NMV: Gestion de la mémoire virtuelle

L'objectif de ce TP est de programmer la gestion de la MMU dans un mini système d'exploitation : "Rackdoll". Ce système est volontairement très simple pour vous permettre d'y tester facilement votre code. Il s'agit néanmoins d'un véritable système d'exploitation 64 bits et doit être exécuté sur une machine, physique ou virtuelle.
Après une rapide phase de *boot*, Rackdoll charge des tâches en mode utilisateur et les exécute en temps partagé. Ces exécutions concurrentes nécessitent l'utilisation de mémoire virtuelle dont vous devez fournir le code de gestion.

Pour compiler Rackdoll, rendez-vous dans la racine du projet et entrez la commande `make`. Pour lancer Rackdoll dans une machine virtuelle Qemu, entrez la commande `make boot`. Cette commande ouvrira une fenêtre vers l'écran de la machine virtuelle à côté de votre terminal. Le terminal quant à lui basculera en mode moniteur d'où vous pouvez tapper les commandes suivantes :

info registers Affiche l'état les principaux registres de la machine.

xp/8g *addr* Affiche le contenu de la mémoire à l'adresse virtuelle *addr*.

Gardez ces commandes en tête, elles constituent de précieux outils de debug.

Une annexe est située à la fin de cet énnoncé qui contient une figure décrivant la structure d'une entrée de la table des pages, une figure décrivant la décomposition d'une adresse virtuelle ainsi qu'une figure décrivant la structure de la table des pages initial de Rackdoll.

Exercice 1

Ce premier exercice vous permet de vous familiariser avec la structure d'une table des pages. On vous propose d'implémenter une fonction qui affiche la structure de la table des pages courante. Cette fonction a le prototype : `void print_pgt(paddr_t pml, uint8_t lvl)`. Cette fonction recursive prend comme paramètre l'adresse physique d'un niveau de la table des pages et la hauteur de ce niveau.

Question 1
Dans le cadre de ce TP on supposera toujours qu'un niveau intermédiaire de la table des pages est mappé par une identité (il est accessible par une adresse virtuelle égale à son adresse physique). Pourquoi cette supposition est-elle importante ?

Question 2
Combien y a-t-il d'entrée par niveau de table des pages pour une architecture x86 64 bits ? Comment la MMU détermine-t-elle si une entrée d'un niveau de table des pages est valide ? Si elle est terminale ?

Question 3
Dans `kernel/main.c`, programmez et testez la nouvelle fonction `print_pgt`. Vous testerez votre nouvelle fonction en l'appellant depuis la fonction `main_multiboot2` située dans le fichier `kernel/main.c` juste avant le chargement des tâches utilisateur. Pour cela vous pouvez utiliser la fonction `uint64_t store_cr3(void)` qui retourne le contenu du registre `CR3`.

Si votre fonction d'affichage fonctionne correctement, vous devriez observer une table des pages avec la structure indiquée dans l'annexe de ce sujet.

Exercice 2

Dans cet exercice on implémente la fonction qui permet de mapper une adresse virtuelle sur une adresse physique. Par soucis de simplicité, on fera les suppositions suivantes :

- Les niveaux intermédiaires de la table des pages sont tous mappés par une identité (l'adresse virtuelle est égale à l'adresse physique).
- Tous les nouveaux mappings seront faits avec les droits utilisateurs (bit 2) et en écriture (bit 1).
- On ne cherche pas à gérer les *huge pages*.
- Si un mapping est demandé pour une adresse virtuelle α, l'adresse virtuelle α n'est pas déjà mappée.

Vous avez à votre disposition la fonction **`paddr_t`** `alloc_page()` qui alloue une nouvelle page déjà mappée par une identité. Le contenu de cette page est indéfini.

Question 1
Étant donné une adresse virtuelle `vaddr` à mapper et la hauteur courante dans la table des pages `lvl` (avec `lvl = 4` pour le niveau indiqué dans le `CR3`), donnez la formule qui indique l'index de l'entrée correspondante dans le niveau courant.

Question 2
Implémentez la fonction **`void`** `map_page(`**`struct task`** `*ctx,` **`vaddr_t`** `vaddr,` **`paddr_t`** `paddr)` dans le fichier `kernel/memory.c` qui mappe l'adresse virtuelle `vaddr` sur l'adresse physique `paddr` sur un espace d'une page pour la tâche `ctx`. L'adresse physique du premier niveau de la table des pages est indiquée par `ctx->pgt`.

Pour cet exercice, n'hésitez pas à tester très régulièrement votre fonction à chaque étape de son implémentation. On pourra pour cela utiliser le morceau de code suivant depuis `kernel/main.c` :

```
struct task fake;
paddr_t new;

fake.pgt = store_cr3();
new = alloc_page();

map_page(&fake, 0x201000, new);
```

Si votre fonction `map_page` fonctionne, vous devriez pouvoir inspecter la mémoire à l'adresse `0x201000` à l'aide du moniteur de Qemu avec la commande **`xp/8g 0x201000`**.

Exercice 3

Dans cet exercice on implémente les fonctions nécessaires au chargement d'une tâche en mémoire. Le fichier `kernel/memory.c` contient, en commentaire, la description du modèle mémoire de Rackdoll. Le modèle mémoire d'un système définit à quelles adresses physiques et virtuelles sont placées chacun des composants.

Question 1
Á cette étape du TP, l'exécution de Rackdoll doit afficher sur le moniteur qu'une faute de page se produit à l'adresse virtuelle `0x2000000030`. Étant donné le modèle mémoire, indiquez ce qui provoque la faute de page.

Question 2
La première étape de la création d'une nouvelle tâche en mémoire est de dériver la table des pages courante en une nouvelle table des pages. Expliquez quelles plages d'adresses de la table courante doivent être conservées dans la nouvelle table et pourquoi.

Une tâche utilisateur `ctx` est constituée de deux parties :
- Le *payload* situé dans la mémoire physique entre `ctx->load_paddr` et `ctx->load_end_paddr` qui contient le code et les données.
- Le *bss* qui doit commencer en espace virtuel immédiatement après le *payload* et s'arrêter à l'adresse virtuelle `ctx->bss_end_vaddr`.

On rappelle que le *bss* est une zone qui doit être initialisée à zero au lancement d'un tâche. Il est possible que certaines tâches aient un bss vide.

Question 3
Donnez les adresses virtuelles de début et de fin du *payload* et du *bss* d'une tâche, calculées en fonction du modèle mémoire et des champs d'une tâche `ctx`.

Question 4
Implémentez la fonction **`void`** `load_task(`**`struct task`** `*ctx)` qui initialise une nouvelle tâche en mémoire sans toutefois charger sa table des pages dans le `CR3`.

Question 5
Implémentez la fonction **`void`** `set_task(`**`struct task`** `*ctx)` qui charge une nouvelle tâche en mémoire en modifiant le `CR3`.

Exercice 4

Dans cet exercice on implémente les fonctions d'allocation.

Question 1
Implémentez la fonction **`void`** `mmap(`**`struct task`** `*ctx,` **`vaddr_t`** `vaddr)` qui alloue une page physique, l'initialise à zero et la mappe à l'adresse virtuelle donnée pour la tâche donnée.

Question 2
Á cette étape du TP, l'exécution de Rackdoll doit afficher sur le moniteur qu'une faute de page se produit à l'adresse virtuelle `0x1fffffff8`. Étant donné le modèle mémoire, indiquez ce qui provoque la faute de page. D'après vous, cette faute est-elle causée par un accès mémoire légitime ?

Question 3
D'après le modèle mémoire de Rackdoll, la pile d'une tâche utilisateur a une taille de 127 GiB, c'est à dire bien plus que la mémoire physique disponible dans la machine virtuelle. La pile est donc allouée de manière paresseuse. Expliquez en quoi consiste l'allocation paresseuse.

Question 4
Implémentez la fonction **`void`** `pgfault(`**`struct interrupt_context`** `*ctx)` qui traite une faute de page dont le contexte est stocké dans `ctx` et où l'adresse qui a causé la faute est stockée dans le registre `CR2` accessible via la fonction **`uint64_t`** `store_cr2(`**`void`**`)`. Rappellez-vous que les seules fautes de page légitimes sont celles de la pile. Toute faute à une adresse en dehors de la pile doit causer une faute de segmentation de la tâche courante (vous pouvez utiliser la fonction **`void`** `exit_task(`**`struct interrupt_context`** `*ctx)` qui termine la tâche courante).

Exercice 5

Dans cet exercice on implémente la fonction de libération. Á cette étape du TP, les tâches doivent s'exécuter brièvement avant d'échouer. Des messages d'avertissement indiquant une pénurie mémoire doivent aussi s'afficher sur le moniteur. Cette pénurie est causée par la tâche "Sieve" qui fait de nombreux appels système `mmap` et `munmap`. Puisque `munmap` n'est pas encore implémenté, aucune page n'est libérée : c'est une fuite mémoire.

Question 1
Implémentez la fonction **`void`** `munmap(`**`struct task`** `*ctx,` **`vaddr_t`** `vaddr)` qui permet de supprimer le mapping d'une adresse virtuelle donnée pour une tâche donnée. Cette fonction doit aussi libérer les pages mémoire qui ne sont plus utilisées à l'aide de la fonction **`void`** `free_page(`**`paddr_t`** `addr)`.

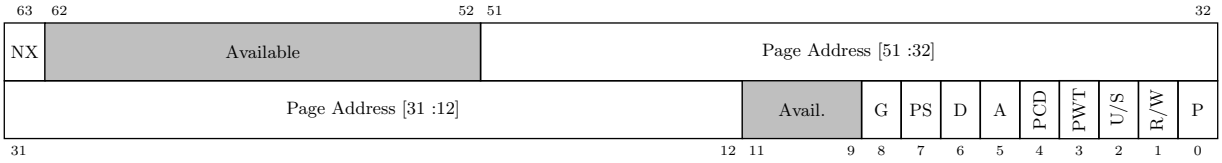
Une fois la fonction `munmap` implémentée, la tâche Sieve ne devrait plus causer de pénurie mémoire et toutes les tâches devraient pouvoir s'exécuter complètement.

Question 2
Il est possible que malgré une exécution complète, la tâche "Adversary" indique un echec. En lisant le code de cette tâche dans `task/adversary.c` et en relisant le code de votre fonction `munmap`, indiquez ce qui peut provoquer cet echec.

Question 3
Corrigez le problème soulevé dans la Question 2 à l'aide d'une fonction définie dans le fichier `include/x86.h`.

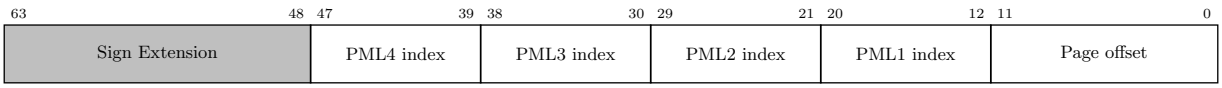
Annexe

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

Décomposition d'une adresse virtuelle :



Structure de la table des pages initiale de Rackdoll (initialisée dans `entry.S`) :