TP1 NMV : Gestion de la mémoire virtuelle

Aymeric Agon-Rambosson

Mardi 19 janvier 2021

Exercice 1

Question 1

Réfléchissons un peu.

Le processeur n'est capable de demander que des adresses virtuelles à sa MMU. Si on veut accéder à une adresse physique bien particulière, parce qu'on ne connaît que l'adresse physique d'une donnée (la table des pages elle-même, par exemple), on doit avoir une transformation virtuel -> physique connue (et bijective !) a priori pour accéder à la donnée.

Techniquement, il existe une entrée de la table des pages qui contient l'adresse physique de cette table des pages. Donc il existe une adresse virtuelle qui après traduction nous donnera bien l'adresse physique de la table des pages. Mais cette adresse virtuelle dépend de la position de l'entrée idoine dans la table des pages, et cette position doit être connue a priori.

Une manière de rendre cette position, donc cette transformation, connue a priori pour le processeur est de mapper les adresses de tous les niveaux de la table des pages par une identité (ce n'est pas la seule manière). On dit bien **tous les niveaux**, et pas seulement les niveaux "intermédiaires" : cette propriété doit aussi tenir pour l'adresse de PML4, contenue dans le registre CR3.

D'ailleurs, après vérification, on a bien le même retour pour `xp/8g 0x104000` et `x/8g 0x104000`.

Question 2

On suppose une taille de page de 4096 octets. On a par hypothèse une taille d'entrée de table des pages de 8 octets.

On aura donc 512 entrées de table des pages par page.

Il y a une seule page PML4, dont l'adresse physique est donnée par le registre CR3, et cette unique page contient 512 entrées.

Il y a 512 pages PML3, dont les adresses physiques sont données par les entrées de la précédente. Chacune de ces 512 pages contient elle-même 512 entrées (soient 262144 entrées en tout).

Il y a 262144 pages PML2, dont les adresses physiques sont données par les entrées de la précédente. Chacune de ces 262144 pages contient une entrée (soient 262144 entrées en tout) contenant l'adresse physique (0x0) d'une page de 2 Mio et 511 entrées (soient 3241052548 entrées en tout).

Il y a 3241052548 pages PML1, dont les adresses physiques sont données par les entrées de la précédente (toutes sauf les 262144 qui contiennent l'adresse de la page de 2 Mio). Chacune de ces pages contient 512 entrées.

Une entrée de table des pages est valide si le bit 0 est à 1.

Une entrée de table des pages est terminale si elle est une entrée de PML1, où si elle est une entrée de PML2 et qu'elle est une hugepage. On peut savoir le niveau d'une entrée, parce qu'on sait quelle partie de l'adresse virtuelle on a dû décoder pour trouver l'entrée.

Question 3

Puisque la taille d'une page est de 4096 octets, on aura les 12 bits de poids faible utilisés pour le décalage dans la page.

Les 40 bits suivants serviront à stocker l'adresse physique de la page.

On peut donc se définir les macros suivantes :

```
#ifndef _INCLUDE_PGT_H_
#define _INCLUDE_PGT_H_

#define PGT_VALID_MASK 0x1
#define PGT_ADDRESS_MASK 0xFFFFFFFFFF000
#define PGT_HUGEPAGE_MASK 0x80

#define PGT_IS_VALID(p) (p & PGT_VALID_MASK)
#define PGT_IS_HUGEPAGE(p) (p & PGT_HUGEPAGE_MASK)
#define PGT_ADDRESS(p) (p & PGT_ADDRESS_MASK)

#endif
```

Voilà notre réalisation de la fonction :

```
void print_pgt(paddr_t pml, uint8_t lvl)
{
        uint64_t *p;
        int i;
        p = (uint64_t *)pml;

        if (lvl == 4)
                printk("cr3 : 0x%lx\n", pml);

        if (lvl < 1)
                return;

        for (i = 0; i < 512; ++i) {
                if (PGT_IS_VALID(p[i])) {
                        printk("pml%d %d : 0x%lx\n", lvl, i, PGT_ADDRESS(p[i]));
                        if (!PGT_IS_HUGEPAGE(p[i])) {
                                print_pgt((paddr_t) PGT_ADDRESS(p[i]), lvl - 1);
                        }
                }
        }
}
```

La fonction marche bien correctement, on a bien la bonne structure affichée.

Exercice 2

Question 1

Plutôt que de donner une formule analytique, on va donner une formule énumérative. On sait que lvl ne peut prendre que les valeurs 4, 3, 2 et 1.

Soit une adresse virtuelle sur 48 bits (on oublie l'extension de signe).

Les 9 bits de poids fort (47 à 39) vont donner l'index dans PML4. On peut donc se débarasser de tout

ce qui se trouve à gauche avec le masque 0xFF8000000000, puis de tout ce qui est à droite avec un décalage à droite de 39.

etc, etc...

On aura donc :

- lvl = 4 -> index := (vaddr & 0xFF8000000000) » 39
- lvl = 3 -> index := (vaddr & 0x7FC0000000) » 30
- lvl = 2 -> index := (vaddr & 0x3FE00000) » 21
- lvl = 1 -> index := (vaddr & 0x1FF000) » 12

Question 2

Sur la base de la question précédente, on peut se donner les macros suivantes :

```
#ifndef _INCLUDE_PGT_H_
#define _INCLUDE_PGT_H_

#define PGT_VALID_MASK 0x1
#define PGT_WRITABLE_MASK 0x2
#define PGT_USER_MASK 0x4
#define PGT_ADDRESS_MASK 0xFFFFFFFFFF000
#define PGT_HUGEPAGE_MASK 0x80

#define PGT_IS_VALID(p) (p & PGT_VALID_MASK)
#define PGT_IS_HUGEPAGE(p) (p & PGT_HUGEPAGE_MASK)
#define PGT_ADDRESS(p) (p & PGT_ADDRESS_MASK)

#define PGT_PML4_INDEX_MASK 0xFF8000000000
#define PGT_PML3_INDEX_MASK 0x7FC0000000
#define PGT_PML2_INDEX_MASK 0x3FE00000
#define PGT_PML1_INDEX_MASK 0x1FF000

#define PGT_PML4_INDEX(v) ((v & PGT_PML4_INDEX_MASK) >> 39)
#define PGT_PML3_INDEX(v) ((v & PGT_PML3_INDEX_MASK) >> 30)
#define PGT_PML2_INDEX(v) ((v & PGT_PML2_INDEX_MASK) >> 21)
#define PGT_PML1_INDEX(v) ((v & PGT_PML1_INDEX_MASK) >> 12)


#endif
```

La fonction à implémenter est en fait assez simple :

- On récupère l'adresse physique du premier niveau de la table des pages.
- On vérifie si l'entrée de ce premier niveau correspondant au PML4 est valide. Si elle ne l'est pas, on alloue une page, on la blanchit, et on met son adresse dans l'entrée correspondante, on positionne le bit de validité, d'écriture autorisée, et d'accessibilité pour l'utilisateur.
- On récupère l'adresse stockée à cette entrée (qu'on sait maintenant nécessairement exister), et on recommence avec le niveau suivant.

Voilà donc notre réalisation de la fonction `map_page` :

```
void map_page(struct task *ctx, vaddr_t vaddr, paddr_t paddr)
{
```

```
uint64_t *p = (uint64_t *)ctx->pgt;
paddr_t tmp;

printk("\n0x%lx 0x%lx 0x%lx 0x%lx\n", PGT_PML4_INDEX(vaddr),
       PGT_PML3_INDEX(vaddr),
       PGT_PML2_INDEX(vaddr),
       PGT_PML1_INDEX(vaddr));

/* PML4 */
if (!PGT_IS_VALID(p[PGT_PML4_INDEX(vaddr)])) {
        printk("Entry is invalid\n");

        tmp = alloc_page(); /* TODO : handle error */

        memset((void *)tmp, 0, 4096);
        p[PGT_PML4_INDEX(vaddr)] |= (tmp | PGT_VALID_MASK |
                                     PGT_WRITABLE_MASK | PGT_USER_MASK);

} else {
        printk("Entry is valid\n");
}

p = (uint64_t *)PGT_ADDRESS(p[PGT_PML4_INDEX(vaddr)]);

printk("0x%lx\n", p);

/* PML3 */
if (!PGT_IS_VALID(p[PGT_PML3_INDEX(vaddr)])) {
        printk("Entry is invalid\n");

        tmp = alloc_page(); /* TODO : handle error */

        memset((void *)tmp, 0, 4096);
        p[PGT_PML3_INDEX(vaddr)] |= (tmp | PGT_VALID_MASK |
                                     PGT_WRITABLE_MASK | PGT_USER_MASK);
} else {
        printk("Entry is valid\n");
}

p = (uint64_t *)PGT_ADDRESS(p[PGT_PML3_INDEX(vaddr)]);

printk("0x%lx\n", p);

/* PML2 */
if (!PGT_IS_VALID(p[PGT_PML2_INDEX(vaddr)])) {
        printk("Entry is invalid\n");

        tmp = alloc_page(); /* TODO : handle error */

        memset((void *)tmp, 0, 4096);
        p[PGT_PML2_INDEX(vaddr)] |= (tmp | PGT_VALID_MASK |
                                     PGT_WRITABLE_MASK | PGT_USER_MASK);
} else {
```

```
                printk("Entry is valid\n");
        }

        p = (uint64_t *)PGT_ADDRESS(p[PGT_PML2_INDEX(vaddr)]);

        printk("0x%lx\n", p);

        /* PML1 */
        if (!PGT_IS_VALID(p[PGT_PML1_INDEX(vaddr)])) {
                printk("Mapping virtual address 0x%lx onto physical address 0x%lx\n",
                       vaddr, paddr);

                memset((void *)paddr, 0, 4096);
                p[PGT_PML1_INDEX(vaddr)] |= (paddr | PGT_VALID_MASK |
                                             PGT_WRITABLE_MASK | PGT_USER_MASK);
        } else {
                printk("[error] Virtual address 0x%lx already mapped\n", vaddr);
        }
}
```

La fonction marche correctement. On ne gère pas encore les erreurs de la fonction `alloc_page`, ou l'erreur liée au fait que l'adresse virtuelle est déjà mappée.

Exercice 3

Question 1

L'adresse virtuelle `0x2000000030` est une adresse virtuelle dans la partie "User (text + data + heap)" (les adresses `0x2000000000` à `0x00007fffffffffff`). D'après la documentation du modèle mémoire, les adresses situées dans cet intervalle doivent être explicitement mappées pour être utilisées. Or, on ne les a mappé à aucun moment : on a donc une faute de page.

Question 2

Il faut garder dans la table courante toutes les adresses qui concernent le noyau, soient toutes les adresses entre `0x0` et `0x40000000`. C'est important parce que toutes les tâches sont susceptibles de faire des appels systèmes, et doivent donc disposer des adresses du code, mais aussi des données du noyau. Le code et les données du noyau sont mappées sur les mêmes adresses physiques pour toutes les tâches du système.

La totalité des pages d'adresse entre `0x0` et `0x400000000` sont toutes référencées par la page pointée par la première entrée du premier PML3 (autrement dit, la première entrée de la page pointée par la première entrée de PML4). Dans l'initiatisation de Rackdoll, cette page est mise à l'adresse `0x106000` pour la première tâche créée par le système.

Il faut que pour deux tâches arbitraires le contenu de la table des pages pour les pages noyau soit tout le temps le même. En effet, si une tâche modifie via un appel système les données du noyau, et ce faisant alloue une nouvelle page dans la zone du noyau, il faut qu'une autre tâche puisse aussi accéder à ces données modifiées, donc que le mapping existe aussi dans la table des pages de cette deuxième tâche. Deux solutions :

- chaque tâche a sa propre version de la table des pages à partir de l'entrée 0 du premier PML3, et à chaque fois qu'une tâche fait une allocation dans la zone noyau de la mémoire, il faut répliquer ce mapping pour toutes les autres tâches de la machine. Possible mais très pénible.
- pour toutes les tâches, l'entrée 0 du premier PML3 pointe vers la même page physique (la famosa `0x106000`, ici hardcodée par le code d'initialisation `entry.S`). Solution beaucoup plus simple,

requiert en revanche un verrou à chaque fois qu'une tâche fait une allocation **dans la zone noyau de la mémoire** (la zone utilisateur n'est pas concernée : différente pour toutes les tâches).

C'est bien entendu la deuxième solution qui va être choisie.

Il va donc s'agir de copier dans la nouvelle table des pages la première entrée du premier PML3 : même adresse, mais aussi mêmes flags.

Question 3

Il est question ici d'une nouvelle tâche, à la table des pages vide (si on excepte les adresses noyau). Donc on peut simplement placer en virtuel le payload en bas de la section "User (text + data + heap)" (soit à l'adresse virtuelle `0x2000000000`) et le bss juste au-dessus, et pour l'allocation dynamique qui ne manquera pas d'arriver, on placera ça encore au-dessus.

De fait, il faut aussi que le placement des sections des tâches dans le modèle mémoire du noyau corresponde aux adresses fixées dans la chaîne de compilation des applications. Or, si on regarde le fichier `task.ld`, on voit que la base des sections est l'adresse virtuelle `0x2000000000`. Donc on peut et **on doit** placer ces sections au-dessus de `0x2000000000`.

La taille du payload est donnée par la différence entre `ctx->load_end_paddr` et `ctx->load_paddr` (on admet la contiguité).

Donc le payload ira de l'adresse virtuelle `0x2000000000` à l'adresse virtuelle `0x2000000000` + (`ctx->load_end_paddr` - `ctx->load_paddr`), et bss ira de l'adresse virtuelle `0x2000000000` + (`ctx->load_end_paddr` - `ctx->load_paddr`) à l'adresse virtuelle `ctx->bss_end_vaddr`.

Le bss de la tâche est vide si et seulement si `ctx->bss_end_vaddr` est égal à `0x2000000000` + (`ctx->load_end_paddr` - `ctx->load_paddr`), ce qui est le cas des deux premières tâches.

On devra donc allouer autant de pages que nécessaire pour aller de `0x2000000000` à `ctx->bss_end_vaddr`. Et on devra en plus initialiser à zéro toutes les pages physiques correspondant à la section bss.

On fait l'hypothèse que le début de la section bss est aligné sur une page (à dire vrai, l'analyse du fichier `task.ld` nous permet de faire cette hypothèse avec confiance).

Question 4

La fonction `load_task()` devra donc initialiser la table des pages, soit :

- Allouer une page pour PML4, et mettre l'adresse de cette page dans le CR3 de la nouvelle page (donc dans `ctx->pgt`).
- Allouer une page pour PML3. Mettre l'adresse de cette page dans la première entrée du PML4. Recopier dans la première entrée de ce PML3 le contenu de la première entrée du premier PML3 de la page courante. Une fois que c'est fait, toutes les adresses noyau accessibles pour la tâche en cours le sont aussi pour la tâche en train d'être initialisée.
- Mapper grâce à la fonction `map_page()` les pages du payload. Les adresses virtuelles commenceront à `ctx->load_vaddr`.
- Allouer, initialiser à 0 et mapper grâce à la fonction `map_page()` les pages du bss, si il y en a.

Voilà à quoi ressemble notre implémentation :

```
void load_task(struct task *ctx)
{
        uint64_t *p;
        paddr_t paddr;
        vaddr_t vaddr;
```

```
        /* Adresses noyau faites manuellement */
        paddr = alloc_page();
        memset((void *)paddr, 0, 4096);
        ctx->pgt = paddr;

        p = (uint64_t *)ctx->pgt;
        paddr = alloc_page();
        memset((void *)paddr, 0, 4096);
        p[0] |= (paddr | PGT_VALID_MASK | PGT_WRITABLE_MASK | PGT_USER_MASK);

        p = (uint64_t *)store_cr3();
        paddr = PGT_ADDRESS(*(uint64_t *)PGT_ADDRESS(*p));
        p = (uint64_t *)PGT_ADDRESS(*((uint64_t *)ctx->pgt));
        p[0] |= (paddr | PGT_VALID_MASK | PGT_WRITABLE_MASK | PGT_USER_MASK);

        /* Adresses payload */
        for (paddr = ctx->load_paddr; paddr < ctx->load_end_paddr; paddr += 4096) {
                vaddr = ctx->load_vaddr + paddr - ctx->load_paddr;
                /* printk("%lx\n", vaddr); */
                /* printk("%lx\n", paddr); */
                map_page(ctx, vaddr, paddr);
        }

        /* Adresses bss */
        for (vaddr = ctx->load_vaddr + ctx->load_end_paddr - ctx->load_paddr; /* On eût pu faire
             vaddr < ctx->bss_end_vaddr; vaddr += 4096) {
                paddr = alloc_page();
                /* printk("%lx\n", vaddr); */
                /* printk("%lx\n", paddr); */
                memset((void *)paddr, 0, 4096);
                map_page(ctx, vaddr, paddr);
        }
}
```

Question 5

Cette fonction consiste simplement en le remplacement du CR3.

Donc :

```
void set_task(struct task *ctx)
{
        load_cr3(ctx->pgt);
}
```

Exercice 4

À ce moment-là, on se retrouve avec le message suivant :

Page fault at 0x200000003a cr2 = 0x1fffffff8

Question 1

Cette question est très simple : la fonction doit simplement allouer une page, l'initialise et la mapper.

```
void mmap(struct task *ctx, vaddr_t vaddr)
{
        paddr_t paddr = alloc_page();

        memset((void *)paddr, 0, 4096);
        map_page(ctx, vaddr, paddr);
}
```

Question 2

Pour être exact, la console affiche le message suivant :

> Page fault at 0x200000003a cr2 = 0x1fffffff8

On sait que l'adresse `0x200000003a` est une adresse à laquelle il est possible d'accéder, on a alloué une page à l'adresse `0x2000000000`, donc toutes les adresses de `0x2000000000` à `0x2000001000` sont valides.

On sait aussi que l'adresse `0x200000003a` est une adresse de la section de texte.

Je pense qu'on doit plutôt interpréter le message comme ça :

Une instruction à l'adresse `0x200000003a` (non alignée sur un mot ? Une spécificité de x86_64 probablement...) fait un accès mémoire à l'adresse `0x1fffffff8`, ce qui a déclenché une faute de page.

Après vérification du modèle mémoire de Rackdoll, on s'aperçoit que l'adresse `0x1fffffff8` est une adresse de la pile. C'est un accès mémoire tout-à-fait légitime : l'application accède à sa pile librement, sans avoir besoin de la faire correspondre explicitement. La responsabilité de cette correspondance est la nôtre.

Question 3

L'allocation paresseuse consiste à faire l'allocation seulement au moment où on en a besoin. Dans le cas de Rackdoll, l'intérêt est évident : on ne peut pas allouer de manière gloutonne les 127 Gio de la pile, la machine ne dispose pas de pareille quantité de mémoire. On doit donc allouer au moment où l'application demande à accéder à une partie non encore allouée de sa pile.

Question 4

La fonction `pgfault()` doit faire les choses suivantes :

Vérifier si l'adresse fautive est une adresse de la pile, autrement dit si c'est use adresse située au dessus de l'adresse de base de la pile, mais en-dessous de l'adresse de base de la section de texte, données et tas.

Si c'est une adresse de la pile, il faut allouer la page correspondante (donc la page **en-dessous**). On peut donc définir les macros suivantes :

```
#define STACK_BASE 0x40000000
#define TEXT_BASE 0x2000000000
#define ADDRESS_FROM_STACK(a) (TEXT_BASE > a && a >= STACK_BASE)
#define PAGE_BELOW(a) ((a / PAGE_SIZE) * PAGE_SIZE)
```

Si ce n'est pas une adresse de la pile, il faut tuer la tâche.

Voilà notre implémentation :

```
void pgfault(struct interrupt_context *ctx)
{
        vaddr_t vaddr = store_cr2();
        paddr_t paddr;
```

```
        if (ADDRESS_FROM_STACK(vaddr)) {
                paddr = alloc_page();
                memset((void *)paddr, 0, PAGE_SIZE);
                map_page(current(), PAGE_BELOW(vaddr), paddr);
        } else {
                printk("Page fault at %p\n", ctx->rip);
                printk("  cr2 = %p\n", vaddr);
                exit_task(ctx);
        }
}
```

Tout fonctionne bien comme prévu : on se retrouve bien avec les erreurs qu'on est censé voir au début de l'exercice 5.

Exercice 5

Question 1

La fonction `munmap()` doit donc :

- Récupérer l'adresse physique de la page virtuelle à libérer.
- Libérer la page physique correspondante.
- Invalider l'entrée de PML1 correspondante.
- Si l'invalidation en question fait que toutes les entrées de la PML1 sont invalides, libérer la page PML1.
- Faire de même en remontant.

On prendra donc la peine de stocker des pointeurs intermédiaires pour les niveaux intermédiaires de la table des pages.

On se donne les macros suivantes, qui sont celles qu'on avait déjà, plus une macro d'invalidation :

```
#define PGT_VALID_MASK 0x1
#define PGT_INVALID_MASK 0x0
#define PGT_ADDRESS_MASK 0xFFFFFFFFFF000

#define PGT_IS_VALID(p) (p & PGT_VALID_MASK)
#define PGT_ADDRESS(p) (p & PGT_ADDRESS_MASK)
#define PGT_INVALIDATE(p) (p &= PGT_INVALID_MASK)

#define PGT_PML4_INDEX_MASK 0xFF8000000000
#define PGT_PML3_INDEX_MASK 0x7FC0000000
#define PGT_PML2_INDEX_MASK 0x3FE00000
#define PGT_PML1_INDEX_MASK 0x1FF000

#define PGT_PML4_INDEX(v) ((v & PGT_PML4_INDEX_MASK) >> 39)
#define PGT_PML3_INDEX(v) ((v & PGT_PML3_INDEX_MASK) >> 30)
#define PGT_PML2_INDEX(v) ((v & PGT_PML2_INDEX_MASK) >> 21)
#define PGT_PML1_INDEX(v) ((v & PGT_PML1_INDEX_MASK) >> 12)
```

Voilà notre implémentation :

```
void munmap(struct task *ctx, vaddr_t vaddr)
{
```

```
        uint64_t *p4, *p3, *p2, *p1;
        paddr_t paddr;
        int i, c;

        p4 = (uint64_t *)ctx->pgt;
        p3 = (uint64_t *)PGT_ADDRESS(p4[PGT_PML4_INDEX(vaddr)]);
        p2 = (uint64_t *)PGT_ADDRESS(p3[PGT_PML3_INDEX(vaddr)]);
        p1 = (uint64_t *)PGT_ADDRESS(p2[PGT_PML2_INDEX(vaddr)]);
        paddr = PGT_ADDRESS(p1[PGT_PML1_INDEX(vaddr)]);

        free_page(paddr);
        PGT_INVALIDATE(p1[PGT_PML1_INDEX(vaddr)]);

        c = 0;
        for (i = 0; i < 512; ++i)
                if (PGT_IS_VALID(p1[i]))
                        ++c;

        if (!c) {
                free_page((paddr_t)p1);
                PGT_INVALIDATE(p2[PGT_PML2_INDEX(vaddr)]);
        }

        c = 0;
        for (i = 0; i < 512; ++i)
                if (PGT_IS_VALID(p2[i]))
                        ++c;

        if (!c) {
                free_page((paddr_t)p2);
                PGT_INVALIDATE(p3[PGT_PML3_INDEX(vaddr)]);
        }

        c = 0;
        for (i = 0; i < 512; ++i)
                if (PGT_IS_VALID(p3[i]))
                        ++c;

        if (!c) {
                free_page((paddr_t)p3);
                PGT_INVALIDATE(p4[PGT_PML4_INDEX(vaddr)]);
        }

        /* On ne va pas enlever le PML4 : on a toujours les adresses noyau */
}
```

Notre implémentation fonctionne bien, les fuites mémoire ont disparu.

Par contre, elle n'est pas vraiment optimale. On pourrait définir plein de raccourcis, et la simplifier.

Question 2

On a effectivement un échec indiqué.

En lisant le code de `adversary()`, on s'aperçoit que l'échec est déclenché s'il est possible de lire un

seul bit à 1 dans une page de la pile qui a été explicitement libérée.

La fonction `adversary()` noircit une page de la pile, puis la libère explicitement. En accédant plus tard à cette même page, la page n'est pas blanche ! Or, dans `pgfault()`, on blanchit la page avant de faire la correspondance.

La solution à cet apparent paradoxe est qu'on ne passe jamais dans `pgfault()` lors du deuxième accès, parce qu'on ne déclenche pas de faute de page, parce qu'on ne parcourt pas la table des pages. La correspondance virtuel-physique qui a été créée lors du premier accès est toujours présente dans le *translation lookaside buffer*. Le processeur la retrouve, et accède aux données sans déclencher de faute de page.

Question 3

Pour régler le problème, on a deux techniques.

La mauvaise : blanchir manuellement toutes les pages qu'on libère. Ça fonctionne, la tâche `adversary()` retourne un succès. Mais ça prend du temps (**on blanchit manuellement toutes les pages qu'on libère**), et on ne sait pas quel effet ça a sur le matériel.

La bonne : on invalide l'entrée du tlb à l'aide de l'instruction assembleur idoine (`invlpg` en x86). Quand la tâche essaie d'accéder à nouveau aux données, elle déclenche une faute de page, et donc blanchit la page. Quand une autre tâche essaie d'accéder à ces mêmes données (donc nécessairement plus tard, après que la précédente est morte), elle déclenche nécessairement une faute de page, donc elle blanchit elle-même la page à laquelle elle tente d'accéder.

Le bonne méthode est la version paresseuse de la mauvaise : on ne blanchit que ce qu'on a besoin de blanchir, **au moment où on a besoin de le faire**. La bonne méthode utilise bien la famosa technique de "déclencher une faute pour obtenir le traitement qu'on voulait".

Par contre, on a le droit de blanchir manuellement les pages de la tables des pages le cas échéant. En effet, on ne connaît que leur adresse physique (et l'instruction `invlpg` prend en entrée une adresse virtuelle), et elles sont en nombre limité.

Voilà donc l'ultime version de notre fonction `munmap()` :

```
void munmap(struct task *ctx, vaddr_t vaddr)
{
        uint64_t *p4, *p3, *p2, *p1;
        paddr_t paddr;
        int i, c;

        p4 = (uint64_t *)ctx->pgt;
        p3 = (uint64_t *)PGT_ADDRESS(p4[PGT_PML4_INDEX(vaddr)]);
        p2 = (uint64_t *)PGT_ADDRESS(p3[PGT_PML3_INDEX(vaddr)]);
        p1 = (uint64_t *)PGT_ADDRESS(p2[PGT_PML2_INDEX(vaddr)]);
        paddr = PGT_ADDRESS(p1[PGT_PML1_INDEX(vaddr)]);

        free_page(paddr);
        PGT_INVALIDATE(p1[PGT_PML1_INDEX(vaddr)]);

        c = 0;
        for (i = 0; i < 512; ++i)
                if (PGT_IS_VALID(p1[i]))
                        ++c;

        if (!c) {
                memset((void *)p1, 0, PAGE_SIZE);
```

```
                free_page((paddr_t)p1);
                PGT_INVALIDATE(p2[PGT_PML2_INDEX(vaddr)]);
        }

        c = 0;
        for (i = 0; i < 512; ++i)
                if (PGT_IS_VALID(p2[i]))
                        ++c;

        if (!c) {
                memset((void *)p2, 0, PAGE_SIZE);
                free_page((paddr_t)p2);
                PGT_INVALIDATE(p3[PGT_PML3_INDEX(vaddr)]);
        }

        c = 0;
        for (i = 0; i < 512; ++i)
                if (PGT_IS_VALID(p3[i]))
                        ++c;

        if (!c) {
                memset((void *)p3, 0, PAGE_SIZE);
                free_page((paddr_t)p3);
                PGT_INVALIDATE(p4[PGT_PML4_INDEX(vaddr)]);
        }

        /* On ne va pas enlever le PML4 : on a toujours les adresses noyau */

        /* On invalide l'entrée du TLB idoine */
        invlpg(vaddr);
}
```

Cette solution règle le problème : on a bien un succès sur la tâche `adversary()`.