



本科生实验报告

实验课程：_____操作系统原理实验_____

实验名称：_____实验 8 从内核态到用户态_____

专业名称：_____计算机科学与技术(超级计算方向) _____

学生姓名：_____林天皓_____

学生学号：_____18324034_____

实验地点：_____

实验成绩：_____

报告时间：_____2021. 6. 19_____

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。

以下使用 **src/3** 中的代码进行测试

为了实现用户态可以使用的系统调用，首先要实现用户态进程

1. 定义用户代码段描述符，并送入 GDT，使得 CPU 可以查找 GDT 得知用户态程序的特权级 3。

```
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;
```

2. 初始化当前 tss

将内核态 ss 写入 tss 中，将内核段描述符的特权级设置为 0 写入 GDT 中。

```
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;
}
```

初始化完成，然后在用户态进程中使用系统调用 `syscall_0`，其中系统调用中断运行流程如下：

1. 子进程执行 `asm_system_cal`，在该函数中，除了基本的保护寄存器之外，还将系统调用的参数按照中断的传参方法传入寄存器。

```
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]
```

中断的传参说明

CPU Stack (i386)

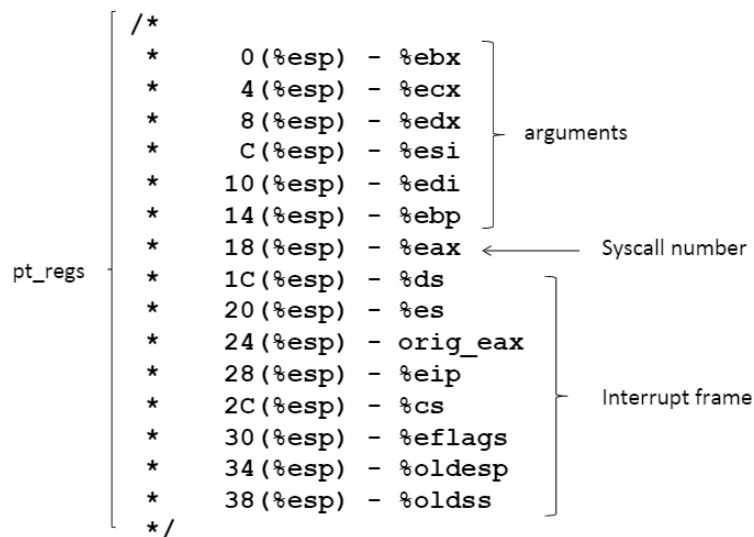


图 1-x86 中断时堆栈

然后调用 `int 80h` 执行系统调用。

对比 `ss` 的变化：初始时，内核代码段的 `ss` 为 16

```
File Edit View Search Terminal Help
../src/kernel/setup.cpp
64     systemService.initialize();
65     // 设置0号系统调用
66     systemService.setSystemCall(0, (int)syscall_0);
67
68     // 创建第一个线程
> 69     int pid = programManager.executeThread(first thread, nullptr, "
70     if (pid == -1)
71     {
72         printf("can not execute thread\n");
73         asm_halt();
74     }
75
76     ListItem *item = programManager.readyPrograms.front();

remote Thread 1.1 In: setup_kernel L69 PC: 0xc0020ba1
$2 = (void *) 0x7bc0
(gdb) ni
0xc0020b9e in setup_kernel () at ../src/kernel/setup.cpp:69
(gdb) print $ss
$3 = 16
(gdb) print $esp
$4 = (void *) 0x7be4
(gdb) █
```

图 2-内核代码段的 ss 为 16

我们查看函数运行到此处时候 tss 中的 esp 与 ss，查看程序运行到此处中保存的 esp 与 ss

```
127
128     iret
129     asm system call:
B+> 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 1.1 In: asm system_call L130 PC: 0xc0022
Continuing.

Breakpoint 1, asm_system_call () at ../src/utils/asm_utils.asm:130
(gdb) print $esp
$1 = (void *) 0x8048fd0
(gdb) print $ss
$2 = 59
(gdb)
```

图 3-查看用户程序的 esp 与 ss

此时 ss 为 59，esp 为 8048fd0，查看当前 tss 中的数据

```
../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 1.1 In: asm system call L130 PC: 0xc0022683
gs = 0, ldt = 0, trace = 0, ioMap = -1073530004}
(gdb) p/x tss
$3 = {backlink = 0x0, esp0 = 0xc0025660, ss0 = 0x10, esp1 = 0x0, ss1 = 0x0,
esp2 = 0x0, ss2 = 0x0, cr3 = 0x0, eip = 0x0, eflags = 0x0, eax = 0x0,
ecx = 0x0, edx = 0x0, ebx = 0x0, esp = 0x0, ebp = 0x0, esi = 0x0, edi = 0x0,
es = 0x0, cs = 0x0, ss = 0x0, ds = 0x0, fs = 0x0, gs = 0x0, ldt = 0x0,
trace = 0x0, ioMap = 0xc0033b6c}
(gdb) █
```

图 4-查看 tss 中的 esp 与 ss

执行 int 80 后，进入后文中描述的 asm_system_call_handler。

```
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 L89 PC: 0xc0022683
$1 = (void *) 0x8048fd0
(gdb) print $ss
$2 = 59
(gdb) si
asm_system_call_handler () at ../src/utils/asm_utils.asm:89
(gdb) print $esp
$3 = (void *) 0xc0025688 <PCB SET+8168>
(gdb) print $ss
$4 = 16
(gdb) █
```

图 5 - esp 与 ss 被从 tss 中载入，完成特权级切换

我们发现 CPU 已经帮助我们完成了 esp 和 ss 的转变，此时已经进入了内核段。

2.int 80h 根据中断向量描述符表寄存器中获取 IDT 的地址，然后根据中断号即 (80h) 查找到我们设置的中断函数地址 asm_system_call_handler，根据中断向量描述符判断当前的特权级是否能访问该中断（因为目前的特权级为 3，描述符中设置的 DPL 也为 3，因此允许访问），跳转到 asm_system_call_handle。由于发生中断跳转，通过 GDT 与 LDT 判断目标地址位于特权级为 0 的内核段中，因此储存在特权级 0 的 TSS 中的栈地址和 ss 会被加载

查看栈中的数据，此时栈上的数据与调用表中对应

```
asm_system_call_handler:
87     push ds
88     push es
89     push fs
90     push gs
91     pushad
92
93     push eax
94
95     ; 栈段会从tss中自动加载
96
97     mov eax, DATA_SELECTOR
98     mov ds, eax
99     mov es, eax
100
```

remote Thread 1.1 In: asm_system_call_handler L89 PC: 0xc0022648
ecx = 0x0, edx = 0x0, ebx = 0x0, esp = 0x0, ebp = 0x0, esi = 0x0, edi = 0x0,
es = 0x0, cs = 0x0, ss = 0x0, ds = 0x0, fs = 0x0, gs = 0x0, ldt = 0x0,
trace = 0x0, ioMap = 0xc0033b6c}

(gdb) si
asm_system_call_handler () at ../src/utils/asm_utils.asm:89
(gdb) p/16x \$esp
Item count other than 1 is meaningless in "print" command.
(gdb) x/16a \$esp

0xc0025648 <PCB_SET+8168>:	0x33	0xc002269f	0x2b	0x212
0xc0025658 <PCB_SET+8184>:	0x8048fb8	oldesp	0x3b	0x0
0xc0025668 <PCB_SET+8200>:	0x0	0x0	0x0	0x0
0xc0025678 <PCB_SET+8216>:	0x0	0x0	0x0	0x0

(gdb) x/16d \$esp

0xc0025648 <PCB_SET+8168>:	51	-1073600865	43	530
0xc0025658 <PCB_SET+8184>:	134516664	59	oldss	0
0xc0025668 <PCB_SET+8200>:	0	0	0	0
0xc0025678 <PCB_SET+8216>:	0	0	0	0

(gdb)

图 6-查看 call 函数前的堆栈

3.进一步跳转到中断向量表中的函数，在这里为 syscall_0

```
24     int syscall_0(int first, int second, int third, int forth, int fifth)
25     {
> 26         printf("system call 0: %d, %d, %d, %d, %d\n",
27             first, second, third, forth, fifth);
28         return first + second + third + forth + fifth;
29     }
30
```

图 7-正式运行 syscall_0

4.运行结束后，函数返回

在 iret 之前，ss 寄存器为 16，esp 为 0xc002564c，查看栈中内容

```
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

remote Thread 1.1 In: asm_system_call_handler          L126  PC: 0xc002267d
(gdb) print $esp
$7 = (void *) 0xc00255f8 <PCB_SET+8088>
(gdb) print $pc
$8 = (void (*)(void)) 0xc0020a2a <syscall_0(int, int, int, int, int)+6>
(gdb) si
(gdb) ni
asm_system_call_handler () at ../src/utils/asm_utils.asm:116
asm_system_call_handler () at ../src/utils/asm_utils.asm:120
asm_system_call_handler () at ../src/utils/asm_utils.asm:122
asm_system_call_handler () at ../src/utils/asm_utils.asm:123
asm_system_call_handler () at ../src/utils/asm_utils.asm:124
asm_system_call_handler () at ../src/utils/asm_utils.asm:125
asm_system_call_handler () at ../src/utils/asm_utils.asm:126
(gdb) print $ss
$9 = 16
(gdb) print $esp
$10 = (void *) 0xc002564c <PCB_SET+8172>
(gdb) x/8a $esp
0xc002564c <PCB_SET+8172>:  0xc002269f  0x2b  0x212  0x8048fb8
0xc002565c <PCB_SET+8188>:  0x3b  0x0  0x0  0x0
(gdb)
```

PC

oldesp

oldss

图 7-iret 前的函数堆栈

指令 iret 返回后

ss 段与 esp 被从栈中恢复，重新回到特权级 3

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

remote Thread 1.1 In: asm_system_call          L148  PC: 0xc002269f
(gdb) print ss
No symbol "ss" in current context.
(gdb) print $ss
$16 = 59
(gdb) print $esp
$17 = (void *) 0x8048fb8
(gdb) print $pc
$18 = (void (*)()) 0xc002269f <asm_system_call+28>
(gdb) 
```

图 7-iret 后 ss 与 esp 回到用户态程序

这样就完成了一次系统调用。

TSS 的作用是什么？

我们使用 tss 的作用是存储特权级为 0 的内核线程的 ss 和 esp，使得调用 int 80h 的时候能够从特权级 3 进入特权级 0，载入储存在 tss 中的 ss 和 esp 进入内核线程栈，运行系统调用函数。

综上，我们根据系统调用的流程，描述了中断调用时候的特权级比较，int 80h 进入时候发生的特权级转换，然后运行系统调用函数。运行完毕后，使用 iret 指令使得特权级从 0 返回 3 继续运行用户程序。

Assignment 2 线程的实现

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`。

Fork 的实现分为以下几个部分。

1. 创建子进程

```
int pid = executeProcess("", 0);
```

该 `pid` 的获取进一步来自

```
int pid = executeThread((ThreadFunction)load_process,  
                        (void *)filename, filename, priority);
```

其中给 `pid` 赋值的是

```
thread->pid = ((int)thread - (int)PCB_SET) / PCB_SIZE;
```

同时父进程也是在这个过程中获得子进程的 `pid` 来返回。

2. 子进程复制栈空间与数据

```
bool flag = copyProcess(parent, child);
```

其中对 `eax` 的赋值为

```
childp->eax = 0;
```

这使得子进程的函数返回时，`pid=0`;

其中还包括子进程创建申请内存与页目录，将父进程的页目录表与页表拷贝进入子进程的页表。使得相同的虚拟地址可以对应不同的物理地址，而且数据相同。

```
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
}

```

下面运行测试，测试代码为 src/5

```

34
B+> 35 int fork() {
36     return asm_system_call(2);
37 }
38
39 int syscall_fork() {
40     return programManager.fork();
41 }
42

remote Thread 1.1 In: fork
0x0000ffff in ?? ()
(gdb) b fork
Breakpoint 1 at 0xc00209f6: fork. (2 locations)
(gdb) c
Continuing.

Breakpoint 1, fork () at ../src/kernel/syscall.cpp:35
(gdb)

```

Booting from Hard Disk...
total memory: 133038080 b
kernel pool
start address: 0x2000
total pages: 15984 (
bitmap start address:
user pool
start address: 0x4070
total pages: 15984 (
bit map start address
kernel virtual pool
start address: 0xC010
total pages: 15984 (
bit map start address
start process

图 8-调用 fork 测试

调用 fork，然后调用系统调用，eax 为 2。调用 int 80 切换到特权级 0。

```
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 1.1 In: asm system call handler      L114  PC: 0xc0022ef5
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)
```

图 9-准备调用 syscall_fork 函数

进入 syscall_fork 函数。

```
> 39      int syscall_fork() {
40          return programManager.fork();
41      }
42
43      void exit(int ret) {
44          asm_system_call(3, ret);
45      }
46

remote Thread 1.1 In: syscall fork      L39  PC: 0xc002113d
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) si
syscall_fork () at ../src/kernel/syscall.cpp:39
(gdb)
```

图 10-准备调用 ProgramManager:: fork 函数

父进程的 pid 返回值为 3

```
384         interruptManager.setInterruptStatus(status);
> 385         return pid;
386     }
387
388     bool ProgramManager::copyProcess(PCB *parent, PCB *child)
389     {
390         // 复制PCB
391         ProcessStartStack *childpps = (ProcessStartStack *)((int)child

remote Thread 1.1 In: ProgramManager::fork L385 PC: 0xc0020afc
    this=0xc00344e4 <interruptManager>) at ../src/kernel/interrupt.cpp:114
0xc0020a09 in ProgramManager::fork (this=0xc0034500 <programManager>)
    at ../src/kernel/program.cpp:356
Value returned is $1 = true
(gdb) ni
(gdb) print pid
$2 = 3
(gdb)
```

图 11 -父进程调用 ProgramManager:: fork 结束，查看返回 pid

然后父进程逐级返回，到 first_thread 中输出子进程 pid。

```
42         if (pid)
43         {
44             printf("I am father pid %d\n",pid);
> 45             asm halt();
46         }
47         else
48         {
49             printf("I am child, exit\n");
50         }
51     }

remote Thread 1.1 In: first process L45 PC: 0x
printf (fmt=0xc00230ca "I am father pid %d\n") at ../src/kernel/stdio.c
(gdb) finish
Run till exit from #0  0xc00218a0 in printf (
    fmt=0xc00230ca "I am father pid %d\n") at ../src/kernel/stdio.cpp:
0xc0021211 in first_process () at ../src/kernel/setup.cpp:44
Value returned is $3 = 18
(gdb) ni
(gdb)
```

图 12 -父进程逐级返回，输出 pid 3

父进程的工作到此结束了，下面我们设置断点到进程调度中，查看接下来子进程的运行状态。

```
../src/utls/asm_utils.asm
195     push ebx
196     push edi
197     push esi
198
199     mov eax, [esp + 5 * 4]
200     mov [eax], esp ; 保存当前栈指针到PCB中，以便日后
201
202     mov eax, [esp + 6 * 4]
203     mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
204
> 205     pop esi
206     pop edi
207     pop ebx
208     pop ebp
209

remote Thread 1.1 In: asm_switch_thread L205 PC: 0xc0022f74
Continuing.

Breakpoint 2, asm_switch_thread () at ../src/utls/asm_utils.asm:194
(gdb) si
(gdb) print $esp
$4 = (void *) 0xc0026008 <PCB_SET+8008>
(gdb) si
asm_switch_thread () at ../src/utls/asm_utils.asm:205
(gdb) print $esp
$5 = (void *) 0xc0027060 <PCB_SET+12192>
(gdb)
```

图 12 -调度程序调度到了 pid=2 的进程

```
53
> 54 void second_thread(void *arg) {
55     printf("thread exit\n");
56     exit(0);
57 }
58
59 void first_thread(void *arg)
60 {
61
62     printf("start process\n");

remote Thread 1.1 In: second_thread L54 PC: 0xc002122e

Breakpoint 2, asm_switch_thread () at ../src/utls/asm_utils.asm:194
(gdb) si
(gdb) print $esp
$4 = (void *) 0xc0026008 <PCB_SET+8008>
(gdb) si
asm_switch_thread () at ../src/utls/asm_utils.asm:205
(gdb) print $esp
$5 = (void *) 0xc0027060 <PCB_SET+12192>
(gdb) x/16g $esp
```

图 13 - pid=2 的进程为 second_thread

目前的运行栈已经切换到了 pid 为 2 的线程中，为 second_thread，还没有到我们 fork 出来的进程，因此我们在切换到下一次进程调度。

```
199      mov eax, [esp + 5 * 4]
200      mov [eax], esp ; 保存当前栈指针到PCB中, 以便日后
201
202      mov eax, [esp + 6 * 4]
203      mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
204
> 205      pop esi
206      pop edi
207      pop ebx
208      pop ebp
209
210      sti

remote Thread 1.1 In: asm_switch_thread L205 PC: 0xc0022f74
0xc0027070 <PCB_SET+12208>: 0xc002122e 0xc002041c 0x0 0x0
0xc0027080 <PCB_SET+12224>: 0x0 0x0 0x0 0x0
0xc0027090 <PCB_SET+12240>: 0x0 0x0 0x0 0x0
(gdb) ni
asm_switch_thread () at ../src/utils/asm_utils.asm:206
asm_switch_thread () at ../src/utils/asm_utils.asm:207
asm_switch_thread () at ../src/utils/asm_utils.asm:208
asm_switch_thread () at ../src/utils/asm_utils.asm:210
asm_switch_thread () at ../src/utils/asm_utils.asm:211
second_thread (arg=0x0) at ../src/kernel/setup.cpp:54
(gdb) c
Continuing.

Breakpoint 2, asm_switch_thread () at ../src/utils/asm_utils.asm:194
(gdb) si
(gdb) print $esp
$6 = (void *) 0xc0026f18 <PCB_SET+11864>
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:205
(gdb) print $esp
$7 = (void *) 0xc0024fb4 <PCB_SET+3828>
(gdb)
```

图 14 -调度程序调度到了 pid=0 的进程

很遗憾，这一次切换到了内核线程上，pid 为 0，再进行一次切换

```
199      mov eax, [esp + 5 * 4]
200      mov [eax], esp ; 保存当前栈指针到PCB中, 以便日后
201
202      mov eax, [esp + 6 * 4]
203      mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
204
> 205      pop esi
206      pop edi
207      pop ebx
208      pop ebp
209
210      sti

remote Thread 1.1 In: asm_switch_thread L205 PC: 0xc0022f74
0xc0025038 <PCB_SET+3960>: 0x0 0x0 0xc0025070 0xc0025058
0xc0025048 <PCB_SET+3976>: 0x0 0xc00250ec 0xc00260c0 0x2
0xc0025058 <PCB_SET+3992>: 0xc002306f 0x20 0x282 0xc00212a4
(gdb) si
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:244
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:245
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:246
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:247
asm_time_interrupt_handler () at ../src/utils/asm_utils.asm:248
asm_halt () at ../src/utils/asm_utils.asm:364
(gdb) c
Continuing.

Breakpoint 2, asm_switch_thread () at ../src/utils/asm_utils.asm:194
(gdb) si
(gdb) print $esp
$9 = (void *) 0xc0024fb4 <PCB_SET+3828>
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:205
(gdb) print $esp
$10 = (void *) 0xc0028060 <PCB_SET+16288>
(gdb)
```

图 15 -调度程序调度到了 pid=3 的进程

第三次切换，终于到了我们 pid 为 3 的被 fork 出的进程，下面我们跟踪该进程第一次被执行的流程。

```
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
49     iret

remote Thread 1.1 In: asm_start_process L41 PC: 0xc0022e80
(gdb) c
Continuing.

Breakpoint 2, asm_switch_thread () at ../src/utils/asm_utils.asm:194
(gdb) si
(gdb) print $esp
$9 = (void *) 0xc0024fb4 <PCB_SET+3828>
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:205
(gdb) print $esp
$10 = (void *) 0xc0028060 <PCB_SET+16288>
(gdb) si
asm_switch_thread () at ../src/utils/asm_utils.asm:206
asm_switch_thread () at ../src/utils/asm_utils.asm:207
asm_switch_thread () at ../src/utils/asm_utils.asm:208
asm_switch_thread () at ../src/utils/asm_utils.asm:210
asm_switch_thread () at ../src/utils/asm_utils.asm:211
(gdb) print $eax
$11 = -1073581888
(gdb) si
asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb)
```

图 16-子进程首先执行 asm_start_process

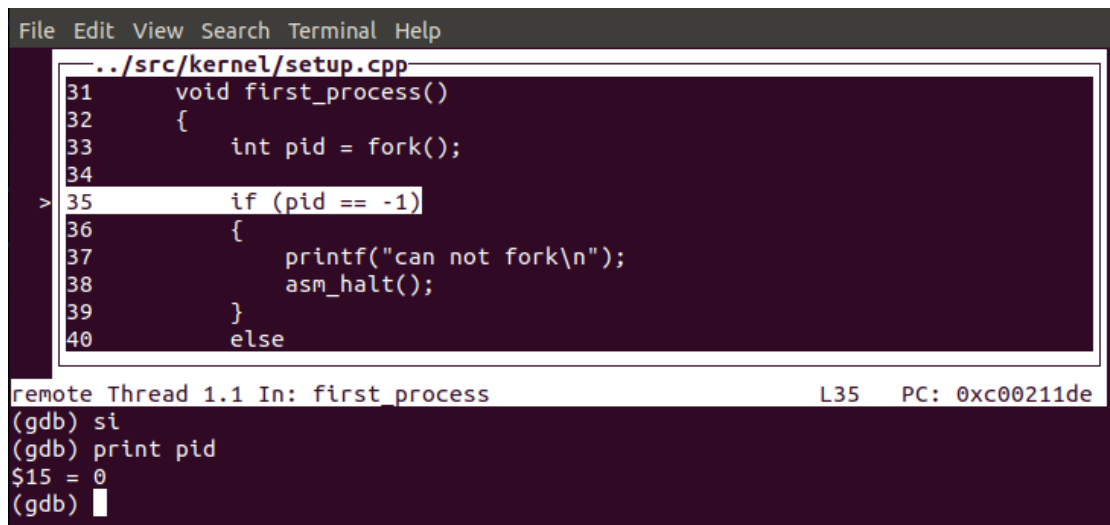
新进程首次执行了 asm_start_process 函数，为了将 interruptStack 中的内容加载到寄存器中。

```
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
49     iret

remote Thread 1.1 In: asm_start_process L47 PC: 0xc0022e8c
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xc00280a8 0xc00280a8 <PCB_SET+16360>
ebp      0x8048fa0 0x8048fa0
esi      0x0      0
edi      0x0      0
eip      0xc0022e8c 0xc0022e8c <asm_start_process+12>
eflags   0x282    [ IOPL=0 IF SF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x33     51
fs       0x33     51
gs       0x0      0
```

图 16-子进程载入 interruptStack

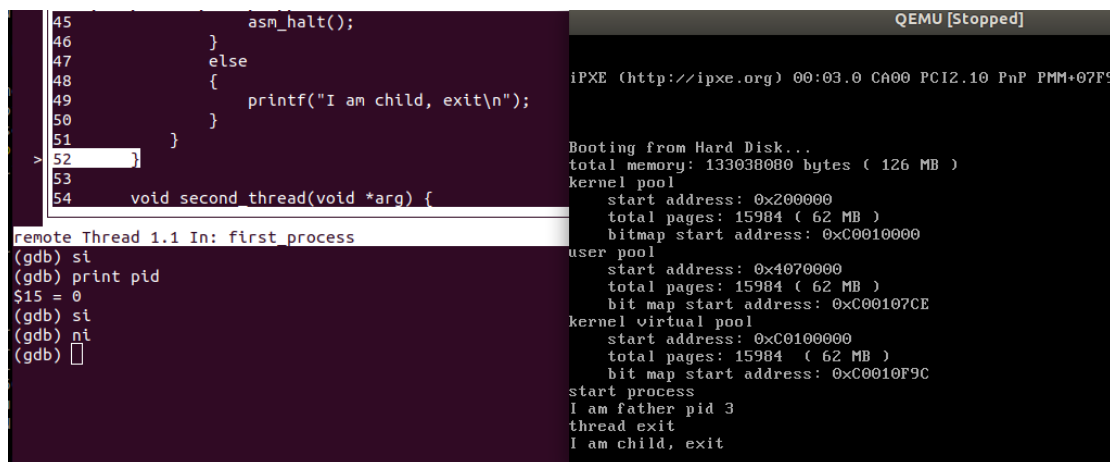
运行到此处，已经将 interruptStack 中的内容加载到了寄存器中，其中就包含了 pid 的返回值 0。接下和父进程一致，经过层层返回之后来到了 first_process 中，且 pid 获得的返回值为 0。



```
File Edit View Search Terminal Help
../src/kernel/setup.cpp
31 void first_process()
32 {
33     int pid = fork();
34
> 35     if (pid == -1)
36     {
37         printf("can not fork\n");
38         asm_halt();
39     }
40     else
    remote Thread 1.1 In: first_process L35 PC: 0xc00211de
(gdb) si
(gdb) print pid
$15 = 0
(gdb) 
```

图 17-子进程 pid 返回值为 0

完整运行结果如下



```
45     asm_halt();
46 }
47 else
48 {
49     printf("I am child, exit\n");
50 }
> 52 }
53
54 void second_thread(void *arg) {
    QEMU [Stopped]
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F9
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 pid 3
thread exit
I am child, exit
remote Thread 1.1 In: first_process
(gdb) si
(gdb) print pid
$15 = 0
(gdb) si
(gdb) ni
(gdb) 
```

图 18 - 全程序运行流程

到此处，fork 出来的子进程也完成了代码的执行。

综上，根据代码完成了一次 fork 的流程

Assignment 3 线程调度切换的秘密

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。

1.exit 调用执行过程

`exit` 调用与 `fork` 同属系统调用,从发起调用到由内核执行 `exit` 函数过程类似。

都是通过 `asm_system_call` 调用参数 (`exit` 为 3) 然后执行 `int 80h`，查中断描述符表找到函数入口，开始执行 `ProgramManager::exit` 函数。

根据代码，该函数有三个部分

1.标记 PCB 状态为 DEAD

```
PCB *program = this->running;
program->retValue = ret;
program->status = ProgramStatus::DEAD;
```

使得 `schedule` 的过程中可以识别到 **DEAD** 的进程，释放其 PCB。

2.释放内存和页表与页目录表

先释放内存，再释放页表内存，最后释放页目录表内存，在释放前需要判断是否存在，释放物理内存完毕释放虚拟内存。

```
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.re
sources.bitmap, bitmapPages);
}

```

3.调度其他进程 (PCB 的释放也在 schedule 中)

```
schedule();
```

其中 PCB 的释放在 schedule 中。

```

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

```

前面略过，直接到 schedule 过程

```
../src/kernel/program.cpp
89
90     return thread->pid;
91 }
92
93 void ProgramManager::schedule()
> 94 {
95     bool status = interruptManager.getInterruptStatus();
96     interruptManager.disableInterrupt();
97
98     if (readyPrograms.size() == 0)
99     {
100         interruptManager.setInterruptStatus(status);
101         return;
102     }
103 }

remote Thread 1.1 In: ProgramManager::schedule L94 PC: 0xc00202c2
asm_system_call_handler () at ../src/utils/asm_utils.asm:114
syscall_exit (ret=0) at ../src/kernel/syscall.cpp:47

Breakpoint 1, ProgramManager::exit (this=0xc0034500 <programManager>, ret=0)
at ../src/kernel/program.cpp:515
(gdb) ni
(gdb) until
(gdb) si
(gdb) ni
(gdb) si
ProgramManager::schedule (this=0xc0034500 <programManager>) at ../src/kernel/program.cpp:94
(gdb) █
```

图 19-exit 最后进入 schedule

当前进程为 DEAD，进入了 releasePCB，释放了 PCB 空间

```
../src/kernel/program.cpp
109 {
110     running->status = ProgramStatus::READY;
111     running->ticks = running->priority * 10;
112     readyPrograms.push_back(&(running->tagInGeneralList));
113 }
114 else if (running->status == ProgramStatus::DEAD)
115 {
> 116     releasePCB(running);
117 }
118
119 ListItem *item = readyPrograms.front();
120 PCB *next = ListItem2PCB(item, tagInGeneralList);
121 PCB *cur = running;
122 }

remote Thread 1.1 In: ProgramManager::schedule L116 PC: 0xc0020382
asm_system_call_handler () at ../src/utils/asm_utils.asm:114
syscall_exit (ret=0) at ../src/kernel/syscall.cpp:47

Breakpoint 1, ProgramManager::exit (this=0xc0034500 <programManager>, ret=0)
at ../src/kernel/program.cpp:515
(gdb) ni
(gdb) until
(gdb) si
(gdb) ni
(gdb) si
ProgramManager::schedule (this=0xc0034500 <programManager>) at ../src/kernel/program.cpp:94
(gdb) ni
(gdb) si
(gdb) █
```

图 19-准备切换到下一个进程 pid=0

下面已经获取到了下一个要被调度的进程，为 pid=0 的进程，当前 pid=3 的进程被切换。

```
119     ListItem *item = readyPrograms.front();
120     PCB *next = ListItem2PCB(item, tagInGeneralList);
121     PCB *cur = running;
122     next->status = ProgramStatus::RUNNING;
> 123     running = next;
124     readyPrograms.pop_front();
125
126     //printf("schedule: %x %x\n", cur, next);
127
remote Thread 1.1 In: ProgramManager::schedule L123 PC: 0xc00203c8
(gdb) nl
(gdb) until
(gdb) si
(gdb) nl
(gdb) si
ProgramManager::schedule (this=0xc0034500 <programManager>) at ../src/kernel/program.cpp:94
(gdb) nl
(gdb) si
(gdb) si
(gdb) nl
(gdb) print next
$1 = (PCB *) 0xc00250c0 <PCB_SET+4096>
(gdb) print cur
$2 = (PCB *) 0xc00270c0 <PCB_SET+12288>
(gdb)
```

图 20 - 切换到下一个进程 pid=0

2.函数能隐式实现 exit 的原因是在 load_process 中设置了线程启动的栈顶上为 exit 的函数地址，且参数为 0。相关代码如下。

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

下面运行

```
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 pid 3
thread exit
I am child, exit
```

图 20 – exit 运行效果

exit 的调用过程结束。

3.wait 调用执行过程

开始过程到进入中断过程与其他系统调用过程类似，略去，直接 ProgramManager::wait 开始。Wait 过程为一个死循环，包括查找当前已经 DEAD 的进程

```
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;
}
```

然后查看当前 DEAD 的进程是不是该进程的子进程。如果是，则获取子进程的相关资源后释放该子进程的 PCB。

```
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;
}
```

值得注意的是，子进程 PCB 已经不再由 schedule 回收。

```
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);
    }
}

```

通过判断 `pageDirectoryAddress` 是否存在判断该线程是否为子进程。这是因为内核创建的线程共享了内核的页表,只有 `fork` 的进程才会用自己独立的页表。

可以通过 `pageDirectoryAddress` 判断该线程是否有父进程。

下面运行测试: 使用 `src/6` 代码

前面的 `fork` 调用过程省略,从已经有两个被 `fork` 的进程结束开始。

```

File Edit View Search Terminal Help
./src/kernel/program.cpp
570 ListItem *item;
571 bool interrupt, flag;
572
573 while (true)
574 {
575     interrupt = interruptManager.getInterruptStatus();
576     interruptManager.disableInterrupt();
577
578     item = this->allPrograms.head.next;
579
580     // 查找子进程
581     flag = true;
582     while (item)
583     {
584         child = ListItem2PCB(item, tagInAllList);
585         if (child->parentPid == this->running->pid)
586         {
587             flag = false;
588             if (child->status == ProgramStatus::DEAD)
589             {
590                 break;
591             }
592         }
593         item = item->next;
594     }
595 }

remote Thread 1.1 In: ProgramManager::wait L575 PC: 0xc00210cf
Run till exit from #0 0xc00210cf in ProgramManager::wait (
this=0xc00347c0 <programManager>, retval=0x8048fd8)
at ./src/kernel/program.cpp:578
Breakpoint 2, ProgramManager::wait (this=0xc00347c0 <programManager>,
retval=0x8048fd8) at ./src/kernel/program.cpp:575
(gdb)

iPXE (http://ipxe.org) 00:03.0 CA00
Booting from Hard Disk...
total memory: 133030000 bytes ( 126
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: 0xC001007
kernel virtual pool
start address: 0xC0100000
total pages: 15984 ( 62 MB )
bit map start address: 0xC00100F
start process
thread exit
exit outer 1, pid: 3
exit inner, pid: 4

```

图 21 – 两个进程已经结束

然后父进程的 `wait` 中找到了这个已经返回的子进程。

```

File Edit View Search Terminal Help
./src/kernel/program.cpp
583 {
584     child = ListItem2PCB(item, tagInAllList);
585     if (child->parentPid == this->running->pid)
586     {
587         flag = false;
588         if (child->status == ProgramStatus::DEAD)
589         {
590             break;
591         }
592     }
593     item = item->next;
594 }
595

remote Thread 1.1 In: ProgramManager::wait L590 PC: 0xc0021112
(gdb) ni
(gdb) print child
$1 = (PCB *) 0xc0025380 <PCB_SET+4096>
(gdb) ni
(gdb) print child
$2 = (PCB *) 0xc0027380 <PCB_SET+12288>
(gdb)

```

图 22 – 父进程找到了已经返回的子进程 `pid=3`

最后读取子进程的返回值与清空子进程 PCB，并返回，完成 wait 函数。

```
../src/kernel/program.cpp
598         if (retval)
599         {
600             *retval = child->retValue;
601         }
602
603         int pid = child->pid;
604         releasePCB(child);
605         interruptManager.setInterruptStatus(interrupt);
> 606         return pid;
607     }
608     else
609     {
610         if (flag) // 子进程已经返回

remote Thread 1.1 In: ProgramManager::wait L606 PC: 0xc0021159
$1 = (PCB *) 0xc0025380 <PCB_SET+4096>
(gdb) ni
(gdb) print child
$2 = (PCB *) 0xc0027380 <PCB_SET+12288>
(gdb) si
(gdb) ni
(gdb)
```

图 23－父进程找到了清空子进程 PCB

经过一系列的返回之后，输出语句

```
40     {
41         while ((pid = wait(&retval)) != -1)
42         {
> 43             printf("wait for a child process, pid: %d\n", pid);
44         }
45
46         printf("all child process exit, programs: %d\n", programs);
47
48         asm halt();

remote Thread 1.1 In: first process L43 PC: 0xc00211391
asm_system_call_handler () at ../src/utlis/asm_utils.asm:124
asm_system_call_handler () at ../src/utlis/asm_utils.asm:125
asm_system_call_handler () at ../src/utlis/asm_utils.asm:126
asm_system_call () at ../src/utlis/asm_utils.asm:148
0xc00212dc in wait (retval=0x8048fd8) at ../src/kernel/syscall.cpp:52
0xc002136c in first_process () at ../src/kernel/setup.cpp:41
(gdb)

Bootimg 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 outer 1, pid: 3
exit inner, pid: 4
wait for a child process, pid: 3, return value: -123
wait for a child process, pid: 4, return value: 123934
```

图 24－父进程输出 wait 成功语句

然后回收 pid=4 的进程

```
40     {
41         while ((pid = wait(&retval)) != -1)
42         {
> 43             printf("wait for a child process, pid: %d, return v
44         }
45
46         printf("all child process exit, programs: %d\n", progra
47
48         asm halt();

remote Thread 1.1 In: first process L43 PC: 0xc00211391
asm_system_call_handler () at ../src/utlis/asm_utils.asm:124
asm_system_call_handler () at ../src/utlis/asm_utils.asm:125
asm_system_call_handler () at ../src/utlis/asm_utils.asm:126
asm_system_call () at ../src/utlis/asm_utils.asm:148
xc00212dc in wait (retval=0x8048fd8) at ../src/kernel/syscall.cpp:52
xc002136c in first_process () at ../src/kernel/setup.cpp:41
(gdb)

en
Bootimg 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 outer 1, pid: 3
exit inner, pid: 4
wait for a child process, pid: 3, return value: -123
wait for a child process, pid: 4, return value: 123934
```

图 25－父进程找到了已经返回的子进程 pid=4

```
../src/kernel/program.cpp
605         interruptManager.setInterruptStatus(interrupt);
606         return pid;
607     }
608     else
609     {
610         if (flag) // 子进程已经返回
611         {
612             > 613             interruptManager.setInterruptStatus(interrupt);
614                 return -1;
615             }
616             else // 存在子进程，但子进程的状态不是DE
617             {
remote Thread 1.1 In: ProgramManager::wait L613 PC: 0xc0021164
Breakpoint 2, ProgramManager::wait (this=0xc00347c0 <programManager>,
retval=0x8048fd8) at ../src/kernel/program.cpp:575
(gdb) ni
(gdb) print item
$4 = (ListItem *) 0x0
(gdb) ni
(gdb)
```

图 26 – 父进程找不到子进程了，返回-1

此时已到达 allprograms 的尾部，item 为 null，代表此时该父进程已经没有未结束的子进程。Return -1。运行结束

```
41         while ((pid = wait(&retval)) != -1)
42         {
43             printf("wait for a child process, pid: %d, return v
44         }
45     }
> 46     printf("all child process exit, programs: %d\n", progra
47
48     asm halt();
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 outer 1, pid: 3
exit inner, 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
remote Thread 1.1 In: first process L46 PC: 0xc00213b4
asm_system_call_handler () at ../src/utlis/asm_utils.asm:124
asm_system_call_handler () at ../src/utlis/asm_utils.asm:125
asm_system_call_handler () at ../src/utlis/asm_utils.asm:126
asm_system_call () at ../src/utlis/asm_utils.asm:148
0xc00212dc in wait (retval=0x8048fd8) at ../src/kernel/syscall.cpp:52
0xc002136c in first_process () at ../src/kernel/setup.cpp:41
```

图 27 – 父进程 wait 完整执行过程

Wait 调用的运行就结束了。

3. 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

我们优化的目标是从状态被标记为 Dead 到开始回收的时间缩短，linux 系统中实现的方法是子进程结束后通过信号发送给父进程，然后父进程接收到信号进行处理。但是我们没有实现进程间通信的机制。因此我们只能假定父进程已经使用 wait 在等待子进程结束，我们只需要在调度器上做手脚，在子进程结束后尽

快的调度到父进程上，使得父进程可以尽快的处理后事。

我们修改 ProgramManager::schedule 函数，使得可以尽快切换到父进程上，

下文中标记的部分为主要修改部分。

```
int normal=1;
ListItem *item;
PCB *next,*cur;
if (running->status == ProgramStatus::RUNNING)
{
    running->status = ProgramStatus::READY;
    running->ticks = running->priority * 10;
    readyPrograms.push_back(&(running->tagInGeneralList));
}
else if (running->status == ProgramStatus::DEAD)
{
    // 回收线程，子进程留到父进程回收
    // printf("running pid=%d son=%d\n",running->pid,running->pageDirectory
Address);
    if(!running->pageDirectoryAddress) {
        releasePCB(running);
    }
    else{
        //尽快切换到父进程
        printf("found dead pid=%d\ntry to find parent pid=%d\n",running->pi
d,running->parentPid);
        ListItem *item = readyPrograms.front();
        PCB *temp = ListItem2PCB(item, tagInGeneralList);
        //寻找父进程
        while(temp->pid != running->parentPid && item){
            item=item->next;
            temp = ListItem2PCB(item, tagInGeneralList);
        }
        if(!item){
            printf("parent pid=%d no found\n down to normal switch",running
->parentPid);
        }
        else{
            printf("switch to parent pid=%d\n",running->parentPid);
            next=temp;//设置 next 进程
            normal=0;
        }
    }
}
```

```

    }
    if(normal){
        item= readyPrograms.front();
        next = ListItem2PCB(item, tagInGeneralList);
    }
    cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.erase(item);
    activateProgramPage(next);
    asm_switch_thread(cur, next);

```

然后我们运行测试

```

    bit map start address: 0xC0010F9C
start process
thread exit
exit outer 1, pid: 3
found dead pid=3
try to find parent pid=1
switch to parent pid=1
wait for a child process, pid: 3, return value: -123
exit inner, pid: 4
found dead pid=4
try to find parent pid=1
switch to parent pid=1
wait for a child process, pid: 4, return value: 123934
all child process exit, programs: 2

```

图 28 – 子进程结束后立刻切换到父进程

目前子进程结束后调度会优先切到父进程了。

对比以前的运行结果，现在僵尸进程被回收的优先级更高了。

```

start address: 0xC0010F9C
total pages: 15984 ( 62 MB )
bit map start address: 0xC0010F9C
start process
thread exit
exit outer 1, pid: 3
exit inner, 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

```

图 29 – 调度算法修改前的运行结果

综上，在 assignment3 中对 fork 和 wait 的流程进行了解释，修改了调度算法使得父进程能够更快的回收僵尸进程的 PCB 空间。

实验感想

本次实验是实现系统调用 `fork`, `wait`, `exit` 的一次实验，并实验了一种解决僵尸进程的方法。总的来说，这次的实验没有什么需要编写的代码，但是理解 x86 保护模式的中断，tss 的切换，cpu 对于特权级别的管理实实在在非常复杂，同时也运用了很多的 trick，tss 的初始化，对 `int` 执行之前的栈构造，对进程创建过程中的 `ProcessStartStack` 的构造的安排，对 GDT 的设置，对函数返回地址在栈中的设置，每一项 trivial convention 都需要花费大量的时间查阅资料，intel 在 datasheet 中也写的不想让人读懂，追踪寄存器和内存的变化也十分困难，最后也只是明白了 x86 怎样用近乎万能的栈实现各种 trick 这样泛用性不强的知识，好像又有一点不值得，实验也没有什么编写代码的需求，总的来说体验略差。

在体会实现的过程中，我仍然没有理解 intel 设计芯片时的心路历程，在三十多年前使用这么复杂的方法。对比 riscv，有专门的 `ra` 寄存器保存程序的返回地址，不用在栈中寻找，在实现不同的特权级中，每个特权级都有属于自己的状态寄存器，不需要通过栈或者 tss 实现切换。在切换特权级时候，允许操作系统控制堆栈，而不是向 x86 中的 `int` 有一些由 cpu 执行的固定的压栈。

参考资料

<https://blog.csdn.net/xzongyuan/article/details/19490817>

https://wiki.osdev.org/System_Calls

https://wiki.osdev.org/Task_State_Segment

<https://stackoverflow.com/questions/22444526/x86-64-linux-syscall-arguments>

https://wiki.osdev.org/Context_Switching

<http://www.ce.uniroma2.it/~pellegrini/didattica/2018/aosv/7.System-Calls-Management.pdf>

<https://blog.csdn.net/wrx1721267632/article/details/52056910>

<https://cloud.tencent.com/developer/article/1492374>

https://wiki.osdev.org/Getting_to_Ring_3

<https://blog.csdn.net/sdulibh/article/details/82852900>

https://blog.csdn.net/weixin_43068469/article/details/88982098

<https://www.felixcloutier.com/x86/ltr>

https://blog.csdn.net/weixin_44489823/article/details/103260332

<https://zhuanlan.zhihu.com/p/143002272>

https://blog.csdn.net/weixin_33705053/article/details/85852062