Contrôle Final 2 - CSC3101 - 19 Mars 2024 - 1h30

Tout document écrit est autorisé. Pour répondre à une question, vous donnez le code correspondant (évitez les répétitions). Par défaut, la visibilité des méthodes est `public` *et celle des attributs est* `private`*. Il n'y a pas à écrire les directives* `import` *et* `package`*.*

Un système d'exploitation multi-tâches exécute plusieurs programmes en même temps. Pour réaliser ceci, le système d'exploitation virtualise la mémoire, c'est-à-dire que chaque programme a l'illusion qu'il est seul à travailler. L'unité de gestion de la mémoire, ou Memory Management Unit (MMU) en anglais, effectue la virtualisation. Dans cet examen, on va implémenter une MMU en Java.

1. La mémoire physique *(~15 minutes, 4pt)*

La mémoire de l'ordinateur est organisée comme un grand tableau (voir Figure 1). Chaque entrée dans ce tableau est appelée un *mot mémoire*. L'index de l'entrée est l'*adresse* du mot. Pour faire un calcul, le CPU récupère un mot mémoire stocké à une certaine adresse. Par exemple, dans la Figure 1, le CPU peut demander le mot à l'adresse 32, celui à l'adresse 33, faire un 'ou' logique entre eux, puis stocker le résultat à l'adresse 35. La taille d'un mot mémoire varie en fonction de l'architecture. Dans cet examen, on va considérer des mots de taille 32 bits (4 octets), à savoir la taille d'un entier en Java.

Avant de se lancer dans le code, on définit quelques constantes utiles.

```
class Constants {
    public static final int MEMORY_SIZE = 256 * 256;
    public static final int BLOCK_SIZE = 4;
    public static final int MAX_NB_PROCESSES = 4;
    public static final int MAX_NB_PAGES_PER_PROCESS = 256;
}
```

Par la suite, on considère que ces constantes sont importées. Ainsi, on écrira `MAX_NB_PROCESSES` pour la constante définissant le nombre maximal de processus gérés par le système d'exploitation.

Dans un premier temps, on s'intéresse à modéliser la mémoire physique de l'ordinateur. Pour ce faire, on va écrire une classe `PhysicalMemory`. Cette classe contient le tableau de tous les mots mémoires. Ce tableau est appelé `content`, et il est l'unique champ de la classe `PhysicalMemory`.

[Q1a] *(1pt)* Donnez le code du constructeur de la classe `PhysicalMemory`. Ce constructeur initialise `content` à un tableau d'entiers de taille `MEMORY_SIZE`.

[Q1b] *(1pt)* La mémoire permet de lire et d'écrire des mots mémoires. Pour ce faire elle offre deux méthodes : `int readWord(int position)` lit le mot mémoire à l'entrée `position`, et `void writeWord(int position, int word)` écrit le mot `word` à l'entrée `position`. Écrivez le code de ces méthodes.

[Q1c] *(2pt)* On souhaite se prémunir d'erreurs utilisateurs. Pour ce faire, on va ajouter une vérification de la position passée en argument. Si la position est négative ou bien supérieure ou égale à la constante `MEMORY_SIZE`, on lève une exception `RuntimeException`. Quelle modification doit-on faire à la méthode `readWord` ? (Pour rappel, la classe `RuntimeException` existe déjà dans Java.)

2. La MMU *(~15 minutes, 4pt)*

Une MMU gère la virtualisation de la mémoire d'un processus. Elle répond à l'interface suivant :

```
interface MMU {
    void malloc(int pid, int size);
    void write(int pid, int addr, int word);
    int read(int pid, int addr);
}
```

La première méthode de cet interface permet à un processus de réserver de la mémoire. La taille est passée en nombre de mots mémoire. Le paramètre `pid` (pour "process identifer" en anglais) correspond à l'identifiant du processus faisant l'appel. Les deux autres méthodes de l'interface `MMU` permettent de lire et écriture à une adresse virtuelle. Le paramètre `addr` est l'adresse virtuelle où est fait l'accès, et `word` correspond au mot mémoire à écrire.

[Q2a] *(1pt)* On va coder la MMU dans une classe `SimpleMMU`. Cette classe implémente l'interface `MMU` ci-dessus. Écrire une classe `SimpleMMU` vide.

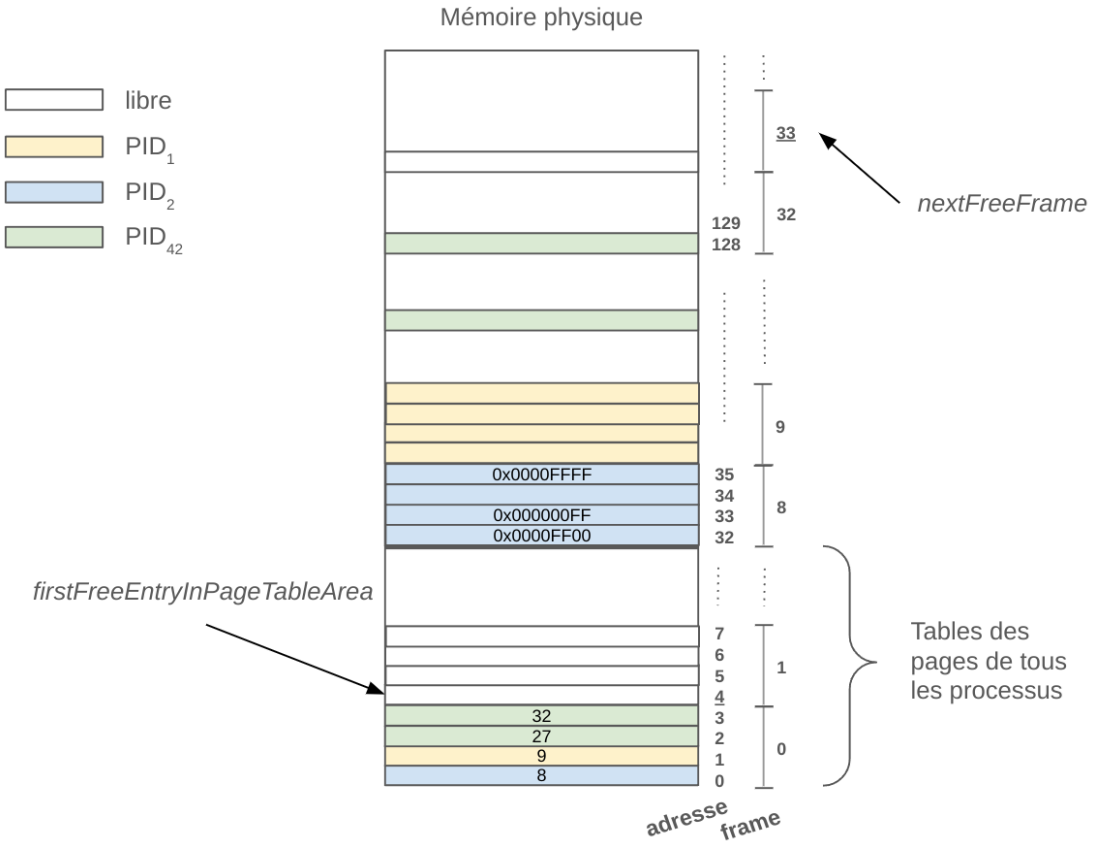

Figure 1: Organisation de la mémoire

La virtualisation de la mémoire est faite par *traduction* entre adresse virtuelle et adresse physique. Par exemple, dans la Figure 1, le processus de PID 2 (en bleu) a réservé 4 mots mémoire. Ces mots sont stockés aux adresses 32-35 par la `MMU`. L'appel à `write(2, 1, 0x000000FF)` écrit `0x000000FF` (soit 255) à l'adresse virtuelle 1 du processus 2. La MMU traduit l'adresse virtuelle 1 en adresse physique 33. Un exemple complet d'utilisation de la `MMU` est donné à la fin de l'examen.

[Q2b] *(3pt)* Dans `SimpleMMU`, donnez le code des méthodes `read` et `write` de l'interface `MMU`. Pour ce faire, vous supposerez l'existence d'un champ `PhysicalMemory memory` qui référence la mémoire physique. On postulera aussi une méthode `private int getPhysicalAdress(int pid, int addr)`. Cette méthode renvoie l'adresse physique correspondant à l'adresse virtuelle `addr` du processus `pid`.

3. Les bases de la virtualisation mémoire *(~30 minutes, 6pt)*

La mémoire (physique ou virtuelle) est organisée en *blocs*. L'adresse d'un mot mémoire correspond à un bloc et son décalage (offset) dans le bloc. En d'autres termes, `addr = block * BLOCK_SIZE + offset`. Les

blocs de la mémoire physique sont appelés *frame*. Ceux de la mémoire virtuelle sont nommés *page*. Ici, on considère que frame et page stockent la même quantité de données, à savoir 4 mots (`BLOCK_SIZE`).

La MMU établit une *correspondance* entre page et frame afin de réaliser un accès (en lecture ou écriture). Pour chaque processus, cette correspondance est maintenue dans *la table des pages* du processus : la frame correspondant à la page k est stockée à l'entrée k de la table des pages. L'ensemble des tables des pages de tous les processus est stocké au début de la mémoire physique.

Cette organisation est illustrée en Figure 1. Le processus de PID 2 a alloué un seul bloc. Ce bloc est stocké dans la frame 8 de la mémoire. Sa table des pages est donc de taille 1. Elle est stockée au début de la mémoire, dans la frame 0, conjointement avec la table des processus de PID 1 (en jaune) et 42 (en vert). Le rôle de `nextFreeFrame` et `firstFreeEntryInPageTableArea` est détaillé plus tard.

[**Q3a**] *(1pt)* D'après la description ci-dessus, la table des pages a besoin d'un mot mémoire pour chaque page. Ce mot est utilisé pour stocker la frame correspondant à la page. Il y a `MAX_NB_PROCESSES` processus au maximum. Pour chaque processus, on a au plus `MAX_NB_PAGES_PER_PROCESS` pages. Écrivez la formule donnant le nombre de mots mémoire pour stocker toutes les tables des pages au début de la mémoire ?

[**Q3b**] *(1pt)* Combien faut-il de frames pour stocker cette quantité d'information ? (Pour rappel, `Math.ceil(double x)` fait un arrondi supérieure du flottant `x` passé en argument.)

Outre le champ `memory`, la classe `SimpleMMU` possède trois champs : `int nextFreeFrame` indique la prochaine frame libre en mémoire physique après la table des pages, `int firstFreeEntryInPageTableArea` est la prochaine entrée en mémoire pour stocker une table des pages, et `Map<Integer, Integer> pageTableBase` indique le début de la table des pages des processus (non illustrée en Figure 1).

On va maintenant coder le constructeur de `SimpleMMU`. Ce constructeur prend en paramètre une instance de la mémoire physique (`PhysicalMemory`), affectée au champ `memory`. Puis il initalise `pageTableBase`, `firstFreeEntryInPageTableArea` et `nextFreeFrame` convenablement. À savoir,
- `pageTableBase` est affecté à une nouvelle instance d'un objet implémentant `Map` de votre choix,
- `firstFreeEntryInPageTableArea` est mis à zéro, et
- `nextFreeFrame` stocke la prochaine frame libre en mémoire physique (en utilisant le résultat de Q3a).

[**Q3c**] *(2pt)* Donnez le code du constructeur de `SimpleMMU` en suivant la logique ci-dessus.

[**Q3d**] *(2pt)* Écrivez le code de la méthode privée `findAndTakeFreeFrame`. Cette méthode renvoie une frame libre en mémoire. À savoir, on retourne la valeur de `nextFreeFrame` avant de l'avoir dûment incrémenté. Vous vérifierez que l'on n'a pas déjà utilisé le nombre maximal de frames en mémoire (donné par `MEMORY_SIZE/BLOCK_SIZE`). Si c'est le cas, on lève une exception de type `RuntimeException`.

4. Mécanismes de virtualisation *(~30 minutes, 6+1pt)*

Dans cette section, nous allons écrire la méthode de traduction vers une adresse physique (`getPhysicalAdress`), et celle allouant de la mémoire pour un processus donné (`malloc`).

[**Q4a**] *(1pt)* Supposons que `addr` est une adresse mémoire virtuelle. Quel est la page où réside `addr` ?

Comme indiqué plus haut, `pageTableBase.get(p)` indique le début de la table des pages du processus `p`. La frame associée à la page `k` de ce processus est donc stockée en mémoire physique à l'adresse `pageTableBase.get(p) + k`. Par exemple dans la Figure 1, pour le processus 2 `pageTableBase.get(2)` retournerait 0. La frame associée à la page 0 du processus 2 est donc stockée dans la première entrée de la mémoire physique (`pageTableBase.get(2) + 0`). Il s'agit de la frame 8.

[**Q4b**] *(2pt)* Comment peut-on calculer la frame correspondant à `addr` ? Par la même approche, trouvez l'offset de `addr` dans cette frame. Enfin, en déduire l'adresse physique de `addr`. (N'oubliez pas ici que page et frame font la même taille, donc les offsets sont identiques.)

[**Q4c**] *(2pt)* Implémentez la méthode privée `int getPhysicalAdress(int pid, int addr)`. Cette méthode doit d'abord vérifier que le processus est enregistré dans `pageTableBase` avec la méthode `boolean`

`containsKey(K k)` de `Map`. (Pour rappel, `containsKey` indique la présence dans la table de la clé `k`.) Puis, elle vérifie que l'adresse passée en paramètre est bien dans l'espace maximale d'adressage (`MEMORY_SIZE`). Si une de ces vérifications échoue, on lève une exception `RuntimeException`. Ensuite, l'adresse physique est retournée en utilisant la réponse à la question Q4b.

La méthode `malloc` (pour "memory allocate" en anglais) permet d'allouer de la mémoire virtuelle à un processus. Pour ce faire, on doit définir un endroit où stocker la table des pages du processus et associer chacune des pages à une frame. En détail, on applique la logique suivante :
- on associe le processus à la valeur de `firstFreeEntryInPageTableArea` dans `pageTableBase`,
- on calcule le nombre de pages requises (arrondi au supérieur),
- on incrémente `firstFreeEntryInPageTableArea` de la taille de la table des pages du processus, puis
- on assigne une nouvelle frame à chacune des pages en utilisant `findAndTakeFreeFrame`.

[**Q4d**] *(1pt)* Donnez le code de `malloc`. Quel est la complexité de votre méthode en fonction des paramètres définis dans la classe `Constants`.

[**Q4e**] *(1pt, bonus)* La méthode `void free(int pid)` libère la mémoire du processus. Elle remet à zéro la table des pages du processus puis retire ce dernier de `pageTableBase`. Écrivez le code de cette méthode. Quel problème voyez-vous sur le long terme lorsque les processus allouent puis désallouent de la mémoire avec `SimpleMMU` ? Proposez une solution sans la codez.

5. Exemple d'utilisation

Un exemple d'utilisation de la MMU est donné ci-dessous. Dans ce programme, deux processus d'identifiants 1 et 2 allouent chacun 4 mots mémoire. Le processus 2 fait des écritures et vérifie que ce qui a été écrit est bien en mémoire. Le programme termine par une désallocation et s'assure que cette dernière fonctionne correctement. Dans ce code, la méthode `Math.ceil` permet de faire un arrondi supérieur. Le mot clé `assert` lève une exception si l'assertion en argument est évaluée à faux.

```
public static void main(String[] args) {
    PhysicalMemory mem = new PhysicalMemory();
    SimpleMMU m = new SimpleMMU(mem);

    // allocate the memory
    m.malloc(2, 4);
    m.malloc(1, 4);

     // do some writes and check them
    int w1 = 0x0000FF00; // 65280
    int w2 = 0x000000FF; // 255
    m.write(2, 0, w1);
    m.write(2, 1, w2);
    int w3 = w1 | w2; // 65535
    m.write(2, 3, w3);
    assert m.read(2, 0) == w1;
    assert m.read(2, 3) == w3;

    // verify that free() works properly
    m.free(1);
    int beg = 1; // skip the page table of process 2
    int requiredPages = (int) Math.ceil((double) 4 / (double) Constants.BLOCK_SIZE);
    int end = beg + requiredPages;
    for (int k=beg; k<end; k++) { assert m.memory.readWord(k) == 0; }
}
```