数据结构的对齐/齐位

Compass

2010 年 5 月 30 日

这是我在 wiki 上看到的一篇文章，尝试着将它翻译出来了，一方面加深印像，另一方面也算是学习 LaTeX 的练习，如果这篇文章有语句不通顺影响您理解的地方，请参考原文，原文地址在：
http://en.wikipedia.org/wiki/Data_structure_alignment
如果您发现其中的错误后能告诉本人，在下感激不尽，我的邮件是：xjtu.ym@gmail.com。

1 前言

数据结构对齐（Data structure alignment）是数据在内存中分配与访问的方式。具体来说包括两个方面：数据对齐（data alignment）与数据结构填充（data structure padding），这两个方面相互独立但又相互关联。当计算机在特定的内存地址读写数据时，将以机器字长 (word) 为单位进行（32 位系统为 4 字节）。数据对齐（data alignment）意味着数据放置的地址与起始地址的距离为机器字长 (word) 的整数倍，这种做法更利于 CPU 对内存的访问，因此能够提升系统性能。有些数据结构大小不是机器字长的整数倍，因而为了达到数据对齐的目的，需要在这些数据的末尾到下一个有效数据之间填充空白字节，这就是所谓的数据结构填充（data structure padding）。

如果计算机的机器字长是 4 字节，读入数据在内存中地址应当是 4 的整数倍。如果这个条件不能满足，比如数据位于内存的第 14 到第 6 字节，则计算机为了读入这个结构而需要读入两个机器字（每一个机器字 4 字节），然后做相应计算以得到真正需要读入的数据，而更糟糕的结果是计算机可能产生一个齐位错误 (alignment fault)。即使是某个数据在第 14 字节

结束，接下来的数据也应该从第 16 字节开始。这样就需要在这两个数据结构之间额外再插入两个字节，以使接下来的数据结构对齐到第 16 字节。

数据结构对齐 (data structure alignment) 是当今大多数计算机的基本议题，但其实许多计算机语言或者语言的具体实现已经对数据对齐文体做了自动处理。汇编语言以及许多 C/C++ 的具体实现允许用户自行控制数据结构填充 (padding)，这对很多特定环境下的应用特别有用，当然用户的这种控制受一些因素的限制，并不能做到完整而随心所欲的控制。

2 定义

如果我们说一个内存地址 a 以 n 字节对齐，则隐含以下两个条件成立：

1. $n = 2^i$(i 为整数)。
2. $a = n * j$(j 为整数)。

本文将字节视为内存的最小访问单元，因此当我们说内存地址 a 指代内存中的第 a 个字节。一个 n 字节为基础对齐的地址化成二进制之后，其中低 $\log_2 n$ 位为 0。

对于某一次内存访问，如果我们需要访问（读或写）的数据块大小刚好为 n 字节 (n 为任意整数)，而且该数据块的起始地址是 n 字节对齐的（n 与前面取值一致），我们称这次内存访问是对齐 (aligned) 的，反之称为没有对齐的 (misaligned)。根据这个定义可知，对单字节的访问都是对齐 (aligned) 的。

如果一个指针所指数据长度为 n 字节并且这个指针只能指向 n 字节对齐的地址，则称这个指针是对齐的 (aligned)，否之则是未对齐的 (unaligned)。对于指向聚合型 (aggregate) 数据（结构或数组）的指针，如果它所指向的数据集中每一个成员都是对齐的，则称这个指针是对齐的 (aligned)。

在上述定义中，我们假设每一个原生数据长度都是 2 的指数，如果不能满足这个条件（比如 x86 系统上具有 80bit 的浮点成员），则这个数据是否对齐取决于上下文。

3 面临的问题 (problems)

计算机访问内存时是以机器字为单位的。只要机器字长不小于计算机所支持的最大原生数据，则一次对齐的访问 (aligned access) 总是会访问一个单独的机器字，该结论对没有对齐的内存访问不一定成立

如果一个数据的分布在不同的机器字中，则计算机访问该数据时需要分多次进行，这就增加了内存中用于定位和访问指定数据的电路的复杂性。如果储存同一个数据的不同机器字分布在不同的内存页面情况可能更糟，为应付这种情况，处理器需要面对两个潜在的问题：1) 验证这些内存页是否都存在 (present)，2) 在执行指令中的每一次内存访问中处理可能的页面缓冲不命中 (TLB miss) 或者页面错误 (page fault)。

访问一个单独的机器字是一种原子操作，因而对该机器字的读写将会一次性完成，而如果此时其他设备想访问同一块内存就必须等待。上述结论不适应于对多个机器字的非对齐访问 (unaligned accesss)，可能的情况是：某个设备刚完成对第一个机器字的读取，另外一个设备同时改写了两个机器字，于是第一个设备读取第二个机器字后其得到的第一个机器字是改写之前的，而第二个机器字是改写之后的。虽然这种情况不常见，但是其一旦发生将很难调试。

4 架构

4.1 RISC

大多数 RISC 架构处理器在执行载入/存储指令时，如果遇到需要访问未对齐的地址 (misaligned address) 的情况，将产生一个齐位错误 (alignment fault)。因此当操作系统 (operating system) 需要做一个未对齐的访问 (misaligned access) 时，可以以其他指令来模拟进行，比如将其分解为用单字节的读写（这样的操作总是对齐的）操作。

有些架构（如 MIPS）有专门针对未对齐读写的指令。如果一个数据存储在连续的机器字 A，B，C 中（从低位到高位排列），其中机器字 A 中只有低位字节有效，而 C 只有高位字节有效，B 全部有效，这样，为了读取该数据，机器用一条未对齐指令读取 A 中有效的部分，另一条未对齐指令读取 C 中有效的部分。对于写入指令也是一样。

Alpha 架构将未对齐的读写指令分两步进行，首先将高位和低位的机

器字读入分别载入不同的寄存器，然后用与 MIPS 指令类似的的专门指令来读写内存中对应的机器字。一个未对齐的读写指令以将改些后的数据写回内存来结束。带来这种复杂度的原因是原生的 Alpha 架构只能读写 32 位或者 64 位，而这种复杂度往往带来代码膨胀以及性能上的下降。为克服这种限制，人们引入了一种称为 Byte Word Extensions(BWX) 的扩展，包括字节/机器字的读取。

4.2 x86 和 x86_64

x86 架构原生就不要求对内存的访问保持对齐，而且目前仍然是这样，但例外的是 SSE2 指令集，因为他要求数据以 128 位（16 字节）对齐，如果在这种架构的 CPU 上使用对齐的数据，则可以带来较大的性能提升[1]。当然，这种架构也有专门针对未对齐访问的指令，如 MOVDQU。

4.3 兼容性

支持未对齐访问带来的好处是：编译器的作者不需要手工对齐内存，而代价则是访问变慢。提升 RISC 处理器（这种架构本来九世为乐获取最大性能而设计的）性能的途径之一是使载入或者存储的数据都处在机器字的边界上[2]。所以尽管内存普遍以字节为单位寻址，但是若要在 64 位机器上载入一个 32 位的整数或者 64 位的浮点数，需要使载入的数据以 64 位对齐，如果不满足这个条件，处理器可以标记一个错误 (flag a fault)，这样做的后果是执行速度的降低，因为处理器需要做额外的工作来指出哪一个或者哪一些机器字存储了这个数据，然后提取出正确的值。

5 数据结构填充 (Data structure padding)

尽管编译器（或者解释器）在分配内存时通常都将独立的数据对齐到了相应的内存边界，但是数据结构的乘员往往具有不同的对齐要求。为了正确的处理对齐问题，翻译器通常在数据结构中插入匿名数据，以使其成

[1] 译者对这句话不是很理解，到底是这种架构的 CPU 上所有的指令对齐后都能带来较大性能提升还是仅限于 SSE2 指令集？原句是 “While the x86 architecture originally did not require aligned memory access and still works without it, SSE2 instructions on x86 CPUs do require the data to be 128-bit (16-byte) aligned and there can be substantial performance advantages from using aligned data on these architectures.”

[2] 是只能处于机器字的起始地址还是起始/结束地址都行？译者认为是前者

员都能对齐。另外，在整个数据结构的尾部可能也填充有匿名数据成员，这样就能使结构数组的每一个元素都能合适的对齐。

上述填充 (padding) 只有在后续成员对对齐的需求很强烈时才会发生，另一种情况是在结构的末位。我们可以同过调整成员顺序来改变为达到对齐目的而填充的匿名成员的大小。如果将数据成员按照所占内存多少排列（升序或者降序），这样得到的结构对齐时所填充的匿名成员最小。这种最小填充所需要的额外空间始终小于结构做到最大程度对齐时所需要的填充空间。计算最大填充所需要的空间要更复杂一些，但这个最大填充空间始终小于对齐结构所有成员所需空间之和减去 2 倍于对齐一半成员所需的最小空间之和。

C 和 C++ 都不允许编译器重新排列结构成员以节省存储空间，而另外一些语言则允许编译器这样做。对于大多数 C/C++ 编译器，我们可以将数据成员包装 (pack) 到某一特定层次的对齐。比如 “pack(2)” 意味着将大于一个字节的数据成员以 2 字节对其，这样，所有的填充数据都为一个字节长。

这种包装的好处之一是节省内存。如果一个结构包含一个单独的字节和一个四字节的整数，为了对齐编译器需要填充三个额外字节。对于个元素为该结构的数组，包装后 (if they are packed) 将节省 37.5% 的内存空间。虽然这样做降低了访问速度，但不失为一中以时间换取空间的策略。

对结构进行包装除了被用以节省内存空间外，也用来传输时格式化数据。但这样做需要特别小心，传输协议规定了明确的字节顺序 (通常是网络字节顺序 network byte order)，而这种顺序有可能与数据的本地字节顺序不一致[3]。

5.1 填充空间大小的计算 (computing padding)

可用方程 (1) 和方程 (2) 计算数据结构起始位置对齐所需填充的字节数（% 为求模运算）。

$$padding = align - (offset \ \% \ align) \tag{1}$$

$$newoffset = offset + padding = offset + align - (offset \ \% \ align) \tag{2}$$

[3]据译者所知，x86 架构采用的是 little-endian 字节顺序，SPARC 架构，POWER 架构采用的都是 big-endian 字节顺序。

按照上式计算得到结束于 0x59d 的结构为了作到以 4 字节对齐，需要填充 3 个字节。下一个结构将开始于 4 的倍数 0x5a0。

如果齐位 (alignment) 的基准 2^n（n 为任意整数），也可以用方程式 (3) 和 (4) 计算：（其中 & 为按位与， $\sim$ 为按位非）

$$padding = (align + ((offset - 1)\& \sim (align - 1))) - offset \tag{3}$$

$$newoffset = align + ((offset - 1)\& \sim (align - 1)) \tag{4}$$

6 x86 架构上典型的 C 结构对齐

C 中的结构体在内存中是按顺序排列的，因次下列结构的 Data1 成员始终在 Data2 的前面而 Data2 始终在 Data1 的前面。

```
struct MyData
{
    short Data1;
    short Data2;
    short Data3;
};
```

如果 short 数据占用 2 个字节，则该结构中的每一个成员都是以 2 字节对齐的。Data1 相对起始地址偏移量为 0，Data2 偏移量为 2，Data3 偏移量为 4，整个结构占用 6 个字节。

一般来说，结构体的成员都有默认的对齐方式，这意味着，除非程序员明确拒绝，否则这些成员都将对齐到预置的边界上。以下对齐方式对于 32 位 X86 架构系统上的 Microsoft，Borland 和 GNU 编译器都是适应的。

- char 类型（占用一个字节）以 1 字节对齐；
- short 类型（占用二个字节）以 2 字节对齐；
- int 类型（占用四个字节）以 4 字节对齐；
- float 类型（占用四个字节）以 4 字节对齐；
- double 类型（占用八个字节）在 Linux 系统上以 4 字节对齐，在 Windows 系统上以 8 字节对齐；

- long double 类型（占用 12 个字节）在 Linux 系统上以 4 字节对齐；
- 所有的指针类型（占用 4 字节）在 Linux 上以 4 字节对齐 (char*, int*)。

64 位 Linux 系统与 32 位系统的不同之处在于：

- double 类型（占用八个字节）以 8 字节对齐，
- long double 类型（占用 16 个字节）在 Linux 系统上以 16 字节对齐；
- 所有的指针类型（占用 8 字节）以 8 字节对齐。

下面是一个带有不同类型成员的结构体，在编译之前，一共占用 8 个字节：

```
struct MixedData
{
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

编译完成后，编译器将对这个结构体作一些填充 (padding)，以确保每一个成员都能正确的对齐：

```
struct MixedData /* 编译完成后 */
{
    char Data1;
    char Padding1[1]; /* 使接下来的 short 类型以 2 字节对齐 */
    short Data2;
    int Data3;
    char Data4;
    char Padding2[3];
};
```

由此可见，该结构编译后将占用 12 个字节。It is important to note that the last member is padded with the number of bytes required to conform to

the largest type of the structure. In this case 3 bytes are added to the last member to pad the structure to the size of a long word[5].

我们可以调整结构的对齐方式以减少其所占用的内存，这可以通过以下两种方式进行：1) 调整结构体种成员的顺序；2) 改变编译器的对成员的对齐方式 (alignment 也就是前面说的 “packing”)。

```
struct MixedData /* after reordering */
{
    char Data1;
    char Data4; /* reordered */
    short Data2;
    int Data3;
};
```

现在这个结构体在编译前后所占内存大小保持不变，都为 8 个字节。调整顺序后 *Padding1[1]* 被 Data4 替换而消失，而 *Padding2[3]* 已经不再需要，因为该结构体已经对齐到一个 long word。

另一种方法是强制使 *MixedData* 结构体以 1 字节对齐，这将使预处理器不再使用预置的对齐方式，也就不会有插入的填充字节。这种方法没有一种统一的实现方式，有些编译器使用 *#pragma* 指示 *(directive)* 明确不需要预置的对齐，以下是一个例子：

```
#pragma pack(push) /* push current alignment to stack */
#pragma pack(1) /* set alignment to 1 byte boundary */
stuct MyPackedData
{
char Data1;
long Data2;
char Data3;
};
#pragma pack(pop) /* restore original alignment from stack*/
```

上面的这个编译后将占用 6 个字节，而这里用到的 *#pragma* 指示 *(directive)* 可以用在 Microsoft, Borland, GNU 以及许多其他编译器中。

[5]这句话什么意思？哪个 largest type of the structure 怎么来的？a long word 又是指什么？

6.1 默认包装以及 #pragma 包装 (Defaut packing and #pragma pack)

在部分 MS 编译器中，特别是针对 RISC 处理器的版本，默认的包装 (packing，the /Zp directive) 和 #pragma 包装指示 (pack directive) 之间存在着大部分用户都不期望有的联系。

#pragma 包装指示只能用来**减小**工程 (project) 对结构的默认包装大小。这将会带来与所用库头文件之间的互操作性问题，例如库文件指定 #pragma pack(8) 而你将整个工程的设置为一个更小的数，则库文件中的设置就会被忽略。这在 msdn 文档中有详细的阐述。

因为这个缘故，程序员应该始终保持工程的包装值（packing）为默认值 8，以免破坏库文件中的 #pragma 指示，从而使使二进制的结构之间不兼容。

需要特别指出的是，设置/Zp1 将覆盖所有的 #pragma 包装指示，#pragma pack(1) 是唯一的例外。

另一个需要注意的是上述限制并不存在于桌面处理器，比如 x86 架构。

7 分配与缓存对齐的内存 (Allocating memory aligned to cache lines)

将内存分配对齐到缓存是有好出的。如果一个数组被分割后交由不同的线程来处理，如果分割后的子数组没有对齐到缓存将使性能下降。下面有一个将分配的内存（8 个 double 元素组成的数组）对齐到 64 字节缓存的例子：

```
#include <stdlib.h>
double *foo(void) {
   double *var;
   int     ok;

   ok = posix_memalign(&var, 64, 8);

   if(ok != 0)
     return NULL;
```

```
    return var;
}
```