Mélanie Bats - Thomas Petazzoni

Kos : Kid Operating System
TX - UTBM

Automne 2003

Suiveur : M. Nicolas Lacaille

Remerciements

Nous tenons à remercier M. Nicolas Lacaille pour avoir suivi notre TX tout au long du semestre.

Nous remercions également David Decotigny et Julien Munier pour avoir travaillé sur le projet Kos depuis sa création, et pour nous avoir aidé et conseillé durant cette TX.

Nous remercions aussi l'équipe de développement de *Bochs*, l'émulateur de PC que nous utilisons, sans lequel le développement de Kos n'aurait pas été possible.

Table des matières

Introduction

Cette TX a été réalisée durant le semestre d'automne 2003 par Mélanie Bats[1] et Thomas Petazzoni[2] sous la responsabilité de Nicolas Lacaille. L'objectif de cette TX était de travailler sur le projet Kos, Kid Operating System, en y ajoutant diverses fonctionnalités.

La première partie donnera une présentation des diverses fonctionnalités de Kos, en mettant l'accent sur deux spécificités de ce système. Les parties suivantes correspondront chacune aux différents thèmes sur lesquels nous avons travaillé durant la TX.

[1] melanie.bats@utbm.fr
[2] thomas.petazzoni@enix.org

Chapitre 1

Présentation du projet Kos

Nous présenterons dans cette partie le projet Kos, en partant de l'objectif initial du projet. Nous étudierons deux caractéristiques techniques originales du projet : sa modularité et le système *Karm* d'accés aux ressources. Ensuite, nous détaillerons les diverses fonctionnalités du système et l'état d'avancement au début de la TX.

1.1 Objectif initial

Le projet Kos, Kid Operating System, consiste à développer un système d'exploitation pour ordinateurs PCs, à base de processeurs compatibles Intel. Le premier objectif du projet Kos n'est pas l'obtention d'un système d'exploitation révolutionnaire et pleinement fonctionnel mais l'apprentissage du mode de fonctionnement interne d'un système d'exploitation et des machines à base de processeurs Intel.

Le projet débuta en juin 1998 à l'initiative de jeunes programmeurs, pour la plupart étudiants ou lycéens. L'inexpérience a donc joué un grand rôle dans l'avancée du développement, de même que la disponibilité des développeurs. Après une période de recherches de plusieurs mois, le projet Kos a réellement commencé au printemps 1999. Cette première version a été développée pendant un an, date à laquelle les principaux développeurs ont décidé de repartir à zéro sur une base plus saine, à partir des connaissances nouvellement acquises.

A l'heure actuelle, le projet Kos compte 3 développeurs actifs.

1.2 Un système modulaire

Le système d'exploitation Kos est un système d'exploitation monolithique modulaire : les diverses fonctionnalités du noyau sont décomposées en modules, tels que *scheduler*, *task*, *fs/fat*, etc...

Chaque module est une portion de code indépendante des autres modules. Tous les modules sont reliés ensemble lors du démarrage du système. Chaque module doit explicitement *exporter* des fonctions pour les rendre accessibles à d'autres modules.

Cette modularité permet d'assurer une certaine indépendance entre les différents éléments du système, et donc d'éviter une imbrication trop forte des différents mécanismes.

Dans chaque module, on retrouve au minimum :

– Un fichier `nomdumodule.c` contenant les fonctions d'initialisation et de désinitialisation du module, ainsi que la liste des fonctions exportées par le module.

- Un fichier `nomdumodule.h` contenant les déclarations publiques relatives au module, et utilisables par d'autres modules.
- Un fichier `_nomdumodule.h` contenant les déclarations privées au module. Il ne doit pas être inclu par d'autres modules.
- D'autres fichiers C implémentant les diverses fonctions exportées et leurs sous fonctions éventuelles.

A l'heure actuelle, Kos est composé d'une trentaine de modules, voir Annexe A.

1.3 Karm : une façon originale d'accéder aux ressources

1.3.1 L'accès aux ressources sous Unix

Le rôle d'un système d'exploitation est de permettre d'accéder de manière simple et uniforme aux ressources matérielles et logicielles disponibles. Pour cela, les systèmes de fichiers de type Unix définissent une abstraction unique : le *fichier*. Toutes les ressources sont représentées par des fichiers (à l'exception des processus et des sockets). On accède à ces ressources par une interface unique de type *read*, *write*, *seek*.

Cette apparente uniformité cache cependant des disparités entre ces ressources : il est parfois nécessaire de demander un traitement spécifique à une ressource, par exemple changer la fréquence d'échantillonage d'une carte son, la résolution d'une carte graphique, etc... Ces traitements spécifiques ne sont pas réalisables via l'interface uniforme d'accès aux fichiers, c'est la raison pour laquelle un appel système spécifique, *ioctl* a été créé.

Cet appel système a le prototype suivant :

```
int ioctl(int d, int request, ...);
```

Le premier argument, *d*, est le descripteur de fichier correspondant à la ressource que l'on souhaite manipuler. Le second argument, *request*, est la commande que l'on souhaite effectuer. Le reste est une liste d'un nombre inconnu d'arguments.

Cet appel système est implémenté pour chaque fichier représenté dans l'espace de nommage, et les commandes associées à chaque fichier dépendent du *pilote* sous-jacent. Typiquement, l'implémentation d'*ioctl* dans un pilote de périphérique ou un système de fichiers est un `switch` sur toutes les valeurs possibles de la commande (*request*). A titre d'exemple, on pourra consulter le fichier `drivers/char/lp.c` du noyau Linux, et plus précisement la fonction `lp_ioctl` (voir B), qui permet de réaliser diverses opérations spécifiques au port parallèle.

L'inconvénient de ce mécanisme d'*ioctl* est que les fonctionnalités diverses d'une ressource ne sont pas disponibles clairement via une interface bien définie. Ces fonctionnalités sont disponibles par des numéros de commande spécifiques à un pilote de périphérique particulier. Par exemple, pour réinitialiser le port parallèle, il convient d'utiliser la commande `LP_RESET`, correspondant au nombre `0x060c`, tandis que ce même nombre pourra avoir une signification différente pour un autre pilote de périphérique.

1.3.2 La proposition du projet Kos

Fonctionnement général de Karm

Les développeurs du projet Kos ont souhaité clarifier l'accès aux diverses fonctionnalités des ressources, en prenant en compte leurs disparités. Le résultat de ces réflexions est le système

appellé KARM, pour *Kos Advanced Ressource Management*. L'idée principale de ce système est de considérer que les ressources sont accessibles via de multiples interfaces.

Sous Unix, lorsqu'on ouvre une ressource, c'est systématiquement selon une interface unique : l'interface *fichier*. A contrario, sous KOS, à l'ouverture d'une ressource, on spécifie selon quelle interface on souhaite l'utiliser. L'appel système `open` a le prototype suivant :

```
int open(char *pathname, unsigned interface, int flags);
```

Grâce à cet appel, on peut ouvrir une ressource en spécifiant son chemin dans l'espace de nommage (`/home/utbm/tx.txt` par exemple), une interface et des drapeaux.

Le système KARM, au sein du noyau, sait quelles interfaces sont supportées par quelles ressources, et peut donc autoriser ou non l'ouverture de la ressource selon l'interface demandée.

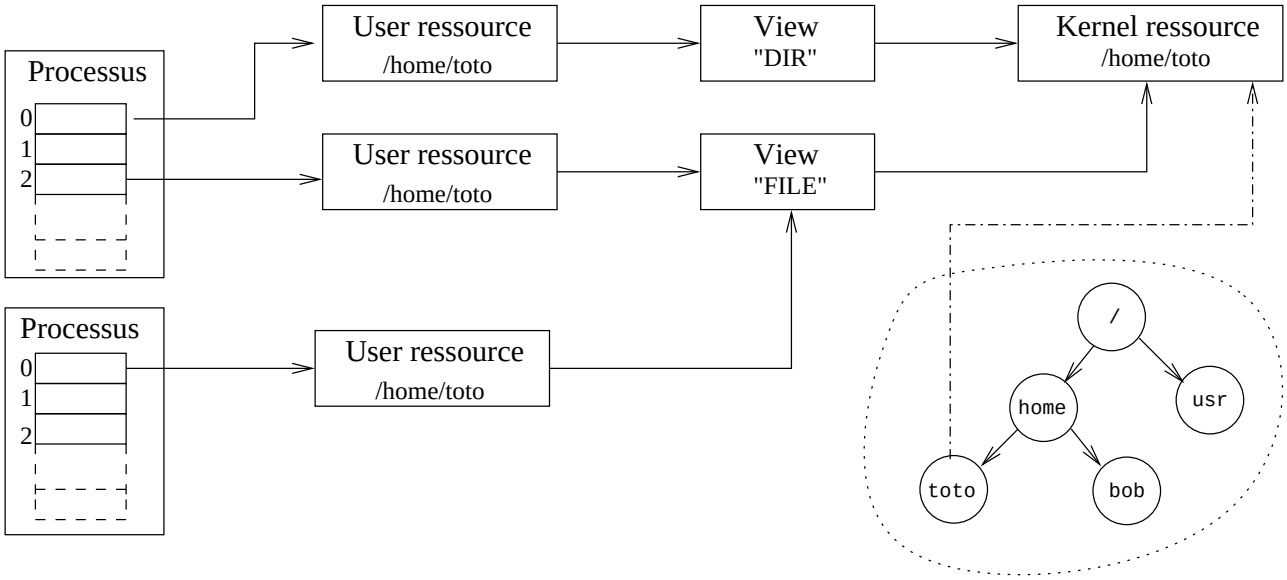
Fonctionnement interne

Au sein du noyau, chaque ressource est représentée par une structure de type `modules/karm/kres.h: struct kres`. Il n'existe qu'une seule et unique `kres` (*Kernel Ressource*) pour chaque ressource. Ainsi, il y a une `kres` pour chaque répertoire, pour chaque fichier, pour chaque port série, pour chaque disque dur, pour chaque carte son, etc...

Pour retrouver une *Kernel Ressource* à partir du chemin associé (par exemple `/home/toto`), un système appellé `nscache` (*NameSpace Cache*) conserve en mémoire une partie de l'arborescence. Chaque noeud de cette arborescence est lié à une *Kernel Ressource*. En maintenant en mémoire une portion de l'arborescence, il permet d'accélérer la recherche des fichiers.

Chacune de ces *Kernel Ressource* supporte un certain nombre d'interfaces, dont l'implémentation est donnée dans une structure de type `modules/karm/view.h: struct view`. Une `view` associe un numéro d'interface à une liste de méthodes implémentant cette interface. Ces `view` sont regroupées par *Kernel Ressource* dans une liste chaînée.

Lorsqu'une application utilisateur appelle la fonction `open`, le système KARM vérifie si la *Kernel Ressource* supporte l'interface demandée. Si c'est le cas, il génère une structure de type `modules/karm/ures.h: struct ures`. Une `ures` (*User Ressource*) correspond donc à une ouverture particulière d'une ressource. Les *User Ressource* sont regroupées par processus dans le tableau des descripteurs de ressources (sous Unix, le tableau des descripteurs de fichiers).



NameSpace Cache

1.4 Autres fonctionnalités

Au début de la TX, nous disposions déjà d'un système d'exploitation présentant de nombreuses fonctionnalités. Nous les détaillons ci-après, structurées par thèmes.

Toutefois, il est intéressant de rappeler que l'ensemble du noyau de Kos est préemptif[1] et réentrant[2].

1.4.1 Modules, *loader* et démarrage

Le démarrage du système Kos se fait par l'intermédiaire du logiciel Grub [3]. Ce logiciel permet au démarrage de choisir le système d'exploitation à lancer, et est une alternative à Lilo offrant de nombreuses fonctionnalités. Ainsi, il est capable d'aller lire un fichier sur une partition, qu'elle soit au format FAT ou ext2, il peut télécharger des fichiers sur le réseau, etc... Grub peut démarrer tous les grands systèmes d'exploitation existants. Pour les autres, il propose un standard appellé *Multiboot* (voir référence [2]).

Grub peut charger en mémoire un noyau et divers modules, avant de passer la main au point d'entrée du noyau. Pour Kos, le noyau donné à Grub est appellé *loader*, et tous les modules passés à Grub sont les modules de Kos. Les sources du loader sont dans le répertoire `loader` des sources de Kos. Son rôle est d'initialiser la pagination, puis d'effectuer un linkage dynamique des différents modules (résolution des symboles), avant de passer la main au "vrai" noyau, c'est à dire les modules. Pour cela, il passe la main à la fonction `init/_bootstrap.c: kernel_init`, qui se charge d'initialiser tous les modules (fonctions `init_module`) avant d'appeler la fonction `kos/wolfgang.c: kernel_start`.

Au début de la TX, tout le système de démarrage de Kos et de linkage dynamique des modules est fonctionnel et largement testé (existe depuis 3 ans).

1.4.2 Gestion de la mémoire

La gestion de la mémoire dans Kos utilise le mécanisme de pagination, avec des pages de 4 Ko (ce qui est la taille par défaut sur les architectures Intel). Il n'y a pas de mapping à l'identique de la mémoire physique, comme on peut le trouver sous Linux, ce qui permet de gérer de manière très souple la mémoire physique.

Cette gestion de la mémoire se répartit dans différents modules : `pmm`, `x86/mm`, `vmm`, `kmem`.

Le module `pmm`

Le module `pmm`, pour *Physical Memory Management*, s'occupe de gérer la mémoire physique. Son rôle se limite donc à maintenir un tableau de descripteurs de pages physiques, dont chaque entrée de type `pmm.h: gpfme_t` contient l'état d'une page, son utilisation, etc ... Il propose des fonctions telles que `pmm/_pmm_get_page.c: get_physical_page` pour allouer une page physique ou `pmm/_pmm_put_page.c: put_physical_page` pour libérer une page physique.

La gestion des pages physiques de `pmm` prend en compte différentes subtilités, notamment les espaces mappant des périphériques matériels ainsi qu'un état *swappable* ou *non swappable* pour chaque page physique.

Au début de la TX, la gestion de la mémoire physique était entièrement fonctionnelle.

[1] Un code est dit préemptible si il est interruptible à tout instant de son éxécution

[2] Un code est dit réentrant si plusieurs éxécutions de ce code peuvent avoir lieu simultanément

[3] http ://www.gnu.org/software/grub/

Le module `x86/mm`

Le module `x86/mm` contient les fonctions de gestion de la mémoire dépendantes de l'architecture. On y trouvera donc :

- des fonctions permettant de mapper et démapper des pages ou changer leurs droits d'accès (fichier `_vmap.c`).
- un mécanisme de *reverse mapping*. Ce mécanisme sert à maintenir un mapping inverse entre les pages physiques et les pages virtuelles associées. En effet, les tables de pages utilisées par le processeur ne donnent que le mapping virtuel vers physique. Pourtant, le mapping inverse est très utile pour simplifier la gestion des zones de mémoire anonyme et pour le swapping (fichier `_rmap.c`).
- la *GDT*, *Global Descriptor Table*, table spécifique aux processeurs Intel précisant les différents segments de la mémoire (fichier `_gdt.c`).
- le point d'entrée du gestionnaire de défauts de page (fichier `_pgflt.c`).
- des fonctions permettant de créer et supprimer des nouveaux espaces d'adressage (fichier `_team_mm_context.c`).

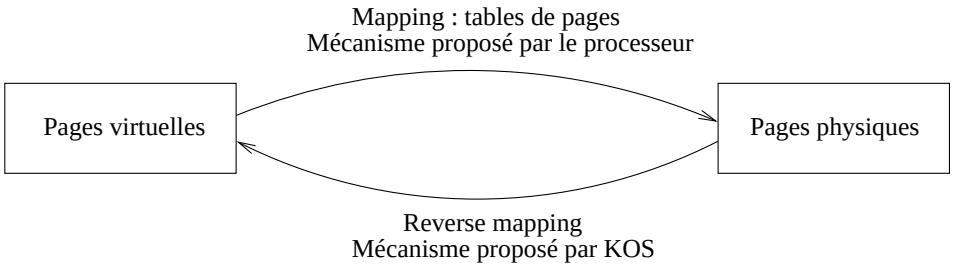


FIG. 1.1 – Mapping et reverse mapping

Au début de la TX, ce module était entièrement fonctionnel.

Le module `vmm`

Le module `vmm` est chargé de la gestion de la mémoire virtuelle, c'est à dire du découpage des espaces d'adressage des processus en régions virtuelles. En effet, l'espace d'adressage de chaque processus est découpé en différentes zones, appellées régions virtuelles. Elles correspondent par exemple au code du programme éxécuté, aux données du programme, à la pile, ou bien aux codes ou données des bibliothèques partagées utilisées.

Ce module s'occupe de la création, destruction et modification de ces régions virtuelles, ainsi que de la gestion des défauts de page (fichier `_vmm_as.c`). Un espace d'adressage est représenté par une structure de type `vmm.h: address_space`, et une région virtuelle par une structure de type `vmm.h: virtual_region`.

Il existe deux types de région virtuelles :

- les régions anonymes, ne correspondant pas à un fichier sur le disque. Elles sont utilisées pour la mémoire allouée dynamiquement (le tas et la pile).
- les régions correspondant à un fichier sur le disque. Dans ce cas, la région virtuelle est liée à une `ures`, ouverte selon l'interface `INTERFACE_MAPPING_ID` (voir `karm/interface/mapping.xml`).

Le schéma suivant donne un aperçu de l'espace d'adressage d'un processus, tel qu'il pourrait être quand KOS supportera les bibliothèques partagées :

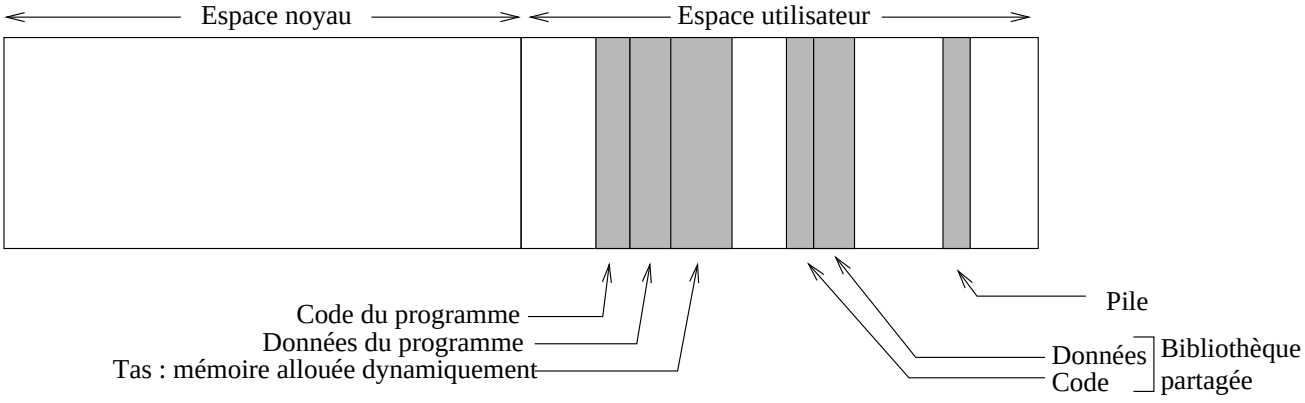


FIG. 1.2 – Les régions virtuelles

Au début de la TX, le module `vmm` était quasiment vide, seul un embryon de création de régions virtuelles était présent.

Le module `kmem`

Le module `kmem` est un allocateur de mémoire pour le noyau. Il propose donc les fonctions classiques d'allocation et libération `kmalloc` et `kfree`. Cet allocateur permet au noyau d'allouer et libérer dynamiquement de petites quantités de mémoire pour stocker ses structures de données internes.

Il repose sur un allocateur de type *Slab Allocator*, proposé par Jeff Bonwick (voir référence [1]).

Au début de la TX, ce module était pleinement fonctionnel, et largement testé depuis plusieurs années.

1.4.3 Gestion des tâches et *scheduling*

Dans KOS, une terminologie bien précise a été définie en ce qui concerne les diverses entités en jeu dans la gestion des tâches, de manière à pallier au flou existant autour de la notion de *processus* sous Unix.

Un *thread* est un fil d'éxécution, défini par un contexte (registres, pointeur d'instruction, pointeur de pile, etc...). Une *team* est une équipe de *threads* fonctionnant dans le même espace d'adressage.

Il existe deux types de *threads* sous KOS : les threads dits *noyau* qui tournent avec les privilèges du noyau et les threads dits *utilisateur* qui tournent avec les privilèges utilisateur, et dans un espace d'adressage particulier.

Un *processus* Unix classique peut donc être vu comme une *team* contenant un seul *thread utilisateur*.

Dans le schéma suivant, la *team 1* contient deux *threads*, et la *team 2* en contient un. La *team 2* peut donc être assimilée à un *processus* Unix. L'éxécution des *threads A* et *B* est protégée de l'éxécution du *thread C* par le cloisonnement des espaces d'adressage.

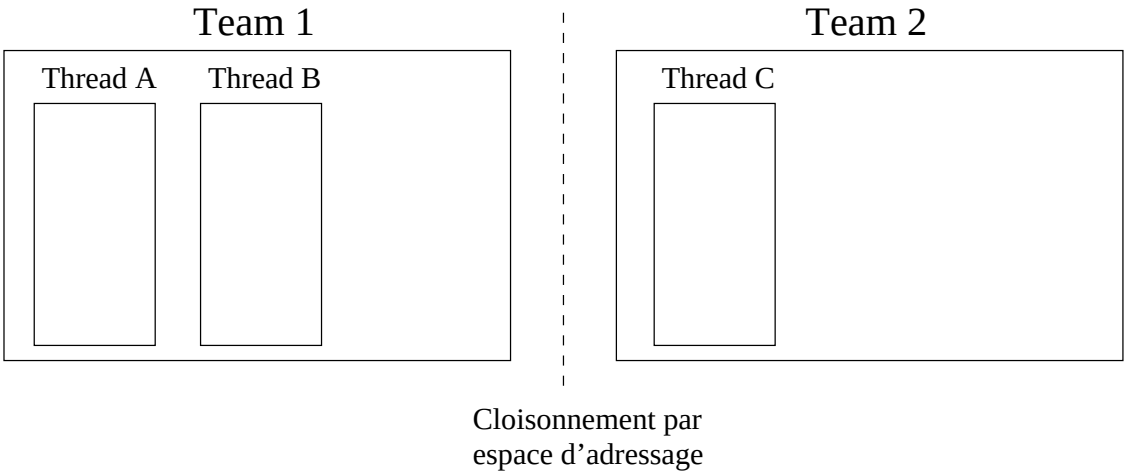


FIG. 1.3 – *Team* et *thread*

Tout *thread* possède une pile au niveau noyau. Dans le cas des *threads noyau*, c'est la pile sur laquelle toute l'éxécution a lieu, tandis que pour les *threads utilisateur*, seule l'éxécution des appels systèmes a lieu sur cette pile. En effet, ce type de *thread* utilise également une pile utilisateur. Une pile noyau est de taille modeste (quelques kilo-octets) et non extensible, une pile utilisateur est nettement plus grosse (quelques mega-octets) et extensible.

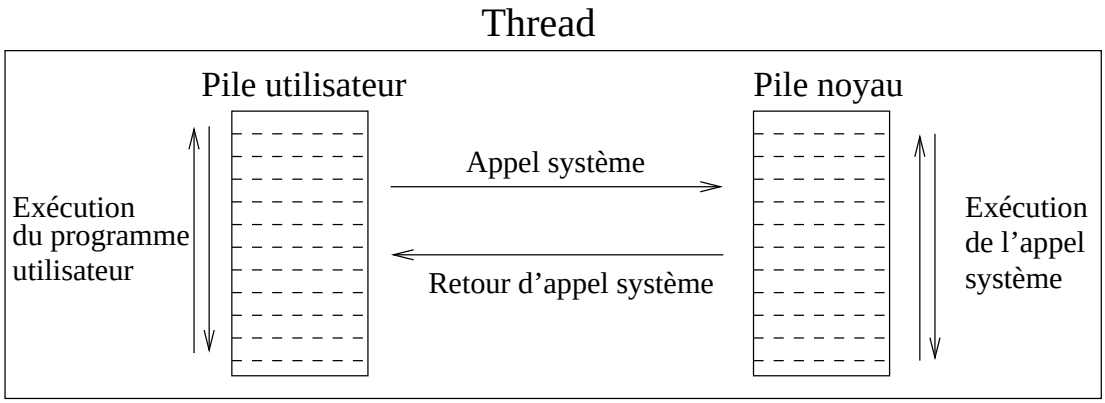


FIG. 1.4 – Les piles d'un *thread* utilisateur

La gestion des tâches est répartie entre les modules `task`, `x86/task` et `scheduler`.

Le module `task`

Ce module contient des fonctions permettant de créer et supprimer des teams et des threads, et de gérer les piles noyaux associées. Une *team* est représentée par une structure de type `task.h: team` et un thread par une structure de type `task.h: thread`.

Au début de la TX, seuls les threads *noyau* étaient supportés, le support des *threads utilisateur* était commencé mais non finalisé.

Le module `x86/task`

Ce module contient les fonctions permettant la gestion des threads au niveau de l'architecture Intel. On y trouve les fonctions de création et de destruction du contexte d'un thread

(fichier `_thread_cpu_context.c`) ainsi que les fonctions de changement de contexte (fichier `_cpl0_switch.c` et `_cpl0_switch_asm.S`). Le contexte de chaque *thread* est représenté par une structure de type `task.h: cpu_state_t`.

Le fichier `_dbflt.c` permet de gérer le *double fault*, l'exception générée par le processeur lors d'un débordement de pile noyau, et le fichier `_tss.c` permet de gérer les *TSS*, *Task State Segment*, une structure de donnée propre à l'architecture Intel. Le fichier `_syscall.c` contient le point d'entrée de l'appel système.

Dans ce module aussi, le support des *threads utilisateur* était commencé mais non finalisé.

Le module `scheduler`

Ce module contient le *scheduler*, ainsi qu'un système de *synchq* (*synchronisation queues*) sur lequel le *scheduler* et toutes les primitives de synchronisation sont basés.

Une *synchq* est une liste de threads. Chaque *thread* est enregistré sur une seule et unique *synchq*, sauf le thread en cours d'éxécution. Le scheduler dispose d'une *synchq*, la *cpu_synchq*, sur laquelles les threads en attente d'éxécution sont enregistrés. Les autres threads sont en attente sur des ressources, et enregistrés sur d'autres *synchq*.

Le code de gestion des *synchq* (initialisation, ajout d'un thread, suppression d'un thread, destruction) se trouve dans le fichier `_synchq.c`. Le code du *scheduler* est très simple : il se contente d'élire le prochain thread dans sa *cpu_synchq*, voir fonction `scheduler_retrieve_next_thread`. On peut enregistrer un nouveau thread dans la file d'attente du *scheduler* avec la fonction `scheduler_register_ready_thread`.

Les fonctions qui déclenchent le changement de contexte sont les fonctions `reschedule_after_interrupt`, `reschedule_after_termination`. La première fonction est appellée après chaque interruption matérielle, tandis que la seconde est appellée lorsqu'un thread se termine.

Le fichier `_sleep.c` implémente des fonctions d'attente active et non active. Ce dernier type d'attente utilise la fonctionnalité des *synchq* pour placer le *thread* en attente.

Le mécanisme de *synchq* a été implémenté récemment, mais était opérationnel au début de la TX.

1.4.4 Gestion des interruptions

La gestion des interruptions est implémentée dans KOS dans le module `idt`. Les interruptions sont de trois types :

- les exceptions, générées par le processeur suite à des erreurs comme la division par zéro ou le défaut de page. Le fichier `_exception.c` permet d'enregistrer des gestionnaires spécifiques pour les exceptions.
- les IRQs, générées par les périphériques matériels pour signaler un évènement, comme l'appui sur une touche ou la fin d'un transfert sur le disque. Le fichier `_irq.c` permet d'enregistrer des gestionnaires spécifiques pour les IRQs.
- l'appel système ou *syscall* est une interruption générée par le logiciel, en particulier par les applications utilisateur désireuses d'éxécuter un service du noyau. Le fichier `_syscall.c` permet d'enregistrer un gestionnaire spécifique pour le syscall.

Le fichier `_i8259.c` contient du code chargé d'initaliser et de configurer le gestionnaire matériel des IRQs, le *i8259*. Ce code est donc dépendant de l'architecture et devrait être déplacé dans un module dépendant de l'architecture.

Les fichiers `_dst.c` et `_dsr.c` implémente les mécanismes de *Deferred Service Thread* et de *Deferred Service Routine*, qui permettent de reporter le traitement d'une interruption. En effet, le temps d'éxécution d'un gestionnaire d'interruption devant être le plus court possible, il

faut limiter la taille de celui-ci, et reporter les traitements plus tard. L'éxécution des *Deffered Service Routine* a lieu une fois que toutes les IRQs en attente ont été traitées, et ces routines doivent être non bloquantes. On retrouve l'équivalent de ce mécanisme sous Linux avec les *bottom halves*. L'éxécution des *Deffered Service Thread* a lieu dans le cadre d'un *thread* normal, c'est à dire n'importe quand et que ces fonctions peuvent être bloquantes.

Le schéma suivant montre un exemple d'enchaînement des éxécutions des différents gestionnaires d'interruptions. Au départ, une tâche A est en cours d'éxécution. Elle est interrompue par une interruption *IRQ 1* : l'éxécution du gestionnaire correspondant commence, et marque le *DSR 24*. Durant cette éxécution, une interruption plus prioritaire *IRQ 0* survient. L'éxécution du gestionnaire correspondant a lieu, et marque le *DST 12*. Lorsque l'éxécution se termine, le gestionnaire de l'interruption *IRQ 1* reprend et termine. Une fois que celui-ci est terminé, il ne reste plus d'interruptions en attente de traitement, les *DSR* sont éxécutés. Ici, seul le *DSR 24* avait été marqué. Une fois terminé, l'éxécution d'une tâche B classique reprend. Plus tard, le *DST 12* sera éxécuté (comme un *thread* normal).

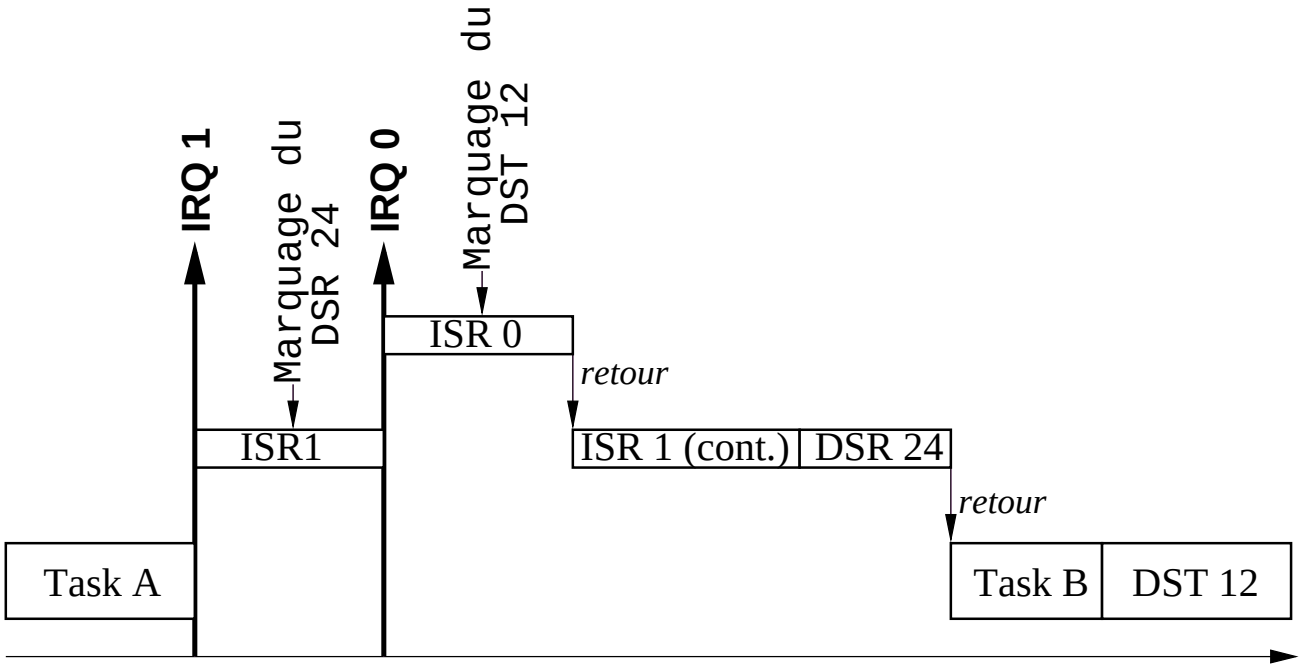


FIG. 1.5 – Enchaînement des différents mécanismes de traitement des interruptions

La gestion des interruptions de base est donc entièrement fonctionnelle dans KOS. En revanche, les *Deffered Service Routine* et *Deffered Service Thread* ont été peu utilisés et donc peu testés.

1.4.5 Synchronisation

KOS étant un système entièrement préemptif, la synchronisation est un problème important. Il s'agit de garantir qu'un accès concurrent aux données sera réalisé.

Pour cela, on dispose de :

- *spinlock* permettant de verouiller une petite portion de code. Leur implémentation sous forme de macro se trouve dans `kos/spinlock.h`.
- *mutex*, implémentées dans `kitc/_kmutex.c`

– *sémaphores*, implémentées dans `kitc/_ksem.c`
– *boîtes aux lettres*, implémentées dans `kitc/_kmsg.c`

Ces fonctionnalités de synchronisation sont réservées au noyau. Kos ne propose pas à l'heure actuelle de fonctions de synchronisation pour les applications utilisateur. D'autre part, seuls les *spinlocks* ont été testés de manière intensive et ils sont utilisés de manière quasi-exclusive dans tout le système. Les autres mécanismes n'ont été que peu testés.

En général, la pratique de programmation de Kos est de coder une partie du système sans synchronisation, puis de la reprendre en prenant en compte le problème de synchronisation. Or, depuis plusieurs mois, ce travail de reprise du code n'a pas été entrepris, et de larges parties de Kos ne comporte pas de synchronisation.

1.4.6 Pilotes de périphériques

Le système Kos dispose déjà de plusieurs pilotes de périphériques :
– Un pilote de périphérique clavier (module `klavier`).
– Un pilote de périphérique console en mode texte (module `console`).
– Un pilote de terminal TTY, qui créé un terminal à partir d'une entrée et d'une sortie (module `tty`).
– Un pilote de périphérique IDE pour lire sur les disques durs (module `ide`).
– Un pilote de "périphérique" pour détecter et lire des partitions (module `part`).
– Un pilote de périphérique série minimal dans le module `debug`.
– Un pilote de système de fichiers FAT, qui fonctionne en lecture uniquement (module `fs/fat`).

1.4.7 Fonctionnalités de débuggage

Kos implémente diverses fonctionnalités facilitant le débuggage dans le module `debug` :
– Des fonctions pour écrire sur la console qui a lancé *Bochs*, l'émulateur utilisé pour les tests. (fichier `bochs.c`).
– Des fonctions pour afficher la pile d'appel, ou *backtrace* (fichier `bt.c`).
– Un système permettant d'associer une adresse à un symbole du noyau (fichier `syms.c`).
– un désassembleur d'instructions (récupéré dans *Bochs*), voir `disasm.c`

On dispose donc de fonctionnalités de débuggage très au point, et très pratiques à utiliser.

1.5 Comment tester Kos

Un document expliquant de manière complète la procédure permettant de tester et débugger Kos a été écrite dans le cadre de ce projet (voir référence [3]).

1.6 Vue d'ensemble des structures de données

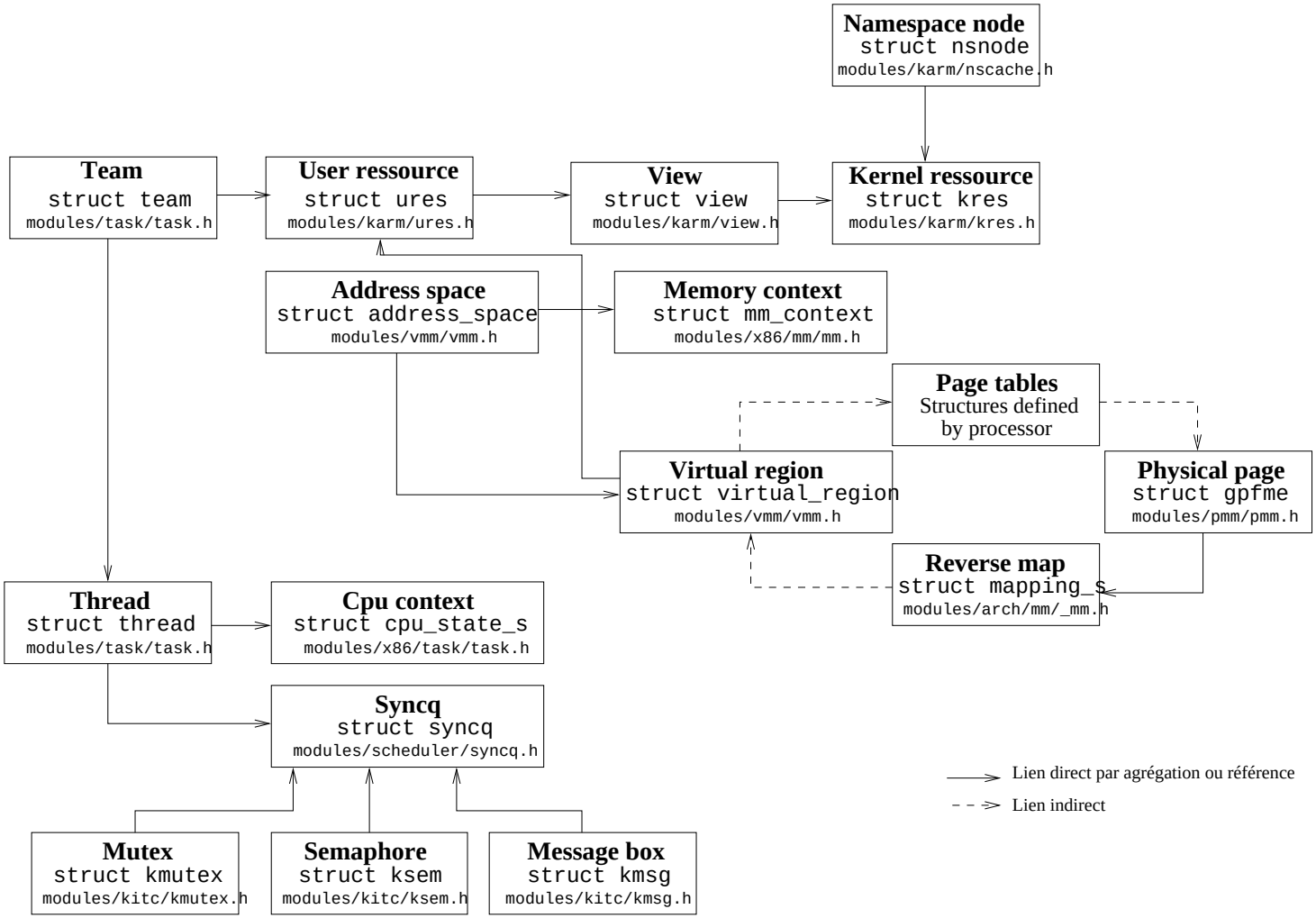


FIG. 1.6 – Vue d'ensemble des principales structures de données

Chapitre 2

Objectifs de la TX

A partir de cet embryon de noyau, nous nous sommes fixés divers objectifs. Les problématiques associées à ces objectifs seront détaillées tout au long du document.

- Pour le système KARM, écriture d'un générateur d'interfaces qui utilise des descriptions XML d'interfaces.
- Développement des fonctionnalités nécessaires pour éxécuter des applications utilisateur (au niveau de l'ordonnancement, de la création des *threads*, de la mémoire virtuelle et des appels systèmes notamment).

Chapitre 3

Générateur d'interfaces

3.1 Problématique

Au début de la TX, les interfaces du système Karm étaient définies en C sous forme de structure de pointeurs de fonctions, comme ci-dessous :

Listing 3.1 – Description d'une interface en C

```
struct block {
  result_t (*read)(struct ures *ur, char *buffer,
                   count_t block_start,
                   count_t *inout_block_count);
  result_t (*write)(struct ures *ur, const char *buffer,
                    count_t block_start,
                    count_t *inout_block_count);
  result_t (*get_device_size)(const struct ures *ur,
                              size_t *out_block_size,
                              count_t *out_block_count);
  result_t (*ctrl_lock)(struct ures *ur, block_lock_op_t op_type,
                        count_t block_start);
};
```

Cette méthode présentait plusieurs inconvénients :
- il n'y avait pas de vérification automatique du nombre de paramètres,
- on ne disposait pas d'identifiants de méthodes et d'interfaces pour l'espace utilisateur,
- on ne pouvait pas distinguer les méthodes destinées au noyau et celles destinées à l'utilisateur.

3.1.1 Vérification automatique du nombre de paramètres

Les méthodes définies par les interfaces Karm et implémentées dans le noyau sont destinées à être utilisées à la fois par le noyau lui même, mais aussi par les applications utilisateur au travers d'un appel système. Au niveau du noyau, la vérification du nombre d'arguments et de leur type est effectuée directement par le compilateur. En effet, les appels aux méthodes des différentes interfaces sont effectués en utilisant les structures telles que celles définies plus haut :

```
result = BLOCK_OPS(block_ures->view->ops)
         ->write(block_ures, block_buffer, block_id, & block_count);
```

En revanche, dans le cas de l'application utilisateur, la valeur des différents paramètres seront passées via un tableau associé au nombre de paramètres (taille du tableau). Les méthodes sont alors appellées en fonction de ce que passe l'utilisateur comme nombre de paramètres, ce qui n'est pas acceptable (risque de corruption de pile). Par exemple, si l'utilisateur appelle la méthode `write` de l'interface *block* en donnant 5 paramètres au lieu de 4, l'appel sera provoqué avec 5 paramètres, alors que la fonction s'attend à n'en recevoir que 4.

Il fallait donc un moyen de vérifier ce nombre de paramètres. Au début de la TX, le seul moyen était de maintenir à la main un tableau donnant pour chaque méthode de chaque interface le nombre de paramètres :

Listing 3.2 – Tableau global des interfaces

```
struct interface interface_array[] = {
  [INTERFACE_CHAR_ID]      = { "char", INTERFACE_CHAR_NB_OPS,
                               NB_ARGS_ARRAY(3, 3) },
  [INTERFACE_BLOCK_ID]     = { "block", INTERFACE_BLOCK_NB_OPS,
                               NB_ARGS_ARRAY(4, 4, 3, 3) },
  [INTERFACE_FILE_ID]      = { "file", INTERFACE_FILE_NB_OPS,
                               NB_ARGS_ARRAY(3, 3, 3) },
  [INTERFACE_DIR_ID]       = { "dir", INTERFACE_DIR_NB_OPS,
                               NB_ARGS_ARRAY(3) },
  [INTERFACE_MAPPING_ID] = { "mapping", INTERFACE_MAPPING_NB_OPS,
                               NB_ARGS_ARRAY(3, 4, 4) }
};
```

Dans l'exemple ci dessus, les deux premières méthodes de l'interface *block* prennent 4 paramètres, tandis que les deux dernières prennent 3 paramètres.

Maintenir ce tableau synchronisé avec les interfaces devenait difficilement gérable : il devenait nécessaire de le générer automatiquement à partir de la description des interfaces.

3.1.2 Identifiants pour l'espace utilisateur

Pour ouvrir une ressource, les applications utilisateur doivent, en plus du chemin classique, donner l'interface selon laquelle elles souhaitent ouvrir la ressource, sous forme d'un entier. Par exemple :

```
int rd = open(''/home/toto/fichier.txt'', INTERFACE_FILE_ID);
```

Cet identifiant correspond à l'entrée de l'interface dans le tableau présentée à la section précédente. Il permet d'identifier l'interface de manière unique, et il serait plus simple si il était généré automatiquement, plutôt que manuellement pour chaque interface.

De la même façon, pour appeler une méthode d'une interface, les applications utilisateur doivent donner le numéro de la méthode. Il serait intéressant de disposer de définitions comme `INTERFACE_FILE_READ` générées automatiquement.

3.1.3 Méthodes utilisateur, méthodes noyau

Comme énoncé précédemment, les méthodes des diverses interfaces peuvent être utilisées par le noyau ou les applications utilisateur. Ces dernières n'étant pas de confiance, il peut s'avérer nécessaires d'effectuer plus de vérifications si l'appel provient d'applications utilisateur que du

noyau. Ainsi pour la même méthode, on peut avoir deux versions différentes en fonction de l'origine de l'appel.

Une description haut niveau des interfaces permettrait de pallier à ce problème, en définissant pour chaque méthode quels appels sont autorisés.

3.2 Solution implémentée

Le meta-langage *XML* nous est apparu comme un choix pertinent pour nos descriptions d'interfaces. En effet, c'est un méta-langage qui permet de réaliser toutes sortes de descriptions de données de manière très souple et il existe de nombreux outils pour analyser les descriptions *XML*.

Nous avons donc décidé de convertir les anciennes descriptions d'interface sous forme de fichier `.h` en descriptions `.xml`. Par exemple, l'interface *block* a maintenant pour description :

Listing 3.3 – Exemple d'interface en XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<interface name="block">

 <code>typedef enum { BLOCK_DEV_LOCK_READ,
                      BLOCK_DEV_LOCK_WRITE,
                      BLOCK_DEV_UNLOCK} block_lock_op_t;
 </code>

 <method name="read">
  <arg type="struct ures*" name="ur"/>
  <arg type="char*" name="buffer"/>
  <arg type="count_t" name="block_start"/>
  <arg type="count_t*" name="inout_block_count"/>
  </method>

 <method name="write">
  <arg type="struct ures*" name="ur"/>
  <arg type="const char*" name="buffer"/>
  <arg type="count_t" name="block_start"/>
  <arg type="count_t*" name="inout_block_count"/>
 </method>

 <method name="get_device_size">
  <arg type="const struct ures*" name="ur"/>
  <arg type="size_t*" name="out_block_size"/>
  <arg type="count_t*" name="out_block_count"/>
 </method>

 <method name="ctrl_lock" domain="kernel">
  <arg type="struct ures*" name="ur"/>
  <arg type="block_lock_op_t" name="op_type"/>
  <arg type="count_t" name="block_start"/>
```

```
  </method>

</interface>
```

Le langage utilisé pour décrire les interfaces est conforme à une *DTD* (*Document Type Definition*) disponible en annexe. Cette *DTD* permet au programme de vérifier la validité du document *XML*.

Brièvement, voici l'utilisation des différentes balises :

interface Cette balise est la racine de la description de l'interface. Elle permet grâce à l'attribut *name* de donner un nom à l'interface.

code Cette balise permet d'inclure du code (définitions de types, de constantes, inclusions de fichiers), qui sera placé dans les divers `.h` générés. Si l'attribut *domain* n'est pas spécifié, le code est copié à la fois dans les `.h` noyau et les `.h` utilisateur. Sinon, en fonction de la valeur de *domain*, il est copié uniquement dans les `.h` noyau ou dans les `.h` utilisateur.

method Cette balise commence la description d'une méthode, en spécifiant son nom par l'attribut *name* et éventuellement son domaine par l'attribut *domain*. Ce dernier attribut permet de limiter l'accès à certaines méthodes.

arg Cette balise décrit un paramètre d'une méthode à l'aide de deux attributs : *type* pour le type et *name* pour le nom du paramètre.

A partir de cette description *XML*, plusieurs sorties sont générées :
- une description des interfaces
- un tableau global des interfaces
- des identifiants de méthodes et d'interfaces
- des *wrappers* d'appels systèmes

L'utilisation de ces diverses sorties est détaillée dans les sections suivantes. Le programme `utils/kosidl.c` est chargé d'analyser le document *XML* pour en produire les différentes sorties. Il utilise la bibliothèque *libxml2* (voir références [7] et [8]).

3.2.1 Description des interfaces

La première sortie que le programme doit générer est un fichier `.h` correspondant aux anciennes descriptions des interfaces, c'est à dire des structures de pointeurs de fonctions. Ces fichiers `.h` sont utilisés exclusivement par le noyau au niveau des modules implémentant les interfaces.

Cette fonctionnalité correspond à l'option `-i` du programme *kosidl*, et est implémentée dans la fonction `generate_include`. Ainsi, pour générer la description C de l'interface, il suffit d'appeler *kosidl* en lui donnant un identifiant unique pour l'interface et le fichier *XML*. Par exemple, le résultat de la commande `kosidl -i 3 block.xml` est le suivant :

Listing 3.4 – Description C d'une interface générée à partir de la description XML

```
#ifndef __BLOCK_H__
#define __BLOCK_H__

#include <karm/karm.h>
#include <kos/types.h>
#include <kos/errno.h>
```

```
#define INTERFACE_BLOCK __kos_interface_block
#define INTERFACE_BLOCK_ID      3
#define BLOCK_OPS(ops) (( struct __kos_interface_block *)(ops))

typedef enum { BLOCK_DEV_LOCK_READ,
               BLOCK_DEV_LOCK_WRITE,
               BLOCK_DEV_UNLOCK} block_lock_op_t;

struct __kos_interface_block {
  result_t (*read)(struct ures *ur, char *buffer,
                   count_t block_start,
                   count_t *inout_block_count);
  result_t (*write)(struct ures *ur, const char *buffer,
                    count_t block_start,
                    count_t *inout_block_count);
  result_t (*get_device_size)(const struct ures *ur,
                              size_t *out_block_size,
                              count_t *out_block_count);
  result_t (*ctrl_lock)(struct ures *ur,
                        block_lock_op_t op_type,
                        count_t block_start);
};

#define INTERFACE_BLOCK_NB_OPS \
        INTERFACE_SIZE(struct __kos_interface_block)
#endif
```

Ces fichiers de description sont générés automatiquement lorsqu'ils sont inclus dans les divers fichiers C les utilisant. Ceci est effectué par la règle du *Makefile* `modules/karm/Makefile` :

```
interface/%.h: interface/%.xml
        my_id=`./get_intf_id.sh $(INTF_ID_FILE) $(basename $(notdir $@))` && \
          $(TOPSRCDIR)/utils/kosidl -i $$my_id $< > $@
```

Tout d'abord, le script *shell* `get_intf_id.sh` est utilisé pour générer un identifiant unique pour l'interface. Ce script maintient une petite base de données des interfaces (fichier `interfaces.lst`, de manière à assigner à chacune d'elles un identifiant.

3.2.2 Tableau global des interfaces

Le tableau global des interfaces classe les interfaces selon leur identifiant, en donnant leur nom, leur nombre de méthodes et le nombre d'arguments pour chacune des méthodes. Ce fichier est utilisé exclusivement par le système Karm dans le noyau pour vérifier la validité des appels systèmes reçus.

L'option `-d` génère pour une interface donnée une entrée de ce tableau. L'implémentation de cette fonctionnalité se trouve dans la fonction `generate_description` du fichier `kosidl.c`. Ainsi, la commande `kosidl -d block.xml` génère la sortie suivante :

```
[INTERFACE_BLOCK_ID] = { "block" ,
```

```
INTERFACE_BLOCK_NB_OPS,
NB_ARGS_ARRAY(4, 4, 3, 3) },
```

Le script *shell* `gen_interface-desc.sh` se charge de générer le fichier `interface-desc.c` en appelant *kosidl* sur chaque interface. Ce script est appellé depuis le même *Makefile* que pour la description de chaque interface.

Le résultat de l'éxécution du script, pour les 5 interfaces existantes est le suivant :

Listing 3.5 – Tableau global des interfaces généré automatiquement

```
#include <karm/interface.h>
#include "interface/block.h"
#include "interface/char.h"
#include "interface/dir.h"
#include "interface/file.h"
#include "interface/mapping.h"
#include "interface/process.h"

#define NB_ARGS_ARRAY(nb_args...) (int[]){ nb_args, -1 }

struct interface interface_array[] = {

  [INTERFACE_BLOCK_ID] = { "block", INTERFACE_BLOCK_NB_OPS,
                           NB_ARGS_ARRAY(4, 4, 3, 3) },
  [INTERFACE_CHAR_ID] = { "char", INTERFACE_CHAR_NB_OPS,
                          NB_ARGS_ARRAY(3, 3) },
  [INTERFACE_DIR_ID] = { "dir", INTERFACE_DIR_NB_OPS,
                         NB_ARGS_ARRAY(3) },
  [INTERFACE_FILE_ID] = { "file", INTERFACE_FILE_NB_OPS,
                          NB_ARGS_ARRAY(3, 3, 3) },
  [INTERFACE_MAPPING_ID] = { "mapping", INTERFACE_MAPPING_NB_OPS,
                             NB_ARGS_ARRAY(2, 4, 4) },
  [INTERFACE_PROCESS_ID] = { "process", INTERFACE_PROCESS_NB_OPS,
                             NB_ARGS_ARRAY(4, 2, 1, 4, 2, 2, 2) },
};

uint interface_array_size =
 sizeof(interface_array) / sizeof(struct interface);
```

3.2.3 Identifiants pour l'espace utilisateur

Un fichier `.h` est généré par le script *shell* `gen_interface-id_list.sh` et contient la liste des identifiants de toutes les méthodes et de toutes les interfaces, ainsi que les prototypes des *wrappers* d'appels systèmes (voir section 3.2.4). Pour le générer, il utilise l'option `-l` de *kosidl*, implémentée dans la fonction `generate_id_list`.

Voici l'extrait concernant l'interface *block* du fichier généré :

Listing 3.6 – Identifiants pour l'espace utilisateur

```
#ifndef __INTERFACE_ID_LIST_H__
```

```
#define __INTERFACE_ID_LIST_H__

/* This file contains all interface ID's and all method ID's. It's
   useful for user programs (mainly libc).
   It is automatically generated by a script : DO NO EDIT */

#include <kos/types.h>
#include <kos/errno.h>

[...]

/* Begin custom code */
typedef enum { BLOCK_DEV_LOCK_READ, BLOCK_DEV_LOCK_WRITE,
   BLOCK_DEV_UNLOCK} block_lock_op_t;
/* End custom code */

#define KOS_INTERFACE_BLOCK_ID 2
#define KOS_INTERFACE_BLOCK_READ 0
extern result_t __kos_sys_block_read (int /* rd */,
                                      char* buffer,
                                      count_t block_start,
                                      count_t* inout_block_count);
#define KOS_INTERFACE_BLOCK_WRITE 1
extern result_t __kos_sys_block_write (int /* rd */,
                                       const char* buffer,
                                       count_t block_start,
                                       count_t* inout_block_count);
#define KOS_INTERFACE_BLOCK_GET_DEVICE_SIZE 2
extern result_t __kos_sys_block_get_device_size (int /* rd */,
                                                 size_t*
                                                    out_block_size,
                                                 count_t*
                                                    out_block_count)
                                                    ;

[...]

#endif
```

Ce fichier est généré par le module `kos-sys` qui contient les applications et bibliothèques utilisateur (voir `kos-sys/libc/Makefile`).

3.2.4 *Wrappers* d'appels systèmes

Pour utiliser les méthodes des diverses interfaces, les programmes et bibliothèques utilisateur doivent réaliser des appels systèmes en indiquant le numéro de méthode, un tableau des paramètres et le nombre de paramètres. Les méthodes permettant de réaliser les appels systèmes

étant génériques, il faut également transtyper[1] certains paramètres. Ceci étant un peu ennuyeux, notre programme *kosidl* génère un fichier source C contenant une fonction pour chaque méthode de chaque interface (option `-s`). Chacune de ces fonctions réalise l'appel système correspondant directement : le programme ou la bibliothèque n'a plus à s'en préoccuper.

Voici l'extrait concernant l'interface *block* du fichier généré :

Listing 3.7 – Identifiants pour l'espace utilisateur

```
#include "kos-interfaces.h"
#include "syscall.h"

[...]

/* Interface block */
result_t __kos_sys_block_read (int __rd__, char* buffer, count_t
   block_start, count_t* inout_block_count)
{
  return __kos_do_syscall3 (__rd__, KOS_INTERFACE_BLOCK_READ,
                (unsigned long)buffer,
                (unsigned long)block_start,
                (unsigned long)inout_block_count);
}

result_t __kos_sys_block_write (int __rd__, const char* buffer,
   count_t block_start, count_t* inout_block_count)
{
  return __kos_do_syscall3 (__rd__, KOS_INTERFACE_BLOCK_WRITE,
                (unsigned long)buffer,
                (unsigned long)block_start,
                (unsigned long)inout_block_count);
}

result_t __kos_sys_block_get_device_size (int __rd__, size_t*
   out_block_size, count_t* out_block_count)
{
  return __kos_do_syscall2 (__rd__,
     KOS_INTERFACE_BLOCK_GET_DEVICE_SIZE,
                (unsigned long)out_block_size,
                (unsigned long)out_block_count);
}

[...]
```

Ce fichier est généré par le module `kos-sys` qui contient les applications et bibliothèques utilisateur (voir `kos-sys/libc/Makefile`).

[1] *cast*, en anglais

3.3 Remarques

La solution implémentée consistant à utiliser le langage *XML* s'est avérée tout à fait valide. En effet, à l'origine, nous n'avions pas prévu de générer des *wrappers* d'appels systèmes pour l'espace utilisateur. Grâce à la souplesse du *XML* cela a été possible de manière très simple.

Le seul point un peu délicat à mettre en place a été la gestion des dépendances dans les *Makefile* : les divers fichiers doivent en effet être générés de manière automatique, et cela n'a pas été facile.

Chapitre 4

Vers l'application utilisateur ...

Le travail accompli durant la TX a permis d'aller relativement loin dans l'éxécution d'applications utilisateur. Nous allons détailler successivement toutes les améliorations qui ont été apportées, dans l'ordre chronologique.

4.1 Amélioration des fonctions de parcours d'arbre

La gestion des régions virtuelles reposant sur des arbres binaires de recherche, nous avions besoin, avant de commencer tout travail en rapport avec la *VMM* d'améliorer les fonctions de parcours d'arbre. Il s'agissait notamment d'écrire les fonctions permettant de parcourir dans l'ordre croissant et dans l'ordre décroissant tous les noeuds d'un arbre. La contrainte était que ces fonctions devaient être non récursives, car la pile d'éxécution en mode noyau est très limitée, et non extensible.

Le résultat de ce travail se situe dans les fonctions `_splay_visit_in_order` et `_splay_visit_in_reverse_order` du fichier `modules/lib/bst/_splay.c`.

4.2 Création des threads utilisateur

La seconde étape a été de réaliser les fonctions permettant de créer des threads utilisateur. La fonction permettant de créer un thread utilisateur est `create_user_thread` dans `modules/task/_task_uthread.c`. Celle-ci alloue les piles pour le thread, initialise son contexte, et enregistre le thread comme prêt à être éxécuté. L'initialisation du contexte est une partie importante du processus, elle consiste à initialiser la pile noyau du thread avec des valeurs permettant de lancer l'éxécution. Le code permettant cette initialisation a été écrit dans la fonction `init_user_thread_context` dans `modules/x86/task/_thread_cpu_context.c`. Il s'agit de remplir une structure de type `cpu_state_t`, représentant l'état d'un thread, en assignant aux différents registres les bonnes valeurs (adresse des piles, pointeur d'instruction, pointeurs de segment, etc...).

Les threads utilisateur permettent d'éxécuter du code contenu dans un binaire, qu'il convient de charger. Pour cela, un petit chargeur de fichiers binaires au format ELF a été écrit dans le module `elf`. Il permet de mapper en mémoire le code du programme ainsi que ses données, et de trouver l'adresse du point d'entrée dans le programme.

Grâce aux fonctions de création d'une team déjà présentes, aux fonctions de création d'un thread utilisateur que nous venions d'écrire et du chargeur ELF, nous pouvions presque lancer un thread.

Il manquait encore deux aspects :

- La mise à jour du *TSS* à chaque changement de contexte d'un thread utilisateur. Le *TSS*, *Task State Segment* est une structure de donnée des processeurs Intel. En théorie, il est utilisé pour sauvegarder tout le contexte d'une tâche, mais en réalité sous Kos est dans beaucoup d'autres systèmes, il est juste utilisé pour qu'un thread utilisateur retrouve l'adresse de sa pile noyau lors d'une interruption. L'adresse de cette pile étant différente pour chaque thread, il convient de mettre à jour le TSS en conséquence. Cette mise à jour intervient dans `scheduler_retrieve_next_thread` du fichier `modules/scheduler/_scheduler.c`.
- Le changement d'espace d'adressage. Lors d'un changement de contexte, si le thread d'origine est dans une team différente du thread destination, il faut changer d'espace d'adressage. C'est ce qui est réalisé dans `modules/task/_task_utils.c: set_current_thread` en appelant `as_switch`.

Ayant implémenté ces fonctionnalités, nous étions capable de lancer un nouveau thread utilisateur. Nous compilions un petit programme grâce à *gcc*. Ensuite, dans l'initialisation de Kos, nous pouvions créer une team, y charger l'éxécutable en question, puis créer le thread. L'éxécution de ce thread était alors fonctionnelle.

4.3 Appel système

Notre thread utilisateur s'éxécute. Toutefois, il ne peut pour l'instant pas interagir avec le noyau : il ne peut pas ouvrir des fichiers, écrire à l'écran, etc... Il fallait donc implémenter l'appel système.

Le point d'entrée de l'appel système est la fonction `modules/x86/task/_task.c: syscall_entry_point`. Il s'occupe de récupérer les arguments de l'appel système dans les différents registres du processeur, et de les passer à la fonction `syscall`, implémentée dans `modules/karm/syscall.c`. Le registre `eax` contient le numéro du descripteur de ressource, le registre `ebx` contient le numéro de la méthode à éxécuter, le registre `ecx` contient l'adresse du tableau de paramètres et le registre `edx` contient le nombre de paramètres.

La fonction `syscall` récupère la *User ressource* identifié par le numéro fourni, puis vérifie que le numéro de la méthode et le nombre d'arguments fournis sont valides. Enfin, elle appelle la méthode elle même, en fonction du nombre d'arguments.

On peut donc maintenant faire des appels systèmes depuis l'espace utilisateur. Les *wrappers* d'appels systèmes (voir section 3.2.4) sont fonctionnels, notre application utilisateur peut maintenant interagir avec le noyau, mais elle ne peut pas lire et écrire sur les entrées sorties standard que sont les descripteurs 0, 1 et 2 sous Unix.

4.4 *Libcharfile* et entrées/sorties standard

Il convenait donc d'ajouter dans chaque team les descripteurs 0, 1, 2, ouvert selon l'interface *fichier*, pour que les threads utilisateurs puissent lire/écrire sur l'entrée/sortie standard comme sous Unix, c'est à dire comme dans des fichiers.

Pour l'heure, la première team contenant des threads utilisateur est initialisée avec un terminal comme entrée et sortie standard. Or, le pilote de périphérique *tty* n'exporte que l'interface *INTERFACE_CHAR_ID*, et non l'interface *INTERFACE_FILE_ID*. Il a donc fallu écrire une bibliothèque, `libcharfile`, qui ajoute le support de l'interface *file* à une ressource supportant l'interface *char*. Le code de cette bibliothèque se trouve dans le répertoire `modules/lib/charfile`.

Une fois cette bibliothèque implémentée, il suffisait d'ouvrir 3 fois le terminal, afin de permettre aux threads de la team de lire et écrire sur l'entrée/sortie standard. Ceci est réalisé dans la fonction `init_first_user_process` du fichier `modules/kos/wolfgang.c`.

4.5 Ressource *process*

Maintenant que notre premier thread utilisateur tourne, et fait des appels au noyau, il fallait lui donner la possibilité de créer de nouveaux processus, d'éxécuter d'autres fichiers, d'ouvrir des fichiers, etc...

Pour réunir toutes ces fonctionnalités, qui ne se rapportent pas à une ressource bien précise comme une socket ou un fichier, nous avons décidé de créer une ressource appelée *process*, présente dans chaque team sous le descripteur numéro `3` (`KOS_SELF`). Cette ressource process permet à chaque team d'appeler des fonctions du noyau comme *fork()* (création d'une nouvelle team), *exec()* (éxécution d'un nouveau programme), *brk()* (allocation de mémoire), etc ...

L'interface *process* est définie dans le fichier `modules/karm/interface/process.xml`, et comporte les méthodes suivantes :

open Ouverture d'une ressource, spécifiée par un chemin et une interface.

close Fermeture d'une ressource, spécifiée par son descripteur.

fork Duplication de la team courante.

exec Éxécution d'un nouveau programme dans la team courante.

getpid Récupération de l'identifiant de la team courante

getppid Récupération de l'identifiant de la team parente

brk Allocation/libération dynamique de mémoire

L'implémentation de cette ressource est réalisée dans le fichier `modules/task/_task_kres.c`. Dans les sections suivantes, nous détaillons l'implémentation des principaux appels systèmes de cette *process* ressource.

4.5.1 Ouverture et fermeture des fichiers

L'ouverture et fermeture des fichiers a été relativement simple à implémenter, puisque des fonctions *open()* et *close()*, fonctionnant sur des *User Ressource* existaient déjà dans le module `karm`. Il s'agissait donc d'utiliser ces fonctions, puis d'insérer ou de supprimer la *User Ressource* du tableau des ressources de la team courante. Pour cela, les fonctions `task_insert_ures` et `task_delete_ures` ont été implémentées dans le fichier `modules/task/_task_ures.c`.

4.5.2 Création d'un nouveau processus

La création d'un nouveau processus a lieu lors de l'appel système *fork()*, qui duplique le processus appelant pour créer un processus fils. Dans ce cas, le terme de processus est utilisé pour designer un ensemble constitué d'une team et d'un thread. Il n'est ainsi pas possible d'appeler *fork()* depuis une team comportant plusieurs threads. En effet, on ne saurait que faire des threads autre que le thread ayant appelé *fork()*. C'est une contrainte que nous avons pu retrouver dans d'autres systèmes d'exploitation.

Le code de *fork()* est situé dans la fonction `task_fork` du fichier `modules/task/_task_team.c`, et réalise les opérations suivantes :

1. Vérification que la team courante (donc appelante) ne possède qu'un seul thread, de type thread utilisateur.
2. Création d'une nouvelle team, via la fonction `create_team`.
3. Duplication du tableau des descripteurs d'*User Ressource* via la fonction `ures_array_copy`, implémentée dans `modules/karm/ures.c`. Elle utilise la fonction `ures_copy` sur chaque *User Ressource*. Pour cela, il a été nécessaire d'ajouter un *callback* supplémentaire dans la `struct view`, afin de disposer d'une fonction pour copier les données privées d'une *User Ressource* (voir `modules/karm/view.h`.
4. Duplication de l'espace d'adressage par la fonction `as_copy` du fichier `modules/vmm/_vmm_as.c`. Elle initialise un nouvel espace d'adressage, et copie les régions en utilisant la fonction `vr_copy`. Un point important est le remappage dans les nouvelles régions des pages physiques des anciennes régions, avec éventuellement une protection contre l'écriture si les régions sont mappées en MAP_PRIVATE. C'est en effet cette protection contre l'écriture qui permettra de faire fonctionner le mécanisme de *Copy On Write* en déclenchant un défaut de page lors de la première écriture (voir les appels à `protect_virtual_range` et `arch_range_dup`).
5. Copie des informations relatives à l'allocation des piles utilisateurs, en utilisant la fonction `copy_user_stack_slots`, voir section 4.8
6. Copie du thread utilisateur dans la nouvelle team en utilisant la fonction `copy_user_thread`, implémentée dans le fichier `modules/task/_task_uthread.c`.

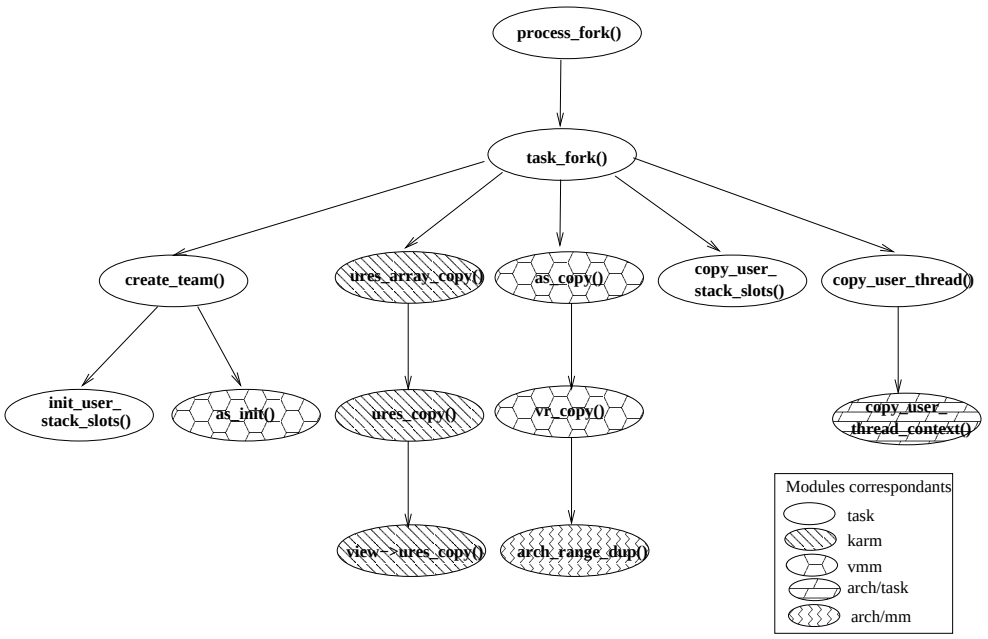


FIG. 4.1 – Graphe d'appel simplifié de `task_fork`

La mise en place de *fork()* a donc nécessité l'implémentation de nombreuses fonctions, l'ajout d'un *callback* dans toutes les *views*, etc... Toutefois, l'implémentation de *fork()* ne suffit pas à faire fonctionner de nouveaux processus : la gestion de la mémoire virtuelle a du également être améliorée, notamment en ce qui concerne la gestion des défauts de page, voir section 4.7.

4.5.3 Chargement d'un nouveau programme

Le chargement d'un nouveau programme est réalisé par l'appel système *exec()*. Il permet de remplacer l'image mémoire courante par l'image mémoire d'un autre programme, dont le fichier binaire est passé en argument.

Cet appel système est implémenté dans la fonction `task_exec` du fichier `modules/task/_task_team.c`, et réalise les opérations suivantes :

1. Vérification que la team courante (donc appelante) ne comporte qu'un seul thread.
2. Ouverture du fichier binaire contenant le programme à éxécuter selon l'interface *FILE*
3. Suppression de toutes les régions de l'espace d'adressage, fonction `as_empty` du fichier `modules/vmm/_vmm_as.c`.
4. Préparation du chargement du fichier binaire, en utilisant la fonction `binfmt_prepare` du fichier `modules/kbs/binfmt.c`. Cette fonction charge l'entête du fichier binaire en mémoire.
5. Fermeture de toutes les *User Ressources* ouvertes, sauf 0 (entrée standard), 1 (sortie standard), 2 (sortie d'erreur) et 3 (ressource *process*).
6. Chargement du binaire par la fonction `binfmt_lookup_handler_and_load` du fichier `modules/kbs/binfmt.c`. Cette fonction va analyser le fichier binaire en fonction de ce format (seul ELF supporté pour l'instant), et mapper en mémoire les portions de fichiers correspondants au code, aux données initialisées et aux données non initialisées.
7. Mise à jour de l'adresse courante du tas (zone pour l'allocation dynamique de mémoire, manipulable grâce à *brk()*), en utilisant la fonction `as_update_heap_start` du fichier `modules/vmm/_vmm_as.c`.
8. Mapping de la pile utilisateur en mémoire. Toutes les régions ayant été supprimées, il faut remapper cette pile via un appel à `as_map_ures`.
9. Initialisation du contexte du thread par la fonction `init_user_thread_context`, ce qui a pour effet d'initialiser la pile avec les valeurs permettant de lancer l'éxécution du thread.

Grâce à cette fonction, un programme utilisateur peut appeler *exec()* pour éxécuter un programme différent.

4.5.4 Gestion du tas

Le tas est une région de mémoire qu'on peut dynamiquement agrandir ou diminuer en utilisant l'appel système *brk()*. Cet appel système est fondamental, car c'est sur lui que repose le fonctionnement de l'allocateur mémoire de la librairie standard C constitué des fonctions `malloc()` et `free()`.

L'appel système `brk()` est implémenté dans la fonction `process_brk` du fichier `modules/task/_task_kres.c`. La principale fonction utilisée est `as_change_heap` du fichier `modules/vmm/_vmm_as.c`, qui permet de changer la taille du tas. Cette fonction gère trois cas : le rétrécissement du tas, sa création ou son agrandissement. L'implémentation de cette fonction a été réalisée en utilisant les nouvelles fonctions de la VMM, voir section 4.7.

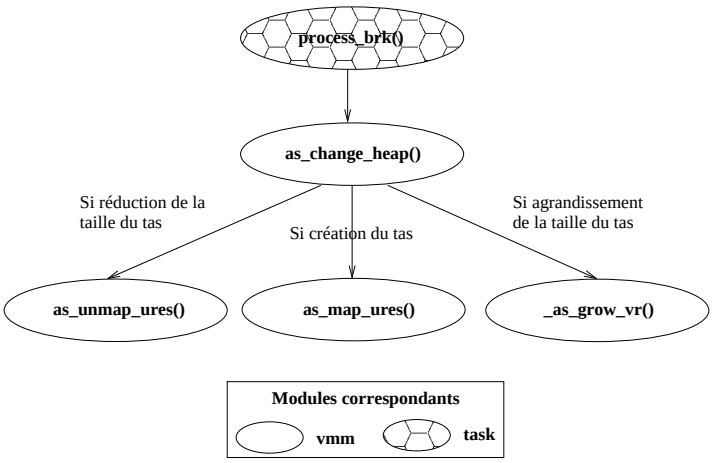


FIG. 4.2 – Graphe d'appel simplifié de `process_brk`

4.6 Chargeur ELF amélioré

ELF est un format de fichier binaire. Il décrit l'organisation des données dans un éxécutable. Il permet notamment de distinguer dans le fichier binaire quelles sont les zones correspondant au code, les zones correspondant aux données, et donne diverses informations concernant le chargement de ce binaire en mémoire.

Au début de la TX, un chargeur minimal avait été mis en place, ne supportant que le chargement des zones de code et de données. Par la suite, un nouveau chargeur a été réalisé, apportant de nouvelles fonctionnalités :

- Couche d'abstraction *binfmt* (*Binary Format*) permettant de s'abstraire du format binaire sous-jacent. Le système fait appel aux fonctions de cette couche d'abstraction qui se charge de détecter automatiquement quel est le format binaire. Le code de cette couche d'abstraction est dans le fichier `modules/kbs/binfmt.c`.
- Plus de vérifications concernant la validité du fichier binaire ELF.
- Recherche d'un interpréteur, inutilisé pour l'instant, mais qui sera utile lorsque l'on supportera les bibliothèques partagées.
- Support des zones ayant une taille mémoire supérieure à la taille fichier, c'est à dire les zones comportant des données non initialisées (*bss*).

4.7 Amélioration de la VMM

La gestion de la mémoire virtuelle est un aspect important d'un système d'exploitation. Dans l'histoire de KOS, différents essais avaient été réalisés pour comprendre comment la gestion de la mémoire virtuelle fonctionnait, mais rien de concret et complet n'était présent. Au début de la TX, seul un début de fonction permettant de créer des régions virtuelles était disponible et elle n'était pas complète.

4.7.1 Gestion des régions

- Amélioration de la fonction permettant de mapper des *User Ressource*, `as_map_ures`, en supportant le positionnement fixe (*MAP_FIXED*), et en supportant correctement les régions anonymes.

– Création de la fonction `as_unmap_ures` pour démapper une portion de l'espace d'adressage. Cette fonction est également utilisée par `as_map_ures` dans le cas où un positionnement fixe de la nouvelle région est demandé. Pour fonctionner, `as_unmap_ures` utilise les fonctions `_as_unmap_split_vr`, `_as_unmap_shrink_vr`, `_as_unmap_del_vr` respectivement pour découper une région, réduire une région ou détruire une région.

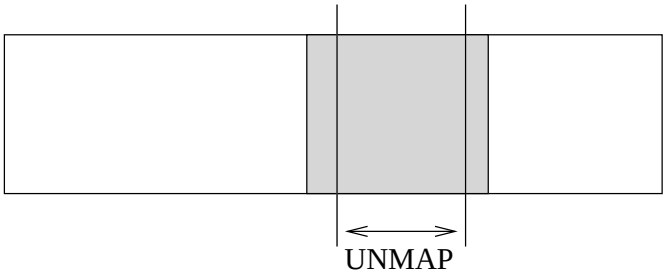


FIG. 4.3 – Exemple d'utilisation de `_as_unmap_split_vr`

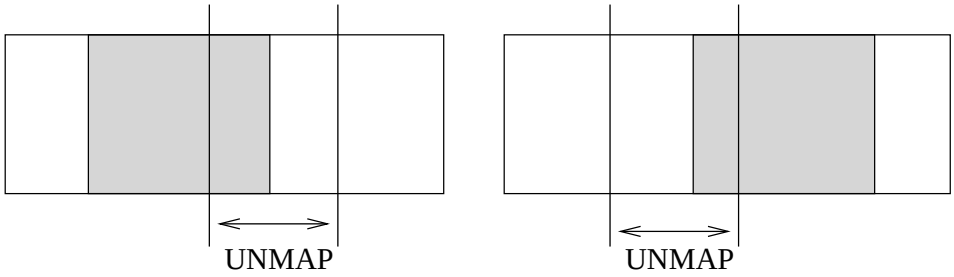


FIG. 4.4 – Exemples d'utilisation de `_as_unmap_shrink_vr`

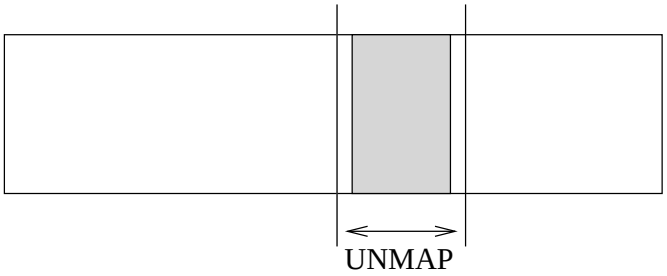


FIG. 4.5 – Exemple d'utilisation de `_as_unmap_del_vr`

– Création des fonctions `as_copy` et `vr_copy` pour copier un espace d'adressage ou une région. Ces fonctions sont utilisées durant un *fork()*.
– Création de la fonction `as_empty` pour vider l'espace d'adressage de ses régions. Cette fonction est utilisée durant un *exec()*.
– Création de la fonction `as_dump` pour afficher toutes les régions d'un espace d'adressage, ainsi que la page physique associée à chaque page virtuelle. C'est une fonction extrêmement utile pour le déboggage.
– Création des fonctions `as_update_heap_start` et `as_change_heap`.

4.7.2 Gestion des défauts de page

La gestion des défauts de page a été entièrement revue, afin de supporter les différents types de régions virtuelles. La gestion des défauts de page est une fonctionnalité importante du système, car elle permet dynamiquement d'allouer de la mémoire et d'y charger des données. Dans Kos, c'est la fonction `as_page_fault` qui est chargée de la résolution des défauts de page. Son algorithme est le suivant.

- Si le défaut de page intervient dans la zone (et non la région, voir 4.8) de pile du thread courant et est du à une page non présente, alors la région de la pile est étendue pour permettre au défaut de page de se produire.
- Sinon, recherche de la région concernée par le défaut de page. Si l'adresse n'est pas située dans une région, renvoi d'une erreur.
- Si l'accès est en écriture alors que la région est en lecture seule, renvoi d'une erreur.
- Si l'accès est en écriture, que le droit d'écriture sur la région est présent, et que la page est déjà présente, c'est un *Copy On Write* qui a lieu suite à un *fork()*. L'étape 1 du traitement du *COW* a lieu, elle consiste à sauvegarder dans un buffer temporaire le contenu de la page.
- Ensuite, que ce soit un *COW* ou non, une nouvelle page est mappé à l'adresse fautive (après avoir été allouée).
- Si les mêmes conditions que celles nécessaires pour lancer l'étape 1 du *COW* sont trouvées, alors l'étape 2 commence, elle consiste à recopier le contenu du buffer temporaire dans la page nouvellement allouée, puis à retourner un succès. Le traitement du *COW* se termine donc.
- Ensuite, si la région est le mapping d'un fichier, on fait appel à la méthode `page_in` de l'interface *MAPPING* pour charger en mémoire le contenu du fichier correspondant.
- Sinon, c'est une région de type anonyme, la nouvelle page est initialisée avec des zéros.

A noter qu'à l'heure actuelle, un retour d'erreur d'un défaut de page provoque l'arrêt du système. Plus tard, il faudra implémenter la destruction des threads pour pouvoir les détruire quand un accès invalide sera détecté (équivalent du *SIGSEGV* ou *Segmentation Fault* sous Linux)

4.8 Allocation des piles utilisateur

Chaque thread utilisateur a besoin pour s'éxécuter d'une pile au niveau utilisateur. Une zone de la carte mémoire est réservée aux piles utilisateur de ces threads, à partir de l'adresse `USER_STACK_AREA_START` qui vaut 3.5 Go sur plateforme *x86* et pour `USER_STACK_AREA_SIZE` octets, qui vaut 512 Mo.

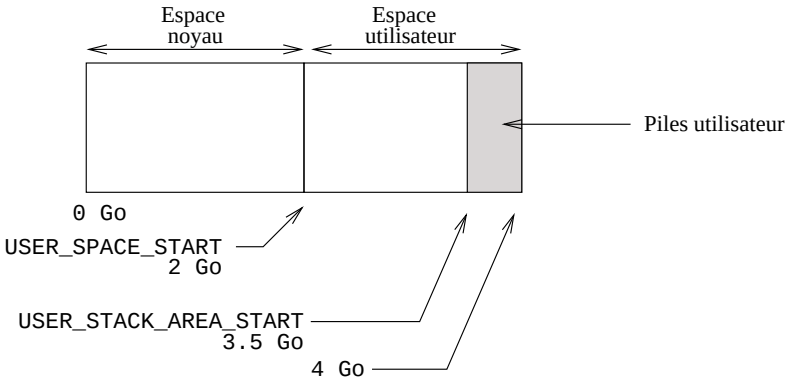


FIG. 4.6 – La zone mémoire pour les piles utilisateur

Chaque team pouvant héberger plusieurs threads utilisateur, il fallait un moyen de distribuer cet espace entre les différents threads. D'autre part, il fallait un mécanisme permettant d'attribuer des tailles de piles différentes à chaque thread.

La solution choisie a été de découper cet espace en slots de taille `USER_STACK_GRANULARITY` (1 Mo par défaut). Chaque pile a donc une taille multiple de cette granularité. Dans chaque team, on trouve un tableau contenant autant d'entrées que de slots. Chaque entrée contient un état (occupé ou libre) ainsi qu'une valeur. Si l'état est libre, alors la valeur contient le nombre de slots libres contigus à partir du slot courant. Si l'état est occupé alors la valeur contient le nombre de slots occupés par la pile courante (seul le premier slot d'une pile contient cette valeur, les autres slots sont à 0).

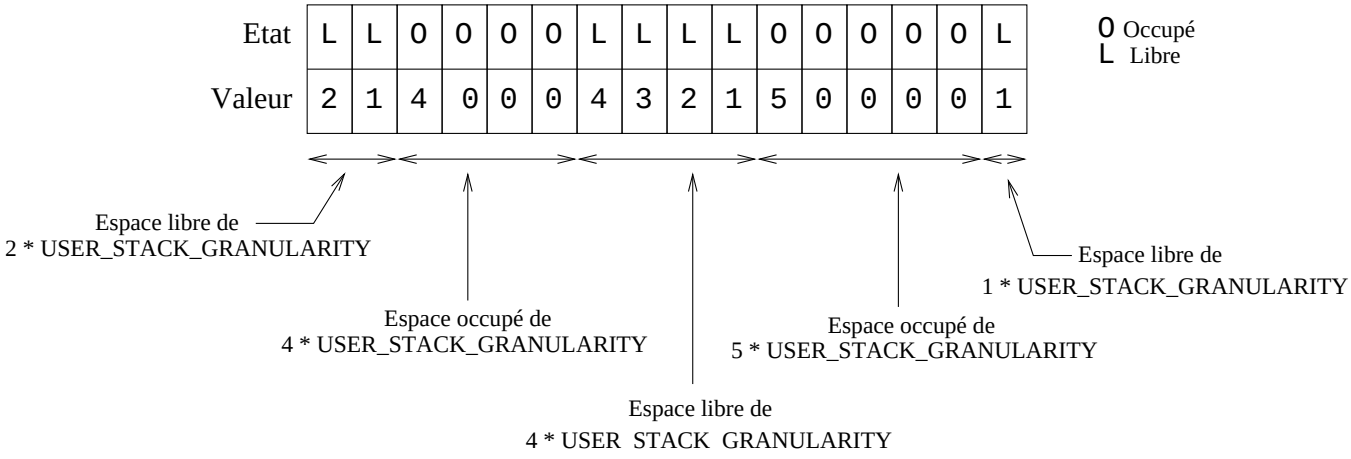


FIG. 4.7 – Exemple d'état d'un tableau des slots

A partir de ce tableau, un algorithme d'allocation et libération a été écrit dans les fonctions `alloc_user_stack_slots` et `free_user_stack_slots` du fichier `modules/task/_task_uthread.c`. L'algorithme d'allocation est de type *best-fit* : plutôt que de couper un espace de slot trop grand, il cherche celui dont la taille est la plus proche de la taille recherchée. L'algorithme de libération fusionne les zones libres adjacentes. Ces deux algorithmes permettent donc de limiter la fragmentation.

Chapitre 5

PCI

A la fin de la TX, un début de pilote de périphérique *PCI* a été mis en place. Le bus *PCI* permet d'accéder aux différentes cartes présentes sur la machine, ainsi qu'aux ponts permettant d'accéder aux cartes *ISA* ou *AGP*.

Le pilote de périphérique implémenté dans le module `pci` ne s'occupe pour l'instant que de la détection des différentes cartes : il identifie leur rôle (carte réseau, carte graphique, ...), leur constructeur, leur modèle ainsi que d'autres caractéristiques intéressantes comme les ports d'entrées sorties ou l'adresse des zones mémoires permettant d'accéder à la carte.

Ce pilote de périphérique *PCI* devra être étendu pour permettre aux autres pilotes de périphériques de détecter les cartes correspondants aux modèles qu'ils supportent et d'accéder aux informations de configuration. Une des premières cartes PCI à supporter sera sans doute une carte réseau.

Chapitre 6

Conclusion et perspectives

Pour le projet Kos, la réalisation de cette TX a permis de grandes avancées. En moins de 6 mois, un système performant et souple de description des interfaces a été mis en place, et Kos est désormais capable de lancer des applications utilisateur et d'en créer de nouvelles. Kos doit maintenant rentrer dans une phase de stabilisation et de tests durant laquelle l'accent devra être mis sur la synchronisation. En effet, c'est une problématique que nous avons quelque peu eclipsé depuis quelques mois, mais qui reste importante, d'autant plus que le système est entièrement préemptif. Pour la suite des fonctionnalités, nous avons déjà regardé comment porter la *GNU libc* au dessus de Kos. Une fois la phase de stabilisation terminée, ce sera sans aucun doute la priorité.

Les apports personnels de cette TX à Mélanie et Thomas ont été différents :
- Pour Mélanie, cette TX a été l'occasion de comprendre le fonctionnement interne d'un système d'exploitation. Grâce à la réalisation du Kosidl, elle a pu parfaire ses connaissances en langage C en utilisant *libxml2*, bibliothèque qui lui était inconnue.
- Pour Thomas, cette TX a été l'occasion d'expliquer à une nouvelle personne le fonctionnement complet de Kos et de participer activement à un projet personnel.

Le bilan de cette TX nous paraît donc très positif, à la fois en terme d'apports au projet Kos et en terme d'apports personnels.

Bibliographie

[1] Jeff Bonwick, Sun Microsystems, *The Slab Allocator : An Object-Caching Kernel Memory Allocator*, http://kos.enix.org/pub/slaballocator.pdf.gz

[2] Bryan Ford, Erich Stefan Boleyn, Kunihiro Ishiguro, Okuji Yoshinori, *The Multiboot Specification*, http://kos.enix.org/pub/multiboot.ps.gz

[3] Thomas Petazzoni, *Compiler, tester et débugger* KOS, http://kos.enix.org/~d2/snapshots/kos_current/doc/testingfr.pdf

[4] David Decotigny, Thomas Petazzoni, *Concepts fondamentaux et structure du noyau Linux*, Linux Magazine Hors Série 16, Septembre 2003, p16-27.

[5] David Decotigny, Thomas Petazzoni, *Voyage à la frontière du noyau*, Linux Magazine Hors Série 16, Septembre 2003, p48-51.

[6] Yannick Fourastier, *Architecture des noyaux : typologie et éléments de comparaison*, Linux Magazine Hors Série 16, Septembre 2003, p28-41.

[7] Yves Mettier, *La bibliothèque libxml2 et petits fichiers XML*, Linux Magazine 51, Juin 2003, p42-51.

[8] *Reference Manual for libxml2*, http://www.xmlsoft.org/html/libxml-lib.html

[9] Charles D. Cranor, *Design and implementation of the UVM virtual memory system*, Washington University, Department of Computer Science, Juillet 1998, http://kos.enix.org/pub/uvm.ps.gz.

[10] Mel Gorman, *Understanding the Linux memory virtual memory manager*, Juillet 2003, http://www.csn.ul.ie/~mel/projects/vm/

Annexe A

Liste des modules

Module	Description
console	Pilote de périphérique pour la console texte
debug	Fonctions de débuggage.
elf	Le chargeur de fichiers éxécutables au format ELF.
fs/devfs	Le système de fichiers des périphériques du système (mounté dans `/dev/`).
fs/fakefs	Le système de fichiers basique monté à la racine au démarrage.
fs/fat	Le système de fichiers FAT (lecture seule pour le moment).
i8042	Le pilote de périphérique pour le chipset *i8042*, gérant le clavier PS2 et la souris PS2.
ide	Le pilote de périphérique IDE
idt	Gestion des interruptions
init	Le module appellé à l'initialisation du système.
karm	Kos Advanced Ressource Management.
kgc	Le garbage collector du noyau
kitc	Fonctions de synchronisation et de communication internes au nouau (sémaphores, mutexes, boîtes aux lettres).
klavier	Le pilote de périphérique pour le clavier.
kmem	Gestionnaire de mémoire pour le noyau
kos	Module principal chargé de la fin de l'initialisation du système.
lib/blockfile	Transforme un périphérique bloc en un fichier.
lib/bst	Librairie de gestion des arbres binaires de recherche.
lib/charfile	Transforme un périphérique caractère en un fichier.
lib/filemap	Permet de mapper un fichier en mémoire
lib/list	Librairie de gestion des listes chaînées.
lib/std	Librairie standard (printf, memcmp, memcpy, strlen, strcpy ..).
part	Pilote de périphérique pour partitions
pmm	Gestion de la mémoire physique
scheduler	L'ordonnanceur
task	Gestion des tâches
test	Suites de tests
tty	Gestionnaire de terminaux
vmm	Gestion de la mémoire virtuelle
x86/debug	Fonctions de débuggage propres à la plateforme x86.
x86/lib	Diverses fonctions propres à la plateforme x86.
x86/mm	Fonctions pour la gestion de la mémoire propres à la plateforme x86.
x86/task	Fonctions pour la gestion des tâches propres à la plateforme x86.

Annexe B

Exemple de `ioctl`

```
static int lp_ioctl(struct inode *inode, struct file *file,
                    unsigned int cmd, unsigned long arg)
{
  [...]
  switch ( cmd )
    {
    case LPCHAR:
      LP_CHAR(minor) = arg;
      break;
    case LPABORT:
      if (arg)
        LP_F(minor) |= LP_ABORT;
      else
        LP_F(minor) &= ~LP_ABORT;
      break;
    case LPABORTOPEN:
      if (arg)
        LP_F(minor) |= LP_ABORTOPEN;
      else
        LP_F(minor) &= ~LP_ABORTOPEN;
      break;
    case LPCAREFUL:
      [...]
    case LPGETSTATUS:
      [...]
    case LPRESET:
      [...]
    case LPGETFLAGS:
      [...]
    case ...:
    default:
      retval = -EINVAL;
    }
  [...]
}
```

Annexe C

La *DTD* des interfaces

Listing C.1 – La *DTD* des interfaces

```
<!ELEMENT interface (code*,method+)>
<!ATTLIST interface
 name CDATA #REQUIRED
>

<!ELEMENT code (#PCDATA)>
<!ATTLIST code
 domain CDATA #IMPLIED
>

<!ELEMENT method (arg+)>
<!ATTLIST method
 name CDATA #REQUIRED
 domain CDATA #IMPLIED
>

<!ELEMENT arg EMPTY>
<!ATTLIST arg
 type CDATA #REQUIRED
 name CDATA #REQUIRED
>
```