

# 操作系统实验八实验报告

## 个人信息

- 姓名：李浩辉
- 学号：21307018

## 实验要求

- 编写系统调用并调用分析其中 TSS 的作用和栈变化情况
- 实现 fork 函数，分析 fork 实现的基本思路。分析子进程的跳转地址、数据寄存器和段寄存器的变化，比较与父进程的异同。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
- 分析 exit 以及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；分析 wait ；实现回收僵尸进程的有效方法

## 实验过程

### Assignment 1

#### 实验要求

编写系统调用并调用分析其中 TSS 的作用和栈变化情况

#### 实验原理

##### 特权级

- RPL, Request Privilege Level, 段选择子的低2位所表示的值
- CPL, Current Privilege Level, 在CS寄存器中的段选择子的RPL, CPL标识了CPU当前的特权级
- DPL, Descriptor Privilege Level, 位于每一个段描述符中

##### 访问规则

- 对于数据段和栈段，访问前进行特权级检查，要求  $DPL \geq \max(CPL, RPL)$
- 一致性代码段，要求  $CPL \geq DPL$ ;
- 非一致性代码段，要求  $CPL = DPL \geq RPL$

##### 特权转移

- 从低特权级向高特权级转移。通过**中断、调用**等方式来实现
- 从高特权级向低特权级转移。通过**中断返回和调用返回**来实现，即 `iret` , `ret` 和 `return` 等指令

## TSS

在这里，TSS的作用仅限于为CPU提供0特权级栈所在的地址和段选择子，即CPU只会用到TSS中的`esp0`和`SS0`

## 系统调用实现

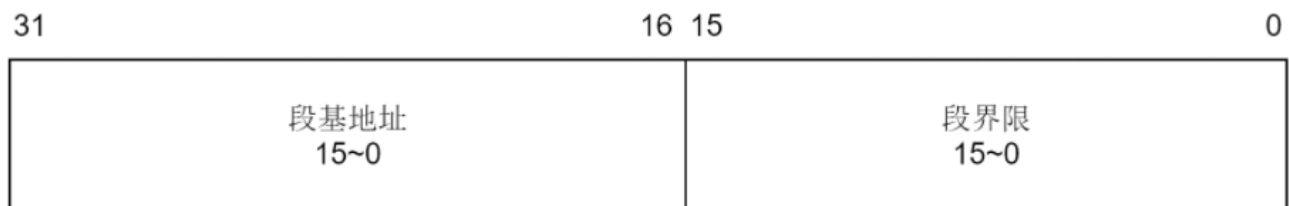
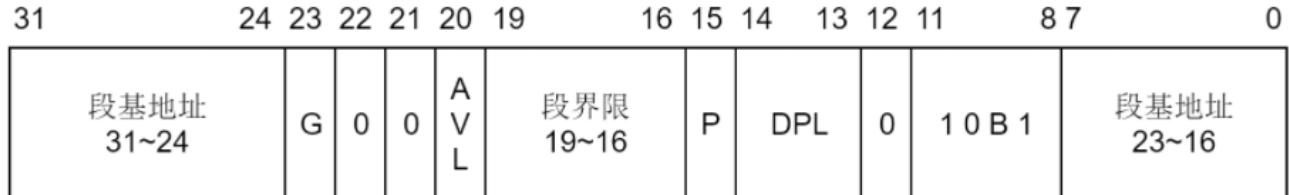
- 通过**中断**实现，设计好系统调用函数在中断表上，让中断描述符特权级为3。  
这时候，特权级为3的用户进程可以通过特权级为0的段选择子，调用特权级3的系统调用函数。
- **参数的传递是通过寄存器来获取**，因为特权级切换会导致栈切换。

## 用户进程

首先提前将用户代码段，数据段和栈段初始化好，放入GDT。

段描述符以及段选择子优先级均设计为3。

初始化TSS，**但只对 `TSS::ss0` 进行复制**，**`TSS::esp0` 会在进程切换时更新**。TSS有类似段描述符的TSS描述符，它的段选择子放在TR寄存器，需要用 `ltr` 完成设置。



## 进程创建

- **创建进程的PCB**，这里将会像创建内核线程基础上创建，运行的函数会是 `load_process`
- **初始化进程的页目录表**，内目录表将会放置在内核地址池，并且768到1022将会和内核页目录表一样，最后一个1023将会指向页目录表本身。
- **初始化进程的虚拟地址池**，将用户进程的可分配的虚拟地址的定义在 `USER_VADDR_START` 和 3GB之间，这是仿照linux的做法。

```
#define USER_VADDR_START 0x8048000
```

## 高特权级到低特权级交接过程

定义好 `ProcessStartStack` 表示启动进程之前栈放入的内容，并将他放置在PCB顶部。

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

    int gs;
    int fs;
    int es;
    int ds;
    int eip;
    int cs;
    int eflags;
    int esp;
    int ss;
};
#endif
```

通过 `load_process` 和 `asm_start_process` 配合完成进入进程的寄存器的初始化，然后一个 `iret` 刚好会跳转到提前设置好的进程函数，整个过程非常精妙！

在x86架构中，IRET指令用于从中断处理程序返回到被中断的程序。当执行IRET指令时，处理器会从堆栈中弹出三个值：**EIP（返回地址）**、**CS（代码段选择子）**和**EFLAGS（标志寄存器）**。这些值一起用于恢复被中断的程序的执行状态。

三个寄存器的内容设置如此

```
interruptStack->cs = programManager.USER_CODE_SELECTOR;    // 用户模式平坦模式

interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1); // IOPL, IF = 1 开中断,
MBS = 1 默认
```

```
interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1);
```

**进程的调度**只需要在原先的线程调度的基础上加

- 切换页目录表
- 更新TSS中的特权级0的栈

```
void ProgramManager::activateProgramPage(PCB *program)
{
    int paddr = PAGE_DIRECTORY;
    if (program->pageDirectoryAddress)
    {
        tss.esp0 = (int)program + PAGE_SIZE;
        paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);
    }
    asm_update_cr3(paddr);
}
```

## 实验过程

printf 会操作显存，在特权级3下我们无法操作，因此我们可以通过系统调用实现特权级转移进而操作显存。

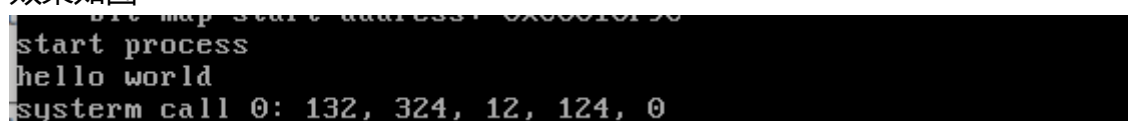
在 setup.cpp 设置好系统调用

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

进程中调用之

```
void first_process()
{
    char* str="hello world\n";
    //printf("Hello World!");
    asm_system_call(1, (int)str);
    asm_system_call(0, 132, 324, 12, 124);
    asm_halt();
}
```

效果如图



```
start process
hello world
system call 0: 132, 324, 12, 124, 0
```

gdb 调试，在我们调用系统调用即将进入中断（特权级由低向高）

可以看到栈指针 `esp` 和栈段选择子 `ss` 都改变了，这是因为我们提前设置好的TSS给的结果

```
../src/utils/asm_utils.asm
```

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

```
remote Thread 1 In: asm_system_call
```

```
L146 PC: 0xc002271d
```

```
ebx      0xc00228b7      -1073600329
esp      0x8048fa8       0x8048fa8
ebp      0x8048fbc       0x8048fbc
esi      0x0             0
```

```
---Type <return> to continue, or q <return> to quit---
```

```
Quit
```

```
(gdb) info registers
```

```
eax      0x1             1
ecx      0x0             0
edx      0x0             0
ebx      0xc00228b7      -1073600329
esp      0x8048fa8       0x8048fa8
ebp      0x8048fbc       0x8048fbc
esi      0x0             0
edi      0x0             0
eip      0xc002271d      0xc002271d <asm_system_call+26>
eflags   0x212          [ AF IF ]
cs       0x2b            43
ss       0x3b            59
ds       0x33            51
es       0x33            51
fs       0x33            51
gs       0x0             0
```

```
(gdb) █
```

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

../src/utils/asm_utils.asm
83      pop ebp
84
85      ret
86      ; int asm_system_call_handler();
87      asm_system_call_handler:
88      push ds
> 89      push es
90      push fs
91      push gs
92      pushad
93
94      push eax
95
96      ; 栈段会从tss中自动加载
97

remote Thread 1 In: asm_system_call_handler L89 PC: 0xc00226c8
ds      0x33      51
es      0x33      51
fs      0x33      51
gs      0x0       0
(gdb) s
asm_system_call_handler () at ../src/utils/asm_utils.asm:89
(gdb) info registers
eax      0x1       1
ecx      0x0       0
edx      0x0       0
ebx      0xc00228b7 -1073600329
esp      0xc0025708 0xc0025708 <PCB_SET+8168>
ebp      0x8048fbc  0x8048fbc
esi      0x0       0
edi      0x0       0
eip      0xc00226c8 0xc00226c8 <asm_system_call_handler+1>
eflags   0x12      [ AF ]
cs       0x20      32
ss       0x10      16
ds       0x33      51
es       0x33      51
fs       0x33      51
gs       0x0       0
(gdb)
```

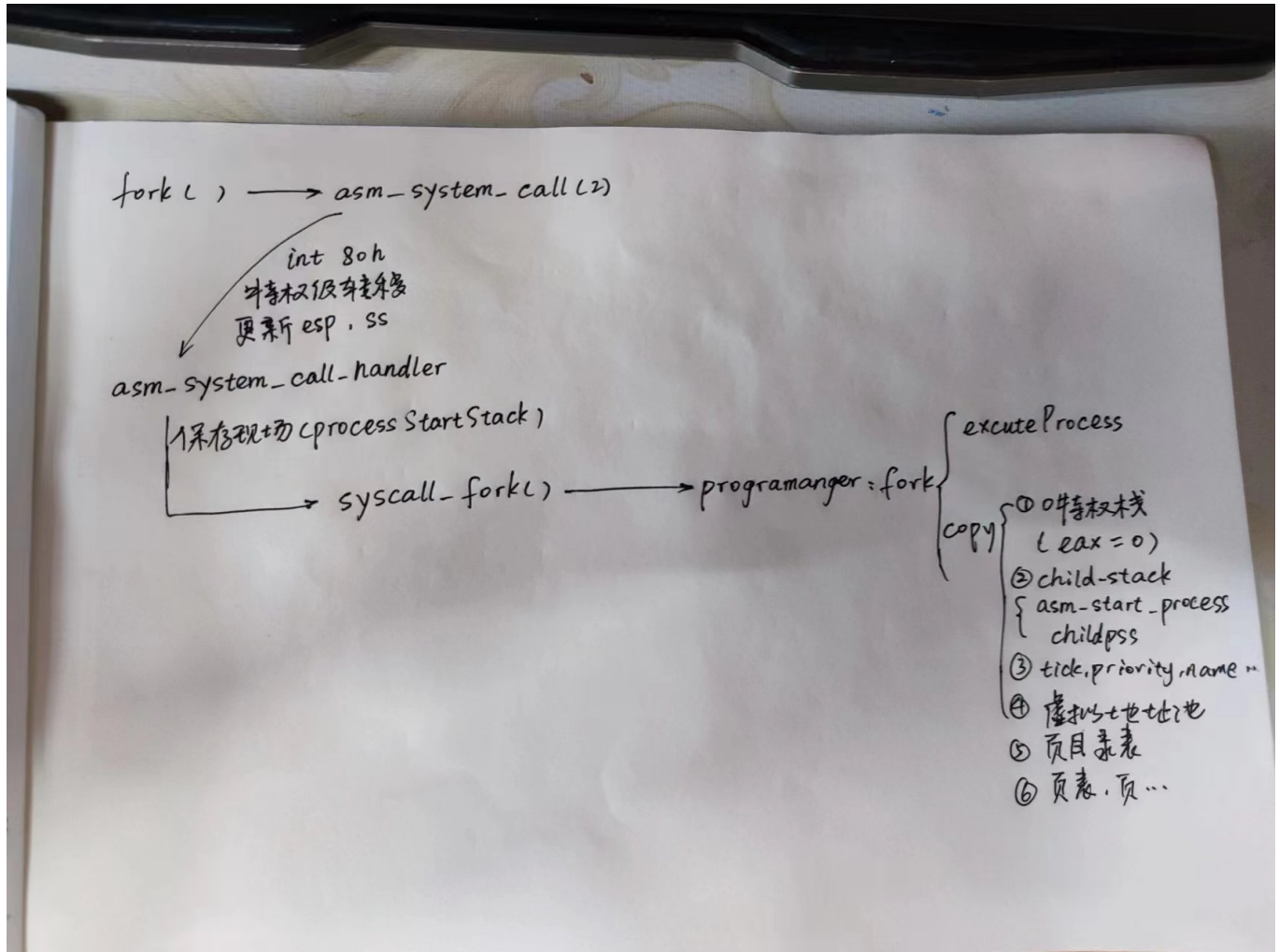
## Assignment 2

### 实验要求

实现 `fork` 函数，分析 `fork` 实现的基本思路。分析子进程的跳转地址、数据寄存器和段寄存器的变化，比较与父进程的异同。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

## 实验原理

涉及许多函数跳转，在此做一个简单的导图避免混乱



着重解决几点：

- `fork` 函数的思路
- 跳转过程中的寄存器安排是怎样的
- 返回值问题，是怎么实现父进程与子进程的不同

通过上面的导图，我们知道到达 `asm_sysmem_call_handler` 以后，我们的栈已经切换到0特权级栈，并且我们即将保存现场即保存 `ProcessStartStack` 到我们的栈上。

一路跳转达到 `ProgramManager::fork()`

在这里需要完成

- `executeProcess` 即创建子进程，上面的assignment已经知道我们其实是创建了一个线程套进程
- 复制父进程的内容

接下来分析复制 `ProgramManager::copyProcess(PCB *parent, PCB *child)` 的过程



- 复制0特权级栈，但是 `childpss->eax` 设置为0
- 准备执行 `asm_switch_thread` 的栈的内容，返回地址设置为 `asm_start_address`，参数则设置为 `childpss`
- 复制好PCB中的一些属性以及创建虚拟地址池
- 复制页目录表和页表，以及分配物理页

代码很长，不做过大具体分析。仅分析其中关键的部分

**为什么 `childpss->eax = 0` 并且 `asm_switch_thread` 切换函数设置为 `asm_start_address` 和对参数设置为 `childpss` ?**

```
// 设置子进程的返回值为0
childpss->eax = 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
```

这样设置到调度到子进程的时候，`asm_switch_thread` 将会将其切换到 `asm_start_process`，接着用传递的参数即复制好的0特权级栈（除了寄存器 `eax` 内容为0）恢复寄存器并返回到 `asm_system_call`（因为保护现场工作是由 `asm_sytem_call_handler` 进行的，它是由 `asm_system_call` 跳转进来，栈上存着返回去的地址），接着便可以回到使用 `fork` 函数后的下一句函数指令。同时因为提前设置了 `eax=0` 返回值就是0。

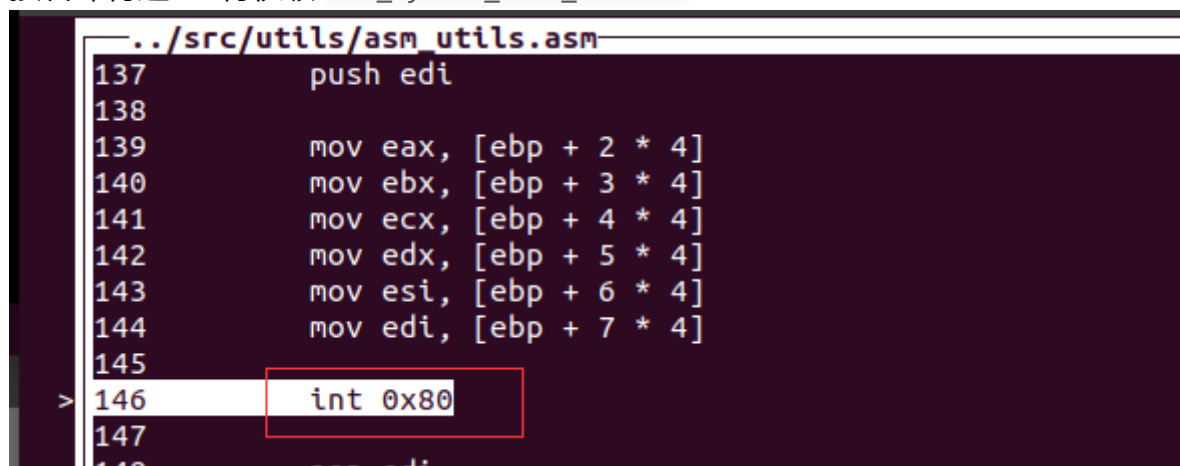
## 实验过程

### 父进程

我们用gdb调试来查看具体步骤

前面进入 `fork` 跳转到 `asm_system_call` 不做赘述

接着即将进入0特权级 `asm_system_call_handler`



```

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

```

在 `asm_system_call_handler` 中我们将会保护现场，形成 `ProcessStartStack` 放在栈上。同时 `call` 进入 `syscall_fork`，继续进入到 `programManager.fork()`

```
38  
39     int syscall_fork() {  
40         return programManager.fork();  
41     }  
42
```

经过 `excuteProcess` 和 `copyProcess` 以后就返回, 返回子进程的 `pid`

```

356 // 创建子进程
357 int pid = executeProcess("", 0);
358 if (pid == -1)
359     return -1;
360
361 // 初始化子进程
362 PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
363 bool flag = copyProcess(parent, child);
364
365 return 0;
366
367 }

```

## 子进程

我们首先设置断点在 `ProgramManager::schedule()` 接着 `continue` 到调度子进程

即将调用 `asm_switch_thread`

```

123         running = next;
124         readyPrograms.pop_front();
125
126         //printf("schedule: %x %x\n", cur, next);
127
128         activateProgramPage(next);
129
130         asm_switch_thread(cur, next);
131
132         interruptManager.setInterruptStatus(status);
133     }
134
135     void program_exit()

```

我们的PCB栈已经设置好了跳转的函数 `asm_start_process` 以及对应的参数，这个参数就是已经复制好的特权级0的栈（除了 `eax` 改变了）

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

-./src/utils/asm_utils.asm-
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
210     sti
> 211     ret
212     ; int asm_interrupt_status();

remote Thread 1 In: asm_switch_thread L211 PC: 0xc0022d19

Breakpoint 1, ProgramManager::schedule (this=0xc0034220 <programManager>)
  at ../src/kernel/program.cpp:94
(gdb) n
(gdb) s
asm_switch_thread () at ../src/utils/asm_utils.asm:194
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:205
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) info registers esp
esp             0xc0026d90             0xc0026d90 <PCB_SET+12208>
(gdb) x/xw 0xc0026d90
0xc0026d90 <PCB_SET+12208>: 0xc0022c20
(gdb) info symbol 0xc0022c20
asm_start_process in section .text
(gdb)
```

不出意外, 我们来到了 `asm_start_process`, 这时候我们的 `esp` 还是指向特权级0的栈, 栈保存着现场, 我们开始恢复寄存器, 同时准备 `iret` 到 `asm_system_call`。

**这时候已经和实行完 `fork` 函数返回到 `asm_system_call_handler` 的父进程除了 `eax` 完全一样了!** 也因此我们就可以回答返回值为什么两个进程可以不一样了。

- 对于父进程, 是执行完 `fork` 得到的子进程的 `pid`, 然后返回
- 对于子进程, 是通过复制父进程返回时候的状态得到的, 而它的返回值已经提前设定为0了, 并且他并没有跑 `int 80h` 以后的函数, 而是复制状态生成的!

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

../src/utils/asm_utils.asm
37     pop eax
38     ret
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
50
51     ; void asm_ltr(int tr)

remote Thread 1 In: asm_start_process L49 PC: 0xc0022c2d
esp      0xc0026d90      0xc0026d90 <PCB_SET+12208>
(gdb) x/xw 0xc0026d90
0xc0026d90 <PCB_SET+12208>:      0xc0022c20
(gdb) info symbol 0xc0022c20
asm_start_process in section .text
(gdb) s
asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb) n
asm_start_process () at ../src/utils/asm_utils.asm:44
asm_start_process () at ../src/utils/asm_utils.asm:45
asm_start_process () at ../src/utils/asm_utils.asm:46
asm_start_process () at ../src/utils/asm_utils.asm:47
asm_start_process () at ../src/utils/asm_utils.asm:49
(gdb) info registers esp
esp      0xc0026dcc      0xc0026dcc <PCB_SET+12268>
(gdb) x/xw 0xc0026dcc
0xc0026dcc <PCB_SET+12268>:      0xc0022ccf
(gdb) info symbol 0xc0022ccf
asm_system_call + 28 in section .text
(gdb)
```

## Assignment 3

### 实验要求

分析 `exit` 以及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；分析 `wait` ；实现回收僵尸进程的有效方法

### 实验原理

`exit` 三步：

- 标记PCB状态为 `DEAD` 并放入返回值

- 如果PCB标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池bitmap的空间。否则不做处理
- 立即执行线程/进程调度

为什么了能够**隐式调用** `exit` 我们提前在每个特权级3的栈上面放了函数，返回地址和参数

```
// 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;
```

wait 函数逻辑：

- 遍历所有PCB
- 查找子进程，如果是 `dead` 则 `releasePCB` 并返回，如果未 `dead` 则线程调度
- 未找到则返回-1

在 `schedule` 回收线程

```
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);
    }
}
```

## 实验过程

说实话，代码逻辑比较简单，我们直接用gdb来说明情况。

先来 `exit`，同样系统调用，流程都会经过 `asm_system_call` 通过 `int 80h` 进入内核态 `asm_system_call_handler` 然后调用函数 `syscall_exit(int ret)`，接着进入 `programManager.exit(ret)`。

```
47 void syscall_exit(int ret) {
> 48     programManager.exit(ret);
49 }
```

设置返回值和状态 `dead`

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

如果是线程，不做处理，否则开始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。

代码比较简单，只是简单判断内容有效位，然后清理释放即可，不做赘述。

最后直接线程调度

```
// 第三步，立即执行线程/进程调度。
```

```
schedule();
```

我们直接来看 `wait` 同样也是系统调用一路来到 `ProgramManager::wait(int *retval)`

函数体有一个大循环，用来未找到调度后也能回来继续查找

先找子进程，如果找到 `dead` 的子进程就可以直接开始释放返回 `pid` 了

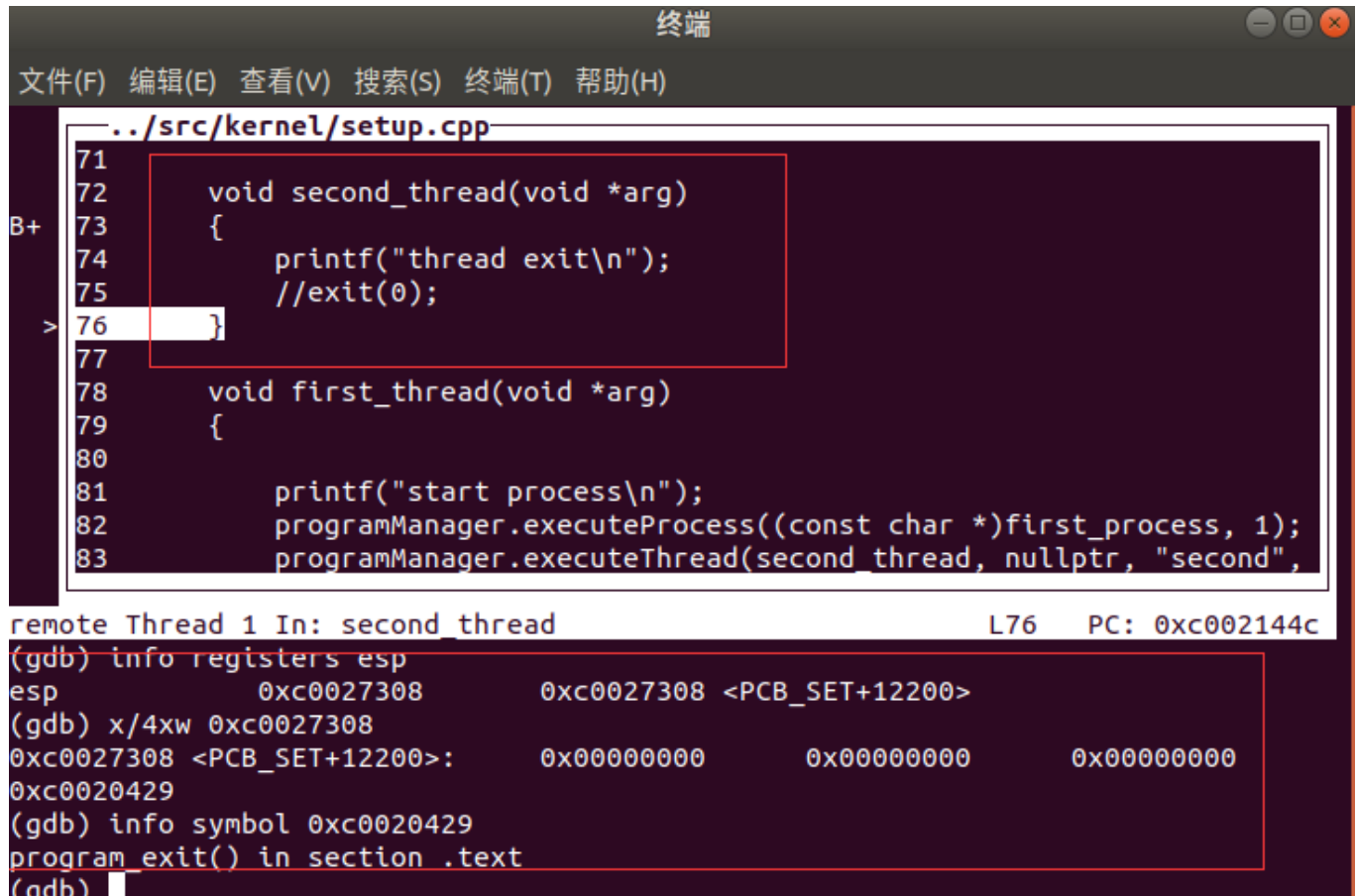
```
// 查找子进程
flag = true;
while (item)
{
    child = ListItem2PCB(item, tagInAllList);
    if (child->parentPid == this->running->pid)
    {
        flag = false;
        if (child->status == ProgramStatus::DEAD)
        {
            break;
        }
    }
    item = item->next;
}
}
```

如果找不到 `dead` 的子进程，如果有子进程则调度，否则返回-1

```
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();
    }
}
```

接下来我们来探究一下如何做到隐式调用 `exit`

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



```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/setup.cpp
71
72 void second_thread(void *arg)
73 {
74     printf("thread exit\n");
75     //exit(0);
76 }
77
78 void first_thread(void *arg)
79 {
80
81     printf("start process\n");
82     programManager.executeProcess((const char *)first_process, 1);
83     programManager.executeThread(second_thread, nullptr, "second",
remote Thread 1 In: second thread L76 PC: 0xc002144c
(gdb) info registers esp
esp 0xc0027308 0xc0027308 <PCB_SET+12200>
(gdb) x/4xw 0xc0027308
0xc0027308 <PCB_SET+12200>: 0x00000000 0x00000000 0x00000000 0x00000000
0xc0020429
(gdb) info symbol 0xc0020429
program_exit() in section .text
(gdb)
```

函数结束，将会清除栈上存放临时变量，然后到 `exit`

## 实现回收僵尸进程的有效方法

原来的方法不仅没有回收僵尸进程，也没有回收没有 `parentPid==0` 的父进程

在 `schedule` 后面加上这段代码即可

即在判断一个进程如果已经 `dead`：

- 如果是线程则直接回收
- 如果是进程则找一下它的父进程如果是0或者已经没有了则直接回收

```
// 回收线程，子进程留到父进程回收
if(!running->pageDirectoryAddress) {
    releasePCB(running);
}
```

```

else if(running->parentPid==0)
{
    releasePCB(running);
}
else
{
    bool flag=1;
    int father_pid=running->parentPid;
    ListItem *item=this->allPrograms.head.next;
    while(item){
        PCB * father = ListItem2PCB(item, tagInAllList);
        if(father->pid==father_pid){
            flag=0;
            break;
        }
        item=item->next;
    }
    if(flag)
        releasePCB(running);
}

```

## 验证

```

void first_process()
{
    int pid = fork();
    if(pid){
        printf("son's pid is %d\n",pid);
        printf("father exit\n");
        exit(1);
    }
    else{
        printf("son exit\n");
        exit(2);
    }
}

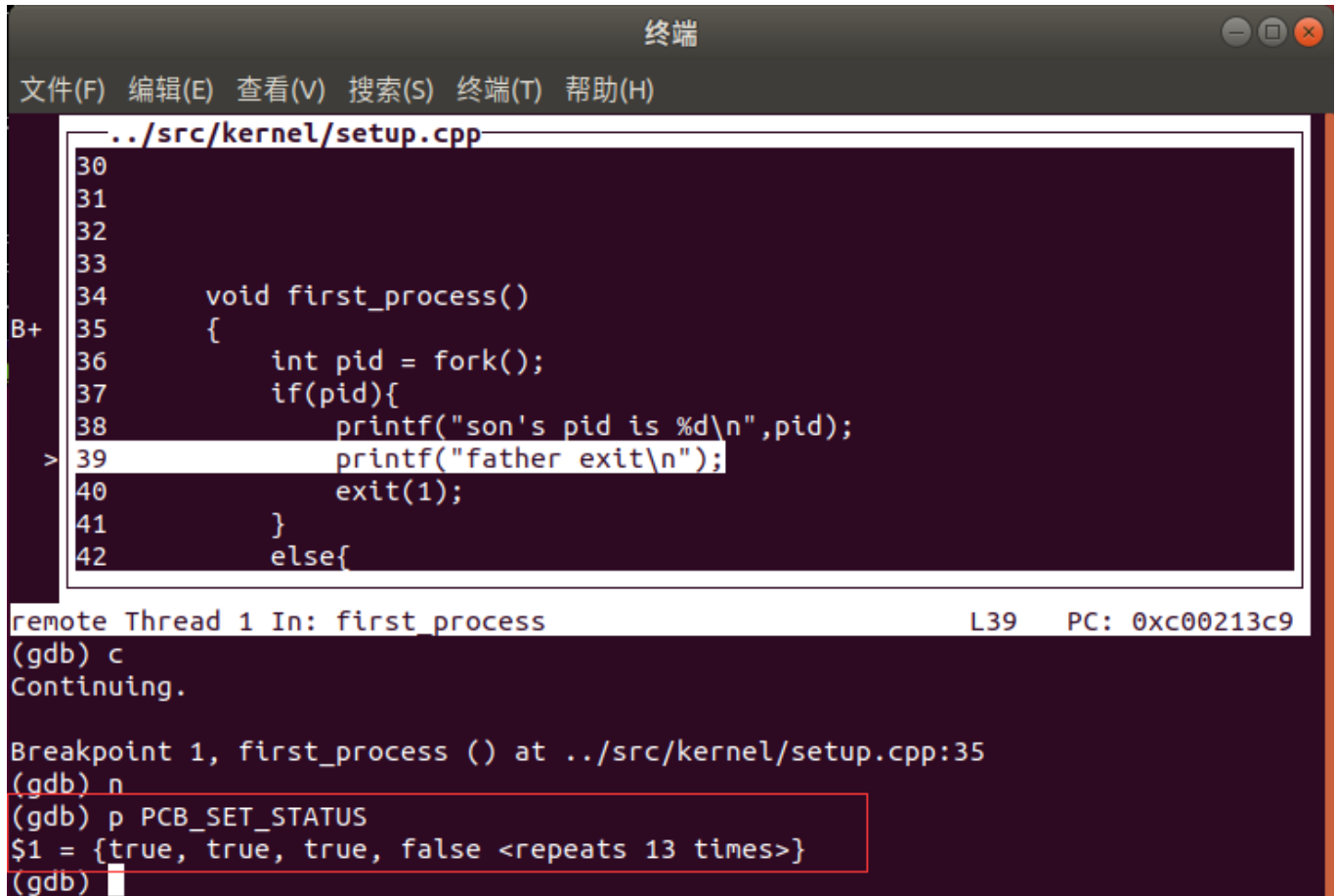
void first_thread(void *arg)
{
    printf("start process\n");
    programManager.executeProcess((const char *)first_process, 1);
    asm_halt();
}

```



make debug 进行测试

三个PCB, 一个线程, 一个父进程, 一个子进程



The screenshot shows a terminal window titled "终端" (Terminal). The menu bar includes "文件(F)", "编辑(E)", "查看(V)", "搜索(S)", "终端(T)", and "帮助(H)". The main content area displays the source code of `../src/kernel/setup.cpp` with line numbers 30 through 42. The code defines a function `first_process()` that uses `fork()` to create a child process. The child process prints its PID and then the parent process prints "father exit\n" before calling `exit(1)`. The terminal also shows GDB output: "remote Thread 1 In: first\_process" with PC address `0xc00213c9`, followed by commands `(gdb) c` and `(gdb) n`. A breakpoint is set at line 35. The command `(gdb) p PCB_SET_STATUS` is executed, showing the output `$1 = {true, true, true, false <repeats 13 times>}`. The prompt `(gdb)` is visible at the bottom.

```
../src/kernel/setup.cpp
30
31
32
33
34     void first_process()
35     {
36         int pid = fork();
37         if(pid){
38             printf("son's pid is %d\n",pid);
39             printf("father exit\n");
40             exit(1);
41         }
42         else{
remote Thread 1 In: first_process                                L39    PC: 0xc00213c9
(gdb) c
Continuing.

Breakpoint 1, first_process () at ../src/kernel/setup.cpp:35
(gdb) n
(gdb) p PCB_SET_STATUS
$1 = {true, true, true, false <repeats 13 times>}
(gdb)
```

一路 `continue` 跑多几次, 发现我们无论父进程 `exit` 完以后还是僵尸子进程 `exit` 以后PCB已经释放完了!

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/kernel/program.cpp

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

remote Thread 1 In: ProgramManager::schedule L94 PC: 0xc00202c2

at ../src/kernel/program.cpp:94

Continuing.

Breakpoint 1, ProgramManager::schedule (this=0xc00346e0 &lt;programManager&gt;)

at ../src/kernel/program.cpp:94

(gdb) p PCB\_SET\_STATUS

\$1 = {true, false &lt;repeats 15 times&gt;}

(gdb)

```
start process
son's pid is 2
father exit
son exit
```