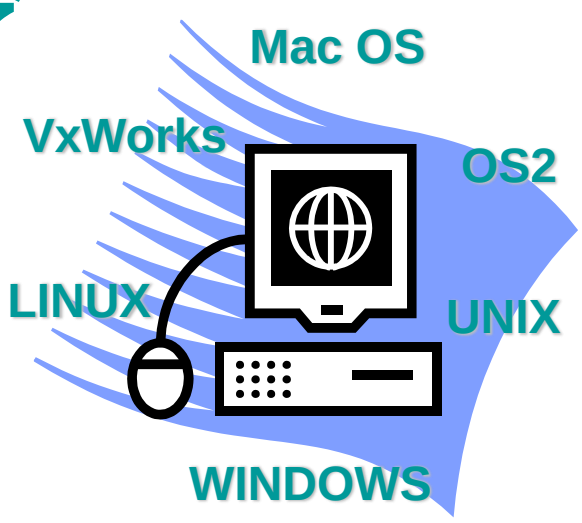
1.3 操作系统基本服务和用户接口

1.3.1 基本服务和用户接口

1.3.2 程序接口与系统调用

1.3.3 作业接口与操作命令



1.3.1 操作系统提供的基本服务

1. 基本服务

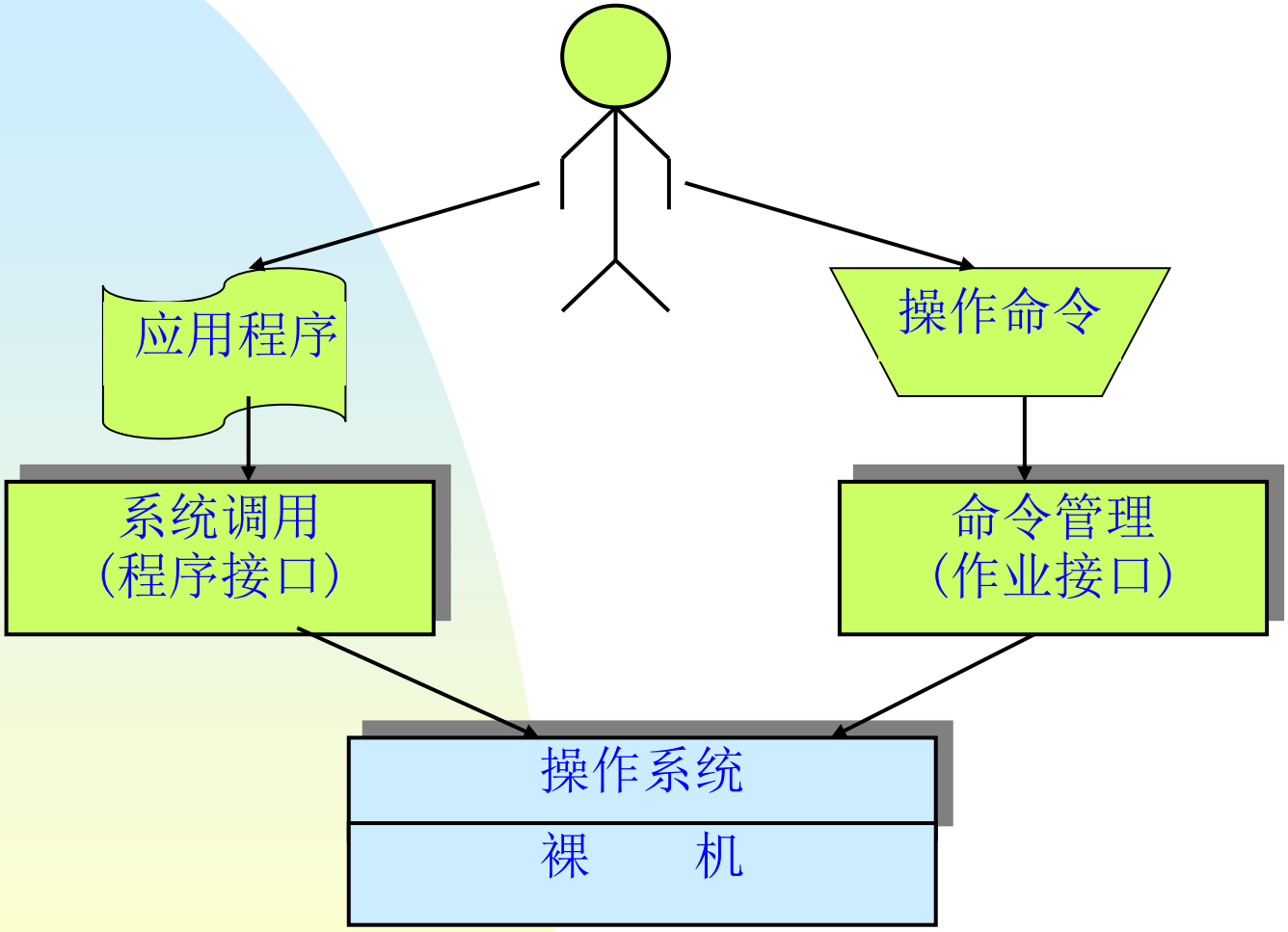
为程序员带来方便，使编程任务得以顺利完成，包括（编写程序，执行程序，数据I/O，信息获取等）。

功能服务：多个程序有效并发执行，共享系统资源，提高系统效率等方面功能。

即：方便用户及用户程序的执行及开发，提高系统操作效率 。

操作系统通过接口将服务提供给用户。

2. 用户接口



用户和操作系统间的接口

程序接口：一组系统调用，获得OS 底层服务，访问使用系统管理的各种资源。

操作命令：一组操作命令，OS为用户提供的组织控制其作业执行的手段。

OS提供服务的基本方式

操作接口(系统命令)：在用户一级使用的服务，用户可以直接在系统终端或机器键盘上使用，系统对每一命令立即响应、执行并回答。

又称作业级接口，操作系统为用户提供的操作控制计算机工作和提供服务手段的集合，通常有操作控制命令、图形操作界面(命令)、以及批处理系统提供的作业控制语言(命令)等等。

程序接口(系统调用)：操作系统提供的最基本服务，是在程序一级使用的命令。（操作系统提供的许多不同功能的子程序，用户程序在执行中可以调用，操作系统提供的这些子程序称为“系统功能调用”程序，或简称“系统调用”）

1.3.2 程序接口与系统调用

1. 系统调用

什么是系统调用？

系统调用的作用？

操作系统的主要功能是为应用程序的运行创建良好环境，为达到这个目的，内核提供一系列具有预定功能的服务例程，通过一组称为系统调用的接口呈现给用户。

系统调用概念

系统调用（system call）：是为了扩充机器功能、增强系统能力、方便用户使用而建立的。用户程序或其他系统程序通过系统调用就可以访问系统资源，调用操作系统功能，而不必了解操作系统内部结构和硬件细节，它是用户程序或其他系统程序获得操作系统服务的唯一途径。

系统调用把应用程序的请求传送至内核，调用相应的内核函数完成所需要的处理，把处理结果返回给应用程序。

内核的主体是系统调用的集合，可以把内核看作特殊的公共子程序。

系统调用的作用

系统调用是一种中介，把用户和硬件隔离开，程序只有通过系统调用才能请求系统服务并且使用系统资源。根本原因是为了对系统进行保护。程序的运行空间分为内核空间和用户空间，各自按不同特权运行。主要作用：

内核可以基于权限和规则对资源访问进行裁决，可以保证系统的安全性；

系统调用对资源进行抽象，提供一致性接口，避免用户在使用资源时发生错误，并且使编程效率提高。

2 API、库函数和系统调用

系统调用本质上是应用程序请求操作系统内核完成某功能时的一种函数调用，但它是一种特殊的函数调用，它与一般的函数调用有下述几方面明显的差别。

调用形式不同。函数（过程）使用一般调用指令，其转向地址是固定不变的，包含在跳转语句中；但系统调用中不包含处理程序入口，而仅仅提供功能号，按功能号调用。

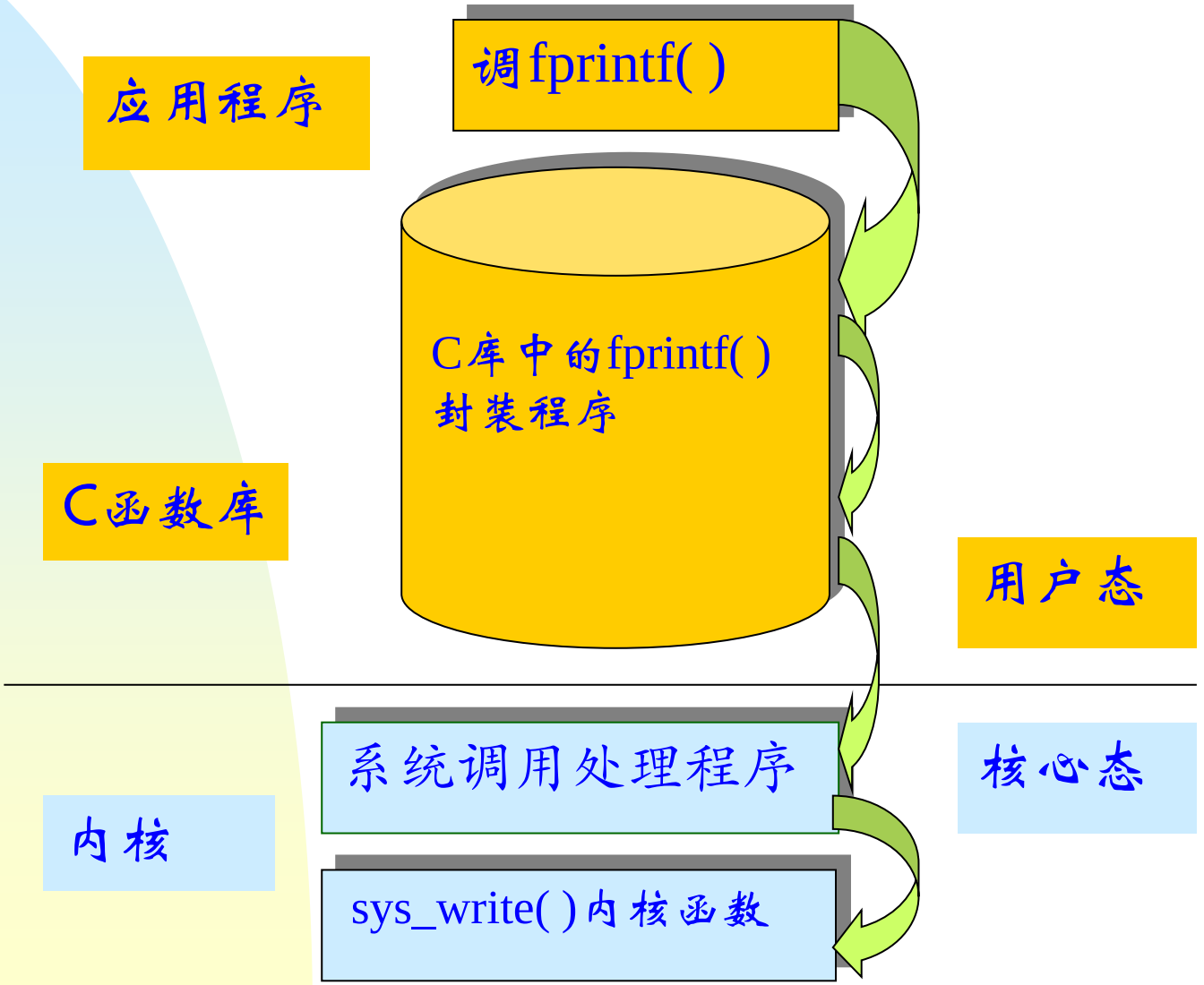
被调用代码的位置不同。函数（过程）调用，调用者和被调用代码在同一程序内，经过连接编译后作为目标代码的一部分。而系统调用的处理代码在调用程序之外（在操作系统中)。

程序使用一般机器指令（跳转指令）来调用过程（函数），是在用户态运行的；程序执行系统调用，是通过中断机构来实现，需要从用户态转变到核心态，在内核状态执行。

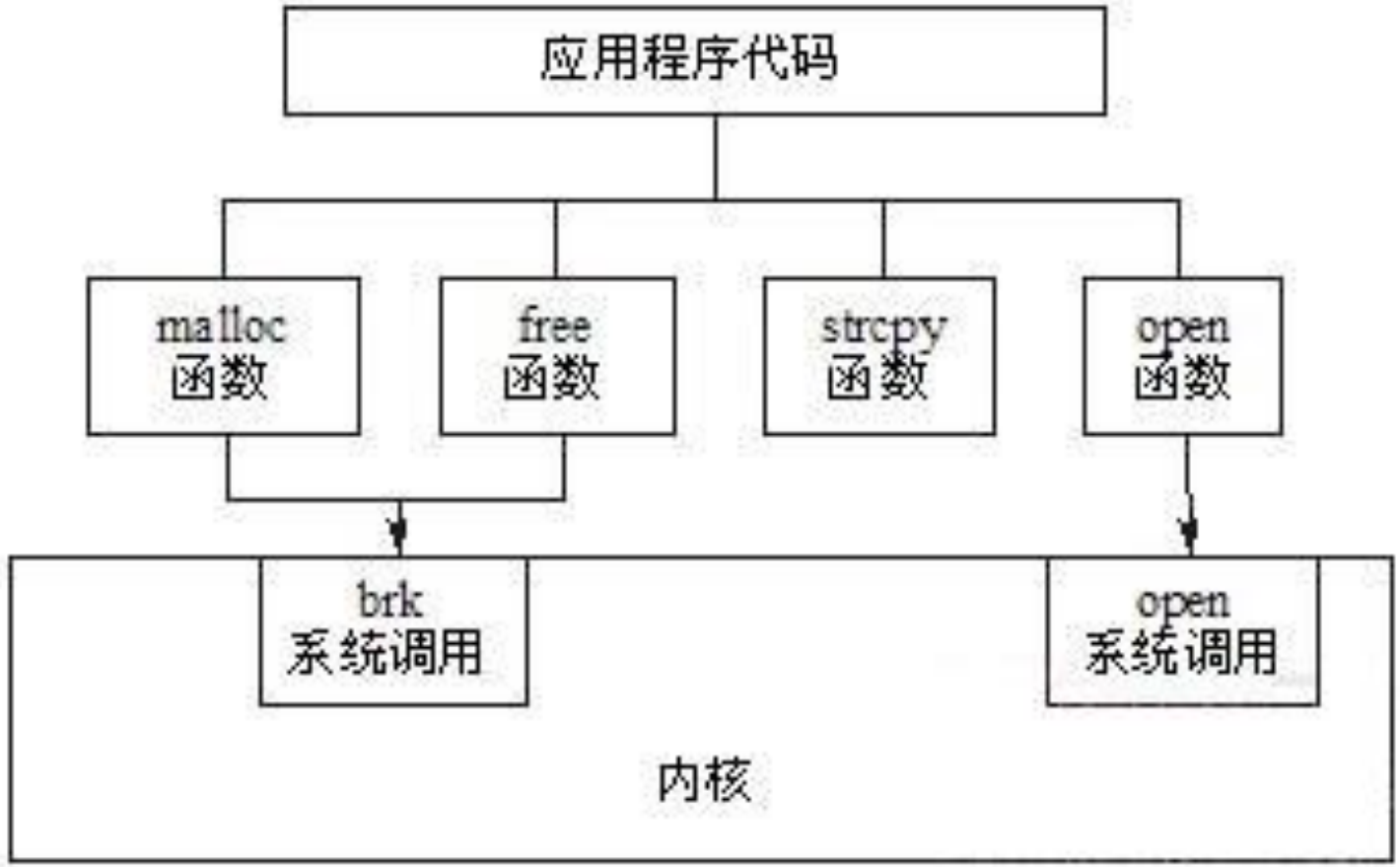
```
#include <stdio.h>
main()
{
    int a, b, sum;
    a=10;
    b=24;
    sum=add(a,b);          //这条语句就是函数调用
    fprintf("sum= %d\n",sum);  //这条语句C库函数调用，
                               //为什么不直接系统调用？
}

int add(int x, int y)
{
    int z;
    z=x+y;
    return(z);
}
```

API、库函数、系统调用的调用关系链



系统调用和C库


应用程序代码
malloc
函数
free
函数
strcpy
函数
open
函数
brk
系统调用
open
系统调用
内核

- 在用户程序角度看，库函数和系统调用之间差别不大，但是在操作系统实现的角度看，两者存在重要差别：
  - 库函数属于用户程序，在CPU用户态工作，系统调用属于系统程序（操作系统），在核心态运行。（处理机态的变化）
  - 用户可以替换库函数，但是不能替换系统调用。

- 回答前面的问题：应用程序为什么不直接系统调用？
- 应用程序直接使用系统调用存在两个困难：
- 第一，接口复杂、使用困难；
- 第二，应用程序的跨平台可移植性受到很大限制。
- POSIX标准：规定内核的系统调用接口标准，应用程序在遵循此标准的不同操作系统之间具有可移植性。

Linux/Unix 遵循POSIX 标准；

Windows操作系统也提供与POSIX兼容的子系统

Windows不公开系统调用，仅提供以库函数形式定义的API，即Win32API，或C函数库等

UNIX/Linux在标准C函数库中为每个系统调用构造一个同名的封装函数，屏蔽其下层的复杂性，负责把操作系统所提供的服务接口——系统调用封装成应用程序能够直接调用API（应用编程接口）。

Win32 API　for windows

POSIX API　for POSIX (UNIX like)

Java API　for java virtual machine

3 系统调用分类（六类了解）

1. 进程控制类系统调用

① 创建和终止进程的系统调用。如：**fork(),exit()**

② 获得和设置进程属性的系统调用。

③ 等待某事件出现的系统调用。

2. 文件操纵类系统调用

① 创建和删除文件。 **create**

② 打开和关闭文件。 **open ,close**

③ 读和写文件。 **read, write(fd,buf,nbyte)**

3. 进程通信类系统调用

在操作系统中经常采用两种进程通信方式，即消息传递方式和共享存储区方式。

4. 设备管理系统调用

申请设备、释放设备、设备I/O和重定向、获得和设置设备属性、逻辑上连接和释放设备。

5. 内存管理系统调用

申请内存和释放内存；虚拟存储器的管理。

6. 信息维护系统调用

建立和断开通信连接、发送和接收消息、传送状态信息、联接和断开远程设备。设置北京时间等。

4 系统调用的实现

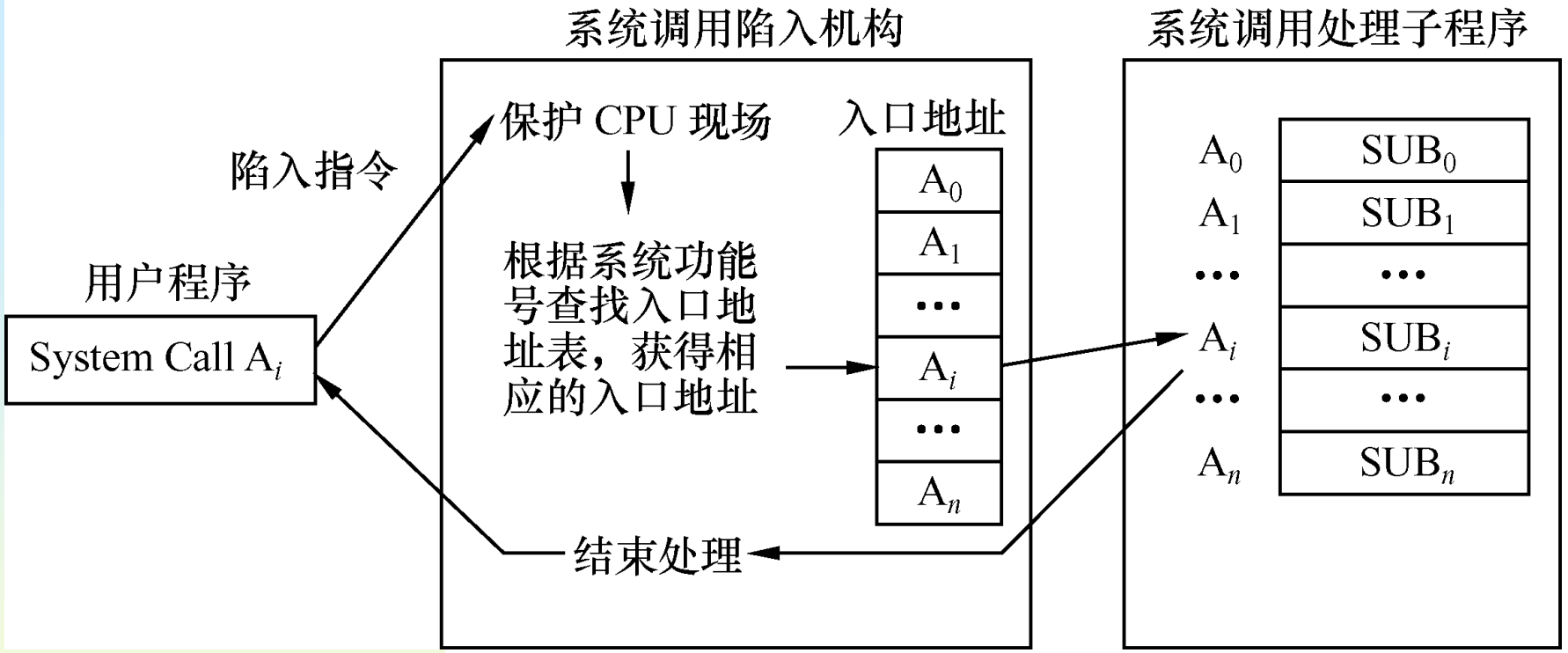
在操作系统中，实现系统调用功能的机制称陷入或异常处理机制，由于系统调用而引起处理器中断的机器指令称访管指令（supervisor）、陷入指令（trap）或异常中断指令（interrupt）。

系统调用的实现有以下几点：

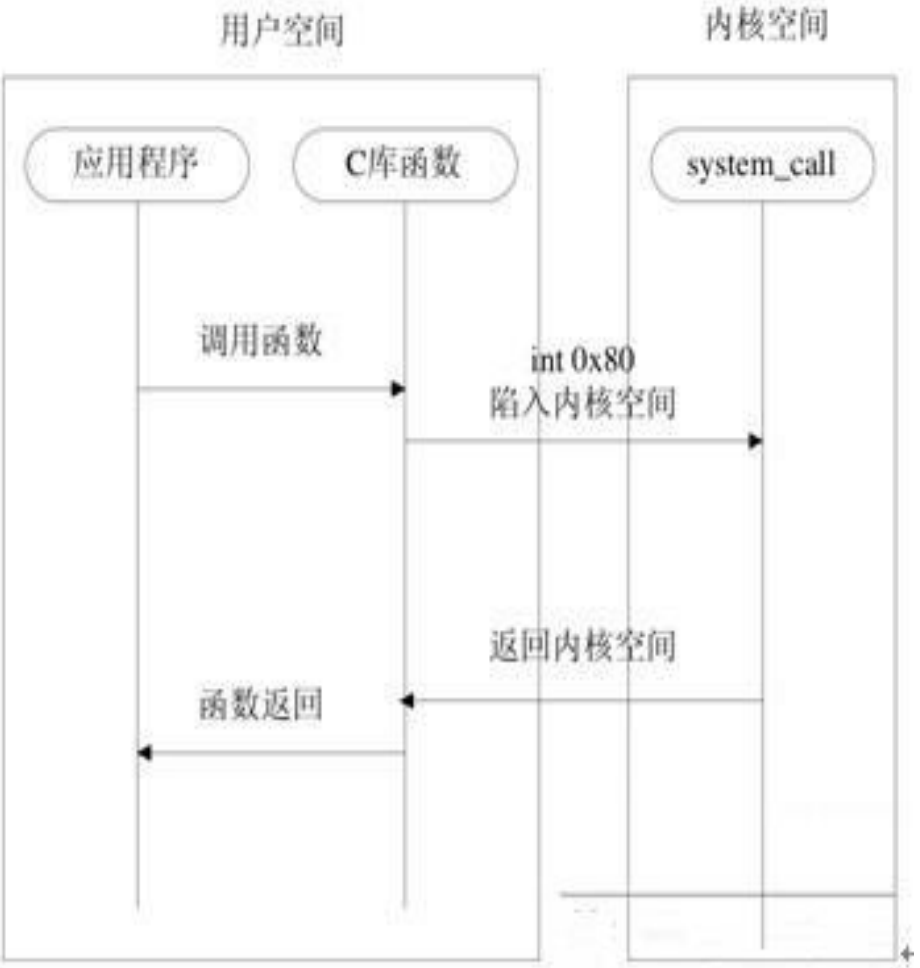
编写系统调用处理程序；

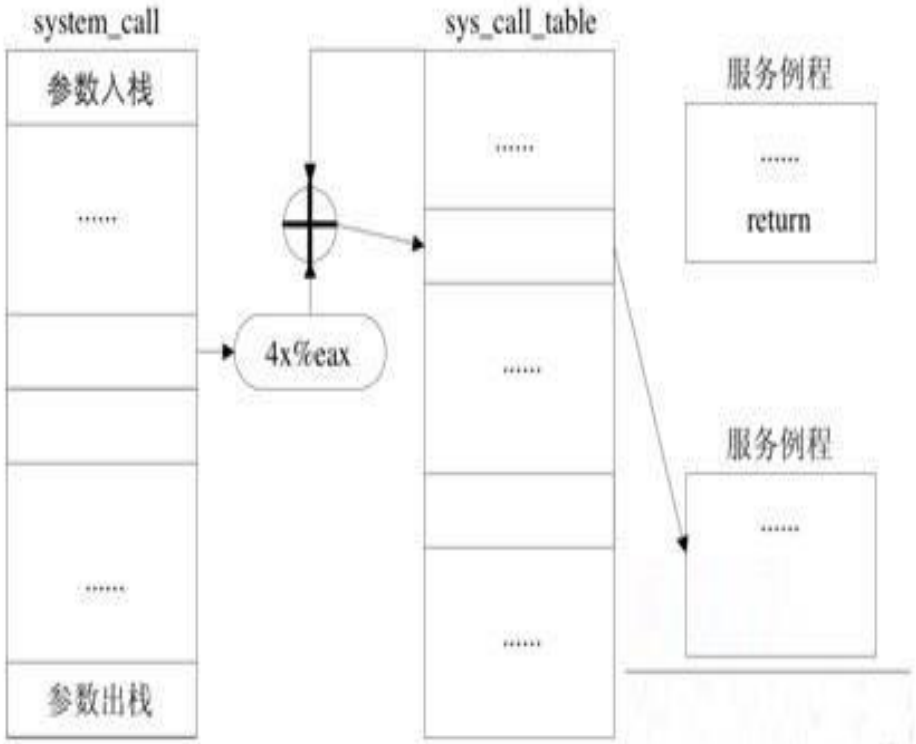
设计一张系统调用入口地址表，每个入口地址都指向一个系统调用的处理程序，有的系统还包含系统调用自带参数的个数；

陷入处理机制，需开辟现场保护区，以保存发生系统调用时的处理器现场。


系统调用陷入机构
系统调用处理子程序
保护 CPU 现场
入口地址
陷入指令
用户程序
System Call A_i
根据系统功能号查找入口地址表，获得相应的入口地址
A_0
A_1
…
A_i
…
A_n
SUB_0
SUB_1
SUB_i
SUB_n
结束处理

图/ 系统调用的处理过程


用户空间
内核空间
应用程序
C库函数
system_call
调用函数
int 0x80
陷入内核空间
返回内核空间
函数返回


system_call
参数入栈
.....
.....
参数出栈
4x%eax
sys_call_table
.....
.....
.....
服务例程
.....
return
服务例程
.....

系统调用（writer为例）处理过程：

write（fd,….）是一个系统调用库函数，用汇编写的，其中包含trap指令。

处理机运行到trap指令时，（按异常处理）保护现场，转入内核总控。

总控进一步保存现场，因为是trap引起的中断，所以转系统调用处理程序

系统调用处理程序根据系统调用号查找系统调用入口表，（内存中一个数据结构），得知该系统调用参数个数及处理程序入口地址，获得参数并转向write处理程序。

write处理程序将数据从用户缓存区考入内核的系统缓冲区，调磁盘驱动程序启动I/O函数，驱动程序生成一个磁盘请求包(数据的位置，要写到什么位置等等)，启动磁盘传输（或排入磁盘请求队列），等待传输完成（保护现场，这时可重新调度进程，CPU切换到另一个进程…）

磁盘I/O完成，产生一个磁盘中断（这时正在运行进程程序被中断），CPU运行磁盘中断处程序，磁盘中断处理程序清中断位，再从磁盘请求队列中取下一个请求，启动磁盘传输。然后找到刚完成的请求包，标志成完成状态，（先前进程重阻塞变成就绪，进行进程调度，先前进程被调度）恢复相应栈中的现场，转write系统调用的后续处理程序……

Linux系统调用及实现机制

1.Linux系统调用执行流程

1) 应用程序：read(fd,buffer,nbytes);

2) C库封装：movel $3,%eax

movel fd,%ebx

movel buffer,%ecx

movel nbytes,%edx

int $0x80

以上在用户态执行

3）执行系统调用，陷入内核态，进入系统调用入口

4）系统调用返回

1.3.3 操作接口与系统程序

操作接口又称作业级接口，是操作系统为用户操作控制计算机工作和提供服务的手段集合，通常可借助操作控制命令、图形操作界面(命令)、以及作业控制语言(命令)等来实现。

作业控制方式

联机作业控制方式与操作控制命令

脱机作业控制方式与作业控制语言

联机用户接口—操作控制命令

命令接口

联机命令接口：是为联机用户提供的，由一组键盘命令和命令解释程序组成。

脱机命令接口：为批处理作业的用户提供的，由一组作业控制语言JCL(job control language)组成。

程序接口

是为用户程序在运行过程中访问系统资源而设定的，也是用户取得操作系统服务的唯一途径，由一组系统调用组成。

图形接口

采用了图形化的操作界面，用图标将系统的各项功能直观逼真的表示出来，通过鼠标、菜单和对话框图来完成相应的操作。

脱机用户接口—作业控制语言

•批处理接口：作业控制语言JCL（Job Control Language）。

•用户使用JCL语句，把运行意图(需要对作业进行的控制和干予)写在作业说明书上，将作业连同作业说明书一起提交给系统。

•批处理作业的调度执行过程，系统调用、JCL语句处理程序或命令解释程序。

```
// HAROLD JOB,WILSON,MSGLEVEL=(2,0),PRTY=6,CLASS=B
// COMP EXEC PGM=IEYFORT
// SYSPRINT DD SYSOUT=A
// SYSIN DD*
     .
     .
  <SOURCE PROGRAM CARDS>
     .
     .
/*
// GO EXEC PGM=FORTLINK
// SYSPRINT DD SYSOUT=A
// FTOTF001 DD UNIT=SYSCP
// GO SYSIN DD*
     .
     .
  <DATA CARDS>
     .
     .
/*
//
```

IBM 370 使用JCL 处理批作业的例子

命令解释程序

接收用户输入命令并解释执行命令。实现起来两种方式：

一，解释程序包含命令执行代码，收到命令后直接转到相应命令处理代码执行，常用系统调用。不宜过多。

二，由专门的“实用程序”实现。执行时把命令对应处理文件装入内存。

Linux命令解释程序shell

Shell是linux提供给用户的命令语言解释程序，不是操作系统的组成部分，却体现许多操作系统特性。

Shell在用户态下运行。

主要功能是解释并执行用户输入的命令，以便执行各种实用程序。

系统程序（支撑程序）

系统程序又称标准程序或实用程序（Utilities），虽非操作系统的核心，但却必不可少，为用户程序的开发、调试、执行、和维护解决带有共性的问题或执行公共操作。

操作系统以外部操作命令形式向用户提供系统程序。它的功能和性能很大程度上反映了操作系统的功能和性能。

用户看待操作系统，不是看系统调用怎么样，而是看系统程序怎么样。

系统程序的分类：

文件管理、语言支持、状态修改、支持程序执行、通信等。

程序调试和排错、分类和合并、复制和存储。

电子邮件、远程登录、Web浏览、文字处理、电子表格管理、数据库管理、画图软件包、汇编、编译等。

XV6 系统调用

```
2924 // These are arbitrarily chosen, but with care not to o
2925 // processor defined exceptions or interrupt vectors.
2926 #define T_SYSCALL       64      // system call
2927 #define T_DEFAULT      500      // catchall
2928
2929 #define T_IRQ0          32      // IRQ 0 corresponds to
2930
2931 #define IRQ_TIMER        0
2932 #define IRQ_KBD          1
2933 #define IRQ_COM1         4
2934 #define IRQ_IDE         14
2935 #define IRQ_ERROR       19
2936 #define IRQ_SPURIOUS    31
2937
```

```
3100 void
3101 trap(struct trapframe *tf)
3102 {
3103   if(tf->trapno == T_SYSCALL){
3104     if(proc->killed)
3105       exit();
3106     proc->tf = tf;
3107     syscall();
3108     if(proc->killed)
3109       exit();
3110     return;
3111   }
3112
```

```
3374 void
3375 syscall(void)
3376 {
3377   int num;
3378
3379   num = proc->tf->eax;
3380   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3381     proc->tf->eax = syscalls[num]();
3382   } else {
3383     cprintf("%d %s: unknown sys call %d\n",
3384             proc->pid, proc->name, num);
3385     proc->tf->eax = -1;
3386   }
3387 }
```

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

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

```
3374    void
3375    syscall(void)
3376    {
3377        int num;
3378
3379        num = proc−>tf−>eax;
3380        if(num > 0 && num < NELEM(syscalls) &&
3381        syscalls[num]) {
3381            proc−>tf−>eax = syscalls[num]();
3382        } else {
3383        cprintf("%d %s: unknown sys call %d\n",
3384                proc−>pid, proc−>name, num);
3385        proc−>tf−>eax = −1;
3386        }
3387    }
```

```
5464  int
5465  sys_read(void)
5466  {
5467      struct file *f;
5468      int n;
5469      char *p;
5470
5471      if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) <
5472          0)
5472      return -1;
5473      return fileread(f, p, n);
5474  }
```

```
5313    // Read from file f.
5314    int
5315    fileread(struct file *f, char *addr, int n)
5316    {
5317        int r;
5319        if(f->readable == 0)
5320         return -1;
5321        if(f->type == FD_PIPE)
5322        return piperead(f->pipe, addr, n);
5323        if(f->type == FD_INODE){
5324            ilock(f->ip);
5325            if((r = readi(f->ip, addr, f->off, n)) > 0)
5326                f->off += r;
5327                 iunlock(f->ip);
5328             return r;
5329        }
5330       panic("fileread");
5331    }
```