Programmation des Systèmes

Système de fichiers et entrées/sorties

Samuel HYM, Philippe MARQUET, Florian VANHEMS, Jean-Stéphane VARRÉ*

Ce document est le support de travaux dirigés et de travaux pratiques relatifs aux 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.

1.1 Retrouver une commande

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

L'utilisateur peut configurer la liste des répertoires qui contiennent des commandes par la *variable d'environnement* (voir l'encart) `$PATH`. Cette variable d'environnement `$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 ces répertoires en les parcourant de gauche à droite. La recherche s'interrompt dès que le fichier est trouvé. 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 :

Variables d'environnement

Les variables d'environnement sont généralement manipulées par le shell (initialisées au démarrage d'une session). Si vous utilisez le shell `bash` (shell par défaut au M5), vous pouvez ainsi affecter une variable `$MYVAR` dans votre configuration `~/.bashrc` (le fichier `.bashrc` directement dans votre répertoire personnel) en y ajoutant une ligne :

```
export MYVAR="123 abc XYZ"
```

Pour accéder à la *valeur* d'une variable dans le shell, on utilise `$` suivi du nom de la variable :

```
$ echo $MYVAR
123 abc XYZ
```

Nous noterons du coup `$MYVAR` dans ce sujet pour désigner la variable d'environnement et éviter la confusion avec une constante `MYVAR` dans le code.
Consultez la page de manuel `environ`(7), dans la section 7 donc, pour en savoir plus.

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 macros fournies par `<limits.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).

```
$ echo $PATH
/bin:/usr/local/bin:/usr/X11R6/bin:/home/alice/bin
$ which ls
/bin/ls
$ echo $?
0
$ which foo
foo: Command not found.
$ echo $?
1
```

Exercice 1 (Trouver une fonction dans le manuel)
Cherchez dans le manuel la fonction permettant d'obtenir la valeur d'une variable d'environnement, par exemple en utilisant la commande `apropos`.

Exercice 2 (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 3 (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 4 (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`. □

1.2 Afficher les métadonnées d'un i-nœud

La commande `stat` permet d'afficher les métadonnées d'un i-nœud (fichier ordinaire, répertoire, etc.). Le but de cet exercice est de réimplémenter cette commande en utilisant les appels

systèmes. Votre objectif devra être d'avoir l'affichage le plus proche possible de celui de la commande `stat` standard.

Toutes les informations nécessaires sont données soit dans l'énoncé soit dans les pages du manuel. Vous aurez besoin de plus de pages que celles sélectionnées dans le mini-man : utilisez la commande `man` ! Si vous utilisez votre ordinateur personnel, veillez à installer les pages de manuel pour développeur (sections 2 et 3) en plus des pages documentant les commandes (section 1).

Exercice 5
Créez un fichier `test` contenant quelques lignes de texte. Exécutez « `stat test` ». Est-ce que vous comprenez la signification de tous les champs ? □

Il n'est pas rare de trouver des pages différentes du manuel ayant le même nom mais dans des sections différentes !

Exercice 6
Exécutez « `whatis stat` » pour lister toutes les pages portant le nom `stat`. Ouvrez ces différentes pages, par exemple en utilisant « `man` *section page* ». □

Beaucoup de pages de manuel contiennent une section « Voir aussi » (« See also ») pour indiquer d'autres pages sur le même thème. Ces liens pourront être utiles pour cet exercice.

Exercice 7
Quel appel système permet à la commande `stat` d'obtenir les métadonnées qu'elle affiche ?

Quel est le type, en C, utilisé pour les représenter ?

Que représentent les champs « *Blocs* » et « *Blocs d'E/S* » (« *IO Blocks* ») affichées par la commande `stat` selon le manuel ? Utilisez les différentes pages pour retrouver cette information. □

Exercice 8
Ajoutez du texte dans votre fichier jusqu'à trouver le seuil de taille pour lequel le nombre de blocs augmente. De combien augmente-t-il ? Pourquoi ? □

Intéressons-nous maintenant au champ « *Liens* ». Les liens permettent d'accéder à un même i-nœud par plusieurs chemins différents. Il existe deux types de liens : les liens *physiques* (hard link) et les liens *symboliques* (soft link or symbolic link).

La commande `ln` (pour *link*) permet de créer un lien.

Exercice 9
Créez un lien physique et un lien symbolique pointant sur votre fichier `test`. Créez aussi une copie de votre fichier `test`. □

Exercice 10
Exécutez `stat` sur le fichier, sur sa copie, et sur ses deux liens et comparez les résultats. Que constatez-vous pour le champ « `Liens` » ? Comparez aussi les numéros d'i-nœuds.

Supprimez le fichier `test`. Pouvez-vous retrouver le texte que vous aviez écrit à travers le lien symbolique ? le lien physique ? la copie ? □

Implémentation

Le code que vous rendrez, pour votre version de la commande `stat` comme pour tous les TPs, fournira un Makefile pour automatiser la compilation. Si besoin, révisez : `https://www.fil.univ-lille1.fr/~marquet/pdc/tp1/tp_introduction005.html`. Vous utiliserez les options de compilation `-Wall -Wextra` pour obtenir toute l'aide possible du compilateur.

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 -Wextra
CFLAGS += -D_XOPEN_SOURCE=500
CFLAGS += -g
```

Dépôt `git` du cours

Vous trouverez dans le dépôt `git` suivant les exemples de code vus en cours. Il contient notamment les `Makefile`s écrits pour compiler les exemples, qui peuvent vous servir de modèles pour vos `Makefile`s. Vous pouvez cloner ce dépôt en tapant :

```
git clone https://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.

Exercice 11
Implémentez une version basique de la commande `stat` ne prenant qu'un nom de fichier en argument. Votre code devra utiliser correctement l'appel système dédié, vérifier les erreurs possibles et afficher le plus simplement possible les champs de la structure associée. En particulier, vous pourrez afficher le champ *mode* comme un nombre, sans en extraire les informations de type et droits d'accès. □

Exercice 12
Reprenez votre code pour qu'il affiche séparément le type du fichier et ses droits d'accès. Vous utiliserez la même formulation que la commande `stat` standard pour le type d'i-nœud (« répertoire », etc.). Essayez de trouver un exemple pour pouvoir tester chaque type de fichier. Vous afficherez les droits d'accès en octal sur 4 chiffres. □

Exercice 13
Améliorez votre code pour qu'il affiche à présent le login et le nom de groupe du propriétaire. Vous utiliserez pour cela les fonctions `getpwuid` et `getgrgid`. Sur les machines du FIL, vous pouvez récupérer le nom des personnes en plus de leur login dans le champ `pw_gecos` de la structure. □

1.3 Afficher un fichier à l'envers

L'objectif de cette section est de définir une commande `retourne` permettant d'afficher le contenu d'un fichier à l'envers. Ainsi si un fichier `fic` contient exactement trois octets « `abc` », `retourne fic` affichera « `cba` ».

Exercice 14 (Version simpliste)
Proposez une implantation de `retourne` utilisant `lseek` (ou `pread`) pour aller lire le fichier source octet par octet en commençant par le dernier. □

Exercice 15 (Version gourmande)
Proposez une implantation de `retourne` qui charge le fichier source complètement en mémoire avant de l'afficher.
Listez les avantages et inconvénients de cette approche. □

Une autre approche est de n'avoir à chaque instant qu'une partie du fichier en mémoire, une partie qui tient dans un tampon mémoire de taille fixée `TAILLE_TAMPON`.

Exercice 16 (Version économe)
Proposez une implantation de `retourne` qui :
- charge les `TAILLE_TAMPON` derniers octets du fichier,
- les affiche en ordre inverse,
- charge les `TAILLE_TAMPON` octets précédents,
- ...

Pensez au cas où la taille du fichier n'est pas un multiple de `TAILLE_TAMPON` : que faut-il faire ? En quoi cet algorithme est-il plus économe ? □

1.4 Décodage du format `tar`

`.tar` est un format d'archive normalisé par POSIX. Il permet de regrouper dans un seul fichier, l'*archive*, plusieurs fichiers ainsi que des métadonnées (notamment droits d'accès, propriétaire, etc.). Ce format ne s'occupe pas de compresser l'archive obtenue : cette tâche est laissée à un autre outil, comme `gzip` ou `xz`. Les archives compressées utilisent alors respectivement les extensions `.tar.gz` (parfois abrégée en `.tgz`) ou `.tar.xz`.

Il existe plusieurs variantes du format `tar`, apparues au cours du temps et des différentes implémentations. Nous utiliserons la variante *UStar* (Unix Standard tar). Si vous utilisez l'implémentation GNU de `tar` (par exemple au M5), indiquez l'option `-H ustar` pour choisir ce format. Si vous utilisez une autre implémentation, consultez la documentation pour savoir comment choisir explicitement ce format. La suite de l'énoncé utilise les options de GNU tar.

Le format `.tar` est simple. Chaque fichier contenu dans une archive est stocké de la façon suivante :
- un bloc de 512 octets contient les métadonnées du fichier : 500 octets contenant la structure décrite ci-dessous et 12 octets de bourrage (*padding*, c'est-à-dire des octets nuls),
- le contenu brut du fichier,
- si besoin, du bourrage, pour que le fichier suivant commence à une position multiple de 512.

L'archive elle-même est une séquence de fichiers les uns à la suite des autres. Elle se termine par quelques blocs de 512 octets tous nuls. Ainsi, en parcourant un fichier archive 512 octets à la fois, on peut en décoder le contenu assez facilement.

La documentation de `tar` (pages Info de l'outil GNU tar et le fichier `tar.h` fourni par la libc, disponible dans le répertoire `/usr/include`) donne les informations suivantes sur le format du bloc de métadonnées :

```
struct posix_header
{                      /* Byte offset    Field type                  */
  char name[100];               /*   0    NUL-terminated if NUL fits */
  char mode[8];                 /* 100                                */
  char uid[8];                  /* 108                                */
  char gid[8];                  /* 116                                */
  char size[12];                /* 124                                */
  char mtime[12];               /* 136                                */
  char chksum[8];               /* 148                                */
  char typeflag;                /* 156    see below                   */
  char linkname[100];           /* 157    NUL-terminated if NUL fits */
  char magic[6];                /* 257    must be TMAGIC (NUL term.)  */
  char version[2];              /* 263    must be TVERSION            */
  char uname[32];               /* 265    NUL-terminated              */
  char gname[32];               /* 297    NUL-terminated              */
  char devmajor[8];             /* 329                                */
```

```
  char devminor[8];          /* 337                                 */
  char prefix[155];          /* 345    NUL-terminated if NUL fits */
                             /* 500                                 */
/* If the first character of prefix is '\0', the file name is name;
   otherwise, it is prefix/name.  Files whose pathnames don't fit in
   that length can not be stored in a tar archive.  */
};

#define TMAGIC   "ustar"        /* ustar and a null */
#define TMAGLEN  6
#define TVERSION "00"           /* 00 and no null */
#define TVERSLEN 2
```

Les champs `magic` et `version` permettent de s'assurer qu'il s'agit bien d'une archive `.tar` au format UStar : si les octets à ces positions-là n'ont pas exactement les valeurs attendues (`TMAGIC` et `TVERSION`), il ne s'agit pas d'une archive au format UStar.

Notez bien que tous les types des champs sont des tableaux de `char` mais il ne s'agit pas toujours de chaîne de caractères !

Le type indique explicitement s'il y a forcément un octet nul pour finir le champ. Même dans le cas où la spécification indique que le champ finit toujours par un octet nul, il est prudent de vérifier que l'octet nul est bien présent avant d'appeler une fonction qui suppose que l'octet est bien là (ce qui est le cas de la plupart des fonctions qui prennent une chaîne de caractères en argument). En effet, de nombreuses failles de sécurité dans les logiciels viennent de l'absence de telles vérifications.

Tous les champs pour lesquels il n'y a pas d'information de type (*field type*) sont des valeurs numériques représentées sous la forme d'une chaîne de caractères en *octal*. Consultez par exemple `strtol`(3) pour convertir une chaîne en entier dans votre code.

Consultez le fichier `tar.h` pour connaître les valeurs possibles du champ `typeflag`.

Exercice 17 (Créer une archive et décoder son contenu à la main)
Créez deux fichiers ordinaires, « `vide` » de 0 octet et « `test` » contenant 20 octets, et créez une archive `test.tar` en lançant : `tar -H ustar -c vide test > test.tar`.

Listez le contenu de l'archive en lançant : `tar -tv < test.tar`.

Utilisez l'outil « `od` » pour lire le contenu de l'archive et la décoder à la main, par exemple en utilisant « `od -Ad -t x1z -v test.tar` » (consultez la page de manuel de « `od` » pour en savoir plus sur ces options et en trouver d'autres qui vous conviennent mieux). En particulier, vous vérifierez que vous retrouvez le nom des fichiers, leur taille, votre `uid` (utilisez la commande `id` pour connaître votre identifiant numérique).

Par exemple, dans l'archive produite avec les fichiers `vide` et `test`, vous devriez trouver la taille de `test` aux positions 636 ($512 + 124$) à 647 (soulignées dans cet extrait de la sortie de `od`) :

```
0000624 37 35 30 00 30 30 30 31 37 35 30 00 30 30 30 30  >750.0001750.0000<
0000640 30 30 30 30 30 32 34 00 31 33 37 32 33 34 34 37  >0000024.13723447<
```

La colonne centrale contient les octets en hexadécimal, la colonne de droite les caractères correspondants quand ils sont affichables, ou un « `.` » quand ils ne le sont pas (notamment pour les octets nuls). Il s'agit bien de l'entier 20, écrit en octal en 11 caractères (donc `00000000024`) et terminé par un octet nul.

Comme l'affichage produit est assez long, vous aurez intérêt à utiliser un *tube* avec la commande `less` pour afficher la sortie de `od` page par page :

```
od -Ad -t x1z -v test.tar | less
```

Dans `less`, utilisez les touches « page précédente » et « page suivante » pour vous déplacer et « `q` » pour quitter. Consultez `less`(1) pour en savoir plus.

Vous pouvez même enchaîner toutes les commandes d'un coup :

```
tar -H ustar -c vide test | od -Ad -t x1z -v | less
```

et éviter ainsi de créer un fichier temporaire `test.tar` juste pour les tests.

Vous aurez sans doute envie de convertir des nombres d'octal ou hexadécimal en décimal et réciproquement :

— la commande « `printf '%d' 0123` » convertit 123 d'octal en décimal,
— la commande « `printf '%d' 0x123` » convertit 123 d'hexadécimal en décimal,
— la commande « `printf '%o' 123` » convertit 123 de décimal en octal, etc.

□

L'objectif des questions suivantes est d'implémenter une commande permettant de lister le contenu d'une archive en reproduisant la sortie de la commande `tar -tv`.

Comme les archives `.tar` sont structurées en bloc de 512 octets, nous allons avoir besoin d'arrondir les tailles de fichier au multiple de 512 supérieur.

Exercice 18 (Arrondi au multiple de 512)
Définissez une fonction `long arrondi512(long n)` qui arrondit l'entier `n` au multiple de 512 supérieur. Par exemple `arrondi(1024)` retournera `1024` mais `arrondi(1025)` retournera `1536`.

Puisque 512 est une puissance de 2 (512 s'écrit `0x200` en hexadécimal), vous pouvez utiliser un masque bit à bit pour calculer ce multiple. Vous pouvez aussi utiliser une division entière, selon ce qui vous semble le plus simple. □

Le tampon mémoire que l'appel système `read` permet de remplir peut avoir un type quelconque. En particulier, il peut s'agir de l'adresse d'une structure :

```
struct test_s s;
read(fd, &s, sizeof(struct test_s));
```

remplit la structure `s` à partir du fichier `fd`. Vérifiez cependant que `sizeof(struct test_s)` octets ont effectivement été lus avant d'interpréter le contenu de la structure.

Pour commencer la liste des fichiers dans l'archive, il faut arriver à décoder les tailles des fichiers.

Exercice 19 (Taille du premier fichier dans une archive)
Définissez une commande `lstar` qui lit le premier bloc de métadonnées UStar d'une archive sur son entrée standard et affiche la taille du fichier. Cette valeur devra être égale à celle affichée par `tar -tv` (troisième champ). Typiquement `tar -H ustar -c test | lstar` affichera 20 tandis que `tar -H ustar -c vide test | lstar` affichera 0. □

La structure de métadonnées est suivie de 12 octets de bourrage puis du contenu du fichier (soit la taille du fichier arrondie au multiple de 512 supérieur) puis le fichier suivant (métadonnées et contenu) commence.

Exercice 20 (Taille des deux premiers fichiers dans une archive)
Modifiez votre commande `lstar` afin qu'elle lise le premier bloc de métadonnées UStar d'une archive sur son entrée standard, trouve la taille du fichier, puis passe au second bloc de métadonnées et affiche les tailles des deux fichiers. Ces valeurs devront toujours être égales à celles affichées par `tar -tv` (troisième champ). On ne s'attend pas à ce que la commande fonctionne correctement si l'archive ne contient qu'un seul fichier. Typiquement `tar -H ustar -c vide test | lstar` affichera 0 et 20. □

La fin de l'archive est signalée par un bloc de métadonnées rempli d'octets nuls. En particulier, les champs `magic` et `version` ne contiennent donc pas les valeurs attendues.

Exercice 21 (Liste des tailles des fichiers)
Améliorez votre commande `lstar` pour afficher la liste des tailles de tous les fichiers de l'archive en vérifiant tous les champs `magic` et `version`. □

Exercice 22 (Liste des fichiers)
Complétez votre commande pour lister toutes les informations comme `tar -tv`. En particulier :
- affichez les chemins complets des fichiers (vous pourrez commencer en affichant seulement le nom, puis afficher le préfixe dans un deuxième temps),
- affichez le type,
- affichez les droits d'accès (l'opérateur `&` est le plus adapté à tester si un bit donné est nul ou pas dans un entier)
- affichez la date de modification en utilisant `localtime`(3) et `strftime`(3).

Testez votre décodage en créant différentes archives. □

Dans les exercices précédents vous devriez avoir utilisé `lseek` pour passer d'un fichier au suivant rapidement. Malheureusement, si vous essayez de lire le contenu d'une archive depuis un tube, par exemple par une commande comme :

```
tar -H ustar -c vide test | ./lstar
```

l'appel système `lseek` échoue (aviez-vous bien mis un `assert` pour détecter ce problème ?). Dans ce cas, il est impossible de passer directement au fichier suivant, il faut explicitement lire tous les octets du contenu (même si c'est juste pour les ignorer).

Exercice 23 (Liste des fichiers sans `lseek`)
Améliorez votre commande pour gérer le cas des tubes : dans le cas précis où `lseek` échoue parce que l'entrée standard est un tube, utilisez une boucle de lectures par bloc de 512 octets pour sauter le contenu des fichiers. □

Le champ `chksum` permet de vérifier que l'archive n'a pas été corrompue (par exemple lors d'une copie). Ce champ contient la somme de tous les octets (considérés comme des entiers non-signés, c'est-à-dire `unsigned char`) de la structure de métadonnées en considérant que le champ `chksum` lui-même contient 8 octets « espace » (`' '`).

Exercice 24 (Vérification de la somme de contrôle)
Modifiez votre commande `lstar` pour qu'elle affiche un avertissement si la somme de contrôle est incorrecte pour une entrée. Créez une archive corrompue (en modifiant au moins un octet d'une archive) pour la tester. □

2 Bibliothèque d'entrées/sorties avec tampons

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.

Chaque invocation d'un appel système (notamment `read`, `write`, etc.) nécessite le transfert du flux d'exécution de l'espace utilisateur vers l'espace noyau, qui est *beaucoup* plus lent qu'un appel de fonction. Il est donc plus efficace de lire ou d'écrire beaucoup de données à chaque appel système plutôt que de faire un appel par octet.

Les fonctions de la bibliothèque manipulent des *flux* (*stream*) implémentés par des `FILE`. Le tampon associé à un flux est dans l'espace utilisateur : chaque invocation d'une opération d'entrée/sortie ne nécessite plus le passage en mode noyau. Ce dernier n'a lieu que quand le tampon est plein (si on écrit) ou vide (si on lit). La taille de ces tampons est choisie pour des raisons de performance.

2.1 Implantation d'une bibliothèque d'entrées/sorties avec tampons

On propose ici une implantation d'une version très rudimentaire d'une bibliothèque d'entrées/sorties avec tampons. Seuls les équivalents des primitives `fopen()`, `fclose()`, `fgetc()`,

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`.

et `fputc()` seront implantés. Pour se distinguer des versions standard, on utilisera le préfixe « k ».

La gestion d'un tampon utilisateur nécessite de mémoriser un certain nombre d'informations. On pourra penser notamment à la 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. On définira donc une structure de données, `KFILE`, pour enregistrer toutes les données associées à un fichier ouvert.

Exercice 25 (Écrire un caractère)
Proposez une fonction `int kputc(int ch, KFILE* kfile)` prenant en argument un caractère et un pointeur vers une structure `KFILE` et écrivant ce caractère dans le flux correspondant. Cette fonction retournant le caractère écrit dans le flux si tout s'est bien passé et `EOF` sinon. L'idée essentielle de cette fonction est de ne provoquer un appel système `write` que si le tampon mémoire du `KFILE` est rempli.

Vous définirez en même temps la structure `KFILE` en fonction des besoins pour écrire la fonction `kputc()`. □

Exercice 26 (Ouvrir et fermer un flux)
Proposez des fonctions :

- `KFILE* kopen(const char *path)` qui ouvre un `KFILE`,
- `int kclose(KFILE* kfile)` qui ferme un `KFILE`.

Vous pourrez profiter de l'écriture de ces fonctions pour définir des fonctions auxiliaires pertinentes. □

Exercice 27 (Lire un caractère)
Proposez une fonction `int kgetc(KFILE* kfile)` qui lit un caractère dans le flux et le retourne, ou retourne `EOF` si la fin du fichier est atteinte. Vous pourrez étendre la structure `KFILE` et modifier les fonctions précédentes si nécessaire. □

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.

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 le contenu des fichiers dont les noms sont fournis en paramètre.

Exercice 28 (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 premier argument de la commande sera la taille de tampon à utiliser.

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. Copiez dans un fichier `script.sh` le contenu suivant :

```
#!/bin/bash
# Exemple de script shell qui mesure le temps d'exécution de
# "macommande" avec des arguments différents

/usr/bin/time -f '%e %U %S' macommande a
/usr/bin/time -f '%e %U %S' macommande b
/usr/bin/time -f '%e %U %S' macommande c
```

Il s'agit ensuite de rendre exécutable ce fichier pour pouvoir le déclencher comme un exécutable normal :

```
$ chmod +x script.sh
$ ./script.sh
[...]
```

Les shells disposent par ailleurs, comme les langages de programmation plus classiques, de structures de contrôle. En `bash` par exemple, vous pouvez faire une boucle :

```
for((i = 0; i < 10; i++)); do
  printf '%d' "$i"
done
```

pour éviter des duplications de code. Consultez la page de manuel de `bash` pour en savoir plus.

Vous ferez des tests avec des tampons de 1 octet à 8 Mo en doublant la taille à chaque fois (1 octet, 2 octets, 4 octets, 8 octets, ...).

1. Donnez une implantation de la commande.
2. Testez cette commande sur un fichier de taille conséquente en entrée, au moins quelques dizaines de Mo.

   Vous ferez attention au type du système de fichiers sur lequel réside ce fichier (option `-T` de la commande `df`) : si le fichier est sur un système de fichiers réseau (comme NFS), vos tests porteront sur les performances réseau plutôt que celles des appels système !

   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 précédente 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 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 page suivante pour un exemple. □

□

Exercice 29 (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 les flux `stdin` et `stdout` et comparez les temps d'exécution avec la version précédente. (Consultez les pages de manuel de `fgetc` et `fputc` pour en savoir plus). □

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
```