1. POSIX（可移植操作系统接口）代表一组基于Unix的标准接口，并设计了一些能力扩展。还有System V、BSD。Windows系统的主要编程环境/接口是win32 API（应用程序接口），为管理进程、线程、内存和外部设备提供了完成的函数集合。

每一个Java应用都唯一对应一个JVM实例，JVM虚拟机内有自己的线程、进程结构，映射到系统上。

不同种类计算机需要不同操作系统（嵌入式、华为大统一的鸿蒙）（linux类操作系统本质上还是看你选择了哪些内核模块等拼出来），比如大型机的操作系统主要是任务协作与充分利用硬件。Linux内核自己编译的话可以定制一些奇奇怪怪的特性，我想或许是对这句话的一种解释。

2. 硬件对操作系统的正常运行（防几种恶意破坏方式）提供了机制：

CPU实模式程序直接运行在物理内存上，直接操作物理内存，现在开机的时候还是先进入实模式然后再进入保护模式，这种方式存在几个问题：

* 地址空间不隔离：程序操作相同地址空间会造成互相影响甚至崩溃，而且安全性也得不到保证；
* 使用效率低：没有特别好的策略保证多个进程对超过物理内存大小的内存需求的满足；
* 程序运行地址不确定：程序运行时，都需要分配空闲区域，而空闲位置不确定，会带来一些重定位问题；
* 访问的内存有限，好像是1M

特权指令（提供指令集是CPU事情）：

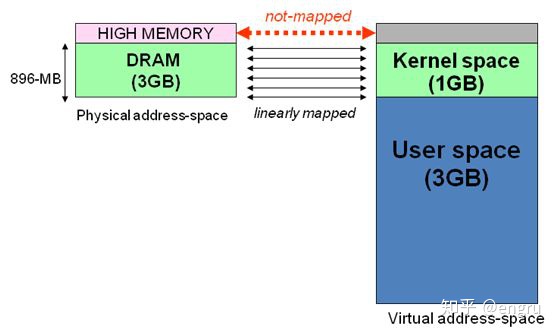
操作系统与CPU一起划分了Ring0操作系统内核层、Ring3用户层，各层有不同的CPU指令执行权限。这种级别存储在CS（code）、DS（data）、SS（stack）、ES（extend）等段寄存器的末尾2bit。CPU特权级别与操作系统用户无关。无论你是Root用户、管理员、来宾还是普通用户，都没有关系。所有用户代码都在环3中运行，所有内核代码都在环0中运行，而不管代码是代表哪个OS用户运行的。如果有异常将触发中断。系统调用（陷进），一种故意的异常中断，root用户有更多权限进行系统调用，调用内核中的代码来控制端口、网络等。所有的中断处理程序都在ring0（中断表就一个，内存地址00000H～003FFH），中断是用户程序交给操作系统内核处理，并存在CPU上下文切换（执行数据保存到CPU cache或者内存栈中，是一次函数调用）。

CPU保护模式（配合操作系统进行寻址方式的改变）：1. 段式内存管理 只能访问段上内存禁止对任意地址的访问（不过段的地址挺大的，分页使用虚拟地址之后才算安全）：在段寄存器(CS,DS,SS,ES)的基础上实现，改变段寄存器的功能，使其从一个单纯的段基址变成指向一个“段描述符”的指针。CPU能够对内存及一些其他外围设备做硬件级的保护。2.虚拟地址，让每一个进程都以为自己独享内存，起到隔离作用并且相比实模式程序变得可移植。页式内存管理 将虚拟地址映射成真实物理地址，所以可以认为与页式内存管理有关。

虚拟8086模式是运行在保护模式中的实模式，为了在32位保护模式下执行纯16位程序。它不是一个真正的CPU模式，还属于保护模式。

3. Linux 对内存节点进行分区；将节点分为DMA、Normal、High Memory 内存区；

* DMA内存区：直接内存访问区，通常为物理内存的起始16M；主要供I/O外设使用，无需CPU参与的外设和内存DMA（方便访问I/O设备）；
* Normal内存区：从16M到896M内存区；内核可以直接使用
* Hight Memory内存区：896M以后的内存区；高端内存，内核不能直接使用



进程空间便按照寄存器 分成 代码段、数据段、堆、栈等

4. DMA 零拷贝（NIO） 直接IO（DIO）

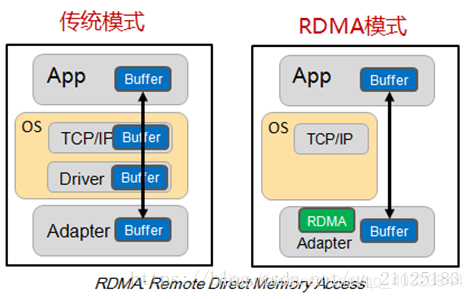
NIO 应该是指优化了内核态的拷贝情况

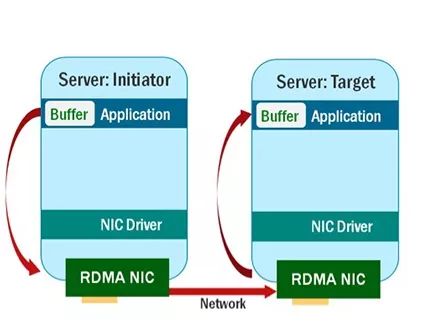
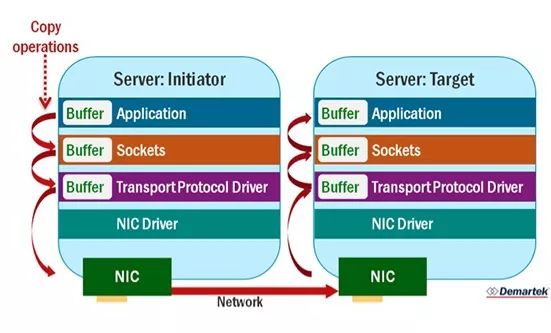
DIO 不经过内核态拷贝

DMA之前的是PIO，PIO又分PMIO和MMIO两种，MMIO不是一个字节一个字节拷贝的。即使是PMIO，32位CPU也可以每次读取四个字节，看Port大小。

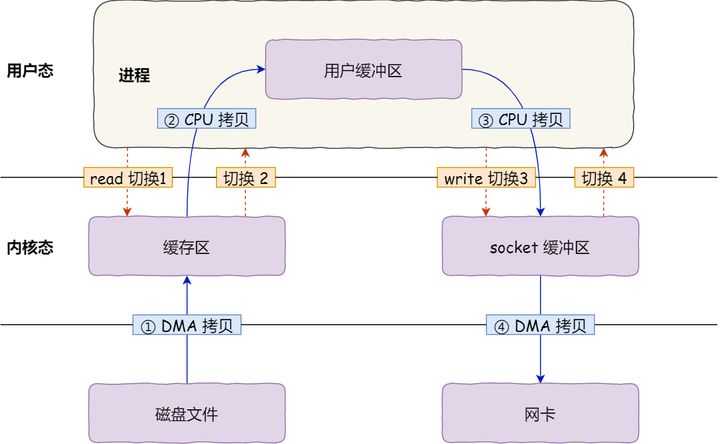
用DMA 方式的目的是减少大批量数据传输时CPU 的开销。采用专用DMA 控制器(DMAC) 生成访存地址并控制访存过程.优点有操作均由硬件电路实现，传输速度快;CPU 基本不干预，仅在初始化和结束时被中断参与， [CPU](https://baike.baidu.com/item/CPU) 与外设并行工作，效率高。拷贝数据需要占用CPU大量的计算周期（时间片）。

RDMA（Remote Direct Memory Access）增强了网卡的能力，使用DMA可能依然存在一定次数的拷贝（内核->内核），具有RDMA引擎（装载固化在硬件上的特定协议，驱动就像内置了一样）的网卡（RNIC，普通网卡NIC）直接和磁盘做pcie p2p，就一次拷贝也不用了，直接用PCIe的BAR就能完成。允许远程计算机之间进行内存访问，而不需要进行CPU处理数据拷贝Zero-copy与上下文切换Kernel bypass，也因此不会影响CPU中的缓存。如果用于与本机的磁盘之间，我估计像下面的SG-DMA一样，经过一次用户态内存而不是完全零拷贝。

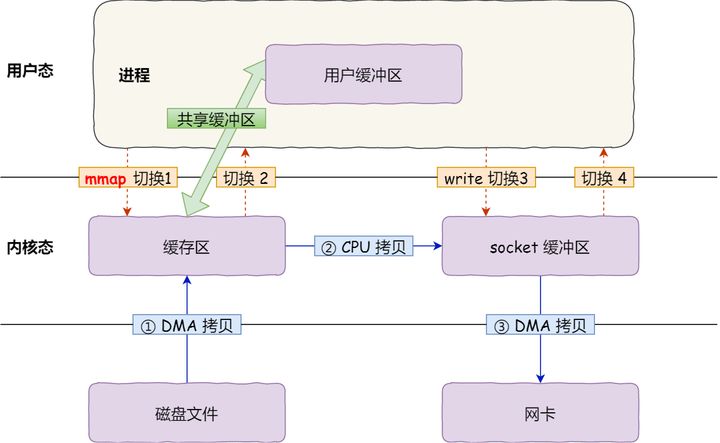




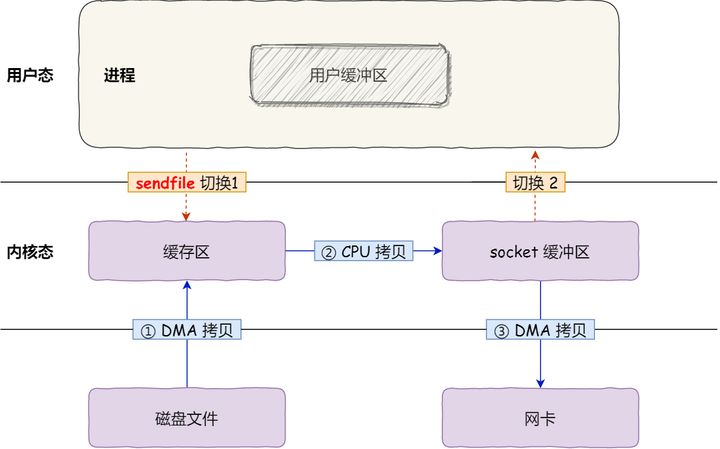
阻塞IO模型的网络请求中存在4次内核态与用户态之间的切换、4次拷贝，提高效率需要减少态切换 与 内存拷贝 次数。 不过没有考虑socket->网卡驱动缓冲区->网卡 的 拷贝。



以mmap代替read 能减少一次数据拷贝，mmap首先是虚拟内存的技术，用来创建新的虚拟存储器区域，映射后的操作走的是虚拟内存那套东西。缓存区是page cache。



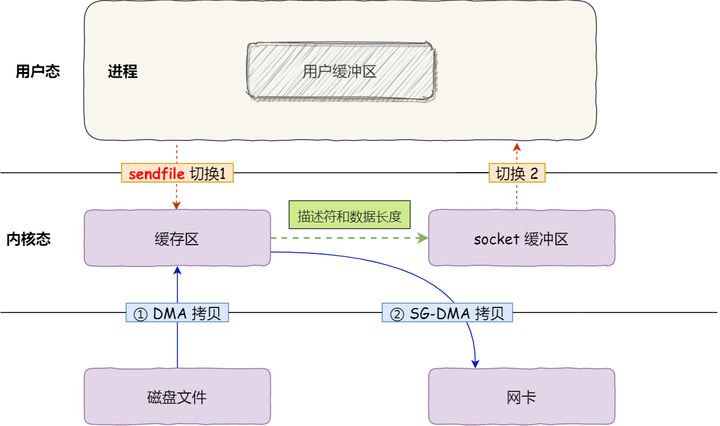
Linux内核2.1起，linux内核函数sendfile，可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝。



Linux内核2.4起，如果网卡支持 SG-DMA（The Scatter-Gather Direct Memory Access）技术（和普通的 DMA 有所不同），我们可以进一步减少通过 CPU 把内核缓冲区里的数据拷贝到 socket 缓冲区的过程。这时候sendfile效果发生了变化。

$ ethtool -k eth0 | grep scatter-gather

scatter-gather: on

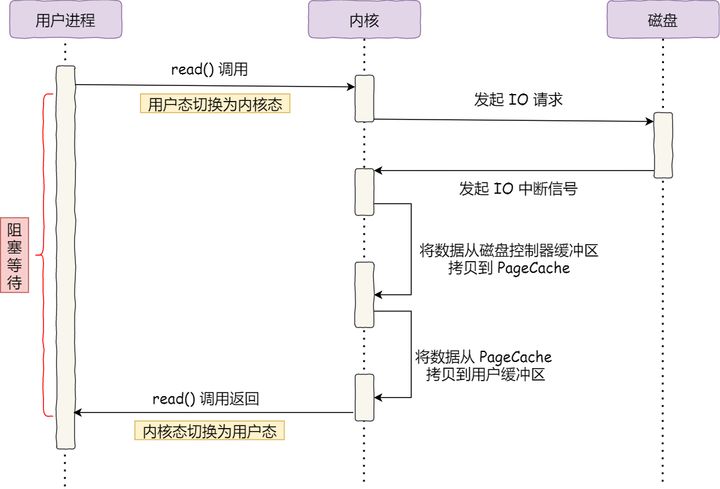


能将文件传输性能提示1倍以上，kafka、nginx中使用sendfile。

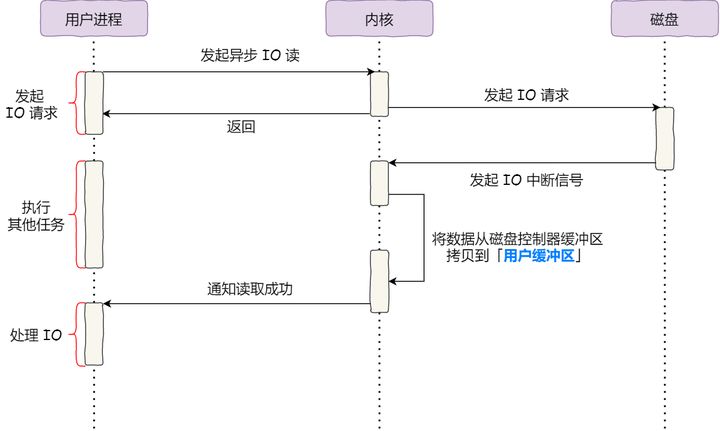
默认文件传输过程，其中第一步都是先需要先把磁盘文件数据拷贝「内核缓冲区」里，这个「内核缓冲区」实际上是磁盘高速缓存（PageCache）。读磁盘数据的时候，优先在 PageCache 找，如果数据存在则可以直接返回；如果没有，则从磁盘中读取，然后缓存 PageCache 中。而且PageCache中加载磁盘内容会预读取。但是传输大文件时，PageCache 预读会不起作用，以及由于长时间被大文件占据，其他「热点」的小文件可能就无法充分使用到。写的时候也会先到pageCache，页面修改后脏回写磁盘。内核的这两点的优化：

内核的 I/O 调度算法会缓存尽可能多的 I/O 请求在 PageCache 中，最后「合并」成一个更大的 I/O 请求再发给磁盘，这样做是为了减少磁盘的寻址操作；（机械硬盘寻址）

内核也会「预读」后续的 I/O 请求放在 PageCache 中，一样是为了减少对磁盘的操作；



以用户空间块设备层缓存区绕开 PageCache 的 I/O 叫直接 I/O，使用 PageCache 的 I/O 则叫缓存 I/O。在高并发的场景下，针对大文件的传输的方式，应该使用「异步 I/O + 直接 I/O」来替代零拷贝技术。 open时设置 O\_DIRECT，数据直接在用户地空间和磁盘间直接通过DMA传输。使用场景：1. 自己管理文件内存，比如数据库 2. 高并发大文件读写。大文件再配RAID技术。



综上，传输大文件的时候，使用「异步 I/O + 直接 I/O」，传输小文件的时候，则使用「零拷贝技术」。在 nginx 中，我们可以用如下配置，来根据文件的大小来使用不同的方式：

location /video/ {

sendfile on;

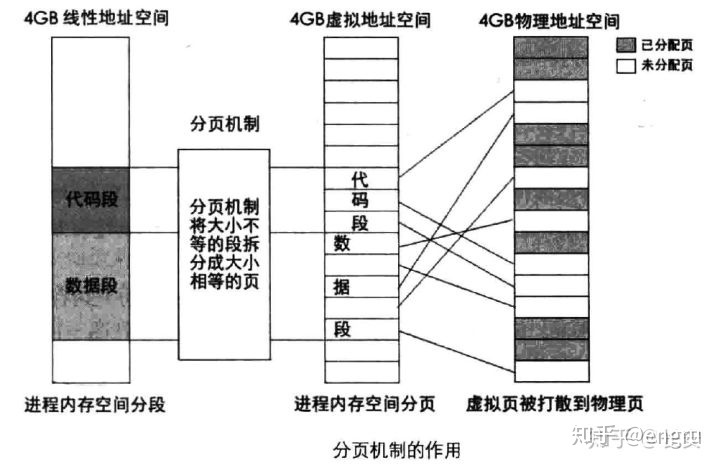
aio on;

directio 1024m;

}

如果你希望对数据进行压缩，这时候数据必须经过用户空间，所以不能使用空拷贝？

5. 分段与分页



分段相比于CPU实模式，分页相比于分字节 都是方便管理。分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护；分页主要用于实现虚拟内存，从而获得更大的地址空间。页的大小不可变，段的大小可以动态改变。

程序虚拟地址=段基址+段内偏移，比如 堆栈地址=SS+SP。不过操作系统并不是以字节为单位管理内存，而是划分成固定大小的页，使用多级页表的数据结构，虚拟地址结合页的映射转为实际地址。 Linux通过缺页中断和swap机制，实现换页。分段与分页的过程中，CPU都会进行权限的把控。

页内碎片与页间碎片都是对内存不可使用的浪费。大量分配小内存占用变量时容易出现页内碎片，解决办法是进行大内存占用，而大内存内进行字节甚至比特粒度的分配。页间碎片是分页算法导致的。

缺页将导致中断，一条MOV指令最多产生6次缺页中断（取指令1次+取数据2次，如果都跨页则X2），操作系统会选择一些空闲内存占用 缓存池一样。

页表的脏位（修改位）如果被标记了，就会写会磁盘交换空间

页面抖动/颠簸：在页面置换过程中的一种最糟糕的情形是，刚刚换出的页面马上又要换入主存，刚刚换入的页面马上就要换出主存，这种频繁的页面调度行为称为抖动，或颠簸。

页面置换算法：

最佳置换算法OPT：淘汰以后永不使用或最长时间内不再被访问的页面，但难以实现，运行这么多程序，并不知道各自会咋样。估计只能以预测或者定制的方式去接近。

FIFO、LRU（最近最少使用，实现一个双向链表，每次要缓冲新的页面时，o(n)遍历链表，选择最近最少使用的页面进行逐出操作。）

CLOCK是LRU的一种近似，它的性能比LRU要好，级别一样常数更小。数组（图形上可以想象环状时钟）各元素存在两个状态是否被访问过，存在两个指针，指针A按固定顺序遍历数组寻找下一个将要淘汰的位置（第一个未被访问过的内存 或者 完整遍历一遍后的起点），指针B按照固定速度将元素的状态置为0（文章里面说的是所有元素，不过我认为不必要）。

100M但是仅90M，除了页面置换，可以考虑内存压缩、代码优化。

6. 操作系统内核中处理 open() 系统调用的内核函数是 sys\_open()，sys\_open() 会调用 do\_sys\_open() 去处理主要的打开操作。它主要做了三件事情：

调用 getname() 从进程地址空间中读取文件的路径名；

do\_sys\_open() 调用 get\_unused\_fd() 从进程的文件表中找到一个空闲的文件表指针，相应的新文件描述符就存放在本地变量 fd 中；

函数 do\_filp\_open() 会根据传入的参数去执行相应的打开操作。

fopen相比较open，只能打开普通文件不能打开设备文件，而且读写都是先到缓冲区。如果是随机字节读写的话，不要缓冲区更快。

7. 优化系统调用：频繁的系统调用可能是系统中的主要耗时，不过这个我基本还没有遇到过，目前一般后端的主要问题还是在CDUD的效率组织上，我想应该需要在一个高并发的网关服务上注意这种事情。传统的int 0x80有点慢, Intel和AMD分别实现了sysenter/sysexit和syscall/ sysret, 即所谓的快速系统调用指令, 使用它们更快, 但是也带来了兼容性的问题。某些系统调用并不会向内核提交参数，而仅仅只是从内核里请求读取某个数据，例如gettimeofday()，vsyscalls即Linux内核一页内存mmap映射到用户空间的固定地址（只允许4个系统调用，可能被栈溢出攻击），该页面包含一些变量和一些系统调用的实现，被映射到用户空间的系统调用即可以在用户空间执行，避免了传统系统调用模式INT 0x80/SYSCALL造成的内核空间和用户空间的上下文切换，这时候更新速度/时间精度在于内核的时间中断情况，我认为是1ms。vDSO (Virtual Dynamically-linked Shared Object) 是vsyscall的主要替代方案，是一个**虚拟动态链接库**，将内存页面以共享对象形式映射到每个进程（不同进程内的不固定地址映射为同一个物理地址）（会映射代码段、只读变量段），用户程序在启动的时候通过动态链接操作，把vDSO链接到自己的内存空间中。动态链接保证了vDSO每次所在的地址都不一样，并且可以支持数量较多的系统调用。

对于获取当前时间系统调用，golang除了读取时钟信息外还会在这个基础上减去一个大致的预估时间。如果用time.ticker在程序内统计耗时，并过段时间同步一次time.now，好像也挺快的（端监控聊天记录待验证）。

8. 内存地址 可读、可写、可执行 权限控制，mmap分配空间的时候可以设置权限flag。违反规则触发中断，如果以read系统调用访问不可访问地址，将导致crash；如果将不可访问地址向有效fd write的话，不会crash 能返回一个errno。

数组越界问题，一些语言是在语言层面上抛出异常；对于C、C++ 直到因为内存页面权限问题，导致crash。

9. 时间片：表明进程在被抢占前所能持续运行的时间。操作系统调度核心=时间片抢占式+多级优先队列。中断时间片长度的基本取决于操作系统，它可能是多次时钟中断（一定是整数倍），只是时间片太短性能就消耗在中断处理上，时间片太长程序并行效果就比较差。

非抢占式内核非抢占式内核:高优先级的进程不能中止正在内核中运行的低优先级的进程而抢占CPU运行。

时间片的测量：时间片的长度由操作系统定义 windows 15ms mac 1ms 而不是CPU产商

#include <iostream>

using std::cout; using std::endl;

#include <windows.h>

int main(void) {

SYSTEMTIME systime;

long temp =0, sub=0;

long ptime =0,ntime =0;

for(long i1=0; i1 < 1e8; ++i1) {

GetLocalTime(&systime);

if(systime.wMilliseconds -temp>1) { //这里大于1有问题

ntime =systime.wMilliseconds;

if(ntime-ptime < 0 )

cout << ntime +1000-ptime << endl;

else

cout << ntime-ptime << endl;

ptime =ntime;

}

temp=systime.wMilliseconds;

}

return 0;

}

上面是学长的代码，出现一点问题，主要这个代码的测试效果不是很灵，但是大部分数据时正确的，得到了时间片是15这个结果。偶尔会出现几个很大的数字。要知道一次循环的时间长度，直接：

for(long i1=0; i1 < 1e8; ++i1) {

GetLocalTime(&systime);

if(systime.wMilliseconds -temp>0) { //这里大于1有问题

cout << systime.wMilliseconds -temp <<endl;

}

temp=systime.wMilliseconds;

}

#include <iostream>

using std::cout; using std::endl;

#include <ctime>

int main(void) {

clock\_t begin,end;

while(1) {

int i1 =0;

begin =clock();

for(int i2=0;i2 < i1; ++i2)

1+2+3;

end =clock();

if(end -begin != 0)

cout << end-begin << ' ' << i1 << endl;

else

++i1;

}

return 0;

}

这是我的代码，还好好奇看了下i1的值才发现有隐匿的神奇。本来以为程序会单独占用一个核心，如果因为时间片到了被T出来，之后也会马上进去，这样的话测量任务就是需找一个i来制作一个占用时间接近时间片长度的for循环。其实这个想法对不上这个程序，因为程序调用中断来进行输出的时候时间片。臆想中i的值应该差不多1e6以上。但是看输出，你会发现i的值很怪异，从小到大增大。后来想通了，是因为本进程时间片用完以后，这个核心下一个时间片并不一定就会给本进程（除非绑定了核心）。

#include <iostream>

using std::cout; using std::endl;

#include <ctime>

int main(void) {

clock\_t begin,end;

while(1) {

int i1 =0;

begin =clock();

end =clock();

if(end -begin != 0)

cout << end-begin << endl;

else

continue;

}

return 0;

}

考虑到装入时间零点几毫秒，所以最后还要人工减一下。

#include <iostream>

using std::cout; using std::endl;

#include <windows.h>

int main(void) {

SYSTEMTIME systime;

int begin,end;

int i =0;

while(1) {

GetLocalTime(&systime);

begin =systime.wMilliseconds;

GetLocalTime(&systime);

end =systime.wMilliseconds;

if(end -begin > 0) i++;

if(i > 10) break;

}

return 0;

}

这段代码排除了因为要输出而使得下一个时间片被分给其他进程这一个猜测，其实不管数不输出下一个时间片都不一定给自己。

10.

敲击键盘/滑动鼠标发生的事情：

1. 硬件捕捉动作：键盘按下电路板连通，此时多个引脚中不同的电信号即一串01组合；鼠标（以光电鼠标为例） 在一个频率附近的速度进行拍照，移动鼠标时拍摄的画面发生变化，对比计算出鼠标移动的相关数据。
2. 数据 以USB或者蓝牙到操作系统，被硬件驱动读取（驱动内置即免驱），触发系统中断，数据会先到操作系统给硬件分配的内存地址中，处理中断的时候被加载到CPU。
3. 中断处理程序会产生一些信号，以FIFO队列的形式被各个程序消费。比如信号会被当前 处于前台 且 激活状态 的程序消费。
4. 对于UI程序，比如QT写的，窗口自身可能并不处理事件，会将事件一部部下传，比如说将键盘事件发送给输入框。
5. 引起的画面变化 会被刷入核显或者独显的显存，然后展示在显示器上。
6. 补充：如果你是特殊信号，比如双击/回车打开程序，或者chrome里面百度搜索完的回车，就要再考虑进程、线程 甚至可能有协程 的更复杂的事情。

11. 微内核将传统宏内核中的驱动程序,甚至包括许多功能,比如文件系统, 网络, GUI等等, 都变成用户态的进程(服务),而内核中只保留最重要的功能: 进程管理,内存管理, 进程间通信, 以及硬件抽象层(HAL)。即使是驱动源码独立于内核仓库,具有明显的模块化特征,具有稳定的驱动接口,也不能是微内核,唯一的标准就是用户态的驱动程序和系统服务。微内核将驱动独立于内核空间, 大大降低了驱动故障的风险, 提升了安全性, 热更新驱动热重启驱动也变得可能；并且减少了内核的代码维护成本。

根据目前的硬件，系统调用时，如果要传递寄存器存不下的东西，比如内存中的一段字符串，那么内核是不能直接读取用户态的内存的（页表不同），内核需要先把用户提供的内存地址翻译到内核中的地址，然后内核分配一块内核使用的内存,将其复制进去,然后处理，返回数据也类似。比如内核处理网卡的数据,并将此数据传给浏览器.

宏内核的过程: 网卡发来中断, CPU执行网卡驱动设置的处理程序,要求网卡使用DMA将数据存到内核中,然后返回.DMA完成后,再次中断,系统的IP子系统分析已经存储的IP数据报,确定是TCP数据,根据端口号唤醒正在等待recv()的浏览器,将数据复制到recv参数里的缓冲区. 整个过程只发生一次数据的复制，两次中断.

那么微内核怎么办呢， 这里以最傻的微内核为例: 网卡发来中断, CPU执行HAL设置的处理程序, HAL确定这个中断应该被/bin/e1000d这个进程处理，于是唤醒/bin/e1000d， 把网卡发来的中断信息发给/bin/e1000d， e1000d将“写网卡内存映射的寄存器.....要求网卡进行DMA..."这种消息发给内核，内核收到之后执行，然后DMA结束之后把内存数据复制给e1000d，e1000d处理之后使用IPC把IP数据报发送给/bin/ipd，ipd处理后把TCP数据发送给浏览器。

12. 交换分区

windows：虚拟内存  
linux：swap分区

windows即使物理内存没有用完也会去用到虚拟内存，而Linux不一样 Linux只有当物理内存用完的时候才会去动用虚拟内存（即swap分区）

swap类似于windows的虚拟内存，不同之处在于，Windows可以设置在windows的任何盘符下面，默认是在C盘，可以和系统文件放在一个分区里。而linux则是独立占用一个分区，方便由于内存需求不够的情况下，把一部分内容放在swap分区里，待内存有空余的情况下再继续执行，也称之为交换分区

13. 内部寄存器指cpu内核中的寄存器，如r0、r1；外设寄存器指，比如STM32中，GPIO、I²C、SPI、CAN都叫做外设寄存器 一般集成在同一块芯片上。cpu的寄存器是单独编址的，内存的是内存编址，外设寄存器有类CPU将它与内存共同编址，还有一类单独编址并提供专用命令。

一个开发板的CPU地址引脚并不是所有的都与内存元器件相连的，如果该板上有外设（如一块独立显卡），那么CPU就需要分出一些引脚来与该外设的地址引脚相连，相当于将一部分内存寻址的空间分给了外设。一般的外设为了加快处理速度都有自己的片内RAM，分出去的地址空间也就与片内RAM物理连接起来，这样CPU就能像访问内存一样去访问外设的片内RAM。

13. 软中断 硬中断

1. 软中断是执行中断指令产生的，而硬中断是由外设引发的。
2. 硬中断的中断号是由中断控制器提供的（中断控制器接多个外部中断源的中断请求，并以电子线路过滤出优先级最高的请求发送到CPU的INTR端，可能集成在CPU内可能在CPU外），软中断的中断号由指令（传统的int 0x80有点慢, Intel和AMD分别实现了sysenter/sysexit和syscall/ sysret）直接指出，无需使用中断控制器。
3. 硬中断是可屏蔽的，软中断不可屏蔽。
4. 硬中断处理程序要确保它能快速地完成任务，这样程序执行时才不会等待较长时间，在中断处理中处于上半部位置。拿网卡来举例，在linux内核中，当网卡一旦接受到数据，网卡会通过中断告诉内核处理数据，内核会在网卡中断处理函数（上半部）执行一些网卡硬件的必要设置，因为这是在中断响应后急切要干的事情。接着，内核调用对应的下半部函数来处理网卡接收到的数据，因为数据处理没必要在中断处理函数里面马上执行，可以将中断让出来做更紧迫的事情。可以有三种方法来实现下半部：软中断、tasklet和等待队列。

3/0 在C语言中，会在CPU实际执行，所以会触发软中断

不过高级语言，可能会被语言内的异常处理机制兜底住

中断是为了避免对数据的阻塞等待，浪费CPU时间片。中断的 优先级决定某一个中断能不能打断另外某一个类型的中断，同类型也是同级。打断只能发生在下半部分。上半部分屏蔽所有类型中断，根本不允许被打断。

假如中断数量很多，而且这些中断都是自启动型的（中断处理函数会导致新的中断产生），则有可能cpu一直在这里执行中断处理函数，会导致用户进程永远得不到调度时间。为了避免这种情况，linux发现中断数量过多时，会把多余的中断处理放到一个单独的线程中去做，就是ksoftirqd线程。这样又保证了中断不多时的响应速度，又保证了中断过多时不会把用户进程饿死。一般情况应该是一个中断一个线程，进行处理。

**硬件中断**又可以分为外部中断（可屏蔽）和内部中断（不可屏蔽）两类。硬中断发生的时间不确定，可能在指令执行到一半的时候发生，中断号由中断处理器提供，一般就是外部IO设备因为IO读写需要涉及到的中断；内部中断是指因硬件出错（如突然掉电、奇偶校验错等）或运算出错（除数为零、运算溢出、单步中断等）所引起的中断。硬中断的屏蔽是从硬件角度，屏蔽电路就在中断信号的发送传送过程中（按照中断的级别屏蔽）。中断向量表（在8088时占用 1K内存，即256个中断向量，好像是一部分来自固件一部分来自操作系统）。

硬中断（或者说中断，包含了汇编的INT指令引起的中断）是分为上半部分与下半部分，上半部分用于完成必要措施——读取寄存器中的中断状态并清除中断标记，然后登记中断，即将下半部分加入到任务队列中。不允许被打断，屏蔽所有中断，这时候软中断还没有被触发）；下半部分进行关于中断具体的处理，可以通过多种机制实现（软中断、tasklet、工作队列）。如果中断处理很简单，就全在上半部分，不存在下半部分。

**软中断（softirq）**通常是硬中断服务程序对内核的中断，软中断可以从内核上屏蔽。软中断个数完全与操作系统的定义有关，linux最多注册32个软中断（目前使用了10个，interrupt.h中定义，轮询检测），估计由内核中的数组结构保存。软中断可以并发运行在多个CPU上，所以操作系统加了一个锁。由于软中断必须使用可重入函数，这就导致设计上的复杂度变高，作为设备驱动程序的开发者来说，增加了负担。而如果某种应用并不需要在多个CPU上并行执行，那么软中断其实是没有必要的。因此诞生了弥补以上两个要求的tasklet。出现在内核代码中的术语”软中断”常常表示可延迟函数(包括软中断和tasklet)的所有总类,即有时候软中断也包括tasklet(应该具体看上下文理解软中断的意思)。

工作队列（work queue）是另外一种将中断的部分工作推后的一种方式，它可以实现一些tasklet不能实现的工作，比如工作队列机制可以睡眠。这种差异的本质原因是，在工作队列机制中，将推后的工作交给一个称之为工作者线程（worker thread）的内核线程去完成（单核下一般会交给默认的线程events/0）。因此，在该机制中，当内核在执行中断的剩余工作时就处在进程上下文（process context）中。也就是说由工作队列所执行的中断代码会表现出进程的一些特性，最典型的就是可以重新调度甚至睡眠。

对于tasklet机制（中断处理程序也是如此），内核在执行时处于中断上下文（interrupt context）中。而中断上下文与进程毫无瓜葛，所以在中断上下文中就不能睡眠。因此，选择tasklet还是工作队列来完成下半部分应该不难选择。当推后的那部分中断程序需要睡眠时，工作队列毫无疑问是你的最佳选择；否则，还是用tasklet吧。

上下文是指创建时带入的参数与原有环境需要保存的一些变量。比如进程上下文即调度切换时需要保存的所有东西，中断上下文就可以理解为硬件传递过来的这些参数和内核需要保存的一些环境。

**信号**则是由内核（或其他进程）对某个进程的中断。

内存掉电是否会引发中断？因为这个应该是没有办法解决的。

驱动的编写其实就是写中断，即编写上下部的处理

中断自身也是函数，可以被更高优先级中断对

14. “GNU”这个名字是“GNU's Not Unix”的递归首字母缩写词。GNU是一个类Unix操作系统。它是由多个应用程序、系统库、开发工具乃至游戏构成的程序集合。GNU的开发始于1984年1月，称为GNU工程。GNU的许多程序在GNU工程下发布；我们称之为[GNU软件包](http://www.gnu.org/software/)。比如windows中安装MinGW，来使用gcc等工具。

15. 一个操作系统的启动过程：（主要介绍linux）

1. BIOS加电启动并自检（uefi、legacy。如果你是在Legacy模式下安装的系统，也只能在legacy模式下进系统。uefi只支持64为系统且磁盘分区必须为gpt模式，传统BIOS使用Int 13中断读取磁盘，每次只能读64KB，非常低效，而UEFI每次可以读1MB，载入更快。此外，Win10，更是进一步优化了UEFI支持，号称可以实现瞬时开机）。如果要从U盘启动PE系统，要靠BIOS。
2. 从MBR分区（扇区 0）引导扇区加载引导Bootstrap，大多数 Linux 发行版本使用的引导加载器有三种：GRUB、GRUB2 和 LILO。更强大的引导支持更多类型的文件系统，内核目录（自解压的压缩存储）可以放到任意位置上。EFI（UEFI）引导更好。
3. 按照用户选择/配置 加载 /boot 目录下的内核
4. 一旦内核自解压完成，则加载 [systemd](https://en.wikipedia.org/wiki/Systemd) 进程（其是老式 System V 系统的 [init](https://en.wikipedia.org/wiki/Init#SysV-style) 程序的替代品)，并转移控制权到 systemd。与init一样都有运行级别，不同的级别可以进行不同的任务配置。



1. 挂载文件系统、读取/etc配置、创建守护进程 。都从init/ systemd的会话组、控制终端剥离独立出来。创建多个虚拟终端tty（mingetty）等、X server（X11界面系统，Xorg是X11的一种实现）是init中级别5的守护进程。在一个终端中执行/bin/login

16. 硬件存储1000进制，CPU、内存、网络1024进制

17. 磁盘中有控制器，控制器中有寄存器。通过修改外设端口的IO命令（in、out）来设置寄存器操控磁盘。

18. 普通的内存是没有EEC（奇偶校验，内存会多出一位）功能的，而且普通的电脑也不需要这种功能，你要是想买这样的内存也不是不行，就是价格昂贵！ECC内存即纠错内存，简单的说，其具有发现错误，纠正错误的功能，一般多应用在高档台式电脑/服务器及图形工作站上，这将使整个电脑系统在工作时更趋于安全稳定。如果出现错误会触发CPU中断。

19. 核心转储（core dump），是操作系统在进程收到某些信号而终止运行时，将此时进程地址空间的内容以及有关进程状态的其他信息写出的一个磁盘文件。这种信息往往用于调试与追查当时的进程问题，linux(gdb)、windows都可以有，linux需要确定ulimit中core file size不要太小。内存越界、数组越界等异常都会产生core dump file。golang估计自己捕获了异常，panic也是可以配置打印堆栈等此时进程内信息。

程序调试断点原理:

**软件断点**：软件断点在X86系统中为中断指令INT 3，其二进制代码opcode是0xCC。当程序执行到INT 3指令时，会引发软件中断。操作系统的INT 3中断处理器会寻找注册在该进程上的调试处理程序。从而像Windbg和VS等等调试器就有了上下其手的机会。程序出错时常看到的”烫烫烫“、”锟斤拷“、”屯屯屯“等与这个终端指令有关。 Visual Studio有个有趣的特性是debug编译后，会把0xcc（INT 3）填入代码的空隙，这样一旦程序越界就会被VS捕捉而容易发现错误。而0xCCCC在中国的GBK编码是“烫”。

Windbg会将自己Attach到该程序的进程，通过程序PE文件的debug节找到调试信息。在调试信息里面找到加断点行所在的机器代码，并把头一个字节用WriteProcessMemory（）函数换成0xCC(INT 3)。注意左边是Windbg窗口，右边是用Process view打开的进程空间。在我们设置断点之前，左右的内容是完全一样的，这里要特别注意printf编译出来的第一个二进制代码0x68，我们会发现push操作代码0x68600e2900的第一个字节被windbg换成了0xCC也就是INT 3。这样windbg就可以在执行到这里时被调度。到达断点后，操作符又被还原为0x68，似乎什么都没有发生。

**硬件断点：**X86系统提供8个调试寄存器（DR0~DR7）和2个MSR用于硬件调试。其中前四个DR0~DR3是硬件断点寄存器，可以放入内存地址或者IO地址，还可以设置为执行、修改等条件。CPU在执行的到这里并满足条件会自动停下来。硬件断点十分强大，但缺点是只有四个，这也是为什么所有调试器的硬件断点只能设置4个原因。我们在调试不能修改的ROM（一般程序在内存RAM中，手机的ROM可擦写）时，只能选择这个，所以要省着点用，在一般情况下还是尽量选择软件断点。

20. CPU时间片抢占调度等 也是一种共享硬件的方式。

不论磁盘还是内存都能存储，来实现程序流

21. 用户态线程（比如JVM线程）与内核线程 1:1,1:n,n:n

用户态线程就是把内核态的线程在用户态实现了一遍而已，目的是更轻量化（更少的内存占用、更少的隔离、更快的调度）和更高的可控性（可以自己控制调度器）。用户态所有东西内核态都「看得见」，只是对于内核而言「用户态线程」只是一堆内存数据而已。操作系统层面的线程就是所谓的「内核态线程」；「用户态线程」则多种多样，只要能满足在同一个内核线程上执行多个任务的都算，例如 coroutine、golang 的 goroutine、C# 的 Task。实际执行是内核态线程。sleep 之后整个进程阻塞是因为混用了用户态和内核态的线程，正确的使用姿势是用用户态线程的替代品，例如 C# 的 Task.Delay。这是使用用户态线程的一个常见问题，即不能混用阻塞的 syscall。

22. 核显将占用一部分内存（300MB），独显用的少一些

23. CPU地址总线36bits，address sizes : 36 bits physical, 48 bits virtual。所以最大64GB内存。

24. 外设IO <https://blog.csdn.net/zz2862625432/article/details/101020933>

IO端口 IO内存： 每个外设都是通过读写其寄存器来控制的，外设寄存器也称为I/O端口，通常包括：控制寄存器、状态寄存器和数据寄存器三大类。根据访问外设寄存器的不同方式，可以把CPU分成两大类。一类CPU（如M68K，Power PC等）把这些寄存器看作内存的一部分，寄存器参与内存统一编址，访问寄存器就通过访问一般的内存指令进行，所以，这种CPU没有专门用于设备I/O的指令。这就是所谓的“I/O内存”方式。另一类CPU（典型的如X86），将外设的寄存器看成一个独立的地址空间，所以访问内存的指令不能用来访问这些寄存器，而要为对外设寄存器的读／写设置专用指令，如IN和OUT指令。这就是所谓的“ I/O端口”方式。但是，用于I/O指令的“地址空间”相对来说是很小的，如x86 CPU的I/O空间就只有64KB（0－0xffff)。

IO空间：X86特有的一个空间，与内存空间彼此独立的地址空间，32位X86有64K的IO空间。

IO端口：当寄存器或内存位于IO空间时，称为IO端口。一般寄存器也俗称I/O端口,或者说I/O ports,这个I/O端口可以被映射在Memory Space,也可以被映射在I/O Space。需要专门的IO指令访问，in/out指令。

IO内存：当寄存器或内存位于内存空间时，称为IO内存。

系统初始化时扫描PCI总线上的各个PCI设备（包括网卡），为这些设备分配总线地址，并建立起其卡上地址和总线地址的映射。写入综合地址后，从0号总线开始，每个PCI桥将综合地址中的总线号和自己的总线号相比，若符合，根据设备号+功能号寻找设备；若不符合，将综合地址传递给下一级总线的PCI桥继续寻找，直到找到设备，最后根据8位寄存器地址找到配置寄存器，此时通过I/O命令读写数据寄存器就可以读写配置寄存器了。设备驱动程序调用pci\_ioremap\_bar()将写入BAR的总线地址（保存在resource数组中）映射到系统内存的虚拟地址，之后CPU就可以通过虚拟地址访问PCI设备的存储空间，而不用再通过IO命令了

25. 上下文切换

上下文切换在不同的场合有不同的含义，在下表中列出：

| **上下文切换种类** | **描述** |
| --- | --- |
| 线程切换 | 同一进程中的两个线程之间的切换 |
| 进程切换 | 两个进程之间的切换 |
| 模式切换 | 在给定线程中，用户模式和内核模式的切换 |
| 地址空间切换 | 将虚拟内存切换到物理内存 |

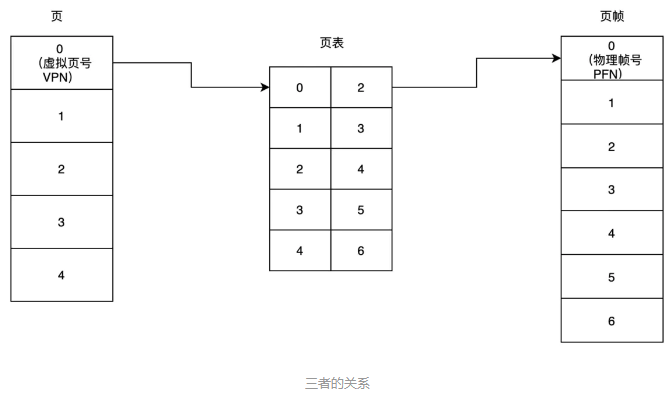
究竟什么时候会发生上下文切换？总共有三种情况：

| **上下文切换发生条件** | **描述** |
| --- | --- |
| 中断处理 | 中断分为硬件中断和软件中断，软件中断包括因为IO阻塞、未抢到资源或者用户代码等原因，线程被挂起 |
| 多任务处理 | 每个程序都有相应的处理时间片，当前任务的时间片用完之后，系统CPU正常调度下一个任务 |
| 用户态切换 | 这种情况下，上下文切换并非一定发生，只在特定操作系统才会发生上下文切换 |

| **损耗种类** | **描述** |
| --- | --- |
| 直接损耗 | CPU寄存器需要保存和加载, CPU 的pipeline需要刷掉, TLB实例（虚拟地址到物理地址翻译的硬件缓存）需要重新加载/页表重新加载（甚至引起页面颠簸）, 进程自己状态修改（PCB结构体）系统调度器的代码需要执行（让入相关队列并从相关队列中取出一个新的PCB） |
| 间接损耗 | 多核的cache之间得共享数据 |

26. 页表 页帧

页表其实是一个数组 用于把虚拟页号和物理帧号对应起来（页=页号+页偏移，映射的是页号），通过页表机制，使段内的页可以不必连续处于同一内存区域。



fork 的COW（exec或者写操作之前）便是复制了父进程的页表，但是没有分配页帧。目前操作系统为多级页表，我猜测实现上这个级数可能不是固定的。

内存外部碎片：未被分配且不能被利用的内存空间

内存内部碎片：被分配但不能被利用的内存空间

内存碎片解决办法：先申请分配一定数量的、大小相等(一般情况下) 的内存块留作备用，当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存。如果完全采用页表实现内存分配，因为页表的大小固定，所以页表之间不会有外部碎片。据说因为段页式中分段的存在，导致了外部碎片的存在？但段还是虚拟地址，页才是将虚拟地址转为物理地址。

深度操作系统：一个适合一般用户，桌面美化等做得很好但是装软件蛋疼，依赖被阉割。不同操作系统作用不同，有的为了大型计算，注重单节点硬件利用率与任务协作（不过现实中，可能重量级业务团队有自己的集群来保障计算能力，是数据将就计算，而不是计算讲究数据）。手机这种可能非对称处理器，需要内核支持 这种优化省电方案。

深度在2014年之前，自己做了一个WebKit/Html5的本地渲染引擎，可以直接用CoffeeScript像网页开发一样来编写桌面环境的界面，但是最后因为WebKit对于常驻性的桌面环境进程来并不适合，长时间不关机非常容易引起内存泄露和性能问题。包括2014年之前，采用的Python/QML来开发的应用都全部由 QtWidget/C++ 替换掉，主要是因为Python在国产CPU平台（龙芯、申威）上运行很慢，QML的解释特性和默认OpenGL渲染的机制都导致软件实际运行的性能不好。

深度这么多年几乎把所有的界面编程语言（Python、C、C++、CoffeeScript/JS、Golang）和几乎所有的图形界面库（wx、GTK+、Qt、WebKit、QML）都折腾过了。

最后停留在 Golang（估计因为方便的协程）、C++、QtWidget 的技术选型上，因为这些技术不论从稳定性、兼容性还是性能都非常好。

桌面：

RedHat主导开发的Gnome, Gnome2的稳定性和性能都非常好，包括后面的Mint桌面都是衍生于Gnome2, Gnome3是我作为个人开发者来看最失望的桌面环境，Gnome3完全抛弃了Gnome2的稳定和性能，Gnome3除了性能和内存控制不好以外，API的胡乱修改导致这5年几乎没有任何重大的进步

Suse主导开发的KDE：可以说功能最为丰富的桌面环境，包括产品质量以及性能每年都在飞速进步，但从个人看比较适合技术折腾党，功能和交互做的太复杂不适合普通用户（比如那个超级复杂的主题设置）

Ubuntu主导开发的Unity: Unity身上还是可以看到很多Gnome2的影子，产品质量和性能都不错，Ubuntu自己独立开发了Mir、Unity桌面环境和一些全局菜单等独特特性，虽然很多开发者很喜欢，但是整体视觉都聚焦在左上角（默认看着很想一把转角直尺）的设计，个人觉得并不符合中国人的使用习惯

Deepin主导开发的DDE: 作为DDE的开发者，客观的评价，产品质量、设计视觉都符合中国人的习惯，很多地方的交互设计细节要比其他桌面环境做的更加精致和简单，主要面向开箱即用的非技术型用户，从功能设计上并不适合技术高手和那些本来就喜欢折腾的技术型用户

其他轻量型的桌面环境，比如LXDE、XFCE、Enlightenment虽然性能极佳，但是从桌面环境的功能完整性和交互设计的精致性上都不够，比较适合有一定技术深度，追求性能和自己配置多于开箱即用的技术型用户

内核、驱动以及固件：主要用于驱动硬件可以正常工作，除了CPU、内存、磁盘外，最主要的要是要广泛兼容不同的网卡、显卡、声卡和外设等硬件设备

桌面环境的技术核心全部都依赖一批古老的协议，比如:x11、xcb、composite等基础技术，互联网上几乎没有文档教你怎么一步一步的构建桌面环境，现成的桌面环境因为发展多年，代码已经非常复杂，最多只能从核心代码片段有所学习。一个人要自上而下框架式的学习桌面环境核心技术并实现一套功能完备的桌面环境几乎不可能。

所以，大部分的Linux发行版都是基于Gnome、KDE改的，大多数都是在网上找一批主题（Gtk+主题、图标主题、光标主题、窗口边框主题、DM主题、锁屏主题）再配上一张好看的壁纸就可以做一套自己DIY的桌面环境。很多Linux社区的用户说，你看我自己都可以做桌面环境，甚至很多国内的Linux发行版厂商也是拿着社区的桌面环境然后换一套皮肤就去骗政府的钱，说自己是自主研发的操作系统（很可悲，这个世界上骗子永远比正直的人多）。

很多按钮的颜色、留白以及控件的对齐都是有问题的，甚至是粗糙的，为什么？这是由GTK+的主题设计导致的，GTK+强调的是一套原生控件打天下，但是面对无穷的应用和需求，不是所有的交互都能靠一套控件才能满足的，在大多数Linux社区开发者不具备控件自绘的能力前提下，大多数开发者都选择套用不合适的控件和控件主题去拼装，而GTK+主题的作者不可能在制作主题的时候测试所有软件，而且从GTK+先天限制也做不到，最后导致不论怎么套用主题，各种软件拼装在一起，就像各种质量参差不齐的补丁拼在一起一样的

但是Android、ios做到了，只是GTK+做的很糟糕

一个完整的桌面系统从技术剖面看，从下到上主要分这几层：

内核驱动层：前面已经说了，主要是保障硬件的基本功能和兼容性，所有的Linux发行版都大同小异，差别就是内核补丁集不一样，或者集成的驱动和固件多少的差异

显示服务层：从内核引导到plymouth(我们俗称的开机动画）后，只要你见到登录界面输入密码的时候，这时候X Server已经起来了， X Server简单来理解就是Linux系统中掌握着绘制图形界面生杀大权的“天神”，所有程序要绘制图形的时候都要发送消息到 X Server, X Server才会给你画出来。同时X Server也是事件输入（键盘鼠标）输出（显示器）的抽象层，开发者可以不用考虑底层驱动和显卡驱动细节，直接就可以使用X11/XCB的API进行应用开发，只不过更多的开发者是使用Gtk+/Qt这些在X11/XCB更上层的API进行应用开发

显示管理器：简单的理解就是你看到的登录界面提示你输入密码的那个地方

资源管理器：这一层主要由一系列的底层守护程序来监控硬件的状态，并汇报给上层的桌面环境和应用进一步操作，比如常见的就有网络、电源、磁盘、蓝牙、声音、键盘、打印等

桌面环境： 以深度桌面环境为例，主要包括桌面环境后台服务和守护进程、桌面环境对外提供图形开发工具库、二进制工具、DBus API服务和桌面环境UI界面层几个部分组成的，后面我会详细讲每一个细节

应用商店：主要提供系统的软件安装、卸载、升级等操作，保证用户可以安全易用的进行软件管理，同时提供了商店的评论和评分等功能

应用程序：主要包括深度开发的系列应用、合作开发的国内应用、Android应用、Windows应用和网页应用，为什么在Linux可以直接运行Android和Windows应用？ 我也后面再讲，嘿嘿

显示服务层X Server启动-》显示管理器启动（深度操作系统使用的是Ubuntu开发的LightDOM）

deepin-wine 支持50+多款Windows应用，通过内置 Android runtime支持200+多款Android应用（比如愤怒小鸟，各种视频客户端），deepin-xwalk 直接支持html5应用（比如Gliffy等）还可以自己记住窗口大小（而不是打开一个贼大的浏览器）

深度开源字体

社区中吵了十几年的Office排版字体，要不就是吐槽，要不就是拷贝微软版权字体，没有人站出来解决问题。5个版权字体，上千个符号，我们一个一个字体，自己重新画了一遍，然后开源给大家免费使用，行动才能推动开源，而不是靠嘴

1、连续分配方式

连续分配方式：为一个用户程序分配一个连续的内存空间。

连续分配方式进一步分为：单一连续分配、固定分区分配、动态分区分配以及动态重定位分区分配。

2、单一连续分配

最简单的一种存储管理方式，但只能用于单用户、单任务的操作系统中。

采用这种存储管理方式时，可把内存分为系统区和用户区两部分，系统区仅提供给OS使用，通常是放在内存的低址部分；

用户区是指除系统区以外的全部内存空间，提供给用户使用。

在早期的单用户、单任务操作系统中，有不少都配置了存储器保护机构，用于防止用户程序对操作系统的破坏。但在近年来常见的单用户操作系统中，都未采取存储器保护措施。一方面可以节省硬件，另一方面因为这是可行的。在单用户环境下，机器由一用户独占，不可能存在其他用户干扰的问题，这时可能出现的破坏行为也只是用户程序自己去破坏操作系统，其后果并不严重，只是会影响该用户程序的运行，且操作系统也很容易通过系统的再次启动而重新装入内存。

进程的两种优先级：一个是静态优先级,一个是动态优先级.静态优先级是用来计算进程运行的时间片长度的,动态优先级是在调度器进行调度时用到的，按照高动态优先级调用。在O(1)调度算法中，同种类型，后者按照一定的规则，在静态优先级上奖励或者惩戒。

进程的调度时间称为时间片，在时间片的过程中，内核支持使用CPU的进程被抢占，耗尽之后内核会为进程重新分配时间片。调度程序每一次选择动态优先级最高的且还有时间片的程序。交互性进程优先级高，但是时间片与后台进程相比更加短。

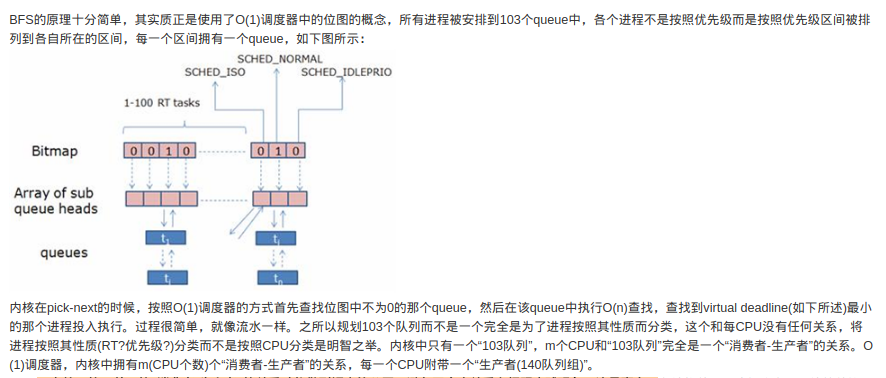
操作系统课上介绍了理论的几种调度算法：FIFO、最短作业优先（随着执行，剩余时间会变短，一般期末考出题目）、截止时间调度、最高优先级优先（其实都是优先级调度）、时间片轮转抢占式调度 。注意持续饥饿。

自Linux 2.6以来(严格说应该是2.5)，O(n)调度器（单一链表）被人们认为是一种千年之前就应该抛弃的东西被重重的甩开了，此后出现了O(1)，CFS，BFS等，再也没人提起O(n)了：

O(1)调度器 ，2.6内核采用了O(1)调度器（调度时间与进程个数无关），该调度器引入了每CPU的优先级队列组，每组队列包含active（存在剩余时间片）和expire（已经耗尽时间片）两个队列，采用启发式算法动态调整优先级，尽可能的进行时间片补偿和惩罚等动态计算，多CPU之间的优先级队列负载均衡。

CFS (Completely Fair Scheduler)，最多支持4096颗CPU的调度，于是就会存在一些复杂的机制。负载均衡机从高负载CPU调度队列取任务之前会先锁住目标对象，锁的代价不小。O(logN)，运行队列基于红黑树实现。因为一个核心一次只能执行一个任务，所以给每一个进程一个变量记录已经运行时间vruntime，从而实现一段时间之内的公平性。每一次调度器选择vruntime最小的执行，优先级高的进程vruntime增长得慢。

BFS (Brain Fuck Scheduler) 针对Linux桌面环境,真正有几个人用有上千 CPU 的电脑呢？为什么要为这种扩展性牺牲桌面性能。BFS 就在其间做了取舍，仅仅支持最多 16 个 CPU ，把问题外沿做小，让算法更简单精悍高效。抛弃了cgroup等特性。bfs在资源分配上比较注重前台资源，注重交互性能，所以有一段时间Android一个分支使用了这个调度器。

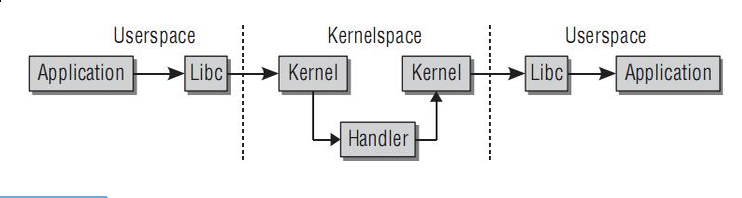


其实内存分配一般有三个原则，最快、最优和最差。最快比较好理解，就是寻找到合适的节点就立即分配内存，我们在前面一篇博客采用的就是这个方法。最优呢，就是寻找可以满足当前内存分配的最小节点，这样不会有很大的浪费，但是有可能会产生碎片节点。最后一种就是最差分配算法，说是最差效果未必最差。因为在大的内存分配的时候至少不会很快产生内存碎片，对整个系统的稳定来说有可能是好事。

系统调用的过程与中断调用类似：

每一个系统调用的函数对应着内核里的一个具体实现，每一个系统函数都有一个相应的数字对应，即系统调用号，这个数字计算出系统调用函数指针的偏移。

  系统调用号通常存入eax寄存器，然后当切换到到内核态后，内核根据系统调用号来查找到对应的系统调用处理例程的函数名，从而找到对应的代码入口地址。系统调用切换过程如图所示：注意libc，windows有libc但是不作为操作系统的核心部分，所以下图描述的应该就是Linux



从libc也可以看到Linux中的中断处理程序是C语言编写的。

特权指令与敏感指令的区别：

特权指令：只能在ring0级别才能正常运行的指令，否则就会报错。

敏感指令：操作特权资源的指令，包括修改虚拟机的运行模式或者下面物理机的状态；读写时钟、中断等寄存器；访问存储保护系统、地址重定位系统及所有的I/O指令。

对于一般 RISC 处理器，如 MIPS，PowerPC 以及 SPARC，敏感指令肯定是特权指令，但是x86 例外，但是x86绝大多数的敏感指令是特权指令，但是由部分敏感指令不是特权指令。

## 分区与文件系统

对分区进行格式化是为了在分区上建立文件系统。一个分区通常只能格式化为一个文件系统，但是磁盘阵列等技术可以将一个分区格式化为多个文件系统。

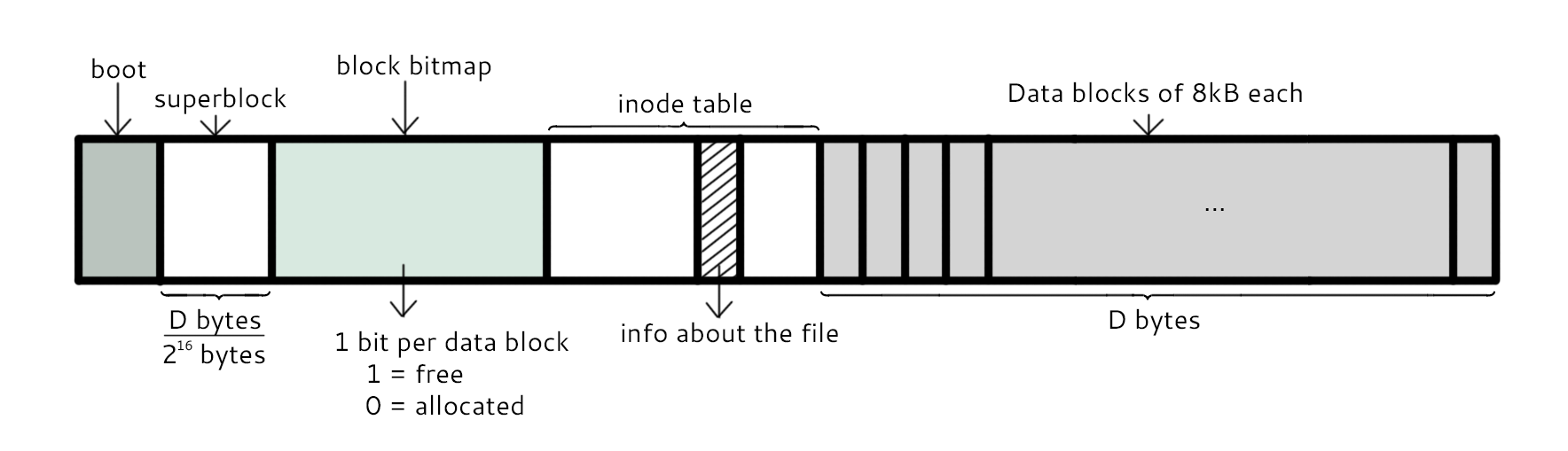
## 文件系统组成

最主要的几个组成部分如下：

* inode：一个文件占用一个 inode，记录文件的属性，同时记录此文件的内容所在的 block 编号；
* block：记录文件的内容，文件太大时，会占用多个 block。

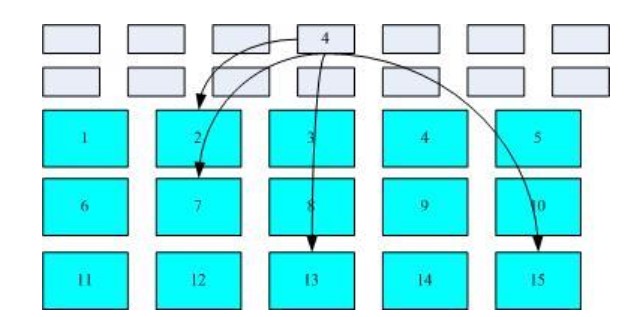
除此之外还包括：

* superblock：记录文件系统的整体信息，包括 inode 和 block 的总量、使用量、剩余量，以及文件系统的格式与相关信息等；
* block bitmap：记录 block 是否被使用的位域。就是为了方便文件删除。

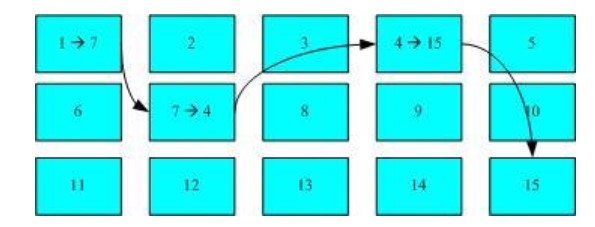


## 文件读取

对于 Ext2 文件系统，当要读取一个文件的内容时，先在 inode 中去查找文件内容所在的所有 block，然后把所有 block 的内容读出来。



而对于 FAT 文件系统（windows常用FAT32、NTFS；linux用ext系列），它没有 inode，每个 block 中存储着下一个 block 的编号。



不管是哪种文件系统，不过特别是FAT，对应文件的内容插入是很方便的。inode中对block顺序更改一下也不是很麻烦的一件事情。FAT是非日志文件系统，如果不用磁盘阵列，掉电可能就坏了（U盘）。

UFS文件系统：一部分直接索引到数据，一部分一级索引，一部分二级索引，一部分三级...

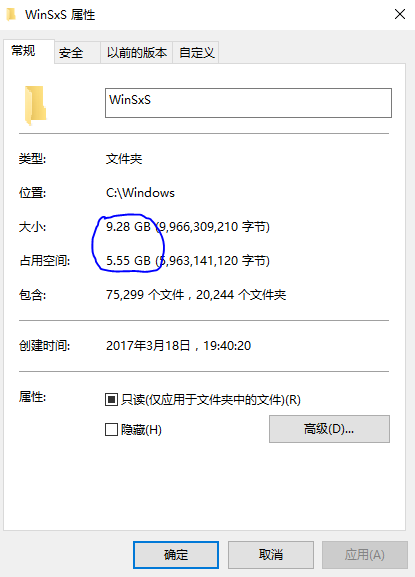
删除文件只是删除了文件系统，磁盘上的数据可能还在。

因为文件按照block占用空间，所以windows文件属性的大小中有一个逻辑大小、物理大小。

如果写一个程序，对文件进行fseek来实现map的话，我估计实际占用很大的空间，linux呢？

有的文件系统能支持的挂载存储有的不是无限的。

内存中为了减少分配次数带来的性能消耗、磁盘上为了顺序访问block，都会分配的大。

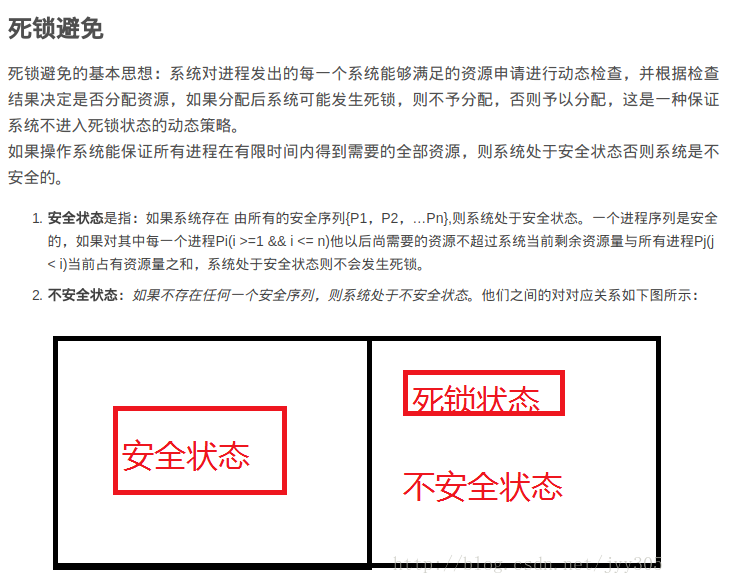


belady Anomaly：增加了缓存容量，但是缺页命中率反而下降了的现象。如果内存中页数更小的集合是内存页数更大的集合的子集，这个算法被称为stack algorithm。可以证明stack algorithm(如LRU)不会出现belady现象，FIFO会出现。

## 死锁产生的四个必要条件

* 互斥条件：有限的每个资源是独占的且排他使用，进程互斥使用资源，即任意时刻一个资源只能给一个进程使用，其他进程若申请一个资源，而该资源被另一进程占有时，则申请者等待直到资源被占有者释放。
* 不可剥夺条件：进程所获得的资源在未使用完毕之前，不被其他进程强行剥夺，而只能由获得该资源的进程资源释放。
* 请求和保持条件：进程每次申请它所需要的一部分资源，在申请新的资源的同时，继续占用已分配到的资源。 （哲学家问题请求到所有的再请求 破坏这个）
* 循环等待条件：在发生死锁时必然存在一个进程等待队列{P1,P2,…,Pn},其中P1等待P2占有的资源，P2等待P3占有的资源，…，Pn等待P1占有的资源，形成一个进程等待环路，环路中每一个进程所占有的资源同时被另一个申请，也就是前一个进程占有后一个进程所深情地资源。 （资源编码按序请求 破坏这个）

循环等待与互斥 构成一个资源有限的图，这个图中，单位不会主动释放资源并继续请求新的资源，同时系统不能强行剥夺资源。



还有极度不适用于大数据、云计算等的死锁解决

还有应用层面上的哲学家获取所有资源才开始执行、资源编号等解决办法。

解决办法（破坏条件）：

* 1. 扩容，给予足够多的资源
  2. 进程推荐顺序不合理 （比如按照编号申请资源；合理申请资源能释放掉不要占死，可以将资源拿全之后再执行，如果不是一次性调用拿全的话，还要考虑资源拿的顺序）
  3. 资源剥夺，甚至撤销进程重新运行任务

死锁会形成一个资源环

操作系统课最后提一种方法：划了一条安全边界，提前预判是否会发生死锁

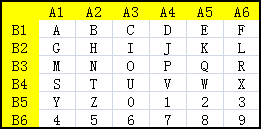
堆的内存不是连续的,只是逻辑上是连续的,但是在物理内存上是离散的,通过链表将已使用内存与空闲内存维护起来，因为会不断分配与释放内存，于是就会导致很多的堆碎片，这样之后就导致堆空间基本越来越少，所以每一次释放空间之后，会将链表遍历过去，如边上有空闲内存就会合并空闲内存，看看是不是可以合并空闲内存。默认堆的空间不受限制，栈的空间有8M左右。

如果你每个程序都只要一小会，那么看起来的多进程没准还是一两个核心ok掉。

 Intel的x86处理器是通过Ring级别来进行访问控制的，级别共分4层，从Ring0到Ring3（后面简称R0、R1、R2、R3）。R0层拥有最高的权限，R3层拥有最低的权限。按照Intel原有的构想，应用程序工作在R3层，只能访问R3层的数据；操作系统工作在R0层，可以访问所有层的数据；而其他驱动程序位于R1、R2层。但现在的OS，包括Windows和Linux都没有采用4层权限，而只是使用2层——R0层和R3层，分别来存放操作系统数据和应用程序数据，从而导致一旦驱动加载了，就运行在R0层，就拥有了和操作系统同样的权限，可以做任何事情，而所谓的rootkit也就随之而生了。特权指令在R0层运行或者在R3限制运行。

键位冲突：

（1）因为芯片的引脚不多（低级的更少），不同引脚的电信号组合成一个按键被芯片识别，所以会有多键冲突。普通的104键键盘是应用16×8的矩阵，来覆盖所有按键。只需要24个数据引脚，可以认为是两层for循环遍历矩阵来识别对应的按键。



由于G键被按下，A1和B2是接通的，  
由于H键被按下，B2和A2是接通的，  
由于B键被按下，A2和B1也是接通的！  
也就是说，现在的电路中，A1和B1将导致按键A被识别！

键盘按下时连同电路，电流一次性点亮了6扇们，用二/三极管保证电流不会从A1->B1，只有真的按下键才行。

（2）USB串口键盘只能做到6键无冲突（PS2接口倒没问题）这里所指的6键，是除去Ctrl、Shift、Alt、Win之外的键，同时按下任意6个都不会有冲突，但第7个键按下就没有反应——取千6后6或者怎么怎么。PS/2协议下，键盘是每次发生按键/抬键动作，都会发送数据信号给主机，如果按住一个键不放，则会不停地向主机循环发送通码，直到抬起按键发送断码，根据10-20kHz的工作频率规范，每位数据的传输时间大约是40-80微秒，加上中间的保留延迟，每个字节会占用0.5-1ms的传输时间。USB协议下，键盘会以某个固定的回报率（每秒125-1000次），定期向主机发送当前按键的状态，每次发送8个字节，这8个字节的具体内容则是：第一个字节：8位分别表示左右的Ctrl、Shift、Alt、Win各自是否被按下。这8个键统称为【modifer key】，因为规范已经事先定义好每一位的含义，从而得以能够只用一个字节就表示8个键的状态。第二个字节：保留（无用）其余6个字节：当前正按下的6个【普通按键】（如果按了7个以上，根据键盘主控芯片内置的程序，可能取最先按的6个，也可能取最后按的6个）。

USB HID协议本身就允许使用自定义的HID Report，只要你的HID Report Descriptor写正确了想多少键无冲突都行。市面上卖的全键无冲突的USB键盘都是用的这个方法，这个办法代价最小，代码改动最小。实现全键无冲有多种方法，如果是通过正确修改USB报告描述符实现的全键无冲，则无需安装特殊的驱动，Windows默认的驱动已经支持。但如果是其他方式实现的全键无冲，可能需要安装特定驱动。一些非常古老的主板的BIOS不支持解析自定义HID描述符，会导致在BIOS阶段键盘无法使用。

GHO镜像文件只是某一个硬盘/硬盘分区的刻录，包含了已安装软件与数据，用于还原。ISO文件则是官方提供的纯镜像，用于安装。安装系统推荐使用？？？进行刻录。

网卡要么开热点要么连WIFI，但是曾经我意外做过同时。