PNL - MU4IN402

TP 01 – Un peu de C avant la programmation noyau

Redha Gouicem, Maxime Lorrillere et Julien Sopena

janvier 2023

Ce premier TP, à réaliser à votre rythme en autonomie, sera l'occasion pour vous de faire un point sur vos connaissances du langage C autour d'exercices centrés sur l'implémentation dun gestionnaire de version. En se basant sur une liste circulaire doublement chaînée, ces exercices vous amèneront à manipuler des structures C et leurs alignements, à gérer les allocations mémoire, à manipuler des pointeurs et à résoudre des problèmes de typage.

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 8 octets qui est remis a zéro à chaque changement majeur puis incrémenté à chaque modification mineure. Par convention, et comme dans beaucoup de projets, les numéros mineurs pairs correspondront aux versions *stables* tandis que les impairs correspondront aux versions *unstables*.

`flags` : un ensemble de drapeaux stockés sur 1 octet.

Ainsi, le fichier version.h défini la structure suivante :

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

Question 1

Pour commencer récupérez les sources présentes dans `\srcDirPath {}`/TP/TP-01, puis compilez et exécutez le programme `testVersion`. Est-ce que le résultat du test de stabilité (sur la parité du numéro mineur) est correct ?

Question 2

Pourtant le calcul pour trouver le premier bit du numéro mineur est syntaxiquement valide et semble correct. À l'aide d'un débogueur, expliquez les raisons de cet affichage.

Question 3

En utilisant le nom des champs, implémentez une nouvelle version du test que vous appellerez `is_unstable_bis`.

Question 4

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

Question 5

Quelle est l'empreinte mémoire d'une structure `struct version` ? Cet encombrement est-il optimum ?

Question 6

Modifiez la structure `struct 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 `struct 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;
        struct commit *next;
        struct commit *prev;
};
```

Question 1

Dans un premier temps, implémentez un petit programme `testOffset.c` qui alloue et initialise une structure `struct 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 `struct 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 *commit_of(struct version *version);
```

Comme nous l'avons vu précédemment le compilateur peut introduire des octets de padding pour réaligner en mémoire les champs de la structure. Cependant ce placement est déterministe et identique sur l'ensemble des instances de la structure.

Pour commencer, affichez le décalage (offset) entre le début de la structure `struct commit` et le champ `version`.

Question 3

À partir de ce décalage, vous pouvez maintenant implémenter la fonction `commit_of` et comparez le résultat obtenu aux adresses obtenues dans la question précédente.

Exercice 3 : Implémentation artisanale d'une liste circulaire 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 `struct commit` pour réaliser une liste ordonnée circulaire doublement chaînée, puis implémenter un ensemble de fonctionnalités liées à l'ajout et au retrait de commit.

Pour simplifier la gestion de cette liste on introduit aussi une nouvelle structure `struct history` qui contiendra un nom, une liste de `commit` ainsi que la taille de cette liste.

```
struct history {
        unsigned long commit_count;
        char *name;
        struct commit *commit_list;
};
```

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

Pour simplifier la suite de ce TP on considérera qu'une liste vide est un commit "fantôme" (qui ne correspond à aucun commit réel) et qui pointe sur lui même. Ce commit demeurera par la suite dans la liste et constituera son point de départ.

Dans un premier temps, implémentez les fonctions `new_commit` et `new_history` qui allouent et initialisent respectivement un commit et un historique. Puis implémentez aussi la fonction `last_commit` qui retourne le dernier commit d'un historique.

Question 3

On peut maintenant créer un historique vide et accéder au dernier élément de celui-ci, reste donc à pouvoir insérer des commits dans la liste circulaire doublement chaînée. Pour ce faire, vous allez développer les fonctions `add_minor_commit` et `add_major_commit`. Vous vous appuierez sur la fonction *static* `insert_commit` qui s'occupera de l'insertion des commits construits par ces deux fonctions.

Question 4

Il est temps maintenant d'utiliser la liste circulaire 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)    'DO NOT PRINT ME !!!'

Historique de 'Tout une histoire' :

Historique de 'Tout une histoire' :
 1:  0.1 (unstable)  'Work 1'
 2:  0.2 (stable)    'Work 2'
 3:  0.3 (unstable)  'Work 3'
 4:  0.4 (stable)    'Work 4'

Historique de 'Tout une histoire' :
 1:  0.1 (unstable)  'Work 1'
 2:  0.2 (stable)    'Work 2'
 3:  0.3 (unstable)  'Work 3'
 4:  0.4 (stable)    'Work 4'

Historique de 'Tout une histoire' :
 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'

Historique de 'Tout une histoire' :
 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'
```

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.

```
Recherche du commit 1.2 :    7:  1.2 (stable)    'Work 2'

Recherche du commit 1.7 :   Not here !!!

Recherche du commit 4.2 :   Not here !!!
```