Linux文件I/O编程

从宏观角度看虚拟内存管理

Linux的虚拟内存管理机制为**应用程序和驱动程序**提供了两种服务

1. 使每个进程都拥有自己独立的内存地址空间
2. 当物理内存不够4GB时，虚**拟内存管理模块会用外存空间模拟内存空间（也即虚拟）**，并且该模拟过程对应用程序是透明的

用户地址空间与内核地址空间

每个进程的4GB的独立地址空间，又划分为用户地址空间低3GB和内核地址空间1GB两部分

> 操作系统内核代码和数据存放在内核地址空间
>
> 每个进程自己私有的代码和数据存放在用户地址空间

> 虽然Linux的内核代码和数据被映射到了每个进程的地址空间中（所有进程看到的内容是相同的），但在实际的物理内存中，只有内核代码和数据的一份拷贝。如下图

用户态与核心态

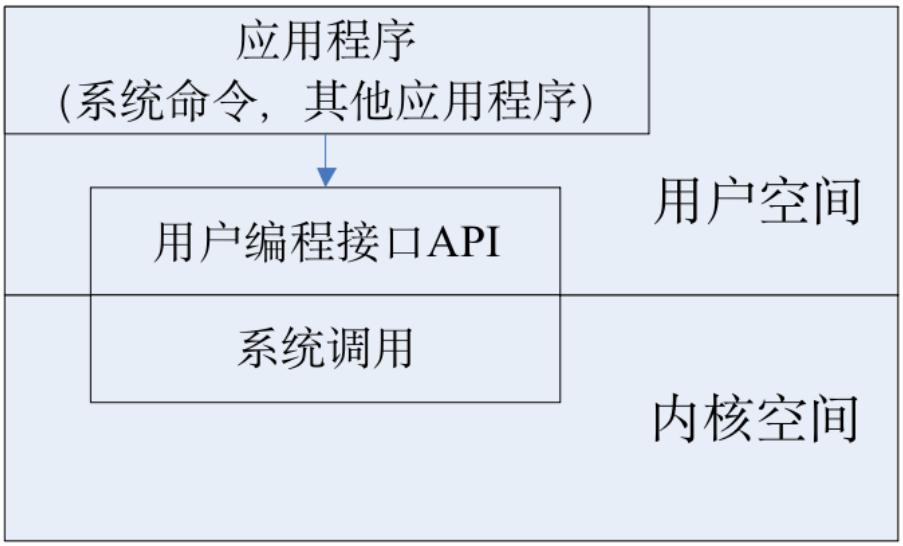
CPU都有几种不同的**指令执行级别**

> 举例：intel x86 CPU有四种不同的执行级别0-3，Linux只使用了其中的0级和3级分别来表示内核态和用户态

1. 高执行级别下，**代码可以执行特权指令，访问任意的物理地址**，这种CPU执行级别就对应着内核态
2. 用户态指相应的低级别执行状态，代码的掌控范围会受到限制，只能执行CPU指令集的一个子集

系统调用和API

区别



1. 系统调用一定在内核空间运行
2. API可能在用户空间，也可能涉及到系统调用在内核空间运行

系统调用

```
Linux系统调用非常精简（250个左右）
进程控制
进程间通信
文件系统控制
存储管理
网络管理
套接字控制
用户管理
```

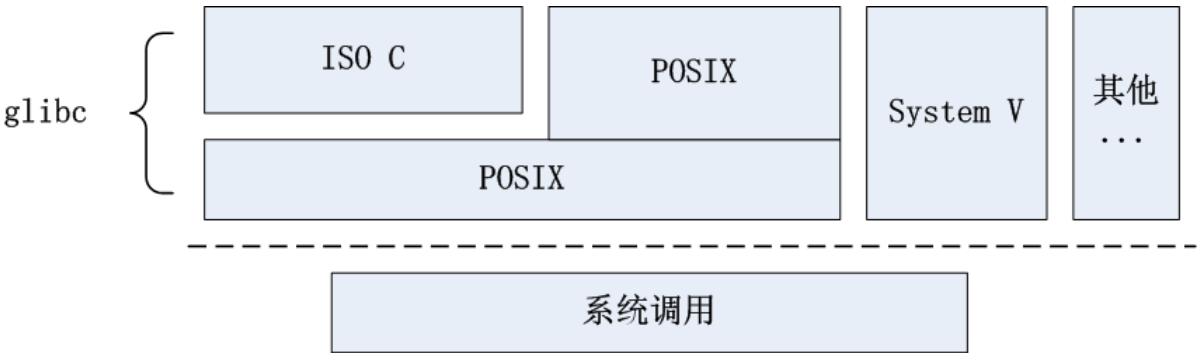
Linux用户编程接口（API）

用户编程接口（API）遵循了在Unix中最流行的应用编程界面标准**POSIX**，用于保证应用程序可以在源代码一级上在**多种操作系统上移植运行**。

这些系统调用编程接口主要是通过系统C库（libc）实现的

实际中程序员直接使用的是用户编程接口，用户编程接口的实现可能用到了系统调用

- 一个API函数对应一个系统调用
- 一个API函数调用了多个系统调用
- 一个API函数没有调用系统调用

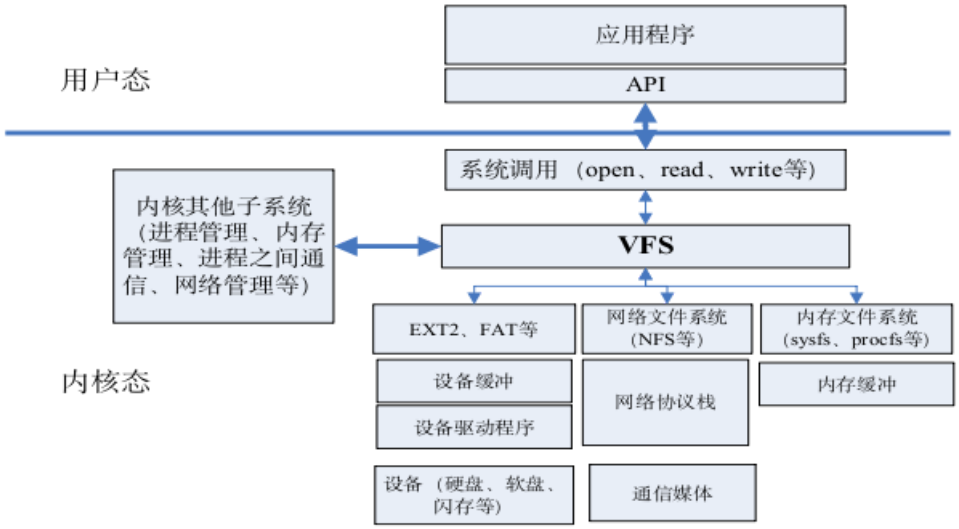


Linux文件I/O系统概述

虚拟文件系统

Linux的文件系统由两层结构构建

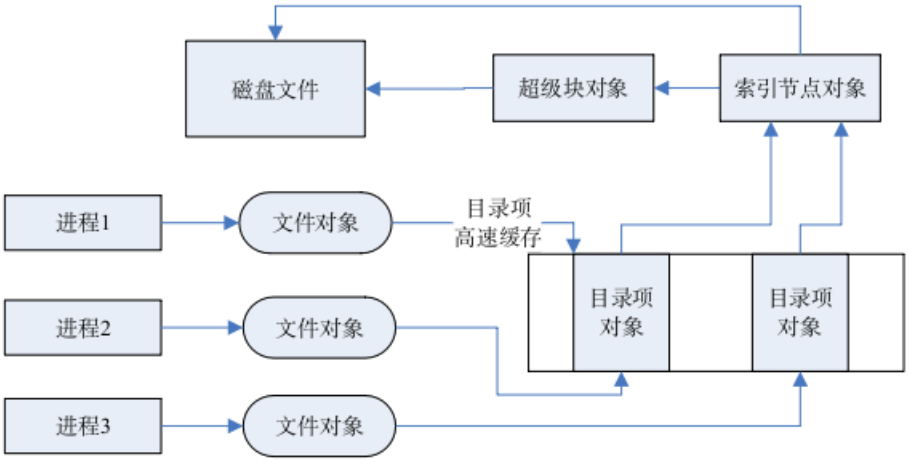
1. 第一层是虚拟文件系统（VFS）
2. 第二层是各种不同的具体的文件系统



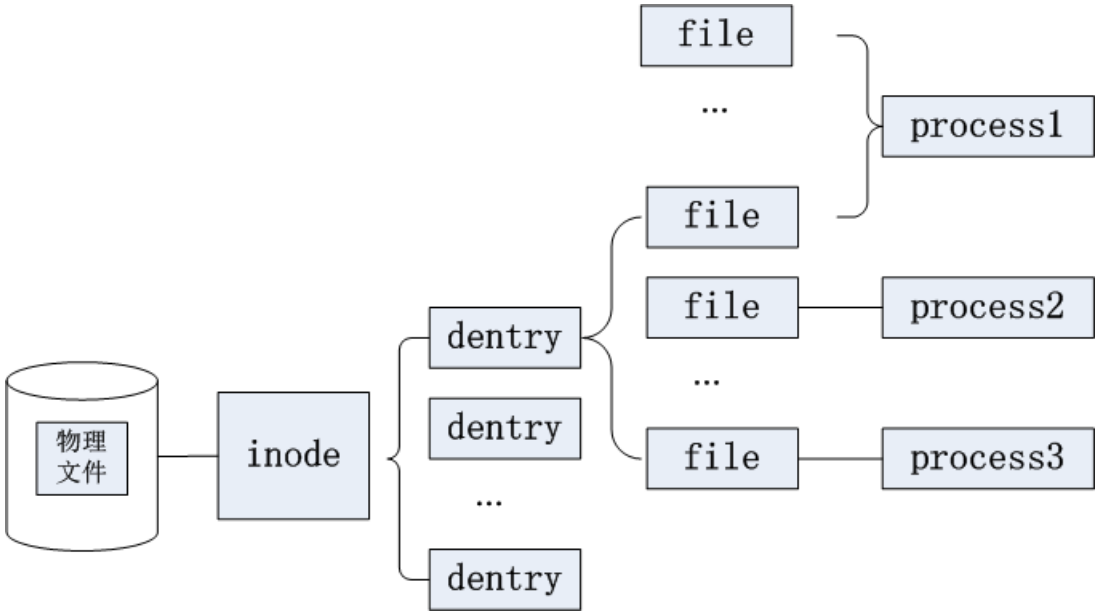
通用文件系统

通用的文件模型（common file model），这个模型的核心是4个对象类型

1. 超级块对象
2. 索引节点对象
3. 目录项对象
4. 文件对象



几个重要的结构体



- Inode结构体主要存储文件的元数据（metadata）信息，包括文件大小、访问权限、修改时间、实体数据存储位置等，但inode中不存储文件名
- dentry结构主要存储文件名及该文件在某个目录树中的位置信息。
- file结构体用于记录一个打开的文件的相关信息，如访问方式（可读、可写）、当前读写位置等。Inode与物理文件一一对应；
- inode和dentry之间是一对多的关系，因为同一个物理文件可以被"挂"到多个目录树上（学习文件硬链接的概念）；dentry与file之间是一对多的关系

文件描述符fd

一个linux进程会同时打开多个文件，因此会在自己的进程地址空间中创建多个file结构体，这些结构体形成一个file数组。

某个打开文件的文件描述符就是该文件对应的file结构体在进程file数组中的索引

- file数组大小限制了进程能够同时打开的文件数目（1024）
- 每个进程一般会默认打开3个文件：标准输入设备、标准输出设备和标准出错输出设备，对应的文件描述符分别是0,1,2

Linux底层文件I/O函数

特点：不带缓存直接对文件进行读写操作。不是ANSI C的组成部分，但是是POSIX的组成部分

> **ANSI C** 提供基本的标准库函数。

> **POSIX C** 增加了系统接口函数，使得程序可以进行更底层的系统操作，如文件系统操作、多线程编程等

详见PPT第七章15页

```
int  open( const char * pathname, int flags, mode_t mode);
int  close(int fd);
ssize_t  read(int fd, void * buf , size_t count);
ssize_t  write (int fd, const void * buf, size_t count);
off_t  lseek(int fd, off_t offset , int whence);
int  creat(const char * pathname, mode_tmode);
int fcntl(int fd, int cmd, struct flock *lock);
```

文件锁

个并发执行的进程可能会访问同一文件，访问包括读/写操作，系统如何控制其访问——文件锁

文件锁包括

- 建议性锁
- 强制性锁

文件锁又可分为

- 读取锁（又称共享锁）
- 写入锁（又称排斥锁）

上锁的函数有fcntl()与lockf()
- lockf()用于对文件施加建议性锁
- fcntl()不仅可以施加建议性锁，还可以施加强制锁,fcntl()还能对文件的某一记录上锁也即记录锁

阻塞式I/O与非阻塞式I/O

```
// 设置flag
O_NONBLOCK   // 非阻塞式
O_BLOCK      // 阻塞式
```

I/O处理模型

总的来说，I/O处理的模型有5种。

阻塞I/O模型

非阻塞模型

I/O多路复用模型🐵

select和poll的I/O转接模型是处理I/O复用的一个高效的方法

信号驱动I/O模型

```
Signal机制
```

异步I/O模型

```
aio
```

Linux的I/O机制

1. 同步阻塞I/O: 用户进程进行I/O操作，一直阻塞到I/O操作完成为止。
2. 同步非阻塞I/O: 用户程序可以通过设置文件描述符的属性O_NONBLOCK，I/O操作可以立即返回，但是并不保证I/O操作成功。
3. 异步事件阻塞I/O: 用户进程可以对I/O事件进行阻塞，但是I/O操作并不阻塞。通过select/poll/epoll等函数调用来达到此目的。
4. 异步事件非阻塞I/O: 也叫做异步I/O(AIO)，用户程序可以通过向内核发出I/O请求命令，不用等带I/O事件真正发生，可以继续做另外的事情，等I/O操作完成，内核会通过函数回调或者信号机制通知用户进程。这样很大程度提高了系统吞吐量

select函数，实现多路复用I/O的代码

```
fd_set：文件描述符集合类型
int FD_ZERO(fd_set *fdset);     /*判断集合是否为空*/
int FD_CLR(int fd, fd_set *fdset);    /*从集合中删除一个元素*/
int FD_SET(int fd, fd_set *fd_set); /*在集合中增加一个元素*/
int FD_ISSET(int fd, fd_set *fdset); /*判断集合是否含有某个元素*/
```

实际上，poll机制与select机制相比效率更高，使用范围更广

poll函数，实现多路复用I/O的代码

嵌入式Linux串口应用编程

Linux下对设备的操作方法与对文件的操作保持一致

1. Open，read，write，close
2. 需要对串口进行参数设置，波特率（115200），起始位比特数（1bit），数据位比特数（8bit），停止位比特数（1bit），流控模式（无）

终端三种工作模式

- **规范模式**，所有的输入是基于行进行处理，终端回显，在用户输入行结束符之前，系统调用read()读不到数据。支持行编辑，一次read()调用最多只读取一行数据
- **非规范模式**，所有输入即时有效，终端回显，不支持行编辑。设置MIN与TIME控制读操作
- **原始模式**，输入数据以字节为单位看待，终端不回显

保存原先串口设置

激活选项

设置波特率

设置字符大小

设置奇偶校验位

设置停止位

设置最少字符和等待时间

清除串口缓冲

激活配置

打开串口

读写串口

标准I/O编程

系统调用的开销，中断处理，用户态/内核态切换。

标准I/O提供流缓冲的目的是尽可能减少使用read()和write()等系统调用的数量，降低开销，提升性能

三种类型的缓冲存储

- 全缓冲：对于存放在磁盘上的文件通常是由标准I/O库实施全缓冲的。当缓冲区已满或手动flush时才会进行磁盘操作
- 行缓冲：当在输入和输出中遇到行结束符时，标准I/O库执行I/O操作。标准输入和标准输出就是使用行缓冲的典型例子。
- 不带缓冲：标准I/O库不对字符进行缓冲。

基本操作

打开文件

`fopen()、fdopen()和freopen()`

实现重定向

关闭文件

读文件

写文件

文件的读写与上锁代码

多路复用式串口操作