NAME

perlipc - Perl interprocess communication (signals, fifos, pipes, safe subprocesses, sockets, and semaphores)

DESCRIPTION

The basic IPC facilities of Perl are built out of the good old Unix signals, named pipes, pipe opens, the Berkeley socket routines, and SysV IPC calls. Each is used in slightly different situations.

Signals

Perl uses a simple signal handling model: the %SIG hash contains names or references of user-installed signal handlers. These handlers will be called with an argument which is the name of the signal that triggered it. A signal may be generated intentionally from a particular keyboard sequence like control-C or control-Z, sent to you from another process, or triggered automatically by the kernel when special events transpire, like a child process exiting, your own process running out of stack space, or hitting a process file-size limit.

For example, to trap an interrupt signal, set up a handler like this:

```
our $shucks;

sub catch_zap {
my $signame = shift;
$shucks++;
die "Somebody sent me a SIG$signame";
}
$SIG{INT} = __PACKAGE__ . "::catch_zap";
$SIG{INT} = \&catch_zap; # best strategy
```

Prior to Perl 5.8.0 it was necessary to do as little as you possibly could in your handler; notice how all we do is set a global variable and then raise an exception. That's because on most systems, libraries are not re-entrant; particularly, memory allocation and I/O routines are not. That meant that doing nearly *anything* in your handler could in theory trigger a memory fault and subsequent core dump - see ''Deferred Signals (Safe Signals)'' below.

The names of the signals are the ones listed out by `kill -l` on your system, or you can retrieve them using the CPAN module IPC::Signal.

You may also choose to assign the strings `"IGNORE"` or `"DEFAULT"` as the handler, in which case Perl will try to discard the signal or do the default thing.

On most Unix platforms, the `CHLD` (sometimes also known as `CLD`) signal has special behavior with respect to a value of `"IGNORE"`. Setting`$SIG{CHLD}` to `"IGNORE"` on such a platform has the ef fect of not creating zombie processes when the parent process fails to `wait()` on its child processes (i.e., child processes are automatically reaped). Calling `wait()` with `$SIG{CHLD}` set to `"IGNORE"` usually returns `-1` on such platforms.

Some signals can be neither trapped nor ignored, such as the KILL and STOP (but not the TSTP) signals. Note that ignoring signals makes them disappear. If you only want them blocked temporarily without them getting lost you'll have to use POSIX' sigprocmask.

Sending a signal to a negative process ID means that you send the signal to the entire Unix process group. This code sends a hang-up signal to all processes in the current process group, and also sets `$SIG`{HUP} to `"IGNORE"` so it doesn't kill itself:

```
# block scope for local
{
local $SIG{HUP} = "IGNORE";
kill HUP => -$$;
# snazzy writing of: kill("HUP", -$$)
}
```

Another interesting signal to send is signal number zero. This doesn't actually affect a child process, but

instead checks whether it's alive or has changed its UIDs.

```
unless (kill 0 => $kid_pid) {
warn "something wicked happened to $kid_pid";
}
```

Signal number zero may fail because you lack permission to send the signal when directed at a process whose real or saved UID is not identical to the real or effective UID of the sending process, even though the process is alive. You may be able to determine the cause of failure using `$!` or `%!`.

```
unless (kill(0 => $pid) || $!{EPERM}) {
warn "$pid looks dead";
}
```

You might also want to employ anonymous functions for simple signal handlers:

```
$SIG{INT} = sub { die "\nOutta here!\n" };
```

SIGCHLD handlers require some special care. If a second child dies while in the signal handler caused by the first death, we won't get another signal. So must loop here else we will leave the unreaped child as a zombie. And the next time two children die we get another zombie. And so on.

```
use POSIX ":sys_wait_h";
$SIG{CHLD} = sub {
while ((my $child = waitpid(-1, WNOHANG)) > 0) {
$Kid_Status{$child} = $?;
}
};
# do something that forks...
```

Be careful: *qx()*, *system()*, and some modules for calling external commands do a *fork()*, then *wait()* for the result. Thus, your signal handler will be called. Because *wait()* was already called by *system()* or *qx()*, the *wait()* in the signal handler will see no more zombies and will therefore block.

The best way to prevent this issue is to use *waitpid()*, as in the following example:

```
use POSIX ":sys_wait_h"; # for nonblocking read

my %children;

$SIG{CHLD} = sub {
# don't change $! and $? outside handler
local ($!, $?);
while ( (my $pid = waitpid(-1, WNOHANG)) > 0 ) {
delete $children{$pid};
cleanup_child($pid, $?);
}
};

while (1) {
my $pid = fork();
die "cannot fork" unless defined $pid;
if ($pid == 0) {
# ...
exit 0;
} else {
$children{$pid}=1;
# ...
system($command);
# ...
```

```
}
}
```

Signal handling is also used for timeouts in Unix. While safely protected within an `eval{}` block, you set a signal handler to trap alarm signals and then schedule to have one delivered to you in some number of seconds. Then try your blocking operation, clearing the alarm when it's done but not before you've exited your `eval{}` block. If it goes off, you'll use *die()* to jump out of the block.

Here's an example:

```
my $ALARM_EXCEPTION = "alarm clock restart";
eval {
local $SIG{ALRM} = sub { die $ALARM_EXCEPTION };
alarm 10;
flock(FH, 2) # blocking write lock
|| die "cannot flock: $!";
alarm 0;
};
if ($@ && $@ !~ quotemeta($ALARM_EXCEPTION)) { die }
```

If the operation being timed out is *system()* or *qx()*, this technique is liable to generate zombies. If this matters to you, you'll need to do your own *fork()* and *exec()*, and kill the errant child process.

For more complex signal handling, you might see the standard POSIX module. Lamentably, this is almost entirely undocumented, but the *ext/POSIX/t/sigaction.t* file from the Perl source distribution has some examples in it.

Handling the SIGHUP Signal in Daemons

A process that usually starts when the system boots and shuts down when the system is shut down is called a daemon (Disk And Execution MONitor). If a daemon process has a configuration file which is modified after the process has been started, there should be a way to tell that process to reread its configuration file without stopping the process. Many daemons provide this mechanism using a `SIGHUP` signal handler. When you want to tell the daemon to reread the file, simply send it the `SIGHUP` signal.

The following example implements a simple daemon, which restarts itself every time the `SIGHUP` signal is received. The actual code is located in the subroutine `code()`, which just prints some debugging info to show that it works; it should be replaced with the real code.

```
#!/usr/bin/perl

use strict;
use warnings;

use POSIX ();
use FindBin ();
use File::Basename ();
use File::Spec::Functions qw(catfile);

$| = 1;

# make the daemon cross-platform, so exec always calls the script
# itself with the right path, no matter how the script was invoked.
my $script = File::Basename::basename($0);
my $SELF = catfile($FindBin::Bin, $script);

# POSIX unmasks the sigprocmask properly
$SIG{HUP} = sub {
print "got SIGHUP\n";
exec($SELF, @ARGV) || die "$0: couldn't restart: $!";
```

```
};

code();

sub code {
print "PID: $$\n";
print "ARGV: @ARGV\n";
my $count = 0;
while (1) {
sleep 2;
print ++$count, "\n";
}
}
```

Deferred Signals (Safe Signals)

Before Perl 5.8.0, installing Perl code to deal with signals exposed you to danger from two things. First, few system library functions are re-entrant. If the signal interrupts while Perl is executing one function (like *malloc(3)* or *printf(3)*), and your signal handler then calls the same function again, you could get unpredictable behavior—often, a core dump. Second, Perl isn't itself re-entrant at the lowest levels. If the signal interrupts Perl while Perl is changing its own internal data structures, similarly unpredictable behavior may result.

There were two things you could do, knowing this: be paranoid or be pragmatic. The paranoid approach was to do as little as possible in your signal handler. Set an existing integer variable that already has a value, and return. This doesn't help you if you're in a slow system call, which will just restart. That means you have to `die` to *longjmp(3)* out of the handler. Even this is a little cavalier for the true paranoiac, who avoids `die` in a handler because the system *is* out to get you. The pragmatic approach was to say "I know the risks, but prefer the convenience", and to do anything you wanted in your signal handler, and be prepared to clean up core dumps now and again.

Perl 5.8.0 and later avoid these problems by "deferring" signals. That is, when the signal is delivered to the process by the system (to the C code that implements Perl) a flag is set, and the handler returns immediately. Then at strategic "safe" points in the Perl interpreter (e.g. when it is about to execute a new opcode) the flags are checked and the Perl level handler from `%SIG` is executed. The "deferred" scheme allows much more flexibility in the coding of signal handlers as we know the Perl interpreter is in a safe state, and that we are not in a system library function when the handler is called. However the implementation does differ from previous Perls in the following ways:

Long-running opcodes

As the Perl interpreter looks at signal flags only when it is about to execute a new opcode, a signal that arrives during a long-running opcode (e.g. a regular expression operation on a very large string) will not be seen until the current opcode completes.

If a signal of any given type fires multiple times during an opcode (such as from a fine-grained timer), the handler for that signal will be called only once, after the opcode completes; all other instances will be discarded. Furthermore, if your system's signal queue gets flooded to the point that there are signals that have been raised but not yet caught (and thus not deferred) at the time an opcode completes, those signals may well be caught and deferred during subsequent opcodes, with sometimes surprising results. For example, you may see alarms delivered even after calling `alarm(0)` as the latter stops the raising of alarms but does not cancel the delivery of alarms raised but not yet caught. Do not depend on the behaviors described in this paragraph as they are side effects of the current implementation and may change in future versions of Perl.

Interrupting IO

When a signal is delivered (e.g., SIGINT from a control-C) the operating system breaks into IO operations like *read(2)*, which is used to implement Perl's *readline()* function, the `<>` operator. On older Perls the handler was called immediately (and as `read` is not "unsafe", this worked well). With the "deferred" scheme the handler is *not* called immediately, and if Perl is using the system's `stdio`

library that library may restart the `read` without returning to Perl to give it a chance to call the `%SIG` handler. If this happens on your system the solution is to use the `:perlio` layer to do IO — at least on those handles that you want to be able to break into with signals. (The `:perlio` layer checks the signal flags and calls `%SIG` handlers before resuming IO operation.)

The default in Perl 5.8.0 and later is to automatically use the `:perlio` layer.

Note that it is not advisable to access a file handle within a signal handler where that signal has interrupted an I/O operation on that same handle. While perl will at least try hard not to crash, there are no guarantees of data integrity; for example, some data might get dropped or written twice.

Some networking library functions like *gethostbyname()* are known to have their own implementations of timeouts which may conflict with your timeouts. If you have problems with such functions, try using the POSIX *sigaction()* function, which bypasses Perl safe signals. Be warned that this does subject you to possible memory corruption, as described above.

Instead of setting `$SIG{ALRM}`:

```
local $SIG{ALRM} = sub { die "alarm" };
```

try something like the following:

```
use POSIX qw(SIGALRM);
POSIX::sigaction(SIGALRM,
POSIX::SigAction->new(sub { die "alarm" }))
|| die "Error setting SIGALRM handler: $!\n";
```

Another way to disable the safe signal behavior locally is to use the `Perl::Unsafe::Signals` module from CPAN, which affects all signals.

Restartable system calls
: On systems that supported it, older versions of Perl used the SA_RESTART flag when installing `%SIG` handlers. This meant that restartable system calls would continue rather than returning when a signal arrived. In order to deliver deferred signals promptly, Perl 5.8.0 and later do *not* use SA_RESTART. Consequently, restartable system calls can fail (with $! set to `EINTR`) in places where they previously would have succeeded.

  The default `:perlio` layer retries `read`, `write` and `close` as described above; interrupted `wait` and `waitpid` calls will always be retried.

Signals as ‘‘faults’’
: Certain signals like SEGV, ILL, and BUS are generated by virtual memory addressing errors and similar ‘‘faults’’. These are normally fatal: there is little a Perl-level handler can do with them. So Perl delivers them immediately rather than attempting to defer them.

Signals triggered by operating system state
: On some operating systems certain signal handlers are supposed to ‘‘do something’’ before returning. One example can be CHLD or CLD, which indicates a child process has completed. On some operating systems the signal handler is expected to `wait` for the completed child process. On such systems the deferred signal scheme will not work for those signals: it does not do the `wait`. Again the failure will look like a loop as the operating system will reissue the signal because there are completed child processes that have not yet been `wait`ed for.

If you want the old signal behavior back despite possible memory corruption, set the environment variable `PERL_SIGNALS` to `"unsafe"`. This feature first appeared in Perl 5.8.1.

Named Pipes

A named pipe (often referred to as a FIFO) is an old Unix IPC mechanism for processes communicating on the same machine. It works just like regular anonymous pipes, except that the processes rendezvous using a filename and need not be related.

To create a named pipe, use the `POSIX::mkfifo()` function.

```
use POSIX qw(mkfifo);
mkfifo($path, 0700) || die "mkfifo $path failed: $!";
```

You can also use the Unix command *mknod(1)*, or on some systems, *mkfifo(1)*. These may not be in your normal path, though.

```
# system return val is backwards, so && not ||
#
$ENV{PATH} .= ":/etc:/usr/etc";
if ( system("mknod", $path, "p")
&& system("mkfifo", $path) )
{
die "mk{nod,fifo} $path failed";
}
```

A fifo is convenient when you want to connect a process to an unrelated one. When you open a fifo, the program will block until there's something on the other end.

For example, let's say you'd like to have your *.signature* file be a named pipe that has a Perl program on the other end. Now every time any program (like a mailer, news reader, finger program, etc.) tries to read from that file, the reading program will read the new signature from your program. We'll use the pipe-checking file-test operator, **-p**, to find out whether anyone (or anything) has accidentally removed our fifo.

```
chdir(); # go home
my $FIFO = ".signature";

while (1) {
unless (-p $FIFO) {
unlink $FIFO; # discard any failure, will catch later
require POSIX; # delayed loading of heavy module
POSIX::mkfifo($FIFO, 0700)
|| die "can't mkfifo $FIFO: $!";
}

# next line blocks till there's a reader
open (FIFO, "> $FIFO") || die "can't open $FIFO: $!";
print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
close(FIFO) || die "can't close $FIFO: $!";
sleep 2; # to avoid dup signals
}
```

Using *open()* for IPC

Perl's basic *open()* statement can also be used for unidirectional interprocess communication by either appending or prepending a pipe symbol to the second argument to *open()*. Here's how to start something up in a child process you intend to write to:

```
open(SPOOLER, "| cat -v | lpr -h 2>/dev/null")
|| die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER || die "bad spool: $! $?";
```

And here's how to start up a child process you intend to read from:

```
open(STATUS, "netstat -an 2>&1 |")
|| die "can't fork: $!";
while (<STATUS>) {
next if /^(tcp|udp)/;
print;
}
close STATUS || die "bad netstat: $! $?";
```

If one can be sure that a particular program is a Perl script expecting filenames in `@ARGV`, the clever programmer can write something like this:

```
% program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

and no matter which sort of shell it's called from, the Perl program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file. Pretty nifty, eh?

You might notice that you could use backticks for much the same effect as opening a pipe for reading:

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstatus ($?)" if $?;
```

While this is true on the surface, it's much more efficient to process the file one line or record at a time because then you don't have to read the whole thing into memory at once. It also gives you finer control of the whole process, letting you kill off the child process early if you'd like.

Be careful to check the return values from both *open()* and *close()*. If you're *writing* to a pipe, you should also trap SIGPIPE. Otherwise, think of what happens when you start up a pipe to a command that doesn't exist: the *open()* will in all likelihood succeed (it only reflects the *fork()*'s success), but then your output will fail — spectacularly. Perl can't know whether the command worked, because your command is actually running in a separate process whose *exec()* might have failed. Therefore, while readers of bogus commands return just a quick EOF, writers to bogus commands will get hit with a signal, which they'd best be prepared to handle. Consider:

```
open(FH, "|bogus") || die "can't fork: $!";
print FH "bang\n"; # neither necessary nor sufficient
# to check print retval!
close(FH) || die "can't close: $!";
```

The reason for not checking the return value from *print()* is because of pipe buffering; physical writes are delayed. That won't blow up until the close, and it will blow up with a SIGPIPE. To catch it, you could use this:

```
$SIG{PIPE} = "IGNORE";
open(FH, "|bogus") || die "can't fork: $!";
print FH "bang\n";
close(FH) || die "can't close: status=$?";
```

Filehandles

Both the main process and any child processes it forks share the same STDIN, STDOUT, and STDERR filehandles. If both processes try to access them at once, strange things can happen. You may also want to close or reopen the filehandles for the child. You can get around this by opening your pipe with *open()*, but on some systems this means that the child process cannot outlive the parent.

Background Processes

You can run a command in the background with:

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

The command's STDOUT and STDERR (and possibly STDIN, depending on your shell) will be the same as the parent's. You won't need to catch SIGCHLD because of the double-fork taking place; see below for details.

Complete Dissociation of Child from Parent

In some cases (starting server processes, for instance) you'll want to completely dissociate the child process from the parent. This is often called daemonization. A well-behaved daemon will also *chdir()* to the root directory so it doesn't prevent unmounting the filesystem containing the directory from which it was launched, and redirect its standard file descriptors from and to */dev/null* so that random output doesn't wind up on the user's terminal.

```
use POSIX "setsid";

sub daemonize {
chdir("/") || die "can't chdir to /: $!";
open(STDIN, "< /dev/null") || die "can't read /dev/null: $!";
open(STDOUT, "> /dev/null") || die "can't write to /dev/null: $!";
defined(my $pid = fork()) || die "can't fork: $!";
exit if $pid; # non-zero now means I am the parent
(setsid() != -1) || die "Can't start a new session: $!";
open(STDERR, ">&STDOUT") || die "can't dup stdout: $!";
}
```

The *fork()* has to come before the *setsid()* to ensure you aren't a process group leader; the *setsid()* will fail if you are. If your system doesn't have the *setsid()* function, open */dev/tty* and use the `TIOCNOTTY` *ioctl()* on it instead. See *tty(4)* for details.

Non-Unix users should check their *Your_OS*`::Process` module for other possible solutions.

Safe Pipe Opens

Another interesting approach to IPC is making your single program go multiprocess and communicate between — or even amongst — yourselves. The *open()* function will accept a file argument of either `"-|"` or `"|-"` to do a very interesting thing: it forks a child connected to the filehandle you've opened. The child is running the same program as the parent. This is useful for safely opening a file when running under an assumed UID or GID, for example. If you open a pipe *to* minus, you can write to the filehandle you opened and your kid will find it in *his* STDIN. If you open a pipe*fr om* minus, you can read from the filehandle you opened whatever your kid writes to *his* STDOUT.

```
use English;
my $PRECIOUS = "/path/to/some/safe/file";
my $sleep_count;
my $pid;

do {
$pid = open(KID_TO_WRITE, "|-");
unless (defined $pid) {
warn "cannot fork: $!";
die "bailing out" if $sleep_count++ > 6;
sleep 10;
}
} until defined $pid;

if ($pid) { # I am the parent
print KID_TO_WRITE @some_data;
close(KID_TO_WRITE) || warn "kid exited $?";
} else { # I am the child
# drop permissions in setuid and/or setgid programs:
($EUID, $EGID) = ($UID, $GID);
open (OUTFILE, "> $PRECIOUS")
|| die "can't open $PRECIOUS: $!";
while (<STDIN>) {
```

```
print OUTFILE; # child's STDIN is parent's KID_TO_WRITE
}
close(OUTFILE) || die "can't close $PRECIOUS: $!";
exit(0) # don't forget this!!
}
```

Another common use for this construct is when you need to execute something without the shell's interference. With *system()*, it's straightforward, but you can't use a pipe open or backticks safely. That's because there's no way to stop the shell from getting its hands on your arguments. Instead, use lower-level control to call *exec()* directly.

Here's a safe backtick or pipe open for read:

```
my $pid = open(KID_TO_READ, "-|");
defined($pid) || die "can't fork: $!";

if ($pid) { # parent
while (<KID_TO_READ>) {
# do something interesting
}
close(KID_TO_READ) || warn "kid exited $?";

} else { # child
($EUID, $EGID) = ($UID, $GID); # suid only
exec($program, @options, @args)
|| die "can't exec program: $!";
# NOTREACHED
}
```

And here's a safe pipe open for writing:

```
my $pid = open(KID_TO_WRITE, "|-");
defined($pid) || die "can't fork: $!";

$SIG{PIPE} = sub { die "whoops, $program pipe broke" };

if ($pid) { # parent
print KID_TO_WRITE @data;
close(KID_TO_WRITE) || warn "kid exited $?";

} else { # child
($EUID, $EGID) = ($UID, $GID);
exec($program, @options, @args)
|| die "can't exec program: $!";
# NOTREACHED
}
```

It is very easy to dead-lock a process using this form of *open()*, or indeed with any use of *pipe()* with multiple subprocesses. The example above is ''safe'' because it is simple and calls *exec()*. See ''Avoiding Pipe Deadlocks'' for general safety principles, but there are extra gotchas with Safe Pipe Opens.

In particular, if you opened the pipe using `open FH, "|-"`, then you cannot simply use *close()* in the parent process to close an unwanted writer. Consider this code:

```
my $pid = open(WRITER, "|-"); # fork open a kid
defined($pid) || die "first fork failed: $!";
if ($pid) {
if (my $sub_pid = fork()) {
defined($sub_pid) || die "second fork failed: $!";
close(WRITER) || die "couldn't close WRITER: $!";
# now do something else...
}
else {
# first write to WRITER
# ...
# then when finished
close(WRITER) || die "couldn't close WRITER: $!";
exit(0)
}
}
else {
# first do something with STDIN, then
exit(0)
}
```

In the example above, the true parent does not want to write to the WRITER filehandle, so it closes it. However, because WRITER was opened using `open FH, "|-"`, it has a special behavior: closing it calls *waitpid()* (see ‘‘waitpid’’ in perlfunc), which waits for the subprocess to exit. If the child process ends up waiting for something happening in the section marked ‘‘do something else’’, you have deadlock.

This can also be a problem with intermediate subprocesses in more complicated code, which will call *waitpid()* on all open filehandles during global destruction — in no predictable order.

To solve this, you must manually use *pipe()*, *fork()*, and the form of *open()* which sets one file descriptor to another, as shown below:

```
pipe(READER, WRITER) || die "pipe failed: $!";
$pid = fork();
defined($pid) || die "first fork failed: $!";
if ($pid) {
close READER;
if (my $sub_pid = fork()) {
defined($sub_pid) || die "first fork failed: $!";
close(WRITER) || die "can't close WRITER: $!";
}
else {
# write to WRITER...
# ...
# then when finished
close(WRITER) || die "can't close WRITER: $!";
exit(0)
}
# write to WRITER...
}
else {
open(STDIN, "<&READER") || die "can't reopen STDIN: $!";
close(WRITER) || die "can't close WRITER: $!";
# do something...
exit(0)
}
```

Since Perl 5.8.0, you can also use the list form of `open` for pipes. This is preferred when you wish to avoid having the shell interpret metacharacters that may be in your command string.

So for example, instead of using:

```
open(PS_PIPE, "ps aux|") || die "can't open ps pipe: $!";
```

One would use either of these:

```
open(PS_PIPE, "-|", "ps", "aux")
|| die "can't open ps pipe: $!";

@ps_args = qw[ ps aux ];
open(PS_PIPE, "-|", @ps_args)
|| die "can't open @ps_args|: $!";
```

Because there are more than three arguments to *open()*, forks the *ps(1)* command*without* spa wning a shell, and reads its standard output via the `PS_PIPE` filehandle. The corresponding syntax to *write* to command pipes is to use `"|-"` in place of `"-|"`.

This was admittedly a rather silly example, because you're using string literals whose content is perfectly safe. There is therefore no cause to resort to the harder-to-read, multi-argument form of pipe *open()*. However, whenever you cannot be assured that the program arguments are free of shell metacharacters, the fancier form of *open()* should be used. For example:

```
@grep_args = ("egrep", "-i", $some_pattern, @many_files);
open(GREP_PIPE, "-|", @grep_args)
|| die "can't open @grep_args|: $!";
```

Here the multi-argument form of pipe *open()* is preferred because the pattern and indeed even the filenames themselves might hold metacharacters.

Be aware that these operations are full Unix forks, which means they may not be correctly implemented on all alien systems.

Avoiding Pipe Deadlocks

Whenever you have more than one subprocess, you must be careful that each closes whichever half of any pipes created for interprocess communication it is not using. This is because any child process reading from the pipe and expecting an EOF will never receive it, and therefore never exit. A single process closing a pipe is not enough to close it; the last process with the pipe open must close it for it to read EOF.

Certain built-in Unix features help prevent this most of the time. For instance, filehandles have a ''close on exec'' flag, which is set *en masse* under control of the `$^F` variable. This is so any filehandles you didn't explicitly route to the STDIN, STDOUT or STDERR of a child *program* will be automatically closed.

Always explicitly and immediately call *close()* on the writable end of any pipe, unless that process is actually writing to it. Even if you don't explicitly call *close()*, Perl will still *close()* all filehandles during global destruction. As previously discussed, if those filehandles have been opened with Safe Pipe Open, this will result in calling *waitpid()*, which may again deadlock.

Bidirectional Communication with Another Process

While this works reasonably well for unidirectional communication, what about bidirectional communication? The most obvious approach doesn't work:

```
# THIS DOES NOT WORK!!
open(PROG_FOR_READING_AND_WRITING, "| some program |")
```

If you forget to `use warnings`, you'll miss out entirely on the helpful diagnostic message:

```
Can't do bidirectional pipe at -e line 1.
```

If you really want to, you can use the standard *open2()* from the `IPC::Open2` module to catch both ends. There's also an *open3()* in `IPC::Open3` for tridirectional I/O so you can also catch your child's STDERR, but doing so would then require an awkward *select()* loop and wouldn't allow you to use normal Perl input operations.

If you look at its source, you'll see that *open2()* uses low-level primitives like the *pipe()* and *exec()* syscalls to create all the connections. Although it might have been more efficient by using *socketpair()*, this would have been even less portable than it already is. The *open2()* and *open3()* functions are unlikely to work anywhere except on a Unix system, or at least one purporting POSIX compliance.

Here's an example of using *open2()*:

```
use FileHandle;
use IPC::Open2;
$pid = open2(*Reader, *Writer, "cat -un");
print Writer "stuff\n";
$got = <Reader>;
```

The problem with this is that buffering is really going to ruin your day. Even though your `Writer` filehandle is auto-flushed so the process on the other end gets your data in a timely manner, you can't usually do anything to force that process to give its data to you in a similarly quick fashion. In this special case, we could actually so, because we gave *cat* a **-u** flag to make it unbuffered. But very few commands are designed to operate over pipes, so this seldom works unless you yourself wrote the program on the other end of the double-ended pipe.

A solution to this is to use a library which uses pseudottys to make your program behave more reasonably. This way you don't have to have control over the source code of the program you're using. The `Expect` module from CPAN also addresses this kind of thing. This module requires two other modules from CPAN, `IO::Pty` and `IO::Stty` It sets up a pseudo terminal to interact with programs that insist on talking to the terminal device driver. If your system is supported, this may be your best bet.

Bidirectional Communication with Yourself

If you want, you may make low-level *pipe()* and *fork()* syscalls to stitch this together by hand. This example only talks to itself, but you could reopen the appropriate handles to STDIN and STDOUT and call other processes. (The following example lacks proper error checking.)

```
#!/usr/bin/perl -w
# pipe1 - bidirectional communication using two pipe pairs
# designed for the socketpair-challenged
use IO::Handle; # thousands of lines just for autoflush :-(
pipe(PARENT_RDR, CHILD_WTR); # XXX: check failure?
pipe(CHILD_RDR, PARENT_WTR); # XXX: check failure?
CHILD_WTR->autoflush(1);
PARENT_WTR->autoflush(1);

if ($pid = fork()) {
close PARENT_RDR;
close PARENT_WTR;
print CHILD_WTR "Parent Pid $$ is sending this\n";
chomp($line = <CHILD_RDR>);
print "Parent Pid $$ just read this: '$line'\n";
close CHILD_RDR; close CHILD_WTR;
waitpid($pid, 0);
} else {
die "cannot fork: $!" unless defined $pid;
close CHILD_RDR;
close CHILD_WTR;
chomp($line = <PARENT_RDR>);
print "Child Pid $$ just read this: '$line'\n";
print PARENT_WTR "Child Pid $$ is sending this\n";
close PARENT_RDR;
close PARENT_WTR;
exit(0)
```

```
}
```

But you don't actually have to make two pipe calls. If you have the *socketpair()* system call, it will do this all for you.

```
#!/usr/bin/perl -w
# pipe2 - bidirectional communication using socketpair
# "the best ones always go both ways"

use Socket;
use IO::Handle; # thousands of lines just for autoflush :-(

# We say AF_UNIX because although *_LOCAL is the
# POSIX 1003.1g form of the constant, many machines
# still don't have it.
socketpair(CHILD, PARENT, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
|| die "socketpair: $!";

CHILD->autoflush(1);
PARENT->autoflush(1);

if ($pid = fork()) {
close PARENT;
print CHILD "Parent Pid $$ is sending this\n";
chomp($line = <CHILD>);
print "Parent Pid $$ just read this: '$line'\n";
close CHILD;
waitpid($pid, 0);
} else {
die "cannot fork: $!" unless defined $pid;
close CHILD;
chomp($line = <PARENT>);
print "Child Pid $$ just read this: '$line'\n";
print PARENT "Child Pid $$ is sending this\n";
close PARENT;
exit(0)
}
```

Sockets: Client/Server Communication

While not entirely limited to Unix-derived operating systems (e.g., WinSock on PCs provides socket support, as do some VMS libraries), you might not have sockets on your system, in which case this section probably isn't going to do you much good. With sockets, you can do both virtual circuits like TCP streams and datagrams like UDP packets. You may be able to do even more depending on your system.

The Perl functions for dealing with sockets have the same names as the corresponding system calls in C, but their arguments tend to differ for two reasons. First, Perl filehandles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don't need to pass that information.

One of the major problems with ancient, antemillennial socket code in Perl was that it used hard-coded values for some of the constants, which severely hurt portability. If you ever see code that does anything like explicitly setting `$AF_INET = 2`, you know you're in for big trouble. An immeasurably superior approach is to use the `Socket` module, which more reliably grants access to the various constants and functions you'll need.

If you're not writing a server/client for an existing protocol like NNTP or SMTP, you should give some thought to how your server will know when the client has finished talking, and vice-versa. Most protocols are based on one-line messages and responses (so one party knows the other has finished when a "\n" is received) or multi-line messages and responses that end with a period on an empty line ("\n.\n" terminates

a message/response).

Internet Line Terminators

The Internet line terminator is "\015\012". Under ASCII variants of Unix, that could usually be written as "\r\n", but under other systems, "\r\n" might at times be "\015\015\012", "\012\012\015", or something completely different. The standards specify writing "\015\012" to be conformant (be strict in what you provide), but they also recommend accepting a lone "\012" on input (be lenient in what you require). We haven't always been very good about that in the code in this manpage, but unless you're on a Mac from way back in its pre-Unix dark ages, you'll probably be ok.

Internet TCP Clients and Servers

Use Internet-domain sockets when you want to do client-server communication that might extend to machines outside of your own system.

Here's a sample TCP client using Internet-domain sockets:

```
#!/usr/bin/perl -w
use strict;
use Socket;
my ($remote, $port, $iaddr, $paddr, $proto, $line);

$remote = shift || "localhost";
$port = shift || 2345; # random port
if ($port =~ /\D/) { $port = getservbyname($port, "tcp") }
die "No port" unless $port;
$iaddr = inet_aton($remote) || die "no host: $remote";
$paddr = sockaddr_in($port, $iaddr);

$proto = getprotobyname("tcp");
socket(SOCK, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
connect(SOCK, $paddr) || die "connect: $!";
while ($line = <SOCK>) {
print $line;
}

close (SOCK) || die "close: $!";
exit(0)
```

And here's a corresponding server to go along with it. We'll leave the address as INADDR_ANY so that the kernel can choose the appropriate interface on multihomed hosts. If you want sit on a particular interface (like the external side of a gateway or firewall machine), fill this in with your real address instead.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = "/usr/bin:/bin" }
use Socket;
use Carp;
my $EOL = "\015\012";

sub logmsg { print "$0 $$: @_ at ", scalar localtime(), "\n" }

my $port = shift || 2345;
die "invalid port" unless $port =~ /^ \d+ $/x;

my $proto = getprotobyname("tcp");

socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
```

```
|| die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on port $port";

my $paddr;

for ( ; $paddr = accept(Client, Server); close Client) {
my($port, $iaddr) = sockaddr_in($paddr);
my $name = gethostbyaddr($iaddr, AF_INET);

logmsg "connection from $name [",
inet_ntoa($iaddr), "]
at port $port";

print Client "Hello there, $name, it's now ",
scalar localtime(), $EOL;
}
```

And here's a multitasking version. It's multitasked in that like most typical servers, it spawns (*fork()*s) a slave server to handle the client request so that the master server can quickly go back to service a new client.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = "/usr/bin:/bin" }
use Socket;
use Carp;
my $EOL = "\015\012";

sub spawn; # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime(), "\n" }

my $port = shift || 2345;
die "invalid port" unless $port =~ /^ \d+ $/x;

my $proto = getprotobyname("tcp");

socket(Server, PF_INET, SOCK_STREAM, $proto) || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
|| die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

use POSIX ":sys_wait_h";
use Errno;

sub REAPER {
local $!; # don't let waitpid() overwrite current error
while ((my $pid = waitpid(-1, WNOHANG)) > 0 && WIFEXITED($?)) {
```

```
logmsg "reaped $waitedpid" . ($? ? " with exit $?" : "");
}
$SIG{CHLD} = \&REAPER; # loathe SysV
}

$SIG{CHLD} = \&REAPER;

while (1) {
$paddr = accept(Client, Server) || do {
# try again if accept() returned because got a signal
next if $!{EINTR};
die "accept: $!";
};
my ($port, $iaddr) = sockaddr_in($paddr);
my $name = gethostbyaddr($iaddr, AF_INET);

logmsg "connection from $name [",
inet_ntoa($iaddr),
"] at port $port";

spawn sub {
$| = 1;
print "Hello there, $name, it's now ",
scalar localtime(),
$EOL;
exec "/usr/games/fortune" # XXX: "wrong" line terminators
or confess "can't exec fortune: $!";
};
close Client;
}

sub spawn {
my $coderef = shift;

unless (@_ == 0 && $coderef && ref($coderef) eq "CODE") {
confess "usage: spawn CODEREF";
}

my $pid;
unless (defined($pid = fork())) {
logmsg "cannot fork: $!";
return;
}
elsif ($pid) {
logmsg "begat $pid";
return; # I'm the parent
}
# else I'm the child -- go spawn

open(STDIN, "<&Client") || die "can't dup client to stdin";
open(STDOUT, ">&Client") || die "can't dup client to stdout";
## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
exit($coderef->());
}
```

This server takes the trouble to clone off a child version via *fork()* for each incoming request. That way it can handle many requests at once, which you might not always want. Even if you don't *fork()*, the *listen()* will allow that many pending connections. Forking servers have to be particularly careful about cleaning up their dead children (called ''zombies'' in Unix parlance), because otherwise you'll quickly fill up your process table. The REAPER subroutine is used here to call *waitpid()* for any child processes that have finished, thereby ensuring that they terminate cleanly and don't join the ranks of the living dead.

Within the while loop we call *accept()* and check to see if it returns a false value. This would normally indicate a system error needs to be reported. However, the introduction of safe signals (see ''Deferred Signals (Safe Signals)'' above) in Perl 5.8.0 means that *accept()* might also be interrupted when the process receives a signal. This typically happens when one of the forked subprocesses exits and notifies the parent process with a CHLD signal.

If *accept()* is interrupted by a signal, $! will be set to EINTR. If this happens, we can safely continue to the next iteration of the loop and another call to *accept()*. It is important that your signal handling code not modify the value of $!, or else this test will likely fail. In the REAPER subroutine we create a local version of $! before calling *waitpid()*. When *waitpid()* sets $! to ECHILD as it inevitably does when it has no more children waiting, it updates the local copy and leaves the original unchanged.

You should use the **-T** flag to enable taint checking (see perlsec) even if we aren't running setuid or setgid. This is always a good idea for servers or any program run on behalf of someone else (like CGI scripts), because it lessens the chances that people from the outside will be able to compromise your system.

Let's look at another TCP client. This one connects to the TCP ''time'' service on a number of different machines and shows how far their clocks differ from the system on which it's being run:

```
#!/usr/bin/perl -w
use strict;
use Socket;

my $SECS_OF_70_YEARS = 2208988800;
sub ctime { scalar localtime(shift() || time()) }

my $iaddr = gethostbyname("localhost");
my $proto = getprotobyname("tcp");
my $port = getservbyname("time", "tcp");
my $paddr = sockaddr_in(0, $iaddr);
my($host);

$| = 1;
printf "%-24s %8s %s\n", "localhost", 0, ctime();

foreach $host (@ARGV) {
printf "%-24s ", $host;
my $hisiaddr = inet_aton($host) || die "unknown host";
my $hispaddr = sockaddr_in($port, $hisiaddr);
socket(SOCKET, PF_INET, SOCK_STREAM, $proto)
|| die "socket: $!";
connect(SOCKET, $hispaddr) || die "connect: $!";
my $rtime = pack("C4", ());
read(SOCKET, $rtime, 4);
close(SOCKET);
my $histime = unpack("N", $rtime) - $SECS_OF_70_YEARS;
printf "%8d %s\n", $histime - time(), ctime($histime);
}
```

Unix-Domain TCP Clients and Servers

That's fine for Internet-domain clients and servers, but what about local communications? While you can use the same setup, sometimes you don't want to. Unix-domain sockets are local to the current host, and are often used internally to implement pipes. Unlike Internet domain sockets, Unix domain sockets can show up in the file system with an *ls(1)* listing.

```
% ls -l /dev/log
srw-rw-rw- 1 root 0 Oct 31 07:23 /dev/log
```

You can test for these with Perl's **-S** file test:

```
unless (-S "/dev/log") {
die "something's wicked with the log system";
}
```

Here's a sample Unix-domain client:

```
#!/usr/bin/perl -w
use Socket;
use strict;
my ($rendezvous, $line);

$rendezvous = shift || "catsock";
socket(SOCK, PF_UNIX, SOCK_STREAM, 0) || die "socket: $!";
connect(SOCK, sockaddr_un($rendezvous)) || die "connect: $!";
while (defined($line = <SOCK>)) {
print $line;
}
exit(0)
```

And here's a corresponding server. You don't have to worry about silly network terminators here because Unix domain sockets are guaranteed to be on the localhost, and thus everything works right.

```
#!/usr/bin/perl -Tw
use strict;
use Socket;
use Carp;

BEGIN { $ENV{PATH} = "/usr/bin:/bin" }
sub spawn; # forward declaration
sub logmsg { print "$0 $$: @_ at ", scalar localtime(), "\n" }

my $NAME = "catsock";
my $uaddr = sockaddr_un($NAME);
my $proto = getprotobyname("tcp");

socket(Server, PF_UNIX, SOCK_STREAM, 0) || die "socket: $!";
unlink($NAME);
bind (Server, $uaddr) || die "bind: $!";
listen(Server, SOMAXCONN) || die "listen: $!";

logmsg "server started on $NAME";

my $waitedpid;

use POSIX ":sys_wait_h";
sub REAPER {
my $child;
while (($waitedpid = waitpid(-1, WNOHANG)) > 0) {
```

```
logmsg "reaped $waitedpid" . ($? ? " with exit $?" : "");
}
$SIG{CHLD} = \&REAPER; # loathe SysV
}

$SIG{CHLD} = \&REAPER;


for ( $waitedpid = 0;
accept(Client, Server) || $waitedpid;
$waitedpid = 0, close Client)
{
next if $waitedpid;
logmsg "connection on $NAME";
spawn sub {
print "Hello there, it's now ", scalar localtime(), "\n";
exec("/usr/games/fortune") || die "can't exec fortune: $!";
};
}

sub spawn {
my $coderef = shift();

unless (@_ == 0 && $coderef && ref($coderef) eq "CODE") {
confess "usage: spawn CODEREF";
}

my $pid;
unless (defined($pid = fork())) {
logmsg "cannot fork: $!";
return;
}
elsif ($pid) {
logmsg "begat $pid";
return; # I'm the parent
}
else {
# I'm the child -- go spawn
}

open(STDIN, "<&Client") || die "can't dup client to stdin";
open(STDOUT, ">&Client") || die "can't dup client to stdout";
## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
exit($coderef->());
}
```

As you see, it's remarkably similar to the Internet domain TCP server, so much so, in fact, that we've omitted several duplicate functions—*spawn()*, *logmsg()*, *ctime()*, and *REAPER()*--which are the same as in the other server.

So why would you ever want to use a Unix domain socket instead of a simpler named pipe? Because a named pipe doesn't give you sessions. You can't tell one process's data from another's. With socket programming, you get a separate session for each client; that's why *accept()* takes two arguments.

For example, let's say that you have a long-running database server daemon that you want folks to be able to access from the Web, but only if they go through a CGI interface. You'd have a small, simple CGI

program that does whatever checks and logging you feel like, and then acts as a Unix-domain client and connects to your private server.

TCP Clients with IO::Socket

For those preferring a higher-level interface to socket programming, the IO::Socket module provides an object-oriented approach. If for some reason you lack this module, you can just fetch IO::Socket from CPAN, where you'll also find modules providing easy interfaces to the following systems: DNS, FTP, Ident (RFC 931), NIS and NISPlus, NNTP, Ping, POP3, SMTP, SNMP, SSLeay, Telnet, and Time — to name just a few.

A Simple Client

Here's a client that creates a TCP connection to the ''daytime'' service at port 13 of the host name ''localhost'' and prints out everything that the server there cares to provide.

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
Proto => "tcp",
PeerAddr => "localhost",
PeerPort => "daytime(13)",
)
|| die "can't connect to daytime service on localhost";
while (<$remote>) { print }
```

When you run this program, you should get something back that looks like this:

```
Wed May 14 08:40:46 MDT 1997
```

Here are what those parameters to the *new()* constructor mean:

`Proto`
: This is which protocol to use. In this case, the socket handle returned will be connected to a TCP socket, because we want a stream-oriented connection, that is, one that acts pretty much like a plain old file. Not all sockets are this of this type. For example, the UDP protocol can be used to make a datagram socket, used for message-passing.

`PeerAddr`
: This is the name or Internet address of the remote host the server is running on. We could have specified a longer name like `"www.perl.com"`, or an address like `"207.171.7.72"`. For demonstration purposes, we've used the special hostname `"localhost"`, which should always mean the current machine you're running on. The corresponding Internet address for localhost is `"127.0.0.1"`, if you'd rather use that.

`PeerPort`
: This is the service name or port number we'd like to connect to. We could have gotten away with using just `"daytime"` on systems with a well-configured system services file,[FOOTNOTE: The system services file is found in */etc/services* under Unixy systems.] but here we've specified the port number (13) in parentheses. Using just the number would have also worked, but numeric literals make careful programmers nervous.

Notice how the return value from the `new` constructor is used as a filehandle in the `while` loop? That's what's called an *indirect filehandle*, a scalar variable containing a filehandle. You can use it the same way you would a normal filehandle. For example, you can read one line from it this way:

```
$line = <$handle>;
```

all remaining lines from is this way:

```
@lines = <$handle>;
```

and send a line of data to it this way:

```
print $handle "some data\n";
```

A Webget Client

Here's a simple client that takes a remote host to fetch a document from, and then a list of files to get from that host. This is a more interesting client than the previous one because it first sends something to the server before fetching the server's response.

```
#!/usr/bin/perl -w
use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host url ..." }
$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;
for my $document (@ARGV) {
$remote = IO::Socket::INET->new( Proto => "tcp",
PeerAddr => $host,
PeerPort => "http(80)",
) || die "cannot connect to httpd on $host";
$remote->autoflush(1);
print $remote "GET $document HTTP/1.0" . $BLANK;
while ( <$remote> ) { print }
close $remote;
}
```

The web server handling the HTTP service is assumed to be at its standard port, number 80. If the server you're trying to connect to is at a different port, like 1080 or 8080, you should specify it as the named-parameter pair, `PeerPort => 8080`. The `autoflush` method is used on the socket because otherwise the system would buffer up the output we sent it. (If you're on a prehistoric Mac, you'll also need to change every `"\n"` in your code that sends data over the network to be a `"\015\012"` instead.)

Connecting to the server is only the first part of the process: once you have the connection, you have to use the server's language. Each server on the network has its own little command language that it expects as input. The string that we send to the server starting with "GET" is in HTTP syntax. In this case, we simply request each specified document. Yes, we really are making a new connection for each document, even though it's the same host. That's the way you always used to have to speak HTTP. Recent versions of web browsers may request that the remote server leave the connection open a little while, but the server doesn't have to honor such a request.

Here's an example of running that program, which we'll call *webget*:

```
% webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html

<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found</H1>
The requested URL /guanaco.html was not found on this server.<P>
</BODY>
```

Ok, so that's not very interesting, because it didn't find that particular document. But a long response wouldn't have fit on this page.

For a more featureful version of this program, you should look to the *lwp-request* program included with the LWP modules from CPAN.

Interactive Client with IO::Socket

Well, that's all fine if you want to send one command and get one answer, but what about setting up something fully interactive, somewhat like the way *telnet* works? That way you can type a line, get the

answer, type a line, get the answer, etc.

This client is more complicated than the two we've done so far, but if you're on a system that supports the powerful `fork` call, the solution isn't that rough. Once you've made the connection to whatever service you'd like to chat with, call `fork` to clone your process. Each of these two identical process has a very simple job to do: the parent copies everything from the socket to standard output, while the child simultaneously copies everything from standard input to the socket. To accomplish the same thing using just one process would be *much* harder, because it's easier to code two processes to do one thing than it is to code one process to do two things. (This keep-it-simple principle a cornerstones of the Unix philosophy, and good software engineering as well, which is probably why it's spread to other systems.)

Here's the code:

```
#!/usr/bin/perl -w
use strict;
use IO::Socket;
my ($host, $port, $kidpid, $handle, $line);

unless (@ARGV == 2) { die "usage: $0 host port" }
($host, $port) = @ARGV;

# create a tcp connection to the specified host and port
$handle = IO::Socket::INET->new(Proto => "tcp",
PeerAddr => $host,
PeerPort => $port)
|| die "can't connect to port $port on $host: $!";

$handle->autoflush(1); # so output gets there right away
print STDERR "[Connected to $host:$port]\n";

# split the program into two processes, identical twins
die "can't fork: $!" unless defined($kidpid = fork());

# the if{} block runs only in the parent process
if ($kidpid) {
# copy the socket to standard output
while (defined ($line = <$handle>)) {
print STDOUT $line;
}
kill("TERM", $kidpid); # send SIGTERM to child
}
# the else{} block runs only in the child process
else {
# copy standard input to the socket
while (defined ($line = <STDIN>)) {
print $handle $line;
}
exit(0) # just in case
}
```

The `kill` function in the parent's `if` block is there to send a signal to our child process, currently running in the `else` block, as soon as the remote server has closed its end of the connection.

If the remote server sends data a byte at time, and you need that data immediately without waiting for a newline (which might not happen), you may wish to replace the `while` loop in the parent with the following:

```
my $byte;
while (sysread($handle, $byte, 1) == 1) {
print STDOUT $byte;
}
```

Making a system call for each byte you want to read is not very efficient (to put it mildly) but is the simplest to explain and works reasonably well.

TCP Servers with IO::Socket

As always, setting up a server is little bit more involved than running a client. The model is that the server creates a special kind of socket that does nothing but listen on a particular port for incoming connections. It does this by calling the `IO::Socket::INET->new()` method with slightly different arguments than the client did.

Proto
: This is which protocol to use. Like our clients, we'll still specify `"tcp"` here.

LocalPort
: We specify a local port in the `LocalPort` argument, which we didn't do for the client. This is service name or port number for which you want to be the server. (Under Unix, ports under 1024 are restricted to the superuser.) In our sample, we'll use port 9000, but you can use any port that's not currently in use on your system. If you try to use one already in used, you'll get an ''Address already in use'' message. Under Unix, the `netstat -a` command will show which services current have servers.

Listen
: The `Listen` parameter is set to the maximum number of pending connections we can accept until we turn away incoming clients. Think of it as a call-waiting queue for your telephone. The low-level Socket module has a special symbol for the system maximum, which is SOMAXCONN.

Reuse
: The `Reuse` parameter is needed so that we restart our server manually without waiting a few minutes to allow system buffers to clear out.

Once the generic server socket has been created using the parameters listed above, the server then waits for a new client to connect to it. The server blocks in the `accept` method, which eventually accepts a bidirectional connection from the remote client. (Make sure to autoflush this handle to circumvent buffering.)

To add to user-friendliness, our server prompts the user for commands. Most servers don't do this. Because of the prompt without a newline, you'll have to use the `sysread` variant of the interactive client above.

This server accepts one of five different commands, sending output back to the client. Unlike most network servers, this one handles only one incoming client at a time. Multitasking servers are covered in Chapter 16 of the Camel.

Here's the code. We'll

```
#!/usr/bin/perl -w
use IO::Socket;
use Net::hostent; # for OOish version of gethostbyaddr

$PORT = 9000; # pick something not in use

$server = IO::Socket::INET->new( Proto => "tcp",
LocalPort => $PORT,
Listen => SOMAXCONN,
Reuse => 1);

die "can't setup server" unless $server;
print "[Server $0 accepting clients]\n";
```

```
while ($client = $server->accept()) {
$client->autoflush(1);
print $client "Welcome to $0; type help for command list.\n";
$hostinfo = gethostbyaddr($client->peeraddr);
printf "[Connect from %s]\n",
$hostinfo ? $hostinfo->name : $client->peerhost;
print $client "Command? ";
while ( <$client>) {
next unless /\S/; # blank line
if (/quit|exit/i) { last }
elsif (/date|time/i) { printf $client "%s\n", scalar localtime() }
elsif (/who/i ) { print $client `who 2>&1` }
elsif (/cookie/i ) { print $client `/usr/games/fortune 2>&1` }
elsif (/motd/i ) { print $client `cat /etc/motd 2>&1` }
else {
print $client "Commands: quit date who cookie motd\n";
}
} continue {
print $client "Command? ";
}
close $client;
}
```

UDP: Message Passing

Another kind of client-server setup is one that uses not connections, but messages. UDP communications involve much lower overhead but also provide less reliability, as there are no promises that messages will arrive at all, let alone in order and unmangled. Still, UDP offers some advantages over TCP, including being able to ‘‘broadcast’’ or ‘‘multicast’’ to a whole bunch of destination hosts at once (usually on your local subnet). If you find yourself overly concerned about reliability and start building checks into your message system, then you probably should use just TCP to start with.

UDP datagrams are *not* a bytestream and should not be treated as such. This makes using I/O mechanisms with internal buffering like stdio (i.e. *print()* and friends) especially cumbersome. Use *syswrite()*, or better *send()*, like in the example below.

Here’s a UDP program similar to the sample Internet TCP client given earlier. However, instead of checking one host at a time, the UDP version will check many of them asynchronously by simulating a multicast and then using *select()* to do a timed-out wait for I/O. To do something similar with TCP, you’d have to use a different socket handle for each host.

```
#!/usr/bin/perl -w
use strict;
use Socket;
use Sys::Hostname;

my ( $count, $hisiaddr, $hispaddr, $histime,
$host, $iaddr, $paddr, $port, $proto,
$rin, $rout, $rtime, $SECS_OF_70_YEARS);

$SECS_OF_70_YEARS = 2_208_988_800;

$iaddr = gethostbyname(hostname());
$proto = getprotobyname("udp");
$port = getservbyname("time", "udp");
$paddr = sockaddr_in(0, $iaddr); # 0 means let kernel pick
```

```
socket(SOCKET, PF_INET, SOCK_DGRAM, $proto) || die "socket: $!";
bind(SOCKET, $paddr) || die "bind: $!";

$| = 1;
printf "%-12s %8s %s\n", "localhost", 0, scalar localtime();
$count = 0;
for $host (@ARGV) {
$count++;
$hisiaddr = inet_aton($host) || die "unknown host";
$hispaddr = sockaddr_in($port, $hisiaddr);
defined(send(SOCKET, 0, 0, $hispaddr)) || die "send $host: $!";
}

$rin = "";
vec($rin, fileno(SOCKET), 1) = 1;

# timeout after 10.0 seconds
while ($count && select($rout = $rin, undef, undef, 10.0)) {
$rtime = "";
$hispaddr = recv(SOCKET, $rtime, 4, 0) || die "recv: $!";
($port, $hisiaddr) = sockaddr_in($hispaddr);
$host = gethostbyaddr($hisiaddr, AF_INET);
$histime = unpack("N", $rtime) - $SECS_OF_70_YEARS;
printf "%-12s ", $host;
printf "%8d %s\n", $histime - time(), scalar localtime($histime);
$count--;
}
```

This example does not include any retries and may consequently fail to contact a reachable host. The most prominent reason for this is congestion of the queues on the sending host if the number of hosts to contact is sufficiently large.

SysV IPC

While System V IPC isn't so widely used as sockets, it still has some interesting uses. However, you cannot use SysV IPC or Berkeley *mmap()* to have a variable shared amongst several processes. That's because Perl would reallocate your string when you weren't wanting it to. You might look into the `IPC::Shareable` or `threads::shared` modules for that.

Here's a small example showing shared memory usage.

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRUSR S_IWUSR);

$size = 2000;
$id = shmget(IPC_PRIVATE, $size, S_IRUSR | S_IWUSR);
defined($id) || die "shmget: $!";
print "shm key $id\n";

$message = "Message #1";
shmwrite($id, $message, 0, 60) || die "shmwrite: $!";
print "wrote: '$message'\n";
shmread($id, $buff, 0, 60) || die "shmread: $!";
print "read : '$buff'\n";

# the buffer of shmread is zero-character end-padded.
substr($buff, index($buff, "\0")) = "";
print "un" unless $buff eq $message;
print "swell\n";
```

```
print "deleting shm $id\n";
shmctl($id, IPC_RMID, 0) || die "shmctl: $!";
```

Here's an example of a semaphore:

```
use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 10, 0666 | IPC_CREAT);
defined($id) || die "semget: $!";
print "sem id $id\n";
```

Put this code in a separate file to be run in more than one process. Call the file *take*:

```
# create a semaphore

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);
defined($id) || die "semget: $!";

$semnum = 0;
$semflag = 0;

# "take" semaphore
# wait for semaphore to be zero
$semop = 0;
$opstring1 = pack("s!s!s!", $semnum, $semop, $semflag);

# Increment the semaphore count
$semop = 1;
$opstring2 = pack("s!s!s!", $semnum, $semop, $semflag);
$opstring = $opstring1 . $opstring2;

semop($id, $opstring) || die "semop: $!";
```

Put this code in a separate file to be run in more than one process. Call this file *give*:

```
# "give" the semaphore
# run this in the original process and you will see
# that the second process continues

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 0, 0);
die unless defined($id);

$semnum = 0;
$semflag = 0;

# Decrement the semaphore count
$semop = -1;
$opstring = pack("s!s!s!", $semnum, $semop, $semflag);

semop($id, $opstring) || die "semop: $!";
```

The SysV IPC code above was written long ago, and it's definitely clunky looking. For a more modern look, see the IPC::SysV module.

A small example demonstrating SysV message queues:

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRUSR S_IWUSR);

my $id = msgget(IPC_PRIVATE, IPC_CREAT | S_IRUSR | S_IWUSR);
defined($id) || die "msgget failed: $!";

my $sent = "message";
my $type_sent = 1234;

msgsnd($id, pack("l! a*", $type_sent, $sent), 0)
|| die "msgsnd failed: $!";

msgrcv($id, my $rcvd_buf, 60, 0, 0)
|| die "msgrcv failed: $!";

my($type_rcvd, $rcvd) = unpack("l! a*", $rcvd_buf);

if ($rcvd eq $sent) {
print "okay\n";
} else {
print "not okay\n";
}

msgctl($id, IPC_RMID, 0) || die "msgctl failed: $!\n";
```

NOTES

Most of these routines quietly but politely return `undef` when they fail instead of causing your program to die right then and there due to an uncaught exception. (Actually, some of the new *Socket* conversion functions do *croak()* on bad arguments.) It is therefore essential to check return values from these functions. Always begin your socket programs this way for optimal success, and don't forget to add the **-T** taint-checking flag to the `#!` line for servers:

```
#!/usr/bin/perl -Tw
use strict;
use sigtrap;
use Socket;
```

BUGS

These routines all create system-specific portability problems. As noted elsewhere, Perl is at the mercy of your C libraries for much of its system behavior. It's probably safest to assume broken SysV semantics for signals and to stick with simple TCP and UDP socket operations; e.g., don't try to pass open file descriptors over a local UDP datagram socket if you want your code to stand a chance of being portable.

AUTHOR

Tom Christiansen, with occasional vestiges of Larry Wall's original version and suggestions from the Perl Porters.

SEE ALSO

There's a lot more to networking than this, but this should get you started.

For intrepid programmers, the indispensable textbook is *Unix Network Programming, 2nd Edition, Volume 1* by W. Richard Stevens (published by Prentice-Hall). Most books on networking address the subject from the perspective of a C programmer; translation to Perl is left as an exercise for the reader.

The *IO::Socket(3)* manpage describes the object library, and the *Socket(3)* manpage describes the low-level interface to sockets. Besides the obvious functions in perlfunc(1), you should also check out the *modules* file at your nearest CPAN site, especially <http://www.cpan.org/modules/00modlist.long.html#ID5_Networking_>. See perlmodlib(1) or best yet, the *Perl FAQ* for a description of what CPAN is and where to get it if the previous link doesn't work for you.

Section 5 of CPAN's *modules* file is devoted to ''Networking, Device Control (modems), and Interprocess Communication'', and contains numerous unbundled modules numerous networking modules, Chat and Expect operations, CGI programming, DCE, FTP, IPC, NNTP, Proxy, Ptty, RPC, SNMP, SMTP, Telnet, Threads, and ToolTalk — to name just a few.