Programmation Système

Fiche 1 : Appels système pour les entrées/sorties

Cette fiche ainsi que les fichiers à télécharger sont disponibles sur le site
https://gforgeron.gitlab.io/progsys/

Dans tous les exercices on cherchera à détecter systématiquement les appels système infructueux, c'est-à-dire produisant un code de retour indiquant une erreur. Pour simplifier le traitement on pourra utiliser la macro `check` définie dans `error.h`. Voici un exemple d'utilisation :

```
int fd = open (filename, O_RDONLY);
check (fd != -1, "Cannot open file %s", filename);
```

Si l'appel système `open` retourne -1, le message affiché contiendra une partie spécifique au contenu de la variable errno entre parenthèses, et possèdera la structure suivante :

```
[mymachine] ./prog
prog.c:27: Error: Cannot open file toto.txt (No such file or directory)
```

Sous VSCode, vous pouvez utiliser Ctrl+Click pour ouvrir le fichier prog.c à la ligne incriminée.

Exercice 1.1 Il s'agit de programmer la commande `tee output-file-name` qui copie à la fois dans le fichier désigné en paramètre et sur sa sortie standard les données présentes sur son entrée standard.

1. Lire le manuel de la commande unix `tee` et tester la version disponible sur votre machine. Lorsque l'on lance directement la commande sous le shell, il sera nécessaire d'entrer `CTRL-D` pour signifier la fin du fichier d'entrée.
2. Programmer une version simplifiée de la commande `mytee.c` qui écrit seulement sur la sortie standard. Tester.
3. Compléter la commande pour écrire également dans le fichier passé en paramètre.

Exercice 1.2 Nous allons voir comment se programme une commande telle que `cp` pour copier efficacement un fichier dans un autre. L'objectif de cet exercice est non seulement de mettre en œuvre les appels système de base (ouvrir, lire, écrire et fermer un fichier) mais aussi de comprendre comment on utilise de tels appels système pour programmer des bibliothèques d'entrées/sorties de plus haut niveau.

1. Compléter le code source `copy.c` de la commande `copy` qui a deux paramètres :

   ```
   copy input-file-name output-file-name
   ```

   qui recopie le contenu du fichier désigné par le premier paramètre dans le fichier désigné par le second. On utilisera les appels système `open(), close(), read(), write()`. On lira et écrira un seul caractère à la fois. On pourra vérifier à l'aide de la commande `diff` l'équivalence entre le fichier source et sa copie.

2. On va écrire la même commande mais en utilisant maintenant les appels de haut niveau de la bibliothèque `C` tels que `fopen(), fread(), fwrite()` et `fclose()`. Pour cela recopier votre programme dans le fichier `fcopy.c` et modifier ce programme.
3. Modifier le programme `copy.c` afin de prendre en compte un troisième paramètre : la taille du buffer de lecture/écriture. On utilisera la fonction `atoi()` pour convertir une chaine de caractère en nombre.
4. Utiliser la commande

   ```
   dd if=/dev/zero of=/tmp/toto count=100000
   ```

   pour créer un fichier `/tmp/toto` de 100 000 blocs de 512 octets (tous à zéro). Déterminer expérimentalement la taille optimale pour minimiser le temps de copie de ce gros fichier.
5. Reproduire l'expérience en ouvrant le fichier destination en utilisant le mode `O_SYNC` de linux.

Exercice 1.3 Il s'agit d'écrire deux commandes pour lire et enregistrer un entier de 64 bits de type `off_t` à une position donnée. Il s'agit ici d'écrire dans un fichier au format binaire le nombre donné en décimal par l'utilisateur. Réciproquement, il s'agit de lire un nombre dans le fichier de format binaire et de l'afficher à l'écran au format décimal. Voici les synopsis des commandes :

```
enregister-entier nom-du-fichier position valeur
lire-entier nom-du-fichier position
```

1. Écrire une version simplifiée de ces programmes où l'écriture et la lecture de l'entier se font qu'en première position. Tester vos programmes et utiliser la commande `hexdump` pour visualiser le contenu du fichier.
2. Étendre vos programmes afin de prendre en considération la position où l'on doit écrire/lire la donnée.
3. Que constate-t-on si l'on cherche à lire ou écrire au-delà de la fin du fichier ?

Exercice 1.4 On cherche à pouvoir accéder rapidement à toute ligne d'un gros fichier. Pour cela on va construire un *index* de toutes les positions du caractère `\n` dans le gros fichier. Pour retrouver la n-ième ligne du fichier, il suffira alors de lire le n-ième entier du fichier d'index pour y trouver l'emplacement de la n-ième ligne et de s'y rendre directement pour effectuer la lecture demandée.

1. Écrire le code de la commande `indexer file` qui produit un fichier `file.idx` mémorisant toutes les positions successives du caractère `\n` dans le fichier donné en paramètre.
2. Écrire le code de la commande `afficher file line` qui affiche la ligne demandée en utilisant le fichier d'index.

Exercice 1.5 À l'aide de l'appel système `dup2()`, faire en sorte que la commande `lire-entier` écrive ses messages d'erreurs dans le fichier `ERREURS-LIRE.log`.

Exercice 1.6 Dans le fichier source `redirection.c`, complétez le code de la fonction `rediriger_vers` de manière à ce que cette fonction redirige temporairement la sortie standard vers un fichier de nom `file`, puis exécute la fonction `f`, puis rétablisse la sortie standard d'origine.

Exercice 1.7 Interpréter les résultats produit par l'exécution du code suivant :

```c
int main()
{
  printf("hello ");
  write(1,"world", 5);
  return 1;
}
```

1. Que se passe-t-il si l'on remplace la première chaîne par `"hello\n"` ? Tester aussi lors d'une redirection dans un fichier.
2. Proposer une clarification du code conservant l'ordre des appels à `printf` et `write`.

Exercice 1.8

On dispose d'une commande `ecrire` dont le programme source est listé ci-dessous :

```c
int main(int argc, char *argv[])
{
  int i, fd;

  if(argc != 2) {
    fprintf(stderr, "Usage: ecrire <position_depart>\n");
    exit(1);
  }

  fd = open("fich", O_WRONLY | O_CREAT, 0666);

  for(i = atoi(argv[1]); i < 10; i += 2) {
    lseek(fd, i*sizeof(int), SEEK_SET);
    write(fd, &i, sizeof(int));
    sleep(1);
  }

  close(fd);
  return 0;
}
```

1. Décrivez précisément le contenu du fichier `fich` après l'exécution de la commande shell suivante :

```
[toto@jaguar.u-bordeaux1.fr] ecrire 0
```

2. Même question lorsque l'on exécute consécutivement ces deux commandes :

```
[toto@jaguar.u-bordeaux1.fr] ecrire 0
[toto@jaguar.u-bordeaux1.fr] ecrire 1
```

3. Que se passe-t-il si l'on exécute les deux commandes `ecrire 0` et `ecrire 1` en parallèle (en les lançant simultanément dans deux terminaux) ? Que pouvez-vous dire à propos du fichier `fich` ?