NMV TP1 : Gestion de la mémoire virtuelle

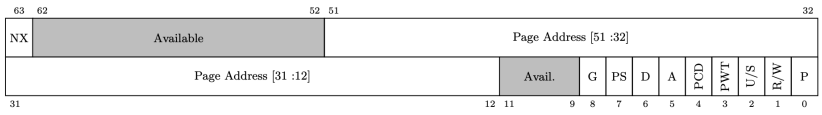
Exercice 1

1. Pourquoi un niveau intermédiaire de la table des pages est mappé par une identité (adresse virtuelle égale à son adresse physique)

Pour effectuer un **page walk**, on a besoin des adresses physiques pour pouvoir charger la prochaine page pour la traduction, il est donc obligatoire d'avoir les adresses virtuelles directement égales aux adresses physiques.

2. Combien d'entrée par niveau de table de pages pour architecture x86 64 bits, comment savoir si une page est valide / terminale

Entrée de table des pages :



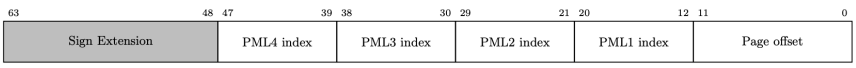
P	l'entrée est valide
R/W	la page est accessible en lecture/écriture
U/S	la page est accessible en mode utilisateur
PWT	les données écrites dans la page ne sont pas mises en cache
PCD	les données de la page sont lues depuis la RAM
A	mis à 1 par le processeur quand la page est accédée
D	mis à 1 par le processeur quand la page est modifiée
PS	la page pointée est une huge page
G	la page reste dans la TLB quand le `CR3` est modifié
NX	la page est inaccessible en exécution

Une page fait **4 kiB**, une adresse fait **8 B**, donc on peut mettre **512** adresses par page. On a **64 bits** par entrée, on utilise **48 bits** pour adresser virtuellement, dans cette adresse, on a **16 bits** d'**extension de signe**. Dans les **32 bits** non-utilisé, on met des **metadata**.

- **bit P**: si la page est valide, sinon raise un **page fault**.
- **bit PS**: si on doit continuer au prochain niveau d'indirection, si il est à **1** c'est qu'on a une **huge page**.

Exercice 2

1. Masques pour trouver l'index de l'entrée correspondant au niveau courant



D'après la structure de l'adresse, on sait que les 16 premiers bits sont pour l'extension de signe. Puis on a 9 bits par niveau, et on sait qu'on a que 4 niveaux. On re-shift a droite pour récupérer la valeur de l'index.

- level 4: `0xff8000000000 >> 39`
- level 3: `0x7fc0000000 >> 30`
- level 2: `0x3fe00000 >> 21`
- level 1: `0x1ff000 >> 12`

Exercice 3

1. Pourquoi un page fault à 0x2000000030

Les tâches partent du principe que les adresses en user sont déja mappées de manière implicite. Ici, l'adresse n'étant pas mappée le noyau lors du chargement de la tâche, le système lève un **page fault** lors de la lecture du code de la tâche.

```
 * +----------------------+ 0xffffffffffffffff
 * | Higher half          |
 * | (unused)             |
 * +----------------------+ 0xffff800000000000
 * | (impossible address) |
 * +----------------------+ 0x00007fffffffffff
 * | User                 |
 * | (text + data + heap) |
 * +----------------------+ 0x2000000000
 * | User                 |
 * | (stack)              |
 * +----------------------+ 0x40000000
 * | Kernel               |
 * +----------------------+ 0x0
```

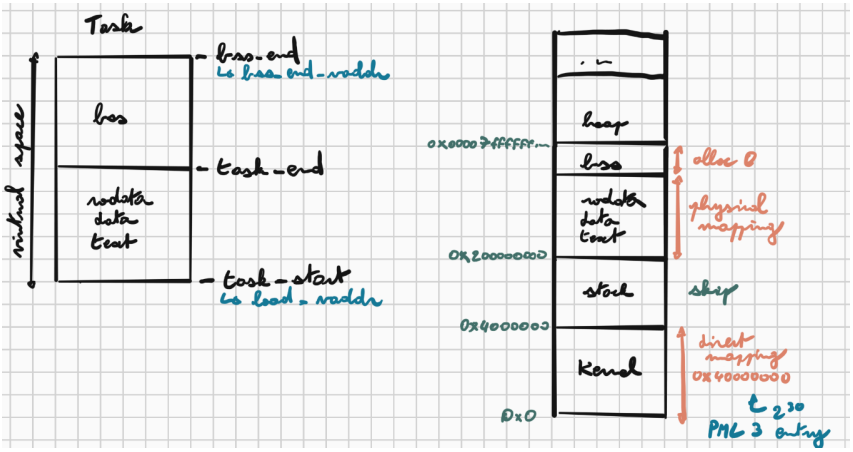
2. Création d'une tâche, plage d'adresse dans la nouvelle table

On partage l'espace kernel entre tout les processus. Il faut donc faire attention aux accès concurrents sur cette zone de mémoire partagée (utilisation de lock).

On sait que l'espace du kernel occupe l'espace entre `0x0` et `0x40000000`. Donc, c'est équivalent à seulement une entrée dans le PML3 (equivaut **30 bits**, donc 2^{30}). Il faut donc alloué un **PML4**, puis un **PML3** dans **PML4[0]**, puis l'entrée **PML3[0]** doit être égale à l'adresse du **PML2** correspondant à l'espace kernel (récupérable dans le **PML3** du père).

Il faut ensuite mapper le payload (le code et les données) en fonction des adresses physiques de la tâche. Puis il faut mapper le reste dans une zone initialisé à 0 (correspond au bss).

- `load_paddr` - `load_end_paddr`: adresses du payload dans l'espace physique.
- `load_vaddr + load_end_paddr - load_paddr` - `bss_end_vaddr`: addresses virtuelles restantes, correspond à la zone bss, doit être initialisée à 0.



Exercice 4

1. Faute de page à l'adresse `0x1fffffff8`

D'après le modèle mémoire, les adresses entre `0x40000000` et `0x2000000000` sont les adresses de la pile de la tâche courante. Par défaut, les adresses de la pile ne sont pas mappées, ce qui explique la faute de page. Ici, l'adresse est bien contenue dans le segment de la pile, c'est donc bien un accès légitime.

2. Allocation paresseuse de la pile

D'après le modèle mémoire, la pile d'une tâche fait une taille de `127 GB`, ce qui ne rentre pas dans la mémoire physique. C'est pour cela qu'elle est allouée physiquement seulement de manière paresseuse lors d'une faute de page.

En cas de faute de page:
- Vérification que l'adresse correspond au segment pour la pile de la tâche.
- Si c'est bien le cas, allocation et mapping de la page nécessaire.
- Sinon, erreur de segmentation puis on termine la tâche courante.

Exerice 5

2. Echec de "Adversary"

Dans la tâche "Adversary":
- Lecture / écriture dans la pile à l'adresse `0x1fffff3000` sur toute la page.
- Appel. à **unmap** pour l'adresse `0x1fffff3000`
- Lecture dans la pile à l'adresse `0x1fffff3000` sur toute la page.

Comme on appel **unmap**, normalement, lors de la seconde lecture, on devrait **page fault**, ce qui devrait nous alloué une page avec données à 0. Or, ici la tâche arrive quand même à lire les données écrites avant **unmap**.

C'est du au **TLB (translation look-aside buffer)** qui n'est pas invalidé lorsque l'on désalloue la page, signifiant que la tâche bypass la table des pages vu que l'entrée dans le TLB est encore valide.

Pour palier à cela, on utilise **invlpg** qui permet d'invalider le TLB.