通过一段代码来描述内存对齐的现象。

```
struct StructOne {
        char a; //1字节
        double b; //8字节
        int c; //4字节
        short d; //2字节
    } MyStruct1;

struct StructTwo {
        double b; //8字节
        char a; //1字节
        short d; //2字节
        int c; //4字节
    } MyStruct2;
NSLog(@"%lu---%lu--", sizeof(MyStruct1), sizeof(MyStruct2));
```

> 上述代码打印出来的结果为:24,16

为什么相同的结构体,只是交换了变量 ab 在结构体中的顺序他们的大小就改变了呢?这就是“内存对齐”的现象。

内存对齐规则

每个特定平台上的编译器都有自己的默认“对齐系数”(也叫对齐模数)。程序员可以通过预编译命令#pragma pack(n),n=1,2,4,8,16来改变这一系数,其中的n就是你要指定的“对齐系数”。

在了解为什么要进行内存对齐之前,先来看看内存对齐的规则:

- 1.数据成员对齐规则:struct 或 union (以下统称结构体)的数据成员,第一个数据成员A放在偏移为 0 的地方,以后每个数据成员B的偏移为(#pragma pack(指定的数n) 与 该数据成员(也就是 B)的自身长度中较小那个数的整数倍,不够整数倍的补齐。

- 2.数据成员为结构体:如果结构体的数据成员还为结构体,则该数据成员的“自身长度”为其内部最大元素的大小。(struct a 里存有 struct b,b 里有 char,int,double等元素,那 b “自身长度”为 8)

- 3.结构体的整体对齐规则:在数据成员按照上述第一步完成各自对齐之后,结构体

本身也要进行对齐。对齐会将结构体的大小调整为(#pragma pack(指定的数n)与 结构体中的最大长度的数据成员中较小那个的整数倍,不够的补齐。

Xcode 中默认为#pragma pack(8)。如果在代码执行前加一句#pragma pack(1) 时就代表不进行内存对齐,上述代码打印的结果就都为 16。

MyStruct1 的进行对齐后结构为:

```
// Shows the actual memory layout
struct StructOne {
        char a; // 1 byte
        char _pad0[7]; //补齐7字节成为8(随后跟着的 double 大小)的倍数,原则一

        double b; // 8 bytes
        int c; // 4 bytes
        short d; // 2 byte

        char _pad1[2]; // 补齐2字节让结构体的大小成为最大成员大小double(8字节)的倍数,原则二
}
```

为了进行验证,我们通过如下代码打印结构体:

```
long a = (long)&;MyStruct1.a;
long b = (long)&;MyStruct1.b ;
long c = (long)&;MyStruct1.c;
long d = (long)&;MyStruct1.d;
NSLog(@"MyStruct1大小---%lu------", sizeof(MyStruct1));
NSLog(@“内存地址---%ld,%ld,%ld,%ld", a, b, c, d);
输出的结果为:MyStruct1大小——24 ;内存地址--
-4539371424,4539371432,4539371440,4539371444。他们的内存占用符合内存对齐的规则。
char a + char _pad0[7] : 4539371424 // 占用 24-31
double b : 4539371432 // 占用 32-39
int c : 4539371440 // 占用 40-43
short d + char _pad1[2]: 4539371444 // 占用 44-47
```

通过上述规则进行对齐后的 MyStruct1 增加了 9 个字节变为 24 字节。而 MyStruct2 的所有数据成员和结构体本身只有 a 运用内存对齐的规则一增加了一个字节,所以大小为 16 字节。

为什么要进行内存对齐

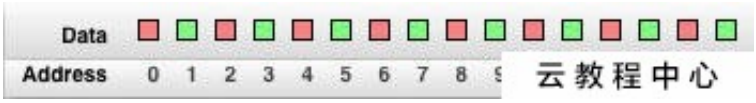
内存对齐应该是编译器的管辖范围。编译器会为程序中的每个数据单元安排在适当的位置上,这个过程对于大部分程序员来说都应该是透明的。但如果你想了解更加底层的秘密,“内存对齐”就不应该对你透明了。

要想掌控这项技术,在了解内存对齐的规则后,还应该知道编译器为什么会进行内存对齐。

很多 CPU(如基于 Alpha,IA-64,MIPS,和 SuperH 体系的)拒绝读取未对齐数据。当一个程序要求这些 CPU 读取未对齐数据时,这时 CPU 会进入异常处理状态并且通知程序不能继续执行。举个例子,在 ARM,MIPS,和 SH 硬件平台上,当操作系统被要求存取一个未对齐数据时会默认给应用程序抛出硬件异常。所以,如果编译器不进行内存对齐,那在很多平台的上的开发将难以进行。

那么,为什么这些 CPU 会拒绝读取未对齐数据?是因为未对齐的数据,会大大降低 CPU 的性能。下边会进行详细的解释。

CPU 存取原理 程序员通常认为内存印象,由一个个的字节组成。

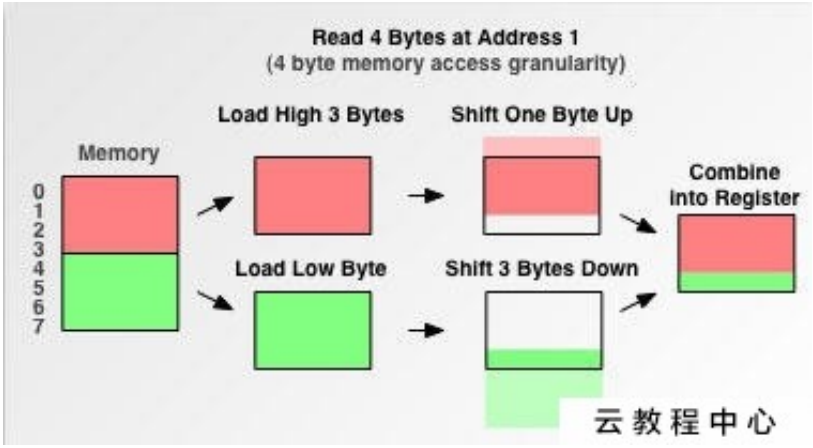


但是,你的 CPU 并不是以字节为单位存取数据的。CPU把内存当成是一块一块的,块的大小可以是2,4,8,16字节大小,因此CPU在读取内存时是一块一块进行读取的。每次内存存取都会产生一个固定的开销,减少内存存取次数将提升程序的性能。所以 CPU 一般会以 2/4/8/16/32 字节为单位来进行存取操作。我们将上述这些存取单位也就是块大小称为(memory access granularity)内存存取粒度。

为了说明内存对齐背后的原理,我们通过一个例子来说明从未地址与对齐地址读取数据的差异。这个例子很简单:在一个存取粒度为 4 字节的内存中,先从地址 0 读取 4 个字节到寄存器,然后从地址 1 读取 4 个字节到寄存器。

当从地址 0 开始读取数据时,是读取对齐地址的数据,直接通过一次读取就能完成。当从地址 1 读取数据时读取的是非对齐地址的数据。需要读取两次数据才能完成。

而且在读取完两次数据后,还要将 0-3 的数据向上偏移 1 字节,将 4-7 的数据向下偏移 3 字节。最后再将两块数据合并放入寄存器。



对一个内存未对齐的数据进行了这么多额外的操作,这对 CPU 的开销很大,大大降低了CPU性能。所以有些处理器才不情愿为你做这些工作。

历史 最初的 68000 处理器的存取粒度是双字节,没有应对非对齐内存地址的电路系统。当遇到非对齐内存地址的存取时,它将抛出一个异常。最初的 Mac OS 并没有妥善处理这个异常,它会直接要求用户重启机器。实在是悲剧。

随后的 680x0 系列,像 68020,放宽了这个的限制,支持了非对齐内存地址存取的相关操作。这解释了为什么一些在 68020 上正常运行的旧软件会在 68000 上崩溃。这也解释了为什么当时一些老 Mac 编程人员会将指针初始化成奇数地址。在最初的 Mac 机器上如果指针在使用前没有被重新赋值成有效地址,Mac 会立即跳到调试器。通常他们通过检查调用堆栈会找到问题所在。

所有的处理器都使用有限的晶体管来完成工作。支持非对齐内存地址的存取操作会消减“晶体管预算”,这些晶体管原本可以用来提升其他模块的速度或者增加新的功能。

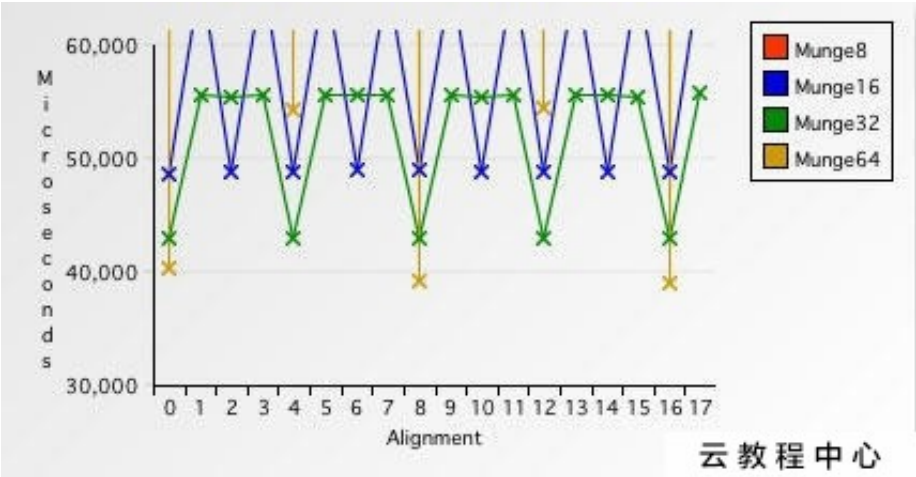
以速度的名义牺牲非对齐内存存取功能的一个例子就是 MIPS。为了提升速度,MIPS 几乎废除了所有的琐碎功能。

PowerPC 各取所长。目前所有的 PowerPC 都在硬件上支持非对齐的 32 位整型的存取。虽然牺牲掉了一部分性能,但这些损失在逐渐减少。

Power 是 1991 年,Apple、IBM、Motorola 组成的 AIM 联盟所发展出的微处理器架构。PowerPC 是整个 AIM 联盟平台的一部分,并且是到目前为止唯一的一部分。但苹果电脑自 2005 年起,将旗下电脑产品转用 Intel CPU。

现今的 PowerPC 处理器缺少对非对齐的 64-bit 浮点型数据的存取的硬件支持。当被要求从非对齐内存读取浮点数时,PowerPC 会抛出异常并让操作系统在软件层面处理内存对齐。软件解决内存对齐要比硬件慢得多。经过 IBM 在 PowerPC 测试,他们效率的差异大概在 4610%。

总结 在 iOS 开发中编译器会帮我们进行内存对齐。所以这些问题都无需考虑。但如果编译器没有提供这些功能,而且 CPU 也不支持读取非对齐数据,CPU 就会抛出硬件异常交给操作系统处理,从而产生 4610% 的差异。如果 CPU 支持读取非对齐数据,相比对齐数据,你还是要承担额外的开销造成的损失。诚然,这种损失绝不会像 4610% 那么大,但还是不能忽略的。



了解了这些后,当我们再声明结构体时就应该合理的安排内部数据的顺序,从而使其占用尽可能小的内存。你也许觉得这并没有什么卵用,但苹果在 Runloop 的源码中就使用了 _padding[3] 来手动对齐内存。

```
struct __CFRunLoopMode {
        CFRuntimeBase _base;
        pthread_mutex_t _lock; /* must have the run loop locked
before locking this */
        CFStringRef _name;
        Boolean _stopped;
        char _padding[3];
```

```
        CFMutableSetRef _sources0;
        CFMutableSetRef _sources1;
        CFMutableArrayRef _observers;
        CFMutableArrayRef _timers;
};
```

> ps:Vc,Vs等编译器默认是#pragma pack(8),所以测试我们的规则会正常;注意gcc默认是#pragma pack(4),并且gcc只支持1,2,4对齐。套用三原则里计算的对齐值是不能大于#pragma pack指定的n值。