NAME

IPC::Open2, open2 - open a process for both reading and writing

SYNOPSIS

```
use IPC::Open2;

$pid = open2(\*CHLD_OUT, \*CHLD_IN, 'some cmd and args');
  # or without using the shell
$pid = open2(\*CHLD_OUT, \*CHLD_IN, 'some', 'cmd', 'and', 'args');

# or with handle autovivification
my($chld_out, $chld_in);
$pid = open2($chld_out, $chld_in, 'some cmd and args');
  # or without using the shell
$pid = open2($chld_out, $chld_in, 'some', 'cmd', 'and', 'args');

waitpid( $pid, 0 );
my $child_exit_status = $? >> 8;
```

DESCRIPTION

The open2() function runs the given $cmd and connects $chld_out for reading and $chld_in for writing. It's what you think should work when you try

```
$pid = open(HANDLE, "|cmd args|");
```

The write filehandle will have autoflush turned on.

If $chld_out is a string (that is, a bareword filehandle rather than a glob or a reference) and it begins with `>&`, then the child will send output directly to that file handle. If $chld_in is a string that begins with `<&`, then $chld_in will be closed in the parent, and the child will read from it directly. In both cases, there will be a dup(2) instead of a pipe(2) made.

If either reader or writer is the null string, this will be replaced by an autogenerated filehandle. If so, you must pass a valid lvalue in the parameter slot so it can be overwritten in the caller, or an exception will be raised.

open2() returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open2:/`. However, `exec` failures in the child are not detected. You'll have to trap SIGPIPE yourself.

open2() does not wait for and reap the child process after it exits. Except for short programs where it's acceptable to let the operating system take care of this, you need to do this yourself. This is normally as simple as calling `waitpid $pid, 0` when you're done with the process. Failing to do this can result in an accumulation of defunct or "zombie" processes. See *"waitpid" in perlfunc* for more information.

This whole affair is quite dangerous, as you may block forever. It assumes it's going to talk to something like **bc**, both writing to it and reading from it. This is presumably safe because you "know" that commands like **bc** will read a line at a time and output a line at a time. Programs like **sort** that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

The IO::Pty and Expect modules from CPAN can help with this, as they provide a real tty (well, a

pseudo-tty, actually), which gets you back to line buffering in the invoked command again.

WARNING

The order of arguments differs from that of open3().

SEE ALSO

See *IPC::Open3* for an alternative that handles STDERR as well. This function is really just a wrapper around open3().