Programmation système avancée
TP n° 7 : Primitives de synchronisation

Traditionnellement, le fichier `/etc/motd` contient un message que l'administrateur système veut transmettre à tous les utilisateurs. Lorsqu'un utilisateur se connecte au système, son *shell* vérifie si ce fichier existe, et, si c'est le cas, affiche son contenu.
Le but de ce TP est d'écrire un programme qui implémente un équivalent à `/etc/motd` en utilisant de la mémoire partagée. À la différence de l'approche traditionnelle, il sera possible d'être notifié de tout changement à `/etc/motd`.
Nous vous fournissons un fichier `motd.h` qui contient les déclarations des onctions que vous devez implémenter. Vous devez écrire trois fichiers :

- `motd.c`, qui contiendra les fonctions utilisées par les deux programmes ci-dessous ;
- `motd-publish.c`, qui contiendra un programme qui change la valeur du *motd* ;
- `motd-listen.c`, qui contiendra un programme qui affiche la valeur du *motd.*.

Exercice 1 :

1. Dans le fichier `motd.c`, écrivez une fonction `motd_open` qui ouvre une zone de mémoire partagée POSIX nommée `motd`, en la créant si nécessaire. Il vérifie ensuite la taille de cette zone (`fstat`), et :
   - si la taille valait 0, il l'étend à `sizeof(struct motd)` octets (`ftruncate`) ;
   - si la taille n'est pas 0 mais est différente de `sizeof(struct motd)`, il retourne une erreur.

   La fonction *mappe* ensuite la zone de mémoire, et l'interprète comme une `struct motd`. Si la zone était initialisement vide, il initialise le sémaphore `mutex` à 1 et le sémaphore `condvar` à 0 (`sem_init`). La fonction retourne un pointeur sur la zone de mémoire *mappées*.
2. Écrivez une fonction `put_data` qui stocke une suite d'octets dans le champ `data` et sa longueur dans le champ `len`. Écrivez un programme `motd-publish.c` qui ouvre la zone de mémoire puis y stocke la chaîne passée en paramètre de ligne de commande. Sous Linux, vous pouvez tester votre programme en faisant :

   ```
   hexdump -C /dev/shm/motd
   ```

3. Le programme précédent ne protège pas la zone de mémoire contre les écritures simultanées. Écrivez une fonction `lock` qui effectue l'opération P sur le sémaphore `mutex`, et une fonction `unlock` qui effectue l'opération V. Ajoutez des appels à `lock` et `unlock` autour de l'appel à `put_data`.
4. Écrivez une fonction `get_data` qui retourne une copie des données stockées dans la zone de mémoire partagée. Écrivez un programme `motd-listen` qui affiche le contenu, en protégeant les accès à l'aide de `lock` et `unlock`.
5. Que se passe-t-il si votre programme est interrompu à l'aide de *Ctrl-C* lorsqu'il détient le *mutex* ? Vérifiez votre hypothèse en ajoutant un appel à `sleep` au bon endroit. Comment pourrait-on éviter le problème ?

Exercice 2 :

Les programmes précédents ne permettent pas d'être notifié d'un changement du *motd*. Dans cet exercice, nous allons implémenter une variable de condition à l'aide des champs `num_readers` et `condvar` de la structure `motd`.

1. Implémentez une fonction `await` qui fait (attention, l'ordre est important) :

```
num_waiters++
V(mutex)
P(condvar)
```

Implémentez aussi une fonction `broadcast_and_unlock` qui fait :

```
while(num_waiters > 0) {
    V(condvar)
    num_waiters--
}
V(mutex)
```

Modifiez le programme `motd-publish` pour qu'il appelle `broadcast_and_unlock` au lieu de `unlock`. Modifiez le programme `motd-listen` pour qu'il ait la forme suivante :

```
lock
while(1) {
     get_data
     print
     await
     lock
 }
 unlock
```

2. Que se passe-t-il si le programme `motd_listen` est interrompu à l'aide de *Ctrl-C* ? Vérifiez votre hypothèse à l'aide de `hexdump -C`. Comment pourrait-on éviter ce problème ?

3. Que se passe-t-il se la valeur de *motd* change entre l'appel à `await` et l'appel à `lock` ?