M3 - Kernel Programming

Nicolas CORMIER
Epitech 2008

Dans le cadre de la réalisation d'un début de micro kernel en mode protégé sur processeur *x86*, ce document décrit l'architecture du noyau ainsi que les choix techniques réalisés.

Introduction

Dans le cadre du module de kernel programming de cinquième année de l'Epitech, réalisation d'un micro kernel sur processeur *x86*.
Ce document décrit les choix architecturaux et techniques qui guideront l'implémentation du micro kernel.
L'implémentation se fera en 5 étapes : le bootstrap, le gestionnaire d'interruption, le gestionnaire de mémoire, le gestionnaire de tâches et enfin le scheduler.
Nous allons décrire dans les sections suivantes l'architecture de chacune de ces 5 étapes.

1 Bootstrap

A écrire ...

2 Gestionnaire d'interruption

A écrire ...

3 Gestionnaire de mémoire

Le gestionnaire de mémoire est le coeur du système d'exploitation, nous allons ici décrire les problématiques à résoudre et les choix techniques et architecturaux qui en découlent.

3.1 Problématiques et buts à atteindre

Un gestionnaire de mémoire moderne se doit de prendre en charge les points suivants:

Partager et maintenir les ressources mémoires du système. Un ordinateur dispose de ressources mémoires limitées, il convient donc de réaliser l'organisation et le partage de ces ressources.

Fournir un mécanisme de gestion de contenu. La mémoire n'est qu'une coquille accueillant du contenu de sources diverses (anonyme, fichier, ...).
Pour faciliter et surtout centraliser les accès à ces différentes sources, le gestionnaire de mémoire doit offrir une interface de chargement, déchargement et sauvegarde de ces divers contenus.

Fournir un mécanisme de partage de contenu. Pour permettre à plusieurs tâches d'utiliser au même instant le même contenu aussi bien en écriture, qu'en lecture.

Partager au maximum les contenus. Il n'est pas rare que le contenu d'un fichier soit utilisé par plusieurs tâches au même moment, il est donc indispensable de partager implicitement ces contenus afin d'améliorer sensiblement les performances du système.

Attendre le dernier moment pour charger le contenu. Dans la majorité des cas, lorsqu'une tâche charge un fichier, elle n'en utilise qu'un fragment. Attendre l'accès pour charger la donnée permet d'optimiser et de sauver de la mémoire qui n'aurait potentiellement jamais servi.

Utiliser une mémoire secondaire. La mémoire principale (RAM) est dans la plupart des cas beaucoup plus limitée que la mémoire secondaire (ex: disque dur). Lorsque la mémoire principale vient à manquer, il est nécessaire, pour que le système puisse continuer à s'exécuter, d'utiliser la mémoire secondaire et ainsi soulager la principale.

Sécuriser l'accès en écriture et en lecture au contenu.

Exécuter les tâches dans une machine virtuelle. Ceci permet de:

a. Faire croire à la tâche qu'elle s'exécute seule sur la machine.
b. Lui masquer le reste du système (le matériel, les autres tâches en cours d'exécution, etc ...).
c. Lui permettre d'utiliser plus de mémoire qu'il en existe physiquement, en garantissant un espace d'adressage important et indépendant du nombre de tâches en cours d'exécution
d. Sécuriser l'accès aux données
e. Éviter de corrompre les données d'une autre tâche

en cours d'exécution

f. Permettre à deux tâches d'accéder à une même adresse mais à un contenu différent.

Offrir au noyau un mécanisme d'allocation optimisé. Tout code s'exécutant dans le noyau doit pouvoir allouer rapidement des objets de toutes tailles.

Avoir une architecture modulaire. Bien qu'une architecture monolithique soit dans la majorité des cas plus réactive qu'une architecture modulaire, il y a de nombreux avantages à choisir une architecture modulaire pour réaliser le gestionnaire de mémoire:

a. Permettre aux modules de s'exécuter dans une tâche tierce (plus performant)

b. Permettre de déporter du code noyau en code utilisateur (plus stable: le système ne crashera pas si le module crashe)

c. Plus de facilité dans le travail de portabilité: si le code dépendant du matériel est contenu dans un module, il suffit de remplacer ce module et uniquement ce module pour faire tourner le noyau sur une autre machine.

d. Segmenter le travail à réaliser et limiter les dépendances dans le code.

3.2 Choix techniques sur *x86*

Les derniers processeurs de la famille de *x86* offrent différents outils pour faciliter la tâche de gestion de la mémoire du système d'exploitation. Il s'agit de la segmentation et de la pagination.

3.2.1 Segmentation

Lorsque le processeur passe en mode protégé la segmentation est alors automatiquement activée.
La segmentation permet de définir des segments[1] personnalisés en terme de taille, de positionnement et de droits d'accès

Il est possible de fournir à la *MMU* différents descripteurs qui vont lui permettre de savoir comment interpréter les adresses logiques.
Il y a deux niveaux de segmentation : global et local. Ces deux niveaux sont représentés physiquement par deux tables de descripteurs, respectivement appelés la GDT et la LDT. En fonction du paramétrage courant de certains registres, la *MMU* va se servir des entrées de ces tables pour effectuer une traduction vers un adressage linéaire.
En changeant certains registres, on peut donc changer complètement la valeur linéaire d'une adresse logique.

Il y a typiquement[2] trois utilisations possibles de la segmentation :

a. le *Basic Flat Model* : chaque segment utilisé couvre la totalité de l'espace d'adressage linéaire.

b. le *Protected Flat Model* : le segment de données et le segment de code sont séparés en terme d'adressage.

c. Le *Multisegment Model* : *N* segments sont définis, le plus souvent un segment par programme ou librairie partagée...

3.2.2 Pagination

Il est possible d'activer la pagination en plus du mécanisme de segmentation.
Une fois l'adresse logique traduite en adresse linéaire par la segmentation, il y a une seconde phase de traduction lorsque la pagination a été activée.
La pagination offre la possibilité d'associer une plage contiguë d'adresses linéaires à une plage d'adresses physiques. La taille de cette plage peut varier selon le processeur et la configuration du processeur, typiquement sur x86 la plage contient 4096 adresses. On appelle cette plage d'adresses une « page frame ».
On peut en plus de cette association définir des droits d'accès à ces adresses.
Au même titre que la segmentation, la pagination est configurée via des registres, il est possible de changer à tout moment le comportement de la *MMU* en ne changeant qu'un registre.

3.2.2 Segmentation/Pagination

La segmentation et la pagination permettent d'associer une adresse « virtuelle » avec une adresse physique.

Les avantages de la segmentation :

a. Possibilité d'associer un segment avec 4 niveaux de privilèges, de 0 à 3.

b. Possibilité d'associer 3 types d'accès différents au segment : READ, WRITE et EXEC.

c. Possibilité d'associer un type au contenu du segment : CODE et DATA.

Les avantages de la pagination :

a. Association adresses linéaires – adresses physiques très fines.

b. Un registre suffit pour paramétrer la pagination.

1 On appelle segment un ensemble d'adresses contiguës.

2 D'après Intel

Notons que la pagination offre aussi un système de vérification d'accès (plus léger que la segmentation) : droit READ/WRITE sur une association, ainsi qu'un flag « propriétaire » USER/SUPERVISOR.

3.2.3 Choix et configuration

La pagination offre un mécanisme suffisant dans notre cas. Nous choisirons donc de masquer le mécanisme de segmentation en utilisant le *Basic Flat Model* et en activant la pagination.

Notre *GDT* contiendra dans un premier temps au minimum deux entrées :
Kernel Code Segment, couvre la totalité de l'espace d'adressage linéaire, son niveau de privilège est de 0 et son type est CODE.
Kernel Data Segment, couvre lui aussi la totalité de l'espace d'adressage linéaire, son niveau de privilège est de 0 et son type est DATA.

Ce modèle nous permet d'avoir un adressage virtuel différent par tâche.

3.3 Architecture

L'architecture du gestionnaire de mémoire doit prendre en compte les choix techniques tout en répondant aux problématiques exposées préalablement.

L'architecture est donc modulaire et offre à l'implémentation le choix du mode de communication inter-modules (direct ou via messages par exemple).

3.3.1 Vue globale de l'architecture

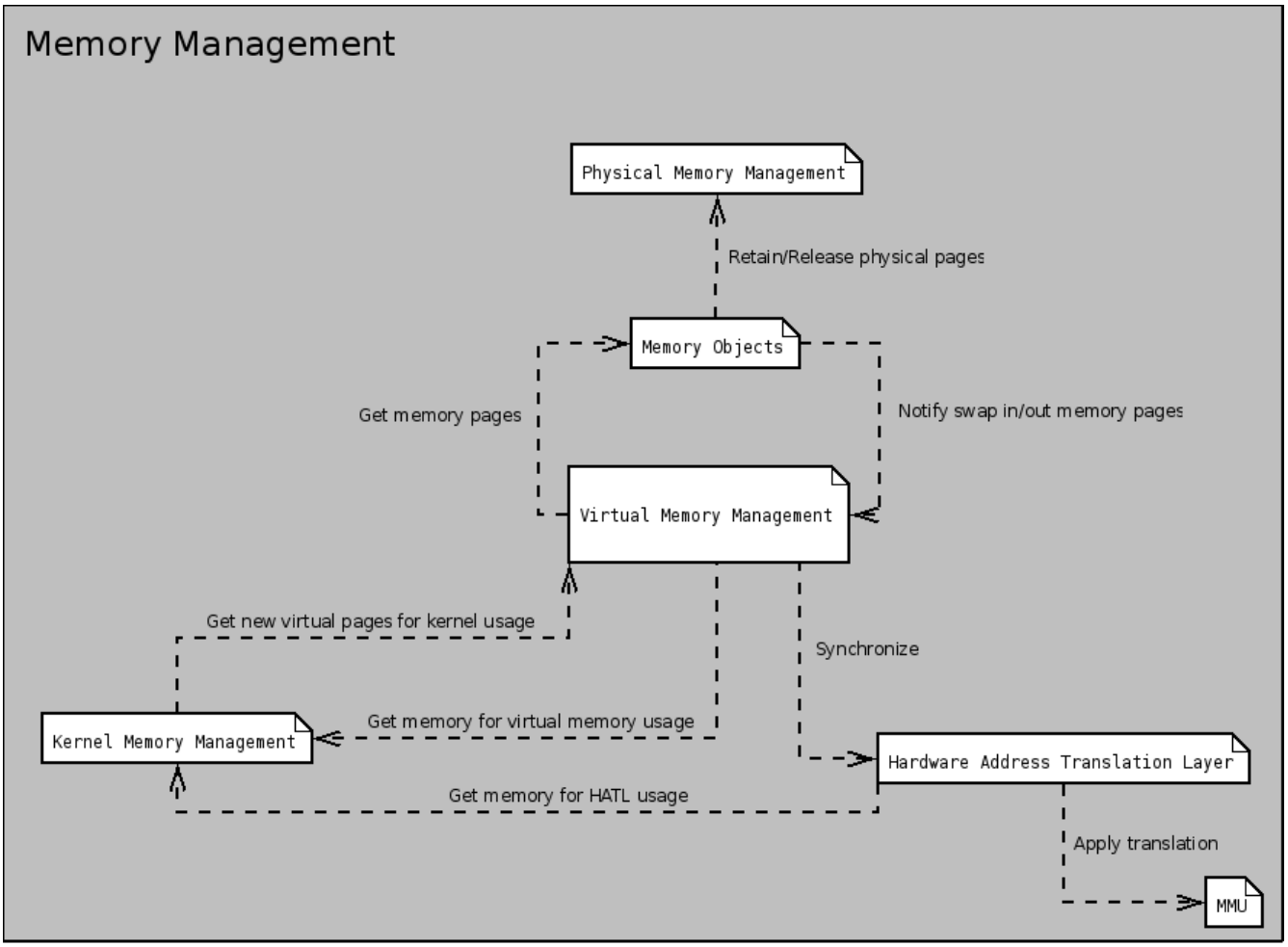

Fig 1 : Architecture globale du gestionnaire de mémoire

L'architecture se divise en 6 composantes, nous allons décrire rapidement chacune de ces composantes ainsi que leurs interactions. Cette architecture est directement liée aux problématiques, chaque composante résout un ou plusieurs des points décrits précédemment.

Physical Memory Management. A pour but de centraliser et de maintenir la distribution des ressources mémoire physique. Il offre la possibilité

d'obtenir une ou plusieurs pages[3], ainsi que de les libérer.

Memory Objects. Toute demande de mémoire du reste du système passent par eux.
Ils garantissent le partage des ressources mémoires, en se servant de la mémoire principale et de la mémoire secondaire.
Ils offrent également une interface permettant de « partager » le contenu de la mémoire entre les différents sous-systèmes. Partage pouvant être implicite ou explicite.

Parlons dans un premier temps du partage implicite. Prenons l'exemple d'une librairie partagée, la librairie standard C.
Cette librairie est chargée par la grande majorité des tâches utilisateurs d'un système d'exploitation de type *UNIX*. Au chargement de la tâche le linker insère dans l'espace d'adressage de cette dernière une partie du code contenu dans la librairie. Ce code est identique quelque soit la tâche qui l'utilise (il n'est habituellement pas modifié).
Les *Memory Objects* offrent donc un mécanisme pour charger le contenu d'un fichier (ou autre), et garantissent à la tâche d'avoir un contenu viable quelques soient les modifications réalisées. Ceci permet en interne d'avoir un système de cache sur le contenu et ainsi de partager au maximum les divers contenus.

Il est aussi possible de partager explicitement du contenu via cette interface. Deux tâches vont ainsi pouvoir demander une zone mémoire commune ou accéder et modifier en même temps le même fichier.

Notons que les *Memory Objects* se chargent aussi de mettre à jour automatiquement les sources des divers contenus chargés via leur interface.

Virtual Memory Management. Maintient les espaces d'adressage virtuel des différentes tâches en cours d'exécution.
Sa principale tâche est d'associer un contenu[4] à une plage d'adresses virtuelles[5].
Une fois que la page fournie par les Memory Objects a été associée à une page frame, il est nécessaire de notifier le matériel de cette association. Cette notification se fait en dialoguant avec le *Hardware Address Translation Layer*.

Hardware Address Translation Layer. Dialogue avec le matériel pour prendre en compte les associations de pages et de page frames.

Kernel Memory Management. Le noyau, comme toute tâche, a besoin de mémoire pour s'exécuter. Pouvoir fournir de la mémoire rapidement au noyau est primordial. Le rôle de ce module est donc de fournir en mémoire rapidement et intelligemment (sans perdre de ressources) le noyau.

3.3.2 Physical Memory Management

3 La page représente une zone contiguë de données d'une taille fixe, par exemple 4096 bytes.

4 Une page

5 Une page frame

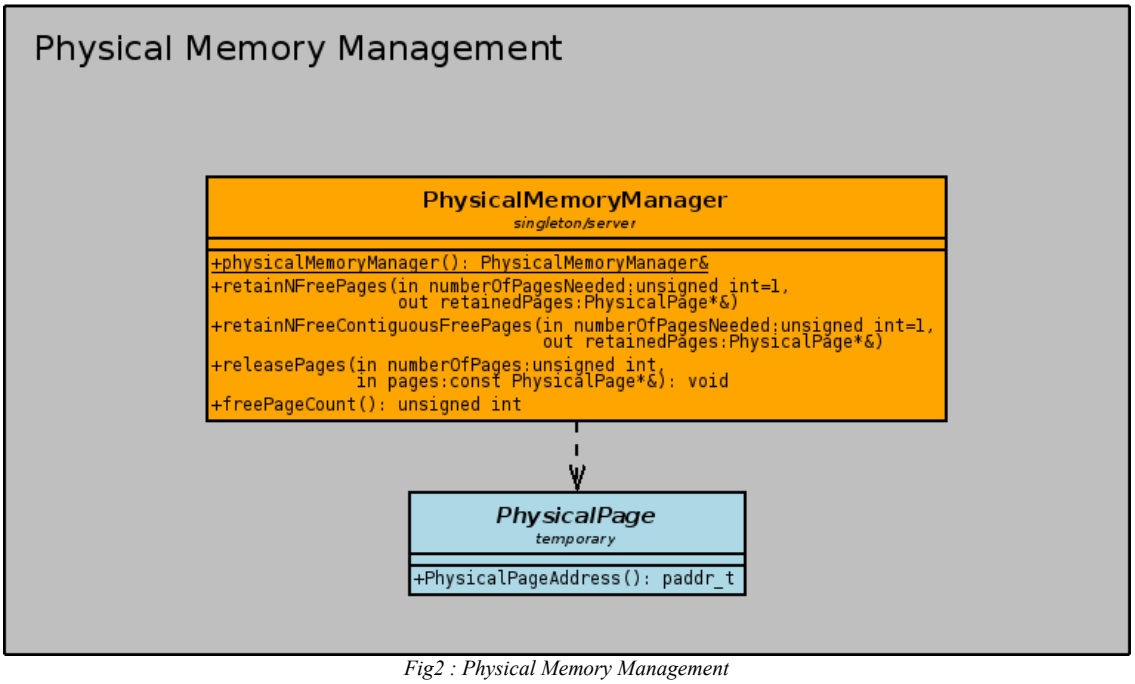

Fig2 : Physical Memory Management

Le fonctionnement de ce module ne nécessite pas de commentaires particuliers.

3.3.3 Memory Objects

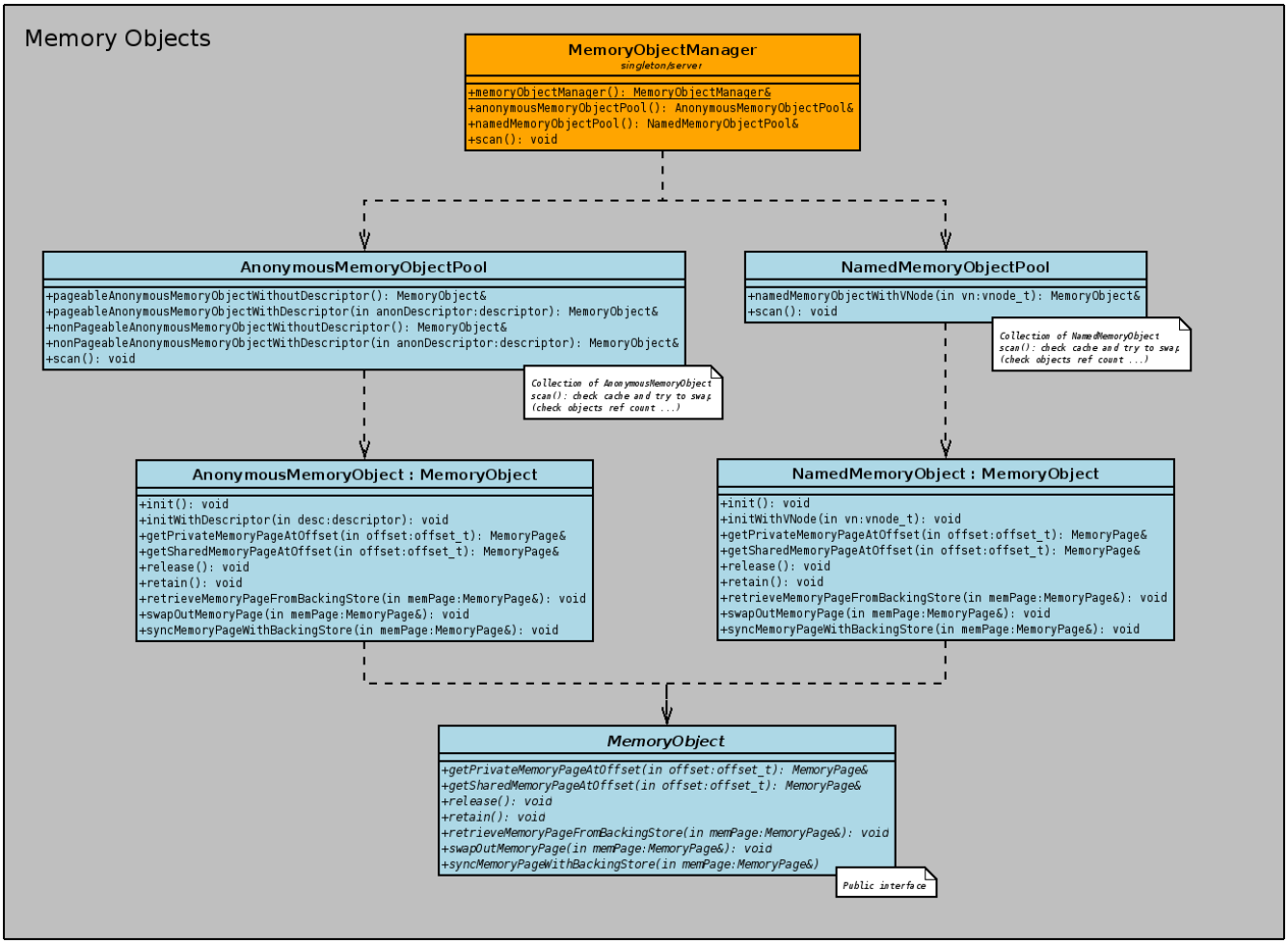

Fig 3 : Memory Objects

Il existe deux types de *Memory Object* différents : *Anonymous Memory Object* et *Named Memory Object*.

Comme son nom l'indique l'*Anonynous Memory Object délivre* de la mémoire anonyme. C'est à dire qu'elle ne provient et ne met à jour aucune source. C'est de la mémoire éphémère. A l'inverse le *Named Memory Object* délivre de le mémoire ayant une source, un fichier par exemple.

Une fois la source sélectionnée (anonyme ou nommée) les *Memory Objects* fonctionnent de la même façon. C'est pour cela qu'ils héritent tout deux d'une même interface.
Cette interface permet de remplir une page avec un fragment du contenu de la source, la page retournée peut être de type *private* ou *shared*.
Le type *private* garantit à l'appelant que le contenu de la page lui est privé. C'est à dire que les modifications éventuelles de tiers sur la source n'impactera pas cette page, mais aussi que les modifications de cette page n'impacteront pas la source.
A l'inverse le type *shared* partage les modifications avec la source et les tiers utilisant cette même page.

L'interface expose aussi le nécessaire pour forcer la page à être stockée sur la mémoire secondaire, et récupérer la page si elle est stockée sur la mémoire secondaire.

Une dernière méthode permet de forcer la mise à jour de la source (dans le cas de mémoire anonyme l'appel à cette méthode ne produira aucun effet).

Notons que le module *Memory Object Manager* ne se limite pas à délivrer de la mémoire, il veille en permanence au partage des ressources mémoires. Ce qui peut l'amener à passer certaines pages en mémoire secondaires.

3.3.4 Virtual Memory Management

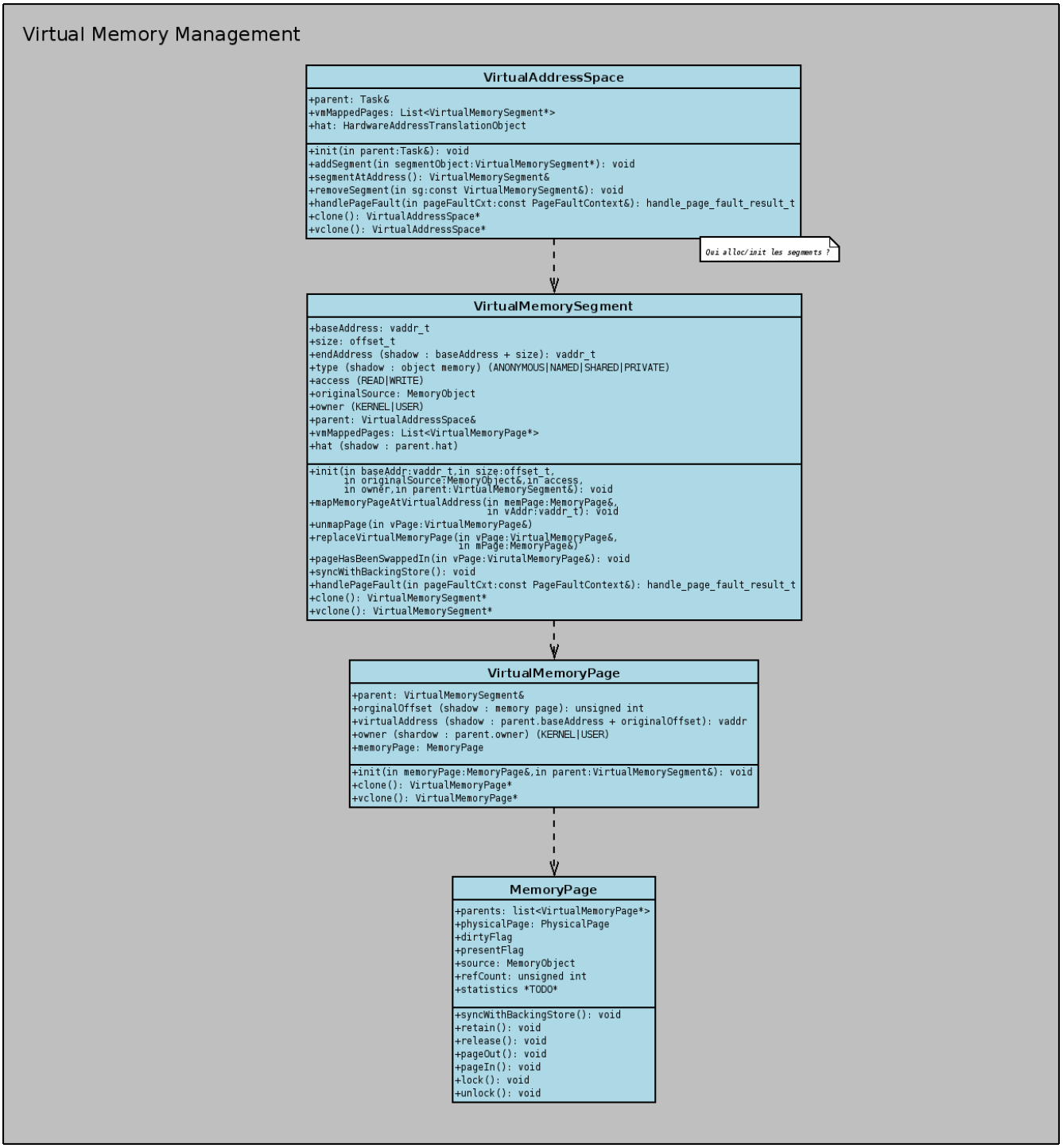

Fig 4 : Virtual Memory Management

Il s'agit plus ici d'une structure de données utilisée par le noyau que d'un module. Cette structure de données matérialise l'espace d'adressage virtuel d'une tâche (représenté par la classe *Virtual Address Space*). L'espace d'adressage n'est rien de plus qu'un ensemble de segments de mémoire virtuelle. On appelle ici un segment, un ensemble de page frames, notons que cet ensemble est de taille variable.
Chaque segment a un *Memory Object* qui lui est attribué, c'est à dire que chaque segment a potentiellement une source différente.
Chaque segment est donc constitué d'un ensemble de page frames dont le contenu est fourni par un *Memory Object*. La classe *Virtual Memory Segment* (la représentation d'un segment de mémoire virtuelle) contient un jeu de méthodes pour associer une page frame à une page. Lorsqu'on fait appel à ces méthodes, leurs actions ne se limitent pas à mettre à jour la structure de données. Elles assurent en même temps la synchronisation avec le *Hardware Address Translation Layer*.
En effet chaque espace d'adressage virtuel est lié avec une instance d'objet servant à informer le matériel sur les associations pages/page frames réalisées. Cet

objet se nomme *Hardware Address Translation Object*, nous le décrirons en détails par la suite.

La classe *Virtual Memory Page* représente une page frame, et enfin la classe *Memory Page* représente une page. Cette dernière est comme nous l'avons dit plus tôt délivrée par les *Memory Objects*.

3.3.5 Kernel Memory Management

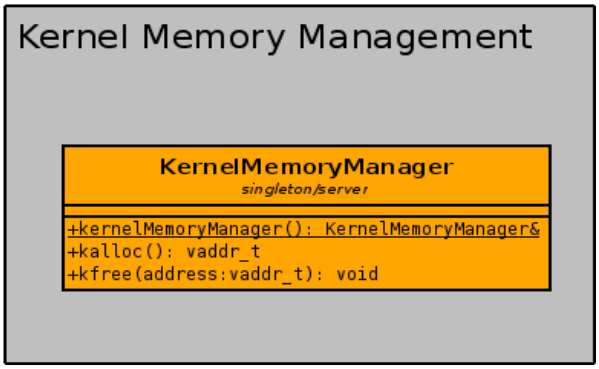

Fig 5 : Kernel Memory Management

Ce module permet au noyau d'allouer de la mémoire. Son fonctionnement interne se base sur le système d'allocation « Slab Allocator ». Le but de cet « allocator » est de subvenir aux demandes suivantes : une demande de mémoire dont la taille est inférieure à celle d'une page, une demande de mémoire dont la taille n'est pas multiple de celle d'une page, une demande dont l'utilisation est éphémère.
Son fonctionnement est relativement simple, le « Slab allocator » crée une collection d'objets de taille variable avant la demande du client et les stocke sur différentes pages qu'il maintient. Ceci lui permet d'anticiper sur la demande mais aussi de réutiliser des objets libérés. Les slabs peuvent être de tailles variables, le système peut demander l'ajout d'un slab de taille spécifique[6], ou bien l'allocator peut lui même décider d'ajouter une taille si la demande de la part du système est importante.

3.3.6 Hardware Translation Layer

6 Fig5 est incomplète pour le moment.

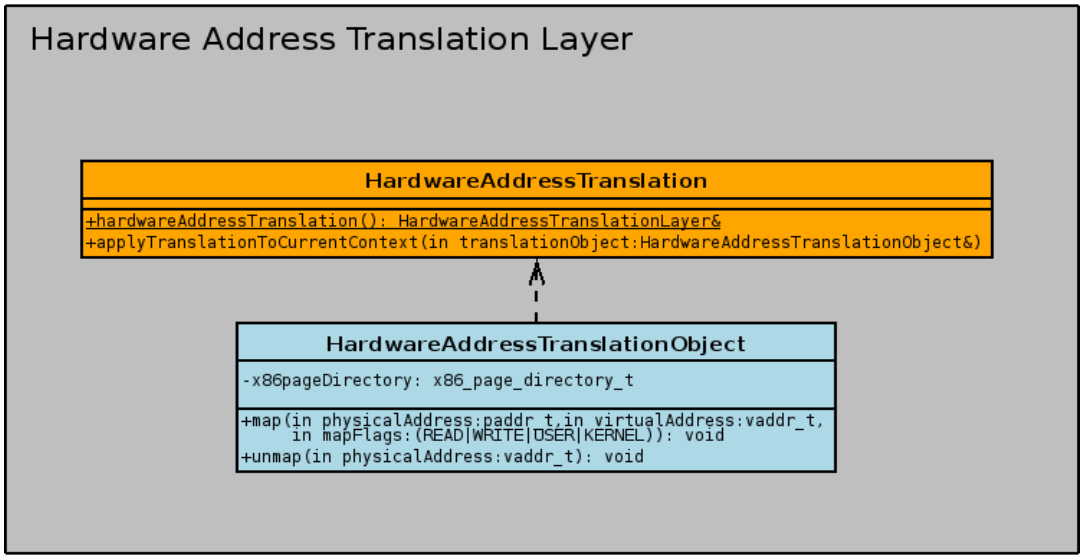

Fig 6 : Hardware Address Translation Layer

Ce module contient le nécessaire pour communiquer les associations page/page frames au matériel.
La classe *Hardware Address Translation Object* contient toute les informations sur ces traductions pour un unique espace d'adressage virtuel. Il existe donc autant de ces objets que d'espaces d'adressage virtuel.
Son fonctionnement étant entièrement dépendant de l'architecture matériel nous ne le détaillerons pas plus. La section suivant couvre une partie de ces détails pour l'architecture *x86*.

3.4 Détails pour l'architecture *x86*

Dans ce paragraphe, nous allons décrire comment le noyau gère les interruptions levées par la *MMU* ainsi que le fonctionnement de la classe *Hardware Address Translation Object* pour l'architecture *x86*.

3.4.1 Interaction avec la *MMU*

3.4.1.1 Gestion des pages faults

Lorsque la MMU rencontre une erreur lors de la traduction d'une adresse linéaire en adresse physique, cette dernière lève une interruption. Le noyau reprend la main et doit gérer cette interruption. Cette interruption se nomme *Page Fault*.
Lorsque ce type interruption est levé cela ne signifie pas obligatoirement qu'il s'agit d'un accès erroné à un contenu.
Un *Page Fault* peut survenir dans les trois scénarios suivant :

a. La tâche accède à une adresse valide dont le contenu n'a pas encore été mis en mémoire principale par le gestionnaire de mémoire.

b. La tâche tente d'accéder à un contenu non autorisé par ses privilèges.

c. La tâche accède une adresse sans contenu.

Les deux derniers scénarios entraînent la fin de la tâche ayant provoqué l'interruption.
Le premier scénario rend la main à la tâche une fois que le contenu a été chargé en mémoire par le gestionnaire de mémoire.

3.4.1.2 *x86* H. A. T. L. Object

Le but de cette classe est de maintenir en permanence les informations nécessaires à la MMU pour traduire les adresses virtuelles en adresses physiques.
Sur x86, cet objet contient au minimum une page contenant le *Page Directory* et de 0 à *N* pages contenant les *Page Tables*.
Notons que nous aurons toujours au moins une entrée dans notre *Page Directory*, ce sera l'adresse de la *Page Directory* elle-même. Ceci afin de faciliter les modifications des *Page Tables* et des *Page Table Entries*.

3.4.1.3 Activation de la pagination

Lorsque la pagination est activée il faut impérativement ne pas perturber le fonctionnement du système. C'est à dire qu'il faut configurer la MMU pour que les adresses linéaires concordent exactement avec les adresses physiques.