CHAPITRE 2

Simulation d'un cache et comparaison des algorithmes de remplacement

2.1 Objectif

Le but de l'exercice est d'explorer la gestion de cache (et de mémoire virtuelle, car de nombreux problèmes sont identiques) grâce à un **simulateur de cache**. Votre mission est de programmer, d'analyser et de comparer différents **algorithmes de remplacement de bloc**. Le reste du code nécessaire à votre simulateur est fourni dans le répertoire `Cache` à l'endroit habituel.

Si le code que vous avez à écrire est assez court, celui qui est fourni est assez important. En conséquence, l'exercice est aussi, d'une certaine manière, une introduction à l'organisation modulaire des programmes en C.

L'énoncé semble long, mais il n'est pas nécessaire de tout lire en détail pour commencer à travailler ! Commencez par lire avec soin la section 2.2 qui présente le problème, ainsi que 2.5 qui décrit le travail à effectuer. Parcourez le reste rapidement. Certes, la section 2.3 est fondamentale puisqu'elle décrit le cœur du sujet, mais il vous suffit de la lire au fur et à mesure que votre travail avance. La section 2.4 est une sorte de manuel de référence sur le code fourni, nécessaire puisque le source ne vous en est pas donné ; vous vous y reporterez quand vous aurez besoin des fonctions qui y sont décrites (en particulier vous aurez surement besoin des indications des sections 2.4.4 et 2.4.5). Enfin la lecture de 2.6 ne sera vraiment utile que lorsque vous aurez réalisé toutes les stratégies de remplacement demandées.

2.2 Présentation du cache

On suppose qu'un programme a besoin de lire et d'écrire dans un gros fichier. Ce fichier est composé d'enregistrements (`records`) de taille fixe (`recordsz`). Le programme accède à ce fichier plus ou moins aléatoirement, enregistrement par enregistrement, en fournissant l'indice de l'enregistrement visé (`ind`) dans le fichier. Le fichier étant vraiment très gros,

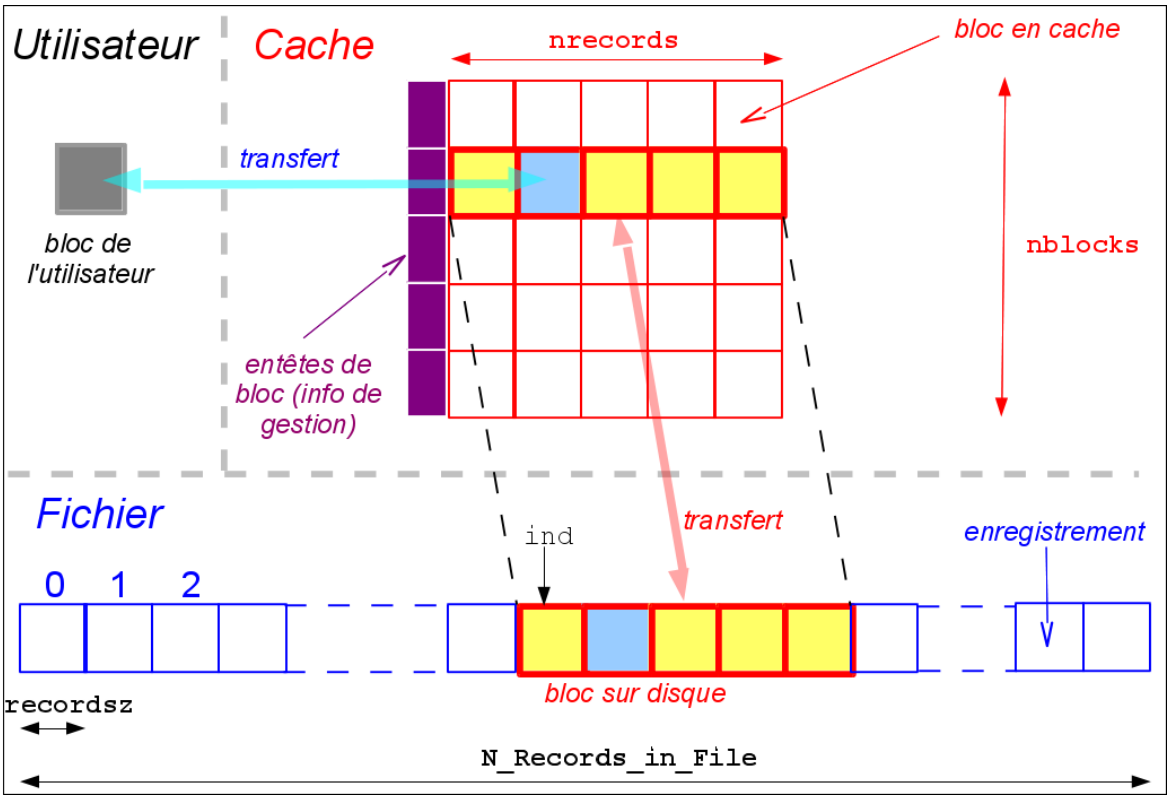

FIGURE 2.1 – Schéma général du cache et de son fichier.

on décide d'essayer d'améliorer les performances d'accès en interposant un cache entre le buffer de l'utilisateur et le fichier (figure 2.1).

Afin de minimiser encore le nombre d'entrées-sorties, l'unité de bloc du cache ne sera pas l'enregistrement lui-même, mais un groupe (*cluster*) de `nrecords` enregistrements consécutifs. Le cache comporte `nblocks` blocs de ce type (soit donc `nblocs`×`nrecords` enregistrements ou encore `nblocs`×`nrecords`×`recordsz` caractères). Le bloc sera l'*unité de transfert* entre le cache et le fichier. Chaque bloc du cache est soit *libre* (on dit aussi non *valide* car le bloc ne contient pas d'information utile) soit affecté à un bloc de taille analogue dans le fichier.

L'algorithme est bien connu :

- L'utilisateur fournit le numéro de l'enregistrement (`ind`) qu'il veut lire ou écrire ainsi que l'adresse d'un buffer dans son espace d'adressage ;
- On regarde si cet enregistrement est dans le cache ; si oui on transfère une copie de l'enregistrement depuis le cache vers le buffer de l'utilisateur pour une lecture, ou en sens inverse pour une écriture ; la requête de l'utilisateur n'induit alors aucune entrée-sortie ;
- Si le cache ne contient pas l'enregistrement demandé, on cherche un bloc libre (i.e., non valide) dans le cache et y fait monter le bloc en copiant son contenu depuis le fichier ; on est alors ramené au problème précédent ;
- Si l'opération précédente n'est pas possible car le cache est plein (i.e., tous ses blocs sont valides), on libère un des blocs du cache pour y faire monter le bloc disque et donc on change son affectation ; sélectionner le bloc à libérer est le rôle de l'*algorithme*

de remplacement de bloc ; bien entendu, si le bloc choisi a été modifié pendant son temps de résidence dans le cache, il faut le réécrire sur disque avant de changer son affectation.

Afin d'assurer la gestion du cache et du remplacement de bloc, on dote chaque bloc (*cluster*) d'un en-tête indiquant l'affectation du bloc au fichier ainsi que de deux indicateurs (bits) :

— **V**, le **bit de validité**, indique que le bloc contient une information valide ; tous les blocs du cache sont initialement invalides et le cache est plein quand tous ses blocs sont valides ;

— **M**, le **bit de modification**, indique si le bloc (c'est-à-dire l'un de ses enregistrements) a été modifié pendant sa résidence dans le cache ; ce bit est mis à 1 chaque fois qu'on écrit dans le bloc, et remis à 0 quand le bloc est réécrit sur disque.

Enfin, de manière régulière, on **synchronise** le contenu du cache avec le fichier en écrivant sur disque tous les blocs qui ont été modifiés (et on remet à 0 leur bit M). Dans notre cas, ceci s'effectue tous les `NSYNC` (une constante) demandes d'accès (en lecture ou écriture).

2.3 ÉTUDE DES ALGORITHMES DE REMPLACEMENT DE BLOC

L'algorithme de remplacement de bloc joue un rôle évidemment crucial dans les performances du cache.

On a indiqué dans le cours que la stratégie de remplacement optimale était connue (principe d'optimalité de Peter DENNING) : on doit remplacer le bloc qui sera utilisé à nouveau par le programme au bout du temps le plus long. Malheureusement, cette stratégie n'est pas causale et la détermination de ce bloc optimal est en fait indécidable.

On est donc conduit soit à ignorer complètement le principe d'optimalité, soit à en utiliser des approximations. De nombreux algorithmes ont été proposés et appliqués sur des systèmes réels. nous en étudierons quatre ici.

2.3.1 Remplacement de bloc au hasard (RAND)

Ici, on ignore complètement le principe d'optimalité, puisque l'on tire au sort (*random*) le bloc à remplacer. L'implémentation de cet algorithme vous est donnée à titre indicatif, mais il est clair qu'ils ne donne pas de bons résultats en général, et qu'il est très peu utilisé, voire pas du tout.

L'implémentation de trois autres algorithmes bien connus (FIFO, LRU, NUR) est votre mission pour cet exercice.

2.3.2 Remplacement du bloc le plus ancien (FIFO)

Dans l'algorithme FIFO (*First In, First Out*), on joue sur le temps de résidence d'un bloc (son âge) : on remplace le bloc le plus vieux du cache (celui qui y est monté dans le cache depuis le plus longtemps). Ici, les vieux ont donc un avenir assez compromis ! (Mais on pourra quand même aller les rechercher, si besoin, ce sera juste un peu plus long.)

Pour implémenter l'algorithme FIFO, on a besoin d'ajouter au cache une structure de données supplémentaire, une liste de pointeurs sur les blocs valides du cache. Chaque fois qu'un bloc du cache change d'affectation (soit il passe de l'état non valide à valide, soit il est affecté à un autre bloc du disque), on le transfère en queue de liste.

Le bloc le plus anciennement affecté se trouve donc en tête [1] de la liste FIFO et c'est lui qui sera remplacé.

2.3.3 Remplacement du bloc le moins récemment utilisé (LRU)

Dans l'algorithme LRU (*Least Recently Used*), on remplace le bloc qui a été le moins récemment accédé. C'est une approximation du principe d'optimalité, considérant que le passé proche est une préfiguration du futur pas trop lointain.

Comme pour FIFO, son implémentation requiert une liste de pointeurs sur les blocs valides du cache, et chaque fois qu'un bloc change d'affectation, on le transfère en queue de liste. Mais on effectue également ce transfert vers la queue chaque fois qu'un bloc est utilisé, c'est-à-dire chaque fois qu'il est atteint par une opération de lecture ou d'écriture.

Ainsi le bloc en tête de la liste LRU est-il le bloc le moins récemment utilisé, et c'est lui qui sera remplacé.

2.3.4 Remplacement d'un bloc non utilisé récemment (NUR)

C'est une variation sur l'idée de l'algorithme précédent et cela en constitue une approximation. NUR signifie *Not Used Recently* : on choisit donc un bloc qui n'a pas (de préférence, *pas du tout*) été utilisé récemment. Celui qui semble ne plus servir à rien est viré ! (Peut-être reviendra-t-il, mais, là encore, ce sera plus long.)

L'avantage de cet algorithme est sa légèreté de mise en œuvre [2]. En effet, comparé aux deux algorithmes précédents qui demandent une structure de données supplémentaire assez lourde à gérer, NUR ne demande qu'un seul bit par bloc du cache, que nous noterons R. Ce bit de référence est utilisé conjointement avec le bit de modification M déjà présent dans l'en-tête du bloc (voir 2.2) [3].

Le bit R est mis à 1 chaque fois que le bloc est référencé (utilisé en lecture ou en écriture). À intervalle régulier [4], il est remis automatiquement à 0 pour tous les blocs du cache. Donc, à un instant donné, nous avons dans le cache quatre types de blocs valides :

1. Dans cette description ainsi que dans celle de l'algorithme LRU, on peut sans dommage échanger le rôle de la tête et de la queue de la liste.
2. Tellement légère que cet algorithme est implémenté dans certaines MMU (*Memory Management Unit*), donc au niveau matériel.
3. Le bit V ne joue aucun rôle dans les algorithmes de remplacement puisque, par définition, ces algorithmes ne sont activés que lorsque tous les blocs sont valides.
4. Dans un vrai système cette remise à 0 se fait avec une période temporelle constante (plusieurs fois par seconde). Dans l'implémentation fournie ici, c'est tous les `nderef` accès, `nderef` étant un paramètre de configuration.

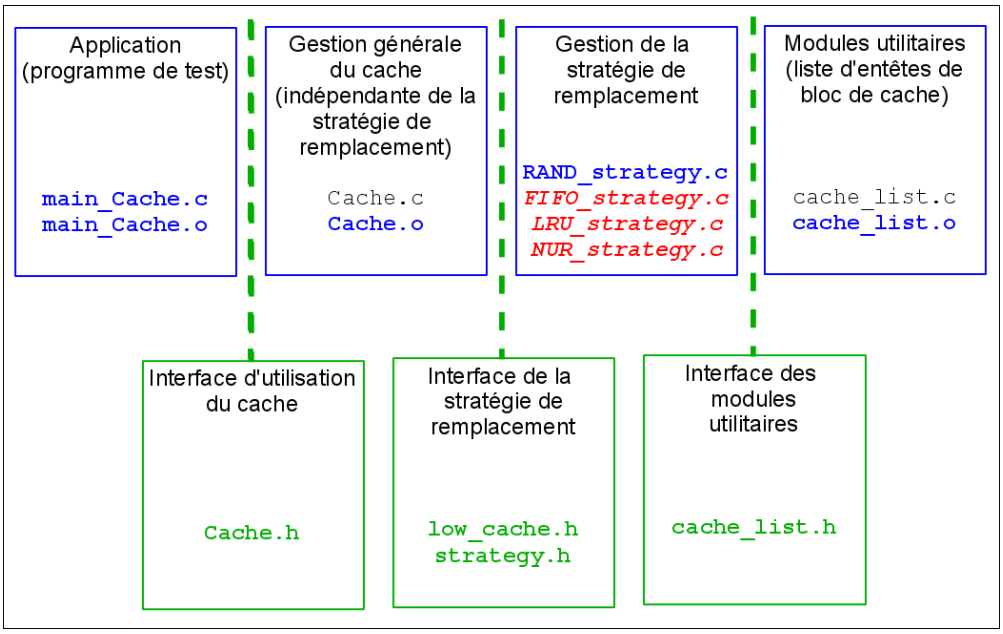

FIGURE 2.2 – Architecture générale du simulateur de cache. Les fichiers-source fournis sont en **bleu gras**, ceux à développer en ***rouge oblique***.

R	*M*	**État du bloc**
0	0	pas utilisé pendant le dernier intervalle, pas modifié
0	1	pas utilisé pendant le dernier intervalle, modifié
1	0	utilisé pendant le dernier intervalle, pas modifié
1	1	utilisé pendant le dernier intervalle, modifié

L'algorithme NUR consiste à choisir de remplacer en priorité les blocs dont le bit R est nul (ils n'ont pas été utilisé « récemment », c'est-à-dire pendant le dernier intervalle) ; dans ceux-là, on préférera ceux dont le bit M est nul, car on gagne ainsi une écriture sur disque. Évidemment, si on ne trouve pas de tels blocs, on est contraint de remplacer un des blocs avec R = 1 (toujours en donnant la priorité à ceux qui n'ont pas été modifiés) [5].

Dit de manière compacte, l'algorithme NUR consiste donc à remplacer l'un des blocs du cache qui minimise le nombre entier $n = 2R + M$.

2.4 CODE FOURNI

2.4.1 Architecture générale

Le programme de gestion du cache, fourni partiellement dans le répertoire `Cache` est organisé de manière modulaire, en quatre couches (figure 2.2).

Cette modularité permet à l'application d'ignorer les détails d'implémentation du cache, et à la gestion générale de ce dernier d'être indépendante de la stratégie de remplacement

5. Dans ce dernier cas, peu importe le bloc que l'on retourne, du moment qu'il correspond au minimum de (R, M). On pourrait évidemment tenter d'être plus sélectif, mais alors il faudrait sans doute mettre en place des structures de données supplémentaires, ce qui risquerait de réduire l'avantage de NUR, sa légèreté.

utilisée. Cette dernière propriété autorise le remplacement d'une stratégie par une autre sans modifier le reste du code. Pour renforcer cette idée, le fichier source de la gestion générale du cache (`cache.c`) n'est pas fourni (en revanche `cache.o` est fourni, ce qui vous permet de l'utiliser comme une bibliothèque). Le source du programme de test lui-même (`main_Cache.c`) est fourni à titre d'information, mais à la limite, vous pouvez faire le travail demandé sans avoir besoin de le comprendre ! Il vous suffit d'utiliser le fichier-objet correspondant, `main_Cache.o`. Enfin, la réalisation des algorithmes LRU et FIFO nécessite une liste (d'en-têtes) de bloc(s). Pour éviter que vous perdiez du temps, l'implémentation d'une telle liste vous est fournie, là encore uniquement sous la forme d'un fichier-objet `cache_list.o`.

En ce qui concerne les stratégies de remplacement, c'est à vous d'en écrire plusieurs versions sous la forme dans des fichiers de source C de noms respectifs `FIFO_strategy.c`, `LRU_strategy.c` et `NUR_strategy.c` (respectez ces noms, sinon la Makefile fournie ne fonctionnera pas). Cependant, la stratégie (rustique, il faut bien le dire) où l'on remplace un bloc au hasard vous est donnée à titre d'exemple (`RAND_strategy.c`).

Bien entendu tous les fichiers d'en-tête (`.h`) sont disponibles. Et, évidemment, vous trouverez tous les sources dans la solution !

2.4.2 Interface entre le cache et l'application (`cache.h` et `cache.o`)

Interface de base

L'interface entre l'application et le cache est définie dans le fichier cache.h, qui contient essentiellement une liste de prototypes de fonctions externes :

`struct Cache *Cache_Create(const char *fic, unsigned nblocks,`
`unsigned nrecords, size_t recordsz, unsigned nderef);`

Crée un cache associé au fichier de nom `fic`, comportant `nblocks`, chaque bloc contenant `nrecords` enregistrements de taille `recordsz` caractères. Le dernier paramètre (`nderef`) n'est utilisé que pour la stratégie NUR ; pour les autres stratégies sa valeur est ignorée. Dans le cas de NUR, le bit de référence devra être remis à 0 (pour tous les blocs du cache) tous les `nderef` accès (lecture ou écriture).

La fonction ouvre le fichier, alloue et initialise les structures de données du cache, et retourne un pointeur sur le nouveau cache.

`int Cache_Close(struct Cache *pcache);`

Détruit le cache pointé par `pcache` : synchronise le cache et le fichier grâce à `Cache_Sync()`, ferme le fichier et détruit toutes les structures de données du cache.

`int Cache_Sync(struct Cache *pcache);`

Synchronise le contenu du cache avec celui du fichier : écrit sur disque tous les blocs dont le bit M vaut 1 et remet à 0 ce bit. L'application peut appeler `Cache_Sync()` quand elle le souhaite, mais il y a un appel automatique tous les `NSYNC` accès (par défaut `NSYNC` vaut 1000, défini dans `low_cache.c`).

`int Cache_Invalidate(struct Cache *pcache);`

Invalide le cache, c'est-à-dire met à 0 le bit V de tous les blocs. C'est donc comme si le cache était vide : aucun bloc ne contient plus d'information utile.

`int Cache_Read(struct Cache *pcache, int irfile, void *precord);`

`int Cache_Write(struct Cache *pcache, int irfile, const void *precord);`
Lecture (resp. écriture) à travers le cache de l'enregistrement d'indice `irfile` dans le fichier. Le paramètre `precord` doit pointer sur un buffer fourni par l'application et au moins de taille `recordsz`. L'enregistrement sera transféré du cache dans ce buffer pour une lecture (resp. du buffer vers le cache pour une écriture).

Instrumentation du cache

Dans le fichier `cache.h` est également définie la structure `Cache_Instrument` contenant un certain nombre de compteurs permettant de collecter des statistiques sur le fonctionnement interne du cache :

```
struct Cache_Instrument
{
    unsigned n_reads;    // Nombre de lectures
    unsigned n_writes;   // Nombre d'écritures
    unsigned n_hits;     // Nombre de fois où l'enregistrement
                         // était déjà dans le cache
    unsigned n_syncs;    // Nombre d'appels à Cache_Sync()
    unsigned n_deref;    // Nombre de déréférençage (NUR)
};
```

Le plus important de ces compteurs est `n_hits`, le nombre de succès, c'est-a-dire le nombre d'accès pour lesquels l'enregistrement était déjà dans le cache. Ainsi que nous l'avons déjà mentionné, le taux de succès (*hit rate*) est le principal critère d'évaluation des performances du cache :

```
hit_rate = n_hits / (n_reads + n_writes)
```

Tous ces compteurs, sauf un, sont mis à jour par l'algorithme général de gestion du cache (dans `cache.c`), vous n'avez donc pas à vous en occuper. L'exception est `n_deref`, qui compte le nombre de fois où l'on a remis à 0 le bit R pour l'ensemble du cache, dans la stratégie NUR. Il est souhaitable que vous le gériez (cependant, ce n'est pas indispensable pour le fonctionnement de l'ensemble).

La fonction `Get_Cache_Instrument()` récupère (un pointeur sur) une copie des statistiques courantes :

`struct Cache_Instrument *Get_Cache_Instrument(struct Cache *pcache);`
Retourne une copie de la structure d'instrumentation du cache pointé par PCACHE. Attention : tous les compteurs de la structure courante sont remis à 0 par cette fonction.

2.4.3 Structure interne du cache (`low_cache.h` et `cache.o`)

Les détails internes du cache sont définis dans le fichier `low_cache.h` sous forme de deux structures C. La structure `Cache_Block_Header` défini un bloc du cache avec son en-tête et ses données :

```
struct Cache_Block_Header
{
    unsigned int flags; // Indicateurs d'état
    int ibfile;         // Index de ce bloc dans le fichier
    int ibcache;        // Index de ce bloc dans le cache
```

```
    char *data;         // Les données (nrecords enregistrements)
};
```

Les flags servent à ranger en particulier les deux bits M et V. Ceux-ci correspondent respectivement aux constantes `VALID` et `MODIF` définies dans le même fichier. Vous pouvez également utiliser ce champ flags pour y mettre le bit R correspondant à la stratégie NUR (mais n'empiétez pas sur M et V !).

Le cache lui-même correspond à la structure `Cache` et contient un tableau d'en-têtes de bloc (alloué dynamiquement et nommé `headers`), plus toutes les informations de configuration :

```
struct Cache
{
    char *file;              // Nom du fichier
    FILE *fp;                // Pointeur sur fichier
    unsigned int nblocks;    // Nb de blocs dans le cache
    unsigned int nrecords;   // Nombre d'enregistrements par bloc
    size_t recordsz;         // Taille d'un enregistrement
    size_t blocksz;          // Taille d'un bloc
    unsigned int nderef;     // Période de déréférençage pour NUR
    void *pstrategy;         // Structure de données dépendant de la stratégie
    struct Cache_Instrument instrument; // Instrumentation du cache
    struct Cache_Block_Header *pfree;   // premier bloc libre
    struct Cache_Block_Header *headers; // Les données elles-mêmes
};
```

Le champ `headers` pointe sur un tableau alloué dynamiquement d'en-têtes de bloc. Comme c'est un tableau, il suffit de l'indexer pour le parcourir (`pcache->headers[i]`).

La fonction `Get_Free_Block()` déclarée dans `low_cache.h` et implémentée dans `cache.c`, permet à la stratégie de demander un bloc libre (c'est-à-dire non valide), s'il en reste :

`struct Cache_Block_Header *Get_Free_Block(struct Cache *pcache);`
Retourne le premier bloc libre du cache ou le pointeur `NULL` si le cache est plein. Cette fonction doit être invoquée par la stratégie de remplacement avant de considérer l'utilisation d'un bloc valide.

2.4.4 Interface de la stratégie de remplacement (`strategy.h`)

Comme indiqué en 2.4.1, la stratégie de remplacement est indépendante de la gestion générale du cache. Cependant cette dernière a besoin d'une interface avec l'algorithme de remplacement. Notons que le seul rôle de l'algorithme de remplacement est de retourner un bloc utilisable par l'algorithme de gestion général (dans `cache.c`) quand celui-ci lui demande. Tout le reste (accès au fichier, entrées-sorties, gestion des bits M et V) est du ressort de la gestion générale.

L'interface est définie dans le fichier `strategy.h` et ce sont ces fonctions que vous devez réaliser, pour les trois stratégies proposées :

`void *Strategy_Create(struct Cache *pcache);`
Crée et initialise les structures de données spécifiques à la stratégie. Évidemment invoqué par `Cache_Create()`. Le pointeur retourné sera affecté (dans `Cache_Create()`) au champ `pstrategy` du cache (celui pointé par `pcache`).

`void Strategy_Close(struct Cache *pcache);`
Libère et détruit les structures de données spécifiques à la stratégie. Évidemment invoqué par `Cache_Close()`.

`void Strategy_Invalidate(struct Cache *pcache);`
Appelé lors de l'invalidation du cache pour effectuer les éventuelles action spécifiques à la stratégie. Cette fonction est appelée par `Cache_Invalidate()` après que tous les blocs du cache aient été marqués invalides.

`struct Cache_Block_Header *Strategy_Replace_Block(struct Cache *pcache);`
Retourne un bloc à remplacer. Cette fonction est évidemment invoquée lorsque l'enregistrement cherché ne se trouve pas dans le cache. Le bloc retourné doit être soit (et en priorité) un des blocs invalides, soit un bloc valide choisi en fonction de la stratégie. Pour chercher un bloc invalide, cette fonction peut utiliser `Get_Free_Block()` décrit précédemment (voir 2.4.3).

`void Strategy_Read(struct Cache *pcache, struct Cache_Block_Header *pbh);`
`void Strategy_Write(struct Cache *pcache, struct Cache_Block_Header *pbh)`
Effectue les actions spécifiques de la stratégie en cas d'écriture (resp. lecture) du bloc pointé par `pbh`. Ces fonctions sont invoquées tout à fait à la fin des fonctions de lecture et d'écriture du cache (`Cache_Read()` et `Cache_Write()`).

`char *Strategy_Name();`
Retourne une chaîne de caractères identifiant la stratégie, quelque chose comme `"LRU"` ou `"NUR"`.

2.4.5 Liste de blocs (`cache_list.h` et `cache_list.o`)

Les algorithmes de remplacement FIFO et LRU ont besoin d'une liste de blocs, en fait d'une liste de pointeurs sur en-tête de bloc. Une telle liste est définie pour vous, avec son interface d'utilisation dans `cache_list.h` :

`struct Cache_List *Cache_List_Create();`
Crée et initialise une nouvelle liste (vide) et retourne un pointeur dessus.

`void Cache_List_Delete(struct Cache_List *list);`
Détruit la liste pointée par list.

`void Cache_List_Append(struct Cache_List *list,`
`struct Cache_Block_Header *pbh);`

`void Cache_List_Prepend(struct Cache_List *list,`
`struct Cache_Block_Header *pbh);`
Crée une nouvelle cellule de liste contenant le pointeur pbh et l'insère en queue (resp. en tête) de la liste.

`struct Cache_Block_Header *Cache_List_Remove_First(struct Cache_List *list);`

`struct Cache_Block_Header *Cache_List_Remove_Last(struct Cache_List *list);`

Détruit la cellule en tête (resp. en queue) de la liste et retourne la valeur du pointeur sur en-tête qu'elle contenait. Retourne `NULL` si la liste est vide.

`struct Cache_Block_Header *Cache_List_Remove(struct Cache_List *list,`
`struct Cache_Block_Header *pbh);`
Cherche la cellule contenant le pointeur pbh et la retire de la liste. Retourne `pbh` (ou `NULL` si la liste est vide ou si elle ne contient pas `pbh`).

`void Cache_List_Clear(struct Cache_List *list);`
Détruit toutes les cellules de la liste, qui redevient donc vide.

`int Cache_List_Is_Empty(struct Cache_List *list);`
Retourne 1 si la liste est vide, 0 sinon.

`void Cache_List_Move_To_End(struct Cache_List *list,`
`struct Cache_Block_Header *pbh);`

`void Cache_List_Move_To_Begin(struct Cache_List *list,`
`struct Cache_Block_Header *pbh);`
Cherche la cellule contenant le pointeur `pbh` et la transfère en queue (resp. en tête) de liste. Si la liste ne contient pas `pbh`, ce dernier est ajouté à la fin (respectivement au début) de la liste. Si `pbh` est déjà à sa position de destination, rien ne se produit.

2.4.6 Divers (`random.h`)

Le fichier `random.h` contient la macro `RANDOM(m, n)` qui constitue la manière correcte de tirer un nombre entier au hasard dans l'intervalle `[m, n[` (`m` inclus, `n` exclus) en utilisant la fonction `rand()` de la bibliothèque C. Affichez donc **man 2 rand** si vous souhaitez des détails.

Cette macro est utilisée par certains des tests du programme principal (`main_Cache.c`) ainsi que par la stratégie de remplacement au hasard (`RAND_strategy.c`). Vous n'en avez pas vous-même besoin, en principe.

2.5 TRAVAIL À EFFECTUER

Il est temps d'y venir !

1. Copiez dans un répertoire propre [6] tous les fichiers fournis (tout le contenu du répertoire `Cache`).
2. Essayez de compiler l'application fournie avec la stratégie donnée par défaut. Il suffit de taper la commande **make** qui produit le fichier `tst_Cache_RAND` que, bien entendu, vous vous empresserez d'exécuter. Après quelques secondes (de l'ordre de deux sur ma machine), vous devez obtenir quelque chose qui ressemble à ce qui suit (**`kheops%`** étant mon prompt de **zsh**) :

```
kheops% tst_Cache_RAND
================ Configuration du cache ================
Paramètres du fichier :
    30000 enregistrements 360000 octets totaux
Paramètres du cache :
    30 blocs 10 enregistrements/bloc 12 octets/enregistrement
```

6. Doublement propre, même : vous appartenant et initialement vide !

```
    120 octets/bloc 3600 octets totaux
    Rapport cache/fichier : 1.00 %
    Stratégie : RAND
Paramètres des tests :
    Nombre d'accès : 90000
    Rapport lectures/écritures : 10
    Nombre maximum accès séquentiels : 5
    Nombre de Working Sets : 100
    Largeur de la fenêtre de localité : 300
    Fréquence de déréférençage pour NUR : 100
=========================================================
Test_1 : boucle de lecture séquentielle :
    29999 lectures 30000 écritures 56887 succès (94.8 %)
    60 syncs 0 déréférençages
Test_2 : boucle écriture aléatoire :
    0 lectures 90000 écritures 888 succès (1.0 %)
    91 syncs 0 déréférençages
Test_3 : boucle lecture/écriture aléatoire :
    81000 lectures 9000 écritures 44736 succès (49.7     91 syncs 0 dé-
référençages
Test_4 : boucle lecture/écriture aléatoire avec localité :
    80965 lectures 9035 écritures 79406 succès (88.2 %)
    91 syncs 0 déréférençages
Test_5 : boucle lecture/écriture séquentielle avec localité :
    81047 lectures 8953 écritures 79449 succès (88.3 %)
    91 syncs 0 déréférençages
kheops%
```

Le programme de simulation s'est exécuté correctement avec toutes les options par défaut (voir 2.6.2) et avec la stratégie choisie (RAND). Il affiche d'abord les paramètres de configuration puis le résultat des cinq boucles de test décrites plus loin (voir 2.6.1).

3. Écrivez les fichiers nécessaires aux différentes stratégies, `FIFO_strategy.c`, `LRU_strategy.c` et `NUR_strategy.c`. Quand vous en avez écrit un, e.g., `FIFO_strategy.c`, vous pouvez le compiler et l'exécuter par

   ```
   kheops% make tst_Cache_FIFO
   ... messages de compilation et d'édition de liens ...
   kheops% tst_Cache_FIFO
   ... résultats des 5 boucles de simulation ...
   kheops%
   ```

   Vous pouvez également éditer la Makefile fournie pour que vous n'ayez à taper que **make** pour compiler (voir le commentaire correspondant dans cette Makefile). Vous pourrez en profiter pour vérifier que les dépendances prévues dans la Makefile pour votre implémentation de la stratégie sont correctes.
4. Essayer de comparer les différentes stratégies en variant les paramètres de configuration. N'essayez pas de tout faire varier à la fois ! Essayez d'utiliser le script **plot.sh** (voir 2.6.4) dont des exemples d'utilisation se trouvent dans la Makefile (exemples

que vous pouvez exécuter grâce à la commande **make plots**). **Cette partie peut être longue et est optionnelle.**

2.6 SIMULATIONS

L'intérêt de cet exercice est de comparer les performances de la gestion de cache suivant plusieurs critères : dimensionnement du fichier et du cache, comportement local ou non des programmes, paramètres de la stratégie, etc.

À cette fin, le programme principal exécute un certain nombre de boucles de test qui visent à simuler divers comportements de programmes. Il a aussi été doté de nombreuses options qui permettent de faire varier certains des paramètre de dimensionnement ou de stratégie.

2.6.1 Boucles de test du programme de simulation

Il y a autant de programmes de test que de stratégies de remplacement, et chacun exécute consécutivement cinq boucles correspondant à différentes caractéristiques de localité (la propriété d'un programme à agréger ses références à la mémoire — ici ses références au fichier — dans la même zone).

Test 1 : boucle de lecture séquentielle La boucle parcourt complètement et séquentiellement les enregistrements du fichier en écrivant l'enregistrement courant et en lisant le précédent. Ce test possède une bonne localité (la séquentialité est un élément d'icelle) et vos taux de succès devraient être bons (de l'ordre de 95 % ou mieux, avec les paramètres par défaut).

Test 2 : boucle d'écriture aléatoire On tire au sort (uniformément) le numéro de l'enregistrement à lire. On effectue cela un certain et grand nombre de fois (`N_Loops`). La localité est évidemment mauvaise et votre taux de succès devrait être de l'ordre du rapport entre la taille du cache et celui du fichier (1 % par défaut).

Test 3 : boucle de lecture/écriture aléatoire Ce test est analogue au précédent avec cependant deux différences :

- on mélange des lectures et des écritures, en effectuant une écriture toutes les `Ratio_Read_Write` lectures (`Ratio_Read_Write` vaut par défaut 10) ;
- à chaque requête, au lieu d'accéder à un seul enregistrement, on accède à un nombre aléatoire (entre 1 et `N_Seq_Access - 1`, `N_Seq_Access` valant par défaut 5) d'enregistrements consécutifs suivant l'enregistrement courant.

Dans ce test, la localité est améliorée à cause de l'anticipation juste évoquée, mais elle n'est pas vraiment suffisante pour un fonctionnement optimal. Vos taux de succès devraient être de l'ordre de 50 % avec les paramètres par défaut.

Test 4 : boucle de lecture/écriture aléatoire avec localité améliorée Ce test vise à se rapprocher un peu du comportement local des programmes réels. Pour cela, nous découpons notre nombre d'itérations `N_Loops` en un certain nombre de phases de travail ou *Working Sets*. Nous désignons par `N_Working_Sets` ce nombre de phases (100 par défaut). À chaque phase correspondront donc un nombre d'accès `nlocal` tel que

```
nlocal = N_Loops / N_Working_Sets
```

Pour chaque phase, on tire au sort, d'abord le numéro `ind` d'un enregistrement de base, puis `nlocal` fois un incrément `incr` entre 1 et `N_Local_Window - 1` et on accède à l'enregistrement `ind + incr`. On reste donc un certain temps (pendant `nlocal` accès) dans la même zone du fichier (voisine de `ind`) ce qui améliore la localité. Cette zone, de longueur `N_Local_Window`, est appelée ici la *fenêtre de localité*.

Dans ce test aussi, le paramètre `Ratio_Read_Write` permet de décider si l'accès est une écriture ou une lecture.

Test 5 : boucle de lecture/écriture séquentielle avec localité améliorée Ce test est analogue au test précédent avec cependant deux différences qui doivent améliorer la localité :

— au lieu de tirer au sort l'enregistrement de base `ind`, on parcourt le fichier séquentiellement ;

— au lieu de lire parmi enregistrements suivants `ind`, on lit parmi ceux le précédant.

2.6.2 Argument du programme de simulation

Les exécutables de test sont nommés `tst_Cache_`*xxx*, où *xxx* désigne la stratégie de remplacement. Chacun peut être invoqué depuis le **shell** avec un certain nombre d'arguments de la ligne de commande décrits ici. Tous ces arguments ont des valeurs par défaut.

Options générales

-h affiche un message d'aide.

-p affiche les valeurs des paramètres mais n'exécute pas de test

-S format de sortie court, pour les tracés de courbe avec **plot.sh** (voir 2.6.4).

Options de configuration du cache

-f nom du fichier (défaut : `"foo"`).

-N *nrf* *nrf* est le nombre d'enregistrements dans le fichier (défaut : 30 000).

-R *nrb* *nrb* est le nombre d'enregistrements par bloc du cache (`nrecords`) (défaut : 10).

-r *rfc* *rfc* est le rapport (entier) entre la taille du fichier et la taille du cache (défaut ; 100). Donc par défaut, le cache a une taille qui est 1 % de celle du fichier.

Options des boucles de test

-t *n* valide le test de numéro *n*. Il peut y avoir plusieurs options **-t**, auquel cas seuls les tests mentionnées seront réalisés. En l'absence de cette option, tous les tests sont exécutés.

-l *v* conditionne le nombre d'itérations (d'accès) dans les tests 2 à 5 (défaut : 3). Ce nombre d'itérations, `N_Loops`, sera $nrf \times v$ (défaut : 90 000).

-w *nrw* pour les tests 3 à 5, il y aura une écriture toutes les *nrw* (`Ratio_Read_Write`) lectures (défaut : 10).

-s *nsa* *nsa* (`N_Seq_Access`) est le nombre maximum de blocs locaux à accéder à proximité du bloc de base, dans le test 3 (défaut : 5).

-W *nws* *nws* (`N_Working_Sets`) est le nombre de phases, ou *Working Sets*, dans les tests 4 et 5 (défaut : 100).

-L *nlw* *nlw* (`N_Local_Window`) est la largeur de la fenêtre de localité dans les tests 4 et 5 (défaut : 300).

-d *ndr* *ndr* (le champ `nderef` du cache, voir 2.4.3) est la période avec laquelle le bit R est remis à 0 dans la stratégie NUR (voir 2.3.4) (défaut : 100). Cette option n'a de sens que pour la stratégie NUR et elle est silencieusement ignorée pour les autres.

2.6.3 Exemples de résultats

Les fichiers `tst_Cache_`*`xxx`*`.out` du répertoire `Cache` (où *xxx* désigne la stratégie de remplacement) présentent la sortie des différents tests avec ma propre réalisation des quatre stratégies. Sauf bug de ma part, toujours possible, vos propres résultats devraient être du même ordre.

2.6.4 Tracé de courbes

Il est intéressant de comparer les différentes stratégies en faisant varier certains des paramètres, et même de tracer les courbes correspondantes. Tout d'abord, il est déconseillé de trop jouer sur la taille du fichier. Celle-ci a été choisie afin d'avoir des résultats significatifs tout en ayant des temps d'exécution acceptables sur nos machines. En revanche les autres paramètres peuvent être modifiés librement (et raisonnablement).

Cependant, vous constaterez que les résultats sont délicats à interpréter, et ceci d'autant plus qu'on fait tout bouger en même temps. Il est conseillé de ne faire varier qu'un petit nombre de paramètres à la fois (1 par exemple).

Pour faciliter le tracé de courbes de comparaison, il est fourni un script **shell**, **plot.s**h, qui permet de faire varier un parmi les paramètres mentionnés ci-dessus (2.6.2) et de tracer le taux de succès (*hit rate*) correspondant pour les quatre stratégies. Ces courbes sont en fait produites sous forme d'un fichier en format EPS (Postscript encapsulé) que l'on peut donc visualiser avec **gv** ou **ghostview**.

Ce script ne peut fonctionner que si **gnuplot** est installé et il s'utilise de la manière suivante :

```
plot.sh options -- option_test suite_valeurs...
```

Les options sont les suivantes :

-o *nom* le nom de base du fichier de courbes (son nom complet sera *nom*`.eps` ; sera également produit un fichier *nom*`.gp` de commandes à **gnuplot**.

-T *titre* titre général du graphique.

-x *étiq* étiquette de l'axe des x (la quantité que l'on fait varier).

-L *x,y* les coordonnées du titre (il est cadré à gauche) ; si ces valeurs sont incorrectes, le titre peut ne pas être affiché.

-t *n* *n* est le numéro du test (1 à 5) ; ici il ne doit y avoir qu'une seule option **`-t`**.

-i mode interactif : au lieu du fichier EPS [7], les graphiques sont affichés directement dans une fenêtre ; vous devez alors taper la commande `q` pour quitter **gnuplot**.

Le paramètre *option_test* de la ligne de commande est la désignation d'une des options décrites précédemment (2.6.2) et *suite_valeurs* la liste des valeurs que l'on veut donner au paramètres correspondant (leur ordre n'a pas d'importance, gnuplot les triera).

La Makefile contient plusieurs exemples d'invocation de **plot.sh** (que l'on peut exécuter par **make plots**). Ainsi

```
plot.sh -T "Effet de la taille du cache" \
        -o Plots/cache_size \
        -x "taille fichier/taille cache" \
        -t 5 -l "400,70" \
        -r 150 125 100 75 50 25 10
```

exécute-t-il le test 5 en faisant varier le rapport (option `-r`) entre la taille du fichier et la taille du cache de 150 à 10 (le cache varie donc de 0,67 % à 10 % de la taille du fichier). Les autres paramètres ont leurs valeurs par défaut. Le résultat est envoyé dans le fichier`Plots/cache_size.eps` et est présenté sur la figure 2.3.

Pour ce test au moins, on peut constater que, à part RAND qui n'est vraiment pas terrible (est-ce une surprise ?), les trois autres stratégies se comportent bien pour des caches de l'ordre de 1 % de la taille du fichier (`r = 100`) et au delà. En revanche, les performances s'effondrent rapidement pour des caches plus petits.

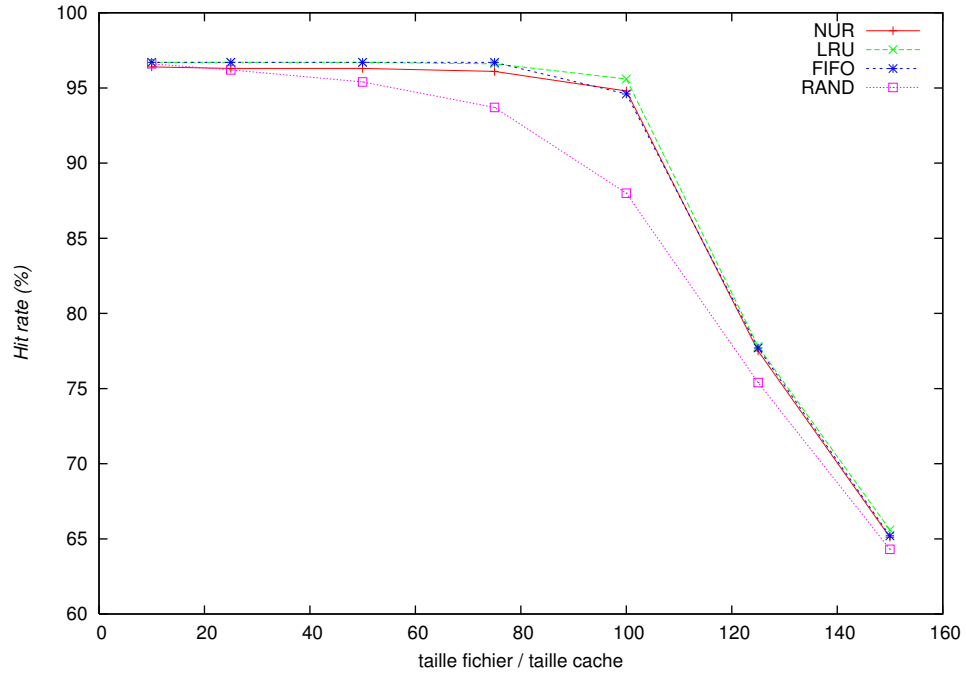

FIGURE 2.3 – Exemple de graphiques produits par le script **plot.sh** : comparaison des stratégie en fonction du rapport taille du fichier / taille du cache.

2.7 POUR TERMINER CET EXERCICE

7. En fait le fichier EPS est bien créé, mais son contenu est inexploitable. Bug ou feature de **gnuplot** ?

2.7.1 Préparation pour les expériences de simulation

Si ce n'est déjà fait :
- Terminer l'implémentation des trois stratégies demandées.
- Exécutez-les avec le programme de test fourni (`main_Cache.c`) et comparez les résultats avec les fichiers `*.out` également fournis.

Une fois tout ceci réalisé et vérifié, il est temps de commencer à utiliser le simulateur de cache ainsi obtenu. Pour cela :
- Lisez et comprenez la description des différents tests exécutés par le programme (section 2.6.1).
- Examinez la description des paramètres du programme de test, en particulier les options de test (section 2.6.2).
- Apprenez à utiliser le script **shell plot.sh** (section 2.6.4). Des exemples d'utilisation sont donnés dans la Makefile, mais vous aurez aussi à faire vos propres expériences.

2.7.2 Expériences de simulation

Vous êtes libres de tenter toutes les expériences que vous estimez pertinentes. Cependant, il ne suffit pas d'expérimenter, encore convient-il aussi d'analyser ! Vous allez donc tenter d'étudier l'influence de la variation de certains paramètres sur le taux de succès (*hit rate*) du cache.

L'étude et l'analyse des points suivants semble un minimum :
- Quel est l'effet du ratio *taille fichier/taille cache* (option **-r**) pour les différents tests ?
- Pourquoi le résultat du test 2 (avec les paramètres par défaut) donne-t-il toujours un taux de succès de 1 %, quelle que soit la stratégie ?
- Quel est l'effet du ratio *nombre de lectures/nombre d'écritures*, utilisé dans les tests 3 à 5 (option **-w**) ?
- Quel est l'effet du *nombre maximum d'itérations* dans les tests 2 à 5 ? Souvenez-vous que le paramètre correspondant (option **-l**) sera multiplié par le nombre d'enregistrements du fichiers (30 000 par défaut) pour donner le nombre total d'itérations.
- Le test 3 lit séquentiellement, à partir de l'enregistrement courant, un nombre (tiré au hasard entre 1 et `N_Seq_Access`) d'enregistrements. Quel est l'effet de ce paramètre `N_Seq_Access` (option **-s**) ?
- Quel est l'effet du nombre de *Working Sets* (option **-W**) dans les tests 4 et 5 ?
- Les tests 4 et 5 utilisent une *fenêtre de localité* (option **-L**). Quel est l'effet de ce paramètre ? Pourquoi note-t-on un effondrement du taux de succès à partir d'une largeur de fenêtre de 300 (quand tous les autres paramètres ont leur valeur par défaut) ?
- Enfin, pour approximer la notion de « récemment » dans la stratégie NUR, on remet à 0 le bit de référence (R) tous les `nderef` accès. Quel est l'effet de ce paramètre qui correspond à l'option **-d** ? Le choix de la valeur par défaut (100) vous semble-t-il judicieux ?

Tous ces points peuvent se traiter avec une utilisation raisonnable du script **plot.sh**, à l'exception du dernier. Pour voir l'effet de cette période de déréférençage (qui ne concerne que NUR) il vous faudra inventer vos propres commandes.

Formats graphique des courbes

Le script **shell plot.sh** produit des fichiers de courbes en format EPS (Postscript encapsulé). Il se peut que votre traitement de textes soit à la peine avec ce format — dont l'intérêt est la possibilité de mise à l'échelle quelconque sans perte de qualité — et que vous préféreriez du JPEG ou du PNG, par exemple. Il existe de nombreux utilitaires qui permettent de convertir ces formats graphiques l'un dans l'autre. L'un des plus efficaces et des plus simples à utiliser (en particulier sous LINUX et CYGWIN) est la commande **convert** d'ImageMagic, qui utilise l'information des extensions de nom de fichier pour faire sa conversion.

Ainsi

```
kheops% convert foo.eps foo.jpg
```

convertit un fichier EPS en un fichier JPEG. Alors que

```
kheops% convert foo.jpg foo.png
```

convertit un fichier JPEG en un fichier PNG. L'utilitaire **convert** connait pratiquement tous les formats d'image. Attention, certaines de ces conversions se font au prix d'une perte de qualité (inévitable).