Amélioration code passé et décodage d'adresse virtuelle

Mirjana Stojilovic & Jean-Cédric Chappelier 2019

Cette semaine et la semaine prochaine vous allez : 1. rendre dynamique les « programmes » (= listes de commandes) de la semaine passée ; 2. coder le « *page-walker* », conversion d'adresses virtuelles en adresses physiques ; 3. être capable de le tester avec les exemples d'images mémoire fournies (coder deux fonctions de lecture de fichier).

Cette partie étant plus conséquente, nous avons décidé de l'étaler sur deux semaines : ce sujet sert donc pour cette semaine et la suivante. De plus, l'état du projet à la fin de ce sujet (fin de semaine prochaine donc) correspond au *premier rendu noté* de ce projet. Il sera à rendre (détails en fin de ce sujet) au plus tard le jeudi 18 avril.

I. Rendre le type `program_t` dynamique

Connaissant maintenant l'allocation dynamique suite au cours de lundi (et à quelques exercices que nous vous recommandons vivement de faire avant), il est temps d'améliorer le type `program_t` :

- dans `commands.h`, supprimez l'allocation statique et faites ce qui est nécessaire pour prévoir de l'allocation dynamique ;
- dans `commands.c`, modifier :
  - `program_init()`, pour allouer dynamiquement 10 `command_t` ; il peut être utile d'utiliser la macro `M_EXIT_IF_NULL` définie dans `error.h` (allez voir) ;
  - `program_add_command()`, gérer l'ajout *dynamique* d'une nouvelle commande ; afin d'être efficace en terme de nombre de réallocations, cette fonction devra à chaque fois que c'est nécessaire, réallouer le **double** de la taille déjà allouée (c'est p.ex. ainsi que fonctionnent les `vector` en C++) : au départ (on a dit ci-dessus) on a alloué la place pour 10 `command_t`, lors de l'ajout de la 11e commande, il faudra réallouer de la place (totale) pour 20 ; et lors de l'ajout de la 21e, de la place pour 40, etc.
- et remplir `program_shrink()` (toujours dans `commands.c`) : il s'agit ici de réadapter la taille d'un `program_t` à la taille minimale requise pour garder

son contenu (qui n'est pas nécessairement un multiple de 10 ; faites au plus simple : juste la taille effectivement utilisée) ; au cas où le programme est vide (pas de commande, `nb_lines` à 0) alors il faut au moins garder la place pour 10 `command_t` comme fait lors de l'initialisation).

II. Page Walk

II.1 Description générale

Le but de cette partie est de traduire les adresses virtuelles en adresses physiques, c.-à-d. de faire la correspondance entre une adresse virtuelle et l'endroit où elle est réellement présente en mémoire (physique). Vous allez pour cela écrire, dans un fichier `page_walk.c` à créer, la fonction `page_walk()` dont le prototype est donné dans `page_walk.h`. Comme d'habitude, commencez par aller voir ce prototype et lire sa description.

Le principe de conversion est résumé sur cette figure :

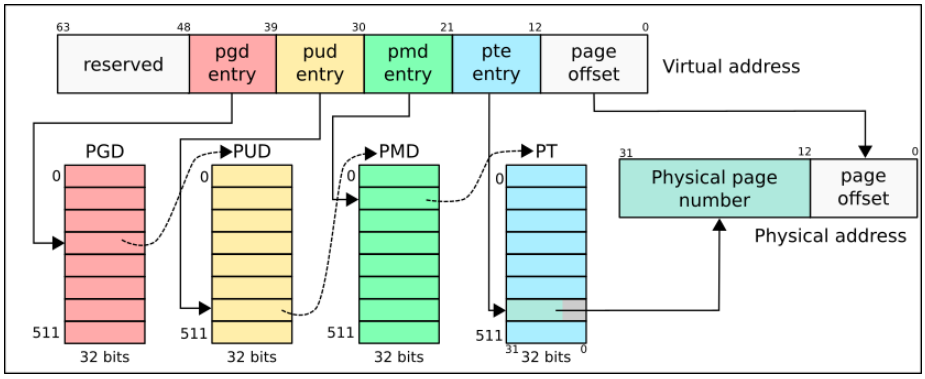


Figure 1: Conversion de de l'adresse virtuelle en adresse physique

déjà illustré dans le descriptif général (n'hésitez pas à aller revoir ce document si nécessaire) : on va, quatre fois (PGD, PUD, PMD et PTE), rechercher une adresse dans une page (tableau), cette adresse indiquant le début de la page dans laquelle faire la recherche suivante. Plus précisément :

- les 9 bits de « *pgd entry* » de l'adresse virtuelle indiquent un **mot** (`pte_t` en fait) dans le « tableau » PGD (qui, rappelons-le, est par convention au tout début de la mémoire, adresse 0) ;
- ce mot (de 32 bits) est lui-même l'adresse (en **octets**) du tableau suivant (PUD ici) ;
- dont les 9 bits correspondants de l'adresse virtuelle (« *pgd entry* ») indiquent un **mot** dans ce « tableau » ;
- ce mot est lui-même l'adresse (en **octets**) du tableau suivant (PMD) ;

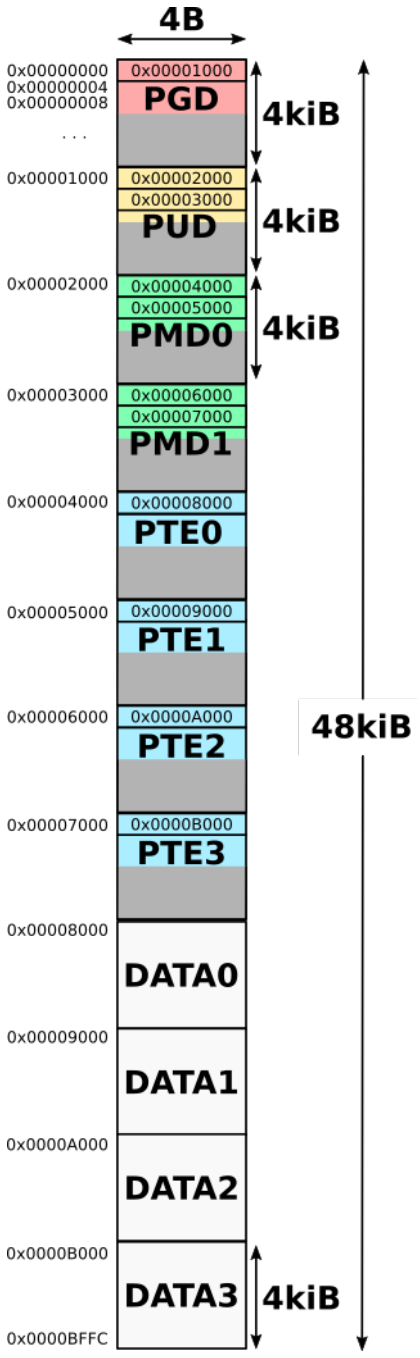
- et ainsi de suite, jusqu'à arriver au contenu du PTE dont les 20 bits les plus hauts sont le numéro de page physique (« *physical_page_number* »), c.-à-d. les 20 bits de poids forts de l'adresse physique.

Les 12 bits de poids faibles de l'adresse physique sont simplement les mêmes (partie « *offset* ») que ceux de l'adresse virtuelle.

Rappel : pour créer une adresse physique, vous aviez fait la fonction `init_phy_addr()`.

II.2 Exemple de traduction d'adresse virtuelle en adresse physique

Par exemple, avec le schéma mémoire suivant (utilisé pour la partie test, voir plus loin ; si nécessaire, cette image est accessible en plus haute résolution en suivant ce lien) :



l'adresse virtuelle `0x0000000040200010` correspond à la 16e valeur de la page « DATA3 ». En effet :

- `0x0000000040200010` se décompose en :

  - pgd_entry = `000000000` (en bits), c.-à.-d. 0 (c'est normal, dans cet exemple il n'y a qu'une seule entrée dans le PGD) ;
  - pud_entry = `000000001`, c.-à.-d. 1 ;
  - pmd_entry = `000000001`, c.-à.-d. 1 ;
  - pt_entry = `000000000`, c.-à.-d. 0 ;
  - page_offset = `000000010000`, c.-à.-d. 16 ;
- on cherche donc le premier (index 0) mot de la PGD, qui est `0x00001000` ; donc la PUD correspondante est à l'adresse `0x00001000` (en octets), soit au 4096e octets, c.-à-d. au 1024e mot (si l'on regarde la mémoire comme un tableau de mots plutôt que comme un tableau d'octet, typiquement dans la fonction `read_page_entry()`) ;
- puis on cherche ensuite le second (index 1) mot de cette PUD ; lequel est `0x00003000` ; donc la PMD correspondante est à l'adresse `0x00003000` (en octets) ;
- puis on cherche le second (index 1) mot de cette PMD ; lequel est `0x00007000` ; donc la PTE correspondante est à l'adresse `0x00007000` (en octets) ;
- puis on cherche le premier (index 0) mot de cette PTE ; lequel est `0x0000B000` ; donc le début de page physique (appelé aussi « numéro de page ») correspondant est `0x0000B000` ;
- et l'offset étant 16, c'est bien au 16e octets de cette page que l'on s'intéresse ; et l'adresse physique correspondante sera `0x0000B010`.

II.3 Fonction outil pour accéder à une information dans une page de traduction d'adresse

Pour implémenter ces recherches d'informations (adresses) dans des pages (tableaux), nous vous conseillons de définir une fonction outil supplémentaire comme ceci :

```
static inline pte_t read_page_entry(const pte_t * start,
                                    pte_t page_start,
                                    uint16_t index)
{ return start[i]; }
```

où :

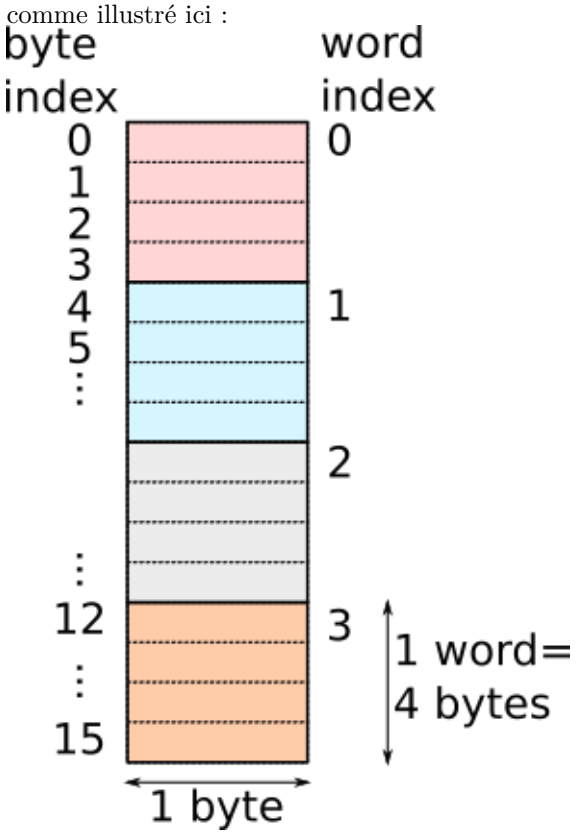
- `start` est simplement l'adresse du début de (toute) la mémoire utilisée ;
- `page_start` est l'index du début de la page que l'on souhaite lire ;
- `index` est l'index, dans cette page, de l'information que l'on recherche.

Avec l'exemple précédent, et en supposant que (l'adresse du début de) la zone mémoire totale est stockée dans le pointeur `main_memory`, alors par exemple la recherche dans la PMD du début de la PTE serait :

```
read_page_entry(main_memory, 0x3000, 1);
```

lequel appel retournerait `0x7000`.

Cette fonction retourne donc simplement la valeur (lue comme un `pte_t`) à l'index « `i` » du tableau `start`. Bien sûr, tout votre travail est de trouver la bonne expression pour `i` en fonction de `page_start` et `index`. Nous voudrions pour cela insister sur un point indiqué plus haut : notez bien que l'indexation dans une page se fait bien en termes de *mot* (`pte_t`), mais que les adresses utilisées pour indiquer les débuts de page(/tableau) sont elles en *octets*, c.-à.-d. qu'elles sont `sizeof(pte_t)` fois plus grandes que si l'on indexait avec des *mots*, comme illustré ici :



III. Tests

Pour vous permettre de tester votre implémentation (et toutes celles futures), nous vous avons constitué deux exemples de contenu mémoire qui sont fournis dans le répertoire `tests/files` et sont décrit plus bas. Mais pour pouvoir lire ces exemples, il vous faut écrire deux fonctions décrites dans `memory.h` :

```
int mem_init_from_dumpfile(const char* filename, void** memory,
                           size_t* mem_capacity_in_bytes);

int mem_init_from_description(const char* master_filename, void** memory,
```

```
                                    size_t* mem_capacity_in_bytes);
```

III.1 `mem_init_from_dumpfile()`

L'idée est de vous donner un dump complet de toute la mémoire sous forme d'un fichier binaire. Le but de cette fonction est donc simplement de lire ce fichier en mémoire.

Pour allouer la bonne taille à la mémoire à lire, il faut être capable de déterminer la taille du fichier. Comme cela n'a pas été détaillé en cours, nous vous donnons ici le code correspondant : pour un fichier `file` correctement ouvert :

```
// va tout au bout du fichier
fseek(file, 0L, SEEK_END);

// indique la position, et donc la taille (en octets)
mem_capacity_in_bytes = (size_t) ftell(file);

// revient au début du fichier (pour le lire par la suite)
rewind(file);
```

Le reste ne devrait poser aucune difficulté (si ce n'est, comme d'habitude, de bien gérer tous les cas d'erreur envisageables ; utilisez le code d'erreur `ERR_IO` en cas de problème avec le fichier et `ERR_MEM` en cas d'erreur mémoire).

III.2 `mem_init_from_description()`

L'idée est ici de vous donner la description du contenu d'une mémoire sous la forme de plusieurs pages décrites globalement dans un fichier texte principale et donnée ensuite chacune sous forme d'un fichier binaire. Un exemple du fichier texte principal est fourni dans `tests/files/memory-desc-01.txt`.

Le format général de ce fichier que l'on peut résumer comme ceci :

```
TOTAL MEMORY SIZE (size_t)
PGD PAGE FILENAME
NUMBER OF TRANSLATION PAGES (PUD+PMD+PTE)
LIST OF TRANSLATION PAGES as INDEX OFFSET    (uint32_t) and FILENAME
LIST OF DATA PAGES        as VIRTUAL ADDRESS (uint64_t) and FILENAME
```

est le suivant :

- la première ligne contient la taille totale de la mémoire (à lire via `"%zu"`) ; à noter que toute la mémoire n'est pas nécessairement remplie, seules les parties spécifiées ensuite (ci-dessous) sont utilisées, mais on allouera toute cette taille pour garantir le bon fonctionnement ;
- la seconde contient le nom du fichier binaire contenant la PGD (il n'y a qu'une seule PGD) ; ce fichier a forcément une taille de 4 kio (même si une PGD n'occupe que la moitié, afin de faciliter la modularisation

(voir fonction conseillée `page_file_read()` ci-dessous) nous avons décidé de donner toutes ces informations sous forme d'une page mémoire, donc 4096 octets) ;
- la troisième ligne contient le nombre de « *translation pages* » (PUD, PDM, PTE) à lire ;
- suivent ensuite autant de ligne (que ce nombre), chacune indiquant :
  - une adresse physique (nombre d'octets par rapport au début de la mémoire) où mettre la page correspondante ;
  - le nom du fichier binaire contenant la page correspondante ;
- la fin du fichier est ensuite constitué d'une liste de pages de données à charger en mémoire, chacune indiquant :
  - l'adresse **virtuelle** où placer cette page ;
  - le nom du fichier binaire contenant la page correspondante.

Pour écrire la fonction `mem_init_from_description()`, nous vous conseillons de procéder étape par étape, en particulier nous vous recommandons d'écrire une fonction `page_file_read()`, avec les arguments qui vous arrangent, permettant de lire et mettre à une adresse physique donnée le contenu d'un fichier binaire.

III.3 Scénarios de tests fournis

Comme évoqué plus haut, nous vous fournissons deux scénarios de tests (contenus de mémoire) :

- l'un sous les deux formes mentionnées ci-dessus : dump binaire complet de la mémoire, et fichier de description avec fichier binaires des pages ;
- et l'autre uniquement sous formes de sa description en page.

Le premier exemple est assez simple et décrit par l'image mémoire donnée plus haut. Il vous permettra de tester votre « *page walker* », ainsi que la fonction `mem_init_from_description()` (en comparant les deux versions). Sa version « dump complet » est donnée dans `tests/files/memory-dump-01.mem`. Sa version « description » est donnée dans `tests/files/memory-desc-01.txt`.

Le second exemple est plus complexe en soi et nous ne donnons pas d'autre information que son fichier description `tests/files/memory-desc-02.txt` (et le binaire de ses 17 pages mémoire). Signalons simplement que cet exemple a deux entrées dans son PGD. Pour mieux le comprendre, nous vous encourageons à en faire un dessin mémoire similaire à celui que nous avons fourni plus haut pour le premier exemple. Bien sûr cela suppose avoir déjà testé ses fonctions sur le premier exemple (y compris la fonction `mem_init_from_description()`).

IV. Rendu

Le code à ce stade (y compris les semaines passées depuis la semaine 4) constitue le premier rendu de la partie projet. Il est à rendre avant le **jeudi 18 avril 23:59**. Pour le rendre, le plus simple est de faire

`make submit1`

Avant la soumission, vérifiez avoir bien ajouté (`git add`), validé (`git commit`) et transmis (`git push`) toutes vos dernières versions de tous vos fichiers sources `.c` et `.h`, ainsi que le `Makefile`. Merci par contre de ne pas ajouter les fichiers `.o` ni les exécutables.

Avant de soumettre, veuillez également retirer (ou commenter) tous les appels à `printf()` superflus que vous auriez pu ajouter. Nous vous conseillons d'ailleurs d'utiliser plutôt le flux d'erreur `stderr` (`fprintf(stderr,`) car nous ne testons pas son contenu.

Ce qui sera considéré comme rendu sera ce que l'on trouvera dans (la branche `master` de) votre dépôt à la date indiquée ci-dessus et marqué d'un tag `projet01_NB`. C'est ce que fait la commande

`make submit1`

La raison pour laquelle nous étiquetons (`git tag`) votre contenu est pour vous permettre de continuer à travailler et prendre de l'avance : ainsi si votre dépôt contient à la date de rendu une version en avance sur le rendu et qui n'est pas fonctionnelle, ce n'est pas grave, nous ne prendrons que la dernière version pour laquelle vous aurez fait

`make submit1`

Ne faites donc pas de « `make submit1` » sur une version qui ne compile pas...