Practical Perl Tools
Parallel Asynchronicity, Part 1

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson) . He has spent close to thirty years in the systems administration/DevOps/SRE field in large multiplatform environments including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard 'g'. dnblankedelman@gmail.com

At some point everyone gets the desire to be able to do multiple things at once or be in multiple places at the same time as a way of getting more done. And although we try to multitask, the research keeps piling up to suggest that humans aren't so good at intentional multitasking. But computers, they do a much better job at this, that is, if we humans can express clearly just how we want them to work on multiple things at once. I thought it might be fun to explore the various ways we can use Perl to write code that performs multiple tasks at once. This can be a fairly wide-ranging topic, so we're going to take it on over multiple columns to give us plenty of space to peruse the subject. Note: for those of you with photographic memories, I touched on a similar subject in this column back in 2007. There will be some overlap (and I might even quote myself), but I'll be bringing the topic up to date by bringing in modules that weren't around in the good ole days.

One quick caveat that my conscience forces me to mention: to avoid writing a book on the topic (been there, done that), these columns will use UNIX or UNIX-derivative operating systems as their reference platform. There's been lots of great work done over the years for other platforms (I'm looking at you Windows), but I won't be making any guarantees that everything written here works on anything but a UNIX-esque system. Caveat Microsoft Emptor and all of that.

Fork!

Usually I don't like to get to forking without a little bit of warm up, but that is indeed the simplest way to get into the parallel processing game. Perl has a fork() function that lets us spin off another copy of a running Perl interpreter that continues to execute the current program. Let's see how it works.

If the Perl interpreter encounters the first line of the program below:

```
my $pid = fork();
print $pid,"\n";
```

a second Perl interpreter comes into being (i.e., a copy is forked off the running interpreter) that is also running the program. That second copy is referred to as a child of the original copy (which, as you would expect, is called the parent process). From the perspective of the program itself, neither copy can tell that anything special has occurred (they are both running the exact same program, have the same file descriptors open, etc.) with one very small difference: when the fork() program line has successfully finished executing, $pid in the *parent* process gets set to the process ID of the child process. In the child process, $pid will be set to 0. So if I run this program as is, I'll get output like this:

```
$ perl fork.pl
6240
0
```

The parent has printed the process ID of the child process (6240), the child printed 0 as expected. As an important aside for your programming, if for some reason the fork() call fails, $pid will be undef in the parent (and yes, you should test for that happening).

The reason this matters is that a parent process has a responsibility for a task in addition to any other work it plans to do that a child does not. A parent process is responsible for "reaping" its children after they exit lest they remain zombies (and we all know zombies are entertaining, I mean bad, right?). Wikipedia has a great description of this [1]:

> On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state". This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped".

There are two common ways for reaping: manually or signal-based. The manual way is the most straightforward. The parent will call wait() or waitpid(), which will block until the child exits. The code looks something like this:

```
my $pid = fork();

die "fork failed!: $!\n";  if (!defined $pid);

if ($pid == 0) {
   # I'm a client
   # do stuff
   exit 0;
}
else {
   # I'm the parent
   # can also do stuff, then reap the child
   waitpid $pid, 0;
}
```

Now, that code just shows a single fork. If we wanted a parent to fork repeatedly, that is as easy as putting a while loop around the fork() call and either keep track of the child process IDs (so we can have the parent wait for each explicitly with waitpid()) or have a similar loop at the end of the parent script that repeatedly calls wait() (which just waits for any child process) the right number of times to clean up after each fork().

If you don't like this approach, another one is to make use of the fact that the parent process should receive a signal when each child process exits (SIGCHLD to be precise). If we add a signal handler that either ignores the signal explicitly or reaps the child that signaled it, we avoid zombies as well. If you plan to go this route, I recommend looking up the Perl Cookbook recipe on "Avoiding Zombie Processes" because it does a good job of laying out some of the caveats you'll need to know.

One last tip for you if you plan to write manually forking code: I've seen far too many fork bombs (where a process forks itself and the machine it is on into oblivion) in my time. Please put logic into your code that limits and/or prevents unbridled forking. Keep a counter, check for a sentinel file (i.e., if a file with a name on disk exists, don't fork), create a dead man's switch (only fork if a file or some other condition is present), and so on. Any strategy to avoid this problem is likely better than no strategy.

Let Someone Else Fork for You

Truth be told, I haven't written code that calls fork() in quite a few years. Ever since I discovered a particular module and how easy it let me spread "run this in parallel" pixie dust on my previously serial code, I really haven't bothered with any of that fork() / wait() drudgery. The module I speak of and love dearly is Parallel::ForkManager. Looking back at the 2007 column, it is clear that my affections haven't waned over that time because the example I gave then is still pertinent today.

Let's quickly revisit that code. At the time, I mentioned having a very large directory tree that I needed to copy from one file system to another. I needed to copy each subdirectory over separately using code similar to this:

```
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 );
}
```

Since the copy operations are not related to each other (except by virtue of touching the same file servers), we could run the copies in parallel. But we have to be a little careful—we probably don't want to perform the task in the maximally parallel fashion (i.e., start up *N* rsyncs where *N* is the number of subdirectories) because that is sure to cause too much I/O and perhaps memory and CPU contention. We'll want to run with a limited number of copies going at a time. Here's some revised code that uses Parallel::ForkManager:

```
# ...read in the list of subdirectories as before

my $pm = new Parallel::ForkManager(5);

foreach my $dir (sort @dirs){
    # we are a child process if we get past this line
    $pm->start and next;

    ( ... do the rsync ... );

    $pm->finish; # terminate child process
}

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

Let's take a walk through the code. The first thing we do is instantiate a Parallel::ForkManager object. When we do, we provide the maximum number of processes we want running at a time (five in this case). We then iterate over each subdirectory we are going to copy. For each directory, we use start() to fork a new process. If the max number of processes is already running, this command will pause until there is a free spot. If we are able to fork(), the line $pm->start returns the pid of the child process to the parent process and 0 to the child as in our last section.

The logic of the "and next" part of the line is a little tricky so let me go to super slow-mo and explain what is going on very carefully. We need to understand two cases: what happens to the parent and what happens to the child process.

If we are the parent process, we'll get a process ID of the child back from start(), and so the line will become something like

```
6240 and next;
```

Because 6240 is considered a "true" value in Perl, the next statement will run and the rest of the contents of the loop will be skipped. This lets the parent move on to the "next" subdirectory so it can start() the next one in the list.

If we are the child process, we'll get a 0 back from the start() call, so the line becomes

```
0 and next;
```

Since Perl short-circuits the "and" construct when it knows the first value is false (as it is here), next isn't called so the contents of the loop (the actual rsync) is run by the child process. The child process then calls $pm->finish to indicate it is done and ready to exit.

At the very end, we call wait_all_children in the parent process so it will hang out to reap the children that were spawned in the process.

As I mentioned in 2007, all it takes is four additional lines for the program to run my actions in parallel, keeping just the right number of processes going at the same time. Easy peasey.

Better Threads

After process forking, the very next topic that usually comes up in a discussion of parallel processing is threads. The usual idea behind threads is they are lightweight entities that live within a single process. They are considered lightweight because they don't require a new process (e.g., with all of the requirements of running a new Perl interpreter) to be spun up for each worker. As a quick aside: modern operating systems do a bunch of fancy tricks to make process spawning/forking not as resource intensive as might first appear, but it still is likely to be heavier than decent threading support. A threading model can sometimes make the programmer work a bit harder for reasons we'll see in the next installment, but it is often worth it.

Allow me to surprise you by ignoring any of the built-in Perl threading support and instead moving on to a module that I think provides for more pleasant use of threads under Perl: Coro. I'll let an excerpt from the module's intro doc explain [2]:

> Coro started as a simple module that implemented a specific form of first class continuations called Coroutines. These basically allow you to capture the current point execution and jump to another point, while allowing you to return at any time, as kind of non-local jump, not unlike C's "setjmp/longjmp."...
>
> One natural application for these is to include a scheduler, resulting in cooperative threads, which is the main use case for Coro today....
>
> A thread is very much like a stripped-down perl interpreter, or a process: Unlike a full interpreter process, a thread doesn't have its own variable or code namespaces—everything is shared. That means that when one thread modifies a variable (or any value, e.g., through a reference), then other threads immediately see this change when they look at the same variable or location.
>
> Cooperative means that these threads must cooperate with each other, when it comes to CPU usage—only one thread ever has the CPU, and if another thread wants the CPU, the running thread has to give it up. The latter is either explicitly, by calling a function to do so, or implicitly, when waiting on a resource (such as a Semaphore, or the completion of some I/O request).

Coro will allow us to write a program where various parts of the program can do some work and then hand off control of the CPU to other parts. If that sounds a lot like a subroutine to you, you are having the same reaction I did when I first started to learn about Coro. One thing that helped me was this comparison between subroutines and coroutines in Wikipedia [3]:

> When subroutines are invoked, execution begins at the start, and once a subroutine exits, it is finished; an instance of a subroutine only returns once, and does not hold state between invocations. By contrast, coroutines can exit by calling other coroutines, which may later return to the point where they were invoked in the original coroutine; from the coroutine's point of view, it is not exiting but calling another coroutine. Thus, a coroutine instance holds state, and varies between invocations; there can be multiple instances of a given coroutine at once. The difference between calling another coroutine by means of "yielding" to it and simply calling another routine (which then, also, would return to the original point) is that the latter is entered in the same continuous manner as the former. The relation between two coroutines which yield to each other is not that of caller-callee, but instead symmetric.

I can't believe I'm going to use this analogy, but it may help you think of this model like a group of people standing in a circle playing Hacky Sack™. One person starts with the footbag, does some tricks, and then (in order for the game to be interesting to all involved) has to pass the bag to another participant who does whatever tricks he or she knows. That person does a few things and then passes it along to the next person and so on. If you could speed up the game such that all of the tricks and all of the passes happen fast enough, you would get to watch a pretty entertaining blur of activity consisting of multiple tasks appearing to basically happen at the same time. This analogy breaks down when you start to talk about the various ways you can synchronize cooperative threads, but it at least gets you started.

We are starting to come to the end of this column (I know, just when it was starting to get good), but before we go, let's learn the very basics of how to use Coro at sort of the "hacky sack analogy" level. Next time we'll pick up right from these basics and look at the more sophisticated features surrounding synchronization and data passing between threads.

The basic way to define a thread in Coro is to use the async function. This function looks almost exactly like a subroutine definition except arguments are passed after the code block:

```
use Coro;
async { print "hi there $_[0]\n"; } 'usenix';
```

So what do you imagine that code prints? If you said "nothing!" you win. If that answer makes you shake your head in confusion, don't worry, it is a bit of a trick question if you haven't worked with this package before. Let me explain.

When the program starts, it is running in what we'll call the "main" thread. This thread runs the async command you see above to queue the requested code as a new thread. Then the main thread exits because there is nothing left in the program to run. As a result, the thread it queued up never got a chance (sob) to run, hence no output. If we wanted that new thread to get some CPU time, we have to give up the CPU in the main thread, as in:

```
use Coro;
async { print "hi there $_[0]\n"; } 'usenix';
cede;
```

Now we get the "hi there usenix" output we expect. We can yield the CPU (which is what most thread packages call it instead of cede) in any thread we want. Let's play around with this idea a bit. What would you guess this program returns? (Warning—it is another trick question.)

```
use Coro;

async {
    print "1\n";
    cede;
    print "back to 1\n";
};
async {
    print "2\n";
    cede;
    print "back to 2\n";
};
async {
    print "3\n";
    cede;

};
cede;
```

Here's the answer:

```
1
2
3
```

Let's walk through what is going on. The main thread starts. It queues thread 1 to run, then thread 2, then thread 3. Finally, the main thread cedes control of the CPU to the next thing that is ready to run, which happens to be thread 1. Thread 1 prints "1" and then cedes control to the next thing in the queue (thread 2). This repeats until thread 3, which cedes control back to the main thread. The main thread has nothing more to do, so the program exits.

If we wanted to return to any of the threads so they can continue and print their second line, the main thread would have to explicitly cede control back to them again. This is just as easy as adding an extra "cede;" to the end of the program. If you are not used to thinking "what is currently running? what could be running?" it can take a little getting used to. Luckily, there are ways to debug Coro programs. We'll talk about that and other advanced subjects in the next installment. Take care, and I'll see you next time.

References

[1] Wikipedia, "Zombie process": https://en.wikipedia.org/wiki/Zombie_process.

[2] Coro: http://search.cpan.org/dist/Coro/coro/intro.pod.

[3] Wikipedia, "Coroutines": https://en.wikipedia.org/wiki/Coroutine.