perlfaq8

Table des matières

1 NAME/NOM

perlfaq8 - Interactions avec le système

2 DESCRIPTION

Cette section de la FAQ Perl traite des questions concernant les interactions avec le système d'exploitation. Cela inclut les mécanismes de communication inter-processus (IPC – Inter Process Communication en anglais), le pilotage de l'interface utilisateur (clavier, écran et souris), et d'une façon générale tout ce qui ne relève pas de la manipulation de données.

Lisez les FAQ et la documentation spécifique au portage de perl sur votre système d'exploitation (par ex. *perlvms*, *perlplan9*, etc.). Vous devriez y trouver de plus amples informations sur les spécificités de votre perl.

2.1 Comment savoir sur quel système d'exploitation je tourne ?

La variable $^O ($OSNAME si vous utilisez le module English) contient une indication sur le nom du système d'exploitation (pas son numéro de version) sur lequel votre exécutable perl a été compilé.

2.2 Pourquoi ne revient-on pas après un exec() ?

Parce que c'est ainsi qu'il fonctionne : il remplace le processus qui tourne par un nouveau. Si vous voulez continuer après (ce qui est probablement le cas si vous vous posez cette question), utilisez plutôt system().

2.3 Comment utiliser le clavier/écran/souris de façon élaborée ?

La façon d'accéder aux (ou de contrôler les) claviers, écrans et souris ("mulots") dépend fortement du système d'exploitation. Essayez les modules suivants :

Clavier

Term::Cap	Distribution standard
Term::ReadKey	CPAN
Term::ReadLine::Gnu	CPAN
Term::ReadLine::Perl	CPAN
Term::Screen	CPAN

Écran

Term::Cap	Distribution standard
Curses	CPAN
Term::ANSIColor	CPAN

Souris

Tk	CPAN

Certains cas spécifiques sont examinés sous forme d'exemples dans d'autres réponses de cette FAQ.

2.4 Comment afficher quelque chose en couleur ?

En général, on ne le fait pas, parce qu'on ne sait pas si le receveur a un afficheur comprenant les couleurs. Cependant, si vous avez la certitude de trouver à l'autre bout un terminal ANSI qui traite la couleur, vous pouvez utiliser le module Term::ANSIColor de CPAN :

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

Ou comme ceci :

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

2.5 Comment lire simplement une touche sans attendre un appui sur "entrée" ?

Le contrôle des tampons en entrée est fortement dépendant du système. Sur la plupart d'entre eux, vous pouvez simplement utiliser la commande **stty** comme montré dans getc in *perlfunc*, mais visiblement, cela vous entraîne déjà dans les méandres de la portabilité.

```
open(TTY, "+</dev/tty") or die "no tty: $!";
system "stty  cbreak </dev/tty >/dev/tty 2>&1";
$key = getc(TTY);           # peut marcher
# OU BIEN
sysread(TTY, $key, 1);      # marche sans doute
system "stty -cbreak </dev/tty >/dev/tty 2>&1";
```

Le module Term::ReadKey de CPAN offre une interface prête à l'emploi, et devrait être plus efficace que de lancer des **stty** pour chaque touche. Il inclut même un support limité pour Windows.

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

Cependant, cela requiert que vous ayez un compilateur C fonctionnel, qui puisse être utilisé pour compiler et installer des modules de CPAN. Voici une solution n'utilisant que le module standard POSIX, qui devrait être disponible en natif sur votre système (s'il est lui-même POSIX).

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

Et voici le module HotKey, qui cache les appels POSIX gérant les structures termios, qui sont assez ésotériques, il faut bien l'avouer :

```
# HotKey.pm
package HotKey;

@ISA = qw(Exporter);
@EXPORT = qw(cbreak cooked readkey);

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

$fd_stdin = fileno(STDIN);
$term     = POSIX::Termios->new();
$term->getattr($fd_stdin);
$oterm     = $term->getlflag();

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

sub cbreak {
    $term->setlflag($noecho);  # ok, on ne veut pas d'écho non plus
    $term->setcc(VTIME, 1);
    $term->setattr($fd_stdin, TCSANOW);
}

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

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

END { cooked() }

1;
```

2.6 Comment vérifier si des données sont en attente depuis le clavier ?

Le moyen le plus facile pour cela est de lire une touche en mode non bloquant, en utilisant le module Term::ReadKey de CPAN, et en lui passant un argument de -1 pour indiquer ce fait.

```
use Term::ReadKey;

ReadMode('cbreak');

if (defined ($char = ReadKey(-1)) ) {
    # il y avait un caractère disponible, maintenant dans $char
} else {
    # pas de touche pressée pour l'instant
}

ReadMode('normal');           # restaure le terminal en mode normal
```

2.7 Comment effacer l'écran ?

Si vous devez le faire de façon occasionnelle, utilisez `system` :

```
system("clear");
```

Si ce doit être une opération fréquente, sauvegardez la séquence de nettoyage pour pouvoir ensuite l'afficher 100 fois sans avoir à appeler un programme externe autant de fois :

```
$clear_string = `clear`;
print $clear_string;
```

Si vous prévoyez d'autres manipulations d'écran, comme le positionnement du curseur, etc., utilisez plutôt le module Term::Cap :

```
use Term::Cap;
$terminal = Term::Cap->Tgetent( {OSPEED => 9600} );
$clear_string = $terminal->Tputs('cl');
```

2.8 Comment obtenir la taille de l'écran ?

Si vous avez le module Term::ReadKey de CPAN, vous pouvez l'utiliser pour récupérer la hauteur et la largeur en caractères et en pixels:

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

Cela est plus portable qu'un appel direct à `ioctl`, mais n'est pas aussi illustratif :

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

2.9 Comment demander un mot de passe à un utilisateur ?

(Cette question n'a rien à voir avec le web. Voir une autre FAQ pour cela.)

Il y a un exemple dans crypt in *perlfunc*. Tout d'abord, on place le terminal en mode "sans écho", puis on lit simplement le mot de passe. Cela peut se faire par un appel à la bonne vieille fonction ioctl(), par l'utilisation du contrôle du terminal de POSIX (voir *POSIX* ou le Camel Book), ou encore par un appel au programme **stty**, avec divers degrés de portabilité.

On peut aussi, sur la plupart des systèmes, utiliser le module Term::ReadKey de CPAN, qui est le plus simple à utiliser et le plus portable en théorie.

```
use Term::ReadKey;

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

2.10 Comment lire et écrire sur le port série ?

Cela dépend du système d'exploitation au-dessus duquel tourne votre programme. Dans la plupart des systèmes Unix, les ports série sont accessibles depuis des fichiers sous /dev ; sur d'autres systèmes, les noms de périphériques seront indubitablement différents. Il y a plusieurs types de problèmes, communs à toutes les interactions avec un périphérique :

verrouillage
: Votre système peut utiliser des fichiers de verrouillage pour contrôler les accès concurrentiels. Soyez certain de suivre le bon protocole de verrouillage. Des comportements imprévisibles peuvent survenir suite à un accès simultané, par différents processus, au même périphérique.

mode d'ouverture
: Si vous prévoyez d'utiliser à la fois des opérations de lecture et d'écriture sur le périphérique, vous devrez l'ouvrir en mode de mise à jour (voir open in *perlfunc* pour plus de détails). Vous pouvez aussi vouloir l'ouvrir sans prendre le risque de bloquer, en utilisant sysopen() et les constantes `O_RDWR|O_NDELAY|O_NOCTTY` fournies par le module Fcntl (qui fait partie de la distribution standard). Se référer à sysopen in *perlfunc* pour plus de précisions quant à cette approche.

fin de ligne
: Certains périphériques vont s'attendre à trouver un "\r" à la fin de chaque ligne plutôt qu'un "\n". Sur certains portages de perl, "\r" et "\n" sont différents de leur valeurs usuelles (Unix) qui en ASCII sont respectivement "\012" et "\015". Il se peut que vous ayez à utiliser ces valeurs numériques directement, sous leur forme octale ("\015"), hexadécimale ("0x0D"), ou sous forme de caractère de contrôle ("\cM").

```
print DEV "atv1\012";       # mauvais, pour certains périphériques
print DEV "atv1\015";       # bon, pour certains périphériques
```

Bien que pour les fichiers de texte normaux, un "\n" fasse l'affaire, il n'y a toujours pas de schéma unifié pour terminer une ligne qui soit portable entre Unix, DOS/Win et Macintosh, sauf à terminer *TOUTES* les lignes par un "\015\012", et de retirer ce dont vous n'avez pas besoin dans le résultat en sortie. Cela s'applique tout particulièrement aux E/S sur les prises (sockets) et à la purge des tampons, problèmes qui sont discutés ci-après.

purge des tampons de sortie
: Si vous vous attendez à ce que tous les caractères que vous émettez avec print() sortent immédiatement, il vous faudra activer la purge automatique des tampons de sortie (autoflush) sur ce descripteur de fichier. Vous pouvez utiliser select() et la variable `$|` pour contrôler cette purge (voir $| in *perlvar* et select in *perlfunc* ou Comment vider/annuler les tampons en sortie ? Pourquoi m'en soucier ? in *perlfaq5*):

```
$oldh = select(DEV);
$| = 1;
select($oldh);
```

Vous verrez aussi du code qui réalise l'opération sans variable temporaire :

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

Ou, si le fait de charger quelque milliers de lignes de code, simplement par peur d'une toute petite variable comme $|, ne vous ennuie pas outre mesure :

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

Comme expliqué dans le point précédent, cela ne marchera pas si vous utilisez des E/S sur une prise entre Unix et un Macintosh. Vous devrez câbler vos terminaisons de ligne, dans ce cas.

entrée non bloquante
: Si vous effectuez une lecture bloquante par read() ou sysread(), vous devrez vous arranger pour que le déclenchement d'une alarme vous fournisse le déblocage nécessaire (voir alarm in *perlfunc*). Si vous avez effectué une ouverture non bloquante, vous bénéficierez certainement d'une lecture non bloquante, ce qui peut forcer l'utilisation de select() (dans sa version avec 4 arguments), pour déterminer si l'E/S est possible ou non sur ce périphérique (voir select in *perlfunc*.)

En cherchant à lire sa boîte vocale, le fameux Jamie Zawinski <jwz@netscape.com>, après de nombreux grincements de dents et une lutte avec sysread, sysopen, les caprices de la fonction POSIX tcgetattr, et de nombreuses autres fonctions promettant de s'éclater la nuit, est finalement arrivé à ceci :

```
sub open_modem {
    use IPC::Open2;
    my $stty = `/bin/stty -g`;
    open2( \*MODEM_IN, \*MODEM_OUT, "cu -l$modem_device -s2400 2>&1");
    # lancer cu trafique les paramètres de /dev/tty, même lorsqu'il a
    # été lancé depuis un tube...
    system("/bin/stty $stty");
    $_ = <MODEM_IN>;
    chomp;
    if ( !m/^Connected/ ) {
        print STDERR "$0: cu printed `$_' instead of `Connected'\n";
    }
}
```

2.11 Comment décoder les fichiers de mots de passe cryptés ?

En dépensant d'énormes quantités d'argent pour du matériel dédié, mais cela finira par attirer l'attention.

Sérieusement, cela n'est pas possible si ce sont des mots de passe Unix – le système de cryptage des mots de passe sur Unix utilise une fonction de hachage à sens unique. C'est plus du hachage que de l'encryption. La meilleure méthode consiste à trouver quelque chose d'autre qui se hache de la même façon. Il n'est pas possible d'inverser la fonction pour retrouver la chaîne d'origine. Des programmes comme Crack peuvent essayer de deviner les mots de passe de façon brutale (et futée), mais ne vont pas (ne peuvent pas) garantir de succès rapide.

Si vous avez peur que vos utilisateurs ne choisissent de mauvais mots de passe, vous devriez le vérifier au vol lorsqu'ils essaient de changer leur mot-de-passe (en modifiant la commande passwd(1) par exemple).

2.12 Comment lancer un processus en arrière plan ?

Plusieurs modules savent lancer d'autres processus sans bloquer votre prorgamme Perl. Vous pouvez utiliser IPC::Open3, Parallel::Jobs, IPC::Run et quelques-uns des modules POE. Voir CPAN pour plus de détails.

Vous pouvez aussi utiliser

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

ou utiliser fork comme expliqué dans fork in *perlfunc*, avec des exemples supplémentaires dans *perlipc*. Voici quelques petites choses dont il vaut mieux être conscient sur un système de type Unix:

STDIN, STDOUT, et STDERR sont partagés.

Le processus principal et celui en arrière plan (le processus "fils") partagent les mêmes descripteurs de fichier STDIN, STDOUT et STDERR. Si les deux essaient d'y accéder en même temps, d'étranges phénomènes peuvent survenir. Il vaudrait mieux les fermer ou les réouvir dans le fils. On peut contourner ce fait en ouvrant un tube via `open` (voir open in *perlfunc*) mais sur certains systèmes, cela implique que le processus fils ne puisse pas survivre à son père.

Signaux

Il faudra capturer les signaux SIGCHLD, et peut être SIGPIPE aussi. Un SIGCHLD est envoyé lorsque le processus en arrière plan se termine. Un SIGPIPE est envoyé lorsque l'on écrit dans un descripteur de fichier que le processus fils a fermé (ne pas capturer un SIGPIPE peut causer silencieusement la mort du processus). Cela n'est pas un problème avec `system("cmd&")`.

Zombies

Il faut vous préparer à "collecter" les processus fils qui se terminent :

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

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

Voous pouvez aussi utiliser la technique du double fork. Vous faites un wait() immédiat de votre fils et c'est le daemon init qui fera le wait() de votre petit-fils lorsqu'il se terminera.

```
unless ($pid = fork) {
   unless (fork) {
     exec "ce que vous voulez réellement faire";
     die "échec d'exec !";
   }
   exit 0;
}
waitpid($pid,0);
```

Voir Signaux in *perlipc* pour d'autres exemple de code réalisant cela. Il n'est pas possible d'obtenir des zombies avec `system("prog &")`.

2.13 Comment capturer un caractère de contrôle, un signal ?

On ne "capture" (en anglais "trap") pas vraiment un caractère de contrôle. En fait, ce caractère génère un signal qui est envoyé au groupe du terminal sur lequel tourne le processus en avant plan, signal que l'on capture ensuite dans le processus. Les signaux sont documentés dans Signaux in *perlipc* et dans le chapitre "Signaux" du Camel Book.

Vous pouvez utiliser la table de hachage %SIG pour y attacher vos fonctions de gestion des signaux. Lorsque perl reçoit un signal, il cherche dans %SIG une clé identique au nom du signal reçu et appelle la fonction associée à cette clé.

```
# via un sous-programme anonyme

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

# via une référence à une fonction

$SIG{INT} = \&ouch;

# via une chaîne content le nom de la fonction
# (référence symbolique)

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

Dans les versions de Perl antérieures à 5.8, le signal était traité dès sa réception via du code C qui captait le signal et appelait directement une éventuelle fonction Perl stockée dans %SIG. Cela pouvait amener perl à planter. Depuis la version 5.8.0, perl regarde dans %SIG *après* la réception du signal, et non au moment de sa réception. Les versions antérieures de cette réponse étaient erronées.

2.14 Comment modifier le fichier masqué (shadow) de mots de passe sous Unix ?

Si perl a été installé correctement et que votre librairie d'accès au fichier masqué est écrite proprement, alors les fonctions getpw*() décrites dans *perlfunc* devraient, en théorie, fournir un accès (en lecture seule) aux mots de passe masqués. Pour changer le fichier, faites une copie du fichier de masque (son format varie selon les systèmes - voir passwd(5) pour les détails) et utilisez pwd_mkdb(8) pour l'installer (voir pwd_mkdb pour plus de détails).

2.15 Comment positionner l'heure et la date ?

En supposant que vous tourniez avec des privilèges suffisants, vous devriez être capables de changer l'heure du système et le temps en lançant la commande date(1). (Il n'y a pas moyen de positionner l'heure et la date pour un processus seulement.) Ce mécanisme fonctionnera sous Unix, MS-DOS, Windows et NT ; sous VMS une commande équivalente est `set time`.

Si par contre vous désirez seulement changer de fuseau horaire, vous pourrez certainement vous en sortir en positionnant une variable d'environnement :

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

2.16 Comment effectuer un sleep() ou alarm() de moins d'une seconde ?

Pour obtenir une granularité plus fine que la seconde obtenue par la fonction sleep(), le plus simple est d'utiliser select() comme décrit dans select in *perlfunc*. Essayez aussi les modules Time::HiRes et BSD::Itimer (disponibles sur CPAN et dans la distribution standard depuis Perl 5.8 pour Time::HiRes).

2.17 Comment mesurer un temps inférieur à une seconde ?

En général, cela risque d'être difficile. Le module Time::HiRes (disponible sur CPAN et dans la distribution standard depuis Perl 5.8) apporte cette fonctionnalité sur certains systèmes.

Si votre système supporte à la fois la fonction syscall() en Perl et un appel système tel que gettimeofday(2), alors vous pouvez peut-être faire quelque chose comme ceci :

```
require 'sys/syscall.ph';

$TIMEVAL_T = "LL";

$done = $start = pack($TIMEVAL_T, ());

syscall(&SYS_gettimeofday, $start, 0) != -1
           or die "gettimeofday: $!";

   ##############################
   # FAITES VOS OPERATIONS ICI #
   ##############################

syscall( &SYS_gettimeofday, $done, 0) != -1
       or die "gettimeofday: $!";

@start = unpack($TIMEVAL_T, $start);
@done  = unpack($TIMEVAL_T, $done);

# corriger les microsecondes
for ($done[1], $start[1]) { $_ /= 1_000_000 }

$delta_time = sprintf "%.4f", ($done[0]  + $done[1]  )
                                                 -
                              ($start[0] + $start[1] );
```

2.18 Comment réaliser un atexit() ou setjmp()/longjmp() ? (traitement d'exceptions)

La version 5 de Perl apporte le bloc END, qui peut être utilisé pour simuler atexit(). Le bloc END de chaque paquetage est appelé lorsque le programme ou le fil d'exécution se termine (voir la page *perlmod* pour plus de details).

Par exemple, on peut utiliser ceci pour s'assurer qu'un programme filtre est bien parvenu à vider son tampon de sortie sans remplir tout le disque :

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

Par contre, le bloc END n'est pas appelé lorsqu'un signal non capturé tue le programme, donc si vous utilisez les blocs END, vous devriez aussi utiliser :

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

En Perl, le traitement des exceptions s'effectue au travers de l'opération eval(). Vous pouvez utiliser eval() en lieu et place de setjmp, et die() pour longjmp(). Pour plus de détails sur cela, lire la section sur les signaux, et plus particulièrement le traitement limitant le temps de blocage pour un flock() dans Signaux in *perlipc* et le chapitre "Sigaux" du Camel Book.

Si tout ce qui vous intéresse est le traitement des exceptions proprement dit, essayez la bibliothèque exceptions.pl (qui fait partie de la distribution standard de Perl).

Si vous préférez la syntaxe de atexit() (et désirez rmexit() aussi), essayez le module AtExit disponible sur CPAN.

2.19 Pourquoi mes programmes avec socket() ne marchent pas sous System V (Solaris) ? Que signifie le message d'erreur « Protocole non supporté » ?

Certains systèmes basés sur Sys-V, et particulièrement Solaris 2.X, ont redéfini certaines constantes liée aux prises (sockets) et qui étaient des standards de fait. Étant donné que ces constantes étaient communes sur toutes les architectures, elles étaient souvent câblées dans le code perl. La bonne manière de résoudre ce problème est d'utiliser "use Socket" pour obtenir les valeurs correctes.

Notez que bien que SunOS et Solaris soient compatibles au niveau des binaires, ces valeurs sont néanmoins différentes. Allez comprendre !

2.20 Comment appeler les fonctions C spécifiques à mon système depuis Perl ?

Dans la plupart des cas, on écrit un module externe – voir la réponse à la question "Où puis-je apprendre à lier du C avec Perl? [h2xs, xsubpp]". Cependant, si la fonction est un appel système et que votre système supporte syscall(), vous pouvez l'utiliser pour ce faire (documentée dans *perlfunc*).

Rappelez-vous de regarder les modules qui sont livrés avec votre distribution, ainsi que ceux disponibles sur CPAN - quelqu'un a peut-être deja écrit un module pour le faire. Sur Windows, essayez Win32::API. Sur Mac, essayez Mac::Carbon. Si aucun module ne propose d'interface vers cette fonction C, vous pouvez insérez un peu de code C directement dans votre code Perl via le module Inline::C.

2.21 Où trouver les fichiers d'inclusion pour ioctl() et syscall() ?

Historiquement, ceux-ci sont générés par l'outil h2ph, inclus dans les distributions standard de perl. Ce programme convertit les directives cpp(1) du fichier d'inclusion C en un fichier contenant des définitions de sous-routines, comme &SYS_getitimer, qui peuvent ensuite être utilisées comme argument de vos fonctions. Cela ne fonctionne pas parfaitement, mais l'essentiel du travail est fait. Des fichiers simples comme *errno.h*, *syscall.h*, et *socket.h* donnent de bons résultats, mais de plus complexes comme *ioctl.h* demandent presque toujours une intervention manuelle après coup. Voici comment installer les fichiers *.ph:

```
1.  devenir super-utilisateur
2.  cd /usr/include
3.  h2ph *.h */*.h
```

Si votre système supporte le chargement dynamique, et pour des raisons de portabilité et de sanité, vous devriez plutôt utiliser h2xs (qui fait lui aussi partie de la distribution standard de perl). Cet outil convertit un fichier d'inclusion C en une extension Perl. Voir *perlxstut* pour savoir comment débuter avec h2xs.

Si votre système ne support pas le chargement dynamique, vous pouvez néanmoins utiliser h2xs. Voir *perlxstut* et *ExtUtils::MakeMaker* pour plus d'information (brièvement, utilisez simplement **make perl** et non un simple **make** pour reconstruire un perl avec une nouvelle extension statiquement liée).

2.22 Pourquoi les scripts perl en setuid se plaignent-ils d'un problème noyau ?

Certains systèmes d'exploitation ont un bug dans leur noyau qui rend les scripts setuid [NDT : fichiers dont le bit 's' est positionné sur l'exécutable, par exemple avec "chmod u+s" sous Unix] non sûrs intrinsèquement. Perl vous offre quelques options (décrites dans *perlsec*) pour contourner ce fait sur ces systèmes.

2.23 Comment ouvrir un tube depuis et vers une commande simultanément ?

Le module IPC::Open2 (qui fait partie de la distribution perl standard) est une approche aisée qui utilise en interne les appels systèmes pipe(), fork() et exec(). Cependant, soyez sûrs de bien lire les avertissements concernant les interblocages dans sa documentation (voir *IPC::Open2*). Voir aussi Communication bidirectionnelle avec un autre processus in *perlipc* et Communication bidirectionnelle avec vous-même in *perlipc*.

On peut aussi utiliser le module IPC:Open3 (lui aussi dans la distribution standard), mais attention: l'ordre des arguments est différent de celui utilisé par IPC::Open2 (voir *IPC::Open3*).

2.24 Pourquoi ne puis-je pas obtenir la sortie d'une commande avec system() ?

Vous confondez system() avec les apostrophes inversées (backticks, ‘‘). La fonction system() lance une commande et retourne sa valeur de sortie (une valeur sur 16 bits : les 7 bits de poids faible indiquent le signal qui a tué le processus le cas échéant, et les 8 bits de poids fort sont la valeur de sortie effective). Les apostrophes inversées (‘‘) lancent une commande et retournent ce que cette commande a émis sur STDOUT.

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

2.25 Comment capturer la sortie STDERR d'une commande externe ?

Il y a trois moyens fondamentaux de lancer une commande externe :

```
system $cmd;                # avec system()
$output = `$cmd`;           # avec les apostrophes inversées (``)
open (PIPE, "cmd |");       # avec open()
```

Avec system(), STDOUT et STDERR vont tous deux aller là où ces descripteurs sont dirigés dans le script lui-même, sauf si la commande system() les redirige exiplicitement par ailleurs. Les apostrophes inversées et open() lisent **uniquement** la sortie (STDOUT) de votre commande externe.

Vous pouvez aussi utiliser la fonction open3() du module IPC::Open3. Benjamin Goldberg propose les quelques exemples de code suivants.

Pour récupérer le STDOUT de votre programme externe et se débarrasser de son STDERR :

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

Pour récupérer le STDERR de votre programme externe et se débarrasser de son STDOUT :

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

Pour récupérer le STDERR de votre programme externe, et rediriger son STDOUT vers votre propre STDERR :

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

Pour récupérer séparément le STDOUT et le STDERR de vote commande externe, vous pouvez les rediriger vers des fichiers temporaires, exécuter la commande puis lire les fichiers temporaires :

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

Mais les *deux* fichiers temporaires ne sont pas absolument indispensables. Le code suivant fonctionnera aussi bien, sans interblocage :

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

Et il sera aussi plus rapide puisque vous pourrez traiter la sortie du programme immédiatement plutôt que d'attendre la terminaison du programme.

Avec toutes ces routines, vous pouvez changer les descripteurs de fichier avant l'appel :

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

ou vous pouvez utiliser les redirections du shell de Bourne :

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

Vous pouvez aussi utiliser les redirections de fichier pour rendre STDERR un synonyme de STDOUT :

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

Cependant, vous ne *pouvez pas* simplement ouvrir STDERR en synonyme de STDOUT depuis votre programme Perl, et ne pas recourir ensuite à la redirection en shell. Ceci ne marche pas :

```
open(STDERR, ">&STDOUT");
$alloutput = `cmd args`;  # stderr n'est toujours pas capturé
```

Cela échoue parce que open() rend STDERR un synonyme de STDOUT au moment où l'appel à open() a été effectué. Les apostrophes inversées redirigent ensuite STDOUT vers une chaîne, mais ne changent pas STDERR (qui va toujours là où allait le STDOUT d'origine).

Notez bien que vous *devez* utiliser la syntaxe de redirection du shell de Bourne (sh(1)), et non celle de csh(1) ! Des précisions sur les raisons pour lesquelles Perl utilise le shell de Bourne pour system() et les apostrophes inversées, ainsi que lors des ouvertures de tubes, se trouvent dans l'article *versus/csh.whynot* de la collection "Far More Than You Ever Wanted To Know" sur http://www.cpan.org/misc/olddoc/FMTEYEWTK.tgz.

Pour capturer à la fois le STDOUT et le STDERR d'une commande :

```
$output = `cmd 2>&1`;                       # soit avec des apostrophes
$pid = open(PH, "cmd 2>&1 |");              # inversées, soit avec un tube
while (<PH>) { }                            #    plus une lecture
```

Pour capturer seulement le STDOUT et jeter le STDERR d'une commande :

```
$output = `cmd 2>/dev/null`;                # soit avec des apostrophes
$pid = open(PH, "cmd 2>/dev/null |");       # inversées, soit avec un tube
while (<PH>) { }                            #    plus une lecture
```

Pour capturer seulement le STDERR et jeter le STDOUT d'une commande :

```
$output = `cmd 2>&1 1>/dev/null`;           # soit avec des apostrophes
$pid = open(PH, "cmd 2>&1 1>/dev/null |");  # inversées, soit avec un tube
while (<PH>) { }                            #    plus une lecture
```

Pour échanger le STDOUT et le STDERR d'une commande afin d'en capturer le STDERR mais laisser le STDOUT sortir à la place du STDERR d'origine :

```
$output = `cmd 3>&1 1>&2 2>&3 3>&-`;        # soit avec des apostrophes
$pid = open(PH, "cmd 3>&1 1>&2 2>&3 3>&-|");# inversées, soit avec un tube
while (<PH>) { }                            #    plus une lecture
```

Pour lire séparément le STDOUT et le STDERR d'une commande, il est plus facile de les rediriger chacun dans un fichier, et ensuite de les lire lorsque la commande est terminée :

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

L'ordre est important dans tout ces exemples. Cela est dû au fait que le shell prend toujours en compte les redirections en les lisant strictement de gauche à droite.

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

La première commande redirige à la fois la sortie standard et la sortie d'erreur dans le fichier temporaire. La seconde commande n'y envoie que la sortie standard d'origine, et la sortie d'erreur d'origine se retrouve envoyé vers la sortie standard d'origine.

2.26 Pourquoi open() ne retourne-t-il pas d'erreur lorsque l'ouverture du tube échoue ?

Si le second argument d'un open() utilisant un tube contient des métacaractères du shell, perl fait un fork() puis exécute un shell pour décoder les métacaractères et éventuellement exécuter le programme voulu. Si le programme ne peut s'exécuter, c'est le shell qui reçoit le message et non Perl. Tout ce que verra Perl c'est s'il a réussi à lancer le shell. Vous pouvez toujours capturer la sortie d'erreur STDERR du shell et y chercher les messages d'erreur. Voir "Comment capturer la sortie de STDERR d'une commande externe ? ou utiliser IPC::Open3.

S'il n'y a pas de métacaractères dans l'argument de open(), Perl exécute la commande directement, sans utiliser le shell, et peut donc correctement rapporter la bonne exécution de la commande.

2.27 L'utilisation des apostrophes inversées dans un contexte vide pose-t-elle problème ?

À strictement parler, non. Par contre, stylistiquement parlant, ce n'est pas un bon moyen d'écrire du code maintenable. Perl propose plusieurs opérateurs permettant l'exécution de commandes externes. Les apostrophes inversées en sont un : elles capturent la sortie produite par la commande pour que vous l'utilisiez dans votre programme. La fonction `system` en est un autre : elle ne fait pas cette capture.

Considérons la ligne suivante :

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

On a oublié de vérifier la valeur de `$?` pour savoir si le programme s'était bien déroulé. Même si l'on avait écrit :

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

ce code pourrait et devrait probablement être écrit ainsi :

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

Ce qui permet d'avoir la sortie rapidement (au fur et à mesure de sa génération, au lieu d'avoir à attendre jusqu'à la fin) et vérifie aussi la valeur de sortie.

Avec system() vous avez aussi un contrôle direct sur une possible interprétation de méta-caractères, alors que ce n'est pas permis avec des apostrophes inversées.

2.28 Comment utiliser des apostrophes inversées sans traitement du shell ?

C'est un peu tordu. Vous ne pouvez pas simplement écrire :

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

Depuis Perl 5.8.0, vous pouvez utiliser open() avec sa syntaxe à plusieurs arguments. De la même manière que pour les appels avec plusieurs arguments des fonctions system() et exec(), il n'y a alors plus de passage pas le shell.

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

Vous pouvez aussi faire :

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

De même qu'avec system(), il n'y a aucune intervention du shell lorsqu'on donne une liste d'arguments à exec(). D'autres exemples de ceci se trouvent dans Ouvertures Sûres d'un Tube in *perlipc*.

Notez que si vous utilisez Microsoft, aucune solution à ce problème vexant n'est possible. Même si Perl pouvait émuler fork(), vous seriez toujours coincés, parce que Microsoft ne fournit aucune API du style argc/argv.

2.29 Pourquoi mon script ne lit-il plus rien de STDIN après que je lui ai envoyé EOF (^D sur Unix, ^Z sur MS-DOS) ?

Certaines implémentations de stdio positionnent des indicateurs d'erreur et de fin de fichier qui demandent à être annulées d'abord. Le module POSIX définit clearerr() que vous pouvez utiliser. C'est la façon correcte de traiter ce problème. Voici d'autres alternatives, moins fiables :

1. Gardez sous la main la position dans le fichier, et retournez-y, ainsi :

   ```
   $where = tell(LOG);
   seek(LOG, $where, 0);
   ```

2. Si cela ne marche pas, essayez de retourner à d'autres parties du fichier, et puis finalement là où vous le voulez.
3. Si cela ne marche toujours pas, essayez d'aller ailleurs dans le fichier, de lire quelque chose, puis de retourner à l'endroit mémorisé.
4. Si cela ne marche toujours pas, abandonnez votre librairie stdio et utilisez sysread directement.

2.30 Comment convertir mon script shell en perl ?

Apprenez Perl et réécrivez le. Sérieusement, il n'y a pas de convertisseur simple. Des choses compliquées en shell sont simples en Perl, et cette complexité même rendrait l'écriture d'un convertisseur shell->perl quasi impossible. En réécrivant le programme, vous penserez plus à ce que vous désirez réellement faire, et vous vous affranchirez du paradigme de flots de données mis bout à bout propre au shell, qui, bien que pratique pour certains problèmes, est à l'origine de nombreuses inefficacités.

2.31 Puis-je utiliser perl pour lancer une session telnet ou ftp ?

Essayez les modules Net::FTP, TCP::Client et Net::Telnet (disponibles sur CPAN). Pour émuler le protocole telnet, on s'aidera de http://www.cpan.org/scripts/netstuff/telnet.emul.shar, mais il est probablement plus facile d'utiliser Net::Telnet directement.

Si tout ce que vous voulez faire est prétendre d'être telnet mais n'avez pas du tout besoin de la négociation initiale, alors l'approche classique bi-processus suffira :

```
use IO::Socket;             # nouveau dans 5.004
$handle = IO::Socket::INET->new('www.perl.com:80')
        || die "can't connect to port 80 on www.perl.com: $!";
$handle->autoflush(1);
if (fork()) {               # XXX: undef signifie un échec
    select($handle);
    print while <STDIN>;    # tout ce qui vient de stdin va vers la prise
} else {
    print while <$handle>;  # tout ce qui vient de la prise va vers stdout
}
close $handle;
exit;
```

2.32 Comment écrire "expect" en Perl ?

Il était une fois une librairie appelée chat2.pl (incluse dans la distribution standard de perl) qui ne fut jamais vraiment terminée. Si vous la trouvez quelque part, *ne l'utilisez pas*. De nos jours, il est plus rentable de regarder du côté du module Expect, disponible sur CPAN, qui requiert aussi deux autres modules de CPAN, IO::Pty and IO::Stty.

2.33 Peut-on cacher les arguments de perl sur la ligne de commande aux programmes comme "ps" ?

Tout d'abord, notez que si vous faites cela pour des raisons de sécurité (afin d'empêcher les autres de voir des mots de passe, par exemple), alors vous feriez mieux de réécrire votre programme de façon à assurer que les données critiques ne sont jamais passées comme argument. Cacher ces arguments ne rendra jamais votre programme complètement sûr.

Pour vraiment altérer la ligne de commande visible, on peut assigner quelque chose à la variable $0, ainsi que documenté dans *perlvar*. Cependant, cela ne marchera pas sur tous les systèmes d'exploitation. Des logiciels démons comme sendmail y placent leur état, comme dans :

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

2.34 J'ai {changé de répertoire, modifié mon environnement} dans un script perl. Pourquoi les changements disparaissent-ils lorsque le script se termine ? Comment rendre mes changements visibles ?

Unix

À proprement parler, ce n'est pas possible – le script s'exécute dans un processus différent du shell qui l'a démarré. Les changements effectués dans ce processus ne sont pas transmis à son parent – seulement à ses propres enfants crées après le changement en question. Il y a cependant une astuce du shell qui peut permettre de le simuler en eval()uant la sortie du script dans votre shell ; voir la FAQ de comp.unix.questions pour plus de détails.

2.35 Comment fermer le descripteur de fichier attaché à un processus sans attendre que ce dernier se termine ?

En supposant que votre système autorise ce genre de chose, il suffit d'envoyer un signal approprié à ce processus (voir kill in *perlfunc*). Il est d'usage d'envoyer d'abord un signal TERM, d'attendre un peu, et ensuite seulement d'envoyer le signal KILL pour y mettre fin.

2.36 Comment lancer un processus démon ?

Si par processus démon vous entendez un processus qui s'est détaché (dissocié de son terminal de contrôle), alors le mode opératoire suivant marche en général sur la plupart des systèmes Unix. Les utilisateurs de plate-forme non-Unix doivent regarder du côté du module Votre_OS::Process pour des solutions alternatives.

– Ouvrir /dev/tty et utiliser l'ioctl TIOCNOTTY dessus. Voir *tty* pour les détails. Ou mieux encore, utiliser la fonction POSIX::setsid(), pour ne pas avoir à se soucier des groupes de processus.
– Positionner le répertoire courant à /
– Re-ouvrir STDIN, STDOUT et STDERR pour qu'ils ne soient plus attachés à leur terminal d'origine.
– Se placer en arrière plan ainsi :

```
fork && exit;
```

Le module Proc::Daemon, disponible sur le CPAN, fournit une fonction qui réalise ces actions pour vous.

2.37 Comment savoir si je tourne de façon interactive ou pas ?

Bonne question. Parfois `-t STDIN` et `-t STDOUT` peuvent donner quelque indice, parfois non.

```
if (-t STDIN && -t STDOUT) {
    print "Now what? ";
}
```

Sur les systèmes POSIX, on peut tester son groupe de processus pour voir s'il correspond au groupe du terminal de contrôle comme suit :

```
use POSIX qw/getpgrp tcgetpgrp/;
open(TTY, "/dev/tty") or die $!;
$tpgrp = tcgetpgrp(fileno(*TTY));
$pgrp = getpgrp();
if ($tpgrp == $pgrp) {
    print "foreground\n";
} else {
    print "background\n";
}
```

2.38 Comment sortir d'un blocage sur événement lent ?

Utiliser la fonction alarm(), probablement avec le recours à une capture de signal, comme documenté dans Signaux in *perlipc* et le chapitre "Signaux" du Camel Book. On peut aussi utiliser le module plus flexible Sys::AlarmCall, disponible sur CPAN.

La fonction alarm() n'est pas disponible dans certaines versions de Windows. Pour le savoir, cherchez dans le documentation de votre distribution Perl.

2.39 Comment limiter le temps CPU ?

Utiliser le module BSD::Resource de CPAN.

2.40 Comment éviter les processus zombies sur un système Unix ?

Utiliser le collecteur de Signaux in *perlipc* pour appeler wait() lorsque le signal SIGCHLD est reçu, ou bien utiliser la technique du double fork() décrite dans Comment lancer un processus en arrière plan ? in *perlfaq8*.

2.41 Comment utiliser une base de données SQL ?

Le module DBI fournit une interface générique vers la plupart des serveurs ou des systèmes de base de données tels que Oracle, DB2, Sybase, mysql, Postgresql, ODBC ou même des fichiers classiques. Le module DBI accède à ces différents types de base de données à travers un pilote spécifique à chaque base de données, ou DBD (Data Base Driver). Vous pouvez obtenir la liste complète des pilotes disponibles sur CPAN en consultant http://www.cpan.org/modules/by-module/DBD/. Lisez http://dbi.perl.org pour en apprendre plus concernant DBI.

D'autres modules donnent accès à des bases spécifiques : Win32::ODBC, Alzabo, iodbc et d'autres encore trouvables en cherchant sur CPAN : http://search.cpan.org.

2.42 Comment terminer un appel à system() avec un control-C ?

Ce n'est pas possible. Vous devez imiter l'appel à system() vous-même (voir *perlipc* pour un exemple de code) et ensuite fournir votre propre routine de traitement pour le signal INT qui passera le signal au sous-processus. Ou vous pouvez vérifier si ce signal a été reçu :

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

2.43 Comment ouvrir un fichier sans bloquer ?

Si vous avez assez de chance pour utiliser un système supportant les lectures non bloquantes (ce qui est le cas de la plupart des systèmes Unix), vous devez simplement utiliser les attributs O_NDELAY ou O_NONBLOCK du module Fcntl en les spécifiant à sysopen() :

```
use Fcntl;
sysopen(FH, "/foo/somefile", O_WRONLY|O_NDELAY|O_CREAT, 0644)
  or die "can't open /foo/somefile: $!";
```

2.44 Comment faire la différence entre les erreurs shell et les erreurs perl ?

(contribution de brian d foy, <bdfoy@cpan.org>)

Lorsque vous lancez un script Perl, il y a quelque chose qui exécute le script pour vous et ce quelque chose peut émettre des messages d'erreur. Le script lui-même peut émettre ses propres messages d'avertissement et d'erreur. La plupart du temps, vous ne pouvez pas savoir lequel a émis quoi.

Vous ne pouvez sans doute pas modifier la chose qui exécute perl, mais vous pouvez modifier la manière dont perl affiche ses avertissements et ses erreurs en définissant vos propres fonctions de gestion de ces messages.

Considérons le script suivant, qui contient une erreur pas visiblement évidente :

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

print "Hello World\n";
```

Lorsque je tente d'exécuter ce script depuis mon shell (il se trouve que c'est bash), j'obtiens une erreur. Il semblerait que perl à oublié sa fonction print() mais en fait c'est ma ligne de shebang (#!...) qui ne contient pas le chemin d'accès à perl. Donc le shell lance le script et j'obtiens l'erreur :

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

Une correction sale et rapide nécessite un petit peu plus de code mais vous aidera à localiser l'origine du problème.

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

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

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

Tous les message de perl seront maintenant préfixés par "Perl: ". Le bloc BEGIN fonctionne dès la compilation et donc tous les messages d'erreur et d'avertissement du compilateur seront eux aussi préfixés par "Perl: ".

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

Si vous ne voyez pas "Perl: " devant le message, c'est qu'il ne vient pas de perl.

Vous pourriez aussi tout simplement connaître toutes les erreurs de perl mais, bien que ce soit effectivement le cas de certaines personnes, ce n'est certainement pas le vôtre. En revanche, elles sont toutes répertoriées dans *perldiag*. Donc si vous n'y trouvez pas cette erreur alors ce que ce n'est probablement pas une erreur perl.

Chercher parmi tous les messages d'erreur n'est pas facile alors laissez donc perl le faire pour vous. Utilisez la directive diagnostics qui transforme les messages d'erreurs normaux de perl en un discours un peu plus long sur le sujet.

```
use diagnostics;
```

Si vous n'obtenez pas un ou deux paragraphes d'explication, il y a de grande chance que ce ne soit pas une erreur perl.

2.45 Comment installer un module du CPAN ?

La façon la plus simple est d'utiliser un module lui aussi appelé CPAN qui le fera pour vous. Ce module est distribué en standard avec les versions de perl 5.004 ou plus.

```
$ perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.59_54)
ReadLine support enabled

cpan> install Some::Module
```

Pour installer manuellement le module CPAN, ou tout autre module de CPAN qui se comporte normalement, suivez les étapes suivantes :

1. Désarchiver les sources dans une zone temporaire.
2. `perl Makefile.PL`
3. `make`
4. `make test`
5. `make install`

Si votre version de perl est compilée sans support pour le chargement dynamique de librairies, alors il vous faudra remplacer l'étape 3 (**make**) par **make perl** et vous obtiendrez un nouvel exécutable *perl* avec votre extension liée statiquement.

Voir *ExtUtils::MakeMaker* pour plus de détails sur les extensions de fabrication. Voir aussi la question suivante, Quelle est la différence entre require et use ? (§2.46).

2.46 Quelle est la différence entre require et use ?

Perl offre différentes solutions pour inclure du code d'un fichier dans un autre. Voici les différences entre les quelques constructions disponibles pour l'inclusion :

```
1)  "do $file" ressemble à "eval `cat $file`", sauf que le premier
    1.1: cherche dans @INC et met à jour %INC.
    1.2: entoure le code eval()ué d'une portée lexicale *indépendante*.

2)  "require $file" ressemble à "do $file", sauf que le premier
    2.1: s'arrange pour éviter de charger deux fois un même fichier.
    2.2: lance une exception si la recherche, la compilation ou
    l'exécution de $file échoue.
```

```
3)  "require Module" ressemble à "require 'Module.pm'", sauf que le premier
    3.1: traduit chaque "::" en votre séparateur de répertoire.
    3.2: indique au compilateur que Module est une classe,
    passible d'appels indirects.

4)  "use Module" ressemble à "require Module", sauf que le premier
    4.1: charge le module à la phase de compilation, et non à l'exécution.
    4.2: importe ses symboles et sémantiques dans la paquetage courant.
```

En général, on utilise `use` avec un module Perl adéquat.

2.47 Comment gérer mon propre répertoire de modules/bibliothèques ?

Lorsque vous fabriquez les modules, utilisez les options PREFIX et LIB à la phase de génération de Makefiles :

```
perl Makefile.PL PREFIX=/mydir/perl LIB=/mydir/perl/lib
```

puis, ou bien positionnez la variable d'environnement PERL5LIB avant de lancer les scripts utilisant ces modules/bibliothèques (voir *perlrun*), ou bien utilisez :

```
use lib '/mydir/perl/lib';
```

C'est presque la même chose que :

```
BEGIN {
    unshift(@INC, '/mydir/perl/lib');
}
```

sauf que le module lib vérifie les sous-répertoires dépendants de la machine. Voir le pragma *lib* de Perl pour plus d'information.

2.48 Comment ajouter le répertoire dans lequel se trouve mon programme dans le chemin de recherche des modules / bibliothèques ?

```
use FindBin;
use lib "$FindBin::Bin";
use vos_propres_modules;
```

2.49 Comment ajouter un répertoire dans mon chemin de recherche (@INC) à l'exécution ?

Voici les moyens suggérés de modifier votre chemin de recherche :

```
la variable d'environnement PERLLIB
la variable d'environnement PERL5LIB
l'option perl -Idir sur la ligne de commande
le pragma use lib, comme dans
    use lib "$ENV{HOME}/ma_propre_biblio_perl";
```

Ce dernier est particulièrement utile, parce qu'il prend en compte les fichiers propres à une architecture donnée. Le module pragmatique lib.pm a été introduit dans la version 5.002 de Perl.

2.50 Qu'est-ce que socket.ph et où l'obtenir ?

C'est un fichier à la mode de perl4 définissant des contantes pour la couche réseau du système. Il est parfois construit en utilisant h2ph lorsque Perl est installé, mais d'autres fois il ne l'est pas. Les programmes modernes utilisent `use Socket;` à la place.

3 AUTEURS

4 TRADUCTION

4.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

4.2 Traducteur

Traduction initiale : Raphaël Manfredi <Raphael_Manfredi@grenoble.hp.com>. Mise à jour : Roland Trique <roland.trique@uhb.fr>, Paul Gaborit (paul.gaborit at enstimac.fr).

4.3 Relecture

Simon Washbrook, Gérard Delafond.

5 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.

Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la version PDF.