# 操作系统

atomic.LoadInt64[3] 很正常，因为对一个 int64 来说，它的大小是 8 个字节，如果是 32 位平台的话（字长 4 字节），CPU 一次最多操作 4 个字节，需要两次才能拿到全部数据，所以封装一个 atomic.LoadInt64 来实现原子操作；但是，对一个 int32 数据来说，它的大小是 4 字节，不管是 32 位平台（字长 4 字节），还是 64 位平台（字长 8 字节），CPU 应该都可以保证一次操作拿到数据，换句话说，如果读取一个 int32 数据，那么本身就应该是原子的，可是为什么会有 atomic.LoadInt32，这不是脱了裤子放屁

在 amd64 平台，Load 是用 Golang 实现的，等价于直接读取：

func Load(ptr \*uint32) uint32 {

return \*ptr

}

在 arm64 平台，Load 是用汇编实现的，并不是简单的一次操作：

TEXT ·Load(SB),NOSPLIT,$0-12

MOVD ptr+0(FP), R0

LDARW (R0), R0

MOVW R0, ret+8(FP)

RET

在 32 位操作系统，因为进程最大只能申请 3 GB 大小的虚拟内存，所以直接申请 8G 内存，会申请失败。

在 64位 位操作系统，因为进程最大只能申请 128 TB 大小的虚拟内存，即使物理内存只有 4GB，申请 8G 内存也是没问题，因为申请的内存是虚拟内存，等这块虚拟内存被访问了，因为物理空间不够，就会发生 OOM。

。假如 CPU 执行一条指令是 1 秒，那么访问内存需要 1 分 40 秒，从 SSD 上随机读取一次数据需要 4 小时 24 分钟，从磁盘读取一次数据需要 23 天多，一次横跨大西洋的网络请求则需要 4.8 年…你现在可以直观地感受到 CPU 有多么快了吧。

进程是无法直接操作IO设备的,其必须通过系统调用来内核协助完成IP动作,而内核会为每个IO设备维护一个buffer.

整个请求为:用户进程发起请求,内核接受到请求后,从IO设备中获取数据到buffer中,再将buffer中的数据copy到用户进程的地址空间.

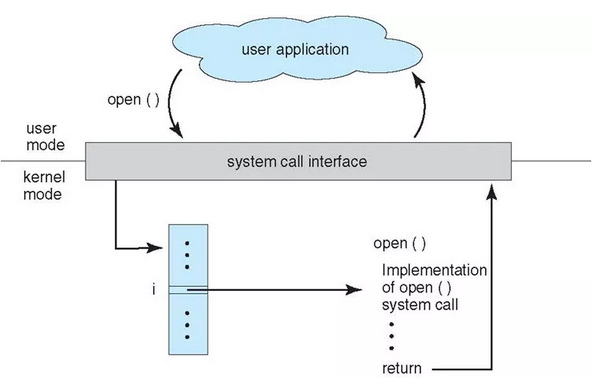
这里是用户进程请求内核把buffer的数据拷贝过来,这里就用到了系统调用read()函数了哦.

如果用户进程在read的时候buffer里面还没有数据,下面就是阻塞非阻塞啊,同步异步那套了.

## 系统调用

这里先说下系统调用,避免下面的不好理解.

如果一个进程在用户态需要使用内核态的功能,就进行系统调用从而陷入内核态,由操作系统代为完成.



Linux的系统调用主要有以下这些

| **Task** | **Commands** |
| --- | --- |
| 进程控制 | fork(); exit(); wait(); |
| 进程通信 | pipe(); shmget(); mmap(); |
| 文件操作 | open(); read(); write(); |
| 设备操作 | ioctl(); read(); write(); |
| 信息维护 | getpid(); alarm(); sleep(); |
| 安全 | chmod(); umask(); chown(); |

**看到了么,fork啊 sleep啊 read()啊这些操作系统的api其实都是系统调用.**

## 基本特征

并发 共享 虚拟 异步

## 作用

内存管理 文件管理 进程管理 设备管理

## 内核分类

大内核,大内核是将操作系统功能作为一个紧密结合的整体放到内核,由于各个模块共享信息,因此有很高的性能.

微内核,由于操作系统不断复杂,因此降一部分操作系统功能移出内核,从而降低内核的复杂性.移出的部分划分成若干的服务,相互独立.

微内核一个模块运行在内核态,其他的运行在用户太,这样需要内核态用户太经常切换,性能会低一些.

## 中断的分类

### 外中断

由cpu执行指令以外的事件引起,如I/O中断(包括键盘,鼠标啊,硬盘这些引起的中断),还有其他的时钟中断,控制台中断等.

### 异常

由CPU执行指令的内部事件引起,如非法操作码,地址越界,除0等.

### 陷入

用户程序中使用了系统调用.

## 进程与线程

进程与线程的区别,特定,状态.

调度算法.

临界区 同步与互斥 信号量

死锁 哲学家吃饭问题 银行家算法

似乎线程有很多优势，比如，数据共享效率高，可应对并发操作，有效利用等待时间等等，但是多线程的编程比多进程要复杂，同时，多进程的可靠性较好，因为进程间不会相互影响。实际情况还是需要自己分析拿捏的。但是一般来说，实际应用中常常采用“进程+线程”结合的方式，而不是非此即彼

## 进程间通信

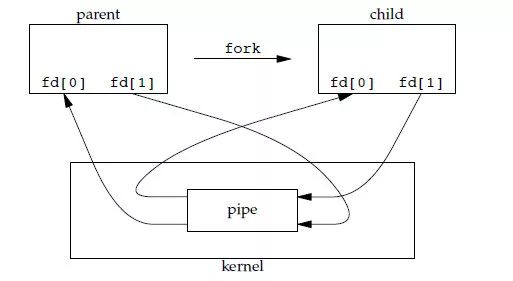
套接字是可以在不同机器进程中通信的方式!!!

### 管道pipe

这个有专门的api

#include <unistd.h>  
int pipe(int fd[2]);

只能在父子进程中使用,



### 消息队列

非常吊,好用.

### 信号量

这玩意只是一个计数器,用于为多个进程提供对共享数据对象的访问,

好像作用很小啊,就是个计数器啊,根本不如消息队列好用.

### 共享存储

允许多个进程共享一个给定的存储区.因为数据不需要再进程之间复制,所以这是最快的一个进程间通信.

以前我们知道进程是独立的拥有资源的单位,这个通信方式是让多个进程共享一块资源,所以这是一个特殊的方式,这个会借助信号量的帮助.

多个进程可以将同一个文件映射到他们的地址空间从而实现共享内存.

### 套接字

套接字可以用于不同机器之间的进程间通信.

## 内存管理

### 虚拟内存

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存,就是这个意思,操作系统把物理内存映射成逻辑内存,通过页关联起来,其实逻辑的内存比真正的物理内存要大,比如你的物理内存只有2g,但是逻辑内存可以映射出8g.

你的程序操作内存操作的是逻辑内存,比如你的这个程序要占用3g内存,那么你用的是逻辑内存的8g,但是呢,这样的话8g肯定无法映射到2g上啊,多啊,其实是这样的,一个程序在运行的时候并不需要把自己占用的逻辑内存全都加载到物理内存中,用到哪些逻辑页,把这些逻辑页加载到物理内存上就可以了.所以你的2g物理内存是可以搞出8g逻辑内存供应用程序使用的.

### 用户空间和内核空间

如果是4g的虚拟内存,对于linux操作系统来说,会有1g当作内核空间,保存内核的代码和数据,然后有3g分给应用程序使用,保存代码和数据.

外面的进程也可以通过系统调用进入内核,内核空间是所有进入内核的进程共享的,所以这样来看,应用进程可以使用的空间还是4g.

但是如果你不通过系统调用就使用0到一个数也就是那1G内存的话会直接coredump.

### 大页存储

大页内存(HugePages)，有时也叫“大内存页”、“内存大页”、“标准大页”。操作系统以内存页为单位管理内存，内存页的大小对系统性能有影响。内存页设得太小，内存页会很多，管理内存页的数组会比较大，耗内存，同时TLB(Translation Lookaside Buffer，页表寄存缓冲器，可理解为页表缓冲)大小是固定的，导致TLB MISS增加。在不同的应用场合，内存页的大小的最优值是不同的。所以一般的系统都支持多种内存页的取值。

大页内存的优势

“大内存页”有助于 Linux 系统进行虚拟内存管理。顾名思义，除了标准的4KB大小的页面外，它还能帮助管理内存中的巨大的页面(通常是2MB)。使用“大内存页”，你最大可以定义 1GB 的页面大小。对于那些内存操作非常频繁的业务来说，可以有效的提高性能。简而言之，通过启用大页内存，系统只需要处理较少的页面映射表，从而减少访问/维护它们的开销！

## BIOS

Cpu加电之后执行的第一条指令是在内存里,内存不是一断电就没了,不是的,内存分为RAM随机读存储,ROM只读存储,断电就没了的是随机读存储,ROM里面还是有以前写的内容的.

ROM只读内存的简称.断电后也不会消失,一般都是装机前就弄好了.

ROM里面就有一个地方存着BIOS的指令,cpu起来后就会去找这个指令,这个指令就会触发到把操作系统的程序(操作系统内核)加载到内存(这个程序叫bootloader),这样操作系统就起来了,然后控制权就给到了操作系统.

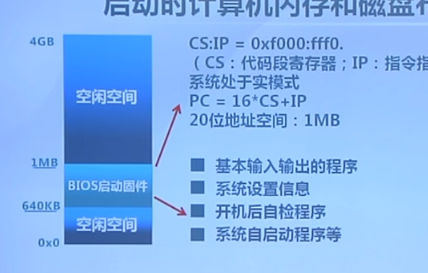
为啥BIOS叫基本输入输出系统呢,因为他要加载操作系统,操作系统内核是在磁盘里面的,所以BIOS必须要由输入输出的功能才可以呀!!!!

上面说了计算机是如何加载操作系统的,操作系统内核被加载进来后,他就是老大了,他也是被信任的东西了.但是还是需要中断和系统调用的.

中断解决,和外设硬件的联系,比如你键盘敲击了,我这我要用中断来弄.

还有就是软件出现异常了,也是中断的任务.

系统调用的是为软件提供服务的,操作系统不能让软件直接和cpu交互,不安全,不规范,只能对外提供接口,也就是系统调用来供应用程序调用.



DOS是一个操作系统,比较老的那种黑色命令行.

而BIOS是基本输入出入系统,通过这个基本输入输出命令,操作系统就是由BIOS拉起来的,也就是BIOS的运行不依赖于操作系统,操作系统不起来,不装BIOS都是可以执行的,可以修改操作系统的一些配置.

## NFS

NFS（Network File System）即网络文件系统，是FreeBSD支持的文件系统中的一种，它允许网络中的计算机之间通过TCP/IP网络共享资源

在NFS的应用中，本地NFS的客户端应用可以透明地读写位于远端NFS服务器上的文件，就像访问本地文件一样

nfs适用于Linux与Unix之间实现文件共享，不能实现Linux与Windows间的文件共享功能

nfs是运行在应用层的协议，其监听于2049/tcp和2049/udp套接字上

nfs服务只能基于IP进行认证

## iptables与firewalld

firewalld 与 iptables 都是 linux 中防火墙的管理程序，但其实其角色主要为对于防火墙策略的管理，真正的防火墙执行者是位于内核中的netfilter

firewalld默认是拒绝的，需要设置以后才能放行。而iptables默认是允许的，需要拒绝的才去限制 (所以k8s才需要手动的去关防火墙)

firewalld自身并不具备防火墙的功能，而是和iptables一样需要通过内核的netfilter来实现。也就是说，firewalld和iptables一样，它们的作用都用于维护规则，而真正使用规则干活的是内核的netfilter。只不过firewalld和iptables的结果以及使用方法不一样！

firewalld是iptables的一个封装，可以让你更容易地管理iptables规则。它并不是iptables的替代品，虽然iptables命令仍可用于firewalld，但建议firewalld时仅使用firewalld命令。

## Selinux

意思就是权限不是直接给用户, root用户也不能获取到所有权限, 而是经常到进程之类的.

SELinux提供了一种灵活的强制访问控制(MAC)系统，且内嵌于Linux Kernel中。SELinux定义了系统中每个【用户】、【进程】、【应用】和【文件】的访问和转变的权限，然后它使用一个安全策略来控制这些实体(用户、进程、应用和文件)之间的交互，安全策略指定如何严格或宽松地进行检查。(权限)