THÈSE de DOCTORAT de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité :

INFORMATIQUE

présentée par

Rémy CARD

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

Sujet de la thèse :

MASIX
un Système d'Exploitation Multi-Environnements utilisant le Micro-Noyau MACH

Conception et Réalisation

soutenue le 7 Mai 1993 devant le jury composé de :

Monsieur	Pascal Estraillier	Président
Monsieur	Claude Bétourné	Rapporteur
Monsieur	Gérard Florin	Rapporteur
Monsieur	Paul Feautrier	Examinateur
Monsieur	Claude Girault	Examinateur
Monsieur	Jean-Marie Rifflet	Examinateur
Monsieur	José Rogado	Examinateur

Résumé

Les systèmes d'exploitation ont traditionnellement été conçus sous forme d'un noyau monolithique offrant tous les services nécessaires. Cette approche pose de gros problèmes de maintenance et d'évolution du système. Cette thèse propose une méthode originale de conception de système sous forme de serveurs utilisant les services offerts par un micro-noyau. La décomposition du système en serveurs permet d'obtenir une structure modulaire et très évolutive. Le système décrit dans cette thèse est composé de plusieurs couches. La première, le système d'accueil, offre tous les services permettant d'inclure dynamiquement de nouveaux services et des interfaces utilisateur différentes. La seconde, l'environnement Unix, est constituée d'un ensemble de serveurs réalisant les abstractions et les sémantiques offertes par le système Unix. Cet environnement repose sur des implémentations nouvelles et optimisées tenant compte des fonctionnalités offertes par le micro-noyau Mach. La troisième couche, le système virtuel, est constituée d'un ensemble de serveurs implémentant des fonctionnalités réparties. Ces serveurs sont inclus dynamiquement dans le système en utilisant les possibilités du système d'accueil. De nombreuses améliorations incluses dans le système permettent d'optimiser ses performances malgré les changements de contexte et échanges de messages nécessaires pour communiquer entre serveurs différents. Le système résultant possède une structure novatrice, facilement extensible et constitue la base de recherches futures dans le domaine des systèmes d'exploitation répartis.

Mots-clés : système d'exploitation réparti, micro-noyau, serveur, environnements multiples, système virtuel, système d'accueil, MACH, UNIX

Abstract

Traditional operating systems have been designed as a monolithic kernel implementing the whole set of the system services. This approach leads to important maintenance end evolution problems. This thesis discusses a new design methodology. The operating system is implemented as a set of servers using the services provided by a micro-kernel. Splitting up the operating system into a set of servers leads to a modular and evolutive structure. The system exposed in this thesis is made up of three layers. The first one, the host system, provides services which allow the dynamic inclusion of new servers and different user interfaces. The second one, the Unix environment, contains a set of servers which provide Unix abstractions and semantics. It is based on new and optimized implementations, using the Mach micro-kernel facilities. The third layer, the virtual system, is made up of a set of servers providing distributed features. These servers are dynamically included in the system by using the host system services. Numerous improvements included in the system imply performances optimizations in spite of context switches and messages exchanges needed to make different servers communicate with each others. The resulting structure is innovative and easy to extend. This system is the basis for future distributed operating system researchs.

Keywords : distributed operating system, micro-kernel, server, multiple environments, virtual system, host system, MACH, UNIX

À André, Danièle, Louis, René, Serge, Solange, Sylvie et Xavier,
pour leur amour et leurs encouragements

Remerciements

Je tiens à remercier très sincèrement :

- Monsieur Pascal Estraillier, Professeur à l'Université Pierre et Marie Curie, pour l'aide constante qu'il m'a apportée à travers ses critiques et suggestions et pour me faire l'honneur de présider le jury de cette thèse ;
- Monsieur Claude Bétourné, Professeur à l'Institut de Recherche en Informatique de Toulouse et Monsieur Gérard Florin, Professeur au Conservatoire National des Arts et Métiers, pour l'intérêt qu'ils ont porté à cette thèse et pour en avoir été les rapporteurs ;
- Monsieur Claude Girault, Professeur à l'Université Pierre et Marie Curie, Responsable de la Formation Doctorale de Systèmes Informatiques, pour m'avoir accueilli dans son équipe et avoir constamment soutenu mon travail dont cette thèse est l'aboutissement ;
- Monsieur Paul Feautrier, Professeur à l'Université de Versailles St-Quentin, Monsieur Jean-Marie Rifflet, Professeur à l'Université Paris VII, et Monsieur José Rogado, Ingénieur de Recherche au centre de recherche OSF de Grenoble, pour l'intérêt qu'ils ont porté à mon travail et pour avoir accepté de participer au jury de cette thèse ;
- Monsieur Alain Greiner, Professeur à l'Université Pierre et Marie Curie, sans qui le projet Masix n'aurait jamais vu le jour ;
- Monsieur Jacques Sakarovitch, Directeur de Recherche à l'Université Pierre et Marie Curie, Directeur de l'Institut Blaise Pascal, pour son soutien constant et sa grande patience ;
- Éric Commelin, Pierre David, Stéphane Dayras, Gilles-Éric Descamps, Bertil Folliot, Michel Gruau, Hubert Lê Van Gông, Franck Mével et Pierre-Guillaume Raverdy pour les nombreuses discussions enrichissantes sur le projet Masix ;
- François Bréant et Jean-Michel Couvreur pour leur aide morale durant la rédaction de cette thèse ;
- les membres du bureau 206 pour leur soutien et l'ambiance chaleureuse qu'ils savent faire régner et, plus généralement, tous les membres de l'équipe Modélisation, Analyse et Réalisation de Systèmes Distribués du Laboratoire MASI ;
- L'ensemble du personnel de l'Institut Blaise Pascal pour leur accueil et l'aide qu'ils ont su me procurer.

Table des Matières

Liste des figures

Chapitre 1

Introduction

Nous présentons un système d'exploitation réparti structuré sous forme de multi-serveurs de manière à permettre une évolution facile et des interfaces d'utilisation multiples. Les possibilités d'extension prévues lors de la conception même du système le destinent à s'adapter facilement aux évolutions futures. Ce système, baptisé MASIX, est en cours de développement au laboratoire MASI et constitue le sujet de cette thèse.

1 Structuration d'un système

La structuration d'un système est un problème essentiel qui conditionne l'évolution de ce système. Alors que certains systèmes, comme Locus, [Popek et Walker 1985], OSF/1 IK [OSF 1990], Sprite [Ousterhout *et al.* 1988] ou Unix [Bach 1986, Leffler *et al.* 1989] sont basés sur un noyau monolithique, d'autres systèmes, comme Amoeba [Mullender *et al.* 1990], Chorus [Rozier *et al.* 1988], OSF/1 MK [Places *et al.* 1992], V [Cheriton 1988] ou Windows NT [Custer 1993], reposent sur une structure à base de serveurs coopérants au dessus d'un micro-noyau.

La structuration traditionnelle sous forme d'un seul noyau permet une communication facile entre les différents modules partageant le même espace d'adressage mais tend à compliquer la maintenance et l'évolution du système. L'utilisation de serveurs en mode utilisateur, par contre, assure de plus grandes possibilités d'évolution du système mais pose des problèmes de performances dans la mesure où elle implique des échanges de messages et des changements de contexte pour réaliser les appels système.

Dans le cas de Masix, nous nous intéressons à l'optimisation de cette deuxième structure afin d'obtenir un système qui soit à la fois évolutif, facilement maintenable et performant.

Nous proposons de nombreuses améliorations à inclure dans la réalisation d'appels système. Nous établissons une nouvelle répartition des appels système et des fonctions de librairie visant à minimiser le nombre de requêtes transmises à des serveurs. Nous définissons également des stratégies optimales de partage de données et d'interactions entre serveurs.

Les réseaux d'ordinateurs fournissent une opportunité pour un système d'exploitation de tirer parti des ressources disponibles sur chacun des éléments du réseau (stations par exemple). Afin de pouvoir bien exploiter la puissance des processeurs connectés par un réseau et de l'ensemble des ressources (processeurs, périphériques), Masix doit offre des fonctionnalités réparties :

- implicites :

  Le système met à la disposition de chaque station l'ensemble des ressources distantes et les utilisateurs y accèdent de manière transparente, comme des ressources locales, sans avoir à modifier leurs programmes ou les commandes qu'ils utilisent.

- explicites :

  Le système fournit aux utilisateurs les moyens de profiter de la puissance répartie en offrant des outils de conception d'applications parallèles. Les utilisateurs ne sont pas concernés par la répartition réelle des ressources mais ils peuvent indiquer au système leur préférences pour leurs applications afin de profiter au maximum de cette répartition.

2 Ajout dynamique de nouvelles sémantiques

Les systèmes d'exploitation définissent usuellement un ensemble d'objets auxquels les processus utilisateur accèdent grâce aux primitives fournies. Chaque type d'objet est géré selon un ensemble de sémantiques prédéfinies qui ne peuvent pas être modifiées ou étendues sans changer la structure du système.

Généralement, la prise en compte de nouvelles sémantiques nécessite l'ajout d'une couche d'indirection dans le système afin d'aiguiller les requêtes des processus utilisateur vers un module chargé de les statisfaire. Cette couche d'indirection concentre la connaissance de l'ensemble des modules concernés.

Dans Masix, nous proposons une méthode permettant de modifier dynamiquement les sémantiques liées aux objets gérés. Cette méthode repose sur la définition de protocoles représentant les opérations disponibles sur un type d'entité. Tout type d'objet géré est caractérisé par un protocole d'accès constitué de toutes les opérations utilisables sur les objets du type. Toute opération demandée sur un objet est transmise de manière transparente au module concerné, sans nécessiter de couche d'indirection.

À notre connaissance, aucun autre système d'exploitation ne permet actuellement cet ajout dynamique d'environnements et de fonctionnalités. Les systèmes existants se contentent de prévoir des émulations limitées dont l'extension nécessite des modifications importantes dans le noyau.

3 Environnements de travail

La coexistence de systèmes d'exploitation pose un problème important de compatibilité pour les utilisateurs. Il est donc important, dans un système, de prévoir la possibilité d'offrir une interface similaire à celle d'un autre système.

Une nouvelle implémentation de fonctionnalités Unix, même si elles sont basées sur une nouvelle structure, ne justifierait pas la conception d'un nouveau système.

Un objectif de Masix est donc de constituer un système ouvert, facilement extensible. Nous souhaitons disposer d'un système évolutif dans lequel de nouvelles fonctionnalités soient faciles à intégrer. Pour cela, le système doit être conçu pour faciliter l'ajout dynamique de composants logiciels, offrant des fonctionnalités étendues ou fournissant une interface différente.

Deux méthodes sont couramment employées :

- Un émulateur exécute des programmes et redirige les appels adressés au système simulé vers le système cible [Malan *et al.* 1991].
- Plusieurs modules d'interface coexistent dans le système et offrent à l'utilisateur des environnements de travail différents [Custer 1993].

Nous avons mis en oeuvre des environnements multiples dans Masix afin de permettre aux utilisateurs de profiter de plusieurs interfaces différentes. Contrairement aux approches existantes qui nécessitent que le système connaisse à l'avance les environnements pouvant être exécutés, nous proposons une décomposition entre un système d'accueil assurant les fonctionnalités de bas niveau et plusieurs environnements pouvant être ajoutés et supprimés de manière dynamique. Ce système d'accueil est chargé de lancer l'exécution de nouveaux serveurs et de gérer les environnements présents en assurant un service de nommage et de surveillance du système.

4 Portabilité et compatibilité

La portabilité d'un système d'exploitation constitue un objectif à part entière. En effet, un système non portable est voué à disparaitre à plus ou moins long terme en raison de l'évolution rapide des architectures matérielles. Le système Multics [Organick 1972] constitue un parfait exemple de ce phénomène : alors qu'il contenait des mécanismes qui sont "redécouverts" de nos jours dans les systèmes d'exploitation modernes, ce système a quasiment disparu parce qu'il était lié à un ordinateur désormais technologiquement dépassé.

Afin d'être portable, un systême doit être structuré avec soin pour que les modules dépendants de la machine cible soit clairement isolés et puissent être remplacés facilement lors d'un portage sur une nouvelle architecture.

Le système Masix offre une interface similaire à celle du système Unix[1]. Cette interface, aussi bien au niveau utilisateur que programmeur, repose sur les fonctionnalités suivantes :

- processus s'exécutant en pseudo-parallélisme ;
- communications entre processus, aussi bien locales (tubes, signaux, sémaphores, mémoire partagée, messages) que via un réseau (*sockets*) ;
- gestion de mémoire virtuelle ;
- systèmes de fichiers hiérarchiques regroupés en une structure arborescente par des opérations de montage ;
- fonctions d'administration.

Il est délicat de parler de compatibilité Unix dans l'absolu. En effet, il existe actuellement plusieurs versions et variantes d'Unix caractérisées par de nombreuses différences. Nous avons donc classé, par ordre de priorité décroissante, les fonctionnalités à offrir selon :

- les fonctions exigées par le standard Posix [IEEE 1990a, Lewine 1991] ;
- les fonctionnalités offertes par la version BSD de Unix, développée à l'université de Berkeley [Quaterman *et al.* 1985] ;
- les fonctionnalités de communication offertes par Unix System V [AT&T 1989].

Comme Masix offre une compatibilité source avec Unix, il est possible d'exploiter les nombreux programmes disponibles pour ce dernier. La recompilation et l'exécution d'applications Unix sous Masix leur permet de profiter des ressources réparties via les accès implicites. Une application conçue et développée sous Masix sera parfaitement compatible avec Unix si elle se restreint à l'ensemble des fonctionnalités réparties implicites.

Néanmoins, une application désirant utiliser de manière optimale les possibilités réparties de Masix devra être modifiée afin d'effectuer des appels explicites aux modules de gestion de la répartition de Masix. Un tel programme ne sera alors plus compatible au niveau source avec Unix.

5 Plan

Nous présentons dans le chapitre 2 un état de l'art sur les systèmes d'exploitation répartis. Un système réparti est formé de la coopération des services offerts par plusieurs systèmes d'exploitation communiquant via un réseau pour former un système uniprocesseur virtuel. Un système réparti offre un plus grand niveau de fiabilité qu'un système centralisé traditionnel.

[1] Unix est une marque déposée de USL (*Unix System Laboratories*)

Il permet, de plus, un partage aisé d'informations entre utilisateurs grâce à ses mécanismes de communication. Enfin, l'extension d'un tel système est relativement facile car il est généralement conçu pour permettre l'ajout ou le remplacement de composants matériels ou logiciels.

Nous étudions dans le chapitre 3 la structure d'un système d'exploitation. Masix, pour sa part, est structuré en couches non monolithiques implémentées par des serveurs coopérants utilisant le micro-noyau Mach comme base. Ces serveurs assurent aussi bien le système d'accueil permettant l'exécution d'environnements multiples que les fonctionnalités Unix et réparties. La structuration du système sous forme de serveurs permet un développement et une mise au point rapide et garantit également une grande extensibilité du système.

Le chapitre 4 détaille les fonctionnalités que Mach met à la disposition d'un concepteur de système d'exploitation. Les abstractions et les mécanismes fournis par Mach peuvent être utilisés par un système s'exécutant en mode utilisateur pour implémenter ses sémantiques. Plusieurs systèmes opérationnels ou en cours de conception au dessus de Mach offrent déjà des sémantiques Unix mais utilisent des structures différentes.

Nous exposons dans le chapitre 5 nos principes de conception. Masix est un système multi-serveurs offrant aussi bien un système d'accueil permettant d'exécuter plusieurs environnements différents que des sémantiques Unix. Le caractère multi-serveurs du système permet d'obtenir une grande dynamicité et offre de nombreuses possibilités d'évolution. Ce découplage entraine également de nouveaux problèmes qu'on ne rencontre pas dans un système implémenté selon une structure traditionnelle. Chacun des serveurs de Masix offre un ensemble de services et gère les communications avec ses clients en assurant le nommage des objets ainsi que leur protection. Nous avons inclus des optimisations dans le système, aussi bien dans les couches d'interface des clients que dans les communications entre serveurs, afin de minimiser les changements de contexte et les échanges de messages.

Le chapitre 6 présente les stratégies implémentées par les serveurs formant le système local de Masix. Un serveur assure les fonctionnalités d'administration du système et de nommage des serveurs. D'autres serveurs implémentent la compatibilité Unix. Un serveur est chargé de gérer les processus et les signaux en utilisant les tâches et *threads* de Mach comme base. La mémoire est gérée par un paginateur qui utilise des zones d'échange qu'il structure afin de maintenir les zones de mémoire virtuelle allouées. Les fichiers sont rendus accessibles en mémoire virtuelle par des paginateurs chargés de lire ou d'écrire les pages nécessaires en interprétant le format des systèmes des fichiers physiques supportés. Enfin, chaque type de périphérique est géré par un serveur qui utilise les pilotes de périphériques fournis par Mach pour effectuer les entrées-sorties physiques.

Nous décrivons dans le chapitre 7 la méthode que nous avons utilisée pour implémenter Masix. Nous avons utilisé le système d'exploitation Linux [Klaus 1993] pour procéder au développement croisé de Masix. Afin de remédier aux déficiences de Linux pour l'accès aux fichiers, nous avons modifié ses stratégies internes et développé deux systèmes de fichiers plus évolués que celui qui était inclus dans la distribution officielle de Linux. Nos extensions ont été ensuite incluses dans Linux et diffusées de la même façon que ce système ce qui nous a permis de profiter des expériences, aussi bien positives que négatives, de l'importante communauté des utilisateurs de ce système de part le monde. Le deuxième des systèmes de fichiers que nous avons développés est désormais le système de fichiers par défaut de nombreuses distributions de Linux et constitue la base de Masix.

Le chapitre 8 décrit les services répartis formant le système virtuel de Masix. Ces services sont intégrés dans Masix en tirant parti des mécanismes de nommage et de communication définis lors de la conception du système. Un processus utilisateur accède à un serveur distant de manière transparente comme s'il s'exécutait sur la même station que lui. Les services répartis inclus dans Masix permettent ainsi de constituer un système réparti apparaissant à l'utilisateur comme un seul système uniprocesseur virtuel.

Le chapitre 9 conclut en faisant le point sur l'état actuel du système Masix et expose les perspectives futures.

Chapitre 2

État de l'art sur les systèmes répartis

1 Introduction

Les réseaux informatiques sont utilisés pour communiquer et pour partager des ressources. Afin de gérer ces accès partagés, des extensions réparties sont ajoutées aux systèmes d'exploitation traditionnels et une nouvelle forme de système, appelée système réparti, est apparue.

La première motivation à la conception de systèmes répartis réside dans leur coût. Il y a une vingtaine d'années, la puissance d'une machine évoluait selon la loi de Grosch : la puissance de calcul d'un processeur est proportionnelle au carré de son prix. En multipliant le prix par deux, il était ainsi possible d'acquérir une machine quatre fois plus puissante.

De nos jours, cette loi ne s'applique plus en raison des nouvelles technologies dans le domaine des processeurs. En doublant le prix, on n'obtient qu'un processeur, fonctionnant à une fréquence d'horloge un peu plus élevée. La solution la plus économique pour constituer un système consiste alors à regrouper un grand nombre de processeurs bon marché. Ainsi, un système réparti constitué de tels processeurs possède un rapport puissance/prix plus intéressant qu'un seul système centralisé.

Un système réparti peut également fournir une meilleure performance qu'un système centralisé. En effet, il est plus facile de cumuler la puissance d'un grand nombre de processeurs que de concevoir un seul processeur assez puissant.

Un système réparti présente, de plus, de nombreux autres avantages par rapport à un système centralisé :

- il est plus fiable : dans un système centralisé, une panne entraîne l'indisponibilité de tout le système ; dans un système réparti, une panne peut causer l'indisponibilité d'une machine mais ne fera que réduire la puissance globale du système qui pourra continuer à s'exécuter, éventuellement en mode dégradé ;

- il permet une extension aisée : alors que l'évolution d'un système centralisé consiste à remplacer la machine par un ordinateur plus puissant ou à ajouter une deuxième machine, un système réparti peut être étendu au cours du temps par l'ajout de processeurs ;
- il permet d'utiliser au mieux les ressources en les partageant entre les utilisateurs ;
- il offre de nombreux mécanismes de communication permettant aux utilisateurs de partager des informations.

Un système réparti présente également certains inconvénients par rapport à un système classique :

- le logiciel système est beaucoup plus complexe à écrire et à mettre au point que dans le cas d'un système centralisé ;
- il repose sur un réseau de communication non fiable :
  - le logiciel doit gérer tous les problèmes de communications ;
  - une panne ou une saturation du réseau peut nécessiter la mise en place de solutions de remplacement ;
- le partage d'informations pose de nombreux problèmes de protection.

Nous présentons tout d'abord les modèles de référence que respectent les systèmes répartis ainsi que les fonctionnalités qu'ils assurent.

Nous détaillons ensuite la façon dont ces fonctionnalités peuvent être implémentées dans des systèmes à architecture répartie basé sur Unix.

Enfin, nous étudions les choix effectués dans les systèmes répartis de référence actuels et les stratégies qu'ils implémentent.

2 Systèmes répartis

2.1 Modèles de référence

On distingue généralement trois modèles de conception de systèmes d'exploitation répartis :

1. le modèle "Stations de travail" ;
2. le modèle "Bancs de processeurs" ;

3. des modèles hybrides.

Un système conçu selon le modèle "Stations de travail" est composé de stations de travail individuelles reliées par un réseau de communication. A tout instant, une station peut être inactive ou occupée par un utilisateur, qui est connecté.

Ce modèle garantit à chaque utilisateur qu'il dispose d'une certaine puissance de traitement, qu'il possède un degré important d'autonomie et qu'il peut organiser sa station comme il le souhaite.

Dans ce modèle, il est nécessaire d'inclure dans le système des mécanismes de répartition de charge afin d'optimiser l'utilisation des stations de travail temporairement inactives [Folliot 1992, Litzkow *et al.* 1988, Nichols 1987].

Un système conçu selon le modèle "Bancs de processeurs" est constitué de deux types de composants :

- des terminaux (graphiques ou non) utilisés pour l'interface utilisateur ;
- des bancs de processeurs groupés qui sont alloués par le système aux utilisateurs en fonction de leurs besoins.

Contrairement au modèle "Stations de travail", il n'y a pas ici de notion de propriété sur les processeurs : chaque utilisateur peut se voir affecté autant de processeurs qu'il en a besoin pendant des périodes de temps déterminées, puis ces processeurs sont rendus aux bancs de processeurs pour être alloués à d'autres traitements.

Une combinaison de ces deux modèles consiste à offrir aux utilisateurs des stations de travail utilisées pour les travaux interactifs couplées avec des bancs de processeurs utilisés pour les traitements importants.

Cette approche combine les avantages des deux modèles : un bon temps de réponse est garanti aux utilisateurs pour les travaux interactifs et les processus non interactifs peuvent utiliser toute la puissance de calcul fournie par les bancs de processeurs sans perturber les utilisateurs.

2.2 Systèmes à architecture répartie et systèmes répartis

Les premiers systèmes offrant des possibilités de partage et de communications sont les systèmes à architecture répartie. Ces systèmes possèdent les caractéristiques suivantes qui les distinguent des systèmes d'exploitation répartis [Tanenbaum et van Renesse 1985] :

- chaque ordinateur exécute son propre système d'exploitation au lieu d'exécuter une partie d'un système global ;

- chaque utilisateur utilise son propre ordinateur, l'utilisation de machines différentes nécessite des commandes spécifiques (connexion à distance, exécution à distance...) au lieu de laisser le système d'exploitation allouer dynamiquement les processeurs ;
- les utilisateurs ont conscience de la localisation de leurs données et doivent y accéder par des commandes spécifiques (transfert de fichiers) au lieu de laisser le système gérer le placement des données ;
- le système ne possède pas (ou peu) de tolérance aux pannes : si une partie des ordinateurs tombe en panne, les utilisateurs de ces ordinateurs ne peuvent plus utiliser le système.

Par contraste, un système réparti apparaît aux utilisateurs comme un système centralisé, mais s'exécute sur des processeurs différents et indépendants. Le concept clé est la transparence, l'utilisation de plusieurs processeurs doit être invisible aux utilisateurs [Tanenbaum et van Renesse 1985].

On peut donc assimiler un système réparti à un système monoprocesseur virtuel formé de la coopération de plusieurs systèmes d'exploitation reliés par un canal de communication.

Un système réparti doit offrir plusieurs formes de transparence [ANSA 1987] au niveau :

- des accès, permettant d'accéder aux données locales et distantes par des opérations identiques ;
- de la localisation, permettant d'accéder à des objets sans connaître leur localisation réelle ;
- de la concurrence, permettant à plusieurs utilisateurs ou programmes d'accéder simultanément à des données partagées sans interférences ;
- des duplications, permettant de stocker plusieurs instances des données utilisées pour accroître les performances et la tolérance aux pannes sans que les utilisateurs en aient conscience ;
- des pannes, permettant aux utilisateurs de continuer à travailler (parfois en mode dégradé) en dépit de pannes matérielles ou logicielles ;
- des migrations, permettant de déplacer des objets au sein du système sans affecter les utilisateurs et les programmes ;
- de la performance, permettant de reconfigurer dynamiquement le système afin d'en optimiser les performances ;
- de la taille, permettant d'étendre dynamiquement le système sans modifier sa structure ni les algorithmes utilisés.

La transparence totale est difficile à atteindre et n'est pas toujours désirable : il n'est pas envisageable, par exemple, qu'un système réparti choisisse de manière transparente les imprimantes qu'il utilise sans en avertir les utilisateurs afin qu'ils sachent où aller chercher leurs impressions.

2.3 Fonctionnalités d'un système réparti

Les systèmes répartis introduisent plusieurs problèmes jusqu'ici ignorés dans les systèmes centralisés [Krakowiak 1987] :

- distance accrue entre la machine virtuelle à réaliser et le ou les réseaux de machines physiques, souvent hétérogènes, communiquant par des voies sujettes aux retards et aux pannes ;
- impossibilité pour les gestionnaires de ressources d'avoir une perception exacte de l'état global du système.

Les problèmes principaux à résoudre sont les communications, la gestion et la synchronisation de processus, le nommage, la protection, l'allocation de ressources et la tolérance aux pannes.

2.3.1 Communications

Les mécanismes de communication sont primordiaux dans un système réparti. Ils permettent aux systèmes et aux divers sous-systèmes présents sur des sites différents de se transmettre des informations. Ils sont à la base de l'accès à des ressources distantes.

Du point de vue des communications, un système réparti est caractérisé par [Tanenbaum et van Renesse 1985, Krakowiak 1987] :

- l'absence de mémoire partagée ;
- l'absence d'horloge commune ;
- des liens de communication non fiables.

Tout système réparti fournit une couche de communication implémentant soit le passage de messages [Basket *et al.* 1977, Rashid et Robertson 1981], soit l'appel de procédures à distance [Nelson 1981, Birrell et Nelson 1984], soit les transactions [Spector 1986].

2.3.2 Synchronisation

La vue globale d'un système réparti est généralement difficile, voire impossible, à obtenir. Le système doit donc fournir une vue partielle et cohérente d'un ensemble de programmes communiquants. Ces vues peuvent être obtenues au moyen de mécanismes de synchronisation.

Il est beaucoup plus difficile d'implémenter des mécanismes de synchronisation dans un système réparti que dans un système centralisé. Ces mécanismes peuvent être fournis grâce à des horloges logiques [Lamport 1978, Ricart et Agrawala 1981], à un passage de jeton sur un anneau virtuel [Lann 1983] ou en utilisant un ordonnancement avec priorités [Goscinski 1991].

Une étude détaillée des problèmes que posent les synchronisations figure dans [Raynal 1984, Raynal 1988].

2.3.3 Nommage

Dans un système, tout objet doit être désigné par un nom afin de pouvoir y accéder. De plus, dans un système réparti, les objets peuvent être déplacés au cours du temps, la migration et la duplication d'entités étant nécessaires à la tolérance aux pannes et à la répartition de charge. Ces déplacements posent des problèmes de localisation [Needham 1989].

Un processus qui veut accéder à un objet doit fournir son nom au système. Cette désignation ne doit pas être dépendante de la localisation des objets car ces derniers doivent pouvoir être déplacés tout en conservant les mêmes noms. Les noms sont caractérisés par trois attributs :

- l'espace de nommage, plat ou hiérarchisé ;
- le temps, un nom peut être statique ou dynamique ;
- le nombre d'objets, un nom peut référencer un objet ou un groupe d'objets.

Il est également possible de référencer des objets par des capacités [Fabry 1974], identificateurs attribués par un serveur, qui sont utilisées aussi bien pour la désignation que pour la protection des objets [Tanenbaum *et al.* 1986].

2.3.4 Protection

Dans la mesure où les informations manipulées sont stockées sur des sites différents et qu'elles empruntent des voies de communication potentiellement peu sûres, un système réparti doit offrir des mécanismes de protection plus élaborés qu'un système centralisé. Ces mécanismes visent à garantir que les personnes non autorisées ne peuvent pas accéder aux données, ni sur l'un des sites composant le système, ni sur le réseau de communication.

Toutes les ressources ne doivent pas être pas accessibles par l'ensemble des utilisateurs. Un système réparti doit donc garantir [Mullender 1989a] :

- la protection des ressources ;
- la sécurité des communications ;
- l'authentification des utilisateurs.

Afin d'être efficaces, les mécanismes de protection doivent être conçus et implémentés comme une partie intégrante du système et non comme une couche supplémentaire. Ils doivent supporter plusieurs protocoles différents de façon à offrir une souplesse maximale aux utilisateurs [Tanenbaum 1992].

La majorité des sytèmes répartis offre un niveau de sécurité assez élevé. Un compromis entre les besoins de partage et les besoins de sécurité doit généralement être trouvé afin de rendre les mécanismes de protection acceptables par les utilisateurs. L'administrateur réseau est alors chargé de cette tâche [Needham 1985, Coulouris et Dollimore 1988].

2.3.5 Allocation et gestion de ressources

L'allocation et la gestion de ressources constituent un problème complexe dans les sytèmes centralisés. Dans un système réparti, des complications supplémentaires sont causées par la répartition physique des ressources, les délais et les erreurs de communication, la redondance, les pannes partielles et l'absence d'état global.

Les ressources peuvent être décomposées en deux catégories [Goscinski 1991] :

- ressources physiques : composants matériels d'un ordinateur (processeurs, mémoire, périphériques...) ;
- ressources logiques : informations stockées sur des ressources physiques (fichiers) ou entités abstraites et temporaires (processus).

Ces deux types de ressources peuvent être gérés en [Mullender 1984] :

- planifiant et identifiant la localisation des ressources, leur disponibilité et leur coût ;
- contrôlant et optimisant l'accès aux ressources ;
- assurant le bon fonctionnement (détection, prévention, correction des interblocages ; gestion des indisponibilités en cas de panne).

Les processeurs et les fichiers constituent des ressources particulièrement importantes dont nous présentons succintement la gestion.

2.3.6 Gestion de processus

Un système réparti doit partager les processeurs répartis dans le système. Pour cela, il doit fournir des mécanismes permettant d'agir sur les processus locaux ou distants : création, nommage, migration, terminaison... De plus, le système doit offrir un contrôle réparti et des outils de synchronisation pour tous les processus communiquants ou utilisateurs de ressources.

La gestion de processus répartis peut être décomposée en deux parties distinctes :

- l'une implémentant les stratégies d'allocation de processeur :

  cette partie doit décider quel processus doit être réparti, sur quel processeur et à quel moment ;

- l'autre les mécanismes :

  cette partie implémente le placement ou la migration de processus.

La migration de processus pose de nombreux problèmes notamment pour déterminer l'état d'un processus à migrer, transférer cet état sur le site de destination et intégrer le processus dans son nouvel environnement. Il est également nécessaire d'assurer un traitement spécial pour les communications liées au processus migré. Il est possible d'assurer la continuité des communications en redirigeant les messages, en prévenant les pertes de messages ou en retransmettant les messages perdus suite à la migration.

2.3.7 Gestion de fichiers répartis

Tout utilisateur est potentiellement un créateur et un consommateur de données. Le service de gestion de fichiers répartis est donc un des plus importants dans un système. Le gestionnaire de fichiers doit offrir la transparence de localisation et l'accès aux fichiers. Sa tâche peut être considérée sous deux aspects : le stockage de fichiers et leur adressage [Coulouris et Dollimore 1988].

On peut distinguer trois classifications de gestionnaires de fichiers répartis :

- la première distingue les serveurs qui reconnaissent les noms textuels et gèrent les répertoires des serveurs de stockage ;
- la deuxième distingue les serveurs de fichiers traditionnels, présents sur la majorité de systèmes, des serveurs robustes offrant un certain niveau de fiabilité aux applications. Ces derniers fournissent généralement des services de mise à jour atomique sous forme de transactions ;
- la dernière distingue deux formes d'accès aux fichiers : les accès explicites où le client invoque le serveur de fichiers et les accès transparents où le système localise automatiquement le serveur gérant le fichier accédé, effectue les opérations de sécurité et transfère les données demandées.

De nombreux systèmes de gestion de fichiers ont été conçus et réalisés. On peut citer AFS [Howard 1988], Athena [Champine *et al.* 1990], Bullet [Tanenbaum 1992], Locus [Mueller *et al.* 1983], NFS [Sandberg *et al.* 1985].

L'accès réparti aux fichiers introduit des problèmes de cohérence des données en raison de l'accès simultané, aussi bien en lecture qu'en écriture, aux mêmes informations. Des solutions à ces problèmes sont présentées dans [Bernstein 1985, Davidson *et al.* 1985, Nicol *et al.* 1988].

2.3.8 Tolérance aux pannes

Les systèmes répartis, de par leur nature, sont plus exposés aux pannes que les systèmes centralisés. En effet, alors que la panne d'un système traditionnel n'a de conséquence que sur lui-même, la panne d'une station dans un système réparti peut avoir des incidences sur les autres.

Un système réparti doit donc fournir des mécanismes de tolérance aux pannes afin de pouvoir offrir des mécanismes de remplacement lors de la défaillance de services. Deux techniques sont généralement utilisées : la redondance et les transactions.

La redondance peut être appliquée aux composants matériels (disques, périphériques...) ou logiciels (informations, processus...) [Nelson 1990]. Elle permet de garantir une certaine résistance aux pannes dans un système réparti en offrant des solutions de remplacement en cas de panne. Elle permet de plus d'optimiser l'accès aux informations en conservant une copie des données sur les sites où elles sont les plus accédées. De la même façon que pour les fichiers partagés, des mécanismes doivent alors être inclus afin de préserver la cohérence des données.

La possibilité de rendre atomique une suite d'opérations constitue un outil de base pour la construction d'applications tolérantes aux défaillances. Les mécanismes de transactions, tout d'abord introduits pour l'accès aux bases de données centralisées [Gray 1978, Bernstein et Goodman 1981], ont ensuite été étendus aux transactions réparties [Spector *et al.* 1985]. Ces mécanismes permettent de faciliter la reprise en cas de panne puisqu'une transaction ne peut aboutir qu'à deux états seulement : succès ou échec.

3 Extensions d'Unix

De nombreuses extensions ont été développées au dessus du système Unix afin de permettre à plusieurs sites connectés en réseau de partager des données de manière plus ou moins transparente. Ces extensions sont généralement implémentées comme une surcouche du système utilisant les possibilités de communication réseau fournies par le noyau [Comer 1991, Rifflet 1990, Stevens 1990].

3.1 Nommage hiérarchique : DNS

Le DNS (*Domain Name Service*) constitue le système de nommage des machines au niveau du réseau Internet [Mockapetris et Dunlap 1988]. Il repose sur la notion de domaines hiérarchiques. Lorsqu'un nom est recherché, une requête est envoyée à un serveur du domaine principal qui transmet la demande à un serveur du domaine inférieur, et ainsi de suite jusqu'à obtention de l'information.

Afin d'offrir un service tolérant aux pannes, les serveurs sont dupliqués. Chaque domaine est géré par :

- un serveur maître, seul capable de modifier les informations ;
- plusieurs serveurs secondaires, possédant une copie des informations qu'ils ne peuvent pas modifier.

Un mécanisme de cache est utilisé afin d'obtenir de bonnes performances lors des recherches. Tout client maintient, de manière locale, les informations qu'il a déjà recherchées et les adresses des serveurs qu'il a déjà contactés. Afin d'assurer la cohérence des informations dans les caches, des stratégies d'invalidation sont implémentées :

- la table des informations d'un domaine possède un numéro de version incrémenté à chaque modification sur le serveur primaire. Les serveurs secondaires vérifient périodiquement que leurs informations sont à jour en comparant leur numéro de version avec celui du serveur maître et ré-actualisent leurs données si nécessaire ;
- à toute information obtenue par une requête au serveur de noms est associé un délai de validité (*Time To Live*). Un client peut conserver une information dans son cache local tant que ce délai n'est pas écoulé. Après l'expiration de ce délai, les données correspondantes sont purgées du cache et le client doit transmettre une nouvelle requête au serveur pour obtenir l'information.

3.2 Partage d'informations : NIS

NIS (*Network Information Service*), développé par Sun Microsystems, permet de partager des informations entre plusieurs stations [Sun 1990]. Ce système regroupe en un domaine logique plusieurs types de stations :

- les serveurs qui possèdent une table d'association entre noms et données ;
- les clients qui accèdent à ces informations par des requêtes aux serveurs. Les serveurs se comportent également de cette façon pour l'accès aux données.

Afin de garantir une disponibilité des données même en cas de panne, NIS permet de définir plusieurs serveurs pour le même domaine :

- le serveur maître possédant le droit de modifier les informations ;

- les serveurs esclaves possédant une copie des informations qu'ils ne sont pas autorisés à modifier.

Lors du lancement du service sur un client, une requête est diffusée sur le réseau. Les serveurs disponibles répondent à cette requête et le client choisit d'interroger le premier serveur ayant répondu. Si ce serveur tombe en panne, les clients qu'ils servait le détectent grâce à des délais et diffusent à nouveau une requête afin de choisir un nouveau serveur.

La cohérence des informations est assurée en restreignant les modifications sur le seul serveur maître. Lors d'une modification, le maître contacte chacun des esclaves afin de lui communiquer les nouvelles données. Le serveur maître est défini de manière statique lors de la configuration de NIS ce qui implique qu'une panne de ce maître entraine un fonctionnement dégradé du système dans lequel les modifications sont impossibles. Il est nécessaire de procéder à une reconfiguration explicite du système dans ce cas.

3.3 Système de fichiers répartis : NFS

NFS (*Network File System*) permet de partager une arborescence de fichiers entre plusieurs stations [Sandberg *et al.* 1985]. Des stations serveurs exportent un sous-ensemble de leurs répertoires et des stations clientes peuvent y accéder de la même façon que pour leurs fichiers locaux par une opération de montage.

Deux protocoles sont utilisés par NFS :

- le protocole de montage : un client transmet un nom de répertoire à un serveur afin d'obtenir sa permission. Si le chemin existe et est exporté vers ce client, le serveur renvoie un descripteur au client. Ce descripteur identifie le système de fichiers monté et doit être fourni par le client lors de chaque requête d'entrées-sorties ;
- le protocole d'accès aux fichiers et aux répertoires : un client transmet un message de requête à un serveur afin de manipuler les attributs, lire ou écrire des fichiers et répertoires.

NFS repose sur la notion de serveur sans état : le protocole ne prévoit pas de requête correspondant à l'ouverture et à la fermeture de fichiers. Toute requête d'entrées-sorties contient le nom du fichier à manipuler. Le serveur n'a donc pas à conserver d'informations internes sur les fichiers ouverts par les clients. Cette absence d'état permet une reprise facile en cas de défaillance d'un serveur : dans la mesure où il ne mémorise pas d'informations sur les fichiers ouverts, il n'est pas nécessaire de reconstituer son état lorsqu'il redémarre.

Cette absence d'état rend difficile l'implémentation exacte de certaines sémantiques Unix, notamment le verrouillage de fichiers. Un mécanisme complémentaire doit alors être associé à NFS.

NFS utilise le mécanisme de protection d'Unix afin de vérifier les droits d'accès aux fichiers. Pour cela, un client transmet un identificateur de l'utilisateur au serveur à chaque requête. Le serveur utilise cet identificateur pour vérifier les droits d'accès sans implémenter de mécanisme d'authentification. Cette confiance du serveur pour le client peut poser des problèmes de sécurité dans un environnement peu fiable [Reid 1990].

Afin de rendre transparent l'accès par NFS à des fichiers distants, une couche d'accès à des systèmes de fichiers virtuels a été ajoutée dans Unix. Cette couche repose sur les *vnodes*, descripteurs internes de fichiers permettant d'aiguiller les requêtes d'entrées-sorties sur les systèmes de fichiers locaux ou les serveurs NFS [Kleiman 1986].

Afin d'obtenir de bonnes performances, le protocole NFS effectue les transferts de données entre clients et serveurs par messages de grande taille. Ceci permet de réduire le nombre de paquets transmis entre stations. Des caches sont également gérés par les clients afin de maintenir les attributs et le contenus de fichiers fréquemment accédés. Ces caches permettent d'éviter de nombreux échanges de messages mais introduisent des problèmes de cohérence lors de l'accès simultané par plusieurs clients aux mêmes fichiers. Aucun protocole de cohérence n'est contenu dans NFS : les clients se contentent, pour minimiser les risques, de purger leurs caches et de transmettre les données modifiées au serveur à intervalles de temps réguliers.

3.4 Exécution répartie : Condor

Condor [Litzkow *et al.* 1988] constitue un sous-système d'exécution répartie au dessus d'Unix. Ses objectifs sont les suivants :

- exécuter des processus localement ou sur une station distante de façon transparente aux utilisateurs ;
- assurer une migration de processus dans le cas où une station tombe en panne ;
- répartir équitablement l'exécution entre l'ensemble des stations ;
- minimiser le temps processeur consommé par les mécanismes de placement et de migration ;
- permettre l'exécution distante de processus très importants ;
- ne pas interférer avec les activités locales des utilisateurs.

Condor repose sur deux projets :

- un algorithme de répartition de charge permettant un accès équitable pour les utilisateurs occasionnels du système : l'algorithme *Up-Down* [Mukta et Livny 1987] ;
- un système d'exécution répartie pour Unix : le *Remote Unix* [Litzkow *et al.* 1988].

Dans Condor, chaque station exécute un ordonnanceur local gérant une liste de tâches de fond. Une station particulière exécute également un coordinateur central. Chaque station gère l'exécution de ses processus alors que le coordinateur est chargé de choisir sur quelles stations doivent être lancés les processus.

Le choix d'une station par le coordinateur est basé sur une table qu'il maintient. À intervalles de temps réguliers, le coordinateur interroge les stations afin de mettre ses informations à jour. Chaque station surveille son activité processeur afin de déterminer si elle est inactive. Si un utilisateur local utilise la station, les tâches de fond sont suspendues.

Le *Remote Unix* utilise des points de reprise pour implémenter la migration afin d'éviter un transfert d'état couteux lorsqu'un processus change de station. Le système crée des points de reprise quand un processus s'exécute afin de pouvoir le reprendre ultérieurement sur une station quelconque. Il utilise, de plus, ces points de reprise pour calculer le temps processeur consommé par les processus.

3.5 Exécution répartie : Gatos

Gatos [Folliot 1992] est un système de placement de processus conçu au dessus d'Unix. Il permet le placement et l'exécution distante de processus.

Gatos utilise des algorithmes de placement multicritères. Il prend en compte, en plus de l'état des machines et des contraintes de placement, différents critères d'allocation :

- le temps processeur de chaque programme ;
- la mémoire nécessaire à l'exécution du programme ;
- les fichiers utilisés et créés par chaque programme ;
- les communications entre les programmes ;
- d'autres critères implémentés par des algorithmes fournis par des programmeurs.

Deux serveurs s'exécutent sur chaque station concernée par le placement. Le premier est chargé du placement de processus et du suivi de leur exécution. Le deuxième serveur est seulement chargé du calcul de la charge locale et de sa diffusion périodique à tous les autres serveurs.

4 Systèmes de référence

4.1 Amoeba

Amoeba [Mullender *et al.* 1990] est un projet de recherche sur les systèmes d'exploitation répartis. Son objectif principal est de réaliser un système orienté objets basé sur des capacités [Tanenbaum *et al.* 1986].

Amoeba combine les modèles "Stations de travail" et "Bancs de processeurs".

Le système est composé d'un micro-noyau et de plusieurs serveurs offrant des fonctionnalités de haut niveau. Amoeba fournit des primitives de communication basées sur des files de messages, appelées ports. Ces ports peuvent être utilisés aussi bien par les serveurs pour gérer les entités dans le système (fichiers, processus, périphériques...) que par des applications utilisateur pour communiquer.

4.1.1 Nommage et Protection

Dans Amoeba, tout objet (fichier, processeur...) est référencé par une capacité [Tanenbaum *et al.* 1986]. Une capacité est formée de quatre champs :

- le numéro de port P du serveur ;
- un numéro d'objet n'ayant de signification que pour le serveur ;
- une liste de droits ;
- un champ de vérification contenant une valeur cryptée calculée à partir de la liste de droits du client et des droits maintenus de manière interne par le serveur.

Ainsi, chaque objet possède un nom unique, contenu dans sa capacité, qui est utilisé pour y accéder. Quand un objet est créé par un serveur, une capacité est fournie au client. Celui-ci ne peut accéder à l'objet que par cette capacité. Les opérations autorisées sur l'objet sont encodées dans la capacité.

Les capacités sont gérées entièrement par les serveurs en mode utilisateur. Le noyau lui-même ne fournit aucun mécanisme de protection. Lorsqu'un serveur alloue une capacité, il génère un nombre aléatoire qu'il mémorise dans une table interne et qu'il utilise pour crypter les droits contenus dans la capacité. Lorsqu'un client demande une opération sur un objet via une capacité, le serveur crypte les droits qu'il maintient de manière interne avec le nombre généré et vérifie le résultat avec les droits contenus dans la capacité.

4.1.2 Système de fichiers répartis

Amoeba distingue deux serveurs pour l'accès aux fichiers partagés : un serveur de répertoires et un serveur de fichiers.

Le serveur de répertoires traduit les noms de fichiers fournis en capacités qui sont ensuite utilisées pour accéder aux fichiers.

Le serveur de fichiers, "Bullet Server", offre un ensemble d'opérations simples [Tanenbaum 1992, van Reneese *et al.* 1989]: création, lecture et suppression de fichier. Un processus peut créer un fichier en spécifiant son contenu initial puis modifier ce contenu mais le fichier ne peut pas être accédé tant qu'il n'a pas été fermé par son créateur. Lorsque le fichier a été fermé, il est écrit immédiatement sur disque et il peut ensuite être lu mais non modifié. Afin de modifier le contenu d'un fichier, il est nécessaire de le supprimer préalablement puis de le créer à nouveau.

Le "Bullet Server" implémente un stockage simple des fichiers en les maintenant contigüs en mémoire et sur disque. Il gère la fragmentation des fichiers en compactant la mémoire et les disques quand nécessaire.

Afin de permettre un accès continu aux fichiers, même en cas de panne, le serveur de fichiers a la responsabilité de les dupliquer sur différents disques alors qu'un serveur d'objets duplique les fichiers entre plusieurs serveurs.

4.1.3 Gestion de processus

Amoeba implémente une gestion simple de processus. Il leur fournit la mémoire virtuelle via les tables de pages gérées par le processeur mais n'implémente pas de va-et-vient ni de chargement à la demande. Un processus est donc entièrement résident en mémoire durant son exécution. Cette absence de chargement de pages permet au système d'être plus efficace au niveau des communications entre processus utilisateur, dans la mesure où il n'est pas nécessaire de vérifier que les données transmises sont chargées en mémoire, mais la mémoire n'est pas gérée de façon optimale.

Amoeba fournit la notion de *thread*, processus caractérisé par un état minimal, qui possède l'accès au même segment de données que son père. Des *threads* multiples dans un serveur permettent de traiter plusieurs requêtes en parallèle et peuvent partager les ressources du serveur.

La création de processus est conçue de façon à utiliser des bancs de processeurs. Tout processus créé est censé s'exécuter sur un nouveau processeur. Un processus lance l'exécution d'un nouveau programme par un appel `exec_file` qui ne duplique pas l'état du processus créateur comme c'est le cas dans Unix.

Lors de la création d'un nouveau processus, le serveur d'exécution choisit un processeur pour l'exécuter en se basant sur plusieurs critères dont la charge des processeurs et la mémoire utilisée. Amoeba ne fournit aucun moyen d'allouer des ressources spécifiques à un processus et repose sur la répartition automatique de charge.

Lorsqu'un processus a commencé son exécution, il n'existe aucun moyen de le migrer. Il existe néanmoins la possibilité de sauvegarder l'état d'un processus, de le terminer et de créer un nouveau processus, placé automatiquement sur un nouveau processeur, avec cet état.

4.1.4 Sécurité

Chaque serveur dispose de ports sur lequels sont émises les requêtes des clients. Les numéros de ports des services publics (par exemple, le serveur de fichiers) sont généralement publics alors que ceux des ports liés à des applications utilisateur sont tenus secrets. Le mécanisme de gestion des ports permet ainsi une authentification partielle des clients puisqu'ils doivent connaître le port associé à un service pour pouvoir l'utiliser [Mullender 1989b]. Ce mécanisme ne permet pas cependant d'authentifier les serveurs.

Afin de protéger les serveurs, un mécanisme comportant deux ports G (Get-port) et P (Put-port) est utilisé. P peut être calculé à partir de G par une fonction univoque. Lors de son initialisation, chaque serveur choisit un numéro de port secret G puis diffuse le numéro de port P calculé à partir de G. Le serveur lit ensuite les requêtes des clients en fournissant G au système qui calcule le numéro P correspondant. Il envoie ensuite ses réponses aux clients sur le port P. De cette façon, un programme "Cheval de Troie" ne peut pas se substituer au serveur puisqu'il ne connait pas le numéro de port G utilisé pour communiquer avec le système.

4.2 Athena

Athena [Balkovitch *et al.* 1985, Champine *et al.* 1990] est un projet de système réparti développé au MIT. Il est basé sur le modèle "Stations de travail".

Ses objectifs sont les suivants [Treese 1988] :

- fournir un environnement cohérent sur un ensemble de stations hétérogènes ;
- fournir un environnement logiciel conséquent ;
- pouvoir s'étendre en taille assez facilement.

Athena est basé sur une version de Unix BSD spécialement modifiée pour supporter une exploitation répartie.

4.2.1 Nommage et Administration

SMS (*Service Management System*), le système de gestion de services d'Athena [Rosenstein *et al.* 1988], vise à fournir des mécanismes de gestion des ressources et des serveurs. Il consiste en une base de données, un gestionnaire de données, et des outils d'accès en lecture et en modification de ces données.

SMS doit gérer des serveurs s'exécutant sur diverses stations et est chargé de les reconfigurer lors de modifications de la base.

Il est constitué de six types de composants :

- la base de données ;
- le serveur SMS qui s'exécute sur la station supportant la base de données et qui y accède pour le compte de ses clients ;
- un ensemble de routines d'accès utilisées par les clients pour transmettre leurs requêtes au serveur SMS ;
- les programmes clients utilisant les routines d'accès à la base ;
- le gestionnaire de données DCM (*Data Control Manager*) qui s'exécute périodiquement et (re-)configure les serveurs gérés ;
- les serveurs de modification qui s'exécutent sur chaque machine exécutant un serveur géré par SMS ; ces serveurs sont contactés par DCM lors de l'installation de nouvelles configurations et doivent en informer les serveurs locaux.

4.2.2 Sécurité

Le système Kerberos [Steiner *et al.* 1988], implémentant les mécanismes de protection dans Athena, est un service d'authentification à trois parties basé sur le modèle présenté dans [Needham et Schroeder 1978].

Kerberos maintient une base de données de ses clients et de leurs clefs privées. La clef privée d'un client n'est connue que de Kerberos et du client lui-même. Elle est encryptée par l'algorithme du DES [DES 1977]. Quand un client commence son exécution, son identité doit être prouvée. Pour cela, il doit transmettre sa clef privée à Kerberos qui lui envoie en réponse une clef aléatoire cryptée utilisée pour les échanges ultérieurs entre le client et Kerberos.

Lorsqu'un client souhaite accéder à un serveur, plusieurs étapes sont nécessaires :

- le client envoie une requête à Kerberos pour obtenir une clef ;
- il utilise ensuite cette clef pour contacter un serveur de tickets qui lui fournit un ticket utilisable dans la communication avec le serveur désiré ;
- le client peut alors contacter le serveur grâce au ticket obtenu.

4.3 Chorus

4.3.1 Gestion de processus

Chorus offre la possibilité d'invoquer des processus sur des sites distants lors des appels `fork` et `exec`. Un appel système `csite` permet de spécifier le site sur lequel tous les processus fils seront créés [Armand *et al.* 1989].

Afin de permettre un nommage global des processus, Chorus utilise des identificateurs de 32 bits : 16 bits sont utilisés pour représenter le site de création et 16 bits sont utilisés pour identifier le processus sur ce site.

4.3.2 Systèmes de fichiers répartis

Afin de permettre un accès transparent aux fichiers distants, Chorus offre la possibilité de créer des noeuds de type port symbolique dans l'arborescence des fichiers. Lors de l'interprétation d'un nom de fichiers, si un tel port est rencontré, la suite de la requête est transmise au serveur correspondant à ce port.

Ce mécanisme est utilisé pour interconnecter les systèmes de fichiers d'un réseau de machines sous Chorus et pour fournir un espace global de désignation.

4.4 Locus

Locus [Popek 1981, Popek et Walker 1985, Walker *et al.* 1983] est un système d'exploitation réparti compatible Unix, développé à l'université de Los Angeles. Les objectifs de ce système sont les suivants :

- donner aux utilisateurs la vision d'un seul système centralisé en leur cachant l'existence de plusieurs stations connectées par un réseau ;
- fournir un système de fichiers répartis ;
- permettre l'exécution répartie de processus de manière transparente ;
- offrir des mécanismes de tolérance aux pannes (par exemple, les transactions) ;
- implémenter une compatibilité ascendante avec Unix.

Locus est bâti selon une structure traditionnelle : un noyau unique s'exécute sur chaque station et offre toutes les fonctionnalités du système.

4.4.1 Systèmes de fichiers répartis

Le sytème de fichiers de Locus est basé sur la notion d'arbre hiérarchique et constitue une extension du système de nommage d'Unix. Les extensions implémentées dans Locus sont :

- la structure d'arbre regroupe tous les objets de toutes les machines ; les noms de fichiers sont totalement transparents à l'utilisateur ;

- les fichiers peuvent être répliqués ; le système garantit que toutes les copies accédées sont cohérentes en dirigeant les accès sur les versions les plus récentes.

Les noms de fichiers, dans Locus, peuvent être représentés à deux niveaux :

- au niveau utilisateur, tous les objets sont référencés par des noms uniques contenus dans un espace d'adressage uniforme et hiérarchique ;
- au niveau du système lui-même, l'espace de nommage repose sur des groupes de fichiers. Un groupe de fichiers est équivalent à la notion de système de fichiers d'Unix et correspond à un disque ou à une partition d'un disque.

Chaque groupe de fichiers contient un sous-ensemble de l'arborescence des fichiers. L'ensemble des groupes de fichiers, regroupés par des opérations de montage, forme un seul arbre accessible par l'ensemble des stations.

Toute opération sur un fichier fait intervenir trois sites :

- le client, qui ouvre le fichier et indique ensuite les données à lire ou écrire ;
- le serveur, qui gère les données physiques représentant le fichier ;
- un site de synchronisation, qui gère la synchronisation des accès sur un groupe de fichiers et choisit le serveur pour chaque requête.

Le site de synchronisation tient à jour la localisation des fichiers, et de leurs copies, pour le groupe qu'il gère et est également chargé de la synchronisation des accès concurrents. Pour cette synchronisation, les caches sont gérés avec des jetons. Pour chaque fichier, on peut avoir soit un seul jeton accordé en mode lecture–écriture soit plusieurs jetons en mode lecture seule. Lorsqu'un fichier est ouvert en écriture par un client, le serveur de synchronisation annule les jetons de lecture détenus par les autres clients.

Les modifications de fichiers sont gérées sous forme de transactions afin d'être atomiques. Pour ceci, le fichier originel ainsi que les modifications apportées sont stockés sur le serveur jusqu'à ce que la transaction soit achevée.

4.4.2 Gestion de processus

Locus supporte les appels système Unix `fork` et `exec` qui ont été étendus pour prendre un compte l'environnement réparti. `fork` permet de créer un processus sur la machine locale ou sur une machine distante. `exec` a été modifié pour permettre la migration de processus lors de l'exécution de nouveau programme. L'appel système `run` combine les effets de `fork` et `exec` et `migrate` permet à un processus de changer de site au cours de son exécution.

Le choix du site d'exécution d'un processus est soumis à plusieurs contraintes :

- l'utilisateur peut spécifier explicitement un ensemble de sites à utiliser pour exécuter son processus ;
- les contraintes matérielles doivent être prises en compte dans le cas des appels `fork` et `migrate` ;
- d'autres contraintes peuvent être ajoutées par l'administrateur du système.

Afin de permettre un nommage unique des processus, Locus les référence par un identificateur formé de deux composantes : un identificateur de site unique et un identificateur de processus unique sur ce site. L'identificateur de site correspond à la machine sur laquelle le processus a été créé et peut donc ne pas correspondre à la machine exécutant le processus si ce dernier a été migré.

4.5 Sprite

Sprite [Ousterhout *et al.* 1988], développé à l'université de Berkeley, est un système réparti basé sur les sémantiques Unix.

Le principal objectif de Sprite est d'offrir une compatibilité parfaite avec le système Unix en offrant la même interface et les mêmes sémantiques. Les fonctionnalités réparties de Sprite permettent à un ensemble de stations de travail connectées par un réseau de partager des ressources (fichiers, processeurs) de manière transparente.

Sprite est bati selon une structure traditionnelle : un noyau unique s'exécute sur chaque station et offre toutes les fonctionnalités du système.

4.5.1 Systèmes de fichiers répartis

Sprite utilise une interface à la Unix pour les accès aux fichiers [Welch 1990]. Tous les accès aux fichiers sont gérés par le noyau : les processus utilisateur effectuent un appel système pour ouvrir les fichiers et obtiennent en retour un descripteur leur permettant d'effectuer des entrées-sorties.

Le noyau d'un site client identifie le site serveur d'un fichier en effectuant une recherche basée sur les premiers caractères du nom du fichier. Il transmet ensuite le chemin d'accès complet au serveur qui se charge de son interprétation et des vérifications de droits d'accès. Le serveur retourne ensuite un descripteur, un nouveau chemin d'accès (dans le cas de l'interprétation de lien symbolique) ou un code d'erreur au client. Le client utilise ensuite ce descripteur pour transmettre les requêtes d'entrées-sorties au serveur.

Sprite intègre des mécanismes de cache visant à assurer la cohérence des fichiers partagés [Nelson *et al.* 1988].

Les caches sont gérés par les serveurs et les clients :

- le serveur maintient un cache des données contenues dans les fichiers ainsi que les informations les concernant (noms, dates de création, attributs...) ;
- les clients maintiennent un cache des données contenues dans les fichiers.

Les écritures sont différées sur les clients et transmises aux caches de serveurs de manière périodique.

Afin d'assurer la cohérence des informations, plusieurs mécanismes sont mis en œuvre :

- lorsqu'un client accède à un fichier, le serveur l'informe de sa date de dernière modification ; si cette date est plus récente que celle contenue dans le cache du client, le cache est purgé ;
- lorsqu'un client ouvre un fichier en écriture, le serveur avertit tous ses clients ayant ouvert le fichier afin qu'ils invalident leurs caches ;
- lorsqu'un client effectue un accès en lecture sur un fichier, le serveur demande au dernier client ayant ouvert le fichier en écriture de lui transmettre toutes les modifications avant de répondre à la requête.

Afin de pouvoir implémenter ces stratégies, le serveur doit maintenir des informations sur les fichiers ouverts, il s'agit donc d'un serveur à état. Un protocole d'interrogation des clients permet à ce serveur de restaurer cet état lors d'une reprise après une panne.

Une étude comparative des performances du système de fichiers répartis de Sprite est présentée dans [Baker *et al.* 1991].

4.5.2 Gestion de processus

Sprite implémente une gestion de processus similaire à celle d'Unix : un processus peut créer une copie de lui-même grâce à l'appel `fork` et remplacer son code par celui d'un programme, i.e. exécuter un autre programme, grâce à l'appel `exec`. De plus, l'appel `fork` possède une option permettant au nouveau processus d'utiliser le même segment de données que son père. Cette option permet d'obtenir la même sémantique que les threads.

Sprite est implémenté selon le modèle “Stations de travail”, les applications sont donc exécutées par défaut sur la station depuis laquelle elles sont lancées. Il est possible pour un processus de demander explicitement de s'exécuter sur un autre site. Un serveur centralisé est chargé de tenir à jour la liste des stations inactives (sans activité clavier ou souris depuis plus de trente secondes), de répartir les applications sur ces stations et de les déplacer lorsque les stations sont à nouveau utilisées de manière interactive.

La migration des processus est rendue transparente aux utilisateurs grâce à l'emploi de processus filtres (*stubs*) assurant la communication entre la station d'origine et les processus distants [Douglis et Ousterhout 1987, Douglis et Ousterhout 1991].

4.5.3 Communications

Dans Sprite, la communication est intégrée dans le système de fichiers en utilisant des "pseudo-périphériques" [Welch et Ousterhout 1988]. Les processus utilisateur peuvent utiliser ces pseudo-périphériques afin de communiquer, de manière synchrone ou asynchrone, via un serveur, par les appels système standards d'entrées- sorties (`read`, `write` et `ioctl`).

4.6 V–System

Le système V [Cheriton 1988], développé à l'université de Stanford, est un projet d'étude des systèmes d'exploitation répartis. Les objectifs de ce système sont :

- fournir un environnement de développement et d'étude des applications réparties ;
- rendre l'utilisation des fonctionnalités réparties facile pour les utilisateurs novices et puissantes pour les utilisateurs initiés ;
- permettre la construction de systèmes plus performants, plus économiques et plus fiables à l'aide de l'environnement réparti.

V est bati sur le modèle client—serveur : le noyau gère principalement les communications entre processus alors que les fonctionnalités de haut niveau (systèmes de fichiers, gestion et protection des ressources...) sont fournies par des processus serveurs s'exécutant au dessus du noyau.

4.6.1 Systèmes de fichiers répartis

L'accès aux fichiers répartis est implémenté par un serveur de fichiers respectant le protocole d'entrées-sorties [Cheriton 1984b]. Ce serveur s'exécute sur une station, possédant une mémoire de masse importante.

Un client qui souhaite accéder à un fichier doit d'abord contacter le serveur de fichiers. Celui-ci crée une instance de serveur qui est chargée de gérer le fichier ouvert. L'identificateur de processus caractérisant cette instance est renvoyé au client qui lui transmet ensuite les requêtes d'entrées-sorties.

De cette façon, l'ordonnancement des accès aux fichiers est assuré par l'ordonnancement des serveurs associés à chaque fichier.

4.6.2 Gestion de processus

Dans V, la gestion des processus et de la mémoire est assurée par un serveur de processus. Ce serveur est implémenté par le noyau lui-même et n'est donc pas un processus à part entière mais plutôt un ensemble de routines de service.

Un processus est identifié par un numéro de processus unique. A chaque processus sont associés une priorité, un état, un espace d'adressage et une pile. Les processus sont organisés en groupes appelés équipes. Tous les processus présents dans une équipe ont accès au même espace d'adressage. Ceci permet un partage du code et des données entre tous les processus de la même équipe mais oblige le système à exécuter tous ces processus sur le même processeur.

V autorise l'exécution de processus sur des stations inactives. Il est possible de spécifier explicitement une station à utiliser ou de laisser le système choisir une station [Theimer *et al.* 1985].

Lorsque le système choisit une station, il diffuse une requête à tous les serveurs de processus s'exécutant dans le système. La réponse de chaque serveur lui indique la quantité de mémoire inutilisée, le pourcentage de temps processeur utilisé récemment (i.e. la charge) et si l'exécution de processus distants est autorisée sur la station.

Lorsque le serveur de processus local a reçu une réponse satisfaisante, il transmet une requête au serveur de processus distant concerné. Ce dernier lance alors l'exécution du processus.

Afin de permettre un nommage uniforme des processus, V construit les identificateurs de processus en prenant en compte la machine où s'exécute le processus et son identificateur local. Ainsi, un identificateur de processus contient le numéro de la machine et celui du processus. De cette façon, le système implémente un espace de nommage global.

V implémente également la migration de processus [Theimer *et al.* 1985]. Cette migration utilise les mêmes stratégies de choix de la machine cible que le placement.

Lorsqu'une station cible a été choisie, le système exécute les étapes suivantes :

- la station source transmet à la station cible les descripteurs du processus afin de lui permettre de créer une copie ; les deux instances du processus sont alors référencées par deux identificateurs différents ;
- l'état du processus est "pré-copié" depuis la station source vers la station cible alors que le processus continue à s'exécuter ; cette "pré-copie" permet de ne pas suspendre le processus pendant la copie [Theimer 1986] ;
- le processus est ensuite suspendu et la copie de son état est terminée ; cette copie consiste à transmettre à la machine cible l'état machine du processus et les pages mémoire modifiées depuis la pré-copie. A l'issue de cette étape, deux instances suspendues du même processus dans le même état existent sur la machine source et sur la machine source ;

- le processus présent sur la machine cible est réveillé et reprend son exécution alors que le processus sur la machine source est supprimé.

5 Conclusion

Nous avons présenté, dans ce chapitre, une étude des systèmes répartis actuels. Nous avons tout d'abord défini les modèles des systèmes répartis selon leur structure puis nous avons différencié les systèmes à architecture répartie qui implémentent une surcouche au dessus de systèmes traditionnels des systèmes répartis qui apparaît comme un système uniprocesseur virtuel.

Nous avons ensuite détaillé les fonctionnalités d'un système réparti en détaillant les problèmes nouveaux qui sont posés. Ces problèmes sont essentiellement causés par :

- les communications par réseau introduisant des délais de transmission et sujettes aux pannes ;
- l'impossibilité pour un processus sur une station du système d'avoir une vision exacte de l'état global du système.

Enfin, nous avons présenté l'implémentation de ces fonctionnalités sous forme d'extensions au système Unix et dans les systèmes répartis majeurs actuels.

Chapitre 3

Architecture fonctionnelle

1 Introduction

Le choix de la structure d'un nouveau système d'exploitation est un problème délicat. En effet, plusieurs structurations peuvent être utilisées. Alors que certains systèmes, dont Locus [Popek et Walker 1985], OSF/1 IK [OSF 1990], Sprite [Ousterhout *et al.* 1988] et Unix [Bach 1986, Leffler *et al.* 1989], sont basés sur un noyau monolithique, d'autres systèmes, dont Amoeba [Mullender *et al.* 1990], OSF/1 MK [Places *et al.* 1992] et Windows NT [Custer 1993], aussi bien issus de la recherche que commerciaux, ont choisi une structure à base de serveurs coopérants au dessus d'un micro-noyau.

Nous étudions tout d'abord chacun de ces modèles en comparant leurs avantages et inconvénients. Nous présentons également les fonctionnalités assurés par quatre micro-noyaux : Amoeba, Chorus, Mach et V puis nous les comparons.

Nous exposons ensuite la structure fonctionnelle de Masix en détaillant les fonctionnalités qu'il offre et la façon dont nous les avons classifiées. Nous définissons les différents composants du système.

Enfin, nous exposons la méthodologie employée lors de la définition du système. Nous justifions notamment la structuration de Masix sous forme de serveurs s'exécutant au dessus d'un micro-noyau et nous expliquons pourquoi nous avons choisi le micro-noyau Mach comme base de notre système.

2 Structure d'un Système d'Exploitation

2.1 Approche Monolithique

De manière traditionnelle, les systèmes d'exploitation ont été conçus sous la forme d'un seul programme. Ce programme, appelé noyau, est chargé de gérer les ressources physiques et de satisfaire les appels système demandés par les processus utilisateur.

Généralement, de tels systèmes ont été conçus comme de petits programmes aux fonctionnalités limitées et ont été étendus au cours du temps. Il est souvent difficile de dégager la structure d'un système d'exploitation implémenté selon cette approche.

Le système Unix [Bach 1986, Leffler *et al.* 1989] constitue un bon exemple de cette approche. Il est constitué de deux parties différentes : le noyau et les programmes système. Le noyau, lui-même, contient un ensemble d'interfaces et de gestionnaires de périphériques, qui ont été ajoutés et modifiés lors de l'évolution d'Unix.

On peut voir Unix comme un ensemble de couches représentées sur la figure 3.1. Le noyau implémente : les systèmes de fichiers, l'ordonnancement des processus, la gestion de la mémoire, et les autres fonctions du système d'exploitation par des appels système. Les programmes système et utilisateur utilisent les appels système fournis par le noyau pour offrir des fonctionnalités de haut niveau à l'utilisateur. Le noyau s'exécute en mode superviseur et la communication entre les processus et le noyau est effectuée en plaçant les paramètres de l'appel système dans la pile et en exécutant ensuite une instruction de trappe permettant de faire continuer l'exécution par une fonction du noyau en mode superviseur. Le noyau examine ensuite les paramètres de l'appel afin d'aiguiller l'exécution sur une fonction de service interne.

Le modèle monolithique offre certains avantages :

- les performances du système sont excellentes dans la mesure où les appels système sont exécutés sous forme de trappes ;
- le regroupement de tous les services système dans le même espace d'adressage leur permet de communiquer aisément entre eux en partageant des données et en s'appelant directement par des appels de sous-programmes.

L'énorme inconvénient de ce principe réside dans les difficultés à maintenir et à étendre un tel système. Comme les divers services système s'appuient les uns sur les autres et sont inter-dépendants, l'ajout ou la modification de service entraîne fréquemment des effets de bord sur les autres et peut même obliger à effectuer une restructuration de l'ensemble du noyau.

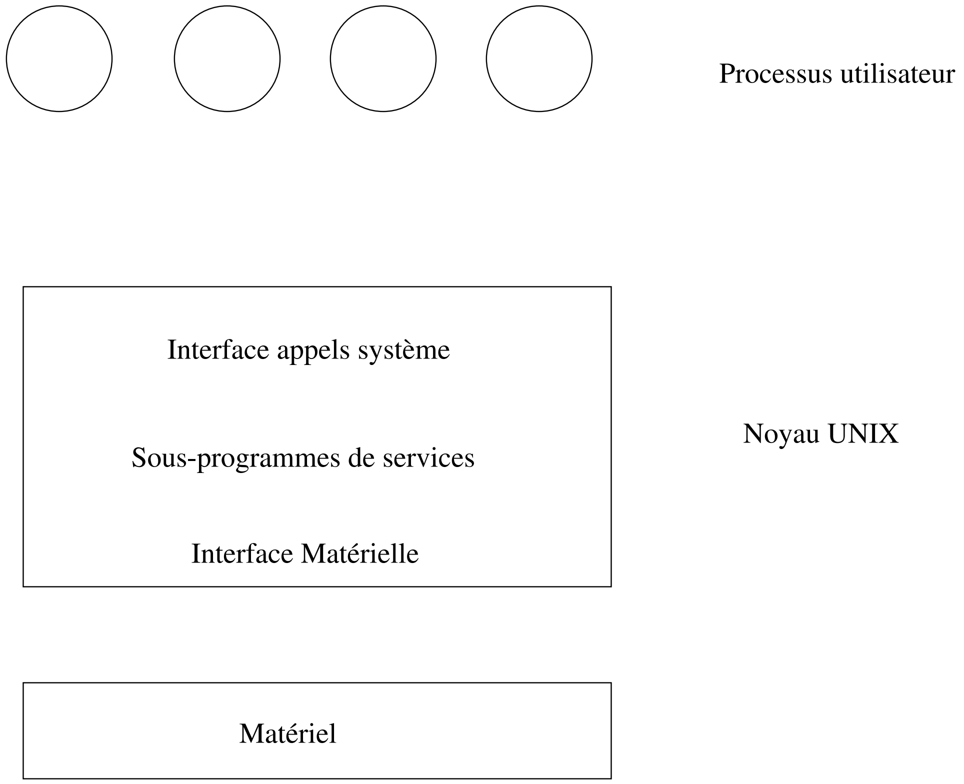

Figure 3.1: Structure du Système Unix

2.2 Systèmes en couches

Une généralisation du principe utilisé dans Unix consiste à organiser le système d'exploitation en un ensemble de couches hiérarchisées, chacune utilisant exclusivement les services offerts par les couches qui lui sont immédiatement inférieures. Chaque couche est implémentée en utilisant uniquement les services offerts par les couches inférieures. Une couche n'a pas besoin de connaître la façon dont ces services sont implémentés ; elle a juste besoin de savoir ce que font ces services. Ainsi, chaque couche cache l'existence de structures de données, opérations internes et mécanismes matériels aux couches de plus haut niveau.

Le système THE [Dijkstra 1968] fut le premier conçu selon ce principe. THE possédait six couches (figure 3.2) :

- la couche 0 était chargée de l'allocation du processeur, en changeant de processus courant lors de l'arrivée d'interruptions ou l'expiration de quantum de temps ; elle offrait donc les fonctionnalités de base de multiprogrammation ;
- la couche 1 s'occupait de la gestion de la mémoire, en allouant de l'espace pour les processus en mémoire centrale et sur un disque utilisé pour les pages de processus non présentes en mémoire ;
- la couche 2 assurait la communication entre chaque processus et la console opérateur ;
- la couche 3 gérait les entrées/sorties ;
- la couche 4 était constituée des programmes utilisateur ;
- la couche 5 exécutait le processus opérateur.

Couche 5	processus opérateur
Couche 4	programmes utilisateur
Couche 3	gestion des entrées-sorties
Couche 2	gestionnaire de la console
Couche 1	gestion de la mémoire
Couche 0	allocation du processeur et multiprogrammation

Figure 3.2: Structure du système THE

La structuration d'un système en couches peut être implémentée de différentes façons. Le système Venus [Liskov 1972] était également structuré en couches : les couches de bas niveau (0 à 4), qui étaient chargées de gérer l'ordonnancement des processus et la gestion de la mémoire, étaient implémentées dans le micro-code du processeur. Ceci permettait d'obtenir une vitesse d'exécution plus importante et une interface bien définie entre les couches micro-codées et les couches de plus haut niveau (voir figure 3.3).

Couche 6	programmes utilisateur
Couche 5	gestionnaire de périphériques
Couche 4	mémoire virtuelle
Couche 3	canaux d'entrées-sorties
Couche 2	ordonnancement
Couche 1	interpréteur d'instructions
Couche 0	matériel

Figure 3.3: Structure du système Venus

Le système Multics [Organick 1972] utilisait une structure similaire. Au lieu de couches, Multics était composé d'un ensemble d'anneaux concentriques, les anneaux les plus internes étant plus privilégiés que les anneaux externes. Quand un sous-programme appartenant à un anneau devait en appeler un autre situé dans un anneau plus privilégié, il devait effectuer l'équivalent d'une trappe via un guichet qui vérifiait la conformité de l'appel avant de l'autoriser. L'anneau 0 était utilisé pour le code et les données accessibles en mode superviseur, les anneaux 1 et 2 contenaient les sous-systèmes et l'anneau 3 était utilisé pour les processus utilisateur.

Il est à noter que l'on peut voir Unix comme un système formé de plusieurs couches [Robboy 1987] comme représenté sur la figure 3.4.

Alors que la décomposition en couches de THE, Unix et Venus constitue uniquement une aide à la conception du système, puisque toutes les parties du système sont combinées en un seul exécutable, le système d'anneaux de Multics est activé lors de l'exécution et renforcé par des contrôles matériels. L'avantage d'un tel mécanisme d'anneaux réside dans ses possibilités d'extension dynamique à des sous-systèmes. Ainsi, si un programme doit appeler des sous-programmes auxquels il ne fait pas confiance, il suffit d'exécuter le programme dans un anneau inférieur à celui de ses sous-programmes afin de le prémunir contre des altérations intentionnelles ou accidentelles de ses données.

2.3 Machines Virtuelles

Un système d'exploitation a pour but d'offrir une interface étendue et des possibilités de multiprogrammation. Il crée l'illusion de plusieurs processus disposant chacun de son propre processeur et leur offre des fonctionnalités avancées, telles que les appels système ou les systèmes de fichiers, qui ne sont pas fournies par la machine physique.

Le principe des machines virtuelles consiste à séparer ces deux fonctions.

Le cœur d'un système conçu selon le principe des machines virtuelles, appelé moniteur de ma-

Gestion du réseau Gestion des systèmes de fichiers Interface des appels système Communications entre processus	Indépendant de la machine
Ordonnancement des processus Gestion des régions mémoire	Assez indépendant de la machine
Chargement de programmes Gestion des signaux Gestion de la mémoire virtuelle	Peu dépendant de la machine
Changements de contexte Gestion de la mémoire physique	Dépendant de la machine
Gestion du matériel Gestion des périphériques Gestion des exceptions et des interruptions	Fortement dépendant de la machine

Figure 3.4: Décomposition en couches du système UNIX

chine virtuelle, gère la machine physique en offrant des fonctionnalités de multiprogrammation et fournit l'interface de plusieurs machines virtuelles. Ces machines virtuelles, contrairement à d'autres systèmes d'exploitation, ne possèdent pas d'interface étendue mais sont les répliques exactes de la machine physique dont elles offrent les mêmes fonctionnalités (mode d'exécution utilisateur/superviseur, entrées-sorties, interruptions...).

Dans la mesure où chaque machine virtuelle est identique à la machine physique, elle peut exécuter un système d'exploitation gérant directement le matériel. Ce système ne fournit pas de multiprogrammation mais uniquement des fonctionnalités étendues à un seul processus.

Quand un processus utilisateur effectue un appel système, une trappe est traitée par le système gérant la machine virtuelle sur laquelle il s'exécute. Ce système exécute alors une instruction d'entrées-sorties pour accéder à sa machine virtuelle. Cette instruction donne lieu à une trappe traitée par le moniteur de machine virtuelle qui émule l'opération demandée sur la machine physique.

Le système VM [Seawright et MacKinnon 1979] constitue un bon exemple de cette approche. Dans ce système, VM fournit un ensemble de machines virtuelles, chacune exécutant CMS qui implémente l'interface utilisateur. La structure de ce système est représentée sur la figure 3.5.

En offrant une séparation complète des fonctions de multiprogrammation et de l'interface étendue, chacune des parties du système peut être rendue plus simple et plus facile à étendre et

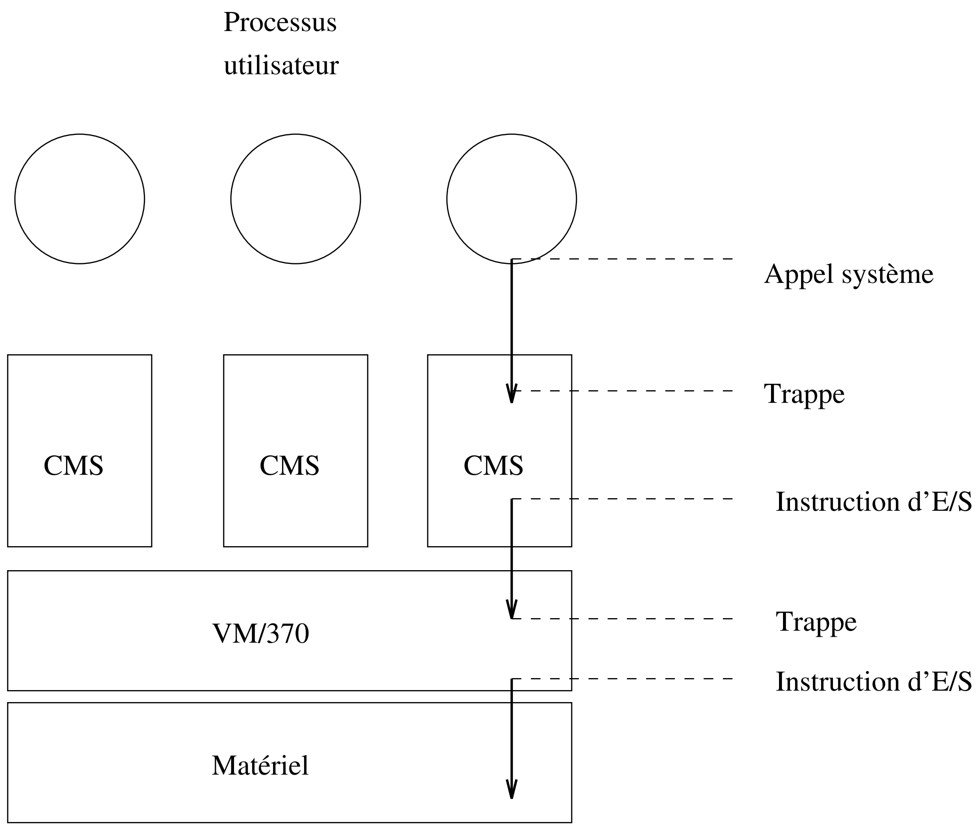

Figure 3.5: Structure du système VM

maintenir.

2.4 Micro-Noyaux

Une tendance actuelle dans la conception de systèmes d'exploitation consiste à répartir les fonctions du système dans des couches logicielles hautes, en conservant un noyau minimal. La plus grande partie du système est alors implémentée par des programmes qui s'exécutent en mode utilisateur.

Afin de demander un service, un processus envoie une requête à un processus serveur qui effectue le traitement demandé et qui renvoie le résultat au processus. Les serveurs utilisent les fonctionnalités de bas-niveau fournies par le noyau, qui se limite à gérer la machine physique et à offrir des mécanismes de communications entre processus.

L'utilisation d'un micro-noyau permet une grande simplicité et l'extensibilité du système. Chaque serveur est relativement simple et peut être mis au point indépendamment des autres puis intégré dans le système lorsqu'il fonctionne. Il est également aisé d'ajouter des serveurs sans modifications du reste du système. De plus, comme les serveurs s'exécutent en mode utilisateur, ils ne peuvent pas accéder à la machine physique directement. Ainsi, si un serveur contient une erreur, le service qu'il assure sera indisponible mais le reste du système ne sera pas affecté.

Un autre avantage de cette approche est sa facilité d'utilisation dans le cadre de systèmes répartis. Lorsqu'un client communique par messages avec un serveur, il n'a pas besoin de savoir si sa requête est traitée sur la machine locale ou transmise à un serveur sur une machine distante.

L'inconvénient majeur de ce type de système réside dans les performances. Un appel système est implémenté sous forme de requêtes à un serveur et donne donc lieu à un ou plusieurs changements de contexte pour être traité. Cette méthode est beaucoup plus coûteuse que le schéma classique à base de trappe et d'appel de sous-programme.

Des études [Dean et Armand 1992, Douglis *et al.* 1991] ont néanmoins montré qu'il est possible d'obtenir des performances voisines de celles des systèmes monolithiques.

La structure de quatre micro-noyaux (Amoeba, Chorus, Mach et V–Kernel) est détaillée dans les sections suivantes.

2.4.1 Amoeba

Amoeba [Mullender *et al.* 1990] est composé d'un micro-noyau et de processus serveurs offrant des fonctionnalités de haut niveau. Le micro-noyau offre les services suivants :

- gestion de processus ;
- gestion de la mémoire ;

- mécanismes de communication de messages entre processus ;
- gestion des périphériques.

Amoeba supporte le concept de processus : un processus est considéré comme une unité d'allocation de ressources. Chaque processus peut contenir plusieurs *threads* qui représentent des unités d'exécution. Un processus est caractérisé par son espace d'adressage mémoire alors qu'un *thread* est caractérisé par le processus qui le contient, la valeur de ses registres, son compteur ordinal et la pile qu'il utilise.

Le micro-noyau fournit une gestion simple de la mémoire. Un *thread* peut allouer et désallouer des blocs de mémoire, appelés segments. Le micro-noyau gère ces segments en les plaçant dans l'espace d'adressage du processus contenant le *thread*. Pour ce faire, il utilise les mécanismes de pagination fournis par le processeur mais n'implémente pas de gestion de mémoire virtuelle : tout segment utilisé est chargé en mémoire dans son intégralité.

La communication entre processus repose sur la notion de port. Un port est un canal de communication sur lequel un processus peut envoyer des messages. Amoeba fournit des primitives de communication permettant d'effectuer des appels de type RPC. Un mécanisme de communication à un groupe est également fourni : le micro-noyau fournit des primitives permettant de créer des groupes logiques de processus auxquels des requêtes peuvent être adressées.

Amoeba contient les gestionnaires de périphériques. Ces modules font partie du micro-noyau et sont rendus accessibles aux processus utilisateur par des ports. Un *thread* qui communique avec un gestionnaire de périphériques utilise les mêmes fonctions que pour s'adresser à un autre processus.

2.4.2 Chorus

Chorus [Rozier *et al.* 1988] est un micro-noyau conçu pour l'implémentation de systèmes d'exploitation répartis sous forme de serveurs [Albinson *et al.* 1991]. Le micro-noyau de Chorus fournit les fonctionnalités suivantes :

- gestion de processus ;
- communication inter-processus ;
- gestion de la mémoire virtuelle ;
- gestion des périphériques.

Les processus sont gérés à l'aide de deux abstractions : les acteurs et les *threads*. Un acteur représente un ensemble de ressources comportant plusieurs *threads* qui partagent les ressources de l'acteur [Armand *et al.* 1990]. Chorus définit trois types d'acteurs : les acteurs utilisateurs, les acteurs systèmes et les acteurs superviseurs. Un acteur utilisateur ne possède aucun privilège et le micro-noyau ne lui fait pas confiance. Un acteur système ne possède aucun privilège mais le micro-noyau lui fait confiance. Enfin, un acteur superviseur s'exécute en mode superviseur et partage l'espace d'adressage du micro-noyau.

La communication entre *threads* contenus dans des acteurs est assurée par le biais de ports. Un port est une file de messages nommée par un identificateur unique. Chorus offre des primitives permettant de transmettre des messages sur les ports. Les ports peuvent être regoupés afin de transmettre une même information à plusieurs *threads* en une seule opération et obtenir ainsi l'équivalent d'une diffusion.

Les mécanismes de communication permettent une communication asynchrone et des appels de procédures à distance. Lors de l'envoi d'un message asynchrone, l'émetteur n'est pas bloqué. Lors d'un appel de procédure à distance, par contre, l'émetteur est suspendu jusqu'à ce que le récepteur ait reçu son message et y ait répondu.

Le gestionnaire de mémoire de Chorus gère des espaces d'adressage séparés associés aux acteurs, appelés contextes. Un contexte est composé d'un ensemble de régions disjointes. Chaque région correspond généralement à un objet situé en mémoire secondaire, appelé segment. Des acteurs externes au micro-noyau, appelés *mappers*, implémentent un protocole d'entrées-sorties sur les segments et permettent de charger ou de sauvegarder des pages [Abrossimov *et al.* 1989].

Afin de permettre l'accès aux périphériques, Chorus permet à des acteurs superviseur d'associer des fonctions à des interruptions. Lorsqu'une interruption est déclenchée, le micro-noyau appelle, l'une après l'autre, chacune des fonctions associées à l'interruption. Chorus offre également un ensemble de primitives utilisables par un acteur superviseur pour accéder aux périphériques physiques [Armand 1991]. Grâce à ces mécanismes, il est possible d'implémenter des gestionnaires de périphériques sous forme d'acteurs en dehors du micro-noyau lui-même.

2.4.3 Mach

Mach [Tevanian et Rashid 1987] est un micro-noyau conçu pour servir de base à l'implémentation de divers systèmes d'exploitation. Nous présentons dans cette section ses fonctionnalités de base. Devant l'importance de Mach dans notre travail, le chapitre 4 lui est consacré.

Mach fournit les fonctionnalités suivantes :

- gestion de processus ;
- gestion de communications entre processus ;
- gestion de la mémoire physique et virtuelle,
- gestion des périphériques.

Mach offre deux abstractions pour représenter les processus : les tâches et les *threads*. Une tâche est une entité passive représentant un ensemble de ressources allouées. Un *thread* est une entité active, contenue dans une tâche, qui s'exécute.

Les communications fournies par Mach reposent sur la notion de port. Un port est une file dans laquelle des messages peuvent être ajoutés et retirés. Les ports sont protégés par des droits gérés par le micro-noyau. Mach fournit de nombreuses options d'émission et de réception permettant d'assurer plusieurs sémantiques différentes (émission de messages sans attente de réponse, appel de procédure à distance...).

Mach gère l'espace d'adressage mémoire associé à chaque tâche en utilisant les fonctionnalités fournies par le processeur. Il traite les défauts de page, gère la table des pages mais ne s'occupe pas du chargement ou de la sauvegarde de pages sur les périphériques. Pour ces tâches, il fait appel à des tâches utilisateur, appelées gestionnaires de mémoire, selon un protocole normalisé.

Les gestionnaires de périphériques sont inclus dans le micro-noyau et sont accessibles aux processus par le biais de ports. Un protocole d'entrées-sorties permet aux processus d'effectuer des opérations de bas niveau sur les périphériques.

2.4.4 V–Kernel

Le V–Kernel [Cheriton 1984a] constitue un micro-noyau fournissant des fonctionnalités de base :

- gestion de processus et de la mémoire ;
- communications inter-processus ;
- gestion de périphériques.

V fournit la notion de processus léger. Un processus représente une unité d'exécution. A chaque processus est associé une équipe. Plusieurs processus peuvent appartenir à la même équipe et partagent dans ce cas le même espace d'adressage. Le regroupement de processus en équipe fournit une sémantique proche de celle des *threads* présents dans les autres micro-noyaux.

V offre trois types de communication entre processus :

1. échange de messages :

   des primitives d'envoi et de réception peuvent être utilisées pour transmettre des messages de taille peu importante (32 octets). La communication est directe, i.e. V n'offre pas de notion de port mais impose à un processus de citer explicitement le destinataire lors de l'envoi ;

2. transfert de données :

   pour la transmission de messages plus longs que 32 octets, V permet au processus destinataire d'effectuer une copie d'une partie de l'espace d'adressage de l'émetteur pendant que ce dernier attend un message de réponse. Le récepteur peut effectuer cette copie dans les deux sens, i.e. lire et écrire des données appartenant à l'émetteur ;

3. diffusion :

   les processus peuvent se regrouper en groupes logiques pour recevoir des messages. Un processus peut émettre des messages à destination d'un groupe et tous les membres de ce groupe les reçoivent. L'émetteur est alors bloqué jusqu'à ce qu'un des membres du groupe lui envoie un message de réponse.

Le micro-noyau gère les périphériques par le biais d'un processus qui s'exécute en mode superviseur et qui peut recevoir des messages de requêtes. Un processus souhaitant accéder à un périphérique doit d'abord acquérir l'identificateur du processus gestionnaire puis lui transmettre une requête par les primitives de communication exposées plus haut. Ces requêtes doivent respecter le protocole d'entrées-sorties implémenté par le micro-noyau [Cheriton 1984b].

2.4.5 Synthèse

L'analyse de ces quatre micro-noyaux montre qu'ils partagent de nombreux points communs aussi bien au niveau des abstractions fournies qu'au niveau des fonctionnalités offertes.

Chacun offre la notion habituelle de processus et celle de processus léger (*threads* dans Amoeba, Chorus et Mach, processus d'une même équipe dans V).

Les communications sont, dans tous les cas, implémentées sous forme de transmission, directe (en citant le processus cible) ou indirecte (en citant un port), de messages. Chacun des micro-noyaux présentés offre des fonctionnalités puissantes permettant aux processus de communiquer selon plusieurs sémantiques.

A l'exception d'Amoeba, qui a choisi de ne pas implémenter de mémoire virtuelle afin d'améliorer les performances d'exécution, les micro-noyaux fournissent une gestion de la mémoire performante dans laquelle une partie des responsabilités est déléguée à des processus externes. Cette utilisation de paginateurs externes permet une grande souplesse d'implémentation des stratégies de gestion de la mémoire.

Les quatre micro-noyaux permettent à des processus d'accéder aux périphériques physiques de la machine. Alors que Amoeba, Mach et V intègrent les gestionnaires dans le noyau et offrent un protocole de communication, Chorus a choisi de rendre directement accessible l'accès physique aux contrôleurs en permettant à des acteurs de spécifier des traitements liés aux interruptions et d'agir sur les périphériques physiques. Cette approche a l'avantage de permettre l'ajout dynamique de nouveaux gestionnaires de périphériques mais oblige les concepteurs de ces gestionnaires à une programmation de très bas niveau.

3 Décomposition Système local / Système virtuel

3.1 Fonctionnalités de Masix

Masix est un système d'exploitation réparti devant offrir plusieurs niveaux de fonctionnalités :

- interface de programmation implémentant les sémantiques Unix ;
- système d'accueil permettant une intégration facile d'interfaces différentes et indépendantes accessibles en parallèle par plusieurs applications ;
- fonctionnalités réparties permettant des accès implicites ou explicites et un partage des ressources sur un réseau de stations.

3.2 Décomposition fonctionnelle

Un système d'exploitation peut être considéré comme un ensemble de couches [Silberschatz *et al.* 1991, Tanenbaum 1992] :

- la couche la plus interne, le noyau, gère la machine physique et offre différents appels systèmes utilisables par les processus utilisateur ;
- au dessus du noyau, les bibliothèques offrent les communications entre processus et noyau afin de permettre les appels système. Elle implémentent, de plus de nombreuses fonctionnalités de haut niveau effectuées sans intervention du noyau ;
- enfin, des programmes système fournissent des outils et services et implémentent l'interface utilisateur.

Nous effectuons ici la décomposition fonctionnelle du noyau de Masix en analysant les différentes fonctionnalités qu'il doit offrir.

3.2.1 Fonctionnalités Unix

Afin de dégager la structure de Masix, nous avons tout d'abord ré-examiné les appels systèmes offerts par Unix.Pour cela, nous nous sommes basés sur :

- les fonctionnalités exigées par le standard Posix [IEEE 1990a, Lewine 1991] qui définit un ensemble de fonctions, de codes d'erreurs et de sémantiques devant être offerts par tout système compatible avec Unix. Cette norme a été élaborée en prenant en compte les fonctionnalités communes à toutes les variantes d'Unix actuelles et correspond donc à un sous-ensemble des fonctions offertes par un système Unix moderne ;

- de nombreuses fonctionnalités implémentées dans la version BSD de Unix [Quaterman *et al.* 1985] afin de compléter le sous-ensemble obtenu à l'étape précédente. Les fonctions qui nous semblaient utiles et qui ne posaient pas de problème de compatibilité avec Posix ont ainsi été incluses dans cette liste ;
- enfin, certaines facilités présentes dans la version System V de Unix [AT&T 1989] ont été prises en compte dans la mesure oú elles n'étaient ni incompatibles ni redondantes avec les deux catégories précédentes.

Nous avons ensuite classifié les fonctionnalités recensées en plusieurs catégories logiques :

- fichiers :

  cette catégorie contient l'ensemble des appels système permettant d'agir sur des fichiers en citant leur nom. Ces opérations sont généralement ponctuelles ;
- entrées-sorties ;

  appels système permettant d'effectuer des entrées-sorties ou des opérations de contrôle sur des fichiers ouverts. Ces opérations agissent sur un objet référencé par un descripteur obtenu du système via une requête d'ouverture.
- processus :

  appels système permettant de contrôler des processus selon les sémantiques Unix. Ces opérations sont notamment :

  - création et terminaison de processus ;
  - obtention et modification d'informations associées à des processus ;
  - exécution de programme ;
  - opération de contrôle sur des processus ;
- signaux :

  appels système permettant de traiter les signaux Unix ;
- mémoire :

  appels système d'allocation, de désallocation et de modification d'attributs de régions mémoire ;
- communications :

  appels système permettant de communiquer de manière locale entre processus apparentés (tubes) qu'entre processus quelconques sur des machines distinctes (*sockets* originaires de Unix BSD) ;
- communications System V :

  appels système offrant les mécanismes de communication locaux entre processus à base de mémoire partagée, files de messages et sémaphores ;

- configuration et administration du système :

  appels système permettant d'obtenir et de modifier des paramètres du système et de contrôler son exécution ;

- divers :

  appels système ne correspondant à aucune des catégories précédentes.

3.2.2 Système d'accueil et interfaces

L'environnement de travail d'un utilisateur est constitué :

- l'interface offerte par le système ;
- les abstractions, leurs sémantiques et les opérations de manipulation fournies par le système.

Afin de permettre à plusieurs environnements de cohabiter dans le système de manière simultanée, Masix doit offrir des services de gestion des interfaces.

Chaque environnement offre des abstractions, des sémantiques et une interface qui lui sont propres. Tout processus peut utiliser différents objets et leurs sémantiques selon l'interface à laquelle il s'adresse. Il est ainsi possible d'avoir plusieurs visions du système de manière simultanée.

La gestion dynamique des environnements nécessite les services suivants :

- enregistrement d'un environnement :

  toute nouvelle interface doit être enregistrée dans le système afin de pouvoir être référencée ;

- nommage d'une interface :

  tout processus doit pouvoir accéder à une interface existante en demandant au système le moyen de communiquer avec l'interface pour transmettre des requêtes à l'environnement correspondant. Le système doit donc fournir des outils de nommage des environnements enregistrés ;

- terminaison d'un environnement :

  un environnement inutilisé doit pouvoir être supprimé dynamiquement du système.

De plus, afin de simplifier la conception et la réalisation des environnements, un ensemble de services génériques doit être fourni. Cette couche est chargée de fournir des fonctions utilitaires indépendantes de l'interface. Ainsi, chacun des environnements peut implémenter ses abstractions et sémantiques en s'appuyant au maximum sur les services génériques communs.

3.2.3 Répartition

La possibilité d'utiliser des ressources réparties sur un ensemble de machines par des accès implicites nécessite une intégration des services distribués dans le système. En effet, les primitives d'accès aux objets doivent être capables de diriger les requêtes vers des gérants d'objets locaux ou vers des gérants d'objets distants.

Les accès explicites, par contre, ne nécessitent qu'un ensemble de services parallèles aux services locaux.

3.2.4 Architecture

Afin d'atteindre les objectifs que nous nous sommes fixés, nous avons décomposé la structure de Masix en plusieurs composants :

- le noyau de base :

  cet ensemble de services gère la machine physique ;

- le système d'accueil :

  ce sous-ensemble du système contient l'ensemble des fonctions de gestion des environnements. Il fournit :

  - les services génériques de bas niveau utilisables par tous les environnements ;
  - les mécanismes de gestion des environnements enregistrés ;
  - les communications entre applications utilisateur et environnements ;

- le système local de base :

  ce sous-ensemble est formé des différents environnements enregistrés dans le système d'accueil. Les abstractions, sémantiques et interfaces fournies par ces environnements offrent les fonctionnalités d'un système d'exploitation centralisé gérant un ordinateur ;

- les mécanismes de partage de ressources entre sites :

  cette couche fournit les fonctions de partage de ressources entre stations exécutant le système local de base. Ces fonctions offrent des mécanismes de communications intermachines, d'accès à des fichiers distants, d'exécution répartie, de gestion des pannes... ;

- le système virtuel :

  ce système est formé de la coopération des services de partage de ressources qui s'exécutent au dessus de chaque système local. Il apparaît aux utilisateurs comme un seul système centralisé virtuel qui cumule les ressources de chacun des sites connectés.

La figure 3.6 représente la structure de ces composants et leurs relations.

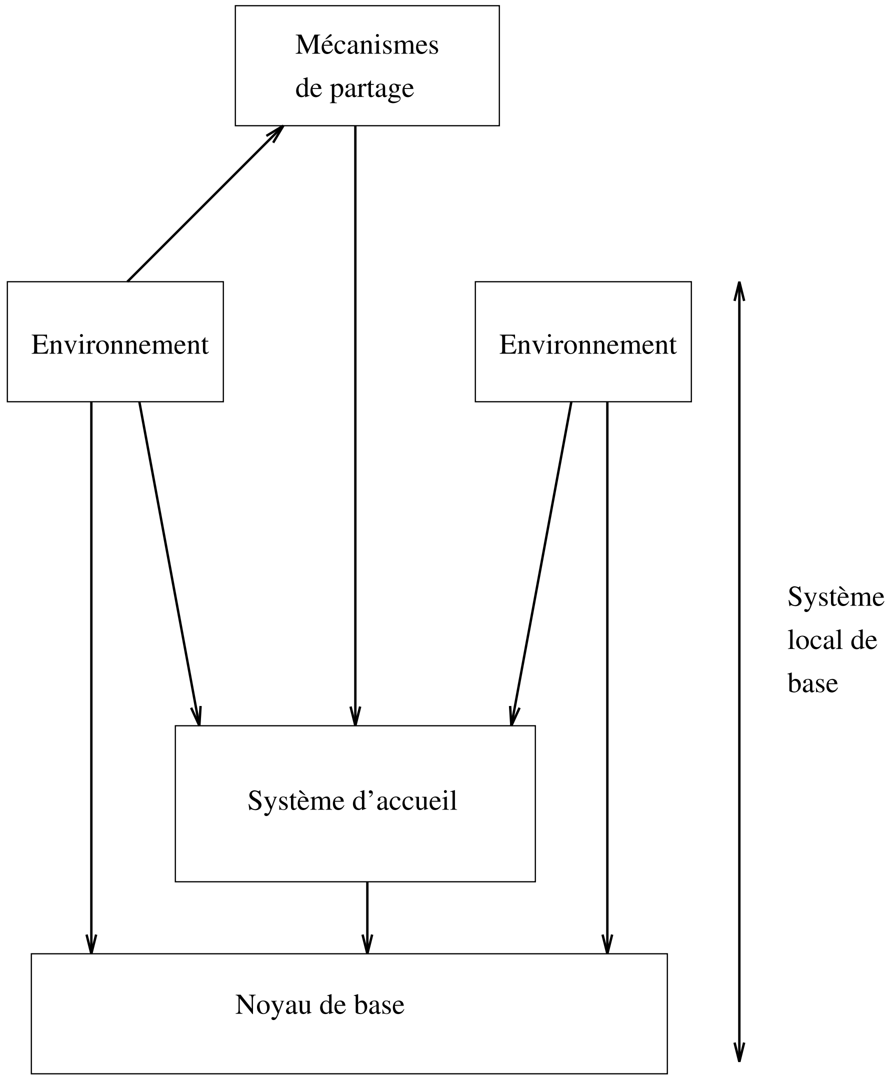

Figure 3.6: Organisation des composants de Masix

4 Principes et Méthodologie

Nous avons décomposé Masix en plusieurs ensembles de services. Cette décomposition repose sur des regroupements logiques de services mais ne correspond pas forcément à l'architecture opérationnelle, i.e. la structure des modules logiciels composant Masix et leurs interactions.

4.1 Noyau de base

Les fonctionnalités offertes par le noyau de base (gestion des périphériques, gestion de la mémoire, ordonnancement de processus, gestion des interruptions et exceptions) correspondent aux services usuellement assurés par les couches basses d'un système [Silberschatz *et al.* 1991, Tanenbaum 1992].

La structuration et l'implémentation de ces services ont été largement explorées dans les systèmes d'exploitation depuis leurs origines et ne présentent que peu d'intérêt dans le contexte de notre travail. Nous avons donc choisi de réutiliser une base logicielle existante offrant les fonctionnalités de bas niveau.

Les micro-noyaux que nous avons présentés dans la section 2.4 implémentent tous ces fonctionnalités. Il nous est donc apparu intéressant de réutiliser l'un de ces micro-noyaux afin de bâtir Masix sous forme de processus s'exécutant en mode utilisateur. L'utilisation d'un micro-noyau nous permet, de plus, de concevoir un système évolutif et facile à maintenir.

Nous avons choisi de concevoir et d'implémenter Masix au dessus de Mach [Tevanian et Rashid 1987]. En effet, Mach nous semble posséder certains avantages par rapport aux autres micro-noyaux :

- il est disponible sur un grand nombre de plate formes matérielles : il existe actuellement des versions de Mach adaptées aux processeurs i386, 68000, 88000, Mips et des portages sur d'autres architectures sont en cours (notamment sur les processeurs Alpha de DEC et Sparc de Sun). Cette disponibilité nous parait très importante dans le cadre d'un système réparti puisqu'elle permet d'envisager l'exploitation de stations de travail hétérogènes ;
- il est disponible gratuitement depuis l'université de Carnegie Mellon contrairement aux autres qui nécessitent des accords et une licence pour être utilisés et ne peuvent pas être redistribués comme une partie d'un projet ;
- de nombreux systèmes ont été conçus au dessus de Mach et il est possible de tirer les enseignements des différents choix qui ont déjà été effectués ;
- comme Mach est distribué librement sous forme de programmes source, de nombreuses organisations ont pu l'utiliser et soumettre à ses auteurs leurs impressions, des améliorations ainsi que des corrections de bogues. Il a été ainsi possible à Mach d'évoluer plus rapidement que d'autres micro-noyaux à distribution plus restreinte.

Afin de situer plus précisément notre travail, une étude détaillée du micro-noyau Mach est présentée dans le chapitre 4.

4.2 Couches logicielles hautes

Les couches logicielles situées au dessus du noyau de base sont conçues en utilisant les services offerts par Mach. Nous nous sommes efforcés lors de leur définition d'utiliser au mieux les mécanismes fournis par le micro-noyau en respectant sa philosophie. Dans la mesure où les abstractions et sémantiques offertes par les autres micro-noyaux sont très semblables à celles de Mach, le portage de Masix sur un autre micro-noyau serait néanmoins possible avec un minimum d'effort.

5 Conclusion

Dans ce chapitre, nous avons étudié les différentes structurations que l'on peut utiliser lors de la conception d'un système d'exploitation. Nous avons décrit les avantages et inconvénients respectifs des approches monolithiques, à base de machines virtuelles et de serveurs coopérants au dessus d'un micro-noyau.

Nous avons également présenté la décomposition fonctionnelle de Masix en présentant l'ensemble des opérations qu'il doit offrir aux processus utilisateur. Les fonctionnalités du système ont été décomposées en plusieurs catégories :

- le système d'accueil offrant les services de gestion d'environnements d'exécution multiples ;
- les services assurant les fonctionnalités Unix.

À partir de cette répartition de services, nous avons distingué les composants nécessaires au système en les classant par couches. Nous avons choisi d'utiliser le micro-noyau Mach pour réaliser le noyau de base, implémentant les fonctionnalités de bas niveau. Les couches de plus haut niveau, implémentant le système d'accueil et les services Unix, seront implémentées sous formes de serveurs coopérants au dessus de Mach.

Le choix du micro-noyau Mach comme base du système Masix permet :

- de ne pas avoir à gérer les mécanismes de bas niveau du système (gestion de la machine physique, ordonnancement des processus, gestion de la mémoire...) et de simplifier ainsi l'écriture de Masix ;
- d'atteindre l'objectif de portabilité que nous nous étions fixé : dans la mesure où Masix ne dépend que de Mach pour accéder à la machine physique, son portage sur une nouvelle architecture est quasi immédiat si Mach lui-même a déjà été porté sur la machine.

Nous présentons plus en détail les services offerts par Mach dans le chapitre 4. La structuration de Masix au dessus de Mach est exposée dans les chapitres 5 et 6.

Chapitre 4

Utilisation du micro-noyau Mach

1 Introduction

Le micro-noyau Mach a été conçu afin de permettre l'implémentation de systèmes d'exploitation sous forme d'un ou plusieurs serveurs s'exécutant en mode utilisateur. Mach gère la machine physique et implémente tous les mécanismes de base d'un système en fournissant un nombre réduit d'abstractions aux serveurs. Ces derniers peuvent utiliser ces abtractions et les primitives offertes par Mach pour implémenter les sémantiques du système d'exploitation à réaliser.

Nous présentons tout d'abord les services offerts par Mach en détaillant les abstractions et les primitives qu'il met à la disposition du concepteur de système d'exploitation. Dans cette partie, nous étudions les mécanismes de communication, les tâches et *threads*, la gestion de la mémoire et des dispositifs matériels.

Nous nous intéressons ensuite aux outils fournis au développeur de serveurs au dessus de Mach et nous exposons la façon dont ils peuvent être utilisés pour implémenter un système comme Masix.

Enfin, nous effectuons une analyse comparative de plusieurs systèmes d'exploitation conçus au dessus de Mach. Nous détaillons la structure et les stratégies retenues dans un système monolithique (OSF/1 IK), deux systèmes mono-serveurs (OSF/1 MK et BSD Single Server) et trois systèmes multi-serveurs (Unix Multi Serveur, GNU Hurd et GUIDE). A l'issue de la présentation de ces systèmes, nous analysons leurs différences, leurs avantages et leurs inconvénients respectifs.

2 Services offerts par Mach

2.1 Objectifs de Mach

Le micro-noyau Mach [Tevanian et Rashid 1987], développé à l'Université de Carnegie Mellon, a été conçu comme une base pour la conception de systèmes d'exploitation. Mach constitue l'aboutissement des travaux effectués dans le cadre des systèmes RIG et Accent [Rashid 1986].

Mach a été conçu pour :

- fournir une base à l'implémentation d'autres systèmes ;
- supporter des espaces d'adressage mémoire importants et clairsemés ;
- offrir un accès transparent aux ressources locales ou distantes ;
- exploiter au maximum le parallélisme dans le système et les applications ;
- être portable.

2.2 Fonctionnalités de Mach

Mach est un micro-noyau et, à ce titre, il ne fournit pas toutes les fonctionnalités traditionnellement offertes par un système. Il offre uniquement :

- la gestion de tâches et de *threads*,

  les tâches et *threads* constituent le support de l'implémentation de processus dans un système ;
- les communications entre tâches,

  les ports et messages fournis par Mach permettent d'implémenter des systèmes utilisant le modèle client-serveur ;
- la gestion de la mémoire physique et virtuelle ;
- la gestion des périphériques physiques.

De nombreuses fonctionnalités, assurées traditionnellement par un système d'exploitation, ne sont pas implémentées par Mach : gestion de systèmes de fichiers, gestion de processus, gestion de terminaux, chargement et écriture de pages, gestion du réseau...

2.3 Abstractions de Mach

Mach fournit cinq abstractions fondamentales :

1. les ports :

   canaux de communication unidirectionnels accessibles via des droits d'émission et de réception ;

2. les messages :

   ensembles de données typées transmis sur les ports ;

3. les tâches (*tasks*) :

   unités d'allocation de ressources (espace d'adressage, ensemble de ports accessibles) ;

4. les activités (*threads*) :

   unités d'exécution ;

5. les objets mémoire (*memory objects*) :

   unités internes de gestion de la mémoire.

Mach fournit également certaines abstractions secondaires :

- les ensembles de ports (*port sets*) ;
- les processeurs (*processors*) ;
- les ensembles de processeurs (*processor sets*) ;
- les hôtes (*hosts*) ;
- les périphériques (*devices*).

2.4 Ports et messages

Mach offre des mécanismes puissants de communication entre tâches [Draves 1990]. Les communications sont effectuées par des ports. Un port est simplement une file dans laquelle des messages peuvent être ajoutés et retirés. Les opérations sur les ports sont effectuées via des droits sur ces ports qui sont accordés aux tâches. Trois types de droits existent :

1. droit de réception, accordé à une seule tâche ;
2. droits d'émission, accordés à plusieurs tâches ;
3. droits d'émission unique, permettant à des tâches d'envoyer un — et un seul — message sur ce port.

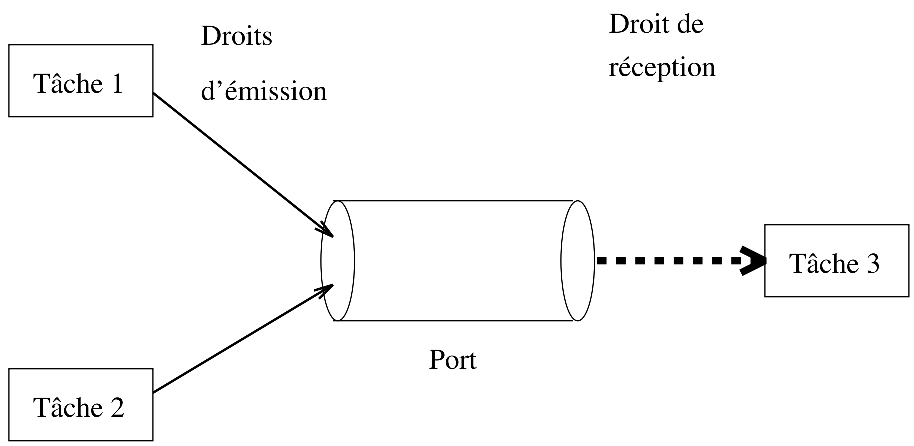

Figure 4.1: Communication par ports dans Mach

Chaque port est géré par une seule tâche qui possède le droit de réception sur ce port. Certains ports privilégiés sont gérés directement par le noyau Mach lui-même.

Les tâches accèdent aux ports via des noms de ports (identificateurs numériques) qui sont convertis de manière interne en droits par le noyau.

Une tâche peut envoyer ou recevoir des messages sur des ports. Un message est simplement une structure contenant des données. Un message est composé :

- d'une en-tête décrivant le message : taille du message, nom du port d'émission, nom du port de réception, type du message, code opération ;
- d'un ensemble de données typées : type de données, nombre de données, valeurs.

Mach offre de nombreuses options pour l'envoi et la réception de messages. Quand un message est envoyé sur un port, la file correspondante peut être pleine. Si elle n'est pas pleine, le message est copié dans la file. Si la file est pleine, la tâche émettrice a le choix entre quatre options :

1. attendre indéfiniment qu'un message soit lu depuis la file :

   la tâche émettrice est suspendue jusqu'à ce qu'une autre lise un message depuis le port et libère ainsi la place nécessaire au stockage du message dans la file ;

2. attendre en spécifiant un délai :

   la tâche émettrice est alors suspendue jusqu'à ce qu'une autre lise un message depuis le port et libère ainsi la place nécessaire au stockage du message dans la file ou jusqu'à ce qu'un délai spécifié soit écoulé ;

3. ne pas attendre,

le message n'est pas ajouté à la file si elle est pleine et la tâche émettrice est prévenue par un code d'erreur ;

4. transmettre le message à Mach,

   le noyau Mach stocke le message et se charge de l'ajouter dans la file quand une place est libérée ; un seul message peut ainsi être transmis à Mach pour une file pleine par tâche.

Mach offre de nombreux appels système permettant :

- d'allouer ou désallouer des ports ;
- de transmettre des droits sur ports en :
  - copiant un droit d'émission pour une autre tâche, les deux tâches possèdent alors ce droit ;
  - offrant un droit d'émission ou de réception à une autre tâche, seule la deuxième tâche possède alors ce droit ;
  - transmettant un droit de réception à une autre tâche en le convertissant en droit d'émission, la première tâche possède alors un droit de réception et la deuxième un droit d'émission sur le même port ;
- d'associer des noms aux ports ;
- d'envoyer et recevoir des messages sur des ports.

2.5 Ensembles de ports

Mach offre la possibilité de regrouper plusieurs droits de réception sur des ports en un ensemble de ports. Une tâche peut regrouper plusieurs ports en un tel ensemble et recevoir des messages sur cet ensemble. Dans ce cas, le premier message reçu sur l'un des ports faisant partie de l'ensemble est transmis à la tâche.

Cette possibilité est particulièrement utile dans le cas d'un serveur chargé de gérer plusieurs objets représentés par des ports : un ou plusieurs *threads* peuvent se placer en attente sur un ensemble de ports et être réveillé lors de la réception de la première requête.

2.6 Tâches et *threads*

Mach offre deux abstractions pour représenter des programmes en cours d'exécution : les tâches et les *threads*.

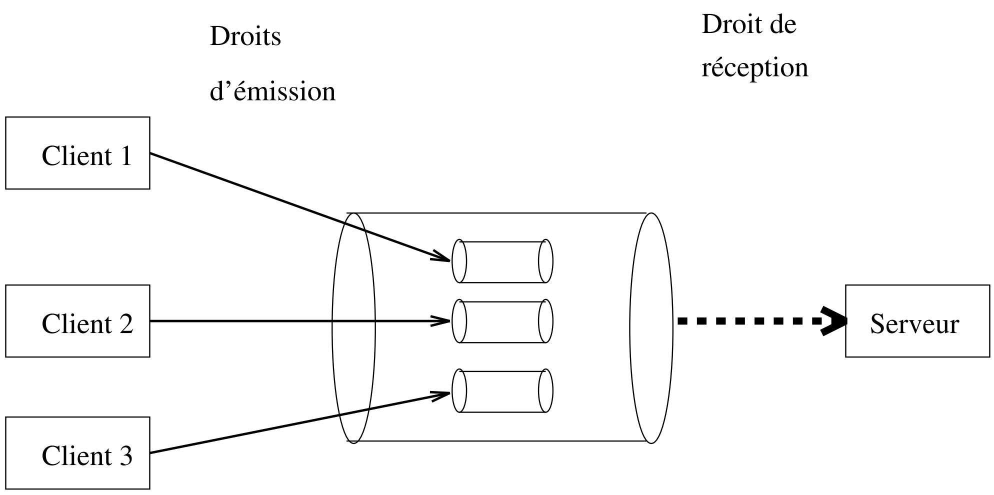

Figure 4.2: Ensemble de ports

Une tâche est une entité passive qui :

- regroupe un ensemble de ressources système. Ces ressources, à l'exception de l'espace d'adressage, sont référencées par des ports et peuvent être partagées avec d'autres tâches si les droits sur les ports leur sont transmis ;
- fournit un espace d'adressage important, éventuellement clairsemé, référencé par des adresses mémoire. Des parties de cet espace d'adressage peuvent être partagées avec d'autres tâches par héritage ou par des optimisations de la gestion mémoire, notamment lors des transmissions de messages ;
- contient un certain nombre de *threads*.

Une tâche a accès à plusieurs ports [Loepere 1992b] :

- *task kernel port*, représentant son nom, toute opération sur une tâche est effectuée en spécifiant ce port ;
- *task exception port*, sur lequel Mach envoie des messages lorsqu'un *thread* de la tâche n'ayant pas de port d'exception provoque une exception (accès mémoire illégal, division par zéro...) [Black *et al.* 1988] ;
- *task bootstrap port*, permettant à la tâche d'accéder à des serveurs ;
- *task registered ports*, qui sont des ports hérités de la tâche créatrice. Ces ports sont usuellement utilisés pour communiquer avec des serveurs globaux (serveurs de noms, serveur d'environnement, serveur de "service").

Un *thread* est une entité active qui :

- représente un point d'exécution dans une tâche ;
- a accès à toutes les ressources de la tâche qui le contient ;
- s'exécute en parallèle avec d'autres *threads* ;
- possède un état minimal.

Un *thread* est caractérisé par la tâche qui le contient, la pile qu'il utilise et la valeur des registres du processeur.

Un *thread* a accès à plusieurs ports [Loepere 1992b] :

- *thread kernel port*, représentant son nom, toute opération sur un *thread* est effectuée en spécifiant ce port ;
- *thread reply port*, permettant de recevoir les messages d'initialisation envoyés par Mach au *thread* ;
- *thread exception port*, sur lequel Mach envoie des messages lorsque le *thread* provoque une exception (accès mémoire illégal, division par zéro...).

2.7 Objets mémoire

Mach implémente une gestion performante de la mémoire physique et virtuelle [Tevanian 1987].

L'espace d'adressage d'une tâche définit l'ensemble des adresses virtuelles que les *threads* de la tâche sont autorisés à référencer. Cet espace d'adressage consiste en un ensemble de régions mémoire : Mach groupe en une région mémoire les pages contiguës possédant les mêmes attributs. Certaines opérations et Mach lui-même voient la mémoire d'une tâche comme une suite de régions mais ces régions sont invisibles à la tâche qui ne voit qu'une suite d'octets.

Mach utilise les régions mémoire afin d'optimiser sa gestion de la mémoire physique. Il implémente des stratégies de gestion mémoire respectant un principe d'évaluation tardive (*lazy evaluation*) :

- Lors de l'allocation de mémoire, les pages nécessaires ne sont réellement allouées dans la table des pages de la tâche appelante que lorsque cette dernière les référence. L'accès à une page non allouée provoque ainsi un défaut de page traité par Mach qui alloue la page.
- Lorsque des régions mémoire sont copiées entre deux tâches, une copie virtuelle est réalisée : les pages ne sont pas dupliquées en mémoire et les tables de pages des deux tâches pointent sur les mêmes cases physiques. Les pages sont toutes marquées en lecture seule et ne sont dupliquées que lors d'une écriture : lorsqu'une tâche écrit dans une page, une trappe "Violation de protection" est déclenchée, Mach duplique la page et modifie la table des pages des deux tâches en marquant la page comme accessible en lecture-écriture.

Mach définit la notion d'objet mémoire afin de référencer les pages chargées en mémoire. Un objet mémoire représente l'état non résident des pages associées à un objet. A chaque objet mémoire correspond un objet mémoire cache qui représente les pages chargées en mémoire.

Lorsqu'une page non présente en mémoire doit être chargée, Mach la lit depuis l'objet mémoire. De manière similaire, lorsqu'une page est évincée de la mémoire et doit être sauvegardée, elle est écrite dans l'objet mémoire.

2.8 Gestionnaires de mémoire

Mach permet à des tâches s'exécutant en mode utilisateur de gérer des objets mémoire. Ces tâches, appelées gestionnaires de mémoire ou paginateurs, sont chargées de lire ou d'écrire des pages depuis des objets mémoire.

En fait, Mach implémente lui-même la gestion de l'espace d'adressage des tâches en gérant les tables de pages, en choisissant les pages à charger ou à évincer et en traitant les défauts de page [Draves 1991] mais il se contente de transmettre les requêtes de chargement ou d'écriture de pages au gestionnaire de mémoire concerné.

A titre d'exemple, considérons le cas d'un *thread* accédant à une page non chargée en mémoire :

- l'accès à la page provoque un défaut de page détecté par le processeur qui provoque une trappe ;
- une routine interne de Mach est appelée pour traiter la trappe ;
- Mach contrôle que l'accès est valide (i.e. que la page est bien comprise dans l'espace d'adressage de la tâche contenant le *thread* ayant causé le défaut de page) ;
- lorsqu'il a vérifié la validité de l'accès mémoire, Mach transmet une requête de chargement sur le port de l'objet mémoire correspondant à la page puis suspend le *thread* ;
- le gestionnaire de mémoire reçoit la requête de chargement du micro-noyau, il charge la page en mémoire centrale puis envoie un message à Mach ;
- lorsque le gestionnaire de mémoire a répondu à sa requête, Mach modifie la table des pages de la tâche afin d'y inclure la page chargée puis réveille le *thread.*

De manière symétrique, Mach effectue des requêtes au gestionnaire de mémoire lorsque des pages modifiées sont évincées de la mémoire et doivent être sauvegardées.

Cette séparation de la gestion de la mémoire physique et des mécanismes de chargement et de sauvegarde de pages permet d'implémenter relativement aisément de nouvelles stratégies de gestion mémoire assurées par plusieurs gestionnaires de mémoire.

2.9 Processeurs

Mach offre un support des ordinateurs multi-processeurs [Black 1990]. Il permet de spécifier les processeurs à utiliser pour l'exécution de *threads*. Chaque processeur physique est référencé par un port. Ce port permet d'agir physiquement sur ce processeur (l'arréter, le démarrer, obtenir des informations).

Mach offre également la possibilité de grouper plusieurs processeurs en un ensemble de processeurs et d'affecter certains *threads* et tâches à cet ensemble. De la sorte, il est possible de contrôler l'allocation de processeurs lors de l'ordonnancement de *threads*.

2.10 Hôtes

Mach offre un accès à la machine physique sur laquelle il s'exécute en exportant deux ports représentant le site (*host*). Le premier port, accessible à toute tâche, permet d'obtenir des informations sur la machine (heure courante, version de Mach, caractéristiques physiques, ...). Le deuxième port, appelé port de contrôle, accessible seulement à la première tâche lancée lors du démarrage du système (*bootstrap task*) permet d'accomplir des opérations privilégiées (modification de l'heure, du gestionnaire de mémoire par défaut, accès aux ports de contrôle des processeurs, redémarrage du système...).

2.11 Gestion des périphériques

Le micro-noyau Mach contient les gestionnaires de périphériques physiques. Ces gestionnaires interagissent avec les périphériques (gestion des interruptions, envoi de requêtes aux contrôleurs, lecture des résultats d'entrées-sorties) [Forin *et al.* 1991] mais n'implémentent pas d'opérations de haut niveau telles que la lecture de caractères en mode canonique sous Unix.

Mach offre la possibilité à des tâches de dialoguer avec ces gestionnaires de péripériques. Pour cela, il exporte un port, appelé port maître des périphérique. Ce port est accessible seulement à la tâche lancée lors du démarrage du système (*bootstrap task*). A partir de ce port, une tâche peut demander à accéder à un périphérique. Elle obtient alors un port lui permettant de communiquer avec le gestionnaire de périphérique inclus dans Mach selon un dialogue normalisé (lecture et écriture de données, obtention et modification de l'état du périphérique).

3 Outils de conception de systèmes au-dessus de Mach

3.1 Gestionnaire d'entités

On peut considérer Mach comme un ensemble d'entités (ou objets), chaque entité étant gérée par un serveur. En général, un serveur gère plusieurs occurrences d'une entité. C'est en fait une tâche dont le rôle est de traiter les requêtes concernant les entités dont elle a la responsabilité.

Chaque occurrence d'entité est caractérisée par un port qui joue un double rôle :

- il identifie l'entité dans le système :

  Dans ce contexte, on ne nomme pas une entité par une adresse ou un identificateur unique mais par le port qui lui est associé ;

- il constitue un canal de communication pour les *threads* souhaitant agir sur cette entité :

  le gestionnaire de l'entité possède le droit de réception sur le port et traite les requêtes transmises sur ce port.

La figure 4.3 représente cette relation.

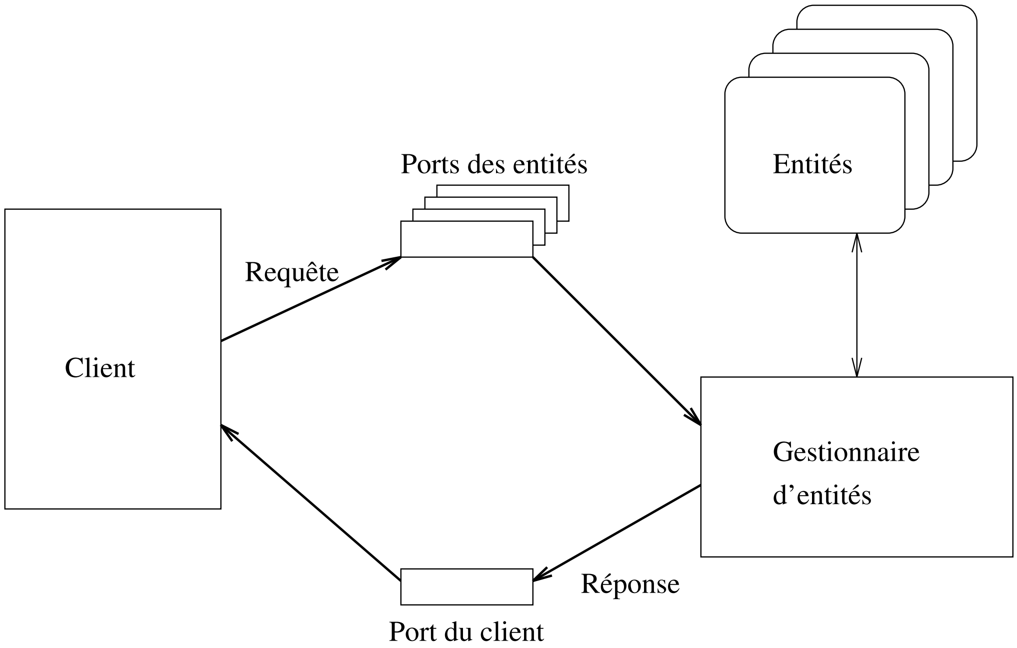

Figure 4.3: Gestionnaire d'entités dans Mach

3.2 Serveurs utilitaires

3.2.1 Serveur de noms

Mach a été conçu afin de permettre des implémentations de système d'exploitation en mode utilisateur sous forme de serveurs coopérants. Un tel schéma implique des communications via des ports accessibles en réception par les serveurs et en émission par les clients.

Un port géré par un serveur est généralement créé par ce serveur, soit lors de son initialisation, soit lors de l'allocation d'un objet géré par le serveur. Ce dernier possède donc automatiquement le droit de réception sur ce port.

Pour le client, par contre, il est nécessaire de disposer d'un moyen d'acquérir le droit d'émission sur ce port pour envoyer une requête au serveur.

Pour cela, Mach offre la possibilité de faire appel à un serveur de noms. Ce serveur est une tâche qui maintient une table associant des droits d'émission sur des ports et des noms de serveurs. Lorsqu'un serveur crée un port sur lequel il a le droit de réception, il le transmet (en le transformant en droit d'émission) au serveur de noms. Un client a ensuite la possibilité de demander au serveur de noms de lui transmettre ce droit d'émission.

Un client doit nécessairement avoir accès au port du serveur de noms pour l'utiliser. Afin de rendre cet accès possible, Mach place un droit d'émission sur ce port dans les ports spéciaux de toute tâche créée.

3.2.2 Serveur de communications en réseau

Mach permet uniquement des communications locales à une machine. Il ne fournit aucun moyen de communiquer avec des tâches s'exécutant sur des sites distants[1].

Le seul moyen de communiquer entre machines consiste donc à utiliser un serveur de communication qui s'exécute sur chaque site. Ce serveur possède un droit de réception sur les ports correspondant aux tâches distantes. Lorsqu'une tâche envoie un message sur l'un de ces ports, le message est reçu par le serveur de messages en réseau. Ce dernier le transmet, via le réseau, au gestionnaire sur la machine où s'exécute la tâche cible. Ce serveur reçoit le message et le transmet localement à la tâche cible.

La figure 4.4 représente la communication entre une tâche émettrice E et une tâche réceptrice R situées sur deux stations différentes, via les gestionnaires de communications par réseau.

L'utilisation des serveurs de communication en réseau, effectuant la transmission réelle entre machines, permet une communication transparente entre tâches situées sur des sites différents.

3.3 Mach Interface Generator (MIG)

3.3.1 Principe

Mach fournit des fonctionnalités puissantes de communication entre tâches mais il est relativement difficile d'utiliser au mieux les primitives fournies. En effet, la programmation d'un serveur

[1] Les dernières versions de Mach incluent des extensions "NORMA" expérimentales permettant de gérer des tâches distantes.

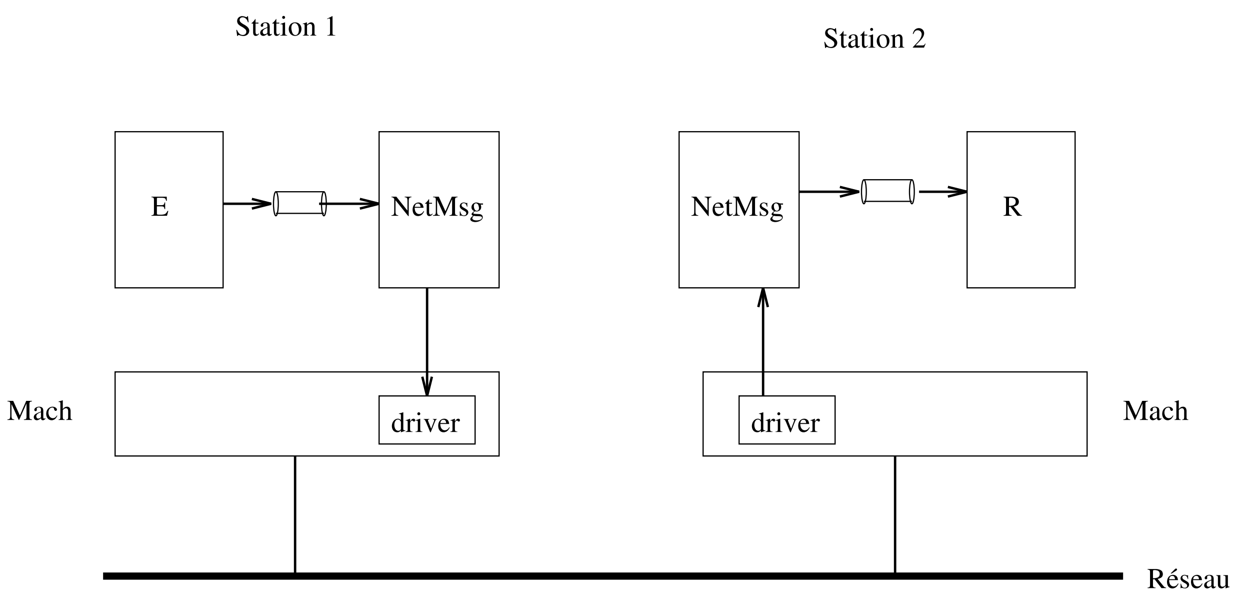

Figure 4.4: Communication entre tâches par réseau

consiste à écrire des fonctions d'attente de message de requête, de déballage des paramètres contenus dans ce message, d'aiguillage vers la fonction de service correspondant à l'opération demandée, d'emballage des résultats dans un message de réponse et d'envoi de ce message. De la même façon, l'envoi de requête à un serveur consiste à emballer les paramètres dans un message, à l'envoyer, à attendre la réception du message de réponse et à déballer les résultats contenus dans ce message. La programmation de telles fonctions est bien évidemment très fastidieuse.

Afin de faciliter l'implémentation de systèmes respectant une architecture client/serveur, le langage Matchmaker [Jones et Rashid 1986] permet au concepteur de serveurs de les spécifier et un programme, le "*Mach Interface Generator*" (MIG) [Drave *et al.* 1989] peut être utilisé pour générer automatiquement des fonctions de communication de type "Appel de procédures à distance".

MIG génère trois programmes source à partir d'une spécification d'opérations effectuées par un serveur :

- le "*User Interface Module*" contenant les fonctions réalisant :
  - l'emballage des paramètres dans des messages ;
  - la création d'un port de réponse ;
  - l'envoi de la requête ;
  - la réception du message de réponse depuis le serveur ;
  - le déballage des résultats ;
- le "*User Header Module*" contenant les définitions de types nécessaires et les prototypes de fonctions générées ;

- le "*Server Interface Module*" contenant :
  - la réception de messages de requêtes ;
  - le déballage des paramètres ;
  - l'aiguillage vers la fonction de service demandée ;
  - l'emballage des résultats dans un message de réponse ;
  - l'envoi du message de réponse.

3.3.2 Spécification MIG

Afin d'utiliser MIG, il est nécessaire de fournir une spécification des opérations à implémenter. Cette spécification comprend :

- une identification de sous-système :

  cette identification spécifie le nom du serveur à générer ainsi que le numéro de base des messages ; MIG utilise cette base pour les codes opération placés dans les messages envoyés au serveur ;
- une déclaration de fonction de démultiplexage,

  cette déclaration spécifie le nom de fonction de multiplexage générée par MIG ; par défaut, MIG génère une fonction appelée *sys*_server ;
- une déclaration de version ;
- des spécifications de types :

  ces spécifications permettent de déclarer, dans un langage proche du Pascal, des types correspondant aux types utilisés dans le langage cible ;
- des déclarations de fichiers à importer :

  ces déclarations sont transformées par MIG en directives d'inclusion de fichiers d'en-tête dans les programmes générés ;
- une description des opérations du serveur :

  chaque opération fournie par le serveur est déclarée en indiquant son nom, son type (routine ou routine simple) et ses paramètres (noms, modes de passage, types) ;
- des déclarations d'options :

  cette section permet de spécifier diverses options comme le mode de transmission des paramètres entre client et serveur et les options d'envoi de messages.

3.3.3 Exemple d'utilisation de MIG

À partir de la spécification fournie par le concepteur de serveur, MIG génère les fonctions d'envoi et de réception de message et de démultiplexage. Du côté client, le programmeur n'a donc qu'à appeler la fonction générée comme un sous-programme local. Cette fonction effectue, de manière transparente la transmission de la requête au serveur et l'attente du message contenant les résultats. Du côté du serveur, le programmeur doit uniquement fournir la fonction de traitement de chaque opération et le programme principal du serveur.

Supposons que nous souhaitions implémenter un serveur chargé de sauvegarder les priorités associées à des processus. Ce serveur peut recevoir deux types de requêtes : `setpriority (pid, priority)` et `getpriority (pid, priority)`. Le fichier de spécification MIG fourni par le programmeur est représenté sur la figure 4.5.

```
subsystem prior 245800;

userprefix prior_;
serverprefix do_;

#include <mach/std_types.defs>
#include <mach/mach_types.defs>

routine setpriority
(
      server      : mach_port_t;
      pid         : mach_port_t;
      priority    : int
);

routine getpriority
(
      server      : mach_port_t;
      pid         : mach_port_t;
      out priority: int
);
```

Figure 4.5: Exemple de spécification MIG

A partir de cette spécification, MIG génère deux fonctions utilisables du côté client pour effectuer l'emballage des paramètres et l'envoi de messages de requêtes `prior_getpriority` et `prior_setpriority` et une fonction utilisable dans le serveur `prior_server`. Le programmeur doit fournir le programme principal du serveur représenté sur la figure 4.6 ainsi que le corps des fonctions `do_getpriority` et `do_setpriority`.

```
main (void)
{
      mach_port_t serv;
      kern_return_t res;

      mach_port_allocate (mach_task_self (), MACH_PORT_RIGHT_RECEIVE, &serv);
      res = netname_check_in (ns_port, ``prior'', mach_task_self (), serv);
      if (res != KERN_SUCCESS)
            die (``Cannot register'');
      while (1)
            res = mach_msg_server (prior_server, MAX_MSG_SIZE, serv);
}
```

Figure 4.6: Exemple de programme principal de serveur

La figure 4.7 représente les fonctions générées par MIG et les interactions avec les fonctions du serveur et du client.

3.4 Librairie C-Threads

3.4.1 Principe

Mach fournit les tâches et *threads* qui permettent d'exécuter des programmes. Il ne fournit pas, par contre, de possibilité de synchronisation entre ces entités autres que la communication par messages.

Pour pallier ce manque, la librairie C-Threads [Cooper et Draves 1988] offre des fonctionnalités de plus haut niveau permettant d'implémenter plus facilement des *threads* exécutés en parallèle dans une application écrite en langage C.

Les fonctionnalités offertes par la libraririe C-Threads permettent la création de *thread*, le rendez-vous de *threads*, la protection de sections critiques avec des mutex et les synchronisations par le biais de variables conditions. Les fonctions de synchronisation permettent d'obtenir une sémantique proche de celle des moniteurs [Hoare 1974].

3.4.2 Gestion de *threads*

Les C-threads sont implémentés avec les *threads* Mach mais il n'existe pas forcément de correspondance exacte entre le nombre de *threads* et le nombre de C-threads. Au fur et à mesure

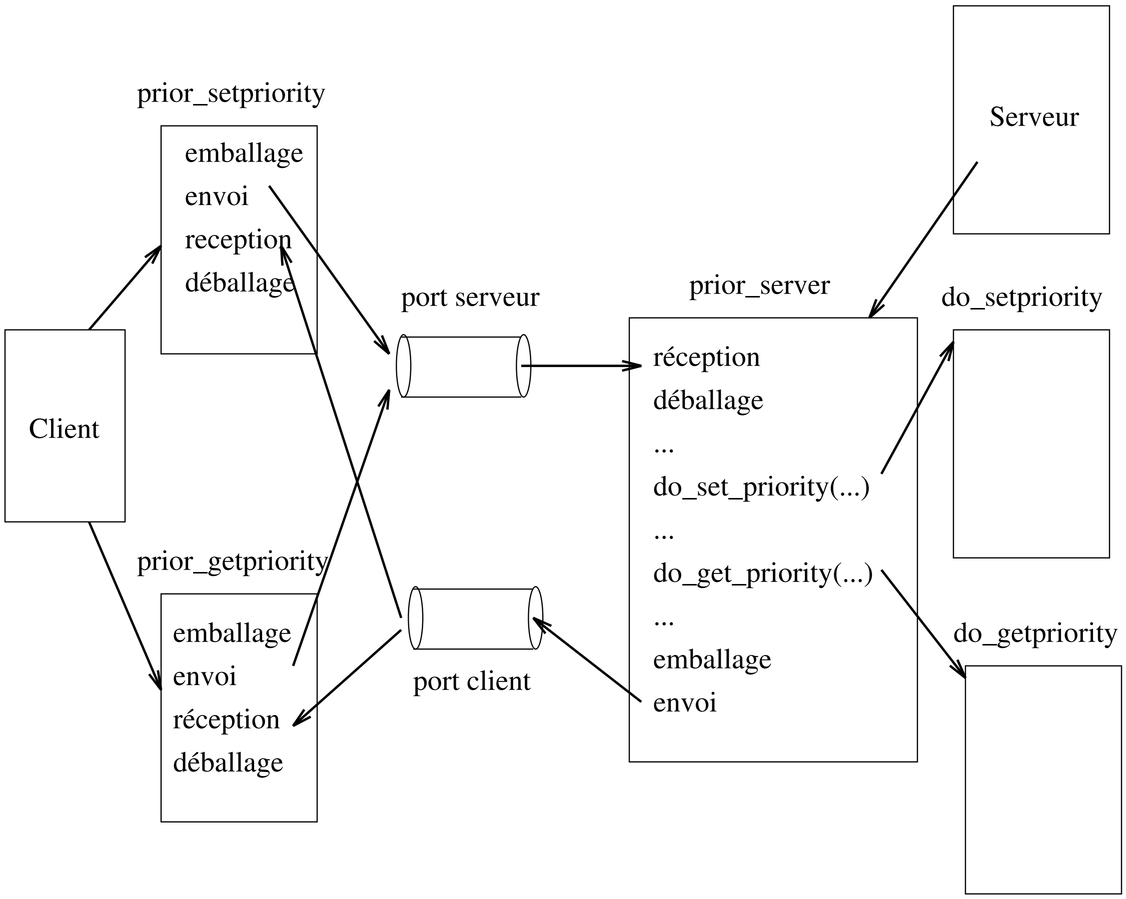

Figure 4.7: Utilisation de MIG

que les C-threads se suspendent et sont réveillés, les *threads* Mach qui les exécutent peuvent changer. Quand un C-Thread se termine, le *thread* Mach continue à exister et peut exécuter un autre C-thread. Ceci constitue un aspect important des C-threads : la librairie effectue les synchronisations et les changements de contexte entre C-threads en mode utilisateur et minimise ainsi le nombre d'appels système à Mach.

La librairie C-Threads utilise trois niveaux :

1. les *threads* fournies par le noyau Mach, que l'on peut voir comme des processeurs virtuels ;
2. les C-procs, que l'on peut voir comme les piles utilisées par les C-threads ;
3. les C-threads, qui représentent le traitement à effectuer (un sous-programme à exécuter et les arguments).

Un C-thread a besoin d'un C-proc et d'un *thread* pour s'exécuter. Si un C-proc est disponible, il est affecté au C-thread sinon ce dernier est placé dans une file d'attente. Quand le C-thread possède un C-proc, il le conserve jusqu'à sa terminaison. Pour s'exécuter, il a encore besoin d'un *thread* Mach. Si un *thread* est disponible, il lui est affecté sinon le C-thread est placé dans une file d'attente.

Quand un C-thread se suspend en raison d'une synchronisation, le *thread* qui l'exécutait est mis à la disposition des autres C-threads. Bien sûr, si un C-thread effectue un appel système bloquant à Mach, le *thread* qui l'exécute est lui aussi suspendu. Afin de suspendre uniquement le C-thread, la librairie fournit des fonctions d'encapsulation de certains appels système (notamment les envois et réceptions de messages) qui ne suspendent pas le *thread* Mach.

Il est à noter que l'utilisation de C-threads nécessite de disposer d'une librairie offrant des sous-programmes ré-entrants [Jones 1991].

3.4.3 Synchronisation

La librairie C-Threads fournit deux outils de synchronisation : les mutex et les conditions.

Un mutex est tout simplement un sémaphore binaire. Un C-thread peut demander le verrouillage d'un mutex (en demandant sa suspension si le mutex est déjà verrouillé), déverrouiller un mutex qu'il avait auparavant verrouillé et essayer de verrouiller un mutex (en recevant un code d'erreur si le mutex est dejà verrouillé).

Une condition est associée à un mutex. Un C-thread peut attendre qu'une condition devienne vraie, signaler qu'une condition est réalisée (en réveillant un C-thread en attente sur la condition) et diffuser une condition (en réveillant tous les C-threads en attente sur la condition).

Un exemple d'utilisation des mutex et conditions est représenté sur la figure 4.8.

```
mutex_lock (m);
while (condition_fausse)
      condition_wait (c, m);
/* Traitement */
mutex_unlock (m);
```

Figure 4.8: Utilisation de mutex et de conditions

4 Systèmes conçus au-dessus de Mach

Dans cette partie, nous présentons les caractéristiques de plusieurs systèmes d'exploitation conçus ou en cours de conception ou de portage au dessus de Mach.

Trois types de systèmes sont présentés ici :

1. un système à architecture monolithique (OSF/1 IK) utilisant Mach comme couches basses du noyau ;
2. deux systèmes composés d'un seul serveur (OSF/1 MK, BSD SS) qui s'exécute au dessus de Mach ;
3. trois systèmes formés d'une coopération de plusieurs serveurs (Unix Multi Server, GNU Hurd, Guide).

4.1 OSF/1 IK

OSF/1 IK [OSF 1990], développé par l'"Open Software Fundation", repose sur Mach 2.5. Ce système reprend la structure monolithique présente dans Unix et se contente d'utiliser les fonctionnalités de Mach pour assurer les sémantiques Unix sans en tirer pleinement profit. Pour ces raisons, nous nous contentons de le décrire brièvement.

Sa structure est représentée sur la figure 4.9.

Les innovations offertes par OSF/1 IK par rapport à Unix, au niveau structurel, sont :

- l'utilisation des fonctionnalités de gestion de tâches et *threads* pour fournir des processus selon la sémantique Unix ;
- la possibilité d'utiliser plusieurs *threads* au sein d'un même processus ;

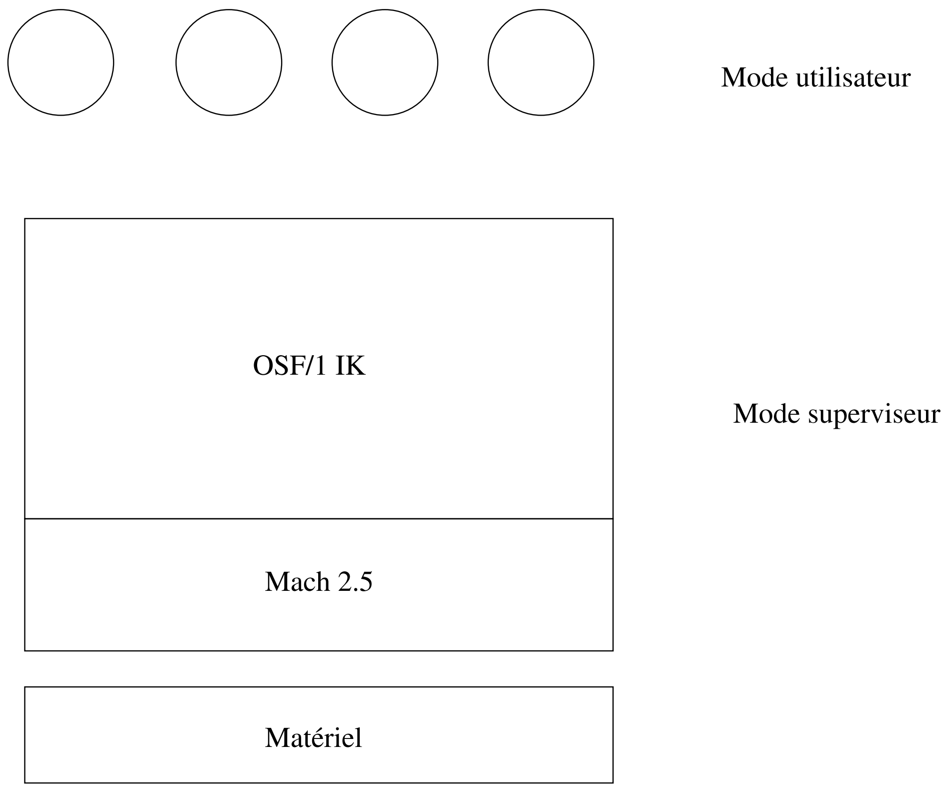

Figure 4.9: Structure de OSF/1 IK

- la parallélisation des fonctions du système, grâce aux fonctionnalités fournies par Mach, afin de permettre une exécution optimale sur des ordinateurs multiprocesseurs ;
- l'utilisation des gestionnaires de périphériques contenus dans Mach pour effectuer les entrées-sorties ;
- l'optimisation de la gestion de la mémoire virtuelle grâce à l'utilisation des mécanismes de Mach (notamment par rapport à la version BSD de Unix qui implémente une gestion peu performante de la mémoire[2] [Leffler *et al.* 1989]).

L'utilisation de fonctionnalités de Mach permet d'alléger le code du système lui-même puisque de nombreuses fonctions internes (ordonnancement de processus, gestion des périphériques, gestion de la mémoire physique...) peuvent être remplacées par des appels à Mach. Néanmoins, de par son caractère monolithique, OSF/1 IK n'est pas différent d'une implémentation classique de fonctionnalités Unix dans la mesure où il est fondé sur la même structure et en possède donc les mêmes inconvénients.

4.2 OSF/1 MK

4.2.1 Structure de OSF/1 MK

OSF/1 MK [Places *et al.* 1992] est actuellement un prototype dans lequel le code de OSF/1 IK s'exécute en mode utilisateur, dans un serveur composé de plusieurs *threads*, au dessus de Mach.

Sa structure est représentée sur la figure 4.10.

Dans OSF/1 MK, les fonctionnalités Unix sont implémentées par un serveur. Ce serveur contient le même code que OSF/1 IK et deux couches chargées d'assurer le bon fonctionnement en mode utilisateur de ce code, conçu pour s'exécuter en mode superviseur :

- l'interface des appels système convertit les appels système effectués par les processus utilisateur sous forme d'appels de procédures à distance (voir 4.2.2), sous la forme attendue par le code OSF/1 (trappes) ;
- les services de base permettent au serveur d'utiliser les mêmes mécanismes qu'en mode superviseur : le code d'OSF/1 IK utilise de nombreuses fonctionnalités qui sont soit offertes par Mach soit impossibles à utiliser en mode utilisateur (le masquage d'interruptions par exemple). Cette couche fournit de nouvelles implémentations de ces fonctionnalités, typiquement en redirigeant les requêtes vers Mach.

[2] pour les versions antérieures à 4.4BSD, cette dernière ayant repris la gestion de la mémoire virtuelle de Mach

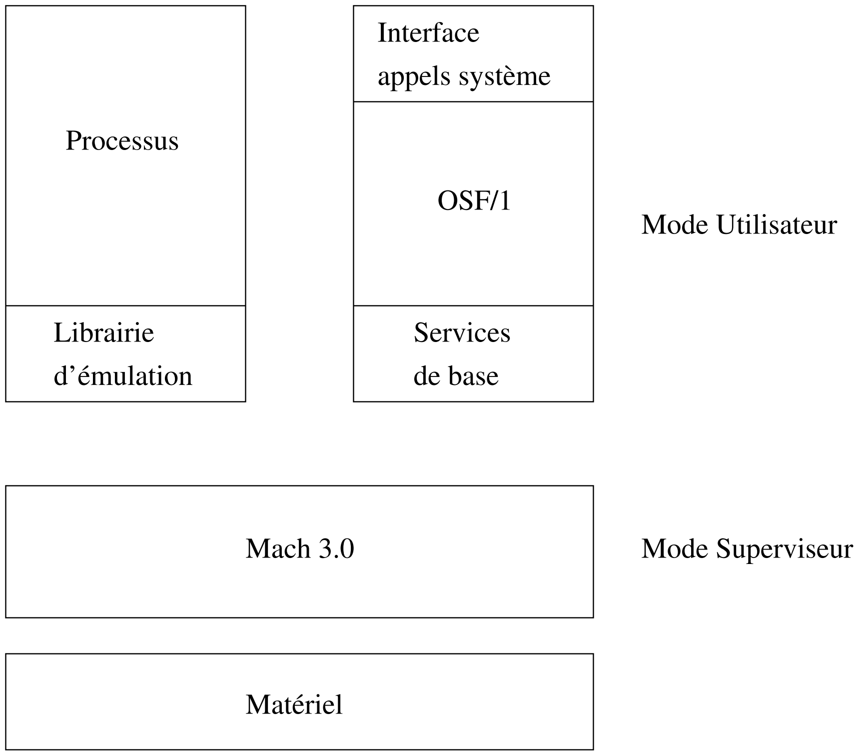

Figure 4.10: Structure de OSF/1 MK

4.2.2 Traitement des appels système dans OSF/1 MK

OSF/1 MK implémente une compatibilité binaire avec Unix en utilisant des possibilités d'émulation fournies par Mach. Lorsqu'un processus effectue un appel système, il exécute une trappe qui passe le contrôle à Mach en mode superviseur. Mach redirige l'appel, d'après un vecteur d'émulation vers la librairie d'émulation qui est chargée d'exécuter l'appel système.

La librairie d'émulation peut traiter les appels système de deux façons différentes :

- dans le cas d'un appel système simple modifiant ou consultant une donnée système associée au processus (figure 4.11), elle exécute elle-même l'appel en accédant directement à la donnée qui est incluse dans son espace d'adressage et dans celui du serveur. Pour des raisons évidentes de protection, cette méthode ne peut pas être utilisée pour les données sensibles ;
- dans le cas d'un appel système plus complexe (figure 4.12), la librairie d'émulation effectue un appel de type "appel de procédure à distance" au serveur qui traite l'appel et retourne les résultats dans un message.

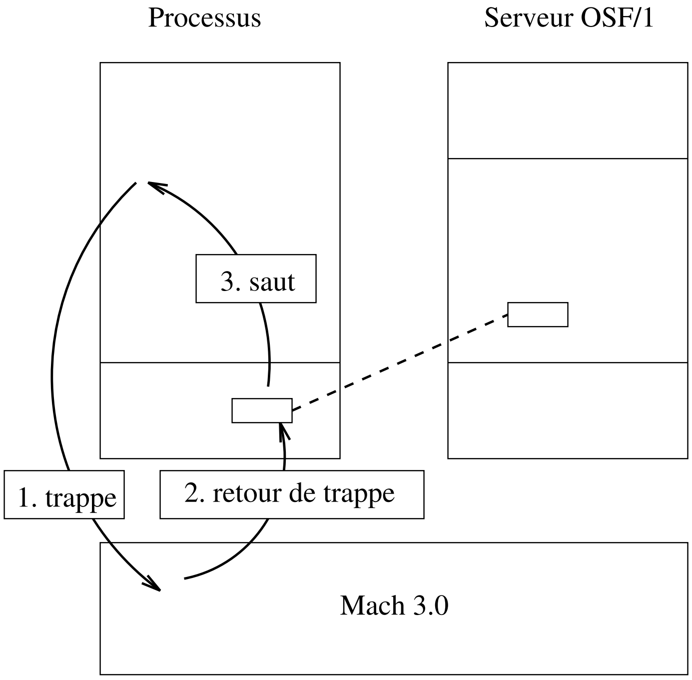

Figure 4.11: Appel système simple dans OSF/1 MK

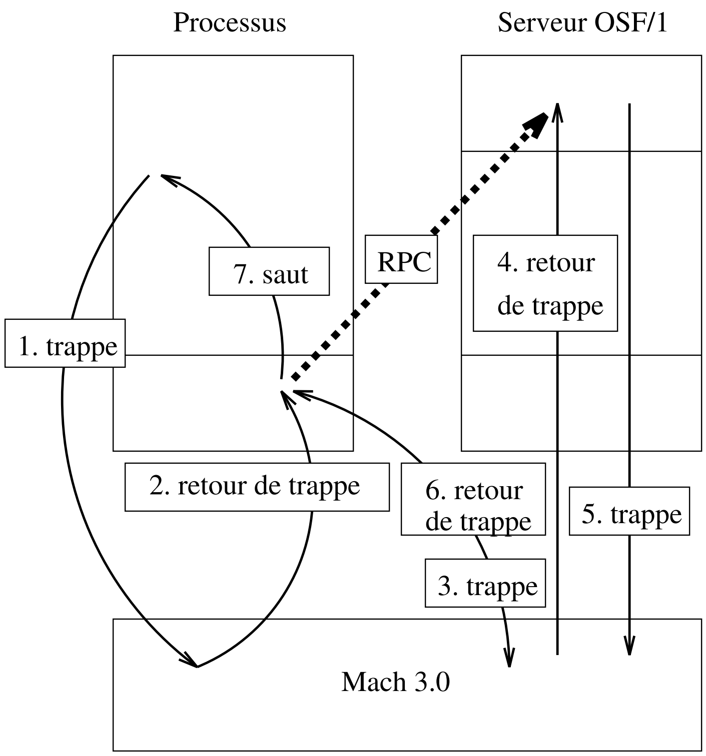

Figure 4.12: Appel système complexe dans OSF/1 MK

4.2.3 Gestion des processus

Les processus sont gérés par le biais des tâches et *threads* Mach. Le serveur OSF/1 maintient des structures de données contenant les informations liées aux sémantiques Unix des processus (identificateur d'utilisateur, identificateurs de groupes d'utilisateurs, fichiers ouverts...) qui ne sont pas gérées directement par Mach.

Les informations maintenues par le serveur sont également accessibles par la librairie d'émulation dans son espace d'adressage afin de lui permettre de traiter directement certains appels système sans faire de requête au serveur.

La mémoire virtuelle des processus est gérée par un paginateur, le "vnode pager" qui fait partie intégrante du serveur.

4.2.4 Gestion des entrées-sorties

Le serveur OSF/1 contient une version non modifiée des fonctions d'entrées-sorties d'Unix BSD (systèmes de fichiers, gestion des sockets, ...). Dans Unix, ces sous-systèmes utilisent des tables de pointeurs sur fonctions pour accéder aux gestionnaires de périphériques [Leffler *et al.* 1989]. Dans OSF/1 MK, ces tables contiennent les adresses de fonctions, contenues dans la couche de services de base, qui redirigent les requêtes vers les gestionnaires de périphériques contenus dans Mach.

4.3 BSD Single Server

Le serveur BSD développé à l'Université de Carnegie-Mellon contient un portage des divers sous-systèmes contenus dans le système 386BSD [Jolitz et Jolitz 1992].

Dans la mesure où ce système est conçu de manière similaire à OSF/1 MK et implémente les mêmes stratégies, nous nous contentons de le citer sans détailler son fonctionnement.

4.4 Unix Multi Server

4.4.1 Structure du système

Dans le prototype de système multi-serveurs développé à l'université de Carnegie-Mellon [Guedes et Julin 1991a], les fonctionnalités du système sont assurées par trois entités :

1. le micro-noyau Mach 3.0, fournissant la gestion de la mémoire, la gestion de tâches, les communications entre tâches et la gestion des périphériques ;

2. un ensemble de serveurs s'exécutant en mode utilisateur, fournissant les fonctionnalités de haut-niveau : fichiers, processus, communications par réseau...) ;
3. une librairie d'émulation, accessible dans l'espace d'adressage de chaque processus, implémentant la communication avec les serveurs.

4.4.2 Traitement des appels système

Les appels système sont implémentés de la même façon que dans OSF/1 MK : les processus utilisateur provoquent une trappe gérée par Mach qui la redirige vers la librairie d'émulation selon un effet de trampoline [Julin *et al.* 1991]. Cette librairie exécute ensuite l'appel système en transmettant une ou plusieurs requêtes aux serveurs.

Le système utilise les ports et les primitives de communication fournis par Mach : toute abstraction Unix (fichier, socket, tube...) est représentée par un port géré par un serveur. Les possibilités de partage de mémoire sont également utilisées pour les communications entre clients et serveurs. Par exemple, le contenu des fichiers est rendu accessible aux clients dans leur espace d'adressage afin de permettre des entrées-sorties rapides.

4.4.3 Structure des serveurs

Les fonctionnalités du système sont séparées en deux couches :

1. la couche de service contient un ensemble de serveurs fournissant des services de base définis de façon aussi générique que possible. Ces services peuvent être utilisés afin d'émuler de systèmes d'exploitation différents ;
2. la couche spécialisée implémente les fonctionnalités spécifiques à un système d'exploitation. Elle convertit les appels respectant les sémantiques du système émulé en appels aux services génériques fournis par la couche de service.

Cette séparation repose sur le fait que les accès aux serveurs sont effectués uniquement par la librairie d'émulation. En faisant convertir les appels spécifiques en appels de services génériques par cette librairie, il est possible d'implémenter des serveurs pouvant être utilisés pour l'émulation de plusieurs systèmes différents.

La communication entre la librairie d'émulation respecte une interface qui n'est pas visible aux utilisateurs.

4.4.4 Orientation objets

Afin d'augmenter la généricité de la couche de service, une communication orientée objets est implémentée. Tout service du système est représenté par un objet abstrait qui exporte un ensemble bien défini d'opérations. Chaque serveur gère plusieurs objets de même type et implémente les opérations sur ces objets.

La librairie d'émulation utilise cette généricité grâce à l'emploi de classes d'objets [Guedes et Julin 1991b].

4.4.5 Optimisations

Les concepteurs de ce système souhaitent le rendre aussi évolutif et dynamique que possible. Pour cela, ils préconisent l'emploi de nombreux serveurs indépendants pouvant être assemblés dynamiquement.

Cette approche permet d'obtenir une grande flexibilité du système, une programmation et une mise au point plus facile de chacun des serveurs et une plus grande sécurité du système en isolant les serveurs défaillants. Cette structuration doit néanmoins être tempérée par certaines considérations pratiques [Julin *et al.* 1991] :

- les interactions entre plusieurs serveurs sont plus couteuses que les communications au sein d'un même serveur. Il convient donc de minimiser au maximum ces interactions ;
- l'état lié aux processus doit être distribué entre les différents serveurs. Ceci introduit des interactions supplémentaires entre eux ;
- les authentifications doivent être effectuées par chaque serveur ;
- bien que les serveurs offrent des fonctionnalités de haut niveau différentes, il est fréquent qu'ils doivent effectuer de nombreuses opérations de bas niveau similaires. Ces fonctions doivent donc être dupliquées dans tous les serveurs.

Afin de limiter les communications entre clients et serveurs, la librairie d'émulation, en plus de la conversion entre les appels spécifiques et les services génériques, implémente elle-même certaines fonctionnalités :

- elle gère directement certaines informations liées aux processus (table des descripteurs de fichiers, masque et gestionnaires des signaux, répertoire courant...) et effectue elle-même certaines fonctions complexes (résolution de chemins d'accès, exécution de programme...) ;
- lorsqu'un serveur fournit un accès à un objet, un fragment de code spécial est inséré dans l'espace d'adressage du client afin d'être utilisé comme représentant local de l'objet. Ce code, fourni par le serveur, implémente une indirection entre le client et le serveur. Il peut se contenter, dans le cas le plus simple, de transmettre une requête au serveur mais peut également effectuer des optimisations permettant de ne pas communiquer avec le serveur ;
- elle peut également opérer comme une entité active pour simplifier le fonctionnement de certains serveurs. Par exemple, un *thread* contenu dans la librairie d'émulation gère les communications asynchrones et les convertit en signaux Unix.

4.4.6 Serveurs

Le système contient deux types de serveurs :

- les serveurs fournissant des services de haut niveau accessibles aux processus utilisateur via la librairie d'émulation :
  - plusieurs serveurs de fichiers gérant différents types de systèmes de fichiers physiques ;
  - un serveur de terminaux gérant les lignes série et implémentant les pseudo terminaux d'Unix ;
  - un serveur de communication local implémentant les communications inter-processus (tubes, sockets...) ;
  - un serveur de processus gérant les groupes de processus et les signaux ;
  - un serveur de communication en réseau implémentant les sockets ;
  - un serveur de périphériques gérant l'accès aux fichiers spéciaux représentant les périphériques ;
- les serveurs fournissant des services aux autres serveurs :
  - un serveur d'authentification ;
  - un serveur de configuration chargé de lancer les autres et de maintenir et de fournir un accès aux informations globales ;
  - un serveur de noms regroupant tous les autres dans un espace d'adressage hiérarchique grâce à un ensemble de points de montage ;
  - un serveur gérant l'état global contenu dans des pages partagées entre serveurs et clients ;
  - divers serveurs secondaires, indépendants du système émulé, fournissant des fonctions de diagnostic, des mécanismes de mémoire virtuelle partagée, des communications par réseau...

4.5 GNU Hurd

4.5.1 Objectifs

Hurd est un système d'exploitation en cours de conception par la "Free Software Foundation". Ses concepteurs souhaitent développer un système qui soit minimal : les utilisateurs doivent seulement être capables de communiquer avec le noyau, le reste du système étant remplaçable dynamiquement. Les utilisateurs peuvent donc utiliser les parties du système qu'ils veulent ainsi que des composants qu'ils ajoutent eux-mêmes.

4.5.2 Communications dans Hurd

Hurd utilise les ports Mach comme méthode de communication entre utilisateurs et serveurs. Chaque port implémente un ensemble précis de protocoles, représentant les opérations pouvant être effectuées sur l'objet représenté par le port. Certains des protocoles utilisés par Hurd sont :

- le protocole d'entrées-sorties, utilisé pour les opérations génériques d'entrées-sorties ;
- le protocole d'accès aux fichiers, utilisé pour les opérations d'entrées-sorties sur systèmes de fichiers ;
- le protocole d'accès aux sockets, utilisé pour les communications réseau ;
- le protocole de gestion de processus, utilisé pour manipuler des processus.

4.5.3 Mécanisme de translation

La plupart des serveurs sont accessibles via des fichiers. Normalement, l'ouverture d'un fichier provoque l'envoi d'un droit d'émission sur un port caractérisant le fichier. Tout système de fichiers est géré par un serveur ; quand un fichier est ouvert, le serveur crée un port, l'associe au fichier et transmet un droit d'émission sur ce port au client.

Un fichier peut avoir un "translateur" qui lui est associé. Dans ce cas, au lieu de retourner un port caractérisant le fichier, le serveur exécute un programme, appelé "translateur". Le serveur fournit à ce programme le port correspondant au fichier ouvert et le translateur doit renvoyer un port au client.

Ce mécanisme de translation est utilisé pour le montage de systèmes de fichiers : quand un utilisateur ouvre le fichier correspondant à un point de montage, le translateur correspondant, qui comprend la structure physique du système de fichiers monté, est exécuté et renvoie un port au client. Les accès ultérieurs aux fichiers sont donc ensuite dirigés, via ce port, sur ce translateur.

Un translateur peut être associé à un fichier par son propriétaire. De la sorte, tout programme peut être utilisé comme translateur à condition qu'il respecte un protocole bien défini. Il est donc ainsi possible, pour tout utilisateur, d'ajouter dynamiquement des fonctionnalités au système par le biais de translateurs.

Aucun des translateurs présents installé dans le système de fichiers ne voit ses privilèges accrus. Les translateurs s'exécutent avec l'identité et les droits des propriétaires des fichiers auxquels ils sont associés, et peuvent être associés à des fichiers ou modifiés seulement par leurs propriétaires. Les protocoles d'entrées-sorties et d'accès aux fichiers sont conçus de façon à ne pas faire confiance aux différents clients et serveurs.

4.5.4 Structure interne de Hurd

Hurd est conçu comme un ensemble de serveurs coopérants. Certains de ces serveurs sont essentiels au fonctionnement du système alors que d'autres peuvent être remplacés grâce au mécanisme de translation.

4.5.4.1 Serveur d'authentification Le serveur d'authentification constitue un point central du système. Chaque port géré par ce serveur identifie un utilisateur et est associé à un "bloc d'identification". Chaque bloc d'identification contient un ensemble d'identités utilisateur et un ensemble d'identités de groupes, chacun de ces ensembles pouvant être vide.

Le serveur d'authentification fournit trois services :

- il offre des opérations booléennes simples sur les ports d'identification. A partir de deux ports, le serveur renvoie un troisième port représentant l'union des ensembles contenus dans les deux ports.
- il autorise tout utilisateur ayant un identificateur nul[3] à créer un port d'authentification quelconque.
- il fournit des appels de procédures à distance qui autorisent des clients et des serveurs, ne se faisant pas confiance, à établir leur identité et à se transmettre des informations initiales.

Tout utilisateur peut écrire un programme qui implémente le protocole d'authentification. Néanmoins, ceci ne met pas en péril la sécurité du système : quand un serveur a besoin d'authentifier un utilisateur, il s'adresse à son propre serveur d'authentification à qui il fait confiance. Si l'utilisateur utilise un autre serveur d'authentification, la vérification échoue et le serveur peut refuser de dialoguer avec le client.

4.5.4.2 Serveur de processus Le serveur de processus est simplement chargé de tenir à jour des informations. Il offre quatre principaux services :

- il maintient les informations relatives à la machine, qui ne sont pas tenues à jour par Mach. Ainsi, le nom de la machine, son identification, la version du système, etc, sont conservés par ce serveur ;
- il tient à jour les sessions et groupes de processus selon les sémantiques Posix ;
- il maintient une correspondance entre identificateurs de processus et tâches Mach. Il offre également aux processus la possibilité d'enregistrer un port qui peut être transmis à tout processus en faisant la demande. Il permet enfin à tout processus d'enregistrer ses paramètres (arguments et environnement) afin de les transmettre à d'autres processus ;

[3] l'identificateur 0 est traditionnellement associée au super-utilisateur sous Unix

- il implémente des “collections de processus”, utilisées pour obtenir les ports de plusieurs processus en une seule opération.

La création de processus est implémentée dans une fonction de librairie : une nouvelle tâche est créée en utilisant les appels système de Mach puis s’enregistre auprès du serveur de processus.

L’exécution de programmes est implémentée par trois programmes :

- la librairie C gère la liste des arguments du programme puis transmet un message de requête au serveur de fichiers correspondant au fichier à exécuter ;
- le serveur de fichiers vérifie les droits d’exécution du fichier et effectue les modifications nécessaires – par exemple, si l’exécution du programme doit modifier l’identité de l’utilisateur et si le serveur de fichiers a les privilèges requis, il modifie cette identification. Le serveur de fichiers peut également, lorsqu’il augmente les privilèges du processus, créer une nouvelle tâche Mach pour continuer l’exécution du processus. Le serveur de fichiers transmet ensuite la requête au serveur d’exécution ;
- le serveur d’exécution charge le code à exécuter depuis le fichier en reconnaissant son format.

4.5.4.3 Serveurs de fichiers Dans Hurd, les systèmes de fichiers sont gérés par des translateurs. À chaque système de fichiers physique correspond un translateur qui comprend le format des fichiers. Toute requête concernant un fichier est transmise au translateur correspondant qui effectue l’entrée-sortie demandée en interprétant la structure physique des fichiers sur les disques.

4.6 Guide

Guide [Balter *et al.* 1991] est un système réparti basé sur les objets. Ce système a d’abord été réalisé sous forme de prototype sous Unix et est en cours de portage au dessus de Mach.

Nous présentons tout d’abord les motivations ayant conduit à ce système, nous exposons ensuite ses principes de base et nous comparons enfin ses implémentations sous Unix et au dessus de Mach.

4.6.1 Objectifs

Les objectifs dy système Guide sont les suivants :

- Orientation Objet ;
- Support d’un langage de haut niveau ;

- Sauvegarde d'objets ;
- Machine virtuelle répartie.

Guide peut être perçu comme une machine virtuelle uniprocesseur répartie dans laquelle la répartition est cachée à l'utilisateur mais où le parallélisme est visible. Les applications sont structurées sous forme d'objets qui sont stockés, de façon transparente, sur plusieurs sites.

L'unité d'exécution utilisée par les applications est le traitement qui fournit un espace d'adressage d'objets et plusieurs activités parallèles. Une application typique est implémentée sous forme d'un traitement unique dont l'exécution est lancé sur un site du système et qui peut accéder aux autre sites selon la localisation des objets qu'elle utilise. Une application complexe nécessitant plusieurs espaces d'adressage peut être représentée par plusieurs traitements coopérants. Au niveau des applications, la communication entre traitements ou activités repose entièrement sur les objets partagés.

4.6.2 Principes de base

Dans Guide, un objet encapsule des données (son état) et des opérations. Les données ne sont accessibles que par les opérations fournies par l'objet. Un type définit les caractéristiques d'un objet (variables publiques et profils des opérations). Une classe définit une implémentation d'un type (représentation des données, code des opérations).

Les objets sont persistants : leur existence n'est pas liée à l'exécution d'un programme. Un objet persistant existe tant qu'il est référencé par un autre objet persistant. Un objet persistant est référencé, de manière interne, par un nom unique indépendant de sa localisation. Les utilisateurs y accèdent via un langage [Krakowiak *et al.* 1990] implémenté au dessus de Guide.

La persistance des objets est assurée par un service de stockage qui gère tous les objets accessibles par des traitements. Un gestionnaire de mémoire virtuelle gère les objets utilisés par des traitements à un instant donné.

Les objets sont stockés de manière répartie en mémoire secondaire. Un objet est stocké sur un seul site mais peut contenir des références à des objets situés sur d'autres sites. L'utilisateur ne peut pas connaitre la localisation réelle des objets qu'il utilise afin de faciliter leur migration entre sites.

Un traitement est composé de plusieurs activités concurrentes qui accèdent à des objets passifs. L'exécution d'une activité consiste en une succession d'appels des méthodes des objets manipulés. A chaque appel, l'objet accédé est localisé, chargé et lié à l'espace de mémoire virtuelle du traitement. L'ensemble des objets liés à un traitement constitue son contexte.

4.6.3 Implémentation sous Unix et au dessus de Mach

Au dessus d'Unix, une activité est représentée par un processus. Les mécanismes de mémoire partagée sont utilisés pour partager les objets entre activités. Au dessus de Mach, par contre les notions de tâche et de *thread* correspondent exactement aux sémantiques de Guide. Chaque traitement est donc implémenté sous forme d'une tâche contenant plusieurs *threads*, chacun représentant une activité. L'accès au même espace d'adressage par tous les *threads* d'une tâche permet un partage des objets.

Dans l'implémentation Unix de Guide, la mémoire virtuelle de chaque site est représentée par une seule région mémoire contenant tous les objets associés au site. Au dessus de Mach, les fonctionnalités de gestion de la mémoire fournies par le micro-noyau sont utilisées :

- chaque objet est chargé dans une région mémoire d'une tâche ;
- les *threads* partagent automatiquement tous les objets chargés par un traitement ;
- le partage d'objets sur un même site est effectué en plaçant les régions mémoire correspondantes dans l'espace d'adressage des tâches qui y accèdent ;

L'accès à un objet distant, sous l'implémentation Unix de Guide, nécessite soit de migrer l'objet sur le site courant s'il n'est pas utilisé par un autre traitement, soit de créer une activité sur le site où est stocké l'objet, dans le cas où il est déjà utilisé, et à y accéder par des requêtes à cette activité. Au dessus de Mach, un paginateur externe est utilisé pour fournir une copie locale de chaque objet accédé. Ainsi, tout objet est accédé localement mais l'ensemble des paginateurs externes doivent implémenter un protocole de cohérence des objets [Boyer 1991].

4.6.4 Comparaison des implémentations Unix et Mach

L'analyse comparée des implémentations de Guide sous Unix et au dessus de Mach [Boyer *et al.* 1991] permet de dégager de nombreux avantages à l'utilisation du micro-noyau :

- en utilisant les tâches et *threads* pour représenter les traitements et activités, les mécanismes de gestion mémoire permettent une protection efficace des objets ainsi que leur partage entre les activités d'un même traitement ;
- en utilisant un ensemble de paginateurs externes pour les accès aux objets depuis plusieurs sites, il est possible d'implémenter un protocole d'accès efficace et cohérent alors que l'implémentation sous Unix oblige à transformer toute invocation en accès local en migrant l'objet ou en créant une activité sur le site contenant l'objet ;
- l'utilisation de paginateurs externes pour l'accès aux objets permet d'intégrer le stockage des objets en mémoire secondaire ;
- l'utilisation des ports Mach pour nommer les objets permet de se dispenser d'une couche de nommage ;

- les communications fournies par Mach sont fortement liées à la gestion de la mémoire et sont plus efficaces que les moyens de communications d'Unix (sockets, messages...) ajoutés comme une surcouche du système ;
- l'utilisation de Mach permet d'obtenir une forte tolérance aux pannes : si une panne se produit, seul le serveur défaillant est touché, un algorithme de duplication des objets implémenté par les paginateurs externes peut permettre au système de continuer à s'exécuter.

4.7 Synthèse

Les systèmes que nous avons présentés sont tous bâtis au dessus de Mach mais sont structurés de façons différentes.

OSF/1 IK constitue un système traditionnel basé sur un noyau monolithique et n'utilise Mach que comme remplacement des couches de bas niveau de Unix. La principale motivation lors de la conception de ce système semble avoir été de minimiser le temps de développement et d'obtenir dans des délais brefs un système complet. Nous pensons qu'une telle utilisation de Mach[4] va a l'encontre des principes et de la philosophie des micro-noyaux.

Les deux systèmes implémentés sous forme de serveur unique au dessus de Mach, OSF/1 MK et BSD SS, reportent les fonctionnalités du système dans une tâche utilisateur. Cette tâche implémente les sémantiques Unix en s'appuyant sur les fonctionnalités de Mach.

L'intérêt d'une telle approche réside dans ses délais de réalisation : dans la mesure où le code d'un système existant est repris et encapsulé pour s'exécuter au dessus de Mach, il est possible de réutiliser une énorme quantité de code déjà testé et d'obtenir ainsi assez rapidement un système opérationnel.

Néanmoins, même si l'utilisation de Mach permet de réduire sensiblement la taille du système, ce dernier n'en demeure pas moins conçu selon le modèle monolithique. Le serveur constitue un gigantesque programme[5] qu'il est difficile de maintenir et de faire évoluer.

Les trois derniers systèmes présentés, Unix Multi Server, Hurd et Guide, sont actuellement en cours de conception ou à l'état de prototype. Il est donc particulièrement délicat de juger les choix qui ont été effectués.

Nous pouvons néanmoins remarquer que ces trois systèmes s'appuient sur des ensembles de serveurs coopérants pour fonctionner. La distribution des fonctionnalités d'un système en serveurs offre de nombreux avantages :

- chaque serveur est plus petit donc plus facile à programmer, à mettre au point, à maintenir et à faire évoluer ;

[4] OSF/1 IK est basé sur Mach 2.5
[5] moins important qu'un “vrai” système monolithique mais néanmoins imposant

- l'assemblage de serveurs pour former un système est un gage de dynamicité si un protocole d'intégration de serveurs dans le système est défini ;
- la séparation en plusieurs serveurs disposant chacun de leur espace d'adressage permet d'offrir une meilleure résistance aux erreurs logicielles contenues dans le code des serveurs.

Le problème majeur posé par cette approche réside dans les performances du système : la décomposition du système en plusieurs serveurs implique que de nombreuses fonctions nécessitent la participation de plusieurs serveurs et donc de nombreux échanges de messages et changements de contexte. Il est donc nécessaire, afin d'obtenir des performances honorables, de procéder à des optimisations limitant le nombre de serveurs intervenant pour satisfaire une requête.

Les multi-serveurs que nous avons présentés ont choisi d'implémenter certaines fonctionnalités, traditionnellement traitées sous forme d'appels système, dans des fonctions de librairie. Ces fonctions permettent de réduire le nombre d'appels à des serveurs. Malheureusement, tout appel système ne peut pas être transformé en fonction de librairie pour des raisons évidentes de sécurité. En effet, pour de nombreux appels, des vérifications de paramètres et des contrôles d'accès doivent être effectués. De telles vérifications ne peuvent être laissées à la seule responsabilité de la librairie d'émulation sans contrôle du système.

5 Conclusion

Nous avons présenté, dans ce chapitre, les services offerts par le micro-noyau Mach. Mach offre des abstractions de bases :

- les tâches et les *threads* représentant des programmes en cours d'exécution ;
- les ports et les messages permettant de communiquer entre tâches ;
- les objets mémoire permettant d'associer une sémantique à une partie de la mémoire et de la faire gérer par une tâche utilisateur ;
- les hôtes, processeurs et périphériques permettant d'accéder à la machine physique via des primitives fournies par Mach ;

Des outils externes à Mach sont également disponibles afin d'assister le concepteur de système d'exploitation. Des serveurs utilitaires assurent des services de nommage et de communication. Un générateur d'interfaces permet d'implémenter facilement des communications sous forme d'appels de procédures à distance entre clients et serveurs sans avoir à programmer les fonctions fastidieuses d'emballage et de déballage de paramètres. Enfin, la librairie C-Threads offre des outils de synchronisation facilitant la synchronisation de *threads* au sein d'une même tâche.

Nous avons également étudié la structure de plusieurs systèmes d'exploitation qui utilisent Mach. Ces systèmes ont choisi des orientations différentes et reposent sur des structures diverses. La structure la plus intéressante repose sur la coopération de serveurs au dessus de Mach car elle permet une grande extensibilité du système. En contrepartie, elle peut donner lieu à des performances moins bonnes qu'un système monolithique traditionnel dans la mesure où elle implique de nombreuses communications entre serveurs par échanges de messages et de changements de contexte pour satisfaire les requêtes des processus utilisateur.

Nous détaillons dans le chapitre 5 la structure de Masix et les stratégies que nous avons mises en œuvre afin d'optimiser les performances du système.

Chapitre 5

Principes de conception de Masix

1 Introduction

La réalisation d'un système d'exploitation sous forme d'un ou plusieurs processus s'exécutant en mode utilisateur pose de nombreux problèmes que l'on ne rencontre pas dans les systèmes à architecture monolithique. Le premier choix stratégique posé concerne la structuration du système : il est possible d'implémenter le système comme un seul serveur [Places *et al.* 1992] ou comme un ensemble de serveurs coopérants [Guedes et Julin 1991a, Guedes et Julin 1991b].

Les autres problèmes posés concernent la façon dont sont implémentées les communications dans le système, incluant les problèmes de nommage des entités référencées.

Nous exposons tout d'abord les choix opérationnels que nous avons été amenés à faire pour Masix. Nous justifions notamment la structure multi-serveurs de Masix. Nous décrivons également comment le nommage d'entités est implémenté et les protocoles de dialogue entre composants du système. Puis, nous étudions la structure des serveurs de Masix et proposons des optimisations nécessaires pour obtenir des performances se rapprochant de celles d'un système mono-serveur ou à structure monolithique. Ces optimisations reposent sur le fait que la compatibilité source avec un système existant, Unix dans notre cas, nécessite de fournir les mêmes sémantiques pour les fonctions appelées par les processus utilisateur. Ces fonctions peuvent être implémentées de manière différente et plus adaptée à la structure de Masix.

2 Structuration du système

2.1 Système multi-serveurs

Le micro-noyau Mach permet d'implémenter les fonctionnalités d'un système d'exploitation en mode utilisateur [Golub *et al.* 1990]. Cette implémentation peut être réalisée sous la forme d'un seul serveur offrant tous les services du système ou de plusieurs serveurs spécialisés coopérants, comme nous l'avons décrit dans le chapitre 4.

L'implémentation d'un système sous forme de serveur unique permet une réutilisation de la majorité du code du système. Le serveur est relativement facile à réaliser puisqu'il suffit d'encapsuler le code d'origine par des couches d'interface permettant de simuler l'exécution du code en mode superviseur.

Cette approche comporte néanmoins des inconvénients importants :

- Elle se ramène au modèle monolithique et pose donc les mêmes problèmes d'évolution et de maintenance.
- Il est relativement difficile d'étendre dynamiquement les fonctionnalités du système. L'interface multi-environnements de Masix n'est donc pas réalisable facilement avec ce schéma.

Par contraste, une implémentation de système à base de serveurs coopérants permet :

- une mise au point facile de chacun des serveurs indépendamment des autres ;
- une maintenance plus aisée de chacun des serveurs et donc de l'ensemble du système résultant ;
- l'insertion dynamique de nouveaux serveurs dans le système si un mécanisme de gestion de services est inclus.

Ces avantages sont contrebalancés par certains inconvénients :

- le partage d'informations entre les différents composants du système est délicat à implémenter car les serveurs disposent d'espaces d'adressage mémoire différents ;
- la réalisation de services nécessite souvent l'intervention de plusieurs serveurs et implique des communications par échanges de messages et des changements de contexte.

Nous pensons qu'il est important de privilégier la dynamicité et la facilité d'évolution du système en permettant aux composants du système de s'insérer facilement dans l'environnement de travail en cours d'exécution. C'est pourquoi nous avons choisi de structurer Masix sous forme de plusieurs serveurs coopérants, afin de permettre une évolution facile du système et l'ajout dynamique d'environnements différents.

2.2 Types de serveurs

Dans Masix, les appels système sont réalisés par l'envoi de messages à un serveur et l'attente de sa réponse, selon un protocole de type "appel de procédure à distance". Le serveur contacté peut lui-même faire appel à d'autres serveurs pour effectuer certaines opérations.

Nous distinguons trois types de serveurs :

1. les serveurs d'interface recevant les requêtes des processus utilisateur ;
2. les serveurs internes assurant des fonctionnalités utilisées par les autres serveurs ;
3. les serveurs génériques faisant partie du système d'accueil et assurant les services permettant d'offrir plusieurs environnements différents de manière simultanée.

2.3 Nommage et protection d'entités dans Masix

Masix est formé de plusieurs serveurs spécialisés. Chaque serveur est chargé d'assurer un ensemble de fonctionnalités bien définies et de gérer un certain type d'objets. Un serveur implémente donc l'ensemble des opérations qui peuvent être effectuées sur les objets dont il a la charge.

Un serveur doit offrir aux clients un moyen de nommer les objets qu'il gère et doit également maintenir les informations relatives à ces objets. Il doit, de plus, implémenter des mécanismes de protection afin de restreindre l'accès à ces objets aux processus autorisés. Pour cela, plusieurs méthodes peuvent être utilisées :

- un identificateur peut être communiqué à un client lors de la création d'un objet. Chaque message de requête d'un client est émis sur un port unique correspondant au serveur et contient cet identificateur. Le serveur peut retrouver le descripteur correspondant en effectuant une recherche dans une table contenant, pour chaque objet, son identificateur et l'adresse de son descripteur ou le descripteur lui-même. Tous les clients adressent donc leurs requêtes sur le même port quel que soit l'objet accédé et le serveur détermine l'objet sur lequel le traitement doit être effectué en utilisant l'identificateur transmis dans le message de requête.

  Avec cette approche, la table de correspondance représente une ressource critique pour les différents *threads* composant le serveur et l'accès à cette table doit donc être protégé par des mécanismes de synchronisation pendant les recherches et les modifications dans cette table. Ces synchronisations réduisent le degré de parallélisme des *threads* composant le serveur.

  De plus, le serveur doit implémenter un mécanisme d'authentification afin de vérifier qu'un client est autorisé à accéder à un objet qu'il cite. Cette protection peut être assurée en incluant une clé cryptée dans l'identificateur et en la vérifiant lors de chaque requête ;

- afin d'éviter une recherche dans une table de correspondance, un serveur peut transmettre aux clients un identificateur permettant de localiser le descripteur. Cet identificateur peut être l'indice du descripteur dans une table ou son adresse dans l'espace d'adressage du serveur, par exemple. Lors de la réception d'une requête contenant cet identificateur, le serveur peut accéder au descripteur en le verrouillant sans avoir à protéger l'ensemble des descripteurs. De la même façon que dans la méthode précédente, toutes les requêtes sont transmises sur le même port.

De même qu'avec l'approche précédente, le serveur doit vérifier la conformité des identificateurs transmis par les clients avant d'autoriser l'accès aux objets. Cette protection peut être implémentée en stockant dans chaque descripteur d'entité la liste des clients autorisés à y accéder et en vérifiant que le client ayant transmis la requête est compris dans cette liste ;

- une troisième méthode consiste à utiliser le nom du port de requête lui-même comme identificateur d'objet. Toute opération sur un port est effectuée via un nom de port correspondant à un droit d'émission ou de réception sur le port. Mach permet à une tâche d'associer un identificateur quelconque à un droit sur un port. Un serveur peut tirer parti de cette possibilité en utilisant l'adresse du descripteur de l'objet comme nom du port représentant l'objet. Toute requête concernant un objet est émise sur le port qui le représente. Ainsi, lors de la réception d'une requête, le nom du port sur laquelle elle a été reçue permet d'identifier de manière unique l'entité référencée.

  En utilisant cette méthode, le serveur reporte les contrôles d'accès au niveau de Mach. En effet, un client ne peut émettre un message sur un port que si un droit d'émission sur ce port lui a été transmis par le serveur. Ainsi, un client ne peut pas émettre de requêtes concernant un objet dont l'accès ne lui a pas été accordé puisqu'il ne possède pas de droit d'émission sur le port correspondant.

  Cette approche permet, de plus, au serveur de gérer facilement la terminaison des objets qu'il gère. Le descripteur d'un objet doit être conservé tant que des clients peuvent référencer l'objet, c'est à dire tant que des clients possèdent encore un droit d'émission sur le port. Mach permet à une tâche de demander à être prévenue lorsque tous les droits d'émission sur un port ont été désalloués. Un serveur peut tirer profit de cet avertissement en désallouant le descripteur d'un objet lorsqu'il reçoit un message l'informant que plus aucun client ne possède de droits d'émission sur le port représentant l'objet, i.e. lorsque l'objet ne peut plus être référencé par un client.

Nous avons choisi d'utiliser cette dernière solution dans Masix car elle permet de simplifier la programmation des serveurs en utilisant les mécanismes de d'adressage et de protection fournis par Mach.

Toute entité Unix (fichier, processus...) est donc représentée par un port. Le serveur chargé de gérer des objets d'un type défini crée un port lors de l'allocation d'un nouvel objet. Le serveur possède un droit de réception sur le port et transmet un droit d'émission sur le port au client souhaitant accéder à l'objet. Le client peut ensuite transmettre ses requêtes sur le port, le serveur les reçoit et les traîte.

Tout serveur gérant plusieurs objets, ce qui est le cas de la majorité des serveurs de Masix, crée des ensembles de ports. Chacun des ports représente un objet géré par le serveur. Le serveur regroupe en un ensemble les ports de tous les objets qu'il gère et attend la réception de la première requête concernant l'un de ces objets.

2.4 Protocoles de dialogue

Nous définissons plusieurs protocoles de dialogue entre processus clients et serveurs. Un protocole de dialogue caractérise l'ensemble des opérations disponibles sur un type d'entités. Tout serveur gérant des objets de ce type doit offrir les services définis dans le protocole en respectant les interfaces définies.

Lors d'une requête effectuée par un client sur un objet, le client appelle une fonction (générée par MIG, cf chapitre 4) chargée de transmettre un message de requête au serveur gérant l'objet. Cette requête est transmise par un message envoyé sur le port représentant l'objet fourni par le client. Ce message est reçu par le serveur qui possède un droit de réception sur le port, c'est à dire le serveur qui gère l'objet.

Cet adressage d'objets par port permet d'accéder à des objets de même type gérés par des serveurs différents. Les requêtes sont dirigées, de manière transparente pour le client, sur le serveur concerné. De la sorte, il est possible de faire coexister dans Masix plusieurs serveurs gérant des entités de même type et traitant les requêtes avec des algorithmes différents.

Considérons le cas des accès aux fichiers. Une station sous Masix peut "monter" des systèmes de fichiers de types différents, chacun géré par un serveur particulier. Par exemple, certains fichiers peuvent être stockés sur une partition locale alors que d'autres sont accédés par des appels à un serveur de fichiers. Lorsqu'un processus effectue une opération sur un fichier, sa requête est transmise au gestionnaire de fichiers concerné. Celui-ci effectue l'opération en la transformant en opération d'entrée-sortie physique, dans le cas d'un fichier local, ou en un requête envoyée au serveur de fichiers, dans le cas d'un fichier distant.

3 Structuration des serveurs

Un serveur peut être structuré de plusieurs façons :

- sous forme d'une tâche contenant un seul *thread* attendant les requêtes sur les ports correspondant aux objets gérés :

  Un serveur structuré de la sorte est simple à programmer. Il traite les requêtes de manière séquentielle au fur et à mesure de leur arrivée. Les performances d'un tel serveur sont médiocres dans la mesure où le traitement des requêtes est sérialisé : si le *thread* se suspend en attente d'un évênement lors du traitement d'une requête, les autres requêtes sont également placées en attente ;

- sous forme d'une tâche contenant un *thread* attendant les requêtes et plusieurs *threads* chargés de les satisfaire :

  Cette structuration permet de traiter plusieurs requêtes en pseudo-parallélisme et diminue ainsi le temps de réponse du serveur aux requêtes. Les *threads* réalisant les services doivent utiliser des mécanismes de synchronisation pour protéger les données partagées. Le *thread* unique chargé de recevoir les requêtes permet d'ordonnancer les requêtes mais représente un goulot d'étranglement. De plus, dans la mesure où les requêtes sont reçues par un *thread* puis traîtés par un autre, il est obligatoire de procéder à un changement de contexte entre les deux *threads* pour réaliser le service demandé ;

- sous forme d'une tâche contenant plusieurs *threads* attendant les requêtes des clients et chargées de les statisfaire.

  Deux solutions extrêmes peuvent être adoptées :

  - Chaque *thread* peut recevoir des requêtes sur un ensemble de ports correspondant aux entités qu'il gère. Les objets gérés sont alloués statiquement à chaque *thread* et toute opération sur un même objet est toujours traitée par le même *thread*. Cette approche revient à décomposer le serveur en un ensemble de "sous-serveurs" composé d'un *thread* chacun.

    L'inconvénient de cette méthode repose dans l'allocation statique d'objets aux *threads* : dans la mesure où les requêtes sur un objet sont toujours traitées par le même *thread*, les demandes de services émanant des clients ne peuvent pas être uniformément distribuées. Il est donc possible d'obtenir une situation dans laquelle certains *threads* sont constamment sollicités alors que d'autres sont inactifs : si des processus font des requêtes fréquentes sur un sous-ensemble d'objets gérés par un seul *thread*, le temps d'attente moyen pour une requête concernant un objet de ce sous-ensemble sera important.

    De plus, l'emploi de cette stratégie ne permet pas, comme on pourrait le croire au premier abord, de se passer des synchronisations entre *threads*. En effet, bien que deux *threads* gèrent des objets différents, il est possible, voire probable qu'ils accêdent à des données communes.
  - Tous les *threads* du serveur reçoivent les requêtes sur un seul ensemble de ports représentant tous les objets gérés par le serveur. Cette méthode permet de répartir dynamiquement les requêtes entre les *threads* composant le serveur.

Afin d'offrir des performances honorables, nous avons écarté la première possibilité et décidé d'utiliser des serveurs composés de plusieurs *threads* chargés de traîter les requêtes en pseudo-parallélisme. Cette méthode peut être comparée, dans une certaine mesure, à la façon dont les appels système sont implémentés dans Unix : un appel système provoque le passage du processus appelant en mode noyau pour traiter l'opération. A un instant donné, plusieurs processus peuvent se trouver en mode noyau mais un seul peut être actif (tous les autres sont bloqués en attente d'un évènement). Si on compare le noyau d'Unix à un serveur, ce schéma correspond à plusieurs *threads* traitant des requêtes en parallèle.

Une différence doit toutefois être notée : alors qu'Unix ne préempte jamais un processus actif en mode noyau (un changement de contexte n'a lieu que si le processus retourne au mode utilisateur ou s'il se bloque en attente d'un évènement), un serveur n'est pas maître de l'ordonnancement de ses *threads*. A tout moment, Mach peut procéder à un changement de *thread* courant sans qu'il soit possible de l'en empécher. Il est donc nécessaire d'utiliser des synchronisations plus strictes que dans Unix.

En effet, dans Unix, un processus en mode noyau ne peut être interrompu que s'il se place volontairement en attente. Il n'a donc besoin de verrouiller les données qu'il utilise que lorsqu'il est sur le point de se suspendre. Dans Masix, par contre, comme l'ordonnanceur de Mach peut changer de *thread* courant à tout moment, un *thread* contenu dans un serveur doit impérativement verrouiller les données qu'il utilise dés le premier accès à ces informations.

Afin d'utiliser uniformément les *threads* de chaque serveur, nous avons choisi de regrouper tous les objets gérés par le serveur en un ensemble de ports sur lequel tous les *threads* attendent des requêtes. Ceci nous permet de ne pas créer de goulot d'étranglement pour les objets faisant partie d'un sous-ensemble fréquemment accédé.

4 Optimisation de la répartition de services entre librairies et serveurs

Dans les systèmes Unix classiques, les sous-programmes sont traditionnellement divisés en deux catégories :

- les appels système :

  fonctionnalités de bas niveau réalisées par des fonctions, contenues dans le noyau du système, activées par des trappes provoquées par les processus utilisateur ;

- les fonctions de librairie :

  fonctionnalités de haut niveau réalisées par des sous-programmes de services situés dans l'espace d'adressage du processus appelant, avec des appels éventuels au noyau.

Dans Masix, les appels système sont réalisés par l'envoi de messages à un serveur et l'attente de sa réponse, selon un protocole de type "appel de procédure à distance". Le serveur contacté peut lui-même faire appel à d'autres serveurs pour effectuer certaines opérations.

Le temps de traitement d'un appel système, consistant en un appel à un serveur, est considérablement supérieur à celui d'un appel traité par une trappe. En effet, alors qu'une trappe provoque uniquement un changement de mode d'exécution et de l'espace d'adressage d'un processus, un envoi de message provoque un changement de contexte, entre le processus utilisateur et le serveur, qui est beaucoup plus coûteux.

Afin d'obtenir des performances honorables, nous avons donc décidé d'inclure des optimisations dans la réalisation des appels système, notamment en réduisant le nombre d'appels implémentés sous forme d'appels de procédure à distance vers un serveur.

Pour cela, nous avons revu la distinction entre les appels système et les fonctions de librairie usuellement effectuée par le système Unix. Nous avons considéré qu'il n'est pas impératif de respecter cette distinction de manière stricte pour offrir un système compatible Unix au niveau source. En effet, pour un programme utilisateur, il importe peu qu'une fonction appelée soit implémentée sous forme d'appel système ou de fonction de librairie. La seule contrainte à respecter est que la fonction effectue un traitement conforme à la sémantique attendue.

A partir de ce principe, nous avons implémenté certains appels système sous forme de fonctions de librairie. Ces appels sont réalisés par des sous-programmes inclus lors de l'édition de liens. Ces sous-programmes réalisent directement les services demandés sans effectuer d'appel à des serveurs. De tels appels sont éxécutés dans le contexte du processus appelant et ne nécessitent pas de changement de contexte pour être traités. Ils sont donc effectués plus rapidement.

Tout appel système ne peut néanmoins pas être implémenté sous forme de fonction de librairie. Plusieurs contraintes doivent être respectées :

- Certains appels nécessitent une connaissance globale de l'état du système. Leur implémentation sous forme de sous-programmes nécessiterait l'accès à ces informations dans l'espace d'adressage de chaque processus.
- Les paramètres de nombreux appels doivent être vérifiés et authentifiés par le système avant d'autoriser l'appel. Afin de préserver un certain niveau de sécurité dans le système, il est impératif de faire effectuer cette vérification par un serveur indépendant du processus appelant. De la sorte, il est possible d'empêcher un processus d'accéder, par erreur ou de manière intentionnelle, à des données vitales qu'il pourrait modifier si elles étaient placées dans son espace d'adressage.
- Certains appels modifient des informations accessibles à plusieurs processus. Dans la mesure où chaque processus possède un espace d'adressage distinct de celui des autres, il est impossible de faire effectuer cette modification par le processus appelant. Il est alors nécessaire de regrouper ces informations dans l'espace d'adressage d'un serveur et d'y accéder par des requêtes à ce serveur.

5 Optimisation des interactions entre serveurs

La conception de Masix sous forme de plusieurs serveurs coopérants permet une extension facile du système mais elle entraîne des interactions supplémentaires entre les serveurs. Nous avons donc cherché à améliorer l'efficacité du système en optimisant les partages de données et de code dans les serveurs.

5.1 Hypothèses

Avant d'exposer les méthodes d'interactions entre serveurs que nous avons retenues dans Masix, il est nécessaire de rappeler nos hypothèses sur l'exécution des serveurs :

- Un serveur est implémenté sous forme d'une tâche Mach possédant son propre espace d'adressage. Chaque serveur est composé de plusieurs *threads* traitant les requêtes concernant les objets gérés par le serveur. Tout objet est référencé par un port sur lequel les clients transmettent leurs demandes de service.

- À l'intérieur d'un même serveur, les *threads* partagent le même espace d'adressage et accèdent à des données communes. Les mécanismes de synchronisation offerts par la librairie C-Threads (mutex) sont utilisés pour protéger l'accès à ces données.
- Les demandes de service entre clients et serveurs sont synchrones : les clients utilisent des appels de procédure à distance sur les serveurs. Ils sont donc suspendus entre l'émission du message de requête au serveur et la réception du message de réponse depuis le serveur.
- Les clients et les serveurs utilisent les mécanismes de communication fournies par Mach. Ces mécanismes permettent une transmission fiable des messages entre tâches. Aucun message ne peut être perdu et l'ordonnancement de messages émis par la même tâche est assuré par Mach.
- En cas d'arrêt d'un serveur, les autres tâches peuvent détecter la panne. En effet, dans ce cas, les droits de réception sur ports détenus par le serveur sont désalloués et toutes les tâches possédant un droit d'émission sur ces ports peuvent être prévenues par Mach de la disparition des ports.

5.2 Optimisations des accès aux données partagées

Chaque serveur a besoin de maintenir des informations relatives aux objets qu'il gère et deux serveurs ont souvent besoin d'accéder aux mêmes informations. Par exemple, un serveur de processus et un serveur de fichiers ont tous deux besoin de connaître l'identité de l'utilisateur sous laquelle s'exécute un processus. Le serveur de processus utilise cette identité pour vérifier la conformité des envois de signaux et le serveur de fichiers l'utilise pour vérifier les accès aux fichiers.

Dans le cas d'un système monolithique comme Unix, les informations partagées sont stockées dans des tables accessibles à l'ensemble des composants du système qui y accèdent dans leur espace d'adressage commun. Dans un système multi-serveurs comme Masix, on peut procéder à une répartition des informations gérées entre plusieurs serveurs : chaque serveur maintient uniquement les données qui lui sont nécessaires. Le partage d'informations entre serveurs est alors un problème délicat et il est nécessaire de le gérer de manière optimale. Plusieurs méthodes peuvent être envisagées :

- le regroupement des informations dans des tables gérées par un serveur particulier.

  Tout serveur désirant accéder à ces données, en lecture ou en écriture, doit transmettre une requête au gérant de la table. Cette méthode permet alors de gérer le partage simplement. Il n'y a plus de partage réel et présente un inconvénient important puisqu'elle implique que :

  - tout accès à une information "partagée" doit se faire par un appel à un serveur qui représente un goulot d'étranglement ;
  - tout appel système nécessitant l'accès à des données communes fait intervenir un serveur supplémentaire et provoque donc un changement de contexte supplémentaire ;

- le partage des informations entre tous les serveurs concernés.

  Grâce aux fonctionnalités de gestion de la mémoire de Mach, le partage de pages mémoire entre tâches permet de placer les données communes dans l'espace d'adressage de chaque serveur intéressé qui peut y accéder directement. Cette méthode permet un accès rapide aux informations qui ne doivent pas être modifiées mais nécessite des synchronisations entre serveurs lorsque des données doivent être modifiées afin de garantir la cohérence des informations. Ces synchronisations entre serveurs, implémentés dans des tâches distinctes, présentent un inconvénient similaire à celui cité dans la méthode précédente ;

- la duplication des informations dans chaque serveur concerné.

  Chaque serveur maintient une copie des données dont il a besoin et peut ainsi y accéder directement. La modification des informations par un serveur implique la diffusion des ces modifications sur tous les autres serveurs afin de respecter la cohérence des données.

Afin de choisir les stratégies optimales de gestion des informations communes, i.e. les stratégies qui donnent lieu au moins de messages possibles, nous avons étudié les différentes possibilités. Si l'on suppose qu'une donnée est accédée en lecture N_l fois et modifiées N_e fois, on peut envisager les stratégies suivantes :

1. Un serveur maintient les informations et les autres doivent lui transmettre une requête pour y accéder. On a alors N_l communications.
2. Un serveur maintient les informations et chaque autre serveur gère un cache contenant ces données. En cas de modification des données, une invalidation des caches est effectuée. Dans ce cas, on a $N_l + N_e$ communications (N_e requêtes d'invalidation de cache et N_l requêtes pour obtenir la nouvelle valeur des informations).
3. Un serveur maintient les informations et chaque autre serveur gère un cache contenant ces données. En cas de modification des données, leur nouvelle valeur est transmise aux autres serveurs afin de mettre leur cache à jour. Dans ce cas, aucun serveur n'effectue de requêtes pour obtenir les informations et on a N_e communications nécessaires pour propager les nouvelles valeurs après modifications.
4. Un serveur maintient les informations et chaque autre serveur gère un cache contenant ces données. En cas de modification des données, une invalidation du cache d'un autre serveur est effectuée si ce serveur a déjà accédé aux informations. Dans ce cas, on a $2min(N_l, N_e)$ communications entre serveurs : une invalidation du cache d'un serveur est effectuée à chaque modification si le serveur a accédé à l'information depuis sa mise à jour précédente.
5. Un serveur maintient les informations et chaque autre serveur gère un cache contenant ces données. En cas de modification des données, une invalidation du cache d'un autre serveur est effectuée si ce serveur a déjà accédé aux informations et la nouvelle valeur des données lui est transmise. Dans ce cas, on a $min(N_l, N_e)$ communications entre serveurs : une invalidation du cache d'un serveur est effectuée à chaque modification si le serveur a accédé à l'information depuis sa mise à jour précédente et il n'est jamais nécessaire pour le serveur de demander à nouveau la valeur de la donnée.

Cette analyse des communications nous a amenés à mettre en place plusieurs méthodes de partage d'information selon la fréquence et le mode d'accès aux données :

- Les données accédées uniquement en lecture (par exemple, l'identificateur d'un processus, qui ne change pas au cours de son exécution) sont dupliquées dans chaque serveur. Ainsi, tous les serveurs peuvent accéder aux informations directement et aucun problème de cohérence ne se pose. Aucune communication entre serveurs n'est nécessaire pour gérer les données. Seule l'initialisation des informations partagées donne lieu à des échanges de messages afin de diffuser la valeur des données aux différents serveurs concernés.

- Les données fréquemment lues et peu souvent modifiées (par exemple, l'identificateur d'utilisateur associé à un processus) sont dupliquées dans chaque serveur qui y accède en lecture directement. La modification des informations résulte d'un appel système traité par un serveur qui transmet alors leurs nouvelles valeurs aux autres serveurs. Cette méthode correspond à la troisième solution présentée précédemment : on a donc N_e messages de mise à jour qui est le nombre minimal de communications quand $N_e < N_l$.

  Le problème lié à cette solution est le risque d'incohérence des informations pendant la mise à jour. En effet, entre le moment où les données ont été modifiées sur un serveur et celui où elles ont été mises à jour sur l'ensemble des serveurs, des requêtes accédant à ces données peuvent être traitées. Il est donc possible que certains serveurs utilisent les anciennes valeurs alors que d'autres serveurs utilisent déjà les nouvelles. Ce problème est délicat en théorie mais peut être aisément résolu en pratique.

  En fait, l'important n'est pas d'assurer la cohérence entre les informations communes à différents serveurs mais de garantir que ces informations ne sont pas modifiées pendant le traitement d'une requête qui les utilise, c'est à dire qu'un serveur ne doit pas prendre en compte la modification des données utilisées par une requête en cours. Afin de n'effectuer les modifications d'informations que lorsque ces dernières ne sont pas en cours d'utilisation, il est possible d'utiliser un mécanisme de verrouillage : toute requête utilisant des données utilise les fonctionnalités de synchronisation de la librairie C-threads afin de protéger l'accès aux informations nécessaires.

  Tant que des informations sont utilisées par un *thread* du serveur, leur accès est protégé par un mutex et la mise à jour des données est différée jusquà ce que le *thread* qui y accède relache le mutex.

- Les données plus fréquemment modifiées que lues sont placées dans l'espace d'adressage d'un serveur seul autorisé à les modifier. Les autres serveurs accèdent aux informations communes en transmettant des requêtes à ce serveur.

  Cette méthode correspond à la première solution présentée plus haut. Elle permet d'obtenir un nombre de messages égal à N_l qui est le nombre minimal de communications entre serveurs quand $N_l < N_e$.

5.3 Coopération entre serveurs

Un appel système implique souvent la coopération de serveurs. Par exemple, la création de processus (appel système `fork` d'Unix) nécessite de dupliquer le contexte du processus appelant.

Cette duplication nécessite que chaque serveur maintenant des informations sur le processus mette à jour ses données et il est donc nécessaire de faire participer plusieurs serveurs pour réaliser l'opération.

Par ailleurs, de nombreuses tâches communes doivent être effectuées par chaque serveur lors de la réalisation d'appels système. Deux méthodes peuvent être utilisées :

1. le regroupement de ces sous-programmes dans un serveur interne : un nouveau serveur est créé et offre des fonctions de réalisation de toutes les tâches communes. Les autres serveurs doivent effectuer un appel de procédure à distance pour exécuter ces fonctions ;
2. la duplication de ces fonctions dans les différents serveurs qui en ont besoin.

La première méthode permet de respecter la structuration du système mais entraine des échanges de messages et des changements de contexte supplémentaires. La deuxième ne présente pas ce dernier inconvénient mais augmente l'espace mémoire utilisé par les serveurs et pose des problèmes de cohérence des sous-programmes : si les sous-programmes utilitaires sont inclus dans chaque serveur, il faut veiller à mettre à jour tous les serveurs concernés lorsqu'une fonction est modifiée.

Nous avons choisi une solution de compromis entre les deux méthodes :

- Tout sous-programme utilitaire commun nécessitant la connaissance d'informations globales au système est implémenté par un serveur. Ainsi, le sous-programme peut manipuler les données globales, maintenues par le même serveur.
- Tout sous-programme utilitaire commun ne nécessitant que l'accès aux informations locales à un serveur est dupliqué dans chaque serveur qui en a besoin. De la sorte, son exécution est effectuée dans le contexte du serveur et ne nécessite pas d'échange de messages ni de changements de contexte. Une optimisation de cette métode consisterait à placer le code de ces sous-programmes dans un espace mémoire partagé entre les différents serveurs, i.e. dans une librairie partagée, en utilisant les fonctionnalités de partage de la mémoire fournis par Mach. De la sorte, chaque serveur accèderait directement à ces sous-programmes qui ne seraient chargés qu'une seule fois en mémoire.

6 Conclusion

Nous avons exposé, dans ce chapitre, les principes de conception du système Masix. Nous avons tout d'abord présenté la structure générale du système qui est formé de plusieurs serveurs coopérants. Chacun des serveurs est chargé d'implémenter un protocole constitué de l'ensemble des opérations disponibles sur les objets qu'il gère. Chaque serveur est chargé du nommage et de la protection d'accŝ à ses objets.

La transmission de requêtes à un serveur, selon le protocole défini, est effectuée via un port représentant l'objet sur lequel l'opération doit être exécutée. Cet adressage de serveur par port d'objet géré permet de faire coexister dans le système plusieurs serveurs gérant des objets de même type selon des sémantiques différentes. L'envoi de messages sur un port transmet directement la requête au serveur concerné.

Nous avons également défini les optimisations nécessaires pour minimiser les changements de contexte et échanges de messages lors de la réalisation d'appels système. Pour cela, la distinction entre appels système et fonctions de librairie a été revue afin de réduire le nombre d'opérations donnant lieu à une requête à un serveur. Des stratégies optimisées d'accès aux données partagées ont aussi été exposées : selon le type et la fréquence des accès aux informations partagées, ces dernières sont dupliquées ou non et des protocoles de gestion de cache sont implémentés par les serveurs.

Les principes énoncés ont été respectés pour la conception du système local de Masix qui est présenté dans le chapitre 6.

Chapitre 6

Le Système local

1 Introduction

Masix est composé d'un ensemble de serveurs coopérants assurant les services nécessaires à l'exécution simultanée d'environnements différents ainsi que des fonctionnalités respectant des fonctionnalités Unix.

Le système d'accueil est implémenté par un serveur chargé de tenir à jour la liste des serveurs lancés et de gérer le système. Les fonctionnalités Unix sont assurées par un ensemble de serveurs coopérants chargés de gérer des entités (processus, mémoire, fichiers...) selon les sémantiques d'Unix.

Nous rappelons tout d'abord les services offerts par le système local. Nous exposons ensuite les principes que respectent les composants de Masix en détaillant le fonctionnement de chaque serveur. Enfin, nous présentons la structure générale du système local de Masix en détaillant les interactions entre serveurs.

2 Services offerts

2.1 Système d'accueil

Un des objectifs de Masix est d'être configurable de manière dynamique et de supporter l'exécution parallèle d'environnements différents. Dans le chapitre 3, nous avons défini la notion de système d'accueil qui doit assurer tous les services de base permettant la gestion de ces environnements.

Nous détaillons maintenant les services qui doivent être assurés par ce système d'accueil :

- il doit à, tous moments, avoir connaissance des environnements disponibles dans Masix. Il doit donc gérer le lancement d'un nouvel environnement, son accessibilité et sa terminaison ;
- il doit offrir les services de communication entre des processus utilisateur et les environnements qui s'exécutent dans le système. Pour ce faire, il doit offrir des mécanismes permettant aux processus de transmettre des requêtes aux serveurs composant ces environnements. Il doit, de plus, veiller à ce que les environnements nécessaires aux processus en cours d'exécution soient toujours présents dans le système ;
- il doit inclure des mécanismes permettant aux divers environnements de se partager les ressources du système.

2.2 Environnement Unix

Comme nous l'avons défini dans les chapitres précédents, Masix doit offrir des fonctionnalités et sémantiques compatibles avec celles du système Unix :

- systèmes de fichiers hiérarchiques rendus accessibles par des opérations de montage logique ;
- processus s'exécutant en pseudo-parallélisme ;
- communications locales (tubes, signaux, mémoire partagée, sémaphores, files de messages) et distantes (*sockets*) entre processus ;
- gestion de mémoire virtuelle ;
- appels système respectant les sémantiques Unix.

Cette compatibilité Unix doit être assurée par un ensemble de serveurs formant un environnement parmi d'autres. Bien que cet environnement soit le premier que nous envisagions dans Masix et qu'il soit le seul à être détaillé dans ce chapitre, il n'a aucun statut privilégié. D'autres environnements offrant les sémantiques d'autres systèmes ou de nouvelles fonctionnalités pourront être intégrés parallèlement dans Masix sans lien et sans répercussion sur l'interface Unix.

3 Principes et stratégies

3.1 Communications entre serveurs

Masix est un système d'exploitation dans lequel plusieurs serveurs s'exécutent en parallèle et coopèrent pour fournir des services aux autres processus. Cette coopération nécessite des échanges de données, la transmission de requête et le renvoi de résultats entre serveurs.

Chaque serveur est constitué d'une tâche Mach et possède un espace d'adressage distinct de celui des autres. La seule méthode de communication utilisable entre serveurs repose sur la transmission de messages sur des ports. Cette communication nécessite la connaissance réciproque des ports gérés par chaque serveur et donc un système de nommage.

3.2 Système d'accueil

Le système d'accueil est à la base de Masix. En effet, il est chargé de gérer les différents environnements du système et joue donc un rôle central.

3.2.1 Gestion des environnements : Le Serveur d'Administration et de Configuration

La gestion d'environnements multiples nécessite le maintien des serveurs lancés dans le système. Le système d'accueil joue donc un double rôle :

1. il assure que tous les serveurs nécessaires sont présents dans le système ;
2. il offre les moyens de communiquer entre clients et serveurs.

Dans Masix, ces deux rôles sont tenus par un serveur chargé de tenir à jour les serveurs lancés. Ce serveur, appelé "Serveur d'Administration et de Configuration" (SAC) est activé lors du démarrage du système avant toute autre tâche. Il est ensuite chargé de lancer les différents autres serveurs nécessaires à l'exécution du système.

Le SAC, une fois lancé, maintient un service de noms : il est chargé de connaître tous les serveurs qui s'exécutent dans le système, de contrôler leur exécution et de permettre aux clients d'y accéder. Pour ce faire, il crée un port sur lequel il possède un droit de réception et transmet ce port par héritage à toutes les tâches créés (serveurs ou clients).

Lorsqu'un serveur démarre son exécution, il effectue son initialisation en plusieurs étapes :

1. il initialise ses données internes ;
2. il crée un port surt lequel il possède un droit de réception. Ce port sera ensuite utilisé pour recevoir les requêtes des clients ;
3. il envoie un message au SAC en lui transmettant :
   - une chaîne de caractères qui le caractérise ;
   - un booléen indiquant si sa présence est indispensable au bon fonctionnement du système ;
   - un droit d'émission sur son port de requête ;

4. il crée les différents *threads* chargés de traiter les requêtes des clients ;
5. les *threads* se placent en attente de requêtes sur le port créé lors de la deuxième étape.

Le SAC maintient une table contenant les informations liées à chaque serveur enregistré. Il accepte plusieurs types de requêtes :

1. les demandes d'enregistrement de serveurs qu'il traite en ajoutant les informations transmises dans sa table ;
2. les demandes de clients :

   lorsqu'un processus client souhaite transmettre une demande de services à un serveur, il lui faut connaître son port de requête. Pour ce faire, il envoie un message au SAC contenant la chaîne identifiant le serveur. Le SAC effectue une recherche dans sa table et envoie en retour un droit d'émission sur le port de requête du serveur ;
3. les demandes de terminaison de serveur qu'il traite en supprimant les informations liées au serveur de sa table.

Le SAC joue également un rôle de surveillance du bon fonctionnement du système : il est chargé de veiller que les serveurs nécessaires sont toujours présents. Pour détecter la disparition d'un serveur, il utilise la possibilité d'être prévenu par le micro-noyau Mach de la disparition du droit de réception sur le port de requête de chaque serveur. Ainsi, si un serveur tombe en panne, le SAC en est prévenu et peut prendre les mesures nécessaires. Ces mesures sont de deux ordres :

1. dans le cas d'un serveur non indispensable au système et non utilisé, le SAC se contente de supprimer les informations correspondantes de sa table ;
2. dans le cas d'un serveur marqué comme indispensable au système, le SAC est alors chargé de terminer l'exécution de Masix. Pour ce faire, il transmet à chaque serveur enregistré un message lui indiquant que le système est en cours d'arrêt. Chaque serveur doit, lors de la réception de ce message, sauvegarder ses données si nécessaire et terminer son exécution. De la sorte, le SAC provoque l'écriture par chaque serveur des données modifiées puis, lorsque tous les serveurs se sont arrétés, il réinitialise le système pour repartir dans un état stable.

3.2.2 Transmission de requêtes aux serveurs

Nous avons présenté, dans le paragraphe précédent, la façon dont les clients s'adressent au SAC afin de connaître le port de requête correspondant à un serveur. Afin de minimiser le nombre de communications entre un client le SAC, les fonctions d'envoi de requêtes au serveur gêrent un cache des ports de requêtes : le port de requête d'un serveur n'est demandé au SAC que lors de l'envoi de la première demande de service. Une fois ce port connu, il est sauvegardé dans une variable globale ce qui permet de transmettre les requêtes suivantes au serveur sans re-demander son port.

Considérons l'exemple d'une fonction d'interface avec un serveur de processus enregistré sous le nom `process_server`. Cette fonction a besoin de demander le port de requête du serveur au SAC lors de son premier appel puis elle sauvegarde ce port dans une variable globale. Les appels suivants utilisent la valeur contenue dans cette variable globale. La figure 6.1 représente le code de cette fonction.

```
/* Port du serveur de processus */
mach_port_t proc_server_port = MACH_PORT_NULL;

/* Port du SAC */
extern mach_port_t sac_port;

int fonction (p1, p2, ..., pn)
{
      kern_return_t r;
      int err;

      if (proc_server_port == MACH_PORT_NULL)
      {
            /* Si port du serveur non connu, requete au SAC */
            r = server_port_lookup (sac_port, "process_server",
                  & proc_server_port);
            if (r != KERN_SUCCESS)
            {
                  /* Serveur non enregistre aupres du SAC */
                  mach_error ("server_port_lookup", r);
                  exit (1);
            }
      }
      /* Transmission de requete au serveur */
      server_fonction (proc_server_port, p1, p2, ..., pn, & err);
      return err;
}
```

Figure 6.1: Fonction d'interface à un serveur

Dans cet exemple, la fonction `fonction` ne demande le port du serveur au SAC que lors du premier accès au serveur puis le stocke dans la variable `proc_server_port`. Lors des appels suivants, cette variable contient le port et la fonction appelle directement `server_fonction` qui envoie un message de requête au serveur.

Dans cet exemple, la variable `proc_server_port` n'est accédée que par cette seule fonction. En réalité, dans Masix, toutes les fonctions d'interface à un même serveur partagent une seule variable globale contenant le port de requête du serveur. Ainsi, seul le premier envoi de messages à un serveur pour un processus donne lieu à une requête au SAC.

Lorsqu'un processus se termine, il transmet au SAC un message par port de requête demandé au cours de son exécution afin de le prévenir qu'il n'utilise plus le serveur. Le SAC peut ainsi gérer facilement, dans sa table de serveurs enregistrés, le nombre de clients faisant accès à chaque serveur : ce nombre est initialisé à 0 lorsque le serveur s'enregistre, est incrémenté à chaque demande de port par un client et décrémenté lorsqu'un client signale qu'il n'utilise plus le serveur lors de sa terminaison. De la sorte, le SAC peut connaître, à tous moments, les serveurs en cours d'utilisation dans le système. Il est ainsi à même de :

- décider si la panne d'un serveur a des répercussions sur le bon fonctionnement du système même si le serveur n'est pas déclaré comme indispensable au fonctionnement du système. En effet, si un serveur, même secondaire, tombe en panne alors que des clients l'utilisent, il est impératif de procéder à un arrêt d'urgence du système ;
- autoriser ou non la terminaison d'un serveur : tout serveur qui se termine envoie un message au SAC afin de se désenregistrer. Si ce serveur est encore en cours d'utilisation par des clients, le SAC choisit de ne pas honorer la requête. Si le serveur décide néanmoins de terminer son exécution, créant ainsi un état instable, le SAC en est prévenu par Mach car il perd le droit d'émission sur le port de requête du serveur. Il adopte alors le même traitement que dans le cas précédent en provoquant une réinitialisation du système.

3.3 Gestion des processus

Mach fournit la notion de tâche et de *thread* pour représenter les programmes en cours d'exécution. Ces deux abstractions peuvent être utilisées afin de représenter un processus Unix. En effet, une tâche contenant un seul *thread* correspond à la sémantique Unix d'un processus.

Les fonctionnalités d'un système d'exploitation relatives aux processus peuvent être décomposées en deux niveaux :

1. les mécanismes internes permettant l'exécution de processus en pseudo-parallélisme. Ces mécanismes comprennent les routines de traitements d'interruptions et de trappes, la gestion de la mémoire allouée à chaque processus et l'ordonnancement des processus ;
2. les mécanismes de haut niveau offerts aux processus pour satisfaire les appels système.

Le micro-noyau Mach fournit l'ensemble des mécanismes internes sous une forme proche d'Unix[1] et l'environnement Unix peut les utiliser en les paramétrant.

En revanche, Mach ne fournit pas les mécanismes de haut niveau attendus par un processus Unix. L'environnement Unix doit donc les assurer. Pour ce faire, il doit :

- assurer l'interface des appels système Unix en utilisant les fonctionnalités de Mach pour les réaliser ;

[1] parce que ces mécanismes ont été dérivés de la version BSD d'Unix.

- maintenir les informations relatives aux processus qui ne sont pas gérées par Mach (par exemple, l'identificateur de processus, l'identificateur d'utilisateur...).

Nous avons choisi de faire gérer les processus par un serveur spécialisé. Ce serveur est chargé de traiter toutes les requêtes de manipulation de processus :

- création (`fork`) et terminaison (`_exit`) ;
- exécution de programme (`execve`) ;
- synchronisation de processus (`wait, waitpid`) ;
- modification d'informations liées à un processus (`setgroups`, `setlogin`, `setpgid`, `setpriority`, `setregid`, `setreuid`, `setsid`) ;
- envoi et traitement de signaux (`kill`, `sigaction`, `sigpause`, `sigpending`, `sigprocmask`, `sigsuspend`).

On remarquera que ce serveur ne gère pas les appels système d'obtention d'informations sur le processus. En effet, ces appels sont implémentés sous forme de fonction de librairie afin d'éviter un appel de procédure à distance au serveur. À titre d'exemple, considérons les fonctions `getgroups` et `setgroups`. Dans Unix, ces deux fonctions sont implémentées sous forme d'appels système modifiant ou consultant une table du noyau. Dans Masix, seule la fonction `setgroups` est implémentée comme un appel au serveur alors que la fonction `getgroups` est une fonction de librairie.

Comme on le voit dans le code présenté sur la figure 6.2, la fonction `setgroups` effectue d'abord un appel au serveur de processus (via la fonction `server_setgroups`) puis, si la requête a été traitée sans erreur par le serveur, effectue une copie des groupes spécifiés dans un tableau situé dans l'espace d'adressage du processus appelant. La fonction `getgroups` peut tirer parti de ces informations directement accessibles comme le montre son code présenté sur la figure 6.3.

3.4 Gestion des signaux

Le système Unix fournit plusieurs mécanismes de communication entre processus :

1. les tubes ;
2. les signaux ;
3. la mémoire partagée ;
4. les sémaphores ;
5. les files de messages ;
6. les *sockets*.

```
extern int errno;

extern int lib_ngroups;
extern gid_t lib_groups [];

int setgroups (int ngroups, gid_t gidset [])
{
      int err;
      int i;

      /* Appel au serveur de processus */
      server_setgroups (ngroups, gidset, & err);
      /* Test du code de retour */
      if (err == 0)
      {
            /* Si OK, copie dans le tableau local */
            memcpy (lib_groups, gidset, ngroups * sizeof (gid_t));
            lib_ngroups = ngroups;
            return 0;
      }
      else
      {
            /* Si erreur, positionnement de errno */
            errno = err;
            return -1;
      }
}
```

Figure 6.2: Code de la fonction `setgroups`

```
extern int lib_ngroups;
extern gid_t lib_groups [];

int getgroups (int gidsetlen, gid_t gidset [])
{
      int n;

      if (gidsetlen > 0)
      {
            n = (gidsetlen < lib_ngroups) ? gidsetlen : lib_ngroups;
            memcpy (gidset, lib_groups, n * sizeof (gid_t));
            return n;
      }
      else
            return lib_ngroups;
}
```

Figure 6.3: Code de la fonction `getgroups`

De ces six mécanismes, seuls les signaux sont implémentés par le serveur de processus. Les autres mécanismes sont assurés par des serveurs spécialisés.

3.4.1 Signaux synchrones et asynchrones

Nous distinguons deux types de signaux :

- les signaux synchrones :

  ces signaux sont envoyés au processus courant en réponse à une erreur de sa part (accès mémoire incorrect, division par zéro...) ;

- les signaux asynchrones :

  ces signaux sont envoyés par le processus courant à un autre processus par l'appel système `kill`.

Ces deux types de signaux doivent être gérés de manière différente : alors que les signaux asynchrones sont envoyés par la fonction `kill` qui transmet une requête au serveur de processus, les signaux synchrones correspondent à des erreurs détectées par Mach qui les gère lui-même. La gestion des erreurs par Mach consiste à envoyer un message décrivant l'erreur sur le port d'exception du *thread* fautif. Afin que le serveur de processus soit informé de ces exceptions, il possède le droit de réception sur le port d'exception de tout processus créé. De la sorte, une exception causée par un processus est gérée par le serveur de processus.

Le serveur de processus est donc appelé aussi bien dans le cas des signaux synchrones (via la réception d'un message sur le port d'exception des *threads* représentant les processus) que dans le cas des signaux asynchrones (via un message de requête envoyé par la fonction `kill`). Il peut ainsi effectuer un traitement commun à ces deux types de signaux.

Sous Unix, le traitement d'un signal provoque l'exécution par le processus cible d'une fonction de traitement. Lors de l'envoi d'un signal à un processus, Unix marque l'arrivée du signal dans le descripteur du processus. Le système vérifie ensuite qu'un processus a reçu un signal à chaque fois qu'il sort du mode noyau, i.e. à la fin du traitement d'un appel système ou lorsque le processus est choisi par l'ordonnanceur pour être exécuté.

Dans le cas de Masix, cette stratégie ne peut pas être utilisée. En effet, l'ordonnancement des processus (des *threads* utilisés pour représenter les processus en fait) est effectué par Mach sans intervention du serveur de processus. Ce dernier n'intervient donc pas sur l'exécution d'un processus qui ne fait aucun appel système et donc aucun appel aux serveurs de Masix. Le serveur de processus doit donc pouvoir intervenir sur des processus en cours d'exécution en mode utilisateur.

3.4.2 Interruption des appels système

Un autre problème posé par les signaux est l'interruption d'appel système en cours. Dans Unix, lors du traitement d'un appel système, un processus peut se suspendre en mode noyau et être réveillé soit par l'arrivée de l'évênement attendu soit par l'arrivée d'un signal qui interrompt alors l'appel système en cours. Dans ce dernier cas, Unix utilise un équivalent de la fonction `longjmp` pour faire revenir le processus en mode utilisateur afin de traiter le signal. Dans le cas d'un système multi-serveurs comme Masix, l'interruption d'appel système est beaucoup plus complexe à gérer. En effet, lors d'un appel système, un processus est suspendu et un *thread* contenu dans un serveur traîte sa requête. L'interruption d'un appel système consiste donc à interrompre le *thread* de chaque serveur impliqué dans le traîtement de l'appel système.

Afin de permettre l'interruption de ces *threads*, nous sommes partis du principe que dans Unix, peu d'appels système provoquent la suspension du processus appelant avec possibilité d'interruption par un signal. De la même façon, dans Masix, peu de traitements d'appels système nécessitent un mécanisme d'interruption et il n'est donc pas nécessaire de prévoir une optimisation poussée de ces cas.

Afin de pouvoir, à tous moments, interrompre un appel système bloquant, le serveur de processus doit tenir trace des *threads* impliqués dans le traitement de l'appel système. Cela nécessite que chaque *thread* concerné communique son identité, ou tout au moins un moyen de le référencer, au serveur de processus.

La stratégie adoptée dans Masix est la suivante :

- tout *thread* contenu dans un serveur et susceptible de se suspendre de manière interruptible pendant le traitement d'un appel système crée un port et transmet un droit d'émission sur ce port au serveur de processus. Ce dernier conserve ce port tant que l'appel système est en cours ;

- le *thread* se suspend ensuite en attente d'un événement, i.e. il attend un message sur un port correspondant à l'événement. En fait, afin de pouvoir être interrompu par le serveur de processus, il attend un message sur un ensemble de ports contenant le port de l'événement et le port transmis au serveur de processus ;
- en cas de signal, le serveur de processus envoie un message sur le port communiqué par le *thread* du serveur, ce qui provoque le réveil de ce *thread* qui termine le traitement de l'appel système en cours en renvoyant un message contenant le résultat partiel de l'appel et le code d'erreur EINTR (appel système interrompu) au processus appelant.

3.4.3 Exécution du traîtement associé à un signal

Nous avons exposé dans les paragraphes précédents comment le serveur de processus est prévenu de l'arrivée d'un signal et comment il gére l'interruption d'appels système en cours. Le dernier point à considérer est la méthode utilisée pour traiter un signal dans un processus.

Trois méthodes peuvent, à priori, être utilisées :

1. tester l'arrivée d'un signal et le traiter au retour de chaque appel système :

   cette méthode, similaire à celle utilisée dans Unix, n'est pas applicable dans Masix. En effet, un processus peut passer un temps indéterminé sans effectuer d'appel système et le traitement d'un signal peut ainsi être différé indéfiniment ;

2. utiliser un deuxième *thread* dans le contexte du processus pour traiter le signal :

   le serveur de processus, lors de la réception d'un signal par un processus, peut suspendre le *thread* représentant le processus et créer un deuxième *thread* chargé d'exécuter la fonction associée au signal. Lorsque ce deuxième *thread* se termine, i.e. lorsqu'il a fini de traiter le signal, le premier est réveillé et reprend l'exécution au point où il en était avant la réception du signal. Bien entendu, si le traitement du signal a provoqué la terminaison du processus, le premier *thread* n'est pas réveillé.

   Cette méthode comporte un inconvénient lié au fait que chaque *thread* d'une tâche utilise sa propre pile. Considérons le programme représenté sur la figure 6.4. Ce programme associe la fonction `handler` au signal `SIGUSR1` puis s'envoie lui-même ce signal. La fonction `handler` accède à la variable `p` contenant l'adresse d'une variable `i` locale à la fonction `main`. Cette variable locale `i` est allouée dans la pile du *thread* exécutant `main`. Si un deuxième *thread* est chargé de traiter le signal `SIGUSR1` et donc d'exécuter la fonction `handler`, la valeur qui sera affichée ne correspondra pas à celle attendue car `p` pointera sur une donnée contenue dans la pile du deuxième *thread* et non sur `i` contenu dans la pile du premier *thread*.

   Avec cette méthode, il est donc impossible d'assurer la même sémantique qu'Unix pour le traitement de signaux ;

3. dérouter l'exécution du processus vers la fonction de traitement du signal :

   cette méthode se rapproche de celle utilisée dans Unix : le serveur de processus sauvegarde l'état du *thread* représentant le processus, sauvegarde l'adresse de l'instruction en cours d'exécution dans la pile du *thread* puis modifie son compteur ordinal pour pointer sur la première instruction de la fonction de traitement du signal. Lorsque le processus a fini d'exécuter cette fonction, son exécution reprend naturellement à l'instruction où il avait été interrompu.

   Cette méthode assure exactement la sémantique Unix et permet, de plus, de traiter l'arrivée d'un signal pendant le traitement d'un autre, i.e. d'empiler les traitements de signaux.

```
#include <signal.h>

static int * p;

void handler *(void)
{
      printf (``*p = %d'', *p);
}

void main (void)
{
      int i = 3;

      p = & i;
      signal (SIGUSR1, handler);
      kill (getpid (), SIGUSR1);
}
```

Figure 6.4: Exemple de programme utilisant des signaux

3.5 Gestion de la mémoire

3.5.1 Gestionnaires de mémoire définis dans Masix

Afin de gérer la mémoire allouée aux processus, nous avons choisi d'utiliser la possibilité de définir des paginateurs externes à Mach. En fait, nous n'utilisons pas le paginateur par défaut contenu dans le micro-noyau car :

- il ne sait que gérer un fichier d'échange placé sur un système de fichiers compatible avec BSD Unix. Dans la mesure où le système de fichiers de Masix repose sur une nouvelle implémentation, il nous a semblé préférable de ne pas utiliser ce paginateur ;

- il ne sait qu'utiliser un fichier d'échange pré-alloué : il n'est pas capable d'allouer de nouveaux blocs et d'accroître ainsi l'espace de mémoire virtuelle.

Nous avons considéré que l'espace d'adressage d'un processus est formé de plusieurs composants :

- le segment de code contenant les instructions binaires à exécuter chargé depuis le fichier contenant le programme exécutable ;
- le segment de données contenant :
  - les données initialisées chargées depuis le fichier exécutable ;
  - les données non initialisées dont la valeur est mise à 0 la première fois qu'elles sont référencées ;
- le segment de pile alloué page par page lorsqu'il est référencé.

Pour gérer ces différents segments, nous avons défini deux paginateurs :

1. un paginateur chargé de gérer les données et la pile des processus. Ce paginateur doit gérer une zone d'échange, y sauvegarder les pages modifiées et charger des pages depuis cette zone, en réponse aux messages émis par Mach ;
2. un paginateur chargé de gérer le code des processus. Masix interdit aux processus de modifier leur code en cours d'exécution et ce paginateur n'a donc pas besoin d'implémenter de sauvegarde de page. Il doit juste assurer qu'une page de code nécessaire pourra être chargée. Dans la mesure où les pages de code ne sont jamais modifiées, il n'est pas nécessaire, pour ce paginateur, d'utiliser une zone d'échange spéciale. En effet, les pages peuvent être chargées directement depuis les fichiers exécutables situés sur les systèmes de fichiers.

Nous nous intéressons, dans ce paragraphe, exclusivement au premier de ces deux paginateurs. Le deuxième sera détaillé dans le paragraphe suivant lorsque nous décrirons l'accès aux fichiers.

La communication entre Mach et un paginateur est effectuée par le biais d'objets mémoire. Lors de la création d'un processus, le serveur de processus appelle le paginateur de données afin de créer les objets mémoire reprśententant les segments de données et de pile. A chacun de ces objets mémoire correspond un ensemble de blocs disque situés dans la zone d'échange. Lorsqu'une page est évincée de la mémoire par Mach, son contenu est écrit dans les blocs correspondants. Quand une page doit être chargée en réponse à un défaut de page, les blocs correspondants sont lus et leur contenu est transmis à Mach.

Nous avons choisi de permettre la création de zones d'échange uniquement sur des partitions séparées. De nombreuses versions d'Unix permettent aujourd'hui d'utiliser des fichiers comme zones d'échange additionnelles en plus des partitions classiques. Nous considérons que cette possibilité, bien que plus souple pour les administrateurs qui peuvent modifier la taille de la mémoire virtuelle sans avoir besoin de repartitionner leurs disques, pose de sérieux problèmes de performances. En effet, alors que l'accès à un bloc contenu dans une partition peut être effectué en une seule opération d'entrées-sorties, l'accès à un bloc contenu dans un fichier nécessite non seulement la lecture du descripteur du fichier (i-nœud) mais également fréquemment la lecture de blocs indirects. De plus, dans le cas de Masix, la mise en œuvre de fichiers d'échange obligerait le paginateur à accéder au serveur de fichiers par un envoi de messages et causerait ainsi une dégradation de performances supplémentaire.

3.5.2 Gestion des zones d'échange par le paginateur de données

Le paginateur de données dispose donc de zones d'échange de taille fixe qu'il peut accéder directement. Il est de sa responsabilité d'organiser la structure de ces zones afin de pouvoir accéder directement aux blocs contenant une page donnée d'un objet mémoire.

L'espace de mémoire virtuelle utilisé par un processus peut varier au cours du temps : le processus peut allouer et désallouer des zones mémoire au fur et à mesure de ses besoins. Il est donc impossible au paginateur de données d'allouer une seule zone de taille fixe à chaque objet mémoire. Le paginateur doit allouer des zones de taille variable à chaque processus.

L'allocation de zones de taille variable peut également poser des problèmes :

- lorsque la taille d'une zone est modifiée, il est nécessaire de procéder à une réallocation avec un déplacement éventuel dans la zone d'échange. Une telle réallocation est très coûteuse ;
- l'allocation de zones de taille variable peut provoquer une fragmentation externe importante obligeant le paginateur à procéder à des compactages périodiques, également très coûteux.

Pour ces raisons, le paginateur de données décompose les zones d'échange en ensembles de blocs de taille fixe. Afin de tenir à jour la liste des ensembles alloués et disponibles, le paginateur gère un tableau contenant un booléen par ensemble : lorsqu'un ensemble est alloué à un objet mémoire, son booléen est mis à 0, lorsqu'un ensemble est désalloué (lors d'une désallocation de mémoire par un processus ou lors d'une terminaison d'objet mémoire), le booléen correspondant est remis à 1. De cette façon, le paginateur peut déterminer facilement les ensembles disponibles et les ensembles alloués.

La deuxième difficulté pour le paginateur consiste à maintenir la liste des ensembles de blocs alloués à chaque objet mémoire. Pour ce faire, il gère pour chaque objet mémoire une table des ensembles qui lui sont alloués. Chaque poste de cette table décrit un ensemble de blocs alloué à l'objet mémoire :

- la zone d'échange concernée ;
- l'adresse du premier bloc de l'ensemble dans la zone.

A partir de cette table, il est aisé pour le paginateur d'accéder à un bloc contenant une page : soit P le numéro d'une page que le paginateur doit lire ou écrire et soit N le nombre de pages regroupées dans chaque ensemble, la page est contenue dans l'ensemble numéro $n = \frac{P}{N}$. Le paginateur accède alors au poste n du tableau de descripteurs d'ensembles alloués à l'objet mémoire et, en ajoutant $P \bmod N$ à l'adresse du premier bloc contenu dans l'ensemble, il obtient l'adresse du bloc contenant la page recherchée.

3.5.3 Pré-allocation

Au niveau de l'allocation d'ensembles dans la zone d'échange, deux stratégies peuvent être utilisées :

1. allouer les ensembles lorsqu'un processus effectue des allocations de mémoire virtuelle (i.e. lors des appels système `brk` et `sbrk` appelés par la fonction de librairie `malloc`). Dans ce cas, on effectue une pré-allocation : l'espace nécessaire dans le "swap" est réservé avant même que le processus ne l'utilise ;
2. allouer les ensembles lorsque le processus référence des pages non encore allouées. On respecte alors le principe d'évaluation tardive en retardant l'allocation jusqu'au moment où elle est absolument nécessaire.

Les deux stratégies possèdent leurs avantages et inconvénients respectifs :

- en pré-allouant la place nécessaire, le système garantit que la mémoire demandée lors des allocations est disponible. En revanche, il se peut que cette mémoire soit mal utilisée : si le processus n'utilise qu'une partie de sa mémoire allouée, la totalité sera néanmoins réservée ;
- en allouant la place mémoire lorsqu'elle est référencée, le système permet d'utiliser la mémoire secondaire de manière optimale : seule la mémoire réellement utilisée par un processus est allouée. En revanche, il ne peut pas garantir que la mémoire qu'il a allouée au processus sera disponible à tout moment.

Les systèmes implémentant la deuxième solution gèrent généralement assez mal le cas où ils ne peuvent pas allouer de mémoire lorsqu'elle est référencée par un processus :

- l'émulation Unix de CMU, au dessus de Mach, provoque une panique système entrainant sa réinitialisation ;
- le système AIX d'IBM envoie un signal `SIGDANGER` aux plus "gros" processus afin de les prévenir du manque de place. Ces processus peuvent choisir, lors de la réception de ce signal, de terminer leur exécution en libérant ainsi l'espace de "swap" qui leur était alloué. Si aucun de ces processus ne se termine, le système provoque leur terminaison brutale un par un jusqu'à ce qu'une place suffisante soit libérée ;

- le système Linux provoque la terminaison brutale de processus un par un jusqu'à ce qu'une place suffisante soit libérée.

Ces trois méthodes comportent l'énorme inconvénient de présenter un comportement indéterministe aux processus qui ne sont pas en mesure de gérer la pénurie de mémoire virtuelle. En effet, La pré-allocation permet à un processus de savoir s'il dispose de la mémoire nécessaire lors de l'allocation (`malloc` retourne `NULL` dans le cas contraire). En revanche, l'allocation “à la demande” peut échouer à tout moment sans laisser au processus la possibilité de gérer l'erreur.

Pour cette raison, nous avons choisi d'utiliser une méthode traditionnelle à base de pré-allocation qui permet aux processus de gérer les pénuries de mémoire virtuelle plutôt que leur présenter un comportement indéterministe. En effet, l'espace disque est désormais si peu coûteux que l'optimisation de sa gestion n'est pas une priorité.

3.6 Accès aux fichiers

3.6.1 Accessibilité des fichiers en mémoire virtuelle

Plutôt que d'adopter un mécanisme traditionnel de gestion de fichiers dans lequel toute entrée-sortie est effectuée par un appel système, nous avons choisi de profiter des facilités de gestion de la mémoire offertes par Mach et d'implémenter des fichiers accessibles dans l'espace d'adressage des processus utilisateur.

Ainsi, lors de l'ouverture d'un fichier, le contenu du fichier est rendu accessible dans l'espace de mémoire virtuelle du processus appelant. Les fonctions de lecture (par exemple `read`) et d'écriture (par exemple `write`) sont ensuite converties en accès mémoire sans aucun appel à un serveur.

Lorsqu'un processus effectue une entrée-sortie sur un fichier, l'opération est transformée de manière transparente en un accès mémoire. Le processus accède donc en fait à un emplacement mémoire où devrait être chargé le fichier. Si la page référencée n'est pas présente en mémoire, un défaut de page est provoqué et Mach utilise un gestionnaire de mémoire pour charger cette page. Les entrées-sorties physiques sur les fichiers sont donc effectuées par un serveur appelé paginateur de fichiers. Ce paginateur, que nous avons évoqué, dans le paragraphe précédent est chargé, lorsqu'une page correspondant à un bloc d'un fichier doit être lue ou écrite, d'interpréter la structure du système de fichiers et de charger ou sauvegarder le bloc. Pour ce faire, à chaque ouverture de fichier, il crée un objet mémoire représentant le fichier.

Les avantages de cette méthode sont les suivants :

- toute entrée-sortie sur fichier est effectuée dans le contexte du processus appelant et ne donne pas lieu à une requête à un serveur comme l'impliquerait un mécanisme traditionnel ;
- les entrées-sorties physiques sont effectuées uniquement lorsque les données des fichiers sont référencées, respectant ainsi la philosophie d'évaluation tardive de Mach ;

- le chargement de page à la demande pour les programmes exécutables peut être implémenté direcement par le paginateur de fichiers ;
- dans la mesure où Mach gère les pages chargées en mémoire et choisit les pages à évincer, il n'y a nul besoin de gérer un ensemble de tampons mémoire cache des disques. En fait, la mémoire physique peut être vue comme un cache de la mémoire secondaire, géré par Mach et "alimenté" par le paginateur de fichiers.

3.6.2 Opérations sur fichiers

Comme nous l'avons exposé dans le paragraphe précédent, les entrées-sorties sur fichiers sont transformées en accès mémoire et gérées par le paginateur de fichiers. Ce mécanisme d'accès qui s'adapte bien aux lectures et aux écritures ne peut toutefois pas être utilisé pour réaliser les opérations de manipulation ponctuelles des fichiers (suppression de fichiers, changement de noms, modification des attributs...). Pour réaliser ces opérations, un mécanisme plus traditionnel doit être utilisé : pour chacune des primitives de manipulation de fichier, un processus client envoie une requête au paginateur de fichiers qui la traite.

Le paginateur de fichiers joue donc un double rôle dans Masix :

- il lit et écrit les blocs composant les fichiers en réponse aux messages de Mach. En ce sens, il se comporte comme un gestionnaire de mémoire ;
- il traite les requêtes de manipulation de fichiers transmises par les processus clients. En ce sens, il se comporte comme tout autre serveur.

3.6.3 Systèmes de fichiers multiples

Afin de permettre une évolution facile des formats des fichiers supportés par Masix, il nous a paru nécessaire de prévoir l'inclusion de plusieurs systèmes de fichiers et la transparence de leur accès par les processus utilisateur. Nous distinguons donc deux niveaux dans la réalisation des appels système sur fichiers :

1. une partie indépendante du format physique des fichiers commune à tous les systèmes de fichiers ;
2. une partie dépendante du format des physique présente pour chaque type de système de fichiers supportés.

Tout appel système est transmis par les processus utilisateur aux routines indépendantes du système de fichiers. Ces routines effectuent les traitements communs puis appellent des fonctions associées à chaque système de fichiers.

Dans un système multi-serveurs comme Masix, il est naturel de définir :

- un serveur regroupant les opérations communes et offrant une interface contenant tous les appels système sur fichiers aux processus :

  ce serveur, appelé "Serveur Virtuel de Fichiers" (SVF), est chargé de recevoir les requêtes transmises par les processus utilisateur et de les traiter en faisant appel aux fonctionnalités offertes par les gestionnaires de systèmes de fichiers physiques ;

- un serveur par type de fichiers physique chargé d'implémenter les entrées-sorties sur les fichiers :

  chacun de ces serveurs, appelés "Serveurs Physiques de Fichiers" (SPF), est chargé d'implémenter un ensemble d'opérations de bas niveau sur les fichiers qu'il gère. Les fonctions offertes par un SPF sont appelées par le SVF lorsque ce dernier a besoin d'accéder au format physique des fichiers.

Afin de permettre une grande dynamicité, les serveurs de fichiers se chargent dynamiquement en mémoire :

- le SVF s'enregistre auprès du SAC lors du démarrage du système afin de transmettre un port sur lequel seront transmises les requêtes sur fichiers ;
- chaque SPF, lors de son initialisation, demande au SAC le port de requête du SVF. Il envoie ensuite sur ce port ses caractéristiques (type de système de fichiers géré ainsi qu'un port de requête) afin que le SVF puisse maintenir la liste des modules lancés.

Nous avons présenté, dans les paragraphes précédents, comment Masix utilise les fonctionnalités de gestion mémoire de Mach afin de rendre le contenu des fichiers accessible dans l'espace d'adressage des processus utilisateur. Nous avons alors parlé d'un paginateur de fichiers chargé de répondre aux requêtes de chargement ou de sauvegarde de pages transmises par Mach. Afin de supporter plusieurs types de systèmes de fichiers, deux stratégies étaient utilisables :

1. définir un seul paginateur, le SVF, qui répond aux requêtes du micro-noyau en transmettant des ordres de lecture ou d'écriture au SPF concerné :

   Cette méthode a l'avantage de ne définir qu'un seul paginateur mais ne permet pas d'obtenir des performances optimales. En effet, lors d'un défaut de page, Mach transmet une requête au SVF qui effectue lui-même un appel de procédure à distance dans un SPF. On a donc deux tâches impliquées dans le chargement ou la sauvegarde de la page et les délais dus aux communications et aux changements de contexte sont plus importants ;

2. définir chaque SPF comme un paginateur : lors de l'ouverture d'un fichier, le SPF correspondant crée un objet mémoire et toutes les requêtes d'entrées-sorties sont ensuite transmises par Mach à ce SPF sans intervention du SVF.

   Cette méthode permet d'éviter une indirection et donc d'économiser des échanges de messages et des changements de contexte à chaque lecture ou écriture de page. En revanche, elle nécessite que chaque SPF implémente un dialogue avec Mach.

Nous avons choisi la deuxième solution car il nous parait très important de minimiser les délais d'entrées-sorties. Chaque SPF est donc un paginateur de fichiers et Mach communique directement avec lui pour les requêtes de chargement et de sauvegarde de pages.

3.6.4 "Montage" d'arborescences

Masix permet d'accéder de rendre accessible un système de fichiers par une opération de montage. Cette opération permet de connecter l'arbre des fichiers contenu dans le système de fichiers sous un répertoire appelé point de montage. Chaque système de fichiers monté est géré par un SPF qui assure les opérations d'entrées-sorties sur les fichiers qu'il contient.

Le SVF doit toutefois intervenir dans les résolutions de noms : lorsqu'un processus effectue une opération sur un fichier, il doit citer le chemin d'accès à ce fichier. Ce chemin d'accès peut être soit absolu, i.e. contenir tous les répertoires à explorer depuis la racine de l'arborescence, soit relatif, i.e. contenir les répertoires à explorer depuis le répertoire courant associé au processus. À partir de ce chemin d'accès, le SVF doit accéder au fichier lui-même, c'est à dire obtenir un port, correspondant au fichier, géré par un SPF.

Dans Unix, la fonction `namei` est chargée de cette analyse de chemin d'accès. Cette fonction analyse le chemin répertoire par répertoire pour obtenir l'i-nœud correspondant au fichier cité.

Dans Masix, il n'est pas possible de faire effectuer cette tâche par une fonction contenue dans le SVF. En effet, cette analyse nécessite la lecture des répertoires et donc la connaissance de leur structure. Or, le SVF est indépendant de la structure physique des répertoires et seul un SPF est capable de lire et d'analyser le contenu des répertoires.

La conversion de chemin d'accès en descripteur de fichier (port) doit donc être effectuée différemment. Deux méthodes peuvent être envisagées :

1. faire effectuer cette conversion par le SVF qui appelle un SPF pour la lecture de chaque répertoire cité dans le chemin. Avec cette stratégie, le SVF maintient la liste des systèmes de fichiers montés et fait valider par le SPF concerné chaque composante du chemin d'accès ;

2. faire valider par chaque SPF concerné une partie du chemin d'accès. Avec cette stratégie, chaque SPF essaie de valider la plus grande partie du chemin d'accès fourni et transmet le reste du chemin au SPF suivant lorsqu'il rencontre un point de montage. Il est donc impératif que chaque SPF connaisse les systèmes de fichiers montés sur des répertoires qu'il gère.

Considérons ces deux méthodes sur un exemple en supposant que l'arborescence des fichiers est formée de trois systèmes de fichiers montés :

1. le système de fichiers racine (`/`) géré par un serveur SPF1 ;
2. un système de fichiers, monté sur le répertoire `/usr/src`, géré par un serveur SPF2 ;
3. un système de fichiers, monté sur le répertoire `/usr/src/masix`, géré par un serveur SPF3.

Lors de l'interprétation d'un nom de fichier, le système doit parcourir tous les répartoires cités dans le chemin afin de les valider. Il n'est pas possible de commencer l'analyse à un autre répertoire que la racine de l'arborescence. En effet, il est indispensable de vérifier les droits d'accès du processus appelant sur toutes les composantes du chemin d'accès.

Etudions le traitement nécessaire si on souhaite accéder au fichier `/usr/src/masix/fs/ext2/dcache.c` avec chacune de ces deux méthodes :

1. avec la première méthode :
   - le SVF appelle SPF1 pour valider `usr`, SPF1 lui retourne un port caractérisant cet répertoire ;
   - le SVF appelle SPF1 pour valider `src` par rapport au répertoire précédent, SPF1 lui retourne un port caractérisant ce répertoire ;
   - le SVF, en consultant sa table des systèmes de fichiers montés, détecte qu'un système de fichiers est monté sur `/usr/src`. Il appelle donc SPF2 pour valider `masix` sur ce système de fichier, SPF2 lui retourne un port caractérisant ce répertoire ;
   - le SVF, en consultant sa table des systèmes de fichiers montés, détecte qu'un système de fichiers est monté sur `/usr/src/masix`. Il appelle donc SPF3 pour valider `fs` sur ce système de fichiers, SPF3 lui retourne un port caractérisant ce répertoire ;
   - le SVF appelle SPF3 pour valider `ext2` par rapport au répertoire précédent, SPF3 lui retourne un port caractérisant ce répertoire ;
   - enfin, le SVF appelle SPF3 pour valider `dcache.c` par rapport au répertoire précédent, SPF3 lui retourne un port caractérisant le fichier ;
2. avec la deuxième méthode :
   - le SVF appelle SPF1 pour valider `/usr/src/masix/fs/ext2/dcache.c`. SPF1 valide `/usr` puis `/usr/src` ;
   - SPF1 détecte qu'un système de fichiers est monté sur `/usr/src` et appelle SPF2 pour valider `/masix/fs/ext2/dcache.c` ;
   - SPF2 valide `/masix` ;
   - SPF2 détecte qu'un système de fichiers est monté sur `/masix` et appelle SPF3 pour valider `/fs/ext2/dcache.c` ;
   - SPF3 valide `/fs`, puis `/fs/ext2` et enfin `/fs/ext2/dcache.c`. Il retourne le port caractérisant le fichier à SPF2.

- SPF2 retourne ce port à SPF1 ;
- SPF1 retourne ce port au SVF qui possède ainsi un descripteur du fichier.

Sur cet exemple, il apparait évident que la deuxième méthode est beaucoup plus efficace que la première. En effet, alors que la première nécessite un appel de procédures à distance entre serveurs à chaque répertoire traversé (6 dans l'exemple), la seconde permet de n'effectuer un appel de procédure de procédure à distance qu'à chaque système de fichiers traversé (3 dans l'exemple). En contrepartie de cette efficacité, chaque SPF doit gérer les systèmes de fichiers montés sur des répertoires qu'il gère alors que la première méthode permettait de limiter la gestion des systèmes de fichiers montés au seul SPF.

Afin d'obtenir les meilleures performances possibles, nous avons choisi d'utiliser la seconde méthode dans Masix.

Il est à noter que les méthodes de parcours que nous avons exposées considèrent le cas d'une arborescence pure. Une arborescence de type Unix peut comporter des liens symboliques créant un graphe et nécessitant un traitement particulier lorsque un lien pointe sur un fichier situé sur un autre système de fichiers. Dans ce dernier cas, en effet, l'analyse doit être recommencée après remplacement du nom du lien par sa valeur.

3.7 Gestion des périphériques

Le micro noyau Mach contient des gestionnaires de périphériques capables d'assurer les entrées-sorties mais n'implémentant pas d'opérations de haut niveau. L'environnement Unix contenu dans Masix offre donc des pilotes de périphériques plus évolués quit utilisent les gestionnaires contenus dans Mach pour effectuer les entrées-sorties physiques.

De la même façon qu'Unix, Masix supporte deux types de pilotes :

- les pilotes de périphériques réels gérant des périphériques physiques ;
- les pilotes de périphériques virtuels ne correspondant pas à des dispositifs matériels.

Le rôle d'un pilote de périphérique de l'environnement Unix de Masix consiste à implémenter les opérations de haut niveau sur les périphériques (opérations liées généralement à l'emploi de l'appel système `ioctl`) ainsi que les entrées-sorties sur les dispositifs matériels sous-jacents.

De même que dans Unix, tout périphérique est caractérisé par deux identificateurs numériques :

1. un indice majeur indiquant le type du périphérique ;
2. un indice mineur.

À tout indice majeur est associé un pilote de périphérique. Ce pilote, dans Masix, est un serveur qui traite des requêtes transmises par des clients. Il s'enregistre auprès du SAC au début de son exécution en spécifiant l'indice majeur des périphériques qu'il gère.

Contrairement aux fichiers "ordinaires", les périphériques ne sont pas gérés par des paginateurs. En effet, alors qu'il est relativement aisé de gérer le contenu d'un fichier comme un segment mémoire, les périphériques se prêtent assez mal à ce traitement pour plusieurs raisons :

- certains périphériques ne sont accessibles que séquentiellement (par exemple, un clavier, une imprimante, une liaison série...) et il serait aberrant de permettre un accès direct en mémoire de leur "contenu" ;
- la taille d'un fichier est forcément soumise à une limite relative à l'implémentation de son format. Il est donc aisé de réserver un sous-ensemble de l'espace d'adressage d'un processus pour contenir le contenu maximal d'un fichier[2]. Par contraste, certains périphériques n'ont pas de taille limite[3] et il est impossible de réserver une place mémoire de taille fixe pour leur "contenu" ;
- dans le cas où la taille d'un périphérique est fixée (par exemple, dans le cas d'un disque), cette taille peut être très importante, approchant ou dépassant la taille de l'espace d'adressage d'un processus. Il peut donc être impossible de réserver un espace suffisant en mémoire virtuelle pour le "contenu" du périphérique.

Pour ces raisons, l'accès à un périphérique sous Masix est effectué via la transmission de requêtes au serveur qui le gère. Un processus utilisateur qui souhaite accéder à un périphérique, réel ou virtuel, ouvre le fichier spécial correspondant au périphérique. Cette ouverture est effectuée par un appel au SVF qui transmet la requête au serveur de périphérique correspondant. Ce dernier crée un port utilisé pour les requêtes ultérieures et retourne un droit d'émission sur ce port au SVF qui retourne lui-même ce port au processus appelant. Ce dernier transmet ensuite ses requêtes d'entrées-sorties sur ce port et communique ainsi directement avec le gestionnaire de périphérique. Le SVF n'intervient donc que lors de l'ouverture du périphérique mais n'est plus concerné par les lectures et écritures ultérieures. Ceci permet d'éviter une indirection lors de chaque opération d'entrée-sortie.

Il est à noter que certains serveurs de Masix, comme les paginateurs, n'utilisent pas ces pilotes mais dialoguent directement avec les gestionnaires de périphériques contenus dans Mach. En effet, les serveurs concernés n'ont pas besoin de fonctionnalités de haut niveau fournies par les pilotes Masix mais uniquement des primitives d'entrées-sorties de bas niveau offertes par Mach. De plus, en appelant directement les fonctions de Mach, les paginateurs évitent une indirection qui dégraderait leurs performances.

[2] quitte à fixer cette taille limite
[3] par exemple, comment savoir le nombre maximal de caractères qui seront tapés sur un clavier ?

4 Conclusion

Nous avons présenté, dans ce chapitre, la structure du système local de Masix formé de la coopération de plusieurs serveurs s'exécutant en mode utilisateur.

Le système d'accueil est implémenté par le Serveur d'Administration qui gère tous les autres serveurs et permet les communications entre clients et serveur en assurant un service de nommage. Il joue également un rôle de surveillance du système en maintenant la liste des serveurs en cours d'utilisation et en intervenant si une partie essentielle du système arrête de fonctionner.

Les processus et les signaux sont gérés par un serveur qui utilise les tâches et *threads* fournis par Mach pour implémenter les processus selon la sémantique Unix. Ce serveur gère également l'envoi et le traitement des signaux en conservant la liste des serveurs impliqués dans le traitement d'un appel système.

La mémoire est gérée par un paginateur qui est chargé de répondre aux requêtes de Mach concernant les pages à charger et à sauvegarder. Le paginateur utilise des zones d'échange sur disque qu'il structure en ensembles de pages afin de tenir à jour la mémoire virtuelle allouée.

Les fichiers sont rendus accessibles par un paginateur qui se charge de charger les blocs nécessaires en mémoire lorsqu'un processus effectue une entrée-sortie physique. Plusieurs paginateurs peuvent coexister dans le système afin de gérer des systèmes de fichiers de formats différents.

Enfin, chaque type de périphérique est géré par un serveur chargé d'implémenter les requêtes d'entrées-sorties selon un protocole prédéfini.

Chapitre 7

Réalisation

1 Introduction

Le premier problème, lors de l'implémentation d'un nouveau système d'exploitation, consiste à réaliser la première version. En effet, tant que le système n'est pas capable d'offrir les services nécessaires à son propre développement, il est nécessaire de travailler sous un autre système d'exploitation et d'effectuer un développement croisé.

Afin de minimiser les probèmes de développement croisé, il est préférable d'utiliser une machine de développement similaire à la machine cible. En effet, l'utilisation de machines différentes entraine de nombreux problèmes supplémentaires liés à la représentation interne différente des données.

Nous avons donc travaillé avec le système d'exploitation Linux pour développer le système local de base de Masix. Linux, qui fonctionne sur PC 386, fournit de nombreuses fonctionnalités Unix.

Nous présentons tout d'abord le système Linux en décrivant ses caractéristiques et en justifant son choix comme système de développement.

Nous exposons ensuite la méthode que nous avons utilisé pour pouvoir initialiser le contenu des systèmes de fichiers de Masix sous Linux. Linux, dans sa version standard, gêre un système de fichiers comprenant d'importantes limitations. Nous avons travaillé à redéfinir la structure interne du système afin de lui permettre d'utiliser en parallèle plusieurs systèmes de fichiers. Nous avons également implémenté deux systèmes de fichiers moins restrictifs.

Nous détaillons enfin la structure du système de fichiers standard de Linux et des deux systèmes de fichiers que nous avons réalisé. Ces deux systèmes de fichiers ont été inclus dans la distribution standard de Linux et le deuxième est maintenant devenu le système de fichiers par défaut.

2 Le système Linux

Afin de réaliser la première version de Masix, nous avons utilisé le système Linux [Klaus 1993] disponible sur micro-ordinateurs PC 386. Ce système, développé par Linus Torvalds, offre un environnement utilisateur complet fournissant un grand nombre de fonctionnalités Unix.

Linux est conçu sous forme d'un noyau monolithique spécialement adapté au processeur i386, ce qui lui permet d'offrir des performances remarquables au prix d'une grande importabilité. Un projet de portage de Linux sur processeur 68000 est néanmoins en cours.

Les principales fonctionnalités offertes par Linux sont les suivantes :

- processus selon la sémantique Unix et le *job control* Posix ;
- mémoire virtuelle (chargement de page à la demande, partage de page avec copie en écriture...) ;
- cache disque dynamique ;
- système de fichiers virtuel permettant l'ajout et la coexistence de plusieurs types de systèmes de fichiers ;
- terminaux virtuels ;
- support des protocoles TCP, UDP, IP de communication en réseau.

Ces fonctionnalités et la compatibilité Unix offerte par Linux permettent de porter très facilement de nombreux logiciels et d'obtenir ainsi un système complet à peu de frais. On peut noter parmi les logiciels importants supportés par Linux :

- l'ensemble des produits GNU, incluant l'éditeur de texte emacs, les compilateurs C et C++ gcc et g++ ;
- le système de fenêtrage X-Window ;
- les formateurs de texte TEXet LATEX.

Linux constitue un système d'exploitation en pleine expansion. De nombreux programmeurs, de part le monde, implémentent de nouvelles fonctionnalités qui sont fréquemment incluses dans les versions "officielles" du système.

Notre choix s'est porté sur ce système pour plusieurs raisons :

- Linux s'exécute sur des micro-ordinateurs PC 386, machines cibles de la première version de Masix. Un développement croisé est plus facile à effectuer en utilisant une machine hôte identique à la machine cible. En effet, en utilisant la même plate-forme matérielle, il est possible de s'affranchir des problèmes liés aux différentes représentations internes des données ;

- la majorité des programmes de Linux sont des produits GNU que nous comptons réutiliser afin de fournir un environnement utilisateur complet sous Masix : seule la conception du “noyau” de Masix nous intéresse et nous comptons ré-utiliser bon nombre de composants logiciels existants pour les autres parties du système (interpréteurs de commandes, programmes utilitaires, commandes utilisateur...) ;
- Linux est disponible et utilisable librement ;
- il est beaucoup plus performant que nombre de systèmes commerciaux[1].

3 Communications entre Linux et Masix

Le développement de Masix sous Linux pose le problème du partage des donnés entre les deux systèmes. En effet, lors de la génération du système, il est nécessaire de pouvoir accéder aux systèmes de fichiers de Masix depuis Linux afin de les initialiser avec les fichiers nécessaires (programmes exécutables et fichiers de configuration).

3.1 Ajout de système de fichiers dans Linux

Initialement, Linux ayant été développé comme un remplacement du système Minix, il ne supportait que le système de fichiers de ce dernier. Ce système de fichiers très simple n'offrait que peu d'intérêt et de nombreuses limitations (noms de fichiers de 14 caractères maximum, partitions de 64 méga octets maximum).

Nous avons utilisé les possibilités du système de fichiers virtuel de Linux pour y inclure un système de fichiers plus élaboré destiné à être supporté par Masix. L'inclusion de ce système de fichiers dans Linux permet un accès transparent au format des fichiers Masix et résoud ainsi le problème d'initialisation des fichiers nécessaires à Masix.

Nous avons implémenté deux versions de notre système de fichiers appelées :

- *Extended File System* ;
- *Second Extended File System*.

Ces deux systèmes de fichiers ont été diffusées auprès des utilisateurs de Linux tout d'abord sous forme d'ajouts au noyau standard puis inclus dans la distribution officielle de Linux. Ils sont largement utilisés en raison des extensions qu'ils apportent par rapport au système de fichiers d'origine et le second système de fichiers “étendu” constitue même le système de fichiers conseillé par défaut dans plusieurs distributions.

[1] par exemple, la compilation de cette thèse par la même version de LaTeXest effectuée plus rapidement par un PC 486 à 33 Mhz sous Linux que par une station de travail Sun IPX sous SunOS 4.1.3. Le matériel de la station Sun étant largement plus performant que celui du PC, seules les performances du système d'exploitation permettent d'expliquer cette différence.

L'implémentation de systèmes de fichiers dans Linux nous a permis :

- d'implémenter et de tester dans un système existant une partie vitale du système Masix ;
- de procéder à des tests réels du système de fichiers en profitant des milliers d'utilisateurs de Linux dans le monde entier et de profiter de leurs remarques, suggestions, critiques et rapports d'anomalies.

3.2 Structure du système de fichiers virtuel de Linux

3.2.1 Principe

Avant l'implémentation de nouveaux systèmes de fichiers que nous avons réalisée, Linux ne supportait qu'un seul système de fichiers, conmpatible avec celui de Minix. La figure 7.1 représente la structure des modules de gestion de fichiers avant notre travail.

Nous avons travaillé sur l'ajout d'une couche d'indirection utilisée lors de la réalisation des appels système sur fichiers. Cette couche, appelée "système de fichiers virtuel" (SFV), est chargée de réaliser tous les appels indépendants du format physique des fichiers et d'appeler les fonctions nécessaires lors d'un accès à la structure physique des fichiers. Ce mécanisme de couche d'indirection est fréquemment utilisée dans les systèmes compatibles Unix afin de permettre l'intégration et la coexistence de plusieurs types de systèmes de fichiers [Kleiman 1986, Seltzer *et al.* 1993].

La figure 7.2 représente la structure des modules de gestion de fichiers sous Linux où plusieurs systèmes de fichiers différents (sur la figure, les systèmes de fichiers compatibles Minix et MS/DOS ainsi que les deux versions du système de fichiers de Masix) peuvent coexister et être utilisés de manière transparente. Les appels système concernant les fichiers sont dirigés vers le SFV qui les transmet au système de fichiers physique concerné. Ce module implémente les opérations physiques sur les fichiers en utilisant les fonctions d'entrées-sorties fournis par la couche de gestion des tampons mémoire (*Buffer Cache*). Cette dernière couche utilise les sous-programmes de lecture et d'écriture de blocs offerts par le pilote du périphérique sur lequel se trouve le système de fichiers.

Linux constitue un noyau monolithique dans lequel toutes les fonctions de services du système sont rassemblées. Les différents modules partagent le même espace d'adressage. Le SFV peut donc accéder directement aux données et aux sous-programmes des autres modules de gestion de fichiers.

Afin de permettre la coexistence et l'appel tranparent à plusieurs système de fichiers différents, le SFV définit une interface que tout système de fichiers doit respecter pour être inclus dans Linux. Cette interface, que nous avons contribué à définir lors de l'inclusion de notre premier système de fichiers[2], est composée d'un ensemble d'opérations qu'un système de fichiers doit offrir.

[2] Avant cette inclusion, Linux ne supportait que le système de fichiers Minix.

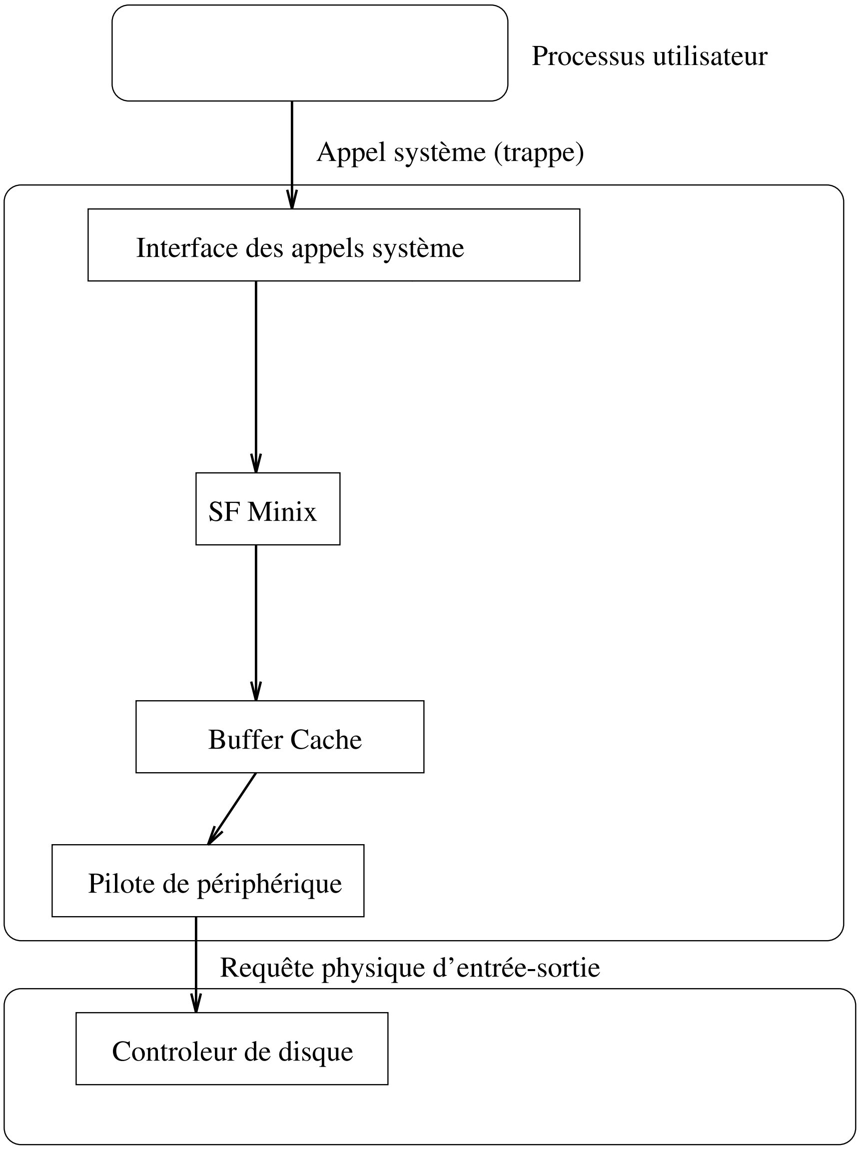

Figure 7.1: Structure des couches de gestion de fichiers dans Linux avant modification

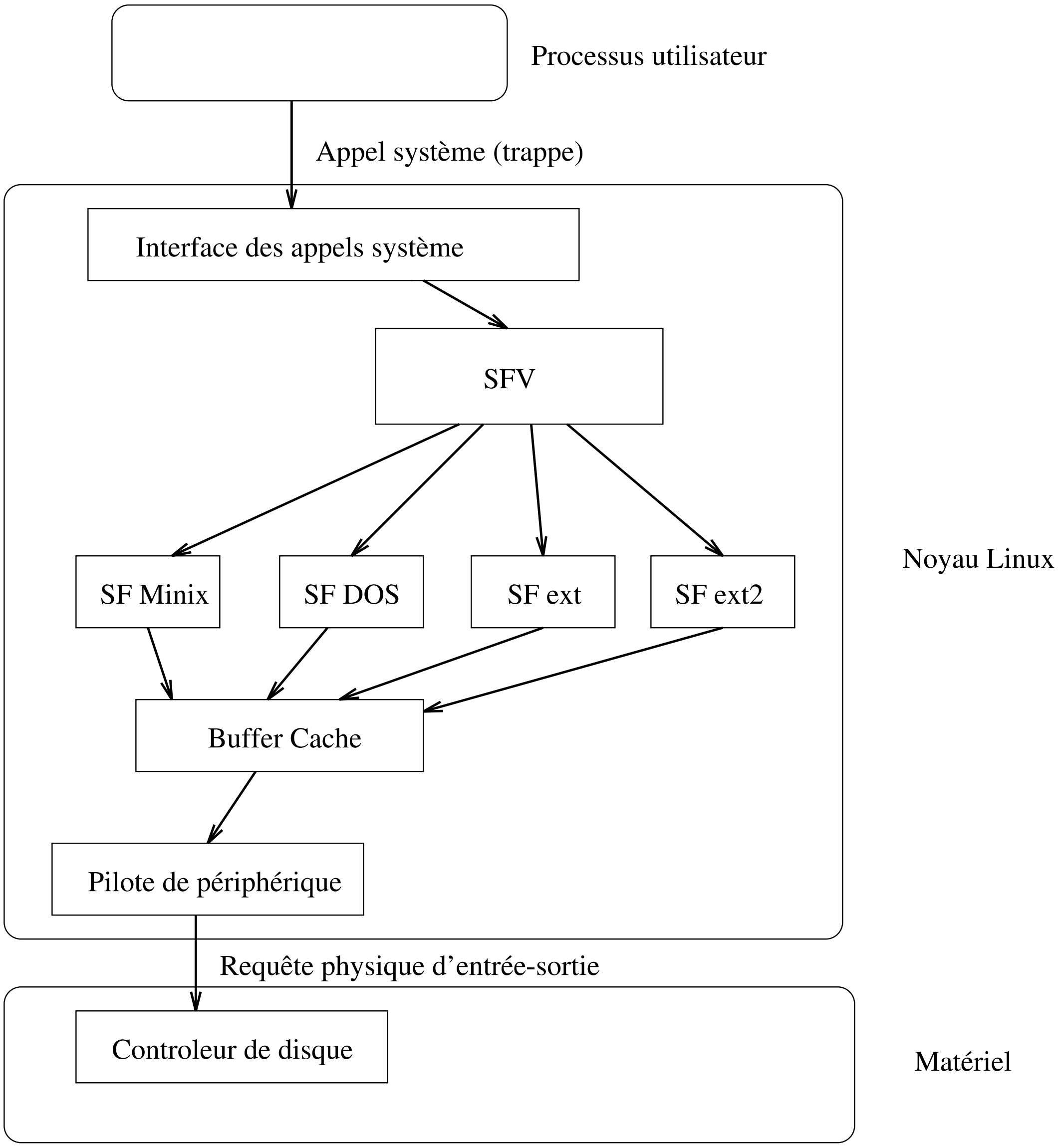

Figure 7.2: Structure des couches de gestion de fichiers dans Linux après modification

Ces opérations peuvent être associées à trois types d'objets :

- le système de fichiers lui-même ;
- un i-nœud, i.e. un descripteur de fichier sur disque ;
- un descripteur de fichier ouvert.

3.2.2 Description d'un système de fichiers

Le SFV a juste besoin de connaître les systèmes de fichiers supportés par le noyau. Pour cela, une table définit, pour chaque système de fichiers inclus dans Linux :

- une chaîne de caractêres définissant son type ;
- un pointeur sur la fonction appelée lorsqu'un système de fichiers de ce type est monté. Cette fonction est chargée d'initialiser les données en mémoire correspondant au système de fichiers et d'indiquer au SFV l'ensemble des opérations disponibles ;
- un booléen indiquant si le système de fichiers nécessite une partition disque ou non.

La fonction appelée lors du montage du système dans l'arborescence des répertoires est chargée de lire la description du système de fichiers physique depuis le disque, d'initialiser ses données internes et de renvoyer au SFV un descripteur de système de fichiers monté.

Un descripteur de système de fichiers contient :

- des informations communes à tous les systèmes de fichiers physiques :
  - le périphérique correspondant au système de fichiers ;
  - la taille des blocs logiques supportés par le système de fichiers ;
  - les paramètres de montage ;
- la liste des opérations définies sur le système de fichiers. Ces opérations sont rendus accessibles au SFV en fournissant à ce dernier des pointeurs sur les fonctions suivants :
  - `read_inode` : lecture en mémoire d'un i-nœud ;
  - `notify_change` : mise à jour du contenu d'un i-nœud ;
  - `write_inode` : ré-écriture sur disque du contenu d'un i-nœud ;
  - `put_inode` : désallocation d'un descripteur de fichier en mémoire ;
  - `put_super` : désallocation du descripteur de système de fichiers, appelée lorsqu'il est démonté ;
  - `write_super` : ré-écriture sur disque des copies mémoire des données ;

  - `statfs` : calcul des statistiques d'utilisation du système de fichiers ;

- les données spécifiques au système de fichiers qu'il est nécessaire de maintenir en mémoire pendant qu'il est monté.

Une fois qu'un système de fichiers est monté, le SFV connait ses opérations de base grâce au descripteur renvoyé par la fonction d'initialisation. Il dialogue ensuite avec le module de gestion du système de fichiers physique en utilisant deux types de descripteurs :

- les i-nœuds (*inodes*) représentant les fichiers accédés par des opérations ponctuelles ne nécessitant pas obligatoirement une ouverture préalable ;
- les fichiers (*files*) représentant les fichiers ayant été ouverts par un appel à la fonction `open`. À chaque fichier ouvert, correspond un i-nœud.

3.2.3 Descripteur d'i-nœuds

Le descripteur d'un i-nœud contient les informations suivantes :

- des informations communes à tous les systèmes de fichiers :
  - le périphérique sur lequel est stocké le fichier ;
  - le numéro de fichier dans le système de fichiers ;
  - le type et les droits d'accès associés au fichier ;
  - le nombre de noms associés au fichier ;
  - l'identificateur de l'utilisateur propriétaire du fichier ;
  - la taille du fichier en octets ;
  - les dates de création, de modification et de dernier accès au fichier ;
  - la taille et le nombre des blocs logiques constituant le fichier ;
- des informations utilisés pour la gestion de l'i-nœud en mémoire :
  - indicateur de modification ;
  - pointeurs sur i-nœuds précédent et suivant ;
  - indicateur de verrouillage en mémoire ;
- la description des opérations possibles sur le fichier. Chaque opération est rendue accessible par un pointeur sur la fonction qui l'implémente dans le module de gestion du système de fichiers physique. Ces opérations sont les suivantes :
  - `create` : création d'un fichier ;
  - `lookup` : recherche du nom d'un fichier dans un répertoire ;

  - `link` : création d'un lien sur un fichier ;
  - `unlink` : suppression d'un lien sur un fichier avec suppression du fichier lui-même lorsque le dernier lien est détruit ;
  - `symlink` : création d'un lien symbolique ;
  - `mkdir` : création d'un répertoire ;
  - `rmdir` : suppression d'un répertoire ;
  - `mknod` : création d'un fichier spécial ;
  - `rename` : changement du nom d'un fichier ;
  - `read_link` : lecture du nom d'un fichier pointé par un lien symbolique ;
  - `follow_link` : renvoi de l'i-nœud d'un fichier pointé par un lien symbolique ;
  - `bmap` : renvoi du numéro de bloc physique correspondant à un bloc logique contenu dans un fichier ;
  - `truncate` : modification de la taille d'un fichier avec désallocation de blocs si nécessaire ;
  - `permission` : vérification de droits d'accès sur un fichier ;
- les informations liées au système de fichiers physique.

3.2.4 Descripteur de fichier ouvert

Un descripteur de fichiers ouvert contient les informations suivantes :

- des informations communes à tous les systèmes de fichiers :
  - le mode d'ouverture ;
  - la position courante dans le fichier ;
  - un pointeur sur le descripteur d'i-nœud associé ;
- des pointeurs sur fonctions repésentant les opérations disponibles sur le fichier :
  - `lseek` : positionnement dans un fichier ;
  - `read` : lecture depuis un fichier ;
  - `write` : écriture dans un fichier ;
  - `readdir` : lecture de répertoire ;
  - `select` : attente de données à lire dans un fichier ;
  - `ioctl` : opération de contrôle ;
  - `mmap` : initialisation du fichier pour rendre son contenu accessible en mémoire virtuelle ;
  - `open` : ouverture d'un fichier ;
  - `release` : fonction appelée lorsque le fichier est fermé par tous les processus l'ayant précédemment ouvert ;
  - `fsync` : ré-écriture de tous les blocs modifiés du fichier sur disque.

4 Analyse des systèmes de fichiers

4.1 Représentation des fichiers sur disques

Les trois systèmes de fichiers que nous allons présenter utilisent des méthodes communes de stockage des fichiers. Nous détaillons ici ces points communs avant d'exposer les particularités de chacun.

Les répertoires dans les trois systèmes de fichiers respectent une structure arborescente. Un répertoire est un catalogue de noms de fichiers et de répertoires. Un fichier est caractérisé par un nom unique constitué des répertoires à parcourir depuis la racine de l'arborescence pour accéder au fichier.

Afin de maintenir les informations liées aux fichiers, les trois systèmes de fichiers utilisent un descripteur de fichier. Ce descripteur, appelé i-nœud, contient les informations caractérisant le fichier. Les trois systèmes de fichiers permettent d'allouer des blocs non contigus aux fichiers. Chaque i-nœud contient des pointeurs sur les blocs alloués aux fichiers. Comme dans Unix [Bach 1986], ces pointeurs peuvent être :

- des pointeurs directs contenant le numéro d'un bloc de données ;
- un pointeur indirect contenant le numéro d'un bloc contenant lui-même les numéros des blocs de données ;
- un pointeur doublement indirect contenant le numéro d'un bloc contenant lui-même des numéros de blocs qui contiennent les numéros des blocs de données ;
- un pointeur triplement indirect (seulement dans le cas des deux systèmes de fichiers "étendus") contenant le numéro d'un blocs qui contient des numéros de blocs contenant eux-mêmes des numéros de blocs qui contiennent les numéros des blocs de données.

La figure 7.3 schématise cette indexation des blocs alloués en se limitant à deux niveaux d'indirection.

4.2 Le système de fichiers Minix

Le premier système de fichier implémenté dans Linux[3] est compatible avec celui du système Minix [Tanenbaum 1987] afin de permettre un partage des données entre les deux systèmes[4].

[3] Le seul implémenté aux débuts du système.
[4] Linux a tout d'abord été développé sous Minix.

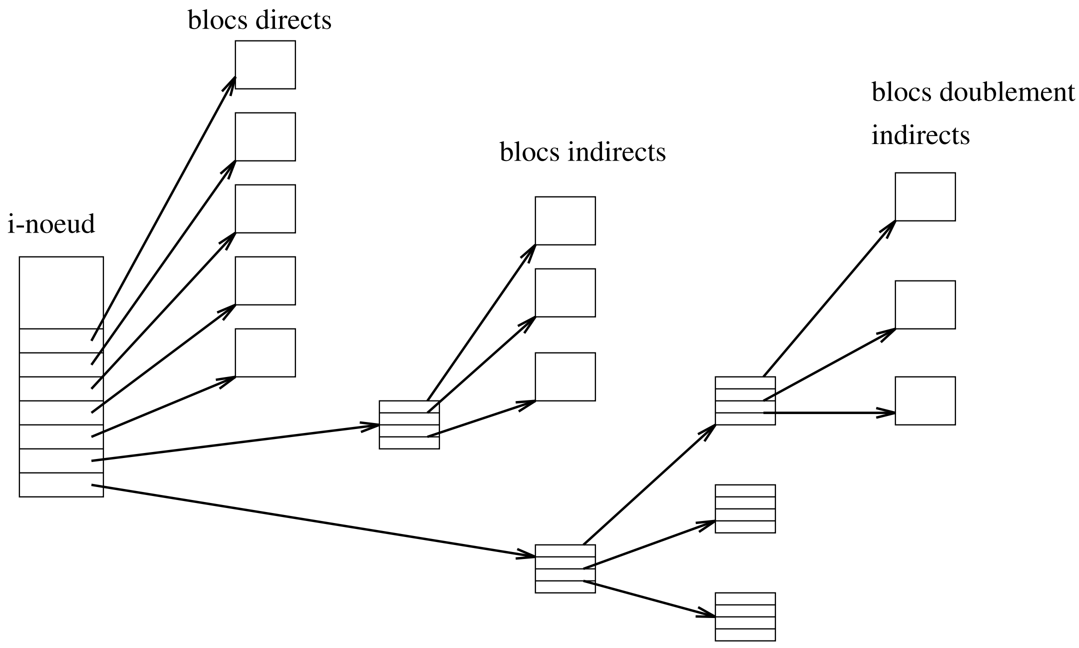

Figure 7.3: Indexation des blocs dans les systèmes de fichiers

Ce système de fichiers est relativement simple et possède deux restrictions importantes :

- les numéros de blocs sont codés sur deux octets ce qui limite la taille d'un système de fichiers à 65535 blocs, soit 64 méga-octets ;
- les entrées de répertoire sont de taille fixe et permettent de conserver des noms de fichiers de 14 caractères maximum.

Un système de fichiers Minix est composé :

- d'un bloc de boot contenant le code lancé lors du démarrage du système ;
- d'un super bloc contenant les caractéristiques physiques du système de fichiers :
  - nombre de blocs ;
  - nombre d'i-nœuds ;
  - nombre de blocs de bitmap ;
  - taille des blocs logiques ;
  - taille maximale d'un fichier ;
  - signature ;
- d'un vecteur de bits indiquant les i-nœuds allouées et les i-nœuds disponibles ;

- d'un vecteur de bits indiquant les blocs alloués et les blocs disponibles ;
- d'une table des i-nœuds contenant les renseignements sur les fichiers existants. Chaque i-nœud caractérisant un fichier contient :
  - le type et les droits d'accès associés au fichier ;
  - la taille du fichier en octets ;
  - la date de dernière modification ;
  - l'identificateur de l'utilisateur propriétaire du fichier ;
  - le nombre de noms (i.e. liens) associés au fichier ;
  - 9 adresses des blocs alloués au fichier (7 adresses de blocs directs, l'adresse d'un bloc indirect et l'adresse d'un bloc doublement indirect) ;
- des blocs de données utilisés pour stocker le contenu des fichiers et des répertoires.

La structure de ce système de fichiers sur disque est représentée sur la figure 7.4.

Secteur de boot	Super Block	Bitmap i-nœuds	Bitmap blocks	Table des i-nœuds	Blocs de données

Figure 7.4: Structure physique du système de fichiers Minix

Lors d'une allocation de bloc ou d'i-nœud, le vecteur de bits correspondant est utilisé : le système de fichiers recherche le premier bit à zéro, le positionne à 1 et alloue le bloc correspondant. Lors d'une désallocation, le bit correspondant dans le vecteur est mis à 0.

Les répertoires dans le système de fichiers Minix sont gérés comme des tableaux de descripteurs de taille fixe. Chaque entrée de répertoire contient :

- un numéro d'i-nœud codé sur deux octets, ce qui limite le nombre d'i-nœuds utilisables à 65535 dans un même système de fichiers ;
- un nom de fichier codé sur 14 caractères.

La structure d'un répertoire est représentée sur la figure 7.5.

4.3 Le premier système de fichiers "étendu"

Nous avons développé le système de fichiers "étendu" afin de supprimer les deux limitations les plus génantes du système de fichiers Minix (noms de fichiers sur 14 caractères, 64 méga-octets

numéro d'i-nœud 1	nom de fichier 1
numéro d'i-nœud 2	nom de fichier 2
..	...
numéro d'i-nœud N	nom de fichier N

Figure 7.5: structure d'un répertoire dans le système de fichiers Minix

de taille maximale pour les systèmes de fichiers). Ce développement a été nécessaire lorsque nous avons voulu compiler le micro-noyau Mach. En effet, les programmes source de Mach ont fréquemment des noms assez longs qu'il n'est pas possible de stocker dans les répertoires du système de fichiers Minix.

Le premier système de fichiers "étendu" utilise de nombreuses stratégies du système de fichiers Minix dont il est issu. Il inclut de plus de nombreuses améliorations que nous détaillons dans les sections suivantes.

4.3.1 Gestion des blocs libres

La première innovation fondamentale est la façon dont il gère les blocs et i-nœuds libres. Alors que le système de fichiers Minix gère un vecteur de bits pour maintenir la liste des blocs et des i-nœuds disponibles, nous avons choisi d'utiliser des listes chaînées : le super bloc contient le numéro du premier bloc disponible qui contient lui-même les numéros de 254 autres blocs libres et le numéro du bloc suivant dans la liste. De la même façon, un i-nœud disponible contient les numéros de 14 autres i-nœuds non alloués et le numéro de l'i-nœud suivant dans la liste.

La figure 7.6 représente la liste chaînée de blocs libres.

Lors d'une allocation de bloc (resp. i-nœud), le premier bloc (resp. i-nœud) de la liste est utilisé. Lorsqu'un bloc (resp. i-nœud) est désalloué, il est ajouté en tête de liste. Cette gestion de liste chaînée permet d'économiser la place utilisée par les vecteurs de bits dans le système de fichiers Minix (1 kilo-octets par 8192 blocs ou i-nœuds).

4.3.2 Structure des répertoires

La deuxième innovation importante du système de fichiers concerne la structure des répertoires : dans ce système de fichiers, les entrées de répertoire sont de taille variable afin de permettre des noms de fichiers de taille plus importante en utilisant l'espace disque de manière optimale.

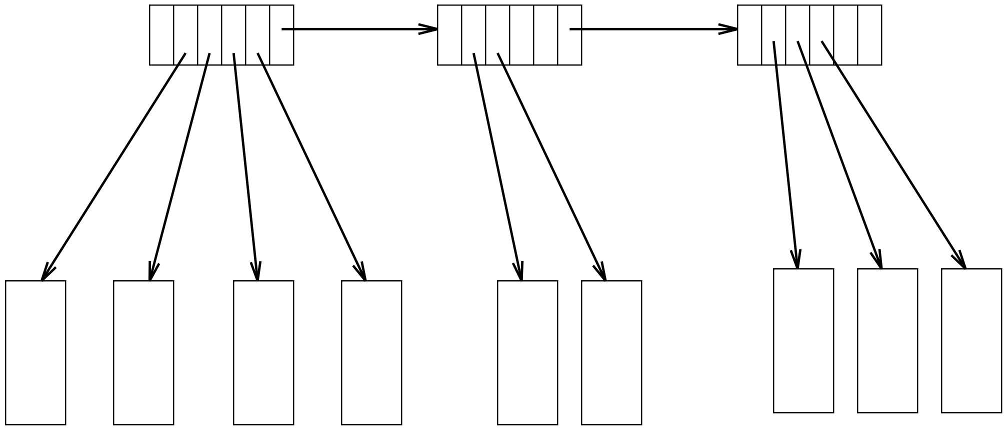

Figure 7.6: Liste chaînée des blocs libres dans le système de fichiers "étendu"

Chaque entrée de répertoire contient :

- un numéro d'i-nœud codé sur 4 octets ;
- la longueur de l'entrée de répertoire, codée sur 2 octets ;
- la taille du nom de fichier contenu dans cette entrée, codée sur 2 octets ;
- le nom de fichier lui-même utilisant un nombre d'octets multiple de 4.

La structure d'un répertoire contenant les fichiers f1, nom_plus_long et f2 est représentée sur la figure 7.7.

i1	12	1	f1	i2	24	13	nom_plus_long	i3	12	2	f2

Figure 7.7: Structure d'un répertoire dans le système de fichiers "étendu"

En fait un répertoire peut être considéré comme une liste chaînée d'entrées : à partir d'une entrée, il est possible d'accéder à la suivante en ajoutant la taille de l'entrée courante à la position actuelle dans le répertoire.

4.3.3 Structure physique du système de fichiers

Un système de fichiers "étendu" est composé :

- d'un bloc de boot contenant le code lancé lors du démarrage du système ;

- d'un super bloc contenant les caractéristiques du système de fichiers :
  - nombre de blocs ;
  - nombre d'i-nœuds ;
  - adresse du premier bloc libre ;
  - nombre de blocs libres ;
  - adresse du premier i-nœud libre ;
  - nombre d'i-nœuds libres ;
  - taille des blocs logiques ;
  - taille maximale d'un fichier ;
  - signature ;
- d'une table des i-nœuds contenant les renseignements sur les fichiers existants. Chaque i-nœud caractérisant un fichier contient :
  - le type et les droits d'accès associés au fichier ;
  - la taille du fichier en octets ;
  - la date de dernière modification ;
  - l'identificateur de l'utilisateur propriétaire du fichier ;
  - le nombre de noms (i.e. liens) associés au fichier ;
  - 12 adresses de blocs (9 adresses de blocs directs, l'adresse d'un bloc indirect, l'adresse d'un bloc doublement indirect et l'adresse d'un bloc triplement indirect) ;
- des blocs de données utilisés pour stocker le contenu des fichiers et des répertoires.

La structure du système de fichiers sur disque est représentée sur la figure 7.8.

Secteur de boot	Super Block	Table des i-nœuds	Blocs de données

Figure 7.8: Structure physique du système de fichiers "étendu"

4.4 Le second système de fichiers "étendu"

4.4.1 Objectifs

Nous avons conçu et développé le second système de fichiers "étendu" afin de lever les limitations héritées du système de fichiers Minix ou introduites dans la première version :

- De même que le système de fichiers Minix, le système de fichiers "étendu" ne conserve que la date de dernière modification d'un fichier sans sauvegarder la date de création ni la date de dernier accès.
- Le nombre de blocs alloués à chaque fichier n'est pas sauvegardé dans les i-nœuds.
- L'utilisation de listes chaînées de blocs et d'i-nœuds disponibles permet d'économiser de la place disque mais entraine une dégradation des performances. En effet, au fur et à mesure que des fichiers sont créés et supprimés, des blocs sont alloués et désalloués dans un ordre quelconque. La liste des blocs libres cesse vite d'être triée et le système alloue des blocs répartis sur tout le système de fichiers. L'accès à un fichier constitué de blocs distants provoque des déplacements importants des têtes de lecture-écriture du disque qui augmentent les temps d'entrées-sorties.

De nouvelles fonctionnalités ont également été ajoutées au système de fichiers :

- la sauvegarde temporaire de fichiers supprimés permettant d'implémenter un programme utilitaire de restauration des fichiers détruits ;
- la réservation d'un certain pourcentage de blocs pour le "super-utilisateur" ;
- la gestion de versions pour les i-nœuds ;
- la duplication des informations cruciales du système de fichiers permettant leur restauration si elles sont endommagées ;
- la gestion de listes de contrôle d'accès permettant une protection des fichiers plus fine que le mécanisme traditionnellement utilisé sous Unix ;
- la gestion de blocs de taille variable : alors que les autres systèmes de fichiers ne supportent que des blocs de 1 kilo-octets, le second système de fichiers "étendu" supporte des blocs de plus grande taille.

Ce système de fichiers offre une structure de répertoires identique à celle de son prédécesseur permettant ainsi de stocker des noms de fichiers de taille variable en optimisant la place utilisée dans chaque entrée de répertoire.

4.4.2 Structure physique du système de fichiers

Un système de fichiers est composé de plusieurs ensembles logiques de blocs comme représenté sur la figure 7.9. La décomposition en ensembles, qui peuvent être considérés comme des “sous systèmes de fichiers”, permet d'offrir :

- un plus grand degré de résistance aux pannes, puisque les informations cruciales sont dupliqués dans plusieurs ensembles ;
- des stratégies d'allocation optimisées présentées plus loin.

Secteur de boot	Ensemble de blocs 1	Ensemble de blocs 2	...	Ensemble de blocs N

Figure 7.9: Structure physique du second système de fichiers “étendu”

4.4.3 Structure physique d'un ensemble de blocs

Chacun des ensembles logiques contenus dans le système de fichiers contient :

- une copie du super bloc contenant :
  - le nombre de blocs ;
  - le nombre de blocs réservés au “super-utilisateur” ;
  - le nombre d'i-nœuds ;
  - le nombre de blocs libres ;
  - le nombre d'i-nœuds libres ;
  - les tailles d'un bloc logique et d'un fragment logique ;
  - le nombre de blocs par ensemble ;
  - le nombre de fragments par ensemble ;
  - le nombre d'i-nœuds par ensemble ;
  - la date de dernier montage du système de fichiers ;
  - la date de dernière écriture du super bloc ;
  - la signature ;
  - un indicateur de validité mis à 0 lorsque le système de fichiers est monté et remis à 1 lorsqu'il est démonté ;

- une copie des descripteurs des ensembles composant le système de fichiers. Chaque descripteur contient :
  - l'adresse du bloc contenant le vecteur de bits indiquant les blocs alloués ;
  - l'adresse du bloc contenant le vecteur de bits indiquant les i-nœuds alloués ;
  - l'adresse du premier bloc contenant la partie de la table des i-nœuds ;
  - le nombre de blocs libres dans le groupe ;
  - le nombre d'i-nœuds libres dans le groupe ;
  - le nombre de répertoires contenus dans le groupe ;
- un vecteur de bits indiquant les blocs alloués et disponibles ;
- un vecteur de bits indiquant les i-nœuds alloués et disponibles ;
- une partie de la table des i-nœuds. Chaque i-nœud contient :
  - le type et les drois d'accès associés au fichier ;
  - la taille du fichier en octets ;
  - les dates de création, de dernier accès et de dernière modification ;
  - la date de suppression du fichier ;
  - l'identificateur de l'utilisateur propriétaire du fichier ;
  - le nombre de noms (i.e. liens) associés au fichier ;
  - le numéro de version du fichier ;
  - le nombre de blocs alloués au fichier ;
  - des pointeurs sur les listes de droits d'accès associés au fichier ;
  - 15 adresses de blocs (12 adresses de blocs directs, l'adresse d'un bloc indirect, l'adresse d'un bloc doublement indirect et l'adresse d'un bloc triplement indirect) ;
  - divers champs réservés pour des extensions futures ;
- des blocs de données utilisés pour stocker le contenu des fichiers et des répertoires.

La structure d'un ensemble de blocs est représentée sur la figure 7.10.

Super bloc	Descripteurs	Bitmap blocs	Bitmap i-nœuds	Table des i-nœuds	Blocs de données

Figure 7.10: Structure d'un ensemble de blocs dans le second système de fichiers “étendu”

4.4.4 Allocations optimisées

Le second système de fichiers utilise de nombreuses optimisations lors d'allocation de blocs et d'i-nœuds :

- Les i-nœuds de fichiers sont alloués, autant que possible, dans le même ensemble que l'i-nœud de son répertoire père. De la sorte, lors de l'analyse de noms de fichiers, il est possible, dans de nombreux cas de limiter la recherche à un seul groupe.
- Les i-nœuds alloués à des répertoires sont répartis sur l'ensemble des ensembles composant le système de fichiers de manière à maintenir une moyenne de répertoires équitable dans l'ensemble des groupes.
- Les blocs d'un fichier sont alloués, autant que possible, dans le même ensemble que l'i-nœud correspondant. De la sorte, lors de l'accès au contenu d'un fichier, la lecture des blocs après la lecture de l'i-nœud donne lieu à un déplacement minimal de la tête de lecture-écriture. De plus, comme la majorité des blocs constituant un fichier sont localisés dans le même groupe, la lecture du fichier dans son intégralité donne lieu également à des déplacements minimaux.
- Lorsque des écritures de taille importante sont effectuées sur un fichier et que plusieurs blocs sont alloués, le système de fichiers essaie d'allouer des blocs consécutifs afin de permettre des performances optimales lors des lectures ultérieures de ces blocs.

Certaines de ces optimisations se rapprochent des stratégies implémentées dans le système de fichiers présent dans la version BSD d'Unix [McKusick *et al.* 1984] mais nous n'avons pas jugé judicieux de prendre en compte les caractéristiques physiques des disques sous-jacents comme le fait ce système de fichiers. La prise en compte de ces caractéristiques entrainerait des algorithmes beaucoup plus complexes. De plus, de nombreux disques modernes présentent au système d'exploitation des caractéristiques "logiques" différentes des caractéristiques physiques réelles. Il est donc vain d'essayer d'optimiser l'allocation des blocs en tenant compte de ces caractéristiques puisque de telles optimisations ne correspondraient pas à la répartition physique des blocs sur le disque.

4.4.5 Utilisation de blocs de tailles variables

Une autre innovation importante de ce système de fichiers consiste à permettre l'utilisation de blocs de taille variable. Alors que les autres systèmes de fichiers ne gèrent que des blocs de un kilo-octets, le second système de fichiers "étendu" permet de spécifier une taille de blocs logique plus importante. L'accroissement de la taille des blocs permet d'obtenir de meilleures performances lors des entrées-sorties. En effet, si on multiplie la taille des blocs par un facteur N, N fois moins de requêtes d'entrées-sorties sont nécessaires pour accéder à la même quantité de données, réduisant ainsi le temps de déplacement des têtes de lecture-écriture. De plus, en multipliant la taille des blocs logiques, de nombreux fichiers peuvent être exprimés sans bloc d'indirection, ce qui réduit ainsi le nombre d'entrées-sorties nécessaires.

L'accroissement de la taille des blocs logiques permet donc d'obtenir une amélioration importante des performances mais elle entraine également une mauvaise utilisation de l'espace disque due à la fragmentation interne. En effet, statistiquement la moité seulement du dernier bloc alloué à un fichier est utilisée. Plus la taille des blocs est importante, plus la perte d'espace disque est considérable. Des études [Leffler *et al.* 1989] ont montré que l'utilisation de blocs de 4 kilo-octets provoque une perte d'environ 45% de l'espace disque.

Afin de limiter la fragmentation interne dans les fichiers, nous avons minimisé la taille du dernier bloc alloué à un fichier. En effet, comme la perte d'espace disque est proportionnelle à la taille du dernier bloc, la réduction de cette taille minimise cette perte.

Afin de réduire la taille du dernier bloc, nous avons considéré des unités d'allocation plus petites qu'un bloc. Dans le second système de fichiers "étendu", un fichier est composé de plusieurs blocs complets et de quelques unités d'allocation consécutives. Considèrons un fichier contenant 14334 octets situé sur un système de fichiers composé de blocs de 4 kilo-octets et d'unités d'allocation de 1 kilo-octets. Un tel fichier sera constitué de 3 blocs complets, soit 12288 octets, et de deux unités d'allocation consécutives, soit 2048 octets. 14336 octets sont donc alloués physiquement au fichier alors que 16384 octets seraient nécessaires si le système de fichiers ne gérait que des blocs de 4 kilo-octets. Il est ainsi possible de réduire la place disque inutilisée de 45% à 11% environ [McKusick *et al.* 1984, Leffler *et al.* 1989].

Afin de gérer des unités d'allocation plus petites que les blocs, le vecteur de bits présent dans chaque ensemble contient un bit par unité d'allocation. Lors de l'allocation d'espace disque, le système recherche autant d'unités d'allocation consécutives qu'il est nécessaire.

5 Conclusion

Nous avons présenté, dans ce chapitre, la méthode de développement utilisée pour implémenter Masix : nous utilisons le système Linux afin d'effectuer un développement croisé de Masix.

Le système Linux offre un ensemble relativement complet des fonctionnalités Unix mais, dans sa version de base, imposait d'importantes restrictions sur la gestion des fichiers rendant quasiment impossible le développement croisé de Masix.

Nous avons exposé comment nous avons amélioré la gestion des fichiers dans Linux en participant à la définition d'une couche intermédiaire de gestion des fichiers et en implémentant deux systèmes de fichiers plus évolués que le système de fichiers Minix présent à l'origine. Ces systèmes de fichiers, inclus dans la distribution officielle de Linux, sont désormais largement utilisés par les milliers d'utilisateurs de Linux de part le monde.

De plus, le deuxième de ces systèmes de fichiers constitue le système de fichiers de base de Masix. Son inclusion dans Linux permet d'accéder facilement au système de fichiers de Masix pour initialiser son contenu.

Chapitre 8

Le Système Virtuel

1 Introduction

Afin d'obtenir un système réparti, il est nécessaire d'ajouter des serveurs de communication et de partage de ressources dans Masix. La coopération de ces serveurs s'exécutant sur plusieurs stations connectées en réseau forme le système virtuel.

Masix a été défini afin de permettre facilement l'ajout de nouveaux modules logiciels et l'exécution simultanée de plusieurs environnements offrant des interfaces et des fonctionnalités différentes. Ces principes de conception permettent d'intégrer aisément les serveurs formant le système virtuel et d'y accéder de manière transparente.

Nous nous intéressons, dans ce chapitre, à la façon dont les serveurs formant le système virtuel peuvent être intégrés dans Masix. Nous présentons tout d'abord les fonctionnalités nécessaires pour former le système virtuel de Masix. Ces fonctionnalités sont ensuite détaillées afin de présenter les services répartis offerts par Masix. Enfin, nous décrivons comment ces services peuvent être intégrés dans le système de manière transparente pour les utilisateurs.

2 Fonctionnalités

Les fonctionnalités réparties de Masix visent à offrir un partage transparent de ressources afin d'optimiser leur utilisation sur un réseau de stations. Ce partage de ressources est effectué par plusieurs serveurs qui se transmettent des informations en utilisant les mécanismes de communication réseaux inclus dans le système.

Ce partage de ressources concerne principalement :

- les processeurs :

  des mécanismes de placement et de migration de processus permettent de répartir les processus sur l'ensemble des stations du réseau.

- les fichiers :

  des systèmes de fichiers partagés permettent d'accéder aux mêmes données sur l'ensemble des stations. Cet accès est effectué de la même façon que pour les fichiers situés sur les disques locaux.

- les périphériques :

  il est fréquent que, dans un réseau, certains périphérique ne soient pas en nombre suffisant pour en connecter un ou plusieurs exemplaires à chaque station. Le système doit donc permettre à des applications d'accéder à des périphériques situées sur des stations distantes. La notion d'accès transparent est, dans ce cas, beaucoup moins importante. En effet, il est généralement nécessaire de citer explicitement la station possédant le périphérique afin d'obtenir le résultat attendu. Il est inconcevable de laisser le système choisir de lui-même un des périphériques sans que l'utilisateur puisse intervenir sur ce choix.

- les informations partagées :

  de nombreuses informations, bien que ne faisant pas partie du système à proprement parler, doivent être accessibles par des applications sur l'ensemble des stations. Par exemple, les informations définissant les utilisateurs doivent pouvoir être utilisées par les applications gérant les connections sur chacune des stations.

  Le système doit donc inclure des mécanismes de partage d'informations permettant à plusieurs applications d'y accéder. Ces mécanismes doivent inclure des fonctionnalités de mises à jour permettant de conserver la cohérence des données.

3 Services offerts

3.1 Communication entre stations

Afin de permettre la mise en œuvre de services répartis, une couche de communication doit être offerte par le système afin de permettre de communiquer via un réseau. Cette couche est utilisée par les serveurs composant le système lorsqu'ils ont besoin de transmettre des informations entre stations.

Les tâches de cette couche sont les suivantes :

- maintenir une liste des stations connectées sur le réseau :

  un serveur doit connaître le nom d'une station pour lui transmettre des informations et la couche de communication doit être capable d'émettre des données vers la station. Ce service de nommage est donc nécessaire.

- maintenir une liste des serveurs communiquants sur la station locale :

  lors de l'arrivée de données adressées à la station, la couche de communication doit être capable de transmettre les informations au serveur local concerné. Elle doit donc gérer une liste faisant correspondre des adresses locales à des serveurs.

- transmettre et recevoir des informations sur le réseau :

  la couche de communication doit, bien évidemment, assurer également la transmission de données sur le réseau. Pour cela, elle doit implémenter un ou plusieurs protocoles de communication réseau.

3.2 Gestion de processus

La gestion répartie de processus permet d'optimiser la charge globale du système en utilisant l'ensemble des processeurs disponibles. Le placement de processus permet de répartir la charge sur l'ensemble des stations lors du début de l'exécution des programmes. La migration dynamique de processus permet, pendant l'exécution de programmes, de modifier cette répartition afin de prendre en compte une évolution des critères de répartition.

Il nous parait intéressant, dans Masix, de prévoir les deux possibilités. En effet, le placement statique de processus ne permet pas d'obtenir une répartition optimale de la charge : dans la mesure où il est effectué à priori lors du lancement des processus, il ne peut prendre en compte l'état du système qu'à ce moment. L'évolution de l'activité des stations n'est pas gérée par le placement statique.

Par contre, la migration de processus permet de remettre en question le placement des processus en fonction de l'évolution de l'état du système réparti. Si une station devient subitement surchargée ou inactive, il est possible de déplacer un ou plusieurs processus afin de ré-équilibrer la charge globale du système.

Un des inconvénients de la migration réside dans son coût. Le déplacement d'un processus d'une station à une autre nécessite de :

- suspendre le processus sur la station d'origine ;
- créer le processus dans un état suspendu sur la station cible ;
- transférer son état de la station d'origine vers la station cible ;
- supprimer le processus sur la station d'origine ;
- reprendre l'exécution du processus sur la station cible.

Ces opérations provoquent un surcoût :

- pour le processus :

  pendant la migration, le processus est suspendu et ne peut être réactivé qu'à la fin du transfert de son état. Pour l'utilisateur, le processus est inactif durant ce temps.

- pour le système :

  la migration d'un processus entraine de nombreuses communications réseau.

Il est donc nécessaire de minimiser le nombre de migrations de processus. De plus, le système doit prendre en compte les communications entre processus : si deux processus communiquent entre eux, il est judicieux de les exécuter sur la même station afin d'utiliser des mécanismes de communication locaux plutôt que des échanges par réseau.

3.3 Systèmes de fichiers

La gestion de fichiers répartis permet d'offrir aux utilisateurs la même représentation des fichiers quelle que soit la station qu'ils utilisent. L'accès à un fichier distant doit être effectué par les mêmes opérations que pour les fichiers locaux. Il est ainsi possible, pour les utilisateurs, d'accéder de manière transparente aux fichiers gérés par le système réparti.

L'accessibilité à une arborescence gérée par une station serveur est rendue possible par une opération de montage logique. Ce montage, de même que pour les systèmes de fichiers locaux, permet de "connecter" un arbre de fichiers dans l'arborescence des fichiers.

Afin d'offrir la même vision des fichiers sur toutes les stations composant le système virtuel, il est nécessaire que les répertoires de montage soient les mêmes sur l'ensemble des machines. Il faut donc, contrairement aux approches classiques, que ce soit la station serveur qui spécifie à la station cliente le point de montage. Ainsi, sur toute station utilisant la même arborescence de fichiers, il est possible de citer les noms de fichiers en se référant aux mêmes chemins d'accès.

3.4 Partage de périphériques

L'accès à des périphériques distants permet d'utiliser tout type de périphérique connecté sur le réseau même s'il n'est pas directement connecté sur la station courante.

Nous distinguons deux modes d'accès différents aux périphériques :

1. l'accès logique :

   un processus accède à un périphérique par un protocole de dialogue qui lui est lié. Un serveur, chargé de gérer le périphérique, reçoit et traite les requêtes transmises par les processus. Ces requêtes ne correspondent pas à des opérations d'entrées-sorties physiques mais à un protocole de plus haut niveau. Les processus désirant accéder au périphérique doivent respecter ce protocole en citant explicitement le serveur. Ce type d'accès n'est donc pas transparent pour les processus utilisateur.

Ce mécanisme est couramment utilisé dans les systèmes actuels pour les périphériques non partagables : un serveur est chargè de traiter séquentiellement les requêtes.

2. l'accès physique :

   un processus accède à un périphérique distant selon les mêmes opérations d'entrées-sorties physiques que pour les périphériques locaux. Une couche d'interface est chargée de transformer les opérations d'entrées-sorties en requêtes transmises à un serveur distant chargé de les réaliser. Ce type d'accès permet d'utiliser des périphériques distants de manière transparente pour l'utilisateur puisqu'il y accède par les mêmes opérations.

Seul l'accès physique nécessite un support de la part du système. En effet, alors que l'accès logique peut être implémenté par des processus utilisateur en implémentant un protocole de haut niveau, l'accès physique implique de rediriger des opérations d'entrées sorties sur des serveurs distants.

3.5 Informations partagées

Le partage d'informations liées à des applications ne nécessite pas de support particulier du système lui-même. Si des applications ont besoin de données globales, elles peuvent utiliser des fonctions d'interface chargées d'accéder aux données. Il est ainsi possible d'implémenter une couche d'accès aux informations en mode utilisateur. Seuls les mécanismes de communication entre processus situés sur des machines distantes doivent être assurés par le système.

4 Stratégies et choix opérationnels

4.1 Nommage et communications

Nous distinguons deux types de communications réseau dans Masix :

1. les communications entre serveurs :
2. les communications entre processus utilisateur.

Alors que les communications entre processus utilisateur peuvent être implémentées en utilisant des appels système spécifiques comme dans Unix, il est nécessaire de permettre une communication transparente entre serveurs, qu'ils s'exécutent sur la même station ou non.

En effet, nous avons expliqué dans le chapitre 5 comment les requêtes concernant une entité sont transmises au serveur concerné en utilisant un port caractérisant l'entité. Afin de permettre une interface uniforme à n'importe quel serveur, qu'il soit local ou distant, il est nécessaire de permettre une communication par port de la même façon.

Masix utilise donc des ports locaux pour représenter les objets et les serveurs distants. Chaque port correspondant à une entité distante est gérée par un serveur d'interface local qui transmet les requêtes au serveur distant.

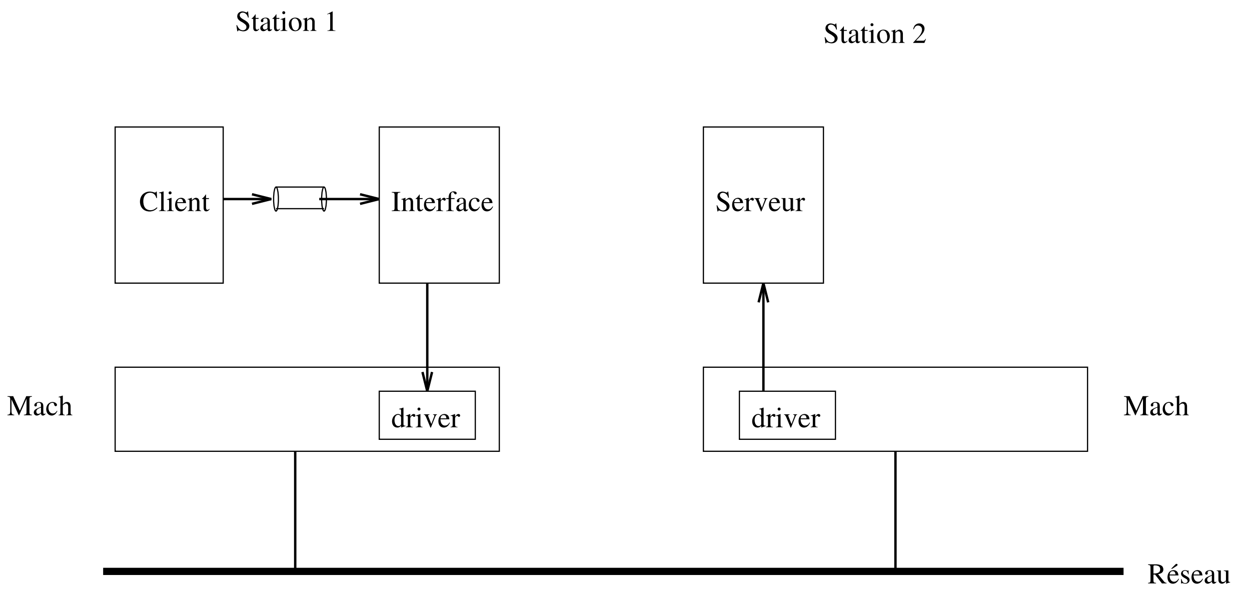

Figure 8.1: Communication entre serveurs distants

Sur la figure 8.1, une communication entre un client et un serveur sur deux machines différentes est représentée. Le client transmet sa requête sur un port local géré par un serveur d'interface local. Ce dernier transmet ensuite la requête au serveur distant en utilisant les mécanismes de communication réseau. Le résultat de la requête est, de manière symétrique, renvoyé par le serveur distant au serveur d'interface qui le transmet au client.

Pour communiquer avec un processus utilisateur distant, par contre, des appels système spécifiques sont utilisés et traités par un serveur de communication réseau. La figure 8.2 représente un processus E transmettant un message à un processus R situé sur une autre station. E transmet sa requête, via un appel système, au serveur de communication réseau qui l'envoie au serveur de communication sur la machine distante. Ce dernier mémorise le message jusqu'à ce que le processus R lise le message par un autre appel système spécifique.

4.2 Partage de fichiers

Comme nous l'avons expliqué dans le chapitre 6, les accès aux fichiers sont transformés par Masix en accès mémoire et les blocs nécessaires sont chargés en mémoire centrale sur défaut de page, lorsque les pages correspondantes sont référencées.

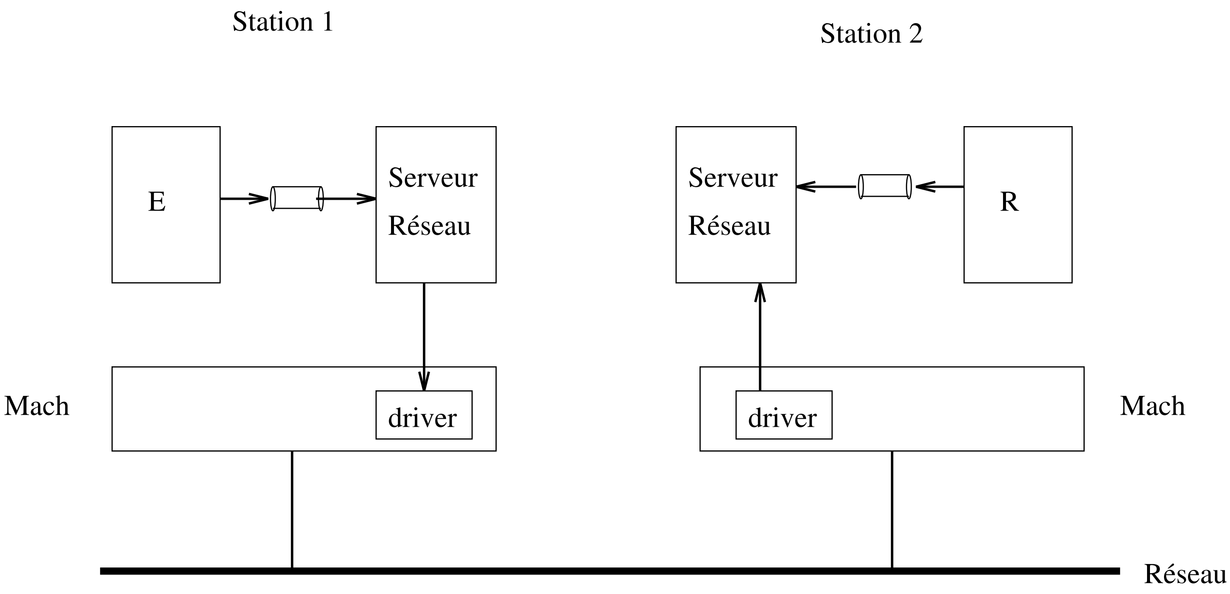

Figure 8.2: Communication entre processus distants

L'ajout d'un système de fichiers répartis dans Masix consiste à lancer l'exécution d'un nouveau SPF dans le système. Ce SPF a pour rôle de traiter les requêtes de manipulation de fichiers et de charger les blocs correspondants en mémoire. L'accès physique aux fichiers est effectué par le SPF en transmettant des requêtes à un serveur distant de fichiers (SDF) qui s'exécute sur la station possédant l'arborescence de fichiers sur un disque local.

Le SDF est chargé de la gestion physique des fichiers. Il répond aux requêtes de lecture et d'écriture de blocs transmises par les SPF situés sur les différentes stations clientes.

L'accès simultané aux mêmes fichiers par plusieurs processus s'exécutant sur des stations différentes peut donner lieu à des problèmes d'incohérence. En effet, comme les blocs des fichiers sont chargés dans la mémoire de chaque station par les SPF lorsqu'ils sont accédés, la modification de leur contenu ne peut être immédiatement connue du serveur et il est possible que le même bloc soit modifié sur plusieurs stations si aucun mécanisme de gestion de cohérence n'est assuré.

Afin d'assurer une gestion cohérente des fichiers, le SDF maintient une liste des fichiers ouverts sur les machines clientes ainsi que la liste des blocs transmis à chaque station cliente. Lorsqu'un SPF lui transmet une requête de lecture de bloc, le SDF choisit de ne pas honorer la requête dans les cas suivants :

- le bloc a déjà été transmis à un processus ayant ouvert le fichier en écriture sur une autre station.
- le bloc est demandé par un processus ayant ouvert le fichier en écriture et le bloc a déjà été transmis à un autre processus ayant ouvert le fichier en lecture sur une autre station.

Dans ces deux cas, le SDF mémorise la requête mais ne la traite pas immédiatement. Lorsque le bloc concerné est retransmis au SDF, la requête est alors honorée. On a ici un principe de jeton par bloc d'un fichier :

- plusieurs stations peuvent posséder simultanément un jeton de lecture d'un bloc ;
- une seule station peut posséder le jeton d'écriture d'un bloc ;
- si une station possède le jeton d'écriture d'un bloc, aucune autre ne peut posséder le jeton de lecture du bloc.

Le SDF est donc un serveur de fichiers à état puisqu'il conserve des informations sur les fichiers ouverts sur les stations clientes. Cet état lui permet de gérer les conflits d'accès aux fichiers mais pose un problème en cas de panne ou de redémarrage de la station serveur. En effet, si la station serveur est réinitialisée, l'état conservé par le SDF est perdu.

Afin de gérer les pannes du serveur de fichiers, chaque SPF conserve également une liste des jetons qui lui ont été attribués et de ceux qu'il a demandés au SDF. Lorsque le SDF s'initialise, il effectue une diffusion sur l'ensemble des stations connectées afin de prévenir chaque SPF de sa panne. En réponse à ce message d'initialisation, chaque SPF doit envoyer au SDF la liste des jetons qu'il possède ou qu'il a demandés. Il est ainsi possible, en cas de panne, de reconstruire l'état du SDF à partir de l'ensemble des SPF.

4.3 Accès aux périphériques distants

L'accès aux périphériques dans Masix est effectuée en référençant chaque périphérique par un port géré par un serveur. Dans le cas de périphérique connecté localement à la station, ce serveur interprête la requête transmise sur le port et l'exécute en appelant le pilote du périphérique contenu dans Mach.

Dans le cas d'un périphérique distant, le serveur local sert uniquement de serveur d'interface. Lorsqu'il reçoit une requête, il la transmet au serveur du périphérique situé sur une autre station, attend que ce dernier lui transmette le résultat de la requête, puis renvoie ce résultat au processus client.

Il s'agit ici de l'application aux périphériques des principes de communication exposés plus haut et représentés sur la figure 8.1.

5 Conclusion

Nous avons présenté, dans ce chapitre, les principes de conception des services formant le système virtuel de Masix.

Les serveurs assurant les services répartis tirent parti du caractère multi-serveurs de Masix et peuvent être intégrés en utilisant les principes de nommage et de communication que nous avions définis dans le chapitre 5.

L'intégration de services répartis est effectuée en utilisant un serveur d'interface sur chaque station. Ce serveur est chargé de transmettre les requêtes au serveur “réel” situé sur une machine distante en dialoguant avec ce dernier. Les processus utilisateur qui communiquent avec le serveur d'interface s'adressent de manière transparente au serveur distant.

La gestion des fichiers partagés comporte des traitements supplémentaire destinés à maintenir la cohérence des informations traités. Le serveur de fichiers conserve un état de tous les blocs des fichiers chargés par les clients et peut différer des transmissions de données lorsqu'il détecte des coflits d'accès. Dans la mesure où Masix transforme les opérations sur fichiers en accès mémoire, cette gestion de fichiers répartie peut être utilisée comme base d'un mécanisme de mémoire virtuelle répartie.

Chapitre 9

Conclusion

Le système d'exploitation MASIX constitue une nouvelle implémentation sous forme de serveurs coopérants s'exécutant en mode utilisateur. Le système a été conçu et structuré de façon à être facile à faire évoluer et à maintenir.

1 Avantages de la structure multi-serveurs

Masix est composé de plusieurs serveurs qui s'exécutent en mode utilisateur. Chacun des serveurs est chargé de gérer un ensemble d'objets d'un certain type et de répondre aux requêtes des clients sur ces objets. Les serveurs et les clients disposent d'espace d'adressage distincts et communiquent par envois de messages contenant les requêtes et les résultats des appels système.

Cette structuration sous forme de serveurs rend l'évolution de Masix facile. En effet, alors que la mise à jour d'un système monolithique nécessite la restructuration de son noyau complet, l'ajout ou la modification de services de Masix se ramène à l'ajout ou au remplacement de serveurs.

Cette modularité facilite également la maintenance du système. Dans la mesure où chaque serveur implémente un nombre d'opérations réduit sur des objets d'un seul type, sa programmation et sa mise au point sont plus faciles. De plus, il est possible de tester un serveur avant de l'inclure dans le système.

La décomposition en serveurs permet enfin d'assurer un haut degré de protection dans le système. Comme chaque serveur possède un espace d'adressage distinct et communique avec le reste du système par envois de messages selon une interface bien définie, il est impossible qu'il puisse, intentionnellement ou non, provoquer des effets de bord sur les autres tâches du systèmes. De plus, la panne d'un serveur ne provoque pas un arrêt brutal du système : si un serveur s'arrête, cela n'a pas d'incidence sur les autres qui peuvent soit continuer à s'exécuter soit procéder à une sauvegarde de leurs données cruciales avant une réinitialisation du système.

2 Optimisations incluses dans Masix

L'utilisation de serveurs coopérants est généralement moins performante que la structure monolithique. En effet, les échanges de message et les changements de contexte sont plus coûteux en temps que les trappes utilisées dans un système traditionnel. Masix contient donc de nombreuses optimisations pour minimiser ce surcoût.

L'implémentation de nombreux services sous forme de fonctions de librairie permet d'exécuter certains traitements dans le contexte du processus appelant en évitant d'effectuer une requête à un serveur. De plus, la conservation d'informations système dans l'espace d'adressage des processus leur permet d'y accéder directement également.

Les communications entre serveurs sont également minimisées. La duplication de données et de code entre plusieurs serveurs permet d'éviter des échanges de messages entre eux.

De plus, nous avons essayé, autant que possible, de supprimer les indirections lors de requêtes. Un processus client s'adresse donc directement au serveur concerné sans passer par une interface centrale. Pour les accès aux fichiers et aux périphériques, un processus dialogue même directement avec le serveur chargé d'assurer les opérations d'entrées-sorties sans effectuer de requêtes au serveur de fichiers virtuel.

3 Communication générique avec les serveurs

Les communications entre clients et serveurs sont effectuées par l'envoi de message sur un port représentant l'objet manipulé. Comme ces messages respectent des protocoles normalisés, il est extrêmement facile d'associer plusieurs sémantiques ou de nouveaux algorithmes à chaque opération. La sélection du serveur à utiliser est dynamique puisqu'elle est effectuée lors de la transmission de la requête.

Ceci permet d'exécuter simultanément des implémentations différentes des mêmes opérations sur des objets de même type. Cela est particulièrement intéressant pour implémenter des opérations réparties de manière transparente. En effet, un processus client effectuera les mêmes requêtes sur un objet qu'il soit local ou distant. Selon la localisation de l'entité accédée, plusieurs serveurs différents pourront être appelés. Dans le cas d'un objet local, l'opération est traitée sur la machine courante alors qu'elle est transmise à un serveur s'exécutant sur une autre machine dans le cas d'un objet distant.

4 Réalisation

Masix est actuellement en cours de développement au dessus de Mach et ne constitue pas encore un système complet. Toutefois, une partie essentielle du système, le système de fichiers, a été complètement implémenté et est déjà opérationnel dans le système Linux.

Ce système de fichiers comporte de nombreuses optimisations. Les routines d'allocation sont prévues pour allouer des blocs contigus à un fichier permettant ainsi de minimiser les temps d'accès séquentiel. L'utilisation de blocs de taille variable permet de diminuer le nombre d'entrées-sorties nécessaires pour l'accès aux fichiers. La gestion d'unités d'allocation de taille inférieure à celle des blocs minimise la fragmentation interne des fichiers.

Ce système de fichier est désormais inclus dans la distribution standard de Linux et est diffusé comme système de fichiers de base du système. À ce titre, il est utilisé par une communauté d'utilisateurs importante de part le monde.

5 Perspectives

Le travail futur sur Masix va consister, à très court terme, à finir l'implémentation du système afin qu'il puisse servir de base à son propre développement. Les serveurs de base seront intégrés un par un pour fournir un système local minimal constitué du système d'accueil et des serveurs composant l'environnement Unix.

La grande modularité et l'extensibilité de Masix tel que nous l'avons conçu vont permettre de l'utiliser à moyen terme comme un système hôte pour l'implémentations de nouvelles fonctionnalités. En effet, la gestion des environnements multiples va permettre, sans aucune modification du système local, d'intégrer de nouveaux serveurs découlant des recherches effectuées au laboratoire MASI. Ainsi, l'inclusion de Gatos [Folliot 1992] est déjà prévue dans Masix.

Nous souhaitons également porter Masix sur d'autres architectures matérielles. La première version de Masix est destinée à des micro-ordinateurs de type PC 386 mais est très peu dépendante du matériel puisqu'elle repose sur les fonctionnalités fournies par le micro-noyau Mach. Le portage sur un autre ordinateur nécessite donc uniquement la disponibilité de Mach sur l'architecture visée. À l'heure actuelle, Mach fonctionne sur de nombreuses architectures et il nous parait très intéressant de porter Masix sur plusieurs plates-formes matérielles afin de pouvoir utiliser ses fonctionnalités réparties dans un environnement hétérogène.

Bibliographie

[Abrossimov *et al.* 1989] V. Abrossimov, M. Rozier, et M. Shapiro. Virtual Memory Management in Chorus. In *Proceedings of Progress in Distributed Operating Systems and Distributed Systems Management*, April 1989.

[Albinson *et al.* 1991] L. Albinson, D. Grabas, P. Piovesan, M. Tromboff, C. Tricot, et H. Yassaie. UNIX on Loosely Coupled Architecture: the CHORUS/MIX Approach. In *Proceedings of the EIT Workshop on Parallel and Distributed Workstation Systems*, September 1991.

[ANSA 1987] *ANSA Reference Manual.* Technical report, Alvey Advanced Network System Architecture Project, 1987.

[Armand *et al.* 1989] F. Armand, M. Gien, F. Herrmann, et M. Rozier. Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues. In *Proceedings of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, pages 153–174, October 1989.

[Armand *et al.* 1990] F. Armand, F. Herrmann, J. Lipkis, et M. Rozier. Multi-threaded Processes in Chorus/MIX. In *Proc. of EUUG Spring'90 Conference*, pages 1–13, April 1990.

[Armand 1991] F. Armand. Give a Process to your drivers! In *Proceedings of the EurOpen Autumn 1991 Conference*, September 1991.

[AT&T 1989] *System V Interface Definition, Issue 3.* American Telephone & Telegraph, 1989.

[Bach 1986] M. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[Baker *et al.* 1991] M. Baker, J. Hartman, M. Kupfer, K. Shiriff, et J. Ousterhout. *Measurements of a Distributed File System.* Technical report, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, 1991.

[Balkovitch *et al.* 1985] E. Balkovitch, S. Lerman, et R. Parmelee. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28(11):1214–1224, November 1985.

[Balter *et al.* 1991] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Van Nguyen, E. Payre, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, et G. Vandôme. Architecture and Implementation of Guide, an Object Oriented Distributed System. *Computing Systems*, 4, 1991.

[Basket *et al.* 1977] F. Basket, J. Haward, et J. Montagne. Task Communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, 1977.

[Bensoussan *et al.* 1972] A. Bensoussan, C. Clingen, et R. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[Bernstein et Goodman 1981] P. Bernstein et N. Goodman. Concurency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[Bernstein 1985] P. Bernstein. A Loosely Coupled Distributed System for Reliable Storing Data. *IEEE Transactions on Software Ingineering*, SE-11(5):446–454, 1985.

[Birrell et Nelson 1984] A. Birrell et D. Nelson. Implementaing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[Black *et al.* 1988] D. Black, D. Golub, R. Rashid, A. Tevanian, et M. Young. *The Mach Exception Handling Facility.* Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, 1988.

[Black 1990] D. Black. *Scheduling and Resources Management Techniques for Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1990.

[Boyer *et al.* 1991] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, et D. Agimont. Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus. In *Proceedings of the Symposium on Experience with Distributed and Multiprocessor Systems*, March 1991.

[Boyer 1991] F. Boyer. A Causal Distributed Shared Memory Based on External Pagers. In *Proceedings of the Second MACH Symposium*, November 1991.

[Champine *et al.* 1990] G. Champine, D. Geer Jr, et W. Ruh. Project Athena as Distributed Computer System. *IEEE Computer*, 23:40–51, 1990.

[Cheriton 1984a] D. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, pages 19–42, 1984.

[Cheriton 1984b] D. Cheriton. *A Uniform I/O Interface and Protocol for Distributed Systems.* Technical Report Z/2 SYS1201, Stanford University, December 1984.

[Cheriton 1988] D. Cheriton. The V Distributed System. *Communications of the ACM*, 31:19–42, 1988.

[Comer et Stevens 1991] D. Comer et D. Stevens. *Internetworking with TCP/IP, Volume 2: Design, Implementation, and Internals.* Prentice Hall, 1991.

[Comer 1991] D. Comer. *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture.* Prentice Hall, second edition, 1991.

[Cooper et Draves 1988] E. Cooper et R. Draves. *C Threads.* Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, 1988.

[Coulouris et Dollimore 1988] G. Coulouris et J. Dollimore. *Distributed Systems: Concepts and Design.* Addison Wesley, 1988.

[Custer 1993] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[Daley et Dennis 1968] R. Daley et J. Dennis. Virtual Memory, Processes and Sharing in Multics. *Communications of the ACM*, 11(5):306–312, May 1968.

[Davidson *et al.* 1985] S. Davidson, H. Garcia, et D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[Dean et Armand 1992] R. Dean et F. Armand. Data Movement in Kernelized Systems. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 243–261, April 1992.

[DES 1977] Federal Information Processing Standards. *Data Encryption Standard*, 1977.

[Dijkstra 1968] E. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[Douglis *et al.* 1991] F. Douglis, M. Kaashoek, A. Tanenbaum, et J. Ousterhout. A Comparison of Two Distributed Systems: Amoeba and Sprite. *Computing Systems*, 4(3):353–384, December 1991.

[Douglis et Ousterhout 1987] F. Douglis et J. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, 1987.

[Douglis et Ousterhout 1991] F. Douglis et J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 9(4):757–785, August 1991.

[Drave *et al.* 1989] R. Drave, M. Jones, et M. Thompson. MIG – The Mach Interface Generator. 1989.

[Draves 1990] R. Draves. A Revised IPC Interface. In *Proceedings of the USENIX Mach Conference*, 1990.

[Draves 1991] R. Draves. Page Replacement and Reference Bit Emulation in Mach. In *Proceedings of the USENIX Mach Conference*, 1991.

[Fabry 1974] R. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Folliot 1992] B. Folliot. *Méthodes et Outils de Partage de Charge pour la Conception et la Mise en Oeuvre d'Application Parallèles dans les Systèmes Répartis Hétérogènes*. PhD thesis, Laboratoire MASI, Université Pierre et Marie Curie, December 1992.

[Forin *et al.* 1991] A. Forin, D. Golub, et B. Bershad. An I/O System for Mach. In *Proceedings of the USENIX Mach Conference*, 1991.

[Golub *et al.* 1990] D. Golub, R. Dean, A. Forin, et R. Rashid. Unix as an Application Program. In *Proceedings of the USENIX Summer Conference*, 1990.

[Goscinski 1991] A. Goscinski. *Distributed Operating Systems – The Logical Design*. Addison Wesley, 1991.

[Gray 1978] J. Gray. Notes on Database Operating Systems. *Lecture Notes in Computer Science*, 60:393–481, 1978.

[Guedes et Julin 1991a] P. Guedes et D. Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. In *Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems*, October 1991.

[Guedes et Julin 1991b] P. Guedes et D. Julin. *Writing the Mach3 Multi–Server System in C++*. Technical report, Design Review of the Research Institute, Open Software Fundation, February 1991.

[Hoare 1974] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 21(10):549–557, October 1974.

[Howard 1988] J. Howard. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[IEEE 1989] *Realtime Extension for Portable Operating Systems - Draft 9*. Institute of Electrical and Electronics Engineers, Inc, 1989.

[IEEE 1990a] *IEEE Standard Portable Operating System Interface for Computer Environments*. Institute of Electrical and Electronics Engineers, Inc, 1990.

[IEEE 1990b] *Threads Extension for Portable Operating Systems - Draft 5*. Institute of Electrical and Electronics Engineers, Inc, 1990.

[IEEE 1992] *SECURITY INTERFACE for the Portable Operating System Interface for Computer Environments - Draft 13*. Institute of Electrical and Electronics Engineers, Inc, 1992.

[Jolitz et Jolitz 1992] W. Jolitz et L. Jolitz. Porting UNIX to the 386: The Final Step. *Dr Dobb's Jounal*, 190:50–56, July 1992.

[Jones et Rashid 1986] M. Jones et R. Rashid. *Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems*. Technical Report CMU-CS-86-123, School of Computer Science, Carnegie Mellon University, 1986.

[Jones 1991] M. Jones. Bringing the C Libraries With Us into a Multi-Thread Future. In *Proceedings of the USENIX Winter Conference*, 1991.

[Julin *et al.* 1991] D. Julin, J. Chew, J. Stevenson, P. Guedes, P. Neves, et P. Roy. Generalized Emulation Services for Mach 3.0 – Overview, Experiences and Current Status. In *Proceedings of the USENIX Mach Symposium*, pages 13–26, November 1991.

[Klaus 1993] T. Klaus. What is Linux and how do I try it out ? *UNIXWorld*, March 1993.

[Kleiman 1986] S. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer USENIX Conference*, pages 260–269, June 1986.

[Krakowiak *et al.* 1990] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, et X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3):11–22, October 1990.

[Krakowiak 1987] S. Krakowiak. Les Systèmes d'Exploitation Répartis : évolution récente, tendances de la recherche. *Techniques et Sciences Informatiques*, 6(2):151–161, 1987.

[Lamport 1978] L. Lamport. Time, Clock and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[Lann 1983] G. Le Lann. Synchronisation in Local Area Networks: An Advanced Course. *Lecture Notes in Computer Science*, 184:361–395, 1983.

[Leffler *et al.* 1989] S. Leffler, M. McKusick, M. Karels, et J. Quaterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison Wesley, 1989.

[Lewine 1991] D. Lewine. *POSIX Programmer's Guide.* O'Reilly and Associates, Inc, 1991.

[Liskov 1972] B. Liskov. The Design of the Venus Operating System. *Communications of the ACM*, 15(3):144–149, March 1972.

[Litzkow *et al.* 1988] M. Litzkow, M. Livny, et M. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988.

[Loepere 1992a] K. Loepere, editor. *Mach 3 Kernel Interface.* Open Software Foundation and Carneggie Mellon University, 1992.

[Loepere 1992b] K. Loepere, editor. *Mach 3 Kernel Principles.* Open Software Foundation and Carneggie Mellon University, 1992.

[Loepere 1992c] K. Loepere, editor. *Server Writer's Guide.* Open Software Foundation and Carneggie Mellon University, 1992.

[Loepere 1992d] K. Loepere, editor. *Server Writer's Interface.* Open Software Foundation and Carneggie Mellon University, 1992.

[Malan *et al.* 1991] G. Malan, R. Rashid, D. Golub, et R. Baron. DOS as a Mach 3.0 Application. In *Proceedings of the USENIX Mach Symposium*, pages 27–40, November 1991.

[McKusick *et al.* 1984] M. McKusick, W. Joy, S. Leffler, et R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[Mockapetris et Dunlap 1988] P. Mockapetris et K. Dunlap. Development of the Domain System. In *Proceedings of the SIGCOMM'88 Symposium on Communications Architectures and Protocols*, pages 123–133, 1988.

[Mueller *et al.* 1983] E. Mueller, J. Moore, et G. Popeck. A Nested Transaction Mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 71–89, 1983.

[Mukta et Livny 1987] M. Mukta et M. Livny. Scheduling Remote Processing Capacity in a Workstations-Processor Bank Computing System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 2–9, 1987.

[Mullender *et al.* 1990] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Reneese, et H. van Straveren. Amoeba – A Distributed Operating System for the 1990s. *Communications of the ACM*, 15(5):44–53, May 1990.

[Mullender 1984] S. Mullender. *Distributed Systems Management in Wide Areas Networks.* Technical Report CS-R8419, Departement of Computer Science, Centre for Mathematics and Computer Science, 1984.

[Mullender 1989a] S. Mullender, editor. *Distributed Systems.* Addison Wesley, 1989.

[Mullender 1989b] S. Mullender. Protection. In S. Mullender, editor, *Distributed Systems.* Addison Wesley, 1989.

[Needham et Schroeder 1978] R. Needham et M. Schroeder. Using Encryption for Authentification in Large Networks of Computers. *Communication of the ACM*, 21:993–999, 1978.

[Needham 1985] R. Needham. Protection in Local Area Networks: an Advanced Course. *Lecture Notes in Computer Science*, 189:261–281, 1985.

[Needham 1989] R. Needham. Names. In S. Mullender, editor, *Distributed Systems.* Addison Wesley, 1989.

[Nelson *et al.* 1988] M. Nelson, B. Welch, et J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[Nelson 1981] B. Nelson. *Remote Procedure Call.* PhD thesis, Carneggie-Mellon University, 1981.

[Nelson 1990] V. Nelson. Fault-tolerant computing: Fundamental concepts. *Computers*, July 1990.

[Nichols 1987] D. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 5–12. ACM, 1987.

[Nicol *et al.* 1988] J. Nicol, G. Blair, et W. Shepherd. An Approach to the Multiple Copy Update. *Distributed Processing*, pages 537–550, 1988.

[Organick 1972] E. Organick. *The Multics System, an Examination of Its Structure.* M.I.T. Press, 1972.

[OSF 1990] *The Design of the OSF/1 Operating System.* Open Software Foundation, 1990.

[Ousterhout *et al.* 1988] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, et B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21:23–26, 1988.

[Places *et al.* 1992] F. Barbou Des Places, P. Bernadat, M. Condict, S. Empereur, J. Febvre, D. George, J. Loveluck, E. McManus, S. Patience, J. Rogado, et P. Roudaud. *Architecture and Benefits of a Multithreaded OSF/1 Server.* Technical report, Open Software Foundation Research Institute, 1992.

[Popek et Walker 1985] G. Popek et B. Walker, editors. *The LOCUS Distributed System Architecture.* MIT Press, 1985.

[Popek 1981] G. Popek. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the Eigth Symposium on Operating Systems Principles*, pages 169–177, 1981.

[Quaterman *et al.* 1985] J. Quaterman, A. Silbetshatz, et J. Peterson. 4.2BSD and 4.3BSD as Examples of the UNIX System. *ACM Computing Surveys*, 17(4):379–418, December 1985.

[Rashid et Robertson 1981] R. Rashid et G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eight Symposium on Operating Systems Principles*, pages 164–175, 1981.

[Rashid 1986] R. Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. In *Proceedings of the Fall Joint Computer Conference*. ACM/IEEE Computer Society, 1986.

[Raynal 1984] M. Raynal. *Algorithmique du Parallélisme: le Problème de l'Exclusion Mutuelle*. Dunod, 1984.

[Raynal 1988] M. Raynal. *Systèmes Répartis, réseaux, concepts, outils et algorithmes*. Eyrolles, 1988.

[Reid 1990] J. Reid. N(e)FS: the Protocol is the Problem. In *Summer 1990 United Kingdom Unix system User Group*, pages 77–84, July 1990.

[Ricart et Agrawala 1981] G. Ricart et A. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):9–17, 1981.

[Rifflet 1986] J.M. Rifflet. *La Programmation sous Unix*. Mc Graw-Hill, 1986.

[Rifflet 1990] J.M. Rifflet. *La Communication sous Unix*. Mc Graw-Hill, 1990.

[Robboy 1987] D. Robboy. A UNIX Port to the 80386. In Mitchell Waite, editor, *UNIX Papers for UNIX Developpers and Power Users*, pages 400–425. Howard W. SAMS & Company, 1987.

[Rosenstein *et al.* 1988] M. Rosenstein, D. Geer Jr, et P. Levine. The Athena Service Management System. In *Proceedings of the USENIX Winter Conference*, pages 203–212, February 1988.

[Rozier *et al.* 1988] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, et W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems*, 1(4):305–370, December 1988.

[Sandberg *et al.* 1985] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, et B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Conference*, 1985.

[Seawright et MacKinnon 1979] L. Seawright et R. MacKinnon. VM/370 – A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[Seltzer *et al.* 1993] M. Seltzer, K. Bostic, M. McKusick, et C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Conference*, January 1993.

[Silberschatz *et al.* 1991] A. Silberschatz, J. Peterson, et P. Galvin. *Operating System Concepts*. Addison Wesley, 3rd edition, 1991.

[Spector *et al.* 1985] A. Spector, D. Daniels, D. Duchamp, J. Eppinger, et R. Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the 10th Sysmposium on Operating System Principles*, pages 127–146. ACM, 1985.

[Spector 1986] A. Spector. Communication Support in Operating Systems for Distributed Transations. *Lecture Notes in Computer Science*, 248:313–324, 1986.

[Steiner *et al.* 1988] J. Steiner, B. Neuman, et J. Schiller. Kerberos: An authentification Service for Open Network Systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988.

[Stevens 1990] R. Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[Sun 1990] Sun Microsystems. *SunOS Reference Manual, Network and System Administration*, 1990.

[Tanenbaum *et al.* 1986] A. Tanenbaum, S. Mullender, et R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[Tanenbaum et van Renesse 1985] A. Tanenbaum et R. van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[Tanenbaum 1987] A. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, 1987.

[Tanenbaum 1992] A. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[Tevanian et Rashid 1987] A. Tevanian et R. Rashid. *Mach: A Basis for Future UNIX Development.* Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University, 1987.

[Tevanian 1987] A. Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1987.

[Theimer *et al.* 1985] M. Theimer, K. Lantz, et D. Cheriton. Pre-emptable Remote Execution Facilities for the V–System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12, 1985.

[Theimer 1986] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986.

[Treese 1988] G. Treese. Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD. In *Proceedings of the USENIX Winter Conference*, pages 175–182, 1988.

[van Reneese *et al.* 1989] R. van Reneese, A. Tanenbaum, et A. Wilschut. The design of a high-performance file server. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 22–27, 1989.

[Walker *et al.* 1983] B. Walker, G. Popek, R. English, C. Kline, et G. Theil. The locus distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 49–70, 1983.

[Welch et Ousterhout 1988] B. Welch et J. Ousterhout. Pseudo devices : User-level extensions to the Sprite file system. In *Proceedings of the USENIX Summer Conference*, pages 37–49, June 1988.

[Welch 1990] B. Welch. *Naming, State Management and User-Level Extensions in the Sprite Distributed File System*. PhD thesis, University of California, Berkeley, February 1990.

Annexe A

Services offerts par le Système Local

Cette annexe décrit les appels système implémentés dans le système local de Masix. Nous présentons tout d'abord les définitions des objets manipulés. Nous spécifions ensuite les codes d'erreur pouvant être retournés par tout appel système. Enfin, nous détaillons la liste des appels système implémentés avec leurs algorithmes.

1 Définitions

Nous définissons ici les termes employés par la suite. Pour chaque objet manipulé par le système, nous distinguons deux visions :

- celle de l'utilisateur ;
- celle des différents modules composant le système ;

1.1 Fichiers

Pour un utilisateur, un fichier est constitué d'une suite finie d'octets référencée par un nom. Un fichier est caractérisé par son nom et son type :

- fichier régulier ;
- répertoire ;
- fichier FIFO ;
- fichier spécial permettant l'accès à un périphérique.

Deux types d'opérations sur les fichiers peuvent être distingués :

- opérations de manipulation de fichiers, agissant de façon ponctuelle sur un fichier référencé par son nom ;
- opérations d'entrées-sorties, agissant sur un fichier ouvert référencé par un identificateur d'entrées-sorties alloué par le système lors de l'ouverture du fichier.

De manière interne, un fichier est représenté par le système comme une liste de blocs stockés sur un périphérique. Le système assure la correspondance entre les numéros logiques d'octets et le numéro de bloc physique qui les contient.

Le système utilise plusieurs types de descripteurs pour gérer les fichiers :

- descripteur d'entrées-sorties (DES) : un DES réference les informations relatives à un fichier ouvert par un processus ;
- descripteur d'entrées-sorties intermédiaire (DESI) : un DESI contient les informations relatives à une ouverture de fichier. Il existe un seul DESI si un processus ouvre un fichier puis crée des processus fils pouvant accéder à ce fichier. À chaque DES correspond un seul DESI et à chaque DESI correspond un ou plusieurs DES ;
- descripteur d'entrées-sorties global (DESG) : un DESG contient les informations relatives à un fichier ouvert dans le système. Il existe un seul DESG si plusieurs processus ouvrent le même fichier. À chaque DESI correspond un seul DESG et à chaque DESG correspond un ou plusieurs DESI ;
- descripteur virtuel de fichier (DVF) : un DVF contient les caractéristiques d'un fichier et est lié à sa localisation physique et à sa représentation. Chaque DVF est géré de manière interne par le module de gestion des fichiers concerné ;
- identificateur d'entrées-sorties (IES) : un IES caractérise un fichier ouvert par un processus. Il est retourné par le système lors d'une ouverture de fichier et est utilisé comme paramètre lors des requêtes d'entrées-sorties suivantes. À partir d'un IES, le système doit pouvoir accéder au DES correspondant ;
- identificateur virtuel de fichier (IVF) : un IVF identifie de manière unique un fichier. Il est indépendant de la localisation physique du fichier et de sa représentation. À partir d'un IVF, le système doit pouvoir accéder au DVF correspondant ;
- système de fichiers (SF) : un SF contient les informations liées aux fichiers sous un format spécifique. Tous les SF de même type dont gérés par les fonctions d'un même module.

1.2 Processus

Pour l'utilisateur, un processus représente un programme en cours d'exécution.

Le système, pour sa part, maintient un descripteur par processus présent. Nous distinguons deux entités :

- descripteur de processus (DP) : un DP contient toutes les informations relatives à un processus en cours d'exécution ;
- identificateur de processus (IP) : un IP permet de référencer un processus. À partir d'un IP, le système doit pouvoir accéder au DP correspondant.

1.3 Mémoire

Tout processus possède un espace d'adressage virtuel contenant :

- le code qu'il exécute ;
- les données sur lesquelles il agit ;
- la pile qu'il utilise.

L'utilisateur voit son espace d'adressage comme un ensemble de segments (code, données, pile).

Le système gère la mémoire associée à un processus en la décomposant en régions mémoire. Une région est caractérisée par :

- l'adresse virtuelle de son premier octet ;
- sa taille ;
- ses attributs (protection, mode d'utilisation, objets contenus).

2 Erreurs système

Tout appel système peut retourner un code d'erreur au processus appelant. Nous listons ici les différents codes d'erreur :

`E2BIG`	Liste d'arguments trop longue
`EACCESS`	Accès interdit
`EAGAIN`	Ressource temporairement indisponible
`EBADF`	Descripteur d'entrée-sortie invalide
`EBUSY`	Ressource occupée
`ECHILD`	Aucun processus fils
`EDEADLK`	Situation d'interblocage détectée
`EDOM`	Erreur de domaine
`EDQUOT`	Quota disque dépassé
`EEXIST`	Nom de fichier déjà existant
`EFAULT`	Adresse invalide
`EFBIG`	Taille de fichier trop importante
`EINTR`	Appel système interrompu
`EINVAL`	Argument invalide
`EIO`	Erreur d'entrée-sortie
`EISDIR`	Nom de fichier autre que répertoire requis
`ELOOP`	Trop de liens symboliques
`EMFILE`	Trop de fichiers ouverts par un processus
`EMLINK`	Trop de liens
`ENAMETOOLONG`	Nom de fichier trop long
`ENFILE`	Trop de fichiers ouverts dans le système
`ENODEV`	Opération non supportée sur un périphérique
`ENOENT`	Nom de fichier invalide
`ENOEXEC`	Format de programme exécutable invalide
`ENOLCK`	Plus de verrous disponibles
`ENOMEM`	Allocation de mémoire impossible
`ENOSPC`	Système de fichiers saturé
`ENOSYS`	Appel système non implémenté
`ENOTBLK`	Périphérique en mode bloc requis
`ENOTDIR`	Nom de répertoire requis
`ENOTEMPTY`	Répertoire non vide
`ENOTTY`	Requête d'entrée-sortie incorrecte pour un périphérique
`ENXIO`	Périphérique invalide
`EPERM`	Droits d'accès incorrects
`EPIPE`	Ecriture dans un tube sans lecteur
`EPROCLIM`	Trop de processus
`ERANGE`	Résultat trop important
`EROFS`	Système de fichiers monté en lecture seule
`ESPIPE`	Tentative de positionnement sur un tube
`ESRCH`	Numéro de processus invalide
`ETXTBUSY`	Fichier exécutable en cours d'utilisation

`EUSERS`	Trop d'utilisateurs
`EXDEV`	Lien entre plusieurs systèmes de fichiers

3 Appels système

Nous présentons, dans cette section, une liste minimale des appels système implémentés dans Masix. Pour chaque appel système, nous précisons :

- son prototype ;
- sa fonction ;
- ses paramètres ;
- les codes d'erreur qu'il peut retourner ;
- son algorithme.

L'algorithme associé à chaque appel système contient une suite d'actions. Afin de ne pas alourdir le pseudo-code présenté, nous considérons que, si une action échoue, les suivantes ne sont pas exécutées et une erreur est retournée au processus appelant.

3.1 Manipulation des fichiers

`int`	`access`	`(const char * path, int mode);`
Fonction :	Test d'accès à un fichier	
Paramètres :	path (in)	nom du fichier
	mode (in)	mode d'accès désiré
Erreurs :	`EACCESS ENAMETOOLONG ENOTDIR ENOENT EROFS`	
Description :	conversion de path en un IVF	
	lecture des attributs liés à l'IVF	
	obtention de l'identificateur d'utilisateur lié au processus	
	comparaison des droits demandés avec les droits de l'utilisateur	

`int`	`chdir`	`(const char * path);`
Fonction :	Changement de répertoire courant	
Paramètre :	path (in)	nom du répertoire
Erreurs :	`EACCESS ENAMETOOLONG ENOTDIR ENOENT`	
Description :	conversion de path en un IVF	
	obtention de l'identificateur d'utilisateur lié au processus	
	vérification que l'IVF correspond à un répertoire	
	vérification des droits de l'utilisateur sur l'IVF	
	mémorisation de l'IVF comme répertoire courant du processus	

`int` `chmod` `(const char * path, mode_t mode);`
Fonction : Modification de droits d'accès sur un fichier
Paramètre : path (in) nom du fichier
mode (in) droits d'accès
Erreurs : `EACCESS ENAMETOOLONG ENOTDIR ENOENT EPERM EROFS`
Description : conversion de path en un IVF
obtention de l'identificateur d'utilisateur lié au processus
vérification de l'appartenance du fichier à l'utilisateur
modification des attributs de l'IVF

`int` `chown` `(const char *path, uid_t owner, gid_t group);`
Fonction : Modification de l'utilisateur propriétaire d'un fichier
Paramètres : path (in) nom du fichier
owner (in) numéro de propriétaire à positionner
group (in) numéro de groupe à positionner
Erreurs : `EACCES EIO ENAMETOOLONG ENOENT ENOTDIR EPERM EROFS`
Description : conversion de path en un IVF
obtention de l'identification de l'utilisateur lié au processus appelant
lecture des attributs liés a l'IVF
vérification que l'utilisateur est le super utilisateur ou que le fichier appartient à l'utilisateur et que le propriétaire n'est pas modifié
modification des attributs liés a l'IVF

`int` `chroot` `(const char *dirname)`
Fonction : Changement de répertoire racine
Paramètres : dirname (in) nom du répertoire
Erreurs : `EACCES EIO ENAMETOOLONG ENOENT ENOTDIR EPERM`
Description : conversion du path en un IVF
obtention de l'identification de l'utilisateur lié au processus appelant
vérification que l'IVF correspond à un répertoire
vérification que l'utilisateur est le super utilisateur
mémorisation de cet IVF comme repertoire racine pour le processus

`int` `link` `(const char * name1, const char * name2);`
Fonction : Création d'un lien sur un fichier
Paramètres : name1 (in) nom du fichier
name2 (in) nom du lien à créer
Erreurs : `EACCES EEXIST EIO EMLINK ENAMETOOLONG ENOENT`
`ENOSPC ENOTDIR EPERM EROFS EXDEV`
Description : conversion de name1 en un IVF ivf1
conversion du répertoire père de name2 en un IVF ivf2
obtention de l'identification de l'utilisateur lié au processus appelant
vérification des droits d'écriture de l'utilisateur sur ivf2
création d'une entrée dans le répertoire caractérisé par ivf2

`int`	`lstat (const char * path, struct stat * buf);`	
Fonction :	Informations sur un fichier	
Paramètres :	path (in)	nom du fichier
	buf (out)	tampon mémoire
Erreurs :	`EACCES EIO ENAMETOOLONG ENOENT ENOTDIR`	
Description :	conversion de path en un IVF sans suivre les liens symboliques lecture des attributs liés a l'IVF renvoyer ces attributs dans la structure buf	

`int`	`mkdir`	`(const char * path, mode_t mode);`
Fonction :	Création d'un répertoire	
Paramètres :	path (in)	nom du répertoire
	mode (in)	droits d'accès à positionner
Erreurs :	`EACCES EEXIST EIO EMLINK ENAMETOOLONG ENOENT ENOTDIR EROFS`	
Description :	conversion du répertoire père de path en un IVF obtention de l'identification de l'utilisateur lié au processus appelant vérification des droits d'écriture de l'utilisateur sur l'IVF allocation et initialisation d'un DVF répertoire création d'une entrée dans le répertoire caractérisé par l'IVF	

`int`	`mkfifo`	`(const char * path, mode_t mode);`
Fonction :	Création d'un fichier FIFO	
Paramètres :	path (in)	nom du fichier
	mode (in)	droits d'accès à positionner
Erreurs :	`EACCES EEXIST ENAMETOOLONG ENOENT ENOSPC ENOTDIR EROFS`	
Description :	conversion du répertoire père de path en un IVF obtention de l'identification de l'utilisateur lié au processus appelant vérification des droits d'écriture de l'utilisateur sur l'IVF allocation et initialisation d'un DVF FIFO création d'une entrée dans le répertoire caractérisé par l'IVF	

`int`	`mknod`	`(const char * path, mode_t mode, dev_t dev);`
Fonction :	Création d'un fichier d'accès à un périphérique	
Paramètres :	path (in)	nom du fichier
	mode (in)	droits d'accès à positionner
	dev (in)	numero du péripherique
Erreurs :	`EACCES EEXIST EIO EISDIR ENAMETOOLONG ENOENT` `ENOSPC ENOTDIR EPERM EROFS`	
Description :	conversion du repertoire père de path en un IVF obtention de l'identification de l'utilisateur lié au processus appelant vérification que l'utilisateur est le super-utilisateur allocation et initialisation d'un DVF spécial création d'une entrée dans le répertoire caractérisé par l'IVF	

`int`	`mount`	`(int type, const char * dirname, int flags,`
		`caddr_t data);`
Fonction :	Montage d'un système de fichiers	
Paramètres :	type (in)	type du système de fichiers
	dirname (in)	répertoire de montage
	flags (in)	options de montage
	data (in)	options spécifiques au type de système de fichiers
Erreurs :	`EACCES EBUSY EIO ENAMETOOLONG ENODEV ENOENT ENOTDIR EPERM`	
Description :	obtention de l'identification de l'utilisateur lié au processus appelant	
	vérification que l'utilisateur est le super utilisateur	
	conversion de dirname en un IVF	
	montage du SF caractérisé par data et type sur cet IVF	

`long`	`pathconf`	`(const char * path, int name);`
Fonction :	Limites associées à un fichier	
Paramètres :	path (in)	nom du fichier
	name (in)	limite désirée
Erreurs : `EACCES EINVAL ENAMETOOLONG ENOENT ENOTDIR`		
Description :	conversion de path en IVF	
	lecture des limites liées a cet IVF	

`int`	`readlink`	`(const char * path, char * buf, int bufsize);`
Fonction :	Lecture de la valeur d'un lien symbolique	
Paramètres :	path (in)	nom du lien
	buf (out)	tampon mémoire
	bufsize (in)	taille du tampon
Erreurs :	`EACCES EINVAL EIO ENAMETOOLONG ENOENT`	
Description :	conversion de path en un IVF	
	lecture des attributs liés à l'IVF	
	verification que cet IVF correspond bien à un lien symbolique	
	lecture du fichier correspondant à cet IVF	
	retourner le contenu de ce fichier dans buf	

`int`	`rename`	`(const char * name1, const char * name2);`
Fonction :	Changement de nom d'un fichier	
Paramètres :	name1 (in)	ancien nom
	name2 (in)	nouveau nom
Erreurs :	`EACCES EBUSY EINVAL EIO EISDIR ENAMETOOLONG ENOENT ENOSPC ENOTDIR ENOTEMPTY EROFS EXDEV`	
Description :	conversion de name1 en un IVF ivf1 conversion du répertoire père de name1 en un IVF ivf2 conversion du répertoire père de name2 en un IVF ivf3 obtention de l'identification de l'utilisateur lié au processus appelant vérification des droits d'écriture de l'utilisateur sur ivf2 vérification des droits d'écriture de l'utilisateur sur ivf3 création d'une entrée dans le répertoire caractérisé par ivf3 suppression d'une entrée dans le répertoire correspondant à ivf2	

`int`	`rmdir`	`(const char * path);`
Fonction :	Supression d'un répertoire	
Paramètres :	path (in)	nom du répertoire
Erreurs :	`EACCES EBUSY EINVAL EIO ENAMETOOLONG ENOENT ENOTDIR ENOTEMPTY EROFS`	
Description :	conversion de path en un IVF ivf1 conversion du père de path en un IVF ivf2 obtention de l'identification de l'utilisateur lié au processus appelant vérification des droits d'écriture de l'utilisateur sur ivf2 vérification que ivf1 correspond à un répertoire vérification que ivf1 est vide désallocation de ivf1 suppression de l'entrée de répertoire correspondant à ivf2	

`int`	`stat`	`(const char * path, struct stat * buf);`
Fonction :	Informations sur un fichier	
Paramètres :	path (in)	nom du fichier
	buf (out)	tampon mémoire
Erreurs :	`EACCES EIO ENAMETOOLONG ENOENT ENOTDIR`	
Description :	conversion de path en un IVF lecture des attributs liés a l'IVF renvoyer ces attributs dans la structure buf	

`int`	`statfs`	`(const char * path, struct statfs * buf);`
Fonction :	Informations sur un système de fichiers	
Paramètres :	path (in)	nom d'un fichier
	buf (out)	tampon mémoire
Erreurs :	`EACCES EIO ENAMETOOLONG ENOENT ENOTDIR`	
Description :	conversion de path en un IVF lecture des attributs liés au SF lié à l'IVF	

`int` `symlink` `(const char * name1, const char * name2);`
Fonction : Création d'un lien symbolique
Paramètres : name1 (in) nom du fichier cible
name2 (in) nom du lien
Erreurs : `EACCES EEXIST EIO ENAMETOOLONG ENOENT ENOSPC ENOTDIR EROFS`
Description : conversion du père de name2 en un IVF
obtention de l'identification de l'utilisateur lié au processus appelant
vérification des droits d'écriture de l'utilisateur sur l'IVF
allocation et initialisation d'un nouveau DVF lien symbolique
création d'une nouvelle entrée dans le répertoire correspondant à l'IVF
écriture de name1 dans l'IVF alloué plus haut

`int` `sync` `(void);`
Fonction : Re-écriture des blocs modifiés
Paramètres : aucun
Erreurs : aucune
Description : ré-écriture des blocs modifiés de tous les FS

`int` `truncate` `(const char * path, off_t length);`
Fonction : Modification de la taille d'un fichier
Paramètres : path (in) nom du fichier
length (in) taille à positionner
Erreurs : `EACCES EIO EISDIR ENAMETOOLONG ENOENT ENOTDIR EROFS`
Description : conversion de path en un IVF
obtention de l'identification de l'utilisateur lié au processus appelant
vérification des droits de l'utilisateur sur cet IVF
modification des attributs correspondant à cet IVF
désallocation d'espace si length < taille initiale du fichier

`mode_t` `umask` `(mode_t numask);`
Fonction : Modification des droits d'accès en création de fichier
Paramètres : numask (in) droits d'accès à positionner
Erreurs : aucune
Description : mémorisation de numask comme droits d'accès utilisés lors des créations

`int` `unlink` `(const char * path);`
Fonction : Suppression d'un lien
Paramètres : path (in) nom du lien
Erreurs : `EACCES EBUSY EINVAL EIO ENAMETOOLONG ENOENT ENOTDIR EPERM EROFS`
Description : conversion du père de path en un IVF ivf1
conversion de path en un IVF ivf2
obtention de l'identification de l'utilisateur lié au processus appelant
vérification des droits d'écriture de l'utilisateur sur ivf1
suppression de l'entrée dans le répertoire correspondant à ivf1
désallocation de ivf2

`int unmount (const char * dir, int flags);`

Fonction : Démontage d'un système de fichiers

Paramètres : dir (in) nom du répertoire de montage
flags (in) options de démontage

Erreurs : `EACCES EBUSY EINVAL EIO ENAMETOOLONG ENOENT ENOTDIR EPERM`

Description : obtention de l'identification de l'utilisateur lié au processus appelant
verification que l'utilisateur est le super utilisateur
conversion de dirname en un IVF
démontage du SF monté sur cet IVF

`int utimes (const char * file, const struct timeval * times);`

Fonction : Modification des dates d'un fichier

Paramètres : file (in) nom du fichier
times (in) dates à positionner

Erreurs : `EACCES EIO ENOENT ENOTDIR EPERM EROFS`

Description : conversion de path en IVF
obtention de l'identification de l'utilisateur lié au processus appelant
vérification des droits de l'utilisateur sur cet IVF
modification des attributs liés a cet IVF

3.2 Entrees-Sorties

`int close (int fd);`

Fonction : Fermeture d'un fichier

Paramètres : fd (in) numéro du descripteur a fermer

Erreurs : `EBADF EINTR`

Description : conversion du DES fd en DESI
Si le DESG correspond à un IVF
    signaler que l'IVF n'est plus utilisé
désallocation du DES fd
décrémentation du nombre d'utilisations du DESI
Si nul
    conversion du DESI en DESG
    désallocation du DESI
    décrémentation du nombre d'utilisations du DESG
    Si nul
        désallocation du DESG

`int dup (int oldd);`

Fonction : Duplication de descripteur

Paramètres : oldd (in) numéro de descripteur

Erreurs : `EBADF EMFILE`

Description : conversion du DES oldd en DESI
allocation d'un nouveau DES (le plus petit disponible) correspondant au DESI
incrémentation du nombre d'utilisations du DESI

`int dup2 (int oldd, int newd);`

Fonction : Duplication de descripteur
Paramètres : oldd (in) numéro de descripteur
newd (in) numéro du nouveau descripteur
Erreurs : `EBADF EMFILE`
Description : conversion du DES oldd en un DESI desi1
Si DES newd utilisé
  conversion du DES newd en un DESI desi2
  désallocation du DES newd
  décrémentation du nombre d'utilisation de desi2
  Si nul
    conversion de desi2 en un DESG desg2
    désallocation de desi2
    décrémentation du nombre d'utilisations de desg2
    Si nul
      désallocation de desg2
allocation d'un nouveau DES newd correspondant à desi1

`int fchmod (int fd, mode_t mode);`

Fonction : Modification des droits d'accès d'un fichier
Paramètres : fd (in) numéro de descripteur
mode (in) droits d'accès à positionner
Erreurs : `EBADF EIO EPERM EROFS`
Description : conversion du DES fd en un DESG
vérification que le DESG correspond à un fichier
conversion du DESG en un IVF
obtention de l'identification de l'utilisateur lié au processus appelant
vérification de l'appartenance du fichier à l'utilisateur
modification des attributs liés a cet IVF

`int fchown (int fd, uid_t owner, gid_t group);`

Fonction : Modification du propriétaire d'un fichier
Paramètres : fd (in) numéro de descripteur
owner (in) numéro d'utilisateur à positionner
group (in) numéro de groupe à positionner
Erreurs : `EBADF EINVAL EIO EPERM EROFS`
Description : conversion du DES fd en un DESG
vérification que le DESG correspond à un fichier
conversion du DESG en un IVF
obtention de l'identification de l'utilisateur lié au processus appelant
lecture des attributs liés a l'IVF
vérification que l'utilisateur est le super-utilisateur ou que le fichier appartient à l'utilisateur et que le propriétaire n'est pas modifié
modification des attributs liés à l'IVF

`int`	`fcntl`	`(int fd, int cmd, int arg);`
Fonction :	Opération de controle sur fichier	
Paramètres :	fd (in)	numéro de descripteur
	cmd (in)	requête de controle
	arg (in)	argument de la requête
Erreurs :	`EACCES EBADF EDEADLK EINTR EINVAL EMFILE ENOLCK`	
Description :	Si cmd = F_DUPFD	
		conversion du DES fd en DESI
		allocation d'un nouveau DES (le plus petit disponible)
		correspondant au DESI
	Sinon	
		obtention ou modification d'attributs liés au DES fd

`int`	`fpathconf`	`(int fd, int name);`
Fonction :	Limites associées a un fichier	
Paramètres :	fd (in)	numéro de descripteur
	name (in)	limite désirée
Erreurs :	`EBADF EINVAL`	
Description :	conversion du DES fd en un DESG	
	vérification que le DESG correspond à un fichier	
	conversion du DESG en un IVF	
	lecture des limites liées a cet IVF	

`int`	`fstat`	`(int fd, struct stat *buf);`
Fonction :	Informations sur un fichier	
Paramètres :	fd (in)	numéro de descripteur
	buf (out)	tampon mémoire
Erreurs :	`EBADF EIO`	
Description :	conversion du DES fd en un DESG	
	vérification que le DESG correspond à un fichier	
	conversion du DESG en un IVF	
	lecture des attributs liés à l'IVF	
	renvoyer ces attributs dans la structure buf	

`int`	`fsync`	`(int fd);`
Fonction :	Re-écriture des blocs modifiés d'un fichier	
Paramètres :	fd (in)	numéro de descripteur
Erreurs :	`EBADF EIO`	
Description :	conversion du DES fd en un DESG	
	vérification que le DESG correspond à un fichier	
	conversion du DESG en un IVF	
	re-écriture de tous les blocs modifiés correspondant à l'IVF	

`int`	`ftruncate (int fd, off_t length);`	
Fonction :	Modification de la taille d'un fichier	
Paramètres :	fd (in)	numéro de descripteur
	length (in)	taille désirée
Erreurs :	`EBADF EIO`	
Description :	conversion du DES fd en un DESG vérification que le DESG correspond à un fichier conversion du DESG en un IVF vérification de la cohérence de l'opération avec le mode d'ouverture modification des attributs liés a l'IVF désallocation d'espace si length < taille initiale du fichier	

`int`	`getdirentries(int fd, char * buf, int nbytes, long * basep);`	
Fonction :	Lecture d'entrées de répertoire	
Paramètres :	fd (in)	numéro de descripteur
	buf (out)	tampon mémoire
	nbytes (in)	taille du tampon
	basep (out)	position du dernier bloc disque lu
Erreurs :	`EBADF EINTR EIO`	
Description :	conversion du DES fd en un DESG vérification que le DESG correspond à un fichier conversion du DESG en un IVF vérification que l'IVF correspond à un répertoire lecture du répertoire correspondant à l'IVF et renvoi de son contenu dans buf	

`int`	`getdtablesize(void);`
Fonction :	Taille de la table des descripteurs
Paramètres :	aucun
Erreurs :	aucune
Description :	obtention de l'identification du processus appelant, retourne le nombre maximum de DES pour ce processus

`int`	`ioctl (int fd, unsigned long request, char *argp);`	
Fonction :	Opération de controle	
Paramètres :	fd (in)	numéro de descripteur
	request (in)	requête désirée
	argp (in/out)	arguments de la requête
Erreurs :	`EBADF EINVAL ENOTTY`	
Description :	conversion du DES fd en DESG vérification que le DESG correspond à un périphérique en mode caractère lancement de la requête sur le périphérique	

`off_t lseek (int fd, off_t offset, int whence);`

Fonction : Positionnement dans un fichier

Paramètres :
fd (in) numéro de descripteur
offset (in) déplacement dans le fichier
whence (in) position de départ

Erreurs : `EBADF EINVAL ESPIPE`

Description :
obtention du DESI correspondant au DES fd
modification du pointeur courant associé au DESI

`int open (const char * path, int flags, mode_t mode);`

Fonction : Ouverture d'un fichier

Paramètres :
path (in) nom du fichier
flags (in) mode d'ouverture
mode (in) droits d'accès a positionner

Erreurs : `EACCES EEXIST EINTR EIO EISDIR EMFILE ENAMETOOLONG ENFILE ENOENT ENOSPC ENOTDIR ENXIO EROFS`

Description :
obtention de l'identification de l'utilisateur lié au processus appelant
Si création demandée
    conversion du répertoire père de path en un IVF ivf2
    vérification des droits d'écriture de l'utilisateur sur ivf2
    conversion de path en un IVF ivf1 (sans arret si erreur)
    Si ivf1 existe
        vérification que O_EXCL n'est pas demandé
        désallocation de l'espace associé a ivf1
    Sinon
        allocation d'un DVF réferencé par un IVF ivf1
Sinon
    conversion de path en un IVF ivf1
    vérification des droits de l'utilisateur sur ivf1 par rapport à mode
Si ivf1 correspond à un terminal
    Si l'identificateur du processus est égal à son identificateur de groupe de processus
        recherche d'un processus ayant ce terminal de contrôle
        Si non trouvé
            mémorisation de l'identificateur de terminal de contrôle associé au processus
signaler que ivf1 est en cours d'utilisation
recherche d'un DESG correspondant à ivf1
Si trouvé
    incrémentation du nombre d'utilisations du DESG
Sinon
allocation d'un DESG correspondant à ivf1
allocation d'un DESG correspondant au DESG
allocation d'un DES correspondant a ce DESI

`ssize_t`	`read`	`(int fd, char * buf, size_t bufsize);`
Fonction :	Lecture de données	
Paramètres :	fd (in)	numéro de descripteur
	buf (out)	tampon mémoire
	bufsize (in)	taille du tampon
Erreurs :	`EAGAIN EBADF EINTR EIO EISDIR`	
Description :	conversion du DES fd en DESI et en DESG vérification de la conformité de l'opération avec le mode d'ouverture du DESI lecture de données depuis le DESG Si fichier ou périphérique en mode bloc mise à jour du pointeur courant associé au DESI	

`ssize_t`	`write`	`(int fd, const char * buf, size_t bufsize);`
Fonction :	Ecriture de données dans un fichier	
Paramètres :	fd (in)	numéro de descripteur
	buf (in)	tampon mémoire
	bufsize (in)	taille du tampon
Erreurs :	`EBADF EFBIG EINTR EIO ENOSPC EPIPE`	
Description :	conversion du DES fg en DESI et DESG vérification de la conformité de l'operation avec le mode d'ouverture du DESI écriture de données dans le DESG Si fichier ou périphérique en mode bloc mise à jour du pointeur courant associé au DESI	

3.3 Processus

`int`	`execve`	`(const char *path, const * char *argv,` `const * char *envp);`
Fonction :	Exécution d'un programme	
Paramètres :	path (in)	nom du fichier
	argv (in)	arguments du programme
	envp (in)	variables d'environnement
Erreurs :	`E2BIG EACCES EIO ENAMETOOLONG ENOENT ENOEXEC ENOMEM ENOTDIR`	

Description :	conversion de path en IVF
	obtention de l'identification de l'utilisateur lié au processus appelant
	vérification des droits d'exécution sur le fichier correspondant à l'IVF
	lecture de l'en-tête du fichier
	réallocation de la mémoire associée au processus
	Pour tout DES alloué
	    Si DES marque close-on-exec
	        conversion du DES en DESI
	        désallocation du DES
	        décrémentation du nombre d'utilisations du DESI
	        Si nul
	            conversion du DESI en DESG	

désallocation du DESI
décrémentation du nombre
d'utilisations du DESG
Si nul
désallocation du DESG
modification des caractéristiques du processus (utilisateur, groupe, ...)
initialisation de la mémoire allouée avec le contenu du fichier correspondant a l'IVF

`void _exit (int status);`

Fonction : Terminaison de processus
Paramètres : status (in) code de retour
Erreurs : aucune
Description : obtention de l'identificateur du processus appelant
Pour tout DES alloué
conversion du DES en DESI
désallocation du DES
décrémentation du nombre d'utilisations du DESI
Si nul
conversion du DESI en DESG
désallocation du DESI
décrémentation du nombre d'utilisations du DESG
Si nul
désallocation du DESG
désallocation de la mémoire allouée à l'appelant
passage du processus dans l'état zombie
attachement de tous les processus fils au processus numéro 1 (init)
réveil du processus père si en attente de terminaison d'un fils

`int fork (void);`

Fonction : Duplication de processus
Paramètres : aucun
Erreurs : `EAGAIN ENOMEM`
Description : allocation d'un descripteur de processus
Pour tout DES alloué
conversion du DES en DESI
incrémentation du nombre d'utilisations du DESI
duplication de tous les DES
duplication de la mémoire allouée
mise a 0 du nombre de ressources utilisées
duplication des caractéristiques du processus
passage du processus créé à l'etat prèt

`gid_t getegid (void);`
Fonction : Obtention d'identificateur de groupe effectif
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention de l'identification du groupe effectif contenu dans ce DP

`uid_t geteuid (void);`
Fonction : Obtention d'identificateur d'utilisateur effectif
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention de l'identification de l'utilisateur effectif contenu dans ce DP

`gid_t getgid (void);`
Fonction : Obtention d'identificateur de groupe réel
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention de l'identification du groupe réel d'utilisateur contenu dans ce DP

`int getgroups (int gidsetlen, gid_t * gidset);`
Fonction : Obtention des identificateurs de groupes
Paramètres : gidsetlen (in) nombre de groupes demandes
gidset (out) tampon mémoire
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention des identificateurs des groupes secondaires d'utilisateur contenu dans ce DP

`char * getlogin (void);`
Fonction : Obtention du nom d'utilisateur
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention du nom d'utilisateur contenu dans ce DP

`pid_t getpid (void);`
Fonction : Obtention du numéro de processus
Paramètres : aucun
Erreurs : aucune
Description : obtention de l'identificateur du processus appelant

`int getpgrp (void);`
Fonction : Obtention du numéro de groupe de processus
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention de l'identification du groupe de processus contenu dans ce DP

`pid_t getppid (void);`
Fonction : Obtention du numéro du processus père
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention de l'identificateur du processus père contenu dans ce DP

`int getrlimit (int resource, struct rlimit * rlp);`
Fonction : Obtention des limites
Paramètres : resource (in) limite désirée
rlp (out) valeur de la limite
Erreurs : `EINVAL`
Description : obtention du descripteur du processus appelant
obtention des limites sur les ressources contenues dans ce DP
retourner la limite demandee

`uid_t getuid (void);`
Fonction : Obtention de l'identificateur d'utilisateur réel
Paramètres : aucun
Erreurs : aucune
Description : obtention du descripteur du processus appelant
obtention de l'identification de l'utilisateur réel contenu dans ce DP

`int setgroups (int ngroups, const gid_t * gidset);`
Fonction : Positionnement des groupes
Paramètres : ngroups (in) nombre de groupes
gidset (in) groupes à positionner
Erreurs : `EPERM`
Description : obtention du descripteur du processus appelant
obtention de l'identification de l'utilisateur lié au processus appelant
vérification que l'utilisateur est le super-utilisateur
mémorisation des identificateurs de groupes d'utilisateur dans le DP

`int`	`setlogin`	`(const char * name);`
Fonction :	Positionnement du nom d'utilisateur	
Paramètres :	name (in)	nom d'utilisateur
Erreurs :	`EPERM`	
Description :	obtention du descripteur du processus appelant obtention de l'identification de l'utilisateur lié au processus appelant vérification que l'utilisateur est le super-utilisateur mémorisation du nom d'utilisateur dans le DP	

`int`	`setpgid`	`(pid_t pid, pid_t pgrp);`
Fonction :	Positionnement du groupe de processus	
Paramètres :	pid (in)	numéro de processus
	pgrp (in)	numéro de groupe de processus
Erreurs :	`EACCES EINVAL EPERM ESRCH`	
Description :	obtention du descripteur du processus appelant mémorisation du groupe de processus dans ce DP	

`int`	`setregid`	`(gid_t rgid, gid_t egid);`
Fonction :	Positionnement des identificateurs de groupes d'utilisateur réel et effectif	
Paramètres :	rgid (in)	numéro de groupe réel
	egid (in)	numéro de groupe effectif
Erreurs :	`EINVAL EPERM`	
Description :	obtention du descripteur du processus appelant obtention de l'identification de l'utilisateur lié au processus appelant vérification que l'utilisateur est le super-utilisateur ou que les groupes specifiés sont valides mémorisation des identificateurs de groupes dans le DP	

`int`	`setreuid`	`(uid_t ruid, uid_t euid);`
Fonction :	Positionnement des identificateurs d'utilisateurs réel et effectif	
Paramètres :	ruid (in)	numéro d'utilisateur réel
	euid (in)	numéro d'utilisateur effectif
Erreurs :	`EINVAL EPERM`	
Description :	obtention du descripteur du processus appelant obtention de l'identification de l'utilisateur lié au processus appelant vérification que l'utilisateur est le super-utilisateur ou que les identificateurs d'utilisateur specifiés sont valides mémorisation des identificateurs d'utilisateurs dans le DP	

`int`	`setrlimit`	`(int resource, struct rlimit * rlp);`
Fonction :	Positionnement des limites	
Paramètres :	resources (in)	ressource désirée
	rlp (in)	limite à positionner
Erreurs :	`EINVAL EPERM`	
Description :	obtention du descripteur du processus appelant mémorisation des limites sur les ressources allouées au processus dans le DP	

```
int           wait          (int * status);
int           wait3         (int * status, int options, struct rusage *rusage);
```

Fonction : Attente de terminaison de processus

Paramètres :
status (out) code de retour
options (in) options d'attente
rusage (out) ressources consommées

Erreurs : `ECHILD EINTR`

Description :
obtention de l'identificateur du processus appelant
Si aucun processus fils terminé
    mémorisation que le processus est en attente d'un processus fils
    suspension du processus
obtention des informations sur l'exécution du processus fils terminé
désallocation du descripteur du processus fils

```
int           wait4         (pid_t wpid, int * status, int options,
                             struct rusage * rusage);
int           waitpid       (pid_t wpid, int *status, int options);
```

Fonction : Attente de terminaison de processus

Paramètres :
wpid (in) numéro de processus a attendre
status (out) code de retour
options (in) options d'attente
rusage (out) ressources consommées

Erreurs : `ECHILD EINTR EINVAL`

Description :
obtention de l'identificateur du processus appelant
Si le processus fils specifié par wpid n'est pas terminé
    mémorisation que le processus est en attente du processus fils specifié par wpid
    suspension du processus
obtention des informations sur l'exécution du processus fils terminé
désallocation du descripteur du processus fils

3.4 Signaux

```
int           kill          (pid_t pid, int sig);
```

Fonction : Envoi de signal à un processus

Paramètres :
pid (in) numéro de processus
sig (in) numéro de signal

Erreurs : `EINVAL EPERM ESRCH`

Description :
obtention du descripteur du processus appelant
obtention du descripteur du processus cible
vérification que le processus appelant possède les droits nécessaires à l'envoi du signal
mémorisation du signal dans le DP cible
restauration éventuelle du masque des signaux sauvegardé
passage éventuel du processus cible à l'état prêt

`int`	`sigaction`	`(int sig, const struct sigaction * act,`
		`struct sigaction * oact);`
Fonction :	Modification d'actions associées à un signal	
Paramètres :	sig (in)	numéro de signal
	act (in)	action à positionner
	oact (out)	action anciennement associée au signal
Erreurs :	`EINVAL`	
Description :	obtention du descripteur du processus appelant	
	obtention de l'action liée au signal sig	
	memorisation de l'action liée au signal sig	

`int`	`sigpending`	`(sigset_t *set);`
Fonction :	Signaux reçus	
Paramètres :	set (out)	liste des signaux reçus
Erreurs :	aucune	
Description :	obtention du descripteur du processus appelant	
	obtention des signaux en attente pour le processus	

`int`	`sigprocmask`	`(int how, const sigset_t * set, sigset_t * oset);`
Fonction :	Modification du masque des signaux	
Paramètres :	how (in)	mode de modification
	set (in)	masque à positionner
	oset (out)	ancien masque
Erreurs :	`EINVAL`	
Description :	obtention du descripteur du processus appelant	
	obtention du masque lié aux signaux	
	modification éventuelle de ce masque (selon la valeur de how)	

`int`	`sigsuspend`	`(const sigset_t * sigmask);`
Fonction :	Attente d'un signal	
Paramètres :	sigmask (in)	masque à positionner
Erreurs :	`EINTR`	
Description :	obtention de l'identificateur du processus appelant	
	sauvegarde du masque lié aux signaux	
	mémorisation du masque lié aux signaux	
	mémorisation du fait que le processus attend un signal	
	suspension du processus	

3.5 Mémoire

`char *`	`brk`	`(const char * addr);`
Fonction :	Modification de la mémoire allouee	
Paramètres :	addr (in)	adresse de fin des données
Erreurs :	`ENOMEM`	
Description :	obtention du descripteur du processus appelant vérification que la requête ne dépasse pas la limite associée au processus modification de la taille de l'espace mémoire alloué au processus	

`int`	`getpagesize`	`(void);`
Fonction :	Obtention de la taille d'une page	
Paramètres :	aucun	
Erreurs :	aucune	
Description :	retourner la taille d'une page	

`char *`	`sbrk`	`(int incr);`
Fonction :	Modification de la mémoire allouée	
Paramètres :	incr (in)	nombre d'octets supplémentaires à allouer
Erreurs :	`ENOMEM`	
Description :	obtention du descripteur du processus appelant vérification que la requête ne dépasse pas la limite associée au processus modification de la taille de l'espace mémoire alloué au processus	

`int`	`swapon`	`(const char * special);`
Fonction :	Ajout d'une zone d'echange pour la mémoire virtuelle	
Paramètres :	special (in)	nom du périphérique utilisé
Erreurs :	`EACESS EBUSY EIO ENAMETOOLONG ENOENT ENOTBLK ENOTDIR ENXIO EPERM`	
Description :	obtention de l'identification de l'utilisateur lié au processus appelant vérification que l'utilisateur est le super-utilisateur conversion de special en un IVF vérification que l'IVF correspond á un fichier ou à un fichier spécial memorisation de l'IVF comme périphérique d'échange	

3.6 Configuration du système

`long`	`sysconf`	`(int name);`
Fonction :	Obtention des caractéristiques du système	
Paramètres :	name (in)	nom de la caractéristique
Erreurs :	`EINVAL`	
Description :	obtention d'une caractéristique du système correspondant à name retourner cette information	

3.7 Administration du système

`long`	`gethostid`	`(void);`
Fonction :	Obtention de l'identificateur de la station	
Paramètres :	aucun	
Erreurs :	aucune	
Description :	obtention de l'identificateur unique lié à la station	

`int`	`gethostname`	`(char * name, int namelen);`
Fonction :	Obtention du nom de la sttion	
Paramètres :	name (out)	tampon mémoire
	namelen (in)	taille du tampon
Erreurs :	aucune	
Description :	obtention du nom de la station	

`int`	`reboot`	`(int howto);`
Fonction :	Arret du système	
Paramètres :	howto (in)	mode d'arrêt
Erreurs :	`EPERM`	
Description :	Réinitialisation du système	

`int`	`sethostid`	`(long hostid);`
Fonction :	Positionnement de l'identificateur de la station	
Paramètres :	hostid (in)	identificateur
Erreurs :	`EPERM`	
Description :	obtention de l'identificateur d'utilisateur lié au processus appelant vérification que l'utilisateur est le super-utilisateur mémorisation de l'identificateur unique lié à la station	

`int`	`sethostname`	`(const char * name, int namelen);`
Fonction :	Positionnement du nom de la station	
Paramètres :	name (in)	nom
	namelen (in)	taille du nom
Erreurs :	`EPERM`	
Description :	obtention de l'identificateur d'utilisateur associé au processus appelant vérification que l'utilisateur est le super-utilisateur mémorisation du nom de la station	

`int`	`uname`	`(struct utsname * name);`
Fonction :	Obtention des informations liées à la station	
Paramètres :	name (out)	Tampon mémoire
Erreurs :	aucune	
Description :	obtention des informations liées à la station	