1. 名词解释（写出中文名称，解释名词含义，并说明其作用）
2. **brk Pointer**：brk指针。

含义：用来表征**堆中可存放数据的起始指针**的位置。

作用：可以通过获得该指针，了解堆**可存放位置**，也可通过调节此指针，动态改变对动态内存的申请。

1. **Hot Spot**：热点。

含义：程序代码中，对**计算机CPU执行占据时间比例最大的部分**（通常以函数为单位）。

作用：通过寻找Hot Spot，可以发现程序中**最值得优化的部分**，通过对其优化，可获得整体代码优化的最佳性价比。

1. **Internal Fragmentation**：内部碎片。
2. **Spatial Localit**y：空间局部性

含义：未来对内存的引用与过去对内存的引用在**空间上是重复的**；或者具有**空间规律性**；或者，指令对内存的引用顺序与数据在内存中的存放顺序**保持一致性**。

作用：提高内存引用的**命中率**，提高指令的**运行效率**

1. **Concurrent Programming**：并发编程

含义：程序指令在执行的时间轴上存在一定的重叠，这种编程统称为并发编程。

作用：可以提高指令运行的效率

1. 简答题
2. 现有如下代码，请指出其中第6行语句的逻辑判断结果的真伪，并解释得此结论的原理

int array[] = { 23, 34, 12, 17, 204, 99, 16 };

#define TOTAL\_ELEMENTS (sizeof(array) /

sizeof(array[0]))

main()

{

int d= -1, x;

/\* ... \*/

**if (d <= TOTAL\_ELEMENTS-2)**

x = array[d+1];

/\* ... \*/

}

此逻辑判断结果为false

由于判断式左右两边的**数据类型**不同，导致编译器采取**整数提升**（integral promotions）。具体讲，**TOTAL\_ELEMENTS被定义为无符号整数，而d则是有符号整数**，编译器会主动将d提升为无符号整数参与逻辑对比操作，导致判断结果为false

1. 现有如下代码，（1）请写出其运行结果（2）这种现象在C语言中被称为什么？

switch (2) {

case 1: printf("case 1 \n");

case 2: printf("case 2 \n");

case 3: printf("case 3 \n");

case 4: printf("case 4 \n");

default: printf("default \n");

}

1. case 2

case 3

case 4

default

1. 这种现象在C语言中被称为Fall Through现象
2. 在IEEE浮点数表达式中，如果要变量由double型强制转换为float，有可能会被理解为+∞，请问发生这种情况的原理是什么？

（1）+∞在float类型中定义为：指数部分（E，占据8位），全为1，同时尾数部分（M，占据23位）全为0

（2）如果将double类型强制转换为float型后，由于double类型的E,M位数均大于float类型。在…情况下由于..的原因，可能会导致指数部分（E）全为1，同时尾数部分（M）全为0，也就是被理解为+∞

1. 对于C函数原型void\*malloc {size\_t size},翻译下面的一段话，

The *malloc* function returns a pointer to a block of memory of 1at least size bytes that is 2suitably aligned for 3any kind of data object that might be contained in the block.

1. malloc函数返回一个指向内存块的指针，该内存块可能被对齐处理从而具有不少于申请空间大小（以字节为单位），以方便在所申请的快中存放各种可能的数据对象
2. at least size bytes：由于对齐的需要，实际分配的空间可能大于申请的空间大小

suitably aligned：对齐是一种可能的操作，主要是为了提高内存访问的效率

any kind of data object：由于malloc函数返回的指针类型为void，因此所申请的空间可能用于存放各种可能的数据类型

1. 在自行设计堆的分配器时，（1）coalescing策略的作用是什么？（2）在什么条件下会触发coalescing策略？（3）改策略解决了何种碎片问题？
2. coalescing策略的作用在于通过对**相邻未使用空间**的合并，增加单个未使用空间的大小，从而**降低堆空间的碎片率**，提高堆空间的使用率
3. 在相邻两个空间状态均为“未使用”的情况下，会触发
4. 解决了内部碎片的问题
5. 在32bit系统中，现有结构myStruct如下（左），另有联合myUnion如下（右）。请分别指出两者在内存分配上的不同，并分别计算其所占的内存大小

Struct myStruct{ union myUnion{

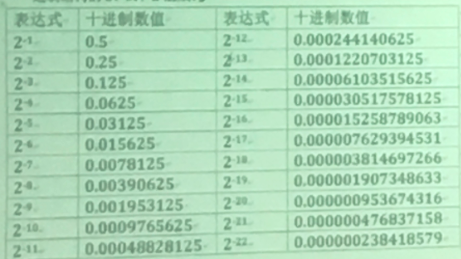
int i,j; int I,

char \*ptr; char\*ptr;

float fh; float fh;

} }

1. 内存分别的策略不同：结构是所有成员的内存占有之和，且考虑到地址对齐的问题；而联合体则是成员中占用地址最大的一个作为总的空间
2. 左面的内存占用16字节，右边的占用4字节
3. 实践题
4. 在测量应用程序性能时，通常采用的度量方法有几种？分别阐述它们的优缺点
5. 工具度量法。该方法采用系统工具，度量的是应用程序总体运行时间，实际占用CPU时间，以及其它应用程序占据CPU的时间等
6. 代码方法。该方法通过自定义的代码函数，测量应用程序实际占用CPU的时间
7. Profile方法。通过对应用程序运行时采样，度量应用程序占用CPU的时间
8. 以上3中方法均可以度量应用程序占用CPU的时间，这在多任务系统中非常有效，可以排除其他应用程序占用CPU的时间。工具度量法与代码方法的优点是简单；但是不能细化度量应用程序中函数对CPU占用的时间分布；对于应用程序优化帮助不大。Profiling方法的优点是可以度量应用程序中各个函数对于CPU占用的时间分布，可以帮助调试寻找应用程序的HOT SPOT，从而发现应用程序最值得优化的部分，缺点是该方法必须针对源代码进行调试。
9. 现有16进制数0x408d0000，若将其声明为4字节带符号整数，其十进制值是多少？若将其声明为4字节浮点数，其十进制值又是多少？（详细演算过程，进行浮点数演算时，需分别标明浮点数二进制结构S,M,E组成）



0x408d0000

* 0100 0000 1000 1101 0000 0000 0000 0000

1. 整数

最高位为0，表示整数

十进制：216 + 218 + 219 + 223 + 230

1. （单精度）浮点数

S：0，代表该浮点数为正数

M：000 1101 0000 0000 0000 0000

E：100 0000 1

该浮点数的形式化可以表示为 S\*（1.M）2\*2E-127

其中：E的十进制为129，E-127=2，,1.M的二进制表示为：

1.000 1101 0000 0000 0000 0000

S\*（1.M）2\*2E-127表示为：

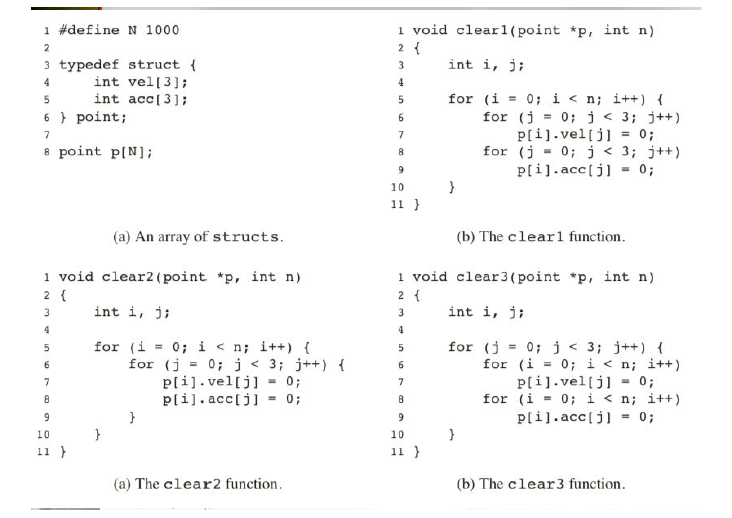
100.0 1101 0000 0000 0000 0000

因此，小数点前的十进制为4，小数点后的十进制为：

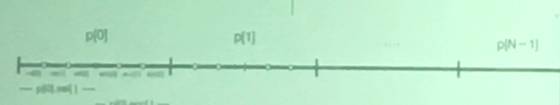
2-2+2-3+2-5=0.40625

故，完整的数值为：4.40625

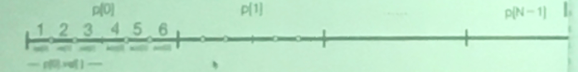
1. 下面的4段代码中，定义了结构point的组成，以及clear1、clear2、clear3三个函数，（1）请绘图表述p在内存中的分配情况，并分别画出三个函数中结构变量被调顺序情况（2）根据局部性的原理，指出三个函数的局部性从优到差的顺序，并说明分析的依据



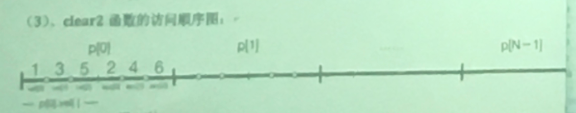
1. 内存顺序图



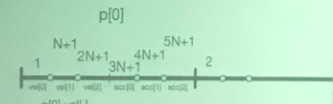
1. Clear1函数的访问顺序图



1. clear2函数访问顺序图



1. clear3函数访问顺序图



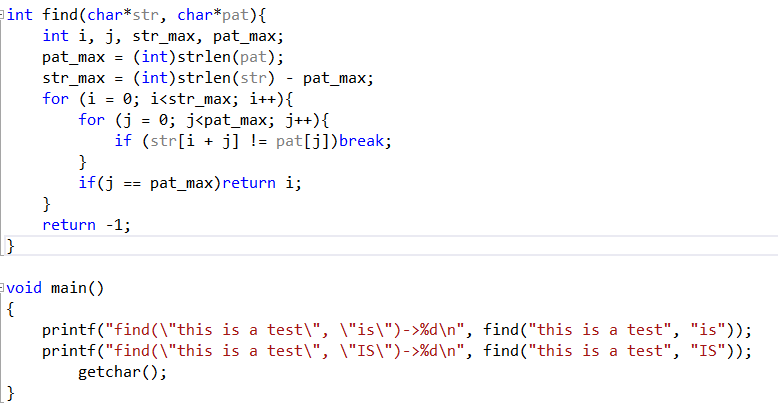
1. 根据局部性原理，如果程序逻辑的访问顺序与物理存储的顺序越一致的话，那么，空间局部性就可以利用缓存的机制，增加内存数据的访问效率，按照以上思想，clear1函数对内存的访问与物理存储顺序严格一致，clear2函数对内存的访问顺序与物理存储顺序存在跳跃，跳跃度为3个整数长度，而clear3函数对内存的访问与物理存储顺序存在大幅跳跃，跳跃幅度为N\*6个整数长度，因此可以推断，从空间局部性的角度看，3个函数对内存的访问效率从小到大的次序为：

Clear1>clear2>clear3

1. 综合题

 unsigned int strlen (char \*s);

1. 在32bit环境下调试如下代码，得知：



1. 主函数main所分配的堆栈（ebp）0=0xbffff9a8，（esp）0=0xbffff980
2. 被调函数find所分配的堆栈（ebp）1=0xbffff978，（esp）1=0xbffff950

回答以下问题：

1. 该代码输出结果是什么？
2. 被调函数的返回地址是什么？说明得出结论的依据
3. （ebp）1地址中所存放的内容是什么？这样做的作用是什么？
4. 如果被调函数find中，局部变量均不存放在寄存器中，那么变量str\_max,pat\_max在内存中的地址分别是什么？
5. 输出结果是：

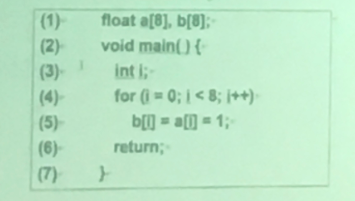
find(“this is a test”,”is”) -> 2

find(“this is a test”,”IS”) -> -1

1. 被调函数的返回地址是0xbffff97c

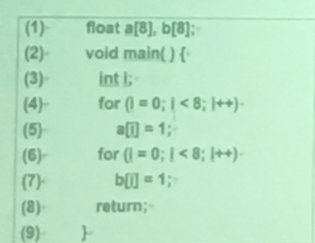
依据：返回地址是从被调函数的（ebp）1的地址起，向高地址偏移4字节

1. （ebp）1地址中存放的内容是调用者函数的ebp，也就是xbffff9a8。这样的好处是，函数执行结束后，可以迅速恢复调用者的堆栈，从而在调用者与被调者之间形成一个机械而稳定的恢复机制。
2. str\_max,pat\_max在内存中的地址分别是0xbffff960和0xbffff95c
3. 现有如下代码，为简单起见，假设采用直接映射型缓存结构，缓存块的大小为16bytes（即可存放4个浮点数）而且缓存有两个set构成，即共有32bytes的缓存空间。假设变量i存放在register中
4. 填写表格中的地址偏移量（从0开始），缓存分组
5. 逐一标明a[],b[]命中和不命中的状态
6. 请改写该段代码，提高该段代码的内存命中率
7. 计算改写前、后的该段代码的内存命中率



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 单元 | 地址偏移量 | 缓存分组 | 单元 | 地址偏移量 | 缓存分组 |
| a[0] | 0 | 0 | b[0] | 32 | 0 |
| a[1] | 4 | 0 | b[1] | 36 | 0 |
| a[2] | 8 | 0 | b[2] | 40 | 0 |
| a[3] | 12 | 0 | b[3] | 44 | 0 |
| a[4] | 16 | 1 | b[4] | 48 | 1 |
| a[5] | 20 | 1 | b[5] | 52 | 1 |
| a[6] | 24 | 1 | b[6] | 56 | 1 |
| a[7] | 28 | 1 | b[7] | 60 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 单元 | 命中状态 | 单元 | 命中状态 |
| a[0] | m | b[0] | m |
| a[1] | m | b[1] | m |
| a[2] | m | b[2] | m |
| a[3] | m | b[3] | m |
| a[4] | m | b[4] | m |
| a[5] | m | b[5] | m |
| a[6] | m | b[6] | m |
| a[7] | m | b[7] | m |



|  |  |  |  |
| --- | --- | --- | --- |
| 单元 | 命中状态 | 单元 | 命中状态 |
| a[0] | m | b[0] | m |
| a[1] | h | b[1] | h |
| a[2] | h | b[2] | h |
| a[3] | h | b[3] | h |
| a[4] | m | b[4] | m |
| a[5] | h | b[5] | h |
| a[6] | h | b[6] | h |
| a[7] | h | b[7] | h |

前：0%

后：75%