PNL - MU4IN402

TP 02 – Les bases de la programmation noyau

Redha Gouicem, Maxime Lorrillere et Julien Sopena

février 2021

Ce deuxième TP s'inscrit dans la suite directe du premier et a pour but de vous familiariser avec les paradigmes objets de la programmation noyau. Ainsi, en reprenant l'exemple d'un gestionnaire de version, il s'attachera, entre autres, à mettre en évidence l'intérêt des structures de données génériques du noyau, et notamment celle des listes chaînées.

Exercice 1 : 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 ? Ce travail pourra-t-il être facilement réutilisé pour d'autres structures ?

Question 2

Plutôt que de réinventer la roue, nous allons utiliser l'implémentation des listes du noyau Linux. Pour commencer, récupérez le fichier `list.h` dans `/usr/data/sopena/pnl/TP/TP-02` et parcourez-le.

Vous attacherez une attention particulière aux fonctions `INIT_LIST_HEAD`, `list_add` et `list_del` et aux macros `list_for_each`, `list_for_each_entry`.

Par curiosité vous pourrez aussi comparer les macros `container_of` et `offsetof` à votre implémentation de la fonction `commitOf`.

Remarque : Ce fichier list.h est en réalité une version légèrement modifiée du header original de façon à limiter les dépendances.

Question 3

Après avoir remplacé les champs `next` et `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 2 : 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 `struct 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 3 : 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 mémoires. 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.

Exercice 4 : Pointeur de fonctions et interface

On veut maintenant modifier l'affichage en fonction du type de commit, i.e. afficher de façon différente les commits correspondant à une version majeure. L'affichage pourrait ainsi être le suivant :

```
0:  ### version 0 : '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:  ### version 1 : 'Realse 1' ####
6:  1-1 (unstable)   'Work 1'
7:  1-2 (stable)     'Work 2'
8:  ### version 2 : 'Realse 2' ####
9:  2-1 (unstable)   'Work 1'
```

Question 1

Une première approche pourrait être d'ajouter un test dans la fonction d'affichage `displayCommit` et ainsi adapter l'affichage en fonction du commit à afficher. Quels problèmes de conception pose cette approche ?

Question 2

Comme dans le noyau nous allons plutôt associer à chaque commit une version de la fonction d'affichage. Il suffit en effet d'ajouter un champ `display` pointant vers la fonction choisie, ce champ étant ensuite utilisé par la fonction d'affichage de l'historique.

Après avoir implémenté une nouvelle fonction `display_major_commit`, vous adapterez votre code pour assurer l'affichage demandé.

Question 3

On veut maintenant utiliser la même technique pour adapter la fonction d'éviction du commit : un commit mineur sera retiré simplement de la liste, tandis que la suppression d'un commit majeur s'accompagnera de la suppression de l'ensemble des commits mineurs associés.

Implémentez un tel comportement en vous basant sur un champ `extract` que vous ajouterez à la structure `struct commit` et deux nouvelles fonctions `extract_minor` et `extract_major`.

Question 4

Si l'utilisation de pointeurs de fonction dans la structure assure une certaine flexibilité et accroît la maintenabilité, elle devient particulièrement lourde lorsque le nombre de fonctionnalités croît.

L'idée, à la manière des interfaces de la programmation-objet, est d'introduire une structure `struct commit_ops` qui définit un ensemble de fonctionnalités. Chaque instance de cette structure fixe un ensemble cohérent d'opérations adaptées à chaque cas (ici majeur et mineur).

Modifiez votre code pour utiliser l'interface suivante :

```
struct commit_ops {
  void (*display)(struct commit *);
  struct commit *(*extract) (struct commit *);
};
```

Exercice 5 : Traitement des erreurs dans le noyau

On veut maintenant améliorer les commentaires en ajoutant au texte, un titre et le nom. On va donc remplacer la chaîne de caractères du champ `comment` par la structure suivante :

```
struct comment {
  int title_size;
  char *title;
  int author_size;
  char *author;
  int text_size;
  char *text;
};
```

Question 1

Récupérez les fichiers `comment.h`, `comment.c` et `testComment.c` dans `/usr/data/sopena/pnl/TP/TP-02`, puis lancez une analyse de la consommation mémoire de ce test. Quels problèmes pose cette implémentation ?

Question 2

Modifiez le code de la fonction `new_comment` pour la rendre résiliente. Vos corrections doivent permettre au test de fonctionner sans modifications.