Chapter 15. Process Management

One of the best parts of being a programmer is launching someone else's code so that you don't have to write it yourself. It's time to learn how to manage your children—child processes, that is—by launching other programs directly from Perl.

And like everything else in Perl, There's More Than One Way To Do It, with lots of overlap, variations, and special features. So, if you don't like the first way, just read on for another page or two for a solution more to your liking.

Perl is very portable; most of the rest of this book doesn't need many notes saying that it works this way in Unix systems and that way in Windows and some other way on VMS. But when you're starting other programs on your machine, different programs are available on a Macintosh than what you'll likely find on an old Cray (which used to be a "super" computer). The examples in this chapter are primarily Unix based; if you have a non-Unix system, you can expect to see some differences.

The system Function

The simplest way to launch a child process in Perl to run a program is with the `system` function. For example, to invoke the Unix *date* command from within Perl, you tell `system` that's the program you want to run:

```
system 'date';
```

These commands depend on your system, what it provides, and how it implements those commands. They aren't Perl, but something Perl asks the system to do for your program. The same Unix command might have

different calling conventions and options on different versions of that operating system.

If you are using Windows, that code will show you the date but also prompt you to enter a new date. Your program will wait for you to enter a new date. You probably want the `/T` switch to suppress that:

```
system 'date /T';
```

You run that from the *parent* process. When it runs, the `system` command creates an identical copy of your Perl program, called the *child* process. The child process immediately changes itself into the command that you want to run, such as *date*, sharing Perl's standard input, standard output, and standard error. This means that the normal short date-and-time string generated by *date* ends up wherever Perl's `STDOUT` was already going.

The parameter to the `system` function is generally whatever you normally type at the shell. So, if it were a more complicated command, like *ls -l $HOME* to list the contents of your home directory, you could put all that into the parameter:

```
system 'ls -l $HOME';
```

That `$HOME` is a shell variable that knows the path to your home directory. It's not a Perl variable and you don't want to interpolate it. If you had it in double quotes, you'd have to escape the `$` to prevent the interpolation:

```
system "ls -l \$HOME";
```

On Windows, the same task uses the *dir* command. The `%` signs belong to the command, not a Perl variable. But hashes don't interpolate in double-quoted strings, so you don't need to escape them:

```
system "cmd /c dir %userprofile%"
```

NOTE

If you have Cygwin or MinGW installed, some of the Windows command shell commands may run something different than what you expect. Using `cmd /c` ensures that you get the Windows version.

Now, the normal Unix *date* command is output-only, but let's say it's a chatty command, asking first "for which time zone do you want the time?", or that the Windows version prompts you for the new date. That message ends up on standard output and the program listens on standard input (inherited from Perl's `STDIN`) for the response. You see the question, and type in the answer (like "Zimbabwe time"), and then *date* will finish its duty.

While the child process is running, Perl patiently waits for it to finish. So if the *date* command takes 37 seconds, Perl pauses for those 37 seconds. You can use the shell's facility to launch a background process, however:

```
system "long_running_command with parameters &";
```

Here, the shell gets launched, which then notices the ampersand at the end of the command line, causing the shell to put `long_running_command` into the background. And then the shell exits rather quickly; Perl notices this and moves on. In this case, the `long_running_command` is really a *grandchild* of the Perl process, to which Perl really has no direct access or knowledge.

Windows doesn't have a backgrounding mechanism, but *start* can run the command without your program waiting for it:

```
system 'start /B long_running_command with parameters'
```

When a command is "simple enough," no shell gets involved. So for the *date* and *ls* commands earlier, Perl directly launched your requested com-

mand, which searches the inherited `PATH` to find the command, if necessary. But if there's anything weird in the string (such as shell metacharacters like the dollar sign, semicolon, or vertical bar), Perl invokes the standard Bourne Shell (*/bin/sh*) in Unix or the shell set in the `PERL5SHELL` environment variable in Windows (by default, that's *cmd /x/d/c*).

NOTE

The `PATH` is the list of directories your system searches to find programs. You can change `PATH` by adjusting `$ENV{'PATH'}` at any time.

For example, you can write an entire little shell script in the argument. This one prints the contents of all the (nonhidden) files in the current directory:

```
system 'for i in *; do echo == $i ==; cat $i; done';
```

Here, again, you're using single quotes because the dollar signs are for the shell and not for Perl. Double quotes would allow Perl to interpolate `$i` to its current Perl value and not let the shell expand it to its own value.

On Windows you don't have those interpolation issues. The `/R` works recursively, so you might end up with a long list of files:

```
system 'for /R %i in (*) DO echo %i & type %i'
```

Note that your ability to do this doesn't mean it's wise for you to do this. You know it's possible, but often there's a pure Perl solution that will do the same thing. On the other hand, Perl is a glue language meant to work in the ugly spaces between programs that need to coordinate with each other.

Avoiding the Shell

The system operator may also be invoked with more than one argument, in which case a shell doesn't get involved, no matter how complicated the text:

```
my $tarfile = 'something*wicked.tar';
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system 'tar', 'cvf', $tarfile, @dirs;
```

NOTE

`system` can use an indirect object, such as `system { 'fred' } 'barney';`, which runs the program `barney`, but lies to it so that it thinks it's called `'fred'`. See the perlsec documentation or *Mastering Perl*'s security chapter for more details.

In this case, the first parameter (`'tar'` here) gives the name of a command found in the normal `PATH`-searching way, while Perl passes the remaining arguments one by one, directly to that command. Even if the arguments have shell-significant characters, such as the name in `$tarfile` or the directory names in `@dirs`, the shell never gets a chance to mangle the string. That *tar* command will get precisely five parameters. Compare that with this security problem:

```
system "tar cvf $tarfile @dirs";  # Oops!
```

Here, you've now piped a bunch of stuff into a *flintstone* command, put it into the background, and opened *betty* for output. That's a relatively tame effect, but what if `@dirs` was something more interesting, such as:

```
my @dirs = qw( ; rm -rf / );
```

It doesn't matter that `@dirs` is a list, because Perl simply interpolates it into the single string to pass to `system`.

And that's a bit scary, especially if those variables are from user input—such as from a web form or something. So if you *can* arrange things so that you can use the multiple-argument version of `system`, you probably should use that way to launch your subprocess. You'll have to give up the ability to have the shell do the work for you to set up I/O redirection, background processes, and the like, though. There's no such thing as a free lunch.

Note that redundantly, a single-argument invocation of `system` is nearly equivalent to the proper multiple-argument version of `system`:

```
system $command_line;
system '/bin/sh', '-c', $command_line;
```

But nobody writes the latter since that's what Perl does already. If you want things processed by a different shell, like the C-shell, you can specify that:

```
system '/bin/csh', '-fc', $command_line;
```

This is also handy for handling whitespace in filenames since the shell doesn't step in to break up the arguments. This command sees exactly one filename:

```
system 'touch', 'name with spaces.txt';
```

NOTE

See *Mastering Perl* for a longer discussion of the security features of the list form of `system`. The perlsec documentation comes in handy too.

On Windows, you can set the `$ENV{PERL5SHELL}` value to be the shell that you want. You'll see environment variables in the next section, so keep reading.

The return value of the system operator is based on the exit status of the child command:

```
unless (system 'date') {
  # Return was zero, meaning success
  print "We gave you a date, OK!\n";
}
```

An exit value of `0` usually means that everything is OK, and a nonzero exit value usually indicates that something went wrong. This is part of the "0 but true" notion that the value of zero is a good thing. This is backward from the normal "true is good—false is bad" strategy for most of the operators, so to write a typical "do this or die" style, we'll need to flip false and true. The easiest way is to simply prefix the `system` operator with a bang (the logical-not operator):

```
!system 'rm -rf files_to_delete' or die 'something went wrong';
```

In this case, including `$!` in the error message is not appropriate, because the failure is most likely somewhere within the experience of the external *rm* command, and it's not a `system`-related error within Perl that `$!` can reveal.

Don't rely on this behavior, though. It's up to each command to decide what it wants to return. Some values may be nonzero but also indicate success. If that's the case, you need to inspect the return value more closely.

The `system` return value is two octets. The "high" octet has the exit value of the program. If you want that, you need to shift the bits down eight bits (remember the bit operators from Chapter 12):

```
my $return_value    = system( ... );
my $child_exit_code = $return_value >> 8;
```

The “low” octet combines several things. The highest bit notes if a core dump happened. The hexadecimal and binary representations (recall them from Chapter 2) can help mask out the parts you don’t want:

```
my $low_octet     = $return_value & 0xFF; # mask out high octet
my $dumped_core   = $low_octet & 0b1_0000000;  # 128
my $signal_number = $low_octet & 0b0111_1111;  # 0x7f, or 127
```

Since Windows doesn’t have signals, the bits in these positions may have other meanings.

NOTE

Your system may have a more specific error message in the variable `$^E` or `${^CHILD_ERROR_NATIVE}`. See perlrun and the `POSIX` module (especially the `W*` macros to decode signals).

The Environment Variables

When you’re starting another process (with any of the methods we show here), you may need to set up its environment in one way or another. As we mentioned earlier, you could start the process with a certain working directory, which it inherits from your process. Another common configuration detail is the environment variables.

One of the best-known environment variables is `PATH`. (If you’ve never heard of it, you probably haven’t used a system that has environment variables.) On Unix and similar systems, `PATH` is a colon-separated list of directories that may hold programs. When you type a command like *rm fred*, the system will look for the *rm* command in that list of directories, in order. Perl (or your system) will use `PATH` whenever it needs to find the program to run. If the program in turn runs other programs, those may also be found along the `PATH`. (Of course, if you give a complete name for a command, such as */bin/echo*, there’s no need to search `PATH`. But that’s generally much less convenient.)

In Perl, the environment variables are available via the special `%ENV` hash; each key in this hash represents one environment variable. At the start of your program's execution, `%ENV` holds values it has inherited from its parent process (generally the shell). Modifying this hash changes the environment variables, which will then be inherited by new processes and possibly used by Perl as well. For example, suppose you wished to run the system's *make* utility (which typically runs other programs), and you want to use a private directory as the first place to look for commands (including *make* itself). And let's say that you don't want the `IFS` environment variable to be set when you run the command, because that might cause *make* or some subcommand to do the wrong thing. Here we go:

```
$ENV{'PATH'} = "/home/rootbeer/bin:$ENV{'PATH'}";
delete $ENV{'IFS'};
my $make_result = system 'make';
```

Different systems construct their paths differently. Unix uses colons, but Windows uses semicolons, for instance. This is your constant headache of working with external programs. You have to know a lot that isn't Perl. But Perl knows about the system it runs on, and you can find out what it knows with the `Config` module through its `%Config` variable. Instead of assuming the `PATH` separator as you did in the previous example, you could use `join` with a glue string you get from `%Config`:

```
use Config;
$ENV{'PATH'} = join $Config{'path_sep'},
    '/home/rootbeer/bin', $ENV{'PATH'};
```

Newly created processes will generally inherit from their parent the environment variables, the current working directory, the standard input, output, and error streams, and a few more esoteric items. See the documentation about programming on your system for more details. (But on most systems, your program can't change the environment for the shell or other parent process that started it.)

The exec Function

Everything we've just said about `system` syntax and semantics is also true about the `exec` function, except for one (very important) thing. The `system` function creates a child process, which then scurries off to perform the requested action while Perl naps. The `exec` function causes the Perl process *itself* to perform the requested action. Think of it as more like a "goto" than a subroutine call.

For example, suppose you wanted to run the *bedrock* command in the */tmp* directory, passing it arguments of *-o args1* followed by whatever arguments your own program was invoked with. That'd look like this:

```
chdir '/tmp' or die "Cannot chdir /tmp: $!";
exec 'bedrock', '-o', 'args1', @ARGV;
```

When you reach the `exec` operation, Perl locates *bedrock* and "jumps into it." At that point, there is no Perl process anymore, even though it's the same process, having performed the Unix `exec` system call (or equivalent). The process ID remains the same, but it's now just the process running the *bedrock* command. When *bedrock* finishes, there's no Perl to come back to.

Why is this useful? Sometimes you want to use Perl to set up the environment for a program. You can affect environment variables, change the current working directory, and change the default filehandles:

```
$ENV{PATH}  = '/bin:/usr/bin';
$ENV{DEBUG} = 1;
$ENV{ROCK}  = 'granite';

chdir '/Users/fred';
open STDOUT, '>', '/tmp/granite.out';

exec 'bedrock';
```

If you use `system` instead of `exec`, you have a Perl program just standing around tapping its toes, waiting for the other program to complete just so Perl could finally immediately exit as well, and that wastes a resource.

Having said that, it's actually quite rare to use `exec`, except in combination with `fork` (which you'll see later). If you are puzzling over `system` versus `exec`, just pick `system`, and nearly all of the time you'll be just fine.

Because Perl is no longer in control once the requested command has started, it doesn't make any sense to have any Perl code following the `exec`, except for handling the error when the requested command cannot be started:

```
exec 'date';
die "date couldn't run: $!";
```

Using Backquotes to Capture Output

With both `system` and `exec`, the output of the launched command ends up wherever Perl's standard output is going. Sometimes it's interesting to capture that output as a string value to perform further processing. And that's done simply by creating a string using backquotes instead of single or double quotes:

```
my $now = `date`;               # grab the output of date
print "The time is now $now"; # newline already present
```

Normally, this *date* command spits out a string approximately 30 characters long to its standard output, giving the current date and time followed by a newline. When you've placed *date* between backquotes, Perl executes the *date* command, arranging to capture its standard output as a string value, and in this case assigning it to the `$now` variable.

This is very similar to the Unix shell's meaning for backquotes. However, the shell also performs the additional job of ripping off the final end-of-line to make it easier to use the value as part of other things. Perl is honest; it gives the real output. To get the same result in Perl, you can simply add an additional `chomp` operation on the result:

```
chomp(my $no_newline_now = `date`);
print "A moment ago, it was $no_newline_now, I think.\n";
```

The value between backquotes is just like the single-argument form of `system` and is interpreted as a double-quoted string, meaning that backslash-escapes and variables are expanded appropriately. For example, to fetch the Perl documentation on a list of Perl functions, we might invoke the *perldoc* command repeatedly, each time with a different argument:

```
my @functions = qw{ int rand sleep length hex eof not exit sqrt umask };
my %about;

foreach (@functions) {
  $about{$_} = `perldoc -t -f $_`;
}
```

Note that `$_` has a different value for each invocation, letting you grab the output of a different command that varies in only one of its parameters. Also note that if you haven't seen some of these functions yet, it might be useful to look them up in the documentation to see what they do!

Instead of the backquotes, you can use the generalized quoting operator, `qx()`, that does the same thing:

```
foreach (@functions) {
  $about{$_} = qx(perldoc -t -f $_);
}
```

As with the other generalized quotes, you mainly use this when the stuff inside the quotes also contains the default delimiter. If you want to have a

literal backquote in your command, you can use the `qx()` mechanism to avoid the hassle of escaping the offending character. There's another benefit to generalized quoting: if you use the single quote as the delimiter, the quoting does not interpolate anything. If you want to use the shell's process ID variable, `$$`, instead of Perl's, you use `qx''` to avoid the interpolation:

```
my $output = qx'echo $$';
```

At the risk of actually introducing the behavior by demonstrating how *not* to do it, we'd also like to suggest that you avoid using backquotes in a place where the value isn't being captured. For example:

```
print "Starting the frobnitzigator:\n";
`frobnitz -enable`; # no need to do this if you ignore the string
print "Done!\n";
```

The problem is that Perl has to work a bit harder to capture the output of this command, even if you don't use it. This is known as *void context* and you should generally avoid asking Perl to do work when you won't use the result. You also lose the option to use multiple arguments to `system` to precisely control the argument list. So from both a security standpoint and an efficiency viewpoint, just use `system` instead, please.

Standard error of a backquoted command goes to the same place as Perl's current standard error output. If the command spits out error messages to the default standard error, you'll probably see them on the terminal, which could be confusing to the user who hasn't personally invoked the *frobnitz* command but still sees its errors. If you want to capture error messages with standard output, you can use the shell's normal "merge standard error to the current standard output," which is spelled `2>&1` in the normal Unix and Windows shells:

```
my $output_with_errors = `frobnitz -enable 2>&1`;
```

Note that this will intermingle the standard error output with the standard output, much as it appears on the terminal (although possibly in a slightly different sequence because of buffering). If you need the output and the error output separated, there are many more flexible solutions, such as `IPC::Open3` in the standard Perl library, or writing your own forking code, as you will see later. Similarly, standard input is inherited from Perl's current standard input. Most commands you typically use with backquotes do not read standard input, so that's rarely a problem. However, let's say the *date* command asked which time zone (as we imagined earlier). That'll be a problem because the prompt for "which time zone" will be sent to standard output, which is being captured as part of the value, and then the *date* command will start trying to read from standard input. But since the user has never seen the prompt, they don't know they should be typing anything! Pretty soon, the user calls you up and tells you your program is stuck.

So, stay away from commands that read standard input. If you're not sure whether something reads from standard input, add a redirection from */dev/null* for input, like this for Unix:

```
my $result = `some_questionable_command arg arg argh </dev/null`;
```

and like this for Windows:

```
my $result = `some_questionable_command arg arg argh < NUL`;
```

Then the child shell will redirect input from the "null device," and the questionable grandchild command will, at worst, try to read and immediately get an end-of-file.

NOTE

The `Capture::Tiny` and `IPC::System::Simple` modules can capture the output while handling the system-specific details for you. Install them from CPAN.

Using Backquotes in a List Context

The scalar context use of backquotes returns the captured output as a single long string, even if it looks to you like there are multiple "lines" because it has newlines. Computers don't care about lines, really. That's something we care about and tell computers to interpret for us. Those newlines are just another character as far as a computer is concerned. However, using the same backquoted string in a list context yields a list containing one line of output per element.

For example, the Unix *who* command normally spits out a line of text for each current login on the system as follows:

```
merlyn     tty/42     Dec 7  19:41
rootbeer   console    Dec 2  14:15
rootbeer   tty/12     Dec 6  23:00
```

The left column is the username, the middle column is the TTY name (that is, the name of the user's connection to the machine), and the rest of the line is the date and time of login (and possibly remote login information, but not in this example). In a scalar context, we get all that at once, which we would then need to split up on our own:

```
my $who_text = `who`;
my @who_lines = split /\n/, $who_text;
```

But in a list context, we automatically get the data broken up by lines:

```
my @who_lines = `who`;
```

You'll have a number of separate elements in `@who_lines`, each one terminated by a newline. Of course, adding a `chomp` around the outside of that will rip off those newlines, but you can go in a different direction. If you put that as part of the value for a `foreach`, you'll iterate over the lines automatically, placing each one in `$_`:

```
foreach (`who`) {
  my($user, $tty, $date) = /(\S+)\s+(\S+)\s+(.*)/;
  $ttys{$user} .= "$tty at $date\n";
}
```

This loop will iterate three times for the sample `who` output. (Your system will probably have more than three active logins at any given time.) Notice that you have a regular expression match, and in the absence of the binding operator (`=~`), it matches against `$_` —which is good, because that's where the data is.

Also notice the regular expression looks for a nonblank word, some whitespace, a nonblank word, some whitespace, and then the rest of the line up to, but not including, the newline (since dot doesn't match newline by default). That's also good, because that's what the data looks like each time in `$_` . That'll make `$1` be `merlyn` , `$2` be `tty/42` , and `$3` be `Dec 7 19:41` , as a successful match on the first time through the loop.

NOTE

Now you can see *why* dot (or `\N`) doesn't match newline by default. It makes it easy to write patterns like this one, in which we don't have to worry about a newline at the end of the string.

However, this regular expression match is in a list context, so you get the list of memories instead of the true/false "did it match" value, as you saw in Chapter 8. So, `$user` ends up being `merlyn` , and so on.

The second statement inside the loop simply stores away the TTY and date information, appending to a (possibly `undef`) value in the hash, because a user might be logged in more than once, as user `rootbeer` was in that example.

External Processes with IPC::System::Simple

Running or capturing output from external commands is tricky business, especially since Perl aims to work on so many diverse platforms, each with its own way of doing things. Paul Fenwick's `IPC::System::Simple` module fixes that by providing a simpler interface that hides the complexity of the operating system–specific stuff. It doesn't come with Perl (yet), so you have to get it from CPAN.

There's really not that much to say about this module, because it is truly simple. You can use it to replace the built-in `system` with its own, more robust version:

```
use IPC::System::Simple qw(system);

my $tarfile = 'something*wicked.tar';
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system 'tar', 'cvf', $tarfile, @dirs;
```

It also provides a `systemx` that never uses the shell, so you should never have the problem of unintended shell actions:

```
systemx 'tar', 'cvf', $tarfile, @dirs;
```

If you want to capture the output, you change the `system` or `systemx` to `capture` or `capturex`, both of which work like backquotes (but better):

```
my @output = capturex 'tar', 'cvf', $tarfile, @dirs;
```

Paul put in a lot of work to ensure that these subroutines do the right thing under Windows. There's a lot more that this module can do to make your life easier, although we'll refer you to the module documentation for that since some of the fancier features require references, which we don't show you until you read *Intermediate Perl*. If you can use this module, we recommend it over the built-in Perl operators for the same thing.

Processes as Filehandles

So far, you've seen ways to deal with synchronous processes, where Perl stays in charge, launches a command, (usually) waits for it to finish, then possibly grabs its output. But Perl can also launch a child process that stays alive, communicating to Perl on an ongoing basis until the task is complete.

The syntax for launching a concurrent (parallel) child process is to put the command as the "filename" for an `open` call, and either precede or follow the command with a vertical bar, which is the "pipe" character. For that reason, this is often called a *piped open*. In the two-argument form, the pipe goes before or after the command that you want to run:

```
open DATE, 'date|' or die "cannot pipe from date: $!";
open MAIL, '|mail merlyn' or die "cannot pipe to mail: $!";
```

In the first example, with the vertical bar on the right, Perl launches the command with its standard output connected to the `DATE` filehandle opened for reading, similar to the way that the command *date | your_program* would work from the shell. In the second example, with the vertical bar on the left, Perl connects the command's standard input to the `MAIL` filehandle opened for writing, similar to what happens with the command *your_program | mail merlyn*. In either case, the command continues independently of the Perl process. The `open` fails if Perl can't start the child process. If the command itself does not exist or exits erroneously, Perl will not see this as an error when opening the filehandle, but as an error when closing it. We'll get to that in a moment.

NOTE

If the Perl process exits before the command is complete, a command that's been reading will see end-of-file, while a command that's been writing will get a "broken pipe" error signal on the next write, by default.

The three-argument form is a bit tricky, because for the read filehandle, the pipe character comes after the command. There are special modes for that, though. For the filehandle mode, if you want a read filehandle, you

use `-|`, and if you want a write filehandle, you use `|-` to show which side of the pipe you want to place the command:

```
open my $date_fh, '-|', 'date' or die "cannot pipe from date: $!";
open my $mail_fh, '|-', 'mail merlyn'
    or die "cannot pipe to mail: $!";
```

The pipe `open` can also take more than three arguments. The fourth and subsequent arguments become the arguments to the command, so you can break up that command string to separate the command name from its arguments:

```
open my $mail_fh, '|-', 'mail', 'merlyn'
    or die "cannot pipe to mail: $!";
```

Sadly, the list form of the piped `open` doesn't work in Windows. You'll have to settle for a module to do that for you.

Either way, for all intents and purposes, the rest of the program doesn't know, doesn't care, and would have to work pretty hard to figure out that this is a filehandle opened on a process rather than on a file. So, to get data from a filehandle opened for reading, you read the filehandle normally:

```
my $now = <$date_fh>;
```

And to send data to the mail process (waiting for the body of a message to deliver to `merlyn` on standard input), a simple print-with-a-filehandle will do:

```
print $mail_fh "The time is now $now"; # presume $now ends in newline
```

In short, you can pretend that these filehandles are hooked up to magical files, one that contains the output of the *date* command, and one that will automatically be mailed by the *mail* command.

If a process is connected to a filehandle that is open for reading, and then exits, the filehandle returns end-of-file, just like reading up to the end of a normal file. When you close a filehandle open for writing to a process, the process will see end-of-file. So, to finish sending the email, close the handle:

```
close $mail_fh;
die "mail: nonzero exit of $?" if $?;
```

When closing a filehandle attached to a process, Perl waits for the process to complete so that it can get the process's exit status. The exit status is then available in the `$?` variable (reminiscent of the same variable in the Bourne Shell) and is the same kind of number as the value returned by the `system` function: zero for success, nonzero for failure. Each new exited process overwrites the previous value, though, so save it quickly if you want it. (The `$?` variable also holds the exit status of the most recent `system` or backquoted command, if you're curious.)

The processes are synchronized just like a pipelined command. If you try to read and no input is available, the process is suspended (without consuming additional CPU time) until the sending program has started speaking again. Similarly, if a writing process gets ahead of the reading process, the writing process is slowed down until the reader starts to catch up. There's a buffer (usually 8 KB or so) in between, so they don't have to stay precisely in lockstep.

Why use processes as filehandles? Well, it's the only easy way to write to a process based on the results of a computation. But if you're just reading, backquotes are often much easier to manage, unless you want to have the results as they come in.

For example, the Unix *find* command locates files based on their attributes, and it can take quite a while if used on a fairly large number of files (such as starting from the root directory). You can put a *find* command inside backquotes, but it's often nicer to see the results as they are found:

```
open my $find_fh, '-|',
  'find', qw( / -atime +90 -size +1000 -print )
    or die "cannot pipe from find: $!";
while (<$find_fh>) {
  chomp;
  printf "%s size %dK last accessed %.2f days ago\n",
    $_, (1023 + -s $_)/1024, -A $_;
}
```

That *find* command looks for all the files that have not been accessed within the past 90 days and that are larger than 1,000 blocks (these are good candidates to move to longer-term storage). While *find* is searching and searching, Perl can wait. As it finds each file, Perl responds to the incoming name and displays some information about that file for further research. Had this been written with backquotes, you would not see any output until the *find* command had completely finished, and it's comforting to see that it's actually doing the job even before it's done.

Getting Down and Dirty with fork

In addition to the high-level interfaces already described, Perl provides nearly direct access to the low-level process management system calls of Unix and some other systems. If you've never done this before, you will probably want to skip this section. While it's a bit much to cover all that stuff in a chapter like this, let's at least look at a quick reimplementation of this:

```
system 'date';
```

You can do that using the low-level system calls:

```
defined(my $pid = fork) or die "Cannot fork: $!";
unless ($pid) {
  # Child process is here
  exec 'date';
  die "cannot exec date: $!";
}
```

```
# Parent process is here
waitpid($pid, 0);
```

NOTE

Windows does not support a native `fork`, but Perl tries to fake it. If you want to do this sort of thing, you can use `Win32::Process` or a similar module for native process management.

Here, you check the return value from `fork`, which is `undef` if it failed. Usually it succeeds, causing two separate processes to continue to the next line, but only the parent process has a nonzero value in `$pid`, so only the child process executes the `exec` function. The parent process skips over that and executes the `waitpid` function, waiting for that particular child to finish (if others finish in the meantime, they are ignored). If that all sounds like gobbledygook, just remember that you can continue to use the `system` function without being laughed at by your friends.

When you go to this extra trouble, you also have full control over creating arbitrary pipes, rearranging filehandles, and noticing your process ID and your parent's process ID (if knowable). But again, that's all a bit complicated for this chapter, so see the details in the perlipc documentation (and in any good book on application programming for your system) for further information.

Sending and Receiving Signals

A Unix signal is a tiny message sent to a process. It can't say much; it's like a car horn honking. Does that honk you hear mean "look out—the bridge collapsed" or "the light has changed—get going" or "stop driving—you've got a baby on the roof" or "hello, world"? Well, fortunately, Unix signals are a little easier to interpret than that because there's a different one for each of these situations. Well, not *exactly* these situations, but analogous Unix-like ones. For these, the signals are `SIGHUP`, `SIGCONT`, `SIGINT`, and the fake `SIGZERO` (signal number zero).

NOTE

Windows implements a subset of POSIX signals, so much of this might not be true on that system.

Different signals are identified by a name (such as `SIGINT`, meaning "interrupt signal") and a corresponding small integer (in the range from 1 to 16, 1 to 32, or 1 to 63, depending on your Unix flavor). Programs or the operating system typically send signals to another program when a significant event happens, such as pressing the interrupt character (typically Ctrl-C) on the terminal, which sends a `SIGINT` to all the processes attached to that terminal. Some signals are sent automatically by the system, but they can also come from another process.

You can send signals from your Perl process to another process, but you have to know the target's process ID number. How you figure that out is a bit complicated, but let's say you know that you want to send a `SIGINT` to process 4201. That's easy enough if you know that `SIGINT` corresponds to the number `2`:

```
kill 2, 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

It's named "kill" because one of the primary purposes of signals is to stop a process that's gone on long enough. You can also use the string `'INT'` in place of the `2`, so you don't have to know the number:

```
kill 'INT', 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

You can even use the `=>` operator to automatically quote the signal name:

```
kill INT => 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

On a Unix system, the *kill* command (not the Perl built-in) can translate between the signal number and the name:

```
$ kill -l 2
INT
```

Or, given a name, it can give you the number:

```
$ kill -l INT
2
```

With no argument to `-l`, it prints all the numbers and names:

```
$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
 5) SIGTRAP     6) SIGABRT     7) SIGEMT      8) SIGFPE
 9) SIGKILL    10) SIGBUS     11) SIGSEGV    12) SIGSYS
13) SIGPIPE    14) SIGALRM    15) SIGTERM    16) SIGURG
17) SIGSTOP    18) SIGTSTP    19) SIGCONT    20) SIGCHLD
21) SIGTTIN    22) SIGTTOU    23) SIGIO      24) SIGXCPU
25) SIGXFSZ    26) SIGVTALRM  27) SIGPROF    28) SIGWINCH
29) SIGINFO    30) SIGUSR1    31) SIGUSR2
```

If you try to interrupt a process that no longer exists or isn't yours, you'll get a false return value.

You can also use this technique to see whether a process is still alive. A special signal number of `0` says, "Just check to see whether I *could* send a signal if I wanted to, but I don't want to, so don't actually send anything." So a process probe might look like:

```
unless (kill 0, $pid) {
  warn "$pid has gone away!";
}
```

Perhaps a little more interesting than sending signals is catching signals. Why might you want to do this? Well, suppose you have a program that creates files in */tmp*, and you normally delete those files at the end of the program. If someone presses Ctrl-C during the execution, that leaves trash

in */tmp*, a very impolite thing to do. To fix this, you can create a signal handler that takes care of the cleanup:

```
my $temp_directory = "/tmp/myprog.$$"; # create files below here
mkdir $temp_directory, 0700 or die "Cannot create $temp_directory: $!";

sub clean_up {
  unlink glob "$temp_directory/*";
  rmdir $temp_directory;
}

sub my_int_handler {
  &clean_up();
  die "interrupted, exiting...\n";
}

$SIG{'INT'} = 'my_int_handler';
...;
  # some unspecified code here
  # Time passes, the program runs, creates some temporary
  # files in the temp directory, maybe someone presses Ctrl-C
...;
  # Now it's the end of normal execution
&clean_up();
```

NOTE

The `File::Temp` module, which comes with Perl, can automatically clean up temporary files and directories.

The assignment into the special `%SIG` hash activates the handler (until revoked). The key is the name of the signal (without the constant `SIG` prefix), and the value is a string naming the subroutine, without the ampersand. From then on, if a `SIGINT` comes along, Perl stops whatever it's doing and jumps immediately to the subroutine. Your subroutine cleans up the temp files and then exits. (And if nobody presses Ctrl-C, we'll still call `&clean_up()` at the end of normal execution.)

If the subroutine returns rather than exiting, execution resumes right where the signal interrupted it. This can be useful if the signal needs to actually interrupt something rather than causing it to stop. For example, suppose processing each line of a file takes a few seconds, which is pretty slow, and you want to abort the overall processing when an interrupt is processed—but not in the middle of processing a line. Just set a flag in the signal procedure and check it at the end of each line's processing:

```
my $int_$flag = 0;
$SIG{'INT'} = 'my_int_handler';
sub my_int_handler { $int_flag = 1; }

while( ... doing stuff .. ) {
  last if $int_flag;
  ...
}

exit();
```

For the most part, Perl will only handle a signal once it reaches a safe point to do so. For instance, Perl will not deliver most signals in the middle of allocating memory or rearranging its internal data structures. Perl delivers some signals, such as `SIGILL`, `SIGBUS`, and `SIGSEGV`, right away, so those are still unsafe. See the perlipc documentation.

Exercises

See "Answers to Chapter 15 Exercises" for answers to these exercises:

1. [6] Write a program that changes to some particular (hardcoded) directory, like the system's root directory, then executes the *ls -l* command to get a long-format directory listing in that directory. (If you use a non-Unix system, use your own system's command to get a detailed directory listing.)
2. [10] Modify the previous program to send the output of the command to a file called *ls.out* in the current directory. The error output should

go to a file called *ls.err*. (You don't need to do anything special about the fact that either of these files may end up being empty.)

3. [8] Write a program to parse the output of the *date* command to determine the current day of the week. If the day of the week is a weekday, print `get to work`; otherwise, print `go play`. The output of the *date* command begins with `Mon` on a Monday. If you don't have a *date* command on your non-Unix system, make a fake little program that simply prints a string like *date* might print. We'll even give you this two-line program if you promise not to ask us how it works:

```
#!/usr/bin/perl
print localtime( ) . "\n";
```

4. [15] (Unix only) Write an infinite loop program that catches signals and reports which signal it caught and how many times it has seen that signal before. Exit if you catch the `INT` signal. If you can use the command-line *kill*, you can send signals like so:

```
$ kill -USR1 12345
```

If you can't use the command-line *kill*, write another program to send signals to it. You might be able to get away with a Perl one-liner:


```
$ perl -e 'kill HUP =>  12345'
```