Process Management in Perl

Houston Perl Mongers
November 12, 2020



OVERVIEW

- Part I – Introduction to the Pain
  - What's a process?
  - Process control in userland (i.e., the shell)
  - Processes versus Threads
  - Note about Perl *ithreads*™
  - Perl's `fork`
- Part II – Making It Less Painful
  - Parallel::ForkManager “family”
  - Perl's Multi-core Engine (MCE) Module
  - Other interesting Perl modules





NOT COVERED

- Efficient IPC among fork'ed perl processes (though this is an interesting topic)
- Anything related to “Perl threads” (ithreads)
- Work scheduling and complicated process management
- “async” frameworks or higher level programming models

PART I

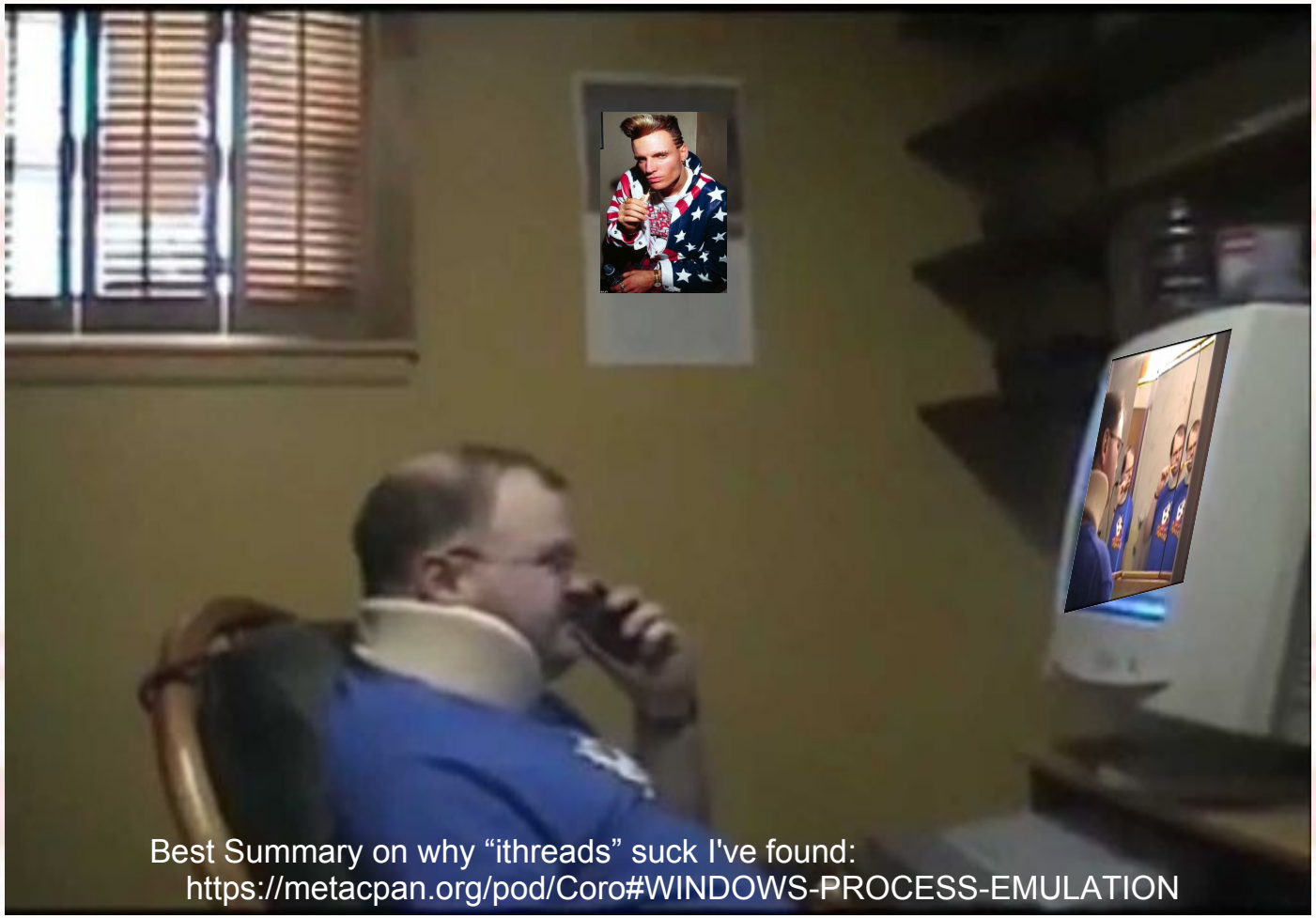
What's a Process?

- *Operating System* concept
- How the OS manages “work” and allocation of sytem resources (time on CPU, memory, network, file system, etc)
- e.g., running a basic Perl script generates a single process
- Most processes we write are single lines of execution – i.e., not “parallel” or “concurrent”

Threads != Processes

- Threads are “light weight” and communicate via shared memory (with “main” thread and sibling threads); all “threads” are part of a single parent logical OS process
- *Forked* processes are full weighted process and do not share memory with parent or siblings, therefore copying all of the memory related is required
- Perl doesn't have 'real' threads and anyone who says it does is lying or ignorant (I usually assume the latter)
- For 'real' threading, see: `OpenMP`, *pthreads*, or the *Qore* scripting language

NOTE ABOUT PERL ITHREADS



Best Summary on why “ithreads” suck I've found:
https://metacpan.org/pod/Coro#WINDOWS-PROCESS-EMULATION

USERLAND PROCESS CONTROL*

this is a grossly inadequate list - "ddg** for much more info"*

Shell (e.g., bash) commands and hints:

- `<ctrl-z>`
  sends a process running into a suspended state
- **`fg`**
  *resumes suspended process into the **f**oreground*
- **`bg`**
  *resumes supended process into the **b**ackground*
- `command&`
  sends a shell command into the background, creates a "child" prcocess
- `(command)&`
  "fire and forget" command in subshell, asynchronously
- **`wait`**
  foreground script control flow aware of child processes

SCRIPTED PROCESS CONTROL
bash EXAMPLE

```
#!/usr/bin/env bash

parentPID=$$
for i in $(seq 1 10); do
  # child subshell
  (
    sleep 1;
    echo "    child process (pid: $!) number says hello" \
         "- I love my parent (pid: $parentPID)";
    sleep 5
  )&
  childPID=$!
  echo my child, $childPID launched!
done

echo waiting
wait
echo all done, exiting...
```

Parent process Id
Child process Id

Child's body

Parent blocking “wait” for ALL children to return

SCRIPTED PROCESS CONTROL
bash EXAMPLE

```
#!/usr/bin/env bash

MAX_CHILD=10
COUNT=0
for i in $(seq 1 55); do
{ ( sleep 1; echo "Child $i of Parent ($$). Child (pid: $!)"; sleep 5)&}
  COUNT=$(($COUNT+1))
  if [ $COUNT -eq $MAX_CHILD ]; then
    echo reached max child process of $MAX_CHILD
    echo waiting for them ALL to finish, then will resume ...
    wait
    echo .. done waiting
    # reset COUNT
    COUNT=0
  fi
done

if [ $COUNT -gt 0 ]; then
  echo waiting for remaining $COUNT children
  wait
fi

echo all done, exiting...
```

"Throttles" number of concurrent child processes using a $COUNT, if/then, and multiple waits

Child's body

wait for remaining child processes if less than $MAX_CHILD

wait

- bash's “wait” will wait for ALL child processes
- perl's “wait” is blocking until ONE of any of the parent's children finish
- perl's “wait” returns “-1” if there are no child processes still running
- “blocking” wait for all in perl requires checking until it returns “-1”

```
while ( 0 < wait ) { };
```

SCRIPTED PROCESS CONTROL
perl EXAMPLE

```
#!/usr/bin/env perl

use strict;
use warnings;
++$|; #autoflush
for my $i ( 1 .. 10 ) {
    my $child_pid = fork();
  CHILD_SUBSHELL:
    if ( 0 == $child_pid ) {
        sleep 1;
        print qq{I am child process ($$)!\n};
        sleep 5;
        exit;   #<- if this is not here, the child will continue the loop
    }
  PARENT_PROCESS:
    if ( 0 < $child_pid ) {
        print qq{I am the parent process ($$)!\n};
    }
}
# "spin wait"
while ( 0 < wait ) { }
print qq{...done\n};
```

"autoflush" for STDOUT

Call to `fork`

Child process Id
Parent process Id

Child's body

`fork` returns:

- "0" for the child process
- PID of child process for the Parent process

Note: this is how you dispatch child/parent code

SCRIPTED PROCESS CONTROL
perl EXAMPLE

```perl
#!/usr/bin/env perl

use strict;
use warnings;
++$|; # autoflush STDOUT
my $MAX_CHILD = 10;
my $COUNT     = 0;
my $parent_pid = $$; # capture parent script PID
for my $i ( 1 .. 55 ) {
    my $child_pid = fork();
    ++$COUNT;    # yes, this value - preincrement is visible to the child
  CHILD_SUBSHELL:
    if ( 0 == $child_pid ) {
        sleep 1;
        print qq{I am child process ($$) my parent is ($parent_pid)!\n};
        sleep 2;
        exit;    #<- if this is not here, the child will continue the loop
    }
  PARENT_PROCESS:
    if ( 0 < $child_pid ) {
        print qq{I am the parent process ($$)!\n};
        if ( $COUNT == $MAX_CHILD ) {
            print qq{reached max child process of $MAX_CHILD\n};
            print qq{waiting for them ALL to finish, then will resume ...\n};
            # "spin wait"
            while ( wait > 0 ) { }
            $COUNT = 0;
            print qq{.. done waiting\n};
        }
    }
}
if ( $COUNT > 0 ) {
    print qq{waiting for remaining $COUNT children\n};
    while ( wait > 0 ) { }
}
print qq{all done, exiting...\n};
```

"autoflush" for STDOUT

Call to `fork`

Child's body

`fork` returns:

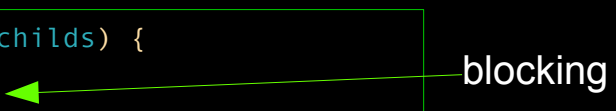
- "0" for the child process
- PID of child process for the Parent process

`waitpid`

- `perl` provides for additional precision in blocking in the parent
- Whereas `wait` proceeds if any child finishes or there are no children
- `waitpid` will wait for a specific child PID to complete

waitpid

```perl
#!/usr/bin/env perl

use strict;
use warnings;
++$|; #autoflush

my @childs = ();
for my $i ( 1 .. 10 ) {

    my $child_pid = fork();

  CHILD_SUBSHELL:
    if ( 0 == $child_pid ) {
        sleep 1;
        print qq{I am child process ($$)!\n};
        sleep 5;
        exit;    #<- if this is not here, the child will continue the loop
    }

  PARENT_PROCESS:
    if ( 0 < $child_pid ) {
        print qq{I am the parent process ($$)!\n};
        push @childs, $child_pid;
    }
}

foreach my $childpid (@childs) {
  # blocking wait
  waitpid($childpid,0);
  print qq{Child PID $childpid has completed\n};
}
print qq{...done\n}
```



waitpid

```perl
#!/usr/bin/env perl

use strict;
use warnings;
use POSIX ':sys_wait_h';
++$|; #autoflush

my @childs = ();
for my $i ( 1 .. 10 ) {
    my $child_pid = fork();
  CHILD_SUBSHELL:
    if ( 0 == $child_pid ) {
        sleep 1;
        print qq{I am child process ($$)!\n};
        sleep 5;
        exit;    #<- if this is not here, the child will continue the loop
    }
  PARENT_PROCESS:
    if ( 0 < $child_pid ) {
        print qq{I am the parent process ($$)!\n};
        push @childs, $child_pid;
    }
}
# treats @childs as a stack that readds child pids that have not
# yet finished
while (my $childpid = pop @childs) {
  # blocking wait
  if ( 0 ==  waitpid($childpid,WNOHANG) ) {
    print qq{Child PID $childpid still running, moving on...\n};
    push @childs, $childpid;
  }
  else {
    print qq{Child PID $childpid has completed\n};
  }
}

print qq{...done\n};
```

non-blocking

waitpid AND $SIG{CHLD}

```perl
#!/usr/bin/env perl

use strict;
use warnings;
++$|;    #autoflush
use POSIX ":sys_wait_h";

my %child_status = ();
$SIG{CHLD} = sub {
    while ( ( my $child = waitpid( -1, WNOHANG ) ) > 0 ) {
        $child_status{$child} = qq{Child process $child completed with status $?\n};
    }
};

my @childs = ();
for my $i ( 1 .. 10 ) {
    my $child_pid = fork();
  CHILD_SUBSHELL:
    if ( 0 == $child_pid ) {
        sleep 1;
        print qq{I am child process ($$)!\n};
        sleep 5;
        exit;    #<- if this is not here, the child will continue the loop
    }
  PARENT_PROCESS:
    if ( 0 < $child_pid ) {
        print qq{I am the parent process ($$)!\n};
        push @childs, $child_pid;
    }
}

while ( 0 < wait ) {};

foreach my $child (keys %child_status) {
  print $child_status{$child};
}
print qq{...done\n};
```

Non-blocking `waitpid` in `$SIG{CHLD}` handler

Blocking "spin" wait

Strengths and advantages of using **bash**

- Straightforward
- Child processes are isolated from execution context (no accidentaly running of current script in child)
- Does what I mean (e.g., `wait` is “wait for all”)
- Less flexibility means it's hard to get too complex without meaning to

Strengths and Advantages of Using `perl`

- Perl language makes if much easier to manage child processes to achieve maximum throughput (target 100% active child PIDs for duration, load balancing, etc)
- CPAN is full of interesting “helper” modules for managing external child processes,

`fork` *vs.* `system`

- Perl provides several ways to spawn subprocesses: `fork`, `system`, and `` `command` ``
- `` `command` `` (*backticks*) semantics is also provided for in the shell (e.g., `bash`)
- `system` and `` `command` `` facilities in Perl are strictly for launching subshells in which the generic commands are executed
- Perl's `fork` starts a *new* `perl` interpreter and copies the current 'context' (variables, etc) to it

`fork` "CONTEXT"

- Perl's `fork` starts a *new* `perl` interpreter and copies the current 'context' (variables, etc) to it
- What does this mean?
- It *means* that it is cloning the current execution of the `perl` interpreter ("the *script*") and that the "child" `perl` process:
  - Running the same script starting from the call to `fork`
  - Maintains knowledge of all variables and program states

Parent-Child Communication and `fork`

- Althought we can set `$SIG{CHLD}`, that's often not sufficient
- There is no “interprocess communication” after fork (unlink in real shared memory threads)
- But the parent can completely control the *state* of the child process at the time of creation; e.g. *variables*
- In this way, `fork` can be said to be a *deep clone* the parent executing `perl` process (full copy of fork, there are no references preserved)
- IPC::Fork::Simple looks interesting, but it's not covered here

Note: facilitating IPC between parent and child is possible, but is a full talk itself (maybe as a follow up to this one) – spoiler: named pipes, “freeze/thaw” to disk, redis, “Mqs”, databases, etc

Parent-child communication and `fork`

What we want.

What we get.

When to use `fork` in `perl`

- You have a lot of resource intensive “tasks” to perform
- You have access to a “bare metal” machine with many cores (or many virtual CPUs on somebody else's computer – `s/cloud/butt/`)
- Task can be dispatched asynchronously and no IPC is required

EXAMPLE TASKS

- Downloading from many URLs (http, ftp, etc)
- Uploading many files to multiple resources (e.g., back ups to cloud, etc)
- Processing many images, documents, or other files
- Regular system-wide crons or “periodic” scripts that affect a large number of users

PART II

`Parallel::ForkManager`

- Implemented as a very light wrapper around `fork`
- Makes it straightforward and easy to schedule work via `fork` efficiently:
  - Set maximum number of child processes
  - Precise blocking (to maximize system resources)
  - Specify communication back from child processes (via `Storable`)
  - Parent level, “event” based callbacks (`run_on_wait`, `run_on_start`)

Parallel::ForkManager

```perl
#!/usr/bin/env perl

use strict;
use warnings;
use Parallel::ForkManager;
++$|;    # autoflush for STDOUT

my $pm = Parallel::ForkManager->new(4);

for my $i ( 1 .. 10 ) {
    print qq{I am the parent process ($$)!\n};
    $pm->start and next; # child proceeds, parent process returns to top of loop
    sleep 1;
    print qq{I am child process ($$)!\n};
    sleep 5;
    $pm->finish;         # terminates child process
}
```

Child's body

Parallel::ForkManager::Segmented

- Built around Parallel::ForkManager
- Not a subclass
- Primary purpose is for a given list of items (anything in an array or list):
  - Spawn `$nproc` works (# of `forks`)
  - Process `$batch_size` per worker spawned
  - Using subroutine reference specified by `$process_item`

Parallel::ForkManager::Segmented

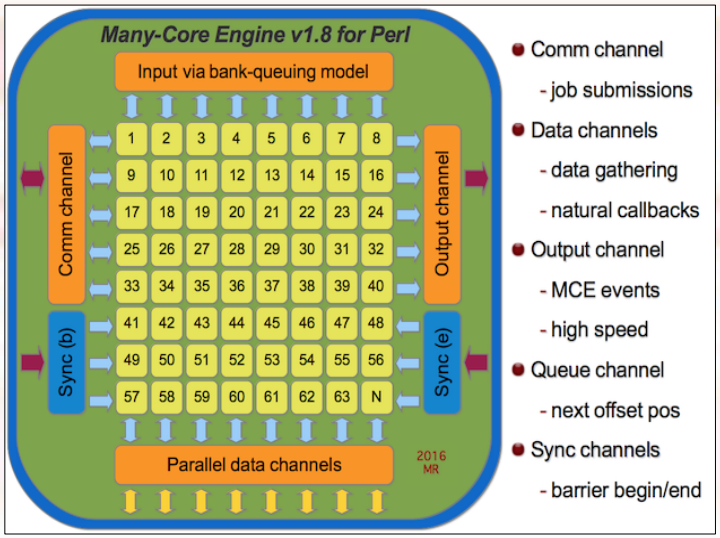
```perl
#!/usr/bin/env perl

use strict;
use warnings;
use Parallel::ForkManager::Segmented;
++$|;    # autoflush for STDOUT

# define list of items (here it's just a list of numbers)
my @queue = ( 1 .. 55 );

# kick off processing
Parallel::ForkManager::Segmented->new->run(
    {
        WITH_PM => 1,         # required to invoke Parallel::ForkManage, othewise it's serial
        items   => \@queue,

        # number of worker processes at any given time
        nproc => 4,

        # chunks of work, per fork
        batch_size => 5,

        # subroutine reference for processing each item in @queue
        process_item => sub {
            my $item = shift;
            sleep 1;
            print qq{I am child process ($$) - items $item!\n};
            sleep 5;
            return;
        },
    }
);
```

Child's body

MCE - Perl Multi-Core Engine

- Fundamentally fork based, but maintains a pool of worker processes that can coordinate and communicate
- Can be coupled with async frameworks like AnyEvent
- Basically, `Parallel::ForkManager::Segmented` on steroids
- Looks well suited to implement things like map/reduce, definitely on a “higher level” than

MCE – Perl Multi-Core Engine



"MCE spawns a pool of workers and therefore does not fork a new process per each element of data. Instead, MCE follows a bank queuing model. Imagine the line being the data and bank-tellers the parallel workers. MCE enhances that model by adding the ability to chunk the next n elements from the input stream to the next available worker."

https://metacpan.org/pod/MCE

MCE - Perl Multi-Core Engine

```perl
#!/usr/bin/env perl

use strict;
use warnings;

use MCE;

my $mce = MCE->new(
   max_workers => 4,
   user_func => sub {
      my ($mce) = @_;
      $mce->say(sprintf(qq{I am child process %s (Worker Id is %s).}, $$, $mce->wid));
   }
);

$mce->run;
```

MCE - Perl Multi-Core Engine

Documentation

MCE::Core	Documentation describing the core MCE API
MCE::Examples	Various examples and demonstrations

Modules

MCE	Many-Core Engine for Perl providing parallel processing capabilities
MCE::Candy	Sugar methods and output iterators
MCE::Channel	Queue-like and two-way communication capability
MCE::Channel::Mutex	Channel for producer(s) and many consumers
MCE::Channel::Simple	Channel tuned for one producer and one consumer
MCE::Channel::Threads	Channel for producer(s) and many consumers
MCE::Child	A threads-like parallelization module compatible with Perl 5.8
MCE::Core::Input::Generator	Sequence of numbers (for task_id > 0)
MCE::Core::Input::Handle	File path and Scalar reference input reader
MCE::Core::Input::Iterator	Iterator reader
MCE::Core::Input::Request	Array reference and Glob reference input reader
MCE::Core::Input::Sequence	Sequence of numbers (for task_id == 0)
MCE::Core::Manager	Core methods for the manager process
MCE::Core::Validation	Core validation methods for Many-Core Engine
MCE::Core::Worker	Core methods for the worker process
MCE::Flow	Parallel flow model for building creative applications
MCE::Grep	Parallel grep model similar to the native grep function
MCE::Loop	MCE model for building parallel loops
MCE::Map	Parallel map model similar to the native map function
MCE::Mutex	Locking for Many-Core Engine
MCE::Mutex::Channel	Mutex locking via a pipe or socket
MCE::Mutex::Channel2	Provides two mutexes using a single channel
MCE::Mutex::Flock	Mutex locking via Fcntl
MCE::Queue	Hybrid (normal and priority) queues
MCE::Relay	Extends Many-Core Engine with relay capabilities
MCE::Signal	Temporary directory creation/cleanup and signal handling
MCE::Step	Parallel step model for building creative steps
MCE::Stream	Parallel stream model for chaining multiple maps and greps
MCE::Subs	Exports functions mapped directly to MCE methods
MCE::Util	Utility functions

Sampling of interesting modules

- `Proc::Fork`
- `Fork::Promise`
- `IPC::Fork::Simple`
- `fork::hook`
- Much of the `Parallel::*` name space
- `Coro`
- `AnyEvent`
- `POE`
- `PDL::Parallel::MPI`

THANK YOU