Linux操作系统分析
Chapter 6 系统调用

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

October 24, 2014

Outline

Outline

系统调用的意义

- 操作系统为用户态进程与硬件设备进行交互提供了一组接口——系统调用
  - 把用户从底层的硬件编程中解放出来
  - 极大的提高了系统的安全性
  - 使用户程序具有可移植性
- 在Linux用户态，通过int 0x80陷入内核以执行系统调用。
- 为避免程序员使用低级的汇编语言编程，通常使用C库封装后的API接口。

API和系统调用

- 应用编程接口(application program interface, API)和系统调用是不同的
  - API只是一个函数定义
  - 系统调用通过软中断向内核发出一个明确的请求
- Libc库定义的一些API引用了封装例程(wrapper routine，唯一目的就是发布系统调用)
  - 一般每个系统调用对应一个封装例程
  - 库再用这些封装例程定义出给用户的API
- 不是每个API都对应一个特定的系统调用。
  - API可能直接提供用户态的服务，如一些数学函数
  - 一个单独的API可能调用几个系统调用
  - 不同的API可能调用了同一个系统调用

API和系统调用

- API的返回值
  - 大部分封装例程返回一个整数，其值的含义依赖于相应的系统调用
  - -1在多数情况下表示内核不能满足进程的请求
  - Libc中定义的errno变量包含特定的出错码

以open和creat为例

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode\_t mode);
int creat(const char *pathname, mode\_t mode);

RETURN VALUE
    open() and creat() return the new file descriptor, or -1 if an error occurred (in
which case, errno is set appropriately).
```

系统调用程序及服务例程

- 当用户态进程调用一个系统调用时，CPU切换到内核态并开始执行相应的内核函数
  - 在Linux中是通过执行int $0x80来执行系统调用的，这条汇编指令产生向量为128的编程异常（回忆，trap_init中系统调用入口的初始化）
  - Intel Pentium II中引入了sysenter指令（快速系统调用），2.6已经支持（本课程不考虑这个）
- 传参：
  内核实现了很多不同的系统调用，进程必须指明需要哪个系统调用，这需要传递一个名为系统调用号的参数
  - Linux使用eax寄存器传递系统调用号

系统调用程序及服务例程

- 所有的系统调用返回一个整数值。
  - 正数或0表示系统调用成功结束
  - 负数表示一个出错条件

以fs/open.c::sys_open为例

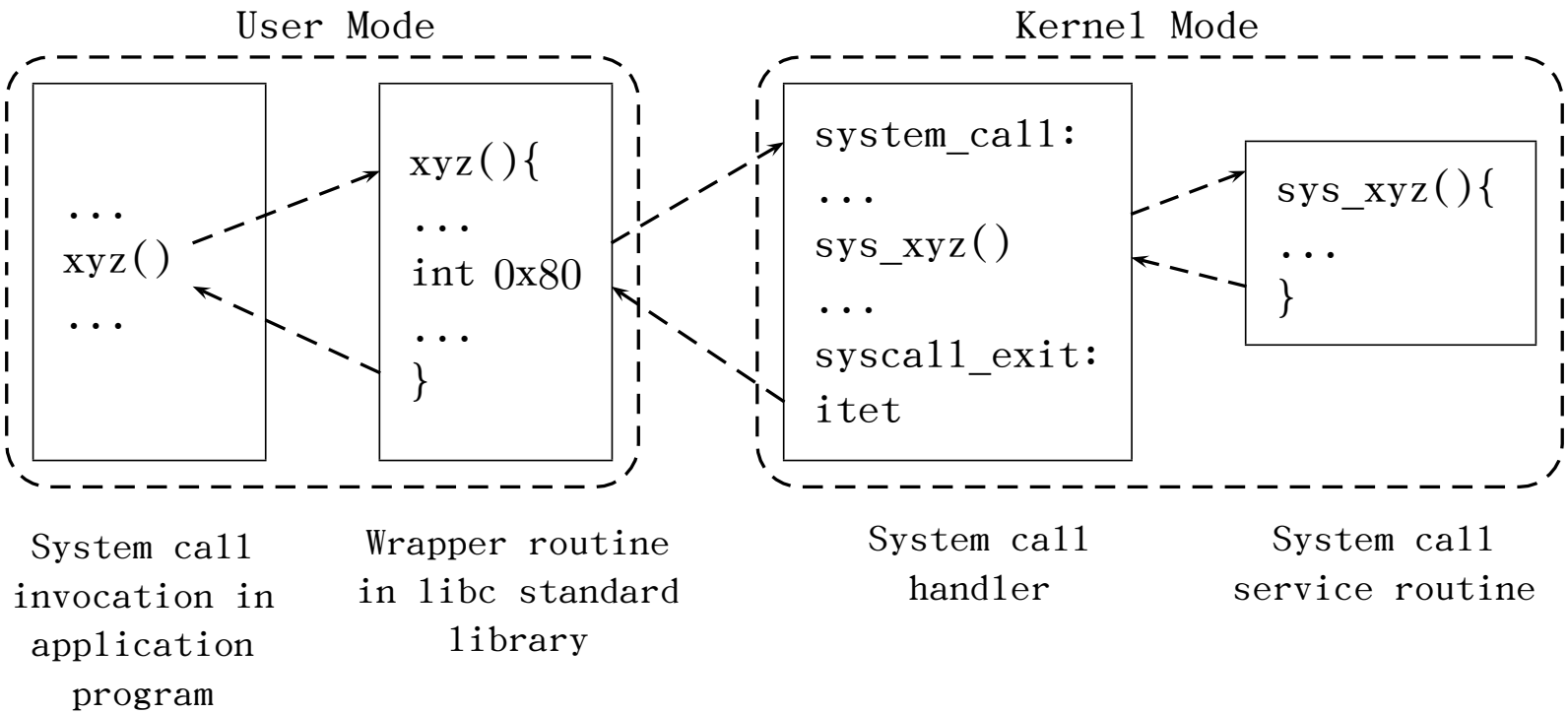
```
asmlinkage long sys_open(const char __user *filename, int flags, int mode) {...}
```

- 系统调用的返回值与封装例程返回值的约定不同
  - 内核没有设置或使用errno变量
  - 封装例程在获得系统调用返回值之后设置这个变量
  - 当系统调用出错时，返回的那个负值被存放在errno变量中返回给应用程序

系统调用程序及服务例程

- 系统调用处理程序也和其他异常处理程序的结构类似
  1. 在进程的内核态堆栈中保存大多数寄存器的内容
     (即保存恢复进程到用户态执行所需要的上下文)
  2. 调用相应的系统调用服务例程sys_xxx处理系统调用
  3. 通过ret_from_sys_call()从系统调用返回

应用程序、封装例程、系统调用处理程序及系统调用服务例程之间的关系



Outline

Outline

2.1 系统调用分派表

- 为了把系统调用号与相应的服务例程关联起来，内核定义了一个系统调用分派表(dispatch table)。
- 这个表存放在sys_call_table数组中，有若干个表项(2.6.26中，总共是327个表项)：
  - 第n个表项对应系统调用号为n的服务例程的入口
- 观察
  - sys_call_table（syscall_table_32.S以及entry_32.S最后）
  - 系统调用分派表的大小：syscall_table_size
  - 系统调用的个数：nr_syscalls

arch/x86/kernel/entry_32.S:

```
#define nr_syscalls ((syscall_table_size)/4)
....
section .rodata," a"
#include " syscall_table_32.S"
syscall_table_size=(.-sys_call_table)
```

2.1 系统调用分派表

syscall_table_32.S

```
ENTRY(sys_call_table)
    .long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting
*/
    .long sys_exit
    .long sys_fork
    .long sys_read
    .long sys_write
    .long sys_open /* 5 */
    .long sys_close
    .long sys_waitpid
    .....
    long sys_getpid /* 20 */
    ...
    long sys_eventfd
    .long sys_fallocate
    .long sys_timerfd_settime /* 325 */
    .long sys_timerfd_gettime
```

Outline

2.2.1 系统入口初始化

- Linux使用IDT表的第0x80(=128)项（SYSCALL_VECTOR）作为系统调用的入口，参见include/asm-x86/mach-default/irq_vectors.h
- Linux初始化期间使用entry_32.S::system_call来初始化系统调用总入口，参见arch/x86/kernel/traps_32.c::trap_init()

```
#define SYSCALL_VECTOR 0x80
```

```
set_system_gate(SYSCALL_VECTOR, &system_call);
```

```
static inline void set_system_gate(unsigned int n, void *addr) {
    ...
    _set_gate(n, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
    ...
}
```

- 使用DPL(描述符特权级）为3的陷阱门：
  - 进入system_call时处于开中断状态
  - 允许用户态进程访问这个门，即在用户程序中使用int $0x80是合法的

2.2.2 system_call()函数

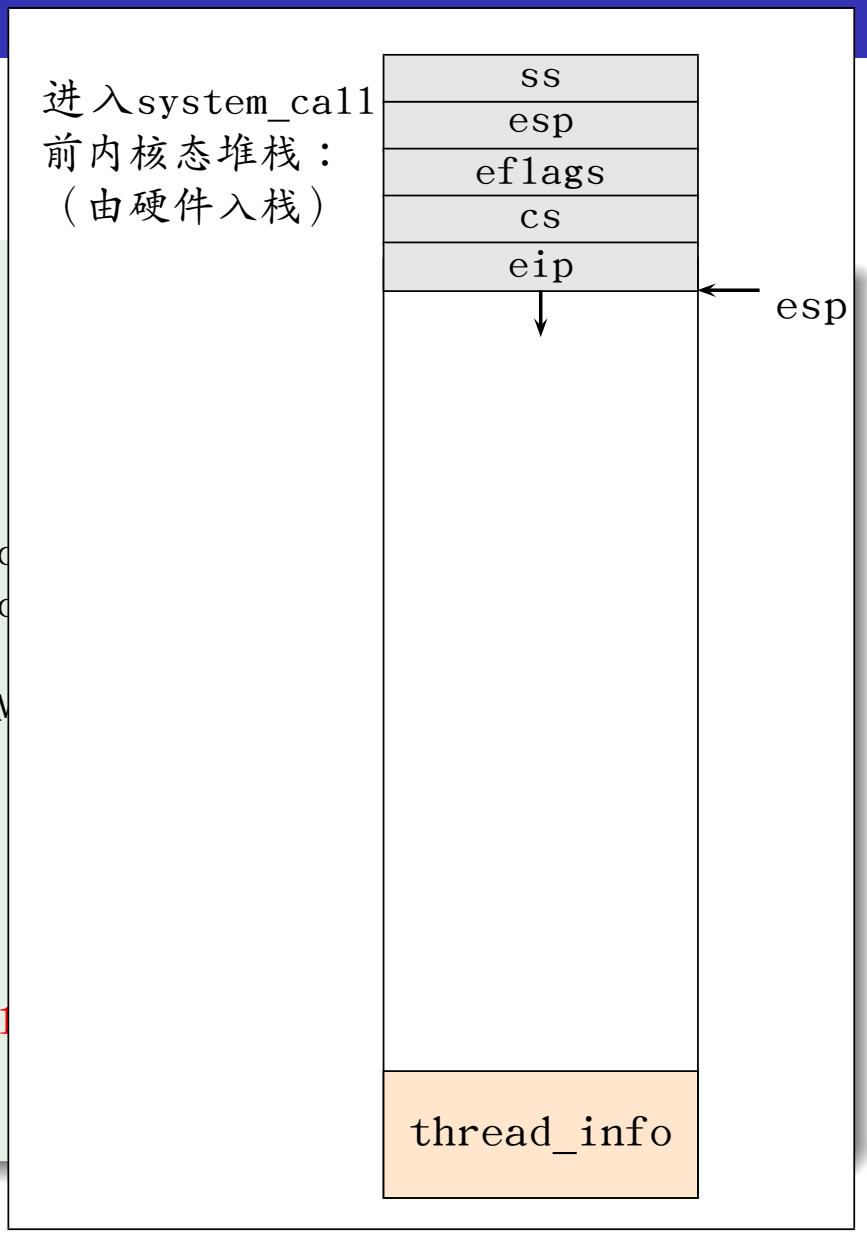
- 参见entry_32.S，精简后如下：

```
# system call handler stub
ENTRY(system_call)
    pushl %eax # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
    # system call tracing in operation / emulation
    /* Note, _TIF_SECCOMP is bit number 8, and so it needs testw and not testb */
    testw
$(_TIF_SYSCALL_EMU|_TIF_SYSCALL_TRACE|_TIF_SECCOMP|_TIF_SYSCALL_AUDIT),TI_flags(%ebp)
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp) # store the return value
syscall_exit:
    ...
```

2.2.2 system_call()函数

- 参见entry_32.S，精简后如下：

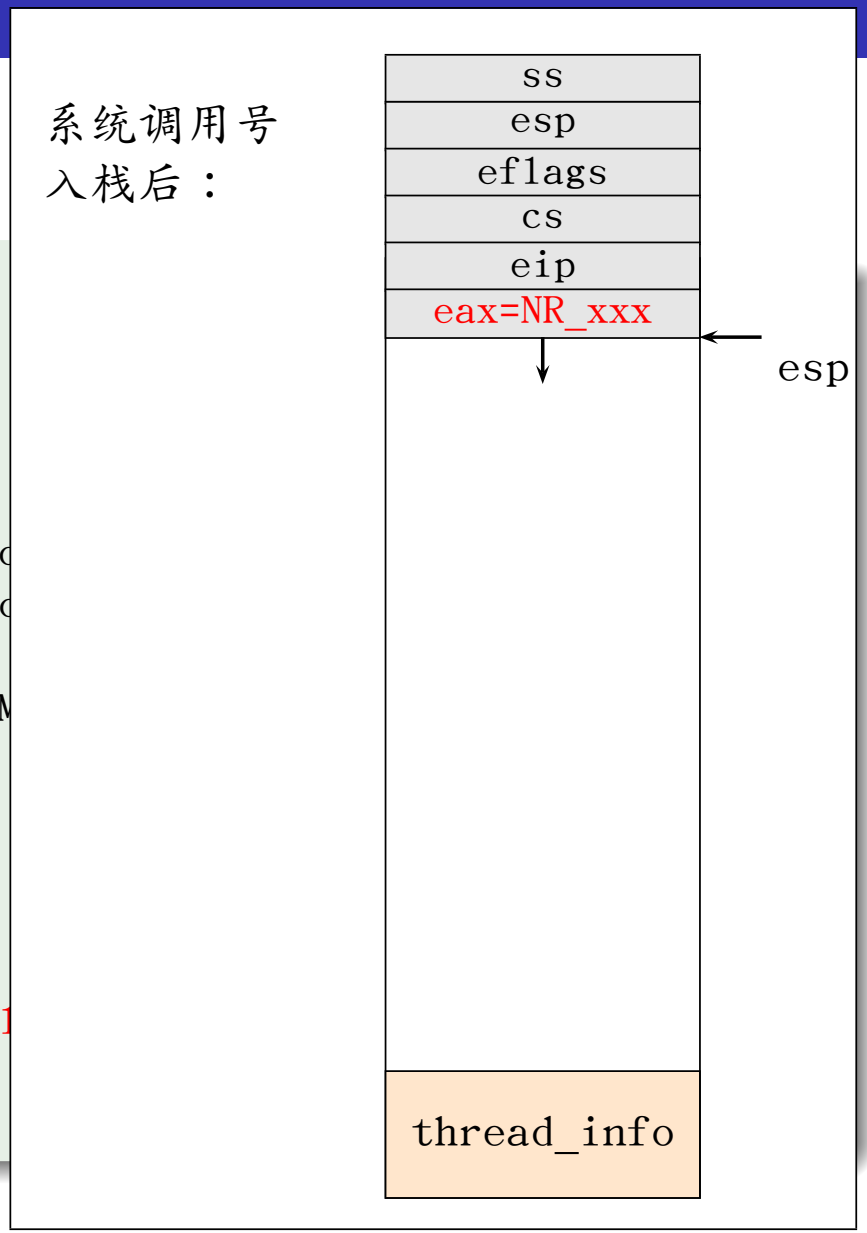
```
# system call handler stub
ENTRY(system_call)
    pushl %eax # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
    # system call tracing in operation / emulatic
    /* Note, _TIF_SECCOMP is bit number 8, and sc
    testw
$(_TIF_SYSCALL_EMU|_TIF_SYSCALL_TRACE|_TIF_SECCOM
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp) # store the return val
syscall_exit:
    ...
```



2.2.2 system_call()函数

- 参见entry_32.S，精简后如下：

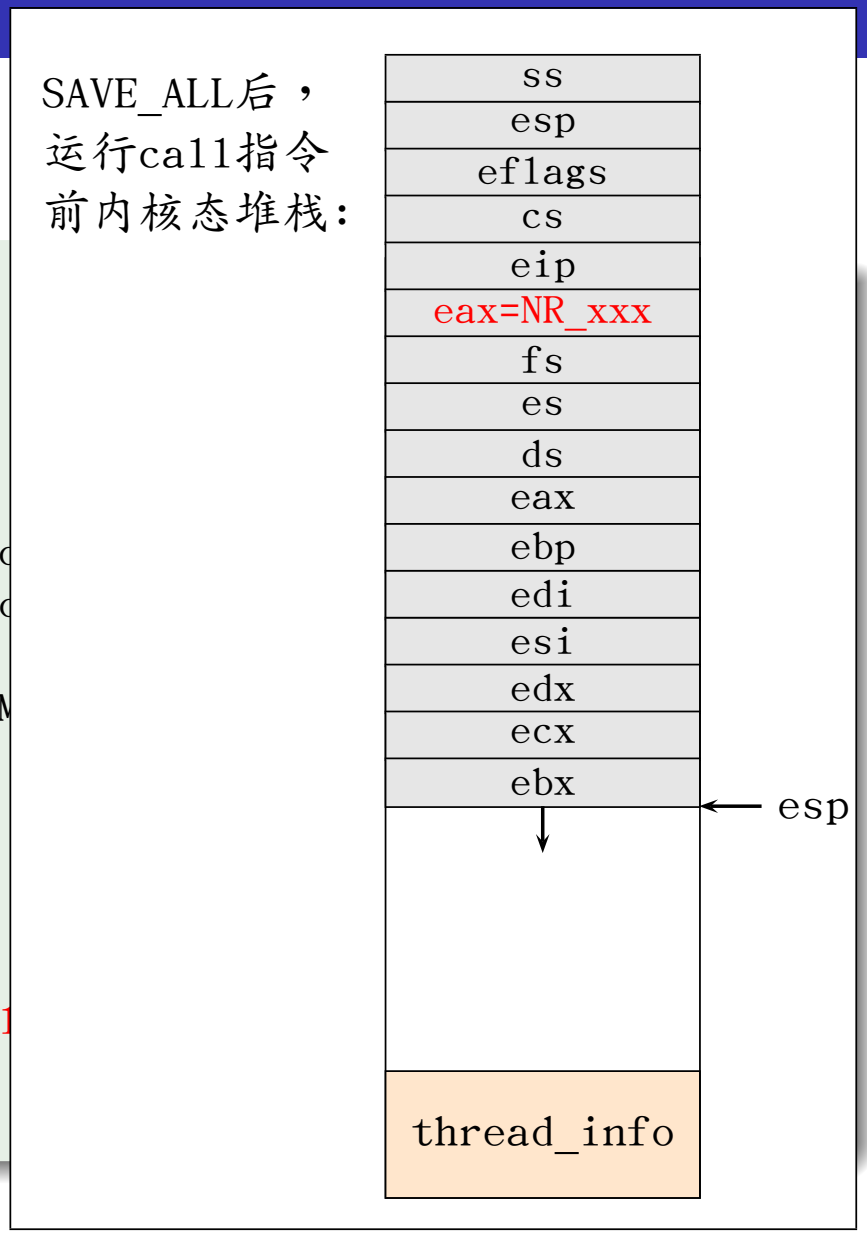
```
# system call handler stub
ENTRY(system_call)
    pushl %eax # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
    # system call tracing in operation / emulatic
    /* Note, _TIF_SECCOMP is bit number 8, and sc
    testw
$(_TIF_SYSCALL_EMU|_TIF_SYSCALL_TRACE|_TIF_SECCOM
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp) # store the return val
syscall_exit:
    ...
```



2.2.2 system_call()函数

- 参见entry_32.S，精简后如下：

```
# system call handler stub
ENTRY(system_call)
    pushl %eax # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
    # system call tracing in operation / emulatic
    /* Note, _TIF_SECCOMP is bit number 8, and sc
    testw
$(_TIF_SYSCALL_EMU|_TIF_SYSCALL_TRACE|_TIF_SECCOM
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp) # store the return val
syscall_exit:
    ...
```



2.2.2 system_call()函数

- 以fork为例，在arch/x86/kernel/process_32.c中sys_fork被定义为：

```
asmlinkage int sys_fork(struct pt_regs regs) {...}
```

  - 其中，asmlinkage的定义：参见include/asm-x86/linkage.h

```
#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
```

  - 因此，sys_fork()被定义为从堆栈传参，需要栈上准备好一个pt_regs结构，正好栈上有。

Outline

2.3.1 系统调用的参数传递

- 系统调用也需要输入参数，例如
  - 实际的值
  - 用户态进程地址空间的变量的地址
  - 甚至是包含指向用户态函数的指针的数据结构的地址
- system_call是linux中所有系统调用的入口点，每个系统调用至少有一个参数，即由eax传递的系统调用号
  - 例如：一个应用程序调用write()，那么在执行int $0x80前必须把eax寄存器的值置为4(即__NR_write)。
  - 这个寄存器的设置是libc库中的封装例程进行的，因此用户一般不关心系统调用号
    - 演示：对C库进行反汇编，查看int $0x80
  - 进入system_call之后，eax携带的系统调用号将立即入栈

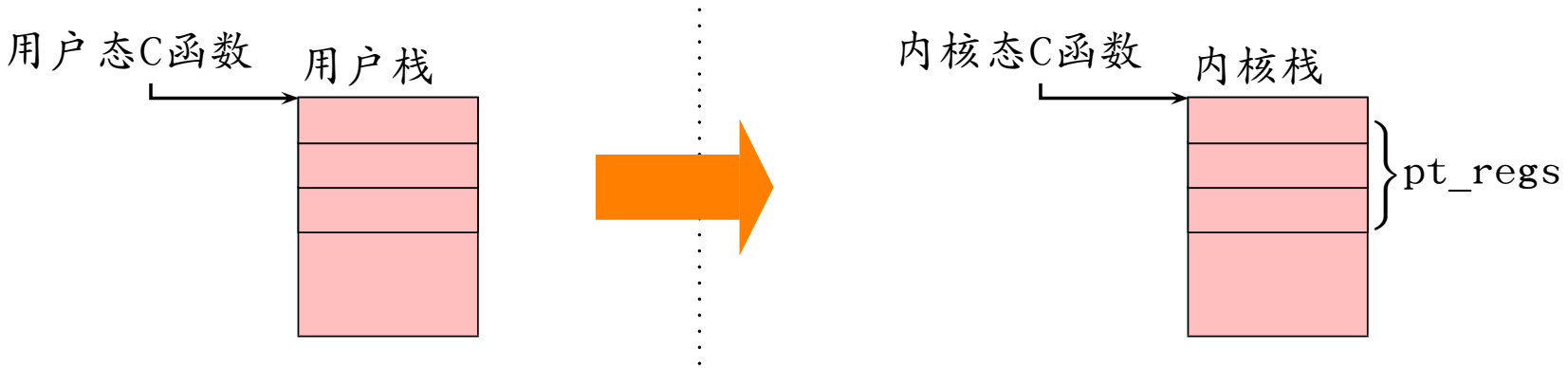
2.3.1 系统调用的参数传递

- 很多系统调用需要不止一个参数，例如

```
asmlinkage ssize_t sys_write(unsigned int fd, const char __user *buf, size_t count);
```

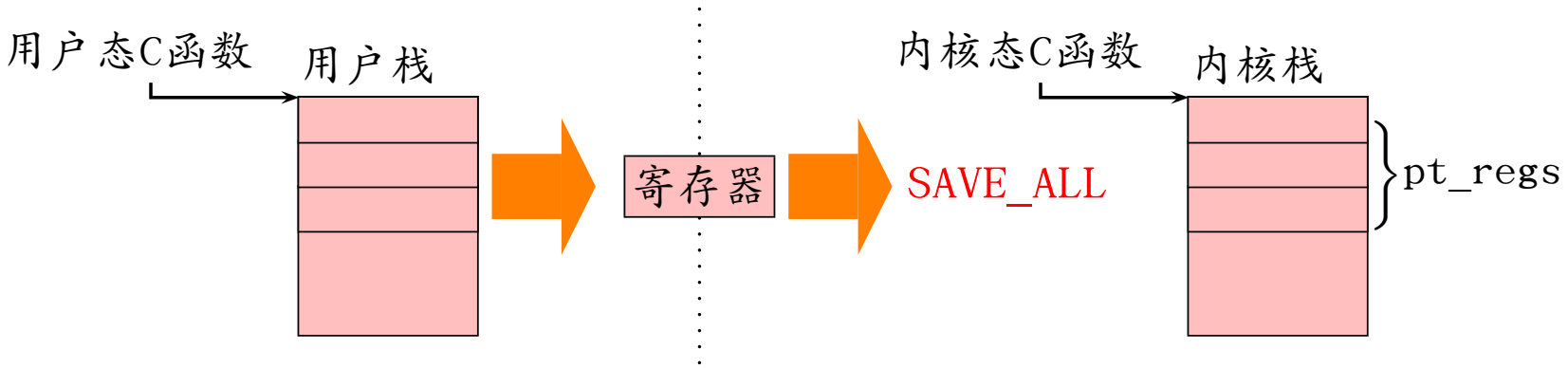
```
ssize_t write(int fd, const void *buf, size_t count);
```

- 普通C函数的参数传递是通过把参数值写入堆栈(用户态堆栈或内核态堆栈)来实现的。但因为系统调用是一种特殊函数，它由用户态进入了内核态，发生了从用户栈到内核栈的切换，所以既不能使用用户栈也不能直接使用内核栈



2.3.1 系统调用的参数传递

- 解决方法：
  1. 在int $0x80汇编指令之前，系统调用的参数被写入CPU的寄存器。
  2. 在进入内核态调用系统调用服务例程之前，内核把存放在CPU寄存器中的参数拷贝到内核态堆栈中，即形成pt_regs。



- pt_regs中的一些寄存器可以被用来传递参数或者pt_regs本身就是参数，例如：

```
asmlinkage ssize_t sys_write(unsigned int fd, const char __user *buf, size_t count);
```

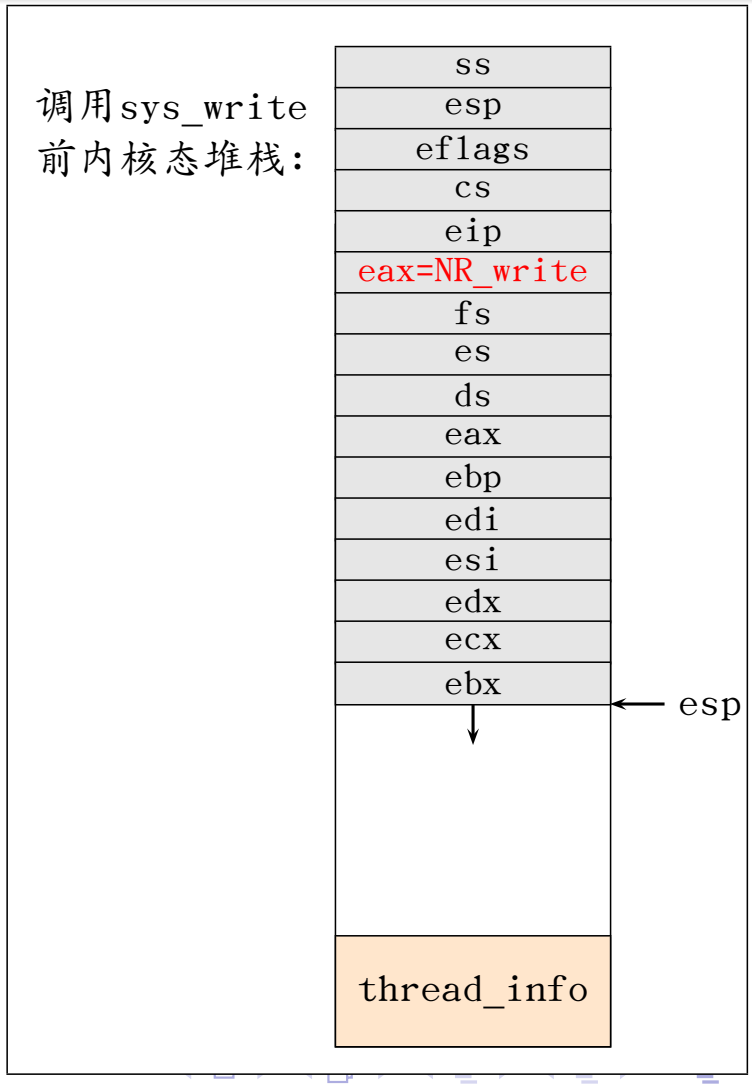
```
asmlinkage int sys_fork(struct pt_regs regs);
```

2.3.1 系统调用的参数传递

```
asmlinkage ssize_t sys_write(unsigned int fd, const char __user *buf, size_t count);
```

- 以处理write()系统调用的sys_write服务例程为例：
  - 该函数期望在栈顶找到fd，buf和count这三个参数

- 在封装write()的封装例程中，将会在ebx、ecx和edx寄存器中分别填入这些参数的值，然后在进入system_call时，SAVE_ALL会把这些寄存器保存在堆栈中，进入sys_write服务例程后，就可以在相应的位置找到这些参数



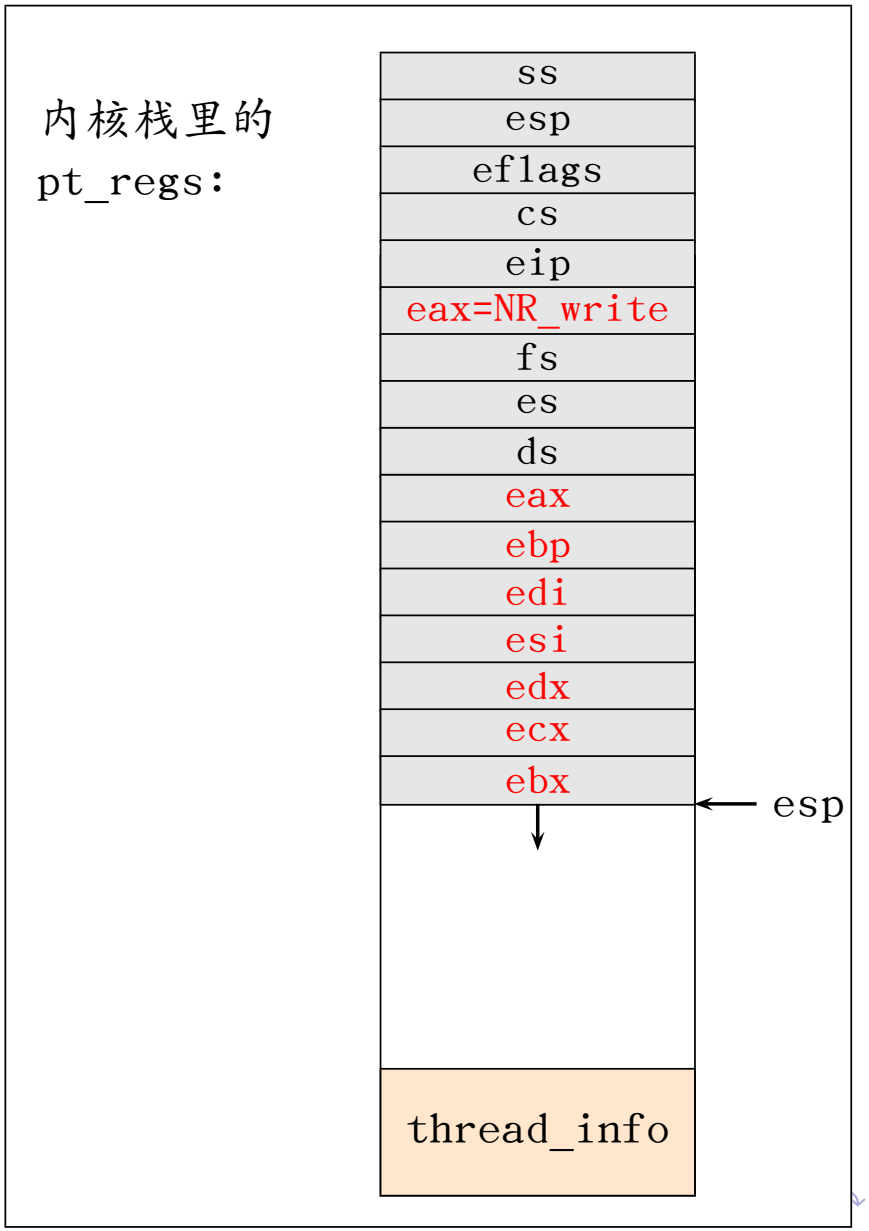
2.3.1 系统调用的参数传递

老版本write()的部分反汇编码

```
197897 000bed90 <__write>:
197898 bed90:  65 83 3d 0c 00 00 00     cmpl $0x0,%gs:0xc
197899 bed97:  00
197900 bed98:  75 1d                    jne bedb7 <__write+0x27>
197901 bed9a:  53                       push %ebx
197902 bed9b:  8b 54 24 10              mov 0x10(%esp),%edx
197903 bed9f:  8b 4c 24 0c              mov 0xc(%esp),%ecx
197904 beda3:  8b 5c 24 08              mov 0x8(%esp),%ebx
197905 beda7:  b8 04 00 00 00           mov $0x4,%eax
197906 bedac:  cd 80                    int $0x80
197907 bedae:  5b                       pop %ebx
197908 bedaf:  3d 01 f0 ff ff           cmp $0xfffff001,%eax
197909 bedb4:  73 2d                    jae bede3 <__write+0x53>
197910 bedb6:  c3                       ret
.............
```

2.3.1 系统调用的参数传递

- 根据pt_regs，使用寄存器传递参数具有如下限制：
  1. 每个参数的长度不能超过寄存器的长度，即32位
  2. 在系统调用号（eax）之外，参数的个数不能超过6个（ebx，ecx，edx，esi，edi，ebp）
     - ？超过6个怎么办？



2.3.2 系统调用返回值的传递

- 服务例程的返回值将被写入eax寄存器中
  - 这是在执行“return”指令时，由编译器自动完成的
- sys_xxx返回后，eax寄存器的值将立即保存到pt_regs结构的eax寄存器中，并在恢复用户上下文时，伴随eax传回用户态封装函数

```
# system call handler stub
ENTRY(system_call)
    ...
syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,PT_EAX(%esp) # store the return value
syscall_exit:
    ...
```

- 例如sys_write()的返回值。

Outline

2.4 系统调用参数的验证

- 在内核打算满足用户的请求之前，必须仔细的检查所有的系统调用参数
  - 比如前面的write()系统调用，fd参数是一个文件描述符，sys_write()必须检查这个fd是否确实是以前已打开文件的一个文件描述符，进程是否有向fd指向的文件的写权限，如果有条件不成立，那这个处理程序必须返回一个负数

2.4 系统调用参数的验证

- 只要一个参数指定的是地址，那么内核必须检查它是否在这个进程的地址空间之内，有两种验证方法：
  1. 验证这个线性地址是否属于进程的地址空间
  2. 仅仅验证这个线性地址小于PAGE_OFFSET
- 对于第一种方法：
  - 费时
  - 大多数情况下，不必要
- 对于第二种方法：
  - 高效
  - 可以在后续的执行过程中，很自然的捕获到出错的情况
- 从linux2.2开始执行第二种检查

对用户地址参数的粗略验证

- 内核代码可以访问到所有的内存
- 必须防止用户将一个内核地址作为参数传递给内核，因为这将导致它借用内核代码来读写任意内存
- 在include/asm-x86/uaccess_32.h中：

```
/**
 * access_ok: - Checks if a user space pointer is valid
 * @type: Type of access: %VERIFY_READ or %VERIFY_WRITE. Note that
 * %VERIFY_WRITE is a superset of %VERIFY_READ - if it is safe
 * to write to a block, it is always safe to read from it.
 * @addr: User space pointer to start of block to check
 * @size: Size of block to check
 *...
 * Returns true (nonzero) if the memory block may be valid, false (zero)
 * if it is definitely invalid.
 *
 * Note that, depending on architecture, this function probably just
 * checks that the pointer is in the user space range - after calling
 * this function, memory access functions may still return -EFAULT.
 */
#define access_ok(type, addr, size) (likely(__range_ok(addr, size) == 0))
```

对用户地址参数的粗略验证

```
/*
 * Test whether a block of memory is a valid user space address.
 * Returns 0 if the range is valid, nonzero otherwise.
 *
 * This is equivalent to the following test:
 * (u33)addr + (u33)size >= (u33)current->addr_limit.seg
 *
 * This needs 33-bit arithmetic. We have a carry...
 */
#define __range_ok(addr, size) \
({ \
    unsigned long flag, roksum; \
    __chk_user_ptr(addr); \
    asm("add1 %3,%1 ; sbb1 %0,%0; cmp1 %1,%4; sbb1 $0,%0" \
        :"=&r" (flag), "=r" (roksum) \
        :"1" (addr), "g" ((int)(size)), \
         "rm" (current_thread_info()->addr_limit.seg)); \
    flag; \
})
```

对用户地址参数的粗略验证

- 检查方法：
  - 最高地址：addr+size-1
    1. 是否超出3G边界
    2. 是否超出当前进程的地址边界
  - 对于用户进程：不大于3G
  - 对于内核线程：可以使用整个4G

Outline

访问进程的地址空间

- 系统调用服务例程需要非常频繁的读写进程地址空间的数据

Function	Action
get_user __get_user	Reads an integer value from user space(1, 2, or 4 bytes)
put_user __put_user	Write an integer value to user space (1, 2, or 4 bytes)
copy_from_user __copy_from_user	Copies a block of arbitrary size from user space
copy_to_user __copy_to_user	Copies a block of arbitrary to user space
strncpy_from_user __strncpy_from_user	Copies a null-terminated string from user space
strlen_user strnlen_user	Returns the length of a null-terminated string in user space
clear_user __clear_user	Fills a memory area in user space with zeros

访问进程地址空间时的缺页

- 内核对进程传递的地址参数只进行粗略的检查
- 访问进程地址空间时的缺页，可以有多种情况，如：
  1. 合理的缺页：来自虚存技术
     - 页框不存在或者写时复制
  2. 由于错误引起的缺页
  3. 由于非法引起的缺页

非法缺页的判定

- 内核规定，只有少数几个函数/宏会访问用户地址空间。因此对于内核发生的非法缺页，一定来自于这些函数/宏
- 可以将访问用户地址空间的指令地址一一列举出来，当发生非法缺页时，根据引起出错的指令地址来定位
- Linux使用了异常表的概念
  - __ex_table, __start___ex_table, __stop___ex_table

在kernel/extable.c中

```
extern struct exception_table_entry __start___ex_table[];
extern struct exception_table_entry __stop___ex_table[];
```

非法缺页的判定

- __ex_table的表项

在include/asm-x86/uaccess_32.h中

```
/*
 * The exception table consists of pairs of addresses: the first is the
 * address of an instruction that is allowed to fault, and the second is
 * the address at which the program should continue. No registers are
 * modified, so it is entirely up to the continuation code to figure out
 * what to do.
 *
 * All the routines below use bits of fixup code that are out of line
 * with the main instruction path. This means when everything is well,
 * we don't even have to jump over them. Further, they do not intrude
 * on our cache or tlb entries.
 */
struct exception_table_entry {
    unsigned long insn, fixup;
};
```

- insn为可能引起出错的指令地址；
  fixup为修正代码入口地址

非法缺页的判定

- 异常表项的查找
  - search_exception_table()根据给定的出错指令地址，找到对应的异常表项

在kernel/extable.c中

```
/* Given an address, look for it in the exception tables. */
const struct exception_table_entry *search_exception_tables(unsigned long addr) {
    const struct exception_table_entry *e;
    e = search_extable(__start___ex_table, __stop___ex_table-1, addr);
    if (!e)
        e = search_module_extables(addr);
    return e;
}
```

非法缺页的判定

- 修正代码的使用
  - fixup_exception()首先调用search_exception_table()找到异常表项，然后将修正代码入口地址填写到pt_regs的eip中

在arch/x86/mm/extable.c中

```
int fixup_exception(struct pt_regs *regs) {
    const struct exception_table_entry *fixup;
    ...
    fixup = search_exception_tables(regs->ip);
    if (fixup) {
        regs->ip = fixup->fixup;
        return 1;
    }
    return 0;
}
```

非法缺页的判定

- 缺页异常对非法缺页的处理
  - 在缺页异常do_page_fault中，若最后发现是非法缺页，就会执行下面的操作

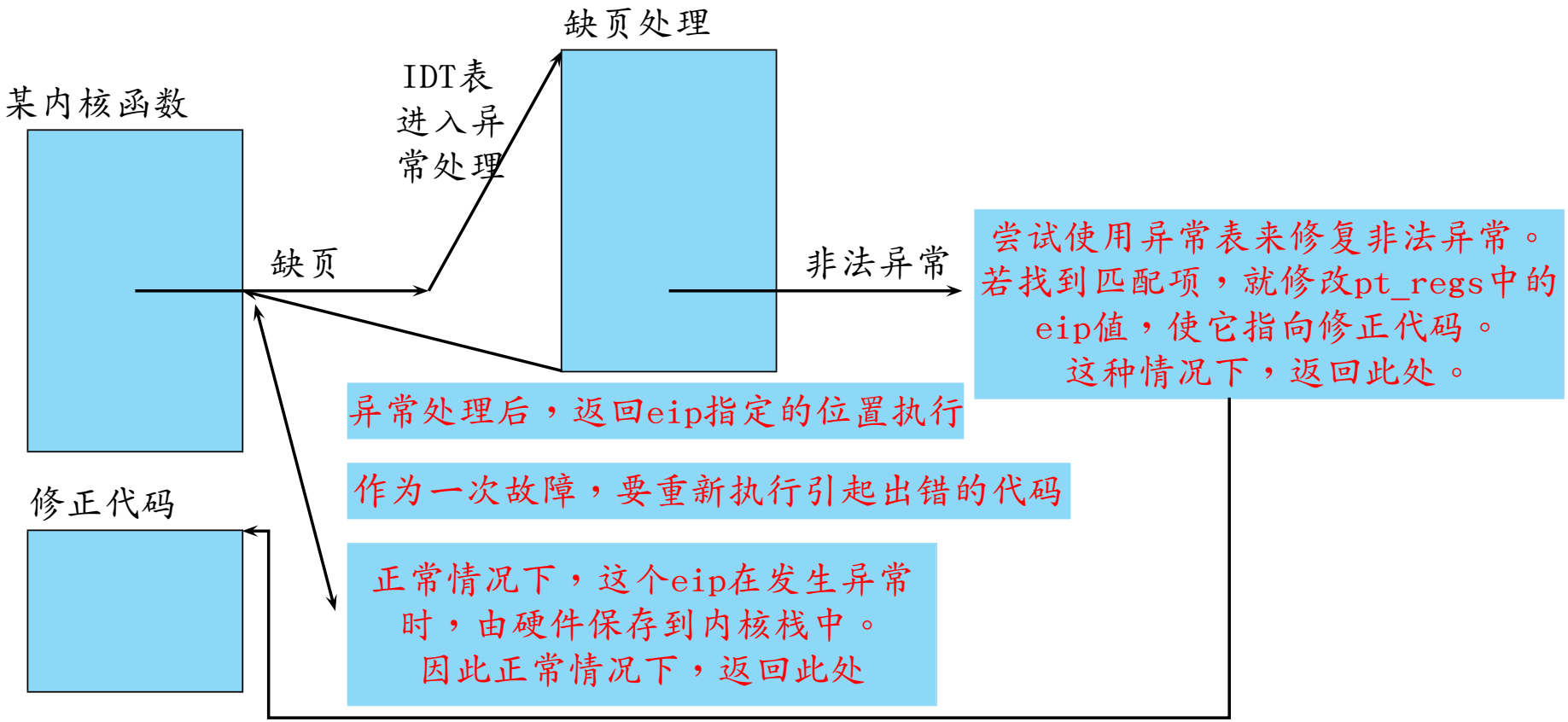
arch/x86/mm/fault.c

```
void __kprobes do_page_fault(struct pt_regs *regs, unsigned long error_code) {
    ...
no_context:
    /* Are we prepared to handle this kernel fault? */
    if (fixup_exception(regs))
        return;
    ...
}
```

  - 该操作尝试使用异常表来处理非法缺页。若处理成功，则pt_regs的eip被修改为修正代码入口地址。

非法缺页的判定

- 缺页异常对非法缺页的处理
  - 处理过程示意图：



异常表的生成和修正代码

- 在源代码中搜索“__ex_table”，你看到了什么？

在arch/x86/kernel/vmlinux_32.lds.S中：

```
...
. = ALIGN(16); /* Exception table */
__ex_table : AT(ADDR(__ex_table) - LOAD_OFFSET) {
    __start___ex_table = .;
    *(__ex_table)
    __stop___ex_table = .;
}
...
```

异常表的生成和修正代码

- 在源代码中搜索“__ex_table”，你看到了什么？

在arch/x86/kernel/vmlinux_32.lds.S中：

```
...
. = ALIGN(16); /* Exception table */
__ex_table : AT(ADDR(__ex_table) - LOAD_OFFSET) {
    __start___ex_table = .;
    *(__ex_table)
    __stop___ex_table = .;
}
...
```

异常表的生成和修正代码

- 以_get_user为例（参见arch/x86/lib/getuser_32.S，精简后）：

```
/* * __get_user_X
 ** Inputs: %eax contains the address
 ** Outputs: %eax is error code (0 or -EFAULT)
 *           %edx contains zero-extended value
 *...
 */
.text
ENTRY(__get_user_1)
    CFI_STARTPROC
    GET_THREAD_INFO(%edx)
    cmpl TI_addr_limit(%edx),%eax
    jae bad_get_user
1: movzbl (%eax),%edx
    xorl %eax,%eax
    ret
    CFI_ENDPROC
ENDPROC(__get_user_1)
```

```
ENTRY(__get_user_2)
    CFI_STARTPROC
    addl $1,%eax
    jc bad_get_user
    GET_THREAD_INFO(%edx)
    cmpl TI_addr_limit(%edx),%eax
    jae bad_get_user
2: movzwl -1(%eax),%edx
    xorl %eax,%eax
    ret
    CFI_ENDPROC
ENDPROC(__get_user_2)
```

异常表的生成和修正代码

```
ENTRY(__get_user_4)
    CFI_STARTPROC
    addl $3,%eax
    jc bad_get_user
    GET_THREAD_INFO(%edx)
    cmpl TI_addr_limit(%edx),%eax
    jae bad_get_user
3:  movl -3(%eax),%edx
    xorl %eax,%eax
    ret
    CFI_ENDPROC
ENDPROC(__get_user_4)
```

```
bad_get_user:
    CFI_STARTPROC
    xorl %edx,%edx
    movl $-14,%eax
    ret
    CFI_ENDPROC
END(bad_get_user)

.section __ex_table,"a"
    .long 1b,bad_get_user
    .long 2b,bad_get_user
    .long 3b,bad_get_user
.previous
```

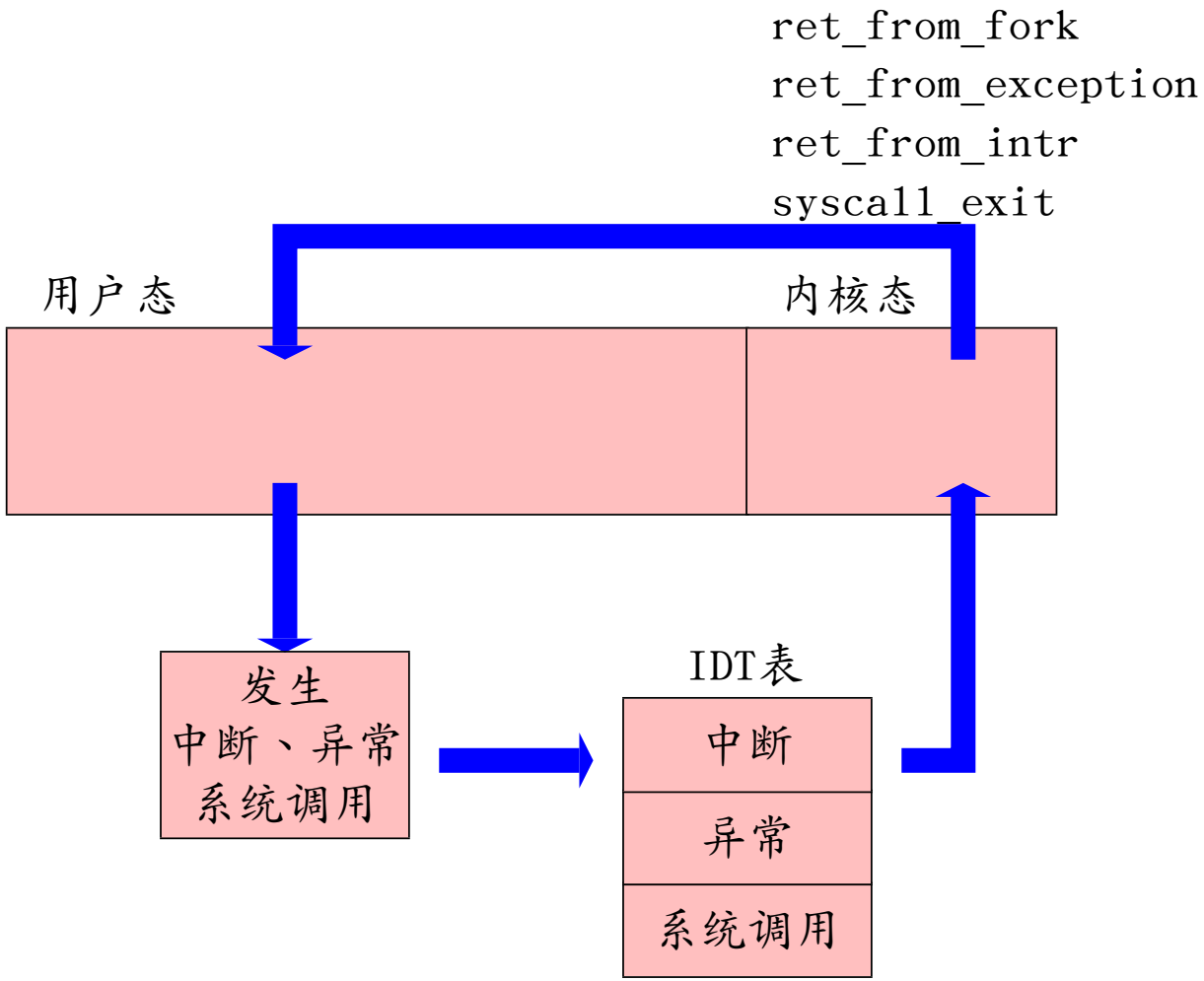
Outline

系统调用的返回

- 系统调用的返回，阅读 arch/x86/kernel/entry_32.S::syscall_exit
- fork的在子进程中的返回，阅读 arch/x86/kernel/entry_32.S::re_from_frok

```
ENTRY(ret_from_fork)
    CFI_STARTPROC
    pushl %eax
    CFI_ADJUST_CFA_OFFSET 4
    call schedule_tail
    GET_THREAD_INFO(%ebp)
    popl %eax
    CFI_ADJUST_CFA_OFFSET -4
    pushl $0x0202 # Reset kernel eflags
    CFI_ADJUST_CFA_OFFSET 4
    popfl
    CFI_ADJUST_CFA_OFFSET -4
    jmp syscall_exit
CFI_ENDPROC END(ret_from_fork)
```

中断、异常、系统调用小结


ret_from_fork
ret_from_exception
ret_from_intr
syscall_exit
用户态
内核态
发生
中断、异常
系统调用
IDT表
中断
异常
系统调用

Outline

Project

- 参见课程主页

作业

1. 什么是系统调用？为什么要有系统调用？
2. Linux-2.6.26中系统调用处理函数根据什么找到系统调用服务例程？
3. Linux-2.6.26中系统调用服务例程的参数从哪里获取？
4. Linux-2.6.26中系统调用服务例程的返回值是如何返回到用户程序中的？

小结

Thanks !

The end.