NAME

perlipc − Perl interprocess communication (signals, fifos, pipes, safe subprocceses, 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 an another process, or triggered automatically by the kernel when special events transpire, like a child process exiting, your process running out of stack space, or hitting file size limit.

For example, to trap an interrupt signal, set up a handler like this. Notice how all we do is set with a global variable and then raise an exception. That's because on most systems system I/O libraries are not re-entrant, so calling any *print()* functions could trigger a memory fault and subsequent core dump.

```
sub catch_zap {
    my $signame = shift;
    $shucks++;
    die "Somebody sent me a SIG$signame";
}
$SIG{INT} = 'catch_zap';  # could fail in modules
$SIG{INT} = \&catch_zap;  # best strategy
```

The names of the signals are the ones listed out by `kill -l` on your system, or you can retrieve them from the Config module. Set up an `@signame` list indexed by number to get the name and a `%signo` table indexed by name to get the number:

```
use Config;
defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
    $signo{$name} = $i;
    $signame[$i] = $name;
    $i++;
}
```

So to check whether signal 17 and SIGALRM were the same, just do this:

```
print "signal #17 = $signame[17]\n";
if ($signo{ALRM}) {
    print "SIGALRM is $signo{ALRM}\n";
}
```

You may also choose to assign IGNORE or DEFAULT as the handler, in which case Perl will try to discard the signal or do the default thing. Some signals can be neither trapped nor ignored, such as the KILL and STOP (but not the TSTP) signals. One strategy for temporarily ignoring signals is to use a *local()* statement, which will be automatically restored once your block is exited. (Remember that *local()* values are "inherited" by functions called from within that block.)

```
sub precious {
    local $SIG{INT} = 'IGNORE';
    &more_functions;
}
sub more_functions {
    # interrupts still ignored, for now...
}
```

Sending a signal to a negative process id in Perl means that you send the signal to the entire Unix process-group. This code send a hang-up signal to all processes in the current process group *except for* the current process itself:

```
{
    local $SIG{HUP} = 'IGNORE';
    kill HUP => -$$;
}
```

Another interesting signal to send is signal number zero. This doesn't actually affect another process, but instead checks whether it's alive or has changed its UID.

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

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

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

But that will be problematic for the more complicated handlers that need to re-install themselves. Because Perl's signal mechanism is currently based on the *signal*(3) function from the C library, you may somtimes be so misfortunate as to run on systems where that function is "broken", that is, it behaves in the old unreliable SysV way rather than the newer, more reasonable BSD and POSIX fashion. So you'll see defensive people writing signal handlers like this:

```
sub REAPER {
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    $waitedpid = wait;
}
$SIG{CHLD} = \&REAPER;
# now do something that forks...
```

or even the more elaborate:

```
use POSIX "wait_h";
sub REAPER {
    my $child;
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    while ($child = waitpid(-1,WNOHANG)) {
        $Kid_Status{$child} = $?;
    }
}
$SIG{CHLD} = \&REAPER;
# do something that forks...
```

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, much as you might using *longjmp()* or *throw()* in other languages.

Here's an example:

```
eval {
    local $SIG{ALRM} = sub { die "alarm clock restart" };
    alarm 10;
    flock(FH, 2);   # blocking write lock
    alarm 0;
};
if ($@ and $@ !~ /alarm clock restart/) { die }
```

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

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 a regular, connected anonymous pipes, except that the processes rendezvous using a filename and don't have to be related.

To create a named pipe, use the Unix command *mknod*(1) or on some systems, *mkfifo*(1). These may not be in your normal path.

```
# 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, newsreader, finger program, etc.) tries to read from that file, the reading program will block and your program will supply the the new signature. We'll use the pipe-checking file test **–p** to find out whether anyone (or anything) has accidentally removed our fifo.

```
chdir; # go home
$FIFO = '.signature';
$ENV{PATH} .= ":/etc:/usr/games";

while (1) {
    unless (-p $FIFO) {
        unlink $FIFO;
        system('mknod', $FIFO, 'p')
            && die "can't mknod $FIFO: $!";
    }

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

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 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 SPOOLER || die "bad netstat: $! $?";
```

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

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

and irrespective of which 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 netstat" 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 to kill off the child process early if you'd like.

Be careful to check both the *open()* and the *close()* return values. 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 just return a quick end of file, writers to bogus command will trigger a signal they'd better be prepared to handle. Consider:

```
open(FH, "|bogus");
print FH "bang\n";
close FH;
```

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 *from* minus, you can read from the filehandle you opened whatever your kid writes to his STDOUT.

```
use English;
my $sleep_count = 0;

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

if ($pid) {  # parent
    print KID @some_data;
    close(KID) || warn "kid exited $?";
} else {     # child
    ($EUID, $EGID) = ($UID, $GID); # suid progs only
    open (FILE, "> /safe/file")
        || die "can't open /safe/file: $!";
    while (<STDIN>) {
        print FILE; # child's STDIN is parent's KID
    }
    exit;  # 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 straigh-forward, 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:

```
# add error processing as above
$pid = open(KID, "-|");

if ($pid) {   # parent
    while (<KID>) {
        # do something interesting
    }
    close(KID) || 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:

```
# add error processing as above
$pid = open(KID, "|-");
$SIG{ALRM} = sub { die "whoops, $program pipe broke" };
```

```
if ($pid) {   # parent
    for (@data) {
        print KID;
    }
    close(KID) || warn "kid exited $?";

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

Note that these operations are full Unix forks, this means they may not be correctly implemented on alien systems. Additionally, these are not true multithreading. If you'd like to learn more about threading, see the *modules* file mentioned below in the the section on *SEE ALSO* section.

Bidirectional Communication

While this works reasonably well for unidirectional communication, what about bidirectional communication? The obvious thing you'd like to do doesn't actually work:

```
open(KID, "| some program |")
```

and if you forgot to use the **−w** flag, then you'll miss out entirely on the diagnostic message:

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

If you really want to, you can use the standard *open2()* library function to catch both ends. There's also an *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 Unix *pipe()* and *exec()* to create all the connections. While it might have been slightly more efficient by using *socketpair()*, it would have then been even less portable than it already is. The *open2()* and *open3()* functions are unlikely to work anywhere except on a Unix system or some other one purporting to be POSIX compliant.

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

```
use FileHandle;
use IPC::Open2;
$pid = open2( \*Reader, \*Writer, "cat −u −n" );
Writer−>autoflush(); # default here, actually
print Writer "stuff\n";
$got = <Reader>;
```

The problem with this is that Unix buffering is going to really ruin your day. Even though your `Writer` filehandle is autoflushed, and the process on the other end will get your data in a timely manner, you can't usually do anything to force it to actually give it back to you in a similarly quick fashion. In this case, we could, because we gave *cat* a **−u** flag to make it unbuffered. But very few Unix 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 the non-standard *Comm.pl* library. It uses pseudo-ttys to make your program behave more reasonably:

```
require 'Comm.pl';
$ph = open_proc('cat -n');
for (1..10) {
    print $ph "a line\n";
    print "got back ", scalar <$ph>;
}
```

This way you don't have to have control over the source code of the program you're using. The *Comm* library also has *expect()* and *interact()* functions. Find the library (and hopefully its successor *IPC::Chat*) at your nearest CPAN archive as detailed in the the section on *SEE ALSO* section below.

Sockets: Client/Server Communication

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

The Perl function calls 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 old 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 various constants and functions you'll need.

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
require 5.002;
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) }
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;
```

And here's a corresponding server to go along with it. We'll leave the address empty (`"\0" x 4`) so that the kernel can choose the appropriate interface on multihomed hosts:

```
#!/usr/bin/perl -Tw
require 5.002;
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;

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

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
socket(SERVER, PF_INET, SOCK_STREAM, $proto)        || die "socket: $!";
setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1) || die "setsockopt: $!";
bind(SERVER, sockaddr_in($port, "\0" x 4))  || die "bind: $!";
listen(SERVER,5)                            || die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

sub REAPER {
    $SIG{CHLD} = \&REAPER;  # loathe sysV
    $waitedpid = wait;
    logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
}

$SIG{CHLD} = \&REAPER;

for ( $waitedpid = 0;
      ($paddr = accept(CLIENT,SERVER)) || $waitedpid;
      $waitedpid = 0, close CLIENT)
{
    next if $waitedpid;
    my($port,$iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr,AF_INET);

    logmsg "connection from $name ", fmtaddr($iaddr), " at port $port";

    spawn sub {
        print "Hello there, $name, it's now ", scalar localtime, "\n";
        exec '/usr/games/fortune'
            or confess "can't exec fortune: $!";
    };

}

sub spawn {
    my $coderef = shift;
```

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

        my $pid;
        if (!defined($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.

We suggest that you use the **−T** flag to use taint checking (see the *perlsec* manpage) even if we aren't running setuid or setgid. This is always a good idea for servers and other programs 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
    require 5.002;
    use strict;
    use Socket;

    my $SECS_of_70_YEARS = 2208988800;
    sub ctime { scalar localtime(shift) }

    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(time());
```

```
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 "bind: $!";
    my $rtime = '    ';
    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 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 print system";
}
```

Here's a sample Unix-domain client:

```
#!/usr/bin/perl -w
require 5.002;
use Socket;
use strict;
my ($rendezvous, $line);

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

And here's a corresponding server.

```
#!/usr/bin/perl -Tw
require 5.002;
use strict;
use Socket;
use Carp;

BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
```

```
my $NAME = '/tmp/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,5)                    || die "listen: $!";

logmsg "server started on $NAME";

$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' or die "can't exec fortune: $!";
    };
}
```

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 exactly 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 from the World Wide Web to be able to access, 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.

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 just use TCP to start with.

Here's a UDP program similar to the sample Internet TCP client given above. 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;
require 5.002;
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      = 2208988800;

$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 time;
$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 = $rin, undef, undef, 10.0)) {
    ($hispaddr = recv(SOCKET, $rtime, 4, 0))        || die "recv: $!";
    ($port, $hisiaddr) = sockaddr_in($hispaddr);
    $host = gethostbyaddr($hisiaddr, AF_INET);
    $rtime = '';
    $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
    printf "%-12s ", $host;
    printf "%8d %s\n", $histime - time, scalar localtime($histime);
    $count--;
}
```

SysV IPC

While System V IPC isn't so widely used as sockets, it still has some interesting uses. You can't, however, effectively use SysV IPC or Berkeley *mmap()* to have shared memory so as to share a variable amongst several processes. That's because Perl would reallocate your string when you weren't wanting it to.

Here's a small example showing shared memory usage.

```
$IPC_PRIVATE = 0;
$IPC_RMID = 0;
$size = 2000;
$key = shmget($IPC_PRIVATE, $size , 0777 );
die unless defined $key;
```

```
$message = "Message #1";
shmwrite($key, $message, 0, 60 ) || die "$!";
shmread($key,$buff,0,60) || die "$!";

print $buff,"\n";

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

Here's an example of a semaphore:

```
$IPC_KEY = 1234;
$IPC_RMID = 0;
$IPC_CREATE = 0001000;
$key = semget($IPC_KEY, $nsems , 0666 | $IPC_CREATE );
die if !defined($key);
print "$key\n";
```

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

```
# create a semaphore

$IPC_KEY = 1234;
$key = semget($IPC_KEY,  0 , 0 );
die if !defined($key);

$semnum = 0;
$semflag = 0;

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

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

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

Put this code in a separate file to be run in more that 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;
$key = semget($IPC_KEY, 0, 0);
die if !defined($key);

$semnum = 0;
$semflag = 0;
```

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

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

WARNING

The SysV IPC code was written long ago, and is definitely clunky looking. It should at the very least be made to `use strict` and `require "sys/ipc.ph"`. Better yet, someone should create an `IPC::SysV` module the way we have the `Socket` module for normal client-server communications.

NOTES

If you are running under a version 5.001 or 5.002, you can use still use most of the examples in this document. You may have to remove the `use strict` and some of the *my()* statements for 5.000, and for both you'll have to load in the non-standard *SockFuncs.pm* module, which was/is/shall be incorporated into the standard *Socket.pm* module.

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. It is therefore essential that you should check the return values fo these functions. Always begin your socket programs this way for optimal success, and don't forget to add **–T** taint checking flag to the pound-bang line for servers:

```
#!/usr/bin/perl -w
require 5.002;
use strict;
use sigtrap;
use Socket;
```

BUGS

All these routines create system-specific portability problems. As noted elsewhere, Perl is at the mercy of your C libraries for much of its system behaviour. 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 filedescriptors 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.

SEE ALSO

Besides the obvious functions in the *perlfunc* manpage, you should also check out the *modules* file at your nearest CPAN site. (See the *perlmod* manpage or best yet, the *Perl FAQ* for a description of what CPAN is and where to get it.) Section 5 of the *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—just to name a few.