UNALIGNED MEMORY ACCESSES

Linux runs on a wide variety of architectures which have varying behaviour when it comes to memory access. This document presents some details about unaligned accesses, why you need to write code that doesn't cause them, and how to write such code!

The definition of an unaligned access

Unaligned memory accesses occur when you try to read N bytes of data starting from an address that is not evenly divisible by N (i.e. addr % N != 0). For example, reading 4 bytes of data from address 0x10004 is fine, but reading 4 bytes of data from address 0x10005 would be an unaligned memory access.

The above may seem a little vague, as memory access can happen in different ways. The context here is at the machine code level: certain instructions read or write a number of bytes to or from memory (e.g. movb, movw, movl in x86 assembly). As will become clear, it is relatively easy to spot C statements which will compile to multiple-byte memory access instructions, namely when dealing with types such as u16, u32 and u64.

Natural alignment

The rule mentioned above forms what we refer to as natural alignment: When accessing N bytes of memory, the base memory address must be evenly divisible by N, i.e. addr % N == 0.

When writing code, assume the target architecture has natural alignment requirements.

In reality, only a few architectures require natural alignment on all sizes of memory access. However, we must consider ALL supported architectures; writing code that satisfies natural alignment requirements is the easiest way to achieve full portability.

Why unaligned access is bad

The effects of performing an unaligned memory access vary from architecture to architecture. It would be easy to write a whole document on the differences here; a summary of the common scenarios is presented below:

- Some architectures are able to perform unaligned memory accesses transparently, but there is usually a significant performance cost.
- Some architectures raise processor exceptions when unaligned accesses happen. The exception handler is able to correct the unaligned access, at significant cost to performance.
- Some architectures raise processor exceptions when unaligned accesses happen, but the exceptions do not contain enough information for the

   unaligned access to be corrected.
 - Some architectures are not capable of unaligned memory access, but will
   silently perform a different memory access to the one that was requested,
   resulting in a subtle code bug that is hard to detect!

It should be obvious from the above that if your code causes unaligned memory accesses to happen, your code will not work correctly on certain platforms and will cause performance problems on others.

Code that does not cause unaligned access

At first, the concepts above may seem a little hard to relate to actual coding practice. After all, you don't have a great deal of control over memory addresses of certain variables, etc.

Fortunately things are not too complex, as in most cases, the compiler ensures that things will work for you. For example, take the following structure:

```
	struct foo {
		u16 field1;
		u32 field2;
		u8 field3;
	};
```

Let us assume that an instance of the above structure resides in memory starting at address 0x10000. With a basic level of understanding, it would not be unreasonable to expect that accessing field2 would cause an unaligned access. You'd be expecting field2 to be located at offset 2 bytes into the structure, i.e. address 0x10002, but that address is not evenly divisible by 4 (remember, we're reading a 4 byte value here).

Fortunately, the compiler understands the alignment constraints, so in the above case it would insert 2 bytes of padding in between field1 and field2. Therefore, for standard structure types you can always rely on the compiler to pad structures so that accesses to fields are suitably aligned (assuming you do not cast the field to a type of different length).

Similarly, you can also rely on the compiler to align variables and function parameters to a naturally aligned scheme, based on the size of the type of the variable.

At this point, it should be clear that accessing a single byte (u8 or char) will never cause an unaligned access, because all memory addresses are evenly divisible by one.

On a related topic, with the above considerations in mind you may observe that you could reorder the fields in the structure in order to place fields where padding would otherwise be inserted, and hence reduce the overall resident memory size of structure instances. The optimal layout of the above example is:

```
	struct foo {
		u32 field2;
```

```
		u16 field1;
		u8 field3;
	};
```

For a natural alignment scheme, the compiler would only have to add a single byte of padding at the end of the structure. This padding is added in order to satisfy alignment constraints for arrays of these structures.

Another point worth mentioning is the use of __attribute__((packed)) on a structure type. This GCC-specific attribute tells the compiler never to insert any padding within structures, useful when you want to use a C struct to represent some data that comes in a fixed arrangement 'off the wire'.

You might be inclined to believe that usage of this attribute can easily lead to unaligned accesses when accessing fields that do not satisfy architectural alignment requirements. However, again, the compiler is aware of the alignment constraints and will generate extra instructions to perform the memory access in a way that does not cause unaligned access. Of course, the extra instructions obviously cause a loss in performance compared to the non-packed case, so the packed attribute should only be used when avoiding structure padding is of importance.

Code that causes unaligned access

With the above in mind, let's move onto a real life example of a function that can cause an unaligned memory access. The following function adapted from include/linux/etherdevice.h is an optimized routine to compare two ethernet MAC addresses for equality.

```
unsigned int compare_ether_addr(const u8 *addr1, const u8 *addr2)
{
	const u16 *a = (const u16 *) addr1;
	const u16 *b = (const u16 *) addr2;
	return ((a[0] ^ b[0]) | (a[1] ^ b[1]) | (a[2] ^ b[2])) != 0;
}
```

In the above function, the reference to a[0] causes 2 bytes (16 bits) to be read from memory starting at address addr1. Think about what would happen if addr1 was an odd address such as 0x10003. (Hint: it'd be an unaligned access.)

Despite the potential unaligned access problems with the above function, it is included in the kernel anyway but is understood to only work on 16-bit-aligned addresses. It is up to the caller to ensure this alignment or not use this function at all. This alignment-unsafe function is still useful as it is a decent optimization for the cases when you can ensure alignment, which is true almost all of the time in ethernet networking context.

Here is another example of some code that could cause unaligned accesses:

```
	void myfunc(u8 *data, u32 value)
	{
		[...]
		*((u32 *) data) = cpu_to_le32(value);
```

```
        [...]
}
```

This code will cause unaligned accesses every time the data parameter points to an address that is not evenly divisible by 4.

In summary, the 2 main scenarios where you may run into unaligned access problems involve:
 1. Casting variables to types of different lengths
 2. Pointer arithmetic followed by access to at least 2 bytes of data

Avoiding unaligned accesses

The easiest way to avoid unaligned access is to use the get_unaligned() and put_unaligned() macros provided by the <asm/unaligned.h> header file.

Going back to an earlier example of code that potentially causes unaligned access:

```
void myfunc(u8 *data, u32 value)
{
        [...]
        *((u32 *) data) = cpu_to_le32(value);
        [...]
}
```

To avoid the unaligned memory access, you would rewrite it as follows:

```
void myfunc(u8 *data, u32 value)
{
        [...]
        value = cpu_to_le32(value);
        put_unaligned(value, (u32 *) data);
        [...]
}
```

The get_unaligned() macro works similarly. Assuming 'data' is a pointer to memory and you wish to avoid unaligned access, its usage is as follows:

```
u32 value = get_unaligned((u32 *) data);
```

These macros work for memory accesses of any length (not just 32 bits as in the examples above). Be aware that when compared to standard access of aligned memory, using these macros to access unaligned memory can be costly in terms of performance.

If use of such macros is not convenient, another option is to use memcpy(), where the source or destination (or both) are of type u8* or unsigned char*. Due to the byte-wise nature of this operation, unaligned accesses are avoided.

Alignment vs. Networking

On architectures that require aligned loads, networking requires that the IP header is aligned on a four-byte boundary to optimise the IP stack. For regular ethernet hardware, the constant NET_IP_ALIGN is used. On most architectures this constant has the value 2 because the normal ethernet header is 14 bytes long, so in order to get proper alignment one needs to DMA to an address which can be expressed as 4*n + 2. One notable exception here is powerpc which defines NET_IP_ALIGN to 0 because DMA to unaligned addresses can be very expensive and dwarf the cost of unaligned loads.

For some ethernet hardware that cannot DMA to unaligned addresses like 4*n+2 or non-ethernet hardware, this can be a problem, and it is then required to copy the incoming frame into an aligned buffer. Because this is unnecessary on architectures that can do unaligned accesses, the code can be made dependent on CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS like so:

```
#ifdef CONFIG_HAVE_EFFICIENT_UNALIGNED_ACCESS
	skb = original skb
#else
	skb = copy skb
#endif
```

--

Authors: Daniel Drake <dsd@gentoo.org>,
         Johannes Berg <johannes@sipsolutions.net>
With help from: Alan Cox, Avuton Olrich, Heikki Orsila, Jan Engelhardt, Kyle McMartin, Kyle Moffett, Randy Dunlap, Robert Hancock, Uli Kunitz, Vadim Lobanov