6.2 系统调用机制

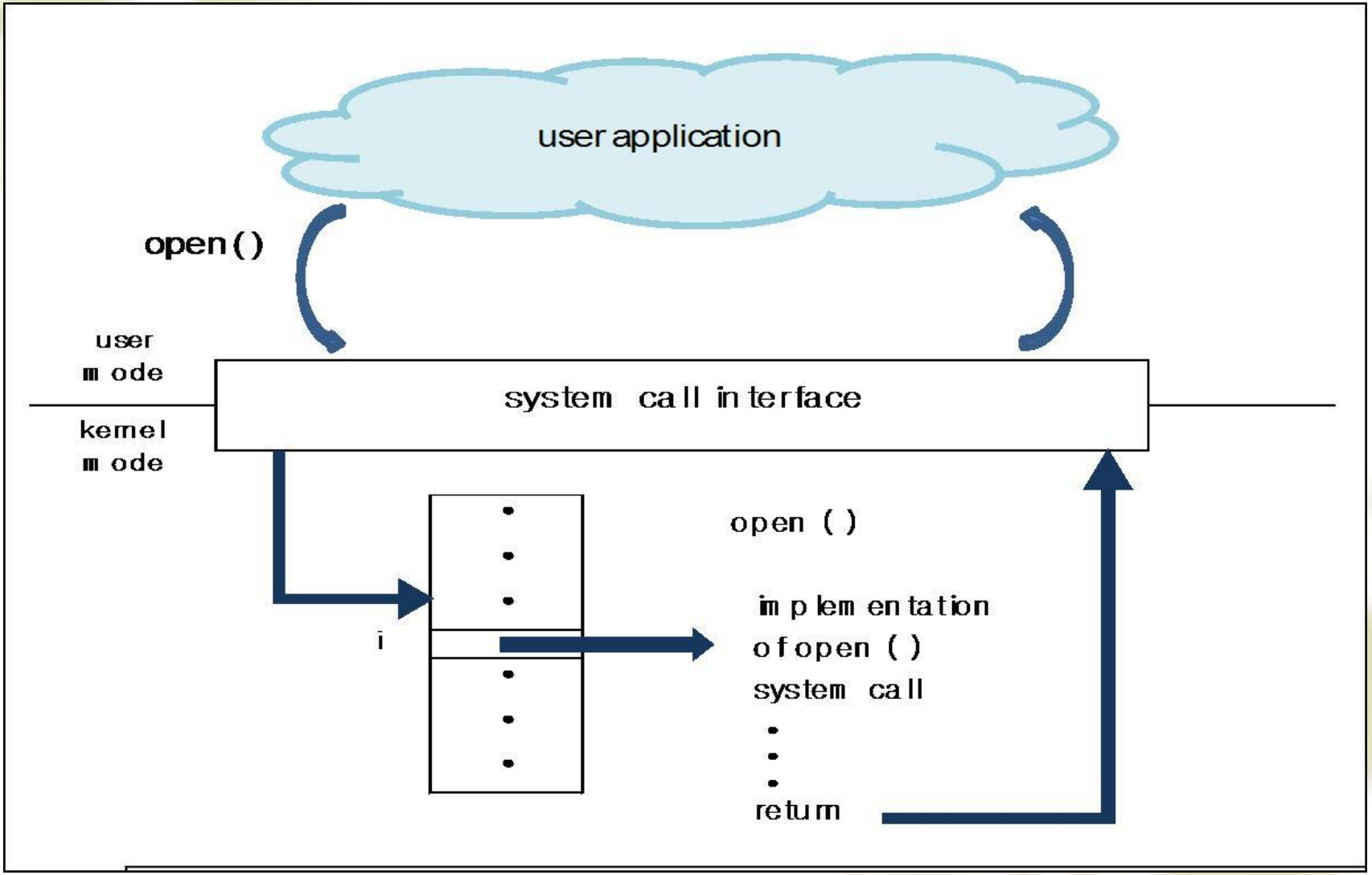
西安邮电大学

系统调用—内核的出口

系统调用，顾名思义，说的是操作系统提供给用户程序调用的一组“特殊”接口。

从逻辑上来说，系统调用可被看成是一个内核与用户空间程序交互的接口——它好比一个中间人，把用户进程的请求传达给内核，待内核把请求处理完毕后再将处理结果送回给用户空间。

系统调用—内核的出口


user application
open()
user mode
kernel mode
system call interface
i
open ( )
implementation of open ( ) system call
return

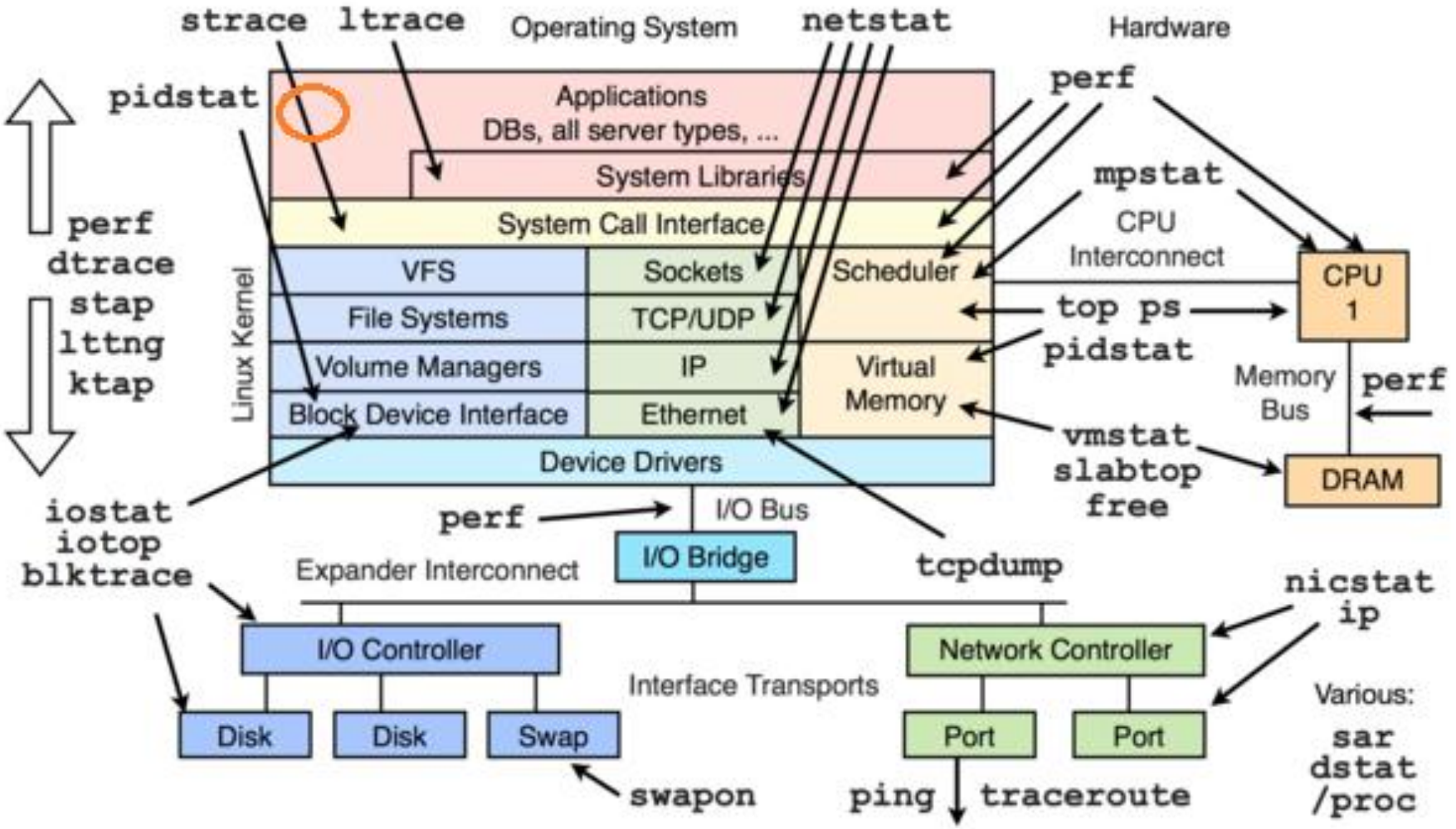
跟踪进程所调用的系统调用

如图为Linux系统中各个子系统相关的工具。

可以通过strace 命令查看一个应用所调用的系统调用

比如 $strace ls

跟踪进程所调用的系统调用



中断、异常和系统调用比较

中断、异常和系统调用本质是属于一类，处理方式上也类似，那么它们之间的差异性表现在哪些方面，在学习了中断后，对学系统调用有哪些帮助？

源头不同

中断:是外设发出的请求

异常:是应用程序意想不到的行为

系统调用：应用程序请求OS提供

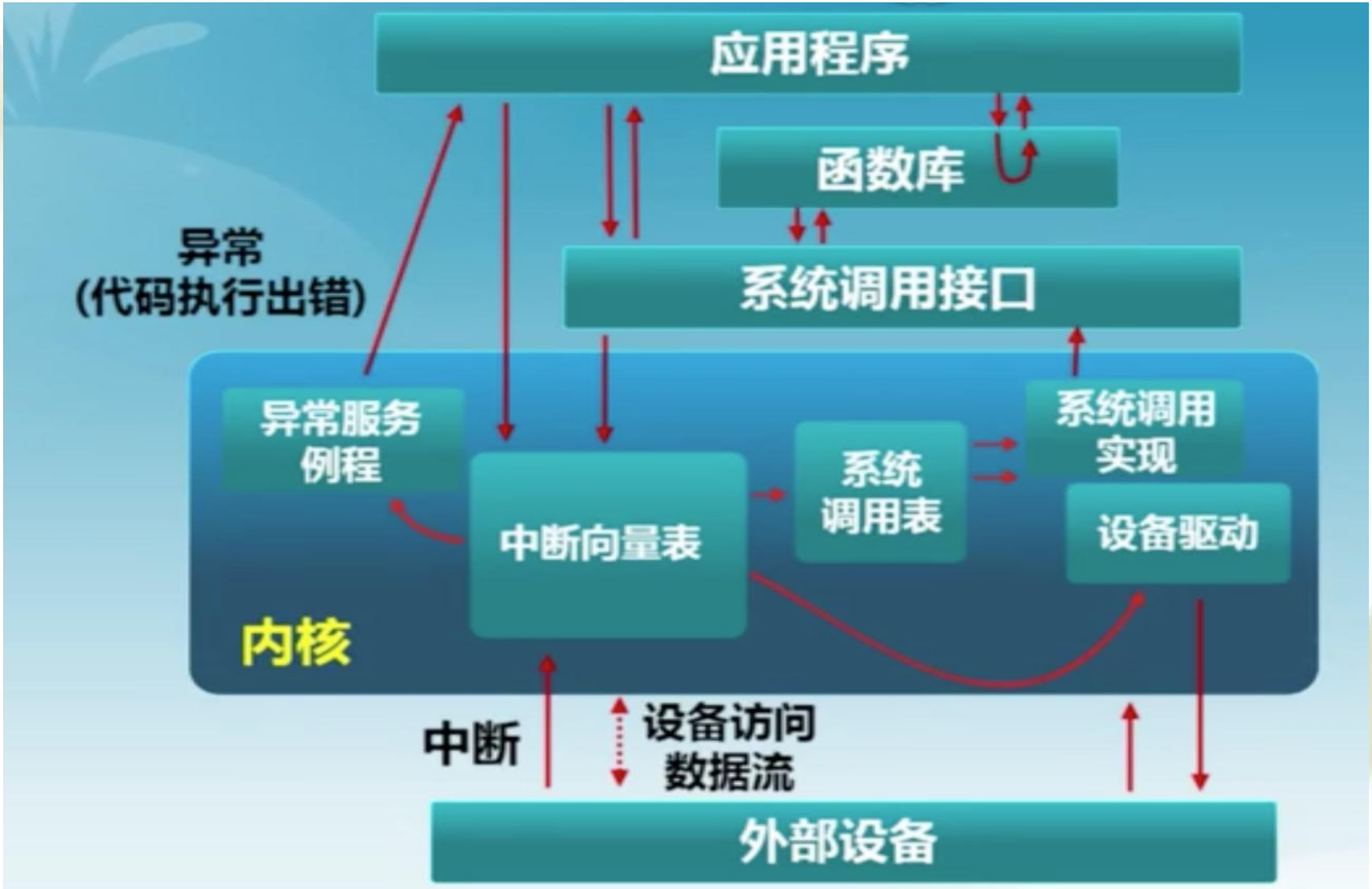
服务响应方式不同

中断是异步的，异常是同步的，而系统调用既可以是异步，也可以是同步。

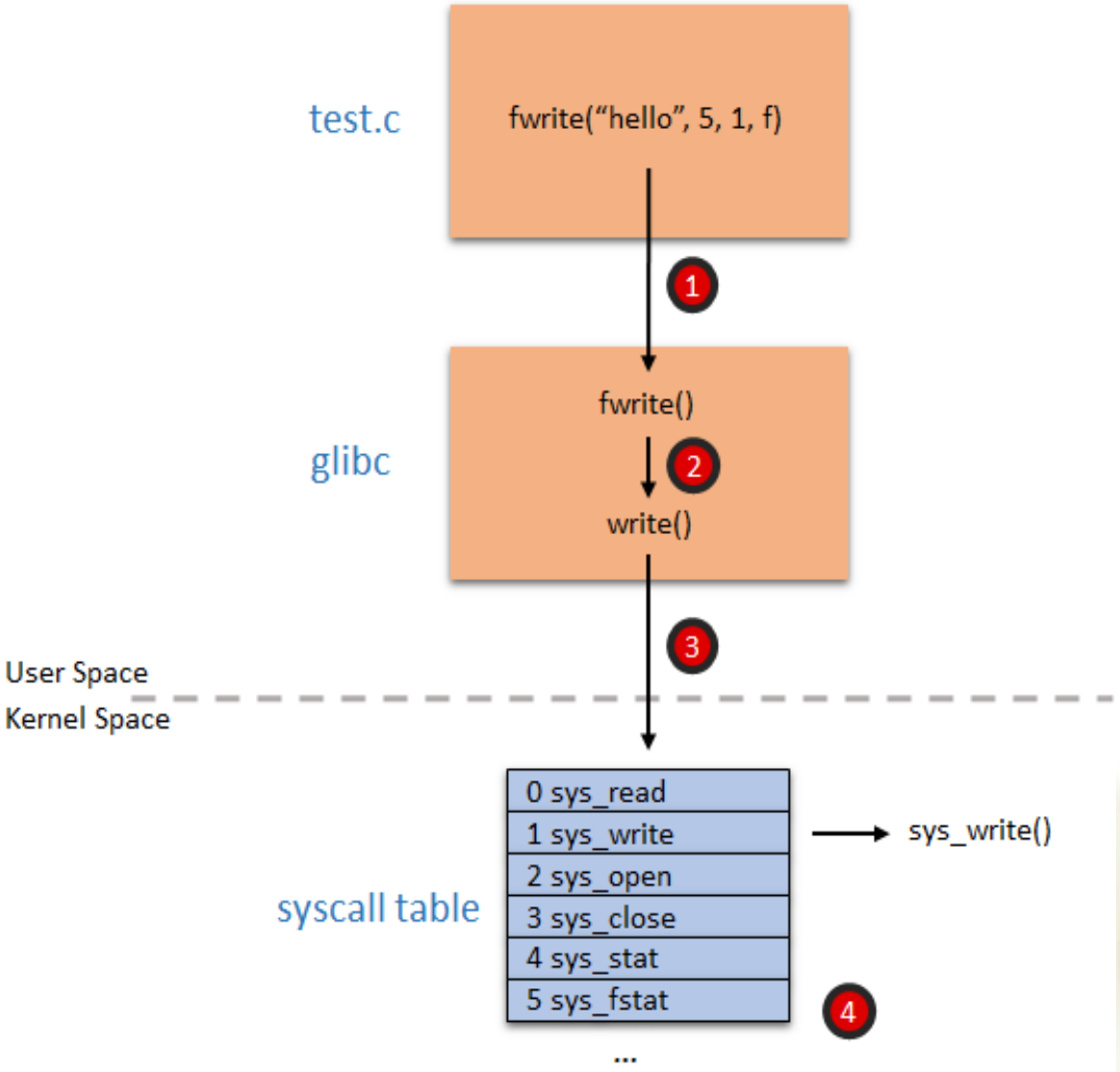
处理机制不同

中断服务程序在内核态下运行，对用户是透明的。异常出现时，或者杀死进程，或者重新执行引起异常的指令。系统调用，用户发出请求后等待OS的服务。

中断、异常和系统调用比较

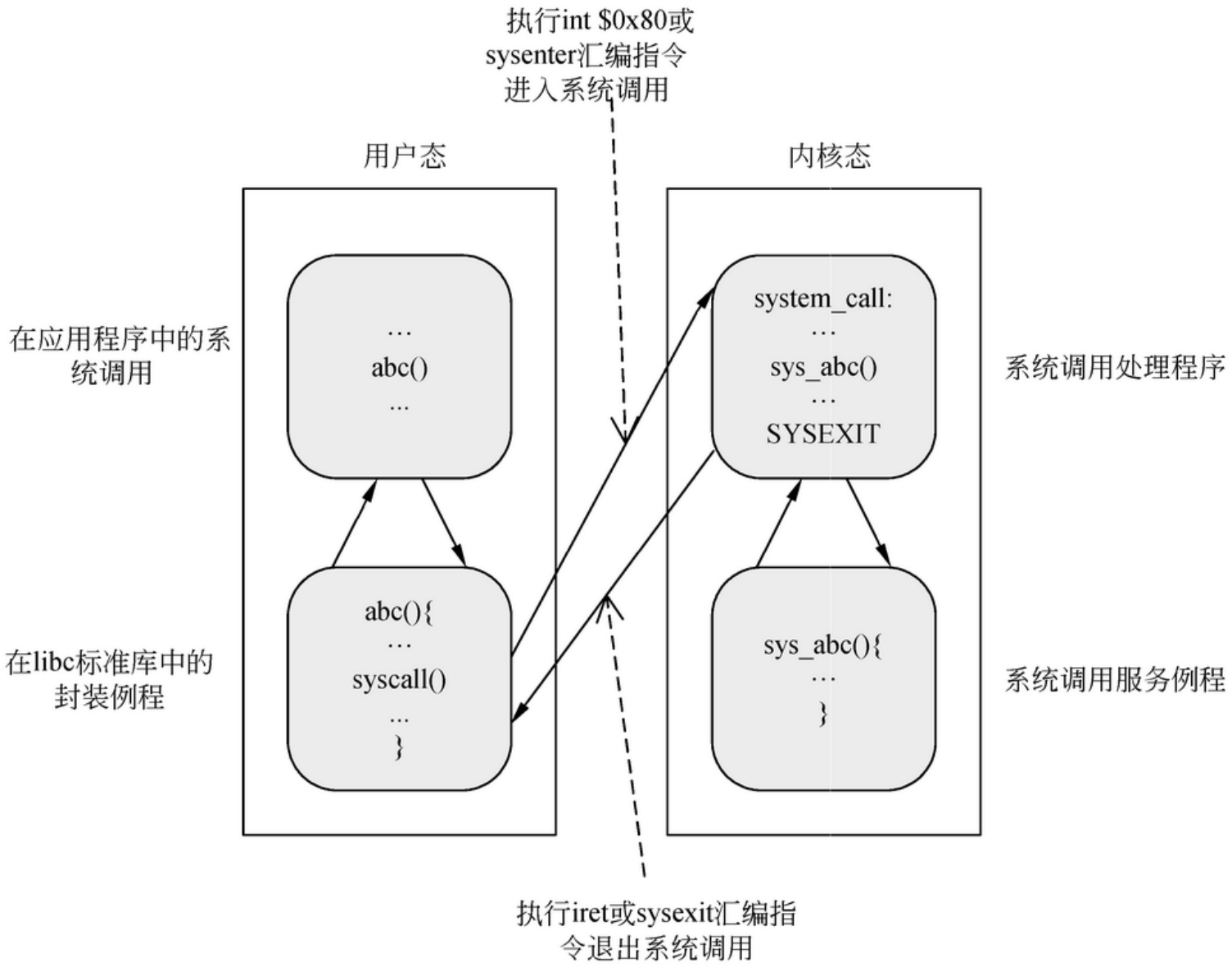


从用户态函数到系统调用



比如在程序中调用fwrite函数，而fwrite函数在glibc库中调用系统调用write()，然后从用户态陷入内核态，查找系统调用表，对应的系统调用服务例程为sys_write

系统调用一般处理流程


执行int $0x80或
sysenter汇编指令
进入系统调用
用户态
内核态
在应用程序中的系统调用
…
abc()
…
system_call:
…
sys_abc()
…
SYSEXIT
系统调用处理程序
在libc标准库中的封装例程
abc(){
…
syscall()
…
}
sys_abc(){
…
}
系统调用服务例程
执行iret或sysexit汇编指令退出系统调用

系统调用一般处理流程

当用户态的进程调用一个系统调用时，在libc的封装例程中会调用int 0x80或者syscall汇编指令切换到内核态，并开始执行一个内核system_call系统调用处理程序

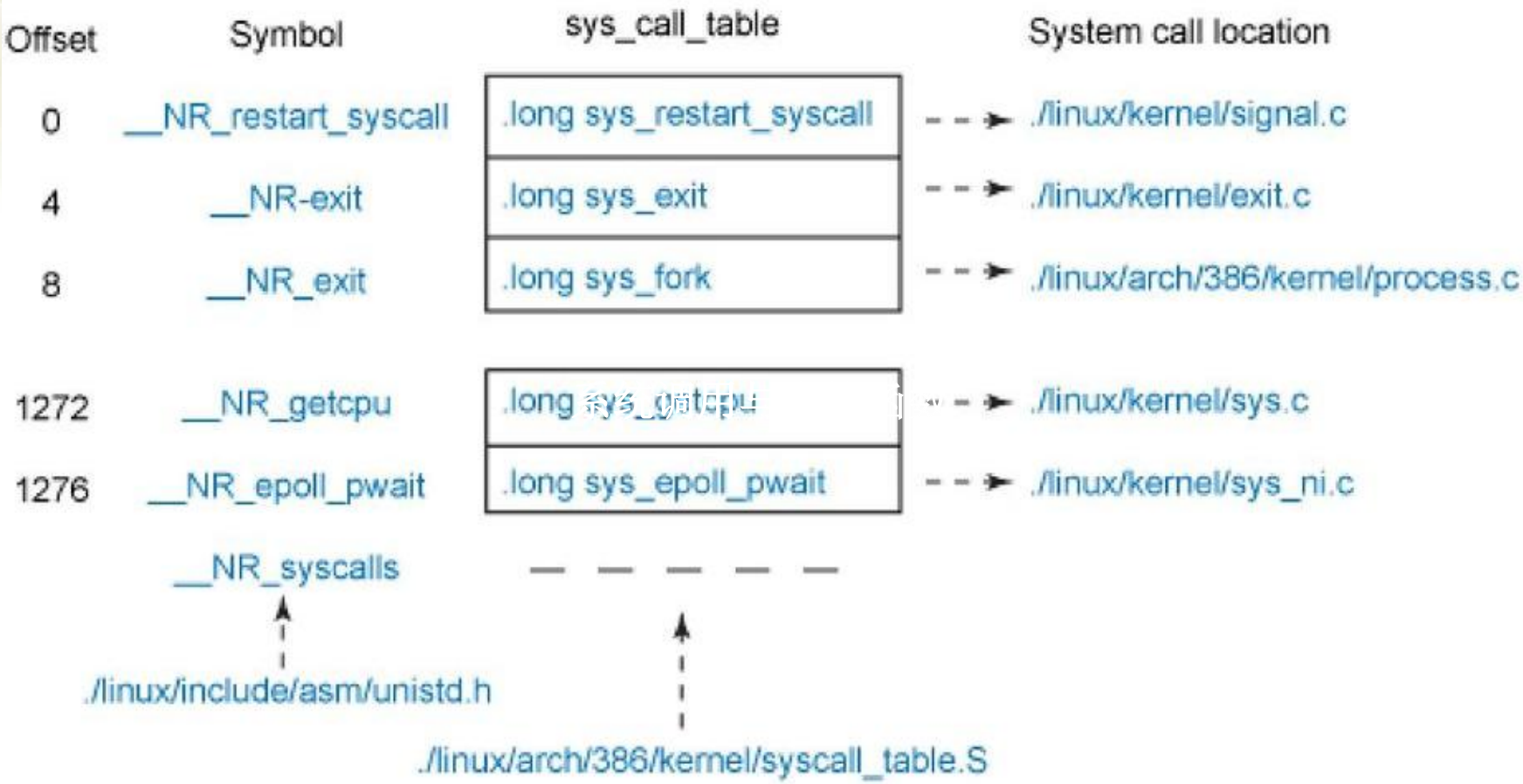
系统调用处理程序执行下列操作：

在内核栈保存大多数寄存器的内容

调用系统调用服务例程来处理系统调用

通过iret或者sysexit汇编指令从系统调用返回

系统调用基本概念


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
./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

系统调用基本概念

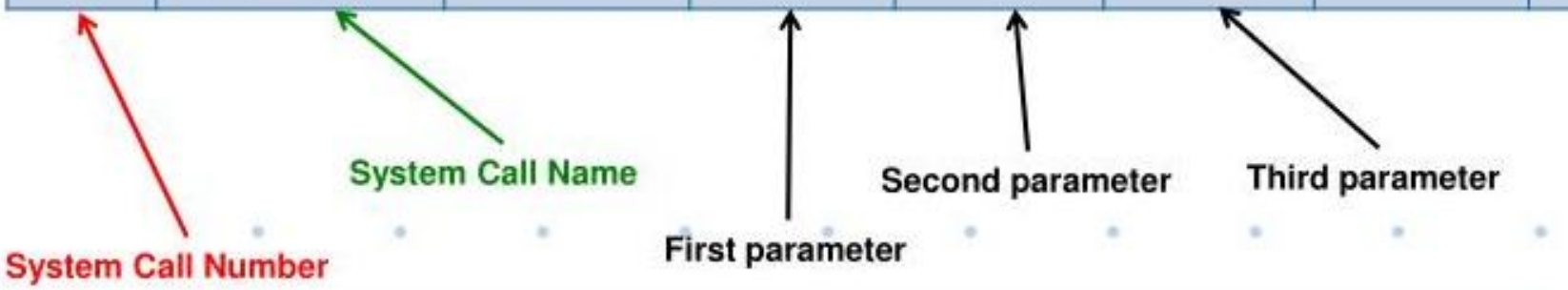
系统调用号：
①用来唯一的标识每个系统调用；
②作为系统调用表的下标，当用户空间的进程 执行一个系统调用时，该系统调用号就被用来指明到底要执行哪个系统调用服务例程。
系统调用表：
是用来把系统调用号和相应的服务例程关联起来。该表存放在sys_call_table数组中：
说明：内核版本不同，这些系统调用号和系统调用所在的头文件有所差别

部分系统调用列表

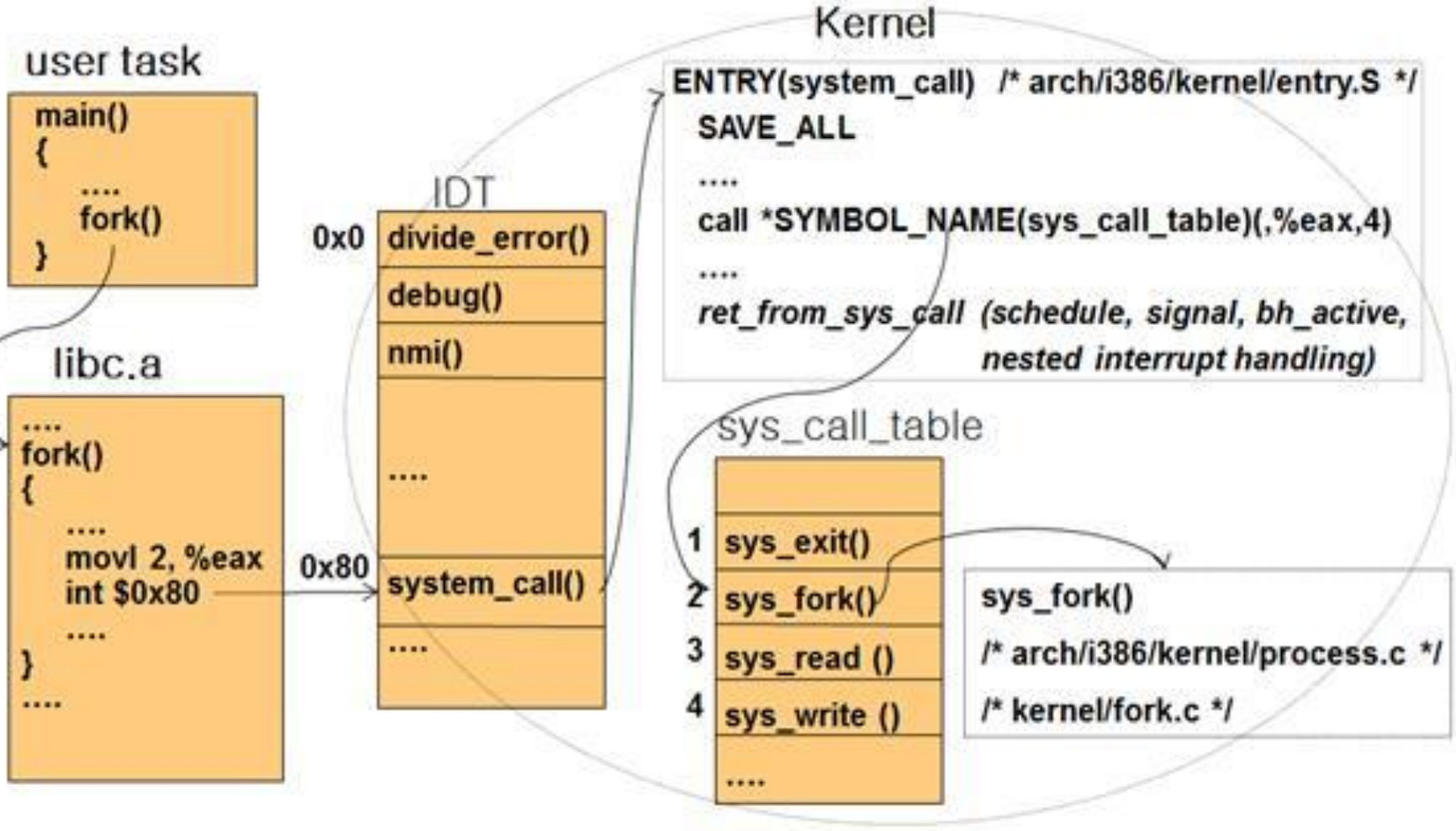
从表看出，系统调用号存放在eax寄存器中。每个系统调用在内核中对应的服务例程以sys打头，它们的实现所在的源文件也各不相同，系统调用的参数存放在寄存器中，一般参数不超过6个（包括系统调用号）。

部分系统调用列表

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int				
3	sys_read	fs/read_write.c	unsigned int	char	size_t		
4	sys_write	fs/read_write.c	unsigned int	const char	size_t		
15	sys_chmod	fs/open.c	const char	mode_t			
20	sys_getpid	kernel/timer.c	void				
21	sys_mount	fs/namespace.c	char __user *	char __user *	char __user *	unsigned long	void __user *
88	sys_reboot	kernel/sys.c	int	int	unsigned int	void __user *	



从用户态跟踪一个系统调用到内核



从用户态跟踪一个系统调用到内核

1. 从用户程序中调用fork
2. 在libc库中把fork对应的系统调用号2放入寄存器eax
3. 通过int 0x80陷入内核
4. 在中断描述表IDT中查到系统调用的入口0x80
5. 进入Linux内核的entry_32(64).S文件，从系统调用表sys_call_table中找到sys_fork的入口地址
6. 执行fork.c中的do_fork代码
7 通过iret或者sysiret返回

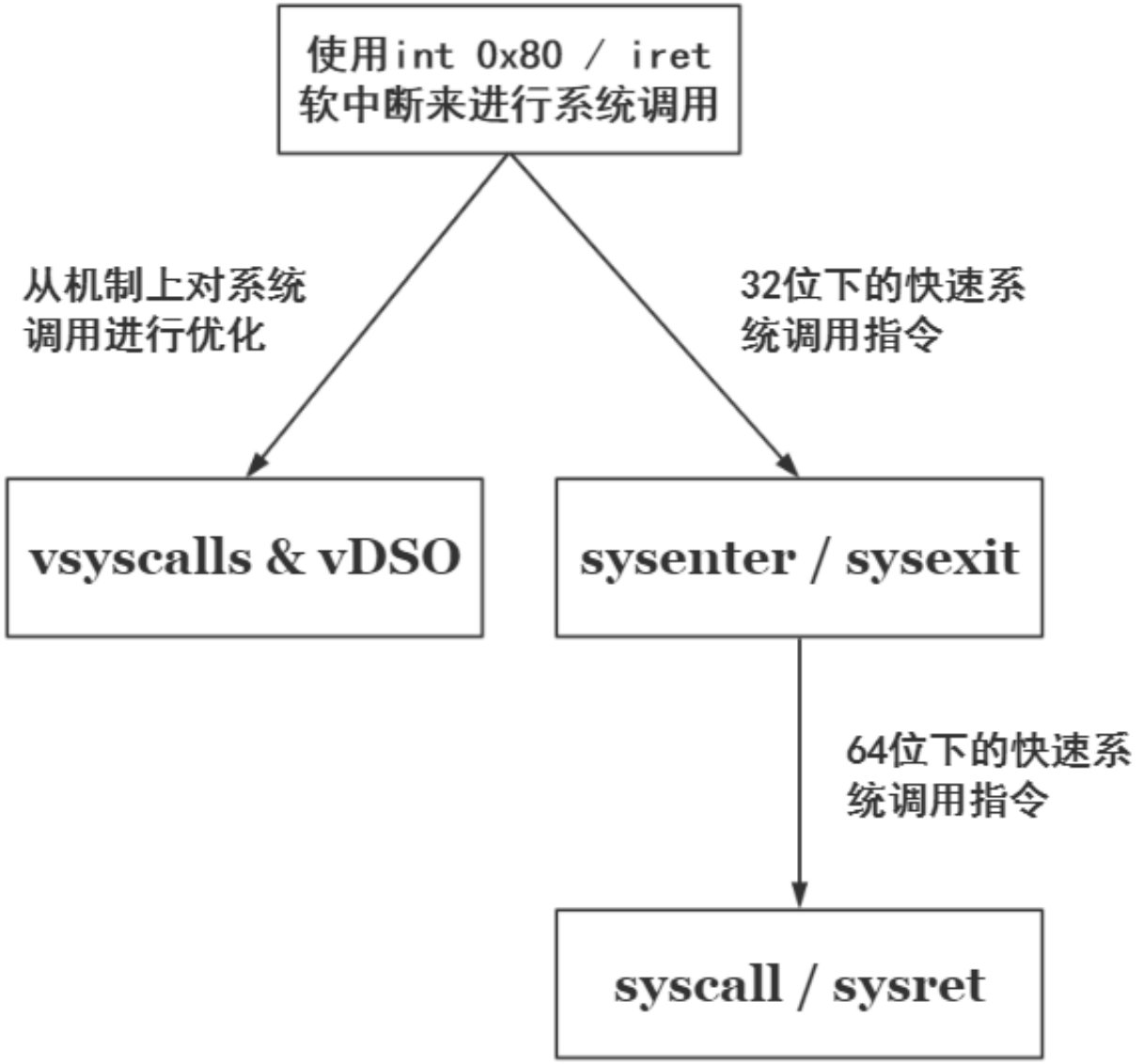
系统调用机制的优化

在2.6的早前版本中，系统调用的实现用的是int 0x80和iret命令。

因为系统调用的实现从用户态切换到内核态，执行完系统调用程序后又从内核态切换回用户态,代价很大，为了加快系统调用的速度，随后先后引入了两种机制——vsycalls和vDSO 。

这两种机制都是从机制上对系统调用速度进行的优化，但是使用软中断来进行系统调用需要进行特权级的切换这一根本问题没有解决。为了解决这一问题，Intel x86 CPU从Pentium II之后，开始支持快速系统调用指令sysenter/sysexit，这两条指令是Intel在32位下提出的，而AMD提出syscall/sysret，64位统一使用这两条指令了。

系统调用机制的优化

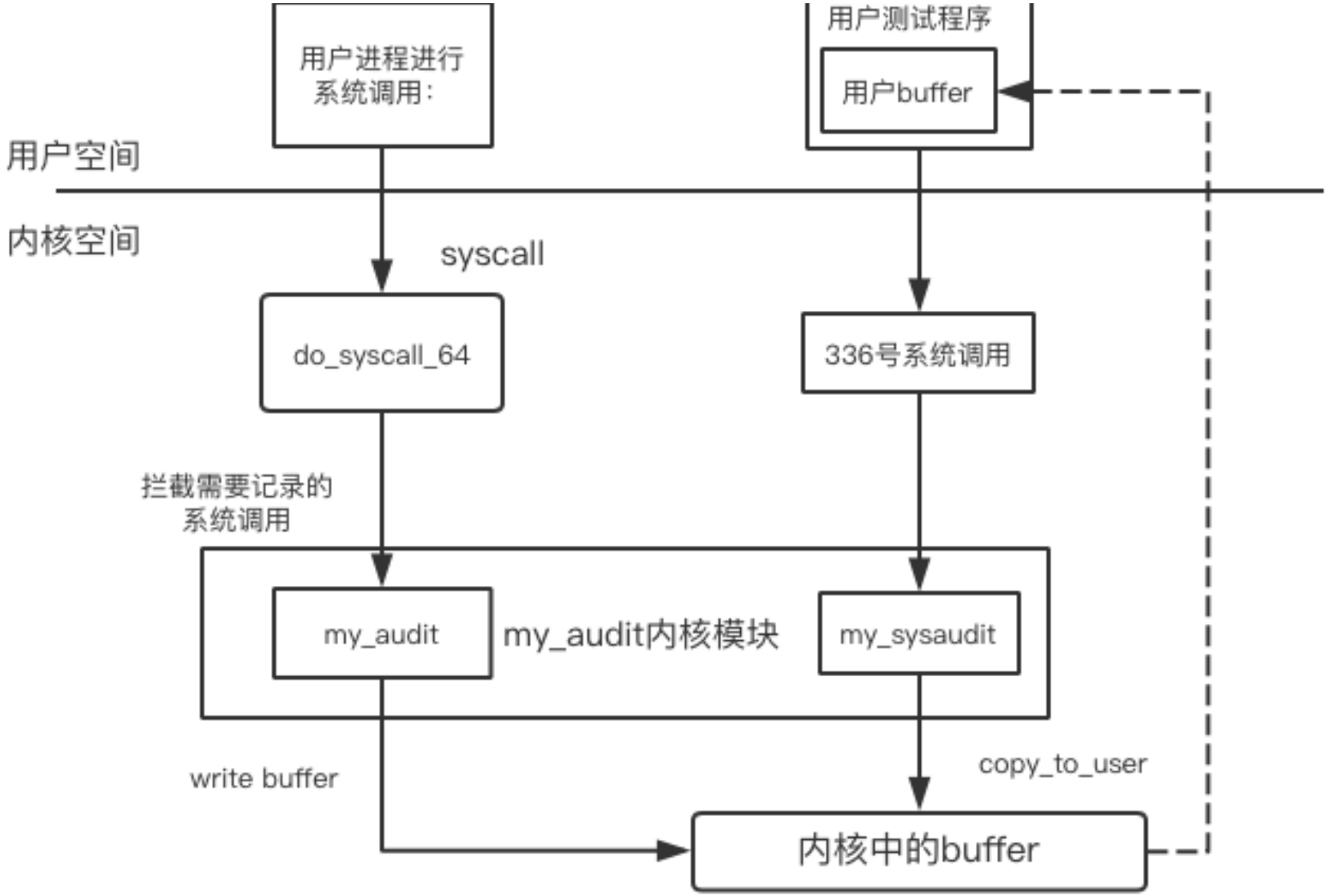

使用int 0x80 / iret
软中断来进行系统调用
从机制上对系统
调用进行优化
32位下的快速系
统调用指令
vsyscalls & vDSO
sysenter / sysexit
64位下的快速系
统调用指令
syscall / sysret

系统调用实例-日志收集系统

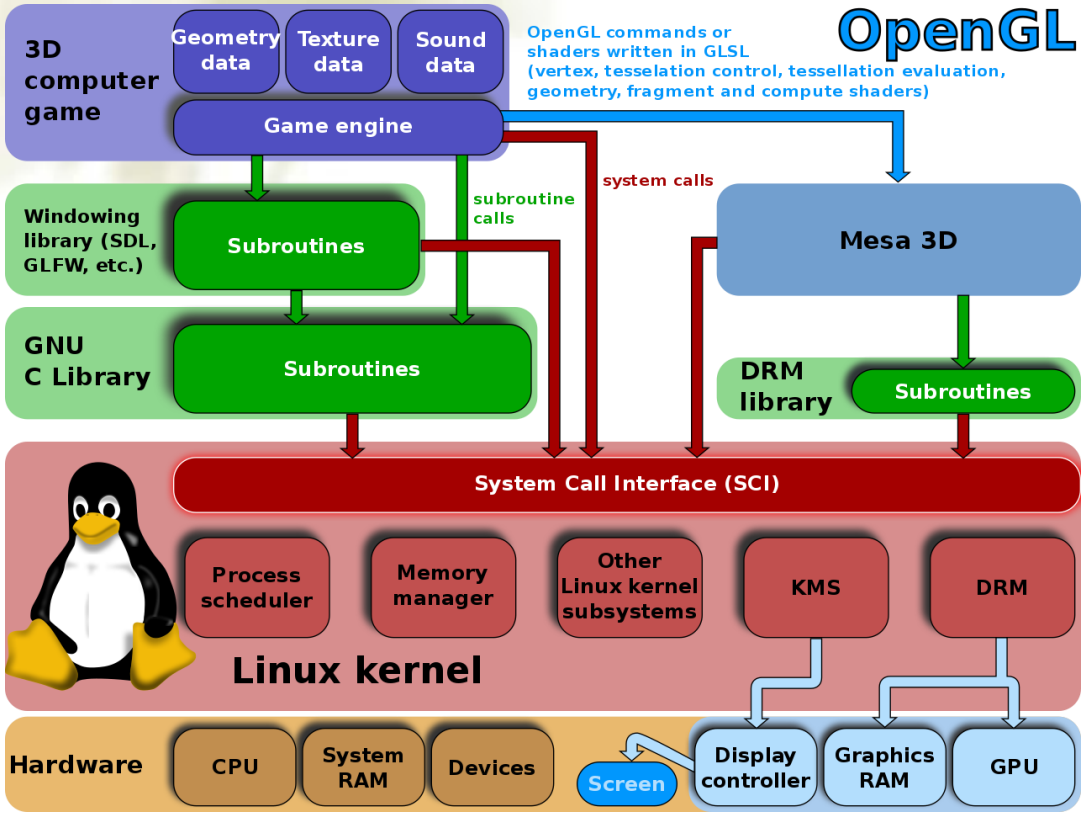
系统调用是用户程序与系统打交道的入口，系统调用的安全直接关系到系统的安全，如果一个用户恶意地不断调用fork()将导致系统负载增加，所以如果能收集到是谁调用了一些有危险的系统调用，以及系统调用的时间和其他信息，将有助于系统管理员进行事后追踪，从而提高系统的安全性。

本实例的实现过程将在下一讲给予具体演示，并说明如何添加一个系统调用。

系统调用实例-日志收集系统



本章小结



系统调用是应用与内核之间的一个接口

Linux内核中系统调用的具体实现与CPU体系结构相关。

是否要添加一个新的系统调用需要认真评估。

动手实战

在Linux内核之旅网站：http://www.kerneltravel.net/

电子杂志栏目第四期“系统调用”，本期重点讨论系统调用机制。其中涉及到了一些及系统调用的性能、上下文切换的深层问题，同时也穿插着讲述了一些内核调试方法。

动手实践，在较早的2.6.x和最新的5.x内核版本下调试系统调用日志收集系统，并给出分析结果。

带着思考离开

系统调用的实现机制进行了多次优化，为什么要进行这样的优化，未来还有优化的空间么？

谢谢大家！

THANK YOU