Parts I skipped summarized

We recalled that modern microprocessor architectures use caches to reduce memory access times, so typical architectures have special hardware to ensure that the caches on the different chips are **coherent**. Since the unit of cache coherence, a **cache line** or **cache block**, is usually larger than a single word of memory, this can have the unfortunate side effect that two threads may be accessing different memory locations, but when the two locations belong to the same cache line, the cache-coherence hardware acts as if the threads were accessing the same memory location. Thus, if one of the threads updates its memory location, and then the other thread tries to read its memory location, it will have to retrieve the value from main memory. That is, the hardware is forcing the thread to act as if it were actually sharing the memory location. Hence, this is called **false sharing**, and it can seriously degrade the performance of a shared-memory program.

Some C functions cache data between calls by declaring variables to be **static**. This can cause errors when multiple threads call the function; since static storage is shared among the threads, one thread can overwrite another thread's data. Such a function is not **thread-safe**, and, unfortunately, there are several such functions in the C library. Sometimes, however, there is a thread-safe variant.

When we looked at the program that used the function that wasn't thread-safe, we saw a particularly insidious problem: when we ran the program with multiple threads and a fixed set of input, it sometimes produced correct output, even though the program was erroneous. This means that even if a program produces correct output during testing, there's no guarantee that it is in fact correct—it's up to us to identify possible race conditions.