Cours "Informatique Embarquée"

François Armand

Exercice N°2, 16 Octobre 2015
À rendre avant le 29 Octobre 23H59
(Limite impérative) sur Didel
2 séances de TP.
Hash Lists

1. Consignes

Les consignes habituelles s'appliquent :

- Estimation du temps, temps réel passé
- Relecture par un autre groupe
- Structure des sources, Makefile...

1. Hash Lists

1.1. Introduction, motivation

Les systèmes d'exploitation utilisent des outils génériques pour remplir leurs tâches. Un des mécanismes les plus courants est de gérer des structures de données (descripteurs de processus, de fichiers, de régions mémoires....). Cela consiste très souvent en opérations d'allocation, de libération et en recherche d'une structure donnée à partir d'un identificateur.

On a aussi parfois à placer une structure dans un ensemble existant. Par exemple, lorsqu'un processus devient éligible, il faut non seulement le "marquer" comme tel, mais aussi l'insérer dans l'ensemble des processus que l'ordonnanceur devra examiner pour trouver celui qui se verra attribuer le processeur.

Ces listes sont aussi utilisées dans le monde applicatif.

Le problème des listes chaînées est que leur taille croît linéairement avec le nombre d'objets qui se trouvent dans la liste. Le temps de parcours (de traitement) d'une liste devient alors proportionnel au nombre d'objets de cette liste. Un moyen de minimiser ce problème est de faire appel à des hash listes (liste de hachage). C'est un outil général dans l'univers du noyau Linux.

Il existe d'autres outils : arbres b-tree, arbres rouges-noirs...

Dans le monde Java, on utilise aussi des hashmap, qui permettent de stocker des éléments en les associant à une clé ou un identifiant, et de les retrouver.

1.2. Listes chaînées

Vous écrirez (en langage C) une bibliothèque fournissant des services de listes de gestion de **listes circulaires doublement chaînées**.

```
/ * init an element */
void INIT_LIST_HEAD (struct list_head *head);
/* add "node" element after "head"
void list_add (struct list_head *node, struct list_head *head);
/* remove element "node" from a list */
void list_del (struct list_head *node);
/* iterate over a list starting at head */
list_for_each_entry(cur, head, member)
```

L'itérateur sera générique et devra permettre de parcourir des listes d'objets incluant des éléments de type `struct list_head` quelle que soit la position de l'élément `struct list_head` dans l'objet. Un objet doit aussi pouvoir appartenir à plusieurs listes simultanément, et l'itérateur doit pouvoir être utilisé pour parcourir l'une ou l'autre des listes.

```
struct my_object {
     ....
     struct list_head my_object_listA;
     ....
     struct list_head my_object_listB;
     ....
}
```

La « complexité » réside essentiellement dans l'itérateur. Pour vous simplifier le travail cet itérateur est fourni :

```
#define container_of(ptr, type, member) ({                              \
     const typeof( ((type *)0)->member ) *__mptr = (ptr);               \
     (type *)( (char *)__mptr - offsetof(type, member) );})

#define list_for_each_entry(cur, head, member)                          \
     for (cur = container_of((head)->next, typeof(*cur), member);       \
          &cur->member !=(head);                                        \
          cur = container_of(cur->member.next, typeof(*cur), member))
```

Vous chercherez donc à comprendre ce que fait cette macro ainsi que les « fonctions » prédéfinies `typeof` et `offsetof`.

1.3. Table de « hash »

Sur la base de ces listes chaînées, vous créerez (en langage C) un service de table de hash permettant de manipuler deux types de structures de données:

- Structure A : vous définirez une structure d'objets avec un identifiant qui soit un entier strictement positif. Exemple : une liste d'i-nodes (descripteurs de fichiers) chaque i-node étant identifié par son numéro.
- Structure B : vous définirez une structure d'objets avec un identifiant qui soit un entier toujours multiple de 4096. Exemple : une liste de descripteurs de pages mémoire, chaque page étant identifiée par son adresse de début (la taille d'une page est souvent de 4096 octets).

Il n'est pas nécessaire de définir un objet contenant à la fois une structure A et une structure B. Vous fournirez :

- une fonction d'initialisation de la liste de hachage,
- une fonction d'insertion,
- une fonction de retrait,
- et une fonction permettant de retrouver un objet à partir de son identifiant

Et évidemment au-delà de ces deux « bibliothèques » (listes chaînées et tables de hash), un petit programme main exerçant ces fonctions.

La rédaction de cette deuxième partie est volontairement peu précise. Contrairement à la première partie, il n'y a pas d'API qui soit imposée. Libre à vous de proposer ce qui vous semble le mieux.