# 1、C语言内存分布(BSS段、数据段、代码段、堆与栈）

**BSS段:**（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。

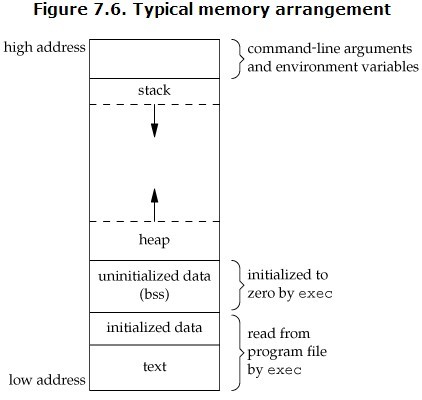
**数据段**：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

**代码段：**代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读, 某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

**堆（heap）**：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc/free等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）/释放的内存从堆中被剔除（堆被缩减）

**栈(stack)**：栈又称堆栈， 存放程序的局部变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，栈用来传递参数和返回值。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。

下图是APUE中的一个典型C内存空间分布图。**栈地址由高到低，堆地址由低到高。**

[](http://blog.chinaunix.net/attachment/201201/4/24708340_1325647816h2y2.jpg)

## 1.1内存基本构成

可编程内存在基本上分为这样的几大部分：静态存储区、堆区和栈区。他们的功能不同，对他们使用方式也就不同。

静态存储区：内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。它主要存放静态数据、全局数据和常量。

栈区：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

堆区：亦称动态内存分配。程序在运行的时候用malloc或new申请任意大小的内存，程序员自己负责在适当的时候用free或delete释放内存。动态内存的生存期可以由我们决定，如果我们不释放内存，程序将在最后才释放掉动态内存。 但是，良好的编程习惯是：如果某动态内存不再使用，需要将其释放掉，否则，我们认为发生了内存泄漏现象。

代码区：存放函数体的二进制代码

文字常量区 —常量字符串就是放在这里的。程序结束后由系统释放

函数指针指向Code区，是程序运行的指令代码，数据指针指向Data,Heap,Stack区，是程序依赖以运行的各种数据

在文件作用域声明inline函数默认为static存储类型，const常量默认为static存储，如果加上extern，则为外部存储类型。

## 1.2 三者之间的区别

我们通过代码段来看看对这样的三部分内存需要怎样的操作和不同，以及应该注意怎样的地方。

例一：静态存储区与栈区

char\* p = “Hello World1”;

char a[] = “Hello World2”;

p[2] = ‘A’;

a[2] = ‘A’;

char\* p1 = “Hello World1;”

这个程序是有错误的，错误发生在p[2] = ‘A’这行代码处，为什么呢，是变量p和变量数组a都存在于栈区的（任何临时变量都是处于栈区的，包括在main（）函数中定义的变量）。但是，数据“Hello World1”和数据“Hello World2”是存储于不同的区域的。

因为数据“Hello World2”存在于数组中，所以，此数据存储于栈区，对它修改是没有任何问题的。因为指针变量p仅仅能够存储某个存储空间的地址，数据“Hello World1”为字符串常量，所以存储在静态存储区。虽然通过p[2]可以访问到静态存储区中的第三个数据单元，即字符‘l’所在的存储的单元。但是因为数据“Hello World1”为字符串常量，不可以改变，所以在程序运行时，会报告内存错误。并且，如果此时对p和p1输出的时候会发现p和p1里面保存的地址是完全相同的。换句话说，在数据区只保留一份相同的数据

例二：栈区与堆区

char\* f1()

{

char\* p = NULL;

char a;

p = &a;

return p;

}

char\* f2()

{

char\* p = NULL:

p =(char\*) new char[4];

return p;

}

这两个函数都是将某个存储空间的地址返回，二者有何区别呢？f1()函数虽然返回的是一个存储空间，但是此空间为临时空间。也就是说，此空间只有短暂的生命周期，它的生命周期在函数f1()调用结束时，也就失去了它的生命价值，即：此空间被释放掉。所以，当调用f1()函数时，如果程序中有下面的语句：

char\* p ;

p = f1();

\*p = ‘a’;

此时，编译并不会报告错误，但是在程序运行时，会发生异常错误。因为，你对不应该操作的内存（即，已经释放掉的存储空间）进行了操作。但是，相比之下，f2()函数不会有任何问题。因为，new这个命令是在堆中申请存储空间，一旦申请成功，除非你将其delete或者程序终结，这块内存将一直存在。也可以这样理解，堆内存是共享单元，能够被多个函数共同访问。如果你需要有多个数据返回却苦无办法，堆内存将是一个很好的选择。但是一定要避免下面的事情发生：

void f()

{

…

char \* p;

p = (char\*)new char[100];

…

}

这个程序做了一件很无意义并且会带来很大危害的事情。因为，虽然申请了堆内存，p保存了堆内存的首地址。但是，此变量是临时变量，当函数调用结束时p变量消失。也就是说，再也没有变量存储这块堆内存的首地址，我们将永远无法再使用那块堆内存了。但是，这块堆内存却一直标识被你所使用（因为没有到程序结束，你也没有将其delete，所以这块堆内存一直被标识拥有者是当前您的程序），进而其他进程或程序无法使用。我们将这种不道德的“流氓行为”（我们不用，却也不让别人使用）称为内存泄漏。这是我们C++程序员的大忌！！请大家一定要避免这件事情的发生。

总之，对于堆区、栈区和静态存储区它们之间最大的不同在于，栈的生命周期很短暂。但是堆区和静态存储区的生命周期相当于与程序的生命同时存在（如果您不在程序运行中间将堆内存delete的话），我们将这种变量或数据成为全局变量或数据。但是，对于堆区的内存空间使用更加灵活，因为它允许你在不需要它的时候，随时将它释放掉，而静态存储区将一直存在于程序的整个生命周期中。

# 2、调试打印

用法1：

#define MYDEBUG

#ifdef MYDEBUG

#define DEBUG(arg...) {\

printf("[debug]:%s:%s:%d ---->",\_\_FILE\_\_,\_\_FUNCTION\_\_,\_\_LINE\_\_);\

printf(arg);\

fflush(stdout);\

}

#else

#define DEBUG(arg...) {}

#endif

用法2：

//#define dbg\_printf(f, a...) \

// do { \

// fprintf(stdout, "%s(%d): " f, \_\_func\_\_, \_\_LINE\_\_, ## a); \

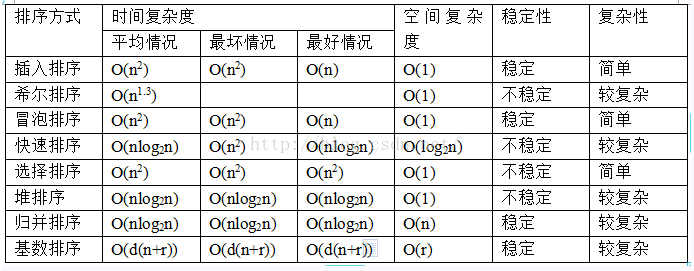
// }while (0)

#define dbg\_printf(f, a...) \

fprintf(stdout, "%s(%d): " f, \_\_func\_\_, \_\_LINE\_\_, ## a)

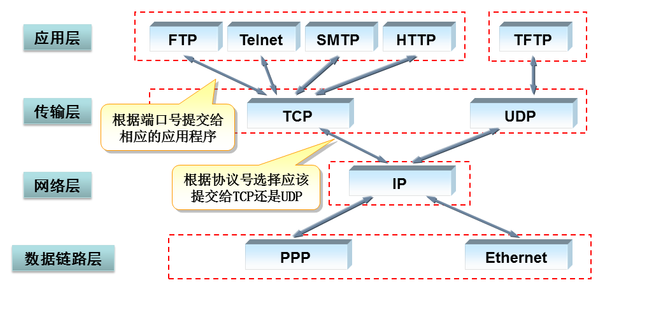
# 3、算法稳定性和复杂度

选择排序、快速排序、希尔排序、堆排序不是稳定的排序算法，

冒泡排序、插入排序、归并排序和基数排序是稳定的排序算法。  
 

冒泡排序是稳定的，算法时间复杂度是O(n ^2)。

# 4、TCP和UDP协议



|  |  |  |
| --- | --- | --- |
| 7 | **应用层** | 例如[HTTP](http://zh.wikipedia.org/wiki/HTTP)、[SMTP](http://zh.wikipedia.org/wiki/SMTP)、[SNMP](http://zh.wikipedia.org/wiki/SNMP)、[FTP](http://zh.wikipedia.org/wiki/FTP)、[Telnet](http://zh.wikipedia.org/wiki/Telnet)、[SIP](http://zh.wikipedia.org/wiki/SIP)、[SSH](http://zh.wikipedia.org/wiki/SSH)、[NFS](http://zh.wikipedia.org/wiki/NFS)、[RTSP](http://zh.wikipedia.org/wiki/RTSP)、[XMPP](http://zh.wikipedia.org/wiki/XMPP)、[Whois](http://zh.wikipedia.org/wiki/Whois)、[ENRP](http://zh.wikipedia.org/w/index.php?title=ENRP&action=edit&redlink=1) |
| 6 | **表示层** | 例如[XDR](http://zh.wikipedia.org/w/index.php?title=External_Data_Representation&action=edit&redlink=1)、[ASN.1](http://zh.wikipedia.org/w/index.php?title=Abstract_Syntax_Notation_1&action=edit&redlink=1)、[SMB](http://zh.wikipedia.org/w/index.php?title=Server_message_block&action=edit&redlink=1)、[AFP](http://zh.wikipedia.org/w/index.php?title=Apple_Filing_Protocol&action=edit&redlink=1)、[NCP](http://zh.wikipedia.org/w/index.php?title=NetWare_Core_Protocol&action=edit&redlink=1) |
| 5 | **会话层** | 例如[ASAP](http://zh.wikipedia.org/w/index.php?title=Aggregate_Server_Access_Protocol&action=edit&redlink=1)、[TLS](http://zh.wikipedia.org/wiki/Transport_Layer_Security)、[SSH](http://zh.wikipedia.org/wiki/SSH)、ISO 8327 / CCITT X.225、[RPC](http://zh.wikipedia.org/w/index.php?title=Remote_procedure_call&action=edit&redlink=1)、[NetBIOS](http://zh.wikipedia.org/w/index.php?title=NetBIOS&action=edit&redlink=1)、[ASP](http://zh.wikipedia.org/w/index.php?title=AppleTalk&action=edit&redlink=1)、[Winsock](http://zh.wikipedia.org/w/index.php?title=Winsock&action=edit&redlink=1)、[BSD sockets](http://zh.wikipedia.org/wiki/Berkeley_sockets) |
| 4 | **传输层** | 例如[TCP](http://zh.wikipedia.org/wiki/TCP)、[UDP](http://zh.wikipedia.org/wiki/User_Datagram_Protocol)、[RTP](http://zh.wikipedia.org/w/index.php?title=Real-time_Transport_Protocol&action=edit&redlink=1)、[SCTP](http://zh.wikipedia.org/w/index.php?title=Stream_Control_Transmission_Protocol&action=edit&redlink=1)、[SPX](http://zh.wikipedia.org/w/index.php?title=Sequenced_packet_exchange&action=edit&redlink=1)、[ATP](http://zh.wikipedia.org/w/index.php?title=AppleTalk&action=edit&redlink=1)、[IL](http://zh.wikipedia.org/w/index.php?title=IL_Protocol&action=edit&redlink=1) |
| 3 | **网络层** | 例如[IP](http://zh.wikipedia.org/wiki/%E7%BD%91%E9%99%85%E5%8D%8F%E8%AE%AE)、[ICMP](http://zh.wikipedia.org/wiki/ICMP)、[IGMP](http://zh.wikipedia.org/wiki/IGMP)、[IPX](http://zh.wikipedia.org/wiki/IPX)、[BGP](http://zh.wikipedia.org/wiki/BGP)、[OSPF](http://zh.wikipedia.org/wiki/OSPF)、[RIP](http://zh.wikipedia.org/wiki/RIP)、[IGRP](http://zh.wikipedia.org/wiki/IGRP)、[EIGRP](http://zh.wikipedia.org/wiki/EIGRP)、[ARP](http://zh.wikipedia.org/wiki/ARP)、[RARP](http://zh.wikipedia.org/wiki/RARP)、 [X.25](http://zh.wikipedia.org/wiki/X.25) |
| 2 | **数据链路层** | 例如[以太网](http://zh.wikipedia.org/wiki/%E4%BB%A5%E5%A4%AA%E7%BD%91)、[令牌环](http://zh.wikipedia.org/wiki/%E4%BB%A4%E7%89%8C%E7%8E%AF)、[HDLC](http://zh.wikipedia.org/wiki/HDLC)、[帧中继](http://zh.wikipedia.org/wiki/%E5%B8%A7%E4%B8%AD%E7%BB%A7)、[ISDN](http://zh.wikipedia.org/wiki/ISDN)、[ATM](http://zh.wikipedia.org/wiki/%E5%BC%82%E6%AD%A5%E4%BC%A0%E8%BE%93%E6%A8%A1%E5%BC%8F)、[IEEE 802.11](http://zh.wikipedia.org/wiki/IEEE_802.11)、[FDDI](http://zh.wikipedia.org/wiki/FDDI)、[PPP](http://zh.wikipedia.org/wiki/PPP) |
| 1 | **物理层** | 例如[线路](http://zh.wikipedia.org/w/index.php?title=%E7%BA%BF%E8%B7%AF&action=edit&redlink=1)、[无线电](http://zh.wikipedia.org/wiki/%E6%97%A0%E7%BA%BF%E7%94%B5)、[光纤](http://zh.wikipedia.org/wiki/%E5%85%89%E7%BA%A4)、[信鸽](http://zh.wikipedia.org/wiki/%E4%BF%A1%E9%B8%BD) |

TCP对应的协议：

（1） FTP：定义了文件传输协议，使用21端口。

（2） Telnet：一种用于远程登陆的端口，使用23端口，用户可以以自己的身份远程连接到计算机上，可提供基于DOS模式下的通信服务。

（3） SMTP：邮件传送协议，用于发送邮件。服务器开放的是25号端口。

（4） POP3：它是和SMTP对应，POP3用于接收邮件。POP3协议所用的是110端口。

（5）HTTP：是从Web服务器传输超文本到本地浏览器的传送协议。

UDP对应的协议：

（1） DNS：用于域名解析服务，将域名地址转换为IP地址。DNS用的是53号端口。

（2） SNMP：简单网络管理协议，使用161号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。

（3） TFTP(Trival File Tran敏感词er Protocal)，简单文件传输协议，该协议在熟知端口69上使用UDP服务。

# 5、7层网络模型

<https://www.cnblogs.com/gdayq/p/5797645.html>

**物理层**：利用传输介质为数据链路层提供屋里连接，实现比特流的透明传输。

**数据链路层**：负责建立和管理节点间的链路。通过各种控制协议，将有差错的物理信道变为无差错的、能可靠传输数据帧的数据链路。接受来自物理层的位流形式的数据，并封装成帧，传送到上一层；同样，也将来自上一层的数据帧，拆装为位流形式的数据转发到物理层；并且还负责处理接受端发回的确认帧的信息，以便提供可靠的数据传输。

**网络层**：是OSI参考模型中最复杂的一层，也是通信子网最高的一层，它在下两层的基础上向资源子网提供服务。通过路由算法，为报文或分组通过通信子网选择最适当的路径。该层控制数据链路层与物理层之间的信息转发，建立、维持与终止网络的连接。具体的说，数据链路层的数据在这一层被转换为数据包，然后通过路径选择、分段组合、顺序、进/出路由等控制，将信息从一个网络设备传送到另一个网络设备。

**传输层**：向高层屏蔽下层数据通信的具体细节，向用户提供可靠的、端到端的差错和流量控制，保证报文的正确传输。

**会话层：**是OSI参考模型的第五层，是用户应用程序和网络之间的接口。向两个实体的表示层提供建立和使用连接的方法。将不同实体之间的表示层的连接称为会话。因此会话层的任务就是组织和协调两个会话进程之间的通信，并对数据交换进行管理。

**应用层**是OSI参考模型的最高层，它是**计算机用户，以及各种应用程序和网络之间的接口**。

# 6、TCP三次握手与四次挥手（断开）

（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

  （2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack (number )=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

  （3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

 （1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

  （2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

 （3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

  （4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

**为什么连接的时候是三次握手，关闭的时候却是四次握手？**

答：

**“三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误”。全双工，单独关闭。**

因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

**为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？**

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。

# 7、进程与线程的区别

1. 一个程序至少有一个进程，一个进程至少有一个线程。

2. 线程的划分尺度小于进程，使得多线程程序的并发性高。

3. 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4. 每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5. 多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。

# 8、进程间通信

**管道**( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

**命名管道**FIFO：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

**信号量**( semophore )：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

**消息队列**( message queue ) ：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

**共享内存**( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

**套接字**( socket ) ： 套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机及其间的进程通信。

# 9、http和https的区别

HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议；http和https使用的是完全不同的连接方式用的端口也不一样,前者是80,后者是443；

# 10、存储器的层次结构

1、CPU寄存器：寄存器

2、主存（内存）：高速缓存、主存储器、磁盘缓存

3、辅存：固定磁盘，可移动存储介质

可执行存储器：寄存器、主存（掉电信息丢失）---属于操作系统存储管理负责分配、回收，以及提供存储层次间数据移动的管理机制。

# 11、UML设计模式

**创建型设计模式**：工厂模式，建造模式，原型模式，单例模式

**工厂模式**分为三种： 简单工厂， 工厂方法，抽象方法

**结构型设计模式**：适配器对象模式，桥接模式，组合模式，装饰模式，外观模式，享元模式，代理模式

**行为型审计模式**：命令模式，迭代器模式，责任链模式，中介者模式，备忘录模式，观察者模式，状态模式，策略模式，访问者模式

泛化（Generalization）, 实现（Realization），关联（Association)，聚合（Aggregation），组合(Composition)，依赖(Dependency)

【泛化关系】：是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为。例如：老虎是动物的一种，即有老虎的特性也有动物的共性。

【实现关系】：是一种类与接口的关系，表示类是接口所有特征和行为的实现.

【关联关系】：是一种拥有的关系，它使一个类知道另一个类的属性和方法；如：老师与学生，丈夫与妻子关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

【聚合关系】：是整体与部分的关系，且部分可以离开整体而单独存在。如车和轮胎是整体和部分的关系，轮胎离开车仍然可以存在。

聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。

【组合关系】：是整体与部分的关系，但部分不能离开整体而单独存在。如公司和部门是整体和部分的关系，没有公司就不存在部门。

组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期。

【依赖关系】：是一种使用的关系，即一个类的实现需要另一个类的协助，所以要尽量不使用双向的互相依赖.

【代码表现】：局部变量、方法的参数或者对静态方法的调用

**泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖**

# 12、操作系统原理

## 12.1 分段和分页

页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而一个系统只能有一种大小的页面。段的长度却不固定，决定于用户所编写的程序，通常由编辑程序在对源程序进行编辑时，根据信息的性质来划分。

　　分页的作业地址空间是一维的，即单一的线性空间，程序员只须利用一个记忆符，即可表示一地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

## 12.2 死锁的防止与避免

产生死锁的条件：**互斥访问、占有并等待、非抢占、循环等待**。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

死锁的处理策略：

死锁的**防止**：静态分配、层次分配。

死锁的**避免**：银行家算法。

死锁的检测和解除。

# 13、'\0'和' '有什么区别

'\0'用作字符串的结束符。它的ASCII数值就是0。

‘ ’是空格字符。它的ASCII数值是32。

# 14、共享内存和全局变量的区别

共享内存用于跨进程，全局变量没法跨进程。

MMU把各个进程的虚拟内存映射到不同的物理内存上，这样就保证了进程的虚拟内存是独立的。然而，物理内存往往远远少于各个进程的虚拟内存的总和。怎么办呢，通常的办法是把暂时不用的内存写到磁盘上去，要用的时候再加载回内存中来。一般会搞一个专门的分区保存内存数据，这就是所谓的交换分区。

共享内存设置为4KB。

# 15、client和server建立TCP连接的过程和函数

1、开始创建socket：

int socket(int domain, int type, int protocol)

domain(域) : AF\_UNIX type : SOCK\_STREAM/ SOCK\_DGRAM :

protocol : 0

2、获取到socket文件描述符之后,还要将其绑定一个文件上

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

sockfd : 传入sock的文件描述符

addr : 用sockaddr\_un表示

addrlen : 结构体长度

bind()函数把一个地址族中的特定地址赋给socket。

3、监听客户端的连接

int listen(int sockfd, int backlog);

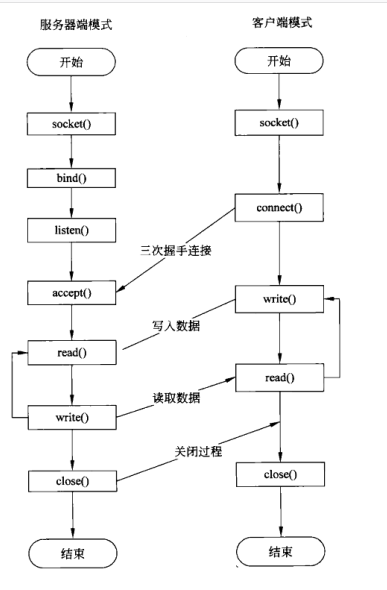
sockfd : 文件描述符

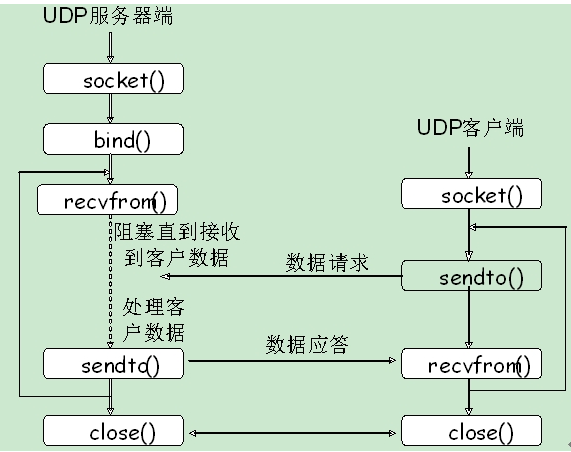
backlog : 连接队列的长度

4、接受客户端的连接

int accept(int socket, struct sockaddr \*restrict address, socklen\_t \*restrict address\_len);

UDP不存在客户端地址的问题，因此这里的addr和addrlen参数可以设置为NULL。





16、.c生成.o可执行文件的过程

**预处理，编译，汇编，链接。**

1、预处理过程（头文件的包涵，去掉注释，宏展开）—#include 预处理过程不做语法检查   
命令：gcc -E helloworld.c -o helloworld.i   
2、 编译：编译过程做语法检查 生成汇编语言   
命令：gcc -S helloworld.i -o helloworld.s   
3、汇编：将汇编语言生成对应的二进制数据   
命令：gcc -c helloworld.s -o helloworld.o   
4、链接：添加对应操作系统可以执行的链接，否则无法在系统下运行   
命令：gcc helloworld.o -o helloworld --------------------- 本文来自 sunzz93 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/sunzz93/article/details/77507980?utm\_source=copy

17、多线程建立的过程和函数

包括了线程的创建，销毁以及线程间的同步控制的一系列概念和实现。

只是单纯的同时写入同个变量是不会导致程序崩溃的，只可能会导致写入的数据不正确，如果32位系统写入的是32位变量，因为是原子操作，这种情况下也不会导致数据不正确。你所谓的崩溃，是因为读取到了不正确的数据，然后处理过程中导致的崩溃。

18、linux下查看进程的堆栈和线程

TOP命令，找到占用CPU最高的进程

第一种：pstack 进程ID

第二种，使用gdb 然后attach 进程ID，然后再使用命令 thread apply all bt

主要涉及gdb的使用。

19、gcc和g++区别

1.后缀为.c的，gcc把它当作是C程序，而g++当作是[c++程序](https://www.baidu.com/s?wd=c%2B%2B%E7%A8%8B%E5%BA%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)；后缀为.cpp的，两者都会认为是[c++程序](https://www.baidu.com/s?wd=c%2B%2B%E7%A8%8B%E5%BA%8F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)

2.编译阶段，g++会调用gcc，对于c++代码，两者是等价的，但是因为gcc命令不能自动和C＋＋程序使用的库联接，所以通常用g++来完成链接，为了统一起见，干脆编译/链接统统用g++了，这就给人一种错觉，好像cpp程序只能用g++似的。

编译可以用gcc/g++，而链接可以用g++或者gcc -lstdc++。因为gcc命令不能自动和C＋＋程序使用的库联接，所以通常使用g++来完成联接。但在编译阶段，g++会自动调用gcc，二者等价。

3、无论是gcc还是g++，用[extern](https://www.baidu.com/s?wd=extern&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao) ["c"](https://www.baidu.com/s?wd=%22c%22&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)时，都是以C的命名方式来为symbol命名，否则，都以c++方式命名。

20、i++在两个线程里边分别执行100次，能得到的最大值和最小值分别是多少？

**多核cpu，最小值2，最大值200，**

i++不是原子操作，也就是说，它不是单独一条指令，而是3条指令：

1、从内存中把i的值取出来放到CPU的寄存器中

2、CPU寄存器的值+1

3、把CPU寄存器的值写回内存

如果是单线程操作，i++毫无问题；但是在多核处理器上，用多线程来做i++会有什么问题呢？

我再仔细地重复一遍问题：进程有一个全局变量i，还有有两个线程。每个线程的功能，就是循环100次，执行i++。问线程代码全部执行完毕后，i的值是否一定是200？如果不是，它的最大最小值是多少？

=========分析=======

i++是由3条指令构成的运算操作，两个线程在i变量上共计需要执行100（次循环）\*3（条指令）\*2（个线程）=600条指令，这600条指令在某种排列下会导致最终i的值仅为2。

（下面是我复制过来的）

假设两个线程的执行步骤如下：

 1. 线程A执行第一次i++，取出内存中的i，值为0，存放到寄存器后执行加1，此时CPU1的寄存器中值为1，内存中为0；

 2. 线程B执行第一次i++，取出内存中的i，值为0，存放到寄存器后执行加1，此时CPU2的寄存器中值为1，内存中为0；

 3. 线程A继续执行完成第99次i++，并把值放回内存，此时CPU1中寄存器的值为99，内存中为99；

 4. 线程B继续执行第一次i++，将其值放回内存，此时CPU2中的寄存器值为1，内存中为1；

 5. 线程A执行第100次i++，将内存中的值取回CPU1的寄存器，并执行加1，此时CPU1的寄存器中的值为2，内存中为1；

 6. 线程B执行完所有操作，并将其放回内存，此时CPU2的寄存器值为100，内存中为100；

 7. 线程A执行100次操作的最后一部分，将CPU1中的寄存器值放回内存，内存中值为2；

**单核cpu，最小值100，最大值200**

两个线程分别记为线程1和线程2，i++相当于取出i的值，加1，再放回去  
第一种极端情况：每次线程一取出i的值后CPU时间切换到线程二，线程二也取出i的值，取到的值和线程一相等，线程二给i加一后放回去，线程一也将i加一后放回去，放回去的值也相等，相当于两个线程都执行一次i++操作，i的值只增加1，这样操作100次i的值为100  
第二种极端情况：线程一和线程二间隔操作，即线程一对i++操作完成，把已经加一的数据放回去之后线程二再操作，轮流进行，最后每个线程都对i加了100次，i的值为200

21、对于海量数据，用什么数据结构存储用户搜索的高频关键词比较合适

包括哈希、bitmap、Bloom filter、堆、mapreduce、trie树。

https://www.cnblogs.com/dong008259/archive/2012/02/17/2355551.html

22、两个进程互相独立，访问同一片共享内存

互斥量保存在共享内存中，在初始化该锁的时候，设置为进程间共享，这样两个进程连接到共享内存后，都可以获得这个互斥锁，因为已经设置了进程间共享，所以对锁的访问的冲突问题，系统已经解决了。

<https://blog.csdn.net/woyimibayi/article/details/80096275>

<http://www.cnblogs.com/my_life/articles/4538299.html>

23、阻塞、非阻塞、同步、异步的关系

同步和异步关注的是消息通信机制，阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。

同步和异步针对应用程序来，关注的是程序中间的协作关系；阻塞与非阻塞更关注的是单个进程的执行状态。

**同步：执行一个操作之后，等待结果，然后才继续执行后续的操作。**

**异步：执行一个操作后，可以去执行其他的操作，然后等待通知再回来执行刚才没执行完的操作。**

**阻塞：进程给CPU传达一个任务之后，一直等待CPU处理完成，然后才执行后面的操作。**

**非阻塞：进程给CPU传达任我后，继续处理后续的操作，隔断时间再来询问之前的操作是否完成。这样的过程其实也叫轮询。**

<https://www.zhihu.com/question/19732473>

<https://www.jianshu.com/p/486b0965c296>

24、网络I/O模型

网络IO的本质是socket的读取，socket在linux系统被抽象为流，IO可以理解为对流的操作。刚才说了，对于一次IO访问（以read举例），数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。所以说，当一个read操作发生时，它会经历两个阶段：

1、第一阶段：等待数据准备 (Waiting for the data to be ready)。

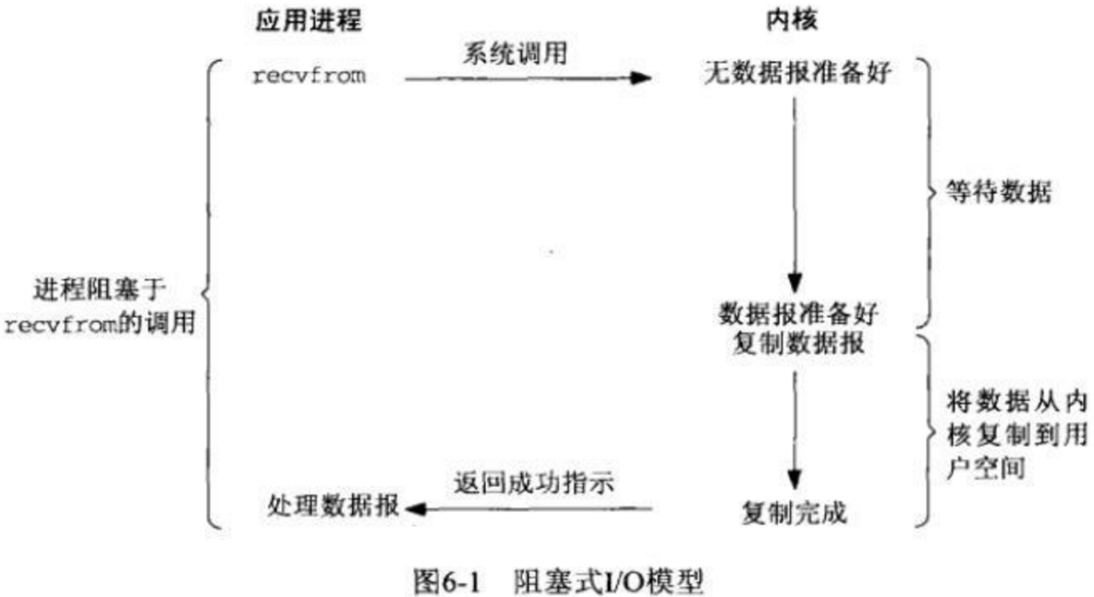
2、第二阶段：将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)。

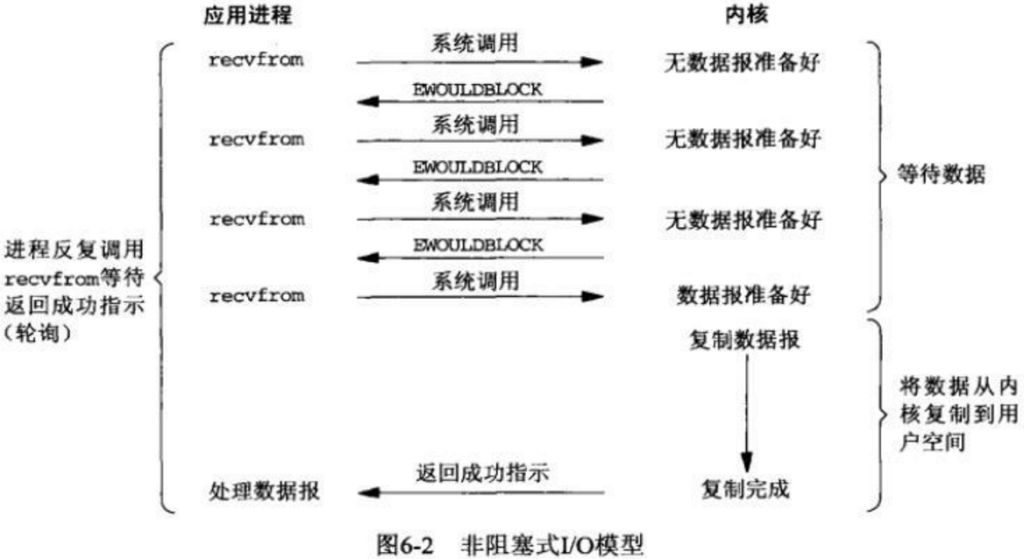
对于socket流而言，

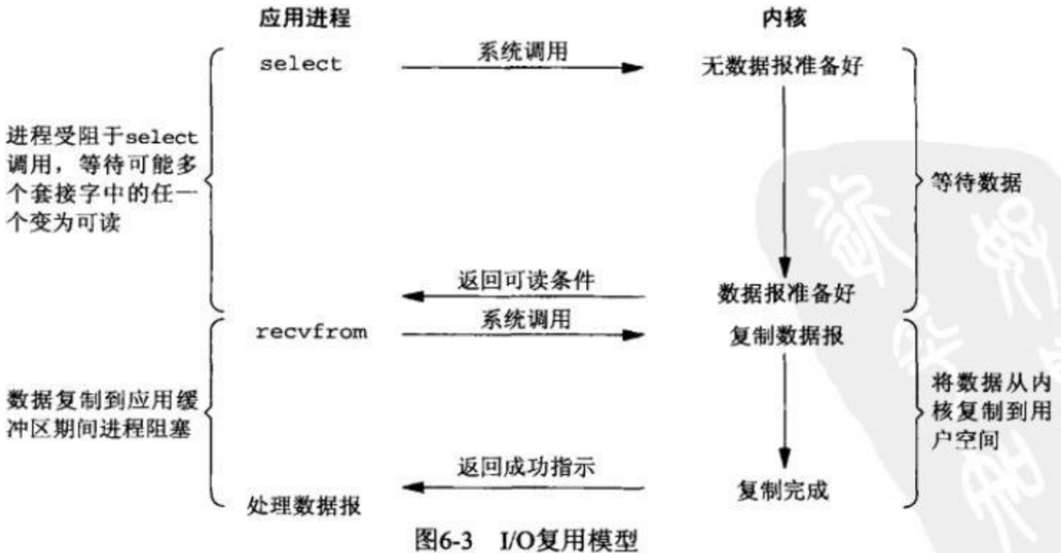
1、第一步：通常涉及等待网络上的数据分组到达，然后被复制到内核的某个缓冲区。

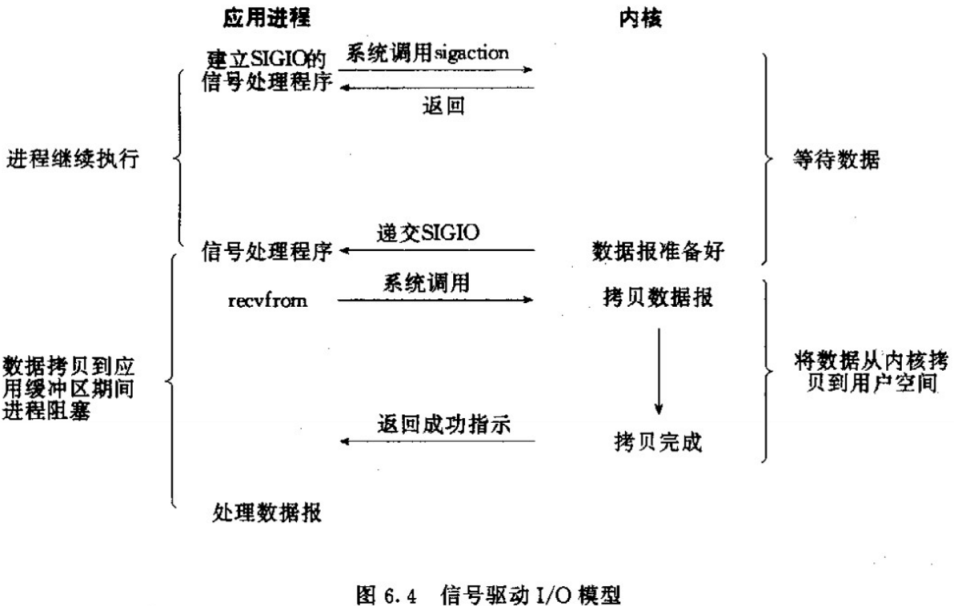
2、第二步：把数据从内核缓冲区复制到应用进程缓冲区。

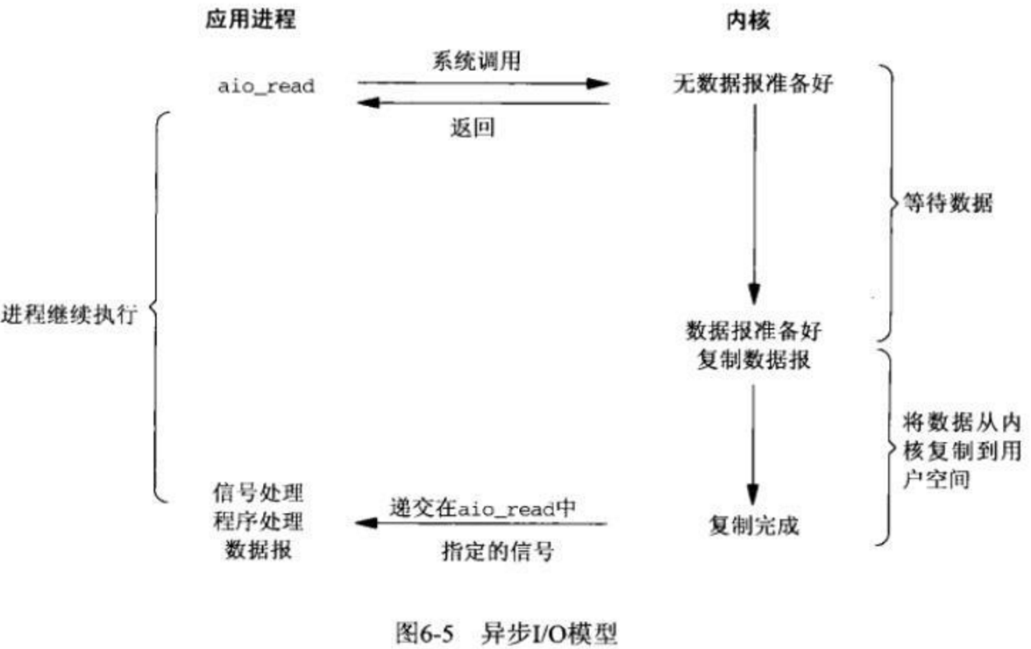
* **同步模型（synchronous IO）**
* 阻塞IO（bloking IO）
* 非阻塞IO（non-blocking IO）
* 多路复用IO（multiplexing IO）
* 信号驱动式IO（signal-driven IO）
* **异步IO（asynchronous IO）**

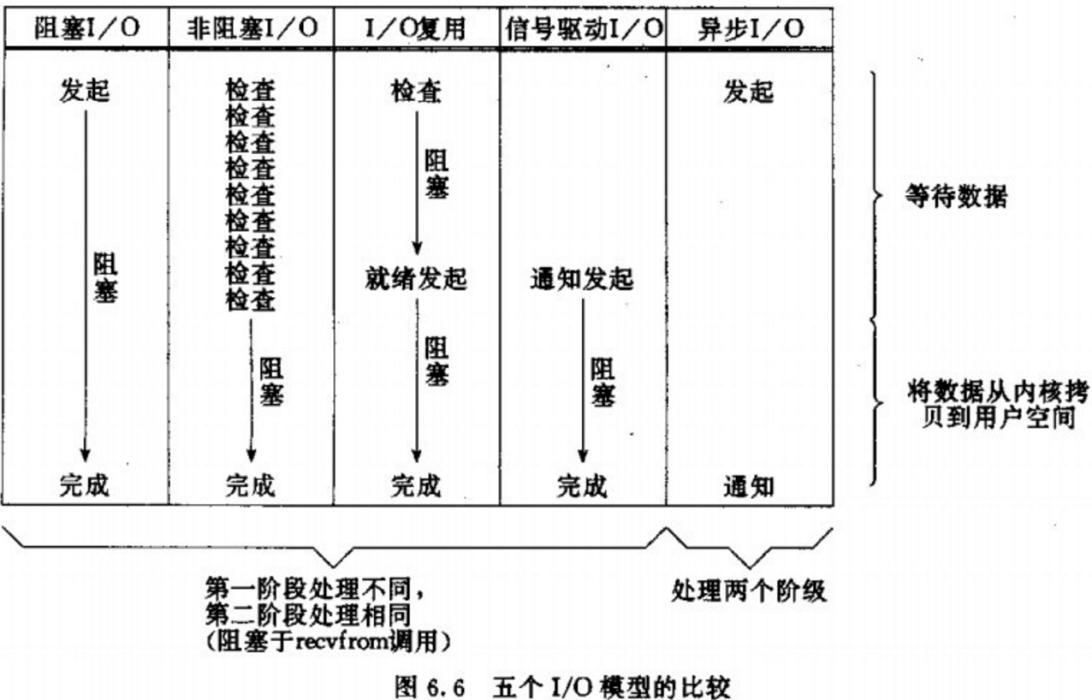












25、单例模式（C++）

<https://www.cnblogs.com/cxjchen/p/3148582.html>

26、 0长度数组

GNU C 的0长度数组, 也叫变长数组, 柔性数组就是这样一个扩展. 对于0长数组的这个特点，很容易构造出变成结构体，如缓冲区，数据包等等：

// 0长度数组

struct zero\_buffer

{

int len;

char data[0];

} \_\_attribute((packed)); // 让编译器取消结构在编译过程中的优化对齐,按照实际占用字节数进行对齐，这样子两边都需要使用 \_\_attribute\_\_ ((packed))取消优化对齐，就不会出现对齐的错位现象。

这样的变长数组常用于网络通信中构造不定长数据包, 不会浪费空间浪费网络流量, 因为char data[0]; 只是个数组名, 是不占用存储空间的,

即 sizeof(struct zero\_buffer) = sizeof(int) = 4 （32位系统下）

长度为0的数组并不占有内存空间, 而指针方式需要占用内存空间；

对于长度为0数组, 在申请内存空间时, 采用一次性分配的原则进行; 对于包含指针的结构体, 才申请空间时需分别进行, 释放时也需分别释放；

对于长度为的数组的访问可采用数组方式进行。

<https://blog.csdn.net/gatieme/article/details/64131322?tdsourcetag=s_pctim_aiomsg>

27、 \_\_attribute((packed))字节对齐优化

1. \_\_attribute\_\_ ((packed)) 的作用就是告诉编译器取消结构在编译过程中的优化对齐,按照实际占用字节数进行对齐，是GCC特有的语法。这个功能是跟操作系统没关系，跟编译器有关，gcc编译器不是紧凑模式的，我在windows下，用vc的编译器也不是紧凑的，用tc的编译器就是紧凑的。例如：

在TC下：struct my{ char ch; int a;} sizeof(int)=2;sizeof(my)=3;（紧凑模式）

在GCC下：struct my{ char ch; int a;} sizeof(int)=4;sizeof(my)=8;（非紧凑模式）

在GCC下：struct my{ char ch; int a;}\_\_attrubte\_\_ ((packed)) sizeof(int)=4;sizeof(my)=5

2. \_\_attribute\_\_关键字主要是用来在函数或数据声明中设置其属性。给函数赋给属性的主要目的在于让编译器进行优化。函数声明中的\_\_attribute\_\_((noreturn))，就是告诉编译器这个函数不会返回给调用者，以便编译器在优化时去掉不必要的函数返回代码。

GNU C的一大特色就是\_\_attribute\_\_机制。\_\_attribute\_\_可以设置函数属性（Function Attribute）、变量属性（Variable Attribute）和类型属性（Type Attribute）。

\_\_attribute\_\_书写特征是：\_\_attribute\_\_前后都有两个下划线，并且后面会紧跟一对括弧，括弧里面是相应的\_\_attribute\_\_参数。

\_\_attribute\_\_语法格式为：

\_\_attribute\_\_ ((attribute-list))

其位置约束：放于声明的尾部“；”之前。

函数属性（Function Attribute）：函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。\_\_attribute\_\_机制也很容易同非GNU应用程序做到兼容之功效。

GNU CC需要使用 –Wall编译器来击活该功能，这是控制警告信息的一个很好的方式。

packed属性：使用该属性可以使得变量或者结构体成员使用最小的对齐方式，即对变量是一字节对齐，对域（field）是位对齐。

<https://blog.csdn.net/ipromiseu/article/details/5955295>

<https://blog.csdn.net/zhangxiong2532/article/details/50826917/>

28、 typeof关键字

它的作用是自动推导表达式的数据类型。typeof构造的主要应用是用在宏定义中。可以使用typeof关键字来引用宏参数的类型。

C语言中 typeof 关键字是用来定义变量数据类型的。在linux内核源代码中广泛使用。

下面是Linux内核源代码中一个关于typeof实例：

#define min(x, y) ({ \

typeof(x) \_min1 = (x); \

typeof(y) \_min2 = (y); \

(void) (&\_min1 == &\_min2); \

\_min1 < \_min2 ? \_min1 : \_min2; })

1.当x的类型为是 int 时 \_min1变量的数据类型则为 int。

2.当x为一个表达式时(例: x = 3-4), \_min1变量的数据类型为这个表达式结果的数据类型。

3.typeof括号中也可以是函数

    例:

int function(int, int);

typeof(function(1. 2)) val;

    此时val的数据类型为 函数function(int, int)返回值的数据类型    ，即int类型。(注意: typeof并不会执行函数function)。

typeof关键字有点类似与c++中的decltype关键字。

https://blog.csdn.net/zhanshen2015/article/details/51495273

29、 container of()函数简介

 container\_of（ptr, type,member）函数的实现包括两部分：

1.  判断ptr 与 member 是否为同意类型

2.  计算size大小，结构体的起始地址 = (type \*)((char \*)ptr - size) (注：强转为该结构体指针)

现在我们知道container\_of()的作用就是**通过一个结构变量中一个成员的地址找到这个结构体变量的首地址。**

container\_of(ptr,type,member),这里面有ptr,type,member分别代表指针、类型、成员。

https://blog.csdn.net/s2603898260/article/details/79371024

30、 do { code; } while(0);

实现复杂的宏定义：函数式宏、符合语句；使用局部变量；在条件语句中使用复杂的宏定义。

Struct student{

Char name[20];

Int age;

};

// #define PRINT(a) printf(“%s\n”, a.name);printf(“%d\n”, a.age);

#define PRINT(a) \

do{printf(“%s\n”, a.name);printf(“%d\n”, a.age);}while(0);

Int main()

{

Struct student stu = {“wit”, 20};

If(0)

PRINT(stu); // 至少运行一次

Return 0;

}

31、 Makefile使用语法

= 是最基本的赋值  
:= 是覆盖之前的值  
?= 是如果没有被赋值过就赋予等号后面的值  
+= 是添加等号后面的值

“$(wildcard \*.c)”来获取工作目录下的所有的.c文件列表

SRC = $(wildcard \*.c)

等于指定编译当前目录下所有.c文件，如果还有子目录，比如子目录为inc，则再增加一个wildcard函数，像这样：

SRC = $(wildcard \*.c) $(wildcard inc/\*.c)

$@ 表示规则中的目标文件集。在模式规则中，如果有多个目标，那么，"$@"就是匹配于目标中模式定义的集合。

$% 仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix下是[.a]，Windows下是[.lib]），那么，其值为空。

$< 依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$? 所有比目标新的依赖目标的集合。以空格分隔。

$^ 所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

在$(patsubst %.c,%.o,$(dir) )中，patsubst把$(dir)中的变量符合后缀是.c的全部替换成.o。

gcc -I -L -l的区别：

gcc -o hello hello.c -I /home/hello/include -L /home/hello/lib -lworld

上面这句表示在编译hello.c时-I /home/hello/include表示将/home/hello/include目录作为第一个寻找头文件的目录，

寻找的顺序是：/home/hello/include-->/usr/include-->/usr/local/include

-L /home/hello/lib表示将/home/hello/lib目录作为第一个寻找库文件的目录，

寻找的顺序是：/home/hello/lib-->/lib-->/usr/lib-->/usr/local/lib

-lworld表示在上面的lib的路径中寻找libworld.so动态库文件（如果gcc编译选项中加入了“-static”表示寻找libworld.a静态库文件）

https://blog.csdn.net/liangkaiming/article/details/6267357

<https://www.jianshu.com/p/5982ccb87af0>

https://www.cnblogs.com/wang\_yb/p/3990952.html

32、 可变参数宏

**#define** debug(fmt, ...) printf(fmt, \_\_VA\_ARGS\_\_) // GCC

\_\_VA\_ARGS\_\_替换可变参数

**#define** debug2(fmt, args...) printf(fmt, args) // GNU

**#define** debug3(fmt, ...) printf(fmt, **##**\_\_VA\_ARGS\_\_) //可以接受无参数(变参为空), ## 屏蔽逗号

**#define** C(x) #x // 一个#将参数字符串化

33、vim IDE配置

https://blog.csdn.net/u012385733/article/details/79038973

https://blog.csdn.net/namecyf/article/details/7787479?utm\_source=blogxgwz0

https://blog.csdn.net/guyue35/article/details/54412428

https://blog.csdn.net/eric\_sunah/article/details/51028874

https://github.com/DemonCloud/dotfile

34、 Webbench源码阅读

https://blog.csdn.net/l\_xrui/article/details/76216079

https://blog.csdn.net/earbao/article/details/77785613

https://www.jianshu.com/p/d59b3b971aa3

35、 值得学习的C语言开源项目

https://www.cnblogs.com/coky/p/6844445.html

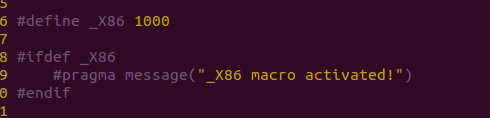
36、 #Pragma 指令

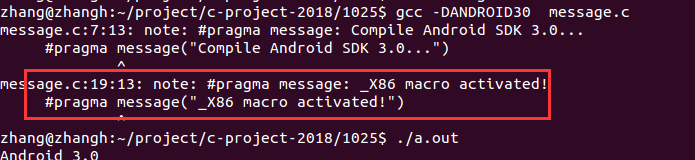
在所有的预处理指令中，#Pragma 指令可能是最复杂的了，它的作用是设定编译器的状态或者是指示编译器完成一些特定的动作。#pragma指令对每个编译器给出了一个方法，在保持与C和C++语言完全兼容的情况下，给出主机或操作系统专有的特征。依据定义，编译指示是机器或操作系统专有的，且对于每个编译器都是不同的。

其格式一般为: #pragma para。其中para为参数，下面来看一些常用的参数。

## （1） message参数

message参数能够在编译信息输出窗口中输出相应的信息，

其使用方法为：    
    #pragma  message("消息文本")    
    当编译器遇到这条指令时就在编译输出窗口中将消息文本打印出来。    
    当我们在程序中定义了许多宏来控制源代码版本的时候，我们自己有可能都会忘记有没有正确的设置这些宏，  
此时我们可以用这条指令在编译的时候就进行检查。假设我们希望判断自己有没有在源代码的什么地方定义了\_X86这个宏,  
可以用下面的方法:  
  
    我们定义了\_X86这个宏以后，应用程序在编译时就会在编译输出窗口里显示"\_86  macro  activated!"。



## （2） once参数

* #pragma one用于保证**头文件只被编译一次**
* #pragma one与编译器相关，不一定被支持
* 在工程中的使用：
* #ifndef \_HEADER\_FILE\_H\_
* #define \_HEADER\_FILE\_H\_
* #pragma once
* // source code
* #endif

## （3） warning参数

#pragma  warning( disable: 4507 34; once: 4385; error: 164 )  
    等价于：    
    #pragma  warning( disable: 4507 34 )    //  不显示4507和34号警告信息    
    #pragma  warning( once: 4385 )          //  4385号警告信息仅报告一次    
    #pragma  warning( error: 164 )          //  把164号警告信息作为一个错误。

同时这个pragma  warning  也支持如下格式：    
    #pragma  warning( push [, n ] )    
    #pragma  warning( pop )    
    这里n代表一个警告等级(1---4)。    
    #pragma  warning( push )保存所有警告信息的现有的警告状态。    
    #pragma  warning( push, n )保存所有警告信息的现有的警告状态，并且把全局警告等级设定为n。      
    #pragma  warning( pop )向栈中弹出最后一个警告信息，在入栈和出栈之间所作的一切改动取消。

例如：    
    #pragma  warning( push )    
    #pragma  warning( disable: 4705 )    
    #pragma  warning( disable: 4706 )    
    #pragma  warning( disable: 4707 )    
    //.......    
    #pragma  warning(  pop  )      
    在这段代码的最后，重新保存所有的警告信息(包括4705，4706和4707)。

https://www.cnblogs.com/fnlingnzb-learner/p/5854494.html

https://blog.csdn.net/jacob\_\_lei/article/details/79475926#t7

37、

38、

39、

40、