The new thread creation is line 29. We pass on to it a reference to the queue we've created, and the prime number we've found.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to die), we pass on the notice to our child and wait for it to exit if we've created a child (lines 32 and 37).

Meanwhile, back in the main thread, we create a queue (line 9) and the initial child thread (line 10), and pre-seed it with the first prime: 2. Then we queue all the numbers from 3 to 1000 for checking (lines 12-14), then queue a die notice (line 16) and wait for the first child thread to terminate (line 17). Because a child won't die until its child has died, we know that we're done once we return from the join.

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

50.12 Different implementations of threads

Some background on thread implementations from the operating system viewpoint. There are three basic categories of threads: user-mode threads, kernel threads, and multiprocessor kernel threads.

User-mode threads are threads that live entirely within a program and its libraries. In this model, the OS knows nothing about threads. As far as it's concerned, your process is just a process.

This is the easiest way to implement threads, and the way most OSes start. The big disadvantage is that, since the OS knows nothing about threads, if one thread blocks they all do. Typical blocking activities include most system calls, most I/O, and things like sleep().

Kernel threads are the next step in thread evolution. The OS knows about kernel threads, and makes allowances for them. The main difference between a kernel thread and a user-mode thread is blocking. With kernel threads, things that block a single thread don't block other threads. This is not the case with user-mode threads, where the kernel blocks at the process level and not the thread level.

This is a big step forward, and can give a threaded program quite a performance boost over non-threaded programs. Threads that block performing I/O, for example, won't block threads that are doing other things. Each process still has only one thread running at once, though, regardless of how many CPUs a system might have.

Since kernel threading can interrupt a thread at any time, they will uncover some of the implicit locking assumptions you may make in your program. For example, something as simple as `$a = $a + 2` can behave unpredictably with kernel threads if $a is visible to other threads, as another thread may have changed $a between the time it was fetched on the right hand side and the time the new value is stored.

Multiprocessor kernel threads are the final step in thread support. With multiprocessor kernel threads on a machine with multiple CPUs, the OS may schedule two or more threads to run simultaneously on different CPUs.

This can give a serious performance boost to your threaded program, since more than one thread will be executing at the same time. As a tradeoff, though, any of those nagging synchronization issues that might not have shown with basic kernel threads will appear with a vengeance.

In addition to the different levels of OS involvement in threads, different OSes (and different thread implementations for a particular OS) allocate CPU cycles to threads in different ways.

Cooperative multitasking systems have running threads give up control if one of two things happen. If a thread calls a yield function, it gives up control. It also gives up control if the thread does something that would cause it to block, such as perform I/O. In a cooperative multitasking implementation, one thread can starve all the others for CPU time if it so chooses.

Preemptive multitasking systems interrupt threads at regular intervals while the system decides which thread should run next. In a preemptive multitasking system, one thread usually won't monopolize the CPU.

On some systems, there can be cooperative and preemptive threads running simultaneously. (Threads running with realtime priorities often behave cooperatively, for example, while threads running at normal priorities behave preemptively.)

Most modern operating systems support preemptive multitasking nowadays.