

操作系统原理实验报告

**实验名称:** 实验八 从内核态到用户态

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

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。

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。

**Assignment 2 Fork的奥秘**

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。

**Assignment 3 哼哈二将 wait & exit**

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。
5. **实验过程及关键代码及实验结果**

**Assignment 1 系统调用**

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

**asm\_utils\_call**

我们利用asm\_utils\_call来保护系统调用的参数。在调用中断前，我们先保护现场，将系统调用的参数放到5个寄存器ebx, ecx, edx, esi, edi中，将系统调用号放到eax中。然后，我们将系统调用的中断向量号定义为0x80。保护现场后，使用指令int 0x80调用0x80中断。0x80中断处理函数会根据保存在eax的系统调用号来调用不同的函数。最后再恢复现场。

**SystemService**

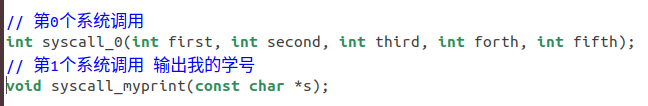
这个函数负责管理系统调用，包括初始化以及需要的系统调用。

**asm\_system\_call\_handler**

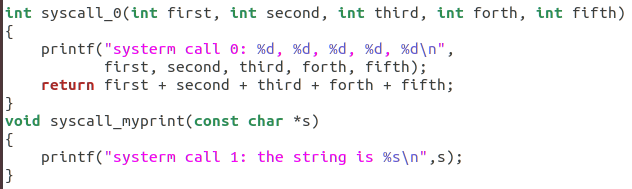
这一函数是asm\_utils\_call中的0x80中断处理函数。它的工作内容包括：保护现场；手动修改ds，es，fs和gs段寄存器；将保存在5个寄存器当中的系统调用函数压栈；开中断，调用系统调用处理的函数；修改esp寄存器；将eax保存在变量ASM\_TEMP中；恢复现场，执行中断返回，至此所有工作结束。

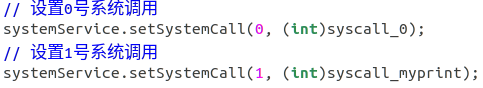
**编写个人系统调用**

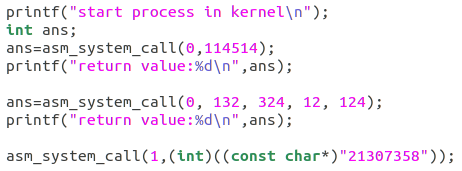
首先在头文件中声明系统调用



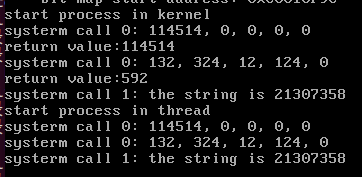
然后，在setup.cpp中实现系统调用。声明后，在文件中的setup\_kernel函数中调用





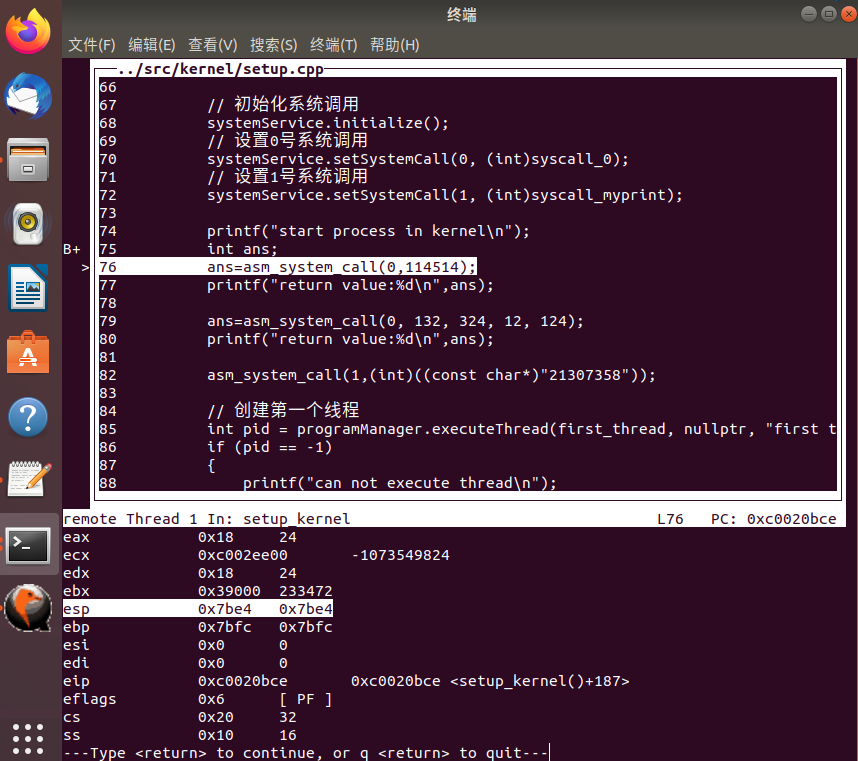


最终得到如图所示的结果。其中，在kernel的process中，可见函数正确的输出了返回值592，说明系统调用成功。

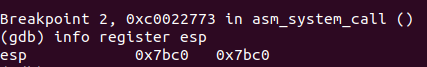
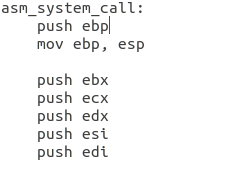


1. 请根据gdb来分析执行系统调用后的栈的变化情况

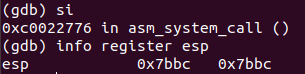
首先，设置断点至准备进入系统调用（asm\_system\_call）时。此时地址为0x7be4



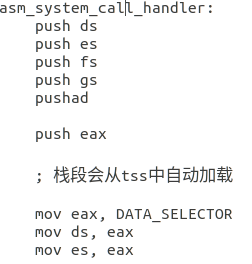
在进入asm\_system\_call后，地址为0x7bc0



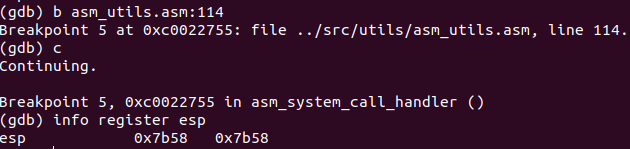
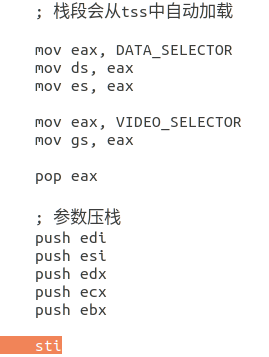
执行到mov ebp，esp后，esp中值如图所示：



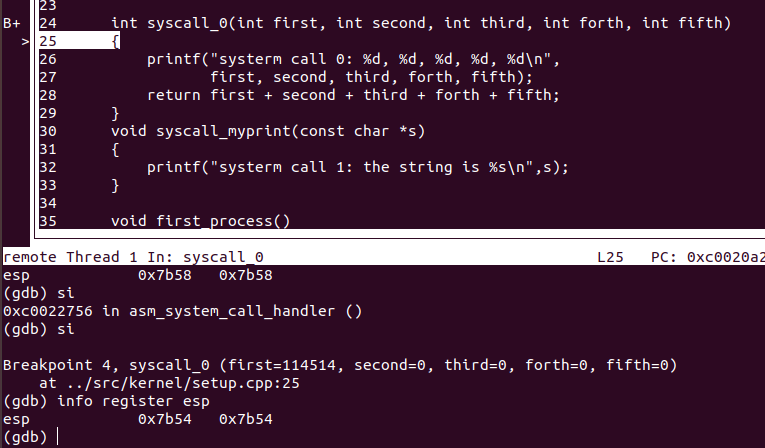
随后，进入中断函数asm\_system\_call\_handler。刚进入时，esp中内容如图所示：



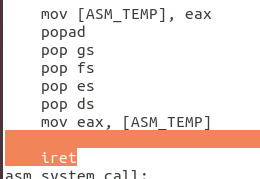
在保存完现场后，esp中值为0x7b58



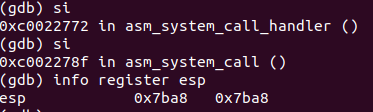
开中断，进入内核态，系统调用时，esp中值如图所示



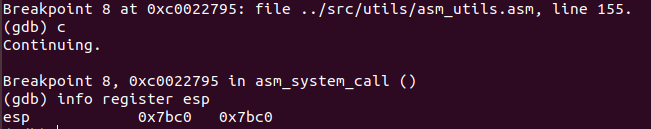
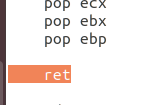
在即将退出中断函数时，esp中值为0x7b9c，与刚进入时对应



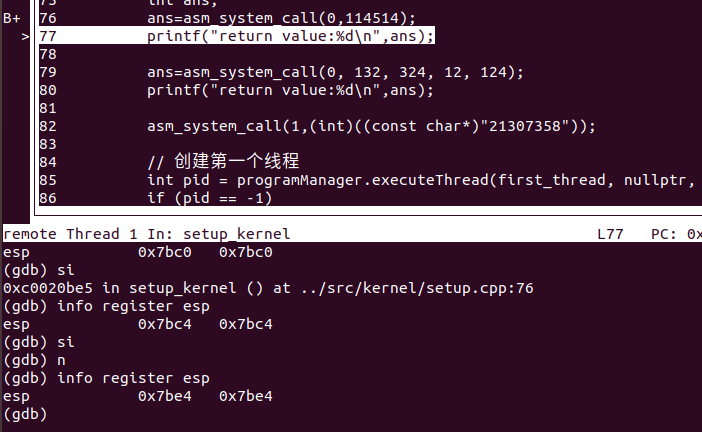
返回到asm\_system\_call时 esp中值如图所示



即将退出asm\_system\_call时 esp中值如图所示,与刚进入时对应



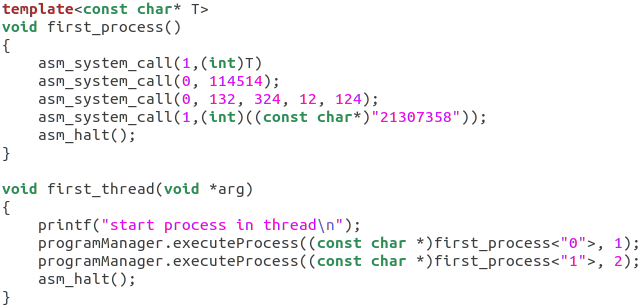
刚返回到setup.cpp时，esp中值为一开始保存的0x7bc4。在经过完整的系统调用后，变为0x7be4



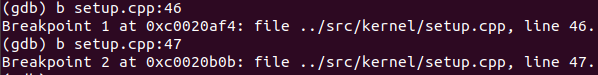
由上述gdb执行过程中esp中值可见，每次进入一个函数时，会将返回地址压入栈中，并在函数执行完毕后，会弹出返回地址，并根据返回地址恢复到调用函数处。在asm\_system\_call\_handler函数中，会发生一次esp寄存器的更改，而其他时候则按照正常的压栈和弹栈操作进行。

1. 请根据gdb来说明TSS在系统调用执行过程中的作用

代码：



设置两个预备进程的断点，其中，线程二的优先级高于线程一。



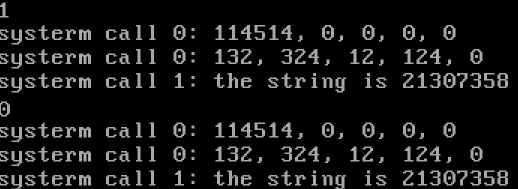
进入第一个线程前的esp



进入第二个线程前的esp，可以观察到保存的是上一个线程的地址。在执行完线程2后，会恢复线程1的状态继续执行



输出结果：



可见，TSS的作用仅限于向CPU提供进程特权级转移时的栈段选择子和栈指针。在系统调用时，从低特权级转移到高特权级时，CPU会从TTS中获取信息，并将高特权集的栈地址加载到esp中。

**Assignment 2 Fork的奥秘**

1. 请根据代码逻辑和执行结果来分析fork实现的基本思路。

**Fork含义：**

fork是一个系统调用，用于创建一个新进程。被创建的进程称为子进程，调用fork的进程被称为父进程。创建新的子进程后，两个进程将从fork的返回点开始执行。Fork的特点在于：只调用了fork一次，fork却能够返回两次。同时，在父子进程的fork返回点中，fork返回的结果是不一样的，fork返回值如下。

* 在父进程中，fork返回新创建子进程的进程ID。
* 在子进程中，fork返回0。
* 如果出现错误，fork返回一个负值。

fork的实现通过ProgramManager::fork来完成。

int ProgramManager::fork()

{

    bool status = interruptManager.getInterruptStatus();

    interruptManager.disableInterrupt();

    // 禁止内核线程调用

    PCB \*parent = this->running;

    if (!parent->pageDirectoryAddress)

    {

        interruptManager.setInterruptStatus(status);

        return -1;

    }

    // 创建子进程

    int pid = executeProcess("", 0);

    if (pid == -1)

    {

        interruptManager.setInterruptStatus(status);

        return -1;

    }

    // 初始化子进程

    PCB \*child = ListItem2PCB(this->allPrograms.back(), tagInAllList);

    bool flag = copyProcess(parent, child);

    if (!flag)

    {

        child->status = ProgramStatus::DEAD;

        interruptManager.setInterruptStatus(status);

        return -1;

    }

    interruptManager.setInterruptStatus(status);

    return pid;

}

其中，ProgramManager::fork() 功能包括：

* 关中断，禁止内核线程调用
* 调用ProgramManager::executeProcess来创建一个子进程
* 调用ProgramManager::copyProcess来复制父进程的资源到子进程中。
* 恢复中断

接下来是copyProcess的函数，这一部分主要负责的是资源的复制。

* 首先，复制父进程的0特权级栈到子进程中。在父进程的0特权级栈中，子进程真正关心的部分应该是asm\_system\_call\_handler的返回地址和执行asm\_system\_call\_handler前保护的现场。这一点是我们实现父子进程从相同的fork返回点处返回的关键。

通过将子进程的eax设置为0，使得在子进程中的fork返回值为0。

// 复制进程0级栈

    ProcessStartStack \*childpss =

        (ProcessStartStack \*)((int)child + PAGE\_SIZE - sizeof(ProcessStartStack));

    ProcessStartStack \*parentpss =

        (ProcessStartStack \*)((int)parent + PAGE\_SIZE - sizeof(ProcessStartStack));

    memcpy(parentpss, childpss, sizeof(ProcessStartStack));

    // 设置子进程的返回值为0

    childpps->eax = 0;

* 继续初始化子进程的0特权级栈。这样做是为了和asm\_switch\_thread的过程对应起来。

 // 准备执行asm\_switch\_thread的栈的内容

    child->stack = (int \*)childpss - 7;

    child->stack[0] = 0;

    child->stack[1] = 0;

    child->stack[2] = 0;

    child->stack[3] = 0;

    child->stack[4] = (int)asm\_start\_process;

    child->stack[5] = 0;         // asm\_start\_process 返回地址

    child->stack[6] = (int)childpss; //asm\_start\_process 参数

* 接着，设置子进程的PCB、复制父进程的管理虚拟地址池的bitmap到子进程的管理虚拟地址池的bitmap。

// 设置子进程的PCB

    child->status = ProgramStatus::READY;

    child->parentPid = parent->pid;

    child->priority = parent->priority;

    child->ticks = parent->ticks;

    child->ticksPassedBy = parent->ticksPassedBy;

    strcpy(parent->name, child->name);

    // 复制用户虚拟地址池

    int bitmapLength = parent->userVirtual.resources.length;

    int bitmapBytes = ceil(bitmapLength, 8);

    memcpy(parent->userVirtual.resources.bitmap, child->userVirtual.resources.bitmap, bitmapBytes);

* 然后，从内核中分配一页来作为数据复制的中转页，将父进程的页目录表复制到子进程中。处理完子进程的页目录表后，我们复制页表和物理页的数据。

// 从内核中分配一页作为中转页

    char \*buffer = (char \*)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);

    if (!buffer)

    {

        child->status = ProgramStatus::DEAD;

        return false;

    }

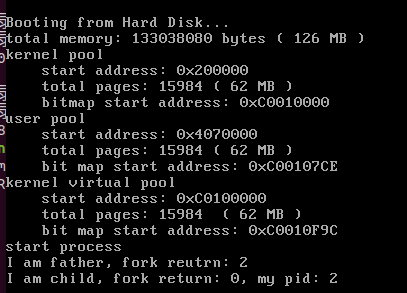
* 最后，我们归还中转页，返回子进程的pid。

// 归还从内核分配的中转页

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

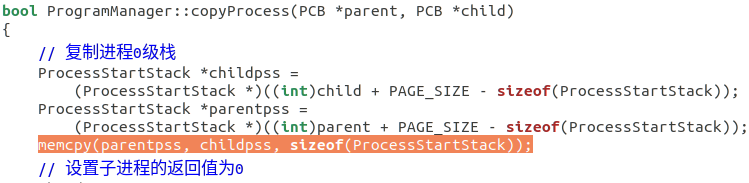
    return true;

最终执行结果如图所示：

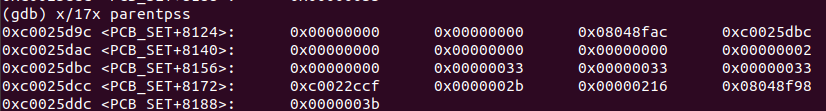


1. 跟踪子进程的执行流程一直到子进程从fork返回，根据gdb来分析子进程的变化。同时，比较父进程执行完ProgramManager::fork后的返回过程的异同。

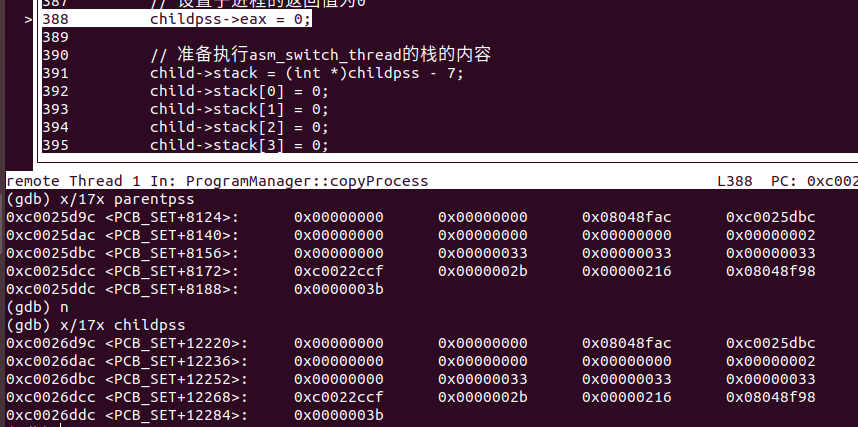
先设置好断点，查看父进程0级栈中内容



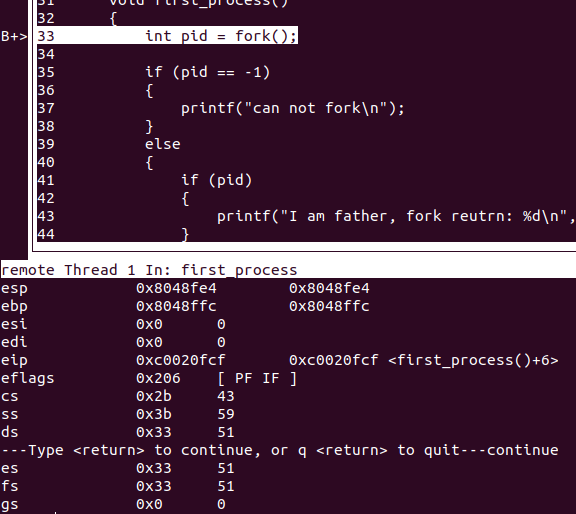
观察其中12与16号寄存器中内容，eip=0xc0022ccf，esp=0x08048f98



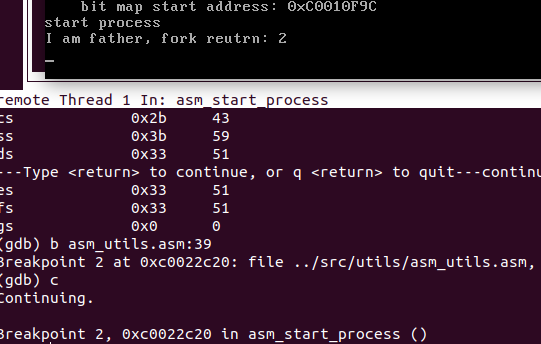
然后，执行到下一步（已完成memcpy(parentpss, childpss, sizeof(ProcessStack))。观察到childpss的12号与16号寄存器中内容与parentpss中一致



在打iret后，观察到寄存器中内容如图所示（主要看cs后的六个寄存器）



将程序执行到输出父进程中内容，然后，代码会跳至asm\_start\_process

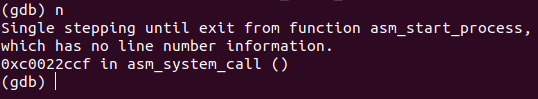


跳转情况：

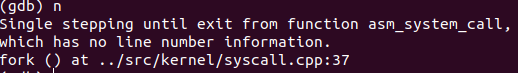
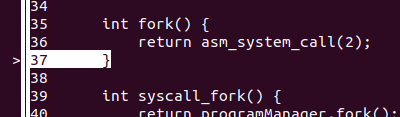


这是因为在初始化子进程0级栈时，stack[5]、stack[6]中的内容即是asm\_start\_process的返回地址和参数，在进行线程切换时，当执行asm\_start\_process函数时，由于我们之前将特权级为0的栈复制到childpss中，因此在asm\_start\_process函数执行iret指令后，会将特权级为0的栈中的eip值加载到eip寄存器中，并执行该地址对应的代码。

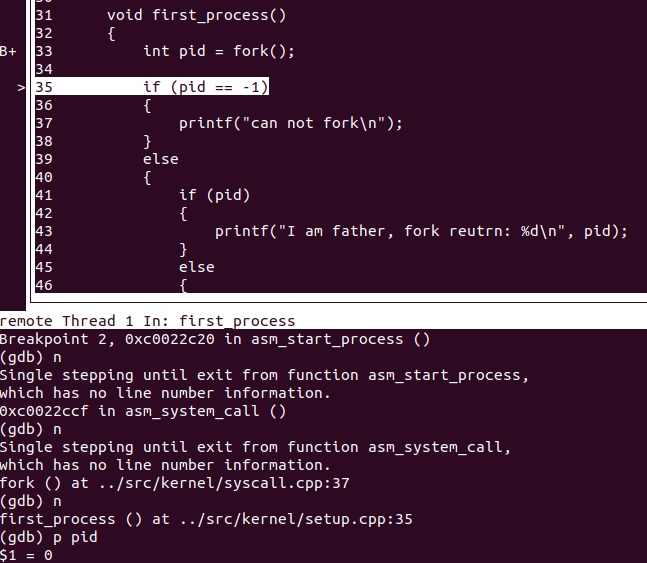
然后，在再一次执行next指令时，代码会跳到保存在0特级权的栈的eip=0xc0022ccf,也就是上文中展示出来asm\_system\_call的代码地址



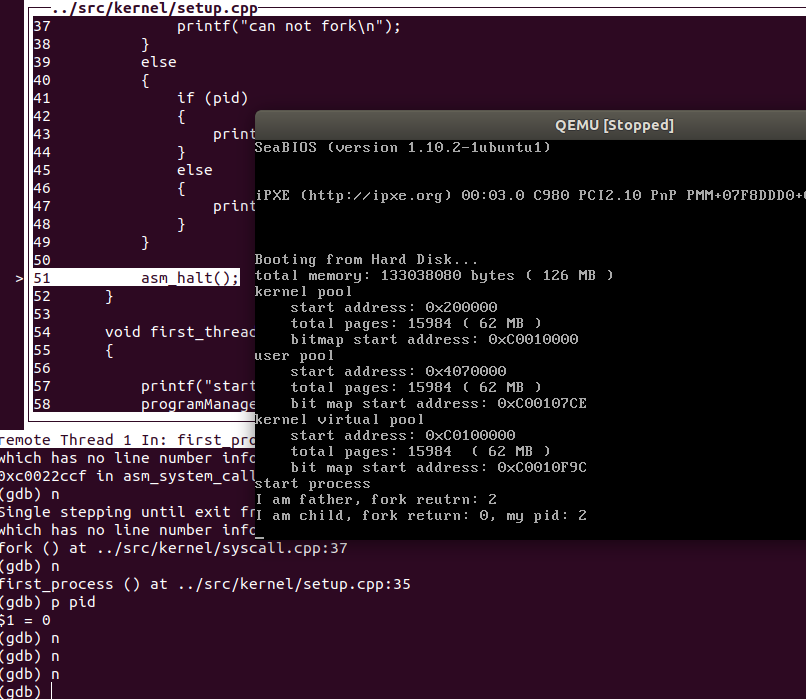
再次next，会调用syscall\_fork，并且从fork中返回



然后，再次next，会返回到最初的调用代码，first process中，可见此时pid为0，返回了子进程ID

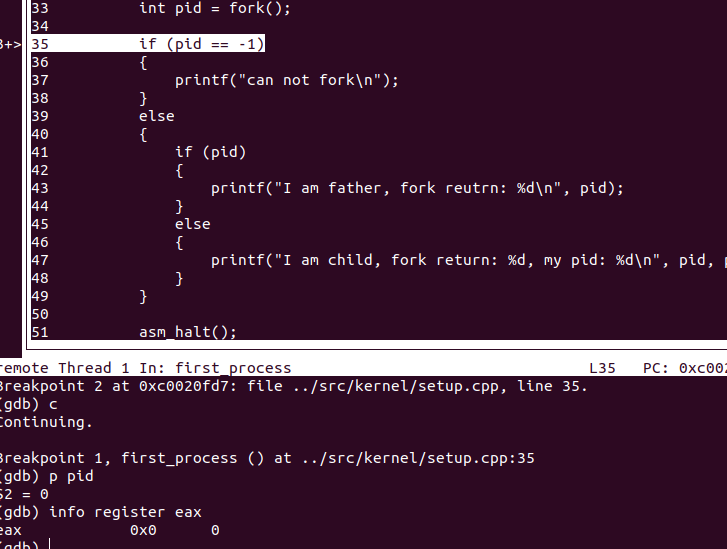


最终结束



1. 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。

在已经执行过一轮fork之后，观察到pid和eax中值如图所示：



此时pid值为0，eax的值也为0。

这是因为在ProgramManager::copyProcess中，当程序复制完进程0级栈后，便将子进程的返回值设置为0：

    // 设置子进程的返回值为0

    childpss->eax = 0;

这一点保证了fork（）的返回值为0。说明在调用fork()函数后，子进程会返回pid=0。

在ProgramManager：fork（）中，创建子进程时就已经设置了：

    // 创建子进程

    int pid = executeProcess("", 0);

这就是函数返回的pid。父进程会返回ProgramManager：fork（）的值，即子进程ID，因此父进程调用fork（）后返回的pid即为子进程ID。

**Assignment 3 哼哈二将 wait & exit**

1. 请结合代码逻辑和具体的实例来分析exit的执行过程。

exit用于进程和线程的主动结束运行，如下所示。其中ret代表返回值。

void exit(int ret) {

    asm\_system\_call(3, ret);

}

void syscall\_exit(int ret) {

    programManager.exit(ret);

}

exit的实现实际上是通过ProgramManager::exit来完成的，总的来看，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。

实际上可以看成是copyProcess的部分逆过程，如下所示。

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;

    int \*pageDir, \*page;

    int 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 = (int \*)program->pageDirectoryAddress;

        for (int i = 0; i < 768; ++i)

        {

            if (!(pageDir[i] & 0x1))

            {

                continue;

            }

            page = (int \*)(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,

                                   (int)program->userVirtual.resources.bitmap,

                                   bitmapPages);

    }

    // 第三步，立即执行线程/进程调度。

    schedule();

}

6~8行：将当前正在运行的进程的PCB状态设置为dead，并且放入返回值。

15行：定义pageDir指向的目标地址：页目录表

16~37行：释放每一个页表指向的物理页

26行：定义page为对应目录表项指向的页表物理地址

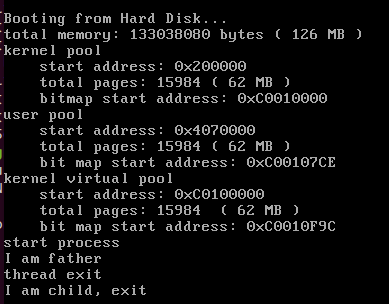
32~33行：找到page对应的物理页地址，然后释放

35行：释放指向页目录表的物理页。

39行：释放bitmap

44行：执行调度函数schedule（），这部分结束。

执行实例后，结果如图所示：



1. 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。

与教程中主动调用exit的情况相反，此时调用exit的方式是像线程一样在函数结束后主动调用exit来结束运行。

exit函数的地址和参数会被放入进程的特权级为3的栈顶部，在进程函数执行完毕后，程序会自动跳转到exit函数。在load\_process函数中，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。这样，当进程函数执行完毕后，程序会自动跳转到exit函数，以结束该进程，即隐式调用。

fork函数会首先执行父进程，然后再执行子进程。因此，最后执行的子进程在执行完毕后会通过interruptStack栈调用exit(0)，确保进程退出时一定会调用exit函数，因此此时exit函数的返回值为0。

1. 请结合代码逻辑和具体的实例来分析wait的执行过程。

进程的状态被标记为DEAD后未被清除，而是在等待其他进程来回收进程的PCB。进程的PCB是通过父进程来回收的。父进程会通过wait系统调用来等待其子进程执行完成并回收子进程。其定义如下。

int wait(int \*retval) {

    return asm\_system\_call(4, (int)retval);

}

int syscall\_wait(int \*retval) {

    return programManager.wait(retval);

}

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。

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();

            }

        }

    }

}

第15-28行，我们试图在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。

第30-41行，我们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，最后返回子进程的pid。releasePCB实现如下。

void ProgramManager::releasePCB(PCB \*program)

{

    int index = ((int)program - (int)PCB\_SET) / PCB\_SIZE;

    PCB\_SET\_STATUS[index] = false;

    this->allPrograms.erase(&(program->tagInAllList));

}

第44-49行，我们并没有找到子进程，因此返回-1。

第50-54行，存在子进程但子进程的状态不是DEAD，因此我们执行调度。注意到该进程的主体是一个死循环，因此当进程的第53语句返回后，wait并不会返回。而是再一次重复上面的步骤，尝试回收子进程。也就是前面所说的，当父进程调用wait后，如果存在子进程但子进程的状态不是DEAD，则父进程会被阻塞，即wait不会返回直到子进程结束。

Schedule（）修改如下：

void ProgramManager::schedule()

{

...

    if (running->status == ProgramStatus::RUNNING)

    {

        running->status = ProgramStatus::READY;

        running->ticks = running->priority \* 10;

        readyPrograms.push\_back(&(running->tagInGeneralList));

    }

    else if (running->status == ProgramStatus::DEAD)

    {

        // 回收线程，子进程留到父进程回收

        if(!running->pageDirectoryAddress) {

            releasePCB(running);

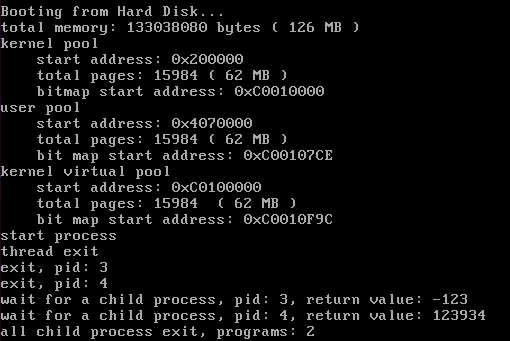
        }

    }

...

}

最终执行实例后结果如下：



1. 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。

在first\_process中，我们设置一个循环判断是否存在状态不为dead的子进程。若存在，就继续循环，否则退出：

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);

        }

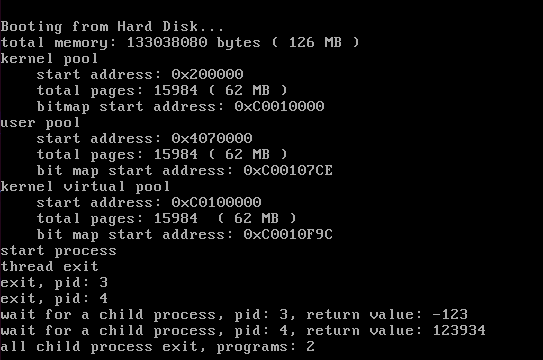
    }

这时，只要存在状态不为DEAD的子进程，程序就会一直运行，因此，在所有子进程状态变为dead前，父进程不会返回，就可以修正父进程先于子进程退出的情况。

只有在子进程状态都变为dead时，所有子进程都会经过循环中的wait(&retval)被回收，因此就有了wait(&retval)) == -1，跳出while循环。

结尾打印的all child process exit即证明所有子进程都被回收，这就可以有效回收僵尸进程。

运行结果：



1. **总结**

通过本次实验我了解了系统调用、fork、wait和exit的具体实验过程，并通过gdb跟踪系统调用的执行过程以及fork时父进程和子进程的衔接关系进一步加深了对这部分的理解，并且还通过分析wait和exit的代码了解了主动调用和隐式调用分别要怎么实现，以及僵尸进程的回收方法，收获颇多。