



本科生实验报告

实验课程：_____操作系统_____

实验名称：_____用户态与内核态_____

专业名称：_____计算机科学与技术_____

学生姓名：_____凌国明_____

学生学号：_____21307077_____

实验地点：_____教室_____

实验成绩：_____

报告时间：_____2023. 06. 20_____

1. 实验要求

- 学习保护模式下的特权级的相关内容
- 学习系统调用的概念和如何通过中断来实现系统调用
- 学习内核态下执行特权指令等，执行完成后返回到用户态

2. 实验任务

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。

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
- 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。

1. 实验过程

1) 实现自己的系统调用

- 第一步, 在 include/syscall.h 中声明, 在 src/kernel/setup.cpp 中实现, 并初始化中断号 1 对应的系统调用

```
// 第0个系统调用
int syscall_0(int first, int second, int third, int forth, int fifth);
int syscall_1(int n, int div_factor);
```

- 系统调用 1 的作用是算出 n 的阶乘, 再除以 div_factor

```
int syscall_1(int n, int div_factor)
{
    int ans = 1;
    printf("system call 1: %d %d, res: ", n, div_factor);
    for(int i = 1; i < n; i++){
        ans *= i;
        printf("%d ", ans);
    }
    ans /= div_factor;
    printf("%d \n", ans);
    return ans;
}
```

- 然后创建第二个进程, 在第二个进程中调用 syscall_1

```
// 初始化系统调用
systemService.initialize();
// 设置0号系统调用
systemService.setSystemCall(0, (int)syscall_0);
systemService.setSystemCall(1, (int)syscall_1);
```

```
void second_process()
{
    asm_system_call(1, 11, 22);
    asm_halt();
}

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

- 结果如下

```
start process
system call 0: 132, 324, 12, 124, 18
system call 1: 11 22, res: 1 2 6 24 120 720 5040 40320 362880 3628800 164945
```

- 第一步：启动 gdb 链接 qemu，设置断点

```
remote Thread 1 In: second_process L51 F
0x0000fff0 in ?? ()
(gdb) b second_process
Breakpoint 1 at 0xc0020b03: file ../src/kernel/setup.cpp, line 51
(gdb) c
Continuing.
```

- second_process 调用前，寄存器情况

<pre> B+ 50 void second_process() 51 { > 52 asm system_call(1, 11, 22); 53 asm_halt(); 54 } 55 remote Thread 1 In: second_process eax 0x0 0 ecx 0x0 0 edx 0x0 0 ebx 0x0 0 esp 0x8048ff4 0x8048ff4 ebp 0x8048ffc 0x8048ffc esi 0x0 0 </pre>	<pre> B+ 50 void second_process() 51 { > 52 asm system_call(1, 11, 22); 53 asm_halt(); 54 } 55 56 void first_thread(void *arg) 57 { 58 printf("start process\n"); </pre>
---	---

CPL 的阐述：<https://www.jianshu.com/p/972f1bd497be>：当前特权级 CPL 被存储在 cs 的低 2 位上，cs=0b-0010-1011，则 CPL=0b11=3，处于用户态

CS	0x2b
SS	0x3b

- eax 等寄存器值都是 0，注意此时 esp 的地址，这是特权级为 3，即用户态的栈的位置

```
remote Thread 1 In: second_process
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048ff4 0x8048ff4
ebp      0x8048ffc 0x8048ffc
```

- 调用 int80 中断前，参数被调入寄存器，其中 eax 是系统调用号，ebx 是第一个参数 11，ecx 是第二个参数 22

```

> 146      int 0x80
147
148      pop edi
149      pop esi

remote Thread 1 In: asm_system_call
eax      0x1      1
ecx      0x16     22
edx      0x0      0
ebx      0xb      11
esp      0x8048fb8 0x8048fb8
ebp      0x8048fcc 0x8048fcc
esi      0x0      0

```

- 进入 system_call_handler，esp 从 tss 段自动加载，esp 变成 0 特权级的栈段。这就体现了 tss 的作用：存储每个特权级栈的栈指针和段选择子。

```

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      ; 栈段会从 tss 中自动加载

remote Thread 1 In: asm_system_call_handler
eax      0x1      1
ecx      0x16     22
edx      0x0      0
ebx      0xb      11
esp      0xc0026788 0xc0026788 <PCB_SET+12264>
ebp      0x8048fcc 0x8048fcc
esi      0x0      0

```

- 此时 cs=0b-0010-0000，当前特权级 CPL=0b00=0

```
remote Thread 1 In: asm_system_call_handler
edi          0x0          0
eip          0xc0022718    0xc0022718 <as
eflags       0x12        [ AF ]
cs           0x20        32
ss           0x10        16
ds           0x33        51
es           0x33        51
```

- 系统调用完成后，各通用寄存器的值改变了，这是因为系统调用函数对这些寄存器的值作了修改，其中 eax 是返回值

```
> 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

remote Thread 1 In: asm_system_call
eax          0x28451    164945
ecx          0x0        0
edx          0x0        0
ebx          0x0        0
esp          0x8048fd0    0x8048fd0
ebp          0x8048ffc    0x8048ffc
esi          0x0        0
```

- 系统调用返回前，将 esp 改回特权级为 3 对应的栈，将通用寄存器的值出栈（返回保存好的现场），将 cs 的值进行修改，改回特权级 3

```
remote Thread 1 In: asm_system_call
edi          0x0          0
eip          0xc0022775    0xc0022
eflags       0x212        [ AF IF ]
cs           0x2b        43
ss           0x3b        59
ds           0x33        51
es           0x33        51
```

返回，系统调用结束

2) 任务二: fork

- 先禁止内核线程调用, 创建子进程, 然后通过 copy 父进程的所有资源来初始化子进程

```
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);
}
```

- 这是复制进程资源的函数, 子进程复制父进程的代码段, 0 特权级栈, PCB, 地址池的内容和页目录表和页表的内容

```
bool ProgramManager::copyProcess(PCB *parent, PCB *child)
{
    // 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;

    // 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
}
```

- 注意，这个过程中，要使父子进程要有同一个返回点，让他们从 fork 返回的地方开始执行，

```
// 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
```

这是通过(int)asm_start_process 这个中断来实现的

```
asm_start_process:
    ; jmp $
    mov eax, dword[esp+4]
    mov esp, eax
    popad
    pop gs;
    pop fs;
    pop es;
    pop ds;

    iret
```

父进程的 eip 存放系统调用后的返回地址，也就是 fork 后的返回地址，只需要将父进程保存的 0 特权级栈中的 eip 的内存，移动到子进程的 eip 中（iret 完成），即可实现父子进程都从 fork 返回的地方开始执行

- 通过系统调用，进入 fork 函数，此时栈为父进程的 0 特权级栈，观察 cs 可知 CPL 为 0

```
343 int ProgramManager::fork()
344 {
345     bool status = interruptManager.getInterruptStatus();
346     interruptManager.disableInterrupt();
347
348     // 禁止内核线程调用
349     PCB *parent = this->running;
350     if (!parent->pageDirectoryAddress)
351     {
352         interruptManager.setInterruptStatus(status);
    }
}

remote Thread 1 In: ProgramManager::fork L349
eax 0x201 513
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0xc0025d38 0xc0025d38 <PCB_SET+8024>
ebp 0xc0025d60 0xc0025d60 <PCB_SET+8064>
esi 0x0 0
...Type <return> to continue, or q <return> to quit...

remote Thread 1 In: ProgramManager::fork
edi 0x0 0
eip 0xc00209e7 0xc00209e7 <Pr
eflags 0x82 [ SF ]
cs 0x20 32
ss 0x10 16
ds 0x8 8
es 0x8 8
```


- 通过拷贝父进程，初始化子进程，这里是父进程的寄存器现场

```
remote Thread 1 In: ProgramManager::copyProcess L391 PC: 0xc0020b03
(gdb) n
(gdb) n
(gdb) p /x *parentpss
$1 = {edi = 0x0, esi = 0x0, ebp = 0x8048fac, esp_dummy = 0xc0025dbc,
      ebx = 0x0, edx = 0x0, ecx = 0x0, eax = 0x2, gs = 0x0, fs = 0x33, es = 0x33,
      ds = 0x33, eip = 0xc0022ccf, cs = 0x2b, eflags = 0x216, esp = 0x8048f98,
      ss = 0x3b}

remote Thread 1 In: ProgramManager::copyProcess L391 PC: 0xc0020b03
(gdb) p /x *childprocess
No symbol "childprocess" in current context.
(gdb) p /x *childpss
$2 = {edi = 0x0, esi = 0x0, ebp = 0x8048fac, esp_dummy = 0xc0025dbc,
      ebx = 0x0, edx = 0x0, ecx = 0x0, eax = 0x0, gs = 0x0, fs = 0x33, es = 0x33,
      ds = 0x33, eip = 0xc0022ccf, cs = 0x2b, eflags = 0x216, esp = 0x8048f98,
      ss = 0x3b}
```

可见子进程的寄存器现场与父进程的现场完全相同

继续进行资源复制，直到系统调用返回

- 返回后，父进程的 eax 是 fork 的返回值，也就是子进程的 pid=2

```
B+ 31 void first_process()
    32 {
    33     int pid = fork();
    34
    > 35     if (pid == -1)
    36     {
    37         printf("can not fork\n");
    38     }
    39     else

remote Thread 1 In: first_process
eax      0x2      2
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0x8048fe4 0x8048fe4
ebp      0x8048ffc 0x8048ffc
esi      0x0      0
```

- 可见父进程确实打印了子进程的 pid=2

```
start process
I am father, fork reutrn: 2
```

3) 任务三

- Exit: 先禁中断, 防止执行中突然发生上下文切换。然后将状态置为终止。
如果这个 program 是一个进程, 则要释放这个进程占用的所有资源, 包括所有页目录表、页表、虚拟页和物理页

```
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);
    }
}
```

- 释放所有页目录表、页表、虚拟页和物理页后, 进行一次调度, 执行下一个任务

```
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);
}

schedule();
```

- 进程结束后能隐式调用 exit 是因为进程结束后跳到 userstack[0], 而这里是 exit 函数的入口地址, userstack[2] 是 exit 的返回值, 设置为 0

```
// 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;
```

- **wait 函数**：循环遍历所有进程，查找由当前父进程创建的子进程。如果找到一个子进程，并且该子进程的状态为已退出，则该函数返回该子进程的进程 ID。同时，它还可以通过传入的指针 `retval` 来获取子进程的返回值。在返回之前，它会释放子进程的资源（PCB）并恢复中断状态。如果没有找到已退出的子进程，但存在子进程，则函数会继续等待或调度其他进程运行。如果没有找到任何子进程，或者所有子进程都处于非退出状态，函数将返回-1 表示等待或没有可用子进程。

```
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;
        }
    }
}
```

4) 任务四：僵尸回收

- 编写回收僵尸进程的函数：先关中断，防止这个过程中产生调度。然后寻找僵尸进程：遍历所有进程，如果当前进程的状态是 DEAD，而且他的父进程已退出，则回收这个进程的 PCB（这个进程的所有页已经回收，所以这里不用回收）。最后复原中断状态，返回。

```
void ProgramManager::Release_Zombie_Pss(){
    // 获取原来的中断状况
    bool init_status = interruptManager.getInterruptStatus();
    // 关中断
    interruptManager.disableInterrupt();
    // 寻找僵尸进程
    ListItem *pss = this->allPrograms.head.next;
    while (pss){
        PCB *pss_pcb = ListItem2PCB(pss, tagInAllList);
        // 是进程
        if (pss_pcb->pageDirectoryAddress){
            // 当前pss的状态为dead, 当前pss的父进程pid不是-1（父进程已退出）
            if (pss_pcb->status == ProgramStatus::DEAD && pss_pcb->parentPid != -1){
                // 回收僵尸进程的pcb
                printf("release zombie pss: %d\n", pss_pcb->pid);
                releasePCB(pss_pcb);
            }
        }
        pss = pss->next;
    }

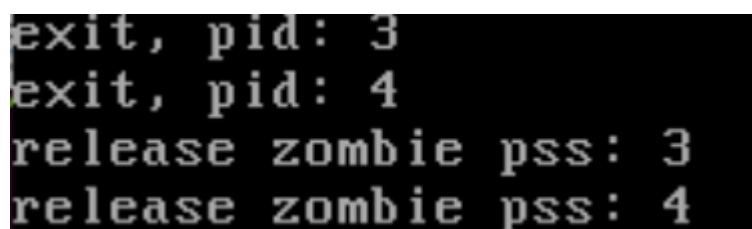
    // 复原中断状态
    interruptManager.setInterruptStatus(init_status);
    return;
}
```

- 将 first_process 中的等待子进程退出的代码删除掉，让父进程先退出，在子进程 dead 后调用 release

```
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);
            }*/
        }
    }
}
```

- 结果



```
exit, pid: 3
exit, pid: 4
release zombie pss: 3
release zombie pss: 4
```

3 关键代码

```
void ProgramManager::Release_Zombie_Pss(){
    // 获取原来的中断状况
    bool init_status = interruptManager.getInterruptStatus();
    // 关中断
    interruptManager.disableInterrupt();
    // 寻找僵尸进程
    ListItem *pss = this->allPrograms.head.next;
    while (pss){
        PCB *pss_pcb = ListItem2PCB(pss, tagInAllList);
        // 是进程
        if (pss_pcb->pageDirectoryAddress){
            // 当前pss的状态为dead, 当前pss的父进程pid不是-1 (父进程已退出)
            if (pss_pcb->status == ProgramStatus::DEAD && pss_pcb->parentPid != -1){
                // 回收僵尸进程的PCB
                printf("release zombie pss: %d\n", pss_pcb->pid);
                releasePCB(pss_pcb);
            }
        }
        pss = pss->next;
    }

    // 复原中断状态
    interruptManager.setInterruptStatus(init_status);
    return;
}
```

先关中断，防止这个过程中产生调度。然后寻找僵尸进程：遍历所有进程，如果当前进程的状态是 DEAD，而且他的父进程已退出，则回收这个进程的 PCB（这个进程的所有页已经回收，所以这里不用回收）。最后复原中断状态，返回。

4 实验结果

1) Assignment1

```
start process
system call 0: 132, 324, 12, 124, 18
system call 1: 11 22, res: 1 2 6 24 120 720 5040 40320 362880 3628800 164945
```

2) Assignment2

```
start process
I am father, fork reutrnr: 2
I am child, fork return: 0, my pid: 2
```

3) Assignment3

```
linggm@linggm-virtual-machine:
QEMU

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
    start address: 0x2000000
    total pages: 15984 ( 62 MB )
    bitmap start address: 0xC0010000
user pool
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0xC00107CE
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0xC0010F9C
start process
all child process exit, programs: 5
thread exit
exit, pid: 3
exit, pid: 4
release zombie pss: 3
release zombie pss: 4
```