Subject: Optimized memcmp for x86
Posted by mirek on Fri, 22 Feb 2008 10:07:16 GMT
View Forum Message <> Reply to Message

Well, this code seems to run 20% faster than intrinsic GCC memcmp on x86-64:

```
#ifdef COMPILER_GCC
inline dword _byteswap_ulong(dword x)
{
 asm("bswap %0" : "=r" (x) : "0" (x));
 return x;
}

inline uint64 _byteswap_uint64(uint64 x)
{
 asm("bswap %0" : "=r" (x) : "0" (x));
 return x;
}

inline word _byteswap_ushort(word x)
{
 __asm__("xchgb %b0,%h0" : "=q" (x) :  "0" (x));
 return x;
}
#endif

int MemCmp(const char *a, const char *b, size_t len)
{
 if(((size_t)a & 3) | ((size_t)b & 3))
  return memcmp(a, b, len);
 const dword *x = (dword *)a;
 const dword *y = (dword *)b;
 const dword *e = x + (len >> 2);
 while(x < e) {
  if(*x != *y)
   return int(_byteswap_ulong(*x) - _byteswap_ulong(*y));
  x++;
  y++;
 }
 if(len & 2)
  if(*(word *)x != *(word *)y)
   return int(_byteswap_ushort(*(word *)x) - _byteswap_ushort(*(word *)y));
 if(len & 1)
  return int(*((byte *)x + 2)) - int(*((byte *)y + 2));
 return 0;
}
```

(Obviously, when both areas are dword aligned, but that happens a lot...).

Mirek

Subject: Re: Optimized memcmp for x86
Posted by mr_ped on Fri, 22 Feb 2008 18:08:11 GMT
View Forum Message <> Reply to Message

luzr wrote on Fri, 22 February 2008 11:07

```
 if(len & 2)
  if(*(word *)x != *(word *)y)
   return int(_byteswap_ushort(*(word *)x) - _byteswap_ushort(*(word *)y));
 if(len & 1)
  return int(*((byte *)x + 2)) - int(*((byte *)y + 2));
```

I don't get this end.

switch (len & 3)
0: it looks ok to me.
1: the return int(*((byte *)x)) - int(*((byte *)y)); should be returned?
2: looks ok
3: looks ok

I would maybe try masking out unused bytes, but that would lead to read out of buffer boundary. Is it safe?
I mean something like this

```
   ...
   const static dword masks[4] = { 0x00000000, 0x000000FF, 0x0000FFFF, 0x00FFFFFF };
//Intel-like endian only!
   return int(_byteswap_ulong(*x & masks[len&3]) - _byteswap_ulong(*y & masks[len&3]));
```

I'm not sure I got the byteswap purpose correctly, but I think I got, so my code is probably ok (but I didn't test it).

Of course it reads beyond buffer end, so you need to know it will not raise exception or crash the application on target platform.

Subject: Re: Optimized memcmp for x86

Posted by mr_ped on Fri, 22 Feb 2008 18:52:54 GMT
View Forum Message <> Reply to Message

oh, and a very very minor optimization (which is probably done by compiler itself anyway?)

The line:
if(((size_t)a & 3) | ((size_t)b & 3))
can be:
if((((size_t)a) | ((size_t)b)) & 3)
(one bit-and less)

Subject: Re: Optimized memcmp for x86
Posted by mirek on Sat, 23 Feb 2008 15:15:21 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Fri, 22 February 2008 13:08luzr wrote on Fri, 22 February 2008 11:07

```
 if(len & 2)
  if(*(word *)x != *(word *)y)
   return int(_byteswap_ushort(*(word *)x) - _byteswap_ushort(*(word *)y));
 if(len & 1)
  return int(*((byte *)x + 2)) - int(*((byte *)y + 2));
```

I don't get this end.

switch (len & 3)
0: it looks ok to me.
1: the return int(*((byte *)x)) - int(*((byte *)y)); should be returned?
2: looks ok
3: looks ok

I would maybe try masking out unused bytes, but that would lead to read out of buffer boundary. Is it safe?
I mean something like this

```
   ...
   const static dword masks[4] = { 0x00000000, 0x000000FF, 0x0000FFFF, 0x00FFFFFF };
//Intel-like endian only!
   return int(_byteswap_ulong(*x & masks[len&3]) - _byteswap_ulong(*y & masks[len&3]));
```

I'm not sure I got the byteswap purpose correctly, but I think I got, so my code is probably ok (but I didn't test it).

Of course it reads beyond buffer end, so you need to know it will not raise exception or crash the

application on target platform.

Yes, you are right, len&1 was wrong... (now is not it nice to have the code checked by posting here?

And the idea of mask is interesting, but poses interesting problem as well:

We are not guaranteed by "len" parameter that we can read the whole dword. At the same time, "external" (like memory model and allocator) seem to guarantee that (because we have started on aligned...). I will have to think hard about this

Mirek

Subject: Re: Optimized memcmp for x86
Posted by mr_ped on Sat, 23 Feb 2008 17:56:06 GMT
View Forum Message <> Reply to Message

Yes, that's the problem.
In good old days of ZX Spectrum I was sure I can read beyond "len" and nothing will happen (except getting weird data).

But nowadays I don't know so much about different platforms and OS to be sure you can read aligned double word without causing some exception or crash of application.

I think x86 works usually with aligned memory allocation, so you basically can NOT allocate like 13 bytes only, but the "THINK" word is the crucial part of this sentence.

The C++ itself does not do any memory read checking, so it's up to OS.
So I think the testing application which will allocate memory by OS mem allocator directly would give us the reliable answer.

Than again if such OS allocators allow to allocate only for example 4kB chunks and not 13 bytes, I think it will never raise exception or crash and you may safely read beyond end of buffer.

Subject: Re: Optimized memcmp for x86
Posted by cbpporter on Sat, 23 Feb 2008 21:18:43 GMT
View Forum Message <> Reply to Message

Quote:Than again if such OS allocators allow to allocate only for example 4kB chunks and not 13 bytes, I think it will never raise exception or crash and you may safely read beyond end of buffer.
That is true, but it will not allocate those 4KB for every 13 bytes you want, only if the previoslly allocated 4KB chunk is full. Your requested pointer may be on the end of that allocated zone, and here you could have big problems.

Anyway, this is a memcmp operation, so even if you don't crash, just getting gibberish data could compromise the functionality.

Subject: Re: Optimized memcmp for x86
Posted by mr_ped on Sat, 23 Feb 2008 22:23:01 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sat, 23 February 2008 22:18Quote:Than again if such OS allocators allow to allocate only for example 4kB chunks and not 13 bytes, I think it will never raise exception or crash and you may safely read beyond end of buffer.
That is true, but it will not allocate those 4KB for every 13 bytes you want, only if the previoslly allocated 4KB chunk is full. Your requested pointer may be on the end of that allocated zone, and here you could have big problems.

Anyway, this is a memcmp operation, so even if you don't crash, just getting gibberish data could compromise the functionality.

If you get truly 13B from end of 4kB chunk, the starting address will be not aligned => classic memcmp will be called.

If starting pointer is aligned and you know the whole 4kB chunk is readable, you may safely read 4bytes even if the last 3 are beyond the original buffer, you can't cross 4kB chunk boundary in any case.

Those gibberish data are masked out before comparison.
You should probably check the original routine and my suggestion firstly to get idea what's the problem with that last double word read from memory.

Subject: Re: Optimized memcmp for x86
Posted by mirek on Tue, 26 Feb 2008 21:07:01 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Sat, 23 February 2008 17:23

If you get truly 13B from end of 4kB chunk, the starting address will be not aligned => classic memcmp will be called.

If starting pointer is aligned and you know the whole 4kB chunk is readable, you may safely read 4bytes even if the last 3 are beyond the original buffer, you can't cross 4kB chunk boundary in any case.

Those gibberish data are masked out before comparison.
You should probably check the original routine and my suggestion firstly to get idea what's the problem with that last double word read from memory.

Yes. I believe you are correct.

(This is post-2008.1 stuff anyway, but it is nice to get ready

Mirek

Subject: Re: Optimized memcmp for x86
Posted by mr_ped on Wed, 27 Feb 2008 10:43:47 GMT
View Forum Message <> Reply to Message

I was thinking about it a bit more, and I think some serious profiling data should be gathered, i.e. how often it is called with aligned pointers.

I think this function itself is rarely used?
And if it is, it may often be used to search trough strings?
And in such case the pointers will be not aligned very often?

I mean, is this really worth of effort? Only some profiling of real applications can tell.

But it was nice mental exercise anyway.

Subject: Re: Optimized memcmp for x86
Posted by mirek on Wed, 12 Mar 2008 20:27:08 GMT
View Forum Message <> Reply to Message

mr_ped wrote on Wed, 27 February 2008 05:43I was thinking about it a bit more, and I think some serious profiling data should be gathered, i.e. how often it is called with aligned pointers.

I think this function itself is rarely used?
And if it is, it may often be used to search trough strings?
And in such case the pointers will be not aligned very often?

I mean, is this really worth of effort? Only some profiling of real applications can tell.

But it was nice mental exercise anyway.

Well, the primary motivation was the speedup for small strings - there data definitely are aligned and even the number of characters is fixed.

This optimization seems to bring in about 5% improvement to container benchmark where Sort is used. That is not bad.

Anyway, for non-small strings, we do have guarantee that data are always aligned, so I will probably directly use even unaligned version For String alone, this optimization is worthwhile.