NAME

IPC::Run – system() and background procs w/ piping, redirs, ptys (Unix, Win32)

SYNOPSIS

```
## First,a command to run:
   my @cat = qw( cat );

## Using run() instead of system():
   use IPC::Run qw( run timeout );

   run \@cat, \$in, \$out, \$err, timeout( 10 ) or die "cat: $?"

   # Can do I/O to sub refs and filenames, too:
   run \@cat, '<', "in.txt", \&out, \&err or die "cat: $?"
   run \@cat, '<', "in.txt", '>>', "out.txt", '2>>', "err.txt";


   # Redirecting using pseudo-terminals instead of pipes.
   run \@cat, '<pty<', \$in,  '>pty>', \$out_and_err;

## Scripting subprocesses (like Expect):

   use IPC::Run qw( start pump finish timeout );

   # Incrementally read from / write to scalars.
   # $in is drained as it is fed to cat's stdin,
   # $out accumulates cat's stdout
   # $err accumulates cat's stderr
   # $h is for "harness".
   my $h = start \@cat, \$in, \$out, \$err, timeout( 10 );

   $in .= "some input\n";
   pump $h until $out =~ /input\n/g;

   $in .= "some more input\n";
   pump $h until $out =~ /\G.*more input\n/;

   $in .= "some final input\n";
   finish $h or die "cat returned $?";

   warn $err if $err;
   print $out;         ## All of cat's output

# Piping between children
   run \@cat, '|', \@gzip;

# Multiple children simultaneously (run() blocks until all
# children exit, use start() for background execution):
   run \@foo1, '&', \@foo2;

# Calling \&set_up_child in the child before it executes the
# command (only works on systems with true fork() & exec())
# exceptions thrown in set_up_child() will be propagated back
# to the parent and thrown from run().
   run \@cat, \$in, \$out,
```

```
      init => \&set_up_child;

# Read from / write to file handles you open and close
   open IN,  '<in.txt'  or die $!;
   open OUT, '>out.txt' or die $!;
   print OUT "preamble\n";
   run \@cat, \*IN, \*OUT or die "cat returned $?";
   print OUT "postamble\n";
   close IN;
   close OUT;

# Create pipes for you to read / write (like IPC::Open2 & 3).
   $h = start
      \@cat,
         '<pipe', \*IN, # may also be a lexical filehandle e.g. \my $infh
         '>pipe', \*OUT,
         '2>pipe', \*ERR
      or die "cat returned $?";
   print IN "some input\n";
   close IN;
   print <OUT>, <ERR>;
   finish $h;

# Mixing input and output modes
   run \@cat, 'in.txt', \&catch_some_out, \*ERR_LOG );

# Other redirection constructs
   run \@cat, '>&', \$out_and_err;
   run \@cat, '2>&1';
   run \@cat, '0<&3';
   run \@cat, '<&-';
   run \@cat, '3<', \$in3;
   run \@cat, '4>', \$out4;
   # etc.

# Passing options:
   run \@cat, 'in.txt', debug => 1;

# Call this system's shell, returns TRUE on 0 exit code
# THIS IS THE OPPOSITE SENSE OF system()'s RETURN VALUE
   run "cat a b c" or die "cat returned $?";

# Launch a sub process directly, no shell.  Can't do redirection
# with this form, it's here to behave like system() with an
# inverted result.
   $r = run "cat a b c";

# Read from a file in to a scalar
   run io( "filename", 'r', \$recv );
   run io( \*HANDLE,   'r', \$recv );
```

DESCRIPTION

IPC::Run allows you to run and interact with child processes using files, pipes, and pseudo-ttys. Both **system()**-style and scripted usages are supported and may be mixed. Likewise, functional and OO API styles are both supported and may be mixed.

Various redirection operators reminiscent of those seen on common Unix and DOS command lines are provided.

Before digging in to the details a few LIMITATIONS are important enough to be mentioned right up front:

Win32 Support
: Win32 support is working but **EXPERIMENTAL**, but does pass all relevant tests on NT 4.0. See "Win32 LIMITATIONS".

pty Support
: If you need pty support, IPC::Run should work well enough most of the time, but IO::Pty is being improved, and IPC::Run will be improved to use IO::Pty's new features when it is release.

  The basic problem is that the pty needs to initialize itself before the parent writes to the master pty, or the data written gets lost. So IPC::Run does a **sleep**(1) in the parent after forking to (hopefully) give the child a chance to run. This is a kludge that works well on non heavily loaded systems :(.

  ptys are not supported yet under Win32, but will be emulated...

Debugging Tip
: You may use the environment variable `IPCRUNDEBUG` to see what's going on under the hood:

```
$ IPCRUNDEBUG=basic   myscript     # prints minimal debugging
$ IPCRUNDEBUG=data    myscript     # prints all data reads/writes
$ IPCRUNDEBUG=details myscript     # prints lots of low-level details
$ IPCRUNDEBUG=gory    myscript     # (Win32 only) prints data moving through
                                   # the helper processes.
```

We now return you to your regularly scheduled documentation.

Harnesses
: Child processes and I/O handles are gathered in to a harness, then started and run until the processing is finished or aborted.

run() vs. start(); pump(); finish();
: There are two modes you can run harnesses in: **run()** functions as an enhanced **system()**, and **start()/pump()/finish()** allow for background processes and scripted interactions with them.

  When using **run()**, all data to be sent to the harness is set up in advance (though one can feed subprocesses input from subroutine refs to get around this limitation). The harness is run and all output is collected from it, then any child processes are waited for:

```
run \@cmd, \<<IN, \$out;
blah
IN

## To precompile harnesses and run them later:
my $h = harness \@cmd, \<<IN, \$out;
blah
IN

run $h;
```

  The background and scripting API is provided by **start()**, **pump()**, and **finish()**: **start()** creates a harness if need be (by calling **harness()**) and launches any subprocesses, **pump()** allows you to poll them for activity, and **finish()** then monitors the harnessed activities until they complete.

```
## Build the harness, open all pipes, and launch the subprocesses
my $h = start \@cat, \$in, \$out;
$in = "first input\n";

## Now do I/O.  start() does no I/O.
pump $h while length $in;  ## Wait for all input to go
```

```
## Now do some more I/O.
$in = "second input\n";
pump $h until $out =~ /second input/;

## Clean up
finish $h or die "cat returned $?";
```

You can optionally compile the harness with **harness()** prior to **start()**ing or **run()**ing, and you may omit **start()** between **harness()** and **pump()**. You might want to do these things if you compile your harnesses ahead of time.

Using regexps to match output

As shown in most of the scripting examples, the read-to-scalar facility for gathering subcommand's output is often used with regular expressions to detect stopping points. This is because subcommand output often arrives in dribbles and drabs, often only a character or line at a time. This output is input for the main program and piles up in variables like the `$out` and `$err` in our examples.

Regular expressions can be used to wait for appropriate output in several ways. The `cat` example in the previous section demonstrates how to **pump()** until some string appears in the output. Here's an example that uses `smb` to fetch files from a remote server:

```
$h = harness \@smbclient, \$in, \$out;

$in = "cd /src\n";
$h->pump until $out =~ /^smb.*> \Z/m;
die "error cding to /src:\n$out" if $out =~ "ERR";
$out = '';

$in = "mget *\n";
$h->pump until $out =~ /^smb.*> \Z/m;
die "error retrieving files:\n$out" if $out =~ "ERR";

$in = "quit\n";
$h->finish;
```

Notice that we carefully clear `$out` after the first command/response cycle? That's because IPC::Run does not delete `$out` when we continue, and we don't want to trip over the old output in the second command/response cycle.

Say you want to accumulate all the output in `$out` and analyze it afterwards. Perl offers incremental regular expression matching using the `m//gc` and pattern matching idiom and the `\G` assertion. IPC::Run is careful not to disturb the current `pos()` value for scalars it appends data to, so we could modify the above so as not to destroy `$out` by adding a couple of `/gc` modifiers. The `/g` keeps us from tripping over the previous prompt and the `/c` keeps us from resetting the prior match position if the expected prompt doesn't materialize immediately:

```
$h = harness \@smbclient, \$in, \$out;

$in = "cd /src\n";
$h->pump until $out =~ /^smb.*> \Z/mgc;
die "error cding to /src:\n$out" if $out =~ "ERR";

$in = "mget *\n";
$h->pump until $out =~ /^smb.*> \Z/mgc;
die "error retrieving files:\n$out" if $out =~ "ERR";

$in = "quit\n";
$h->finish;
```

```
   analyze( $out );
```

When using this technique, you may want to preallocate `$out` to have plenty of memory or you may find that the act of growing `$out` each time new input arrives causes an `O(length($out)^2)` slowdown as `$out` grows. Say we expect no more than 10,000 characters of input at the most. To preallocate memory to `$out`, do something like:

```
   my $out = "x" x 10_000;
   $out = "";
```

`perl` will allocate at least 10,000 characters' worth of space, then mark the `$out` as having 0 length without freeing all that yummy RAM.

Timeouts and Timers

More than likely, you don't want your subprocesses to run forever, and sometimes it's nice to know that they're going a little slowly. Timeouts throw exceptions after a some time has elapsed, timers merely cause **pump()** to return after some time has elapsed. Neither is reset/restarted automatically.

Timeout objects are created by calling timeout(`$interval`) and passing the result to **run()**, **start()** or **harness()**. The timeout period starts ticking just after all the child processes have been **fork()**ed or **spawn()**ed, and are polled for expiration in **run()**, **pump()** and **finish()**. If/when they expire, an exception is thrown. This is typically useful to keep a subprocess from taking too long.

If a timeout occurs in **run()**, all child processes will be terminated and all file/pipe/ptty descriptors opened by **run()** will be closed. File descriptors opened by the parent process and passed in to **run()** are not closed in this event.

If a timeout occurs in **pump()**, **pump_nb()**, or **finish()**, it's up to you to decide whether to **kill_kill()** all the children or to implement some more graceful fallback. No I/O will be closed in **pump()**, **pump_nb()** or **finish()** by such an exception (though I/O is often closed down in those routines during the natural course of events).

Often an exception is too harsh. timer(`$interval`) creates timer objects that merely prevent **pump()** from blocking forever. This can be useful for detecting stalled I/O or printing a soothing message or "." to pacify an anxious user.

Timeouts and timers can both be restarted at any time using the timer's **start()** method (this is not the **start()** that launches subprocesses). To restart a timer, you need to keep a reference to the timer:

```
   ## Start with a nice long timeout to let smbclient connect.  If
   ## pump or finish take too long, an exception will be thrown.

 my $h;
 eval {
   $h = harness \@smbclient, \$in, \$out, \$err, ( my $t = timeout 30 );
   sleep 11;  # No effect: timer not running yet

   start $h;
   $in = "cd /src\n";
   pump $h until ! length $in;

   $in = "ls\n";
   ## Now use a short timeout, since this should be faster
   $t->start( 5 );
   pump $h until ! length $in;

   $t->start( 10 );  ## Give smbclient a little while to shut down.
   $h->finish;
 };
```

```
if ( $@ ) {
  my $x = $@;     ## Preserve $@ in case another exception occurs
  $h->kill_kill; ## kill it gently, then brutally if need be, or just
                  ## brutally on Win32.
  die $x;
}
```

Timeouts and timers are *not* checked once the subprocesses are shut down; they will not expire in the interval between the last valid process and when IPC::Run scoops up the processes' result codes, for instance.

Spawning synchronization, child exception propagation

start() pauses the parent until the child executes the command or CODE reference and propagates any exceptions thrown (including **exec()** failure) back to the parent. This has several pleasant effects: any exceptions thrown in the child, including **exec()** failure, come flying out of **start()** or **run()** as though they had occurred in the parent.

This includes exceptions your code thrown from init subs. In this example:

```
eval {
   run \@cmd, init => sub { die "blast it! foiled again!" };
};
print $@;
```

the exception ''blast it! foiled again'' will be thrown from the child process (preventing the **exec()**) and printed by the parent.

In situations like

```
run \@cmd1, "|", \@cmd2, "|", \@cmd3;
```

`@cmd1` will be initted and **exec()**ed before `@cmd2`, and `@cmd2` before `@cmd3`. This can save time and prevent oddball errors emitted by later commands when earlier commands fail to execute. Note that IPC::Run doesn't start any commands unless it can find the executables referenced by all commands. These executables must pass both the `-f` and `-x` tests described in perlfunc.

Another nice effect is that **init()** subs can take their time doing things and there will be no problems caused by a parent continuing to execute before a child's **init()** routine is complete. Say the **init()** routine needs to open a socket or a temp file that the parent wants to connect to; without this synchronization, the parent will need to implement a retry loop to wait for the child to run, since often, the parent gets a lot of things done before the child's first timeslice is allocated.

This is also quite necessary for pseudo-tty initialization, which needs to take place before the parent writes to the child via pty. Writes that occur before the pty is set up can get lost.

A final, minor, nicety is that debugging output from the child will be emitted before the parent continues on, making for much clearer debugging output in complex situations.

The only drawback I can conceive of is that the parent can't continue to operate while the child is being initted. If this ever becomes a problem in the field, we can implement an option to avoid this behavior, but I don't expect it to.

Win32: executing CODE references isn't supported on Win32, see ''Win32 LIMITATIONS'' for details.

Syntax

run(), **start()**, and **harness()** can all take a harness specification as input. A harness specification is either a single string to be passed to the systems' shell:

```
run "echo 'hi there'";
```

or a list of commands, io operations, and/or timers/timeouts to execute. Consecutive commands must be separated by a pipe operator '|' or an '&'. External commands are passed in as array references, and, on systems supporting **fork()**, Perl code may be passed in as subs:

```
run \@cmd;
run \@cmd1, '|', \@cmd2;
run \@cmd1, '&', \@cmd2;
run \&sub1;
run \&sub1, '|', \&sub2;
run \&sub1, '&', \&sub2;
```

'|' pipes the stdout of \@cmd1 the stdin of \@cmd2, just like a shell pipe. '&' does not. Child processes to the right of a '&' will have their stdin closed unless it's redirected-to.

IPC::Run::IO objects may be passed in as well, whether or not child processes are also specified:

```
run io( "infile", ">", \$in ), io( "outfile", "<", \$in );
```

as can IPC::Run::Timer objects:

```
run \@cmd, io( "outfile", "<", \$in ), timeout( 10 );
```

Commands may be followed by scalar, sub, or i/o handle references for redirecting child process input & output:

```
run \@cmd,  \undef,            \$out;
run \@cmd,  \$in,              \$out;
run \@cmd1, \&in, '|', \@cmd2, \*OUT;
run \@cmd1, \*IN, '|', \@cmd2, \&out;
```

This is known as succinct redirection syntax, since **run**(), **start**() and **harness**(), figure out which file descriptor to redirect and how. File descriptor 0 is presumed to be an input for the child process, all others are outputs. The assumed file descriptor always starts at 0, unless the command is being piped to, in which case it starts at 1.

To be explicit about your redirects, or if you need to do more complex things, there's also a redirection operator syntax:

```
run \@cmd, '<', \undef, '>',  \$out;
run \@cmd, '<', \undef, '>&', \$out_and_err;
run(
   \@cmd1,
      '<', \$in,
   '|', \@cmd2,
      \$out
);
```

Operator syntax is required if you need to do something other than simple redirection to/from scalars or subs, like duping or closing file descriptors or redirecting to/from a named file. The operators are covered in detail below.

After each \@cmd (or \&foo), parsing begins in succinct mode and toggles to operator syntax mode when an operator (ie plain scalar, not a ref) is seen. Once in operator syntax mode, parsing only reverts to succinct mode when a '|' or '&' is seen.

In succinct mode, each parameter after the \@cmd specifies what to do with the next highest file descriptor. These File descriptor start with 0 (stdin) unless stdin is being piped to (`'|', \@cmd`), in which case they start with 1 (stdout). Currently, being on the left of a pipe (`\@cmd, \$out, \$err, '|'`) does *not* cause stdout to be skipped, though this may change since it's not as DWIMerly as it could be. Only stdin is assumed to be an input in succinct mode, all others are assumed to be outputs.

If no piping or redirection is specified for a child, it will inherit the parent's open file handles as dictated by your system's close-on-exec behavior and the $^F flag, except that processes after a '&' will not inherit the parent's stdin. Also note that $^F does not affect file descriptors obtained via POSIX, since it only applies to full-fledged Perl file handles. Such processes will have their stdin closed unless it has been redirected-to.

If you want to close a child processes stdin, you may do any of:

```
run \@cmd, \undef;
run \@cmd, \"";
run \@cmd, '<&-';
run \@cmd, '0<&-';
```

Redirection is done by placing redirection specifications immediately after a command or child subroutine:

```
run \@cmd1,       \$in, '|', \@cmd2,       \$out;
run \@cmd1, '<', \$in, '|', \@cmd2, '>', \$out;
```

If you omit the redirection operators, descriptors are counted starting at 0. Descriptor 0 is assumed to be input, all others are outputs. A leading '|' consumes descriptor 0, so this works as expected.

```
run \@cmd1, \$in, '|', \@cmd2, \$out;
```

The parameter following a redirection operator can be a scalar ref, a subroutine ref, a file name, an open filehandle, or a closed filehandle.

If it's a scalar ref, the child reads input from or sends output to that variable:

```
$in = "Hello World.\n";
run \@cat, \$in, \$out;
print $out;
```

Scalars used in incremental (**start()/pump()/finish()**) applications are treated as queues: input is removed from input scalers, resulting in them dwindling to '', and output is appended to output scalars. This is not true of harnesses **run()** in batch mode.

It's usually wise to append new input to be sent to the child to the input queue, and you'll often want to zap output queues to '' before pumping.

```
$h = start \@cat, \$in;
$in = "line 1\n";
pump $h;
$in .= "line 2\n";
pump $h;
$in .= "line 3\n";
finish $h;
```

The final call to **finish()** must be there: it allows the child process(es) to run to completion and waits for their exit values.

OBSTINATE CHILDREN

Interactive applications are usually optimized for human use. This can help or hinder trying to interact with them through modules like IPC::Run. Frequently, programs alter their behavior when they detect that stdin, stdout, or stderr are not connected to a tty, assuming that they are being run in batch mode. Whether this helps or hurts depends on which optimizations change. And there's often no way of telling what a program does in these areas other than trial and error and occasionally, reading the source. This includes different versions and implementations of the same program.

All hope is not lost, however. Most programs behave in reasonably tractable manners, once you figure out what it's trying to do.

Here are some of the issues you might need to be aware of.

- **fflush()**ing stdout and stderr

  This lets the user see stdout and stderr immediately. Many programs undo this optimization if stdout is not a tty, making them harder to manage by things like IPC::Run.

  Many programs decline to fflush stdout or stderr if they do not detect a tty there. Some ftp commands do this, for instance.

  If this happens to you, look for a way to force interactive behavior, like a command line switch or command. If you can't, you will need to use a pseudo terminal ('<pty<' and '>pty>').

- false prompts

  Interactive programs generally do not guarantee that output from user commands won't contain a prompt string. For example, your shell prompt might be a '$', and a file named '$' might be the only file in a directory listing.

  This can make it hard to guarantee that your output parser won't be fooled into early termination of results.

  To help work around this, you can see if the program can alter it's prompt, and use something you feel is never going to occur in actual practice.

  You should also look for your prompt to be the only thing on a line:

  ```
  pump $h until $out =~ /^<SILLYPROMPT>\s?\z/m;
  ```

  (use `(?!\n)\Z` in place of `\z` on older perls).

  You can also take the approach that IPC::ChildSafe takes and emit a command with known output after each 'real' command you issue, then look for this known output. See **new_appender()** and **new_chunker()** for filters that can help with this task.

  If it's not convenient or possibly to alter a prompt or use a known command/response pair, you might need to autodetect the prompt in case the local version of the child program is different then the one you tested with, or if the user has control over the look & feel of the prompt.

- Refusing to accept input unless stdin is a tty.

  Some programs, for security reasons, will only accept certain types of input from a tty. su, notable, will not prompt for a password unless it's connected to a tty.

  If this is your situation, use a pseudo terminal ('<pty<' and '>pty>').

- Not prompting unless connected to a tty.

  Some programs don't prompt unless stdin or stdout is a tty. See if you can turn prompting back on. If not, see if you can come up with a command that you can issue after every real command and look for it's output, as IPC::ChildSafe does. There are two filters included with IPC::Run that can help with doing this: appender and chunker (see **new_appender()** and **new_chunker()**).

- Different output format when not connected to a tty.

  Some commands alter their formats to ease machine parsability when they aren't connected to a pipe. This is actually good, but can be surprising.

PSEUDO TERMINALS

On systems providing pseudo terminals under /dev, IPC::Run can use IO::Pty (available on CPAN) to provide a terminal environment to subprocesses. This is necessary when the subprocess really wants to think it's connected to a real terminal.

CAVEATS

Pseudo-terminals are not pipes, though they are similar. Here are some differences to watch out for.

Echoing

Sending to stdin will cause an echo on stdout, which occurs before each line is passed to the child program. There is currently no way to disable this, although the child process can and should disable it for things like passwords.

Shutdown

IPC::Run cannot close a pty until all output has been collected. This means that it is not possible to send an EOF to stdin by half-closing the pty, as we can when using a pipe to stdin.

This means that you need to send the child process an exit command or signal, or **run() / finish()** will time out. Be careful not to expect a prompt after sending the exit command.

Command line editing
: Some subprocesses, notable shells that depend on the user's prompt settings, will reissue the prompt plus the command line input so far once for each character.

'>pty>' means '&>pty>', not '1>pty>'
: The pseudo terminal redirects both stdout and stderr unless you specify a file descriptor. If you want to grab stderr separately, do this:

```
start \@cmd, '<pty<', \$in, '>pty>', \$out, '2>', \$err;
```

stdin, stdout, and stderr not inherited
: Child processes harnessed to a pseudo terminal have their stdin, stdout, and stderr completely closed before any redirection operators take effect. This casts of the bonds of the controlling terminal. This is not done when using pipes.

  Right now, this affects all children in a harness that has a pty in use, even if that pty would not affect a particular child. That's a bug and will be fixed. Until it is, it's best not to mix-and-match children.

Redirection Operators

```
Operator        SHNP   Description
========        ====   ===========
<, N<           SHN    Redirects input to a child's fd N (0 assumed)

>, N>           SHN    Redirects output from a child's fd N (1 assumed)
>>, N>>         SHN    Like '>', but appends to scalars or named files
>&, &>          SHN    Redirects stdout & stderr from a child process

<pty, N<pty     S      Like '<', but uses a pseudo-tty instead of a pipe
>pty, N>pty     S      Like '>', but uses a pseudo-tty instead of a pipe

N<&M                   Dups input fd N to input fd M
M>&N                   Dups output fd N to input fd M
N<&-                   Closes fd N

<pipe, N<pipe     P    Pipe opens H for caller to read, write, close.
>pipe, N>pipe     P    Pipe opens H for caller to read, write, close.
```

'N' and 'M' are placeholders for integer file descriptor numbers. The terms 'input' and 'output' are from the child process's perspective.

The SHNP field indicates what parameters an operator can take:

```
S: \$scalar or \&function references.  Filters may be used with
   these operators (and only these).
H: \*HANDLE or IO::Handle for caller to open, and close
N: "file name".
P: \*HANDLE or lexical filehandle opened by IPC::Run as the parent end of a pipe,
   and written to and closed by the caller (like IPC::Open3).
```

Redirecting input: [n]<, [n]<pipe
: You can input the child reads on file descriptor number n to come from a scalar variable, subroutine, file handle, or a named file. If stdin is not redirected, the parent's stdin is inherited.

```
run \@cat, \undef          ## Closes child's stdin immediately
   or die "cat returned $?";

run \@cat, \$in;

run \@cat, \<<TOHERE;
blah
```

```
TOHERE

run \@cat, \&input;        ## Calls &input, feeding data returned
                            ## to child's.  Closes child's stdin
                            ## when undef is returned.
```

Redirecting from named files requires you to use the input redirection operator:

```
run \@cat, '<.profile';
run \@cat, '<', '.profile';

open IN, "<foo";
run \@cat, \*IN;
run \@cat, *IN{IO};
```

The form used second example here is the safest, since filenames like "0" and "&more\n" won't confuse &run:

You can't do either of

```
run \@a, *IN;       ## INVALID
run \@a, '<', *IN; ## BUGGY: Reads file named like "*main::A"
```

because perl passes a scalar containing a string that looks like "*main::A" to &run, and &run can't tell the difference between that and a redirection operator or a file name. &run guarantees that any scalar you pass after a redirection operator is a file name.

If your child process will take input from file descriptors other than 0 (stdin), you can use a redirection operator with any of the valid input forms (scalar ref, sub ref, etc.):

```
run \@cat, '3<', \$in3;
```

When redirecting input from a scalar ref, the scalar ref is used as a queue. This allows you to use &harness and **pump()** to feed incremental bits of input to a coprocess. See "Coprocesses" below for more information.

The <pipe operator opens the write half of a pipe on the filehandle glob reference it takes as an argument:

```
$h = start \@cat, '<pipe', \*IN;
print IN "hello world\n";
pump $h;
close IN;
finish $h;
```

Unlike the other '<' operators, IPC::Run does nothing further with it: you are responsible for it. The previous example is functionally equivalent to:

```
pipe( \*R, \*IN ) or die $!;
$h = start \@cat, '<', \*IN;
print IN "hello world\n";
pump $h;
close IN;
finish $h;
```

This is like the behavior of IPC::Open2 and IPC::Open3.

Win32: The handle returned is actually a socket handle, so you can use **select()** on it.

Redirecting output: [n]>, [n]>>, [n]>&[m], [n]>pipe

You can redirect any output the child emits to a scalar variable, subroutine, file handle, or file name. You can have &run truncate or append to named files or scalars. If you are redirecting stdin as well, or if the command is on the receiving end of a pipeline ('|'), you can omit the redirection operator:

```
@ls = ( 'ls' );
run \@ls, \undef, \$out
   or die "ls returned $?";

run \@ls, \undef, \&out;  ## Calls &out each time some output
                           ## is received from the child's
                           ## when undef is returned.

run \@ls, \undef, '2>ls.err';
run \@ls, '2>', 'ls.err';
```

The two parameter form guarantees that the filename will not be interpreted as a redirection operator:

```
run \@ls, '>', "&more";
run \@ls, '2>', ">foo\n";
```

You can pass file handles you've opened for writing:

```
open( *OUT, ">out.txt" );
open( *ERR, ">err.txt" );
run \@cat, \*OUT, \*ERR;
```

Passing a scalar reference and a code reference requires a little more work, but allows you to capture all of the output in a scalar or each piece of output by a callback:

These two do the same things:

```
run( [ 'ls' ], '2>', sub { $err_out .= $_[0] } );
```

does the same basic thing as:

```
run( [ 'ls' ], '2>', \$err_out );
```

The subroutine will be called each time some data is read from the child.

The >pipe operator is different in concept than the other '>' operators, although it's syntax is similar:

```
$h = start \@cat, $in, '>pipe', \*OUT, '2>pipe', \*ERR;
$in = "hello world\n";
finish $h;
print <OUT>;
print <ERR>;
close OUT;
close ERR;
```

causes two pipe to be created, with one end attached to cat's stdout and stderr, respectively, and the other left open on OUT and ERR, so that the script can manually **read()**, **select()**, etc. on them. This is like the behavior of IPC::Open2 and IPC::Open3.

Win32: The handle returned is actually a socket handle, so you can use **select()** on it.

Duplicating output descriptors: >&m, n>&m
: This duplicates output descriptor number n (default is 1 if n is omitted) from descriptor number m.

Duplicating input descriptors: <&m, n<&m
: This duplicates input descriptor number n (default is 0 if n is omitted) from descriptor number m

Closing descriptors: <&-, 3<&-
: This closes descriptor number n (default is 0 if n is omitted). The following commands are equivalent:

```
run \@cmd, \undef;
run \@cmd, '<&-';
run \@cmd, '<in.txt', '<&-';
```

Doing

```
run \@cmd, \$in, '<&-';     ## SIGPIPE recipe.
```

is dangerous: the parent will get a SIGPIPE if `$in` is not empty.

Redirecting both stdout and stderr: &>, >&, &>pipe, >pipe&

The following pairs of commands are equivalent:

```
run \@cmd, '>&', \$out;        run \@cmd, '>', \$out,     '2>&1';
run \@cmd, '>&', 'out.txt';    run \@cmd, '>', 'out.txt', '2>&1';
```

etc.

File descriptor numbers are not permitted to the left or the right of these operators, and the '&' may occur on either end of the operator.

The '&>pipe' and '>pipe&' variants behave like the '>pipe' operator, except that both stdout and stderr write to the created pipe.

Redirection Filters

Both input redirections and output redirections that use scalars or subs as endpoints may have an arbitrary number of filter subs placed between them and the child process. This is useful if you want to receive output in chunks, or if you want to massage each chunk of data sent to the child. To use this feature, you must use operator syntax:

```
run(
   \@cmd
      '<', \&in_filter_2, \&in_filter_1, $in,
      '>', \&out_filter_1, \&in_filter_2, $out,
);
```

This capability is not provided for IO handles or named files.

Two filters are provided by IPC::Run: appender and chunker. Because these may take an argument, you need to use the constructor functions **new_appender()** and **new_chunker()** rather than using \& syntax:

```
run(
   \@cmd
      '<', new_appender( "\n" ), $in,
      '>', new_chunker, $out,
);
```

Just doing I/O

If you just want to do I/O to a handle or file you open yourself, you may specify a filehandle or filename instead of a command in the harness specification:

```
run io( "filename", '>', \$recv );

$h = start io( $io, '>', \$recv );

$h = harness \@cmd, '&', io( "file", '<', \$send );
```

Options

Options are passed in as name/value pairs:

```
run \@cat, \$in, debug => 1;
```

If you pass the debug option, you may want to pass it in first, so you can see what parsing is going on:

```
run debug => 1, \@cat, \$in;
```

debug

Enables debugging output in parent and child. Debugging info is emitted to the STDERR that was present when IPC::Run was first `use()`ed (it's `dup()`ed out of the way so that it can be redirected in children without having debugging output emitted on it).

RETURN VALUES

harness() and **start()** return a reference to an IPC::Run harness. This is blessed in to the IPC::Run package, so you may make later calls to functions as members if you like:

```
$h = harness( ... );
$h->start;
$h->pump;
$h->finish;

$h = start( .... );
$h->pump;
...
```

Of course, using method call syntax lets you deal with any IPC::Run subclasses that might crop up, but don't hold your breath waiting for any.

run() and **finish()** return TRUE when all subcommands exit with a 0 result code. **This is the opposite of perl's system() command**.

All routines raise exceptions (via **die()**) when error conditions are recognized. A non-zero command result is not treated as an error condition, since some commands are tests whose results are reported in their exit codes.

ROUTINES

run Run takes a harness or harness specification and runs it, pumping all input to the child(ren), closing the input pipes when no more input is available, collecting all output that arrives, until the pipes delivering output are closed, then waiting for the children to exit and reaping their result codes.

You may think of `run( ... )` as being like

```
start( ... )->finish();
```

, though there is one subtle difference: **run()** does not set \$input_scalars to '' like **finish()** does. If an exception is thrown from **run()**, all children will be killed off ''gently'', and then ''annihilated'' if they do not go gently (in to that dark night. sorry).

If any exceptions are thrown, this does a ''kill_kill'' before propagating them.

signal

```
## To send it a specific signal by name ("USR1"):
signal $h, "USR1";
$h->signal ( "USR1" );
```

If `$signal` is provided and defined, sends a signal to all child processes. Try not to send numeric signals, use `"KILL"` instead of `9`, for instance. Numeric signals aren't portable.

Throws an exception if `$signal` is undef.

This will *not* clean up the harness, `finish` it if you kill it.

Normally TERM kills a process gracefully (this is what the command line utility `kill` does by default), INT is sent by one of the keys `^C`, `Backspace` or `<Del>`, and `QUIT` is used to kill a process and make it coredump.

The `HUP` signal is often used to get a process to ''restart'', rereading config files, and `USR1` and `USR2` for really application-specific things.

Often, running `kill -l` (that's a lower case ''L'') on the command line will list the signals

present on your operating system.

WARNING: The signal subsystem is not at all portable. We *may* offer to simulate TERM and KILL on some operating systems, submit code to me if you want this.

WARNING 2: Up to and including perl v5.6.1, doing almost anything in a signal handler could be dangerous. The most safe code avoids all mallocs and system calls, usually by preallocating a flag before entering the signal handler, altering the flag's value in the handler, and responding to the changed value in the main system:

```
my $got_usr1 = 0;
sub usr1_handler { ++$got_signal }

$SIG{USR1} = \&usr1_handler;
while () { sleep 1; print "GOT IT" while $got_usr1--; }
```

Even this approach is perilous if ++ and — aren't atomic on your system (I've never heard of this on any modern CPU large enough to run perl).

kill_kill

```
## To kill off a process:
$h->kill_kill;
kill_kill $h;

## To specify the grace period other than 30 seconds:
kill_kill $h, grace => 5;

## To send QUIT instead of KILL if a process refuses to die:
kill_kill $h, coup_d_grace => "QUIT";
```

Sends a TERM, waits for all children to exit for up to 30 seconds, then sends a KILL to any that survived the TERM.

Will wait for up to 30 more seconds for the OS to successfully KILL the processes.

The 30 seconds may be overridden by setting the grace option, this overrides both timers.

The harness is then cleaned up.

The doubled name indicates that this function may kill again and avoids colliding with the core Perl kill function.

Returns a 1 if the TERM was sufficient, or a 0 if KILL was required. Throws an exception if KILL did not permit the children to be reaped.

NOTE: The grace period is actually up to 1 second longer than that given. This is because the granularity of time is 1 second. Let me know if you need finer granularity, we can leverage Time::HiRes here.

Win32: Win32 does not know how to send real signals, so TERM is a full-force kill on Win32. Thus all talk of grace periods, etc. do not apply to Win32.

harness

Takes a harness specification and returns a harness. This harness is blessed in to IPC::Run, allowing you to use method call syntax for **run()**, **start()**, et al if you like.

harness() is provided so that you can pre-build harnesses if you would like to, but it's not required..

You may proceed to **run()**, **start()** or **pump()** after calling **harness()** (**pump()** calls **start()** if need be). Alternatively, you may pass your harness specification to **run()** or **start()** and let them **harness()** for you. You can't pass harness specifications to **pump()**, though.

close_terminal

This is used as (or in) an init sub to cast off the bonds of a controlling terminal. It must precede all other redirection ops that affect STDIN, STDOUT, or STDERR to be guaranteed effective.

start

```
$h = start(
   \@cmd, \$in, \$out, ...,
   timeout( 30, name => "process timeout" ),
   $stall_timeout = timeout( 10, name => "stall timeout"   ),
);

$h = start \@cmd, '<', \$in, '|', \@cmd2, ...;
```

start() accepts a harness or harness specification and returns a harness after building all of the pipes and launching (via **fork()/exec()**, or, maybe someday, **spawn()**) all the child processes. It does not send or receive any data on the pipes, see **pump()** and **finish()** for that.

You may call **harness()** and then pass it's result to **start()** if you like, but you only need to if it helps you structure or tune your application. If you do call **harness()**, you may skip **start()** and proceed directly to pump.

start() also starts all timers in the harness. See IPC::Run::Timer for more information.

start() flushes STDOUT and STDERR to help you avoid duplicate output. It has no way of asking Perl to flush all your open filehandles, so you are going to need to flush any others you have open. Sorry.

Here's how if you don't want to alter the state of $| for your filehandle:

```
$ofh = select HANDLE; $of = $|; $| = 1; $| = $of; select $ofh;
```

If you don't mind leaving output unbuffered on HANDLE, you can do the slightly shorter

```
$ofh = select HANDLE; $| = 1; select $ofh;
```

Or, you can use IO::Handle's **flush()** method:

```
use IO::Handle;
flush HANDLE;
```

Perl needs the equivalent of C's fflush((FILE *)NULL).

adopt

Experimental feature. NOT FUNCTIONAL YET, NEED TO CLOSE FDS BETTER IN CHILDREN. SEE t/adopt.t for a test suite.

pump

```
pump $h;
$h->pump;
```

Pump accepts a single parameter harness. It blocks until it delivers some input or receives some output. It returns TRUE if there is still input or output to be done, FALSE otherwise.

pump() will automatically call **start()** if need be, so you may call **harness()** then proceed to **pump()** if that helps you structure your application.

If **pump()** is called after all harnessed activities have completed, a ''process ended prematurely'' exception to be thrown. This allows for simple scripting of external applications without having to add lots of error handling code at each step of the script:

```
$h = harness \@smbclient, \$in, \$out, $err;

$in = "cd /foo\n";
$h->pump until $out =~ /^smb.*> \Z/m;
```

```
    die "error cding to /foo:\n$out" if $out =~ "ERR";
    $out = '';

    $in = "mget *\n";
    $h->pump until $out =~ /^smb.*> \Z/m;
    die "error retrieving files:\n$out" if $out =~ "ERR";

    $h->finish;

    warn $err if $err;
```

pump_nb

```
    pump_nb $h;
    $h->pump_nb;
```

"**pump()** non-blocking", pumps if anything's ready to be pumped, returns immediately otherwise. This is useful if you're doing some long-running task in the foreground, but don't want to starve any child processes.

pumpable

Returns TRUE if calling **pump()** won't throw an immediate "process ended prematurely" exception. This means that there are open I/O channels or active processes. May yield the parent processes' time slice for 0.01 second if all pipes are to the child and all are paused. In this case we can't tell if the child is dead, so we yield the processor and then attempt to reap the child in a nonblocking way.

reap_nb

Attempts to reap child processes, but does not block.

Does not currently take any parameters, one day it will allow specific children to be reaped.

Only call this from a signal handler if your `perl` is recent enough to have safe signal handling (5.6.1 did not, IIRC, but it was being discussed on perl5-porters). Calling this (or doing any significant work) in a signal handler on older `perl`s is asking for seg faults.

finish

This must be called after the last **start()** or **pump()** call for a harness, or your system will accumulate defunct processes and you may "leak" file descriptors.

finish() returns TRUE if all children returned 0 (and were not signaled and did not coredump, ie ! $?), and FALSE otherwise (this is like **run()**, and the opposite of **system()**).

Once a harness has been finished, it may be **run()** or **start()**ed again, including by **pump()**s auto-start.

If this throws an exception rather than a normal exit, the harness may be left in an unstable state, it's best to kill the harness to get rid of all the child processes, etc.

Specifically, if a timeout expires in **finish()**, **finish()** will not kill all the children. Call <$h-kill_kill>> in this case if you care. This differs from the behavior of "run".

result

```
    $h->result;
```

Returns the first non-zero result code (ie $? >> 8). See "full_result" to get the $? value for a child process.

To get the result of a particular child, do:

```
    $h->result( 0 );  # first child's $? >> 8
    $h->result( 1 );  # second child
```

or

```
($h->results)[0]
($h->results)[1]
```

Returns undef if no child processes were spawned and no child number was specified. Throws an exception if an out-of-range child number is passed.

results

Returns a list of child exit values. See ''full_results'' if you want to know if a signal killed the child.

Throws an exception if the harness is not in a finished state.

full_result

```
$h->full_result;
```

Returns the first non-zero $?. See ''result'' to get the first $? >> 8 value for a child process.

To get the result of a particular child, do:

```
$h->full_result( 0 );  # first child's $?
$h->full_result( 1 );  # second child
```

or

```
($h->full_results)[0]
($h->full_results)[1]
```

Returns undef if no child processes were spawned and no child number was specified. Throws an exception if an out-of-range child number is passed.

full_results

Returns a list of child exit values as returned by `wait`. See ''results'' if you don't care about coredumps or signals.

Throws an exception if the harness is not in a finished state.

FILTERS

These filters are used to modify input our output between a child process and a scalar or subroutine endpoint.

binary

```
run \@cmd, ">", binary, \$out;
run \@cmd, ">", binary, \$out;  ## Any TRUE value to enable
run \@cmd, ">", binary 0, \$out;  ## Any FALSE value to disable
```

This is a constructor for a ''binmode'' ''filter'' that tells IPC::Run to keep the carriage returns that would ordinarily be edited out for you (binmode is usually off). This is not a real filter, but an option masquerading as a filter.

It's not named ''binmode'' because you're likely to want to call Perl's binmode in programs that are piping binary data around.

new_chunker

This breaks a stream of data in to chunks, based on an optional scalar or regular expression parameter. The default is the Perl input record separator in $/, which is a newline be default.

```
run \@cmd, '>', new_chunker, \&lines_handler;
run \@cmd, '>', new_chunker( "\r\n" ), \&lines_handler;
```

Because this uses $/ by default, you should always pass in a parameter if you are worried about other code (modules, etc) modifying $/.

If this filter is last in a filter chain that dumps in to a scalar, the scalar must be set to '' before a new chunk will be written to it.

As an example of how a filter like this can be written, here's a chunker that splits on newlines:

```
sub line_splitter {
   my ( $in_ref, $out_ref ) = @_;

   return 0 if length $$out_ref;

   return input_avail && do {
      while (1) {
         if ( $$in_ref =~ s/\A(.*?\n)// ) {
            $$out_ref .= $1;
            return 1;
         }
         my $hmm = get_more_input;
         unless ( defined $hmm ) {
            $$out_ref = $$in_ref;
            $$in_ref = '';
            return length $$out_ref ? 1 : 0;
         }
         return 0 if $hmm eq 0;
      }
   }
};
```

new_appender
: This appends a fixed string to each chunk of data read from the source scalar or sub. This might be useful if you're writing commands to a child process that always must end in a fixed string, like "\n":

```
run( \@cmd,
   '<', new_appender( "\n" ), \&commands,
);
```

Here's a typical filter sub that might be created by **new_appender()**:

```
sub newline_appender {
   my ( $in_ref, $out_ref ) = @_;

   return input_avail && do {
      $$out_ref = join( '', $$out_ref, $$in_ref, "\n" );
      $$in_ref = '';
      1;
   }
};
```

new_string_source
: TODO: Needs confirmation. Was previously undocumented. in this module.

This is a filter which is exportable. Returns a sub which appends the data passed in to the output buffer and returns 1 if data was appended. 0 if it was an empty string and undef if no data was passed.

NOTE: Any additional variables passed to new_string_source will be passed to the sub every time it's called and appended to the output.

new_string_sink
: TODO: Needs confirmation. Was previously undocumented.

This is a filter which is exportable. Returns a sub which pops the data out of the input stream and pushes it onto the string.

io
: Takes a filename or filehandle, a redirection operator, optional filters, and a source or destination (depends on the redirection operator). Returns an IPC::Run::IO object suitable for **harness()**ing (including via **start()** or **run()**).

This is shorthand for

```
require IPC::Run::IO;

   ... IPC::Run::IO->new(...) ...
```

timer

```
$h = start( \@cmd, \$in, \$out, $t = timer( 5 ) );

pump $h until $out =~ /expected stuff/ || $t->is_expired;
```

Instantiates a non-fatal timer. **pump()** returns once each time a timer expires. Has no direct effect on **run()**, but you can pass a subroutine to fire when the timer expires.

See "timeout" for building timers that throw exceptions on expiration.

See "timer" in IPC::Run::Timer for details.

timeout

```
$h = start( \@cmd, \$in, \$out, $t = timeout( 5 ) );

pump $h until $out =~ /expected stuff/;
```

Instantiates a timer that throws an exception when it expires. If you don't provide an exception, a default exception that matches /^IPC::Run: .*timed out/ is thrown by default. You can pass in your own exception scalar or reference:

```
$h = start(
   \@cmd, \$in, \$out,
   $t = timeout( 5, exception => 'slowpoke' ),
);
```

or set the name used in debugging message and in the default exception string:

```
$h = start(
   \@cmd, \$in, \$out,
   timeout( 50, name => 'process timer' ),
   $stall_timer = timeout( 5, name => 'stall timer' ),
);

pump $h until $out =~ /started/;

$in = 'command 1';
$stall_timer->start;
pump $h until $out =~ /command 1 finished/;

$in = 'command 2';
$stall_timer->start;
pump $h until $out =~ /command 2 finished/;

$in = 'very slow command 3';
$stall_timer->start( 10 );
pump $h until $out =~ /command 3 finished/;

$stall_timer->start( 5 );
$in = 'command 4';
pump $h until $out =~ /command 4 finished/;

$stall_timer->reset; # Prevent restarting or expirng
finish $h;
```

See ''timer'' for building non-fatal timers.

See ''timer'' in IPC::Run::Timer for details.

FILTER IMPLEMENTATION FUNCTIONS

These functions are for use from within filters.

input_avail

Returns TRUE if input is available. If none is available, then &get_more_input is called and its result is returned.

This is usually used in preference to &get_more_input so that the calling filter removes all data from the `$in_ref` before more data gets read in to `$in_ref`.

`input_avail` is usually used as part of a return expression:

```
return input_avail && do {
   ## process the input just gotten
   1;
};
```

This technique allows input_avail to return the undef or 0 that a filter normally returns when there's no input to process. If a filter stores intermediate values, however, it will need to react to an undef:

```
my $got = input_avail;
if ( ! defined $got ) {
   ## No more input ever, flush internal buffers to $out_ref
}
return $got unless $got;
## Got some input, move as much as need be
return 1 if $added_to_out_ref;
```

get_more_input

This is used to fetch more input in to the input variable. It returns undef if there will never be any more input, 0 if there is none now, but there might be in the future, and TRUE if more input was gotten.

`get_more_input` is usually used as part of a return expression, see ''input_avail'' for more information.

TODO

These will be addressed as needed and as time allows.

Stall timeout.

Expose a list of child process objects. When I do this, each child process is likely to be blessed into IPC::Run::Proc.

`$kid->`**abort()**, `$kid->`**kill()**, `$kid->`signal(`$num_or_name`).

Write tests for /(full_)?results?/ subs.

Currently, **pump()** and **run()** only work on systems where **select()** works on the filehandles returned by **pipe()**. This does *not* include ActiveState on Win32, although it does work on cygwin under Win32 (thought the tests whine a bit). I'd like to rectify that, suggestions and patches welcome.

Likewise **start()** only fully works on **fork()**/**exec()** machines (well, just **fork()** if you only ever pass perl subs as subprocesses). There's some scaffolding for calling **Open3::spawn_with_handles()**, but that's untested, and not that useful with limited **select()**.

Support for `\@sub_cmd` as an argument to a command which gets replaced with /dev/fd or the name of a temporary file containing foo's output. This is like <(sub_cmd ...) found in bash and csh (IIRC).

Allow multiple harnesses to be combined as independent sets of processes in to one 'meta-harness'.

Allow a harness to be passed in place of an \@cmd. This would allow multiple harnesses to be aggregated.

Ability to add external file descriptors w/ filter chains and endpoints.

Ability to add timeouts and timing generators (i.e. repeating timeouts).

High resolution timeouts.

Win32 LIMITATIONS

Fails on Win9X

If you want Win9X support, you'll have to debug it or fund me because I don't use that system any more. The Win32 subsysem has been extended to use temporary files in simple **run()** invocations and these may actually work on Win9X too, but I don't have time to work on it.

May deadlock on Win2K (but not WinNT4 or WinXPPro)

Spawning more than one subprocess on Win2K causes a deadlock I haven't figured out yet, but simple uses of **run()** often work. Passes all tests on WinXPPro and WinNT.

no support yet for <pty< and >pty>

These are likely to be implemented as "<" and ">" with binmode on, not sure.

no support for file descriptors higher than 2 (stderr)

Win32 only allows passing explicit fds 0, 1, and 2. If you really, really need to pass file handles, us Win32API:: **GetOsFHandle()** or ::**FdGetOsFHandle()** to get the integer handle and pass it to the child process using the command line, environment, stdin, intermediary file, or other IPC mechanism. Then use that handle in the child (Win32API.pm provides ways to reconstitute Perl file handles from Win32 file handles).

no support for subroutine subprocesses (CODE refs)

Can't **fork()**, so the subroutines would have no context, and closures certainly have no meaning

Perhaps with Win32 **fork()** emulation, this can be supported in a limited fashion, but there are other very serious problems with that: all parent fds get **dup()**ed in to the thread emulating the forked process, and that keeps the parent from being able to close all of the appropriate fds.

no support for init => sub {} routines.

Win32 processes are created from scratch, there is no way to do an init routine that will affect the running child. Some limited support might be implemented one day, do **chdir()** and `%ENV` changes can be made.

signals

Win32 does not fully support signals. **signal()** is likely to cause errors unless sending a signal that Perl emulates, and `kill_kill()` is immediately fatal (there is no grace period).

helper processes

IPC::Run uses helper processes, one per redirected file, to adapt between the anonymous pipe connected to the child and the TCP socket connected to the parent. This is a waste of resources and will change in the future to either use threads (instead of helper processes) or a WaitForMultipleObjects call (instead of select). Please contact me if you can help with the **WaitForMultipleObjects()** approach; I haven't figured out how to get at it without C code.

shutdown pause

There seems to be a pause of up to 1 second between when a child program exits and the corresponding sockets indicate that they are closed in the parent. Not sure why.

binmode

binmode is not supported yet. The underpinnings are implemented, just ask if you need it.

IPC::Run::IO

IPC::Run::IO objects can be used on Unix to read or write arbitrary files. On Win32, they will need to use the same helper processes to adapt from non-**select()**able filehandles to **select()**able ones (or perhaps **WaitForMultipleObjects()** will work with them, not sure).

startup race conditions

There seems to be an occasional race condition between child process startup and pipe closings. It seems like if the child is not fully created by the time CreateProcess returns and we close the TCP socket being handed to it, the parent socket can also get closed. This is seen with the Win32 pumper applications, not the "real" child process being spawned.

I assume this is because the kernel hasn't gotten around to incrementing the reference count on the child's end (since the child was slow in starting), so the parent's closing of the child end causes the socket to be closed, thus closing the parent socket.

Being a race condition, it's hard to reproduce, but I encountered it while testing this code on a drive share to a samba box. In this case, it takes t/run.t a long time to spawn it's child processes (the parent hangs in the first select for several seconds until the child emits any debugging output).

I have not seen it on local drives, and can't reproduce it at will, unfortunately. The symptom is a "bad file descriptor in **select()**" error, and, by turning on debugging, it's possible to see that **select()** is being called on a no longer open file descriptor that was returned from the **_socket()** routine in Win32Helper. There's a new **confess()** that checks for this ("PARENT_HANDLE no longer open"), but I haven't been able to reproduce it (typically).

LIMITATIONS

On Unix, requires a system that supports `waitpid( $pid, WNOHANG )` so it can tell if a child process is still running.

PTYs don't seem to be non-blocking on some versions of Solaris. Here's a test script contributed by Borislav Deianov <borislav@ensim.com> to see if you have the problem. If it dies, you have the problem.

```
#!/usr/bin/perl

use IPC::Run qw(run);
use Fcntl;
use IO::Pty;

sub makecmd {
    return ['perl', '-e',
            '<STDIN>, print "\n" x '.$_[0].'; while(<STDIN>){last if /end/}'];
}

#pipe R, W;
#fcntl(W, F_SETFL, O_NONBLOCK);
#while (syswrite(W, "\n", 1)) { $pipebuf++ };
#print "pipe buffer size is $pipebuf\n";
my $pipebuf=4096;
my $in = "\n" x ($pipebuf * 2) . "end\n";
my $out;

$SIG{ALRM} = sub { die "Never completed!\n" };

print "reading from scalar via pipe...";
alarm( 2 );
run(makecmd($pipebuf * 2), '<', \$in, '>', \$out);
alarm( 0 );
print "done\n";

print "reading from code via pipe... ";
alarm( 2 );
run(makecmd($pipebuf * 3), '<', sub { $t = $in; undef $in; $t}, '>', \$out);
alarm( 0 );
```

```
print "done\n";

$pty = IO::Pty->new();
$pty->blocking(0);
$slave = $pty->slave();
while ($pty->syswrite("\n", 1)) { $ptybuf++ };
print "pty buffer size is $ptybuf\n";
$in = "\n" x ($ptybuf * 3) . "end\n";

print "reading via pty... ";
alarm( 2 );
run(makecmd($ptybuf * 3), '<pty<', \$in, '>', \$out);
alarm(0);
print "done\n";
```

No support for ';', '&&', '||', '{ ... }', etc: use perl's, since **run()** returns TRUE when the command exits with a 0 result code.

Does not provide shell-like string interpolation.

No support for `cd`, `setenv`, or `export`: do these in an **init()** sub

```
run(
   \cmd,
      ...
      init => sub {
         chdir $dir or die $!;
         $ENV{FOO}='BAR'
      }
);
```

Timeout calculation does not allow absolute times, or specification of days, months, etc.

WARNING: Function coprocesses (`run \&foo, ...`) suffer from two limitations. The first is that it is difficult to close all filehandles the child inherits from the parent, since there is no way to scan all open FILEHANDLEs in Perl and it both painful and a bit dangerous to close all open file descriptors with `POSIX::close()`. Painful because we can't tell which fds are open at the POSIX level, either, so we'd have to scan all possible fds and close any that we don't want open (normally `exec()` closes any non-inheritable but we don't `exec()` for &sub processes.

The second problem is that Perl's DESTROY subs and other on-exit cleanup gets run in the child process. If objects are instantiated in the parent before the child is forked, the DESTROY will get run once in the parent and once in the child. When coprocess subs exit, POSIX::_exit is called to work around this, but it means that objects that are still referred to at that time are not cleaned up. So setting package vars or closure vars to point to objects that rely on DESTROY to affect things outside the process (files, etc), will lead to bugs.

I goofed on the syntax: "<pipe" vs. "<pty<" and ">filename" are both oddities.

TODO

Allow one harness to "adopt" another:

```
$new_h = harness \@cmd2;
$h->adopt( $new_h );
```

Close all filehandles not explicitly marked to stay open.

The problem with this one is that there's no good way to scan all open FILEHANDLEs in Perl, yet you don't want child processes inheriting handles willy-nilly.

INSPIRATION

Well, **select()** and **waitpid()** badly needed wrapping, and **open3()** isn't open-minded enough for me.

The shell-like API inspired by a message Russ Allbery sent to perl5-porters, which included:

```
I've thought for some time that it would be
nice to have a module that could handle full Bourne shell pipe syntax
internally, with fork and exec, without ever invoking a shell.  Something
that you could give things like:

pipeopen (PIPE, [ qw/cat file/ ], '|', [ 'analyze', @args ], '>&3');
```

Message ylln51p2b6.fsf@windlord.stanford.edu, on 2000/02/04.

SUPPORT

Bugs should always be submitted via the GitHub bug tracker

<https://github.com/toddr/IPC-Run/issues>

AUTHORS

Adam Kennedy <adamk@cpan.org>

Barrie Slaymaker <barries@slaysys.com>

COPYRIGHT