1 系统调用

虽然在第一次报告中提到过系统调用，但是由于当时自己了解地不够全面，所以这次重新完整地学习系统调用。

1.1 系统调用的过程

用户空间的程序无法直接执行内容代码，也就无法直接调用内核空间中的函数。系统通过以下方式实现系统调用。

- 应用程序以软中断的方式通知内核自己需要一个系统调用。

  在 x86 上，软中断的中断向量号为 128，所以可以通过 int $0x80 指令触发软中断。int $0x80 之后，系统切换到内核态，并执行第 128 号异常处理程序 system_call()。这个程序在 entry_64.S 文件中用汇编语言编写。

- 内核代表应用程序在内核空间执行系统调用。

  在陷入内核前，应用程序将相应的系统调用号放入 eax 中，当系统调用处理程序运行时，可能直接从 eax 中得到数据。在执行相应的系统调用前，需要验证系统调用号的有效性。实现代码如下。

```
1    cmpl $__NR_syscall_max, %eax
2    ja 1f ; 如果系统调用号大于__NR_syscall_max，则返回—ENOSYS
```

  如果系统调用号有效，就执行系统调用表中相应的系统调用。

```
1    ; %rax存放着系统调用号
2    ; 因为是64位系统，系统调用表中的表项是以8字节类型存放的，所以需要将给定的
         系统调用号乘以8
3    call *sys_call_table(,%rax,8)
```

上述过程只是讲述了内核空间如何执行系统调用。除此之外，还需要明白应用程序在用户空间是怎么以软中断的方式通知内核自己需要一个系统调用。

Linux 在 unistd.h 中提供了一组宏，用于直接对系统调用进行访问。这组宏可以设置好寄存器并调用陷入指令。实现代码如下。

```
1    #define _syscall0(type, name) \
2    type name(void) \
3    {\
4        long __res; \
5        __asm__ volatile(
6            "int $0x80" \
7            : "=a" (__res)
8            : "0" (__NR_##name)); \
9        if(__res >= 0) \
```

```
10              return (type) __res; \
11          errno = -__res; \
12          return -1;\
13      }
```

假设现在有一个系统调用 foo()，我们就可以用下面的方式在用户空间调用它。

```
1   _syscall0(long, foo)
2   // 这里的宏会展开为 long foo(){/*...*/}
3
4   int main()
5   {
6       // ...
7       foo();
8       //...
9       return 0;
10  }
```

需要注意的是，系统不只拥有 _syscall0() 这一个宏，它提供了一组宏，这组宏是 _syscalln()。这里的 n 代表了需要传递给系统调用的参数个数。宏所需要的参数个数是 2+2n，第一个参数是系统调用函数返回类型，第二个参数是系统调用名称，随后是系统调用每个参数的类型和名称。

1.2 注册系统调用的步骤

下面通过将 foo() 加入为系统调用来说明这个步骤。

- 首先，需要在系统调用表的最后加入一个表项。现在要把 foo() 注册为一个正式的系统调用，就要把 sys_foo 加入到系统调用表中。实现代码如下：

```
1   ENTRY(sys_call_table)
2       .long sys_restart_syscall
3       ;...
4       .long sys_perf_event_open ; 最后一项
5       ; 将sys_foo加到这个表的末尾
6       .long sys_foo
```

因为我对 linux 汇编还不是很熟悉，所以去查询了以下几条指令的意思：

```
1   ENTRY()
2   ; 这是一个宏，定义于linux-2.6.35.5/include/Linux/linkage.h。格式如下：
3   #define ENTRY(name) \
4   .globl name \
5   ALIGN \
6   name:
7
8   ; .globl symbol 的含义是：
9   ; 定义该symbol为global的，也就是其他文件可以访问并使用该symbol
10
11  ; .long val 的含义是：
```

```
12      ; 该指令在当前区定义一个32位长整数的常数。需要注意该指令无法在bss段中使用
```

- 接下来，将 foo() 的系统调用号加到 <arch/arm/include/uapi/asm/unistd.h> 中。实现代码如下：

```
1   #if defined(__thumb) || defined(__ARM_EABI)
2   #define __NR_SYSCALL_BASE 0
3   #else
4   #define __NR_SYSCALL_BASE __NR_OABI_SYSCALL_BASE
5   #endif
6
7   #define __NR_restart_syscall (__NR_SYSCALL_BASE + 0)
8   // ...
9   // 最后一个系统调用号
10  #define __NR_pkey_free (__NR_SYSCALL_BASE + 396)
11  // 在最后一行添加foo()的系统调用号
12  #define __NR_foo (__NR_SYSCALL_BASE + 397)
```

- 最后在内核代码中定义这个系统调用函数。

```
1   asmlinkage long sys_foo()
2   {
3       return THREAD_SIZE;
4   }
5
6   // asmlinkage在linkage.h中有定义
7   #ifdef CONFIG_X86_32
8   #define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
9   #endif
10  // __attribute__((regparm(0)))告诉编译器参数只能通过堆栈来传递
11  // X86里面的系统调用都是先将参数压入stack以后调用sys_*函数的，所以必须告诉
        编译器只能通过堆栈传递参数
```

1.3 学习系统调用后的反思

在前面几个小节，我学会了如何在操作系统中新建一个系统调用。这一开始让我兴奋，随后而来的是疑惑，因为我并不了解系统是如何支持系统调用这个行为的。我也不了解 int $0x80 之后，系统具体发生了什么。我觉得这还需要我学习了系统的中断机制和特权保护机制后，知道如何去实现它们后，才能真正地理解系统调用。

2 LDT 的实现

2.1 LDT 的定义

首先我们需要在 GDT 表中增加局部描述符表的描述符。

```
        [SECTION .gdt]
LABEL_GDT: Descriptor 0,LDT
; 添加的描述符
LABEL_DESC_LDT: Descriptor 0,LDTLen-1,DA_LDT
; 相应的选择符
SelectorLDT equ LABEL_DESC_LDT - LABEL_GDT
```

然后我们需要定义一个 LDT 表。LDT 表和 GDT 表其实很类似，我在其中定义了一个指向 CODEA 代码段的段描述符。

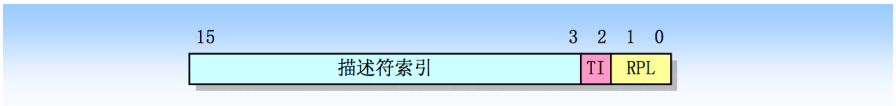
```
        ; 定义LDT表
        [SECTION .ldt]
        ALIGN 32
        LABEL_LDT:
        LABEL_LDT_DESC_CODEA: Descriptor 0,CodeALen-1,DA_C+DA_32
        LDTLEN equ $-LABEL_LDT

        SelectorLDTCodeA equ LABEL_LDT_DESC_CODEA-LABEL_LDT+4
        LDTLen equ $-LABEL_LDT

        ; 定义在LDT表中段描述符指向的代码段
        [SECTION .la]
        ALIGN 32
        [BITS 32]
        LABEL_CODE_A:
            mov ax,SelectorVideo
            mov gs,ax
            mov edi,(80*12+0)*2
            mov ah,0Ch
            mov al,'L'
            mov [gs:edi],ax
        CodeALen equ $-LABEL_CODE_A
```

上面代码段中，我们应该注意的是这个语句。

```
        SelectorLDTCodeA equ LABEL_LDT_DESC_CODEA-LABEL_LDT+4
```

这个选择符的定义和 GDT 选择符的定义有不同。在解释为什么这么写之前，我想重新说一下自己对段选择符的认识。因为在第二次学习报告中，我对它的描述实在过于粗糙。

段选择符是段的一个 16 位标志符。段选择符并不直接指向段，而是指向段描述符表中定义段的段描述符。段选择符的结构如下图。



从图中可以看出，段选择符有 3 个字段内容:

- 请求特权级 RPL。RPL 被用于特权级保护机制中。
- 表指示标志 TI。当 TI=0 时，表示描述符在 GDT 中。当 TI=1 时，表示描述符在 LDT 中。因为一个任务执行时，可以同时访问到 LDT 和 GDT，所以必须做这样的区别，以防在索引时放生混淆。
- 索引值。用于索引在 GDT 表或 LDT 表中的段描述符。

```
1        ; 这里特意加4，就是为了将段选择符中的第2位TI标志置一
2        SelectorLDTCodeA  equ  LABEL_LDT_DESC_CODEA-LABEL_LDT+4
```

2.2 LDT 的初始化

需要注意的是，既然在 GDT 表中添加了指向 LDT 表的段描述符，就应该在 16 位代码段中初始化它。

```
1        [SECTION  .16]
2        [BITS  16]
3
4            ; 初始化LDT在GDT中的描述符
5            xor  eax,eax
6            mov  ax,ds
7            shl  eax,4
8            add  eax,LABEL_LDT
9            mov  word  [LABEL_DESC_LDT + 2],  ax
10           shr  eax,16
11           mov  byte  [LABEL_DESC_LDT + 4],  al
12           mov  byte  [LABEL_DESC_LDT + 7],  ah
```

LDT 表和 GDT 表区别仅仅在于全局和局部的不同，所以初始化 LDT 表中描述符和之前的操作很类似。具体情况看下面的代码。

```
1        [SECTION  .16]
2        [BITS  16]
3            ; 初始化LDT表中的描述符
4            xor  eax,eax
5            mov  ax,ds
6            shl  eax,4
7            add  eax,LABEL_CODE_A
8            mov  word  [LABEL_LDT_DESC_CODEA + 2],  ax
9            shr  eax,16
10           mov  byte  [LABEL_LDT_DESC_CODEA + 4],  al
```

```
11          mov byte [LABEL_LDT_DESC_CODEA + 7], ah
```

2.3 调用 LDT 中的代码段

```
1       [SECTION .s32]
2       [BITS 32]
3           ; 将LDT表的段选择符加载进LDTR寄存器中
4           ; 在LDTR寄存器中的LDT段描述符存放着LDT表的基址
5           ; 对LDT表中的代码段描述符进行寻址时，以LDTR中LDT表的描述符中的基址为准
6           ; 偏移量由段选择符制定，用于索引LDT表中存放着的代码段描述符
7           ; 当发生任务切换时，LDTR会更换新任务的LDT
8           mov ax, SelectorLDT
9           lldt ax
10          ; CPU根据代码段描述符中的TI标志判断是索引GDT表还是索引LDT表
11          jmp SelectorLDTCodeA:0
```

2.4 总结如何添加 LDT 表

- 添加一个 LDT 表，里面可以类似于 GDT 表，存放代码段、数据段或堆栈段。
- 在 GDT 表中添加新 LDT 表的段描述符。
- 在 16 位代码段中初始化新 LDT 表的段描述符。同时初始化 LDT 表中存放着的所有段描述符。

3 保护机制的实现

3.1 保护机制的相关介绍

虽然我在第二次学习报告中也提到了保护机制，但是过于粗糙。我自己感觉对它的认识也不够深入，所以在这里重新整理一下对保护机制的认识。

3.1.1 段级保护

段限长检查。段描述符的段限长字段用于防止程序寻址到段外内存位置。当 G=0 时，段限长最大为 1MB。当 G=1 时，段限长最大为 4GB。除了检查段限长，处理器也会检查描述符表的长度。GDTR、IDTR 和 LDTR 寄存器都包含有描述符的限长值，用于防止程序在描述符表的外面选择描述符。

段类型检查。段描述符中有 S 标志和 TYPE 字段用于标识段的类型。S=1 时，为系统类型的段。S=0 时，为代码或数据类型的段。TYPE 字段的 4 个比特位用于定义代码、数据和系统描述符的各种类型。处理器在以下两种情况会检查段的类型信息:

- 当段选择符加载进一个段寄存器时需要检查段的类型，因为某些段寄存器只能存放特定类型的描述符。
- 一些指令操作不被允许在某些类型的段上执行。比如任何指令不能写一个可执行段。

特权级检查。段保护机制有 4 个特权级，由段描述符的 DPL 字段中定义。另外还需要知道，CPL 是当前任务的特权级，RPL 是段描述符的特权级。特权级检查可以分为 3 种情况。

- 访问数据段时的特权级检查。

  为了访问数据段中的操作数，数据段中的段选择符需要加载进数据段寄存器或堆栈寄存器中，此时就需要特权级检查。处理器会比较 CPL、RPL 和 DPL，只有当 DPL 的数值大等于 CPL 和 RPL 两者时，处理器才会把选择符加载进段寄存器，否则就会产生一个一般保护异常。需要注意的是，当使用堆栈选择符加载 SS 段寄存器时，DPL、RPL 和 CPL 都需要相同，否则就会产生一个一般保护异常。

- 直接调用或跳转到代码段时的特权级检查。

  首先说明一下一致代码段与非一致代码段的概念。一致代码段允许当前特权级任务跳转到更高特权级的代码段，并且当前任务的 CPL 也会设置为更高特权级代码段的 DPL 值。而非一致代码段不允许这样的行为。段描述符中的 C 标志用于标识一致代码段和非一致代码段。C=1 时，为一致代码段。C=0 时，为非一致代码段。需要注意的是，一致代码段和非一致代码段都不允许当前任务跳转到更低特权级的代码段。

当访问一致代码段时，处理器会忽略对 RPL 的检查，只要求 CPL 大等于 DPL。当访问非一致代码段时，CPL 必须等于 DPL，而 RPL 必须小等于 DPL。

- 通过调用门访问代码段时的特权级检查。

  因为处理器通过调用门描述符去访问代码段描述符，所以还需要对调用门描述符进行特权级检查。处理器要求 CPL 和 RPL 都要小于调用门描述符的 DPL。随后进行对代码段的特权级检查。当使用 CALL 指令时，对于一致代码段和非一致代码段都只要求 DPL 小等于 CPL。当使用 JMP 指令时，对于一致代码段要求 DPL 小等于 CPL，对于非一致代码段要求 DPL 等于 CPL。

3.1.2 页级保护

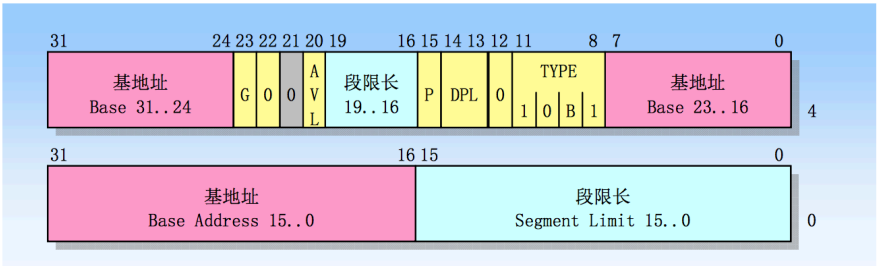
只有当所有的段级保护通过后，才会进行页级保护检查。

页级保护分为两类权限，分别为超级用户级和普通用户级。特权级 0、1、2 的任务被归类为超级用户级，特权级 3 的任务被归类为普通用户级。页面也分为超级用户级和普通用户级。页目录和页表项中的 U/S 标志用于标识该页面的级别。只有当两者的 U/S 都为 0，页面才是超级用户级，否则是普通用户级。普通用户级的程序不可访问超级用户级的页面，超级用户级的程序可以读/写/执行所有页面。

普通用户级的程序只能访问普通用户级的页面，但是不一定可以写。只有当页面和页目录的读写标志 R/W 都为 1 时，普通用户级的程序才能写普通用户级的页面。

3.2 编写保护机制

在编写保护机制前，我觉得有必要先把段描述符的属性搞清楚。先来回忆一下段描述符的格式，如下图。



可以清楚地看到，描述符的第 6 和第 7 个字节是属性与段限长的混合。对这 16 位进行赋值。假设有 20 位的段限长，首先给描述符低 16 位的段限长赋值，代码为 LEN & 0FFFFh。然后将其右移 8 位，剩 12 位，最后取剩余的 12 位的高 4 位，并对第 7 字节中的段限长字段赋值。代码为 ((LEN » 8) & 0F00h)。现在第 6 和第 7 字节仅剩属性还未赋值，代码为 ATTR & 0F0FFh。现在重温一下段描述符的数据结构，我觉得现在对这个数据结构的理解完全掌握了。

```
%marco Descriptor 3
    dw %2 & 0FFFFh
    dw %1 & 0FFFFh
    dw (%1 >> 16) & 0FFh
    dw ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh)
    db (%1 >> 24) & 0FFh
%endmarco
```

3.2.1 对段描述符属性的编程

根据宏定义，段的属性由第三个参数决定。下面定义一些宏，用于定义段的属性。在代码中我会注释为什么这么定义宏。

```
; 第三个参数的高4位分别为G、D/B、保留比特位和AVL位。
; 4h=0100b，相当于将D/B设置为1。
; 对于代码段，这个标志用于指出该段中的指令引用有效地址和操作数的默认长度。
; D/B为1时，有效地址为32位，操作数长度为32位或8位。
; D/B为0时，有效地址为16位，操作数长度为16位或8位。
; DA_32代表段为32位段
DA_32 EQU 4000h

; 第三参数的低8位为P、DPL、S和TYPE。
; P表示段是否存在在内存中
; DPL表示段描述符的特权级
; S表示该段是系统段描述符或门描述符还是代码或数据段

; 给出存储段描述符的属性

; TYPE有4位，最高位为0时，为数据段，最高位为1时，为代码段。
; 为数据段时，TYPE为0、E(扩展方向)、W(可写)和A(已访问)。
; 为代码段时，TYPE为1、C(一致性)、R(可读)和A(已访问)。
; 后面会有一张表，用于整理TYPE的各种情况。

; 数据段
DA_DR EQU 90h ; 存在的只读数据段
DA_DRA EQU 91h ; 存在的已访问只读数据段
DA_DRW EQU 92h ; 存在的可读写数据段
DA_DRWA EQU 93h ; 存在的已访问的可读写数据段
DA_C EQU 98h ; 存在的仅执行代码段

; 代码段
DA_CA EQU 99h ; 存在的已访问的仅执行代码段
DA_CR EQU 9Ah ; 存在的可执行可读代码段
DA_CCO EQU 9Ch ; 存在的可执行的一致代码段
DA_CCOR EQU 9Eh ; 存在的可执行的可读一致代码段

; 给出系统段描述符的属性

; 此时TYPE有16种情况，除了3个保留情况以外，有13种段描述符
; 后面会有一张表，用于整理系统段描述符的各种情况

DA_LDT EQU 82h ; LDT表描述符
DA_TaskGate EQU 85h ; 任务门描述符
```

```
41      DA_386TSS EQU 89h ; 32位TSS描述符
42      DA_386CGate EQU 8Ch ; 32位调用门描述符
43      DA_386IGate EQU 8Eh ; 32位中断门描述符
44      DA_386TGate EQU 8Fh ; 32位陷阱门描述符
```

下面是各种类型的代码段描述符和数据段描述符。

类型（TYPE）字段					描述符类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已访问

下面是各种类型的系统段描述符和门描述符。

类型（TYPE）字段					说明	
十进制	位 11	位 10	位 9	位 8		
0	0	0	0	0	Reserved	保留
1	0	0	0	1	16-Bit TSS (Available)	16 位 TSS（可用）
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-Bit TSS (Busy)	16 位 TSS（忙）
4	0	1	0	0	16-Bit Call Gate	16 位调用门
5	0	1	0	1	Task Gate	任务门
6	0	1	1	0	16-Bit Interrupt Gate	16 位中断门
7	0	1	1	1	16-Bit Trap Gate	16 位陷阱门
8	1	0	0	0	Reserved	保留
9	1	0	0	1	32-Bit TSS (Available)	32 位 TSS（可用）
10	1	0	1	0	Reserved	保留
11	1	0	1	1	32-Bit TSS (Busy)	32 位 TSS（忙）
12	1	1	0	0	32-Bit Call gate	32 位调用门
13	1	1	0	1	Reserved	保留
14	1	1	1	0	32-Bit Interrupt Gate	32 位中断门
15	1	1	1	1	32-Bit Trap Gate	32 位陷阱门