TELECOM INT
Février- Juillet 2001

PORTAGE D'UNE COUCHE DE COMMUNICATION NOYAU EN MODE UTILISATEUR

ABDELLOU Djilali
sous l'encadrement de
GLUCK Olivier

Remerciements

Je remercie l’ensemble de l’équipe de l’ASIM pour m’avoir accueilli au sein de son département en particulier, Alain Greiner pour m’avoir permis d’effectuer ce stage.

Également, un grand merci à Olivier Gluck qui m’a encadré, Alex Fenyo pour son aide, Daniel Millot et Philippe Lalavée pour le soutien qu’ils m’ont apporté tout au long de ce stage.

Résumé

Ce stage s'inscrit dans le cadre du projet MPC qui a démarré en 1995 sous la direction d'Alain Greiner. Il a pour but la réalisation d'une machine parallèle à base d'ordinateurs du commerce interconnectés sur un réseau haute performance.

Mon travail au cours de ce stage est le portage de la couche de communication noyau dans l'espace utilisateur. Pour valider le fonctionnement de cette nouvelle couche, on s'est durant le reste du stage concentré sur la mesure de ses performances.

Abstract

This training period is a part of the global project called MPC, which started in 1995 under the direction of Pr. Alain Greiner. The projet s'aim is to build a low cost parallel computer, based on standard PC interconnected with a high performance network.

My role in the project was first to port the PUT communication layer , which was in kernel mode to the user mode. After that, I measure the performance of this layer to compare with the old layer.

Sommaire

Chapitre 1 : Introduction

Chapitre 2 : Généralités

Chapitre 3 : La machine MPC

Chapitre 4 : Put en mode utilisateur

Table des figures

Chapitre 1

A. Sujet du stage

A.1 Objectif

A.2 Description

A.3 Moyens utilisés

A.4 Encadrants

B. Introduction et contexte du stage

B.1 Présentation de l'organisme d'accueil

B.2 L'université Paris VI

A. Sujet du Stage

A.1 Objectif :

Mon stage s'est effectué au sein du département **ASIM** (Architecture des systèmes intégrés et microélectronique) du laboratoire d'Informatique de **Paris VI** le **LIP6** (figure 1) .

Le projet **MPC** est l'un des huit projets transversaux qui déterminent la politique scientifique du **LIP6**. L'idée générale est d'aidé certains demandeurs de puissance de calcul du **LIP6** à mener des expériences sur la machine **MPC** développée au laboratoire dans le thème **ASIM**. Le but du projet **MPC** est la conception d'une machine parallèle performante et à faible coût.
Le stage que j'ai effectue s'inscrit dans ce projet, le thème étant le portage en mode utilisateur de la couche de communication noyau existante.

A.2 Description :

Le projet **MPC** (*Multi-PC*), a démarré en janvier 1995 sous la responsabilité d'**Alain GREINER** et s'intègre dans la volonté de concevoir des architectures parallèles à base de réseau de stations du commerce. La machine **MPC** est constituée de cartes processeurs Pentium du commerce interconnectées à travers un réseau rapide Giga bits composé de cartes utilisant la technologie **HSL** devenue le standard **IEEE 1355**. Pour utiliser au mieux les performances de la carte **FastHSL** un certain nombre de couches logicielles ont été développées. Parmi elle, la couche **PUT** occupe une place principale, elle accède directement à la carte **FastHSL** pour lui fournir les informations sur les données à transmettre. Cette dernière développée en mode noyau oblige à l'utilisation d'appels systèmes coûteux en temps pour accéder à ses fonctionnalités. Soucieux d'accroître les performances de la machine **MPC** , le choix fut pris d'éliminer les appels systèmes pour accéder à **PUT**.

Mon stage s'inscrit dans cet objectif, il a commencé par une prise en main des différentes couches logicielles composant **MPC-OS**. Une fois cette étape franchit, il a fallut simplifier au maximum cette couche de communication en ne gardant que les fonctionnalités nécessaires et rendre un maximum de fonctionnalités en mode utilisateur. La deuxième partie a été consacrée à la mesure de performances.

A.3 Moyens utilisés :

Pour mener cette tache a bien nous disposions de l'ensemble du package composant **MPC-OS** (les sources et la documentation) a cela s'ajoute deux **PC** (Pentium 166) sous **LINUX** (**RedHat 7.0**) ainsi que de deux cartes **FASTHSL** .

A.4 Encadrants :

Directeur de stage : Alain GREINER
Encadrant de stage : Olivier GLUCK
Conseiller d'étude : Daniel MILLOT

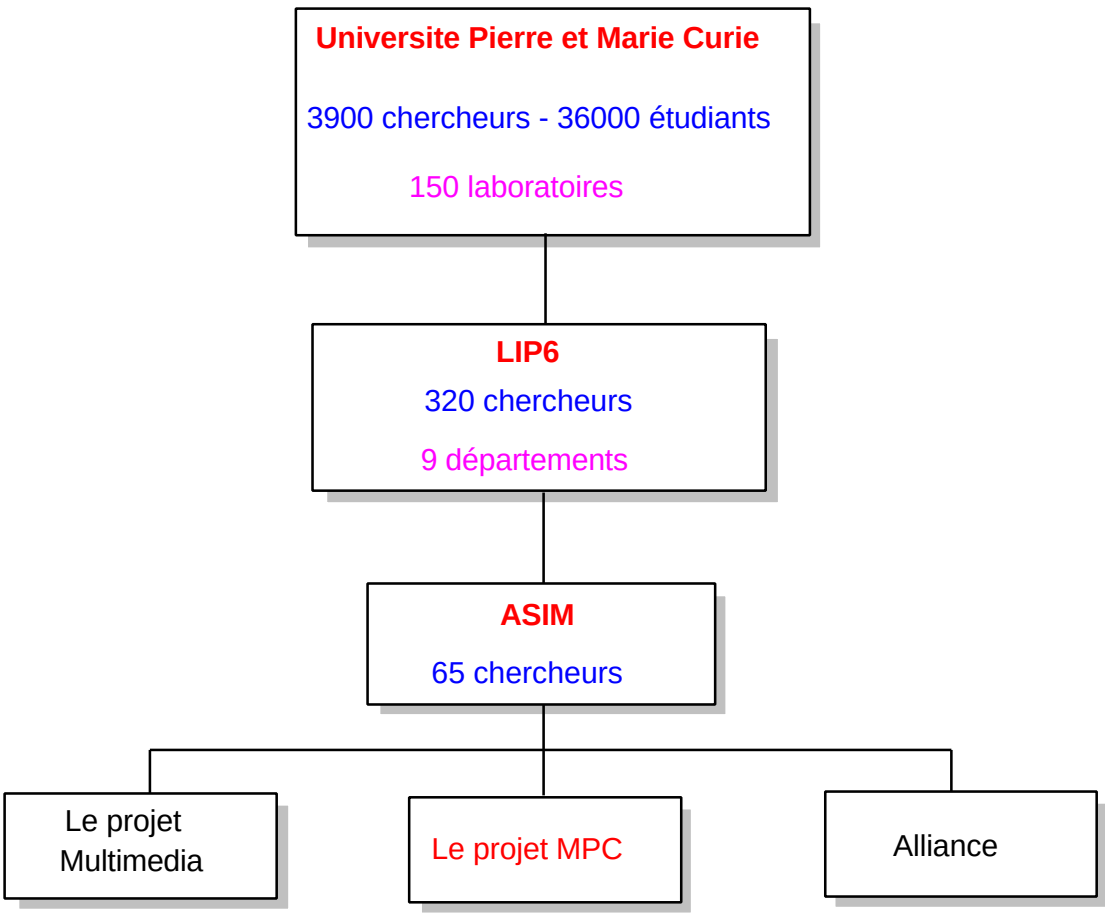

Figure 1 : de UPMCP6 au projet **MPC**

B. Introduction et contexte du stage

B.1 Présentation de l'organisme d'accueil :

Le laboratoire d'Informatique de **UPMCP6** est une unité mixte de recherche de l'université Paris VI et du Centre National de la Recherche Scientifique (**CNRS**). Près de 300 personnes travaillent au **LIP6**, assurant des travaux d'enseignements et de recherche dans le domaine de l'informatique. Plus de la moitié de cet effectif est composé d'étudiants préparant leur thèse de doctorat.

Le **LIP6** est réparti sur deux sites: une partie de l'effectif est implanté sur le campus de Jussieu tandis que l'autre moitié est rue de l'Amiral Scott (Paris XV). Le laboratoire se divise en 9 départements, répartis sur ces deux sites :

ANP	: Algorithmique numérique et parallélisme
APA	: Apprentissage et acquisition de connaissances
ASIM	: Architecture des systèmes intégrés et micro-électronique
CALFOR	: Calcul formel
OASIS	: Objets et Agents pour Systèmes d'Informations et de Simulation
RP	: Réseaux et Performances
SPI	: Sémantique, Preuve et Implantation
SRC	: Systèmes Répartis et Coopératifs
SYSDEF	: Systèmes d'aide à la décision et à la formation

pour plus d'information : http://www.lip6.fr

Le département **ASIM** se consacre à la conception de circuits intégrés, au développement de logiciels de **CAO** pour la **VLSI** et de l'étude des architectures matérielles des systèmes. Il regroupe 65 personnes sur le campus de Jussieu.

les enseignements dispensés par le département sont :

- Le **DEA** en architecture des systèmes intégrés et micro-électronique **ASIME**
- Le **DESS** de circuits intégrés et systèmes analogique numérique **CISAN**
- Les cours d'option architecture de la maîtrise d'informatique de Paris VI
- Les cours d'option **MEMI** de la maîtrise **EEA** de Paris VI

Les activités de recherche du département se répartissent autour de trois projets (figure 1) :

- Le projet **MPC**
- Le projet d'indexation multimédia
- Le projet **CAO** de circuits et systèmes (**Alliance**)

pour plus d'information : http://www-asim.lip6.fr

B.2 L’université de Paris VI :

Principale héritière de l’ancienne faculté des sciences de Paris la **Sorbonne**, l’université **Pierre et Marie CURIE** forme plus de 3000 cadres haute technologie par an, elle est la première université scientifique et médicale de France.
L’université en quelques chiffres:

- 36000 étudiants dont 10000 en troisième 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 VI dispense plusieurs types de formations :

- Formations scientifiques
- Formations médicales
- Formations professionnelles
- Formations d’ingénieurs
- Formations permanentes (5000 stagiaires par an)

Chapitre 2

A Quelques définitions

A.1 Machine virtuelle

A.2 Mode noyau, mode utilisateur

A.3 Appel système

A.3.1 Généralités
A.3.2 Procédure d'exécution

A.4 Les modules chargeables sous Linux

A.4.1 Fonction init_module()
A.4.2 Fonction cleanup_module
A.4.3 Compilation

B. Les architectures parallèles

B.1 Grappe de PCs

B.2 Le modelé de programmation

A. Quelques définitions

A.1 Machine virtuelle :

Un système d'exploitation offre une machine virtuelle à l'utilisateur et aux programmes qu'il exécute. Le système d'exploitation s'exécute sur une machine physique qui possède une interface de programmation de bas niveau et fournit des abstractions de hauts niveaux et une interface de programmation et d'utilisation plus évoluée.

Contrairement aux années 50, durant lesquelles les programmeurs devaient connaître l'interface physique de la machine afin de la programmer, les systèmes d'exploitations modernes offrent un tel niveau d'abstraction, qu'ils traduisent les requêtes de hauts niveaux effectuées par l'utilisateur en requêtes physique de bas niveau. Les systèmes d'exploitations sont une interface entre les applications et la machine physique (figure 2).

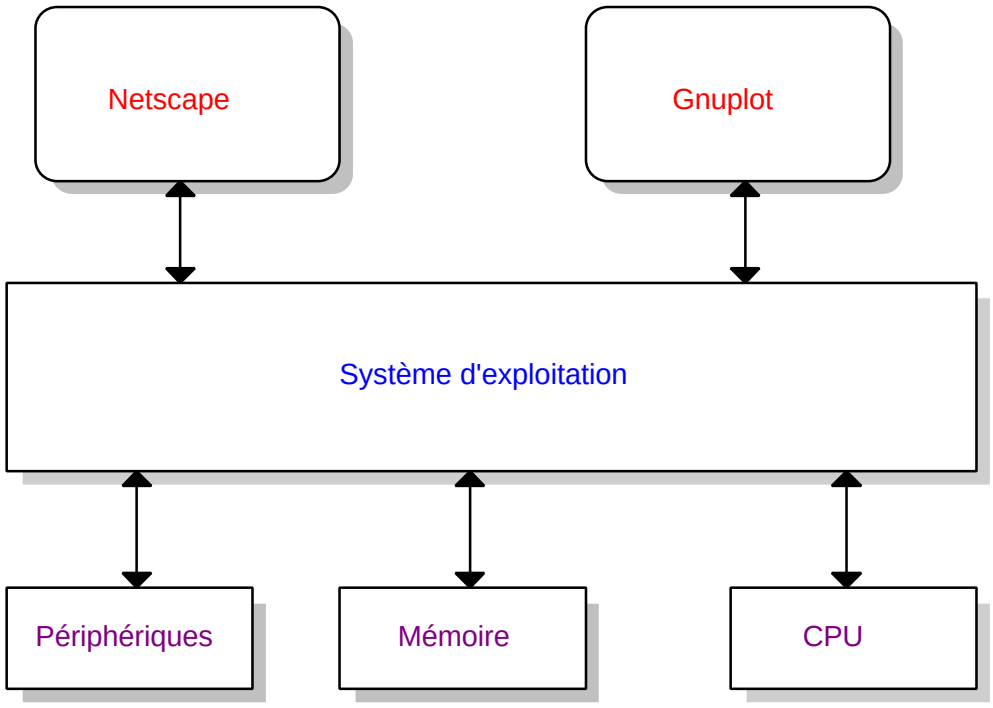

Figure 2 : schéma de l'interaction application système d'exploitation et machine physique

Tous les accès aux périphériques (internes ou externes, mémoire, carte Ethernet) sont réalisés par le système d'exploitation. Cette abstraction de la machine matérielle permet de libérer les développeurs de la complexité liée à la gestion des différents périphériques et d'éviter de se limiter à une seule machine.

Lorsqu'un développeur désire lire le contenu d'un fichier, il effectue la même opération que le fichier soit sur le disque dur ou sur une disquette. Le code du programmeur reste le même, mais le noyau effectue des opérations différentes en fonction du périphérique sur lequel se trouve le fichier.

A.2 Mode noyau, mode utilisateur :

Un processus sur un système **UNIX** (**Linux**) possède deux niveaux d'exécution (noyau et utilisateur)

- Le mode noyau est un mode privilégié : dans ce mode, aucune restriction n'est imposée au noyau du système. Il peut utiliser l'ensemble des instructions du processeur, manipuler toute la mémoire et dialoguer directement avec tous les contrôleurs des périphériques.

- Le mode utilisateur : est le mode d'exécution normal d'un processus. Dans ce mode le processus ne possède aucun privilège. Certaines instructions lui sont interdites, il ne peut interagir directement avec la machine physique.

A.3 Appel système :

A.3.1 Généralités :

Un processus qui s'exécute en mode utilisateur sous Linux, ne peut accéder directement aux ressources de la machine. Il doit pour cela effectuer des appels systèmes. Un appel système est une requête transmise par un processus au noyau afin que la requête soit traitée en mode noyau. Une fois cet appel terminé le résultat est retourné au processus appelant qui reprend son exécution. Sous les systèmes d'exploitation de type **UNIX** (**Linux**) l'appel système provoque une interruption logicielle qui fait passer le processus en mode noyau et qui ne repassera en mode utilisateur qu'à la fin de l'exécution de l'appel.

A.3.2 Procédure d'exécution :

Un appel système est caractérisé par un nom et un numéro qui l'identifie. La bibliothèque C fournit des fonctions permettant d'exécuter des appels systèmes. Ces fonctions portent le même nom que les appels systèmes associés. Pour exécuter un appel système, il suffit sous linux d'exécuter la fonction C associée.

-l'ensemble des appels systèmes sont répertoriés dans le fichier **entry.S** dans répertoire **/usr/src/linuxx.xx.xx/i386/kernel**

-tandis que les numéros eux sont regroupés dans le fichier **/usr/include/asm/Unistd.h**

Lorsqu'un processus exécute un appel système, il fait appel à la fonction correspondant de la bibliothèque C. Cette fonction fait passer le processus en mode noyau. Sur l'architecture **X86** ce passage s'effectué de la manière suivante.

- 1 les paramètres de l'appel système sont placés dans des registres du processeurs

- 2 Une trappe est provoquée en déclenchant une interruption logicielle 0x80

- 3 Cette trappe provoque le passage en mode noyau, le processus exécute la fonction system_call définie dans le fichier **entry.S**

- 4 Cette fonction utilise le numéro de l'appel système pour appeler la fonction noyau correspondant.

- 5 au retour de cette fonction, system_call retourne à l'appelant.

A.4 Les modules chargeables sous Linux :

Le but n'est pas ici d'expliquer tout le fonctionnement et les techniques qui sont liées à la programmation de modules chargeables sous **Linux**, mais simplement d'en donner les grandes lignes.

Les modules chargeables de **Linux** sont des fichiers objets issus d'une compilation standard. Ils peuvent contenir une ou deux fonctions nommées *init_module()* et *cleanup_module()*. Un module chargeable ne contient pas de fonction principale *main()*, la fonction *init_module* fait office de fonction main.

A.4.1 Fonction init_module :

La fonction *init_module()*, obligatoire, doit avoir le prototype suivant :
init_module(void);
C'est la première fonction appelée lorsque le module est chargé en machine. Elle permet d'initialiser le module et d'enregistrer les fonctions dans les différentes tables du noyau (driver, handler d'interruption, ...). Le code de retour de cette fonction suit la convention suivante:

0 si tout c'est déroulé correctement
sinon un code d'erreur dans le cas contraire

A.4.2 Fonction cleanup_module :

La fonction *cleanup_module()* est facultative, elle a pour prototype:
void cleanup_module(void);
Elle est appelée lorsque le module doit être décharger de la mémoire. Elle permet de défaire les fonctions que le module a enregistrées dans les tables du noyau.

A.4.3 Compilation d'un module :

Pour compiler un module il faut inclure les fichiers suivants:
linux/module.h
linux/kernel.h
sans oublier de définir les deux macros suivantes:
#define __KERNEL__
#define MODULE
La compilation est celle d'un fichier objet courant, il faut cependant l'option **-o**. De plus, il est possible d'associer plusieurs fichiers objets en un seul en utilisant la commande ld -r.

A.5 Les drivers sous Linux :

Il existe plusieurs types de drivers sous **Linux**. Les drivers de caractères, les drivers de blocs et ceux orientés réseaux. Nous allons nous concentrer sur les drivers caractères utilisés par **MPC-OS**. Nous ne ferons ici qu'une approche pratique. Le module chargeable peut indiquer au noyau les nouvelles fonctionnalités qu'il fournit. Une des ces fonctionnalités est l'accessibilité via un fichier spécial. Accessibilité qui peut se faire du noyau vers l'utilisateur ou inversement. Nous allons voir comment déterminer la fonction qu'il doit effectivement appeler.

La plus grande partie du travail est réalisée lorsque nous ouvrons le fichier. Ce qui en C se traduit par les lignes suivantes :

```
int fd;
fd = open (/dev/cmem", O_RDWR)
```

/dev/cmem est un fichier spécial qui donne accès au driver cmem

La librairie associée déclenche un appel système, le 5 sous **Linux**, par une interruption logiciel qui fait passer le processus en mode noyau qui exécute la fonction *syst_open()*. L'opération suivante qu'effectue cette fonction est d'aller chercher les opérations qui sont autorisées sur le disque associé au fichier.
Étant donné qu'il s'agit d'un fichier spécial, il n'y a pas de contenu réel, le système va trouver l'iNode associée au fichier pour déterminer qu'il est spécial.

Les fichiers spéciaux ont la particularité d'être crées avec deux attributs (le major et le minor) qui sont stockés à la place de la taille du fichier. Le major est indexé dans une

table gérée par le noyau alors que le minor est un nombre passé aux fonctions dudriver afin d'avoir un comportement différent pour plusieurs matériels d'un même type. Une fois l'iNode récupéré, il ne reste plus qu'a savoir quelle fonction appelée et qui a été associée au *open(),* cela est simple lorsqu'on sait que la structure d'un iNode contient un pointeur sur une structure des opérations réalisables sur ce nœud. La fonction *open()* de ce dernier est appelée. Le fichier étant spécial la fonction appelée est *chrdev_open(),* elle recherche dans sa table (chrdevs), le pointeur sur la structure de pointeur vers toutes les fonctions associées aux drivers (open, close, read, write, ioctl), la table est indexée par le major du fichier. La fonction réelle du fichier spécial peut alors être appelée à travers cette dernière structure.

```
struct file_operation{

        int (*open) (struct inode* , struct file*);

}
```

Le table qui permet d'associer le major à la structure pointant sur les fonctions réelles du driver est vide, il faut qu'au moment de l'initialisation du driver que celui-ci s'enregistre au noyau. Pour réaliser cette opération il faut exécuter *register_chrdev(),* qui prend en paramètre un major (la valeur 0 signifie qu'on demande au système d'en trouver un libre) , le nom du driver ainsi qu'une structure sur les opération réalisables.

```
int register_chrdev(unsigned int major,
      const  char *name,
      struct file_operation *fops);
```

Ces fonctions sont appelées par un appel système qui vérifiera la validité du descripteur de fichier. La structure file_operation de la table des fichiers ouverts contenant les pointeurs sur les fonctions a appelé est alors consultée et l'appel est transmis à la fonction réelle du fichier ouvert.

B. Les architectures parallèles

B.1 Grappe de PCs :

Depuis le début de l'informatique les performances du matériel associé n'ont cessé de s'accroître. Malgré tout, dans certains domaines cela reste insuffisant et afin de palier à ce problème des machines spécifiques ont été développées tel que la **CRAY** ou encore la **PARAGON** d'**Intel**.

Ces machines complexes a réaliser ont un coût prohibitif, c'est pour cette raison qu'une alternative a été trouvée dont l'idée générale consiste au raccordement de plusieurs machines au dessus d'un réseau rapide.

L'un des premiers projets à voir le jour fut celui initié par une équipe de **Berkeley USA** et qui a rassemblé pour cette expérimentation 100 **ULTRA Sparc**, 40 **Sparc Stations Sun** sous **Solaris** , 30 PC sous **NT** et **UNIX** , 300 stations de travail sous **HP**.

B.2 Le modèle de programmation :

Une machine parallèle sert à programmer des applications parallèles, par conséquent un modèle de programmation est nécessaire.

Les grappes de PC ne disposant pas de mémoire partagée, le modèle de programmation le plus couramment utilisé est basé sur l'échange de messages. Le programmeur doit de lui-même gérer l'ensemble des données. Ce modèle est basé sur deux fonctions principales (**send** et **receive**), le send permet de transférer les messages tandis que la fonction receive permet d'en recevoir.

Il existe principalement deux bibliothèques de passage de message utilisées dans les réseaux de stations.

- **PVM** (**P**arallel **V**irtual **M**achine)
- **MPI** (**M**essage **P**assing **I**nterface)

Elles sont construites sur des langages de programmation séquentiels standards comme le C ou encore le Fortran.

Chapitre 3

A Le projet MPC

B Architecture générale de la machine MPC

B.1 La zéro copie

B.2 La Carte FastHSL

B.2.1 L'espace I/O PCI et l'espace mémoire PCI
B.2.2 L'espace de configuration PCI
B.2.3 La carte FastHSL

B.3 Présentation de MPC-OS sous Linux

B.3.1 structure de la distribution
B.3.2 Le module CMEM
B.3.3 Le module HSL
B.3.4 Les démons de contrôle

A. Le projet MPC

Le projet **MPC** est né suite aux activités de design de circuits intégrés du département **ASIM**, après la conception d'un routeur hautes performances **Rcube** destiné à plusieurs applications, tels que les commutateurs giga bits Ethernet ou encore les machines parallèles. C'est dans cette deuxième voie que le projet **MPC** s'est bâti. Le but du projet est le développement d'une machine parallèle utilisant des ordinateurs du marché connectés au dessus d'un réseau hautes performances.

Ce projet regroupe les chercheurs du département **ASIM**, du **PRISM** de l'Université de Versailles, du **LARIA** de L'Université de Picardie Jules Verne ainsi que des chercheurs du département informatique de l'**INT** (**INT-Evry**) et de l'école Nationale Supérieure de Télécommunications de Paris (**ENST-Paris**).

B. Architecture générale de la machine MPC

Dans cette partie nous essayerons d'expliquer le fonctionnement général de la machine **MPC**. Csection a été découpée en trois sous parties avec pour commencer une section justifiant le choix effectué pour la méthode de zéro copie. Cette partie laissera place à une deuxième section qui fera un point sur les interfaces PCI et expliquer celle développée pour la carte **FastHSL** pour enfin nous concentré sur les couches logicielles développé à l'occasion de ce projet.

B.1 La zéro copie :

Pour comprendre l'enjeu mis en oeuvre, nous allons essayer d'expliquer en quoi le choix de la zéro copie peut-être un élément qui va forcément augmenter les performances de machines communicants au dessus d'un réseau. Plaçons nous dans le cas ou une station au sein d'un réseau veut dialoguer avec une autre connectée à ce même réseau. Typiquement, au sein de la première station, un processus utilisateur va pour cela utiliser une primitive d'envoi "send". Celle-ci pour réaliser cet envoi va dans tous les cas faire un appel système, c'est à dire un passage en mode noyau, car comme nous l'avons vu précédemment ce cheminement est inévitable, seul le noyau peut avoir accès au matériel (carte Ethernet).

Au cours de cet appel, le processus recopie les données a transmettre de l'espace utilisateur vers l'espace noyau. Une fois en mode noyau, le pilote (driver) de la carte va se mettre au travail et va recopier les données à transmettre dans la mémoire d'entrée/sortie de la carte qu'il acheminera jusqu'au nœud destinataire. Une fois que l'ensemble des données sont arrivées, la carte génère une interruption pour signaler leur arrivée. Cette interruption fait passer la machine en mode noyau, qui par l'intermédiaire du pilote de la carte va récupérer les données pour ensuite les transmettre au processus auquel elles sont destinées.

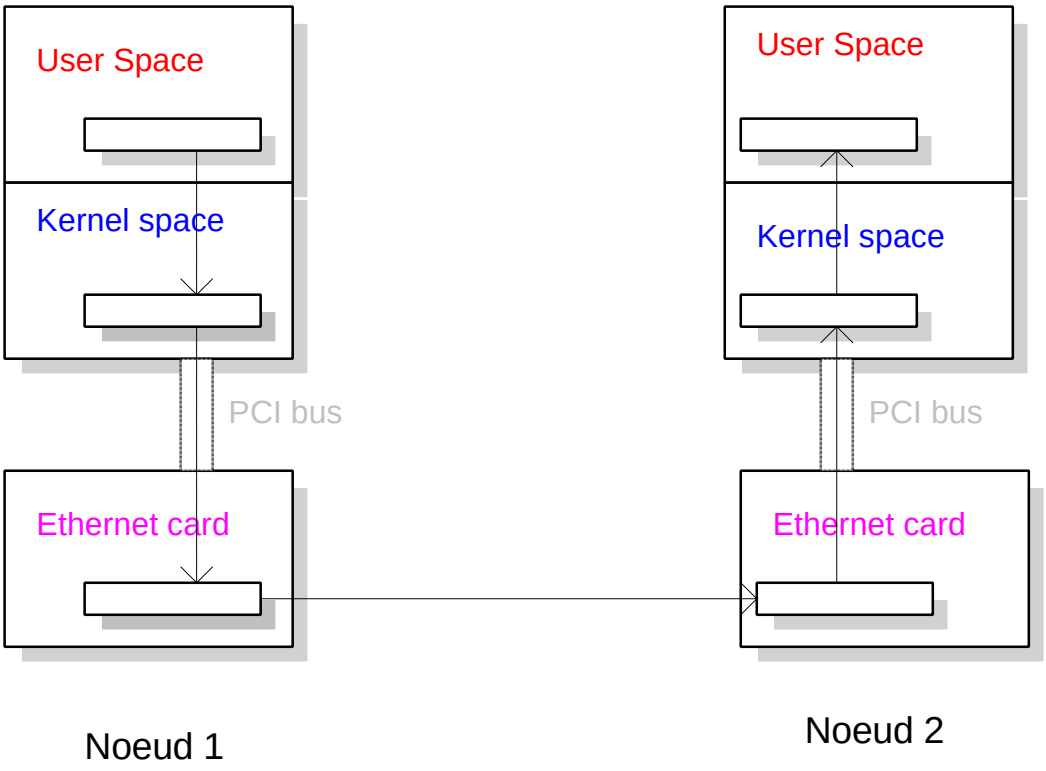

Figure 3: Échange d'information entre 2 nœuds dans un réseau Ethernet

Durant cette phase de communication, le processus de chaque machine a effectué deux copies des même données. Le circuit **Rcube** est capable de débiter 1Gbits/s, mais si son exploitation est faite avec les mécanismes classiques, les performances ne seront pas aussi importantes que le circuit le permette. Pour utiliser cet élément au maximum de ces capacités, le principe de la zéro copie complété par le mécanisme de **DMA** ont été choisies. Le **DMA** est un mécanisme permettant à une périphérie d'avoir accès directement aux données en mémoire centrale (soit en lecture, soit en écriture) sans passer par le processus. On a donc adjoint à **Rcube** un second composant **PCIDDC** qui implémente ce mécanisme afin de tirer partie de toutes les possibilités offertes par **Rcube**.

B.2 La carte FastHSL :

Les circuits **Rcube** et **PCIDDC** sont montés sur une carte **PCI**, appelée carte **FastHSL** (HSL : **H**igh **S**peed **L**ink), qui prend place dans un slot d'extension du PC. Mais avant de débuter l'explication sur le fonctionnement de cette carte, nous allons au préalable faire un bref rappel sur l'interface **PCI**. **PCI** (**P**eripheral **C**omponent *I*nterconnect) est un standard qui décrit comment connecter différents périphériques et de quelle manière il faut les contrôler. Chaque carte **PCI** qui compose un ordinateur possède trois espaces qui sont d'une part l'espace mémoire, d'autre part l'espace de configuration et enfin l'espace d'entrée/sortie.

B.2.1 L'espace d'entrée/sortie PCI et l'espace mémoire PCI :

Ces deux espaces sont utilisés par la carte **PCI** pour communiquer avec le driver. Typiquement les drivers des cartes vidéo utilise très largement ces espaces afin de stocker les informations vidéo a afficher. De manière générale une application en mode utilisateur doit seulement remapper l'espace d'entrée/sortie de la carte **PCI** dans le sien pour permettre à ce même processus d'échanger des informations avec cette carte.

B.2.2 L'espace de configuration PCI :

Comme son nom l'indique ce troisième et dernier espace permet la configuration de la carte. Un tel espace permet au système de récupérer des informations sur la carte afin de la configurer au mieux. A cet effet une structure figée a été imposée à cet espace dont l'allure est la suivante :

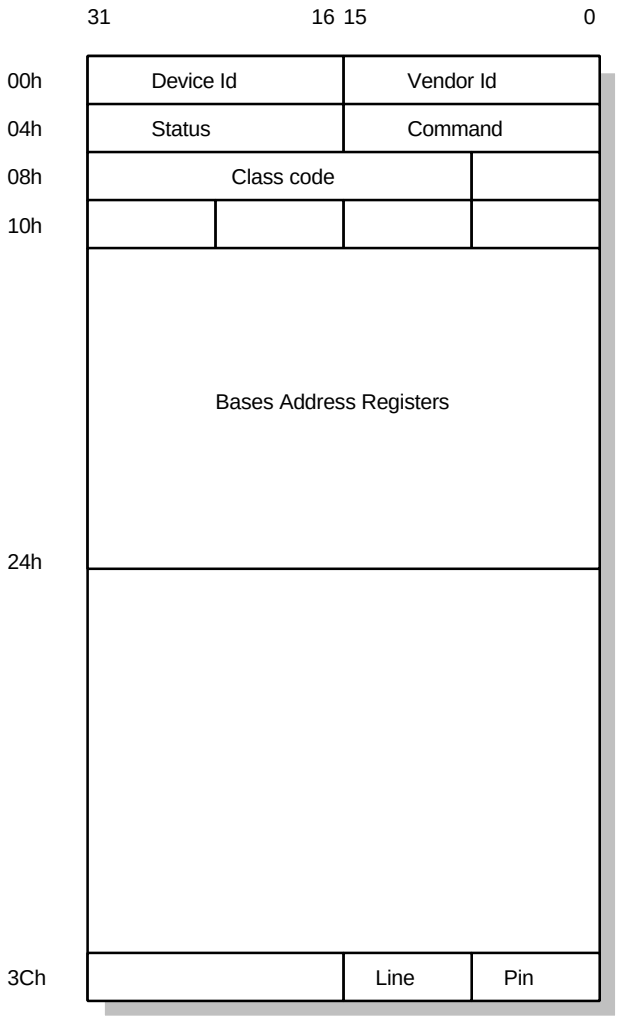

Figure 4 : Espace de configuration d'une interface PCI

- Vendor Id : Vendor Identifier
  le vendor Id est un numéro unique identifiant le constructeur de la carte
  exemple : pour Intel cette valeur en hexadécimale est 0x8086
  dans le cas de la carte FastHSL cette valeur a été imposée 0x0000
- Device Id : Device Identifier
  Comme le vendor Id le device Id est un numéro unique permettant d'identifier le type de la carte
  exemple : la carte fast Ethernet de Digital a pour le device la valeur 0x0009
- Status : Ce champs permet d'avoir l'état de la carte, chaque bit de ce champs fournit une information spécifique.

B.2.3 La carte FastHSL :

La carte **FastHSL** constituant la machine **MPC** se distingue des cartes **PCI** traditionnelles dans le sens ou, contrairement à ces dernières, la carte **FastHSL** ne dispose que de l'espace de configuration. Ceci présente l'avantage de n'avoir qu'un seul espace aussi bien pour la configuration que pour la transmission des données, mais l'inconvénient est l'impossibilité de mapper cet espace afin de permettre à un processus de transmettre directement ces données à la carte. De ce fait l'échange des données est forcement réalisé par le noyau. Des registres supplémentaires ont été définis pour fournir les informations sur la localisation physique des données à transmettre à **PCIDDC**.

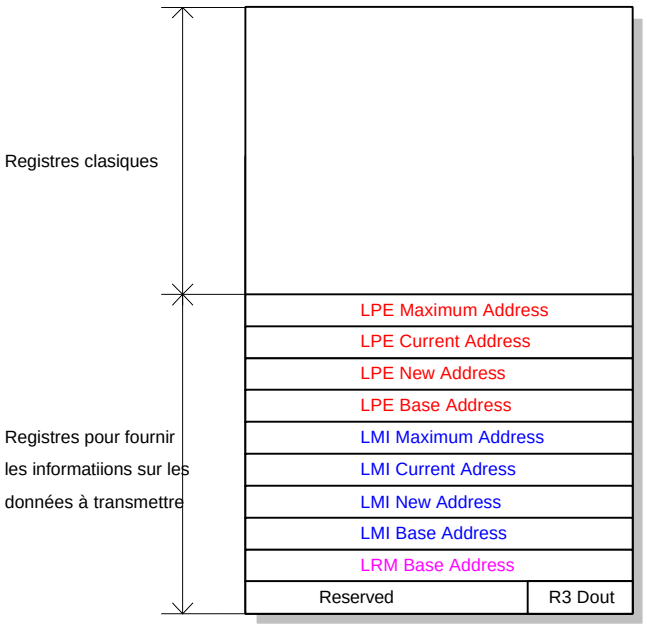

Figure 5 : Espace de configuration de FastHSL

La carte **FastHSL** utilise le principe des boites à lettres, le concept est assez simple: on ne lui fournit que les adresses des données a émettre du nœud expéditeur et l'adresse de l'emplacement mémoire chez le destinateur.

Pour pouvoir aussi bien recevoir qu'émettre, huit registres ont été définis, la première moitié pour l'émission et la seconde pour la réception. Pour l'émission le processus teint à jour une liste circulaire des messages a émettre dont l'adresse de début et de fin sont stockées dans les registres de la carte (**LPE_MAX** et **LPE_BASE**). Lorsqu'un processus fait un send il va dans un premier temps copier dans la liste circulaire, la **LPE** (**L**ist **P**age **E**mit), l'ensemble des informations liées aux données, notamment l'adresse mémoire physique des données à émettre et l'adresse mémoire physique de leur destination.Une fois cette opération effectuée, le processus modifie la valeur du registre **LPE_NEW** de la carte pour lui indiquer ou aller chercher les informations sur les données et surtout qu'il y a quelque chose a transferer. La carte constatant que les registres **LPE_NEW** et **LPE_CURRENT** sont différents lit les informations sur les données à transmettre. Une fois l'envoieffectué le valeur du **LPE_CURRENT** est incrémentée et l'opération est répétée jusqu'à ce que la valeur du **LPE_CURENT** rattrape celle du registre **LPE_NEW**.

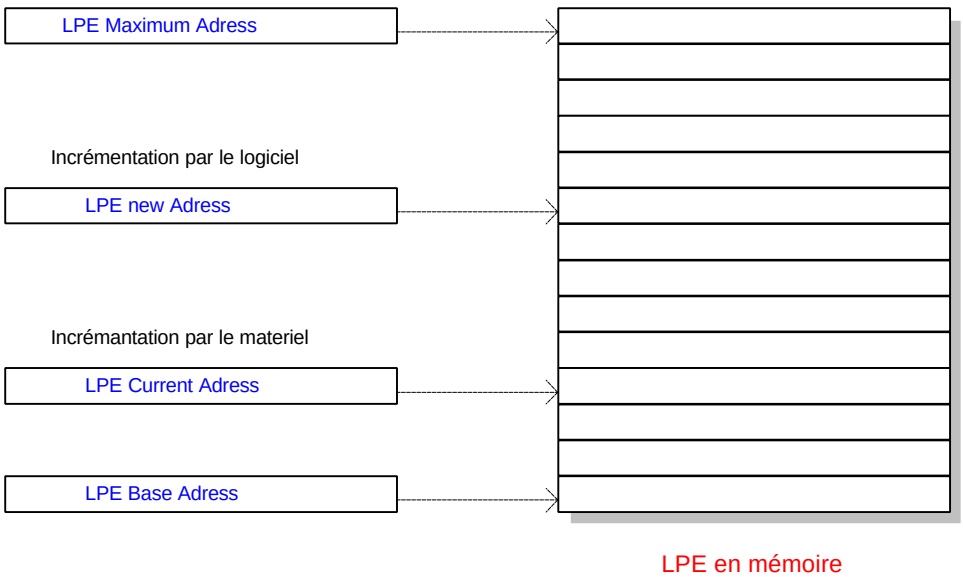

Figure 6: gestion de la LPE

Pour la réception, on utilise le même principe avec une liste circulaire différente, la **LMI** (**L**ist of **M**essages **I**dentifier), elle dispose de deux registres qui délimitent la borne inférieure et supérieure de cette liste et deux autres registres **LMI_CURRENT** et **LMI_NEW** qui se déplacent entre ces deux extrémités. Lors de la réception des messages **LMI_NEW** est actualisé par la carte **FastHSL** tandis que le **LMI_CURRENT** est lui est incrémenté par la couche de communication.

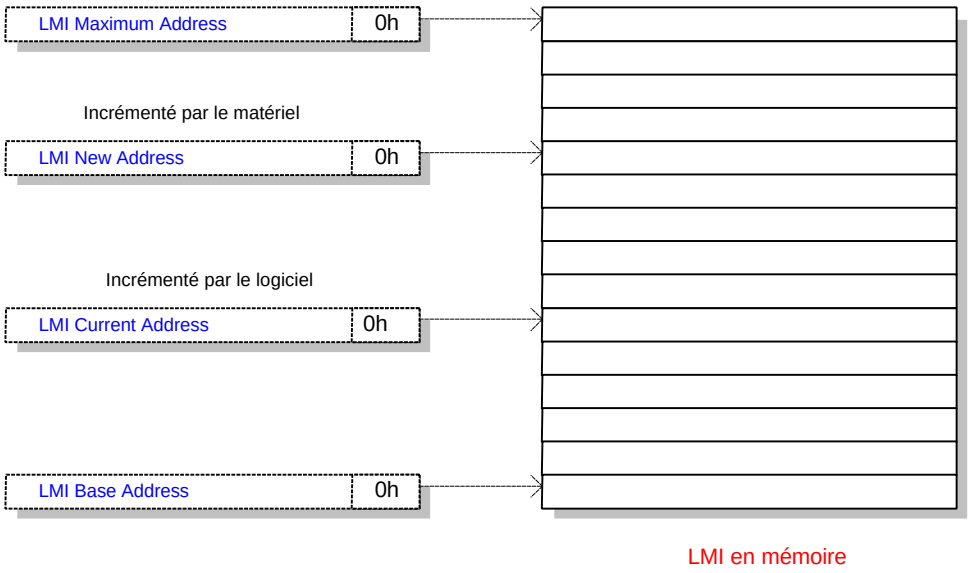

Figure 7 : gestion de la LMI

B.3 Présentation de MPC-OS sous Linux

Dans cette partie nous allons essayer de détailler la constitution logicielle de ce projet. Elle aura aussi pour rôle de décrire les différents éléments constituant **MPC-OS**, ainsi que leurs interactions.

B.3.1 Structure de la distribution MPC-OS :

La distribution du projet **MPC** se présente sous la forme d'un fichier tar zipper qui une fois décompacté donne l'arborescence suivante:

- mpc-linux/ : Racine de la distribution
- mpc-linux/modules : Modules **CMEM** et **HSL**
- mpc-linux/doc : Documentation
- mpc-linux/routing : Configuration des tables de routages
- mpc-linux/src : Utilitaires divers
- mpc-linux/src/emu : Démons de contrôle
- mpc-linux/src/testput : Utilitaires de test (testputbench)

La mise en route des machines MPC nécessite une procédure d'initialisation. Cette procédure impose le chargement des modules **CMEM** et **HSL** suivit du lancement des deux démons de contrôle **HSLclient** et **HSLserveur**. Une fois cette opération effectuée, il est possible à un processus quelconque d'utiliser **PUT**.

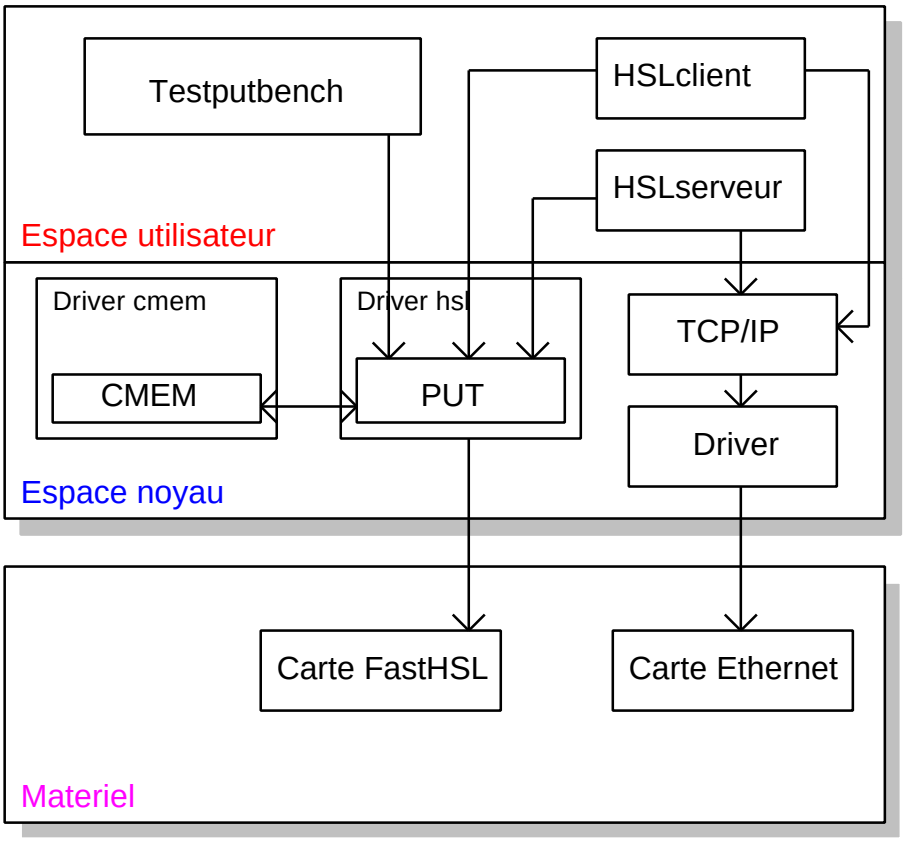

Figure 8: Agencements logiciels

B.3.2 Le module CMEM :

Le module **CMEM** fournit aux autres modules chargeables, un ensemble de fonctions permettant d'allouer des blocs mémoires contiguës afin de répondre à un besoin d'allocation de mémoire physiquement continue dans la machine **MPC**. **PCIDDC** voit la mémoire depuis le bus **PCI**, il n'a pas de vision translatée de la mémoire par la **MMU**. Par conséquent lorsque **PCIDDC** effectue un transfert, cet échange s'effectue uniquement sur des blocs mémoires physiquement contiguës. Lorsque l'on souhaite transférer une zone de mémoire quelconque d'une machine à une autre, on doit dans un premier temps examiner chaque page mémoire virtuelle source et destination, déterminer leurs adresses physiques pour enfin demander à **PCIDDC** de faire le transfert.

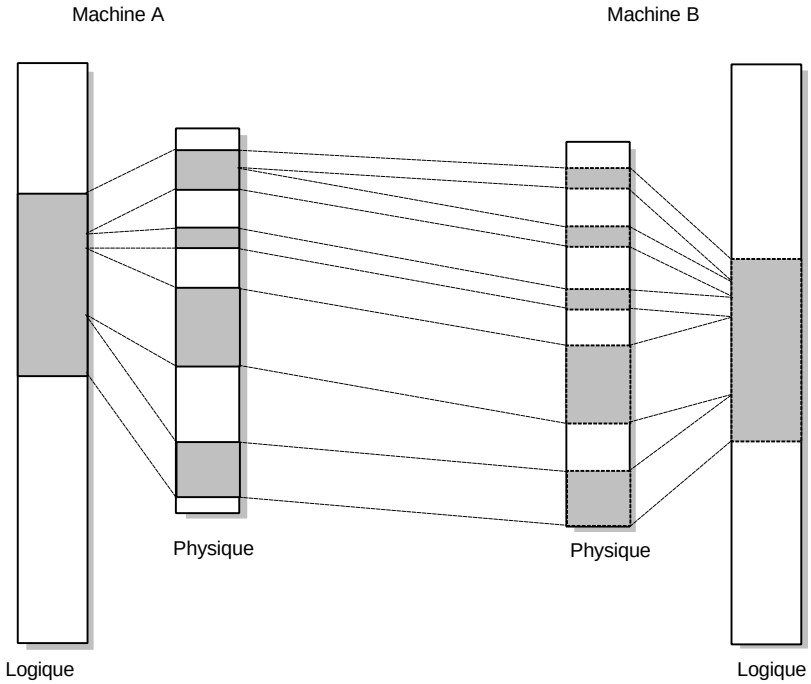

Figure 9: transmission de données non contiguës

Cette pénurie de mémoire physique contiguë, est incontournable, le système va tout au long de son fonctionnement allouer et désallouer des blocs mémoires (figure 10). On se retrouve par conséquent avec une mémoire physique morcelée, et nos données dispatchées là où il a de la place. Un processus utilisant **PUT** créant un buffer de taille importante aura de grande chance d'être dispatché en mémoire physique et il faudra plus d'un appel à **PUT** pour transférer l'ensemble des données (igure 9).

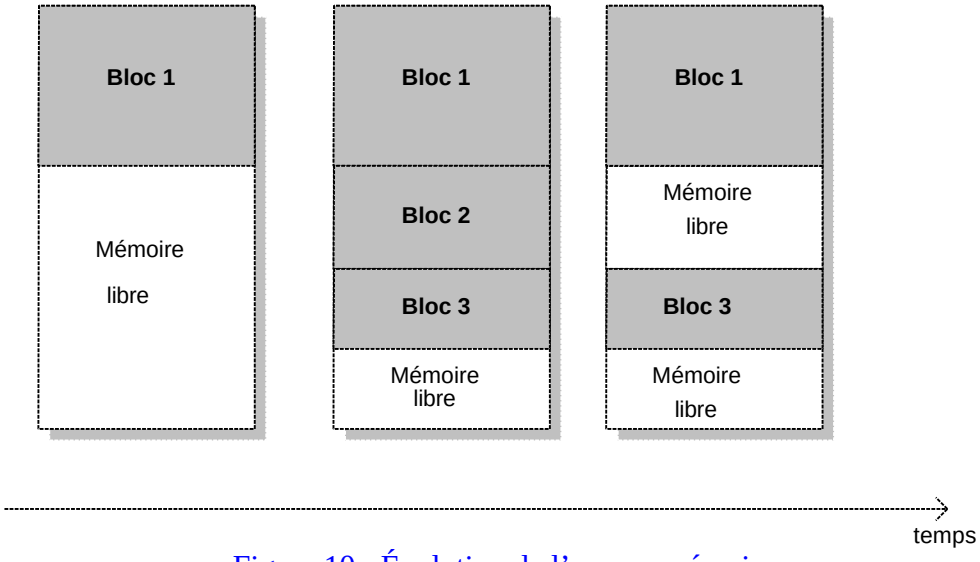

Figure 10 : Évolution de l'espace mémoire

Le module **CMEM** gère uniquement une table d'allocation de la mémoire dont il dispose. C'est un tableau de type cmem_slot_t:

```
typedef struct _cmem_slot{
        struct _cmem_slot *next_slot;
        struct _cmem _slot *prev_slot;
        unsigned long phys_start;
        void  *virt_start;
        size_t len;
        char name[CMEM_NAMELN];
} cmem_slot_t;
```

Les pointeurs *next_slot* et *prev_slot* servent au chaînage des slots utilisés. *phys_start* contient l'adresse physique de début du bloc mémoire tandis que *virt_start* l'adresse virtuelle du slot en mode noyau. Les deux derniers paramètres comme leur nom l'indique fournissent l'information sur le nom du slot et sur sa taille.

Les informations sur les slots alloués sont contenues dans les éléments constituant la liste chaînée. Afin d'actualiser tous ces champs, le driver **CMEM** dispose de quatre fonctions.

- *cmem_getmem()* permet à un utilisateur d'allouer un bloc **CMEM**
- *cmem_releasement()* libère un bloc mémoire alloué
- *cmem_virtaddr()* donne l'adresse virtuelle associée à une adresse physique
- *cmem_phys_addr()* donne l'adresse physique associée à une adresse virtuelle

B.3.3 Le module HSL :

Comme on peut le voir grâce à la figure 8 le module **HSL** est l'élément central de cette architecture, il a la charge d'exploiter la carte **FastHSL** à l'aide de **PUT**. **PUT** se divise en quatre fonctionnalités: configuration, signalisation, gestion des interruptions, et émissions des données.

La configuration : se fait a l'aide de quatre fonctions qui sont:

- *put_init()* permet l'initialisation de put et de la carte **FastHSL**
- *put_end()* permet la terminaison de **PUT**
- *put_register_SAP()* enregistre une application comme utilisatrice de **PUT**
- *put_unregister_SAP()* indique qu'une application ne veut plus utiliser **PUT**

La signalisation :

- *put_get_node()* renvoie le numéro du nœud
- *put_get_lpe_high()* renvoie un pointeur sur l'entrée de **LPE** courante
- *put_attach_mi_range()* attribue une plage de **MI** (**M**essage **I**dentifier) à l'utilisateur
- *put_get_mi_start()* renvoie le premier **MI** libre dans la plage attribuée
- *put_flush_lpe()* fait du polling sur le LPE pour notifier l'émission
- *put-flush_lmi()* fait du polling sur la LMI pour notifier la réception

L'émission :

- *put_add_entry()* ajoute une entrée dans la **LPE**

La gestion des interruptions :

- *put_interrupt_handler()* gestionnaire d'interruption

Le module **HSL** n'est pas seulement constitué de **PUT** il contient trois autres fichiers qui sont:

hsldriver.c qui implémente le driver en mode caractère
pci_ioctl.c contient les *ioctl()* d'accès au bus **PCI**
put_ioctl.c regroupe les *ioctl()* pour accéder aux fonctions de **PUT**

B.3.4 Les Démons de contrôle :

Les démons de contrôle du réseau **HSL** sont au nombre de deux, **HSLclient** et **HSLserveur**. Leur rôle est d'échanger les informations de configuration de la machine via le réseau de contrôle lors de l'initialisation de **PUT**. La seconde fonctionnalité de ces deux programmes est d'émulée les échanges de messages à travers le réseau de contrôle lorsque le réseau **HSL** ne fonctionne pas.

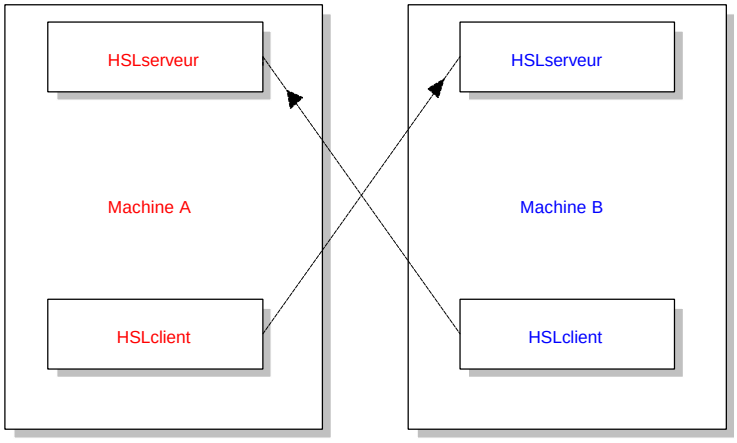

Figure 11 : Interaction entre les démons de deux nœuds

Chapitre 4

A. Objectif du stage dans le projet MPC

A.1 Le but du stage

A.2 Le changement de contexte

B. La couche de communication PUT

B.1 Vers le mode utilisateur

B.1.1 Structure du PUT noyau
B.1.2 Les problèmes identifiés
B.1.3 PUT en mode utilisateur

B.2 Simplification du code PUT

B.3 Travail effectué sur PUT

B.3.1 L'accès au bus PCI
B.3.2 L'initialisation
B.3.3 La signalisation
B.3.4 L'émission
B.3.5 Le partage des ressources

B.4 Les utilitaires

B.4.1 Testputbench
B.4.2 La gestion des MI
B.4.3 Testputsend_loop
B.4.4 Les démons
B.4.5 Visualisation de la LPE et la LMI
B.4.6 Utilisation du PUT utilisateur

A. Objectif du stage dans le projet MPC

A.1 Le but du stage :

Comme nous l'avons déjà indiqué dans les chapitres précédents, l'objectif premier de ce stage de six mois était la réécriture de **PUT** en mode utilisateur. En observant la figure 8, on constate qu'effectivement dans la distribution actuelle de **MPC-OS** (**M**ulti **PC** **O**perating **S**ystem) la couche de communication **PUT** est dans l'espace noyau. Pour qu'un processus quelconque puisse l'utiliser il est obligé de faire l'appel système *ioctl()*. Les soucis d'une telle architecture étant de toujours gagner en performances " rapidité " il a été constaté que lors des échanges entre deux nœuds le changement de contexte des processus communicants induisait une perte de temps non négligeable. Suite à cette constatation la solution la plus judicieuse était le portage de cette couche de communication en mode utilisateur.

A.2 Le changement de contexte :

Le driver **HSL** est accessible à travers un fichier placé dans /dev. Pour exécuter des fonctions spécifiques implémentées par **PUT** on utilise l'appel système *ioctl()*, auquel on passe un identifiant (lettre ou nombre) ainsi qu'un pointeur sur les données à transmettre.
Une fois l'appel effectué, ioctl passe la main à la méthode *ioctl()* du driver qui va identifier l'opération à réaliser, traite les arguments et les renvoies au processus initial. Bien entendu l'ioctl peut être utilisé de différentes manières avec ou sans argument.

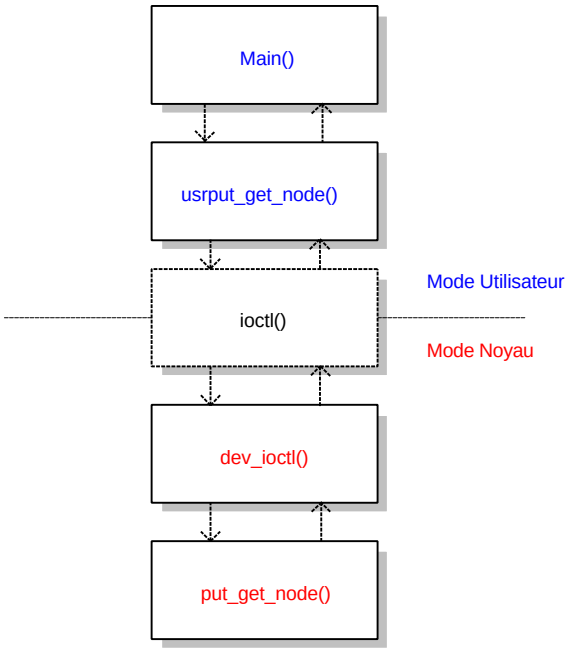

Figure 12 : cheminement d'appel de fonctions

Si nous prenons l'exemple de la figure 12 ou l'on désire connaître le numéro du nœud sur lequel on se trouve, le programme en mode utilisateur *Main()* va faire appel à la la fonction *userput_get_node()* qui va exécuter l'appel système ioctl afin continuer en mode noyau. En mode noyau la main est passée à la fonction *dev_ioctl()* du driver qui va déterminer à partir des paramètres de l'ioctl la fonction à exécuter. Une fois l'exécution terminée, on bascule en mode utilisateur avec les informations que le noyau nous a fournies. Le surcoût engendré par l'appel système a été évalué à **1.1**m**S**,
pour une couche de communication ayant une latence (temps d'un transfert) de **4**m**S** ce surcoût est gênant, c'est pour cette raison qu'en portant cette couche en mode utilisateur on supprime les appels systèmes qui se traduit par un gain en rapidité durant les phases de transfert.

B. La couche de communication PUT

Pour pouvoir réécrire cette couche de communication, il a fallut auparavant passer par un certain nombre d'étapes. Avant de commencer tout travail, la première étape a consisté à comprendre le code. Suite à cela, il a fallu définir les parties du code devant rester dans le noyau et celle que l'on peut importer en mode utilisateur. **PUT** ayant été réécrit on a du réadapter les utilitaires existants pour le tester.

B.1 Vers le mode utilisateur:

Initialement le code de **PUT** étant en mode noyau, il n'y avait aucun problème sur la répartition des fonctions, tout était exécute dans le noyau. Le passage de cette couche de communication en mode utilisateur nous a obligé à définir les fonctions devant rester dans le noyau et celles que l'on peut importer en mode utilisateur. La seule contrainte qui était imposée à cette répartition était liée aux fonctions permettant les communications.

Pour mener cette tache à bien il a fallu résoudre trois difficultés majeures :

- Le partages de ressources (**LPE**, **LMI**, carte **FastHSL**)
- Les accès au bus PCI en mode utilisateur
- Les interruptions

B.1.1 Structure du PUT en mode noyau :

La structure du **PUT** noyau était la suivante

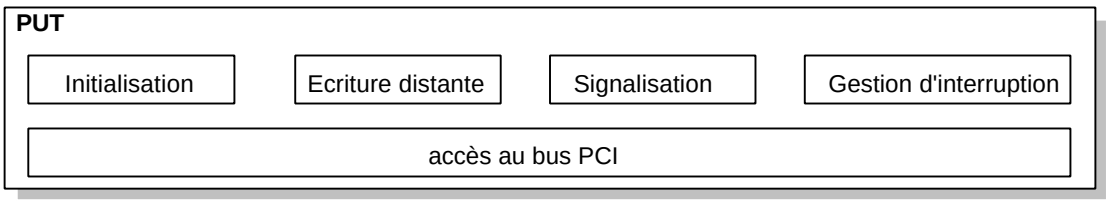

Figure 13 : Structure du put noyau

Cette couche de communication est composée de cinq modules : Initialisation, Écriture distante, Signalisation, Gestion d'interruption et Accès au bus PCI.

- L'initialisation est de deux types, matérielle et logicielle. L'initialisation matérielle va tout simplement modifier les valeurs des registres de la carte FastHSL tandis de l'initialisation logicielle est exécutée pour fournir à chaque application un **SAP** (Service access point) et une tranche de **MI** (message identifier). L'initialisation matérielle est réalisée par l'appel à la fonction *put_init()* qui prend en paramètre le nombre d'entrée de la **LPE**, le numéro du nœud sur lequel on se trouve et la table de routage qui sera chargée dans le routeur **Rcube**. Pour ce qui est de l'initialisation logicielle il faut exécuter *put_register_SAP()* pour s'enregistrer comme utilisateur de put puis faire appel a *put_attach_mi_range()* pour que **PUT** associe à ce **SAP** une tranche de **MI** pour enfin finir par *put_get_mi_start()* qui retourne la première valeur de la tranche.

- Le module d'écriture distante permet l'ajout d'une entrée dans la **LPE** et l'actualisation du registre **LPE_NEW** de la carte.

La signalisation du PUT noyau peut être de deux types, par interruption ou par scrutation. Dans tous les cas ce module permet d'un part de faire un choix entre ces deux types et d'autre part de véhiculer l'information.

- Le module de gestion d'interruption n'est utilisé que dans le cas du mode de signalisation par interruption. Il récupère les interruptions émises par **PCIDDC** et informe le module de signalisation que des données ont été reçues ou émises.

- Le dernier module "accès au bus PCI" fait les accès au bus PCI pour actualiser les registres de la carte aussi bien durant l'initialisation de lors des communications.

Il ne faut pas oublier que tous ces modules sont dans le noyau et que pour y accéder on est obligé d'utiliser un ioctl.

B.1.2 Les problèmes identifiés :

Comme nous l'avons dit précédemment afin de parvenir au portage de PUT, il a fallu auparavant résoudre trois problèmes !

Le premier d'entre eux est le partage des ressources. En mode noyau la LPE et la LMI ainsi que les registres images de ceux de la carte n'étaient accessibles que par un seul PUT, en mode utilisateur les choses sont différentes, chaque processus utilisant la nouvelle couche de communication va avoir accès à ces ressources. Or lorsqu'un processus ajoute aussi bien dans la LMI que dans la LPE des informations il ne doit pas être interrompu, il est important que l'accès soit contrôle.

Pour ce qui est des registres images de la carte ils sont utilisés pour actualiser ceux de la carte donc comme précédemment ils doivent être actualiser que par un seul processus à la fois et le changement doivent être visible par l'ensemble des processus.

Le second problème qu'il a fallu résoudre est l'accès au bus PCI. En mode noyau ces accès étaient réalisés par l'appel à des fonctions du noyau (*pci_read_config_dword()*, *pci_write_config-dword()*). Il fallait donc pouvoir effectuer ces accès en mode utilisateur.

La dernière des difficultés qui nous avons résolu est liée à la gestion des interruptions. Pour gérer les interruptions à partir d'un programme en mode utilisateur cela impliquait des changements de contexte. Le PUT utilisateur a été écrit dans l'optique d'éviter ces changements, c'est donc pour cette raison que nous les avons tout simplement supprime au profit du polling.

B.1.3 Le PUT en mode utilisateur :

La couche de communication PUT diffère de manière notable avec la couche initiale, il n'y plus comme auparavant cinq modules, mais plus que quatre.

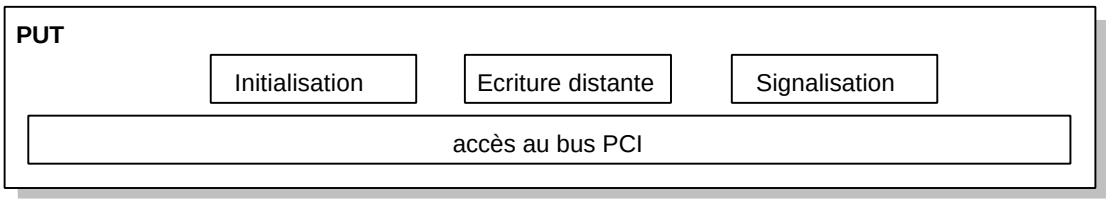

Figure 14 : Structure du put utilisateur

Le module d'initialisation a été réduit, il ne fait que l'initialisation logicielle de cette nouvelle couche de communication. L'initialisation matérielle devant fait qu'une seule fois nous l'avons incorporé dans CMEM.

Le module d'écriture distante est quant à lui rester à l'identique. Le module de signalisation tient toujours le même rôle que sont prédécesseur, avec au passage une réduction des possibilités initiales. Il ne permet d'avoir qu'une signalisation par scrutation.

Le dernier module permet quant à lui les accès au bus PCI. Pour pouvoir réaliser cette opération nous avons développé une interface.

B.2 Simplification du code

L'étape qui a succédé à la compréhension du code, fut celle de sa simplification. Les raisons qui ont imposé ce travail sont les suivantes. La première raison est liée au matériel, lors du développement logiciel de **MPCOS** et durant les phases de tests, l'équipe **ASIM** a constaté qu'un nombre de bugs matériels apparaissaient. Afin de palier à ces problèmes un certain nombre de contournement logiciel ont été mis en place, mais la première version de la carte **FastHSL** ayant fait place à une plus récente, tous ces ajouts de code n'avait plus lieu d'être, il a fallu donc le retirer. Une fois cette étape franchit avec succès, une autre simplification du code devait être faite. Cette simplification concerne la **LRM** (**L**iste des **M**essages **R**eçus) que la couche de communication **PUT** devait utiliser pour gérer les messages transitant dans le réseau lorsque celui était adaptatif. Or la **LRM** ne fonctionnant pas elle a été retirée.

B.3 PUT en mode utilisateur

Dans les pages qui vont succéder nous allons détailler le contenu des modules formant la couche PUT en mode utilisateur.

B.3.1 Accès au bus PCI :

PUT accédant aux registres de la carte **FastHSL**, aussi bien durant l'initialisation que lors des communications. Il était primordial que cette fonctionnalité puisse encore être présente sur cette nouvelle couche de communication. C'est pour cette raison que nous avons développé une couche logiciel qui permet ces accès. Mais avant d'en expliquer son fonctionnement, nous allons auparavant voir le principe général.

L'accès au bus **PCI** n'est pas simple. Tout processus utilisateur n'a pas systématiquement accès au entrée\sortie, il faut au préalable lui en donner les droits. Ce privilège est de plus seulement réserve aux processus appartenant à l'utilisateur root. La fonction C permettant cette opération est la fonction *iopl()*, elle prend en paramètre le niveau de priorité désiré sachant que le niveau maximum est de trois et que par défaut il est à zéro. Dans notre cas nous avons imposé cette valeur à son maximum. Pour plus d'information sur cette fonction voir l'annexe 1.

L'autre aspect qu'il faut souligner sur ces accès, c'est que le processus n'accède pas directement aux cartes **PCI**. Le processus s'adresser au **chipset** de la carte mère en lui fournissant des informations sur les registres de la carte qu'il veut lire.

La procédure mise en œuvre suit le protocole suivant:

- le processus fournit au chipset une adresse avec un format spécifique sur la carte qu'il veut lire
- en retour de cette information le chipset lui fournit l'adresse ou se trouve ce registre.

- une fois cette information acquise, le processus n'a plus qu'à faire soit *inl()* pour lire le registre soit un *oul()* pour l'écrire. Bien évidemment ces deux fonctions prennent en paramètre une adresse.

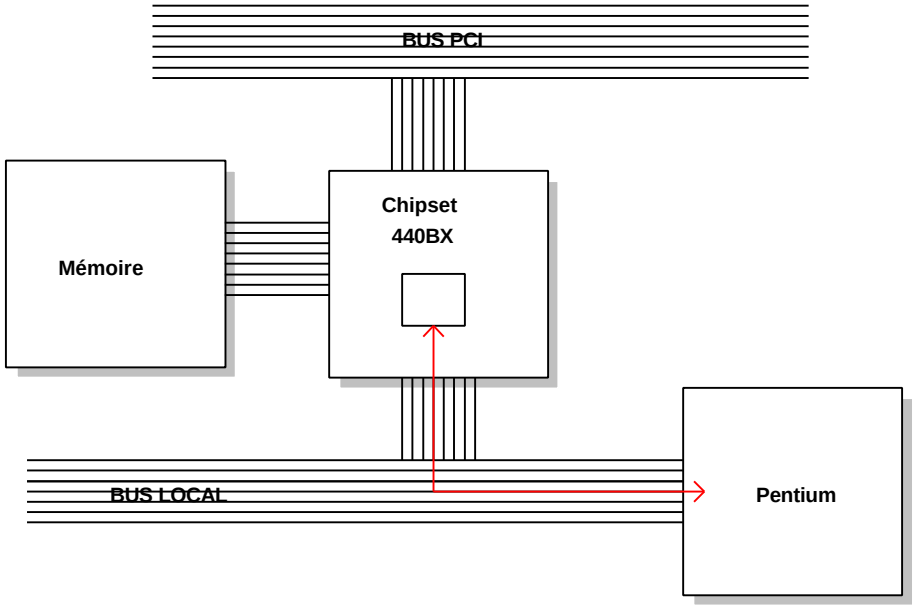

Figure 15 : Dialogue processus carte PCI

L'interface réalisée pour accéder au bus est le fichier **pci.c** avec le header associé **pci.h**. Ce programme dispose de cinq catégories de fonctions :
l'accès au bus, la lecture/écriture du bus, la construction de l'adresse de la carte,d'une fonction d'accès au entrée/sortie et enfin d'une fonction pour identifier l'adresse à laquelle notre carte se trouve.

Comme nous l'avons dit précédemment il faut donner les droits au processus pour accéder au bus d'entrée/sortie. Cette fonctionnalité est implémentée par la fonction *int PCI_acces()* qui ne prend aucun paramètre et retourne zéro en cas de succès.
Chaque carte PCI est identifiée par une adresse avec un format bien particulier, cette adresse a l'aspect suivant figure 14.

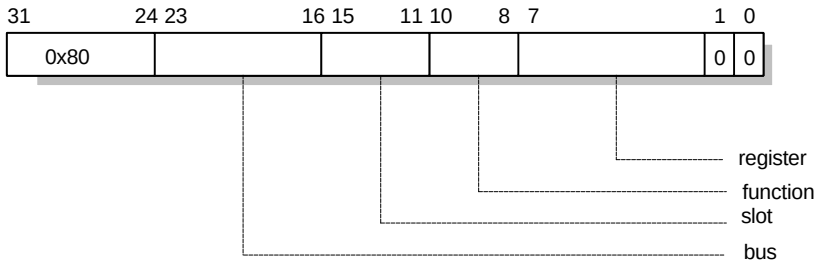

Figure 16 : format d'une adresse

L'adresse est composée d'une partie appelée **register** qui permet d'identifier le registre à lire ou à écrire. Notre carte étant de type **PCI** la valeur du paramètre **function** est constante et égale à zéro, dans le cas des carte **AGP** cette valeur est égale à un. Les deux champs restants **slot** et **bus** permettent d'identifier physiquement la carte désirée, la valeur de **bus** est comprise entre 0x00 et 0x3F tandis que celle du champ **slot** est comprise entre 0x00 et 0xFF.

Pour construire cette adresse nous disposons de la fonction *unsigned int PCI_card(bus, slot, function)*.

Une fois cette adresse construite les deux fonctions *u_char PCI_read_dword()* *void PCI_write_dwod()* nous permettent de lire et écrire les registres de la carte, les paramètres qui leur sont passés sont l'adresse de la carte et le registre désiré.

Pour finir la fonction *int PCI_find()* est utilisée pour trouver l'adresse de notre carte **FastHSL**, cette recherche est effectuée en lisant les registres **vendorID** et **deviceID** qui dans notre cas sont tous les deux égaux à 0x0000.

```
void hsl_conf_write(int minor, u_char reg, u_int val){
        if(put_info[0].hsl_found==FALSE)
                retrun;
        PCI_write_dword(put_info[0].hsl_tag_bis,reg, val)
}
```

Utilisation de l'interface dans PUT utilisateur

B.3.2 L'initialisation

Dans le module d'initialisation nous avions deux parties, l'une faisait l'initialisation matérielle de la carte **HSL** et l'autre l'initialisation logicielle, la première partie ayant été intégré au driver **CMEM** il ne restait plus que l'initialisation logicielle. Cette initialisation avait pour tache l'attribution d'un **SAP** (**S**ervice **Access P**oint) et d'une tranche de **MI** dont la longueur désirée était fournit par l'application. L'utilité de cette opération était de fournir à différentes applications des points d'entrée à **PUT**, à titre d'exemple il était possible à deux applications telles que **PVM** et **MPI** d'utiliser **PUT** simultanément. Pour des raisons de gain en performance la seule application utilisée au dessus de **PUT** n'était que **MPI**, il nous a parut plus simple de ne définir qu'un seul **SAP** de lui associé la totalité des **MI** possibles et de déléguer la gestion des **MI** aux applications utilisant **PUT**.

L'autre raison qui nous a poussée a faire ce choix est directement liée à la manière avec laquelle l'appel aux call_back était fait. Aussi bien dans le code de *put_flush_lpe()* que dans celui de *put_flush_lmi()* l'appel de la call_back était précède d'une vérification du **MI** pour savoir celles qu'il fallait activer. Une fois cette vérification faite la call_back se trouvant dans le noyau était activée pour une tranche **MI**, or dans notre cas **PUT** étant en mode utilisateur chaque processus utilisant **PUT** dispose de sa propre call_back qu´íl est le seul à activer. Une telle procédure de notification n'est plus applicable, le choix qui a été fait est de n'avoir d'un seul SAP et d'attribuer à l'application MPI la totalité des **MI**. La gestion étant faite dans les call-backs des taches que l'application MPI a lancé.

L'initialisation matérielle de la carte est faite en mode noyau, hors durant cette phase d'initialisation un certain nombre de paramètres décrivant l'état de la carte sont initialisés, il faut donc avoir accès à ces informations. En plus de ces informations il faut permettre à cette nouvelle couche de communication d'accéder au slot **PCIDDC** de **CMEM**, au mode noyau ce problème ne se posait pas, il suffisait à **PUT** de faire appel à *cmem_virtaddr()* pour avoir l'adresse virtuelle de ce slot et par conséquent y accéder directement. Dans le cas de **PUT** en mode utilisateur les choses sont toutes autres, il faut d'une part avoir cette adresse virtuelle en mode noyau et enfin la mapper en mode utilisateur. Pour réaliser ces deux taches nous avons pour cela défini deux fonctions *put_activate()* et *put_mmap_PCIDDC_slot()*. La fonction *put_activate()* va dans un premier temps faire une copie de la structure put_info contenant toutes les informations sur la carte après l'initialisation, la copie se fait à travers un *ioctl()* du noyau vers le mode utilisateur. Une fois cette opération réalisée avec succès, *put_activate()* récupère les informations sur le slot **PCIDDC** par un appel à un autre *ioctl()*. Toutes les informations ayant été rassemblées *put_activate()* fait appel à *put_mmap_PCIDDC_slot()*, à laquelle elle fournit une structure de type *opt_contig_mem_t* contenant toutes les informations sur ce slot.

```
typedef struct _opt_contig_mem_t {
        u_char *ptr;     // adresse virtuelle du slot
        u_long virt;     // adresse physique du slot
        size_t size;     // taille du slot
}opt_contig_mem_t;
```

Cette fonction va donc tout simplement faire appel à la fonction C *mmap()* qui va permettre de mapper ce slot dans l'espace d'adressage du processus et donc pouvoir y accéder en mode utilisateur.

B.3.3 La signalisation:

Dans notre cas pour des raisons de gain en performance le modèle de signalisation par scrutation a été choisi, nous disposons pour cela de deux fonctions nous permettant de nous garantir la fin d'émission ou celle de la réception.

- La fonction de signalisation de fin d'émission est invoquée à travers *put_flush_lpe()* lorsqu'un message est complètement émis. Pour réaliser cette opération, elle lit les registres images de ceux de la carte **FastHSL** et compare leur valeur (**LPE_NEW** et **LPE_CURRENT**).

Si les valeurs diffères cela implique qu'une entrée a été ajoutée dans la **LPE**, la fonction va donc appeler la call_back de l'application afin de lui notifier l'émission des données. Un nombre de données sont passées en paramètre à cette call_back notamment le **MI** (**M**essage **I**dentifier) ainsi que le contenu de l'entrée qui a été émise.

- En réception : lorsque la réception d'un message est achevée *put_flush_lmi()* invoque la call_back afin de notifier à l'application la réception des données lui étant destinées. Le paramètre qui lui est fourni est le **MI**.

B.3.4 L'émission:

La seule modification qui a été apportée à ce module est l'ajout de la fonction *put_add_several_entries()*. Les paramètres qu'on lui passe sont le pointeur sur la premier entrée à envoyer et le nombre d'entrées. Une fois ces informations acquises cette fonction va tout simplement faire appel à *put_add_entry()*.

B.3.5 Le partage des ressources:

On est dans le cas d'un système ou plusieurs processus peuvent accéder aux même variables (les registres de la carte). Il est donc primordial de n'autoriser qu'un accès à la fois à ces registres, c'est pour cette raison que nous avons défini quatre verrous, **LPE_lock**, **LPE_flush_lock**, **LMI_flush_lock** et enfin **HSL_lock**. Le problème sur le choix de la méthode à utiliser pour manipuler ces verrous se posait, utiliser les opérations classiques d'incrémentation et de décrémentation pour matérialiser l'état libre ou pris ne suffisait pas. Lorsque plusieurs processus sont lancés au sein d'une même machine, chacun d'eux est exécuté pendant un laps de temps défini par le système puis swapé pour passer à l'exécution d'un autre. Or les opérations d'incrémentation et de décrémentation n'étant pas atomiques on peut se trouver dans le cas ou deux processus à la fois prennent le verrou (figure17).

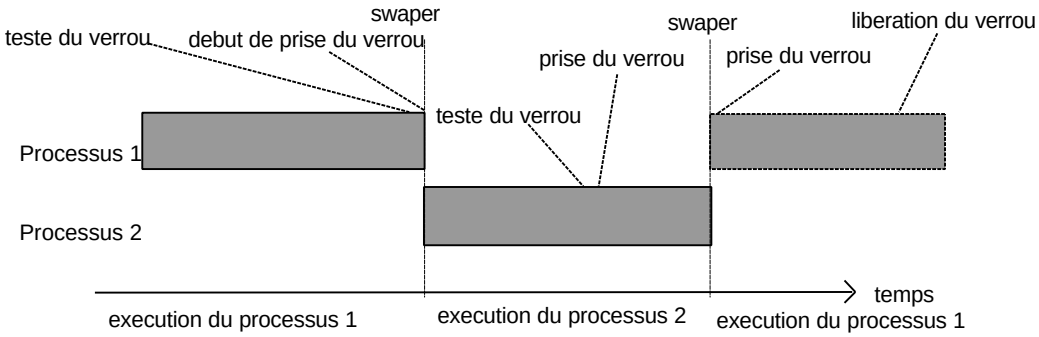

Figure 17 : exécution de deux processus

Comme le montre la figure 17 avec deux processus, une fois que le premier processus s'est assuré que le verrou était libre et qu'il s'apprête à le prendre par une décrémentation il peut arriver qu'au moment de réaliser cette opération il soit swapé au profit du deuxième processus qui lui va comme le premier constater que le verrou est libre et va le prendre. Or jusqu'ici aucun problème n'apparaît, il ne se manifestera qu'au moment ou le deuxième processus ayant toujours le verrou est swapé On revient dans ce cas à l'exécution du premier qui s'était avant d'être swaper assuré que le verrou était libre, et va donc poursuivre sont exécution et finir la prise du verrou.

On se trouve dans le cas ou le verrou est pris par deux processus à la fois et donc chacun d'eux est libre d'actualiser à sa guise les registres de la carte.

Pour éviter qu'un tel problème puisse apparaître nous avons utilisé des opérations atomiques pour lesquelles nous sommes garantis que le processus n'est pas swapé au moment de leur exécution (atomique = 1 instruction processeur). Les fonctions que nous avons utilisées pour vérifier que le verrou était libre et pour le prendre sont :

- *atomic_read()*
- *atomic_dec()*
- *atomic_set()*

atomic_read() comme son nom nous l'indique va nous permettre de vérifier si le verrou est libre avec la convention suivante:

1 indique que le verrou est libre
0 indique que le verrou est pris

atomic_dec() cette fonction va décrémenter la valeur du verrou pour le prendre tandis que la fonction *atomic_set()* va initialiser le verrou à la valeur qu'on lui fournit en paramètre.

Lors de la mise en place de ces verrous nous n'en avions défini que trois, le verrou **HSL_lock** a été rajouté à la suite d'un problème qui se manifestait systématiquement lorsque nous lancions deux processus utilisant **PUT**. Ce problème se traduisait par des valeurs hasardeuses aux moments ou l'on accédait aux registres de la carte. Les raisons de ce problème sont simples, les trois verrous que nous avons définis **LPE_lock**, **LPE_flush_lock** et enfin **LMI_flush_lock** permettent respectivement d'avoir qu'un accès à la fois à l'ajout d'une entrée, à la notification d'un envoi et enfin à la notification d'une réception. Cela n'empêchait pas d'avoir un processus qui notifiait l'émission alors que l'autre faisait celle de la réception, Or les opérations utilisées pour accéder à la carte ne sont pas atomiques, il suffisait que l'un d'eux ait pris le bus **PCI** et qu'il soit swapé juste après pour que le deuxième ait des valeurs incohérentes au moment de la lecture, c'est donc pour cette raison que nous l'avons ajouté.

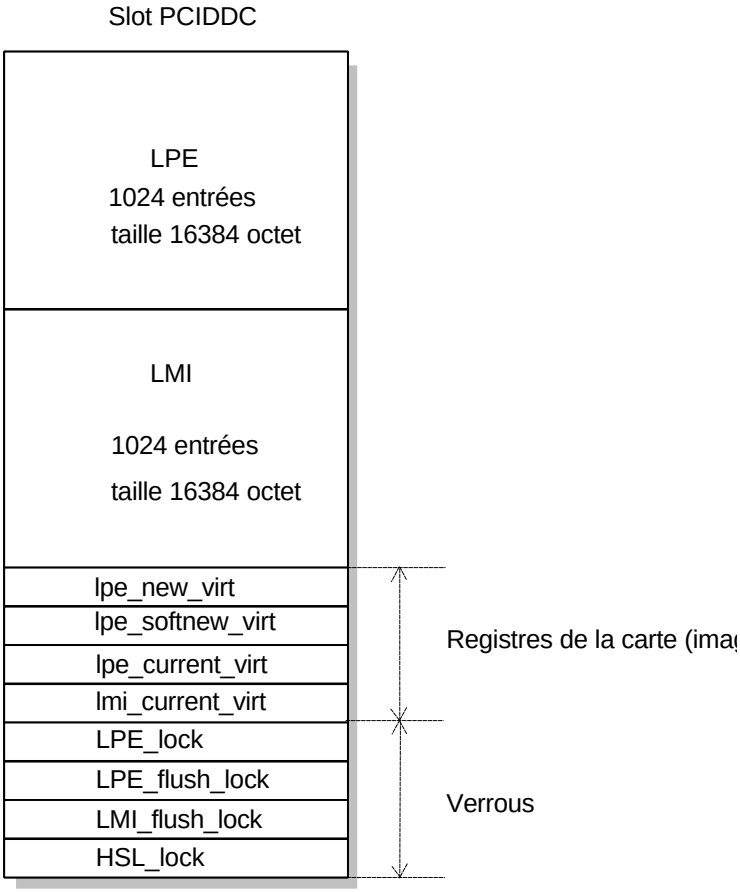

Figure 18 : Détail du contenu du slot PCIDDC

Nom du verrou	Variables protègées	Fonction(s) utilisatrice(s)	opération lorsque le verrou est pris
LPE_lock	la lpe	put_add_several_entrie	attente active, la boucle jusqu'a sa
LPE_flush_lock	pointeurs de la lpe	put_flush_lpe	si le verou est pris on de la fonction sans code de
LMI_flush_lock	pointeurd de la lmi	put_flush_lmi	si le verou est pris on de la fonction sans code de
HSL_lock	l'accès a la carte hsl	hsl_conf_read(),	si le verrou est pris on jusqu'a sa

Figure 19 : Tableau récapitulatif des verrous

Nous avons procédé à une phase de test des verrous à travers le programme ci-dessous. Ce programme implémente le classique concept du producteur et des consommateurs. Dans le cas de l'utilisation des verrous avec les fonctions atomiques, nous n'avons à aucun moment pu mettre le programme en défaut contrairement au cas avec l'incrémentation et la décrémentation pour lequel il arrivait que deux processus s'accaparent en même temps une variable devant être initialisée que par un processus à la fois.

Producteur vs Consommateur

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h >
#include <asm/atomic.h>

#define TRUE        1
#define FALSE   0
#define ATOMIC_OPERATION

char *prod = "Producteur";
char *cons = "Consommateur";
char *error_message = "impossible de creer un thread";
int num_thread = 1;
int stock = 0;
atomic_t lock;
int verrou = 1;
int num_pthread_conso = 50;

// *********************
// *        Fonction exécutée par le producteur
// *********************
void *producteur ( void *arg){
        while(TRUE)
        {
#ifdef ATOMIC_OPERATION
                if(atomic_dec_and_test(&lock))
                {
#endif
#ifndef ATOMIC_OPERATION
                if(verrou)
                {
                        verrou = 0;
#endif
                        if(!stock)
                        {
                                printf("<Prood> rupture de stock\n");
                                stock = (int) random() % 1000;
                                while(stock > 1000 || stock == 0)
                                        stock = (int) random();
                                printf("<prod>la quantite %d\n",stock\n");
                        }
#ifdef ATOMIC_OPERATION
                        atomic_set(&lock,2);
```

```
#endif
        #ifndef ATOMIC_OPERATION
                                verrou =1;
        #endif
                  }
                return NULL;
        }

        // ********************
        // * Fonction exécutée par les consommateurs
        // ********************
        void  *consommateur( void *arg){
                int num_conso;
                num_conso = num_thread;

                while(TRUE)
                {
        #ifdef ATOMIC_OPERATION
                        if(atomic_dec__and_test(&lock))
                        {
        #endif
        #ifndef ATOMIC_OPERATION
                        if(verrou)
                        {
                                verrou = verrou -1;
        #endif
                                if(stock)
                                {
                                        printf(<Conso %d> stock  de %d\n"num_conso,stock);
                                        stock = 0;
                                        sleep(1);
                                }
        #ifdef ATOMIC_OPERATION
                                atomic_set(&lock,2);
                        }
        #endif
        #ifndef ATOMIC_OPERATION
                                verrou = verrou + 1;
                        }
        #endif
                   }
                return NULL;
        }
// ******************
// * Fonction Main
// ******************
int main(int argc, char **argv[ ]){
        pthread_t thread;
        int i;
        lock.counter = 2;
        if(pthread_create(&thread,NULL,producteur,(void *) prod)
        {
                write(1,error_message,strlen(error_message));
                exit(-1);
        }
```

```
for(i=0; i<num_pthread_conso;i++){
                if(pthread_create(&thread,NULL,consommateur,(void *) cons)){
                        write(1,error_message,strlen(error_message));
                        exit(-1);
                }
        }
        pthread_exit(NULL);
        return 0;
}
```

Remarque :

L'autre possibilité qui nous était donnée pour mettre en place des verrous était d'utiliser des sémaphores. Le problème qui reste récurant est d'avoir une couche performante or l'utilisation de sémaphore n'aurait fait que la ralentir, c'est pour cette raison que nous avons préféré utiliser les fonctions atomiques

B.4 Les utilitaires :

La couche de communication ayant été réécris, il a fallu la tester afin de s'assurer de son bon fonctionnement. Il existait dans la version initiale de **MPC-OS** un certain nombre de programmes permettant de réaliser tous les tests nécessaires.

B.4.1 Testputbench :

Le premier utilitaire à adapter était **testputbench.c** qui implémentait un ping-pong entre deux nœuds. Initialement pour réaliser ce ping-pong le nœud en phase d'émission utilisait un *iotl()* pour faire un ajout dans la **LPE** auquel il passait une structure de type lpe_entry_t qui contenait les informations suivantes.

```
typedef struct _lpe_entry {
        unsigned short page_length;   // longueur des données
        unsigned short routing_part;  // nœud destinataire
        unsigned long control;// informations de contrôle
        caddr_t PRSA,PLSA;            // adresse physique du tampon
                                      // d'émission et de réception
} lpe_entry_t;
```

Tandis que de l'autre coté le processus récepteur après un *ioctl()* qui le met en veille attendait les données.
Une fois les données réceptionnées, le processus faisait une vérification afin de s'assurer de leur validité pour ensuite passer à une phase d'émission. Contrairement à sa version initiale le programme actuel accède directement aux fonctions d'émission et de notification d'émission et de réception **PUT**.

Le lancement de cet utilitaire se fait de la manière suivante:

./testputbench -c -i 12 -o 8 -M 166 -s 4096 -l 10000 -n 1 1 -a 0x7800000 -e 0

- l'option -c oblige **testputbench** a effectué une vérification des données reçues
- les options -i 12 et -o 8 permettent de définir deux zones encadrant le tampon d'émission de huit octets initialisés à 12.
- pour s'assurer de mesures correctes l'option -M suivit de la fréquence du processeur est nécessaire.
- la carte **FastHSL** pouvant émettre des tampons de taille variables, -s fournit cette information à la couche de communication.
- le programme testputbench implémentant un ping-pong il faut lui définir la nombre d'échanges, il suffit pour cela d'utiliser l'option -l suivit de la valeur.
- -n 1 identifie le nœud destinataire et -a 0x780000 l'adresse physique du tampon de réception.
- les deux dernière options 1 et -e 0 permettent respectivement de définir quel programme va initier l'échange et la valeur du **MI** qui est attribué aux messages.

Remarque :
Contrairement à sa version initiale, le programme actuel a été débuggé afin d'autoriser des transferts de tampon de taille supérieure à 64Koctest.

B.4.2 La gestion des MI :

Comme nous l'avons dit précédemment la gestion des **MI** se fait au niveau des applications utilisant **PUT**. Dans le cas de **testputbench** il suffit de lui attribuer un **MI** dont la valeur lui est passée sur la ligne de commande. La gestion de ces **MI** est faite par les call_back de cette application, aux moments des émissions et de réceptions les call_back vont dans un premier temps vérifier si la valeur du **MI** correspond à celle de l'application pour ensuite notifier l'émission ou la réception. Dans le cas ou cette valeur ne correspond pas, les call_back libère le verrou **LMI_flush_lock** ou **LPE_flush_lock** puis fait appel à la fonction u*sleep()* pour passer la main à l'application pour laquelle cette information est destinée.

B.4.3 Testputsend_loop :

PUT étant destinée à être utilisé par plusieurs processus à la fois il fallait s'assurer de cette possibilité durant la phase de test. C'est dans cette optique que le programme **testputsend_loop.c** a été réécrit, son rôle est de créer du trafic sans que les informations sur les données ne soient mémorisées dans la **LMI**. L'autre intérêt de ce programme est de vérifier que la couche de communication **PUT** n'introduit pas de pertes ou d'erreurs sur les données échangées.

Pour réaliser ce test nous avons dans un premier temps lancer un ping-pong entre deux machines puis sur l'une d'elles **testputsend_loop.c** afin de vérifier que dans les cas de surcharge **PCIDDC** se comporte bien.

Le lancement de ce programme se fait de la manière suivante:

./testputsend_loop -s 4096 -L 0x7800000 -l 10000 -n 1 -M 166

B.4.4 Les démons :

Sur la figure 8 nous avons représenté deux entités qui sont **HSLclient** et **HSLserveur** qui permettent autant dans leur version initiale que dans l'actuelle l'initialisation de la carte et notamment l'appel à la fonction *put_init()* à travers un *ioctl()*. Ces deux programmes ne permettent plus d'émuler l'échange des messages à travers le réseau de contrôle.

B.4.5 Visualisation de la LPE et la LMI :

Tous les programmes nécessaires pour utiliser et initialiser **PUT** ayant été réécrit et vérifié, il fallait maintenant s'assurer qu'aux moments des échanges la **LPE** et la **LMI** étaient correctement initialisées. Pour effectuer cette vérification nous avons du réécrire **dumpPCIDDC.c** dont le rôle est la scrutation et l'affichage des informations contenu dans ces deux listes. Pour y accéder ce programme fait appel à un *ioctl()* afin d'avoir l'adresse du slot **PCIDDC** que le driver **CMEM** lui fournit. En plus du contenu de ces deux listes il affiche la position des pointeurs de la **LPE** et de la **LMI** par une lecture directe qu'il effectue à travers l'interface d'accès au bus **PCI**. Cette utilitaire a aussi été enrichi de quelques autres fonctions notamment pour afficher la valeur des pointeurs images des registres de la carte ainsi que l'état des verrous.

Les derniers utilitaires **regview.c, putslotview.c** ont été écrits essentiellement pour nous aider à débuguer les utilitaires ci-dessus. **regview.c** permet d'afficher le contenu des registres de la carte tandis que **putslotview.c** celui du slot **PUT**, **slotaddress** qu'en a lui fournit l'adresse du slot d'émission nécessaire à **testputbench.c**. Il suffit pour d'ajouter l'option s suivit du numéro du slot. L'option -s 0 correspond au premier tampon tandis que -s 1 au deuxième.

B.4.6 Utilisation de PUT utilisateur :

Contrairement à son homologue en mode noyau l'utilisation du nouveau **PUT** nécessite une phase d'initialisation. Elle débute par l'appel à la fonction *put_activate()*, suivit de l'appel à toutes les fonctions permettant d'une part de réserver un **SAP** et d'autre part une tranche de **MI**.

Il faut débuter par l'appel à la fonction *put_register_SAP(int,void*,void*)*, elle prend en paramètre un entier " minor" toujours égal à zéro et les fonctions de call_back en émission et en réception. En retour cette fonction fournit un numéro de **SAP**. En as de succès la procedure d'initialisation se poursuit en appelant *put_attach_mi_range(int minor,int numberlayer,int size)* qui va se charger de réserver une tranche de **MI** au **SAP** correspond. Comme nous l'avons dit précédemment quelque soit la taille fournit on a automatiquement l'ensemble des **MI**. La dernière fonction que l'on peur éventuellement exécuté est *put_get_mi_start(int minor,int numberlayer)* qui retourne le premier **MI** de la tranche.

Procedure d'utilisation de put en mode utilisateur

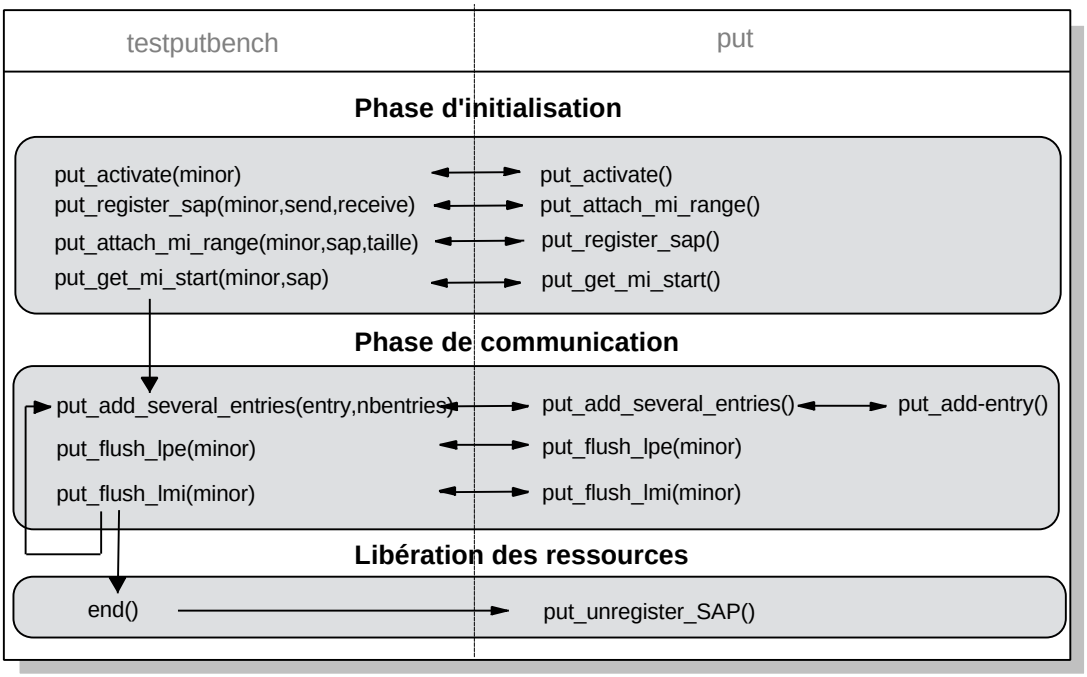

Figure 20 : utilisation de put utilisateur

Chapitre 5

A. Les performances de PUT

A.1 Le scheduling
A.2 Les politiques d'ordonnancement
A.3 Les fonctions d'ordonnancement

B. Les mesures

B.1 Conditions expérimentales
B.2 Analyse des mesures

C. Les problèmes rencontrés

D. Conclusion

D.1 Les Objectifs
D.2 Le déroulement du stage
D.3 Le futur de MPC

A. Les performances de PUT

Comme son titre l'indique ce dernier chapitre sera consacré à l'évaluation des performances de **PUT**. Mais avant d'en fournir les résultats nous allons au préalable faire un point sur les problèmes que nous avons du résoudre. Contrairement à son homologue en mode noyau, le **PUT** utilisateur n'échappe aux règles de scheduling du système d'exploitation. Le programme **testputbench** utilisant notre nouvelle couche de communication avait une priorité d'exécution inférieure à celle des démons s'exécutant. Or lors de son lancement il était souvent swaper au profit de ces derniers. Nous avons du donc modifier le niveau de priorité du processus puis la règle de scheduling afin de se rapprocher des conditions d'exécution du **PUT** noyau. Afin de comprendre au mieux les choix qui ont été faits nous allons débuter par un bref rappel des concepts de scheduling sur les systèmes de type **UNIX**.

A.1 Le scheduling :

Sur une machine monoprocesseur, un processus à la fois peut s'exécuter. Le choix du processus qui doit monopoliser la **CPU** est effectué par le scheduler. Le choix est basé sur le niveau de priorité du processus, celui dont la priorité sera la plus élevée sera exécuté en premier. Par conséquent tout processus qui veut s'exécuter va devoir s'enregistrer dans la base de données du scheduler qui dispose de 64 niveaux.

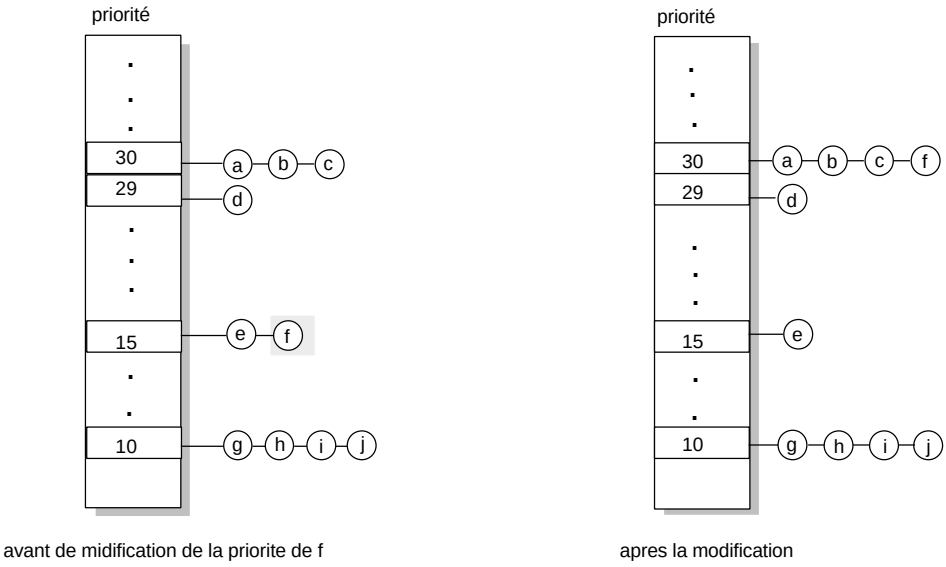

Figure 21: ordre d'exécution de processus

Le niveau de priorité d'un processus ne suffit pas a définir celui qui sera exécuté, il faut lui imposer une politique d'ordonnancement parmi les trois existantes.

A.2 Les politiques d'ordonnancement :

La politique d'ordonnancement permet une grande flexibilité dans le gestion des processus. Nous disposons à cet effet de trois politiques:

- time-sharing :

Mode d'exécution par défaut de l'ensemble des processus, le système attribut à chacun un quantum de temps durant lequel il s'exécutera.

- first-in-first-out :

Dans ce mode, le processus ayant pris la main la garde jusqu'a ce qu'il se termine ou qu'il cède délibérément la main.

- round-robin :

Le round-robin est un mélange du time-sharing et du first-in-first-out. L'ensemble de processus ayant la même priorité et dont la valeur est la plus importante seront exécutés jusqu'à ce qu'ils se terminent en leurs appliquant le time-sharing. Une fois cet ensemble vide le scheduler passe à l'ensemble de niveau de priorité juste inférieur.

A.3 Les fonctions d'ordonnancement :

La fonction *nice()* nous a permis de modifier le niveau d'exécution de **testputbench**. Mais afin d'avoir les performances réelles de **PUT**, il fallait se mettre dans les meilleures conditions, arrêter le maximum de démons afin de s'assurer que le programme n'était que rarement swapé. Or avec l'utilisation de cette fonction les résultats des mesures n'étaient pas constants, on a donc opté pour l'ensemble des fonctions qui nous ont permis d'exécuter notre programme comme une tache réelle (voir annexe 2). Le code qui a été ajouté à celui de **testputbench** est le suivant:

```
#ifnde PRIORITY_MAX
        nice(-20);
#endif
#ifdef PRIORITY_MAX
        schedparam.sched_priorrity = 99;
        res = sched _setc=scheduler(0,SCHED_FIFO,&schedparam);
        CHECK(res, "sched_setscheduler");
#endif
```

B. Les mesures

B.1 Conditions expérimentales

Tous le tests de performances ont été faits sur les mêmes machines: deux PC identiques équipés de Pentium 166, 128 Mo de mémoire et d'une carte mère HX.

Pour effectuer ces tests, le programme **testputbench** a été utilisé. Il fait appel aux fonctions du programme **timer.c** développé par Philippe Lalevée de l'**INT**. Ce programme lit directement le temps qui s'est écoulé entre deux mesures dans le registre de processeur.

Pour le PUT en mode utilisateur le point de référence des mesures a été mis juste avant de procéder à l'appel de la fonction *put_add_entry()*. Les temps que nous avons mesuré correspondent à l'ajout d'une entrée est suivie de la notification en émission et en réception. Dans le cas du PUT noyau le point de référence se situe avant l'appel système *ioctl()* qui permet l'ajout d'une entrée tandis que la fin de la mesure est juste après le retour de l'appel système *ioctl()* qui notifie la réception (Figure 22).

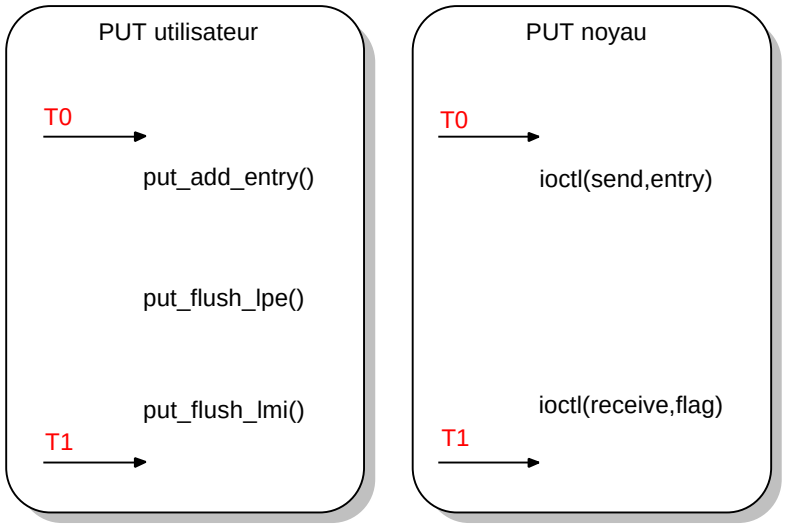

Figure 22: procédure des mesures

LES COURBES

B.2 Analyse des mesures

A travers ces deux courbes on constate un gain significatif en rapidité du **PUT** en mode utilisateur. L'amélioration se situe surtout pour les petits paquets, à titre d'exemple pour une taille de 100 octets le débit de cette nouvelle couche est plus de deux fois supérieur à l'ancienne. Pour des paquets de plus grande taille (supérieure à 16384 octets) le débit des deux couches se rejoignent, avec une légère différence en faveur de notre couche, qui atteint un débit maximum de 485 Mbits/s pour une taille de 65535 contre 476 Mbits/s pour la couche noyau.

Sur les courbes on distingue deux discontinuités situées respectivement à 540 et 1080 octets. Lorsqu'une page est suffisamment petite elle est constituée que d'un seul paquet, mais si l'on ajoute progressivement des octets de données, on arrive à un moment ou **PCIDDC** prendra la décision de clore le paquet en cours et d'en initier un second pour terminer la page. L'ajout de 27 octets d'en tête du second paquet fait chuter le débit.

Afin de comparer au mieux ces deux couches, nous avons affiné nos mesures en décomposant chaque étape du send-receive. Les résultats de ces mesures ont été regroupés dans la figure 23.

PUT utilisateur		PUT noyau	
Fonction	temps microseconde	Fonction	temps microseconde
put_add_entry()	4,4	put_add_entry()	4,1
put_flush_lpe()	12	call_back d'émmision	16,6
put_flush_lmi()	15,57	call_back de reception	41,5

les 15,57 et 41,5 microsecondes correspondent au temps d'un send suivi d'un receive le tout divisé par 2

Figure 23 : découpage des mesures

Le gain de cette nouvelle couche se situe essentiellement durant les phases de notification que se soit en émission qu'en réception.

Sous MPI les résultats obtenus sont :
21 micros-secondes de latence pour le put utilisateur contre 29 micros secondes pour le put noyau.

C. Les problèmes rencontrés

Le seul problème que nous avons rencontré et qui reste insolvable, apparaît durant les phases de lecture des registres de le carte **FastHSL** lorsque deux **testputbench** sont lancés ou bien lorsqu'on exécute **testputsend_loop**. Initialement cette bizarrerie " lecture de la valeur **0xFFFFFFFF** " avait été rencontrée que sur des cartes mères de type **BX** et un contournement logiciel avait été mis en place. Il consistait en une boucle de lecture qui s'assurait que cette valeur n'apparaissait pas. Nos PC étant équipés de carte mère de type **HX** ce problème n'avait jamais été rencontré. Nos soupçons se sont donc portés sur les verrous, mais après des phases de tests aucune erreur n'a été trouvée. Une autre démarche qui serait intéressante de mettre en oeuvre serait de tester **PUT** sur des machines avec des chipset différents.

D. Conclusion

D.1 Rappel des objectifs

L'objectif de ce stage était le portage de la couche de communication noyau existante en mode utilisateur dans le but de gagner en performance lors du transfert des messages. Cet objectif a été atteint et les tests que nous avons réalisé avec les utilitaires précédemment cités le confirmes.

D.2 Déroulement du stage

Le premier mois de ce stage a été consacré exclusivement a l'installation de Linux et du package **MPC-OS** sur nos machines. Les problèmes que nous avons résolus sont la détection et la prise en charge de nos cartes réseau par Linux, et de la compilation du noyau afin que **MPC-OS** puise disposer de tous les headers dont il a besoin. Une fois cette étape franchit nous nous sommes fortement investit dans la programmation noyau et sur la compréhension du fonctionnement de **PUT**. Chaques fonctions composant **PUT** ayant été identifié ainsi que leur rôle, nous nous attelé à sa réécriture ainsi qu'à celle de tous les utilitaires dont nous avions besoin pour valider son fonctionnement. Le portage étant fait, il a fallu s'assurer que cette nouvelle couche apportait des améliorations, ce que nous avons fait en procédant aux mesures de performances aussi bien à partir des utilitaires qu'au-dessus de **MPI**.

D.3 Le futur de MPC

Les performances qu'apporte le nouveau **PUT** sont très attrayantes. On est passé d'une latence de 29 micros secondes à celle 21 micros secondes sous **MPI**, en incorporant l'ensemble du travail qui a été fait sur le réordonnancement de la mémoire des processus utilisant **PUT** on peut espérer descendre sous les 20 micros secondes. L'élément le plus réconfortant sur l'avenir de MPC et qui ressortit des réunions auxquelles nous avons assister est le désir de l'ensemble de l'équipe de poursuivre son développement aussi bien logiciel que matériel.

Bibliographie

Sites Internet :

http://www.linuxdoc.org/LPD/tlk/tlk.html

http://www_rp.lip6.fr/~deleuze/§HOWTOS/kernel.html

http://www.guide.asso.fr/miroirs/LPD/HOWTO/mini/IO-Port-Programming-2.html

http://bat710.univ-lyon1.fr/~bonnev/DocLinux/khg/HyperNews/get/devices/reference.html

http://mpc.lip6.fr/

Ouvrages :

La programmation sous Unix
par Jean-Marie Rifflet
édition : Ediscience international

Programmation Linux 2.0
API système et fonctionnement du noyau
par Rémy Card, Éric Dumas et Frank Mével
édition : Eyrolles

Understanding the linux kernel
par Daniel P.Bovet et Marco Cesati
édition : O'Reilly

LINUX device drivers
par Alessandro Rubini
édition : O'Reilly

Linux Man Page for IOPL (2)IOPL (2)
change I/O privilege level

SYNOPSIS

#include <unistd.h> /* for libc5 */ #include <sys/io.h> /* for glibc */ int
iopl(int level);

DESCRIPTION

iopl changes the I/O privilege level of the current process, as specified in level . This call is necessary to allow 8514-compatible X servers to run under Linux. Since these X servers require access to all 65536 I/O ports, the ioperm call is not sufficient. In addition to granting unrestricted I/O port access, running at a higher I/O
privilege level also allows the process to disable interrupts. This will probably crash the system, and is not recommended. Permissions are inherited by fork and exec. The I/O privilege level for a normal process is 0.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

ERRORS

EINVAL
level is greater than 3.
EPERM
The current user is not the super-user.

NOTES FROM THE KERNEL SOURCE

iopl has to be used when you want to access the I/O ports beyond the 0x3ff range: to get the full 65536 ports bitmapped you'd need 8kB of bitmaps/process, which is a bit excessive.

CONFORMING TO

iopl is Linux specific and should not be used in processes intended to be portable.

NOTES

Libc5 treats it as a system call and has a prototype in <unistd.h> . Glibcl does not have a prototype. Glibc2 has a prototype both in <sys/io.h> and in <sys/perm.h> . Avoid the latter, it is available on i386 only.

SEE ALSO

- ioperm (2) -

sched_setscheduler(3R) manual pageTable of Contents

Name

sched_setscheduler - set scheduling policy and scheduling parameters

Synopsis

cc [flag...] file... -lrt [library...]
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);

Description

The sched_setscheduler() function sets the scheduling policy and scheduling parameters of the process specified by pid to policy and the parameters specified in the sched_param structure pointed to by param, respectively. The
value of the sched_priority member in the sched_param structure is any integer ithin the inclusive priority range for the scheduling policy specified by policy. If the value of pid is negative, the behavior of the sched_setscheduler() function is unspecified. The possible values for the policy parameter are defined in the header file <sched.h>:

SCHED_FIFO
(realtime), First-In-First-Out; processes scheduled to this policy, if not pre-empted by a higher priority or interrupted by a signal, will proceed until completion.

SCHED_RR
(realtime), Round-Robin; processes scheduled to this policy, if not pre-empted by a higher priority or interrupted by a signal, will execute for a time period, returned by sched_rr_get_interval(3R) or by the system.

SCHED_OTHER
(time-sharing)
If a process specified by pid exists and if the calling process has permission, the scheduling policy and scheduling parameters are set for the process whose process ID is equal to pid. The real or effective user ID of the calling process must match the real or saved (from exec(2)) user ID of the target process unless the effective user ID of the calling process is 0. See intro(2).If pid is 0, the scheduling policy and scheduling parameters are set for the calling process. To change the policy of any process to either of the real time policies

SCHED_FIFO or SCHED_RR, the calling process must either have the SCHED_FIFO, or SCHED_RR policy or have an effective user ID of 0. The sched_setscheduler() function is considered successful if it succeeds in
setting the scheduling policy and scheduling parameters of the process specified by pid to the values specified by policy and the structure pointed to by param, respectively. The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:
For threads with system scheduling contention scope, these functions have no effect on their scheduling.
For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters will not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions. The system supports a two-level scheduling model in which library threads are multiplexed on top of several kernel scheduled entities. The underlying kernel scheduled entities for the system contention scope threads will not be affected by these functions.

The underlying kernel scheduled entities for the process contention scope threads will have their scheduling policy and associated scheduling parameters changed to the values specified in policy and param, respectively. Kernel scheduled entities for use by process contention scope threads that are created after this call completes inherit their scheduling policy and associated scheduling parameters from the process. This function is not atomic with respect to other threads in the process. Threads are allowed to continue to execute while this function call is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel scheduled entities used by the process contention scope threads.

Return Values

Upon successful completion, the function returns the former scheduling policy of the specified process. If the sched_setscheduler() function fails to complete successfully, the policy and scheduling paramenters remain unchanged, and the function returns -1 and sets errno to indicate the error.

Errors

The sched_setscheduler() function will fail if:
EINVAL
The value of policy is invalid, or one or more of the parameters contained in param is outside the valid range for the specified scheduling policy.
ENOSYS
The sched_setscheduler() function is not supported by the system.
EPERM
The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.
ESRCH
No process can be found corresponding to that specified by pid.

Attributes

See attributes(5) for descriptions of the following attributes:
lw(2.750000i)| lw(2.750000i).
ATTRIBUTE TYPE
MT-Level

See Also

priocntl(1) , intro(2) , exec(2) , priocntl(2) , sched_get_priority_max(3R) , sched_getparam(3R) , sched_getscheduler(3R) , sched_setparam(3R) , attributes(5) , sched(5)

Compilation du noyau

A Kernel 2.2.16

Lors de la compilation, plusieurs problèmes liés aux inclusions sont survenus.

- linux/modversions.h

la solution consiste à sauvegarder la répertoire /usr/include/linux sous /usr/include/linux.ori et de le remplacer par un lien symbolique en effectuant la commande.

```
ln -s /usr/src/linux-2.2.16 /usr/include/linux
```

L'opération doit être répéter sur le répertoire /usr/include/asm que l'on renomme /usr/include/asm.ori

B Kernel 2.4.0

Le passage sur le noyau 2.4.0 n'a été aussi simple que sur la version 2.2.16. Les structures utilisées par le noyau pour gérer la mémoire ayant changer le portage de **CMEM**, élément important de **MPC-OS**, n'a pu se faire. nous avons préfère nous concentrer sur nos stages pour éventuellement y revenir plus tard.

Driver CMEM

A. Les slots utilisés

Les slots qu'il faut réserver afin d'utiliser put et testputbench sont :

- un slot PCIDDC il contient la LPE, la LMI et les pointeurs images de la carte ainsi que les verrous

- un slot PUT qui est utiliser comme tempo d'émission et de réception par l'utilitaire testputbench. Dans le cas ou l'on désire lancer deux testputbenchs simultanément il faut en plus du slot PUT réserver un autre slot PUTBIS.

B. Les ioctls

pour accéder au driver CMEM nous avons défini des ioctl.

- CMEMDRIVER_INITPUT : permet de lancer l'initialisation matérielle de la carte FastHSL

- CMEMDRIVER_GETSTRUCT : copie la structure put_info du mode noyau vers le mode yutilisateur

- CMEMDRIVER_GETOPTCONTIGMEM : renvoi une structure contenant toutes les informations sur le slot PCIDDC

- CMEMDRIVER_PUTSLOT1 : retourne une structure contenant les informations sur le slot PUT

- CMEMDRIVER_PUTSLOT2 : idem pour le slot PUTBIS