NAME

perlfaq8 - System Interaction

VERSION

version 5.021011

DESCRIPTION

This section of the Perl FAQ covers questions involving operating system interaction. Topics include interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices), and most anything else not related to data manipulation.

Read the FAQs and documentation specific to the port of perl to your operating system (eg, *perlvms*, *perlplan9*, ...). These should contain more detailed information on the vagaries of your perl.

How do I find out which operating system I'm running under?

The `$^O` variable (`$OSNAME` if you use `English`) contains an indication of the name of the operating system (not its release number) that your perl binary was built for.

How come exec() doesn't return?

(contributed by brian d foy)

The `exec` function's job is to turn your process into another command and never to return. If that's not what you want to do, don't use `exec`. :)

If you want to run an external command and still keep your Perl process going, look at a piped `open`, `fork`, or `system`.

How do I do fancy stuff with the keyboard/screen/mouse?

How you access/control keyboards, screens, and pointing devices ("mice") is system-dependent. Try the following modules:

Keyboard

```
Term::Cap               Standard perl distribution
Term::ReadKey           CPAN
Term::ReadLine::Gnu     CPAN
Term::ReadLine::Perl    CPAN
Term::Screen            CPAN
```

Screen

```
Term::Cap               Standard perl distribution
Curses                  CPAN
Term::ANSIColor         CPAN
```

Mouse

```
Tk                      CPAN
Wx                      CPAN
Gtk2                    CPAN
Qt4                     kdebindings4 package
```

Some of these specific cases are shown as examples in other answers in this section of the perlfaq.

How do I print something out in color?

In general, you don't, because you don't know whether the recipient has a color-aware display device. If you know that they have an ANSI terminal that understands color, you can use the *Term::ANSIColor* module from CPAN:

```
use Term::ANSIColor;
print color("red"), "Stop!\n", color("reset");
print color("green"), "Go!\n", color("reset");
```

Or like this:

```
use Term::ANSIColor qw(:constants);
print RED, "Stop!\n", RESET;
print GREEN, "Go!\n", RESET;
```

How do I read just one key without waiting for a return key?

Controlling input buffering is a remarkably system-dependent matter. On many systems, you can just use the **stty** command as shown in *"getc" in perlfunc*, but as you see, that's already getting you into portability snags.

```
open(TTY, "+</dev/tty") or die "no tty: $!";
system "stty  cbreak </dev/tty >/dev/tty 2>&1";
$key = getc(TTY);        # perhaps this works
# OR ELSE
sysread(TTY, $key, 1);    # probably this does
system "stty -cbreak </dev/tty >/dev/tty 2>&1";
```

The *Term::ReadKey* module from CPAN offers an easy-to-use interface that should be more efficient than shelling out to **stty** for each key. It even includes limited support for Windows.

```
use Term::ReadKey;
ReadMode('cbreak');
$key = ReadKey(0);
ReadMode('normal');
```

However, using the code requires that you have a working C compiler and can use it to build and install a CPAN module. Here's a solution using the standard *POSIX* module, which is already on your system (assuming your system supports POSIX).

```
use HotKey;
$key = readkey();
```

And here's the `HotKey` module, which hides the somewhat mystifying calls to manipulate the POSIX termios structures.

```
# HotKey.pm
package HotKey;

use strict;
use warnings;

use parent 'Exporter';
our @EXPORT = qw(cbreak cooked readkey);

use POSIX qw(:termios_h);
my ($term, $oterm, $echo, $noecho, $fd_stdin);

$fd_stdin = fileno(STDIN);
$term     = POSIX::Termios->new();
```

```
$term->getattr($fd_stdin);
$oterm     = $term->getlflag();

$echo     = ECHO | ECHOK | ICANON;
$noecho   = $oterm & ~$echo;

sub cbreak {
    $term->setlflag($noecho);  # ok, so i don't want echo either
    $term->setcc(VTIME, 1);
    $term->setattr($fd_stdin, TCSANOW);
}

sub cooked {
    $term->setlflag($oterm);
    $term->setcc(VTIME, 0);
    $term->setattr($fd_stdin, TCSANOW);
}

sub readkey {
    my $key = '';
    cbreak();
    sysread(STDIN, $key, 1);
    cooked();
    return $key;
}

END { cooked() }

1;
```

How do I check whether input is ready on the keyboard?

The easiest way to do this is to read a key in nonblocking mode with the *Term::ReadKey* module from CPAN, passing it an argument of -1 to indicate not to block:

```
use Term::ReadKey;

ReadMode('cbreak');

if (defined (my $char = ReadKey(-1)) ) {
    # input was waiting and it was $char
} else {
    # no input was waiting
}

ReadMode('normal');                 # restore normal tty settings
```

How do I clear the screen?

(contributed by brian d foy)

To clear the screen, you just have to print the special sequence that tells the terminal to clear the screen. Once you have that sequence, output it when you want to clear the screen.

You can use the *Term::ANSIScreen* module to get the special sequence. Import the `cls` function (or

the `:screen` tag):

```
use Term::ANSIScreen qw(cls);
my $clear_screen = cls();

print $clear_screen;
```

The *Term::Cap* module can also get the special sequence if you want to deal with the low-level details of terminal control. The `Tputs` method returns the string for the given capability:

```
use Term::Cap;

my $terminal = Term::Cap->Tgetent( { OSPEED => 9600 } );
my $clear_screen = $terminal->Tputs('cl');

print $clear_screen;
```

On Windows, you can use the *Win32::Console* module. After creating an object for the output filehandle you want to affect, call the `Cls` method:

```
Win32::Console;

my $OUT = Win32::Console->new(STD_OUTPUT_HANDLE);
my $clear_string = $OUT->Cls;

print $clear_screen;
```

If you have a command-line program that does the job, you can call it in backticks to capture whatever it outputs so you can use it later:

```
my $clear_string = `clear`;

print $clear_string;
```

How do I get the screen size?

If you have *Term::ReadKey* module installed from CPAN, you can use it to fetch the width and height in characters and in pixels:

```
use Term::ReadKey;
my ($wchar, $hchar, $wpixels, $hpixels) = GetTerminalSize();
```

This is more portable than the raw `ioctl`, but not as illustrative:

```
require './sys/ioctl.ph';
die "no TIOCGWINSZ " unless defined &TIOCGWINSZ;
open(my $tty_fh, "+</dev/tty")                    or die "No tty: $!";
unless (ioctl($tty_fh, &TIOCGWINSZ, $winsize='')) {
    die sprintf "$0: ioctl TIOCGWINSZ (%08x: $!)\n", &TIOCGWINSZ;
}
my ($row, $col, $xpixel, $ypixel) = unpack('S4', $winsize);
print "(row,col) = ($row,$col)";
print "  (xpixel,ypixel) = ($xpixel,$ypixel)" if $xpixel || $ypixel;
print "\n";
```

How do I ask the user for a password?

(This question has nothing to do with the web. See a different FAQ for that.)

There's an example of this in *"crypt" in perlfunc*). First, you put the terminal into "no echo" mode, then just read the password normally. You may do this with an old-style `ioctl()` function, POSIX terminal control (see *POSIX* or its documentation the Camel Book), or a call to the **stty** program, with varying degrees of portability.

You can also do this for most systems using the *Term::ReadKey* module from CPAN, which is easier to use and in theory more portable.

```
use Term::ReadKey;

ReadMode('noecho');
my $password = ReadLine(0);
```

How do I read and write the serial port?

This depends on which operating system your program is running on. In the case of Unix, the serial ports will be accessible through files in `/dev`; on other systems, device names will doubtless differ. Several problem areas common to all device interaction are the following:

lockfiles

Your system may use lockfiles to control multiple access. Make sure you follow the correct protocol. Unpredictable behavior can result from multiple processes reading from one device.

open mode

If you expect to use both read and write operations on the device, you'll have to open it for update (see *"open" in perlfunc* for details). You may wish to open it without running the risk of blocking by using `sysopen()` and `O_RDWR|O_NDELAY|O_NOCTTY` from the *Fcntl* module (part of the standard perl distribution). See *"sysopen" in perlfunc* for more on this approach.

end of line

Some devices will be expecting a "\r" at the end of each line rather than a "\n". In some ports of perl, "\r" and "\n" are different from their usual (Unix) ASCII values of "\015" and "\012". You may have to give the numeric values you want directly, using octal ("\015"), hex ("0x0D"), or as a control-character specification ("\cM").

```
print DEV "atv1\012";    # wrong, for some devices
print DEV "atv1\015";    # right, for some devices
```

Even though with normal text files a "\n" will do the trick, there is still no unified scheme for terminating a line that is portable between Unix, DOS/Win, and Macintosh, except to terminate *ALL* line ends with "\015\012", and strip what you don't need from the output. This applies especially to socket I/O and autoflushing, discussed next.

flushing output

If you expect characters to get to your device when you `print()` them, you'll want to autoflush that filehandle. You can use `select()` and the `$|` variable to control autoflushing (see *"$|" in perlvar* and *"select" in perlfunc*, or *perlfaq5*, "How do I flush/unbuffer an output filehandle? Why must I do this?"):

```
my $old_handle = select($dev_fh);
$| = 1;
select($old_handle);
```

You'll also see code that does this without a temporary variable, as in

```
select((select($deb_handle), $| = 1)[0]);
```

Or if you don't mind pulling in a few thousand lines of code just because you're afraid of a little `$|` variable:

```
use IO::Handle;
$dev_fh->autoflush(1);
```

As mentioned in the previous item, this still doesn't work when using socket I/O between Unix and Macintosh. You'll need to hard code your line terminators, in that case.

non-blocking input

If you are doing a blocking `read()` or `sysread()`, you'll have to arrange for an alarm handler to provide a timeout (see *"alarm" in perlfunc*). If you have a non-blocking open, you'll likely have a non-blocking read, which means you may have to use a 4-arg `select()` to determine whether I/O is ready on that device (see *"select" in perlfunc*.

While trying to read from his caller-id box, the notorious Jamie Zawinski `<jwz@netscape.com>`, after much gnashing of teeth and fighting with `sysread`, `sysopen`, POSIX's `tcgetattr` business, and various other functions that go bump in the night, finally came up with this:

```
sub open_modem {
    use IPC::Open2;
    my $stty = `/bin/stty -g`;
    open2( \*MODEM_IN, \*MODEM_OUT, "cu -l$modem_device -s2400 2>&1");
    # starting cu hoses /dev/tty's stty settings, even when it has
    # been opened on a pipe...
    system("/bin/stty $stty");
    $_ = <MODEM_IN>;
    chomp;
    if ( !m/^Connected/ ) {
        print STDERR "$0: cu printed `$_' instead of `Connected'\n";
    }
}
```

How do I decode encrypted password files?

You spend lots and lots of money on dedicated hardware, but this is bound to get you talked about.

Seriously, you can't if they are Unix password files--the Unix password system employs one-way encryption. It's more like hashing than encryption. The best you can do is check whether something else hashes to the same string. You can't turn a hash back into the original string. Programs like Crack can forcibly (and intelligently) try to guess passwords, but don't (can't) guarantee quick success.

If you're worried about users selecting bad passwords, you should proactively check when they try to change their password (by modifying *passwd(1)*, for example).

How do I start a process in the background?

(contributed by brian d foy)

There's not a single way to run code in the background so you don't have to wait for it to finish before your program moves on to other tasks. Process management depends on your particular operating system, and many of the techniques are covered in *perlipc*.

Several CPAN modules may be able to help, including *IPC::Open2* or *IPC::Open3*, *IPC::Run*, *Parallel::Jobs*, *Parallel::ForkManager*, *POE*, *Proc::Background*, and *Win32::Process*. There are many other modules you might use, so check those namespaces for other options too.

If you are on a Unix-like system, you might be able to get away with a system call where you put an `&` on the end of the command:

```
system("cmd &")
```

You can also try using `fork`, as described in *perlfunc* (although this is the same thing that many of the modules will do for you).

STDIN, STDOUT, and STDERR are shared

Both the main process and the backgrounded one (the "child" process) share the same STDIN, STDOUT and STDERR filehandles. If both try to access them at once, strange things can happen. You may want to close or reopen these for the child. You can get around this with opening a pipe (see *"open" in perlfunc*) but on some systems this means that the child process cannot outlive the parent.

Signals

You'll have to catch the SIGCHLD signal, and possibly SIGPIPE too. SIGCHLD is sent when the backgrounded process finishes. SIGPIPE is sent when you write to a filehandle whose child process has closed (an untrapped SIGPIPE can cause your program to silently die). This is not an issue with `system("cmd&")`.

Zombies

You have to be prepared to "reap" the child process when it finishes.

```
$SIG{CHLD} = sub { wait };

$SIG{CHLD} = 'IGNORE';
```

You can also use a double fork. You immediately `wait()` for your first child, and the init daemon will `wait()` for your grandchild once it exits.

```
unless ($pid = fork) {
    unless (fork) {
        exec "what you really wanna do";
        die "exec failed!";
    }
    exit 0;
}
waitpid($pid, 0);
```

See *"Signals" in perlipc* for other examples of code to do this. Zombies are not an issue with `system("prog &")`.

How do I trap control characters/signals?

You don't actually "trap" a control character. Instead, that character generates a signal which is sent to your terminal's currently foregrounded process group, which you then trap in your process. Signals are documented in *"Signals" in perlipc* and the section on "Signals" in the Camel.

You can set the values of the `%SIG` hash to be the functions you want to handle the signal. After perl catches the signal, it looks in `%SIG` for a key with the same name as the signal, then calls the subroutine value for that key.

```
# as an anonymous subroutine

$SIG{INT} = sub { syswrite(STDERR, "ouch\n", 5 ) };

# or a reference to a function

$SIG{INT} = \&ouch;
```

```
    # or the name of the function as a string

    $SIG{INT} = "ouch";
```

Perl versions before 5.8 had in its C source code signal handlers which would catch the signal and possibly run a Perl function that you had set in %SIG. This violated the rules of signal handling at that level causing perl to dump core. Since version 5.8.0, perl looks at %SIG **after** the signal has been caught, rather than while it is being caught. Previous versions of this answer were incorrect.

How do I modify the shadow password file on a Unix system?

If perl was installed correctly and your shadow library was written properly, the getpw*() functions described in *perlfunc* should in theory provide (read-only) access to entries in the shadow password file. To change the file, make a new shadow password file (the format varies from system to system--see *passwd(1)* for specifics) and use pwd_mkdb(8) to install it (see *pwd_mkdb(8)* for more details).

How do I set the time and date?

Assuming you're running under sufficient permissions, you should be able to set the system-wide date and time by running the date(1) program. (There is no way to set the time and date on a per-process basis.) This mechanism will work for Unix, MS-DOS, Windows, and NT; the VMS equivalent is set time.

However, if all you want to do is change your time zone, you can probably get away with setting an environment variable:

```
$ENV{TZ} = "MST7MDT";           # Unixish
$ENV{'SYS$TIMEZONE_DIFFERENTIAL'}="-5" # vms
system('trn', 'comp.lang.perl.misc');
```

How can I sleep() or alarm() for under a second?

If you want finer granularity than the 1 second that the sleep() function provides, the easiest way is to use the select() function as documented in *"select" in perlfunc*. Try the *Time::HiRes* and the *BSD::Itimer* modules (available from CPAN, and starting from Perl 5.8 *Time::HiRes* is part of the standard distribution).

How can I measure time under a second?

(contributed by brian d foy)

The *Time::HiRes* module (part of the standard distribution as of Perl 5.8) measures time with the gettimeofday() system call, which returns the time in microseconds since the epoch. If you can't install *Time::HiRes* for older Perls and you are on a Unixish system, you may be able to call gettimeofday(2) directly. See *"syscall" in perlfunc*.

How can I do an atexit() or setjmp()/longjmp()? (Exception handling)

You can use the END block to simulate atexit(). Each package's END block is called when the program or thread ends. See the *perlmod* manpage for more details about END blocks.

For example, you can use this to make sure your filter program managed to finish its output without filling up the disk:

```
END {
    close(STDOUT) || die "stdout close failed: $!";
}
```

The END block isn't called when untrapped signals kill the program, though, so if you use END blocks you should also use

```
use sigtrap qw(die normal-signals);
```

Perl's exception-handling mechanism is its `eval()` operator. You can use `eval()` as `setjmp` and `die()` as `longjmp`. For details of this, see the section on signals, especially the time-out handler for a blocking `flock()` in *"Signals" in perlipc* or the section on "Signals" in *Programming Perl*.

If exception handling is all you're interested in, use one of the many CPAN modules that handle exceptions, such as *Try::Tiny*.

If you want the `atexit()` syntax (and an `rmexit()` as well), try the `AtExit` module available from CPAN.

Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?

Some Sys-V based systems, notably Solaris 2.X, redefined some of the standard socket constants. Since these were constant across all architectures, they were often hardwired into perl code. The proper way to deal with this is to "use Socket" to get the correct values.

Note that even though SunOS and Solaris are binary compatible, these values are different. Go figure.

How can I call my system's unique C functions from Perl?

In most cases, you write an external module to do it--see the answer to "Where can I learn about linking C with Perl? [h2xs, xsubpp]". However, if the function is a system call, and your system supports `syscall()`, you can use the `syscall` function (documented in *perlfunc*).

Remember to check the modules that came with your distribution, and CPAN as well--someone may already have written a module to do it. On Windows, try *Win32::API*. On Macs, try *Mac::Carbon*. If no module has an interface to the C function, you can inline a bit of C in your Perl source with *Inline::C*.

Where do I get the include files to do ioctl() or syscall()?

Historically, these would be generated by the *h2ph* tool, part of the standard perl distribution. This program converts `cpp(1)` directives in C header files to files containing subroutine definitions, like `SYS_getitimer()`, which you can use as arguments to your functions. It doesn't work perfectly, but it usually gets most of the job done. Simple files like *errno.h*, *syscall.h*, and *socket.h* were fine, but the hard ones like *ioctl.h* nearly always need to be hand-edited. Here's how to install the *.ph files:

```
1. Become the super-user
2. cd /usr/include
3. h2ph *.h */*.h
```

If your system supports dynamic loading, for reasons of portability and sanity you probably ought to use *h2xs* (also part of the standard perl distribution). This tool converts C header files to Perl extensions. See *perlxstut* for how to get started with *h2xs*.

If your system doesn't support dynamic loading, you still probably ought to use *h2xs*. See *perlxstut* and *ExtUtils::MakeMaker* for more information (in brief, just use **make perl** instead of a plain **make** to rebuild perl with a new static extension).

Why do setuid perl scripts complain about kernel problems?

Some operating systems have bugs in the kernel that make setuid scripts inherently insecure. Perl gives you a number of options (described in *perlsec*) to work around such systems.

How can I open a pipe both to and from a command?

The *IPC::Open2* module (part of the standard perl distribution) is an easy-to-use approach that internally uses `pipe()`, `fork()`, and `exec()` to do the job. Make sure you read the deadlock warnings in its documentation, though (see *IPC::Open2*). See *"Bidirectional Communication with*

Another Process" in perlipc and *"Bidirectional Communication with Yourself" in perlipc*

You may also use the *IPC::Open3* module (part of the standard perl distribution), but be warned that it has a different order of arguments from *IPC::Open2* (see *IPC::Open3*).

Why can't I get the output of a command with system()?

You're confusing the purpose of `system()` and backticks (``). `system()` runs a command and returns exit status information (as a 16 bit value: the low 7 bits are the signal the process died from, if any, and the high 8 bits are the actual exit value). Backticks (``) run a command and return what it sent to STDOUT.

```
my $exit_status   = system("mail-users");
my $output_string = `ls`;
```

How can I capture STDERR from an external command?

There are three basic ways of running external commands:

```
system $cmd;        # using system()
my $output = `$cmd`;        # using backticks (``)
open (my $pipe_fh, "$cmd |");    # using open()
```

With `system()`, both STDOUT and STDERR will go the same place as the script's STDOUT and STDERR, unless the `system()` command redirects them. Backticks and `open()` read **only** the STDOUT of your command.

You can also use the `open3()` function from *IPC::Open3*. Benjamin Goldberg provides some sample code:

To capture a program's STDOUT, but discard its STDERR:

```
use IPC::Open3;
use File::Spec;
my $in = '';
open(NULL, ">", File::Spec->devnull);
my $pid = open3($in, \*PH, ">&NULL", "cmd");
while( <PH> ) { }
waitpid($pid, 0);
```

To capture a program's STDERR, but discard its STDOUT:

```
use IPC::Open3;
use File::Spec;
my $in = '';
open(NULL, ">", File::Spec->devnull);
my $pid = open3($in, ">&NULL", \*PH, "cmd");
while( <PH> ) { }
waitpid($pid, 0);
```

To capture a program's STDERR, and let its STDOUT go to our own STDERR:

```
use IPC::Open3;
my $in = '';
my $pid = open3($in, ">&STDERR", \*PH, "cmd");
while( <PH> ) { }
waitpid($pid, 0);
```

To read both a command's STDOUT and its STDERR separately, you can redirect them to temp files,

let the command run, then read the temp files:

```
use IPC::Open3;
use IO::File;
my $in = '';
local *CATCHOUT = IO::File->new_tmpfile;
local *CATCHERR = IO::File->new_tmpfile;
my $pid = open3($in, ">&CATCHOUT", ">&CATCHERR", "cmd");
waitpid($pid, 0);
seek $_, 0, 0 for \*CATCHOUT, \*CATCHERR;
while( <CATCHOUT> ) {}
while( <CATCHERR> ) {}
```

But there's no real need for **both** to be tempfiles... the following should work just as well, without deadlocking:

```
use IPC::Open3;
my $in = '';
use IO::File;
local *CATCHERR = IO::File->new_tmpfile;
my $pid = open3($in, \*CATCHOUT, ">&CATCHERR", "cmd");
while( <CATCHOUT> ) {}
waitpid($pid, 0);
seek CATCHERR, 0, 0;
while( <CATCHERR> ) {}
```

And it'll be faster, too, since we can begin processing the program's stdout immediately, rather than waiting for the program to finish.

With any of these, you can change file descriptors before the call:

```
open(STDOUT, ">logfile");
system("ls");
```

or you can use Bourne shell file-descriptor redirection:

```
$output = `$cmd 2>some_file`;
open (PIPE, "cmd 2>some_file |");
```

You can also use file-descriptor redirection to make STDERR a duplicate of STDOUT:

```
$output = `$cmd 2>&1`;
open (PIPE, "cmd 2>&1 |");
```

Note that you *cannot* simply open STDERR to be a dup of STDOUT in your Perl program and avoid calling the shell to do the redirection. This doesn't work:

```
open(STDERR, ">&STDOUT");
$alloutput = `cmd args`;  # stderr still escapes
```

This fails because the `open()` makes STDERR go to where STDOUT was going at the time of the `open()`. The backticks then make STDOUT go to a string, but don't change STDERR (which still goes to the old STDOUT).

Note that you *must* use Bourne shell (`sh(1)`) redirection syntax in backticks, not `csh(1)`! Details on why Perl's `system()` and backtick and pipe opens all use the Bourne shell are in the *versus/csh.whynot* article in the "Far More Than You Ever Wanted To Know" collection in

http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz . To capture a command's STDERR and STDOUT together:

```
$output = `cmd 2>&1`;                       # either with backticks
$pid = open(PH, "cmd 2>&1 |");              # or with an open pipe
while (<PH>) { }                            #    plus a read
```

To capture a command's STDOUT but discard its STDERR:

```
$output = `cmd 2>/dev/null`;                # either with backticks
$pid = open(PH, "cmd 2>/dev/null |");       # or with an open pipe
while (<PH>) { }                            #    plus a read
```

To capture a command's STDERR but discard its STDOUT:

```
$output = `cmd 2>&1 1>/dev/null`;           # either with backticks
$pid = open(PH, "cmd 2>&1 1>/dev/null |");  # or with an open pipe
while (<PH>) { }                            #    plus a read
```

To exchange a command's STDOUT and STDERR in order to capture the STDERR but leave its STDOUT to come out our old STDERR:

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;        # either with backticks
$pid = open(PH, "cmd 3>&1 1>&2 2>&3 3>&-|");# or with an open pipe
while (<PH>) { }                            #    plus a read
```

To read both a command's STDOUT and its STDERR separately, it's easiest to redirect them separately to files, and then read from those files when the program is done:

```
system("program args 1>program.stdout 2>program.stderr");
```

Ordering is important in all these examples. That's because the shell processes file descriptor redirections in strictly left to right order.

```
system("prog args 1>tmpfile 2>&1");
system("prog args 2>&1 1>tmpfile");
```

The first command sends both standard out and standard error to the temporary file. The second command sends only the old standard output there, and the old standard error shows up on the old standard out.

Why doesn't open() return an error when a pipe open fails?

If the second argument to a piped `open()` contains shell metacharacters, perl `fork()`s, then `exec()`s a shell to decode the metacharacters and eventually run the desired program. If the program couldn't be run, it's the shell that gets the message, not Perl. All your Perl program can find out is whether the shell itself could be successfully started. You can still capture the shell's STDERR and check it for error messages. See *How can I capture STDERR from an external command?* elsewhere in this document, or use the *IPC::Open3* module.

If there are no shell metacharacters in the argument of `open()`, Perl runs the command directly, without using the shell, and can correctly report whether the command started.

What's wrong with using backticks in a void context?

Strictly speaking, nothing. Stylistically speaking, it's not a good way to write maintainable code. Perl has several operators for running external commands. Backticks are one; they collect the output from the command for use in your program. The `system` function is another; it doesn't do this.

Writing backticks in your program sends a clear message to the readers of your code that you wanted to collect the output of the command. Why send a clear message that isn't true?

Consider this line:

```
`cat /etc/termcap`;
```

You forgot to check $? to see whether the program even ran correctly. Even if you wrote

```
print `cat /etc/termcap`;
```

this code could and probably should be written as

```
system("cat /etc/termcap") == 0
or die "cat program failed!";
```

which will echo the cat command's output as it is generated, instead of waiting until the program has completed to print it out. It also checks the return value.

`system` also provides direct control over whether shell wildcard processing may take place, whereas backticks do not.

How can I call backticks without shell processing?

This is a bit tricky. You can't simply write the command like this:

```
@ok = `grep @opts '$search_string' @filenames`;
```

As of Perl 5.8.0, you can use `open()` with multiple arguments. Just like the list forms of `system()` and `exec()`, no shell escapes happen.

```
open( GREP, "-|", 'grep', @opts, $search_string, @filenames );
chomp(@ok = <GREP>);
close GREP;
```

You can also:

```
my @ok = ();
if (open(GREP, "-|")) {
    while (<GREP>) {
        chomp;
        push(@ok, $_);
    }
    close GREP;
} else {
    exec 'grep', @opts, $search_string, @filenames;
}
```

Just as with `system()`, no shell escapes happen when you `exec()` a list. Further examples of this can be found in *"Safe Pipe Opens" in perlipc*.

Note that if you're using Windows, no solution to this vexing issue is even possible. Even though Perl emulates `fork()`, you'll still be stuck, because Windows does not have an argc/argv-style API.

Why can't my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MS-DOS)?

This happens only if your perl is compiled to use stdio instead of perlio, which is the default. Some (maybe all?) stdios set error and eof flags that you may need to clear. The *POSIX* module defines `clearerr()` that you can use. That is the technically correct way to do it. Here are some less

reliable workarounds:

1 Try keeping around the seekpointer and go there, like this:

```
my $where = tell($log_fh);
seek($log_fh, $where, 0);
```

2 If that doesn't work, try seeking to a different part of the file and then back.

3 If that doesn't work, try seeking to a different part of the file, reading something, and then seeking back.

4 If that doesn't work, give up on your stdio package and use sysread.

How can I convert my shell script to perl?

Learn Perl and rewrite it. Seriously, there's no simple converter. Things that are awkward to do in the shell are easy to do in Perl, and this very awkwardness is what would make a shell->perl converter nigh-on impossible to write. By rewriting it, you'll think about what you're really trying to do, and hopefully will escape the shell's pipeline datastream paradigm, which while convenient for some matters, causes many inefficiencies.

Can I use perl to run a telnet or ftp session?

Try the *Net::FTP*, *TCP::Client*, and *Net::Telnet* modules (available from CPAN). *http://www.cpan.org/scripts/netstuff/telnet.emul.shar* will also help for emulating the telnet protocol, but *Net::Telnet* is quite probably easier to use.

If all you want to do is pretend to be telnet but don't need the initial telnet handshaking, then the standard dual-process approach will suffice:

```
use IO::Socket;             # new in 5.004
my $handle = IO::Socket::INET->new('www.perl.com:80')
    or die "can't connect to port 80 on www.perl.com $!";
$handle->autoflush(1);
if (fork()) {               # XXX: undef means failure
    select($handle);
    print while <STDIN>;    # everything from stdin to socket
} else {
    print while <$handle>;  # everything from socket to stdout
}
close $handle;
exit;
```

How can I write expect in Perl?

Once upon a time, there was a library called *chat2.pl* (part of the standard perl distribution), which never really got finished. If you find it somewhere, *don't use it*. These days, your best bet is to look at the *Expect* module available from CPAN, which also requires two other modules from CPAN, *IO::Pty* and *IO::Stty*.

Is there a way to hide perl's command line from programs such as "ps"?

First of all note that if you're doing this for security reasons (to avoid people seeing passwords, for example) then you should rewrite your program so that critical information is never given as an argument. Hiding the arguments won't make your program completely secure.

To actually alter the visible command line, you can assign to the variable $0 as documented in *perlvar* . This won't work on all operating systems, though. Daemon programs like sendmail place their state there, as in:

```
$0 = "orcus [accepting connections]";
```

I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?

Unix

In the strictest sense, it can't be done--the script executes as a different process from the shell it was started from. Changes to a process are not reflected in its parent--only in any children created after the change. There is shell magic that may allow you to fake it by `eval()`ing the script's output in your shell; check out the comp.unix.questions FAQ for details.

How do I close a process's filehandle without waiting for it to complete?

Assuming your system supports such things, just send an appropriate signal to the process (see *"kill" in perlfunc*). It's common to first send a TERM signal, wait a little bit, and then send a KILL signal to finish it off.

How do I fork a daemon process?

If by daemon process you mean one that's detached (disassociated from its tty), then the following process is reported to work on most Unixish systems. Non-Unix users should check their Your_OS::Process module for other solutions.

- Open /dev/tty and use the TIOCNOTTY ioctl on it. See *tty(1)* for details. Or better yet, you can just use the `POSIX::setsid()` function, so you don't have to worry about process groups.
- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they're not connected to the old tty.
- Background yourself like this:

```
fork && exit;
```

The *Proc::Daemon* module, available from CPAN, provides a function to perform these actions for you.

How do I find out if I'm running interactively or not?

(contributed by brian d foy)

This is a difficult question to answer, and the best answer is only a guess.

What do you really want to know? If you merely want to know if one of your filehandles is connected to a terminal, you can try the `-t` file test:

```
if( -t STDOUT ) {
    print "I'm connected to a terminal!\n";
}
```

However, you might be out of luck if you expect that means there is a real person on the other side. With the *Expect* module, another program can pretend to be a person. The program might even come close to passing the Turing test.

The *IO::Interactive* module does the best it can to give you an answer. Its `is_interactive` function returns an output filehandle; that filehandle points to standard output if the module thinks the session is interactive. Otherwise, the filehandle is a null handle that simply discards the output:

```
use IO::Interactive;

print { is_interactive } "I might go to standard output!\n";
```

This still doesn't guarantee that a real person is answering your prompts or reading your output.

If you want to know how to handle automated testing for your distribution, you can check the environment. The CPAN Testers, for instance, set the value of `AUTOMATED_TESTING`:

```
unless( $ENV{AUTOMATED_TESTING} ) {
    print "Hello interactive tester!\n";
}
```

How do I timeout a slow event?

Use the `alarm()` function, probably in conjunction with a signal handler, as documented in *"Signals" in perlipc* and the section on "Signals" in the Camel. You may instead use the more flexible *Sys::AlarmCall* module available from CPAN.

The `alarm()` function is not implemented on all versions of Windows. Check the documentation for your specific version of Perl.

How do I set CPU limits?

(contributed by Xho)

Use the *BSD::Resource* module from CPAN. As an example:

```
use BSD::Resource;
setrlimit(RLIMIT_CPU,10,20) or die $!;
```

This sets the soft and hard limits to 10 and 20 seconds, respectively. After 10 seconds of time spent running on the CPU (not "wall" time), the process will be sent a signal (XCPU on some systems) which, if not trapped, will cause the process to terminate. If that signal is trapped, then after 10 more seconds (20 seconds in total) the process will be killed with a non-trappable signal.

See the *BSD::Resource* and your systems documentation for the gory details.

How do I avoid zombies on a Unix system?

Use the reaper code from *"Signals" in perlipc* to call `wait()` when a SIGCHLD is received, or else use the double-fork technique described in *"How do I start a process in the background?" in perlfaq8*.

How do I use an SQL database?

The *DBI* module provides an abstract interface to most database servers and types, including Oracle, DB2, Sybase, mysql, Postgresql, ODBC, and flat files. The DBI module accesses each database type through a database driver, or DBD. You can see a complete list of available drivers on CPAN: *http://www.cpan.org/modules/by-module/DBD/* . You can read more about DBI on *http://dbi.perl.org/* .

Other modules provide more specific access: *Win32::ODBC*, *Alzabo*, `iodbc`, and others found on CPAN Search: *http://search.cpan.org/* .

How do I make a system() exit on control-C?

You can't. You need to imitate the `system()` call (see *perlipc* for sample code) and then have a signal handler for the INT signal that passes the signal on to the subprocess. Or you can check for it:

```
$rc = system($cmd);
if ($rc & 127) { die "signal death" }
```

How do I open a file without blocking?

If you're lucky enough to be using a system that supports non-blocking reads (most Unixish systems do), you need only to use the `O_NDELAY` or `O_NONBLOCK` flag from the `Fcntl` module in conjunction with `sysopen()`:

```
use Fcntl;
sysopen(my $fh, "/foo/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
```

```
        or die "can't open /foo/somefile: $!":
```

How do I tell the difference between errors from the shell and perl?

(answer contributed by brian d foy)

When you run a Perl script, something else is running the script for you, and that something else may output error messages. The script might emit its own warnings and error messages. Most of the time you cannot tell who said what.

You probably cannot fix the thing that runs perl, but you can change how perl outputs its warnings by defining a custom warning and die functions.

Consider this script, which has an error you may not notice immediately.

```
#!/usr/locl/bin/perl

print "Hello World\n";
```

I get an error when I run this from my shell (which happens to be bash). That may look like perl forgot it has a `print()` function, but my shebang line is not the path to perl, so the shell runs the script, and I get the error.

```
$ ./test
./test: line 3: print: command not found
```

A quick and dirty fix involves a little bit of code, but this may be all you need to figure out the problem.

```
#!/usr/bin/perl -w

BEGIN {
    $SIG{__WARN__} = sub{ print STDERR "Perl: ", @_; };
    $SIG{__DIE__}  = sub{ print STDERR "Perl: ", @_; exit 1};
}

$a = 1 + undef;
$x / 0;
__END__
```

The perl message comes out with "Perl" in front. The `BEGIN` block works at compile time so all of the compilation errors and warnings get the "Perl:" prefix too.

```
Perl: Useless use of division (/) in void context at ./test line 9.
Perl: Name "main::a" used only once: possible typo at ./test line 8.
Perl: Name "main::x" used only once: possible typo at ./test line 9.
Perl: Use of uninitialized value in addition (+) at ./test line 8.
Perl: Use of uninitialized value in division (/) at ./test line 9.
Perl: Illegal division by zero at ./test line 9.
Perl: Illegal division by zero at -e line 3.
```

If I don't see that "Perl:", it's not from perl.

You could also just know all the perl errors, and although there are some people who may know all of them, you probably don't. However, they all should be in the *perldiag* manpage. If you don't find the error in there, it probably isn't a perl error.

Looking up every message is not the easiest way, so let perl to do it for you. Use the diagnostics

pragma with turns perl's normal messages into longer discussions on the topic.

```
use diagnostics;
```

If you don't get a paragraph or two of expanded discussion, it might not be perl's message.

How do I install a module from CPAN?

(contributed by brian d foy)

The easiest way is to have a module also named CPAN do it for you by using the `cpan` command that comes with Perl. You can give it a list of modules to install:

```
$ cpan IO::Interactive Getopt::Whatever
```

If you prefer `CPANPLUS`, it's just as easy:

```
$ cpanp i IO::Interactive Getopt::Whatever
```

If you want to install a distribution from the current directory, you can tell `CPAN.pm` to install `.` (the full stop):

```
$ cpan .
```

See the documentation for either of those commands to see what else you can do.

If you want to try to install a distribution by yourself, resolving all dependencies on your own, you follow one of two possible build paths.

For distributions that use *Makefile.PL*:

```
$ perl Makefile.PL
$ make test install
```

For distributions that use *Build.PL*:

```
$ perl Build.PL
$ ./Build test
$ ./Build install
```

Some distributions may need to link to libraries or other third-party code and their build and installation sequences may be more complicated. Check any *README* or *INSTALL* files that you may find.

What's the difference between require and use?

(contributed by brian d foy)

Perl runs `require` statement at run-time. Once Perl loads, compiles, and runs the file, it doesn't do anything else. The `use` statement is the same as a `require` run at compile-time, but Perl also calls the `import` method for the loaded package. These two are the same:

```
use MODULE qw(import list);

BEGIN {
    require MODULE;
    MODULE->import(import list);
}
```

However, you can suppress the `import` by using an explicit, empty import list. Both of these still happen at compile-time:

```
use MODULE ();

BEGIN {
    require MODULE;
}
```

Since `use` will also call the `import` method, the actual value for `MODULE` must be a bareword. That is, `use` cannot load files by name, although `require` can:

```
require "$ENV{HOME}/lib/Foo.pm"; # no @INC searching!
```

See the entry for `use` in *perlfunc* for more details.

How do I keep my own module/library directory?

When you build modules, tell Perl where to install the modules.

If you want to install modules for your own use, the easiest way might be *local::lib*, which you can download from CPAN. It sets various installation settings for you, and uses those same settings within your programs.

If you want more flexibility, you need to configure your CPAN client for your particular situation.

For `Makefile.PL`-based distributions, use the INSTALL_BASE option when generating Makefiles:

```
perl Makefile.PL INSTALL_BASE=/mydir/perl
```

You can set this in your `CPAN.pm` configuration so modules automatically install in your private library directory when you use the CPAN.pm shell:

```
% cpan
cpan> o conf makepl_arg INSTALL_BASE=/mydir/perl
cpan> o conf commit
```

For `Build.PL`-based distributions, use the --install_base option:

```
perl Build.PL --install_base /mydir/perl
```

You can configure `CPAN.pm` to automatically use this option too:

```
% cpan
cpan> o conf mbuild_arg "--install_base /mydir/perl"
cpan> o conf commit
```

INSTALL_BASE tells these tools to put your modules into */mydir/perl/lib/perl5*. See *How do I add a directory to my include path (@INC) at runtime?* for details on how to run your newly installed modules.

There is one caveat with INSTALL_BASE, though, since it acts differently from the PREFIX and LIB settings that older versions of *ExtUtils::MakeMaker* advocated. INSTALL_BASE does not support installing modules for multiple versions of Perl or different architectures under the same directory. You should consider whether you really want that and, if you do, use the older PREFIX and LIB settings. See the *ExtUtils::Makemaker* documentation for more details.

How do I add the directory my program lives in to the module/library search path?

(contributed by brian d foy)

If you know the directory already, you can add it to `@INC` as you would for any other directory. You might <use lib> if you know the directory at compile time:

```
use lib $directory;
```

The trick in this task is to find the directory. Before your script does anything else (such as a `chdir`), you can get the current working directory with the `Cwd` module, which comes with Perl:

```
BEGIN {
    use Cwd;
    our $directory = cwd;
}

use lib $directory;
```

You can do a similar thing with the value of `$0`, which holds the script name. That might hold a relative path, but `rel2abs` can turn it into an absolute path. Once you have the

```
BEGIN {
    use File::Spec::Functions qw(rel2abs);
    use File::Basename qw(dirname);

    my $path   = rel2abs( $0 );
    our $directory = dirname( $path );
}

use lib $directory;
```

The *FindBin* module, which comes with Perl, might work. It finds the directory of the currently running script and puts it in `$Bin`, which you can then use to construct the right library path:

```
use FindBin qw($Bin);
```

You can also use *local::lib* to do much of the same thing. Install modules using *local::lib*'s settings then use the module in your program:

```
 use local::lib; # sets up a local lib at ~/perl5
```

See the *local::lib* documentation for more details.

How do I add a directory to my include path (@INC) at runtime?

Here are the suggested ways of modifying your include path, including environment variables, run-time switches, and in-code statements:

the `PERLLIB` environment variable

```
$ export PERLLIB=/path/to/my/dir
$ perl program.pl
```

the `PERL5LIB` environment variable

```
$ export PERL5LIB=/path/to/my/dir
$ perl program.pl
```

the `perl -Idir` command line flag

```
$ perl -I/path/to/my/dir program.pl
```

the `lib` pragma:

```
use lib "$ENV{HOME}/myown_perllib";
```

the *local::lib* module:

```
use local::lib;

use local::lib "~/myown_perllib";
```

The last is particularly useful because it knows about machine-dependent architectures. The `lib.pm` pragmatic module was first included with the 5.002 release of Perl.

Where are modules installed?

Modules are installed on a case-by-case basis (as provided by the methods described in the previous section), and in the operating system. All of these paths are stored in @INC, which you can display with the one-liner

```
perl -e 'print join("\n",@INC,"")'
```

The same information is displayed at the end of the output from the command

```
perl -V
```

To find out where a module's source code is located, use

```
perldoc -l Encode
```

to display the path to the module. In some cases (for example, the `AutoLoader` module), this command will show the path to a separate `pod` file; the module itself should be in the same directory, with a 'pm' file extension.

What is socket.ph and where do I get it?

It's a Perl 4 style file defining values for system networking constants. Sometimes it is built using *h2ph* when Perl is installed, but other times it is not. Modern programs should use `use Socket;` instead.

AUTHOR AND COPYRIGHT