

**二分查找法：**

int begin = 0;

int end = len - 1;

int mid ;

while(begin <= end)

{

mid = (begin + end) / 2;

if(a[mid] > n)

{

end = mid - 1;

}

else if(a[mid] < n)

{

begin = mid + 1;

}

else

{

break;

}

}

if(begin <= end)

{

printf("found! position = %d, value = %d\n", mid, n);

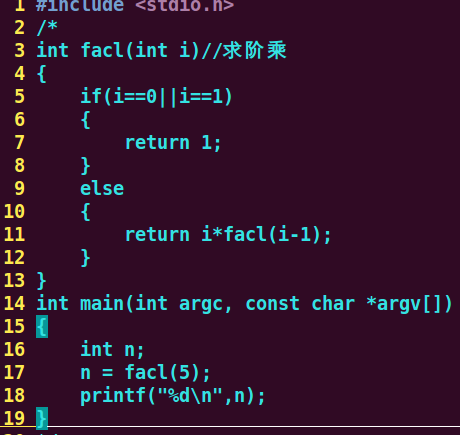
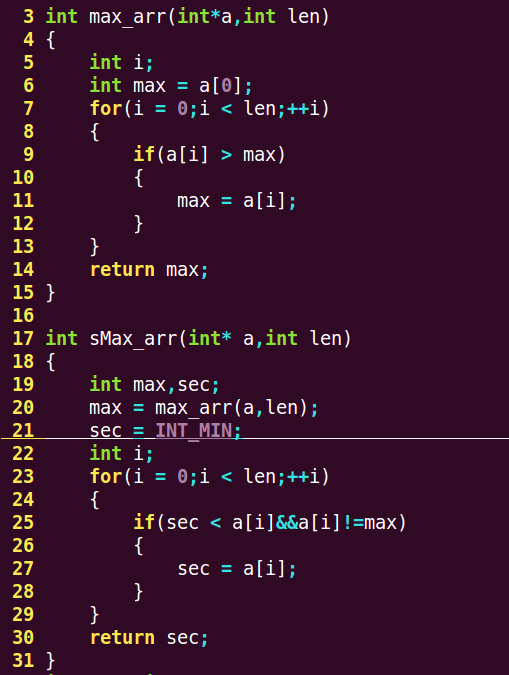
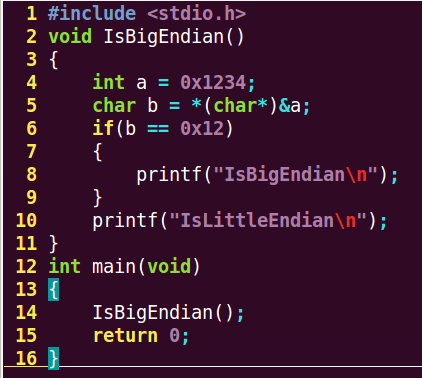
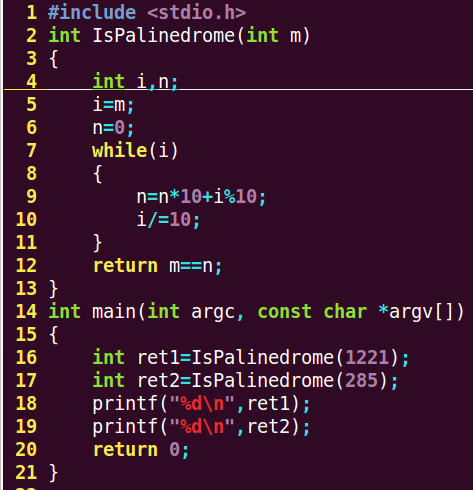
}

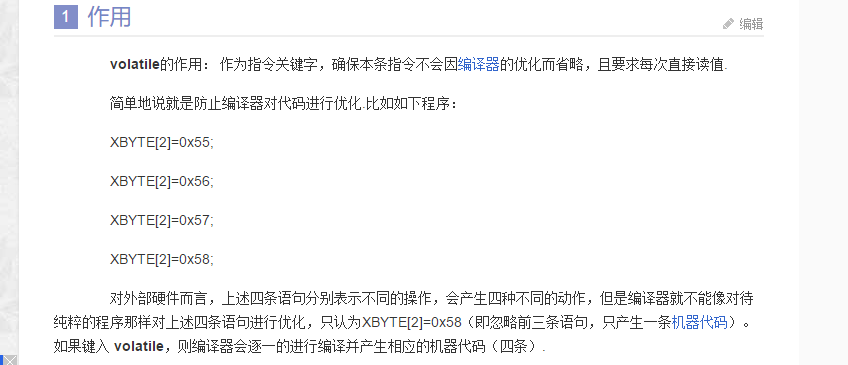
else

{

printf("not found!\n");

}





volatile 变量是随时可能发生变化的，与volatile变量有关的运算，不要进行编译优化，以免出错

例如：   
volatile int i=10;   
int j = i;   
...   
int k = i;   
volatile 告诉编译器i是随时可能发生变化的，每次使用它的时候必须从i的地址中读取，因而编译器生成的可执行码会重新从i的地址读取数据放在k中。   
而优化做法是，由于编译器发现两次从i读数据的代码之间的代码没有对i进行过操作，它会自动把上次读的数据放在k中。而不是重新从i里面读。这样以来，如果i是一个寄存器变量或者表示一个端口数据就容易出错，所以说volatile可以保证对特殊地址的稳定访问，不会出错。

下面是volatile变量的几个例子：   
1) 并行设备的硬件寄存器（如：状态寄存器）   
2) 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)   
3) 多线程应用中被几个任务共享的变量

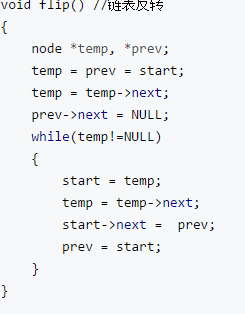
1. 一个参数既可以是const还可以是volatile吗？解释为什么。   
   2); 一个指针可以是volatile 吗？解释为什么。   
   3); 下面的函数有什么错误：   
   int square(volatile int \*ptr)   
   {   
   return \*ptr \* \*ptr;   
   }   
   下面是答案：   
   1)是的。一个例子是只读的状态寄存器。它是volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它。   
   2); 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个buffer的指针时。

int square(volatile int \*ptr)   
{   
int a,b;   
a = \*ptr;   
b = \*ptr;   
return a \* b;   
}   
由于\*ptr的值可能被意想不到地该变，因此a和b可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：   
long square(volatile int \*ptr)   
{   
int a;   
a = \*ptr;   
return a \* a;   
}

**一：告诉compiler不能做任何优化**

**二：表示用volatile定义的变量会在程序外被改变,每次都必须从内存中读取，而不能把他放在cache或寄存器中重复使用。**

**单链的反转**



单向链表删除

int del\_list(LinkList\* list,char\* name)

{

LinkNode \*q=list->head;

LinkNode \*p=list->head;

if(NULL == list->head->next)//只有一个元素

{

if(0==strcmp(list->head->data.name,name))

free(list->head);

//free(list);

list->head=NULL;

}

else

{

while(p)

{

if(0==strcmp(p->data.name,name))//不止一个元素

{

if(p == q)//第一个就是要找的

{

list->head = p->next;

}

else

{

q->next = p->next;

}

free(p);

break;

}

else

{

q = p;

p=p->next;

}

}

}

return 0;



操作系统面试题

1、进程和线程的区别？

解析：（1）进程是资源的分配和调度的一个独立单元，而线程是CPU调度的基本单元

          （2）同一个进程中可以包括多个线程，并且线程共享整个进程的资源（寄存器、堆栈、上下文），一个进行至少包括一个线程。

          （3）进程的创建调用fork或者vfork，而线程的创建调用pthread\_create，进程结束后它拥有的所有线程都将销毁，而线程的结束不会影响同个进程中的其他线程的结束

          （4）线程是轻两级的进程，它的创建和销毁所需要的时间比进程小很多，所有操作系统中的执行功能都是创建线程去完成的

          （5）线程中执行时一般都要进行同步和互斥，因为他们共享同一进程的所有资源

          （6）线程有自己的私有属性TCB，线程id，寄存器、硬件上下文，而进程也有自己的私有属性进程控制块PCB，这些私有属性是不被共享的，用来标示一个进程或一个线程的标志

2、死锁？死锁产生的原因？死锁的必要条件？怎么处理死锁？

解析：（--）相互等待资源而产生的一种僵持状态，如果没有外力的干预将一直持续这个状态

          （--）系统资源不足、相互竞争资源、请求资源顺序不当

          （2）互斥、不可抢占、循环等待、请求与保持

          （3）因为互斥是不可改变的，所以只能破坏其他三个条件中的一个来解除死锁，方法：剥夺资源、杀死其中一个线程

3、Windows内存管理方式：段存储、页存储、段页存储

解析：

4、进程的几种状态？

解析：（1）run（运行状态）：正在运行的进程或在等待队列中对待的进程，等待的进程只要以得到cpu就可以运行

          （2）Sleep（可中断休眠状态）：相当于阻塞或在等待的状态

          （3）D（不可中断休眠状态）：在磁盘上的进程

          （4）T（停止状态）：这中状态无法直观的看见，因为是进程停止后就释放了资源，所以不会留在linux中

          （5）Z（僵尸状态）：子进程先与父进程结束，但父进程没有调用wait或waitpid来回收子进程的资源，所以子进程就成了僵尸进程，如果父进程结束后任然没有回收子进程的资源，那么1号进程将回收

5、IPC通信方式？

解析：（1）管道（匿名管道（pipe亲缘关系的进程通信）、命名管道（mkfifo/mknod））

          （2）消息队列：是基于消息的、用无亲缘关系的进程间通信，主要函数：msgget、msgsend、msgrecv、msgctl

          （3）信号量：相当于一把互斥锁，通过p、v操作，主要函数：semget、semop、semctl

          （4）共享内存：是进程间通信速度最快的，所以用经常是集合信号量或互斥锁来实现同步，shmget、shmat、shmdt、shmctl

6、什么是虚拟内存？

解析：是将进程部分装入内存中，从而能实现一个很大的程序能在一个比它小的内存中运行，它的主要实现是靠程序的换进换出来实现的，因为内存中0~3G是用户使用，3~4G才是内存使用，通过映射来实现来进行逻辑地址到物理地址的映射

7、虚拟地址、逻辑地址、线性地址、物理地址的区别？

解析： 分段机制把一个逻辑地址转换为线性地址；接着，分页机制把一个线性地址转换为物理地址。

（1）虚拟地址：虚拟内存映射出来的地址

（2）逻辑地址：程序的段加偏移量形成的，C/C++程序中取地址求出来的地址就是逻辑地址

（3）线性地址：是逻辑地址到物理地址的中间层，只有启动分页机制的时候才有线性地址，如果没有分页机制，那么线性地址就是物理地址

（4）物理地址：是内存中实实在在存在的硬件地址，

逻辑地址（启动分段）--》线性地址（启动分页）--》物理地址

原因请看上面的对比。

**1）需要频繁创建销毁的优先用线程**

这种原则最常见的应用就是Web服务器了，来一个连接建立一个线程，断了就销毁线程，要是用进程，创建和销毁的代价是很难承受的

**2）需要进行大量计算的优先使用线程**

所谓大量计算，当然就是要耗费很多CPU，切换频繁了，这种情况下线程是最合适的。

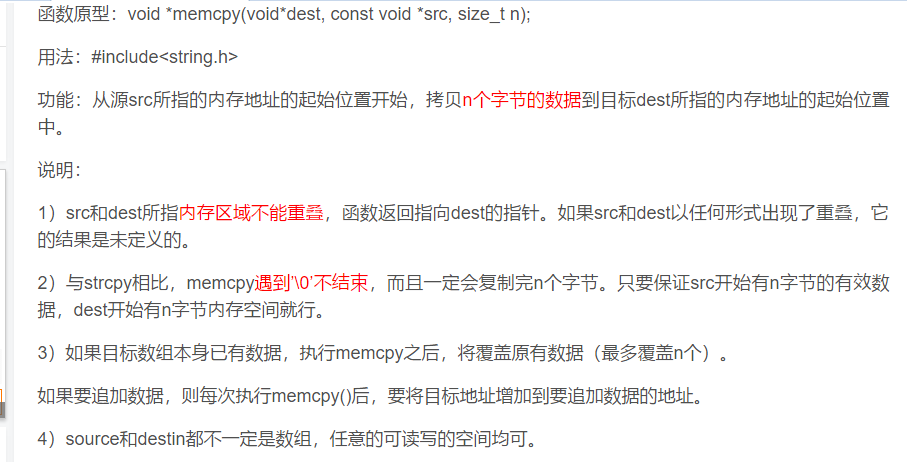
这种原则最常见的是图像处理、算法处理。

**3）强相关的处理用线程，弱相关的处理用进程**

什么叫强相关、弱相关？理论上很难定义，给个简单的例子就明白了。

一般的Server需要完成如下任务：消息收发、消息处理。“消息收发”和“消息处理”就是弱相关的任务，而“消息处理”里面可能又分为“消息解码”、“业务处理”，这两个任务相对来说相关性就要强多了。因此“消息收发”和“消息处理”可以分进程设计，“消息解码”、“业务处理”可以分线程设计。

当然这种划分方式不是一成不变的，也可以根据实际情况进行调整。



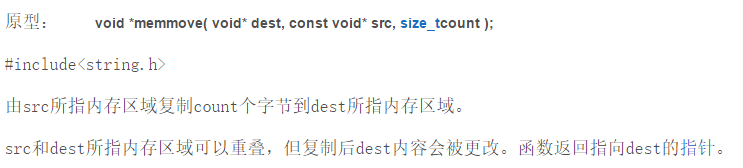
原型：extern void \*memcpy(void \*dest, void \*src, unsigned int count);

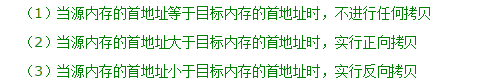
用法：#include <string.h>

功能：由src所指内存区域复制count个字节到dest所指内存区域。

说明：src和dest所指内存区域不能重叠，函数返回指向dest的指针。







Const char \*p；不能通过\*p修改\*p所指向的对象，可以通过其他指针修改\*p所指向的对象；

