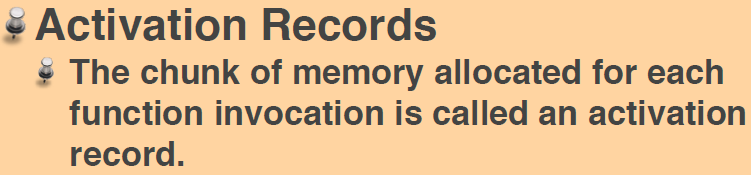
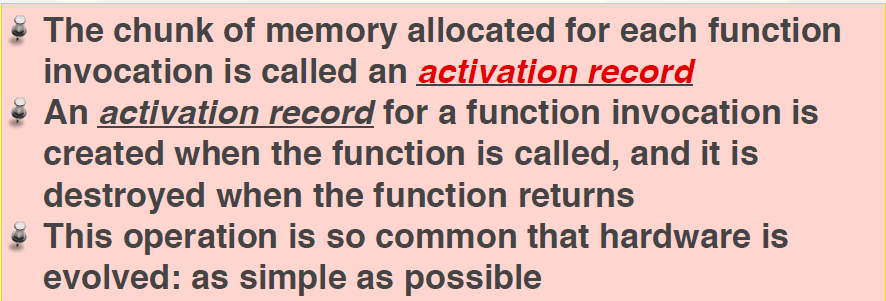
1. 名词解释
2. **Activation Record**

活动记录表（即栈帧）。含义：函数调用时，为该函数分配的内存块

作用：用于记录函数信息的存储块。（因为活动记录使用栈存储，一个活动记录又称栈帧（Stack Frame））

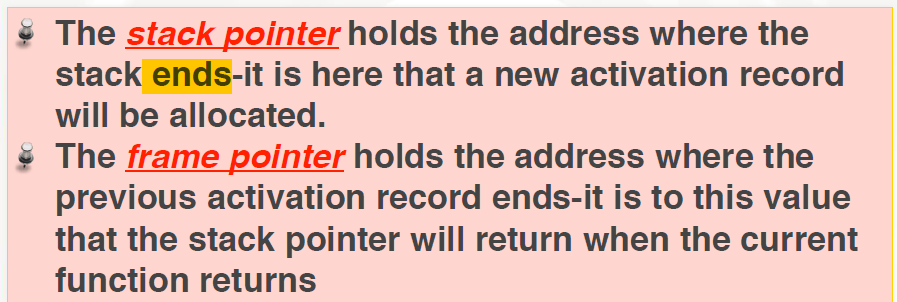




1. **Frame Pointer**

帧指针。含义：记录了当前活动记录的结束地址，也就是函数返回时，栈指针指向的位置。又称活动记录基地址。在Pentium里称为EBP

作用：方便函数调用栈



1. **Stack Pointer**

栈指针。指向栈顶，即下一次活动记录开始分配的地方，保存在寄存器ESP中

作用：方便访问内存栈中当前位置的内容

1. **Heap**

堆。含义：由程序员分配和释放的内存区域，通常位于低地址段

作用：动态分配内存，赋予程序员之间操作内存的权利

1. **Stack**

栈。含义：由系统根据需要自动分配和回收的空间

作用：隐式分配内存，不必担心因为忘记释放内存而造成的内存泄露问题

1. **Hit Ratio**

命中率。含义：指缓存命中率，即程序运行时，CPU可以从缓存中取到的数据或指令大小与程序所需数据指令大小之比。

作用：为了优化程序运行效率，我们需要计算cache的命中率，通过局部性原理，修改代码，从而提高缓存的命中率。

1. Program Counter

程序计数器。储存CPU下一条要执行的指令地址。

1. Memory Hierarchy

存储器层次结构。寄存器，cache, memory disk等，依次速度降低，容量增大，价格减少

1. 简答题
2. **ANSI C标准中规定 的括号“(”嵌套最多可以有多少层？在32位系统中，ANSI C中规定的long int最大值是多少（以16进制表示）ANSI C标准中规定的函数定义中，最多多少个参数？**

32,0x7fffffff，31个参数

1. C语言中存在哪些抽象？

使用变量名，而不是直接通过地址值进行访问变量

使用array[i]的形式访问数组元素，而不需要自己计算元素的地址

使用c=a+b，这样的代码实现加法，而不需要直接向CPU下达命令

1. 解释未初始化的局部变量与已知初始化局部变量在内存表现上有何相同点与不同点？

相同点：都存放在栈中

不同点：未初始化的局部变量的位置与已初始化的存放区域不同

1. **未初始化的全局变量和初始化的全局变量在内存中表现上有何区别**

初始化的全局变量安置在一个连续的内存区域（Initialized Data），为了使程序高效加载，未初始化全局变量被放在Uninitialized Data区域的固定位置，并被置为0

|  |
| --- |
| Kernel |
| Stack |
|  |
| Shared Libraries |
|  |
| Heap |
| Uninitialized Data |
| Initialized Data |
| Program Text |
| Page Zero |

1. **在缓存存储结构表达中，S,E,B分别代表什么含义**

S：组的个数 E：每个组的行数 B：块的大小

1. **IEEE标准下，从double向float强制转换过程中，可能会出现什么意外情况，试从原理上说明原因**

精度丢失，若指数第8位以上全为0，由于底数只能取前23位，则会发生精度损失

溢出，若指数第8位以前全为1，则会因为指数过大而溢出

|  |  |  |  |
| --- | --- | --- | --- |
|  | S | E | M |
| float | 1 | 8 | 23 |
| double | 1 | 11 | 52 |

注：一些特殊数的表示

正负0：S=正负1，M全为0，E全为0

无穷大：S=正负1，M全为0，E全为1

NaN：M不等于0，E全为1

1. **在自行构建一个堆的分配器（Allocator）时，需要考虑的策略有哪些？**

**组织策略**：对已分配的内存进行跟踪

**放置策略**：选择一个合适的空闲块来放置一个新分配的块

**分割策略**：在我们将一个新分配的快放置到某个空闲块之后，我们如何处理这个空闲块中的剩余部分

**合并策略**：释放块之后，如何将它与邻块连接起来

1. **在设计一个层次结构的缓存机制时，需要考虑哪些策略？**

**读取：**何时从较低层的存储结构读取数据，以及每次读多少数据

**放置：**被读取的数据应该如何放置，它将如何响应被CPU请求

**替换：**当存储空间已满，无法存储读取数据，哪些数据将被换出

**更新：**当缓存数据发生了改变，较低层的存储器是应该立刻更新还是稍后更新

1. **假设某编译器强制以4字节对齐，那么对于下列数据结构，以sizeof()函数计算出的字节是多少**？20

myStruct{

int i,j;

char \*ptr;

char b;

float fh;

}

1. **在堆分配操作中，通常要求8字节对齐，其原理是什么？**
2. 减少内存碎片（2） 兼容更多的数据类型
3. **系统使用malloc分配内存时，会分配哪些数据？**

头部（Header），有效载荷（Payload），填充（Padding）

The block size including the header, payload and padding

Malloc returns a pointer to the beginning of the payload

Header

|  |  |
| --- | --- |
| Block size | 0 0 a |
| Payload | |
| Padding | |

1. **写出Amdahl’s Law的形式化表达式，注明各个符号的含义，并阐明其对代码性能优化作业的指导意义**。

：优化前的运行时间；：优化后的运行时间

：可优化部分所占的比例；：速度提升倍速

：任何程序都存在可优化的部分，局部优化代码带来的效率提升是有限的

1. **80/20原理**

80%的时间都是由20%的代码耗费的，每个程序都存在性能提升的瓶颈，只要克服了这个瓶颈，性能就会大大提升。

1. **假设某浮点数的16进制表达式为0x40490FDB，请将其转换为10进制的和数值，并详细标明演算的过程，以及浮点数表达式的3个关键参数。**

40490FDB

🡪0100 0000 0100 1001 1111 1101 1011

S：0

E：100 0000 0

M：100 1001 1111 1101 1011

(1.100 1001 1111 1101 1011)2\*10128-12

=(11.00 1001 1111 1101 1011)2

=3.14

1. **现有两个整数，分别存放在地址\*x和\*y中，请用逻辑操作的方式书写一段函数代码，两个地址中的内容互换。**

void inplace\_swap(int \*x,int \*y){

\*y = \*x^\*y;

\*x = \*x^\*y;

\*y = \*x^\*y;

}

1. **在标准C中，堆摧毁可以使用函数free(void \*ptr)。请解释为啥函数定义中未指定销毁内存的大小？**

在用malloc函数动态分配内存时，通过参数size\_t，我们指定了大小为size\_t的开辟空间，并返回一个指向这个空间首地址的指针ptr。这块区域的大小已经保存在这块区域中。因此，释放内存时，不需要指定大小。没有意义。

1. **GCC编译环境下，一个C源代码编译为可执行代码需要几个内部过程？**

Hello.c🡪预处理🡪hello.i🡪编译🡪hello.s🡪汇编🡪hello.o//printf.o🡪链接🡪hello.exe

预处理：根据#开头的语句修改原始c程序，得另一个c程序\*.i

编译：翻译，生成汇编语言程序\*.s

汇编：翻译成机器语言指令，打包成一个可重定位目标程序\*.o，二进制文件

链接：合并printf.o为可执行程序

1. **C语言中实现动态内存分配的方法有几种，他们之间的主要差异是什么？**

堆分配和栈分配

栈分配在堆栈帧中完成，通过代码中声明局部变量或使用alloca函数实现，无需手动释放空间，随活动记录一同释放。堆分配是在内存的堆区域中申请空间，一般用malloc或calloc完成，需要手动使用free释放空间

1. **解释静态分配与动态分配的概念以及它们的不同点，并说明它们的用途**

静态分配在全局中申请变量，或在函数体中用static，在声明时若没有被赋初始值，则被赋为0，放于内存的低地址区域，生命周期是整个程序的生命周期，在编译阶段决定

动态分配通过在函数调用中声明局部变量实现，生命周期是此函数的生命周期

1. **系统自动分配堆栈时，应保存哪些数据。绘图表示这些数据在堆栈中的位子（简述使用堆栈进行函数调用的过程）**

栈：函数调用时，首先将参数从右向左入栈，接着压入函数的返回地址，接着压入Frame Pointer，接着压入局部变量，执行函数的代码，将返回值压入寄存器中，将sp设为Frame Pointer，弹出栈顶FP，弹出栈顶的返回地址并返回函数

堆：一般在堆的头部用一个字节存放堆的大小，堆中的具体内容由程序猿安排

|  |
| --- |
| 参数 |
| 返回地址 |
| ebp |
| Local variable |
| esp |

1. **设计一个层次结构的缓存机制，需要考虑哪些策略**

Fetch Strategy：CPU要求的数据必须在cache中被立刻找到，当miss时，在下层中找到

Placement Strategy：当一个值进入cache，它的内存中的地址要被记下来使CPU匹配

Replacement Strategy：当cache不够时，如何选择一个cache line驱逐

Update Strategy：当cache中的数据值改变时，如何用主存中的更新

1. **显示动态分配与隐式动态分配**

显示：内存由程序动态分配，由程序释放

隐式：内存由程序动态分配，但是由系统自动回收，需要垃圾回收机制

1. **悬挂指针（dangling pointer problem）**

程序虽然保留了内存指针，但是这部分内存却已经被释放和重新分配，导致使用指针取到无关值。

1. **垃圾回收机制有哪些？**
2. 标记-清除算法

首先标记处所有需要回收的对象，在标记完成后统一收掉所有被标记的对象。缺点有两个：一个事效率问题。标记和清除过程的效率都不高；另一个是空间问题，标记清除之后会产生大量的不连续的内存碎片。

1. 复制算法

它将可用内存按容量划分成大小相等的两块，每次只使用其中的一块。当这块的内存用完了，就将还存活的对象复制到另一块上面，然后再把已使用的内存空间一次清理掉。内存分配时也就不用考虑碎片的问题了，只要移动堆顶指针，按顺序分配内存即可实现简单，运行高效。但代价是内存缩小为原来的一半。

1. 标记-整理算法

前半部分与（1）相似，但不是直接回收，而是让所有存货的对象都向一端移动，然后直接清理掉边界以外的内存。

1. 引用计数法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值加1，当引用失效时，减1，任何时候计数器为0的对象都是不可能再被使用的，可以被清除掉。它不能解决对象之间的相互循环引用问题

1. 分代收集算法

根据对象的存活周期不同将内存划分为几块。一般是把java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

1. **内存泄露**

用动态存储分配函数开辟的空间，在使用结束后未释放，结果导致一直占据该内存单元，直至程序结束

1. 与传统的单线程相比，多线程设计时需要注意哪些问题？

使用安全的C函数，对共享成员的访问尽量独立化

1. **内存对齐的原理（原因？）是？（Alignment）**

平台原因：不是所有的硬件平台都能访问任意地址上的任意数据，某些硬件平台只能在某些地址取某些特定类型的数据，否则抛出硬件异常

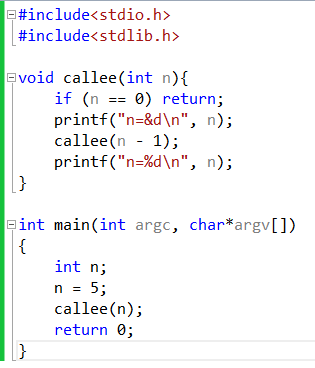
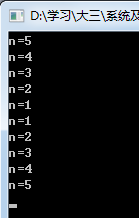
性能原因：数据结构应该尽可能在自然边界上对齐，为了访问未对齐的内存，处理器需要作两次内存访问，而对齐的内存仅需要一次

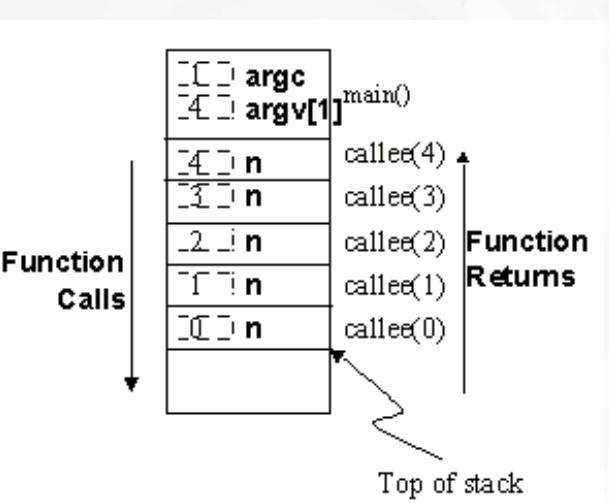
1. **多任务操作系统中，代码执行的总运行时间由几个部分时间组成？**

总运行时间 = CPU执行时间 + 等待I/O时间

CPU执行时间 = user time（直接运行代码的时间）+ system time（OS代理代码的运行时间）

1. **阅读如下代码，写出运行输出结果，并在内存分布图中画出函数调用时的堆栈分配时序图以及堆栈销毁时序图**



1. **在计算机逻辑代数中，“&”、“|”、“&&”分别表示什么？假设现有两个整数，分别是：a=0x06，b=0x07，那么a&b是多少？（a&b）&&（a|b）是多少？**

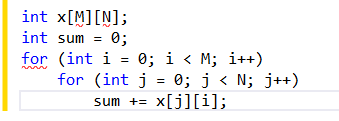
a=0000 0110

b=0000 0111

a&b=0000 0110=0x06

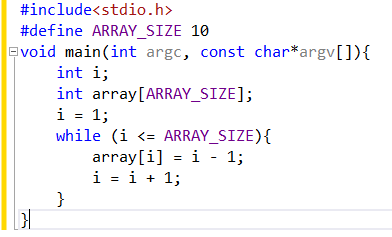
（a&b）&&（a|b）=0000 0110 && 0000 0111=1

1. 假设如下代码的M,N值足够大，为什么该代码的空间局部性较差？（绘图表示）



内存中对二维数组通常是行存储，数组通过列进行访问，相当于stride步长为N，当M,N很大时，命中率极低。

1. 综合题
2. **阅读如下代码，并结合内存分配的结构图以及程序的执行逻辑两个角度阐明为什么该代码会陷入死循环状态**



|  |  |
| --- | --- |
| a[0] |  |
| a[1] | 0 |
| a[2] | 1 |
| a[3] | 2 |
| a[4] | 3 |
| a[5] | 4 |
| a[6] | 5 |
| a[7] | 6 |
| a[8] | 7 |
| a[9] | 8 |
| i |  |

注意，在内存中，局部变量的位置。数组a[10]结束后，是变量i，因此，对a[10]进行操作时，实际上是对变量i进行操作，所以变量i的值被修改为9，再次满足进入循环的条件。I的值一直在9和10之间摆动，while条件总成立，造成了死循环。

1. **如下表，某个缓存结构的参数是（S,E,B,m）=(4,1,2,4)。表中表示了16个地址在缓存中的分布状态。填写表格**

第一步：读取地址为0的内容 无命中

第二步：读取地址为1的内容 命中

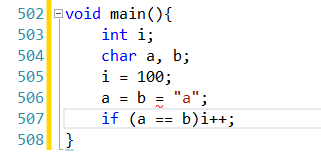
第三步：读取地址为13的内容 无命中

第四步：读取地址为8的内容 无命中

第五步：读取地址为0的内容 无命中

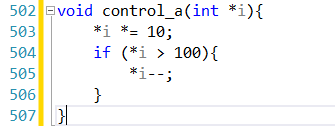
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set | Valid | Tag | Block(0) | Block(1) |
| 0 | Y | 0 （1）（0） | 0（8）（0） | 1（9）（1） |
| 1 |  |  |  |  |
| 2 | Y | 1 | 12 | 13 |
| 3 |  |  |  |  |

1. 改错题
2. 数据初始化



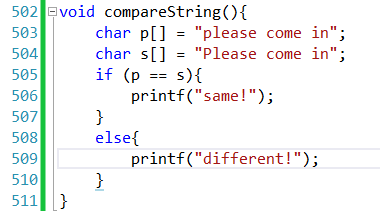
Line506：’a’

1. 对指针i指向的数值控制，如果其值大于100，则减1



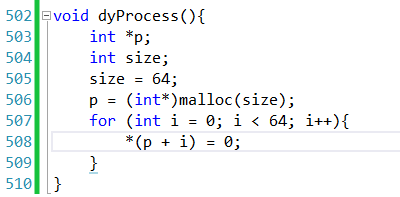
Line505：(\*i)—

1. 字符串比较



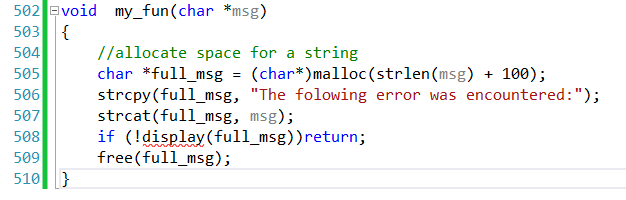
Line505：strcmp(p,s);

1. 动态内存分配



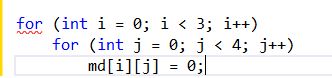
Line506：p=(int\*)malloc(size\*sizeof(int));

1. 初始化一个堆，存放错误信息



Line508：free(full\_msg);

1. 优化题
2. 按照指针地址赋值的方法改写

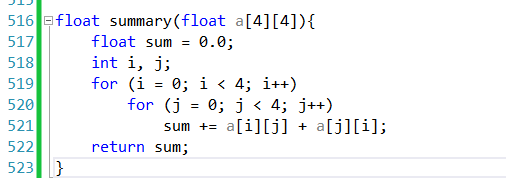


\*(md+i\*3+j)=0;

1. 逻辑操作改写x==y

!(x&~y)

1. 现有如下代码，假设采用直接映射型缓存结构，缓存块的大小为16bytes，而且缓存块有两个组构成，即有32bytes的缓存空间。在不考虑变量i,j,sum的情况下：



填写表格中的命中率

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| i,j | a[i][j] | 缓存分组 | a[i][j]的命中 | a[j][i]的命中 |
| 00 | 0 | 0 | m | h |
| 01 | 4 | 0 | h | m |
| 02 | 8 | 0 | h | m |
| 03 | 12 | 0 | m | m |
| 10 | 16 | 1 | m | h |
| 11 | 20 | 1 | h | h |
| 12 | 24 | 1 | h | m |
| 13 | 28 | 1 | h | m |
| 20 | 32 | 0 | h | m |
| 21 | 36 | 0 | m | m |
| 22 | 40 | 0 | h | h |
| 23 | 44 | 0 | h | m |
| 30 | 48 | 1 | h | m |
| 31 | 52 | 1 | h | m |
| 32 | 56 | 1 | m | m |
| 33 | 60 | 1 | h | h |

计算该段代码的命中率：50%

如何提高该段代码的命中率？改写后的命中率？

Sum+=a[i][j]\*2 75%