



本科生实验报告

实验课程: _____ 操作系统 _____

实验名称: _____ 从内核态到用户态 _____

专业名称: _____ 网络空间安全 _____

学生姓名: _____ 陈浚铭 _____

学生学号: _____ 20337021 _____

实验地点: _____ 温暖的家 _____

实验成绩: _____

报告时间: _____

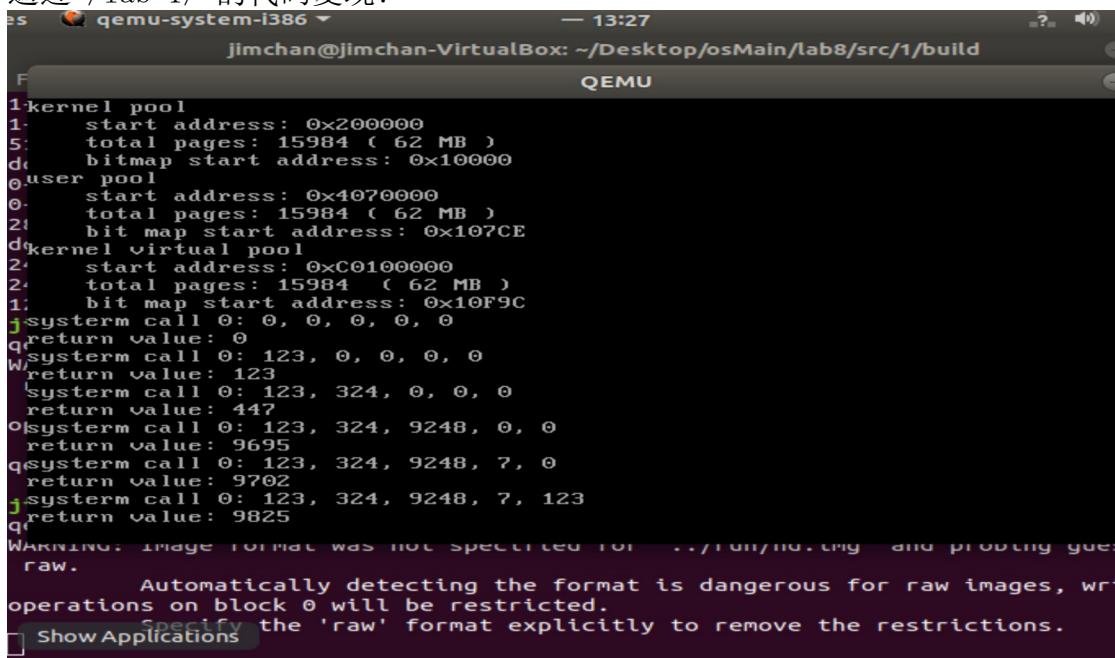
1. 实验要求

深入了解系统调用，`fork()`, `exec()` 和 `wait()` 三个在 Linux 有关进程的函数

2. 实验过程与结果

assignment 1:

通过 `/lab 1/` 的代码复现：



The screenshot shows a terminal window titled "qemu-system-i386" running on a VirtualBox machine. The command "qemu-system-i386" is in the title bar. The terminal window displays assembly code and system call interactions. The assembly code includes memory pool definitions and system call sequences. The system calls involve registers like ECX, EDX, ECX, and ECX, with values such as 123, 324, 9248, and 7. The terminal also shows a warning about raw image format and a note about automatically detecting the format being dangerous.

```
kernel pool
1. start address: 0x200000
5. total pages: 15984 ( 62 MB )
d. bitmap start address: 0x10000
o user pool
0. start address: 0x4070000
0. total pages: 15984 ( 62 MB )
2. bit map start address: 0x107CE
d kernel virtual pool
2. start address: 0xC0100000
2. total pages: 15984 ( 62 MB )
1. bit map start address: 0x10F9C
j system call 0: 0, 0, 0, 0, 0, 0
q return value: 0
w system call 0: 123, 0, 0, 0, 0, 0
w return value: 123
l system call 0: 123, 324, 0, 0, 0
return value: 447
o system call 0: 123, 324, 9248, 0, 0
return value: 9695
q system call 0: 123, 324, 9248, 7, 0
return value: 9702
j system call 0: 123, 324, 9248, 7, 123
q return value: 9825
q
WARNING: Image format was not specified for .../run/initrd.raw and probing
        raw.
        Automatically detecting the format is dangerous for raw images, wr-
operations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.
 Show Applications
```

系统调用的传统实现方法是通过软中断的。用户进程通过写入适合的值到特定的寄存器或者在栈上来初始化一个预定义的中断，并从而让内核执行。在 Linux 上，使用 `0x80` 这个数来代表中断来系统调用。

在第二章中，我们把代码段，数据段和栈段的段描述符的 DPL 设置为 0，从而表明这几个段是特权资源所在的段。注意我们将这几个段设置为非一致性代码段。而非一致性代码段要求 `CPL = DPL` 才能访问代码段，而我们实现的用户进程运行时的 `CPL = 3`，因此进程无法访问 `DPL = 0` 的数据段和栈段。为了访问 `DPL = 0` 的段我们所需要从低权级转移到高特权级。我们希望通过 `0x80` 编号的中断来实现从低权级转移到高特权级。

我们通过在汇编语言实现一个系统调用的入口函数，如下：

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

通过系统调用表，我们可以把每一个系统调用函数对应一个系统调用号，并通过系统调用号在系统调用表中找出系统调用函数的位置，并调用它。

通过管理系统调用的类 `SystemService`，来实现系统调用的功能。

当中，在类的初始化函数 `initialize()`，通过：

```
setInterruptDescriptor(0x80, (uint)asm_system_call_handler, 3)
```

设定下 `0x80` 中断编号的处理函数为 `asm_system_call_handler`，并设置 `DPL=3`。

以上已说过 `asm_system_call_handler` 为 `0x80` 的中断处理函数。它的功能

是通过汇编语言调用系统调用函数。从教程的代码里，

asm_system_call_handler 的代码流程为：

(1) 修改 ds, es, gs 的段寄存器

(2) 把 5 个寄存器当中的系统调用函数参数压栈

(3) 开中断，并调用系统调用处理函数。注意到我们的函数参数当中寄存器 eax 是保存着系统调用号，而 system_call_table 保存着系统调用表的地址，

因此 [system_call_table + eax*4] 就是系统调用号的系统调用处理函数的地址。

(4) 把 5 个参数弹出栈 (add esp, 5*4)

(5) 在系统调用函数返回后，函数的返回会放在 eax 中，但是我们在从栈弹出寄存器的值的时候，会修改了 eax 的值，因此通过一个变量 ASM_TEMP 临时保存 eax 的值，并在弹出寄存器的值以后把 eax 恢复为系统调用函数的返回值。

类中的函数 setSystemCall(int index, int function) 函数把 system_call_table 的 index 编号的元素设置为 function 的函数地址。

教程中实现的系统调用函数 syscall_0 把打印 5 个输入变量，并返回它们的总和。之后通过在 setup_kernel() 函数里面初始化上面所说的 systemService 管理系统调用的类，把 syscall_0 系统调用函数放在 system_call_table 的 0 位置。之后 asm_system_call 函数通过 0x80 号中断来调用在 system_call_table[0] 的地址的函数，也就是 syscall_0。

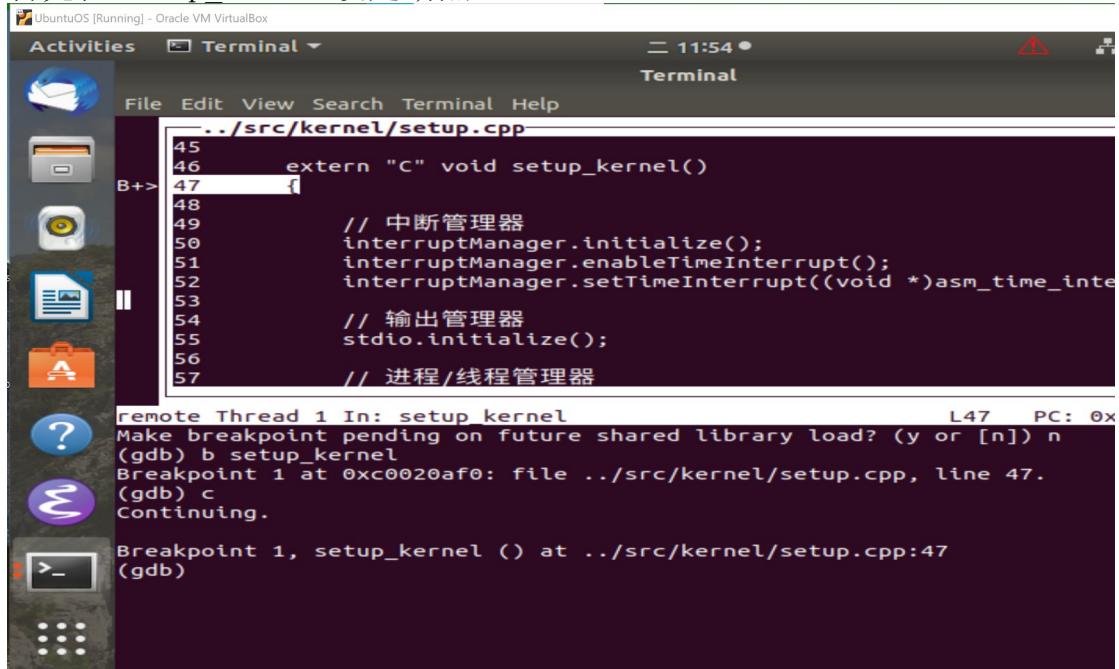
- 系统调用的栈变化以及 TSS 在系统调用执行的功能

我们在这个实验涉及到内核和用户特权级之间的转换。在实现用户进程之前，运行的特权级在 0 下，我们调度用户进程上处理器时候，用户运行在特权级 3，因此我们需要从高特权级向低特权级转移。我们在实验上通过在 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送入段寄存器和栈指针，其中我们会在每一个进程的 PCB 的用户虚拟地址池中分配一页作为栈的空间。另外，在进程执行系统调用的时候，在系统调用执行时，我们需要改变 current privilege level(CPL) 从 PL3 (用户级) 到 PL0 (内核级)。这是通过 0x80 号的中断来实现的，并且使用 tss.esp0 的栈指针。

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

注意当进程的 PCB 首次加载到处理器执行时，CPU 会进入 load_process(const char *filename)，当中 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。

首先在 setup_kernel 设定断点，

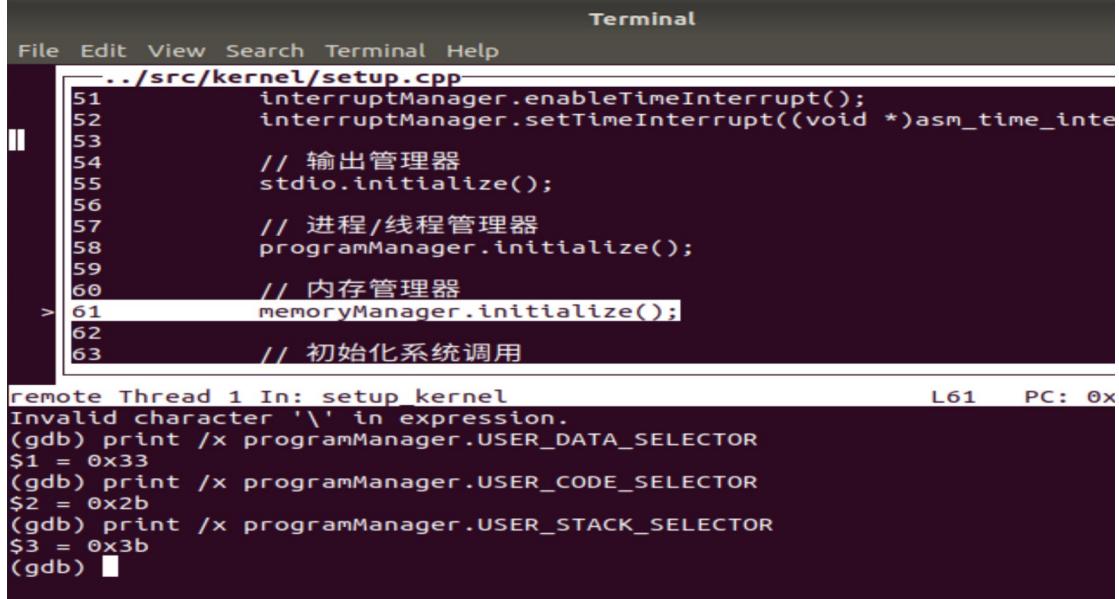


A screenshot of a Linux terminal window titled "Terminal". The window shows a GDB session for the file "setup.cpp". A breakpoint is set at line 47. The code in setup.cpp includes comments for interrupt manager, output manager, and process/thread manager initialization. The GDB prompt shows the command "b setup_kernel" being entered, followed by "Breakpoint 1 at 0xc0020af0: file/src/kernel/setup.cpp, line 47." The command "c" is then run to continue execution.

```
UbuntuOS [Running] - Oracle VM VirtualBox
Activities Terminal 11:54 Terminal
File Edit View Search Terminal Help
..../src/kernel/setup.cpp
45     extern "C" void setup_kernel()
46 {
47 // 中断管理器
48     interruptManager.initialize();
49     interruptManager.enableTimeInterrupt();
50     interruptManager.setTimeInterrupt((void *)asm_time_int
51
52 // 输出管理器
53     stdio.initialize();
54
55 // 进程/线程管理器
56
57
remote Thread 1 In: setup kernel L47 PC: 0x
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b setup_kernel
Breakpoint 1 at 0xc0020af0: file ..../src/kernel/setup.cpp, line 47.
(gdb) c
Continuing.

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

一直执行到 programManager. initialize() 为止，看看 programManager 类的 USER_STACK_SELECTOR, USER_CODE_SELECTOR, USER_STACK_SELECTOR 选择子的值。



A screenshot of a Linux terminal window titled "Terminal". The window shows a GDB session for the file "setup.cpp". Execution has stopped at line 47. The code continues with memory manager and system call initialization. The GDB prompt shows the command "print /x programManager.USER_DATA_SELECTOR" being entered, followed by the values \$1 = 0x33, \$2 = 0x2b, and \$3 = 0x3b. The command "c" is then run to continue execution.

```
File Edit View Search Terminal Help
..../src/kernel/setup.cpp
51     interruptManager.enableTimeInterrupt();
52     interruptManager.setTimeInterrupt((void *)asm_time_int
53
54 // 输出管理器
55     stdio.initialize();
56
57 // 进程/线程管理器
58     programManager.initialize();
59
60 // 内存管理器
61     memoryManager.initialize();
62
63 // 初始化系统调用

remote Thread 1 In: setup kernel L61 PC: 0x
Invalid character '\' in expression.
(gdb) print /x programManager.USER_DATA_SELECTOR
$1 = 0x33
(gdb) print /x programManager.USER_CODE_SELECTOR
$2 = 0x2b
(gdb) print /x programManager.USER_STACK_SELECTOR
$3 = 0x3b
(gdb)
```

我们在这里去看通过 load_process 最后的代码语句
asm_start_process((int)interruptStack) 执行前后观察段寄存器以及栈指针的变化。

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

执行 asm_start_process 之前：

```
Terminal
File Edit View Search Terminal Help

[ No Source Available ]

remote Thread 1 In: asm_start_process L?? PC: 0xc0022
eax      0xc002565c      -1073588644
ecx      0xc0010000      -1073676288
edx      0x3b      59
ebx      0x0      0
esp      0xc0025624      0xc0025624 <PCB_SET+8068>
ebp      0xc0025650      0xc0025650 <PCB_SET+8112>
esi      0x0      0
---Type <return> to continue, or q <return> to quit---
```

```
Terminal
File Edit View Search Terminal Help

[ No Source Available ]

remote Thread 1 In: asm_start_process L?? PC: 0xc0022
edi      0x0      0
eip      0xc0022620      0xc0022620 <asm_start_process>
eflags   0x92      [ AF SF ]
cs       0x20      32
ss       0x10      16
ds       0x8       8
es       0x8       8
---Type <return> to continue, or q <return> to quit---
```

在执行 `asm_start_process` 之后:

The image consists of three vertically stacked screenshots of a Linux terminal window. Each screenshot shows a different state of the assembly code execution and the register dump.

Screenshot 1 (Top): The terminal window title is "Activities Terminal". It shows the assembly code for `first_process`. The instruction at address `0xc0022620` is highlighted. The output shows the QEMU environment (SeaBIOS, iPXE) and memory details. The command `(gdb) n` is entered, indicating single-stepping.

```
File Edit View Search Terminal Help
./src/kernel/setup.cpp
27         first, second, third, forth, fifth);
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
32 {
33     asm_system_call(0, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
Booting from Hard Disk...
remote Thread 1 In: first_process
Continuing.
Breakpoint 1, 0xc0022620 in
(gdb) n
Single stepping until exit from
which has no line number info
first_process () at ./src/k
(gdb) 
```

Screenshot 2 (Middle): The terminal window title is "Terminal". It shows the assembly code for `first_process`. The instruction at address `0xc0020a62` is highlighted. The output shows the register dump for thread 1. The command `(gdb) n` is entered, indicating single-stepping.

```
File Edit View Search Terminal Help
./src/kernel/setup.cpp
27         first, second, third, forth, fifth);
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
32 {
33     asm_system_call(0, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
remote Thread 1 In: first_process
edi          0x0          0
eip          0xc0020a62      0xc0020a62 <first_process()>
eflags        0x202        [ IF ]
cs           0x2b          43
ss           0x3b          59
ds           0x33          51
es           0x33          51
---Type <return> to continue, or q <return> to quit---
```

Screenshot 3 (Bottom): The terminal window title is "Terminal". It shows the assembly code for `first_process`. The instruction at address `0xc0020a62` is highlighted. The output shows the register dump for thread 1. The command `(gdb) n` is entered, indicating single-stepping.

```
File Edit View Search Terminal Help
./src/kernel/setup.cpp
27         first, second, third, forth, fifth);
28     return first + second + third + forth + fifth;
29 }
30
31 void first_process()
32 {
33     asm_system_call(0, 132, 324, 12, 124);
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
remote Thread 1 In: first_process
eax          0x0          0
ecx          0x0          0
edx          0x0          0
ebx          0x0          0
esp          0x8049000      0x8049000
ebp          0x0          0x0
esi          0x0          0
---Type <return> to continue, or q <return> to quit---
```

这里看到我们得到的段选择子，栈指针对应我们在 load_process_ 进程所设置的。因此我们成功在启动进程前，把段选择子等信息放到栈中。我们再进一步分析。

再 load_process 里面我们有一个 ProcessStartStack 类的变量 interruptStack 代表启动进程前放入栈的内容。我们对应的放入 USER_DATA_SELECTOR 放入 es, fs, ds, USER_CODE_SELECTOR 放入到 cs, 而 USER_STACK_SELECTOR 放入到 ss。再第 26 行的代码语句：

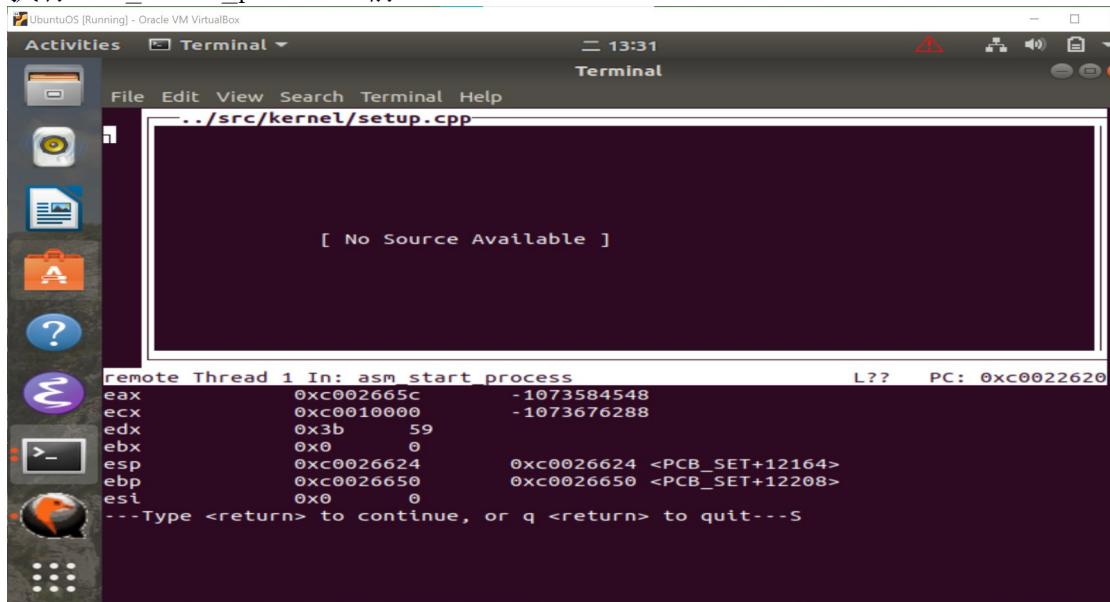
```
interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1);
```

因为每一个特权级都有自己的栈，所以通过内存管理器在用户虚拟空间中分配一页来作为进程的特权级 3 栈。注意到我们定义了每一个进程的 PCB 的用户虚拟地址池的开始地址 USER_VADDR_START 为 0x8048000。因此栈的低地址为 0x8048000，而因为我们分配一页，所以我们的栈指针指向栈的栈顶的地址，也就是

`esp = USER_V_ADDR_START + PAGE_SIZE = 0x8048000 + 4096 = 0x8049000`
这跟我们 gdb 显示的结果是对应的。

之后，我们对于第二次和第三次调用进程进行分析：

执行 `asm_start_process` 前：



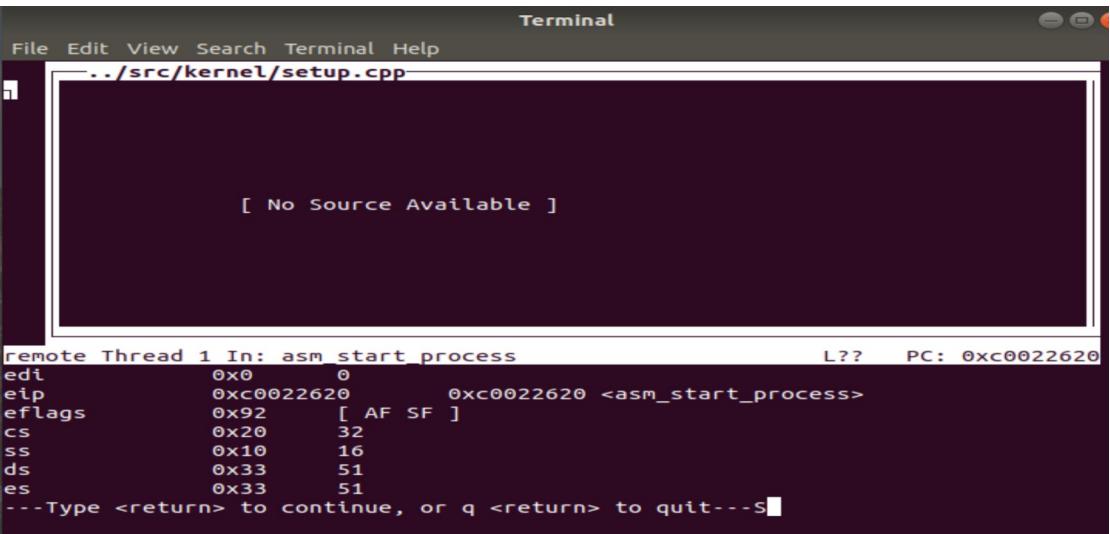
The screenshot shows a terminal window titled "Terminal" running on an Ubuntu OS. The command `./src/kernel/setup.cpp` is being executed. The output shows:

```
[ No Source Available ]
```

remote Thread 1 In: asm start process L?? PC: 0xc00222620

	eax	ecx	edx	ebx	esp	ebp	esi
remote	0xc002665c	0xc0010000	0x3b	0x0	0xc0026624	0xc0026650	0x0
Thread	-1073584548	-1073676288	59	0	0xc0026624 <PCB_SET+12164>	0xc0026650 <PCB_SET+12208>	0
1							
In:							
asm							
start							
process							

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



The screenshot shows a second terminal window titled "Terminal" running on the same Ubuntu OS. The command `./src/kernel/setup.cpp` is being executed. The output shows:

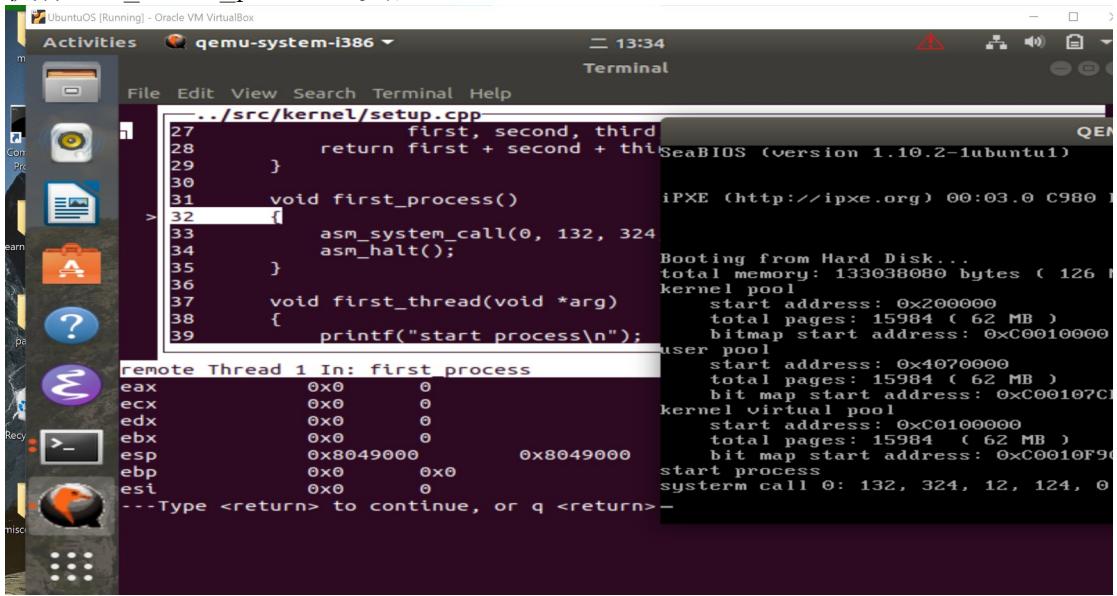
```
[ No Source Available ]
```

remote Thread 1 In: asm start process L?? PC: 0xc00222620

	edi	eip	eflags	cs	ss	ds	es
remote	0x0	0xc00222620	0x92	0x20	0x10	0x33	0x33
Thread	0	0xc00222620 <asm_start_process>	[AF SF]	32	16	51	51
1							
In:							
asm							
start							
process							

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

执行 `asm_start_process` 以后:



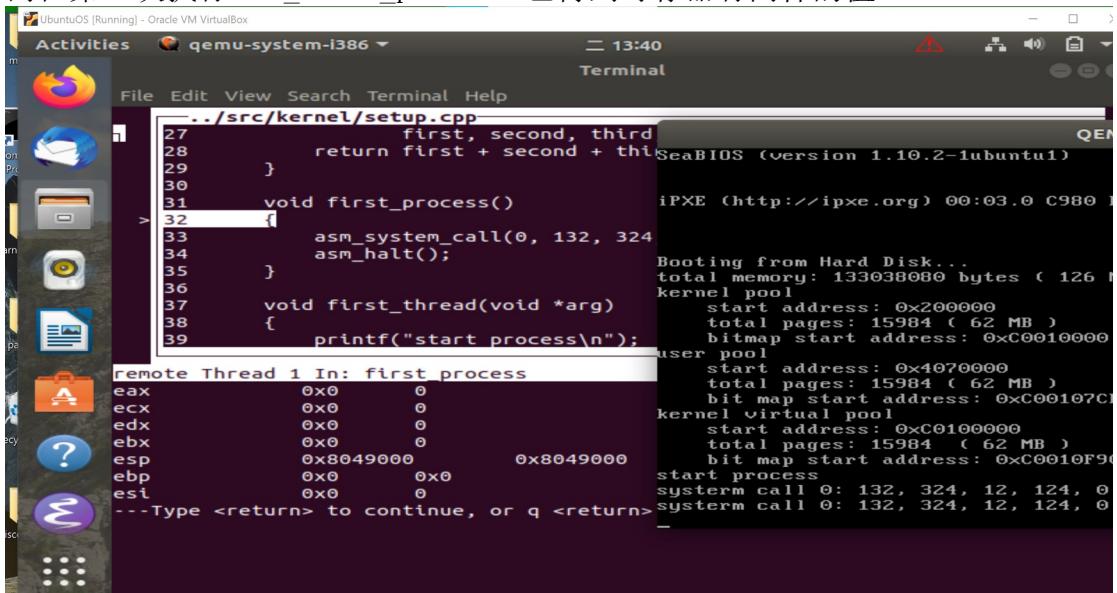
The screenshot shows a terminal window titled "Activities" running on "qemu-system-i386". The window displays assembly code from "setup.cpp" and its execution output. The assembly code includes calls to `asm_system_call`, `asm_halt`, and `printf`. The terminal output shows memory dump information for kernel and user pools, starting addresses, and total pages. The寄存器 dump section shows the state of registers (eax, ecx, edx, ebx, esp, ebp, esi) across three threads. The stack pointer (esp) for all threads is consistently shown as 0x8049000.

```
File Edit View Search Terminal Help
./src/kernel/setup.cpp
27     first, second, third
28     return first + second + thi
29 }
30
31 void first_process()
32 {
33     asm_system_call(0, 132, 324
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
remote Thread 1 In: first process
eax    0x0      0
ecx    0x0      0
edx    0x0      0
ebx    0x0      0
esp   0x8049000      0x8049000
ebp    0x0      0x0
esi    0x0      0
---Type <return> to continue, or q <return>--
```

Seabios (version 1.10.2-1ubuntu1)
IPXE (http://ipxe.org) 00:03.0 C980
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: 0xC00107C0
kernel virtual pool
 start address: 0xC0100000
 total pages: 15984 (62 MB)
 bit map start address: 0xC0010F90
start process
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0

我们得到同样的结果。

而在第三次执行 `asm_start_process` 也得到寄存器有同样的值。



This screenshot is identical to the one above, showing the same assembly code execution and memory dump. The寄存器 dump section again shows the stack pointer (esp) for all threads at 0x8049000.

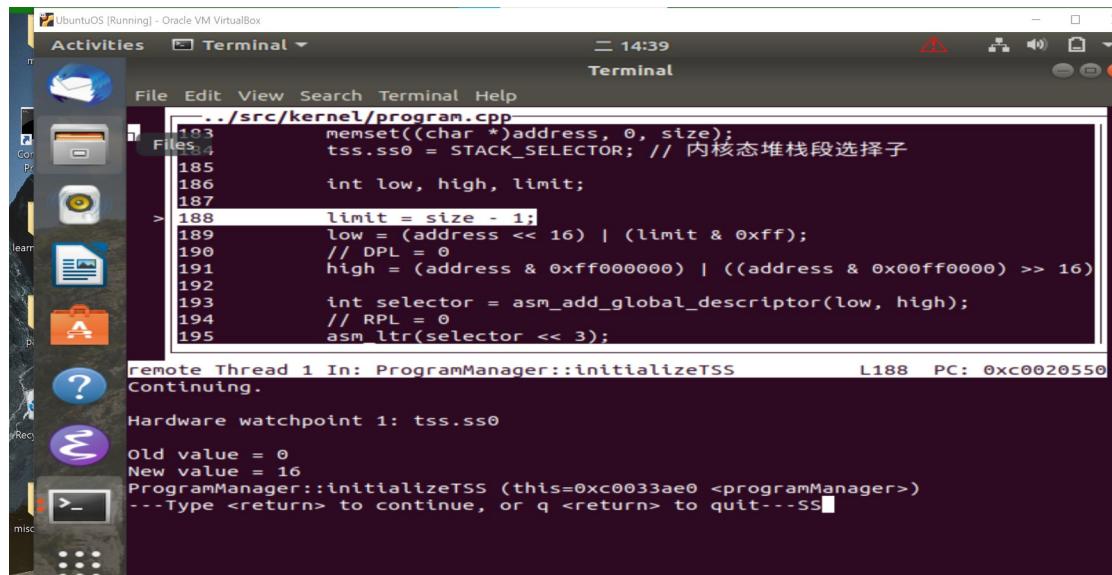
```
File Edit View Search Terminal Help
./src/kernel/setup.cpp
27     first, second, third
28     return first + second + thi
29 }
30
31 void first_process()
32 {
33     asm_system_call(0, 132, 324
34     asm_halt();
35 }
36
37 void first_thread(void *arg)
38 {
39     printf("start process\n");
remote Thread 1 In: first process
eax    0x0      0
ecx    0x0      0
edx    0x0      0
ebx    0x0      0
esp   0x8049000      0x8049000
ebp    0x0      0x0
esi    0x0      0
---Type <return> to continue, or q <return>--
```

Seabios (version 1.10.2-1ubuntu1)
IPXE (http://ipxe.org) 00:03.0 C980
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: 0xC00107C0
kernel virtual pool
 start address: 0xC0100000
 total pages: 15984 (62 MB)
 bit map start address: 0xC0010F90
start process
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0

我们在这里看到 3 次启动进程前 esp 栈指针的值都是 0x8049000 因为我们每一个进程都有它自己的用户虚拟空间。而我们在用户虚拟空间都分配一页作为栈的空间。因此我们就验证了通过 iret 的中断指令，在启动进程前，我们把低特权栈的信息在进入中断前被保存在高特权栈中，从而能够让进程使用低特权级栈的 SS, ESP。

Task state segment (TSS) 本案例来是用于 CPU 原生的进程且黄的转台保存器，弹我们并不打算使用 CPU 原生的进程切换方案，而是使用和 Linux 一样的进程切换方案，因此我们在本实验 TSS 的作用仅限于向 CPU 传送进程特权转移时候的栈段选择子和栈指针。

我们观察 tss.ss0 的值，他是在初始化函数 initializeTSS() 设置为 tss.ss0 = 16



A screenshot of a terminal window titled "UbuntuOS [Running] - Oracle VM VirtualBox". The window shows assembly code from a file named "program.cpp" with line numbers 183 to 195. The code initializes a Task State Segment (TSS) with a stack selector and sets the SS register to 16. Below the code, the terminal shows the assembly code being run and a hardware watchpoint for tss.ss0. The value of tss.ss0 is shown as 16.

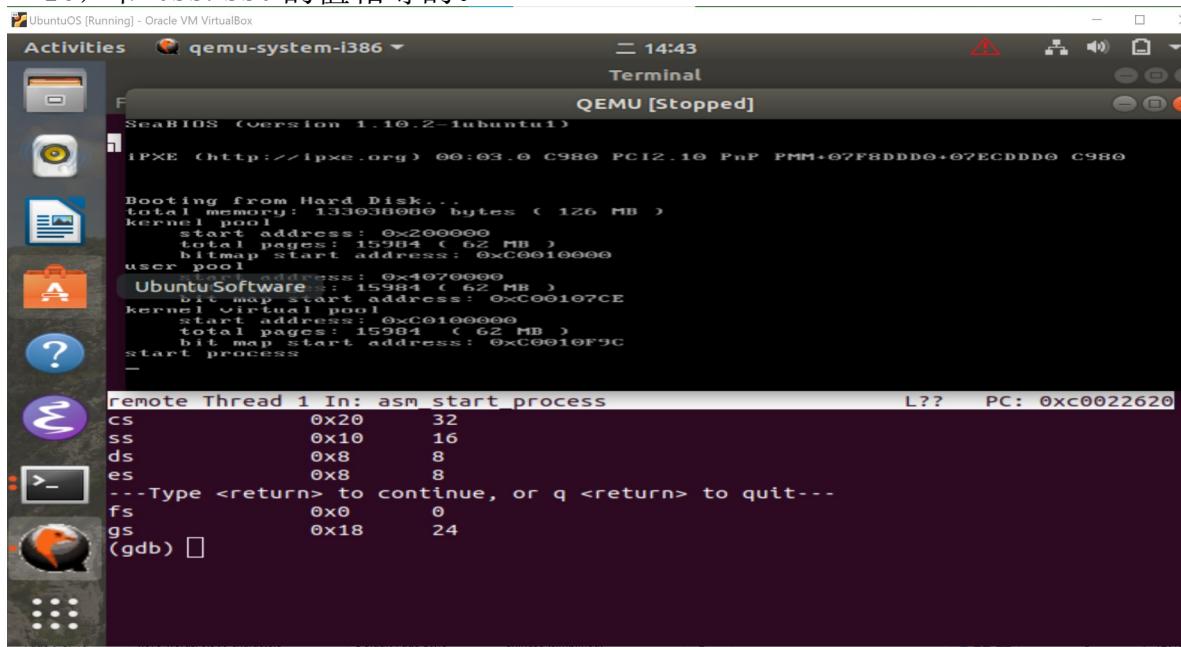
```
File Edit View Search Terminal Help
./src/kernel/program.cpp
183     memset((char *)address, 0, size);
184     tss.ss0 = STACK_SELECTOR; // 内核态堆栈段选择子
185
186     int low, high, limit;
187
188     limit = size - 1;
189     low = (address << 16) | (limit & 0xff);
190     // DPL = 0
191     high = (address & 0xffff0000) | ((address & 0x00ff0000) >> 16)
192
193     int selector = asm_add_global_descriptor(low, high);
194     // RPL = 0
195     asm_ltr(selector << 3);

remote Thread 1 In: ProgramManager::initializeTSS
Continuing.

Hardware watchpoint 1: tss.ss0

Old value = 0
New value = 16
ProgramManager::initializeTSS (this=0xc0033ae0 <programManager>)
---Type <return> to continue, or q <return> to quit---SS
```

而我们发现栈段寄存器在 asm_start_process 断点前设置为 ss = 16，和 tss.ss0 的值相等的。



A screenshot of a terminal window titled "Activities" and "qemu-system-i386". The window shows QEMU boot logs for SeaBIOS version 1.10.2-ubuntu1. It details memory pools and booting from a hard disk. Below the logs, the terminal shows assembly code for the "asm start process" instruction, including register values cs, ss, ds, es, fs, and gs. The value for ss is listed as 16.

```
Activities qemu-system-i386 Terminal 14:43 QEMU [Stopped]

SeaBIOS (version 1.10.2-ubuntu1)
IPXE (http://ipxe.org) 00:03.0 C980 PCI2.1@ PnP PMM+07F80000+07EC0000 C980

Booting from Hard Disk...
total memory: 133030000 bytes ( 126 MB )
kernel pool
    start address: 0x200000
    total pages: 15904 ( 62 MB )
    bitmap start address: 0xC0010000
user pool
    start address: 0x4070000
UbuntuSoftware 15904 ( 62 MB )
    bit map start address: 0xC00107CE
kernel virtual pool
    start address: 0xC0100000
    total pages: 15904 ( 62 MB )
    bit map start address: 0xC0010F9C
start process

remote Thread 1 In: asm start process L?? PC: 0xc0022620
cs      0x20      32
ss      0x10      16
ds      0x8       8
es      0x8       8
fs      0x0        0
gs      0x18      24
---Type <return> to continue, or q <return> to quit---
(gdb) 
```

对于 tss.esp0，我们是在 schedule 函数每一次需要调换进程的时候执行 activateProgramPage() 时设置 tss.esp0 的值为进程函数地址加上一页的大小：
tss.esp0 = (int)program + PAGE_SIZE，我们第一次进行进程调换设置
tss.esp0 = 0xc00256a0

图一

A screenshot of a Linux desktop environment showing a terminal window. The terminal title is 'Terminal' and the date/time is '14:56'. The window contains assembly code from file '/src/kernel/program.cpp' and a GDB session. The assembly code includes instructions like 'int paddr = PAGE_DIRECTORY;', 'if (program->pageDirectoryAddress)', and 'asm_update_cr3(paddr);'. The GDB session shows a print command: '(gdb) print /x tss.esp0' with output '\$1 = 0xc00256a0'. The terminal window has a dark background with light-colored text.

图二

A screenshot of a Linux desktop environment showing a terminal window. The terminal title is 'Terminal' and the date/time is '14:56'. The window contains assembly code from file '/src/kernel/program.cpp' and a GDB session. The assembly code is identical to Figure 1. The GDB session shows a print command: '(gdb) print /x tss.esp0' with output '\$1 = 0xc00256a0'. Below the assembly code, there is a register dump:
eax 0xc00256a0 -1073588576
ecx 0x1 1
edx 0x0 0
ebx 0x0 0
esp 0xc0024588 0xc0024588 <PCB_SET+3816>
ebp 0xc00245a0 0xc00245a0 <PCB_SET+3840>
esi 0x0 0
---Type <return> to continue, or q <return> to quit---
The terminal window has a dark background with light-colored text.

图三

QEMU
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PC
Booting from Hard Disk...
total memory: 133038080 bytes (126 MB)
kernel pool
start address: 0x200000
total pages: 15984 (62 MB)
bitmap start address: 0xC0010000
user pool
start address: 0x4070000
total pages: 15984 (62 MB)
bit map start address: 0xC0010F9C
kernel virtual pool
start address: 0xC0100000
total pages: 15984 (62 MB)
bit map start address: 0xC0010F9C
remote Thread 1 In: ProgramManager::activate
Old value = -1073588576
New value = -1073584480
ProgramManager::activateProgramPage (this=0xstart process
---Type <return> to continue, or q <return> system call 0: 132, 324, 12, 124, 0
program=0xc00255a0 <PCB_SET+8192>) at .
(gdb) print /x tss.esp0
\$2 = 0xc00266a0
(gdb)

图四

QEMU
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PC
Booting from Hard Disk...
total memory: 133038080 bytes (126 MB)
kernel pool
start address: 0x200000
total pages: 15984 (62 MB)
bitmap start address: 0xC0010000
user pool
start address: 0x4070000
total pages: 15984 (62 MB)
bit map start address: 0xC0010F9C
kernel virtual pool
start address: 0xC0100000
total pages: 15984 (62 MB)
bit map start address: 0xC0010F9C
remote Thread 1 In: ProgramManager::activate
Old value = -1073584480
New value = -1073580384
ProgramManager::activateProgramPage (this=0xstart process
---Type <return> to continue, or q <return> system call 0: 132, 324, 12, 124, 0
program=0xc00266a0 <PCB_SET+12288>) at .
(gdb) print /x tss.esp0
\$3 = 0xc00276a0
(gdb)

因此我们发现每次在进程调换的时候都会更新 tss.esp0 的值。注意到每一次更新的 tss.esp0 的值都相差 0x1000 (4096 或 4KB)，也就是一页的大小，因此我们是连续地分配。

在每一次进程调换，CPU 每次都使用中断（执行汇编代码 int 0x80）从**低特权级转移到高特权级**。CPU 首先会从 TSS (Task State Segment) 中取出高特权级的段选择子和栈指针，然后将高特权级的段选择子和栈指针送入 SS, ESP。此时，栈发生变化，此时的栈已经变化了 TSS 保存的高特权级的栈。接着，中断发生前的 SS, ESP, EFLAGS, CS, EIP 被依次压入高特权级栈。我们从图 2 可以看到 ebp 的值为 ebp = 0xc00245a0，而 esp = 0xc0024588，当中相差 0x18 = 24 = 6*4，因此可见栈会把中断发生前的 SS, ESP, EFLAGS, CS, EIP 6 个值推进栈上。

另外我们注意到 tss.esp0 指向的地址为内核空间里的，因为我们谁设定加载内核的内核虚拟地址为 **KERNEL_VIRTUAL_ADDRESS = 0xc0020000**

我们比较以下图 1 和图 2，发现在 activateProgramPage 的时候，我们只是更新了 tss.esp0 为下一个进程的栈的值 (0xc00255a0)，而 esp 没有更新，依然为 esp = 0xc00245a0。因此我们是在调用 0x80 中断的时候 CPU 才使用到 tss.esp0 的值，也就是进程 0 特权级栈的地址，放入栈指针 esp 的。

assignment 2

首先通过设置断点为 copyProcess()，在 ProgramManager::fork() 函数里面把父进程的 0 级栈拷贝到子进程的 0 级栈里面的过程。copyProcess 的作用就是把中断的那一刻保存的寄存器的内容复制到子进程的 0 特权级栈中。

图 1，上通过 x /17x parentpss 指令看到父进程 0 级栈的内容。我们在这里要注意的是当中的第 16 个寄存器 esp = 0x08048f98，

图 1

The screenshot shows a terminal window on an Ubuntu OS system. The terminal title is "Terminal". The command run is ". ./src/kernel/program.cpp". The assembly code shown is:

```
381     // 复制进程0级栈
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     // 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;
```

Below the code, a memory dump is shown for "Remote Thread 1 In: ProgramManager::copyProcess". The dump includes registers and memory addresses:

Register	Value	Register	Value
0xc0025d9c <PCB_SET+8124>	0x00000000	0x00000000	0x08048fac
0xc0025dbc		0x00000000	0x00000000
0xc0025dac <PCB_SET+8140>	0x00000000	0x00000000	0x00000000
0x00000002		0x000000033	0x000000033
0xc0025d3c <PCB_SET+8156>	0x00000000	0x000000033	0x000000033
0x000000033		0x00000002b	0x000000216
0xc0025dcc <PCB_SET+8172>	0xc0022ccf	0x00000002b	0x000000216

At the bottom, the prompt is: "----Type <return> to continue, or q <return> to quit--".

第 12 个寄存器 eip = 0xc0022ccf。然后，在执行了 `memcpy(parentpss, childpss, sizeof(ProcessStack))` 以后，我们发现 childpss 的内容和 parentpss 相等。而我们发现地址 eip=0xc0022ccf 为 `asm_system_call_handler` 的返回地址。

之后我们分析一下，第一次执行 `asm_start_process` 的时候，执行到 `iret` 的结果。在第一次执行 `asm_start_process` 里，我们期望得到跟上一次实验相同的结果，也就是执行 `iret` 后启动进程 `first_process()`。

The screenshot shows a terminal window on an Ubuntu OS system. The terminal title is "Terminal". The command run is ". ./src/kernel/setup.cpp". The assembly code shown is:

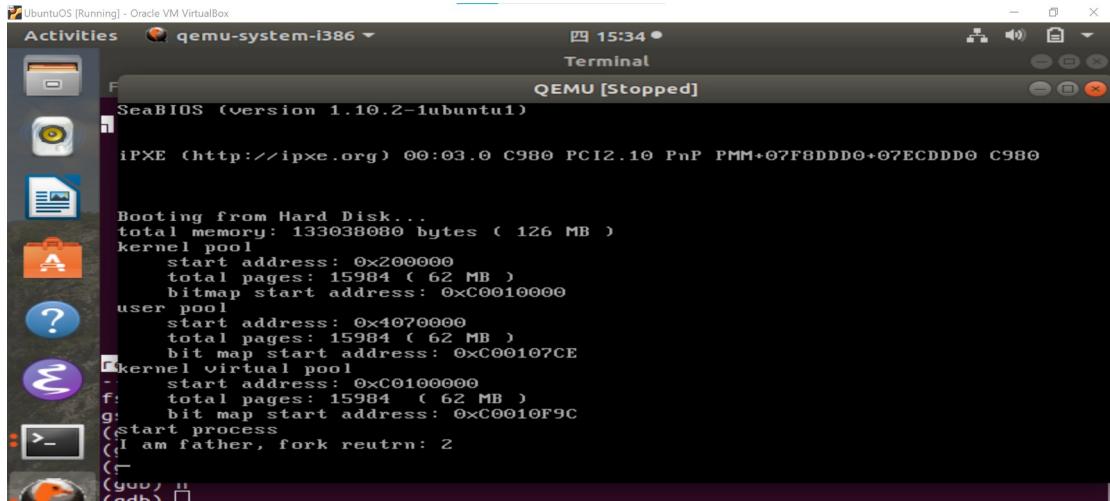
```
27             first, second, third, forth, fifth);
28         return first + second + third + forth + fifth;
29     }
30
31     void first_process()
32     {
33         int pid = fork();
34
35         if (pid == -1)
36         {
37             printf("can not fork\n");
38         }
39         else
```

Below the code, a memory dump is shown for "Remote Thread 1 In: first_process". The dump includes registers and memory addresses:

Register	Value	Register	Value
cs	0x2b	43	
ss	0x3b	59	
ds	0x33	51	
es	0x33	51	
fs	0x33	51	
gs	0x0	0	

At the bottom, the prompt is: "----Type <return> to continue, or q <return> to quit--".

之后，一直执行 `next` 指令知道运行到 `if` 语句块里面，输出以下。这就执行完父进程的代码部分。



之后，我们在 gdb 一直执行 next 指令到执行完 first_process 的代码部分以后，我们发现在代码跳转到 asm_start_process。注意这是因为我们在 copyProcess() 初始化子进程的 0 级栈为以下内容。

```

// 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

```

当中，这使得在调换线程的时候，会执行 asm_start_process 的代码。因为我们先前把 0 特权栈拷贝到 childpss 里面，所以我们在启动函数 asm_start_process 执行了 iret 以后会把 0 特权级栈的 eip 送入 eip 中，并执行该地址的代码。如下通过在 asm_start_process() 的 eip 再执行一次 next 指令，会跳到到保存再 0 特权级的栈的 eip = 0xc0022ccf，也就是 asm_system_call 的代码地址，（这个结果在下面图 2 里面执行到 asm_start_process，通过 next 指令跳到 asm_system_call 的地址验证了。）当中会调用系统号为 2 的系统调用，也就是 syscall_fork()。最后从 fork 返回。之后执行了 fork 调用以后（图 3），又跳转到 first_process() 里面（图 4）。

图 2

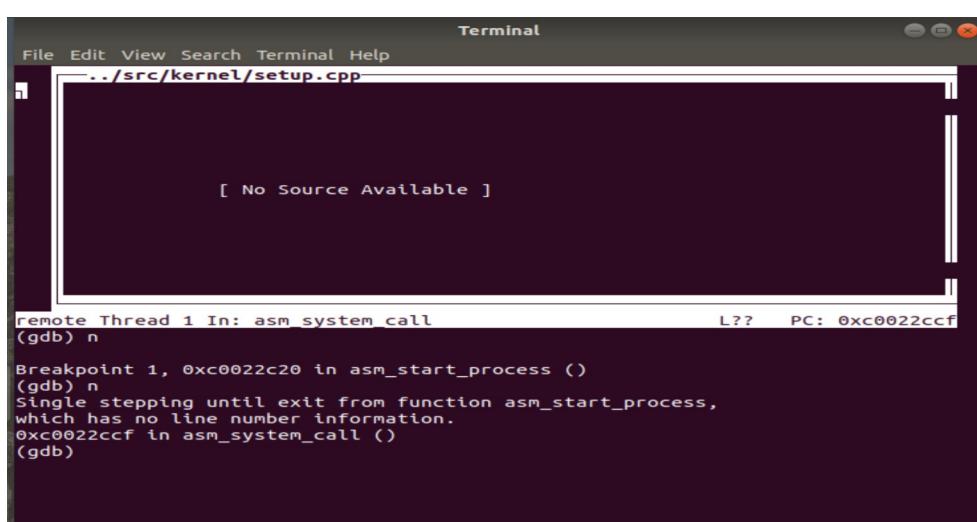


图 3

A screenshot of a terminal window titled "Terminal". The window shows assembly code from a file named "syscall.cpp". The code includes a "fork" function and a "syscall_fork" wrapper. The assembly code is annotated with line numbers 32 through 44. The terminal then displays a GDB session for a remote thread. It shows the instruction at address 0xc0020f71, which is a single-step operation until it exits the "asm_start_process" function, which has no line number information. The assembly code for "asm_system_call" is shown as 0xc0022ccf. The user then types "n" to continue, and the next instruction is at address 0xc0020fd7, which is a single-step operation until it exits the "asm_system_call" function, which also has no line number information. The assembly code for "fork" is shown as "fork () at/src/kernel/syscall.cpp:37".

```
File Edit View Search Terminal Help
./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 In: fork
Single stepping until exit from function asm_start_process,
which has no line number information.
0xc0022ccf in asm_system_call ()
(gdb) n
Single stepping until exit from function asm_system_call,
which has no line number information.
fork () at ..../src/kernel/syscall.cpp:37
(gdb) 
```

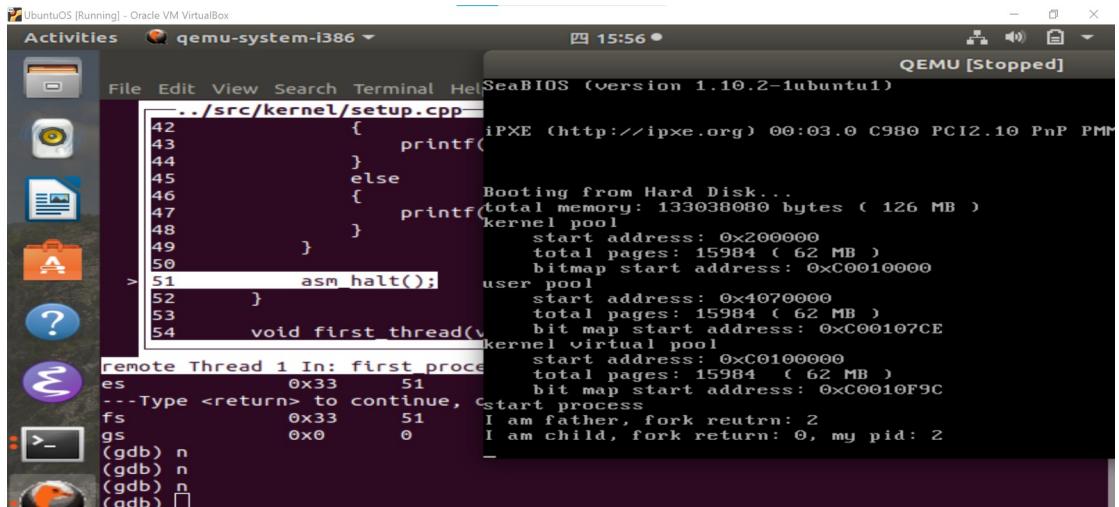
图 4

A screenshot of a terminal window titled "Terminal". The window shows assembly code from a file named "setup.cpp". The code includes a "first_process" function. The assembly code is annotated with line numbers 30 through 42. The terminal then displays a GDB session for a remote thread. It shows the instruction at address 0xc0020fd7, which is a "fork" operation. The assembly code for "fork" is shown as "fork () at/src/kernel/syscall.cpp:37". The user then types "n" to continue, and the next instruction is at address 0xc0020fd7, which is a "print pid" operation. The assembly code for "first_process" is shown as "first_process () at/src/kernel/setup.cpp:35". The user then types "info register eax", and the output shows "eax 0x0 0".

```
UbuntuOS [Running] - Oracle VM VirtualBox
Activities Terminal Terminal
File Edit View Search Terminal Help
./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 In: first_process
fork () at ..../src/kernel/syscall.cpp:37
(gdb) n
first_process () at ..../src/kernel/setup.cpp:35
(gdb) print pid
$1 = 0
(gdb) info register eax
eax          0x0      0
(gdb) 
```

从我们发现 pid 的值为 0，而观察到 eax 的值为 0，这是因为再 copyProcess() 函数里面把 childpps→eax 设置为 0，从而保证子进程调用 fork() 的返回值为 0。这就反映了子进程调用 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。最后我们得出以下的结果。



当中，我们注意到，第一句写着“`I am father, fork return: 2`”，意味着这是父进程的执行部分，`fork()`返回的值为 2，第二句为“`I am child, fork return: 0, my pid: 2`”，意味着这是子进程的执行部分，`fork()`返回的值为 0，而它的进程 ID 为 2。

通过比较父进程的`fork()`返回值和子进程返回值，我们发现父进程返回子进程的进程 ID，而子进程的`fork()`返回值返回 0。从`programManager.fork()`函数里面看到，函数返回的`pid`是从`executeProcess("", 0)`得到的，而父进程就是返回`programManager.fork()`的值。而因为我们在调用`executeProcess`当中我们给出的函数地址，也就是函数名`filename`为`NULL`，`priority`设置为 0，这就意味着我们。我们设置`filename`输入变量为`NULL`，是因为在`copyProcess`里面会在子进程的 0 级栈里面设置调用放函数地址`asm_start_process`，使得在`asm_switch_thread`的时候调用`asm_start_process`。因此通过以下代码：

```
// 创建子进程
int pid = executeProcess("", 0);
```

它同时通过在`executeProcess()`里面调用`executeThread()`而`executeThread`调用`allocatePCB`以分配一个子进程的 PCB。并同时定义了

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

这就同时分配了一个子进程的 PCB，而且使得`programManager.fork()`返回子进程的 ID。

对于子进程，我们在`copyProcess`函数里的代码设定了`childpss->eax = 0`使得子进程的`fork()`返回值为 0。

最后我们通过代码逻辑分析`fork`调用如何在进程之间复制资源。

在`copyProcess`函数里面，除了把父进程的 0 级栈拷贝到子进程里，还需要把父子进程使用到相同的代码，因此需要共享代码。首先我们把父进程的管理虚拟地址池的`bitmap`拷贝到的子进程的`bitmap`里：

```
int bitmapLength = parent->userVirtual.resources.length;
int bitmapBytes = ceil(bitmapLength, 8);
memcpy(parent->userVirtual.resources.bitmap,
child->userVirtual.resources.bitmap, bitmapBytes);
```

之后，需要从内核中分配一夜来作为数据复制的中转页。这是因为我们通过分

页机制实现了地址隔离，父进程无法将数据复制到具有相同虚拟地址的子进程中。通过内核空间的中转页，先把父进程的虚拟地址空间下把数据复制到中转页中，再切换到子进程的虚拟地址空间中，然后把中转页复制到子进程对应的位置。主要实现的代码如下：

```
// pageVaddr 是物理页的起始虚拟地址。  
void *pageVaddr = (void *) (i << 22) + (j << 12));  
// 复制出父进程的物理页的内容到中转页  
memcpy(pageVaddr, buffer, PAGESIZE);  
// 从中转页中复制到子进程的物理页  
memcpy(buffer, pageVaddr, PAGE_SIZE);  
因此，我们就实现到父进程和子进程的代码共享。
```

assignment 3:

有时候我们的进程或者线程希望主动结束运行，在linux操作系统就有系统调用exit负责这个功能。

当中我们有两个函数

```
void exit(int ret)  
void syscall_exit(int ret)
```

当中syscall_exit就是对应系统调用号3的系统调用函数。exit就是从系统调用表找出该系统调用执行，并有输入参数ret，代表返回值。

在PCB里面我们有一个返回值变量int retValue，来代表系统调用exit后的返回值。调用exit是提供了返回值，而返回值的处理实在线程或进程的PCB回收是进行的。

我们注意到exit是提供ProgramManager::exit实现的。而exit主要分为以下三部分：

1. 标记PCB状态为DEAD并放回返回值
2. 如果PCB标识是进程，就释放进程所占用的物理页，页目录表和虚拟地址池bitmap的空间。否者不做处理
3. 立刻执行线程或进程的调度

我们进一步分析ProgramManager::exit的代码

```

1 void ProgramManager::exit(int ret)
2 {
3     // 关中断
4     interruptManager.disableInterrupt();
5
6     // 第一步，标记PCB状态为`DEAD`并放入返回值。
7     PCB *program = this->running;
8     program->returnValue = ret;
9     program->status = ProgramStatus::DEAD;
10
11    int *pageDir, *page;
12    int paddr;
13
14    // 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。
15    if (program->pageDirectoryAddress)
16    {
17        pageDir = (int *)program->pageDirectoryAddress;
18        for (int i = 0; i < 768; ++i)
19        {
20            if (!(pageDir[i] & 0x1))
21            {
22                continue;
23            }
24
25            page = (int *) (0xffc00000 + (i << 12));
26
27            for (int j = 0; j < 1024; ++j)
28            {
29                if (!(page[j] & 0x1))
30                {
31                    continue;
32                }
33
34                paddr = memoryManager.vaddr2paddr((i << 22) + (j << 12));
35                memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);
36            }
37
38            paddr = memoryManager.vaddr2paddr((int)page);
39            memoryManager.releasePhysicalPages(AddressPoolType::USER, paddr, 1);
40        }
41
42        memoryManager.releasePages(AddressPoolType::KERNEL, (int)pageDir, 1);
43
44        int bitmapBytes = ceil(program->userVirtual.resources.length, 8);
45        int bitmapPages = ceil(bitmapBytes, PAGE_SIZE);
46
47        memoryManager.releasePages(AddressPoolType::KERNEL, (int)program->userVirtual.resources.bitmap, bitmapPages);
48    }
49
50    // 第三步，立即执行线程/进程调度。
51    schedule();
52
53 }

```

在第 7-9 行，设置正在运行的进程 program 的 PCB 的返回值 `program->returnValue = ret`，另外状态为 DEAD。

在第 15-48 行，如果 program 为一个进程而不是线程就把进程所占用的物理页，页目录表和虚拟地址池 bitmap 的空间进行释放。我们是通过第 15 行的 if 语句判断是进程还是线程的，因为进程有页目录表，而线程没有。

在 17 行定义 `pageDir` 指向页目录表的地址。

在 18-39 行，我们对于页目录表指向的页表的物理地址以及每一个页表指向的物理页进行释放。

在第 25 行，定义 `page` 为对应页目录表项指向的页表的物理地址。在第 37 行找出了它的物理页地址，并在第 38 行释放它。

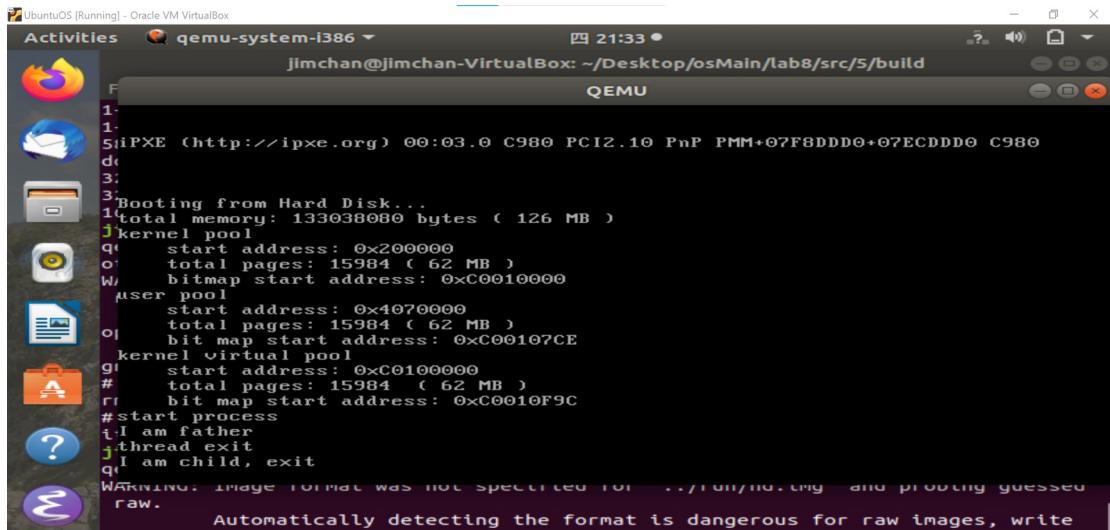
第 27-35 行是释放每一个页表的所有物理页。在第 33 行找出物理页的地址，并在第 34 行释放它。

第 41 行释放指向页目录表的物理页。

在 43-46 行释放 bitmap。

最后在第 51 行执行调度函数 `schedule()`

之后运行 /src/5/ 的代码。



我们注意到在上一个实验我们的 fork 调用没有实现到一个返回机制，我们是需要在进程的结尾加上 `asm_halt()` 来阻止进程返回。为了实现返回机制，可以使用到我们实现的退出函数 `exit`。通过在进程的 3 特权级栈的顶部放入 `exit` 的地址和参数，使得参数的函数推出后主动跳转到 `exit`。

通过在 `load_process` 函数里面，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，自动跳入 `exit` 以结束该进程。因为 `fork` 实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通过 `interruptStack` 的栈调用 `exit(0)`，保证了进程推出后能够肯定第调用 `exit` 和此时的 `exit` 返回值为 0。

在我们图中所显示的实例上展示了 `exit` 的运作原理。这个实例有一个 `first_process` 进程和 `second_thread` 线程按顺序执行。首先执行第一个进程 `first_process`，在执行了父进程以后，`schedule` 函数会通过 RR 调度算法到 `second_thread` 执行（因为 `first_process` 已经执行到 `time quantum` 的时间长度）。在执行 `second_thread` 会打印“`thread exit`”然后执行 `exit(0)` 从而结束进程。因为 `exit` 函数的代码最后会执行 `schedule()` 以调度进程，所以会继续执行进程 `first_process` 的子进程部分，并打印出“`I am child, exit`”。执行子进程的代码部分和上一个实验没有什么不同，但是在 `first_process` 里面，子进程执行 `else` 语句的代码部分不需要 `asm_halt` 要结束进程。而整体来说我们目前的 `first_process()` 代码跟上一个实验的 `first_process()` 的不同之处在于我们现在的 `first_process` 只是在执行完父进程的代码部分需要执行 `asm_halt()`，而执行子进程的代码部分最后不需要执行 `asm_halt()`，这是因为我们先前再 `load_process()` 函数里把 `first_process` 的第三级栈的顶部加入了 `exit` 函数地址，`exit` 的地址 (0) 和参数 (0)，这样使得 `first_process` 最后会执行 `exit(0)` 来退出并结束进程。

之后分析系统调用 `wait`。我们前面提到，再进程的状态被标记为 `DEAD` 后并被清除，而是再等待其他进程来回收进程的 PCB。进程的 PCB 是通过 `wait` 系统调用 来等待其子进程执行完成并回收子进程。

```

int wait(int *retval);
int syscall_wait(int *retval);

```

`wait` 函数的参数 `retval` 用来存放子进程的返回值。如果 `retval==nullptr`，说

明我们不关心子进程的返回值。wait 的返回值是被回收的子进程。如果没有子进程，wait 会返回-1。再付进程调用了 wait 后，如果存在子进程的状态不是 DEAD，则父进程会一直被阻塞，则 wait 不会返回直到子进程结束。

wait 函数的实现是通过 ProgramManager::wait 来实现的：

```
1 int ProgramManager::wait(int *retval)
2 {
3     PCB *child;
4     ListItem *item;
5     bool interrupt, flag;
6
7     while (true)
8     {
9         interrupt = interruptManager.getInterruptStatus();
10        interruptManager.disableInterrupt();
11
12        item = this->allPrograms.head.next;
13
14        // 查找子进程
15        flag = true;
16        while (item)
17        {
18            child = ListItem2PCB(item, tagInAllList);
19            if (child->parentPid == this->running->pid)
20            {
21                flag = false;
22                if (child->status == ProgramStatus::DEAD)
23                {
24                    break;
25                }
26                item = item->next;
27            }
28
29            if (item) // 找到一个可返回的子进程
30            {
31                if (retval)
32                {
33                    *retval = child->returnValue;
34                }
35
36                int pid = child->pid;
37                releasePCB(child);
38                interruptManager.setInterruptStatus(interrupt);
39                return pid;
40            }
41        else
42        {
43            if (flag) // 子进程已经返回
44            {
45
46                interruptManager.setInterruptStatus(interrupt);
47                return -1;
48            }
49            else // 存在子进程，但子进程的状态不是DEAD
50            {
51                interruptManager.setInterruptStatus(interrupt);
52                schedule();
53            }
54        }
55    }
56 }
57
58 }
```

从第 12 行到第 28 行我们从 allPrograms 找到一个状态为 DEAD 的子进程。

从第 30-41 行，我们从之前找出来可回收子进程。

在第 32-35 行的 if 语句，当 retval 不为 nullptr 时，我们取出子进程的返回值放入到 retval 指向的变量中

在第 37 行，取出子进程的 pid 定义为变量 pid。

在第 38 行调用 releasePCB 来回收子进程的 PCB

在第 39 行返回子进程的 pid。

releasePCB 的函数实现如下：

```
void ProgramManager::releasePCB(PCB *program)
{
    int index = ((int) program - (int) PCB_SET) / PCB_SIZE;
    PCB_SET_STATUS[index] = false;
    this->allPrograms.erase(&(program->tagInAllList));
}
```

注意到第 5 行时我们现在实现的 releasePCB 函数增加的语句。

第 44-49 行的 else 块，我们并没有找到子进程，所有返回-1。

第 50-54 行的语句，是对于存在子进程但子进程的状态不是 DEAD，因此执行调度。注意该进程的主体是一个死循环，因此当进程的第 53 语句 schedule() 返回后，wait 并不会返回，而是再一次重复上面的步骤，尝试回收子进程。也就是前面所说的，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，则父进程会被一直阻塞，则 wait 不会返回直到子进程结束。

我们对于 schedule 调度函数进行修改，增加了处理 DEAD 状态的进程。

```
Else if(running->status == ProgramStatus::DEAD)
{
    // 回收子进程留到父进程回收
    if(!running->pageDirectoryAddress)
    {
        releasePCB(running);
    }
}
```

回收僵尸进程的有效方法

如果一个父进程先于子进程退出，那么紫禁城在退出之前就会被称为孤儿进程 (orphaned process)。子进程退出后，从状态被标记为 DEAD 开始被回收，子进程会被称为僵尸进程 (zombie process)。我们通过在 Linux 环境实现一个例子来看看僵尸进程。

```
Jimchan@Jimchan-VirtualBox: ~/Desktop/osMain/lab8/src/6/src
File Edit View Search Terminal Help
1598 Ssl
1618 Sll
1717 Ssl
1786 Sl+
1823 Sl+
1977 I
2001 I
2051 I
2077 S
2131 Ssl
2141 Ss
2227 R+
jimchan@Jimchan-VirtualBox:~/De 1786 Sl+
child process id: 2231 has pare 1823 Sl+
parent process: 2230 has grand 1977 I
jimchan@Jimchan-VirtualBox:~/De 2001 I
child process id: 2268 has pare 2051 I
^C
jimchan@Jimchan-VirtualBox:~/De 2131 Ssl
jimchan@Jimchan-VirtualBox:~/De 2141 Ss
The child process is2275 2243 Ss
I'm a child. 2263 I
jimchan@Jimchan-VirtualBox:~/De 2286 S+
jimchan@Jimchan-VirtualBox:~/De 2287 Z+
jimchan@Jimchan-VirtualBox:~/De 2288 R+
The child process is2287 2288 R+
I'm a child.
jimchan@Jimchan-VirtualBox:~$
```

从上图可以看到我们执行的子进程 ID 为 2287，而通过 ps -e -o pid, stat 可以看到这个子进程的状态为 Z+，也就是它是一个僵尸进程。这代表了子进程已经执行完毕了以后，但是它依然在进程列表里面而没有被释放。这使得父进程执行完毕以后会先于子进程被释放。

执行/src/6/的代码以实现回收僵尸进程的有效方法

```
UbuntuOS [Running] - Oracle VM VirtualBox
Activities qemu-system-i386
jimchan@Jimchan-VirtualBox: ~/Desktop/osMain/lab8/src/6/build
QEMU
Booting from Hard Disk...
dtotal memory: 133038000 bytes ( 126 MB )
ekernel pool
j start address: 0x200000
j total pages: 15984 ( 62 MB )
j bitmap start address: 0xC0010000
guser pool
j start address: 0x4070000
j total pages: 15984 ( 62 MB )
j bit map start address: 0xC00107CE
1 kernel virtual pool
j start address: 0xC0100000
j total pages: 15984 ( 62 MB )
j bit map start address: 0xC0010F9C
j start process
j thread exit
qexit, pid: 3
Wexit, 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
of
specify the raw format explicitly to remove the restrictions.
```

```

1 void first_process()
2 {
3     int pid = fork();
4     int retval;
5
6     if (pid)
7     {
8         pid = fork();
9         if (pid)
10        {
11            while ((pid = wait(&retval)) != -1)
12            {
13                printf("wait for a child process, pid: %d, return value: %d\n", pid, retval);
14            }
15
16            printf("all child process exit, programs: %d\n", programManager.allPrograms.size());
17
18            asm_halt();
19        }
20        else
21        {
22            uint32 tmp = 0xffffffff;
23            while (tmp)
24                --tmp;
25            printf("exit, pid: %d\n", programManager.running->pid);
26            exit(123934);
27        }
28    }
29    else
30    {
31        uint32 tmp = 0xffffffff;
32        while (tmp)
33            --tmp;
34        printf("exit, pid: %d\n", programManager.running->pid);
35        exit(-123);
36    }
37 }

```

在 first_process 进程里面，我们可以分为父进程，子进程和孙进程。子进程执行的部分为第 29-36 行，孙进程执行的部分为第 20-27 行。首先是子进程执行，在执行完毕后，打印出 “exit, pid: 3”，显示子进程的 ID = 3，并调用 exit(-123)，来结束子进程并设置它的 PCB status = DEAD，返回值 retVal = -123。之后是孙进程执行，在执行完毕后，打印出 “exit, pid: 4”，显示子进程的 ID = 4，并调用 exit(123934)，来结束子进程并设置它的 PCB status = DEAD，返回值 retVal = 123934。

父进程的部分为第 9-19 行。注意到在第 11 行的 while 语句里面：

`while((pid = wait(&retval)) != -1)`

在这个语句里的 wait(&retval) 只要有子进程的状态不为 DEAD，它都一直不会返回的，因此在存在状态不为 DEAD 的子进程的情况下，一直都是死循环的。因此我们对应到运行结果，发现父进程在子进程和孙进程都执行完毕的情况下以后，才会执行 while 语句里面第 13 行的语句，并且对于该可返回的进程进行回收。然后，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 语句的迭代步的 wait(&retVal) 被回收，因此现在的 wait(&retval) = -1，从而跳出 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，当中的 2 代表子进程的总数量。因此我们通过 exit 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. 代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4 #include <unistd.h>
5 #include <sys/wait.h>
6
7 int main()
8 {
9     int c_pid;
10    int pid;
11
12    if((pid = fork()) == -1)
13    {
14        // parent process
15        c_pid = pid;
16        printf("The child process is %d\n", c_pid);
17        sleep(20);
18        exit(0);
19    }
20    else
21    {
22        printf("I'm a child.\n");
23        exit(0);
24    }
25 }
```

4. 总结

在这次实验里，我主要遇到的困难在于理解教程中所讲解的系统调用的运行过程。我自己需要通过在 gdb 调试才能真正体会更深入体会和理解到系统调用实现的原理。同时我发现这个实验需要回顾到之前的实验的知识点，特别是中断、调度、内存管理的实验，来讲解这次实验的原理。