Réordonnancement de la mémoire d'un processus

NON CONFIDENTIEL

Laurent Valeyre

29 juin 2001

Résumé

Ce stage s'inscrit dans le cadre du projet **MPC** développé au laboratoire d'informatique de Paris 6 sous la direction d'Alain Greiner. Il a pour but la réalisation d'une machine parallèle à coût réduit, dont les noeuds de calcul sont des PCs interconnectés par un réseau haut débit.
Mon travail, au cour du stage a consisté à transporter la globalité de la zone mémoire du processus dans une zone oú celle ci sera contigüe.

Abstract

This training period is a part of the global research project called MPC developped in the research computer science laboratory of Paris 6, under the direction of Pr. Alain Greiner. The project'aim is to build a low cost parallel computer, based on standart PCs as processing node interconnected with a high performance network.
My role in the project was to move up to cover all process's memory in a contiguous location.

Remerciement

Le tiens à remercier les enseignants-chercheurs , les thésards, et plus généralement toute l'équipe du département **ASIM** du **LIP6** pour le soutien qu'ils ont pu m'apporter, leur ouverture d'esprit, et leur bonne humeur.

Je remercie en particulier, Alain Greiner pour m'avoir permis d'effectuer ce stage.

Je remercie également Philippe Lalevée et Daniel Millot(**INT**) pour leur soutien.

Enfin, je remercie plus encore, Olivier Glück qui m'a soutenu et encadré durant tout ce stage. Il a eu la patience de m'expliquer , de résoudre mes problèmes tout en me laissant une très grande autonomie de travail.

Table des matières

Chapitre 1

Sujet du stage

1.1 Objectif

On cherche à éviter les appels systèmes lors des communications dans une machine parallèle de type **Grappes de PCs**. Une des techniques possibles pour atteindre cet objectif est de recopier systématiquement la mémoire du processus **UNIX** qui souhaite communiquer dans une zone de mémoire physique contigüe, au moment de son démarrage.

1.2 Description

Ce stage s'inscrit dans le cadre du projet **MPC** développé au laboratoire d'informatique de Paris 6. La machine parallèle **MPC** est constituée de cartes processeurs Pentium ainsi que d'un réseau d'interconnexion rapide Gigabit composé de cartes réseau utilisant la technologie **HSL**. Des couches logicielles permettant d'exploiter au mieux les possibilités de ce réseau à haute performance ont été développées sur le système **UNIX**.
Les couches basses de communication de la machine **MPC** ne peuvent transporter que des zones mémoire qui soient contigües en mémoire physique. L'objectif de ce stage est d'étudier comment est géré la mémoire d'un processus sous **UNIX**, et de développer un système qui permettrait de recopier juste après son démarrage tout ou partie de la mémoire du processus dans une zone de mémoire qui soit contigüe en mémoire physique. L'allocation de la mémoire au processus devient alors statique. Ce mécanisme est possible car une des couches basses du système d'exploitation de la machine **MPC** permet de réserver un zone de mémoire physique contigüe de plusieurs Mega-Octets. Les développements se feront en langage C sous **UNIX** FreeBSD-3.4 et/ou **Linux** .
Enfin, le stagiaire sera intégré à l'équipe de recherche qui travaille sur le projet **MPC**. Cela pourra être pour lui l'occasion de suivre les évolutions d'un tel projet en assistant aux réunions mensuelles qui rassemblent les différents

partenaires du projet **MPC**. Ce travail pourra donner lieu à une publication.

1.3 Moyens utilisés

Le stagiaire disposera de l'ensemble des sources du système de communication de la machine **MPC**, d'une plate-forme de développement sous **Linux** et de l'ensemble de la documentation de l'équipe.

1.4 Encadrants

Directeur de stage
- Alain GREINER, professeur
- Olivier GLÜCK, laboratoire d'informatique de Paris 6

Conseiller d'étude
- Philippe LALEVÉE, enseignant-chercheur à l'**INT**

Chapitre 2

Introduction et contexte du stage

2.1 Le laboratoire

Ce stage s'est effectué dans le laboratoire de recherche du département **ASIM** du **LIP**6(Laboratoire d'Informatique de Paris 6)

2.1.1 L'université de Paris 6

Principale héritière de l'ancienne Faculté des sciences de Paris en Sorbonne, elle forme 3000 cadres haute technologie par an. L'Université Pierre et Marie Curie est la première Université scientifique et médicale de France.
L'université en quelques chiffres

- 36000 étudiants dont 10000 en $3eme$ cycle
- 2600 enseignants-chercheurs
- 1300 chercheurs
- 150 laboratoires de recherche
- 2000 **DEA** ou **DESS** sont délivrés chaque année (10% de la totalité française)
- 900 Thèses de doctorat (20% de la totalité française)

L'université de Paris 6 dispense plusieurs types de formations :

- Formations scientifiques
- Formations médicales
- Formations professionnels
- Ecoles d'ingénieurs
- Formations permanentes (5000 stagiaires par an)

2.1.2 Le laboratoire d'informatique de Paris 6

Le **LIP**6 est l'instrument de la politique scientifique de l'université Pierre et Marie Curie en matière d'informatique, exprimée dans le cadre du contrat

quadriennal 2001 – 2005 qui lie l'université, le **CNRS** et le ministère.
Le **LIP**6 en quelques chiffres :

- 9 thèmes de recherche
- un effectif d'environ 320 personnes (doctorants compris)
- 114 enseignants-chercheurs et chercheurs permanents
- 165 thésards et post-docs

Le laboratoire se compose de 9 thèmes de recherche, dont les activités recouvrent une large part de l'informatique :

- Algorithme numérique et parallélisme (**ANP**)
- Apprentissage et acquisition de connaissances (**APA**)
- Architecture des systèmes intégrés et micro-élecronique(**ASIM**)
- Calcul formel (**CALFOR**)
- Objets et Agents pour Systèmes d'Information et de Simulation (**OASIS**)
- Réseaux et performances (**RP**)
- Sémantique, preuve et implantation (**SPI**)
- Systèmes répartis et coopératifs (**SRC**)
- Systèmes d'aide à la décision et à la formation (**SYSDEF**)

Outre leurs activités propres, ces thèmes collaborent dans divers actions transversales et sont engagés dans de nombreux projets en partenariat avec des entreprises.
Huit projets transversaux déterminent la politique scientifique du **LIP**6. La machine **MPC** fait partie de ces projets.

2.1.3 Le département ASIM du LIP6

Le département **ASIM** concentre ses recherches dans les domaines suivants :

- La conception de circuits intégrés hautement complexes
- Le développement d'outils **CAO** avancés pour la **VLSI**
- L'architecture matériel des systémes

Son effectif est de 65 personnes environ.
Les enseignements dispensés par le département sont :

- **DEA** en architecture des systèmes Intégrés et Micro-Electronique (**ASIME**)
- **DESS** de Circuits Intgrés et Systèmes Analogiques Numériques
- Maîtrise Informatique (Option Architecture et Maîtrise **EEA** (Option **MEMI**))

La recherche du département se décompose principalement en trois projets :

- La machine **MPC**
- Projet d'indexation multimédia
- Projet **CAO** de circuits et systèmes (Alliance)

Alliance est un outil avancé de **CAO** pour la **VLSI**. Les activités de l'indexation Multimédia sont l'extraction d'information de contenu, la recherche d'information et les ontologies et la représentation de connaissances.

Le projet **MPC** est un projet de longue haleine. L'idée générale est d'aider certains demandeurs de puissance de calcul du **LIP6** à mener des expériences sur la machine **MPC** développée au laboratoire dans le thème **ASIM**. Mais surtout, le but du projet **MPC** est la conception d'une machine parallèle performante et à faible coût. La contribution du laboratoire s'est traduite par l'acquisition d'un réseau à quatre noeuds. Ceci est suffisant pour mettre au point des expériences mais il serait bon d'avoir au moins huit noeuds pour obtenir des résultats significatifs.

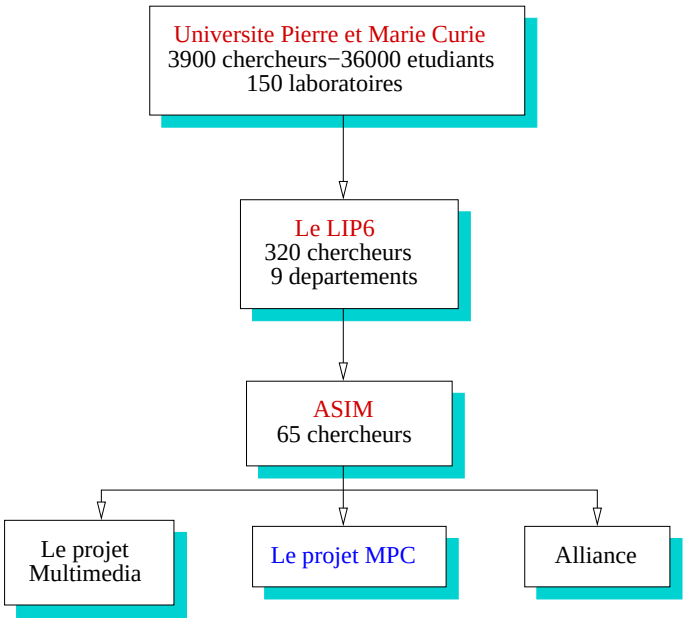

FIG. 2.1 – De l'UPMCP6 au projet **MPC**

Chapitre 3

La machine MPC

3.1 Description de la machine MPC

Le projet **MPC** est l'un des projets les plus importants du département **ASIM** du **LIP**6. Il a pour but la conception d'une machine parallèle performante à faible coût.

3.2 Architecture materielle

La machine **MPC** du **LIP**6 est une machine parallèle de type *grappe de PCs*. Elle est constituée d'un ensemble de PCs standarts interconnectés par un réseau à trés haut débit dont la technologie a été développée par le laboratoire **LIP**6. Elle est composée de 4 noeuds de calcul Bi-Pentium haut de gamme et d'une console pour améliorer l'exploitation de la machine. Le réseau d'interconnexions rapides est un réseau à 1 Gbits/s en full duplex à la norme **IEEE** 1335 **HSL**. Il est composé de liens **HSL** (câbles coaxiaux) et d'une carte **FastHSL** sur chaque noeud. Tous les noeuds sont également équipés d'une carte Ethernet pour constituer un réseau de contrôle non seulement ente eux, mais aussi avec la console.
La carte **FastHSL** a été conçue au laboratoire **LIP**6. Elle contient, en particulier 2 circuits **VLSI** :

- Un controleur de bus **PCI** intelligent appelé **PCI-DDC**
- Un routeur rapide possédant 8 liens **HSL** à 1 Gbits/s appelé ***Rcube***

PCI-DDC réalise le protocole de communication et **RCUBE** le routage des paquets.
Le réseau ainsi constitué fournit une primitive de communication extrêmement efficace d'écriture en mémoire distante qui peut être assimilée à un Remote DMA. L'enjeu est de faire bénéficier les applications de la trés faible latence matériel du réseau, en minimisant le côut des couches logicielles. **MPC** promet donc de trés bonnes performances via son réseau d'interconnexions haut-débit.

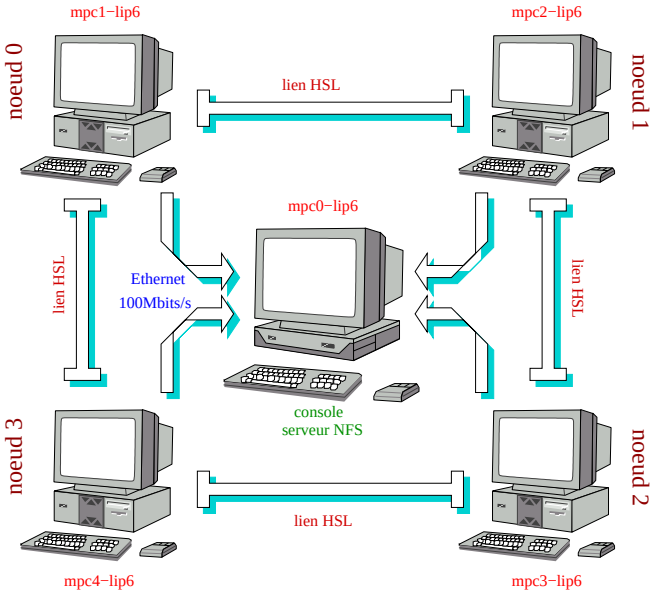

FIG. 3.1 – Architecture de la machine **MPC**

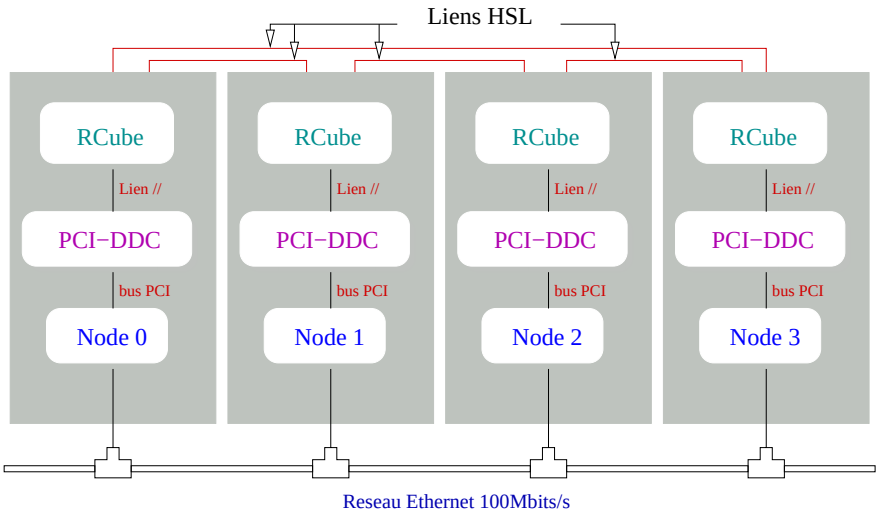

FIG. 3.2 – Quatre cartes **FastHSL**

3.3 Architecture logiciel

3.3.1 L'écriture distante

Principe de l'écriture distante

Le principe général de l'écriture distante ou Remote-write est de permettre à un processus utilisateur du réseau **HSL** et local à un noeud, d'aller écrire dans la mémoire physique d'un noeud distant. Cette opération peut se faire par l'intermédiaire de **PCI-DDC** qui peut accéder directement à la mémoire physique locale. Il prend ses ordres dans une structure de données présentes en mémoire centrale, appelée **LPE** (liste des pages à émettre). Les couches de communication **MPC** permettent aux processus émetteurs d'ajouter dans la **LPE** des descripteurs de type **DMA** qui définisent les paramêtres du transfert à effectuer, et de prévenir le composant **PCI-DDC** de cet ajout. Il se charge alors de ce transfert et signal au processeur émetteur et/ou récepteur la fin du transfert.
Chaque entrée de **LPE** définit une page de données à transmettre. Nous parlons ici d'une page au sens **HSL** : une page est une zone de mémoire physique contigüe dans la mémoire de l'émetteur comme dans la mémoire du récepteur.
Un descripteur de page (entrée de **LPE**) contient les champs suivants :

- Message Identifier (MI), permet d'étiqueter les messages
- Numéro du noeud destinataire
- Adresse physique locale des données à émettre
- Nombre d'octets à transférer
- Adresse physique distante
- Drapeaux

La transmission d'un message se décompose sur trois principales étapes.

- *Préparation :* (1) Une entrée est ajoutée dans la **LPE**. Désormais, le processus émetteur n'intervient plus. (2) **PCI-DDC** est informé dans les couches de communication de la modification de la **LPE**.
- *Transmission :* (3) Le **PCI-DDC** émetteur demande le bus **PCI** et lit le descripteur de **LPE** par accès **DMA**. **PCI-DDC** décompose le message en plusieurs paquets qui contiennent chacun l'adresse physique distante où les paquets doivent être écrits. Quand le dernier paquet est parti, le processeur émetteur peut être prévenu par une interruption (**IT**1) (si le drapeau est positionné).
- *Réception :* (5) Dés que le **PCI-DDC** récepteur commence à recevoir des paquets, il demande le bus **PCI** et écrit les données en mémoire centrale par accès **DMA**. (6) Dans le cas d'un réseau adaptatif, les paquets peuvent arriver dans un ordre quelconque (dépendant de la configuration des tables de routage) ; **PCI-DDC** utilise la **LRM** (liste of received messages) pour compter les paquets reçus. Ce n'est pas le cas de la machine **MPC** actuelle. (7) Une fois que le dernier paquet

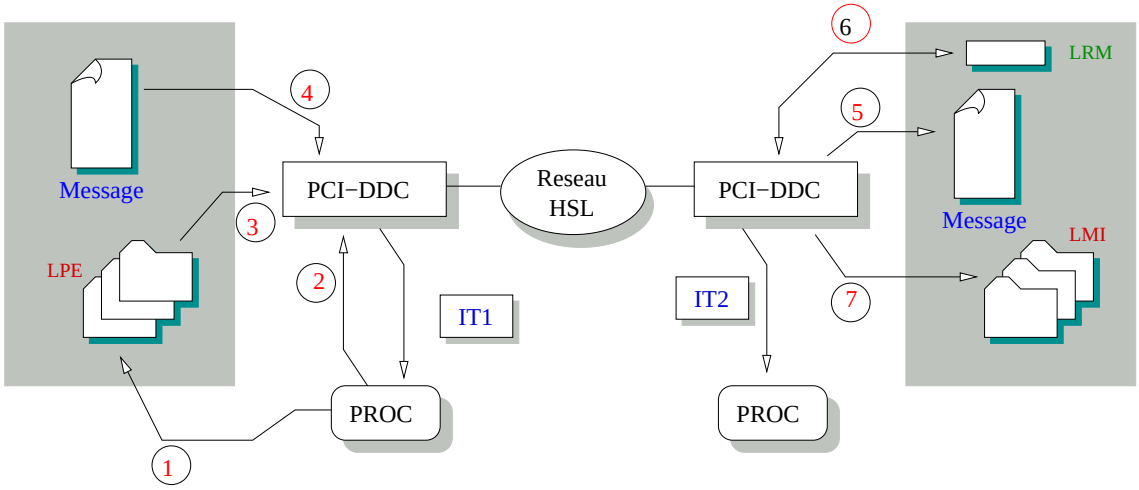

FIG. 3.3 – Les étapes d'une écriture distante

a été reçu, **PCI-DDC** prévient le processeur récepteur soit par un signal d'interruption (**IT**2), soit par une écriture dans la **LMI** (list of message identifiers). Chaque message est étiqueté par un **MI**.
Le message est l'entité au niveau applicatif. Un message étant défini en mémoire virtuelle, il peut être discontinu en mémoire physique s'il est réparti sur plusieurs pages. Il lui correspond alors plusieurs entrées de **LPE**. C'est pourquoi il est nécessaire d'étiqueter le message par un **MI** afin de pouvoir reconstituer le message à la réception ; Chaque descripteur de page contient le **MI**. Toutes ces manipulations (traduction adresses physiques en adresses virtuelles et réciproquement, gestion de **MI**, gestion de la **LPE**, etc...) sont gérées par les couches de communication **MPC**.

Avantage et inconvenient du remote-write

L'avantage principale de l'écriture distante est qu'elle utilise un protocole simple. Mais surtout, ce protocole est *zero-copie*. **PCI-DDC** prend directement les données dans la mémoire de l'émetteur et écrit directement dans la mémoire centrale du récepteur. Cela permet de bénéficier de la très faible latence matériel.
Un autre avantage est l'utilisation d'un lien bidirectionnel. Les communications peuvent se faire dans les deux sens, sans concurrence. Le contrôle de flux se fait au niveau materiel.
En revanche, ce protocole n'est pas classique dans la mesure où le récepteur est actif. Le modèle de communication est le passage de messages sous le mode *Receiver-driven*. L'écriture est asynchrone vis à vis du récepteur. Autrement dit, le transfert du message se déroule sous le contrôle du récepteur.

Ce modèle suppose un rendez-vous préalable entre lémetteur et le récepteur sinon, l'émetteur ne sait pas où il peut écrire dans la mémoire du récepteur. Le rendez-vous permet au récepteur de transmettre à l'émetteur l'adresse physique dans la mémoire du récepteur. De plus, l'émetteur ne prévient pas quand il va écrire. Il écrit directement en mémoire physique, ce qui suppose que les applications utilisateurs ont verrouillées à l'avance des tampons en mémoire physique. Enfin, le récepteur ne connait pas la taille des messages qu'il reçoit.
De ce fait, il est difficile d'adapter des environnements de communication entre **PVM** aux couches basses **MPC** car la phylosophie n'est pas la même. Par exemple, une émission asynchrone au niveau **PVM** peut être pénalisée par le fait qu'une émission suppose d'attendre au préalable un message du récepteur indiquant l'adresse physique dans la mémoire du récepteur.

Chapitre 4

Espace d'adressage d'un processus

4.1 Problèmatique

L'bjectif de ce stage, comme je le rappelerai assez fréquement, est de positionner la mémoire d'un processus dans une zone où l'on sera assuré de la contigüité des adresses physiques. Cela aura pour concéquence un verrouillage automatique des données et nous n'aurons nullement besoin d'une traduction d'adresse que l'on expliquera ultèrieurement. La modification de la mémoire d'un processus implique une connaissance parfaite de son architecture et l'évolution au file du temps.

4.2 Concepts de base

4.2.1 Espace d'adressage d'un processus

A tout processus est associé un espace d'adressage qui représente les zones de la mémoire allouée au processus. Cet espace d'adressage représenté sur la figure 4.1 inclut :

- le code du processus
- les données du processus
  Celles-ci sont dćomposées en deux segments, d'une part *data* qui contient les variables initialisées, et d'autre part *bbs* qui contient les variables non initialisées.
- le code et les données des bibliothèques partagées
- le tas
- la pile

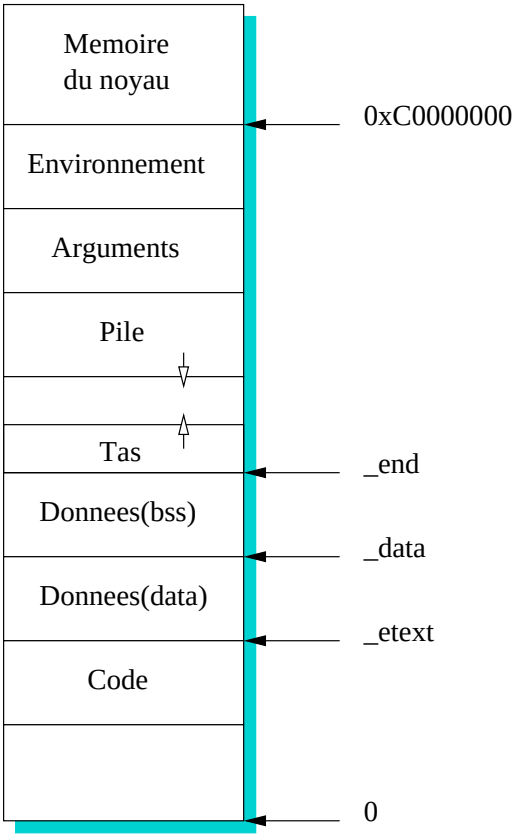

FIG. 4.1 – Espace d'adressage d'un processus

4.2.2 Allocation de mémoire

Lorsqu'un processus commence son exécution, ses segments possèdent une taille fixe. Il existe toutefois des fonctions d'allocation et de désallocation de mémoire, qui permettent à un processus de manipuler des variables dont le nombre ou la taille n'est pas connue au moment de la compilation.

4.3 Appel système de base

4.3.1 Introduction

Commer nous le voyons sur la figure 4.1, l'espace mémoire est composé d'une partie statique (code, données, arguments et environnement) et d'une partie dynamique (tas et pile). Si l'on prend l'exemple du tas, la création d'espace mémoire par la fonction **malloc** augmente la taille du tas et donc la mémoire du processus tout comme sa libération par la commande **free** qui diminuera cette même taille. La *libc* met à disposition plusieurs fonctions permettant de faire évoluer cette taille.

4.3.2 Changement de la taille du segment de données

Un processus peut modifier la taille de son segment de données. A cet effet, Linux fournit l'appel système *brk* :

```
#include <unistd.h>
int brk(void *end_data_segment);
```

le paramêtre *end_data_segment* spécifie l'adresse de fin du segment de données comme nous pouvons le voir dans la figure 4.2.
Une fonction de bibliothèque permet également au processus courant de modifier la taille de son segment de données.

```
#include <unistd.h>
#include <systypes.h>
int *sbrk(ptrdiff_t increment);
```

Le paramètre *increment* spécifie le nombre d'octets à ajouter au segment de données.

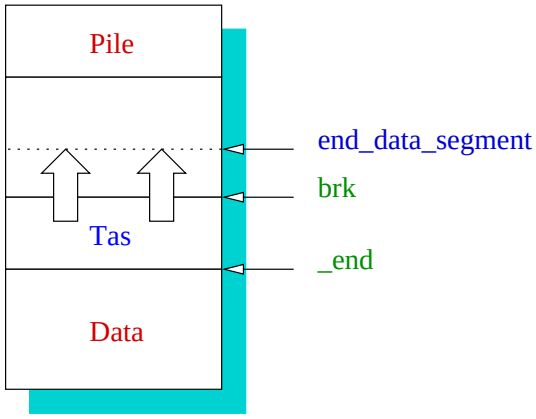

FIG. 4.2 – Evolution de brk

4.3.3 Fonctions de la bibliothèque standard

```
#include <unistd.h>
void *malloc(size_t size) ;
void *calloc(size_t nmemb,size_t size) ;
void *realloc(void *ptr,size_t size) ;
void free(void *ptr) ;
```

Bien qu'il soit possible de gérer dynamiquement la mémoire à l'aide des fonctions **brk** et **sbrk**, il est relativement fastidieux de procéder de la sorte. En effet, si l'allocation de mémoire est aisée, car il suffit d'augmenter la taille du segment de données, la désallocation est plus ardue, car il est nécessaire de tenir le compte des zones mémoire utilisées afin de diminuer la taille du segment de données quand cela est nécessaire. Pour cette raison, on utilise généralement des fonctions d'allocation fournies par la bibliothèque standart.
Ces fonctions utilisent de manière interne **brk** et **sbrk** pour allouer et désallouer des zones mémoire, et elles gèrent la structuration des blocs de mémoire.

4.3.4 Verrouillage de page mémoire

Un processus privilégié peut verrouiller des pages en mémoire. Linux offre plusieurs appels systèmes pour cela :

```
#include <sys.mman.h>
int mlock(const void *addr,size_t len);
int munlock(const void *addr,size_t len);
int mlockall(int flags);
int munlockall(void);
```

mlock permet de verrouiller une zone mémoire.
munlock permet de déverrouiller une zone mémoire.
mlockall permet de verrouiller la totalité de la zone mémoire du processus alors que **munlockall** fait l'inverse.

4.3.5 Projection en mémoire

Linux fournit plusieurs appels système permettant de projeter le contenu de fichiers en mémoire :

```
#include <unistd.h>
#include <sys.mman.h>
void *mmap(void *start,size_t len,int prot,int flags,int fd,off_t offset);
int munmap(void *start,size_t len);
void *mremap(void *old_start,size_l old_len,size_t new_len,unsigned long flags);
```

L'appel système **mmap** projette le contenu d'un fichier en mémoire, dans l'espace d'adressage du processus courant.
La primitive **munmap** supprime quand à elle la projection en mémoire d'un fichier.
Pour finir, **mremap** modifie la taille d'une zone mémoire.

Chapitre 5

Gestion de la mémoire dans Linux

5.1 Structures

La structure de la mémoire se caractérise par la définition de deux structures. Ces structures sont accessibles depuis l'espace kernel, c'est à dire au moyen de requêtes **ioctl**.

5.2 Espace mémoire

5.2.1 Descripteur d'espace d'adressage

Linux maintient un descripteur de l'espace d'adressage. Ce descripteur est accessible par le champ **mm** contenu dans le descripteur du processus. Chaque processus possède normalement un descripteur propre mais deux processus clones peuvent partager le même descripteur si l'option **CLONE_VM** est spécifiée lors de l'appel de la primitive *clone*.

Structure *mm_struct*

La structure *mm_struct* est déclarée dans le fichier d'en-tête <**linux/sched.h**>

```
struct mm_struct {
        struct vm_area_struct * mmap;
        /* list of VMAs */
        struct vm_area_struct * mmap_avl;
        /* tree of VMAs */
        struct vm_area_struct * mmap_cache;
        /* last find_vma result */
        pgd_t * pgd;
```

```
        atomic_t mm_users;
        /* How many users with user space? */
        atomic_t mm_count;
        /* How many references to "struct mm_struct" (users count as 1) */
        int map_count;
        /* number of VMAs */
        struct semaphore mmap_sem;
        spinlock_t page_table_lock;

        struct list_head mmlist;
        /* List of all active mm's */

        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_cnt; /* number of pages to swap on next pass */
        unsigned long swap_address;

        /* Architecture-specific MM context */
        mm_context_t context;
};
```

Explication

Cette structure est le point de départ pour l'analyse de la mémoire du processus. Elle nous donne l'adresse de la table **pgd** que l'on analysera plus tard, ainsi que la première structure d'espace mémoire (analogie au premier module d'une liste chainée). Nous aurons aussi des paramêtres représentant les différents symboles de la pile. Ces symboles nous seront d'une grande utilité lors des modifications de la pile du processus.

5.2.2 Descripteur de régions de mémoire

L'espace d'adressage des processus peut être formé de plusieurs régions mémoire comme nous l'avons expliqué au debut de cet article.
Le noyau maintient en mémoire une description des régions utilisées par un processus. La structure *vm_area_struct*, déclarée dans le fichier d'en-tête <**linux/mm.h**> définit le format du descripteur de chaque région.

kernel 2.2.xx

```
struct vm_area_struct { /* VM area parameters */
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;

        pgprot_t vm_page_prot;
        unsigned short vm_flags;

        /* AVL tree of VM areas per task, sorted by address */
        short vm_avl_height;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;

        /* For areas with inode, the list inode->i_mmap, for shm areas,
         * the list of attaches, otherwise unused.
         */
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        struct vm_operations_struct * vm_ops;
        unsigned long vm_offset;
        struct file * vm_file;
        unsigned long vm_pte;                   /* shared mem */
};
```

kernel 2.4.xx

```
struct vm_area_struct { /* VM area parameters */
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;

        /* linked list of VM areas per task, sorted by address */
        struct vm_area_struct *vm_next;

        pgprot_t vm_page_prot;
        unsigned long vm_flags;

        /* AVL tree of VM areas per task, sorted by address */
```

```
        short vm_avl_height;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;

        /* For areas with an address space and backing store,
         * one of the address_space->i_mmap{,shared} lists,
         * for shm areas, the list of attaches, otherwise unused.
         */
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;

        struct vm_operations_struct * vm_ops;
        unsigned long vm_pgoff;
        /* offset in PAGE_SIZE units, *not* PAGE_CACHE_SIZE */
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;
        /* was vm_pte (shared mem) */
};
```

Explication

La structure **vm_area_struct** nous permet de visualiser la globalité de la mémoire processus, mais du point de vue noyau. Un peu comme le parcourt d'une liste chainée où l'on visualiserait les adresses de départ et d'arrivée ainsi que le contenu, il en sera de même avec cette structure qui , par analogie à la liste, définie un module mémoire. On peut voir cette analogie avec les listes chaînées dans la figure suivante (5.1). L'approche est assez simple puisqu'à partir de la structure du process (**task_struct**), on accède à la structure représentant les caractéristiques de la mémoire (**mm_struct**) et l'on commence par la première structure mémoire (**vm_area_struct**) en sélectionnant **mmap** de **mm_struct**. Chaque zone mémoire caractérisée par la structure **vm_area_struct** permet de visualiser le début (**vm_start**) et la fin des adresses (**vm_end**) de cette zone mémoire et la zone suivante (**vm_next**). Et c'est ainsi que l'on pourra parcourir la globalité de la zone mémoire d'un processus.

Remarque

On remarquera une différence notoire entre les deux structures **vm_area_struct**. Cette différence bloque la compatibilité de **CMEM** que l'on explicitera plus tard, avec le noyau 2.4.x.

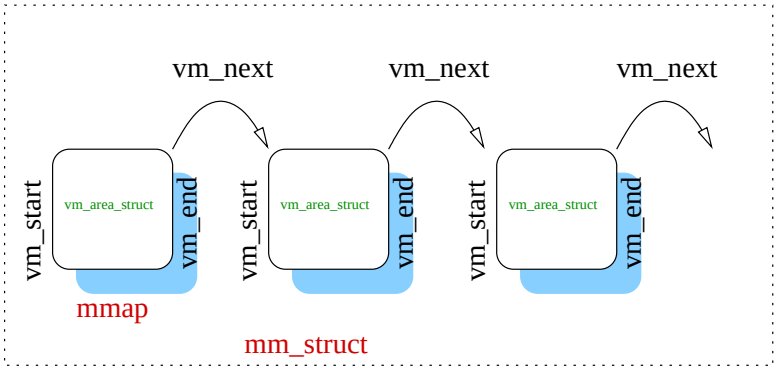

FIG. 5.1 – Structure mémoire

5.3 Virtuelle versus Physique

Très brièvement, l'adresse virtuelle est décomposée en trois champs.
- le champ **Directory** sur 10 bits
- le champ **Table** sur 10 bits
- le champ **Offset** sur 12 bits

La translation est accomplie en deux phases, chacune étant basée sur une translation de table. La première table de translation est appelée **Page Directory** et la seconde **Page Table**. L'adresse physique de la **Page Directory** en cour est stoquée dans le registre **CR3** du processeur. Le champ **Directory** a l'intérieur de l'adresse linéaire détermine l'entrée dans la **Page Directory**. Le champ **Table** de l'adresse virtuelle, additionné au contenu pointé dans la **Page Directory** pointera sur la page physique. Il ne restera plus qu'à se déplacer avec le champ **Offset** de facon à obtenir l'adresse physique.
Par le biais d'un petit calcul , on peut savoir la taille adressée.

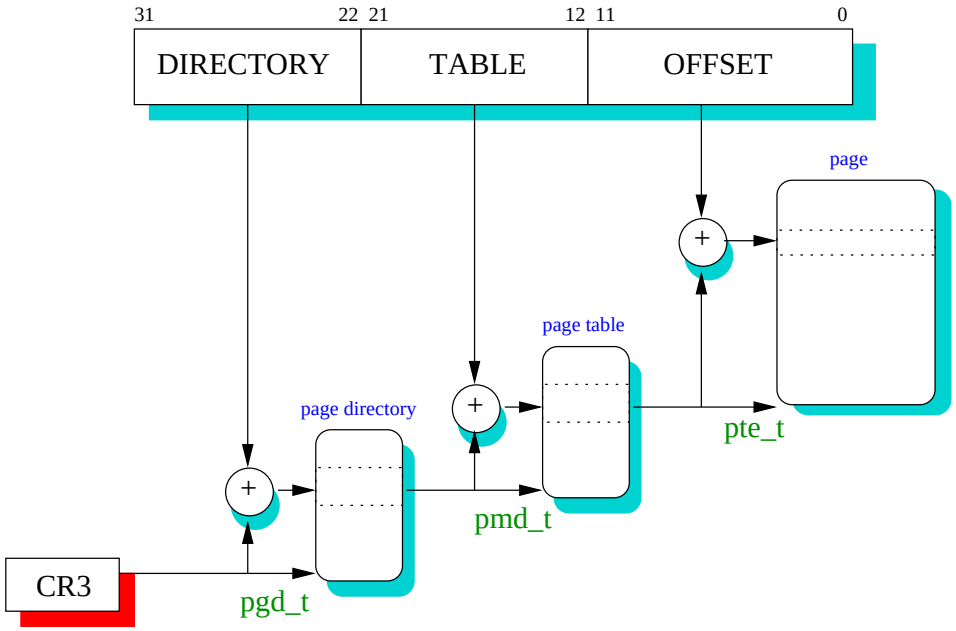

FIG. 5.2 – Conversion virtuel physique

$$2^{10} * 2^{10} * 2^{12} = 4294967296$$
$$= 4Goctets$$

5.3.1 Morceau choisis

```
unsigned long vir2phys(struct mm_struct *mm,unsigned long adresse)
{
pgd_t * fissure_pgd_local;
pmd_t * fissure_pmd_local;
pte_t * fissure_pte_local;
fissure_pgd_local = pgd_offset(mm,adresse);
fissure_pmd_local = pmd_offset(fissure_pgd_local,adresse);
fissure_pte_local = pte_offset(fissure_pmd_local,adresse);
return __pa(pte_page(*fissure_pte_local));
}
```

Ce morceau de code me permet de trouver la page physique dans laquelle se trouve la mm_struct. Un peu comme une histoire à tiroir, on part depuis le **pgd** ou le registre **cr3** que l'on additionne à l'offset **Directory**. On pointe à ce moment sur la page du **pmd** que l'on va aussi additionner à l'offset **Table**.

Nous arrivons enfin sur le début de la page physique. Chaque page physique, contrairement aux pages **pgd** et **pmd** qui contiennent chacune 1024 adresses, contient quand à elle 4096 mots. Il nous suffit plus que d'additionner cette page terminale à l'offset de 12 bits et nous tombons sur notre mot de 4 octets.
Ce code présente 4 fonctions :

- **pgd_offset** décalle l'adresse de 22 pour fournir le **pgd**
- **pmd_offset** fournit l'adresse de la **pmd**
- **pte_offset** fournit l'adresse de la **pte**
- **pte_page** filtre le résultat par un **ET** logique avec **PAGE_MASK**.

Ces différentes fonctions sont situées dans le header *pgtable.h* On peut voir ces différents paramêtres dans la figure 5.2. Un autre morceau de code nous donne la manière de mémoriser le registre **CR3**

```
unsigned long cr3 ;
__asm__ __volatile__("movl (tmpreg) : :"memory") ;
data->cr3= (unsigned long)tmpreg ;
```

Chapitre 6

Problèmatique

6.1 Linux vs FreeBsd

On peut représenter l'existant sous la de forme de couche
Description des différentes couches de la figure 6.1 :

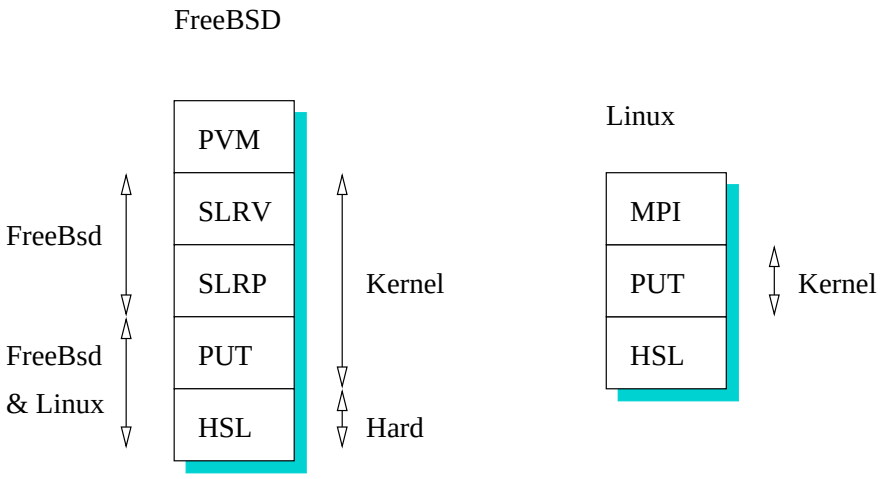

FIG. 6.1 – Architecture logicielle

- La couche **PUT** constitue le service de communication de plus bas niveau et fournit la fonction *remote-write* de l'espace d'adressage physique local vers l'espace physique d'un noeud distant.
- La couche **SLRP** fournit un service d'échange de zone de mémoire physique entre noeuds, sur des canaux virtuels
- La couche **SLRV** fournit un service équivalent mais en manipulant des zones localisées dans des espaces de mémoire virtuelle.
- **CMEM** et **HSL** sont les deux drivers **MPC** qui permettent d'accéder aux fonctionnalités de la carte **FastHSL**

6.1.1 Latence

PUT	$4\mu s$
MPI	$26\mu s$
PVM	$200\mu s$
SLRV	$75\mu s$
SLRP	$45\mu s$

On peut noter que le calcul de la latence s'effectue par un simple **ping-pong**. On envoie 1000 fois un élément binaire. On divise le tout par 2 (aller-retour). On calcule enfin la moyenne.

6.2 MPI_Send

Nous avons deux type de messages.

- message de contrôle
- message de donnée

Si la taille des données ne dépasse pas 128 octets , elles seront considérées comme message de contrôle. L'envoi des messages est précédé d'un échange de messages de contrôle pour indiquer où doit être fait le transfert.

6.3 Problème de continuité

Le problème majeur dans le transfert d'une donnée est que celui-ci ne se fait pas en une seule fois. La fonction **PUT** envoie des données au niveau physique, mais pour l'utilisateur, chaque donnée est représentée par son adresse virtuelle.
Sur la figure 6.2, on crée un tableau de caractères contigües dans l'espace

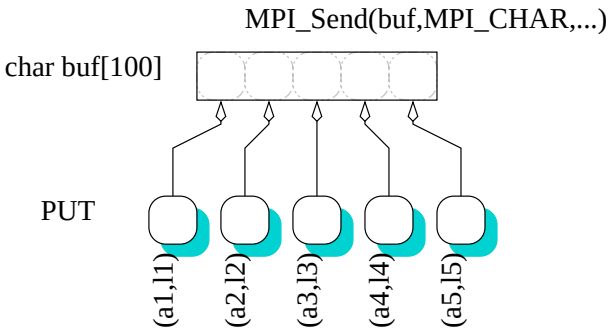

FIG. 6.2 – virtuel vs physique

d'adressage virtuel du processus (les adresses se suivent). Il en est tout autre sur la partie physique puisque les données sont positionnées un peu n'importe où. De ce fait, on effectuera autant de **PUT** qu'il y aura de fragments.

Voila ce qui peut se passer entre deux machines.

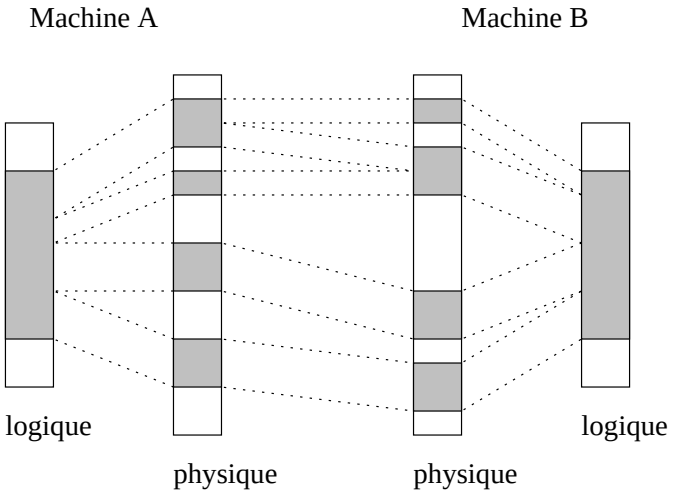

FIG. 6.3 – transmission discontigüe

6.4 Résultat

Comme on vient de le remarquer, nous avons un gros problème de verrouillage, c'est à dire que nos données processus ne sont pas forcément en mémoire, mais aussi que chaque fragment de données nécessite de connaître leurs adresses physiques. On rappelera que la fonction **PUT** ne travaille qu'au niveau physique et que chaque traduction d'adresse se caractérise par un appel système, ce qui est beaucoup trop couteux. D'où l'intêret de diminuer ses appels systèmes en connaissant d'avance ces adresses physiques, mais aussi en étant sùr que ces données sont déjà en mémoire. Ce sera le cas si l'on repositionne notre mémoire processus dans une zone gérée par **CMEM** , c'est à dire une zone de mémoire physique contigüe.

6.5 Problème de temps

Un autre problème vient se greffer dans la continuité des données contigües. Chaque transfert(appel à **PUT**) fait appel à un appel système. Malheureusement, ces appels sont couteux en temps, d'ou l'interêt de diminuer le nombre de **PUT**. En se référant à la figure 6.2, on remarque que le nombre de **PUT** est au nombre de 4. L'objectif sera de passer à un seul **PUT**.

Chapitre 7

Approche MPI

7.1 Introduction

Ce chapitre technique présente la méthode de recueil d'information lors de l'initialisation d'un exécutable **MPI**, Il permettra surtout de connaître la taille mémoire attribuée au tas et à la pile en fonction du nombre de processus.

7.2 Objectif

Nous devons maintenant tenir compte de l'architecture de la machine **MPC**. C'est a dire du nombre de Nodes (noeuds), des processus sur chaque machine, et bien sur de l'attribution de la memoire **CMEM** pour chacun d'eux.

7.3 Parametres

- NNODES Nombre de nodes dans la machine MPC. Ce paramètre peut être accessible depuis le paramétre de configuration *mpid/ch_mpc/mpc.conf*. On verifiera que les sources *mpihsl.c* et *mpidriver.h* ne fournissent pas ce paramêtre.
- CMEM_FREE Ce paramètre nous donne la taille (en octet) de mémoire **CMEM** libre. On vérifiera dans *cmem.c* s'il n'existe pas de fonction permettant d'obtenir cette valeur. On peut bien sur visualiser cette donnée ainsi que d'autres en effectuant un *cat /dev/cmem*.
- NPROCESS Ce paramètre définit le nombre de processus à lancer pour l'application **MPI**. Sur une application **MPI**, le lancement s'effectue de la manière suivante.
  mpirun -np NPROCESS exécutable

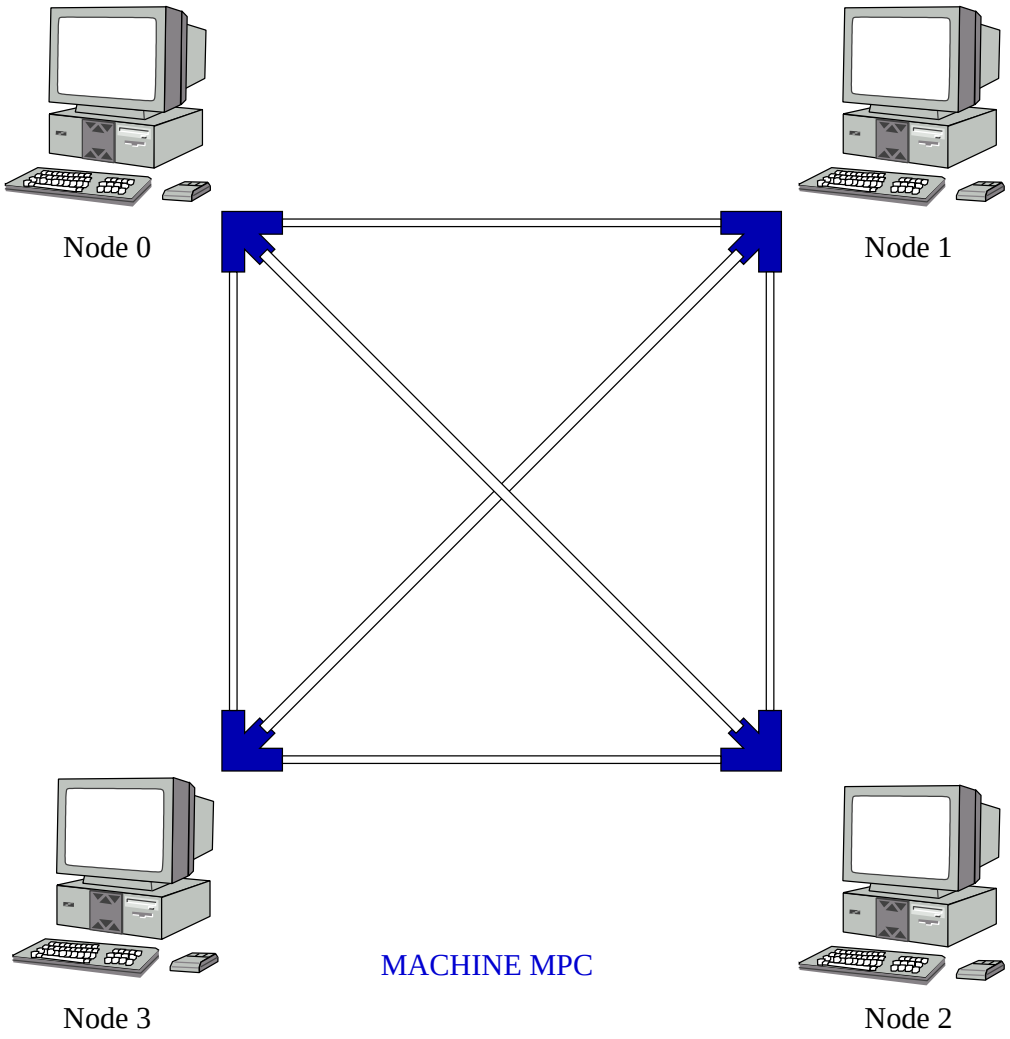

FIG. 7.1 – Machine MPC

Actuellement

$$NPROCESS \leq NNODES$$

- NPROCESS_PER_NODE Ce paramètre définit le nombre de processus par node. On peut l'obtenir par

$$\frac{NPROCESS}{NNODES}$$

  Actuellement, nous avons comme valeur 0 ou 1.
- CMEM_PER_PROCESS Paramètre définissant la mémoire à attribuer à chaque processus. On obtient cette valeur par

$$\frac{CMEM\_FREE}{NPROCESS\_PER\_NODE}$$

  Afin d'uniformiser cette attribution de mémoire sur chaque node, on préfèrera plutôt

$$\frac{CMEM\_FREE}{max(NPROCESS\_PER\_NODE)}$$

  ou max(NPROCESS_PER_NODE) représente le nombre maximum de processus, tout node confondu.
- HS_PER_PROCESS Exprimé en pourcentage, cette valeur positionnera le tas entre 0 et 100.

$$SS\_PER\_PROCESS = 100 - HS\_PER\_PROCESS$$

7.4 Définition de la configuration

L'objectif premier est de lire et extirper les informations issues du fichier *procgroup_file.mpc* ainsi que celles de *mpc.conf*. Cette fonction, initialement application sera intégrée dans la bibilothèque dynamique et sera bien sur exécutée par chaque processus. La forme de cette fonction sera la suivante :
mpi_task_info read_procgroup()
Cette fonction fournit une structure *mpi_task_info* comportant

- NNODES fournie par *int mpi_get_read_nbnodes()* dans *mpihsl.c*
- NPROCESS_PER_NODE
- CMEM_FREE
- CMEM_PER_PROCESS
- MyWorldSize comptabilisant le nombre global de processus
- MyWorldRank définissant le numéro de la tâche **MPI**
- Node étant le numéro du noeud HSL, il est fournie par *int mpi_get_node()*

- HS_PER_PROCESS
- local_desc MyMPIWorld[NNODES], tableau de structure *local_desc* avec struct *local_desc* int nodes ;int tasks[MAX_NROCESS_PER_NODE] cela permet de connaître les differents processus en local avec leur propre *MyWorldRank*.

Exemple de fichiers *procgroup_file.mpc* et *mpc.conf*.
procgroup_file.mpc

mpc1-lip6	2	50	/tmp
mpc2-lip6	1	48	/tmp
mpc3-lip6	4	20	/tmp
mpc1-lip6	1	50	/tmp

Chaque ligne correspond au nom de la machine, le nombre de process, le pourcentage accordé au tas et le répertoire d'accueil de l'exécutable.
mpc.conf

mpc1-lip6	10
mpc3-lip6	20
mpc4-lip6	30
mpc2-lip6	40

Chaque ligne correspond au nom de la machine ainsi qu'à son numéro d'identification.

7.5 Algorithme d'attribution de *MyWorldRank*

La méthode d'attribution des numéros de tâche **MPI** se fait de la manière suivante. On attribue une seule tâche **MPI** à chaque noeud en suivant l'ordre croissant de leur numéro. et l'on recommence dés que la globalité des noeuds comportants des process a été passée en revue. On décrémente leurs nombre de processus et ainsi de suite jusqu'à ce nombre de processus soit nul. Nous pouvons nous appuyer sur la figure 7.2 pour plus de détails.

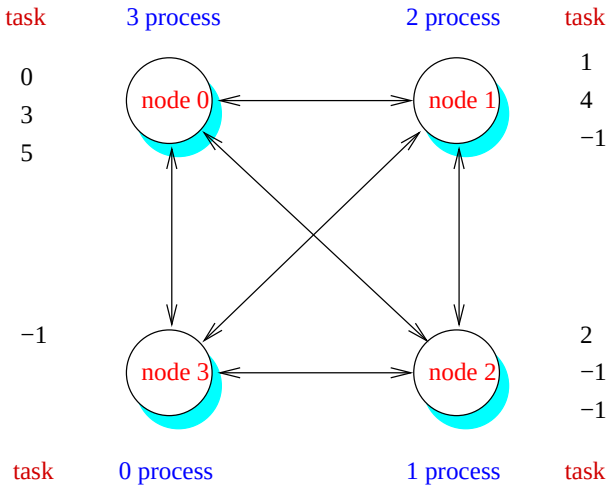

FIG. 7.2 – Machine MPC

Chapitre 8

Evolution

8.1 Objectif

Ce chapitre décrit les différentes phases de progression dans le stage. La première phase, bien sùr est de s'acclimater avec les différentes ressources proposées tel que **CMEM** .

8.2 Les différentes étapes

Avant de rentrer dans les détails, j'ai du effectuer une modification sur le header **cmem.h** en ajoutant les lignes suivantes :

```
extern unsigned cmem_nslots ;
extern size_t cmem_size ;
extern u_long cmem_phys_start ;
extern void * virt_start ;
extern cmem_slot_t *slots ;
```

Les différentes étapes sont clairement définies :

- récupérer la taille proposée de la pile et du tas(len1 et len2)
- réserver ces deux tailles dans **CMEM** en utilisant deux slots.
- modifier la taille du tas pour s'ajuster avec la taille du slot **CMEM** réservé
- effectuer une sauvegarde sur deux fichiers différents de la pile ainsi que l'ensemble code, data, tas modifié.
- remapper le slot **CMEM** comportant le tas ainsi que les datas et codes à l'emlacement commençant à l'adresse $0x8048000$ de même que le slot de la pile.
- on finit par la recopie du fichier comportant le tas , le code et les datas à partir de l'adresse $0x8048000$ ainsi que le fichier de la pile à l'adresse du **mmap**. En regardant la figure 8.1, on part d'un état où la mémoire du processus est morcelée, vers un ensemble plus compact regroupant

deux slots, l'un pour l'ensemble code, data et tas, et l'autre pour la pile.

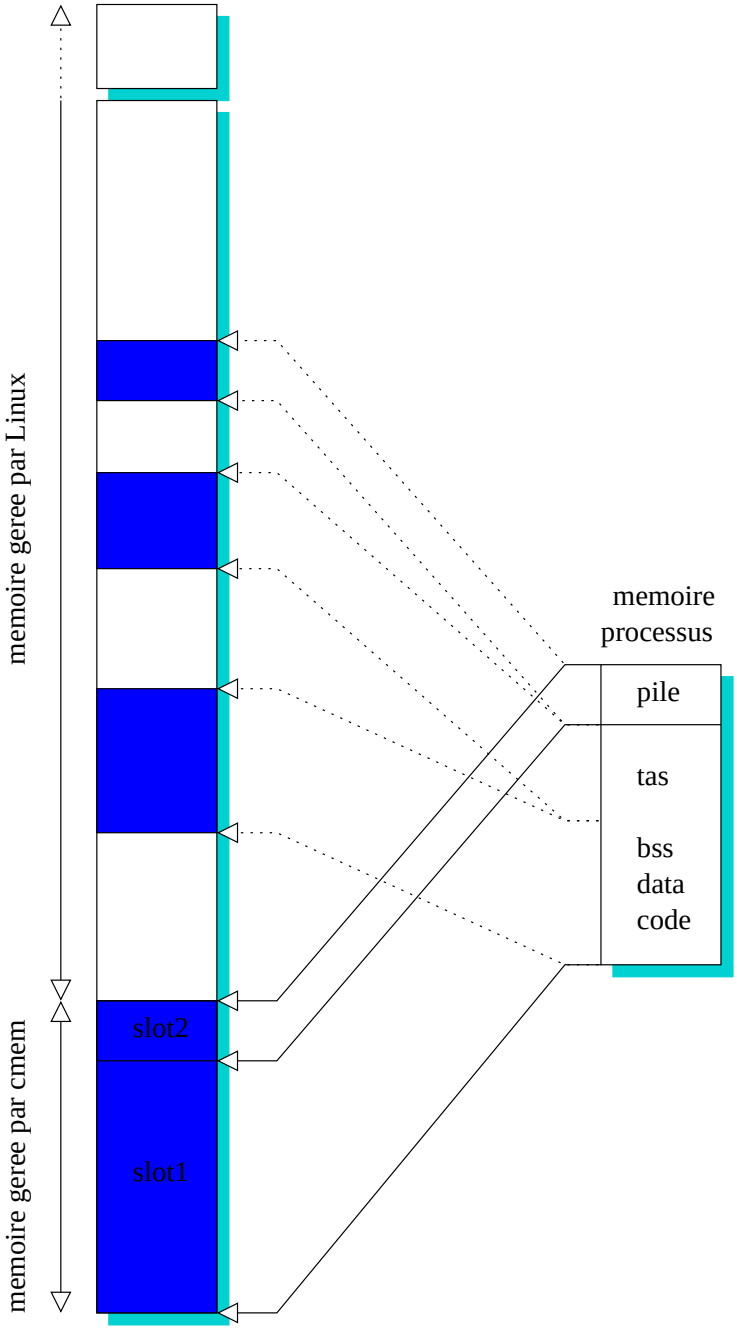

FIG. 8.1 – Objectif

Cette figure (8.2) permet de voir plus précisement le résultat des différentes phases.
Les deux étapes sont présentes dans cette figure 8.2 :

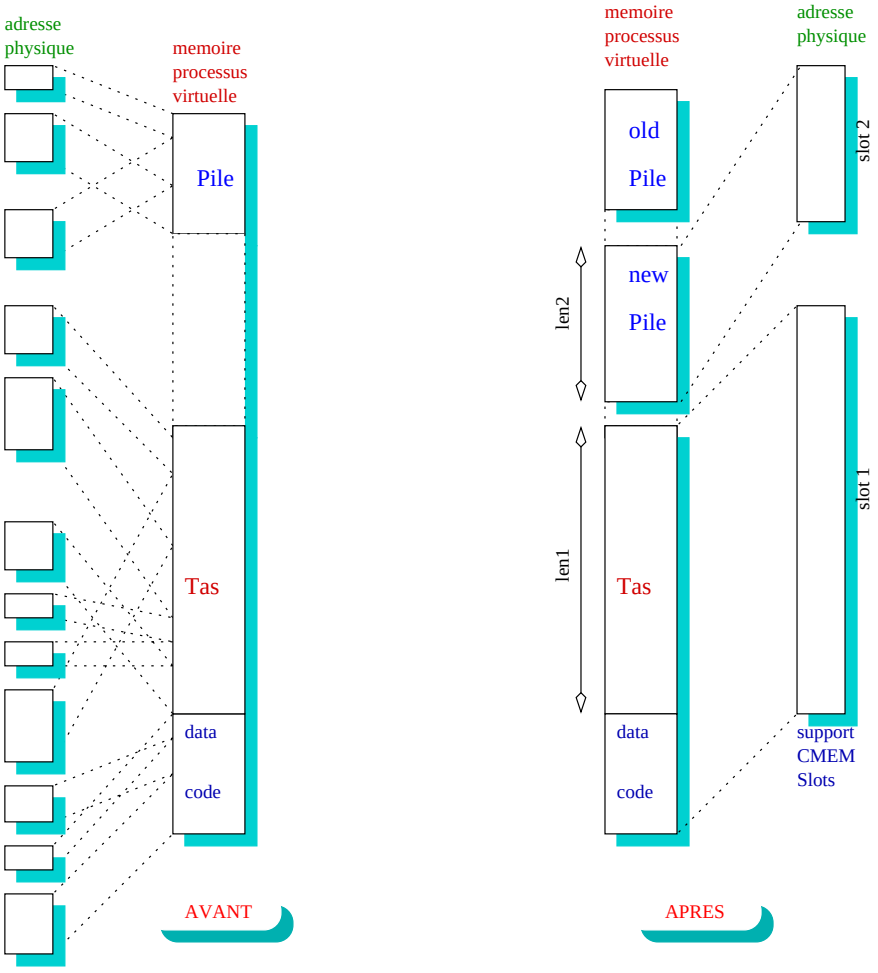

FIG. 8.2 – Morcellement

- Avant
  On redéfinit la taille du tas en fonction de la valeur fixée par l'utilisateur en effectuant plusieurs **malloc** concécutifs.(voir chapitre 8.3)
- Après
  L'ancienne pile n'est plus valide. la nouvelle pile contient l'environnement et les données relatives au **main**. La taille quand à elle aura aussi été fixée par l'utilisateur (voir chapitre 8.4). On se trouve avant le **JMP** dans la bibliothèque dynamique.

8.3 Problème du tas

8.3.1 Problématique

Le problème de cette allocation réside dans le fait que tout s'effectue en dynamique. Jusqu'à présent, nous avons pu réserve une taille fixe de la mémoire, soit de l'adresse $0x08048000$ jusqu'au symbole &_end. Le tas, comme nous l'avons vu dans l'introduction, évolue en fonction des **malloc** générés tout au long du processus. De grosses différences de fonctionnements sont à noter entre **malloc()**, **brk()** et **sbrk()**. Le cumul des malloc s'effectue par bloc de 4096 bytes(voir figure 8.3). Dés que l'on réserve quelques octets, le système vérifie que l'on ne dépasse pas la valeur maximum du bloc, et c'est ainsi que l'on pourra de nouveau piocher dans cette ressource jusqu'à la valeur maximum. Si celle ci est dépassé pour un nouveau malloc, le système attribuera un ou plusieurs blocs de 4096 bytes pour le processus jusqu'à la hauteur du pointeur de pile - $16K$ bytes puisque cette valeur définie par le système oblige un écart de $16K$ entre l'adresse de **brk** et l'adresse de la pile. Les fonctions **brk()** et **sbrk()** , quand à elles, fonctionnent non pas par bloc mais par octet. **brk()** prendra comme paramètre la fin du segment de la mémoire allouée, alors que **sbrk()** prendra la valeur d'incrémentation.
On notera qu'au départ, le symbole **brk** est généralement identique au symbole _end (fonction de l'alignement), et à la première allocation, il s'incrémentera d'un modulo $4K$ avec le complément de l'adresse du symbole _end jusqu'à l'obtention de la valeur ronde. Finalement, il aura une adresse virtuelle ronde du type $08049a000$ mais sa page sera inaccessible.

8.3.2 Solution

La reservation de la mémoire pour le tas s'effectuera par l'intermédiaire du **malloc()**. Nous n'effectuerons pas un seul malloc , mais plusieurs de même capacité du type

$$buffer = (char*)malloc(0x1FFF4 * sizeof(char)); \tag{8.1}$$

$$0x1FFF4 = 131060; \tag{8.2}$$

Le reste sera complémenté par un nouveau **malloc()**. Nous finirons tout cette ensemble de **malloc()** par un **malloc()** frontière comme on peut le voir sur la figure 8.4. Il délimitera cette zone de mémoire attribuée au tas. Ces opérations étant faites, il nous suffira de libérer ces zones de mémoire tout en concervant la frontière , cette opération s'effectuera par la commande
.

$$free(buffer); \tag{8.3}$$

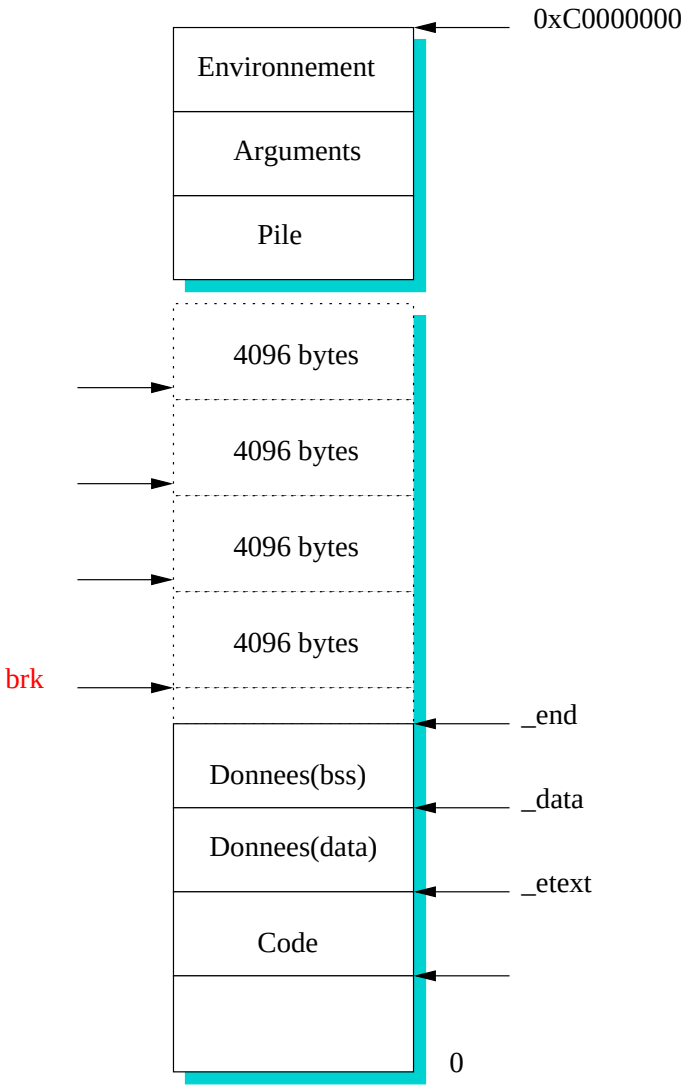

FIG. 8.3 – Evolution de brk

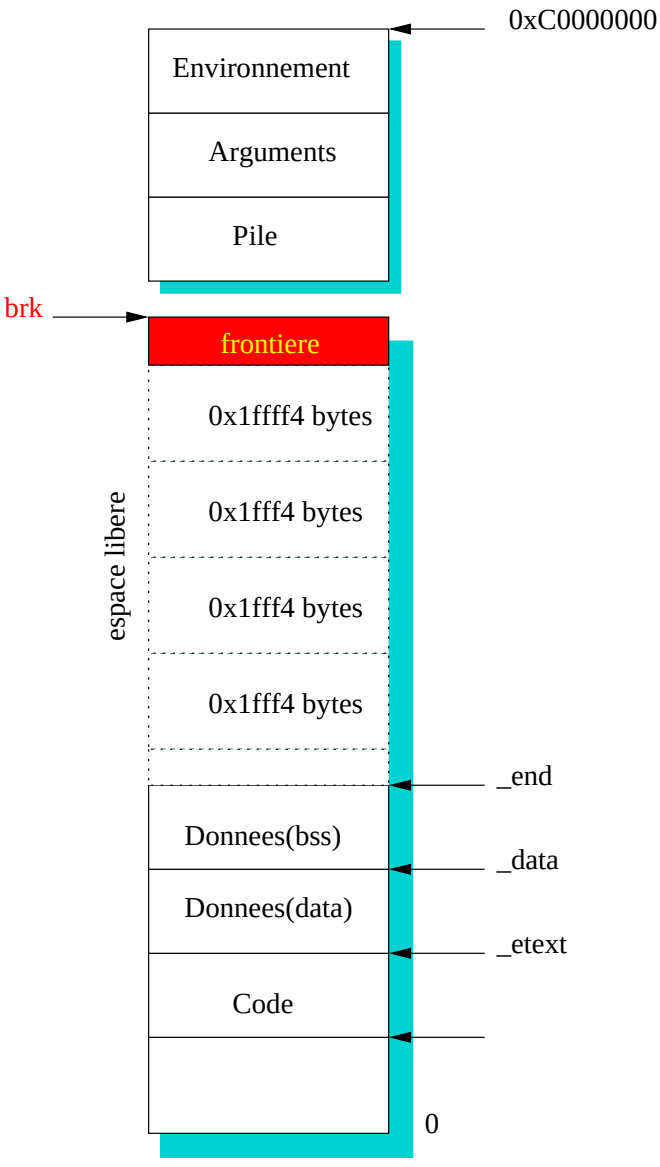

FIG. 8.4 – Evolution de brk

8.3.3 Complément

Comme nous le disions au début de ce document, les **malloc** et autres commandes savent gérer les zones de mémoires libres et occupées. C'est à dire que si l'on effectue un **malloc** concéquent, le pointeur **brk** augmentera, mais il reprendra sa place si l'on libère cette zone mémoire. Le tout est de jouer sur une imperfection de celui ci. **malloc** malgrés toutes ses qualités ne sait pas gérer les trous, c'est à dire les zones libres comprises entre des zones utilisées. Et c'est pourquoi nous effectuons plusieurs **malloc** à hauteur de la place disponible et l'on finit par un petit **malloc** qui lui servira de délimiteur. Si l'on libère les autres **malloc** en dessous de celui ci, le pointeur **brk** ne pourra redescendre puisqu'il reste une zone occupée (on tâchera bien sùr de ne pas effacer cette petite zone). A partir de là, il ne nous restera plus qu'à réserver une taille mémoire de la hauteur de tous ces **malloc** dans **CMEM** pour s'assurer qu'ensuite, le tas sera verrouillé et que nous n'auront plus de problème de traduction d'adresse.

8.3.4 Remarque

Si l'on devait dépasser la taille requise pour le tas, **Linux** effectuera ses **malloc** dans le reste de la mémoire et nous perdrions bien sùr les propriétés de **CMEM** en retombant dans un contexte normal d'affectation de mémoire.

8.4 Gestion de la pile

Le problème du tas n'est malheureusement pas applicable sur la pile puisque cette pile représente un instant de l'évolution du processus. Si nous écrasions cette dîte pile, le processus perdrait ses marques et ceci résulterait d'un *segment fault*. L'objectif sera de réserver un espace (un slot **CMEM**) pour la future pile en respectant bien la taille de celle ci par rapport au paramètres lus dans les fichiers de configuration. Arrivée au terme de l'exécution de la bibilothèque dynamique, on effectuera un saut (**CALL**) sur le symbole **main**. Ce saut est necessaire pour shunter le **RET** en fin de la bibliothèque dynamique puisque ce retour nous ramenerait sur l'ancienne pile. Plusieurs solutions s'offraient à nous, mais la création global de la pile pour la fonction **main** paraît la plus simple.

8.4.1 Problèmatique

Toute l'astuce consiste comme nous le disions prcédement, à effectuer un **CALL** sur le symbole **main**. L'avantage de cette primitive est qu'elle mémorise l'adresse de l'instruction suivante avant d'effectuer son saut qui n'est ni plus ni moins un **JMP**. Malheureusement, nous perdons la globalité des paramètres liées au processus, c'est à dire :

- int *argc*
- char ***argv*
- Toutes les variable d'environnement

La solution sera d'effectuer plusieurs opérations au préalable.

- sauvegarde de la pile depuis le symbole **start_stack** jusqu'à l'adresse $0xC0000000$ qui délimite la mémoire utilisateur de la mémoire kernel.
- réserver un slot **CMEM** pour accueillir cette nouvelle pile. Cette zone sera au minimum égale à la taille de la pile mémorisée.
- mapper le fichier de sauvegarde dans le haut de ce nouveau slot **CMEM** réservé préalablement.
- modifier dans cette nouvelle zone les adresses des arguments **argv[x]**.
- positionner le pointeur **%esp**
- effectuer les **PUSH** successifs de manière à positionner les différents arguments avant le **CALL**
- enfin, effectuer le **CALL**

On note bien dans la figure 8.5 que la pile sauvegardée depuis le symbole **start_stack** jusqu'à $0xC0000000$, est translatée sur une zone définie au préalable par un **mmap**.

8.4.2 Architecture de la pile

La pile est caractérisée par plusieurs symboles délimitant chaque zone. Mais cette caractéristique se situe essentiellement en mode kernel. La partie

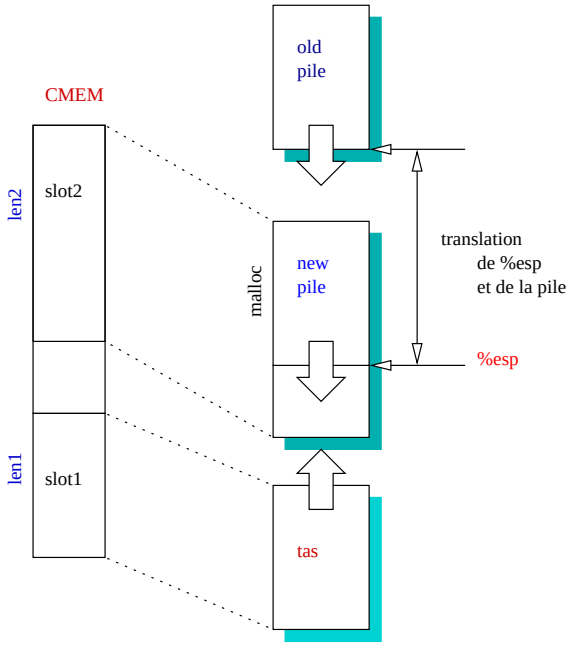

FIG. 8.5 – Creation de la pile

sauvegardée est représentée sur la figure 8.6. Elle se situe donc entre le symbole **start_stack** et l'adresse $0xC0000000$.

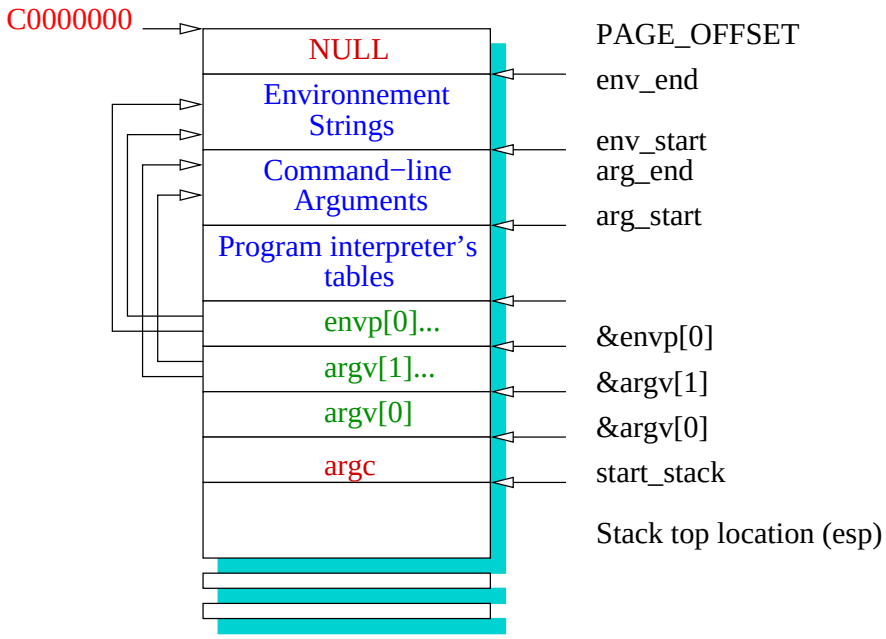

FIG. 8.6 – Structure du haut de la pile

8.4.3 Appel d'une fonction

Cette méthode s'appliquera aussi bien sur une simple fonction que sur la fonction main :

- on effectue un **PUSH** des paramètres dans le sens inverse de leur arrivée
- on effectue un **CALL** sur le symbole main

Pour notre exemple, les paramètres sont composés de pointeurs. A nous de modifier ainsi ces différentes adresses. Le tout est de connaître la translation de cette nouvelle pile positionnée à partir d'une adresse positionnée par le malloc que notre bibliothèque aura effectué.
Puisque la méthode **Procmem ; ;Assembleur** est composée de uniquement d'instructions en assembleur, il est impératif de cummuler la fin de cette méthode par la suite :

```
movl %ebp,%esp
pop %ebp
ret
```

Ce bout de code permet de retrouver l'ancien contexte, c'est à dire l'ancienne pile. Pour terminer, on effectuera un **exit(0)** sur la class **Dynamique** à l'origine de la création de de la méthode comportant les primitives assembleur (**Procmem.cc**). La figure 8.7 représente la pile aprés le **main**

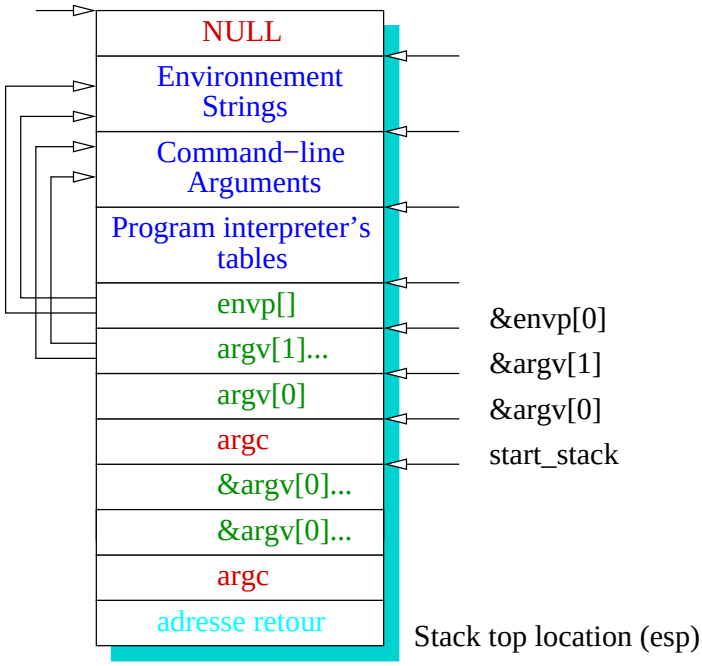

FIG. 8.7 – Structure de la pile à l'entrée du main

Complément à deux

Le calcul de ces nouvelles adresses pose le problème du débordement. Le calcul directe de $bfffff620 - 401ee00b$ provoque une erreur de débordement. L'astuce consiste à effectuer l'algorithme de la figure 8.8

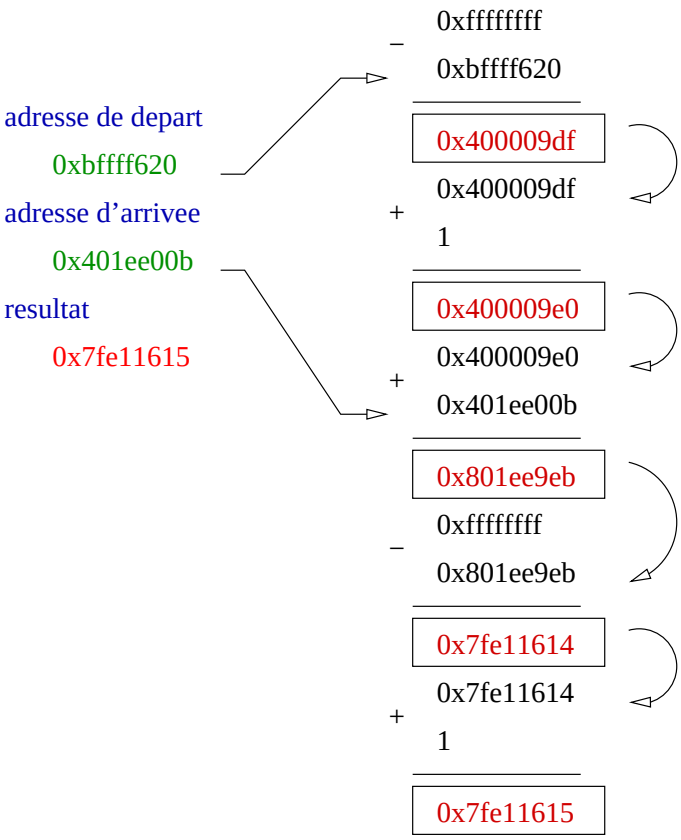

FIG. 8.8 – Soustraction

A titre indicatif, voici le programme en assembleur permettant ce petit calcul de soustraction :

```
movl $0xffffffff,%ecx
sub %ebx,%ecx
add $0x1,%ecx
add %eax,%ecx
movl $0xffffffff,%ebx
sub %ecx,%ebx
add $0x1,%ebx
push %ebx
```

8.4.4 Assembleur

La méthode la plus propre consiste à insérer un morceau de code en assembleur dans la bibliothèque dynamique. Les paramètres de passages seront l'adresse nouvelle du bas de la pile ainsi que la valeur de la translation de l'ancienne pile à la nouvelle. Je rappelle que la nouvelle pile prendra place dans une partie où l'on aura effectué un **malloc** et remapé un slot de **CMEM** . Comme au début du tas, on effectue une sauvegarde de la pile depuis la position **start_stack** jusqu'à l'adresse $0xC0000000$. Ensuite on repositionne les différentes adresses des arguments en fonction du nouvelle emplacement. A titre toujour indicatif, ce petit morceau de programme permet de modifier la globalité des adresses des arguments en fonction du nombre d'argument ici considéré comme compteur.

```
push (%ebp)
movl %ebp,%ecx
BOUCLE :
pop %edx
dec %edx
push %edx
add $0x4,%ecx
push (%ecx)
movl $0xffffffff,%eax
sub (%esp),%eax
add $0x1,%eax
add 0x8(%esp),%eax
movl $0xffffffff,%edx
sub %eax,%edx
add $0x1,%edx
movl %edx,(%ecx)
add $0x4,%esp
cmp $0x0,(%esp)
jnz BOUCLE
```

Le programme suit scrupuleusement l'algorithme de la figure 8.8

8.5 Conclusion

A partir de ce moment, toutes les données relatives au processus, et plus exactement à l'exécutable situé sous le **main**, sont positionnées dans les deux slots **CMEM** . Nous pouvons nous assurer en utilisant le programme de conversion virtuelle-physique vu précédement que nos adresses virtuelles sont identiques aux adresses physiques à un offset près. C'est à dire qu'en connaissant cet offset, nous pouvons deviner l'adresse physique des données. Bien sùr, cet offset sera différent entre les deux slots, mais sur chaque slot trouver l'adresse physique devient aisé pour la fonction **PUT** puisque celle ci fonctionne sur la couche physique, mais on peut aissi dire que nos données sont verrouillées en mémoire, c'es à dire qu'elles sont présentes et qu'il est donc inutile d'effectuer un **lock** ou un **lockall**.

adresse physique = adresse virtuelle + offset

Ce petit calcul nous amène donc à conclure qu'il n'y a plus de traduction d'adresse à effectuer.

Chapitre 9

Conclusion

9.1 Rappel des objectifs

L'objectif de ce stage était de recopier intégralement la mémoire du processus dans une zone de données contigüe. Cela permettait de verrouiller systématiquement ces données et aussi, évitait une traduction d'adresses, augmentant bien sur le coût. J'ai pu, progressivement atteindre cet objectif durant mon stage. La dernière semaine fut entièrement consacrée à l'élaboration de mon rapport ainsi qu'à la documentation technique pour les futurs utilisateurs de ce logiciel.

9.2 Déroulement du stage

Les premières semaines m'ont permis de prendre contact avec le milieu universitaire. J'ai pu découvrir ce qu'est un laboratoire de recherche surtout que le **LIP6** est le laboratoire de recherche le plus grand de France.
La deuxième étape a été de me familiariser avec toutes les subtilités d'installation qu'apporte **Linux** dans la gestion des périphériques.
La troisième étape et pas la moindre a été de m'investir fortement dans la programmation noyau. Cela m'a incité à utiliser énormement de documentation en englais et effectuer pas mal de recherches sur le net.
La dernière étape a été de comprendre rapidement les rouages de la programmation machine sous **Linux** (l'assembleur) sur les parties systèmes.
Mon stage s'est terminé sur la période de test et la mise au point.

9.3 Apports du stage

Ce stage m'a été bénéfique à de nombreux points de vue. Outre la prise de connaissance du monde de la recherche, il m'a permis de conduire un projet du début jusqu'à la fin.

D'un point de vue technique, ce stage m'a permis d'élargir mes connaissances du langage C mais aussi du monde **UNIX**. J'ai pu découvrir les faces cachées de **Linux** , la programmation noyau, les finesses de la mémoire d'un processus.

9.4 Le projet MPC

Le département **ASIM** du **LIP6** est fortement impliqué, depuis 1993 dans le développement d'une technololgie d'interconnexion haute performance qui est devenue le standart **IEEE** 1355. Le département **ASIM** a été sollicité par différents industriels (Bull, SGS-Thomson, Parsytec..) pour participer à quatre projets europeens visant le développement ou l'exploitation de cette technologie **HSL**. Les composants **VLSI**, **RCUBE** et **PCI-DDC**, qui ont été entièrement conçus à l'université Pierre et Marie Curie, sont actuellement utilisés dans plusieurs machines industrielles.
Le projet **MPC** qui a démarré en janvier 1995 sous la responsabilité de Alain GREINER, possède un fort impact puisqu'il existe des plates formes **MPC** à Versailles (**PRISM**), Evry (**INT**), Toulouse, Amiens, etc...
Les cartes **FastHSL** et les composants **VLSI**, **RCUBE** et **PCI-DDC** sont commercialisés par la société **Tachys Technologie**, qui est une start-up du département **ASIM**

Chapitre 10

Annexe A

10.1 Utilisation du package CMEM

10.2 Compilation du noyau

10.2.1 **Kernel** 2.2.16

Nous partons de ce noyau issue de la distribution **REDHAT** .

10.2.2 **Kernel** 2.2.17

Les premiers problèmes surviennent sur la carte réseau **3c**5x9. Il suffit pour cela de télécharger le driver **3c**90x sur le site

http ://cesdis.gsfc.nasa.gov/linux/drivers/vortex.html

La suite consiste ensuite à charger en module ce driver, c'est à dire à modifier le fichier de configuration **/etc/modules.conf** et lui ajouter

alias eth0 3c90x.0

Il est bien sùr necessaire de copier ce fichier compilé dans le répertoire

/lib/modules/2.2.17/net

10.2.3 **Kernel** 2.4.0

Un autre problème survient dans ce nouveau kernel. La carte **3c**90x n'est bien sùr pas reconnue, mais on a une modification notoire du répertoire concernant la disposition des modules.
Sur les versions antérieurs, l'arborescence des modules était constituée de la manière suivante

/lib/modules/2.2.xx/

Sur les versions 2.4.x, l'arborescence des modules est constituée de cette manière

/lib/modules/2.4.xx/kernel

10.2.4 **Kernel** 2.4.1

10.3 Compilation de CMEM

10.3.1 Problème et résolution de CMEM

Lors de la compilation, plusieurs problèmes liés aux inclusions vont surgir.

- **linux/modversions.h**
  La solution consiste premièrement à sauvegarder le répertoire **/usr/include/linux** sous **/usr/include/linux.ori** et le remplacer par un lien symbolique en effectuant la commande

  ln -sf /usr/src/linux/include/linux /usr/include/linux

  On tiendra à vérifier que le lien symbolique **/usr/src/linux** existe et se positionne sur le répertoire du kernel utilisé. Si tel n'était pas le cas, il suffirait d'effectuer la commande suivante

  ln -sf /usr/src/linux-2.2.*xx /usr/src/linux*

  Deuxièment, on effectue la même opération sur le répertoire **/usr/include/asm** que l'on renomme **/usr/include/asm.ori**. Il sera remplacé par le lien symbolique en effectuant la commande suivante

  ln -sf /usr/src/linux/include/asm /usr/include/asm

  On remarque que **/usr/src/linux/include/asm** est déjà un lien symbolique.
- **mpcview.o**
  Le point concerne l'utilisation de **bzero**. Une première solution, peu ortodoxe consiste à modifier le **forms.h**, c'est à dire mettre en commentaire la ligne suivante

  #include <string.h>

Enfin, Le passage sur la version 2.4.x a entrainé une modification notoire de la gestion de la mémoire, comme nous l'avons vu précédement, puisque la structure *vm_area_struct* a été modifiée considérablement. Ce qui nous empêche pour le moment d'utiliser **CMEM** sur les nouveaux noyaux.

10.3.2 Problème de compilation des kernels

Le problème principal lors de la cohabitation des différents kernels est bien sur le problème des liens. l'autre petit problème est la mise à jour des modules. Avant chaque recompilation et installations des modules, il est souhaitable de faire table raz dans le répertoire correspondant au noyau concerné

rm -rf /lib/modules/2.2.xx

Une autre solution beaucoup plus propre consiste à modifier le **Makefile**, soit modifier la variable *EXTRAVERSION* en l'incrémentant par exemple, à chaque évolution du kernel.

Bibliographie

[1] http ://www.kerneltrap.com/.

[2] http ://www.linuxhq.com/.

[3] http ://www.linuxdoc.org.

[4] http ://www.tux.org/lkml/.

[5] http ://neworder.box.sk.

[6] Rémy CARD Eric DUMAS Franck MEVEL. *Programmation Linux 2.0.* Eyrolles, 2000.

[7] Daniel P.BOVET Marco CESATI. *Understanding the Linux Kernel.* O'Reilly, 2001.

[8] Scott MAXWELL. *LINUX Core Kernel Commentary.* Coriolis, 1994.

[9] Jean Marie RIFFLET. *La programmation sous UNIX.* Ediscience, 1998.

[10] Alessandro RUBINI. *Linux Device Driver.* O'Reilly, 1998.

[11] Marshall Kirk MC KUSICK Keith BOSTIC Michael J.KARELS John S.QUATERMAN. *The Design and Implementation of the 4.4BSD.* Addison-Wesley, 1996.