DAVID BLANK-EDELMAN

practical Perl tools: spawning

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the book *Perl for System Administration* (O'Reilly, 2000). He has spent the past 20 years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and was one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

AS THE PARENT OF A NEW BABY, I'VE really come to appreciate the idea of multitasking. Remind me to tell you about the times I've successfully fed my child from a bottle while simultaneously going to the bathroom and petting a cat that asserted it could not live another moment without some attention. (Ok, maybe I won't tell you about them, though wouldn't that make a swell column?) Having the number of uninterrupted time slots shrink considerably since our child was born has made me a fan of things in the Perl world that help get tasks done more quickly or efficiently. Multitasking is one of those techniques, and that's just what we're going to talk about in today's column.

Fork()ing

Let's start out with one of the basic building blocks of most multitasking Perl code: fork()ing. The fork() function comes from the UNIX system call of the same name, though it works on most other operating systems as well. This now includes Windows operating systems, thanks to work in 1999 by ActiveState that Microsoft itself sponsored.

Here's a quick review of fork() for those of you who didn't grow up teething on (or perhaps even loading) the V7 magtapes. When you call fork() from a Perl program, a copy of the running process is made. This copy receives all of the context of the program that called fork(), including the run-time environment, any variables in play at the time, and open file handles. The new process is called the "child process" with the original one dubbed the "parent process." Since the program that called fork() continues to run in both the child and the parent processes it is up to your code to ensure that the child process knows to do childlike things (processing something, etc.) and that the parent acts parental (e.g., creating new children or waiting for existing children to finish).

How does your code know whether it is running as the child or the parent process? Almost everything about the two running processes is the same. The key difference is that the parent receives back the pid of the child spawned by fork() as a return code from that fork() call. The child receives a 0 from the same call. This leads to the following idiom you see in most code that uses fork():

```
my $childpid = fork();
die "Fork failed:$!\n" if !defined $childpid; # fork returns undef on failure

# if $childpid == 0, we are in the child process and need to do stuff
if ($childpid == 0) { ... do stuff }

# we're the parent and so we need to reap the fork()d process when done
else {
  # waitpid waits for a particular pid vs. wait() which will
  # reap any child process that has completed
  waitpid($childpid,0);
}
```

A parent process must retrieve the exit status of all of its children once they have completed, a process known as "reaping," or the child processes continue to live on as "zombies" until the parent process dies. Reaping is performed by wait() or waitpid() as seen in the code above. The waitpid()/wait() functions will (you guessed it!) wait until the child in question has finished before returning.

Warning: Because we have a student fork-bomb one of our machines at least once or twice a year (mostly unintentionally) I feel compelled to mention that code that either looks like, acts like, or boils down to this:

```
while (1) { fork(); } # bad bad bad bad
```

is, as the comment says, bad bad bad bad. A program that fork()s out of control like this is a fork-bomb. If process limits allow (e.g., Solaris defaults; see the maxuprc kernel parameter) this program will consume all available spots in your process table, causing the system to melt down and be a real pain to clean up to boot. If you are going to write code that repeatedly fork()s, always build in some kind of big red button to shut the process down, for example:

```
while (1) { fork() unless (-f /tmp/stop); } # create /tmp/stop to end
```

or impose limits to keep the problem self-quashing:

```
while (1) { die "Over the fork limit" if $limit++>100; fork(); }
```

With this little snippet we've covered all of the basics you have to know to begin programming using fork(). Toward the end of the column we'll see an easier way to use this functionality.

Basic Threading

The second simple method for multitasking involves threading. Before we go much further I want to insert a number of caveats about threading under Perl:

1. Thread support is comparatively new to Perl. Actually, it is more accurate to say that *this* kind of thread support is new to Perl. There was an attempt in the past (around Perl 5.005 or so) to add threads but that model never proved to be stable and was eventually moved to "deprecated" status and will leave the code as fast as the Perl 5 developers can get it out (with version 5.10, as I understand it). Support for the current model looks pretty good and is under active maintenance, so you probably don't need to worry. Still, I want you to know that the

ground around this issue is a little softer than usual, so you'll need to step carefully. Things such as debugger support for threads are still a little shaky (better in 5.8.5+ but still incomplete), but threads are definitely usable today.

2. Perl threads are likely to be different from any of the other threading models you have seen before. They are not your granpappy's lightweight processes or precisely any other thread implementation you've encountered before. We'll talk about what they are, but I thought it best to prime you for what they are *not* first.
3. To use the current threading model, your Perl interpreter has to be built with a special option at compile time. This is not enabled by default in a source build, and different OS vendors are more or less adventurous. For example, until Solaris 10 Sun did not have it turned on in the Perl interpreter that ships with Solaris; Apple does turn it on for OS X. To tell if you have it enabled, look for the line that contains useithreads= in the output of perl -V. If it says define you are all set. If it says undef, then you will have to rebuild the interpreter. As a related aside, there is a module called forks that provides the same API as the threads module we'll be using but does it using fork() calls. If you'd like to play with the threads stuff on an OS that doesn't provide a threaded Perl but does support fork() natively, this module may do the trick for you.

If you are still with me after passing the "Danger, This Means You!" paragraphs above, it means you are interested in how threads work in Perl. Let's look at that now. Modern versions of Perl, when enabled at compile time, provide something called "interpreter threads" or "ithreads" for short. I'll use "thread" and "ithread" to mean the same thing in this column.

A standard Perl program gets run by a single Perl interpreter thread that handles the interpretation and execution of a program from start to finish. Perl threads allow you to start up additional Perl interpreters that independently execute parts of your code. Each interpreter thread gets its own copy of the state of the Perl program at that point. If this sounds to you a little bit like a fork() situation we've already covered, that shows you are on the ball.

One difference from fork() is the ability to actually share data between threads. With fork(), the children get a copy of all of the variables in play but changes made by a child aren't seen in the parent's copy unless they work out some sort of external synchronization mechanism. With ithreads, the situation is a little different. By default, ithreads don't actually share any data; the copies they have of the program's state are completely independent. However, if you'd like two ithreads to share access to a variable, you do have the ability to mark that variable as shared using a separate pragma. A shared variable can be changed by one thread and all other threads will see this change as well. This provides considerable power to the programmer but also potential peril, since reading and writing to a shared resource need to be carefully coordinated. Let's take a look at some sample code and then we'll see how such issues are addressed in Perl.

```
use threads;

sub threaded_sub {
                print "running in thread id: " . threads->self->tid() . "\n";
        }

my $thread = threads->create(\&threaded_sub);
```

```
# this shows a scalar return result, but we could also pass a list
my $result = $thread->join;
```

This code shows nearly the simplest use of ithreads I can demonstrate. To use ithreads, we define a subroutine whose code will be executed in a thread. As a thread is created, it is assigned an id (with the main or initial thread when the program first runs being called thread id 0). The thread being created here isn't particularly exciting, since it only prints its id and then exits, but you get the idea.

The create() line takes that code and spins off a new thread to run it. At that point the subroutine threaded_sub is happily executing in a separate thread. The result of the create() command is a thread object we can use for thread control operations. The next line executes one of these operations called join(), a method similar in intent to wait()/waitpid() from our fork examples. join() waits for the desired thread to complete and retrieves the return value from that thread. If we did not include the join() statement, Perl would complain when the program exits leaving behind an unjoined thread:

```
Perl exited with active threads:
0 running and unjoined
1 finished and unjoined
0 running and detached
```

If we didn't care at all about the results of that thread, we could replace the method join() with one called detach().

Now let's get a bit more sophisticated. I mentioned before that different threads do not share the same data unless explicitly instructed to do so. That sharing is performed by adding the threads::shared pragma and marking certain variables with this status:

```
use threads;
use threads::shared;

my $data : shared = 1; # share($data) can also be used
# now all threads will read and write to the same variable
```

Congratulations: With this step we've now stepped onto the rickety bridge over the deadly gorge of Parallel Programming Peril (cue the dramatic alliteration music)! With race conditions and other nasty beasts waiting for us at the bottom of the gorge we have to step very carefully. As soon as you begin to deal with a shared resource, you need to make sure that the right piece of code updates that resource at the right time. The other pieces of code running at the same time must also follow the right protocol to avoid having that update get inadvertently overwritten. To deal with these circumstances Perl offers a set of functions such as lock().

As you can probably guess, lock() attempts to place a lock on a variable and blocks until it succeeds. It's useful when several threads want to modify a shared value:

```
{ lock($data); $data++; }
```

Why use curly braces in that example? Perl's threading model has no function called "unlock()"; locks are released when they pass out of scope. By using curly braces around these statements we've set up a temporary scope that just includes the update to the variable $data after the lock is in place.

There are other idioms for thread programming that avoid doing this sort of dance. I don't want to go too deeply into parallel programming techniques, but this one bears a quick look because it comes up so frequently.

Here's a modified version of the example used in the official Perl threads tutorial (perldoc perlthrtut):

```
use threads;
use Thread::Queue;

my $DataQueue = Thread::Queue->new;
$thr = threads->create(
    sub {
        while ( defined( $DataElement = $DataQueue->dequeue ) ) {
            print "Thread "
                . threads->self->tid()
                . " popped $DataElement off the queue\n";
        }
        print "Thread " . threads->self->tid() . " ready to exit\n";
    }
);

print "Thread " . threads->self->tid() . " queues 12\n";
$DataQueue->enqueue(12);
print "Thread " . threads->self->tid() . " queues A, B and C\n";
$DataQueue->enqueue( "A", "B", "C" );
print "Thread " . threads->self->tid() . " queues undef\n";
$DataQueue->enqueue(undef);
$thr->join;
```

Let's go over the code in some detail, because it might not be immediately clear what is going on. First, we create a queue for the two threads we are going to use to share. One thread will place new values at the end of the queue (enqueue()); the other will grab values from the top of the queue (dequeue()) to work on. By using this scheme the threads don't have to worry about bumping into each other.

After creating a queue, the second thread (i.e., the one that is not the main thread) gets defined and launched via the create command. The subroutine defined in this command just attempts to pop a value off the queue and print it. It will do this for as long as it can retrieve defined elements from the queue. It may not be readily apparent from the code here, but when faced with an empty queue, dequeue() will sit patiently (block/hang), waiting for new items to be added. Think of the second thread as always waiting for new elements to appear in the queue so it can retrieve and print them.

The rest of the program takes place in the main thread while the second thread is running. It pushes several values onto the queue: the first a scalar, the second a list, and the third an undefined value. It ends with an attempt to join the second thread. The net result is that the code prints something like this:

```
Thread 0 queues 12
Thread 0 queues A, B and C
Thread 0 queues undef
Thread 1 popped 12 off the queue
Thread 1 popped A off the queue
Thread 1 popped B off the queue
Thread 1 popped C off the queue
Thread 1 ready to exit
```

This looks like all of the action first takes place in the main thread (0) followed by the second thread's work, but that's just the order the output is received. If you step through the main thread with a debugger, you'll find that the main thread will queue a value, the second thread prints it, the main thread queues another value, the second thread prints it, and so on. Thread::Queue has its limitations (some of which are solved by other modules), but in general it provides a fine way to pass things around among threads that all have to coordinate tasks.

Now that you know about queues, we've finished a good surface look at threads in Perl. Be sure to see the documentation for the threads and threads::shared pragmas and the Perl thread tutorial (perlthrtut) for more information on other available functionality.

Convenience Modules

We could spend a lot more time talking about threads, but I want to make sure I mention one more topic before we come to an end. As you probably guessed, Perl has its share of modules that make working with fork() and threading a little easier. Let me show you one that I'm particularly fond of using: Parallel::ForkManager.

I like Parallel::ForkManager because it makes adding parallel processing to a script easy. For example, I have a script I use when I want to rsync the individual directories of a filesystem to another destination. I use this in cases where it is necessary to copy over each subdirectory separately for some reason. Here are some choice pieces from the code:

```
opendir( DIR, $startdir ) or die "unable to open $startdir:$!\n";
while ( $_ = readdir(DIR) ) {
    next if $_ eq ".";
    next if $_ eq "..";
    push( @dirs, $_ );
}

closedir(DIR);

foreach my $dir ( sort @dirs ) {
    ( do the rsync );
}
```

One day I realized that directory copies like this don't have to take place serially. Several directories can be copied simultaneously with no ill effects. Adding this parallel-processing functionality was just a matter of changing the code that said:

```
foreach my $dir ( sort @dirs ) {
    ( do the rsync );
}
```

to:

```
# run up to 5 copy jobs in parallel
my $pm = new Parallel::ForkManager(5);

foreach my $dir (sort @dirs){
    # ->start returns 0 for child, so only parent process can start new
    # children, once we get past this line, we know we are a child process
    $pm->start and next;

    ( do the rsync );

    $pm->finish; # terminate child process
}

$pm->wait_all_children; # hang out until all processes have completed
```

The added code creates a new Parallel::ForkManager object that can be used to fork a limited number of child processes (->start), have them exit at the right time (->finish), and then clean up after all children with one command (->wait_all_children). The module does all of the scut work behind the scenes necessary to keep only a limited number of fork()ed processes going. I find the ease of adding parallel processing to my scripts (just four lines of code) has made me much more likely to create scripts that handle several tasks simultaneously. There are other convenience modules that are worth looking at (e.g., Parallel::Forker does all that Parallel::ForkManager can do, but it allows you to specify that certain child processes must wait to run after others have completed). Be sure to do searches for "fork," "parallel," and "thread" at search.cpan.org to see what is available. If you find yourself needing a really sophisticated multi-tasking framework, you'd be well served to check out the POE framework at poe.perl.org.

Oops, I have to go back to getting many, many things done at the same time. Take care, and I'll see you next time.