TLB simple (non hiérarchique)

Mirjana Stojilovic & Jean-Cédric Chappelier 2019

L'objectif de cette semaine est double :

1. (purement pédagogique) vous faire travailler concrètement avec des listes chaînées ;
2. implémenter une première version des TLB (revoir si nécessaire le descriptif général) ;

mais le faire à un niveau d'abstraction qui devrait faciliter la création de modèles plus complexes, hiérarchiques, implémentés dès la semaine prochaine.

Il faut donc bien comprendre que pour cette raison, certains choix de conception proposés cette semaine peuvent paraître trop compliqués pour ce que nous vous demandons de faire cette semaine : il faut les comprendre dans la perspective d'une plus grande abstraction aidant à concevoir la suite. A noter également, qu'en tant que tel, le travail de cette semaine n'est pas strictement nécessaire pour la suite ; il se veut être un approfondissement pédagogique des listes chaînées et une préparation au sujet de la semaine suivante et fera à ce titre pleinement partie de la notation finale, mais ne pas le réaliser ne bloquera pas le travail des semaines restantes (revoir si nécessaire le graphe général des modules du projet donné dans le descriptif général).

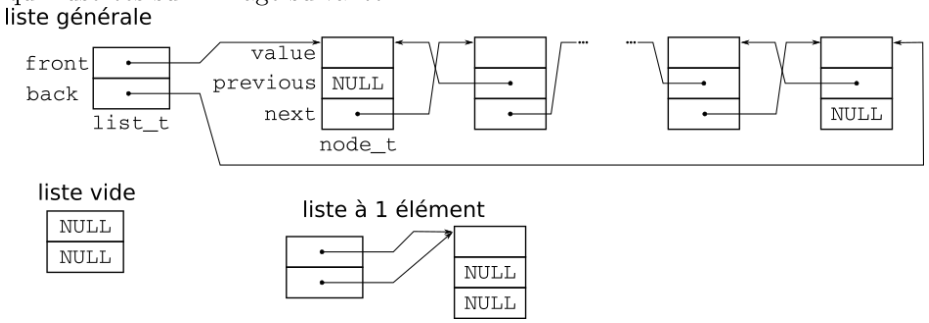
Le but de cette semaine est d'implémenter un TLB simple (non hiérarchique) utilisant une politique de remplacement que l'on peut décrire à l'aide d'une liste chaînée : LIFO, FIFO, LRU, ... (voir plus bas). Nous allons donc avoir besoin pour cela de listes chaînées. L'idée est ici (point numéro 1 ci-dessus) de les coder en tant que telles, comme une structure de données indépendante que vous pourriez utiliser par la suite (e.g. dans d'autres projets).

Ce que vous avez concrètement à faire pour cette semaine c'est de (pensez à vous répartir le travail) :

- créer et remplir le fichier `list.c` (section I ci-dessous) ;
- créer la structure `replacement_policy_t` dans `tlb_mng.h` (section II) ;
- créer la structure `tlb_entry_t` dans `tlb.h` (section III.1) ;
- créer et remplir le fichier `tlb_mng.c` (section III.2).

I. Listes doublement chaînées

Le but de cette partie est d'implémenter des listes doublement chaînées, telles qu'illustrées sur l'image suivante :



Les structures correspondant à ces listes sont définies dans `list.h`. Commencez par regarder tout le contenu de ce fichier, puis créez et complétez (et `git add`) un fichier `list.c` implémentant les fonctions décrites dans `list.h`.

Pour la politique de remplacement que nous utiliserons (LRU), outre les fonctions d'initialisation et de libération, vous n'aurez besoin que des fonctions `move_back()` et `push_back()`. Les fonctions `push_front()`, `pop_back()` et `pop_front()` ne seront pas utiles pour la suite, mais seront testées dans le test spécifique des listes (et feront partie de la notation finale ; mais elles ne remettent pas en cause la suite du projet si elles ne sont pas implémentées).

En plus des explications fournies dans `list.h`, voici quelques informations supplémentaires pour vous aider dans leur réalisation :

- une liste vide est uniquement définie par le fait d'avoir ses deux pointeurs `back` et `front` tous le deux `NULL` (cf illustration ci-dessus) ;
- `init_list()` doit simplement initialiser la liste reçue à la liste vide (elle ne doit pas faire l'allocation dynamique de cette liste en tant que telle. On pourrait en effet tout à fait l'utiliser comme suit :

  ```
  list_t ma_liste;
  init_list(&ma_liste);
  ```

)
- `clear_list()`, par contre, **libère** tous les chaînons de la liste, puis la rend vide ;
- comme toujours, nous vous conseillons de modulariser votre code.

Pour l'affichage d'une liste, nous vous demandons de respecter le format suivant :

```
(98, 2608, 809, 3, 247)
```

Notez que la macro `print_node()` est fournie dans `list.h`.

II. Politiques de remplacement

Une politique de remplacement est un moyen de décider quelle cellule remplacer dans une mémoire cache/TLB lorsque celle ci est pleine et qu'une nouvelle valeur doit y être mémoriser. Il existe un grand nombre de politiques de remplacement (voir cette page Wikipedia), mais plusieurs peuvent utiliser une liste chaînée comme structure de données pour gérer cette politique.

C'est ce point de vue que nous avons voulu prendre pour ce projet : même si nous n'allons finalement n'implémenter que la politique LRU dans nos tests, nous souhaitons en principe pouvoir représenter n'importe quelle politique de remplacement à base de liste chaînée.

Une telle politique de remplacement peut alors être représentée par :

- la liste chaînée qu'elle utilise pour gérer l'ordre des priorités ;
- une fonction qui définit comment ajouter un nouvel élément à la liste ;
- une fonction qui définit comment déplacer un élément de la liste à un bout de celle-ci ; cette fonction sera appliquée à chaque élément accédé dans le TLB .

Par exemple, en prenant la convention qu'en cas de TLB pleine, ce soit toujours l'élément en tête de liste qui soit remplacé alors :

- si la fonction d'ajout ajoute toujours en tête de liste et la fonction qui déplace l'élément accédé ne fait rien, on a alors simplement une politique LIFO (une pile) : le dernier élément *ajouté* (et non pas *accédé*) sera le premier supprimé ;
- si la fonction d'ajout ajoute toujours en fin de liste et la fonction qui déplace l'élément accédé ne fait rien, on a alors simplement une politique FIFO (une file) : le premier élément *ajouté* (et non pas *accédé*) sera le premier supprimé ;
- si la fonction d'ajout ajoute toujours en fin de liste et la fonction qui déplace l'élément accédé le déplace aussi en fin de liste, on a alors une politique LRU (*least recently used*) : l'élément non manipulé (accès ou ajout) depuis le plus longtemps se retrouve en effet bien en tête de liste.

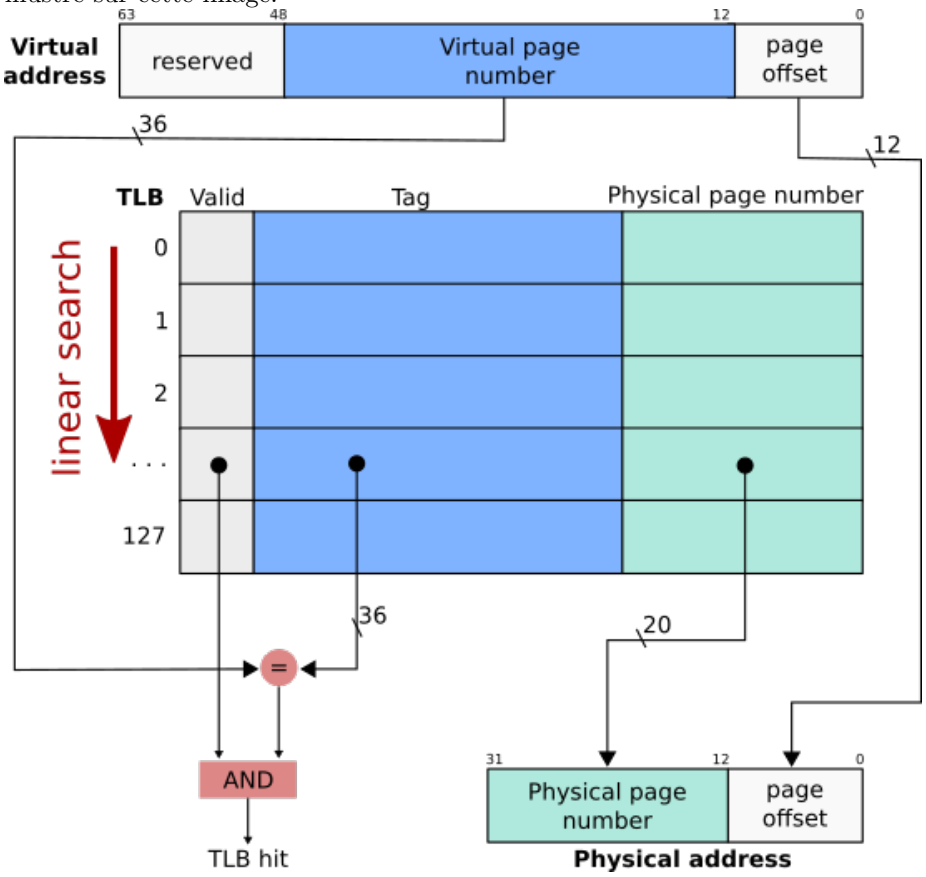
Concrètement, dans le fichier `tlb_mng.h`, définissez le type `replacement_policy_t` regroupant :

- un champ `ll`, pointeur sur une liste chaînée ;
- un champ `push_back`, pointeur sur une fonction d'ajout ayant même prototype que la fonction `push_back()` des listes doublement chaînées ;
- un champ `move_back`, pointeur sur une fonction de déplacement ayant même prototype que la fonction `move_back()` des listes doublement chaînées.

III. TLB

III.1 Type `tlb_entry_t`

Le TLB considéré ici est « *fully associative* », c.-à-d. que les pages virtuelles seront entrées dans ce TLB par ordre d'arrivée jusqu'à ce qu'il soit plein, comme illustré sur cette image:



Une fois plein, c'est la politique de remplacement (cf section précédente) qui s'applique.

Dans le fichier `tlb.h`, définissez le type `tlb_entry_t` comme un bitfield ayant :

- `VIRT_PAGE_NUM` bits, nommés `tag` ; on lira la valeur de ces bits au travers d'un `uint64_t` ;
- `PHY_PAGE_NUM` bits, nommés `phy_page_num` ; on lira la valeur de ces bits au travers d'un `uint32_t` ;
- 1 bit de validation, nommé simplement `v` ; on lira la valeur de ce bit au travers d'un `uint8_t`.

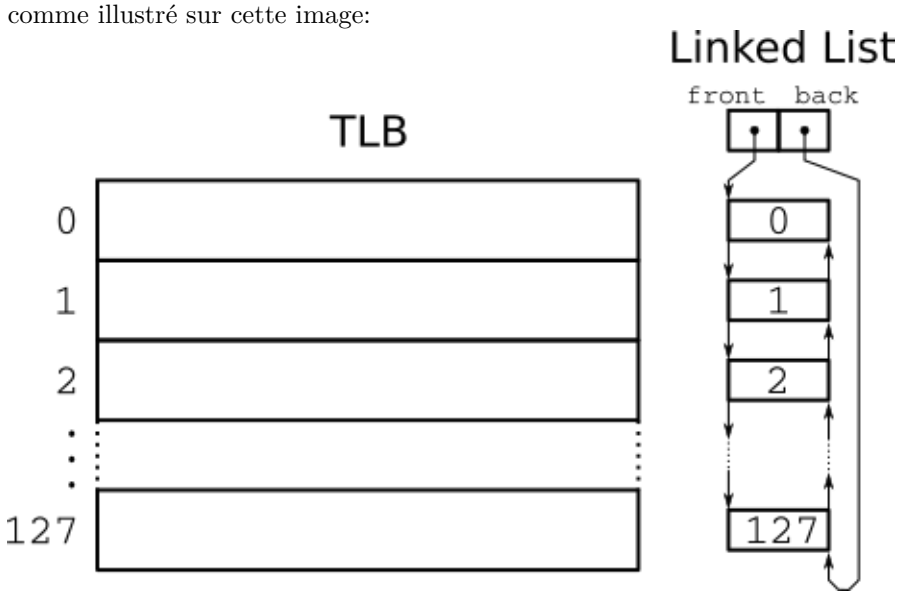
III.2 Implémentation du « fully associative TLB »

Un TLB « fully associative » sera simplement un tableau de `TLB_LINES` `tlb_entry_t` ; p.ex. :

```
tlb_entry_t tlb[TLB_LINES];
```

Nous n'avons donc pas défini de type particulier pour cela.

Pour sa politique de remplacement, il devra être accompagné d'une liste (doublement) chaînée, initialisée avec des valeurs croissantes de `0` à `TLB_LINES - 1`, comme illustré sur cette image:



ce qui correspondrait par exemple au code suivant :

```
tlb_entry_t tlb[TLB_LINES];
tlb_flush(tlb);
list_t ll;
init_list(&ll);
for (list_content_t line_index = 0; line_index < TLB_LINES; ++line_index) {
    (void)push_back(&ll, &line_index);
}
```

Vous n'avez pas à faire explicitement cette construction/initialisation dans les fichiers C du cœur du projet, mais aurez uniquement à la faire à chaque fois que vous souhaiterez utiliser un tel TLB, typiquement dans vos tests (nous ferons de même de notre coté dans nos propres tests).

Le type `tlb_entry_t` étant défini, l'implémentation d'un TLB « fully associative » consiste alors en la définition de fonctions permettant :

- d'initialiser une `tlb_entry_t` (`tlb_entry_init()`) ;
- d'initialiser un TLB (`tlb_flush()`) ;

- d'insérer une adresse dans le TLB en suivant sa politique de replacement (`tlb_insert()`) ;
- de vérifier si une adresse est déjà dans le TLB (`tlb_hit()`) ;
- (la fonction de plus haut niveau) de demander au TLB une traduction d'adresse virtuelle (`tlb_search()`).

Commencez par regarder le fichier `tlb_mng.h` pour une description de ces fonctions, puis créez et complétez (et `git add`) un fichier `tlb_mng.c` implémentant ces fonctions.

III.2.1 `tlb_entry_init()`

Mis à part les vérifications d'usage, cette fonction doit simplement initialiser le `tlb_entry_t` fourni en troisième argument avec les deux informations fournies. Il faut bien sûr extraire, sur 64 bits, le « *virtual page number* » de l'adresse virtuelle fournie en premier argument.

Le bit de validité de la `tlb_entry_t` sera mis à 1.

III.2.2 `tlb_flush()`

Cette fonction met simplement à 0 tout le contenu du TLB reçu (on suppose ici que c'est un tableau de `TLB_LINES` `tlb_entry_t`).

III.2.3 `tlb_insert()`

Cette fonction met simplement à jour l'entrée du TLB située à l'index `line_index` (1er argument) à partir des information stockées dans le `tlb_entry_t` fourni en second argument.

III.2.4 `tlb_hit()`

Contrairement à la majorité de nos fonctions, cette fonction ne retourne pas de code d'erreur, mais simplement 0 si l'adresse virtuelle n'est pas déjà dans le TLB et 1 si elle l'est. On retournera donc également 0 dès qu'il y a le moindre problème (puisqu'on aura alors pas trouvé l'adresse virtuelle recherchée).

Pour rechercher l'adresse virtuelle fournie, il faut bien sûr commencer par extraire, sur 64 bits, le « *virtual page number* » de l'adresse virtuelle fournie en premier argument (cf image ci-dessus). Puis ensuite on cherche ce numéro de page dans le TLB en le parcourant dans l'ordre inverse des index stockés dans la liste chaînée associée. Autrement dit : on parcourt cette liste chaînée de sa fin vers son début et on vérifie si le « *virtual page number* » recherché se trouve dans le TBL à l'index correspondant stocké dans la liste.

Note : on aurait bien sûr pu faire que la liste chaînée contiennent les valeurs elles-mêmes, mais nous avons tenu à découpler les deux en gardant d'un coté un tableau (comme un vrai TLB et comme ce sera encore le cas par la suite) et de l'autre une liste chaînée d'entiers (qui ne sait donc rien de la notion de

`tlb_entry_t` ou autre) ; une telle liste chaînée peut ainsi indexer n'importe quoi (un troisième choix aurait été d'en faire une liste chaînée de pointeurs génériques (`void*`)).

[**fin de note**]

Si le numéro de page recherché est trouvé dans le TLB et qu'il est valide (champ `v`), alors :

- on affecte l'adresse physique (second argument de la fonction) avec les valeurs correspondante (cf image ci-dessus) ;
- on déplace le nœud de la liste correspondant en « fin » de liste (appel de la méthode `move_back()` de la politique de remplacement) ;
- et on retourne 1.

III.2.5 `tlb_search()`

Cette fonction gère de bout en bout la recherche d'une adresse physique à partir d'une adresse virtuelle :

- on commence par voir si l'adresse recherchée est dans le TLB (« *hit* », fonction précédente, laquelle affecte également l'adresse physique – cf ci-dessus) ;
- si elle n'y est pas (« *miss* »), alors :
  - on appelle le « *page walker* » pour traduire l'adresse virtuelle ;
  - puis (s'il n'y a pas d'erreur) on initialise un `tlb_entry_t` avec les valeurs obtenues ;
  - puis on l'insère dans le TLB à l'index stocké en *tête* de liste chaînée ;
  - et on déplace cette tête en « fin » de liste (appel de la méthode `move_back()` de la politique de remplacement).

III.3 Tests

Comme toujours, nous vous conseillons de créer vos propres tests et de tester systématiquement et progressivement votre code. Pour vous aider dans cette tâche, nous vous fournissons :

- des tests de feedback sur les listes chaînées ;
- des tests de feedback sur les TLB ;
- 1 outil de test des TLB et 1 scénario de test associé.

Pour les feedback, vous les obtenez en faisant

```
make feedback
```

après avoir récupéré la dernière version de l'image docker fournie à cet effet. Si vous n'êtes pas sur les VM des CO (lesquelles récupèrent à chaque fois l'image !!), récupérez cette dernière version en faisant (dans un terminal) :

```
docker pull arashpz/feedback
```

Pour l'outil de test, celui-ci peut être simplement compilé et utilisé en local, soit directement avec vos propres scénarios (expliqué plus bas) soit en faisant

```
make check
```

qui teste le scénario que nous vous fournissons (expliqué ci-dessous).

L'outil en question est `test-tlb_simple.c`. A recopier, donc, depuis `provided/` dans `done/` puis à ajouter à votre `Makefile`. Il utilise trois arguments :

- un fichier de commandes tels qu'utilisés en semaine 5 et dont vous avez des exemples dans `tests/files/commands01.txt` et `tests/files/commands02.txt` ;
- un fichier binaire image d'un contenu mémoire, tel que vous en avez un dans `tests/files/memory-dump-01.mem` (revoir si nécessaire les exemples de tests fournis pour les deux dernières semaines) ;
- un nom de fichier pour écrire le résultat ;

par exemple :

```
./test-tlb_simple tests/files/commands02.txt tests/files/memory-dump-01.mem resultat.txt
```

Note : cet exemple est en fait exactement ce que fait `make check` (qui lance `tests/08.basic.sh` qui exécute exactement la commande ci-dessus et vérifie le résultat). Le résultat attendu est fourni dans `tests/files/output/tlb-simple-01-out.txt`.

Vous pouvez bien sûr écrire vos propres scénarios d'accès à des adresses virtuelles (comme le fichier `tests/files/commands02.txt`) afin de vérifier le fonctionnement de votre TLB. Nous vous encourageons même à partager de tels scénarios de tests, par exemple dans un fil de discussion dédié sur le forum Moodle, en fournissant le fichier de commandes et son fichier résultat attendu.

IV. Rendu

Concernant le rendu, le travail de cette semaine ne sera pas évalué en tant de tel (c.-à-d. seul), mais devra faire partie du second rendu intermédiaire du projet (délai : le dimanche 12 mai 23:59). Néanmoins, nous vous conseillons toujours de travailler régulièrement et faire des (tests et des) commits réguliers.