Implémentation d'une table de hachage parallèle

Raphaël Prévost | Wargan Solutions

19 juillet 2010

Résumé

La table de hachage est une structure de données généralement utilisée pour implémenter tableaux associatifs, caches, ou plus généralement associer une clé à une valeur.

Dans le cadre de notre projet, nous avons choisi de réaliser un cache de données reposant sur cette structure, particulièrement intéressante dans notre cas pour ses temps d'accès relativement constants par rapport au nombre d'entrées dans la table.

Cependant, la majorité des implémentations libres ne sont pas optimisées pour une utilisation parallèle. Nous avons donc décidé de programmer cette structure nous-mêmes, en nous basant sur les travaux publiés sur le sujet. Nous décrirons dans cet article le cheminement que nous avons suivi afin de mener à bien cette implémentation.

Première partie

Dans l'intimité d'une table de hachage

La table de hachage n'est pas une structure complexe : c'est en réalité un vulgaire tableau. Tout l'art et la difficulté de son implémentation se résume à deux problèmes principaux : le premier consiste à trouver une fonction injective (fonction de hachage) associant *efficacement* à toute clé un emplacement quelconque dans le tableau.

Le second est plus subtil et procède directement du premier : notre tableau ne disposant que d'un nombre fini d'emplacements, il est nécessaire de gérer le cas où la fonction de hachage attribue le même emplacement à plusieurs clés différentes (collision). C'est le problème de la gestion de ces collisions qui est le plus ardu, et il existe de nombreuses façons de le traiter. Nous allons les passer en revue concisément dans les paragraphes suivants.

1 Deux écoles principales : «*chaining*» contre «*open addressing*»

Globalement, on peut diviser les algorithmes de résolution de collisions en deux familles principales, le «*chaining*» favorisant plutôt la montée en charge, et l'«*open addressing*» privilégiant une consommation mémoire réduite.

1.1 «*Chaining*»

Cet algorithme consiste à régler effectivement la collision en stockant les deux paires clé-valeur conflictuelles dans le même emplacement. Ceci est rendu possible en définissant chaque élément du tableau comme le point d'entrée d'une liste chaînée.

Cette méthode présente les avantages d'être simple à implémenter et de conserver de bonnes performances même lorsque le tableau se remplit ; en effet, le temps requis pour retrouver une clé augmente linéairement avec le nombre de collisions.

En revanche, elle a tous les inconvénients des listes chaînées : dommageable pour le cache du processeur, suppression des éléments avec une complexité linéaire, et consommation mémoire accrue par la necessité de stocker les pointeurs.

De nombreuses implémentations se basent sur cet algorithme, car il n'impose pas de redimensionner la table ; en effet, il est toujours possible d'y insérer une nouvelle clé au prix d'une certaine perte de performance.

1.2 «*Open addressing*»

L'alternative, «*open addressing*», enregistre la clé conflictuelle directement dans un emplacement disponible arbitraire dans le tableau.

Les coordonnées de cet autre emplacement sont calculées soit à partir de l'emplacement où la clé aurait dû être insérée, soit en utilisant une autre fonction de hachage.

Quand le tableau se remplit, il peut être nécessaire d'effectuer ces calculs un certain nombre de fois avant de trouver un emplacement libre.

Cette méthode est plus efficace au niveau de la gestion de la mémoire, puisqu'il n'est pas nécessaire d'en allouer et que les données sont disposées idéalement pour le cache processeur. Les performances se dégradent toutefois de manière très marquée dès que la table atteint un certain seuil de charge.

2 Les hybrides

Il existe d'autres algorithmes empruntant certains principes propres aux deux méthodes présentées précédemment, en les combinant pour compenser leurs faiblesses respectives. Nous nous sommes particulièrement intéressés à ces derniers, dans le but de réaliser une implémentation à la fois rapide et efficace.

2.1 «*Cuckoo hashing*»

Cette méthode, dérivée de l'«*open addressing*» est exposée dans un article de recherche éponyme de Rasmus Pagh et Flemming Rodler publié en 2001.

Le principe consiste à utiliser deux fonctions de hachage, afin de disposer de deux emplacements possibles pour chaque clé. Lors de l'insertion d'une clé, si les deux emplacements sont occupés, l'algorithme va purement et simplement «éjecter» la clé déjà en place, comme un bébé coucou éjecte les œufs du nid qu'il parasite, pour la remplacer par les nouvelles données.

La clé éjectée est ensuite réinsérée dans le tableau suivant le même algorithme. On voit cependant que ce principe boucle infiniment quand le tableau devient saturé. Il est possible de retarder cet instant en utilisant un plus grand nombre de fonctions de hachage, augmentant ainsi le nombre d'emplacements potentiels.

Cet algorithme a pour principal intérêt d'offrir des temps d'insertion et de lecture constants, ainsi qu'une bonne utilisation de la mémoire (dépendante toutefois du nombre de fonctions de hachage utilisées).

2.2 «*Cuckoo hashing*» et panier

Dans un article d'Adam Kirsh, Michael Mitzenmacher et Udi Wieder publié en 2008, l'idée est avancée de pallier les défauts du «*cuckoo hashing*» en adjoignant un «panier» à la table de hachage, de façon à recueillir les clés «tombées du nid».

Ce «panier» se présente sous la forme d'un tableau secondaire de taille fixe, dans lequel sont insérées temporairement les clés pour lesquelles un emplacement n'a pu être trouvé. Ce procédé permet de retarder efficacement l'agrandissement de la table, sans utiliser plus de fonctions de hachage.

Finalement, le «*cuckoo hashing*» permet d'améliorer grandement les performances du modèle «*open addressing*» en offrant des performances constantes en dépit du nombre d'entrées dans la table.

Son inconvénient majeur, l'impossibilité d'insérer de nouvelles clés au delà d'une certaine charge, peut être contourné en augmentant le nombre de fonctions de hachage et par l'adjonction d'un «panier» pour recueillir les clés problèmatiques.

Examinons maintenant les solutions permettant d'améliorer les algorithmes basés sur le «*chaining*».

2.3 «*Chaining*» interne et aggrégation

Les améliorations apportées aux tables utilisant le «*chaining*» concernent d'avantage la consommation et la disposition des données en mémoire plutôt que les performances.

Une optimisation courante consiste à inclure le premier maillon de la liste chaînée directement dans l'emplacement réservé du tableau, afin de permettre une meilleure utilisation du cache dans le cas où il n'y a pas de collisions.

Il est également possible de remplacer la liste par un tableau dynamique, afin de préserver le cache du processeur et d'économiser la mémoire qui autrement serait utilisée pour stocker les pointeurs de la liste.

D'autres algorithmes vont plus loin et n'utilisent pas de réelles listes : les éléments sont directement placés dans le tableau lui-même, ce qui est appelé «*chaining*» interne. Cette méthode se rapproche de l'«*open addressing*», mais sans présenter l'inconvénient de calculer la position des emplacements alternatifs.

Deuxième partie
Notre implémentation

On a vu qu'en adoptant une approche hybride entre «*chaining*» et «*open addressing*», il est possible de réaliser des algorithmes avec des performances et une occupation mémoire plus équilibrées. Nous nous sommes donc efforcés d'effectuer une synthèse des différents travaux que nous avons évoqué précédemment dans le but de trouver un compromis satisfaisant entre performances brutes et utilisation de la mémoire.

Le «*cuckoo hashing*» nous a particulièrement interressés du fait de ses temps d'accès constants ; nous l'avons donc utilisé comme base tout en cherchant à pallier le problème des insertions impossibles dont cet algorithme est affligé quand la table devient trop pleine.

1 Algorithme de résolution des collisions

La solution que nous avons retenue est, sans surprise, un «*cuckoo hashing*» adjoint d'un «panier». Le «*cuckoo hashing*» permet une bonne utilisation de l'espace de stockage de la table, tout en permettant un recyclage automatique des emplacements occupés par des clés supprimées. Avec une fonction de hachage de qualité raisonnable, le «panier» reste en pratique de taille suffisamment réduite pour n'avoir qu'un impact négligeable sur les performances, tout en permettant une bien meilleure montée en charge.

Le «panier» statique exposé dans l'article que nous avons mentionné a cependant l'inconvénient de ne pas éliminer totalement le cas où une clé ne peut être enregistrée, surtout s'il est de petite taille. Cela oblige parfois à effectuer un agrandissement prématuré de la table, qui reste une opération coûteuse. De grande taille, le panier rend ces redimensionnements forcés

inexistants au prix d'un certain gaspillage mémoire. Le panier dynamique nous est donc apparu la solution la plus performante.

Nous avons également cherché à optimiser l'accès aux données du «panier». Afin d'éviter de devoir faire une recherche linéaire pour retrouver un élement placé dans le «panier», nous avons eu recours à un «*chaining*» interne entre l'emplacement où la clé aurait dû être insérée, et sa position réelle dans le panier.

Notre première version du «panier» utilisait un tableau dynamique, mais nous avons déterminé expérimentalement qu'une liste chaînée apportait un gain de performances significatif à la fois lors de l'insertion et de la recherche grâce à la diminution du nombre d'allocations mémoire.

Ce changement a néanmoins un inconvénient : dans le cas pathologique où tous les emplacements possibles du tableau sont occupés et déjà liés à une place dans le «panier», la recherche linéaire plus lente ne peut être évitée. En pratique, cette situation est si rare que l'impact sur les performances est virtuellement nul.

2 Structure de la table de hachage

Notre première implémentation se présentait sous la forme d'un tableau de structures contenant :

- La taille codée sur 16 bits de la clé. Une taille nulle indique un emplacement supprimé.
- L'index lui aussi codé sur 16 bits d'un éventuel doublon dans le panier.
- Un pointeur sur un emplacement mémoire contenant la clé. Celle-ci est directement préfixée par un pointeur sur la valeur qui lui est associée.

Le panier était implémenté comme un tableau dynamique de ces mêmes structures, limité à 65535 élements au maximum. En pratique, cette limite ne devrait pas être atteinte (le panier ne comporte généralement pas plus d'une dizaine d'élements pour un million de couples clé-valeur).

Cette disposition mémoire permettait de tirer partie du cache processeur, et d'avoir une surcharge mémoire raisonnable de 32 bits par clé pour une occupation mémoire totale de 64 bits par emplacement (96 bits sur architecture 64 bits).

Après differents tests, nous avons modifié ce programme initial afin d'améliorer les performances.

La table de hachage a été remplacée par un simple tableau de pointeurs, réduisant ainsi la taille des emplacements. Les données ont été déplacées pour être stockées avec la clé dans un bloc mémoire dynamiquement alloué.

Le «panier» quant à lui a été changé en une liste chaînée, permettant ainsi une croissance illimitée. Le «*chaining*» fut réimplémenté en conséquence pour utiliser un pointeur au lieu d'un index.

Ces modifications ont amélioré les performances d'environ 20% en lecture et 10% en insertion, mais l'occupation mémoire excédentaire a augmenté de 32 bits par clé.

Afin de permettre une énumération aisée des éléments stockés dans notre table de hachage, nous avons ajouté un pointeur supplémentaire à la structure contenant les données. Ainsi, une liste chaînée de tous les éléments est construite au gré des insertions, permettant de les énumérer dans l'ordre suivant lequel ils ont été insérés.

Cela nous a permis d'optimiser du même coup le redimensionnement de la table, en ne parcourant que les éléments de la liste au lieu de la table entière. Grâce à cette modification, l'insertion est 20% plus rapide.

Troisième partie

Le défi du parallèlisme

Afin de rendre l'accès à cette table parallèlisable, nous lui avons ajouté un verrou de type lecteur/écrivain. Ce type de verrou autorise plusieurs processus à lire les données simultanément tant qu'aucun d'entre eux n'écrit dans la table.

Lorsqu'un processus écrit, tous les autres doivent attendre que l'insertion soit terminée avant de pouvoir accéder de nouveau à la table.

Cette solution est simple mais peu satisfaisante ; en effet, rien ne justifie de bloquer tous les processus lecteurs dans le cas où un processus écrivain modifie une clé à laquelle ils ne tentent pas d'accéder.

Nous avons donc cherché à améliorer la granularité de la synchronisation en modifiant la structure de la table.

1 Accès parallèles

Nous avons implémenté notre table de hachage dans le but de l'utiliser au sein d'un serveur, c'est à dire un environnement où toutes les ressources doivent être simultanément accessibles en parallèle par de nombreux processus.

Ce point est d'autant plus crucial que l'avènement des processeurs disposant de multiples cœurs permet des gains de performance très importants pour peu qu'on utilise des structures favorisant un parallèlisme d'échelle.

Il n'était donc pas possible de se limiter au schéma de verrouillage primitif évoqué dans la partie précédente, car sa mauvaise granularité empêche sans légitimité les accès concurrents à des éléments indépendants.

Nous avons choisi de modifier la structure originale de notre table de hachage pour la diviser en segments, permettant d'obtenir une granularité de verrouillage plus fine de la synchronisation.

Le niveau le plus précis imposait d'attribuer un verrou à chaque emplacement du tableau, ce qui nous a paru bien trop coûteux en occupation mémoire.

La segmentation permet d'augmenter le nombre d'accès concurrents à moindre coût et d'ouvrir d'autres perspectives intéressantes que nous allons détailler.

2 Une table de hachage segmentée

Plutôt que de rejeter notre implémentation initiale et de tout réécrire, nous avons choisi de redéfinir ce que nous appelions précédemment table de hachage, **segment**, et de construire ce que nous appellerons dans la suite de ce document **table de hachage** au dessus de la structure existante.

La nouvelle table de hachage est donc une collection de segments indépendants ; cela pose *a priori* un nouveau problème : comment décider dans quel segment chaque clé doit être insérée ?

Il existe de nombreuses réponses à cette question, comme les filtres de Bloom ou les tables de hachage à multiples niveaux, qui nous ont paru d'une complexité indésirable dans ce contexte où les performances brutes sont primordiales.

Nous avons choisi la solution, plus simple, de composer la table de hachage d'un nombre fixe de 256 segments, et de distribuer les clés dans ces derniers suivant leur somme CRC8.

Le CRC8 est un algorithme de somme de contrôle permettant d'associer à toute clé un entier codé sur 8 bits, que nous utilisons ici comme index de segment. Le CRC8 étant un calcul extrêment simple et rapide, cela permet de n'ajouter qu'un minimum de complexité à l'algorithme existant.

Il aurait été possible d'économiser de la mémoire en utilisant un nombre dynamique de segments, mais cette solution impose de réinsérer toutes les clés enregistrées à chaque ajout ou suppression d'un segment. En effet, changer le nombre des ces derniers modifie la distribution des clés.

Cette opération coûteuse verrouillerait de surcroît toute la table pendant son déroulement, allant à l'encontre de notre objectif de parallèlisme. Le coût mémoire de la solution retenue est plus important mais reste constant. Nous allons voir que cette solution améliore également les performances générales de l'implémentation sous-jacente.

3 La dissolution des collisions

L'utilisation de segments nous a permis de favoriser les accès concurrents aux données de la table de hachage, mais présente également une propriété intéressante.

En distribuant les clés suivant leur CRC8, cet algorithme diminue la probabilité de collision dans chaque segment, en allégeant leur charge respective et en opérant une première répartition pseudo-aléatoire.

Il en résulte des temps d'insertion et d'accès plus rapides du fait de la réduction du nombre de collisions, «diluées» par la présélection à laquelle sont soumises les clés.

Au final, l'implémentation complète présente des temps d'accès plus avantageux en utilisant plusieurs processus parallèles, ce qui correspond à l'objectif que nous nous étions fixé. Dans le cas où la table ne doit être utilisée que par un unique processus, il reste toutefois plus avantageux d'utiliser un simple segment comme table de hachage afin de réduire l'empreinte mémoire.

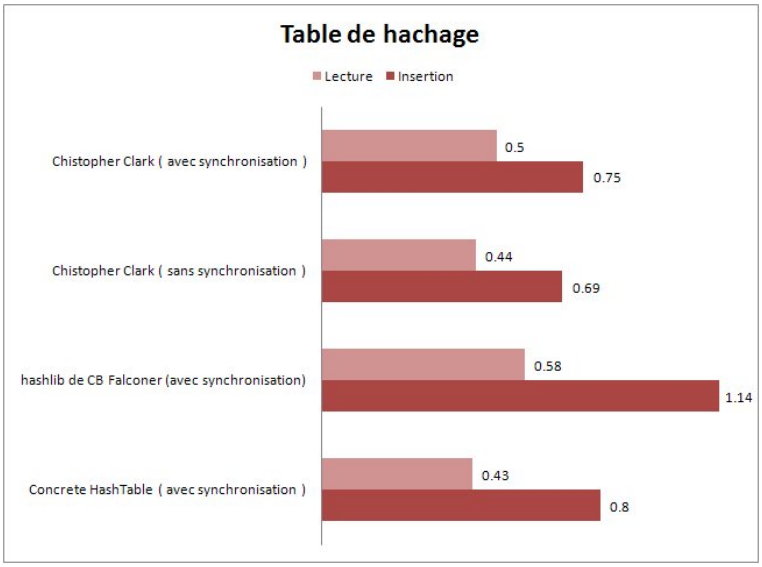
A Annexe : Quelques chiffres

Notre implémentation finale a des performances compétitives ; insérer 800 000 clés est effectué en 0.800 secondes, les relire en 0.430 secondes sur un Intel Core Duo 2GHz. Le remplacement de toutes les clés est un peu plus long, prenant 0.560 secondes. La suppression est réalisée en 0.470 secondes.

À titre de comparaison, la bibliothèque hashlib de C. B. Falconer modifiée afin d'être synchronisée, utilisant la même fonction de hachage et les même données de test réalise l'insertion des 800 000 élements en 1.140 secondes, la relecture en 0.500 secondes et le remplacement en 0.700 secondes.

Une autre implémentation de référence, celle de Christopher Clark, insère les 800 000 clés en 0.750 secondes et les retrouve en 0.500 secondes, avec synchronisation. Sans verrous, ses temps sont plus rapides : 0.690 secondes en insertion et 0.440 secondes en lecture.

Tous ces résultats ont été obtenus avec les mêmes options de compilation, la même fonction de hachage (lookup3 de Bob Jenkins) et le même échantillon de données.



B Annexe : Bibliographie

1. *Cuckoo Hashing*, Pagh, Rodler, 2001.
http ://citeseer.ist.psu.edu/pagh01cuckoo.html

2. *More Robust Hashing : Cuckoo Hashing with a Stash*, Kirsch, Mitzenmacher, Wieder, 2008.
http ://research.microsoft.com/users/uwieder/papers/stash-full.pdf

3. *Hash Functions and Block Ciphers*, Jenkins
http ://burtleburtle.net/bob/hash/

C Annexe : Implémentation

```c
/*******************************************************************************
 *  Concrete Server                                                            *
 *  Copyright (c) 2005-2010 Raphael Prevost                                    *
 *                                                                             *
 *  This library is free software; you can redistribute it and/or              *
 *  modify it under the terms of the GNU Lesser General Public                 *
 *  License as published by the Free Software Foundation; either               *
 *  version 2.1 of the License, or (at your option) any later version.         *
 *                                                                             *
 *  This library is distributed in the hope that it will be useful,            *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of             *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU           *
 *  Lesser General Public License for more details.                            *
 *                                                                             *
 *  You should have received a copy of the GNU Lesser General Public           *
 *  License along with this library; if not, write to the Free Software        *
 *  Foundation, Inc., 51 Franklin Street, 5th Floor, Boston, MA 02110-1301 USA *
 *                                                                             *
 *******************************************************************************/

#ifndef M_HASHTABLE_H

#define M_HASHTABLE_H

#ifdef _ENABLE_HASHTABLE

#include "m_core_def.h"

#ifdef FAST_TABLE
#define CACHE_HASHFNCOUNT          5      /* number of hash functions to use */
#define CACHE_CUCKOORETRY          8      /* retries if the bucket is full */
#define CACHE_GROWTHRATIO        1.6      /* threshold to grow the table */
#else
#define CACHE_HASHFNCOUNT          8     /* number of hash functions to use */
#define CACHE_CUCKOORETRY          8     /* retries if the bucket is full */
#define CACHE_GROWTHRATIO       1.25     /* threshold to grow the table (75%) */
#endif

/** @defgroup cache module::cache */

typedef struct m_cache {
    pthread_rwlock_t *_lock;
    size_t _bucket_size;
    size_t _bucket_count;
    char **_index;
    char **_bucket;
    char *_basket;
    void (*_freeval)(void *);
} m_cache;

/**
 * @ingroup cache
 * @struct m_cache
 *
 * This structure supports the implementation of a thread safe hash table.
 *
 * Each key/value pair is stored in this structure with an overhead of
 * sizeof(uint16_t) + sizeof(char *) bytes.
 *
 * @Note Even if this structure is thread safe, it is not recommended
 * to use it in a context where concurrency is important, due to its
 * simplistic locking scheme. For good concurrency, it is better to use
 * the @ref m_hashtable structure and the related API.
```

```
64  *
65  */
66
67 typedef struct m_hashtable {
68     m_cache *_segment[256];
69 } m_hashtable;
70
71 /* ---------------------------------------------------------------------- */
72
73 public m_cache *cache_alloc(void (*freeval)(void *));
74
75 /* ---------------------------------------------------------------------- */
76
77 public inline void *cache_push(m_cache *h, const char *k, size_t l, void *v);
78
79 /* ---------------------------------------------------------------------- */
80
81 public inline void *cache_add(m_cache *h, const char *k, size_t l, void *v);
82
83 /* ---------------------------------------------------------------------- */
84
85 public void *cache_findexec(m_cache *h, const char *key, size_t len,
86                             void *(*function)(void *));
87
88 /* ---------------------------------------------------------------------- */
89
90 public inline void *cache_find(m_cache *h, const char *key, size_t len);
91
92 /* ---------------------------------------------------------------------- */
93
94 public void cache_foreach(m_cache *h,
95                           int (*function)(const char *, size_t, void *));
96
97 /* ---------------------------------------------------------------------- */
98
99 public void *cache_pop(m_cache *h, const char *key, size_t len);
100
101 /* ---------------------------------------------------------------------- */
102
103 public m_cache *cache_free(m_cache *h);
104
105 /* ---------------------------------------------------------------------- */
106 /* Segmented hash table */
107 /* ---------------------------------------------------------------------- */
108
109 public m_hashtable *hashtable_alloc(void (*freeval)(void *));
110
111 /* ---------------------------------------------------------------------- */
112
113 public inline void *hashtable_insert(m_hashtable *h, const char *key,
114                                      size_t len, void *val);
115
116 /* ---------------------------------------------------------------------- */
117
118 public inline void *hashtable_update(m_hashtable *h, const char *key,
119                                      size_t len, void *val);
120
121 /* ---------------------------------------------------------------------- */
122
123 public inline void *hashtable_remove(m_hashtable *h, const char *key,
124                                      size_t len);
125
126 /* ---------------------------------------------------------------------- */
127
128 public inline void *hashtable_findexec(m_hashtable *h, const char *key,
```

```
129                                     size_t len, void *(*function)(void *));
130
131 /* ------------------------------------------------------------------------ */
132
133 public void hashtable_foreach(m_hashtable *h,
134                                int (*function)(const char *, size_t, void *));
135
136 /* ------------------------------------------------------------------------ */
137
138 public inline void *hashtable_find(m_hashtable *h, const char *key, size_t len);
139
140 /* ------------------------------------------------------------------------ */
141
142 public inline m_hashtable *hashtable_free(m_hashtable *h);
143
144 /* ------------------------------------------------------------------------ */
145
146 /* _ENABLE_HASHTABLE */
147 #endif
148
149 #endif
```

```c
/*****************************************************************************
 *   Concrete Server                                                         *
 *   Copyright (c) 2005-2010 Raphael Prevost                                 *
 *                                                                           *
 *   This library is free software; you can redistribute it and/or           *
 *   modify it under the terms of the GNU Lesser General Public              *
 *   License as published by the Free Software Foundation; either            *
 *   version 2.1 of the License, or (at your option) any later version.      *
 *                                                                           *
 *   This library is distributed in the hope that it will be useful,         *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of          *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU        *
 *   Lesser General Public License for more details.                         *
 *                                                                           *
 *   You should have received a copy of the GNU Lesser General Public        *
 *   License along with this library; if not, write to the Free Software     *
 *   Foundation, Inc., 51 Franklin Street, 5th Floor, Boston, MA 02110-1301 USA *
 *                                                                           *
 *****************************************************************************/

#include "m_hashtable.h"

/* ------------------------------------------------------------------------- */
#ifdef _ENABLE_HASHTABLE
/* ------------------------------------------------------------------------- */

/* crc8 lookup table (Maxim/Dallas 1 wire) */
static uint8_t _crc_lut[256] = {
    0x00, 0x5e, 0xbc, 0xe2, 0x61, 0x3f, 0xdd, 0x83,
    0xc2, 0x9c, 0x7e, 0x20, 0xa3, 0xfd, 0x1f, 0x41,
    0x9d, 0xc3, 0x21, 0x7f, 0xfc, 0xa2, 0x40, 0x1e,
    0x5f, 0x01, 0xe3, 0xbd, 0x3e, 0x60, 0x82, 0xdc,
    0x23, 0x7d, 0x9f, 0xc1, 0x42, 0x1c, 0xfe, 0xa0,
    0xe1, 0xbf, 0x5d, 0x03, 0x80, 0xde, 0x3c, 0x62,
    0xbe, 0xe0, 0x02, 0x5c, 0xdf, 0x81, 0x63, 0x3d,
    0x7c, 0x22, 0xc0, 0x9e, 0x1d, 0x43, 0xa1, 0xff,
    0x46, 0x18, 0xfa, 0xa4, 0x27, 0x79, 0x9b, 0xc5,
    0x84, 0xda, 0x38, 0x66, 0xe5, 0xbb, 0x59, 0x07,
    0xdb, 0x85, 0x67, 0x39, 0xba, 0xe4, 0x06, 0x58,
    0x19, 0x47, 0xa5, 0xfb, 0x78, 0x26, 0xc4, 0x9a,
    0x65, 0x3b, 0xd9, 0x87, 0x04, 0x5a, 0xb8, 0xe6,
    0xa7, 0xf9, 0x1b, 0x45, 0xc6, 0x98, 0x7a, 0x24,
    0xf8, 0xa6, 0x44, 0x1a, 0x99, 0xc7, 0x25, 0x7b,
    0x3a, 0x64, 0x86, 0xd8, 0x5b, 0x05, 0xe7, 0xb9,
    0x8c, 0xd2, 0x30, 0x6e, 0xed, 0xb3, 0x51, 0x0f,
    0x4e, 0x10, 0xf2, 0xac, 0x2f, 0x71, 0x93, 0xcd,
    0x11, 0x4f, 0xad, 0xf3, 0x70, 0x2e, 0xcc, 0x92,
    0xd3, 0x8d, 0x6f, 0x31, 0xb2, 0xec, 0x0e, 0x50,
    0xaf, 0xf1, 0x13, 0x4d, 0xce, 0x90, 0x72, 0x2c,
    0x6d, 0x33, 0xd1, 0x8f, 0x0c, 0x52, 0xb0, 0xee,
    0x32, 0x6c, 0x8e, 0xd0, 0x53, 0x0d, 0xef, 0xb1,
    0xf0, 0xae, 0x4c, 0x12, 0x91, 0xcf, 0x2d, 0x73,
    0xca, 0x94, 0x76, 0x28, 0xab, 0xf5, 0x17, 0x49,
    0x08, 0x56, 0xb4, 0xea, 0x69, 0x37, 0xd5, 0x8b,
    0x57, 0x09, 0xeb, 0xb5, 0x36, 0x68, 0x8a, 0xd4,
    0x95, 0xcb, 0x29, 0x77, 0xf4, 0xaa, 0x48, 0x16,
    0xe9, 0xb7, 0x55, 0x0b, 0x88, 0xd6, 0x34, 0x6a,
    0x2b, 0x75, 0x97, 0xc9, 0x4a, 0x14, 0xf6, 0xa8,
    0x74, 0x2a, 0xc8, 0x96, 0x15, 0x4b, 0xa9, 0xf7,
    0xb6, 0xe8, 0x0a, 0x54, 0xd7, 0x89, 0x6b, 0x35
};

#define _key(b)  ((b) + sizeof(uint16_t))
#define _len(b)  (*((uint16_t *) (b)))
/* if your arch requires aligned reads, you can modify this macro */
```

```
66 #define _pad(b)  (*((uint16_t *) (b)))
67 #define _ptr(b)  (*((char **) ((b) + sizeof(uint16_t) + _pad(b))))
68 #define _val(b)            \
69 ((void *) *((int **) ((b) + sizeof(uint16_t) + _pad(b) + sizeof(int *))))
70
71 #define _set_len(b, l)    \
72 do { *((uint16_t *) (b)) = (l); } while (0)
73
74 #define _set_key(b, k, l) \
75 do { memcpy((b) + sizeof(uint16_t), (k), (l)); } while (0)
76
77 #define _set_ptr(b, p)    \
78 do { *((char **) ((b) + sizeof(uint16_t) + _pad(b))) = (p); } while (0)
79
80 #define _cpy_ptr(b0, b1)  \
81 do { \
82     *((char **) ((b0) + sizeof(uint16_t) + _pad(b0))) = \
83     *((char **) ((b1) + sizeof(uint16_t) + _pad(b1)));  \
84 } while (0)
85
86 #define _clr_ptr(b)       \
87 do { *((char **) ((b) + sizeof(uint16_t) + _pad(b))) = NULL; } while (0)
88
89 #define _set_val(b, p)    \
90 do { \
91     *((int **) ((b) + sizeof(uint16_t) + _pad(b) + sizeof(int *))) = (p); \
92 } while (0)
93
94 /* ---------------------------------------------------------------------- */
95
96 static inline uint32_t _hash(const char *data, size_t len, uint32_t initval)
97 {
98     /* Bob Jenkins' lookup3 hash function, we use it for the bucket index */
99     register uint32_t a, b, c;                  /* internal state */
100    union { const void *ptr; size_t i; } u;     /* needed for Mac Powerbook G4 */
101
102    #if (defined(__BYTE_ORDER) && defined(__LITTLE_ENDIAN) && \
103         __BYTE_ORDER == __LITTLE_ENDIAN) || \
104        (defined(i386) || defined(__i386__) || defined(__i486__) || \
105         defined(__i586__) || defined(__i686__) || defined(vax) || \
106         defined(MIPSEL))
107        #define HASH_LITTLE_ENDIAN 1
108        #define HASH_BIG_ENDIAN 0
109    #elif (defined(__BYTE_ORDER) && defined(__BIG_ENDIAN) && \
110           __BYTE_ORDER == __BIG_ENDIAN) || \
111          (defined(sparc) || defined(POWERPC) || defined(mc68000) || \
112           defined(sel))
113        #define HASH_LITTLE_ENDIAN 0
114        #define HASH_BIG_ENDIAN 1
115    #else
116        #define HASH_LITTLE_ENDIAN 0
117        #define HASH_BIG_ENDIAN 0
118    #endif
119
120    #define rot(x, k) (((x) << (k)) | ((x) >> (32 - (k))))
121
122    #define mix(a, b, c) \
123    { \
124        a -= c; a ^= rot(c, 4); c += b; \
125        b -= a; b ^= rot(a, 6); a += c; \
126        c -= b; c ^= rot(b, 8); b += a; \
127        a -= c; a ^= rot(c,16); c += b; \
128        b -= a; b ^= rot(a,19); a += c; \
129        c -= b; c ^= rot(b, 4); b += a; \
130    }
```

```
131
132     #define final(a, b, c) \
133     { \
134         c ^= b; c -= rot(b,14); \
135         a ^= c; a -= rot(c,11); \
136         b ^= a; b -= rot(a,25); \
137         c ^= b; c -= rot(b,16); \
138         a ^= c; a -= rot(c, 4); \
139         b ^= a; b -= rot(a,14); \
140         c ^= b; c -= rot(b,24); \
141     }
142
143     /* set up the internal state */
144     a = b = c = 0x52661314 + ((uint32_t) len) + initval; /* wo ai ni Lulu */
145
146     u.ptr = data;
147
148     #if (HASH_LITTLE_ENDIAN)
149     if ((u.i & 0x3) == 0) {
150         const uint32_t *k = (const uint32_t *) data; /* read 32-bit chunks */
151         #ifdef DEBUG
152         const uint8_t *k8;
153         #endif
154
155         /* all but last block: aligned reads and affect 32 bits of (a, b, c) */
156         while (len > 12) {
157             a += k[0]; b += k[1]; c += k[2];
158             mix(a, b, c);
159             len -= 12; k += 3;
160         }
161
162         /* - handle the last (probably partial) block
163          *
164          * "k[2]&0xffffff" actually reads beyond the end of the string, but
165          * then masks off the part it's not allowed to read.  Because the
166          * string is aligned, the masked-off tail is in the same word as the
167          * rest of the string.  Every machine with memory protection I've seen
168          * does it on word boundaries, so is OK with this.  But VALGRIND will
169          * still catch it and complain.  The masking trick does make the hash
170          * noticably faster for short strings (like English words).
171          */
172         #ifndef DEBUG
173
174         switch(len) {
175             case 12: c += k[2]; b += k[1]; a += k[0]; break;
176             case 11: c += k[2] & 0xffffff; b += k[1]; a += k[0]; break;
177             case 10: c += k[2] & 0xffff; b += k[1]; a += k[0]; break;
178             case 9 : c += k[2] & 0xff; b += k[1]; a += k[0]; break;
179             case 8 : b += k[1]; a += k[0]; break;
180             case 7 : b += k[1] & 0xffffff; a += k[0]; break;
181             case 6 : b += k[1] & 0xffff; a += k[0]; break;
182             case 5 : b += k[1] & 0xff; a += k[0]; break;
183             case 4 : a += k[0]; break;
184             case 3 : a += k[0] & 0xffffff; break;
185             case 2 : a += k[0] & 0xffff; break;
186             case 1 : a += k[0] & 0xff; break;
187             case 0 : return c; /* zero length strings require no mixing */
188         }
189
190         #else /* make valgrind happy */
191
192         k8 = (const uint8_t *) k;
193         switch(len) {
194             case 12: c += k[2]; b += k[1]; a += k[0]; break;
195             case 11: c += ((uint32_t) k8[10]) << 16;  /* fall through */
```

```
196             case 10: c += ((uint32_t) k8[9]) << 8;      /* fall through */
197             case 9 : c += k8[8];                        /* fall through */
198             case 8 : b += k[1]; a += k[0]; break;
199             case 7 : b += ((uint32_t) k8[6]) << 16;     /* fall through */
200             case 6 : b += ((uint32_t) k8[5]) << 8;      /* fall through */
201             case 5 : b += k8[4];                        /* fall through */
202             case 4 : a += k[0]; break;
203             case 3 : a += ((uint32_t) k8[2]) << 16;     /* fall through */
204             case 2 : a += ((uint32_t) k8[1]) << 8;      /* fall through */
205             case 1 : a += k8[0]; break;
206             case 0 : return c;
207         }
208
209         #endif /* !valgrind */
210
211     } else if ((u.i & 0x1) == 0) {
212         const uint16_t *k = (const uint16_t *) data; /* read 16-bit chunks */
213         const uint8_t *k8;
214
215         /* - all but last block: aligned reads and different mixing */
216         while (len > 12) {
217             a += k[0] + (((uint32_t) k[1]) << 16);
218             b += k[2] + (((uint32_t) k[3]) << 16);
219             c += k[4] + (((uint32_t) k[5]) << 16);
220             mix(a, b, c);
221             len -= 12; k += 6;
222         }
223
224         /* - handle the last (probably partial) block */
225         k8 = (const uint8_t *) k;
226         switch(len) {
227             case 12: c += k[4] + (((uint32_t) k[5]) << 16);
228                      b += k[2] + (((uint32_t) k[3]) << 16);
229                      a += k[0] + (((uint32_t) k[1]) << 16);
230                      break;
231             case 11: c += ((uint32_t) k8[10]) << 16;      /* fall through */
232             case 10: c += k[4];
233                      b += k[2] + (((uint32_t) k[3]) << 16);
234                      a += k[0] + (((uint32_t) k[1]) << 16);
235                      break;
236             case 9 : c += k8[8];                          /* fall through */
237             case 8 : b += k[2] + (((uint32_t) k[3]) << 16);
238                      a += k[0] + (((uint32_t) k[1]) << 16);
239                      break;
240             case 7 : b += ((uint32_t) k8[6]) << 16;       /* fall through */
241             case 6 : b += k[2];
242                      a += k[0]+(((uint32_t) k[1]) << 16);
243                      break;
244             case 5 : b += k8[4];                          /* fall through */
245             case 4 : a += k[0] + (((uint32_t) k[1]) << 16);
246                      break;
247             case 3 : a += ((uint32_t) k8[2]) << 16;       /* fall through */
248             case 2 : a += k[0];
249                      break;
250             case 1 : a += k8[0];
251                      break;
252             case 0 : return c; /* zero length requires no mixing */
253         }
254
255     }
256     #else
257     if ((u.i & 0x3) == 0) {
258         const uint32_t *k = (const uint32_t *) data; /* read 32-bit chunks */
259         #ifdef DEBUG
260         const uint8_t *k8;
```

```
261         #endif
262
263         /* all but last block: aligned reads and affect 32 bits of (a, b, c) */
264         while (len > 12) {
265             a += k[0]; b += k[1]; c += k[2];
266             mix(a, b, c);
267             len -= 12; k += 3;
268         }
269
270         /* - handle the last (probably partial) block */
271         #ifndef DEBUG
272
273         switch(len) {
274             case 12: c += k[2]; b += k[1]; a += k[0]; break;
275             case 11: c += k[2] & 0xffffff00; b += k[1]; a += k[0]; break;
276             case 10: c += k[2] & 0xffff0000; b += k[1]; a += k[0]; break;
277             case 9 : c += k[2] & 0xff000000; b += k[1]; a += k[0]; break;
278             case 8 : b += k[1]; a += k[0]; break;
279             case 7 : b += k[1] & 0xffffff00; a += k[0]; break;
280             case 6 : b += k[1] & 0xffff0000; a += k[0]; break;
281             case 5 : b += k[1] & 0xff000000; a += k[0]; break;
282             case 4 : a += k[0]; break;
283             case 3 : a += k[0] & 0xffffff00; break;
284             case 2 : a += k[0] & 0xffff0000; break;
285             case 1 : a += k[0] & 0xff000000; break;
286             case 0 : return c; /* zero length strings require no mixing */
287         }
288
289         #else  /* make valgrind happy */
290
291         k8 = (const uint8_t *) k;
292         switch(len) {
293             case 12: c += k[2]; b += k[1]; a += k[0]; break;
294             case 11: c += ((uint32_t) k8[10]) << 8;  /* fall through */
295             case 10: c += ((uint32_t) k8[9]) << 16;  /* fall through */
296             case 9 : c += ((uint32_t) k8[8]) << 24;  /* fall through */
297             case 8 : b += k[1]; a += k[0]; break;
298             case 7 : b += ((uint32_t) k8[6]) << 8;   /* fall through */
299             case 6 : b += ((uint32_t) k8[5]) << 16;  /* fall through */
300             case 5 : b += ((uint32_t) k8[4]) << 24;  /* fall through */
301             case 4 : a += k[0]; break;
302             case 3 : a += ((uint32_t) k8[2]) << 8;   /* fall through */
303             case 2 : a += ((uint32_t) k8[1]) << 16;  /* fall through */
304             case 1 : a += ((uint32_t) k8[0]) << 24; break;
305             case 0 : return c;
306         }
307
308         #endif /* !VALGRIND */
309
310     }
311     #endif
312     else { /* need to read the key one byte at a time */
313         const uint8_t *k = (const uint8_t *) data;
314
315         /* all but the last block: affect some 32 bits of (a, b, c) */
316         while (len > 12) {
317             a += k[0];
318             a += ((uint32_t) k[1]) << 8;
319             a += ((uint32_t) k[2]) << 16;
320             a += ((uint32_t) k[3]) << 24;
321             b += k[4];
322             b += ((uint32_t) k[5]) << 8;
323             b += ((uint32_t) k[6]) << 16;
324             b += ((uint32_t) k[7]) << 24;
325             c += k[8];
```

```
326             c += ((uint32_t) k[9]) << 8;
327             c += ((uint32_t) k[10]) << 16;
328             c += ((uint32_t) k[11]) << 24;
329             mix(a, b, c);
330             len -= 12; k += 12;
331         }
332
333         /* - last block: affect all 32 bits of (c) */
334         switch(len) {
335             case 12: c += ((uint32_t) k[11]) << 24;
336             case 11: c += ((uint32_t) k[10]) << 16;
337             case 10: c += ((uint32_t) k[9]) << 8;
338             case 9 : c += k[8];
339             case 8 : b += ((uint32_t) k[7]) << 24;
340             case 7 : b += ((uint32_t) k[6]) << 16;
341             case 6 : b += ((uint32_t) k[5]) << 8;
342             case 5 : b += k[4];
343             case 4 : a += ((uint32_t) k[3]) << 24;
344             case 3 : a += ((uint32_t) k[2]) << 16;
345             case 2 : a += ((uint32_t) k[1]) << 8;
346             case 1 : a += k[0]; break;
347             case 0 : return c;
348         }
349     }
350
351     final(a, b, c);
352
353     return c;
354 }
355
356 /* -------------------------------------------------------------------------- */
357
358 static inline uint8_t _crc8(const char *string, size_t len)
359 {
360     register uint8_t crc = 0xff;
361     while (len --) crc = _crc_lut[*string ++ ^ crc];
362     return crc;
363 }
364
365 /* -------------------------------------------------------------------------- */
366
367 static inline int _memcmp32(const char *a, const char *b, size_t len)
368 {
369     register unsigned int i = 0;
370
371     switch (len) {
372     case  8: return (*((uint64_t *) a) == *((uint64_t *) b)) ? 0 : -1;
373     case  5: if (a[4] != b[4]) return -1;
374     case  4: return (*((uint32_t *) a) == *((uint32_t *) b)) ? 0 : -1;
375     case  3: if (a[2] != b[2]) return -1;
376     case  2: return (*((uint16_t *) a) == *((uint16_t *) b)) ? 0 : -1;
377     case  1: return (*a == *b) ? 0 : -1;
378     default:
379         for (i = 0; i < len - sizeof(uint32_t); i += sizeof(uint32_t)) {
380             if (*((uint32_t *) (a + i)) != *((uint32_t *) (b + i)))
381                 return -1;
382
383             if (i >= len - (i + sizeof(uint32_t)))
384                 return 0;
385
386             if (*((uint32_t *) (a + len - (i + sizeof(uint32_t)))) !=
387                 *((uint32_t *) (b + len - (i + sizeof(uint32_t)))))
388                 return -1;
389         }
390     }
```

```
391
392     return 0;
393 }
394
395 /* ------------------------------------------------------------------------ */
396
397 static inline void *_cache_push(m_cache *h, char *key, int replace)
398 {
399     unsigned int i = 0, index = 0;
400     unsigned int retry = 0;
401     unsigned int hash_cache[CACHE_HASHFNCOUNT];
402     char *tmp = NULL, *slot = NULL;
403     uint32_t mask = h->_bucket_size - 1;
404
405     if (replace) {
406         /* look for the key in the buckets */
407         for (i = 0; i < CACHE_HASHFNCOUNT; i ++) {
408             index = _hash(_key(key), _len(key), 0x520 + i) & mask;
409             hash_cache[i] = index;
410
411             if (! (slot = h->_bucket[index]) ) {
412                 /* got a free slot */
413                 h->_bucket[index] = key;
414                 h->_bucket_count ++;
415                 return NULL;
416             }
417
418             if (! _memcmp32(key, slot, _len(key) + sizeof(uint16_t))) {
419                 if (replace == -1) tmp = _val(key);
420                 else { tmp = _val(slot); _set_val(slot, _val(key)); }
421                 free(key - sizeof(char *));
422                 return tmp;
423             }
424         }
425
426         /* couldn't find it, look in the basket */
427         for (slot = h->_basket; slot; slot = _ptr(slot)) {
428             if (! _memcmp32(key, slot, _len(key) + sizeof(uint16_t))) {
429                 if (replace == -1) tmp = _val(key);
430                 else { tmp = _val(slot); _set_val(slot, _val(key)); }
431                 free(key - sizeof(char *));
432                 return tmp;
433             }
434         }
435
436     } else {
437
438         /* no replacement, look directly for a free slot */
439         for (i = 0; i < CACHE_HASHFNCOUNT; i ++) {
440             index = _hash(_key(key), _len(key), 0x520 + i) & mask;
441             hash_cache[i] = index;
442             if (! h->_bucket[index]) {
443                 /* this slot is free, we can insert */
444                 h->_bucket[index] = key;
445                 h->_bucket_count ++;
446                 return NULL;
447             }
448         }
449     }
450
451     /* we did not find a free slot, try cuckoo hashing */
452     for (index = hash_cache[0]; retry < CACHE_CUCKOORETRY; retry ++) {
453         /* get data off the last slot and replace them */
454         tmp = h->_bucket[index]; h->_bucket[index] = key; key = tmp;
455
```

```
456         /* get rid of tombstones */
457         if (! _len(key)) return NULL;
458
459         for (i = 0; i < CACHE_HASHFNCOUNT; i ++) {
460             index = _hash(_key(key), _len(key), 0x520 + i) & mask;
461             hash_cache[i] = index;
462
463             if (! h->_bucket[index]) {
464                 /* this slot is free, we can insert */
465                 h->_bucket[index] = key;
466                 h->_bucket_count ++;
467                 return NULL;
468             }
469         }
470
471     }
472
473     /* cuckoo hashing did not help, store the key in the basket... */
474     _set_ptr(key, h->_basket); h->_basket = key; h->_bucket_count ++;
475
476     return NULL;
477 }
478
479 /* ---------------------------------------------------------------------- */
480
481 static inline int _cache_resize(m_cache *h, uint32_t size)
482 {
483     char **bucket = NULL, **next = NULL;
484     char *tmp = NULL;
485
486     if (unlikely(! h)) {
487         debug("_cache_resize(): bad parameters.\n");
488         return -1;
489     }
490
491     if (h->_bucket_count * CACHE_GROWTHRATIO < h->_bucket_size) return 0;
492
493     /* round the size to the next highest power of 2 */
494     size --; size |= size >> 1; size |= size >> 2;
495     size |= size >> 4; size |= size >> 8; size |= size >> 16;
496     size ++; size += (size == 0);
497
498     free(h->_bucket);
499
500     /* try to resize the bucket array */
501     if (! (h->_bucket = calloc(size, sizeof(*bucket))) ) {
502         perror(ERR(_cache_resize, calloc));
503         return -1;
504     }
505
506     /* update the state */
507     h->_bucket_size = size; h->_bucket_count = 0; h->_basket = NULL;
508     bucket = h->_index; h->_index = NULL;
509
510     /* rehash old buckets */
511     while (bucket) {
512         tmp = ((char *) bucket) + sizeof(char *);
513         next = (char **) *bucket;
514
515         if (_len(tmp)) {
516             _cache_push(h, tmp, 0); *bucket = NULL;
517             *bucket = (char *) h->_index; h->_index = bucket;
518         } else free(bucket);
519
520         bucket = next;
```

```
521     }
522
523     return 0;
524 }
525
526 /* ---------------------------------------------------------------------- */
527
528 public m_cache *cache_alloc(void (*freeval)(void *))
529 {
530     m_cache *h = malloc(sizeof(*h));
531
532     if (! h) {
533         perror(ERR(cache_alloc, malloc));
534         return NULL;
535     }
536
537     if (! (h->_lock = malloc(sizeof(*h->_lock))) ) {
538         perror(ERR(cache_alloc, malloc));
539         goto _err_lock;
540     }
541
542     if (pthread_rwlock_init(h->_lock, NULL) == -1) {
543         perror(ERR(cache_alloc, pthread_rwlock_init));
544         goto _err_init;
545     }
546
547     h->_bucket = NULL;
548     h->_bucket_count = h->_bucket_size = 0; h->_basket = NULL;
549     h->_index = NULL;
550     h->_freeval = freeval;
551
552     if (_cache_resize(h, 2) == -1) {
553         debug("cache_alloc(): could not resize the hash table.\n");
554         goto _err_size;
555     }
556
557     return h;
558
559 _err_size:
560     free(h->_bucket);
561     free(h->_basket);
562 _err_init:
563     free(h->_lock);
564 _err_lock:
565     free(h);
566
567     return NULL;
568 }
569
570 /* ---------------------------------------------------------------------- */
571
572 static void *_cache_add(m_cache *h, const char *key, size_t len, void *val,
573                         int replace)
574 {
575     char *mkey = NULL;
576     char **keyptr = NULL;
577
578     if (unlikely(! h || ! key || ! len)) {
579         debug("_cache_add(): bad parameters.\n");
580         return val;
581     }
582
583     /* replace the key by a dynamically allocated one */
584     if (! (keyptr = malloc(sizeof(uint16_t) + (3 * sizeof(int *)) + len)) ) {
585         perror(ERR(cache_push, malloc));
```

```
586         return val;
587     }
588
589     mkey = ((char *) keyptr) + sizeof(char *);
590
591     _set_len(mkey, len); _set_key(mkey, key, len); _set_val(mkey, val);
592
593     pthread_rwlock_wrlock(h->_lock);
594
595     if (! (val = _cache_push(h, mkey, replace)) ) {
596         *keyptr = (char *) h->_index; h->_index = keyptr;
597     }
598
599     /* try to expand the hashtable if the load is too important */
600     if (h->_basket && _ptr(h->_basket))
601         _cache_resize(h, h->_bucket_size + 1);
602
603     pthread_rwlock_unlock(h->_lock);
604
605     return val;
606 }
607
608 /* ---------------------------------------------------------------------- */
609
610 public inline void *cache_push(m_cache *h, const char *k, size_t l, void *v)
611 {
612     return _cache_add(h, k, l, v, 1);
613 }
614
615 /* ---------------------------------------------------------------------- */
616
617 public inline void *cache_add(m_cache *h, const char *k, size_t l, void *v)
618 {
619     return _cache_add(h, k, l, v, -1);
620 }
621
622 /* ---------------------------------------------------------------------- */
623
624 public void *cache_findexec(m_cache *h, const char *key, size_t len,
625                             void *(*function)(void *))
626 {
627     register unsigned int i = 0, j = 0;
628     const char *ptr = NULL;
629     void *res = NULL;
630     uint32_t mask = 0;
631
632     if (unlikely(! h || ! key || ! len)) {
633         debug("cache_findexec(): bad parameters.\n");
634         return NULL;
635     }
636
637     pthread_rwlock_rdlock(h->_lock);
638
639     mask = h->_bucket_size - 1;
640
641     for (i = 0; i < CACHE_HASHFNCOUNT; i ++) {
642         j = _hash(key, len, 0x520 + i) & mask;
643
644         /* if an empty slot is found, no need to look further */
645         if (! (ptr = h->_bucket[j])) break;
646
647         if (_len(ptr) != len) continue;
648
649         if (_memcmp32(_key(ptr), key, len) == 0) {
650             res = _val(ptr); if (function) res = function(res);
```

```
651             pthread_rwlock_unlock(h->_lock);
652             return res;
653         }
654     }
655
656     /* unlucky, scan the basket for orphan keys */
657     for (ptr = h->_basket; ptr; ptr = _ptr(ptr)) {
658         if (_len(ptr) == len) {
659             if (_memcmp32(_key(ptr), key, len) == 0) {
660                 res = _val(ptr); if (function) res = function(res);
661                 pthread_rwlock_unlock(h->_lock);
662                 return res;
663             }
664         }
665     }
666
667     pthread_rwlock_unlock(h->_lock);
668
669     return NULL;
670 }
671
672 /* ---------------------------------------------------------------------- */
673
674 public inline void *cache_find(m_cache *h, const char *key, size_t len)
675 {
676     return cache_findexec(h, key, len, NULL);
677 }
678
679 /* ---------------------------------------------------------------------- */
680
681 public void cache_foreach(m_cache *h,
682                           int (*function)(const char *, size_t, void *))
683 {
684     char **bucket = NULL, *tmp = NULL;
685
686     if (! h || ! function) {
687         debug("cache_foreach(): bad parameters.\n");
688         return;
689     }
690
691     pthread_rwlock_wrlock(h->_lock);
692
693     for (bucket = h->_index; bucket; bucket = (char **) *bucket) {
694         tmp = ((char *) bucket) + sizeof(char *);
695
696         if (_len(tmp)) {
697             if (function(_key(tmp), _len(tmp), _val(tmp)) == -1) {
698                 /* delete the record */
699                 _set_len(tmp, 0);
700             }
701         }
702     }
703
704     pthread_rwlock_unlock(h->_lock);
705
706     return;
707 }
708
709 /* ---------------------------------------------------------------------- */
710
711 public void *cache_pop(m_cache *h, const char *key, size_t len)
712 {
713     register unsigned int i = 0, j = 0;
714     char *ptr = NULL, *tmp = NULL, *prev = NULL;
715     void *result = NULL;
```

```
716     uint32_t mask = 0;
717
718     if (unlikely(! h || ! key || ! len)) {
719         debug("cache_pop(): bad parameters.\n");
720         return NULL;
721     }
722
723     pthread_rwlock_wrlock(h->_lock);
724
725     mask = h->_bucket_size - 1;
726
727     for (i = 0; i < CACHE_HASHFNCOUNT; i ++) {
728         j = _hash(key, len, 0x520 + i) & mask;
729
730         if (! h->_bucket[j] || _len(h->_bucket[j]) != len) continue;
731
732         if (_memcmp32(_key(h->_bucket[j]), key, len) == 0) {
733             /* remove from the bucket */
734             result = _val(h->_bucket[j]); ptr = _ptr(h->_bucket[j]);
735             /* a length of 0 indicates a tombstone */
736             _set_len(h->_bucket[j], 0);
737             pthread_rwlock_unlock(h->_lock);
738             return result;
739         }
740     }
741
742     /* unlucky, scan the basket for orphan keys */
743     for (tmp = prev = h->_basket; tmp; prev = tmp, tmp = _ptr(tmp)) {
744         if (_len(tmp) == len) {
745             if (_memcmp32(_key(tmp), key, len) == 0) {
746                 /* just remove the orphan sibling from the basket */
747                 result = _val(tmp);
748                 if (tmp == h->_basket) h->_basket = _ptr(tmp);
749                 else { _set_ptr(prev, _ptr(tmp)); _set_len(tmp, 0); }
750
751                 pthread_rwlock_unlock(h->_lock);
752
753                 return result;
754             }
755         }
756     }
757
758     /* garbage collection if necessary */
759     _cache_resize(h, h->_bucket_count * CACHE_GROWTHRATIO);
760
761     pthread_rwlock_unlock(h->_lock);
762
763     return NULL;
764 }
765
766 /* ------------------------------------------------------------------------ */
767
768 public size_t cache_footprint(m_cache *h, size_t *overhead)
769 {
770     char **bucket = NULL, **next = NULL;
771     char *tmp = NULL;
772     size_t key = 0;
773     size_t ret = sizeof(*h);
774
775     if (! h) {
776         debug("cache_footprint(): bad parameters.\n");
777         return 0;
778     }
779
780     /* the lock is dynamically allocated */
```

```
781     ret += sizeof(*h->_lock);
782
783     pthread_rwlock_wrlock(h->_lock);
784
785     if (h->_bucket_size) {
786         /* segment bucket size */
787         ret += h->_bucket_size * sizeof(*h->_bucket);
788         /* keys */
789         for (bucket = h->_index; bucket; bucket = next) {
790             tmp = ((char *) bucket) + sizeof(char *);
791             next = (char **) *bucket;
792             if (_len(tmp)) {
793                 /* key length + key recorded size + next and value pointers */
794                 ret += sizeof(char *) + _len(tmp) + sizeof(uint16_t) +
795                        sizeof(char *) + sizeof(void *);
796                 /* key length and value pointer are not overhead */
797                 key += sizeof(uint16_t) + _len(tmp) + sizeof(void *);
798             }
799         }
800     }
801
802     pthread_rwlock_unlock(h->_lock);
803
804     if (overhead) *overhead = ret - key;
805
806     return ret;
807 }
808
809 /* ---------------------------------------------------------------------- */
810
811 public m_cache *cache_free(m_cache *h)
812 {
813     char **bucket = NULL, **next = NULL;
814     char *keyptr = NULL;
815
816     if (! h) return NULL;
817
818     for (bucket = h->_index; bucket; bucket = next) {
819         keyptr = ((char *) bucket) + sizeof(char *);
820         next = (char **) *bucket;
821         if (h->_freeval && _len(keyptr)) h->_freeval(_val(keyptr));
822         free(bucket);
823     }
824
825     free(h->_bucket);
826     pthread_rwlock_destroy(h->_lock);
827     free(h->_lock); free(h);
828
829     return NULL;
830 }
831
832 /* ---------------------------------------------------------------------- */
833
834 public m_hashtable *hashtable_alloc(void (*freeval)(void *))
835 {
836     unsigned int i = 0;
837
838     m_hashtable *h = malloc(sizeof(*h));
839
840     if (! h) {
841         perror(ERR(hash_alloc, malloc));
842         return NULL;
843     }
844
845     for (i = 0; i < 256; i ++) h->_segment[i] = cache_alloc(freeval);
```

```
846
847     return h;
848 }
849
850 /* ---------------------------------------------------------------------- */
851
852 public size_t hashtable_footprint(m_hashtable *h, size_t *overhead)
853 {
854     unsigned int i = 0;
855     size_t key = 0, over = 0;
856     size_t ret = sizeof(*h);
857
858     if (! h) {
859         debug("hashtable_footprint(): bad parameters.\n");
860         return 0;
861     }
862
863     ret += 256 * sizeof(void *);
864
865     for (i = 0; i < 256; i ++) {
866         ret += cache_footprint(h->_segment[i], & over);
867         key += over; over = 0;
868     }
869
870     if (overhead) *overhead = key;
871
872     return ret;
873 }
874
875 /* ---------------------------------------------------------------------- */
876
877 public inline void *hashtable_insert(m_hashtable *h, const char *key,
878                                      size_t len, void *val)
879 {
880     if (! key || ! len) return NULL;
881
882     if (! h) return val;
883
884     return _cache_add(h->_segment[_crc8(key, len)], key, len, val, -1);
885 }
886
887 /* ---------------------------------------------------------------------- */
888
889 public inline void *hashtable_update(m_hashtable *h, const char *key,
890                                      size_t len, void *val)
891 {
892     if (! key || ! len) return NULL;
893
894     if (! h) return val;
895
896     return _cache_add(h->_segment[_crc8(key, len)], key, len, val, 1);
897 }
898
899 /* ---------------------------------------------------------------------- */
900
901 public inline void *hashtable_remove(m_hashtable *h, const char *key, size_t len)
902 {
903     if (! h || ! key || ! len) return NULL;
904
905     return cache_pop(h->_segment[_crc8(key, len)], key, len);
906 }
907
908 /* ---------------------------------------------------------------------- */
909
910 public inline void *hashtable_findexec(m_hashtable *h, const char *key,
```

```
911                                     size_t len, void *(*function)(void *))
912 {
913     if (! h || ! key || ! len) return NULL;
914 
915     return cache_findexec(h->_segment[_crc8(key, len)], key, len, function);
916 }
917 
918 /* ---------------------------------------------------------------------- */
919 
920 public void hashtable_foreach(m_hashtable *h,
921                               int (*function)(const char *, size_t, void *))
922 {
923     unsigned int i = 0;
924 
925     if (! h || ! function) return;
926 
927     for (i = 0; i < 256; i ++) {
928         if (h->_segment[i]->_index)
929             cache_foreach(h->_segment[i], function);
930     }
931 
932     return;
933 }
934 
935 /* ---------------------------------------------------------------------- */
936 
937 public inline void *hashtable_find(m_hashtable *h, const char *key, size_t len)
938 {
939     if (! h || ! key || ! len) return NULL;
940 
941     return cache_find(h->_segment[_crc8(key, len)], key, len);
942 }
943 
944 /* ---------------------------------------------------------------------- */
945 
946 public inline m_hashtable *hashtable_free(m_hashtable *h)
947 {
948     unsigned int i = 0;
949 
950     if (! h) return NULL;
951 
952     for (i = 0; i < 256; i ++) cache_free(h->_segment[i]); free(h);
953 
954     return NULL;
955 }
956 
957 /* ---------------------------------------------------------------------- */
958 #else
959 /* ---------------------------------------------------------------------- */
960 
961 /* Hashtable support will not be compiled in the Concrete Library */
962 __attribute__ ((unused)) static int __dummy__ = 0;
963 
964 /* ---------------------------------------------------------------------- */
965 #endif
966 /* ---------------------------------------------------------------------- */
```