THINK : vers une architecture de systèmes flexibles

Jean-Philippe Fassino

▶ To cite this version:

Jean-Philippe Fassino. THINK : vers une architecture de systèmes flexibles. Réseaux et télécommunications [cs.NI]. Télécom ParisTech, 2001. Français. NNT : . tel-00005776

HAL Id: tel-00005776
https://pastel.hal.science/tel-00005776

Submitted on 5 Apr 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée et soutenue publiquement par

Jean-Philippe FASSINO

pour obtenir le grade de DOCTEUR
de l'ÉCOLE NATIONALE SUPÉRIEURE DES TÉLÉCOMMUNICATIONS

Spécialité : **Informatique et Télécommunications**

THINK : vers une architecture de systèmes flexibles

Date de soutenance : 11 décembre 2001

Composition du jury :

Président :	Rachid	GUERRAOUI	
Rapporteurs :	Sacha	KRAKOWIAK	
	Gilles	MULLER	
Examinateurs :	Isabelle	DEMEURE	(directeur)
	Jean-Bernard	STEFANI	(directeur)
	Pascal	DÉCHAMBOUX	

Thèse préparée au

laboratoire **Architecture des Systèmes Répartis**
de la **Direction des Techniques Logicielles**
de **France Télécom Recherche & Développement**

Avant-propos

Le travail présenté dans cette thèse a été effectué au sein du département Architecture des Systèmes Répartis de la Direction des Techniques Logicielles (DTL/ASR) de France Télécom Recherche & Développement. Je tiens à remercier cet organisme qui m'a permis d'effecteur mon travail dans les meilleures conditions matérielle, scientifique et technique que l'on puisse espérer.

Je suis particulièrement reconnaissant envers Isabelle DEMEURE, maître de conférence à l'École Nationale Supérieure des Télécommunications, pour avoir accepté de diriger cette thèse malgré la distance nous séparant ainsi que pour ses conseils avisés qui m'ont permis d'avancer vers un résultat meilleur.

Je tiens à exprimer ma vive reconnaissance et ma profonde amitié à Jean-Bernard STEFANI, directeur de recherche à l'INRIA, pour son encadrement tout au long de ces trois années ainsi que pour la confiance qu'il m'a témoigné en me proposant cette thèse. Son rôle, ses critiques constructives et ses encouragements m'ont permis de mener à bien cette thèse et ce manuscrit. Ses nombreux conseils, ses propositions et son goût pour les ordinateurs dignes de ce nom ont été primordiaux dans les résultats obtenus durant ce travail. Bref, je lui dois beaucoup.

J'exprime ma profonde gratitude à Rachid GUERRAOUI, professeur à l'École Polytechnique de Lausanne, pour l'honneur qu'il m'a fait en présidant mon jury de thèse. Je souhaite remercier Sacha KRAKOWIAK, professeur à l'université Joseph Fourier, et Gilles MULLER, chargé de recherche à l'INRIA, pour avoir accepté la lourde charge d'être rapporteur et pour leurs précieuses remarques qui m'ont permis d'améliorer ce manuscrit. Je remercie aussi Pascal DÉCHAMBOUX, ingénieur à France Télécom R&D, d'avoir accepté de faire partie de mon jury et pour les nombreuses discussions que nous avons eu ensemble. Je remercie à nouveau Gilles MULLER pour avoir baptisé mes travaux et aidé à mettre un peu d'ordre dans mes idées. Je remercie encore vivement Sacha KRAKOWIAK qui le premier m'a fait découvrir le monde de la recherche scientifique ainsi que pour les nombreux enseignements que j'en ai retirés.

Mes remerciements vont naturellement à l'ensemble des gens que j'ai côtoyé durant ces trois années et qui m'ont aidé ou tout simplement rendu le déroulement de cette thèse agréable. Je citerai entre autres :

- Christelle LE MEZEC notre assistante et grande organisatrice de la pause café,
- François-Gaël OTTOGALLI pour nos discussions sur tout et surtout sur n'importe quoi,
- Alexandre LEFEBVRE pour sa sympathie et sa gentillesse qui le perdra,
- Christian BAYLE le grand gourou des bidouilles,
- François HORN et Fabien DELPIANO qui m'ont fait comprendre bien des choses,

- Florence GERMAIN pour ses relectures,
- Stephen BRANDALISE qui m'a permis de voir la réalité quotidienne des utilisateurs de l'informatique et surtout les progrès qu'ils restent aux informaticiens à accomplir.

Je tiens aussi à remercier Kathleen MILSTED et toute l'équipe du département DTL/ASR qui m'offrent la possibilité et les moyens de poursuivre ces travaux de recherche et de développement pour le meilleur et pour le pire.

Pour finir, je remercie tous ceux qui, en dehors du travail, m'ont accompagné et soutenu durant ces trois années. Je réserve une pensée particulière pour Sandrine qui partage mes rêves et mes motivations et pour Onyx qui tous les soirs me change les idées et me ramène les pieds sur terre en m'emmenant promener.

Table des matières

1 Introduction **1**
1.1 Contexte et objectif de la thèse . . . 1
1.2 Pourquoi une nouvelle architecture ? . . . 2
1.2.1 Fonction et service d'un système d'exploitation . . . 2
1.2.2 Évolution des systèmes d'exploitation . . . 3
1.2.3 Comment aller plus loin ? . . . 5
1.3 Plan de la thèse . . . 6

2 Principes et techniques de construction des systèmes **9**
2.1 Définitions . . . 9
2.2 Abstraction et virtualisation des ressources . . . 10
2.2.1 Ressources matérielles . . . 11
2.2.2 Ressources systèmes . . . 14
2.2.3 Processus . . . 17
2.2.4 Machines virtuelles . . . 19
2.3 Concepts d'utilisation des ressources . . . 21
2.3.1 Désignation et liaison . . . 21
2.3.2 Allocation . . . 22
2.3.3 Synchronisation . . . 23
2.4 Principes d'organisation des systèmes . . . 24
2.4.1 Structuration en couches . . . 25
2.4.2 Structuration orientée objets . . . 27
2.4.3 Structuration orientée événements . . . 28
2.4.4 Structuration par référentiel . . . 28
2.4.5 Structuration par flot de données . . . 28
2.4.6 Langages de description d'architecture . . . 29
2.5 Interposition . . . 30
2.5.1 Proxy . . . 30
2.6 Modèle de communication dans les systèmes . . . 34
2.6.1 Modèle de communication par variable partagée . . . 35
2.6.2 Modèle de communication par appel de méthode . . . 36
2.6.3 Modèle de communication par message . . . 37
2.6.4 Modèle de communication par événements . . . 38

2.7 Noyaux de systèmes d'exploitation . . . 39
2.7.1 Systèmes à domaine de protection unique . . . 40
2.7.2 Noyaux monolithiques . . . 41
2.7.3 Micronoyaux . . . 42
2.7.4 Noyaux extensibles . . . 44
2.7.5 Exonoyaux . . . 45
2.8 Infrastructures logicielles réparties . . . 46
2.8.1 Intergiciels . . . 46
2.8.2 Systèmes d'exploitation répartis . . . 46
2.9 Conclusion . . . 47

3 Architecture logicielle THINK 49
3.1 Objectifs de l'architecture . . . 49
3.1.1 Ce qui n'est pas abordé . . . 50
3.2 Concepts mis en œuvre . . . 50
3.2.1 Concepts de base . . . 50
3.2.2 Liaison flexible . . . 52
3.2.3 Domaine . . . 58
3.2.4 Ressource . . . 62
3.3 Conclusion . . . 65

4 Instanciation de l'architecture THINK dans les systèmes d'exploitation 67
4.1 Aperçu de l'implantation . . . 67
4.1.1 Composants . . . 67
4.1.2 Interfaces & liaisons . . . 68
4.1.3 Composition des composants . . . 68
4.1.4 Portabilité . . . 68
4.2 Implantation du composant et de la liaison . . . 68
4.2.1 Représentation binaire des interfaces . . . 69
4.2.2 Langage de description d'interfaces . . . 73
4.2.3 Compilation des interfaces . . . 74
4.2.4 Exemple de codes . . . 75
4.3 Implantation des domaines . . . 77
4.3.1 Classification des domaines . . . 77
4.3.2 Amorçage des domaines . . . 79
4.3.3 Modèle d'exécution . . . 79
4.4 Implantation de la composition . . . 80
4.4.1 Langage de description d'architecture . . . 80
4.4.2 Convention de nommage pour les composants . . . 82
4.4.3 Outils . . . 84
4.5 Mise en œuvre . . . 86
4.5.1 Chaîne de génération . . . 86
4.5.2 Règles de programmation des composants . . . 87

4.5.3 Règles de programmation des usines à liaisons 87
4.5.4 Règles de programmation des systèmes 87

5 Bibliothèque de composants KORTEX **89**
5.1 Prototype 89
5.1.1 Philosophie 89
5.1.2 Caractéristiques 90
5.1.3 Amorçage du système 90
5.2 Composants POWERPC 90
5.2.1 Exceptions 90
5.2.2 Unité de gestion de la mémoire 92
5.2.3 Pilotes de périphériques 93
5.3 Composants mémoire 93
5.3.1 Modèle de mémoire paginée 93
5.3.2 Allocateur dynamique de mémoire 95
5.4 Composants d'exécution 96
5.4.1 Fils d'exécution & ordonnancement 96
5.4.2 Processus 98
5.5 Composants réseaux 98
5.5.1 Usine à paquets 99
5.5.2 Modèle de pile de protocoles 99
5.5.3 Interfaces hautes, les applications 102
5.5.4 Interfaces basses, les cartes réseaux 103
5.6 Liaisons 103
5.6.1 Liaison appel système 104
5.6.2 Liaison remontée système 106
5.6.3 Liaison signal 108
5.6.4 Liaison LRPC 108
5.6.5 Liaison distante RPC 108
5.6.6 Liaison de synchronisation 111
5.7 Composants domaines 112
5.7.1 Le mini-courtier 112
5.7.2 Chargeur dynamique 113
5.8 Conclusion 114

6 Évaluations **117**
6.1 Construction de systèmes d'exploitation 117
6.1.1 Construction du noyau 117
6.1.2 Évaluation des composants POWERPC 121
6.1.3 Évaluation des liaisons systèmes 122
6.1.4 Évaluation de la liaison distante RPC 125
6.2 Nonnoyau 126
6.3 Systèmes dédiés 127

6.3.1 PlanP 127
6.3.2 Kaffe 129
6.3.3 Doom 132
6.4 Conclusion 133
6.4.1 Évaluation qualitative 133
6.4.2 Évaluation quantitative 134

7 Conclusion générale 135
7.1 Bilan 135
7.2 Comparaison avec l'état de l'art 135
7.3 Limites 137
7.4 Perspectives 138

Chapitre 1

Introduction

1.1 Contexte et objectif de la thèse

Cette thèse s'inscrit dans la perspective de l'émergence d'un environnement global de traitement de l'information, dans lequel la plupart des objets physiques qui nous entourent sont équipés de processeurs et sont dotés de capacités de communication. Un tel environnement engendre inévitablement une grande diversité tant dans le matériel physique que dans les applications logicielles. Chacune de ces applications possède des contraintes plus ou moins fortes, qui lui sont propres. Ces contraintes peuvent être très variées. Il peut s'agir de contraintes de qualité de service comme dans les systèmes temps réels ou le multimédia. Il peut aussi s'agir de contraintes de fonctionnement, telles que la taille mémoire ou la consommation d'énergie, comme dans le domaine de l'embarqué par exemple les assistants personnels et les téléphones portables.

Les travaux menés ces dernières années sur les infrastructures logicielles, par exemple dans le domaine du temps réel [Schmidt et al. 1998], du multimédia [Blair et Stefani 1997] ou de la mobilité [Noble et al. 1997], montrent qu'il est impossible de disposer de mécanismes universels de gestion des ressources, des communications, etc. Ces travaux suggèrent qu'on ne puisse envisager de construire une infrastructure logicielle universelle où toutes les fonctions seraient adaptées à toutes les utilisations possibles. Il faut donc s'orienter vers une architecture suffisamment flexible. Celle-ci serait pensée pour être adaptée aux contraintes des applications et à l'environnement matériel. Cette architecture pourrait se décliner en différentes instances et permettre de construire des systèmes flexibles ou adaptés à des domaines fortement spécialisés.

L'objectif de cette thèse est de contribuer à la définition d'une architecture de systèmes flexibles. Pour montrer la faisabilité et la validité de l'approche, nous avons implanté des canevas logiciels reprenant les principaux éléments de cette architecture et développé une bibliothèque de composants systèmes. Ces derniers permettent par assemblage la construction de différents noyaux d'infrastructure réalisant les fonctions minimums d'un système d'exploitation.

Le travail présenté dans cette thèse a été effectué au sein du département Architecture des Systèmes Répartis de la Direction des Techniques Logicielles (DTL/ASR) de France Télécom Recherche & Développement. Ce département est chargé des études et de l'expertise sur les systèmes répartis et leurs architectures logicielles. Ce travail de thèse s'insère dans les travaux menés sur les intergiciels « middleware » et les infrastructures logicielles pour la mise en place de plates-formes adaptables de services de télécommunications.

1.2 Pourquoi une nouvelle architecture ?

Dans cette thèse, nous nous intéressons à une architecture de système pour environnement réparti. Il serait possible de travailler au niveau intergiciel « middleware » dont le rôle est d'assurer un lien entre des applications distinctes. Mais nous voulons travailler au plus bas niveau et montrer la possibilité d'implanter une telle architecture dans les systèmes d'exploitation. Nous allons commencer par expliquer les fonctions d'un système d'exploitation et faire un rapide historique de ces derniers. Puis nous identifions la façon d'aller plus loin vis-à-vis de la flexibilité.

1.2.1 Fonction et service d'un système d'exploitation

Le *système d'exploitation* est une partie essentielle d'un système informatique. Il assure l'intermédiaire entre les programmes des utilisateurs et les ressources matérielles d'une ou de plusieurs machines. Son rôle est de fournir un environnement dans lequel les utilisateurs peuvent exécuter leurs propres programmes conçus en vue d'une utilisation particulière, comme la gestion, le calcul, la programmation, les jeux, etc. Ces différents programmes sont appelés des applications. L'objectif principal d'un système d'exploitation est de rendre l'utilisation des ressources matérielles aussi simple que possible et de les multiplexer entre les différentes applications. Son deuxième objectif est de permettre une utilisation aussi efficace que possible des ressources matérielles. Ces deux objectifs, transparence et efficacité, sont potentiellement contradictoires et nécessitent d'effectuer des compromis. Par exemple, les mécanismes de protection mis en œuvre dans les systèmes, qui permettent d'isoler les applications et qui fournissent l'illusion d'une machine non partagée, sont coûteux. Mais ils permettent une utilisation plus sûre de la machine.

Le système d'exploitation, une machine étendue

Pour permettre l'utilisation des ressources, le système d'exploitation fournit aux applications un ensemble de services. Ces services sont des abstractions construites à partir des ressources matérielles, comme les périphériques et les contrôleurs. Ces abstractions sont typiquement : de la mémoire virtuelle, des processus, des fichiers, des « sockets », etc. Le système se charge pour sa part de traduire, de la façon la plus efficace possible, les requêtes effectuées sur ces abstractions par des instructions pour microprocesseur, coprocesseur, contrôleur divers. Un système d'exploitation peut donc être vu comme une machine étendue, c'est-à-dire une machine virtuelle.

Le système d'exploitation, un multiplexeur

Les problèmes se compliquent lorsqu'une même machine est partagée simultanément par plusieurs utilisateurs exécutant chacun leurs propres applications. C'est en effet au système d'exploitation de contrôler et de coordonner l'exploitation des ressources matérielles et des ressources abstraites entre les différentes applications. Il doit donc offrir des mécanismes d'exécution concurrente tels que des primitives de synchronisation pour éviter les conflits d'accès, et des mécanismes de communication tels que de la mémoire partagée. Ici, le système d'exploitation peut être vu comme un multiplexeur de ressources.

L'infrastructure logicielle répartie, une extension naturelle

De nouvelles difficultés apparaissent lorsque les applications se répartissent en s'exécutant simultanément sur plusieurs machines. Le rôle de l'*infrastructure logicielle répartie*, qui peut être vue comme une extension naturelle du système d'exploitation, est alors de fournir des services offrant des mécanismes de répartition. Ces mécanismes sont par exemple la communication, la tolérance aux défaillances des nombreuses entités mises en jeu, etc. Les bénéfices attendus sont par exemple, le partage des ressources comme les imprimantes ou les disques, l'amélioration des performances par la parallélisation des traitements, la fiabilisation par la redondance ou tout simplement l'échange de données comme la messagerie électronique.

1.2.2 Évolution des systèmes d'exploitation

La préhistoire

Sur les machines à tubes électroniques des années 1940, la programmation se faisait intégralement en langage machine sur des cartes perforées [Tanenbaum 1994, Silberschatz et Galvin 1998]. Le système d'exploitation était alors inconnu et les applications s'exécutaient directement sur la machine. D'autre part, l'introduction du transistor, inventé par Bell en 1947, permit la conception de machines plus fiables, plus petites et plus performantes.

Traitement par lots, le premier concept

Au milieu des années 1950, le *traitement par lots* fut adopté pour limiter les manipulations de cartes perforées. L'idée était de collecter un ensemble de travaux avec un petit ordinateur, puis de les transférer via une bande magnétique sur un ordinateur plus performant mais aussi plus onéreux. Un programme spécial, en fait l'ancêtre des systèmes d'exploitation d'aujourd'hui, permettait de lire, de mettre en mémoire et d'exécuter de façon séquentielle l'ensemble des travaux d'un lot. À cette époque, les ordinateurs étaient surtout utilisés pour les calculs scientifiques programmés en Fortran ou en assembleur. Les systèmes d'exploitation les plus connus étaient FMS (Fortran Monitor System) et IBSYS d'IBM.

Spool et multiprogrammation, la parallélisation des travaux

Jusqu'au début des années 1960, l'utilisation du processeur restait limitée par la lenteur des lecteurs de cartes. La solution proposée alors consistait à faire chevaucher les opérations d'entrées-sorties avec l'exécution des travaux. Ce concept, appelé « spool » (« Simultaneous Peripheral Operation On Line »), put être mis en œuvre avec l'apparition des disques qui étaient utilisés comme un grand tampon. Ainsi, les travaux étaient transférés depuis les cartes sur le disque simultanément à l'exécution du travail en cours. Dès que ce travail se terminait, le système d'exploitation chargeait alors depuis le disque le nouveau travail. Le concept de « spool » était utilisé de façon similaire pour les impressions.

Néanmoins, lorsque le travail en cours attendait la fin d'une opération d'entrée-sortie, le processeur restait inoccupé. Pour remédier à cela, le concept de *multiprogrammation* fut proposé au début des années 1960 par IBM avec l'OS/360. L'idée était de partitionner la mémoire afin d'y mettre simultanément plusieurs travaux. Lorsque le travail courant devait attendre, le processeur pouvait exécuter

un autre travail. Ainsi, le processeur pouvait pratiquement être utilisé en permanence. Le choix du travail à exécuter était décidé par un ordonnanceur. C'était la première fois que le système d'exploitation prenait lui-même des décisions pour les utilisateurs.

Le temps partagé, l'exécution concurrente

À elle seule, la multiprogrammation fournissait un environnement d'exécution suffisamment complet pour une utilisation efficace des ressources. Mais les travaux étaient toujours traités par lots et il n'était pas possible d'interagir avec un travail, même à des fins de débogage ou de contrôle, avant la fin complète de son exécution. Ce besoin de temps de réponse rapide a conduit vers la notion de *temps partagé*, une évolution logique de la multiprogrammation, qui donnait l'illusion d'une exécution concurrente et simultanée de plusieurs travaux grâce à un changement fréquent de travail courant. L'idée reposait sur la supposition qu'une interaction, par exemple via une console, entre un travail et un utilisateur était toujours de courte durée. Le premier système à temps partagé fut CTSS (« Compatible Time-Sharing System ») développé au MIT en 1962.

La mémoire virtuelle, s'affranchir des limites physiques

Le succès du système CTSS fit naître MULTICS (« MULTiplexed Information and Computing Service ») [Organick 1972], écrit en PL/I par Bell et General Electric à partir de 1965. MULTICS a fourni de nombreux concepts, comme les système de fichier hiérarchique et le processus. Mais on retiendra principalement celui de la *mémoire virtuelle*. L'idée de la mémoire virtuelle était d'une part de permettre l'exécution des processus qui n'étaient pas totalement en mémoire et dont certaines parties résidaient sur disque et d'autre part d'isoler les processus. Ce concept était d'autant plus important à l'époque que les applications et le système étaient d'un seul bloc et devenaient énormes.

Noyau monolithique, simplifier les applications

Au début des années 1970, Bell développa une version simplifiée de MULTICS, appelé par dérision UNIX (« Uniplexed Information and Computing Service ») [Ritchie et Thompson 1974]. Ce nouveau système encourageait une nouvelle approche pour la conception des logiciels consistant à résoudre les problèmes en interconnectant des outils basiques, plutôt qu'en créant de grandes applications. Cette approche reposait sur un nouveau concept de communication entre les processus, le *pipe* permettant la communication par flot de données. UNIX était architecturé sous forme d'un *noyau monolithique* intégrant tous les services systèmes potentiellement requis par les applications. Le noyau résultant de cette architecture était gros et complexe, sa maintenance et son évolution devenaient extrêmement difficiles et posaient de gros problèmes de fiabilité. De plus, les politiques implantées dans le noyau ne correspondaient pas toujours aux besoins des applications et leurs modifications étaient impossibles. Il devenait donc essentiel de structurer les noyaux en vue de simplifier leur conception et de permettre aux applications de mieux contrôler le fonctionnement du noyau.

Système flexible, repenser l'architecture des systèmes

Depuis le début des années 1980, on assiste à un foisonnement de propositions d'architectures flexibles ou extensibles censées résoudre le problème des noyaux monolithiques. Ces systèmes sont

communément appelés des *systèmes flexibles*. La flexibilité résulte de la recherche d'un compromis entre transparence et efficacité. Les principales innovations classées chronologiquement sont les suivantes.

En 1984, l'université Carnegie-Mellon présentait le système Mach basé sur un *micronoyau* [Accetta et al. 1986]. L'idée est de sortir les services systèmes du noyau afin de les exécuter dans des processus séparés, appelés serveurs, de façon à isoler leurs éventuelles défaillances et à permettre leur construction à posteriori et leur lancement de façon dynamique. Le noyau peut alors se limiter à des abstractions de processus et de communication inter-processus. Le problème de cette approche est le coût des communications dû aux changements de contexte, ce qui a pour conséquence de limiter la granularité des services. Malgré des techniques d'optimisation proposées dès 1993 par l'université de Dresden avec le micronoyau L3 [Liedtke 1993] puis L4 [Liedtke 1995], ce problème inhérent à l'approche micronoyau subsiste encore.

En 1987, l'université de Illinois proposait le système d'exploitation orienté objets Choices [Campbell et al. 1993] basé sur une approche totalement différente. L'idée est d'architecturer le noyau à base d'objets pouvant être remplacés ou modifiés en vue d'étendre et de reconfigurer le système en fonction des contraintes des applications. Cette approche repose sur l'utilisation des langages de programmation orientée objets. Cette structuration simplifie grandement la conception et la complexité des noyaux et permet donc d'augmenter ainsi leur fiabilité.

Au début des années 1990, les systèmes monolithiques ont été enrichis de mécanisme d'extensibilité dynamique permettant d'ajouter du code arbitraire dans le noyau, comme avec Spin [Bershad et al. 1995] et Vino [Small et Seltzer 1994]. Mais l'extensibilité se limite à celle prévue à la conception du système, comme par exemple l'ajout de pilotes ou de systèmes de fichiers. De plus, cette approche ne règle pas les problèmes des éventuelles défaillances des services ajoutés qui peuvent corrompre tout le système. Néanmoins, l'écriture du service dans un langage sûr, ou l'utilisation de « sandbox », étend un noyau de façon sûre.

En 1992, Sony et l'université de Keio proposaient le système d'exploitation réflexif Apertos [Yokote 1992]. L'idée est de réifier les sémantiques des objets composants le système sous la forme d'un ensemble de méta-objets permettant de changer le fonctionnement des objets. Cette approche repose sur les techniques de réflexion introduites dans les langages à objets.

En 1995, le MIT propose le système Aegis basé sur un *exonoyau* [Engler et al. 1995]. L'idée est de proposer des services sous forme de librairies systèmes s'exécutant directement dans l'espace des applications et non plus dans le noyau ni dans des serveurs. La philosophie prônée est d'éliminer toutes les abstractions du noyau et de voir le noyau strictement comme un multiplexeur de ressources. Cette approche apporte un degré de flexibilité supplémentaire en permettant à chaque application de modifier individuellement les services. Néanmoins, quelques abstractions subsistent dans le noyau, comme le processus et le quantum de temps. De plus, le modèle de programmation n'est pas clair et cela rend la programmation des systèmes complexes.

1.2.3 Comment aller plus loin ?

Ce foisonnement de systèmes flexibles montre la difficulté de concevoir un tel système et nous voyons qu'il n'y a pas de consensus dans ce domaine. Certains proposent des techniques de structuration du noyau. D'autres proposent d'éclater le noyau, de le supprimer ou de le diminuer. Chaque approche comporte ses avantages et ses inconvénients qui sont plus ou moins sensibles en fonction de l'utilisation faite du noyau.

L'idée développée dans cette thèse est de concevoir une architecture de système et de l'appliquer au plus bas niveau. Cette architecture ne doit pas faire de choix d'implantation ayant pour conséquence de se fixer sur une des approches que nous venons de voir. L'ensemble de ces approches forme un espace de conception, dont une illustration est donnée à la figure 2.26 du chapitre suivant page 47. L'architecture doit être suffisamment flexible et offrir des moyens pour couvrir tout cet espace de conception et laisser aux concepteurs de systèmes le choix des compromis satisfaisant aux contraintes des applications cibles.

Définir une architecture de système

La première contribution de cette thèse est la spécification d'une architecture de système. Cette architecture est applicable à toutes les infrastructures logicielles réparties et à tous les systèmes d'exploitation, qu'ils soient centralisés, embarqués ou dédiés à une application. Nous clarifions les concepts et les invariants de structure des systèmes. Cela peut conduire à terme à la formalisation de l'architecture permettant de raisonner sur les structures des systèmes et de prouver certaines propriétés. Nous mettons en œuvre cette architecture avec des canevas logiciels et des outils de développements associés.

L'architecture proposée repose essentiellement sur les notions de composants, de liaisons flexibles et de domaines. Dans cette architecture, un composant modélise tout type de ressources, qu'elles soient matérielles ou logicielles. Un composant est l'unité d'encapsulation, il est uniquement accessible via une ou plusieurs interfaces. La liaison modélise toutes les interactions entre les composants systèmes ou applicatifs, y compris les communications avec les composants matériels et avec les composants distants. La flexibilité de la liaison permet de modifier dynamiquement la sémantique de communication sans modification des composants. Le domaine modélise l'isolation des composants engendrée par les contraintes physiques comme avec la répartition ou des contraintes de protection comme avec les processus. La forme générale des liaisons flexibles et des domaines est donnée par des canevas logiciels minimaux.

Repartir à zéro

La deuxième contribution de cette thèse est la réalisation d'une bibliothèque de services systèmes conformes à l'architecture proposée. Cette bibliothèque offre un ensemble de composants implantant des services communément rencontrés dans les systèmes d'exploitation. Elle offre en outre des composants permettant de réifier et de manipuler les ressources matérielles. Tous ces services sont construits selon une philosophie qui consiste à minimiser le nombre d'abstractions fournies par un composant, en vue de maximiser les possibilités de composition et de réutilisation des composants.

Cette bibliothèque permet de construire très simplement des systèmes ou des noyaux par la réutilisation de composants systèmes standards ou spécialisés. Elle est ouverte de façon à permettre le remplacement d'un service inadapté à l'utilisateur par un autre spécialisé aux besoins de l'utilisation.

1.3 Plan de la thèse

Ce mémoire de thèse est organisé en cinq chapitres, de 2 à 6, auxquels il faut ajouter le chapitre 1 de l'introduction et le chapitre 7 de la conclusion. Au travers de ces chapitres, nous montrons

étape par étape comment sont construits les systèmes d'exploitation depuis la formalisation jusqu'à l'évaluation.

Le chapitre 2 décrit l'état de l'art, il est consacré à la présentation des principes et des techniques de construction des systèmes d'exploitation embarqués, centralisés ou répartis. Nous commençons par aborder les différents concepts mis en œuvre dans les systèmes. Puis nous voyons les principes d'organisation et de structuration facilitant la construction des systèmes et les différents modèles de communications. Nous finissons par une analyse détaillée de l'architecture des noyaux des systèmes d'exploitation existants.

Le chapitre 3 est dédié à la présentation de THINK, le modèle d'architecture proposé. Nous expliquons ici les concepts minimums, dégagés durant cette thèse, permettant de modéliser les systèmes, à savoir le composant, la liaison et le domaine. Cette modélisation couvre toutes les techniques de construction et toutes les architectures de systèmes vues au chapitre 2. Nous montrons comment ces concepts sont mis en œuvre dans un système d'exploitation.

Le chapitre 4 présente une proposition concrète d'implantation des concepts du modèle d'architecture THINK. Cette implantation est indépendante des langages de programmation et peut donc être utilisée avec des langages de très bas niveau, comme C ou l'assembleur, et des langages de plus haut niveau, comme Java. Nous décrivons aussi les différents outils de développement associés au modèle.

Le chapitre 5 est consacré à la présentation d'une bibliothèque associée au modèle THINK, nommée KORTEX. Nous décrivons les composants systèmes et les liaisons systèmes fournis par cette bibliothèque. Ces composants et ces liaisons, conformes au modèle THINK, permettent par composition la construction arbitraire d'infrastructures de système d'exploitation. Les composants systèmes fournissent des services allant du plus bas niveau (pilotes de périphérique) jusqu'au plus haut niveau (processus). Les liaisons systèmes réalisent les interactions entre les composants quels que soient leurs domaines d'exécution. Ce chapitre montre en outre comment il est possible de spécifier un système d'exploitation grâce aux interfaces.

Le chapitre 6 détaille les noyaux d'infrastructure de système réalisés grâce à la bibliothèque KORTEX, comme un micronoyau et un noyau monolithique, ainsi que les différents systèmes dédiés à des applications spécialisées, comme un routeur de réseau actif et un système Java. Ce chapitre donne aussi une évaluation qualitative et quantitative du modèle d'architecture THINK et de la bibliothèque KORTEX. Nous examinons en particulier le coût des liaisons systèmes à travers leurs utilisations dans un noyau et leur influence sur les performances générales.

Le chapitre 7 présente la conclusion, les limitations et les perspectives des travaux de recherche menés durant cette thèse. Ce chapitre présente aussi une comparaison du travail avec l'état de l'art.

Chapitre 2

Principes et techniques de construction des systèmes

Ce chapitre présente un état de l'art sur les différents principes et techniques de construction et de structuration des systèmes d'exploitation centralisés et répartis. Plutôt que d'énumérer l'ensemble des systèmes d'exploitation existants et d'analyser leurs caractéristiques, la démarche choisie est conceptuelle et présente de façon synthétique les techniques existantes. Ces techniques devenues désormais classiques ont mis de nombreuses années à émerger. Pour chacune d'elles, nous présentons dans la mesure du possible, le système ayant introduit la technique et des exemples récents d'utilisation.

Cette analyse n'est pas exhaustive et se limite à étudier les techniques requises afin de bien comprendre l'architecture de système proposée dans cette thèse. Nous commençons par étudier, à la section 2.2, les différents concepts mis en œuvre dans un système d'exploitation et les fonctions qu'ils permettent de réaliser. Nous étudions ensuite, à la section 2.3, comment le système permet aux applications d'utiliser ces concepts via des notions de désignation, d'allocation et de synchronisation. La section 2.4 présente les principes d'organisation et de structuration des systèmes. La section 2.6 présente les modèles de communications rencontrés dans les systèmes. Pour finir, nous passons en revue en section 2.7 les architectures classiques des systèmes d'exploitation et les systèmes qui les utilisent.

2.1 Définitions

Avant de débuter cette analyse des principes et techniques de construction des systèmes, nous introduisons quelques définitions préliminaires sur les trois concepts qui nous semblent suffisants pour décrire un système.

Ressource

On appelle une *ressource*, tout objet logiciel ou matériel pouvant être utilisé par un programme [Krakowiak 1985]. Une ressource peut être un périphérique ou un contrôleur matériel, comme par exemple un disque, ou une mémoire ou une information, comme par exemple une donnée dans une mémoire.

Interface

Une ressource peut être utilisée grâce à un ensemble de primitives d'accès constituant son *interface* et un ensemble de règles constituant son mode d'emploi. En fait, une interface définit un langage qui permet aux usagers de communiquer avec la ressource [Krakowiak 1985]. Toute l'information nécessaire à la bonne utilisation d'une ressource par un utilisateur est contenue dans l'interface. Par exemple, l'interface d'accès à la mémoire physique est constituée des primitives `load` et `store`, l'interface d'un périphérique est constituée des primitives `in` et `out` dans des registres matériels, l'interface d'un allocateur de mémoire est constituée des primitives `alloc` et `free`. Les règles expriment en général des restrictions sur l'usage de la ressource, par exemple la lecture seule ou l'ordre d'exécution des méthodes.

Programme

Un *programme* est composé d'une suite d'instructions, dont l'exécution fait évoluer l'état de la machine [Krakowiak 1985]. L'activité résultant de l'exécution d'un programme est appelée une *tâche*. Concrètement, un programme est une suite d'instructions effectuant des appels aux interfaces des ressources composant le système.

2.2 Abstraction et virtualisation des ressources

Un des rôles d'un système d'exploitation est d'offrir une vue abstraite d'une machine matérielle et de ses ressources. Un système d'exploitation peut construire des abstractions nouvelles qui permettent d'offrir une machine logicielle avec des formes différentes de la machine matérielle. Il peut aussi virtualiser la machine physique, à des fins de partage, en offrant une vue logicielle fidèle dans le sens où il n'ajoute pas d'abstraction supplémentaire.

L'*abstraction* masque et redéfinit l'organisation d'une ressource en définissant une nouvelle ressource de plus haut niveau dotée d'une interface éventuellement simplifiée. Par exemple, l'interface d'un pilote de carte réseau est constituée des primitives `send` et `receive`, et elle simplifie l'utilisation du périphérique qui réside principalement dans des combinaisons de décalages de bits et de suite d'accès à des registres matériels.

Un raffinement du processus d'abstraction est la *virtualisation*. Elle consiste à simuler le comportement d'une ressource n'ayant pas d'existence physique propre. Une telle ressource est appelée une *ressource virtuelle* et est construite à l'aide d'une ou de plusieurs ressources de plus bas niveau. La virtualisation s'applique généralement aux ressources de bas niveau, comme le processeur ou la mémoire. Généralement, l'objectif d'une ressource virtuelle est de multiplexer une ressource physique en donnant l'illusion d'utiliser directement la ressource physique. Par extension, on parle de machine virtuelle lorsque toutes les ressources d'une machine physique sont virtualisées. Comme toutes les ressources virtuelles, une machine virtuelle n'a pas d'existence propre, mais son comportement est simulé, directement sur une machine physique ou dans un processus d'un système d'exploitation [Krakowiak 1985].

Cette section présente les concepts existants pour réaliser la fonction principale d'un système d'exploitation, à savoir gérer les ressources. Nous passons en revue brièvement l'ensemble des ressources rencontrées dans un système et nous examinons l'ensemble des dépendances entre celles-ci.

Nous commençons par les différentes ressources matérielles rencontrées dans les ordinateurs. Nous présentons ensuite les abstractions que le système d'exploitation fabrique à partir de ces ressources. Pour finir, nous examinons comment il est possible de virtualiser complètement une machine. Pour de plus amples détails sur le fonctionnement et l'utilisation de ces ressources, on consultera les ouvrages à usage général sur les systèmes d'exploitation. Nous pouvons citer celui en français de [Krakowiak 1985] et ceux en anglais de [Tanenbaum 1994] et [Silberschatz et Galvin 1998]. Nous pouvons aussi citer la présentation de [Krakowiak 2001].

2.2.1 Ressources matérielles

Un ordinateur se compose d'un ou de plusieurs processeurs, d'une mémoire principale, d'un ou plusieurs disques constituant la mémoire secondaire, de cartes réseaux, de contrôleurs matériels et de périphériques d'entrées-sorties. Ces différents composants sont reliés par différents bus d'interconnexion, voir figure 2.1.

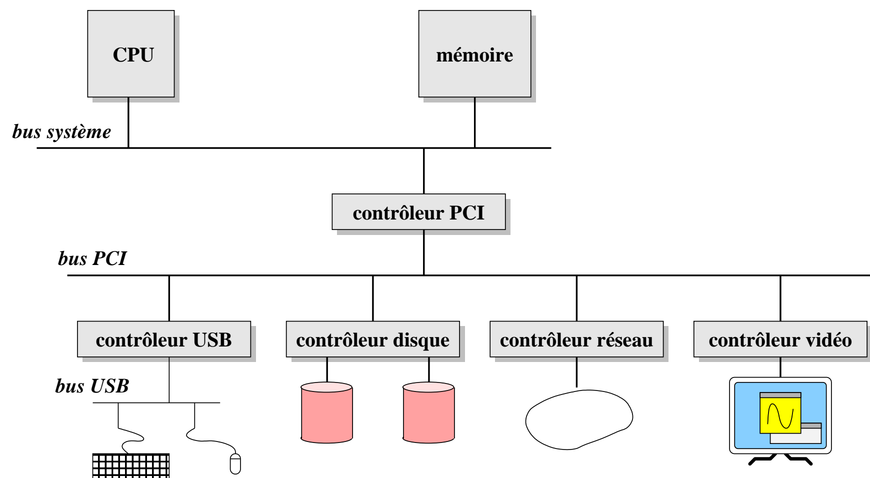


FIGURE 2.1 – Exemple d'architecture d'un ordinateur moderne

Pour de plus ample information sur l'architecture des ordinateurs, on consultera les ouvrages en anglais de [Tanenbaum 1990] ou de [Hennessy et Patterson 1996]. Il est aussi possible de consulter les ouvrages spécifiques à une architecture de machine, par exemple pour les machines à base de PowerPC [Apple Computer Inc et al. 1995] et plus spécifiquement pour les PowerMacintosh [Apple Computer Inc 1995].

Processeur

Un *processeur* « Central Processing Unit » (CPU) est le composant destiné, dans un ordinateur, à interpréter et à exécuter de façon séquentielle les instructions en langage machine d'un programme, nous parlons d'*exécution séquentielle*. C'est le cœur de l'ordinateur, c'est lui qui coordonne le reste des éléments.

Pour une étude complète des processeurs, on consultera les manuels de référence des processeurs. Pour le PowerPC, on consultera les documentations [Motorola Inc 1997a] et [Motorola Inc 1997b].

Pour le Pentium, on consultera la documentation [Intel Corporation 1995]. Un processeur est, au minimum, composé des quatre entités suivantes.

- *unité arithmétique et logique* (UAL). Le rôle de l'unité arithmétique et logique est d'effectuer les opérations arithmétiques et logiques comme les additions ou les comparaisons.
- *unité de contrôle* (UC). Le rôle de l'unité de contrôle est d'extraire les instructions stockées en mémoire, de les décoder et de les exécuter en utilisant si nécessaire l'UAL.
- *bus d'adresse* & *bus de données*. Le bus d'adresse et le bus de données relient les différentes entités.
- *horloge*. Le processeur est rythmé par une *horloge*. À chaque cycle d'horloge, le processeur peut effectuer une ou plusieurs opérations (en pipeline ou en superscalaire).

Un processeur fournit un modèle de programmation sur lequel repose la conception du système d'exploitation. Ce modèle de programmation peut être caractérisé par les quatre points suivants.

- *jeu d'instructions*. Le jeu d'instructions constitue l'interface permettant d'utiliser le processeur.
- *contexte du processeur*. Le contexte du processeur n'est autre que l'ensemble des registres du processeur. Ce contexte est constitué des registres généraux et des registres spéciaux décrivant l'état du processeur.
- *exception* & *événement*. Lorsqu'une exception, comme un déroutement suite à une anomalie du programme (division par zéro, accès mémoire invalide, instruction invalide, ...), ou un événement, comme une interruption matérielle déclenchée par un périphérique, surviennent, une *commutation de contexte* est effectuée par le processeur. Cette commutation interrompt l'exécution séquentielle du programme de façon à exécuter un traitant d'exception, voir figure 2.2. Ce traitant d'exception doit être mis en place par le système d'exploitation pour chaque type d'exception supporté par le processeur. Pour pouvoir reprendre l'exécution de la séquence d'instruction interrompue, le système d'exploitation doit, au minimum, sauver le contexte du processeur que le traitant va modifier durant sa propre exécution.

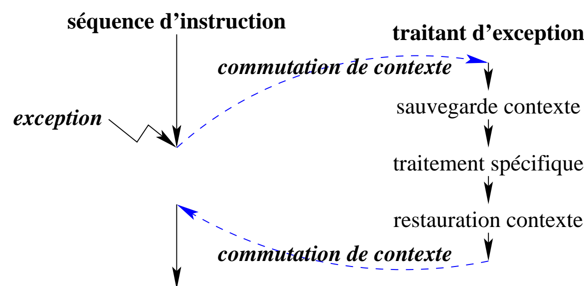


FIGURE 2.2 – Commutation de contexte pour la prise en compte d'une exception

- *mode utilisateur* & *mode utilisateur*. Pour des raisons de protection, un processeur fournit au minimum deux modes de fonctionnement ; le mode utilisateur et le mode superviseur. Cela permet de réserver des *instructions privilégiées* qui sont exécutables uniquement en mode superviseur. Un programme s'exécutant en mode utilisateur peut appeler un service nécessitant

l'exécution en mode superviseur en effectuant un *appel système*. Ce dernier se comporte comme une exception et comme le traitant est mis en place par le système d'exploitation, ce dernier peut ainsi garantir la cohérence globale du système. Nous reviendrons plus en détail par la suite sur ce qui se cache derrière l'appel système.

Mémoire principale

La *mémoire principale* est une mémoire physique « Random Access Memory » (RAM) interne à la machine. Cette mémoire stocke de manière temporaire des données et les instructions lors de l'exécution des programmes. Le processeur ainsi que les contrôleurs de périphériques peuvent y accéder directement. La mémoire physique est généralement composée des composants suivants.

- *bus système*. Un bus système assure la connexion du processeur à la mémoire principale. La vitesse de ce bus détermine le temps d'accès à la mémoire. Comme les processeurs deviennent de plus en plus rapides, ce bus constitue l'un des principaux goulots d'étranglement sur les ordinateurs modernes. Du fait de la lenteur de ce bus, le temps d'accès à la mémoire n'est pas instantané et peut consommer de nombreux cycles d'horloge du processeur. La figure 2.3 montre le cheminement des données et de la traduction des adresses entre les différents composants lors de l'accès à la mémoire.
- *mémoire cache*. L'utilisation d'un composant mémoire cache très rapide diminue les temps d'accès à la mémoire en conservant dans cette mémoire les dernières données accédées. Bien évidemment, un tel mécanisme engendre des problèmes de cohérence et de politique de remplacement des données. La mémoire cache peut être interne (cache L1) ou externe (cache L2) au processeur. Il peut aussi exister des caches non unifiés ; un cache pour les instructions et un cache pour les données.
- *unité de gestion de la mémoire* « Memory Management Unit » (MMU). Une unité de gestion de la mémoire est un composant matériel permettant de traduire les requêtes de lectures ou d'écritures à des *adresses logiques* (utilisées par le processeur) en des requêtes à des *adresses physiques* (utilisées par la mémoire principale). Cette translation est réalisée grâce à une table appelée « Translation Look-aside Buffer » (TLB). Ce composant est utilisé pour construire une *mémoire virtuelle* et traiter les exceptions lors de l'accès à une adresse logique n'ayant pas de correspondance en mémoire principale, nous y reviendrons à la section suivante. La mémoire virtuelle est divisée en petites unités appelées des *pages*. Une *table de pages* permet au système de décrire la mémoire virtuelle. La TLB est en fait un cache de cette table de page. Généralement, ce composant est intégré au processeur. Certaines machines n'en sont pas pourvues, c'est le cas pour les systèmes embarqués.

Périphériques d'entrées-sorties

Les *périphériques* d'entrées-sorties « device » permettent à la machine de communiquer avec l'extérieur et en particulier avec les utilisateurs. Ces périphériques sont multiples et très divers : interfaces réseaux, disques, écrans, claviers, souris, imprimantes, carte son, etc. Chaque type de périphérique est contrôlé par un dispositif de commande appelé *contrôleur*. Tous les contrôleurs sont reliés au bus système via un bus, par exemple un bus local « Peripheral Component Interconnect » (PCI). Le fonctionnement d'un contrôleur de périphérique peut être caractérisé de la façon suivante.

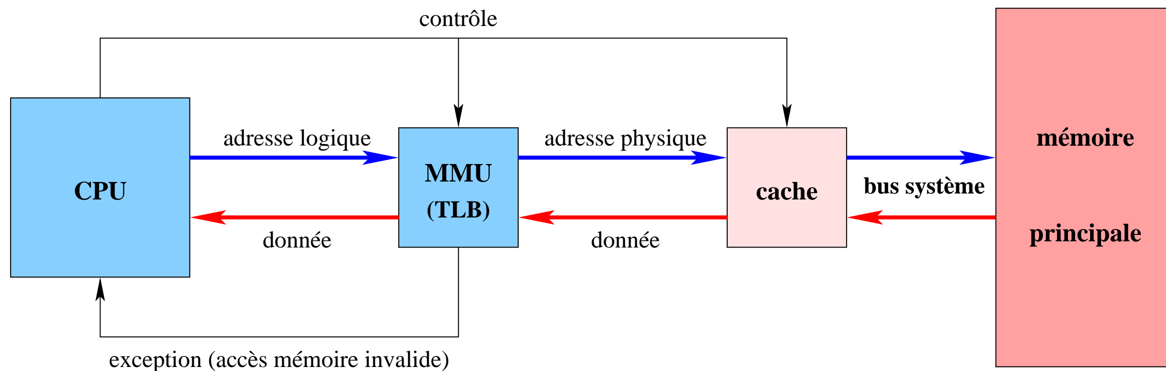


FIGURE 2.3 – Flux d'exécution lors de l'accès à la mémoire principale

- Le rôle de ce contrôleur est de faire fonctionner le périphérique de façon autonome. Il permet ainsi de décharger le processeur durant les entrées-sorties en réalisant lui-même l'enchaînement des commandes permettant de piloter le périphérique. Par exemple, lors d'une requête de lecture d'un secteur d'un disque, le contrôleur s'occupe seul de déplacer la tête de lecture, de la stabiliser et de lire le secteur voulu.
- Lorsqu'un contrôleur a terminé son travail. Il informe le système en générant une interruption matérielle, voir page 11. Certains contrôleurs n'offrent pas cette possibilité, le système doit alors consulter périodiquement les registres de données pour connaître l'état du contrôleur.
- Un contrôleur transfère de l'information entre le périphérique et le processeur ou la mémoire principale. Les transferts avec le processeur sont réalisés via des *registres de commandes* (permettant de commander le contrôleur) et des *registres de données* (permettant de consulter l'état du contrôleur) offerts par le contrôleur et directement accessibles par le processeur. Ces registres constituent l'interface du contrôleur de périphérique. Les transferts avec la mémoire peuvent être réalisés via une *unité d'accès direct à la mémoire* « Direct Memory Access » (DMA) qui doit être programmée par le système.

2.2.2 Ressources systèmes

Les ressources systèmes sont des abstractions de bases classiquement offertes par les systèmes d'exploitation. Elles sont soit directement construites au-dessus des ressources physiques, soit construites à partir d'autres ressources abstraites.

Fils d'exécution

$$\textit{processeur (CPU)} \stackrel{virtualisation}{\longrightarrow} \textit{fil d'exécution}$$

Un processeur classique ne peut exécuter qu'une seule exécution séquentielle à la fois. Néanmoins, de nombreux programmes nécessitent plusieurs chemins d'exécution simultanés. C'est pour cela qu'a été introduit le concept de *fil d'exécution* « threads » permettant de virtualiser le processeur.

Cette virtualisation permet le partage du processeur entre plusieurs fils d'exécution, en donnant l'illusion d'une exécution parallèle. Ainsi, un programme peut être conçu en utilisant plusieurs tâches, on parle alors de programmation *multitâche*. Les principaux intérêts de cette programmation sont les suivants.

- Lors d'une lecture sur un disque, le fil d'exécution ayant émis la requête se retrouve bloqué durant tout le temps de l'entrée-sortie. Or, en mettant en œuvre un second fil d'exécution, on peut continuer à effectuer des traitements pendant que le premier est bloqué. Cela permet une plus grande efficacité du système en profitant au mieux de la capacité de traitement.
- La deuxième utilisation des fils d'exécution est la parallélisation des algorithmes dans un but d'efficacité. Néanmoins, cet argument est valable uniquement si la machine est dotée de plusieurs processeurs. En effet, le concept de fils d'exécution n'accélère pas le processeur mais il permet uniquement de mieux l'utiliser en le virtualisant.
- Les fils d'exécution sont aussi utilisés à des fins de structuration. Il est par exemple plus simple d'utiliser un ou plusieurs fils d'exécution pour traiter des événements plutôt que d'ajouter des conditions de déroutement dans l'exécution principale.

Les fils d'exécution sont en concurrence entre eux pour l'accès aux ressources. Par exemple, ils partagent la même mémoire d'exécution du programme. D'autre part, certaines ressources possèdent des contraintes propres, par exemple elles peuvent ne pas être accessibles simultanément. Il peut être alors nécessaire de sérialiser l'accès à ces ressources pour conserver leur intégrité. Cette opération, appelée *synchronisation*, est abordée à la section 2.3.3.

La gestion et le partage du processeur sont réalisés grâce à un *ordonnanceur* [Silberschatz et Galvin 1998]. Son rôle est d'allouer le processeur aux tâches. L'interface d'un ordonnanceur est généralement composée des méthodes permettant de créer, d'endormir et de réveiller les tâches. L'ordonnancement peut être coopératif, c'est-à-dire que le processeur est réalloué explicitement. Il peut être préemptif, c'est-à-dire que l'exécution de la tâche active peut être implicitement suspendue. Le choix de la tâche à qui est alloué le processeur dépend de la politique d'allocation ; circulaire « round-robin », à priorité, etc. Nous reviendrons sur le concept d'allocation à la section 2.3.2.

Mémoire virtuelle

$$\left.\begin{array}{l}\textit{mémoire principale (RAM)}\\ \textit{unité de gestion de la mémoire (MMU)}\end{array}\right\} \xrightarrow{virtualisation} \textit{mémoire virtuelle}$$

La *mémoire virtuelle* sépare la mémoire logique utilisée par les programmes de la mémoire physique. Cette virtualisation de la mémoire est réalisée grâce à la MMU (voir page 13). En fait, la MMU réalise le *couplage* entre les adresses de la mémoire virtuelle et les adresses de la mémoire physique [Silberschatz et Galvin 1998]. Ainsi, les adresses de la mémoire virtuelle peuvent être continues alors que les adresses de la mémoire physique ne le sont pas. Par définition, l'ensemble des adresses logiques valides et accessibles par un programme est appelé son *espace d'adressage*. Si un programme s'exécute en mémoire physique, son espace d'adressage est la mémoire physique, s'il s'exécute en mémoire virtuelle, son espace d'adressage est la mémoire virtuelle. La mémoire virtuelle offre remplit deux fonctions.

- *protection* & *structuration*. La protection donne lieu à la création de *domaines de protection*. Un domaine de protection protège et isole les programmes et le système d'exploitation. C'est-à-dire qu'un programme qui s'exécute dans un domaine ne peut perturber la mémoire des autres programmes qui s'exécutent dans d'autres domaines de protection. Chaque programme possède alors son propre espace d'adressage. La partie du système d'exploitation s'exécutant dans un domaine de protection privilégié est appelée le noyau, mais nous y reviendrons à la fin de ce chapitre, voir section 2.7. Le concept de mémoire virtuelle permet la création de plusieurs domaines de protection, par exemple un ou plusieurs par processus.

  La mémoire virtuelle peut aussi être segmentée. Un *segment* est une suite d'emplacements consécutifs. Une adresse virtuelle peut alors s'exprimer comme un numéro de segment et un déplacement dans celui-ci. En général, les *segments* correspondent à un découpage logique d'un programme (segment de pile, de données, de code, etc.).

- *pagination*. La pagination donne l'illusion d'un espace plus grand que la mémoire et il est possible d'exécuter des programmes nécessitant plus de mémoire que n'en dispose la machine en créant une mémoire virtuelle extrêmement large. Seules les parties du programme les plus récemment utilisées sont en mémoire physique, les autres sont stockés sur une mémoire secondaire, par exemple un disque. Un mécanisme de va-et-vient permet le passage des données d'une mémoire à l'autre. Ce mécanisme est déclenché lors de l'accès à une page qui n'est pas en mémoire principale. Si un couplage existe, cela génère une exception appelée un *défaut de page*, c'est-à-dire que le page est sur le disque. Sinon, cela génère une exception appelée *faute d'adressage*, c'est-à-dire que la page n'existe pas, cela peut conduire à la terminaison brutale du processus.

Fichiers

$$\textit{disque} \overset{abstraction}{\longrightarrow} \textit{fichiers}$$

Un *fichier* est une ressource permettant de stocker de façon permanente des données sur disque. Pour l'utilisateur, les fichiers constituent la partie la plus visible d'un système d'exploitation. Cette abstraction offre une vue logique unifiée de la mémoire secondaire. Chaque fichier est désigné par un nom. Pour faciliter le nommage, le concept de *répertoire* regroupe des fichiers. Comme un répertoire est aussi un fichier, ce modèle engendre une arborescence.

La correspondance entre l'organisation logique des fichiers et l'organisation physique de la mémoire secondaire est réalisée par un *système de gestion de fichiers* (SGF). C'est lui qui implante les fonctions d'accès permettant de manipuler et de protéger les fichiers et les répertoires. Dans le monde Unix, l'interface offerte est relativement standard et se nomme « Virtual File System » (VFS). Cette interface peut être encapsulée dans une interface de type *flot* « stream » offrant les méthodes `put` et `get`.

Sockets

$$\textit{interface réseau} \overset{abstraction}{\longrightarrow} \textit{socket}$$

La notion de *socket* est une abstraction qui représente la connexion entre deux ordinateurs. Le rôle des sockets est d'offrir une interface permettant d'accéder aux différents protocoles réseaux.

Les *protocoles réseaux* sont des abstractions de communications offrant des services d'adressage, de fiabilité, de contrôle de flux, etc. On consultera les livres de Stevens pour une description des protocoles réseaux [Stevens 1994]. Comme pour les fichiers, les sockets peuvent offrir une interface flot.

Pilotes de périphériques

$$\textit{périphériques} \xrightarrow{abstraction} \textit{pilotes de périphérique}$$

La partie logicielle permettant de commander un contrôleur de périphérique se nomme un *pilote de périphérique* « driver ». Le pilote utilise directement l'interface fournie par le contrôleur et gère éventuellement la programmation des DMA. Le pilote traite les interruptions émises par le matériel et détecte et corrige les cas d'erreur. Bien évidemment, Il existe un pilote différent pour chaque type de contrôleur. Le pilote offre une interface de plus haut niveau permettant d'effectuer simplement des requêtes sur les périphériques, voir figure 2.4. Par exemple, un pilote de contrôleur d'interface réseau fournie une interface constituée des primitives `send` et `receive`.

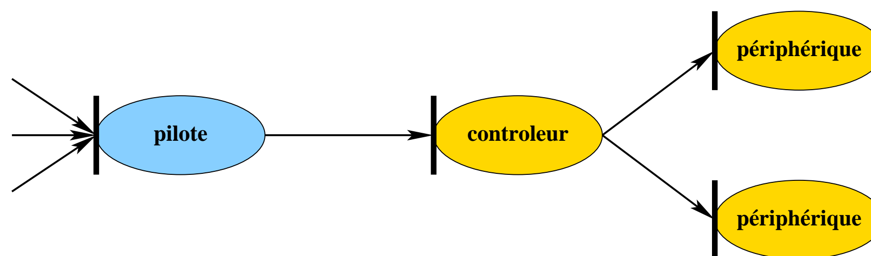


FIGURE 2.4 – Abstraction d'un pilote de périphériques

L'écriture d'un pilote de périphérique est complexe. Cette programmation réside principalement dans des combinaisons de décalages de bits et de suite d'accès à des registres. Cela engendre de nombreuses erreurs de programmation. Certaines solutions ont été apportées. Par exemple, le projet UDI [UDI Project 1999] propose une approche de normalisation des interfaces décrivant des pilotes portables. De plus, Devil [Mérillon et al. 2000] propose un langage dédié capturant des invariants du matériel. Cela simplifie le développement des pilotes et permet de vérifier certaines propriétés.

2.2.3 Processus

$$\left.\begin{array}{l}\textit{fils d'exécution}\\ \textit{mémoire virtuelle}\end{array}\right\} \xrightarrow{abstraction} \textit{processus}$$

Un *processus* est l'abstraction d'un programme qui s'exécute, c'est l'unité de travail dans un système. Un processus modélise les applications des utilisateurs, comme un navigateur Internet, et les services système, comme un serveur Web. Cette abstraction se retrouve dans tous les systèmes d'exploitation. Pour accomplir sa tâche, un programme requiert des ressources. Un processus regroupe en

fait toutes les ressources dont le programme a besoin pour accomplir sa tâche : du temps processeur, de la mémoire, des flux de données, etc. Ces ressources, que nous venons de voir, lui sont fournies par le système d'exploitation. Le système est donc un multiplexeur de ressources entre les différents processus. Les techniques de multiplexage sont abordées à la section 2.3.

Protection du système d'exploitation

Dans les systèmes d'exploitation multiprogrammés, les processus sont construits en utilisant un ou plusieurs fils d'exécution et un ou plusieurs domaines de protection. Pour des raisons de cohérence de fonctionnement et de protection des processus, l'accès aux ressources et aux informations du système doit être possible uniquement via les interfaces des abstractions fournies par le système d'exploitation. Il est donc nécessaire de protéger le système et de contrôler les appels à ses interfaces. Mais comment garantir cela sachant que le système ne peut avoir qu'une confiance limitée dans les processus ?

Bien que la protection puisse être obtenue par des techniques logicielles, par exemple en se basant sur des langages de programmation sûrs, la manière la plus sûre et la plus efficace consiste à utiliser les mécanismes de protection matériels fournis par le processeur. À cet effet, le processeur fournit trois mécanismes que nous avons déjà abordés : le mode superviseur, l'unité de gestion de la mémoire et l'appel système. Ces mécanismes sont suffisants pour assurer cette protection. Premièrement, un domaine de protection réservé au système d'exploitation garanti que le système ne sera pas corrompu ni espionné de façon volontaire ou involontaire par les processus s'exécutant dans leur propre domaine de protection. Deuxièmement, les exécutions du système en mode superviseur et des processus en mode utilisateur garantissent que les processus ne peuvent accéder directement aux ressources matérielles et en particulier aux informations privilégiées du processeur. Troisièmement, le seul point d'entrée dans le système et donc le seul moyen de passer en mode superviseur étant l'appel système, le système n'a plus qu'à vérifier à ce moment, la validité des appels à ses interfaces pour garantir la protection du système.

Organisation de la mémoire d'un processus

Nous venons de voir que le concept de mémoire virtuelle est utilisé par le système d'exploitation pour garantir la protection du système. Voyons maintenant comment peut être défini l'espace d'adressage d'un processus. Cet espace est habituellement découpé en deux parties, une pour le domaine de protection du système d'exploitation et une pour le domaine de protection du processus. Chaque partie est appelée un *segment* et il est possible de définir des droits d'accès sur chaque segment. Le segment du système d'exploitation, placé en début ou en fin de l'espace, est commun à chaque processus. Ce segment n'est pas accessible par le processus. L'intérêt de ce découpage est de rendre le partage d'information entre le système d'exploitation et le processus aussi efficace que possible. À l'intérieur de son segment, un processus fait ce qu'il veut, voir figure 2.5. Ce segment peut être découpé en plusieurs segments, par exemple, un pour le code, un pour la pile, un pour les données statiques et un pour les données dynamiques. La même technique peut aussi être utilisée pour les librairies partagées entre différents processus.

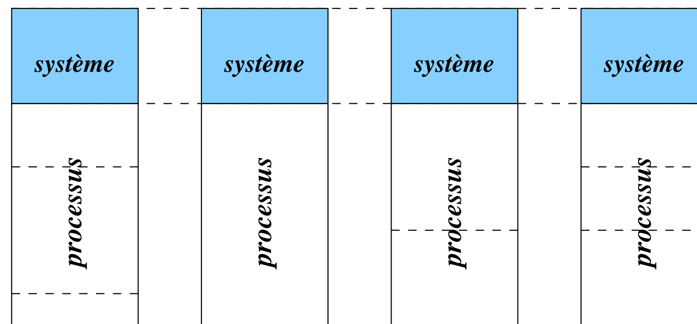


FIGURE 2.5 – Exemple de partitions d'espaces d'adressage

2.2.4 Machines virtuelles

$$\textit{machine physique} \stackrel{virtualisation}{\longrightarrow} \textit{machine virtuelle}$$

Nous avons vu qu'une machine virtuelle est la virtualisation complète d'une machine physique. L'interface offerte par une telle machine peut ou non être la même ou être identique à celle de la machine physique sous-jacente.

Interface différente de la machine physique

Une telle machine virtuelle est utilisée à des fins de portabilité ou de mobilité, par exemple des applications. Selon cette définition, le système d'exploitation est une machine virtuelle, car il transforme une configuration matérielle en une configuration virtuelle. Une machine virtuelle peut aussi interpréter le code binaire des programmes, c'est le cas pour la machine virtuelle Java (JVM) [Lindholm et Yellin 1999]. L'objectif de cette machine est de programmer les applications en langage Java [Gosling et al. 2000a] indépendamment du système d'exploitation sous-jacent. En revanche, la JVM s'exécute dans un processus du système et elle est écrite spécifiquement pour lui. Le problème de l'interprétation est son coût. Même s'il est possible, grâce à un compilateur « just-in-time » (JIT), de transformer dynamiquement en code machine le code interprété.

Une autre utilisation possible de la virtualisation des machines est proposée par le projet Cellular Disco [Govil et al. 1999] à l'université de Stanford. L'objectif est de transformer une machine multiprocesseurs à mémoire partagée en une grappe virtuelle supportant le confinement des fautes et l'hétérogénéité, tout en limitant le goulot d'étranglement lors du passage à l'échelle du système d'exploitation. La structure cellulaire de Cellular Disco confine les fautes matérielles à la cellule où elle se produit, voir figure 2.6. Au-dessus de la machine virtuelle peut s'exécuter le système d'exploitation qui requiert très peu de modifications sur son code source.

Interface identique à la machine physique

Une telle machine virtuelle est utilisée à des fins de partage. L'intérêt principal de cette virtualisation est de pouvoir exécuter un système d'exploitation dans le processus d'un autre système. Le premier système d'exploitation ayant fourni une telle machine virtuelle a été CP/67 sur les IBM 360/67.

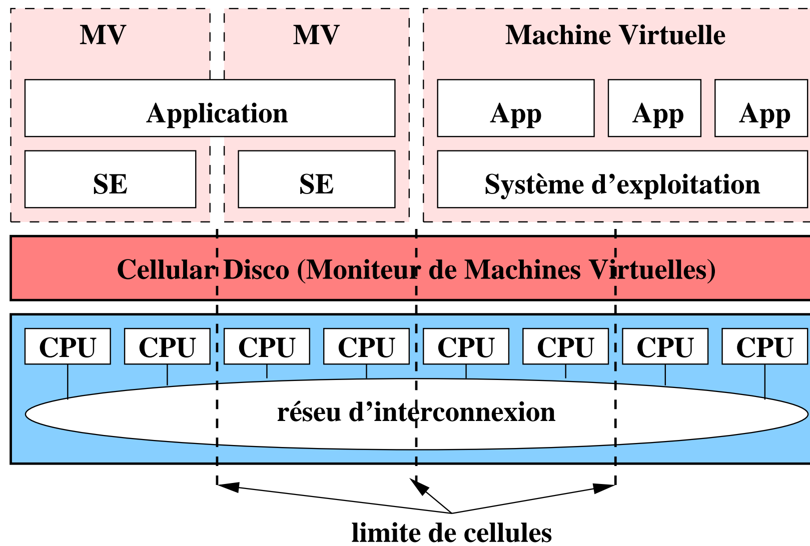


FIGURE 2.6 – Architecture de Cellular Disco

CP/67 fournissait à chaque utilisateur un IBM 360/65 virtuel, incluant les périphériques d'entrées-sorties. Le système d'exploitation commercial IBM VM/370 [Seawright et MacKinnon 1979] est dérivé du CP/67.

La plate-forme virtuelle VMware [VMware] réalisée plus récemment est une couche logicielle qui émule une machine de type x86 sur un système d'exploitation. C'est donc une machine virtuelle qui permet d'exécuter simultanément un ou plusieurs systèmes d'exploitation clients dans une fenêtre ou en plein écran sur un système d'exploitation hôte, tout en disposant de toutes les fonctionnalités du système hôte. VMware simule l'exécution simultanée des systèmes clients BSD, Dos, Linux et Windows dans un système Linux ou Windows, voir figure 2.7.

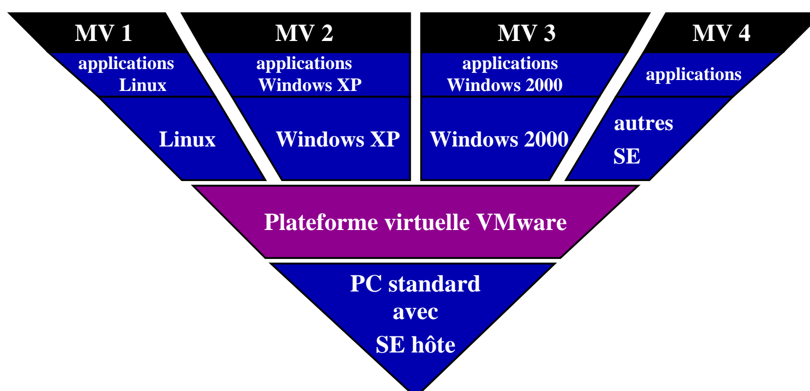


FIGURE 2.7 – Plateforme virtuelle VMware

2.3 Concepts d'utilisation des ressources

Le rôle d'un système d'exploitation est d'abstraire les ressources en vue de simplifier leur utilisation par les processus. Mais il ne suffit pas de connaître l'existence d'une ressource pour pouvoir l'utiliser. La question est de savoir comment un processus peut et doit utiliser les ressources proposées par le système d'exploitation.

Un processus obtient les ressources en les demandant au système qui les lui alloue dans la mesure du possible. Cette allocation permet au système de partager l'ensemble des ressources entre les différents processus et elle renvoie généralement au processus un nom lui permettant d'accéder à la ressource. Les noms sont utilisés par le système et les processus pour désigner les ressources qui les composent. Comme une ressource peut être utilisée simultanément par plusieurs processus, il est nécessaire de synchroniser les accès à celle-ci. L'opération de synchronisation coordonne les accès à une ressource en vue de garantir son intégrité.

2.3.1 Désignation et liaison

Un système utilise des noms pour désigner les ressources qui le composent [Krakowiak 1985]. Le *nom* d'une ressource est une information qui a une double fonction : (i) *désignation* de la ressource, c'est-à-dire la distinguer des autres ressources du système ; (ii) création d'une *liaison* pour pouvoir accéder la ressource dans le système. Ces différentes notions sont précisées dans les sections qui suivent.

Noms

Une ressource est généralement désignée par plusieurs noms, selon le niveau d'abstraction auquel on se place [Balter et al. 1991]. Par exemple, un fichier peut être désigné d'au moins quatre manières différentes : par un nom symbolique, par un numéro attribué lors de son ouverture, par l'adresse d'un descripteur en mémoire et par l'adresse d'un descripteur sur disque.

En pratique, la fonction de désignation associe un *nom symbolique* (par exemple une chaîne de caractères) et un *nom interne* interprétable par les couches basses du système ou directement par le matériel. Ce nom interne est souvent une adresse, ou encore un identificateur unique à partir duquel le système peut retrouver l'adresse par l'association statique ou dynamique à un nom de niveau inférieur. La suite de relations passant d'un nom symbolique à une ressource qu'il désigne s'appelle la *résolution de nom*.

Contexte de désignation

Les noms n'existent pas de manière absolue, ils font généralement référence à un *contexte de désignation*. Ce dernier fait référence à une autorité de gestion qui est responsable de la forme et de la création des noms. Cette notion s'applique aussi bien aux noms internes qu'aux noms symboliques. Les contextes de désignation peuvent être hiérarchisés, c'est-à-dire qu'ils sont relatifs à d'autres contextes. C'est par exemple le cas pour les répertoires dans un système de fichiers. Un contexte de désignation peut aussi restreindre l'ensemble des noms utilisables à un instant donné par un processus, notamment pour des raisons de protection ou d'efficacité [Balter et al. 1991, Krakowiak

1985]. En effet, si un processus ne peut désigner une ressource, il ne pourra y accéder et on évite ainsi des vérifications coûteuses à l'exécution.

Liaison

Une liaison est un canal de communication permettant à un programme d'interagir avec une ressource. Une liaison peut éventuellement être composite et être composée de plusieurs sous liaisons. Selon le moment de liaison, une liaison peut être statique ou dynamique. Dans le premier cas, la liaison est fixée une fois pour toutes, soit lors de l'écriture du programme, soit dans une phase de chargement et d'édition de liens, préalable à l'exécution. Dans ce cas, le nom fait aussi office de liaison, comme par exemple un nom langage. Dans le second cas, la liaison est réalisée au moment de l'exécution, soit lors du premier accès, soit à chaque accès.

La liaison dynamique est nécessaire quand les informations nécessaires à la liaison sont connues uniquement à l'exécution. Par exemple, quand les ressources sont créées dynamiquement. Une méthode générale pour réaliser la liaison dynamique repose sur l'*indirection* à travers un descripteur dont le nom est connu statiquement et dont le contenu est modifiable.

2.3.2 Allocation

L'*allocation* de ressource à un processus est implantée par un composant logiciel appelé *allocateur*. Absolument toutes les ressources sont contrôlées par un allocateur, sachant que toutes les ressources matérielles sont allouées au système lors de son initialisation. L'utilisation d'une ressource peut être découpée en trois phases. Au préalable, une phase d'*acquisition* permet d'allouer la ressource au processus demandeur. Deuxièmement, la phase d'utilisation proprement dite. Finalement, une phase de *libération* permettant de rendre la ressource. En fait, un processus peut utiliser une ressource uniquement si elle lui a été allouée par le système d'exploitation. C'est-à-dire si le système a donné au processus des moyens pour accéder à la ressource et qu'il lui a donné un nom ou directement une liaison.

L'allocation peut être temporelle, c'est-à-dire que le processus utilisant la même ressource va changer au court du temps, c'est par exemple le cas pour le processeur. L'allocation peut aussi être un découpage de la ressource, c'est-à-dire que la ressource va être découpée en plusieurs ressources plus petites qui pourront être utilisées par différents processus, le découpage s'effectue selon l'*unité d'allocation* de la ressource. L'objectif de l'allocation est donc d'utiliser au mieux la ressource selon certaines politiques, par exemple le respect de certaines contraintes de qualité de service.

Acquisition

L'acquisition d'une ressource par un processus peut être explicite ou implicite. Si l'acquisition est implicite, c'est le système qui alloue automatiquement la ressource au processus. Par exemple, lors du démarrage d'un processus, le système lui alloue implicitement de la mémoire, pour ses données et ses instructions, et du temps processeur permettant de commencer son exécution. Si l'acquisition est explicite, l'acquisition doit être explicitement formulée sous la forme d'une requête à l'allocateur de la ressource.

La réaction de l'allocateur d'une ressource à une requête peut prendre différentes formes. Soit

la requête peut être satisfaite et la ressource est allouée. Soit la requête ne peut pas être satisfaite, par exemple, si la ressource ou tous ses fragments sont déjà alloués. L'allocateur peut alors refuser l'allocation ou mettre en attente le processus jusqu'à ce qu'une ressource équivalente soit libérée par un autre processus.

Libération

À la fin de l'utilisation de la ressource, le processus doit la libérer. Cette libération peut être explicite ou implicite. La libération explicite est le dual de l'allocation explicite. Par exemple les ressources d'un processus sont libérées automatiquement sans code spécifique, par le système, à la terminaison du processus. La libération implicite peut aussi être une *réquisition* ou une *préemption*, c'est à dire un retrait forcé de la ressource par l'allocateur. C'est par exemple ce que fait un ordonnanceur lorsqu'il retire le processeur à un fil d'exécution pour l'allouer à un autre. Cette réquisition est aussi utilisée dans les algorithmes de mémoire virtuelle.

Unité d'allocation

L'*unité d'allocation*, ou le *grain d'allocation*, est la taille avec laquelle une ressource peut être découpée. S'il s'agit d'une ressource physique, c'est généralement le matériel qui fixe cette taille. Par exemple, la mémoire physique est partitionnée en pages dont la taille est fixée par la MMU. Par exemple sur les processeurs de 32 bits, elle est généralement de 4 Ko. Le disque est fragmenté en bloc de 512 octets. Le réseau est fragmenté en trames d'une taille bornée. Le temps processeur peut être découpé de façon arbitraire en *quantum de temps*. Par exemple sur Linux il est de 100 Hz soit $1/100^{eme}$ de seconde. L'unité d'allocation peut aussi être de taille variable. C'est le cas pour un allocateur de blocs de mémoire ou pour certains ordonnanceurs.

2.3.3 Synchronisation

Nous avons vu qu'un système consiste en un ensemble de processus, ces processus sont eux-mêmes composés d'un ou plusieurs fils d'exécution. Ces fils d'exécution peuvent partager des ressources et accéder de façon concurrente à celles-ci. Lors de l'accès concurrent à des ressources il se pose des problèmes d'incohérence. Il faut donc mettre en œuvre des mécanismes de synchronisation qui permet d'ordonnancer de façon séquentielle les fils d'exécution. Il existe plusieurs mécanismes de synchronisation que nous allons voir ci-dessous. Il faut noter que la synchronisation est inutile s'il n'y a qu'un seul fil d'exécution. La synchronisation est aussi utilisée en interne dans le système d'exploitation pour sérialiser ses accès concurrents aux ressources.

Exclusion mutuelle

Le problème de l'exclusion mutuelle a été formulé par [Dijkstra 1965a]. L'*exclusion mutuelle* consiste à étendre à des séquences d'actions la propriété d'indivisibilité des actions du niveau de base [Krakowiak 1985]. La propriété d'indivisibilité est à la base des mécanismes de synchronisation. La séquence de code exécutée en exclusion mutuelle est appelée une *section critique*. L'exclusion mutuelle est utilisée pour garantir qu'une ressource n'est pas accédée par plus d'un fil d'exécution simultanément.

De nombreux algorithmes d'exclusion mutuelle, nommés *mutex*, ont été présentés. Le premier algorithme à attente active pour 2 processeurs est celui de Dekker [Dijkstra 1965a] reposant uniquement sur l'indivision des accès en écriture ou en lecture, une version simplifiée a été proposée par Peterson [Peterson 1981]. Des algorithmes pour n processeurs sont par exemple ceux de [Dijkstra 1965b] et [Lamport 1974]. Une discussion complète du problème de l'exclusion mutuelle a été proposée plus récemment par [Lamport 1986, Lamport 1991].

Sémaphores

Un autre mécanisme de synchronisation suggéré et utilisé par [Dijkstra 1965a, Dijkstra 1968] est le *sémaphore*, il supprime le problème de l'attente active. Le sémaphore est l'association d'un compteur et d'une file d'attente. La file sert à bloquer les processus en attente du signal d'une condition de réveil. L'interface offerte par les sémaphores est `P` (attente) et `V` (signal). Le compteur est décrémenté lors d'une attente et incrémenté lors d'un signal. L'attente est effective uniquement si le compteur devient négatif. Ainsi, si un signal est exécuté par le processeur avant l'attente, celle-ci ne sera pas bloquante.

Moniteurs

L'utilisation du sémaphore suppose le respect de règles, par exemple un `P` puis un `V`, formant des règles de programmation auxquelles les programmeurs peuvent ne pas obéir. Dans ce cas, aucune synchronisation n'est garantie.

Une solution à ce problème présenté par [Brinch Hansen 1973] et par [Hoare 1974] est le *moniteur*. L'objectif du moniteur est d'intégrer avec les interfaces de la ressource et de façon implicite le code nécessaire à la synchronisation et d'assurer automatiquement le verrouillage lors d'une invocation de méthode. Le moniteur assure que seul un fil d'exécution pourra s'exécuter dans les méthodes de l'interface du moniteur. En fait, le code assurant la synchronisation est ajouté par le compilateur dans l'en-tête des méthodes. Les fils d'exécution peuvent aussi être synchronisés explicitement par l'utilisation de condition et des méthodes `wait` et `signal`. Le moniteur est utilisé dans de nombreux langages à haut niveau, citons Modula [Nelson 1991] et Java [Gosling et al. 2000a].

Interblocage

Le problème majeur rencontré avec l'exclusion mutuelle est l'*interblocage* « deadlock ». Cette situation se produit par exemple lorsqu'un fil d'exécution doit entrer dans une section critique détenue par un autre et que ce dernier doit lui-même entrer dans une section critique détenue par le premier.

2.4 Principes d'organisation des systèmes

Le rôle d'un système d'exploitation est, d'une part offrir des abstractions de ressources pour simplifier leurs utilisations, d'autre part de multiplexer les ressources entre les différents processus. La question maintenant est de savoir comment il est possible de les construire, de les organiser et de les composer. La *composition* est l'opération permettant d'assembler et de mettre en relation les

différentes entités en vue de réaliser le système voulu. Les questions posées par la composition sont d'une part les entités à utiliser et d'autre part comment elles seront assemblées.

Cette section présente les cinq styles de structuration des architectures logicielles ; structuration en couches, structuration orientée objets, structuration orientée événements, structuration par flot de données. Un style d'architecture est un ensemble de règles de composition, elles sont généralement spécifiques à un domaine d'application. L'intérêt d'un style d'architecture est de simplifier le développement des applications dans un domaine donné. Pour de plus amples informations sur les styles d'architecture on consultera [Garlan et Shaw 1994, Shaw et Garlan 1996]. Cette section présente aussi les langages de description d'architecture, ils permettent de décrire les composants et la composition des systèmes.

2.4.1 Structuration en couches

La *structuration en couches* consiste à structurer le système en différentes couches de 0 à n. Dans une structuration stricte, chaque couche $i_{i>0}$ s'appuie sur celle qui lui est immédiatement inférieure $i-1$ et exploite la notion d'abstraction. Néanmoins, dans une structuration moins stricte, une couche $i_{i>0}$ peut appeler une couche $i - x_{1<x<i}$. Cette conception descendante consiste à décomposer le problème en une succession de sous problèmes plus élémentaires que l'on espère résoudre plus aisément [Krakowiak 1985]. La spécification d'une couche pour ses utilisateurs se réduit à celle de son interface. Cela permet de séparer l'implantation de la couche de son utilisation.

Appels intercouches

Dans un système à couches, les appels sont généralement possibles uniquement entre deux couches successives. Le problème majeur de ce découpage est le coût engendré par la traversée successive des couches. L'appel d'une couche supérieure vers une couche inférieure est appelé un *appel descendant*, l'inverse est appelé un *appel ascendant* « upcall », voir figure 2.8.

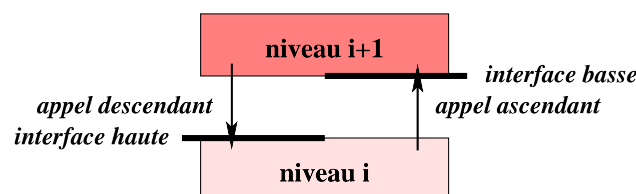


FIGURE 2.8 – Appel descendant et appel ascendant

Couches logicielles

Le premier système à utiliser cette technique a été le système multiprogrammé THE développé à l'université d'Eindhoven [Dijkstra 1968]. Ce système est organisé en six couches logicielles, voir figure 2.9. Les couches du système étaient uniquement utilisées durant la programmation et n'apparaissent pas à l'exécution.

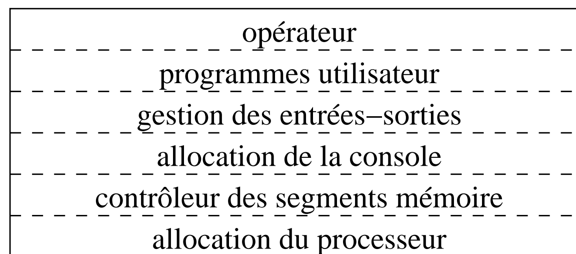


FIGURE 2.9 – Couches du système THE

Couche d'abstraction du matériel

La structuration en couche est aussi utilisée en vue de réaliser la portabilité des systèmes, on parle de couche « Hardware Abstraction Layer » (HAL). Cette couche réifie de façon homogène et portable les abstractions offertes par le matériel. Évidemment, l'implantation d'une couche HAL dépend du matériel sous-jacent mais ceci est masqué dans l'interface homogène qu'offre la HAL. Néanmoins, cette uniformisation ne permet pas la réification des particularités de chaque machine. Nous pouvons citer deux systèmes utilisant ou ayant utilisé des couches HAL : eCos de RedHat [eCos], Windows NT de Microsoft. Il faut noter que, pour des raisons d'optimisation de performance, Windows NT n'utilise plus de couche HAL dans les versions récentes.

Couche noyau

Un *noyau* est un ensemble éventuellement minimum de fonction d'un système d'exploitation sur lequel s'appuient les autres fonctions du système d'exploitation. Pour des raisons de protection, le noyau s'exécute généralement dans un domaine de protection propre et dans le mode superviseur du processeur ce qui lui donne tous les droits d'accès aux ressources. Le noyau peut être vu comme une couche entre les processus exécutant des applications ou des fonctions systèmes et le matériel, voir la figure 2.10. Ici, l'appel entre un processus et le noyau est appelé un *appel système* « syscall », l'appel inverse est appelé un *signal*. Un appel entre le noyau et le matériel est une entrée-sortie au sens large du terme (manipulation de registres, d'horloge, etc.). L'appel inverse est une interruption.

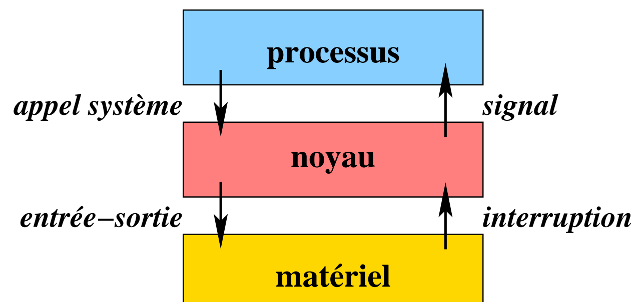


FIGURE 2.10 – Noyau et appels dans un système d'exploitation

2.4.2 Structuration orientée objets

La *structuration orientée objets* consiste à structurer le système en différents éléments. Ces différents éléments, qui communiquent entre eux, sont assemblés en vue de réaliser le ou les services que le système doit rendre. Différentes terminologies existent pour appeler ces éléments. Ils sont appelés des composants dans les systèmes à composants, des objets dans les intergiciels de type CORBA et dans le modèle de référence ODP, etc. Quelle que soit l'appellation, le rôle est le même et consiste à offrir un service clairement identifié, tel que la gestion mémoire, les pilotes de périphériques, les protocoles réseaux, etc. Pour la suite du discours, nous appelons ces éléments des composants.

la structuration orientée objets n'implique pas l'utilisation d'un langage de programmation à objets. En particulier, la structuration orientée objets ne requiert pas la notion d'héritage offerte par les langages de programmation à objets. Il est néanmoins possible d'utiliser ces langages pour la programmation des systèmes à objets.

Composants

La structuration orientée objets permet d'identifier clairement les différents composants qui composent un système. Par définition, un composant est l'unité de réutilisation et de construction des systèmes et des applications. Nous utilisons le terme composant avec la sémantique française habituelle qui signifie par définition qu'il peut être assemblé avec d'autres composants, voir la figure 2.11. Contrairement à la structuration en couche, l'assemblage avec la structuration orientée objets forme un graphe qui peut potentiellement comporter des cycles.

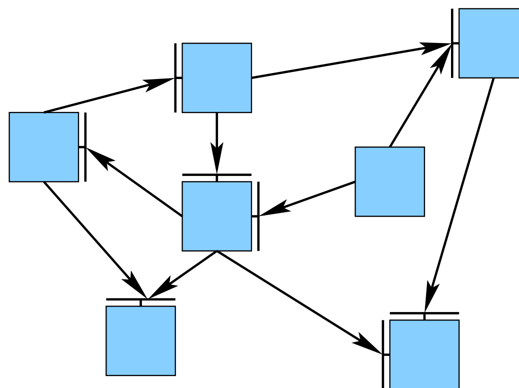


FIGURE 2.11 – Exemple de structuration orientée objets

L'approche par composants offre une grande souplesse dans la conception des systèmes en permettant de développer de manière itérative et de gérer les changements qui peuvent survenir. Cette évolution des composants est possible en utilisant la notion d'interface qui offre l'indépendance entre l'utilisation et l'implantation des composants. C'est pour cette raison que la structuration orientée objets est naturellement la base de la flexibilité. Cette approche autorise la réutilisation des composants entre différents systèmes. Les bibliothèques partagées sont des exemples de composants.

Un composant se caractérise généralement par l'encapsulation. L'*encapsulation* est la propriété spécifiant que les informations contenues dans un composant sont accessibles uniquement au travers

des interfaces qu'il supporte. Comme un composant est encapsulé, une interaction avec un composant ne peut pas modifier l'état d'un composant tiers sans une autre interaction avec celui-ci. Ainsi, chaque changement d'état d'un composant peut se produire uniquement comme résultat d'une action interne du composant ou comme résultat d'une interaction du composant avec son environnement.

Modèle client-serveur

Le *modèle client-serveur* est le modèle le plus utilisé pour la structuration orientée objets des applications et des systèmes centralisés ou les infrastructures logicielles réparties. Dans ce modèle, le serveur fournit un ensemble de services exécutables à des clients. Les clients effectuent des requêtes pour demander l'exécution d'un service. Le rôle d'un serveur est donc d'exécuter et de répondre aux requêtes émises par les clients, voir figure 2.12. L'implantation d'un serveur est arbitraire, il peut exécuter simultanément ou de façon séquentielle plusieurs requêtes provenant de différents clients. Le modèle client-serveur est généralement implanté en utilisant l'appel de méthode que nous aborderons à la section 2.6.

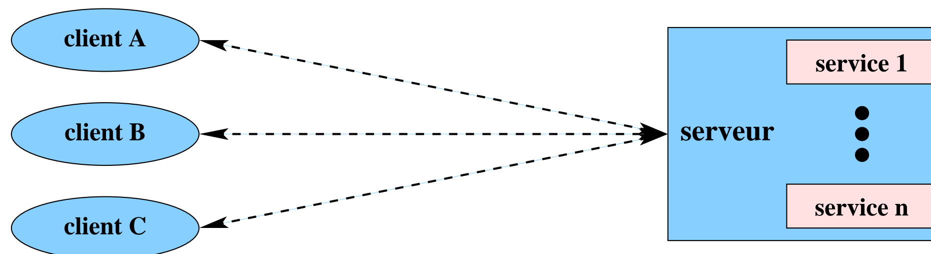


FIGURE 2.12 – Modèle client-serveur

2.4.3 Structuration orientée événements

Dans la *structuration orientée événements* un composant diffuse des événements. D'autres composants du système peuvent réagir à cet événement. La différence fondamentale avec la structuration orientée objets est qu'un composant ne sait pas quels autres composants peuvent être affectés par ses événements. Nous reviendrons sur le modèle de communication par événements à la section 2.6.4.

2.4.4 Structuration par référentiel

Deux sortes de composants dont présents dans la *structuration par référentiel*, une structure de données centrales qui représente l'état courant, et une collection de composants indépendants qui fonctionnent sur les données centralisées.

2.4.5 Structuration par flot de données

La *structuration par flot de données* est une variation de la structuration orientée objets, elle est utilisée lorsque chaque composant possède un ensemble d'entrée et de sortie. Un tel composant est

appelé un *filtre*. Un filtre lit un flot de données sur ses entrées, il effectue un traitement local et il produit un flot de données sur ses sorties. Le lien, entre la sortie d'un filtre et l'entrée d'un autre, est appelé un *connecteur*. La composition des filtres réalise la fonctionnalité du système, voir figure 2.13. Le graphe résultant de cette composition peut être une file, un arbre ou même un graphe.

FIGURE 2.13 – Structuration par flot de données

x-Kernel [Hutchinson et Peterson 1991] propose un style d'architecture, dirigée par le flot de données, pour la construction de protocoles réseaux. Le canevas logiciel proposé, composé d'une interface uniforme pour tous les protocoles, permet de les remplacer, sous réserve qu'ils aient la même sémantique, par exemple de contrôle de flux. Une généralisation du principe de x-Kernel est faite dans le système Click [Morris et al. 1999]. Ce système propose un canevas pour la composition de routeurs. Ce canevas est essentiellement basé sur la notion de `push` et de `pull` entre des composants implantant des queues de messages et des composants de traitement.

2.4.6 Langages de description d'architecture

Un langage de description d'architecture « Architecture Definition Language » (ADL) permet de décrire et de manipuler la structure d'un système. L'objectif d'un tel langage déclaratif est de faciliter la réutilisation et l'assemblage des composants. D'une part, le langage permet de décrire les dépendances en termes d'interfaces utilisées et d'interfaces fournies par un composant. D'autre part, le langage permet de décrire les composants requis par un système, éventuellement comment ils sont assemblés, et les options de configuration offertes par un composant. Par exemple, une option de configuration peut juste être de dire le nombre de priorités que propose un ordonnanceur à priorité.

Les outils associés à ce langage effectuent la vérification de la sémantique d'assemblage. Par exemple, en vérifiant la conformité des interconnexions entre les interfaces fournies et utilisées. Il est aussi possible de vérifier des contraintes, globales ou déclarées par les composants, sur le fonctionnement du système. Par exemple, un composant prévu pour fonctionner avec un seul fil d'exécution ne doit pas être utilisé avec plusieurs. Les outils facilitent aussi l'installation et l'évolution du système, soit en automatisant le processus d'installation, soit en générant directement l'exécutable du système.

Knit [Reid et al. 2000] fournit un langage et des outils de description d'architecture pour la composition statique de systèmes. L'objectif est essentiellement d'offrir des outils permettant d'assembler statiquement un système à partir de composants standards écrits en code C. Le langage offre une syntaxe permettant de décrire pour chaque composant ses dépendances, d'importation et d'exportation, exprimées en termes de symboles langages. Il permet aussi de décrire les composants formant le système, ainsi que les contraintes d'assemblages, comme ne pas appeler sous contexte d'interruption une méthode pouvant potentiellement se bloquer. Les outils fournissent l'ordonnancement automatique de l'initialisation et de terminaison des composants et la vérification des contraintes. En cas de conflit de symboles entre deux composants, les outils peuvent renommer automatiquement les symboles.

Un autre exemple de langage de description d'architecture est proposé avec le système embarqué eCos [eCos]. eCos fournit un ensemble d'outils graphiques de développement et de configuration du système. Ces outils permettent principalement la configuration des options de configuration de

composant et la vérification des contraintes d'assemblage du système. Les technologies de conteneurs, EJB [Thomas 1998] et CCM [OMG 1999], proposent aussi un langage de description d'architecture permettant la composition des composants.

2.5 Interposition

Une des techniques clés en informatique est l'*interposition*. Elle est utilisée dans une multitude de mécanismes systèmes très divers, comme le talon pour la communication, le mandataire pour adapter les serveurs, la réflexion procédurale et les sous-contrats pour la spécialisation, ou encore les conteneurs pour l'encapsulation. Ces exemples sont détaillés par la suite. L'interposition consiste à ajouter une indirection sous forme d'une couche logicielle supplémentaire entre un composant appelant et un composant appelé, voir figure 2.14. Le rôle de l'interposition est d'intercepter les appels entre le composant appelant client et le composant appelé serveur. Ce qui est possible grâce à la notion d'interface. De plus, grâce à l'indépendance entre l'interface et l'implantation, l'interception est transparente pour l'appelant et l'appelé. C'est-à-dire que les codes de l'appelant, de l'appelé et de l'interface exportée par le composant appelé sont inchangés. Bien évidemment, cette construction est complètement récursive et il est possible d'interposer successivement plusieurs couches.

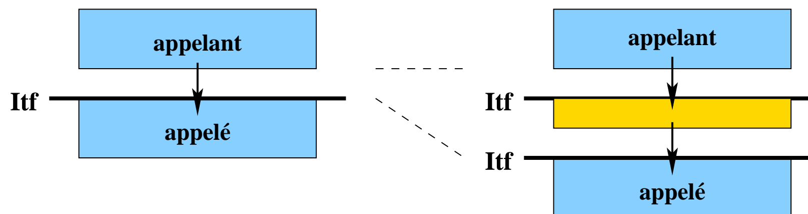


FIGURE 2.14 – Fonctionnement de l'interposition

Les exemples d'utilisation de l'interposition sont très nombreux : traçage des appels, statistiques, détournement des appels, etc. Bien évidemment, un composant d'interposition est spécifique à une interface. En conséquence, si un programmeur désire ajouter une fonction particulière à toutes les interactions du système, par exemple le traçage des appels, il doit développer un composant spécifique pour chaque type d'interface. Généralement, ces composants d'interposition sont générés de façon automatique, statiquement ou dynamiquement, par des outils spécifiques.

L'utilité de l'interposition ne se limite pas à la collecte d'informations. En effet, il est possible d'utiliser l'interposition pour mettre en œuvre des mécanismes complexes comme la communication entre sites distants, la spécialisation du système, la réflexion du comportement du système. Ces utilisations sont détaillées dans les sections qui suivent.

2.5.1 Proxy

L'interposition se réalise généralement par un mécanisme de *proxy* [Shapiro 1986]. Le proxy est un composant qui s'interpose entre un client et un serveur, il permet de simuler localement un composant distant. Lorsqu'un client appelle un serveur, le proxy intercepte l'appel. L'interface du proxy doit donc théoriquement être la même que celle du serveur. Il existe de nombreuses instances

du proxy ; talon, mandataire, réflexion, conteneur, sous-contrats, que nous présentons ci-dessous.

Talon

L'interposition est aussi utilisée pour réaliser l'*appel de procédure distante* « Remote Procedure Call » (RPC). L'idée du RPC est d'utiliser l'appel de procédure pour programmer les applications réparties. L'objectif est donc d'exécuter un service présenté sous la forme d'une procédure située sur un site distant. La difficulté est d'obtenir la forme et les effets identiques à ceux d'un appel local. Par exemple en cas un service distant n'est pas forcément joignable, ce qui ne peut pas être le cas en centralisé. Le RPC est mis en œuvre en utilisant la notion de *talon*. Un talon assure le rôle d'un représentant distant, voir figure 2.15.

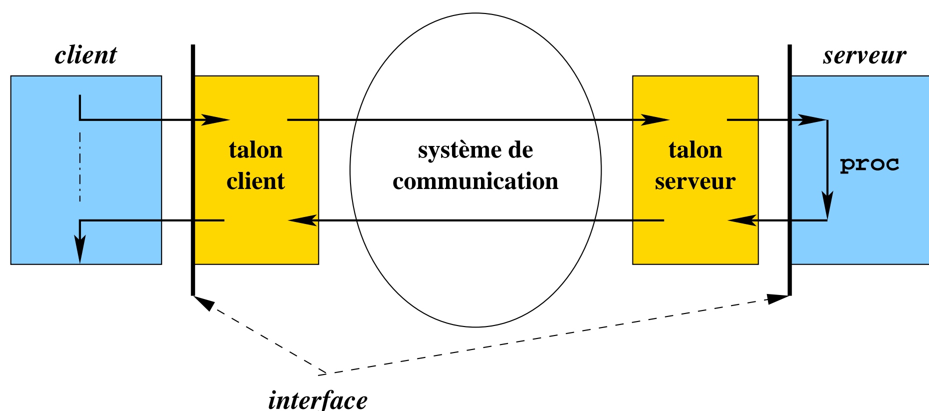


FIGURE 2.15 – Utilisation du talon dans l'appel de procédure distante

Deux talons distincts sont donc utilisés lors d'un appel distant, un du coté client représentant le serveur et un du coté serveur représentant le client. Lorsque le client appelle le serveur, le talon client emballe les paramètres et les envoie au talon serveur. Le talon serveur déballe les paramètres, reconstruit un appel local et appel la procédure cible. La procédure s'exécute, le talon serveur récupère les résultats, les emballe et les retourne au talon client. Ce dernier déballe alors les résultats et les retourne au client comme dans un retour de procédure local.

Généralement, et c'est là tout l'intérêt de l'approche, les talons sont générés par un compilateur à partir de l'interface de la procédure. La première proposition de RPC est celle de Xerox [Birrell et Nelson 1984]. La plus connue et la plus utilisée est SunRPC [Sun Microsystems Inc 1988] qui utilise un format unique de représentations des données « eXternal Data Representation » (XDR) [Sun Microsystems Inc 1987]. La même technique est utilisé dans les « Object Request Broker » (ORB), comme CORBA [OMG 2001] et Java « Remote Method Invocation » (RMI) [Sun Microsystems Inc 1999b].

Mandataire

Un exemple d'utilisation du proxy est le *mandataire*. Ce dernier permet d'adapter les serveurs et les systèmes de communications aux caractéristiques des clients. L'objectif est de programmer le serveur indépendamment des clients. Le mandataire s'interpose entre le client et le serveur (voir la

figure 2.16). Un mandataire est potentiellement partagé par plusieurs clients ayant les mêmes caractéristiques.

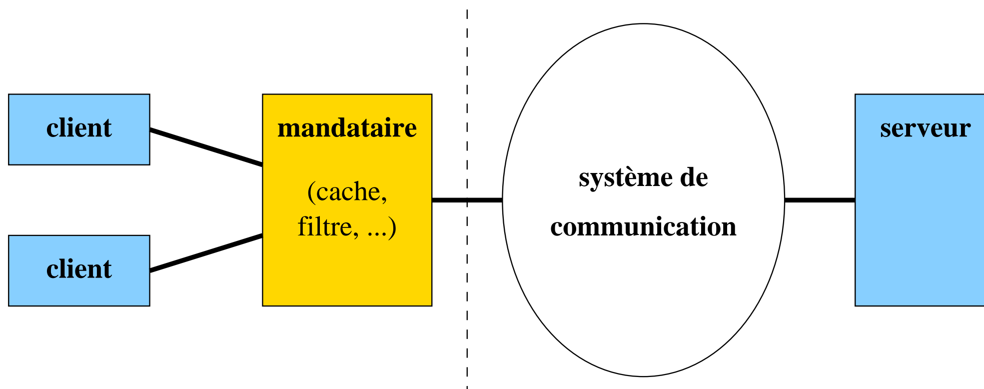


FIGURE 2.16 – Le mandataire

Pour diminuer l'impact du coût des communications avec un serveur, un mandataire peut implanter un cache mémorisant les dernières informations accédées par les clients du serveur. La localisation géographique d'un cache est arbitraire. Soit, il réside sur la même machine que le client, comme pour un système de gestion de fichiers répartis, tel que « Network File System » NFS [Sandberg et al. 1985]. Soit, il réside sur une machine intermédiaire, comme pour un cache Internet.

Le mandataire peut aussi être utilisé pour porter une application répartie complexe sur des clients légers, comme les PDA. La méthode consiste ici à reporter les fonctions coûteuses, qui ne peuvent être réalisées sur le client léger, dans un mandataire. Le but est par exemple de minimiser, soit la charge de travail restant en vue de pouvoir l'exécuter sur clients légers, soit le débit réseau requis par l'application, soit tout simplement d'adapter une donnée en fonction des contraintes matérielles, comme la taille de l'écran. Par exemple, un mandataire pour l'adaptation de flux vidéo de type MPEG dégrade la qualité de la vidéo, soit en diminuant la résolution de l'image, soit en réduisant le nombre d'images transmises par secondes [Hess et al. 2000].

Réflexion procédurale

Un autre exemple d'utilisation du proxy est pour mettre en œuvre la *réflexion procédurale* des appels de méthodes [Malenfant et al. 1996]. Cette réflexion permet de réifier les invocations sous la forme d'un objet utilisé comme paramètre dans l'invocation d'un méta-objet. Un *méta-objet* est un objet qui joue le rôle de contrôleur d'un ou de plusieurs autres objets, voir figure 2.17.

Des mécanismes de réflexion procédurale ont été introduits récemment dans Java avec les « proxy class » [Sun Microsystems Inc 1999a], voir figure 2.17. Un proxy est spécifique à une interface et il peut être généré dynamiquement. La réflexion procédurale peut aussi être introduite de façon native dans un système et transformer toutes les invocations en un appel à un méta-objet. Cela est réalisé soit statiquement en modifiant le compilateur, soit dynamiquement en modifiant l'environnement d'exécution du langage.

La réflexion a surtout été étudiée et mise en oeuvre dans des langages de programmation, comme Smalltalk, Clos ou Abcl/R. Cette technique logicielle est désormais appliquée dans les systèmes d'ex-

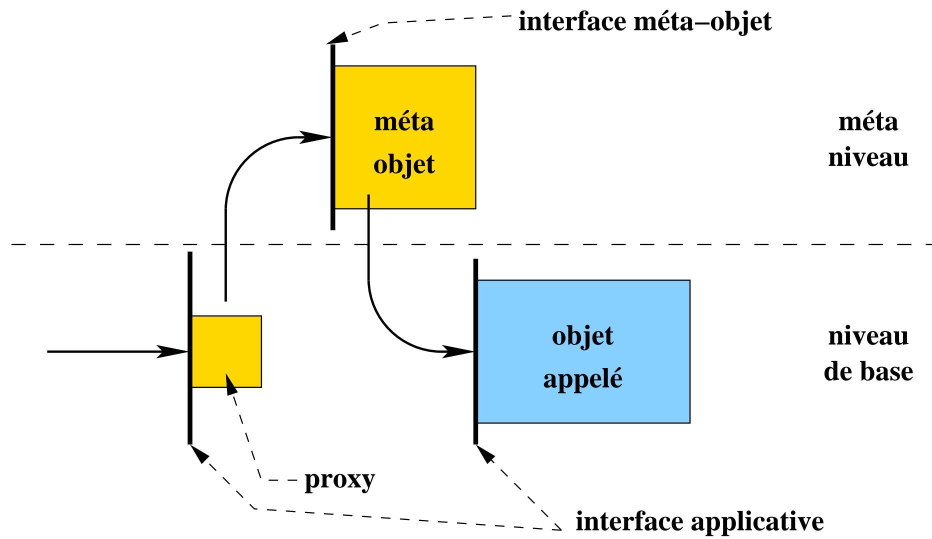


FIGURE 2.17 – Interposition pour la réflexion procédurale

ploitation, comme dans le système d'exploitation Apertos [Yokote 1992] qui est complètement architecturé selon le principe des méta-objets.

Conteneur

L'utilisation d'un proxy permet l'encapsulation d'un composant ou de plusieurs composants dans un autre composant que nous appelons un *conteneur*, voir figure 2.18. L'objectif de ce dernier est de prendre en charge les services systèmes requis par les composants, comme la désignation, la sécurité, les transactions, la persistance, etc. C'est donc un modèle pour associer des propriétés non fonctionnelles à des composants. Un conteneur est généralement conçu en utilisant les techniques de réflexion procédurale que nous venons de voir.

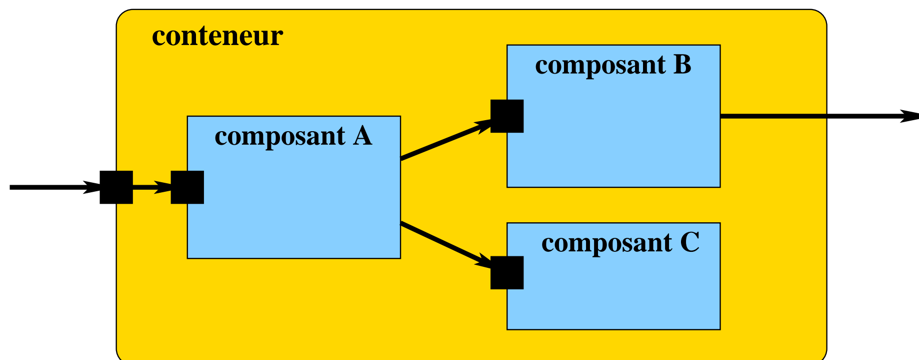


FIGURE 2.18 – Conteneur

Les principales technologies de conteneurs sont ; la technologie « Enterprise JavaBean » (EJB)

pour Java [Thomas 1998], la technologie « Corba Component Model » (CCM) [OMG 1999] et la technologie « Component Object Model » (COM) pour plates-formes Windows.

Sous-contrats

L'interposition peut aussi être utilisée pour spécialiser le comportement du système dans le but d'offrir un degré d'adaptabilité et d'extensibilité. Le *sous-contrat* permet de spécialiser le système en surchargeant les composants, voir figure 2.19. Cela permet de définir et d'ajouter des sémantiques différentes selon les besoins des applications ou du système, comme par exemple ajouter des propriétés non fonctionnelles aux composants ; synchronisation, persistance des composants, traçage des appels, prise de statistiques, etc. Ainsi, le composant ne contient que du code fonctionnel.

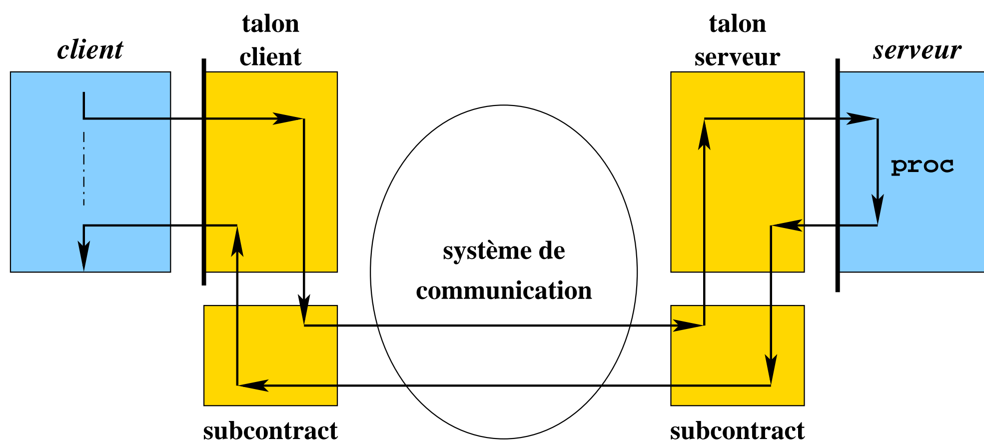


FIGURE 2.19 – Les « subcontract » dans Spring

Le sous-contrat a été proposé avec le système à objets Spring [Hamilton et Kougiouris 1993], sous le nom de « subcontract » [Hamilton et al. 1993]. Ils permettent de surcharger les talons utilisés lors de la communication distante, voir figure 2.19. La notion de « subcontract » a été reprise dans Java avec son mécanisme de sérialisation des objets pour JavaRMI [Sun Microsystems Inc 1998b, Sun Microsystems Inc 1999b].

2.6 Modèle de communication dans les systèmes

Nous allons étudier le problème de la communication engendré par la décomposition des systèmes selon les principes d'organisation que nous venons de voir. Il existe plusieurs modèles de communication utilisés dans les systèmes d'exploitation centralisés ou répartis ; par variable partagée, par appel de méthode, par message et par événement.

Cette section présente les différents modèles de communications existants. Nous examinons avec quels types d'interactions un modèle de communication est utilisable. Nous considérons les interactions entre deux processus situés sur une même machine ou entre deux processus situés sur des machines distantes, les interactions entre les processus et le système, les interactions à l'intérieur d'un même processus, et les interactions avec le matériel.

2.6.1 Modèle de communication par variable partagée

Le modèle de communication par variable partagée est un modèle permettant de programmer les communications entre les différentes tâches d'un système ou d'une application en utilisant une mémoire commune comme espace de communication. L'interface de programmation fournie par ce modèle de communication est du type `load` et `store`, voir figure 2.20. Ce modèle est utilisé par les fils d'exécution s'exécutant dans un même domaine de protection pour communiquer, mais pas uniquement comme nous allons le voir. Ce modèle de communication est particulièrement bien adapté pour tout ce qui relève du traitement sur des données. C'est par exemple le cas pour les calculs scientifiques sur des matrices.

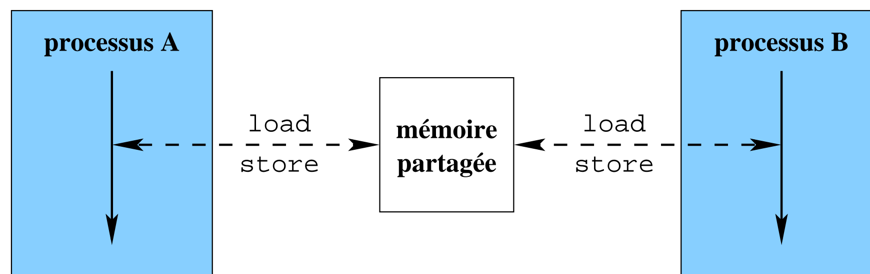


FIGURE 2.20 – Modèle de communication par variable partagée

Comme les fils d'exécution partagent la même mémoire, il peut être nécessaire de synchroniser les accès aux variables, voir section 2.3.3. Dans ce modèle, les noms peuvent être des adresses mémoire uniques si l'adressage est unique, c'est-à-dire qu'une donnée se trouve toujours à la même adresse quel que soit le processus. Sinon, il est nécessaire d'utiliser un espace de noms distincts nécessitant une traduction des noms lors du premier accès ou à chaque accès. Les mécanismes de communication par variables partagées sont les suivants.

Segment partagé

L'objectif du *segment partagé* est une abstraction permettant à deux processus ne partageant pas le même domaine de protection de partager de la mémoire. On dit qu'un segment est partagé entre deux processus si le couplage avec la mémoire physique est réalisé à la même adresse [Silberschatz et Galvin 1998]. L'adressage n'est pas forcément unique, le même segment peut se trouver à plusieurs adresses différentes dans les processus. Un segment partagé n'est pas forcément couplé de la même manière dans les processus, il peut être en lecture seule d'un coté et en écriture de l'autre.

Le segment partagé est très utilisé dans les systèmes d'exploitation. Par exemple, il permet d'implanter les tubes de communication « pipe » entre deux processus. Le segment partagé peut aussi être utilisé non pas entre deux processus, mais entre un processus et le système d'exploitation. Cette technique est par exemple utilisée pour rendre accessibles les informations du système d'exploitation. Le segment partagé peut aussi être utilisé pour partager du code, comme une librairie.

Mémoire virtuelle répartie

L'objectif de la Mémoire Virtuelle Répartie (MVR) est de fournir une abstraction permettant le partage de données entre différents processus ne partageant pas la même mémoire physique. La mémoire virtuelle répartie masque la répartition. Cela permet de programmer des systèmes ou applications répartis en utilisant le modèle de communication par variables partagées, comme dans une application centralisée.

Pour des raisons de performance des accès locaux et des contraintes physiques, chaque donnée utilisée par un processus est recopiée localement sur la machine d'exécution du processus. Si la donnée est dupliquée, on est alors confronté à des problèmes de cohérence. Il existe différents modèles de cohérence, la cohérence séquentielle, la cohérence causale, la cohérence relâchée basée sur les opérations de synchronisation définies explicitement par l'application. Cette dernière peut se décliner en plusieurs variantes ; la cohérence faible, la cohérence à la sortie [Gharachorloo et al. 1990] et la cohérence à l'entrée [Bershad et al. 1993]. Pour de plus amples informations, voir [Mosberger 1993]. Certains systèmes, comme Opal [Chase et al. 1994], offre un adressage mémoire unique sur 64bits. C'est-à-dire qu'une même donnée s'accède depuis une même adresse quelle que soit la machine.

2.6.2 Modèle de communication par appel de méthode

La programmation par appel de méthode est celle que nous utilisons tous les jours lorsque nous programmons dans des langages à objets, comme Java. Dans ce modèle de communication nous incluons aussi l'appel de procédure des langages procéduraux, comme le C. Un appel de méthode ou de procédure est synchrone, c'est-à-dire que l'appelant d'une méthode est bloqué tant que la méthode appelée n'a pas fini son exécution, voir figure 2.21. Les méthodes peuvent être appelées récursivement, ce qui nécessite l'utilisation d'une *pile d'exécution*. L'interface de programmation fournie par l'appel de méthode possède un nombre arbitraire de paramètres qu'il est nécessaire d'empiler ou de mettre dans des registres avant l'appel. Lors du retour de la méthode, un résultat peut être renvoyé soit en le recopiant sur la pile, soit en le mettant dans des registres.

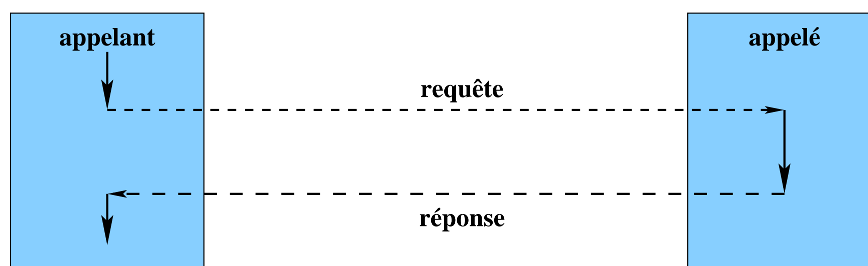


FIGURE 2.21 – Modèle de communication par appel de méthode

Intuitivement, nous voyons que l'appel de méthode se prête naturellement bien au modèle client-serveur. En effet, le modèle client-serveur peut être utilisé en local entre différents processus ou en réparti entre processus situés sur des sites distants. Il peut aussi être utilisé pour demander aux systèmes d'exploitation l'exécution de services. Le modèle client-serveur est omniprésent dans les systèmes d'exploitation centralisés et dans les infrastructures réparties.

Appel de procédure distante

La mise en œuvre du modèle client-serveur dans un système réparti s'effectue grâce au RPC, nous avons déjà abordé ce mécanisme à la section 2.4.2. L'objectif de ce service est d'exécuter une procédure, située sur un site distant, tout en conservant la forme et les effets identiques à celui d'un appel local. C'est-à-dire conserver la même sémantique et la même interface que pour un appel local. Par exemple, le passage des paramètres par référence est traité par sérialisation et la possibilité de défaillance n'existe pas pour un appel local.

Appel de méthode à distance

L'objectif de l'appel de méthode distante est d'appeler une méthode d'un objet distant, on parle alors de « Object Request Broker » (ORB). L'unité de désignation et de distribution est donc l'objet. Il s'agit d'objet langage lorsque la représentation est propre au langage, c'est une instance d'une classe. Il s'agit d'objet système lorsque la représentation est arbitraire et définie par l'environnement d'exécution. Par exemple, Java « Remote Method Invocation » (RMI) [Sun Microsystems Inc 1999b] réalise l'appel de méthode distante sur des objets langages et la norme CORBA [OMG 2001], réalise l'appel de méthode distante sur des objets systèmes.

Appel léger de procédure distante

L'*appel léger de procédure distante* « Lightweight Remote Procedure Call » (LRPC) est similaire au RPC distant, mais est optimisé pour la communication entre deux processus situés sur la même machine. Comme pour un appel distant, deux talons sont nécessaires.

Le premier mécanisme, appelé Firefly RPC, est proposé par [Schroeder et Burrows 1989], suivi de près par le LRPC de [Bershad et al. 1989]. Dans cette seconde implantation une pile partagée par les deux processus est utilisée pour le passage des paramètres. L'implantation de ce LRPC repose donc sur le concept de segment partagé, voir section 2.6.1.

Appel système

Pour finir, l'appel système est aussi un appel de méthode. En effet, un système d'exploitation est architecturé selon le modèle client-serveur. L'implantation de l'appel système repose sur le mécanisme d'appel au superviseur et sur une convention de passage de paramètre : sur la pile, dans des registres ou à une zone mémoire.

2.6.3 Modèle de communication par message

Le modèle de communication par message est un mode de communication très ancien. C'est le mode classique de programmation des réseaux, il est par exemple utilisé pour le courrier électronique. Cette communication est complètement asynchrone, c'est-à-dire qu'une émission de message est non bloquante. Une réception de message est bloquante jusqu'à la réception effective d'un message. L'interface de programmation fournie par ce modèle est du type `send` et `receive`, voir figure 2.22. Ici, le processus d'émission est appelé l'émetteur, le processus de réception est appelé le récepteur.

FIGURE 2.22 – Modèle de communication par message

Envoi de messages

Le modèle de communication par message peut être direct entre les processus. Le gestionnaire de message ne conserve pas les messages émis. C'est-à-dire que, si le processus récepteur n'est pas joignable à la réception du message, il est perdu. Ce modèle est typiquement celui fourni par les protocoles réseaux, comme par exemple IP ou UDP. Ici, un ou plusieurs fils d'exécution traitent les messages reçus. Mais si la machine est saturée (plus de fil d'exécution libre) ou en cas de panne, le message est perdu. La communication par envoi de message ne présente pas que des avantages, si le message est perdu, l'émetteur ne le sait pas nécessairement. De plus, l'ordre de réception des messages ne respecte pas nécessairement l'ordre d'émission. Bien évidemment il est possible de réaliser des échanges synchrones par un acquittement de la réception.

File de messages

La communication par message peut aussi être indirecte et passer par une file de messages. La file mémorise les messages émis et les restitue au récepteur quand il le désire ou quand il est prêt. En reprenant l'exemple précédent, l'ajout d'une file de messages à un protocole réseau permet de limiter la perte de messages, c'est par exemple ce qui est fait dans les sockets Unix qui mémorisent les messages jusqu'à ce que le récepteur les récupère.

Communication inter-processus

La *communication inter-processus* « Inter-Process Communication » (IPC) est un modèle de communication par message utilisé entre deux processus d'une même machine. Bien que l'IPC existe sur les systèmes monolithiques de type Unix avec par exemple les IPC système V, son rôle devient essentiel dans les micronoyaux. C'est un mécanisme central pour ce type d'architecture.

2.6.4 Modèle de communication par événements

La communication événementielle est basée sur les concepts d'événements et de réactions. La communication est ici anonyme et est basée sur la diffusion, l'émetteur d'un événement ne sait pas quels seront le ou les récepteurs. L'interface de programmation fournie par ce modèle est du type `publish` émettant un événement et `subscribe` attachant dynamiquement une réaction à un nom

d'événement, voir figure 2.23. Lors de l'émission d'un événement, la réaction est implicitement appelée, on parle d'ailleurs d'invocation implicite [Garlan et Shaw 1994]. Ce modèle de communication est par exemple proposé par les langages de programmation réactifs synchrones, comme Esterel ou Lustre.

FIGURE 2.23 – Modèle de communication par événement

Intergiciel orienté messages

L'appellation *intergiciel orienté messages* « Message Oriented Middleware » (MOM) regroupe les intergiciels offrant, soit le modèle de communication par messages, soit le modèle de communication par événements, soit les deux. Ce type d'intergiciel est basé sur des échanges de message en temps différé. Un programme peut envoyer un message à un autre sans se soucier de sa présence effective. Le destinataire lira son message lorsqu'il sera disponible. Les avantages de ce type d'intergiciel sont la simplicité, la robustesse et l'extensibilité. Cela présente un intérêt certain dans le contexte des plates-formes mobiles qui fonctionne généralement en mode déconnecté. Une interface de programmation des MOM pour les environnements Java est « Java Message Service » (JMS) [Sun Microsystems Inc 1998a].

Signal

Le signal, appel ascendant que nous avons déjà abordé à la section 2.4.1, fonctionne comme un événement. Il permet de déclencher depuis le système d'exploitation un événement dans une application. Ce déclenchement est potentiellement consécutif à une interruption matérielle ou à une faute de l'application. Les systèmes d'exploitation de type Unix implantent un signal en modifiant directement le contexte d'exécution du fil d'exécution actif d'un processus pour le faire exécuter une réaction. Les systèmes d'exploitation de type micronoyaux, comme L4 [Liedtke 1995], utilisent un fil d'exécution spécifique qui scrute l'occurrence de l'événement et exécute la réaction adéquate.

2.7 Noyaux de systèmes d'exploitation

Dans les sections précédentes, nous avons vu les concepts des différentes ressources qu'un système d'exploitation propose sous forme de service. Nous avons vu aussi comment un processus peut les utiliser. Ensuite, nous nous sommes intéressés à la composition et à la décomposition des systèmes. Pour finir, nous venons de voir les modèles de communication utilisés dans les systèmes. Nous

allons maintenant nous intéresser à la structure des systèmes d'exploitation et à l'organisation des différentes composantes du système.

Comme nous l'avons vu, l'architecture des systèmes d'exploitation est généralement découpée en deux parties : le noyau s'exécutant en mode superviseur et les processus s'exécutant en mode utilisateur les applications. Les processus utilisent les abstractions fournies par le noyau en effectuant des appels systèmes. Ces derniers traversent la frontière séparant le domaine de protection d'une application du domaine de protection du noyau. L'intérêt principal de cette séparation est la sécurité : un programme s'exécutant en mode utilisateur est isolé et ne peut perturber ni l'exécution des autres programmes ni l'exécution du noyau. L'inconvénient majeur de ce découpage est le surcoût engendré par les commutations entre domaines de protection.

Les noyaux des systèmes d'exploitation peuvent être classés en fonction de l'architecture mise en oeuvre pour l'exécution des services du système : système à domaine de protection unique, noyaux monolithiques, micronoyaux, noyaux extensibles, exonoyaux. Nous détaillons ces architectures et les systèmes basés sur celles-ci dans les sections suivantes en nous aidant de la figure 2.24.

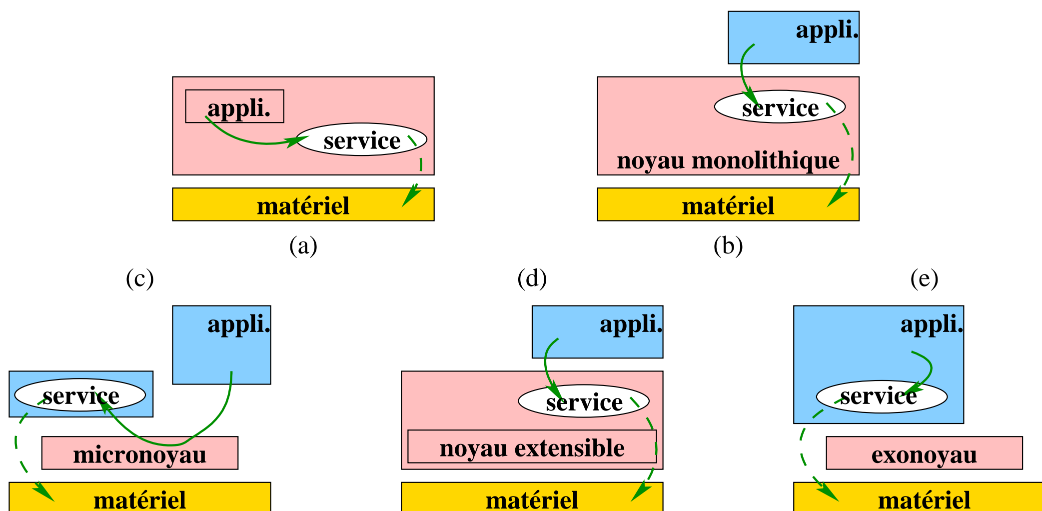


FIGURE 2.24 – Architecture des différents noyaux

2.7.1 Systèmes à domaine de protection unique

Un *système à domaine de protection unique*, voir la figure 2.24(a), exécute les services du système d'exploitation et la ou les applications dans le même domaine de protection. C'est par exemple le cas pour le système d'exploitation développé par IBM et Microsoft : DOS [Tanenbaum 1994]. Le système fournit juste une interface permettant aux applications d'accéder aux ressources matérielles. De nombreux systèmes embarqués, où l'application est incluse dans le système, sont construits selon cette architecture. C'est par exemple le cas pour le système PalmOS [PalmOS] initialement développé par 3Com.

L'intérêt de cette architecture à domaine de protection unique est l'efficacité du système. Mais aucune garantie de sécurité ne peut être fournie par un tel système : une application peut directement corrompre le système et les autres applications en accédant directement au matériel et en particulier

à la totalité de la mémoire.

2.7.2 Noyaux monolithiques

Dans un *noyau monolithique*, voir la figure 2.24(b), tous les services du système s'exécutent dans le même domaine de protection. Le système est réalisé d'un seul bloc sans structure apparente, seule son interface externe est identifiée. Il peut donc être vu comme une boîte noire. Une telle architecture n'est donc pas modifiable et encore moins extensible. L'avantage de cette organisation est la rapidité d'exécution.

Un noyau monolithique est présent dans les systèmes de type Unix : AIX développé par IBM, Linux développé initialement par Linus Torvalds, Solaris développé par Sun Microsystems, etc. Ce type de noyau offre tous les services dont un système d'exploitation a besoin : processus, gestion de la mémoire, multiprogrammation, communication, pilotes de périphériques, systèmes de gestion de fichier, etc. Du fait même de leur conception, les noyaux monolithiques offrent probablement les meilleures performances tout en garantissant la sécurité vis-à-vis de la défaillance des applications.

Le premier noyau monolithique, construit pour Unix [Ritchie et Thompson 1974], était construit de façon peu structurée. Sa maintenance et son évolution étaient extrêmement difficiles et posaient de gros problèmes de fiabilité. Même s'ils sont aujourd'hui architecturés de façon modulaire, ce problème subsiste. La notion de noyau monolithique conduit à des noyaux qui sont gros et complexes et qui posent des problèmes d'extensibilité. Par exemple, les politiques implantées dans le noyau ne correspondent pas toujours aux besoins des applications et leurs modifications sont impossibles.

Systèmes à anneaux

Nous ne pouvons pas analyser les noyaux de système sans parler du système à temps partagé MULTICS « MULTiplexed Information and Computing Service » développé au MIT [Organick 1972]. Ce système était monolithique, il n'était pas organisé sous la forme d'un noyau unique, mais sous la forme d'une série d'anneaux concentriques « ring » [Schroeder et Saltzer 1972], voir figure 2.25. Ce système pouvait comporter jusqu'à 64 anneaux. L'invariant était qu'un anneau extérieur disposait de moins de droit qu'un anneau intérieur, cela faisait office de modèle de protection.

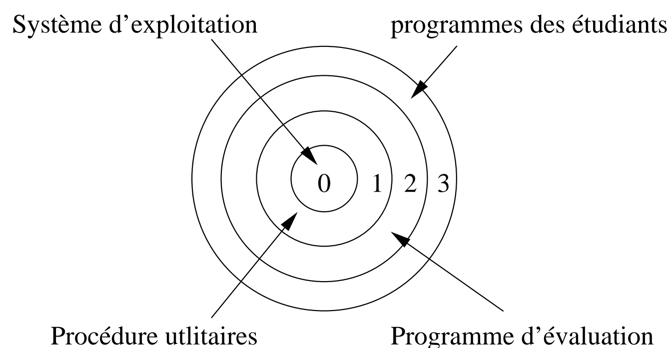


FIGURE 2.25 – Processus MULTICS avec quatre anneaux

2.7.3 Micronoyaux

L'approche *micronoyau* consiste à exécuter les services du système d'exploitation dans des processus séparés selon le modèle client-serveur, voir la figure 2.24(c). Pour obtenir un service, les applications communiquent avec des serveurs applicatifs qui se trouve à l'extérieur du noyau. Ces services peuvent être ainsi démarrés et arrêtés dynamiquement de la même façon que les applications. Les noyaux résultant de cette architecture sont théoriquement beaucoup plus petits et donc plus faciles à maintenir, mais aussi beaucoup plus sûrs. Il existe de nombreux systèmes d'exploitation basés sur un micronoyau.

La grande difficulté de cette architecture est de réaliser des communications IPC ou LRPC aussi efficaces que possible, voir les sections 2.6.3 et 2.6.2. En effet, chaque demande de service passe par le noyau qui appelle ensuite le service, et réciproquement pour le retour. Cela ajoute donc deux changements d'espaces d'adressage au coût de chaque communication avec un service. Ce problème mis à part, les micronoyaux de première génération, comme Amoeba, Mach et Chorus, étaient basés sur des IPC trop lents. Cette lenteur était due à de trop coûteuses opérations de recopie des messages, du processus émetteur vers l'espace du noyau puis de l'espace noyau vers le processus destinataire. Maintenant, des techniques existent pour améliorer l'échange de messages, et offrir un gain de performance considérable, même si le coût des changements d'espaces d'adressage subsiste. Nous parlons alors de micronoyaux de secondes générations, comme systèmes L4, QNX, 2K, etc, que nous allons voir ci-dessous.

$1^{ère}$ **génération**

Les micronoyaux de première génération datent des années 1980. Ils furent construits pour résoudre les problèmes de complexités des noyaux monolithiques, on peut citer Amoeba [Tanenbaum 1994], Mach [Accetta et al. 1986] et Chorus [Rozier et al. 1988]. Néanmoins, ces premiers micronoyaux étaient très inefficaces, avec l'IPC basé sur des échanges peu optimisés de messages.

Par exemple, le micronoyau Mach, développé initialement à l'Université Carnegie Mellon, a été construit comme une base au-dessus duquel peuvent être émulés simultanément d'autres systèmes d'exploitations de type Unix ou autre. Cette émulation est réalisée par un serveur dédié. Mach, comme tous les autres micronoyaux, fournit la gestion des processus, la gestion de la mémoire, les communications et les services d'entrée-sortie. Les fichiers et autres fonctions des systèmes d'exploitation traditionnels sont gérés dans des serveurs applicatifs.

Le micronoyau de Mach fournit cinq abstractions : les processus, les fils d'exécution, les objets mémoire, les ports d'entrée-sortie et les messages. Dans ses premières versions, le micronoyau Mach utilisait une implantation qui impliquait une double recopie pour l'échange de messages. Les versions plus récentes de Mach utilisent des techniques de gestion de mémoire virtuelle pour améliorer les performances des IPC : l'espace d'adressage avec le message de l'émetteur est traduit dans l'espace d'adressage du destinataire, évitant ainsi de recopier le message. Mach pourrait désormais être classé dans les micronoyaux de deuxième génération.

Mach a influencé le développement d'un certain nombre de systèmes d'exploitation commerciaux. L'ancien Mach 2.5 (avec une architecture monolithique) a été adopté par le consortium OSF « Open Software Foundation » en 1989 comme la base de leur système d'exploitation à la Unix ; OSF/1. Mach 2.5 était également la base du système d'exploitation sur les stations de travail NeXT ; NextStep. Le micronoyau Mach 3 et ses successeurs sont utilisés pour construire de nombreux sys-

tèmes d'exploitation. C'est le cas pour le nouveau système MacOS X d'Apple, pour le système GNU Hurd et même de certaines versions de Linux ; mkLinux.

$2^{ème}$ génération

Les micronoyaux de deuxième génération ont gardé les concepts des micronoyaux d'origine. En revanche, de nouvelles techniques de communication permettent d'obtenir de bonnes performances en intégrant cette notion dès le début de la conception du noyau. Cela offre un gain de performance considérable aux micronoyaux de seconde génération.

Ces techniques, proposées par [Liedtke 1993] pour le système L3 puis pour le système L4 [Liedtke et al.], permettent d'améliorer grandement les échanges de messages entre fils d'exécution s'exécutant dans des domaines de protection différents. La méthode proposée réside dans plusieurs règles à suivre dans la conception d'un système à base de micronoyau, pour tenir compte des exigences des IPC. Ces règles sont ;

- utilisation des registres pour les messages courts,
- utilisation de segments de mémoires partagés pour les messages longs,
- ordonnancement paresseux « Lazy Scheduling » limitant les manipulations des fils d'attente,
- possibilité de donner son quantum de temps courant au fil d'exécution réalisant le service,
- petit espace d'adressage d'un segment unique limitant le coût des changements de contexte sur machine segmentée.

L'objectif du projet L4 [Liedtke 1995, Härtig et al. 1997], développé par GMD et IBM et qui fait suite à L3, est de déterminer les concepts minimaux qu'un micronoyau doit implanter. Le critère de choix est la fonctionnalité et non la performance. Plus précisément, un concept est toléré dans le noyau uniquement si son implantation hors du noyau ne permet pas d'atteindre la fonctionnalité requise. L'interface du micronoyau doit fournir une bonne abstraction des ressources physiques à différents niveaux. Avec seulement quelques appels systèmes le noyau fournit un maximum de stabilité et d'intégrité aux applications. Le micronoyau du système L4 propose uniquement trois concepts ; gestion des fils d'exécution, gestion de la mémoire et communication. Ces trois concepts sont les concepts minimums qu'un micronoyau doit implanter. Pour obtenir de bonnes performances, le système est entièrement écrit en assembleur et un soin particulier a été porté aux IPC, comme nous venons de le voir cette étude fait aujourd'hui référence.

QNX [Hildebrand 1992], développé par la société QNX Software Systems, est un système d'exploitation dédié aux applications temps réel et aux applications embarquées. C'est un micronoyau multitâche préemptif extensible de très petite taille (12Ko). Le système peut être doté d'un serveur permettant d'offrir une interface de type Unix. Cette interface est certifiée POSIX, facilitant grandement le portage des applications existantes. QNX a trouvé une utilisation dans les domaines suivants ; automatisation industrielle, instrumentation, finance et points de vente, télécommunications, informatiques de poche, électroniques grand public, etc. Le micronoyau est fréquemment cité comme une référence en matière de fiabilité et de performance de système.

Le système d'exploitation Nemesis [Reed et Fairbairns 1997] a été développé à l'Université de Cambridge dans le projet Pegasus. L'objectif du système est de multiplexer les ressources partagées entre les applications et non pas de fournir des abstractions de plus haut niveau en donnant l'illusion d'une machine virtuelle. En effet, cette illusion n'est pas compatible avec des applications temps

réels. Ainsi, une application peut faire ce qu'elle veut de la ressource qui lui a été allouée. Les politiques d'allocation résidantes dans le noyau sont réduites au minimum et le code de contrôle de la ressource s'exécute dans les applications. Nemesis utilise un espace d'adressage de 64bits unique pour tout le système, ainsi chaque domaine de protection possède une adresse virtuelle propre, les adresses virtuelles et les adresses physiques sont égales. La protection est assurée en couplant uniquement les segments de mémoire d'une application dans son domaine et non les segments des autres applications. Cette construction simplifie l'utilisation de données partagées et diminue le coût des changements de contexte. Les communications sont assurées via un ORB utilisant un langage de description d'interface nommé MIDDL.

Choices [Campbell et al. 1993] est un système orienté objet pour la programmation parallèle, conduit à l'université d'Illinois à Urbana-Champaign. Il a ensuite évolué en μChoices [Campbell et Tan 1995], structuré suivant un nanonoyau [Tan et al. 1995], englobant la partie dépendante de la machine du noyau et réifiant les ressources matérielles, et un ensemble d'abstractions de plus haut niveau complétant le micronoyau. En fait, un nanonoyau n'est pas un noyau, c'est uniquement une librairie permettant de réifier le fonctionnement du processeur. L'objectif de ce découpage est de rendre le micronoyau portable et plus flexible.

2.7.4 Noyaux extensibles

Un *noyau extensible* peut être étendu par le *chargement dynamique* de code dans le noyau. Les systèmes Spin [Bershad et al. 1995] et Vino [Small et Seltzer 1994] sont des systèmes extensibles, mais aussi tous les noyaux monolithiques modernes, comme Linux, AIX ou Solaris. Cette extensibilité diminue l'empreinte mémoire initiale du noyau, en intégrant le strict minimum de services. Elle offre une meilleure utilisation de la mémoire, par l'opération duale consistant à supprimer dynamiquement les services inutilisés. Seuls les services requis durant l'exécution du système sont chargés dynamiquement. Par exemple, les pilotes de périphériques sont chargés uniquement si le périphérique associé est présent sur la machine.

Un noyau monolithique peut être extensible. Mais cette extensibilité est souvent limitée à certains services pré-définis à la conception du noyau, généralement les pilotes de périphériques et les systèmes de gestion de fichiers. Par exemple, il n'est pas possible de changer la gestion du processeur. Ce type de flexibilité convient aux applications standards. Néanmoins, un noyau monolithique reste *fermé* car il ne peut être spécialisé en fonction d'applications arbitraires.

Le gros problème de l'extension dynamique d'un noyau est la sécurité. En effet, comme le service est ajouté directement dans l'espace d'adressage du noyau, ses éventuelles défaillances peuvent engendrer des dysfonctionnements dans tout le système. Néanmoins, il existe des mécanismes matériels ou logiciels permettant l'extensibilité sûre des noyaux.

- *protection matérielle*. L'extensibilité des noyaux par l'utilisation des techniques de protection matérielle (MMU) isole ainsi les services du reste du système. Cela revient bien évidemment à mettre en œuvre un micronoyau.

- *langage sûr*. Développer les services dans un langage offrant des propriétés de sûreté permet de sécuriser l'extensibilité. Ces propriétés sont essentiellement basées sur un typage fort et une gestion automatique de la mémoire par la suppression de la notion de pointeur.

  Cette solution a été proposée par l'université de Washington avec le système Spin [Bershad et al. 1995]. Ici, le noyau et les services systèmes sont écrits en Modula-3 qui fournit la protection

requise. Les applications peuvent être écrites dans n'importe quel langage et s'exécutent dans leurs propres espaces d'adressage. Les services ajoutés s'exécutent dans leur propre domaine de protection qui défini les nomes accessibles durant l'exécution. Les domaines sont implantés au niveau langage en Modula-3, et non grâce à des espaces d'adressage virtuels. Les domaines sont utilisés pour contrôler la liaison dynamique des extensions dans le noyau qui aura uniquement accès aux noms définis. D'un point de vu conceptuel, Spin peut être vu comme un micronoyau.

- *sandbox*. L'interposition peut être utilisée pour assurer la protection du système par des techniques logicielles. Nous avons vu, qu'un programme n'est qu'une suite d'appels à des ressources. Sans protection, le programme peut ainsi accéder à toute la mémoire [Seltzer et al. 1996]. Cette protection logicielle, appelée « sandbox » [Wahbe et al. 1993], consiste à interposer du code entre le programme et les ressources. Cela permet de protéger mutuellement plusieurs programmes partageant un même domaine de protection. À chaque programme est associée une zone de confinement. Dans la mesure du possible, tout accès mémoire est vérifié statiquement avant exécution. Sinon ils sont vérifiés dynamiquement par l'insertion d'un code vérifiant avant chaque accès la validité du pointeur. Cette solution a été utilisée avec le système Vino [Small et Seltzer 1994] par l'université de Harvard.

2.7.5 Exonoyaux

Un *exonoyau* implante tous les services systèmes sous forme de librairie applicative. Les services s'exécutent alors dans les domaines de protection des applications. Cette philosophie consiste à limiter le système d'exploitation au multiplexage et à la protection des ressources et à éliminer toutes les abstractions du noyau [Engler et Kaashoek 1995]. Un tel noyau réifie directement aux applications les événements logiciels et matériels. Tout ce qui relève du contrôle des ressources est exécuté dans les processus applicatifs et ainsi mis sous le contrôle potentiel des applications n'ayant pas de droits particuliers.

L'exonoyau permet de résoudre le problème posé par la multiplication des abstractions de ressources ayant pour conséquence d'augmenter les temps d'accès aux ressources matérielles et de faire perdre aux applications le contrôle des ressources. Or cette perte de contrôle est d'autant plus pénalisante si l'application dispose d'information, lui permettant d'exploiter mieux les ressources. Par exemple, la mise en œuvre de la qualité de service n'est pas possible si elle n'est pas prise en compte dans le système.

Ce style de système se caractérise par un noyau vu comme une représentation (très proche) du matériel et des librairies applicatives, appelées des « libos », définissant les abstractions et services de plus haut niveau du système. D'un point de vu conceptuel, comme le souligne [Härtig et al. 1997], un noyau d'exonoyau est identique à un noyau d'un micronoyau minimum, dans le sens ou il propose les mêmes concepts ; fils d"exécution, domaine de protection et communication. Le choix de librairies pour implanter les extensions du système est justifié par la souplesse que cela procure aux développeurs d'applications, ainsi que par le fait qu'ils ont l'habitude d'en manipuler. Ces librairies sont le point d'accès à la flexibilité : tout utilisateur peut définir sa propre librairie et ainsi étendre le système. Cette dernière sera liée à l'application à la génération du code exécutable, ce qui permet de transporter les extensions dans le binaire.

Le premier système construit selon cette architecture a été Aegis [Engler et al. 1995, Kaashoek et al. 1997] développé au MIT à Boston. Parce qu'il était le premier et le seul exemple de cette architecture il est communément appelé Exokernel. Les expérimentations montrent que les gains apportés

par cette architecture peuvent être importants. Mais la conception d'un tel noyau est relativement complexe. Par exemple, le multiplexage du disque requiert un langage dédié décrivant les meta-données des systèmes de gestions des fichiers et dont les scripts sont mémorisés sur disques de façon à persister à l'application et interprétés par le noyau. Au dire des concepteurs, l'implantation de la gestion seule du disque prit trois ans.

2.8 Infrastructures logicielles réparties

Le rôle d'une *infrastructure logicielle répartie* est de permettre l'exécution des applications s'exécutant simultanément sur plusieurs machines. L'infrastructure fournit des fonctions de communication et de répartition qui facilite la mise en œuvre des applications réparties ; la communication, la défaillance des nombreuses entités mises en jeu, la sécurité des accès, etc. Deux approches permettent de construire une infrastructure logicielle répartie ; l'intergiciel et le système d'exploitation répartie.

2.8.1 Intergiciels

Un *intergiciel*, « middleware » en anglais, assure un lien entre des composants applicatifs distants et fait l'hypothèse d'un système d'exploitation sous-jacent sur chaque machine, les systèmes sont potentiellement fortement hétérogènes. Un intergiciel offre des fonctions de communication aux applications et se présente comme une bibliothèque logicielle éventuellement associée à un environnement applicatif, par exemple sous la forme d'un démon. Le rôle d'un intergiciel n'est pas de donner l'illusion d'une machine unique, il n'y a pas de contrôle unique des ressources.

Un intergiciel fournit un ensemble de services comme la communication, la désignation, les transactions, la persistance, etc., pour la mise en œuvre d'applications réparties. Il offre aux applications une transparence à l'hétérogénéité des systèmes d'exploitation, des protocoles de communication, des couches réseau traversées et bien évidemment du matériel sous-jacent. Pour des raisons d'interopérabilité les intergiciels sont généralement normalisés. La norme la plus connue est CORBA [OMG 2001].

2.8.2 Systèmes d'exploitation répartis

Un *système d'exploitation réparti* donne l'illusion d'une machine unique aux applications. Par exemple, le placement des applications s'effectue automatiquement sans intervention de l'utilisateur. Un système d'exploitation réparti mobilise et gère les ressources distantes. De part leur architecture basée sur le modèle client-serveur, les systèmes d'exploitation répartis sont généralement basés sur un micronoyau. Les mécanismes de communication sont étendus pour communiquer de façon transparente avec des machines géographiquement distantes, c'est le RPC qui est utilisé dans ce cas.

Spring [Hamilton et Kougiouris 1993] est un système d'exploitation réparti développé par les laboratoires de recherche de Sun Microsystems et destiné à des applications orientées objets. Il est basé sur un micronoyau qui fournit un mécanisme d'IPC orienté objets, appelé « doors », qui sont des points terminaux de communication, comme des sockets en Unix BSD ou des ports en Mach. D'autres services comme le nommage ou la pagination sont fournis en mode utilisateur hors le noyau. Les caractéristiques de Spring sont l'utilisation d'un langage IDL pour la définition de toutes les interfaces clés du système et la notion de « subcontract », voir la section 2.5. Le but est de bien

séparer les interfaces des implantations et de permettre la coexistence de différentes implantations de la même interface.

Influencé par Spring, 2K est un projet mené par l'Université d'Illinois à Urbana-Champaign, qui repose sur une approche à composants et intergiciel [Kon et al. 1999, Román et al. 1999]. Il est constitué d'un micronoyau Off++ au-dessus duquel se trouve un ORB réflexif, dynamicTAO, par lequel les applications peuvent charger des composants pour le système. Off++ est un micronoyau orienté objet, réparti et adaptable, chargé d'exporter des ressources (matérielles) distribuées au système 2K. Il est possible de charger du code utilisateur dans le noyau pour améliorer les performances ou mettre à jour du code.

2.9 Conclusion

L'ensemble des choix d'implantation que l'on peut rencontrer lorsque l'on veut concevoir un système d'exploitation ou une infrastructure logicielle répartie forme un espace de conception illustré à la figure 2.26. Il faut décider si le système et les applications partagent le même domaine de protection comme dans un système à domaine de protection unique, sinon cela signifie l'utilisation d'un noyau. Il faut ensuite décider où s'exécutent les services du système : dans le noyau avec un noyau monolithique, dans des processus séparés avec un micronoyau, sous forme de librairie applicative avec un exonoyau. Il faut aussi décider si le noyau est extensible et éventuellement de façon sûre. Dans le cas d'une infrastructure logicielle répartie, il faut décider si le système fournit lui-même les services de communication ou s'ils sont fournis par une couche intergiciel séparée.

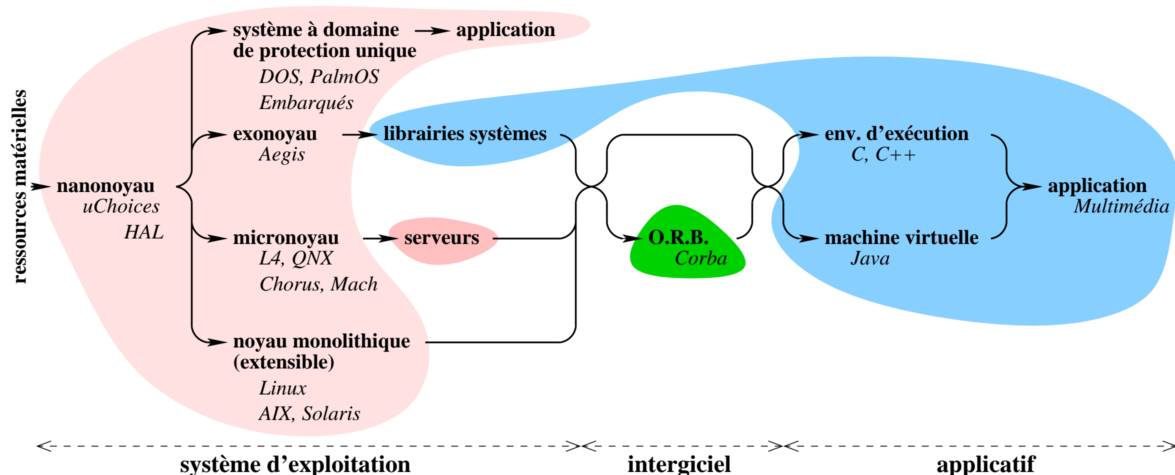


FIGURE 2.26 – Espace de conception

Comme nous l'avons vu dans ce chapitre, toutes les solutions ont des avantages et des inconvénients, qui en fonction de l'utilisation, peuvent être plus ou moins marqués. Le tableau 2.1 liste de façon synthétise les avantages et les inconvénients de chaque infrastructure système.

L'objectif principal de l'architecture de système nommée THINK, que nous allons voir dans le chapitre suivant, est de ne pas se limiter à un point de cet espace de conception. En d'autres termes, cette architecture doit être suffisamment flexible pour permettre de nous déplacer dans tout l'espace et

	système à domaine unique	**noyau monolithique**	**micronoyau**	**noyau extensible**	**exonoyau**
protection[a]	$-$	$+$	$+$	$+$[b] ou $--$[c]	.
efficacité	$++$	$+$	$-$[d]	$+$	.
flexibilité	$+$[e]	$-$	$+$	$+$	$+$
reconfiguration dynamique	$+$	$-$	.	.	$+$
simplicité de dév.	.	$-$	$+$	$+$	.

[a] des services et du système
[b] pour les noyaux utilisant des techniques de « sandbox » ou de langage sûr
[c] si aucune technique de protection n'est utilisée
[d] dû au coût des IPC
[e] sous réserve que l'infrastructure le permette

TABLEAU 2.1 – Analyse synthétique des infrastructures systèmes

de créer des instances de tous ces systèmes. Une instanciation de THINK se place clairement dans une philosophie de noyau extensible, mais il est possible de faire varier cette instanciation d'un système à domaine unique à un noyau monolithique en passant par un exonoyau. Dans une instanciation particulière, on retrouve les mêmes types de compromis et de caractéristique générales.

Chapitre 3

Architecture logicielle THINK

Ce chapitre présente l'architecture logicielle THINK [1]. Cette architecture comprend des concepts, des règles et des invariants de structure que l'on va aussi utiliser de manière systématique. Cette architecture clarifie l'organisation des systèmes d'exploitation et des infrastructures logicielles réparties. La formalisation de ces concepts permettra à terme de raisonner sur les systèmes. L'objectif de l'architecture est de fournir un cadre logiciel flexible permettant de construire des systèmes à la carte en intégrant uniquement les services requis. Ces services peuvent être soit standards, c'est-à-dire qu'ils sont pris sur étagère, ou spécifiques, c'est-à-dire qu'ils sont développés spécifiquement pour un système.

3.1 Objectifs de l'architecture

L'architecture logicielle THINK permet de modéliser de façon uniforme les systèmes d'exploitation et les infrastructures logicielles réparties. L'architecture doit être suffisamment flexible et offrir des moyens pour couvrir tout l'espace de conception, vu à la figure 2.26, et laisser aux concepteurs de systèmes le choix des compromis satisfaisant aux contraintes des applications cibles, ils pourront pour cela s'aider du tableau 2.1. Un système THINK, c'est-à-dire un système conforme à l'architecture logicielle THINK, possède les caractéristiques suivantes :

- *ouvert*. Dans un système ouvert, toutes les interfaces sont explicites. Cela autorise l'inter fonctionnement et la portabilité des systèmes s'y conformant. Un tel système peut donc être déployé sur des architectures matérielles très hétérogènes.
- *intégré*. Un système est intégré s'il peut être construit, de façon simple et sans développement ad hoc coûteux, à partir de différents sous-systèmes et différentes ressources. Cela peut comprendre des systèmes avec différentes architectures, différentes ressources et différentes performances.
- *flexible*. Un système est flexible s'il peut évoluer et s'accommoder, de façon dynamique ou de façon statique, aux changements pouvant survenir sur l'environnement du système.
- *modulaire*. La modularité est la propriété de bâtir un système par morceau. L'objectif est qu'en cassant un système complexe par morceau, la complexité devient plus facile à gérer. La mo-

[1] THink Is Not a Kernel.

dularité est aussi la base de la flexibilité. Les dimensions d'un système modulaire sont non figées.

- *administrable*. Cette caractéristique permet à une ressource, logicielle ou matérielle, du système d'être observée, contrôlée et gérée en vue de supporter la configuration, la qualité de service et les différentes politiques.
- *programmation uniforme*. Le modèle de programmation uniforme de l'architecture est exploitable en centralisé ou en réparti. Ce modèle de programmation uniforme offre la transparence vis-à-vis des problèmes causés par la répartition et facilite la construction d'applications réparties.

L'architecture logicielle THINK offre le support minimum permettant de construire des systèmes ayant ces caractéristiques. Mais ces caractéristiques ne peuvent être obtenues si le concepteur de tout ou partie d'un système respecte les règles proposées par l'architecture. Dans le cas contraire, aucune garantie ne peut être avancée.

3.1.1 Ce qui n'est pas abordé

Un certain nombre de caractéristiques que l'on est en droit d'attendre d'une telle architecture ne sont actuellement pas traitées par l'architecture logicielle THINK. C'est notamment le cas pour les caractéristiques suivantes.

- *sécurité*. Un système sécurisé permet la protection des données et du système lui-même contre les éventuels accès non autorisés.
- *sûreté*. Un système est sûr de fonctionnement s'il est tolérant aux pannes qui peuvent survenir dans l'environnement.

3.2 Concepts mis en œuvre

L'architecture logicielle THINK [Fassino et Stefani 2001] s'organise autour d'un petit nombre de concepts. Ces concepts sont appliqués de manière systématique à tous les niveaux d'un système, que ce soit pour l'organisation d'un noyau THINK, d'une machine unique ou d'un système réparti dans son ensemble.

Un système THINK est formé d'un ensemble de domaines s'exécutant en parallèle. Chaque domaine consiste en un ensemble de composants. Un composant exporte des services sous forme d'interface et interagit avec d'autres composants au travers de liaisons connectant leurs interfaces. Les domaines et les liaisons sont réifiés sous forme de composant et peuvent être fabriqués par composition de composants. Nous allons détailler dans cette section les trois concepts fondamentaux de l'architecture THINK qui sont : les composants, les liaisons flexibles et les domaines. Puis, nous verrons un modèle de gestion des ressources mis en œuvre dans un système THINK.

3.2.1 Concepts de base

L'architecture THINK repose sur le concept de composant permettant de modéliser tous les éléments manipulés et sur le concept d'interface permettant l'interaction entre composants, que ce soit

au niveau logiciel ou matériel. Ces concepts sont directement inspirés des concepts d'objet et d'interface proposés par le modèle de référence pour le traitement réparti ouvert (RM-ODP)[2] [ISO/IEC 1998, ISO/IEC 1995a, ISO/IEC 1995b, Blair et Stefani 1997].

Composants

Le concept de *composant* permet de modéliser les éléments manipulés dans l'architecture logicielle THINK. Un composant est un concept de l'environnement exécution bien connu, voir [Meyer 1997]. Il se caractérise par l'encapsulation et l'abstraction, il contient un état et il fournit des services clairement identifiés. Un système est composé d'un ensemble de composants, la figure 3.1 donne une représentation d'un système avec des composants de tailles arbitraires. Cette notion de composant, grâce à la notion d'interface, sépare l'implantation de son utilisation et permet de remplacer un composant par un autre, c'est aussi l'unité de réutilisation. Un composant est capable d'évoluer par des interactions avec l'environnement extérieur grâce à des points d'accès qui sont ses interfaces.

Nous ne faisons pas dans la notion de composant de supposition de taille. Les composants peuvent être de granularité arbitraire. Par exemple, un composant peut être aussi grand qu'un réseau téléphonique ou aussi petit qu'un entier. Un composant peut avoir des comportements arbitraires et par exemple avoir un niveau de parallélisme interne arbitraire. De la même manière les interactions peuvent prendre des sémantiques très variées. Par exemple, les interactions peuvent être synchrones ou asynchrones. Dans notre architecture, un composant peut donc être un élément primaire tel qu'une ressource matérielle ou un élément complexe tel qu'un pilote de périphérique ou un gestionnaire réseau. Le concept de composant ne se limite pas aux parties logicielles d'un système. Il permet aussi de modéliser les composants matériels, comme le processeur, les contrôleurs, les périphériques, etc.

Nous ne faisons aucune hypothèse sur le langage de programmation utilisé pour implanter ces composants. Il peut s'agir de l'assembleur, du C et même de langage de plus haut niveau, comme Java, sous réserve de posséder le compilateur ou l'interpréteur adéquat. Le concept de composant que nous adoptons n'est pas dépendant de l'utilisation de la notion d'objet offert par les langages de programmation orientés objets « Object Oriented Programming » (OOP), tels que les langages de type C++ ou Java. En particulier, le composant ne repose pas sur les notions d'héritages. Il est possible néanmoins d'utiliser les langages de programmation objets pour implanter les composants, mais ce n'est pas nécessaire.

Interfaces

Une *interface* est un point d'accès à un composant. Un composant interagit uniquement avec son environnement au travers de ses interfaces, il peut offrir une ou plusieurs interfaces et il peut aussi interagir avec lui-même. La figure 3.1 donne une représentation d'un système avec des composants dotés d'interfaces. L'intérêt d'offrir plusieurs interfaces réside dans la séparation des fonctions et dans la distribution. Cela peut, par exemple, être utilisé pour séparer l'interface fonctionnelle de l'interface d'administration. Comme dans le modèle de référence ODP, les interfaces sont typées. Aucun choix particulier de système de types n'est fait. Nous supposons uniquement que chaque interface est équipée d'un type et que le système de types s'organise en un treillis (ordonné par une relation de sous-typage) doté d'un plus grand élément, `Top`.

[2] Reference-Model for Open Distributed Processing

Le concept d'interface permet la réutilisation et la flexibilité des composants. En effet, une interface décrit de façon exhaustive le composant et permet de séparer la manière d'utiliser celui-ci de son implantation. Il faut aussi noter que même les composants matériels respectent cette architecture et offrent une ou plusieurs interfaces. Par exemple, le processeur offre l'ensemble de ses instructions comme interface.

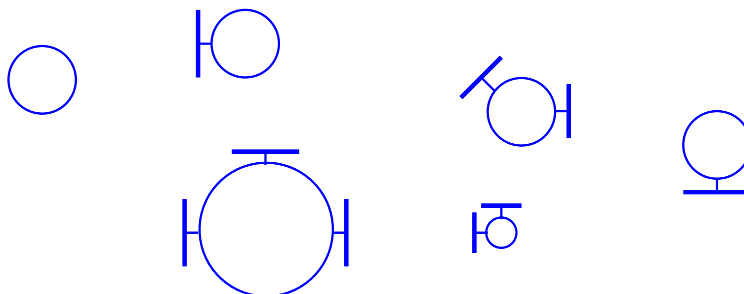


FIGURE 3.1 – Système à base de composants et d'interfaces

Dépendances

Pour pouvoir s'exécuter, un composant dépend d'un certain nombre de services. Ces services sont fournis par d'autres composants et sont accessibles via des interfaces clairement identifiées. Les dépendances sont alors exprimées en termes d'interfaces. Comme le concept de composant modélise les composants matériels, une dépendance peut être exprimée avec ces composants. Par exemple, le gestionnaire du processeur PowerPC dépend du PowerPC. L'intérêt de modéliser les composants matériels est ici immédiat, et permet de gérer l'hétérogénéité et le portage des composants.

Avec ce concept, nous supposons qu'un composant peut exprimer d'une part les interfaces qu'il fournit, et d'autre part les interfaces qu'il utilise. Cette description peut par exemple être donnée dans un langage spécialisé ou peut être directement contenue dans le langage de programmation du composant. L'intérêt de cette description est de pouvoir travailler sur les composants, par exemple en visualisant les dépendances entre les composants lors du déploiement d'un système ou en vérifiant les dépendances non résolues.

3.2.2 Liaison flexible

Le concept de *liaison flexible* modélise de façon uniforme l'ensemble des interactions entre les différents composants du système. Intuitivement, une liaison peut être vue comme un canal de communication entre deux ou plusieurs composants quelconques, éventuellement situés dans des espaces d'adressage séparés qui eux-mêmes peuvent être situés dans des machines géographiquement distantes. Différentes formes de communication ont été vues au chapitre 2. La figure 3.2 reprend l'exemple du système, vu à la figure 3.1 en y ajoutant des liaisons.

Une liaison flexible réifie dynamiquement les chaînes de communication et offre la possibilité de créer dynamiquement différentes formes de liaison permettant d'obtenir différentes sémantiques de

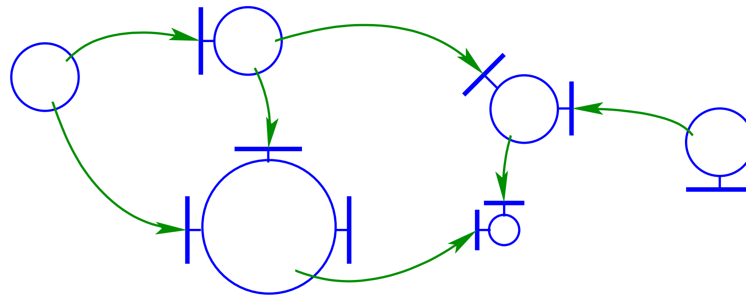


FIGURE 3.2 – Système à base de composants, d'interfaces et de liaisons

communication, par exemple : la sérialisation (dans le cas d'interactions distantes), la synchronisation (pour le multiplexage des accès), l'interposition (pour détourner des appels comme réalisés dans les mandataires) ou la persistance (pour les accès à des composants persistants).

L'introduction de la notion de liaison à deux intérêts. Premièrement, cette notion autorise la mise en œuvre de formes arbitraires de communication. Deuxièmement, elle autorise la mise en place de liaisons spécialisées entre composants, permettant d'optimiser la réalisation. Cette capacité d'optimisation doit permettre des granularités plus faibles de composant par rapport à des modèles classiques où la sémantique de communication est prédéterminée.

Le concept de liaison flexible, qui modélise les interactions entre les composants, est construit autour de plusieurs autres concepts qui sont l'interface (déjà vu), le nom, le contexte de désignation et la liaison.

Noms

Chaque interface dans l'architecture THINK est désignée par un *nom*. Comme dans le modèle de référence ODP, les noms dans THINK sont des noms contextuels, c'est-à-dire qu'ils sont tous relatifs à un contexte de désignation. Un *contexte de désignation* regroupe des conventions de désignation et des fonctions de création et d'attribution de noms.

Un contexte de désignation donné peut connaître plusieurs autres contextes de désignation, autorisant la communication de noms et la création d'alias (« aliasing ») entre ces contextes. La connaissance d'un contexte de désignation ne signifie pas forcément la connaissance de l'ensemble des noms générés par ce contexte de désignation. Nous ne postulons pas une organisation a priori des contextes de désignation. En particulier, l'existence d'un contexte racine unique n'est pas requise, comme dans les schémas de désignation hiérarchiques classiques. En général, les contextes de désignation d'un système donné s'organisent en un graphe orienté.

L'architecture logicielle THINK ne place aucune contrainte sur la forme des noms. Il doit juste être possible, à partir d'un nom, d'une part d'obtenir le contexte courant associé à un nom, d'autre part d'obtenir une forme sérialisée du nom, par exemple sous la forme d'une chaîne d'octets. Il y a tout aussi peu de contraintes portant sur les contextes de désignation. Un contexte de désignation doit juste autoriser la reconstruction d'un nom à partir d'une forme sérialisée de ce nom et autoriser la création de noms pour désigner une interface donnée. La création d'un nom dans un contexte donné

résulte en la création d'une association entre le nom et l'interface qu'il représente. Cette association représente un *lien de désignation*.

Liaisons

Le lien de désignation ne correspond pas à un chemin de communication avec l'interface ainsi désignée. Dans l'architecture THINK, comme dans le modèle de référence ODP, accéder à un composant et interagir avec lui suppose au préalable qu'une liaison ait été établie (implicitement ou explicitement) avec une ou plusieurs interfaces du composant considéré. Nous appelons *liaison* un chemin de communication entre composants. Un tel chemin de communication peut prendre des formes multiples.

- *liaison langage*. Il s'agit là de l'association par le compilateur entre un symbole langage et une structure mémoire ou par extension des registres de contrôleur de périphérique. Un symbole langage correspond au nom que nous donnons à nos variables et à nos méthodes dans un programme. L'ensemble de ces symboles forme le contexte de désignation du programme traité par le compilateur. Dans ce cas, le lien de désignation est aussi un chemin de communication. On peut utiliser directement le symbole pour interagir avec le composant qu'il référence (typiquement, en invoquant une opération de l'interface référencée).

- *liaison système*. Il s'agit en général de la combinaison de liaisons langage avec un ou plusieurs canaux de communications entre composants. Ces canaux peuvent impliquer notamment des communications IPC ou LRPC entre applications au sein d'une même machine et des connexions réseau entre machines distantes.

  Une liaison n'est pas nécessairement limitée à une sémantique de communication. Il est possible d'ajouter à une liaison des traitements différents. Ce type de liaison permet par exemple de séparer le code fonctionnel d'un composant des propriétés non fonctionnelles, comme la synchronisation ou la persistance. La partie non fonctionnelle du composant est alors implantée dans une liaison spécialisée qui réalise le traitement non fonctionnel requis. Une liaison peut alors éventuellement ne pas invoquer le composant cible, c'est par exemple le cas avec une liaison assurant le cache des accès.

Dans l'architecture logicielle THINK, une liaison peut être composite et être constituée de plusieurs composants dits *composant de liaison*. Par exemple, une liaison de type IPC peut être composée d'un composant, situé chez l'appelant, réalisant un appel système et d'un composant, situé dans le noyau, réalisant une remontée système et appelant le composant cible.

La création des liaisons est le rôle de composants usines particulières, appelés *usines à liaisons*, voir la figure 3.3. Là encore, l'architecture THINK ne place que peu de contraintes sur une usine à liaisons, il doit juste être possible de demander la création d'une liaison avec une interface donnée, désignée par un nom. C'est donc l'usine à liaisons qui est responsable de la mise en place de composants de liaison assurant la liaison.

De façon schématique, dans le cas d'une liaison point à point entre deux composants, une interaction peut se résumer à l'équation donnée à la figure 3.4. Cette équation met en évidence la liaison comme étant le "**.**" dans un appel de méthode d'une interface. La liaison flexible permet de réifier, de façon optionnelle, cette liaison sous la forme d'un composant de liaison exportant la même interface et lui ajoutant ainsi une sémantique arbitraire.

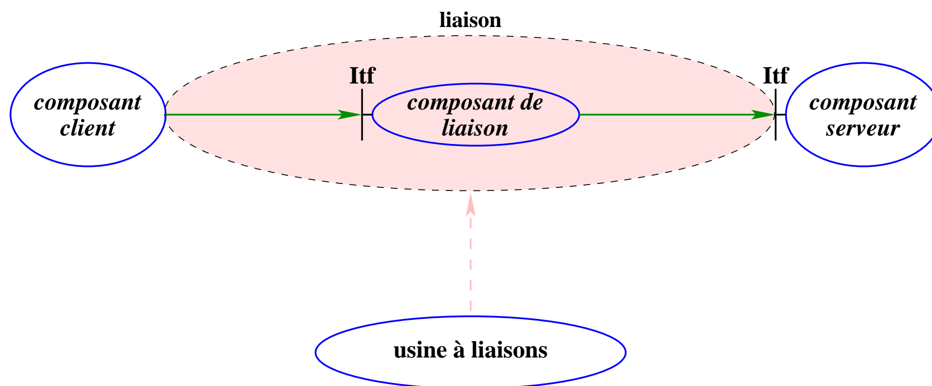


FIGURE 3.3 – Liaison et usine à liaisons

$$\mathbf{i.m(arg_1, \ldots, arg_m)} \text{ où } \left\{ \begin{array}{rcl} \mathbf{i} & = & \text{référence de l'interface appelée} \\ \mathbf{m} & = & \text{nom de la méthode invoquée} \\ \{\mathbf{arg_n}\} & = & \text{ensemble des arguments} \\ . & = & \text{liaison à modéliser} \end{array} \right.$$

FIGURE 3.4 – Écriture d'une liaison réalisant l'appel de méthode sur une interface

Bien évidemment, un tel composant peut être généré automatiquement par compilation, ce qui permet d'obtenir un certain niveau de transparence. C'est d'ailleurs là, tout l'intérêt et toute la puissance de la liaison flexible. En effet, lorsque qu'une interface est utilisée dans un type de liaison particulière, le composant de liaison est généré par compilation. Ce composant est mis en place dynamiquement par l'usine à liaisons au moment de la création de la liaison. Nous étudierons en détail, au chapitre 5, comment les composants de liaison sont utilisés pour réaliser des liaisons spécialisées, comme les IPC, LRPC et RPC.

Canevas logiciel

Le concept de liaison flexible est manifesté dans THINK par un canevas logiciel minimal, décrit ci-dessous. Ce canevas logiciel reprend l'essentiel du canevas central de la plate-forme Jonathan [Dumant et al. 1998].

Pour décrire les interfaces, il est préférable de fixer un système de types induit par un langage de description d'interfaces. Par exemple, il est possible d'utiliser comme langage : l'IDL de Corba [OMG 2001], Java [Gosling et al. 2000b] ou même de définir un nouveau langage spécifique. De fait de la simplicité du langage Java, nous utilisons son sous-langage de définition d'interfaces. Le choix de ce langage ne présage pas du ou des langages utilisés pour l'écriture des composants.

L'interface `Top`, voir figure 3.5, est le plus grand élément du treillis de types dans THINK. Il s'agit du type commun à toutes les interfaces de THINK (chaque interface dans THINK étend implicitement `Top`).

L'interface `Name`, voir figure 3.6, correspond au type commun de tous les noms dans THINK.

```
interface Top {
}
```

FIGURE 3.5 – Interface Top

L'opération getDefaultNC permet d'obtenir le contexte de désignation du nom, tandis que l'opération toByte permet d'obtenir une forme sérialisée du nom.

```
interface Name {
   NamingContext getDefaultNC();
   String        toByte();
}
```

FIGURE 3.6 – Interface Name

L'interface NamingContext, voir figure 3.7, correspond au type commun de tous les contextes de désignation dans THINK. L'opération byteToName permet d'obtenir un nom à partir de sa forme sérialisée. L'opération export permet de créer un nom pour l'interface *itf* passée en paramètre. Cette opération a pour effet de créer un lien de désignation entre le nom retourné par l'opération et l'interface passée en paramètre. Il faut noter que cette opération peut ne pas créer de nouveau nom si un lien de désignation associant l'interface passée en paramètre existe déjà dans le contexte. Pour des raisons d'optimisation, cette opération peut avoir pour effet de bord d'instancier tout ou partie d'une liaison avec l'interface passée en paramètre, par exemple la création d'une « socket » et d'un talon côté serveur dans le cas d'une liaison client-serveur classique. Le choix du contexte de désignation à appeler lors d'un export dépend complètement de la sémantique de l'espace de communication que l'on veut choisir, généralement il s'agit du contexte de désignation locale. Par exemple, si un composant serveur désire être dupliqué, il va s'adresser à une usine à liaisons qui désigne un groupe et qui aura pour conséquence de créer des copies de lui-même et le nom retourné est celui du groupe.

```
interface NamingContext {
   Name        byteToName(String strname);
   Name        export(Top itf);
}
```

FIGURE 3.7 – Interface NamingContext

L'interface BindingFactory, voir figure 3.8, correspond au type commun de toutes les usines à liaison dans THINK. L'opération bind crée une liaison vers l'interface désignée par le nom *name* passé en paramètre de l'opération. La forme d'une liaison dans THINK peut-être arbitraire, néanmoins, chaque liaison doit maintenir l'intégrité des liens de désignation entre composants. Ainsi, une liaison distante, i.e. une liaison autorisant une communication entre le contexte de désignation courant et plusieurs contextes de désignation différents, doit s'assurer que chaque interface passée en

paramètre d'une opération sur la liaison dispose d'un nom valide (i.e. a été exportée ou est connue) dans le contexte courant. Ceci correspond à une post-condition sur le traitement en sortie par la liaison (« marshalling ») des interfaces passées en paramètre. De façon duale, une liaison distante est responsable de l'introduction dans le contexte courant de nouveaux noms correspondant aux interfaces référencées dans d'autres contextes de désignation et non encore connues dans le contexte de désignation courant. Ceci correspond à une post-condition sur le traitement en entrée par la liaison (« unmarshalling ») des interfaces passées en paramètre.

```
interface BindingFactory {
   Top         bind(Name name);
}
```

FIGURE 3.8 – Interface `BindingFactory`

Exemples de création d'une liaison

Avant de continuer plus avant dans la présentation des concepts mis en œuvres, il nous semble nécessaire de montrer comment le canevas logiciel est utilisé par le programmeur d'une application ou du système lui-même. Dans l'exemple ci-dessous, nous supposons que le contexte de désignation et l'usine à liaisons sont désignés par des noms langages et qu'il est possible de les atteindre directement grâce à une liaison langage créée statiquement par le compilateur. En fonction de l'implantation, le composant exportant l'interface `NamingContext` et le composant exportant l'interface `BindingFactory` peuvent être les mêmes, c'est à dire un composant unique exportant les deux interfaces.

La figure 3.9 montre comment un composant serveur nommé *componentserver* exporte une interface `InterfaceServer` qu'il implante. Tout d'abord, il exporte l'interface auprès du contexte de désignation nommé *naming*. Cette opération lui renvoie un nom *Name*. L'opération `toByte` permet alors d'obtenir une forme sérialisée de ce nom. Celle-ci peut alors être communiquée à d'autres composants clients. Dans la suite de l'exemple, nous supposons que la forme sérialisée du nom *namestr* est `"bar"`.

```
NamingContext naming;

Name name = naming.export((InterfaceServer)componentserver);
namestr = name.toByte();
```

FIGURE 3.9 – Exportation d'une interface depuis un composant serveur

La figure 3.10 montre comment un composant client peut se lier à l'interface de type `InterfaceServer` exportée précédemment par le composant serveur. Le client doit préalablement demander l'établissement d'une liaison avec celui-ci. Pour ce faire, il doit tout d'abord récupérer le nom de cette interface, soit en obtenant directement la référence sur le nom, soit en obtenant une forme sérialisée du nom, par exemple saisie par l'utilisateur. Cette forme sérialisée, pour nous `"bar"`,

lui permet de récupérer un nom valide auprès du même contexte de désignation *naming*. Il peut alors demander, à l'usine à liaisons *factory*, la création d'une liaison permettant d'atteindre le composant serveur. La référence d'interface *itf* obtenue en résultat est du même type que l'interface `InterfaceServer` exportée par le serveur, mais est potentiellement exporté par un composant de liaison. Ensuite, il est directement possible d'invoquer des opérations sur cette interface, par exemple `foo`. Quelle que soit la configuration de la liaison, les éventuels composants de liaison se chargeront d'acheminer l'invocation.

```
NamingContext naming;
BindingFactory factory;

Name name = naming.ByteToName("bar");
InterfaceServer itf = (InterfaceServer)factory.bind(name);
itf.foo(...);
```

FIGURE 3.10 – Liaison avec une interface depuis un composant client

Cet exemple permet de bien comprendre deux choses. Premièrement, l'utilisation des interfaces peut être complètement séparée de l'implantation du composant. Par exemple, les interfaces `NamingContext`, `BindingFactory` et `Name` sont utilisées sans aucune information sur le ou les composants les implantant. Deuxièmement, nous pouvons voir que quelle que soit la configuration de la liaison, l'invocation à une méthode d'une interface s'effectue de manière similaire grâce au modèle de programmation uniforme.

3.2.3 Domaine

Un système n'est pas une structure homogène plate. Une infrastructure logicielle répartie est par exemple organisée en grappes « cluster », elles-mêmes organisées en nœuds. Localement, sur chaque nœud s'exécute une instance d'un système centralisé. Chaque système centralisé est organisé en espaces d'adressage, comme par exemple un noyau et des processus. L'objectif du concept de domaine est de modéliser l'ensemble de cette organisation.

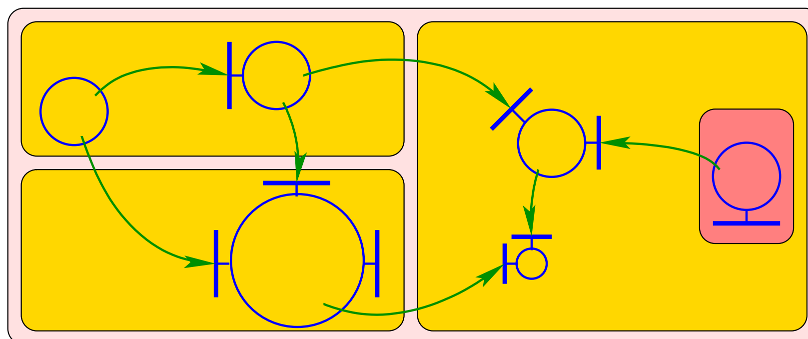


FIGURE 3.11 – Système à base de composants, de liaisons et de domaines

Un système conforme à l'architecture THINK se présente comme un ensemble de domaines s'exécutant en parallèle. Un *domaine* comprend un ensemble de composants, appelé le *contenu* du domaine. Ce contenu est placé sous le contrôle d'un composant, appelé le *conteneur* du domaine. Le concept de domaine est introduit dans THINK pour réifier les différentes formes de sous-systèmes manifestes dans des infrastructures logicielles réparties. Parmi les exemples de domaines intéressant directement THINK, nous pouvons citer les suivants.

- *domaine de désignation*. ensemble de composants régis par un même ensemble de conventions de désignation et d'allocation de noms. Un contexte de désignation, manifesté par le type `NamingContext` du canevas présenté à la section précédente, correspond ainsi au composant conteneur d'un domaine de désignation dans THINK.
- *domaine de ressources*. ensemble de composants dépendant d'un même ensemble de ressources, sous le contrôle d'un même gestionnaire de ressources. Les notions de *capsule* dans le modèle de référence ODP [ISO/IEC 1995b] (par ex. processus dans un système d'exploitation classique), ou de machine (*nœud de traitement* dans le modèle de référence ODP) en fournissent des exemples.
- *domaine de défaillance*. ensemble de composants susceptibles de défaillir de façon conjointe, selon les mêmes modes, sur occurrence des mêmes fautes. Des domaines de ressources constituent en général également des domaines de défaillances (par exemple nœud « fail-safe »).
- *domaine de sécurité*. ensemble de composants dépendant d'un même ensemble de politiques de sécurité (contrôle d'accès notamment). Un sous-réseau ou un serveur protégés par un pare-feu fournissent de bons exemples de domaines de sécurité.

L'architecture THINK impose peu de contraintes sur la forme générale des domaines. Outre les exemples donnés ci-dessus, la notion de domaine recouvre également des constructions telles que conteneurs de composants (par exemple conteneurs EJB ou Corba Component Model). Pour une première tentative de formalisation de la notion de domaine telle que présentée ici, on pourra consulter [Stefani et al. 2000].

Le conteneur d'un domaine a des rôles différents. Le premier rôle est d'offrir une structure d'accueil pour l'exécution des composants. Le deuxième rôle est de permettre la configuration et la manipulation du contenu d'un domaine. Le troisième rôle est de contrôler la sémantique du domaine. Ces trois rôles sont présentés ci-dessous. Ils ont été séparés pour des raisons de clarté, ce qui n'est pas le cas en pratique. Un exemple de conteneur de domaine est donné à la figure 3.12, nous reviendrons par la suite sur la signification des interfaces.

Structure d'accueil

Un domaine est avant tout une structure d'accueil pour l'exécution des composants correspondant à son contenu. Quel que soit le type de domaine, le code du composant est le même, il y a transparence du composant vis-à-vis du domaine. Nous avons vu qu'un composant utilise, via des liaisons, d'autres composants. Le rôle de la *structure d'accueil* est de mettre en place les liaisons implicites nécessaires au composant pour son exécution.

La structure d'accueil se compose donc d'un chargeur de composants et d'un ensemble de *noms bien connus*, dont la connaissance est locale. Le chargeur, qui connaît ces noms, peut implicitement

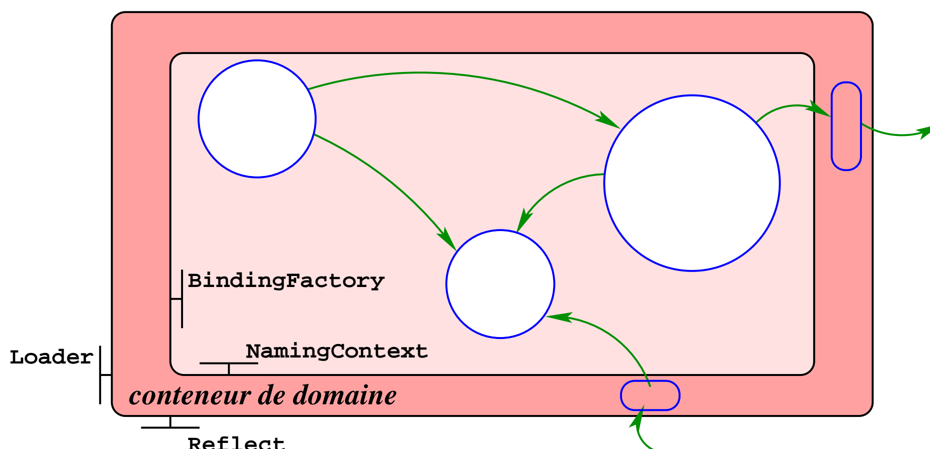


FIGURE 3.12 – Conteneur de domaine

mettre en place les liaisons et permettre ainsi l'exécution du composant. Ce chargeur est soit statique, comme le programme `ld`, soit dynamique.

Le nombre de noms bien connus est arbitraire et dépend du type de l'implantation du modèle. Néanmoins, au moins deux noms biens connus sont nécessaires, celui d'un contexte de désignation qui permet d'exporter ses interfaces et celui d'une usine à liaisons qui permet au composant de se lier avec les interfaces qu'il requiert. Sans ces deux noms, le composant est dans l'impossibilité d'interagir avec l'extérieur, car il n'aurait aucun moyen de créer des liaisons[3].

Configurateur

Un domaine est aussi un conteneur de composants. Ce conteneur fournit diverses fonctions permettant la manipulation des composants. Il maintient pour cela le graphe de composition du contenu du domaine. Cette connaissance peut facilement être obtenue si les usines à liaisons présentent dans un domaine mémorise l'ensemble des liaisons qu'elles ont créé.

Contrôleur

Le contrôleur d'un domaine implante la sémantique du domaine, comme par exemple des fonctions de sécurité ou de persistance. Le contrôleur doit jouer le rôle d'un filtre sur l'ensemble des requêtes envoyées aux composants du domaine et émises depuis l'extérieur. Il doit être capable d'intercepter et de gérer toutes les requêtes sortantes ou entrantes du domaine. Il peut alors ajouter aux liaisons des traitements permettant d'assurer la sémantique du domaine. Ces traitements, ajoutés sous forme de composants de liaisons, font intégralement partie du contrôleur. Par exemple, un domaine de sécurité peut vérifier les accès aux composants. D'autres raisons peuvent nécessiter de mettre en place des intercepteurs. C'est le cas quand l'interaction entre deux domaines nécessite des traitements spéciaux, comme pour un appel système ou pour un IPC. Un composant de liaison joue alors un rôle de contrôle et un rôle de communication.

[3] C'est le problème de la poule et de l'oeuf.

Avant de continuer plus avant, il est nécessaire de bien fixer certains détails. Tout d'abord, comme nous avons pu le voir, le concept de domaine est une extension conservatrice du concept de liaison flexible car ses fonctions et sa sémantique sont implantées en utilisant la liaison. Deuxièmement, le contrôleur, mais aussi le conteneur, sont intégralement construits en se basant sur le contexte de désignation et l'usine à liaisons interne au domaine.

Canevas logiciel

Le concept de domaine est manifesté dans THINK par un canevas logiciel minimal, décrit ci-dessous. La forme générale d'un domaine est donnée à la figure 3.12. Sur cette figure, nous voyons apparaître les différentes notions que nous venons d'aborder. À savoir, les composants, les intercepteurs sur les liaisons avec l'extérieur du domaine, et le composant implantant les services du domaine. En effet, un domaine est aussi un composant dans le domaine englobant.

Pour connaître le graphe de composition du domaine et capter les interactions, un domaine est construit en utilisant des contextes de désignation et des usines à liaisons. Il doit donc au minimum fournir une implantation des interfaces `NamingContext` et `BindingFactory`, voir la figure 3.7 et la figure 3.8.

Le domaine joue aussi le rôle de chargeur de composant. L'interface `Loader`, voir figure 3.13, correspond au type commun des chargeurs dynamiques. L'opération `load` permet le chargement d'un composant désigné par exemple par son nom de fichier binaire. Cette opération retourne un résultat de type `Component` identifiant la description mémoire du composant. L'opération duale `unload` permet le déchargement du composant identifié par le paramètre de type `Component`.

```
interface Component {}
interface Loader {
   Component load(String fname);
   void      unload(Component component);
}
```

FIGURE 3.13 – Interface du chargeur dynamique de composants

L'interface `Reflect`, voir figure 3.14, correspond au type d'introspection du contenu d'un domaine. Cette interface offre deux méthodes `getComponent` et `getBinding`. La première méthode permet d'obtenir l'ensemble des composants d'un domaine. Cette opération ne réifie pas les composants de liaisons, et donc ne réifie pas les composants d'interceptions. Cette opération retourne un tableau de `Component`. La deuxième méthode permet d'obtenir l'ensemble des liaisons à l'intérieur d'un domaine. Cette opération retourne un tableau de `Binding`, voir la figure 3.14. Cette interface propose deux opérations `from` et `to` retournant un résultat de type `Component`. Elles permettent d'identifier les deux composants mis en jeu dans une liaison.

L'interface `Domain`, voir figure 3.15, correspond au type commun d'un domaine. Cette interface correspond à celle exportée par le composant du domaine auprès du domaine englobant. Elle est simplement une extension des interfaces `Loader` et `Reflect` que nous venons de voir. L'interface `DomainInt`, voir figure 3.15, correspond au type commun d'un domaine vu depuis les composants interne du domaine. Cette interface étend `Domain`, mais aussi `NamingContext` et

```
interface Binding {
   Component from();
   Component[] to();
}
interface Reflect {
   Component[] getComponent();
   Binding[] getBinding();
}
```

FIGURE 3.14 – Interface d'introspection

BindingFactory. Cette interface est accessible depuis les composants par un ou plusieurs noms biens connus.

```
interface Domain extends
              Loader,
              Reflect {
}
interface DomainInt extends
                Domain,
                NamingContext,
                BindingFactory {
}
```

FIGURE 3.15 – Interface Domain

Nous avons vu que la structure d'accueil doit au minimum fournir les noms biens connus du contexte de désignation et de l'usine à liaisons. En fait, il s'agira du nom de contexte de désignation et du nom de l'usine à liaisons fournis par le contrôleur et le conteneur du domaine.

3.2.4 Ressource

Ces concepts de liaison et de domaine peuvent s'appliquer dans des systèmes de toutes formes ; réparti, monolithique, micronoyau, exonoyau, embarqué et dédié à une application. Néanmoins, tous les composants d'un système n'ont pas le même rôle. Par exemple, il y a des ressources et des gestionnaires de ressources, comme les fils d'exécution et l'ordonnanceur. Il est donc intéressant de faire ressortir ces notions.

L'objectif du modèle de ressource est de fournir une modélisation cohérente et complète de toutes les ressources du système quel que soit le niveau d'abstraction. Par exemple, une approche cohérente doit convenir pour la gestion du processeur, de la mémoire physique ou des ressources réseaux. De façon similaire, le modèle doit inclure les ressources de plus haut niveau comme la mémoire virtuelle, les processus ou les connexions réseaux. Le modèle proposé ici est directement issu des propositions ReTINA [Blair et al. 1999, ReTINA 1999].

Dans ce modèle, le rôle d'un gestionnaire est de permettre le partage d'une ressource ou d'un ensemble de ressources entre plusieurs utilisateurs. Un gestionnaire fournit donc aux utilisateurs une vue abstraite de la ressource qu'il gère. Il contrôle aussi la manière dont les ressources sont utilisées. La figure 3.16 montre comment une ressource de haut niveau est construite à partir d'une ressource de bas niveau par un gestionnaire.

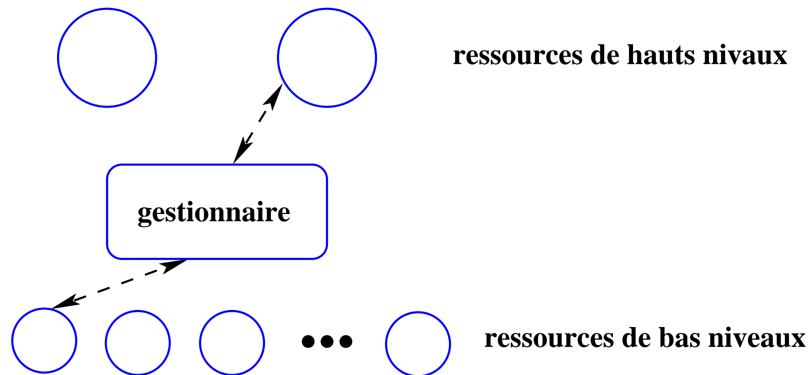


FIGURE 3.16 – Modélisation d'un gestionnaire de ressources

Le modèle de ressource n'est en fait qu'un patron de conception destiné à faciliter la compréhension du code et à simplifier le développement de composants systèmes. Les interfaces décrites dans ce modèle fournissent un ensemble d'interface. Les méthodes sont données à titre indicatif pour expliquer la fonctionnalité attendue. Mais les méthodes réellement utilisées peuvent nécessiter plus ou moins d'arguments et la sémantique indiquée peut varier d'une implantation à une autre.

Ressources abstraites

Dans ce modèle, toutes les ressources de hauts niveaux ou de bas niveaux sont des composants et sont du type `AbstractResource`, voir figure 3.17. La méthode `release` permet de libérer la ressource quand elle n'est plus utilisée. La ressource peut donc être réutilisée par ailleurs. La seule contrainte est de ne plus utiliser la référence sur la ressource après l'appel de cette méthode.

```
interface AbstractResource {
   void            release();
}
```

FIGURE 3.17 – Interface `AbstractResource`

Gestionnaires

Différentes catégories de gestionnaires peuvent être conçues. Par exemple, un gestionnaire peut multiplexer des ressources de hauts niveaux sur un ensemble de ressources de bas niveaux, comme avec un ordonnanceur. Soit, il peut les coupler directement sur des ressources de bas niveaux, comme

avec un allocateur de mémoire. De façon similaire, un gestionnaire peut composer des ressources de bas niveaux pour créer une ressource plus abstraite de haut niveau, comme avec un processus. La forme générale d'un gestionnaire est donnée par l'interface `ResourceManager` donnée à la figure 3.18. Cette interface permet uniquement de qualifier un composant, elle est purement indicative et aucune méthode n'est requise.

```
interface ResourceManager { }
```

FIGURE 3.18 – Interface `ResourceManager`

Quand un programme nécessite l'accès à une ressource, il a deux moyens à sa disposition. Soit, il demande la création d'une nouvelle instance de la ressource. Soit, il possède le nom de la ressource et il demande l'établissement d'une liaison avec la ressource. Nous supposons ici que lors de la création d'une ressource, la liaison est aussi établie. Dans le modèle, la création d'une instance est effectuée par une *usine* « factory », la création d'une liaison est effectuée par un *lieur* « binder ». Tous deux étendent le gestionnaire et diffèrent uniquement dans la manière de fournir les ressources. Ils sont décrits ci-dessous.

- Comme mentionné précédemment, une usine permet la création de nouvelles ressources du type `AbstractResource`. Une usine fournie l'interface `ResourceFactory` décrite à la figure 3.19. La méthode `create` donne en résultat une référence sur la ressource créée. Ses arguments doivent contenir les informations requises par l'usine pour construire et gérer la ressource, comme par exemple la taille de la ressource à créer, la qualité de service requise, etc. Lorsque la méthode `release` est invoquée sur la ressource, l'usine peut ne pas la détruire effectivement, mais conserver la ressource en vue d'une réutilisation future, on parle alors de groupe de ressource « pool ».

```
interface ResourceFactory extends ResourceManager {
   AbstractResource create(...);
}
```

FIGURE 3.19 – Interface `ResourceFactory`

  La notion d'usine permet de modéliser de nombreux concepts d'un système d'exploitation. Par exemple, l'allocateur de mémoire physique ou virtuelle, l'ordonnanceur, etc.

- Posséder un nom sur une ressource ne signifie pas pouvoir interagir avec. La ressource peut par exemple être inutilisable ou inaccessible dans sa forme courante. Le rôle du lieur est de rendre accessible une ressource, son interface `ResourceBinder` est donnée à la figure 3.20. La méthode `bind` met en place les mécanismes permettant l'accès à la ressource. Le nom de cette ressource, permettant sa désignation, doit être passé en paramètre de la méthode. Elle retourne une référence permettant d'interagir avec la ressource. Le modèle permet potentiellement au lieur de retourner, non pas, une référence directe sur la ressource, mais une référence sur un composant intermédiaire. Ce qui offre la possibilité de placer une indirection, comme par exemple un mandataire ou un composant de synchronisation comme un moniteur.

```
interface ResourceBinder extends ResourceManager {
   AbstractResource bind(...);
}
```

FIGURE 3.20 – Interface `ResourceBinder`

Dans le modèle d'architecture, le lieur correspond, au nom près, aux usines à liaisons, que nous avons vu à la section 3.2.2 sur les liaisons flexibles. L'usine à liaisons est uniquement une utilisation concrète de l'interface du lieur et doit en toute rigueur étendre cette interface.

Application du modèle de ressources à la gestion du processeur

Pour illustrer l'utilisation du modèle de ressources, nous étudions comment il peut être instancié pour la gestion de l'activité dans un système. Comme nous l'avons vu au chapitre 2, l'activité dans un système est modélisée par des fils d'exécution qui sont couplés, par un ordonnanceur, sur un ou plusieurs processeurs. Les processeurs et les fils d'exécution peuvent être vus comme des ressources abstraites qui permettent de faire évoluer le système.

Un fil d'exécution offre l'interface `Thread`, voir la figure 3.21. La méthode `run` permet de démarrer l'activité par exemple sur une méthode. L'interface des fils d'exécution peut par exemple être enrichie de méthodes permettant de changer les paramètres d'ordonnancement.

```
interface Thread extends AbstractResource {
   void     run(Method meth);
}
```

FIGURE 3.21 – Interface `Thread`

Un ordonnanceur est en fait une usine capable de créer des fils d'exécution et de les multiplexer sur une ressource de plus bas niveau, qui peut être le processeur ou un autre fil d'exécution. Ce deuxième cas est par exemple mis en œuvre avec les ordonnanceurs applicatifs, comme les « threads Posix ». L'interface d'un ordonnanceur, nommée `Scheduler`, est donnée à la figure 3.22. Comme nous pouvons le voir cette interface supporte la méthode `create`, héritée de `ResourceFactory` qui retourne un nouveau `Thread`. Ici, cette méthode prend en paramètre le paramètre initial d'ordonnancement `SchedParams` qui peut être une priorité, une date limite, etc. L'interface de l'ordonnanceur peut aussi comporter des méthodes de synchronisations comme des méthodes `wait` et `notify`, mais nous y reviendrons en détail au chapitre 5 sur la réalisation.

3.3 Conclusion

Le modèle d'architecture s'applique à tous les systèmes qu'ils soient centralisés ou qu'ils soient répartis indépendamment de ou des machines sous-jacentes et du ou des langages de programmation. Le concept de liaison flexible permet de modéliser et d'implanter, généralement par la génération

```
interface Scheduler extends ResourceFactory {
   Thread   create(SchedParams params);
}
```

FIGURE 3.22 – Interface `Scheduler`

automatique de composants de liaisons, l'ensemble des interactions que l'on rencontre dans un système ; appel système, remontée système, LRPC, RPC, etc. Le concept de domaine permet de modéliser l'ensemble des isolations comme les contraintes physiques avec la répartition et les contraintes de protection avec les processus. Le modèle de ressource facilite la compréhension du code et simplifie le développement des composants du système.

Dans le chapitre suivant, nous nous intéressons à l'implantation et à l'utilisation dans un système d'exploitation de ces différents concepts. Il est important de noter les interfaces définies dans ce chapitre sont directement exploitées dans l'implantation décrit au chapitre suivant.

Chapitre 4

Instanciation de l'architecture THINK dans les systèmes d'exploitation

Dans le chapitre précédent, nous avons présenté l'architecture logicielle THINK. Nous nous intéressons maintenant à l'utilisation concrète de cette architecture pour la construction de systèmes d'exploitation. Nous étudions ici comment implanter de façon efficace le concept d'interface dans un langage qui n'en est pas pourvu et comment automatiser, via des outils, la génération des composants de liaisons. Même si l'architecture THINK est instanciée sur une architecture matérielle particulière, ici le POWERPC, cette instanciation est indépendante de l'architecture matérielle. Nous étudions aussi comment implanter les domaines en vue d'offrir la transparence requise. Nous voyons comment est réalisée la composition des différents composants d'un système et quels sont les outils de développement associés. Cette implantation de l'architecture THINK est une proposition, cette architecture peut aussi être instanciée différemment.

4.1 Aperçu de l'implantation

L'architecture logicielle THINK repose principalement sur le concept de composant, sur le concept d'interface et sur le concept de liaison flexible. Nous survolons brièvement dans cette section l'implantation, nous la décrivons en détail dans la suite du chapitre.

4.1.1 Composants

Un composant est une structure de l'environnement d'exécution. Il peut être présenté sous une forme exécutable binaire. Nous utilisons le format binaire portable « Executable and Linking Format » (ELF) [TIS 1993] pour la représentation de l'exécutable d'un composant. Ce format binaire est généré par tous les compilateurs de code natif, comme l'assembleur, le C ou le C++. Dans le prototype actuel, le code des composants est programmé en C et en assembleur pour les composants de plus bas niveau. Le code binaire d'un composant peut être lié statiquement à un exécutable de système, il peut aussi être chargé dynamiquement durant l'exécution du système. Quelques règles sont définies, par exemple la définition d'une routine d'initialisation.

4.1.2 Interfaces & liaisons

Nous utilisons un sous-ensemble du langage Java comme langage de description des interfaces. L'implantation concrète des liaisons nécessite de fixer un format de représentation binaire des interfaces. L'intérêt de fixer un format binaire est double. Tout d'abord il offre le modèle de programmation uniforme. Ensuite, il permet de ne pas être restreint à un langage de programmation unique. Les composants peuvent ainsi être programmés dans un langage quelconque à la condition que ce langage permette de générer le format de représentation binaire des interfaces éventuellement via l'utilisation d'un environnement d'exécution.

Comme cette représentation est binaire, elle permet l'utilisation de langage de bas niveau très efficace comme le C ou de très bas niveau comme l'assembleur, évitant ainsi la perte de performance inhérente à un langage de haut niveau comme Java ou Modula. Par ailleurs certains composants, comme les composants de liaisons, peuvent être directement générés en assembleur afin d'optimiser les interactions. L'implantation des composants et des liaisons est décrite en détail à la section 4.2

4.1.3 Composition des composants

Une fois le code des composants programmés, il est nécessaire de les composer statiquement. Pour cela, il faut connaître les interfaces implantées par un composant et les interfaces qu'il requiert pour son exécution. Pour décrire les dépendances des composants, nous utilisons le langage XML [XML] que nous enrichissons de quelques balises. Le concepteur d'un composant doit donc décrire dans un fichier XML les interfaces de son composant. Il doit aussi décrire dans des fichiers Java les interfaces nouvelles implantées par son composant. La composition des composants est décrite à la section 4.4.

4.1.4 Portabilité

L'architecture logicielle THINK est universelle. Comme nous le voyons à la section 4.2, cette implantation de l'architecture est portable, car le format binaire ne dépend pas d'un processeur. Il s'agit uniquement d'une représentation mémoire des interfaces. Les conventions d'appels, comme le passage de paramètres, restent celles spécifiées pour le processeur. Bien évidemment, certains composants codés en assembleur ou spécifiques à une architecture de machine ou de processeur ne sont pas portables. Néanmoins, tous les autres composants peuvent être portables. Nous reviendrons sur cela au chapitre 5.

4.2 Implantation du composant et de la liaison

Nous avons vu au chapitre 3 que le concept de liaison flexible offre un modèle de programmation uniforme offrant la transparence d'accès. La flexibilité permet de définir ses propres composants de liaison optimisés en fonction des contraintes d'architecture sous-jacente. Dans cette section, nous allons commencer par décrire les mécanismes proposés. Puis, à la sous-section 4.2.4 nous donnons des exemples de programmation des liaisons et des composants qui illustrent les mécanismes proposés.

Toutes les propositions d'implantation du code des composants faites ici ne sont pas spécifiques à un langage de programmation, ni à un processeur. Néanmoins, comme le prototype est codé en C,

les exemples d'illustration sont donnés dans ce langage. Nous n'utilisons pas le C++, car les gains apportés par les classes et l'héritage sont potentiellement utiles mais ne sont pas pertinents ici. Ce qui apporte est le concept d'interface qu'il aurait aussi été nécessaire de mettre en place avec le C++. De plus, l'environnement d'exécution du langage requiert un allocateur de mémoire alors qu'on peut envisager des systèmes sans allocateur.

4.2.1 Représentation binaire des interfaces

L'intérêt de définir une représentation binaire des interfaces est de permettre l'utilisation du concept d'interface avec des langages n'offrant pas ce mécanisme, c'est par exemple le cas pour le C ou l'assembleur. Le mécanisme construit doit être aussi peu coûteux que possible pour ne pas gaspiller tout l'avantage du modèle et surtout pour ne pas détériorer le coût des interactions entre deux composants, principalement si les deux composants sont situés dans un même domaine. Par exemple, le coût d'une liaison dans le cas d'une interaction locale ne doit pas être supérieur à celui d'une indirection simple similaire à celui mis en œuvre lors de l'appel à une méthode virtuelle dans les langages à objets.

Analyse

Le concept d'interface permet de séparer une implantation de son utilisation. Quand un utilisateur souhaite appeler une interface, il est nécessaire qu'il connaisse l'ensemble des méthodes composant celle-ci. Nous voyons donc apparaître la nécessité d'avoir une structure ou un tableau référençant toutes les méthodes de l'interface.

Mais à un même code de composant peut être associé plusieurs instances. Par exemple, dans un système, plusieurs sessions IP peuvent être ouvertes, mais le code de traitement est identique et seulement les données sont différentes. Il est donc nécessaire de donner en paramètre aux méthodes de l'interface la référence vers les données du composant. Bien évidemment, l'utilisateur du composant ne connaît pas la structure des données du composant et ne peut donc pas interpréter son contenu.

Une référence vers une interface est donc composée d'une référence vers les méthodes et d'une référence vers les données, c'est-à-dire un couple de deux références. Ces références sont en fait des pointeurs. D'une façon générale, il y a deux solutions pour représenter une référence d'interface.

- La référence d'interface peut directement être le couple. C'est-à-dire que le passage de référence d'interface nécessite de passer les deux valeurs du couple. Malheureusement, cette construction est totalement inefficace sur les processeurs de type RISC[1], même si elle permet de supprimer l'indirection engendrée par l'utilisation des interfaces.
- La référence d'interface peut être un pointeur sur le couple. En conséquence, la référence peut être partagée. Mais en plus d'allouer le composant, il est nécessaire d'allouer de la mémoire pour chaque référence. Or, l'allocation est aussi très coûteuse et nécessite un allocateur de

[1] Référencer les interfaces en utilisant des structures de deux pointeurs double la taille des références. Cela consomme plus de mémoire et double les instructions de manipulations des références. Mais ce problème n'explique pas à lui seul l'inefficacité de cette solution. En effet, sur un processeur de type RISC, les arguments entiers des méthodes sont passés par convention via les registres et ne sont pas empilés. Cela permet d'effectuer des appels sans accéder à la pile. Or, dans les conventions d'appel, les structures sont toujours mises sur la pile (il n'y a pas de passage de structure via un ensemble de pointeur). Donc, au lieu de passer la référence, via un pointeur, elle est passée via la pile. Cela nécessite quatre accès mémoire supplémentaires pour chaque passage de référence, ce qui casse le pipeline du processeur.

mémoire. Alors qu'on peut envisager des systèmes sans allocateur de mémoire. Il faut donc que notre proposition autorise ce genre de chose.

Ces deux méthodes sont coûteuses, il est donc nécessaire d'envisager des optimisations pour THINK. Tout d'abord, pour éviter le passage par structure, nous utilisons un pointeur vers cette structure, ce qui requiert un allocateur. Pour éviter cela, nous proposons des optimisations utilisables en fonction du composant. Tout d'abord, si un composant exporte une seule interface, il est possible d'associer la référence vers les méthodes de l'interface directement dans les données du composant et éviter ainsi l'allocation de la structure. Ce mécanisme est en fait celui utilisé par les compilateurs de types C++[2]. Ce mécanisme est aussi utilisable si les données d'un composant sont allouées statiquement. Dans ce cas, la structure est, elle aussi, allouée statiquement.

Référence & descripteur d'interface

Cette proposition d'implantation permet des optimisations sur les références tout en offrant le modèle de programmation uniforme. Ce mécanisme consiste donc à séparer l'accès aux méthodes de l'accès aux composants. Nous pouvons constater que ce modèle est exactement celui proposé par le concept d'interface. Ainsi, l'utilisateur n'a jamais accès au contenu du composant mais seulement aux méthodes de l'interface exportée.

Le mécanisme flexible proposé est basé uniquement sur la manipulation d'une référence d'interface se résumant à un pointeur vers un *descripteur d'interface*. Par définition, nous appelons un *pointeur* un entier non signé dont la taille est fixée par le processeur et dont la valeur est une adresse mémoire. Un descripteur contient au minimum en en-tête un pointeur vers la structure référençant les méthodes de l'interface, voir la figure 4.1. L'invariant du mécanisme est le descripteur d'interface dont le rôle est de référencer en en-tête la structure contenant les références des méthodes de l'interface.

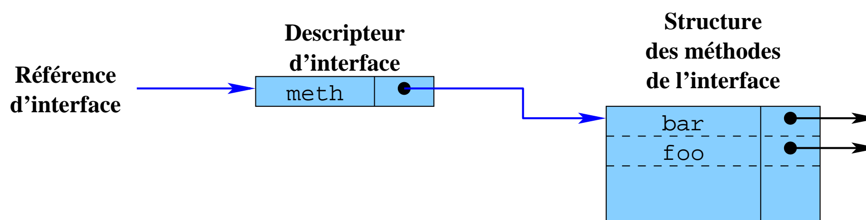


FIGURE 4.1 – Mécanisme de représentation des références d'interface

Convention d'appel

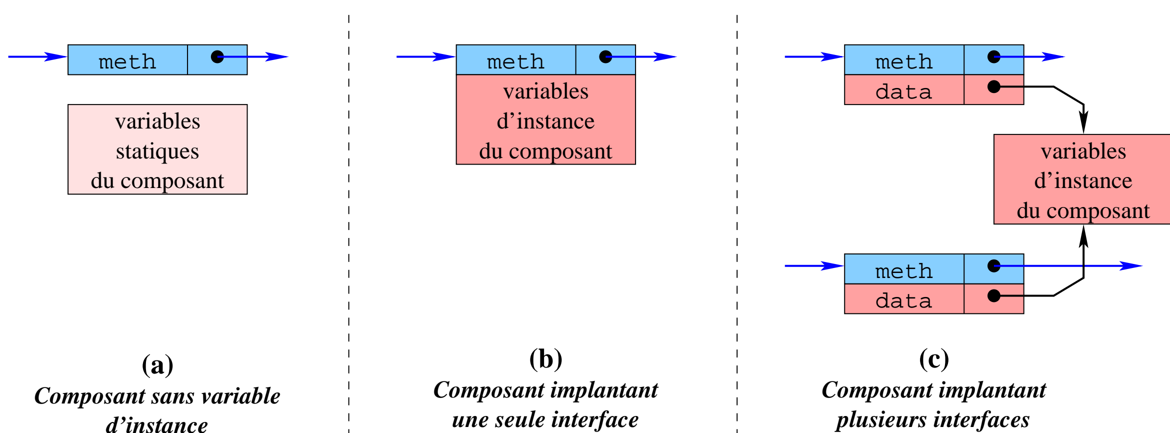
Pour que le composant puisse utiliser ses données, il est nécessaire de passer en paramètre la référence d'interface. En fait, nous ajoutons l'équivalent du paramètre *this* utilisé avec le langage C++. Néanmoins, nous le passons explicitement alors qu'il est passé implicitement avec le C++. Pour toutes les conventions relatives aux passages des arguments et à la récupération du résultat, nous respectons les spécifications d'appels du Système V accompagnant les processeurs, voir [System V 1995] pour le POWERPC.

[2] Java, qui fonctionne en interprété, utilise une table de « hash » nécessitant une recherche de la méthode appelée à chaque appel.

Programmation du composant

Nous voyons que, dans ce mécanisme, la référence vers les données n'apparaît pas, c'est la référence vers le descripteur qui sert aussi de référence vers les données. Il est possible d'ajouter à la suite du descripteur d'interface une référence vers les données ou directement ajouter les données. Il est donc du ressort du programmeur du composant de définir cela. Grâce à ce mécanisme, le programmeur a la possibilité de changer l'implantation mémoire de son composant en fonction de son format et des optimisations locales envisageables. Bien évidemment, l'utilisateur du composant n'a pas connaissance de ces distinctions et il effectue toujours de la même manière l'invocation sur les méthodes de l'interface. Nous pouvons citer plusieurs cas exploitant le mécanisme flexible proposé. Étudions, avec l'aide de la figure 4.2, les cas d'association entre les interfaces et les composants.

a. Lorsqu'il n'existe qu'une instance d'un composant dans tout le système et donc que les données sont toutes statiques, le descripteur d'interface et les données du composant peuvent alors être allouées statiquement par le compilateur. Ce cas peut aussi être utilisé si le composant exporte plusieurs interfaces. Le compilateur alloue statiquement un descripteur d'interface pour chaque interface. Dans ce cas, l'argument *this* peut ne pas être utilisé par le code du composant.



(a)
Composant sans variable d'instance

(b)
Composant implantant une seule interface

(c)
Composant implantant plusieurs interfaces

FIGURE 4.2 – Exemple d'utilisation du descripteur d'interface

b. Si un composant implante une et une seule interface, le descripteur d'interface peut être alloué avec les données de l'instance du composant, en mettant par exemple les données à la suite du descripteur d'interface. Le composant doit alors accéder ses données derrière le descripteur. Il y a donc ici plusieurs composants du même type, c'est le paramètre *this* qui va identifier le composant appelé.

c. Dans le cas général où il existe plusieurs instances d'un composant et qu'il implante plusieurs interfaces, aucune optimisation n'est possible et il est nécessaire d'allouer un descripteur d'interface par interface et par composant.

Représentation des méthodes de l'interface

Il y a deux façons de représenter les méthodes de l'interface. La première consiste à référencer toutes les méthodes dans un tableau, dont les entrées sont des pointeurs de méthodes non typés. Le

problème de cette solution est de supprimer la vérification des types passés en paramètres lors d'un appel. La deuxième solution consiste à typer chaque méthode de l'interface et donc à mettre les références à ces méthodes dans une structure, qui se comporte dynamiquement comme un tableau. Cette structure est générée automatiquement depuis la description de l'interface en Java. Par convention, les méthodes dans cette structure sont triées par ordre alphabétique. L'intérêt du tri est de pouvoir associer à chaque méthode d'une interface un numéro correspondant à l'index de la méthode dans le tri. Ce numéro permet d'identifier la méthode.

Lors de la construction d'un composant qui implante une interface, le concepteur doit programmer le remplissage d'une instance de cette structure avec les références vers les méthodes de son composant. La référence vers cette instance permet de compléter le descripteur d'interface.

Représentation de l'héritage

Dans le prototype actuel, seul l'héritage simple d'interface est possible. Un exemple d'héritage est donné à la figure 4.3 qui montre comment l'interface `NameExt` hérite de l'interface `Name`. L'héritage multiple, qui nécessite un mécanisme de changement de type non trivial, n'est pas pris en compte. Néanmoins, il est possible de déclarer et d'exporter plusieurs interfaces par composant. Ce qui permet d'obtenir des effets similaires à l'héritage multiple.

```
interface NameExt extends Name {
    void     foo();
}
```

FIGURE 4.3 – Héritage d'interface

L'héritage simple est géré en ajoutant les nouvelles méthodes à la suite de l'interface étendue. On obtient ainsi une structure des méthodes de la nouvelle interface qui possède en préfixe celle de l'interface héritée, voir la figure 4.4. Cela est suffisant car l'héritage simple d'interface se réduit uniquement à déclarer les méthodes de l'interface héritée dans une nouvelle interface.

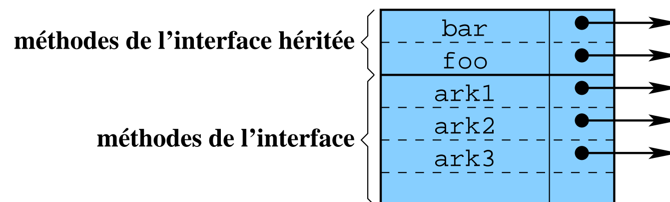


FIGURE 4.4 – Représentation de l'héritage

L'intérêt de cette solution est de pouvoir passer d'une interface héritée à celle héritant simplement sans nécessiter de changement de type dynamique. En effet, le changement de type est purement langage, car les pointeurs vers l'interface héritée et vers l'interface héritant sont égaux.

4.2.2 Langage de description d'interfaces

Dans la section précédente, nous avons défini le format et le mécanisme de représentation et d'utilisation du concept d'interface avec un langage quelconque. Cette définition est indépendante du langage de description des interfaces. Comme nous l'avons vu, nous utilisons le langage Java pour décrire les interfaces. Il est donc nécessaire de convertir cette description au format de notre mécanisme et aux types de chaque langage utilisé pour l'implantation des composants. Pour cela, nous spécifions un ensemble de règles de conversion. Ces règles sont utilisées par les outils de conversion et de génération, comme le compilateur que nous voyons à la section 4.2.3, mais aussi par les concepteurs et les utilisateurs des composants.

Conversion des types

Actuellement, aucune sérialisation des composants n'est réalisable. Seuls les types primitifs sont gérés. Ces types sont utilisés uniquement pour la description des méthodes de l'interface. C'est-à-dire pour le type des arguments et pour le type de retour. Le langage Java définit les types et les valeurs supportées, mais la conversion des types spécifie comment un programmeur voit ces valeurs.

Le tableau 4.1 récapitule la conversion des types supportés par cette implantation de l'architecture THINK vers les types binaires et les types C. Cette liste est incomplète et mériterait d'être enrichie dans les utilisations futures. Dans la mesure du possible, les types Java sont traduits par les types binaires similaires du processeur. Néanmoins, pour simplifier l'utilisation des interfaces, nous ajoutons des types non primitifs et dont la signification ici n'est pas celle de Java. Par exemple, la classe `java.lang.String`, pour les chaînes de caractères, est traduite comme un tableau de caractères terminant par le caractère nulle. La taille de la chaîne peut alors être obtenue par la méthode `strlen`. Les tableaux `X[]` sont traduits par une suite mémoire de valeurs, mais ici la taille du tableau disparaît. Elle doit donc être codée par ailleurs si sa valeur est requise. Toutes les autres classes ne sont pas supportées par les spécifications.

Type Java	Type binaire	Type C
`void`		`void`
`boolean`	8 bits non signé	`unsigned char`
`byte`	8 bits signé	`signed char`
`char`	8 bits non signé	`unsigned char`
`short`	16 bits signé	`short`
`int`	32 bits signé	`int`
`long`	32 bits non signé	`unsigned long`
`X[]`	pointeur	`X*`
`java.lang.String`	pointeur	`unsigned char*`
interface `X`	référence d'interface	`XItf`

TABLEAU 4.1 – Spécification de la conversion de types

Convention de nommage pour les interfaces

Des noms Java apparaissent à plusieurs endroits dans la description d'une interface. Il y a en fait quatre types de nom ; le nom du paquetage, le nom de l'interface, le nom de chaque méthode et le nom de chaque argument de chaque méthode. Ces noms sont utilisés pour générer les noms langages permettant de décrire dans le langage les interfaces.

- *nom du paquetage.* Pour ne pas agrandir inutilement la longueur des noms, le nom du paquetage n'est pas utilisé actuellement. En conséquence, il est impossible de définir dans deux paquetages différents deux interfaces possédant le même nom. Cette limitation peut facilement être supprimée en préfixant le nom de l'interface par le nom du paquetage.
- *nom de l'interface.* Le nom de l'interface sert à générer les structures requises par le descripteur d'interface dans le langage d'implantation du composant. L'ajout du suffixe `Meth` au nom d'une interface donne le nom de la structure des méthodes de l'interface. L'ajout du suffixe `Itf` au nom de l'interface donne le nom de la structure du descripteur d'interface. En prenant l'exemple de l'interface `Name`, nous obtenons en C les noms `NameMeth` et `NameItf`, ces structures sont illustrées à la section 4.2.4
- *nom des méthodes.* Le nom de la méthode reste inchangé dans la structure des méthodes de l'interface qui référence cette méthode. Cette facilité pose potentiellement des problèmes lors de l'utilisation d'un mot clé réservé du langage comme nom d'une méthode. En revanche, le concepteur du composant utilise les noms qu'il souhaite pour nommer ses méthodes qui sont d'ailleurs généralement statiques et donc locales à son composant. La seule contrainte est qu'il complète correctement la structure des méthodes de l'interface.
- *nom des arguments.* Les noms Java des arguments des méthodes sont complètement arbitraires et ne sont pas utilisés. Par convention, tous les arguments des méthodes sont nommés arg_1 à arg_n. Mais comme pour le nom de la méthode le concepteur du composant les nomme comme il le souhaite à l'intérieur de sa méthode.

4.2.3 Compilation des interfaces

Les interfaces écrites dans le langage de description d'interface Java, que nous venons de voir, ne sont pas utilisées telles quelles. Elles doivent être converties au format de représentation des interfaces. De plus, elles peuvent et doivent être utilisées pour la génération des composants de liaisons. Cette conversion et ces générations sont réalisées par un ou plusieurs compilateurs d'interface.

Nous voulons un *compilateur ouvert d'interface* pouvant être spécialisé par des *générateurs* de composant de liaison. Chaque type de liaison est accompagné d'un générateur générant ses propres composants de liaisons pour chaque type d'interface. Des générateurs spéciaux, appelés des *formateurs*, sont aussi définis afin de générer pour un langage donné le code de déclaration des interfaces conformes aux références et aux descripteurs des interfaces. Ce code permet à un utilisateur d'appeler une interface depuis un composant programmé dans le langage donné.

Compilateur ouvert & générateurs

Les interfaces à compiler sont tout d'abord compilées en code binaire Java « bytecode » grâce au compilateur `javac` du JDK. Ainsi, la vérification des types est laissée à la charge du compilateur

`javac` et n'est donc pas à la charge du compilateur ouvert ou des générateurs. Pour des raisons de portabilité et de simplicité d'écriture, le compilateur ouvert et les générateurs sont écrits en Java. Le protocole de communication, entre le compilateur et les générateurs, est défini par une interface qui va être présentée par la suite. La structure générale du compilateur est donnée à la figure 4.5.

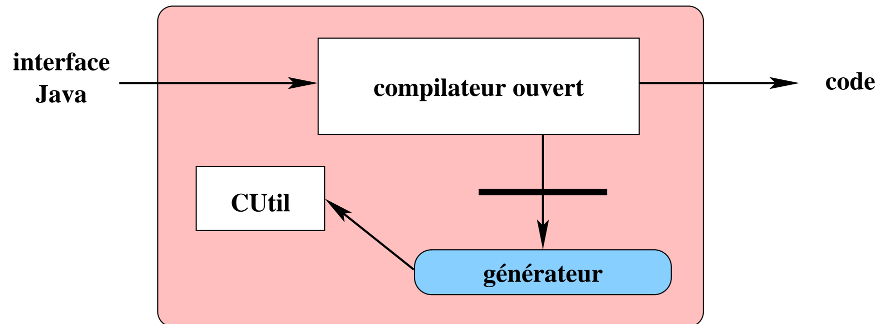


FIGURE 4.5 – Structure du compilateur

Lors d'une compilation, le générateur à utiliser est donné en paramètre au lancement du compilateur ouvert. Le fonctionnement du compilateur est le suivant. Tout d'abord, il charge dynamiquement le code binaire du générateur. Il charge ensuite le code binaire de l'interface Java à compiler. Le compilateur utilise ensuite les techniques de réflexion Java pour analyser les méthodes de l'interface et les méthodes des interfaces éventuellement héritées. Ces méthodes sont triées et, pour chaque méthode, le générateur est invoqué. C'est donc à lui de générer le code du composant de liaison. Pour simplifier le développement des générateurs, le concepteur peut utiliser les bibliothèques définies dans le compilateur, comme `CUtil`. Ces bibliothèques permettent par exemple de générer l'en-tête des méthodes C. Le générateur est aussi invoqué avant et après la compilation pour lui permettre de générer du code d'initialisation et de terminaison.

Interface des générateurs

L'interface `Generator`, voir la figure 4.6, est l'interface protocolaire utilisée entre le compilateur ouvert et un générateur. La méthode `init` est utilisée pour démarrer le générateur. Les méthodes `getName` et `getPath` sont utilisées pour obtenir le nom et le chemin du fichier généré. La méthode `writeMethod` est appelée pour chaque méthode de l'interface. Les méthodes `writeInitializer` et `writeFoot` sont utilisées pour commencer et terminer le fichier généré. Le générateur n'a donc qu'à implanter ces méthodes pour spécialiser le compilateur.

4.2.4 Exemple de codes

Pour illustrer l'implantation des composants et des liaisons, nous reprenons l'interface `Name` vue au chapitre précédent. Les exemples, donnés ci-dessous, montrent : comment cette interface est convertie en C, comment il est possible d'invoquer une méthode sur cette interface, et comment un composant peut implanter cette interface.

```
interface Generator {
   void     init(Config config) throws Exception;

   String   getPath(Class itf);
   String   getName(Class itf);

   void     writeInitializer(Class itf, PrintWriter out);
   void     writeMethod(Class itf, PrintWriter out,
                     Method method);
   void     writeFoot(Class itf, PrintWriter out) ;
}
```

FIGURE 4.6 – Interface `Generator` des générateurs

Conversion des interfaces en C

La figure 4.7 donne le code C résultant de la conversion par un formateur pour le C d'une interface au format de représentation binaire pour l'interface `Name`. Cette conversion est effectuée par un formateur pour le C. La structure `NameItf` est le type du descripteur d'interface et la référence d'interface est uniquement un pointeur sur cette structure. La structure `NameMeth` est le type de la structure référençant les deux méthodes de l'interface `Name`. Nous voyons ici le respect des conventions pour l'appel avec l'argument *this* et pour les conversions des types pour les arguments et les résultats.

```
#ifndef APIS_uORB_Name_H
#define APIS_uORB_Name_H

struct NameMeth;
typedef struct {
  struct NameMeth  *meth;
} NameItf;

#include <NamingContext.h>
struct NameMeth {
  NamingContextItf* (*getDefaultNC)(NameItf* this);
  unsigned char*   (*toByte)(NameItf* this);
};
#endif /* APIS_uORB_Name_H */
```

FIGURE 4.7 – Code C de représentation de l'interface `Name`

Appel à une méthode d'une interface

La figure 4.8 montre comment s'effectue en C un appel à une méthode de l'interface `Name`. Pour simplifier le développement, une macro réalisant l'appel a été définie.

```
NameItf* name;
name->meth->toByte(name);
```

FIGURE 4.8 – Appel de méthode d'une interface en C

Code d'un composant

La figure 4.9 donne le squelette du code C d'un composant exportant l'interface `Name`. L'implantation de ce composant, qui exporte une seule interface, est optimisée en mettant les variables de l'instance du composant (ici seulement *ptr*) à la suite du descripteur d'interface. Nous voyons la définition des deux méthodes ainsi que le remplissage de la structure des méthodes. La méthode *export* donne un exemple de code que pourrait offrir un contexte de désignation pour créer de nouveaux noms. Le code ci-dessous est d'ailleurs issu de l'implantation de la liaison locale. La variable *allocator* référence un allocateur de mémoire, nous supposons ici que son nom est bien connu, ce qui n'est pas forcément vrai. Nous reviendrons sur les interfaces et les composants de gestion de la mémoire au chapitre suivant.

4.3 Implantation des domaines

Actuellement, le concept de domaine n'est que sommairement implanté dans THINK. L'implantation ne supporte pas l'introspection du contenu d'un domaine, ni la génération de composants de liaisons réalisant le contrôle des domaines. Un chargeur de composants permet l'extensibilité dynamique des domaines, mais il n'est pas possible de supprimer de composants à un domaine. Le chargeur dynamique de composant est présenté au chapitre 5.

4.3.1 Classification des domaines

Dans l'implantation actuelle, trois classes de domaine sont présentes ; le domaine matériel, le domaine superviseur et les domaines applicatifs. Ces domaines sont ceux classiquement rencontrés dans les systèmes d'exploitation.

Domaine matériel

Nous appelons *domaine matériel* l'ensemble des ressources matérielles d'une machine. Un tel domaine est considéré comme statique et il ne peut pas évoluer au cours du temps. Comme tous les domaines, ce domaine est constitué de composants qui sont le processeur, la mémoire, les périphériques, etc. Comme tous les composants, ces composants exportent chacun une interface.

```
struct mynamedata {
  struct NameMeth* meth;
  unsigned long    ptr;
};
static NamingContextItf* getDefaultNC(NameItf* this) {
  struct mynamedata* name = (struct mynamedata*)this;
  ...
}
static unsigned char* toByte(NameItf* this) {
  struct mynamedata* name = (struct mynamedata*) this;
  ...
}
static struct NameMeth defaultNameMeth = {
  defaultName_getDefaultNC,
  defaultName_toByte
};

NameItf* export(TopItf* top) {
  struct mynamedata* name = (struct mynamedata*)
    CALL1(allocator, alloc, sizeof(struct mynamedata));
  name->meth = &defaultNameMeth;
  name->ptr = (unsigned long)top;
  return (NameItf*)name;
}
```

FIGURE 4.9 – Code d'un composant exportant l'interface Name

Domaine superviseur

Par défaut un système conforme à l'architecture THINK s'exécute dans un domaine superviseur. Un *domaine superviseur* est un domaine dont l'exécution est dans le mode superviseur du processeur permettant l'accès à toutes les ressources du domaine matériel. Un domaine superviseur correspond classiquement à un noyau de système d'exploitation.

Domaines applicatifs

Quand le système s'amorce, il met en place le domaine superviseur dans lequel s'exécute le système. En fonction des besoins, le système a la possibilité de créer des *domaines applicatifs*, par exemple en utilisant les composants de la bibliothèque KORTEX, voir chapitre 5. Les domaines applicatifs ne s'exécutent pas en mode privilégié et ne peuvent donc pas accéder directement aux ressources du domaine matériel.

4.3.2 Amorçage des domaines

L'amorçage d'un domaine superviseur ou d'un domaine applicatif doit mettre en place la structure d'accueil pour les composants. Les codes d'amorçage sont différents. Pour un domaine superviseur, l'amorçage doit mettre la mémoire et le processeur dans un état bien connu. Pour un domaine applicatif, l'amorçage est en fait un chargeur d'application.

Une fois, l'amorçage du domaine terminé, une méthode nommée `kernelstart` est appelée. Ce nom est le même pour un domaine superviseur ou pour un domaine applicatif. L'objectif de cette méthode est d'initialiser les composants du système ou de l'application et elle est programmée par le concepteur.

4.3.3 Modèle d'exécution

Cette implantation du modèle THINK n'impose aucun modèle d'exécution, comme les fils d'exécution et la mémoire. En revanche, la bibliothèque KORTEX fournit un ensemble de composants implantant différents modèles d'exécution. Ces différents composants peuvent être utilisés en fonction des besoins du système construit.

Fil d'exécution

Aucun fil d'exécution, ni les quantums de temps, ni l'ordonnanceur, ne sont imposés. Cela permet, par exemple, d'utiliser des langages avec un seul fil d'exécution, comme Esterel. Cela permet aussi de construire des systèmes sans ordonnanceur, évitant ainsi de payer le coût des changements de contexte.

Interruptions

Nous avons vu que l'exécution séquentielle du processeur peut être interrompue par des événements, comme les exceptions ou les interruptions. Cette interruption déclenche l'exécution d'un traitant système situé dans le domaine superviseur. Or, nous voulons que le domaine soit transparent pour les composants. C'est-à-dire qu'il doit être possible de déclencher des événements dans les domaines applicatifs, voir la figure 4.10.

En conséquence, l'implantation de l'architecture THINK offre la même interface basse dans les domaines applicatifs que dans le domaine superviseur. Cela permet par exemple, de faire remonter, via le domaine superviseur, un événement processeur. Cela permet aussi d'effectuer un équivalent de l'appel système dans un domaine applicatif. De plus, lorsqu'un domaine exécute un traitant d'interruption, les interruptions doivent toujours être masquées. C'est-à-dire que seulement une interruption peut être prise en compte à un instant donné.

Mémoire

Aucun gestionnaire mémoire, ni espace d'adressage, ne sont imposés. Ainsi, il est possible de construire des systèmes s'exécutant directement dans la mémoire physique, en désactivant la translation d'adresse, ou des systèmes s'exécutant dans des mémoires virtuelles. Quel que soit le modèle

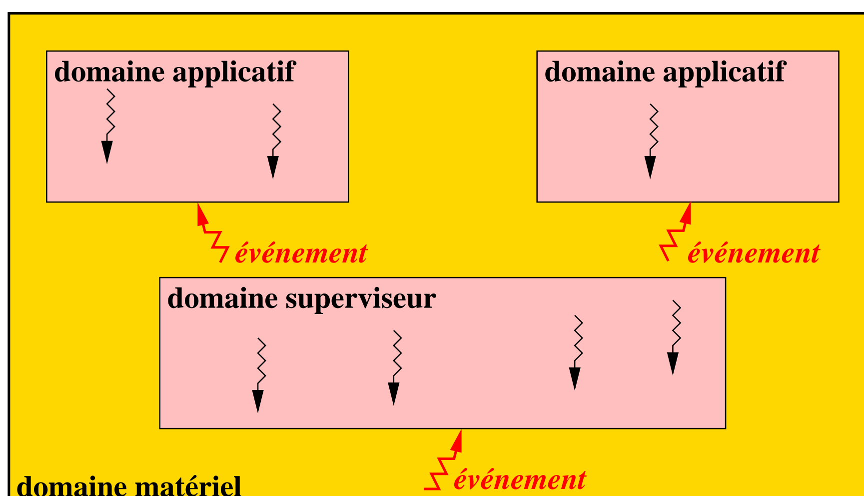


FIGURE 4.10 – Mécanisme de déroutement dans un domaine

choisi, seuls le ou les composants de gestion de la mémoire changent, mais les interfaces d'accès restent identiques. Cela assure la transparence vis-à-vis du modèle de mémoire.

4.4 Implantation de la composition

Comme nous l'avons vu au chapitre 2, le rôle de la composition est d'assembler les différents codes des composants en vue de fabriquer le système désiré. Dans cette implantation de l'architecture THINK, nous définissons un langage de description d'architecture. Ce langage permet au concepteur du code d'un composant d'exprimer les contraintes de son composant. Il permet aussi au concepteur d'un système de définir le code des composants qu'il souhaite utiliser dans son système. Cette description permet de créer le *graphe de composition* pour lequel les nœuds sont des composants et les arrêtes orientées sont des dépendances.

Ce langage permet la conception d'outils de développement facilitant la conception des systèmes. Tous ces outils sont purement statiques et ils n'ajoutent aucun coût à l'exécution car ils n'y sont pas présents. Ainsi, un lieur crée l'exécutable du système en intégrant uniquement et automatiquement les composants requis par un système. Un ordonnanceur détermine l'ordre de lancement des composants en fonction du graphe de composition. Un visualiseur offre une visualisation du graphe de composition pour un système donné. Tous ces outils vérifient la validité de la configuration du système conçu. Par exemple, ils vérifient que toutes les dépendances sont résolues, c'est-à-dire qu'au moins un composant implante une interface requise par un ou plusieurs autres composants.

4.4.1 Langage de description d'architecture

Nous utilisons XML [XML] comme langage de description d'architecture. Pour cela, nous définissons quelques balises permettant la description voulue. Un analyseur syntaxique de langage XML permet de représenter en mémoire le graphe de composition. Ce graphe de composition est

alors utilisé par les outils. Dans la mesure du possible, nous essayons de garder le même vocabulaire que le langage de description d'architecture CCM de CORBA [OMG 1999]. Les descriptions des composants et des compositions sont écrites dans des fichiers additionnels au fichier des composants. Un même fichier peut contenir plusieurs descriptions.

Description des composants

Ce langage permet au concepteur du code d'un composant de décrire pour un composant ; son nom, ses dépendances, ses interfaces, le nom du fichier binaire, etc. Nous supposons ici que le nommage des interfaces est universel, c'est-à-dire qu'il n'y a qu'un seul espace de nommage. Il ne peut donc y avoir deux interfaces du même nom. Cette règle est aussi appliquée pour le nommage des composants. Néanmoins, cela n'est pas problématique car il n'y a que très peu de contraintes sur le format des noms, presque tous les caractères sont acceptés. Les limitations proviennent du langage utilisé pour concevoir les composants. Les balises proposées pour la description des composants sont les suivantes.

- `<component name="..." class="..."/>` Cette balise permet de décrire le composant. L'attribut `name` est le nom du composant, il sert à l'identifier. Ce nom est utilisé par les outils et n'est en rien lié aux noms des interfaces. Comme nous l'avons vu, il doit être unique. L'attribut `class` est la classe du composant. Un composant peut ne pas avoir de classe ou en avoir plusieurs. Cet attribut est utilisé par le visualiseur pour restreindre l'affichage à une classe de composant. Les balises de descriptions d'un composant sont les suivantes.
  - `<optionnal value="..."/>` Cette balise permet de décrire si un composant est optionnel lors de l'exécution. Cette balise est typiquement utilisée pour les pilotes dont le composant est actif uniquement si le périphérique associé est présent sur la machine. L'attribut peut donc prendre les valeurs `true` ou `false`. Ainsi, si un composant dépend d'une interface fournie par un composant optionnel, il doit être lancé uniquement après lancement de tous les composants implantant l'interface voulue.
  - `<file name="..."/>` Cette balise permet de donner les fichiers binaires à lier lors de la création d'un exécutable nécessitant ce composant. Il peut y avoir plusieurs fichiers par composant, il peut donc y avoir plusieurs attributs de ce type par composant.
  - `<probe value="..."/>` Cette balise permet de spécifier si un composant possède un constructeur. C'est-à-dire si une méthode doit être appelée lors du lancement du composant afin qu'il s'initialise. Cet attribut peut prendre les valeurs `true` ou `false`. Par convention, le nom du constructeur est la concaténation du nom du composant et de `Probe`. Nous reviendrons sur ces conventions de nommage par la suite.
  - `<consumes interface="..."/>` Cette balise permet de spécifier pour un composant une interface requise pour son exécution. L'attribut `interface` est le nom de l'interface. Bien évidemment, il peut donc y avoir plusieurs attributs de ce type par composant.
  - `<produces interface="..."/>` Cette balise permet de spécifier pour un composant une interface exportée. L'attribut `interface` est le nom de l'interface. Bien évidemment, il peut donc y avoir plusieurs attributs de ce type par composant.

La figure 4.11 donne un exemple de description de composant pour un des pilotes de périphérique rencontrés dans les PowerMacintosh. Nous reviendrons sur le nom et la fonction des interfaces au chapitre 5 sur la bibliothèque de composant KORTEX.

```
<component name="gmac" class="drivers" class="network">
   <optionnal value="true"/>
   <file name="gmac"/>
   <produces interface="net"/>
   <consumes interface="delay"/>
   <consumes interface="irq"/>
   <consumes interface="printk"/>
   <consumes interface="space"/>
   <consumes interface="allocator"/>
   <consumes interface="FW"/>
   <consumes interface="BF"/>
   <consumes interface="trader"/>
   <consumes interface="packetfactory"/>
</component>
```

FIGURE 4.11 – Exemple de de description d'un composant

Description de la composition

Ce langage permet de décrire la composition des composants. Ici, seuls les composants requis par un système doivent être spécifiés. Tout ce qui relève des techniques d'assemblage relève des liaisons flexibles à l'exécution. Les balises proposées pour la description de la composition sont les suivantes.

- `<composite name="..."/>` Cette balise permet de décrire la composition d'un système. Nous appelons composite l'ensemble des composants composés. Cette construction n'est pas récursive et un composite n'est pas lui-même un composant, c'est un système. L'attribut `name` est le nom du système composite, c'est une chaîne de caractères. Une seule balise est requise pour la description d'un composite et est la suivante.
  - `<needs component="..."/>` Cette balise permet de lister tous les composants nécessaires à la composition du système. L'attribut `component` est le nom du composant. Bien évidemment, il peut donc y avoir plusieurs attributs de ce type par composite.

La figure 4.12 donne un exemple de description d'un composite. Nous reviendrons sur le nom des composants au chapitre 5 sur la bibliothèque de composant KORTEX.

4.4.2 Convention de nommage pour les composants

Nous avons vu que les composants sont nommés. Le nom du composant est donné de façon arbitraire par le concepteur dans la description du composant. Il est néanmoins nécessaire de définir certaines conventions de nommage. Tout d'abord pour nommer les composants générés. Deuxièmement, il est nécessaire de fixer certains noms langages, dit noms biens connus, servant de point d'entrée dans le composant.

```
<composite name="demo">
   <needs component="think"/>
   <needs component="libc"/>
   <needs component="BFlocale"/>
   <needs component="trader"/>
   <needs component="dlmalloc"/>
   <needs component="sbrk"/>
   <needs component="flat"/>
   <needs component="FW"/>
   <needs component="IRQ"/>
   <needs component="console"/>
   <needs component="offb"/>
   <needs component="F8x16"/>
   <needs component="printk"/>
   <needs component="packetfactory"/>
</composite>
```

FIGURE 4.12 – Exemple de description de la composition d'un système

Nommage des composants de liaison

Certains composants, comme les composants de liaison, sont générés par un générateur. Ils ne possèdent donc pas de description XML. Pour pouvoir être utilisés, ces composants doivent être nommés de façon unique et automatique. Par convention, le nom d'un composant construit par un générateur à partir d'une interface est la concaténation du nom de l'interface avec le nom du générateur de la liaison. Par exemple, le composant de liaison généré depuis l'interface `Null` par le générateur `stub` de la liaison distante, se nomme `Null_stub`. Il faut noter qu'une liaison peut nécessiter plusieurs générateurs, comme la liaison distante avec les talons clients et serveurs.

Noms langages dans un composant

Lorsqu'un composant débute son exécution, il fait des traitements. Par exemple, un pilote de périphérique teste si la carte qu'il gère est présente sur la machine. Puis, en fonction du résultat des traitements, il exporte sa ou ses interfaces. Le problème est de savoir comment lancer l'exécution du composant. Sachant qu'un composant est soit lancé au lancement du système soit, lors de son chargement dynamique. Le même problème survient pour la terminaison des composants.

Pour pouvoir être lancé et terminer, un composant doit fournir deux méthodes dont les noms sont bien connus. Ces méthodes s'apparentent aux constructeurs et aux destructeurs dans les langages à objets comme C++. Par convention, le nom de la méthode de lancement du composant est la concaténation du nom du composant et de `Probe`[3], le nom de la méthode de terminaison est la concaténation du nom du composant et de `Done`. Par exemple, le composant `gmac` doit offrir une méthode de lancement `gmacProbe` et une méthode de terminaison `gmacDone`. Si un composant n'a pas besoin d'être lancé ou terminé, il n'a pas besoin de fournir les deux méthodes. Il faut noter qu'en C, ces

[3] `Probe` pour des raisons historiques.

méthodes doivent être déclarées comme non statiques dans le code du composant pour pouvoir être accédées depuis l'extérieur.

Chaque composant de liaison généré possède un nom langage permettant de créer des instances du code du composant de liaison. Par convention, le nom langage de ce constructeur est le même que le nom du code du composant. Les arguments à donner au constructeur dépendent du type de liaison. C'est l'usine à liaison correspondante qui s'occupe de demander la création du composant et qui donne les arguments requis par le constructeur.

4.4.3 Outils

Le rôle des outils de développement est de simplifier le développement des composants et des systèmes ou des applications. Ils réalisent différents services permettant d'automatiser certaines tâches. Ils permettent aussi de mieux appréhender la structure des systèmes grâce à une représentation graphique du graphe de composition. Ils contrôlent aussi la validité sémantique de la composition. Actuellement, ils contrôlent uniquement si d'une part toutes les interfaces requises par l'ensemble des composants sont toutes implantées par au moins un composant et d'autre part si tous les composants requis par une composition existent. Ces vérifications sont très utiles pour un programmeur de système. Tous ces outils sont purement statiques et ne résident pas en mémoire durant l'exécution du programme.

Pour des raisons de portabilité, tous les outils sont écrits en Java. Ils prennent en entrée l'ensemble des fichiers XML décrivant d'une part tous les composants et d'autre part la composition d'un système. Un analyseur syntaxique permet de contrôler la validité des fichiers. Un analyseur sémantique permet de contrôler la validité de la configuration et il crée une représentation mémoire du graphe de composition. Trois outils de développement existent ; un lieur d'exécutable, un ordonnanceur de composants et un visualiseur de composition.

Le lieur

Cet outil, appelé lieur, crée l'exécutable à partir de la description de la composition du système. Pour cela il détermine les fichiers binaires associés à chaque composant requis. Cela est possible grâce à l'information renseignée, dans à la balise `file`, par le concepteur du composant.

En fait, le travail réalisé par cet outil est un parcours de tous les nœuds du graphe de composition. Ce parcours donne l'ensemble des composants qui donne une liste des fichiers requis, voir figure 4.13. Cette liste de fichiers est alors passée en paramètre d'un lieur statique de code, par exemple `ld`.

$$\left\{ \begin{array}{c} fichier_1.xml \\ \vdots \\ fichier_n.xml \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} composant_1 \\ \vdots \\ composant_m \end{array} \right\}_{m>=n} \Longrightarrow \left\{ \begin{array}{c} fichier_1.o \\ \vdots \\ fichier_p.o \end{array} \right\}_{p>=m}$$

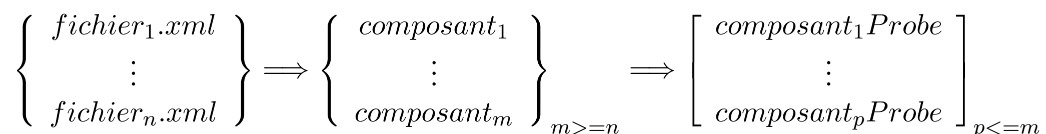
FIGURE 4.13 – Fonctionnement du lieur de composant

L'ordonnanceur

Cet outil, appelé ordonnanceur, automatise le processus d'initialisation des composants. En effet, déterminer l'ordre de lancement des composants est une opération fastidieuse et difficile qui requiert une connaissance approfondie des composants. Dans un système monolithique ou dans un système fixe, un programmeur expert peut écrire avec soin une méthode qui appelle tous les constructeurs dans le bon ordre, une fois pour toute. Ce n'est pas une bonne option dans THINK, ou l'ordre correct dépend des composants participant à la configuration. De plus, le moindre changement dans la configuration du système peut remettre en question l'ordre de lancement. Cette opération ne peut donc pas être réalisée par un novice.

Cet outil calcule l'ordre de lancement des composants en fonction de leurs dépendances et de la composition du système et devient donc une aide précieuse dans l'exploration des configurations de composition. L'algorithme sous-jacent construisant cette liste est un parcours du graphe de composition en profondeur d'abord. Actuellement, cet outil suppose qu'il n'y a pas de cycle dans le graphe de composition. Or, dans la pratique, il peut y en avoir et le programmeur d'un composant fautif doit actuellement s'arranger pour les supprimer. La solution à terme est d'expliciter les dépendances durant l'initialisation des dépendances durant l'exécution, ce qui permet d'ordonnancer un graphe composant des cycles.

Cet outil calcule l'ordre de lancement des composants, voir figure 4.14. Le résultat se présente sous la forme d'une méthode, nommée par convention le nom du système concaténé avec `Probe`. Cette méthode appelle l'ensemble des méthodes d'initialisation des composants. Par convention et comme nous l'avons déjà vu, la méthode d'initialisation d'un composant `name` se nomme `name-Probe`. Si un composant ne possède pas de méthode d'initialisation, cet appel n'a pas lieu. Cette information est donnée par la balise `probe`. Cette méthode doit être appelée lors de l'initialisation du système, généralement dans la méthode `kernelstart` appelée à la fin de l'amorçage du domaine. Cette méthode est compilée et le résultat est lié à l'exécutable.



$$\left\{ \begin{array}{c} fichier_1.xml \\ \vdots \\ fichier_n.xml \end{array} \right\} \Longrightarrow \left\{ \begin{array}{c} composant_1 \\ \vdots \\ composant_m \end{array} \right\}_{m>=n} \Longrightarrow \left[\begin{array}{c} composant_1Probe \\ \vdots \\ composant_pProbe \end{array} \right]_{p<=m}$$

FIGURE 4.14 – Fonctionnement de l'ordonnanceur de l'initialisation

Le visualiseur

Le dernier outil permet de visualiser graphiquement le graphe de composition d'un système en mettant en évidence la notion de dépendance entre un composant client et un composant serveur, via une interface. Cet outil permet de mieux appréhender la complexité lorsque le nombre de composants dans un système augmente. L'outil permet d'afficher les composants et les dépendances sous forme d'interfaces fournies et d'interfaces exportées. C'est donc une aide très précieuse pour le programmeur d'un système, mais aussi pour toute personne désirant apprendre la structure d'un système. La figure 6.3 du chapitre 6 montre un affichage proposé par cet outil.

4.5 Mise en œuvre

Cette section montre la mise en œuvre de l'implantation de l'architecture THINK et permet de récapituler ce chapitre. Nous voyons ici comment sont générés les exécutables d'un système et les règles de programmation des composants et des usines à liaisons. Nous voyons aussi comment construire un système THINK.

4.5.1 Chaîne de génération

La chaîne de génération est illustrée à la figure 4.15. Elle peut être découpée en trois phases.

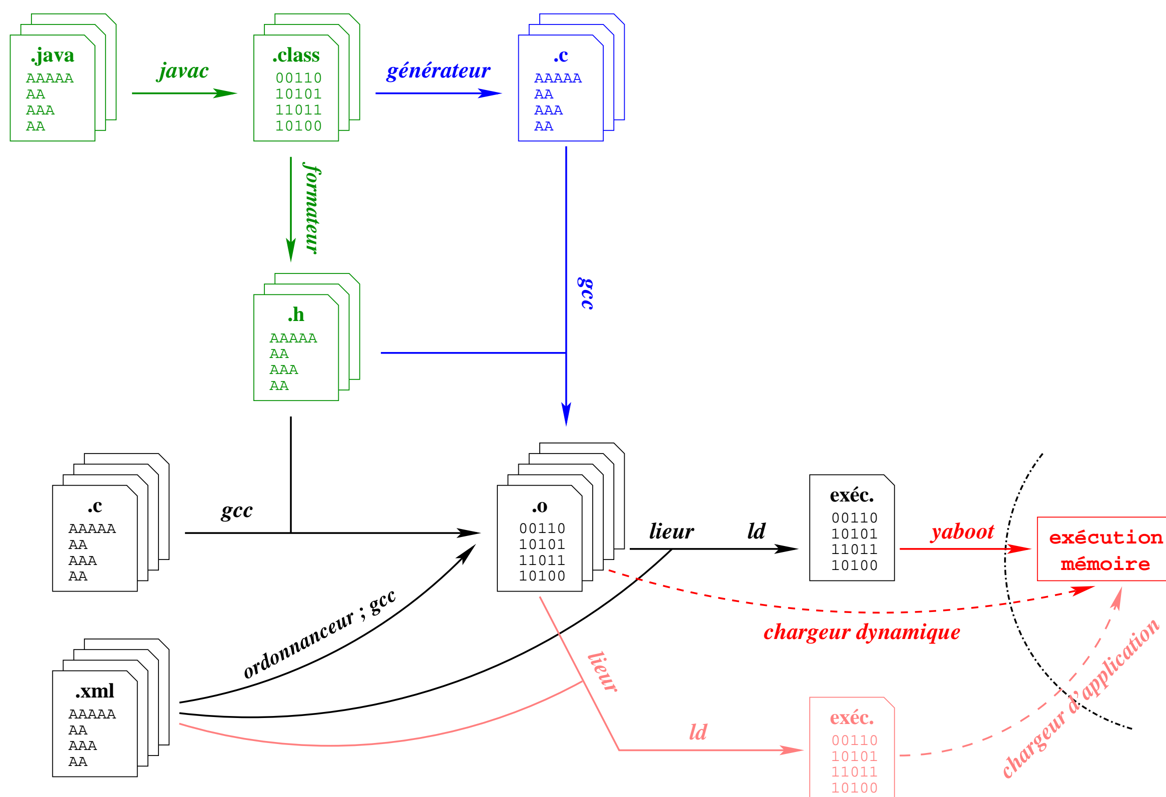


FIGURE 4.15 – Chaîne de génération

La première étape consiste à compiler les composants et les interfaces. Les interfaces écrites en Java sont compilées en « bytecode » via `javac`. Le formateur convertit alors ces interfaces au format binaire de représentation des interfaces dans le langage C. Les codes des composants sont compilés, via `gcc`, en binaire au format ELF. Chaque générateur génère pour toutes les interfaces les composants de liaisons de l'usine à liaison associée. Ces composants sont eux aussi compilés, via `gcc`.

La deuxième étape consiste à construire l'exécutable d'un système. L'ordonnanceur génère, pour un système donné, la méthode dont le code appelle de façon ordonnée les méthodes d'initialisation des composants, il demande ensuite la compilation de cette méthode en binaire. Le lieur, pour un

système donné, récupère les fichiers requis par la composition et demande le liage des binaires à ld qui produit l'exécutable du système au format ELF. Il faut noter ici que la même étape est utilisée pour construire une application qui peut être vue comme un système particulier s'exécutant dans un domaine applicatif.

La troisième étape consiste à exécuter le système. Un système est chargé en mémoire sur une machine d'exécution, via un utilitaire nommé ici yaboot, nous reviendrons sur cette utilitaire au chapitre suivant. Durant l'exécution, il est éventuellement possible de charger dynamiquement les binaires des composants n'ayant pas été liés statiquement à l'exécutable, via le chargeur dynamique. Celui-ci doit avoir été lié statiquement au système. Il est aussi éventuellement possible de lancer des applications, via un chargeur d'application. Celui-ci doit avoir été lié statiquement au système.

4.5.2 Règles de programmation des composants

Lorsqu'un concepteur souhaite fabriquer un composant, nommé par exemple name, il doit déclarer les éventuelles nouvelles interfaces en Java et programmer en C le code du composant. Il doit ensuite décrire son composant en XML. Ces différents fichiers sont utilisés dans la chaîne de génération et le code du composant pourra alors être intégré dans la composition du système lorsque ce dernier l'aura requis pour sa composition.

Si le composant requiert une méthode d'initialisation, il doit le déclarer dans le fichier XML et fournir une méthode nommée nameProbe. Cette méthode peut par exemple être utilisée pour exporter les interfaces du composant.

4.5.3 Règles de programmation des usines à liaisons

Lorsqu'un concepteur souhaite fabriquer une nouvelle usine à liaisons, nommée par exemple UL, il doit fournir un générateur pour les composants de liaison utilisés par l'usine. Il doit aussi programmer le code du composant de l'usine à liaison en suivant les règles de la section précédente. Ce code du composant et le générateur de l'usine à liaison sont alors intégrés dans la chaîne de génération.

Le code d'un composant de liaison, généré pour l'interface itf, est nommé par convention itf_UL. Ce composant doit fournir une méthode nommée par convention itf_UL. Cette méthode est appelée par le code du composant de l'usine à liaison. Les arguments passés dépendent de l'usine à liaison et doivent donc être définis par le programmeur.

4.5.4 Règles de programmation des systèmes

Lorsqu'un concepteur souhaite fabriquer un système, nommé par exemple OS, il décrit dans un fichier XML les composants requis par la composition du système. Il doit aussi fournir une méthode kernelstart réalisant l'initialisation du système. Cette méthode pourra éventuellement appeler la méthode OSProbe réalisant l'initialisation ordonnée de tous les composants requis. L'exécutable du système est alors construit selon la chaîne de génération.

Chapitre 5

Bibliothèque de composants KORTEX

Nous avons développé une bibliothèque de services systèmes que nous retrouvons communément dans les systèmes d'exploitation. Cette bibliothèque de composants est appelée KORTEX. Ces composants fournissent des abstractions permettant de gérer le processeur, la mémoire, les périphériques et le réseau. KORTEX fournit aussi un ensemble de liaisons et d'outils associés offrant différents types d'interaction. Différents exemples d'usage de cette bibliothèque sont donnés au chapitre suivant.

5.1 Prototype

La bibliothèque de composants KORTEX est réalisée sur POWERMACINTOSH, une machine basée sur un processeur RISC POWERPC. Les motivations pour le choix de cette architecture matérielle sont doubles. Tout d'abord, cela minimise la variété des périphériques rencontrés. Deuxièmement, l'architecture du processeur impose très peu de contraintes sur le modèle de programmation. Nous pouvons ainsi définir nos propres sémantiques, par exemple de fil d'exécution et de mémoire, et offrir un degré de flexibilité bien supérieur comparé à une architecture de type x86.

5.1.1 Philosophie

La conception de la bibliothèque respecte scrupuleusement le modèle d'architecture THINK et son implantation. Nous nous imposons aussi une certaine philosophie dans le but de maximiser la flexibilité de la bibliothèque. Cette philosophie repose sur les points suivants.

- *minimisation des abstractions.* Les abstractions offertes par les composants sont minimales, dans le but évident de ne pas faire de choix pénalisant les utilisations futures.
- *grain fin.* Un composant offre une seule abstraction. Dans la mesure du possible, les gros composants sont coupés en petits composants avec des interfaces clairement identifiées. Cela maximise les opportunités de remplacement facile d'un composant par un autre mieux adapté à l'utilisation.
- *exposition directe du matériel.* Dans la mesure du possible, les composants exposent directement les interfaces du matériel sans ajouter une sémantique non présente dans le matériel. Ainsi, les politiques de gestion ne sont pas présentes dans les composants. Elles sont ajoutées

en fonction de l'utilisation.

5.1.2 Caractéristiques

Les composants de la bibliothèque KORTEX sont développés en offrant différentes caractéristiques. Ces caractéristiques sont les suivantes.

- *portabilité.* Du fait, de l'utilisation d'interfaces clairement identifiées, les composants non spécifiques à une machine ou à un processeur sont portables sans modification. Seule une recompilation est nécessaire. Nous supposons ici que les composants accèdent aux ressources matérielles via des composants offrant des interfaces d'abstractions indépendantes de la machine sous-jacente. Le portage sur d'autre architecture de machine doit être relativement rapide.
- *transparence vis-à-vis du domaine.* Les composants peuvent s'exécuter indépendamment dans un domaine superviseur ou dans un domaine applicatif. Cette caractéristique permet à partir des mêmes composants de concevoir des noyaux monolithiques, des micronoyaux et même des exonoyaux. Cela est rendu possible par l'utilisation du concept de liaison flexible et du concept de domaine. La liaison occupe des traitements requis par les changements de domaines pouvant intervenir entre deux composants. Le domaine assure une structure d'accueil uniforme. Les concepteurs des composants et des systèmes n'ont plus à se soucier de ce problème.

  Cette caractéristique ne signifie pas que tous les composants peuvent s'exécuter dans les domaines applicatifs. En effet certains composants doivent s'exécuter dans le mode superviseur, c'est par exemple le cas pour les composants utilisant les instructions privilégiées du processeur.

5.1.3 Amorçage du système

Le chargement proprement dit de l'exécutable du système en mémoire est réalisé grâce à `BootX` ou à `yaboot`. Ensuite, le système débute son exécution et peut s'amorcer. Le rôle de l'amorçage est de mettre la machine dans un état connu que nous ne détaillons pas ici. Cet état permet l'initialisation des composants.

5.2 Composants POWERPC

KORTEX fournit deux composants POWERPC permettant d'une part de réifier les exceptions et d'autre part de contrôler l'unité de gestion de la mémoire (MMU). Les opérations supportées par ces composants sont purement fonctionnelles et ne modifient pas l'état du processeur, sauf sur demande explicite du système, par exemple lors du changement du masque des interruptions. KORTEX fournit aussi des composants implantant différents pilotes de périphériques et de contrôleurs. Ces composants sont décrits ci-dessous.

5.2.1 Exceptions

Le composant POWERPC pour les exceptions supporte une interface `Trap` donnée à la figure 5.1. Ce composant permet à un système de traiter les exceptions matérielles. L'objectif de cette interface

est de réifier efficacement les exceptions sans modifier leur sémantique. En particulier, sur le POWERPC, le traitement des exceptions débute en mode superviseur avec les interruptions masquées et la translation mémoire désactivée, interdisant les exceptions récursives.

```
interface Trap {
   void      Register(int id, Handler handler);
   void      Unregister(int id);
   void      SetContext(int phyctx, Context virtctx);
   Context   GetContext();
   void      Return();
  }
```

FIGURE 5.1 – Interface `Trap` pour les exceptions POWERPC

Lorsqu'une exception *id* survient, le processeur invoque une des méthodes internes du composant `Enter`$_{id}$. Conformément aux spécifications du POWERPC [Motorola Inc 1997a], un traitant d'exception doit être placé dans chaque entrée du vecteur d'interruption. La valeur *id* correspond à l'adresse d'une entrée dans le vecteur. Généralement, le vecteur commence à l'adresse mémoire physique 0.

`Enter`$_{id}$ commence par sauver les registres généraux, que nous appelons ici le *contexte d'exécution minimum* du processeur. Ces registres sont sauvés à une adresse mémoire préalablement spécifiée par le système grâce à la méthode `SetContext`. Cette méthode requiert en argument l'adresse physique *phys* et l'adresse virtuelle *virt* du contexte d'exécution, car ce composant accède à ces informations avec la translation mémoire activée ou désactivée. De plus, il ne connaît pas le modèle de mémoire mis en place par le système. Si la translation d'adresse n'est pas active, l'adresse virtuelle est supposée être la même que l'adresse physique.

`Enter`$_{id}$ installe ensuite une *pile de traitement unique* utilisée durant tout le traitement de l'exception. Cette pile est unique car le processeur masque les interruptions pendant le traitement d'une exception. Pour finir, `Enter`$_{id}$ appelle le traitant d'exception *handler* spécifique au système et préalablement enregistrée en utilisant la méthode `Register`. Le type `Handler` correspond à une référence sur un traitant d'exception. Le mécanisme mis en œuvre pour le traitement des exceptions est schématisé à la figure 5.2. À la fin du traitement, l'exécution peut soit revenir sur le fil d'exécution interrompu, soit revenir sur un autre fil d'exécution.

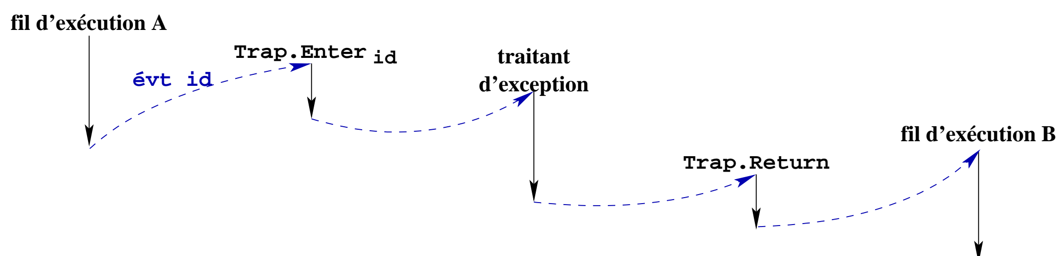


FIGURE 5.2 – Mécanisme mis en œuvre pour le traitement des exceptions

Quand le traitant termine, il appelle la méthode `Return`. Le rôle de cette méthode est de restaurer

le contexte d'exécution minimum. La méthode `Unregister` supprime le traitant d'exception associé à *id*. La méthode `GetContext` retourne l'adresse virtuelle du contexte d'exécution minimum. Cela permet par exemple de récupérer le contenu des registres de l'exécution interrompue. Lorsqu'on désigne un contexte d'exécution minimum par son adresse virtuelle (argument *virt* de `SetContext`, retour de `GetContext`) elle est typée, car le contexte peut être manipulé.

Bien que minimum, cette interface fournie suffisamment de fonctionnalité pour construire par exemple un ordonnanceur en utilisant la méthode `SetContext`. Nous reviendrons sur cela à la section 5.4. Comme nous pouvons le voir, ce composant est totalement indépendant du modèle de fil d'exécution implanté par le système.

5.2.2 Unité de gestion de la mémoire

Les algorithmes MMU mis en œuvres dans le POWERPC sont efficaces mais relativement complexes. Il fonctionne sur une table de pages inversée unique [Motorola Inc 1997a], la taille des pages du POWERPC est de 4 Ko. Le calcul des entrées dans cette table est basé selon un code de hache. Le composant POWERPC pour la gestion de la MMU implante la partie logicielle cet algorithme. Ce composant n'est pas requis pour un système qui ne nécessite pas de mémoire virtuelle, par exemple pour des raisons d'optimisation. L'interface exportée par ce composant est donnée à la figure 5.3.

```
interface MMU {
   void      SetPageTable(int virt, int phys, int size);
   void      AddMapping(int vsid, int virt, int phys,
                     int wimg, int pp);
   void      RemoveMapping(int vsid, int virt);
   PTE       GetMapping(int vsid, int virt);
   void      SetSegment(int vsid, int virtbase);
   void      SetBAT(int no, int virt, int phys,
                  int size, int wimg, int pp);
   void      RemoveBAT(int no);
}
```

FIGURE 5.3 – Interface du composant POWERPC pour la MMU

La méthode `SetPageTable` est utilisée pour spécifier la localisation en mémoire physique *phys* et en mémoire virtuelle *phys* du domaine superviseur de la table de page unique d'une taille *size*. Comme le POWERPC est un processeur segmenté, la méthode `SetSegment` positionne les 16 registres de segments de 0 à 15. Chaque segment, d'une taille de 256 Mo, est identifié de façon unique par une valeur. Ainsi, un espace d'adressage virtuel de 4 Go est entièrement décrit par les 16 registres. Cette méthode associe à l'adresse virtuelle *virtbase* un segment identifié par *vsid*. Le registre de segment est déterminé par les quatre bits de poids fort de *virtbase*.

Les méthodes `AddMapping`, `RemoveMapping` et `GetMapping` permettent respectivement de créer, supprimer et consulter une translation de page relative à un segment identifié par *vsid*. L'argument *virt* est l'adresse virtuelle de la page et l'argument *phys* est l'adresse physique. L'argument

wing[1234] spécifie les attributs de cache pour la page physique. L'argument *pp* renseigne les droits d'accès à la page, par exemple uniquement en mode superviseur. Le type `PTE` retourné par `GetMapping` correspond à l'entrée de la translation dans la table de page. Ce type est spécifié par le POWERPC.

Les méthodes `SetBAT` et `RemoveBAT` permettent respectivement de programmer et d'effacer les quatre registres BAT « Block Address Translation ». Ces registres offrent un moyen commode de créer des translations arbitraires de blocs mémoires sans nécessiter de mettre en place une table de page. Modulo la taille *size* qui spécifie la taille du bloc, les arguments de `SetBAT` sont les mêmes que `AddMapping`.

5.2.3 Pilotes de périphériques

La bibliothèque KORTEX fournit des composants implantant des pilotes de périphériques. Ces composants permettent de contrôler la majorité des périphériques qui apparaissent dans les POWERMACINTOSH. Cela inclut les bus PCI, l'accès au logiciel de pilotage du matériel « firmware », les contrôleurs d'interruption PIC, les cartes réseaux Ethernet et les cartes graphiques via le « frame buffer ». Il y a une interface spécifique par type de périphérique. Chaque interface est aussi proche que possible des ressources matérielles. Le problème de cette approche est de masquer les spécificités des périphériques. Un utilisateur qui souhaite accéder à ces dernières pourra toujours développer son propre pilote. Nous présentons à la section 5.5.4 l'interface offerte par les pilotes de cartes réseaux. La liste des pilotes est donnée à la section 5.8

5.3 Composants mémoire

KORTEX fournit des composants de gestion de la mémoire implantant différents modèles de mémoire ; la mémoire paginée et la mémoire plate. Les composants offrant la mémoire paginée peuvent être utilisés par les systèmes nécessitant plusieurs espaces d'adressage, par exemple pour offrir le concept de processus. Les composants offrant la mémoire plate peuvent être utilisés par les systèmes nécessitant uniquement un espace d'adressage noyau, comme les systèmes dédiés à une application. KORTEX fournit aussi des composants offrant l'allocateur dynamique de mémoire. Ces différentes implantations sont expliquées ci-dessous.

5.3.1 Modèle de mémoire paginée

Plusieurs composants sont nécessaires pour implanter le modèle de mémoire paginée. Ces composants peuvent bien évidemment être utilisés individuellement pour par exemple implanter d'autres modèles de mémoire.

La pagination requiert tout d'abord un composant permettant d'allouer individuellement des pages physiques. Ces pages pourront alors être couplées dans les différents espaces d'adressage virtuels. KORTEX fournit un composant implantant un « buddy system » [Knuth 1973] permettant l'allocation des pages physiques. L'interface `Page` fournie par ce composant est donnée à la figure 5.4.

[1] "W" pour « Write-Through »
[2] "I" pour « caching-Inhibited »
[3] "M" pour « Memory Coherency »
[4] "G" pour « Guarded Memory »

La méthode `alloc` permet l'allocation d'un nombre 2^{order} de pages physiques consécutives, l'entier retourné est l'adresse de la première page physique. La méthode `free` permet la libération des pages. La méthode `size` est utilisée pour connaître la taille de la mémoire physique.

```
interface Page extends ResourceFactory{
   int      alloc(int order);
   void     free(int page, int order);
   int      size();
}
```

FIGURE 5.4 – Interface du « buddy system »

L'interface `Space` donnée à la figure 5.5 représente l'abstraction d'un espace d'adressage. Cette interface offre toutes les méthodes permettant de gérer les translations d'adresse. C'est une ressource abstraite et elle étend donc l'interface `AbstractResource` du modèle de ressources. La méthode `map` couple la page virtuelle *virt* avec la page physique *phys*, `unmap` est l'opération duale. La méthode `tophys` retourne l'adresse physique correspondant à l'adresse virtuelle *virt*. La méthode `ioremap` couple dans l'espace d'adressage virtuel une adresse physique spécifique, `iounmap` est l'opération duale. Ces deux méthodes sont utilisées pour accéder depuis un espace virtuel aux registres des différents périphériques. La méthode `ioalloc` alloue des pages mémoires en vue de les utiliser avec les différents périphériques, `iofree` est l'opération de libération. Cette méthode est nécessaire car certains périphériques ont des contraintes d'alignement sur les données qu'ils accèdent.

La méthode `getid` retourne un identifiant qui identifie de façon unique l'espace d'adressage. Cette valeur est utilisée pour construire les seize segments qui seront identifiés par de $getid << 4$ à $getid << 4 + 16$. La méthode `setspace` est utilisée pour activer un espace d'adressage, elle est utilisée durant les changements de contexte.

Cette interface est implantée par le composant `pagetable`. Durant son initialisation, ce composant met en place une table de page et l'espace d'adressage virtuel du domaine superviseur dont l'identifiant est 0. Ce composant est implanté en utilisant l'interface `MMU`, voir la section 5.2.2, mais fournit une abstraction de plus haut niveau comprenant la gestion des registres de segments. Pour des raisons d'efficacité, 1 Go de l'espace de 4 Go est réservé pour le domaine superviseur et est partagé par tous les autres domaines. Néanmoins, cet espace n'est accessible qu'en mode superviseur.

L'interface `SpaceFactory` est utilisée lors de la création de nouveaux espaces d'adressage pour les domaines applicatifs. Cette interface est une usine et étend donc `ResourceFactory`. La seule méthode est `new` et elle crée un nouvel espace d'adressage de type `Space`. Cette interface est implantée par le composant `spacefactory`.

Modèle de mémoire plate

Le modèle de mémoire plate est plus simple à mettre en œuvre et nécessite un seul composant. Ce modèle peut être utilisé lorsque le système requiert uniquement un seul espace d'adressage, à savoir le domaine superviseur. Ce composant, nommé `flat`, implante la même interface `Space` que pour le modèle de mémoire paginée. L'implantation de ce composant est donc essentiellement vide et les adresses virtuelles sont égales aux adresses physiques. Son unique rôle est de procurer

```
interface Space extends AbstractResource {
   void      map(int virtpage, int physpage,
                int wimg, int pp);
   void      unmap(int virtpage);
   int       tophys(int virtpage);

   char[]    ioremap(int physaddr, int size, int wimg);
   void      iounmap(char[] virtaddr, int size);

   char[]    ioalloc(int size);
   void      iofree(char[] virtaddr, int size);

   int       getid();
   void      setspace();
}
```

FIGURE 5.5 – Interface Space

```
interface SpaceFactory extends ResourceFactory {
   Space     newSpace();
}
```

FIGURE 5.6 – Interface SpaceFactory

un accès transparent aux espaces d'adressage pour tous les autres composants, comme les pilotes de périphérique qui requièrent des possibilités de couplage de pages pour accéder aux registres matériels. Cela permet de faire fonctionner le même composant indépendamment sur la mémoire plate ou la mémoire paginée.

5.3.2 Allocateur dynamique de mémoire

Un allocateur dynamique de mémoire offre classiquement les opérations d'allocation et de libération de zone mémoire. Un allocateur implante l'interface `Allocator` donnée à la figure 5.7. La méthode `alloc` permet d'allouer de la mémoire et la méthode `free` permet de la libérer.

```
interface Allocator extends ResourceFactory {
   char[]    alloc(int size);
   void      free(char[] addr);
}
```

FIGURE 5.7 – Interface d'un allocateur dynamique de mémoire

Cette interface est actuellement implantée par un composant fournissant l'implantation de l'allocateur de mémoire standard de GNU. Cet algorithme est du type « best-fit » [Knuth 1973] et permet de minimiser la fragmentation. Cet allocateur fonctionne indépendamment du modèle de mémoire sous-jacent grâce à un composant intermédiaire réalisant l'équivalent de l'appel système UNIX `sbrk`. Un tel composant implante l'interface `Sbrk` donnée à la figure 5.8. La méthode `alloc` permet d'accroître la taille du segment de données du domaine. Cet intermédiaire est nécessaire car le segment de données ne doit pas être discontinu.

```
interface Sbrk {
   char[]   alloc(int size);
}
```

FIGURE 5.8 – Interface `Sbrk`

Cette interface est implantée par deux composants. Le premier fonctionne sur le modèle de mémoire plate. Son fonctionnement est trivial et consiste uniquement à incrémenter un pointeur de fin de segment. Le deuxième fonctionne sur le modèle de mémoire paginée. Son fonctionnement consiste à allouer des pages physiques et à créer les translations associées quand le segment croît. Outre le fait de pouvoir être utilisé dans le domaine superviseur, le deuxième composant peut aussi être utilisé dans les domaines applicatifs. Ainsi, l'allocateur est lui aussi utilisable quel que soit le domaine.

5.4 Composants d'exécution

La bibliothèque KORTEX fournit des composants permettant la virtualisation et la gestion le processeur. Ces composants peuvent être utilisés par les systèmes nécessitant l'exécution de plusieurs fils d'exécution. De nombreux autres composants requièrent la présence de fils d'exécution. C'est par exemple le cas pour la pile de protocole que nous verrons à la section 5.5. Ces composants sont implantés selon le modèle de ressources vu à la section 3.2.4 qui lui-même est issu de ReTINA [ReTINA 1999].

5.4.1 Fils d'exécution & ordonnancement

Les composants d'ordonnancement permettent les opérations usuelles sur les fils d'exécutions ; création, destruction, mais aussi la possibilité d'attendre sur une condition et d'en être informé. Les fils d'exécution ne sont que des registres du processeur et ne fournissent pas par exemple de pile d'exécution, c'est à l'utilisateur de la définir si elle est requise par son langage de programmation. Si les fils d'exécution ne s'exécutent pas dans le même espace d'adressage, l'ordonnanceur effectue aussi les opérations requises par les changements d'espaces d'adressage.

Fils d'exécution

L'interface `Thread` donnée à la figure 5.9 représente l'abstraction d'un fil d'exécution. Cette interface offre deux méthodes qui permettent de manipuler un fil d'exécution. Tout d'abord, la méthode `run` lance l'exécution du fil d'exécution. Elle prend en argument le contexte minimum du processeur

initial du fil d'exécution. Ce contexte correspond à celui vu à la section 5.1. Ce contexte est initialisé avec la valeur initiale des registres, par exemple le pointeur d'instruction contient l'adresse de la méthode ou débute le fil d'exécution, et les registres *r3* à *r13* contiennent les arguments. Deuxièmement, la méthode `ctx` récupère durant l'exécution le contexte associé au fil d'exécution.

```
interface Thread extends AbstractResource {
   void     run(Context ctx);
   Context  ctx();
}
```

FIGURE 5.9 – Interface d'un fil d'exécution

L'implantation de cette interface dépend de l'ordonnanceur. Il n'y a pas de code de composant autonome pour les fils d'exécution. Le code de composant des fils d'exécution est intégré aux ordonnanceurs qui sont présentés à la section ci-dessous.

Ordonnanceurs

L'interface `Scheduler` donnée à la figure 5.10 est utilisée pour l'ordonnanceur qui réalise la gestion du processeur. Elle crée des `Thread`, c'est donc une usine qui étend `ResourceFactory`. La méthode `newThread` est utilisée pour créer de nouveaux fils d'exécution. Elle prend en argument l'espace d'adressage dans lequel va s'exécuter le nouveau fil d'exécution. Elle prend aussi la priorité du fil d'exécution. En fonction de l'implantation, cet argument peut ne pas être utilisé. La méthode `destroyThread` permet la destruction du fil d'exécution donné en paramètre. Les méthodes `getThread` et `getSpace` permettent respectivement d'obtenir le fil d'exécution et l'espace d'adressage courant.

Le reste des méthodes de l'interface `Scheduler` réalise la synchronisation des fils d'exécution. La méthode `yield` permet au fil d'exécution courant de libérer le processeur et déclencher sa réallocation à un autre fil d'exécution. La méthode `wait` endort le fil d'exécution sur le verrou *lock*. Le fil d'exécution sera réveillé lors d'un appel par un autre fil d'exécution de la méthode `notify` sur le même verrou *lock*. La méthode `waitto` est similaire à la précédente, mais l'attente est limitée à un délai en milliseconde fixée par *timeout*.

Actuellement, un ordonnanceur coopératif et deux ordonnanceurs préemptifs sont implantés dans KORTEX. Le composant nommé `cooperative` implantant l'ordonnanceur coopératif est basique et utilise le co-routinage[5]. Un changement de contexte est effectué uniquement lors d'un appel à `yield` par le composant actif. Les composants implantant des ordonnanceurs préemptifs allouent le processeur aux fils d'exécution selon un quantum de temps fixé à 100 Hz. Pour un composant nommé `roundrobin`, l'allocation est circulaire, pour l'autre nommé `priority` elle est basée sur la priorité des fils d'exécution. Pour de plus amples informations sur les politiques d'allocations du processeur, on consultera [Silberschatz et Galvin 1998].

[5]Le co-routinage consiste à utiliser les procédures `setjmp` et `longjmp` offertes par la librairie C. L'ordonnancement applicatif utilisant le co-routinage offre théoriquement de meilleures performances pour la gestion des fils d'exécution que pour un ordonnancement système. Mais cette implantation pose deux problèmes ; cela n'exploite pas les architectures multiprocesseurs, lorsqu'un fil d'exécution appelle le noyau il bloque tout le processus.

```
interface Scheduler extends ResourceFactory {
   Thread   newThread(Space space, int prio);
   void     destroyThread(Thread thread);
   Thread   getThread();
   Space    getSpace();

   void     yield();
   void     wait(Lock lock);
   int      waitto(Lock lock, int timeout);
   void     notify(Lock lock);
}
```

FIGURE 5.10 – Interface d'un ordonnanceur

Les ordonnanceurs sont implantés en utilisant l'interface `Trap` des exceptions du POWERPC, voir la section 5.2.1. Les ordonnanceurs préemptifs sont implantés simplement en installant un traitant d'exception temporelle, en fait un decrémenteur sur le POWERPC. À chaque exception temporelle, l'utilisation de la méthode `SetContext` met en place le contexte minimum du fil d'exécution à activer qui sera restauré à la fin du traitant.

5.4.2 Processus

Le concept de processus identifie toutes les ressources utilisées par une application ; espaces d'adressage, fils d'exécution, les fichiers ouverts, etc. Actuellement, KORTEX n'implante pas de composant offrant le concept de processus. Seule la notion de fils d'exécution s'exécutant dans un espace d'adressage existe.

Chargeur d'application

Le seul composant relatif à la gestion des processus offert par KORTEX est un chargeur d'application. Ce composant lance une nouvelle application à partir d'un fichier contenant un exécutable au format ELF [TIS 1993]. Ce composant requiert la mémoire paginée pour pouvoir mettre en place l'espace d'adressage de l'application. Ce composant requiert aussi un ordonnanceur permettant de lancer le fil d'exécution initial de l'application.

5.5 Composants réseaux

KORTEX fournit un ensemble de composant pour construire des piles de protocoles par l'assemblage de composants implantant des protocoles. Actuellement, les protocoles implantés sont les suivants ; Ethernet, ARP, IP[6], UDP, TCP[7], NFS et les SunRPC. On consultera les livres de Stevens

[6] Actuellement, IP ne gère pas la fragmentation.
[7] Actuellement, TCP supporte uniquement un contrôle de flux minimum.

pour une description des protocoles réseaux [Stevens 1994]. Tous ces composants sont architecturés selon le modèle x-Kernel [Hutchinson et Peterson 1991].

5.5.1 Usine à paquets

L'interface `packet`, donnée à la figure 5.11, représente un paquet. Cette interface offre deux méthodes `getphysaddr` et `getvirtaddr` permettant respectivement d'obtenir l'adresse physique et l'adresse virtuelle des données d'un paquet. L'adresse physique est utilisée par les pilotes de périphériques.

```
interface packet extends AbstractResource {
   int       getphysaddr();
   char[]    getvirtaddr();
}
```

FIGURE 5.11 – Interface d'un paquet

L'interface `packetfactory`, donnée à la figure 5.12, est une usine à paquets. La méthode `alloc` permet l'allocation d'un paquet de taille *size*. La méthode `free` permet sa suppression.

```
interface packetfactory extends ResourceFactory {
   packet    alloc(short size);
   void      free(packet buf);
}
```

FIGURE 5.12 – Interface d'une usine à paquets

5.5.2 Modèle de pile de protocoles

Le modèle x-Kernel propose un canevas pour la composition de protocoles. Cette architecture est suffisamment générale pour fonctionner avec des protocoles réseaux très variés et cela sans perte significative de performance. x-Kernel repose essentiellement sur trois caractéristiques. Premièrement, la définition d'une interface uniforme pour tous les protocoles permet de les remplacer, sous réserve qu'ils aient la même sémantique, par exemple de contrôle de flux. Deuxièmement, la composition entre les différents protocoles est effectuée dynamiquement. Troisièmement, le passage de message d'un protocole à un autre ne nécessite, ni de changement de contexte grâce à l'utilisation d'un fil d'exécution par message, ni de recopie de message grâce à une représentation sous forme d'arbre des bouts de message.

Le modèle x-Kernel peut directement être instancié dans l'architecture THINK grâce au concept des liaisons flexibles. Ce modèle est composé de protocoles et de sessions, voir figure 5.13. Un protocole est une abstraction d'un protocole réseau, comme TCP. Une session représente un canal de communication particulier avec un site distant. Une session est dynamiquement créée par un protocole et envoie et reçoit des messages au format du protocole considéré à un site distant. Un protocole

peut donc être vu comme une usine `ResourceFactory` et une session comme une ressource abstraite `AbstractResource`. Un protocole exporte deux interfaces : une interface basse, qui est utilisée pour démultiplexer les messages arrivant et pour gérer les demandes distantes de connexion et de déconnexion ; une interface haute, qui est utilisée pour créer de nouvelles sessions. Une session exporte deux interfaces : une interface haute qui envoie des messages à un site distant ; une interface basse qui traite les messages arrivant ayant préalablement été démultiplexées par le protocole.

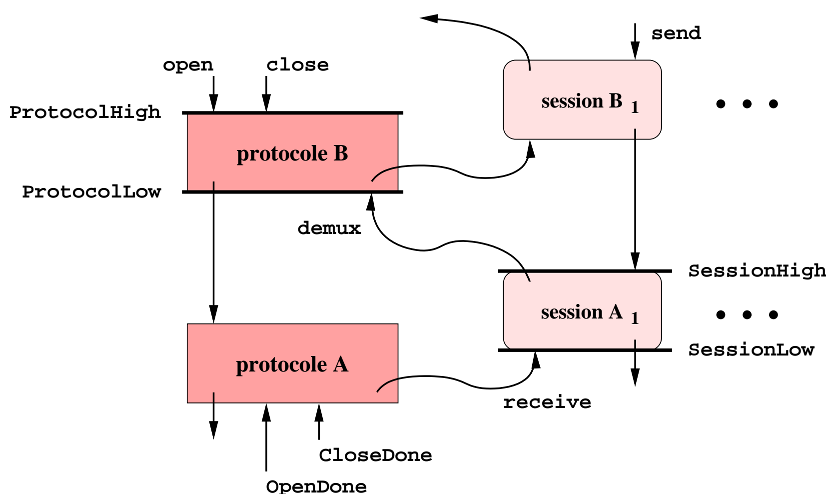


FIGURE 5.13 – Modèle x-Kernel de pile de protocole

Modèle d'exécution

Le modèle d'exécution proposé consiste à utiliser un fil d'exécution par message plutôt qu'un fil d'exécution par protocole. L'inconvénient de cette solution est la nécessité de synchroniser les fils d'exécution si la pile peut traiter plusieurs paquets simultanément. L'intérêt est la vitesse d'exécution qui ne nécessite pas de changement de contexte lors du passage d'un message d'un protocole à un autre.

Lors de la réception d'un paquet, le système doit mettre en place la politique de traitement du paquet. Il peut décider de traiter le paquet sous contexte d'interruption. Sinon, il peut activer un fil d'exécution qui se charge de traverser les différents étages de la pile de protocole. Le nombre de fils d'exécution potentiellement actifs est du ressort du système. Bien évidemment, s'il emploie plusieurs fils d'exécution simultanément actifs, il doit mettre en place un politique de synchronisation, soit aux extrémités de la pile, soit entre chaque protocole.

Protocoles

L'interface `ProtocolHigh`, donnée à la figure 5.14, correspond à l'interface haute d'un protocole. La méthode `open` ouvre une connexion avec un site identifié par *remotehost*. Le format de l'adresse dépend du protocole. Cette méthode ouvre potentiellement une connexion avec le protocole inférieur et renvoie une nouvelle session. L'argument *hlp* correspond à l'interface basse du protocole supérieur réalisant l'appel, *key* est la clé de démultiplexage, par exemple 6 pour TCP et 17 pour UDP.

La méthode `openEnable` signifie au protocole que le protocole supérieur qui effectue l'appel *hlp* identifié par la clé *key* est en attente d'une connexion. Le protocole supérieur est averti d'une demande de connexion grâce à la méthode `openDone` de l'interface `ProtocolLow`, nous allons y revenir. La méthode `close` ferme la connexion réalisée par la session *session*.

```
interface ProtocolHigh extends ResourceFactory {
   SessionHigh open(char[] remotehost,
                    ProtocolLow hlp, int key);
   int       openEnable(ProtocolLow hlp, int key);
   void      close(SessionHigh session);
}
```

FIGURE 5.14 – Interface haute d'un protocole

L'interface `ProtocolLow`, donnée à la figure 5.15, correspond à l'interface basse d'un protocole. La méthode `demux` est utilisée pour informer le protocole qu'un paquet lui étant destiné est arrivé sur la session inférieure *lls*. L'argument *msg* est le paquet de taille *size*. L'argument *reserve* est la taille des en-têtes décodés successivement par les sessions inférieures. Ainsi, l'en-tête de ce protocole commence au début du paquet décalé de *reserve*. Cette méthode démultiplexe le message et l'envoie à la bonne session correspondant à la clé donnée à l'ouverture de la connexion. La méthode `openDone` informe ce protocole d'une demande de connexion. L'argument *session* correspond à la session inférieure créée. Cette méthode créée potentiellement une nouvelle session et elle informe potentiellement le protocole supérieur. La méthode `openDone` informe le protocole que le site distant a fermé une connexion réalisée par la session inférieure *session*.

```
interface ProtocolLow {
   int       demux(SessionHigh lls,
                   packet msg, int reserve, int size);
   int       openDone(SessionHigh lls);
   void      closeDone(SessionHigh lls);
}
```

FIGURE 5.15 – Interface basse d'un protocole

Sessions

L'interface `SessionHigh`, donnée à la figure 5.16, correspond à l'interface haute d'une session. La méthode `send` envoie un paquet *msg* de taille *size*. L'argument *reserve* est la taille réservée pour les en-têtes des protocoles inférieurs. L'argument *options* passe des options. Les méthodes `addresssrc` et `addressdst` donnent l'adresse locale et l'adresse distante de la connexion. Le format de l'adresse retournée dépend du protocole. La méthode `mtu` retourne la taille maximum d'un paquet pouvant être envoyé par cette session. Cette taille est par exemple utilisée par TCP pour découper au plus tôt les paquets et éviter les recopies de messages. La méthode `maxHeaderSize`

retourne la somme maximum des en-têtes des protocoles inférieurs. Cette taille correspond à l'argument *reserve* de la méthode `send`. Elle évite les recopies en écrivant à la bonne place chaque en-tête dans le paquet.

```
interface SessionHigh extends AbstractResource {
   char[]   addresssrc();
   char[]   addressdst();
   int      send(packet msg, int reserve, int size,
                char[] options);
   int      mtu();
   int      maxHeaderSize();
}
```

FIGURE 5.16 – Interface haute d'une session

L'interface `SessionLow`, donnée à la figure 5.17, correspond à l'interface basse d'une session. La méthode `receive` est appelée lorsque le protocole de la session a démultipléxé le paquet *msg*. Les arguments *reserve* et *size* sont les mêmes que pour la méthode `demux`. Cette méthode est utilisée pour traiter le paquet et réaliser les éventuels acquittements et contrôles de flux. Cette méthode appelle ensuite le protocole supérieur pour qu'il démultiplexe à son tour le paquet.

```
interface SessionLow {
   int      receive(packet msg, int reserve, int size);
}
```

FIGURE 5.17 – Interface basse d'une session

5.5.3 Interfaces hautes, les applications

Les interfaces hautes sont les interfaces de plus hauts niveaux que l'on retrouve au sommet de la pile de protocoles. Elles peuvent être de type protocole `ProtocolHigh` et `SessionHigh`. Elles peuvent aussi être spécifiques à une utilisation donnée.

Sockets

Les protocoles UDP et TCP offrent une interface haute identique de type socket. Cette interface offre les méthodes standards de flux de données des sockets à savoir ; `socket`, `read`, `write`, `accept`, `close`, etc.

Systèmes de fichiers virtuels

Le protocole NFS offre une interface haute de type système de fichiers VFS permettant de manipuler les fichiers et les répertoires. Cette interface n'est pas spécifique à NFS. Elle peut être utilisée

avec tous les systèmes de fichiers. Elle offre les méthodes ; `open`, `readdir`, `read`, `write`, etc.

5.5.4 Interfaces basses, les cartes réseaux

L'interface `net` fournie par les composants implantant les pilotes de cartes réseaux est donnée à la figure 5.18. Les méthodes `start` et `stop` permettent respectivement de démarrer et d'arrêter la carte réseau. L'argument *protocole* est expliqué au paragraphe suivant. La méthode `transmit` envoie le paquet *msg* de taille *size* sur le réseau. L'argument réserve à la même signification que précédemment. La méthode `set_promiscious` positionne le mode de réception. La méthode `get_mac` donne l'adresse Ethernet de la carte.

```
interface net {
   int      start(netif protocol);
   int      stop();
   int      transmit(packet msg, int reserve, int size);
   void     set_promiscious(boolean promiscious);
   char[]   get_mac();
}
```

FIGURE 5.18 – Interface `net`

L'interface `netif` est utilisée pour informer la pile de protocole de l'arrivée d'un paquet sur cette carte. Quand cet événement survient, la méthode `rx` est appelée. L'argument *msg* est le nouveau paquet de taille *size*. L'argument a toujours la même signification. C'est en définissant un composant spécialisé implantant cette interface que le système met en place le modèle d'exécution de la pile de protocole.

```
interface netif {
   int      rx(packet msg, int reserve, int size);
}
```

FIGURE 5.19 – Interface `netif`

5.6 Liaisons

La bibliothèque de composants KORTEX fournit différents types de liaisons permettant les interactions entre deux composants situés dans des domaines différents. Chaque type de liaison possède sa propre usine à liaisons qui met en place le ou les composants de liaisons requis pas la liaison. Pour chaque type de liaison, le ou les composants de liaisons sont générés par un générateur propre à ce type.

Pour nous aider à comprendre le fonctionnement des composants de liaison générés, nous utilisons l'interface `Null`, donnée à la figure 5.20 définissant une seule méthode `call` avec un seul argument

arg et retournant un entier.

```
interface Null {
   int    call(int arg);
}
```

FIGURE 5.20 – Interface `Null`

5.6.1 Liaison appel système

Cette liaison, appelée liaison appel système, permet aux applications s'exécutant dans leur propre espace d'adressage d'appeler des composants s'exécutant dans le domaine superviseur. Cette liaison est nécessaire uniquement si le système utilise l'isolation par protection mémoire matérielle des applications. Cette liaison implante tous les mécanismes requis pour effectuer des appels système. Elle repose sur un composant de liaison qui s'exécute dans l'application et sur un traitant d'appel système, voir la figure 5.21. Comme un appel système est similaire à une interruption, l'exécution du composant appelé utilise la pile de traitement unique du domaine noyau.

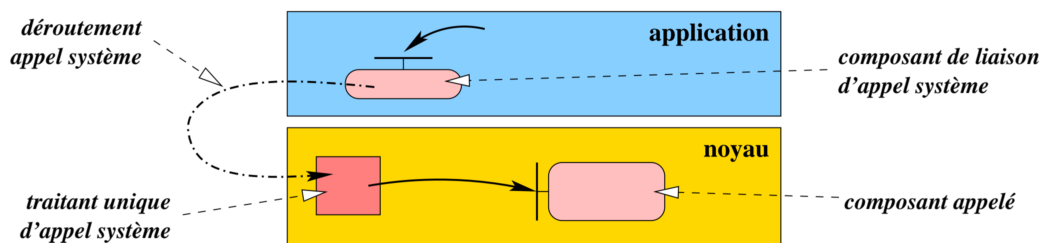


FIGURE 5.21 – Fonctionnement de la liaison appel système

Composants de liaisons

Le rôle du composant de la liaison appel système est d'intercepter les appels, émis par les applications, à un composant situé dans le noyau. Il y a autant d'instances du composant de liaison qu'il y a d'applications accédant au composant appelé. Les arguments de l'appel système sont passés via les registres *r4* à *r10* et donc seuls les types primitifs sont acceptés par cette liaison. Comme sur le POWERPC les arguments sont passés par les registres, ils contiennent déjà les arguments que l'appelant aura complété. Ainsi, le seul travail effectué par le composant de liaison est de remplacer la valeur du registre *r3* contenant *this* par la référence noyau de l'interface du composant appelé. Cette information est contenue dans les variables d'instance du composant de liaison. Il positionne ensuite le registre *r0* au numéro de la méthode. Puis il effectue l'appel système proprement dit par l'instruction POWERPC `sc`, voir la figure 5.22 qui donne le code pour la méthode `call` de l'interface `Null`.

Ce composant de liaison est généré grâce à un générateur particulier construit pour ce type de liaison. Le générateur génère du code assembleur pour d'une part accéder aux instructions processeur et d'autre part optimiser les traitements. Ce composant est donc spécifique au POWERPC et n'est donc

```
static int call(NullItf* this, int arg1) {
  struct syscalldata* syscall = (struct syscalldata*)this;
  register unsigned long __sc_0 __asm__ ("r0");
  register unsigned long __sc_3 __asm__ ("r3");
  register unsigned long __sc_4 __asm__ ("r4");
  __sc_0 = 0*4;
  __sc_3 = (unsigned long)syscall->id;
  __sc_4 = (unsigned long)arg1;
  __asm__ __volatile__
    ("crclr␣0\n\t"
     "sc"
    : "=&r" (__sc_3)
    : "r" (__sc_0),
      "r" (__sc_3),
      "r" (__sc_4));
}
```

FIGURE 5.22 – Composant de liaison pour la liaison appel système

pas portable. Le nom du composant de liaison est, par convention, le nom de l'interface concaténé avec "`_syscall`". Actuellement, cette liaison n'est pas sécurisée car il est possible de construire, depuis l'application, des références noyau et de les invoquer.

Traitant système

L'appel système est traité par le traitant du noyau. Celui-ci est unique dans le système. Il est mis en place à l'initialisation de l'usine à liaisons dans sa méthode `SCserverProbe` en appelant la méthode `Register` de l'interface `Trap`, voir section 5.2.1. En fait, ce traitant est un démultiplexeur dont le rôle est d'invoquer la bonne méthode de l'interface du composant appelé. La référence de l'interface est contenue dans le registre *r3* positionné par le composant de liaison, le numéro de la méthode appelée est contenu dans *r0*. Ces deux informations sont utilisées pour rechercher l'adresse de la méthode dans le descripteur de l'interface. Le code du traitant des appels système est donné à la figure 5.23 pour montrer la simplicité du mécanisme de démultiplexage. À la fin de l'exécution du composant appelé, le traitant se termine et l'instruction de fin d'interruption du POWERPC `rfi` « return from interrupt » est exécutée.

Une optimisation possible de la liaison appel système est réalisable en exploitant les spécifications des conventions d'appels « Application Binary Interface » (ABI) de [System V 1995]. Il est dit que les registres *r1* et *r14* à *r31* sont non volatiles entre les appels de méthodes et il n'est donc pas nécessaire de les sauver lors d'un appel système. Les autres registres *r0* et *r3* à *r13* sont perdus durant les appels de méthodes et il n'est donc pas non plus nécessaire de les sauver. Cette optimisation suppose manifestement que les conventions ABI sont respectées pour les appels de méthodes. Tous les compilateurs standards respectent ces conventions. Cette optimisation sauve 70 cycles d'horloge par appel système. Néanmoins, cette modification doit être faite dans le composant POWERPC `trap` qui est responsable de la sauvegarde et de la restauration des registres lors d'une exception.

```
syscall:
 mfspr r2, SPRN_SPRG3
 lwz   r10, 0(r3)     // recherche et appel de la
 lwz   r0, GPR0(r2)   // methode de l'interface␣du
 lwzx  r10, r10, r0   // composant appele
 mtlr  r10
 blrl
 b     TrapReturnSC   // retour a l'application
```

FIGURE 5.23 – Traitant noyau des appels système

5.6.2 Liaison remontée système

Cette liaison, appelée une remontée système, permet au noyau d'interagir avec les applications s'exécutant dans leurs propres espaces d'adressage. D'un point de vue conceptuel, c'est l'inverse de la liaison appel système. Elle est utilisée pour propager les événements aux applications. Cette liaison est utile uniquement si le système utilise l'isolation par protection mémoire matérielle des applications. Comme pour la liaison appel système, la liaison remontée système utilise un composant de liaison qui s'exécute dans le noyau. Chaque application implante un traitant de remontée système qui sera exécuté par le noyau pour simuler le déclenchement d'une exception dans l'application, voir la figure 5.24.

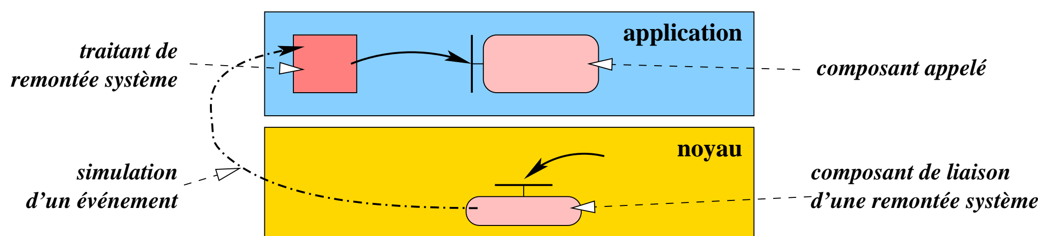


FIGURE 5.24 – Fonctionnement de la liaison remontée système

Composant de liaison

Le rôle de ce composant de liaison est d'intercepter les appels du noyau à un composant situé dans une application. Le fonctionnement de ce composant est similaire au fonctionnement du composant de liaison appel système. Il positionne les mêmes registres et `r3` contient la référence du composant appelé dans l'espace d'adressage de l'application. La grande différence vient de la remontée système. En effet, il n'y a pas d'instruction pour cela. Il est donc nécessaire de simuler cette remontée système. La remontée système doit d'une part, activer l'espace d'adressage, et d'autre part déclencher l'exécution du traitant applicatif de la remontée système. Ce déclenchement est réalisé en simulant un retour d'interruption `rfi`, voir la figure 5.24. Au retour de l'application, le composant réactive l'espace d'adressage interrompu.

```
static int call(NullItf* this, int arg1) {
  register struct upcalldata* self __asm__ ("r31") =
    (struct upcalldata*)this;
  CALL0(self->space, setspace);
  {
    register SpaceItf* prev __asm__ ("r29") =
      CALL0(self->sched, getSpace);
    register unsigned int ret __asm__ ("r30");
    register unsigned int r3 __asm__ ("r3") = self->target;
    __asm__ __volatile__
      ("lis␣30,next0@ha\n\t"
       "mr␣31,1\n\t"
       "li␣0,%1\n\t"
       "mtspr␣%3,%4\n\t"
       "mtspr␣%5,%6\n\t"
       "la␣30,next0@l(30)\n\t"
       "rfi\n\t"
       "next0:␣mr␣%0,3"
       : "=r" (ret)
       : "g" (0*4), "r" (r3), "g" (SPRN_SRR1),
         "r" (MSR_KERNEL | MSR_PR), "g" (SPRN_SRR0),
         "r" (self->upcall)
       : "r30", "r31", "r0", "r3");
    CALL0(prev, setspace);
    return ret;
  }
}
```

FIGURE 5.25 – Composant de liaison pour la liaison remontée système

Comme pour la liaison appel système, les composants de liaison sont générés par un générateur spécialisé à la liaison et il contient des instructions assembleurs. Ce n'est donc pas le même que pour la liaison appel système. Le nom du composant de liaison est, par convention, le nom de l'interface concaténé avec "`_upcall`".

Traitant applicatif

Le traitant applicatif est mis en place par l'usine à liaisons. Ce traitant est unique pour toute l'application et se nomme `upcall` par convention. Cela permet au noyau de récupérer l'adresse du traitant lors du chargement de l'application. Le code du traitant de remontée système est le même que le traitant d'appel système. Seule la mise en place de la pile unique de traitement des interruptions est ajoutée.

Actuellement, cette liaison n'est pas sécurisée. Si l'application ne rend pas la main et monopolise le processeur, le noyau ne reprendra jamais le contrôle. Il existe de nombreuses solutions à ce pro-

blème standard, comme par exemple activer un timeout qui va interrompre le traitement s'il est trop long. Une autre solution est d'utiliser des fils d'exécution de service qui seront activés pour traiter la requête, par exemple grâce à la méthode `yield`. Dans ce cas, il peut être nécessaire d'utiliser des segments mémoires partagés pour le passage des arguments.

5.6.3 Liaison signal

Cette liaison, appelée un signal, est similaire à la liaison remontée système. Elle est uniquement utilisée pour propager un événement à l'application en cours d'exécution. Cela permet par exemple de mettre en place des traitants applicatifs pour les exceptions, comme les fautes de page. La seule différence est que le composant de liaison ne change pas l'espace d'adressage actif, cela nécessite donc une usine à liaison et un générateur particulier. Pour tout le reste, le code est le même et nous n'y revenons pas.

5.6.4 Liaison LRPC

Cette liaison, appelée liaison LRPC, permet les interactions entre deux composants situés dans deux applications distinctes. Cette liaison est uniquement la combinaison d'une liaison appel système et d'une liaison remontée système. Ici, le composant de liaison appel système appelle le composant de liaison remontée système qui appelle le composant appelé. L'usine à liaisons met donc en place une liaison contenant deux composants de liaison.

Comme cette liaison est la combinaison de la liaison appel système et de la liaison remontée système, il n'est pas possible de passer en arguments plus de 7 registres (*r4* à *r10*), soit 7 valeurs entières. Des communications plus complexes peuvent être réalisées en utilisant des segments partagés entre les applications ou en ajoutant au LRPC la possibilité de copier des messages plus longs entre les espaces d'adressage.

5.6.5 Liaison distante RPC

La liaison, appelée liaison distante RPC, permet les interactions entre deux composants situés sur des machines géographiquement distantes. Cette liaison repose sur deux composants de liaison ; un talon client et un talon serveur. Elle s'insère directement dans la pile de protocole et elle fonctionne sur Ethernet. Si cette pile s'exécute exclusivement dans le domaine superviseur, il est possible d'utiliser cette liaison depuis les domaines applicatifs en combinant une liaison appel système et une liaison remontée système.

Pour comprendre le fonctionnement de la liaison distante RPC, nous nous basons sur la figure 5.26. Nous allons étudier les composants de liaisons et les sessions mis en œuvre dans cette liaison en suivant le cheminement d'un appel à la méthode `call` de l'interface `Null` d'un composant distant.

Talon client

L'appel est tout d'abord intercepté par le talon client qui implante l'interface `Null`. Ce composant de liaison est mis en place par l'usine à liaisons lors de la création de la liaison. Son nom est, par

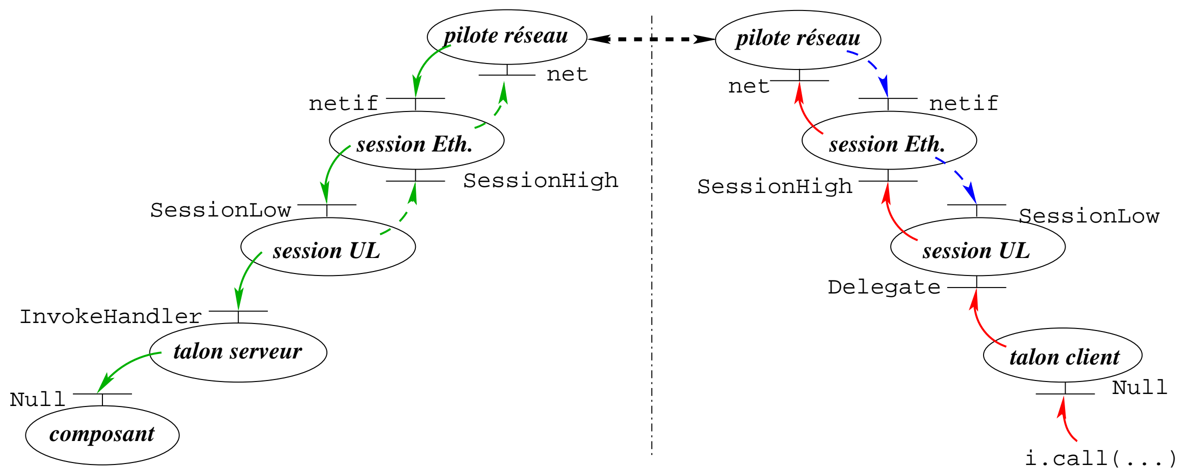


FIGURE 5.26 – Fonctionnement de la liaison distante RPC

convention, le nom de l'interface concaténé avec "_stub". Le rôle du talon client est d'emballer les paramètres dans un paquet réseau dont le format est donné à la figure 5.27. La méthode appelée est identifiée par son ordre dans le tri des méthodes de l'interface. Par convention, un appel à une méthode sans résultat est considéré asynchrone. Nous utilisons donc le même mécanisme pour une communication événementielle.

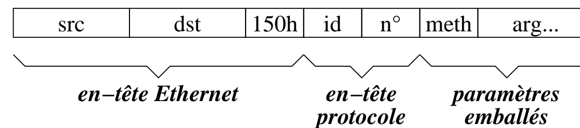


FIGURE 5.27 – Format d'un paquet de la liaison distante

Un exemple de code de talon client, généré par un compilateur dédié, pour l'interface `Null` est donné à la figure 5.28. Le talon commence par demander un paquet à l'usine à paquets, puis il écrit l'entier identifiant la méthode. Il écrit ensuite les arguments de l'appel. Conformément à la pile de protocole et pour éviter les recopies, l'adresse des paramètres dans le paquet est calculée en fonction de la taille de l'en-tête Ethernet et de l'en-tête de la session client sous-jacente. Le talon appelle ensuite la session client avec les arguments que nous allons étudier dans la section suivante. Un paquet contenant le résultat est retourné si l'appel distant est synchrone. L'appel à la session client est alors bloquant. Le talon doit alors déballer le résultat et libérer le paquet. Si l'appel distant est asynchrone, le talon retourne directement à l'appelant. L'appel à la session client n'est pas bloquant.

Session client

La session client est mise en place par l'usine à liaison lors de la création de la liaison, le protocole est mis en place à l'initialisation. Elle exporte l'interface `Delegate` donnée à la figure 5.29. Cette interface n'a qu'une méthode `invoke`. Ses arguments sont un paquet *msg* de taille *size*. La taille des données réservées pour les en-têtes du protocole et de Ethernet est donnée par *reserve*. Si l'appel est

```
static int call(Null2Itf* this, int arg1) {
  struct stubdata* stub = (struct stubdata*)this;
  unsigned int size = 4+4;
  packetItf *request, *reply;
  unsigned char *data, *datareply;
  int result;
  request = CALL1(stub->PF, alloc, size+stub->reserve);
  data = CALL0(request,getvirtaddr)+stub->reserve;
  *(int*)(data) = 1;    data += 4;
  *(int*)(data) = arg1; data += 4;
  reply = CALL4(stub->delegate, invoke, request,
             stub->reserve, size, 1);
  data = CALL0(reply,getvirtaddr)+stub->reserve;
  result = *(int*)(data+0);
  CALL1(stub->PF, free, reply);
  return result;
}
```

FIGURE 5.28 – Composant de liaison pour le talon client

synchrone, l'argument *synchrone* est vrai sinon il est faux.

```
interface Delegate {
   packet   invoke(packet msg, int reserve, int size,
                   int synchrone);
}
```

FIGURE 5.29 – Interface `Delegate`

Le rôle de la méthode `invoke` est d'écrire les paramètres identifiant l'appel distant ; le numéro permettant d'identifier la requête distante et le composant appelé identifié via un index dans une table d'indirection. Cette méthode envoie ensuite le paquet à la session Ethernet sous-jacente avec pour numéro de protocole 150h. Si l'appel distant est asynchrone, la méthode retourne directement. Si l'appel distant est synchrone, la méthode bloque le fil d'exécution sur une condition, par un appel à la méthode `wait` de l'ordonnanceur. Lorsque la session client reçoit, via l'interface `SessionLow`, le paquet contenant le résultat et préalablement démultiplexé par le protocole grâce au numéro de requête, elle réveille le fil d'exécution bloqué par une notification sur la condition. Une fois réveillé, le fil d'exécution retourne le paquet reçu au talon client.

Session serveur

La session client est mise en place par l'usine à liaisons lors de l'exportation d'une interface. Cette session s'insère dans la pile de protocole et elle implante l'interface `SessionLow`. Lorsqu'un

paquet démultiplexé via la clé 150h arrive à cette session, elle commence par vérifier l'existence du composant appelé. Puis elle appelle le talon serveur, via l'interface `InvokeHandler`, que nous allons étudier dans la section suivante. Ce talon va déballer les paramètres et le cas échéant emballer le résultat si l'appel distant est synchrone. Dans ce cas, le même paquet est réutilisé. L'index du composant appelé est positionné sur -1 signifiant une réponse. Le paquet est alors envoyé à la session Ethernet sous-jacente. Le numéro de requête est inchangé, ce qui permet du côté client de démultiplexer les réponses.

Talon serveur

Le talon serveur est mis en place par l'usine à liaisons lors de l'exportation d'une interface. Le rôle du talon est de déballer les paramètres de l'appel distant et de construire un appel au composant serveur. Il exporte l'interface `InvokeHandler` donnée à la figure 5.30. Cette interface n'a qu'une méthode `invoke` qui est appelée par la session serveur. Son seul argument *args* est un tableau d'octets, référençant directement les données du paquet, contenant les paramètres de l'appel distant.

```
interface InvokeHandler {
   int      invoke(byte[] args);
}
```

FIGURE 5.30 – Interface `InvokeHandler`

Un exemple de code de talon serveur, généré par un générateur dédié, pour l'interface `Null` est donné à la figure 5.31. Après démultiplexage de la méthode, le talon déballe les paramètres et crée les arguments requis pour appeler le composant serveur. Il effectue ensuite l'appel à la méthode de l'interface. Si la méthode retourne un résultat, l'appel distant est synchrone et le client est donc en attente d'un résultat. Ce résultat est emballé dans le même tableau d'octets et la taille du résultat emballé est retournée comme résultat à la session serveur.

5.6.6 Liaison de synchronisation

Comme nous l'avons vu, la liaison flexible peut aussi être utilisée pour assurer diverses fonctionnalités. Nous avons vu que les composants n'implantent pas de synchronisation pour ne pas forcer ce coût si elle n'est pas nécessaire. La liaison de synchronisation permet donc de synchroniser les accès à un composant en reportant la synchronisation dans un composant de liaison, voir la figure 5.32. En fait, cette liaison offre la fonctionnalité de moniteur.

Composant de liaison

Ce composant de liaison est mis en place par l'usine à liaisons lors de la création de la liaison. Son nom est par convention le nom de l'interface concaténé avec `"_monitor"`. Son rôle est de synchroniser les appels à un composant. Un exemple d'un tel composant est donné à la figure 5.33. Ce composant commence par verrouiller l'interface, puis il appelle le composant sous-jacent. Pour finir, il déverrouille l'interface et renvoie le résultat. Dans l'implantation actuelle, si un composant est

```
static int invoke(InvokeHandlerItf* this,
                  unsigned char *data) {
  struct skeletondata* skeleton = (struct skeletondata*)this;
  switch(*(int*)(data+0)) {
  case 1: {
      int arg0;
      unsigned char* datareply = data;
      int result;
      data += 4;
      arg0 = *(int*)data;
      data+=4;
      result = CALL1(skeleton->target, call, arg0);
      *(int*)(datareply) = result;
      return sizeof(int);
    };
  };
}
```

FIGURE 5.31 – Composant de liaison pour le talon serveur

FIGURE 5.32 – Liaison de synchronisation

accessible par deux interfaces, les accès ne sont pas synchronisés du fait de l'utilisation d'un verrou pas interface et non par composant.

5.7 Composants domaines

La bibliothèque KORTEX implante que très sommairement le concept des domaines. Actuellement, seuls un mini-courtier et un chargeur dynamique sont proposés.

5.7.1 Le mini-courtier

Dans le modèle d'architecture THINK, certains noms sont bien connus, comme le nom de l'usine à liaisons locale. Mais les autres noms ne seront connus qu'au moment de l'exécution lorsque le composant va exporter ses interfaces. Or, il doit être possible de désigner statiquement certaines interfaces, comme l'allocateur de mémoire. La solution consiste à utiliser un courtier de nom bien connu, en fait un serveur de noms, permettant d'associer un attribut, en fait un nom symbolique, à un nom de type `Name`.

```
static int call(NullItf* this, int arg1) {
  struct monitordata* monitor = (struct monitordata*)this;
  int result;
  CALL0(monitor->lock, lock);
  result = CALL1(monitor->target, call, arg1);
  CALL0(monitor->lock, unlock);
  return result;
}
```

FIGURE 5.33 – Composant de liaison pour la synchronisation

L'interface `Trader`, donnée à la figure 5.34, est l'interface offerte par un composant courtier. La méthode `_register` enregistre l'association d'un attribut, sous forme de chaîne de caractère, avec un nom. La méthode `lookup` recherche le nom correspondant à un attribut préalablement enregistré.

```
interface Trader {
   void     _register(Name name, String attr);
   Name     lookup(String attr, int idx);
}
```

FIGURE 5.34 – Interface du mini-courtier

Ainsi, quand un composant exporte une interface, il peut enregistrer le nom obtenu auprès du mini-courtier. Le choix de l'attribut est arbitraire, il peut être spécifique à une interface ou à un composant, c'est le concepteur qui décide. Pour simplifier, l'attribut peut éventuellement être le nom de l'interface. Il est préférable d'associer un attribut à une interface, car certains attributs sont partagés entre plusieurs composants, c'est le cas pour les pilotes de carte réseau. Cet attribut fait partie de la spécification du composant. L'utilisateur peut alors utiliser le composant, via un `lookup` et un `bind`.

5.7.2 Chargeur dynamique

Le composant chargeur dynamique ajoute dynamiquement des composants dans un domaine. Ce composant charge et lie dynamiquement un composant contenu dans un objet au format binaire ELF « Executable and Linking Format » [TIS 1993] issu d'une compilation. Le chargeur est aussi utilisé par les usines à liaisons pour charger les composants de liaisons. Son interface a déjà été présentée à la figure 3.13 du chapitre 3.

Le rôle du chargeur dynamique est de résoudre les liens sur les symboles langages qui n'ont pas pu l'être statiquement car l'adresse mémoire est connue seulement à l'exécution. L'association entre un symbole et une adresse mémoire est mémorisée dans un contexte de nom. Les exportations de symboles sont réalisées par le domaine en fonction de sa sémantique, avec comme exportation minimum, les noms biens connus de la structure d'accueil. Ensuite, le chargeur démarre le composant en appelant son constructeur nommé *nomProbe*, comme nous l'avons vu à la section 4.4.2.

5.8 Conclusion

Nous avons présenté dans ce chapitre l'ensemble des composants et liaisons fournis par la bibliothèque KORTEX. Grâce à l'utilisation du concept d'interface, la spécification et l'utilisation de tels composants sont grandement simplifiées. En fait, l'utilisation de l'architecture logicielle KORTEX simplifie le développement des systèmes, mais aussi et surtout sa maintenance, en le ramenant à la composition d'un ensemble de composants. Par exemple, l'utilisation de ce modèle pour le développement d'un noyau monolithique, même gros et complexe comme Linux, simplifierait grandement le processus et la maintenance tout en minimisant les risques de défaillances.

Cette bibliothèque KORTEX est encore actuellement très incomplète. Il n'y a par exemple aucun composant pour piloter les disques, ni composant pour gérer les fichiers sur disques. Il n'y a pas non plus de composants pour la gestion des caches des fichiers. KORTEX ne fournit pas non plus de composants permettant de traiter les éventuelles fautes d'accès mémoire des applications, ni de composants de gestion d'échange de page avec le disque « swapping ».

Néanmoins, comme nous le verrons cette bibliothèque permet d'exécuter de nombreuses applications et de construire des systèmes déjà relativement complets. Les tableaux 5.1 et 5.2 récapitulent partiellement les interfaces rencontrées dans KORTEX. Pour chaque interface nous donnons l'attribut enregistré dans le mini-courtier, ainsi que les composants qui l'implantent. La dernière colonne donne le nombre de lignes de code C commenté des composants. À titre d'information, la somme des lignes de tous les composants est d'environ 30 000 lignes de code C.

Interface	Attribut	Composant	Description	Lignes
		Mémoire		
`MMU`		mmu	MMU POWERPC	280
`Page`	`"page"`	buddy	buddy-system	270
`Allocator`	`"allocator"`	dlmalloc	allocateur	474
`Sbrk`	`"sbrk"`	sbrkmap	sbrk mémoire paginée	94
"	"	sbrk	sbrk mémoire plate	72
`Space`	`"space"`	flat	mémoire plate	134
"	"	pagetable	mémoire paginée	292
`SpaceFactory`	`"spacefactory"`	spacefactory	usine à espaces d'adressage	65
		Processeur		
`Trap`		trap	exceptions POWERPC	109+90[a]
`Scheduler`	`"Scheduler"`	roundrobin	ordon. préemptif circulaire	396
"	"	priority	ordon. préemptif à priorités	415
"	"	cooperative	ordon. coopératif	377
`irq`	`"irq"`	IRQ	gestionnaire interruption	121
		Domaine		
`Trader`		trader	mini-courtier	91
`Loader`	`"loader"`	loader	chargeur dynamique	341
`Starter`	`"starter"`	starter	chargeur application	310

[a] lignes d'assembleur

TABLEAU 5.1 – Récapitulatif des composants KORTEX

Le tableau 5.3 récapitule les composants usines à liaisons rencontrées dans KORTEX. Dans l'implantation, une usine à liaisons est composée, soit d'un composant unique qui exporte les interfaces

Interface	Attribut	Composant	Description	Lignes
		Réseau		
net	"eth"	mace	pilote carte mace	659
"	"	bmac	pilote carte bmac	855
"	"	gmac	pilote carte gmac	1063
"	"	tulip	pilote carte PCI tulip	2208
PacketFactory	"packetfactory"	packet	usine à paquets	149
ProtocolHigh	"ethernet"	ethernet	protocole Ethernet	417
"	"iphigh"	ip	protocole IP	476
"	"udphigh"	udp	protocole UDP	279
"	"tcphigh"	tcp	protocole TCP	802
"	"rpchigh"	rcp	protocole RPC	335
Arp	"arp"	arp	protocole ARP	240
Socket	"tcp"	tcp	socket TCP	
"	"ucp"	ucp	socket UDP	
VFS	"nfs"	nfs	protocole NFS	752
		Graphique		
framebuffer	"fb"	offb	pilote carte vidéo	366
font	"font"	F8x16	police de caractères	4667
console	"console"	console	console	129
		Contrôleurs		
FW	"FW"	firmware	pilote firmware	821
pci	"pci"	uninorth	pilote PCI uninorth	315
"	"	grackle	pilote PCI grackle	243
"	"	pmac	pilote PCI bandit & chaos	398
pic	"pic"	openpic	pilote PIC openpic	405
"	"	pmacpic	pilote PIC macio & GC	198
via	"via"	pmu	pilote ADB pmu	692
"	"	cuda	pilote ADB cuda	450

TABLEAU 5.2 – Récapitulatif des composants KORTEX (suite)

BindingFactory et NamingContext, soit de deux composants qui exportent chacun une interface et qui s'exécutent chacun dans des domaines distincts. Pour chaque usine à liaisons nous donnons le ou les composants ainsi que le domaine d'exécution des composants.

Nous verrons concrètement au chapitre suivant comment tous ces composants et interfaces se composent en vue de former le système voulu.

Interface	Composant	Description	Domaine d'exécution
`BindingFactory`	BFlocal	liaison locale	tous
”	BFremote	liaison distante	superviseur
”	SCclient	liaison appel système	applicatif
”	UCclient	liaison remontée système	superviseur
”	Sigclient	liaison signal	superviseur
`NamingContext`	BFlocal	liaison locale	tous
”	BFremote	liaison distante	superviseur
”	SCserver	liaison appel système	superviseur
”	UCserver	liaison remontée système	applicatif
”	Sigserver	liaison signal	applicatif

TABLEAU 5.3 – Récapitulatif des usines à liaisons KORTEX

Chapitre 6

Évaluations

Ce chapitre présente l'évaluation qualitative et quantitative de l'architecture THINK et de son implantation KORTEX sous la forme d'une bibliothèque de composants systèmes. L'évaluation qualitative doit montrer que l'approche satisfait aux objectifs fixés, à savoir : la possibilité d'implanter au plus bas niveau des fonctions classiquement fournies par les intergiciels, et la possibilité de parcourir l'espace de conception vu à la figure 2.26 de la page 47. L'évaluation quantitative doit montrer que l'architecture n'implique pas un coût d'exécution inacceptable qui ne rendrait pas viables les propositions, aussi belles soient-elles.

Les expérimentations consistent donc à implanter différents systèmes en vue de couvrir au maximum l'espace de conception ; micronoyau, système monolithique, système dédié, etc. Pour chaque système, nous évaluons le coût global mesuré et nous voyons comment est construit le système par composition d'un ensemble de composants de la bibliothèque KORTEX.

6.1 Construction de systèmes d'exploitation

Pour l'évaluation, de façon qualitative et quantitative, des propositions faites dans cette thèse, différents noyaux de systèmes d'exploitation ont été construits. Nous allons commencer par étudier comment sont construits ces noyaux en utilisant d'une part, les outils de développement proposés et d'autre part, la bibliothèque KORTEX. Nous évaluons pour cela le coût des composants POWERPC, le coût des liaisons systèmes et le coût de la liaison distante. Cette expérience montre concrètement que KORTEX peut être utilisé pour construire des systèmes d'exploitation complexes et comme nous le verrons par la suite des systèmes minimums dédiés à des applications spécialisées.

6.1.1 Construction du noyau

La construction d'un système, ou d'un noyau de système, s'effectue par composition de composants KORTEX à l'aide des outils proposés. L'ajout de composant au noyau étend, de façon statique ou dynamique, le noyau. Les sections suivantes montrent comment construire un noyau d'un système basé sur un micronoyau minimum. Puis en ajoutant des composants à ce noyau, nous verrons comment il peut évoluer vers un noyau extensible et finalement vers un noyau de système réparti.

Noyau pour micronoyau minimum de type L4

Nous avons conçu un noyau minimum en utilisant les concepts du micronoyau L4 [Liedtke 1995]. Nous avons choisi L4 car il a été conçu pour être minimum et efficace. Néanmoins, quoique fonctionnellement équivalents, les services L4 implantés ici ne respectent pas encore les interfaces spécifiées par ses développeurs [Liedtke 1996a].

Les concepts L4 sont implantés en utilisant exclusivement les composants de la bibliothèque KORTEX. Les concepts présents dans un tel micronoyau sont les espaces d'adressage, les fils d'exécution et les IPC (asynchrone). Les composants `priority`, `pagetable` et `spacefactory` sont donc utilisés pour construire le domaine superviseur correspondant au noyau. Le noyau construit requiert aussi l'allocateur de mémoire `dlmalloc` car certains composants nécessitent une allocation dynamique de la mémoire.

La figure 6.1 schématise l'organisation des composants KORTEX dans le micronoyau minimum de type L4. Ce schéma illustre concrètement l'utilisation et le fonctionnement de différents composants KORTEX décrits au chapitre précédent. Il montre les composants, les interfaces que ces derniers exportent et les différentes interactions via des liaisons langages et des liaisons systèmes locales. Ces dernières sont mises en place par l'usine à liaisons locales `BFlocal` en passant lors de la demande de création d'une liaison (via l'interface `BindingFactory`) le nom de l'interface cible préalablement exportée par un composant (via l'interface `NamingContext`). Ce nom peut potentiellement être récupéré depuis le mini-courtier `trader` d'interface `Trader`.

Les concepts d'espaces d'adressage et de fils d'exécution sont fournis par le composant usine à espaces d'adressage `spacefactory` et par le composant ordonnanceur `priority`. Ils permettent de créer respectivement les ressources fils d'exécution de type `Thread` et les ressources espaces d'adressage applicatifs de type `Space`, ces ressources sont aussi des composants. Plusieurs fils d'exécution peuvent s'exécuter dans un même espace d'adressage. Pour réaliser les changements de contexte, l'ordonnanceur requiert l'interface `Trap` (implantée par le composant `trap` du POWERPC permettant de réifier les exceptions, ici l'exception temporelle). Les composants implantant l'interface `Space`, permettant la manipulation des espaces d'adressage, requièrent d'une part l'interface `Page` (implantée par le composant « buddy-system » `buddy` permettant l'allocation des pages physiques) et d'autre part l'interface `MMU` (implantée par le composant `mmu` du POWERPC permettant le couplage des pages physiques avec des pages virtuelles).

Le composant `pagetable` représentant l'espace d'adressage du noyau est particulier. Il est identique aux ressources représentant les espaces d'adressages applicatifs et il exporte la même interface `Space`. Mais, il n'est pas créé par l'usine à espaces d'adressage et il est démarré bien avant car son constructeur positionne l'unique table de page inversée conformément aux spécifications du POWERPC.

Pour créer dynamiquement des composants, les deux usines (`priority` et `spacefactory`) requièrent l'interface `Allocator` permettant d'allouer des blocs de mémoire. Cette interface est fournie par le composant `dlmalloc` qui requiert l'interface `Sbrk` dont la fonction est d'agrandir le segment de données du noyau. Cette interface est implantée par le composant `sbrkmap` qui requiert d'une part le « buddy-system » (que nous venons de voir) pour allouer des pages physiques et d'autre part l'espace d'adressage du noyau (que nous venons de voir) afin de demander le couplage des pages physiques dans l'espace d'adressage.

Pour finir, l'interface `FW` (implantée par le composant `firmware`) est utilisée pour obtenir des informations sur la machine, comme la fréquence du processeur, la taille de la mémoire, la taille

initiale du système, etc.

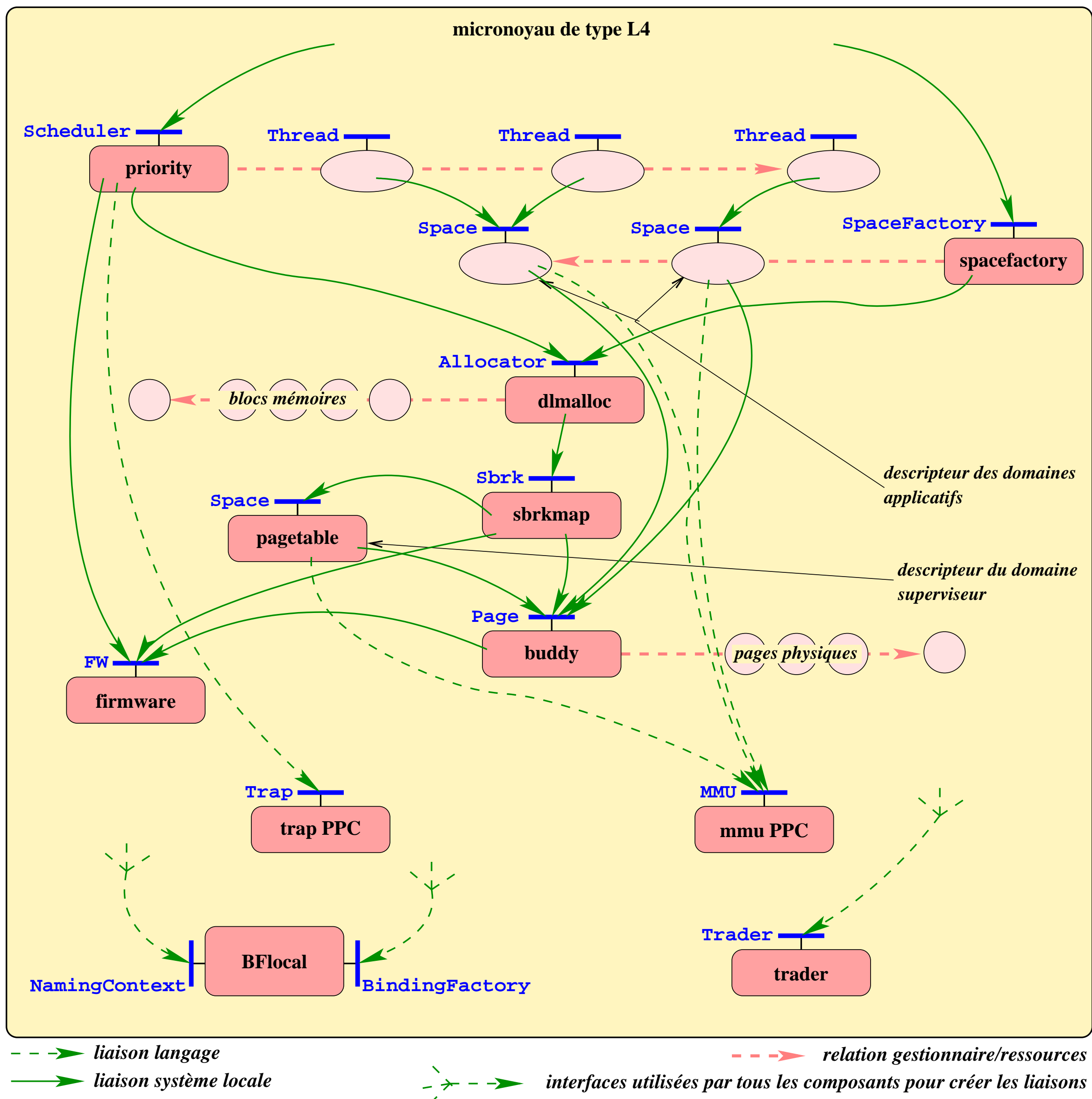


FIGURE 6.1 – Organisation schématique du micronoyau minimum

La figure 6.2 donne le graphe de composition tel que le propose l'outil de visualisation. Tous ces composants implantant les concepts L4 s'exécutent dans le domaine superviseur. Les liaisons appel système, remontée système et LRPC sont utilisées pour réaliser les interactions entres les différentes applications et le noyau, elles n'apparaissent pas sur le graphe. Le composant `powerpc` comprend les composants `mmu` et `trap`, il est utilisé pour limiter à l'affichage le nombre de dépendance.

La taille totale de l'exécutable de ce noyau est de 43 Ko. La majeure partie de cet exécutable est requise uniquement pendant l'initialisation. En la rassemblant dans un segment spécial elle peut être supprimée à la fin de l'initialisation. Finalement, la taille du code, comprenant tous les composants

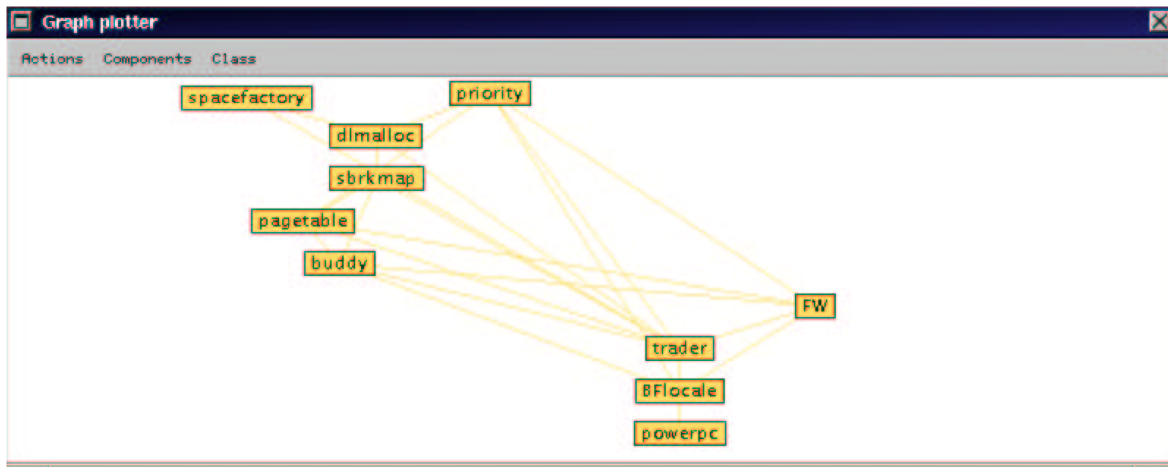


FIGURE 6.2 – Graphe de composition d'un micronoyau minimum de type L4

exposés à la figure 6.2, de ce micronoyau est d'environ 16 Ko. Ce qui est très peu vis-à-vis des fonctionnalités offertes. En comparaison, la taille de l'exécutable du micronoyau L4 complètement écrit en assembleur se situe entre 10 Ko et 15 Ko. Le micronoyau généralement cité pour sa taille réduite est QNX, dont l'exécutable a une taille de 12 Ko [Hildebrand 1992].

Noyau pour système extensible

La flexibilité de l'architecture THINK permet d'exécuter les composants sans les modifier soit dans le domaine superviseur, soit dans des domaines applicatifs. Donc, pour des raisons d'efficacité, les composants peuvent s'exécuter dans le domaine superviseur. Cela revient à étendre statiquement le domaine superviseur et donc le noyau. En ajoutant des fonctions de chargement dynamique de composant à ce domaine, nous obtenons un noyau pour système d'exploitation extensible. Pour obtenir cela, il suffit d'ajouter le composant `loader` au noyau lors de sa construction. Néanmoins, les éléments pour un système réellement extensible ne sont pas encore présents. Par exemple, il n'est pas encore possible de décharger un composant, il n'y a donc pas de mécanisme vérifiant l'intégrité du système avec une telle opération.

Le chargement dynamique de code requiert des services de communication permettant de récupérer le fichier binaire contenant le composant. Comme la gestion des disques n'est actuellement pas proposée par KORTEX, la lecture des fichiers s'effectue par NFS. Comme le composant `NFS` s'insère dans une pile de protocole et requiert une multitude d'interfaces d'accès aux ressources matérielles, les composants, implantant ces interfaces, sont aussi ajoutés au noyau. Dans le cadre qui nous intéresse, nous avons fait un noyau intégrant tout le code pour une famille de machine POWERMACINTOSH et donc le noyau intègre tous les pilotes pour chaque type de périphériques rencontrés : réseaux, PCI, PIC, etc. La figure 6.3 donne le graphe de composition pour un tel noyau. À titre d'information, la taille de l'exécutable de ce noyau extensible est de 160 Ko.

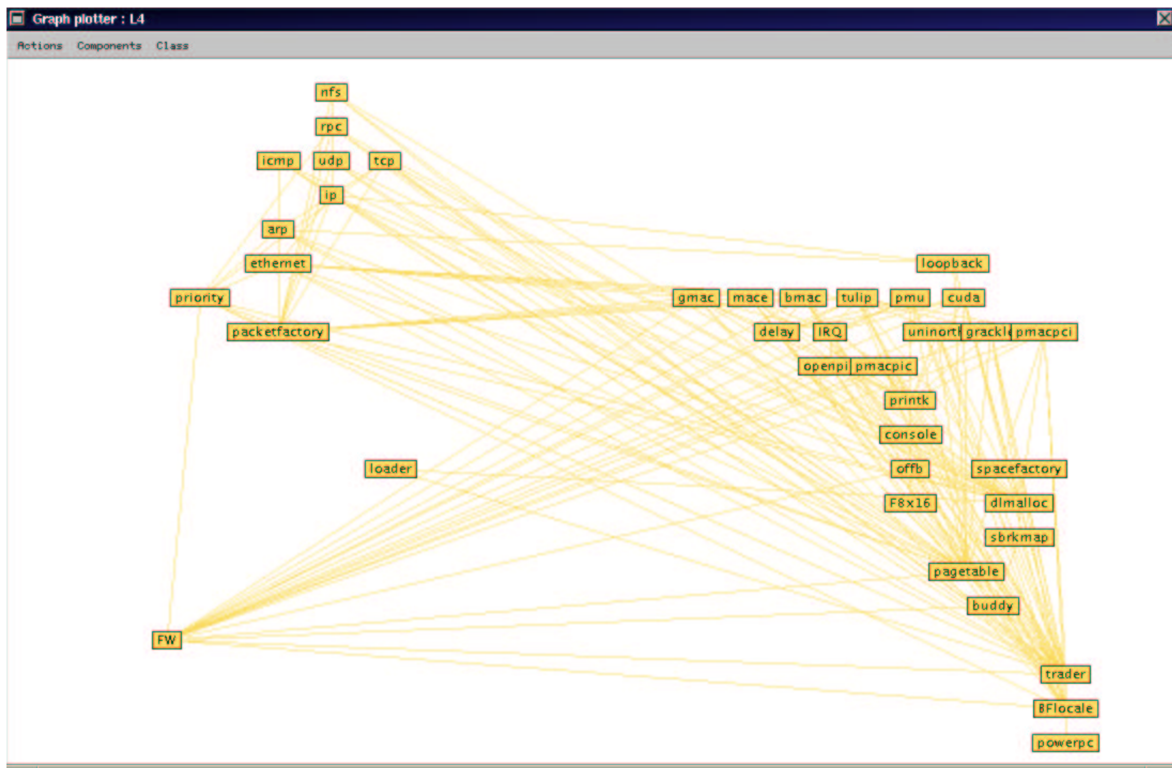


FIGURE 6.3 – Graphe de composition d'un noyau extensible

Noyau pour infrastructure logicielle répartie

Avec le noyau précédent, nous avons un noyau extensible pour système d'exploitation centralisé. En ajoutant simplement la liaison distante offerte par KORTEX au noyau, le système devient de facto une infrastructure logicielle répartie.

6.1.2 Évaluation des composants POWERPC

La première évaluation réalisée est celle des composants `trap` et `mmu` permettant de réifier respectivement les exceptions et la MMU du POWERPC, voir la section 5.2. Nous allons donc étudier, en utilisant implantation du noyau L4, le coût de ces composants lors de la prise en compte d'une exception et lors de changement de contexte.

Traitement d'une exception

Lorsqu'une exception est levée, elle est récupérée par le composant `trap` qui appelle le traitant système enregistré. Une fois le traitement système terminé, un appel à la méthode `trap.Return` termine l'exception. Le coût de l'entrée et de la terminaison d'une exception est de 135 cycles et il est détaillé au tableau 6.1. Sur un POWERPC G4 à 500 MHz, cela est équivalent à un temps de 270 nanosecondes.

méthode	instructions	temps	cycles
`trap.Enter`$_{id}$	57	0,160 μs	80
`trap.Return`	48	0,110 μs	55
total	105	0,270 μs	135

TABLEAU 6.1 – Évaluation du traitement d'une exception

Changement de contexte

Comme nous l'avons vu au chapitre précédent, la réalisation du changement de contexte est basée sur les méthodes `Enter`, `Return` et `SetContext` de l'interface `Trap`. La réalisation des changements d'espace d'adressage est basée sur la méthode `SetSegment` de l'interface `MMU`. Du fait de leur simplicité et de leur efficacité, le changement de contexte peut être très efficace.

Le tableau 6.2 donne les coûts des différentes opérations de changement de contexte. Tous les temps sont donnés pour un POWERPC G4 à 500 MHz et ils ne comptent pas le coût de l'algorithme d'ordonnancement.

changement	instructions	temps	cycles
fil d'exécution	111	0,284 μs	142
processus	147	0,394 μs	197
espace d'adressage	32	0,104 μs	52

TABLEAU 6.2 – Évaluation d'un changement de contexte

Un changement de contexte entre deux fils d'exécution d'un même espace d'adressage prend 142 cycles, soit 284 nanosecondes. Un changement de contexte de processus, comprenant un changement d'espace d'adressage, prend 197 cycles, soit 394 nanosecondes. Du fait de l'architecture POWERPC qui ne nécessite pas de vider la TLB lors d'un changement d'espace d'adressage, le coût d'un tel changement est de 52 cycles, soit 104 nanosecondes. Cela permet l'utilisation de quantum de temps extrêmement court, ce qui est très utile pour les systèmes temps réels ou les micronoyaux.

6.1.3 Évaluation des liaisons systèmes

Pour évaluer les différentes liaisons fournies par KORTEX, nous utilisons l'interface `Test` donnée à la figure 6.4. La méthode `call` permet d'effectuer des appels synchrones avec un seul argument *this* et elle retourne un entier.

```
interface Test {
   int    call();
}
```

FIGURE 6.4 – Interface `Test`

La figure 6.5 donne le code du composant serveur utilisé. Ce composant exporte l'interface `Test` et ne fait aucun traitement. Nous ne donnons pas le code du constructeur requis pour l'exportation de l'interface auprès du contexte de désignation de la liaison.

```
static unsigned int call(NullItf* this) {
  return 0x12345;
}
static struct TestMeth testmeth = {
  nullcall
};
TestItf testitf = {&testmeth};
```

FIGURE 6.5 – Code serveur d'un composant exportant l'interface `Test`

La figure 6.6 donne le code client réalisant l'appel du composant serveur. Il est utile de rappeler que ce code est le même quel que soit le type de liaison ; locale, appel système, remontée système, distante, etc. Cela montre le modèle de programmation uniforme offerte par le concept de liaison flexible. Nous ne donnons pas le code requis pour la création de la liaison.

```
result = target->meth->call(target);
```

FIGURE 6.6 – Code client de l'appel d'un composant serveur

Performance des liaisons systèmes

Toutes les évaluations sur les liaisons sont réalisées sur le micronoyau de type L4. Tous les temps des mesures sont donnés pour un POWERPC G4 à 500 MHz avec 128 Mo de mémoire. Le tableau 6.3 résume le coût des interactions synchrones sur chaque type de liaison fournie par KORTEX. Une interaction, via une liaison locale, prend 8 cycles, soit 16 nanosecondes. Cette mesure montre que le coût d'une interaction locale entre des composants conformes à l'architecture THINK n'engendre pas une pénalité significative.

liaison	instructions	temps	cycles
local	6	0,016 μs	8
appel système	115	0,300 μs	150
appel système optimisé	50	0,162 μs	81
signal	35	0,128 μs	64
remontée système	107	0,346 μs	173
LRPC	217	0,630 μs	315
LRPC optimisé	152	0,490 μs	245

TABLEAU 6.3 – Performance des liaisons KORTEX

L'évaluation montre que la liaison appel système prend 150 cycles, soit 300 nanosecondes. Ce coût peut être diminué à seulement 81 cycles, soit 182 nanosecondes, en appliquant l'optimisation sur les appels systèmes. En comparaison, un appel système `getpid` sur un noyau Linux 2.4.x prend 217 cycles, soit 434 nanosecondes. Ce qui représente un coût supplémentaire de 67 cycles (44 %) comparé à la version KORTEX non optimisée, et un coût supplémentaire de 136 cycles (167 %) comparé à la version optimisée.

La liaison signal permettant de propager une exception dans un domaine applicatif prend 64 cycles, soit 128 nanosecondes. La liaison remontée système permettant d'appeler depuis le domaine superviseur un composant situé dans un domaine applicatif prend 173 cycles, soit 346 nanosecondes. Cette valeur correspond en temps à un signal et à deux changements de contexte. Il est très difficile de comparer ces chiffres avec un autre système car la sémantique varie fortement d'un système à un autre. Par exemple, les signaux Unix changent la sémantique de l'exception avant de la propager. De plus, la propagation des exceptions dans le micronoyau L4 est réalisée via un fil d'exécution en attente qu'il faut réveiller lors de l'exception.

La liaison LRPC permettant d'appeler depuis un domaine applicatif une méthode d'une interface d'un composant situé dans un autre domaine applicatif prend 315 cycles, soit 630 nanosecondes. Comme nous l'avons vu, cette liaison est la combinaison d'une liaison appel système et d'une liaison remontée système. L'utilisation de la liaison appel système optimisé diminue le coût de la liaison LRPC à 245 cycles, soit 490 nanosecondes.

Il est intéressant de comparer cette liaison LRPC à l'IPC du micronoyau L4 [Liedtke 1996b, Liedtke et al.]. Un IPC L4 consiste à activer un fil d'exécution de service d'un autre processus en attente d'une condition [Liedtke 1993]. Ce qui revient en fait à faire un IPC dans chaque sens pour faire un LRPC. Les IPC L4 ne sont pas sécurisés, c'est au fil d'exécution de service de vérifier la validité des requêtes. Dans ces papiers, il est dit que le coût d'un IPC simple est de $2 * 121$ cycles sur Pentium ($2 * 73$ nanosecondes à 166 MHz), $2 * 86$ sur MIPS ($2 * 860$ nanosecondes à 100 MHz) et $2 * 45$ cycles sur Alpha (100 nanosecondes à $2 * 433$ MHz). Mais cet IPC est optimisé pour être utilisé sur de petits espaces d'adressage, voir la section 2.7.3. Comme notre LRPC fonctionne sur de grands espaces d'adressage, 40 % du coût est dû aux changements d'espace d'adressage, les coûts sont détaillés dans le paragraphe suivant. Il est donc difficile de comparer la liaison LRPC avec l'IPC L4. Même si d'après certaines explications ont peut supposer que le coût d'un LRPC construit sur un vrai IPC L4 est de 622 cycles sur un Pentium à 133 MHz [Hohmuth 1999]. Néanmoins, il serait intéressant d'implanter l'IPC L4 avec ses optimisations sous forme d'un nouveau type de liaison THINK pour comparer réellement les résultats.

Le tableau 6.4 propose une analyse détaillée d'une liaison LRPC. Pour chaque opération effectuée, le coût en cycle et le domaine d'exécution sont donnés. Comme la liaison LRPC est la combinaison de deux autres liaisons, la liaison correspondante à l'opération est aussi donnée. Toutes ces opérations peuvent être retrouvées dans les codes des composants de liaison donnés à la figure 5.22 page 105 et à la figure 5.25 page 107, ainsi que dans le code du traitant système donné à la figure 5.23 page 106.

Comme nous venons de le voir, toutes les interactions, via des liaisons KORTEX, sont très efficaces. Cette performance est obtenue en respectant complètement le modèle de programmation uniforme défini dans THINK. Cela prouve que l'architecture THINK n'est pas contraire à l'efficacité. Cette architecture peut donc réellement être utilisée pour construire des systèmes flexibles et extensibles.

opération	**cycles**	**domaine**	**liaison**
appel composant appel système	5	utilisateur	appel système optimisé
emballage paramètre	2	”	”
instruction assembleur `sc`	8	”	”
sauvegarde registres	31	noyau	”
appel composant remontée système	19		”
changement espace d'adressage	49	”	remontée système
simulation interruption applicative	23	”	”
appel cible	7	utilisateur	”
retour d'interruption	11	”	”
restauration registres	11	noyau	”
changement espace d'adressage	47	”	”
retour de l'appel système	31	”	appel système optimisé
déballage résultat	1	utilisateur	”
total	245		

TABLEAU 6.4 – Analyse détaillée d'une liaison LRPC optimisée

6.1.4 Évaluation de la liaison distante RPC

La liaison distante RPC permet les interactions entre deux composants situés dans des domaines superviseurs situés sur des machines géographiquement distantes. L'expérimentation est réalisée avec deux POWERMACINTOSH G4 à 500 MHz dotés de cartes réseau Tulip Asanté Fast PCI à 100 Mbits. Le tableau 6.5 donne les résultats pour une interaction distante synchrone. Les mesures sont données pour un réseau Ethernet à 10 Mbits et à 100 Mbits. Pour un réseau à 10 Mbits le coût de l'interaction est de 180 microsecondes et pour un réseau à 100 Mbits le coût est de 40 microsecondes.

réseau	**temps (μs)**			
	total	**réseau**		**emballage**
		lien	**pilote**	**& déballage**
10baseT	180 μs	164,7 (91,5 %)	11,3 (6,3 %)	4 (2,2 %)
100baseT	40 μs	24,7 (61,7 %)	11,3 (28,3 %)	4 (10 %)

TABLEAU 6.5 – Performance de la liaison distante RPC synchrone

Une référence standard pour l'implantation de communication RPC efficace, dont la sémantique est similaire à la nôtre, est faite par [Thekkath et Levy 1993] pour des réseaux à 10 Mbits. Ces travaux ont mesuré le coût théorique minimum d'un RPC sur une DECstation 5000/200 utilisant un MIPS R3000 à 25 MHz et sur une SparcStation I utilisant un Sparc à 25 MHz. Sur la DECstation, le coût de l'interaction est de 340 microsecondes, dont 140 microsecondes de traitement logiciel et 200 microsecondes de coût matériel. Pour la SparcStation, le coût est de 496 microsecondes, dont 296 microsecondes de traitement logiciel et 200 microsecondes de coût matériel. La comparaison doit porter sur le coût logiciel et le coût matériel. Le coût matériel est de 164.7 microsecondes sur

KORTEX, modulo les évolutions des techniques matérielles, cette valeur peut être considérée comme identique aux 200 microsecondes mesurées par [Thekkath et Levy 1993]. Le coût logiciel sur le G4 est de 15.3 microsecondes. En supposant qu'à la fréquence de processeur équivalente les performances de calcul sont équivalentes, le coût logiciel estimé de notre RPC à 25MHz est de $(500/25) * 15.3 = 306$ microsecondes. Ce coût est le double comparé à la DECstation, mais il est comparable à celui sur la SparcStation, et ce probablement pour les mêmes raisons qui ne sont pas clairement identifiées dans [Thekkath et Levy 1993]. Nous pouvons donc penser que le coût de cette liaison est proche du minimum.

Les performances de la liaison distante peuvent aussi être comparées avec les différents ORBs connus. Bien évidemment, les services fournis par cette liaison sont moindres. Mais cela ne porte pas à conséquence lorsque l'objectif d'une application est seulement d'interagir avec des composants distants. Par exemple, le coût d'une interaction sur l'ORB Java Jonathan [Dumant et al. 1998] est de 2 à 3 millisecondes. Certains ORBs, écrits en langage natif, peuvent baisser le coût de l'interaction à 1 milliseconde, mais rarement moins. Le gain dû au débit réseau est ici non significatif.

Nous voyons ici toute la puissance de la flexibilité qui permet d'atteindre de très bonnes performances en construisant des systèmes spécialisés en fonctions des besoins et des contraintes tout en gardant un modèle de programmation uniforme. Par exemple, il est possible de construire des mécanismes de communications distantes d'un coût de 40 microsecondes par interaction, alors que le mécanisme générique offert par les ORBs est au minimum 25 fois plus lent.

6.2 Nonnoyau

Une expérience intéressante est la construction d'un *nonnoyau*. Un nonnoyau est en fait un noyau qui propose aucune abstraction. Son rôle est uniquement de réifier les exceptions auprès des applications qui s'exécutent dans des domaines applicatifs. De plus, chaque application a un accès direct aux registres des contrôleurs de périphériques, voir la figure 6.7. L'utilisation de la liaison signal propage les exceptions aux applications.

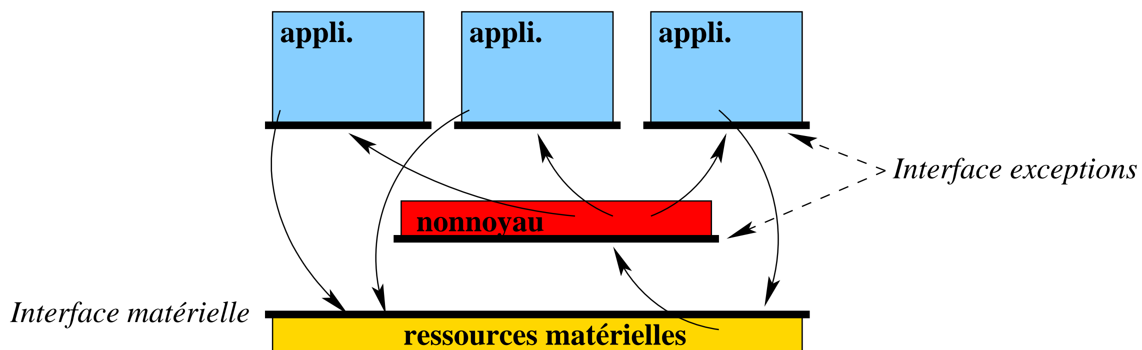


FIGURE 6.7 – Infrastructure du nonnoyau

Un tel noyau ne fournit aucun mécanisme d'allocation, c'est-à-dire que les applications doivent coopérer entre elles pour se partager les ressources. Aucune protection n'est fournie par le nonnoyau pour assurer la sécurité du système. C'est-à-dire qu'une application peut écraser les données d'une autre en couplant ses pages physiques, elle peut aussi corrompre les ressources matérielles en mo-

difiant de façon arbitraire les registres des contrôleurs. De plus, rien n'est dit sur la désignation de l'application à qui est propagée une interruption matérielle. Actuellement, quelle que soit l'exception, elle est toujours propagée à l'application courante. Un tel noyau pose aussi des problèmes d'amorçage et du lancement de la première application qui est actuellement inclut dans l'exécutable du noyau.

Bien évidemment, un tel noyau n'a que peu d'intérêt, si ce n'est de montrer qu'on peut aller au maximum dans la philosophie de l'exonoyau et offrir une flexibilité supérieure. Par exemple, le non-noyau n'offre pas de mécanisme de sécurité, ni de partage, nous atteignons donc un point beaucoup plus extrême qu'avec un exonoyau. Cela montre que l'architecture THINK et la bibliothèque KORTEX offrent un degré de flexibilité importante.

6.3 Systèmes dédiés

Dans cette section, nous décrivons différentes expérimentations dans la composition de système dédié à des applications spécialisées, comme des routeurs et des applications embarquées. Pour ces différentes expériences, le système se résume à un domaine superviseur, c'est-à-dire à un noyau. Plus précisément, nous avons implanté un noyau pour un routeur de réseau actif, un noyau pour une machine virtuelle Java et un noyau pour exécuter le jeu Doom sur une machine nue.

Tous ces noyaux sont construits par la composition de composants de la bibliothèque KORTEX. L'intérêt des différents systèmes est de prouver d'une part l'adéquation de l'architecture en termes de flexibilité et d'autre part de tester en termes d'efficacité et de fiabilité les différents composants de la bibliothèque. Les trois applications développées en utilisant KORTEX permettent de tester le réseau, le processeur et la mémoire et finalement le graphique.

6.3.1 PlanP

PlanP [Thibault et al. 1998] est un langage pour programmer les routeurs de réseaux actifs. Le prototype a été initialement développé comme un module noyau du système d'exploitation Solaris. Le langage PlanP permet d'exprimer précisément des protocoles dans un langage de haut niveau. L'environnement d'exécution reste néanmoins efficace grâce à l'utilisation d'un JIT « Just-In-Time compiler ». Les programmes PlanP sont quelque peu plus lents que des implantations comparables codées en C. Mais les applications faisant une utilisation intensive du réseau, tel un pont Ethernet intelligent, obtiennent le même débit dans un programme PlanP que dans un programme C équivalent. Cela suggère que Solaris doit être un goulot d'étranglement.

À la vue de ce que nous venons de voir, l'application PlanP constitue une bonne application permettant de prouver d'une part l'utilité de concevoir des noyaux dédiés à une application et d'autre part la viabilité de l'approche THINK ainsi que de son implantation KORTEX. Cela sous réserve d'accroître les performances, c'est ce que nous allons voir maintenant.

Système dédié

Le système dédié, à base de composants KORTEX, pour PlanP est minimal. Lorsqu'un paquet est reçu par le composant pilote réseau, il est mis dans une file d'attente par un composant implantant l'interface `netif`. Le système utilise un fil d'exécution unique dont le travail est de prendre les paquets en attentes et d'appeler directement le programme PlanP compilé par le JIT. Ces programmes

peuvent accéder au composant réseau pour envoyer des paquets. Le système dédié ne requiert par exemple pas d'ordonnanceur et s'exécute sur une mémoire plate. L'outil de visualisation donne le graphe de dépendance donné à la figure 6.8 et montre l'ensemble de composants KORTEX utilisés à l'exception du composant `trap`.

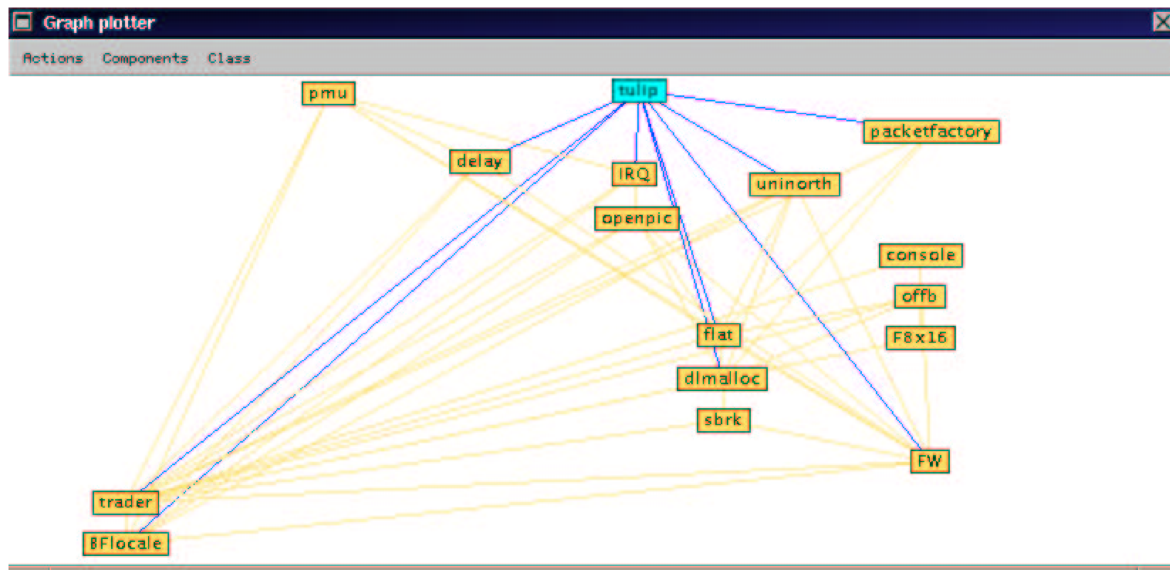


FIGURE 6.8 – Graphe de composition du système dédié à PlanP

Comparaison des performances

Pour comparer les performances, nous utilisons un programme écrit en PlanP réalisant un pont Ethernet intelligent. Le rôle de ce programme, nommé `plearn`, est d'apprendre les machines accessibles par les différentes cartes connectées à des réseaux distincts. Ainsi, lorsqu'un paquet est capté sur un réseau par le routeur, il est uniquement retransmis sur le réseau sur lequel se trouve la machine cible.

L'environnement d'expérimentation mis en œuvre consiste à exécuter `plearn` sur un routeur, c'est-à-dire une machine avec plusieurs cartes réseaux. Deux stations de travail sont directement connectées à ce routeur via des câbles croisés. L'expérience consiste à mesurer avec `ttcp` le débit obtenu entre ces deux machines. Nous avons réalisé trois expériences. La première en connectant directement les stations, c'est-à-dire sans routeur, la mesure obtenue sert de référence. La deuxième en exécutant `plearn` avec Solaris sur une Sparc Ultra 1 à 170 à 166 MHz doté de cartes réseaux à 100 Mbits. La dernière en exécutant `plearn` avec un système dédié construit en utilisant KORTEX sur un POWERPC G4 à 350 MHz doté de cartes réseaux Tulip Asanté Fast PCI à 100 Mbits. Le tableau 6.6 montre les performances de débit et de temps de calcul `plearn` pour le traitement d'un paquet. Comme nous pouvons le voir la construction d'un système dédié à partir de KORTEX double le débit réseau en atteignant un débit de 87.6 Mbits avec KORTEX contre seulement 42.0 Mbits avec Solaris.

routeur	débit	exécution plearn
aucun	91,6 Mbits	/
PlanP/Solaris, Sparc	42,0 Mbits	6 μs
PlanP/KORTEX, G4	87,6 Mbits	3 μs

TABLEAU 6.6 – Performance de PlanP sur Solaris et sur KORTEX

Estimation de la puissance de traitement

Malheureusement, comme les machines s'exécutent à des fréquences de processeurs différentes, il est difficile d'affirmer que le gain est dû à la flexibilité de THINK. Nous avons donc réalisé sur KORTEX les mêmes tests en ajoutant un délai pour chaque paquet. Cela revient à allonger la durée de traitement d'un paquet et donc à diminuer la fréquence du processeur. L'estimation de la fréquence est possible car nous savons que, sur un POWERPC G4 à 350 MHz, le temps de traitement d'un paquet est de 19.14 microsecondes. Cette valeur a été obtenue en chronométrant de pilote inclus à pilote inclus le temps de traitement logiciel pour un paquet. Cette valeur comprend les 3 microsecondes de calcul de `plearn`.

La figure 6.9 donne le débit obtenu sur KORTEX en faisant varier la fréquence du POWERPC G4 et montre à titre indicatif le débit sur Solaris à fréquence de processeur Sparc fixe de 166 MHz. Les labels ajoutés donnent le temps de traitement `plearn` pour un paquet. Sur Solaris comme la fréquence est fixe, le temps est toujours de 6 microsecondes. Ainsi, lorsque le temps de traitement sur POWERPC est de 6 microsecondes, la fréquence de processeur estimée est de 175 MHz et le débit n'est pas diminué et reste de 87.6 Mbits. Cela prouve indéniablement que le ralentissement sur Sparc est uniquement dû à Solaris et que le gain obtenu est dû au système dédié et non à la vitesse du processeur.

La courbe montre aussi que même en diminuant la fréquence du processeur à 80 MHz, le temps de traitement sur POWERPC est de 12.5 microsecondes et le débit n'est pas diminué. De plus, il faut diminuer la fréquence du processeur à 45 MHz pour descendre au même débit que sur Solaris. À ce point, le temps de traitement sur POWERPC est de 26 microsecondes, soit 4 fois plus qu'avec Solaris.

6.3.2 Kaffe

Kaffe [Kaffe] est une machine virtuelle Java Open source conforme aux spécifications de l'environnement Java. Le JDK fournit par cette machine est le 1.1.x standard. Kaffe inclut tous les mécanismes d'une machine virtuelle : l'interpréteur, le ramasse miette, les fils d'exécution, le chargement dynamique des classes, etc.

Système dédié

Cette machine virtuelle est conçue en gardant à l'esprit la portabilité. Ainsi, la machine est découpée en plusieurs sous systèmes ; la multiprogrammation, la gestion de la mémoire, les méthodes Java natives et l'interfaçage avec le système sous-jacent. Le portage de Kaffe consiste à coupler toutes les dépendances avec le système sur des composants KORTEX. Par exemple, les fils d'exécution Java sont implantés en utilisant un des ordonnanceurs proposés par KORTEX. Ainsi, l'ordonnanceur préemptif

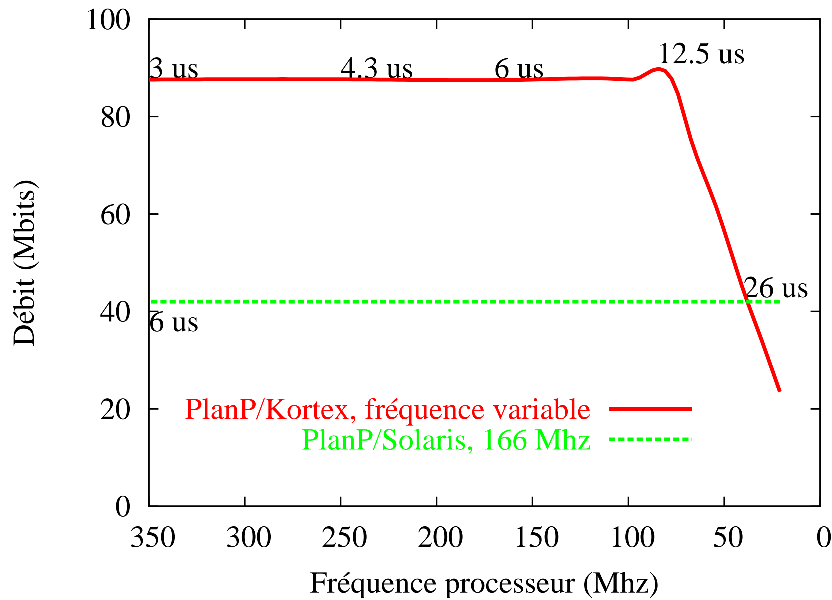


FIGURE 6.9 – Débit sur KORTEX en fonction de la fréquence du processeur

à priorité et l'ordonnanceur coopératif ont été utilisées. De plus, le traitement des exceptions dues à l'utilisation de référence nulle est directement enregistré sur le composant POWERPC `trap`. Comme la gestion du disque n'est actuellement pas implantée dans KORTEX, les fichiers contenant les classes Java et les méthodes natives du JDK et de l'application sont lus par NFS.

Le temps passé au portage de Kaffe a principalement été passé à l'adaptation des méthodes natives. Grâce à l'utilisation de la bibliothèque KORTEX, l'implantation de Kaffe a pris seulement une semaine. Le portage est actuellement suffisamment complet pour exécuter des applications complexes comme le proxy Web JigSaw développé par le consortium W3C [JigSaw].

L'exécutable du système dédié comprenant les composants KORTEX requis et la machine virtuelle Kaffe a une taille d'environ 460 Ko. En exécutant des applications Java standard avec des besoins en termes de mémoire faible, l'empreinte mémoire est de 125 Ko pour les composants KORTEX, 475 Ko pour la machine virtuelle Kaffe et 1 Mo pour le « bytecode », les méthodes natives et les données allouées dynamiquement. Ainsi, avec un peu plus de 1.6 Mo de mémoire principale, il est possible d'exécuter des applications Java. Ce qui, compte tenu du fait qu'il s'agit d'un JDK standard, est vraiment très peu. En comparaison, la machine virtuelle Java KVM [KVM] destinée à l'embarqué et dotée d'un JDK minimal « Java 2 Micro Edition » spécialisée pour l'embarquée, requiert environ 512 Ko de mémoire, sans compter le système d'exploitation.

Comparaison des performances

L'expérimentation est réalisée sur POWERMACINTOSH G4 à 500 MHz avec 128 Mo de mémoire. Nous comparons les résultats avec la même version de Kaffe s'exécutant sur LinuxPPC 2.4.x. Les performances mesurées sont données au tableau 6.7. L'expérimentation consiste d'une part à faire des opérations de synchronisation avec un fil d'exécution, et d'autre part à lancer deux fils d'exécu-

tion qui se passent successivement la main via `Thread.yield()`. Les fils d'exécution Kaffe sont directement implantés soit en utilisant l'ordonnanceur préemptif à priorité et on obtient des fils d'exécution similaires aux « native-threads », soit en utilisant l'ordonnanceur coopératif et on obtient des fils d'exécution similaires aux « java-threads ».

« **Benchmark** »	**Kaffe/Linux** (java-thread)	**Kaffe/KORTEX** (java-thread)	**Kaffe/KORTEX** (native-thread)
`synchronized(o) {}`	0,527 μs	0,363 μs	0,363 μs
`try {} catch(...) {}`	1,790 μs	1,585 μs	1,594 μs
`try {null.x()} catch(...) {}`	12,031 μs	5,094 μs	5,059 μs
`try {throw} catch(...) {}`	3,441 μs	2,448 μs	2,434 μs
`Thread.yield()`	6,960 μs	6,042 μs	6,258 μs

TABLEAU 6.7 – Performance de Kaffe sur KORTEX

Les mesures obtenues sont comparées avec celles obtenues sur Kaffe s'exécutant sur Linux et utilisant des « java-threads » basés comme notre implantation sur le co-routinage. Les tests montrent qu'il est possible d'améliorer grandement la gestion des exceptions, en particulier, pour les pointeurs nuls qui sont directement traités par le composant `trap`. Les résultats montrent aussi que quels que soient les autres tests, l'exécution sur KORTEX améliore les performances, la différence nous donne donc le coût d'exécution pur de Linux. Comme nous pouvons aussi le constater, la différence entre les « native-threads » et les « java-threads » de Kaffe sur KORTEX n'est pas significative. Comme les « native-threads » ne sont pas supportés par Kaffe sur Linux, il n'est pas possible de comparer cette gestion des fils d'exécution avec les mesures sur KORTEX. Il faut noter que ces coûts comprennent le temps d'interprétation du « bytecode » qui est non négligeable. Le ratio des coûts systèmes est donc beaucoup plus important qu'il n'y parait ici.

Jigsaw

Pour évaluer globalement le portage de Kaffe sur KORTEX, nous évaluons les performances obtenues avec une application Java. Nous avons choisi le proxy Web JigSaw, développé par le consortium W3C [JigSaw], car il a des exigences systèmes fortes. Par exemple, il lance de nombreux fils d'exécution et il fait une utilisation intensive du réseau. La même exécution de JigSaw sur Kaffe sur Linux compare les performances avec notre implantation.

Le tableau 6.10 donne le temps de transfert en fonction de la taille d'un fichier mémorisé dans le cache. Comme nous pouvons le voir, l'exécution du proxy sur une machine nue offre de meilleurs résultats que son exécution sur un système d'exploitation. L'exécution de JigSaw sur KORTEX permet de diviser par deux les temps de transfert pour les fichiers de taille moyenne qui doivent être découpés en plusieurs paquets. Pour les fichiers de petite taille ou de très grande taille, le débit est limité par les performances du réseau. Nous voyons encore une fois l'intérêt de concevoir des systèmes dédiés à des applications spécialisées.

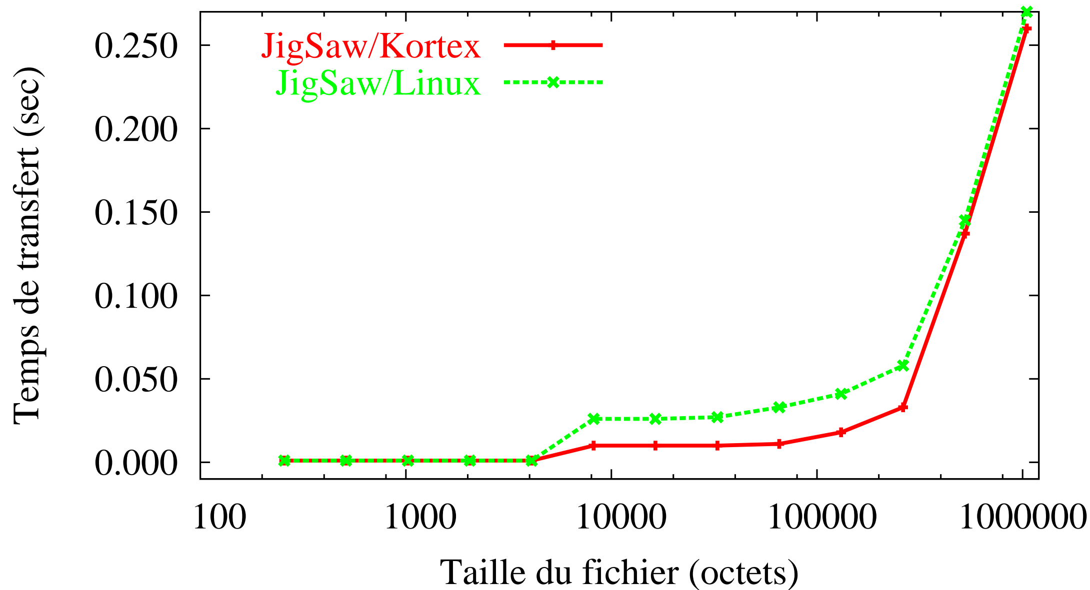


FIGURE 6.10 – Performance de JigSaw sur Kaffe sur KORTEX

6.3.3 Doom

Une expérimentation intéressante est de construire un système dédié permettant l'exécution d'un jeu vidéo. Pour cela, nous avons porté Doom, version LxDoom [LxDoom], en utilisant KORTEX. Le portage a pris deux jours qui ont principalement été consacrés à la compréhension du contrôleur graphique. Ce temps de portage montre que l'approche de construction de système par composants est viable et permet de développer facilement et rapidement des systèmes.

Système dédié

La gestion du disque n'est actuellement pas proposée par KORTEX, le fichier contenant le scénario du jeu est donc lu par NFS. Cela nécessite de construire un système intégrant une pile de protocole et un ordonnanceur, même si ces derniers ne sont pas utilisés par Doom. Nous stoppons donc l'ordonnanceur après chargement du fichier. Nous utilisons le composant implantant la mémoire plate comme gestionnaire mémoire. La taille de l'exécutable noyau résultant est de 527 Ko. L'empreinte mémoire requise pour l'exécution est : 105 Ko pour les composants KORTEX, 900 Ko pour le moteur de Doom et 5 Mo pour le scénario du jeu (`doom.wad`).

Comparaison des performances

L'expérimentation est réalisée sur POWERMACINTOSH G4 à 500 MHz avec 128 Mo de mémoire. Nous comparons les résultats avec la même version de Doom s'exécutant sur LinuxPPC 2.4.x en mode utilisateur unique (« single user ») et dessinant directement dans la mémoire du « frame-buffer ». Les performances mesurées pour Doom sont données au tableau 6.8. Comme nous pouvons le constater,

l'exécution de Doom sur KORTEX est entre 3 % et 6 % plus rapide que sur Linux.

Résolution graphique	Doom/KORTEX (mémoire plate)	Doom/KORTEX (mémoire paginée)	Doom/Linux
320x200	1955	1914	1894
640x480	491	485	483
1024x768	177	171	167

TABLEAU 6.8 – Performance de Doom en images par secondes

Comme durant le test il n'y a aucun appel système et que le jeu effectue uniquement des calculs numériques et des recopies mémoires, la différence de performance est due à l'activité des démons Linux résidents et à l'utilisation de la mémoire plate qui n'utilise pas la MMU du processeur. Pour vérifier ce coût, nous avons réalisé la même application en changeant simplement le composant de gestion de la mémoire par le composant de même interface et offrant la mémoire paginée. Comme nous pouvons le constater, le coût d'utilisation de la MMU est d'environ 2% sur le temps d'exécution globale. Ce scénario illustre parfaitement les bénéfices potentiels de l'utilisation de l'approche THINK dans la construction rapide de système d'exploitation dédié et optimisé pour une utilisation particulière.

6.4 Conclusion

6.4.1 Évaluation qualitative

L'évaluation qualitative de l'architecture THINK et de son implantation KORTEX doit montrer que l'approche satisfait à deux objectifs. Le premier objectif est la possibilité d'implanter au plus bas niveau des fonctions classiquement offertes par les intergiciels, c'est-à-dire d'une part faire bénéficier le système des mécanismes de liaison et d'autre part permettre à l'intergiciel de manipuler les ressources matérielles. Le deuxième objectif est la possibilité de parcourir l'espace de conception vu à la figure 2.26 de la page 47, c'est-à-dire offrir un modèle suffisamment flexible pour la composition de systèmes arbitraires. Les expérimentations montrent que ces deux objectifs sont effectivement satisfaits et que les principes d'organisation proposés sont adéquats. Plusieurs raisons permettent d'affirmer cela.

- La variété des noyaux de systèmes d'exploitation construits et la diversité des systèmes dédiés à des applications montrent la possibilité de parcourir tout l'espace de conception à partir de la même bibliothèque de composants. Cela démontre la flexibilité de l'architecture qui repose sur trois notions ;
  - La transparence d'exécution vis-à-vis du domaine superviseur ou applicatif permet d'utiliser un même composant quelles que soient les contraintes d'isolation.
  - La liaison flexible qui permet les interactions entre composants quels que soient leurs domaines d'exécution ; local, superviseur, applicatif ou distant.
  - La notion de composants qui permet la composition arbitraire de systèmes.

  Nous voyons ici tout l'intérêt d'utiliser le concept des liaisons flexibles dans les systèmes d'exploitation et pas uniquement au niveau applicatif comme c'est le cas dans les intergiciels.

- La rapidité de mise en œuvre de systèmes dédiés à des applications spécialisées montre la pertinence de la bibliothèque KORTEX. Cette rapidité est due d'une part à la composition de composants et d'autre part aux outils de développement qui effectuent automatiquement des tâches fastidieuses. Cette rapidité est primordiale, car le temps de développement est souvent un frein au développement de tels systèmes.

- Le plein et fin contrôle des ressources matérielles réifiées par des composants permet la manipulation de ces ressources directement par les applications. Cela permet l'implantation de politique de gestion arbitraire. Cette construction est possible uniquement grâce à l'utilisation au plus bas niveau du concept de liaison flexible.

- La liaison distante confère à un système des fonctionnalités de communication, il devient alors un système réparti.

6.4.2 Évaluation quantitative

L'évaluation quantitative de l'architecture THINK et de son implantation KORTEX montre que l'approche est viable et que le modèle n'implique pas un coût inacceptable. Plusieurs raisons permettent d'affirmer cela.

- Les différents « benchmarks » montrent que l'implantation du concept de liaison flexible ajoute un coût négligeable à l'exécution. Comme nous l'avons vu, ce coût est uniquement de quelques cycles, grâce à l'implantation native faite du concept d'interface.

- La flexibilité offerte par le concept liaison flexible permet la conception de liaisons systèmes spécialisées en fonction des contraintes sous-jacentes. Ainsi, ces liaisons systèmes offrent des résultats similaires à ceux de la littérature scientifique.

- Les techniques de flexibilité et d'adaptation offertes par le modèle permettent de spécialiser les noyaux. Ainsi, les systèmes dédiés à une application donnée offrent des performances d'exécution bien meilleures dans certains domaines comparés à l'utilisation de noyaux classiques. De plus, l'empreinte mémoire de ces systèmes est suffisamment petite pour envisager leur utilisation dans des environnements fortement contraints.

Chapitre 7

Conclusion générale

7.1 Bilan

Dans cette thèse, nous avons montré qu'il était possible de capturer sous la forme de canevas logiciels des concepts et des principes d'architecture rencontrés dans les systèmes d'exploitation centralisés ou répartis. Ces concepts sont au nombre de trois ; des composants, des liaisons flexibles modélisant les interactions, et des domaines modélisant l'isolation. Cette approche autorise la construction de noyaux d'infrastructure variés, susceptibles d'être mis en place, statiquement ou dynamiquement, depuis des systèmes dédiés à une application jusqu'aux systèmes monolithiques classiques en passant par les différentes formes de micronoyaux.

Pour valider ce modèle d'architecture, nommé THINK, nous avons implanté sur ce modèle une bibliothèque, nommée KORTEX et destinée aux machines POWERMACINTOSH. Cette bibliothèque propose un ensemble de composants systèmes, offrant d'une part des services du plus bas niveau comme les pilotes de périphériques, et d'autre part des services de haut niveau comme la gestion des processus et des connexions réseaux. La composition arbitraire de ces composants construit le noyau d'infrastructure désiré. La diversité des noyaux d'infrastructure construits à partir de cette bibliothèque, la rapidité et la simplicité de programmation offertes par les outils de développement, prouvent l'adéquation du modèle.

Les évaluations quantitatives du modèle d'architecture THINK et de la bibliothèque KORTEX de composants systèmes démontrent la viabilité de l'approche proposée. Les implantations du concept de liaison flexible n'engendrent pas un coût inacceptable et elles permettent même d'obtenir des performances similaires à celles de la littérature. De plus, les bénéfices apportés par la structure flexible peuvent potentiellement être importants. Tout d'abord, les évaluations montrent un accroissement des performances d'exécution sur des noyaux de systèmes classiques. Ensuite, les besoins en ressources matérielles requis pour l'exécution des systèmes sont grandement diminués, ce qui offre des perspectives d'utilisation dans les environnements fortement contraints.

7.2 Comparaison avec l'état de l'art

La flexibilité limitée des micronoyaux, comme Mach [Accetta et al. 1986] ou Chorus [Rozier et al. 1988], n'est pas uniquement due à des problèmes de performances partiellement résolus dans

les micronoyaux récents, comme L3 [Liedtke 1993]. En effet, la granularité est aussi limitée au niveau de l'espace d'adressage, alors que le modèle d'architecture THINK permet une granularité au niveau du composant seul. Cette limitation, intrinsèque aux micronoyaux, subsiste dans le micronoyau plus récent Pebble [Gabber et al. 1999] même s'il permet de composer plusieurs composants dans un même domaine de protection. De plus, même si certaines constructions de systèmes ne nécessitent pas ou sont incompatibles avec la protection offerte par la mémoire virtuelle, celle-ci ne peut pas être supprimée dans les micronoyaux, aussi petits soient-ils, comme L4. La philosophie de THINK, au contraire, est de ne pas la mettre par défaut et de l'ajouter si besoin est.

Même si la philosophie de l'exonoyau est de supprimer toutes les abstractions du noyau [Engler et Kaashoek 1995, Engler et al. 1995], il en subsiste quelques-unes. C'est par exemple le cas pour l'allocation du processeur, basée sur un quantum de temps, qui est directement contrôlée par le noyau et qui ne peut être manipulée par ailleurs. Par contraste, THINK permet une application stricte de la philosophie et offre uniquement des mécanismes permettant de construire des mécanismes d'ordonnancement. C'est d'ailleurs ce qui est fait dans la bibliothèque KORTEX.

Les composants processeurs de la bibliothèque KORTEX se rapprochent, au niveau des interfaces, du nanonoyau, qui est en fait une couche HAL, proposé dans le micronoyau à objets μChoices [Tan et al. 1995]. Mais elle diffère radicalement dans le sens ou même le nanonoyau est vu comme un composant dans l'implantation. Il n'y a donc pas d'entorse au modèle qui est appliqué partout, et tout le système est construit par la composition d'un ensemble de composants.

Spin [Bershad et al. 1995] est un noyau extensible de façon sûre grâce à l'utilisation du Modula-3 offrant des propriétés de sûreté, il offre donc un modèle de protection des extensions que nous n'avons pas. La structure du noyau est classique et les applications sont écrites dans un langage quelconque. Ici, les services sont des objets Modula-3 qui ne sont pas très éloignés de nos composants, mais sans le support de plusieurs interfaces par composant. Néanmoins, Spin n'offre aucun modèle de liaison, d'ailleurs, aucun autre système n'offre un tel modèle.

Nemesis [Reed et Fairbairns 1997] est un système à espace d'adressage unique destiné aux applications multimédias. Nemesis offre un modèle de composant et de liaison similaire à ceux de THINK. Les composants ne peuvent exporter qu'une seule interface, mais leur implantation est similaire à notre descripteur d'interface. Néanmoins, le modèle de liaison de Nemesis n'est pas aussi général que celui de THINK. Nemesis permet la définition de différentes formes de liaison entre des composants s'exécutant dans des domaines différents, alors que THINK permet la définition de différentes formes de liaison entre des composants arbitraires, y compris les composants du plus bas niveau s'exécutant dans le noyau.

OSKit [Ford et al. 1997] a montré comment construire une bibliothèque de services systèmes pour construire différents noyaux de système d'exploitation, et plus récemment Knit [Reid et al. 2000] a proposé un modèle de composition utilisable avec OSKit. La bibliothèque KORTEX est donc comparable à OSKit et les outils THINK sont comparables à Knit. Mais la granularité de leur approche laisse à désirer et la composition est uniquement statique. Contrairement à OSKit, notre implantation repose sur un modèle lui conférant certains avantages indéniables comme la flexibilité. Par exemple, les composants OSKit de gestion de la mémoire et des fils d'exécution ne peuvent être remplacés, ni statiquement ni dynamiquement. En conséquence, l'approche OSKit est difficilement exploitable pour la construction de petit système. De plus, les composants sont limités au noyau et ne peuvent être utilisés dans une approche d'un système complet, alors que les outils et les composants THINK peuvent être utilisés dans les applications.

Notre approche en ce qui concerne les composants se rapproche de celle faite par MMLite [Helan-

der et Forin 1998]. Leur architecture est basée sur le modèle de composant binaire COM de Microsoft pour la construction de noyau de système. Contrairement à nous qui n'avons que les éléments pour une reconfiguration, MMLite offre des mécanismes pour le remplacement dynamique de composant. Mais les mécanismes qu'ils proposent ne sont pas automatiques et le respect des contraintes d'intégrité est à la charge du programmeur. Comme le nôtre, leur modèle permet l'exécution des composants quels que soient le domaine (superviseur ou applicatif), et le modèle de mémoire sous-jacent (paginée ou plate). En revanche, notre modèle de composant est plus léger dans le sens où, contrairement à nous, leur approche laisse penser qu'elle nécessite un fil d'exécution par composant du fait de la présence d'une pile par composant. De plus, leur architecture ne propose aucun modèle de liaison.

Le concept de liaisons flexible du modèle THINK permet d'optimiser les chemins critiques en appliquant toutes les techniques développées pour accroître les performances des interactions, avec les IPC [Liedtke 1993], les LRPC [Bershad et al. 1989] ou des techniques de spécialisation [Pu et al. 1995]. En fait, c'est là l'un des principaux intérêts du concept de liaison flexible. En outre, le concept ajoute un degré supplémentaire en offrant un modèle de programmation uniforme, vis-à-vis de la localisation des composants, basé sur la notion d'interface.

Le système Scout [Mosberger et Peterson 1996] et son prédécesseur x-Kernel [Hutchinson et Peterson 1991] consistent en un ensemble de composants pour développer des applications pour les réseaux ou des protocoles. Click [Morris et al. 1999] fournit des composants dédiés au développement d'un seul type de système destiné aux routeurs. Ces systèmes offrent un modèle pour optimiser le résultat de la composition des composants. Contrairement à la liaison flexible, la liaison est ici une abstraction pour le transfert de flux de paquets, nommée « path » dans Scout, elle n'est donc pas utilisée dans tout le système et la sémantique est plus contrainte. La liaison flexible pourrait aussi être utilisée pour construire de tels systèmes.

L'architecture THINK est inspirée par différents travaux sur les intergiciels répartis, comme le modèle de référence pour le traitement réparti ouvert (RM-ODP) [ISO/IEC 1995b]. THINK applique le modèle de liaison flexible défini dans ces travaux pour la construction de noyau de système d'exploitation et ne se limite pas au niveau des applications.

7.3 Limites

Bien évidemment, le travail réalisé dans cette thèse n'est pas complet. Plusieurs travaux restent à mener pour la construction d'une infrastructure répartie pleinement adaptable et reconfigurable dynamiquement.

Tout d'abord, les problèmes liés à la sécurité du système, des applications et des composants contre les fautes accidentelles ou intentionnelles n'ont pas été traités. Les liaisons n'offrent aucun mécanisme de protection permettant de contrôler les interactions afin de garantir l'intégrité du système et des données manipulées. Seuls les mécanismes d'isolation des mémoires sont mis en œuvre.

Deuxièmement, le canevas logiciel associé au concept de domaine n'est que très sommairement défini. Ce canevas n'a donc pas été affiné, ni expérimenté et évalué dans la pratique. Par exemple, aucun mécanisme de reconfiguration dynamique du système n'est proposé dans le prototype. Cette reconfiguration suppose d'une part la possibilité de modifier ou de supprimer dynamiquement les liaisons entre les composants et d'autre part la possibilité de décharger les composants.

Finalement, un manque important dans le modèle d'architecture est un canevas logiciel pour la ré-

flexion structurelle, c'est-à-dire la possibilité de consulter l'état interne des composants. Cette notion est cruciale lorsque l'on s'intéresse à l'administration des systèmes. Cette réflexion est partiellement définie pour les domaines, mais il est nécessaire de l'étendre pour les composants primaires, par exemple de façon similaire à la réflexion proposée dans Java.

7.4 Perspectives

Il reste aujourd'hui des travaux à effectuer pour s'affranchir des limitations décrites plus haut.

En particulier, il faudrait mettre en place des mécanismes pour le déchargement ou le remplacement de composants. Cette opération est complexe et nécessite le respect de propriétés d'intégrité, comme ne pas décharger un composant en cours d'utilisation sous peine de causer des défaillances. Une solution envisageable est de geler les composants. La définition d'une liaison spécialisée permettant de geler les composants en bloquant les interactions pourrait être une bonne approche.

Il serait intéressant d'enrichir la bibliothèque de composants KORTEX avec les services systèmes manquants, comme par exemple la gestion des disques. Une expérience intéressante serait alors de refaire, à partir de la composition de composants, un système à la Linux offrant des possibilités d'extensibilité et de fiabilité accrues. Sous réserve de fournir la même interface système, le noyau résultant de l'expérimentation pourrait se substituer au noyau Linux standard. Cela permettrait d'évaluer de façon définitive l'impact des liaisons et des composants sur les performances du système.

La flexibilité et les performances offertes par l'architecture THINK permettent d'envisager une multitude d'utilisations.

Nous pouvons envisager d'utiliser l'architecture THINK pour la construction de noyaux sûrs. Il s'agirait notamment d'exploiter systématiquement la notion de domaine pour la construction de zones de confinement logicielles à des fins de protection et de tolérances aux fautes. Il serait intéressant d'appliquer les techniques de « sandbox », utilisées dans Vino [Small et Seltzer 1994], aux liaisons flexibles et au chargeur dynamique de code binaire afin de contrôler les accès effectués par un composant.

Il serait aussi intéressant de programmer les composants dans un langage offrant des propriétés de sûreté, comme Java qui offre en plus la notion d'interface. L'idée ici serait d'utiliser ou de concevoir un compilateur Java générant du code binaire natif, comme par exemple Harissa [Muller et al. 1997]. Le résultat pourrait alors être comparé en terme de performance et de taille mémoire avec l'approche en Modula-3 de Spin [Bershad et al. 1995].

Même si l'utilisation d'interfaces clairement identifiées simplifie la programmation système, elle reste complexe. L'utilisation de langages dédiés « Domain-specific languages » (DSL) pour la programmation des pilotes [Mérillon et al. 2000], des protocoles [Thibault et al. 1998] ou encore des ordonnanceurs permettrait d'imposer des règles de programmation et de contrôler l'utilisation des interfaces. Cela devrait permettre de baisser le niveau d'expertise requis pour le développement des systèmes. Cette approche pourrait donc être un bon complément, à l'architecture THINK, pour la conception des composants systèmes.

Nous pouvons aussi envisager d'utiliser l'architecture THINK et la bibliothèque KORTEX pour la construction de noyaux temps réel. Il s'agirait en particulier de construire un noyau pleinement déterministe et d'y implanter une machine d'exécution minimale pour langages synchrones tels que Esterel. Un tel noyau laisse supposer des gains de performances sur certaines applications en sup-

primant le coût des changements de contexte dû aux interruptions. Par exemple les routeurs dont le comportement est déterministe (réception, traitement, envoi) se prêteraient bien à cette approche.

L'approche THINK pourrait être exploitée dans des environnements matériels différents. Il s'agirait par exemple de disposer de noyaux d'infrastructure pour d'autres machines et dans des environnements contraints comme les assistants personnels ou les téléphones portables. Cette expérimentation montrerait pleinement la portabilité fournie par l'architecture. Pour faire comme tout le monde, il serait aussi nécessaire de porter notre architecture sur les processeurs Intel, tout en sachant que ce portage sera au détriment de la flexibilité.

À l'opposé, l'approche THINK pourrait être exploitée dans des environnements répartis. Il s'agirait par exemple de construire un noyau d'infrastructure pour grappes de stations de travail afin d'exécuter des applications parallèles hautes performances telles que les simulations météorologiques. L'intérêt ici serait d'exploiter le modèle de programmation uniforme pour la construction d'application répartie tout en offrant des communications distantes très efficaces.

Références bibliographiques

Accetta et al. 1986
Mike J. ACCETTA, Robert V. BARON, David B. GOLUB, Richard F. RASHID, Avadis TEVANIAN, JR., et Michael W. YOUNG. « Mach : A New Kernel Foundation for UNIX Development ». Dans *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, juin 1986.

Apple Computer Inc 1995
APPLE COMPUTER INC. *Macintosh Technology in the Common Hardware Reference Platform*. Morgan Kaufmann Publishers Inc, 1995.

Apple Computer Inc et al. 1995
APPLE COMPUTER INC, INTERNATIONAL BUSINESS MACHINES CORPORATION, et MOTOROLA INC. *PowerPC Microprocessor Common Hardware Reference Platform : A System Architecture*. Morgan Kaufmann Publishers Inc, novembre 1995.

Balter et al. 1991
Roland BALTER, Jean-Pierre BANÂTRE, et Sacha KRAKOWIAK. *Construction des systèmes d'exploitation répartis*. INRIA, Rocquencourt, juillet 1991.

Bershad et al. 1989
Brian N. BERSHAD, Thomas E. ANDERSON, Edward D. LAZOWSKA, et Henry M. LEVY. « Lightweight remote procedure call ». Dans *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP'1989)*, pages 102–113, décembre 1989.

Bershad et al. 1993
Brian N. BERSHAD, Matthew J. ZEKAUSKAS, et Wayne A. SAWDON. « The Midway Distributed Shared Memory System ». Dans *Proceedings of the 38th IEEE Computer Conference*, pages 528–537, février 1993.

Bershad et al. 1995
Brian N. BERSHAD, Stefan SAVAGE, Przemyslaw PARDYAK, Emin G. SIRER, Marc E. FIUCZYNSKI, David BECKER, Craig CHAMBERS, et Susan EGGERS. « Extensibility, Safety and Performance in the SPIN Operating System ». Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'1995)*, pages 267–284, Copper Mountain Resort, Colorado, décembre 1995.

Birrell et Nelson 1984
Andrew D. BIRRELL et Bruce J. NELSON. « Implementing Remote Procedure Call ». *ACM Transactions on Computer Systems*, 2(1) : 39–59, février 1984.

Blair et Stefani 1997
Gordon S. BLAIR et Jean-Bernard STEFANI. *Open Distributed Processing and Multimedia*. Addison-Wesley, Harlow, England, 1997.

Blair et al. 1999
Gordon S. BLAIR, Fábio COSTA, Geoff COULSON, Fabien DELPIANO, Hector DURAN, Bruon DUMANT, François HORN, Nikos PARLAVANTZAS, et Jean-Bernard STEFANI. « The Design of a Resource-Aware Reflective Middleware Architecture ». Dans *Proceedings of the 2nd International Conference on Metalevel Architectures and Reflection (Reflection'1999)*, Saint-Malo, France, juillet 1999.

Brinch Hansen 1973
P. BRINCH HANSEN. *Operating System Principles*. Prentice Hall, Englewood Cliffs, 1973.

Campbell et Tan 1995
Roy H. CAMPBELL et See-Mong TAN. « μChoices : an Object-Oriented Multimedia Operating System ». Dans *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS'1995)*, pages 90–94, Orcas Island, Washington, mai 1995. IEEE Computer Society.

Campbell et al. 1993
Roy H. CAMPBELL, Nayeem ISLAM, David RAILA, et Peter MADANY. « Designing and Implementing Choices : An Object-Oriented System in C++ ». *Communications of the ACM*, 36(9) : 117–126, septembre 1993.

Chase et al. 1994
Jeffrey S. CHASE, Henry M. LEVY, Michael J. FEELEY, et Edward D. LAZOWSKA. « Sharing and Protection in a Single-Address-Space Operating System ». *ACM Transactions on Computer Systems*, 12(4) : 271–307, novembre 1994.

Dijkstra 1965a
Edsger W. DIJKSTRA. « Cooperating sequential processes ». Rapport Technique EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

Dijkstra 1965b
Edsger W. DIJKSTRA. « Solution of a Problem in Concurrent Programming Control ». *Communications of the ACM*, 8(9) : 569, septembre 1965.

Dijkstra 1968
Edsger W. DIJKSTRA. « The structure of the “THE”-multiprogramming system ». *Communications of the ACM*, 11(5) : 341–346, mai 1968.

Dumant et al. 1998
Bruno DUMANT, Frédéric DANG TRANG, François HORN, et Jean-Bernard STEFANI. « Jonathan : an Open Distributed Processing Environment in Java ». Dans *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'1998)*, septembre 1998.

eCos
REDHAT. « eCos : Embedded Configurable Operating System ». http : //sources.redhat.-com/eCos.

Engler et Kaashoek 1995
Dawson R. ENGLER et M. Frans KAASHOEK. « Exterminate all operating system abstractions ». Dans *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS'1995)*, pages 78–83, Orcas Island, Washington, mai 1995. IEEE Computer Society.

Engler et al. 1995
Dawson R. ENGLER, M. Frans KAASHOEK, et James W. O'TOOLE JR.. « Exokernel : an operating system architecture for application-level resource management ». Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'1995)*, pages 251–266, Copper Mountain Resort, Colorado, décembre 1995.

Fassino et Stefani 2001
Jean-Philippe FASSINO et Jean-Bernard STEFANI. « Think : un noyau d'infrastructure répartie adaptable ». Dans $2^{ème}$ *Conférence Française sur les Systèmes d'exploitation (CFSE'2001)*, pages 95–106, avril 2001.

Ford et al. 1997
Bryan FORD, Godmar BACK, Greg BENSON, Jay LEPREAU, Albert LIN, et Olin SHIVERS. « The Flux OSKit : A substrate for kernel and language research ». Dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'1997)*, pages 38–51, Saint-Malo, France, octobre 1997.

Gabber et al. 1999
Eran GABBER, Christopher SMALL, John BRUNO, José BRUSTOLONI, et Avi SILBERSCHATZ. « The Pebble Component-Based Operating System ». Dans *Proceedings of the USENIX Annual Technical Conference (USENIX'1999)*, pages 267–282, Berkeley, CA, juin 1999. USENIX Association.

Garlan et Shaw 1994
David GARLAN et Mary SHAW. « An Introduction to Software Architecture ». Rapport Technique CMU-CS-94-166, Carnegie Mellon University, janvier 1994.

Gharachorloo et al. 1990
Kourosh GHARACHORLOO, Daniel LENOSKI, James LAUDON, Philip GIBBONS, Anoop GUPTA, et John HENNESSY. « Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors ». Dans *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'1990)*, pages 15–26, Seattle, WA, juin 1990.

Gosling et al. 2000a
James GOSLING, Bill JOY, Guy STEELE, et Gilad BRACHA. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Massachusetts, 2000.

Gosling et al. 2000b
James GOSLING, Bill JOY, Guy STEELE, et Gilad BRACHA. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Massa-chusetts, 2000.

Govil et al. 1999
Kingshuk GOVIL, Dan TEODOSIU, Yongqiang HUANG, et Mendel ROSENBLUM. « Cellular Disco : resource management using virtual clusters on shared-memory multiprocessors ». Dans *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'1999)*, pages 154–169, Kiawah Island, SC, décembre 1999.

Hamilton et Kougiouris 1993
Graham HAMILTON et P. KOUGIOURIS. « The Spring nucleus : a microkernel for objects ». Dans *Proceedings of the USENIX Summer 1993 Technical Conference*, pages 147–159, Cincinnati, Ohio, juin 1993.

Hamilton et al. 1993
Graham HAMILTON, Michael L. POWELL, et James G. MITCHELL. « Subcontract : a flexible base for distributed programming ». Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'1993)*, pages 69–79, Asheville, décembre 1993.

Härtig et al. 1997
Hermann HÄRTIG, Michal HOHMUTH, Jochen LIEDTKE, Sebastian SCHÖNBERG, et Jean WOLTER. « The Performance of μkernel-based Systems ». Dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'1997)*, pages 66–77, Saint-Malo, France, octobre 1997.

Helander et Forin 1998
Johannes HELANDER et Alessandro FORIN. « MMLite : A Highly Componentized System Architecture ». Dans *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 96–103, Sintra, Portugal, septembre 1998.

Hennessy et Patterson 1996
John L. HENNESSY et David A. PATTERSON. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc, 2ème édition, 1996.

Hess et al. 2000
Christopher K. HESS, David RAILA, Roy H. CAMPBELL, et Dennis MICKUNAS. « Design and Performance of MPEG Video Streaming to Palmtop Computers ». Dans *Proceedings of the Multimedia Computing and Networking (MMCN'2000)*, pages 39–59, San Jose, Canada, janvier 2000.

Hildebrand 1992
Dan HILDEBRAND. « An Architectural Overview of QNX ». Dans *Proceedings of the USENIX Workshop on Micro-Kernels and other Kernel Architectures*, pages 113–126, Seattle, Washington, avril 1992.

Hoare 1974
C. A. R. HOARE. « Monitors : An Operating System Structuring Concept ». *Communications of the ACM*, 17(10) : 549–557, octobre 1974.

Hohmuth 1999
Michael HOHMUTH. « The Fiasco Kernel : System Architecture ». Rapport Technique, Dresden University of Technology, 1999. http : //os.inf.tu-dresden.de/fiasco/status.html.

Hutchinson et Peterson 1991
Norman C. HUTCHINSON et Larry L. PETERSON. « The *x*-Kernel : An Architecture for Implementing Network Protocols ». *IEEE Transactions on Software Engineering*, 17(1) : 64–76, janvier 1991.

Intel Corporation 1995
INTEL CORPORATION. « *Pentium processor family developer's manual* », 1995.

ISO/IEC 1995a
ISO/IEC. « Open Distributed Processing - Reference Model, Part 2 : Fundations ». ITU-T Recommendation X.902 International Standard 10746-2, ISO/IEC, 1995.

ISO/IEC 1995b
ISO/IEC. « Open Distributed Processing - Reference Model, Part 3 : Architecture ». ITU-T Recommendation X.903 International Standard 10746-3, ISO/IEC, 1995.

ISO/IEC 1998
ISO/IEC. « Open Distributed Processing - Reference Model, Part 1 : Overview ». ITU-T Recommendation X.901 International Standard 10746-1, ISO/IEC, 1998.

JigSaw
WORLD WIDE WEB CONSORTIUM. « JigSaw ». http : //www.w3.org/Jigsaw/.

Kaashoek et al. 1997
M. Frans KAASHOEK, David R. ENGLER, Gregory R. GANGER, Héctor M. BRICEÑO, Russell HUNT, David MAZIÈRES, Thomas PINCKNEY, Robert GRIMM, John JANNOTTI, et Kenneth MACKENZIE. « Application Performance and Flexibility on Exokernel Systems ». Dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'1997)*, pages 52–65, Saint-Malo, France, octobre 1997.

Kaffe
TRANSVIRTUAL TECHNOLOGIES INC. « Kaffe : A free virtual machine to run Java code ». http : //www.transvirtual.org/.

Knuth 1973
Donald E. KNUTH. *The Art of Computer Programming : Fundamental Algorithms*, volume I. Addison-Wesley, $2^{ème}$ édition, 1973.

Kon et al. 1999
Fabio KON, Roy H. CAMPBELL, M.D. MICKUNAS, et Klara NAHRSTEDT. « 2K : a Distributed Operating System for Heterogeneous Environments ». Rapport Technique 2132, University of Illinois, décembre 1999.

Krakowiak 1985
Sacha KRAKOWIAK. *Principes des systèmes d'exploitation des ordinateurs*. Dunod informatique, Paris, 1985.

Krakowiak 2001
Sacha KRAKOWIAK. « Architecture des systèmes, passé et avenir ». Conférence invitée CFSE-REMPAR-SYMPA, avril 2001.

KVM
SUN MICROSYSTEMS INC. « K Virtual Machine ». http : //java.sun.com/products/cldc/.

Lamport 1974
Leslie LAMPORT. « A New Solution of Dijkstra's Concurrent Programming Program ». *Communications of the ACM*, 17(8) : 453–455, août 1974.

Lamport 1986
Leslie LAMPORT. « The Mutual Exclusion Problem ». *Journal of the ACM*, 33(2) : 313–348, avril 1986.

Lamport 1991
Leslie LAMPORT. « The Mutual Exclusion Problem has been Solved ». *Communications of the ACM*, 34(1) : 110–111, décembre 1991.

Liedtke 1993
Jochen LIEDTKE. « Improving IPC by Kernel Design ». Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'1993)*, pages 175–188, Asheville, décembre 1993.

Liedtke 1995
Jochen LIEDTKE. « On μ-Kernel Construction ». Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'1995)*, pages 237–250, Copper Mountain Resort, Colorado, décembre 1995.

Liedtke 1996a
Jochen LIEDTKE. « *L4 Reference Manual : 486, Pentium, Pentium Pro* ». GMD, 2.0 édition, septembre 1996.

Liedtke 1996b
Jochen LIEDTKE. « μ-Kernels Must And Can Be Small ». Dans *Proceedings of 5th International Workshop on Object-Orientation in Operating Systems (IWOOOS'1996)*, pages 152–155, Washington, DC, 1996.

Liedtke et al.
Jochen LIEDTKE, Kevin ELPHINSTONE, Sebastian SCHÖNBERG, et Hermann HÄRTIG.

« Achieved IPC Performance ». Dans *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS'1997)*, Chatham, Massachusetts. IEEE Computer Society.

Lindholm et Yellin 1999
Tim LINDHOLM et Frank YELLIN. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1999.

LxDoom
« LxDoom ». http : //prboom.sourceforge.net/.

Malenfant et al. 1996
Jacques MALENFANT, M. JACQUES, et F.-N. DEMERS. « A Tutorial on Behavioral Reflection and its Implementation ». Dans *Proceedings of Reflection (Reflection'1996)*, pages 1–20, San Francisco, California, avril 1996.

Mérillon et al. 2000
Fabrice MÉRILLON, Laurent RÉVEILLÈRE, Charles CONSEL, Renaud MARLET, et Gilles MULLER. « Devil : An IDL for Hardware Programming ». Dans *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'2000)*, pages 17–30, San Diego, California, octobre 2000. USENIX Association.

Meyer 1997
Bertrand MEYER. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, USA, $2^{ème}$ édition, 1997.

Morris et al. 1999
Robert MORRIS, Eddie KOHLER, John JANNOTTI, et M. Frans KAASHOEK. « The Click modular router ». Dans *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'1999)*, pages 217–231, Kiawah Island, SC, décembre 1999.

Mosberger 1993
David MOSBERGER. « Memory Consistency Models ». *ACM Operating Systems Review*, 27(1) : 18–26, janvier 1993.

Mosberger et Peterson 1996
David MOSBERGER et Larry L. PETERSON. « Making Paths Explicit in the Scout Operating System ». Dans *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'1996)*, pages 153–167, Seattle, Washington, octobre 1996. USENIX Association.

Motorola Inc 1997a
MOTOROLA INC. « *PowerPC Microprocessor Family : The Programming Environments For 32-Bit Microprocessors* », 1997.

Motorola Inc 1997b
MOTOROLA INC. « *PowerPC Microprocessor Family : The Programming Environments For 64-Bit Microprocessors* », 1997.

Muller et al. 1997
Gilles MULLER, Bárbara MOURA, Fabrice BELLARD, et Charles CONSEL. « Harissa : A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code ». Dans *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS'1997)*, pages 1–20, Portland, Oregon, juin 1997. USENIX Association.

Nelson 1991
Greg NELSON. *Systems Programming with Modula-3*. Prentice-Hall, Upper Saddle River, USA, 1991.

Noble et al. 1997
Brian D. NOBLE, M. SATYANARAYANAN, Dushyanth NARAYANAN, James Eric TILTON, Jason FLINN, et Kevin R. WALKER. « Agile Application-Aware Adaptation for Mobility ». Dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'1997)*, pages 276–287, Saint-Malo, France, octobre 1997.

OMG 1999
OMG. « *CORBA Component Model Joint Revised Submission* ». Object Management Group, juillet 1999. 99-07-01.

OMG 2001
OMG. « *The Common Object Request Broker : Architecture and Specification* ». Object Management Group, février 2001. CORBA 2.4.2.

Organick 1972
Elliott I. ORGANICK. *The Multics System : An Examination of Its Structure*. MIT Press, Cambridge, MA, 1972.

PalmOS
PALM INC. « PalmOS ». http : //www.palmos.com.

Peterson 1981
Gary L. PETERSON. « Myths about the mutual exclusion problem ». *Information Processing Letters*, 12(3) : 115–116, juin 1981.

Pu et al. 1995
Calton PU, Tito AUTREY, Andrew BLACK, Charles CONSEL, Crispen COWAN, Jon INOUYE, Lakshmi KETHANA, Jonathan WALPOLE, et Ke ZHANG. « Optimistic Incremental Specialization : Streamlining a Commercial Operating System ». Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'1995)*, pages 314–324, Copper Mountain Resort, Colorado, décembre 1995.

Reed et Fairbairns 1997
Dickon REED et Robin FAIRBAIRNS. « Nemesis Kernel Overview ». Rapport Technique, University of Cambridge, mai 1997.

Reid et al. 2000
Alastair REID, Matthew FLATT, Leigh STOLLER, Jay LEPREAU, et Eric EIDE. « Knit : Component Composition for Systems Software ». Dans *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'2000)*, pages 347–360, San Diego, California, octobre 2000. USENIX Association.

ReTINA 1999
RETINA. « Extended DPE Resource Control Framework Specifications ». Limited Specifications AC048/D1.01xtn, ACTS Project AC048, janvier 1999.

Ritchie et Thompson 1974
Dennis M. RITCHIE et Ken THOMPSON. « The UNIX Time-Sharing System ». *Communications of the ACM*, 17(7) : 365–375, juillet 1974.

Román et al. 1999
Manuel ROMÁN, Fabio KON, et Roy H. CAMPBELL. « Design and Implementation of Runtime Reflection in Communication Middleware : the dynamicTAO Case ». Dans *Proceedings of the ICDCS'1999 Workshop on Middleware*, pages 122–127, Austin, Texas, juin 1999.

Rozier et al. 1988
Marc ROZIER, Vadim ABROSSIMOV, François ARMAND, Ivan BOULE, Michel GIEN, Marc

GUILLEMONT, Frédéric HERRMANN, Claude KAISER, Sylvain LANGLOIS, Pierre LÉONARD, et Will NEUHAUSER. « Chorus distributed operating system ». *Computing Systems*, 1(4) : 305–370, 1988.

Sandberg et al. 1985
Russel SANDBERG, David GOLDBERG, Steve KLEIMAN, Dan WALSH, et Bob LYON. « Design and implementation of the Sun Network Filesystem ». Dans *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, juin 1985.

Schmidt et al. 1998
Douglas C. SCHMIDT, David L. LEVINE, et Sumedh MUNGEE. « The design of the TAO real-time object request broker ». *Computer Communications*, 21(4), avril 1998.

Schroeder et Burrows 1989
Michael SCHROEDER et Michael BURROWS. « Peformance of Firefly RPC ». Dans *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP'1989)*, pages 83–90, décembre 1989.

Schroeder et Saltzer 1972
Michael D. SCHROEDER et Jerome H. SALTZER. « A Hardware Architecture for Implementing Protection Rings ». *Communications of the ACM*, 15(3) : 157–170, mars 1972.

Seawright et MacKinnon 1979
L. H. SEAWRIGHT et R. A. MACKINNON. « VM/370 : a study of multiplicity and usefulness ». *IBM Systems Journal*, 18(1) : 4–17, 1979.

Seltzer et al. 1996
Margo I. SELTZER, Yasuhiro ENDO, Christopher SMALL, et Keith A. SMITH. « Dealing with disaster : surviving misbehaved kernel extensions ». Dans *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'1996)*, pages 213–227, Seattle, Washington, octobre 1996. USENIX Association.

Shapiro 1986
Marc SHAPIRO. « Structure and Encapsulation in Distributed Systems : The Proxy Principle ». Dans *Proceedings of the 6th International Conference on Distributed Computer Systems (ICDCS'1986)*, pages 198–205. IEEE Computer Society, mai 1986.

Shaw et Garlan 1996
Mary SHAW et David GARLAN. *Software Architecture : Perspective on an Emerging Discipline*. Prentice-Hall, 1996.

Silberschatz et Galvin 1998
Abraham SILBERSCHATZ et Peter Baer GALVIN. *Operating Systems Concepts*. Addison-Wesley, Reading, Massachusetts, 5ème édition, 1998.

Small et Seltzer 1994
Christopher SMALL et Margo SELTZER. « VINO : An Integrated Platform for Operating Systems and Database Research ». Rapport Technique TR-30-94, Harvard University, Cambridge, Massachussets, 1994.

Stefani et al. 2000
Jean-Bernard STEFANI, Florence GERMAIN, et Elie NAJM. « Elements of an object-based model for distributed and mobile computation ». Dans *Proceedings of the 4th International Conference on Formal Methods for Open Obect-based Distributed Systems (FMOODS'2000)*, Palo Alto, California, septembre 2000.

Stevens 1994
W. Richard STEVENS. *TCP/IP Illustrated, Volume 1 : The Protocols*. Addison-Wesley, Reading, MA, USA, 1994.

Sun Microsystems Inc 1987
SUN MICROSYSTEMS INC. « RFC 1014 : XDR : External Data Representation standard », juin 1987.

Sun Microsystems Inc 1988
SUN MICROSYSTEMS INC. « RFC 1057 : RPC : Remote Procedure Call Protocol specification », juin 1988.

Sun Microsystems Inc 1998a
SUN MICROSYSTEMS INC. « *Java Message Service* », 1998.

Sun Microsystems Inc 1998b
SUN MICROSYSTEMS INC. « *Java Object Serialization Specification* », novembre 1998.

Sun Microsystems Inc 1999a
SUN MICROSYSTEMS INC. « *Dynamic Proxy Classes* », 1999.

Sun Microsystems Inc 1999b
SUN MICROSYSTEMS INC. « *Java Remote Method Invocation Specification* », décembre 1999.

System V 1995
SYSTEM V. « *Application Binary Interface : PowerPC Processor Supplement* », 1995.

Tan et al. 1995
See-Mong TAN, David RAILA, et Roy H. CAMPBELL. « An Object-Oriented Nano-Kernel for Operating System Hardware Support ». Dans *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS'1995)*, pages 220–223, Lund, Sweden, août 1995. IEEE Computer Society.

Tanenbaum 1990
Andrew S. TANENBAUM. *Structured Computer Organisation*. Prentice Hall, Englewood Cliffs, 3ème édition, 1990.

Tanenbaum 1994
Andrew S. TANENBAUM. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, 1994.

Thekkath et Levy 1993
Chandramohan A. THEKKATH et Henry M. LEVY. « Limits to Low-Latency Communication on High-Speed Networks ». *ACM Transactions on Computer Systems*, 11(2) : 179–203, 1993.

Thibault et al. 1998
Scott THIBAULT, Charles CONSEL, et Gilles MULLER. « Safe and Efficient Active Network Programming ». Dans *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS'1998)*, pages 135–143, West Lafayette, Indiana, octobre 1998. IEEE Computer Society.

Thomas 1998
Anne THOMAS. « *Enterprise Java Beans Technology : Server Component Model for the Java Platform* », décembre 1998.

TIS 1993
TIS. « *Tools Interface Standards : Portable Formats Specification* », 1.1 édition, 1993.

UDI Project 1999

UDI PROJECT. « *UDI Specifications* », septembre 1999. http : //www.project-udi.org.

VMware

VMWARE INC. « Plateforme virtuelle VMware ». http : //www.vmware.com/.

Wahbe et al. 1993

Robert WAHBE, Steven LUCCO, Thomas E. ANDERSON, et Susan L. GRAHAM. « Efficient Software-Based Fault Isolation ». Dans *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'1993)*, pages 203–216, Asheville, décembre 1993.

XML

WORLD WIDE WEB CONSORTIUM. « Extensible Markup Language (XML) ». http : //www.w3c.org/XML.

Yokote 1992

Yasuhiko YOKOTE. « The Apertos Reflective Operating System : the Concept and its Implementation ». Dans *Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA'1992)*, pages 414–434, Portland, USA, septembre 1992.

THINK : vers une architecture de systèmes flexibles

L'objectif de cette thèse est de spécifier et d'implanter une architecture de système d'exploitation flexibles. Cette architecture est nommée THINK.

Nous montrons qu'il est possible de capturer sous la forme de canevas logiciels des concepts et des principes d'architecture rencontrés dans les systèmes d'exploitation centralisés, embarqués ou répartis. Ces concepts sont au nombre de trois ; des composants, des liaisons modélisant les interactions, et des domaines modélisant l'isolation. Cette approche autorise la construction de noyaux d'infrastructure variés, susceptibles d'être mis en place, statiquement ou dynamiquement, depuis des systèmes dédiés à une application jusqu'aux systèmes monolithiques classiques en passant par les différentes formes de micronoyaux.

Nous implantons sur ce modèle une bibliothèque, nommée KORTEX, destinée aux machines POWERMACINTOSH. Cette bibliothèque propose un ensemble de composants systèmes, offrant d'une part des services du plus bas niveau comme les pilotes de périphérique, et d'autre part des services de haut niveau comme la gestion des processus et des connexions réseaux. La composition arbitraire de ces composants construit le noyau d'infrastructure désiré. La diversité des noyaux d'infrastructure construits à partir de cette bibliothèque, la rapidité et la simplicité de programmation offertes par les outils de développement, prouvent l'adéquation du modèle.

Les évaluations quantitatives du modèle d'architecture THINK et de la bibliothèque KORTEX de composants systèmes démontrent la viabilité de l'approche proposée. Les implantations du concept de liaison n'engendrent pas un coût inacceptable et elles permettent même d'obtenir des performances similaires à celles de la littérature. De plus, les bénéfices apportés par la structure flexible peuvent potentiellement être importants. Tout d'abord, les évaluations montrent un accroissement des performances d'exécution sur des noyaux de systèmes classiques. Ensuite, les besoins en ressources matérielles requis pour l'exécution des systèmes sont grandement diminués, ce qui offre des perspectives d'utilisation dans les environnements fortement contraints.

Mots clés : système d'exploitation, flexible, architecture, canevas logiciel, composant, liaison, domaine, adaptabilité, noyau, embarqué, répartition.

THINK: toward a flexible operating system architecture

The aim of this thesis is to specify and implement a uniform software architecture for flexible operating systems. This architecture is called THINK.

We show how the concepts and architectural principles underlying diverse kinds of operating systems (centralized, distributed and embedded) can be captured in a software framework. Three key concepts are exploited: components, bindings (which model interactions) and domains (which model isolation). This approach allows various kernel infrastructures to be built either statically or dynamically, ranging from micro-kernels through to classical monolithic kernels and application-specific kernels.

Using the architecture THINK, we have designed and implemented a library, called KORTEX, targeted for POWERMACINTOSH machines. This library supplies operating system components that implement low-level services such as drivers and high-level services such as thread management and network connection. Arbitrary compositions of these components result in the required kernel infrastructure. The diversity of kernels developed using this library, along with the increased rapidity and simplicity in developing such kernels demonstrate the suitability of our approach.

Performance evaluations of the THINK architecture and the KORTEX library are extremely promising. Implementations of kernel interactions in terms of bindings result in performances comparable to systems described in the literature. Indeed, the benefits achieved by increased flexibility could be potentially important. Our benchmarks show a gain in execution speed for monolithic kernels. Additionally, our kernels achieve small footprints and can execute with less hardware requirements, an important factor in resource-constrained environments such as embedded systems.

Keywords: operating system, flexible, software framework, component, binding, domain, adaptable, kernel, embedded, distributed.