

lab8 从内核态到用户态

1、实验要求

Assignment 1 系统调用

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

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

Assignment 2 Fork的奥秘

实现fork函数，并回答以下问题。

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

Assignment 3 哼哈二将 wait & exit

实现wait函数和exit函数，并回答以下问题。

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

2、实验过程+实验结果

Assignment 1

系统调用的入口声明函数，在 `asm_utils.h`

```
extern "C" int asm_system_call(int index, int first = 0, int second = 0, int  
third = 0, int forth = 0, int fifth = 0);
```

当进行系统调用时，系统调用的参数通过5个寄存器来传递，因此系统调用的参数不可以超过5个

用户程序使用系统调用时会进行特权级转移，如果使用栈来传递参数，在调用系统调用的时候，系统调用的参数（即 `asm_system_call` 的参数）就会被保存在用户程序的栈中，即低特权级的栈中。系统调用发生后，从低特权级转移到高特权级，此时CPU会从TSS中加载高特权级的栈地址到esp寄存器中。C语言的代码在编译后会使用esp和ebp来访问栈的参数，但是前面保存参数的栈和现在期望取出函数参数而访问的栈并不是同一个栈，因此CPU无法在栈中找到函数的参数。因此需要通过寄存器来传递系统调用的参数。

`asm_system_call` 的实现，在 `asm_utils.asm`

```
asm_system_call:  
    push  ebp  
    mov   ebp, esp
```

在调用中断前，保护现场将系统调用的参数放到5个寄存器ebx, ecx, edx, esi, edi中，将系统调用号放到eax中

```
push  ebx  
push  ecx  
push  edx  
push  esi  
push  edi  
push  ds  
push  es  
push  fs  
push  gs  
  
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]
```

将系统调用的中断向量号定义为 `0x80`。保护现场后，使用指令 `int 0x80` 调用 `0x80` 中断。`0x80` 中断处理函数会根据保存在eax的系统调用号来调用不同的函数。

```
int 0x80
```

恢复现场

```
pop gs
pop fs
pop es
pop ds
pop edi
pop esi
pop edx
pop ecx
pop ebx
pop ebp
ret
```

SystemService 类的实现

```
void SystemService::initialize()
{
    memset((char *)system_call_table, 0, sizeof(int) * MAX_SYSTEM_CALL);
    // 代码段选择子默认是DPL=0的平坦模式代码段选择子, DPL=3, 否则用户态程序无法使用该中断
描述符
    interruptManager.setInterruptDescriptor(0x80,
(uint32)asm_system_call_handler, 3);
}

bool SystemService::setSystemCall(int index, int function)
{
    system_call_table[index] = function;
    return true;
}
```

在 setup.cpp 中编写系统调用

打印5个参数的值，并返回5个参数的总和

```
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;
}
```

创建进程，在进程中调用系统调用

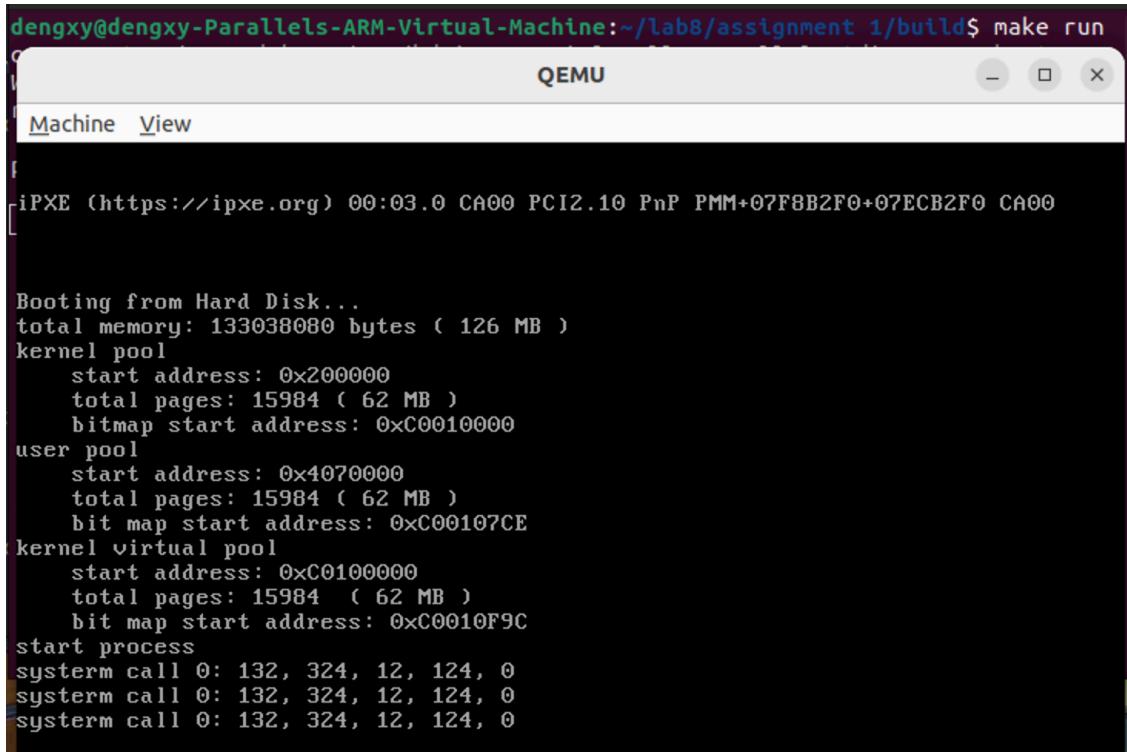
```
void first_process()
{
    asm_system_call(0, 132, 324, 12, 124);
    asm_halt();
}
```

调用第一个进程

```
// 初始化系统调用
systemService.initialize();
// 设置0号系统调用
systemService.setSystemCall(0, (int)syscall_0);

// 创建第一个线程
int pid = programManager.executeThread(first_thread, nullptr, "first
thread", 1);
if (pid == -1)
{
    printf("can not execute thread\n");
    asm_halt();
}
```

运行结果如图：



```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab8/assignment 1/build$ make run
QEMU
Machine View

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
    start address: 0x200000
    total pages: 15984 ( 62 MB )
    bitmap start address: 0xC0010000
user pool
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0xC00107CE
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0xC0010F9C
start process
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
```

执行系统调用后的栈的变化情况

内核和用户特权级之间的转换。

在实现用户进程之前，运行的特权级为0，当调度用户进程上处理器时候，用户运行在特权级3，因此需要从高特权级向低特权级转移。

通过使用 `asm_start_process` 的 `iret` 指令强制把低特权级下的段选择子和栈送入段寄存器和栈指针，在每一个进程的 PCB 的用户虚拟地址池中分配一页作为栈的空间。

在进程执行系统调用的时候，要改变 `current privilege level(CPL)` 从 PL3 (用户级) 到 PL0 (内核级)，通过 `0x80` 中断来实现的，并且使用 `tss.esp0` 的栈指针。

当进程的 PCB 首次加载到处理器执行时，CPU 会进入 load_process(const char *filename)，其中 filename 为进程函数的地址，load_process 会初始化启动用户进程需要的栈结构

以下验证把低特权级（用户级）的段选择子，栈放进 gs, fs, es, ds, cs, ss, esp 寄存器的过程

```
void load_process(const char *filename)
{
    interruptManager.disableInterrupt();

    PCB *process = programManager.running;
    ProcessStartStack *interruptStack =
        (ProcessStartStack *)((int)process + PAGE_SIZE -
sizeof(ProcessStartStack));

    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;

    interruptStack->fs = programManager.USER_DATA_SELECTOR;
    interruptStack->es = programManager.USER_DATA_SELECTOR;
    interruptStack->ds = programManager.USER_DATA_SELECTOR;

    interruptStack->eip = (int)filename;
    interruptStack->cs = programManager.USER_CODE_SELECTOR; // 用户模式平坦模
式
    // IOPL, IF = 1 开中断, MBS = 1 默认
    interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1);

    interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER,
1);
    if (interruptStack->esp == 0)
    {
        printf("can not build process!\n");
        process->status = ProgramStatus::DEAD;
        asm_halt();
    }
    interruptStack->esp += PAGE_SIZE;
    interruptStack->ss = programManager.USER_STACK_SELECTOR;

    asm_start_process((int)interruptStack);
}
```

在 setup_kernel 设置断点

```
./src/kernel/setup.cpp
40     programManager.executeProcess((const char *)first_process, 1);
41     programManager.executeProcess((const char *)first_process, 1);
42     programManager.executeProcess((const char *)first_process, 1);
43     asm_halt();
44 }
45
46 extern "C" void setup_kernel()
B+> 47 {
48
49     // 中断管理器
50     interruptManager.initialize();
51     interruptManager.enableTimeInterrupt();
52     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_ha

remote Thread 1.1 In: setup_kernel                                L47    PC: 0xc0021389
0x0000ffff in ?? ()
(gdb) b setup_kernel
Breakpoint 1 at 0xc0021389: file ./src/kernel/setup.cpp, line 47.
(gdb) c
Continuing.

Breakpoint 1, setup_kernel () at ./src/kernel/setup.cpp:47
```

执行到 programManager.initialize()，查看 programManager 类的

USER_DATA_SELECTOR、USER_CODE_SELECTOR 和 USER_STACK_SELECTOR 选择子的值

```
./src/kernel/setup.cpp
51     interruptManager.enableTimeInterrupt();
52     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_ha
53
54     // 输出管理器
55     stdio.initialize();
56
57     // 进程/线程管理器
B+> 58     programManager.initialize();
59
60     // 内存管理器
B+> 61     memoryManager.initialize();
62
63     // 初始化系统调用
64     systemService.initialize();
65     // 设置0号系统调用

remote Thread 1.1 In: setup_kernel                                L61    PC: 0xc00213e4

Breakpoint 3, setup_kernel () at ./src/kernel/setup.cpp:61
(gdb) print /x programManager.USER_DATA_SELECTOR
$2 = 0x33
(gdb) print /x programManager.USER_CODE_SELECTOR
$3 = 0x2b
(gdb) print /x programManager.USER_STACK_SELECTOR
$4 = 0x3b
```

可以看到

```
programManager.USER_DATA_SELECTOR = 0x33
programManager.USER_CODE_SELECTOR = 0x2b
programManager.USER_STACK_SELECTOR = 0x3b
```

第一次调用进程

在 asm_start_process 设置断点，观察寄存器

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

remote Thread 1.1 In: asm_start_process          L41    PC: 0xc0022620
eax      0xc002567c      -1073588612
ecx      0xc0010000      -1073676288
edx      0x3b          59
ebx      0x0           0
esp      0xc0025644      0xc0025644 <PCB_SET+8068>
ebp      0xc0025670      0xc0025670 <PCB_SET+8112>
esi      0x0          0
--Type <RET> for more, q to quit, c to continue without paging--
```

在 `first_process` 设置断点，观察寄存器的值

```

..../src/kernel/setup.cpp
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
32 {
B+> 33     asm_system_call(0, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
40     programManager.executeProcess((const char *)first_process, 1);
41     programManager.executeProcess((const char *)first_process, 1);
42     programManager.executeProcess((const char *)first_process, 1);
43     asm_halt();
44 }
45
46 extern "C" void setup_kernel()
B+ 47 {
48
49     // 中断管理器
50     interruptManager.initialize();
51     interruptManager.enableTimeInterrupt();
52     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_ha
53
54     // 输出管理器

```

remote Thread 1.1 In: first_process		L33	PC: 0xc0021301
eax	0x0	0	
ecx	0x0	0	
edx	0x0	0	
ebx	0x0	0	
esp	0x8048ff4	0x8048ff4	
ebp	0x8048ffc	0x8048ffc	
esi	0x0	0	
edi	0x0	0	
eip	0xc0021301	0xc0021301 <first_process()>+6>	
eflags	0x202	[IOPL=0 IF]	
cs	0x2b	43	
ss	0x3b	59	
ds	0x33	51	
es	0x33	51	

`load_process` 有一个 `ProcessStartStack` 类的变量，即启动进程前放入栈的内容，将 `USER_DATA_SELECTOR` 放入 `es`, `fs`, `ds`, `USER_CODE_SELECTOR` 放入到 `cs`, `USER_STACK_SELECTOR` 放入到 `ss`。

- `programManager.USER_DATA_SELECTOR = 0x33, es=0x33, ds=0x33, fs=0x33`
- `programManager.USER_CODE_SELECTOR = 0x2b, cs=0x2b`
- `programManager.USER_STACK_SELECTOR = 0x3b, ss=0x3b`

每一个特权级都有自己的栈，因此通过内存管理器在用户虚拟空间中分配一页来作为进程特权级3的栈。

定义每一个进程的 PCB 的用户虚拟地址池的开始地址 `USER_VADDR_START` 为 `0x8047FF4`，栈的低地址为 `8047FF4`，分配一页，所以栈指针指向栈的栈顶的地址，即

$$esp = USER_V_ADDR_START + PAGE_SIZE = 0x8047FF4 + 4096 = 0x8048FF4 \quad ($$

第二次调用进程和第三次调用进程与第一次调用进程的过程类似，以下不再赘述

第二次调用进程

```
./src/utils/asm_utils.asm
29 ASM_GDTR dw 0
30         dd 0
31 ASM_TEMP dd 0
32 ; void asm_update_cr3(int address)
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 $  
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;
```

remote Thread 1.1 In: asm_start_process L41 PC: 0xc0022620

eax	0xc002667c	-1073584516
ecx	0xc0010000	-1073676288
edx	0x3b	59
ebx	0x0	0
esp	0xc0026644	0xc0026644 <PCB_SET+12164>
ebp	0xc0026670	0xc0026670 <PCB_SET+12208>
esi	0x0	0
edi	0x0	0
eip	0xc0022620	0xc0022620 <asm_start_process>
eflags	0x92	[IOPL=0 SF AF]

```

..../src/kernel/setup.cpp
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
31 void first_process()
32 {
33     asm system call(0, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
40     programManager.executeProcess((const char *)first_process, 1);
41     programManager.executeProcess((const char *)first_process, 1);
42     programManager.executeProcess((const char *)first_process, 1);
43     asm_halt();
44 }
45
46 extern "C" void setup_kernel()
47 {
48
49     // 中断管理器
50     interruptManager.initialize();
51     interruptManager.enableTimeInterrupt();
52     interruptManager.setTimeInterrupt((void *)asm_time_interrupt_ha

```

remote Thread 1.1 In: first_process	L33	PC: 0xc0021301
eax	0x0	0
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048ff4	0x8048ff4
ebp	0x8048ffc	0x8048ffc
esi	0x0	0
edi	0x0	0
eip	0xc0021301	0xc0021301 <first_process()>+6>
eflags	0x202	[IOPL=0 IF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51

第三次调用进程

```
./src/utils/asm_utils.asm
25 ASM_UNHANDLED_INTERRUPT_INFO db 'Unhandled interrupt happened, halt
26           db 0
27 ASM_IDTR dw 0
28           dd 0
29 ASM_GDTR dw 0
30           dd 0
31 ASM_TEMP dd 0
32 ; void asm_update_cr3(int address)
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 $
B+> 41     mov eax, dword[esp+4]
42     mov esp, eax
```

```
remote Thread 1.1 In: asm_start_process          L41    PC: 0xc0022620
eax      0xc002767c      -1073580420
ecx      0xc0010000      -1073676288
edx      0x3b          59
ebx      0x0           0
esp      0xc0027644      0xc0027644 <PCB_SET+16260>
ebp      0xc0027670      0xc0027670 <PCB_SET+16304>
esi      0x0           0
edi      0x0           0
eip      0xc0022620      0xc0022620 <asm_start_process>
```

```

..../src/kernel/setup.cpp
21 // Task State Segment
22 TSS tss;
23
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
31 void first_process()
32 {
33     asm_system_call(0, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
40     programManager.executeProcess((const char *)first_process, 1);
41     programManager.executeProcess((const char *)first_process, 1);
42     programManager.executeProcess((const char *)first_process, 1);
43     asm_halt();
44 }
45
46 extern "C" void setup_kernel()
B+ 47 {

```

remote Thread 1.1 In: first_process		L33	PC: 0xc0021301
eax	0x0	0	
ecx	0x0	0	
edx	0x0	0	
ebx	0x0	0	
esp	0x8048ff4	0x8048ff4	
ebp	0x8048ffc	0x8048ffc	
esi	0x0	0	
edi	0x0	0	
eip	0xc0021301	0xc0021301 <first_process()>+6>	
eflags	0x202	[IOPL=0 IF]	
cs	0x2b	43	
ss	0x3b	59	
ds	0x33	51	
es	0x33	51	

通过以上过程可以发现，3次启动进程前 esp 栈指针的值都是 0x8048FF4，因为每一个进程都有自己的用户虚拟空间，而在用户虚拟空间都分配一页作为栈的空间，由此验证了通过 iret 的中断指令，在启动进程前，把低特权栈的信息在进入中断前保存在高特权栈中，从而能够让进程使用低特权级栈的 ss 和 esp。

TSS在系统调用执行过程中的作用

从低特权级向高特权级转移时，CPU首先会在TSS中找到高特权级栈的段选择子和栈指针，然后送入SS，ESP。此时，栈发生变化，此时的栈已经变成了TSS保存的高特权级的栈。

接着，中断发生前的SS、ESP、EFLAGS、CS、EIP被依次压入了高特权级栈。

TSS本来是用于CPU原生的进程切换的状态保存器，但在实验中不使用CPU原生的进程切换方案，因此TSS的作用仅限于 向CPU提供进程特权级转移时的栈段选择子和栈指针。

watch tss.ss0，观察 tss.ss0 的值，初始化函数 initializeTSS() 将 tss.ss0 设置为16

```
./src/kernel/program.cpp
182
183     memset((char *)address, 0, size);
B+ 184     tss.ss0 = STACK_SELECTOR; // 内核态堆栈段选择子
185
186     int low, high, limit;
187
> 188     limit = size - 1;
189     low = (address << 16) | (limit & 0xff);
190     // DPL = 0
191     high = (address & 0xff000000) | ((address & 0x00ff0000) >> 16)
192
193     int selector = asm_add_global_descriptor(low, high);
194     // RPL = 0
```



```
remote Thread 1.1 In: ProgramManager::initializeTSS      L188  PC: 0xc0020e56
Hardware watchpoint 3: tss.ss0
Old value = 0
New value = 16
ProgramManager::initializeTSS (this=0xc0033700 <programManager>) at ./src/kernel/program.cpp:188
(gdb) █
```

在 `asm_start_process` 设置断点，栈段寄存器在断点前设置 `ss = 16`，和上述 `tss.ss0` 的值相等

```

./src/utils/asm_utils.asm
28      dd 0
29  ASM_GDTR dw 0
30      dd 0
31  ASM_TEMP dd 0
32 ; void asm_update_cr3(int address)
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 $
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)
52  asm_ltr:
53      ltr word[esp + 1 * 4]
54      ret
55

```

		L41 PC: 0xc0022620
eax	0xc002567c	-1073588612
ecx	0xc0010000	-1073676288
edx	0x3b	59
ebx	0x0	0
esp	0xc0025644	0xc0025644 <PCB_SET+8068>
ebp	0xc0025670	0xc0025670 <PCB_SET+8112>
esi	0x0	0
edi	0x0	0
eip	0xc0022620	0xc0022620 <asm_start_process>
eflags	0x92	[IOPL=0 SF AF]
cs	0x20	32
ss	0x10	16
ds	0x8	8
es	0x8	8

对于 `tss.esp0`，在 `schedule` 函数每一次需要调度进程的时候执行 `activateProgramPage()` 时设置 `tss.esp0` 的值为进程函数地址加上一页的大小，即

$$tss.esp0 = (int)program + PAGE_SIZE \quad (2)$$

```

./src/kernel/program.cpp
324
325     asm_start_process((int)interruptStack);
326 }
327
328 void ProgramManager::activateProgramPage(PCB *program)
B+> 329 {
330     int paddr = PAGE_DIRECTORY;
331
332     if (program->pageDirectoryAddress)
333     {
334         tss.esp0 = (int)program + PAGE_SIZE;
335         paddr = memoryManager.vaddr2paddr(program->pageDirectoryAdd
336     }

```

remote Thread 1.1 In: ProgramManager::activateProgramPage L329 PC: 0xc002126e
(gdb) b ProgramManager::activateProgramPage
Breakpoint 1 at 0xc002126e: file ./src/kernel/program.cpp, line 329.
(gdb) c
Continuing.

Breakpoint 1, ProgramManager::activateProgramPage (this=0xc0033700 <programManager>, program=0xc00246c0 <PCB_SET+4096>) at ./src/kernel/program.cpp:329
(gdb)

第一次进行进程调换设置 tss.esp0 = 0xC00246C0

```

Breakpoint 2, ProgramManager::activateProgramPage (this=0xc0033700 <programManager>, program=0xc00246c0 <PCB_SET+4096>) at ./src/kernel/program.cpp:335
(gdb) print /x tss.esp0
$1 = 0xc00256c0
Breakpoint 2, ProgramManager::activateProgramPage (this=0xc0033700 <programManager>, program=0xc00256c0 <PCB_SET+8192>) at ./src/kernel/program.cpp:335
(gdb) print /x tss.esp0
$3 = 0xc00266c0
Breakpoint 2, ProgramManager::activateProgramPage (this=0xc0033700 <programManager>, program=0xc00266c0 <PCB_SET+12288>) at ./src/kernel/program.cpp:335
(gdb) print /x tss.esp0
$4 = 0xc00276c0

```

每次进程调度都会更新 tss.esp0 的值，每次更新的 tss.esp0 的值都相差 0x1000(4096 或 4KB)，即一页的大小，因此是连续地分配。

```

./src/kernel/program.cpp
330     int paddr = PAGE_DIRECTORY;
331
332     if (program->pageDirectoryAddress)
333     {
334         tss.esp0 = (int)program + PAGE_SIZE;
B+> 335         paddr = memoryManager.vaddr2paddr(program->pageDirectoryAdd
336     }
337
338     asm_update_cr3(paddr);
339 }
340
341
342

```

remote Thread 1.1 In: ProgramManager::activateProgramPage L335 PC: 0xc0021292
eax 0xc00256c0 -1073588544
ecx 0x1 1
edx 0x0 0
ebx 0x0 0
esp 0xc00245a8 0xc00245a8 <PCB_SET+3816>
ebp 0xc00245c0 0xc00245c0 <PCB_SET+3840>
esi 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--

在每一次进程调换，CPU 每次都使用中断0x80从低特权级转移到高特权级。

由上图可以看到 ebp 的值为PCB_SET+3816, esp= PCB_SET+3840, 相差 $24=4\times6$, 因此栈会把中断发生前的SS, ESP, EBP, EFLAGS, CS, EIP 6 个值推到栈上。

Assignment 2

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

fork是一个系统调用，用于创建一个新进程。被创建的进程称为子进程，调用fork的进程被称为父进程。前面已经提到，子进程是父进程的副本。父子进程共享代码段，但对于数据段、栈段等其他资源，父进程调用的fork函数会将这部分资源完全复制到子进程中。因此，对于这部分资源，父子进程并不共享。

创建新的子进程后，两个进程将从fork的返回点开始执行。这就是fork最精妙的地方，因为我们只调用了fork一次，fork却能够返回两次。同时，在父子进程的fork返回点中，fork返回的结果是不一样的，fork返回值如下：

- 在父进程中，fork返回新创建子进程的进程ID。
- 在子进程中，fork返回0。
- 如果出现错误，fork返回一个负值。

在 src/kernel/syscall.cpp 中实现fork系统调用

```
int fork() {
    return asm_system_call(2);
}

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

进入内核态后，fork的实现通过 ProgramManager::fork 来完成

fork是进程的系统调用，禁止内核线程调用<因为内核线程并没有设置 PCB::pageDirectoryAddress，所以该项为0。相反，进程有页目录表，所以该项不为0。通过判断 PCB::pageDirectoryAddress 是否为0来判断当前执行的是线程还是进程。

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

调用 ProgramManager::executeProcess 来创建一个子进程

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

找到刚刚创建的子进程，然后调用 ProgramManager::copyProcess 来复制父进程的资源到子进程中。

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

在 setup.cpp 中修改 first_process 和 first_thread 函数

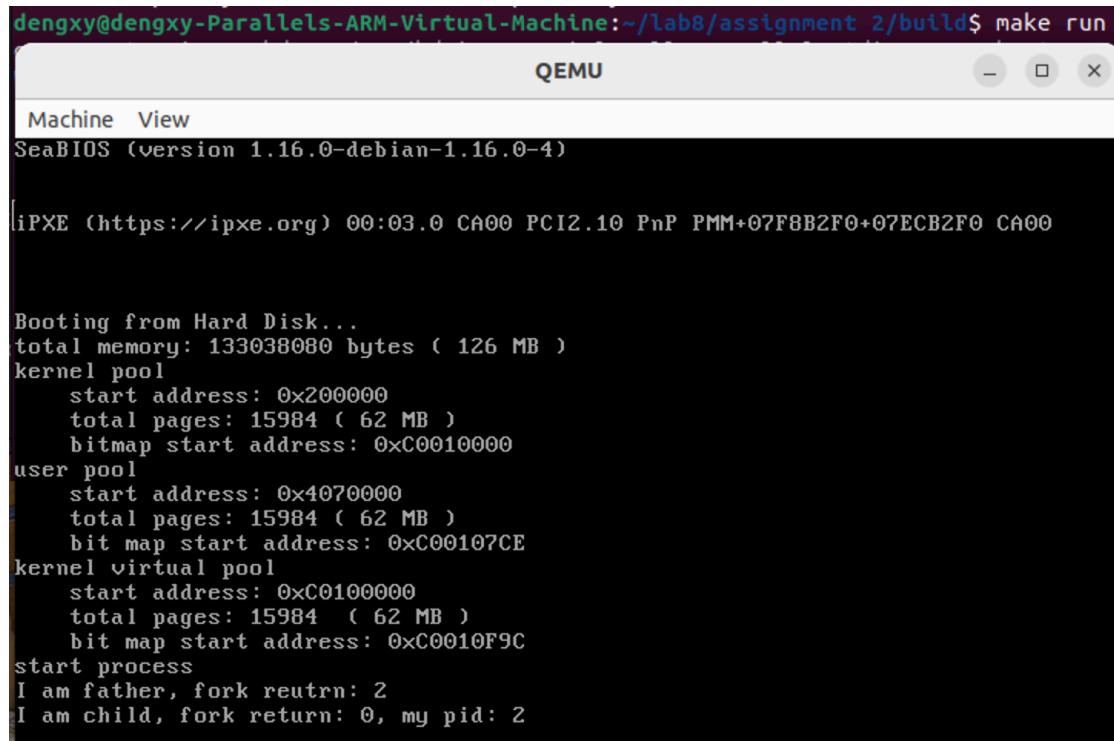
```
void first_process()
{
    int pid = fork();

    if (pid == -1)
    {
        printf("can not fork\n");
    }
    else
    {
        if (pid)
        {
            printf("I am father, fork reutrn: %d\n", pid);
        }
        else
        {
            printf("I am child, fork return: %d, my pid: %d\n", pid,
                   programManager.running->pid);
        }
    }

    asm_halt();
}

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

运行结果如图：



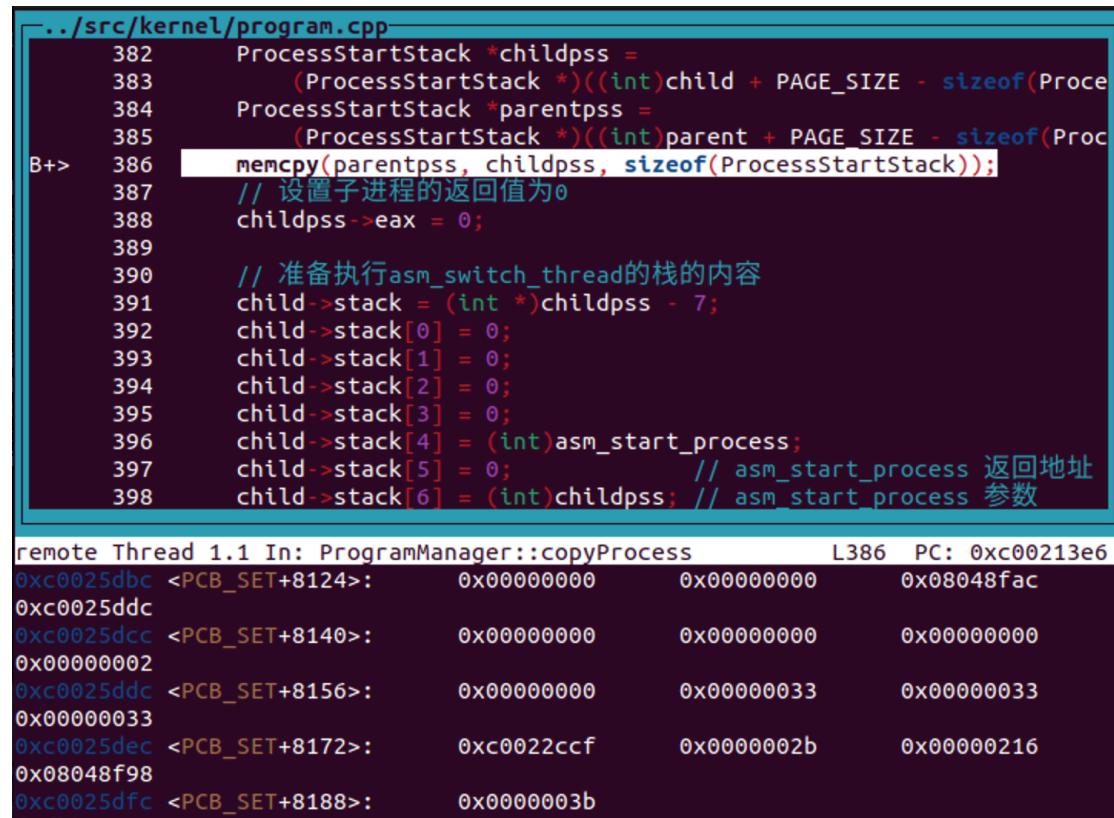
```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab8/assignment_2/build$ make run
QEMU
Machine View
SeaBIOS (version 1.16.0-debian-1.16.0-4)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
    start address: 0x200000
    total pages: 15984 ( 62 MB )
    bitmap start address: 0xC0010000
user pool
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0xC00107CE
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0xC0010F9C
start process
I am father, fork reutrn: 2
I am child, fork return: 0, my pid: 2
```

copyProcess 的作用是把中断时刻保存的寄存器的内容复制到子进程的0特权级栈中

在 ProgramManager::copyProcess 设置断点，使用 x /17x parentpss 查看父进程0级栈的内容



```
./src/kernel/program.cpp
382     ProcessStartStack *childpss =
383         (ProcessStartStack *)((int)child + PAGE_SIZE - sizeof(ProcessStartStack));
384     ProcessStartStack *parentpss =
385         (ProcessStartStack *)((int)parent + PAGE_SIZE - sizeof(ProcessStartStack));
B+> 386     memcpy(parentpss, childpss, sizeof(ProcessStartStack));
387     // 设置子进程的返回值为0
388     childpss->eax = 0;
389
390     // 准备执行asm_switch_thread的栈的内容
391     child->stack = (int *)childpss - 7;
392     child->stack[0] = 0;
393     child->stack[1] = 0;
394     child->stack[2] = 0;
395     child->stack[3] = 0;
396     child->stack[4] = (int)asm_start_process;
397     child->stack[5] = 0;          // asm_start_process 返回地址
398     child->stack[6] = (int)childpss; // asm_start_process 参数
```

remote Thread 1.1 In: ProgramManager::copyProcess	L386	PC: 0xc00213e6
0xc0025dbc <PCB_SET+8124>: 0x0000000000	0x0000000000	0x08048fac
0xc0025ddc		
0xc0025dcc <PCB_SET+8140>: 0x0000000000	0x0000000000	0x0000000000
0x0000000002		
0xc0025ddc <PCB_SET+8156>: 0x0000000000	0x0000000033	0x0000000033
0x0000000033		
0xc0025dec <PCB_SET+8172>: 0xc0022ccf	0x000000002b	0x000000216
0x08048f98		
0xc0025dfc <PCB_SET+8188>: 0x00000003b		

由上图可知，其中第16个寄存器的值为0x08048f98，第13个寄存器 eip=0xc0022ccf，在执行 memcpy(parentpss, childpss, sizeof(ProcessStack)) 后，发现 childpss 的内容和 parentpss 相同。而地址 eip=0xc0022ccf 为 asm_system_call_handler 的返回地址。

第一次执行 `asm_start_process`，在执行 `iret` 后启动进程 `first_process()`

```
./src/kernel/setup.cpp
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
B+> 32 {
33     int pid = fork();
34
35     if (pid == -1)
36     {
37         printf("can not fork\n");
38     }
39     else
40     {
41         if (pid)
```

remote Thread 1.1 In: first_process	L32	PC: 0xc00217f2
0xc00217f2	0xc00217f2 <first_process()>	
eflags	0x202	[IOPL=0 IF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51
fs	0x33	51
gs	0x0	0

一直执行，直到父进程完成为止

```

..../src/kernel/setup.cpp
46     {
47         printf("I am child, fork return: %d, my pid: %d\n", pid
48     }
49 }
50
> 51     asm_halt();
52 }
53
54 void first_thread(void *arg)
55 {
56
57     printf("start process\n");
58     programManager.executeProcess((const char *)first_process, 1);

```

remote Thread 1.1 In: first_process L51 PC: 0xc002184f

QEMU [Paused]

Machine View
SeaBIOS (version 1.16.0-debian-1.16.0-4)

iPXE (<https://ipxe.org>) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00

Booting from Hard Disk...
total memory: 133038080 bytes (126 MB)
kernel pool
 start address: 0x200000
 total pages: 15984 (62 MB)
 bitmap start address: 0xC0010000
user pool
 start address: 0x4070000
 total pages: 15984 (62 MB)
 bit map start address: 0xC00107CE
kernel virtual pool
 start address: 0xC0100000
 total pages: 15984 (62 MB)
 bit map start address: 0xC0010F9C
start process
I am father, fork reutrn: 2

执行完 first_process 部分的代码后，代码再次跳转到 asm_start_process

```

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

remote Thread 1.1 In: asm_start_process L41 PC: 0xc0022c20
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n

Breakpoint 1, asm_start_process () at ..../src/utils/asm_utils.asm:41
(gdb) 

```

在 copyProcess() 初始化子进程的0特权级栈内容如下：

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

当子进程被调度执行时，子进程能够从 `asm_switch_thread` 跳转到 `asm_start_process` 处执行。

事先把0特权栈拷贝到 `childpss` 里面，所以在启动函数 `asm_start_process` 执行了 `iret` 后会把0特权级栈的 `eip` 送入 `eip` 中，并执行该地址的代码。

继续执行代码，会跳到保存在0特权级的栈的 `eip = 0xc0022ccf`，即 `asm_system_call` 的代码地址

```
./src/utils/asm_utils.asm
143    mov esi, [ebp + 6 * 4]
144    mov edi, [ebp + 7 * 4]
145
146    int 0x80
147
> 148    pop edi
149    pop esi
150    pop edx
151    pop ecx
152    pop ebx
153    pop ebp
154
155    ret

remote Thread 1.1 In: asm system call          L148  PC: 0xc0022ccf
asm_start_process () at ./src/utils/asm_utils.asm:46
(gdb) n
asm_start_process () at ./src/utils/asm_utils.asm:47
(gdb) n
asm_start_process () at ./src/utils/asm_utils.asm:49
(gdb) n
asm_system_call () at ./src/utils/asm_utils.asm:148
(gdb) 
```

然后调用系统调用 `syscall_fork()`，跳转到 `fork()`，执行 `fork()` 调用

```

./src/kernel/syscall.cpp
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
44

```

remote Thread 1.1 In: fork L37 PC: 0xc0022375
(gdb) n
fork () at ./src/kernel/syscall.cpp:37
(gdb)

执行完 fork() 调用以后，再次跳转到 first_process

```

./src/kernel/setup.cpp
30
31 void first_process()
32 {
33     int pid = fork();
34
> 35     if (pid == -1)
36     {
37         printf("can not fork\n");
38     }
39     else
40     {
41         if (pid)
42         {

```

remote Thread 1.1 In: first_process L35 PC: 0xc0021800
fork () at ./src/kernel/syscall.cpp:37
(gdb) n
first_process () at ./src/kernel/setup.cpp:35
(gdb) print pid
\$1 = 0
(gdb) info registers eax
eax 0x0 0
(gdb)

使用 print pid 指令打印 pid 的值，发现 pid 的值为0，而观察到 eax 的值也为0，因为 copyProcess() 函数把 childpss->eax 设置为 0，从而保证子进程调用 fork() 的返回值为0。

因此子进程调用 fork() 后返回的 pid = 0，而父进程调用 fork() 返回的 pid 为新创建子进程的进程 ID。

根据结果：

```

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

```

父进程 fork() 的返回值为2，子进程 fork() 的返回值为0， pid 为2

```

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

```

从上述代码可知，数返回的 pid 是从 executeProcess("", 0) 得到，因此父进程就 programManager.fork() 返回的值。通过在 executeProcess() 里调用 executeThread()，而 executeThread 调用 allocatePCB 来分配一个子进程的 PCB。同时定义 thread->pid = ((int)thread - (int)PCB_SET) / PCB_SIZE，分配了一个子进程的 PCB，使得 programManager.fork() 可以返回子进程的 ID。对于子进程，在了 copyProcess 函数里的设定了 childpss->eax = 0 使得子进程的 fork() 返回值为 0。

Assignment 3

exit

exit 用于进程和线程的主动结束运行

```

// 第3个系统调用, exit
void exit(int ret);
void syscall_exit(int ret);

```

在进程或线程调用 exit 后，会释放其占用的所有资源，只保留 PCB。此时线程或进程的状态被标记为 DEAD。进程或线程的 PCB 由专门的线程或进程来回收。在 PCB 被回收之前的 DEAD 线程或进程也被称为 僵尸线程或僵尸进程。

exit 的实现实际上是通过 ProgramManager::exit 来完成，分为以下三步

1. 标记 PCB 状态为 DEAD 并放入返回值。
2. 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。
3. 立即执行线程/进程调度。

```

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

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

    int *pageDir, *page;
    int paddr;

```

```

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

        page = (int *) (0xffffc00000 + (i << 12));

        for (int j = 0; j < 1024; ++j)
        {
            if (!(page[j] & 0x1)) {
                continue;
            }

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

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

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

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

memoryManager.releasePages(AddressPoolType::KERNEL,
(int)program-
>userVirtual.resources.bitmap,
bitmapPages);
}
// 第三步，立即执行线程/进程调度。
schedule();
}

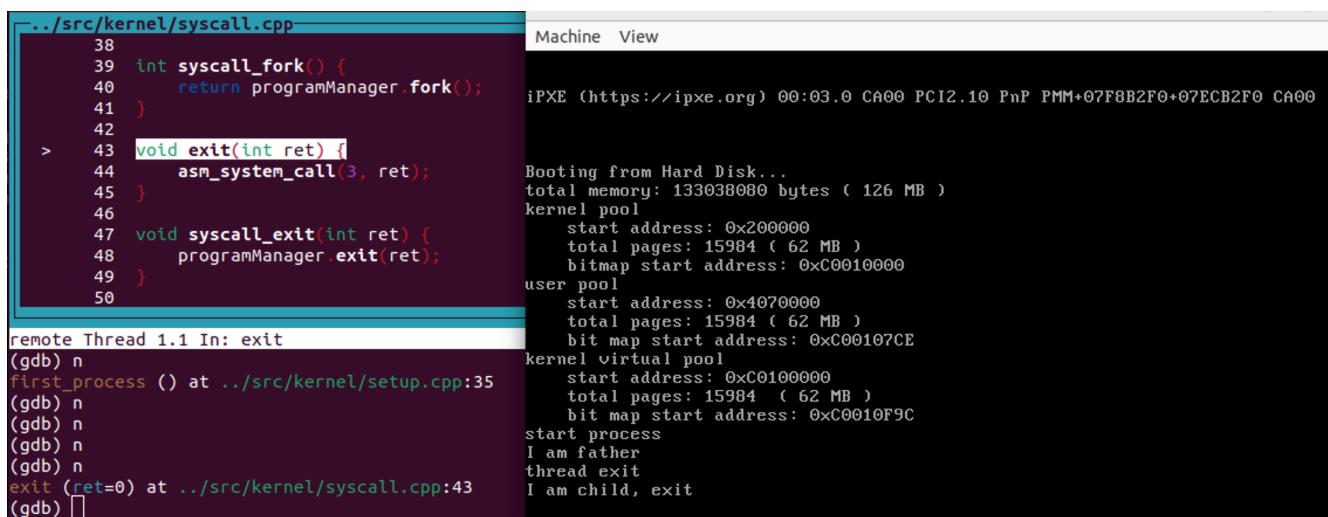
```

同时，修改 `load_process` 函数，在进程的3特权级栈的顶部放入exit的地址和参数，使得当执行进程的函数退出后就会主动跳转到exit

```
interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER,  
1);  
if (interruptStack->esp == 0)  
{  
    printf("can not build process!\n");  
    process->status = ProgramStatus::DEAD;  
    asm_halt();  
}  
interruptStack->esp += PAGE_SIZE;  
  
// 设置进程返回地址  
int *userStack = (int *)interruptStack->esp;  
userStack -= 3;  
/* 在进程的3特权级栈中的栈顶处userStack[0]放入exit的地址，CPU会认为userStack[1]是exit  
的返回地址，userStack[2]是exit的参数 */  
userStack[0] = (int)exit;  
userStack[1] = 0;  
userStack[2] = 0;  
  
interruptStack->esp = (int)userStack;
```

在 `interruptStack` 的栈依次加入exit的地址，exit的返回地址，以及exit的参数ret，使得在进程函数执行完毕后，自动跳转exit以来该进程。因为fork执行了父进程然后再执行子进程，因此最后执行的子进程执行完一定会调用 `interruptStack` 的栈调用 `exit(0)`，保证了进程退出后必定调用 `exit`，此时的 `exit` 返回值为0。

如图：在执行完子进程后，调用exit，返回值为0



The screenshot shows a terminal window with two panes. The left pane displays a GDB session on a file named `./src/kernel/syscall.cpp`. The code includes implementations for `syscall_fork()`, `void exit(int ret)`, and `void syscall_exit(int ret)`. The right pane shows the kernel boot log from iPXE, which includes memory information and pool statistics.

```
./src/kernel/syscall.cpp  
38  
39 int syscall_fork()  
40     return programManager.fork();  
41 }  
42  
> 43 void exit(int ret){  
44     asm_system_call(3, ret);  
45 }  
46  
47 void syscall_exit(int ret){  
48     programManager.exit(ret);  
49 }  
50  
remote Thread 1.1 In: exit  
(gdb) n  
first_process () at ./src/kernel/setup.cpp:35  
(gdb) n  
(gdb) n  
(gdb) n  
(gdb) n  
exit (ret=0) at ./src/kernel/syscall.cpp:43  
(gdb)   
  
Machine View  
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00  
Booting from Hard Disk...  
total memory: 133038080 bytes ( 126 MB )  
kernel pool  
    start address: 0x200000  
    total pages: 15984 ( 62 MB )  
    bitmap start address: 0xC0010000  
user pool  
    start address: 0x4070000  
    total pages: 15984 ( 62 MB )  
    bit map start address: 0xC00107CE  
kernel virtual pool  
    start address: 0xC0100000  
    total pages: 15984 ( 62 MB )  
    bit map start address: 0xC0010F9C  
start process  
I am father  
thread exit  
I am child, exit
```

完整运行：

修改 `setup.cpp` 函数

```
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");
        }
    }
}

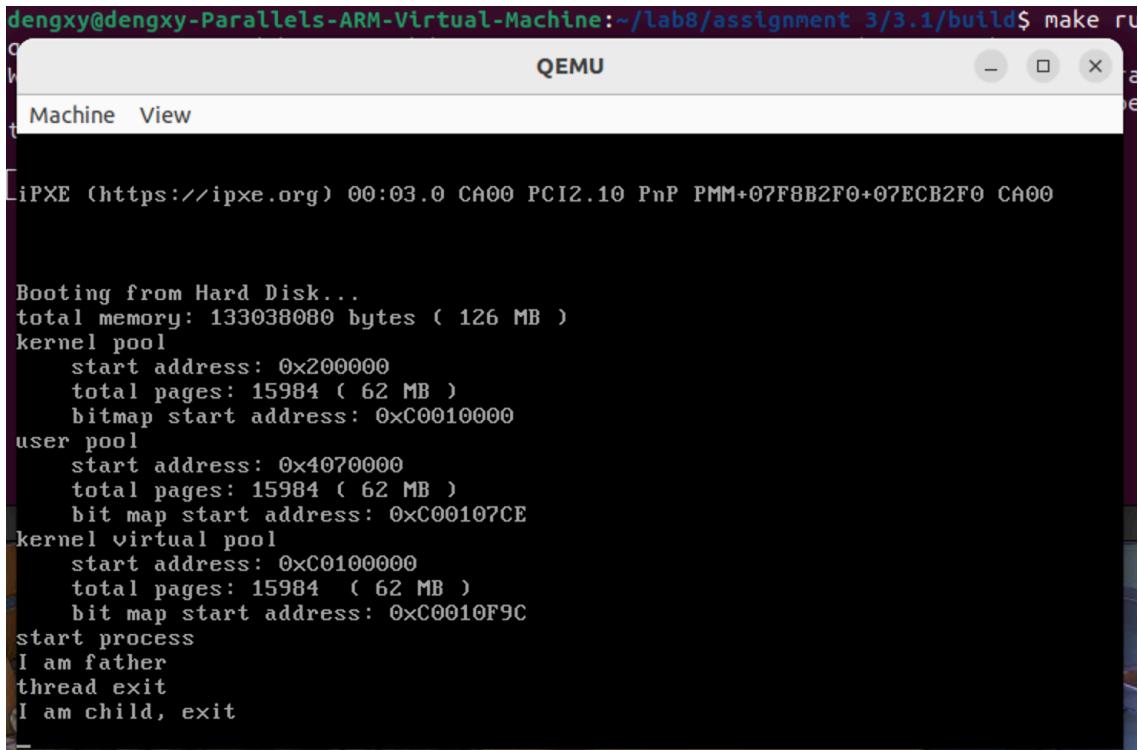
void second_thread(void *arg) {
    printf("thread exit\n");
    exit(0);
}

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

extern "C" void setup_kernel()
{
    // 设置3号系统调用
    systemService.setSystemCall(3, (int)syscall_exit);
}

```

运行结果如图：



dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab8/assignment_3/3.1/build\$ make run
QEMU
Machine View
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B2F0+07ECB2F0 CA00

Booting from Hard Disk...
total memory: 133038080 bytes (126 MB)
kernel pool
 start address: 0x200000
 total pages: 15984 (62 MB)
 bitmap start address: 0xC0010000
user pool
 start address: 0x4070000
 total pages: 15984 (62 MB)
 bit map start address: 0xC00107CE
kernel virtual pool
 start address: 0xC0100000
 total pages: 15984 (62 MB)
 bit map start address: 0xC0010F9C
start process
I am father
thread exit
I am child, exit

wait

在进程的状态被标记为DEAD后并未被清除，而是在等待其他进程来回收进程的PCB。进程的PCB是通过父进程来回收的。父进程会通过wait系统调用来等待其子进程执行完成并回收子进程。

```
// 第4个系统调用, wait  
int wait(int *retval);  
int syscall_wait(int *retval);
```

wait的参数 `retval` 用来存放子进程的返回值，如果 `retval==nullptr`，则说明父进程不关心子进程的返回值。wait的返回值是被回收的子进程的pid。如果没有子进程，则wait返回 -1。在父进程调用了wait后，如果存在子进程但子进程的状态不是 DEAD，则父进程会被阻塞，即wait不会返回直到子进程结束。

wait的实现是由 `ProgramManager` 来完成

```
int ProgramManager::wait(int *retval)  
{  
    PCB *child;  
    ListItem *item;  
    bool interrupt, flag;  
  
    while (true)  
    {  
        interrupt = interruptManager.getInterruptStatus();  
        interruptManager.disableInterrupt();  
  
        item = this->allPrograms.head.next;  
  
        // 查找子进程
```

```

//在allPrograms中找到一个状态为DEAD的子进程
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;
}
/*找到了一个可回收的子进程
当retval不为nullptr时，取出子进程的返回值放入到retval指向的变量中
取出子进程的pid，调用releasePCB来回收子进程的PCB
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;
}
else
{
    if (flag) // 子进程已经返回
    {
        interruptManager.setInterruptStatus(interrupt);
        return -1;//没有找到子进程，返回-1
    }
    else // 存在子进程，但子进程的状态不是DEAD，执行调度
    {
        interruptManager.setInterruptStatus(interrupt);
        schedule();
    }
}
}
}

```

当父进程调用wait后，如果存在子进程但子进程的状态不是 DEAD，则父进程会被阻塞，即wait不会返回直到子进程结束。

修改 schedule 函数

```
void ProgramManager::schedule()
{
    ...
    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
        if (!running->pageDirectoryAddress) {
            releasePCB(running);
        }
    }
    ...
}
```

修改 setup.cpp 来测试 wait() 函数

```
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();
        }
        //子进程2
    }
}
```

```

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

//子进程1
else
{
    uint32 tmp = 0xffffffff;
    while (tmp)
        --tmp;
    printf("exit, pid: %d\n", programManager.running->pid);
    exit(-123);
}
}

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

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

extern "C" void setup_kernel()
{
    ...
    // 设置4号系统调用
    systemService.setSystemCall(4, (int)syscall_wait);
    ...
}

```

运行结果如图：

```
dengxy@dengxy-Parallels-ARM-Virtual-Machine:~/lab8/assignment 3/3.2/build$ make run  
qemu-system-i386 ./run/hd.img -serial null -parallel stdio -no-reboot  
WARNING: Using /dev/null as a serial port.  
QEMU  
Machine View  
at:  
Booting from Hard Disk...  
total memory: 133038080 bytes ( 126 MB )  
kernel pool  
    start address: 0x200000  
    total pages: 15984 ( 62 MB )  
    bitmap start address: 0xC0010000  
user pool  
    start address: 0x4070000  
    total pages: 15984 ( 62 MB )  
    bit map start address: 0xC00107CE  
kernel virtual pool  
    start address: 0xC0100000  
    total pages: 15984 ( 62 MB )  
    bit map start address: 0xC0010F9C  
start process  
thread exit  
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
```

首先执行子进程1，打印 `exit, pid: 3`，子进程1的ID为3，调用 `exit(-123)`，结束子进程1，并将其 PCB status 设置为 DEAD，返回值 `retVal=123`

接着执行子进程2，打印 `exit, pid: 4`，子进程2的ID位4，调用 `exit(123934)`，结束子进程2，并将其 PCB status 设置为 DEAD， `retVal=123934`

父进程的代码中有 `while((pid = wait(&retval)) != -1)`，说明 `wait(&retval)` 只要有子进程的状态不为 DEAD，它都一直不会返回，因此如果存在状态不为 DEAD 的子进程的情况，一直都是死循环的。

对应运行结果，发现父进程在子进程1和子进程2都执行完毕的情况下以后，才会执行while循环里面的语句，并且对于可返回的进程进行回收。如果没有子进程的状态都是 dead 的话，代表所有的子进程都已经通过之前while语句 `wait(&retVal)` 被回收。现在执行 `wait(&retval) = -1`，跳出 while 语句，打印 `all child process exit, programs: 2`，代表所有的子进程（包括子进程的子进程）都被回收，其中数字 2代表子进程的总数量。

由此通过 `exit` 和 `wait` 系统调用来实现所有子进程在执行父进程前进行释放，从而避免子进程成为孤儿进程，也解决了僵尸进程的问题。

3、关键代码

Assignment 1

```
asm_system_call:  
    push ebp  
    mov ebp, esp  
  
    push ebx  
    push ecx  
    push edx
```

```

push esi
push edi
push ds
push es
push fs
push gs

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]

int 0x80

pop gs
pop fs
pop es
pop ds
pop edi
pop esi
pop edx
pop ecx
pop ebx
pop ebp

ret

```

Assignment 2

```

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

    // 准备执行asm_switch_thread的栈的内容
    child->stack = (int *)childpss - 7;
}

```

```

child->stack[ 0 ] = 0;
child->stack[ 1 ] = 0;
child->stack[ 2 ] = 0;
child->stack[ 3 ] = 0;
child->stack[ 4 ] = (int)asm_start_process;
child->stack[ 5 ] = 0;           // asm_start_process 返回地址
child->stack[ 6 ] = (int)childpss; // asm_start_process 参数

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

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

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

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

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

// 复制页目录表

```

```

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

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

    // 页目录项
    int pde = parentPageDir[i];
    // 构造页表的起始虚拟地址
    int *pageTableVaddr = (int *) (0xffc00000 + (i << 12));

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

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

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

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

    // 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
}

}

// 归还从内核分配的中转页
memoryManager.releasePages(AddressPoolType::KERNEL, (int)buffer, 1);
return true;
}

```

Assignment 3

```

exit

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

    // 第一步，标记PCB状态为`DEAD`并放入返回值。

```

```

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

int *pageDir, *page;
int paddr;

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

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

        for (int j = 0; j < 1024; ++j)
        {
            if (!(page[j] & 0x1)) {
                continue;
            }

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

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

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

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

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

```

```
}

// 第三步，立即执行线程/进程调度。
schedule();
}
```

wait

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

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

        item = this->allPrograms.head.next;

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

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

            int pid = child->pid;
            releasePCB(child);
        }
    }
}
```

```
        interruptManager.setInterruptStatus(interrupt);
        return pid;
    }
    else
    {
        if (flag) // 子进程已经返回
        {

            interruptManager.setInterruptStatus(interrupt);
            return -1;
        }
        else // 存在子进程，但子进程的状态不是DEAD
        {
            interruptManager.setInterruptStatus(interrupt);
            schedule();
        }
    }
}
```

4、总结

1. 学习了如何实现系统调用，了解了系统调用的基本原理，通过使用gdb调试工具深入分析系统调用的栈的变化和TSS的作用。
2. 学习了fork函数的基本原理和实现方式，借助gdb调试工具，深入分析了子进程的执行流程和数据寄存器、段寄存器的变化以及父子进程之间执行的差异。
3. 学习了wait和exit函数的基本原理和实现方式，了解进程状态和进程的退出机制，在实际的操作过程中，必须注意子进程的回收问题，避免出现孤儿进程和僵尸进程。