Adding Forks to Your Perly Tableware

Handling fixed-width records in parallel

Steven Lembark
Workhorse Computing
lembark@wrkhors.com

You may hear things like:

Perl is too slow for processing fixed-width records so I wrote the latest version in C++ since it can handle the data in parallel.

There are a few misunderstandings here:

- That Perl is slow.
- That Perl does not handle parallel processing well.
- That Perl does not handle fixed-width records.

It isn't, does, and can.

Comparing C++ and Perl:

Perl's I/O is a fairly thin layer over unistd.h library calls.

Perl and C++ block at the same rate.

Forked processes can easily share data stream with separate file handles by letting the O/S buffer data.

unpack is reasonably efficient and more dynamic than using struct's in C++.

Most of us use variable-width, delimited records.

These are the usual newline-terminated data we all know and [lh][oa][vt]e.

Perl handles these via $\, readline, and split or regexen.

Common examples: logs or FASTA and FASTQ.

Read using the buffered I/O with readline or read.

Used for large records, variable or self-described data.

Fixed-Width Records

Fixed number of *bytes* per record.

Small records with space- or zero-padding per field.

Common in financial data – derived from card images used on mainframe systems.

Record sizes tend to be small.

Files with lots of rows leaving them “tall & narrow”.

Fixed-width reads

Perl can read them with readline, read, or sysread.

read() uses Perl's buffered I/O to read *characters*.

sysread() bypasses buffered I/O and reads *bytes*.

$/ does record-based character I/O with maximum record size if the O/S supports it.

The thinnest layer over the O/S is sysread.

UTF8: When fixed-width isn't

Traditional fixed-width data has fixed *bytes* per record.

Records read into a fixed-width buffer N bytes at a time.

UTF8 has *characters* not *bytes*.

Result: Use read() with I/O disciplines to deal with data that may contain UTF8-encoded strings.

Buffered I/O system deals with layered I/O and disciplines.

Using sysread

Copy up to N bytes from file handle as-is into a buffer:

```
sysread FILEHANDLE,SCALAR,LENGTH,OFFSET
```

Bypasses process buffers and file handles.

On *NIX, Perl's sysread is a thin layer over read(2).

Copies data from the kernel buffer to process space.

Sharing a buffer

Common view: threads & shared memory “right” way.

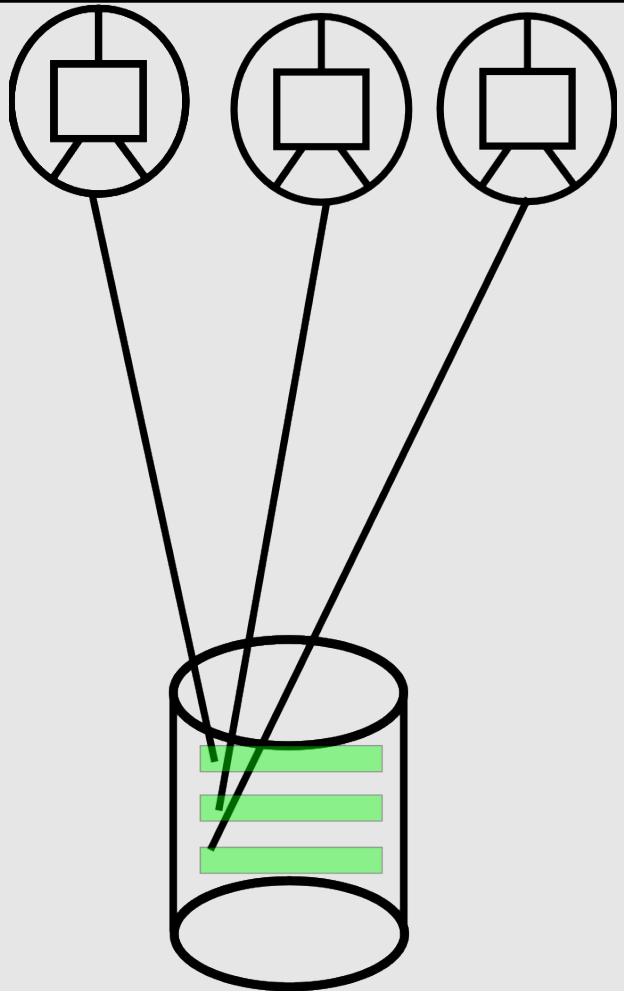
Threads share the filehandle and in-process buffer.

- Threads are a lot of work to program & test
- Locking overhead may kill any time advantage.

Small records can share a kernel buffer.

Reads from the data stream pull in record-size chunks.

Traditional view of file handles

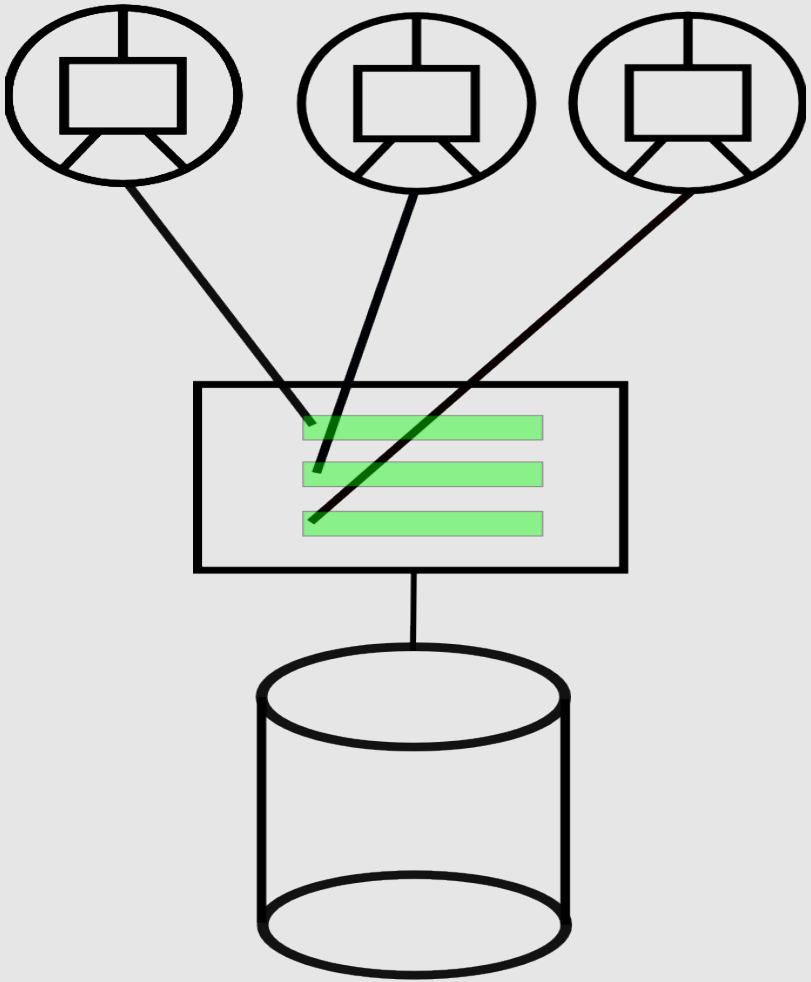


Handles connect to hardware.

Buffer the data in their own space for efficiency.

readline and read use their own buffers for this reason.

Kernel buffers make a big difference



Modern O/S use memory-mapped I/O to a kernel buffer.

Data transfer via memcpy to userspace.

Data faulted into the buffer.

File handles read from the kernel buffer, *not* the device.

Example: Stock Trading Data

Table 1 Daily TAQ File Details

FILE	FORMAT	RECORD SIZE (BYTES)	FTP SIZE (COMPRESSED)	NUMBER OF ROWS	FILE TIME AVAILABILITY (EST)
TAQ Master	ASCII	252	360 KB	8000	8pm (20:00)
TAQ Master Beta	ASCII	Variable – Pipe Delimited	Approximately: *.txt - 750kb *.xls - 2.6mb	8000	Midnight (00:00)
TAQ Quotes	ASCII	96	6 GB	550 million	11pm (23:00)
TAQ Trades	ASCII	73	200 MB	24 million	9pm (21:00)
TAQ NBBO	ASCII	146	1.2 GB	110 million	11pm (23:00)

From the NYX daily files documentation.

Kernel buffers hold ~42 Quotes in an 4KB page.

Reads from multiple file handles will hit the buffer more often than the disk.

Data Format

Most of the fields are single-char tags.

Record has small memory footprint.

Field	Offset	Size	Format	
time	0	9	hhmmssXXX	msec
exchange	9	1	char	dictionary
symbol	10	16	char	6+10
bid price	26	11	float	%.04f
bid size	37	7	int	%7d
ask price	44	11	float	%.04f
ask size	55	11	int	%7d
quote condition	62	1	char	dictionary
filler	63	4	text	four blanks
bid exchange	67	1	char	dictionary
ask exchange	68	1	char	dictionary
sequence no	69	16	int	%d
national bbo	85	1	digit	dictionary
nasdaq bbo	86	1	digit	dictionary
cancel/correct	87	1	char	dictionary
source	88	1	char	dictionary
retal int flag	89	1	char	dictionary
short sale flag	90	1	char	dictionary
CQS	91	1	char	dictionary
UTP	92	1	char	dictionary
finra adf	93	1	char	dictionary
line	94	2	text	\cM\cJ

Handling records

Unpack is the fastest way to split this up.

“A” is a space-padded ascii character sequence.

“x” skips a number of bytes.

No math on the values: don't convert to C int or float.

DBI stringifies all values anyway.

Also need to discard fixed with header record.

Parallel processing

Forks are actually simpler than they sound.

CPAN makes them even simpler:

I'll use Parallel::Queue for the examples here.

This takes a list of closures (or a job-creator object).

It forks them N-way parallel, aborting the queue if any jobs exit non-zero.

Deals with exit (vs. return) to avoid phorktosis.

Simplest approach: Manager + Worker

A manager process forks off workers to read the data and then cleans up after them.

Workers are given a filehandle.

Each worker reads a single record with sysread.

The natural order of reads will have most of the proc's doing buffer copies most of the time.

```
my @fieldz =
(
    [ qw( time A 9 ) ],
    [ qw( exch A 1 ) ],
    ...
    [ qw( fill x 4 ) ],
    ...
);


my $template
= join '',
map
{
    join '', @{ $_ }[ 1, 2 ];
}
@fieldz;


my $size = sum map { $_->[2] } @fieldz;
```

Describing record

Template uses space-padded values.

“A” and a width for data loaded with DBI.

“x” ignores filler.

Read size == sum of A & x fields.

Dispatching reads.

```
#!/bin/env perl
use v5.22;
use autodie;
use Parallel::Queue;

my ( $path, $jobs ) = getopt ...;

my @fieldz   = ... ;
my $template = ... ;
my $size     = ... ;
my $buffer   = '';

open my $fh, '<', $path;
sysread $fh, $buffer, 92   # fixed header
// die "Failed header: $!";

sub read_recs { ... }

my @queue = ( sub { read_recs ) x $jobs;

my @failed = runqueue $jobs, @queue
or die 'Failed:', \@unfinished;
```

Closures dispatch reads.

Pass the queue to P::Q.

Failed jobs are returned as a list of closures.

Production code needs check on $. for restarts!

```
sub read_recs
{
  my $dbh    = DBI->connect( ... );
  my $sth    = $dbh->prepare( ... );
  my $i      = 0

  for(;;)
  {
    $i =  sysread $fh, $buffer, $size
    // die;
    $i or last;
    $i == $size
    or die "Runt: $i at $.\n";

    $sth->execute
    (
      unpack $template => $buffer
    );
  }
  return
}
```

Reading buffers

Not much code.

sysread pulls whole records.

Sync via blocking.

Each process gets its own $dbh, $sth.

What if the data is zipped?

Same basic process: share a filehandle.

```
open my $fh, '-|', “gzip -dc $path”;
```

After that fork and share a file handle.

Named pipes or *filesystem* sockets also be useful.

Network sockets have packet issues with record boundaries – server needs to pre-chunk the data.

Improving DBI performance

Calling $sth->execute(@_) is expensive.

The faster approach is calling bind and using lexical variables for the output...

But that doesn't fit onto a single slide.

Quick fix: bind array elements with “`\$row[$i]`”

Saves managing a dozen+ lexical variables.

Half-billion records makes this worth benchmarking.

Multiple blocks help reduce overhead

Read N records < system page size.

Kernel call overhead more than larger memcopy.

Multiple records avoid starving jobs.

Check read with ! ($i % $size).

Apply template with substr or multiply template and chunk array.

Using a job-object with P::Q

Generate the jobs as needed.

Blessed queue entry that can('next_job').

This can be handy for processing multiple files:

The original “queue” has an object for each file.

N jobs generated for each file by handler object.

```
open my $fh, '<', $path;
my $size    = ... ;
my $buffer  = ' ' $size;
my $i       = 0;
my $j       = 0;
my $wall    = 0;
sysread $fh, $buffer, 92;  # discard header

for(;;)
{
    $i  = sysread $fh, $buffer, $size
    or last;

    $i == $size
    or die "Runt read: $i at $.";

    @row    = unpack $template => $buffer;

    ++$j % 65_536
    and next;

    $wall    = tv_interval $t0, [ gettimeofday ];

    say "$$ $j / $wall = " . int( $j / $wall );
}
$i // die "Failed read: $!";

say "Final: $$ $j / $wall = " . int( $j / $wall );
```

Simple benchmark

Read & split rows.

Ignore DBI overhead.

Vary buffer size to check chunked read speed.

Loop with fork to check number of jobs.

Running a million records through...

```
13455 4 65536 / 1.077347 = 60830
13452 1 65536 / 1.127445 = 58127
13453 2 65536 / 1.324299 = 49487
13454 3 65536 / 1.739874 = 37667
13455 4 131072 / 2.139257 = 61269
13452 1 131072 / 2.188887 = 59880
13453 2 131072 / 2.383899 = 54982
13454 3 131072 / 3.157574 = 41510
13455 4 196608 / 3.19973 = 61445
13452 1 196608 / 3.251169 = 60473
13453 2 196608 / 3.445435 = 57063
13454 3 196608 / 4.21482 = 46646
13455 4 262144 / 4.274705 = 61324
13452 1 262144 / 4.303842 = 60909

Final: 13455 4 268606 / 4.380693 = 61315
Final: 13454 3 207351 / 4.389554 = 47237
Final: 13452 1 267971 / 4.398593 = 60921
Final: 13453 2 256075 / 4.397525 = 58231
```

~ 4.4 sec wallclock for 1Mrec.

About 3Ksec for the daily file's full 550Mrec.

Un-zipping this many records to /dev/null takes about 1.6sec.

10Mrec looks about the same

```
13182 4 2424832 / 39.453094 = 61461
13181 3 2490368 / 39.666493 = 62782
13179 1 2490368 / 40.221195 = 61916
13180 2 2490368 / 40.26747 = 61845
```

Better per-process balance with longer running tasks.

Still ~60KHz per process.

Summary: it's easier than you think.

sysread makes reading fixed records easy.

Parallel::Queue makes it simple to fork.

Kernel buffering makes the streaming data efficient.

No need for shared memory or locking.

Net result: parallel processing of fixed-width data in Perl is actually pretty easy.

References

As always, perldoc is your friend:

- perlpacktut
- perlperf
- perlfork

Try “perldoc perl” just to see how much there is!