An optimal memcpy

There is a substantial DEMO here as well as a project to write.

READ this specification.

READ the DEMO. The DEMO is just a demo.

Do not include it in your project.

FUN

I think this assignment is one of the most fun I assign because it reveals how much *learning there is to be had from something as simple as copying a bunch of bytes from one place to another.*

Please advise me of typos / missing words in the specification.

First, some learning

`memcpy(* destination, * source, size)` - sounds simple right?

Before you go onto the actual project, let's learning from a demo program.

This program presents 6 implementations of memcpy-like functions including the real `memcpy()` for speed comparison.

These are:

- A naive implementation which uses the typical index counted `for` loop (`memcpy_1()`).
- A better implementation which does away with the overhead of the loop counter (`memcpy_2()`).
- A still better implementation which transfers one `u_int64_t` per loop rather than than a `u_char` (`memcpy_3()`).
- A still better implementation which unrolls the loop 8 times (`memcpy_4()`).
- The original `memcpy` (`memcpy_5()`).
- The unrolled loop version from `memcpy_4()` but with two threads.

The buffers are a little less than 17 million bytes in size (1 « 24).

Four hundred iterations of each `memcpy` method is used in an interleaved manner. The interleaving ought to give more representative results as the other things going on on your computer ought to even out.

The program shows reasonable uses of `typedef`.

The program shows reasonable use of function pointers.

The program shows use of `chrono` for high accuracy timing.

Additional learning opportunity

This is a great example of how "release" mode differs from "debug" mode performance. The difference is very large.

Compiling with optimization on maximum:

```
g++ -O3 -Wall -std=c++11 demo.cpp -lpthread
```

Compiling with debugging (and no optimization):

```
g++ -g -Wall -std=c++11 demo.cpp -lpthread
```

Sample performance with optimization on an M1-based Mac:

```
Timing 6 implementations of memcpy-like functions moving 16777216 bytes 400 times
V1: 0.2023490980000002 --- 0.0005058727450000
V2: 0.2035686450000003 --- 0.0005089216125000
V3: 0.2033044660000001 --- 0.0005082611650000
V4: 0.0248840480000000 --- 0.0000622101200000
V5: 0.1209772489999999 --- 0.0003024431225000
V6: 0.0394600820000000 --- 0.0000986502050000
```

Performance without optimization (with debugging) on an M1-based Mac.

```
Timing 6 implementations of memcpy-like functions moving 16777216 bytes 400 times
V1: 6.3365170459999955 --- 0.0158412926150000
V2: 4.5549721779999999 --- 0.0113874304450000
V3: 0.6144272630000004 --- 0.0015360681575000
V4: 0.1236855160000000 --- 0.0003092137900000
V5: 0.1211884510000000 --- 0.0003029711275000
V6: 0.0965033389999999 --- 0.0002412583475000
```

A question to ask: "Why did the execution time of the original memcpy (shown as V5) not change much?"

Answer: Because that code comes from a preexisting library and is not affected by our command line.

Additional learning opportunity

Inside the innermost loop, there is ONE statement that forces the loop to use a reference (`auto & f`) rather than a copy. Which is it?

Answer: The line adding to `f.total_time` - if `f` is not a reference, the accumulated value will be zero since the copy would be changed, not the original.

Additional learning opportunity

Looking at the code, you will find `aligned_alloc`.

This kind of memory allocation makes sure moving more than one byte at a time works.

Additional learning opportunity

Some hardware implementations provide hardware support for transferring massive blocks of memory. They offer *DMA* units which asynchronously do large transfers. The CPU launches the hardware unit then at some point later receives an *interrupt* saying the transfer is done.

DMA stands for "Direct Memory Access."

Additional learning opportunity

The real `memcpy` needs to deal with the case where the source and destination memory may overlap. You can assume this is not the case for you (and can confirm this by looking at the source code for the framework you are provided).

Additional learning opportunity

There are some nice practices shown in the framework code.

Can multiple cores help?

It depends. One approach to speeding up computation is to spread it over multiple cores. This is not always possible but in this case, it trivially is. For two cores, for example, have the two cores each copy half the memory.

However, it is **not** always faster to use multiple cores because it takes time to launch the threads. The amount of useful work may be too small to compensate for this added overhead. Even if the amount of useful work is large, you may run up against other bottleneck such as saturating access to RAM. That is, more threads won't make you're memory controller or memory itself work any faster. Once you get to the fastest memory can do, adding more threads will not help.

Implementation V6 uses C++ threads to divide the work in half as described above. It turns out, we just eek out a little savings.

A question to ask: "How would the savings vary by the number of bytes to transfer?"

Answer: If the number of bytes to transfer (the amount of work) is made smaller, the relative impact of the thread overhead is made larger - so the multithreaded version will be slower at some point.

A look at the code

You are encouraged to look at the read code but here are snippets.

Naive approach

```
void memcpy_1(u_char * d, u_char * s, size_t size) {
    for (size_t i = 0; i < size; i++) {
```

```
        *(d++) = *(s++);
    }
}
```

Better - no loop counter overhead

```
void memcpy_2(u_char * d, u_char * s, size_t size) {
    u_char * e = s + size;
    while (s < e) {
        *(d++) = *(s++);
    }
}
```

Even better - move 8 bytes at once rather than 1

```
void memcpy_3(u_char * d, u_char * s, size_t size) {
    int szl = sizeof(u_int64_t);
    u_int64_t * ls = (u_int64_t *) s;
    u_int64_t * ld = (u_int64_t *) d;
    u_int64_t * le = ls + (size / szl);

    while (ls < le) {
        *(ld++) = *(ls++);
    }
}
```

but this places restrictions on the length of your buffer. You **must** relax this restriction in your project. That is, your project requires that there be no constraint upon the number of bytes to be copied.

Even betterer (sic) - unroll the loop

```
void memcpy_4(u_char * d, u_char * s, size_t size) {
    int szll = sizeof(u_int64_t);
    u_int64_t * ls = (u_int64_t *) s;
    u_int64_t * ld = (u_int64_t *) d;
    u_int64_t * le = ls + (size / szll / 8);

    while (ls < le) {
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
        *(ld++) = *(ls++);
```

```
    }
}
```

This is faster still but puts even more restrictions on the length of the buffer - now the length of the buffer in bytes, must be a multiple of 8 times 8.

Maybe most betterer (sic) - using threads

```
void memcpy_6(u_char * d, u_char * s, size_t size) {
    thread t1(memcpy_4, d, s, size / 2);
    thread t2(memcpy_4, d + size / 2, s + size / 2, size / 2);
    t1.join();
    t2.join();
}
```

Maybe this is better. Maybe it isn't. Thank you for asking why that is.

Framework

You are **given** the framework with which you'll drive your ultra super fantabulistic (sic) `memcpy()`. The program is here.

The main loop is simply this:

```
for (int i = 0; i < iterations; i++) {
    MEMCPY(dst, src, length);
}
```

where you write `MEMCPY()` in AARCH64 assembly language.

You'll note that `MEMCPY()` was forward declared with:

```
extern "C" void MEMCPY(u_int8_t *, u_int8_t *, u_int64_t);
```

This is the calling convention your assembly language implementation must operate with. Explicitly:

Reg	Contents
x0	Pointer to destination
x1	Pointer to source
x2	Length of copy

Assumptions / Constraints

In the above demo code, we leveraged certain constraints - namely that the number of bytes to copy was a multiple of 64 (`memcpy_4()`).

Your code must be able to efficiently copy ANY number of bytes including odd numbers of bytes.

You may continue to assume that both the source and destination are aligned to 16 bytes boundaries.

Lessons from the demo code

Your code should *not* use threading. The most important lessons in the demo code include:

- Loop unrolling (`memcpy_4`)
- Widening the width of what is copied per instruction (`memcpy_3` and `memcpy_4`)

Loop unrolling

Unroll at most 8 times. This is not an arbitrary value, though not the only "good" value.

You will want to do the majority of your work in a specially crafted loop. Take this and the mention of the Duff Device below as a hint.

Fetching / storing various widths

This table shows loads. Similar instructions exist for stores.

Note that some of these instructions place constraints upon what registers to use.

Mnemonic	Meaning	Length	Register Type
ldrb	Load a byte	1	w
ldrh	Load a short	2	w
ldr	Load an int	4	w
ldr	Load a long	8	x
ldr	Load a long long	16	q

This entire subsection is one giant hint.

The Duff Device

This entire subsection is a giant hint. Explaining this code is up to you but there *is* a Wikipedia article on this.

Further, we have studied jump tables earlier in this class. You must dust off that knowledge to go as fast as possible.

```
/* The original Duff Device - THIS IS NOT CODE FOR YOUR PROJECT */
send(to, from, count)
register short *to, *from;
register count;
```

```
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

Note that this is written in the *original* dialect of C. Today we:

- do not need `register` - optimizers take care of it
- declare function signatures differently

Question to ask: Why is the Duff Device mentioned here?

Answer: For buffers of any size larger than or equal to 32 bytes, you will want to unroll a loop. This is a hint.

More giant hints

You can load 32 bytes with one instruction. You can store 32 bytes with one instruction. This the most bytes you can copy in the smallest number of instructions. For a large buffer, this is how most of the work is done.

You can load and store 16 bytes in one instruction respectively.

You can load and store 8 bytes in one instruction respectively.

You can load and store 4 bytes in one instruction respectively.

You can load and store 2 bytes in one instruction respectively.

You can load and store 1 byte in one instruction respectively.

No MOD operator

Note that there is no `mod` operator on the ARM. We looked at one way to compute `mod` in the past. That isn't the only way. Either way, you will need an implementation for the assembly language you write.

Meaning of the framework's command line options

Option	Argument	Meaning
i	N	Sets the number of iterations to perform to N
l	N	Sets the number of bytes to copy to N
L	N	Sets the number of bytes to copy to 2^^N

Expectation of the number of lines of code in your solution

Including lots of comments and blank lines, my solution is 150 lines long.

Expectation of help

- Expect little help on this project.
- No appointments will be given.
- Any help that is offered will be done as part of class.
- No help will be given specific to your code (see previous).

Partner rules

All work is solo. Feel free to discuss approaches in any size grouping you wish. But no code sharing.

Due Date

This project is due on the last day of classes (11:59 PM on May 19).

Comment about testing your code

Make sure you test with different lengths which should include:

- An odd length
- A length not divisible by 4
- A length not divisible by 8
- A length not divisible by 16
- A length not divisible by 32

This entire subsection is one giant hint.

ChatGPT

I've already mined ChatGPT for a solution:

- It is not as fast as mine.
- It has a distinctive look - I will recognize it.

Sample output

This is on an M1 with 32 GiB of RAM.

```
 % ./a.out -L 22 -i 1024
Bytes to copy: 4194304
Iterations:    1024
Total time:    0.104714
Average time:  0.00010226
GB per second: 38.1993
Correct copy:  true
%
```

Summary

There are a lot of giant hints in this document.

Boomer Rant

It is so very disappointing to me to see code copied wholesale from my tutorials and example directly into your projects. **COPY / PASTE is not an acceptable substitute for learning.**

If my overly dramatic contention that "Software Kills" is true, the "copy / paste from Stack Overflow" generation is going to cause havoc.

You are supposed to **learn** from examples, **not parrot them**.