



# 本科生实验报告

实验课程: 操作系统原理实验

实验名称: 从内核态到用户态

专业名称: 计算机科学与技术

学生姓名: 数据删除

学生学号: 数据删除

实验成绩:

报告时间: 2025 年 5 月 29 日

# 1. 实验要求

## Assignment 1 系统调用

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

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

## Assignment 2 Fork 的奥秘

实现 fork 函数，并回答以下问题。

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

## Assignment 3 哼哈二将 wait & exit

实现 wait 函数和 exit 函数，并回答以下问题。

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

的有效方法。

## 2. 实验过程

### Assignment 1 系统调用

系统调用的具体流程为：

1. 初始化系统调用服务，设置中断描述符表中 0x80 号中断为系统调用入口
2. 将系统调用处理函数注册到系统调用表中
3. 在用户程序中使用 `asm_system_call` 调用系统调用
4. 系统调用参数通过寄存器传递 (`eax, ebx, ecx, edx, esi`)
5. 触发 0x80 中断，CPU 从用户态切换到内核态
6. 执行相应的系统调用处理函数
7. 从内核态返回用户态，继续执行用户程序

系统调用从用户态切换到内核态时，会发生特权级切换，CPU 自动从 TSS 中加载内核栈指针 (`esp0`)，并在内核栈中保存用户态的上下文 (`ss, esp, eflags, cs, eip`)。CPU 送入 `esp` 的值是在将当前进程调度上的 CPU 执行时，通过 `ProgramManager::activateProgramPage` 放入了 TSS 的。

因此测试系统调用的基本流程为：

1. 创建一个用户进程，观察 `esp` 是否从内核空间转移到用户空间，同时观察 CPU 是否已经准备好 `esp0` 以供切换。
2. 编写简单的系统调用并从用户进程调用之。
3. 在 `int 0x80` 前后下断，观察 `esp` 的变化情况。
4. 进入系统调用后，检查用户态上下文是否正确保存于系统栈。
5. 检查系统调用参数是否正确传递。
6. 检查系统调用后是否正确回到用户空间。

## Assignment 2 Fork 的奥秘

Fork 函数的主要功能是创建当前进程的一个副本，即子进程。子进程与父进程几乎完全相同，但有以下区别：

1. 它们有不同的进程 ID (PID)
2. 子进程的父进程 ID 是父进程的 PID
3. 子进程获得父进程资源的副本，而不是共享这些资源
4. 在父进程中，fork 返回子进程的 PID；在子进程中，fork 返回 0
5. 实现 fork 的关键是完整复制父进程的资源，包括：
  - a) 0 特权级栈（内核栈）
  - b) PCB（进程控制块）
  - c) 页目录表和页表
  - d) 物理页内容
  - e) 虚拟地址池
6. fork 函数在子进程中返回 0，在父进程中返回子进程的 PID。这是通过在复制进程 0 级栈时设置返回值寄存器 eax 来实现的。

fork 的具体步骤为：

1. 创建一个新的进程
2. 复制父进程的 PCB 信息到子进程
3. 复制父进程的用户虚拟地址池到子进程
4. 复制父进程的页目录表、页表和物理页到子进程
5. 设置子进程 fork 的返回值为 0
6. 设置父进程的返回值为子进程的 PID

父进程在 fork 的返回值是子进程 pid，这是显然的，但要让子进程的 pid 返回值是 0 就需要费一番功

夫。具体来说，需要改动子进程 0 级栈的 eax 值为 0，然后调用 asm\_load\_process 装载 pss，让子进程在相同上下文但不同的栈和物理页上恢复运行。

## Assignment 3 哼哈二将 wait & exit

exit 函数的实现步骤为：

1. 标记当前进程的 PCB 状态为 DEAD
2. 保存退出码到 PCB 的 returnValue 字段
3. 释放进程占用的资源（物理页、页表、页目录表、虚拟地址池 bitmap 等）
4. 调用调度器进行进程切换
5. 要在进程正常结束时自动调用 exit(0)，需要在创建进程时，在用户栈中设置返回地址为 exit，参数为 0。

wait 函数的实现步骤为：

1. 遍历所有进程，查找当前进程的子进程
2. 如果找到一个 DEAD 状态的子进程，获取其返回值，释放其 PCB，并返回其 PID
3. 如果没有子进程，则返回-1
4. 如果存在子进程但都未终止，则调用 ProgramManager::schedule 进行调度。
5. 子进程的回收由父进程处理。

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。如果放置不管，子进程退出后会成为僵尸进程。因为子进程回收由父进程处理，父进程退出后必须指定另外一个进程接管子进程。因此仿照 linux 的实现创建一个 init 进程，该进程不会退出且 pid 固定为 1，当父进程退出时，在 exit 函数里找到其所有子进程，把这些子进程的父进程 pid 全部改成 1，就可以做到接管的效果。init 进程不断进行 wait 操作，把处于 DEAD 状态的子进程及时释放。

在内核线程里创建的进程，其父进程 pid 均为 0，退出后同样无法回收，同理在 exit 时进行检查，如果即将退出的进程没有父进程，也将其挂到 init 用以销毁。

### 3. 关键代码

#### Assignment 1 系统调用

测试代码：

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

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

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

#### Assignment 2 Fork 的奥秘

测试代码：

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

    if (pid == -1) {
        printf("can not fork\n");
    }
    else {
        if (pid) {
            printf("I am father, fork return: %d\n", pid);
        }
        else {

```

```

        printf("I am child, fork return: %d, my pid: %d\n", pid,
programManager.running->pid);
    }
}

asm_halt();
}

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

```

## Assignment 3 哼哈二将 wait & exit

修改 ProgramManager::exit，使其能够处理孤儿进程和僵尸进程：

```

void ProgramManager::exit(int ret) {
    // 关中断
    interruptManager.disableInterrupt();
    // 第一步，标记 PCB 状态为`DEAD`并放入返回值。
    PCB* program = this->running;
    program->retValue = ret;
    program->status = ProgramStatus::DEAD;

    printf("exit program pid=%d name \"%s\" ret=%d\n", program->pid, program->name,
ret);

    uint *pageDir, *page;
    uint paddr;

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

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

            for (int j = 0; j < 1024; ++j) {

```

```

        if (!(page[j] & 0x1)) {
            continue;
        }

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

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

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

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

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

// 第三步，将未完成的子进程挂到 init 进程中
for (auto pid : allPrograms) {
    PCB* child = getPCB(pid);
    if (child->parentPid == program->pid) {
        child->parentPid = 1; // 将子进程的父进程设置为 init 进程
    }
}

// 第四步，如果当前进程没有父进程，则将自己也挂到 init 进程中。
if (program->parentPid == 0) {
    program->parentPid = 1; // 将当前进程的父进程设置为 init 进程
}

// 第五步，立即执行线程/进程调度。
schedule();
}

```

测试代码：

```

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

    if (pid) {
        pid = fork();
        if (pid) {
            // while ((pid = wait(&retval)) != -1) {

```

```
// printf("wait for a child process, pid: %d, return value: %d\n",
//         pid, retval);
// }

// printf("all child process exit, programs: %d\n",
//     programManager.allPrograms.size());

printf("father process exit, pid: %d\n", programManager.running->pid);
// 直接退出父进程, 不使用 exit
}

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);
}

}

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

void init() {
    while (int pid = wait(nullptr)) {
        if (pid != -1) {
            printf("process %d exit. current programs count = %d\n", pid,
programManager.allPrograms.size());
            for (int pid : programManager.allPrograms) {
                PCB* program = programManager.getPCB(pid);
                printf("pid: %d, name: %s, status: %d\n", pid, program->name,
(int)program->status);
            }
        }
    }
}

void first_thread(void*) {
    printf("start process\n");
}
```

```

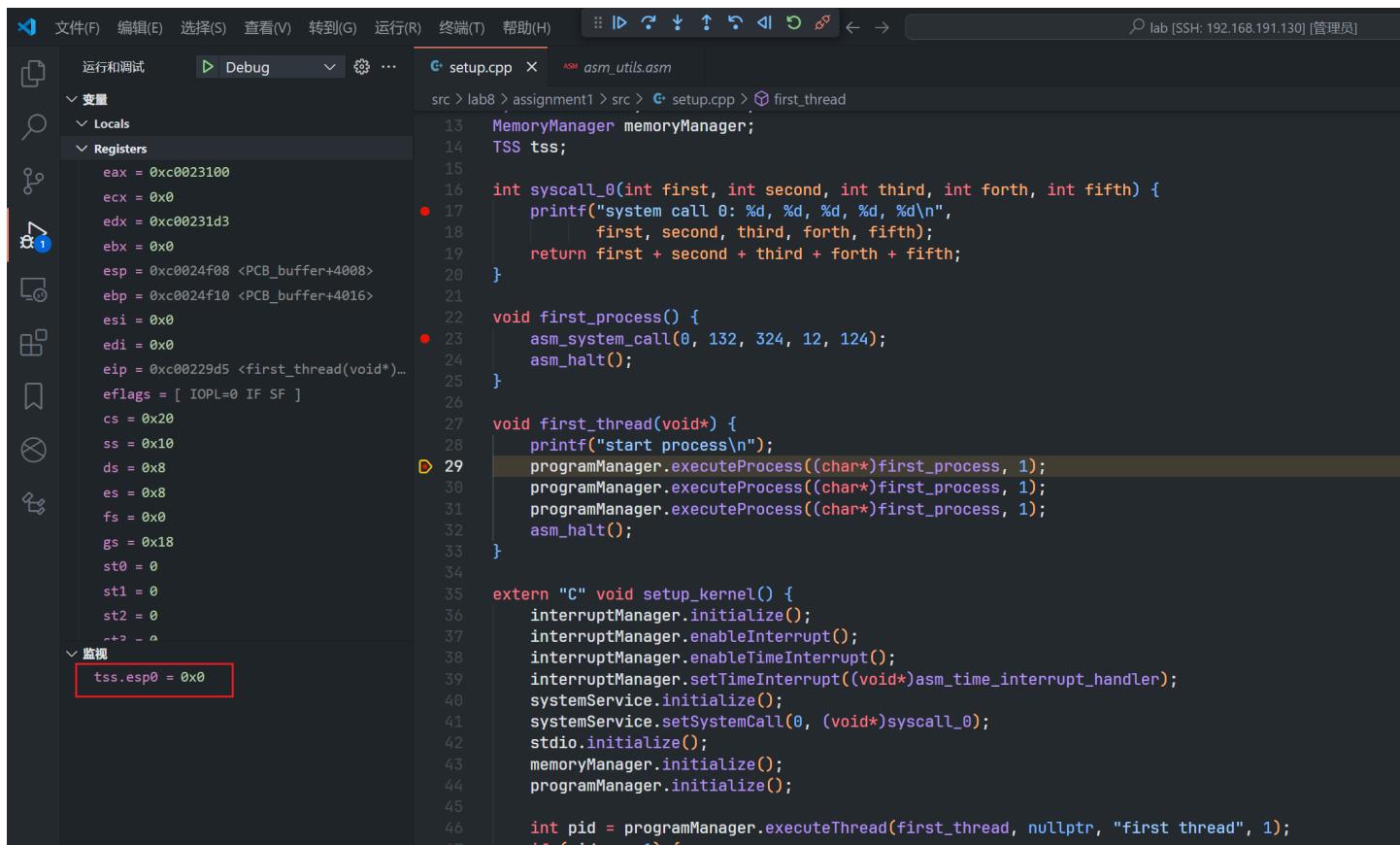
programManager.executeProcess((const char*)init, 1);
programManager.executeProcess((const char*)first_process, 1);
programManager.executeThread(second_thread, nullptr, "second", 1);
asm_halt();
}

```

## 4. 实验结果

### Assignment 1 系统调用

首先在内核态下断，观察 tss.esp0，目前仍然是 0：



```

文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H) | 运行和调试 Debug 变量 ... | C+ setup.cpp x ASM asm_utils.asm
src > lab8 > assignment1 > src > C+ setup.cpp > first_thread
13 MemoryManager memoryManager;
14 TSS tss;
15
16 int syscall_0(int first, int second, int third, int forth, int fifth) {
17     printf("system call 0: %d, %d, %d, %d, %d\n",
18            first, second, third, forth, fifth);
19     return first + second + third + forth + fifth;
20 }
21
22 void first_process() {
23     asm_system_call(0, 132, 324, 12, 124);
24     asm_halt();
25 }
26
27 void first_thread(void*) {
28     printf("start process\n");
29     programManager.executeProcess((char*)first_process, 1);
30     programManager.executeProcess((char*)first_process, 1);
31     programManager.executeProcess((char*)first_process, 1);
32     asm_halt();
33 }
34
35 extern "C" void setup_kernel() {
36     interruptManager.initialize();
37     interruptManager.enableInterrupt();
38     interruptManager.enableTimeInterrupt();
39     interruptManager.setTimeInterrupt((void*)asm_time_interrupt_handler);
40     systemService.initialize();
41     systemService.setSystemCall(0, (void*)syscall_0);
42     stdio.initialize();
43     memoryManager.initialize();
44     programManager.initialize();
45
46     int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
47 }

```

创建进程进入用户态，经过调度和 activateProgramPage 后，esp0 和 esp 均变化，esp0 现在变为内核栈地址，而 esp 所在的地址区域变为用户内存：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试 (Run and Debug), Debug (断点), Settings, ...
- Code Editor:** setup.cpp (ASM asm\_utils.asm) tab. The code includes C++ and assembly sections. A red box highlights the assembly section:

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

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

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

extern "C" void setup_kernel() {
    interruptManager.initialize();
    interruptManager.enableInterrupt();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void*)asm_time_interrupt_handler);
    systemService.initialize();
    systemService.setSystemCall(0, (void*)syscall_0);
    stdio.initialize();
    memoryManager.initialize();
    programManager.initialize();

    int pid = programManager.executeThread(first_thread, nullptr, "first thread", 1);
    if (pid < 0) {
        // handle error
    }
}
```
- Registers View:** 显示了多个寄存器的值，其中 esp 和 tss.esp0 被高亮显示并用红色边框包围。
- Locals View:** 显示了局部变量的值。
- Breakpoints:** 在代码中设置了一些断点，其中第 17 行和第 251 行被标记为已命中 (hit).

单步进入来到中断前一刻：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试 (Run and Debug), Debug (断点), Settings, ...
- Code Editor:** setup.cpp (ASM asm\_utils.asm) tab. The code includes assembly instructions. A red box highlights the assembly section:

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

mov eax, [ebp + 2 * 4]
mov ebx, [ebp + 3 * 4]
mov ecx, [ebp + 4 * 4]
mov edx, [ebp + 5 * 4]
mov esi, [ebp + 6 * 4]
mov edi, [ebp + 7 * 4]

int 0x80

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

ret
```
- Registers View:** 显示了多个寄存器的值，其中 ebx、ecx、edx、esi、edi、ds、es、fs 和 gs 被高亮显示。
- Locals View:** 显示了局部变量的值。
- Breakpoints:** 在代码中设置了一些断点，其中第 251 行被标记为已命中 (hit).

进入中断，esp 变化到内核区：

The screenshot shows a debugger interface with the following details:

- Registers:** A list of registers with their current values:
  - eax = 0x0
  - ecx = 0x144
  - edx = 0xc
  - ebx = 0x84
  - esp = 0xc0025f4c <PCB\_buffer+8172>
  - ebp = 0x8048fc0
  - esi = 0x7c
  - edi = 0x0
  - eip = 0xc0022df4 <asm\_system\_call\_handler+1>
  - eflags = [ IOPL=0 ]
  - cs = 0x20
  - ss = 0x10
  - ds = 0x33
  - es = 0x33
  - fs = 0x33
  - gs = 0x0
  - st0 = 0
  - st1 = 0
  - st2 = 0
  - ^+ - ^
- 监视 (Watch):** A list of memory locations:
  - tss.esp0 = 0xc0025f60
- ASM View:** The assembly code for the `asm_system_call_handler` function, showing the stack setup and parameter pushing.

发现此时 esp 为 0xc0025f4c，比 esp0 少了 20，说明 CPU 自动将一些数据写入了内核栈，用于中断返回。

打印栈顶的 5 个 4 字节整数：

```
x/5xw $esp
0xc0025f4c <PCB_buffer+8172>: 0xc0022de7      0x0000002b      0x00000202      0x08048f9c
0xc0025f5c <PCB_buffer+8188>: 0x0000003b
```

发现这 5 个数（从先到后）分别是原用户栈的 ss, esp, eflags, cs, next eip，可以对比前两张图得到该结论。

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试, Debug, ...
- Code Editor:** 显示文件 setup.cpp 和 asm\_utils.asm。代码中包含对 syscall\_0 的调用，以及对内存管理器、中断管理和进程管理器的初始化。
- Registers View:** 显示了多个寄存器的值，包括 eax, ecx, edx, ebx, esp, ebp, esi, edi, eip, eflags, cs, ss, ds, es 等。
- Locals View:** 显示了局部变量 first, second, third, forth, fifth 的值。
- Watch View:** 监视 tss.esp0 = 0xc0025f60。

继续执行，成功跳转到 syscall\_0 并输出：

The terminal window displays the following output:

```
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
start process
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
```

输出了正确的数字，说明参数成功从用户内存转移到内核内存。

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试 (Run and Debug), 变量 (Variables), Locals, Registers, 监视 (Watch), 搜索 (Search), 窗口 (Windows), 帮助 (Help).
- Registers Tab:** 显示了寄存器的值，如 eax = 0x250, ecx = 0x0, edx = 0x0, ebx = 0x0, esp = 0x8048fe8, ebp = 0x8048ff0, esi = 0x0, edi = 0x0, eip = 0xc00229b7, eflags = [ IOPL=0 IF PF ], cs = 0x2b, ss = 0x3b, ds = 0x33, es = 0x33, fs = 0x33, gs = 0x0, st0 = 0, st1 = 0, st2 = 0.
- Locals Tab:** 显示了 tss.esp0 = 0xc0025f60.
- Assembly Tab:** 显示了以下汇编代码：

```
src > lab8 > assignment1 > src > setup.cpp > first_process
1 #include "interrupt.hpp"
2 #include "program.hpp"
3 #include "stdio.hpp"
4 #include "syscall.hpp"
5 #include "asm_utils.hpp"
6 #include "memory.hpp"
7 #include "tss.hpp"
8
9 InterruptManager interruptManager;
10 ProgramManager programManager;
11 STDIO stdio;
12 SystemService systemService;
13 MemoryManager memoryManager;
14 TSS tss;
15
16 int syscall_0(int first, int second, int third, int forth, int fifth) {
17     printf("system call 0: %d, %d, %d, %d, %d\n",
18           first, second, third, forth, fifth);
19     return first + second + third + forth + fifth;
20 }
21
22 void first_process() {
23     asm_system_call(0, 132, 324, 12, 124);
24     asm_halt();
25 }
26
27 void first_thread(void*) {
28     printf("start process\n");
29     programManager.executeProcess((char*)first_process, 1);
30     programManager.executeProcess((char*)first_process, 1);
31     programManager.executeProcess((char*)first_process, 1);
32     asm_halt();
33 }
```

调用结束，成功回到用户态的下一条语句。

## Assignment 2 Fork 的奥秘

追踪数据复制的过程，copyProcess 首先把 0 级栈复制，然后将 childpss->eax 设为 0，子进程的上下文就和父进程一致，除了返回值，因此子进程在进行 asm\_start\_process 之后就能直接从 fork 返回，该过程全程对用户透明。

```

    < file(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H) < lab [SSH: 192.168.191.130] [管理员]
    运行和调试 Debug ... C program.cpp 1
    src > lab8 > common > src > kernel > C program.cpp > ProgramManager::copyProcess
    348     interruptManager.setInterruptStatus(status);
    349     return pid;
    350 }
    351
    352 bool ProgramManager::copyProcess(PCB* parent, PCB* child) {
    353     // 复制进程0级栈
    354     auto childpss = (ProcessStartStack*)((int)child + PAGE_SIZE - sizeof(ProcessStartStack));
    355     auto parentpss = (ProcessStartStack*)((int)parent + PAGE_SIZE - sizeof(ProcessStartStack));
    356     memcpy(childpss, parentpss, sizeof(ProcessStartStack));
    357     // 设置子进程的返回值为0
    358     childpss->eax = 0;
    359
    360     // 准备执行asm_switch_thread的栈的内容
    361     child->stack = (int*)childpss - 7;
    362     child->stack[0] = 0;
    363     child->stack[1] = 0;
    364     child->stack[2] = 0;
    365     child->stack[3] = 0;
    366     child->stack[4] = (int)asm_start_process;
    367     child->stack[5] = 0;           // asm_start_process 返回地址
    368     child->stack[6] = (int)childpss; // asm_start_process 参数
    369
    370     // 设置子进程的PCB
    371     child->status = ProgramStatus::READY;
    372     child->parentPid = parent->pid;
    373     child->priority = parent->priority;
    374     child->ticks = parent->ticks;
    375     child->ticksPassedBy = parent->ticksPassedBy;
    376     strcpy(child->name, parent->name);
    377
    378     // 复制用户虚拟地址池
    379     int bitmapLength = parent->userVirtual.resources.length;
    380     int bitmapBytes = ceil(bitmapLength, 8);
    381     memcpy(child->userVirtual.resources.bitmap, parent->userVirtual.resources.bitmap, bitmapBytes);
    382
    383     // 从内核中分配一页作为中转页
    384     uint buffer = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    385     if (!buffer) {
    386         child->status = ProgramStatus::DEAD;
    387         return false;
    388     }

```

已知父进程的 pid 是 1，因此子进程 pid 应当为 2。在 `asm_switch_thread` 下断，找到 pid 为 2 的调度：

```

    < file(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H) < lab [SSH: 192.168.191.130] [管理员]
    运行和调试 Debug ... C program.cpp 1
    src > lab8 > common > src > kernel > C program.cpp > ProgramManager::schedule
    146     }
    147     else if (running->status == ProgramStatus::DEAD) {
    148         // 回收线程，子进程留到父进程回收
    149         if (!running->pageDirectoryAddress) {
    150             releasePCB(running);
    151         }
    152     }
    153
    154     int item = readyPrograms.front();
    155     readyPrograms.pop();
    156     PCB* next = getPCB(item);
    157     PCB* cur = running;
    158     next->status = ProgramStatus::RUNNING;
    159     running = next;
    160
    161     activateProgramPage(next);
    162     asm_switch_thread(cur, next);
    163
    164     interruptManager.setInterruptStatus(status);
    165
    166 }
    167
    168 > void ProgramManager::MESA_WakeUp(PCB* program) { ...
    169 }
    170
    171 > void ProgramManager::initializeTSS() { ...
    172 }
    173
    174 > int ProgramManager::executeProcess(const char* filename, int priority) {
    175     bool status = interruptManager.getInterruptStatus();
    176     interruptManager.disableInterrupt();
    177
    178     // 在线程创建的基础上初步创建进程的PCB
    179     int pid = executeThread((ThreadFunction)load_process, (void*)filename, nullptr, priority);
    180     if (pid == -1) {
    181
    182     }

```

单步进入，在 `asm_switch_thread` 结束后预期跳转到 `asm_start_process`：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试, Debug, 变量, Registers, 等。
- Registers Panel:** 显示了多个寄存器的值，如 eax = 0xc00261e0, ebx = 0x0, esp = 0xc0027194, eip = 0xc0022fed, etc.
- Assembly View:** 显示了.asm文件的内容，包括asm\_update\_cr3, asm\_start\_process, 和asm\_halt等函数的汇编代码。

这是因为在 ProgramManager::copyProcess 中，子进程栈的返回地址被设为了 asm\_start\_process，参数则指向了复制了的 0 级栈。

打印栈顶数据：

```
x/20wx $esp
0xc0027194 <PCB_buffer+12212>: 0x00000000      0xc002719c      0x00000000      0x00000000
0xc00271a4 <PCB_buffer+12228>: 0x08048fa0      0xc00261bc      0x00000000      0x00000000
0xc00271b4 <PCB_buffer+12244>: 0x00000000      0x00000000      0x00000000      0x00000033
0xc00271c4 <PCB_buffer+12260>: 0x00000033      0x00000033      0xc0022f57      0x0000002b
0xc00271d4 <PCB_buffer+12276>: 0x00000206      0x08048f7c      0x0000003b      0x00000000
```

可以看到 esp+4 为 0xc002719c，即 childpss 的地址。该地址即将被赋至 esp，这样栈空间就转移到了子进程的栈空间：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbars:** 运行和调试, Debug, 变量, Registers, Locals, 监视
- Registers Panel:** 显示了多个寄存器的值，其中 **eax** 的值为 **0xc002719c**，被高亮显示。
- Locals Panel:** 显示了当前函数的局部变量。
- Assembly View:** 显示了汇编代码，包含三个主要部分：
  - asm\_update\_cr3:** 处理 CR3 寄存器更新。
  - asm\_start\_process:** 处理新进程启动，将堆栈指针 **esp** 设为 **eax** 的值。
  - asm\_ltr:** 处理线程切换，将线程描述符 **ltr** 存入堆栈。
- Call Stack:** 显示了调用堆栈，从 **asm\_start\_process** 到 **asm\_ltr**。

同时 **esp+10** 为 **0x8048fa0**，这个是进程调用 **int 0x80** 后一条指令的地址，通过它来返回用户空间。

**popad** 取出通用寄存器的值，此时 **eax** 变为 0，即 **fork** 返回值在子进程的结果，同时段寄存器也变为用户段选择子，**ebp** 变为 **0x8048fa0**，完全恢复为调用系统调用前的状态：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(E) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbars:** 工具栏 (包含各种调试图标)
- Registers View:** 显示了多个寄存器的值，包括 eax, ecx, edx, ebx, esp, ebp, esi, edi, eip, eflags, cs, ss, ds, es, fs, gs, st0, st1, st2。esp 的值为 0xc00271cc，被高亮显示。
- Locals View:** 显示了 tss.esp0 = 0xc00271e0。
- Assembly View:** 显示了 asm\_utils.asm 的汇编代码。当前光标在第 358 行，标记为 iret。其他可见的汇编指令包括 pop esi, pop ebx, pop ebp, ret, push eax, mov eax, dword[esp+8], mov cr3, eax, pop eax, ret, mov eax, dword[esp+4], mov esp, eax, popad, pop gs, pop fs, pop es, pop ds，以及 ltr word[esp + 1 \* 4]。

iret 后成功返回 asm\_system\_call 在 int 0x80 后一条语句的位置：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbars:** 运行和调试, Debug, ...
- Search Bar:** la
- Registers View:** 显示了多个寄存器的值，包括 eax=0x0, ecx=0x0, edx=0x0, ebx=0x0, esp=0x8048f7c, ebp=0x8048fa0, esi=0x0, edi=0x0, eip=0xc0022f57, eflags=[IOPL=0 IF PF], cs=0x2b, ss=0x3b, ds=0x33, es=0x33, fs=0x33, gs=0x0, st0=0, st1=0, st2=0。
- Assembly View:** 显示了汇编代码，包含 `asm\_system\_call` 段落，其中包含 `push` 和 `pop` 指令以保存堆栈上的寄存器值。当前光标在第 253 行的 `pop gs` 指令上。

成功返回，eax=0，调用栈和父进程一致：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试, Debug, ...
- Variables View:**
  - Locals:** eax = 0x0, ecx = 0x0, edx = 0x0, ebx = 0x0, esp = 0x8048fc8, ebp = 0x8048fd0, esi = 0x0, edi = 0x0, eip = 0xc0022a59 <fork()+29>, eflags = [ IOPL=0 IF ], cs = 0x2b, ss = 0x3b, ds = 0x33, es = 0x33, fs = 0x33, gs = 0x0, st0 = 0, st1 = 0, st2 = 0.
  - Registers:** tss.esp0 = 0xc00271e0
  - Watch:** 调用堆栈 (因 step 已暂停)
- Code View:** syscall.cpp
 

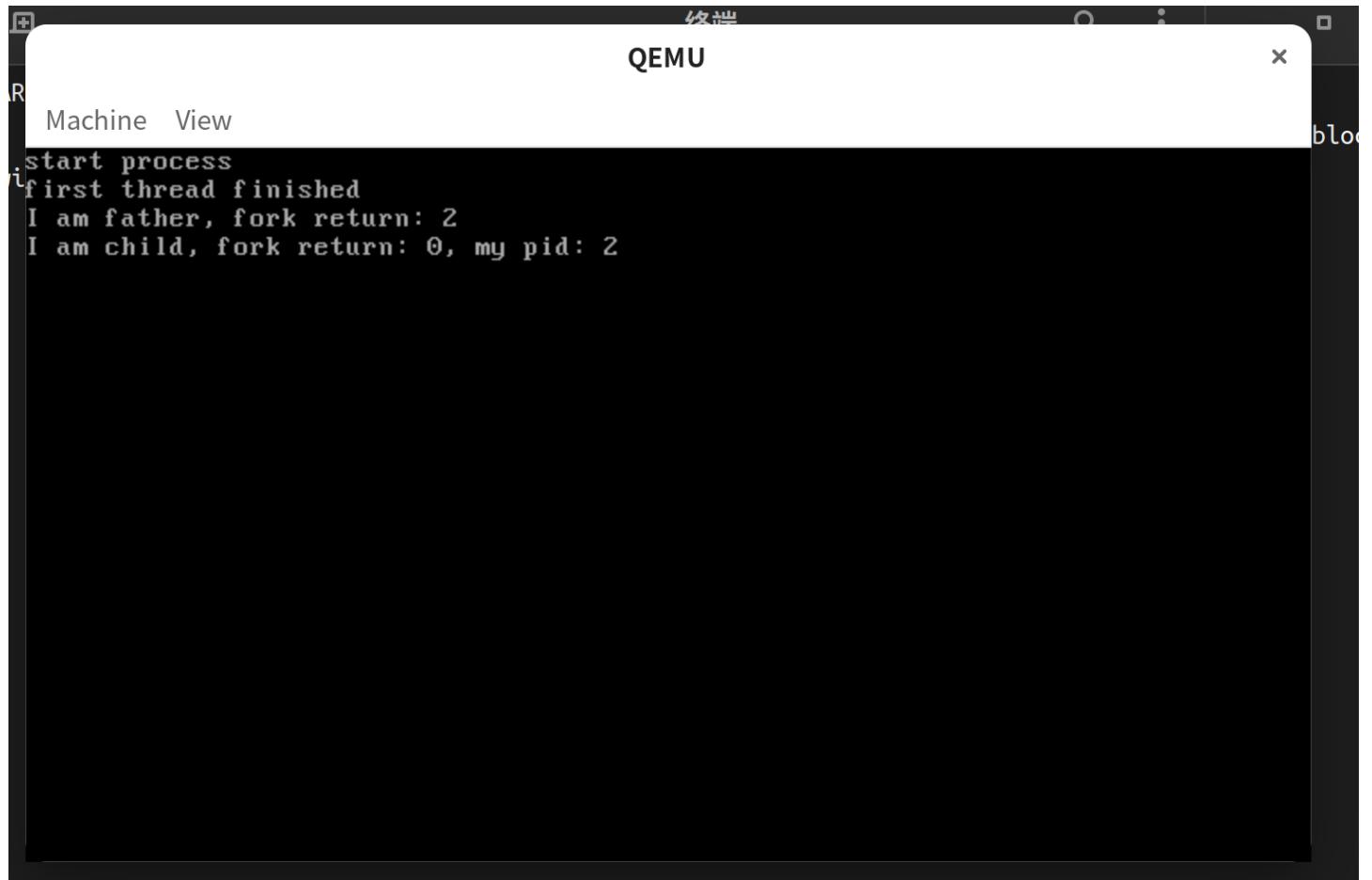
```

6
7     uint system_call_table[MAX_SYSTEM_CALL];
8
9     SystemService::SystemService() {
10         initialize();
11     }
12
13     void SystemService::initialize() {
14         memset((char*)system_call_table, 0, sizeof(int) * MAX_SYSTEM_CALL);
15         // 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
16         interruptManager.setInterruptDescriptor(0x80, (uint32)asm_system_call_handler, 3);
17     }
18
19     bool SystemService::setSystemCall(int index, void* function) {
20         system_call_table[index] = (uint)function;
21         return true;
22     }
23
24     int fork() {
25         return asm_system_call(2);
26     }
27
28     int syscall_fork() {
29         return programManager.fork();
30     }
31
32     void exit(int ret) {
33         asm_system_call(3, ret);
34     }
35
36     void syscall_exit(int ret) {
37         programManager.exit(ret);
38     }
39
40     int wait(int *retval) {
41         return asm_system_call(4, (int)retval);
42     }
43
44     int syscall_wait(int *retval) {
45         return programManager.wait(retval);
46     }
      
```
- Output View:** 问题, 输出, 调试控制台, 终端
 

输出	调试控制台	终端	
breakpoint 2, programManager...schedule (this=0x000004000 &programManager=0x00000000)			
163	asm_switch_thread(cur, next);		
x/20wx \$esp			
0xc0027194 <PCB_buffer+12212>: 0x00000000	0xc002719c	0x00000000	0x00000000
0xc00271a4 <PCB_buffer+12228>: 0x08048fa0	0xc00261bc	0x00000000	0x00000000

可以认为返回过程和父进程几乎一致，但从上下文的角度也可以说子进程是在 `copyProcess(parent, child)` 之后就返回，没有进行接下来的 `interruptManager.setInterruptStatus(status);` 和 `return pid;` 操作，因此不会返回自己的 pid。

运行结果：

A screenshot of a QEMU terminal window titled "QEMU". The window has a dark background and white text. At the top, there are icons for "File", "Machine", "View", and "Help". The title bar says "QEMU". The main area of the terminal shows the following text:

```
start process
first thread finished
I am father, fork return: 2
I am child, fork return: 0, my pid: 2
```

## Assignment 3 哼哈二将 wait & exit

exit 和 wait 的验证比较简单，只需要在对应函数下断然后对关键过程进行分析即可。

首先断在入口点，通过监视可知此时将要 exit 的是 pid=2 的进程，分析可知这是 first\_process 的主进程，它在进行两次 fork 之后直接退出，留下两个孤儿进程：

The screenshot shows a debugger interface with the following details:

- Top Bar:** Includes file operations (文件(F)、编辑(E)、选择(S)、查看(V)、转到(G)、运行(R)、终端(T)、帮助(H)), terminal icons, and a connection status (lab [SSH: 192.168.191.130] [管理]).
- Left Sidebar:** Contains icons for file operations, search, and navigation.
- Toolbars:** Includes "运行和调试" (Run and Debug), "Debug" dropdown, and other configuration icons.
- Variables View:** Shows the current state of variables:
  - Locals:** This section is expanded, showing local variables: `this` (args), `ret` (nullptr), `program` (Object@0xa0), `pageDir` (Object@0xc003493c), `page` (Object@0x1), and `paddr` (Object@0xc0034880).
  - Registers:** This section is selected and expanded, showing CPU register values.
- Registers View:** Shows the current state of CPU registers.
- Breakpoints View:** Shows a list of breakpoints, with one entry highlighted: "因 breakpoint 已暂停".
- Code Editor:** Displays the source code for `program.cpp`. The code is annotated with line numbers (456-494). A specific line, 475, is highlighted with a yellow background, indicating it is the current instruction being executed.

从变量窗口读出 `ret=0`, 是正常退出时的默认参数。

第一第二步忽略，来到第三步，现在需要将两个孤儿进程交给 `init` 进程管理：

```
src > lab8 > common > src > kernel > program.cpp > ProgramManager::exit
468     // 回退从内核分离的线程
469     memoryManager.releasePages(AddressPoolType::KERNEL, buffer, 1);
470     return true;
471 }

void ProgramManager::exit(int ret) {
    // 关中断
    interruptManager.disableInterrupt();
    // 第一步，标记PCB状态为DEAD 并放入返回值。
    PCB* program = this->running;
    program->retValue = ret;
    program->status = ProgramStatus::DEAD;

    printf("exit program pid=%d name \"%s\" ret=%d\n", program->pid, program->name, ret);

    uint *pageDir, *page;
    uint paddr;

    // 第二步，如果PCB标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池bitmap的空间。
    if (program->pageDirectoryAddress) {...}
}

// 第三步，将未完成的子进程挂到 init 进程中
for (auto pid : allPrograms) {
    PCB* child = getPCB(pid);
    if (child->parentPid == program->pid) {
        child->parentPid = 1; // 将子进程的父进程设置为 init 进程
    }
}

// 第四步，如果当前进程没有父进程，则将自己也挂到 init 进程中。
if (program->parentPid == 0) {
    program->parentPid = 1; // 将当前进程的父进程设置为 init 进程
}

// 第五步，立即执行线程/进程调度。
schedule();
}

int ProgramManager::wait(int* retval) {...
```

然后发现父进程的父进程为 pid=0，也将他挂到 init 下，方便销毁：

```
src / lab8 > common > src > kernel > program.cpp > ProgramManager::exit
468     // 从内核分离的内存页
469     memoryManager.releasePages(AddressPoolType::KERNEL, buffer, 1);
470     return true;
471 }

void ProgramManager::exit(int ret) {
    // 关中断
    interruptManager.disableInterrupt();
    // 第一步，标记PCB状态为`DEAD`并放入返回值。
    PCB* program = this->running;
    program->retValue = ret;
    program->status = ProgramStatus::DEAD;

    printf("exit program pid=%d name \"%s\" ret=%d\n", program->pid, program->name, ret);

    uint *pageDir, *page;
    uint paddr;

    // 第二步，如果PCB标识的是进程，则释放进程所占用的物理页、页表、页目录表和虚拟地址池bitmap的空间。
    if (program->pageDirectoryAddress) {...}
}

// 第三步，将未完成的子进程挂到 init 进程中
for (auto pid : allPrograms) {
    PCB* child = getPCB(pid);
    if (child->parentPid == program->pid) {
        child->parentPid = 1; // 将子进程的父进程设置为 init 进程
    }
}

// 第四步，如果当前进程没有父进程，则将自己也挂到 init 进程中。
if (program->parentPid == 0) {
    program->parentPid = 1; // 将当前进程的父进程设置为 init 进程
}

// 第五步，立即执行线程/进程调度。
schedule();
}

int ProgramManager::wait(int* retval) { ... }
```

之后调用 `schedule`, `exit` 流程结束。

`wait` 函数的流程也类似：

```
src > lab8 > common > src > kernel > program.cpp > ProgramManager::wait
529     program->parentPid = 1; // 将当前进程的父进程设置为 init 进程
530 }
531
532 // 第五步，立即执行线程/进程调度。
533 schedule();
534 }
535
536 int ProgramManager::wait(int* retval) {
537     bool interrupt, flag;
538
539     while (true) {
540         interrupt = interruptManager.getInterruptStatus();
541         interruptManager.disableInterrupt();
542         auto it = allPrograms.find_if([&](int pid) {
543             return getPCB(pid)->parentPid == this->running->pid;
544         });
545
546         if (it) // 找到一个可返回的子进程
547         {
548             auto child = getPCB(*it);
549             if (child->status == ProgramStatus::DEAD) {
550                 int pid = child->pid;
551                 if (retval) {
552                     *retval = child->retValue;
553                 }
554                 releasePCB(child);
555                 interruptManager.setInterruptStatus(interrupt);
556                 return pid;
557             }
558             else {
559                 interruptManager.setInterruptStatus(interrupt);
560                 schedule(); // 存在子进程，但子进程的状态不是DEAD
561             }
562             else {
563                 interruptManager.setInterruptStatus(interrupt);
564                 return -1;
565             }
566         }
567     }
568 }
```

从 running->pid 读出当前正在 wait 的进程 pid 为 1，即 init 进程。

在找到可返回子进程位置下断，以 pid=5 的子进程的销毁过程为例：

The screenshot shows a debugger interface with the following details:

- File Menu:** 文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
- Toolbar:** 运行和调试 (Run and Debug), Debug, ...
- Left Sidebar:** 变量 (Variables), Locals, Registers, 监视 (Watch), 调用堆栈 (Call Stack), 因 step 已暂停 (Step has paused).
- Current File:** program.cpp 1
- Code View:** The code is in C++ and shows a loop for waiting on processes. A break point is set at line 551. The current line being executed is highlighted.

```
src > lab8 > common > src > kernel > program.cpp > ProgramManager::wait
    program->parentPid = 1; // 将当前进程的父进程设置为 init 进程
}
// 第五步，立即执行线程/进程调度。
schedule();

int ProgramManager::wait(int* retval) {
    bool interrupt, flag;
    while (true) {
        interrupt = interruptManager.getInterruptStatus();
        interruptManager.disableInterrupt();
        auto it = allPrograms.find_if([&](int pid) {
            return getPCB(pid)->parentPid == this->running->pid;
        });

        if (it) // 找到一个可返回的子进程
        {
            auto child = getPCB(*it);
            if (child->status == ProgramStatus::DEAD) {
                int pid = child->pid;
                if (retval) {
                    *retval = child->returnValue;
                }
                releasePCB(child);
                interruptManager.setInterruptStatus(interrupt);
                return pid;
            }
            else {
                interruptManager.setInterruptStatus(interrupt);
                schedule(); // 存在子进程，但子进程的状态不是DEAD
            }
        }
        else {
            interruptManager.setInterruptStatus(interrupt);
            return -1;
        }
    }
}
```

该子进程的返回值在 `child->returnValue` 中，读出 `0x1e41e`，即 `123934`，符合预期，该值将被赋给 `retval`。

然后调用 `releasePCB` 释放 PCB，并返回子进程 `pid`。

运行结果：

The screenshot shows a terminal window titled "QEMU" with the following log output:

```
start process
father process exit, pid: 2
exit program pid=2 name "" ret=0
thread exit
release PCB pid=3 name "second"
exit, pid: 4
exit program pid=4 name "" ret=-123
exit, pid: 5
exit program pid=5 name "" ret=123934
release PCB pid=4 name ""
process 4 exit. return -123. current programs count = 4
pid: 5, name: , status: 4
pid: 0, name: first thread, status: 2
pid: 1, name: , status: 1
pid: 2, name: , status: 4
release PCB pid=5 name ""
process 5 exit. return 123934. current programs count = 3
pid: 0, name: first thread, status: 2
pid: 1, name: , status: 1
pid: 2, name: , status: 4
release PCB pid=2 name ""
process 2 exit. return 0. current programs count = 2
pid: 0, name: first thread, status: 2
pid: 1, name: , status: 1
```

最终只剩下内核线程和 init 进程在执行，符合预期。

## 5. 总结

下发的参考代码里 `memcpy` 的参数 `src` 和 `dst` 顺序是反过来的，因为这个调 `assignment 2` 调了一晚。