

## 个人信息

---

【院系】计算机学院

【专业】计算机科学与技术

【学号】20302124

【姓名】庄阳阳

## 实验题目

---

从内核态到用户态

## 实验目的

---

1. 理解区分内核态和用户态的必要性。
2. 编写一个系统调用，并分析系统调用前后栈的变化。
3. 掌握用户态与内核态之间切换的过程。
4. 掌握进程创建的过程。
5. 分析fork/exit/wait指令执行过程。
6. 学习如何通过分页机制隔离进程间的地址空间。

## 实验要求

---

1. 编写一个系统调用。
2. 掌握用户态与内核态之间切换的过程。
3. 实现fork/exit/wait指令。
4. 实现进程之间的隔离。
5. 撰写实验报告。

## 实验方案

---

### Assignment 1 系统调用

---

#### 存储方式--寄存器 VS 栈

栈的问题：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。

`asm_system_call` 是通过汇编来实现的，如下所示。

```
asm_system_call:
    push ebp
    mov ebp, esp

    push ebx
    push ecx
    push edx
    push esi
    push edi ; 系统调用的五个参数
    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 ;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。

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

    ret
```

## 系统调用实现

创建一个管理系统调用的类 `SystemService`，如下所示，代码放在 `syscall.h` 中。

```
#ifndef SYSCALL_H
#define SYSCALL_H

#include "os_constant.h"

class SystemService
{
public:
```

```

   SystemService();
    void initialize();
    // 设置系统调用, index=系统调用号, function=处理第index个系统调用函数的地址
    bool setSystemCall(int index, int function);
};

// 第0个系统调用
int syscall_0(int first, int second, int third, int forth, int fifth);

#endif

```

在用户程序使用系统调用之前，我们首先要对系统调用表进行初始化。其次，由于我们的系统调用是通过0x80号中断来完成的，我们加入0x80中断对应的中断描述符。

```

void SystemService::initialize()
{
    memset((char *)system_call_table, 0, sizeof(int) * MAX_SYSTEM_CALL);
    // 代码段的选择子默认是DPL=0的平坦模式代码段选择子,
    // 但中断描述符的DPL=3, 否则用户态程序无法使用该中断描述符
    interruptManager.setInterruptDescriptor(0x80, (uint32)asm_system_call_handler, 3);
}

```

我们现在回到 `SystemService` 中，加入设置系统调用的函数。

```

bool SystemService::setSystemCall(int index, int function)
{
    system_call_table[index] = function;
    return true;
}

```

其中，`function` 是第 `index` 个系统调用的处理函数的地址。

最后设置系统调用号为0的系统调用处理函数。

## 初始化TSS和用户段描述符

我们首先向 `ProgramManager` 中加入存储3个代码段、数据段和栈段描述符的变量。

```

class ProgramManager
{
public:
    List allPrograms;           // 所有状态的线程/进程的队列
    List readyPrograms;        // 处于ready(就绪态)的线程/进程的队列
    PCB *running;              // 当前执行的线程
    int USER_CODE_SELECTOR;    // 用户代码段选择子
    int USER_DATA_SELECTOR;    // 用户数据段选择子
    int USER_STACK_SELECTOR;   // 用户栈段选择子
    ...
}

```

由于进程的运行环境需要用到TSS、特权级3下的平坦模式代码段和数据段描述符，我们现在来初始化这些内容，如下所示。

```

void ProgramManager::initialize()
{
    allPrograms.initialize();
    readyPrograms.initialize();
    running = nullptr;

    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        PCB_SET_STATUS[i] = false;
    }

    // 初始化用户代码段、数据段和栈段
    int selector;

    selector = asm_add_global_descriptor(USER_CODE_LOW, USER_CODE_HIGH);
    // USER_CODE_LOW 0x0000ffff    USER_CODE_HIGH 0x00cff800
    USER_CODE_SELECTOR = (selector << 3) | 0x3;

    selector = asm_add_global_descriptor(USER_DATA_LOW, USER_DATA_HIGH);
    USER_DATA_SELECTOR = (selector << 3) | 0x3;
    // USER_DATA_LOW 0x0000ffff    USER_DATA_HIGH 0x00cff200

    selector = asm_add_global_descriptor(USER_STACK_LOW, USER_STACK_HIGH);
    USER_STACK_SELECTOR = (selector << 3) | 0x3;
    // USER_STACK_LOW 0x00000000    USER_STACK_HIGH 0x0040f600
    initializeTSS();
}

```

将用户代码段描述符，数据段描述符和栈段描述符送入GDT（第三个LAB曾经有具体讲过）。这3个描述符和之前的描述符不同之处在于DPL。之前的描述符的DPL为0，而这几个描述符的DPL为3。

接下创建一个TSS的结构体。

```

#ifndef TSS_H
#define TSS_H

```

```

struct TSS
{
public:
    int backlink;
    int esp0;
    int ss0;
    int esp1;
    int ss1;
    int esp2;
    int ss2;
    int cr3;
    int eip;
    int eflags;
    int eax;
    int ecx;
    int edx;
    int ebx;
    int esp;
    int ebp;
    int esi;
    int edi;
    int es;
    int cs;
    int ss;
    int ds;
    int fs;
    int gs;
    int ldt;
    int trace;
    int ioMap;
};
#endif

```

注意，TSS的内容必须如此安排，因为CPU规定了TSS中的内容。如果不做上述安排，当CPU加载TSS时就会加载到错误的地址。

TSS的初始化如下。

```

void ProgramManager::initializeTSS()
{
    int size = sizeof(TSS);
    int address = (int)&tss; // 地址

    memset((char *)address, 0, size); // 赋0初始化

    tss.ss0 = STACK_SELECTOR; // 内核态堆栈段选择子

    int low, high, limit;
}

```

```

    limit = size - 1; // 段界限长度
    low = (address << 16) | (limit & 0xff);
    // DPL = 0
    high = (address & 0xff000000) | ((address & 0x00ff0000) >> 16) | ((limit & 0xff00) << 16) |
    0x00008900;

    int selector = asm_add_global_descriptor(low, high); //将TSS送入GDT
    // RPL = 0
    asm_ltr(selector << 3);
    tss.ioMap = address + size;
}

```

由于TSS的作用仅限于提供0特权级下的栈指针和栈段选择子，因此我们关心 `TSS::ss0` 和 `TSS::esp0`。但在这里我们只对 `TSS::ss0` 进行复制，`TSS::esp0` 会在进程切换时更新。

其中，`STACK_SELECTOR` 是特权级0下的栈段选择子，也就是我们在bootloader中放入了SS的选择子。

## 进程的创建

进程和线程的区别在于进程有自己的虚拟地址空间和相应的分页机制。

进程的创建分为3步。

- 创建进程的PCB。
- 初始化进程的页目录表。
- 初始化进程的虚拟地址池。

大部分与线程创建是类似的，就不再赘述。

不同的地方：

需要定义一个类 `ProgramStartStack` 来表示启动进程之前栈放入的内容。

```

#ifndef PROCESS_H
#define PROCESS_H

struct ProcessStartStack
{
    int edi;
    int esi;
    int ebp;
    int esp_dummy;
    int ebx;
    int edx;
    int ecx;
    int eax;

    int gs;
    int fs;
    int es;
    int ds;

    int eip;
}

```

```

    int cs;
    int eflags;
    int esp;
    int ss;
};

#endif

```

还需要中断返回来启动进程：

```

; void asm_start_process(int stack);
asm_start_process:
    ; jmp $
    mov eax, dword[esp+4]
    mov esp, eax
    popad
    pop gs;
    pop fs;
    pop es;
    pop ds;

    iret

```

我们将 `ProcessStartStack` 的起始地址送入了 `esp`，然后通过一系列的 `pop` 指令和 `iret` 语句更新寄存器，最后中断返回后，特权级3的选择子被放入到段寄存器中，代码跳转到进程的起始处执行。

用户进程和内核线程使用的是不同的代码段、数据段和栈段选择子。我们之前在设计基于时钟中断的调度时，并没有对 `ds, fs, es, gs` 寄存器进行保护和恢复。所以，当我们在内核线程和用户进程之前切换的时候，上面提到的4个寄存器可能没有被切换到用户进程或内核线程使用的 `ds, fs, es, gs`。但对于 `cs, ss`，CPU会自动保护和恢复。为了解决这个问题，我们修改 `asm_time_interrupt_handler`，加入保护和恢复 `ds, fs, es, gs` 的代码。

```

asm_time_interrupt_handler:
    pushad
    push ds
    push es
    push fs
    push gs

    ; 发送EOI消息，否则下一次中断不发生
    mov al, 0x20
    out 0x20, al
    out 0xa0, al

    call c_time_interrupt_handler

    pop gs
    pop fs
    pop es
    pop ds

    popad

```

```
iret
```

最后做一些简单的修改就可以调用进程了。

## print

简单写个系统调用就可以了。

```
void syscall_print(const char*s)
{
    printf(s);
}

void first_process()
{
    asm_system_call(1,int("hello world\n"));
    asm_halt();
}
```

## findProgramByPid

简单的利用线性查找即可

```
PCB* ProgramManager::findProgramByPid(int pid)
{
    return (PCB*)((int)PCB_SET + pid * PAGE_SIZE);
}
```

## Assignment 2 Fork的奥秘

进入内核态后，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，因为内核线程并没有设置
    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); // 复制父进程的资源到子进程中。

    if (!flag)
    {
        child->status = ProgramStatus::DEAD;
        interruptManager.setInterruptStatus(status);
        return -1;
    }

    interruptManager.setInterruptStatus(status);
    return pid;
}

```

## 资源的复制

1. 复制父进程的0特权级栈到子进程中。为此，需要找到父进程的0特权级栈的地址。

当我们调用fork系统调用后，我们会从用户态进入内核态。

需要复制的父进程0特权级栈的起始地址只是进程的PCB所在的页的顶部减去一个 `ProgramStartStack` 的大小，长度便是一个 `ProgramStartStack` 的大小。这就是 `copyProcess` 开头的语句。我们实际上就是把在中断的那一刻保存的寄存器的内容复制到子进程的0特权级栈中。

```

// 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
childpss->eax = 0;

```

2. 初始化子进程的0特权级栈。

```

// 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

```

这样做是为了和 `asm_switch_thread` 的过程对应起来。当子进程被调度执行时，子进程能够从 `asm_switch_thread` 跳转到 `asm_start_proces` 处执行。

### 3. 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，我们从内核中分配一页来作为数据复制的中转页。

```
// 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;
}
```

分页机制实现了地址隔离，父进程就无法将数据复制到具有相同虚拟地址的子进程中。因此，我们需要借助于内核空间的中转页。我们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。

### 4. 将父进程的页目录表复制到子进程中。

```
// 子进程页目录表物理地址
int childPageDirPaddr = memoryManager.vaddr2paddr(child->pageDirectoryAddress);
// 父进程页目录表物理地址
int parentPageDirPaddr = memoryManager.vaddr2paddr(parent->pageDirectoryAddress);
// 子进程页目录表指针(虚拟地址)
int *childPageDir = (int *)child->pageDirectoryAddress;
// 父进程页目录表指针(虚拟地址)
int *parentPageDir = (int *)parent->pageDirectoryAddress;

// 子进程页目录表初始化
memset((void *)child->pageDirectoryAddress, 0, 768 * 4);

// 复制页目录表
for (int i = 0; i < 768; ++i)
{
    // 无对应页表
```

```

    if (!(parentPageDir[i] & 0x1))
    {
        continue;
    }

    // 从用户物理地址池中分配一页，作为子进程的页目录项指向的页表
    int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    if (!paddr)
    {
        child->status = ProgramStatus::DEAD;
        return false;
    }
    // 页目录项
    int pde = parentPageDir[i];
    // 构造页表的起始虚拟地址
    int *pageTableVaddr = (int *)(0xffc00000 + (i << 12));

    asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

    childPageDir[i] = (pde & 0x00000fff) | paddr; // 设置子进程页目录表的页目录项
    memset(pageTableVaddr, 0, PAGE_SIZE); // 初始化页目录项指向的页表

    asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
}

```

##### 5. 复制页表和物理页的数据。

```

// 复制页表和物理页
for (int i = 0; i < 768; ++i)
{
    // 无对应页表
    if (!(parentPageDir[i] & 0x1))
    {
        continue;
    }

    // 计算页表的虚拟地址
    int *pageTableVaddr = (int *)(0xffc00000 + (i << 12));

    // 复制物理页
    for (int j = 0; j < 1024; ++j)
    {
        // 无对应物理页
        if (!(pageTableVaddr[j] & 0x1))
        {
            continue;
        }

        // 从用户物理地址池中分配一页，作为子进程的页表项指向的物理页
        int paddr = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);

        if (!paddr)

```

```

    {
        child->status = ProgramStatus::DEAD;
        return false;
    }

    // 构造物理页的起始虚拟地址
    void *pageVaddr = (void *)((i << 22) + (j << 12));
    // 页表项
    int pte = pageTableVaddr[j];
    // 复制出父进程物理页的内容到中转页
    memcpy(pageVaddr, buffer, PAGE_SIZE);

    asm_update_cr3(childPageDirPaddr); // 进入子进程虚拟地址空间

    pageTableVaddr[j] = (pte & 0x00000fff) | paddr;
    // 从中转页中复制到子进程的物理页
    memcpy(buffer, pageVaddr, PAGE_SIZE);

    asm_update_cr3(parentPageDirPaddr); // 回到父进程虚拟地址空间
}
}

```

到这里，fork的核心框架便实现了。

## Assignment 3 哼哈二将 wait & exit

### exit

总的来看，exit的实现主要分为三步。

1. 标记PCB状态为 `DEAD` 并放入返回值。

```

// 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;

```

2. 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。

```

// 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);
}

```

### 3. 立即执行线程/进程调度。

```

// 第三步，立即执行线程/进程调度。
schedule();

```

## wait

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。

wait的实现实际上是由 `ProgramManager` 来完成的。

```

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, 找到一个状态为DEAD的子进程
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) // 不为nullptr
    {
        *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; // 没有找到子进程, 返回-1
    }
    else // 存在子进程, 但子进程的状态不是DEAD
    {
        interruptManager.setInterruptStatus(interrupt);
        schedule();
    }
}
}
}

```

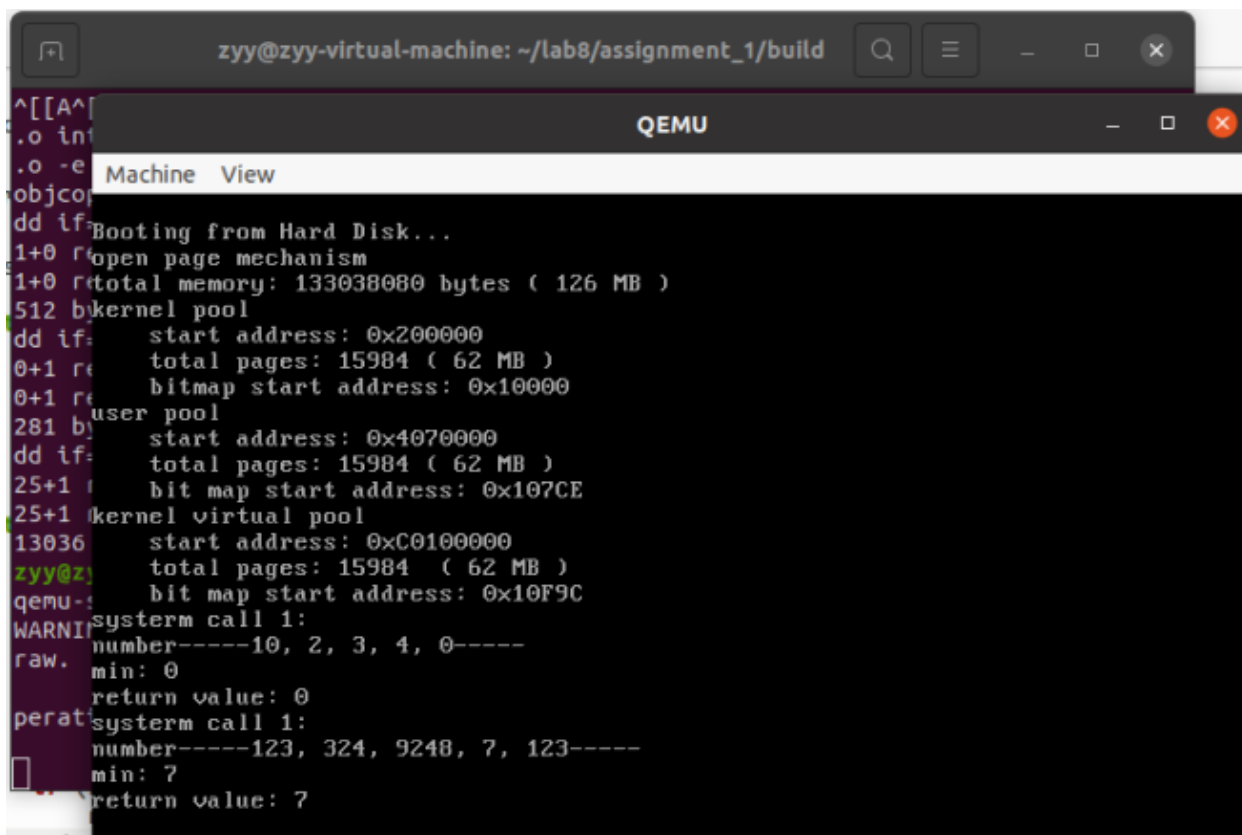
最后存在子进程但子进程的状态不是 `DEAD`，因此我们执行调度。注意到该进程的主体是一个死循环，因此当进程的第53语句返回后，`wait`并不会返回。而是再一次重复上面的步骤，尝试回收子进程。也就是前面所说的，当父进程调用`wait`后，如果存在子进程但子进程的状态不是 `DEAD`，则父进程会被阻塞，即`wait`不会返回直到子进程结束。

## 实验过程

### Assignment 1 系统调用

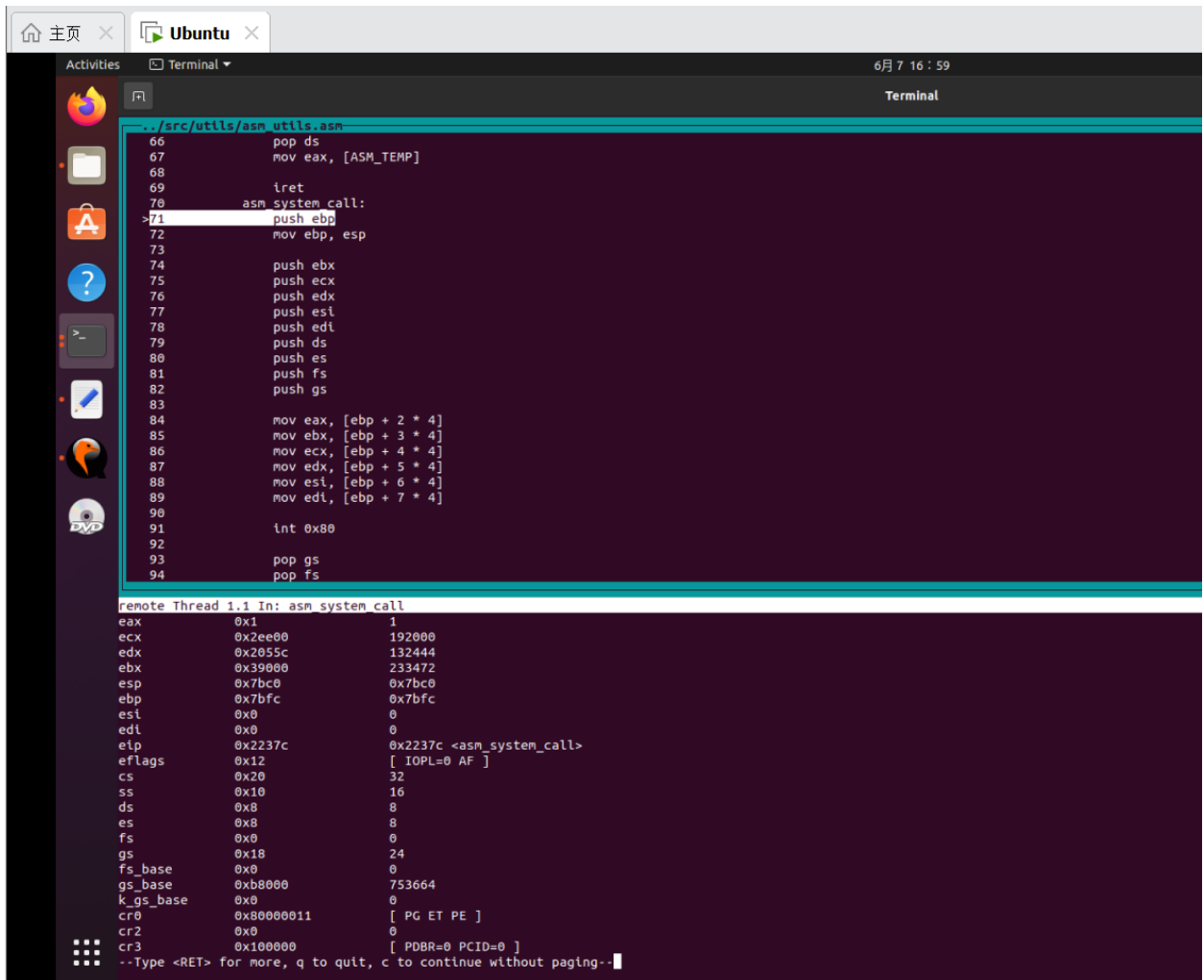
编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

- 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。

A screenshot of a QEMU virtual machine window. The title bar shows the user 'zyy' on a 'zyy-virtual-machine' with the current directory '~/lab8/assignment\_1/build'. The terminal window has a title bar 'QEMU' and a 'Machine View' tab. The terminal output shows the booting process from a hard disk, memory allocation for kernel and user pools, and two system calls. The first system call (call 1) has arguments 10, 2, 3, 4, 0 and returns 0. The second system call (call 1) has arguments 123, 324, 9248, 7, 123 and returns 7. There are some warnings and raw output visible in the terminal.

```
^[[A^
.o in
.o -e
objco
dd if=
dd if=
1+0 r
open page mechanism
1+0 r
total memory: 133038080 bytes ( 126 MB )
512 b
kernel pool
dd if=
start address: 0x200000
0+1 r
total pages: 15984 ( 62 MB )
0+1 r
bitmap start address: 0x10000
0+1 r
user pool
281 b
start address: 0x4070000
dd if=
total pages: 15984 ( 62 MB )
25+1 r
bit map start address: 0x107CE
25+1 r
kernel virtual pool
13036
start address: 0xC0100000
zyy@z
total pages: 15984 ( 62 MB )
qemu-
bit map start address: 0x10F9C
system call 1:
WARNI
number----10, 2, 3, 4, 0----
raw.
min: 0
return value: 0
perat
system call 1:
number----123, 324, 9248, 7, 123----
min: 7
return value: 7
```

- 请根据gdb来分析执行系统调用后的栈的变化情况。





```
Activities Terminal 6月 7 17:0
Terminal
../src/utls/asm_utils.asm
66      pop ds
67      mov eax, [ASM_TEMP]
68
69      iret
70      asm_system_call:
71      push ebp
72      mov ebp, esp
73
74      push ebx
75      push ecx
76      push edx
77      push esi
78      push edi
79      push ds
80      push es
81      push fs
82      push gs
83
84      mov eax, [ebp + 2 * 4]
85      mov ebx, [ebp + 3 * 4]
86      mov ecx, [ebp + 4 * 4]
87      mov edx, [ebp + 5 * 4]
88      mov esi, [ebp + 6 * 4]
89      mov edi, [ebp + 7 * 4]
90
91      int 0x80
92
93      pop gs
94      pop fs

remote Thread 1.1 In: asm system call
eax      0x1      1
ecx      0x2      2
edx      0x3      3
ebx      0xa      10
esp      0x7b98   0x7b98
ebp      0x7bbc   0x7bbc
esi      0x4      4
edi      0x0      0
eip      0x2239c   0x2239c <asm_system_call+32>
eflags   0x12     [ IOPL=0 AF ]
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      0x100000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

传入的参数分别为1,10, 2, 3, 4, 根据上面两张图可以看出, eax本身一直为1, 没有变化, 而根据寻参规则, ebx, ecx,edx,esi,edx分别为10, 2, 3, 4, 0 (未传参, 值为默认参数0), ebp, esp也随之发生变化

- 请根据gdb来说明TSS在系统调用执行过程中的作用。

```
Activities Terminal 6月 7 23:13
Terminal
../src/utls/asm_utils.asm
129     asm_system_call:
130         push ebp
131         mov ebp, esp
132
133         push ebx
134         push ecx
135         push edx
136         push esi
137         push edi
138
139         mov eax, [ebp + 2 * 4]
140         mov ebx, [ebp + 3 * 4]
141         mov ecx, [ebp + 4 * 4]
142         mov edx, [ebp + 5 * 4]
143         mov esi, [ebp + 6 * 4]
144         mov edi, [ebp + 7 * 4]
145
146     > int 0x80
147
148         pop edi
149         pop esi
150         pop edx
151         pop ecx
152         pop ebx
153         pop ebp
154
155         ret
156
157     ; void asm_init_page_reg(int *directory);

remote Thread 1.1 In: asm_system_call
eax      0x0      0
ecx      0x144    324
edx      0xc      12
ebx      0x84     132
esp      0x8048fb8 0x8048fb8
ebp      0x8048fcc 0x8048fcc
esi      0x7c     124
edi      0x0      0
eip      0xc002284d 0xc002284d <asm_system_call+26>
eflags   0x212    [ IOPL=0 IF AF ]
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      0x200000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

在调用 `0x80` 中断之前处于用户态，根据GDB调试可知 `ss=0x3b`，`esp=0x8048fb8`，`cs=0x2b`。

```
Activities Terminal 6月 7 23 : 09
Terminal
./src/utls/asm_utils.asm
76     shr eax, 3
77
78     add word[ASM_GDTR], 8
79     lgdt [ASM_GDTR]
80
81     pop esi
82     pop ebx
83     pop ebp
84
85     ret
86     ; int asm_system_call_handler();
87     asm_system_call_handler:
88     push ds
>89     push es
90     push fs
91     push gs
92     pushad
93
94     push eax
95
96     ; ^&^ &^ $^ $^ ss^$^ (^ %^ %^ (^
97
98     mov eax, DATA_SELECTOR
99     mov ds, eax
100    mov es, eax
101
102    mov eax, VIDEO_SELECTOR
103    mov gs, eax
104

remote Thread 1.1 In: asm_system_call_handler
eax      0x0      0
ecx      0x144    324
edx      0xc     12
ebx      0x84    132
esp      0xc0025808 0xc0025808 <PCB_SET+8168>
ebp      0x8048fcc 0x8048fcc
esi      0x7c    124
edi      0x0     0
eip      0xc00227f8 0xc00227f8 <asm_system_call_handler+1>
eflags   0x12     [ IOPL=0 AF ]
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      0x200000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

在调用 `0x80` 中断之后处于内核态，根据GDB调试可知 `ss=0x10`，`esp=0xc0025808`，`cs=0x20`，`CODE_SELECTOR 0x20`，`STACK_SELECTOR 0x10`，可以看出用户态转入到了内核态，堆栈也切换到了0特权级栈。TSS的作用便是在从低特权级向高特权级转变的过程中提供0特权级栈所在段选择子和段内偏移，TSS在进程切换时起着重要的作用，通过它保存CPU中各寄存器的值，实现进程的挂起和恢复。

The screenshot shows a GDB terminal window with the following content:

```
..src/utls/asm utls.asm
135      push edx
136      push esi
137      push edi
138
139      mov eax, [ebp + 2 * 4]
140      mov ebx, [ebp + 3 * 4]
141      mov ecx, [ebp + 4 * 4]
142      mov edx, [ebp + 5 * 4]
143      mov esi, [ebp + 6 * 4]
144      mov edi, [ebp + 7 * 4]
145
146      int 0x80
147
148      pop edi
149      pop esi
150      pop edx
151      pop ecx
152      pop ebx
153      pop ebp
154
155      ret
156
157      ; void asm_init_page_reg(int *directory);
158      asm_init_page_reg:
159      push ebp
160      mov ebp, esp
161
162      push eax
163
remote Thread 1.1 In: asm system call
eax      0x250      592
ecx      0x144      324
edx      0xc        12
ebx      0x84       132
esp      0x8048fb8  0x8048fb8
ebp      0x8048fcc  0x8048fcc
esi      0x7c       124
edi      0x0        0
eip      0xc002284f 0xc002284f <asm_system_call+28>
eflags   0x212      [ IOPL=0 IF AF ]
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      0x200000   [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

返回中断后，我们可以看见现在已经处于用户态，根据GDB调试可知 `ss=0x3b`，`esp=0x8048fb8`，`cs=0x2b`，已经恢复原来的寄存器值

## printf 验证

```
zyy@zyy-virtual-machine: ~/lab8/assignment_1/printf/build
qemu-system-x86_64 -smp 1 -m 128M -hda ./disk.img -nographic
WARNING: Please do not use raw disk images
raw.

perati 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
hello world
hello world
hello world
```

## Assignment 2 Fork的奥秘

- 请根据代码逻辑和执行结果来分析fork实现的基本思路。

已经在实验方案中详细叙述。运行结果如下：

```
zyy@zyy-virtual-machine: ~/lab8/assignment_2/fork/build
qemu-system-x86_64 -smp 1 -m 128M -hda ./disk.img -nographic
WARNING: Please do not use raw disk images
raw.

perati 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, fork reutrnr: 2
I am child, fork return: 0, my pid: 2
```

- 父进程的执行流程

The screenshot shows a terminal window with the following content:

```

.. /src/utls/asm utls.asm
123     pop fs
124     pop es
125     pop ds
126     mov eax, [ASM_TEMP]
127
128     iret
129     asm_system_call:
130     push ebp
131     mov ebp, esp
132
133     push ebx
134     push ecx
135     push edx
136     push esi
137     push edi
138
139     mov eax, [ebp + 2 * 4]
140     mov ebx, [ebp + 3 * 4]
141     mov ecx, [ebp + 4 * 4]
142     mov edx, [ebp + 5 * 4]
143     mov esi, [ebp + 6 * 4]
144     mov edi, [ebp + 7 * 4]
145
->146     int 0x80
147
148     pop edi
149     pop esi
150     pop edx
151     pop ecx

remote Thread 1.1 In: asm system call
eax      0x2          2
ecx      0x0          0
edx      0x0          0
ebx      0x0          0
esp      0x8048f98    0x8048f98
ebp      0x8048fac    0x8048fac
esi      0x0          0
edi      0x0          0
elp      0xc0022e6d    0xc0022e6d <asm_system_call+26>
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      0x200000      [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--

```

进入中断前的寄存器/栈值

```
Activities Terminal 6月 8 10:2
Terminal
../src/utls/asm_utils.asm
83      pop ebp
84
85      ret
86      ; int asm_system_call_handler();
87      asm system_call_handler:
B+>88      push ds
89      push es
90      push fs
91      push gs
92      pushad
93
94      push eax
95
96      ; ^&^ &^ $^ $^ ss^$^ (^ %^ %^ (^
97
98      mov eax, DATA_SELECTOR
99      mov ds, eax
100     mov es, eax
101
102     mov eax, VIDEO_SELECTOR
103     mov gs, eax
104
105     pop eax
106
107     ; ^%^ &^ %^ &^
108     push edi
109     push esi
110     push edx
111     push ecx

remote Thread 1.1 In: asm system call handler
eax      0x1      1
ecx      0x0      0
edx      0x0      0
ebx      0x7b3b   31547
esp      0x7a6c   0x7a6c
ebp      0x7a8c   0x7a8c
esi      0x0      0
edi      0x0      0
eip      0xc0022e17 0xc0022e17 <asm_system_call_handler>
eflags   0x16     [ IOPL=0 AF 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      0x100000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

发生中断

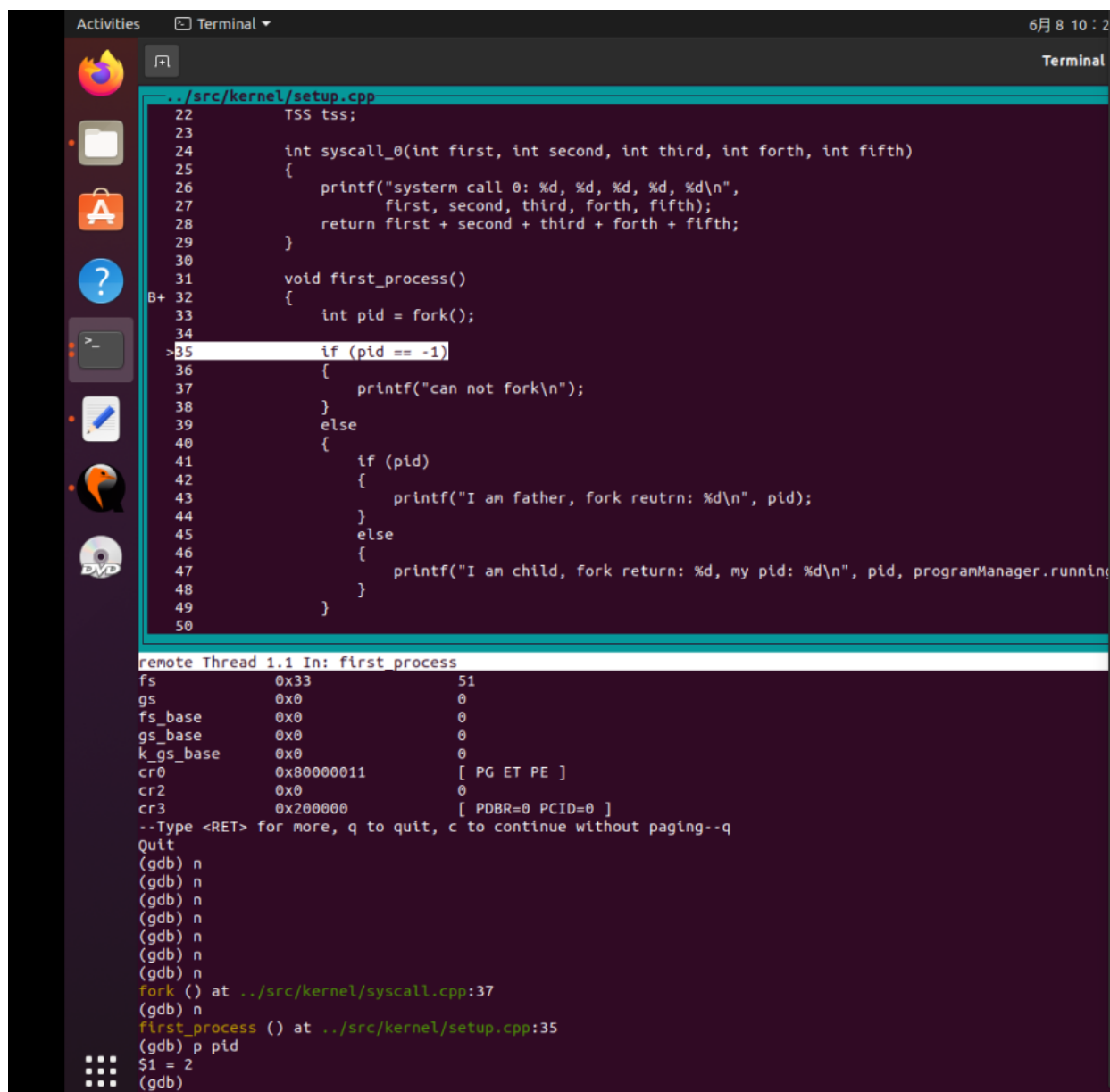
```
Activities Terminal 6月 8 1
Term

../src/utils/asm_utils.asm
129     asm_system_call:
130         push ebp
131         mov ebp, esp
132
133         push ebx
134         push ecx
135         push edx
136         push esi
137         push edi
138
139         mov eax, [ebp + 2 * 4]
140         mov ebx, [ebp + 3 * 4]
141         mov ecx, [ebp + 4 * 4]
142         mov edx, [ebp + 5 * 4]
143         mov esi, [ebp + 6 * 4]
144         mov edi, [ebp + 7 * 4]
145
146         int 0x80
147
>148     pop edi
149         pop esi
150         pop edx
151         pop ecx
152         pop ebx
153         pop ebp
154
155         ret
156
157     ; void asm_init_page_reg(int *directory);

remote Thread 1.1 In: asm_system_call
eax      0x2      2
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      0x200000 [ PDBR=0 PCID=0 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

还原中断发生之前父进程通用寄存器的值





The screenshot shows a terminal window with a dark background. The top bar indicates 'Activities', 'Terminal', and the date '6月 8 10:2'. The left sidebar contains icons for various applications. The main area displays C code from `../src/kernel/setup.cpp`. The code defines a `syscall_0` function and a `first_process` function. The `first_process` function calls `fork()` and prints messages for both parent and child processes. The code is being edited in a text editor, with line 35 highlighted. Below the code, the GDB debugger output is shown, displaying register values and the execution flow. The output shows the `fork()` call at line 37 and the `first_process()` call at line 35. The parent process returns `pid = 2`.

```
../src/kernel/setup.cpp
22     TSS tss;
23
24     int syscall_0(int first, int second, int third, int forth, int fifth)
25     {
26         printf("system call 0: %d, %d, %d, %d, %d\n",
27             first, second, third, forth, fifth);
28         return first + second + third + forth + fifth;
29     }
30
31     void first_process()
32     {
33         int pid = fork();
34
35         if (pid == -1)
36         {
37             printf("can not fork\n");
38         }
39         else
40         {
41             if (pid)
42             {
43                 printf("I am father, fork reutrnr: %d\n", pid);
44             }
45             else
46             {
47                 printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running);
48             }
49         }
50     }
```

remote Thread 1.1 In: first process

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	0x200000	[ PDBR=0 PCID=0 ]

--Type <RET> for more, q to quit, c to continue without paging--q

Quit

(gdb) n

(gdb) n

(gdb) n

(gdb) n

(gdb) n

(gdb) n

(gdb) n

(gdb) n

fork () at ../src/kernel/syscall.cpp:37

(gdb) n

first\_process () at ../src/kernel/setup.cpp:35

(gdb) p pid

\$1 = 2

(gdb)

最后可以看到父进程返回 `pid=2`

- 子进程的执行流程

首先区别于父进程，子进程由 `asm_start_process` 进入

数据寄存器和段寄存器变化

```
Activities Terminal 6月 8 10:3
Terminal
../src/utils/asm_utils.asm
28      dd 0
29      ASM_GDTR dw 0
30      dd 0
31      ASM_TEMP dd 0
32      ; void asm_update_cr3(int address)
33      asm_update_cr3:
34          push eax
35          mov eax, dword[esp+8]
36          mov cr3, eax
37          pop eax
38          ret
39      asm_start_process:
40          ; jmp $
B+ 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
49          iret
50
51      ; void asm_ltr(int tr)
52      asm_ltr:
53          ltr word[esp + 1 * 4]
54          ret
55
56      ; int asm_add_global_descriptor(int low, int high);

remote Thread 1.1 In: asm_start_process
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xc0026f5c 0xc0026f5c <PCB_SET+12252>
ebp      0x8048fac 0x8048fac
esi      0x0      0
edi      0x0      0
eip      0xc0022dc7 0xc0022dc7 <asm_start_process+7>
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 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

根据目前的cs,ss,esp值，也可以看出目前操作系统处于内核态，eax,esp已经被初始化

```
Activities Terminal 6月 8 10:3
Terminal
../src/utils/asm_utils.asm
28         dd 0
29     ASM_GDTR dw 0
30         dd 0
31     ASM_TEMP dd 0
32     ; void asm_update_cr3(int address)
33     asm_update_cr3:
34         push eax
35         mov eax, dword[esp+8]
36         mov cr3, eax
37         pop eax
38         ret
39     asm_start_process:
40         ; jmp $
41     B+ 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
49     >49     iret
50
51     ; void asm_ltr(int tr)
52     asm_ltr:
53         ltr word[esp + 1 * 4]
54         ret
55
56     ; int asm_add_global_descriptor(int low, int high);

remote Thread 1.1 In: asm_start_process
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 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

看看即将 `iret` 前的寄存器/栈值，已经成功将父进程栈指针的值赋给子进程的，可以看到操作系统应当仍处于内核态，但是 `ss`, `cs` 值已经改变了

```
主 页  Ubuntu  6月 8 10:35
Terminal
Terminal

../src/utls/asm_utls.asm
135     push edx
136     push esi
137     push edi
138
139     mov eax, [ebp + 2 * 4]
140     mov ebx, [ebp + 3 * 4]
141     mov ecx, [ebp + 4 * 4]
142     mov edx, [ebp + 5 * 4]
143     mov esi, [ebp + 6 * 4]
144     mov edi, [ebp + 7 * 4]
145
146     int 0x80
147
>148     pop edi
149     pop esi
150     pop edx
151     pop ecx
152     pop ebx
153     pop ebp
154
155     ret
156
157     ; void asm_init_page_reg(int *directory);
158     asm_init_page_reg:
159     push ebp
160     mov ebp, esp
161
162     push eax
163

remote Thread 1.1 In: asm system call
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 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

此时中断返回，内核态转用户态

```
Activities Terminal 6月 8 10:36
Terminal
../src/utlis/asm_utils.asm
135     push edx
136     push esi
137     push edi
138
139     mov eax, [ebp + 2 * 4]
140     mov ebx, [ebp + 3 * 4]
141     mov ecx, [ebp + 4 * 4]
142     mov edx, [ebp + 5 * 4]
143     mov esi, [ebp + 6 * 4]
144     mov edi, [ebp + 7 * 4]
145
146     int 0x80
147
148     pop edi
149     pop esi
150     pop edx
151     pop ecx
152     pop ebx
153     pop ebp
154
>155     ret
156
157     ; void asm_init_page_reg(int *directory);
158     asm_init_page_reg:
159         push ebp
160         mov ebp, esp
161
162         push eax
163
remote Thread 1.1 In: asm system call
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048fb0 0x8048fb0
ebp      0x8048fdc 0x8048fdc
esi      0x0      0
edi      0x0      0
ebp      0xc0022e75 0xc0022e75 <asm_system_call+34>
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 ]
--Type <RET> for more, q to quit, c to continue without paging--
```

恢复父进程的5个寄存器值，返回调用fork（）处

```

..src/kernel/setup.cpp
22     TSS tss;
23
24     int syscall_0(int first, int second, int third, int forth, int fifth)
25     {
26         printf("system call 0: %d, %d, %d, %d, %d\n",
27             first, second, third, forth, fifth);
28         return first + second + third + forth + fifth;
29     }
30
31     void first_process()
32     {
33         int pid = fork();
34
35         if (pid == -1)
36         {
37             printf("can not fork\n");
38         }
39         else
40         {
41             if (pid)
42             {
43                 printf("I am father, fork return: %d\n", pid);
44             }
45             else
46             {
47                 printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running->pid);
48             }
49         }
50     }

```

```

remote Thread 1.1 In: first process
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 ]
--Type <RET> for more, q to quit, c to continue without paging--
Quit
(gdb) n
fork () at ../src/kernel/syscall.cpp:37
(gdb) p pid
No symbol "pid" in current context.
(gdb) n
first_process () at ../src/kernel/setup.cpp:35
(gdb) p pid
$3 = 0
(gdb)

```

到这里已经可以看到子进程返回 `pid = 0`

父进程子进程的不同之处在于子进程会通过栈上保存的返回地址为asm\_start\_process来进入进程，其它总体过程我们可以看到基本是一致的，

- 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。

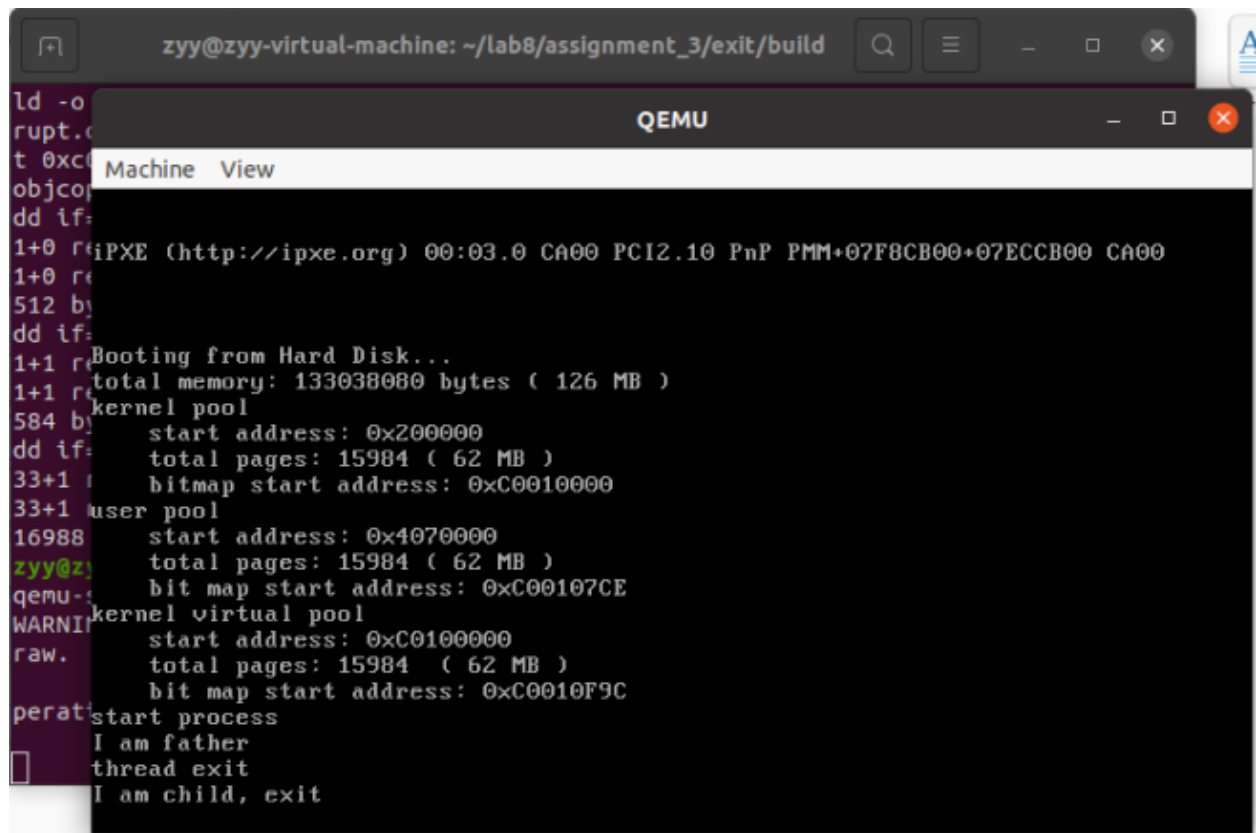
```

// 设置子进程的返回值为0
childpps->eax = 0;

```

设置eax这一部分的内容就是我们fork返回值不同的关键，函数的返回值是通过 `eax` 寄存器来保存，在子进程复制父进程资源后，eax的值如上代码，被修改为0，而后其值一直未被改变，所以最后返回值为0。如下：





- 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。

我们先回忆一下添加了exit的load\_process

```
void load_process(const char *filename)
{
    ...

    interruptStack->esp = memoryManager.allocatePages(AddressPoolType::USER, 1);
    if (interruptStack->esp == 0)
    {
        printf("can not build process!\n");
        process->status = ProgramStatus::DEAD;
        asm_halt();
    }
    interruptStack->esp += PAGE_SIZE;

    // 设置进程返回地址
    int *userStack = (int *)interruptStack->esp;
    userStack -= 3;
    userStack[0] = (int)exit;
    userStack[1] = 0;
    userStack[2] = 0;

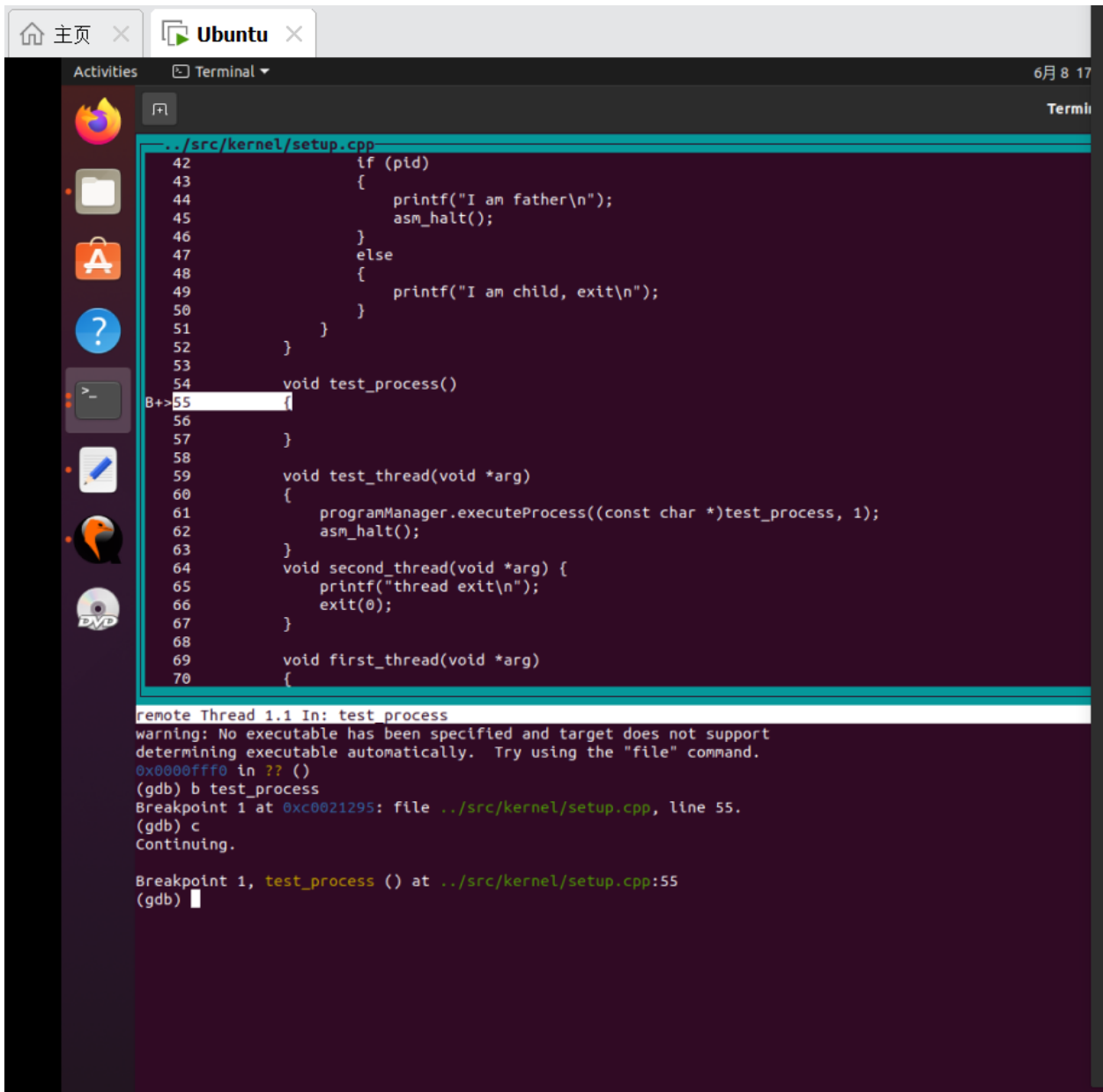
    interruptStack->esp = (int)userStack;

    ...
}
```



`void exit(int ret)` 函数的起始地址以及参数 `ret` 的值放置在堆栈中，栈顶是返回地址 `userStack[0] = (int)exit`，如果一个进程结束，就会从栈中找到返回地址，放入 `eip`，所以这样进程就可以隐式跳转到 `void exit(int ret)`。而 `userStack[1]` 是 `exit` 的返回地址，`userStack[2]` 是 `exit` 的参数 `ret`。故返回值为 0。

根据上述所说，创建一个空进程，执行后隐式调用了 `exit`

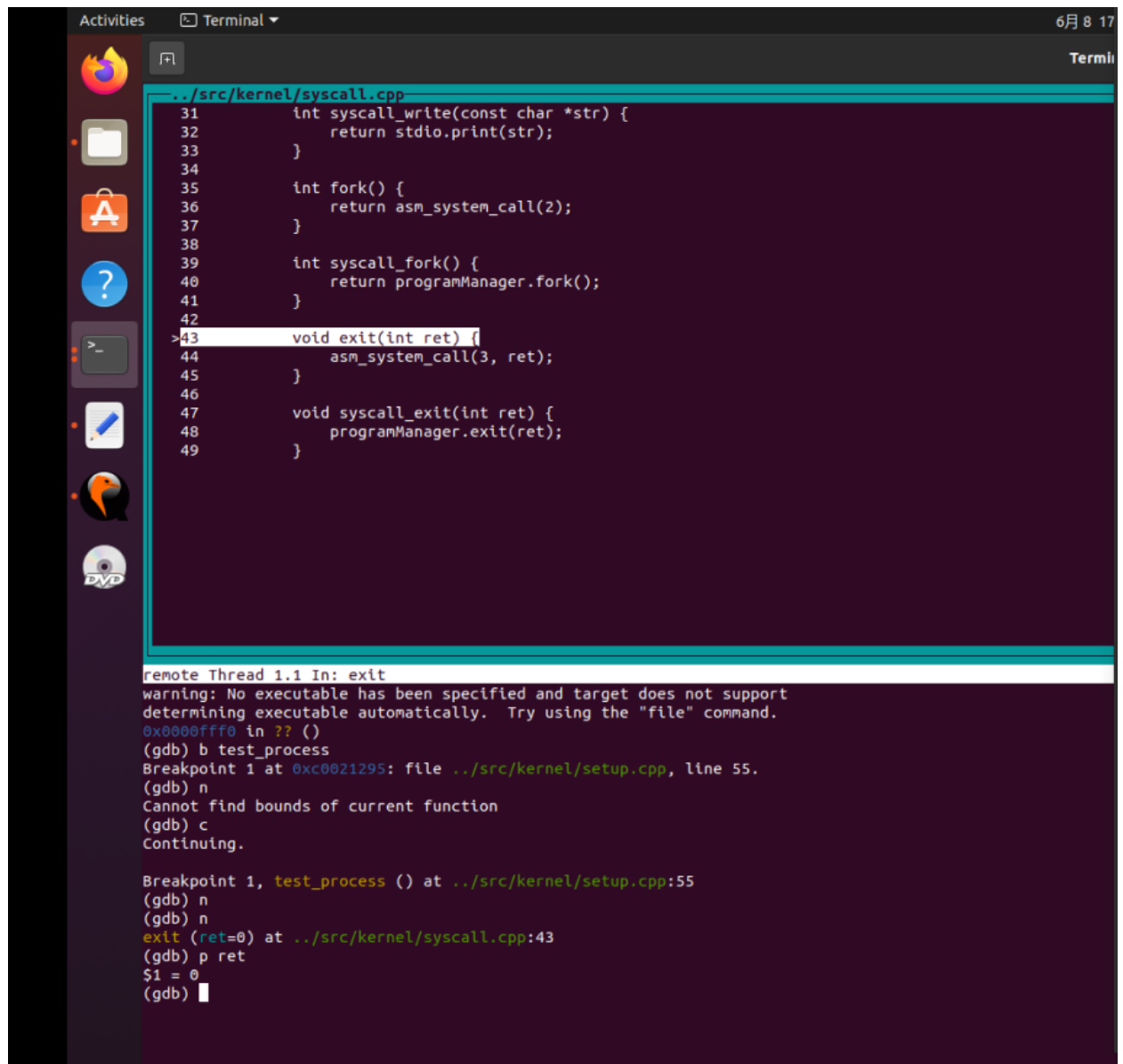


The screenshot shows a terminal window with a Ubuntu desktop environment. The terminal displays the source code of `../src/kernel/setup.cpp` and the output of a GDB session. The code defines a `test_process` function that calls `asm_halt()` and a `test_thread` function that calls `programManager.executeProcess` to run `test_process`. The GDB output shows a warning about no executable specified, a breakpoint set at line 55 of `test_process`, and the program continuing.

```
..src/kernel/setup.cpp
42         if (pid)
43         {
44             printf("I am father\n");
45             asm_halt();
46         }
47         else
48         {
49             printf("I am child, exit\n");
50         }
51     }
52 }
53
54 void test_process()
55 {
56
57 }
58
59 void test_thread(void *arg)
60 {
61     programManager.executeProcess((const char *)test_process, 1);
62     asm_halt();
63 }
64 void second_thread(void *arg) {
65     printf("thread exit\n");
66     exit(0);
67 }
68
69 void first_thread(void *arg)
70 {
    remote Thread 1.1 In: test process
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) b test_process
Breakpoint 1 at 0xc0021295: file ../src/kernel/setup.cpp, line 55.
(gdb) c
Continuing.

Breakpoint 1, test_process () at ../src/kernel/setup.cpp:55
(gdb) █
```

如下图，可以看见返回值是0。



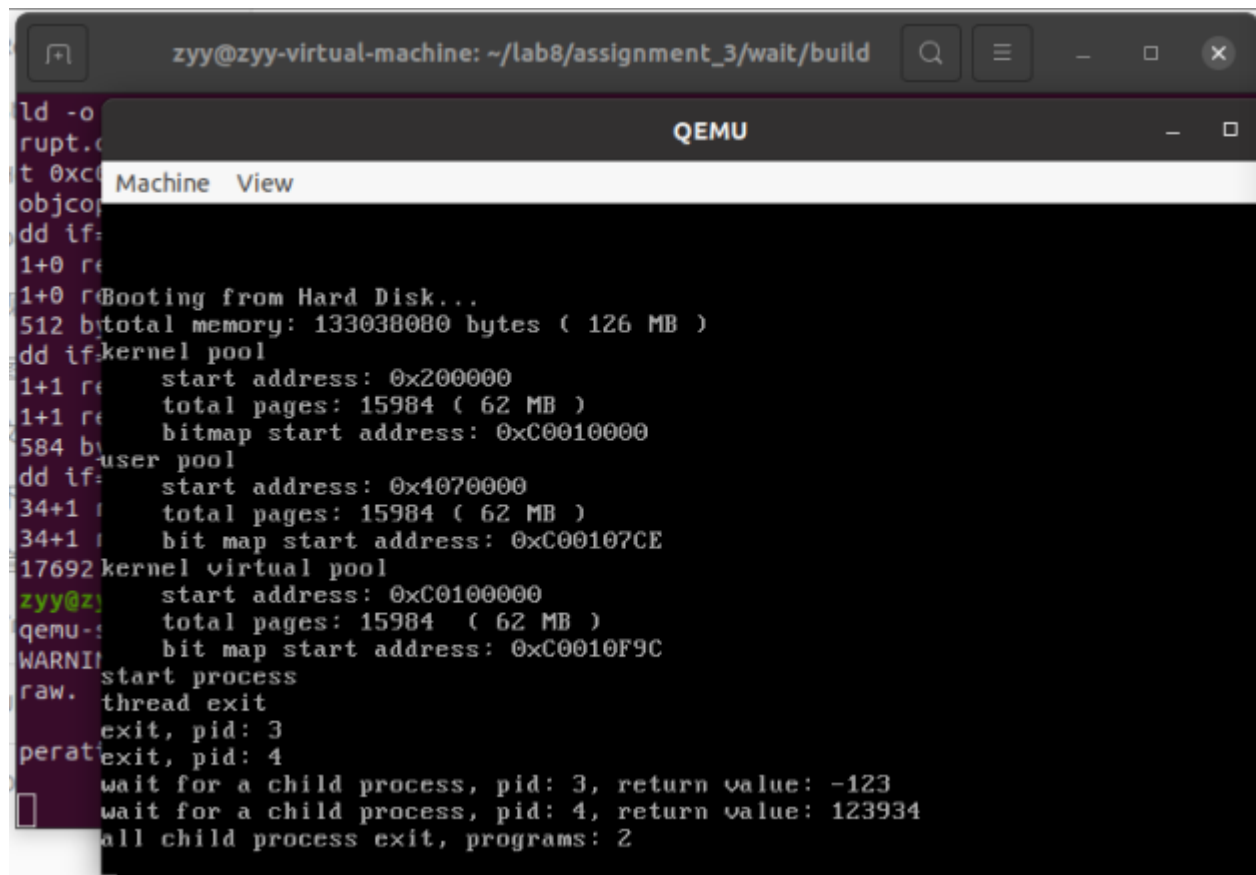
The screenshot shows a Linux desktop environment with a terminal window open. The terminal displays the source code of a C program in `../src/kernel/syscall.cpp`. The code defines several system call wrappers: `syscall_write`, `fork`, `syscall_fork`, `exit`, and `syscall_exit`. The `exit` function is highlighted with a cursor on line 43. Below the code, the GDB debugger output is shown, indicating a breakpoint at line 55 of `../src/kernel/setup.cpp` and the execution of the `exit` function at line 43 of `../src/kernel/syscall.cpp`.

```
..../src/kernel/syscall.cpp
31     int syscall_write(const char *str) {
32         return stdio.print(str);
33     }
34
35     int fork() {
36         return asm_system_call(2);
37     }
38
39     int syscall_fork() {
40         return programManager.fork();
41     }
42
43     void exit(int ret) {
44         asm_system_call(3, ret);
45     }
46
47     void syscall_exit(int ret) {
48         programManager.exit(ret);
49     }
```

remote Thread 1.1 In: exit  
warning: No executable has been specified and target does not support  
determining executable automatically. Try using the "file" command.  
0x0000ffff in ?? ()  
(gdb) b test\_process  
Breakpoint 1 at 0xc0021295: file ../src/kernel/setup.cpp, line 55.  
(gdb) n  
Cannot find bounds of current function  
(gdb) c  
Continuing.  
  
Breakpoint 1, test\_process () at ../src/kernel/setup.cpp:55  
(gdb) n  
(gdb) n  
exit (ret=0) at ../src/kernel/syscall.cpp:43  
(gdb) p ret  
\$1 = 0  
(gdb) █

- 请结合代码逻辑和具体的实例来分析wait的执行过程。

见实验方案，运行结果如下：



```
zyy@zyy-virtual-machine: ~/lab8/assignment_3/wait/build
ld -o
rupt.c
t 0xc
objco
dd if=
1+0 re
1+0 re
512 by
total memory: 133038080 bytes ( 126 MB )
dd if=
kernel pool
1+1 re
start address: 0x200000
1+1 re
total pages: 15984 ( 62 MB )
584 by
bitmap start address: 0xC0010000
dd if=
user pool
34+1
start address: 0x4070000
34+1
total pages: 15984 ( 62 MB )
17692
bit map start address: 0xC00107CE
kernel virtual pool
zyy@zy
start address: 0xC0100000
qemu-
total pages: 15984 ( 62 MB )
WARNI
bit map start address: 0xC0010F9C
raw.
start process
thread exit
exit, pid: 3
perat
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
```

- 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。

参考一下资料，有父进程回收法以及init进程回收法两种方法

[处理僵尸进程的两种经典方法 - 穆晨 - 博客园](#)

## 父进程回收法

`wait`函数将使其调用者阻塞，直到其某个子进程终止。故父进程可调用`wait`函数回收其僵尸子进程。这个修改也很简单，如下：

```
void first_process()
{
    int pid = fork();
    if (pid) // 父进程
    {
        asm_system_call(0, programManager.running->pid, 1);
        int pi = wait(nullptr);
        printf("father process exit\n");
    }
    else
    {
        asm_system_call(0, programManager.running->pid, 0);
        printf("child process exit\n");
        exit(0);
    }
}
```



```

PCB *child;
ListItem *item = this->allPrograms.head.next;
while (item)
{
    child = ListItem2PCB(item, tagInAllList);
    // 找到子进程
    if (child->parentPid == this->running->pid)
    {
        // 修改子进程的父亲
        child->parentPid = this->running->parentPid;
    }
    item = item->next;
}

```

实验结果如下，符合预期

```

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
father process pid value: : 1
test process pid: 1
child process pid value: : 2
test process pid: 2

```

## 实验总结

在本次实验中，又是一个全新的挑战。首先简单学习了保护模式下的特权级的相关内容。学习如何通过特权级保护，区分内核态和用户态，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。其次学习了系统调用，系统调用通过三步来创建进程。这里，首先回顾了上次实验的分页机制等知识，在这些基础上修改，实现进程之间的虚拟地址空间的隔离。最后学习了fork/wait/exit的一种简洁的实现思路。

这次的实验更多偏向于理解，自己写代码部分变少了，更多是利用GDB去探查栈和寄存器变化，更好的去理解fork/wait/exit函数的实现原理与实现过程。这次实验，教程还给我们提了很多问题，需要我们自己理解教程后去解决问题。面对一些陌生的代码，直接想解决问题是不可能的，必须反复看教程，结合理论课所学，才能更好的去回答问题，不那么理解的地方可以通过上网查，和同学讨论来解决。我认为fork的关键在于资源的复制，做好这一

点，其它边边角角的修改就不难操作了，然后要理解父进程与子进程执行完fork后的返回过程的异同。而exit 的要点在于理解它的执行过程，为什么能隐式调用exit并且返回0值。wait函数可以阻塞，理解它的实现过程后，我们可以用它帮助实现孤儿进程（僵尸进程）问题的解决，一个方案是父进程回收法，另一个是init进程回收法（我写了这两个）。

通过这次实验，也是收获了很多东西。

## 参考文献

---

[https://gitee.com/code\\_sysu/os2021-lab8](https://gitee.com/code_sysu/os2021-lab8)

[处理僵尸进程的两种经典方法 - 穆晨 - 博客园](#)