Université de Lille 1 SCIENCES ET TECHNOLOGIES

Master d'informatique
1re année
Module d'architecture des systèmes d'exploitation

Virtualisation de la mémoire du microprocesseur

Simon DUQUENNOY Gilles GRIMAUD Philippe MARQUET

Novembre 2009
Révision de décembre 2016

Ce document est disponible en ligne à partir de `www.lifl.fr/~marquet/ens/mmu/`. Cet accès en ligne autorise des copier/coller... ne vous en privez pas.

1 Unité de virtualisation de la mémoire

Le principe de l'unité de virtualisation de la mémoire est de permettre à un logiciel d'intercepter les accès à la mémoire centrale. Le mécanisme de mémoire virtuelle est un dispositif matériel qui se place en coupure entre le microprocesseur et la mémoire physique. Il donne ainsi au système d'exploitation le moyen

— d'une part de contrôler, éventuellement interdire, les accès à la mémoire physique, et
— d'autre part de remplacer l'adresse demandée par l'application (alors appelée adresse virtuelle) par une autre adresse (l'adresse physique) désignant une zone de la mémoire réelle.

La bibliothèque d'émulation matérielle `hardware` que nous utilisons est capable d'émuler le fonctionnement d'une MMU, *Memory Management Unit*. On activera la MMU en le précisant dans un fichier `hw_config.ini` de configuration du matériel :

```
MMU_ENABLE      = 1
```

La MMU activée, les défauts d'accès à la mémoire sont reportés sous forme d'interruptions (comme dans un microprocesseur disposant d'une MMU). Le niveau d'interruption associé à la MMU, et le registre matériel contenant l'adresse qui a généré la faute sont précisés dans le fichier de configuration :

```
MMU_IRQ         = 13
MMU_FAULT_ADDR  = 0xCD
```

Le matériel émulé gère dans l'espace d'adressage disponible deux zones de mémoire distinctes.

La première zone commence à l'adresse `physical_memory` et correspond à la zone de mémoire physique. La seconde zone commence à l'adresse `virtual_memory` et correspond à l'espace d'adressage virtuel. La table 1 décrit plus précisément ces deux zones de mémoire.

TABLE 1 – Zones de mémoire physique et virtuelle

	Mémoire physique	Mémoire virtuelle
taille des pages	`PAGE_SIZE: 4 ko`	`PAGE_SIZE: 4 ko`
nombre de pages	`PM_PAGES: 1<<8` `PM_PAGES: 256`	`VM_PAGES: 1<<12` `VM_PAGES: 4096`
taille	`PM_SIZE: PM_PAGES * PAGE_SIZE` `PM_SIZE: 1 Mo`	`VM_SIZE: VM_PAGES * PAGE_SIZE` `VM_SIZE: 16 Mo`
adresse de début	`physical_memory`	`virtual_memory`
adresse de fin	`physical_memory + PM_SIZE - 1`	`virtual_memory + VM_SIZE - 1`

Cette configuration est proche de celles des microprocesseurs MIPS et ARM. La mémoire physique et la mémoire virtuelle coexistent dans l'espace d'adressage du microprocesseur. Cependant, le microprocesseur distingue deux modes de fonctionnement.

— En mode *maître*, le microprocesseur peut accéder à la mémoire physique, mais l'accès à un espace d'adressage virtuel (ou à toute autre zone de mémoire non définie) est interdit, il génère une interruption MMU.

— À l'inverse, en mode *utilisateur*, le microprocesseur peut accéder à la mémoire virtuelle (plus précisément aux pages de mémoire virtuelle autorisées par la MMU), mais l'accès à la mémoire physique (ou à toute autre zone de mémoire non définie) génère une interruption.

Notons encore que lorsqu'une instruction provoque une interruption MMU, l'instruction n'est pas exécutée par le microprocesseur. Aussi, une fois le traitement de l'interruption terminé, si un simple retour d'interruption est produit, le microprocesseur re-exécute l'instruction fautive.

Nous allons explorer plusieurs utilisations possibles de ce mécanisme de MMU.

2 Protection de la mémoire

Accès mémoire illégal

Vous avez tous expérimenté le fait que dans un processus Unix, une tentative d'accès à l'adresse 0 provoque l'apparition d'un message de type *segmentation fault*. Ce message signifie que le processus courant a tenté d'accéder à une adresse mémoire à laquelle il n'a pas légitimement accès ; ce peut être l'adresse 0 ou une autre adresse.

Si l'accès à un pointeur nul génère une erreur de segmentation, ce n'est pas parce que le programme est incorrect mais parce qu'il tente d'accéder à une zone de la mémoire que le système d'exploitation ne lui a pas confié. Ainsi, le noyau d'un système d'exploitation peut (et est parfois amené à) écrire à l'adresse 0 de la mémoire physique, et le compilateur C n'interdit pas, *a priori*, cette opération.

Exercice 1 (Accès illégal ?)
Question 1.1 Décrivez le comportement du programme suivant :

```
static void
mmuhandler()
{
    printf("tentative d'accès illégal à l'adresse %p\n",
           _in(MMU_FAULT_ADDR));
}

int
main(int argc, char **argv)
{
    char *ptr;

    ... /* init_hardware() */
    IRQVECTOR[MMU_IRQ] = mmuhandler;
    _mask(1);

    ptr = (char*)0;
    *ptr = 'c';
}
```

□

Question 1.2 Proposez une correction du programme précédent de telle sorte qu'il se comporte correctement lorsqu'une erreur d'accès à la mémoire se produit. □

Modes maître et utilisateur

Le mode maître et le mode utilisateur sont deux états de fonctionnement possibles des microprocesseurs permettant l'isolation des fautes et de la mémoire entre différents processus.

En mode maître, seule la mémoire physique est accessible, alors qu'en mode utilisateur c'est la mémoire virtuelle qui est accessible. Le début de la mémoire physique est occupé par le vecteur d'interruption (que nous utilisons sous le nom de variable `IRQVECTOR`).

Après l'initialisation du matériel réalisée par la fonction `init_hardware()`, le microprocesseur est en mode maître.

À chaque fois qu'une interruption est appelée, le microprocesseur exécute le code de l'interruption en mode maître.

Le bit 12 du registre de masque indique si le microprocesseur est en mode maître ou utilisateur. La valeur 0 est associée au mode maître, la valeur 1 au mode utilisateur.

À titre d'exemple, l'appel suivant

```
_mask(0x1001);
```

— demande à passer en mode utilisateur (bit 12 à 1), et
— autorise toutes les interruptions matérielles (octet de poids faible à 1).

Exercice 2 (Passage en mode utilisateur !)
Question 2.1 Quel est le comportement du programme ci-dessous ?

```
static void
mmuhandler()
{
    printf("tentative d'accès illégal à l'adresse %p\n",
            _in(MMU_FAULT_ADDR));
}

int
main(int argc, char **argv)
{
    ... /* init_hardware() */
    IRQVECTOR[MMU_IRQ] = mmuhandler;
    _mask(0x1001);
    IRQVECTOR[MMU_IRQ] = mmuhandler;
}
```

□

Question 2.2 Quelles sont les valeurs possibles de `ADDRESS` qui mènent à terminaison sur un succès ?

```
#define ADDRESS  ...

static void
mmuhandler()
{
    printf("tentative d'accès illégal à l'adresse %p\n",
            _in(MMU_FAULT_ADDRESS));
    exit(EXIT_FAILURE);
}
```

```
int
main(int argc, char **argv)
{
    char *ptr;

    ... /* init_hardware() */
    IRQVECTOR[MMU_IRQ] = mmuhandler;
    _mask(0x1001);

    ptr = ADDRESS;
    *ptr = 'c';

    exit(EXIT_SUCCESS);
}
```

□

3 Isolation mémoire de processus

Une MMU est couramment utilisée pour isoler la mémoire de différents processus à l'aide d'un mécanisme de TLB, *Tranlation Lookaside Buffer*. L'isolation mémoire permet à un processus d'accéder à *sa* mémoire et lui interdit d'accéder à la mémoire des autres processus.

La MMU émulée par notre bibliothèque `hardware` fonctionne par simple usage d'une TLB. La TLB contient `TLB_ENTRIES` entrées, 32 par défaut. Chaque entrée décrit la correspondance entre une page virtuelle et une page physique, ainsi que les droits d'accès à la page.

Exercice 3 (Mapping statique)
Nous souhaitons donc partager N pages de mémoire physique (N est considéré pair) entre deux processus utilisateurs. Chacun des deux processus adresse la mémoire virtuelle pour accéder à une matrice sur laquelle il effectue quelques calculs.

Question 3.1 Proposez un *mapping* de N/2 pages pris dans l'espace d'adressage virtuel, placé dans l'espace physique lorsque le premier processus s'exécute, puis un autre *mapping* pour les mêmes N/2 pages de mémoire virtuelle vers N/2 pages de mémoire physique différentes. □

On rappelle que les pages virtuelles comme physiques sont de taille 4 ko (soit 2^{12}). Le numéro de page correspondant à une adresse peut donc être obtenu en ignorant les 12 bits de poids faible de l'adresse.

Question 3.2 Implémentez une fonction :

```
static int ppage_of_vpage(int process, unsigned vpage);
```

qui retourne le numéro de la page physique associée à la page virtuelle `vpage` du process `process` (0 pour le premier processus, 1 pour le second). La fonction retourne -1 si l'adresse de page virtuelle est en dehors de l'espace alloué au processus. □

Quand une faute d'accès se produit, la MMU déclenche une interruption. Lorsque le noyau traite l'interruption, il peut typiquement

— interrompre l'exécution du processus à l'origine de la faute (indiquant une erreur de segmentation), ou
— ajouter la page de l'adresse fautive dans la TLB. L'instruction fautive est alors rejouée, mais ne déclenche plus de faute d'accès.

À un instant donné, les données de la TLB ne sont valides que pour le processus qui est en train de s'exécuter. La TLB sert de cache de translation à la MMU. À chaque changement

TABLE 2 – Format des entrées de la TLB (par ordre décroissant de poids de bits)

taille	contenu
8 bits	R.F.U.
12 bits	page virtuelle
8 bits	page physique
1 bit	accès en exécution
1 bit	accès en écriture
1 bit	accès en lecture
1 bit	entrée utilisée ou non

TABLE 3 – Ports et commandes de gestion de la MMU

port	R/W	commande ou données	objet
`MMU_CMD`	W	`MMU_PROCESS` `MMU_RESET`	désactive / active la MMU réinitialise la MMU
`MMU_FAULT_ADDR`	R	adresse mémoire	adresse fautive courante
`TLB_ADD_ENTRY`	W	entrée TLB	entrée à ajouter dans la TLB
`TLB_DEL_ENTRY`	W	entrée TLB	entrée à supprimer de la TLB
`TLB_ENTRIES`	R/W	`TLB_ENTRIES` × entrée TLB	ensemble des entrées de la TLB

de contexte, le noyau doit donc vider la TLB, car les *mappings* qu'elle contient ne seront plus valables après le changement de contexte.

Le tableau 2 décrit la structure des entrées de la TLB. Chaque entrée a une taille de 32 bits.

Exercice 4 (Entrées de la TLB)
En concordance avec le tableau 2, définissez un type `struct tlb_entry_s` qui sera utilisé pour représenter une entrée de la TLB. □

En mode maître, la MMU peut être pilotée *via* des lectures ou écritures (`_in()` et `_out()`) sur les ports de contrôle de la MMU. Le tableau 3 décrit les ports et commandes permettant de gérer la MMU et sa TLB. Ces commandes prennent effet directement (donc aucune interruption n'est déclenchée pour en notifier la terminaison).

Exercice 5
Écrivez un *handler* pour l'interruption `MMU_IRQ` capable de mettre en œuvre le *mapping* décrit précédemment. Une variable globale `current_process` indique le numéro du processus qui s'exécute. Lorsqu'une faute se produit, il est nécessaire d'ajouter dans la TLB une nouvelle entrée correspondant à la page accédée, ou de lever une erreur de segmentation. □

Interruptions logicielles

La bibliothèque `hardware` permet également de manipuler des interruptions logicielles. Comme vu précédemment, les 16 premières entrées du vecteur d'interruption `IRQVECTOR[]` sont associées à des interruptions matérielles. En fait, ce vecteur est de taille `IRQ_VECTOR_SIZE`, soit 256. Les interruptions suivantes sont dédiées aux interruptions logicielles.

Les interruptions logicielles peuvent être déclenchées par un processus utilisateur. Elles permettent aux processus de demander au noyau d'exécuter une tâche, comme par exemple accéder aux I/O du système. C'est ce mécanisme qui permet l'implémentation d'appels systèmes.

Le déclenchement d'interruptions logicielles se fait *via* la commande `_int()`. Par exemple, `_int(23)` va déclencher l'interruption 23. La fonction présente en position 23 dans le vecteur d'interruptions est alors exécutée en mode maître. Quand elle retourne, l'exécution reprend après l'instruction `_int()`.

Exercice 6 (Programme utilisateur et code système)
Question 6.1 Quel est le comportement du programme suivant ? □

```
static int current_process;

static int
sum(void *ptr)
{
    int i;
    int sum = 0;

    for(i = 0; i < PAGE_SIZE * N/2 ; i++)
        sum += ((char*)ptr)[i];
    return sum;
}

static void
switch_to_process0(void)
{
    current_process = 0;
    _out(MMU_CMD, MMU_RESET);
}

static void
switch_to_process1(void)
{
    current_process = 1;
    _out(MMU_CMD, MMU_RESET);
}

int
main(int argc, char **argv)
{
    void *ptr;
    int res;

    ... /* init_hardware(); */
    IRQVECTOR[16] = switch_to_process0;
    IRQVECTOR[17] = switch_to_process1;
    _mask(0x1001);

    ptr = virtual_memory;

    _int(16);
    memset(ptr, 1, PAGE_SIZE * N/2);
    _int(17);
    memset(ptr, 3, PAGE_SIZE * N/2);

    _int(16);
    res = sum(ptr);
    printf("Resultat du processus 0 : %d\n",res);
    _int(17);
    res = sum(ptr);
    printf("Resultat processus 1 : %d\n",res);
}
```

Manipulation pratique

On retrouvera le code présenté question 6.1 dans les fichiers `misc/mi.{c,h}` du dépôt

`https://gitlab-etu.fil.univ-lille1.fr/ms1-ase/src`

On y ajoutera le code du *handler* défini à l'exercice 5.
Comme décrit à la question 6.2, on séparera le code noyau et le code utilisateur.

Question 6.2 On séparera en deux fichiers sources `mi_kernel.c` et `mi_user.c` les fonctions et séquences de code exécutées en mode maître et utilisateur.

Le point d'entrée sera le « boot » du système, en mode noyau. Ce code noyau consistera en une initialisation du matériel et du vecteur d'interruptions. Il passera ensuite la main au code utilisateur par l'appel d'une fonction `init()` qui passera alternativement à l'exécution du processus 0 et du processus 1.

Ces deux fichiers se partageront un fichier d'entête `mi_syscall.h` qui identifie les « appels systèmes » possibles :

```
#define SYSCALL_SWTCH_0 16
#define SYSCALL_SWTCH_1 17
```

et identifie le point d'entrée du fonctionnement en mode utilisateur :

```
void init(void);
```

□

4 Mémoire virtuelle et swap disque

Le mécanisme de *swap* disque autorise la manipulation par un processus utilisateur d'un espace d'adressage mémoire de taille supérieure à l'espace mémoire physique disponible sur le matériel. Le principe est d'utiliser une *partition de swap* pour étendre l'espace mémoire physique.

Nous allons illustrer ce mécanisme sur des processus menant des calculs sur des matrices de très grande taille. Voir l'encart page 9.

Dans un premier temps, nous allons présenter les fonctions utiles à la gestion de la partition de swap. Nous utiliserons ensuite ces fonctions pour supporter une mémoire virtuelle aussi grande que possible, malgré la taille réduite de la mémoire physique. Nous utiliserons enfin cette mémoire virtuelle pour différents algorithmes et allocations mémoire de matrices.

Gestion du swap disque

Une partition de swap est un ensemble de blocs pouvant être directement lus ou écrits sur un disque. À un bloc correspond une page virtuelle. La taille de la partition est égale à la taille de la plage mémoire adressable virtuellement.

On ne dispose d'une unique partition de swap. Les deux fonctions utilitaires suivante assure transfert de données entre le swap et la mémoire physique :

```
int store_to_swap(int vpage, int ppage);
int fetch_from_swap(int vpage, int ppage);
```

— la fonction `store_to_swap()` stocke dans le bloc ad hoc du swap la page de numéro `vpage` dont le contenu est actuellement stocké dans la page physique `ppage`;
— la fonction `fetch_from_swap()` lit dans le fichier de swap la page virtuelle numéro `vpage` et en écrit le contenu dans la page physique `ppage`.

Les fonctions retournent une valeur négative en cas d'échec, 0 en cas de succès.

Ces fonctions doivent bien entendu être exécutées en mode maître du microprocesseur.

On trouvera une implémentation de ces fonctions dans le fichier `misc/swap.c` accessible sur le dépôt

```
https://gitlab-etu.fil.univ-lille1.fr/ms1-ase/src
```

Première virtualisation de la mémoire avec swap

On se propose maintenant d'implémenter un nouveau handler d'interruption de la MMU capable de gérer un grand nombre de pages virtuelles dans une petite mémoire physique, à l'aide du swap disque. Dans un premier temps, la politique de remplacement est très simple : elle n'utilise qu'une seule page de mémoire physique. Toutes les pages de mémoire virtuelle sont *mappées* sur la page de mémoire physique numéro 1 (et pas la page 0 qui contient le vecteur d'interruptions).

Le handler d'interruptions associé à la MMU est donc appelé quand l'adresse accédée est fautive. Si elle est dans l'espace d'adressage virtuel, comme elle est fautive, elle ne correspond pas à la page qui est en actuellement en mémoire physique. On sauvegarde donc cette page dans le fichier de swap et on supprime le mapping dans la TLB pour cette page. Il s'agit ensuite de charger la page correspondant à l'adresse fautive et de mettre à jour la TLB en conséquence.

Exercice 7 (Premier handler avec swap)
Implémentez le nouveau handler d'interruptions gérant une unique page de mémoire physique à l'aide du fichier de swap. □

Virtualisation de la mémoire avec swap

La première implémentation fournie est très lente car elle sous-exploite la mémoire physique disponible. À un instant donné, seule une page de données est stockée en mémoire. Toutes les autres données sont placées dans le fichier swap.

Nous allons maintenant étendre ce mécanisme afin d'utiliser toute la mémoire physique dont on dispose. Quand on a besoin d'une page de mémoire physique pour réaliser le mapping d'une nouvelle page de mémoire virtuelle, on vide une page de mémoire physique sur le fichier de swap.

On choisira la page de mémoire physique à vider sur le fichier de swap par un simple algorithme tourniquet (*round robin*).

Il est nécessaire de mémoriser les correspondances entre les pages virtuelles et les pages physiques, correspondances qui évoluent au fur et à mesure des accès mémoire réalisés. La TLB ne mémorisant que les associations récemment utilisées sous la forme d'un cache, nous nous devons de définir une structure permanente pour mémoriser ces associations. Il nous sera nécessaire d'accéder à cette information à la fois à partir de l'adresse de page virtuelle, mais aussi à partir de l'adresse de page physique.

Exercice 8 (Tables des pages mappées)
Définissez deux tables `vm_mapping[]` et `pm_mapping[]` permettant respectivement de retrouver pour toute page virtuelle la page physique associée, ou pour toute page physique la page virtuelle associée et de déterminer si la page est en mémoire ou doit être rechargée depuis le fichier de swap. □

L'implantation du handler d'interruptions de la MMU consiste maintenant à choisir, si nécessaire, une nouvelle page de mémoire physique pour la page de mémoire virtuelle fautive, à maintenir les deux tableaux `vm_mapping[]` et `pm_mapping[]`, et à renseigner la TLB avec l'association résultante. On prendra garde à ne pas utiliser la page 0 de mémoire physique.

Multiplication de « grandes » matrices

Nous souhaitons donc manipuler de « grandes » matrices, c'est-à-dire des matrices dont la taille est supérieure à celle de la mémoire physique dont on dispose.
Nous manipulons des matrices carrés de dimension N. Pour ranger 3 matrices carrées, il nous faut $3 \times N^2 \times$ `sizeof(unsigned)` octets de mémoire.
On vérifiera que la valeur de N (`MATRIX_SIZE`) définie assure en effet une utilisation mémoire de taille supérieure à la mémoire physique disponible.
Le code d'initialisation, addition, et multiplication de matrices vous est fourni. Le dépôt

```
https://gitlab-etu.fil.univ-lille1.fr/ms1-ase/src
```

contient dans `matmul` une base de travail pour le TP sur la virtualisation avec swap. Le `Makefile` construit deux binaires :

- Le programme `mmu_manager` issu de `mmu_manager.c` est un squelette que vous devrez compléter. Pour l'instant, il se contente d'appeler la fonction `user_process()` fournie dans le fichier `user_process.c`. Cette fonction doit être appelée en mode utilisateur du CPU, et se charge d'additionner ou de multiplier deux matrices de grande taille. Vous serez amenés à modifier les constantes définies dans `matrix.h` et `user_process.c` pour paramétrer la taille des matrices et le type de calcul à réaliser.
- Le programme `oracle` se charge de vérifier que le résultat du calcul matriciel est correct, afin de s'assurer que la mémoire virtuelle fonctionne correctement, sans compromettre les calculs. Ce programme lit sur l'entrée standard la sortie de la fonction `user_process`. Il effectue, sans émulation de MMU, les mêmes calculs matriciels, puis affiche OK ou KO après avoir comparé ses résultats aux résultats de `user_process`.

Le programme `oracle` analysant le résultat affichée par votre programme sur la sortie standard, ne modifiez par ce résultat, préférez donc l'utilisation de `fprint(stderr, ...)` à de simples `printf()`.
Pour valider vos développements, utilisez par exemple

```
./mmu_manager | ./oracle
```

afin de lancer les calculs, puis d'en vérifier les résultats.
On pourra aussi utiliser

```
./mmu_manager | tee /dev/stderr | ./oracle
```

pour dupliquer la sortie standard vers le terminal et la commande `oracle`.

Exercice 9 (Handler avec swap)
Implémentez le nouveau handler d'interruptions exploitant toute la mémoire physique disponible à l'aide du fichier de swap. □

Programmeur, le matériel tu n'oublieras point !

On s'intéresse à la manipulation de la matrice suivante :

```
char matrix[N][N];
```

Exercice 10 (Défauts de page ?)
Avec des pages de taille `P`, combien de défauts de page se produisent lors de l'exécution du code suivant ?

```
int i, j;

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        matrix[i][j] = (i + j) % 128;
```

□

Exercice 11 (Inversion de boucles)
On change légèrement le code précédent en accédant au tableau par simple échange de l'ordre d'imbrication des boucles. Comment cela affecte-t-il le nombre de défauts de pages ?

```
int i, j;

for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
        matrix[i][j] = (i + j) % 128;
```

□

Rappelons que la multiplication de matrices $C = A \times B$ « bête et méchante » correspond au code

```
int i, j;

for (i = 0; i < N; j++)
    for (j = 0; j < N; i++)
        for (k = 0; k < N; i++)
            C[i][j] += A[i,k] + B[k,j];
```

Exercice 12 (Allocation pour la multiplication de matrices)
En s'inspirant des exercices précédents, proposez un moyen d'accélérer la multiplication de matrices dans notre mémoire virtuelle. □

fin du document (pour cette année...)