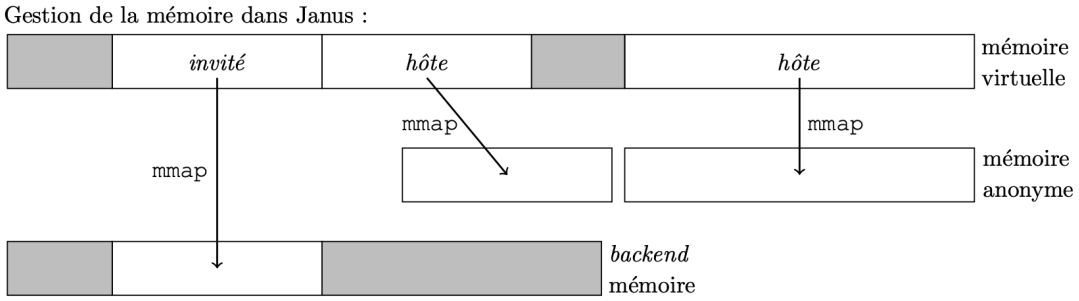
TP3 : Virtualization

Introduction

Janus (hypervisor) crée un fichier qui sert de mémoire physique a Rackdoll (guest). Janus et Rackdoll partagent le même espace d'adressage virtuel, dans un processus utilisateur.



Pour cela, le fichier est directement `unlink` mais reste en mémoire tant que le programme n'est pas terminé, puis nous utilisons :

```
mmap((void *)vaddr, len, PROT_READ | PROT_WRITE | PROT_EXEC,
       MAP_SHARED | MAP_FIXED, guest_memory_backend, paddr);
```

Plusieurs options sont utilisées pour avoir le comportement attendu :

- `MAP_FIXED` : Force l'adresse virtuelle `addr` à être alignée, dans notre cas à la taille d'une page de 4 KB, sinon l'offset est interprété comme un hint.
- `MAP_SHARED` : Permet de forcer les modifications faite au fichier à être directement propagés.

Exercice 1

Quel mapping Janus doit-il mettre en place pour donner l'impression a Rackdoll qu'il accède directement à la mémoire physique ?

```
/*
* Memory access emulation and shadow
paging.
* Memory model for Janus VMM
*
* +----------------------+
0xffffffffffffff
* | Monitor              |
* | (forbidden accesses) |
* +----------------------+ 0x700000000000
* | Guest                |
* +----------------------+ 0x100000000
* | Monitor              |
* | (forbidden accesses) |
* +----------------------+ 0x80000000
* | Guest                |
* +----------------------+ 0x100000
* | Monitor              |
* | (trapped accesses)   |
* +----------------------+ 0x0
*/
```

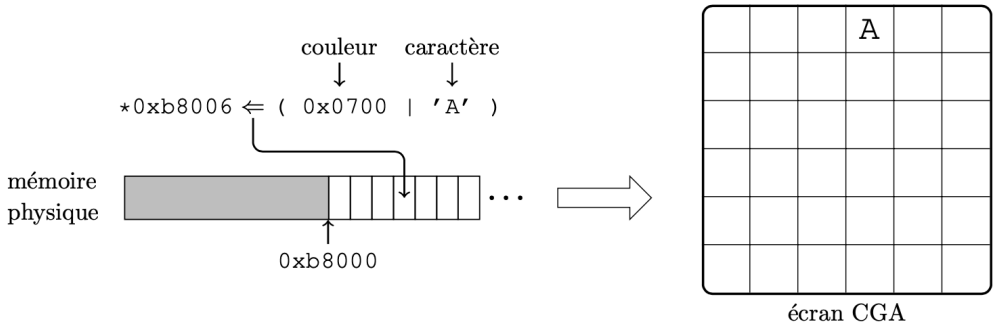
```
/*
* Memory model for the guest
*
* +----------------------+
0xffffffffffffff
* | Forbidden            |
* +----------------------+ 0x700000000000
* | Guest memory         |
* | Automatic mapping    |
* +----------------------+ 0x100000000
* | Forbidden            |
* +----------------------+ 0x80000000
* | Guest memory         |
* | Manual mapping       |
* +----------------------+ 0x200000
* | Guest memory         |
* +----------------------+ 0x100000
* | Device mapped memory |
* +----------------------+ 0x0
*/
```

Nous sommes dans un modèle plat, dans ce cas les adresses virtuelles de l'invité corresponde directement aux adresses physiques. L'hyperviseur doit donc mettre en place une correspondance identité (1 pour 1).

Les accès aux zones réservées (matériel par exemple) se font à l'aide d'un **trapped access** en placant une protection avec `mprotect`. Cela permet à l'hyperviseur d'intercepter tout accès interdit afin d'émuler ou refuser les opérations de l'invité sur ces zones.

Exercice 2

Fonctionnement de l'écran CGA :



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

Dans un modèle mémoire plat, l'adresse virtuelle correspond à l'adresse physique, il suffit donc d'écrire à la même adresse.

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

Lorsque le système invité écrit un mot dans la zone réservé au VGA, une interruption est déclenché sur le système hôte car la zone est protégée. L'instruction mémoire est ensuite émulé, et `trap_write` est appelé et va s'occuper d'aller écrire dans le VGA virtuel dans la structure `guest_state`.

Exercice 3

Les mappings vaddr → paddr de la table des pages de chaque processus sont enregistrés dans une structure `shadow_table`.
Cela nous permet lors de la modification de la table des pages de mettre à jours les mappings pour le processus courant dans la table des pages de l'hote à l'aide de `update_mapping`.

Il ne faut pas oublier de **reset** les mappings de la shadow page table vu que qu'ils sont **tous** récupérés dans `parse_page_table`.

```
struct shadow_table_entry {
  paddr_t paddr;
  vaddr_t vaddr;
  size_t size;
};

struct shadow_table {
  paddr_t cr3;
  struct shadow_table_entry entries[SHADOW_TABLE_ENTRIES];
  size_t len;
```

```
};

extern struct shadow_table shadow_tables[SHADOW_TABLE_NB];
```

Exercice 4

Nous utilisons un modèle mémoire plat, et la table des pages de l'invité et protégée en écriture, ce qui nous permet d'intercepter au niveau de l'hyperviseur toutes modifications lui étant faite.

Pourquoi ne peut-on pas protéger en écriture l'adresse contenue dans le CR3 ?

Il n'est pas possible de protéger le CR3 car il contient une adresse physique (du PML4) et non virtuelle. De plus, le CR3 étant un pointer vers la table des pages et non son contenu, le protéger serait inutile pour intercepter la modification des entrées de la table.

Enregistrement des niveaux intermédiaires

Afin de protéger les entrées de la table des pages, nous enregistrons les adresses physiques des pages contenant les niveaux intermédiaires. Il suffit ensuite de les protéger avec `mprotect` sur la taille d'une page, ici 4 KB.
De plus, il faut vérifier que les adresses physique devant être protégées sont bien mappées dans la table des pages, en utilisant la shadow page table.

Après protection des pages, nous observer qu'un nouveau mapping a bien été créé et correspond bien aux adresses des pages intermédiaires initiales de la table des pages :

```
00100000-00105000 rwxs 00100000 00:22 3709                               /tmp/janus-
backend.mem (deleted)
# new entry here
# we are mapping pages 0x105000 and 0x106000 of 0x1000 sizes.
00105000-00107000 r--s 00105000 00:22 3709                               /tmp/janus-
backend.mem (deleted)
00107000-80000000 rwxs 00107000 00:22 3709                               /tmp/janus-
backend.mem (deleted)
```

Mise à jour de la table des pages

Après interception de l'interruption en écriture, il faut mettre à jour l'entrée dans la table des pages. Pour cela, il faut lever la protection en écriture en rajoutant des permission avec `mprotect` après avoir vérifier que l'adresse est bien une page intermédiaire valide de la table des pages.

Exercice 5

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

La mémoire dans la stack est alloué de façon paresseuse en se basant sur les `PAGE_FAULT` pour allouer les pages demandées. Or, comme la table des pages de l'hyperviseur est utilisée, il faut donc gérer les `PAGE_FAULT` au niveau de l'hôte.

Notes

- Il ne faut pas mapper la partie haute de l'invité car ses pages seront mappées en utilisant l'allocation paresseuse (PAGEFAULT).
- Il faudrait rajouter une question pour unmap les pages plus utilisées, la solution mise en place ici est de trap les operations de page unmap. Ajout de l'exemple, doit segfault sur qemu et janus :

```
// at the end of do_zero_copy
munmap(MAP_VADDR_LOW_0);
munmap(MAP_VADDR_LOW_1);
// uncommenting me should trap read (unmapped entry)
// printk("  %p = %u %u %u\n", ptr1, ptr1[0], ptr1[1], ptr1[2]);
```