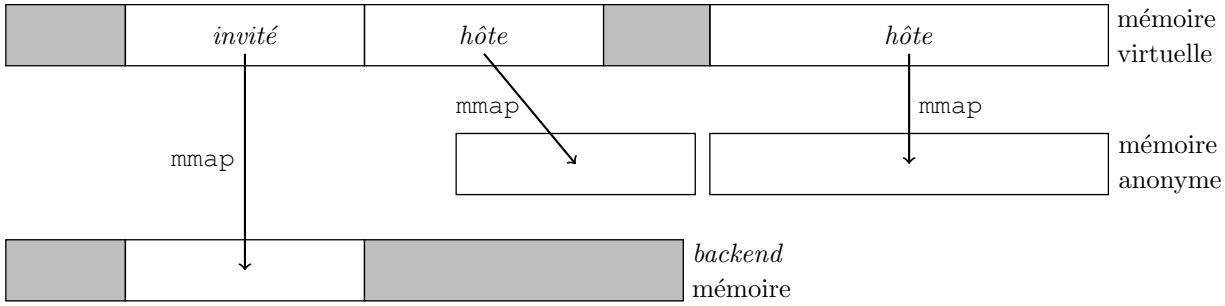
NMV: Virtualisation

Dans ce TP, on vous propose d'implémenter le système de gestion de la mémoire virtuelle dans un hyperviseur x86 de type 2 : "Janus". Cet hyperviseur fonctionne par exécution directe. Cette méthode ne permet pas de virtualiser l'ensemble du comportement d'un processeur x86 (voir cours). Cependant cette technique est suffisante pour exécuter un système d'exploitation "Rackdoll" similaire à celui utilisé dans un TP précédent. La gestion de la mémoire du système invité se fera via une *shadow page table* que vous devrez implémenter.

Cette version de Rackdoll effectue différentes tâches en mode noyau qui font appel à la mémoire virtuelle. On pourra vérifier que notre hyperviseur a un comportement correct en comparant les résultats de Rackdoll exécuté par cet hyperviseur avec ceux obtenus sous Qemu. La commande `make check` permet de compiler Janus et le système invité et d'exécuter ce dernier dans Janus. La commande `make qemu` permet de compiler le système invité et de le lancer dans l'émulateur Qemu.

La gestion de la mémoire dans Janus se fait de la manière suivante : au lancement de l'hyperviseur, celui-ci crée un fichier qui servira de mémoire physique à la machine virtuelle. On appelle ce fichier le *backend* mémoire. On rappelle que grâce au cache d'entrée / sortie de Linux, une écriture dans ce fichier ne se traduit pas immédiatement par une écriture sur disque. L'hyperviseur donne ensuite accès à ce *backend* mémoire en mappant ce fichier en mémoire virtuelle.

Gestion de la mémoire dans Janus :



Dans Janus, l'hyperviseur et le système invité partagent le même espace d'adressage virtuel au sein d'un unique processus en mode utilisateur.

Exercice 1

Pendant les premiers instants de *boot* d'un processeur x86, la mémoire paginée n'est pas encore activée et le système accède directement à la mémoire via des adresses physiques. On parle alors de modèle mémoire plat. Dans cet exercice, on implémente les fonctions qui émulent un modèle mémoire plat.

Question 1
Quel mapping l'hyperviseur doit-il mettre en place pour donner l'impression au système invité qu'il accède directement à la mémoire physique ?

L'hyperviseur doit mettre en place un mapping identité pour donner l'impression d'un modèle plat au système invité.

Le fichier `include/monitor/shadow.h` contient une description du modèle mémoire de Janus. Ce modèle porte uniquement sur la mémoire **virtuelle** du processus Janus. Vous constatez que certaines zones sont interdites au système invité. Ces zones contiennent le code ou les données de l'hyperviseur.

Question 2
Le fichier `monitor/shadow.c` contient le squelette de la fonction **`void`** `set_flat_mapping(`**`size_t`** `ram)` qui donne accès au système invité à la mémoire physique de `ram` octets via un modèle plat. Implémentez la fonction `set_flat_mapping`. Vous disposez pour cela de la fonction **`void`** `map_page(`**`vaddr_t`** `vaddr,` **`paddr_t`** `paddr,` **`size_t`** `len)` qui mappe l'adresse virtuelle `vaddr` sur l'adresse physique `paddr` du *backend* mémoire pour une taille de `len` octets.

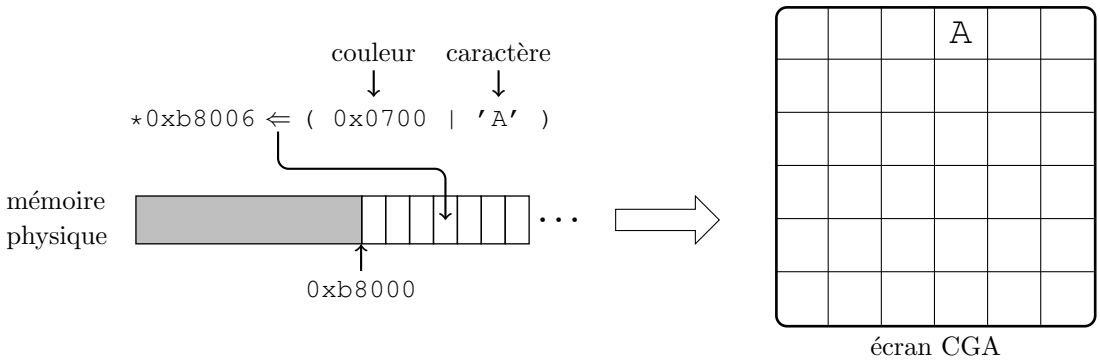
Attention : votre fonction `set_flat_mapping` ne doit pas donner accès à plus de mémoire physique que celle indiquée par le paramètre `ram`.

Note : la fonction **`void`** `display_mapping(`**`void`**`)`, disponible via `include/monitor/memory.h`, affiche sur la sortie standard le mapping du processus courant. Le format d'affichage est décrit par `man 5 proc` à la section `/proc/[pid]/maps`. Vous pouvez vous en servir pour vérifier le bon fonctionnement de vos fonctions.

Exercice 2

Un des premiers mode de communication d'un système d'exploitation sur un PC est l'écran CGA. Le système peut afficher du texte sur cet écran en écrivant dans une zone mémoire située à l'adresse `0xb8000`. Sur une machine physique, cette adresse pointe sur la mémoire vidéo, et les caractères écrits dans cette zone sont affichés à l'écran. Dans cet exercice on implémente les fonctions d'émulation d'accès à la mémoire CGA. La taille de la zone mémoire vidéo varie en fonction du mode d'affichage. Dans ce TP, on admettra qu'elle a une taille d'une page mémoire.

Fonctionnement de l'écran CGA :



Question 1
Avec un modèle mémoire plat, à quelle adresse virtuelle le système invité doit-il écrire pour atteindre l'adresse physique `0xb8000` ?

Avec un modèle mémoire plat, le système invité doit écrire à l'adresse virtuelle `0xb8000` pour atteindre l'adresse physique `0xb8000`.

Question 2
Étant donné le modèle mémoire de Janus, que se passe-t-il quand le système invité essaye d'afficher un caractère sur l'écran CGA ?

Quand le système invité essaye d'afficher un caractère à l'écran, il fait une écriture à une adresse virtuelle protégée. Cette ériture est donc interceptée par l'hyperviseur.

Le fichier `monitor/shadow.c` contient le corps de la fonction **`int`** `trap_write(`**`vaddr_t`** `addr,` **`size_t`** `size,` **`uint64_t`** `val)`, invoquée quand un accès mémoire du système invité est intercepté. L'accès intercepté est une écriture sur l'adresse `addr` et modifie `size` octets en mémoire pour y inscrire la valeur `val`. Cette fonction retourne `1` pour indiquer que l'instruction a été émulée et que le système invité peut passer à l'instruction suivante. Si la fonction retourne `0`, le système invité essayera d'exécuter à nouveau l'instruction.

Question 3

Completez le corps la fonction `trap_write` pour émuler l'affichage de caractères sur un écran CGA. Vous pouvez pour cela utiliser la fonction **`void`** `write_vga(`**`uint16_t`** `off,` **`uint16_t`** `val)` qui marque le caractère `val` dans l'écran virtuel CGA à la position `off` (la position `0` indique le caractère en haut à gauche de l'écran).

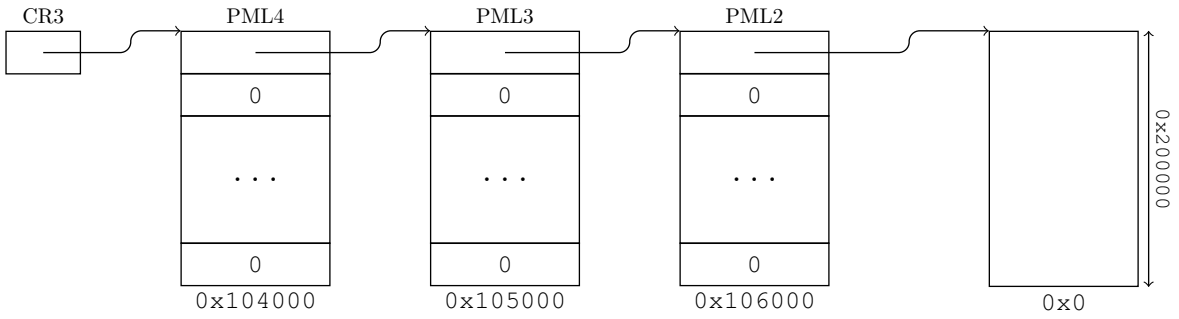
Note : La fonction `write_vga` ne fait pas d'affichage sur votre terminal. Pour afficher l'écran virtuel CGA sur votre terminal, vous pouvez invoquer la fonction **`void`** `display_vga(`**`void`**`)`.

Exercice 3

À cette étape, vous devriez avoir un modèle mémoire plat et un affichage CGA fonctionnel. Le système invité devrait être capable d'effectuer certaines des quatre tâches prévues avant d'échouer à cause d'un accès mémoire illégal. Avant de poursuivre vérifiez que tel est bien le cas. Vous pouvez aussi comparer les résultats obtenus sous Janus et sous Qemu.

Vous devriez observer que les résultats obtenus sous Janus et sous Qemu sont différents. En effet, le système invité utilise la table des pages pour créer des mappings particuliers qui influent sur les résultats affichés, or Janus n'émule pour l'instant qu'un modèle mémoire plat. Dans cet exercice, on implémente la mise en place d'une table des pages par le système invité.

La fonction **`void`** `set_page_table(`**`void`**`)` est invoquée quand le système invité met en place une nouvelle table des pages, par exemple en modifiant le `CR3`. La première table des pages mise en place par le système invité a la structure suivante :



Question 1
Implémentez une nouvelle fonction, que nous appellerons `parse_page_table`, qui parcours la nouvelle table

des pages mise en place par le système invité. Cette fonction affichera quels mappings virtuel → physique sont indiqués par la table des pages. On pourra, pour cette question, s'aider des réponses aux questions du TP mémoire virtuelle. Vous invoquerez cette fonction depuis `set_page_table`.

Attention : les niveaux intermédiaires de la table des pages sont indiqués par des adresses physiques. La fonction `void read_physical(void *dest, size_t size, paddr_t paddr)` permet de lire `size` octets dans le *backend* mémoire à l'adresse physique `paddr` et de les stocker dans le buffer `dest`.

Question 2
Pour des raisons pratiques, on vous propose d'enregistrer chaque mapping virtuel → physique indiqué dans la table des pages dans un tableau. On définit un mapping par trois propriétés :

- l'adresse virtuelle initiale
- l'adresse physique initiale
- la taille

Pour ce TP, on suppose qu'une table des pages n'indiquera jamais plus de 64 mappings. Modifiez les fonctions `set_page_table` et `parse_page_table` pour qu'elles enregistrent les mappings de la table des pages mise en place au lieu de les afficher.

Question 3
On va maintenant parcourir les mappings enregistrés pour les rendre effectifs. Implémentez une nouvelle fonction, que nous appellerons `update_mappings`, qui parcours les mappings enregistrés et les rend effectifs en mappant les adresses virtuelles indiquées sur le *backend* mémoire aux adresses physiques indiquées. On vous rappelle que vous disposez pour cela de la fonction `map_page`. Vous n'oublierez pas, dans cette nouvelle fonction, de supprimer l'ancien mapping mémoire via la fonction `void unmap_page(vaddr_t vaddr, size_t len)`.

Attention : on rappelle que le modèle mémoire de Janus interdit certaines plages d'adresses virtuelles au système invité. On pourra se contenter d'ignorer de telles plages d'adresse.

Question 4
Modifiez votre fonction `set_page_table` pour qu'elle rende effectifs les mappings mémoire enregistrés, à l'aide de votre fonction `update_mappings`. Vous pouvez vérifier le bon fonctionement de vos fonctions à l'aide de `display_mapping` (comparez le mapping obtenus avec ce que vous avez observé à la Question 1).

Exercice 4

L'hyperviseur Janus peut à présent émuler la mise en place d'une table des pages par le système invité. Cependant, vous devez remarquer que ce système invité est toujours incapable d'effectuer la totalité des tâches prévues. La première de ces tâches consiste à tester que les modifications apportées par le système invité à une table des pages déjà en place sont bien prises en compte. Ce n'est pas le cas dans l'implémentation actuelle de Janus.

La table des pages mise en place par le système invité n'est jamais lue par la MMU. Elle est seulement lue par votre fonction `parse_page_table` qui modifie ensuite la véritable table des pages, maintenue par Linux, à l'aide des fonctions `map_page` et `unmap_page`. Il faut donc que l'hyperviseur détecte les modifications apportées à la table des pages du système invité pour les retranscrire dans la table des pages du système hôte.

La détection de modification sur la table invitée se fait en protégeant les plages d'adresses virtuelles qui permettent d'accéder à cette table, comme dans le schéma suivant :

Les adresses virtuelles de la table invitée sont protégées en écriture, ainsi, toute modification par le système invité sera interceptée par l'hyperviseur.

Question 1
On rappelle que le `CR3` contient l'adresse physique du premier niveau de la table des pages courante. Pourquoi ne peut-on pas protéger en écriture l'adresse contenue dans le `CR3` ?

On ne peut protéger que des adresses virtuelles, protéger une adresse physique n'a pas de sens. Or `CR3` contient une adresse physique.

Pour résoudre ce problème, on propose la méthode suivante. La fonction `parse_page_table` enregistre non seulement les mappings, mais également les adresses physiques des niveaux intermédiaires de la table des pages. Plus tard, une fois la fonction `update_mappings` exécutée, on parcours toutes les entrées physiques enregistrées. Si un mapping correspond à cette entrée, on protège l'adresse virtuelle en écriture.

Question 2
On considère que les tables de pages ne contiennent jamais plus de 64 entrées intermédiaires. Modifiez la fonction `parse_page_table` pour enregistrer dans un nouveau tableau, les adresses physiques des entrées intermédiaires.

Question 3
Implémentez une nouvelle fonction, que nous appellerons `protect_pagelvls`, qui pour chaque adresse physique enregistrée, vérifie si un ou plusieurs mappings enregistrés contiennent l'adresse physique en question. Si tel est le cas, `protect_pagelvls` invoquera la fonction POSIX `mprotect` sur la ou les adresses virtuelles correspondante pour enlever les droits en écriture et en exécution.

Question 4
Modifiez votre fonction `set_page_table` pour qu'elle protège en écriture la table invitée. Vous pouvez vérifier le bon fonctionement de vos fonctions à l'aide de `display_mapping`.

Question 5
Si votre table invitée est correctement protégée, Janus devrait intercepter une requête d'écriture avec la fonction `trap_write` que vous avez déjà modifié dans l'exercice 2. Modifiez la fonction `trap_write` pour prendre en compte la modification par le système invité. Votre fonction devra modifier la table des page, sans oublier pour celà de donner les droits en écriture sur l'adresse modifiée, puis appeller la fonction `set_page_table` qui prendra en compte les modifications apportées.

Exercice 5

Malgré une gestion correcte de la table invitée, certains accès mémoire sont toujours interceptés par Janus en dehors de la zone d'adresses basses définie dans le modèle mémoire.

Question 1
En vous rappellant du fonctionement de l'allocation paresseuse, expliquez ce qui provoque ces interceptions.

Dans un système autorisant l'évaluation paresseuse, il est normal que des fautes de pages se produisent. Dans le cadre d'un système virtualisé, ces fautes sont interceptées par l'hyperviseur.

Question 2

Le code suivant permet d'injecter une interruption `PAGE_FAULT` dans le système invité pour une faute à l'adresse `addr` :

```
set_control(addr, 2);
trigger_interrupt(INTERRUPT_PF, 0);
```

Corrigez les fonctions `trap_read` et `trap_write` pour gérer les interceptions identifiées à la question précédente.

Question 3

Comparez les résultats obtenus par le système invité sous Janus et sous Qemu. Vérifiez que les résultats sont bien identiques.