

中山大學

本科实验报告

课程名称:	操作系统原理
实验名称:	从内核态到用户态
专业名称:	保密管理
学生姓名:	武自厚
学生学号:	20336014
实验地点:	东校园实验楼 D403
实验成绩:	
报告时间:	2022 年 6 月 1 日

一、 实验要求

1. 系统调用

编写一个系统调用，然后在进程中调用之，根据结果回答以下问题。

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

2. fork() 的奥秘

实现 fork() 函数，并回答以下问题。

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

3. 哼哈二将 exit()&wait()

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。
- 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。

二、 实验过程与结果

1. 系统调用

首先实现一个提供加法服务的系统调用 syscall_0。

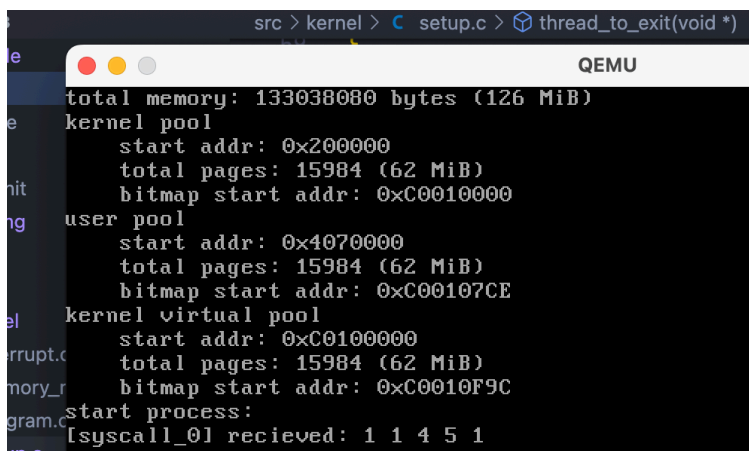
```
int syscall_0(int first, int second, int third, int forth, int fifth) {  
    printf("[syscall_0] recieved: %d %d %d %d %d\n", first, second, third, forth, fifth);  
    return first + second + third + forth + fifth;  
}
```

它可以返回参数的和，并且将它们打印到屏幕上。

根据文档实现了处理中断的函数 asm_system_call()。并实现了系统调用类SystemService。然后编写如下使用了系统调用的线程。

```
void zero_process() {  
    asm_system_call(0, 1, 1, 4, 5, 1);  
    asm_halt();  
}
```

观察实验结果，



```
src > kernel > C setup.c > thread_to_exit(void *)  
QEMU  
total memory: 133038080 bytes (126 MiB)  
kernel pool  
  start addr: 0x200000  
  total pages: 15984 (62 MiB)  
  bitmap start addr: 0xC0010000  
user pool  
  start addr: 0x4070000  
  total pages: 15984 (62 MiB)  
  bitmap start addr: 0xC00107CE  
kernel virtual pool  
  start addr: 0xC0100000  
  total pages: 15984 (62 MiB)  
  bitmap start addr: 0xC0010F9C  
start process:  
[syscall_0] recieved: 1 1 4 5 1
```

可以发现系统调用得到了正确调用。

2. fork() 的设计与实现

在 `fork()` 之前，我们先需要实现进程以及用户地址空间，具体实现步骤大体参照文档完成。

为了实现进程，需要将内核的相对地址提升到 `0x0c000000` 之后，并且提前开启分页机制。之后需要对 TSS 进行定义和实现。准备工作完成后，可以开始实现进程。

用户级进程拥有自己的地址池，所以需要在 PCB 中加上用户地址池成员以及对应的页表地址。所以在初始化进程时，需要进行 PCB 分配、页表分配以及用户地址池初始化三步。

而进程的调度需要在已经实现的线程调度上增加更换内存页目录表以及更换用户栈两个过程。具体实现根据文档的指引完成，此处不在赘述。

为了验证结果，将调用 `syscall_0` 的线程改为进程。

```
void first_thread(void* args) {  
    printf("start process:\n");  
    execute_process(&program_maneger, (char*)zero_process, 2);  
    //execute_process(&program_maneger, (char*)first_process, 1);  
    //execute_thread(&program_maneger, (ThreadFunc)thread_to_exit  
    asm_halt();  
}
```

观察实验结果，

```
src > kernel > C setup.c > thread_to_exit(void *)
QEMU
total memory: 133038080 bytes (126 MiB)
kernel pool
  start addr: 0x2000000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC0010000
user pool
  start addr: 0x4070000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC00107CE
kernel virtual pool
  start addr: 0xC0100000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC0010F9C
start process:
[syscall_0] recieved: 1 1 4 5 1
```

即使变成了进程，我们的系统调用依然可以正确运行。

那么就可以接着实现 `fork()` 了，它的作用是复制调用它的进程，如果他是上级进程，则返回复制生成的新进程的 pid，如果是下级进程则返回 0。根据文档依次实现新进程的创建、PCB 的复制、用户栈的复制以及最为关键的返回地址的复制，于是完成了这个系统调用。

为了验证结果，构造以下进程。

```
void fork_test() {
    int pid = fork();
    if (pid) {
        printf("I am sup-process and return: %d\n", pid);
    } else {
        printf("I am sub-process and return: %d\n", pid);
    }
}
```

观察实验结果，

```
SLAB
src > kernel > C setup.c > fork_test()
QEMU
total memory: 133038080 bytes (126 MiB)
kernel pool
  start addr: 0x2000000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC0010000
user pool
  start addr: 0x4070000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC00107CE
kernel virtual pool
  start addr: 0xC0100000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC0010F9C
start process:
[syscall_0] recieved: 1 1 4 5 1
I am sup-process and return: 3
I am sub-process and return: 0
```

可以看到 `fork()` 在上下级两个进程各返回了一次，且上级进程返回的是下级进程的 pid。

3. wait() 与 exit() 的设计与实现

`exit()` 的实现较为简单，只需要按照文档完成“将 PCB 状态置为 DEAD 并放入返回值”，“释放进程的用户地址空间（如果有）”以及“立即进行进程调度”三个步骤即可。

而 `wait()` 的作用是令上级进程等待其下级进程执行完毕并回收 PCB、获取返回值。实现过程参照课程文档：使用这个系统调用的进程会遍历其下级进程，如果不存在下级进程则返回-1，如果存在且可以回收则返回其返回值，如果存在但不可回收则进程阻塞直到下级进程可以回收。

实现后构造以下进程验证：

```
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", li_size(&program_manager.all_pr
asm_halt());
        }
        else {
            uint32 tmp = 0xffffffff;
            while (tmp)
                --tmp;
            printf("exit, pid: %d\n", program_manager.running->pid);
            exit(114514);
        }
    }
    else {
        uint32 tmp = 0xffffffff;
        while (tmp)
            --tmp;
        printf("exit, pid: %d\n", program_manager.running->pid);
        exit(-1919810);
    }
}
```

可以看到进程中会使用 `fork()` 创造为三级进程，层层等待其下级进程的返回。观察结果：

```
QEMU
total memory: 133038080 bytes (126 MiB)
kernel pool
  start addr: 0x200000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC0010000
user pool
  start addr: 0x4070000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC00107CE
kernel virtual pool
  start addr: 0xC0100000
  total pages: 15984 (62 MiB)
  bitmap start addr: 0xC0010F9C
start process:
终端 [syscall_0] recieved: 1 1 4 5 1
exit, pid: 3
nasl exit, pid: 4
ld wait for a child process, pid: 3, return value: -1919810
am wait for a child process, pid: 4, return value: 114514
obj all child process exit, programs: 3
```

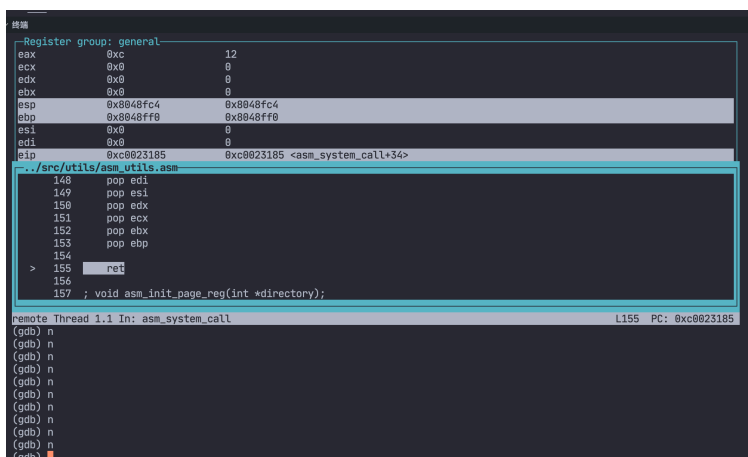
可以观察到三级进程全部成功阻塞并返回。至此代码复现任务完成。

三、问答题

1. 根据 gdb 来分析执行系统调用后的栈的变化情况

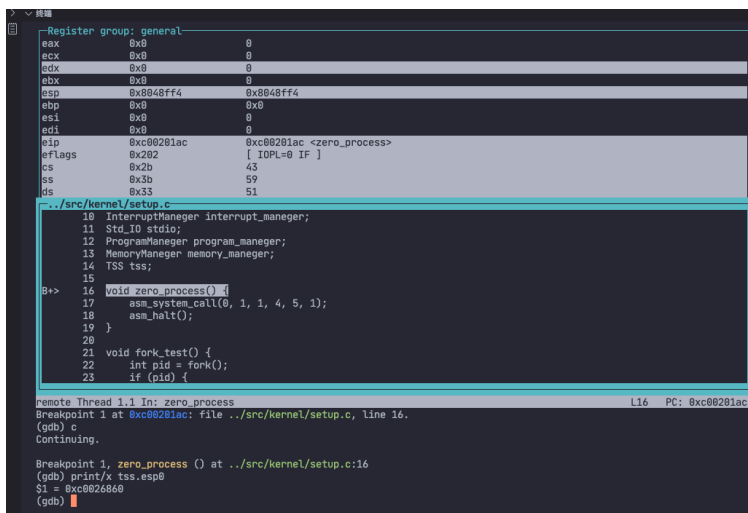
可以看到执行系统调用之后，栈内将会储存参数以及返回值，栈空间会出现一个先变大再变小的过程。这一点可以从 `esp` 的变化中观察到：

The image shows a GDB debugging session. The top panel displays the GDB register window with values for eax, ecx, edx, ebx, esp, ebp, esi, and edi. The middle panel shows the GDB command window with the command 'remote Thread 1.1 In: zero_process' and the output 'Breakpoint 1 at 0xc08231a0: file ../src/kernel/setup.c, line 16.' The bottom panel shows the GDB disassembly window with the assembly code for the 'asm_system_call' function, including instructions like 'push ebp', 'mov ebp, esp', 'push ebx', 'push ecx', 'push edx', 'push esi', 'push edi', and 'mov eax, [ebp + 2 * 4]'. The status bar at the bottom indicates the current instruction is at PC: 0xc08231a0.

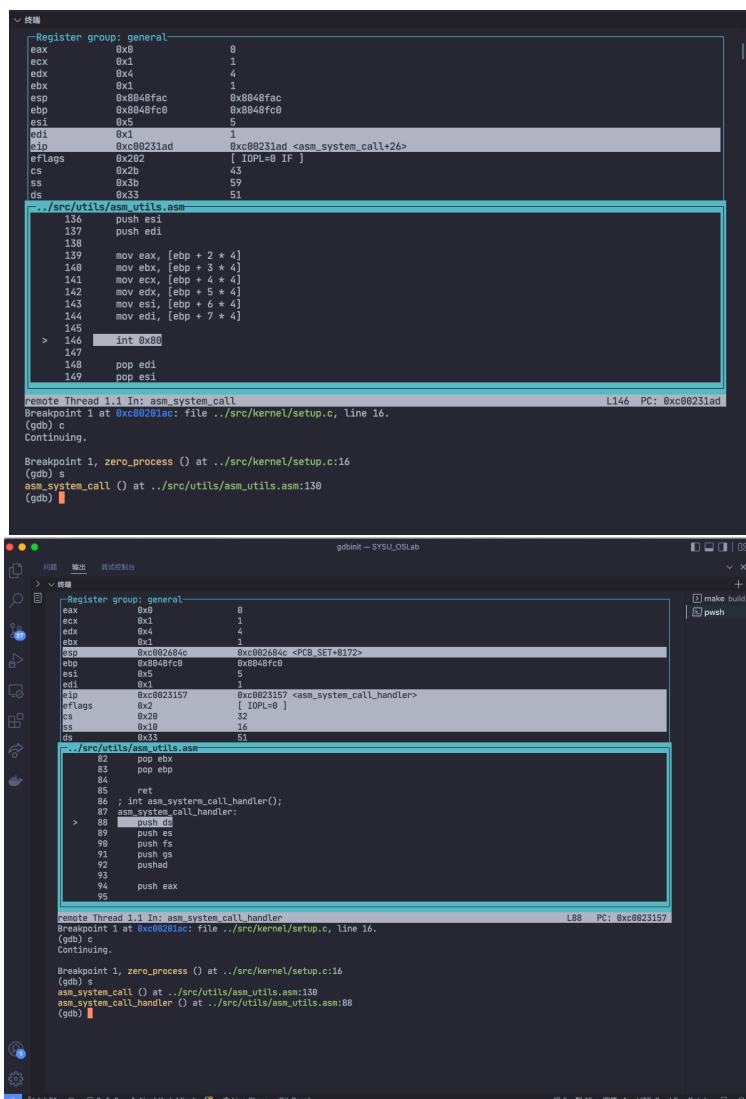


2. 根据 gdb 来说明 TSS 在系统调用执行过程中的作用

先找到一个发起了系统调用的用户级进程，并查看 TSS 保存的系统级栈地址。



可以看到 `tss.esp0 = 0xc0026860`。然后跟踪到特权级切换的中断的指令。



The image contains two screenshots of a GDB debugger interface. The top screenshot shows the assembly code for `asm_system_call` at address `0xc00231ad`. The register window shows `esp` at `0x0048fac` and `ebp` at `0x0048fc0`. The bottom screenshot shows the assembly code for `asm_system_call_handler` at address `0xc0023157`. The register window shows `esp` at `0xc002684c`, which is 5 bytes higher than the previous `esp` value, indicating the stack has grown by 5 bytes (for `ss`, `esp`, `eflags`, `cs`, and `eip`). The GDB console shows the execution flow from `asm_system_call` to `asm_system_call_handler`.

可以看到特权级切换之后栈地址 `esp` 变成了 `0xc002684c`，与 `tss.esp0` 相差了 5 个字节（恰好是 CPU 自动弹出的 `ss`, `esp`, `eflags`, `cs`, `eip`）。因此，TSS 的作用就是保存高特权级栈顶地址。

3. 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
4. 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`

`ProgramManager::fork()` 函数的返回值是新创建的进程的 `pid`，也就是子进程的 `pid`。而在子进程复制父进程的 `pss` 时代码将 `pss` 中的 `eax` 设置为 0，也就导致了子进程的返回值从自己的 `pid` 变成了 0。

5. 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

因为在加载函数 `ProgramManeger::load_process()` 中将用户栈 `pss` 的第 0、1 位分别设为了 `exit()` 的地址以及 0。这样 CPU 就会认为这个进程返回时需要调用 `exit()` 且其参数为 0。

6. 回收僵尸进程