Adresses physiques et virtuelles

Mirjana Stojilovic & Jean-Cédric Chappelier 2019

Le but du travail de cette semaine est :

1. de prendre pleine connaissance du cadre et des concepts du projet ;
2. et de créer les structures de données relatives aux adresses physiques et virtuelles.

Ce que vous avez concrètement à faire cette semaine est de :

1. lire le fichier de description principal du projet afin de bien comprendre le cadre général ; n'y manquez pas le diagramme général d'organisation du travail réalisé afin de vous aider à organiser au mieux votre travail dans le groupe et sur le semestre ;
2. définir 5 types dans le fichier `addr.h` ;
3. définir 7 fonctions dans un fichier `addr_mng.c` ;
4. compléter le `Makefile` fourni ;
5. ajouter vos propres tests dans `test-addr.c`.

Commencez donc par le point 1 (lecture).

I. Implémentation des types adresse physique et adresse virtuelle

Avant tout, commencez par recopier la totalité du répertoire `provided` dans `done` :

```
cp -r provided done
```

C'est en effet dans le répartoire `done` qu'il vous faut travailler et, comme pour le premier devoir : vous ne devez **jamais** modifier le contenu du répertoire `provided` de votre dépôt GitHub !

Pour avoir une idée du travail à faire, allez voir (et lisez) le contenu de `addr.h` et `addr_mng.h`. Puis définissez à l'endroit indiqué dans `addr.h` :

- le type `word_t` correspondant à un entier positif (= non signé) sur 32 bits ;
- le type `byte_t` correspondant à un entier positif sur 8 bits ;
- le type `pte_t` correspondant à un entier positif sur 32 bits ;

- le type `virt_addr_t` correspondant à un « *bitfield* » (revoir si nécessaire le travail de la semaine passée) comprenant :
  - `VIRT_ADDR_RES` bits réservés (appelés `reserved`) ;
  - `PGD_ENTRY` bits (appelés `pgd_entry`) servant d'index dans le PGD ;
  - `PUD_ENTRY` bits (appelés `pud_entry`) servant d'index dans le PUD ;
  - `PMD_ENTRY` bits (appelés `pmd_entry`) servant d'index dans le PMD ;
  - `PTE_ENTRY` bits (appelés `pte_entry`) servant d'index dans la PT ;
  - `PAGE_OFFSET` bits (appelés `page_offset`) servant d'index dans la page de mémoire physique ;

  on accédera à chacune de ces parties du « *bitfield* » au travers d'entiers positifs sur 16 bits ; les macros mentionnées ci-dessus (`VIRT_ADDR_RES`, `PGD_ENTRY`, etc.) sont définies dans `addr.h` ;
- et enfin le type `phy_addr_t` correspondant à un « *bitfield* » comprenant :
  - `PHY_PAGE_NUM` bits (appelés `phy_page_num`) représentant l'adresse d'un début de page de mémoire physique divisée par la capacité d'une page (4Kio), c.-a.-d. les 20 bits de poids fort de l'adresse, décalé vers le poids faibles;
  - `PAGE_OFFSET` bits (appelés `page_offset`) servant d'index dans la page en question ;

  on accédera à `phy_page_num` au travers d'un entier positif sur 32 bits, et à `page_offset` au travers d'un entier positif sur 16 bits.

Note : nous prenons la convention de nommer tous les types suivant le schéma `X_t`. Pour les types comme des structures ou des types énumérés, il s'agit donc d'alias (vers la `struct` ou l'`enum`). Par exemple le type `virt_addr_t` est en fait un alias vers le `struct` « *bitfield* » correspondant.

II. Définitions des fonctions outils pour adresses physiques et adresses virtuelles

Il faut maintenant définir les sept fonctions décrites dans `addr_mng.h`. Revoyez si nécessaire ce fichier.

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire. Et n'oubliez pas de tester correctement les fonctionnalités d'une étape **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas… A ce sujet (tests), voyez la section III.

Fonction `print_virtual_address()`

Dans le fichier `addr_mng.c` (à créer), définissez la fonction `print_virtual_address()` conformément à son prototype donné dans `addr_mng.h`. Nous vous conseillons de commencer par écrire la fonction `print_virtual_address()`, parce que :

1. elle ne présente pas vraiment de difficulté ;
2. et surtout elle vous permettra de facilement tester toutes les fonctions qui manipulent une structure « adresse virtuelle », simplement en affichant son contenu.

Cette fonction doit afficher, dans le flot `where` passé en argument, les différents champs d'une `virt_addr_t`, excepté le champ `reserved`, en utilisant **strictement** le format illustré par l'exemple suivant :

```
PGD=0x0; PUD=0x0; PMD=0x0; PTE=0x0; offset=0x1C
```

Pour écrire un `uint16_t` en hexadécimal (avec lettres majuscules), utilisez la macro `PRIX16` définie dans `inttypes.h`. Cette macro se met **hors** de la chaîne de caractères, c.-à-d. **hors** des double-guillemets ; p.ex. :

```
printf("valeur = 0x%" PRIX16 " et la suite du texte ici...", value);
```

La fonction `print_virtual_address()` devra retourner le nombre de caractères affichés. **Cela se fait très simplement !** (allez voir la valeur de retour de `printf()`).

Fonction `init_virt_addr()`

Toujours dans le fichier `addr_mng.c`, définissez la fonction `init_virt_addr()` conformément à son prototype donné dans `addr_mng.h`. Cette fonction doit :

- tester la validité de ses arguments et retourner un code d'erreur s'ils ne sont pas corrects ; utilisez pour cela les macros `M_REQUIRE` fournies dans `erreur.h` (aller voir) ; c'est à vous d'imaginer les tests à effectuer (il n'y en a pas beaucoup) ; en cas de mauvais paramètre, la fonction retourne le code d'erreur `ERR_BAD_PARAMETER` ;

  cet aspect des fonctions (tester la validité de leurs arguments) devra être systématique et ne sera plus rappelé par la suite (ce devrait être un automatisme de programmeur) ; les codes d'erreurs possibles à utiliser sont donnés dans `error.h` (et voir si nécessaire `error.c` pour leur message d'erreur associé) ;

- affecter les différents arguments reçus aux parties correspondantes de l'adresse virtuelle à initialiser ; la partie réservée de l'adresse virtuelle sera systématiquement nulle dans tout ce projet (mais dans la réalité, elle est effectivement utilisée pour passer diverses informations que nous laisserons de coté dans ce projet) ;

- retourner le code `ERR_NONE` (= « pas d'erreur ») en fin de travail.

Pour rappel, pour accéder à un champ `x` d'une structure `s` passée par référence (`s` est alors un pointeur sur la structure), il faut utiliser la syntaxe `s->x`.

Fonction `init_virt_addr64`

Dans le fichier `addr_mng.c`, définissez la fonction `init_virt_addr64()` conformément à son prototype donné dans `addr_mng.h`. Cette fonction doit récupérer chacune des parties suivant le schéma suivant :

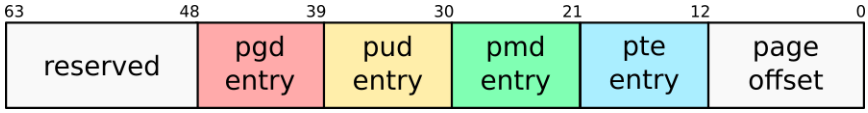

Figure 1: 64 bits de l'adresse virtuelle

Inspirez vous pour cela ce que vous avez fait la semaine passée.

Pensez ensuite à tester votre code, p.ex. avec des valeurs bien choisies. Utilisez pour cela l'écriture hexadécimale en C, p.ex. :

```c
uint64_t test = 0x0; // Bon, on peut aussi ici écrire simplement 0
test = 0x1;          // Bon, on peut aussi ici écrire simplement 1
test = 0x1000;       // 1er bit de PTE à 1, tous les autres à 0
test = 0x200000;     // 1er bit de PMD à 1
test = 0x40000000;   // 1er bit de PUD à 1
test = 0x8000000000; // 1er bit de PGD à 1
// ... etc.
```

Fonction `virt_addr_t_to_virtual_page_number()`

Dans le fichier `addr_mng.c`, définissez la fonction `virt_addr_t_to_virtual_page_number()` conformément à son prototype donné dans `addr_mng.h`. Cette fonction est presque la réciproque de `init_virt_addr64()` sauf qu'elle ignore les information d'offset de page et « *reserved* » : elle ne convertit donc que les champs `X_entry` d'une structure d'adresse virtuelle en leur écriture sur 64 bits comme illustré ci-dessus, mais décalé de `PAGE_OFFSET` bits vers la droite (il n'y a **pas** lieu de faire ce décalage : nous voulons simplement dire que l'**illustration** ci-dessus est décalée de tous ses bits d'offset qui ne sont pas présents) :

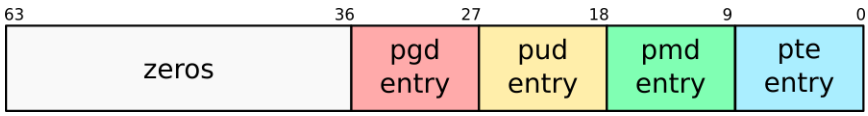

Figure 2: partie principale (Virtual Page Number) de l'adresse virtuelle

Concrètement, la fonction `virt_addr_t_to_virtual_page_number()` doit donc simplement affecter les différents bits suivant le schéma ci-dessus. Pour faire cela, inspirez vous à nouveau de ce que vous avez fait la semaine passée.

Fonction `virt_addr_t_to_uint64_t()`

Dans le fichier `addr_mng.c`, définissez la fonction `virt_addr_t_to_uint64_t()` conformément à son prototype donné dans `addr_mng.h`. Cette fonction est la réciproque de `init_virt_addr64()` : elle convertit une structure d'adresse virtuelle en son écriture sur 64 bits. Concrètement, elle doit simplement décaler le résultat de `virt_addr_t_to_virtual_page_number()` et ajouter les bits de l'offset de page.

Fonction `print_physical_address()`

Dans le fichier `addr_mng.c`, définissez la fonction `print_physical_address()` conformément à son prototype donné dans `addr_mng.h`. Cette fonction doit afficher, dans le flot `where` passé en argument, les deux champs d'une `phy_addr_t`, en utilisant **strictement** le format illustré par l'exemple suivant :

```
page num=0xDD; offset=0x1C
```

Pour écrire un `uint32_t` en hexadécimal (avec lettres majuscules), utilisez la macro `PRIX32` de façon similaire à `PRIX16` utilisée précédemment.

La fonction `print_physical_address()` devra retourner le nombre de caractères affichés. [Cela se fait toujours très simplement !]

Fonction `init_phy_addr()`

Dans le fichier `addr_mng.c`, définissez la fonction `init_phy_addr()` conformément à son prototype donné dans `addr_mng.h`. Cette fonction doit simplement :

1. récupérer les (32 - `PAGE_OFFSET`) bits de poids forts de `page_begin` pour les mettre en bits de poids faibles dans le champs `phy_page_num` (p.ex. avec `PAGE_OFFSET` à 12, le 12-ièem bit sera mis en bit 0, le 13-ième en bit 1, etc.) :
2. affecter `page_offset` au champ `page_offset`.

III. Tests

Vos propres tests

Nous vous fournissons un fichier `test-addr.c` que nous vous conseillons fortement d'éditer pour y ajouter vos propres tests : supprimer les deux lignes de `puts`, et inspirez vous de l'exemple donné pour ajouter tous les tests que vous jugez nécessaire. Les principales fonctions de test disponibles dans l'environnement que nous utilisons (Check) sont décrites là-bas : https://libcheck.github.io/check/doc/check_html/check_4.html#Convenience-Test-Functions. Par exemple, pour tester si deux `int` sont égaux, utilisez alors la « fonction » `ck_assert_int_eq` : `ck_assert_int_eq(a, b)`.

Nous avons également défini les « fonctions » suivantes dans `test.h` :

- `ck_assert_err_none(int erreur)` : teste si l'erreur `erreur` est `ERR_NONE` (c.-à-d. correspond à un retour de fonction sans erreur ; voir `erreur.h`) ;
- `ck_assert_bad_param(int erreur)` : teste si l'erreur `erreur` est `ERR_BAD_PARAMETER` (c.-à-d. correspond à une erreur d'une fonction ayant reçu un mauvais paramètre ; voir `erreur.h`) ;
- `ck_assert_ptr_nonnull(void* pointeur)` : teste si le pointeur `pointeur` n'est pas `NULL` ;
- `ck_assert_ptr_null(void* pointeur)` : teste si le pointeur `pointeur` est `NULL`.

Pour faire l'édition de lien avec la bibliothèque Check, il faut ajouter les options `-lcheck -lm -lrt -pthread` à l'édition de liens ; par exemple :

```
gcc test-hashtable.o hashtable.o error.o -lcheck -lm -lrt -pthread -o test-hashtable
```

Sur certaines architectures, il faut aussi ajouter la bibliothèque `-lsubunit`. N'oubliez pas de mettre à jour votre `Makefile` si nécessaire.

Les tests que nous vous fournissons comme feedback

Afin de vous donner une idée de l'état de votre programme, nous vous fournissons *certains* (pas tous) des scenarii que nous utiliserons pour évaluer votre code (lequel sera également lu par des correcteurs humains). Pour profiter de ce feedback, tapez simplement (dans `done/`) :

```
make feedback
```

Sachez profiter de ce compte-rendu qui est là pour vous aider.

IV. Organisation du travail

REMARQUE IMPORTANTE : en raison de vos connaissances en C (synchronisation avec le cours), le travail de cette semaine-ci est assez léger ; celui de la semaine prochaine (on attend les pointeurs !) sera par contre *très conséquent* : ce sera toute la mise en place des entrées/sorties fichiers nécessaires aux tests du projet.

Mais vous pouvez néanmoins commencer sa *mise en place* sans plus attendre. Pour cela (vous aider à équilibrer la charge de travail), nous avons déjà mis en ligne le sujet de la semaine prochaine). Nous vous conseillons de le commencer gentiment en laissant si nécessaire de coté les *quelques* points qui nécessiteraient des pointeurs. L'essentiel peut déjà être mis en place dès cette semaine.

Libre à vous, donc, de vous organiser au mieux dans votre travail suivant vos contraintes. Et pensez à vous répartir correctement la tâche entre les deux membres du groupe.

A ce sujet (charge de travail), si vous ne l'avez pas encore lue, nous vous conseillons la lecture de la page expliquant le barème du cours (ici en HTML et ici en PDF).

V. Rendu

Vous n'avez pas à rendre de suite le travail de cette première semaine de projet, celui-ci ne sera à rendre qu'à la fin de la semaine 8 (délai : le dimanche 14 avril 23:59) en même tant que le travail des semaines 5 à 7.

Ceci dit, nous vous conseillons de marquer par un commit lorsque vous pensez avoir terminé le travail correspondant à cette semaine :

1. ajoutez les quatre fichiers suivants à un répertoire `done/` de votre dépôt GitHub **de groupe** (c.-à-d. correspondant au projet) : `Makefile`, `addr.h`, `addr_mng.c` et `test-addr.c` ;
2. faites le commit (vérifiez bien que tout est ok) :

   ```
   git commit -m "version finale week04"
   ```

Nous vous conseillons en effet fortement de travailler régulièrement et faire systématiquement ces commits réguliers, au moins hebdomadaires, lorsque votre travail est opérationnel. Cela vous aidera vous-mêmes à mesurer votre progression.

Et n'oubliez pas de faire le rendu (individuel, depuis votre dépôt *personnel*) de la semaine passée avant ce dimanche soir.