

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。

```

1      [SECTION .gdt]
2      LABEL_GDT: Descriptor 0,LDT
3      ; 添加的描述符
4      LABEL_DESC_LDT: Descriptor 0,LDTLen-1,DA_LDT
5      ; 相应的选择符
6      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。

```

1      ; 定义LDT表
2      [SECTION .ldt]
3      ALIGN 32
4      LABEL_LDT:
5      LABEL_LDT_DESC_CODEA: Descriptor 0,CodeALen-1,DA_C+DA_32
6      LDTLEN equ $-LABEL_LDT
7
8      SelectorLDTCodeA equ LABEL_LDT_DESC_CODEA-LABEL_LDT+4
9      LDTLen equ $-LABEL_LDT
10
11     ; 定义在LDT表中段描述符指向的代码段
12     [SECTION .la]
13     ALIGN 32
14     [BITS 32]
15     LABEL_CODE_A:
16         mov ax,SelectorVideo
17         mov gs,ax
18         mov edi,(80*12+0)*2
19         mov ah,0Ch
20         mov al,'L'
21         mov [gs:edi],ax
22     CodeALen equ $-LABEL_CODE_A

```

上面代码段中，我们应该注意的是这个语句。

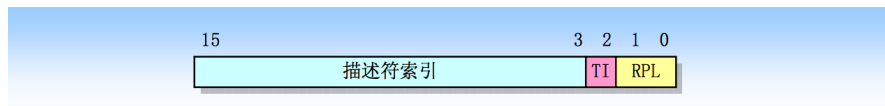
```

1      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 保护机制的相应理论

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。