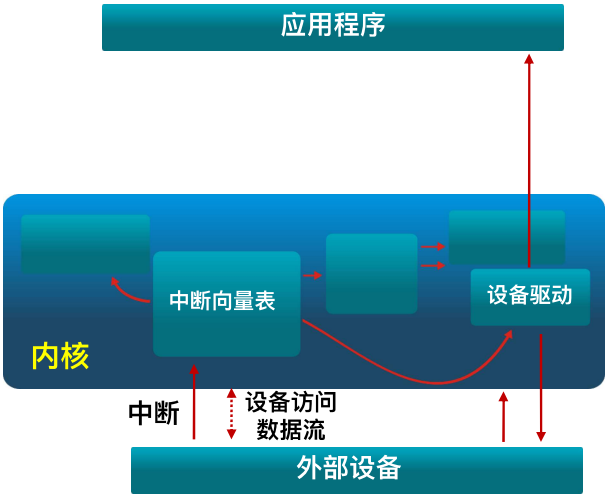
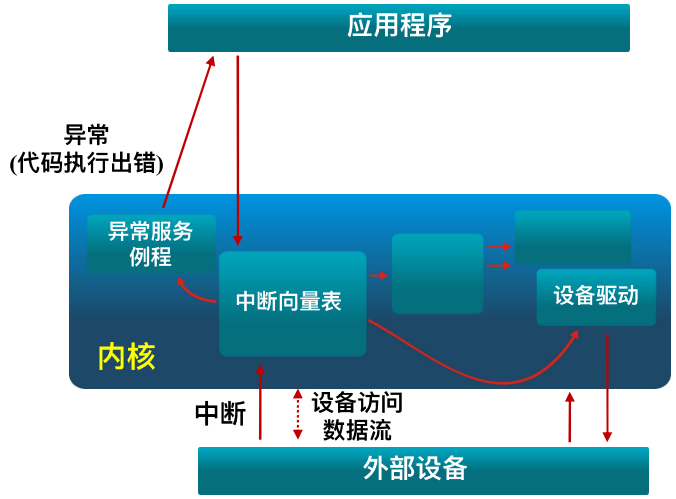
用户指令与内核如何交互

- 参考中断的机制，OS中设计了三种可以在运行时“呼叫”操作系统，并提升权限的方式
  - 中断（Interrupt）（在刚才所述场景中需要使用的是中断）
  - 异常（Exception）
  - 系统调用（System Call）
- 这些机制能够提升当前CPU的权限（改变当前的CPU状态到ring0），同时，跳转PC到指定的函数入口处（提升权限后只能执行OS预定的函数，例如中断向量表中的函数）

中断过程的运行机制



异常过程的运行机制



中断和异常是由硬件实现的一种跳转机制，在某些特定事件发生时如设备按下（中断），或除数为0导致指令无法执行（异常），需要打断现有的指令流，改变当前权限状态，并调用预设的处理函数（即PC设置到该函数的入口地址）

中断、异常和系统调用

- 异常(exception)
  - 非法指令或者其他原因导致当前指令执行失败(如：内存出错)后的处理请求
- 中断(hardware interrupt)
  - 来自硬件设备的处理请求
- 系统调用（system call）
  - 应用程序需要执行某个高权限操作时，主动向操作系统发出的服务请求

系统调用有多少个？应用需要OS为它做多少事？

$2	系统调用名	源代码	$4	$5	$6	$7	通过堆栈
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
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	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-

sys_number

https://www.cnblogs.com/ggjucheng/archive/2012/01/08/2316695.html

南開大學 Nankai University

系统调用：用一个中断（异常）解决所有的问题

```
sfs_filetest1.c: ret=read(fd,data,len);
 ……
  8029a1:      8b 45 10              mov   0x10(%ebp),%eax
  8029a4:      89 44 24 08           mov   %eax,0x8(%esp)
  8029a8:      8b 45 0c              mov   0xc(%ebp),%eax
  8029ab:      89 44 24 04           mov   %eax,0x4(%esp)
  8029af:      8b 45 08              mov   0x8(%ebp),%eax
  8029b2:      89 04 24              mov   %eax,(%esp)    ; 以上代码在填充栈
  8029b5:      e8 33 d8 ff ff        call  8001ed <read> ； 这里的read是宏==6
syscall(int num, ...) {
...
        asm volatile (
                    "int %1;"//这一句会个产生一个中断
                    : "=a" (ret)
                    : "i" (T_SYSCALL), //这句指定中断号
                      "a" (num),     //这一句的作用是把6放进了eax
                      "d" (a[0]),
                      "c" (a[1]),
                      "b" (a[2]),
                      "D" (a[3]),
                      "S" (a[4])
                    : "cc", "memory");
        return ret;
```

mov ……..
mov %eax 06H
int 80H

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
#define SYS_mknod  11
#define SYS_unlink 12
#define SYS_fstat  13
#define SYS_link   14
#define SYS_mkdir  15
#define SYS_chdir  16
#define SYS_dup    17
#define SYS_getpid 18
#define SYS_sbrk   19
#define SYS_sleep  20
#define SYS_procmem 21
```

X86版本

南開大學 Nankai University

系统调用的“中断响应函数”read(fd, buffer, length)的实现

从别的地方读length的长度的data放到buffer处（需提前准备）

1. kern/trap/trapentry.S: alltraps()
2. kern/trap/trap.c: trap()
   tf->trapno == T_SYSCALL
3. kern/syscall/syscall.c: syscall()
   tf->tf_regs.reg_eax ==SYS_read
4. kern/syscall/syscall.c: sys_read()
   从 tf->sp 获取 fd, buf, length
5. kern/fs/sysfile.c: sysfile_read()
   读取文件
6. kern/trap/trapentry.S: trapret()

X86版本

系统调用：用一个中断（异常）解决所有的问题

```
int
sys_read(int64_t fd, void *base, size_t len) {
    return syscall(SYS_read, fd, base, len);
}
```

```
syscall(int num, ...) {
...
        asm volatile (
                    "lw a0, %1\n"
      "lw a1, %2\n“
      "lw a2, %3\n"
      "lw a3, %4\n"
      "lw a4, %5\n"
      "lw a5, %6\n"
      "ecall\n“//这一句会个产生一个”中断”
      "sw a0, %0"
      : "=m" (ret)
      : "m" (num), //这一句的作用是把102放进了a0
        "m" (a[0]),
        "m" (a[1]),
        "m" (a[2]),
        "m" (a[3]),
        "m" (a[4])
      : "memory"
    );   return ret;
```

```
/* syscall number */
#define SYS_exit            1
#define SYS_fork            2
#define SYS_wait            3
#define SYS_exec            4
#define SYS_clone           5
#define SYS_yield           10
#define SYS_sleep           11
#define SYS_kill            12
#define SYS_gettime         17
#define SYS_getpid          18
#define SYS_mmap            20
#define SYS_munmap          21
#define SYS_shmem           22
#define SYS_putc            30
#define SYS_pgdir           31
#define SYS_open            100
#define SYS_close           101
#define SYS_read            102
#define SYS_write           103
#define SYS_seek            104
```

系统调用事件有一个固定中断号，同时借助一个寄存器，传递了系统调用中具体哪个调用（系统调用号）

RISCV版本

南開大學 Nankai University

系统调用的“中断响应函数”read(fd, buffer, length)的实现

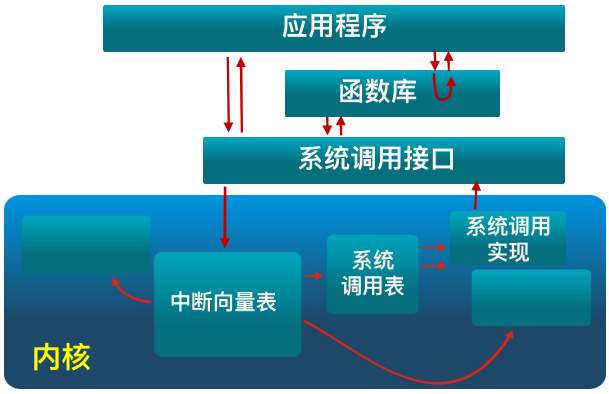
1. kern/trap/trapentry.S: __alltraps()
2. kern/trap/trap.c: trap()→ trap_dispatch
   tf->cause == CAUSE_USER_ECALL
3. kern/syscall/syscall.c: syscall()
   tf->tf_regs.a0 ==SYS_read
4. kern/syscall/syscall.c: sys_read()
   从 tf->sp 获取 fd, buf, length
5. kern/fs/sysfile.c: sysfile_read()
   读取文件
6. kern/trap/trapentry.S: trapret()

RISCV
版本

南開大學 Nankai University

系统调用的实现

- 每个系统调用对应一个系统调用号 sys_number
  - 系统调用接口根据系统调用号来维护表的索引
- 系统调用接口调用内核态中的系统调用功能实现，并返回系统调用的状态和结果
- 用户不需要知道系统调用的实现
  - 需要设置调用参数和获取返回结果
  - 操作系统接口的细节大部分都隐藏在应用编程接口后
    - 通过运行程序支持的库来管理



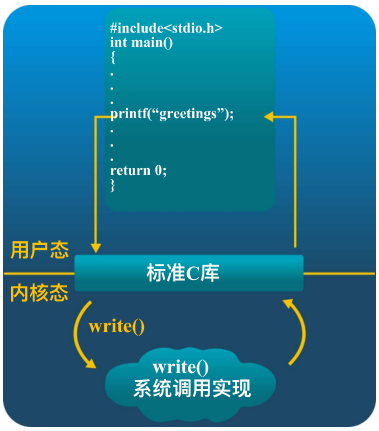
系统调用

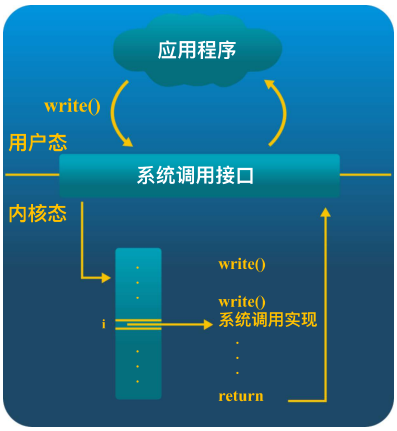
- 操作系统服务的编程接口，用户进程主动“呼叫”OS的编程方式
- 通常由高级语言编写（C或者C++）
- 程序访问通常是通过高层次的API接口而不是直接进行系统调用
- 三种最常用的应用程序编程接口（API）
  - Win32 API 用于 Windows
  - POSIX API 用于 POSIX-based systems (包括UNIX，LINUX，Mac OS X的所有版本)
  - Java API 用于JAVA虚拟机(JVM)

POSIX实现了标准化，即所有的OS与应用程序约定的调用号都一样（理论上），在所有的系统上都可以运行
https://syscalls.mebeim.net/

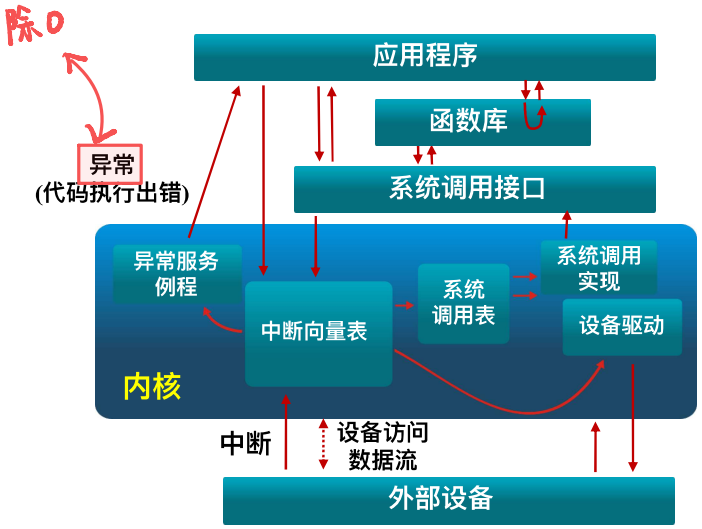
标准C库的例子

- 应用程序调用printf() 时，会触发系统调用write()。





内核的进入与退出


除0
异常
(代码执行出错)
应用程序
函数库
系统调用接口
异常服务例程
中断向量表
系统调用表
系统调用实现
设备驱动
内核
中断
设备访问
数据流
外部设备

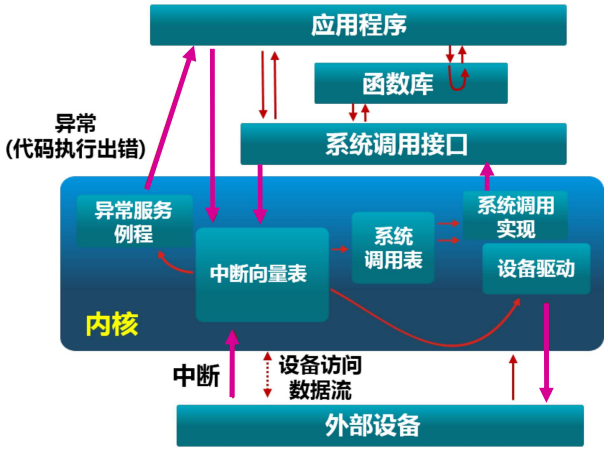
南開大學 Nankai University

中断、异常和系统调用比较

- 源头
  - 中断：外设
  - 异常：应用程序意想不到的行为
  - 系统调用：应用程序请求操作提供服务
- 响应方式
  - 中断：异步
  - 异常：同步
  - 系统调用：异步或同步
- 处理机制
  - 中断：持续，对用户应用程序是透明的
  - 异常：杀死或者重新执行意想不到的应用程序指令
  - 系统调用：等待和持续

等到某条指令执行完才触发

会杀死某条进行中的指令



南開大學 Nankai University

前情提要

- 段机制是页机制之外的一种内存管理的手段，可以独立使用，也可以与页相结合，但缺少足够的硬件支持
- 进程生活在虚拟地址空间中，CPU上发出的地址都是虚拟的，需要经MMU查询转化后得到物理地址
- 每个进程有自己的页表，以管理独立的地址空间
- 在创建新的进程时，Linux中使用fork函数实现进程空间的（半）浅拷贝，然后利用写时复制(copy-on-write, COW)的方法节省新进程的创建时间。新的进程一般会调用exec，按照可执行文件头部的信息重建自己的虚拟地址空间布局图，并借助page fault实现数据加载和完善自己的页表
- 进程数量增加可以提升CPU的利用率，但是过多的进程，会导致page fault频发，反而利用率下降，这种现象称为抖动。消除抖动的关键是控制同时处于内存中的进程数量
- 进程无序竞争以争抢页面的方式并不合理，因此智能手机系统中加入了对页面属性的追踪，以提升换页的精准度
- 多个进程在同一台机器上共生时，数据安全成为一个关键问题，通常需要得到硬件的帮助才能保证软件的保护措施不被轻易绕过
- 硬件安全机制包括分段模式的时候对段长度的限制，以分隔应用程序；以及分页机制时页表的隔离，以及对页面权限的标记区分OS区域和用户区域
- 对应于页面的权限标记，CPU中也有相应的权限标记，对应于当前处于OS状态还是用户状态。仅有CPU权限标记与页面权限能够满足要求时，才能放行相庆的访问请求
- 中断的响应需要提升CPU的运行状态权限，以完成对硬件的处理，因此硬件设计了类似门机制以提升权限，并运行指定的代码。
- 运行于用户态的程序有主动提升权限的需求，因此借鉴类似中断的模式，设计了系统调用，用以调用内核提供的函数，并实现用户开发与内核开发的隔离