Memory alignment fun

July 17, 2018 forwolk Uncategorized 0

Let's start this post with a little quiz. What do you think is the size of this data structure?

```
public struct StructWithIncorrectLayout
{
public short hitpoints;
public int x;
public short attackPower;
public int y;
}
```

If you said 12 bytes, you were wrong. It's actually 16!

To explain this phenomenon, we will have to introduce two concepts. **Memory alignment** and **padding**.

It turns out for the processor to be able to efficiently read memory, memory must be properly aligned.

A processor can usually execute read cycles only on the addresses, divisible by 4 or 8.

SEARCH …

RECENT POSTS

Memory alignment fun
Unity: Memory profiling
IL2CPP optimisations
Dictionary: Best practices

RECENT COMMENTS

ARCHIVES

July 2018
June 2018
May 2018

CATEGORIES

Uncategorized

META

Log in
Entries RSS
Comments RSS
WordPress.org

For instance, some processors would not even allow you to write a short into an odd address, resulting in an exception. Modern processors (excluding some old ARM processors) while tolerating **memory misalignment**, will have to issue more read instructions to be able to consume data. The processor will execute 2 read cycles using different memory offsets and then via bitwise operations present you with a final value. To avoid that there are certain requirements to data alignment of different types. For instance, **char** has to be 1-byte aligned, **short** 2-byte aligned and **int** 4-bytes aligned. Let's bring up the example of what happens when the data is misaligned using the class defined above.

Read cycles	0				1				2			
Address	0	1	2	3	4	5	6	7	8	9	10	11
Bytes occupied												
Data	HP		X				AP		Y			

As you can see, to retrieve the value of *X* field, a processor has to execute two read operations on two different offsets. Because of this, the default strategy of the compilers is to sacrifice memory for the performance. Compilers will insert **padding** between elements of your struct so that any of its members can be read in a single read cycle. Knowing about this we can illustrate our original example above.

Read cycles	0				1				2				3			
Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bytes occupied																
Data	HP		Padding		X				AP		Padding		Y			

As you can see there is now padding inserted by the compiler and even though the structure occupies more space all the memory lookups will be completed in one read cycle.

C# has a way of controlling how data is laid out via **StructLayout** attribute. If the attribute is not explicitly specified, compiler behavior will be similar to **LayoutKind.Sequential** mode. It means that memory layout will be defined by the order in which fields are declared. Automatic padding will be generated for every field which is smaller than the biggest in a struct. In the case above it means the following:

1. Int is the largest type in our structure
2. It consists of 4 bytes
3. Thus, every short in our struct automatically gets plus 2 bytes.

This is how we get our 16 bytes instead of 12.

We can go down to 12 bytes and still keep the memory aligned. We just have to rewrite our struct the following way:

```
public struct StructWithCorrectLayout
{
public int x;
public int y;
public short hitpoints;
public short attackPower;
}
```

The size of the struct is now 12 bytes, as the **shorts** are coming after **ints**, forming an aligned 4 bytes value. That's a strategy you can opt for if you want to reduce the memory footprint of your structs. Declare the fields in the descending order of their sizes.

Theoretically, this is how **LayoutKind.Auto** mode should behave but in my case, it was still making the 16 bytes size structure. Please check it out for yourself.

If you don't care about **memory alignment** (and you've tested that your target ARM processors don't care about it either) but memory itself is a constraint for you, you can try to setup padding manually, further reducing the memory footprint of your application. By specifying struct attribute this way **StructLayout(LayoutKind.Sequential, Pack = 1)**, you are saying that there should be no padding between any elements of the struct. If you had set Pack = 0, default padding would be used instead.

Present situation:

Now that we've discussed ways of controlling memory alignment in C#, let's take a look at the impact of using misaligned memory on the performance. According to this article, memory alignment performance penalty is a thing of the past for x86 processors and can be experienced only in very specific cases. Intel documents describing **Nehalem** processor architecture (which hit the market in 2008), state that misaligned memory access was heavily optimized and does not incur the performance penalty it used to. Our own benchmarks show that this seems to be true. We've run tests on MacBook Pro and Samsung Galaxy S6 and saw no significant difference in operations on aligned and misaligned

memory. If you are curious, please check the benchmarks for yourself.

At the same time, there is a case when having misaligned memory results in a very poor performance. That is when you use atomic operations. In C# they are wrapped in an Interlocked class. Our benchmarks show that executing Interlocked.Exchange on an unaligned memory may be **52** times slower than calling it on an aligned one. This happens if a variable got physically allocated between two cache lines.

Another consequence of using non-aligned memory is that the read / write operations are no longer atomic. This means that if you are reading and writing to the same variable from 2 different threads without synchronization constructs you may experience "torn reads". This means that when you write 16 to a variable, upon reading from it you may suddenly get 18405345.

In essence:
1. If you care about memory consumption of your structs, make sure their fields are defined in the proper order.
2. If you want to reduce memory consumption even further (potentially introducing issues in your code on certain processors), make your structs ignore memory alignment requirements via special attributes.
3. If you are using atomic operations, always use aligned memory.

Benchmark sources are available **here** and **here**.

This article was co-authored by Anton Trukhan and Lenar Sharipov.

« **PREVIOUS**
Unity: Memory profiling