the main thread simply blocks itself on the barrier, knowing that the barrier will unblock only after the worker threads have invoked pthread_barrier_wait() as well.

As mentioned earlier, with the *pthread_join()*, the worker threads are done and dead in order for the main thread to synchronize with them. But with the barrier, the threads are alive and well. In fact, they've just unblocked from the *pthread_barrier_wait()* when all have completed. The wrinkle introduced here is that you should be prepared to do something with these threads! In our graphics example, there's nothing for them to do (as we've written it). In real life, you may wish to start the next frame calculations.

Multiple threads on a single CPU

Suppose that we modify our example slightly so that we can illustrate why it's also sometimes a good idea to have multiple threads even on a single-CPU system.

In this modified example, one node on a network is responsible for calculating the raster lines (same as the graphics example, above). However, when a line is computed, its data should be sent over the network to another node, which will perform the display functions. Here's our modified *main()* (from the original example, without threads):

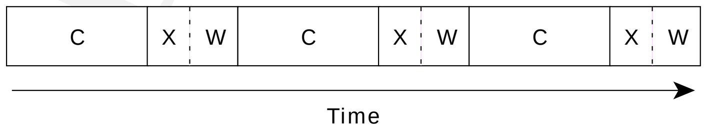
```
int
main (int argc, char *argv[])
{
    int x1;

    ...    // perform initializations

    for (x1 = 0; x1 < num_x_lines; x1++) {
        do_one_line (x1);           // "C" in our diagram, below
        tx_one_line_wait_ack (x1);  // "X" and "W" in diagram below
    }
}
```

You'll notice that we've eliminated the display portion and instead added a *tx_one_line_wait_ack()* function. Let's further suppose that we're dealing with a reasonably slow network, but that the CPU doesn't really get involved in the transmission aspects — it fires the data off to some hardware that then worries about transmitting it. The *tx_one_line_wait_ack()* uses a bit of CPU to get the data to the hardware, but then uses *no CPU* while it's waiting for the acknowledgment from the far end.

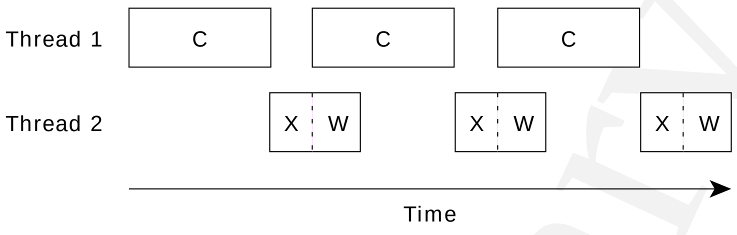
Here's a diagram showing the CPU usage (we've used "C" for the graphics compute part, "X" for the transmit part, and "W" for waiting for the acknowledgment from the far end):



Serialized, single CPU.

Wait a minute! We're wasting precious seconds waiting for the hardware to do its thing!

If we made this multithreaded, we should be able to reduce the total running time, right?



Multithreaded, single CPU.

This is much better, because now, even though each thread spends time waiting, we've reduced the total running time.

If our times were $T_{compute}$ to compute, T_{tx} to transmit, and T_{wait} to let the hardware do its thing, in the first case our total running time would be:

$$(T_{compute} + T_{tx} + T_{wait}) \times num\_x\_lines$$

whereas with the two threads it would be

$$(T_{compute} + T_{tx}) \times num\_x\_lines + T_{wait}$$

which is shorter by

$$T_{wait} \times (num\_x\_lines - 1)$$

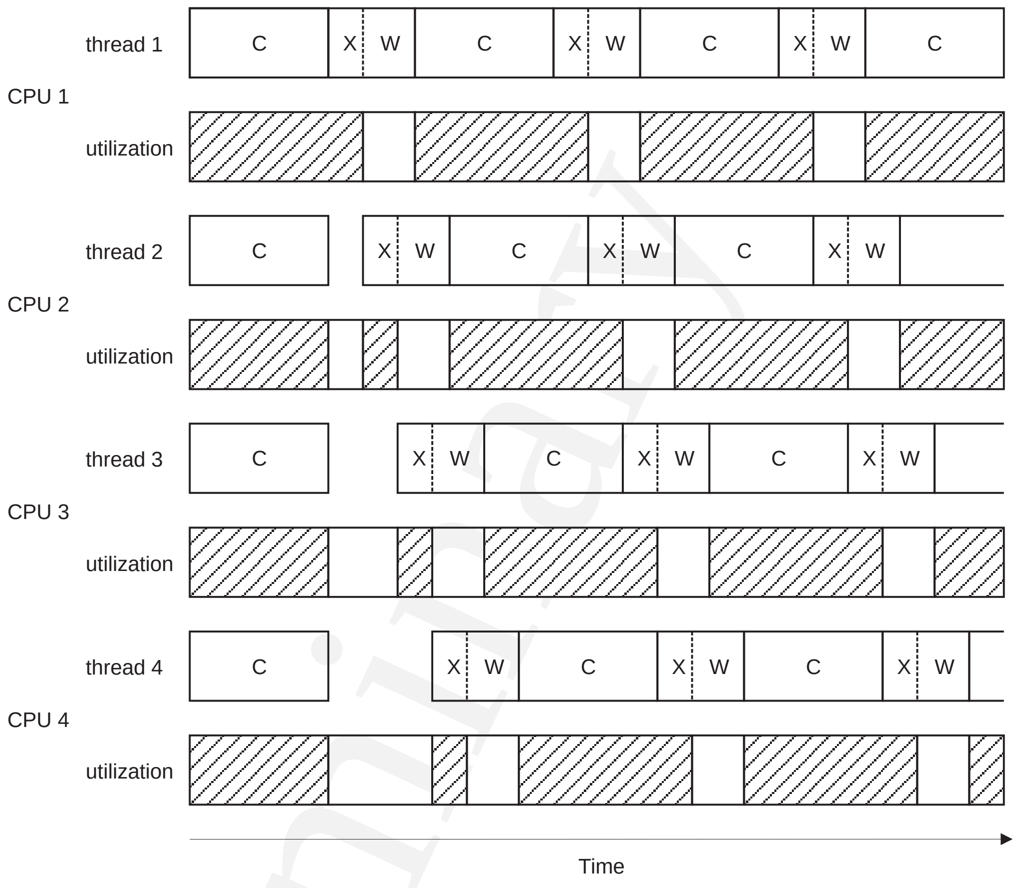
assuming of course that $T_{wait} \leq T_{compute}$.

Note that we will ultimately be constrained by:

$$T_{compute} + T_{tx} \times num\_x\_lines + T_{wait}$$

because we'll have to incur at least one full computation, and we'll have to transmit the data out the hardware — while we can use multithreading to overlay the computation cycles, we have only one hardware resource for the transmit.

Now, if we created a four-thread version and ran it on an SMP system with 4 CPUs, we'd end up with something that looked like this:



Four threads, four CPUs.

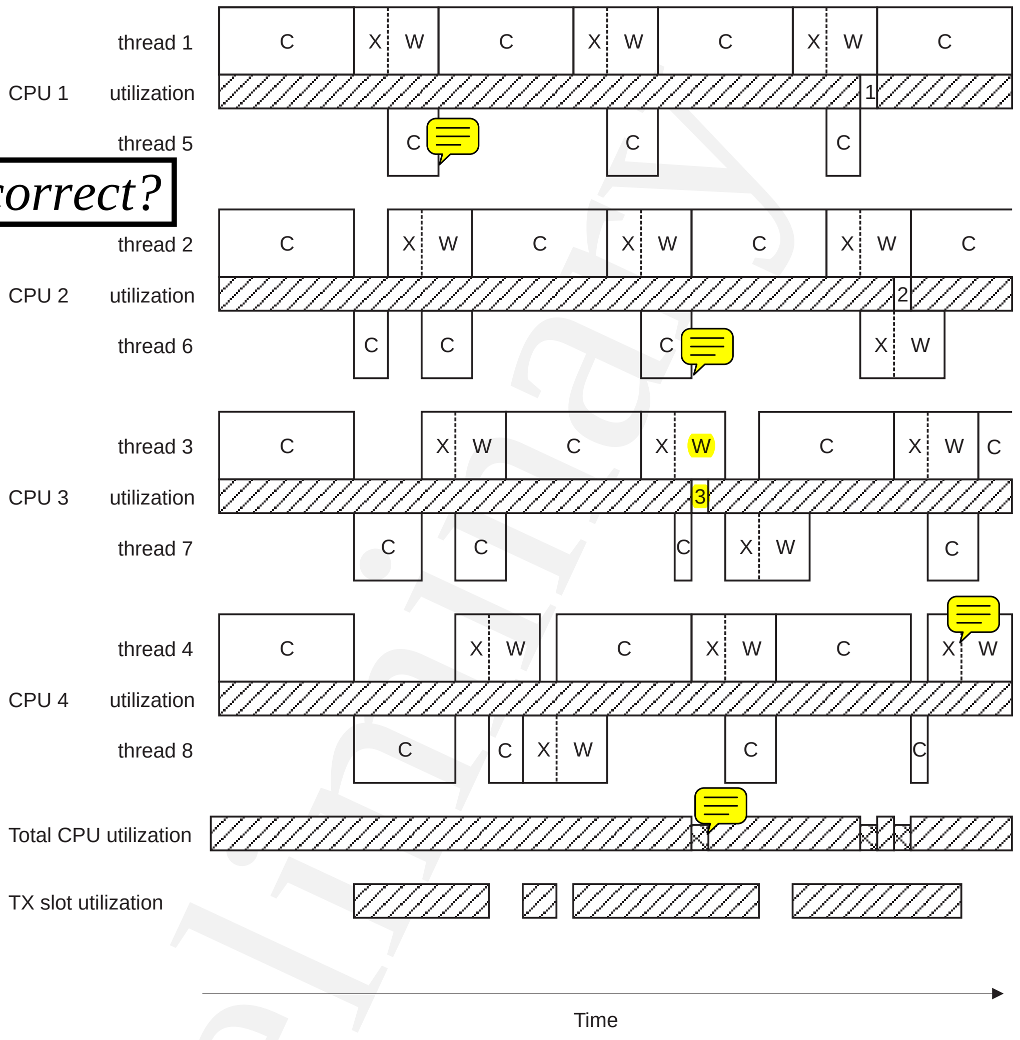
Notice how each of the four CPUs is underutilized (as indicated by the empty rectangles in the "utilization" graph). There are two interesting areas in the figure above. When the four threads start, they each compute. Unfortunately, when the threads are finished each computation, they're contending for the transmit hardware (the "X" parts in the figure are offset — only one transmission may be in progress at a time). This gives us a small anomaly in the startup part. Once the threads are past this stage, they're naturally synchronized to the transmit hardware given certain assumptions. The total running time has a lower bound given by the following formula. The bound can be tight if $T_{compute} > (num\_cpus - 1) \times T_{tx} - T_{wait}$ and if *num_x_lines/num_cpus* is large.

$$(T_{compute} + T_{tx} + T_{wait}) \times num\_x\_lines \,/\, num\_cpus$$

This formula states that using four threads on four CPUs will be approximately 4 times faster than the single-threaded model we started out with.

By combining what we learned from simply having a multithreaded single-processor version, we would ideally like to have more threads than CPUs, so that the extra threads can "soak up" the idle CPU time from the transmit acknowledge waits (and the transmit slot contention waits) that naturally occur. In that case, we'd have something like this:

Is this figure correct?



Eight threads, four CPUs.

This figure assumes a few things:

- threads 5, 6, 7, and 8 are bound to processors 1, 2, 3, and 4 (for simplification)
- once a transmit begins it does so at a higher priority than a computation
- a transmit is a non-interruptible operation

Notice from the diagram that even though we now have twice as many threads as CPUs, we still run into places where the CPUs are under-utilized. In the diagram, there are three such places where the CPU is "stalled"; these are indicated by numbers in the individual CPU utilization bar graphs:

1 Thread 1 was waiting for the acknowledgment (the "W" state), while thread 5 had completed a calculation and was waiting for the transmitter.

2 Both thread 2 and thread 6 were waiting for an acknowledgment.

3 Thread 3 was waiting for the acknowledgment while thread 7 had completed a calculation and was waiting for the transmitter.

This example also serves as an important lesson — you can't just keep adding CPUs in the hopes that things will keep getting faster. There are limiting factors. In some cases, these limiting factors are simply governed by the design of the multi-CPU motherboard — how much memory and device contention occurs when many CPUs try to access the same area of memory. In our case, notice that the "TX Slot Utilization" bar graph was starting to become full. If we added enough CPUs, they would eventually run into problems because their threads would be stalled, waiting to transmit.

In any event, by using "soaker" threads to "soak up" spare CPU, we now have better CPU utilization. A lower bound on total running time is now:

$$(T_{compute} + T_{tx}) \times \mathit{num\_x\_lines} \,/\, \mathit{num\_cpus} + T_{wait}$$

In the computation *per se*, we're limited only by the amount of CPU we have; we're not idling any processor waiting for acknowledgment. (Obviously, that's the ideal case. As you saw in the diagram there are a few times when we're idling one CPU periodically. Also, as noted above,

$$T_{compute} + T_{tx} \times \mathit{num\_x\_lines} + T_{wait}$$

is our ultimate lower bound on the total running time.)

Things to watch out for when using SMP

While in general you can simply "ignore" whether or not you're running on an SMP architecture or a single processor, there are certain things that *will* bite you. Unfortunately, they may be such low-probability events that they won't show up during development but rather during testing, demos, or the worst: out in the field. Taking a few moments now to program defensively will save problems down the road.

Here are the kinds of things that you're going to run up against on an SMP system:

- Threads really *can* and *do* run concurrently — relying on things like FIFO scheduling or prioritization for synchronization is a no-no.
- Threads and Interrupt Service Routines (ISRs) also *do* run concurrently — this means that not only will you have to protect the thread from the ISR, but you'll also have to protect the ISR from the thread. See the Interrupts chapter for more details.