Programmation des Systèmes

Système de fichiers et entrées/sorties

Philippe MARQUET
et les équipes pédagogiques successives*

Ce document est le support de travaux dirigés et de travaux pratiques relatifs au notions de système de fichiers et d'entrée/sortie. On étudie la manipulation de ces notions par les primitives POSIX et la bibliothèque C standard.

1 Quelques commandes Unix

Il est proposé de réimplanter quelques commandes Unix courantes. Ces quelques exercices illustrent l'utilisation des primitives POSIX relatives au système de fichiers : autorisations d'accès, parcours d'un répertoire, parcours d'un système de fichiers, entrées/sorties.

Vérifier les droits d'accès... et expliquer

La commande Unix `test` permet d'effectuer de très nombreux tests. Nous allons nous intéresser ici à la vérification des droits d'accès (lecture : option `-r`, écriture : option `-w`, exécution : option `-x`) à un fichier donné.

La commande `test` n'affiche rien mais retourne son résultat sous la forme d'une terminaison sur un succès ou un échec. Ce comportement peut par exemple être mis en valeur dans une session telle la suivante (Il s'agit d'une session C-shell. Pour un shell de la famille de `sh` tel `bash`, on utilisera `$?` en remplacement de `$status`) :

```
% test -r /tmp
% echo $status
0
% test -r /tmp && echo OK
OK
% test -w /etc/passwd
% echo $status
1
% test -w /etc/passwd && echo OK
%
```

Une autre approche possible est de configurer votre shell afin qu'il affiche systématiquement le retour de la dernière commande exécutée. Si vous utilisez `bash`, vous trouverez dans le dépôt `git` du cours un modèle de `.bashrc`, contenant quelques options de configuration que vous pouvez utiliser : voir l'encart page suivante.

Dans le cadre de ces exercices, il s'agit de fournir une commande `maccess` [1] qui supporte les options `-r`, `-w`, `-x` (et non toutes les nombreuses options de `test`), ainsi que l'option supplémen-

1. Cette commande sera basée simplement sur la fonction `access()`, d'où son nom...

Contraintes POSIX

La norme POSIX fournit un certain nombres de contraintes de taille ou longueur diverses. Ces contraintes définissent des valeurs minimales que doit garantir toute implémentation de la norme. La norme POSIX impose aussi que les valeurs effectivement garanties par l'implémentation soient accessibles aux applications.
Parmi les macro-définitions fournies par les fichiers `<limits.h>` et `<unistd.h>` on trouve

`NAME_MAX` qui est la longueur maximale d'un nom d'entrée dans le système de fichiers (dont la valeur minimale requise est 14) ;

`PATH_MAX` qui est la longueur maximale d'un chemin dans le système de fichiers (dont la valeur minimale requise est 255).

Code source POSIX & `Makefile`

Afin de spécifier au compilateur que le code source C que l'on désire compiler est conforme à la norme POSIX, on définira la macro `_XOPEN_SOURCE` avec une valeur supérieure ou égale à 500 (les curieux pourront consulter le contenu du fichier `/usr/include/features.h`).
Un `Makefile` comportera donc par exemple :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS += -D_XOPEN_SOURCE=500
CFLAGS += -g
```

taire `-v` (*verbose*) expliquant, en cas d'échec, ce pourquoi l'accès est impossible. Vous consulterez la page de manuel de `access()` pour savoir quels sont les motifs d'échec possible.

Exercice 1 (`prlimit`, information de configuration)
Fournissez une commande `prlimit` qui affiche les valeurs des constantes `NAME_MAX` et `PATH_MAX` sur votre système, voir l'encart. □

Exercice 2 (`maccess`)
Fournissez une commande `maccess` qui permette de tester les droits d'accès à un fichier (par les options `-r`, `-w` et `-x`) et qui propose une option `-v` fournissant un des motifs d'erreur possibles.

Les permissions d'accès à un fichier peuvent être vérifiées par l'utilisation de l'appel système `access()`. La page de manuel :

```
% man 2 access
```

pourra être consultée.

Pour assurer un traitement simple, vous pourrez imposer l'ordre des arguments de la ligne de commande. □

Dépôt `git` du cours

Afin de vous mettre le pied à l'étrier, vous trouverez dans le dépôt `git` du cours, en plus du code des exemples vus en cours, un modèle de `Makefile` et des éléments de configuration `bash`.
Vous pouvez cloner ce dépôt en tapant :

```
git clone http://www.fil.univ-lille1.fr/~hym/d/pds.git
```

À l'avenir, la commande `git pull` (à lancer depuis le répertoire cloné) vous permettra d'obtenir les mises à jour.

Copie d'une session

Il existe plusieurs façons de garder une trace d'une session.

- La commande `script` permet de fournir dans un fichier une copie d'une session shell. Elle démarre un nouveau shell et enregistre toutes les entrées-sorties jusqu'à la fin de l'exécution du shell (Contrôle-D ou `exit` pour terminer `bash`). Consultez la page de manuel de `script` pour en savoir plus.
- Il est possible de lancer un shell directement dans l'éditeur `emacs` (dans les modes `shell` et `eshell`). Toute la session peut alors être éditée (presque) comme d'habitude.
- `screen` et les outils similaires (notamment `tmux`) permettent de multiplexer plusieurs shell dans un seul terminal, mais également de conserver l'historique, etc. Consultez leur page de manuel pour en savoir plus.

Exercice 3 (Trouver toutes les erreurs)
Fournissez un exemple de session shell comportant une suite d'invocations de la commande `maccess` produisant toutes les erreurs possibles, c'est-à-dire toutes les causes d'échec de la fonction `access()`. Vous expliquerez pourquoi, s'il y a des erreurs que vous ne pouvez pas produire.

Cette session inclura les commandes de création des fichiers sur lesquels la commande `maccess` est invoquée.

La meilleure façon de mettre en évidence ces erreurs est de proposer un script shell qui crée chacun des cas, déclenche `maccess`, et vérifie le code de retour. Ce script sera déclenché pour la cible `test` de votre `Makefile`.

Voir l'encart sur la manière de réaliser une copie d'une session qui vous permettra de rendre une trace du déclenchement de `make test`. □

Exercice 4 (`maccess+`)
Proposez une seconde version de `maccess` reposant sur la fonction `getopt()` pour analyser plus facilement les arguments. On pourra alors tester à la fois les droits de lecture et d'écriture, par exemple.

La page de manuel :

```
% man 3 getopt
```

fournit toutes les explications nécessaires, notamment un exemple complet d'utilisation.

Retrouver une commande

La commande Unix `which` recherche les commandes dans les chemins de recherche de l'utilisateur.

La variable d'environnement `$PATH` contient une liste de répertoires séparés par des « : ». Lors de l'invocation d'une commande depuis le shell, un fichier exécutable du nom de la commande est recherchée dans l'ordre dans ces répertoires. C'est ce fichier qui est invoqué pour l'exécution de la commande. Si aucun fichier ne peut être trouvé, le shell le signale par un message.

La commande `which` affiche le chemin du fichier qui serait trouvé par le shell lors de l'invocation de la commande :

```
% echo $PATH
/bin:/usr/local/bin:/usr/X11R6/bin:/users/phm/bin
% which ls
/bin/ls
% echo $status
0
% which foo
foo: Command not found.
% echo $status
```

```
1
```

Exercice 5 (Liste des répertoires de recherche)
Dans un premier temps, proposez une fonction `filldirs()` qui remplit un tableau `dirs` contenant la liste des noms des répertoires de `$PATH`. Ce tableau sera terminé par un pointeur `NULL`. □

Exercice 6 (Une fonction `which()`)
Écrivez maintenant une fonction `which()` qui affiche le chemin absolu correspondant à la commande dont le nom est passé en paramètre ou un message d'erreur si la commande ne peut être trouvée dans les répertoires de recherche de `$PATH`. Cette fonction retourne une valeur booléenne indiquant un succès ou un échec de la recherche.

On pourra utiliser l'appel système

```
#include <unistd.h>
int access(const char *path, int mode);
```

qui vérifie les permissions fournies sous la forme d'un « ou » des valeurs `R_OK` (*read*, lecture), `W_OK` (*write*, écriture), `X_OK` (*execution*, exécution), `F_OK` (existence). □

Exercice 7 (La commande `which`)
Terminez par écrire votre propre version de la commande `which` qui se termine par un succès si et seulement si toutes les commandes données en paramètre ont été trouvées dans les répertoires de recherche désignés par `$PATH`. □

Afficher le répertoire courant

La commande `pwd` (*print working directory*) affiche le chemin absolu du répertoire de travail courant.

La fonction

```
char *getcwd(char *buf, size_t size);
```

de la bibliothèque C copie le chemin absolu du répertoire courant dans le tableau `buf` de taille `size`. Si `buf` est `NULL`, la bibliothèque alloue dynamiquement un espace nécessaire pour y stocker le résultat et en retourne alors l'adresse.

Exercice 8 (Utilisation de `getcwd()`)
Il est immédiat de fournir une implantation de la commande `pwd` basée sur une utilisation de `getcwd()`. □

L'implantation de cette fonction `getcwd()` ne repose pas sur un appel système unique mais exploite la structuration des répertoires et les liens `.` et `..` pour construire ce chemin absolu.

Pour un processus donné, le système ne conserve que le numéro d'inœud courant, à partir duquel il lui est possible d'accéder au répertoire courant. La navigation à partir de ce répertoire est utilisé pour tous les chemins relatifs, en particulier ceux vers des répertoires parents, tels `..` ou `../..`, etc.

Le système conserve aussi l'inœud du répertoire racine `/` qui lui permet d'accéder à tous les chemins absolus.

Il s'agit maintenant de proposer une implantation de la commande `pwd` ne reposant pas sur la bibliothèque C. Le principe en est le suivant :

— on part du répertoire `.` ;
— on cherche dans le répertoire `..` le nom du lien qui référence le même inœud que `.` ; ce nom existe assurément ; soit `cname` ce nom ; le chemin absolu que l'on cherche à construire est de la forme `/.../cname` ;
— on cherche alors dans le répertoire `../..` le nom du lien qui référence le même inœud que `..` ; soit `pname` ce nom ; le chemin absolu que l'on cherche à construire est de la forme `/.../pname/cname` ;
— on itère cette remontée jusqu'à trouver la racine du système de fichiers ;

Informations sur une entrée

Deux valeurs peuvent être utilisées pour rechercher des informations sur une entrée dans un système de fichiers :

- des valeurs de type `struct dirent` retournée par l'appel système `readdir()` ;
- des valeurs de type `struct stat` retournée par l'appel système `stat()`, `lstat()` (dans le cas d'un lien symbolique, informations sur le fichier lui-même et non sur le fichier désigné par le lien symbolique), ou `fstat()` (informations sur un fichier désigné par un descripteur).

Reportez-vous au manuel pour les détails des champs que ces structures contiennent et des macros définies pour tester les valeurs de ces champs (en particulier les macros testant le type d'un fichier (ordinaire, répertoire, etc.)).

- la racine du système de fichier peut être identifiée :
  - par le fait qu'elle correspond à l'inœud du chemin `/`, ou
  - par le fait qu'elle est son propre parent.

Exercice 9 (Racine du système de fichiers)
Donnez le code d'une fonction qui détermine si un chemin donné en paramètre correspond à la racine du système de fichiers. □

Exercice 10 (Mon nom dans le répertoire parent)
Fournissez une fonction `print_name_in_parent()` qui affiche sur la sortie standard le nom du lien d'un nœud donné dans son répertoire père. □

Exercice 11 (Construction du chemin absolu)
L'affichage du chemin absolu d'un répertoire pouvant être produit par l'affichage du chemin absolu du répertoire père suivit du nom du répertoire courant dans le répertoire père, proposez une définition de la fonction récursive `print_node_dirname()` qui affiche le chemin absolu du répertoire passé en paramètre. □

Exercice 12 (Fonction `pwd()`)
Terminez par une fonction `pwd()` qui affiche le chemin absolu du répertoire courant. □

Parcours d'une hiérarchie

La commande `du` (*disk usage*) rapporte la taille disque utilisée par un répertoire et l'ensemble de ses fichiers (y compris ses sous-répertoires).

Deux tailles peuvent être prises en compte pour un fichier :

- la taille apparente, qui est le nombre d'octets contenus dans le fichier ;
- la taille réelle, qui correspond effectivement au nombre de blocs disque utilisés par le fichier.

Les champs `st_size` et `st_blocks` d'une structure de type `struct stat` retournée par l'appel système `stat()` contiennent respectivement la taille apparente et la taille réelle d'un fichier ; vous trouverez plus de détails dans le manuel de `stat`.

Comme pour toutes les commandes réalisant un parcours d'une hiérarchie, il faut préciser le traitement réalisé sur les liens rencontrés : doivent-ils être suivis ou non ?

La commande `du` comporte donc deux options

`-L` indiquant de suivre les liens symboliques ; ce qui n'est pas le cas par défaut ;

`-b` indiquant de rapporter les tailles apparentes ; ce sont les tailles réelles qui sont rapportées sinon.

Une possible implantation peut définir deux variables globales pour mémoriser l'état de ces options :

Validation de votre `du`

Votre `du` devra être validé rigoureusement par une série de tests. Vous devrez ainsi créer des fichiers et des répertoires, de tailles différentes, avec ou sans liens symboliques, etc. et vérifier que votre `du` donne les mêmes résultats que la commande standard.
Pour vous comparer au `du` standard dans les salles du FIL, vous utiliserez, suivant les cas de tests, les options :

- `-b` pour calculer la taille apparente,
- `-B 512` pour calculer l'occupation réelle,
- `-L` pour suivre les liens symboliques.

Ainsi, en notant `mdu` votre `du` :

- `./mdu rep` devra donner le même résultat final que `du -B 512 rep`,
- `./mdu -b rep` devra donner le même résultat final que `du -b rep`,
- l'option `-L` devra avoir le même sens pour les deux commandes.

Référez-vous au manuel pour en savoir plus sur `du`.

```
static int opt_follow_links = 0;
static int opt_apparent_size = 0;
```

Dans un premier temps on suppose que l'option `opt_follow_links` est définie à faux.

Exercice 13 (Taille d'un fichier)
Proposez une implantation d'une fonction récursive

```
int du_file(const char *pathname);
```

qui retourne la taille occupée par le fichier désigné et ses éventuels sous-répertoires.

Comme dans un premier temps on ne suit pas les liens symboliques, la taille d'un lien symbolique est la taille occupée par le lien, et non par le fichier visé.

Votre implantation devra faire en particulier attention à filtrer les entrées des répertoires : il faut ignorer `.` et `..` afin d'éviter une boucle infinie. □

Exercice 14 (Suivre les liens symboliques)
Modifiez l'implantation précédente pour suivre les liens symboliques. □

Exercice 15 (Comptage multiple ?)
Un nœud qui serait référencé plusieurs fois dans une hiérarchie dont on veut afficher la taille serait comptabilisé plusieurs fois. En quoi est-ce gênant ? Comment s'en affranchir ? □

Afficher la fin d'un fichier

La commande Unix `tail` affiche les dernières lignes des fichiers désignés par les paramètres de la ligne de commande. L'option `-n` *N*, ou directement `-`*N*, (où *N* est un entier) indique d'afficher les *N* dernières lignes. Par défaut, les 10 dernières lignes sont produites.

Exercice 16 (Version simpliste de `tail`)
Une version simpliste de la commande détermine le nombre de lignes du fichier puis parcourt le fichier pour débuter l'affichage *n* lignes avant la fin.

Malgré l'inconvénient majeur de cette approche, proposez une telle implantation. □

Une solution plus efficace consiste à lire le fichier depuis la fin. On lit un tampon de caractères d'une taille arbitraire (mais judicieusement choisie !). On y compte le nombre de retours à la ligne. S'il est supérieur à *n*, on peut afficher les *n* dernières lignes. Sinon, on lit le tampon précédent...

Exercice 17 (Fin d'un tampon)
Écrivez une fonction

Validation de votre `tail`

Votre `tail` devra être validé rigoureusement par une série de tests et en comparant vos résultats avec ceux du `tail` standard. Vous penserez à tester en particulier :

- des fichiers qui contiennent moins que le nombre de lignes que vous demandez à `tail`,
- des fichiers dans lesquels le dernier octet est une fin de ligne et des fichiers dans lesquels il est différent.

Si votre `tail` ne supporte pas tous les types de fichiers, vous penserez à montrer malgré tout ce cas par un test.
Référez-vous au manuel pour en savoir plus sur `tail`.

```
int index_tail_buffer(const char *buffer, int bufsize,
                      int ntail, int *nlines);
```

qui retourne l'index du début des `ntail` dernières lignes. Cet index est relatif au tampon de taille `bufsize`. Si le tampon comporte moins de `ntail` lignes, une valeur négative est retournée et `nlines` est positionné avec le nombre de lignes du buffer. □

Exercice 18 (Fonction `tail()` relative)
Écrivez une fonction récursive

```
tail_before_pos(int fd, unsigned int pos, int ntail);
```

qui considère le contenu du fichier référencé par le descripteur `fd` jusqu'à sa position `pos`. Cette position `pos` est comprise comme un déplacement depuis la fin du fichier. La fonction récursive `tail_before_pos()` affiche les `ntail` dernières lignes du fichier. □

Exercice 19 (Version efficace de `tail`)
Écrivez enfin une fonction

```
tail(const char *path, int ntail);
```

qui affiche les `ntail` dernières lignes du fichier désigné. □

Exercice 20 (Version utile de `tail`)
Une première version utile de la commande lit le fichier depuis le début en conservant dans un tampon circulaire les n dernières lignes lues. Ce tampon est affiché quand la fin du fichier est atteinte.

Pourquoi une telle implantation peut-elle être qualifiée d'utile ? □

2 Bibliothèque d'entrées/sorties tamporisées

Les primitives d'entrées/sorties fournies par l'interface POSIX manipulant des descripteurs de fichiers sont complétées par les fonctions de la bibliothèque C standard manipulant des structures de type `FILE`.

L'objet des exercices proposés ici est d'implanter, au-dessus des appels systèmes POSIX, une bibliothèque utilisateur reprenant certaines des caractéristiques de la bibliothèque C.

Appels système POSIX

Les appels systèmes POSIX manipulent des *descripteurs de fichiers* qui sont référencés par des entiers `int`. Ces primitives utilisent des tampons qui ne sont pas accessibles à l'utilisateur et qui sont partagés par tous les processus. La taille de ces tampons est directement liée aux périphériques utilisés ; c'est par exemple la taille des secteurs d'un disque...

Chaque invocation d'une de ces primitives nécessite le transfert du flux d'exécution de l'espace utilisateur vers l'espace noyau ; transfert qui est coûteux.

Les entrées/sorties de ce niveau ont donc intérêt à être réalisées sur des données de taille multiple de la taille des tampons et sur une frontière multiple de cette même taille.

C'est ce type de contraintes qu'assurent les fonctions de la bibliothèque d'entrées/sorties.

Bibliothèque d'entrées/sorties standard

Les fonctions de la bibliothèque manipulent des *flots* (*stream*) implémentés par des `FILE`.

Le tampon associé à un flot est dans l'espace utilisateur : chaque invocation d'une opération d'entrée/sortie ne nécessite plus le passage en mode noyau. Ce dernier n'ayant lieu que pour vider ou remplir un tampon utilisateur.

La taille de ces tampons est choisie pour correspondre aux caractéristiques du périphérique. Les entrées/sorties de bas niveau se feront donc bien par taille et sur des frontières multiples de la taille des tampons système.

Ces tampons système conservent cependant leur intérêt puisqu'ils permettent à un processus de ne pas attendre l'écriture réelle sur le disque et autorisent la lecture anticipée.

Outre le fait que ces entrées/sorties soient tamporisées, la bibliothèque C standard fournit aussi des fonctions de plus haut niveau qui impriment et analysent les données sous le contrôle d'une spécification de format (`printf()`, etc.).

Implantation d'une bibliothèque d'entrées/sorties tamporisées

On propose ici une implantation d'une version très basique d'une bibliothèque d'entrées/sorties tamporisées. Seuls les équivalents des primitives `fopen()`, `fclose()`, `fgetc()`, et `fputc()` seront implantés.

On se limite aussi dans un premier temps à un mode d'accès en lecture/écriture alors que la bibliothèque standard propose aussi des accès en lecture seule ou écriture seule.

La gestion d'un tampon utilisateur nécessite de mémoriser une position courante dans le tampon, l'index du tampon dans le fichier, le nombre de caractères valides dans le tampon, un indicateur signalant que le contenu du tampon a été modifié, et une référence au fichier sous la forme du descripteur associé aux appels système.

Exercice 21 (Type de données `pds_file_t`)
Proposez un type de données `pds_file_t`, pendant du `FILE` de la bibliothèque standard, pour mémoriser les informations associées à un de nos flots de données. □

La bibliothèque est organisée autour d'une structure de données regroupant toutes les informations nécessaires à la gestion de l'ensemble des fichiers ouverts par un processus. Le nombre de fichiers pouvant être ouverts par un processus étant borné, l'ensemble des structures associées aux flots ouverts peut être mémorisé dans un tableau.

Trois flots de données sont associés aux descripteurs `STDIN_FILENO`, `STDOUT_FILENO`, et `STDERR_FILENO` (0, 1 et 2) ; ils sont désignés par les symboles `pds_stdin`, `pds_stdout`, et `pds_stderr` qui eux-mêmes désignent les premiers éléments du tableau des références de flots.

Exercice 22 (Table des flots et entrées/sorties standard)
Donnez la définition et initialisation du tableau des descripteurs de flots ainsi que celles des symboles référençant les entrées/sorties standard. □

Lors de l'ouverture d'un flot par un `pds_fopen()`, un emplacement libre est recherché dans le tableau des flots à partir du début du tableau. Une référence vers cet emplacement sera retournée en cas de succès. L'initialisation des différents champs de la structure est réalisée. En particulier, un appel système permet de réaliser l'ouverture du fichier associé au flot. L'allocation dynamique du tampon est réalisée, son contenu est lu depuis le fichier.

Exercice 23 (Création et ouverture d'un flot)
Donnez la définition d'une fonction

Validation de la bibliothèque d'entrées/sorties tamporisées

L'implantation d'une bibliothèque comporte aussi une phase de test et validation à laquelle une durée conséquente du temps de développement doit être consacrée.
Dans un premier temps, des tests unitaires ad hoc pourront vérifier le comportement d'appels bien choisis de la bibliothèque pour en explorer toutes les fonctionnalités, y compris dans les cas limites.
Ces programmes de tests relèvent de la distribution d'une bibliothèque ; ils font aussi partie de ce qui doit être rendu en terme de travaux pratiques !

```
pds_file_t *pds_fopen(const char *path);
```

réalisant l'ouverture d'un flot de données et retournant une référence sur l'élément désignant ce flot dans le tableau des flots. La fonction `pds_fopen()` positionne le curseur de flot sur le début du flot. Si le fichier n'existe pas, `pds_fopen()` le crée. □

Exercice 24 (Vidage d'un tampon)
Implantez la fonction

```
int pds_fflush(pds_file_t *stream);
```

qui vide le tampon utilisateur si nécessaire par un appel système. □

Exercice 25 (Fermeture d'un flot)
Implantez la fonction

```
int pds_fclose(pds_file_t *stream);
```

qui termine l'utilisation d'un flot. □

Exercice 26 (Se déplacer)
Implantez la fonction

```
int pds_fseek(pds_file_t *stream, int position);
```

qui déplace la position courante à l'octet `position` du flot. □

Exercice 27 (Lire un caractère)
Réalisez la fonction

```
int pds_fgetc(pds_file_t *stream);
```

qui va chercher le prochain caractère du flot d'entrée référencé par `stream` et déplace l'indicateur de positionnement du flot.

Si le flot est en fin de fichier, retourne `PDS_EOF`. En cas d'erreur, l'indicateur d'erreur est positionné et c'est aussi cette valeur `PDS_EOF` qui est retournée. □

Exercice 28 (Écrire un caractère)
Donnez une implantation de la fonction

```
int pds_fputc(int c, pds_file_t *stream);
```

qui écrit le caractère `c` à la position courante du flot `stream` et incrémente cette position.

La fonction `pds_fputc()` retourne le caractère écrit. En cas d'erreur, la fonction retourne `PDS_EOF`. □

Durée d'exécution d'une commande

La commande `time` affiche le temps d'exécution d'une commande donnée en paramètre. Trois informations sont reportées (voir aussi l'option `-p`) :

- le temps écoulé entre le début et la fin de la commande (*elapsed time* ou *real time*) ;
- le temps processeur utilisé en mode utilisateur (*user time*) ;
- le temps processeur utilisé en mode système pour le compte du programme (*system time* ou *kernel time*).

Attention, il existe aussi une commande interne du shell nommée `time` qui reporte ces informations sous un autre format.
La commande `time` qui nous intéresse est souvent placée dans `/usr/bin/time`. Cette version comporte une option `-f` permettant de spécifier le format d'affiche du résultat à la manière de `printf`.

Performance d'une bibliothèque d'entrées/sorties tamporisées

Une bibliothèque d'entrées/sorties tamporisées telle que la bibliothèque standard ou celle proposée ici apporte des gains significatifs en terme de performance au prix d'une recopie supplémentaire.

Exercice 29 (Commandes `cat`)
Proposez trois versions de la commande `cat` qui recopie sur la sortie standard les contenus des fichiers donnés en paramètre :

- `catbib` qui utilise la bibliothèque ;
- `catsysc` qui utilise directement les appels système pour lire et écrire un caractère à la fois ;
- `catsysb` qui utilise des tampons de taille définie par la variable d'environnement `$CATSYSB_BUFSIZ` (par défaut 1024).

La comparaison des performances de `catbib` et `catsysc` permet d'évaluer les avantages de la tamporisation. La comparaison des performances de `catbib` et `catsysb` permet d'évaluer le surcoût de l'allocation du tampon utilisateur et des recopies dans ce tampon utilisateur.

Par ailleurs, des comparaisons sur des fichiers de différentes tailles et des tailles de tampon variable permettent de mesurer l'influence de ces paramètres sur les performances des entrées/sorties, par exemple mesurées en megaoctets par seconde. □

3 Performances des entrées/sorties

Les quelques exercices suivants visent à montrer que l'utilisation de tampons améliore très sensiblement les performances des opérations d'entrées/sorties. Deux niveaux d'utilisation de tampons peuvent être distingués :

- les tampons mis en œuvre dans l'espace utilisateur, par exemple par la bibliothèque standard d'entrées/sorties ;
- les tampons mis en œuvre dans l'espace noyau par le système d'exploitation lui-même et autorisant la non synchronisation entre les appels systèmes et les écritures effectives sur les périphériques.

Il va s'agir de produire différentes versions de la commande `cat` et d'en comparer les performances.

Il est rappelé que la commande `cat` produit sur la sortie standard (`STDOUT_FILENO` ou `stdout`) le contenu des fichiers dont les noms sont fournis en paramètre. Notre implantation n'accepte aucune option.

Exercice 30 (Influence de la taille des tampons)
Il s'agit d'écrire une version de la commande `cat` utilisant directement les appels systèmes `read()` et `write()`. Le tampon est alloué dynamiquement au début de l'exécution, sa taille est

Programmer en shell : les scripts shell

Un script shell est une suite de commandes shell regroupées dans un fichier dont l'exécution invoquera un shell avec la suite des commandes du fichier. En dehors des commandes habituellement utilisées en interactif, un shell reconnaît aussi des commandes telles des `if` ou `for` permettant le contrôle du flux d'exécution. Vous pourrez vous inspirer de l'exemple fourni ici. Sauvegardez le contenu suivant dans un fichier `mcat.sh`

```
#! /bin/sh -uf
#
# mcat -- campagne d'appels à mcat-scd
#
# squelette de shell script

# La commande à tester
MCAT=./mcat-scd
# le fichier à lui mettre en entrée
MCAT_INPUT=bigfile
# Le fichier de temps à générer
TIME_FILE=mcat-tm.dat

# la commande gnu time
TIME_CMD="/usr/bin/time"
# les options de cette commande
TIME_OPT="-f %e %U %S"

# initialisation du fichier de résultats
rm -f $TIME_FILE && echo "# real user sys" > $TIME_FILE

# un exemple de boucle
for str in  foo bar gee ; do
    echo $str
done

# un autre exemple de boucle
for size in `awk 'BEGIN { for( i=1; i<=8388608; i*=2 ) print i }'`; do
    export MCAT_BUFSIZ=$size
    echo $MCAT_BUFSIZ
done

# un exemple de redirection des sorties standard et d'erreur
$TIME_CMD "$TIME_OPT" ls > /dev/null 2>> $TIME_FILE

# eof
```

Il s'agit ensuite de rendre exécutable ce fichier :

```
% chmod +x mcat.sh
```

que l'on peut ensuite invoquer :

```
% ./mcat.sh
foo
bar
gee
1
2
4
8
[...]
% cat mcat-tm.dat
# real user sys
 0.01 0.00 0.01
```

Tracé de courbes avec `gnuplot`

`Gnuplot` est un utilitaire de tracé de courbes. Dans sa version interactive, `gnuplot` accepte des commandes au prompt. Ces commandes acceptent parfois des arguments. Les arguments chaîne de caractères doivent être fournis entre quotes simples `'` ou doubles `"`.
La commande `plot` permet de tracer des courbes. Les données sont lues depuis un fichier à raison d'un couple abscisse/ordonnée par ligne comme dans cet exemple basique, fichier `ex1.dat` (les `#` introduisent des commentaires) :

```
# taille duree
10   540.25
12   398.25
16   653.75
```

que l'on donne en paramètre à la commande `plot` :

```
gnuplot> plot "ex1.dat"
```

Il est possible de désigner les colonnes à considérer dans un fichier en comportant plus de deux comme celui-ci (`ex2.dat`) :

```
# taille duree vitesse
10   540.25  56
12   398.25  35
16   653.75  21
```

on utilise alors l'option `using` de la commande `plot` :

```
gnuplot> set xlabel "taille en lignes"
gnuplot> set ylabel "vitesse en octets/s"
gnuplot> plot "ex2.dat" using 1:3
```

L'exemple suivant, plus complet, introduit d'autres commandes et options. On utilise ici le fait qu'il est possible d'invoquer `gnuplot` avec en paramètre un fichier comportant des commandes

```
% cat run.gnu
set title "Temps et vitesse d'execution"
set logscale x
set xlabel "taille en lignes"
set logscale y
set ylabel "temps en s ou vitesse en octets/s"
set style data linespoints
plot "ex2.dat" using 1:2 title "temps", \
     "ex2.dat" using 1:3 title "vitesse"
pause -1  "Hit return to continue"
% gnuplot run.gnu
```

définie par la variable d'environnement `$MCAT_BUFSIZ`. Cette version sera nommée `mcat-scd` (mon `cat`, *sc* pour *system call* — appel système, *d* pour allocation dynamique du tampon). On invoquera successivement la commande avec en utilisant des tampons de 1 octet à 8 Mo en doublant la taille à chaque fois.

1. Donnez une implantation de `mcat-scd`.
2. Testez cette commande sur un fichier de taille conséquente en entrée, typiquement quelques dizaines de Mo. On se souciera aussi du type du système de fichiers sur lequel réside ce fichier (option `-T` de la commande `df`). La sortie pourra être redirigée vers `/dev/null`.
3. Menez une campagne d'évaluation des performances de la commande en croisant le temps d'exécution de la commande (voir l'encart page 10 à propos de la commande `time` de mesure de temps) et la taille du tampon. On automatisera cette série d'exécutions par un script shell ; se référer à l'encart page précédente pour un exemple.
4. Visualisez les résultats par une courbe en utilisant la commande `gnuplot`. Il s'agit encore de se référer à l'encart pour un exemple. □

Exercice 31 (Version appel système optimale)
Implantez maintenant une version `mcat-scs` (suffixe `s` pour *static*, statique) qui repose sur l'utilisation d'un tampon alloué statiquement de la taille optimale déterminée à l'étape précédente.

Exercice 32 (En utilisant la bibliothèque standard)
Implantez une version `mcat-lib` qui utilise les primitives `fgetc()` et `fputc()` de la bibliothèque standard d'entrées/sorties et comparez les temps d'exécution avec la version précédente.

Exercice 33 (Écritures en mode synchronisé)
Afin de mettre en évidence l'influence des tampons systèmes, comparez la version précédente `mcat-scs` avec une version `mcat-osync` qui positionne l'attribut `O_SYNC` sur le descripteur de la sortie standard.

Exercice 34 (Synchronisation finale)
Pour terminer, produisez `mcat-fsync`, version modifiée de `mcat_scs` qui vide les tampons disque suite aux écritures par un appel à `fsync()` et comparez les performances avec la version précédente.