東北大學秦皇島分校
Northeastern University at Qinhuangdao

Linux系统与内核分析

--系统调用

于七龙

目录

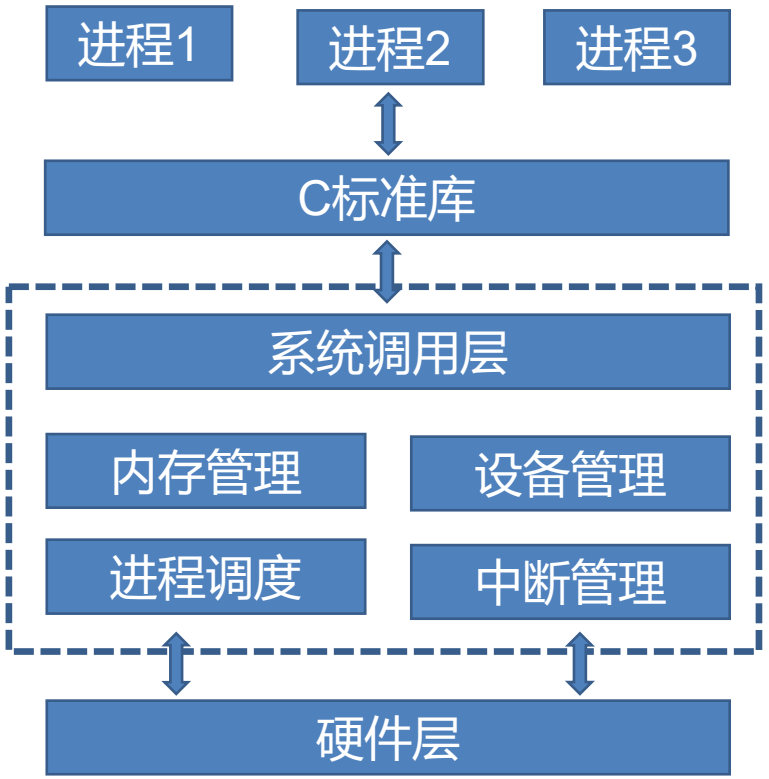
Part 1 系统调用概述

系统调用

- 系统调用是操作系统为用户态运行的进程和硬件设备(如CPU、磁盘、打印机)进行交互提供的一组接口
  - 使得编程更加容易，程序员不用关心硬件设备特性
  - 提高了系统的安全性，内核可在满足某个请求前在接口级检查请求的正确性
  - 最重要的是使得程序更具有可移植性

系统调用

- 系统调用在操作系统中的位置



Linux系统架构

系统调用 vs API

- 程序员或系统管理员并非直接与系统调用打交道，实际中调用的是API
  - Linux的API遵循POSIX，通常由C库提供，API是一个函数定义，函数说明了如何获得一个给定的服务
  - 系统调用通过中断向内核发出明确的请求，其核心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
- API比系统调用范围更广

系统调用 vs 系统命令

- 系统命令相对API更高一层，每个系统命令都是一个可执行程序，比如ls、hostname等
  - 系统命令的实现利用了系统调用
  - 通过strace 命令可以查看相应系统命令所调用的系统调用
    - strace ls

系统调用 vs 内核函数

- 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，但由内核函数实现
- 内核函数在形式上与普通函数一致，但在内核实现的，需满足一些内核编程的要求
- 每个系统调用有与之对应的内核函数，该内核函数被称为“服务例程”，系统调用称为“封装例程”
  - 封装例程与对应的服务例程由命名规范，如系统调用getpid()的服务例程为sys_getpid()

系统调用 vs 内核函数

- 系统调用与内核函数举例

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
int main(void)
{
        long ID1, ID2;
        // 直接调用内核函数
        ID1 = syscall(SYS_getpid);
        printf ("syscall(SYS_getpid)=%ld\n", ID1);

        //调用系统调用 */
        ID2 = getpid();
        printf ("getpid()=%ld\n", ID2);
        return(0);
}
```

```
root@bogon:/code/syscall6.1# ./syscall_test
syscall(SYS_getpid)=99560
getpid()=99560
```

- 程序运行结果

系统调用号

- Linux中以系统调用号来标识每一个系统调用
- 可在系统中查看相应系统调用号
  - /usr/include/asm/unistb_32.h

```
[root@localhost ~]# cat /usr/include/asm/unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#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
```

系统调用号

系统调用号

- 系统调用号使用非常关键，一旦分配好就不能再有任何改变，否则编译好的应用程序可能会因为调用到错误的系统调用而导致程序崩溃
  - 即使该系统调用删除，其系统调用号也不能被回收利用
- 可使用“man 2 syscalls”浏览所有系统调用的添加历史

系统调用表

- 系统调用表sys_call_table存储了所有系统调用对应的内核函数的函数地址
  - arch\x86\syscalls\syscall_**.tbl

```
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The abi is always "i386" for this file.
#
0   i386    restart_syscall     sys_restart_syscall
1   i386    exit            sys_exit
2   i386    fork            sys_fork            stub32_fork
3   i386    read            sys_read
4   i386    write           sys_write
5   i386    open            sys_open            compat_sys_open
```

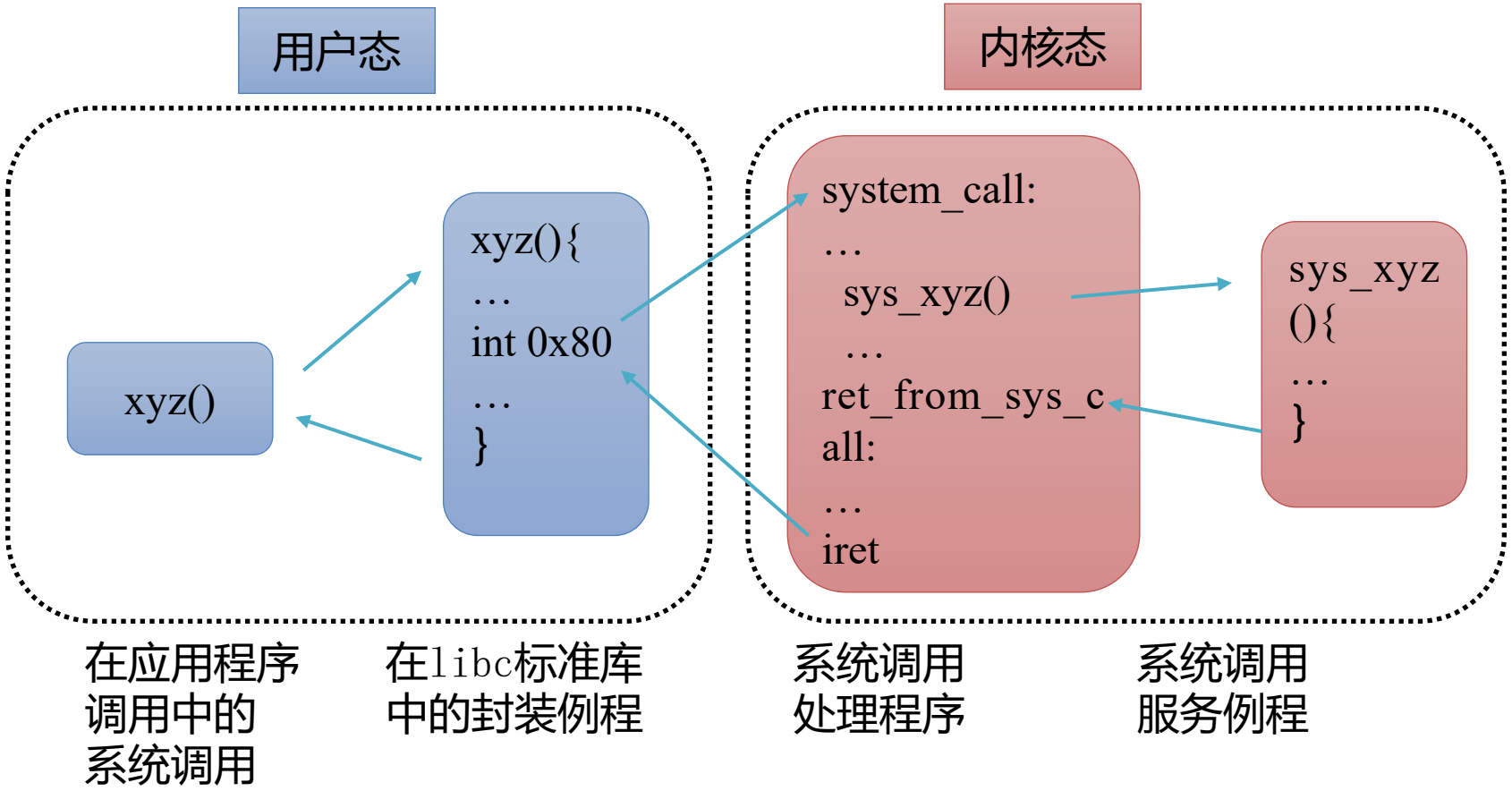
系统调用表

Part 2 系统调用执行

系统调用执行过程

- 当用户态进程调用一个系统调用时，CPU从用户态切换到内核态并开始执行一个内核函数
- Linux对系统调用的调用必须通过执行 INT $0x80 汇编指令产生中断向量为128的中断
- 基于可移植性和安全考虑，用户应用不能直接调用INT $0x80指令， INT $0x80指令被封装在C库中

系统调用执行过程



系统调用执行过程

系统调用初始化

- 内核初始化期间调用trap_init()函数建立IDT表中128号向量对应的表项
  - 系统调用DPL为3，即允许用户态进程调用

```
#arch/x86/kenel/traps.c

set_system_trap_gate(SYSCALL_VECTOR, &system_call);
```

0x80

system_call()函数

- system_call()函数实现系统调用处理程序
  - \arch\x86\kernel\entry.s

```
ENTRY(system_call)
    RING0_INT_FRAME         # can't unwind into user space anyway
    ASM_CLAC
    pushl_cfi %eax          # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
                    # system call tracing in operation / emulation
    testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%ebp)
    jnz syscall_trace_entry
    cmpl $(NR_syscalls), %eax
    jae syscall_badsys
syscall_call:
    call *sys_call_table(,%eax,4)
syscall_after_call:
    movl %eax,PT_EAX(%esp)      # store the return value
syscall_exit:
```

  - system_call()函数执行内容

system_call()函数

- system_call()函数处理
  - 第一步：把系统调用号及异常处理程序需用到的所有CPU寄存器保存到相应栈中

```
pushl_cfi %eax      # save orig_eax
SAVE_ALL
GET_THREAD_INFO(%ebp)
```

将EAX寄存器中的系统调用号压栈，如fork()函数系统调用号为2

保存上下文环境

将当前进程PCB地址存放于ebp中

system_call()函数

- system_call()函数处理
  - 第二步：对用户态传递来的系统调用号进行有效性检查

```
cmpl $(NR_syscalls), %eax
jae syscall_badsys
```

将系统调用号与NR_syscalls比较，若大于NR_syscalls则终止程序，若系统调用号无效则跳转syscall_badsys

system_call()函数

- system_call()函数处理
  - 第三步：根据系统调用号调用对应服务例程

```
call *sys_call_table(, %eax, 4)
```

根据EAX中所包含的系统调用号，找到对应处理程序

system_call()函数

- system_call()函数处理
  - 第四步：获得返回值

```
syscall_after_call:

movl %eax,PT_EAX(%esp)      # store the return value
```

将EAX中的返回值存于栈中

寄存器EAX既负责传递系统调用号，
也用于存放系统调用的返回值

Part 3 添加系统调用

系统调用定义流程

- 系统调用添加步骤
  - 编写系统调用服务例程
  - 定义系统调用原型声明
  - 添加系统调用号
  - 修改系统调用表
  - 重新编译内核

添加自定义系统调用，需先在Linux系统中安装相应版本源代码

系统调用定义流程

- 1.编写系统调用服务例程
  - 进入到自己对应的linux内核源码的kernel目录下
  - 在kernel/sys.c文件中添加相应代码

```
asmlinkage long sys_hello(void)
{
        printk("Hello\n");

        return 0;
}
```

系统调用定义流程

- 2.定义系统调用原型声明
  - 在include/linux/syscalls.h文件中添加相应代码

```
asmlinkage long sys_hello(void);
```

系统调用定义流程

- 3.添加系统调用号
  - 在arch\x86\syscalls\syscall_64.tbl文件中添加相应代码
  - 添加代码前一定要先确认调用号未使用

```
332     common     hello                sys_hello
```

系统调用定义流程

- 4.编译内核

```
make mrproper                  //删除之前编译所生成的文件、配置文件和备份文件等
cp /boot/config-3.10.0-693.2.2.el7.x86_64 .config     //复制当前内核配置文件
make                           //编译内核，时间较长（约3h）
make modules_install     //安装内核模块，约5min
make install       //安装内核

reboot                 //重启并选择相应版本内核内核
```