Fonctions de base d'Unix concernant les fichiers

©A. Dragut
Université Aix-Marseille I.U.T.d'Aix en Provence - Département Informatique
Dernière mise à jour :7/10/2012

Rappel structure des répertoires pour les TP

- *allez dans sous-répertoire **TP***
- *allez dans le sous-répertoire **tpsystem***
- *vous deviez y avoir créé le sous-répertoire **tpfile**, correspondant à l'unité de TP commencé en intro*
- *vous deviez également y avoir créé les sous-répertoires **exo_01** et **exo_02** et travaillé dedans*
- *copiez (avec l'option **-r**), le répertoire **exo_02** pour créer encore cinq autre répertoires nommés **exo_03**, **exo_04**, etc.*

*Vous mettrez tous les fichiers créés au cours de chaque exercice des TP dans le répertoire **dirfile** de l'exercice courant, qui sera considéré comme **répertoire courant**. En passant d'un exercice à l'autre vous allez recopier le contenu de **include** et, souvent, y rajouter également des fonctions. Vous mettrez dans le répertoire*

- **include** tous les **.h** contenant des en-têtes avec des déclarations et des définitions inline, pour les
  - wrappers
  - exceptions
  - quelques fonctions utilitaires

  et un fichier de définitions pour **make**, appelé **INCLUDE_H**.
- **Pour faire de la place dans vos répertoires**, lorsque vous finirez ce TP et passerez au suivant
  - **Effacez bien tous les exécutables et les objets**, soit
    - avec le `make clean` (que vous mettrez en place) dans le `dir*` de chaque `exo_*`, soit
    - avec une simple commande

      ```
      rm */*/*.{o,a,run}
      ```

      dans le répertoire `tpfile`. La même procédure devra bien entendu être appliquée pour la suite.
  - **Compressez le contenu de chaque répertoire d'exercice** avec `tar` et `gzip`, avec la commande suivante (exemple pour `exo_01`) :

    ```
    tar cvf - exo_01 | gzip -9 - > exo_01.tar.gz
    ```

    Attention aux espaces autour des tirets seuls, ces tirets demandent l'utilisation de l'entrée, respectivement sortie standard (au lieu d'utiliser des noms de fichiers). Ainsi on peut enchaîner ces commandes et tout déposer directement dans `exo_01.tar.gz` Par contre, le `-9` est bien ensemble, cette option demande la meilleure compression.

- **Respectez** l'ordre des exercices, car il y a une progression déterminée, qu'il est impératif de suivre.

Attention: à la fin de ce TP vous devrez rendre une feuille avec des réponses à des questions du TP.

Sommaire

Liste des fonctions et concepts système étudiés

*On étudie les opérations d'entrée-sortie sur les fichiers : **`open()`**, **`read()`**, **`write()`**, **`close()`** (exercices de 03 à 04), la destruction d'un fichier avec **`unlink()`** (exercice 05), les fonctions système pour le contenu des répertoires (exercice 06) **`chdir()`**, **`getcwd()`**, **`opendir()`**, **`readdir()`**, **`closedir()`**, **`lstat()`** et enfin l'accès concurrent en lecture et écriture (exercice 07).*

exo_03, copie fichier, taille du tampon

Dans cet exercice

vous écrivez deux fonctions

- ***`FileCopy()`** pour copier des fichiers utilisant les fonctions **`open()`**, **`read()`**, **`write()`**, **`close()`** en utilisant des tampons mémoire de taille différentes obtenus avec la fonction*
- ***`donneTailleMorceau()`***

*le corps principal du main entre le **`try-catch`** qui appelle d'abord **`donneTailleMorceau()`** e après **`FileCopy()`** pour montrer la différence en temps d'exécution pour les différentes tailles de tampons.*

Travail à faire

- ***Récupérez les fichiers modelmain.cxx, nsSysteme.h, nsSysteme.cxx et mettez-les au bon endroit, après avoir brièvement regardé les nouveaux wrappers.***
- ***Écrivez le wrapper (déclaration et définition inline) de la fonction **`write()`** dans l'espace de noms **`nsSysteme`** (du fichier **`nsSysteme.h`** . Utilisez man 2 write.***
- ***Dans le fichier **`nsSysteme.cxx`** , dans un nouveau **`espace de noms nsFctShell`**, définissez la fonction***

```
FileCopy(const char *dest, const char *source, const size_t NbBytes);
```

qui

- ***ouvre le fichier source en mode lecture à l'aide d'un appel système***

- ouvre le fichier destination en mode écriture, création et troncature l'aide d'un appel système
- réserve un tampon mémoire de `NbBytes +1 octets`
- dans une boucle, tant qu'on peut lire du fichier source (i.e. l'appel système `Read()` n'est pas nul -- lisez dans le man 2 read, le paragraphe de la valeur renvoyée)
  - lit (au plus) `NbBytes` octets dans le tampon
  - écrit le nombre d'octets ainsi lus, depuis le tampon, dans le fichier destination, avec `Write()`
- ferme enfin les deux fichiers l'aide d'appels système

- Dans le fichier `exo_03.cxx` , dans l'`espace de noms anonyme`, définissez la fonction

```
std::size_t donneTailleEnOctets(const string &comment, const string &nomFichier);
```

  Cette fonction utilise un appel à `Stat()` pour obtenir les tailles (en octets) des différents tampons mémoire qui peuvent être utilisés (variable NbBytes du `read(), write()`) pour les E/S sur un fichier. L'appel remplie une `struct stat S` . Ses champs contiennent les informations desireés. Le premier `string` de `donneTailleEnOctets()` peut avoir comme valeur
  - "`char`" : on demande une copie par caractère et alors on rend taille d'un caractère, c.à.d. 1 octet
  - "`all`" : on demande de copier le fichier source en entier et alors on rend la taille du fichier (champ `.st_size` de la `struct stat`)
  - "`block`" : on demande de copier le fichier source par en utilisant des tampons de la taille d'un bloc standard pour les E/S (champ `.st_blksize` de la `struct stat`)
- Dans le même fichier `exo_03.cxx`, écrivez la fonction `main()` qui
  - prend trois arguments : comment copier, la source et la destination
  - appelle `donneTailleEnOctets()` pour connaitre la taille en octets du tampon à utiliser pour copier la source
  - affiche la taille du tampon ainsi obtenue
  - appelle enfin `FileCopy()` pour copier le fichier source sur la destination en utilisant la taille du tampon donnée par `donneTailleEnOctets()`
- Compilez le programme, et essayez-le pour chacune des valeurs du premier paramètre, sur un fichier assez grand. On peut prendre par exemple l'exécutable du shell `bash` , à trouver avec la commande `which bash`. Servez-vous également des commandes `time` et `strace`.

```
time ./exo_03.run all /bin/bash copieBash
```

  La commande `strace` est un puissant outil de débogage. Utilisée comme suit, elle écrit dans le fichier `Trace.txt` tous les appels des fonctions système `open()`, `close()`, `read()` et `write()` effectués lors de l'exécution de la commande `./exo_03.run`

```
strace -o Trace.txt -e trace=read,write,open,close ./exo_03.run block /bin/bash copie
```

- QUESTION: Écrivez sur une feuille à rendre les temps d'exécution obtenus pour la copie de /bin/bash avec toute les options. Sauvegardez le contenu de votre fichier Trace.txt et envoyez le par courriel à votre enseignant.
- Rajoutez temporairement le drapeau suplimentaire `O_SYNC` dans l'appel d' `open()` qui ouvre le fichier destination. L'option `O_SYNC`, force l'écriture physique de la destination à chaque appel de `Write()`. Le temps supplémentaire peut ainsi devenir écrasant, pour un mauvais choix de taille de tampon ! Compilez et testez ensuite avec `time` pour voir la différence. N'oubliez pas de l'effacer.
- Rajoutez temporairement le drapeau suplimentaire `O_APPEND` dans l'appel d' `open()`qui ouvre le fichier destination. Écrivez un fichier source `test.txt` contenant "test open". Compilez et testez ensuite deux fois avec `test.txt` et regardez le contenu de la

destination pour voir la différence. N'oubliez pas d'effacer le drapeau.

- *QUESTION: Quelle est la différence avec et sans O_APPEND?*

---

exo_04 Lire et écrire différents types de données -- `char, int, char *`

Dans cet exercice

on se familiarise avec le stockage/récupération de données dites formattées dans les fichiers.

Quoi de neuf

En général, à l'intérieur d'un programme on utilise des variables qui peuvent avoir des types différents : entier, caractère, virgule flottante, etc. Lors de la compilation et exécution, des zones mémoire de longueur prédeterminée sont alors réservées pour chaque telle variable.
Le but de cet exercice est de vous faire lire de telles valeurs depuis un fichier, en les mettant directement en mémoire, pour initialiser de telles variables.
Les langages C et C++ disposent d'un opérateur appelé `sizeof()` qui renvoie le nombre d'octets occupés en mémoire par un type (ou une variable) donnée. Il nous sera donc très utile comme argument à donner à `Read()` respectivement `Write()`, pour transférer le bon nombre d'octets.

Travail à faire

- ***Récupérez le fichier donnees.fic et placez-le dans le répertoire `dirfile` de l'exo 4. Ce fichier contient des donnés selon le format suivant, divisé en quatre zones :***
  1. ***d'abord -- un seul caractère***
  2. ***ensuite, un entier (donc occupant `sizeof(int)` octets).***
  3. ***ensuite, un nombre de caractères égal à la valeur du nombre entier du point précédent***
  4. ***enfin, un autre nombre (quelconque) de caractères, jusqu'à la fin du fichier***

  Donc par exemple, si l'entier en question valait 5, alors la zone 1 se trouve à l'octet 0 et occupe un octet, la zone 2 commence à l'octet 1 et occupe `sizeof(int)` octets, la zone 3 commence à l'octet 1+`sizeof(int)` et occupe 5 octets, et la dernière zone commence à l'octet 6+`sizeof(int)`.
- ***Écrivez un programme (récupérer le squelette du `main()` de l'exercice précédent -- `try, catch`) qui***
  - ***ouvre ce fichier***
  - ***lit soigneusement les renseignements de chaque zone***
  - ***les affiche à l'écran***
- ***QUESTION: Sauvegardez le contenu de votre fichier ainsi lu et envoyez le par courriel à votre enseignant.***
- ***OPTIONNEL : À partir de l'entier ainsi lu, construisez (et affichez) un autre entier qui a les premiers `sizeof(int)/2` octets mis à la place des derniers `sizeof(int)/2` octets, et viceversa (on change ainsi l'endianness -- big endian et little endian sont deux conventions de stockage en mémoire des valeurs numériques). Donc par exemple, si `sizeof(int)` vaut 4, alors les octets 2 et 3 prennent la place des octets 0 et 1, et inversement. Indication: vous pouvez vous servir des ET et respectivement OU binaires pour isoler les octets, et respectivement les "placer" dans l'entier destination. Affichez également tous les bits des deux entiers (initialement lu, et endianness-inversé)***

Petit mot sur l'ordre des octets et des bits

Lorsqu'on lit un nombre en binaire écrit sur un papier, l'usage fait du chiffre le plus à gauche le plus significatif car associé à la plus grande puissance de 2 dans l'écriture du nombre. La colonne des 8 représente le bit le plus significatif (msb, most significant bit), puisqu'elle contient la plus grande valeur; et la colonne des 1 représente le bit le moins significatif (lsb, least significant bit), puisqu'elle contient la plus petite valeur. L'ordre usuel sur papier est appelé ordre "big endian" (grand boutien). Donc le nombre 3 (base dix) sur un octet s'écrit

0000 0011 (msb<-lsb)

L'autre ordre utilisé est l'ordre du petit boutien (en anglais-- little endian bit ordre), le bit le plus à droite étant celui le moins significatif. Donc le nombre 3 (base dix) s'écrit en héxa lsb sur un octet

1100 0000 (lsb->msb)

Le plus grand probleme est l'ordre dans lequel les octets ("bytes") sont organisés en mémoire ou dans une communication. Ce probleme est appelé "endianness". En général en mémoire on va placer un nombre en binaire dans l'ordre inverse des octets. L'ordre des bits suit généralement l'ordre des octets. Soit le nombre 3 (en base dix) sur quatre octets, pour une structure de mémoire basée sur une unité atomique de 1 octet et d'un incrément d'adresse de 1 octet.

En Big Endian il est:

byte/octet addr	0	1	2	3
bit offset	01234567	01234567	01234567	01234567
binaire	00000000	00000000	00000000	00000011

Si on le lit en Little Endian directement, la même écriture binaire sera interprétée différemment

byte/octet addr	3	2	1	0
bit offset	76543210	76543210	76543210	76543210
binaire	00000000	00000000	00000000	00000011

Le nombre lu sera en Big Endian un autre:

byte/octet addr	0	1	2	3
bit offset	01234567	01234567	01234567	01234567
binaire	11000000	00000000	00000000	00000000

Il faut connaître l'ordre pour lire correctement. Il faut renverser l'ordre ("swapper" les bits) pour ne pas lire un autre nombre.
Si vous souhaitez vous amuser avec ces concepts en regardant un petit programme et en faisant tourner, venez par ici.
Plus de details dans le Linux Jurnal (Byte and Bit Order Dissection, By Kevin Kaichuan He) et sur le site IBM developerworks Writing endian-independent code in C, by Harsha S. Adiga.

Les processeurs Intel (x86 et Pentium) travaillent en mode Little Endian pour les octets, tandis que les processeurs Motorola (la série 680x0) travaillent en mode Big Endian pour les octets. Le MacOS était Big Endian, et il est devenu Little Endian sur Intel. Si des zones de la mémoire vive contenant des nombres sont copiées telles quelles dans un fichier, donc directement, le fichier ne pourra être lu (sans traitement particulier) que par les ordinateurs ayant une unité centrale travaillant avec le même ordre. Chaque fois qu'un ordinateur accède à fichier

audio/vidéo ou un flux multimédia, l'ordinateur a besoin de savoir comment le fichier est construit. Par exemple: si vous écrivez un fichier graphique (comme un fichier .JPEG, qui est Big-Endian format) sur une machine avec un CPU Little-Endian, vous devez d'abord inverser l'ordre des octets de chaque entier vous écrivez ou un autre programme "standard" ne sera pas capable de lire le fichier.

Le même problème de l'ordre d'octets se pose dans les transfers par réseau. Les protocoles d'Internet utilisent l'ordre Big Endian pour les octets, appelé donc aussi network byte ordre (ordre des octets du réseau). Il y a des fonctions qui vous aident à changer d'ordre, comme par exemple `htonl()` et `ntohl()`, sur 16 et respectivement 32 bits (HostTONework, etc.).

Plus sur les formats des fichiers et les protocols de bus et réseau sur le site d' Intel dans Endianness White Paper.

---

exo_05 effacement -- unlink() -- d'un fichier pendant l'accès

Dans cet exercice

on ouvre un fichier, on lit depuis, et on teste si le système nous permet de continuer à lire après avoir appelé `unlink()`

Que se passe-t-il ?

Puisque le système est multi-utilisateur et multi-processus, les fichiers peuvent être manipulés de manière concurrante. Prenons la succession d'événements suivante sur un fichier. Ouvrez deux xterms, créez un fichier de plusieurs lignes (faites par exemple `echo Coucou >> ~/Test.txt` quelques dizaines de fois -- touche "flêche vers le haut" pour rappeler la dernière commande), et puis

- *dans un des xterms, faire `less ~/Test.txt` et le laisser comme ça, au milieu du fichier*
- *dans l'autre xterm, lister le fichier avec `ls -l`, puis l'effacer avec `rm`, ressayer de le lister et voir qu'il n'y est plus, et*
- *revenir à l'xterm avec le less et continuer à faire défiler le fichier comme si de rien n'était.*

Maintenant on va detailler le mécanisme système qui permet l'implementation de cette concurrence.

Qu'y a-t-il à faire, alors, dans cet exercice ?

1. *Wrapper de `unlink()`*
2. *Essais d' effacement d'un fichier et tests d'accès après destruction*

Travail à faire

- *Dans le fichier `nsSysteme.h` rajoutez la déclaration et la définition `inline` de `Unlink()` (le wrapper de `unlink()`).*
- *Dans le fichier `exo_05.cxx`, écrivez la fonction `main()` qui*
  - *prend comme arguments un fichier `Source`*

    - *montre/vérifie la présence du fichier `Source` dans le répertoire courant (avec `system("ls -l")` pour montrer et avec `stat()` pour vérifier)*
    - *laisse au lecteur le temps de lire le message du `ls`*
    - *ouvre le fichier `Source`*
    - *efface le fichier `Source` avec `Unlink()`,*
    - *montre/vérifie à nouveau l'absence du fichier `Source` dans le répertoire courant*
    - *laisse au lecteur le temps de lire le message du `ls`*
    - *écrit sur la sortie standard les prémiers 10 charactères du fichier `Source`*
- ***QUESTION: Grace à quelle variable avez-vous reussi à copier le fichier?***

OPTIONNEL Exo 6 Fonctions sur les répertoires

exo_06 1. Répertoire courant

Dans cet exercice

vous écrivez un programme qui affiche et change le répertoire courant, avec les fonctions `getcwd()` et `chdir()`.

Quoi de neuf

Tout processus a son répertoire courant. La fonction `char * getcwd(char *buf, size_t size)` remplit `buf` avec son chemin (NTCS). Si la NTCS est de longueur supérieure ou égale à size, elle retourne en erreur avec le pointeur à `NULL`. La fonction `int chdir(const char *path);` change effectivement ce répertoire pour celui donné en argument, renvoyant zéro si succès, et -1 si échec.

Travail à faire

- ***Écrivez (dans `nsSysteme.h`) les wrappers de `getcwd(), chdir()`. Utilisez `man 3 getcwd(), chdir()`***
- ***Écrivez (dans `exo_05_1.cxx`) la partie `try ... catch` du `main()` qui***
    - ***déclare un tableau de 1024 caractères pour stocker le nom d'un répertoire***
    - ***change le répertoire courant en celui donné en `argv[1]`***
    - ***obtient à nouveau le répertoire courant et l'affiche***

exo_06 2. Émulation partielle de la commande `ls -la` sur un répertoire

Dans cet exercice

vous vous servirez de `man 2 chdir()`, `man 3 opendir()`, `man 3 readdir()`, `man 2 lstat()`, ainsi que de quelques macros et masques, et puis de `man 3 closedir()`, pour écrire un `main()` qui énumère le contenu d'un répertoire `argv[1]` ainsi

```
drwx    4096    .
drwx    4096    ..
-rw-    645     Makefile
-rw-    1016    exo_06.h
-rw-    1489    exo_06.cxx
-rw-    28704   exo_06.o
drwx    4096    sousRepertoire
```

```
-rwx    17440   exo_06.run
lrwx    10      lienSymbolique
.rw-    0       ceciEstBienUnePipe
```

Le premier caractère indique le type de l'élément du répertoire. Nous utilisons le '.' pour tout ce qui n'est pas répertoire, lien symbolique ou fichier. De même, seulement les droits d'accès pour le propriétaire sont affichés, suivis de la taille en octets (comme auparavant).

Quoi de neuf

- `opendir()` -- pour ouvrir un répertoire, accessible ensuite par le biais d'un pointeur vers `DIR` -- un "directory stream" (wrapper `OpenDir()`).
- `readdir()` -- pour parcourir un répertoire ainsi ouvert, élément par élément (un par appel, avance "toute seule" d'un appel à l'autre). Rend un pointeur vers une structure `dirent`, dont le membre `d_name` contient le nom de l'élement courant du répertoire. Rend le pointeur nul à la fin du parcours. Il faut donc l'appeler de manière répétée pour obtenir tous les éléments (wrapper `ReadDir()`).

```
struct dirent {
            ino_t          d_ino;      /* numero d'inoeud */
            off_t          d_off;      /* decalage jusqu'a la dirent suivante */
            unsigned short d_reclen;   /* longueur de cet enregistrement */
            unsigned char  d_type;     /* type du fichier */
            char           d_name[256];/* nom du fichier */
};
```

- `closedir()` -- pour fermer le `DIR` (wrapper `CloseDir()`).
- `lstat()` -- même chose que `stat()`, mais sans poursuivre les liens symboliques
- tests, où `mode` est la valeur du champ `st_mode` d'une `struct stat` remplie par `lstat()`
  - macros
    - `S_ISDIR(mode)` -- vrai s'il s'agit d'un répertoire
    - `S_ISREG(mode)` -- vrai s'il s'agit d'un fichier normal
  - masques -- à tester avec le ET binaire
    - `S_IFLNK & mode` -- vrai s'il s'agit d'un lien symbolique
    - `S_IRUSR & mode` -- vrai si le propriétaire a le droit de lire
    - `S_IWUSR & mode` -- vrai si le propriétaire a le droit d'écriture
    - `S_IXUSR & mode` -- vrai si le propriétaire a le droit d'exécution

Travail à faire

- Écrivez des wrappers (déclaration et définition inline) dans `nsSysteme.h`, selon les indications suivantes.
  - pour `lstat()`, `chdir()` et `closedir()` un simple test (non-zéro) de la valeur de retour de la fonction système suffit pour détecter l'erreur et lever l'exception, tout comme pour `Stat()`.
  - pour `opendir()` l'erreur survient lors d'un pointeur rendu nul par son appel. Détecter l'erreur et lever l'exception, mais n'oubliez pas de renvoyer l'information utile s'il n'y a pas d'erreur.
  - pour `readdir()`, l'erreur survient lors d'un pointeur nul ET lorsque `errno` n'est pas nulle. Suivez ces pas :
    1. `errno` prend la valeur 0
    2. on déclare et définit un `ddirent * pEntry` récupérant le résultat de l'appel de `readdir()`
    3. si `pEntry` est nul et `errno` n'est pas nulle, alors lever l'exception
    4. rendre `pEntry`

Pourquoi ces complications pour `ReadDir()` par rapport à `OpenDir()` ?
Car un pointeur rendu nul par `readdir()` signifie soit une erreur, soit la fin du parcours du répertoire sans erreur. C'est donc `errno` qui permet de distinguer ces deux cas. De plus, comme `errno` est une variable globale, assignable n'importe où, il faut la mettre à zéro dans le wrapper avant l'appel de `readdir()`. Pourquoi ? Puisqu'on ne sait pas quel autre appel système précédent a pu échouer, mais sans terminer fatalement le programme. On aurait ainsi une `errno` "sale", et il faut donc la nettoyer dans ce cas.
Enfin, le cas de fin de parcours, sans erreur, est à gérer par l'utilisateur du wrapper, qui peut donc ainsi rendre un pointeur nul sans lever d'exception dans ce cas-là.

- ***Écrivez le `main()` qui prend un seul argument -- le chemin du répertoire, et qui énumère son contenu comme illustré et expliqué plus haut, à l'aide des wrappers de ces outils. Les pas à faire sont les suivants :***
  - ***ouverture du répertoire avec `OpenDir()`, récupérant le pointeur `pDir` vers un `DIR`***
  - ***changement de répertoire courant avec `ChDir()`(pour simplifier l'appel de `lstat()` -- chemin relatif "immédiat")***
  - ***tant qu'il y a encore des entrées à lire dans le répertoire `pDir` faire***
    - ***retrouvez la nouvelle `pEntry` avec `ReadDir()`***
    - ***obtenez la taille et le type de `pEntry` à l'aide d'une variable `S` de type structure `stat` en utilisant `Lstat()`***
    - ***Attention: `Lstat()` a besoin du nom de fichier, tandis que `pEntry` renvoie un pointeur envers une `struct dirent`***
    - ***affichez le type et les droits de l'entrée courante en utilisant les macros** (par exemple `S_ISDIR`, `S_IRUSR`)*
    - ***affichez la taille et puis le nom lui-même***
  - ***fin de boucle tant que***
  - ***fermer le `DIR` avec `CloseDir()`***
- ***QUESTION: Écrivez sur une feuille à rendre le resultat de l'exécution pour ../include***

OPTIONNEL exo_07 accès concurrent en lecture, respectivement lecture/écriture

Dans cet exercice

on finalise un programme qui à l'aide des fonctions système

- *écrit des caractères un par un dans un fichier, ou bien*
- *lit des caractères un par un depuis un fichier*

selon les arguments sur la ligne de commande. On se sert ensuite de ce programme pour étudier ce qui se passe lorsque plusieurs processus accèdent au même fichier. On voit ainsi que

- *les lectures n'interfèrent en rien*
- *les écritures interfèrent, en s'écrasant l'une l'autre (la dernière dans le temps "gagne")*
- *une écriture est tout de suite visible chez tous les lecteurs*

1. *On finit le programme pour les essais*
2. *On fait les essais lectures concurrentes*
3. *On fait les essais écritures concurrentes*
4. *On fait les essais lectures/écritures concurrentes*

exo_07 1. Programme

Dans cet exercice

- *on finit le programme capable d'écrire respectivement de lire des fichiers, de manière temporisée*

Comment ça marche

Le programme prendra comme arguments

- *soit* `lire <combienALaFois> <delaiInitial> <delai> <nomFic>`
- *soit* `ecrire <1erChar> <combienAuTotal> <delaiFinal> <delaiInitial> <delai> <nomFic>`

Par exemple, pour les arguments `ecrire A 4 5 2 1 monFic.txt`**, le programme**

- ***attendra deux secondes***
- ***écrira dans le fichier*** `monFic.txt` ***les caractères*** `A B C D`***, attendant une seconde d'un caractère à l'autre***
- ***attendra encore cinq secondes***
- ***fermera le fichier***

Travail à faire

- ***Récupérez le fichier exo_07.cxx et placez-le au bon endroit.***
- ***Dans ce fichier il y a un endroit marqué*** `TRAVAIL A FAIRE` ***en commentaire, avec les indications nécessaires. Il faut donc écrire une boucle pour***
  - ***lire depuis le descripteur*** `fd`***, un caractere à la fois***
  - ***afficher aussitôt chaque caractere lu, rappelant aussi les paramètres de lancement***
  - ***mettre*** `qContinue` ***à faux lorsqu'on a atteint la fin du fichier***
- ***Regardez brièvement le reste du programme, pour comprendre comment il fonctionne***
- ***Compilez, et testez***
  - ***l'écriture****, en créant un fichier* `exo_07p1.txt` *contenant les cinq premières lettres minuscules de l'alphabet.*
  - ***la lecture****, en lisant depuis ce fichier*
  - ***des essais de lectures concurrentes : lancez simultanément deux fois l'exécution de ce programme en lecture depuis le fichier*** `exo_07p1.txt` ***(créé au point précédent) ainsi : munissez-vous de deux terminaux (visibles simultanément), et dans un des deux, lancez***

    ```
    exo_07.run lire 1 1 1 exo_07p1.txt
    ```

    et dans l'autre, lancez

    ```
    exo_07.run lire 2 2 1 exo_07p1.txt
    ```

- ***QUESTION: Avez-vous l'impression que l'exécution d'un processus (la progression dans le fichier) gêne l'exécution de l'autre ? Écrivez votre réponse sur une feuille à rendre à votre enseignant.***

exo_07 2. Essais écritures concurrentes

Dans cet exercice

on fait tourner en parallèle deux processus qui accèdent en écriture au même fichier, montrant qu'ils interfèrent, car l'ensemble des tampons d'un fichier physique est unique. Ces tampons sont utilisés concurramment par toutes les applications écrivant dans le fichier, et la dernière écrase les valeurs des précédentes.

Travail à faire

- ***Comme au point précédent, munissez-vous de deux terminaux (visibles simultanément), et dans un des deux, lancez***

  `exo_07.run ecrire a 3 2 3 2  exo_07p2.txt`

  et dans l'autre, lancez

  `exo_07.run ecrire  A 3 3 0 4 exo_07p2.txt`

- ***QUESTION: Écrivez sur une feuille à rendre à votre enseignant le chronogramme de l'exécution des deux processus, c'est-à-dire le déroulement dans le temps de chaque écriture. Il faut donc faire un tableau avec autant de colonnes que de secondes écoulées et une ligne par processus, avec ce qu'ils auront écrit et au bon moment.***
  Exemple de chronogramme

temps	***0***	***1***	***2***	***3***	***4***
processus 1		***A***		***B***	***F***
processus 2	***a***	***b***	***c***		***F***

  Ici F représente la fermeture du fichier L'opération d'écriture prend un temps négligeable ici, donc il faut compter toutes les cases.

En consultant le contenu du fichier résultat ***`exo_07p2.txt`*** *, nous constatons que les deux applications partagent le même fichier physique, et les mêmes buffers. En effet, chaque* ***octet*** *du fichier contient la dernière valeur à avoir été écrite, quel que soit le processus qui l'a écrite et quel que soit le moment de la fermeture : chaque processus écrase les valeurs précédentes, et le pointeur avance* ***quand un même processus réécrit*** *(i.e. il n'y a que trois caractères dans le fichier). S'il y avait plusieurs buffers (un par application), ce serait le dernier buffer modifié qui serait sauvegardé physiquement, et les caractères ne seraient pas mélangés.*

exo_07 3. Essais lectures/écritures concurrentes

Dans cet exercice

on fait tourner en parallèle deux processus qui accèdent un en écriture et l'autre en lecture au même fichier, montrant qu'une écriture est tout de suite visible chez les lecteurs, à nouveau car les tampons sont uniques pour un fichier physique donné. Quelle que soit sa localisation dans la mémoire (mémoire principale - buffers d'E/S du système - ou mémoire secondaire - fichier disque proprement dit), toute modification du contenu par l'un des processus est immédiatement visible de tous les processus.

Travail à faire

- ***Créez un fichier*** ***`exo_07p3.txt`*** ***qui doit contenir les 26 lettres de l'alphabet.***
- ***Lancez le processus 1 avec les paramètres*** ***`lire 5 0 4 exo_07p3.txt`*** ***et en même temps le processus 2 avec les paramètres*** ***`ecrire 1 7 0 1 0 exo_07p3.txt`*** ***.*** *On constate que les modifications effectuées par le processus 2 sont immédiatement visibles par le processus 1. Le processus 1 lit en une seule fois les cinq premiers caractères, puis il attend deux secondes, et puis il recommence. Le processus 2 attend une seconde, et écrit les caractères 1 2 3 4 5 6 7 dans le fichier. Donc le processus 1 lit d'abord les lettres, mais à sa deuxième itération il ne peut plus lire que deux caractères, lesquels sont déjà des chiffres.*

Solutions

Solution exo_03

exo_03.cxx
ModelMain.cxx
Makefile
nsSysteme.cxx
CExc.h
INCLUDE_H
nsSysteme.h

Solution exo_04

exo_04.cxx
exo_04W.cxx
exo_04Opt.c
exo_04OptVar.cxx
Makefile
nsSysteme.cxx
CExc.h
INCLUDE_H
nsSysteme.h

Solution exo_05

exo_05.cxx
Makefile
nsSysteme.cxx
CExc.h
INCLUDE_H
nsSysteme.h

Solution exo_06

exo_06_1.cxx
exo_06_2.cxx
Makefile
nsSysteme.cxx
CExc.h
INCLUDE_H
nsSysteme.h

Solution exo_07

exo_07.cxx
exo_07p3.txt
