



实验课程: 操作系统

实验名称: 从用户态到内核态

专业名称: 计算机科学与技术

学生姓名: 吴臻

学生学号: 21307371

实验地点: 实验中心大楼D栋501

实验成绩:

报告时间: 2023/5/28

- 1. 实验要求
- 2. 预备知识与实验环境
- 3. 实验任务
- 4. 实验步骤/关键代码/实验结果

- [Assignment 1 实现系统调用的方法](#)
- [Assignment 2 进程的创建和调度](#)
- [Assignment 3 fork的实现](#)
- [Assignment 4 wait & exit 的实现](#)
- 5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)
- 6. 参考资料清单

## 1. 实验要求

---

在本章中，我们首先会简单讨论保护模式下的特权级的相关内容。特权级保护是保护模式的特点之一，通过特权级保护，我们区分了内核态和用户态，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。但是，用户态的代码有时不得不使用一些特权指令，如输入输出等。因此，我们介绍了系统调用的概念和如何通过中断来实现系统调用。通过系统调用，我们可以实现从用户态到内核态转移，然后在内核态下执行特权指令等，执行完成后返回到用户态。在实现了系统调用后，我们通过三步来创建了进程。这里，我们需要重点理解我们是如何通过分页机制来实现进程之间的虚拟地址空间的隔离。最后，我们介绍了fork/wait/exit的一种简洁的实现思路。

## 2. 预备知识与实验环境

---

## 3. 实验任务

---

1. 实现系统调用的方法
2. 进程的创建和调度
3. fork的实现
4. wait & exit 的实现

## 4. 实验步骤/关键代码/实验结果

---

### Assignment 1 实现系统调用的方法

---

复现截图：

```

kernel pool
  start address: 0x2000000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0x10000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
system call 0: 0, 0, 0, 0, 0
return value: 0
system call 0: 123, 0, 0, 0, 0
return value: 123
system call 0: 123, 324, 0, 0, 0
return value: 447
system call 0: 123, 324, 9248, 0, 0
return value: 9695
system call 0: 123, 324, 9248, 7, 0
return value: 9702
system call 0: 123, 324, 9248, 7, 123
return value: 9825

```

具体要求如下：

1. 请解释为什么需要使用寄存器来传递系统调用的参数，以及我们是如何在执行 `int 0x80` 前在栈中找到参数并放入寄存器的。

回答：（1）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。

（2）使用栈指针，加上相应的偏移量就可以找到参数

2. 请使用gdb来分析在我们调用了 `int 0x80` 后，系统的栈发生了怎样的变化？ `esp` 的值和在 `setup.cpp` 中定义的变量 `tss` 有什么关系？此外还有哪些段寄存器发生了变化？变化后的内容是什么？

```
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

remote Thread 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      0xc00226cd 0xc00226cd <asm_system_call+26>
eflags   0x212   [ AF IF ]
cs       0x2b    43
ss       0x3b    59
ds       0x33    51
es       0x33    51
fs       0x33    51
gs       0x0     0
(gdb) p $ss
$1 = 59
(gdb)

126      mov eax, [ASM_TEMP]
127
B+> 128      iret
129      asm_system_call:
B+ 130      push ebp
131      mov ebp, esp

remote Thread 1 In: asm system call handler
eax      0x250    592
ecx      0x144    324
edx      0xc     12
ebx      0x84    132
esp      0xc002568c 0xc002568c <PCB_SET+8172>
ebp      0x8048fcc 0x8048fcc
esi      0x7c    124
edi      0x0     0
eip      0xc00226b2 0xc00226b2 <asm_system_call_handler+59>
eflags   0x86    [ PF SF ]
cs       0x20    32
ss       0x10    16
ds       0x33    51
es       0x33    51
fs       0x33    51
gs       0x0     0
(gdb) p $ss
$2 = 16
(gdb)
```

回答:

- (1) 调用了 `int 0x80` 后, 系统的栈从特权级3的栈 (`0x8048fb8`) 变为特权级0的栈 (`0xc002568c`)
- (2) `tss`的作用仅限于为CPU提供0特权级栈所在的地址和段选择子, 即CPU只会用到`tss`中的`esp0`和`SS0`, `esp`的值和`tss`中的`esp0`相等
- (3) `cs`和`ss`段寄存器发生了变化, 即代码段和栈段发生了变化, 从特权级3到特权级0, `cs`从`0x2b`到`0x20`, `ss`从`0x2b`到`0x10`

3. 请使用gdb来分析在进入 `asm_system_call_handler` 的那一刻, 栈顶的地址是什么? 栈中存放的内容是什么? 为什么存放的是这些内容?

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

../src/utils/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

remote Thread 1 In: asm system call handler L88 PC
Breakpoint 1 at 0xc0022677: file ../src/utils/asm_utils.asm, line 8
(gdb) c
Continuing.

Breakpoint 1, asm_system_call_handler () at ../src/utils/asm_utils.
(gdb) p $esp
$1 = (void *) 0xc002568c <PCB_SET+8172>
(gdb)
```

```
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     ; 栈段会从tss中自动加载
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 In: asm_system_call_handler
(gdb) x/16a $esp
0xc002568c <PCB_SET+8172>: 0xc00226cf 0x2b 0x212 0x8048fb8
0xc002569c <PCB_SET+8188>: 0x3b 0xc0026640 0x83e58955 0xec8308ec
0xc00256ac <PCB_SET+8204>: 0x6a08 0x0 0x0 0x2
0xc00256bc <PCB_SET+8220>: 0x1 0x2 0xa 0x0
(gdb)
```

回答:

- (1) 栈顶的地址是0xc002568c;
- (2) 栈中存放着原来的栈(特权级3)的部分寄存器, 比如: \$esp, \$eflags, \$cs, \$ss
- (3) 当系统调用结束时, 需要从内核态返回后回到系统调用前现场, 保存这些寄存器可以恢复现场

4. 请结合代码分析 `asm_system_call_handler` 是如何找到中断向量号 `index` 对应的函数的。

(1) 创建 `int system_call_table[MAX_SYSTEM_CALL];`

(2) 设置系统调用函数, `function` 是第 `index` 个系统调用的处理函数的地址

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

(3) 调用函数(跳转到函数所在地址)

```
sti
call dword[system_call_table + eax * 4]
cli
```

5. 请使用gdb来分析在 `asm_system_call_handler` 中执行 `iret` 后, 哪些段寄存器发生了变化? 变化后的内容是什么? 这些内容来自于什么地方?

```
148         int ebx;
149         pop edi;
150         pop esi;
151         pop edx;
152         pop ecx;
153         pop ebx;
154         pop ebp;
155         ret;
156
157         ; void asm_init_page_reg(int *directory);
158         asm_init_page_reg:
159         push ebp;
160         mov ebp, esp;

remote Thread 1 In: asm system call
asm_system_call () at ../src/utils/asm_utils.asm:148
(gdb) info registers
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             0xc00226cf        0xc00226cf <asm_system_call+28>
eflags         0x212             [ AF IF ]
cs              0x2b              43
ss              0x3b              59
ds              0x33              51
es              0x33              51
fs              0x33              51
gs              0x0               0
(gdb)
```

回答：cs和ss段寄存器发生了变化，变为调用int0x80前的值，即特权级3下的值

## Assignment 2 进程的创建和调度

复现截图：

```
QEMU
PXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDC
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
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
```

1. 请结合代码分析我们是如何在线程的基础上创建进程的PCB的（即分析进程创建的三个步骤）。

回答：

（1）修改PCB内容(进程有自己的虚拟地址空间和相应的分页机制，也就是虚拟地址池和页目录表)

```
struct PCB
{
    ...
    int pageDirectoryAddress;           // 页目录表地址
```

```
AddressPool userVirtual;           // 用户程序虚拟地址池
};
```

## (2) 创建进程的PCB。

```
// 在线程创建的基础上初步创建进程的PCB
int pid = executeThread((ThreadFunction)load_process,
                        (void *)filename, filename, priority);

if (pid == -1)
{
    interruptManager.setInterruptStatus(status);
    return -1;
}
```

(3) 初始化进程的页目录表。从内核中分配的页的虚拟地址已经被提升到3GB以上的空间。为了使进程能够访问内核资源，根据之前的约定，我们令用户虚拟地址的3GB-4GB的空间指向内核空间。为此，我们需要将内核的第768-1022个页目录项复制到进程的页目录表的相同位置。值得注意的是，我们需要构造出页目录表的第768-1022个页目录项分别在内核页目录表 and 用户进程中的虚拟地址。

```
int ProgramManager::createProcessPageDirectory()
{
    // 从内核地址池中分配一页存储用户进程的页目录表
    int vaddr = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    if (!vaddr)
    {
        //printf("can not create page from kernel\n");
        return 0;
    }

    memset((char *)vaddr, 0, PAGE_SIZE);

    // 复制内核目录项到虚拟地址的高1GB
    int *src = (int *) (0xfffff000 + 0x300 * 4);
    int *dst = (int *) (vaddr + 0x300 * 4);
    for (int i = 0; i < 256; ++i)
    {
        dst[i] = src[i];
    }

    // 用户进程页目录表的最后一项指向用户进程页目录表本身
    ((int *)vaddr)[1023] = memoryManager.vaddr2paddr(vaddr) | 0x7;

    return vaddr;
}
```

## (4) 初始化进程的虚拟地址池。

```

bool ProgramManager::createUserVirtualPool(PCB *process)
{
    int sourcesCount = (0xc0000000 - USER_VADDR_START) / PAGE_SIZE;
    int bitmapLength = ceil(sourcesCount, 8);

    // 计算位图所占的页数
    int pageCount = ceil(bitmapLength, PAGE_SIZE);

    int start = memoryManager.allocatePages(AddressPoolType::KERNEL,
    pageCount);

    if (!start)
    {
        return false;
    }

    memset((char *)start, 0, PAGE_SIZE * pageCount);
    (process->userVirtual).initialize((char *)start, bitmapLength,
    USER_VADDR_START);

    return true;
}

```

2. 在进程的PCB第一次被调度执行时，进程实际上并不是跳转到进程的第一条指令处，而是跳转到 `load_process` 。请结合代码逻辑和gdb来分析为什么 `asm_switch_thread` 在执行 `ret` 后会跳转到 `load_process` 。

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

```

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/utls/asm_utils.asm
206      pop edi
207      pop ebx
208      pop ebp
209
210      sti
B+> 211      ret
212      ; int asm_interrupt_status();
213      asm_interrupt_status:
214      xor eax, eax
215      pushfd
216      pop eax
217      and eax, 0x200
218      ret

remote Thread 1 In: asm_switch_thread L211 PC: 0xc00227:
Breakpoint 2, asm_switch_thread () at ../src/utls/asm_utils.asm:211
(gdb) x/16a $esp
0xc0025650 <PCB_SET+8112>: 0xc0020833 0xc002041c 0xc0020a62
0x0
0xc0025660 <PCB_SET+8128>: 0x0 0x0 0x0 0x0
0xc0025670 <PCB_SET+8144>: 0x0 0x0 0x0 0x0
0xc0025680 <PCB_SET+8160>: 0x0 0x0 0x0 0x0
(gdb)

```



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/program.cpp
285
286     return true;
287 }
288
289 void load_process(const char *filename)
> 290 {
291     interruptManager.disableInterrupt();
292
293     PCB *process = programManager.running;
294     ProcessStartStack *interruptStack =
295         (ProcessStartStack *)((int)process + PAGE_SIZE -
296
297     interruptStack->edi = 0;

Remote Thread 1 In: load_process L290 PC:
load_process (
  filename=0xc0020a62 <first_process()> "U\211\345\203\354\b\203\35
  at ../src/kernel/program.cpp:290
(gdb) p
The history is empty.
(gdb) p $eip
$1 = (void (*)(void)) 0xc0020833 <load_process(char const*)>
(gdb) 
```

回答：调用asm\_switch\_thread(cur,next)时，将next和cur依次压入栈中，然后进入该函数，执行以上代码，函数一开始先压入四个寄存器，所以`[esp + 5 \* 4]`为cur，保存当前栈指针到PCB中，`[esp + 6 \* 4]`为next (PCB\*指针)，指针（无偏移）一开始指向PCB首地址（即存放stack指针的地址），取其内容（得到stack）并赋值给esp就完成了栈的切换，从截图1可以看到此时\$esp指向的地址里存储0xc0020833（从截图2可以看到这就是load\_process的地址），执行ret 指令会取出栈顶的返回地址，并跳转到该地址所指向的位置

- 在跳转到 load\_process 后，我们巧妙地设置了 ProcessStartStack 的内容，然后在 asm\_start\_process 中跳转到进程第一条指令处执行。请结合代码逻辑和gdb来分析我们是如何设置 ProcessStartStack 的内容，从而使得我们能够在 asm\_start\_process 中实现内核态到用户态的转移，即从特权级0转移到特权级3下，并使用 iret 指令成功启动进程的。

## 特权级0

```
39     asm_start_process:
40     ;imp $
> 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]

Remote Thread 1 In: asm_start_process
asm_start_process () at ../src/utils/asm_utils.asm:41
(gdb) info registers
eax      0xc002565c      -1073588644
ecx      0xc0010000      -1073676288
edx      0x3b          59
ebx      0x0           0
esp      0xc0025624      0xc0025624 <PCB_SET+8068>
ebp      0xc0025650      0xc0025650 <PCB_SET+8112>
esi      0x0           0
edi      0x0           0
eip      0xc0022620      0xc0022620 <asm_start_process>
eflags   0x20          [ AF SF ]
cs       0x20          32
ss       0x10          16
ds       0x8           8
es       0x8           8
fs       0x0           0
gs       0x18          24
(gdb) 
```

## 特权级3

```
48      pop ebx
3+> 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);
57      asm_add_global_descriptor:
58          push ebp
59          mov ebp, esp
60
61          push ebx
62          push esi
63
remote Thread 1 In: asm_start_process
Breakpoint 3, asm_start_process () at ../src/utls/asm_utils.asm:49
(gdb) info registers
eax            0x00000000
ecx            0x00000000
edx            0x00000000
ebx            0x00000000
esp            0xc002568c      0xc002568c <PCB_SET+8172>
ebp            0x00000000
esi            0x00000000
edi            0x00000000
eip            0xc002262d      0xc002262d <asm_start_process+13>
eflags         0x00000092      [ AF SF ]
cs             0x20000000
ss             0x10000000
ds             0x33000000
fs             0x33000000
gs             0x33000000
(gdb)
```

\$esp的内容和进程执行函数的地址一致

```
终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/setup.cpp
27      first, second, third, forth, fifth);
28      return first + second + third + forth + fifth;
29  }
30
31  void first_process()
> 32  {
33      asm_system_call(0, 132, 324, 12, 124);
34      asm_halt();
35  }
36
37  void first_thread(void *arg)
38  {
39      printf("start process\n");
}

remote Thread 1 In: first_process      L32      PC: 0xc002
(gdb) x/4a $esp
0xc002568c <PCB_SET+8172>:      0xc0020a62      0x2b      0x202      0x8049000
(gdb) s
first_process () at ../src/kernel/setup.cpp:32
(gdb) p $eip
$1 = (void (*)(void)) 0xc0020a62 <first_process()>
(gdb)
```

回答：由于参数 stack 是通过函数调用从栈中传递进来的，因此可以通过 esp+4 的方式来读取该参数的值并保存到 eax 寄存器中。接着，将 eax 中保存的参数值赋值给 esp 寄存器，从而改变了当前的栈顶位置，让 esp 指向了新进程的堆栈顶部。然后，该函数使用 popad 指令依次将通用寄存器 eax、ebx、ecx、edx、esp\_dummy、edi、esi 和 ebp 中保存的值取出，并分别恢复到相应的寄存器中。接着，该函数使用 pop gs、pop fs、pop es 和 pop ds 等指令依次将特殊寄存器 gs、fs、es 和 ds 中保存的值取出，并恢复到相应的寄存器中。这一系列操作使得 CPU 的状态得以完全恢复，以便重新执行新进程的代码，执行完上述操作后，\$esp指向stack里\$eip的地址（进程执行函数的地址），使用 iret 指令，便可以成功启动进程

4. 结合代码，分析在创建进程后，我们对 ProgramManager::schedule 作了哪些修改？这样做的目的是什么？

回答：

（1）切换页目录表。使 CPU 开始使用由参数 program 指向的进程的页表。这样，进程就能够正常地访问自己的虚拟地址空间了。

（2）更新TSS中的特权级0的栈。作用是为将要执行的进程设置正确的内核堆栈地址，从而确保进程在执行内核态代码时能够正常地使用高特权级栈

```

void ProgramManager::activateProgramPage(PCB *program)
{
    int paddr = PAGE_DIRECTORY;

    if (program->pageDirectoryAddress)
    {
        tss.esp0 = (int)program + PAGE_SIZE;
        paddr = memoryManager.vaddr2paddr(program->pageDirectoryAddress);
    }

    asm_update_cr3(paddr);
}

```

5. 需要编写一个 ProgramManager 的成员函数 findProgramByPid,并用上面这个函数替换指导书中提到的"存在风险的语句"

实现思路:

PCB\_SET的地址我们已经知道,我们是按顺序存储PCB的,所以只需偏移pid个PCB\_SIZE大小的位置即可

```

PCB *findProgramByPid(int pid)
{
    return (PCB *)((int)PCB_SET + PCB_SIZE * pid);
}

```

结果截图

```

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0

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
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0
system call 0: 132, 324, 12, 124, 0

```

## Assignment 3 fork的实现

复现截图：

```

SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 P

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

```

1. 请根据代码逻辑概括 `fork` 的实现的基本思路，并简要分析我们是如何解决"四个关键问题"的。

## 基本思路

- (1) 禁止内核线程调用
- (2) 创建子进程
- (3) 初始化子进程（复制父进程的资源到子进程）

## 问题解决方法

Q1: 如何实现父子进程的代码段共享?

A1: 我们使用了函数来模拟一个进程，而函数的代码是放在内核中的，进程又划分了3GB~4GB的空间来实现内核共享，因此进程的代码天然就是共享的

Q2: 如何使得父子进程从相同的返回点开始执行?

A2: `ProgramStartProcess`中保存了父进程的eip, eip的内容也是`asm_system_call_handler`的返回地址, `asm_start_process`的最后的`iret`会将上面说到的保存在0特权级栈的eip的内容送入到eip中。执行完eip后，子进程便可以从父进程的返回点处开始执行，即`asm_system_call_handler`的返回地址。然后子进程依次返回到`syscall_fork`, `asm_system_call_handler`，最终从`fork`返回

Q3: 除代码段外，进程包含的资源有哪些?

A3: 进程包含的资源有0特权级栈，PCB、虚拟地址池、页目录表、页表及其指向的物理页。

Q4: 如何实现进程的资源在进程之间的复制?

A4: 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

2. 请根据gdb来分析子进程第一次被调度执行时，即在 `asm_switch_thread` 切换到子进程的栈中时，`esp` 的地址是什么？栈中保存的内容是什么？

```
B+> 211         ret
      212         ; int asm_interrupt_status();
      213         asm_interrupt_status:
      214             xor eax, eax
      215             pushfd
      216             pop eax
      217             and eax, 0x200
      218             ret
      219
      220         ; void asm_disable_interrupt();
      221         asm_disable_interrupt:
      222             cli
      223             ret
      224         ; void asm_init_page_reg(int *directory);
      225
      226         asm_enable_interrupt:
      227             sti
      228             ret

remote Thread 1 In: asm_switch_thread
Breakpoint 1, asm_switch_thread () at ../src/utils/asm_utils.asm:194
(gdb) c
Continuing.

Breakpoint 2, asm_switch_thread () at ../src/utils/asm_utils.asm:211
(gdb) p $esp
$2 = (void *) 0xc0026d90 <PCB_SET+12208>
(gdb) x/4a $esp
0xc0026d90 <PCB_SET+12208>:    0xc0022c20    0x0    0xc0026d9c    0x0
(gdb)

38         ret
      39         asm_start_process:
      40             ; jmp $
> 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

remote Thread 1 In: asm_start_process
(gdb) si
asm_start_process () at ../src/utils/asm_utils.asm:4
(gdb) p $eip
$3 = (void (*)()) 0xc0022c20 <asm_start_process>
(gdb)
```

回答：esp 的地址是0xc0026d90，栈中保存了asm\_start\_process的地址，成功跳转进asm\_start\_process 以保证正确返回子进程。

- 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。

```
B+> 210         sti
      211         ret
      212         ; int asm_interrupt_status();
      213         asm_interrupt_status:
      214             xor eax, eax
      215             pushfd
      216             pop eax
      217             and eax, 0x200
      218             ret
      219
      220         ; void asm_disable_interrupt();
      221         asm_disable_interrupt:
      222             cli

remote Thread 1 In: asm_switch_thread
--Type <return> to continue, or q <return> to quit---q
Quit
(gdb) info registers
eax             0xc0023de0             -1073594912
ecx             0x1                    1
edx             0x0                    0
ebx             0x0                    0
esp             0xc0024ce4             0xc0024ce4 <PCB_SET+3844>
ebp             0xc0024d10             0xc0024d10 <PCB_SET+3888>
esi             0x0                    0
edi             0x0                    0
eip             0xc0022d19             0xc0022d19 <asm_switch_thread+21>
eflags          0x286                  [ PF SF IF ]
cs              0x20                    32
ss              0x10                    16
ds              0x33                    51
es              0x33                    51
fs              0x33                    51
gs              0x0                    0
(gdb)
```

```
39 asm_start_process:
40     ; jmp $
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
B+> 49     lret
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);
57     asm_add_global_descriptor:
58     push ebp
59     mov ebp, esp
60     push ebx

remote Thread 1 In: asm_start_process
Breakpoint 2, asm_start_process () at ../src/utils/asm_utils.asm:4
(gdb) info registers
eax            0x0            0
ecx            0x0            0
edx            0x0            0
ebx            0x0            0
esp            0xc0026dcc      0xc0026dcc <PCB_SET+12268>
ebp            0x8048fac      0x8048fac
esi            0x0            0
edi            0x0            0
eip            0xc0022c2d      0xc0022c2d <asm_start_process+13>
eflags        0x286          [ PF SF IF ]
cs             0x20          32
ss             0x10          16
ds             0x33          51
es             0x33          51
fs             0x33          51
gs             0x0            0
(gdb)

148     pop edi
149     pop esi
150     pop edx
151     pop ecx
152     pop ebx
153     pop ebp
154
B+> 155     ret
156
157     ; void asm_init_page_reg(int *directory);
158     asm_init_page_reg:
159     push ebp
160     mov ebp, esp

remote Thread 1 In: asm_system_call
Breakpoint 3, asm_system_call () at ../src/utils/asm_utils.asm:155
(gdb) info registers
eax            0x0            0
ecx            0x0            0
edx            0x0            0
ebx            0x0            0
esp            0x8048fb0      0x8048fb0
ebp            0x8048fdc      0x8048fdc
esi            0x0            0
edi            0x0            0
eip            0xc0022cd5      0xc0022cd5 <asm_system_call+34>
eflags        0x216          [ PF AF IF ]
cs             0x2b          43
ss             0x3b          59
ds             0x33          51
es             0x33          51
fs             0x33          51
gs             0x0            0
(gdb)
```

回答：子进程执行流程：asm\_switch\_thread（内核态） -> asm\_start\_process（内核态） -> asm\_system\_call(回到用户态)

父进程执行完 ProgramManager::fork 后的返回过程：asm\_system\_call\_handler（内核态） -> asm\_system\_call(回到用户态) -> fork()

4. 请根据代码逻辑和gdb来解释子进程的 fork 返回值为什么是0，而父进程的 fork 返回值是子进程的pid。

父进程

```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
../src/kernel/syscall.cpp
32     return stdio.print(str);
33 }
34
B+> 35     int fork() {
36         return asm_system_call(2);
37     }
38
39     int syscall_fork() {
40         return programManager.fork();
41     }^?
42
43
44

remote Thread 1 In: fork
(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) s
fork () at ../src/kernel/syscall.cpp:37
(gdb) p $eax
$1 = 2
(gdb)
```

```
../src/kernel/syscall.cpp
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
44

remote Thread 1 In: fork
Continuing.

Breakpoint 9, asm_system_call () at ../src/utils/asm_ut
(gdb) s
fork () at ../src/kernel/syscall.cpp:37
(gdb) p $eax
$2 = 0
(gdb)
```

回答：父进程执行完 ProgramManager::fork 后返回值是子进程的pid（放在\$eax中），而子进程在赋值时childpps->eax = 0，相当于利用函数调用返回值的规范，底层直接修改了返回值，使返回值为零

5. 请解释在 ProgramManager::schedule 中，我们是如何从一个进程的虚拟地址空间切换到另外一个进程的虚拟地址空间的。

回答：调用activateProgramPage(next);切换页目录表，更新TSS中的特权级0的栈。

# Assignment 4 wait & exit 的实现

复现截图：

```
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
thread exit
exit, pid: 3
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
```

1. 请结合代码逻辑和具体的实例来分析 exit 的执行过程。

回答：



作为系统调用，`exit` 从发起调用到由内核执行函数过程类似与所有系统调用类似，都是通过 `asm_system_call` 调用参数（`exit` 为 3）然后执行 `int 80h`，查中断描述符表找到函数入口，开始执行 `ProgramManager::exit` 函数。

`ProgramManager::exit` 函数步骤如下：

（1）标记PCB状态为DEAD并放入返回值。

（2）如果PCB标识的是进程，则（按顺序）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。

（3）立即执行线程/进程调度

2. 请解释进程退出后能够隐式调用 `exit` 的原因。（tips：从栈的角度分析）

回答：

只要在进程的3特权级栈的顶部放入`exit`的地址和参数即可，当执行进程的函数退出后就会主动跳转到`exit`

```
// 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;
```

3. 请结合代码逻辑和具体的实例来分析 `wait` 的执行过程。

回答：

作为系统调用，`wait` 从发起调用到由内核执行函数过程类似与所有系统调用类似，都是通过 `asm_system_call` 调用参数（`exit` 为 4）然后执行 `int 80h`，查中断描述符表找到函数入口，开始执行 `ProgramManager::wait` 函数。

`ProgramManager::wait` 函数步骤如下：

(1)首先，`wait` 函数会关闭中断，以确保在查找子进程和更新其状态时不会被打断。

(2)然后，使用一个 `while` 循环来遍历所有进程，查找当前进程的子进程。如果找到了子进程，就检查子进程的状态是否为 `DEAD`。如果是，就根据需要设置 `retval` 并释放 `PCB`，并返回子进程的进程 `ID`。

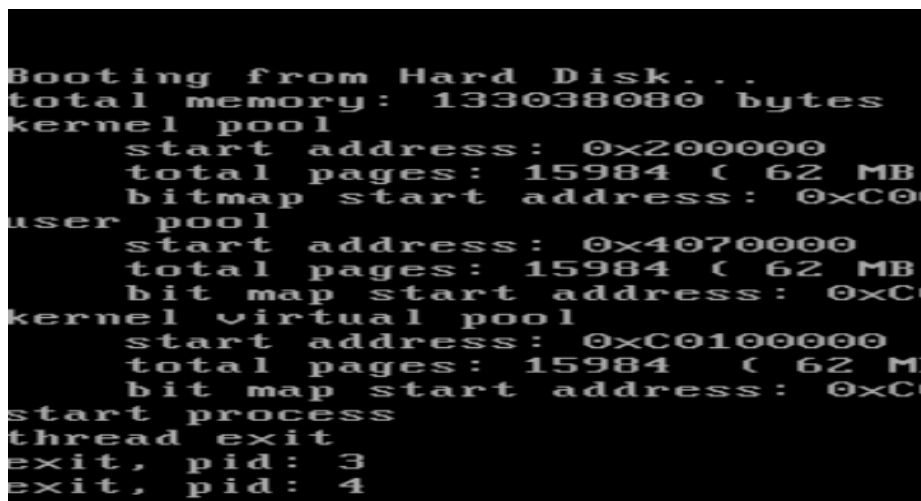
(3)如果没有找到可返回的子进程，则判断是否存在子进程。如果没有子进程，说明它们已经返回，并且当前进程可以继续执行（结束循环）。如果有子进程，但他们的状态不是 **DEAD**，则程序会调用 **schedule()** 函数来切换到其他进程，直到某个子进程结束并返回。

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。

## 构造僵尸进程

方法：将父进程的wait操作删去即可

## 截图



## 回收僵尸进程

思路：创建一个新的系统调用专门来回收僵尸进程，系统调用函数先关中断，然后遍历所有PCB模块，遇到内核模块就跳过，当遇到的PCB模块状态为**DEAD**时，就释放该模块，这样就成功回收僵尸进程

## 关键代码

## 系统调用函数

```
int ProgramManager::delete_zombie()
{
    // 关中断
    bool interrupt;
    interrupt = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();
```

```

ListItem *item;
PCB *itemPCB;

item = this->allPrograms.head.next;
//遍历所有PCB
while (item)
{
    itemPCB = ListItem2PCB(item, tagInAllList);
    // 跳过内核模块
    if (itemPCB->pageDirectoryAddress)
    {
        if (itemPCB->status == ProgramStatus::DEAD)
        {
            printf("Successfully delete zombie process [Pid: %d]\n", itemPCB-
>pid);
            releasePCB(itemPCB);
        }
    }
    item = item->next;
}
interruptManager.setInterruptStatus(interrupt);
return 0;
}

```

结果截图

```

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
thread exit
exit, pid: 3
exit, pid: 4
Successfully delete zombie process [Pid: 3]
Successfully delete zombie process [Pid: 4]

```

## 5. 总结(对实验过程中遇到的问题进行总结，可以提出对实验设置的改进意见)

本次实验与理论课结合比较紧密，大致内容比较容易看懂，但是理论课上看似很容易的函数跳转，一到写汇编代码，就需要考虑许多，比如压入栈的顺序、栈的结构等，哪怕是给了正确的代码，理解起来也是十分费劲，这次实验gdb发挥了较大的作用，跟着gdb一步步走从而知道了系统调用的过程，比如看似简单的一个fork()函数，它需要先调用asm\_system\_call，进入内核态，然后在内核态中再调用programManager.fork()，最后再一步步返回

## 6. 参考资料清单

---