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

January 14, 2015

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swi指令陷入内核以执行系统调用。
  - 提示：反汇编arm可执行文件，搜索swi命令
- 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swi指令来执行系统调用的
- 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
  - ARM中，系统调用号与swi的操作码混在一起

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

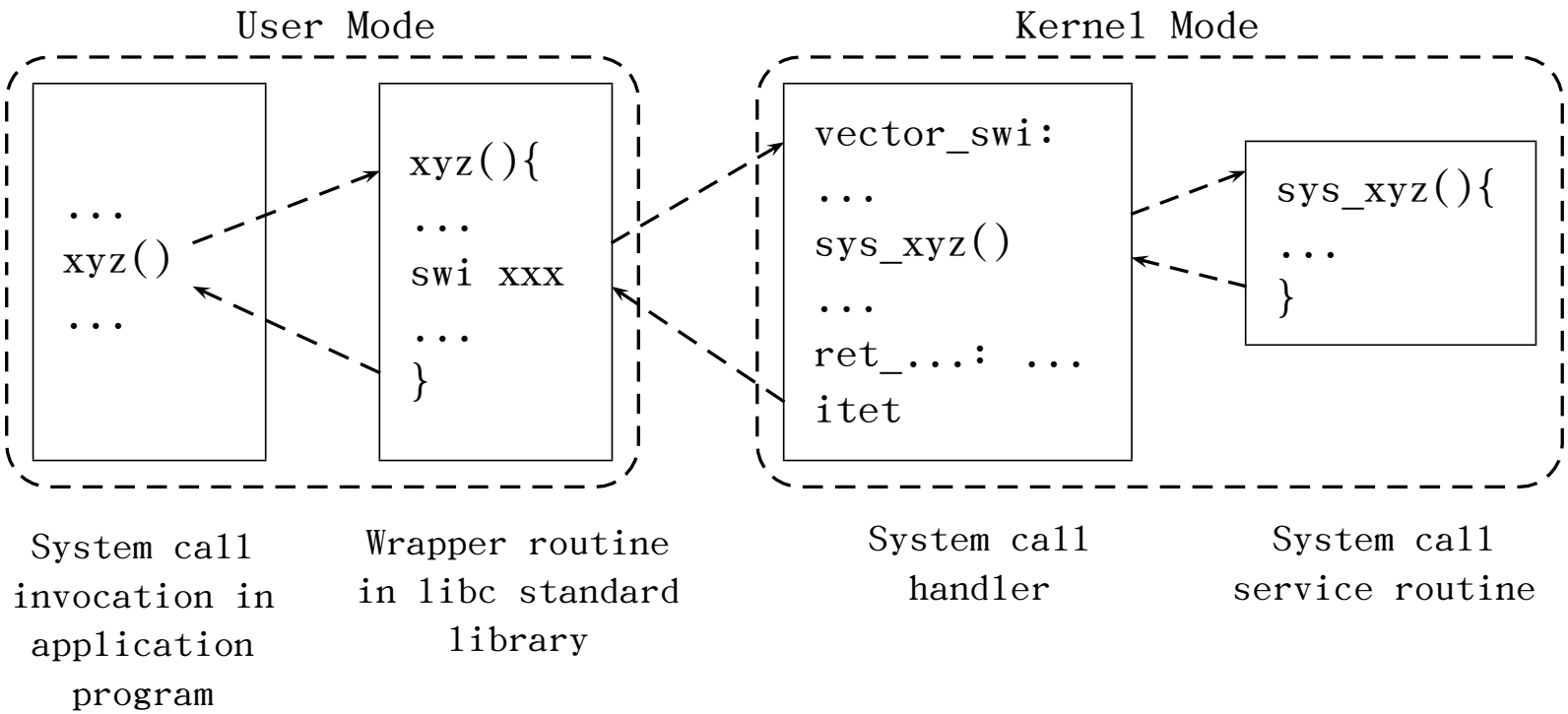
```
asmlinkage long sys_open(const char __user *filename, int flags, int mode) {...}
```

- 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errno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_slow_syscall()或者ret_fast_syscall()等从系统调用返回

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355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（arch/arm/kernel/calls.S以及entry_common.S）
  - 系统调用的个数：NR_syscalls

Outline

vector_swi

- 参见entry_common.S

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
- vector_swi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，即系统调用号
  - 用户一般不关心系统调用号，C库封装例程使用
    - 演示：对C库进行反汇编，查看swi指令
  - 进入vector_swi之后，将处理swi指令以获得系统调用号

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

fs/read_write.c

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

man L 2 write

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

  - 反汇编arm的c库，了解参数是如何传递的

2.3.1 系统调用的参数传递

- 内核中有些sys_xxx的参数与pt_regs有关，例如：

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

  - 则还需要一层封装：

entry_common.S:

```
sys_fork_wrapper:
        add r0, sp, #S_OFF
        b sys_fork
```

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r0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，r0寄存器的值将保存到pt_regs结构的r0寄存器中，并在恢复用户上下文时，伴随r0传回用户态封装函数

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-arm/uaccess.h中：

```
#define access_ok(type, addr, size) (__range_ok(addr, size) == 0)
```

对用户地址参数的粗略验证

```
/* We use 33-bit arithmetic here... */
#define __range_ok(addr,size) ({ \
        unsigned long flag, roksum; \
        __chk_user_ptr(addr); \
        __asm__(" adds %1, %2, %3; sbcccs %1, %1, %0; movcc %0, #0" \
                : " =&r" (flag), " =&r" (roksum) \
                : " r" (addr), " Ir" (size), " 0"
(current_thread_info()->addr_limit) \
                : " cc"); \
        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

- 基本上分别有对应的.S文件对应

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-arm/uaccess.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/arm/mm/extable.c中

```
int fixup_exception(struct pt_regs *regs) {
    const struct exception_table_entry *fixup;

    fixup = search_exception_tables(instruction_pointer(regs));
    if (fixup)
        regs->ARM_pc = fixup->fixup;

    return fixup != NULL;
}
```

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，若最后发现是内核缺页，就会执行下面的操作

arch/arm/mm/fault.c

```
/*
 * Oops. The kernel tried to access some page that wasn't present.
 */
static void __do_kernel_fault(struct mm_struct *mm, unsigned long addr,
              unsigned int fsr, struct pt_regs *regs) {
    /*
     * 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PC被修改为修正代码入口地址。

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/arm/kernel/vmlinux.lds.S中：

```
    ...
    /*
     * The exception fixup table (might need resorting at runtime)
     */
    . = ALIGN(32);
    __start___ex_table = .;
#ifdef CONFIG_MMU
    *(__ex_table)
#endif
    __stop___ex_table = .;
...
```

- 不妨以arch/arm/lib/putuser.S为例，看一看修正代码和__ex_table的定义

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/arm/kernel/vmlinux.lds.S中：

```
    ...
    /*
     * The exception fixup table (might need resorting at runtime)
     */
    . = ALIGN(32);
    __start___ex_table = .;
#ifdef CONFIG_MMU
    *(__ex_table)
#endif
    __stop___ex_table = .;
...
```

- 不妨以arch/arm/lib/putuser.S为例，看一看修正代码和__ex_table的定义

Outline

系统调用的返回

- 系统调用的返回，阅读arch/arm/kernel/entry_common.S::ret_fast_syscall和ret_slow_syscall
- fork的在子进程中的返回，阅读arch/arm/kernel/entry_common.S::ret_from_frok

Outline

小结

1. 系统调用和API
2. 系统调用机制的实现
   - 系统调用分派表
   - 系统调用处理向量的入口vector_swi
   - 系统调用的参数传递
   - 系统调用参数的验证
   - 如何访问进程的地址空间
   - 系统调用的返回
3. 小结和作业

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？

Thanks !

The end.