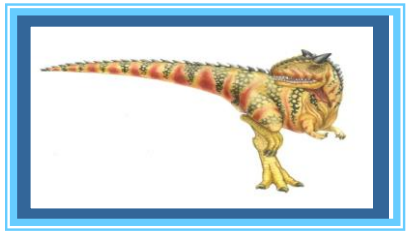
系统调用



本章内容

- 系统调用基础知识
- 数据结构和代码
- 系统调用getuid()的实现
- 添加一个系统调用mysyscall

一个简单的例子

```
#include <linux/unistd.h> /* all sysystem calls need this header */
int main(){
        int  i = getuid();
        printf("Hello World! This is my uid: %d\n", i);
}
```

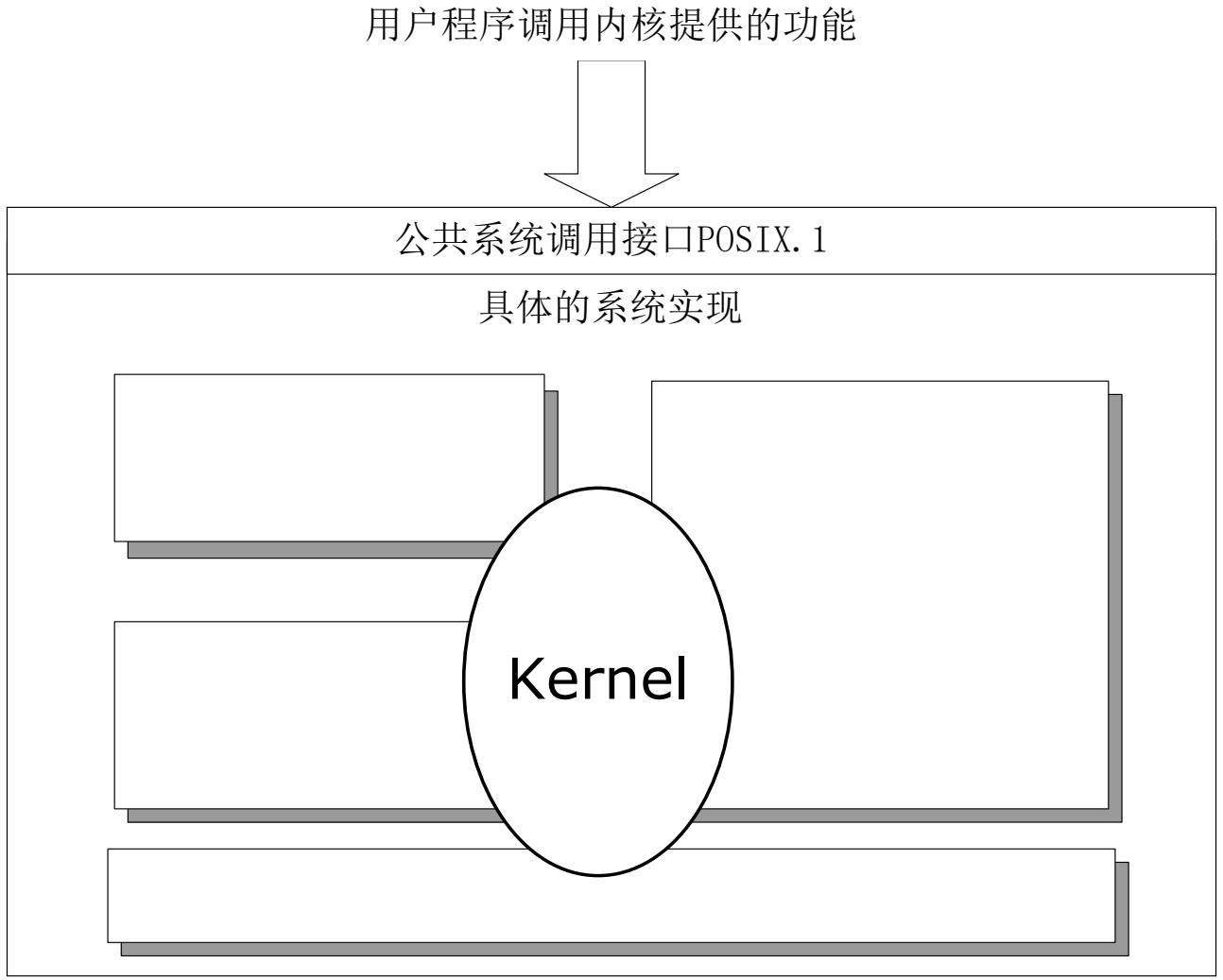
普通函数？

- 第1行：包括unistd.h这个头文件。所有用到系统调用的程序都需要包括它。
- 第2行：进行getuid()系统调用，并将返回值赋给变量i

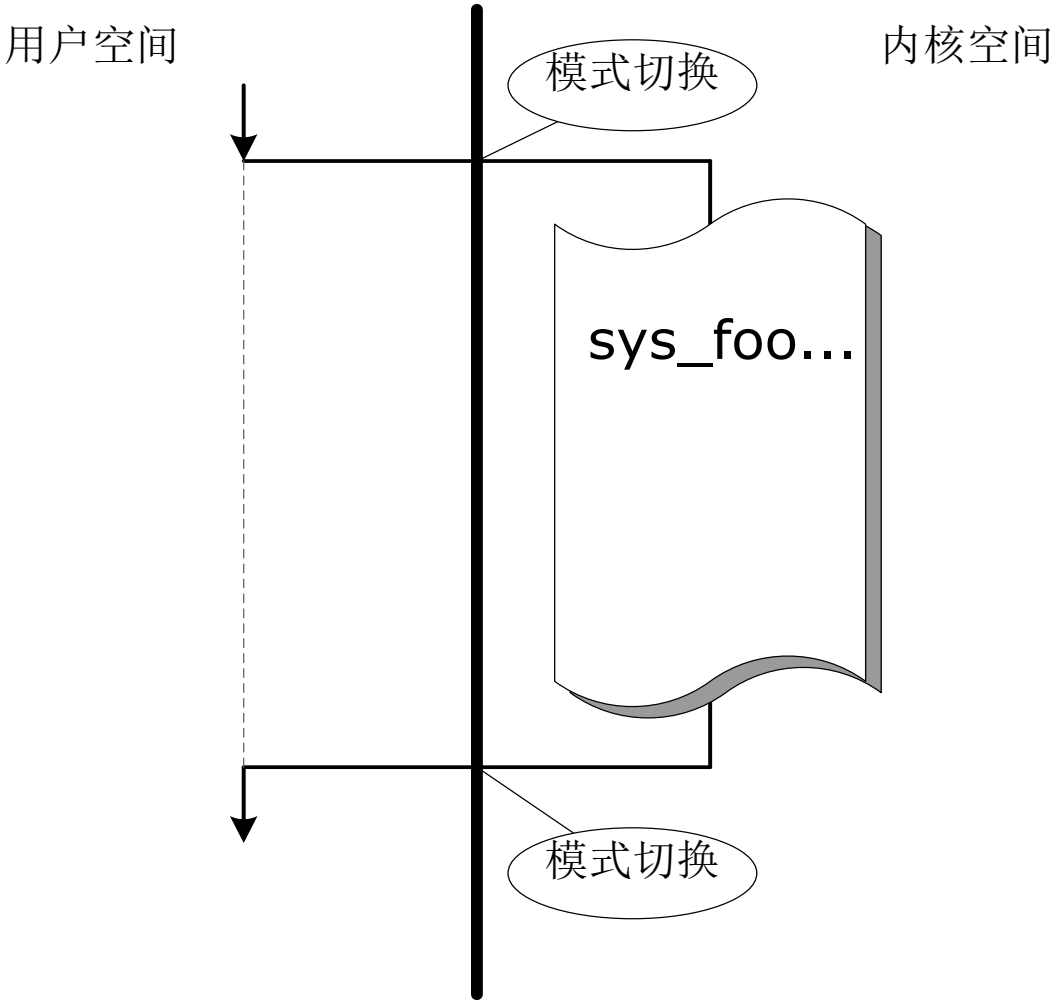
为什么需要系统调用

- 系统调用是内核向用户进程提供服务的唯一方法，应用程序调用操作系统提供的功能模块（函数）。
- 用户程序通过系统调用从用户态（user mode）切换到核心态（kernel mode），从而可以访问相应的资源。这样做的好处是：
  - 为用户空间提供了一种硬件的抽象接口，使编程更加容易。
  - 有利于系统安全。
  - 有利于每个进程度运行在虚拟系统中，接口统一有利于移植。

为什么需要系统调用(续)



为什么需要系统调用(续)



运行模式、地址空间、上下文（for x86）

- 运行模式（mode）
  - Linux使用了其中的两个：特权级0和特权级3，即内核模式(kernel mode)和用户模式(user mode）
- 地址空间（space）
  - 每个进程的虚拟地址空间可以划分为两个部分：用户空间和内核空间。
  - 在用户态下只能访问用户空间；而在内核态下，既可以访问用户空间，又可以访问内核空间。
  - 内核空间在每个进程的虚拟地址空间中都是固定的（虚拟地址为3G～4G的地址空间）。

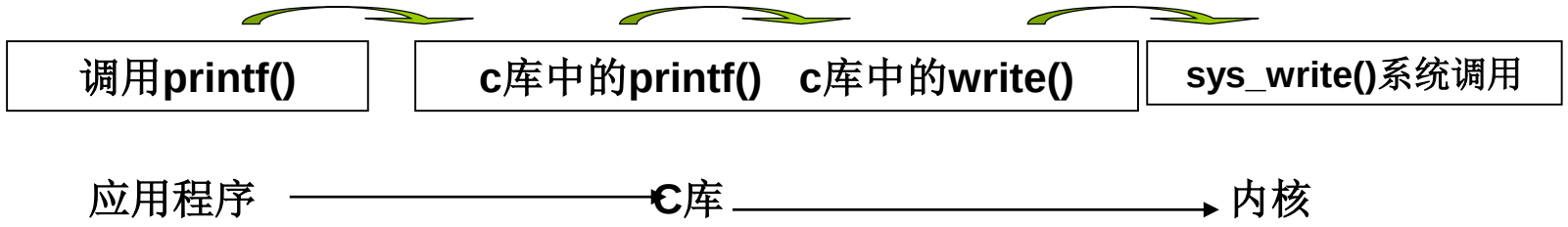
运行模式、地址空间、上下文

- 上下文（context）。一个进程的上下文可以分为三个部分：用户级上下文、寄存器上下文以及系统级上下文。
  - 用户级上下文：正文（代码）、数据、用户栈以及共享存储区；
  - 寄存器上下文：通用寄存器、程序寄存器（eip）、处理机状态寄存器（eflags）、栈指针（esp）；
  - 系统级上下文：进程控制块task_struct、内存管理信息(mm_struct、vm_area_struct、 pgd、pmd、pte等)、核心栈等。

传统Unix这样划分

系统调用、API和C库

- Linux的应用编程接口（API）遵循 POSIX标准
- Linux的系统调用作为c库的一部分提供。c库中实现了Linux的主要API，包括标准c库函数和系统调用。
- 应用编程接口(API)其实是一组函数定义，这些函数说明了如何获得一个给定的服务；而系统调用是通过软中断向内核发出一个明确的请求，每个系统调用对应一个封装例程（wrapper routine，唯一目的就是发布系统调用）。一些API应用了封装例程。
  - API还包含各种编程接口，如：C库函数、OpenGL编程接口等
- 系统调用的实现是在内核完成的，而用户态的函数是在函数库中实现的



系统调用与操作系统命令

- 操作系统命令相对API更高一层，每个操作系统命令都是一个可执行程序，比如ls、hostname等，
- 操作系统命令的实现调用了系统调用
- 通过strace命令可以查看操作系统命令所调用的系统调用，如：
  - strace ls
  - strace -o log.txt hostname

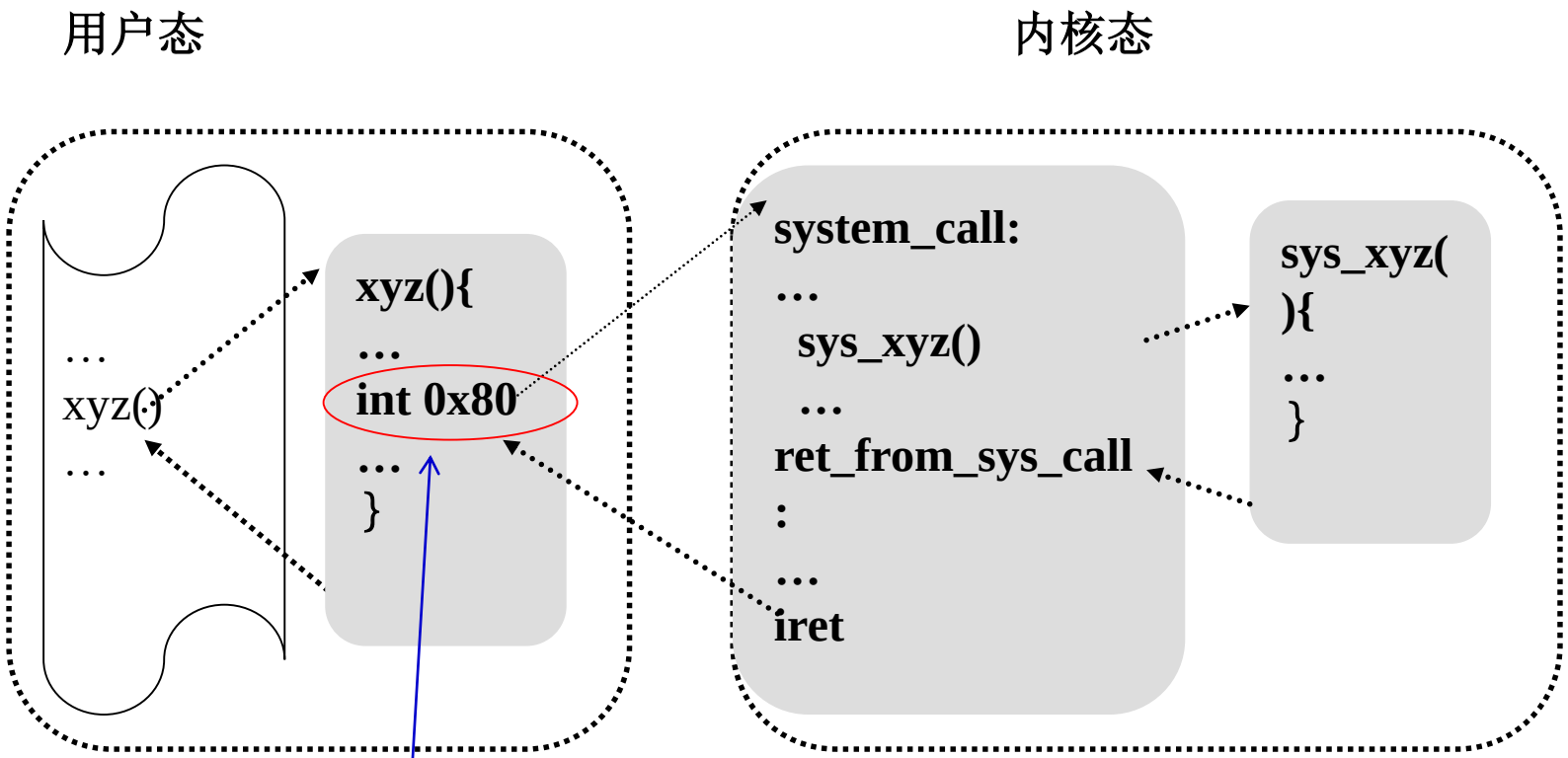
系统调用与内核函数

- 内核函数在形式上与普通函数一样，但它是在内核实现的，需要满足一些内核编程的要求
- 系统调用是用户进程进入内核的接口层，它本身并非内核函数，但它是由内核函数实现的
- 进入内核后，不同的系统调用会找到各自对应的内核函数，这些内核函数被称为系统调用的“服务例程”

系统调用处理程序及服务例程（x86）

- 当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数
- **系统调用(中断)处理程序**执行下列操作：
  - 在内核栈保存大多数寄存器的内容
  - 调用名为系统调用服务例程（system call service routine）的相应的C函数来处理系统调用
  - 通过ret_from_sys_call( )函数从系统调用返回

调用一个系统调用（x86）



初始化系统调用

- 内核初始化（操作系统启动）期间调用trap_init()函数建立IDT表中128(0x80)号向量对应的表项：
  - set_system_gate(0x80， &system_call);
- 该调用把下列值装入该门描述符的相应域：
  - segment selector：内核代码段__KERNEL_CS的段选择符
  - offest：指向system_call( )异常处理程序
  - type：置为15，表示该异常是一个陷入
  - DPL（描述符特权级）：置为3，这就允许用户态进程调用这个异常处理程序

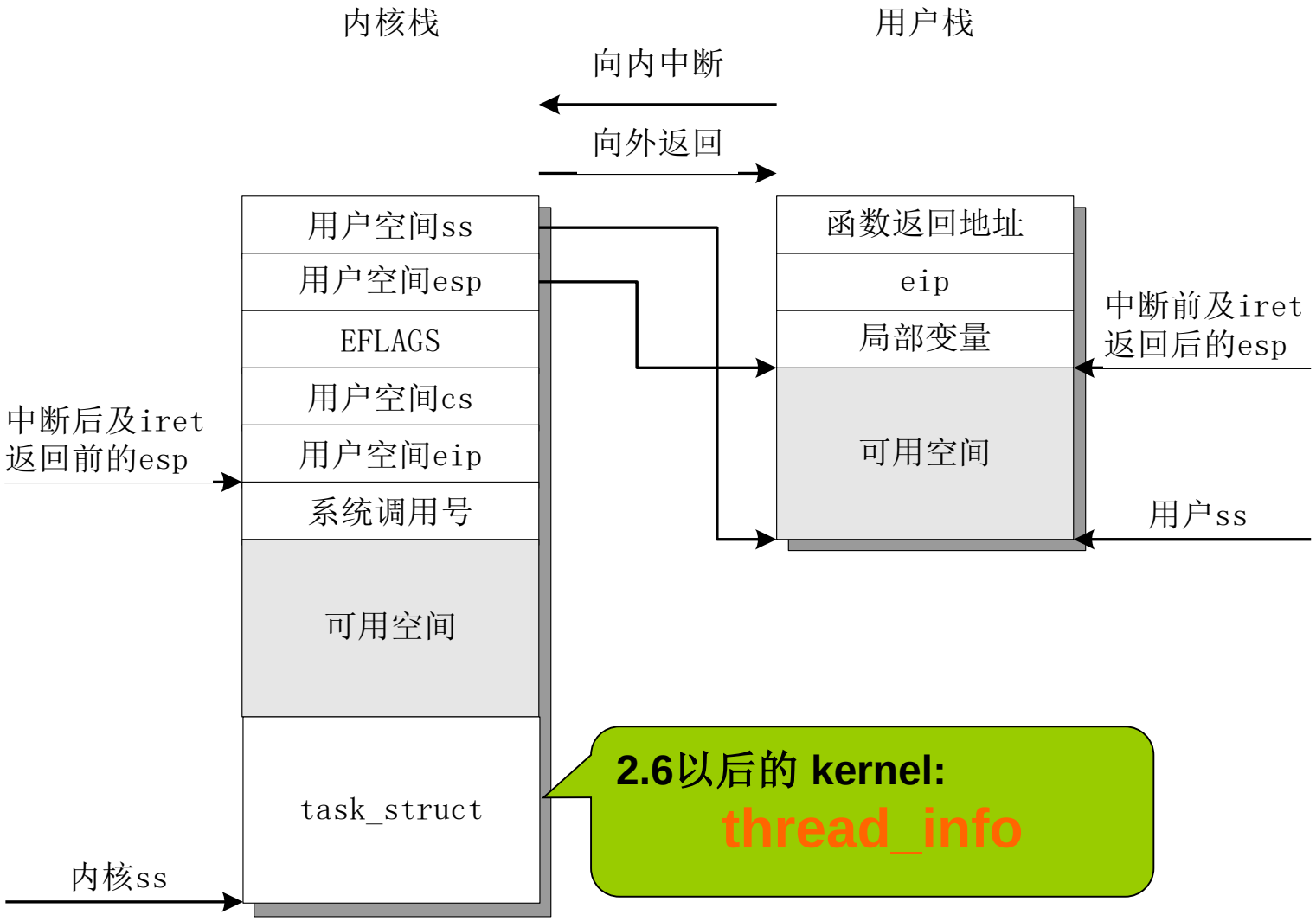
为 int 0x80 做好了准备

数据结构和代码

- 与系统调用相关的内核代码文件：
  - arch/x86/kernel/entry.S（entry_32.S、entry_64.S）
    - 系统调用时的内核栈
    - sys_call_table
    - system_call和ret_from_sys_call
  - arch/x86/kernel/traps.c //异常处理程序
  - include/linux/unistd.h
    - 系统调用编号
    - 宏定义展开系统调用

V4.x: arch/x86/entry/entry_32.S
arch/x86/entry/syscalls/syscall_32.tbl

系统调用时的内核栈


内核栈
用户栈
向内中断
向外返回
用户空间ss
用户空间esp
EFLAGS
用户空间cs
用户空间eip
系统调用号
可用空间
task_struct
函数返回地址
eip
局部变量
可用空间
中断后及iret
返回前的esp
中断前及iret
返回后的esp
用户ss
内核ss
2.6以后的 kernel:
thread_info

系统调用时的内核栈(续)

arch/x86/kernel/entry.S

或 arch/x86/entry/entry_32.S

```
18  * Stack layout in 'ret_from_system_call':
19  *     ptrace needs to have all regs on the stack.
20  *     if the order here is changed, it needs to be
21  *     updated in fork.c:copy_process, signal.c:do_signal,
22  *     ptrace.c and ptrace.h
23  *
24  *      0(%esp) - %ebx
25  *      4(%esp) - %ecx
26  *      8(%esp) - %edx
27  *      C(%esp) - %esi
28  *     10(%esp) - %edi
29  *     14(%esp) - %ebp
30  *     18(%esp) - %eax
31  *     1C(%esp) - %ds
32  *     20(%esp) - %es
33  *     24(%esp) - orig_eax
34  *     28(%esp) - %eip
35  *     2C(%esp) - %cs
36  *     30(%esp) - %eflags
37  *     34(%esp) - %oldesp
38  *     38(%esp) - %oldss
39  *
40  * "current" is in register %ebx during any slow entries.
```

system_call()函数

system_call()函数实现了系统调用中断处理程序：

1. 它首先把系统调用号和该异常处理程序用到的所有CPU寄存器保存到相应的栈中， SAVE_ALL
2. 把当前进程task_struct （thread_info）结构的地址存放在ebx中
3. 对用户态进程传递来的系统调用号进行有效性检查。若调用号大于或等于NR_syscalls，系统调用处理程序终止。 （sys_call_table）
4. 若系统调用号无效，函数就把-ENOSYS值存放在栈中eax寄存器所在的单元，再跳到ret_from_sys_call()
5. 根据eax中所包含的系统调用号调用对应的特定服务例程

system_call (续)

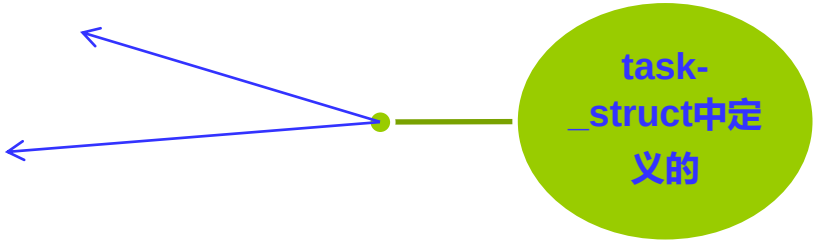
arch/x86/kernel/entry.S

```
194 ENTRY(system_call)
195     pushl %eax                      # save orig_eax
196     SAVE_ALL
197     GET_CURRENT(%ebx)/*获取当前进程的task_strut指针
198     testb $0x02,tsk_ptrace(%ebx)  # PT_TRACESYS
199     jne tracesys
200     cmpl $(NR_syscalls),%eax
201     jae badsys
202     call *SYMBOL_NAME(sys_call_table)(,%eax,4)
203     movl %eax,EAX(%esp)  # save the return value
```

2.6~: GET_THREAD_INFO(%ebp)

system_call (续)

```
ENTRY(ret_from_sys_call)
        cli    # need_resched and signals atomic test,关中断
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
    restore_all:
        RESTORE_ALL
```



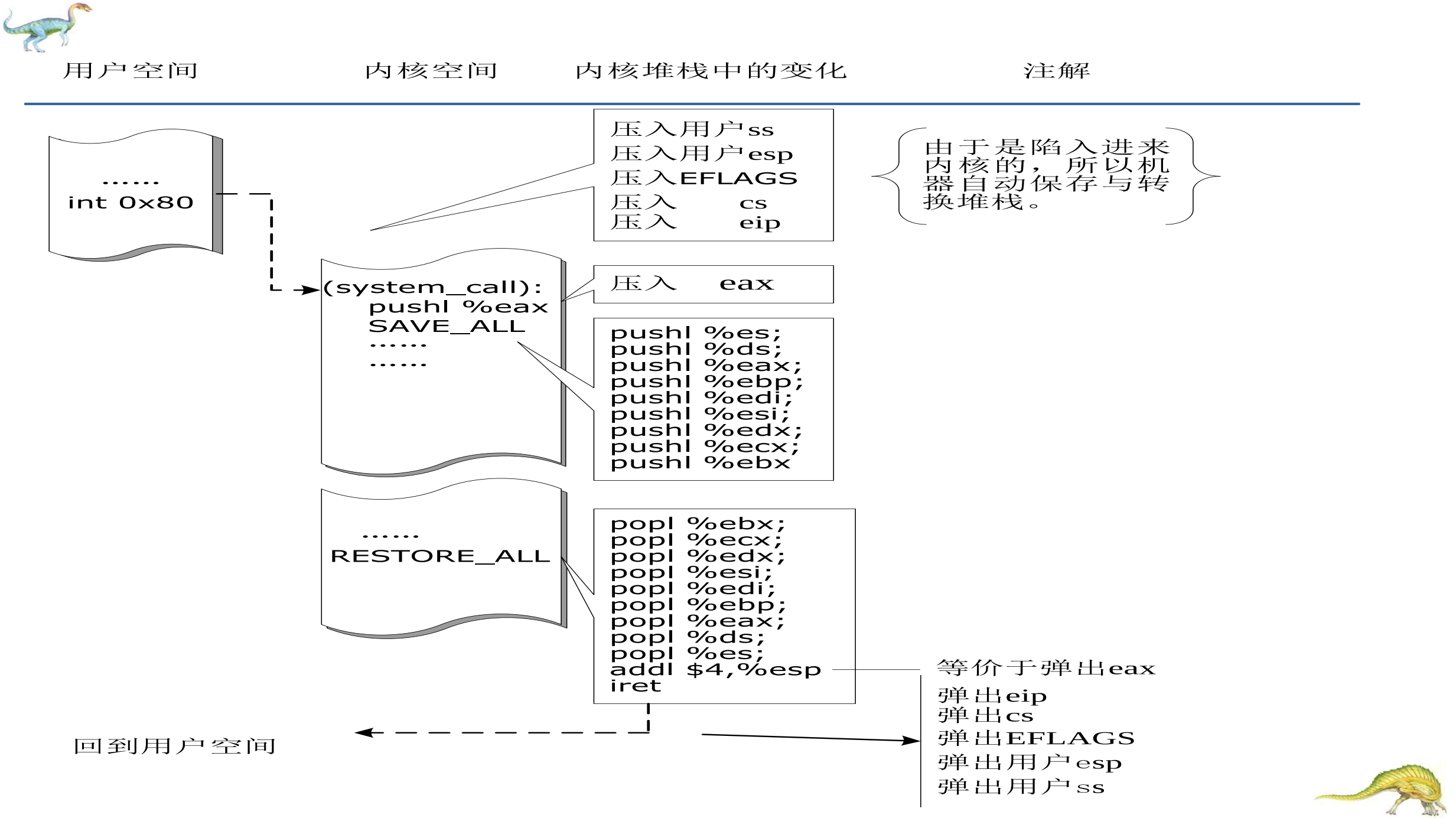
SAVE_ALL定义

```
#define SAVE_ALL
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS),%edx; \
        movl %edx,%ds; \
        movl %edx,%es; \
```

使用内核数据段

RESTORE_ALL定义

```
#define RESTORE_ALL     \
        popl %ebx;      \
        popl %ecx;      \
        popl %edx;      \
        popl %esi;      \
        popl %edi;      \
        popl %ebp;      \
        popl %eax;      \
    1:  popl %ds;       \
    2:  popl %es;       \
        addl $4,%esp;       \
    3:  iret;           \
```


用户空间
内核空间
内核堆栈中的变化
注解
.....
int 0x80
压入用户ss
压入用户esp
压入EFLAGS
压入 cs
压入 eip
由于是陷入进来内核的，所以机器自动保存与转换堆栈。
(system_call):
pushl %eax
SAVE_ALL
.....
.....
压入 eax
pushl %es;
pushl %ds;
pushl %eax;
pushl %ebp;
pushl %edi;
pushl %esi;
pushl %edx;
pushl %ecx;
pushl %ebx
.....
RESTORE_ALL
popl %ebx;
popl %ecx;
popl %edx;
popl %esi;
popl %edi;
popl %ebp;
popl %eax;
popl %ds;
popl %es;
addl $4,%esp
iret
等价于弹出eax
弹出eip
弹出cs
弹出EFLAGS
弹出用户esp
弹出用户ss
回到用户空间

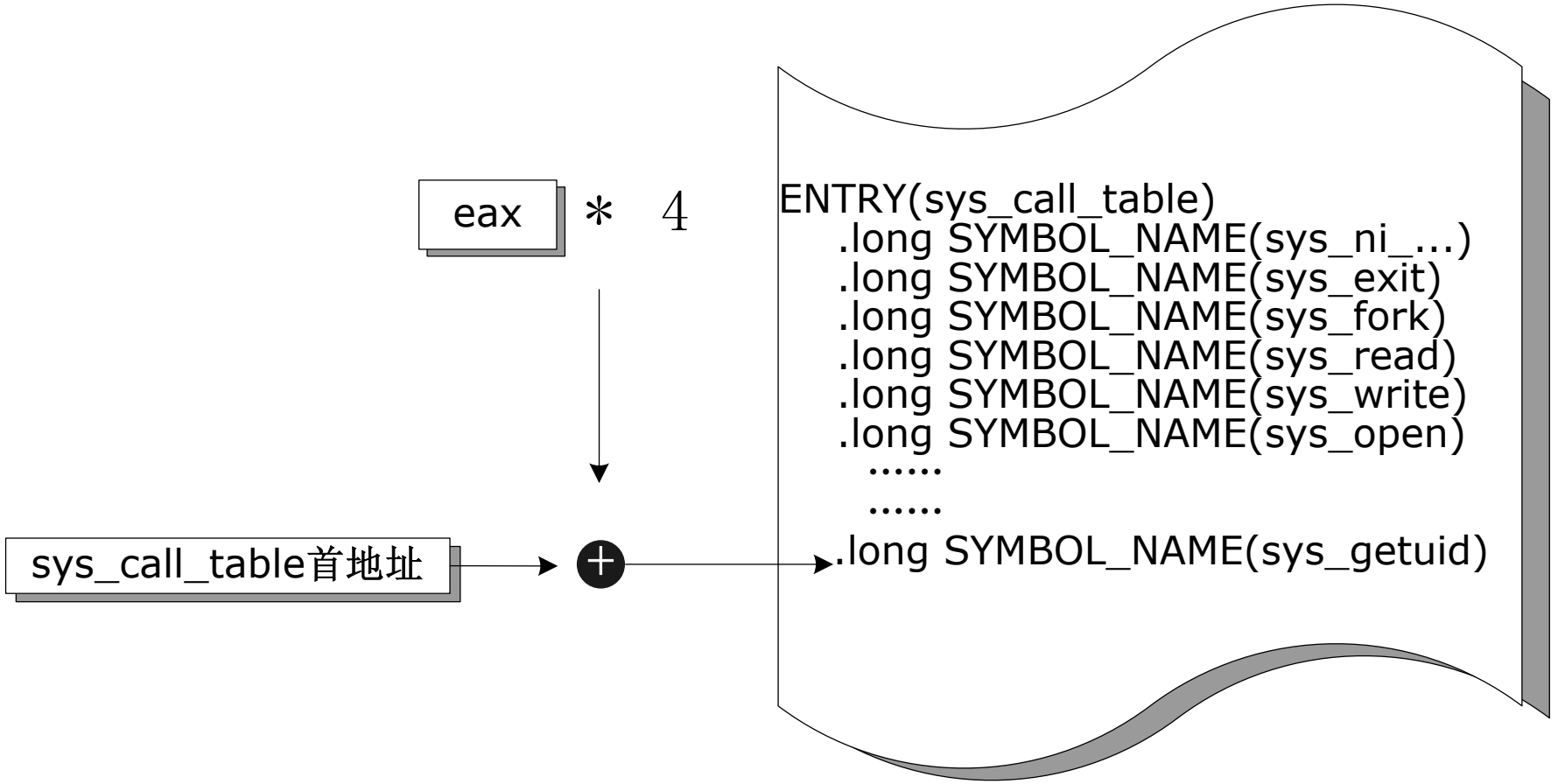
sys_call_table

arch/x86/kernel/syscall_table.S或 arch/x86/entry/syscalls/syscall_32.tbl

```
ENTRY(sys_call_table)
    .long sys_restart_syscall      /* 0 - old "setup()" system call, used for restarting */
    .long sys_exit                  /* 1 */
    .long sys_fork
    .long sys_read
    .long sys_write
    .long sys_open                 /* 5 */
    .long sys_close
          …
    .long sys_ioprio_get           /* 290 */
    .long sys_inotify_init
    .long sys_inotify_add_watch
    .long sys_inotify_rm_watch
```

每个系统调用服务例程的入口地址，每个地址4个字节

sys_call_table (续)



系统调用编号

include/asm-i386/unistd.h 或include/uapi/asm-generic/unistd.h)

```
#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
   ……
#define __NR_ioprio_get                290
#define __NR_inotify_init              291
#define __NR_inotify_add_watch         292
#define __NR_inotify_rm_watch          293

#define NR_syscalls 294
```

定义每个系统调用的编号

系统调用中参数传递

- 每个系统调用至少有一个参数，即通过eax寄存器传递来的系统调用号
- 用寄存器传递参数必须满足两个条件：
  - 每个参数的长度不能超过寄存器的长度
  - 参数的个数不能超过6个（包括eax中传递的系统调用号），否则，用一个单独的寄存器指向进程地址空间中这些参数值所在的一个内存区即可
- 在少数情况下，系统调用不使用任何参数
- 服务例程的返回值必须写到eax寄存器中

宏定义展开系统调用

- 宏定义syscallN()用于系统调用的格式转换和参数的传递。

Include/asm-i386/unistd.h

```
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
    long __res; \
     __asm__ volatile ("int $0x80" \

      : "=a" (__res) \
      : "" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),  "d" ((long)(arg3))); \

    __syscall_return(type,__res); \
}
```

第一个":"是output
第二个":"是input

系统调用中普通参数的传递

- 3个参数系统调用生成汇编代码

```
movl        $__NR_##name, %eax    //系统调用号给eax寄存器
movl        arg1, %ebx
movl        arg2, %ecx
movl        arg3, %edx
#APP
int   $0x80
#NO_APP
movl        %eax, __res       //最后处理输出参数
```

- syscallN()第一个参数说明响应函数返回值的类型，第二个参数为系统调用的名称（即name），其余的参数依次为系统调用参数的类型和名称。例如：

  _syscall3(int, open, const char * pathname, int flag, int mode)

  说明了系统调用命令

  int open(const char *pathname, int flags, mode_t mode)

系统调用参数与寄存器

参数	参数在堆栈的位置	传递参数的寄存器
arg1	00(%esp)	ebx
arg2	04(%esp)	ecx
arg3	08(%esp)	edx
arg4	0c(%esp)	esi
arg5	10(%esp)	edi

系统调用小结

- 应用程序执行系统调用大致可归结为以下几个步骤：
  - 1、程序调用libc库的封装函数。
  - 2、调用软中断 int 0x80 进入内核。
  - 3、在内核中首先执行system_call函数，接着根据系统调用号在系统调用表中查找到对应的系统调用服务例程 。
  - 4、执行该服务例程。
  - 5、执行完毕后，转入ret_from_sys_call例程，从系统调用返回

例：系统调用getuid()的实现

- 一个简单的程序，但包含系统调用和库函数调用

```
#include        <linux/unistd.h>
        /* all system calls need this header */
int main(){
    int   i = getuid();
    printf("Hello World! This is my uid: %d\n", i);
}
```

- 假定<unistd.h>定义了“宏”

```
    _syscall0( int, getuid);
```

例：系统调用getuid()的实现(续)

- 这个“宏”被getuid()展开后

```
int getuid(void)
{
        long __res;
        __asm__ volatile ("int $0x80"
                        : "=a" (__res)
                        : "" (__NR_getuid));
        __syscall_return(int,__res);
}
```

- 显然，__NR_getuid（24）放入eax，并int 0x80

例：系统调用getuid()的实现(续)

- 因为系统初始化时设定了

  set_system_gate(SYSCALL_VECTOR,&system_call);

- 所以进入system_call

```
194 ENTRY(system_call)
195         pushl %eax                  # save orig_eax
196         SAVE_ALL
197         GET_CURRENT(%ebx)
198         testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
199         jne tracesys
200         cmpl $(NR_syscalls),%eax
201         jae badsys
202         call *SYMBOL_NAME(sys_call_table)(,%eax,4)
203         movl %eax,EAX(%esp)
```

例：系统调用getuid()的实现(续)

- 注意第202行，此时eax为24
- 查sys_call_table，得到call指令的操作数sys_getuid16
- 调用函数sys_getuid16()

```
145 asmlinkage long sys_getuid16(void)
146 {
147         return high2lowuid(current->uid);
148 }
```

例：系统调用getuid()的实现(续)

- 第202行完成后，接着执行第203行后面的指令

```
203         movl %eax,EAX(%esp)
204 ENTRY(ret_from_sys_call)
205         cli
206         cmpl $0,need_resched(%ebx)
207         jne reschedule
208         cmpl $0,sigpending(%ebx)
209         jne signal_return
210 restore_all:
211         RESTORE_ALL
```

例：系统调用getuid()的实现(续)

- 第203行：返回值从eax移到堆栈中eax的位置
- 假设没有什么意外发生，于是ret_from_sys_call直接到RESTORE_ALL，从堆栈里面弹出保存的寄存器，堆栈切换，iret
- 进程回到用户态，返回值保存在eax中
- printf打印出

  Hello World! This is my uid：551

实验：添加一个系统调用mysyscall

- 实验内容：

  在现有的系统中添加一个不用传递参数的系统调用。实现统计操作系统缺页总次数和当前进程的缺页次数及每个进程“脏”页面数。

添加一个系统调用mysyscall (续)

- 缺页统计：
  - 每发生一次缺页都要进入缺页中断服务函数do_page_fault一次，所以可以认为执行该函数的次数就是系统发生缺页的次数。
  - 可以定义一个全局变量pfcount作为计数变量，在执行do_page_fault时，该变量值加1。
  - 在当前进程控制块task_struct中定义一个变量pf记录当前进程缺页次数，在执行do_page_fault时，这个变量值加1。

添加一个系统调用mysyscall (续)

- 实验内容：
  - 添加系统调用的名字
  - 利用标准C库进行包装
  - 添加系统调用号
  - 在系统调用表中添加相应表项
  - 修改统计缺页次数相关的内核结构和函数
  - sys_mysyscall的实现
  - 编写用户态测试程序

实验

- 实验2 添加系统调用