[内核](http://www.so.com/s?q=%E5%86%85%E6%A0%B8&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)是[操作系统](http://www.so.com/s?q=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)的[内部](http://www.so.com/s?q=%E5%86%85%E9%83%A8&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)[核心](http://www.so.com/s?q=%E6%A0%B8%E5%BF%83&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)[程序](http://www.so.com/s?q=%E7%A8%8B%E5%BA%8F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)，它向[外部](http://www.so.com/s?q=%E5%A4%96%E9%83%A8&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)提供了对计算机设备的核心管理调用。我们将操作系统的[代码](http://www.so.com/s?q=%E4%BB%A3%E7%A0%81&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)分成2部分。内核所在的[地址空间](http://www.so.com/s?q=%E5%9C%B0%E5%9D%80%E7%A9%BA%E9%97%B4&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)称作[内核空间](http://www.so.com/s?q=%E5%86%85%E6%A0%B8%E7%A9%BA%E9%97%B4&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。而在内核以外的统称为外部管理程序，它们大部分是对外围设备的管理和[界面](http://www.so.com/s?q=%E7%95%8C%E9%9D%A2&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)操作。外部管理程序与用户进程所占据的地址空间称为[外部空间](http://www.so.com/s?q=%E5%A4%96%E9%83%A8%E7%A9%BA%E9%97%B4&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。通常，一个程序会跨越两个[空间](http://www.so.com/s?q=%E7%A9%BA%E9%97%B4&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。当执行到内河空间的一段代码时，我们称程序处于内核态，而当程序执行到外部空间代码时，我们称程序处于用户态。 从UNIX起，人们开始用[高级语言](http://www.so.com/s?q=%E9%AB%98%E7%BA%A7%E8%AF%AD%E8%A8%80&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)(UNIX上最具有代表性的就是UNIX的系统级语言C语言)编写内核代码，使得内核具有良好的扩展性。单一内核(monolithic kernel)是当时操作系统的[主流](http://www.so.com/s?q=%E4%B8%BB%E6%B5%81&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)，操作系统中所有的系统相关功能都被封装在内核中，它们与外部程序处于不同的[内存地址](http://www.so.com/s?q=%E5%86%85%E5%AD%98%E5%9C%B0%E5%9D%80&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)空间中，并通过各种[方式](http://www.so.com/s?q=%E6%96%B9%E5%BC%8F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)(在Intel IA-32体系中采用386[保护模式](http://www.so.com/s?q=%E4%BF%9D%E6%8A%A4%E6%A8%A1%E5%BC%8F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank))防止 外部程序直接访问内核[结构](http://www.so.com/s?q=%E7%BB%93%E6%9E%84&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。程序只有通过一套称作[系统调用](http://www.so.com/s?q=%E7%B3%BB%E7%BB%9F%E8%B0%83%E7%94%A8&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)(system call)的界面访问内核结构。近些年来，[微内核](http://www.so.com/s?q=%E5%BE%AE%E5%86%85%E6%A0%B8&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)(micro kernel)结构逐渐流行起来，成为操作系统的主要潮流。1986年，Tanenbaum提出Mach kernel，而后，他的minix和GNU的Hurd操作系统更是微内核系统的典范。 在微内核结构中，操作系统的内核只需要提供最基本、最核心的一部分操作(比如创建和删除任务、[内存管理](http://www.so.com/s?q=%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)、中断管理等)即可，而其他的管理程序(如[文件系统](http://www.so.com/s?q=%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)、[网络协议](http://www.so.com/s?q=%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)栈等)则尽可能的放在内核之外。这些外部程序可以独立运行，并对外部[用户程序](http://www.so.com/s?q=%E7%94%A8%E6%88%B7%E7%A8%8B%E5%BA%8F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)提供操作系统服务，服务之间使用进程间通信机制(IPC)进行交互，只在需要内核的协助时，才通过一套[接口](http://www.so.com/s?q=%E6%8E%A5%E5%8F%A3&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)对内核发出调用请求。 微内核系统的[优点](http://www.so.com/s?q=%E4%BC%98%E7%82%B9&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)时操作系统具有良好的灵活性。它使得操作系统内部结构简单清晰。程序代码的维护非常之方便。但是也有不足之处。微内核系统由于[核心态](http://www.so.com/s?q=%E6%A0%B8%E5%BF%83%E6%80%81&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)只实现了最基本的系统操作，这样内核以外的外部程序之间由于独立运行使得系统难以进行良好的[整体](http://www.so.com/s?q=%E6%95%B4%E4%BD%93&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)优化。另外，进程间互相通信的开销也较单一内核系统要大许多。从整体上看，在当前的[硬件](http://www.so.com/s?q=%E7%A1%AC%E4%BB%B6&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)条件下，微内核在效率上的损失小于其在结构上获得的收益，故而选取微内核成为操作系统的一大潮流。 然而，Linux系统却恰恰使用了单一内核结构。这是由于Linux是一个[实用主义](http://www.so.com/s?q=%E5%AE%9E%E7%94%A8%E4%B8%BB%E4%B9%89&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)的操作系统。Linux Tovarlds以代码执行效率为自己操作系统的第一要务，并没有进行过一个系统的设计工作，而是任由Linux在使用中不断发展。在这样的发展过程中，参与Linux开发的程序员大多为世界各地的黑客们。比起结构的清晰，他们更加注重功能的强大和高效的代码。于是，他们将大量的精力放在优化代码上，而这样的全局性优化必然以丧失结构精简为代价，导致Linux中的每个[部件](http://www.so.com/s?q=%E9%83%A8%E4%BB%B6&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)都不能轻易被拆除。否则必然破坏整体效率。 虽然Linux是单一内核体系，但是它与[传统](http://www.so.com/s?q=%E4%BC%A0%E7%BB%9F&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)的单一内核UNIX操作系统不同。在普通的单一内核系统中，所有的内核代码都是被[静态](http://www.so.com/s?q=%E9%9D%99%E6%80%81&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)编译联入的，而在Linux中，可以动态装入和卸载内河中的部分代码。Linux将这些[代码段](http://www.so.com/s?q=%E4%BB%A3%E7%A0%81%E6%AE%B5&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)称为[模块](http://www.so.com/s?q=%E6%A8%A1%E5%9D%97&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。(module)，并对模块给予了强有力的支持。在Linux中，可以在需要时自动装入和卸载模块。 Linux不支持用户态[线程](http://www.so.com/s?q=%E7%BA%BF%E7%A8%8B&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。在用户态中，Linux认为线程就是共享[上下文](http://www.so.com/s?q=%E4%B8%8A%E4%B8%8B%E6%96%87&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)(Context)的进程。Linux通过LWP(light weight thread)的机制来实现用户态线程的[概念](http://www.so.com/s?q=%E6%A6%82%E5%BF%B5&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。通过系统调用clone()创建新的线程。 Linux的内核为非抢占式的。即，Linux不能通过改变优先权来影响内核当前的执行流程。因此，Linux在实现实时操作时就[有问题](http://www.so.com/s?q=%E6%9C%89%E9%97%AE%E9%A2%98&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)。Linux并不是一个“硬”实时操作系统。 在Linux内核中，包括了进程管理(process management)、[定时器](http://www.so.com/s?q=%E5%AE%9A%E6%97%B6%E5%99%A8&ie=utf-8&src=internal_wenda_recommend_textn" \t "https://wenda.so.com/q/_blank)(timer)、中断管理(interrupt management)、内存管理(memory management)、模

这个程序编译通过，但是问什么程序运行会出错？ //p指向的是一块常量数据区，不能修改其值！！！！

因为s是一个数组，分配的是一块栈上的内存，而上面只是一个指针，没申请内存！s在栈，“abcdef”在数据区，p在栈，“abcdef”在数据区

char \*p="abcdef"; //"abcdef" 静态存储区，是常量区，不可改, p[2]企图通过p这个指针改变常量区的内容

char s[]="abcdef";//"abcdef" 静态存储区, 你赋值给s[]这个变量后，改变s[2]，所以不出错。

可以简单认为数据区是不可写的,栈区是可写的   
  
  
是常量数据区不可写！！！！！！

char s[]="abcdef";//这是告诉编译器，要用一个字符数组，里面放上“abcdef”这个内容（编译器开一个空间放上字符数组的内容）   
char \*p="abcdef"; //这个是告诉编译器，要用一个指针（编译器开一个空间放指针），让指针指向放着“abcdef”内容的地方，eoh，这时候没为“abcdef”开空间，那编译器就认为这个是常量字符串了，自己把它放常量区里，让指针指向常量区的这个串。既然是常量区的东西，那以后就不能改了，尽管p能指向并读取，但不能改写

首先，讨论的堆和栈指的是内存中的“堆区”和“栈区”，OC语言是C语言的超集，所以先了解C语言的内存模型的内存管理会有很大的帮助。C语言的内存模型分为5个区：栈区、堆区、静态区、常量区、代码区。每个区存储的内容如下：

1、栈区：存放函数的参数值、局部变量等，由编译器自动分配和释放，通常在函数执行完后就释放了，其操作方式类似于数据结构中的栈。栈内存分配运算内置于CPU的指令集，效率很高，但是分配的内存量有限，比如iOS中栈区的大小是2M。

2、堆区：就是通过new、malloc、realloc分配的内存块，编译器不会负责它们的释放工作，需要用程序区释放。分配方式类似于数据结构中的链表。在iOS开发中所说的“内存泄漏”说的就是堆区的内存。

3、静态区：全局变量和静态变量（在iOS中就是用static修饰的局部变量或者是全局全局变量）的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后，由系统释放。

4、常量区：常量存储在这里，不允许修改。

5、代码区：存放函数体的二进制代码。

堆和栈的区别：

1、堆空间的内存是动态分配的，一般存放对象，并且需要手动释放内存。当然，iOS引入了ARC（自动引用计数管理技术）之后，程序员就不需要用代码管理对象的内存了，之前MRC（手动管理内存）的时候，程序员需要手动release对象。另外，ARC只是一种中间层的技术，虽然在ARC模式下，程序员不需要像之前那么麻烦管理内存，但是需要遵循ARC技术的规范操作，比如使用属性限定符weak、strong、assigen等。因此，如果程序员没有按ARC的规则并合理的使用这些属性限定符的话，同样是会造成内存泄漏的。

2、栈空间的内存是由系统自动分配，一般存放局部变量，比如对象的地址等值，不需要程序员对这块内存进行管理，比如，函数中的局部变量的作用范围（生命周期）就是在调完这个函数之后就结束了。这些在系统层面都已经限定住了，程序员只需要在这种约束下进行程序编程就好，根本就没有把这块内存的管理权给到程序员，肯定也就不存在让程序员管理一说。

从申请的大小方面讲：

栈空间比较小；

堆空间比较大。

从数据存储方面来说：

栈空间中一般存储基本数据类型，对象的地址；

堆空间一般存放对象本身，block的copy等。