



本科生实验报告

学生姓名： 丁晓琪

学生学号： 22336057

专业名称： 计科

Assignment1

- 1.实验要求
- 2.实验过程
- 3.关键代码
- 4.实验结果
- 5.总结

Assignment2

- 1.实验要求
- 2.实验过程
- 3.关键代码
- 4.实验结果
- 5.总结

Assignment3

- 1.实验要求
- 2.实验过程
- 3.关键代码
- 4.实验结果
- 5.总结

Assignment1

1.实验要求

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，结果截图且说明实现思路。
- 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.实验过程

1. 实现系统调用

- 实现系统调用服务类，初始化系统调用表
- 实现系统调用入口函数
- 实现系统调用处理函数

2. 实现用户进程：

- 进程创建前的准备：在跳转到内核前开启分页机制，将内核的虚拟地址提升到3GB-4GB
- 初始化TSS和用户段描述符
- 进程创建：创建PCB，初始化页目录表，初始化虚拟池
- 进程调度

3. 添加print系统调用

4. gdb调试（在实验结果中）

3.关键代码

3.1实现系统调用

- 实现系统调用服务类：
 - 设置系统调用表： `system_call_table` 表项为各种系统调用的处理函数
 - 系统调用服务类的初始化： `initialize()` 初始化系统调用表，且为系统调用中断0x80设置中断描述符。中断描述符的DPL描述需要的特权级，设置为3，能让用户进程调用。
 - 系统调用的设置： `setSystemCall` 将系统调用的处理执行函数放入到系统调用表对应索引位置

```
1  int system_call_table[MAX_SYSTEM_CALL];
2
3  SystemService::SystemService() {
4      initialize();
5  }
6
7  void SystemService::initialize()
8  {
9      memset((char *)system_call_table, 0, sizeof(int) * MAX_SYSTEM_CALL);
10     // 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
11     interruptManager.setInterruptDescriptor(0x80,
12     (uint32)asm_system_call_handler, 3);
13 }
14 bool SystemService::setSystemCall(int index, int function)
15 {
16     system_call_table[index] = function;
17     return true;
18 }
```

- 实现系统调用入口函数 `asm_system_call`：

- 说明：可以由用户进程直接调用，提供一个处理所有系统调用的接口，提供一些系统调用前的准备，如保护当前进程的现场。执行时用户进程仍然会在特权级3，未跳转到内核态
- 步骤：保护现场，转移系统调用参数到寄存器，通过0x80中断跳转系统调用服务函数，系统调用结束返回恢复现场
- 注意：用户进程传入的系统调用参数在这里要进一步处理，一般由C++函数传入汇编函数的参数是在栈上取得，但是这里需要将栈上的参数暂时转移到寄存器再传入系统调用服务函数。经过中断跳入0x80的中断函数也就是系统调用服务函数时，tss自动加载，此时已经从用户态转为内核态了，栈已经从用户栈转为内核栈，参数是无法再从用户栈获得了

```

1      asm_system_call:
2          push ebp
3          mov ebp, esp
4          push ebx
5          push ecx
6          push edx
7          push esi
8          push edi
9          push ds
10         push es
11         push fs ;保存现场
12         push gs
13         mov eax, [ebp + 2 * 4] ;保存系统调用号
14         mov ebx, [ebp + 3 * 4] ;保存五个参数
15         mov ecx, [ebp + 4 * 4]
16         mov edx, [ebp + 5 * 4]
17         mov esi, [ebp + 6 * 4]
18         mov edi, [ebp + 7 * 4]
19         int 0x80 ;调用0x80，会根据eax的系统调用号来调用不同的函数
20         pop gs ;恢复现场
21         pop fs
22         pop es
23         pop ds
24         pop edi
25         pop esi
26         pop edx
27         pop ecx
28         pop ebx
29         pop ebp
30         ret

```

- 0x80中断对应的系统处理函数
 - 为什么用中断的形式实现系统调用
 - 用户进程系统调用涉及到特权转换，系统调用的处理都在内核态
 - 中断处理提供特权转换
 - 1.通过中断向量找到中断描述符
 - 2.处理器会检查中断描述符中的DPL是否满足条件（目标代码段DPL≤CPL&&CPL≤中断描述符DPL）
 - 3.检查通过后，处理器加载目标代码段的选择子到CS，包含RPL（请求特权级），特权级转换
 - 4.中断服务处理结束后，iret返回且切换回原来的特权

(0x80的中断描述符的DPL设置为3, 在前面实验中内核的段选择子的RPL都默认被设置为0)

○ 步骤:

- 1.保护内核态跳转真正的系统调用处理函数前的现场
- 2.修改特权级转换后的ds, es, fs, gs寄存器
- 3.将 `asm_system_call` 放在寄存器中传入的参数压栈, 通过栈传到系统调用处理函数
- 4.根据系统调用号在系统调用表中找函数地址跳转执行
- 5.恢复现场, 且将系统调用处理函数的返回值放在eax中

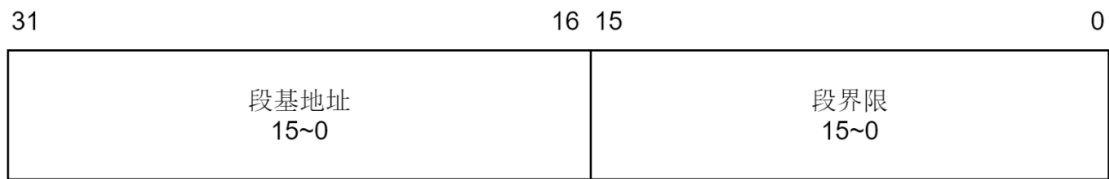
```
1  asm_system_call_handler:
2      push ds
3      push es
4      push fs
5      push gs
6      pushad
7      push eax
8      ; 栈段会从tss中自动加载
9      mov eax, DATA_SELECTOR
10     mov ds, eax
11     mov es, eax
12     mov eax, VIDEO_SELECTOR
13     mov gs, eax
14     pop eax
15     ; 参数压栈
16     push edi
17     push esi
18     push edx
19     push ecx
20     push ebx
21     sti
22     call dword[system_call_table + eax * 4]
23     cli
24     add esp, 5 * 4
25     mov [ASM_TEMP], eax
26     popad
27     pop gs
28     pop fs
29     pop es
30     pop ds
31     mov eax, [ASM_TEMP]
32
33     iret
```

3.2实现用户进程

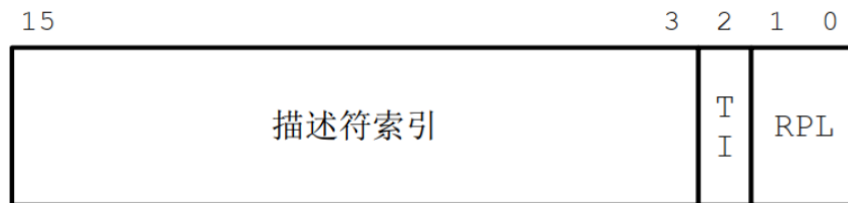
- 初始化TSS和用户段描述符

1. 初始化用户段描述符

段描述符:



段选择子:



```

1  int selector;
2
3  selector = asm_add_global_descriptor(USER_CODE_LOW, USER_CODE_HIGH);
4  USER_CODE_SELECTOR = (selector << 3) | 0x3;
5
6  selector = asm_add_global_descriptor(USER_DATA_LOW, USER_DATA_HIGH);
7  USER_DATA_SELECTOR = (selector << 3) | 0x3;
8
9  selector = asm_add_global_descriptor(USER_STACK_LOW,
10 USER_STACK_HIGH);
11  USER_STACK_SELECTOR = (selector << 3) | 0x3;

```

```

1  ;添加段描述符
2  asm_add_global_descriptor:
3      push ebp
4      mov ebp, esp
5      push ebx
6      push esi
7      sgdt [ASM_GDTR] ;将GDTR的内容读到ASM_GDTR中
8      mov ebx, [ASM_GDTR + 2] ; GDT地址, ASM--GDTR移动高2个字节, 读取32位
   的GDT地址
9      xor esi, esi
10     mov si, word[ASM_GDTR] ; GDT界限, 读取16位的界限
11     add esi, 1
12     mov eax, [ebp + 2 * 4] ; low 用户段描述符低32位
13     mov dword [ebx + esi], eax ;把段描述符写入
14     mov eax, [ebp + 3 * 4] ; high, 用户段描述符高32位
15     mov dword [ebx + esi + 4], eax ;把段描述符写入
16     mov eax, esi

```

```

17     shr eax, 3 ;计算段描述符在段表中偏移, 这个变成返回数据, 是基于段描述符一个
    一个的索引, 右移3位表示除8, 一个段描述符8个字节
18     add word[ASM_GDTR], 8
19     lgdt [ASM_GDTR]
20     pop esi
21     pop ebx
22     pop ebp
23     ret

```

2. 初始化TSS

- TSS作用: 保存和恢复关键寄存器栈信息, 支持特权级的安全切换
- 内容: 不同特权级的栈基地址寄存器ss, 和栈顶地址寄存器, 还有其他的重要寄存器保护特权级切换时的现场 (代码略)
- 初始化:

```

1 void ProgramManager::initializeTSS()
2 {
3
4     int size = sizeof(TSS);
5     int address = (int)&tss; //TSS地址
6
7     memset((char *)address, 0, size);
8     tss.ss0 = STACK_SELECTOR; // 内核态堆栈段选择子
9
10    int low, high, limit;
11
12    limit = size - 1;
13    low = (address << 16) | (limit & 0xff); //段基地址+界限
14    // DPL = 0 段基地址高8位 中8位
15    // 段界限3位 段相关设置位
16    high = (address & 0xff000000) | ((address & 0x00ff0000) >>
17    16) | ((limit & 0xff00) << 16) | 0x00008900;
18
19    int selector = asm_add_global_descriptor(low, high); //段描述
    符写入
20    // RPL = 0
21    asm_ltr(selector << 3); //tss段选择子写入TR寄存器, 处理器特权切换时
    自动加载TR寄存器里面的tss
22    tss.ioMap = address + size; //
23 }

```

• 进程的创建:

1. 总: 创建进程的PCB, 用户进程还需要在内核线程的基础上增加用户空间中的项目录表 and 用户空间虚拟地址池

```

1 int ProgramManager::executeProcess(const char *filename, int priority)
2 {
3     bool status = interruptManager.getInterruptStatus();
4     interruptManager.disableInterrupt();
5     // 在线程创建的基础上初步创建进程的PCB
6     int pid = executeThread((ThreadFunction)load_process,
7                             (void *)filename, filename, priority); //先像
    创建一个线程一样创建进程, 线程函数是load_process

```

```

8     if (pid == -1)
9     {
10         interruptManager.setInterruptStatus(status);
11         return -1;
12     }
13     // 找到刚刚创建的PCB
14     PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
15     // 创建进程的页目录表
16     process->pageDirectoryAddress = createProcessPageDirectory();
17     if (!process->pageDirectoryAddress)
18     {
19         process->status = ProgramStatus::DEAD;
20         interruptManager.setInterruptStatus(status);
21         return -1;
22     }
23     // 创建进程的虚拟地址池
24     bool res = createUserVirtualPool(process);
25
26     if (!res)
27     {
28         process->status = ProgramStatus::DEAD;
29         interruptManager.setInterruptStatus(status);
30         return -1;
31     }
32     interruptManager.setInterruptStatus(status);
33     return pid;
34 }
35

```

2. 创建页目录表:

- 创建原因: 进程有自己的虚拟地址空间和分页机制
- 注意2: 定义好用户的虚拟空间地址3-4GB和内核的虚拟空间地址3-4GB时共享的, 需要将用户的虚拟地址映射到内核上, 使进程在用户进程的用户空间虚拟地址和内核虚拟地址不冲突的情况下能够访问到内核资源。(用户进程进入内核态后用的还是自己的虚拟地址空间, 根据用户进程的页目录表转换物理地址而不是根据内核的页目录表, 如果虚拟地址为0-3GB访问的还是用户地址空间, 但是为3GB-4GB时访问的是内核地址空间0。为什么用户进程进到内核能无障碍访问, 也是因为内核空间定义的时候, 也是将虚拟地址映射到了3GB-4GB, 并且内核虚拟地址3GB-4GB和用户的映射的物理内容相同。)
- 操作: 用户进程页目录表的第768项到第1022项和内核页目录表的第768项到第1022项一致

```

1  int ProgramManager::createProcessPageDirectory()
2  {
3      // 从内核地址池中分配一页存储用户进程的页目录表
4      int vaddr = memoryManager.allocatePages(AddressPoolType::KERNEL,
5      1);
6      if (!vaddr)
7      {
8          //printf("can not create page from kernel\n");
9          return 0;
10     }
11     memset((char *)vaddr, 0, PAGE_SIZE);

```

```

12
13     // 复制内核目录项到虚拟地址的高1GB
14     //现在还是在内核态中，用的是内核虚拟地址，0xffffffff000是内核页目录的起始地址
15     int *src = (int *) (0xffffffff000 + 0x300 * 4); //0x300大小为768，目的
    要将内核目录表的768项到最后（3-4GB）复制给用户页目录表的（3-4GB），打通内核与用
    户共享
16     int *dst = (int *) (vaddr + 0x300 * 4);
17     for (int i = 0; i < 256; ++i)
18     {
19         dst[i] = src[i];
20     }
21
22     // 用户进程页目录表的最后一项指向用户进程页目录表本身
23     ((int *)vaddr)[1023] = memoryManager.vaddr2paddr(vaddr) | 0x7; //
    用户页目录表本身在内核地址池中
24
25     return vaddr;
26 }
27

```

3. 创建虚拟地址池

```

1 bool ProgramManager::createUserVirtualPool(PCB *process)
2 {
3     int sourcesCount = (0xc0000000 - USER_VADDR_START) / PAGE_SIZE;
    //计算出用户虚拟空间所占页
4     int bitmapLength = ceil(sourcesCount, 8);
5     // 计算位图所占的页数
6     int pageCount = ceil(bitmapLength, PAGE_SIZE);
7     //分配位图所需空间，也是从kernel中分，但是不需要报备用户页目录表的（用
    户空间的情况）
8     int start = memoryManager.allocatePages(AddressPoolType::KERNEL,
    pageCount);
9     if (!start)
10     {
11         return false;
12     }
13     //初始化位图分配得到的空间
14     memset((char *)start, 0, PAGE_SIZE * pageCount);
15     //初始化地址池
16     (process->userVirtual).initialize((char *)start, bitmapLength,
    USER_VADDR_START);
17
18     return true;
19 }

```

4. 进程启动时的加载函数 load_process

- 注意：这里也涉及到特权级的转换，load_process 是为了完成这个特权转换的过程。
- 步骤：将用户进程用户态的相关信息先保存在 interruptStack 中，跳转到 asm_start_process 中根据 interruptStack 更新寄存器，通过 iret 返回切换特权级且执行用户进程函数。interruptStack 应该是挂靠在进程PCB的特权级0栈

```

1 void load_process(const char *filename) //filename是要跳转执行的某个
    函数的地址，这里简化了从磁盘加载程序

```



```

2  {
3      interruptManager.disableInterrupt();
4
5      PCB *process = programManager.running;
6      ProcessStartStack *interruptStack =
7          (ProcessStartStack *)((int)process + PAGE_SIZE -
sizeof(ProcessStartStack));
8
9      interruptStack->edi = 0;
10     interruptStack->esi = 0;
11     interruptStack->ebp = 0;
12     interruptStack->esp_dummy = 0;
13     interruptStack->ebx = 0;
14     interruptStack->edx = 0;
15     interruptStack->ecx = 0;
16     interruptStack->eax = 0;
17     interruptStack->gs = 0;
18
19     interruptStack->fs = programManager.USER_DATA_SELECTOR;
20     interruptStack->es = programManager.USER_DATA_SELECTOR;
21     interruptStack->ds = programManager.USER_DATA_SELECTOR;    //
初始化栈
22
23     interruptStack->eip = (int)filename;
24     interruptStack->cs = programManager.USER_CODE_SELECTOR;    //
用户模式平坦模式
25     interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1); //
IOPL, IF = 1 开中断, MBS = 1 默认
26
27     interruptStack->esp =
memoryManager.allocatePages(AddressPoolType::USER, 1);
28     if (interruptStack->esp == 0)
29     {
30         printf("can not build process!\n");
31         process->status = ProgramStatus::DEAD;
32         asm_halt();
33     }
34     interruptStack->esp += PAGE_SIZE;
35     interruptStack->ss = programManager.USER_STACK_SELECTOR;
36
37     asm_start_process((int)interruptStack); //中断返回进程执行函数
38 }

```

```

1  asm_start_process:
2      ;jmp $
3      mov eax, dword[esp+4] ;取出开始栈的起始地址
4      mov esp, eax ;将当前栈转为开始栈
5      popad
6      pop gs;
7      pop fs;
8      pop es;
9      pop ds;
10
11     iret ;根据开始栈的内容更新寄存器, 特权级3的选择子被放入到段寄存器中, 代码跳
转到进程的起始处执行。

```

5. 进程调度

- 说明：进程和线程调度区别根本在于进程执行时在用户态，需要进程特权级别切换，并且进程有自己的页目录表。则进程切换调度时，不能简单地切换PCB，而是要更新TSS中对应的内核态栈和更新CR3中进程页目录表的地址

```
1 void ProgramManager::activateProgramPage(PCB *program)
2 {
3     int paddr = PAGE_DIRECTORY;
4
5     if (program->pageDirectoryAddress) //如果是用户态进程要加载tts
6     {
7         tss.esp0 = (int)program + PAGE_SIZE;
8         paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);
9     }
10
11     asm_update_cr3(paddr); //更新页目录表
12 }
```

3.3. 添加系统调用

添加打印字符串系统调用

```
1 systemService.setSystemCall(1, (int)syscall_1);
2 int syscall_1(int first, int second, int third, int forth, int fifth){
3     char* print=(char*) first;
4     printf("%s", print);
5 }
6 void first_process()
7 {
8     char* print="hello world\n";
9     int help_print=(int)print;
10    asm_system_call(0, 132, 324, 12, 124);
11    asm_system_call(1, help_print, 0, 0, 0, 0);
12    asm_halt();
13 }
```

4. 实验结果

1.

```

QEMU
Machine View

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

```

2.根据gdb来分析执行系统调用后的栈的变化情况和TSS的作用

- 系统调用前:

```

37         char* print= "hello world\n";
38         int help_print=(int)print;
B+>39         asm system call(0, 132, 324, 12, 124);
40         asm_system_call(1,help_print,0,0,0,0);
41         asm_halt();
42     }
43
44     void first_thread(void *arg)
45     {
46         printf("start process\n");
47         programManager.executeProcess((const char *)first_process, 1);
48         programManager.executeProcess((const char *)first_process, 1);
49         programManager.executeProcess((const char *)first_process, 1);
50         asm_halt();
51     }
52
53     extern "C" void setup_kernel()
54     {
55
56         // ^$^ &^ '^ '^ %^
57         interruptManager.initialize();
58         interruptManager.enableTimeInterrupt();
59         interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler)
60
61         // ^(^ ^ %^ '^ '^ %^
62         stdio.initialize();

```

remote Thread 1.1 In: first_process		
eax	0xc0022a3f	-1073599937
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048fe4	0x8048fe4
ebp	0x8048ffc	0x8048ffc
esi	0x0	0
edi	0x0	0
eip	0xc0020ae1	0xc0020ae1 <first_process()+23>
eflags	0x206	[IOPL=0 IF PF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51
fs	0x33	51
gs	0x0	0
fs_base	0x0	0
gs_base	0x0	0
k_gs_base	0x0	0
cr0	0x80000011	[PG ET PE]
cr2	0x0	0
cr3	0x200000	[PDBR=0 PCID=0]

用户态栈地址，不是特权级栈
特权级栈还包括很多寄存器的信息

系统调用前用户进程的CPL为3 (cs寄存器的低两位)

- 系统调用入口函数:

```

145
>146      int 0x80
147
148      pop edi
149      pop esi
150      pop edx
151      pop ecx
152      pop ebx
153      pop ebp
154
155      ret
156
157      ; void asm_init_page_reg(int *directory);
158      asm_init_page_reg:
159      push ebp

```

```

remote Thread 1.1 In: asm_system_call
eax      0x0      0
ecx      0x144    324
edx      0xc      12
ebx      0x84     132
esp      0x8048fa8 0x8048fa8
ebp      0x8048fbc 0x8048fbc
esi      0x7c     124
edi      0x0      0
eip      0xc00228ad 0xc00228ad <asm_system_call+26>
eflags   0x212    [ IOPL=0 IF AF ]
cs       0x2b     43
ss       0x3b     59
ds       0x33     51
es       0x33     51
fs       0x33     51
gs       0x0      0
fs_base  0x0      0
gs_base  0x0      0
k_gs_base 0x0      0
cr0      0x80000011 [ PG ET PE ]
cr2      0x0      0
cr3      0x200000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--

```

- 系统调用服务函数：TSS重要性在int 0x80中断发生内核态转换时，帮助切换栈和其他信息

```

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         ; ^^ &^ $^ $^ ss^$^ (^ %^ %^ (^
97
98         mov eax, DATA_SELECTOR
99         mov ds, eax
100        mov es, eax
101
102        mov eax, VIDEO_SELECTOR
103        mov gs, eax
104
remote Thread 1.1 In: asm system call handler
eax      0x0      0      ← 通过寄存器传入的参数
ecx      0x144    324
edx      0xc      12
ebx      0x84     132
esp      0xc0025888 0xc0025888 <PCB_SET+8168> ← 栈换为内核态上的栈
ebp      0x8048fbc 0x8048fbc
esi      0x7c     124
edi      0x0      0
eip      0xc0022858 0xc0022858 <asm_system_call_handler+1>
eflags   0x12     [ IOPL=0 AF ]
cs       0x20     32
ss       0x10     16 ←
ds       0x33     51  CPL为0, 特权级为0, 经过中断进入
es       0x33     51  到了内核态
fs       0x33     51
gs       0x0      0
fs_base  0x0      0
gs_base  0x0      0
k_gs_base 0x0      0
cr0      0x80000011 [ PG ET PE ]
cr2      0x0      0
cr3      0x200000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--

```

- 调用系统处理函数返回：

```

124        pop es
125        pop ds
126        mov eax, [ASM_TEMP]
127
>128        iret
129     asm_system_call:
130        push ebp
131        mov ebp, esp

remote Thread 1.1 In: asm system call handler
eax      0x250    592 ← 函数返回参数
ecx      0x144    324
edx      0xc      12
ebx      0x84     132
esp      0xc002588c 0xc002588c <PCB_SET+8172>
ebp      0x8048fbc 0x8048fbc
esi      0x7c     124
edi      0x0      0
eip      0xc0022892 0xc0022892 <asm_system_call_handler+59>
eflags   0x86     [ IOPL=0 SF PF ]
cs       0x20     32 ← 仍然在内核态, 特权级为0
ss       0x10     16
ds       0x33     51
es       0x33     51
fs       0x33     51
gs       0x0      0
fs_base  0x0      0
gs_base  0x0      0
k_gs_base 0x0      0
cr0      0x80000011 [ PG ET PE ]
cr2      0x0      0
cr3      0x200000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--

```

- iret中断返回，返回到系统调用入口函数：特权级切换回用户态

```
148      pop edi
149      pop esi
150      pop edx
151      pop ecx
152      pop ebx
153      pop ebp
154
155      ret
156
157      ; void asm_init_page_reg(int *directory);
158      asm_init_page_reg:
159          push ebp
160          mov ebp, esp
161
162          push eax
163
remote Thread 1.1 In: asm_system_call
eax      0x250      592
ecx      0x144      324
edx      0xc        12
ebx      0x84       132
esp      0x8048fa8  0x8048fa8 ← 切换回用户态的栈
ebp      0x8048fbc  0x8048fbc
esi      0x7c       124
edi      0x0        0
eip      0xc00228af 0xc00228af <asm_system_call+28>
eflags   0x212      [ IOPL=0 IF AF ]
cs       0x2b       43 ←
ss       0x3b       59
ds       0x33       51    CPL为3, 特权级为3, 返回用户态
es       0x33       51
fs       0x33       51
gs       0x0        0
fs_base  0x0        0
gs_base  0x0        0
k_gs_base 0x0        0
cr0      0x80000011 [ PG ET PE ]
cr2      0x0        0
cr3      0x200000   [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

- 系统调用返回:

```

38         tnt_help_print=(tnt)print,
B+ 39         asm system call(0, 132, 324, 12, 124);
>40         asm system call(1,help_print,0,0,0,0);
41         asm_halt();
42     }
43
44     void first_thread(void *arg)
45     {
46         printf("start process\n");
47         programManager.executeProcess((const char *)first_process, 1);
48         programManager.executeProcess((const char *)first_process, 1);
49         programManager.executeProcess((const char *)first_process, 1);
50         asm_halt();
51     }
52
53     extern "C" void setup_kernel()
54     {
55
remote Thread 1.1 In: first_process
eax          0x250          592
ecx          0x0           0
edx          0x0           0
ebx          0x0           0
esp          0x8048fe4      0x8048fe4
ebp          0x8048ffc      0x8048ffc
esi          0x0           0
edi          0x0           0
eip          0xc0020afe      0xc0020afe <first_process()+52>
eflags       0x206          [ IOPL=0 IF PF ]
cs           0x2b          43
ss           0x3b          59
ds           0x33          51
es           0x33          51
fs           0x33          51
gs           0x0           0
fs_base      0x0           0
gs_base      0x0           0
kgs_base     0x0           0
cr0          0x80000011      [ PG ET PE ]
cr2          0x0           0
cr3          0x200000        [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--

```

- TSS作用：

每次特权级切换，处理器从TSS中加载出对应特权级的栈信息

5.总结

- 为什么中断会导致特权级的切换：[前文跳转](#)
- 为什么需要把用户进程的3GB-4GB映射到内核：[前文跳转2](#)
- 特权级切换实现：
 - 系统调用：通过中断和中断返回实现，调用0x80时就已经自动加载TSS转换0特权栈和段选择子了
(这里的特权栈0就是挂载在进程PCB上面的初始栈（具体见下面用户进程调度特权级跳转），但是在用户进程调度时特权级0栈在 `load_process` 里初始化满了用户态的信息，但是在 `asm_start_process` 里又全部pop出去了，现在这个栈是干净的)
 - 用户进程调度特权级跳转：
 - 进程调度时先进入 `activateProgramPage`，先把特权级0栈指定到进程PCB的开始（`tss.esp0 = (int)program + PAGE_SIZE;`）
 - 然后到 `asm_switch_thread` 切换进程，跳转到被调度进程的函数地址 `load_process` 执行（什么时候指定的：在 `executeProcess` 中调用 `executeThread` 指定的）

- 在 `load_process` 中，先初始化了PCB的初始栈 `interruptStack`（其实就是前面指定挂靠的特权0栈（`ProcessStartStack *interruptStack =(ProcessStartStack *) ((int)process + PAGE_SIZE - sizeof(ProcessStartStack));`），然后跳转到 `asm_start_process`
- 在 `asm_start_process` 中把实现的初始栈也就是特权0栈的内容全部弹出，`iret`返回切换用户态

（`iret`为什么能切换回用户态：在 `asm_start_process` 已经把当前的栈 `esp` 转到初始栈的位置了，`iret`会将当前栈的段地址和偏移地址弹出（会恢复CS，IP，EFLAGS等等），而这些关于段的寄存器已经在初始栈变更为用户态下的了。`iret`可以到用户态去）

Assignment2

1.实验要求

- 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。

2.实验过程

1. fork()实现

- 添加fork()系统调用（同Assignment1添加，具体看fork函数的实现）
- 创建子进程
- 复制父进程的资源到子进程：0特权级栈，PCB，虚拟地址池，页目录表，页表及其指向的物理页

2. GDB调试（实验结果中）

3.关键代码

3.1总：fork()函数

- 创建子进程：没有为子进程传入进程函数，后续可以将子进程的内核PCB栈中的函数地址改为父进程的进程函数地址
- 子进程初始化：为子进程复制父进程资源

```
1  int ProgramManager::fork()
2  {
3      bool status = interruptManager.getInterruptStatus();
4      interruptManager.disableInterrupt();
5      // 禁止内核线程调用
6      PCB *parent = this->running;
7      if (!parent->pageDirectoryAddress) //内核线程没有这个
8      {
9          interruptManager.setInterruptStatus(status);
10         return -1;
11     }
```



```

12 // 创建子进程
13 int pid = executeProcess("", 0); //创建子进程时，没有传入子进程的函数，准备直接
    copy父进程的
14 if (pid == -1)
15 {
16     interruptManager.setInterruptStatus(status);
17     return -1;
18 }
19 // 初始化子进程
20 PCB *child = ListItem2PCB(this->allPrograms.back(), tagInAllList);
21 bool flag = copyProcess(parent, child); //资源复制
22 if (!flag)
23 {
24     child->status = ProgramStatus::DEAD;
25     interruptManager.setInterruptStatus(status);
26     return -1;
27 }
28 interruptManager.setInterruptStatus(status);
29 return pid;
30 }
31

```

3.2父进程资源的复制 copyProcess(parent,child):

3.2.1父进程0特权栈的复制和子进程内核PCB栈的初始化:

- 0特权栈的复制:

- 为什么: 这样可以使得父子进程从相同的返回点开始执行, 要找到相同的返回点就要找到父进程系统调用中断前的相关信息, 下面分析父进程系统调用中断前的相关信息在哪里:

在初始化tss时已经将tss的段选择子写入了CPU自动读取的寄存器中, tss中包含了0特权级栈的地址。父进程系统调用的 `asm_system_call_handler` 中, CPU自动通过寄存器加载出了tss中的0级特权级栈, 并且将中断前的相关信息送入栈中(`pushad`,还包含 `eip` 寄存器)。

那么就是要找到0级特权栈在哪里, 就可以获得相关信息, 复制到子进程的启动栈中(存储启动的相关信息)。然后子进程启动时 `asm_start_process` 将启动栈内容加载入相关寄存器, `iret`返回后就能和父进程在相同的返回点开始执行。

- 怎么做:

- 找到tss的0特权栈: 在 `activateProgramPage` 中可以看到 `tss.esp0 = (int)program + PAGE_SIZE`; 每次调度进程前会将tss的0特权栈放到要激活进程的PCB的顶部, 自顶向下拓展。

那么激活调度父进程前就将0特权栈放在了父进程PCB的顶部, 直接可以到父进程PCB的顶部找到0特权栈

- 0特权栈的结构和启动栈一样, 将在父进程PCB中的0特权栈复制到子进程PCB的相同位置中

```

1 // 复制进程0级栈
2 ProcessStartStack *childpss =(ProcessStartStack *)((int)child +
PAGE_SIZE - sizeof(ProcessStartStack));//
3 ProcessStartStack *parentpss =(ProcessStartStack *)((int)parent +
PAGE_SIZE - sizeof(ProcessStartStack));//0特权栈的esp当前位置
4 memcpy(parentpss, childpss, sizeof(ProcessStartStack));
5 // 设置子进程的返回值为0
6 childpss->eax = 0;

```

- 子进程PCB内核栈的初始化

- PCB中还有一个栈保存内核函数调用的局部变量、返回地址等信息，用于支持内核函数的执行。为了能让子进程能够成功加载执行要将内核栈的函数地址初始化为 `asm_start_process`，传入参数为刚刚复制的0特权栈 `childpss`。

```

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

```

3.2.2复制虚拟地址池

- 设置子进程PCB

```

1 child->status = ProgramStatus::READY;
2 child->parentPid = parent->pid;
3 child->priority = parent->priority;
4 child->ticks = parent->ticks;
5 child->ticksPassedBy = parent->ticksPassedBy;
6 strcpy(parent->name, child->name);

```

- 复制父进程的虚拟地址池的位图

```

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

```

- 复制父进程的页目录表

- 步骤：遍历父进程页目录表的0-768项，查看有效位是否有效，有效则为子进程分配一个物理页，结合父进程页目录表项的索引和子进程对应的映射的物理页构造子进程页目录表项，写入子进程页目录表中，并且初始化这个物理页
- 为什么只复制0-768页目录项（进程用户空间虚拟地址的0-3GB）？

在用 `executeProcess` 创建子进程时，调用了 `createProcessPageDirectory` 将768-1023项（3GB-4GB）初始化好了（内核和用户进程的共享内存）

- 为什么在复制入子进程的页目录表时需要 `asm_update_cr3` 切换处理器中虚拟地址对应的页目录表：

因为将一个页表项复制入子进程后，需要对这个页表项对应的物理页表初始化，而此时只好通过它在子进程中的虚拟地址定位它，然后进行初始化。

为什么进入子进程的虚拟地址空间后还能通过进入前的 `childPageDir` 找到子进程页目录表的位置？进程的页目录表都是在内核地址空间上分配的，而内核地址空间和用户进程虚拟地址的3-4GB是共享的，相同地址映射相同物理页（也体现了用户3GB-4GB映射的好处） 3

```

1 // 子进程页目录表物理地址
2 int childPageDirPaddr = memoryManager.vaddr2paddr(child-
>pageDirectoryAddress);
3 // 父进程页目录表物理地址
4 int parentPageDirPaddr = memoryManager.vaddr2paddr(parent-
>pageDirectoryAddress);
5 // 子进程页目录表指针(虚拟地址)
6 int *childPageDir = (int *)child->pageDirectoryAddress;
7 // 父进程页目录表指针(虚拟地址)
8 int *parentPageDir = (int *)parent->pageDirectoryAddress;
9 // 子进程页目录表初始化
10 memset((void *)child->pageDirectoryAddress, 0, 768 * 4);
11 // 复制页目录表
12 for (int i = 0; i < 768; ++i) // 只有0-3GB的需要构造，3-4GB的在创建的时候就
映射到内核上了
13 {
14     // 无对应页表
15     if (!(parentPageDir[i] & 0x1))
16     {
17         continue;
18     }
19     // 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
20     int paddr =
memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
21     if (!paddr)
22     {
23         child->status = ProgramStatus::DEAD;
24         return false;
25     }
26     // 页目录项
27     int pde = parentPageDir[i]; // 先完全复制页表项
28     // 构造页表的起始虚拟地址
29     int *pageTablevaddr = (int *) (0xffc00000 + (i << 12)); // 对于子进程
虚拟地址而言的页表起始地址
30     asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间
31     childPageDir[i] = (pde & 0x00000fff) | paddr; // 更新页表项的真实物理
页地址
32     memset(pageTablevaddr, 0, PAGE_SIZE);
33     asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
34 }

```

- 复制页表和物理页

- 步骤：遍历父进程的每个页表，遍历页表中每个页表项像复制页目录项一样复制到子进程上。对于每个页表项对于的物理页数据，先拷贝到中转页上，然后切换子进程虚拟空间时将它拷贝到子进程对于的物理页上。

- 为什么需要中转页：需要复制的数据在父进程和子进程中的虚拟地址都一样，需要在内核中分配一个中转页，**中转页由于在内核中所以切换了虚拟空间，子进程还是能找到数据的位置。**4 不然如果直接复制时，切换了子进程的虚拟空间就无法根据数据在父进程的虚拟地址找到数据的物理位置，然后拷贝（也体现了用户3GB到4GB映射的好处）

```
1 //设置中转页
2 char *buffer = (char
*)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
3 // 复制页表和物理页
4 for (int i = 0; i < 768; ++i)
5 {
6     // 无对应页表
7     if (!(parentPageDir[i] & 0x1))
8     {
9         continue;
10    }
11    // 计算页表的虚拟地址
12    int *pageTablevaddr = (int *) (0xffc00000 + (i << 12));
13    // 复制物理页
14    for (int j = 0; j < 1024; ++j)
15    {
16        // 无对应物理页
17        if (!(pageTablevaddr[j] & 0x1))
18        {
19            continue;
20        }
21        // 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
22        int paddr =
memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
23        if (!paddr)
24        {
25            child->status = ProgramStatus::DEAD;
26            return false;
27        }
28        // 构造物理页的起始虚拟地址
29        void *pageVaddr = (void *) ((i << 22) + (j << 12));
30        // 页表项
31        int pte = pageTablevaddr[j];
32        // 复制出父进程物理页的内容到中转页
33        memcpy(pageVaddr, buffer, PAGE_SIZE);
34        asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间
35        pageTablevaddr[j] = (pte & 0x00000fff) | paddr;
36        // 从中转页中复制到子进程的物理页
37        memcpy(buffer, pageVaddr, PAGE_SIZE);
38        asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
39    }
40 }
```

4.实验结果

1. 进程设置：

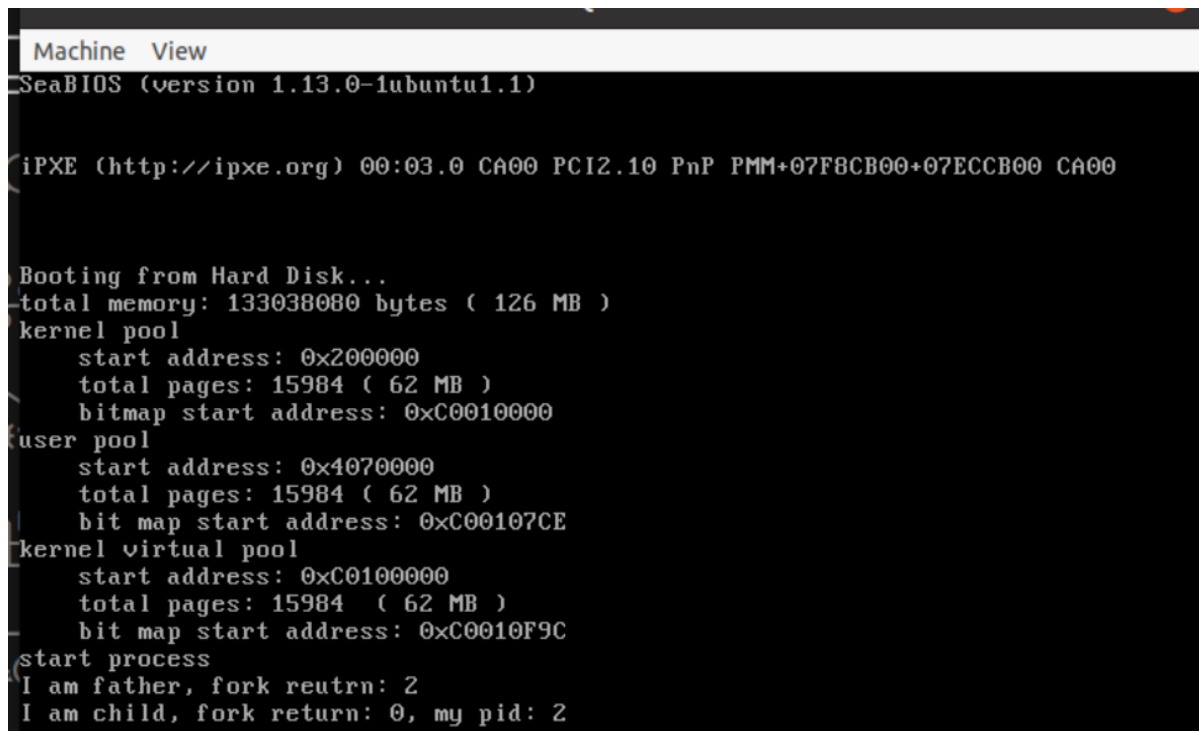
```
1 void first_process()
2 {
```

```

3     int pid = fork();
4     if (pid == -1)
5     {
6         printf("can not fork\n");
7     }
8     else
9     {
10        if (pid)
11        {
12            printf("I am father, fork reutrnr: %d\n", pid);
13        }
14        else
15        {
16            printf("I am child, fork return: %d, my pid: %d\n", pid,
programManager.running->pid);
17        }
18    }
19    asm_halt();
20 }

```

执行结果：父进程成功fork子进程pid为2，子进程被成功调度



The screenshot shows a SeaBIOS boot screen with the following text:

```

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

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

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 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
I am child, fork return: 0, my pid: 2

```

2. 子进程调度执行

- 准备调度子进程:

```

118
119     ListItem *item = readyPrograms.front();
120     PCB *next = ListItem2PCB(item, tagInGeneralList);
B+>121     PCB *cur = running;
122     next->status = ProgramStatus::RUNNING;
123     running = next;
124     readyPrograms.pop_front();
125
126     //printf("schedule: %x %x\n", cur, next);
127
128     activateProgramPage(next);
129
130     asm_switch_thread(cur, next);
131
132     interruptManager.setInterruptStatus(status);
133 }

```

remote Thread 1.1 In: ProgramManager::schedule
Continuing.

Breakpoint 3, ProgramManager::schedule (this=0xc00343c0 <programManager>) at ../src/kernel/program.cpp:121
(gdb) p next->pid
\$1 = 0
(gdb) p cur->pid
Cannot access memory at address 0x22
(gdb) step
(gdb) p cur->pid
\$2 = 1
(gdb) c
Continuing.

Breakpoint 2, ProgramManager::schedule (this=0xc00343c0 <programManager>) at ../src/kernel/program.cpp:94
(gdb) pc
Undefined command: "pc". Try "help".
(gdb) c
Continuing.

Breakpoint 3, ProgramManager::schedule (this=0xc00343c0 <programManager>) at ../src/kernel/program.cpp:121
(gdb) p next->pid
\$3 = 2
准备调度子进程，子进程pid为2

- asm_switch刚切换完进程，跳转到内核栈中的用户进程地址 asm_start_process，此时未加载特权级栈

```

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)
52     asm_ltr:
53     ltr word[esp + 1 * 4]
54     ret
55
56     ; int asm_add_global_descriptor(int low, int high);

```

remote Thread 1.1 In: asm_start_process

eax	0xc0025f80	-1073586304
ecx	0x1	1
edx	0x219000	2199552
ebx	0x0	0
esp	0xc0026f34	0xc0026f34 <PCB_SET+12212>
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0xc0022dc0	0xc0022dc0 <asm_start_process>
eflags	0x286	[IOPL=0 IF SF PF]
cs	0x20	32
ss	0x10	16
ds	0x8	8
es	0x8	8
fs	0x0	0

- 特权级栈加载完成，返回地址eip变化，eax返回值为0，堆栈寄存器es和附加段寄存器fs变化，还没用执行iret代码段寄存器和浮点寄存器暂时没有变化

```
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     >49     iret
50
51     ; void asm_ltr(int tr)
52     asm_ltr:
53         ltr word[esp + 1 * 4]
54         ret
55
56     ; int asm_add_global_descriptor(int low, int high);
57     asm_add_global_descriptor:
58         push ebp
59         mov ebp, esp
60
61         push ebx
62         push esi
```

remote Thread 1.1 In: asm_start_process

eax	0x0	0	从0特权级栈中加载出为0的eax, 返回值为0
ecx	0x0	0	
edx	0x0	0	
ebx	0x0	0	
esp	0xc0026f6c	0xc0026f6c <PCB_SET+12268>	
ebp	0x8048fac	0x8048fac	
esi	0x0	0	
edi	0x0	0	
eip	0xc0022dcd	0xc0022dcd <asm_start_process+13>	
eflags	0x286	[IOPL=0 IF SF PF]	
cs	0x20	32	
ss	0x10	16	
ds	0x33	51	
es	0x33	51	
fs	0x33	51	
gs	0x0	0	
fs_base	0x0	0	
gs_base	0x0	0	
k_gs_base	0x0	0	
cr0	0x80000011	[PG ET PE]	
cr2	0x0	0	
cr3	0x219000	[PDBR=0 PCID=0]	

- 跳转到执行完0x80中断处理函数的位置，中断返回特权级切换，ds数据寄存器和cs代码段寄存器切换，0特权级栈切换为3特权级栈，返回地址eip也变化了

```

145
146         int 0x80
147
148         pop edi
>149         pop esi
150         pop edx
151         pop ecx
152         pop ebx
153         pop ebp
154
155         ret
156
157         ; void asm_init_page_reg(int *directory);

```

remote Thread 1.1 In: asm system call

eax	0x0	0
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048f9c	0x8048f9c
ebp	0x8048fac	0x8048fac
esi	0x0	0
edi	0x0	0
eip	0xc0022e70	0xc0022e70 <asm_system_call+29>
eflags	0x216	[IOPL=0 IF AF PF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51
fs	0x33	51
gs	0x0	0
fs_base	0x0	0
gs_base	0x0	0
k_gs_base	0x0	0
cr0	0x80000011	[PG ET PE]
cr2	0x0	0
cr3	0x219000	[PDBR=0 PCID=0]

--Type <RET> for more, q to quit, c to continue without paging--

- 返回到fork()函数结束位置:

```

30
31     void first_process()
32     {
B+ 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                 printf("I am father, fork reutrn: %d\n", pid);
44             }
45             else
46             {
47                 printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running->pid);
48             }
49         }
50

```

remote Thread 1.1 In: first_process

eax	0x0	0
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048fe4	0x8048fe4
ebp	0x8048ffc	0x8048ffc
esi	0x0	0
edi	0x0	0
eip	0xc0021035	0xc0021035 <first_process()+18>
eflags	0x206	[IOPL=0 IF PF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51
fs	0x33	51
gs	0x0	0
fs_base	0x0	0
gs_base	0x0	0
k_gs_base	0x0	0
cr0	0x80000011	[PG ET PE]
cr2	0x0	0
cr3	0x219000	[PDBR=0 PCID=0]

--Type <RET> for more, q to quit, c to continue without paging--

3. 父进程执行完fork返回:

- 父进程准备执行完fork(): 返回值eax为2

```
374
375         interruptManager.setInterruptStatus(status);
B+ 376         return pid;
>377     }
378
379     bool ProgramManager::copyProcess(PCB *parent, PCB *child)
380     {
381         // ^%^ %^ (^ ^ ' ^ ^ ^ &^
382         ProcessStartStack *childpss =
383             (ProcessStartStack *)((int)child + PAGE_SIZE - sizeof(ProcessStartStack));
384         ProcessStartStack *parentpss =
385             (ProcessStartStack *)((int)parent + PAGE_SIZE - sizeof(ProcessStartStack));
386         memcpy(parentpss, childpss, sizeof(ProcessStartStack));
387         // ^(^ ^ ^ ^ %^ (^ ^ ' ^ ' ^ (^ ^ %^ %^ ^ $^
388         childpss->eax = 0;
389
390         // ^%^ %^ &^ (^ sm_switch_thread^'^ &^ ' ^ %^ %^
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; // asm_start_process^'^ %^ &^ &^ (
397         child->stack[5] = 0; // asm_start_process ^(^ ^ %^ %^ %^
398         child->stack[6] = (int)childpss; // asm_start_process ^%^ &^
399
remote Thread 1.1 In: ProgramManager::fork
eax          0x2          2
ecx          0xc0010f9c   -1073672292
edx          0x0          0
ebx          0x0          0
esp          0xc0025ed8   0xc0025ed8 <PCB_SET+8024>
ebp          0xc0025f00   0xc0025f00 <PCB_SET+8064>
esi          0x0          0
edi          0x0          0
eip          0xc0020afd   0xc0020afd <ProgramManager::fork()+269>
eflags      0x286        [ IOPL=0 IF SF PF ]
cs          0x20          32
ss          0x10          16
ds          0x8           8
es          0x8           8
fs          0x33          51
gs          0x18          24
fs_base     0x0           0
gs_base     0xb8000       753664
k_gs_base   0x0           0
cr0         0x80000011    [ PG ET PE ]
cr2         0x0           0
cr3         0x200000      [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

- 先返回到系统调用服务函数中, 中断处理未结束

```

107         ; ^^ &^ %^ &^
108         push edi
109         push esi
110         push edx
111         push ecx
112         push ebx
113
114         sti
115         call dword[system_call_table + eax * 4]
116         cli
117
118         add esp, 5 * 4
119
120         mov [ASM_TEMP], eax
121         popad
122         pop gs
123         pop fs
124         pop es
>125         pop ds
126         mov eax, [ASM_TEMP]
127
128         iret
129     asm_system_call:
130         push ebp
131         mov ebp, esp

```

remote Thread 1.1 In: asm_system_call_handler

eax	0x2	2
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0xc0025f68	0xc0025f68 <PCB_SET+8168>
ebp	0x8048fac	0x8048fac
esi	0x0	0
edi	0x0	0
eip	0xc0022e4c	0xc0022e4c <asm_system_call_handler+53>
eflags	0x86	[IOPL=0 SF PF]
cs	0x20	32
ss	0x10	16
ds	0x8	8
es	0x33	51
fs	0x33	51
gs	0x0	0
fs_base	0x0	0
gs_base	0x0	0
k_gs_base	0x0	0
cr0	0x80000011	[PG ET PE]
cr2	0x0	0
cr3	0x200000	[PDBR=0 PCID=0]

- 返回中断入口函数:

```
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
152         pop ebx
153         pop ebp
154
155         ret
156
157         ; void asm_init_page_reg(int *directory);
158         asm_init_page_reg:
159         push ebp
160         mov ebp, esp
161
162         push eax
163
```

remote Thread 1.1 In: asm_system_call

eax	0x2	2
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048f98	0x8048f98
ebp	0x8048fac	0x8048fac
esi	0x0	0
edi	0x0	0
eip	0xc0022e6f	0xc0022e6f <asm_system_call+28>
eflags	0x216	[IOPL=0 IF AF PF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51
es	0x33	51
fs	0x33	51
gs	0x0	0
fs_base	0x0	0
gs_base	0x0	0
k_gs_base	0x0	0
cr0	0x80000011	[PG ET PE]
cr2	0x0	0
cr3	0x200000	[PDBR=0 PCID=0]

4. 比较：父进程从fork返回时还要先返回到int 0x80的中断处理函数，而子进程直接返回到系统调用入口函数，（子进程的返回地址是来自于保留在父进程PCB中的0特权级栈，而0特权级栈中保留的返回地址是特权级切换前的地址也就是系统调用入口函数中）

相同点：父子进程fork返回的寄存器值中除了eax函数返回值不一样其他都相同

5. 子进程返回值为0原因：子进程在复制0特权级的时候就把其中的eax设为0（copyProcess），然后调度执行到asm_start_process时会把特权级栈中的保存的寄存器值pop出来，而eax就被pop出为0，同时eax还是函数的返回值，这样就会使子进程的fork函数返回值为0
6. 父进程fork函数返回值为子进程的pid:在progamManager的 fork 函数中直接返回子进程pid号

5.总结

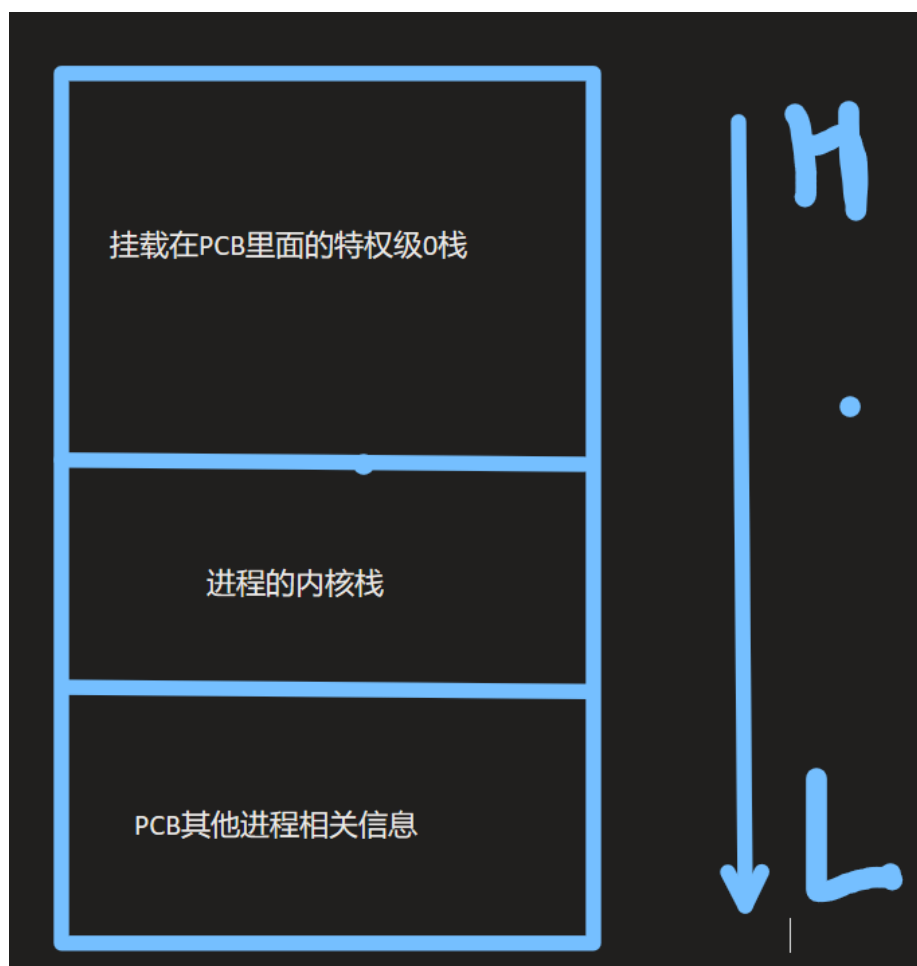
1. 虽然每个进程的PCB中都有内核栈和0特权栈，**0特权栈和内核栈是不一样的**
- 位置上：0特权级栈位于内核栈的上方（高地址）

```

1 //内核栈:
2 thread->stack = (int *)((int)thread + PCB_SIZE -
  sizeof(ProcessStartStack));
3 //0特权级栈
4 tss.esp0 = (int)program + PAGE_SIZE;

```

- 含义:
 - 内核栈: 内核栈是每个进程都有的, 且是私有的, 在进程切换时不会保存和恢复, 用于保存内核函数调用的局部变量、返回地址等信息。
 - 0特权栈: 主要保存CPU从用户态切换到内核态时的任务状态, 包括寄存器的值、中断信息、系统调用参数等, 是特权级特有, 但是暂时存储在PCB中
- 作用:
 - 0特权栈: 仅在CPU从用户态切换到内核态的短暂过程中使用, 一旦退出内核态就不再需要。
 - 内核栈: 在内核态下的整个执行过程中都有效, 用于支持内核函数的执行和切换。
- 图解:



2. 用户态虚拟空间3GB-4GB都映射内核的plus: [前文跳转3](#), [前文跳转4](#)
3. “注意到 ProgramStartProcess 中保存了父进程的 eip, eip 的内容也是 asm_system_call_handler 的返回地址。”父进程的0特权级栈的eip存的返回地址到底是什么, 而且是在哪里存的?
 - 一开始以为是在 asm_system_call_handler 里面存的, 而是通过pushad存进特权0栈的, 但是经过查询资料 pushad 只能将通用寄存器入栈, 而不能将 eip 入栈, 而且子进程开启时的函数也不是在 asm_system_call_handler 开头

- 回顾TSS自动调出的时间是在 `asm_system_call` 中的 `int 0x80` 早在进入80对应的中断处理函数时就已经转换特权栈0了，而且跳转 `asm_system_call_handler` 前已经将返回地址当前的 `eip`入栈。这就能解释为什么子进程启动时到的地方和父进程执行完系统调用返回的位置不一样了。子进程启动时到达的地方刚刚好就是在 `asm_system_call` 执行完中断返回的位置

Assignment3

1.实验要求

- 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

1. `exit`的实现（隐式调用分析在实验结果中）
2. `wait`的实现
3. 回收僵尸进程：在每个进程`exit`的时候查看是否有没`dead`的父进程存在，若有则不释放PCB，否则需要自己释放掉PCB

3.关键代码

3.1`exit`的实现

- `exit`的系统调用处理函数：
 - 步骤：
 - 标记PCB的状态为`DEAD`;
 - 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;
 - 立即执行线程/进程调度。
 - 注意：此处没有释放进程的PCB

```
1 void ProgramManager::exit(int ret)
2 {
3     // 关中断
4     interruptManager.disableInterrupt();
5     // 第一步，标记PCB状态为`DEAD`并放入返回值。
6     PCB *program = this->running;
7     program->retValue = ret;
8     program->status = ProgramStatus::DEAD;
9     int *pageDir, *page;
10    int paddr;
11    // 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。
12    if (program->pageDirectoryAddress)
13    {
```

```

14     pageDir = (int *)program->pageDirectoryAddress;
15     for (int i = 0; i < 768; ++i) //只用释放掉0GB-3GB, 内核的3GB-4GB不要释放掉
    了
16     {
17         if (!(pageDir[i] & 0x1)) //检查页目录项是否有效
18         {
19             continue; //无效则继续
20         }
21
22         page = (int *) (0xffc00000 + (i << 12)); //页目录表项对应的虚拟页表地址
23
24         for (int j = 0; j < 1024; ++j) //遍历页表
25         {
26             if (!(page[j] & 0x1)) {
27                 continue; //若页表项无效继续遍历
28             }
29
30             paddr = memoryManager.vaddr2paddr((i << 22) + (j << 12)); //页
    的基址虚拟地址
31             memoryManager.releasePhysicalPages(AddressPoolType::USER,
    paddr, 1); //释放物理页
32         }
33
34         paddr = memoryManager.vaddr2paddr((int)page); //释放页表
35         memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr,
    1);
36     }
37
38     memoryManager.releasePages(AddressPoolType::KERNEL, (int)pageDir,
    1); //释放页目录表
39
40     int bitmapBytes = ceil(program->userVirtual.resources.length, 8);
41     int bitmapPages = ceil(bitmapBytes, PAGE_SIZE);
42
43     memoryManager.releasePages(AddressPoolType::KERNEL, (int)program-
    >userVirtual.resources.bitmap, bitmapPages); //释放位图
44
45 }
46
47 // 第三步, 立即执行线程/进程调度。
48 schedule();
49 }

```

- 为了实现进程退出时的隐式调用, 修改进程用户态栈的退出函数为 `exit`, 当进程结束时

```

1 void load_process(const char *filename)
2 {
3     ...
4     // 设置进程返回地址
5     int *userStack = (int *)interruptStack->esp;
6     userStack -= 3;
7     userStack[0] = (int)exit; //退出函数
8     userStack[1] = 0; //exit函数的返回地址
9     userStack[2] = 0; //exit函数的参数
10

```

```

11     interruptStack->esp = (int)userStack;
12
13     interruptStack->ss = programManager.USER_STACK_SELECTOR;
14
15     asm_start_process((int)interruptStack);
16 }

```

3.2wait的实现

- 步骤：
 - 查找当前所有进程中是否有查询进程的子进程。
 - 若有则看是否为dead状态。
 - 如果存在子进程而且状态为dead，若有返回值要求则存下返回值并且释放子进程的PCB，返回其的pid；
 - 如果没有子进程，直接返回-1；
 - 如果存在子进程但是子进程的状态不是dead，阻塞等待

```

1  int ProgramManager::wait(int *retval)
2  {
3      PCB *child;
4      ListItem *item;
5      bool interrupt, flag;
6      while (true)
7      {
8          interrupt = interruptManager.getInterruptStatus();
9          interruptManager.disableInterrupt();
10         item = this->allPrograms.head.next;
11         // 查找子进程
12         flag = true;
13         while (item)
14         {
15             child = ListItem2PCB(item, tagInAllList);
16             //存在子进程
17             if (child->parentPid == this->running->pid)
18             {
19                 flag = false;
20                 //检查子进程的状态
21                 if (child->status == ProgramStatus::DEAD)
22                 {
23                     break;
24                 }
25             }
26             item = item->next;
27         }
28         // 找到一个子进程且状态为dead
29         if (item)
30         {
31             if (retval)//接收子进程的返回值
32             {
33                 *retval = child->retvalue;
34             }
35             int pid = child->pid;
36             releasePCB(child);

```

```

37         interruptManager.setInterruptStatus(interrupt);
38         return pid;
39     }
40     else
41     { // 没有找到子进程，直接返回-1
42         if (flag)
43         {
44             interruptManager.setInterruptStatus(interrupt);
45             return -1;
46         }
47         else // 存在子进程，但子进程的状态不是DEAD，调度其他进程阻塞等待
48         {
49             interruptManager.setInterruptStatus(interrupt);
50             schedule();
51         }
52     }
53 }
54 }

```

3.3回收僵尸进程

- 思路：
 - 在每个进程exit的时候查看当前所有进程中是否存在该进程的父进程。
 - 若不存在父进程或者存在父进程但是父进程的状态为dead，进程在exit释放掉PCB
 - 若存在父进程且父进程状态不为dead，那进程就不在exit释放PCB而是要等待父进程释放掉子进程的PCB
- 查找每个进程的父进程 checkparent：
 - 传入参数pid为子进程PCB中记录的父进程pid
 - 遍历所有进程和线程，查找父进程
 - 父进程存在且状态为dead，返回1
 - 父进程不存在，返回-1
 - 父进程存在但不为dead，返回0

```

1  int ProgramManager::checkparent(int pid)
2  {
3      PCB *parent;
4      ListItem *item;
5      bool interrupt, flag;
6      interrupt = interruptManager.getInterruptStatus();
7      interruptManager.disableInterrupt();
8      item = this->allPrograms.head.next;
9      // 查找父进程
10     flag = true;
11     while (item)
12     {
13         parent = ListItem2PCB(item, tagInAllList);
14         //父进程存在
15         if (parent->pid == pid)
16         { //父进程存在且状态为dead
17             flag = false;
18             if (parent->status == ProgramStatus::DEAD)

```



```

19         {
20             return 1;
21         }
22     }
23     item = item->next;
24 }
25 if (flag)
26 {
27     //父进程不存在
28     interruptManager.setInterruptStatus(interrupt);
29     return -1;
30 }
31 else // 存在父进程，但子进程的状态不是DEAD
32 {
33     interruptManager.setInterruptStatus(interrupt);
34     return 0;
35 }
36 }

```

- 在 exit 函数的末尾检测是否需要释放PCB
 - 检查是否存在父进程
 - 父进程存在且状态为dead和不存在父进程，释放PCB
 - 存在父进程且状态不为dead，直接调度

```

1 void ProgramManager::exit(int ret)
2 {
3     .....
4
5     int id=checkparent(program->parentPid);
6     if(id!=0){
7         if(id==1){
8             printf("my parent has dead,exit\n");
9         }
10        else{
11            printf("I have no parent,exit\n");
12        }
13        releasePCB(program);
14    }
15    schedule();
16 }

```

- 思考: 5

针对不存在父进程的情况讨论:

进程在初始化的时候，PCB中父进程的id都被初始化为0了，即使进程并不存在fork()上的父进程，但是PCB上的父进程id会指向第一个线程（pid=0）。而第一个线程不会退出，可能会存在进程没有父进程而且状态为dead，但是不会释放PCB。

可以改良如下，使没有父进程的进程（`program->parentPid==0`）在exit时可以直接释放PCB。

```

1      int id=checkparent(program->parentPid);
2      if(id!=0 || program->parentPid==0){
3          if(id==1 || program->parentPid!=0){
4              printf("my parent has dead,exit\n");
5          }
6          else{
7              printf("I have no parent,exit\n");
8          }
9          releasePCB(program);
10     }
11     schedule();

```

4.实验结果

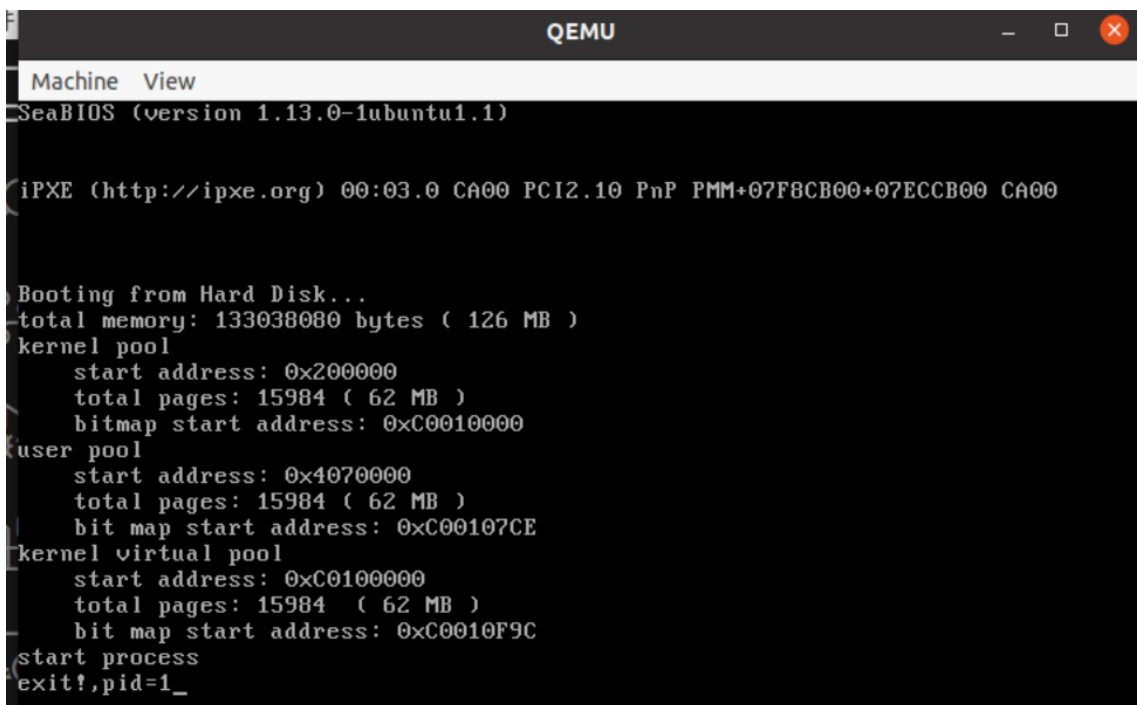
4.1:exit实例和执行过程

- 实例：由执行结果可见成功退出并且打印相关信息

```

1  void first_process()
2  {
3      int a = 0;
4      exit(0);
5  }
6  void ProgramManager::exit(int ret)
7  {
8      .....
9      if (program->pageDirectoryAddress)
10     {
11         .....
12         printf("exit!,pid=%d",program->pid);
13     }
14
15     schedule();
16 }

```



```

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

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

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
exit!,pid=1_

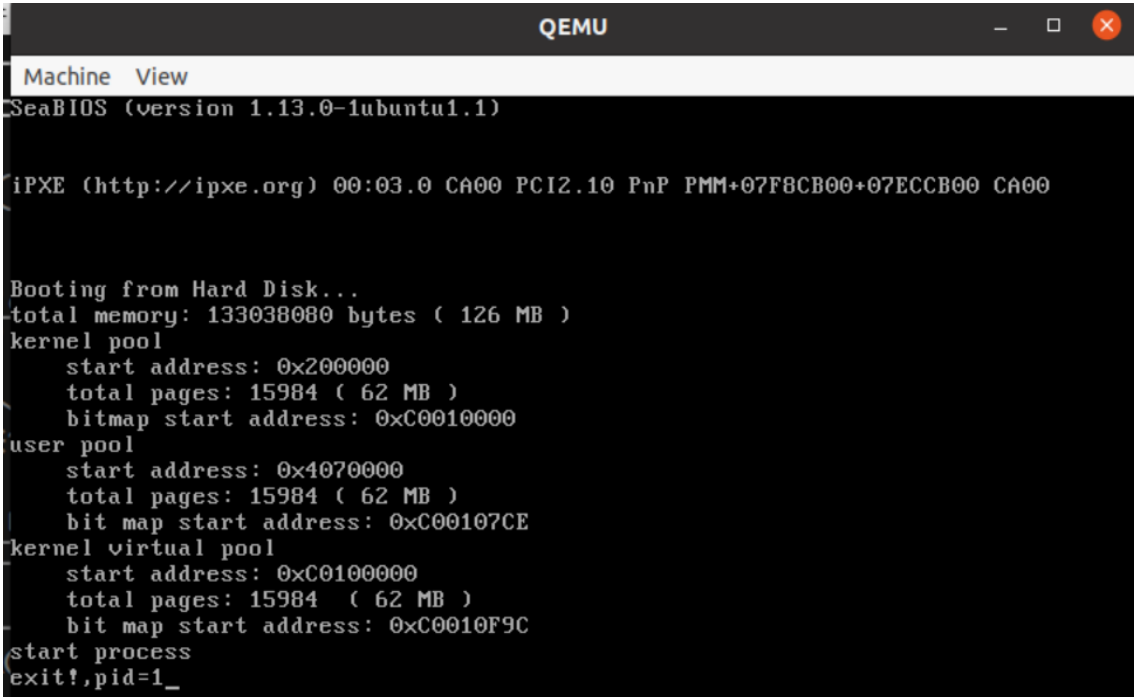
```

- 执行步骤：调用exit函数，调用对于exit的系统调用函数：设置进程状态为dead，释放资源（页目录表，页表，页，位图），打印相关信息，调度其他进程

4.2隐式调用exit

- 为什么：在进程的启动函数load_process中，把exit的地址,默认参数0,exit返回地址0放在了进程的用户栈上，当进程执行结束时从用户栈弹出返回地址为exit和返回，跳转到exit中执行
- 步骤和显示调用exit没有区别，结果相同

```
1 void first_process()  
2 {  
3     int a = 0;  
4 }
```

A screenshot of a QEMU terminal window. The title bar says 'QEMU'. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal text shows the boot process: 'SeaBIOS (version 1.13.0-1ubuntu1.1)', 'iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00', 'Booting from Hard Disk...', 'total memory: 133038080 bytes (126 MB)', 'kernel pool' with its start address, total pages, and bitmap start address, 'user pool' with its start address, total pages, and bit map start address, 'kernel virtual pool' with its start address, total pages, and bit map start address, 'start process', and finally 'exit!,pid=1_'.

```
QEMU  
Machine View  
SeaBIOS (version 1.13.0-1ubuntu1.1)  
  
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00  
  
Booting from Hard Disk...  
total memory: 133038080 bytes ( 126 MB )  
kernel pool  
  start address: 0x200000  
  total pages: 15984 ( 62 MB )  
  bitmap start address: 0xC0010000  
user pool  
  start address: 0x4070000  
  total pages: 15984 ( 62 MB )  
  bit map start address: 0xC00107CE  
kernel virtual pool  
  start address: 0xC0100000  
  total pages: 15984 ( 62 MB )  
  bit map start address: 0xC0010F9C  
start process  
exit!,pid=1_
```

- gdb过程：
 - 进程执行结束：

```

52         // }
53         void first_process()
54         {
B+ 55             int a = 0;
>56         }
57
58         void first_thread(void *arg)

```

```

remote Thread 1.1 In: first process
eax          0x0          0
ecx          0x0          0
edx          0x0          0
ebx          0x0          0
esp          0x8048fe0     0x8048fe0
ebp          0x8048ff0     0x8048ff0
esi          0x0          0
edi          0x0          0
eip          0xc00213ea     0xc00213ea <first_process()+17>
eflags       0x202        [ IOPL=0 IF ]
cs           0x2b         43
ss           0x3b         59
ds           0x33         51
es           0x33         51
fs           0x33         51
gs           0x0          0
fs_base      0x0          0
gs_base      0x0          0
k_gs_base    0x0          0
cr0          0x80000011    [ PG ET PE ]
cr2          0x0          0
cr3          0x200000      [ PDBR=0 PCID=0 ]
Type <RET> for more, q to quit, c to continue without paging

```

- 跳转到 `exit`:

```

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
51     int wait(int *retval) {
52         return asm_system_call(4, (int)retval);

```

```

remote Thread 1.1 In: exit
edx          0x0          0
ebx          0x0          0
esp          0x8048ff8     0x8048ff8
ebp          0x0          0x0
esi          0x0          0
edi          0x0          0
eip          0xc00212c3     0xc00212c3 <exit(int)>
eflags       0x202        [ IOPL=0 IF ]
cs           0x2b         43
ss           0x3b         59
ds           0x33         51
es           0x33         51
fs           0x33         51
gs           0x0          0
fs_base      0x0          0
gs_base      0x0          0
k_gs_base    0x0          0
cr0          0x80000011    [ PG ET PE ]
cr2          0x0          0
cr3          0x200000      [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--

```

4.3wait执行过程和实例

- 实例：执行结果同预期一样，父进程(pid=1)等待子进程(pid=2)结束后才退出

```

1  void first_process()
2  {
3      int pid = fork();
4

```

```

5     if (pid == -1)
6     {
7         printf("can not fork\n");
8         asm_halt();
9     }
10    else
11    {
12        if (pid)
13        {
14            printf("I am father\n");
15            wait(0);
16        }
17        else
18        {
19            printf("I am child, exit\n");
20        }
21    }
22 }

```

The screenshot shows a QEMU window titled "Machine View". The console output displays the boot process starting with IPXE, followed by booting from a hard disk. It shows memory statistics (133038080 bytes / 126 MB) and the initialization of kernel and user pools. The program output is as follows:

```

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

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
I am father
I am child, exit
exit!,pid=2
exit!,pid=1

```

- wait执行过程:

调用wait函数，调用wait系统调用，查询是否有对应子进程，若有且未结束阻塞等待；若有且结束则收集子进程的返回值并且返回子进程的pid；若无子进程，返回-1

4.4回收僵尸进程

- 实例：父进程执行完直接退出不等待子进程，但是子进程也可以成功释放PCB

```

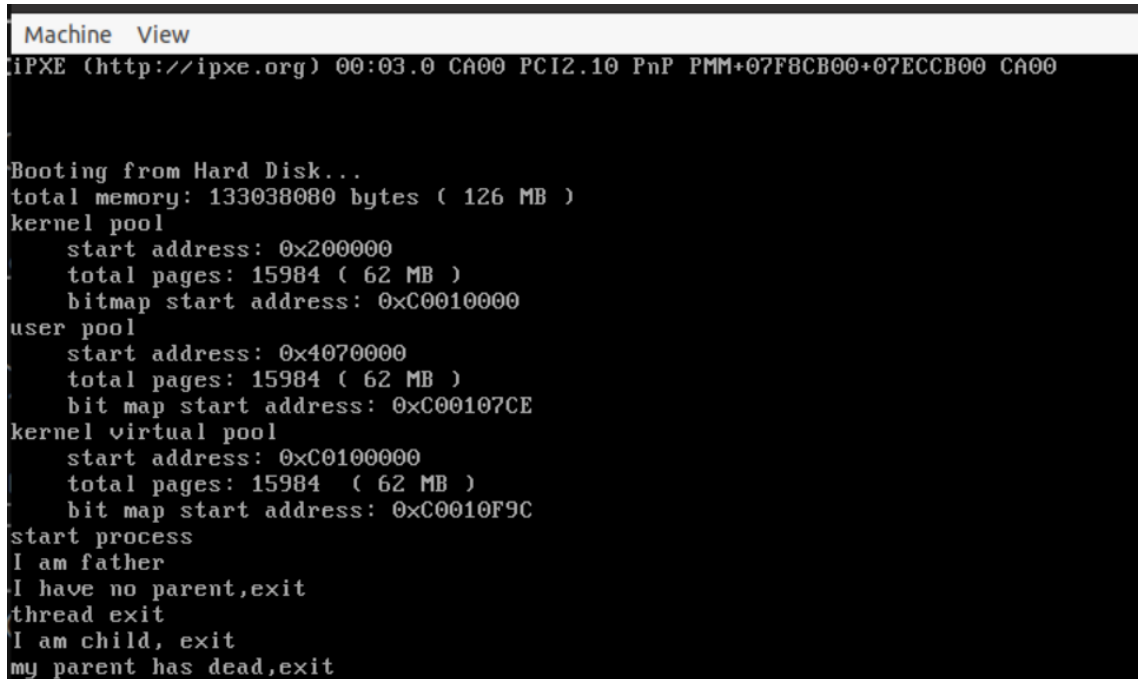
1 void first_process()
2 {
3     int pid = fork();
4
5     if (pid == -1)
6     {
7         printf("can not fork\n");
8         asm_halt();
9     }

```

```

10     else
11     {
12         if (pid)
13         {
14             printf("I am father\n");
15             exit(0);
16         }
17         else
18         {
19             printf("I am child, exit\n");
20         }
21     }
22 }

```



```

Machine View
iPXE (http://ipxe.org) 00:03.0 CA00 PC12.10 PnP PMM+07F8CB00+07ECCB00 CA00

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
I am father
I have no parent,exit
thread exit
I am child, exit
my parent has dead,exit

```

5.总结

进程有父线程但是没有父进程的思考：[前文跳转5](#)