中国科学技术大学软件学院
SCHOOL OF SOFTWARE ENGINEERING OF USTC

System Call

based on Linux/glibc

孟宁

E-mail：mengning@ustc.edu.cn
主页：http://staff.ustc.edu.cn/~mengning

Agenda

- 系统调用的意义
- API和系统调用
- 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

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

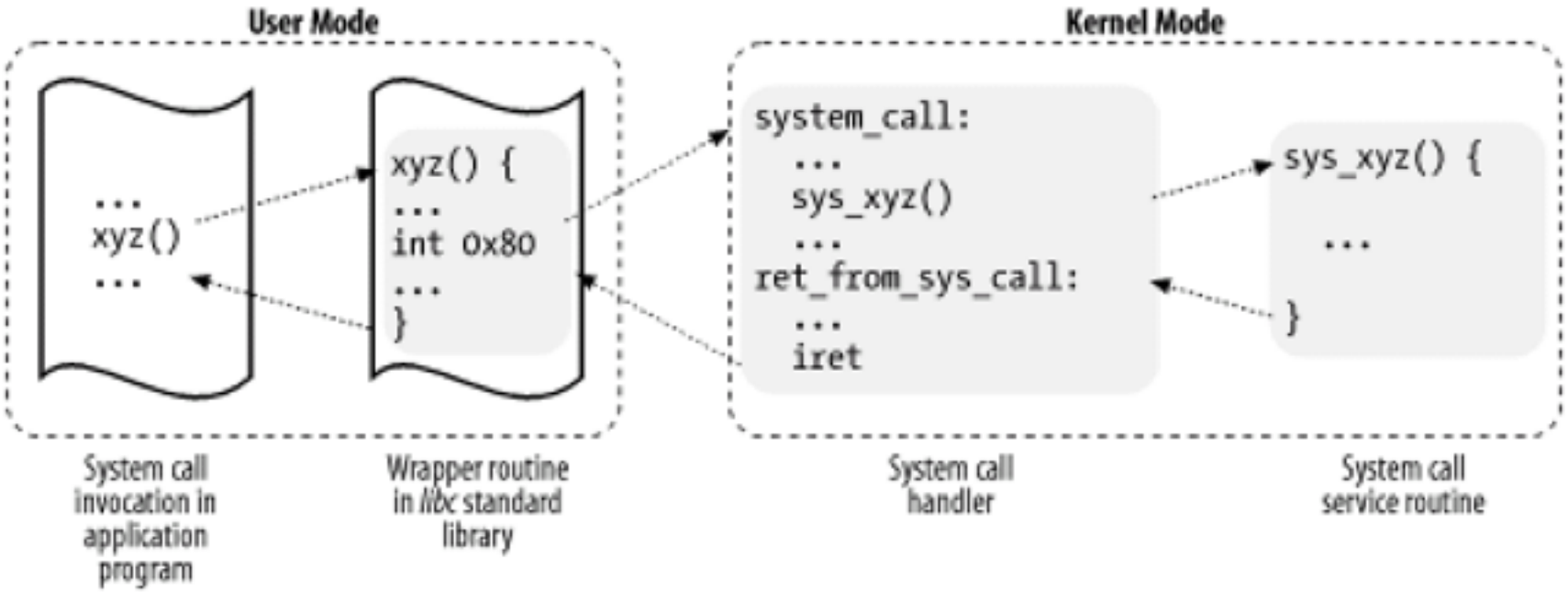
API和系统调用

- 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

API和系统调用

- 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
- 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

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



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一个内核函数。
  - 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
  - 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
  - 使用eax寄存器

参数传递

- 系统调用也需要输入输出参数，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
  - 一个应用程序调用fork()封装例程，那么在执行int $0x80之前就把eax寄存器的值置为2(即__NR_fork)。
  - 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
  - 进入sys_call之后，立即将eax的值压入内核堆栈
  - 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?

```
#include <stdio.h>
#include <time.h>

int  main()
{
        time_t tt;
        struct tm *t;
#if 0
        time(&tt);
        printf("tt:%x\n",tt);
#else
        asm volatile(
        "mov $0,%%ebx\n\t"
        "mov $0xd,%%eax\n\t"
        "int $0x80\n\t"
        "mov %%eax,%0\n\t"
        : "=m" (tt)
        );
        printf("tt:%x\n",tt);
#endif
        t = localtime(&tt);
        printf("time:%d:%d:%d:%d:%d:%d\n",t->tm_year+1900, t->tm_mon, t->tm_mday, t->tm_hour, t->tm_min, t->tm_sec);
        return 0;
}
```

系统调用的内核代码

- 系统调用分派表(dispatch table)存放在sys_call_table数组
  - \arch\x86\kernel\syscall_table_32.S

```
ENTRY(sys_call_table)
.long sys_time /* 13 */
```

- \arch\x86\kernel\entry_32.S

```
•ENTRY(system_call)
•SAVE_ALL
•cmpl $(nr_syscalls), %eax
•call *sys_call_table(,%eax,4)
•movl %eax,PT_EAX(%esp)                    # store the return value
•syscall_exit:
```

系统调用的机制的初始化

- \init\main.c start_kernel

```
trap_init();
```

- \arch\x86\kernel\traps.c

```
#ifdef CONFIG_X86_32
        set_system_trap_gate(SYSCALL_VECTOR, &system_call);
        set_bit(SYSCALL_VECTOR, used_vectors);
#endif
```

sys_time

- 以下代码来自linux-3.2.1\kernel\time.c

```
SYSCALL_DEFINE1(time, time_t __user *, tloc)
{
        time_t i = get_seconds();

        if (tloc) {
                if (put_user(i,tloc))
                        return -EFAULT;
        }
        force_successful_syscall_return();
        return i;
}
```

为政抓不着要处，忙也无益。

抓着要处，忙更有益。 -- 阎锡山

谢谢大家！

参考资料：
《深入理解Linux内核》第三版
http://www.gnu.org/software/libc/