次 pop gs、fs、es、ds (对应以上 12-15 行), 最后 iret, 将 ProcessStartStack 中的 eip 载入当前 eip 中, 随后 CPU 会自动加载 cs、eflags、esp、ss。

```
1 ; void asm_start_process(int stack);
2 asm_start_process:
3     ; jmp $
4     mov eax, dword[esp+4]
5     mov esp, eax
6     popad
7     pop gs;
8     pop fs;
9     pop es;
10    pop ds;
11
12    iret
```

至此子进程的 PCB 创建成功, 处在就绪队列的末尾。下面还将对其初始化才能等待调度。

在 copyProcess 中依次将父进程包含的资源有 0 特权级栈, PCB、虚拟地址池、页目录表、页表及其指向的物理页复制到子进程中, 并设置子进程的返回值为 0。

至此进程已创建完成, 恢复中断状态等待换上处理器上执行。

GDB 追踪

创建好子进程后, 待内核线程和父进程的时间片耗尽, 子进程即将换上处理器执行。在 asm_switch_thread 处设置断点, 切换线程栈后, 即将跳转到 asm_start_process 处。

```

Register group: general
eax      0xc0025f80      -1073586304
ecx      0x1             1
edx      0x219000        2199552
ebx      0x0             0
esp      0xc0026f30      0xc0026f30 <PCB_SET+12208>
ebp      0x0             0x0
esi      0x0             0
edi      0x0             0
eip      0xc0022eb9      0xc0022eb9 <asm_switch_thread+21>
eflags   0x286           [ IOPL=0 IF SF PF ]
cs       0x20            32
ss       0x10            16
ds       0x8             8

./src/utils/asm_utils.asm
203      mov esp, [eax] ; ^&^ &^ &^ %^ ' ^ $^ ur^&^ %^ &^ %^ ext^&^
204
205      pop esi
206      pop edi
207      pop ebx
208      pop ebp
209
210      sti
211      ret
212      ; int asm_interrupt_status();
213      asm_interrupt_status:
214      xor eax, eax
215      pushfd

remote Thread 1.1 in: asm_switch_thread
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
asm_switch_thread () at ./src/utils/asm_utils.asm:205
(gdb) s
asm_switch_thread () at ./src/utils/asm_utils.asm:206
(gdb) s
asm_switch_thread () at ./src/utils/asm_utils.asm:207
(gdb) s
asm_switch_thread () at ./src/utils/asm_utils.asm:208
(gdb) s
asm_switch_thread () at ./src/utils/asm_utils.asm:210
(gdb) s
asm_switch_thread () at ./src/utils/asm_utils.asm:211
(gdb) x/1aw 0xc0026f30
0xc0026f30 <PCB_SET+12208>: 0xc0022dc0
(gdb) x/1aw 0xc0022dc0
0xc0022dc0 <asm_start_process>: 0x424448b
(gdb)

```

跳转到到 `asm_start_process` 后查看子进程的 `ProcessStartStack`，如下所示，这与父进程在 `fork` 时的 `ProcessStartStack` 是相同的，`ProcessStartStack` 即进程的 0 级栈：

```

../src/kernel/program.cpp
361         return -1;
362     }
363
364     // ^%^ %^ %^ (%^ ^'
365     PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
366     bool flag = copyProcess(parent, child);
367
368     if (!flag)
369     {
370         child->status = ProgramStatus::DEAD;
371         interruptManager.setInterruptStatus(status);
372         return -1;
373     }

```

remote Thread 1.1 In: ProgramManager::fork

```

(gdb) s
InterruptManager::disableInterrupt (this=0xc00343a4 <interruptManager>) at ../src/kernel/interrupt.cpp:111
(gdb) s
ProgramManager::fork (this=0xc00343c0 <programManager>) at ../src/kernel/program.cpp:349
(gdb) s
(gdb) n
(gdb) n
(gdb) n
(gdb) layout regs
(gdb) n
(gdb) x/16aw 0xc0026f3c
0xc0026f3c <PCB_SET+12220>:  0x0  0x0  0x0  0x0
0xc0026f4c <PCB_SET+12236>:  0x0  0x0  0x0  0x0
0xc0026f5c <PCB_SET+12252>:  0x0  0x0  0x0  0x0
0xc0026f6c <PCB_SET+12268>:  0x0  0x0  0x0  0x0
(gdb) x/16aw 0xc0025f3c
0xc0025f3c <PCB_SET+8124>:  0x0  0x0  0x8048fac  0xc0025f5c
0xc0025f4c <PCB_SET+8140>:  0x0  0x0  0x0  0x2
0xc0025f5c <PCB_SET+8156>:  0x0  0x33  0x33  0x33
0xc0025f6c <PCB_SET+8172>:  0xc0022e6f  0x2b  0x216  0x8048f98
(gdb)

```

```

../src/utils/asm_utils.asm
36     mov cr3, eax
37     pop eax
38     ret
39     asm_start_process:
40     ;jmp $
B+>41     mov eax, dword[esp+4]
42     mov esp, eax
Visual Studio Code
43     popad
44     pop gs;
45     pop fs;
46     pop es;
47     pop ds;
48

```

remote Thread 1.1 In: asm_start_process

```

asm_switch_thread () at ../src/utils/asm_utils.asm:208
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:210
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:211
(gdb) n
Breakpoint 2, asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb) x/4aw 0xc0026f34
0xc0026f34 <PCB_SET+12212>:  0x0  0xc0026f3c  0x0  0x0
(gdb) x/16aw 0xc0026f3c
0xc0026f3c <PCB_SET+12220>:  0x0  0x0  0x8048fac  0xc0025f5c
0xc0026f4c <PCB_SET+12236>:  0x0  0x0  0x0  0x0
0xc0026f5c <PCB_SET+12252>:  0x0  0x33  0x33  0x33
0xc0026f6c <PCB_SET+12268>:  0xc0022e6f  0x2b  0x216  0x8048f98
(gdb) x/16aw 0xc0025f3c

```

在 `asm_start_process` 中依次恢复各个数据寄存器和段寄存器，最后通过 `iret` 返回到 `asm_system_call` 中：

```

(gdb) x/1aw 0xc0026f6c
0xc0026f6c <PCB_SET+12268>:  0xc0022e6f
(gdb) x/1aw 0xc0022e6f
0xc0022e6f <asm_system_call+28>:  0x595a5e5f

```

int 80H的后一条

再返回到 fork():

```
(gdb) n
(gdb) x/1aw 0x8048fb0
0x8048fb0: 0xc0020fc0
(gdb) x/1aw 0xc0020fc0
0xc0020fc0 <fork()+30>: 0xc920c483
(gdb)
```

最后回到进程:

```
./src/kernel/setup.cpp
30
31     void first_process()
32     {
33         int pid = fork();
34
35         if (pid == -1)
36         {
37             printf("can not fork\n");
38         }
39         else
40         {
41             if (pid)
42             {
remote Thread 1.1 In: first_process
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) nn
Undefined command: "nn". Try "help".
(gdb) n
(gdb) n
(gdb) nn
Undefined command: "nn". Try "help".
Undefined command: "nn". Try "help".
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) x/1aw 0x8048fb0
0x8048fb0: 0xc0020fc0
(gdb) x/1aw 0xc0020fc0
0xc0020fc0 <fork()+30>: 0xc920c483
(gdb) n
fork () at ../src/kernel/syscall.cpp:37
(gdb) n
first_process () at ../src/kernel/setup.cpp:35
(gdb)
```

这样子进程就回到了 fork() 返回点。

父进程执行完 ProgramManager::fork() 后先返回到 system_fork, 然后返回到 asm_system_call_handler, 再返回到 asm_system_call, 再返回 fork(), 最后返回进程中调用 fork() 的位置:

system_fork:

```
39     int syscall_fork() {
40         return programManager.fork();
41     }
```

asm_system_call_handler:

```
113
114         sti
115         call dword[system_call_table + eax * 4]
>116         cli
117
```

asm_system_call:

```
145
146         int 0x80
147
>148         pop edi
149         pop esi
150         pop edx
151         pop ecx
152         pop ebx
```

fork():

```
34
B+ 35         int fork() {
36             return asm_system_call(2);
>37         }
38
```

进程中调用 fork() 的位置:

```
30
31         void first_process()
32         {
33             int pid = fork();
34
>35             if (pid == -1)
```

综上可以概述为，父进程按正常过程返回，子进程被调度后从 asm_system_call 中断返回处开始执行，从而保证了每个进程会从 fork 的返回点开始执行。

fork 返回值解释

先说说父进程。

结合上述父进程的返回过程，父进程的 fork 的返回值先返回给 system_fork，然后保存在寄存器 eax 中返回给 asm_system_call_handler，在 asm_system_call_handler 中 call 了 system_fork 后得到：

```
Register group: general
eax      0x2      2      父进程中的PID
ecx      0xc0010f9c  -1073672292
edx      0x0      0
ebx      0x0      0
esp      0xc0025f28  0xc0025f28 <PCB_SET+8104>
ebp      0x8048fac  0x8048fac
esi      0x0      0
edi      0x0      0
eip      0xc0022e3d  0xc0022e3d <asm_system_call_handl
eflags   0x286     [ IOPL=0 IF SF PF ]
cs       0x20     32
ss       0x10     16
帮助     0x8       8
src/utls/asm_utils.asm
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
122      pop gs

remote Thread 1.1 In: asm system call handler
```

结合 Assignment1 所述，这个值即 `asm_system_call` 的返回值，即父进程中 `fork` 的返回值。

在子进程中，`fork` 的返回值是从 `ProcessStartStack` 中得到的。

结合前面所述，在 `asm_start_process` 中，子进程恢复完数据寄存器、段寄存器等后 `iret` 到 `asm_system_call` 中，恢复数据寄存器的 `popad` 语句将 0 载入了 `eax` 中，随后的过程和父进程一样返回。

```

Register group: general
eax 0x0 0
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xc0026f5c 0xc0026f5c <PCB_SET+12252>
ebp 0x8048fac 0x8048fac
esi 0x0 0
Ubuntu Software 0xc0022dc7 0xc0022dc7 <asm_start_process+7>
eflags 0x286 [ IOPL=0 IF SF PF ]
cs 0x20 32
ss 0x10 16
ds 0x8 8

./src/utils/asm utils.asm
36 mov cr3, eax
37 pop eax
38 ret
39 asm_start_process:
40 ;jmp $
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

remote Thread 1.1 In: asm_start_process

```

```

Register group: general
eax 0x0 0 子进程中的PID
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0x8048f98 0x8048f98
ebp 0x8048fac 0x8048fac
esi 0x0 0
edi 0x0 0
eip 0xc0022e6f 0xc0022e6f <asm_system_call
eflags 0x216 [ IOPL=0 IF AF PF ]
cs 0x2b 43
ss 0x3b 59
ds 0x33 51

./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
Visual Studio Code pop edx
pop ecx
152 pop ebx
153 pop ebp
154
155 ret

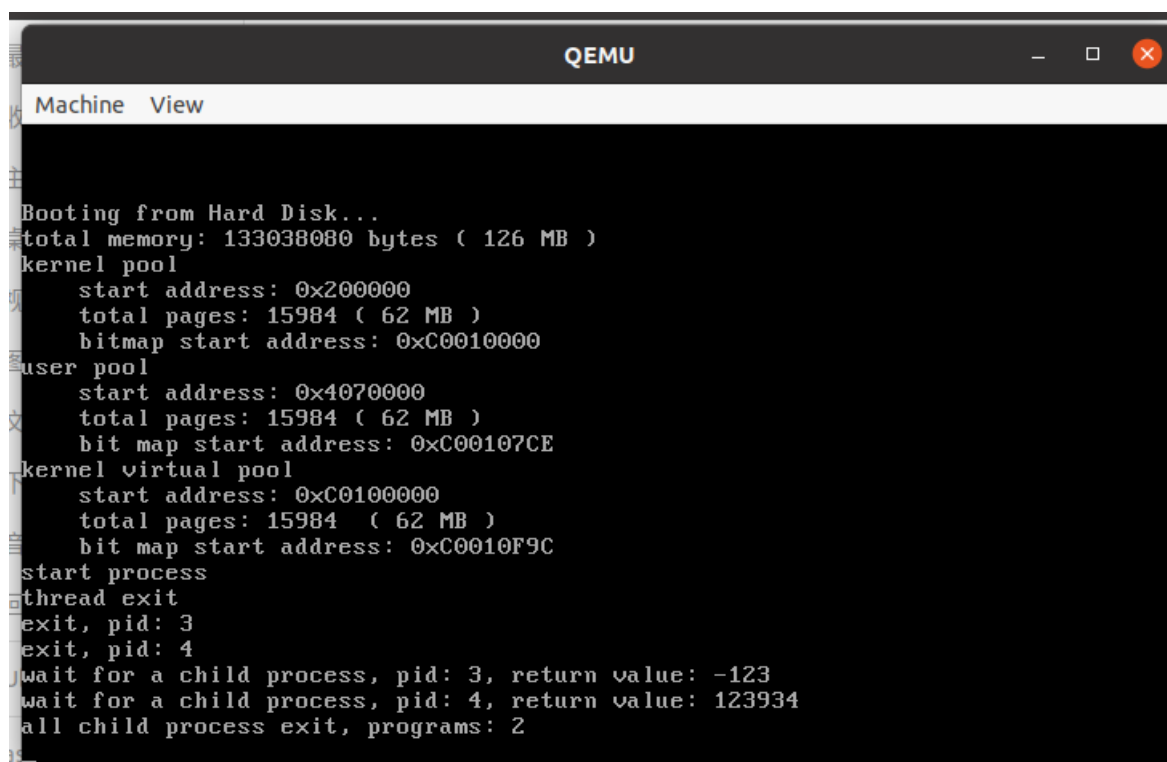
```

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。

复现结果：



```
QEMU
Machine View
Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x2000000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
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
```

分析 exit 执行过程

在 first_process 中，父进程创建的两个子进程依次退出，返回值分别为 123934 和 -123，这可以看作 exit 执行的两个实例。

```
1 void first_process()
2 {
3     int pid = fork();
4     int retval;
5 }
```

```

6     if (pid)
7     {
8         pid = fork();
9         if (pid)
10        {
11            while ((pid = wait(&retval)) != -1)
12            {
13                printf("wait for a child process, pid: %d, return value:
14                    %d\n", pid, retval);
15            }
16
17            printf("all child process exit, programs: %d\n",
18                programManager.allPrograms.size());
19
20            asm_halt();
21        }
22        else
23        {
24            uint32 tmp = 0xffffffff;
25            while (tmp)
26                --tmp;
27            printf("exit, pid: %d\n", programManager.running->pid);
28            exit(123934);
29        }
30    }
31    else
32    {
33        uint32 tmp = 0xffffffff;
34        while (tmp)
35            --tmp;
36        printf("exit, pid: %d\n", programManager.running->pid);
37        exit(-123);
38    }
39}void first_process()
40{
41    int pid = fork();
42    int retval;
43
44    if (pid)
45    {
46        pid = fork();
47        if (pid)
48        {
49            while ((pid = wait(&retval)) != -1)
50            {

```

```

49         printf("wait for a child process, pid: %d, return value:
           %d\n", pid, retval);
50     }
51
52     printf("all child process exit, programs: %d\n",
           programManager.allPrograms.size());
53
54     asm_halt();
55 }
56 else
57 {
58     uint32 tmp = 0xffffffff;
59     while (tmp)
60         --tmp;
61     printf("exit, pid: %d\n", programManager.running->pid);
62     exit(123934);
63 }
64 }
65 else
66 {
67     uint32 tmp = 0xffffffff;
68     while (tmp)
69         --tmp;
70     printf("exit, pid: %d\n", programManager.running->pid);
71     exit(-123);
72 }
73 }

```

下面将就其中一个实例具体分析。

pid=3 的子进程调用 exit 函数请求退出，返回值为-123:

```

1  exit(-123);

```

exit 会调用 3 号系统调用，3 号系统调用为 syscall_exit，syscall_exit 再调用 programManager.exit(ret):

```

1  void exit(int ret) {
2      asm_system_call(3, ret);
3  }
4
5  void syscall_exit(int ret) {
6      programManager.exit(ret);
7  }

```

关于在 ProgramManager::exit 是如何实现进程和线程的退出在指导中已有详尽叙述，故不再复述。

主进程的 `load_process` 中已在用户栈的栈顶放入 `exit` 的地址，按照函数调用规则，在结束后它将会跳转到 `exit`，过程同上，不再复述。

分析进程退出后隐式地调用 `exit` 和此时 `exit` 返回值是 0 的原因

在 `load_process` 中将用户栈依次放入了如下参数：

```
1 void load_process(const char *filename)
2 {
3     ...
4
5     interruptStack->esp = memoryManager.allocatePages(AddressPoolType::
        USER, 1);
6     if (interruptStack->esp == 0)
7     {
8         printf("can not build process!\n");
9         process->status = ProgramStatus::DEAD;
10        asm_halt();
11    }
12    interruptStack->esp += PAGE_SIZE;
13
14    // 设置进程返回地址
15    int *userStack = (int *)interruptStack->esp;
16    userStack -= 3;
17    userStack[0] = (int)exit;
18    userStack[1] = 0;
19    userStack[2] = 0;
20
21    interruptStack->esp = (int)userStack;
22
23    ...
24 }
```

按照函数调用规则，在结束后它将会跳转到 `exit` 处执行（栈顶 `userStack[0]`），它的返回地址（`userStack[1]`）为 `0x0`，事实上不会用到这个地址，因为到了 `exit` 后它出去就 `schedule` 了。`userStack[2]` 为 0，它被 CPU 认为是 `exit` 的参数，因此此时的 `exit` 返回值为 0。

分析 `wait` 执行过程

上述父进程等待两个子进程返回是 `wait` 执行的一个实例，在此对其进行分析。

父进程调用 `wait` 函数等待 2 个子进程：

```
1 ...
2     while ((pid = wait(&retval)) != -1)
```



```

3         {
4             printf("wait for a child process, pid: %d, return value:
                    %d\n", pid, retval);
5         }
6     ...

```

在所有子进程都退出之前，循环不会停止。

wait 会调用 4 号系统调用，4 号系统调用为 syscall_wait，syscall_wait 再调用 programManager.wait(ret):

```

1 int wait(int *retval) {
2     return asm_system_call(4, (int)retval);
3 }
4
5 int syscall_wait(int *retval) {
6     return programManager.wait(retval);
7 }

```

ProgramManager::wait 的实现逻辑在指导中已有详尽介绍，故不加赘述。若 wait 返回-1，则说明队列中再无子进程，否则 schedule 等待。

回收僵尸进程的方法

创建 Init 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 Init 进程。
2. 如果 Init 的子进程结束，则 Init 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。

Init 进程如下：

```

1 void Init()
2 {
3     int pid;
4     int retval;
5     while (true)
6     {
7         uint32 tmp = 0xffffffff;
8         while (tmp)
9             --tmp;
10        while((pid = wait(&retval)) != -1){
11            printf("wait for a child process, pid: %d, return value: %d\n",
                    pid, retval);
12        }
13        printf("all child process of Init exit, programs: %d\n",
                programManager.allPrograms.size());
14    }

```

```
15 }
```

它会周期性的回收它的子进程，它的子进程都是僵尸进程。

在 `exit` 中加入如下片段，若该进程的父进程已经结束则将其父进程改为 `Init`，对应的 `Pid` 为 1.:

```
1 void ProgramManager::exit(int ret)
2 {
3     ...
4     int parentPid = program->parentPid;
5     // 若父进程的PCB未被回收
6     if(PCB_SET_STATUS[parentPid]){
7         PCB *Parent = (PCB *)((int)PCB_SET + parentPid * PCB_SIZE);
8         // 若父进程已经DEAD
9         if(Parent->status == ProgramStatus::DEAD){
10             program->parentPid = 1;
11         }
12     }
13     else
14         program->parentPid = 1;
15
16     schedule();
17 }
```

最后在第一个进程中添加测试方法:

```
1 void first_process()
2 {
3     int pid = fork(); // 第一个子进程
4     int retval;
5     if (pid) // 主进程
6     {
7         while ((pid = wait(&retval)) != -1)
8         {
9             printf("wait for a child process, pid: %d, return value: %d\n", pid, retval);
10        }
11        printf("all child process of first process exit, programs: %d\n", programManager.allPrograms.size());
12        asm_halt();
13    }
14    else // 进入第一个子进程
15    {
16        pid = fork(); // 第一个子进程的子进程
17        // 释放第一个子进程
18        if(pid){
19            exit(6);
20        }
21    }
22 }
```

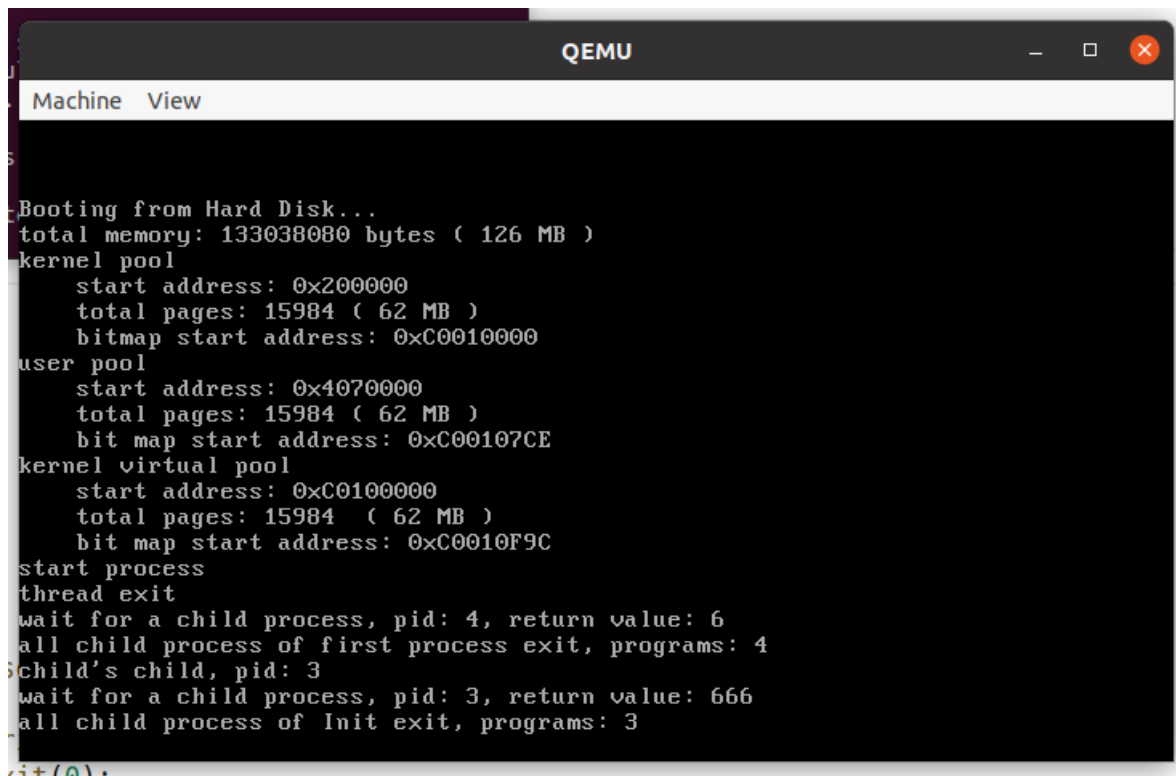
```

20     }
21     uint32 tmp = 0xffffffff;
22     while (tmp)
23         --tmp;
24     printf("child's child, pid: %d\n", programManager.running->pid);
25     exit(666);
26 }
27 }
28 void second_thread(void *arg)
29 {
30     printf("thread exit\n");
31     exit(0);
32 }
33 void first_thread(void *arg)
34 {
35     printf("start process\n");
36     programManager.executeProcess((const char *)Init, 1);
37     programManager.executeProcess((const char *)first_process, 1);
38     programManager.executeThread(second_thread, nullptr, "second", 1);
39     asm_halt();
40 }

```

创建第一个子进程，随后在子进程中创建子进程，即孙进程，随后子进程 exit，并且在父进程中回收掉子进程，这样孙进程就成了孤儿进程，exit 后成了僵尸线程。

执行结果如下：



```

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
wait for a child process, pid: 4, return value: 6
all child process of first process exit, programs: 4
child's child, pid: 3
wait for a child process, pid: 3, return value: 666
all child process of Init exit, programs: 3

```

在父进程中回收掉所有子进程后，programs 的数量为 4，包括第一个内核、Init 进程、父进程和孙进程，此时孙进程为孤儿进程。

孙进程运行完毕后，在 exit 时检测到它的父进程，即子进程已经运行结束，故将其父进程改为 Init 进程，Init 进程 wait 并回收这个孙进程。

最后回收完毕后 programs 的数量为 3，符合预期，分别为第一个内核线程、Init 进程、父进程。

总结

通过这次实验，我对系统调用、进程的创建、fork 的实现等有了初步的了解。

这次实验的收获更多的是在反复看代码、理逻辑得来的，它与前面的内核线程、内存管理、中断部分都有关联，学习的过程中少不了翻找之前的资料，在这个过程中也对之前囫圇吞枣过的内容有了进一步的理解和认识。

总而言之，此次实验使我受益匪浅。