

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 6 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] received: %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，

The screenshot shows a terminal window titled 'src > kernel > C setup.c > thread_to_exit(void *)' running under QEMU. The output displays memory pool configurations:

- 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.

At the bottom, it shows a message from 'start_process': [syscall_0] received: 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_manager, (char*)zero_process, 2);
    //execute_process(&program_manager, (char*)first_process, 1);
    //execute_thread(&program_manager, (ThreadFunc)thread_to_exit
    asm_halt();
}

```

观察实验结果，

```

src > kernel > C setup.c > thread_to_exit(void *)
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_01] received: 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()
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_01] received: 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。实现过程参考课程文档：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_programs));
            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：

```
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_01 received: 1 1 4 5 1
exit, pid: 3
nasexit, 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：

Register group: general

eax	0xc	12
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048fc4	0x8048fc4
ebp	0x8048ff0	0x8048ff0
esi	0x0	0
edi	0x0	0
ebp	0xc0023185	0xc0023185 <asm_system_call+34>

.../src/utils/asm_utils.asm

```

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 L155 PC: 0xc0023185

(gdb) n
(gdb) n

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。

Register group: General

eax	0xd	0
ecx	0x0	0
edx	0x0	0
ebx	0x0	0
esp	0x8048ff4	0x8048ff4
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0xc00201ac	0xc00201ac <zero_process>
eflags	0x282	[IOPL=0 IF]
cs	0x2b	43
ss	0x3b	59
ds	0x33	51

.../src/kernel/setup.c

```

10 InterruptManager interrupt_manager;
11 Std_IDI stdio;
12 ProgramManager program_manager;
13 MemoryManager memory_manager;
14 TSS tss;
15
B+> 16 void zero_process() {
17     asm_system_call(0, 1, 1, 4, 5, 1);
18     asm_halt();
19 }
20
21 void fork_test() {
22     int pid = fork();
23     if (pid) {

```

remote Thread 1.1 In: zero_process L16 PC: 0xc00201ac

Breakpoint 1 at 0xc00201ac: file ..src/kernel/setup.c, line 16.

(gdb) c
Continuing.

Breakpoint 3, zero_process () at ..src/kernel/setup.c:16

(gdb) print/x tss.esp0
\$1 = 0xc0026860

(gdb)

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 shows two vertically stacked GDB sessions. Both sessions are attached to a remote thread (Thread 1.1) and show the assembly code from `asm_utils.asm`. The top session is at address `0xc00231ad` and the bottom session is at address `0xc0023157`. The assembly code is identical in both sessions:

```

    .366 push esi
    .377 push edi
    .388
    .399 mov eax, [ebp + 2 * 4]
    .400 mov ebx, [ebp + 3 * 4]
    .401 mov ecx, [ebp + 4 * 4]
    .402 mov edx, [ebp + 5 * 4]
    .403 mov esi, [ebp + 6 * 4]
    .404 mov edi, [ebp + 7 * 4]
    .405
    > .406 int 0x80
    .407
    .408 pop edi
    .409 pop esi

```

The registers shown in the top session are:

Register	Value
eax	0x0
ecx	0x1
edx	0x4
ebx	0x1
esp	0x8848fcac
ebp	0x8848fc0
esi	5
edi	1
eflags	0x202 [IOPL=0 IF]
cs	0x2b
ss	0x3b
ds	0x33

The registers shown in the bottom session are:

Register	Value
eax	0x0
ecx	0x1
edx	0x4
ebx	0x1
esp	0xc002684c <PCB SET=8172>
ebp	0x8848fc8
esi	5
edi	1
eflags	0x102 [IOPL=0]
cs	0x20
ss	0x10
ds	0x33

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

子进程第一次调度时直接进入到了一个函数的中间，显然它继承了父进程的上下文。

The screenshot shows the GDB debugger interface with the title "gdbinit - SYSU_OSLab". The assembly code window displays the following assembly instructions:

```

    ...
    > 148    pop edi
    149    pop esi
    150    pop edx
    ...

```

The registers window shows the following state:

Register	Value
eax	0x0
ecx	0x0
edx	0x0
ebx	0x0
esp	0x8048fc8
ebp	0x8048fa0
esi	0x0
edi	0x0
eax	0xc002315f
eflags	0x298 [IOPL=0 IF PF]
cs	0x2b
ss	0x3b
ds	0x33
es	0x33
fs	0x33
gs	0x0

The stack dump window shows the current stack contents:

```

    ...
    > 148    pop edi
    149    pop esi
    150    pop edx
    ...

```

The command line shows the following session:

```

(gdb) n
fork_test () at ../src/kernel/setup.c:23
(gdb) c
Continuing.

Breakpoint 1, asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb) s
asm_start_process () at ../src/utils/asm_utils.asm:44
asm_start_process () at ../src/utils/asm_utils.asm:45
asm_start_process () at ../src/utils/asm_utils.asm:46
asm_start_process () at ../src/utils/asm_utils.asm:47
asm_start_process () at ../src/utils/asm_utils.asm:49
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) 

```

可以发现 `esp`, `cs`, `ds` 寄存器发生了变化，在执行这段代码时系统已经完成了换栈。跳转地址显示这段程序将直接跳到 `fork()` 函数然后进行返回。

The screenshot shows the GDB debugger interface with the title "gdbinit - SYSU_OSLab". The assembly code window displays the following assembly instructions:

```

    ...
    > 34    return asm_system_call(2, 0, 0, 0, 0, 0);
    35    }
    36
    37    int syscall_fork() {
    ...

```

The registers window shows the following state:

Register	Value
eax	0x0
ecx	0x0
edx	0x0
ebx	0x0
esp	0x8048fc8
ebp	0x8048fd0
esi	0x0
edi	0x0
eax	0xc0028105
eflags	0x292 [IOPL=0 IF]
cs	0x2b
ss	0x3b
ds	0x33
es	0x33
fs	0x33
gs	0x0

The stack dump window shows the current stack contents:

```

    ...
    > 34    return asm_system_call(2, 0, 0, 0, 0, 0);
    35    }
    36
    37    int syscall_fork() {
    ...

```

The command line shows the following session:

```

(gdb) c
Continuing.

Breakpoint 1, asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb) s
asm_start_process () at ../src/utils/asm_utils.asm:44
asm_start_process () at ../src/utils/asm_utils.asm:45
asm_start_process () at ../src/utils/asm_utils.asm:46
asm_start_process () at ../src/utils/asm_utils.asm:47
asm_start_process () at ../src/utils/asm_utils.asm:49
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) s
fork () at ../src/kernel/syscall.c:35
(gdb) 

```

对比父进程从 `ProgramManager::fork()` 返回时的过程：

gdbinit - SYSU_OSLab

```

Register group: general
eax 0x1 1
ecx 0xc0010f9c -1073672292
edx 0x0 0
ebx 0x0 0
esp 0xc0026758 0xc0026758 <PCB_SET+8024>
ebp 0xc0026780 0xc0026780 <PCB_SET+8064>
esi 0x0 0
edi 0x0 0
eip 0x0022125 0x0022125 spin_fork+243>
[ IOPL=0 IF SF ]
cs 0x28 32
ss 0x10 16
ds 0x8 8
es 0x9 8
fs 0x33 51
gs 0x18 24

```

```

./src/kernel/program.c
321     set_interrupt_status($interrupt_manager, status);
> 322         return pid;
323     }
324 }
```

```

remote Thread 1.1 In: pm_fork L322 PC: 0x0022125
asm_system_call_handler () at ./src/utils/asm_utils.asm:92
asm_system_call_handler () at ./src/utils/asm_utils.asm:94
asm_system_call_handler () at ./src/utils/asm_utils.asm:98
asm_system_call_handler () at ./src/utils/asm_utils.asm:108
asm_system_call_handler () at ./src/utils/asm_utils.asm:109
asm_system_call_handler () at ./src/utils/asm_utils.asm:110
asm_system_call_handler () at ./src/utils/asm_utils.asm:111
asm_system_call_handler () at ./src/utils/asm_utils.asm:112
asm_system_call_handler () at ./src/utils/asm_utils.asm:114
syscall_fork () at ./src/kernel/syscall.c:38
(gdb) n
```

```

Breakpoint 2, pm_fork (pm=0xc00247a0 <program_manager>) at ./src/kernel/program.c:296
(gdb) n
```

选择 目录 转到 终端 线程 遥控 帮助

行 6, 列 10 空格: 4 UTF-8 LF Batch

gdbinit - SYSU_OSLab

```

Register group: general
eax 0x2 2
ecx 0xc0010f9c -1073672292
edx 0x0 0
ebx 0x0 0
esp 0xc0026798 0xc0026798 <PCB_SET+8088>
ebp 0xc00267a0 0xc00267a0 <PCB_SET+8096>
esi 0x0 0
edi 0x0 0
eip 0x0028121 0x0028121 <syscall_fork+26>
eflags 0x28 [ IOPL=0 IF SF ]
cs 0x28 32
ss 0x10 16
ds 0x8 8
es 0x8 8
fs 0x33 51
gs 0x18 24

```

```

./src/kernel/syscall.c
38     return pm_fork(&program_manager);
> 39     }
40
41 void exit(int ret) {

```

```

remote Thread 1.1 In: syscall_fork L39 PC: 0x0028121
asm_system_call_handler () at ./src/utils/asm_utils.asm:98
asm_system_call_handler () at ./src/utils/asm_utils.asm:108
asm_system_call_handler () at ./src/utils/asm_utils.asm:109
asm_system_call_handler () at ./src/utils/asm_utils.asm:110
asm_system_call_handler () at ./src/utils/asm_utils.asm:111
asm_system_call_handler () at ./src/utils/asm_utils.asm:112
asm_system_call_handler () at ./src/utils/asm_utils.asm:114
syscall_fork () at ./src/kernel/syscall.c:38
(gdb) n
```

```

Breakpoint 2, pm_fork (pm=0xc00247a0 <program_manager>) at ./src/kernel/program.c:296
(gdb) n
syscall_fork () at ./src/kernel/syscall.c:39
(gdb) n
```

选择 目录 转到 终端 线程 遥控 帮助

行 6, 列 10 空格: 4 UTF-8 LF Batch

gdbinit - SYSU_OSLab

```

Register group: general
eax 0x2 2
ecx 0x0 0
edx 0x0 0
ebx 0x0 0
esp 0x8048fc8 0x8048fc8
ebp 0x8048fd0 0x8048fd0
esi 0x0 0
edi 0x0 0
eip 0x0020105 0x0020105 <fork+33>
eflags 0x28 [ IOPL=0 IF ]
cs 0x2b 43
ss 0x3b 59
ds 0x33 51
es 0x33 51
fs 0x33 51
gs 0x0 0

```

```

./src/kernel/syscall.c
34     return asm_system_call(2, 0, 0, 0, 0, 0);
> 35     }
36
37 int syscall_fork() {

```

```

remote Thread 1.1 In: fork L35 PC: 0x0020105
Breakpoint 2, pm_fork (pm=0xc00247a0 <program_manager>) at ./src/kernel/program.c:296
(gdb) n
syscall_fork () at ./src/kernel/syscall.c:39
asm_system_call_handler () at ./src/utils/asm_utils.asm:116
asm_system_call_handler () at ./src/utils/asm_utils.asm:120
asm_system_call_handler () at ./src/utils/asm_utils.asm:122
asm_system_call_handler () at ./src/utils/asm_utils.asm:123
asm_system_call_handler () at ./src/utils/asm_utils.asm:124
asm_system_call_handler () at ./src/utils/asm_utils.asm:125
asm_system_call_handler () at ./src/utils/asm_utils.asm:126
asm_system_call () at ./src/utils/asm_utils.asm:148
fork () at ./src/kernel/syscall.c:35
(gdb) n
```

选择 目录 转到 终端 线程 遥控 帮助

行 6, 列 10 空格: 4 UTF-8 LF Batch

可以看到父进程完成了一个完整的系统调用的过程，而子进程在创建时这个系统调用已经执行了一半，所以只需要经过剩下一半的系统调用。符合子进程继承父进程的上下文的特点。

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 ProgramManager::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

僵尸进程即为父进程先于该进程退出的进程。所以回收的方法很简单：如果在进程调度发现当前时 DEAD 状态，且在所有进程中没有找到其父进程，就直接释放其 PCB。具体代码实现为下图用绿色标记的行。

```

106     if (pm->running->status == RUNNING) {
107         pm->running->status = READY;
108         pm->running->ticks = pm->running->priority * 10;
109         li_push_back(&(pm->ready_programs), &(pm->running->tag_in_general_list));
110     } else if (pm->running->status == DEAD) {
111         if (!pm->running->page_directory_addr) {
112             release_pcb(pm->running);
113         } else {
114             ListItem* item = li_front(&pm->all_programs)->next;
115             PCB *temp;
116             Bool hasParent = false;
117             while (item) {
118                 temp = ListItem2PCB(item, tag_in_all_list);
119                 if (temp->pid == pm->running->parent_pid) {
120                     hasParent = true;
121                     break;
122                 }
123             }
124             if (!hasParent) {
125                 release_pcb(pm->running);
126             }
127         }
128     }
129 }
```