



# 本科生实验报告

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：\_\_\_\_熊彦钧\_\_\_\_

学生学号：\_\_\_\_23336266\_\_\_\_

实验地点：\_\_\_\_实验楼 B203\_\_\_\_

实验成绩：\_\_\_\_

报告时间：\_\_\_\_2025 年 6 月 5 日\_\_\_\_

## Section 1 实验概述

- 1. 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。
- 2. 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`。
- 3. 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 （对应 assignment1） -----

- 任务要求： 编写一个系统调用，然后在进程中调用它，并使用 `gdb` 分析系统调用的过程。
- 实验步骤：①编写并调用我的系统调用：首先我们要在 `syscall.h` 中声明系统调用：

```

1 #ifndef SYSCALL_H
2 #define SYSCALL_H
3
4 #include "os_constant.h"
5
6 class SystemService
7 {
8 public:
9     SystemService();
10    void initialize();
11    // 设置系统调用, index=系统调用号, function=处理第index个系统调用函数的地址
12    bool setSystemCall(int index, int function);
13 };
14
15 // 第0个系统调用
16 int syscall_0(int first, int second, int third, int forth, int fifth);
17 // 第1个系统调用(printf)
18 int syscall_1(const char *str);
19 // 第2个系统调用(printf)
20 int syscall_getpid();
21
22 #endif

```

经过实践，这里即使不声明这些函数，只要在 `setup.cpp` 中正确设置了系统调用表，这些系统调用也能正常执行。但是，我们还是在 `syscall.h` 中声明一下这些函数，以便于系统调用的管理和查找。

随后，我们在 `setup.cpp` 中，先分别实现这些函数：

```

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

int syscall_1(const char* str){
    stdio.print(str);
    return 0;
}

int syscall_getpid(){
    printf("current process's pid: %d\n", programManager.running->pid);
    return 0;
}

```

其中 0 号系统调用的作用是对传参求和，1 号系统调用的作用是打印传入的字符串，2 号系统调用的作用是输出当前进程的 pid。

实现了这些系统调用后，我们还需要在 `setup_kernel` 中把它们写入到系统调用表中，使线程能够通过系统调用号调用它们：

```

// 设置0号系统调用
systemService.setSystemCall(0, (int)syscall_0);
systemService.setSystemCall(1, (int)syscall_1);
systemService.setSystemCall(2, (int)syscall_getpid);

```

系统调用实现后，我们需要创建进程并调用它：

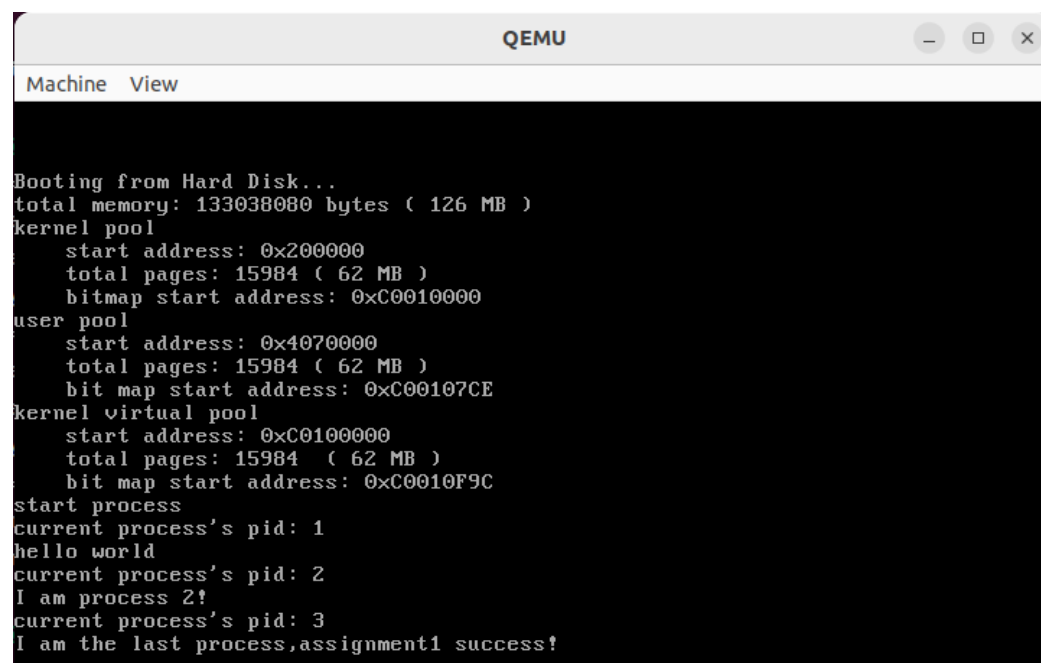
```
void first_process()
{
    asm_system_call(2);
    char* str="hello world\n";
    asm_system_call(1,(int)str);
    asm_halt();
}

void second_process(){
    asm_system_call(2);
    char* str="I am process 2!\n";
    asm_system_call(1,(int)str);
    asm_halt();
}

void third_process(){
    asm_system_call(2);
    char* str="I am the last process,assignment1 success!\n";
    asm_system_call(1,(int)str);
    asm_halt();
}

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

程序运行后输出如下：

A screenshot of a QEMU window titled "QEMU" with standard window controls. The window contains a terminal-like output showing the boot process and execution of three processes. The output text is as follows:

```
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
current process's pid: 1
hello world
current process's pid: 2
I am process 2!
current process's pid: 3
I am the last process,assignment1 success!
```

我们可以看到创建的进程成功调用了系统调用，这一部分的任务完成。

②根据 `gdb` 来分析执行系统调用后的栈的变化情况：

首先我们总结一下系统调用的流程，明确我们需要在什么地方设置断点。

- 1.用户程序调用 `asm_system_call` 函数
- 2.`asm_system_call` 设置参数并触发 `int 0x80` 中断
- 3.CPU 切换到内核态，保存上下文并跳转到 `asm_system_call_handler`
- 4.处理系统调用
- 5.恢复上下文并返回用户态

因此，我们需要在 `asm_system_call` 函数、`int 0x80` 中断、`asm_system_call_handler` 函数前后设置断点，查看栈的内容。

注：为了方便我们 `debug`，我们搬用 `src3` 的文件进行 `debug`（源码为 `assignment1 (for debug)`），并把第 `0` 个系统调用改为第一个系统调用，以便于查看寄存器数值的改变。

（1）我们先查看 `asm_system_call` 函数，在进入 `asm_system_call` 函数前，栈的情况如下：

```
esp -> 返回地址
      str 指针
      1 (系统调用号)
```

我们输入如下指令以在 `first_process` 的 `asm_system_call` 函数之前设置断点：

```
make debug
b setup.cpp:31
list
c
```

我们使用 `info register` 查看栈的情况

```
Terminal
../src/kernel/setup.cpp
27     first, second, third, forth, fifth);
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
32 {
33     asm_system_call(1, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
}

remote Thread 1.1 In: first_process L32 PC: 0xc00212f7
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8049000 0x8049000
ebp      0x0      0x0
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--

remote Thread 1.1 In: first_process L32 PC: 0xc00212f7
x0      0
eip      0xc00212f7 0xc00212f7 <first_process()>
eflags   0x202     [ IOPL=0 IF ]
cs       0x2b      43
ss       0x3b      59
ds       0x33      51
es       0x33      51
```

进入 `asm_system_call` 后，栈的情况应该如下：

```
esp -> edi
      esi
      edx
      ecx
      ebx
ebp -> 旧的 ebp
      返回地址
      str 指针
      1 (系统调用号)
```

```
Terminal
./src/kernel/setup.cpp
27         first, second, third, forth, fifth);
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
32 {
33     asm_system_call(1, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
}

remote Thread 1.1 In: first_process L33 PC: 0xc00212fd
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048ff4 0x8048ff4
ebp      0x8048ffc 0x8048ffc
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--

remote Thread 1.1 In: first_process L33 PC: 0xc00212fd
x0      0
eip      0xc00212fd 0xc00212fd <first_process()+6>
eflags   0x202     [ IOPL=0 IF ]
cs       0x2b      43
ss       0x3b      59
ds       0x33      51
es       0x33      51
```

我们可以看到:当前特权级 CPL 被存储在 cs 的低 2 位上,cs=43=00101011;因此 CPL=3,当前处于用户级。并且此时我们可以看到 esp 的地址也显示了当前为用户态;

另外:当运行到 asm\_halt() 时,我们可以看到 eax 的值为 592=1+132+324+12+124,系统调用正确。

```
Terminal
+1
./src/kernel/setup.cpp
27         first, second, third, forth, fifth);
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
B+ 32 {
B+ 33     asm_system_call(1, 132, 324, 12, 124);
B+> 34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
}

remote Thread 1.1 In: first_process L34 PC: 0xc002131a
eax      0x250      592
ecx      0x0        0
edx      0x0        0
ebx      0x0        0
esp      0x8048ff4   0x8048ff4
ebp      0x8048ffc   0x8048ffc
esi      0x0        0
--Type <RET> for more, q to quit, c to continue without paging--
```

(2) 随后我们查看 `int 0x80` 中断前后的栈情况：触发 `0x80` 中断后，CPU 会自动 `eflags`、`cs`、`eip` 压栈，压栈后栈的情况如下：

```
esp -> eip (返回地址)
    cs
    eflags
    edi
    esi
    edx
    ecx
    ebx
ebp -> old ebp
    调用者返回地址
    str 指针
    1 (系统调用号)
```



```
Terminal
+
./src/utils/asm_utils.asm
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]
B+> 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 1.1 In: asm_system_call L144 PC: 0xc00226ba
eax      0x1      1
ecx      0x144    324
edx      0xc      12
ebx      0x84     132
esp      0x8048fb8 0x8048fb8
ebp      0x8048fcc 0x8048fcc
esi      0x7c     124
--Type <RET> for more, q to quit, c to continue without paging--
```

我们可以看到此时的 `eax` 为系统调用号 1, `ebx`、`ecx`、`edx`、`esi` 为传入的参数大小。另外我们发现 `esp` 也确实发生了变化。

(3) 最后我们进入 `asm_system_call_handler` 函数。这个函数首先会保存更多的寄存器：

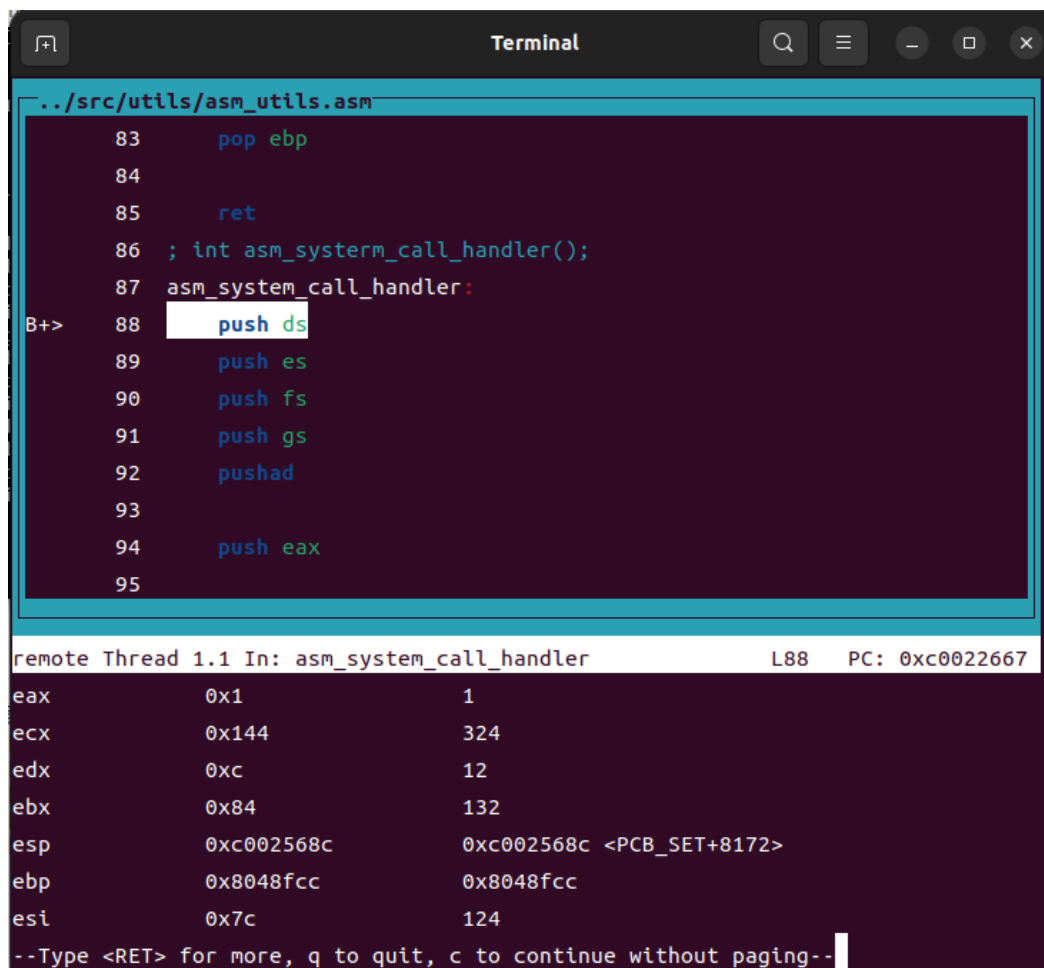
```
push ds
push es
push fs
push gs
pushad
```

`pushad` 会按顺序压入：`EAX`，`ECX`，`EDX`，`EBX`，`ESP`(原始值)，`EBP`，`ESI`，`EDI`；

此时栈的情况如下：

```
esp -> edi (由 pushad)
      esi
      ebp
      esp (原始值)
```

ebx  
edx  
ecx  
eax  
gs  
fs  
es  
ds  
eip (中断返回地址)  
cs  
eflags  
edi (之前保存的)  
esi  
edx  
ecx  
ebx  
old ebp  
调用者返回地址  
str 指针  
1 (系统调用号)



The screenshot shows a terminal window titled "Terminal" with a search icon, a menu icon, and window control buttons. The terminal displays assembly code from a file `../src/utils/asm_utils.asm`. The code includes instructions like `pop ebp`, `ret`, and a comment `; int asm_system_call_handler();`. A function `asm_system_call_handler:` is defined, starting with `push ds` (highlighted by a cursor), followed by `push es`, `push fs`, `push gs`, `pushad`, and `push eax`. Below the code, a register dump for "remote Thread 1.1 In: asm\_system\_call\_handler" is shown at address L88, PC: 0xc0022667. The dump lists registers `eax` through `esi` with their hexadecimal values and decimal equivalents. At the bottom, a prompt `--Type <RET> for more, q to quit, c to continue without paging--` is visible.

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

remote Thread 1.1 In: asm_system_call_handler      L88    PC: 0xc0022667
eax          0x1                1
ecx          0x144              324
edx          0xc                12
ebx          0x84               132
esp          0xc002568c         0xc002568c <PCB_SET+8172>
ebp          0x8048fcc          0x8048fcc
esi          0x7c               124
--Type <RET> for more, q to quit, c to continue without paging--
```

```

x0          0
eip         0xc0022667      0xc0022667 <asm_system_call_handler>
eflags      0x12           [ IOPL=0 AF ]
cs          0x20           32
ss          0x10           16
ds          0x33           51
es          0x33           51

```

此时我们可以看到 `esp` 的地址已经转为了 `0` 特权级的栈段(即内核的栈段)，并且 `cs=32=00100000`，低两位为 `0`，代表当前特权级 `CPL=0`；

在系统调用完成后(`asm_system_call` 函数的后半段)，系统会逐渐将各个寄存器出栈(`pop`)，最后会把特权级重新修改为用户级(`CPL=3`)：

The screenshot shows a GDB terminal window with the following content:

```

Terminal
../src/utls/asm_utils.asm
146     int 0x80
147
148     pop edi
149     pop esi
150     pop edx
151     pop ecx
152     pop ebx
B+> 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      L153  PC: 0xc00226c4
x0          0
eip         0xc00226c4      0xc00226c4 <asm_system_call+33>
eflags      0x212          [ IOPL=0 IF AF ]
cs          0x2b           43
ss          0x3b           59
ds          0x33           51
es          0x33           51
--Type <RET> for more, q to quit, c to continue without paging--

```

至此，我们已经大致了解了系统调用后栈的变化，栈的变化反映了系统从用户态到内核态转换的过程。

### ③根据 `gdb` 来说明 `TSS` 在系统调用执行过程中的作用

首先我们可以在 `include/tss.h` 中查看 `TSS` 的结构（此处省略），随后，

TSS 会在 `setup_kernel` 中被初始化，我们可以输入 ‘`p/x tss`’ 查看初始值。

```
remote Thread 1.1 In: asm_system_call L130 PC: 0xc00226a3
Breakpoint 1, asm_system_call () at ../src/utils/asm_utils.asm:130
(gdb) p/x tss
$1 = {backlink = 0x0, esp0 = 0xc00256a0, 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 = 0xc00337ac}
(gdb)
```

在用户态调用系统调用时，我们查看当前栈指针发现 `esp` 指向的是用户态的地址（这在上面已经提到了）：

```
(gdb) info registers esp
esp          0x8048fd0          0x8048fd0
```

当触发 `int 0x80` 中断时，CPU 会从 TSS 中加载 `ss0` 和 `esp0` 作为新的栈指针，并自动压入用户态 SS、ESP、EFLAGS、CS、EIP。我们通过 `gdb` 观察栈变化：

```
(gdb) si
asm_system_call_handler () at ../src/utils/asm_utils.asm:88
(gdb) info registers esp
esp          0xc002568c          0xc002568c <PCB_SET+8172>
```

我们可以看到确实如此。

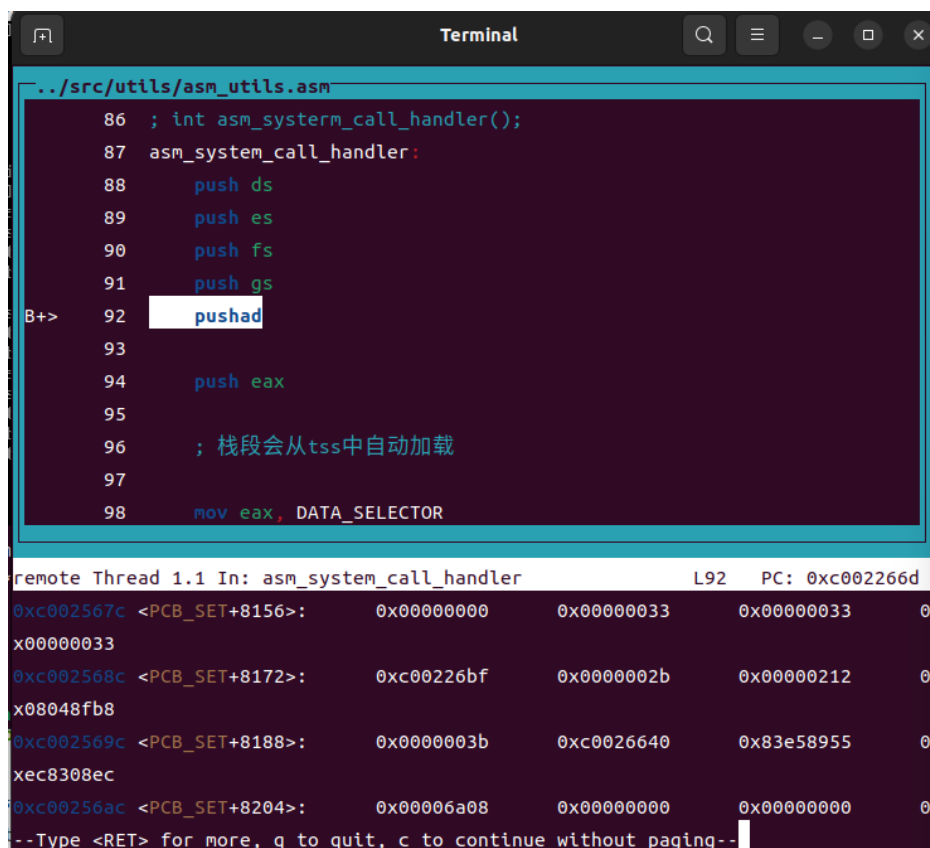
当进入系统调用处理程序后，此时栈的内容如下：

```
Terminal
../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

remote Thread 1.1 In: asm_system_call_handler L88 PC: 0xc0022667
(gdb) info registers esp
esp          0xc002568c          0xc002568c <PCB_SET+8172>
(gdb) x/8x $esp
0xc002568c <PCB_SET+8172>: 0xc00226bf  0x0000002b  0x000000212
0x0000002b  0x0000002b  0x0000002b  0x0000002b  0x0000002b  0x0000002b
0xc002569c <PCB_SET+8188>: 0x0000003b  0xc0026640  0x83e58955
0xec8308ec
```

栈的内容分别是 `edi`、`CS`、`eflags`、`eip`（返回地址）……（详细请见上面同一位置下对 `esp` 栈内容的展示）

随着寄存器的不断保存，栈会跟着增长，并且按照我们前面提到的栈结构的顺序入栈，本人对比过后发现确实是一样的，这里就不再一一阐述了：



```
Terminal
../src/utils/asm_utils.asm
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
98     mov eax, DATA_SELECTOR

remote Thread 1.1 In: asm_system_call_handler L92 PC: 0xc002266d
0xc002567c <PCB_SET+8156>: 0x00000000 0x00000033 0x00000033 0
0x00000033
0xc002568c <PCB_SET+8172>: 0xc00226bf 0x0000002b 0x00000212 0
0x08048fb8
0xc002569c <PCB_SET+8188>: 0x0000003b 0xc0022640 0x83e58955 0
0xec8308ec
0xc00256ac <PCB_SET+8204>: 0x000006a0 0x00000000 0x00000000 0
--Type <RET> for more, q to quit, c to continue without paging--
```

返回用户态时（即执行 `iret` 时），栈指针会恢复为用户态栈指针，这里展示返回前后对比：



```
Breakpoint 4, asm_system_call_handler () at ../src/utils/asm_utils.asm:128
(gdb) info register esp
esp             0xc002568c             0xc002568c <PCB_SET+8172>
(gdb) si
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) info register esp
esp             0x8048fb8              0x8048fb8
(gdb)
```

至此，我们分析完毕，TSS 在系统调用的作用如下：

### 1. 特权级栈切换：

当从用户态(CPL=3)通过 `int 0x80` 进入内核态(CPL=0)时，CPU 自动从 TSS 中加载 `ss0` 和 `esp0` 作为新的栈指针；

确保内核使用独立的内核栈，与用户栈隔离；

## 2. 安全上下文保存：

CPU 自动将用户态的 SS/ESP 压入新内核栈；

通过 `iret` 指令自动恢复用户态上下文；

## 3. 多任务支持：

每个任务有自己的 TSS，保存任务状态；

在任务切换时，CPU 自动保存/恢复任务状态；

## 4. 隔离保护：

内核栈位置由 TSS 决定，用户程序无法直接修改；

确保每次系统调用都有干净的执行环境。

至此，实验任务 1 完成。

# ----- 实验任务 2 （对应 assignment2） -----

- 任务要求：实现 `fork` 函数，并用 `gdb` 来理解 `fork` 函数的过程。

- 实验步骤：①根据代码逻辑和执行结果来分析 `fork` 实现的基本思路

`fork` 的实现主要位于 `ProgramManager::fork()` 和 `ProgramManager::copyProcess()` 两个函数中：

1. 权限检查：首先检查当前进程是否是内核线程（通过检查 `pageDirectoryAddress` 是否存在）；

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

2. 创建子进程：调用 `executeProcess()` 创建子进程的 PCB 结构；

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

executeProcess 函数的具体内容为：分配 PCB 结构、创建进程页目录表、创建用户虚拟地址池、初始化进程状态为 READY。

```
int ProgramManager::executeProcess(const char *filename, int priority)
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    // 在线程创建的基础上初步创建进程的PCB
    int pid = executeThread((ThreadFunction)load_process,
                           (void *)filename, filename, priority);

    if (pid == -1)
    {
        interruptManager.setInterruptStatus(status);
        return -1;
    }

    // 找到刚刚创建的PCB
    PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);

    // 创建进程的页目录表
    process->pageDirectoryAddress = createProcessPageDirectory();
    //printf("%x\n", process->pageDirectoryAddress);

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

    // 创建进程的虚拟地址池
    bool res = createUserVirtualPool(process);

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

    interruptManager.setInterruptStatus(status);

    return pid;
}
```

3. 复制父进程：调用 copyProcess()复制父进程的状态到子进程；

```
// 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;
}
```

CopyProcess 的具体内容如下（仅展示部分代码截图）：

1. 复制进程栈：复制父进程的 ProcessStartStack 到子进程；
2. 设置子进程返回值：将子进程的 eax 寄存器值设为 0（这是 fork 返回 0 的关键）；

3. 设置子进程 PCB: 复制优先级、ticks 等属性;
4. 复制虚拟地址池: 复制父进程的虚拟地址池位图;
5. 复制页表和物理页:
  - 为子进程分配新的页表;
  - 为子进程分配新的物理页;
  - 复制父进程页表项到子进程;
  - 复制父进程物理页内容到子进程;

对于复制页表和物理页部分, 系统需要先遍历父进程页目录表, 对于每个存在的页表, 需要为子进程分配新的页表、复制页表项、为每个物理页分配新的物理页并复制内容。最后还需要使用 CR3 寄存器切换地址空间进行复制操作。

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

    // 计算页表的虚拟地址
    int *pageTableVaddr = (int *) (0xffc00000 + (i << 12));

    // 复制物理页
    for (int j = 0; j < 1024; ++j)
    {
        // 无对应物理页
        if (!(pageTableVaddr[j] & 0x1))
        {
            continue;
        }

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

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

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

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

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

4. 设置返回值: 父进程返回子进程 PID, 子进程返回 0;

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



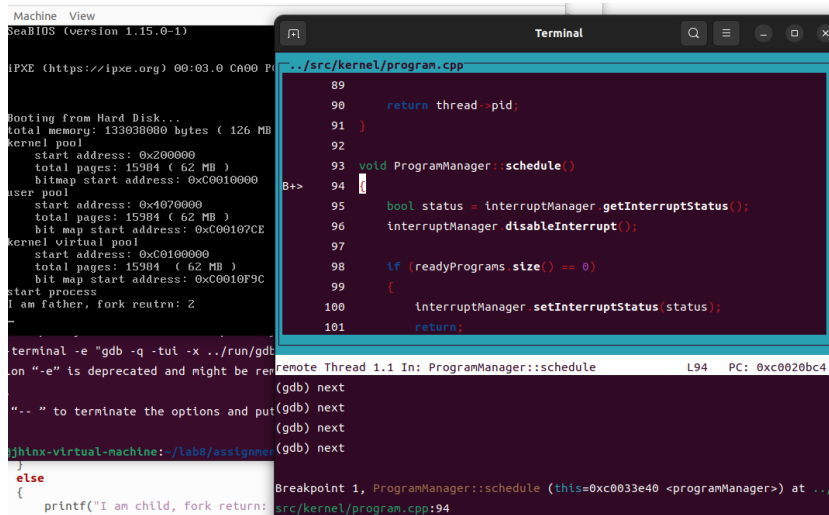
运行程序，结果如下：

```
start process
I am father, fork reutrnr: 2
I am child, fork return: 0, my pid: 2
```

从结果上我们可以看到，父进程会返回子进程的 `pid`，子进程则返回 `0`。这符合 Linux 的 `fork` 逻辑，这一部分实验任务完成。

②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，以及父子进程 `fork` 后返回过程的异同。

子进程第一次被调度执行应该是从 `schedule` 函数调用后开始，我们先在 `first_process` 处设置断点，随后不断输入 `next` 直到 `qemu` 输出父进程信息，然后我们会发现最后一次输入 `next`，系统又跳回了 `schedule`，此时子进程正式被第一次调度：



The screenshot shows a virtual machine window on the left and a GDB terminal window on the right. The virtual machine window displays the booting process of SeaBIOS, including memory allocation for kernel and user pools. The GDB terminal window shows the execution of a program, with the `ProgramManager::schedule` function being called. The GDB terminal also shows the output of the program, which prints "I am father, fork reutrnr: 2" and "I am child, fork return: 0, my pid: 2".

```
Machine View
SeaBIOS (version 1.15.0-1)
IPXE (https://ipxe.org) 00:03.0 CA00 P...
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
I am father, fork reutrnr: 2

terminal -e "gdb -q -tui -x ../run/gdb...
on "-e" is deprecated and might be re...
remote Thread 1.1 In: ProgramManager::schedule
L94 PC: 0xc0020bc4
(gdb) next
(gdb) next
(gdb) next
(gdb) next
Breakpoint 1, ProgramManager::schedule (this=0xc0033e40 <programManager>) at ../src/kernel/program.cpp:94
  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     }
```

系统在调度的前置工作结束后，会进入 `activateProgramPage` 函数：

```
Terminal
./src/kernel/program.cpp
327     asm_start_process((int)interruptStack);
328 }
329
330 void ProgramManager::activateProgramPage(PCB *program)
331 {
> 332     int paddr = PAGE_DIRECTORY;
333
334     if (program->pageDirectoryAddress)
335     {
336         tss.esp0 = (int)program + PAGE_SIZE;
337         paddr = memoryManager.vaddr2paddr(program->pageDirectoryAdd
338     }
339
remote Thread 1.1 In: ProgramManager::activateProgramPage L332 PC: 0xc0021270
(gdb) s
(gdb) s
ProgramManager::schedule (this=0xc0033e40 <programManager>) at ./src/kernel/pro
gram.cpp:128
(gdb) s
ProgramManager::activateProgramPage (this=0xc0033e40 <programManager>, program=0
xc0023e00 <PCB_SET>) at ./src/kernel/program.cpp:332
```

在这里，系统会设置 CR3 寄存器指向子进程页目录表，并更新 TSS 的 esp0 指针：

```
336         tss.esp0 = (int)program + PAGE_SIZE;
> 337         paddr = memoryManager.vaddr2paddr(program->pageDirectoryAdd
338     }
339
340     asm_update_cr3(paddr);
341 }
remote Thread 1.1 In: ProgramManager::activateProgramPage L337 PC: 0xc002128e
    trace = 0x0, ioMap = 0xc0033f0c}
(gdb) s
(gdb) p/x tss
$2 = {backlink = 0x0, esp0 = 0xc0025e00, 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 = 0xc0033f0c}
(gdb)
```

```
33 asm_update_cr3:
> 34 push eax
35 mov eax, dword[esp+8]
36 mov cr3, eax
37 pop eax
38 ret
39 asm_start_process:
40 ;jmp $
41 mov eax, dword[esp+4]
42 mov esp, eax

remote Thread 1.1 In: asm_update_cr3 L34 PC: 0xc0022c16
8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
xmm1 {v4_float = {0x0, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x0 <repeats 16 times>}, v8_int16 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_int32 = {0x0, 0x0, 0x0, 0x0}, v2_int64 = {0x0, 0x0}, uint128 = 0x0}
--Type <RET> for more, q to quit, c to continue without paging--qQuit
(gdb) s
asm_update_cr3 () at ../src/utils/asm_utils.asm:34
(gdb)

> 38 ret
39 asm_start_process:
40 ;jmp $
41 mov eax, dword[esp+4]

remote Thread 1.1 In: asm_update_cr3 L38 PC:
asm_update_cr3 () at ../src/utils/asm_utils.asm:34
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) info register cr3
cr3 0x100000 [ PDBR=256 PCID=0 ]
```

我们可以看到 `cr3` 被设置成段选择子 `0x100000`。随后，系统会进入漫长的复制、加载子进程、切换进程的流程，这里我们就不再阐述了，我们直接跳转到子进程开始执行的时候，即先跳转到 `schedule` 运行后，再跳转到 `asm_start_process`:

```
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
49      iret
50
51 ; void asm_ltr(int tr)

remote Thread 1.1 In: asm_start_process L41 PC: 0xc0022c20
(gdb) s
(gdb) b asm_start_process
Breakpoint 2 at 0xc0022c20: file ../src/utils/asm_utils.asm, line 41.
(gdb) c
Continuing.

Breakpoint 2, asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb)
```

我们可以看到系统会先从 ProcessStartStack 加载段寄存器，并设置为用户态数据段选择子。

```
> 49      iret
50
51 ; void asm_ltr(int tr)

remote Thread 1.1 In: asm_start_process L49 PC: 0xc0022c2d
x0          0
eip         0xc0022c2d      0xc0022c2d <asm_start_process+13>
eflags      0x92           [ IOPL=0 SF AF ]
cs          0x20           32
ss          0x10           16
ds          0x33           51
es          0x33           51
```

系统还会设置栈指针：esp 指向用户栈顶，ss 为用户态栈段选择子

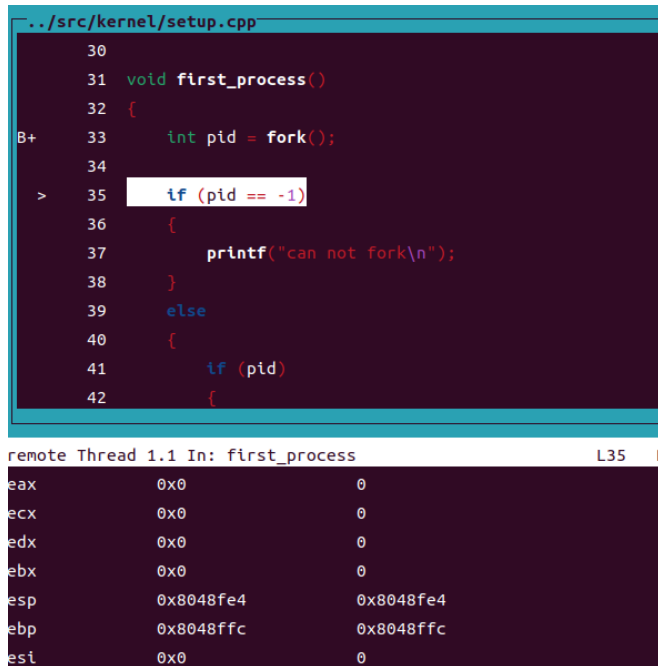
```
(gdb) info register esp ss
esp          0xc0026dec      0xc0026dec <PCB_SET+12268>
ss          0x10           16
```

最后，系统会设置代码段和指令指针：cs 为用户态代码段选择子，eip 指向 first\_process 函数：

```
(gdb) info register eip cs
eip          0xc0022c2d          0xc0022c2d <asm_start_process+13>
cs           0x20                32
```

最后，`iret` 指令执行后，系统就会跳转到 `eip` 指向的地址，并从内核态切换到用户态。

子进程在 `first_process` 函数中执行时，我们可以通过查看 `eax` 寄存器来确认返回值：



The screenshot shows a debugger window with the source code of `first_process` in `../src/kernel/setup.cpp`. The code is as follows:

```

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         {

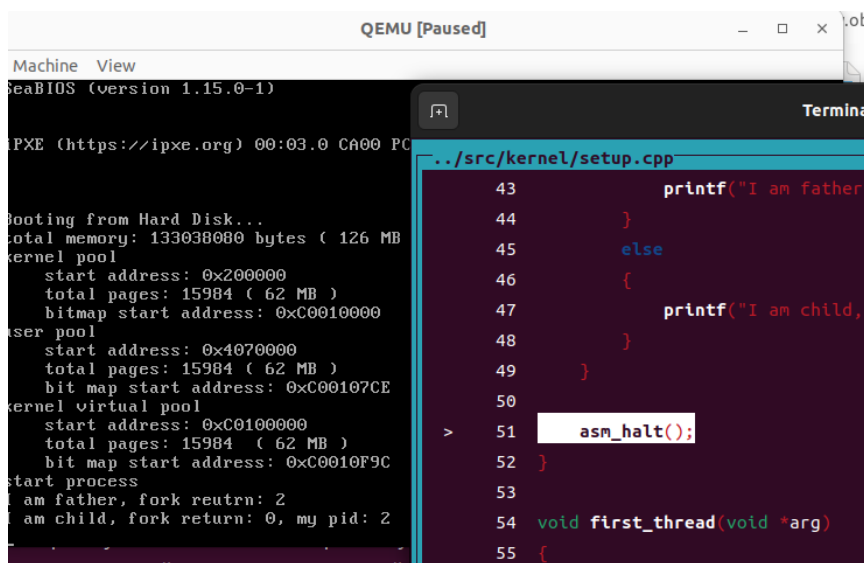
```

Below the source code, a register window titled "remote Thread 1.1 In: first\_process" shows the state of various registers. The `eax` register is highlighted and contains the value `0x0`.

Register	Value
eax	0x0
ecx	0x0
edx	0x0
ebx	0x0
esp	0x8048fe4
ebp	0x8048ffc
esi	0x0

此时的 `eax` 等于 `0`，确实是子进程。

不断运行后，`qemu` 最后输出内容，并即将终止进程，子进程正式返回：



The screenshot shows a QEMU terminal window titled "QEMU [Paused]". The terminal output includes the following text:

```

Machine View
SeaBIOS (version 1.15.0-1)
PXE (https://ipxe.org) 00:03.0 CA00 PC
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

```

Overlaid on the terminal is a debugger window showing the source code of `first_thread` in `../src/kernel/setup.cpp`. The code is as follows:

```

43     printf("I am father,
44     )
45     else
46     {
47         printf("I am child,
48         )
49     }
50
51     asm_halt();
52 }
53
54 void first_thread(void *arg)
55 {

```

现在，我们对比父进程在这一过程的区别。想要进入父进程，我们直接在

first\_process 上设置断点即可。

```
seabios (version 1.6.3)
Booting from Hard Disk
total memory: 1024000
kernel pool
start address: 0x00000000
total pages: 1024
bitmap start: 0x00000000
user pool
start address: 0x00000000
total pages: 1024
bit map start: 0x00000000
kernel virtual
start address: 0x00000000
total pages: 1024
bit map start: 0x00000000
start process

Terminal
../src/kernel/syscall.cpp
31 int syscall_write(const char *str) {
32     return stdio_print(str);
33 }
34
35 int fork() {
36     return asm_system_call(2);
37 }
38
39 int syscall_fork() {
40     return programManager.fork();
41 }
42
43

remote Thread 1.1 In: fork
(gdb) c
Continuing.

Breakpoint 1, first_process () at ../src/kernel/setup.cpp:32
(gdb) s
(gdb) s
fork () at ../src/kernel/syscall.cpp:36
```

执行完 `asm_system_call` 函数后，程序就正式进入了父进程中，此时我们可以看到 `eax` 寄存器的值为 2，代表父进程的 `pid`：

```
B+
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         {
43             // Parent process
44         }
45         else
46         {
47             // Child process
48         }
49     }
50 }

remote Thread 1.1 In: first_process
eax      0x2      2
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048fe4 0x8048fe4
ebp      0x8048ffc 0x8048ffc
esi      0x0      0
```

打印完信息后，父进程任务就完成了，但是此时它不会被立刻回收，而会进入子进程的逻辑中。

③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。

在前面的任务中，我们已经通过 gdb 知道了 eax 寄存器可以存储父子进程各自的 pid。现在我们结合代码来说明原理：

首先在 fork 函数中，我们可以看到，executeProcess 函数返回的是子进程的 pid，因此 fork 函数返回的也是子进程的 pid。父进程通过 fork 创建子进程，这就保证了父进程可以返回子进程的 pid。

```
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;
}
```

随后，子进程需要通过 copyProcess 复制父进程的内容，这几行代码通过显式设置，保证了子进程的返回值为 0：

```
// 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;
```

gdb 部分我们就不再重复了。实验任务 2 完成。

### ----- 实验任务 3 （对应 assignment3） -----

- 任务要求： ①复现 wait 和 exit 函数，并结合代码分析它们的执行过程； ②

实现回收孤儿进程和僵尸进程的有效方法。

- 实验步骤：①分析 `exit` 函数的执行过程，并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。

首先我们看看 `exit` 函数的执行过程，当进程显式调用 `exit(ret)` 时：

1. 关中断： `interruptManager.disableInterrupt()`;
2. 设置 PCB 状态：标记当前运行进程状态为 `DEAD`，保存返回值 `ret` 到 PCB 的 `retValue` 字段；

// 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;
```

3. 资源释放：如果是进程(有页目录表)，释放所有物理页、页表和页目录表，释放虚拟地址池 `bitmap` 占用的内核空间；

```
// 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);
}
```

4. 调度新进程：调用 `schedule()` 选择下一个运行的进程；

我们通过运行程序验证。主程序如下：



```

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

    if (pid == -1)
    {
        printf("can not fork\n");
        asm_halt();
    }
    else
    {
        if (pid)
        {
            printf("I am father\n");
            asm_halt();
        }
        else
        {
            printf("I am child, exit\n");
        }
    }
}

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

```

```

start process
I am father
thread exit
I am child, exit

```

观察程序运行结果，我们可以看到，`second_thread` 显式调用了 `exit` 函数，而 `first_process` 则隐式调用了 `exit` 函数。

这就是 `exit` 的执行过程，接下来我们分析进程退出后为什么隐式调用 `exit`，并说明为什么返回值是 `0`。这一原因就藏在进程创建的时候，系统会把 `exit` 设置为返回地址（下面一段代码位于 `load_process` 函数中）：

```

// 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;

```

正式有这一用户栈设置，当进程的主函数返回时，系统会调用 `exit`，成功实现隐式调用。

另外，我们注意到 `load_process` 函数的进程启动栈初始化时，我们把 `eax` 寄存器赋值为 0：

```
interruptStack->edi = 0;
interruptStack->esi = 0;
interruptStack->ebp = 0;
interruptStack->esp_dummy = 0;
interruptStack->ebx = 0;
interruptStack->edx = 0;
interruptStack->ecx = 0;
interruptStack->eax = 0;
interruptStack->gs = 0;
```

而函数隐式调用 `exit` 时，系统就会调用 `eax` 的值作为返回值，因此隐式调用 `exit` 的返回值都为 0。至此，`exit` 函数的几个细节我们都已分析完毕，这一部分实验任务完成。

②结合代码逻辑和具体的实例来分析 `wait` 的执行过程。`wait` 函数的主要功能是让父进程等待子进程结束，并获取子进程的退出状态，它的执行过程如下：

1. 进入循环：`wait` 函数进入一个无限循环，直到找到可返回的子进程或确认没有子进程。

2. 保存中断状态并禁用中断：在每次循环开始时，保存当前中断状态并禁用中断，以保证操作的原子性。这里和我们前面分析过的进程函数不同，`wait` 函数需要在每一次循环中都保证关中断。

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

item = this->allPrograms.head.next;
```

3. 遍历所有进程：从 `allPrograms` 链表的头部开始遍历所有进程：检查每个进程的 `parentPid` 是否等于当前进程的 `pid`；如果找到子进程，设置 `flag` 为 `false`；如果找到状态为 `DEAD` 的子进程，跳出循环。

```
// 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;
}
```

4. 处理找到的子进程：如果找到 DEAD 状态的子进程，将子进程的返回值存入 `retval`、释放子进程的 PCB 资源、恢复中断状态、返回子进程的 `pid`；

```
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;
}
```

5. 未找到可返回子进程的情况：如果没有子进程，则恢复中断状态并返回 -1；如果有子进程但都没结束，则恢复中断状态并调用 `schedule()` 让出 CPU。

```
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();
    }
}
```

接下来我们用测试样例具体展示 `wait` 的逻辑：

```
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);
        }
    }
}
```

这个程序的执行流程如下：

第一次 `fork()` 后，父进程继续执行，子进程进入 `else` 分支，执行操作后调用 `exit(-123)` 退出；

第二次 `fork()` 后：父进程继续执行，新的子进程进入 `else` 分支，执行操作后调用 `exit(123934)` 退出；

在两个操作中间，父进程会进入 `wait` 的 `while` 循环，等待子进程。

程序运行结果如下：

```
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
```

我们可以看到，父进程等待子进程退出直到子进程全部执行完。`wait` 函数成功实现。

③实现回收僵尸进程的有效方法：经过询问助教，我们得知助教的本意是想要实现回收孤儿进程的有效方法。其实这一块的概念存在重合，如果父进程先于子进程退出，这时候子进程叫做孤儿进程，而孤儿进程死亡后也被称为僵尸进程；而单纯的子进程死亡后也叫僵尸进程。因此这里我们干脆两个都实现了。

首先我们需要明确：回收僵尸进程是父进程的任务，父进程在自己退出的时候，需要为自己创建的子进程“兜底”，即把已经 `DEAD` 的子进程的资源交换给系统。而孤儿进程，由于它的父进程因为意外提前退出了，当它自己 `DEAD` 的时候，没有父进程为它收拾。因此对于孤儿进程，我们需要作额外的处理，以确保孤儿进程在变成僵尸进程后资源能够正常地被回收。

了解了这一系列的过程后，我们先试图实现僵尸进程的回收，即修改 `wait` 函数。原来的 `wait` 函数，父进程只会等待一个子进程退出。虽然在没有孤儿进程的时候也显得合理，但是为了我们后面实现回收孤儿进程作铺垫，我们改为每一次调用 `wait` 函数都回收所有的僵尸进程，函数返回最后一个进程的 `pid`：

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

```

int lastPid = -1; // 记录最后一个回收的子进程 PID
while (true)
{
    interrupt = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();
    item = this->allPrograms.head.next;
    flag = true; // 假设没有子进程
    // 第一遍扫描: 回收所有僵尸子进程
    while (item)
    {
        child = ListItem2PCB(item, tagInAllList);
        item = item->next; // 提前获取下一个, 因为当前 item 可能被释放

        if (child->parentPid == this->running->pid)
        {
            flag = false; // 有子进程
            if (child->status == ProgramStatus::DEAD)
            {
                // 回收这个僵尸子进程
                if (retval)
                {
                    *retval = child->retValue;
                }
                lastPid = child->pid;
                releasePCB(child);
                printf("recycle      zombie      process:      %d\n", child->pid, child->parentPid);
            }
        }
    }
    // 如果没有子进程了, 返回最后一个回收的 PID 或 -1
    if (flag)
    {
        interruptManager.setInterruptStatus(interrupt);
        return lastPid != -1 ? lastPid : -1;
    }
    // 如果有子进程但都不是僵尸状态, 则调度等待
    else if (lastPid == -1)
    {
        interruptManager.setInterruptStatus(interrupt);
        schedule();
    }
    // 如果回收了至少一个子进程, 返回最后一个回收的 PID
    else

```

```

        {
            interruptManager.setInterruptStatus(interrupt);
            return lastPid;
        }
    }
}

```

随后，我们需要实现回收孤儿进程，我们的解决办法是：既然孤儿进程在父进程退出后就没有父进程了，那我们干脆创建一个进程（`init_process`）专门用于定期回收孤儿进程，当父进程意外退出时，我们手动把孤儿进程的父进程设置为 `init_process`，这样当 `init_process` 定期调用 `wait` 函数的时候，就可以实现对已经变为僵尸进程的孤儿进程的回收。

由于一个进程的退出，无论如何都会调用 `exit` 函数，因此我们修改 `exit` 函数的逻辑，将这个进程的子进程都交由 `init_process` 管理，核心修改如下：

```

// 1. 将当前进程的子进程交给init进程(PID=1)接管
// 2. 检查当前进程是否是孤儿进程(父进程已退出)
ListItem *item = this->allPrograms.head.next;
PCB *pcb;
bool parentFound = false;
bool parentDead = false;

while (item)
{
    pcb = ListItem2PCB(item, tagInAllList);

    // 处理当前进程的子进程（将它们变为孤儿）
    if (pcb->parentPid == program->pid)
    {
        pcb->parentPid = 1;
        printf("Process %d is now orphan, adopted by init process\n", pcb->pid);
    }

    // 检查当前进程的父进程状态
    if (program->parentPid != 0 && pcb->pid == program->parentPid)
    {
        parentFound = true;
        if (pcb->status == ProgramStatus::DEAD)
        {
            parentDead = true;
        }
    }

    item = item->next;
}

// 如果当前进程是孤儿进程（父进程不存在或已死亡）
if (program->parentPid != 0 && (!parentFound || parentDead))
{
    program->parentPid = 1;
    printf("Process %d is orphan, changing parent to init process\n", program->pid);
}

```

首先我们会把当前退出进程的所有子进程交给 `init_process` 管理；随后，为了避免“孤儿竟是我自己”的情况，我们还会检查当前退出的进程是不是孤儿

进程,如果是,则也将它交由 `init_process` 管理。最后,这些交由 `init_process` 管理的孤儿进程都会在变为僵尸进程后由 `init_process` 的 `wait` 函数定期回收。

我们编写测试样例来检验机制是否正确: 首先我们实现这一回收机制下必要的进程,即 `init_process` 进程:

```
void init_process()
{
    int retval;
    int pid;
    while (true)
    {
        while ((pid = wait(&retval)) != -1)
        {
            printf("recycle orphan process, pid: %d, return value: %d\n", pid, retval);
        }
    }
}

void first_thread(void *arg)
{
    printf("start process\n");
    programManager.executeProcess((const char *)init_process, 3);
    programManager.executeProcess((const char *)first_process, 3);
    programManager.executeProcess((const char *)second_process, 3);
    asm_halt();
}
```

`init_process (pid=1)` 会在它被调度的时候不断调用 `wait` 函数,回收僵尸进程。

随后我们编写了两个测试进程 `first_process` 和 `second_process`:

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

    if (pid)
    {
        printf("father process1 exit, pid: %d\n", programManager.running->pid);
        exit(123934);
    }
    else
    {
        uint32 tmp = 0xffffffff;
        while (tmp)
            --tmp;
        printf("child process1 exit, pid: %d\n", programManager.running->pid);
        exit(-123);
    }
}
```

```

void second_process()
{
    uint32 tmp = 0x1fffffff;
    while (tmp)
        --tmp;

    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("grandchild process2 exit, pid: %d\n", programManager.running->pid);
            exit(2333);
        }
    }
    else
    {
        uint32 tmp = 0xffffffff;
        while (tmp)
            --tmp;
        printf("child process2 exit, pid: %d\n", programManager.running->pid);
        exit(6266);
    }
}

```

程序运行，如下：

```

start process
father process1 exit, pid: 2
Process 4 is now orphan, adopted by init process
child process1 exit, pid: 4
child process2 exit, pid: 5
grandchild process2 exit, pid: 6
recycle zombie process: 5 (father: 3)
recycle zombie process: 6 (father: 3)
wait for a child process, pid: 6, return value: 2333
all child process exit, programs: 5
recycle zombie process: 4 (father: 1)
recycle orphan process, pid: 4, return value: -123

```

我们可以看到成功实现了回收孤儿进程和僵尸进程的机制，实验任务完成。

## Section 5 实验总结与心得体会

这一实验代码量大，重难点在于使用 `gdb` 分析函数运行的逻辑、串联这一部分。以下几点思考/建议：

（1）在使用 `gdb` 进行 `debug` 的时候，指令 ‘s’ 固然可以单步执行，但是在遇到循环的时候，特别是几百上千次的循环，使用 `s` 会一遍一遍地执行循环，浪费大量的时间，这时候我们可以采用指令 ‘next’，它是一种“进阶版”的单步执行，可以跳过循环；

（2）在一开始试图实现回收孤儿进程的时候，我的计划是额外编写一个回



收孤儿进程的函数，然后在 `first_thread` 中每隔一段时间就执行一次这个函数，但是在实际运行的时候我们发现，无论我们是否显式调用 `exit` 退出父进程，在父进程退出的一瞬间，系统就会闪退。我猜测的原因是父进程退出后，父子进程直接的共享进程会导致资源泄露。最后，我们迫不得已将回收机制嵌入到了 `exit` 函数和 `wait` 函数中；

（3）在采用 `gdb` 进行 `debug` 的过程中，如果我们光看 `debug` 的窗口，我们很容易会不知道程序运行到什么程度了，这时可以观看 `qemu` 的输出了解进度。比如我们看到 `qemu` 已经输出父进程函数里的信息了，我们就可以确认此时已经进入了子进程中。只有这样，我们才能知道现在看的寄存器存储的是父进程的信息还是子进程的信息。

（4）这一 `fork` 函数，只实现了资源复制，却没有实现写时复制（在 `lab9` 中）。

## Section 6 附录：代码清单

代码请见附件。