

本科生实验报告

实验课程: 操作系统原理实验

实验名称: 进程 专业名称: 计算机科学与技术

学生姓名: 张玉瑶

学生学号: 23336316

实验地点: 实验楼B203

实验成绩:

报告时间: 2025年6月2日

**Section 1 实验概述**

* 实验任务1：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。

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

* 实验任务2：实现fork函数，并回答以下问题。

1. 请根据代码逻辑和执行结果来分析fork实现的基本思路。
2. 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。
3. 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。

* 实验任务3：实现wait函数和exit函数，并回答以下问题。

1. 请结合代码逻辑和具体的实例来分析exit的执行过程。
2. 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。
3. 请结合代码逻辑和具体的实例来分析wait的执行过程。
4. 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。

**Section 2 实验步骤与实验结果**

------------------------- **实验任务1** -------------------------

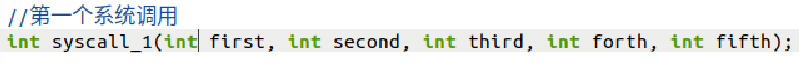
* 任务要求：编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

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

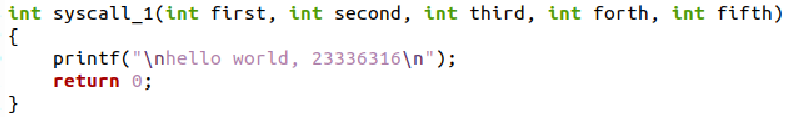
* 思路分析：利用gdb跟踪。
* 实验步骤：

1. 展现系统调用执行结果的正确性，结果截图并并说说你的实现思路。

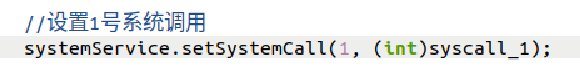
1）在syscall.h中添加新的系统调用函数声明。



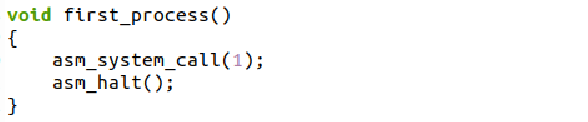
2）在setup.cpp中实现新的系统调用函数。



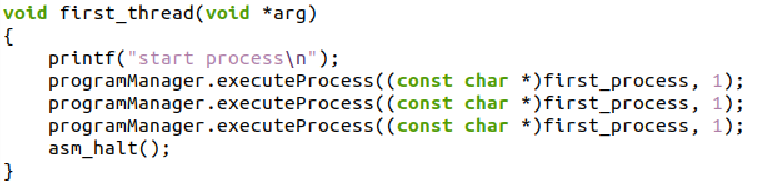
3）在setup\_kernel()函数中，添加对新系统调用的注册。



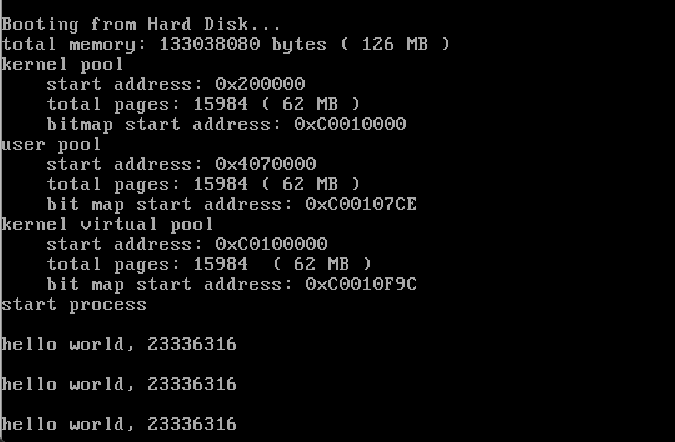
4）在进程first\_process()中添加新的系统调用。



5）在线程first\_thread()中运行三次first\_process()。

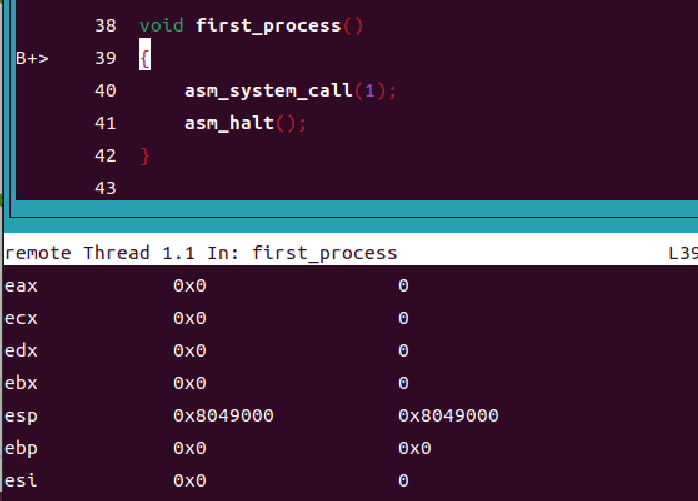


6）编译运行。可以看到新加入的系统调用也被成功调用，输出了三行“hello world，23336316”。

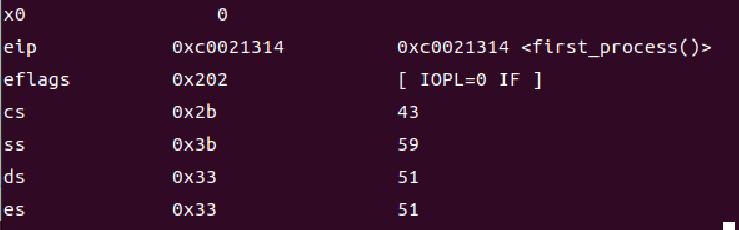


2、3. 请根据gdb来分析执行系统调用后的栈的变化情况并分析TSS的作用。

1）在first\_process()处打断点，看看系统调用前的栈：



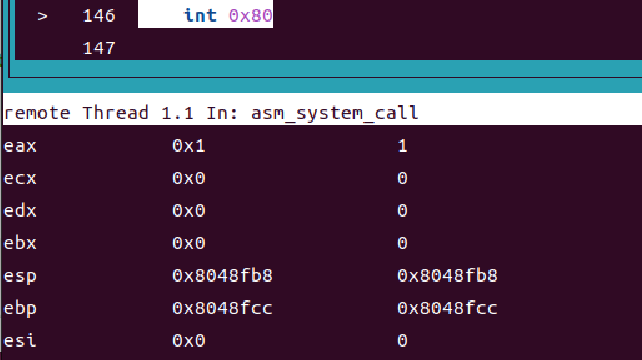




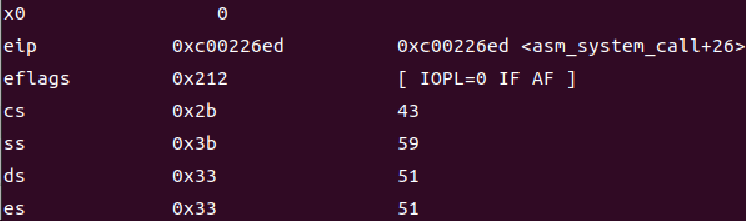


从上图中可以看到cs寄存器为0x2b，尾2位为11，即特权级为3，位于用户态。esp为0x804900，是在用户栈。ss值为0x3b。

2）进入asm\_system\_call，观察中断int 0x80执行前的栈：



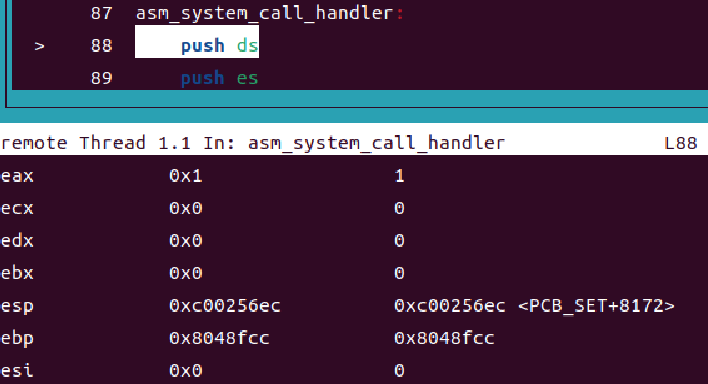




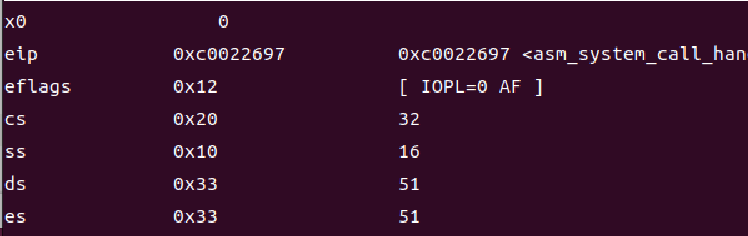


从上图中可以看到cs寄存器为0x2b，尾2位为11，即特权级为3，位于用户态。esp为0x8048fb8，是在用户栈。ss值为0x3b。

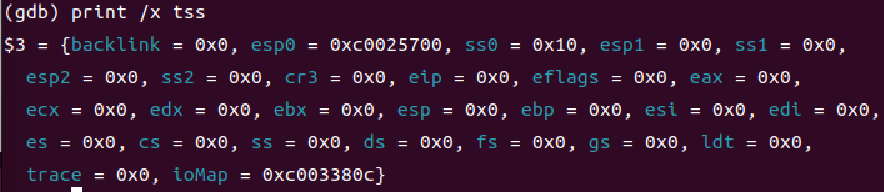
3）执行中断int 0x80，进入asm\_system\_call\_handler：









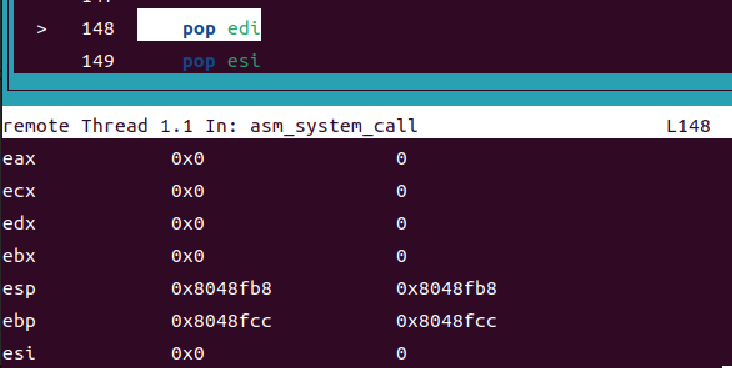




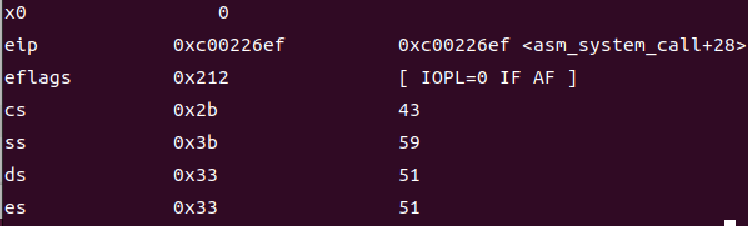
从上图中可以看到cs寄存器为0x20，尾2位为00，即特权级为0，这意味着切换到了内核态。

此时CPU自发地从TSS加载esp0 和 ss0 到 esp 和 ss，切换到内核栈。可以看到TSS的esp0为0xc0025700，eps的值为0xc00256ec（值不一样但和接近，因为CPU把esp设置为TSS.eps0后立即压入了用户态上下文，导致esp值减小，栈向低地址增长），ss的值也变成了TSS的ss值0x10。

4）中断0x80执行完成后：









可以看出中断执行完成后，esp值、cs值、ss值又变回了中断前的值，回到了用户栈，特权级变成3，ss变成0x3b。这说明又回到了用户态。

------------------------- **实验任务2** -------------------------

* 任务要求：实现fork函数，并回答以下问题。

1. 请根据代码逻辑和执行结果来分析fork实现的基本思路。
2. 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。
3. 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。

* 思路分析：利用gdb跟踪。
* 实验步骤：

**1.分析fork实现的基本思路。**

1）系统调用框架

用户态调用 fork() 会触发 int 0x80 中断，进入内核态后调用 syscall\_fork()，最终由 ProgramManager::fork() 处理。内核线程无权调用 fork()，通过检查 PCB::pageDirectoryAddress 是否为 0 来判断。

2）创建子进程

调用 executeProcess("", 0) 创建子进程的 PCB，此时子进程尚未复制资源。子进程的 PCB 被加入调度队列，但处于未初始化状态。

3）复制父进程资源

通过 copyProcess(parent, child) 复制父进程资源：

i.复制内核栈：父进程的内核栈位于 (int)parent + PAGE\_SIZE - sizeof(ProcessStartStack)，直接 memcpy 复制到子进程。设置子进程的 eax=0，使其 fork() 返回 0，父进程返回子进程 PID。

ii. 初始化子进程 PCB：继承父进程的优先级、时间片等属性，设置 parentPid 为父进程 PID。

iii. 复制虚拟地址池：复制父进程的虚拟内存管理位图。

iv.复制页表和物理内存：为子进程分配新的页目录表，复制父进程前 768 项（用户空间映射）。对每个有效的页表项，分配新的物理页，通过内核中转页复制父进程数据到子进程，更新子进程的页表项指向新物理页。

4) 子进程启动准备

设置子进程的 stack 指向一个伪造的栈帧，使其被调度时跳转到 asm\_start\_process。asm\_start\_process 通过 iret 返回到用户态，从 fork() 的返回点继续执行。

5) 测试逻辑

void first\_process() {

int pid = fork(); // 父子进程在此分叉

if (pid) {

printf("I am father, fork return: %d\n", pid);

} else {

printf("I am child, fork return: %d\n", pid);

}

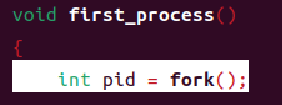
}

输出：



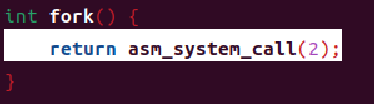
**2. 从子进程第一次被调度执行时开始，逐步跟踪子进程的执行流程一直到子进程从fork返回，根据gdb来分析子进程的跳转地址、数据寄存器和段寄存器的变化。同时，比较上述过程和父进程执行完ProgramManager::fork后的返回过程的异同。**

首先进入first\_process中的fork()函数中开始跟踪。



可以发现fork()返回的是一个系统调用。



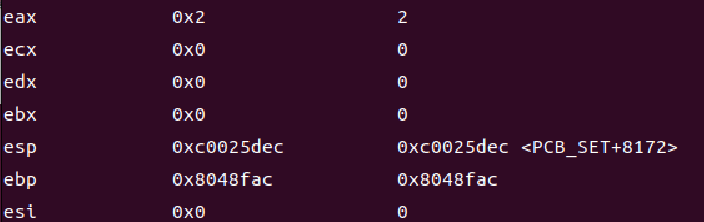


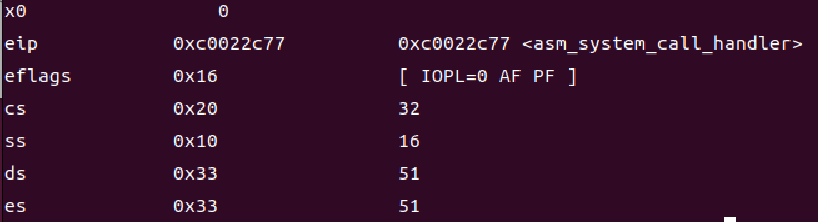
进入系统调用。进入asm\_system\_call之后，跟踪中断的执行过程。





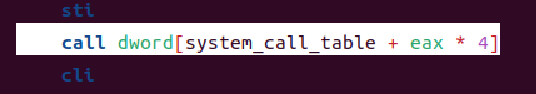
进入int 0x80。展示寄存器。可以发现特权级语句变成0，进入了内核态。



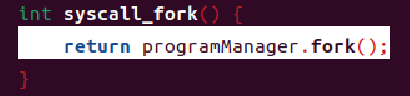


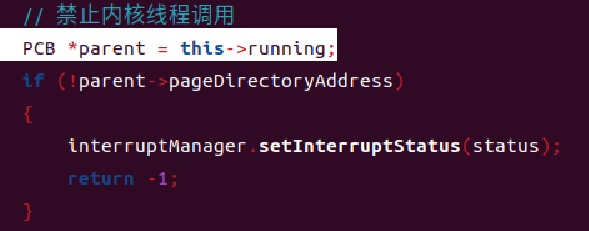
之后call调用具体的系统调用处理函数。eax的值为2，那么调用system\_call\_table[2]即为syscall\_fork。





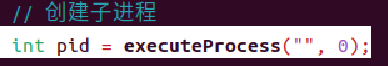
可以发现syscall\_fork()返回的是本任务最关键的函数ProgramManager::fork()。





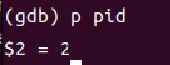
fork是进程的系统调用，因此我们禁止内核线程调用。因为内核线程并没有设置PCB::pageDirectoryAddress，所以该项为0。相反，进程有页目录表，所以该项不为0。因此，我们通过判断PCB::pageDirectoryAddress是否为0来判断当前执行的是线程还是进程。

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

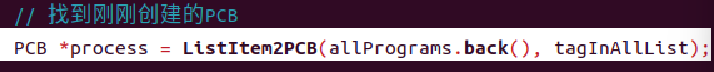


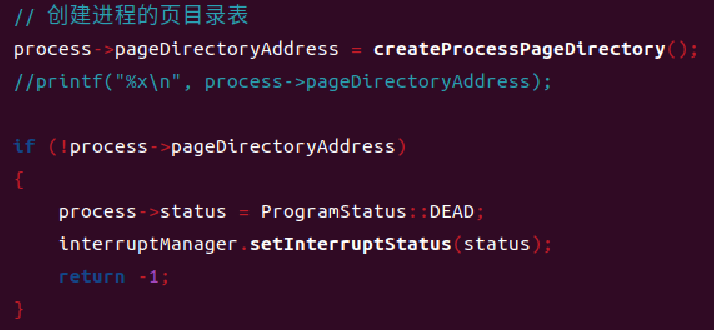
进入executeProcess的执行逻辑。我们像创建一个线程一样创建进程的PCB。打印pid可以发现值为2。

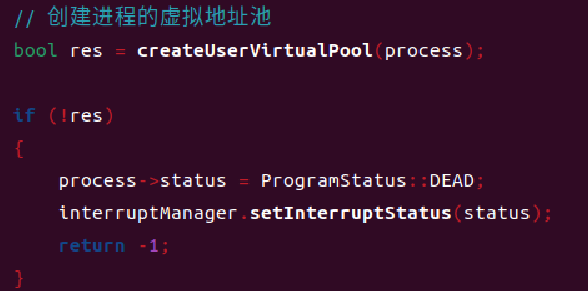




找到刚刚新建的PCB，根据executeThread的实现，一个新创建的PCB总是被放在allPrograms的末尾的。

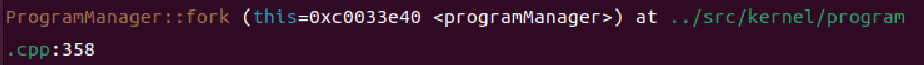




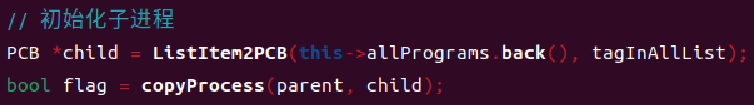


在创建进程的页目录表和进程的虚拟地址池后，我们完成了进程的创建。

接着函数返回到ProgramManager::fork()。



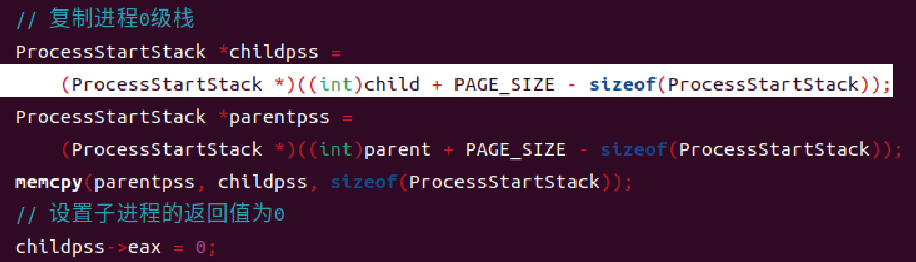
初始化子进程。然后找到刚刚创建的子进程，然后调用ProgramManager::copyProcess来复制父进程的资源到子进程中。



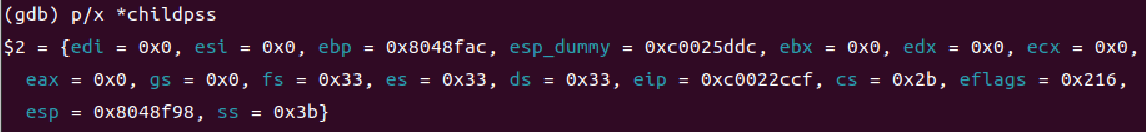
我们来研究fork实现最关键的部分——资源的复制，即函数ProgramManager::copyProcess。

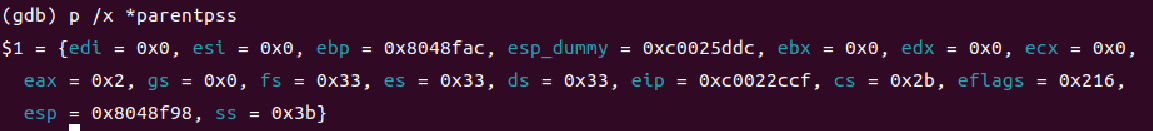
第一步，复制进程0级栈。

父进程和子进程的 0 级栈位于各自 PCB 的顶部（PAGE\_SIZE - sizeof(ProcessStartStack)）。我们从父进程的栈中复制 ProcessStartStack 到子进程的栈。最后设置子进程的eax为0（这是 fork 的返回值，保证子进程返回的pid为0）。



可以发现操作过后childpss和parentpss的信息中，除了eax都是一模一样的。其中父进程eax值为2，子进程为0。



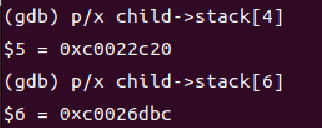


第二步，设置子进程的线程切换栈。

stack 是 线程切换时的栈帧，用于 asm\_switch\_thread 切换上下文。asm\_start\_process 是子进程的入口函数，它会加载子进程的上下文并开始执行。stack[6] 是 asm\_start\_process 的参数，即子进程的 0 级栈（childpss）。

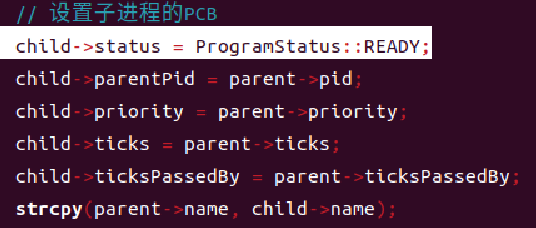
调度器切换到子进程时，会从child->stack恢复上下文，并且处理器会将stack[4]作为EIP，stack[5]作为返回地址。





第三步，复制 PCB信息。

让子进程继承父进程的 调度属性（优先级、时间片等），设置父子关系（parentPid），复制进程名。

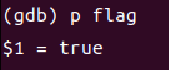


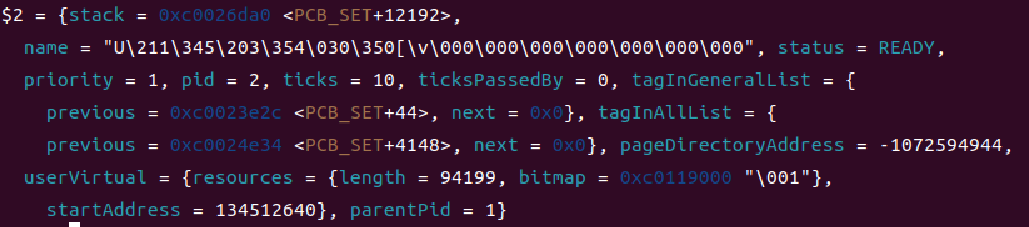
第四步，复制内存分配信息（虚拟地址池 + 页表 + 物理页）。

接下来将父进程的虚拟地址位图、页目录表、页表、物理页内容都复制给子进程，让子进程拥有和父进程相同的用户空间内存布局。

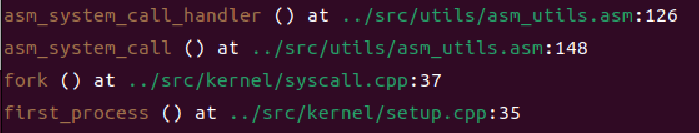


经过copyProcess的一系列复制操作之后，flag值为true，说明父进程的资源被成功复制到子进程。

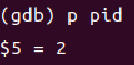


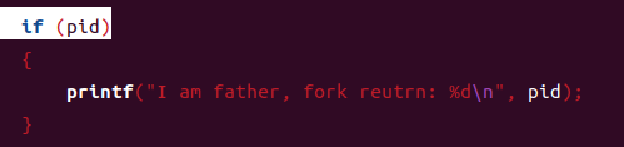


接下来返回一堆函数，最终返回到first\_process()。



来到关键的打印步骤。打印pid可以发现pid为2，输出父进程的语句。

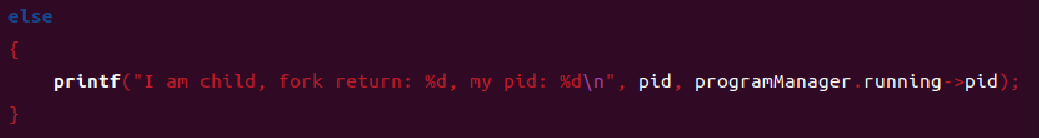






然后执行asm\_halt()。显示asm\_halt执行了两次。第二次执行后qemu又输出了语句，子进程也输出了。



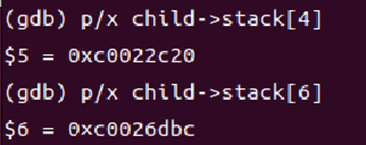




**最后对子进程的一些内容和运行过程做一些总结：**

子进程的入口函数是asm\_start\_process。我们已经把它赋值给了child->stack[4]。

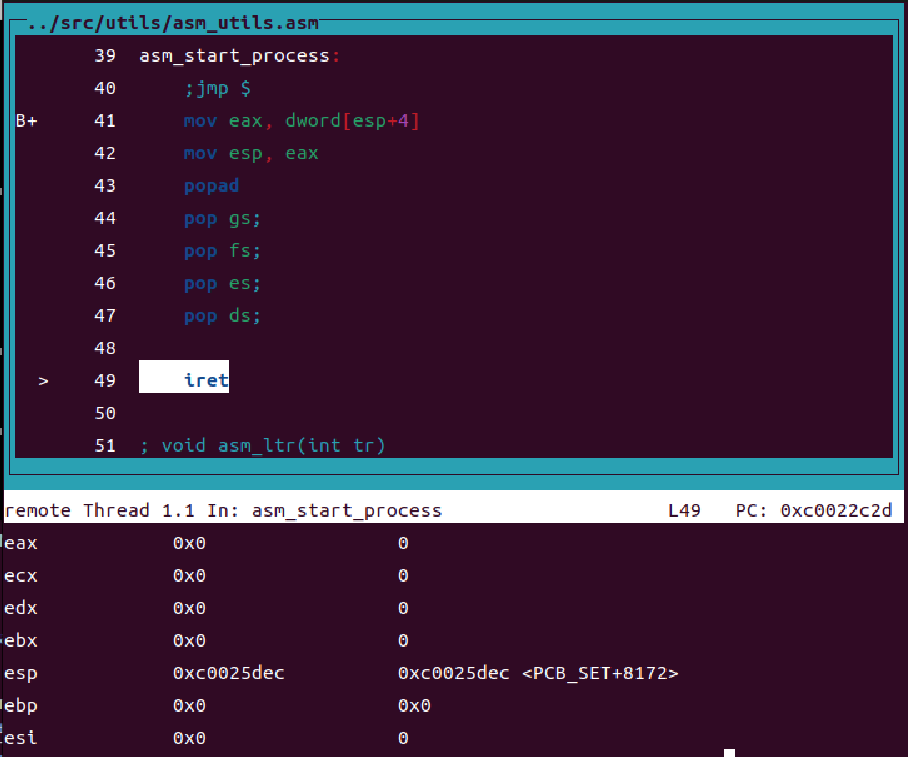


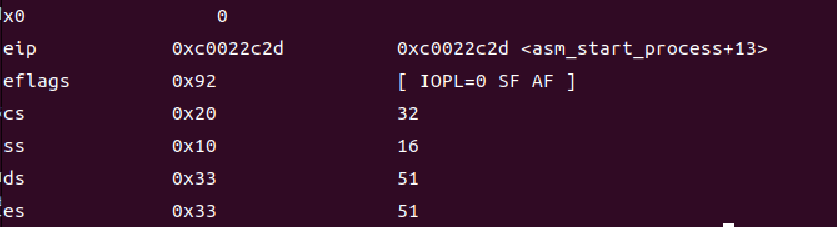


当线程切换时，asm\_switch\_thread 会从子进程的 stack 中弹出返回地址（即 asm\_start\_process）。之后，asm\_start\_process 从子进程的 0 级栈（childpss） 恢复寄存器状态，其中eip指向了和父进程相同的fork()的返回地址，esp指向了子进程的用户栈。

当通过iret指令跳转到eip时，子进程从fork()返回。

第一次跳转到asm\_start\_process。

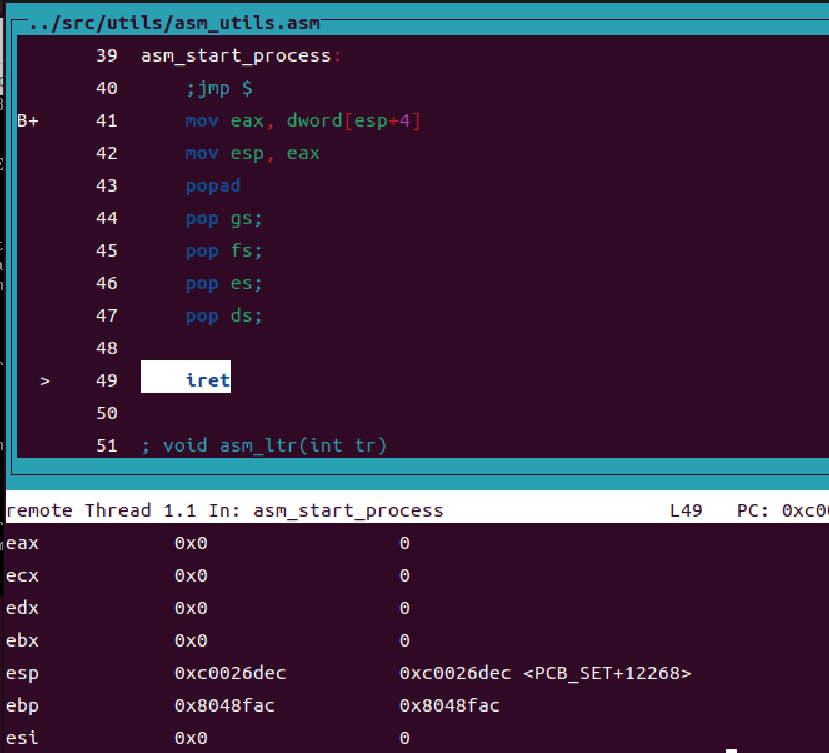


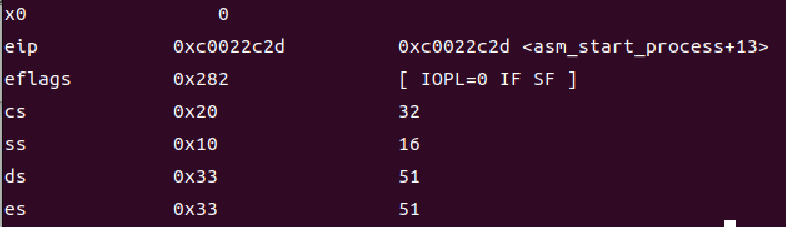




当first\_process打印完一次“I am father,fork return :2”进入了asm\_halt()，此时在gdb输入next发现再一次跳转到了asm\_start\_process。

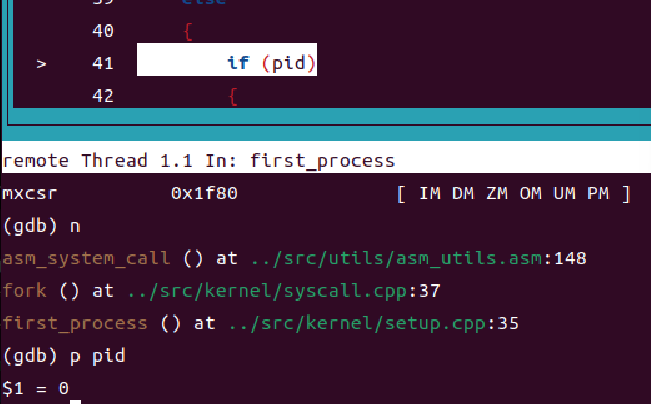
第二次跳到asm\_start\_process。







这次return之后回到了first\_process，再一次打印pid，发现值变为0。



qemu打印子进程语句。



可以对子进程和父进程的返回过程进行一下对比：

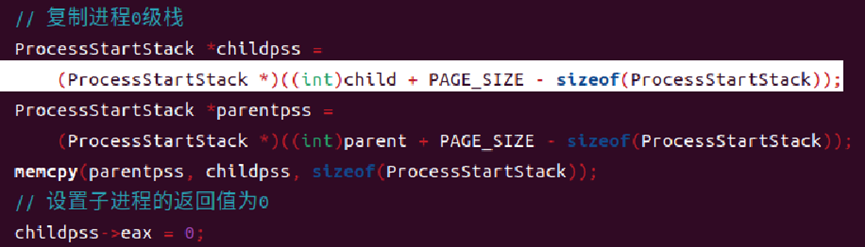
父进程通过系统调用进入内核态执行 syscall\_fork，内核为其创建一个子进程的 PCB 并复制内存布局后，父进程从内核态返回到用户态，此时 eax 寄存器保存子进程的 PID，父进程从 fork() 的调用处继续执行后续代码，表现上如同普通函数返回。

子进程由调度器选中后，通过 asm\_start\_process 汇编入口恢复上下文，从复制的 childpss 结构中加载寄存器状态（包括 eip 指向父进程 fork() 的返回地址，但 eax 强制置 0），最后通过 iret 指令切换到用户态。子进程从相同的 fork() 返回处开始执行，但返回值 (eax) 为 0，从而区分父子进程的逻辑分支。

**3. 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**。

父进程严格来讲返回的就是自己的pid，只不过是子进程复制了父进程的pid。

子进程返回0的过程是通过内核在创建子进程时显式设置其上下文中的eax寄存器为0实现的：在copyProcess函数中，子进程的0级栈（childpss）被初始化为父进程的副本后，强制将childpss->eax赋值为0；当子进程首次被调度执行时，调度器通过asm\_start\_process汇编函数从childpss恢复寄存器状态，此时eax被加载为0，接着通过iret指令返回到用户态，使得子进程从fork()的返回处继续执行时，eax自然为0，从而与父进程（返回子进程PID）区分开。这一机制完全依赖内核态对子进程上下文的精准初始化，无需运行时判断，是fork()"一次调用，两次返回"的核心设计。



------------------------- **实验任务3** -------------------------

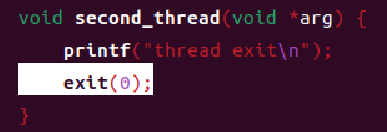
* 任务要求：实现wait函数和exit函数，并回答以下问题。

1. 请结合代码逻辑和具体的实例来分析exit的执行过程。
2. 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。
3. 请结合代码逻辑和具体的实例来分析wait的执行过程。
4. 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。

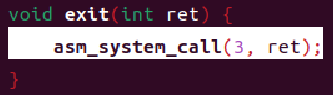
* 思路分析：熟悉exit和wait的操作，设计样例制造僵尸进程，并利用schedule解决。
* 实验步骤：

**1. 请结合代码逻辑和具体的实例来分析exit的执行过程。**

1）实例：second\_thread显式调用exit。



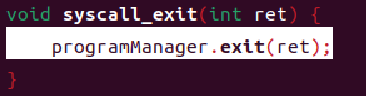
2）在exit(0)中触发三号系统调用。



3）CPU通过中断int0x80进入内核态，查找系统调用表，调用syscall\_exit：

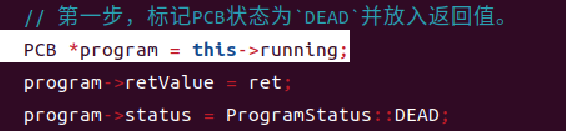






4）执行ProgramManager::exit()

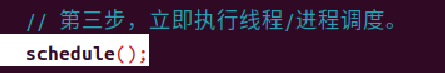
先标记当前运行线程为DEAD状态并保存返回值。



然后检查当前运行的是进程还是线程，线程不做处理。在此次由于是线程（program->pageDirectoryAddress==0），跳过了资源释放。



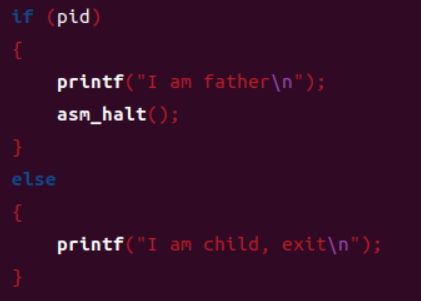
最后开始调度，切换到下一个可运行线程。

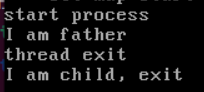


后续是当前线程PCB变成僵尸线程，等待被CPU回收。然后下一个线程被调度运行。

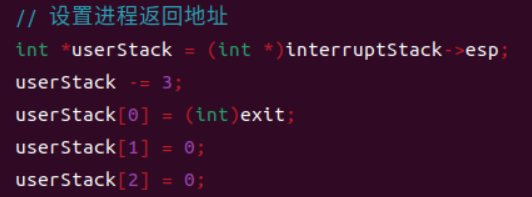
**2. 请分析进程退出后能够隐式地调用exit和此时的exit返回值是0的原因。**

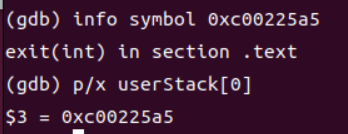
1）我们可以发现进程first\_process并没有显式调用exit，但是输出了“thread exit”，说明exit被隐式调用了。下面分析执行过程。





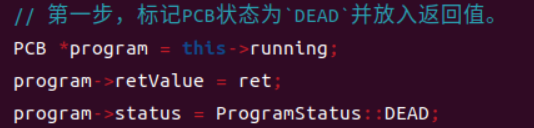
2）进程在初始化时在load\_process()中设置了用户栈。我们在进程的3特权级栈中的栈顶处userStack[0]放入exit的地址，然后CPU会认为userStack[1]是exit的返回地址，userStack[2]是exit的参数。当进程first\_process返回时， CPU会按照正常函数返回流程从栈中弹出这些预设值。 CPU从栈中弹出返回地址，得到exit地址，接着弹出exit的返回地址(0)和参数(0)

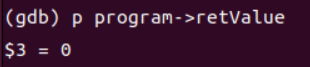




3）当exit被隐式调用时，一步步跳转进入关键函数ProgramManager::exit（），在这里我们标记了PCB状态和保存返回值。

ret就是exit(0)传入的参数0，把program->retValue设置为0，进程状态设置为DEAD。

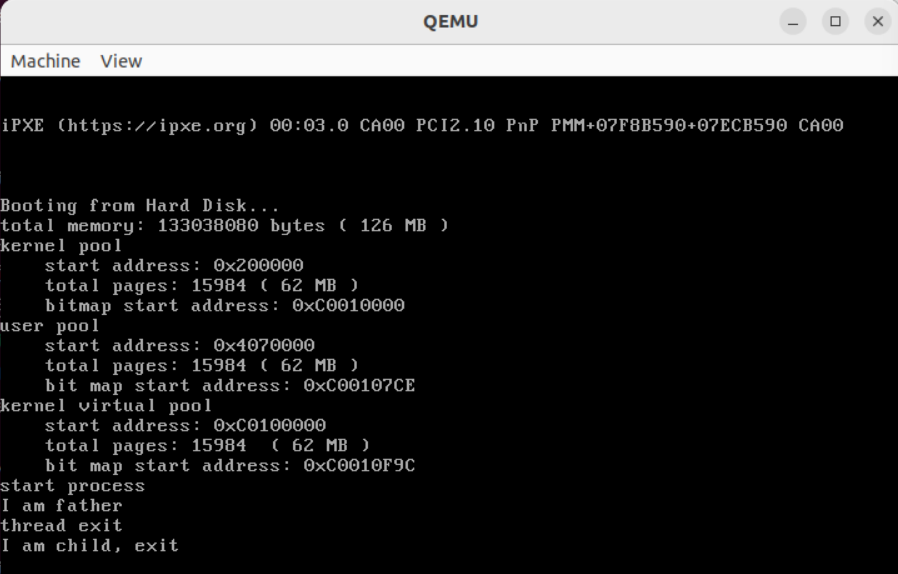




之后进程释放资源，CPU开始进程调度。

4）综上所述，进程退出后能够隐式调用exit是因为内核在创建进程时预先在其用户栈顶部设置了exit函数的地址和参数（0），当进程主函数执行完毕后，CPU会按照正常函数返回流程从栈中弹出这些预设值，从而自动跳转到exit并传入返回值0。

* 实验结果展示：



**3. 请结合代码逻辑和具体的实例来分析wait的执行过程。**

1）在first\_process中设置实例并测试。父进程调用wait。

void first\_process()

{

int pid = fork();

int retval;

if (pid)

{

pid = fork();

if (pid)

{

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

{

printf("wait for a child process, pid: %d, return value: %d\n",

pid, retval);

}

printf("all child process exit, programs: %d\n",

programManager.allPrograms.size());

asm\_halt();

}

else

{

uint32 tmp = 0xffffff;

while (tmp)

--tmp;

printf("exit, pid: %d\n", programManager.running->pid);

exit(123934);

}

}

else

{

uint32 tmp = 0xffffff;

while (tmp)

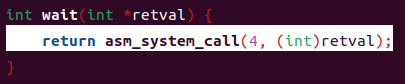
--tmp;

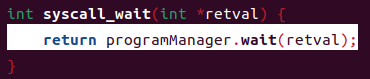
printf("exit, pid: %d\n", programManager.running->pid);

exit(-123);

}

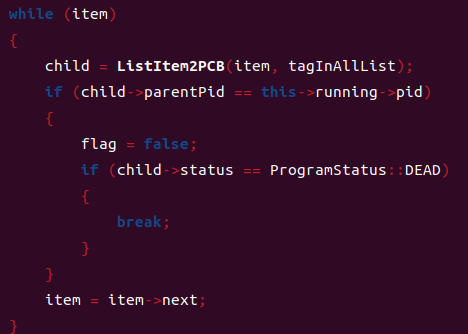
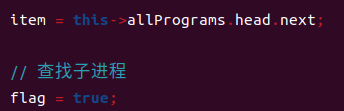
}



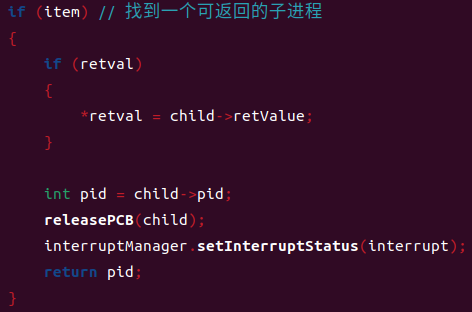


2）进入ProgramManager::wait()执行流程。

第一步，查找子进程。试图在allPrograms中找到一个状态为DEAD的子进程。allPrograms中包含了所有状态的所有进程和线程。

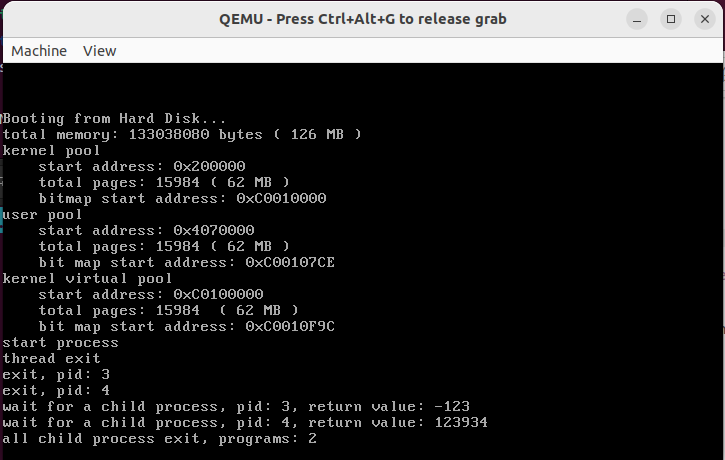


接下来找到了一个可回收的子进程。当retval不为nullptr时，我们取出子进程的返回值放入到retval指向的变量中。然后取出子进程的pid，调用releasePCB来回收子进程的PCB。





3）结果展示。

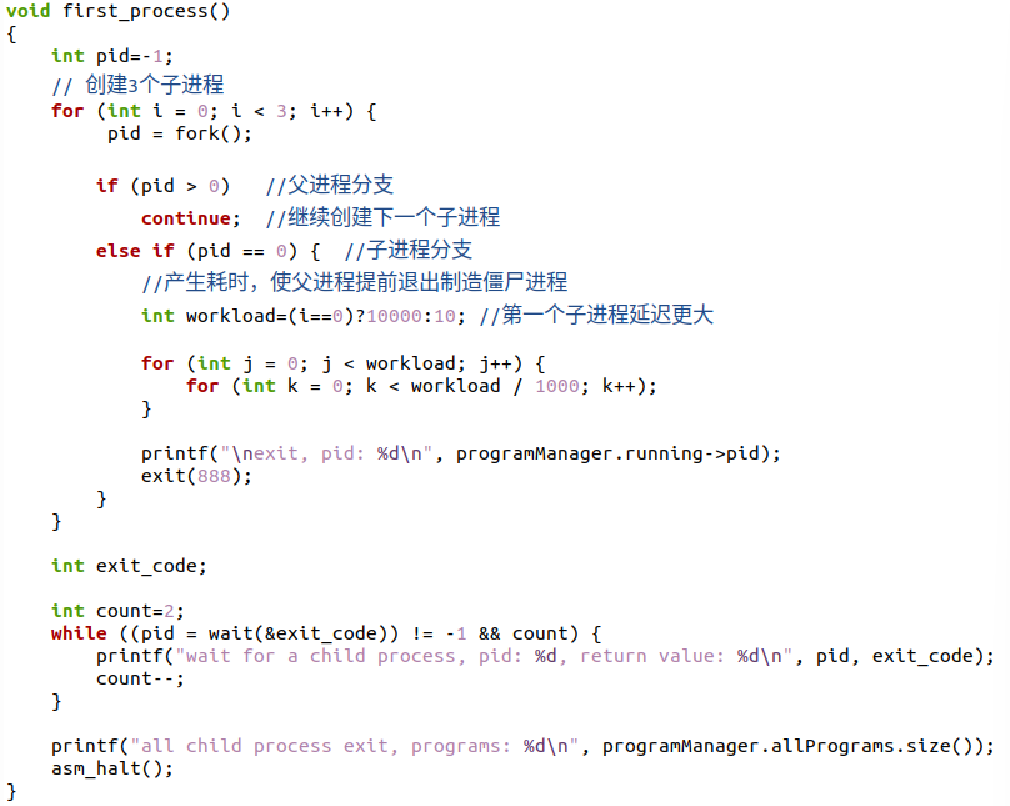


**4. 如果一个父进程先于子进程退出，那么子进程在退出之前会被称为孤儿进程。子进程在退出后，从状态被标记为DEAD开始到被回收，子进程会被称为僵尸进程。请对代码做出修改，实现回收僵尸进程的有效方法。**

1）设计测试样例制造僵尸/孤儿进程。

在first\_process中创建3个子进程。

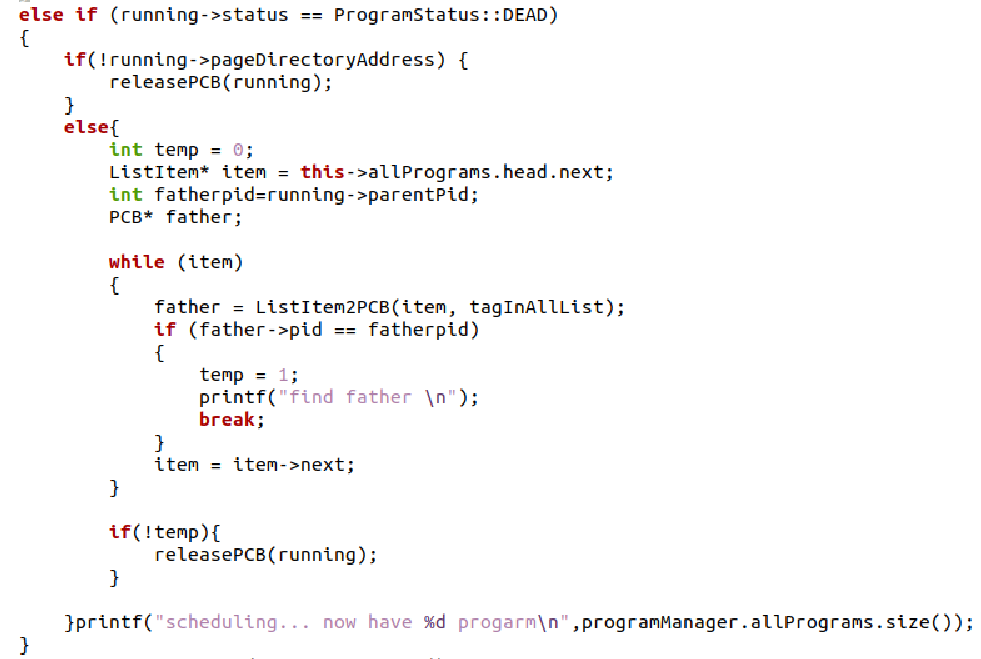
在样例中故意只wait两次，导致有一个子进程得不到wait，从而变成僵尸进程。



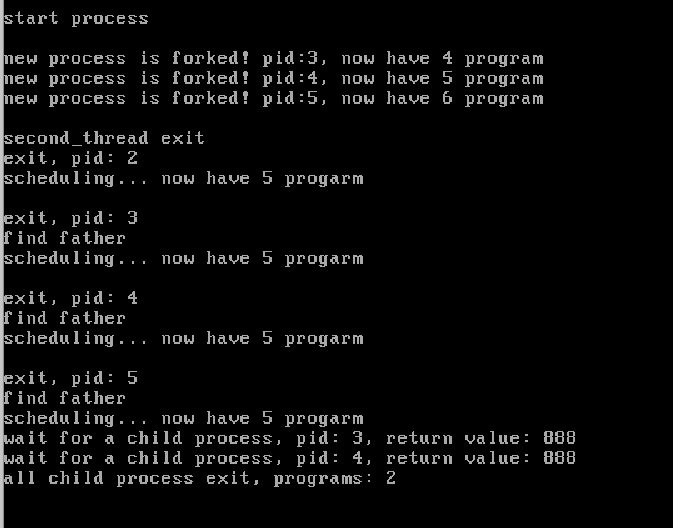


2）修改schedule函数。

在schedule中加入如图所示内容：当一个进程执行完后，如果状态设置为DEAD，检查父进程是否存在：如果父进程存在，打印“find father”并退出检查操作；否则进程自行释放PCB，完成回收操作。



3）测试运行。可以发现三个进程退出时都出现了“find father”，说明退出时父进程仍在。后面调用了两次wait，分别释放了3，4。可以看到在释放wait前一共有5个进程，wait之后父进程结束输出“all child process exit”，此时显示只有两个进程，子进程完全被回收。说明僵尸进程pid=5也被回收了。



**Section 5 实验总结与心得体会**

本次学习主要学习了进程之间的相关机制。我学习了跟踪子进程的创建，僵尸进程和孤儿进程的回收，exit和wait的实现逻辑，更进一步地加深了我对操作系统设计的理解。期间当然也遇到不少困难，比如说错误理解了僵尸进程和孤儿进程的概念，没理解透彻wait的逻辑，走了不少弯路，好在最后一一克服，收获颇丰。