

NMV - NI436

TP 1 – Les bases de la programmation noyau

Julien Sopena

Octobre 2015

Exercice 1 : Les structures

Dans ce premier exercice nous allons implémenter l'ensemble des mécanismes liés à la numérotation des commits. Ainsi chaque version sera associée à trois entiers :

major : un entier sur 2 octets qui est incrémenté à chaque changement majeur ;

minor : un entier sur 4 octets qui est remis à zéro à chaque changement majeur puis incrémenté à chaque modification mineure. Par convention, et comme dans beaucoup de projets, les numéros mineurs paires correspondront aux versions *stables* tandis que les impaires correspondront aux versions *unstable*.

flags : un ensemble de drapeaux stockés sur 1 octet.

Ainsi, le fichier `version.h` définit la structure suivante :

```
struct version {  
    unsigned short major;  
    unsigned long minor;  
    char flags;  
};
```

Question 1

Pour commencer faites un `makefile` qui compile `testVersion.c` et exécutez le programme obtenu. Que remarquez vous ?

Question 2

A l'aide d'un débogueur expliquez les raisons de cet affichage.

Question 3

Implémentez une nouvelle version du test que vous appellerez `isUnstableBis`.

Question 4

Modifiez la fonction `displayVersion` pour qu'elle prenne en argument la fonction de test à utiliser.

Question 5

Quel est l’empreinte mémoire d’une structure `version`? Cet encombrement est-il optimum?

Question 6

Modifiez la structure `version` pour optimiser ce code.

Exercice 2 : Calcul d’offset

Nous allons maintenant définir la structure associée à un `commit`. Cette dernière utilise la numérotation de version de l’exercice précédent en incluant une structure `version`. Elle y associe :

id : un entier non signé sur 8 octets qui sera unique à chaque *commit*;

comment : un pointeur vers une chaîne de caractères contenant un commentaire du *commit*.

On se basera ainsi sur la structure suivante :

```
struct commit {
    unsigned long id;
    struct version version;
    char *comment;
};
```

Question 1

Dans un premier temps, implémentez un petit test qui alloue et initialise une structure `commit` puis affiche les adresses des différents champs.

Question 2

On veut maintenant développer une fonction qui permet de retrouver l’adresse de la structure `commit` contenant une structure `version`. Cette nouvelle fonction doit être totalement indépendante de l’ordre et du nombre des champs de la structure `commit` et suivre le prototype suivant :

```
struct commit *commitOf(struct version *version);
```

En vous aidant de la méthode vue en cours, implémentez cette fonction et comparez son résultat aux adresses obtenues dans la question précédente.

Exercice 3 : Implémentation artisanale d'une liste doublement chaînée

Le maintien d'un historique nécessite, non seulement de pouvoir enregistrer des commits, mais aussi de retenir l'ordre de ces derniers. Si la numérotation peut être suffisante l'utilisation d'une structure de données adaptée améliorera considérablement les performances de notre système.

Nous allons donc modifier la structure `commit` pour réaliser une liste ordonnée doublement chaînée, puis implémenter un ensemble de fonctionnalités liées à l'ajout et au retrait de commit.

Question 1

Commencez par récupérer le fichier `commit.h`, `commit.c` et `testCommit.c`, puis complétez votre `Makefile` pour obtenir un exécutable de test.

Question 2

Dans un premier temps implémentez la fonction qui alloue et initialise un commit.

Question 3

La structure construite précédemment n'est pas encore insérée dans la liste doublement chaînée. Pour ce faire, vous allez développer les fonctions `add_minor_commit` et `add_major_commit`. Vous vous appuyerez sur la fonction `static insert_commit` qui s'occupera de l'insertion des commit construit par ces deux fonctions.

Question 4

Il est temps maintenant d'utiliser la liste doublement chaînée pour parcourir l'historique des commits. Implémentez la fonction d'affichage `display_history` de façon à obtenir l'affichage suivant :

```
0: 0-0 (stable)      'First !'
0: 0-0 (stable)      'First !'
1: 0-1 (unstable)    'Work 1'
2: 0-2 (stable)      'Work 2'
3: 0-3 (unstable)    'Work 3'
4: 0-4 (stable)      'Work 4'

0: 0-0 (stable)      'First !'
1: 0-1 (unstable)    'Work 1'
2: 0-2 (stable)      'Work 2'
3: 0-3 (unstable)    'Work 3'
4: 0-4 (stable)      'Work 4'

0: 0-0 (stable)      'First !'
1: 0-1 (unstable)    'Work 1'
2: 0-2 (stable)      'Work 2'
3: 0-3 (unstable)    'Work 3'
4: 0-4 (stable)      'Work 4'
5: 1-0 (stable)      'Realse 1'
6: 1-1 (unstable)    'Work 1'
7: 1-2 (stable)      'Work 2'
8: 2-0 (stable)      'Realse 2'
9: 2-1 (unstable)    'Work 1'

0: 0-0 (stable)      'First !'
1: 0-1 (unstable)    'Work 1'
2: 0-2 (stable)      'Work 2'
3: 0-3 (unstable)    'Work 3'
4: 0-4 (stable)      'Work 4'
10: 0-5 (unstable)    'Oversight !!!'
5: 1-0 (stable)      'Realse 1'
6: 1-1 (unstable)    'Work 1'
7: 1-2 (stable)      'Work 2'
8: 2-0 (stable)      'Realse 2'
9: 2-1 (unstable)    'Work 1'

7: 1-2 (stable)      'Work 2'
1-7 : Not here !!!
4-2 : Not here !!!
```

Question 5



Pour finir implémentez une fonction qui parcourt l'historique à la recherche d'un commit correspondant au numéro de version passé en paramètre. Cette fonction `infos` affichera le contenu du commit si elle le trouve ou "Not here !!!" dans le cas contraire.

Exercice 4 : De l'intérêt des `list_head`

Pour optimiser la recherche d'un commit, nous pourrions commencer par parcourir les commits majeurs pour limiter le parcours des versions mineures à celles de la version majeure désirée.

Question 1

Pour réaliser cette optimisation nous allons avoir besoin d'une deuxième liste doublement chaînée limitée aux commits majeurs. Peut-on envisager une factorisation du code d'insertion et de suppression tel que nous l'avons conçu ?

Question 2

Plutôt que de réinventer la roue, nous allons utiliser l'implémentation des listes du noyau Linux. Pour commencer, chargez le fichier `list.h` et parcourez le. Ce fichier est en réalité une version légèrement modifiée du header original de façon à limiter les dépendances.

Question 3

Après avoir remplacé vos champs `next/prev` par un champ `list` de type `list_head` dans `commit.h`, adaptez votre implémentation des fonctions du squelette. Appréciez la simplicité de ce nouveau code.

Exercice 5 : Double liste et raccourcis

Nous allons maintenant pouvoir ajouter une deuxième liste dédiée aux versions majeures. Cette nouvelle implémentation sera basée sur deux nouveaux champs :

`major_list` de type `list_head` il servira à relier les commits majeurs ;

`major_parent` un pointeur qui permet depuis chaque version mineure de retrouver la version majeure correspondante.

Question 1

Après avoir ajouté ces deux champs dans la structure `commit`, corrigez les fonctions d'insertion et testez les. Pour l'instant vous laisserez de côté la fonction de suppression `del_commit`.

Question 2

Donnez une nouvelle implémentation de la fonction `infos` qui utilise cette nouvelle liste pour améliorer les performances de la recherche.

Exercice 6 : Audit mémoire et destruction d'une liste

Les fuites mémoire sont choses fréquentes en C et sortir un commit de la liste ne suffit pas pour libérer les ressources. Si elles sont gênantes dans un programme "classique", elles sont critiques dans le noyau.

Question 1

Après avoir recompilé votre code avec l'option -g, auditez le avec le programme valgrind.

```
valgrind -leak-check=full ./testCommit
```

Question 2

Si besoin modifiez votre code pour corriger toutes les fuites mémoire. Vous pourrez implémenter une fonction `void freeHistory(struct commit *from)` qui libérera la mémoire occupée par l'ensemble des éléments d'une liste de commit.