时间片的测量：

#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，经过判断是因为那句大于1的if判断，进程因为时间片到而离开到之后立即装入所需要的时间大部分是0-2毫秒（要么0要么1要么2，其实主要应该是个接近1的小数，C++做不到再精确了，一旦出现1，那么得到的测量就是几个时间片的时间和）

要知道装入时间（时间片与时间片之间时间）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。

#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。

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

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、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

轻量级的xcfe。

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

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），通过和Intel合作开发 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。

LRU是最近最少使用页面置换算法(Least Recently Used),也就是首先淘汰最长时间未被使用的页面!

LFU是最近最不常用页面置换算法(Least Frequently Used),也就是淘汰一定时期内被访问次数最少的页!

操作系统控制文件的删除就像是智能指针一样，一旦引用没有了就会删除，所以在文件结构中存在对引用的计数。文件的存储实际上就是名字到存储结构体最后再到实际存储硬盘块就好。

堆文件是最简单的文件结构。一个堆文件中的数据在文件页中以任意的顺序排列。

堆文件的格式是libpcap的一种。这种格式中，包含了被捕捉到的包的二进制数据，并且，这种格式是许多网络工具所使用的一种标准，这些工具包括WinDump，Etheral和Snort。THE PILE: Least complicated form of file organization. Data are collected in the order they arrive. Each record consist of one burst of data.Purpose is simply to accumalate the mass of data and save it. Record access is by exhaustive search.

CPU调度、资源分配的基本单位——进程，线程是占用核心的基本单位。

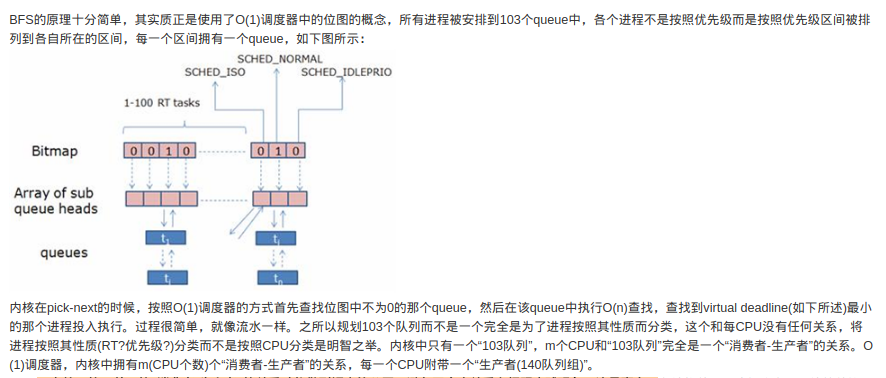
操作系统可以不要内核，后果就是你玩游戏的时候就不能开迅雷了。内核的一个作用就是实现任务调度，让计算机的资源都被均衡的调度。没有内核的后果就是不能同时处理多个任务。

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，该调度器引入了每CPU的优先级队列组，每组队列包含active和expire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。



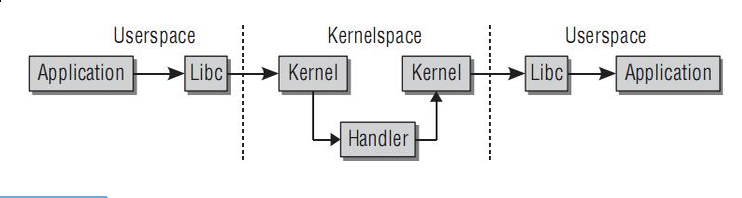
其实内存分配一般有三个原则，最快、最优和最差。最快比较好理解，就是寻找到合适的节点就立即分配内存，我们在前面一篇博客采用的就是这个方法。最优呢，就是寻找可以满足当前内存分配的最小节点，这样不会有很大的浪费，但是有可能会产生碎片节点。最后一种就是最差分配算法，说是最差效果未必最差。因为在大的内存分配的时候至少不会很快产生内存碎片，对整个系统的稳定来说有可能是好事。

文件write操作(从前FUSE没有出来之前，Linux的文件系统完全工作在内核态)、对网络数据的send操作 都是系统调用，也就是会触发ring0

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，这个数字事实上是系统调用函数指针的偏移。

      当我们运行一个系统调用时，运行时库通过查找这个表来决定对应的函数代码，即系统调用号，然后存入到寄存器中，通常为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。

根据中断源的不同，可以把中断分为硬件中断和软件中断两大类，而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。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。外部中断一般就是外部IO设备因为IO读写需要涉及到的中断。软件中断属于不可屏蔽中断，因为产生在CPU内部，是不能够避免的中断，除了单步中断都不可屏蔽。屏蔽电路就在中断信号的发送传送过程中。

个人最新的理解：软中断的产生有两种情况：（1）程序内部触发软中断（2）硬中断触发软中断，第一种情况的软中断我记得内核里面有一个函数可以关闭所有的软中断，在软件层面上直接截住了，这倒也算是一种屏蔽；第二种情况打印机的例子也只能是在软中断屏蔽了，软中断被登记之后进入工作队列。红色部分表示概念冲突部分，最新理解可以解决。

屏蔽中断并不是取消中断，只是在一段时间内不希望中断过来打扰。中断屏蔽电路应该在CPU内部。

软中断的产生有两种情况：（1）程序内部触发软中断（2）硬中断触发软中断，第一种情况的软中断我记得内核里面有一个函数可以关闭所有的软中断，在软件层面上直接截住了，这倒也算是一种屏蔽；第二种情况打印机的例子也只能是在软中断屏蔽了，软中断被登记之后进入工作队列。软中断可以被打断，具体见下面一幅图片。

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。

中断是分为上半部分与下半部分，硬件中断是中断的上半部分（功能用于登记中断，不允许被打断，屏蔽所有中断，这时候软中断还没有被触发），上半部分执行过程中屏蔽所有其他中断，下半部分执行的任务是执行与中断处理密切相关，但中断处理程序本身不执行的工作（下半部分在中断返回之前）。和上半部分只能通过中断处理程序实现不同，下半部分可以通过多种机制实现（软中断、tasklet、工作队列）。

一般来说，软中断是由[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8" \t "/home/vega/Documents\\x/_blank)机制的触发事件引起的（例如进程运行超时），但是不可忽视有大量的软中断也是由于和硬件有关的中断引起的，最后软中断都会传送给内核而不是直接与处理器沟通，内核负责与处理器进行进一步沟通，例如当打印机端口产生一个[硬件中断](https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6%E4%B8%AD%E6%96%AD" \t "/home/vega/Documents\\x/_blank)时，会通知和硬件相关的硬中断，硬中断就会产生一个软中断并送到[操作系统内核](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E5%86%85%E6%A0%B8" \t "/home/vega/Documents\\x/_blank)里，这样内核就会根据这个软中断唤醒睡眠在打印机任务队列中的处理进程。

硬中断触发之后会屏蔽同一类硬中断。

<https://blog.csdn.net/godleading/article/details/52971179> 这篇文章中说的软中断是可以被打断的，那么软中断的发生位置以及优先级问题？之前见到过一个函数用于关闭软中断。

Tasklet机制：Linux中的中断延迟是处理机制。机制流程：当[linux](https://baike.baidu.com/item/linux" \t "/home/vega/Documents\\x/_blank)接收到硬件中断之后，通过tasklet函数来设定软中断被执行的优先程度从而导致软中断处理函数被优先执行的差异性。那么中断所有请求都是存在工作队列中。

软中断用轮询的方式处理。假如正好是最后一种中断，则必须循环完所有的中断类型，才能最终执行对应的处理函数。显然当年开发人员为了保证轮询的效率，于是限制中断个数为32个。为了提高中断处理数量，顺道改进处理效率，于是产生了tasklet机制。

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。

软中断与硬件中断的区别：

硬中断一般是由于外部设别对CPU产生的中断信号，这时候中断信号是可屏蔽的；时钟中断是不可屏蔽信号。

软中断由于程序产生，使用int汇编指令，严格来讲只有3号中断，但是不少教材认为所有的int中断都是软中断。

硬中断的中断号是由中断控制器提供的，软中断的中断号由指令直接指出，无需使用中断控制器。

上半部--》硬中断：完成紧急但很快完成的事

下半部：分为软中断、tasklet（当然它也是一种软中断）、工作队列（当然它也是一种软中断） 作用：处理不紧急但比较耗时间的事

linux的驱动程序分两个部分实现：top-half和bottom-half。top-half在运行时，不能被其他任何中断再次中断，也不能被其他进程中断，它通过对CPU内的中断屏蔽置位实现，而bottom-half则只对top-half开中断。

countdownlatch、semaphore 是java中的共享锁，共享锁只是保证了能够多个程序同时占用临界区资源。读共享锁与写锁的排斥要靠自己代码中分开来使用读锁与写锁实现。并不是共享锁自动会排斥其他锁。

## **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。

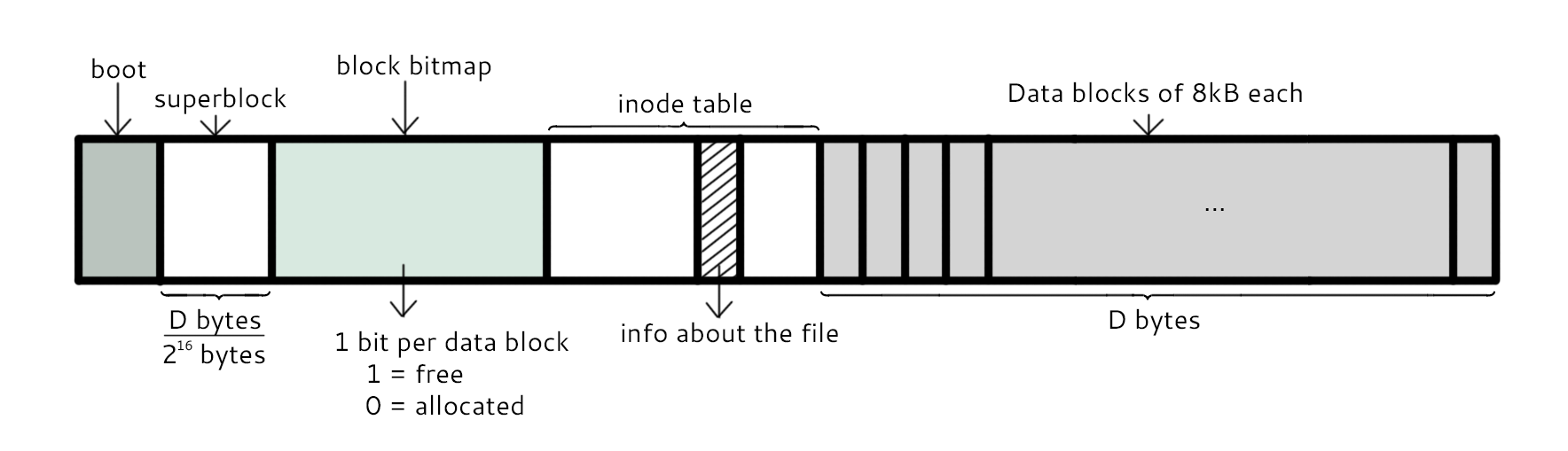
## **文件系统组成**

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。

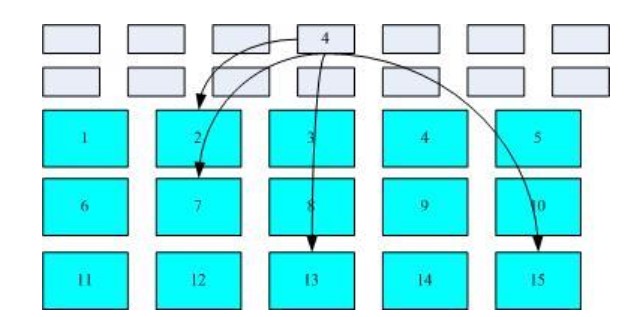
除此之外还包括：

* 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。

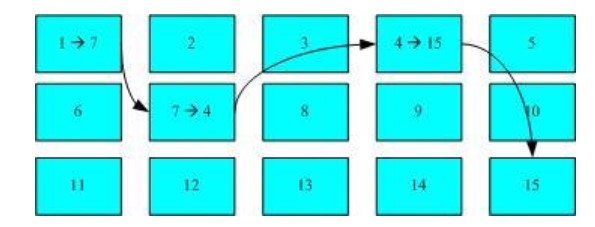


## **文件读取**

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，它没有 inode，每个 block 中存储着下一个 block 的编号。



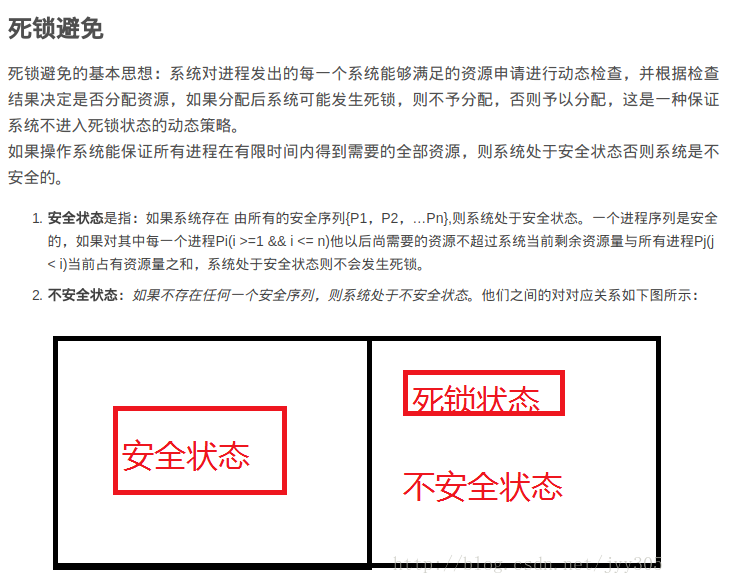
不管是哪种文件系统，不过特别是FAT，对应文件的内容插入是很方便的。

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：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）  
  以上给出了导致死锁的四个必要条件，只要系统发生死锁则以上四个条件至少有一个成立。事实上****循环等待****的成立蕴含了前三个条件的成立，似乎没有必要列出然而考虑这些条件对死锁的预防是有利的，因为可以通过破坏四个条件中的任何一个来预防死锁的发生。

互不请循



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。

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。