Systèmes d'Exploitation - Examen

12X009 - TP04

Noah Munz (19-815-489)

Département d'Informatique
Université de Genève

Mardi 31 Janvier 2023

Code: Lien GitHub du code
Rapport: Lien GitHub du rapport

Table of Contents

Table of Contents

Rappel : But du TP

- L'objectif général du TP est de créer un programme qui permet de faire de créer des "verrou d'enregistrement interactif".
- Le programme doit être capable poser un verrou et l'enlever sur une partie d'un fichier, de détécter la présence de verrous posés par les autres processus, le tout dans un mini "shell" (CLI basique)

Les principaux défis de ce TP sont :

- Comprendre les différents types de verrous et leurs implications
- Comprendre les différents arguments et flags possible à passer à `fcntl()` (file control)

Table of Contents

TADs & leurs relations: Décomposition modulaire

5 modules (+ `main.c`) ont été créés pour la réalisation du TP :

1. Le fichier `main.c` qui contient la fonction main qui permet de lancer le programme.
2. Le module `inp.c`

   Définit la struct `struct Inp` qui stock et gère l'input entré par l'utilisateur, pour ensuite l'utiliser dans `lock.c`.
3. Le module `lock.c`

   Utilise l'entrée de l'utilisateur pour ensuite remplir un `struct flock` avec les données nécessaires, et effectuer l'appel correspondant à `fcntl` pour lock/unlock les fichers.

TADs & leurs relations: Décomposition modulaire

4. Le module `optprsr.c` (repris du TP02, openssl)
   contient 3 fonctions, servant à gérer les options passées en paramètres.

5. Le module `files.c` (repris du TP03, ultra-cp)
   S'occupe de la gestion fichers, (existence, type, taille etc.), des path des fichiers (absolute path, concatenation...) et de la gestion des erreurs liées à ces opérations.

6. Le module `util.c` (repris du TP03, ultra-cp)
   Fonctions servant divers usages allant de la gestion d'erreurs à la gestion de chaînes de caractères, en passant par des wrappers qui incluent lesdites fonctions de gestions d'erreurs.

TADs & leurs relations : Structures de données utilisées

Une structure de donnée (opaque) `Inp` à été définie dont une explication brève a été donnée précédemment. Voici un apperçu de ses attributs :

```
//inp.c

struct Inp {
    /**
     * Contains concatenation of mode etc...
     * props[0] to props[2] contains each 'command' 'lock type' 'whence'.
     * Inp struct is opaque to enable "setting 'props' private".
     */
    char* props;
    long start, length;


};
```

```
//inp.h
/**
 *  Representation of a user input, i.e. wraps these:
 * -- 1. Command (1 char) 'g' (F_GETLK), 's' (F_SETLK), or 'w' (F_SETLKW)
 * -- 2. Lock type (1 char) 'r' (F_RDLCK), 'w' (F_WRLCK), or 'u' (F_UNLCK)
 * -- 3. start (int)
 * -- 4. length (int)
 * -- 5. whence (1 char) 's' (SEEK_SET , default), 'c' (SEEK_CUR), or 'e' (SEEK_END)
 *
 * Use the getters inp_* (inp_cmd...) to access to the command, lock type... (i.e. attributes of this struct.)
 */
typedef struct Inp Inp;
```

TADs & leurs relations

Avoir stocké toutes les options dans une seule chaîne de caractère et non pas plusieurs variables permet de l'initialiser comme suit :

```
int parseInput(char* cmd, char* ltype, long* start, long* length,
    char* whence) {
    /* ... */
    int an = sscanf(buf, "%s␣%s␣%ld␣%ld␣%s", cmd, ltype, start,
        length, whence); // argument number
    switch (an) { /* ... */ }
}
Inp* inp = malloc(sizeof(Inp));
inp->props = calloc(strArgNb + 1, sizeof(char)); // strArgNb = 3
//...
parseInput(&inp->props[0], &inp->props[1], &inp->start,
    &inp->length, &inp->props[2]);
// ...
```

(Où `parseInput()` fait principalement un `fscanf` avec les arguments qui lui sont passés puis vérifie la validité de chacun et gère les éventuelles erreurs)

TADs & leurs relations: Décomposition fonctionnelle

Les principales fonctions utilisés sont simplements des fonction de parsing d'input, manipulations de struct, wrapper simplifiant l'utilisation de fonctions de librairie C + gestion d'erreur...
Le "réel" vérouillage de fichier... est réalisé par des appels `fcntl()`, du type `fcntln(fd, fl_cmd, &fl)`
où `fl` est une `struct flock` qui a été converti directement depuis une `struct Inp` et `fl_cmd` est la commande entrée par l'utilisateur stockée dans la `struct Inp`.

En effet, une très grande partie du code est de la gestion d'erreur et du parsing d'argument en entrée, (voir fonction `lock()` du module `lock`, 1 ligne d'appel à `fcntl` et $\approx$ 65 lignes d'analyse de code d'erreur retourné, en fonctions de tel ou tel command (`F_GETLK`, `F_SETLK`...)
La partie qui "change" vraiment (outre la théorie sur les locks) est la mini structure de REPL / Shell que la CLI fournie par notre programme doit avoir.

TADs & leurs relations: Décomposition fonctionnelle

En résumé, les mécanismes de gestions d'erreurs et de parsing d'arguments étant un thème déjà longuement abordé dans les précédents et suivants TPs, ils ne seront pas plus détaillés ici.

Table of Contents

Tests realisés pour valider le fonctionnement du TP

Les tests réalisés ont simplement été d'ouvrire 2 instances du programme dans 2 sessions de shells différents puis de placer des locks à différents à droit endroit avec le premier et de tester s'ils existent et sont bien reconnus avec le 2e. i.e. :

```
noahl@noah-munz-hp-spectre in ~/Documents/ba3/os/TP04 on master ✓ (origin/master)
$ (19:31:52) $ ./main res/test.txt

Operating on file: "/home/noahl/Documents/ba3/os/TP04/res/test.txt"
Enter ? for help or q to exit.
PID=5359> ?
[PID=5359]
        Format:  cmd   l_type  start   length   [whence]
        'cmd'       ---     'g' (F_GETLK), 's' (F_SETLK), or 'w' (F_SETLKW)
        'l_type'    ---     'r' (F_RDLCK), 'w' (F_WRLCK), or 'u' (F_UNLCK)
        'start'     ---     lock starting offset
        'length'    ---     number of bytes to lock
        'whence'    ---     's' (SEEK_SET, default), 'c' (SEEK_CUR), or 'e' (SEEK_END)
--------

Enter ? for help or q to exit.
PID=5359> s w 0 10
[PID=5359] Successfully set lock
--------

Enter ? for help or q to exit.
PID=5359>
```

```
noahl@noah-munz-hp-spectre in ~/Documents/ba3/os/TP04 on master ✓ (origin/master)
$ (19:32:32) $ ./main res/test.txt

Operating on file: "/home/noahl/Documents/ba3/os/TP04/res/test.txt"
Enter ? for help or q to exit.
PID=5428> g w 0 5
[PID=5428] w lock present (held by 5359)
--------

Enter ? for help or q to exit.
PID=5428> g w 0 10
[PID=5428] w lock present (held by 5359)
--------

Enter ? for help or q to exit.
PID=5428> s r 0 5
[PID=5428] Resource temporarily unavailable: Could not add r lock at 0:5 (whence = s)
Denied by r lock on 0:5 (held by PID 5428)
--------

Enter ? for help or q to exit.
PID=5428>
```

Figure – Test de l'implémentation du TP04.

Table of Contents

Réponses aux questions (de l'énoncé du TP)

1. *Que se passera-t-il si nous déverrouillons un fichier (ou une partie du fichier) qui n'est pas verrouillé ?*
   - On reçoit erreur `EINVAL` et `strerror(errno)` donne "*Invalid Argument*" (Source : `man 2 flock`)

2. *Que se passera-t-il si nous mettons un nouveau verrou sur une section déjà verrouillée ? Le type de verrou changera t-il le résultat ? Expliquer dans la situation avec le même processus et avec 2 processus différents.*
   - Lorsqu'on a 2 processus différents, Le type de verrou changera le résultat : si on a déjà un read lock sur un fichier, on peut quand même mettre des read locks dessus. Par contre, si un fichier a un write lock, on ne pourra mettre ni de read lock, ni de write lock supplémentaires.
     Pareil si on essai de mettre un write lock sur une portion de fichier protégé par un read lock. ont obtient l'erreur `Resource temporarily unavailable`. *(Voir screenshot slide **12**)*
     Par contre si on est dans le cas d'un seul processus, il n'y a pas ce problème. En effet, il ne peut s'auto-empêcher d'accéder / vérouiller le contenu qu'il a lui même protégé.

Table of Contents

Réponses aux questions (générales)

Code : Lien GitHub du code
Rapport : Lien GitHub du rapport