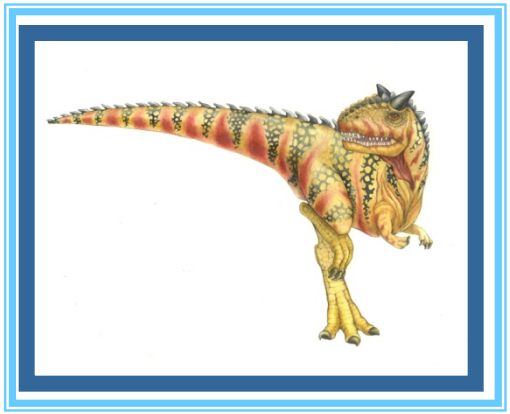
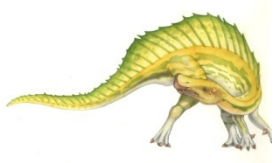
系统调用





本章内容

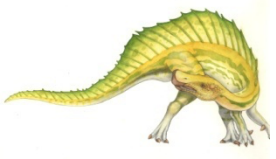
- 系统调用基础知识
- 数据结构和代码
- 系统调用getuid()的实现
- 添加一个系统调用mysyscall

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);
}
```

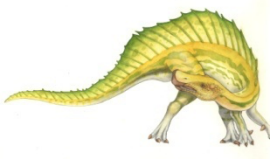
普通函数？

- 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



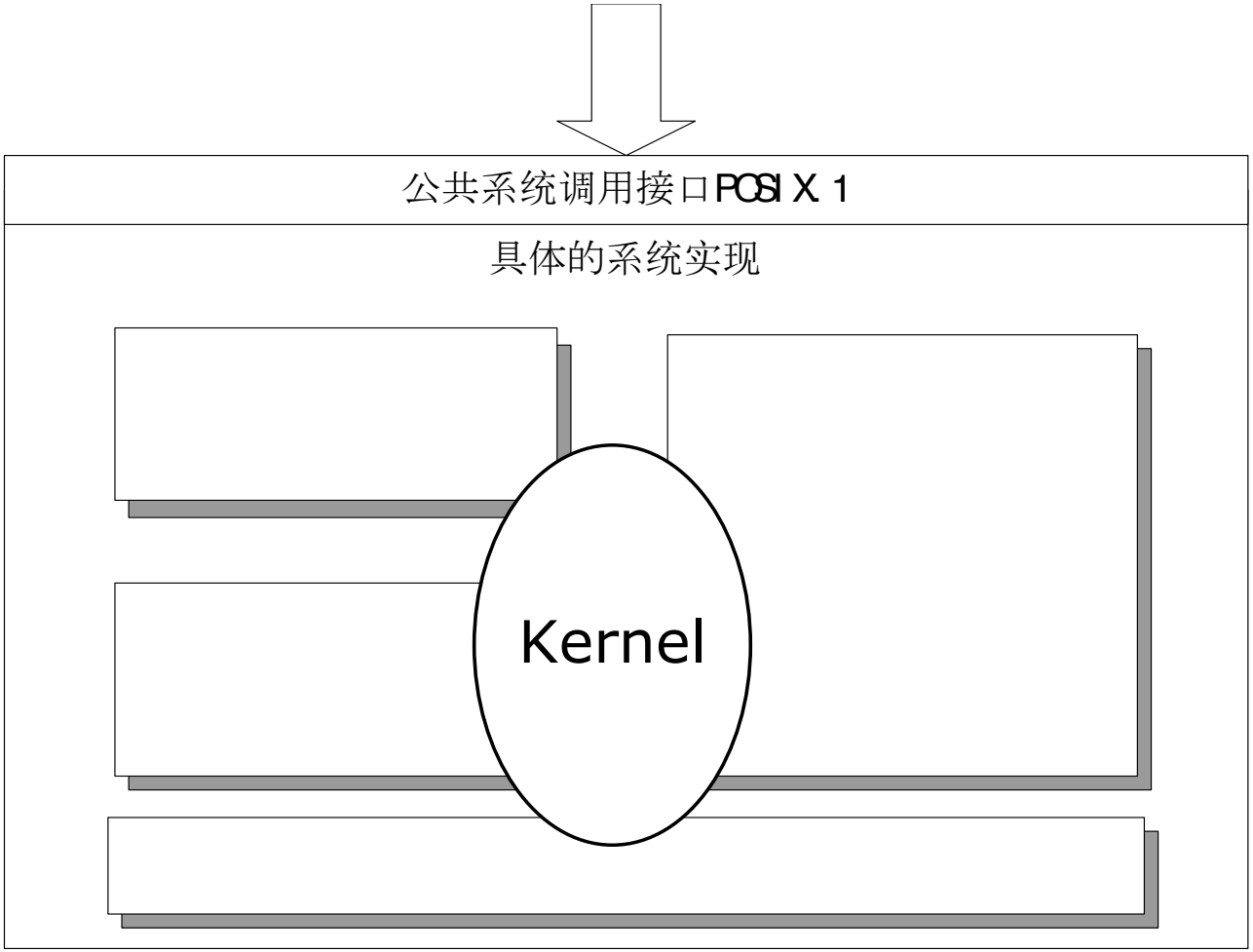
为什么需要系统调用

- 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。

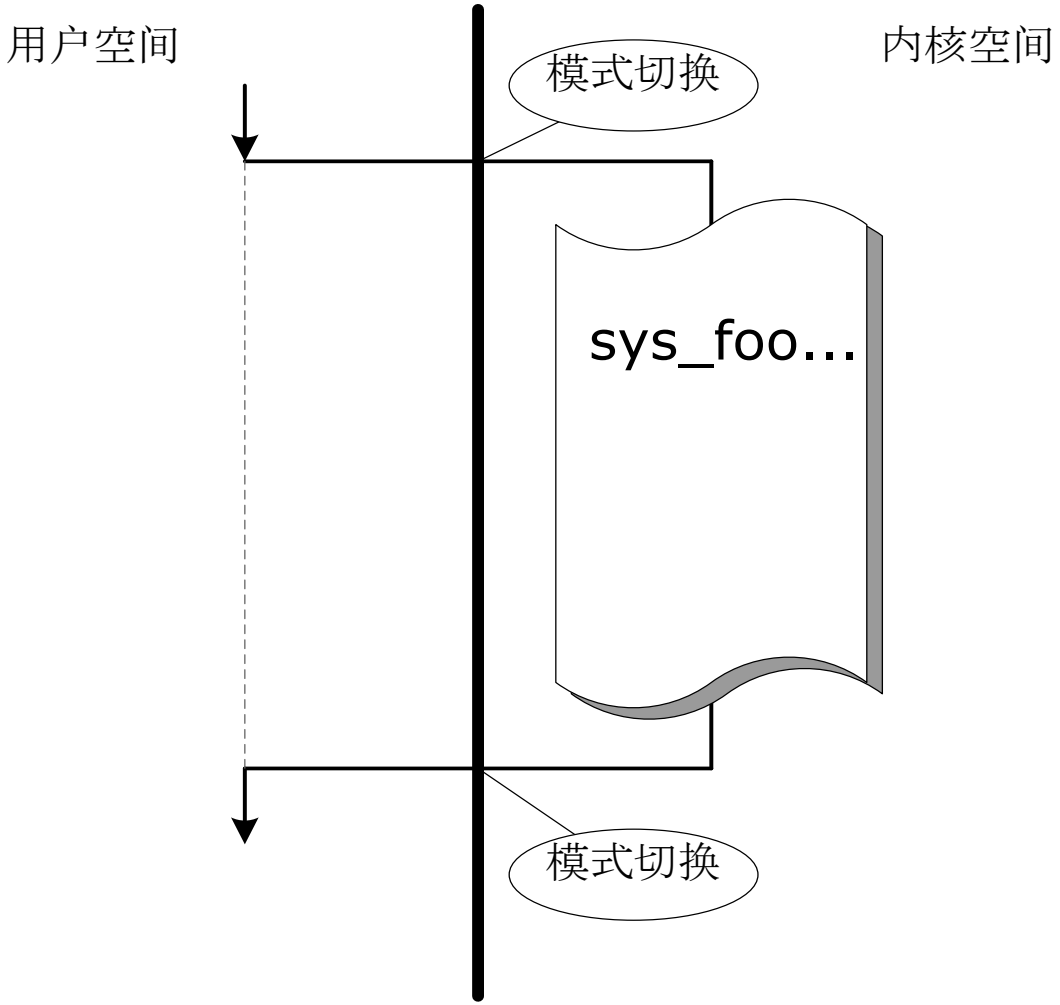


为什么需要系统调用(续)

用户程序调用内核提供的功能

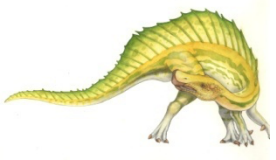


为什么需要系统调用(续)



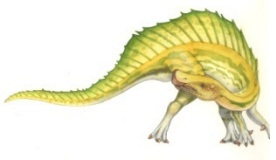
运行模式、地址空间、上下文（for x86）

- 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；而在核心态下，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。**

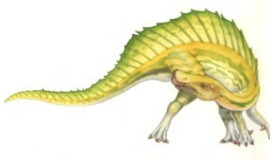


系统调用、API和C库

- 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



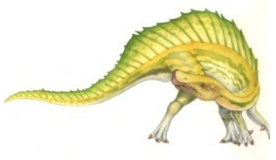
系统调用与操作系统命令

strace ls

```
open(".",
O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY|O_CL
OEXEC) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
fcntl64(3, F_GETFD)                    = 0x1 (flags FD_CLOEXEC)
getdents64(3, /* 18 entries */, 4096)   = 496
getdents64(3, /* 0 entries */, 4096)    = 0
close(3)                               = 0
fstat64(1, {st_mode=S_IFIFO|0600, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f2c000
write(1, "autofs\nbackups\ncache\nflexlm\ngames"..., 86autofsA
```

系统调用与内核函数

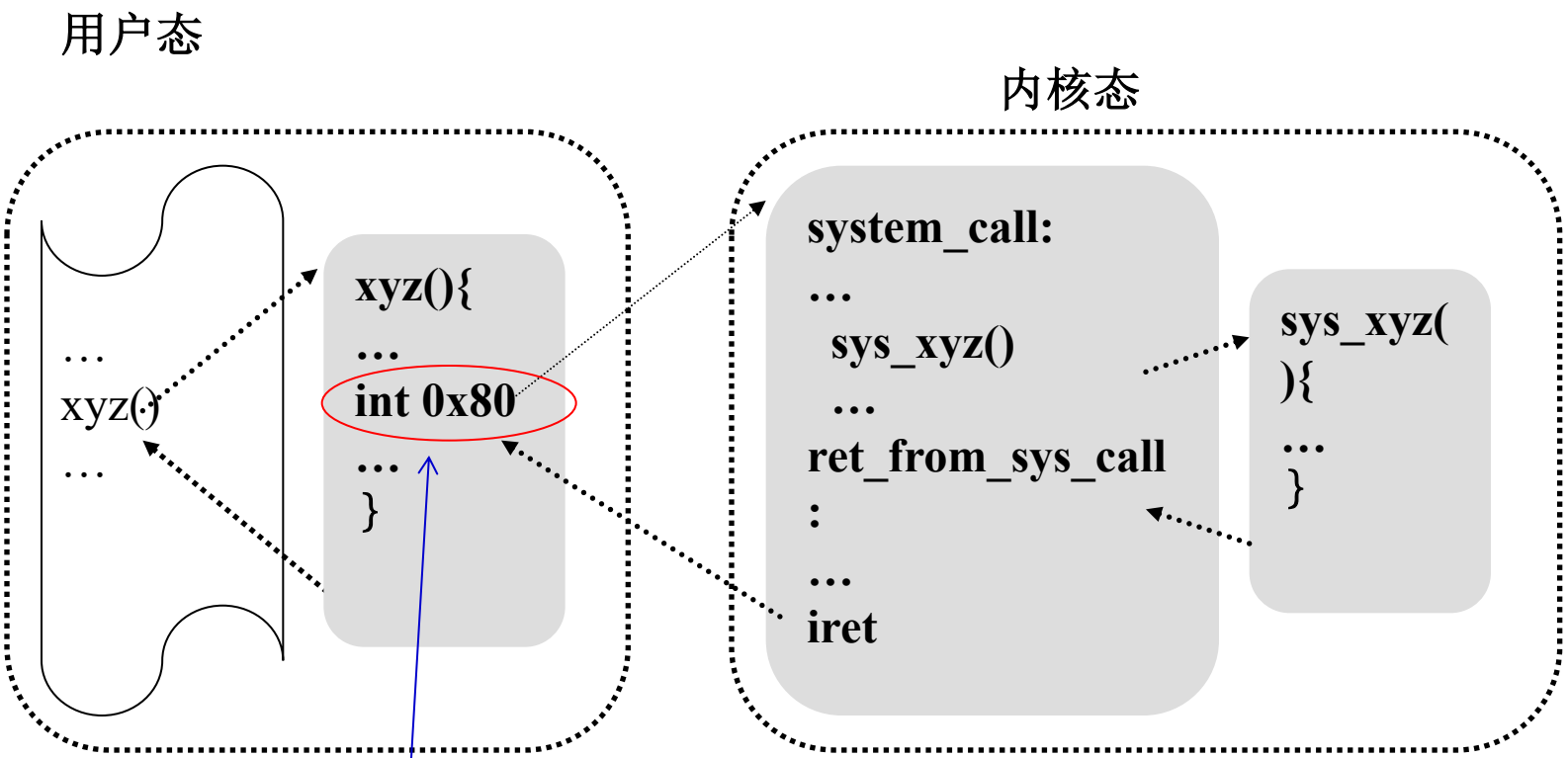
- **内核函数**在形式上与普通函数一样，但它是在内核实现的，需要满足一些内核编程的要求
- **系统调用是用户进程进入内核的接口层**，它本身并非内核函数，但它是**由内核函数实现**的
- 进入内核后，不同的系统调用会找到各自对应的内核函数，这些内核函数被称为系统调用的“**服务例程**”



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）



在应用程序调用系统调用（API）

在libc标准库中的封装例程

系统调用中断处理程序

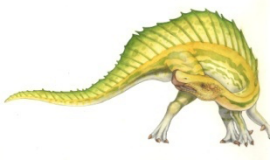
系统调用服务例程

新的系统调用sysenter指令，从用户态到内核态快速切换方法

调用一个系统调用（x86）

- 用户调用fork -> eax=2（保存系统调用号到寄存器中）
- int 0x80（触发中断，切换到内核态）
- 在中断向量表中查找（0x80号），执行0x80对应的中断服务程序（system_call）
- 在系统调用表中找到系统调用号为2的那一项（通过之前保存的eax=2）-> 执行系统调用（sys_fork）

```
void main(void)
{
    fork();
}
```



IDT：系统中断表

向量范围	用途
0～19	不可屏蔽中断和异常
20～31	Intel保留
32～127	外部中断（IRQ）
128（0x80）	用于系统调用的可编程异常
129～238	外部中断
239	本地APIC时钟中断
240	本地APIC高温中断
241～250	Linux保留
251～253	处理器间中断
254	本地APIC错误中断
255	本地APIC伪中断

- 不可屏蔽中断/异常
  - 0 -- 除零
  - 1 -- 单步调试
  - 4 -- 算术溢出
  - 6 --非法操作数
  - 12 -- 栈异常
  - 13 -- 保护性错误
  - 14 -- 缺页异常

sched_init

```
00506: void sched_init (void)
00507: {
00508:     int i;
00509:     struct desc_struct *p;                    // 描 述 符 表 结 构 指 针 。
00510:
00511:     if (sizeof (struct sigaction) != 16)      // sigaction 是 存 放 有 关 信 号 状 态 的 结 构 。
00512:         panic ("Struct sigaction MUST be 16 bytes");
00513:
00514:     // 设 置 初 始 任 务 （ 任 务 0） 的 任 务 状 态 段 描 述 符 和 局 部 数 据 表 描 述 符 (inc
00515:     set_tss_desc (gdt + FIRST_TSS_ENTRY, &(init_task.task.tss));
00516:     set_ldt_desc (gdt + FIRST_LDT_ENTRY, &(init_task.task.ldt));
00517:     // 清 任 务 数 组 和 描 述 符 表 项 （ 注 意 i=1 开 始 ， 所 以 初 始 任 务 的 描 述 符 还 在
00518:     p = gdt + 2 + FIRST_TSS_ENTRY;
00519:     for (i = 1; i < NR_TASKS; i++)
00520:     {
00521:         task[i] = NULL;
00522:         p->a = p->b = 0;
00523:         p++;
00524:         p->a = p->b = 0;
00525:         p++;
00526:     }
00527: /* 清 除 标 志 寄 存 器 中 的 位 NT， 这 样 以 后 就 不 会 有 麻 烦 */
00528: // NT 标 志 用 于 控 制 程 序 的 递 归 调 用 (Nested Task)。 当 NT 置 位 时 ， 那 么 当 前 中 断
00529: // iret 指 令 时 就 会 引 起 任 务 切 换 。 NT 指 出 TSS 中 的 back_link 字 段 是 否 有 效 。
00530: //  __asm__ ("pushfl ; andl $0xffffbfff,(%esp) ; popfl"); // 复 位 NT 标 志 。
00531:     _asm pushfd; _asm and dword ptr ss:[esp],0xffffbfff; _asm popfd;
00532:     ltr (0);              // 将 任 务 0 的 TSS 加 载 到 任 务 寄 存 器 tr。
00533:     lldt (0);             // 将 局 部 描 述 符 表 加 载 到 局 部 描 述 符 表 寄 存 器 。
00534: // 注 意 ！ ！ 是 将 GDT 中 相 应 LDT 描 述 符 的 选 择 符 加 载 到 ldtr。 只 明 确 加 载 这 一
00535: // LDT 的 加 载 ， 是 CPU 根 据 TSS 中 的 LDT 项 自 动 加 载 。
00536: // 下 面 代 码 用 于 初 始 化 8253 定 时 器 。
00537:     outb_p (0x36, 0x43);          /* binary, mode 3, LSB/ MSB, ch 0 */
00538:     outb_p (LATCH & 0xff, 0x40);    /* LSB */// 定 时 值 低 字 节 。
00539:     outb (LATCH >> 8, 0x40);      /* MSB */// 定 时 值 高 字 节 。
00540:   // 设 置 时 钟 中 断 处 理 程 序 句 柄 （ 设 置 时 钟 中 断 门 ） 。
00541:     set_intr_gate (0x20, &timer_interrupt);
00542:   // 修 改 中 断 控 制 器 屏 蔽 码 ， 允 许 时 钟 中 断 。
00543:     outb (inb_p (0x21) & ~0x01, 0x21);
00544:   // 设 置 系 统 调 用 中 断 门 。
00545:     set_system_gate (0x80, &system_call);
00546: } ? end sched_init ?
```

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
  - offset：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

老版内核版本的子目录名字为“i386”

- 与系统调用相关的内核代码文件：
  - arch/x86/kernel/entry.S

    系统调用时的内核栈

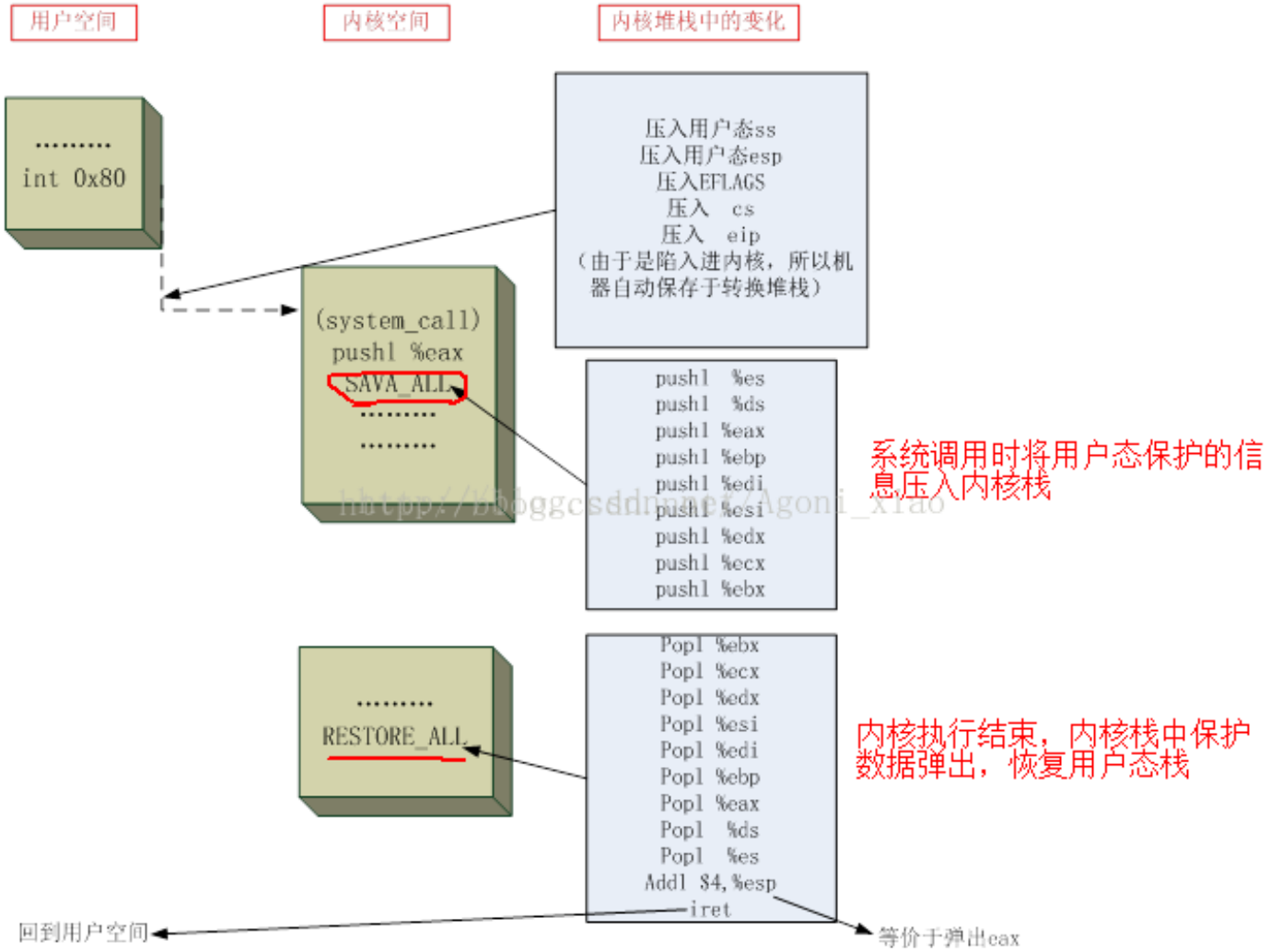
    sys_call_table

    system_call和ret_from_sys_call
  - arch/x86/kernel/traps.c //错误处理程序文件
  - include/linux/unistd.h

    系统调用编号

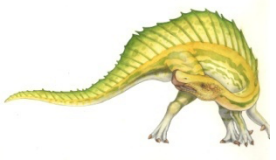
    宏定义展开系统调用

系统调用全过程


用户空间
内核空间
内核堆栈中的变化
.....
int 0x80
压入用户态ss
压入用户态esp
压入EFLAGS
压入 cs
压入 eip
（由于是陷入进内核，所以机器自动保存于转换堆栈）
(system_call)
pushl %eax
SAVA_ALL
.....
.....
pushl %es
pushl %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
系统调用时将用户态保护的信息压入内核栈
.....
RESTORE_ALL
Popl %ebx
Popl %ecx
Popl %edx
Popl %esi
Popl %edi
Popl %ebp
Popl %eax
Popl %ds
Popl %es
Addl $4,%esp
iret
内核执行结束，内核栈中保护数据弹出，恢复用户态栈
回到用户空间
等价于弹出eax

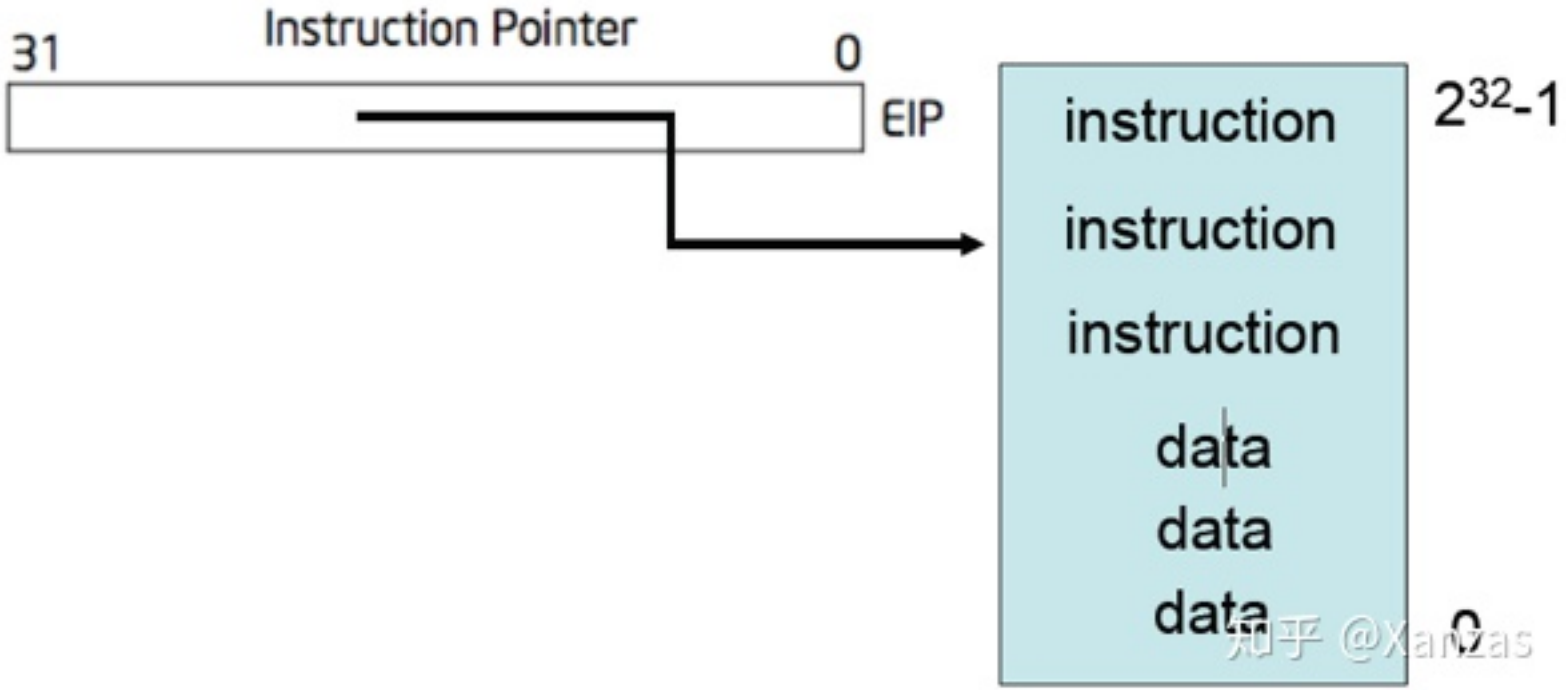
系统调用时的内核栈

- 调用0x80中断时，程序执行流程从用户态切换到内核态，当前栈也必须相应的从用户栈切换到内核栈。从中断处理程序中返回时，再切换回用户栈
- “当前栈”指的是ESP的值所在的栈空间，若ESP的值位于用户栈的范围内(0G-3G)，那个当前栈就是用户栈，反之就是内核栈(3G-4G)。此外，寄存器SS的值还要指向当前栈所在的页



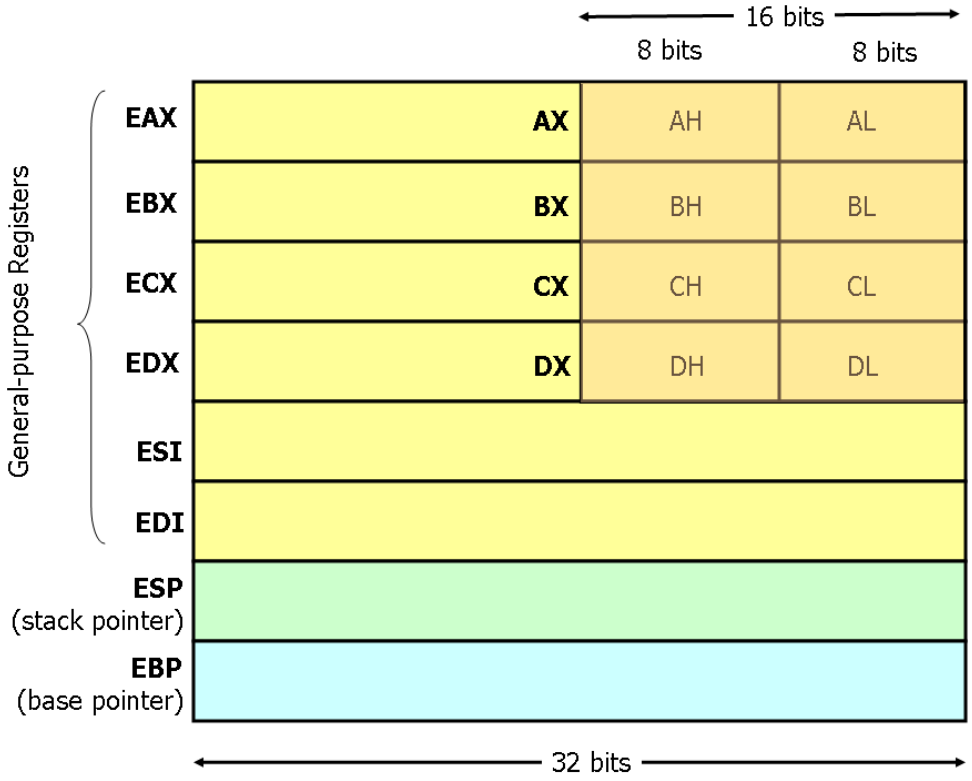
Linux常用寄存器

- 每条指令读取后eip自增
- 指令的长度可能不同
- 通过call，ret，jmp等指令可以修改eip的值



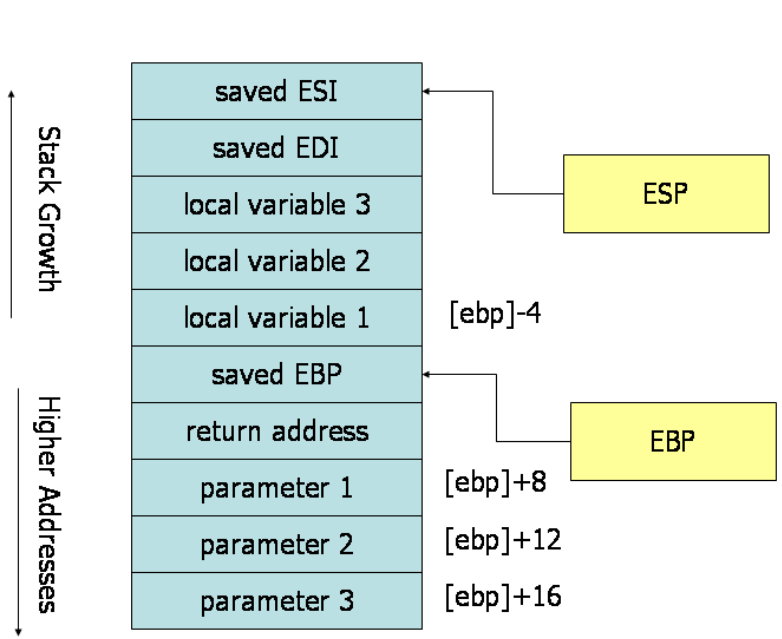
Linux常用寄存器

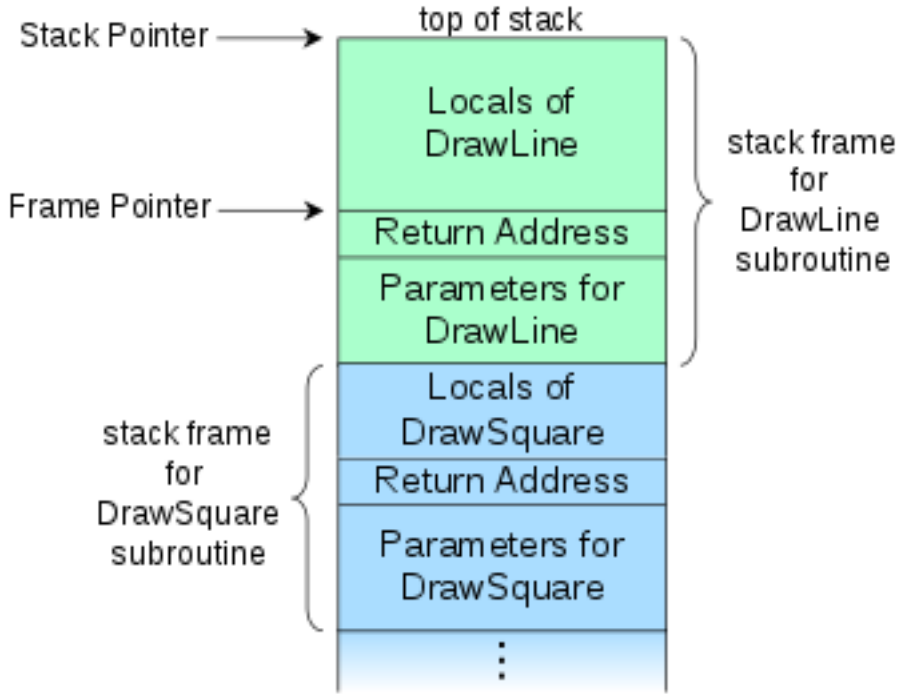
- esp：堆栈顶指针，寄存器中存放栈顶地址。
- ebp：堆栈基指针，寄存器中存放栈底地址。
- eip：指令指针，寄存器中存放着下一条指令的地址。
- eax：累加器。
- CS：代码段寄存器。
- DS：数据段寄存器。
- SS：堆栈段寄存器。



Linux常用寄存器

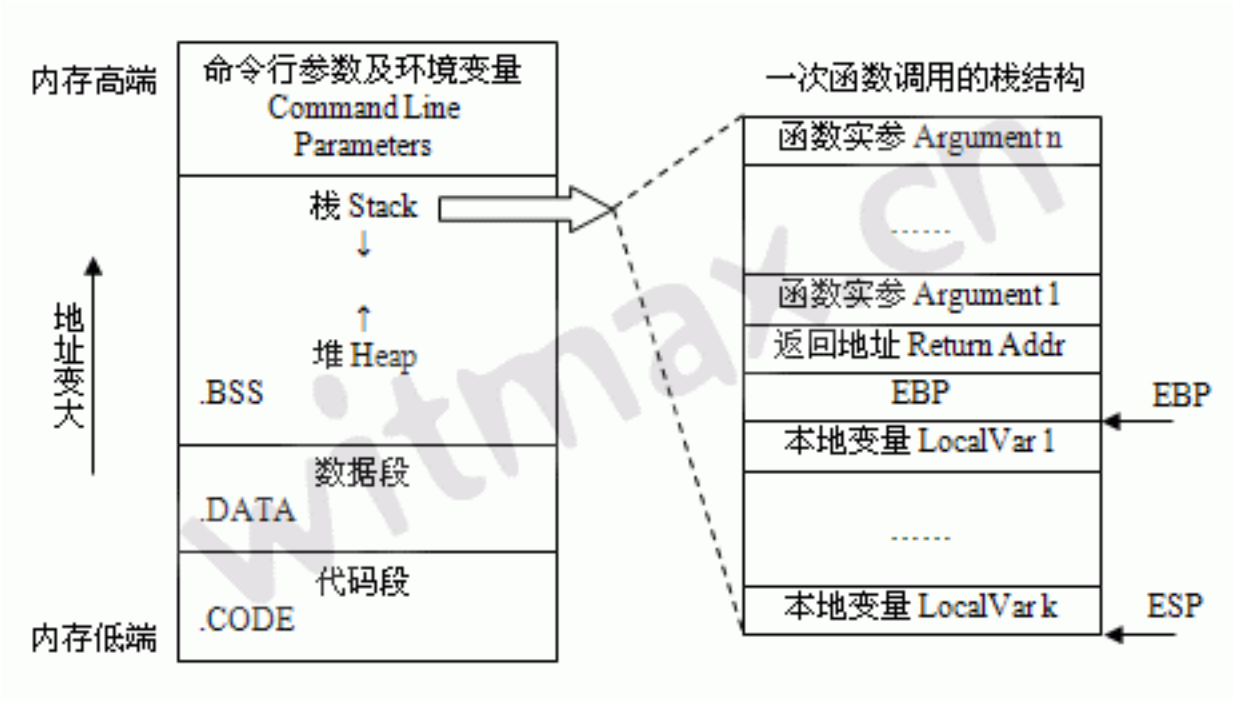
- **函数调用时的堆栈管理**

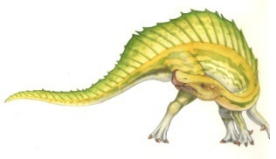




常用堆栈

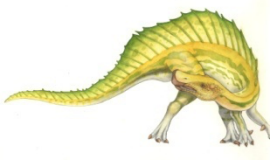
- 在实模式下，SS*4是堆栈段的基地址，程序初始化时，堆栈为空栈（SP被初始化为堆栈的大小，即堆栈的深度），此时栈底等于栈顶。SS：SP始终指向栈底（但只有堆栈为空时，SS:SP既是栈顶又是栈底）。随着压栈操作，SP逐渐减小，SP越来越趋于0。当SP等于0时，SS:SP实际上就是SS：00。一个进程栈的默认大小是1M



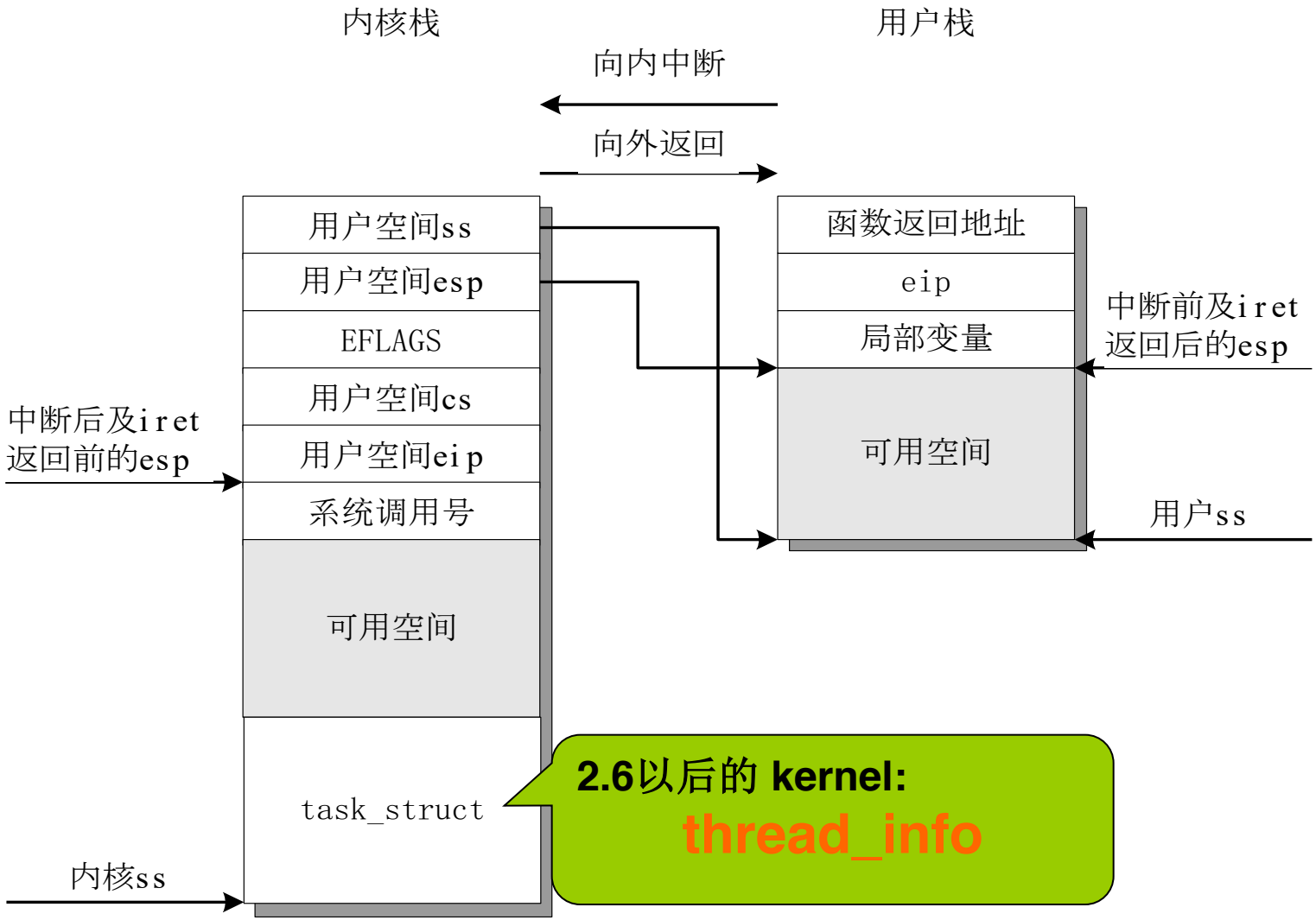


系统调用时的内核栈

- 用户栈 -> 内核栈的实际行为就是：
  - 保存当前的ESP，SS的值 -> 将ESP SS的值设置为内核栈的相应值
- 内核栈 -> 用户栈的实际行为就是：
  - 恢复原来的ESP SS的值
- 中断发生时，CPU切入内核态，还会接着做下面几件事
  - 找到当前进程的内核栈（每个进程都有独立的内核栈） -> 在内核栈中一次压入用户态的寄存器SS、ESP、EFLAGS、CS、EIP
- 系统从系统调用中返回时，需要用iret指令回到用户态，iret会从内核态中弹出寄存器SS、ESP、EFLAGS、CS、EIP的值，使得栈恢复到用户态的状态



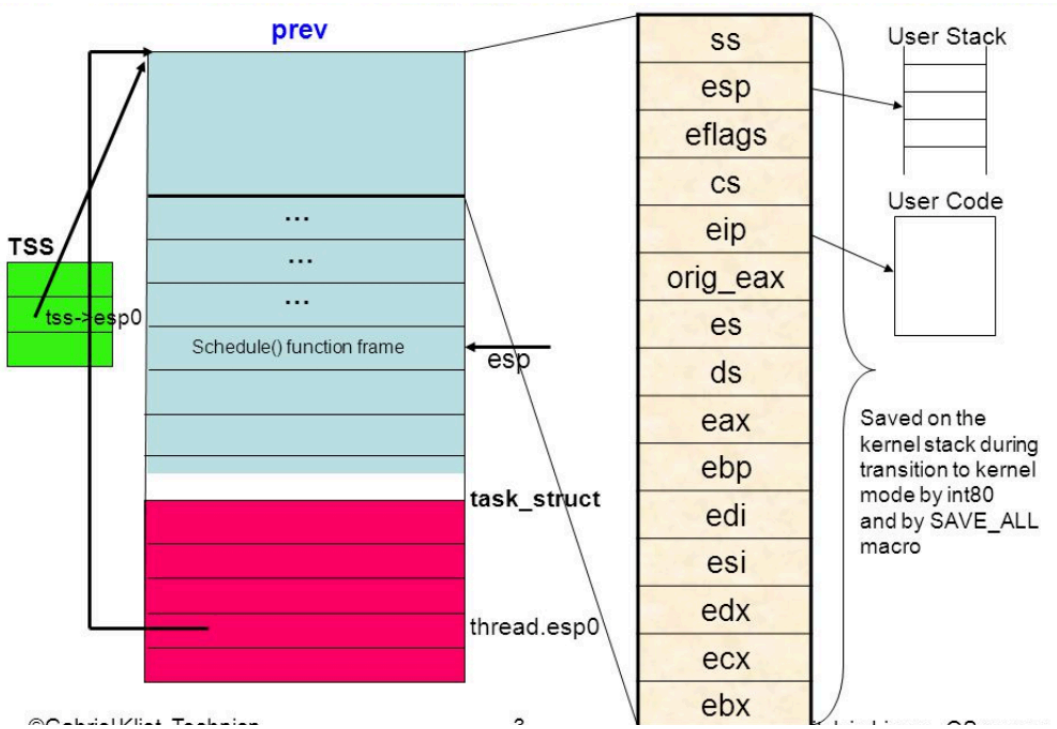
系统调用时的内核栈


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

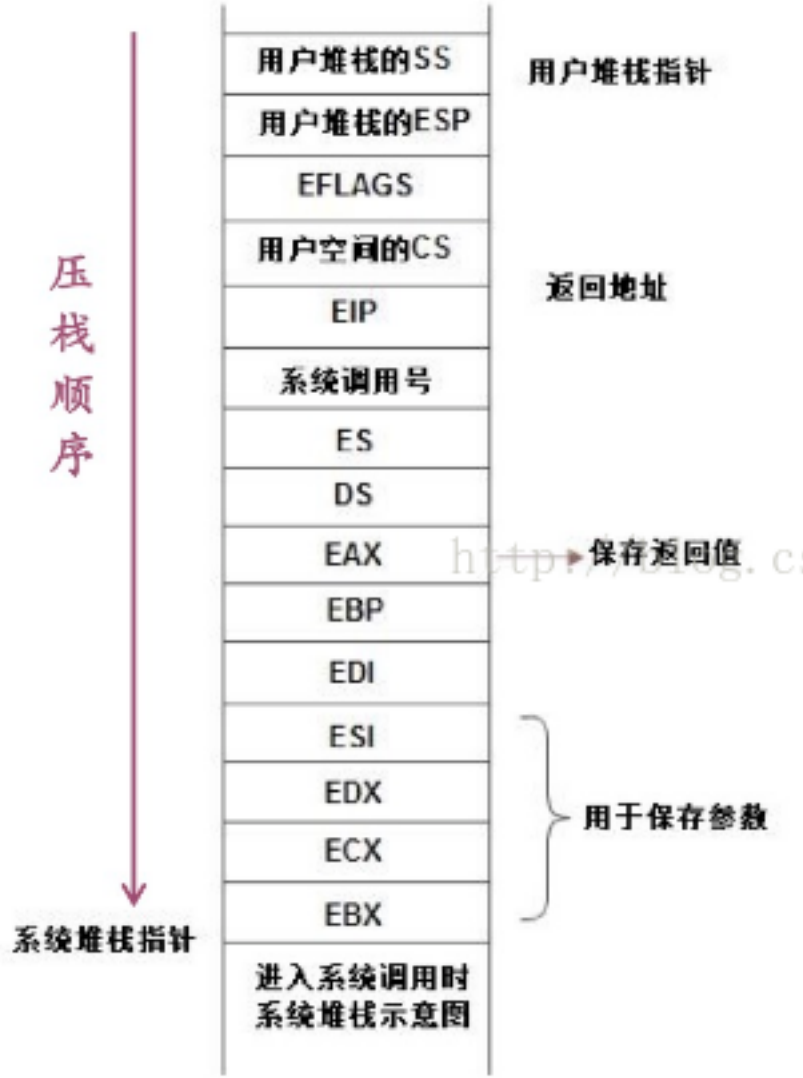
系统调用时的内核栈(续)

arch/x86/kernel/entry.S

```
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.
```



压栈顺序




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

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)

```
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_struct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

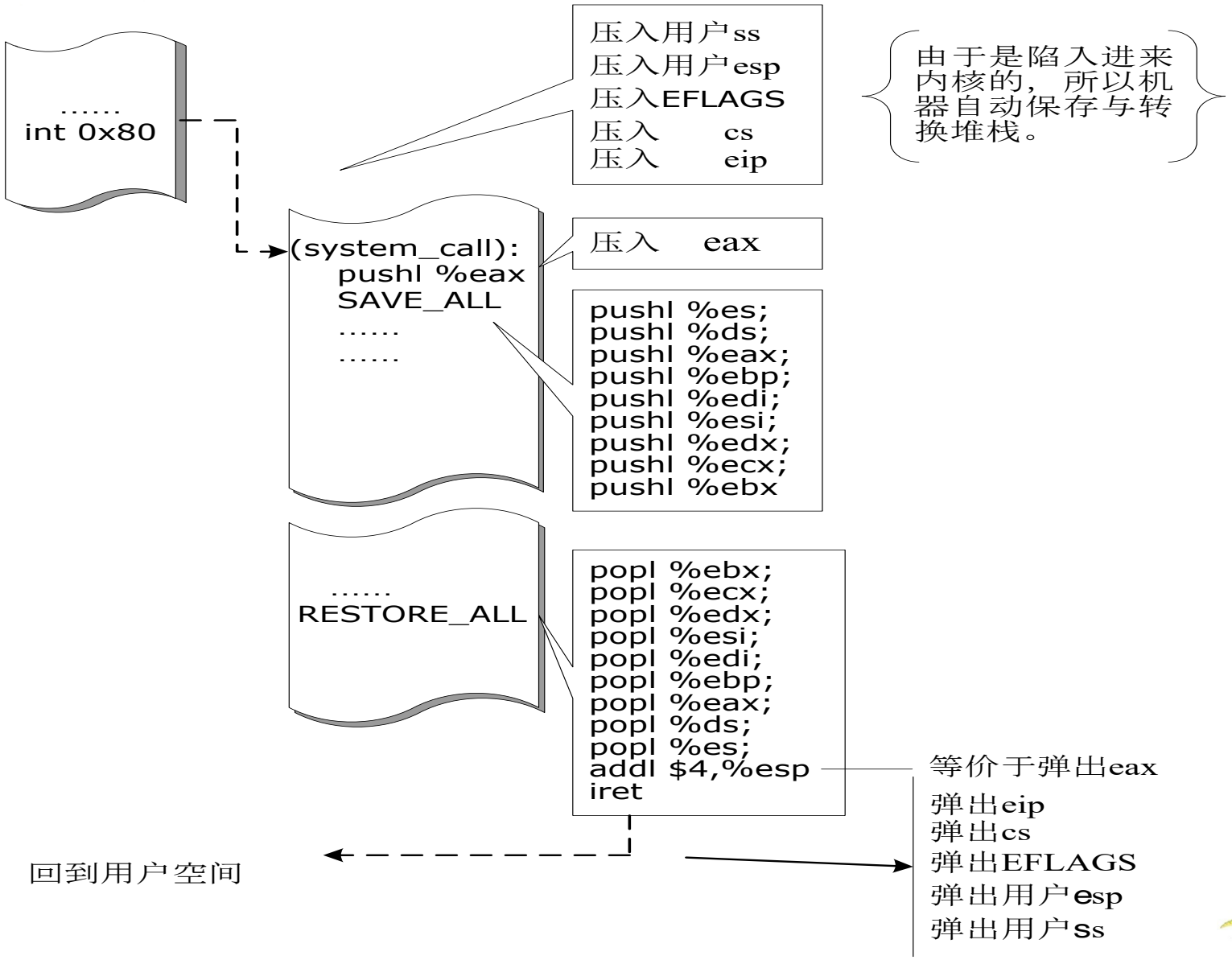
```
#define SAVE_ALL
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS),%edx; \
        movl %edx,%ds; \
       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



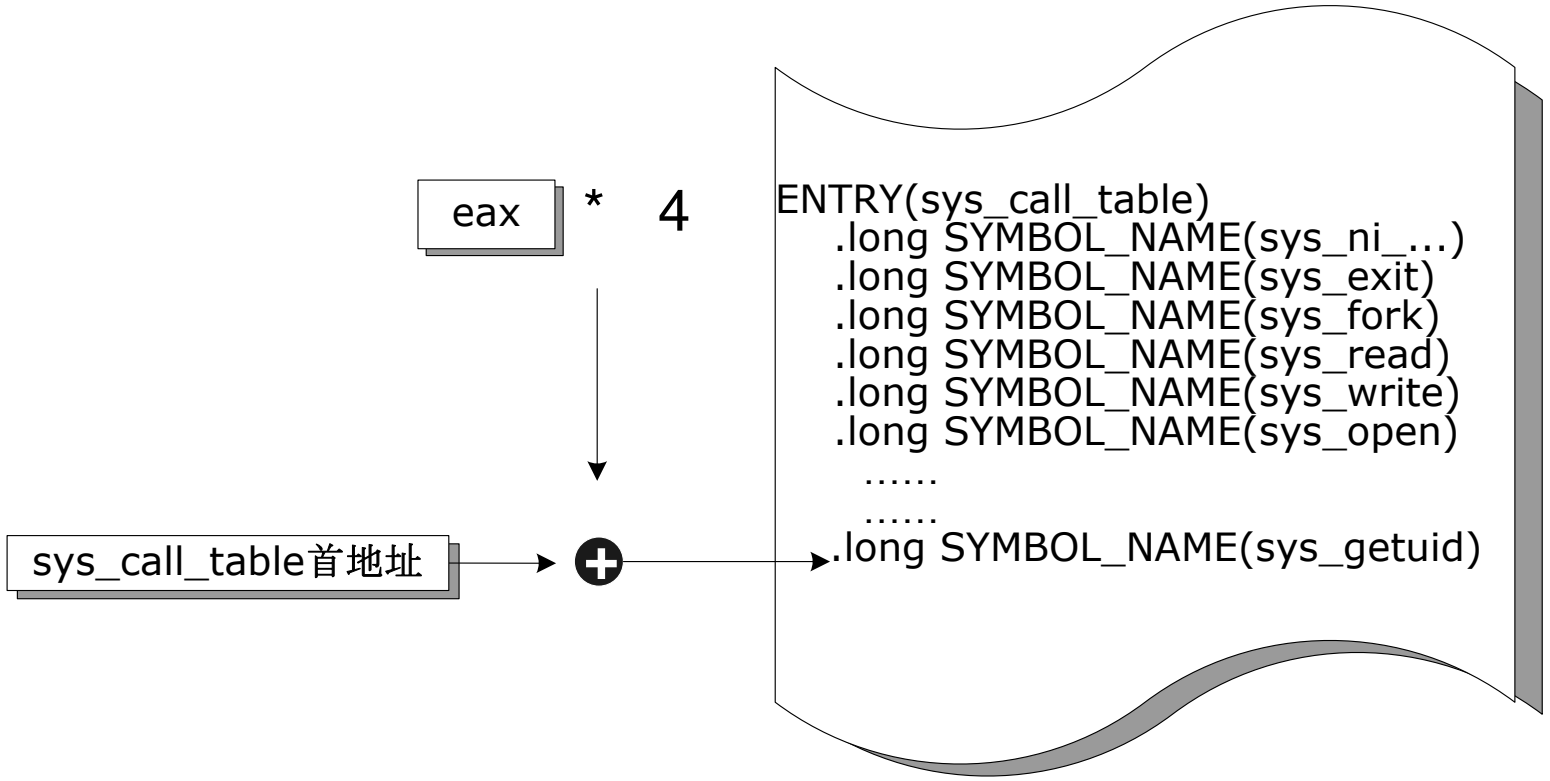
sys_call_table

arch/x86/kernel/syscall_table.S

```
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)



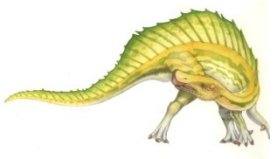
系统调用编号

arch/x86/include/generated/uapi/asm/unistd.h

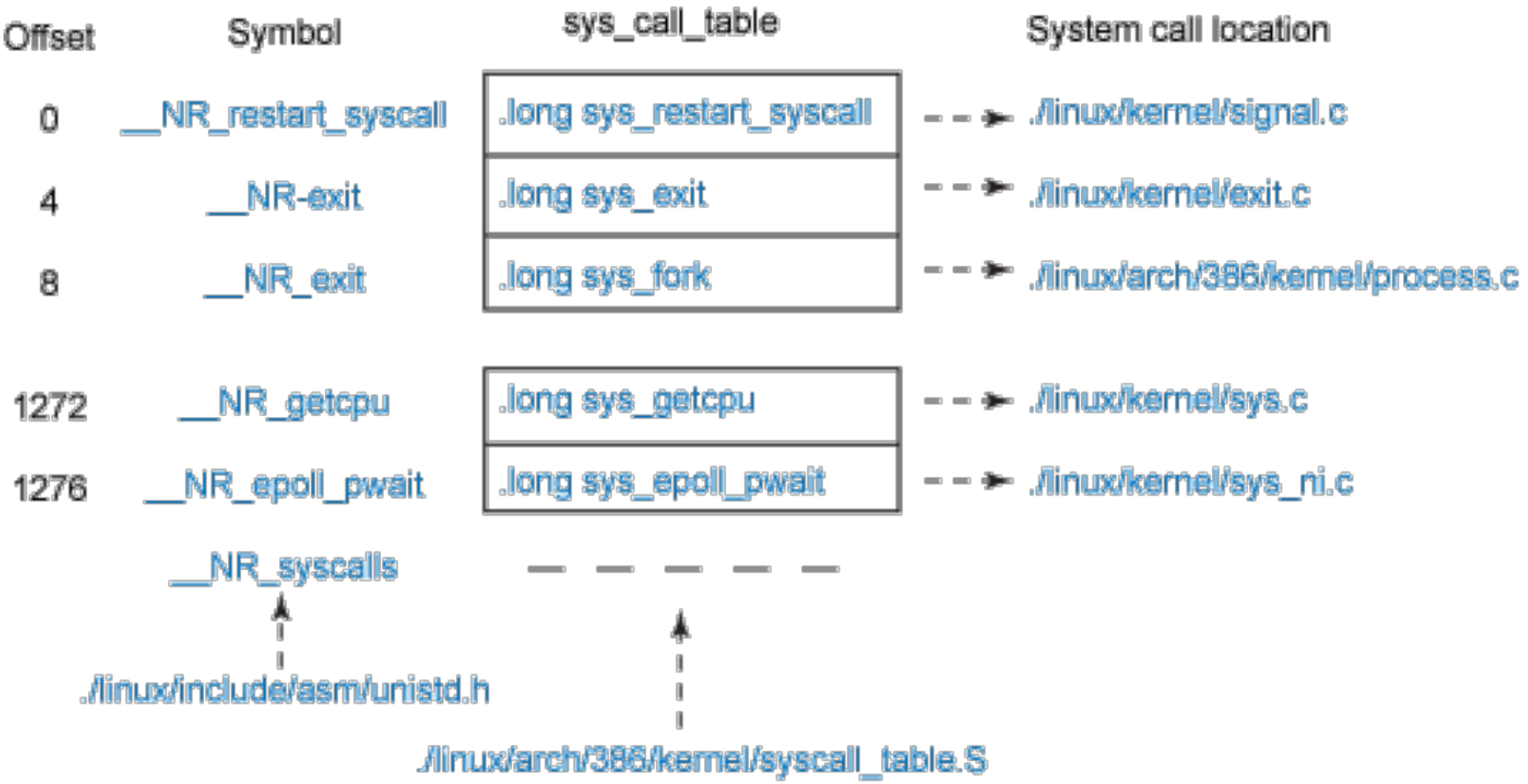
```
#define __NR_restart_syscall     0
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
   ……
#define __NR_ioprio_get          290
#define __NR_inotify_init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

#define NR_syscalls 294
```

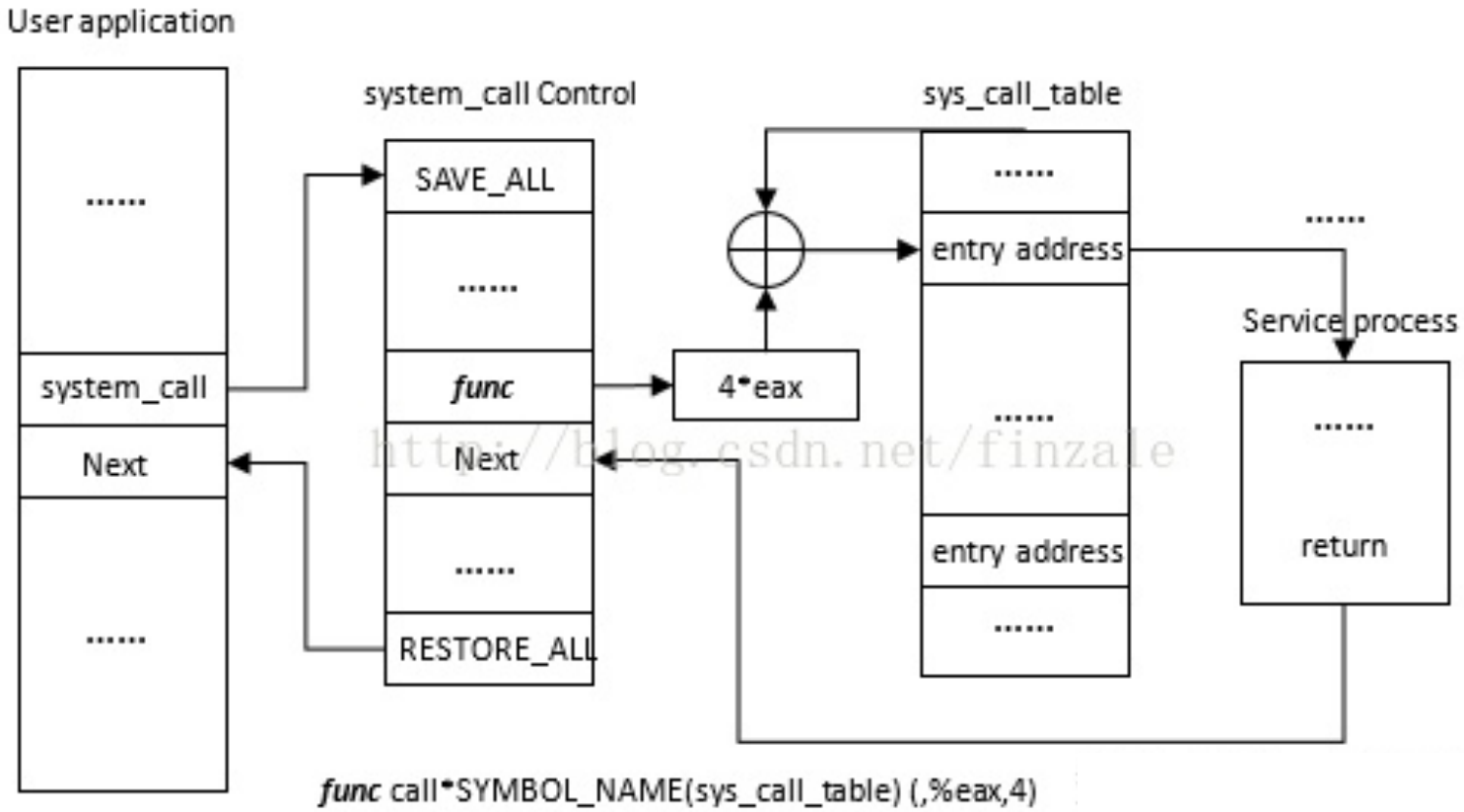
定义每个系统调用的编号

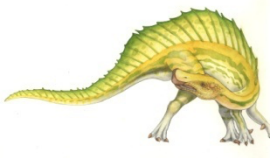


系统调用编号


Offset
Symbol
sys_call_table
System call location
0
__NR_restart_syscall
.long sys_restart_syscall
./linux/kernel/signal.c
4
__NR-exit
.long sys_exit
./linux/kernel/exit.c
8
__NR_exit
.long sys_fork
./linux/arch/386/kernel/process.c
1272
__NR_getcpu
.long sys_getcpu
./linux/kernel/sys.c
1276
__NR_epoll_pwait
.long sys_epoll_pwait
./linux/kernel/sys_ni.c
__NR_syscalls
./linux/include/asm/unistd.h
./linux/arch/386/kernel/syscall_table.S

系统调用号





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。

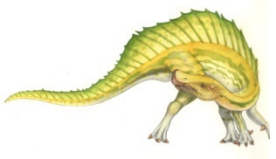
Include/asm-i386/unistd.h

```
#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：

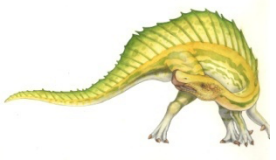
  _syscall3(int, open, const char * pathname, int flag, int mode)

  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

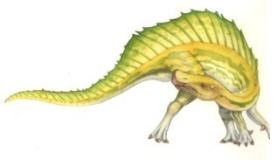
例：系统调用getuid()的实现

- 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。

添加一个系统调用mysyscall (续)

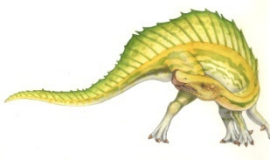
- 缺页统计：
  - 每发生一次缺页都要进入缺页中断服务函数do_page_fault一次，所以可以认为执行该函数的次数就是系统发生缺页的次数。
  - 可以定义一个全局变量pfcount作为计数变量，在执行do_page_fault时，该变量值加1。
  - 在当前进程控制块中定义一个变量pf记录当前进程缺页次数，在执行do_page_fault时，这个变量值加1。

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

基本步骤

- 更新unistd.h
  - define __NR_mysyscall 223
  - __SYSCALL(__NR_mysyscall,sys_mysyscall)
- 更新syscall_table.S
  - 223 i386 mysyscall sys_mysyscall
  - 224 i386 gettid sys_gettid
  - 225 i386 readahead sys_readahead



基本步骤

- 实现mysyscall
  - asmlinkage int sys_mysyscall(void)
  - {
  - …//printk(“当前进程缺页次数：%lu,curre->pf”)
  - return 0;
  - }
- 重新编译内核

基本步骤

- 编写一个用户程序
  - #include <linux/unistd.h>
  - # include <sys/syscall.h>
  - #define __NR_ mysyscall 223
  - int main()
  - {
  - syscall(__NR_mysyscall);   /*或syscall(223) */
  - ……
  - }