An optimal memcpy

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

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.
- A better implementation which does away with the overhead of the loop counter.
- A still better implementation which transfers one `u_int64_t` per loop rather than than a `u_char`.
- A still better implementation which unrolls the loop 8 times.
- The original `memcpy`.
- The unrolled loop version but with two threads.

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 use 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:

```
g++ -O3 -Wall -std=c++17 main.cpp -lpthread
```

Compiling with debugging:

```
g++ -g -Wall -std=c++17 main.cpp-lpthread
```

Sample performance with optimization on a Core i5-based Mac:

```
hyde pk_memcpy $> g++ -O3 -Wall -std=c++17 main.cpp -lpthread
hyde pk_memcpy $> ./a.out
Timing 6 implementations of memcpy-like functions moving 16777216 bytes 400 times
V1: 0.779964
V2: 0.777756
V3: 0.777452
V4: 0.0983639
V5: 0.620984
V6: 0.0905187
hyde pk_memcpy $>
```

Sample performance when built with debugging from the Core i5-based Mac:

```
hyde pk_memcpy $> g++ -g -Wall -std=c++17 main.cpp -lpthread
hyde pk_memcpy $> ./a.out
Timing 6 implementations of memcpy-like functions moving 16777216 bytes 400 times
V1: 14.757
V2: 15.803
V3: 2.07889
V4: 0.236038
V5: 0.632458
V6: 0.158866
hyde pk_memcpy $>
```

Note these execution times were on an x64 running natively not on the emulated ARM.

Here is performance on the M1-based Mac (native ARM) with full optimization:

```
perryk@ROCI pk_memcpy_asm % ./a.out
Timing 6 implementations of memcpy-like functions moving 16777216 bytes 400 times
V1: 0.22681
V2: 0.218958
V3: 0.221321
V4: 0.0302979
V5: 0.168199
V6: 0.0435328
perryk@ROCI pk_memcpy_asm %
```

NOTE THE NEED FOR C++ 17. WHY? BECAUSE APPLE.

A question to ask: "Why did the execution time of the original memcpy not change much?"

Answer: Because that code was not compiled with debugging as it comes from a preexisting library.

Additional learning opportunity

Inside the innermost loop, there is ONE statement that forces the loop to use a reference 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`. On the Mac, this requires C++ 17. On all real computers, it comes with C++ 11. Why? Because Apple.

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.

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 on each core. 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 controll or memory 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 importance 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.

An approach so naive I didn't implement

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

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.

Even betterer - 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.

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.

Your project

Framework

You are given the framework with which you'll drive your ultra super fantabulistic `memcpy()`. The program is here. It must be built with C++ 17. It doesn't use threading, you won't need that in this project.

The main loop is simply this:

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

where you write `MEMCPY()`.

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 number of times.

Every AARCH64 instruction is 32 bits in length. A single cache line in the AARCH64 is 128 bits. This means 16 instructions fit in just 4 cache lines meaning your main loop(s) will fit in just 5 cache lines - a very small number so likely to stay in the cache for as long as your process is on the CPU.

This entire subsection is one giant 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
ldrb	Load a byte
ldrh	Load a short
ldr	Load an int
ldr	Load a long
ldr	Load a long long

This entire subsection is one giant hint.

The Duff Device

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

```
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

Hint: *Another example of how close C is to assembly language*

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, write a function to compute mods so you can reuse this code.

This is the first time you will write what might be called a **JUMP TABLE**.

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

Expectation of the number of lines of code in your solution

Not a small number.

Partner rules

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

Duration of assignment

Assigned on a Tuesday. Due the next week Thursday (9 days).

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

This entire subsection is one giant hint.

Summary

There are a lot of giant hints in this document.