perlfaq5

Table des matières

1 NAME/NOM

perlfaq5 - Fichiers et formats

2 DESCRIPTION

Cette section traite de E/S (Entrées et Sorties) et autres éléments connexes : descripteurs de fichiers, vidage de tampons, formats d'écriture et pieds de page.

2.1 Comment vider ou désactiver les tampons en sortie ? Pourquoi m'en soucier ?

Perl ne propose pas vraiment de sorties sans tampon (sauf en passant par `syswrite(OUT, $char, 1)`) mais plutôt une sorte de tampon par commande où une écriture réelle est effectuée après chaque commande de sortie.

La bibliothèque standard d'E/S du C (stdio) accumule normalement les caractères envoyés aux divers périphériques dans des tampons dans le but d'éviter d'effectuer un appel système pour chaque octet. Dans la plupart des implémentations de stdio, le type d'accumulation en sortie et la taille des tampons varient suivant le périphérique utilisé. Les fonctions Perl print() et write() utilisent normalement ces tampons alors que syswrite() passe outre.

Si vous souhaitez que vos sorties soient envoyées immédiatement lorsque vous effectuez un print() ou un write() (par exemple, dans le cas de certains protocoles réseau), il vous faudra activer le mode d'écriture systématique (autoflush) des tampons attachés au descripteur de fichier. Ce mode est représenté par la variable Perl $| et si elle contient une valeur vraie, Perl videra le tampon du descripteur de fichier après chaque appel à print() ou write(). Une modification de $| modifie la gestion des tampons du descripteur de fichier sélectionné par défaut. Vous pouvez choisir ce descripteur de fichier via un appel à la fonction select() avec un seul argument (voir perlvar|$| et select in *perlfunc*).

Utilisez select() pour choisir le bon descripteur de fichier puis positionnez les variables de pilotage.

```
$old_fh = select(OUTPUT_HANDLE);
$| = 1;
select($old_fh);
```

Ou, de façon plus idiomatique, en une seule instruction :

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

$| = 1, select $_ for select OUTPUT_HANDLE;
```

Certains modules proposent un accès orienté objet aux descripteurs de fichiers et à leur paramètres (ils seront sans doute un peu lourd si vous ne leur demandez que cela). Par exemple IO::Handle :

```
use IO::Handle;
open(DEV, ">/dev/printer");   # à quoi ça sert ?
DEV->autoflush(1);
```

Ou IO::Socket :

```
use IO::Socket;   # une sorte de tube ?
my $sock = IO::Socket::INET->new( 'www.example.com:80' );

$sock->autoflush();
```

2.2 Comment changer une ligne, effacer une ligne, insérer une ligne au milieu ou ajouter une ligne en tête d'un fichier ?

Utiliser le module Tie::File qui fait partie de la distribution standard depuis Perl 5.8.0.

2.3 Comment déterminer le nombre de lignes d'un fichier ?

Un moyen assez efficace est de compter les caractères de fin de ligne dans le fichier. Le programme suivant utilise une propriété de tr///, décrite dans *perlop*. Si votre fichier de texte ne se termine pas par un caractère de fin de ligne, alors ce n'est pas vraiment un fichier de texte correct, et ce programme vous surprendra en indiquant une ligne de moins.

```
$lines = 0;
open(FILE, $filename) or die "Can't open `$filename': $!";
while (sysread FILE, $buffer, 4096) {
    $lines += ($buffer =~ tr/\n//);
}
close FILE;
```

On supposera qu'il n'y a aucune traduction parasite des caractères de fin de ligne à déplorer.

2.4 Comment utiliser l'option `-i` de Perl depuis l'intérieur d'un programme ?

L'option `-i` modifie la valeur de la variable Perl `$^I` qui modifie le comportement de <> ; voir *perlrun* pour plus de détails. En modifiant directement les bonnes variables, vous pouvez obtenir le même comportement dans votre programme. Par exemple :

```
# ...
{
  local($^I, @ARGV) = ('.orig', glob("*.c"));
  while (<>) {
    if ($. == 1) {
      print "Cette ligne sera en tete de chaque fichier\n";
    }
    s/\b(p)earl\b/${1}erl/i; # Correction en tenant
                             # compte de la casse
    print;
    close ARGV if eof;       # Réinitialise $.
  }
}
# $^I et @ARGV reprennent ici leur ancienne valeur
```

Ce bloc de code modifie tous les fichier `.c` du répertoire courant en créant une copie de sauvegarde de chaque fichier original dans un fichier `.c.orig`.

2.5 Comment puis-je copier un fichier ?

(contribution de brian d foy)

Utilisez le module File::Copy. Il vient avec Perl et sait faire de vraies copies entre différents systèmes de fichiers, tout cela de manière portable.

```
use File::Copy;

copy( $original, $new_copy ) or die "Échec de la copie : $!";
```

Si vous ne pouvez pas utiliser File::Copy, vous devrez faire le travail vous-même : ouvrez le fichier original, ouvrez le fichier de destination puis écrivez dans le fichier de destination au fur et à mesure que vous lisez le fichier original.

2.6 Comment créer un fichier temporaire ?

Si vous n'avez pas besoin de connaître le nom du fichier temporaire, vous pouvez utiliser `open()` en passant `undef` à la place d'un nom de fichier. Le fonction `open()` créera alors un fichier temporaire anonyme.

```
open my $tmp, '+>', undef or die $!;
```

Sinon vous pouvez utiliser le module File::Temp.

```
use File::Temp qw/ tempfile tempdir /;

$dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );

# ou si vous n'avez pas besoin du nom du fichier

$fh = tempfile( DIR => $dir );
```

Le module File::Temp est un module standard depuis Perl 5.6.1. Si vous ne disposez pas d'une version moderne de Perl, utilisez la méthode de classe `new_tmpfile` du module IO::File pour obtenir un descripteur de fichier ouvert en lecture écriture. À utiliser si le nom dudit fichier importe peu.

```
use IO::File;
$fh = IO::File->new_tmpfile()
   or die "Impossible de créer un fichier temporaire : $!";
```

Si vous tenez absolument à tout faire vous-même, utilisez l'ID du processus et/ou la valeur du compteur de temps. Si vous avez besoin de plusieurs fichiers temporaires, ayez recours à un compteur :

```
BEGIN {
    use Fcntl;
    my $temp_dir = -d '/tmp' ? '/tmp' : $ENV{TMPDIR} || $ENV{TEMP};
    my $base_name = sprintf("%s/%d-%d-0000", $temp_dir, $$, time());
    sub temp_file {
        local *FH;
        my $count = 0;
        until (defined(fileno(FH)) || $count++ > 100) {
            $base_name =~ s/-(\d+)$/"-" . (1 + $1)/e;
            # O_EXCL est indispensable pour la sécurité.
            sysopen(FH, $base_name, O_WRONLY|O_EXCL|O_CREAT);
        }
        if (defined(fileno(FH))
            return (*FH, $base_name);
        } else {
            return ();
        }
    }
}
```

2.7 Comment manipuler un fichier avec des enregistrements de longueur fixe ?

Le plus efficace est d'utiliser pack() et unpack(). C'est plus rapide que d'utiliser substr() sur de très nombreuses chaînes. C'est plus lent pour seulement quelques unes.

Voici un morceau de code démontrant comment découper et ensuite réassembler des lignes formattées selon un schéma donné, ici la sortie du programme ps, version Berkeley :

```
# exemple de ligne à traiter
#   15158 p5  T      0:00 perl /home/ram/bin/scripts/now-what
my $PS_T = 'A6 A4 A7 A5 A*';
open my $ps, '-|', 'ps';
print scalar <$ps>;
my @fields = qw( pid tt stat time command );
while (<$ps>) {
    my %process;
    @process{@fields} = unpack($PS_T, $_);
    for my $field ( @fields ) {
        print "$field: <$process{$field}>\n";
    }
    print 'line=', pack($PS_T, @process{@fields} ), "\n";
}
```

Nous avons utilisé une tranche de table de hachage afin de gérer facilement les champs de chaque ligne. Le stockage des clés dans un tableau permet de les utiliser facilement toutes ensemble et de leur appliquer des boucles. Cela évite aussi de polluer le programme avec des variables globales ou d'utiliser des références symboliques.

2.8 Comment rendre un descripteur de fichier local à une routine ? Comment passer des descripteurs à d'autres routines ? Comment construire un tableau de descripteurs ?

Depuis perl 5.6, open() autovivifie les descripteurs de fichier ou de répertoire en tant que référence si vous lui passez une variable scalaire non définie. Vous pouvez passer ces références à d'autres routines comme n'importe quel autre scalaire et les utiliser à la place des noms de descripteur.

```
open my    $fh, $file_name;

open local $fh, $file_name;

print $fh "Hello World!\n";

process_file( $fh );
```

Avant perl 5.6, il fallait jongler avec différents formes idiomatiques liées aux types universels (typeglob). Vous les rencontrerez dans d'anciens codes.

```
open FILE, "> $filename";
process_typeglob(   *FILE );
process_reference( \*FILE );

sub process_typeglob  { local *FH = shift; print FH  "Typeglob!" }
sub process_reference { local $fh = shift; print $fh "Reference!" }
```

Si vous voulez créer plusieurs descripteurs anonymes, vous devriez jeter un oeil aux modules Symbol et IO::Handle.

2.9 Comment utiliser un descripteur de fichier indirectement ?

Un descripteur de fichier indirect s'utilise par l'intermédiaire d'une variable placée là où, normalement, le langage s'attend à trouver un descripteur de fichier. On obtient une telle variable ainsi :

```
  $fh =   SOME_FH;       # un mot brut est mal-aimé de 'strict subs'
  $fh =  "SOME_FH";      # mal-aimé de 'strict refs'; même package seulement
  $fh =  *SOME_FH;       # type universel
  $fh = \*SOME_FH;       # réference sur un type universel (bénissable)
  $fh =  *SOME_FH{IO};   # IO::Handle béni du type universel *SOME_FH
```

Ou en utilisant la méthode `new` de l'un des modules IO::* pour créer un descripteur anonyme, l'affecter dans une variable scalaire, puis l'utiliser ensuite comme si c'était un descripteur de fichier normal.

```
use IO::Handle;                     # 5.004 ou mieux
$fh = IO::Handle->new();
```

Vous pouvez alors utiliser ces objets comme un descripteur de fichier normal. Partout où Perl s'attend à trouver un descripteur de fichier, un descripteur indirect peut être substitué. Ce descripteur indirect est tout simplement une variable scalaire contenant un descripteur de fichier. Des fonctions comme `print`, `open`, `seek` ou l'opérateur diamant `<FH>` acceptent soit un descripteur de fichier sous forme de nom soit une variable scalaire contenant un descripteur :

```
($ifh, $ofh, $efh) = (*STDIN, *STDOUT, *STDERR);
print $ofh "Type it: ";
$got = <$ifh>;
print $efh "What was that: $got";
```

Quand on veut passer un descripteur de fichier à une fonction, il y a deux manières d'écrire la routine :

```
sub accept_fh {
    my $fh = shift;
    print $fh "Sending to indirect filehandle\n";
}
```

Ou il est possible de localiser un type universel (typeglob) et d'utiliser le nom de descripteur ainsi obtenu directement :

```
sub accept_fh {
    local *FH = shift;
    print  FH "Sending to localized filehandle\n";
}
```

Ces deux styles marchent aussi bien avec des objets, des types universels ou des descripteurs de fichiers réels. (Ils pourraient aussi se contenter de chaînes simples, dans certains cas, mais c'est plutôt risqué.)

```
accept_fh(*STDOUT);
accept_fh($handle);
```

Dans les exemples ci-dessus, nous avons affecté le descripteur de fichier à une variable scalaire avant de l'utiliser. La raison est que seules de simples variables scalaires, par opposition à des expressions ou des notations indicées dans des tableaux normaux ou associatifs, peuvent être ainsi utilisées avec des fonctions natives comme `print`, `printf`, ou l'opérateur diamant. Les exemples suivant sont invalides et ne passeront pas la phase de compilation :

```
@fd = (*STDIN, *STDOUT, *STDERR);
print $fd[1] "Type it: ";                   # INVALIDE
$got = <$fd[0]>                             # INVALIDE
print $fd[2] "What was that: $got";         # INVALIDE
```

Avec `print` et `printf`, on peut s'en sortir avec un bloc contenant une expression à la place du descripteur de fichier normalement attendu :

```
print  { $fd[1] } "funny stuff\n";
printf { $fd[1] } "Pity the poor %x.\n", 3_735_928_559;
# On obtient dans $fd[1] : Pity the poor deadbeef.
```

Ce bloc est un bloc ordinaire, semblable à tout autre, donc on peut y placer des expressions plus complexes. Ceci envoie le message vers une destination parmi deux :

```
$ok = -x "/bin/cat";
print { $ok ? $fd[1] : $fd[2] } "cat stat $ok\n";
print { $fd[ 1+ ($ok || 0) ]  } "cat stat $ok\n";
```

Cette façon de traiter `print` et `printf` comme si c'étaient des appels à des méthodes objets ne fonctionne pas avec l'opérateur diamant. Et ce parce que c'est vraiment un opérateur et pas seulement une fonction avec un argument spécial, non délimité par une virgule. En supposant que l'on ait stocké divers types universels dans une structure, comme montré ci-avant, on peut utiliser la fonction native `readline` pour lire un enregistrement comme le ferait `<>`. Avec l'initialisation montrée ci-dessus pour @fd, cela marcherait, mais seulement parce que readline() demande un type universel. Cela ne marchera pas avec des objets ou des chaînes, ce qui pourrait bien être un de ces bugs non encore corrigés.

```
$got = readline($fd[0]);
```

Notons ici que cet exotisme des descripteurs indirects ne dépend pas du fait qu'ils peuvent prendre la forme de chaînes, types universels, objets, ou autres. C'est simplement dû à la syntaxe des opérateurs fondamentaux. Jouer à l'orienté objet ne serait d'aucune aide ici.

2.10 Comment mettre en place un pied-de-page avec write() ?

Il n'y a pas de méthode native pour accomplir cela, mais *perlform* indique une ou deux techniques qui permettent aux programmeurs intrépides de s'en sortir.

2.11 Comment rediriger un `write()` dans une chaîne ?

Voir Accéder aux formats de l'intérieur in *perlform* pour un exemple de fonction swrite().

2.12 Comment afficher mes nombres avec des virgules pour délimiter les milliers ?

(contribution de brian d foy et de Benjamin Goldberg)

Vous pouvez utiliser le module *Number::Format* pour délimiter vos nombres. Ce module tient compte des informations fournies par les 'locale' pour ceux qui voudraient utiliser un espace comme délimiteur (ou n'importe quel autre caractère).

La routine suivante ajoute des virgules dans un nombre :

```
sub commify {
    local $_  = shift;
    1 while s/^([-+]?\d+)(\d{3})/$1,$2/;
    return $_;
}
```

Cette expression rationnelle de Benjamin Goldberg ajoute des virgules aux nombres :

```
s/(^[-+]?\d+?(?=(?>(?:\d{3})+)(?!\d))|\G\d{3}(?=\d))/$1,/g;
```

Elle est plus lisible avec des commentaires :

```
s/(
    ^[-+]?            # début d'un nombre.
    \d+?              # les premiers chiffres avant une virgule
    (?=               # suivis par,
                      # (sans faire partie de ce qui est reconnu)
       (?>(?:\d{3})+) # un ou plusieurs groupes de 3 chiffres
       (?!\d)         # un multiple *exact* et non x * 3 + 1 ou autre.
    )
   |                  # ou :
    \G\d{3}           # le dernier groupe avec 3 chiffres
    (?=\d)            # mais sans aucun chiffre ensuite.
)/$1,/xg;
```

2.13 Comment traduire les tildes (~) dans un nom de fichier ?

Utiliser l'opérateur <> (appelé glob()), tel que décrit dans *perlfunc*. Les versions anciennes de Perl nécessitaient la présence d'un shell qui comprenne les tildes. Les versions récentes de Perl gèrent cette fonction nativement. Le module Glob::KGlob (disponible sur CPAN) implémente une fonctionnalité de glob plus portable.

En Perl, on peut utiliser ceci directement :

```
$filename =~ s{
  ^ ~             # cherche le tilde en tête
  (               # sauvegarde dans $1 :
      [^/]        # tout caractère sauf un slash
           *      # et ce 0 ou plusieurs fois (0 pour mon propre login)
  )
}{
  $1
      ? (getpwnam($1))[7]
      : ( $ENV{HOME} || $ENV{LOGDIR} )
}ex;
```

2.14 Pourquoi les fichiers que j'ouvre en lecture-écriture se voient-ils effacés ?

Parce que vous faites quelque chose du genre indiqué ci-après, qui tronque d'abord le fichier et *seulement ensuite* donnne un accès en lecture-écriture :

```
open(FH, "+> /path/name");          # MAUVAIS (en général)
```

Il faudrait faire comme ceci, ce qui échouera si le fichier n'existe pas déjà.

```
open(FH, "+< /path/name");          # ouvert pour mise à jour
```

L'utilisation de ">" crée ou met toujours à zéro. L'utilisation de "<" ne le fait jamais. Le "+" n'y change rien.
Voici différents exemples d'ouverture. Tous ceux qui utilisent sysopen() supposent que l'on a déjà fait :

```
use Fcntl;
```

Pour ouvrir un fichier en lecture :

```
open(FH, "< $path")                                 || die $!;
sysopen(FH, $path, O_RDONLY)                        || die $!;
```

Pour ouvrir un fichier en écriture, en le créant si c'est un nouveau fichier ou en le tronquant s'il est préexistant :

```
open(FH, "> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT)        || die $!;
sysopen(FH, $path, O_WRONLY|O_TRUNC|O_CREAT, 0666)  || die $!;
```

Pour ouvrir un fichier en écriture, en créant un fichier qui n'existe pas déjà :

```
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT)         || die $!;
sysopen(FH, $path, O_WRONLY|O_EXCL|O_CREAT, 0666)   || die $!;
```

Pour ouvrir un fichier avec ajout en fin, en le créant si nécessaire :

```
open(FH, ">> $path") || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT)       || die $!;
sysopen(FH, $path, O_WRONLY|O_APPEND|O_CREAT, 0666) || die $!;
```

Pour ouvrir un fichier existant avec ajout en fin :

```
sysopen(FH, $path, O_WRONLY|O_APPEND)               || die $!;
```

Pour ouvrir un fichier existant en mode de mise à jour :

```
open(FH, "+< $path")                                || die $!;
sysopen(FH, $path, O_RDWR)                          || die $!;
```

Pour ouvrir un fichier en mode de mise à jour, avec création si besoin :

```
sysopen(FH, $path, O_RDWR|O_CREAT)                  || die $!;
sysopen(FH, $path, O_RDWR|O_CREAT, 0666)            || die $!;
```

Pour ouvrir en mise à jour un fichier qui n'existe pas déjà :

```
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT)           || die $!;
sysopen(FH, $path, O_RDWR|O_EXCL|O_CREAT, 0666)     || die $!;
```

Enfin, pour ouvrir un fichier sans bloquer, avec création éventuelle :

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

Attention : ni la création, ni la destruction de fichier n'est garantie être atomique à travers NFS. C'est-à-dire que deux processus pourraient simultanément arriver à créer ou à effacer le même fichier sans erreur. En d'autres termes, O_EXCL n'est pas aussi exclusif que ce que l'on pourrait penser de prime abord.
Voir aussi la nouvelle page de documentation *perlopentut* si vous en disposez (à partir de la version 5.6).

2.15 Pourquoi <*> donne de temps en temps l'erreur "Argument list too long" ?

L'opérateur <> est utilisé pour faire appel à glob() (cf. ci-dessus). Dans les versions de Perl précédant la v5.6.0, l'opérateur glob() interne lance csh(1) pour effectuer ladite complétion, mais csh ne peut pas traiter plus de 127 éléments et retourne donc le message d'erreur `Argument list too long`. Ceux qui ont installé tcsh à la place de csh n'auront pas ce problème, mais les utilisateurs de leur code pourront en être surpris.

Pour contourner cela, soit vous mettez Perl à jour vers une version 5.6.0 ou supérieur, soit vous faites votre complétion vous-même avec readdir() et des motifs, soit vous utilisez un module comme File::KGlob, qui n'a pas recours au shell pour faire cette complétion.

2.16 Y a-t-il une fuite / un bug avec glob() ?

De par son implémentation sur certains systèmes d'exploitation, l'utilisation de la fonction glob(), directement ou sous sa forme <> dans un contexte scalaire, peut provoquer une fuite mémoire ou un comportement non-prédictible. Il est donc préférable de n'utiliser glob() que dans un contexte de liste.

2.17 Commend ouvrir un fichier dont le nom commence par ">" ou se termine par des espaces ?

(contribution de Brian McCauley)

La fonction Perl open(), appelée dans sa forme spéciale à deux arguments, ignore les espaces en fin de nom de fichier et déduit le mode d'ouverture du fichier en se basant sur la présence de certains caractères au début du nom (ou d'un "|" en fin). Dans les anciennes versions de Perl, c'était le seul moyen d'utiliser la fonction open() si bien qu'on retrouve cette forme dans de nombreux codes ou livres anciens.

À moins d'avoir des raisons particulières d'utiliser cette forme d'appel à deux arguments, vous devriez utiliser l'appel à trois arguments qui ne traite aucun caractère comme étant spécial dans le nom du fichier.

```
open FILE, "<", "  fichier  ";  # le nom du fichier est "   fichier   "
open FILE, ">", ">fichier";     # le nom du fichier est  ">fichier"
```

2.18 Comment renommer un fichier de façon sûre ?

Si votre système d'exploitation fournit un programme mv(1) correct ou équivalent moral, ceci fontionnera :

```
rename($old, $new) or system("mv", $old, $new);
```

Pour être plus portable, il peut être intéressant d'utiliser le module File::Copy. On copie simplement le fichier sur le nouveau nom (en vérifiant bien les codes de retour de chaque fonction), puis on efface l'ancien nom. En revanche, cela n'a pas tout à fait la même sémantique que le véritable rename() qui, lui, préservera des méta-informations comme les droits, les dates diverses et autres informations liées au fichier.

Les versions récentes de File::Copy fournissent une fonction move().

2.19 Comment verrouiller un fichier ?

La fonction flock() fournie par Perl (cf. *perlfunc* pour plus de détails) appelle flock(2) si elle est disponible, sinon fcntl(2) (pour les versions de perl supérieure à 5.004) ou finalement lockf(3) si aucun des deux appels systèmes précédents n'est disponible. Sur certains systèmes, une forme native de verrouillage peut même être utilisée. Voici quelques avertissements relatifs à l'utilisation du flock() de Perl :

1. La fonction produit une erreur fatale si aucun des trois appels (ou proches équivalents) n'existe.
2. lockf(3) ne fournit pas de verrouillage partagé et impose que le descripteur de fichier soit ouvert au minimum en mode écriture (et donc aussi en mode ajout ou en mode lecture/écriture).

3. Certaines versions de flock() ne peuvent pas verrouiller de fichiers à travers un réseau (par exemple via NFS). En conséquence, vous devriez forcer Perl à utiliser fcntl(2) lors de sa compilation. Mais même ceci n'est pas sûr. Voir à ce sujet flock dans *perlfunc* et le fichier *INSTALL* dans la distribution source pour savoir comment procéder.

   Deux sémantiques potentielles de flock peu évidentes mais traditionnelles sont qu'il attend indéfiniment jusqu'à obtenir le verrouillage et qu'il verrouille *simplement pour information*. De tels verrouillages discrétionnaires sont plus flexibles, mais offrent des garanties moindres. Ceci signifie que les fichiers verrouillés avec flock() peuvent être modifiés par des programmes qui eux n'utilisent pas flock(). Les voitures qui respectent les feux de signalisation s'entendent bien entre elles, mais pas avec les voitures qui grillent les feux rouges. Voir *perlport*, la documentation spécifique à votre portage, ou vos pages de manuel locales spécifiques à votre système pour plus de détails. Il est conseillé de choisir un comportement traditionnel si vous écrivez des programmes portables (mais si ce n'est pas le cas, vous êtes absolument libre d'écrire selon les idiosyncrasies de votre propre système (parfois appelées des "caractéristiques"). L'adhérence aveugle aux soucis de portabilité ne devrait pas vous empêcher de faire votre boulot).

   Pour plus d'informations sur le verrouillage de fichiers, voir aussi Verrouillage de fichier in *perlopentut* si vous en disposez (à partir de la version 5.6).

2.20 Pourquoi ne pas faire simplement `open(FH, ">file.lock")` ?

Un bout de code **À NE PAS UTILISER** est :

```
sleep(3) while -e "file.lock";      # MERCI de NE PAS UTILISER
open(LCK, "> file.lock");           # ce code FOIREUX
```

C'est un cas classique de conflit d'accès (race condition) : on fait en deux temps quelque chose qui devrait être réalisé en une seule opération. C'est pourquoi les microprocesseurs fournissent une instruction atomique appelée test-and-set. En théorie, ceci devrait fonctionner :

```
sysopen(FH, "file.lock", O_WRONLY|O_EXCL|O_CREAT)
            or die "can't open  file.lock: $!";
```

sauf que, lamentablement, la création (ou l'effacement) n'est pas atomique à travers NFS, donc cela ne marche pas (du moins, pas tout le temps) à travers le réseau. De multiples schémas utilisant link() ont été suggérés, mais ils ont tous tendance à mettre en jeu une boucle d'attente active, ce qui est tout autant indésirable.

2.21 Je ne comprends toujours pas le verrouillage. Je veux seulement incrémenter un compteur dans un ficher. Comment faire ?

Ne vous a-t-on jamais expliqué que les compteurs d'accès aux pages web étaient inutiles ? Ils ne comptent pas vraiment le nombre d'accès, ils sont une perte de temps et ils ne servent qu'à gonfler la vanité de l'auteur. Il vaudrait mieux choisir un nombre au hasard. Ce serait plus réaliste.

Quoiqu'il en soit, voici ce que vous pouvez faire si vous ne pouvez vraiment pas vous en empêcher :

```
use Fcntl ':flock';
sysopen(FH, "numfile", O_RDWR|O_CREAT)    or die "can't open numfile: $!";
flock(FH, LOCK_EX)                        or die "can't flock numfile: $!";
$num = <FH> || 0;
seek(FH, 0, 0)                            or die "can't rewind numfile: $!";
truncate(FH, 0)                           or die "can't truncate numfile: $!";
(print FH $num+1, "\n")                   or die "can't write numfile: $!";
close FH                                  or die "can't close numfile: $!";
```

Voici un bien meilleur compteur d'accès aux pages web :

```
$hits = int( (time() - 850_000_000) / rand(1_000) );
```

À défaut de la valeur du compteur lui-même, ce code pourrait impressionner vos amis... :-)

2.22 Je souhaite juste ajouter un peu de texte à la fin d'un fichier. Dois-je tout de même utiliser le verrouillage ?

Si vous utilisez un système qui implémente correctement flock() et si vous utilisez l'exemple de code d'ajout proposé par "perldoc -f flock", tout devrait bien se passer même si votre système ne gère pas bien le mode d'ajout (si un tel système existe). Donc, si vous vous limitez aux systèmes qui implémentent flock() (et ce n'est pas vraiment une limitation), tout ira bien.

Si vous êtes sûr que les systèmes visés implémentent correctement le mode d'ajout (c.-à-d. pas Win32) alors vous pouvez même omettre le seek() dans le code évoqué ci-dessus.

Si vous êtes sûr d'écrire du code pour un système d'exploitation et un système de fichiers qui implémentent correctement l'ajout (par exemple, un système de fichier local sur un Unix moderne) et si vous laissez le fichier en mode tampon par bloc et si vous la taille de ce que vous écrivez n'excède pas la taille du tampon entre chaque écriture réelle manuelle, alors vous avez la garantie que l'écriture du tampon à la fin du fichier se fera de manière atomique sans risque de mélange avec d'autres écritures. Vous pouvez aussi utiliser la fonction syswrite() qui n'est qu'un habillage de la fonction système write(2).

Il existe encore un risque infime qu'un signal interrompe l'appel système write() avant sa terminaison. Certains implémentations de STDIO peuvent aussi, rarement, effectuer plusieurs appels à write() alors que le tampon était bien vide au prélable. Il y a certains systèmes où la probabilité que cela arrive est proche de zéro.

2.23 Comment modifier un fichier binaire directement ?

Si vous souhaitez juste modifier un binaire, un code simple, comme ci-dessous, fonctionne bien.

```
perl -i -pe 's{window manager}{window mangler}g' /usr/bin/emacs
```

En revanche, si vous avez des enregistrements de taille fixe, alors vous pourriez faire plutôt comme ceci :

```
$RECSIZE = 220; # taille en octets de l'enregistrement
$recno   = 37;  # numéro d'enregistrement à modifier
open(FH, "+<somewhere") || die "can't update somewhere: $!";
seek(FH, $recno * $RECSIZE, 0);
read(FH, $record, $RECSIZE) == $RECSIZE || die "can't read record $recno: $!";
# modifie l'enregistrement
seek(FH, -$RECSIZE, 1);
print FH $record;
close FH;
```

Le verrouillage et le traitement des erreurs sont laissés en exercice au lecteur. Ne les oubliez pas, ou vous vous en mordrez les doigts.

2.24 Comment recupérer la date d'un fichier en perl ?

Si vous voulez recupérer la dernière date à laquelle le fichier a été lu, écrit ou a vu ses méta-informations (propriétaire, etc.) changées, utilisez les opérations de test sur fichier **-M**, **-A** ou **-C**, documentées dans *perlfunc*. Elles récupèrent l'âge du fichier (mesuré par rapport à l'heure de démarrage du programme) exprimée en fraction de jours. Selon la plateforme, certaines de ces dates ne sont pas disponibles. Pour récupérer l'âge "brut" en secondes depuis l'origine des temps (du système), il faut utiliser la fonction stat(), puis utiliser localtime(), gmtime() ou POSIX::strftime() pour convertir ce nombre dans un format lisible par un humain.

Voici un exemple :

```
$write_secs = (stat($file))[9];
printf "file %s updated at %s\n", $file,
    scalar localtime($write_secs);
```

Si vous préférez quelque chose de plus lisible, utilisez le module File::stat (qui fait partie de la distribution standard depuis la version 5.004) :

```
# gestion des erreurs laissée en exercice au lecteur.
use File::stat;
use Time::localtime;
$date_string = ctime(stat($file)->mtime);
print "file $file updated at $date_string\n";
```

L'approche POSIX::strftime() a le bénéfice d'être, en théorie, indépendante de la localisation courante. Voir *perllocale* pour plus de détails.

2.25 Comment modifier la date d'un fichier en perl ?

Utilisez le fonction utime() documentée dans utime in *perlfunc*. À titre d'exemple, voici un petit programme qui applique récupère les dates de dernier accès et de dernière modification de son premier argument et les applique à tous les autres :

```
if (@ARGV < 2) {
    die "usage: cptimes timestamp_file other_files ...\n";
}
$timestamp = shift;
($atime, $mtime) = (stat($timestamp))[8,9];
utime $atime, $mtime, @ARGV;
```

Comme d'habitude, le traitement des erreurs est laissé en exercice au lecteur.

La documentation de la fonction utime() donne aussi un exemple qui a le même effet que la fonction touch(1) surt les fichiers *existants*.

Certains systèmes de fichiers limitent la précision de stockage de ces dates. Par exemple, les systèmes de fichiers FAT et HPFS ne peuvent pas descendre en dessous de deux secondes de précision. Ce sont des limitations de ces systèmes de fichiers et non de la fonction utime().

2.26 Comment écrire dans plusieurs fichiers simultanément ?

Pour connecter un descripteur de fichier à plusieurs descripteurs en sortie, vous pouvez utiliser les modules IO::Tee orou Tie::FileHandle::Multiplex.

Pour une utilisation unique, on peut recourir à :

```
for $fh (FH1, FH2, FH3) { print $fh "whatever\n" }
```

2.27 Comment lire le contenu d'un fichier d'un seul coup ?

Vous pouvez utiliser le module File::Slurp pour faire cela :

```
use File::Slurp;

$all_of_it = read_file($filename);
     # tout le fichier dans un scalaire
@all_lines = read_file($filename);
     # une ligne du fichier par élément
```

L'approche habituelle en Perl pour traiter toutes les lignes d'un fichier est de le faire ligne par ligne :

```
open (INPUT, $file)         || die "can't open $file: $!";
while (<INPUT>) {
    chomp;
    # traiter la ligne qui est dans $_
}
close(INPUT)                || die "can't close $file: $!";
```

Ceci est nettement plus efficace que de lire le fichier tout entier en mémoire en tant que tableau de lignes puis de le traiter élément par élément, ce qui est souvent (sinon presque toujours) la mauvaise approche. Chaque fois que vous voyez quelqu'un faire ceci :

```
@lines = <INPUT>;
```

vous devriez réfléchir longuement et profondément à la raison jusitifant que tout soit chargé en même temps. Ce n'est tout simplement pas une solution extensible. Vous pourriez aussi trouver plus amusant d'utiliser le module Tie::File ou les liens $DB_RECNO du module standard DB_File, qui vous permettent de lier un tableau à un fichier tel que l'accès à un élément du tableau accède en vérité à la ligne correspondante dans le fichier.

Vous pouvez lire la totalité du fichier dans un scalaire :

```
{
    local(*INPUT, $/);
    open (INPUT, $file)     || die "can't open $file: $!";
    $var = <INPUT>;
}
```

Ce code donne temporairement la valeur indéfinie à votre séparateur d'enregistrements et ferme automatiquement le fichier à la sortie du bloc. Si le fichier est déjà ouvert, utilisez juste ceci :

```
$var = do { local $/; <INPUT> };
```

Pour des fichiers ordinaires, vous pouvez aussi utiliser la fonction read :

```
    read( INPUT, $var, -s INPUT );
```

Le troisième argument détermine le nombre d'octets contenus dans le filehandle INPUT pour tous les lire dans la variable $var.

2.28 Comment lire un fichier paragraphe par paragraphe ?

Utilisez la variable `$/` (voir *perlvar* pour plus de détails). Vous pouvez la positionner soit à `""` pour éliminer les paragraphes vides (`"abc\n\n\n\ndef"`, par exemple, sera traité comme deux paragraphes et non trois), soit à `"\n\n"` pour accepter les paragraphes vides.

Notez qu'une ligne blanche ne doit pas contenir de blancs. Ainsi, `"fred\n \nstuff\n\n"` est seul un paragraphe alors que `"fred\n\nstuff\n\n"` en fait deux.

2.29 Comment lire un seul caractère d'un fichier ? Et du clavier ?

Vous pouvez utiliser la fonction native `getc()` sur la plupart des descripteurs de fichier, mais elle ne marchera pas (facilement) sur un terminal. Pour STDIN, utilisez soit le module Term::ReadKey disponible sur CPAN, soit l'exemple fourni dans getc in *perlfunc*.

Si votre système supporte POSIX (portable operating system programming interface), vous pouvez utiliser le code suivant qui, vous l'aurez noté, supprime aussi l'echo pendant le traitement.

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
    my $got;
    print "gimme: ";
    $got = getone();
    print "--> $got\n";
}
exit;

BEGIN {
    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);
        $term->setcc(VTIME, 1);
        $term->setattr($fd_stdin, TCSANOW);
    }

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

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

}

END { cooked() }
```

Le module Term::ReadKey de CPAN est sans doute plus facile à utiliser. Les versions récentes incluent aussi un support pour les systèmes non portables.

```
use Term::ReadKey;
open(TTY, "</dev/tty");
print "Gimme a char: ";
ReadMode "raw";
$key = ReadKey 0, *TTY;
ReadMode "normal";
printf "\nYou said %s, char number %03d\n",
    $key, ord $key;
```

2.30 Comment savoir si un caractère est disponible sur un descripteur de fichier ?

La première chose à faire, c'est de se procurer le module Term::ReadKey depuis CPAN. Comme nous le mentionnions plus haut, il contient même désormais un support limité pour les systèmes non portables (comprendre : non ouverts, fermés, propriétaires, non POSIX, non Unix, etc.).

Vous devriez aussi lire la Foire Aux Questions de comp.unix.* pour ce genre de chose : la réponse est sensiblement identique. C'est très dépendant du système d'exploitation utilisé. Voici une solution qui marche sur les systèmes BSD :

```
sub key_ready {
    my($rin, $nfd);
    vec($rin, fileno(STDIN), 1) = 1;
    return $nfd = select($rin,undef,undef,0);
}
```

Si vous désirez savoir combien de caractères sont en attente, regardez du côté de ioctl() et de FIONREAD. L'outil *h2ph* qui est fourni avec Perl essaie de convertir les fichiers d'inclusion du C en code Perl, qui peut alors être utilisé via `require`. FIONREAD se retrouve défini comme une fonction dans le fichier *sys/ioctl.ph* :

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

$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)    or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

Si *h2ph* n'est pas installé ou s'il ne fonctionne pas pour vous, il est possible d'utiliser directement *grep* sur les fichiers d'en-tête :

```
% grep FIONREAD /usr/include/*/*
/usr/include/asm/ioctls.h:#define FIONREAD      0x541B
```

Ou écrivez un petit programme C, en utilisant l'éditeur des champions :

```
% cat > fionread.c
#include <sys/ioctl.h>
main() {
    printf("%#08x\n", FIONREAD);
}
^D
% cc -o fionread fionread.c
% ./fionread
0x4004667f
```

Puis, utilisez directement cette valeur, en laissant le problème de portage comme exercice à votre successeur.

```
$FIONREAD = 0x4004667f;         # XXX: depend du système d'exploitation

$size = pack("L", 0);
ioctl(FH, $FIONREAD, $size)     or die "Couldn't call ioctl: $!\n";
$size = unpack("L", $size);
```

FIONREAD impose un descripteur connecté à un canal (stream), ce qui signifie qu'il fonctionne bien avec les prises (sockets), les tubes (pipes) et les terminaux (tty) mais *pas* avec les fichiers.

2.31 Comment écrire un `tail -f` en perl ?

Essayez d'abord :

```
seek(GWFILE, 0, 1);
```

La ligne `seek(GWFILE, 0, 1)` ne change pas la position courante, mais elle efface toute indication de fin de fichier sur le descripteur, de sorte que le prochain <GWFILE> conduira Perl à essayer de nouveau de lire quelque chose.

Si cela ne fonctionne pas (cela demande certaines propriétés à votre implémentation stdio), alors vous pouvez essayer quelque chose comme :

```
for (;;) {
  for ($curpos = tell(GWFILE); <GWFILE>; $curpos = tell(GWFILE)) {
        # cherche des trucs, et mets-les quelque part
  }
  # attendre un peu
  seek(GWFILE, $curpos, 0);  # retourne où nous en étions
}
```

Si cela ne marche pas non plus, regardez du côté du module POSIX. POSIX définit la fonction clearerr() qui peut ôter la condition de fin de fichier sur le descripteur. La méthode : lire jusqu'à obtenir une fin de fichier, clearerr(), lire la suite. Nettoyer, rincer, et ainsi de suite.

Il existe aussi un module File::Tail sur le CPAN.

2.32 Comment faire un dup() sur un descripteur en Perl ?

Voir open in *perlfunc* pour obtenir divers moyens d'appeler open() qui pourraient vous convenir. Par exemple :

```
open(LOG, ">>/tmp/logfile");
open(STDERR, ">&LOG");
```

Ou même avec des descripteurs numériques :

```
$fd = $ENV{MHCONTEXTFD};
open(MHCONTEXT, "<&=$fd");   # comme fdopen(3S)
```

Noter que "<&STDIN" donne un clone, mais que "<&=STDIN" donne un alias. Cela veut dire que si vous fermez un descripteur possédant des alias, ceux-ci deviennent indisponibles. C'est faux pour un clone.

Comme d'habitude, le traitement d'erreur est laissé en exercice au lecteur.

2.33 Comment fermer un descripteur connu par son numéro ?

Cela n'est que très rarement nécessaire puisque la fonction close() de Perl n'est censée être utilisée que pour des choses ouvertes par Perl, même si c'est un clone (dup) de descripteur numérique comme MHCONTEXT ci-dessus. Mais si c'est absolument nécessaire, vous pouvez essayer :

```
require 'sys/syscall.ph';
$rc = syscall(&SYS_close, $fd + 0);  # doit forcer une valeur numérique
die "can't sysclose $fd: $!" unless $rc == -1;
```

Ou bien utilisez juste la caractéristique fdopen(3S) de open() :

```
{
    local *F;
    open F, "<&=$fd" or die "Cannot reopen fd=$fd: $!";
    close F;
}
```

2.34 Pourquoi "C:\temp\foo" n'indique pas un fichier DOS ? Et même "C:\temp\foo.exe" ne marche pas ?

Ouille ! Vous venez de mettre une tabulation et un formfeed dans le nom du fichier. Rappelez-vous qu'à l'intérieur des guillemets ("\tel quel"), le backslash est un caractère d'échappement. La liste complète de telles séquences est dans Opérateurs apostrophe et type apostrophe in *perlop*. Evidemment, vous n'avez pas de fichier appelé "c:(tab)emp(formfeed)oo" ou "c:(tab)emp(formfeed)oo.exe" sur votre système de fichiers DOS originel.

Utilisez soit des guillemets simples (apostrophes) pour délimiter vos chaînes, ou (mieux) utilisez des slashes. Toutes les versions de DOS et de Windows venant après MS-DOS 2.0 traitent / et \ de la même façon dans les noms de fichier, donc autant utiliser une forme compatible avec Perl – ainsi qu'avec le shell POSIX, ANSI C et C++, awk, Tcl, Java, ou Python, pour n'en mentionner que quelques autres. Les chemins POSIX sont aussi plus portables.

2.35 Pourquoi `glob("*.*")` ne donne-t-il pas tous les fichiers ?

Parce que, même sur les portages non-Unix, la fonction glob() de Perl suit la sémantique normale de complétion d'Unix. Il faut utiliser `glob("*")` pour obtenir tous les fichiers (non cachés). Cela contribue à rendre glob() portable y compris sur les systèmes ancestraux. Votre portage peut aussi inclure des fonctions de globalisation propriétaires. Regardez sa documentation pour plus de détails.

2.36 Pourquoi Perl me laisse effacer des fichiers protégés en écriture ? Pourquoi `-i` écrit-il dans des fichiers protégés ? N'est-ce pas un bug de Perl ?

Ce sujet est traité de façon complète et fastidieuse dans un article de la collection "Far More Than You Ever Wanted To Know" disponible sur http://www.perl.com/CPAN/doc/FMTEYEWTK/file-dir-perms .

En résumé, apprenez comment fonctionne votre système de fichiers. Les permissions sur un fichier indiquent seulement ce qui peut arriver aux données dudit fichier. Les permissions sur le répertoire indiquent ce qui peut survenir à la liste des fichiers contenus dans ce répertoire. Effacer un fichier revient à ôter son nom de la liste du répertoire (donc l'opération est régie par les permissions sur le répertoire, pas sur le fichier). Si vous essayez d'écrire dans le fichier alors les permissions du fichiers sont prises en compte pour déterminer si vous en avez le droit.

2.37 Comment sélectionner une ligne au hasard dans un fichier ?

Voici un algorithme tiré du Camel Book :

```
srand;
rand($.) < 1 && ($line = $_) while <>;
```

Il a un énorme avantage en espace par rapport à la solution consistant à tout lire en mémoire. Une preuve que cette méthode est correcte se trouve dans *The Art of Computer Programming*, Volume 2, Section 3.4.2, par Donald E. Knuth.

Vous pouvez utiliser le module File::Random qui fournit une fonction reposant sur cette méthode.

```
use File::Random qw/random_line/;
my $line = random_line($filename);
```

Un autre moyen passe par le module Tie::File qui donne accès à tout un fichier comme si c'était un tableau. Il suffit alors de choisir au hasard un élément du tableau.

2.38 Pourquoi obtient-on des espaces étranges lorsqu'on affiche un tableau de lignes ?

Le fait de dire :

```
print "@lines\n";
```

joint les éléments de `@lines` avec un espace entre eux. Si `@lines` contient `("little", "fluffy", "clouds")` alors l'instruction précédente affiche :

```
little fluffy clouds
```

mais si chaque élément de `@lines` est une ligne de texte, terminée par une fin de ligne, `("little\n", "fluffy\n", "clouds\n")`, alors elle affiche :

```
little
 fluffy
 clouds
```

Si votre tableau contient déjà des lignes, affichez les simplement :

```
print @lines;
```

3 AUTHOR AND COPYRIGHT (on Original English Version)

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

Raphaël Manfredi <Raphael_Manfredi@grenoble.hp.com>.

Roland Trique <*roland.trique@uhb.fr*>, Paul Gaborit *paul.gaborit @ enstimac.fr* (mise à jour).

4.3 Relecture

Régis Julie <*Régis.Julie@Cetelem.fr*>, 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.