WIKIPEDIA

Data structure alignment

Data structure alignment refers to the way data is arranged and accessed in computer memory. It consists of three separate but related issues: **data alignment**, **data structure padding**, and **packing**.

The CPU in modern computer hardware performs reads and writes to memory most efficiently when the data is *naturally aligned*, which generally means that the data address is a multiple of the data size. *Data alignment* refers to aligning elements according to their natural alignment. To ensure natural alignment, it may be necessary to insert some *padding* between structure elements or after the last element of a structure.

Although data structure alignment is a fundamental issue for all modern computers, many computer languages and computer language implementations handle data alignment automatically. Ada,[1][2] PL/I,[3] Pascal,[4] certain C and C++ implementations, D,[5] Rust,[6] and assembly language allow at least partial control of data structure padding, which may be useful in certain special circumstances.

Contents

Definitions

A memory address *a*, is said to be *n-byte aligned* when *a* is a multiple of *n* bytes (where *n* is a power of 2). In this context a byte is the smallest unit of memory access, i.e. each memory address specifies a different byte. An *n*-byte aligned address would have a minimum of $log_2(n)$ least-significant zeros when expressed in binary.

The alternate wording *b-bit aligned* designates a *b/8 byte aligned* address (ex. 64-bit aligned is 8 bytes aligned).

A memory access is said to be *aligned* when the datum being accessed is *n* bytes long and the datum address is *n*-byte aligned. When a memory access is not aligned, it is said to be *misaligned*. Note that by definition byte memory accesses are always aligned.

A memory pointer that refers to primitive data that is *n* bytes long is said to be *aligned* if it is only allowed to contain addresses that are *n*-byte aligned, otherwise it is said to be *unaligned*. A memory pointer that refers to a data aggregate (a data structure or array) is *aligned* if (and only if) each primitive datum in the aggregate is aligned.

Note that the definitions above assume that each primitive datum is a power of two bytes long. When this is not the case (as with 80-bit floating-point on x86) the context influences the conditions where the datum is considered aligned or not.

Data structures can be stored in memory on the stack with a static size known as *bounded* or on the heap with a dynamic size known as *unbounded*.

Problems

A computer accesses memory by a single memory word at a time. As long as the memory word size is at least as large as the largest primitive data type supported by the computer, aligned accesses will always access a single memory word. This may not be true for misaligned data accesses.

If the highest and lowest bytes in a datum are not within the same memory word the computer must split the datum access into multiple memory accesses. This requires a lot of complex circuitry to generate the memory accesses and coordinate them. To handle the case where the memory words are in different memory pages the processor must either verify that both pages are present before executing the instruction or be able to handle a TLB miss or a page fault on any memory access during the instruction execution.

When a single memory word is accessed the operation is atomic, i.e. the whole memory word is read or written at once and other devices must wait until the read or write operation completes before they can access it. This may not be true for unaligned accesses to multiple memory words, e.g. the first word might be read by one device, both words written by another device and then the second word read by the first device so that the value read is neither the original value nor the updated value. Although such failures are rare, they can be very difficult to identify.

Data structure padding

Although the compiler (or interpreter) normally allocates individual data items on aligned boundaries, data structures often have members with different alignment requirements. To maintain proper alignment the translator normally inserts additional unnamed data members so that each member is properly aligned. In addition the data structure as a whole may be padded with a final unnamed member. This allows each member of an array of structures to be properly aligned.

Padding is only inserted when a structure member is followed by a member with a larger alignment requirement or at the end of the structure. By changing the ordering of members in a structure, it is possible to change the amount of padding required to maintain alignment. For example, if members are sorted by descending alignment requirements a minimal amount of padding is required. The minimal amount of padding required is always less than the largest alignment in the structure. Computing the maximum amount of padding required is more complicated, but is always less than the sum of the alignment requirements for all members minus twice the sum of the alignment requirements for the least aligned half of the structure members.

Although C and C++ do not allow the compiler to reorder structure members to save space, other languages might. It is also possible to tell most C and C++ compilers to "pack" the members of a structure to a certain level of alignment, e.g. "pack(2)" means align data members larger than a byte to a two-byte boundary so that any padding members are at most one byte long.

One use for such "packed" structures is to conserve memory. For example, a structure containing a single byte and a four-byte integer would require three additional bytes of padding. A large array of such structures would use 37.5% less memory if they are packed, although accessing each structure might take longer. This compromise may be considered a form of space–time tradeoff.

Although use of "packed" structures is most frequently used to conserve memory space, it may also be used to format a data structure for transmission using a standard protocol. However, in this usage, care must also be taken to ensure that the values of the struct members are stored with the endianness required by the protocol (often network byte order), which may be different from the endianness used natively by the host machine.

Computing padding

The following formulas provide the number of padding bytes required to align the start of a data structure (where *mod* is the modulo operator):

```
padding = (align - (offset mod align)) mod align
aligned = offset + padding
        = offset + ((align - (offset mod align)) mod align)
```

For example, the padding to add to offset 0x59d for a 4-bit aligned structure is 3. The structure will then start at 0x5a0, which is a multiple of 4. However, when the alignment of *offset* is already equal to that of *align*, the second modulo in *(align - (offset mod align)) mod align* will return zero, therefore the original value is left unchanged.

Since the alignment is by definition a power of two, the modulo operation can be reduced to a bitwise boolean AND operation.

The following formulas produce the aligned offset (where *&* is a bitwise AND and ~ a bitwise NOT):

```
padding = (align - (offset & (align - 1))) & (align - 1)
        = (-offset & (align - 1))
aligned = (offset + (align - 1)) & ~(align - 1)
        = (offset + (align - 1)) & -align
```

Typical alignment of C structs on x86

Data structure members are stored sequentially in memory so that, in the structure below, the member Data1 will always precede Data2; and Data2 will always precede Data3:

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

If the type "short" is stored in two bytes of memory then each member of the data structure depicted above would be 2-byte aligned. Data1 would be at offset 0, Data2 at offset 2, and Data3 at offset 4. The size of this structure would be 6 bytes.

The type of each member of the structure usually has a default alignment, meaning that it will, unless otherwise requested by the programmer, be aligned on a pre-determined boundary. The following typical alignments are valid for compilers from Microsoft (Visual C++), Borland/CodeGear (C++Builder), Digital Mars (DMC), and GNU (GCC) when compiling for

32-bit x86:

- A **char** (one byte) will be 1-byte aligned.
- A **short** (two bytes) will be 2-byte aligned.
- An **int** (four bytes) will be 4-byte aligned.
- A **long** (four bytes) will be 4-byte aligned.
- A **float** (four bytes) will be 4-byte aligned.
- A **double** (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux (8-byte with *-malign-double* compile time option).
- A **long long** (eight bytes) will be 4-byte aligned.
- A **long double** (ten bytes with C++Builder and DMC, eight bytes with Visual C++, twelve bytes with GCC) will be 8-byte aligned with C++Builder, 2-byte aligned with DMC, 8-byte aligned with Visual C++, and 4-byte aligned with GCC.
- Any **pointer** (four bytes) will be 4-byte aligned. (e.g.: char*, int*)

The only notable differences in alignment for an LP64 64-bit system when compared to a 32-bit system are:

- A **long** (eight bytes) will be 8-byte aligned.
- A **double** (eight bytes) will be 8-byte aligned.
- A **long long** (eight bytes) will be 8-byte aligned.
- A **long double** (eight bytes with Visual C++, sixteen bytes with GCC) will be 8-byte aligned with Visual C++ and 16-byte aligned with GCC.
- Any **pointer** (eight bytes) will be 8-byte aligned.

Some data types are dependent on the implementation.

Here is a structure with members of various types, totaling **8 bytes** before compilation:

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

After compilation the data structure will be supplemented with padding bytes to ensure a proper alignment for each of its members:

```
struct MixedData  /* After compilation in 32-bit x86 machine */
{
    char Data1; /* 1 byte */
    char Padding1[1]; /* 1 byte for the following 'short' to be aligned on a 2 byte boundary
assuming that the address where structure begins is an even number */
    short Data2; /* 2 bytes */
    int Data3;  /* 4 bytes - largest structure member */
    char Data4; /* 1 byte */
    char Padding2[3]; /* 3 bytes to make total size of the structure 12 bytes */
};
```

The compiled size of the structure is now 12 bytes. It is important to note that the last member is padded with the number of bytes required so that the total size of the structure should be a multiple of the largest alignment of any structure member (alignment(int) in this case, which = 4 on linux-32bit/gcc).

In this case 3 bytes are added to the last member to pad the structure to the size of a 12 bytes (alignment(int) × 3).

```
struct FinalPad {
  float x;
  char n[1];
};
```

In this example the total size of the structure `sizeof(FinalPad) == 8`, not 5 (so that the size is a multiple of 4 (alignment of float)).

```
struct FinalPadShort {
  short s;
  char n[3];
};
```

In this example the total size of the structure `sizeof(FinalPadShort) == 6`, not 5 (not 8 either) (so that the size is a multiple of 2 (alignment(short) = 2 on linux-32bit/gcc)).

It is possible to change the alignment of structures to reduce the memory they require (or to conform to an existing format) by reordering structure members or changing the compiler's alignment (or "packing") of structure members.

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

The compiled size of the structure now matches the pre-compiled size of **8 bytes**. Note that `Padding1[1]` has been replaced (and thus eliminated) by `Data4` and `Padding2[3]` is no longer necessary as the structure is already aligned to the size of a long word.

The alternative method of enforcing the `MixedData` structure to be aligned to a one byte boundary will cause the pre-processor to discard the pre-determined alignment of the structure members and thus no padding bytes would be inserted.

While there is no standard way of defining the alignment of structure members, some compilers use `#pragma` directives to specify packing inside source files. Here is an example:

```
#pragma pack(push)  /* push current alignment to stack */
#pragma pack(1)     /* set alignment to 1 byte boundary */

struct MyPackedData
{
    char Data1;
    long Data2;
    char Data3;
};

#pragma pack(pop)   /* restore original alignment from stack */
```

This structure would have a compiled size of **6 bytes** on a 32-bit system. The above directives are available in compilers from Microsoft,[7] Borland, GNU,[8] and many others.

Another example:

```
struct MyPackedData
{
    char Data1;
    long Data2 __attribute__((packed));
    char Data3;
};
```

Default packing and `#pragma pack`

On some Microsoft compilers, particularly for the RISC processor, there is an unexpected relationship between project default packing (the /Zp directive) and the `#pragma pack` directive. The `#pragma pack` directive can only be used to **reduce** the packing size of a structure from the project default packing.[9] This leads to interoperability problems with library headers which use, for example, `#pragma pack(8)`, if the project packing is smaller than this. For this reason, setting the project packing to any value other than the default of 8 bytes would break the `#pragma pack` directives used in library headers and result in binary incompatibilities between structures. This limitation is not present when compiling for x86.

Allocating memory aligned to cache lines

It would be beneficial to allocate memory aligned to cache lines. If an array is partitioned for more than one thread to operate on, having the sub-array boundaries unaligned to cache lines could lead to performance degradation. Here is an example to allocate memory (double array of size 10) aligned to cache of 64 bytes.

```
#include <stdlib.h>
double *foo(void) {
   double *var;//create array of size 10
   int     ok;

   ok = posix_memalign((void**)&var, 64, 10*sizeof(double));

   if(ok != 0)
     return NULL;

   return var;
}
```

Hardware significance of alignment requirements

Alignment concerns can affect areas much larger than a C structure when the purpose is the efficient mapping of that area through a hardware address translation mechanism (PCI remapping, operation of a MMU).

For instance, on a 32-bit operating system, a 4 KB page is not just an arbitrary 4 KB chunk of data. Instead, it is usually a region of memory that's aligned on a 4 KB boundary. This is because aligning a page on a page-sized boundary lets the hardware map a virtual address to a physical address by substituting the higher bits in the address, rather than doing complex arithmetic.

Example: Assume that we have a TLB mapping of virtual address 0x2cfc7000 to physical address 0x12345000. (Note that both these addresses are aligned at 4 KB boundaries.) Accessing data located at virtual address va=0x2cfc7abc causes a TLB resolution of 0x2cfc7 to 0x12345 to issue a physical access to pa=0x12345abc. Here, the 20/12-bit split luckily matches the hexadecimal representation split at 5/3 digits. The hardware can implement this translation by simply combining the first 20 bits of the physical address (0x12345) and the last 12 bits of the virtual address (0xabc). This is also referred to as virtually indexed (abc) physically tagged (12345).

A block of data of size 2^(n+1)-1 always has one sub-block of size 2^n aligned on 2^n bytes.

This is how a dynamic allocator that has no knowledge of alignment, can be used to provide aligned buffers, at the price of a factor two in space loss.

```
// Example: get a 12-bit aligned 4 KBytes buffer with malloc()
```

```
// unaligned pointer to large area
void *up = malloc((1 << 13) - 1);
// well-aligned pointer to 4 KBytes
void *ap = alignonext(up, 12);
```

where alignonext(p, r) works by adding an aligned increment, then clearing the r least significant bits of p. A possible implementation is

```
// Assume `uint32_t p, bits;` for readability
#define alignto(p, bits)      (((p) >> bits) << bits)
#define alignonext(p, bits)  alignto(((p) + (1 << bits) - 1), bits)
```

References

1. "Ada Representation Clauses and Pragmas" (http://docs.adacore.com/gnat_rm-docs/html/gnat_rm/gnat_rm/representation_clauses_and_pragmas.html). *GNAT Reference Manual 7.4.0w documentation*. Retrieved 2015-08-30.
2. "F.8 Representation Clauses". *SPARCompiler Ada Programmer's Guide* (http://docs.oracle.com/cd/E19957-01/802-3641/802-3641.pdf) (PDF). Retrieved 2015-08-30.
3. *IBM System/360 Operating System PL/I Language Specifications* (http://www.bitsavers.org/pdf/ibm/360/pli/C28-6571-3_PL_I_Language_Specifications_Jul66.pdf) (PDF). IBM. July 1966. pp. 55–56. C28-6571-3.
4. Niklaus Wirth (July 1973). "The Programming Language Pascal (Revised Report)" (http://www.standardpascal.com/The_Programming_Language_Pascal_1973.pdf) (PDF). p. 12.
5. "Attributes - D Programming Language: Align Attribute" (http://dlang.org/attribute.html#align). Retrieved 2012-04-13.
6. "The Rustonomicon - Alternative Representations" (https://doc.rust-lang.org/nomicon/other-reprs.html). Retrieved 2016-06-19.
7. pack (https://msdn.microsoft.com/en-us/library/2e70t5y1(v=vs.140).aspx)
8. 6.58.8 Structure-Packing Pragmas (https://gcc.gnu.org/onlinedocs/gcc-4.8.4/gcc/Structure-Packing-Pragmas.html)
9. "Working with Packing Structures" (http://msdn.microsoft.com/en-us/library/ms253935.aspx). *MSDN Library*. Microsoft. 2007-07-09. Retrieved 2011-01-11.

Further reading

- Bryant, Randal E.; David, O'Hallaron (2003). *Computer Systems: A Programmer's Perspective* (http://csapp.cs.cmu.edu/) (2003 ed.). Upper Saddle River, NJ: Pearson Education. ISBN 0-13-034074-X

External links

- IBM developerWorks article on data alignment (http://www.ibm.com/developerworks/library/pa-dalign/)
- Article on data alignment and performance (https://fylux.github.io/2017/07/11/Memory_Alignment/)
- MSDN article on data alignment (http://msdn2.microsoft.com/en-us/library/ms253949.aspx)
- Article on data alignment and data portability (http://www.codesynthesis.com/~boris/blog/2009/04/06/cxx-data-alignment-portability/)
- Byte Alignment and Ordering (http://www.eventhelix.com/RealtimeMantra/ByteAlignmentAndOrdering.htm)