

本科生实验报告

实验课程： 操作系统原理实验

任课教师： 刘宁

实验题目： Lab8 从用户态到内核态

专业名称： 计算机科学与技术

学生姓名： 孙凯

学生学号： 23336212

实验地点： 实验中心B202

实验时间： 2025.6.5

一、实验要求：

在本次实验中，我们首先会简单讨论保护模式下的特权级的相关内容。特权级保护是保护模式的特点之一，通过特权级保护，我们区分了内核态和用户态，从而限制用户态的代码对特权指令的使用或对资源的访问等。但是，用户态的代码有时不得不使用一些特权指令，如输入/输出等。因此，我们介绍了系统调用的概念和通过中断实现系统调用的方法。通过系统调用，我们可以实现从用户态到内核态转移，然后在内核态下执行特权指令，执行完成后返回到用户态。在实现了系统调用后，我们通过三个步骤创建进程。在这里，我们需要重点理解如何通过分页机制来实现进程之间的虚拟地址空间的隔离。最后，本次实验将介绍 fork/wait/exit 的一种简洁的实现思路。

具体内容：

1. 系统调用的实现
2. 进程的创建和调度
3. fork的实现
4. wait&exit的实现

二、预备知识和实验环境：

预备知识： qemu+gdb调试方法

实验环境：

1. 虚拟机版本/处理器型号： Ubuntu 20.04 LTS
2. 代码编辑环境： Vscode+nasm/C/C++插件+qemu仿真平台
3. 代码编译工具： gcc/g++ （64位）
4. 重要三方库信息： 无
5. 代码程序调试工具： gdb

三、实验任务：

- 任务1：实现系统调用的方法
- 任务2：进程的创建与调度
- 任务3：fork的实现
- 任务4：wait&exit的实现

四、实验步骤和实验结果：

实验任务1：实现系统调用的方法

代码放在 lab8/assignment1 中

- 任务要求：复现指导书中“系统调用的实现”一节，并回答以下问题：
 1. 请解释为什么需要使用寄存器来传递系统调用的参数，以及我们是如何在执行 `int 0x80` 前在栈中找到参数并放入寄存器的
 2. 请使用gdb来分析在我们调用了 `int 0x80` 后，系统的栈发生了怎样的变化？`esp` 的值和在 `setup.cpp` 中定义的变量 `tss` 有什么关系？此外还有哪些段寄存器发生了变化？变化后的内容是什么？
 3. 请使用gdb来分析在进入 `asm_system_call_handler` 的那一刻，栈顶的地址是什么？栈中存放的内容是什么？为什么存放的是这些内容？
 4. 请结合代码分析 `asm_system_call_handler` 是如何找到中断向量号 `index` 对应的函数的
 5. 请使用gdb来分析在 `asm_system_call_handler` 中执行 `iret` 后，哪些段寄存器发生了变化？变化后的内容是什么？这些内容来自于什么地方？
- 实验思路：
 1. 定义并实现系统调用函数,其中要实现0x80中断函数,将从用户态进入内核态，由操作系统内核来完成这些服务，然后再从内核态返回用户态，最后得到服务的结果。
 2. 定义并实现系统调用管理类,创建并调用系统调用函数.
 3. 创建系统调用管理类,使用它调用系统调用函数.
 4. 借助gdb在关键函数处进行调试，并查看寄存器的值，进行前后对比。
- 实验步骤：
 1. 复现仓库里的代码
 2. 在 `asm_utils.asm` 的第91行的语句处 `int 0x80` 打上断点，然后跳转执行，查看寄存器的值。

```
b asm_utils.asm:146
c
info registers esp ss cs ds es fs gs eflags eip
```

3. 然后单步执行，进入 `asm_system_call_handler`，然后查看当前寄存器的值以及 `esp` 寄存器中的值。

```
si
info registers esp ss cs ds es fs gs eflags eip
x/20x $esp
```

4. 在 `asm_system_call_handler` 的 `iret` 语句处打上断点，然后单步执行，查看当前寄存器的值。

```
b asm_utils.asm:128
c
info registers esp ss cs ds es fs gs eflags eip
ni
info registers esp ss cs ds es fs gs eflags eip
```

• 实验结果及问题回答：

1. **为什么需要使用寄存器来传递系统调用的参数：**这是因为用户程序使用系统调用时会进行特权级转移。如果我们使用栈来传递参数，在我们调用系统调用的时候，系统调用的参数（即 `asm_system_call` 的参数）就会被保存在用户程序的栈中，也就是低特权级的栈中。系统调用发生后，我们从低特权级转移到高特权级，此时CPU会从TSS中加载高特权级的栈地址到`esp`寄存器中。而C语言的代码在编译后会使用`esp`和`ebp`来访问栈的参数，但是前面保存参数的栈和现在期望取出函数参数而访问的栈并不是同一个栈，因此CPU无法在栈中找到函数的参数。为了解决这个问题，我们通过寄存器来传递系统调用的参数。

如何在执行 `int 0x80` 前在栈中找到参数并放入寄存器的：通过 `mov` 指令把参数从栈上取出，放到指定寄存器，然后执行 `int 0x80`，完成参数传递。

2. 执行 `int 0x80` 前，`tss` 中的寄存器值和当前寄存器的值如下：

```
(gdb) p tss
$1 = {backlink = 0, esp0 = -1073588192, ss0 = 16, esp1 = 0, ss1 = 0, esp2 = 0, ss2 = 0, cr3 = 0, eip = 0, eflags = 0,
      eax = 0, ecx = 0, edx = 0, ebx = 0, esp = 0, ebp = 0, esi = 0, edi = 0, es = 0, cs = 0, ss = 0, ds = 0, fs = 0,
      gs = 0, ldt = 0, trace = 0, ioMap = -1073529556}
(gdb) info registers esp ss cs ds es fs gs eflags eip
esp      0x8048fb8      0x8048fb8
ss       0x3b         59
cs       0x2b         43
ds       0x33         51
es       0x33         51
fs       0x33         51
gs       0x0          0
eflags   0x212        [ IOPL=0 IF AF ]
eip      0xc002284d    0xc002284d <asm_system_call+26>
```

执行 `int 0x80` 之后，当前寄存器的值变为：

```
(gdb) b asm_system_call_handler
Breakpoint 2 at 0xc00227f7: file ../src/utils/asm_utils.asm, line 88.
(gdb) c
Continuing.

Breakpoint 2, asm_system_call_handler () at ../src/utils/asm_utils.asm:88
(gdb) info registers esp ss cs ds es fs gs eflags eip
esp      0xc002580c    0xc002580c <PCB_SET+8172>
ss       0x10         16
cs       0x20         32
ds       0x33         51
es       0x33         51
fs       0x33         51
gs       0x0          0
eflags   0x12        [ IOPL=0 AF ]
eip      0xc00227f7    0xc00227f7 <asm_system_call_handler>
(gdb) x/20x $esp
0xc002580c <PCB_SET+8172>: 0xc002284f 0x0000002b 0x00000212 0x08048fb8
0xc002581c <PCB_SET+8188>: 0x0000003b 0xc00267c0 0xfb1e0ff3 0x83e58955
0xc002582c <PCB_SET+8204>: 0xec8308ec 0x000006a08 0x00000000 0x00000002
0xc002583c <PCB_SET+8220>: 0x00000001 0x00000002 0x0000000a 0x00000000
0xc002584c <PCB_SET+8236>: 0xc0033c68 0xc002684c 0xc0024854 0xc0026854
(gdb)
```

结论：可以观察到执行 `int 0x80` 后

- **esp和ss 变化**: 执行 `int 0x80` 时, CPU 检查当前特权级 CPL (用户态为3, 内核为0), 发生特权级切换。CPU 自动将用户态 esp/ss、eflags、cs、eip 压入**内核栈**, 并将 esp/ss 切换为 tss.esp0/tss.ss0 指定的内核栈。
- **tss 作用**: tss 结构体中的 esp0 字段保存内核栈顶地址, `int 0x80` 时 esp 切换为 tss.esp0。
- **其他寄存器的变化**: cs由43变化为32, 这个32是IDT 表中 0x80 号中断对应的门描述符中的段选择子决定的; eflags变化为0x12, eip变化为0xc00227f7, 这是内核态对应的寄存器的值。

3. 进入 `asm_system_call_handler` 的那一刻, 栈顶的地址为 `0xc002580c`, 按照理论这个应该是 tss.esp0, 是指向内核栈的。但是为什么和上述 esp0 的值 `-1073588192` 换算为十六进制为 `0xc0025800` 不同呢? 这是因为在进入 `asm_system_call_handler` 时, 自动压入了一些寄存器的值, 所以地址会有所相差。

栈中存放的内容从栈顶开始是用户态的 ss, esp, eflags, cs, eip。

```
(gdb) b asm_system_call_handler
Breakpoint 2 at 0xc00227f7: file ../src/utils/asm_utils.asm, line 88.
(gdb) c
Continuing.

Breakpoint 2, asm_system_call_handler () at ../src/utils/asm_utils.asm:88
(gdb) info registers esp ss cs ds es fs gs eflags eip
esp             0xc002580c             0xc002580c <PCB_SET+8172>
ss              0x10                  16
cs              0x20                  32
ds              0x33                  51
es              0x33                  51
fs              0x33                  51
gs              0x0                   0
eflags          0x12                  [ IOPL=0 AF ]
eip             0xc00227f7             0xc00227f7 <asm_system_call_handler>
(gdb) x/20x $esp
0xc002580c <PCB_SET+8172>: 0xc002284f 0x0000002b 0x00000212 0x08048fb8
0xc002581c <PCB_SET+8188>: 0x0000003b 0xc00267c0 0xfb1e0ff3 0x83e58955
0xc002582c <PCB_SET+8204>: 0xec8308ec 0x000006a08 0x00000000 0x00000002
0xc002583c <PCB_SET+8220>: 0x00000001 0x00000002 0x0000000a 0x00000000
0xc002584c <PCB_SET+8236>: 0xc0033c68 0xc002684c 0xc0024854 0xc0026854
(gdb)
```

4. `asm_system_call_handler` 如何找到系统调用号对应的函数:

`asm_system_call_handler` 先保存现场, 然后将 `eax` 作为系统调用号。通过 `call dword[system_call_table + eax * 4]`, 查找系统调用表中对应的函数指针并调用。

5. `asm_system_call_handler` 中执行 `iret` 后, 发现 cs、eip、ss、esp、eflags 都会恢复为用户态的值, 这些值来自于 `int 0x80` 时 CPU 自动压入内核栈的内容。

```

(gdb) si
(gdb) info registers esp ss cs ds es fs gs eflags eip
esp      0xc002580c      0xc002580c <PCB_SET+8172>
ss       0x10          16
cs       0x20          32
ds       0x33          51
es       0x33          51
fs       0x33          51
gs       0x0           0
eflags   0x82          [ IOPL=0 SF ]
eip      0xc0022832      0xc0022832 <asm_system_call_handler+59>
(gdb) si
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) info registers esp ss cs ds es fs gs eflags eip
esp      0x8048fb8      0x8048fb8
ss       0x3b          59
cs       0x2b          43
ds       0x33          51
es       0x33          51
fs       0x33          51
gs       0x0           0
eflags   0x212         [ IOPL=0 IF AF ]
eip      0xc002284f      0xc002284f <asm_system_call+28>
(gdb)

```

实验任务2：进程的创建与调度

代码放在 lab8/assignment2 中

- 实验要求：复现“进程的实现”，“进程的调度”，“第一个进程”三节，并回答以下问题：
 - 请结合代码分析我们是如何在线程的基础上创建进程的PCB的(即分析进程创建的三个步骤)
 - 在进程的PCB第一次被调度执行时，进程实际上并不是跳转到进程的第一条指令处，而是跳转到 `load process`。请结合代码逻辑和gdb来分析为什么 `asm_switch_thread` 在执行 `iret` 后会跳转到 `load process`。
 - 在跳转到 `load process` 后，我们巧妙地设置了 `ProcessStartStack` 的内容，然后在 `asm_start_process` 中跳转到进程第一条指令处执行。请结合代码逻辑和gdb来分析我们是如何设置 `ProcessStartStack` 的内容，从而使得我们能够在 `asm_start_process` 中实现内核态到用户态的转移，即从特权级0转移到特权级3下，并使用 `iret` 指令成功启动进程的。
 - 结合代码，分析在创建进程后，我们对 `ProgramManager::schedule` 作了哪些修改? 这样做的目的是什么?
 - 在进程的创建过程中，我们存在如下语句：

```

int ProgramManager::executeProcess(const char *filename, int priority)
{
    ...
    //找到刚刚创建的PCB
    PCB *process = ListItem2PCB(allPrograms.back(), tagInAllList);
    ...
}

```

正如教程中所提到，“.....但是，这样做是存在风险的，我们应该通过pid来找到刚刚创建的PCB。.....”。现在，同学们需要编写一个 `ProgramManager` 的成员函数 `findProgramByPid`：

```
PCB *findProgramByPid(int pid);
```

并用上面这个函数替换指导书中提到的"存在风险的语句", 替换结果如下:

```
int ProgramManager::executeProcess(const char *filename, int priority)
{
    ...
    //找到刚刚创建的PCB
    PCB *process =findProgramByPid(pid);
    ...
}
```

自行测试通过后, 说一说你的实现思路, 并保存结果截图。

- 实验思路: 结合代码和gdb分析, 通过查看关键语句执行前后, 段寄存器值的变化来分析进程的调度与实现原理。
- 实验步骤:

1. 复现仓库里的代码。

2. 在 `asm_switch_thread` 的 `iret` 打下断点, 然后跳转到这里, 查看 `esp` 栈中存的内容。单步执行之后, 再查看 `eip` 的地址。

```
b asm_switch_thread:iret
c
x/20x $esp
si
info registers eip
```

3. 在 `load_process` 的 `asm_start_process((int)interruptStack);` 打上断点, 跳转到这里, 查看 `ProcessStartStack` 和 `esp` 中的内容。再单步执行, 查看 `esp` 中的值。

```
b program.cpp:328
c
set $pcb = programManager.running
set $pss = (struct ProcessStartStack *)((int)$pcb + 4096 - sizeof(struct ProcessStartStack))
p *$pss
x/20x $esp
```

4. 然后执行到 `asm_start_process` 的 `iret` 语句, 查看前后的寄存器的值。

```
x/20x $esp
info registers esp ss cs ds es fs gs eflags eip
```

5. 编写 `findProgramByPid` 函数:

思路:

- 通过遍历 `allPrograms` 链表, 逐个比较 `pid`, 确保找到的 `PCB` 一定是刚刚分配的那个。
- 这样避免了 `allPrograms.back()` 可能因为并发或插入顺序导致的错误。

```
PCB *ProgramManager::findProgramByPid(int pid){
    if (pid < 0 || pid >= MAX_PROGRAM_AMOUNT)
    {
        return nullptr;
    }

    PCB *program = ListItem2PCB(allPrograms.front(), tagInAllList);
    while (program)
    {
        if (program->pid == pid)
        {
            return program;
        }
        program = ListItem2PCB(program->tagInAllList.next, tagInAllList);
    }

    return nullptr;
}
```

实验结果及问题回答：

1. 进程的PCB创建过程分为三步：

- 用线程方式分配和初始化PCB（基础结构，设置入口为 load_process）。
- 查找刚刚创建的PCB，确保后续操作针对正确的进程。
- 为PCB分配进程专属资源（页目录表、用户虚拟地址池），使其成为真正的“进程PCB”。

2. 为什么 asm_switch_thread 在执行 iret 后会跳转到 load_process：

可以看到在 ProgramManager::executeThread 里：

```
thread->stack = (int *)((int)thread + PCB_SIZE - sizeof(ProcessStartStack));
thread->stack -= 7;
thread->stack[0] = 0;
thread->stack[1] = 0;
thread->stack[2] = 0;
thread->stack[3] = 0;
thread->stack[4] = (int)function; // 这里 function 是 load_process
thread->stack[5] = (int)program_exit;
thread->stack[6] = (int)parameter;
```

- 这里 function 对于进程来说就是 load_process。
- 这些值模拟了线程/进程切换时的栈帧，让切换回来时能“假装”是从 function 开始执行。

所以这是因为进程的PCB在第一次被调度时，其内核栈内容是人为构造的，返回地址就是 load_process。

在实验结果中，可以看到在执行 iret 指令前，esp的栈顶就已经存入了 load_process 的地址 0xc002085f，然后单步执行之后，可以看到跳转到了 load_process，eip的地址也为 0xc002085f


```

(gdb) info registers esp
esp                0xc00257d0                0xc00257d0 <PCB_SET+8112>
(gdb) x/20x $esp
0xc00257d0 <PCB_SET+8112>:  0xc002085f    0xc002042c    0xc0020aa4    0x00000000
0xc00257e0 <PCB_SET+8128>:  0x00000000    0x00000000    0x00000000    0x00000000
0xc00257f0 <PCB_SET+8144>:  0x00000000    0x00000000    0x00000000    0x00000000
0xc0025800 <PCB_SET+8160>:  0x00000000    0x00000000    0x00000000    0x00000000
0xc0025810 <PCB_SET+8176>:  0x00000000    0x00000000    0x00000000    0x00000000

(gdb) si
load_process (filename=0xc0020aa4 <first_process(> "\363\017\036\373U\211\345\203\354\b\203\354\bj")
at ../src/kernel/program.cpp:290
(gdb) info registers esp
esp                0xc00257d4                0xc00257d4 <PCB_SET+8116>
(gdb) x/20x $esp
0xc00257d4 <PCB_SET+8116>:  0xc002042c    0xc0020aa4    0x00000000    0x00000000
0xc00257e4 <PCB_SET+8132>:  0x00000000    0x00000000    0x00000000    0x00000000
0xc00257f4 <PCB_SET+8148>:  0x00000000    0x00000000    0x00000000    0x00000000
0xc0025804 <PCB_SET+8164>:  0x00000000    0x00000000    0x00000000    0x00000000
0xc0025814 <PCB_SET+8180>:  0x00000000    0x00000000    0x00000000    0xc00267c0
(gdb) info registers eip
eip                0xc002085f                0xc002085f <load_process(char const*)>

```

3. 我们是如何设置 ProcessStartStack 的内容，从而使得我们能够在 asm_start_process 中实现内核态到用户态的转移，即从特权级0转移到特权级3下，并使用 iret 指令成功启动进程的：

根据 load_process 代码可以看到 load_process 构造了一个“伪中断返回栈帧”，内容完全模拟了从内核态返回到用户态的情形。

```

void load_process(const char *filename)
{
    ...
    PCB *process = programManager.running;
    ProcessStartStack *interruptStack =
        (ProcessStartStack *)((int)process + PAGE_SIZE - sizeof(ProcessStartStack));

    // 通用寄存器和段寄存器
    interruptStack->edi = 0;
    interruptStack->esi = 0;
    interruptStack->ebp = 0;
    interruptStack->esp_dummy = 0;
    interruptStack->ebx = 0;
    interruptStack->edx = 0;
    interruptStack->ecx = 0;
    interruptStack->eax = 0;
    interruptStack->gs = 0;
    interruptStack->fs = programManager.USER_DATA_SELECTOR;
    interruptStack->es = programManager.USER_DATA_SELECTOR;
    interruptStack->ds = programManager.USER_DATA_SELECTOR;

    // 关键：设置用户态的 eip、cs、eflags、esp、ss
    interruptStack->eip = (int)filename; // 用户进程入口
    interruptStack->cs = programManager.USER_CODE_SELECTOR; // DPL=3
    interruptStack->eflags = (0 << 12) | (1 << 9) | (1 << 1); // IOPL=0, IF=1, MBS=1

    interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1);
    interruptStack->esp += PAGE_SIZE;
    interruptStack->ss = programManager.USER_STACK_SELECTOR; // DPL=3
}

```



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

然后 `asm_start_process` 通过切换 `esp` 并执行 `iret`，让 CPU 自动完成从内核态到用户态的跳转，用户进程从入口开始执行，拥有自己的用户栈和段寄存器。

实验结果如下：在执行 `iret` 之前，`ss` 寄存器中的 `RPL=0`，`cs` 寄存器中的 `CPL=0`，说明现在还在内核态，执行之后，可以观察到 `ss` 寄存器的 `RPL=3`，`cs` 寄存器中的 `CPL=3`，证明 CPU 成功从内核态跳转到用户态。

```
(gdb) x/20x $esp
0xc002588c <PCB_SET+8172>: 0xc0020b02 0x0000002b 0x00000202 0x08049000
0xc002589c <PCB_SET+8188>: 0x0000003b 0xc0026840 0xfb1e0ff3 0x83e58955
0xc00258ac <PCB_SET+8204>: 0xec8308ec 0x000006a08 0x00000000 0x00000002
0xc00258bc <PCB_SET+8220>: 0x00000001 0x00000002 0x0000000a 0x00000000
0xc00258cc <PCB_SET+8236>: 0xc0033ce8 0xc00268cc 0xc00248d4 0xc00268d4

(gdb) info registers esp ss cs ds es fs gs eflags eip
esp      0xc002588c      0xc002588c <PCB_SET+8172>
ss       0x10           16
cs       0x20           32
ds       0x33           51
es       0x33           51
fs       0x33           51
gs       0x0            0
eflags   0x92           [ IOPL=0 SF AF ]
eip      0xc002280d      0xc002280d <asm_start_process+13>
```

```
(gdb) info registers esp ss cs ds es fs gs eflags eip
esp      0x8049000      0x8049000
ss       0x3b           59
cs       0x2b           43
ds       0x33           51
es       0x33           51
fs       0x33           51
gs       0x0            0
eflags   0x202         [ IOPL=0 IF ]
eip      0xc0020b02      0xc0020b02 <first_process(>)
(gdb)
```

4. 对 `ProgramManager::schedule` 作了哪些修改? 这样做的目的是什么?

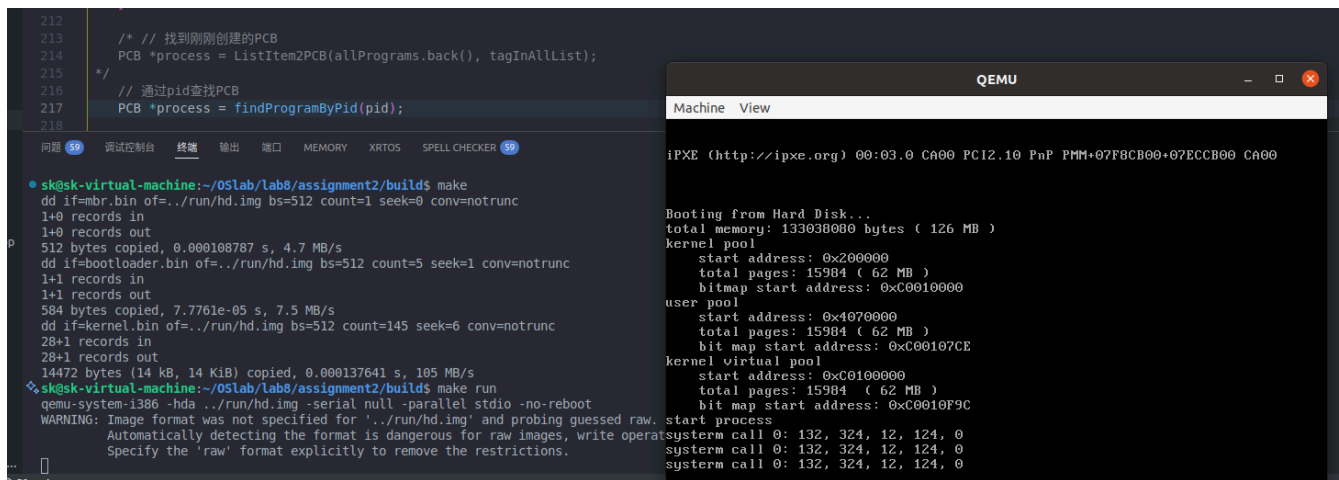
主要增加了页表切换 `activateProgramPage(next);`

- 在切换到下一个进程（或线程）之前，调用 `activateProgramPage(next)`。
- 该函数会把 TSS 的 `esp0` 设置为新进程的内核栈顶，并用 `asm_update_cr3` 切换到新进程的页目录表（虚拟地址空间）。

目的是：

- 实现进程的地址空间隔离。** 每个进程有独立的页表，切换页表后，CPU 访问的虚拟地址空间就变成了新进程的空间，保证了进程间的内存隔离和安全。
- 为后续的内核态到用户态切换做准备。** 只有先切换到正确的页表，`asm_start_process` 和 `iret` 才能让用户进程在自己的空间中正常运行。

5. 成功实现 `findProgramByPid` 函数，并且测试通过。



实验任务3：fork的实现

代码放在 lab8/assignment3 中

- 实验要求：复现“fork”一小节的内容，并回答以下问题：
 - 请根据代码逻辑概括 fork 的实现的基本思路，并简要分析我们是如何解决“四个关键问题”的。
 - 请根据gdb来分析子进程第一次被调度执行时，即在 `asm_switch_thread` 切换到子进程的栈中时，esp 的地址是什么？栈中保存的内容是什么？
 - 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来解释子进程的 fork 返回值为什么是0，而父进程的 fork 返回值是子进程的pid。
 - 请解释在 `Programanager::schedule` 中，我们是如何从一个进程的虚拟地址空间切换到另外一个进程的虚拟地址空间的。
- 实验思路：复现仓库代码，然后使用gdb从fork调用开始，逐步进行调试，观察寄存器的值，分析整个fork调用的流程。
- 实验步骤：
 - 复现仓库代码。
 - 在 `asm_switch_thread` 打上断点，然后逐步执行，观察esp的地址和esp中保存的内容。

```
b asm_switch_thread
c
si...
info registers esp
x/20x $esp
```

- 从 `asm_switch_thread` 开始，逐步执行，观察整个子进程的执行流程。

实验结果和问题回答：

- 概括 fork 的实现的基本思路，并简要分析我们是如何解决“四个关键问题”的：

基本思路：

- 父进程调用 `fork`，通过系统调用陷入内核，实际调用 `ProgramManager::fork()`。

- **创建子进程 PCB:** `ProgramManager::fork()` 内部调用 `executeProcess`, 分配新的 PCB、页目录、虚拟地址池等资源。
- **复制父进程资源到子进程:** 调用 `copyProcess`, 复制父进程的内核栈、虚拟地址空间 (页表和物理页)、虚拟地址池等, 使子进程拥有与父进程一致的执行上下文。
- **设置返回值:** 父进程 `fork` 返回子进程 `pid`, 子进程 `fork` 返回 0。
- **调度器调度:** 子进程被加入就绪队列, 等待调度, 首次调度时通过切换内核栈和上下文, 恢复到 `fork` 后的用户态继续执行。

解决问题:

1. 如何实现父子进程的代码段共享: 我们使用了函数来模拟一个进程, 而函数的代码是放在内核中的, 进程又划分了 3GB~4GB 的空间来实现内核共享, 因此进程的代码天然就是共享的。
 2. 如何使得父子进程从相同的返回点开始执行: 在 `fork` 时, `copyProcess` 会复制父进程的 0 级栈 (内核栈) 到子进程, 并设置子进程的 `eax` (`fork` 的返回值) 为 0。这样, 子进程被调度时, 恢复的上下文和父进程 `fork` 返回时完全一致, 只是 `eax` 不同, 因此父子进程会从 `fork` 返回点继续执行, 逻辑一致。
 3. 除代码段外, 进程包含的资源有哪些: 在我们的操作系统中, 进程包含的资源有 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。
 4. 如何实现进程的资源在进程之间的复制: 首先在父进程的虚拟地址空间下将数据复制到中转页中, 再切换到子进程的虚拟地址空间中, 然后将中转页复制到子进程对应的位置。
2. 在 `asm_switch_thread` 切换到子进程的栈中时, `esp` 的地址是什么? 栈中保存的内容是什么?

`esp` 的地址为 `0xc0026f20`, 这个地址是子进程的 PCB 的内核栈顶, 也就是 `child->stack 0xc0026f20`。栈中保存的内容 (从栈顶开始) 是如下代码中的内容

```
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
```

观察到 `esp` 中的内容确实符合代码中的内容, 说明成功切换到子进程。

```
ProgramManager::copyProcess (this=0xc00343c0 <programManager>, parent=0xc0024f80 <PCB_SET+4096>,
child=0xc0025f80 <PCB_SET+8192>) at ../src/kernel/program.cpp:380
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) n
(gdb) p childpss
$1 = (ProcessStartStack *) 0xc0026f3c <PCB_SET+12220>
(gdb) p child
$2 = (PCB *) 0xc0025f80 <PCB_SET+8192>
(gdb) p child->stack
$3 = (int *) 0xc0026f20 <PCB_SET+12192>
(gdb)
```

```
(gdb) n
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:205
(gdb) info registers esp
esp             0xc0026f20             0xc0026f20 <PCB_SET+12192>
(gdb) x/20x $esp
0xc0026f20 <PCB_SET+12192>:  0x00000000      0x00000000      0x00000000      0x00000000
0xc0026f30 <PCB_SET+12208>:  0xc0022dc0      0x00000000      0xc0026f3c      0x00000000
0xc0026f40 <PCB_SET+12224>:  0x00000000      0x08048fac      0xc0025f5c      0x00000000
0xc0026f50 <PCB_SET+12240>:  0x00000000      0x00000000      0x00000000      0x00000000
0xc0026f60 <PCB_SET+12256>:  0x00000033      0x00000033      0x00000033      0xc0022e6f
(gdb)
```

3. 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，比较上述过程和父进程执行完 `ProgramManager::fork` 后的返回过程的异同。

- 将子进程调度上CPU执行：

首先进入 `asm_switch_thread`，观察子进程如何被调度上CPU执行。可以看到在 `schedule` 调用 `asm_switch_thread` 前，子进程next地址为 `0xc0025f80`，子进程栈next->stack地址为 `0xc0026f20`。进入 `asm_switch_thread` 后，可以看到esp指向了子进程栈，说明子进程已经调度执行了。

```
(gdb) p next
$5 = (PCB *) 0xc0025f80 <PCB_SET+8192>
(gdb) p next->stack
$6 = (int *) 0xc0026f20 <PCB_SET+12192>
(gdb) p cur
$7 = (PCB *) 0xc0023f80 <PCB_SET>
(gdb) p cur->stack
$8 = (int *) 0xc0024e74 <PCB_SET+3828>
(gdb)
```

```
(gdb) n
asm_switch_thread () at ../src/utils/asm_utils.asm:205
(gdb) info registers
eax             0xc0025f80             -1073586304
ecx             0x1                    1
edx             0x219000               2199552
ebx             0x0                    0
esp             0xc0026f20             0xc0026f20 <PCB_SET+12192>
ebp             0xc0024eb0             0xc0024eb0 <PCB_SET+3888>
esi             0x0                    0
edi             0x0                    0
eip             0xc0022eb4             0xc0022eb4 <asm_switch_thread+16>
eflags         0x86                    [ IOPL=0 SF PF ]
cs              0x20                   32
ss              0x10                   16
ds              0x8                    8
es              0x8                    8
fs              0x0                    0
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             0x219000                [ PDBR=0 PCID=0 ]
cr4             0x0                    [ ]
cr8             0x0                    0
efer            0x0                    [ ]
```

- 逐步执行，跳转到 `asm_start_process`，子进程开始执行。

观察到 `ds, fs, es, gs` 寄存器被切换到内核线程使用的 `ds, fs, es, gs`，说明子进程开始执行。

```

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

```

```

asm_switch_thread () at ../src/utils/asm_utils.asm:211
(gdb) s
asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb) info registers
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
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             0x219000            [ PDBR=0 PCID=0 ]
cr4             0x0                 [ ]
cr8             0x0                 0
efer            0x0                 [ ]

```

```
(gdb) s
asm_start_process () at ../src/utils/asm_utils.asm:49
(gdb) info registers
eax                0x0                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 ]
cr4                0x0                [ ]
cr8                0x0                0
efer               0x0                [ ]
```

- 然后跳转到 `asm_system_call`，子进程fork系统调用返回。

可以观察到cs和ss发生变化，ss寄存器的RPL变为3，cs寄存器中的CPL变为3，证明CPU成功从内核态跳转到用户态，系统调用返回，eax为0，说明子进程返回值为0。


```
fixes: 0x1f80 [ 1f 0f 2f 0f 0f 1f ]
(gdb) s
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) info registers
eax            0x0            0
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            0x219000       [ PDBR=0 PCID=0 ]
cr4            0x0            [ ]
cr8            0x0            0
efer           0x0            [ ]
```

父子进程 fork 返回的异同:

父进程返回情况

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

子进程返回情况

```
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
```

项目	父进程 fork 返回	子进程 fork 返回（第一次被调度）
返回值 eax	子进程 pid	0
eip	fork 调用后	fork 调用后（同一位置）
数据寄存器	保持原值	恢复自父进程（memcpy），除了 eax=0
段寄存器	保持原值	恢复自父进程（memcpy）
跳转方式	普通函数返回	进程切换+iret 跳转到用户态
栈	没有切换	esp 切换到子进程内核栈，再切到用户栈

4. 根据代码逻辑和gdb来解释子进程的 fork 返回值为什么是0，而父进程的 fork 返回值是子进程的pid。

可以看到父进程在调用fork时，执行 `int pid = executeProcess("", 0);` 创建子进程，这个 pid 会通过系统调用返回到父进程的 eax 寄存器。所以父进程返回值为子进程pid=2。

```
eax      0x2      2
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xc0025f04 0xc0025f04 <PCB_SET+8068>
ebp      0xc0025f20 0xc0025f20 <PCB_SET+8096>
esi      0x0      0
edi      0x0      0
eip      0xc00209f0 0xc00209f0 <ProgramManager::fork()>
eflags   0x296    [ IOPL=0 IF SF AF PF ]
cs       0x20     32
ss       0x10     16
ds       0x8      8
es       0x8      8
fs       0x33     51
gs       0x18     24
```

```
(gdb) n
(gdb) p pid
$1 = 2
```

```
374
375 interruptManager.setInterruptStatus(status);
>376 return pid;
```

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

而子进程并不是直接执行 `fork()`，而是被调度后，通过内核栈和上下文恢复机制，从 `fork` 调用点“返回”。

- 关键代码在 `copyProcess`

```
// 复制父进程的0级栈
memcpy(parentpss, childpss, sizeof(ProcessStartStack));
// 设置子进程的返回值为0
childpss->eax = 0;
```

- 也就是说，子进程的 `eax` 被强制设置为 0，其余寄存器内容和父进程 `fork` 时一致。当子进程第一次被调度运行时，通过 `asm_start_process` 和 `iret`，恢复了 `ProcessStartStack`，此时 `eax=0`，`eip` 指向 `fork` 返回点。
- 所以子进程 `fork` 返回的是 0。

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

5. 请解释在 `Programanager::schedule` 中，我们是如何从一个进程的虚拟地址空间切换到另外一个进程的虚拟地址空间的。

在 `schedule` 中通过调用 `activateProgramPage(next)`

```

void ProgramManager::activateProgramPage(PCB *program)
{
    int paddr = PAGE_DIRECTORY;

    if (program->pageDirectoryAddress) //该进程的页目录表虚拟地址
    {
        tss.esp0 = (int)program + PAGE_SIZE;
        paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress); //得到其物理地址
    }

    asm_update_cr3(paddr); //将物理地址写入cr3寄存器
}

```

再调用 `asm_update_cr3(paddr)` 将下一个进程的页目录物理地址写入 `cr3` 寄存器，这样 CPU 的内存访问就会按照新进程的页表进行地址转换，实现了虚拟地址空间的切换。

实验任务4：wait & exit 的实现

代码放在 `lab8/assignment4` 中

- 实验要求：参考指导书中“wait”和“exit”两节的内容，实现 `wait` 函数和 `exit` 函数，回答以下问题：
 - 请结合代码逻辑和具体的实例来分析 `exit` 的执行过程。
 - 请解释进程退出后能够隐式调用 `exit` 的原因。(tips:从栈的角度分析)
 - 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。

- 实验思路：

`exit`的实现主要分为三步：

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

`wait`根据定义实现：

- `wait`的参数`retval`用来存放子进程的返回值，如果`retval==nullptr`，则说明父进程不关心子进程的返回值。

- `wait`的返回值是被回收的子进程的pid。如果没有子进程，则`wait`返回-1。
- 在父进程调用了`wait`后，如果存在子进程但子进程的状态不是`DEAD`，则父进程会被阻塞，即`wait`不会返回直到子进程结束。

- 实验步骤：

- 复现仓库代码。
- 结合代码分析`exit`的执行过程：

- 用户进程调用 `exit(0)`：

用户进程代码中直接调用 `exit(0)`，实际调用的是 `syscall.cpp` 里的：

```
void exit(int *ret*) {  
    asm_system_call(3, ret);  
}
```

触发 `int 0x80`，即软中断，进入内核态。

```
>43      void exit(int ret) {  
44          asm_system_call(3, ret);  
45      }  
46  
145  
>146      int 0x80  
147  
148          pop edi  
149          pop esi  
150          pop edx  
151          pop ecx  
152          pop ebx  
153          pop ebp  
154
```

■ 系统调用中断处理:

在 `asm_utils.asm` 的 `asm_system_call_handler` 里，内核会根据 `eax` 的值（此处为 3）查找系统调用表，调用 `syscall_exit`：

```
void syscall_exit(int *ret*) {  
    programManager.exit(ret);  
}
```

```
(gdb) s  
asm_system_call_handler () at ../src/utils/asm_utils.asm:108  
(gdb) info registers eax  
eax          0x3          3  
(gdb) s  
47      void syscall_exit(int ret) {  
>48          programManager.exit(ret);  
49      }
```

■ 内核态执行 `ProgramManager::exit`

进入 `program.cpp` 的 `ProgramManager::exit(int ret)`，主要做三件事：

- 标记当前 PCB 状态为 DEAD，并保存返回值。

```
(gdb) n  
(gdb) p program->status  
$1 = DEAD  
(gdb) p program->retValue  
$2 = 0
```

- 如果是进程（有页目录），释放所有物理页、页表、页目录、虚拟地址池 bitmap。
- 调用 `schedule()` 进行调度，切换到下一个可运行线程/进程。

观察到esp已经指向下一个进程的栈了，说明切换到下一个可运行进程。

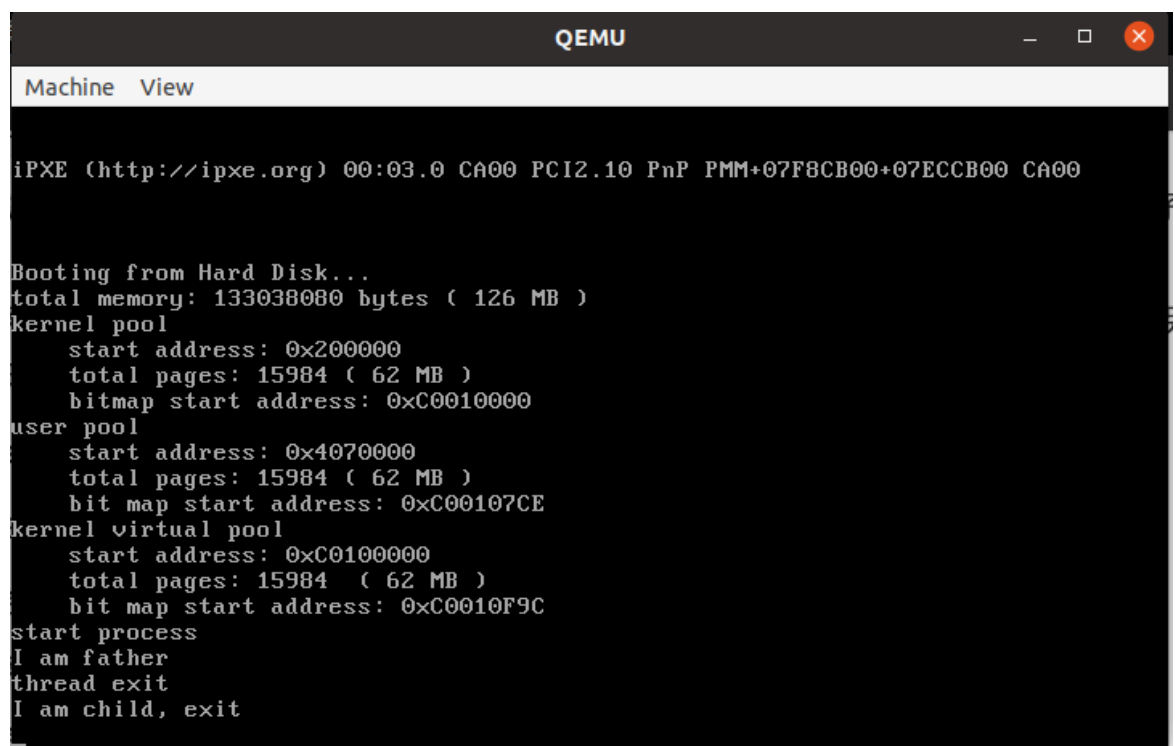
```
(gdb) n
(gdb) p next
$1 = (PCB *) 0xc0024260 <PCB_SET>
(gdb) p next->stack
$2 = (int *) 0xc0025154 <PCB_SET+3828>
(gdb) p cur
$3 = (PCB *) 0xc0026260 <PCB_SET+8192>
(gdb) p cur->stack
$4 = (int *) 0xc0027200 <PCB_SET+12192>

(gdb) s
asm_switch_thread () at ../src/utils/asm_utils.asm:205
(gdb) info registers esp
esp                0xc0025154                0xc0025154 <PCB_SET+3828>
(gdb) x/20x $esp
0xc0025154 <PCB_SET+3828>: 0x00000000 0x00000000 0x00000000 0xc0025190
0xc0025164 <PCB_SET+3844>: 0xc0020412 0xc0024260 0xc0025260 0xc0025190
0xc0025174 <PCB_SET+3860>: 0xc0021e33 0xc0024260 0xc0025260 0xc002528c
0xc0025184 <PCB_SET+3876>: 0x00021e7f 0xc0034684 0x00017000 0xc00251c0
0xc0025194 <PCB_SET+3892>: 0xc0021e1d 0xc00346a0 0xc00346a8 0x00000246
```

■ 调度与切换

`schedule()` 会选择下一个 READY 状态的 PCB，切换页表，调用 `asm_switch_thread` 完成上下文切换。

成功完成进程的exit退出。



```
QEMU
Machine View
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
thread exit
I am child, exit
```

3. 解释进程退出后能够隐式调用 `exit` 的原因：

主要是在 `load_process` 中，设置了用户栈的返回地址，这样，用户进程的栈顶（`esp`）指向一个“假的”返回地址，即 `exit` 的地址。


```
// 设置进程返回地址
int *userStack = (int *)interruptStack->esp;
userStack -= 3;
userStack[0] = (int)exit; // 返回地址为exit的地址
userStack[1] = 0;
userStack[2] = 0;
interruptStack->esp = (int)userStack;
interruptStack->ss = programManager.USER_STACK_SELECTOR;
```

当用户进程执行完毕后，执行 `ret` 指令。`ret` 会弹出栈顶的返回地址（即 `exit` 的地址），跳转到 `exit` 函数。于是即使用户代码没有显式调用 `exit`，也会自动进入 `exit`，完成资源回收和进程退出。

我将 `exit(0)` 注释掉，查看 `exit` 的调用。

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

根据下图可以发现线程的栈顶返回地址被设置为 `program_exit`，当 `second_thread` 执行完毕后，遇到 `ret`，会自动跳转到 `program_exit`，从而完成线程资源回收和调度。

```
(gdb) b second_thread(void*)
Breakpoint 1 at 0xc0021295: file ../src/kernel/setup.cpp, line 54.
(gdb) c
Continuing.

Breakpoint 1, second_thread (arg=0x0) at ../src/kernel/setup.cpp:54
(gdb) n
(gdb) n
(gdb) info registers esp
esp             0xc0027208             0xc0027208 <PCB_SET+12200>
(gdb) x/4xw 0xc0027208
0xc0027208 <PCB_SET+12200>:  0x00000000  0x00000000  0x00000000  0xc002042c
(gdb) info symbol 0xc002042c
program_exit() in section .text
(gdb)
```

4. 结合代码逻辑和具体的实例来分析 `wait` 的执行过程：

- 系统调用：

`wait` 实际调用 `asm_system_call(4, (int)retval)`。然后触发 `int 0x80`，进入内核态。由 `system_call_table[4]` 指向 `syscall_wait`，即 `programManager.wait(retval)`。

- `programManager.wait(retval)` 执行过程就是：

1. 关闭中断，遍历所有进程，查找自己的子进程。
2. 如果有子进程且已退出，回收资源，返回其 `pid` 和返回值。
3. 如果有子进程但都没退出，当前进程让出CPU，等待子进程退出，之后再次检查。
4. 如果没有子进程，返回 -1。

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

    while (true)
```

```

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

    item = this->allPrograms.head.next;

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

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

        int pid = child->pid;
        releasePCB(child);
        interruptManager.setInterruptStatus(interrupt);
        return pid;
    }
    else
    {
        if (flag) // 子进程已经返回
        {
            interruptManager.setInterruptStatus(interrupt);
            return -1;
        }
        else // 存在子进程，但子进程的状态不是DEAD
        {
            interruptManager.setInterruptStatus(interrupt);
            schedule();
        }
    }
}
}

```

根据 `first_process` 实例：

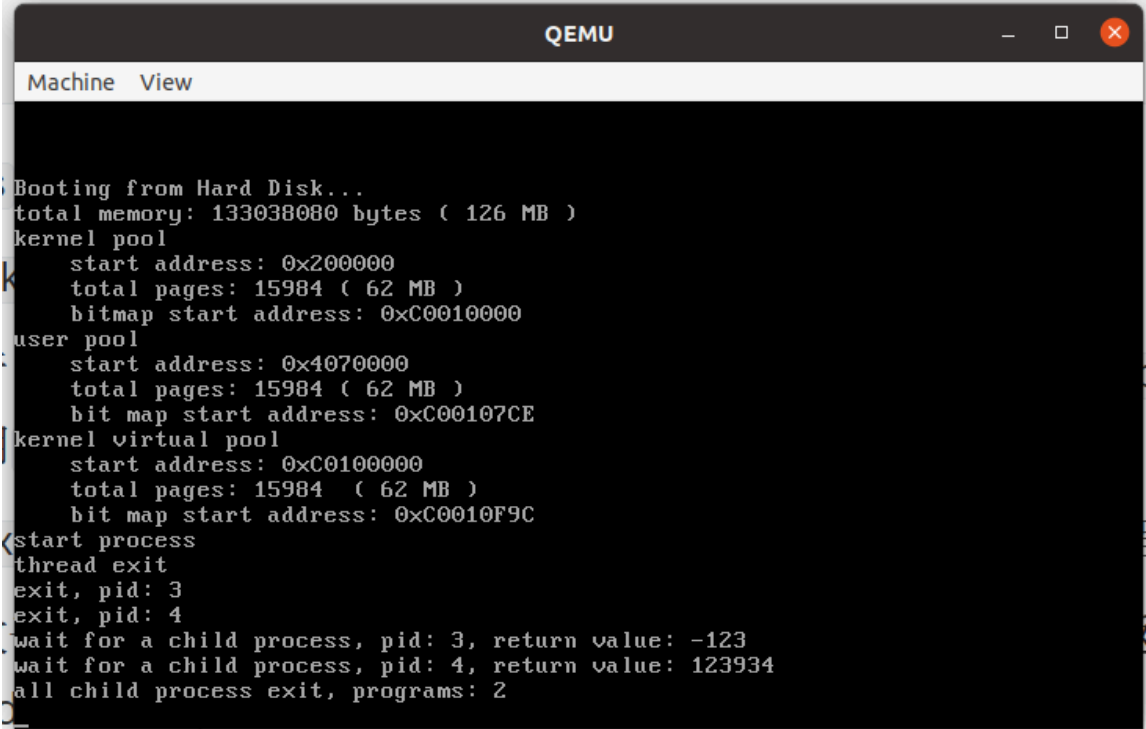
步骤1: 父进程A fork 两个子进程B和C, 然后B和C分别执行耗时操作后 exit。

步骤2: 父进程A执行 `wait(&retval)`。遍历所有PCB, 查找 `parentPid == A.pid` 的子进程 (B和C)。如果有子进程且状态不是DEAD, A调用 `schedule()` 让出CPU, 等待子进程退出。

步骤3: B或C执行 `exit(ret)`, 设置自身 `status = DEAD`, 并保存返回值。进程调度后, A被唤醒, 继续执行wait。

步骤4: 父进程A回收子进程: wait再次遍历PCB, 发现有子进程DEAD。取出其返回值, 调用 `releasePCB(child)` 释放PCB资源。返回子进程pid和返回值, A打印信息。

步骤5: 重复等待: A继续wait, 直到所有子进程都被回收。如果没有子进程了 (flag为true), wait返回-1, 循环结束。



```
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
thread exit
exit, pid: 3
exit, pid: 4
wait for a child process, pid: 3, return value: -123
wait for a child process, pid: 4, return value: 123934
all child process exit, programs: 2
```

5. 请对代码做出修改, 实现回收僵尸进程的有效方法:

首先创建一个僵尸进程, 修改 `setup.cpp`, 让父进程不调用wait来回收子进程的资源。

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

    if (pid)
    {
        pid = fork();
        if (pid)
        {
            //父进程直接退出
            printf("parent process exit, pid: %d\n", programManager.running->pid);
            exit(0);
        }
        else
        {
            uint32 tmp = 0xffffffff;
```

```

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

```

为了显示僵尸进程的存在，我添加了 `void ProgramManager::showZombieProcesses()` 函数显示僵尸进程和 `void monitor_zombie_thread(void **arg*)` 定时监控僵尸进程的线程。

```

// 显示所有僵尸进程信息的函数
void ProgramManager::showZombieProcesses()
{
    // 保存并关闭中断，保证遍历进程链表的原子性
    bool interrupt = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    printf("==== Process Status Monitor =====\n");

    ListItem *item = allPrograms.head.next;
    int zombieCount = 0;

    // 遍历所有进程，查找状态为DEAD的进程（即僵尸进程）
    while (item)
    {
        PCB *pcb = ListItem2PCB(item, tagInAllList);
        if (pcb->status == ProgramStatus::DEAD)
        {
            // 打印僵尸进程的PID、父进程PID和返回值
            printf("ZOMBIE: PID=%d, ParentPID=%d, RetValue=%d\n",
                pcb->pid, pcb->parentPid, pcb->retValue);
            zombieCount++;
        }
        item = item->next;
    }

    // 根据僵尸进程数量输出提示
    if (zombieCount == 0) {
        printf("No zombie processes found.\n");
    } else {
        printf("Total zombie processes: %d\n", zombieCount);
    }
}

```

```

printf("=====\n");

// 恢复中断状态
interruptManager.setInterruptStatus(interrupt);
}

```

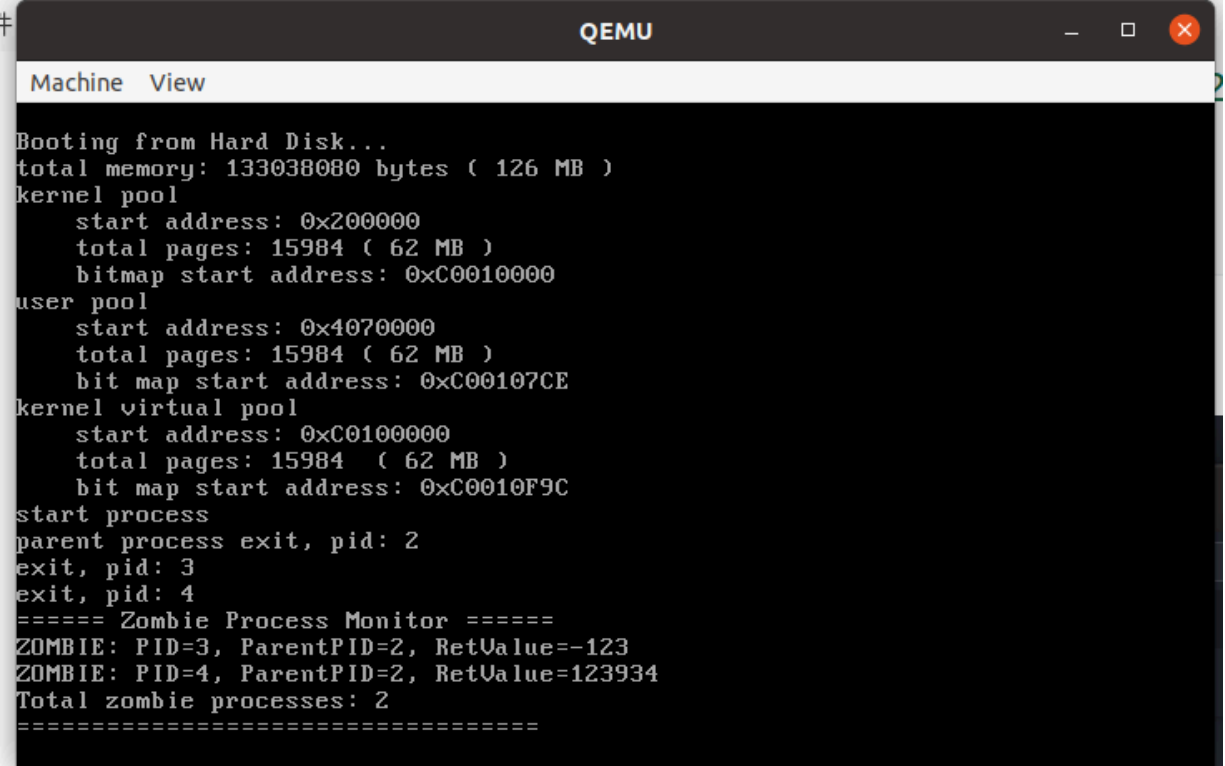
```

//添加一个监控僵尸进程的线程
void monitor_zombie_thread(void *arg)
{
    while(true)
    {
        //延迟
        uint tmp = 0xffffffff;
        while (tmp)
            --tmp;
        programManager.showZombieProcesses();
    }
}

```

僵尸进程结果如下：

两个子进程exit之后，成为了僵尸进程。



```

# 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
parent process exit, pid: 2
exit, pid: 3
exit, pid: 4
===== Zombie Process Monitor =====
ZOMBIE: PID=3, ParentPID=2, RetValue=-123
ZOMBIE: PID=4, ParentPID=2, RetValue=123934
Total zombie processes: 2
=====

```

解决僵尸进程的方法：

1. 利用实验指导书中的wait操作，回收僵尸进程资源

```
//在主进程中调用wait操作,
while ((pid = wait(&retval)) != -1)
{
    printf("wait for a child process, pid: %d, return value: %d\n",
pid, retval);
}
```

2. 设置一个init进程来回收僵尸进程:

在 `void ProgramManager::exit(int ret)` 系统调用中添加一个将子进程托管给init进程处理的逻辑。

```
ListItem *item = allPrograms.head.next;
while (item) {
    PCB *process = ListItem2PCB(item, tagInAllList);
    if (process->parentPid == program->pid) {
        printf("Process %d orphaned, assigned to init process\n", process-
>pid);
        process->parentPid = 1; // 1号进程为init进程
    }
    item = item->next;
}
```

然后在 `setup.cpp` 中添加init进程用来回收僵尸进程:

```
// init进程, 回收所有僵尸进程
void init_process(void *arg)
{
    printf("Init process started (PID=1), waiting for zombie processes\n");

    int retval;
    while (1)
    {
        // 尝试回收所有子进程
        int pid;
        while ((pid = wait(&retval)) != -1)
        {
            printf("Init: collected zombie process PID=%d (retval=%d)\n",
                pid, retval);
        }

        // 延时一段时间
        uint32 tmp = 0xffffffff;
        while (tmp) --tmp;
    }
}
```



```

void first_thread(void *arg)
{

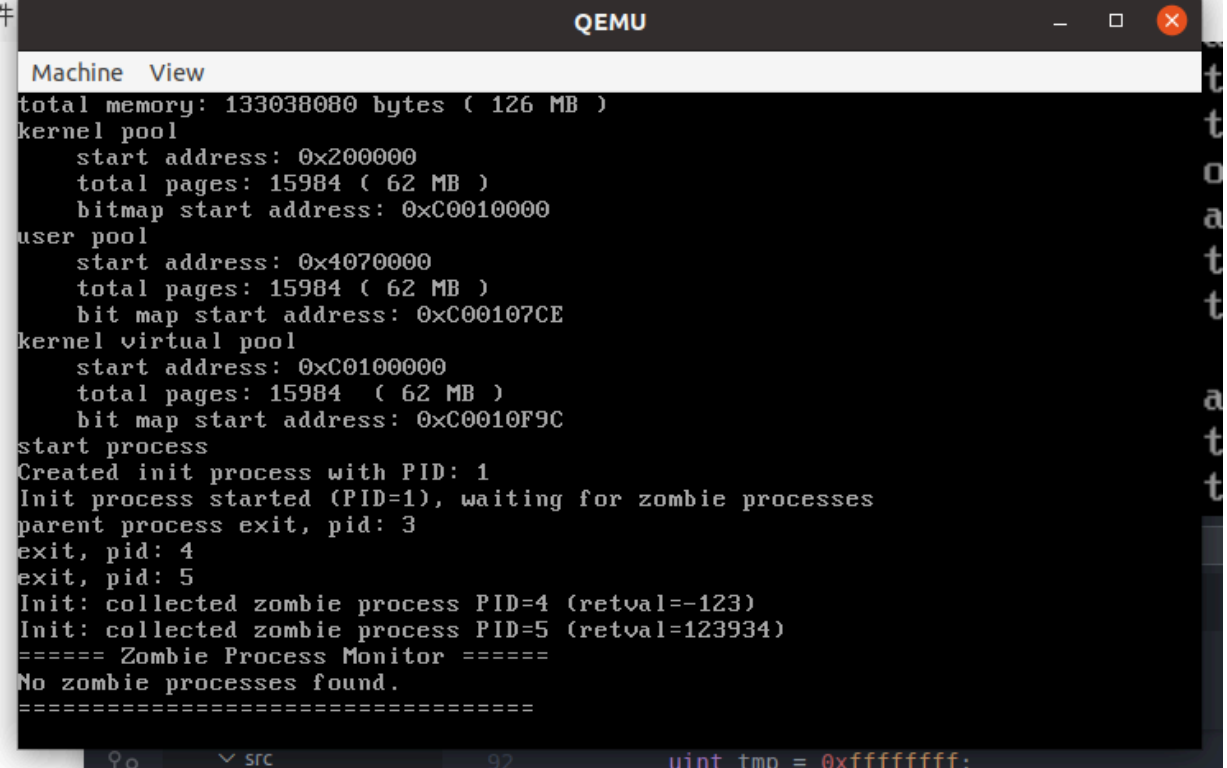
    printf("start process\n");

    // 首先创建init进程 (确保pid为1)
    int initPid = programManager.executeThread(init_process, nullptr, "init", 1);
    printf("Created init process with PID: %d\n", initPid);
    // 创建一个监控僵尸进程的线程
    programManager.executeThread(monitor_zombie_thread, nullptr, "monitor zombie",
1);
    programManager.executeProcess((const char *)first_process, 1);
    asm_halt();
}

```

运行结果如下:

可以看到init进程将子进程4和5回收了剩余资源, 成功回收了这两个僵尸进程。



```

Machine View
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
Created init process with PID: 1
Init process started (PID=1), waiting for zombie processes
parent process exit, pid: 3
exit, pid: 4
exit, pid: 5
Init: collected zombie process PID=4 (retval=-123)
Init: collected zombie process PID=5 (retval=123934)
===== Zombie Process Monitor =====
No zombie processes found.
=====
uint tmp = 0xffffffff;

```

五、实验总结和心得体会

1. 这次的实验充分体现了调试追踪程序运行的重要性, 在实验中通过gdb的断点调试和单步执行, 我更清晰的了解到整个系统调用的全过程。
2. 这也是这个学期最后的一次实验。一个学期下来, 依照仓库中的实验指导, 我也是慢慢的把实验做了下来。我感觉老师和学长们编写的这个教程是非常详尽的, 基本上把每一步的原理都给我们讲清楚了, 而且还层层递进, 使得整个实验的知识更加层次化。

3. 认识到操作系统区分用户进程与内核进程的目的是为了保护计算机，防止进程不该访问的内容被访问或被攻击。而用户态与内核态靠特权级区分，CPU也将对应不同特权级给与不同的操作。因此我们需要通过设定中断来实现中断，以此达到特权级转换的效果。而在上述创建的用户进程中，当我们需要访问到内核代码即特权级更高的部分时，就需要借助特权级转移来实现。而这一特权级转移的中转站就是系统调用，即改变特权级访问内核后还能将特权级换回来的函数。

六、参考资料

1. <https://www.cnblogs.com/yychuyu/p/15553400.html>
2. <https://zhuanlan.zhihu.com/p/356414911>
3. <https://www.cnblogs.com/Gotogoo/p/5250622.html>