N° d'ordre : 304
N° bibliothèque : 04ENSL0 304

École normale supérieure de Lyon
Laboratoire de l'Informatique et du Parallélisme
Laboratoire Bordelais de Recherche en Informatique

THÈSE

pour obtenir le grade de

Docteur de l'École normale supérieure de Lyon
Spécialité : Informatique

au titre de l'école doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 23/12/2004

par Monsieur Vincent DANJEAN

Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs

Directeur de thèse :	Monsieur	Raymond	NAMYST	
Après avis de :	Monsieur	Jean-François	MÉHAUT,	Membre/Rapporteur
	Madame	Christine	MORIN,	Membre/Rapporteur

Devant la commission d'examen formée de :

	Monsieur	Frédéric	DESPREZ,	Membre
	Monsieur	Jean-François	MÉHAUT,	Membre/Rapporteur
	Madame	Christine	MORIN,	Membre/Rapporteur
	Monsieur	Raymond	NAMYST,	Membre
	Monsieur	Jean	PAPADOPOULO,	Membre

Remerciements

Réaliser une thèse, c'est un travail de plusieurs années que je n'aurais jamais pu mener à terme sans le soutien de nombreuses personnes. Établir une liste exhaustive de tous ceux qui m'ont entouré et encouragé m'est impossible. Aussi, ne soyez pas déçu si votre nom n'apparaît pas sur cette page. Sont donc cités ci-dessous quelques personnes qui m'ont particulièrement marqué pour une raison ou une autre.

Merci à Alvin et MT, mes premiers cobureaux, pour m'avoir supporté et fait découvrir la vraie vie™[1], à Sylvie, Anne-Pascale et Corinne qui m'ont bien facilité la vie en prenant en charge les aspects administratifs et la logistique des goûters du LIP, et plus globalement aux chercheurs du LIP pour les bons moments passés avec eux (ou avec leur machine). Merci à Gino pour ses pizzas, à Colette, ma prof de mat. sup., qui m'a changé les idées en me forçant à faire du hors piste.

Merci à toute mon équipe, en commençant par le Chef qui organise plein de soirées décadentes, à Guillaume qui choisit les films des soirées du Chef (et qui a un super site web), à Olivier pour ses adresses à Lyon, au Professeur Wacrenier pour tous ses ~~ragots~~ conseils, à Alex pour la réalisation du CD de Doudou, à Sam pour m'avoir fait découvrir 20^th^ Century Boy (au fait, si tu lis ça, je n'ai pas encore lu le dernier tome…), à Guillaume et Isabelle pour m'avoir fait découvrir les meilleurs restos de Bordeaux et à tout le LaBRI pour m'avoir accueilli quand j'ai dû quitter Lyon.

Et, parce qu'une thèse ce n'est pas que du travail, merci à tous les copains. Merci à Anass pour son thé à la menthe, à Rémy pour ses expériences culinaires, à Lescable & Maugée pour leurs petits animaux de compagnie, à tous les dinoïstes pour tous leurs débats passionnants, à Fred qui a bien voulu me laisser l'appart pour moi tout seul pendant presque toute ma première année de thèse, à JC et Juliette qui m'ont souvent traîné au ciné pour me faire sortir et à tous les autres (la liste est vraiment trop longue ici).

Une thèse, ça débute après un DEA et ça finit par ~~un pot~~[3] une soutenance. La mienne a presque été un cadeau de Noël (à deux jours près). Je tiens donc à remercier le jury qui a bien voulu venir m'écouter ce jour-là et qui a accepté de me donner le titre de Docteur, ainsi que toutes les personnes présentes qui ont dégusté tout ce qu'avait préparé ma famille et ma (future) belle famille. Il faut aussi que je remercie Aurélien pour la cravate qu'il m'a fait porter lors de ma soutenance.

Et enfin, merci à mes parents qui m'ont soutenu tout au long de mes longues études et qui, même s'ils n'ont pas trop compris ce que je faisais, ont relu attentivement toute ma thèse. Merci à Claire pour son soutien inconditionnel pendant toute ma rédaction. Si, si, elle a réussi à ne pas craquer, c'est un exploit ! Mais je sais bien que c'est la meilleure. À mon tour de l'aider maintenant…

[1] Ils ont plein de gamins[2] tous plus mignons les uns que les autres et ils jouent super bien à frozen-bubble.
[2] Il y en a même un qui a été capable de faire planter LINUX alors qu'il n'avait même pas un an.
[3] Parfois même, des vitres se cassent toutes seules.

Résumé : En informatique, la notion de processus léger ou *thread* est désormais omniprésente. En effet, les processus légers permettent à un programme d'une part d'exploiter pleinement les ordinateurs multiprocesseurs et d'autre part d'exprimer son parallélisme intrinsèque. Dans le domaine du calcul hautes performances, les processus légers permettent de recouvrir des communications ou plus généralement des entrées/sorties avec du calcul. Ils permettent aussi aux divers composants de l'application de progresser indépendamment les uns des autres, ce qui est nécessaire avec l'utilisation d'environnements de programmation toujours plus complexes comme MPI ou CORBA.

Mes travaux avaient pour objectif principal d'aboutir à la conception d'une bibliothèque de processus légers performante sur une vaste gamme d'architectures (machines mono- ou multiprocesseurs, technologie « multithreading simultané », etc.) et capable d'offrir les fonctionnalités demandées par les programmes de calcul hautes performances. Dans un premier temps, j'ai proposé une extension du modèle des Scheduler Activations pour le noyau Linux permettant d'être réactif aux interruptions dans une bibliothèque de processus utilisateurs. J'ai ensuite étendu ce mécanisme de manière à unifier la gestion des interruptions et des scrutations dans un environnement multithreadé. Enfin, j'ai proposé un mécanisme de prise de traces permettant de reconstituer précisément le déroulement d'un programme multithreadé, y compris lorsque l'ordonnancement est à deux niveaux.

Ces travaux ont été implémentés au sein du logiciel PM^2. La bibliothèque Marcel offre des processus performants sur une vaste gamme de processeurs et de systèmes en restant suffisamment flexible pour permettre aux applications qui le nécessitent de diriger précisément l'ordonnancement de leurs processus. Les applications peuvent être tracées de manière à pouvoir reconstituer et observer leur comportement précis. La trace obtenue peut être convertie au format du logiciel Pajé qui permet alors de visualiser graphiquement le déroulement de l'application.

Mots-clés : Processus légers, ordonnancement à deux niveaux, calcul hautes performances, réactivité, Scheduler Activations, traces

Abstract: Nowadays, threads are widely spread in computer science. Indeed, multithreading allows applications not only to fully exploit multiprocessor computers, but also to reveal its intrinsic parallelism. In the context of high performance computing, threads are commonly used to overlap computation with communication. They also allow various execution flows within the application to progress independently one from another. This is a mandatory functionality regarding the implementation of complex middleware such as MPI or CORBA.

My work aims at providing an efficient threads library targeting a wide range of architectures (mono- or multiprocessor computers, SMT technology, etc.) and able to fulfil the requirements of high performance computing programs. First, I have extended and implemented the Scheduler Activation model within the Linux kernel, so that user threads can be extremely reactive to hardware interruptions. Then, I did expend this mechanism to unify the management of interrupts and polling in multithreaded environments. Finally, I have designed new tracing mechanisms allowing to precisely rebuild the execution of multithreaded programs, even with a two-level scheduling.

All these works have been implemented within the PM^2 software suite. The Marcel library provides efficient multithreading on a large range of processors and systems. Marcel is flexible enough to allow an application to precisely manage its threads scheduling when needed. Applications can be traced in order to observe their precise behavior. The generated traces can be converted to the Pajé software format, so that application behavior can be graphically observed.

Keywords: Threads, two-level scheduling, high-performance computing, reactivity, Scheduler Activations, traces

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Ce document présente les travaux que j'ai réalisés durant ma thèse sous la direction de Raymond NAMYST d'abord au LIP (*Laboratoire de l'Informatique et du Parallélisme*) à l'École normale supérieure de Lyon, puis au LABRI (*Laboratoire Bordelais de Recherche en Informatique*) à l'Université Bordeaux 1.

1.1 Le calcul hautes performances

Les besoins de puissance de calcul informatique dans quelque domaine que ce soit, comme le calcul numérique ou financier, la modélisation, la réalité virtuelle ou la fouille de données, ont toujours augmenté, nécessitant une évolution régulière du matériel. Les premières architectures disponibles ont été les supercalculateurs parallèles. Développés dans les années 70, ils ont connu leur âge d'or dans les années 80 avant de s'effondrer pour être remplacés, dans les années 90, par des grappes de station. Ces dernières étaient parfois un peu moins puissantes que les supercalculateurs, mais les économies financières réalisées ont convaincu de nombreux centres de calcul.

Cette évolution a eu plusieurs conséquences importantes relatives à la recherche en informatique. Les supercalculateurs, de par leur coût, étaient rarement accessibles aux chercheurs académiques. La micro-informatique s'est répandue beaucoup plus facilement dans les laboratoires de recherche. Cela a permis l'émergence de nombreuses études concernant non plus le matériel informatique, mais le logiciel. La nécessité de faire collaborer un grand nombre de stations de travail pour obtenir les puissances de calculs désirées a mis en évidence l'existence de nombreux problèmes à résoudre. Cet effort de recherche concernant le logiciel n'a cessé de croître depuis lors.

L'apparition de stations supportant plusieurs flots d'exécution simultanément (machines SMP, NUMA, technologie hyperthreading, etc.) a mis en exergue les processus légers qui ont vu leur emploi se généraliser, y compris dans le domaine du calcul hautes performances. Cependant, l'interface de la plupart des bibliothèques de processus légers, en se conformant au standard en vigueur, s'est révélée inadaptée aux besoins particuliers du calcul hautes performances : pas de contrôle possible de l'application sur l'ordonnancement, intégration minimale avec les communications, etc. Pour bénéficier des processus légers de manière réellement efficace, les applications doivent prendre en compte la spécificité de la plate-forme matérielle (nombre de processeurs sur une machine SMP, etc.) et y être spécifiquement ajustées. Cela est bien évidemment contraire à l'objectif de portabilité permettant de développer des applications sur une « petite » machine et de l'exécuter par la suite sur une « grosse » grappe.

1.2 Objectifs de la thèse

Il s'agit, dans un premier temps, de faire le point sur les besoins actuels dans le domaine des processus légers concernant le calcul hautes performances. Les bibliothèques de processus légers actuelles ne répondent pas totalement aux attentes du calcul parallèle ; il faut donc identifier précisément ces manques.

Cela nous conduira à proposer un certain nombre d'extensions et de fonctionnalités dans le but d'adapter les bibliothèques de processus légers à nos besoins. Plusieurs axes seront explorés plus en détail. On peut citer :

– la notion de « portabilité des performances » qui permet à une application de bénéficier automatiquement de l'environnement logiciel en adéquation avec la plate-forme logicielle utilisée ;
– la notion de *réactivité* et plus généralement l'intégration de la problématique des événements d'entrée/sortie en présence de processus légers ;
– les outils nécessaires à une exploitation aisée des environnements multithreadés, en particulier les mécanismes d'observation à granularité fine des applications multithreadées.

Afin d'évaluer ces travaux, ils seront tous intégrés à une même bibliothèque de processus légers développée à cette occasion. Bien évidemment, d'autres bibliothèques peuvent bénéficier de ces travaux si l'effort de développement associé est réalisé.

1.3 Contribution

La contribution majeure de ma thèse consiste en la conception d'une architecture originale d'ordonnanceur de processus légers capable de fournir des fonctionnalités puissantes aux applications de manière transparente à l'architecture sous-jacente. Ces travaux ont abouti à un ensemble de réalisations intégrées à l'environnement de programmation parallèle distribuée PM^2. Cet environnement est développé de manière collaborative au sein de l'équipe RUNTIME. Mes réalisations, qui correspondent à plus de 170 000 lignes de code, vont de l'implémentation d'une bibliothèque de processus légers (MARCEL) à l'extension du noyau LINUX en passant par la réalisation d'un environnement de prise de trace permettant de comprendre le fonctionnement et les performances de l'ensemble. Ces travaux ont fait l'objet de plusieurs publications citées au chapitre 10 (page 147) ainsi que de nombreux séminaires.

1.4 Plan du manuscrit

Le chapitre 2 présente le contexte dans lequel s'inscrit mon travail. Nous y présentons les principales techniques de programmation parallèle en mettant en évidence l'utilisation toujours croissante des processus légers. Nous montrons ensuite que la présence de ces derniers dans les applications de calcul parallèle hautes performances nécessite de résoudre quelques problèmes si l'on souhaite un maximum d'efficacité et de performance. Le chapitre 3 présente en détail les modèles de processus légers existants en mettant en lumière leurs atouts et leurs limitations respectives. Dans le chapitre 4, nous exposons les techniques novatrices développées au cours de cette thèse pour obtenir une bibliothèque de processus légers nommée MARCEL parfaitement adaptée au calcul hautes performances. Avec le chapitre 5, nous descendons à un niveau un peu plus technique pour explorer quelques éléments représentatifs de notre implémentation. En particulier, nous étudions quelques problèmes relatifs à la synchronisation interne de notre bibliothèque. La gestion de quelques ressources critiques est également abordée. Afin de pouvoir observer le déroulement précis de l'ordonnancement des processus légers de nos programmes, nous avons également conçu des outils de trace pour environnements multithreadés qui font l'objet du chapitre 6. Nous arrivons alors tout naturellement à la section de test et d'évaluation, au chapitre 7. Nous comparons notre bibliothèque aux autres et nous montrons l'efficacité des mécanismes que nous avons introduits. Ces observations nous permettent de conclure sur ces travaux et de dégager les principales perspectives d'avenir qui en découlent.

Chapitre 2

Parallélisme applicatif pour le calcul hautes performances

Sommaire

Ce chapitre a pour objectif d'expliquer les motivations qui ont conduit à la réalisation de ma thèse. Il présente le contexte dans lequel se sont inscrits mes travaux en mettant en évidence les insuffisances que je me suis efforcé de combler.

L'équipe dans laquelle je me suis inséré bénéficie d'une expérience certaine dans le domaine du calcul parallèle hautes performances. Partant du constat que les applications et les environnements de calcul parallèle (re-)programmaient généralement les supports exécutifs sur lesquels ils s'appuient, notre équipe s'est intéressée principalement à comprendre pourquoi des supports exécutifs génériques n'étaient pas utilisés[Nam01]. Cet effort de recherche permet de mettre en évidence les lacunes dans les supports exécutifs actuels et aussi les besoins spécifiques de certains environnements. Bien sûr, nous ne sommes pas les premiers à nous intéresser à ces problèmes et des solutions ont été proposées dans certains cas, en particulier dans le domaine des bibliothèques de communication. Ainsi, un standard comme MPI

[Mes95] est utilisé par de très nombreuses applications parallèles et des implémentations efficaces existent pour de nombreux réseaux. Dans notre équipe, nous avons développé la bibliothèque de communication MADELEINE 3 permettant à une application d'exploiter de manière efficace et transparente une vaste gamme de réseaux haut débit (GIGA ETHERNET, SCI, MYRINET, etc.) Cependant, en dehors des bibliothèques dédiées aux communications, la situation est beaucoup plus mitigée.

Au cours de mes travaux, je me suis intéressé plus particulièrement au multithreading dans le domaine du calcul hautes performances. Après être longtemps restés confinés à des domaines particuliers[1], les processus légers sont largement répandus de nos jours. Ce chapitre est l'occasion d'expliquer les raisons de cet engouement pour ce paradigme, mais également de mettre en évidence ses limites vis-à-vis des supports exécutifs dans le domaine du calcul hautes performances.

2.1 Des supports exécutifs omniprésents dans le parallélisme

En 1972, Micheal J. FLYNN a proposé une taxonomie[Fly72] pour les architectures parallèles qui sert encore souvent de référence. Elle distingue les systèmes en fonction des flots d'exécution et des flots de données et définit quatre classes :

SISD (*Single Intruction, Single Data*) : c'est le modèle traditionnel des machines séquentielles avec programme et données stockées en mémoire ;

SIMD (*Single Instruction, Multiple Data*) : les mêmes instructions sont exécutées sur des flots de données différents. On trouve dans cette catégorie les supercalculateurs *MasPar* et *Thinking Machine* où plusieurs processeurs effectuent les mêmes instructions en même temps sur des données différentes. Cette technique est ensuite reprise dans les processeurs vectoriels comme le *Cray-1* et continue d'être mise en œuvre dans les processeurs actuels à travers, par exemple, la technologie MMX ™. Cela permet de multiplier les unités de calcul sans complexifier la gestion du flot d'exécution mais nécessite des problèmes adaptés (nombreuses données indépendantes à traiter de manière identique) ;

MISD (*Multiple Instruction, Single Data*) : plusieurs instructions travaillent sur le même flot de données. Ce modèle est également désigné sous le nom de *pipeline* : en découpant chaque instruction en plusieurs étapes, par exemple chargement, décodage, lecture et exécution, le processeur peut commencer une nouvelle instruction dès qu'il a terminé la première étape de l'instruction précédente.

MIMD (*Multiple Instruction, Multiple Data*) : plusieurs flots d'exécution travaillent sur des données différentes. C'est le cas, par exemple, des processeurs *superscalaires* qui sont capables de réordonnancer de manière interne les instructions ou parties d'instruction à exécuter de manière à maximiser l'utilisation des ressources matérielles disponibles (unités de calcul entier ou flottant, unités de lecture/écriture, unités de branchement, etc.)

FIG. 2.1 – Taxonomie de FLYNN

Depuis le début de l'informatique, la vitesse d'exécution des microprocesseurs ne cesse d'augmenter. De nos jours, il est courant de trouver des processeurs cadencés à plusieurs

[1] Les processus légers sont apparus dans les systèmes propriétaires dans les années 80, voire à la fin des années 70. Leur rôle premier était de supporter les moniteurs transactionnels comme par exemple CICS chez IBM ou TDS chez BULL. En effet, les applications transactionnelles étant destinées à servir des centaines, voire des milliers d'utilisateurs, attribuer un processus à chaque utilisateur, comme cela avait été possible pour les systèmes à temps partagé, aurait été beaucoup trop coûteux ici.

La taxonomie de FLYNN n'est pas la seule qui soit pertinente. En particulier, il est souvent important de différencier les systèmes suivant les relations entre les processeurs et la mémoire. On distingue ainsi deux familles :

DMS (*Distributed Memory systems* ou « systèmes à mémoire distribuée ») : les processeurs d'un système à mémoire distribuée utilisent des échanges de messages pour se communiquer des données ;

SMS (*Shared-Memory systems* ou « systèmes à mémoire partagée ») : tous les processeurs d'un système à mémoire partagée peuvent accéder à la même mémoire partagée très rapidement. Trois modèles peuvent être distingués ici :

- **UMA (*Uniform Memory Access*) :** tous les processeurs ont exactement la même vision de la mémoire (même adressage physique et même temps d'accès) ;
- **NUMA (*Non-Uniform Memory Access*) :** les processeurs voient la même mémoire physique, mais les temps d'accès peuvent varier en fonction du processeur et de la zone mémoire visée. Parmi les NUMA, on distingue les CCNUMA (*cache coherent NUMA*) plus facile à programmer grâce à une cohérence mémoire garantie par le matériel ;
- **COMA (*Cahe-Only Memory Access*) :** les processeurs rapatrient dans leur mémoire local les données dont ils ont besoin, *i.e.* les données changent de localisation physique automatiquement.

Pour compléter cette classification, ces architectures peuvent être divisées en deux catégories :

- **AMP (*Asymetric MultiProcessor*) :** certaines parties du système d'exploitation comme les entrées/sorties ne peuvent être exécutées que depuis le processeur « maître ». Cette architecture n'est pas très extensible en raison du goulot d'étranglement introduit par le processeur maître ;
- **SMP (*Symetric MultiProcessor*) :** tous les processeurs peuvent exécuter n'importe quel code du système d'exploitation ;

Cette classification est orthogonale à celle de FLYNN — on peut avoir des systèmes SM-MIMD (machine SMP classique), DM-SIMD (grappe de stations de travail programmées avec MPI), etc.

Il est à noter que le système d'exploitation, le *firmware* (logiciel interne) ou même le matériel peuvent cacher à l'utilisateur la véritable nature d'une machine. Ainsi, une machine à mémoire partagée peut se révéler être en fait un système à mémoire distribuée interconnecté par un réseau (très) rapide avec du matériel et des logiciels assurant au programmeur une vision d'un système à mémoire partagée. C'est le cas, par exemple, de l'Origin 2000 de SGI.

Au niveau purement logiciel, on trouve de nombreuses contributions — MPI [Mes95] étant probablement la plus connue — permettant d'exploiter les machines à mémoire partagée avec un système de passage de messages. Réciproquement, des logiciels comme HPF[BB00] ou Kerrighed[VLR+03] permettent de programmer des systèmes à mémoire distribuée comme des systèmes à mémoire partagée.

FIG. 2.2 – Classification des architectures parallèles par rapport aux accès à la mémoire

gigahertz. Bien sûr, chaque instruction assembleur du programme n'est pas exécutée en un seul cycle à cette fréquence : il y a trop d'opérations logiques à effectuer. Mais, grâce à des techniques de pipeline et de réordonnancement des instructions au sein même du processeur pour occuper au mieux les unités de calcul, on peut espérer en moyenne cette puissance de calcul. Cela reste cependant très inférieur à ce qui est nécessaire pour la résolution de nombreux problèmes relatifs à la cryptographie ou la physique des particules par exemple.

Une équipe de chercheurs français, issue de la Direction Centrale de la Sécurité des Systèmes d'Information (DCSSI) et de l'Université de Versailles-Saint Quentin a réussi à créer une collision du code SHA-0, algorithme de hachage développé dans les années 90 par la National Security Agency. Cet objectif a été atteint à l'aide du supercalculateur TeraNova, un cluster constitué de serveurs Bull NovaScale®. En assurant un fonctionnement ininterrompu pendant trois semaines, le cluster TeraNova de Bull a permis de casser le code SHA-0 en 80 000 heures * processeurs, démontrant ainsi sa puissance et sa fiabilité.

« Depuis 1998, les progrès en algorithmique ont permis de gagner un facteur 1000 sur la complexité de l'attaque de SHA-0. Malgré ce gain important, nous n'aurions pas pu créer aussi rapidement et aussi simplement une collision, sans l'appui d'un supercalculateur de la rapidité et de la flexibilité de TeraNova. » indique Antoine Joux [...]

D'une puissance de 1,3 teraflops, ce cluster composé de 16 serveurs de 16 processeurs Itanium® 2 d'Intel a été déployé dans le cadre du projet Ter@tec, au centre DAM-Ile de France du CEA. [...]

Extrait d'un communiqué de presse du 29 septembre 2004 de l'entreprise BULL. *Le communiqué complet peut être trouvé sur son site web (*`http://www.bull.fr/`*).*

L'algorithme utilisé a été proposé lors de la découverte d'une faille de ce protocole SHA-0 en 1998. Cependant, la puissance de calcul des machines parallèles de l'époque n'était pas suffisante alors pour exploiter la faille.

FIG. 2.3 – Développement des calculs cryptographiques grâce au parallélisme

Le parallélisme, *i.e.* faire travailler plusieurs unités simultanément, permet de multiplier la puissance de calcul en multipliant le matériel utilisé. Pour ce faire, il existe diverses architectures qui peuvent être classifiées de différentes manières (cf. Figures 2.1 et 2.2). Ce parallélisme matériel peut être mis en place à des échelles très variées. On peut ainsi distinguer :

– les machines multiprocesseurs de petite taille ;
– les machines multiprocesseurs de grande taille avec organisation interne complexe (accès mémoire non uniformes) ;
– les grappes où plusieurs machines sont reliées par un réseau dédié et rapide ;
– la grille où plusieurs sites dispersés sur la toile INTERNET collaborent les uns avec les autres.

Bien évidemment, ces solutions ne sont pas exclusives et l'on va généralement trouver des combinaisons de ces environnements. Ainsi, il est courant de nos jours de trouver les machines parallèles constituées d'une grappe de grappes de machines multiprocesseurs. Les recherches en parallélisme ont d'ailleurs donné lieu à un foisonnement de solutions et techniques.

L'exploitation efficace de ces machines parallèles éventuellement hiérarchiques nécessite une programmation adaptée : un programme séquentiel peut difficilement tirer partie d'une plate-forme matérielle parallèle sans adaptation. Malheureusement, la programmation parallèle n'est pas encore facilement accessible ni largement utilisée par la communauté « non spécialiste ». Pour anecdote, un collègue mathématicien[2] programmant un solveur numérique a constaté avec étonnement que passer d'une machine biprocesseur à une machine à 16 processeurs n'accélérait pas l'exécution de son programme. Son code était largement parallélisable, mais écrit de manière séquentielle et compilé classiquement avec `gcc`...

Après de nombreuses années de recherches pour obtenir des outils de parallélisation entièrement automatiques, force est de constater que le résultat n'est pas à la hauteur des premières espérances. On espérait pouvoir écrire une application de manière « séquentielle » avec des outils capables d'extraire le parallélisme du programme pour exploiter efficacement les machines parallèles. En pratique, les résultats obtenus ne concernent que quelques cas particuliers (nids de boucles, etc.) De fait, pour être efficace, une application parallèle, sauf cas particuliers, ne peut pas être écrite sans tenir compte de cette parallélisation. L'informatique parallèle est cependant cruciale puisqu'elle permet de réaliser des calculs impossibles sans cela, comme en témoigne ce résultat en cryptographie obtenu récemment par l'entreprise BULL (cf. Figure 2.3).

2.1.1 Comment programme-t-on les machines parallèles ?

Afin de simplifier la tâche du programmeur lors de la conception de son application parallèle, de nombreux outils sont disponibles. Ils peuvent prendre des aspects très variés et intervenir à différents stades du développement de l'application. Nous en présentons ici un panorama représentatif.

2.1.1.1 Langages et compilateurs

La première classe d'outils concerne les langages et les compilateurs. Cette solution permet au programmeur de se reposer sur le compilateur pour la parallélisation de son application. Le plus simple serait bien sûr que le compilateur prenne en charge totalement et de façon transparente la parallélisation de l'application. Néanmoins, ce problème est encore bien trop ardu pour le niveau de connaissance actuel. Aussi, le programmeur est-il obligé d'aider le compilateur à réaliser sa mission.

Certains langages proposent au programmeur d'*annoter* leur code pour signaler au compilateur des propriétés relatives à la parallélisation. C'est le cas des langages comme OPENMP [Ope99a] ou encore HPF [BB00] qui ont motivé de nombreux travaux [Dar99]. Cependant, même avec cette aide, l'histoire a montré que la parallélisation reste très difficile. Les efforts dans ce domaine ont diminué : les équipes ont souvent réorienté leurs recherches vers d'autres aspects de la compilation. En définitive, un langage tel que HPF n'a pas confirmé les espoirs que l'on avait placés en lui...

C'est pourquoi d'autres langages se contentent de mettre à la disposition du programmeur des primitives explicites de parallélisation. Des primitives permettant de déclarer de nouveaux flots d'exécution parallèles ainsi que des fonctions de synchronisation sont fournies. À charge du programmeur de les utiliser correctement. La tâche du compilateur est alors grandement simplifiée puisque le parallélisme à exploiter lui est fourni explicitement. Mais il lui faut encore mettre en adéquation le parallélisme demandé par l'application et celui fourni par la plate-forme matérielle d'exécution, ce qui est loin d'être évident, surtout dans le cas de machines parallèles hiérarchiques (grappe de SMP, machines NUMA, etc.)

[2] Il a préféré garder l'anonymat.

2.1.1.2 Bibliothèques de haut niveau spécialisées

Dans le domaine du calcul parallèle hautes performances, il existe toute une classe de problèmes récurrents qui font désormais l'objet de bibliothèques parallèles spécialisées comme la bibliothèque SCALAPACK [BCC+96]. Celle-ci propose de nombreuses opérations matricielles (factorisation LU, etc.) qui peuvent être accélérées en exploitant plusieurs processeurs simultanément. Lorsque l'on désire utiliser ces fonctions, il suffit d'appeler la bibliothèque spécialisée sans se soucier de la parallélisation qui a déjà été effectuée.

Il est à noter que l'efficacité de ces bibliothèques dépend fortement des données qui leur sont soumises. Lorsque les données sont régulières, de nombreux résultats obtenus en algorithmie théorique permettent d'offrir des garanties d'optimalité. En revanche, avec des données irrégulières comme avec les matrices creuses, il reste encore beaucoup de travail de recherche à effectuer tels ceux menés au sein du projet PASTIX [HRR00].

Enfin, l'utilisation de ces bibliothèques nécessite la plupart du temps que l'utilisateur spécifie lui-même l'enchaînement des opérations ainsi que la répartition des données. Dans le cas d'opérations matricielles conséquentes, les transferts de données peuvent devenir un goulot d'étranglement ; définir un placement des données optimal n'est pas aisé. On voit depuis peu apparaître des bibliothèques qui essaient de prendre en compte ces déplacements de données ou qui fournissent des outils pour les gérer depuis l'application comme les bibliothèques FAST/DIET [CDL+02] par exemple.

2.1.1.3 Environnements de programmation

Lorsque les applications ont des besoins particuliers ou doivent fonctionner dans des environnements très hétérogènes (*i.e.* sur la grille par exemple), ni les bibliothèques spécialisées ni les compilateurs ne conviennent. En effet, les bibliothèques spécialisées, comme leur nom l'indique, ne répondent qu'à des problèmes ciblés[3]. Les langages et les compilateurs, eux, ciblent généralement une machine particulière ou éventuellement une grappe homogène. Ceci est compréhensible : les combinaisons possibles entre les différents types d'architectures matérielles (processeurs, types de réseaux, topologies des réseaux, etc.) dépassent de loin ce que l'on est capable de gérer automatiquement de nos jours. Les compilateurs sont inaptes, dans le cas général, à générer par eux-mêmes des motifs de communication performants, des découpages et répartitions de charge équilibrée, etc.

Pour gérer ces situations complexes, l'aide du programmeur est donc indispensable. Heureusement, afin de simplifier sa tâche, des environnements de programmation parallèle ont été développés. Il s'agit ici de fournir au programmeur tout un ensemble d'outils qui l'aideront à construire son application parallèle en lui évitant de réinventer la roue à chaque fois.

Ces environnements peuvent fournir des fonctionnalités dans des domaines assez variés comme les communications[SDGM94], synchronisations distribuées[vRBM96], distribution des données[FGIS97], régulation de charge[CRCM95], etc. Certains environnements se focalisent sur un paradigme particulier (mémoire distribuée, etc.) rendant difficile l'utilisation de plusieurs paradigmes différents au sein du même programme. D'autres, au contraire, essaient de proposer le maximum d'outils diversifiés afin que le programmeur puisse choisir

[3] La majorité des bibliothèques spécialisées proposent de réaliser des calculs numériques classiques (opérations matricielles, etc.) optimisés pour une architecture particulière.

ce qui l'intéresse le plus. À cet égard, l'un des plus connus est l'environnement GLOBUS [FK97] qui est souvent présenté comme une « boîte à outils » pour le parallélisme.

2.1.2 Le rôle des supports exécutifs

Un point commun à l'ensemble des solutions présentées précédemment s'avère l'utilisation d'un support exécutif pour faire tourner les applications. Un support exécutif est une couche logicielle à la limite entre le système d'exploitation et le reste de l'application. Il a généralement pour but d'améliorer et d'étendre les fonctionnalités offertes par le système d'exploitation. L'utilisation de supports exécutifs répond à des besoins variés.

Dans le cas des langages, l'utilisation d'un support exécutif simplifie la tâche du compilateur qui peut alors insérer des appels directs à ce support au lieu de devoir émettre l'ensemble des instructions nécessaires à une fonctionnalité particulière. Cela permet également de gérer les cas où la stratégie qui sera employée n'est pas connue au moment de la compilation. Ainsi, pour certains types de programmes parallèles irréguliers, la répartition de la charge devra se faire dynamiquement à l'exécution. Un compilateur ne pourrait pas prévoir la charge lors de la compilation ; le support exécutif prend alors le relais à l'exécution.

En ce qui concerne les bibliothèques spécialisées, la situation est légèrement différente. Destinées à des problèmes bien précis, ces bibliothèques ont souvent peu de besoins particuliers, excepté dans le domaine des communications. L'utilisation d'un support exécutif permettant d'exploiter efficacement les réseaux permet à une telle bibliothèque d'être utilisable sur une gamme plus vaste de matériel. L'essor des machines multiprocesseurs est en train de modifier ce paysage : le multithreading est de plus en plus souvent intégré afin d'exploiter efficacement ces machines. Ainsi, les prochaines versions des bibliothèques distribuées BLAS reposeront sur l'utilisation conjointe de MPI pour les communications et du multithreading pour les machines multiprocesseurs.

Enfin, les environnements de programmation utilisent les supports exécutifs afin d'être plus facilement portables. Certains continuent, bien sûr, d'être écrits de manière monolithique, mais en grande majorité, ils sont structurés en couches de façon à pouvoir être adaptés plus facilement sur des architectures variées. De cette façon, lorsqu'un nouveau matériel apparaît, seul le support exécutif doit être modifié pour en tirer partie.

En résumé, les supports exécutifs permettent d'étendre efficacement les fonctionnalités offertes par les systèmes d'exploitation. Ils permettent également d'offrir une interface de portabilité pour les applications ou les environnements, de sorte que ces derniers deviennent plus indépendants des matériels exploités. Cela simplifie alors leur portage vers de nouveaux matériels : seul le support exécutif doit être adapté/porté.

2.2 Processus légers : une technologie bien adaptée aux supports exécutifs parallèles

L'utilisation de processus légers dans les applications n'est pas encore très répandue, particulièrement dans les applications de calcul hautes performances. L'engouement suscité autour de langages comme JAVA modifiera peut-être cet état de fait d'ici quelques années, mais très peu de codes de calcul hautes performances sont actuellement écrits en utilisant des processus légers. Et pourtant, les supports exécutifs offrent de plus en plus fréquemment ce paradigme. Plusieurs raisons peuvent l'expliquer : les processus légers permettent d'exploiter efficacement les nouvelles architectures parallèles et ils offrent des fonctionnalités très utiles aux compilateurs parallèles ou aux environnements de programmation, même si l'application ne les utilise pas directement.

2.2.1 Exploitation des nouvelles architectures

Les processus légers sont souvent préconisés pour exploiter avec efficacité les machines multiprocesseurs [4]. Cet aspect se développera de plus en plus en raison de l'évolution actuelle des architectures matérielles : les *threads* sont désormais présents directement dans le matériel.

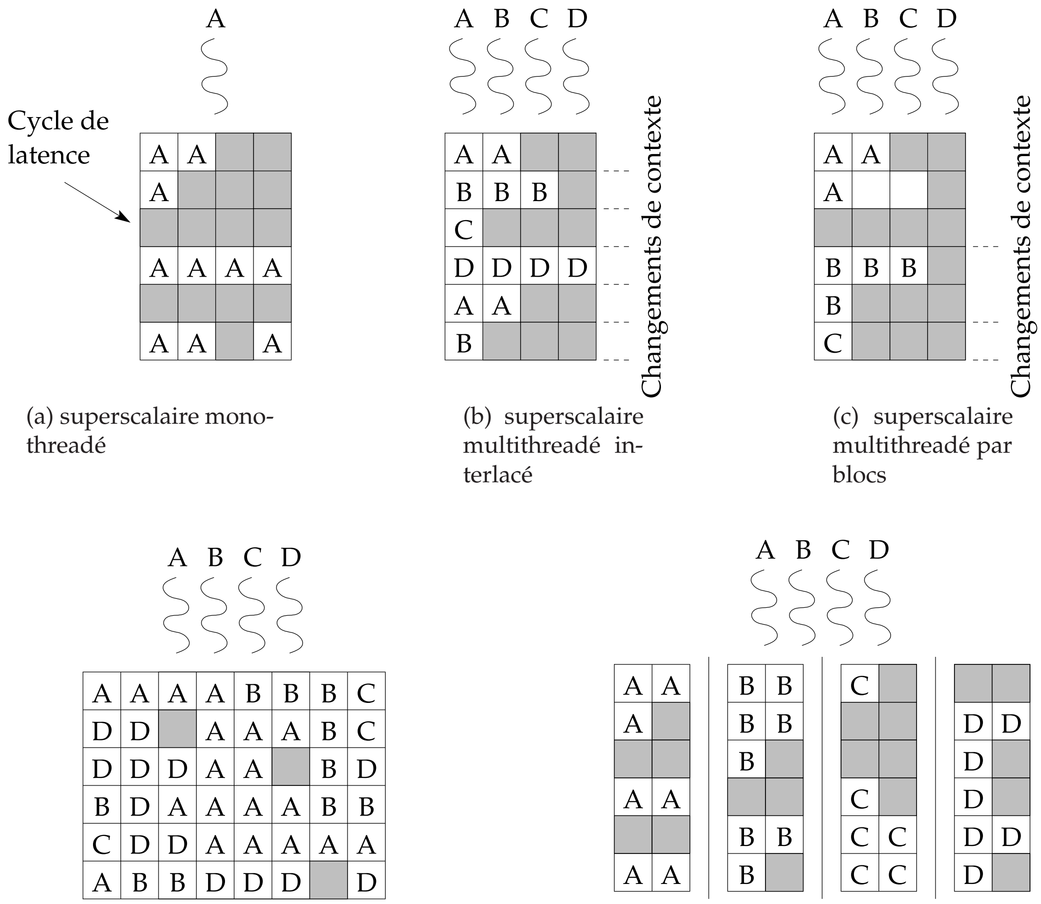
***Instruction Level Parallelism* (ILP).** Pour offrir des processeurs de plus en plus puissants, les constructeurs ont augmenté le parallélisme intrinsèque de leurs circuits. La première piste a été d'extraire et exploiter le parallélisme contenu dans le flot d'exécution grâce au pipeline et/ou au réordonnancement interne des instructions. Les dernières générations de processeurs sont capables d'exécuter des instructions *a priori* sans savoir si elles seront réellement utiles ou non. On peut ainsi exécuter en parallèle les deux branches d'un test avant de connaître le résultat de ce test. Ainsi, le processeur ITANIUM permet au compilateur de marquer des instructions avec un prédicat, les résultats de ces instructions ne seront pris en compte que si le prédicat se révèle vrai.

Cette approche possède l'immense avantage qu'elle ne suppose aucune aide de la part du programmeur. L'ILP est exploité par la magie des compilateurs et par l'exécution dans le désordre gérée par le matériel.

***Thread Level Parallelism* (TLP).** Le parallélisme intrinsèque à un flot d'exécution est toutefois limité. Pour occuper les unités de calcul disponibles, d'autres sources de parallélisme doivent être trouvées. En outre, le mur de la densité thermique interdit d'augmenter trop les fréquences des processeurs et donc la complexité du microcode si on ne veut pas avoir des instructions trop lentes à s'exécuter. C'est pourquoi on voit de plus en plus apparaître des processeurs multithreadés, c'est-à-dire capables de partager les ressources de calcul entre plusieurs flots d'exécution simultanément. L'article [URŠ03] constitue une excellente étude des différents types de processeurs présentés dans la Figure 2.4 offrant du multithreading matériel explicite.

On peut noter que cette évolution du matériel (ILP vers TLP) est récente. L'exploitation de l'ILP grâce aux compilateurs et aux bibliothèques de passage de messages pour réseaux d'interconnexion rapides a été très poussée, allant jusqu'au point d'utiliser un processus par nœud sur les machines à mémoire partagée avec MPI pour communiquer. La nécessiter d'évoluer vers le TLP au niveau du matériel pour les nœuds des grappes relance l'intérêt d'exploiter efficacement les architectures à mémoire partagée sans pour autant perturber la communication par messages qui devrait normalement se restreindre au parallélisme inter-nœuds. L'intérêt d'un multithreading efficace, en particulier en présence de communications, apparaît donc clairement ici.

[4] Il existe cependant quelques cas où l'absence d'espaces d'adressage distincts constitue un désavantage. Par exemple, des espaces d'adressage distincts sont indispensables pour pouvoir exploiter plus de 4 Go de mémoire sur architecture X86 avec le mode PAE (*Physical Address Extension*). Dans ce cas, on préférera utiliser des processus lourds distincts communiquant avec, par exemple, MPI.



(a) superscalaire mono-threadé

(b) superscalaire multithreadé interlacé

(c) superscalaire multithreadé par blocs

(d) multithreading simultané

(e) multiprocesseur sur puce

Pour gérer plusieurs flots d'exécution, diverses stratégies sont possibles. Quelques unes sont présentées ici. Une étude complète peut être trouvée dans [URŠ03].

2.4(a) : *un seul flot d'exécution est géré. Certains cycles sont perdus car aucune instruction n'est prête ;*

2.4(b) : *à chaque cycle, un* thread *différent occupe (si possible) les unités de calcul ;*

2.4(c) : *un* thread *occupe les unités de calcul tant qu'il le peut. Un cycle de latence peut être nécessaire pour effectuer le changement de contexte ;*

2.4(d) : *n'importe quel* thread *occupe n'importe quelle unité de calcul à chaque cycle ;*

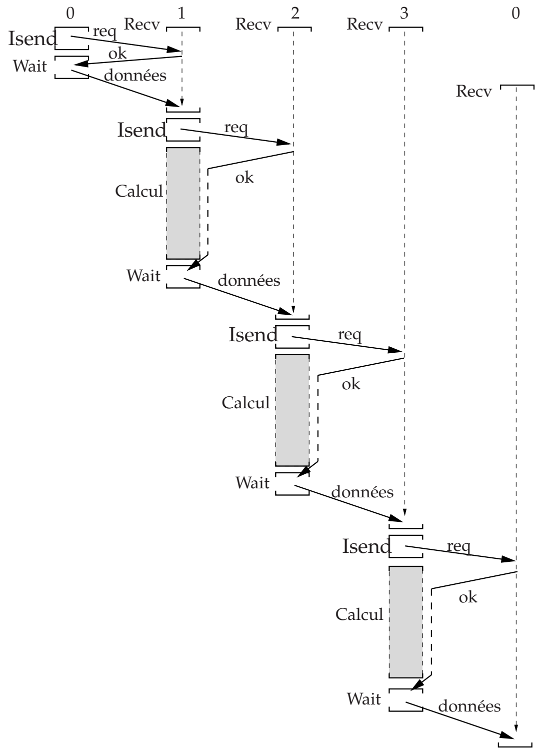
2.4(e) : *chacune des unités de calcul est dédiée à un seul et unique* thread.

FIG. 2.4 – *Processeurs avec multithreading explicite*

2.2.2 Les vertus des processus légers

La première raison pour laquelle les processus légers sont utilisés semble, comme nous venons de le voir, l'exploitation des nouvelles architectures matérielles. Mais l'exploitation de machines où plusieurs flots d'exécution partagent une même mémoire n'est pas leur unique utilité.

```
if (mynode==0) {
  MPI_Isend(next);
  MPI_Wait();
  MPI_Recv();
} else {
  MPI_Recv();
  MPI_Isend(next);
  Calcul();
  MPI_Wait();
}
```



Avec un tel code, on met en évidence un défaut courant des implémentations MPI *: les calculs sont sérialisés sur les différents nœuds car l'acquittement reçu en réponse à la requête asynchrone* `Isend()` *n'est pas pris en compte avant la fin du calcul et le traitement de la fonction* MPI `Wait()`.

Le recouvrement du calcul par les communications auquel on aurait pu s'attendre avec l'emploi de la primitive asynchrone `Isend()` *n'a pas lieu.*

FIG. 2.5 – Un code MPI parallèle exécuté en séquentiel

Les processus légers permettent d'exprimer le parallélisme intrinsèque des applications. Même si le matériel sous-jacent ne permet pas d'exploiter directement ce parallélisme[5], cette structuration a de nombreux intérêts. Elle permet, entre autres, de recouvrir facilement des temps d'attente lors des entrées/sorties : il suffit de basculer l'exécution vers un autre processus léger tant que l'événement attendu n'est pas survenu. L'application n'a plus à s'occuper elle-même des recouvrements.

Un second intérêt est la gestion facilitée de l'asynchronisme très fréquemment présent dans le calcul distribué. Une application n'est pas toujours capable de prévoir quand elle recevra une communication (message réseau), un signal, etc. L'utilisation de processus légers permet d'exprimer facilement le besoin de gérer ces événements asynchrones et d'y répondre rapidement.

Enfin, les processus légers permettent à plusieurs composants de l'application de fonctionner de manière indépendante. Cette capacité devient de plus en plus importante de nos jours avec le développement d'environnements de programmation ayant à effectuer des tâches indépendamment de l'application. L'absence de tel support conduit parfois à des

[5]Une machine monoprocesseur ne permet pas d'exécuter simultanément plusieurs flots d'exécution.

comportements contre-intuitifs comme, par exemple, la non progression des communications asynchrones dans MPI (voir Figure 2.5).

2.3 Discussion : que sait-on faire aujourd'hui, que manque-t-il ?

Les processus légers sont devenus un élément essentiel de nombreux supports exécutifs. Cependant, comme exposé plus en détail dans le chapitre suivant, il existe plusieurs types de processus légers avec chacun leurs qualités et leurs défauts. De ce fait, aucune bibliothèque de processus légers ne s'est encore imposée comme « la » bibliothèque à utiliser en toutes circonstances. Il s'agit ici d'identifier les besoins du calcul hautes performances et de mettre en évidence les lacunes dans les solutions actuelles.

2.3.1 Portabilité des performances

Parler de performances dans le calcul hautes performances semble une évidence. Mais que signifie cette notion appliquée à un support exécutif ? Il faut, bien sûr, que le support permette d'approcher les performances offertes par le matériel sous-jacent. Mais ce qui intéressera plus encore les applications, c'est la capacité qu'aura le support exécutif à offrir de bonnes performances quel que soit le matériel sur lequel il tourne. Il s'agit ici non plus de performance, mais de *portabilité des performances*. L'application ne doit pas nécessiter d'être modifiée ou adaptée pour fonctionner de manière efficace si la plate-forme d'exécution est changée. Ce doit être le travail du support exécutif seul. Et pourtant, cette exigence reste rarement satisfaite dans le domaine du multithreading en raison des nombreuses contraintes que nous allons détailler maintenant.

Diversité du matériel. Concernant le support matériel des flots d'exécution, les machines sont très variées, allant du plus simple avec les machines monoprocesseurs au plus compliqué avec, par exemple, des processeurs multicores multithreadés[6]. Exploiter efficacement ce matériel nécessite un comportement adapté de l'application : non seulement il faut demander au système d'exploitation les ressources adéquates, mais il faut également les exploiter de manière judicieuse. Ainsi, sur une machine NUMA, les processus légers doivent être correctement placés sur les différents processeurs en respectant les affinités mémoires. Mais comment effectuer cette répartition ? Actuellement, le placement des processus légers est fait soit automatiquement par le système d'exploitation (rarement optimal pour l'application), soit manuellement par l'application elle-même qui devient alors difficilement portable. Il faut l'adapter à chaque plate-forme matérielle.

Modules indépendants. Lorsque l'application est constituée de modules indépendants, il lui devient beaucoup plus ardu de gérer correctement ses ressources. En effet, un environnement de programmation ou un autre support exécutif peut être amené à créer de nouveaux processus légers pour gérer des communications de manière asynchrone. Or, un environnement de programmation a pour vocation d'abstraire et de cacher à l'application les détails de la plate-forme sous-jacente. L'application peut alors difficilement tenir compte de ces besoins lorsqu'elle demande au système les ressources adéquates.

[6] Il s'agit de composants contenant plusieurs processeurs logiques (multicores) chacun capable de gérer plusieurs flots d'exécution (multithreadés). INTEL devrait bientôt proposer de tels processeurs.

Ordonnancement contrôlé. Certaines applications, grâce à la connaissance de propriétés particulières de leurs tâches (durées, motifs de synchronisation, etc.), possèdent un ordonnancement optimal plus ou moins facilement prévisible. Si elles veulent utiliser des processus légers, ces applications doivent choisir entre utiliser les bibliothèques classiques et donc abandonner l'idée de diriger l'ordonnancement[7], ou bien développer une bibliothèque spécifique avec tout l'effort de développement que cela suppose mais qui respectera les choix d'ordonnancement de l'application.

Pour illustrer ce point, on peut prendre l'exemple d'un code développé au CEA-DAM où un domaine de calcul est subdivisé en sous-domaines. Le calcul complet consiste en la répétition de deux phases successives : l'une de calculs parallèles sur les sous-domaines et l'autre de communication entre sous-domaines. Le code, autrefois uniquement parallélisé avec MPI, utilise maintenant des processus légers pour exploiter des machines NUMA. Ils ont donc développé deux modèles[8] : le premier avec les processus légers du système où ils n'ont aucun contrôle sur l'ordonnancement et le placement des processus légers sur les processeurs, et le second à base de changement de contexte où ils gèrent tout manuellement. Cela leur permet d'exploiter au mieux les effets de cache dans les phases de calcul. Une application devrait être capable d'utiliser des processus légers tout en dirigeant l'ordonnancement de ses tâches si elle le désire.

Bilan. Les supports exécutifs doivent permettre à une application d'exploiter efficacement une vaste gamme de matériel et cela, sans nécessiter de modifier l'application. Dans le domaine du multithreading, cet objectif est encore difficile à atteindre :

– les bibliothèques de processus légers ne permettent pas aux applications d'exploiter avec efficacité mais de manière transparente les architectures matérielles variées ;
– les applications peuvent difficilement diriger l'ordonnanceur des bibliothèques de processus légers.

On s'aperçoit que la principale limitation des bibliothèques de processus légers est leur interface. Elle convient lorsqu'une application désire utiliser des processus légers pour exploiter une architecture donnée, mais elle est beaucoup trop limitée pour satisfaire un support exécutif. La portabilité des performances exigées des supports exécutifs nécessite des fonctionnalités supplémentaires.

2.3.2 Processus légers et communications

Outre la notion de flots d'exécution parallèles, le calcul parallèle hautes performances est également lié à la notion de réseau de communication rapide. L'utilisation conjointe de processus légers et de bibliothèques de communication a mis en évidence des difficultés d'intégration et a fait apparaître des propriétés qui se sont révélées cruciales alors qu'elles étaient jusque-là souvent négligées.

[7] La seule manière de diriger l'ordonnancement des processus légers classiques est d'utiliser des priorités. Elles ne sont cependant pas toujours disponibles et, lorsqu'elles le sont, il faut parfois avoir des privilèges pour les utiliser (il faut avoir les droits `root` sous LINUX).

[8] Ils ont également un troisième modèle qui utilise la bibliothèque de processus légers que nous avons développée. Ils bénéficient alors des fonctionnalités des processus légers (opérations de synchronisation, etc.) tout en étant capables de diriger leur ordonnancement.

2.3.2.1 Intégration de composants

En proposant simultanément des processus légers et des moyens de communication, les supports exécutifs se sont heurtés à des difficultés : faire coopérer efficacement des composants qui, tout en étant individuellement très performants, ne se prêtent pas nécessairement à leur intégration.

De nombreux travaux [Gin97, HCM94] ont proposé des supports exécutifs intégrant communication et processus légers en se contentant d'associer une bibliothèque de communication comme MPI ou PVM et une bibliothèque de processus légers. Les résultats ont souvent été décevants faisant apparaître des problèmes d'incompatibilité et de performances. La majorité des bibliothèques de communication, en effet, avait été écrite sans tenir compte d'éventuels processus légers. Il faut alors les rendre réentrantes, ce qui nécessite parfois des modifications profondes. Plus problématique se révèle l'utilisation, par les bibliothèques de communication, de fonctionnalités non ou mal supportées par les bibliothèques de processus légers. En définitive, on se rend compte qu'il est nécessaire de concevoir les différentes briques de base de manière conjointe si l'on souhaite que l'ensemble soit aussi performant que les parties individuelles.

2.3.2.2 Réactivité

La présence de processus légers n'est malgré tout pas suffisante pour qu'une bibliothèque de communication se comporte le mieux possible. Si l'on prend la situation de la bibliothèque MADELEINE sur une passerelle où des processus légers assurent la retransmission des messages entre les deux réseaux, il est important ici que les messages en transit soient traités rapidement pour ne pas faire patienter les deux autres nœuds communiquants. Une solution est de dédier cette passerelle au travail de communication. C'est ce qui a été retenu dans PACX-MPI [BGR97] qui utilise deux nœuds de chacune des grappes interconnectées pour réaliser les opérations d'entrée/sortie avec l'inconvénient de diminuer les ressources de calcul disponibles pour l'application. Une autre solution est de rendre le traitement de ces messages à renvoyer prioritaire sur le nœud passerelle. Une nouvelle notion prend alors toute son importance dans cet environnement : la notion de *réactivité*.

Définition. La réactivité d'une application à un ensemble d'événements extérieurs est la capacité de cette application à exécuter rapidement le traitement approprié dès l'arrivée de l'un de ces événements.

Dans le cas des communications, la réactivité est en quelque sorte la latence du réseau vue de l'application. Elle se démarque de la latence réseau qui se mesure généralement avec la bande passante lors des tests de performance d'une bibliothèque de communication. Ce n'est pas parce qu'un réseau a une faible latence que l'environnement applicatif qui l'utilise sera réactif. Pour reprendre notre exemple, même si un message devient disponible sur une interface réseau de la passerelle MADELEINE, le processus léger responsable du traitement de ce message et de son renvoi sur l'autre interface réseau ne sera pas nécessairement ordonnancé rapidement.

Applications à réactivité critique. Certaines applications souffriront plus que d'autres d'un manque de réactivité. C'est le cas, par exemple, des environnements de mémoire partagée distribuée (*Distributed Shared Memory*). Lorsqu'un nœud attend une page mémoire d'un autre nœud pour continuer, il est alors important que le second nœud réponde rapidement, même si lui-même a encore du travail. Ce besoin de réactivité était apparu clairement dans la plate-forme java distribuée Hyperion[ABH$^+$01a]. Les nœuds en attente de page mémoire étaient fortement pénalisés si les autres nœuds ordonnançaient des processus légers de l'application au lieu de répondre aux demandes.

Quelles solutions pour la réactivité actuellement ? Il n'existe pas de solution standard à ce problème. Les applications doivent s'adapter à chaque système et à chaque bibliothèque de communication. Il est possible d'utiliser les priorités des processus légers lorsqu'elles sont disponibles. Mais cette solution reste incomplète : avec une priorité trop forte (par rapport au nombre de processus légers de l'application), le processus léger réactif va prendre trop souvent la main et donc diminuer les performances de l'application. Avec une priorité trop faible, le processus léger réactif ne va pas prendre la main assez souvent pour être réellement réactif. La bonne priorité est donc variable et dépend du nombre de processus légers de l'application. Les priorités de processus légers ne sont donc ni complètement satisfaisantes, ni assez génériques pour être utilisées sur tous les systèmes.

2.3.3 Analyse et compréhension des performances

Il est assez aisé de suivre le déroulement d'un code de calcul séquentiel : la lecture du code source permet de comprendre assez facilement son évolution. Dans le cas de codes parallèles, cela devient beaucoup plus compliqué du fait de l'absence de prévisibilité du comportement de l'ordonnanceur. Il est impossible de prévoir *a priori* le déroulement du code. Avec des architectures complexes (NUMA, acrSMT, etc.), l'ordonnancement final de l'application parallèle dépend à la fois du matériel, des autres processus du système, des événements extérieurs, etc.

Or, si l'on veut analyser en détail le comportement d'une application, comme pour découvrir des goulots d'étranglement ou pour comprendre la portée réelle d'une optimisation, alors il est nécessaire de connaître le fonctionnement précis de l'application, par exemple en récoltant des traces au cours de son exécution. Malheureusement, la plupart des outils de traces et de profilage n'incluent pas (encore) la notion de processus léger. Or, la connaissance précise de l'ordonnancement des processus légers d'une application (tel processus léger sur tel processeur physique à tout instant) est parfois nécessaire pour comprendre des phénomènes (cache mal utilisé, etc.) Des efforts dans ce domaine sont donc requis.

2.4 Conclusion

Les nouvelles architectures matérielles ainsi que les environnements de programmation toujours plus complexes et puissants expliquent l'utilisation croissante des processus légers dans le calcul hautes performances. L'emploi de ce paradigme induit de nouveaux défis à relever si l'on veut offrir les meilleures performances aux applications. Une réflexion est nécessaire pour abstraire correctement les architectures physiques aux applications tout en leur permettant de les exploiter le plus efficacement possible. Les environnements doivent prendre en compte ce nouveau paradigme dès leur conception pour rester performants et aussi pour intégrer correctement les problématiques comme la réactivité. Enfin, des outils spécialisés restent encore à développer pour certaines tâches comme l'observation de programmes multithreadés.

Bien entendu, des solutions (partielles) à ces problèmes existent déjà. Le chapitre suivant sera l'occasion de les présenter en détail à travers l'étude des bibliothèques de processus légers existantes.

Chapitre 3

Les processus légers : état de l'art

Sommaire

Comme nous l'avons vu dans le chapitre précédent, les processus légers sont désormais souvent utilisés au sein des environnements de calcul parallèle. Leur diffusion est somme toute très large : on les retrouve même dans les agendas électroniques ! Tous les systèmes d'exploitation contemporains en proposent une implémentation et la plupart des langages offrent ce paradigme, soit de façon native pour les langages récents, soit par le truchement de bibliothèques plus ou moins standardisées pour les autres.

Alors que le recours aux processus légers s'est banalisé, des implémentations radicalement différentes co-existent. Ainsi, pour certains systèmes d'exploitation, on peut trouver de nombreuses bibliothèques encore activement développées. En fait, chaque implémentation a ses propres points forts, répondant ainsi à des objectifs précis. Aussi, aucun consensus sur la « bibliothèque de processus légers idéale » ne s'est-il dégagé.

Après avoir rappelé la « success story » des processus légers au travers des systèmes d'exploitation et des langages de programmation, nous ferons le point sur les caractéristiques essentielles des bibliothèques de processus légers. En particulier, nous présenterons les différents modèles existants en mettant en évidence les différences fondamentales qui les distinguent. Nous présenterons ensuite quelques implémentations de bibliothèques de processus légers. Ce sera l'occasion d'illustrer les concepts et caractéristiques présentés auparavant tout en mettant en lumière leurs points forts et limitations respectives.

3.1 Introduction aux processus légers

La notion de processus léger existe à la fois dans les systèmes d'exploitation et dans les langages. À travers quelques exemples tirés de l'histoire de l'informatique, nous montrons ici que cette notion est apparue pour répondre à des objectifs différents. Pour les systèmes, les processus légers ont permis d'exploiter les machines multiprocesseurs au sein d'un unique programme. Alors que pour les langages, c'est avant tout la recherche du moyen d'exprimer le parallélisme intrinsèque de l'application qui a été le moteur de l'introduction de processus léger.

3.1.1 Bref historique des processus légers dans les systèmes

La notion de processus léger, au sens de flot de contrôle séquentiel dans les systèmes d'exploitation, est très ancienne dans l'histoire des systèmes. On en trouve trace dès 1965 dans le Système à Temps partagé de Berkeley [1] (*Berkeley Timesharing System*[L$^+$66]), bien que le nom lui-même ne fût pas encore utilisé. Il était alors question de *processus*[Dij68] qui

[1] Le Système à Temps partagé de Berkeley[L$^+$66] a probablement été le premier système à temps partagé avec mémoire virtuelle paginée à la demande qui fut commercialisé.

Ce système protégeait la plupart des ressources, dont la mémoire, utilisateur par utilisateur. Chaque utilisateur pouvait avoir un accès exclusif ou partagé à au plus 128K mots de 24 bits. Chaque processus ne pouvait adresser que 16K mots à la fois. Les utilisateurs pouvaient avoir un grand nombre de processus où chacun d'entre eux pouvait lire et modifier sa propre configuration mémoire, ce qui permettait de partager arbitrairement des pages parmi celles accessibles à l'utilisateur courant. Les processus ne consommaient que très peu de ressources : l'état complet tenait dans 15 mots de 24 bits. La description d'un processus comprenait la configuration mémoire (les 8 pages du processus) et trois registres (accumulateur, registre d'index et compteur de programme).

interagissaient par l'intermédiaire de variables partagées, sémaphores ou autres moyens similaires.

Cependant, cette notion a été « oubliée » pendant plusieurs années au profit du développement d'UNIX et de sa notion de *processus*[2]. Un processus signifie alors un flot de contrôle séquentiel associé à un espace d'adressage virtuel privé. Comme les processus ne peuvent pas partager de mémoire[3] (chacun ayant son propre espace d'adressage), ceux-ci doivent interagir entre eux via des mécanismes spéciaux (tubes, signaux, etc.) ; la mémoire partagée n'a été introduite dans les systèmes que plus tardivement.

À la fin des années 70 ou au début des années 80, la demande de flots d'exécution indépendants au sein d'un même processus est apparue, en particulier dans le domaine du temps réel : les coûts de communication entre processus lourds étaient trop importants comparés à ceux des processus légers. Cela a conduit à la définition de la norme POSIX relative aux processus légers (voir ci-dessous) qui ont été progressivement implémentés dans tous les systèmes. L'apparition et le développement des machines SMP puis NUMA de nos jours a joué un rôle important pour la diffusion des processus légers : ils ont alors permis à un processus d'exploiter entièrement ces nouvelles machines à lui tout seul.

3.1.2 Apparition de la concurrence dans les langages de programmation

Systèmes d'exploitation et langages de programmation ont évolué de concert. Les premiers langages n'étaient qu'une traduction directe des instructions machines : à chaque famille de processeurs correspondait un langage assembleur. Ces langages sont toujours utilisés de nos jours : ils sont nécessaires pour exploiter les particularités d'un processeur donné, pour comprendre le fonctionnement précis d'un programme compilé, mais généralement, seuls les concepteurs de compilateurs ou de bibliothèques spécialisées s'intéressent à ces langages.

Très vite, le besoin de portabilité s'est fait sentir. Une application devait pouvoir fonctionner sur un ensemble de machines sans avoir à être entièrement réécrite. Grâce à l'introduction de la compilation, des langages plus évolués sont apparus rendant les programmes indépendants du processeur cible. Au fur et à mesure des recherches et des développements, de plus en plus de concepts ont été intégrés directement aux langages afin de permettre aux programmeurs de les utiliser sans avoir à se préoccuper de leur implémentation. Les exemples de tels concepts sont très variés et les décrire dépasse de loin l'objectif de ce document. On peut néanmoins citer les objets (avec les notions associées d'héritage, de méthodes, etc.), les fonctions d'ordre supérieur (à travers les langages ML), etc.

Le concept nous intéressant plus particulièrement est celui de la concurrence. Alors que les systèmes ont proposé du parallélisme au sein d'un même processus, les langages, eux, ont proposé du parallélisme dans la sémantique même des programmes.

Les processus légers peuvent être créés de différentes manières :

coroutines (le flot est dirigé explicitement) : Simula I, SL5, BLISS, Modula-2 ;
fork/join : PL/I, Mesa ;
cobegin/coend : ALGOL 68, CSP, Edison, Argus ;
déclaration de processus : DP, SR, Concurrent Pascal, Modula, PLITS, Ada

Et ils interagissent suivant différents paradigmes :

sémaphores : ALGOL 68
régions critiques conditionnelles : Edison, DP, Argus
moniteurs : Concurrent Pascal, Modula
passage de messages : CSP, PLITS, Gypsy, Actors
appel de procédure à distance (RPC) : DP, *Mod
rendez-vous : Ada, SR
transactions atomiques : Argus

Le site `http://ei.cs.vt.edu/~cs3304/lang.paradigms.html` *propose une classification des langages parallèles mettant en exergue les différents paradigmes utilisés.*

FIG. *3.1 – Langages, parallélisme et interface*

3.1.2.1 Les débuts

De nombreux langages ont introduit en leur sein le parallélisme sous des formes très variées comme l'illustre la Figure 3.1. Nous reprenons ici l'histoire de deux langages apparus vers la fin des années 70 : MODULA-2 et ADA.

Modula-2. Après avoir développé le langage PASCAL à l'École Polytechnique Fédérale de Zürich, Niklaus WIRTH lança vers 1973 un projet pour étudier les techniques et les problèmes de la multiprogrammation qui a donné naissance au langage expérimental MODULA [Wir77] implémenté sur un PDP-11. La base de la multiprogrammation était ici des moniteurs et des conditions (appelés ici signaux) comme proposés par C.A.R. HOARE [Hoa74], les moniteurs étant généralisés et intégrés à la notion de module. Ce projet conduisit à la conclusion qu'il était impossible de choisir une meilleure façon d'exprimer la concurrence : processus lourds ou légers, signaux et sémaphores, moniteurs et régions critiques, chacun avait des avantages et des inconvénients qui dépendaient de l'application ciblée.

Au cours des années 1978–1980, la conception à Zürich d'une station de travail, nommée LILITH, donna l'occasion à Niklaus WIRTH d'introduire le langage MODULA-2 successeur de MODULA.

Suivant les conclusions du projet MODULA, il fut décidé de n'inclure dans MODULA-2

On peut donc considérer qu'on a eu ici la première implémentation de processus légers gérés par le système d'exploitation.

[2] La FAQ du groupe de discussion `comp.os.research` offre des précisions supplémentaires sur cette évolution.

[3] Le fait pour les processus de ne pas partager de mémoire est l'une de leurs caractéristiques fondamentales. Cela assure une stricte indépendance entre les applications qui peuvent alors coexister sans conflit tout en partageant les ressources disponibles sur un même système.

que la notion fondamentale de coroutines ; les abstractions de plus haut niveau devraient être implémentées à partir de là sous forme de modules supplémentaires. Cette décision a été prise sachant que LILITH était destiné à être un système mono utilisateur, sans besoin complexe de multiprogrammation. Cependant, la croyance que les coroutines seraient une base solide pour la multiprogrammation, y compris pour les multiprocesseurs, n'était pas fondée, comme l'ont montré les concepteurs du langage MODULA-2+ [Rov86].

Ada. En 1974, le Ministère de la Défense américain désira mettre de l'ordre dans les technologies nécessaires à ses développements logiciels embarqués. Pour cela, les spécifications d'un langage idéal furent rédigées progressivement pour aboutir en juin 1978 à un cahier des charges.

9. Parallel Processing
9A. Parallel Processing. It shall be possible to define parallel processes. Processes (i.e., activation instances of such a definition) may be initiated at any point within the scope of the definition. Each process (activation) must have a name. It shall not be possible to exit the scope of a process name unless the process is terminated (or uninitiated).
9B. Parallel Process Implementation. The parallel processing facility shall be designed to minimize execution time and space. Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor.
9C. Shared Variables and Mutual Exclusion. It shall be possible to mark variables that are shared among parallel processes. An unmarked variable that is assigned on one path and used on another shall cause a warning. It shall be possible efficiently to perform mutual exclusion in programs. The language shall not require any use of mutual exclusion.
9D. Scheduling. The semantics of the built-in scheduling algorithm shall be first-in-first-out within priorities. A process may alter its own priority. If the language provides a default priority for new processes it shall be the priority of its initiating process. The built-in scheduling algorithm shall not require that simultaneously executed processes on different processors have the same priority. [Note that this rule gives maximum scheduling control to the user without loss of efficiency. Note also that priority specification does not impose a specific execution order among parallel paths and thus does not provide a means for mutual exclusion.]
9E. Real Time. It shall be possible to access a real time clock. There shall be translation time constants to convert between the implementation units and the program units for real time. On any control path, it shall be possible to delay until at least a specified time before continuing execution. A process may have an accessible clock giving the cumulative processing time (i.e., CPU time) for that process.
9G. Asynchronous Termination. It shall be possible to terminate another process. The terminated process may designate the sequence of statements it will execute in response to the induced termination.
9H. Passing Data. It shall be possible to pass data between processes that do not share variables. It shall be possible to delay such data transfers until both the sending and receiving processes have requested the transfer.
9I. Signalling. It shall be possible to set a signal (without waiting), and to wait for a signal (without delay, if it is already set). Setting a signal, that is not already set, shall cause exactly one waiting path to continue.
9J. Waiting. It shall be possible to wait for, determine, and act upon the first completed of several wait operations (including those used for data passing, signalling, and real time).

FIG. 3.2 – Spécifications des besoins en termes d'expression du parallélisme du Ministère de la Défense américain

Comme aucun langage existant ne satisfaisait aux spécifications, un concours fut organisé et, en 1979, le gagnant fut désigné : il s'agissait du langage Ada présenté par l'équipe de Jean ICHBIACH. La première version officielle du langage se nomme ADA 83, la seconde ADA 95. On peut trouver un historique plus complet de la naissance de ce langage dans [daC84].

Un programme Ada peut être constitué de plusieurs flots d'exécution indépendants, sauf aux points de synchronisation. Les tâches peuvent communiquer grâce aux différentes formes d'interaction prévues par le langage ou par l'accès à des données partagées dont la cohérence doit être assurée par l'utilisation des autres formes d'interaction. En pratique, les supports d'exécution pour le langage ADA utilisent les processus légers pour implémenter les tâches. On peut noter qu'une des premières bibliothèques de processus légers développées l'a été pour un support exécutif d'ADA (voir ci-dessous 3.4.1.1).

3.1.2.2 La concurrence dans les langages actuels

Comme nous l'avons vu, la notion de tâche ADA correspond aux processus légers des systèmes d'exploitation ; de plus, les processus légers sont également utilisés dans les supports exécutifs des langages successeurs de MODULA-2 tels que MODULA-2+, MODULA-3, OBERON-2, etc. Cependant, l'adéquation entre le modèle de processus léger et la notion de concurrence du langage n'est pas toujours aussi directe. Nous allons voir à travers quelques exemples comment peut s'articuler la relation entre ces deux notions.

OpenMP. Ce langage est une extension du langage C. Grâce à des directives insérées par le programmeur dans un code séquentiel et à une analyse des dépendances entre les données, un compilateur OPENMP est capable d'extraire du parallélisme des boucles de calcul en vue d'une exécution sur machine multiprocesseurs. Or, les processus légers sont souvent utilisés par les systèmes d'exploitation des machines multiprocesseurs. Ainsi, même sans apparaître explicitement dans le langage, les processus légers sont nécessaires pour les supports exécutifs d'OPENMP.

Athapascan1. Ce modèle de programmation a pour but de permettre une programmation simple, portable et efficace d'applications parallèles. Ces dernières sont décomposées explicitement en tâches de calculs communiquant entre elles par l'intermédiaire d'objets en mémoire partagée. La sémantique des accès aux données partagées est quasi séquentielle et les précédences entre les tâches sont implicitement définies pour respecter cette sémantique. Ces dépendances de données entre tâches sont analysées à l'exécution ; le flot de données se construit donc dynamiquement. On trouvera dans [Gal99] une description et une analyse théorique de ce modèle, dans [Dor99] une présentation de l'environnement de programmation avec son application à la factorisation creuse de Cholesky entre autres, et dans [Cal99] une présentation du support exécutif d'ATHAPASCAN1.

Comme dans le cas OPENMP, la notion de processus léger n'apparaît pas explicitement. Cependant, pour gérer ses flots de contrôles parallèles, le support exécutif d'ATHAPASCAN1 utilise lui aussi des processus légers. Ces derniers permettent d'abstraire pour ATHAPASCAN1 la notion de processeur virtuel et fournissent ainsi des primitives classiques pour gérer des flots parallèles au sein d'une application.

Cilk. Le but de ce langage est de permettre l'écriture de programmes parallèles efficaces quelle que soit l'architecture sous-jacente, en particulier le nombre de processeurs de la machine. CILK-5 [FLR98, BJK+95, BL94] est une extension du langage C permettant de lancer des tâches de manière extrêmement efficace : le surcoût du lancement d'une nouvelle tâche n'est que de 2 à 6 fois le coût d'appel d'une fonction C classique.

Pour atteindre une telle efficacité, seuls quelques processus légers, créés au démarrage du programme, s'exécutent jusqu'à sa fin et aucune synchronisation entre eux n'est utilisée au cours du programme : ce ne sont que des « *containers* » du point de vue de CILK. Les tâches ou fonctions à exécuter en parallèle sont rassemblées dans une queue de travaux. Ceux-ci sont élus puis exécutés dès qu'un processeur est disponible. Afin de minimiser les surcoûts dus à la parallélisation (création de tâche, synchronisation à la terminaison), les fonctions sont compilées en deux versions : une parallèle et une séquentielle. Le support exécutif se charge de choisir dynamiquement la bonne version à utiliser selon le contexte. Enfin, des mécanismes de synchronisation très efficaces (sans verrou) sont utilisés pour régir l'accès aux queues de travaux.

Cette approche permet une parallélisation de programme très efficace et pratiquement sans surcoût sur machine monoprocesseur. Cependant, cela nécessite d'adopter le style de programmation de CILK : une tâche ne doit pas se synchroniser directement avec une autre, mais utiliser une continuation pour attendre un autre résultat. CILK est donc peu utilisable pour implémenter, par exemple, un support exécutif pour ADA. Le modèle de programmation imposé est trop contraignant pour cela.

3.1.3 Bilan

Les processus légers sont apparus progressivement dans les systèmes d'exploitation pour finalement être présents dans chacun d'entre eux. Ils fournissent, en effet, une abstraction permettant au système d'offrir aux applications du parallélisme entre plusieurs flots d'exécution. Ces possibilités fournies par les systèmes sont exploitées par les langages, soit de manière plus ou moins transparente en identifiant les processus légers du système et une notion (tâche, *thread*, etc.) du langage, soit en proposant un autre modèle pour le parallélisme mais en s'appuyant sur les processus légers des systèmes pour implémenter le support exécutif. Seuls quelques langages parallèles avec des modèles de programmation bien particuliers implémentent un autre type de support pour gérer le parallélisme de l'application. Toutefois, leur modèle de programmation qui leur permet d'être efficace est trop spécifique pour que les environnements logiciels classiques s'y adaptent. Un support exécutif pour applications parallèles se voulant générique doit nécessairement proposer une notion de processus léger.

3.2 Terminologie et caractéristiques

3.2.1 Mais quelle est la définition exacte d'un processus léger ?

La signification exacte du terme *processus léger* ou *thread* (anglicisme désormais fréquent dans la communauté) n'est pas tout à fait consensuelle. Une définition possible serait « encapsulation d'un flot d'exécution dans un processus »[4].

[4] Il s'agir là des processus légers du point de vue applicatif. On peut également parler de « *thread* noyau » dans les systèmes d'exploitation, mais dans ce cas, il s'agit des entités ordonnancées par l'ordonnanceur noyau

D'un point de vue matériel, un processus léger possède très peu de ressources propres, à savoir un identifiant, une pile et un jeu de registres. Toutes les autres ressources appartiennent au processus (lourd) et sont partagées par les processus légers.

3.2.2 Les principales familles de processus légers et la norme POSIX

Il existe plusieurs familles de processus légers[5]. Nous les présentons ici rapidement puis nous discutons sur la famille la plus courante de nos jours.

Les processus légers "à la POSIX". Cette famille se compose de trois sous-groupes :

Les "vrais" processus légers POSIX, basés sur le standard IEEE POSIX 1003.1C-1995 (également connu sous le nom ISO/IEC 9945-1:1996) qui constitue une partie du standard ANSI/IEEE 1003.1, édition 1996. Ce standard traitant des processus légers à la section 1003.1c, les bibliothèques qui le suivent sont également appelées PTHREADS, POSIX.1C THREADS ou encore *draft* 10 de POSIX.1C puisque c'est la 10e proposition qui est devenue le standard.

Les processus légers DCE, basés sur la quatrième proposition (*draft* 4) du standard POSIX concernant les processus légers. Quelques implémentations d'UNIX proposent de telles implémentations, mais elles ont tendance à disparaître au profit de la norme POSIX actuelle.

Les processus légers INTERNATIONAUX UNIX (UNIX *International (UI) threads*), également connus sous le nom de *threads* SOLARIS, basés sur le standard UNIX INTERNATIONAL THREADS. Ce standard est très proche de POSIX. Les deux seuls principaux distributeurs d'UNIX qui supportent les processus légers UI sont SOLARIS 2 de SUN et UNIX WARE 2 de SCO.

Tous ces standards sont en fait assez proches les uns des autres. Les deux derniers (DCE et UI) sont pratiquement compatibles POSIX mais offrent quelques fonctionnalités supplémentaires ou comportements différents. Par exemple, un appel à `fork()` sous SOLARIS duplique tous les processus légers, alors que POSIX ne duplique que le flot d'exécution qui a appelé `fork()`. Le comportement vis-à-vis des signaux `TIMER` est également légèrement différent. Enfin et surtout, le modèle UI a une interface plus fournie de manière à contrôler le modèle de processus légers plus complexe proposé par SOLARIS (voir la section 3.4.3.1 pour plus de précisions sur ce modèle). Aussi, un peu de travail est-il nécessaire pour convertir les processus légers DCE ou UI en "vrais" processus légers POSIX.

Les processus légers MICROSOFT. MICROSOFT a développé deux bibliothèques de processus légers : les **processus légers OS/2** pour le système d'exploitation d'IBM OS/2 et les **processus légers WIN32** pour les systèmes d'exploitation MICROSOFT WINDOWS 95 et WINDOWS NT. Convertir un programme de l'une à l'autre de ces interfaces nécessite un travail d'adaptation.

(processus, démons internes, etc.) et la définition précédemment donnée ne s'applique pas vraiment. On pourrait la généraliser en « encapsulation d'un flot d'exécution dans une entité possédant un espace d'adressage », l'entité étant le processus pour une application et le noyau pour le système d'exploitation.

[5]On pourra se rapporter à la FAQ du groupe de discussion `comp.programming.threads` pour plus de précision.

Avec le dernier système d'exploitation WINDOWSXP de MICROSOFT, les processus légers sont offerts au travers des interfaces WIN32 et POSIX. Des adaptations pour les systèmes antérieurs, comme WINDOWS 95, sont également disponibles.

Discussion sur la norme POSIX

L'émergence de la norme POSIX répond à la volonté de créer un ensemble d'API s que toutes les implémentations d'UNIX pourraient garantir. Une API définissant le modèle multithreads fut alors proposée et acceptée en tant que brouillon (*draft*) rattaché à une extension temps réel. Les processus légers étant peu répandus, il s'agissait surtout de constituer un cadre de travail commun avant l'apparition d'implémentations concurrentes. Depuis, l'API des processus légers POSIX a connu de nombreuses révisions avant d'être officiellement intégrée.

Cet effort de normalisation a porté ses fruits puisque l'interface définie correspond à la plupart des besoins des applications tout en étant suffisamment souple pour ne pas imposer de choix d'implémentation trop contraignant. Ainsi, il est extrêmement rare de trouver des bibliothèques de processus légers dont l'interface n'est pas semblable à celle de cette norme puisque cela permet de compiler un vaste ensemble d'applications grâce à la compatibilité au niveau source. Notons cependant que de larges parts de ce standard demeurent optionnelles[6]. C'est pourquoi un programmeur désirant utiliser des processus légers de façon portable est contraint de se limiter aux fonctionnalités essentielles de cette API. Parfois même, des parties obligatoires de la norme ne sont pas correctement implémentées dans les bibliothèques des systèmes. C'est le cas en particulier de la gestion des signaux particulièrement complexe à mettre en œuvre conformément au standard : les signaux doivent être partagés au sein du processus, mais chaque processus léger peut posséder son propre masque.

La norme POSIX n'est pas très loquace en ce qui concerne certaines fonctionnalités avancées, comme le contrôle de l'application sur l'ordonnancement. Ceci constitue *a priori* un atout : une spécification plus avancée aurait certainement réduit ou figé les possibilités d'implémentation du standard. Néanmoins, les programmeurs d'applications de calcul, connaissant le schéma de calcul/communications, peuvent le regretter : pouvoir spécifier l'ordonnancement en alternant, par exemple, des phases de calculs et des phases de synchronisations/communications permet d'optimiser l'utilisation du cache du processeur et donc d'augmenter les performances.

En définitive, la norme POSIX est devenue le standard incontournable en matière de processus légers puisque la plupart des systèmes d'exploitation UNIX et même d'autres (WINDOWSXP, etc.) en proposent désormais une implémentation et que de nombreuses applications et bibliothèques l'utilisent. Ceci assure une bonne qualité de portabilité. Toutefois, le faible nombre de fonctionnalités garanties explique la prolifération de bibliothèques de processus légers qui, tout en restant plus ou moins compatibles avec la norme POSIX, proposent des extensions pour des besoins particuliers (ordonnancement dirigé, migration de processus légers entre machines, etc.). De nombreux codes peuvent alors être facilement portés vers ces bibliothèques et légèrement modifiés pour tirer parti de ces nouvelles fonctionnalités.

[6]POSIX définit plusieurs ensembles de fonctionnalités, les principales (création, synchronisation, destruction, etc.) étant, bien sûr, obligatoires. Pourtant, des fonctionnalités plus avancées, comme les classes d'ordonnancement, ne sont pas requises par la norme. Des constantes permettent de préciser les options implémentées.

3.2.3 Définitions

Nous présentons ici quelques notions usuelles du monde des processus légers. Les premières caractérisent plus particulièrement les bibliothèques elles-mêmes. Les suivantes s'appliquent davantage aux paradigmes de programmation utilisés en environnement multithreadé.

3.2.3.1 Caractérisations des bibliothèques de processus légers

Degré de parallélisme : nombre de flots d'exécution susceptibles de s'exécuter réellement simultanément. Il est, bien sûr, borné par le nombre de processeur(s) de la machine[7]. Cependant, une bibliothèque donnée peut, par son implémentation, limiter ce degré de parallélisme, la plupart du temps en n'autorisant alors qu'un seul flot d'exécution à la fois.

Il est à noter que la norme POSIX n'impose rien à ce sujet au niveau des implémentations. En revanche, les applications conformes doivent être écrites de manière à supporter un degré de parallélisme quelconque.

Concurrence : fait de considérer plusieurs flots d'exécution indépendants. La concurrence est une caractéristique essentielle de la programmation avec des processus légers. Les applications utilisant ce paradigme doivent protéger leurs données des accès concurrents si nécessaire. Remarquons que la concurrence n'implique pas nécessairement un degré de parallélisme supérieur à 1. Un système monoprocesseur, où l'ordonnanceur fait alterner les flots d'exécution, provoque de la concurrence : tous les flots progressent indépendamment les uns des autres, sans aucune garantie sur leur entrelacement réel.

Préemption : fait de passer régulièrement la main d'un flot d'exécution à un autre sans indication particulière dans les flots d'exécution eux-mêmes. Dans une bibliothèque non préemptive, un processus léger gardera la main jusqu'à ce qu'il bloque, par exemple en voulant acquérir un verrou, ou qu'il cède la main volontairement, comme avec une primitive dédiée telle que `yield()`.

Là encore, la norme POSIX laisse le choix aux implémentations d'être préemptives ou non. Pour une bibliothèque de processus légers à usage général, les programmeurs préfèrent généralement une bibliothèque préemptive. Cependant, des bibliothèques non préemptives ont été développées et existent encore car elles peuvent se révéler plus efficaces ou bien l'usage qui leur est destiné rend inutile la préemption. Par exemple, avec un programme utilisant des processus légers uniquement pour séparer plusieurs calculs indépendants, la préemption aurait pour effet de rajouter du code (les changements de contexte) et surtout de diminuer l'utilisation du cache du processeur en le forçant à changer de tâche au cours des calculs.

Il peut être intéressant de noter qu'un système non préemptif et avec un unique flot de contrôle à tout instant constitue le paradigme appelé *coroutine* sur lequel reposent des langages comme MODULA-2 (voir la section 3.1.2.1). Les problèmes de synchronisation et de partage de ressources qui surviennent en environnement multithreadé sont le plus souvent trivialement résolus par les programmeurs de coroutines. En revanche, l'unique flot

[7] Plus exactement, le degré de parallélisme est borné par la somme des nombres de flots d'exécution gérés par les processeurs. En effet, un processeur multithreadé peut être capable de gérer 2, 4, ou plus flots d'exécution simultanément.

de contrôle a pour conséquence d'empêcher l'exploitation des machines multiprocesseurs ; c'est pourquoi les coroutines ne seront pas traitées plus en détail dans ce document.

3.2.3.2 Propriétés et notions relatives aux processus légers

Quelques notions reviennent fréquemment lorsque l'on discute de bibliothèques de processus légers. Cette section a pour objectif de définir certaines d'entre elles afin de faciliter la lecture de la suite de ce document.

Section critique : région du programme où l'on souhaite limiter (généralement à un seul) le nombre de flots d'exécution. Ces régions correspondent généralement à des zones où des invariants (approximés) sur des données peuvent ne pas être respectés. Par exemple, si l'on gère une structure avec une liste d'objets et un compteur des objets de cette liste, un invariant pourra être que le compteur contient le nombre exact d'objets de la liste. La section critique pourra ainsi contenir les primitives d'ajout ou de retrait d'objets de la liste.

Il existe de nombreuses techniques basées sur les outils de synchronisation pour implémenter les sections critiques. Les plus classiques invoquent des moniteurs (verrous et conditions) ou des sémaphores. Les sections critiques réduisent le degré de concurrence de certaines parties du code mais sont essentielles pour garantir un fonctionnement correct du programme lorsque l'on doit partager des données entre plusieurs processus légers. Écrire correctement et efficacement ces sections critiques est généralement la partie difficile de la programmation multithreadée. Cela nécessite généralement une compréhension fine de la sémantique du code hors de portée des outils actuels de parallélisation automatique.

Attente active : se dit d'une méthode de synchronisation où l'entité, attendant de passer cette synchronisation, garde le contrôle du processeur. Dans un environnement multithreadé, le programmeur évitera, dans la mesure du possible, d'utiliser des mécanismes d'attente active et préférera donner la main à un autre processus léger prêt.

Néanmoins, l'attente active peut parfois être nécessaire. Cela se produit généralement dans les couches internes d'une bibliothèque de synchronisation. Les primitives de synchronisation de haut niveau gérant les files d'attente sur une ressource donnée tel qu'un verrou ont besoin de sections critiques. Pour gérer la contention, elles peuvent alors soit reposer sur les primitives du système d'exploitation qui sont coûteuses (nécessité d'un appel système), soit faire de l'attente active en se disant que le flot actuellement dans la section critique la quittera bientôt. Ceci est généralement vrai si le flot en question est actuellement ordonnancé sur un processeur, faux sinon. C'est pourquoi la plupart des bibliothèques de synchronisation utilisent généralement un peu d'attente active pour leur synchronisation interne, puis en cas d'échec au bout d'un laps de temps, elles demandent volontairement au système de passer la main à d'autres flots d'exécution grâce, par exemple, à la fonction `sched_yield()`.

Atomique : se dit d'une instruction dont le résultat de l'exécution ne dépend pas de l'entrelacement avec d'autres instructions. Des instructions simples comme une lecture ou une écriture en mémoire sont atomiques. Cependant, les instructions atomiques complexes sont souvent plus intéressantes. Ainsi, l'incrémentation d'une variable n'est généralement pas une instruction atomique : une instruction dans un autre flot d'exécution en parallèle peut modifier la variable entre sa lecture et son écriture incrémentée.

Tous les processeurs actuels fournissent des instructions spéciales permettant de réaliser des opérations de manière atomique. Les plus classiques sont les incrémentations/décrémentations de variables, les instructions *Test And Set* (TAS) pour tester et modifier une variable et les instructions d'échange conditionnel de valeur (*Compare and Exchange*). Ces instructions processeurs atomiques font généralement intervenir des protocoles spéciaux sur les bus afin de garantir l'atomicité de l'opération même sur des machines multiprocesseurs.

Ces instructions atomiques dépendent fortement du processeur. Pour une même architecture, différentes primitives peuvent être disponibles selon le modèle de processeur ciblé. Les codes les utilisant sont donc écrits directement en assembleur et sont non portables. Il est néanmoins important de réaliser que toutes les méthodes de synchronisation, en définitive, reposent à un niveau ou à un autre sur ces instructions atomiques.

Réentrance : fait, pour une fonction, de pouvoir être exécutée pendant son exécution. Une fonction sans effet de bord et travaillant uniquement avec des variables locales est de fait automatiquement réentrante.

On distingue principalement deux types de réentrance :

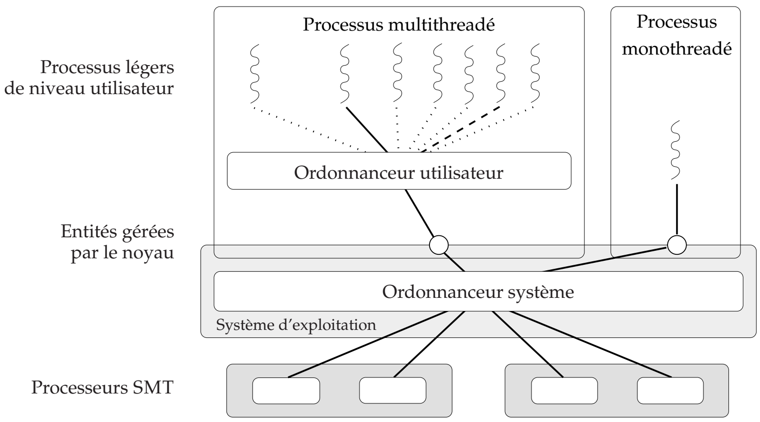
réentrance par rapport aux signaux : la fonction peut être utilisée dans un traitant de signal. Pour ce faire, une méthode consiste à bloquer les signaux pendant le déroulement de la fonction. En règle générale, la plupart des fonctions qui ne sont pas naturellement réentrantes ne le sont pas vis-à-vis des signaux. Un code réentrant par rapport aux signaux sera dit *async-safe* (*asynchronous safe*) ;

réentrance par rapport aux processus légers : la fonction peut être utilisée dans plusieurs processus légers simultanément. De nos jours, la plupart des bibliothèques sont réentrantes pour les processus légers : elles protègent leurs données globales avec les primitives POSIX de synchronisation de processus légers. Un code réentrant par rapport aux signaux sera dit *thread-safe*.

Notons que la simple réentrance d'un code par rapport aux processus légers ne suffit pas à qualifier ce code de *MT-safe* (*MultiThread safe*). Pour cela, il est en outre nécessaire que le code passe raisonnablement à l'échelle, c'est-à-dire accepte un certain niveau de réelle concurrence. Protéger tous les accès à une bibliothèque avec un verrou permet de la rendre facilement réentrante vis-à-vis des processus légers, cependant ce code ne sera alors pas qualifié de *MT-safe* : à tout instant, un seul processus au plus peut exécuter du code de cette bibliothèque.

3.3 Les différents types de processus légers

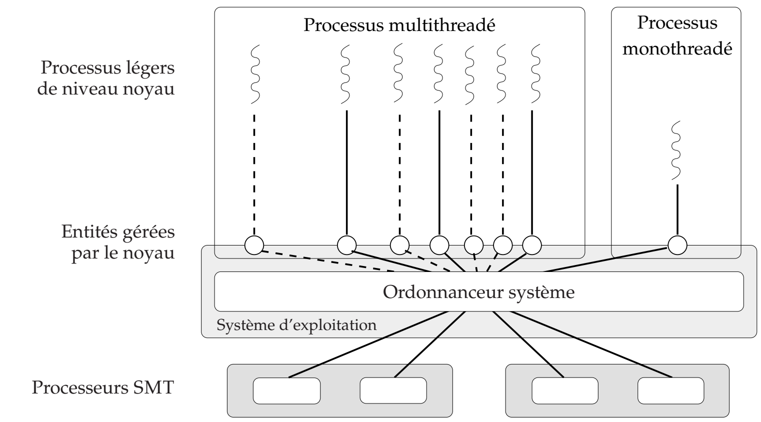
Le niveau auquel est implémenté l'ordonnanceur constitue la caractéristique intrinsèque la plus importante des bibliothèques de processus légers : les processus légers peuvent être gérés dans une bibliothèque entièrement en espace utilisateur, ou bien au sein même du noyau (en espace protégé) avec l'aide éventuelle d'une bibliothèque en espace utilisateur. Ce choix de conception, fondamental, a des répercussions importantes sur de nombreuses caractéristiques et propriétés de la bibliothèque.



(a) Bibliothèque de niveau utilisateur

3.3(a) : *Le système n'intervient pas pour gérer les processus légers. Ne les connaissant pas, il ne peut pas allouer plusieurs processeurs à l'application.*

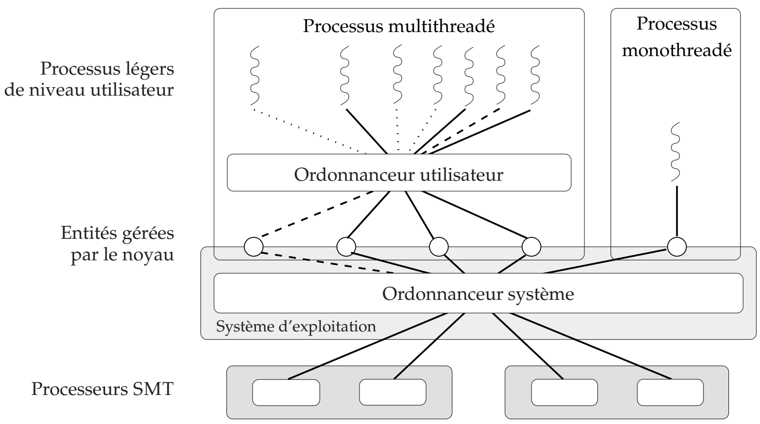
Un appel système bloquant interrompt toute l'application.



(b) Bibliothèque de niveau noyau

3.3(b) : *Les manipulations des processus légers font intervenir le système qui peut les répartir sur les processeurs disponibles.*

Un appel système bloquant ne suspend que le processus léger qui l'a exécuté.



(c) Bibliothèque mixte

3.3(c) : *Les processus légers sont gérés par l'application, mais cette dernière dispose de plusieurs flots d'exécution parallèles pouvant occuper les processeurs.*

Un appel système bloquant retire un des flots d'exécution disponible pour l'application.

FIG. 3.3 – Les différents types de processus légers

3.3.1 Présentation des architectures

Trois principales architectures illustrées sur la Figure 3.3 peuvent exister concernant les bibliothèques de processus légers :

architecture utilisateur : les processus légers sont gérés entièrement en espace utilisateur. Le noyau n'a pas connaissance des processus légers de l'application. L'ordonnanceur fait partie d'une bibliothèque en espace utilisateur ;

architecture noyau : les processus légers sont gérés par l'ordonnanceur du noyau. Ce dernier connaît donc exactement les processus légers de l'application ;

architecture mixte : c'est un mélange des deux précédentes propositions. Un ensemble de processus légers gérés par le noyau exécutent des processus légers de niveau utilisateur. Deux ordonnanceurs interviennent : celui du noyau et celui de l'application en mode utilisateur. Le noyau ne connaît donc pas exactement les processus légers de l'application, seulement les processus légers propulseurs (LWP).

Terminologie. Il convient de précisé définitivement le vocabulaire employé dans la suite de ce document. Les termes anglais sont indiqués entre parenthèses.

processus léger [de niveau] utilisateur (*user[-level] thread*) : processus léger géré par un ordonnanceur fonctionnant en mode utilisateur.

processus léger [de niveau] noyau (*kernel[-level] thread*) : processus léger géré par l'ordonnanceur noyau.

LWP (*light-weight process*) : processus léger de niveau noyau servant à ordonnancer des processus légers de niveau utilisateur. Ce terme est principalement utilisé dans le modèle mixte. Un LWP peut également être désigné sous le nom de **Processus léger de poids moyen**.

3.3.2 Architectures et caractéristiques

Le choix d'une architecture particulière a de nombreuses répercussions sur les caractéristiques de l'implémentation considérée. Les principales sont présentées ci-dessous et synthétisées dans le tableau 3.1.

3.3.2.1 Performances

Une bibliothèque de niveau utilisateur aura de bien meilleures performances qu'une bibliothèque noyau en ce qui concerne les opérations sur les processus légers telles que leur création, destruction et synchronisation. En effet, les processus légers étant entièrement gérés en espace utilisateur, aucune opération ne nécessite l'intervention du noyau. En revanche, pour une bibliothèque de niveau noyau, l'application doit nécessairement informer le système à chaque opération susceptible de modifier le comportement de l'ordonnanceur, en particulier lors des ajouts, suppressions ou suspensions de processus légers.

On peut noter que pour améliorer ses performances, une bibliothèque de niveau noyau cherchera à limiter au maximum le coûteux basculement du mode utilisateur au mode noyau, en effectuant le maximum d'opérations en mode utilisateur. Les performances grandement améliorées de la nouvelle bibliothèque de niveau noyau NPTL sous LINUX sont principalement dues à un effort de conception important visant à limiter autant que possible les basculements en mode noyau. Par exemple, l'acquisition d'un verrou sans contention se fait désormais en espace utilisateur[8] ; bien entendu, s'il y a contention, le noyau doit nécessairement intervenir puisque l'ordonnanceur doit stopper le processus léger.

3.3.2.2 Flexibilité

Il est plus simple de modifier une bibliothèque de niveau utilisateur qu'une bibliothèque de niveau noyau : en mode noyau, la gestion de la mémoire, des entrées/sorties, etc. est très différente de ce que l'on trouve en espace utilisateur. Beaucoup de fonctionnalités ne sont pas présentes[9]. Développer en espace utilisateur permet de bénéficier éventuellement d'un grand nombre de bibliothèques et réduit les contraintes que l'on pourrait subir quant à l'allocation de mémoire.

De plus, en mode utilisateur, comme l'ordonnanceur des processus légers est local au processus, le modifier ne changera pas le comportement des autres applications du système. Dans le cas d'une bibliothèque de niveau noyau, modifier l'ordonnanceur système est toujours délicat et influe sur le système tout entier. Il faut alors respecter des contraintes supplémentaires : quelques tâches du noyau doivent, par exemple, toujours être ordonnancées en priorité.

3.3.2.3 Machines multiprocesseurs

Les processus légers permettent d'exprimer un parallélisme entre les diverses tâches d'une application. Sur une machine multiprocesseur où un réel parallélisme est possible, un processus peut en tirer partie pour exploiter pleinement la machine. Cependant, les processeurs d'une machine sont gérés et alloués par le système d'exploitation. C'est pourquoi seuls les processus légers de niveau noyau ou mixte sont capables d'exploiter simultanément plusieurs processeurs d'une machine. Avec une bibliothèque de processus légers de niveau purement utilisateur, le système d'exploitation ignore l'existence des multiples flots d'exécution dans le processus. Il n'alloue donc au plus qu'un processeur à l'application.

Les bibliothèques mixtes permettent l'exploitation des machines multiprocesseurs tout en gardant des processus légers de niveau utilisateur (donc efficaces et flexibles), ceci grâce aux LWP qui peuvent occuper chacun un processeur différent.

3.3.2.4 Appels systèmes bloquants

Une autre caractéristique importante des bibliothèques est leur capacité à ordonnancer ou non un autre processus léger lorsque l'un d'entre eux se bloque dans le noyau, par exemple en attente d'entrées/sorties. Avec une bibliothèque de niveau noyau, le système d'exploitation connaît les processus légers de l'application (c'est lui qui les ordonnance). À l'occasion d'un appel système bloquant, il peut donc donner la main à un autre flot d'exécution de l'application. En revanche, avec une bibliothèque de niveau utilisateur, le noyau ne

[8] La synchronisation de processus légers de niveau noyau en espace utilisateur lorsqu'il n'y a pas de contention est possible grâce au nouveau mécanisme nommé FUTEX introduit dans les noyaux LINUX récents [Rus02b].

[9] Aucune fonction de la bibliothèque C n'est présente dans le noyau. En effet, le code noyau n'est pas lié avec les bibliothèques classiques de l'espace utilisateur.

connaissant pas les processus légers de l'application, un seul appel système bloquant dans l'application va bloquer l'ensemble de ses flots d'exécution.

Avec une bibliothèque mixte, lorsqu'un processus léger de niveau utilisateur effectue un appel bloquant dans un LWP, ce dernier est bloqué, mais les autres LWP peuvent continuer à exécuter les autres processus légers de l'application. Si l'application s'attend à faire de nombreux appels systèmes bloquants, elle peut demander la création d'un plus grand nombre de LWP que de processeurs. Toutefois, lorsqu'aucun des LWP n'est bloqué, la bibliothèque peut alors souffrir de mauvaises performances : les synchronisations internes se font généralement par attente active et si le LWP d'un processus léger possédant un verrou perd la main au profit d'un autre demandant le même verrou, ce dernier devra attendre que le premier soit réordonnancé. Le plus efficace pour une bibliothèque mixte est d'avoir autant de LWP en exécution que de processeurs physiques sur la machine.

3.3.3 Bilan

	Caractéristiques			
Bibliothèque	Performances	Flexibilité	SMP	Appels systèmes bloquants
Utilisateur	+	+	-	-
Noyau	-	-	+	+
Mixte	+	+	+	limités

TAB. 3.1 – Principales caractéristiques des différents types de processus légers

Comme on peut le voir dans le tableau 3.1, aucun modèle n'est parfait. Parfois, peu importe car les fonctionnalités manquantes ne sont pas utilisées par l'application développée. En revanche, pour le calcul hautes performances parallèles, il paraît souhaitable de chercher à concilier performance, flexibilité et exploitation des machines multiprocesseurs.

Les processus légers de niveau utilisateur ont pour eux des performances excellentes ainsi qu'une grande flexibilité pour effectuer des développements particuliers ou offrir des fonctionnalités non standards, mais ils sont incapables d'exploiter les machines multiprocesseurs ni de se comporter correctement avec les appels systèmes bloquants. À l'inverse, les processus légers de niveau noyau tirent partie des machines multiprocesseurs et gèrent correctement les appels systèmes bloquants au sein des processus légers, ceci au prix d'une implémentation moins efficace et surtout beaucoup plus figée. La plupart des systèmes sont livrés en standard avec ce type de processus légers qui permettent d'exploiter pleinement les ressources matérielles (processeurs) de la machine.

Les bibliothèques à deux niveaux semblent les plus intéressantes puisqu'elles réunissent le maximum d'avantages. Cependant, leur complexité inhérente, due à la nécessité de gérer simultanément deux ordonnanceurs différents, associée à leur manque de maturité (certains problèmes ne sont pas résolus, en particulier ceux concernant les appels systèmes bloquants) font que ces bibliothèques ne sont pas encore très répandues.

En fin de compte, chaque modèle présente suffisamment d'avantages pour qu'il ne soit pas complètement abandonné. C'est pourquoi on trouve de nos jours, sur les systèmes, plusieurs implémentations différentes de bibliothèques de processus légers. Pourtant, grâce à la standardisation, le choix de l'utilisation de l'une ou l'autre de ces bibliothèques pour une application donnée ne devrait plus dépendre du programmeur mais s'effectuer au moment de l'exécution de celle-ci.

3.4 Présentation de quelques bibliothèques de processus légers

La variété et la difficulté des compromis à faire inévitablement entre efficacité, flexibilité, richesse fonctionnelle et portabilité justifient le grand nombre de bibliothèques de processus légers encore présentes sur les systèmes actuels. Nous avons ici choisi de présenter quelques bibliothèques que nous avons jugées représentatives, en privilégiant celles fonctionnant sur le système d'exploitation LINUX, non pas seulement pour des questions de commodités d'installation, mais aussi à des fins de comparaisons objectives.

3.4.1 Bibliothèques de niveau utilisateur

Les bibliothèques de niveau utilisateur sont les premières apparues sur les systèmes sans doute parce que, comme nous l'avons déjà évoqué, il est plus facile de travailler en espace utilisateur que de modifier le système d'exploitation. Malgré leurs limitations, comme l'impossibilité d'exploiter les machines multiprocesseurs, certaines bibliothèques de niveau utilisateur sont encore utilisées et développées en raison, entre autres, de leur facilité à les adapter à un besoin particulier et, n'ayant pas à dialoguer avec le système d'exploitation, de leur efficacité.

3.4.1.1 FSU Pthreads

Conçue par Frank MUELLER, la bibliothèque de processus légers de niveau utilisateur FSU PTHREADS [Mue93] est écrite de manière relativement portable ; aussi est-elle disponible sur SUNOS 4.1.x, SOLARIS 2.x, SCO UNIX, FREEBSD, LINUX et DOS.

Au départ, cette bibliothèque était destinée au support exécutif de la plate-forme ADA (voir section 3.1.2.1) de GNU. L'un de ses objectifs principaux a été d'implémenter correctement le standard POSIX (plus exactement le *Draft* 6 de la norme). Elle a ainsi été l'une des premières bibliothèques s'affichant comme compatible avec ce standard émergeant. De plus, elle était préemptive grâce à l'utilisation de signaux (`SIGALARM`).

Néanmoins, cette compatibilité avec la norme POSIX a réduit ses performances. En particulier, l'obligation de gérer correctement les points d'abandon (*cancellation points*) ainsi que les masques de signaux de chacun des processus légers a alourdi le code. Beaucoup d'appels systèmes pour les entrées/sorties ont également dû être détournés pour éviter des appels systèmes bloquants qui auraient paralysé l'application entière (et non pas uniquement le processus léger incriminé).

Cette bibliothèque a inspiré de nombreux travaux. Elle n'est plus développée, mais elle est souvent citée en référence car elle a été la première bibliothèque de processus légers disponible sur de nombreux systèmes.

3.4.1.2 GnuPth

GNU PORTABLE THREADS (GNUPTH) [Eng99, Eng00] est une bibliothèque utilisateur de processus légers développée par le projet OSSP (*Open Source Software Project*) et qui est devenue par la suite une partie du projet GNU.

Portabilité. Son objectif principal est, comme son nom l'indique, la *portabilité*. Cet objectif est réalisé en n'utilisant pour le développement de cette bibliothèque que des fonctions POSIX. Aucun code assembleur n'est requis. La portabilité est donc excellente : il suffit que le système cible soit conforme à la norme POSIX. Par contre, cela peut conduire à quelques constructions complexes (passage par un traitant de signal pour créer une nouvelle pile, par exemple) qui ont pour but de ne pas manipuler directement les registres de la machine et qui dégradent les performances de cette implémentation. L'interface applicative proposée est conforme à la norme POSIX, mais contient également quelques extensions permettant de manipuler les fonctionnalités propres de cette bibliothèque (envoi de messages inter processus légers, etc.)

Fonctionnalités. GNUPTH ne propose pas de fonctionnalités limitant sa portabilité. La bibliothèque fournit un système robuste, suffisamment complet pour gérer des processus légers, mais *sans préemption*. La norme POSIX ne requiert pas cette fonctionnalité, bien qu'elle soit généralement très appréciée par les développeurs d'applications multithreadées. Cela a plusieurs conséquences :
- GNUPTH s'interface assez facilement avec les bibliothèques extérieures. En effet, comme il n'y a pas de préemption, les appels aux fonctions des bibliothèques externes ne sont pas interrompus. Pour être compatible avec GNUPTH, il n'est pas nécessaire qu'une bibliothèque soit réentrante ;
- l'application ne peut pas utiliser GNUPTH pour obtenir automatiquement un entrelacement de l'exécution de plusieurs tâches.

Ce dernier point signifie que, avec cette bibliothèque, les processus légers ne permettent pas d'améliorer la réactivité d'une application aux événements extérieurs. Si un processus léger est créé pour attendre un événement asynchrone, il ne sera pas réveillé pendant l'exécution d'une tâche de calcul même si cette dernière est très longue[10].

Du fait de l'implémentation au niveau utilisateur, les appels systèmes bloquants demeurent bien évidemment problématiques. Ils sont donc détournés et réimplémentés à l'aide de mécanismes de scrutation proposés par GNUPTH. La bibliothèque n'étant pas préemptive, cette scrutation ne peut être effectuée qu'au moment où l'ordonnanceur reprend la main, c'est-à-dire lorsqu'un processus léger bloque ou lorsqu'il est appelé explicitement avec, par exemple, la fonction `pth_yield()`.

Par ailleurs, GNUPTH essaie de respecter scrupuleusement la norme POSIX, y compris la partie relative aux signaux. Cela signifie que chaque processus léger peut avoir son propre masque de signaux. Mais cela impose à la bibliothèque de modifier le masque de signaux de l'application à chaque changement de contexte pour l'adapter à celui du processus léger ordonnancé. Ces appels systèmes à chaque changement de contexte peuvent devenir coûteux pour une bibliothèque de niveau utilisateur.

Ordonnancement. L'application peut diriger l'ordonnancement par l'utilisation de priorités. Une autre possibilité est d'utiliser la fonction `pth_yield(`*`to`*`)` qui, contrairement à la fonction similaire de la norme POSIX, permet de spécifier le processus léger auquel on veut passer la main. Cette fonctionnalité est facilement implémentable dans une bibliothèque de

[10]Ou alors, il faudrait que, régulièrement, la tâche de calcul redonne explicitement la main avec la fonction `pth_yield()` pour que la scrutation puisse avoir lieu. Or, c'est justement pour éviter cette instrumentation du code[11] que l'on cherche généralement à utiliser des processus légers.

niveau utilisateur : le parallélisme n'étant pas réel, on est sûr que les autres processus légers ne sont pas en cours d'exécution et, de plus, comme la bibliothèque n'est pas préemptive, le programmeur peut s'assurer assez facilement que le processus léger auquel il veut passer la main ne s'est pas terminé entre temps. Il est ainsi très facile de construire une bibliothèque de coroutines au-dessus de GNUPTH.

Une particularité de cette bibliothèque est d'utiliser un processus léger spécifique pour réaliser l'ordonnancement. Cela signifie qu'un changement de contexte s'effectue en deux temps : d'abord le processus léger en cours d'exécution donne la main au processus léger de l'ordonnanceur, puis ce dernier donne la main au processus léger suivant. Ceci a pour avantage d'améliorer la robustesse de la bibliothèque : l'ordonnanceur n'utilise pas les ressources des autres processus légers (pile[12], variable `errno`, etc.). En contrepartie, cela rend les changements de contexte plus lents : il faut changer deux fois les registres, et surtout il faut modifier deux fois le masque de signaux de l'application (avec des appels systèmes).

Synthèse. GNUPTH est une bibliothèque de niveau utilisateur non préemptive privilégiant la portabilité par rapport aux fonctionnalités et à l'efficacité. N'étant pas préemptive, elle sera particulièrement adaptée aux applications de type serveur où les processus légers sont fréquemment suspendus, en attente d'événements. GNUPTH a servi de socle à la bibliothèque mixte NGPT présentée par la suite.

3.4.1.3 Capriccio

CAPRICCIO [vBCZ+03] est une bibliothèque de processus légers écrite pour être utilisée par des serveurs devant supporter des charges très importantes. Ses auteurs veulent démontrer qu'il est possible de programmer de telles applications avec le modèle standard de processus légers sans nécessiter le recours à une programmation événementielle. Sa cible est donc similaire à celle de GNUPTH, avec cependant des objectifs légèrement différents. Dans CAPRICCIO la portabilité est sacrifiée au profit de l'efficacité.

Portabilité. CAPRICCIO est développée exclusivement pour le système d'exploitation LINUX et exploite de nombreux mécanismes introduits dans les noyaux récents, comme l'interface `epoll`, pour s'assurer de performances optimales. L'interface proposée est semblable à celle de la norme POSIX, mais elle ne l'implémente pas complètement, en particulier en ce qui concerne les signaux ou les sémaphores.

Fonctionnalités. Partageant des objectifs communs avec GNUPTH (gérer des applications devant traiter un grand nombre de requêtes), il est normal qu'un certain nombre de choix de conception soient similaires. En particulier, CAPRICCIO est également une bibliothèque *non préemptive*. Comme pour GNUPTH, cela lui permet d'être facilement réentrante vis-à-vis des bibliothèques extérieures. Cela lui permet également de proposer des primitives de synchronisation efficaces : ni les processus légers ni l'ordonnanceur ne peuvent être interrompus pendant une section critique, aucune protection particulière n'est donc à prévoir.

[12]L'utilisation d'une pile différente pour l'ordonnanceur permet de supprimer facilement un processus léger terminé : sa pile n'est plus en utilisation lorsque l'ordonnanceur est appelé, il peut donc désallouer la zone mémoire correspondante.

Pour offrir une efficacité maximale, CAPRICCIO utilise plusieurs techniques particulières. Tout d'abord, CAPRICCIO propose une gestion originale des piles des processus légers. Les piles ne sont pas allouées d'un bloc au début de l'exécution d'un processus léger. Elles sont allouées progressivement, suivant les besoins du processus léger. Pour réaliser cela, le code applicatif est analysé à la compilation afin d'introduire, au besoin, des points de contrôle pour la bibliothèque qui lui permettent alors d'allouer dynamiquement à l'exécution la place nécessaire pour les piles des processus légers. CAPRICCIO est ainsi capable de gérer un très grand nombre de processus légers (jusqu'à 100 000).

Comme GNUPTH, CAPRICCIO ne peut pas utiliser d'appels systèmes bloquants. Ces derniers sont donc détournés et remplacés par des mécanismes implémentés à partir des nouvelles primitives d'entrées/sorties asynchrones du noyau LINUX et de l'interface `epoll`. Associés à une implémentation de toutes les opérations relatives aux processus légers en temps constant, ces mécanismes garantissent à CAPRICCIO une efficacité remarquable.

Ordonnancement. CAPRICCIO propose un ordonnancement qui tient compte de la consommation des ressources par les processus légers. Cet ordonnancement est spécifiquement adapté pour les applications ciblées (serveurs de requêtes) en offrant une réactivité maximum vis-à-vis des requêtes réseaux. Il offre ainsi des performances intéressantes avec le serveur web APACHE (15 % d'amélioration avec CAPRICCIO, cf [vBCZ+03]). Cependant, rien n'est prévu pour l'adapter à d'autres classes d'applications.

Synthèse. Ciblant une classe d'applications particulière, cette bibliothèque parvient à offrir ses services avec une très grande efficacité tout en permettant de garder le modèle de programmation multithreadée. Néanmoins, les excellentes performances affichées ici ont été obtenues au prix d'une spécialisation de la bibliothèque (utilisation de mécanismes propres au système LINUX, algorithmes d'ordonnancement adaptés à la classe d'applications ciblée, fonctionnalités restreintes). Il est également nécessaire de compiler les applications avec les outils fournis (analyse du code pour la gestion des piles). Cela signifie que certains codes ne pourront pas fonctionner avec CAPRICCIO (langages non supportés, fonctionnalités non implémentées par CAPRICCIO, etc.)

CAPRICCIO est en cours de réécriture pour supporter les machines multiprocesseurs et la préemption. Il sera intéressant de quantifier l'impact sur ces performances de l'ajout de telles fonctionnalités.

3.4.1.4 Bilan

La plupart des bibliothèques de niveau utilisateur ont pour principaux objectifs la portabilité, les fonctionnalités et/ou l'efficacité. Aucune n'est bien sûr capable d'exploiter des machines multiprocesseurs, et les appels systèmes bloquants restent sources de problèmes. Dans ce dernier cas, certaines bibliothèques considèrent que c'est à l'application de ne pas faire d'appel système bloquant, d'autres fournissent des fonctionnalités équivalentes mais non bloquantes pour l'application. Pour être transparentes aux applications, ces bibliothèques détournent les fonctions bloquantes et les remplacent par la version de leur cru : une intégration avec les bibliothèques systèmes, en particulier la `libc`, est alors requise.

3.4.2 Bibliothèques de niveau noyau

À l'opposé des bibliothèques de niveau utilisateur, les bibliothèques de niveau noyau sont généralement dédiées à un système particulier. Ces bibliothèques utilisent l'ordonnanceur du système pour leurs processus légers ; une étroite intégration entre le système et la bibliothèque s'avère donc nécessaire. Les bibliothèques fournies avec les systèmes d'exploitation récents sont généralement des bibliothèques de niveau noyau (plus rarement de niveau mixte), ceci afin de permettre aux processus légers d'exploiter véritablement les machines multiprocesseurs.

Nous présentons ici l'exemple du système LINUX où deux bibliothèques de processus légers noyau ont été développées.

3.4.2.1 LinuxThread

LINUXTHREAD [Ler96] a été développée par Xavier LEROY lorsque le noyau LINUX s'est enrichi suffisamment pour permettre la création de processus légers de niveau noyau. Elle est devenue la bibliothèque de processus légers standard de LINUX.

Portabilité. LINUXTHREAD est une bibliothèque spécifique au système d'exploitation LINUX. L'ajout de la primitive `clone()` dans le noyau expérimental 1.3.56 (janvier 1996) a permis de créer des processus partageant leur mémoire, table de fichiers ouverts, traitants de signaux, etc. À titre de comparaison, des bibliothèques de niveau utilisateur étaient disponibles dès le noyau stable 1.0.9 (avril 1994). L'interface applicative se veut conforme à celle de la norme POSIX malgré quelques imperfections, notamment vis-à-vis de la gestion des signaux (voir ci-après).

Fonctionnalités. De par l'objectif fixé (être la bibliothèque de processus légers du système LINUX), LINUXTHREAD doit être le plus possible conforme à la norme POSIX. Les fonctionnalités offertes sont donc celles de la norme.

Elle souffre cependant de quelques imperfections : le support noyau est insuffisant pour implémenter complètement la norme POSIX. En particulier, le modèle des signaux n'est pas conforme à celui de POSIX. Linus TORVALDS, le développeur du noyau, trouvait (à tort ou à raison) son modèle plus sain que celui de POSIX et ne voulait pas ajouter le support adéquat au noyau. Cela a longtemps été la source de son incompatibilité avec des programmes écrits conformément à la norme POSIX. La bibliothèque NPTL, présentée ci-après, adresse ces difficultés.

Un autre problème de cette bibliothèque provient du fait qu'elle repose sur l'envoi et la réception de signaux pour sa synchronisation interne. La tendance à perdre parfois des signaux[13] a fait que cette bibliothèque n'a jamais été très stable en utilisation intensive.

Enfin, comme le système ne reconnaît pas les processus multithreadés en tant que tel (cf. Figure 3.4), chaque processus léger possède un identifiant de processus (`pid`) différent, ce qui est contraire à la norme POSIX. Ceci explique que la bibliothèque ait besoin d'un processus léger leader, père de tous les autres : la fin d'un processus léger ne doit pas conduire d'autres processus légers à être rattachés au processus initial (`init`), ce qui serait le cas si le processus léger terminé était le père d'autres processus légers. Le leader permet donc de

[13] Il s'agissait de bugs soit dans la bibliothèque, soit dans le noyau qui n'ont jamais été totalement éliminés.

garder une cohérence dans les relations de filiation entre un processus multithreadé et ses fils, mais au prix d'une complexification coûteuse du mécanisme de création des processus légers.

Ordonnancement. Étant de niveau noyau, cette bibliothèque utilise directement l'ordonnanceur du noyau. Les possibilités de diriger l'ordonnancement sont donc dépendantes des fonctionnalités offertes par le système. Les processus légers ont donc un ordonnancement préemptif basé sur des priorités statiques et dynamiques.

On notera qu'il est possible de demander un ordonnancement basé sur des priorités fixes (ordonnancement temps-réel), mais que cela nécessite des droits étendus (`root`) pour l'application. Toute spécification de priorité se fait par rapport à l'ensemble des processus du système, et non pas uniquement par rapport aux autres processus légers de l'application.

Synthèse. LINUXTHREAD a longtemps été la bibliothèque de processus légers de référence sur les systèmes LINUX. Étant de niveau noyau, elle s'intègre parfaitement avec le système (ordonnancement, appels systèmes bloquants, exploitation des multiprocesseurs, etc.), mais s'adapte difficilement à des besoins particuliers comme un ordonnancement non préemptif pour exploiter au maximum les caches des processeurs.

Les lacunes des anciens noyaux LINUX pour permettre un multithreading efficace compatible POSIX (voir la Figure 3.4) expliquent le développement de la nouvelle bibliothèque, présentée ci-après, qui utilise les nouveaux supports introduits dans le noyau LINUX.

3.4.2.2 NPTL

La bibliothèque NPTL [DM03] (*Native* POSIX *Thread Library*) a été développée par Ulrich Drepper. Cette bibliothèque de processus légers de niveau noyau pour LINUX a été écrite avec, pour principaux objectifs, une conformité à la norme POSIX parfaite et un passage à l'échelle aussi performant que possible.

Portabilité. Comme LINUXTHREAD, cette bibliothèque est dédiée au système d'exploitation LINUX. Elle offre une interface totalement conforme à la norme POSIX mais, pour cela, nécessite un noyau LINUX récent offrant les fonctionnalités requises au niveau du noyau (voir Figure 3.4).

Fonctionnalités. Les fonctionnalités offertes par NPTL sont, comme pour LINUXTHREAD, celles décrites dans la norme POSIX. Le support de la norme est cependant ici beaucoup plus complet et correct, grâce à l'exploitation des nouvelles fonctionnalités introduites dans les noyaux LINUX récents.

Outre, un meilleur respect de la norme POSIX, ces services supplémentaires offerts par le noyau permettent à cette bibliothèque d'offrir de bien meilleures performances que LINUXTHREAD. Tout d'abord, des mécanismes sont simplifiés, comme la création des processus légers qui ne doit plus faire intervenir de processus léger leader. Et surtout, au niveau de la synchronisation, l'introduction du mécanisme de FUTEX [Rus02b] permet de synchroniser plusieurs flots d'exécution parallèles efficacement : en l'absence de contention, la synchronisation est faite entièrement en espace utilisateur. Le noyau n'est sollicité avec un appel système que s'il est nécessaire d'arbitrer l'accès à la ressource.

Processus multithreadé. Le noyau LINUX ne reconnaissait pas la notion de processus multithreadé, seulement celle de processus partageant des ressources (mémoire, traitants de signaux, etc.). Les différents processus légers obtenaient donc chacun un identifiant de processus (`pid`) différent !

Désormais, une notion de groupe de tâches a été introduite. Ces flots d'exécution partagent le même `pid`. Un nouvel appel système (`gettid`) a été ajouté pour obtenir l'identifiant de la tâche courante dans le processus.

Signaux. Même si plusieurs tâches peuvent depuis longtemps partager les traitants des signaux dans le noyau LINUX, les signaux générés étaient destinés à une tâche précise, et non pas à un processus. Cela signifie qu'un signal bloqué par un processus léger pouvait être ignoré par l'application alors qu'un autre processus léger aurait dû le recevoir.

Le mécanisme de gestion des signaux a été complètement réécrit permettant d'adresser des signaux à un processus multithreadé plutôt qu'à une tâche particulière. La diffusion du signal à l'un des processus légers les acceptant est prise en charge correctement par le noyau.

Accréditations. Les accréditations sont les identités collectives associées à un processus ou une tâche, comme l'`uid` sous lequel tourne la tâche ou les groupes UNIX auxquels elle appartient. Le noyau LINUX ne permettaient pas de partager ces accréditations au sein d'un processus multithreadé. Des améliorations ont été apportées sur ce point dans les noyaux récents, mais le support n'est pas encore complet.

Annulation des sémaphores. La norme POSIX demande que les sémaphores puissent être réinitialisés lors de la terminaison d'un processus. Or LINUX le faisait à la fin des tâches, ce qui était problématique pour les processus multithreadés. Désormais, il est possible de demander au système de gérer l'annulation des sémaphores à la fin des processus, conformément à la norme.

Actions relatives aux processus entiers. Certaines actions sont, d'après la norme POSIX, relatives aux processus alors que LINUX les implémentait relativement aux tâches. Ceci était problématique dans le sens où ces actions pouvaient être initiées par le noyau lui-même, donc impossible à intercepter et émuler correctement au niveau d'une bibliothèque. On peut ainsi considérer :

la terminaison. La terminaison d'un processus, éventuellement à cause d'un signal, peut survenir dans n'importe quelle tâche mais doit générer un code de retour pour le processus et non la tâche ;

l'exécution. L'appel système `execve()` doit remplacer le processus courant et non la tâche courante. Cela signifie que les autres processus légers éventuels du processus doivent être supprimés du système ;

la suspension, le redémarrage. Certains signaux (`SIGSTOP`, `SIGCONT`) doivent agir sur le processus entier et non sur une tâche particulière.

L'article [McC02] analyse les lacunes des anciens noyaux LINUX *pour permettre une gestion des processus légers conforme à la norme* POSIX*. Les plus problématiques ont fait l'objet d'adaptation dans les noyaux récents (fin de la série 2.4 et nouvelle série 2.6) depuis maintenant un peu plus d'un an. Les modifications du noyau ont parfois nécessitée de très longues argumentations avec Linus* TORVALDS *pour le convaincre de leur bien-fondé.*

FIG. 3.4 – *Le noyau* LINUX *et le support pour le multithreading noyau*

Ordonnancement. Utilisant également l'ordonnanceur du système, les mêmes remarques que pour la bibliothèque LINUXTHREAD s'appliquent.

Synthèse. De manière générale, NPTL a des propriétés très semblables à celles de LINUX-THREAD mais corrige la plupart de ses défauts internes. Cette amélioration a été rendue possible par l'introduction dans le noyau LINUX de nouvelles fonctionnalités. Cette bibliothèque est maintenant stabilisée ; elle est fournie avec la majorité des distributions actuelles. Elle nécessite cependant pour fonctionner un noyau LINUX 2.6.x.

3.4.2.3 Bilan sur les bibliothèques de niveau noyau

Les bibliothèques de niveau noyau nécessitent un support spécifique du système d'exploitation comme l'illustre bien l'histoire de l'intégration des processus légers au système LINUX. Rappelons que la première bibliothèque de processus légers noyau est apparue avec l'introduction de l'appel système `clone()`. Malgré de nombreuses améliorations, cette bibliothèque n'a pu être mise à la norme POSIX et sa robustesse reste limitée en cas d'utilisation intensive. L'introduction de supports supplémentaires dans le noyau a résolu ces problèmes et conduit au développement de la bibliothèque NPTL.

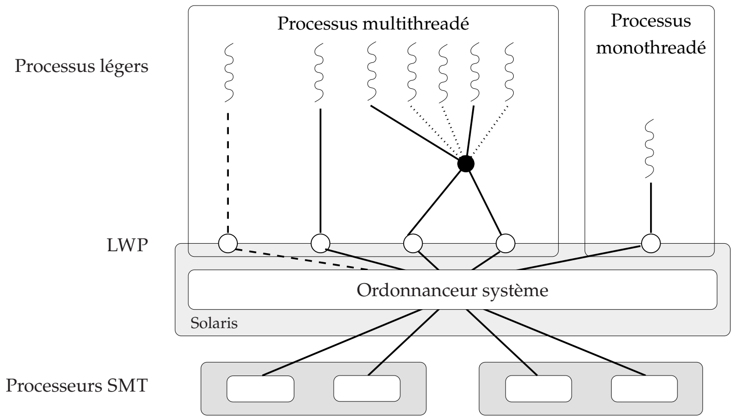
Cette nécessité d'interaction fine entre la bibliothèque de processus légers et le système d'exploitation explique qu'il existe peu de bibliothèques de niveau noyau par rapport aux bibliothèques de niveau utilisateur : la difficulté de développement et d'intégration n'incite pas à développer une bibliothèque noyau adaptée à une application donnée (ordonnancement particulier, fonctionnalités exotiques, etc.)

Les systèmes d'exploitation modernes intègrent des bibliothèques de processus légers de niveau noyau, permettant ainsi aux applications multithreadées d'exploiter les architectures matérielles parallèles. Toutefois, les fonctionnalités offertes sont très souvent restreintes à celles définies par la norme POSIX, et donc insuffisantes lorsque l'application a des besoins particuliers (ordonnancement spécifique, intégration des processus légers et des communications pour être réactif, etc.)

Spécialiser ces bibliothèques pour des besoins spécifiques est très difficile car cela nécessite généralement d'intervenir sur le système, c'est-à-dire sur l'ensemble des processus de la machine et pas uniquement sur l'application considérée. Linus TORVALDS a souvent été réticent à modifier le comportement du noyau LINUX uniquement au bénéfice des applications multithreadées. Un second reproche, dû au fait que ces bibliothèques ne proposent que l'interface de la norme POSIX, est que rien n'est proposé pour intégrer les processus légers à d'autres composants des applications, en particulier les communications. Aucun support concernant la réactivité n'est disponible par exemple.

3.4.3 Bibliothèques mixtes

Comme les bibliothèques de niveau noyau, les bibliothèques mixtes sont également capables d'exploiter les machines multiprocesseurs. De plus, elles ont l'avantage de garder un ordonnanceur en espace utilisateur et donc d'être efficaces et flexibles. Toutefois, la nécessité de faire coopérer deux ordonnanceurs rend l'écriture de telles bibliothèques plus difficile.



Au sein d'une même application, les processus légers peuvent soit être associés directement à un LWP, *soit être gérés par un pool de* LWP. *On a donc à la fois le modèle « noyau » et le modèle « mixte »*
Lorsque suffisamment d'appels systèmes bloquants ont suspendu tous les LWP *disponibles,* SOLARIS *est capable d'envoyer un signal spécial à l'application en lui créant un nouveau* LWP. *Les autres processus légers vont pouvoir continuer à s'ordonnancer.*

FIG. 3.5 – Le modèle de processus légers de SOLARIS

3.4.3.1 Solaris

Le système d'exploitation SOLARIS est celui qui a popularisé les bibliothèques de processus légers de niveau mixte. À l'occasion de son développement, le terme LWP a été introduit.

Portabilité. SUN a beaucoup contribué à l'élaboration de la norme POSIX. Cela explique que sa bibliothèque de processus légers a, dès l'origine, une interface très proche de celle de la norme. Elle est cependant plus riche en permettant de manipuler les LWP. C'est pourquoi SUN a gardé cette interface native tout en proposant une interface POSIX conforme pour pouvoir facilement compiler les applications POSIX.

Fonctionnalités. Le modèle de multithreading de cette bibliothèque permet de définir un ensemble de processus légers de niveau noyau (LWP) liés ou non à des processeurs physiques. Dans ces LWP sont ordonnancés en espace utilisateur des processus légers de niveau utilisateur qui peuvent être liés à un ou plusieurs LWP. La Figure 3.5 illustre les différentes possibilités d'organisation des processus légers.

Pour gérer les appels systèmes bloquants sans bloquer toute l'application, les programmeurs de SOLARIS ont introduit la fonctionnalité suivante : un signal, à destination de la bibliothèque de processus légers, est envoyé par le noyau lorsque celui-ci détecte que tous les LWP d'un processus donné sont bloqués. De cette manière, la bibliothèque peut créer un nouveau LWP pour que d'autres processus légers de niveau utilisateur puissent continuer à s'exécuter.

Pour profiter des mécanismes précédemment décrits, l'application doit utiliser l'interface propre de SOLARIS ; la norme POSIX ne définit pas de propriétés similaires à celles offertes par le modèle de SOLARIS.

Ordonnancement. Comme les bibliothèques précédentes, seule la notion de priorité permet de diriger l'ordonnancement des processus légers. Aucun support n'est prévu dans SOLARIS pour permettre d'intégrer efficacement des communications avec des processus légers. Si l'on désire un ordonnancement particulier, il faut alors réécrire une bibliothèque de processus légers, en utilisant éventuellement les mécanismes proposés par le système.

synthèse. La bibliothèque de processus légers de SOLARIS est souvent citée en exemple pour la richesse des fonctionnalités offertes et sa souplesse d'utilisation. Malheureusement, elle est intimement liée au système d'exploitation et ne tourne donc que sur l'architecture `sparc` avec le système d'exploitation SOLARIS. Il existe aussi quelques versions de SOLARIS pour architecture `x86`, mais, jusqu'à présent, ses performances sont loin d'égaler celles de l'architecture `sparc`. Enfin, même si le modèle est souple, la spécialisation de la bibliothèque pour l'adapter à des applications particulières n'est pas prévue.

3.4.3.2 NGPT

Sous LINUX, il n'existe qu'une seule bibliothèque mixte qui s'est vraiment répandue : la bibliothèque NGPT [ADH+02] (*New Generation* POSIX *Threading*). Elle a été développée chez IBM avant que NPTL ne voie le jour. Sous l'impulsion de ses développeurs, certaines fonctionnalités absentes du noyau mais nécessaires à la compatibilité POSIX ont été ajoutées au noyau LINUX. Ces fonctionnalités, décrites dans la Figure 3.4, ont ensuite été utilisées par NPTL.

Portabilité. NGPT est basée sur la bibliothèque GNUPTH. Toutefois, pour offrir un modèle mixte, elle utilise des fonctionnalités propres à LINUX. Par conséquent, contrairement à GNUPTH, NGPT est spécifique à ce système d'exploitation. Concernant l'interface utilisateur, NGPT vise à offrir l'interface POSIX. Quelques fonctionnalités supplémentaires provenant de GNUPTH sont également offertes (passages de messages inter processus légers, acquisition de verrous avec temps limite, etc.)

Fonctionnalités. La plupart des fonctionnalités de cette bibliothèque découlent directement de son origine. Dérivée de GNUPTH, elle s'en distingue par l'utilisation de plusieurs LWP lui permettant ainsi d'exploiter les machines multiprocesseurs. Comme GNUPTH, elle est *non préemptive*. Sa synchronisation interne, qui doit désormais prendre en compte les flots parallèles, est assurée à l'aide du mécanisme des FUTEX, comme pour NPTL.

Destinée spécifiquement au système LINUX, son interface est compatible au niveau binaire avec celle de la bibliothèque LINUXTHREAD. Cela assure que les bibliothèques extérieures réentrantes vis-à-vis de LINUXTHREAD le sont aussi vis-à-vis de NGPT. De plus, NGPT redéfinit tous les appels systèmes bloquants en les remplaçant par des variantes non bloquantes pour éviter que quelques processus légers bloquent la bibliothèque entière. Cette substitution est faite directement au niveau de l'interface avec la bibliothèque C du système ; elle est donc totalement transparente aux applications qui en bénéficient automatiquement.

La volonté d'afficher une conformité POSIX parfaite a nécessité de garder l'utilisation systématique de deux appels systèmes aux changements de contexte (pour bloquer les signaux pendant l'exécution du processus léger « ordonnanceur » et ensuite pour installer le masque du processus légers suivant) présents dans GNUPTH. Ceci a considérablement réduit le gain qu'on aurait pu espérer pour les changements de contexte entre processus légers de niveau utilisateur.

Ordonnancement. NGPT utilise le même ordonnanceur que GNUPTH avec une file de processus légers prêts ordonnée par priorité. Cependant, en raison des multiples LWP, les accès à cette file sont sérialisés avec un verrou. La file unique permet d'éviter un équilibrage de charge entre les LWP, mais introduit un point de contention sur les systèmes avec de nombreux processeurs. Là encore, aucun mécanisme autre que des priorités, permettant à l'application de diriger son ordonnancement, n'est offert.

Synthèse. La création de la bibliothèque NGPT a permis de montrer que la bibliothèque LINUXTHREAD pouvait être grandement améliorée si le noyau LINUX fournissait de nouveaux supports. Son développement a cependant cessé avec l'apparition de la bibliothèque NPTL. Les coûts de certaines de ses opérations (changement de contexte) ne lui ont pas permis d'afficher des performances réellement meilleures que NPTL, alors que son architecture était plus complexe et condamnait certaines fonctionnalités comme l'utilisation d'appels systèmes bloquants ou la présence de préemption dont l'absence a été beaucoup critiquée.

3.4.3.3 Bilan sur les bibliothèques mixtes

Pour être efficaces et fonctionnelles, les bibliothèques mixtes présentées ici s'appuient sur des fonctionnalités offertes par le système. À l'instar des bibliothèques de niveau utilisateur, les appels systèmes bloquants sont problématiques. Ils peuvent alors être remplacés par de la scrutation comme dans NGPT, ou bien des fonctionnalités particulières du noyau peuvent être exploitées : SOLARIS crée un nouvel LWP en cas de besoin pour éviter de bloquer toute l'application. On remarquera que cette solution peut conduire à la création de nombreux LWP si les processus légers font fréquemment des appels systèmes bloquants, se rapprochant alors du modèle de processus légers de niveau noyau.

À l'exception de la bibliothèque de processus légers du système SOLARIS, les bibliothèques mixtes restent peu développées. Le développement de NGPT a été abandonné peu de temps après l'apparition de NPTL. Il est en effet difficile de faire coopérer efficacement plusieurs ordonnanceurs, et le support complet de la norme POSIX peut détruire les bénéfices que l'on pourrait attendre d'une bibliothèque mixte par rapport à une bibliothèque de niveau noyau.

3.5 Conclusion

Les processus légers sont devenus incontournables non seulement dans les systèmes, pour permettre l'exploitation des machines multiprocesseurs, mais également dans les langages, pour exprimer le parallélisme d'une application (ou d'un support exécutif).

Comme nous l'avons vu, il existe une grande variété de bibliothèques de processus légers. Cela s'explique par la nécessité de choix fondamentaux lors de leur conception, choix qui conditionnent certaines propriétés de la bibliothèque. La multitude de bibliothèques découle de la variété des compromis possibles concernant ces propriétés, chacun restant généralement bénéfique à une classe d'applications particulière.

Les supports exécutifs des environnements parallèles hautes performances nécessitent, eux aussi, une bibliothèque de processus légers spécifique. Ainsi que nous l'avons mis en évidence dans le chapitre précédent, cela signifie qu'une telle bibliothèque devra permettre d'abstraire la plate-forme matérielle tout en permettant à l'application de l'exploiter le plus efficacement possible. Par ailleurs, l'interface POSIX est clairement insuffisante : rien ne permet de diriger l'ordonnancement par exemple. Aucune extension de la norme n'est en discussion sur ce sujet actuellement ; seules quelques bibliothèques comme GNUPTH proposent des solutions particulières (`yield(to)`). Enfin, aucune des bibliothèques rencontrées n'offre des fonctionnalités qui autoriseraient une intégration efficace avec des bibliothèques de communication. En particulier, la perte de réactivité des environnements due parfois au grand nombre de processus légers qui peuvent s'exécuter dans l'application n'est jamais considérée. Le chapitre suivant présentera les techniques que nous avons conçues et développées pour répondre à ces besoins.

Chapitre 4

Une bibliothèque de processus légers universelle

Sommaire

Ce chapitre présente le cœur de mes travaux : la définition et la réalisation d'une bibliothèque de processus légers qui soit à la fois performante, portable et réactive. Nous allons d'abord montrer les obstacles majeurs à une telle réalisation qui nous ont conduits à chercher des solutions innovantes. Grâce à la mise en évidence et à l'analyse des besoins et contraintes issus du calcul hautes performances, je présente ici des solutions à intégrer dans les bibliothèques de processus légers, notamment dans les domaines de la réactivité des applications et de l'ordonnancement. Les problèmes plus spécifiques ou techniques, liés à la mise en œuvre de ces solutions, seront présentés dans le chapitre 5 relatif à l'implémentation de mes travaux.

4.1 Points clés de la démarche

La qualité des performances, l'étendue des fonctionnalités et la portabilité sont les trois critères naturels de comparaison des bibliothèques de processus légers. Dans une certaine mesure, l'obtention de bonnes performances a tendance à se faire au détriment des deux autres critères. Par exemple, pour être idéalement portable, une bibliothèque doit respecter scrupuleusement une norme alors que l'abandon de certaines fonctionnalités permettrait d'alléger la bibliothèque afin de gagner en performance. Le but de mes travaux est de concevoir et de développer une bibliothèque de processus légers pour le calcul hautes performances fonctionnant de manière optimale sur une vaste gamme de systèmes. Aussi, l'efficacité de notre bibliothèque a-t-elle été le principal critère retenu, ce qui a conduit à proposer de nouvelles fonctionnalités originales pour améliorer les performances mais qui ne sont pas standards.

Les configurations à exploiter sont variées, que ce soit au niveau de la famille de processeurs, au niveau de l'architecture de la machine et, bien entendu, au niveau du système d'exploitation (avec ou sans support pour activités concurrentes). Mon objectif est bien d'assurer la *portabilité des performances*, c'est-à-dire la faculté pour une application donnée de rester automatiquement efficace quel que soit l'environnement sur lequel elle s'exécute. Cette contrainte oblige à une réflexion sur les besoins réels d'une application et la façon dont elle peut les formuler indépendamment de la configuration envisagée : il ne faut pas lier l'application à une spécificité d'un système donné qui sera absente ou inefficace sur un autre système.

4.1.1 Des processus légers de niveau utilisateur

Ainsi que nous l'avons vu au chapitre 3, il existe plusieurs types de bibliothèques de processus légers : elles peuvent être de niveau noyau, utilisateur ou mixte. Notre objectif est de s'intégrer dans un support exécutif pour le calcul parallèle hautes performances. Il convient de choisir consciencieusement le modèle utilisé, ce choix ayant de lourdes conséquences sur les caractéristiques de bibliothèque développée (voir la section 3.3 pour plus de détails).

Nous avons constaté que les bibliothèques de processus légers de niveau noyau ne conviennent pas à la fois pour des raisons de performance et d'extensibilité. Les opérations de création, de destruction et de synchronisation de tels processus légers nécessitent le basculement du mode utilisateur en mode noyau. Ce basculement a un coût important : sur les opérations de gestion des processus légers, on observe un facteur 10 entre les bibliothèques de niveau noyau et celles de niveau utilisateur (voir partie 7.1). En outre, la plupart des bibliothèques de niveau noyau sont intimement liées au système d'exploitation dont elles font finalement partie. Elles sont donc peu portables et leurs adaptations sont limitées car elles correspondent à des modifications du système d'exploitation... Enfin, pour permettre l'exécution de toute application multithreadée, ces bibliothèques tendent à implémenter intégralement les standards, ce qui entraîne des surcoûts inutiles pour le domaine des hautes performances.

Les bibliothèques de niveau purement utilisateur offrent de bonnes performances et une excellente flexibilité. Le code résidant intégralement dans l'espace applicatif, aucune interférence n'est à craindre avec le système d'exploitation; elles peuvent donc être adaptées à une application spécifique. Cependant, de par leur principe même, elles sont incapables d'exploiter les machines multiprocesseurs.

Une bibliothèque mixte est de ce fait nécessaire. Très peu de bibliothèques mixtes ont été développées, sans doute parce que leur conception est délicate (il faut faire coopérer deux ordonnanceurs) et que cela demande généralement une bonne intégration avec le système. Nous verrons que les solutions proposées ici permettent de concevoir une telle bibliothèque portable sur un ensemble varié de systèmes et adaptée au calcul hautes performances où les machines parallèles sont en général dédiées à l'application en cours.

Même si aucune bibliothèque existante ne répond totalement à nos besoins, beaucoup peuvent être adaptées. La bibliothèque de processus légers MARCEL a été développée par Jean-François Méhaut et Raymond Namyst au milieu des années 1990. Elle a pour caractéristiques d'être très légère (moins de 5 500 lignes de C et 500 d'assembleur), très performante et portable sur de nombreuses architectures. De plus, un nombre respectable d'équipes de recherche l'ont utilisée et l'utilisent encore directement ou via des environnements de plus haut niveau. Mais, en tant que bibliothèque de niveau purement utilisateur, elle n'exploite pas les machines multiprocesseurs. Nous avons donc choisi de l'améliorer, non seulement pour ses qualités intrinsèques, mais aussi pour bénéficier de l'expérience capitalisée par Raymond Namyst et des retours d'expériences disponibles. Toutefois, la majeure partie des travaux décrits par la suite, en particulier tout ce qui a trait à la réactivité du système par rapport aux entrées/sorties, est adaptable aux autres bibliothèques de processus légers.

4.1.2 Une bibliothèque caméléon

Nous voulons que l'application puisse exploiter au mieux différentes architectures et systèmes d'exploitation sans être contrainte à tenir compte de leurs spécificités. Le support exécutif doit donc optimiser l'exécution de l'application sur la machine et ce, pour un surcoût minimal. Un ordonnancement à deux niveaux est indispensable sur une machine multiprocesseur SMP pour pouvoir utiliser efficacement des processus légers de niveau utilisateur. Par contre, il est inutile de payer le surcoût de cet ordonnancement sur une machine monoprocesseur. L'ordonnanceur doit pouvoir être spécialisé en fonction de l'architecture et du système sous-jacent.

Notre second objectif est l'adaptabilité aux besoins spécifiques des applications de calcul hautes performances. Ces besoins sont relatifs à la politique d'ordonnancement (placement, durée du quantum) ou à des fonctionnalités (migration de processus légers, gestion des signaux, réactivité aux événements, synchronisation de groupe de processus légers, etc.). Là encore, il s'agit de proposer un support exécutif suffisamment souple pour offrir toutes les fonctionnalités requises sans payer le coût de celles qui ne le seraient pas. Dans cet objectif, le support exécutif doit être capable de se spécialiser (à la compilation, au déploiement ou à l'exécution) en fonction des ressources disponibles et des besoins de l'application.

Cette capacité de spécialisation a d'importantes répercussions sur l'architecture interne du support exécutif. Il n'est pas envisageable de développer un support exécutif particulier parfaitement optimisé pour chaque combinaison des ressources éventuellement disponibles et pour chacun des besoins applicatifs possibles. Le code du support exécutif doit donc être complètement modulaire. De cette façon, une application de calcul hautes performances désirant un ordonnancement particulier doit pouvoir changer la fonction de choix de l'ordonnanceur sans avoir à réécrire l'ensemble des mécanismes de gestion des processus légers.

Enfin, l'interface du support exécutif doit être suffisamment évolutive pour pouvoir s'adapter facilement aux besoins applicatifs et aux possibilités du système. Si l'on désire être compatible avec l'ensemble des applications existantes, alors il faut non seulement fournir l'API (Application Programming Interface / Interface de Programmation Applicative) de la bibliothèque de processus légers standard du système cible[1], mais aussi son ABI[2] (Application Binary Interface / Interface Binaire Applicative). La compatibilité au niveau de l'ABI permet de faire fonctionner un programme avec la nouvelle bibliothèque sans même le recompiler (grâce à l'édition de lien dynamique), la compatibilité au niveau de l'API permet de faire fonctionner un programme à condition de le recompiler, ce qui permet alors parfois de faire des optimisations supplémentaires (suppression d'appels de fonctions grâce à des fonctions en-ligne, etc.). Bien sûr, si le programmeur désire utiliser des fonctionnalités supplémentaires offertes par notre support exécutif (réactivité, migration, etc.), alors il doit utiliser une interface particulière.

4.1.3 Intégration de la problématique de la réactivité

Comme nous l'avons vu en section 2.3.2, la réactivité d'un support exécutif repose sur le dialogue entre l'ordonnanceur des processus légers et les gestionnaires de communication. Dans cette perspective, les développements des bibliothèques de processus légers et celles, par exemple, de communication ne peuvent être complètement dissociés. De même, avec le développement des accès aux mémoires de masses par des interfaces semblables aux réseaux (disques sur IP, iSCSI, etc.), les mêmes problèmes de réactivité vont survenir pour les entrées/sorties disques. La diversité des événements à considérer et celle des bibliothèques

[1] L'API standard de la plupart des systèmes actuels est celui de la norme POSIX. Elle est parfois légèrement étendue (sous LINUX, quelques fonctions ont été rajoutées) ou même parfois complètement revisitée comme c'est le cas sous SOLARIS : à côté des fonctions POSIX préfixées par `pthread_` est présent tout un ensemble de fonctions préfixées par `thr_` permettant de contrôler les processus légers natifs (à deux niveaux) de SOLARIS

[2] L'ABI d'une bibliothèque définit pour un système donné les conventions d'appels de fonctions (quel paramètre est placé dans quel registre du processeur), la liste des fonctions disponibles ainsi que les structures de données de cette bibliothèque (la taille et les champs des structures sont utilisés par le compilateur lors de la compilation du programme).

de communication m'ont conduit à proposer l'intégration d'un support générique de la réactivité aux entrées/sorties au sein même de l'ordonnanceur.

4.2 Structure générale et interfaces

Le cœur d'une bibliothèque de processus légers mixte reste bien sûr l'ordonnanceur. Celui-ci décide que tel processus léger de niveau utilisateur est à ordonnancer sur tel processus léger de niveau noyau (LWP). Mais un ordonnanceur seul ne constitue pas une bibliothèque de processus légers. Il faut lui adjoindre un ensemble de fonctionnalités telles que la création, la destruction et la synchronisation de processus légers qui permettront aux programmes applicatifs d'exploiter au mieux les processus légers.

Notre volonté de proposer une bibliothèque mixte de processus légers à la fois extensible et portable nous a conduits à adopter un développement modulaire. Cette démarche modulaire est intéressante à plusieurs titres. Classiquement, l'approche modulaire facilite le développement des fonctionnalités indépendamment des particularités du système sous-jacent et, réciproquement, permet d'uniformiser l'accès aux ressources du système indépendamment de leur implémentation. Ainsi, les mécanismes de synchronisation entre processus légers utilisateur (verrous, conditions, sémaphores, etc.) peuvent être les mêmes, que la bibliothèque de processus légers soit mixte ou de niveau purement utilisateur. De même, le choix de l'interface de flot parallèle (`thr_` sous SOLARIS, appel système `clone()` sous LINUX, interface `pthread_`) peut être retardé jusqu'à la compilation.

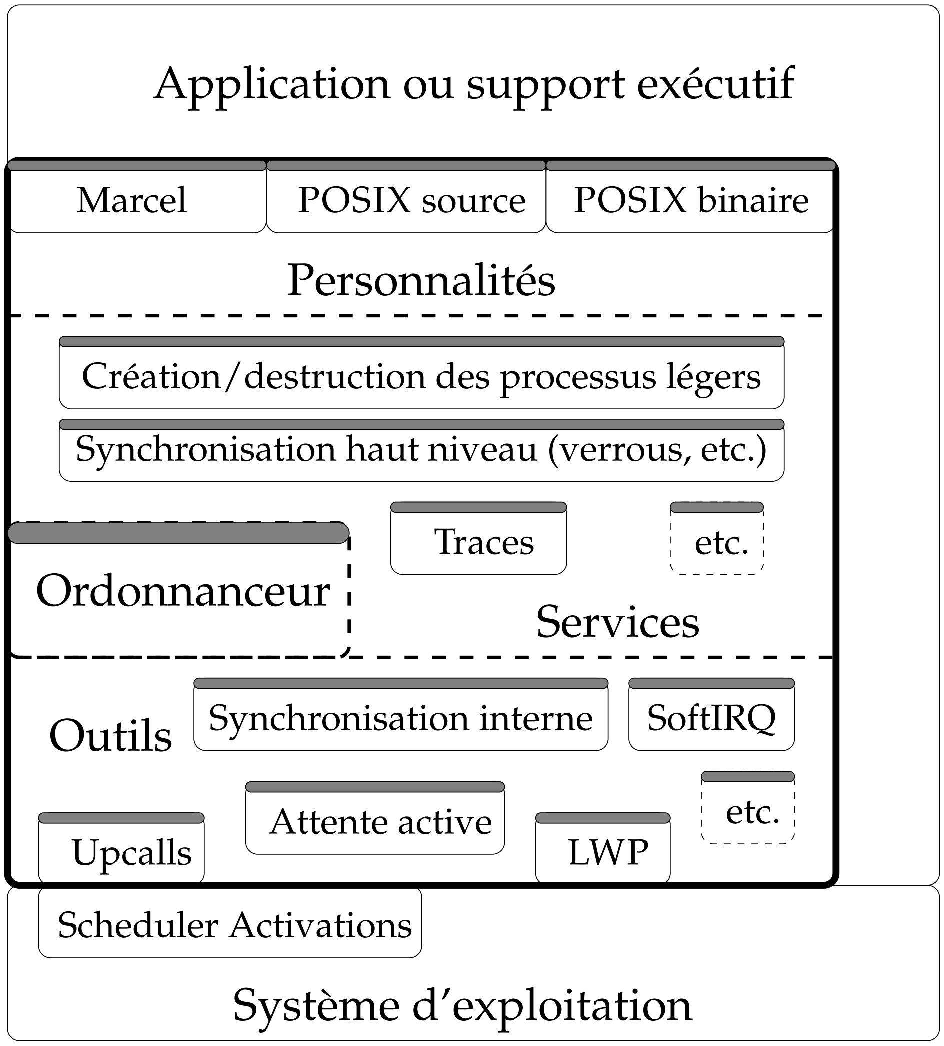
Cette approche modulaire permet aussi de spécialiser le support exécutif en n'y embarquant que les fonctionnalités utiles aux applications ciblées : l'exécution d'une application ainsi construite n'est pas pénalisée par un surcoût que pourrait entraîner l'étendue des fonctionnalités proposées. Par exemple, la plupart des codes de calcul n'ont pas besoin de la fonctionnalité des signaux POSIX, alors qu'une machine virtuelle JAVA le nécessitera.

Dès le départ, l'ordonnanceur de la bibliothèque est lui-même considéré comme un module plutôt que comme l'élément central autour duquel tout le reste s'articule. Pour cela, il est nécessaire de concevoir soigneusement l'architecture afin de permettre le développement des autres parties de la bibliothèque sans les lier à un modèle d'ordonnancement spécifique.

La Figure 4.1 présente l'organisation générale de notre bibliothèque de processus légers. Cette bibliothèque propose aux programmes applicatifs des interfaces (telles que POSIX, MARCEL,..) que l'on dénomme *personnalités*. Elle utilise l'interface *système* pour accéder aux ressources de celui-ci (tels que les processus légers de niveau noyau, les activations, les informations sur l'architecture de la machine, etc.). Enfin, elle est structurée en *composants internes*[3] à savoir :

– l'*ordonnanceur*, *i.e.* le cœur de la bibliothèque ;
– les *services* qui regroupent l'ensemble des fonctionnalités offertes aux programmes applicatifs. Il est possible de sélectionner un sous-ensemble de fonctionnalités en fonction des besoins applicatifs ;
– les *outils* qui regroupent les fonctionnalités et abstractions partagées par les différents composants internes de la bibliothèque.

[3]La notion de composant, ici, doit être comprise comme *entité de structuration logique de la bibliothèque*, et non pas dans le sens d'objet composant lié aux modèles de développement et de déploiement logiciels (composants CORBA, modèle CCM, etc.)



Les interfaces de chaque composant étant définies, on peut développer une nouvelle version d'un composant sans avoir à modifier les autres parties de la bibliothèque. Ainsi, adapter la bibliothèque à une architecture particulière consiste à fournir les composants bas niveau dépendant de l'architecture.

La difficulté de cette approche réside essentiellement dans la mise en évidence des composants et la définition de leur interface :

- *l'interface d'un composant doit être suffisamment expressive et riche afin que son utilisation apporte un réel bénéfice aux autres composants ;*
- *l'interface doit rester concise et générique afin de permettre le développement facile de versions optimisées et/ou spécifiques des composants (par exemple, un ordonnanceur dédié à une application particulière) ;*

FIG. 4.1 – *Structure globale de notre bibliothèque de threads*

4.2.1 Outils

L'ordonnanceur et les services sont construits à l'aide d'une boîte à outils afin de mettre en facteur le code commun, et aussi pour mieux abstraire le matériel et le système. La difficulté est d'identifier les bons concepts à abstraire ainsi que choisir leur niveau d'abstraction. Nous présentons ici un bréviaire représentatif des outils que nous avons développés. Le lecteur pourra se reporter au code de la bibliothèque pour une connaissance plus complète de l'ensemble des outils.

4.2.1.1 Outils d'abstractions du matériel et du système

Il s'agit des outils à adapter pour porter l'ensemble de la bibliothèque sur une architecture matérielle ou sur un système d'exploitation. Grâce à eux, les éléments de plus haut niveau sont entièrement portables.

Outils dépendants de l'architecture. Ces outils, souvent écrits directement en assembleur, sont à adapter lorsque l'on désire porter la bibliothèque sur une nouvelle architecture. On y trouve par exemple :

Gestion de contextes : ce sont des primitives pour basculer d'un contexte à un autre. Pour la plupart des architectures, cet outil est implémenté à partir des primitives `setjmp()` et `longjmp()`. Cependant, certaines architectures particulières nécessitent des développements spécifiques. Ainsi l'`IA64` (architecture de l'ITANIUM) possède une zone de sauvegarde automatique des registres processeurs qu'il faut gérer spécialement ;

Attente active : il s'agit d'un mécanisme permettant de réaliser des exclusions mutuelles à base d'attente active. Il est généralement basé sur des instructions processeurs particulières permettant de tester et modifier atomiquement une valeur en mémoire. Cet outil est utilisé uniquement en mode mixte (en présence de plusieurs processus noyau) de manière à les synchroniser entre eux sans passer par des appels systèmes trop coûteux ;

Cohérence mémoire : c'est un ensemble d'opérations exprimant la cohérence mémoire nécessaire aux autres parties de la bibliothèque : certaines architectures comme l'IA64 ou l'ALPHA nécessitent des instructions assembleurs particulières pour garantir certains schémas de cohérence entre les diverses opérations en mémoire[4]. D'autres architec-

[4]Plusieurs type de cohérences mémoires peuvent être considérées :

`memory_barrier()` : cohérence la plus forte. Toutes les actions précédentes sont effectuées avant de continuer ;

`wmb()` : les écritures précédentes sont effectuées avant de continuer. En effet, certains processeurs comme l'ALPHA peuvent réordonnancer les écritures en mémoire.

`rmb()` : toutes les lectures sont faites avant de continuer.

Par exemple, considérons deux variables `a` et `b` valant respectivement 0 et 1 au départ et le code suivant :

Processeur 0	Processeur 1
`a = 2;`	
`wmb();`	
`b = 3;`	`y = b;`
	`rmb();`
	`x = a;`

Dès qu'on enlève l'une des deux instructions de cohérence mémoire, il devient possible que `y` prenne pour valeur 1 et `x` pour valeur 3.

tures, comme l'IA32, possèdent quant à elles une cohérence très forte des accès à la mémoire ; dans ce cas, ces opérations ne sont simplement pas traduites.

Outils dépendant du système. Notre bibliothèque aura besoin de certains services variant d'un système d'exploitation à l'autre. Les abstraire permet de simplifier le portage de la bibliothèque mais aussi de choisir le service du système le plus adapté quand plusieurs sont disponibles.

Horloge : cet outil permet d'uniformiser la gestion du temps entre les différents systèmes d'exploitation en délivrant régulièrement les événements asynchrones nécessaires aux bibliothèques préemptives. Cet outil doit offrir à l'application une horloge aussi précise que le permet le système cible. Certaines architectures ont un compteur de cycles dans le processeur accessible en mode utilisateur. En l'utilisant, on évite des appels systèmes qui sont toujours coûteux.

Gestion des LWP : cet outil permet de gérer les processus légers de niveau noyau (LWP) des bibliothèques mixtes. Il offre principalement une couche d'abstraction aux interfaces systèmes disponibles, à savoir `thr_` sous SOLARIS, `clone()` sous LINUX et `pthread_` sous tous les systèmes. On pourra se reporter à la partie 5.3.3 pour une discussion sur l'influence de l'interface utilisée sur la réentrance de la bibliothèque de processus légers vis-à-vis des autres bibliothèques du système.

4.2.1.2 Outils génériques

Lors du développement des services de notre bibliothèque, certains problèmes récurrents sont apparus. Plutôt que développer à chaque fois des solutions similaires, il a semblé plus opportun de définir un ensemble d'outils réutilisables. La difficulté a été d'établir une définition correcte de ces outils : il a fallu isoler les concepts à implémenter tout en restant assez générique pour qu'ils soient réellement utilisables par plusieurs composants. Ce travail d'analyse et synthèse a été mené progressivement tout au long de ma thèse, principalement grâce à l'expérience acquise et à l'étude des problématiques similaires dans d'autres projets, comme le noyau LINUX. Voici une brève description de ces principaux outils :

Opérations asynchrones (SoftIRQ) : il arrive que des composants doivent exécuter une action sans pouvoir le faire immédiatement, principalement pour des raisons de cohérence. L'action doit alors être retardée jusqu'à ce que le contexte devienne propice à son exécution. Par exemple, la gestion d'un signal ou d'un *upcall* [5] doit être retardée jusqu'à ce que le processus courant quitte une section critique.

Inspiré du noyau LINUX, ce mécanisme permet aux composants de la bibliothèque de générer des événements de manière asynchrone et de faire exécuter un traitant approprié de façon synchronisée et réentrante avec le reste de la bibliothèque. Ce mécanisme peut être vu comme un modèle d'interruptions logicielles masquables ;

Synchronisation interne : les divers composants de la bibliothèque ont naturellement besoin de se synchroniser entre eux. Il s'agit ici de définir un ensemble d'outils de synchronisation de bas niveau, c'est-à-dire sans intervention de l'ordonnanceur. Les processus légers ne cèdent pas la main à un autre si les ressources souhaitées ne sont pas libres.

[5] Les *upcalls* font partie du modèle des *Scheduler Activations* détaillé à la section 4.4.

Ces primitives sont souvent basées sur les outils d'attente active et de cohérence mémoire. Elles permettent, entre autres, de définir des sections critiques où les SOFTIRQ sont interdites;

Timer : plusieurs services ont besoin de réagir après un certain temps, généralement pour abandonner une opération si elle n'a pas réussi auparavant, comme par exemple pour la fonction `pthread_cond_timedwait()` (attente bornée dans le temps sur une condition). Ce peut être également pour requérir une opération régulièrement; le *timer* est alors reprogrammé à la fin de l'exécution de son traitant.

Cet outil permet donc d'enregistrer une fonction qui sera exécutée après un délai donné. Il est possible d'annuler un *timer* avant son exécution s'il n'est plus nécessaire. La précision des délais dépend de celle de l'outil HORLOGE ;

Tasklet : le mécanisme de SOFTIRQ ne répond pas à tous les besoins. Par exemple, un problème survient lorsque plusieurs processeurs sont disponibles pour exécuter une fonction asynchrone lancée par un composant; celle-ci ne doit pas être exécutée plusieurs fois simultanément.

Les TASKLET répondent à ce besoin en permettant l'exécution de fonctions asynchrones mutuellement exclusives sur SMP. Il s'agit d'un outil construit à l'aide des SOFTIRQ et des mécanismes de synchronisation interne. Comme nous le verrons en section 5.1.2, la mise en œuvre des mécanismes de scrutation repose sur cet outil.

4.2.2 Les services

On appellera services les outils utilisés non seulement par les composants internes mais aussi par l'application elle-même[6].

4.2.2.1 Services de base

Certains services sont indispensables à toute bibliothèque de processus légers car ils constituent le fondement du modèle de programmation multithreads. Toute application, bibliothèque ou support exécutif désirant utiliser des processus légers voudra avoir accès à ces services. Voici les deux principaux services de base :

Gestion de la vie des processus légers. Ce service est responsable de la création et de la terminaison des processus légers. Il s'agit de gérer l'allocation et la libération des ressources nécessaires aux processus légers comme sa pile, ses variables privées, etc. La valeur de retour d'un processus léger, lorsqu'il n'est pas détaché, est également gérée par ce service.

Un système d'attributs attachés aux processus légers permet de préciser certaines de leurs propriétés lors de leur création. Ainsi, on peut spécifier une priorité, une affinité avec un groupe de processeurs, un nom (utile au déverminage), etc. Ce service accepte la spécification de piles externes; une application peut ainsi gérer l'ensemble de ses piles et peut, par exemple, migrer ses processus légers d'une machine à une autre.

[6] La distinction entre outil de haut niveau et service peut sembler parfois un peu formelle : un service peut offrir une interface étendue à des composants internes de la bibliothèque, etc.

Synchronisation. Ce service regroupe les mécanismes classiques de synchronisation de processus légers. Contrairement au cas de l'attente active, il s'agit d'ordonnancer un autre processus léger lorsque celui qui est en cours d'exécution doit attendre une ressource. L'ordonnanceur est sollicité pour endormir ou réveiller un processus léger. La cohérence de ces actions (test de l'état courant du processus léger, réveil, endormissement, etc.) est assurée grâce aux outils de synchronisation interne. Les principaux mécanismes de synchronisation offerts sont : les moniteurs (verrous et conditions), les sémaphores et les barrières.

Ces trois paradigmes de synchronisation sont théoriquement redondants, mais il est plus efficace de définir une implémentation spécifique pour chacun d'eux. De plus, on peut envisager de spécialiser l'implémentation de ces primitives en fonction de l'architecture sous-jacente. Par exemple, l'implémentation des barrières sur machine avec accès mémoire non uniforme (NUMA) pourrait se faire de manière hiérarchique avec tout d'abord une barrière pour les nœuds locaux puis les "leaders" de chaque nœud se synchroniseraient.

4.2.2.2 Services optionnels

Outre les services indispensables à toute bibliothèque de processus légers, nous proposons un ensemble de services « optionnels », en ce sens qu'ils ne seront inclus à la compilation qu'en fonction de l'application à exécuter. Certains de ces services peuvent donc se révéler essentiels pour une application donnée. Cela permet d'éviter des surcoûts inutiles à certaines applications tout en offrant des fonctionnalités avancées aux autres. Voici une brève présentation de quelques uns de ces services optionnels :

Gestion des signaux POSIX : la gestion des signaux de manière conforme à la norme POSIX est délicate et complexe. La plupart des programmes multithreadés de calcul parallèle n'utilisent pas les signaux et le surcoût de leur gestion complète est inutile [7].

Services pour la réactivité : un ensemble de services pour améliorer la réactivité des processus légers aux événements extérieurs est proposé. Il est décrit en détail dans la section traitant de la réactivité (section 4.3).

Traces pour mesures et déverminage : un programme peut générer des traces pour aider au déverminage de l'application et à la compréhension de ses performances. Le surcoût, même faible, est inutile aux applications en exploitation. Ce service est décrit en détail dans le chapitre 6.

4.2.3 Les ordonnanceurs

L'ordonnanceur est le cœur de toute bibliothèque de processus légers. Il est responsable des décisions d'ordonnancement, c'est-à-dire du choix du processus léger applicatif à exé-

[7] La plupart des applications de calcul hautes performances n'ont aucun intérêt à être exécutées par un support respectant fidèlement la norme POSIX quant aux signaux. En effet, pour qu'une bibliothèque de processus légers de niveau utilisateur implémente correctement cette norme, elle doit prendre en compte qu'un signal peut être généré à tout instant par le noyau. Or, un changement de contexte entre processus légers de niveau utilisateur nécessite, d'une part, de changer le masque de signal courant pour l'adapter à celui du nouveau processus léger, et d'autre part, de sauver puis de restaurer les registres du processeur. Ainsi, pour éviter qu'un signal bloqué par un processus léger puisse lui être délivré par erreur entre les deux opérations, il est nécessaire de bloquer tous les signaux avant le changement de contexte et de restaurer ensuite le masque correct. Cela impose donc des appels systèmes à chaque changement de contexte particulièrement coûteux pour une bibliothèque de niveau utilisateur. On notera que la bibliothèque de processus légers à deux niveaux NGPT (cf. section 3.4.3.2 page 46) a un coût de changement de contexte important en raison de ces deux appels systèmes à `setprocmask()`.

cuter dans chacun des LWP disponibles. L'interface de l'ordonnanceur possède peu de primitives : le processus léger peut changer son état (prêt/bloqué/mort), passer la main ou encore réveiller un processus léger endormi. L'interface est suffisamment souple pour ne pas imposer un type précis d'ordonnanceur. Notre expérience des versions précédentes de MARCEL (ordonnanceur en $O(1)$ mono- ou multifile) ainsi que l'étude approfondie des ordonnanceurs du noyau LINUX (file unique comme pour les anciens noyaux LINUX, multifile en $O(1)$ des noyaux 2.6, avec gestion de domaines[Cor04] en cours de développement, etc.) nous a convaincus que cette interface est adaptée à de nombreux types d'ordonnanceurs.

4.2.3.1 L'ordonnanceur par défaut

La bibliothèque MARCEL est fournie avec un ordonnanceur générique qui possède une file globale, des files spécifiques aux LWP et deux niveaux de priorités. Par défaut, les processus légers ont la priorité standard et sont rangés dans la file globale. L'application peut modifier ce comportement via une primitive de changement de priorité ou par l'utilisation d'attributs spéciaux à la création des processus légers. Aucun équilibrage de charge automatique n'est fait entre les files spécifiques aux LWP. Il est assez simple de rajouter des niveaux de priorité et/ou des files supplémentaires attachées, par exemple, à un sous-groupe de processeurs.

La distinction entre file unique et file spécifique à un (ou quelques) LWP prend toute son importance dès que la machine possède plusieurs processeurs. En cas de file unique, on pourra observer une forte contention au niveau du verrou de protection de la file ; cette contention est en particulier observable sous le noyau LINUX jusque dans ses versions 2.4.x. En revanche, si chaque LWP a sa propre file, les possibilités de contention se limitent aux insertions (resp. retraits) des processus légers réveillés (resp. endormis), mais il devient alors nécessaire d'assurer l'équilibrage de la charge sur les différentes files.

4.2.3.2 D'autres ordonnanceurs

L'interface entre l'ordonnanceur et le reste de notre bibliothèque est assez réduite et permet donc son remplacement *relativement* facilement. Il est malgré tout très important de remarquer que l'ordonnanceur n'est pas lui-même une entité indépendante vis-à-vis de l'exécution du reste du programme. Les décisions d'ordonnancement sont prises individuellement par chaque LWP. En particulier, il n'y a rien, *a priori*, qui permette à un LWP donné d'ordonnancer à cet instant un ensemble de processus légers sur un ensemble de LWP. Les décisions d'ordonnancement sont prises de manière décentralisée. Les LWP peuvent juste choisir le processus léger à ordonnancer en leur sein et éventuellement positionner des variables que les autres LWP consulteront et prendront en compte lorsqu'ils se rendront, eux aussi, à un point d'ordonnancement.

Bien que facilitées par la présence d'une interface concise, la création et la mise au point d'un nouvel ordonnanceur restent complexes. En effet, les mécanismes de protection et de synchronisation sont délicats à mettre en œuvre correctement, en particulier si l'on désire éviter une importante contention comme celle provoquée par un verrou global protégeant l'ensemble des données d'un ordonnanceur. Une piste de recherche intéressante paraît être de développer des ordonnanceurs directement paramétrables par l'application. On demande alors directement à l'application ses critères d'élection du processus à ordonnancer. Ou encore, on développe des interfaces plus abstraites, afin que l'application transmette à

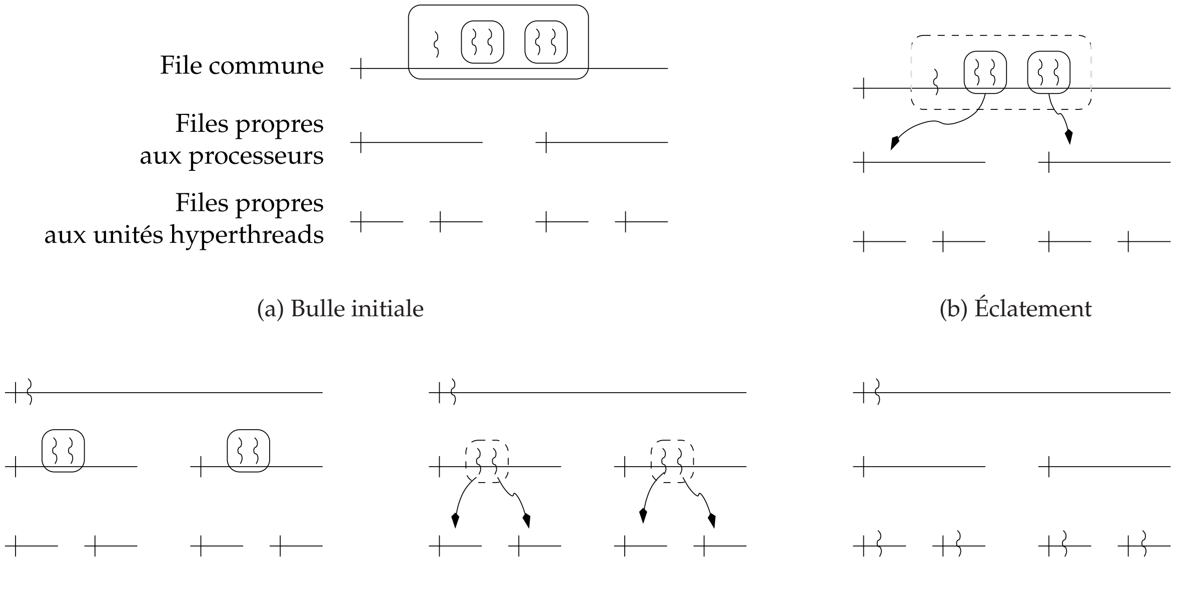
Ici, les processus légers sont structurés par l'application au moyen de *bulles*, éventuellement imbriquées. De plus, ces bulles possèdent des propriétés d'ordonnancement relatives à la façon d'ordonnancer ou de répartir les entités (sous-bulles ou processus légers) qui les composent, par exemple :

– les entités ont peu de dépendances entre elles, donc elles s'exécutent parfaitement en parallèle ;
– les entités travaillent sur des données communes, donc il faudrait les placer sur des processeurs assez proches vis-à-vis de la hiérarchie mémoire.

Parallèlement à ces bulles, l'ordonnanceur fournit un ensemble de files d'ordonnancement correspondant à chaque hiérarchie de processeur. Ainsi, pour une machine biprocesseur hyperthreadée, l'ordonnanceur définit $1 + 2 + 4 = 7$ files :

– une pour les travaux pouvant s'exécuter sur n'importe quelle unité de calcul ;
– deux pour chacun des deux processeurs. Un travail dans une telle file s'exécutera nécessairement sur le processeur correspondant ;
– quatre pour chacune des quatre unités de calcul (deux *threads* par processeur). Là encore, un travail placé dans une telle file s'exécutera uniquement dans l'unité de calcul correspondante.

Le programme peut alors diriger l'ordonnancement en exprimant comment l'ordonnanceur doit essayer de répartir les entités d'une bulle lorsqu'il ordonnance celle-ci. On peut descendre ou non dans la hiérarchie des files, ce qui localise ou non les travaux. On peut également rassembler les travaux sur une même file ou, au contraire, les disperser sur des files différentes, ce qui permet de favoriser le parallélisme ou bien la localité lors de l'exécution des travaux.



(a) Bulle initiale

(b) Éclatement

(c) Répartition sur les processeurs

(d) Éclatement des sous-bulles

(e) Répartition sur les unités hyperthreadées

4.2(a) : *la plus grande bulle a été placée sur la file générale ;*

4.2(b) : *elle éclate, libérant une tâche de gestion qui peut s'exécuter n'importe où (file générale) et deux sous-bulles dont les travaux ont vocation à être exécutés en parallèle sur des processeurs différents ;*

4.2(c) : *les deux sous-bulles sont placées sur des processeurs différents ;*

4.2(d) : *les deux sous-bulles éclatent en parallèle, libérant chacune deux tâches de calcul à exécuter sur des unités ayant une très forte affinité mémoire ;*

4.2(e) : *on profite de la technologie* SMT *pour répartir les tâches de calcul sur les unités parallèles des processeurs. Elles peuvent démarrer en parallèle.*

FIG. 4.2 – Ordonnanceur dirigé par l'application

l'ordonnanceur des informations sur les relations et interactions de ses différents processus légers. Ainsi, au sein de notre équipe, Samuel Thibault [Thi04] a implémenté un ordonnanceur générique souple et adaptable sans manipulation des mécanismes de synchronisation interne. En exprimant des propriétés de haut niveau (ensemble de processus légers en interaction forte, processus de calcul intensif, etc.), l'application peut diriger cet ordonnanceur afin de mettre en œuvre une politique de répartition des processus légers sur les processeurs, une politique de *gang-scheduling* pour des groupes de processus légers ou une politique d'affinité d'un processus léger avec un processeur, un cache, voire un banc mémoire particulier sur une architecture hiérarchisée (processeurs SMT et multicore, architectures SMP et NUMA). La Figure 4.2 présente succinctement cet ordonnanceur.

Enfin, le fait d'avoir calqué l'interface de l'ordonnanceur et celles de certains outils de bas niveau sur celles du noyau LINUX simplifie l'intégration à notre bibliothèque des travaux réalisés par la communauté LINUX sur le noyau. Par exemple, on peut ainsi bénéficier des travaux de l'environnement BOSSA [8] [PB03, LMD04] qui introduisent une technique pour modifier dynamiquement l'ordonnanceur d'un noyau LINUX monoprocesseur. Les politiques d'ordonnancement sont décrites dans un langage spécialisé adapté puis utilisées par le noyau.

4.2.4 Les personnalités

Chacun reconnaît le rôle fondamental de l'interface d'une bibliothèque. Toutefois, cette notion recouvre différents aspects. Il convient, de fait, de distinguer deux sortes d'interfaces :

l'interface binaire, également nommée ABI pour *Application Binary Interface*. Il s'agit de l'interface entre bibliothèques et programmes *déjà compilés*. Cette interface définit les conventions d'appels de fonction (au niveau du code machine), la liste des symboles (fonctions et variables) disponibles dans une bibliothèque, la composition des objets manipulés (taille et placement des champs des structures de données partagées), etc. ;

l'interface de programmation, également nommée API pour *Application Programming Interface*. Il s'agit de l'interface au niveau du langage de programmation. Elle définit les noms des objets[9] que le programmeur peut utiliser dans son code sans avoir à se soucier de l'implémentation interne de ces objets : il peut être amené à manipuler une structure de données sans connaître l'organisation interne de cette structure ni même sa taille.

Nous verrons ci-dessous que notre bibliothèque sera amenée à proposer plusieurs interfaces différentes aux programmes. Pour pouvoir les distinguer facilement, nous avons nommé *personnalité* toute interface de notre bibliothèque permettant de manipuler les processus légers. Voyons maintenant pourquoi plusieurs personnalités se sont révélées nécessaires.

4.2.4.1 Fonctionnalités

L'interface de programmation proposée par le standard POSIX est incontournable car utilisée de nos jours par la majorité des applications et bibliothèques manipulant des proces-

[8] `http://www.emn.fr/x-info/bossa/`

[9] Il s'agit ici des objets au sens large (fonction, variable, macro, type, structure, etc.) et non pas des objets des « langages objets ».

sus légers. Toutefois, notre bibliothèque utilise dans certaines configurations la bibliothèque de processus légers du système pour gérer ses LWP. Employant les symboles définis par le standard POSIX, elle ne peut pas les offrir elle-même aux applications. C'est pourquoi MARCEL propose une première personnalité, nommée **POSIX source** qui est identique à celle définie par la norme POSIX au préfixe près : le préfixe `pmarcel_` remplace le préfixe `pthread_` des fonctions, types, etc. Cela permet ainsi un portage rapide des programmes et bibliothèques sur MARCEL.

Cependant, l'interface POSIX n'est pas suffisamment riche. Aussi, pour profiter des fonctionnalités supplémentaires offertes par la bibliothèque MARCEL, un programme ou une bibliothèque devra utiliser une interface étendue. Ces fonctions sont préfixées par `marcel_`. Outre les nouvelles fonctionnalités, elles reprennent également les fonctions de la norme POSIX, mais en les optimisant. Cette optimisation, lorsqu'elle est possible, peut prendre plusieurs formes dont l'utilisation de code en ligne (*inline*) qui évite un appel de fonction, ou encore l'abandon de spécificités de la norme POSIX rarement employées. Par exemple, pour les verrous (*mutex*), seul le type standard est implémenté pour la version `marcel_` alors que la version `pmarcel_` supporte aussi les verrous récursifs ou ceux avec vérification d'erreur. L'ensemble de ces fonctions définit la personnalité **Marcel** ; celle-ci est la personnalité la plus écartée de la norme mais, en contrepartie, elle offre des optimisations et des fonctionnalités supplémentaires.

4.2.4.2 Compatibilité et réentrance

L'utilisation de bibliothèques externes peut provoquer des problèmes de réentrance vis-à-vis des processus légers. Plusieurs cas peuvent se présenter :

1. la bibliothèque n'utilise aucune donnée de classe de stockage globale, elle fonctionne très bien en environnement multithreadé sans aucune modification ;
2. la bibliothèque utilise des données globales mais rien n'a été prévu pour son utilisation en environnement multithreadé ;
3. la bibliothèque utilise des données globales mais protège ses structures grâce aux primitives de synchronisation de processus légers de la norme POSIX.

De nos jours, la plupart des bibliothèques correspondent aux cas 1 ou 3. Mais quelques unes n'ont pas été modifiées pour supporter le multithreading. C'est le cas de la bibliothèque de communication bas niveau GM pour les réseaux MYRINET.

Le cas 1 ne pose pas de problème. Pour utiliser une bibliothèque non protégée (cas 2), un programme multithreadé utilisant MARCEL ou la bibliothèque du système devra protéger lui-même ses accès à la bibliothèque en question en s'interdisant, par exemple grâce à un verrou, de lui faire deux appels simultanés.

Le cas 3 est plus problématique : puisque MARCEL gère lui-même les processus légers utilisateur, le système ne les connaît pas. Et lorsque la bibliothèque demande au système de protéger les accès à ses variables globales, le système n'est d'aucune aide puisque, de son point de vue, il n'y a qu'un seul flot d'exécution. Trois approches sont alors possibles pour gérer cette situation :

recompiler la bibliothèque en lui faisant utiliser les primitives MARCEL plutôt que celles du système. Cela résout les problèmes ; la bibliothèque peut même utiliser les primitives optimisées de MARCEL. Par contre, cela nécessite le code source de la bibliothèque et sa recompilation. Cette solution sera donc plutôt adaptée aux bibliothèques directement associées au programme développé ;

considérer que la bibliothèque n'est pas protégée des processus légers et la traiter comme le cas 2. Cette solution sera utilisée quand la bibliothèque n'est pas recompilable (sources non disponibles) ou que l'effort de recompilation est trop important vis-à-vis du gain en parallélisme espéré en évitant un verrou global protecteur (cas de la plupart des bibliothèques propres d'un système) ;

rendre MARCEL compatible au niveau binaire en adoptant dans MARCEL les mêmes structures que la bibliothèque du système considéré.

Ce dernier cas possède l'immense avantage de rendre automatiquement réentrante vis-à-vis de MARCEL toute bibliothèque réentrante vis-à-vis de la bibliothèque du système. Toutefois, cela nécessite une importante adaptation de MARCEL [10]. De plus, il faut que la bibliothèque de processus légers du système soit indépendante du reste du système, en particulier de la bibliothèque `C`, sans quoi il faudrait adapter tout le système. Il faut également pouvoir manipuler des LWP sans utiliser les processus légers du système. Pour l'instant, cette adaptation n'a été réalisée que pour le système LINUX sur l'architecture IA32 [11].

Lorsque que ce support est possible, cette nouvelle interface constitue ce que nous avons appelé la personnalité **POSIX binaire**. Il faut bien noter que devoir respecter les contraintes supplémentaires imposées par l'ABI de la bibliothèque système oblige parfois à des contorsions qui pénalisent les optimisations de notre bibliothèque. Cette personnalité est très utile pour permettre à des bibliothèques tierces d'être réentrantes vis-à-vis de MARCEL, mais elle ne permet pas de profiter de toute la richesse de notre bibliothèque.

Afin d'être le plus souple possible, notre bibliothèque permet d'inclure plusieurs personnalités lorsqu'elle est compilée. Il est alors de la responsabilité de l'application d'utiliser ces différentes personnalités de manière cohérente. Un objet (processus léger, verrou, etc.) créé à travers une personnalité doit toujours être manipulé avec cette même personnalité. En effet, des objets similaires de deux personnalités différentes, par exemple un verrou « MARCEL » et un verrou « POSIX binaire » auront très probablement des structures internes différentes. Appeler `pthread_mutex_lock()` puis `marcel_mutex_unlock()` sur le même objet a de grandes chances de conduire à des erreurs à l'exécution. Dans la pratique, on utilisera les personnalités :

POSIX binaire pour les bibliothèques tierces déjà compilées ;

POSIX source pour les applications déjà écrites de manière compatible POSIX ;

MARCEL pour bénéficier des fonctionnalités supplémentaires offertes.

4.2.5 Bilan

Pour concevoir la bibliothèque MARCEL nous avons produit un important effort de génie logiciel. Cet effort structurel était indispensable pour obtenir une bibliothèque facilement et hautement spécialisable sur de nombreux systèmes et architectures. L'obtention d'une bibliothèque adaptable et flexible tout en restant efficace a nécessité un travail important

[10] les structures binaires des objets, par exemple celles codant les verrous, doivent devenir compatibles (même taille, même utilisation des champs).

[11] Sur ce système, la bibliothèque de processus léger du système est bien séparée des autres bibliothèques systèmes. De plus, les LWP peuvent être manipulés à travers l'interface autour de l'appel système `clone()`.

d'identification des composants et de leur interface. Il a fallu parvenir à identifier les bons concepts à abstraire de manière à pouvoir écrire un seul *code caméléon* qui se spécialisera lors de la compilation.

Reprenons l'exemple emblématique de l'ordonnanceur. Son interface simple et générique ne contraint pas l'organisation interne. Cela permet d'envisager des ordonnanceurs spécifiques à certaines applications particulières. Par ailleurs, le code de l'ordonnanceur par défaut est écrit de manière portable vis-à-vis des architectures, c'est-à-dire que l'on utilise des LWP (bibliothèque mixte) ou non (bibliothèque de processus légers de niveau purement utilisateur). Ceci s'obtient grâce à l'utilisation des outils d'abstractions que nous avons conçus et qui sont intensément utilisés dans le code de l'ordonnanceur. Ces outils se spécialisent à la compilation. Par exemple, lorsque la bibliothèque à construire est de niveau purement utilisateur, les primitives de synchronisation interne sont grandement simplifiées[12]. En définitive, grâce à la structuration retenue, la bibliothèque est constituée d'un seul code qui se spécialise à la compilation, facilitant le cycle de vie de notre bibliothèque et celui de ses programmeurs !

4.3 Réactivité aux entrées/sorties

Un axe important de mes travaux est la prise en compte de la réactivité dans les programmes multithreadés. Nous avons vu dans la partie 2.3.2 combien cette notion est cruciale pour les environnements multithreadés complexes dans le cadre du calcul hautes performances. Être réactif à un événement suppose d'abord de détecter cet événement. Or, il existe de nombreuses méthodes de détection, chacune ayant son intérêt et ses inconvénients, mais chacune imposant aussi une structuration du code applicatif et des contraintes d'utilisation spécifiques. Après avoir passé en revue les principales méthodes de détection d'événements, nous verrons comment nous avons abstrait cette diversité au sein de notre bibliothèque, rendant, de ce fait, l'application indépendante (et ignorante) des méthodes de détection réellement mises en œuvre à l'exécution.

4.3.1 Les diverses méthodes de détection d'événements

Nous nous intéressons ici aux événements générés de manière externe à l'application par un autre processus, par une bibliothèque indépendante, ou encore par des périphériques matériels (carte réseau, contrôleur disque, etc.) En effet, il existe suffisamment de moyens de synchronisation au sein d'une bibliothèque de processus légers (moniteurs, sémaphores, barrières, etc.) pour pouvoir traiter efficacement les situations où la génération et la réception d'un événement sont internes au programme multithreadé. Un événement peut être notifié ou détecté. La notification est réalisée par l'environnement du programme qui modifie son déroulement normal pour le prévenir alors que la détection est faite à l'initiative du programme lui-même. Les méthodes correspondantes seront alors dites respectivement *passives* et *actives*.

[12] Sans LWP multiples, la synchronisation interne ne requiert pas l'utilisation de mécanismes d'attente active. Cet aspect est détaillé dans la section 5.1.1.

4.3.1.1 Les méthodes passives

Dans les méthodes passives, l'événement externe, comme par exemple une interruption matérielle d'une carte réseau, doit agir sur le programme. Pour des raisons évidentes de sécurité, dans les systèmes actuels, une application ne peut pas installer son propre traitant d'interruptions matérielles en espace utilisateur. Ces dernières sont donc nécessairement récupérées et gérées par le système d'exploitation. Pour faire propager ce type d'événement vers une application, les systèmes d'exploitation utilisent deux techniques : les signaux et les réveils d'appels systèmes bloquants.

À propos des signaux. Au niveau conceptuel, les signaux sont très semblables aux interruptions matérielles : le flot d'exécution d'un programme (ou d'un processus léger) est interrompu pour exécuter le traitant du signal installé par l'application lors de la délivrance du signal en question. Les signaux peuvent être ignorés ou masqués. Dans le cas de processus légers, la norme POSIX considère que les traitants de signaux sont globaux à l'application alors que les masques de signaux sont propres aux processus légers. Les signaux permettent donc de détecter des événements extérieurs. Plusieurs contraintes cependant limitent leur intérêt :

implémentation : la synchronisation des traitants de signaux avec le reste du code doit se faire à l'aide des fonctions de masquage de signaux. Elle nécessite également une implémentation correcte des signaux POSIX pour les processus légers utilisateur qui, eux aussi, auront éventuellement à modifier le masque de signal à chaque changement de contexte. En mode utilisateur, ceci génère des appels systèmes qui rendent coûteuse cette technique ;

réentrance : les traitants de signaux ne peuvent appeler qu'un sous-ensemble réduit de fonctions. Alors que de nombreuses bibliothèques sont *thread-safe* (*i.e.* peuvent être appelées depuis des processus légers différents simultanément), très peu sont *async-safe* (*i.e.* peuvent être appelées depuis un traitant de signal à tout instant). Par exemple, les primitives de synchronisation de processus légers de la norme POSIX (verrous, conditions, sémaphore, etc.) ne peuvent pas être appelées depuis un traitant de signal en toute sécurité, ce qui serait utile, par exemple, pour réveiller un processus léger à l'arrivée d'un événement ;

disponibilité : dans les systèmes actuels, l'arrivée d'un événement ne provoque malheureusement pas systématiquent le déclenchement d'un signal. La norme POSIX AIO (IEEE STD 1003.1-2001) a standardisé, en partie, l'interface de programmation des événements. Mais, d'une part, tous les événements d'un système ne sont pas couverts, d'autre part, les gestionnaires matériels sous-jacents n'implémentent pas nécessairement cette fonctionnalité. C'est en particulier le cas pour les gestionnaires des cartes de réseaux rapides comme SCI ou MYRINET.

fonctionnalités : même lorsque le système propose cette interface, elle ne possède pas toujours toutes les fonctionnalités désirées. Ainsi, les noyaux LINUX récents permettent de générer un signal lorsqu'un descripteur de fichier est disponible. Malheureusement, on ne peut pas spécifier les états intéressants : si un descripteur de fichier est ouvert en lecture et en écriture (ce qui est le cas de l'entrée/sortie standard dans un shell) et que l'on désire attendre des données disponibles en entrée, on risque de recevoir continuellement des signaux nous informant qu'il est possible d'écrire dans ce descripteur de fichier...

Les signaux constituent un moyen très pratique à une application pour être alertée des événements asynchrones. Cependant, cette technique n'est pas toujours disponible tant au niveau du système d'exploitation qu'à celui du gestionnaire de périphérique. L'absence de support systématique ou la présence fréquente de mauvaises propriétés comme la non réentrance de fonctions à leur encontre expliquent que ce mécanisme ne puisse pas être systématiquement utilisé.

Appels systèmes bloquants. Les appels systèmes bloquants sont le second moyen pour le noyau de propager l'arrivée d'une interruption matérielle vers l'espace utilisateur : le processus fait un appel au noyau qui le suspend tant que l'événement n'est pas survenu. C'est le mécanisme généralement utilisé dans la plupart des applications, les appels systèmes bloquants par exemple `read()`,`write()`,`connect()`.

Ce mécanisme repose sur la possibilité pour le système d'exploitation de suspendre un flot d'exécution. Ceci constitue un problème dans notre situation[13]. Si un processus léger utilisateur effectue un appel système bloquant, l'ensemble des processus légers sera alors bloqué ! En effet, le noyau n'ayant pas connaissance des autres processus légers utilisateur, il ne peut pas leur donner la main. Dans le cas d'une bibliothèque à deux niveaux, seul le LWP dans lequel un processus léger utilisateur a exécuté un appel système bloquant est suspendu. Les autres LWP peuvent continuer à s'exécuter. Cependant, les bibliothèques à deux niveaux cherchent à exploiter au mieux les machines multiprocesseurs en créant autant de LWP que de processeurs. Aussi, chaque appel système bloquant réduit fortement les moyens de calcul disponibles pour l'application.

En pratique, les applications utilisant une bibliothèque de processus légers de niveau utilisateur évitent les appels systèmes bloquants, soit explicitement lors de leur écriture, soit par un détournement des fonctions incriminées par la bibliothèque de processus légers elle-même. Des versions non bloquantes de ces primitives sont alors utilisées, associées à des mécanismes d'attente active (scrutation).

4.3.1.2 Les méthodes actives

Nous avons vu les deux principales techniques utilisées par les systèmes d'exploitation pour notifier l'arrivée d'un événement aux applications. Voyons maintenant comment une application peut détecter elle-même la disponibilité d'un événement sans pour autant se bloquer. Les techniques classiques de détection sont l'utilisation d'un appel système non bloquant ou la consultation d'un emplacement mémoire : certains réseaux rapides comme SCI permettent de réaliser ainsi des communications entièrement en espace utilisateur. Si l'on désire capter des événements dont l'arrivée est temporellement imprévisible, il est alors nécessaire de scruter régulièrement l'état de la ressource. Le problème qui se pose alors aux applications est de mettre en œuvre une scrutation régulière.

Pour les programmes séquentiels, la mise en œuvre d'une telle scrutation revient à la simple insertion de tests dans le code source. On trouve ainsi de nombreux programmes MPI utilisant des communications asynchrones dont le code contient des appels à la fonction `mpi_testany()`. Le compilateur peut également aider le programmeur en insérant

[13] Tous nos processus légers applicatifs sont de niveau utilisateur.

ces scrutations à des endroits judicieusement déterminés par analyse du code et des graphes d'appels. Cependant, cette analyse n'est pas si aisée et fait encore l'objet de travaux de recherche [Joh95]. Évidemment, si le programme appelle des fonctions d'une bibliothèque extérieure, aucune scrutation ne sera effectuée durant ces appels.

Un programme multithreadé peut profiter de la multiplicité des flots d'exécution en déléguant la scrutation à un processus léger. La difficulté est de faire en sorte que ce processus léger prenne la main à intervalle régulier, afin de *garantir* une certaine fréquence de scrutation. En effet, s'il a la main trop souvent, il consommera trop de temps de calcul au détriment de l'application. S'il a rarement la main, un événement pourra attendre longtemps avant d'être pris en compte, rendant l'application peu réactive. Les interfaces standards des bibliothèques de processus légers offrent peu de moyens pour régler la fréquence d'ordonnancement d'un processus léger : elles proposent la fonction `yield()` qui interrompt le processus léger courant pour redonner la main à l'ordonnanceur, et parfois, une gestion des priorités[14]. La fonction `yield()` ne permet pas de garantir la fréquence des scrutations : plus le système aura de processus légers, moins le processus léger effectuant la scrutation aura la main, et donc, moins le programme sera réactif. Les priorités ne permettent pas non plus de résoudre ce problème : la priorité des processus scrutateurs devrait être fonction du nombre total de processus légers [15]. Les techniques standards manquent d'adaptabilité et n'offrent pas de solution satisfaisante à ce problème.

4.3.1.3 Discussion

Nous avons présenté les deux grandes familles de techniques permettant de récolter des événements externes. Comme souvent, aucune technique n'est meilleure qu'une autre dans l'absolu ; son efficacité dépend de son contexte d'utilisation. Parfois même, combiner plusieurs techniques permet d'obtenir une meilleure réactivité. Ainsi, dans [LRBB96], un système multithreadé de niveau utilisateur utilise à la fois la scrutation et les interruptions pour réagir aux événements issus d'un réseau de type MYRINET. Comme cette technologie permet aux applications de dialoguer avec la carte réseau sans passer par le noyau, le coût d'une scrutation est très faible. Aussi, dans le prototype développé, la scrutation est-elle utilisée lorsqu'aucun processus léger n'est prêt et évite-t-elle le surcoût d'une interruption matérielle. En revanche, lorsque des processus légers sont ordonnancés, rendant difficile la mise en place d'une scrutation régulière, les messages réseaux génèrent alors des interruptions[16].

Remarquons également que l'utilisation d'une technique particulière telle que les signaux, les appels systèmes bloquants et la scrutation influence la structure générale du programme. Ceci pose les problèmes suivants :

– une fois la technique choisie, l'application devient dépendante de cette technique et difficilement portable sur des architectures où cette méthode n'est pas utilisable ;
– le choix de la solution optimale dépend fortement du contexte d'exécution. Par exemple, les appels systèmes bloquants sont optimaux pour les connections TCP, pas

[14]Certains systèmes comme LINUX requièrent des droits d'administration pour qu'une application puisse utiliser des priorités fortes, appelées « priorités temps réel ».

[15]Dans un programme avec peu de processus légers, celui effectuant les scrutations ne devra pas avoir une priorité trop élevée pour ne pas consommer trop de temps processeur. En revanche, en présence de nombreux processus légers, il devrait avoir une priorité importante pour prendre la main régulièrement.

[16]Le comportement intelligent de la carte réseau (générer ou non des interruptions) est rendu possible grâce aux possibilités de programmation des cartes MYRINET.

nécessairement pour un réseau rapide de type MYRINET. Or, l'application ne connaît pas nécessairement à l'avance sur quel réseau elle fonctionnera ;
– de nouvelles techniques peuvent être introduites, par exemple l'interface `epoll` [Lib01] des noyaux LINUX 2.6.

Aussi, afin d'adapter facilement une application aux nombreuses techniques de récolte d'événements présentes ou à venir, il semble judicieux d'abstraire ces services et de les proposer à l'application à travers une interface uniforme. La conception de ce nouveau service devra prendre en compte des contraintes issues des environnements multithreadés (appels systèmes bloquants, fréquence de scrutation, etc.)

4.3.2 Notre proposition : un serveur uniforme d'événements asynchrones

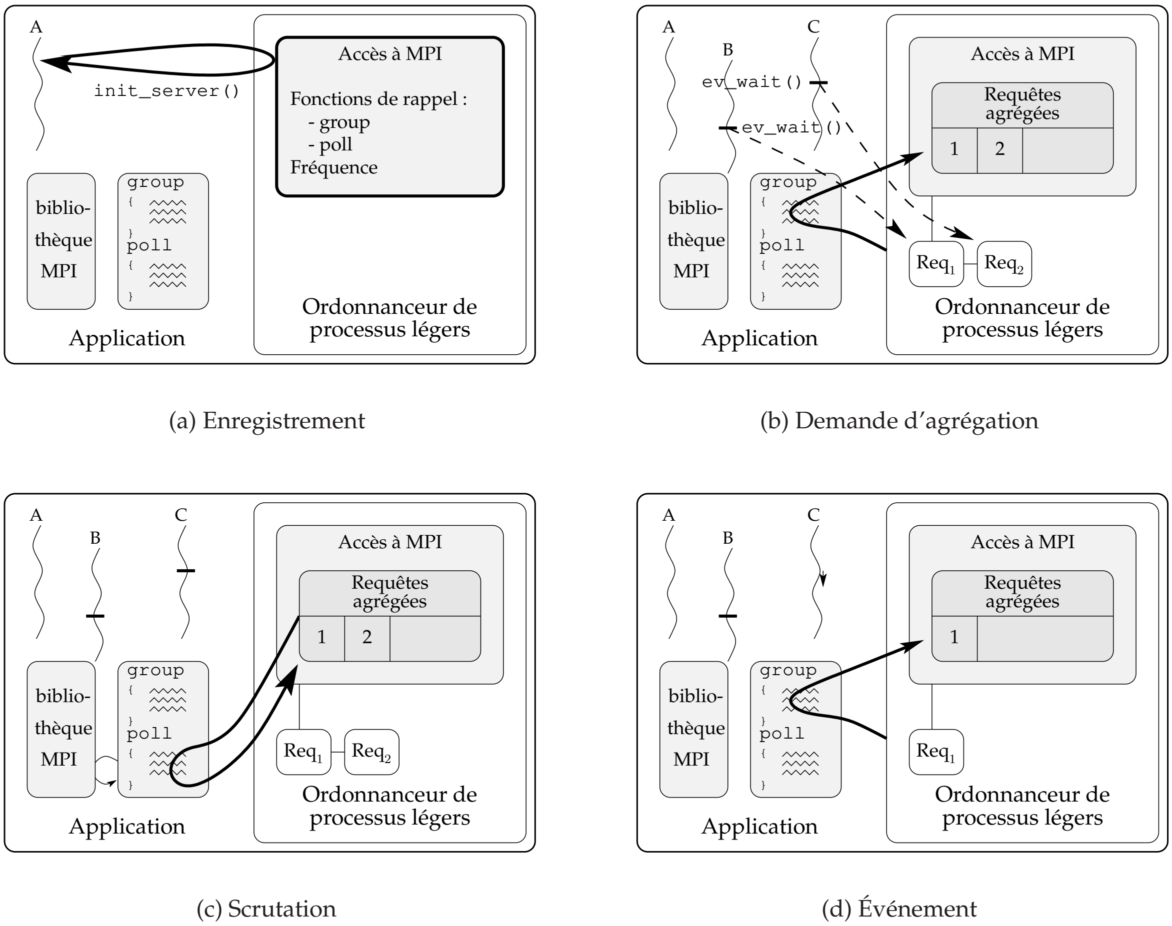
Afin de résoudre au mieux le problème de réactivité des applications multithreadées, nous proposons de créer un *serveur d'événements asynchrones*. L'idée est qu'à son démarrage, l'application enregistre des fonctions de rappels (*callbacks*) auprès du serveur. Lorsqu'un processus léger est en attente d'un événement, il soumet sa requête au serveur qui utilise alors un ou plusieurs *callbacks* pour déterminer si l'événement est présent. Cette structuration autour d'un serveur d'événements permet de bien séparer l'application de la méthode de détection employée. Mais l'élément primordial de notre proposition est d'intégrer ce serveur, non pas à la bibliothèque de communication, mais bien à celle des processus légers, et ce, pour plusieurs raisons :

Un choix éclairé de la méthode de détection. L'enregistrement d'un *callback* n'étant réalisé qu'à l'exécution, il est possible d'utiliser la méthode de détection la plus performante pour la configuration courante.

Une synchronisation plus aisée. L'intégration du serveur à la bibliothèque permet de proposer des primitives très adaptées de synchronisation entre les processus légers et les *callbacks*. Ainsi, les primitives classiques de synchronisation n'ont pas à subir de surcoût comme, par exemple, désactiver les signaux.

Une fréquence de scrutation garantie. Dans le cas où le serveur a sélectionné la scrutation comme méthode de détection, l'intégration du serveur à la bibliothèque de processus légers permet d'assurer une fréquence de scrutation constante. En effet, ce n'est plus un processus léger qui doit prendre la main régulièrement pour faire la scrutation. Les *callbacks* peuvent désormais être appelés régulièrement, par exemple à chaque changement de contexte, indépendamment du nombre de processus légers en exécution et indépendamment de leurs priorités.

Une scrutation plus efficace. Lorsqu'un processus léger demande au serveur un événement qu'il faut scruter, l'ordonnanceur place ce processus dans l'état bloqué jusqu'à l'arrivée de l'événement. La charge de l'ordonnanceur est alors réduite puisqu'on évite les changements de contexte nécessaires à la scrutation classique : l'exécution régulière des *callbacks* de scrutation par la bibliothèque ne nécessite pas l'ordonnancement du processus léger en attente.



(a) Enregistrement

(b) Demande d'agrégation

(c) Scrutation

(d) Événement

4.3(a) L'application enregistre les paramètres qui seront utilisés pour MPI dans l'ordonnanceur : les fonctions de rappel à utiliser (pour scruter, agréger les requêtes, etc.), et la fréquence de scrutation (le nombre de quanta de temps entre chaque scrutation).

4.3(b) Deux processus légers utilisateur B et C ont soumis une requête de détection d'événements à l'ordonnanceur. Ils sont mis dans l'état endormi et enlevés de la liste des processus légers prêts. L'ordonnanceur utilise la fonction de rappel enregistrée *group* pour agréger les requêtes.

4.3(c) L'ordonnanceur scrute le réseau tous les n changements de contexte en utilisant la fonction de rappel *poll*. Comme les deux requêtes sont agrégées, un seul appel à MPI est nécessaire.

4.3(d) L'événement attendu est survenu. L'ordonnanceur réveille le processus léger correspondant (ici le processus léger C). Il utilise la fonction de rappel *group* pour agréger les requêtes restantes (ici, une seule).

FIG. 4.3 – Un scénario MPI avec scrutation.

De plus, la centralisation des requêtes permet d'en factoriser certaines, ce qui est particulièrement intéressant lorsque la scrutation se fait par un appel système non bloquant. Ainsi, lorsque plusieurs processus légers attendent des événements sur des descripteurs de fichiers différents, la factorisation permet d'interroger l'état de l'ensemble de ces descripteurs de fichier en un seul appel système (un `select()` multiple) au lieu d'un appel système non bloquant par processus léger en attente.

4.3.3 Présentation de l'interface du serveur d'événements

L'un des objectifs du serveur d'événements est de rendre l'application indépendante des méthodes de détection employées. Pour ce faire, il est doté de deux interfaces : celle qui permet aux applications d'interroger le serveur et celle qui permet d'indiquer au serveur les méthodes de détection disponibles. Généralement, ces deux interfaces seront utilisées non pas directement par le code applicatif, mais plutôt par une bibliothèque de communication. À son initialisation, cette bibliothèque fournira au serveur les méthodes de détection, puis elle redirigera vers le serveur les requêtes des processus légers.

4.3.3.1 Interface d'interrogation du serveur d'événements

Cette interface doit être neutre (*i.e.* ne pas imposer un paradigme particulier) et permettre d'implémenter facilement les interfaces classiques des bibliothèques de communication. Ces interfaces peuvent être regroupées en deux grandes familles :

les interfaces synchrones : le flot d'exécution est suspendu en attente de l'événement. C'est, par exemple, le cas de la plupart des opérations sur les descripteurs de fichier dans leur état par défaut (appels systèmes `read()`, `write()`, `connect()`, etc.). C'est également le cas pour les primitives MPI synchrones, c'est-à-dire `MPI_send()`, `MPI_recv()`, etc. ;

les interfaces asynchrones : le flot d'exécution n'est jamais suspendu à la demande d'événement ; en cas d'échec de la requête, le programme devra, selon le cas, soit soumettre à nouveau sa demande, soit tester l'arrivée de l'événement lorsque la demande est mémorisée par le système. C'est ainsi, par exemple, le cas des opérations sur les descripteurs de fichier dans leur état *non bloquant* ; ces opérations, plutôt que de bloquer le flux d'exécution, retournent une erreur et la variable `errno` est positionnée à `EAGAIN`. C'est aussi le cas avec MPI, où l'on peut soumettre des envois ou des réceptions avec `MPI_isend()` ou `MPI_irecv()` qui retournent immédiatement, sans attendre la terminaison de la communication ; cette terminaison peut alors être testée avec `MPI_test()` ou attendue avec `MPI_wait()`.

Dans un programme séquentiel, le choix de l'une ou l'autre de ces familles d'interfaces influe profondément sur l'organisation du code applicatif. Cela est moins vrai dans le cas de programmes multithreadés. Il est effectivement facile d'écrire une interface synchrone à partir d'une interface asynchrone : il suffit d'attendre en continu l'événement. Dans le cas d'environnements multithreadés, la réciproque est vraie aussi : il est possible de créer un processus léger supplémentaire qui utilisera l'interface synchrone sans bloquer le processus léger exprimant sa requête.

Nous avons préféré doter le serveur d'événements d'une interface synchrone : si le programmeur préfère l'asynchronisme, il utilisera les processus légers avec tous les mécanismes

```
/****************************************************************
 * Fonctions à l'usage des threads applicatifs
 */

/* Attribut pouvant être attaché aux événements */
enum {
        /* Désactive la requête lorsque survient l'occurrence
         * suivante de l'événement */
        MARCEL_EV_ATTR_ONE_SHOT=1,
        /* Ne réveille pas les threads en attente d'événements du
         * serveur (ie marcel_ev_server_wait())*/
        MARCEL_EV_ATTR_NO_WAKE_SERVER=2,
};

/* Un raccourci pratique des fonctions suivantes, utile si l'on ne
 * soumet la requête qu'une seule fois. Les opérations suivantes sont
 * effectuées : initialisation, soumission et attente d'un
 * événement avec ONE_SHOT positionné */
int marcel_ev_wait(marcel_ev_server_t server, marcel_ev_req_t req,
                   marcel_ev_wait_t wait, marcel_time_t timeout);

/* Initialisation d'un événement
 * (à appeler en premier si l'on utilise autre chose que marcel_ev_wait) */
int marcel_ev_req_init(marcel_ev_req_t req);

/* Ajout d'un attribut spécifique à une requête */
int marcel_ev_req_attr_set(marcel_ev_req_t req, int attr);

/* Soumission d'une requête (le serveur PEUT commencer à scruter si cela
 * lui convient) */
int marcel_ev_req_submit(marcel_ev_server_t server, marcel_ev_req_t req);

/* Abandon d'une requête et retour des threads en attente sur cette
 * requête (avec le code de retour fourni) */
int marcel_ev_req_cancel(marcel_ev_req_t req, int ret_code);

/* Attente bloquante d'un événement sur une requête déjà enregistrée */
int marcel_ev_req_wait(marcel_ev_req_t req, marcel_ev_wait_t wait,
                       marcel_time_t timeout);

/* Attente bloquante d'un événement sur une quelconque requête du serveur */
int marcel_ev_server_wait(marcel_ev_server_t server, marcel_time_t timeout);

/* Renvoie une requête survenue n'ayant pas l'attribut NO_WAKE_SERVER
 * (utile au retour de server_wait())
 * À l'abandon d'une requête (wait, req_cancel ou ONE_SHOT) la requête
 * est également retirée de cette file (donc n'est plus consultable) */
marcel_ev_req_t marcel_ev_get_success_req(marcel_ev_server_t server);
```

FIG. 4.4 – Prototype des pricipales fonctions de l'interface d'utilisation du serveur d'événements

de synchronisation associés (verrous, conditions, etc.). De plus, une interface synchrone permet de suspendre un processus léger lorsqu'il attend une requête et donc d'alléger un peu la charge de l'ordonnanceur qui aura moins d'entités à gérer. La demande d'un événement se fait en plusieurs étapes :

1. initialisation d'une structure décrivant la requête ;
2. soumission de la requête au serveur d'événements en la rattachant au groupe d'événements dont elle dépend (afin de désigner l'ensemble des *callbacks* qui seront utilisés pour la gérer) ;
3. attente d'un événement (avec suspension du processus léger en attente) ;
4. annulation éventuelle d'une requête.

Distinguer la soumission de la requête et l'attente proprement dite de l'événement permet quelques optimisations. Tout d'abord, il devient possible d'anticiper des requêtes comme l'illustre le scénario suivant : à l'ouverture d'une connexion réseau, le programme enregistre une requête afin d'initier la scrutation. À la demande effective de l'événement (un processus léger veut réellement lire des données), il est possible que le résultat puisse être disponible immédiatement. Ensuite, cette distinction permet d'éviter la répétition de l'enregistrement d'une requête lorsque l'on désire plusieurs occurrences d'un événement. Par exemple, lorsque l'on veut lire successivement plusieurs données dans une connexion réseau, la requête ne sera enregistrée qu'à l'ouverture de la connexion et non à chaque demande d'événement. Enfin, cette séparation autorise une factorisation des requêtes (comme la primitive `select` pour les opérations sur les descripteurs de fichiers), permettant d'attendre un événement quelconque parmi l'ensemble des requêtes enregistrées.

Le prototype de ces fonctions est donné dans la Figure 4.4. L'annexe A contient la totalité de l'interface de MARCEL concernant le serveur d'événements.

4.3.3.2 L'enregistrement et les *callbacks*

Afin de pouvoir répondre aux requêtes qui lui sont soumises, le serveur d'événements doit être informé de toutes les fonctions de détection disponibles ; en présence de plusieurs méthodes de détection d'un même événement, MARCEL choisira la fonction la plus adaptée au système sous-jacent. À cette fin, la bibliothèque enregistre les informations propres à la détection des événements qu'elle gère dans une structure particulière (`struct marcel_ev_server`) ; ses requêtes seront alors rattachées à cette structure lors de leur soumission. Ces informations sont enregistrées avec deux fonctions :

`marcel_ev_server_add_callback()` qui permet d'enregistrer un *callback* supplémentaire (les différents types de *callbacks* pouvant être enregistrés sont décrits par la suite) ;

`marcel_ev_server_set_poll_settings()` qui permet de choisir des paramètres propres à la scrutation active.

Lorsque la bibliothèque inscrit ses *callbacks*, MARCEL met à sa disposition un ensemble de macros et de fonctions[17] lui permettant de connaître l'ensemble des requêtes soumises, de signaler une requête prête, de réveiller un processus léger en attente sur une requête, etc.

[17] le fichier en-tête est présenté en annexe A.

Appels systèmes bloquants. Comme nous l'avons vu dans la partie 4.3.1.1, les appels systèmes bloquants ne sont pas toujours opérationnels avec des bibliothèques de processus légers utilisateur. Aussi faut-il envisager leur substitution. Dans ce cas, il est quand même parfois possible d'utiliser un appel système bloquant unique (à la `select()`) pour l'ensemble des requêtes soumises. Pour cela, il faut pouvoir :

1. rassembler l'ensemble des requêtes en un seul appel ;
2. rajouter une requête à celles déjà en attente ;
3. abandonner l'attente en cours.

Cette solution permet d'envisager l'utilisation d'un processus léger de niveau noyau spécifique pour l'attente de ces événements. Ce processus léger noyau serait alors bloqué sur l'appel système, sauf en cas d'événement, et ne perturberait pas le reste de la bibliothèque de processus légers.

Pour les appels systèmes bloquants, cinq types de *callbacks* peuvent ainsi être renseignés :

MARCEL_EV_FUNCTYPE_BLOCK_WAITONE doit attendre un événement avec un appel système bloquant ;

MARCEL_EV_FUNCTYPE_UNBLOCK_WAITONE doit interrompre l'appel système précédent ;

MARCEL_EV_FUNCTYPE_BLOCK_WAITANY doit attendre avec un seul appel système bloquant que l'une quelconque des requêtes en attente soit satisfaite ;

MARCEL_EV_FUNCTYPE_UNBLOCK_WAITANY doit interrompre l'appel système précédent ;

MARCEL_EV_FUNCTYPE_BLOCK_GROUP est appelé chaque fois qu'une requête est ajoutée ou supprimée dans la liste de celles que doit gérer `MARCEL_EV_FUNCTYPE_BLOCK_WAITANY`.

Scrutation active. Placer directement dans l'ordonnanceur le support de scrutation active des événements permet d'assurer une fréquence de scrutation quasi constante, indépendante du nombre de processus légers en exécution sur le système. Les paramètres de cette scrutation sont définis par la fonction `marcel_ev_server_set_poll_settings()` qui permet de choisir à quel(s) instant(s) la scrutation doit être effectuée : tous les n quanta de temps, au changement de contexte, à chaque appel de fonction de la bibliothèque de processus légers et/ou pendant l'exécution de `idle()` (*i.e.* lorsqu'aucun processus léger n'est prêt). La scrutation peut aussi être déclenchée manuellement par l'application (fonction `marcel_ev_poll_force()`).

Trois *callbacks* sont associés à la scrutation :

MARCEL_EV_FUNCTYPE_POLL_POLLONE doit scruter l'état d'une requête particulière ;

MARCEL_EV_FUNCTYPE_POLL_POLLANY doit scruter l'ensemble des requêtes en attente ;

MARCEL_EV_FUNCTYPE_POLL_GROUP est appelé chaque fois qu'une requête est ajoutée ou supprimée.

Ce dernier *callback* permet de factoriser les requêtes en attente afin que le *callback* `POLL_POLLANY` soit plus efficace. Par exemple, pour des descripteurs de fichiers, le *callback* `POLL_GROUP` construira un ensemble de descripteurs de fichier regroupant tous ceux des requêtes en attente et le *callback* `POLL_POLLANY` utilisera cet ensemble pour ne faire qu'un seul appel système non bloquant à `select()`.

FIG. 4.5 – Exemple d'utilisation de la scrutation du serveur avec un réseau MPI

```
/* Structure contenant les variables du serveur d'événements MPI */
typedef struct MPI_server {
  /* Le type opaque de tout ensemble d'événements géré par Marcel */
  struct marcel_ev_server mserver;

  /* Données propres à cet ensemble d'événements,
      dans ce cas, il s'agit du nombre et du tableau des requêtes enregistrées.
      Ces variables sont utilisées par les callbacks (cf. Figure ~4.6(c)) */
  int count;
  MPI_Request requests[MAX_MPI_REQUEST];
} *MPI_server_t;

/* Variable globale définissant notre serveur */
struct MPI_server MPI_server = {
  .server=MARCEL_EV_SERVER_INIT(MPI_server.mserver, "MPI I/O"),
  .count=0,
};

void init(void) {
  /* Fonctions de rappels qui seront appelées par Marcel */
  marcel_ev_server_add_callback(&MPI_server.mserver,
                               MARCEL_EV_FUNCTYPE_POLL_GROUP, &MPI_group);
  marcel_ev_server_add_callback(&MPI_server.mserver,
                               MARCEL_EV_FUNCTYPE_POLL_POLLANY, &MPI_poll);

  /* Points de scrutation et fréquence */
  marcel_ev_server_set_poll_settings(&MPI_server.mserver,
      MARCEL_POLL_AT_TIMER_SIG|MARCEL_POLL_AT_YIELD|MARCEL_POLL_AT_IDLE, 1);

  /* Démarrage du serveur */
  marcel_ev_server_start(&MPI_server.mserver);
}
```

(a) Enregistrement des *callbacks* de scrutation

```
/* Structure contenant les variables relatives à une requête MPI */
typedef struct MPI_req {
  /* Le type opaque de toute requête gérée par notre bibliothèque */
  struct marcel_ev_req mreq;
  /* Les données propres aux requêtes MPI */
  MPI_Request request;
} *MPI_req_t;

int foo(...) {
  struct MPI_req MPI_req;
  struct marcel_ev_wait mwait;
  ...
  /* On appelle une fonction MPI asynchrone pour éviter de bloquer... */
  MPI_Irecv(buf, size, ..., &MPI_ev.request);
  /* ... et on attend sa terminaison avec notre mécanisme.
              Ainsi les autres processus légers peuvent s'exécuter. */
  marcel_ev_wait(&MPI_ev_server.mserver, &MPI_req.mreq, &mwait, 0);
  ...
}
```

(b) Utilisation du service

```
/* Factorisation des requêtes */
int MPI_group(marcel_ev_server_t server, marcel_ev_op_t op,
              marcel_ev_req_t req, int nb_ev, int option) {
  MPI_server_t MPI_server=struct_up(server, struct MPI_server, mserver);
  MPI_req_t MPI_req;
  MPI_server->count=0;
  /* Itérateur sur les requêtes enregistrées */
  FOREACH_REQ_POLL(MPI_req, server, mreq) {
    MPI_server->requests[MPI_server->count++] = MPI_req->request;
  }
  return 0;
}

/* Fonction de scrutation appelée une fois pour toutes les requêtes */
int MPI_poll(marcel_ev_server_t server, marcel_ev_op_t op,
             marcel_ev_req_t req, int nb_ev, int option) {
  MPI_server_t MPI_server=struct_up(server, struct MPI_server, mserver);
  int index, flag;

  MPI_Testany(MPI_server->count, MPI_server->requests, &index, &flag, ...);
  if (flag) {
    MPI_req_t MPI_req;
    FOREACH_REQ_POLL(MPI_req, server, mreq) {
      if (MPI_req->request == MPI_server->requests[index]) {
        /* Signalisation d'une requête satisfaite */
        MARCEL_EV_REQ_SUCCESS(&MPI_req->mreq);
        break;
      }
    }
  }
  return 0;
}
```

(c) Implémentation des *callbacks* de scrutation

Cette figure illustre la manière d'utiliser notre serveur d'événements. Nous considérons ici que nous utilisons MPI *pour communiquer. Nous ne voulons pas utiliser directement des primitives bloquantes pour ne pas risquer de bloquer l'ensemble des processus légers. Nous demandons donc à* MARCEL *de nous signaler la fin de nos communications.*

4.6(a) *Il s'agit d'enregistrer les* callbacks *qui pourront être utilisés par* MARCEL *par la suite. Afin de ne pas surcharger cet exemple, nous ne fournissons à* MARCEL *que les* callbacks *permettant de faire de la scrutation, et non ceux lui permettant de faire éventuellement des appels bloquants.*

4.6(b) *On utilise* `marcel_ev_wait()` *pour attendre l'événement. Cette fonction prend quatre arguments :*
- *le serveur* MARCEL *pour savoir quels* callbacks *il faudra utiliser ;*
- *la requête à enregistrer avant d'attendre et à supprimer une fois l'événement survenu. Ces opérations sont effectuées automatiquement par* `marcel_ev_wait()`*, mais il est possible de les réaliser séparément (voir le prototype des autres fonctions sur la Figure 4.4 page 71) ;*
- *une structure opaque à l'application (*`struct marcel_ev_wait`*) dont* MARCEL *se sert pour gérer l'attente.*
- *un timeout à 0 ici car non utilisé.*

4.6(c) callbacks *utilisés par* MARCEL*. Le premier est utilisé à chaque fois qu'une requête est soumise ou retirée. Le second est utilisé chaque fois qu'une scrutation est effectuée.*

On remarquera que pour le serveur et les requêtes, nous avons défini une structure englobant la structure opaque de MARCEL *et des variables propres à* MPI*. La macro* `struct_up()` *(utilisée par exemple dans les* callbacks*) permet de retrouver l'adresse de la structure englobante connaissant celle de la structure interne (à l'aide du type et du nom du champ correspondant).*

Signaux. L'utilisation de signaux pour détecter des événements ne nécessite pas d'interface supplémentaire. Il suffit d'utiliser les mécanismes de scrutation active décrits précédemment en désactivant la scrutation périodique, tout en forçant cette scrutation depuis le traitant du signal.

4.3.4 Bilan

Auparavant, les applications devaient intégrer directement dans leur code des mécanismes de détection d'événements. Dans le cas d'applications multithreadées, chaque processus léger détectait lui-même ses événements. Cela conduisait à une redondance de code, mais surtout à une structuration du code autour du mécanisme de détection choisi et à un arbitrage difficile à gérer entre les différents processus légers en compétition. Un serveur de détection d'événements permet d'éviter la duplication du code de détection ; mais il permet surtout d'abstraire ce code de l'application. Cette dernière n'est plus dépendante des techniques qui seront effectivement employées à l'exécution du programme pour récolter les événements. De plus, la centralisation des requêtes au sein d'un unique composant permet d'en factoriser certaines, diminuant ainsi le coût global de la détection des événements. En outre, les bibliothèques de processus légers de niveau utilisateur ou mixte sont souvent contraintes d'utiliser des mécanismes de scrutation. L'intégration d'un serveur de détection à une bibliothèque de processus légers permet alors d'assurer une fréquence de scrutation constante, indépendante du nombre de processus légers ordonnancés et, par conséquent, permet de garantir à l'application une borne de réactivité (égale au quantum de temps de la bibliothèque).

Notons que notre approche permet également d'utiliser des appels systèmes bloquants pour la détection d'événements, sous réserve que la bibliothèque de processus légers les gère correctement. Malheureusement, rares sont bibliothèques de processus légers de niveau utilisateur prenant en compte les appels systèmes bloquants. Nous allons aborder ce problème dans la section suivante.

4.4 Prolongation dans le système : les activations

La borne de réactivité d'une application utilisant la scrutation est au mieux égale au quantum utilisé par l'ordonnanceur. Pour obtenir une meilleure réactivité, on peut augmenter la fréquence de scrutation en truffant (manuellement ou automatiquement) le code de demande de scrutations, au risque de rendre cette scrutation très coûteuse. Une autre solution est de permettre l'emploi des appels systèmes bloquants au sein de processus légers de niveau utilisateur. Pour cela, il faut mettre en place un nouveau mécanisme : les *Scheduler Activations*. Ce mécanisme a été proposé par Anderson *et al.* en 1991[ABLL91]. Après avoir exposé le concept original, nous présenterons notre adaptation de ce concept au calcul hautes performances.

4.4.1 Concept original

Le but premier des *Scheduler Activations* est de permettre l'emploi d'appels systèmes bloquants dans les bibliothèques de processus légers de niveau utilisateur. L'idée principale est de faire coopérer l'ordonnanceur du système (ordonnanceur des LWP) et celui de l'application (ordonnanceur des processus légers de niveau utilisateur). Chaque fois que le noyau exécute une action d'ordonnancement affectant un quelconque processus léger de l'application, alors l'application en est informée et a l'opportunité de prendre des décisions pour (ré)ordonnancer les processus légers utilisateur sous son contrôle.

Cette coopération bidirectionelle entre les ordonnanceurs s'effectue par des appels systèmes dans le sens espace utilisateur vers noyau et par des *upcalls* dans le sens noyau vers application. Le noyau met à la disposition de l'application un certain nombre de LWP que nous appellerons *activations*. Le noyau se sert alors des *upcalls* (similaires aux traitants de signaux mais avec des paramètres différents) pour permettre à l'application de réagir lorsque ses activations sont ordonnancées sur un processeur ou, à l'inverse, lorsqu'elles perdent l'usage du processeur sur lequel elles s'exécutaient.

4.4.1.1 Modèle original

Le modèle des *Scheduler Activations* proposé par Anderson est fondé sur deux principes : (1) le système d'exploitation a le contrôle de l'allocation des processeurs physiques aux différents processus et (2) l'application connaît, à chaque instant, les processeurs qui lui sont alloués. Le système d'exploitation a ainsi la possibilité de changer le nombre de processeurs assignés à une application au cours de l'exécution de celle-ci. De son côté, l'application a un contrôle total sur les processus utilisateur s'exécutant sur les processeurs qui lui sont alloués. Elle peut alors rendre un processeur au système d'exploitation ou, au contraire, en obtenir de nouveaux selon ses besoins et les disponibilités.

Le noyau peut faire remonter à l'application quatre types d'informations :

`New(`*`numéro processeur`*`) :` une nouvelle activation (processeur virtuel) est disponible. L'application peut ordonnancer un processus utilisateur dessus ;

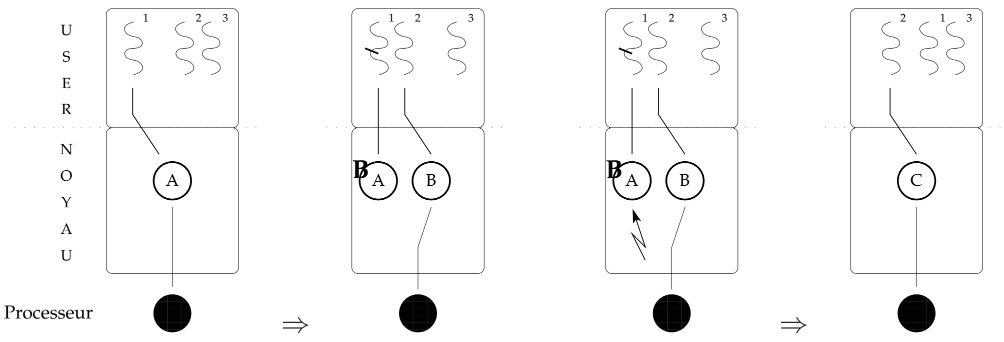
`Preemption(`*`numéro activation préemptée et état du processus léger`*`) :` une activation a été préemptée (le processeur a été donné à une autre application, etc.) Le processus léger interrompu est remis dans la liste des processus légers prêts (l'état fourni permettra de le redémarrer) ;

`Block(`*`numéro activation`*`) :` l'activation est bloquée (appel système bloquant, faute de page, etc.) et n'utilise plus son processeur ;

`Unblock(`*`numéro activation et état du processus léger`*`) :` le processus léger bloqué peut désormais être remis dans la liste des processus légers prêts.

Ces informations peuvent être regroupées au sein d'un même *upcall*. Ainsi, comme l'illustre la Figure 4.6, une nouvelle activation est créée lors d'un appel système bloquant (au blocage même de l'activation) ; l'*upcall* correspondant contient les informations `New()` et `Block()`. De même, au moment du réveil, l'*upcall* contiendra les trois informations suivantes : `Unblock()` accompagné de l'état du processus léger de l'activation réveillée, `New()` signifiant que cette nouvelle activation est disponible et `Preemption()` accompagné de l'état du processus léger de l'activation qui a été préemptée pour attribuer son processeur à l'activation exécutant l'*upcall*.

Dans ce modèle, une activation est créée lorsque le noyau a besoin de transmettre des informations à l'application via un *upcall*. Un processus léger s'exécute alors en son sein jusqu'à ce que l'activation se bloque ou soit préemptée. L'activation alors suspendue ne reprendra jamais la main, mais le processus léger qui s'y exécutait pourra continuer son exécution sur une autre activation dès que son état aura été transmis à l'application (via les *upcalls* `Unblock()` ou `Preempt()`).



(d) Exécution normale — (e) Appel système bloquant — (f) Réveil — (g) Exécution normale

4.6(d) Le processus léger utilisateur 1 s'exécute au sein de l'activation A liée à un processeur.

4.6(e) Le processus léger utilisateur 1 exécute un appel système qui bloque l'activation A. Une autre activation B est créée et démarrée. Elle exécute alors un autre processus léger utilisateur 2.

4.6(f) Une interruption matérielle survient qui réveille l'activation bloquée A. Elle ne peut redémarrer car elle n'a pas de processeur disponible.

4.6(g) Une nouvelle activation C est créée pour informer l'application du réveil. L'activation C obtient le processeur (l'activation B est donc préemptée) et elle peut alors choisir quel processus léger utilisateur exécuter. Les activations A et B peuvent être détruites une fois que l'état des processus légers utilisateur qui s'exécutaient en leur sein a été envoyé à l'application grâce à l'*upcall*.

FIG. 4.6 – Appel système bloquant avec les *Scheduler Activations*

4.4.1.2 Discussion

Le modèle original des *Scheduler Activations* a été implémenté au sein de TOPAZ, le système d'exploitation natif des stations de travail multiprocesseurs DEC SRC FIREFLY. Le code de cette implémentation n'est malheureusement pas disponible car ce logiciel est sous licence propriétaire et interdit de diffusion. En cherchant à utiliser ce modèle pour nos travaux, nous avons identifié plusieurs caractéristiques limitant la possibilité d'obtenir une implémentation efficace pour le calcul hautes performances.

Nombreux transferts de données. Le modèle original requiert de nombreuses et (relativement) volumineuses communications entre le noyau et l'application. Nombreuses, parce qu'à chaque préemption le noyau en informe l'application. Or, les préemptions sont fréquentes dans les systèmes d'exploitation à temps partagé comme LINUX. Même si la machine est réservée à l'exécution d'une seule application (ce qui n'est pas absurde dans le cadre du calcul parallèle intensif), les démons du système d'exploitation ou du noyau reprennent périodiquement la main.

Les communications ont un volume important puisque les informations apportées par `Preemption()` et `Unblock()` contiennent l'état complet d'un processus léger[18]. De plus,

[18]Suivant le processeur, l'état complet d'un processus léger prend plus ou moins de place. Un processeur de

`Preemption()` est une information fréquemment communiquée car la probabilité que le système ait un processeur disponible pour exécuter un *upcall* donné est faible, ce qui est souhaitable ! Ainsi, simplement pour pouvoir s'exécuter, chaque *upcall* contiendra en moyenne au moins une[19] information `Preemption()`.

Réentrance des *upcalls*. Une difficulté importante du modèle d'Anderson concerne la réentrance des *upcalls*. En effet, une préemption peut survenir à tout instant en réponse à une modification du nombre de processeurs disponibles pour l'application. Par conséquent, le code applicatif, en particulier celui des *upcalls*, doit être réentrant ou protégé. Cela pose deux difficultés.

La première concerne les sections critiques du code. Lorsqu'une activation est préemptée alors que son processus léger est dans une section critique, il est crucial que ce processus léger soit réordonnancé au plus tôt. On risque sinon d'obtenir le dead-lock suivant : si le verrou protégeant la liste des processus légers prêts est détenu par le processus léger préempté, tout autre processus léger voulant passer la main à ce dernier attendra indéfiniment le verrou. Anderson *et al* résolvent cette difficulté d'astucieuse façon : les processus légers interrompus en section critique sont réordonnancés immédiatement mais, au relâchement du verrou, le contrôle revient au traitement classique de l'*upcall*. Pour obtenir ce comportement à faible coût, le code des sections critiques est dupliqué automatiquement à la compilation, la copie redirigeant le flot d'exécution vers le traitement de l'*upcall*. Le basculement du code original à celui de la copie est effectué dans l'*upcall* lorsque l'on découvre que le processus léger interrompu était en section critique. Toutefois, ce mécanisme devrait être complexifié si on autorise un processus léger à prendre plusieurs verrous simultanément, situation banale lorsqu'on désire réduire les contentions des sections critiques sur machines SMP.

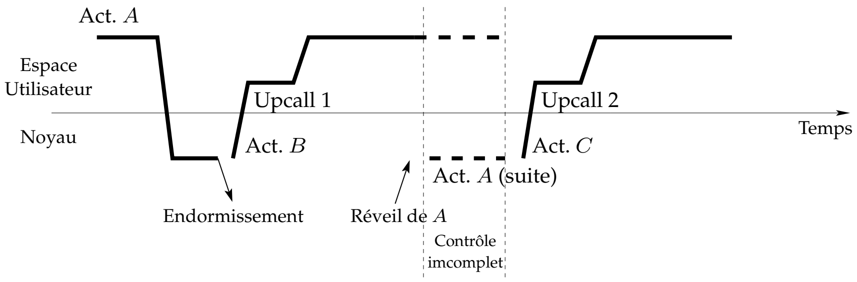
La seconde difficulté concerne la pile utilisateur nécessitée par l'exécution d'un *upcall*. Une solution possible est que l'*upcall* utilise, moyennant quelques précautions, la pile du processus léger utilisateur préempté. Cependant, lorsqu'il s'agit d'une nouvelle activation, par exemple après un appel système bloquant, alors il n'y a aucune pile disponible. Les articles d'Anderson taisent ce problème ainsi que sa solution retenue dans leur prototype.

Contrôle applicatif non complet. Ce modèle des activations donne un très grand contrôle à l'application sur l'ordonnancement de ses processus légers puisqu'elle est informée et a la possibilité de prendre des décisions chaque fois qu'un processeur physique lui est alloué ou retiré. Cependant, il arrive qu'un processus léger s'exécute alors que l'application le croit bloqué. Cette période d'exécution ignorée par l'application survient entre le réveil du processus léger bloqué dans le noyau et sa signalisation à la sortie du code noyau. En effet, comme l'état complet du processus léger est transmis lors de la signalisation du réveil, il est impossible pour des raisons de sécurité de donner l'état du processus léger en mode noyau [20]. Il doit donc avoir terminé la partie du code en mode noyau avant de pouvoir être signalé à l'application. Cela signifie que l'application n'est pas complètement maîtresse

la famille `x86` a besoin d'environ 150 octets pour sauver tous ses registres. Un processeur `ia64` aura besoin de plus d'un kilo octet en raison de son grand nombre de registres.

[19] Pour les préemptions et les réveils de processus léger, l'*upcall* contiendra un autre état complet d'un processus léger.

[20] Pendant un appel système, en mode noyau, l'état du processeur contient des informations privilégiées, comme des droits d'accès ; l'application ne doit pouvoir ni les lire ni les modifier.



Le scénario est le même que celui de la Figure 4.6. Deux *upcalls* sont générés :

***Upcall* 1** {`Block(`A`)` + `New(`B`)`} **:** un appel système bloquant survient, une activation (B) est crée ;

***Upcall* 2** {`Unblock(état(`A`))` + `Preempt(état(`B`))` + `New(`C`)`} **:** l'activation B est préemptée pour permettre, à travers l'activation C, de signaler la fin de l'appel système bloquant de l'activation A dont l'application récupère ici l'état complet.

Entre le moment où l'activation A a été réveillée et celui où elle est prête à repasser en mode utilisateur, le processeur est occupé par l'activation B ou A sans contrôle de la part de l'application.

FIG. 4.7 – Occupation d'un processeur lors d'un appel système bloquant

de l'ordonnancement de ses processus légers, ce qui peut être gênant si l'on désire contrôler très précisément l'ordonnancement afin d'assurer, par exemple, une réactivité maximale pour certains processus légers.

4.4.2 Amélioration du modèle d'Anderson

Nous voulons offrir une bibliothèque permettant à une application de calcul hautes performances de maîtriser, si elle le souhaite, aussi finement que possible son ordonnancement. On doit pouvoir être assuré que les contraintes applicatives d'ordonnancement sont en permanence respectées. Ce point est particulièrement important si l'on désire, par exemple, s'assurer qu'un processus léger en attente d'un événement avec un appel système bloquant est le plus réactif possible en lui redonnant la main dès que l'événement est détecté par le système d'exploitation. Ou bien, au contraire, cet événement n'est pas prioritaire et la fin de l'appel système ne doit pas interrompre un calcul exploitant le cache du processeur. Le contrôle incomplet de l'ordonnancement final dans le modèle d'Anderson est en contradiction avec ces objectifs. En outre, le coût en temps processeur des *upcalls* et la complexité de la gestion de la réentrance amoindrissent l'intérêt du modèle d'Anderson. C'est pour corriger ces défauts que nous avons revisité ce modèle et proposé une implémentation basée sur des principes légèrement différents.

Principes novateurs. Le modèle que nous proposons est très largement inspiré de celui d'Anderson : les notions d'*upcalls* et d'*activations* sont conservées. Notre modèle diffère en deux points : **l'état des activations n'est plus transmis au sein des *upcalls*** et **le système d'exploitation ne peut plus changer de sa propre initiative le nombre de processeurs alloués à une application**. Le second point fait que les événements de préemption ne sont plus à signaler et donc qu'une activation pourra être suspendue (*i.e.* ne plus être ordonnancée sur un processeur physique) sans que l'application en soit prévenue.

Comme nous allons le découvrir, ces deux différences ont de nombreuses et profondes implications sur les propriétés et l'implémentation du modèle des activations.

4.4.2.1 Une nouvelle interface

Voici les informations communiquées à l'application par le noyau :

`New(`*`numéro processeur`*`)` : une nouvelle activation (processeur virtuel) est disponible. L'application peut y ordonnancer un processus utilisateur. Cette information est jumelée à l'information `Block()` (sauf à la création des premières activations) ;

`Block(`*`numéro activation`*`)` : l'activation est bloquée (appel système bloquant, page fault, etc.) et n'utilise plus son processeur. Cette information est jumelée avec une information `New()` ou `Restarted`;

`Unblock(`*`numéro activation réveillée`*`)` : le processus léger bloqué pourra désormais être redémarré lorsque l'application le souhaitera ;

`Restarted:` le processus léger qui était bloqué vient de redémarrer en mode utilisateur. Cette information est transmise soit après un appel système `ActRestart()`, soit en conjugaison avec l'information `Block()` afin d'éviter la création d'une activation alors que d'autres sont prêtes ;

`Timer:` un quantum de temps s'est écoulé. Cet *upcall* n'est pas induit par le modèle, mais il permet d'implémenter plus facilement une bibliothèque de processus légers préemptible.

Par ailleurs, un appel système supplémentaire est utilisé :

`ActRestart(`*`numéro activation à redémarrer`*`)` : l'activation courante est supprimée du système et remplacée par l'activation réveillée (signalée auparavant par un *upcall* `Unblock`).

Un exemple de communication entre le noyau et l'application employant ces mécanismes lors d'un appel système bloquant est donné dans la Figure 4.8.

Gestion des piles. À l'initialisation, l'application déclare une pile destinée à l'exécution des *upcalls* pour chaque processeur. Un *upcall* avec l'information `New` utilise tout d'abord cette pile puis bascule sur celle du processus léger utilisateur qu'il choisit d'ordonnancer. Les autres *upcalls* utilisent la pile du processus léger en cours d'exécution sur le processeur où se déroule l'*upcall*, à la manière des signaux UNIX et à la fin de l'*upcall*, le processus léger reprend normalement son exécution. L'application a la responsabilité de libérer la pile utilisée lors d'une information `New` avant que ne survienne un événement, par exemple un appel système bloquant, conduisant à sa réutilisation.

4.4.2.2 Une efficacité accrue

Comme nous l'avons remarqué, sur les machines multiprocesseurs équipées d'un système à temps partagé, le modèle d'Anderson est pénalisé par les nombreuses notifications de préemption , en particulier par celles dues aux démons du système. L'abandon de la notification de la préemption diminue de manière significative le nombre de communications entre le noyau et l'application qui peut alors profiter pleinement du temps processeur qui lui est alloué.

Par ailleurs, sans la gestion de la préemption, le traitement des *upcalls* par la bibliothèque de processus légers utilisateur est grandement simplifié. En effet, dans le modèle original, la préemption pouvant survenir à tout instant, le code des *upcalls* doit être parfaitement réentrant. Cela s'avère complexe à gérer puisque ce code doit manipuler des verrous (pour les files de processus légers prêts, bloqués, etc.) et surtout obtenir une pile utilisateur libre à chaque *upcall*. L'utilisation lors des *upcalls* de la pile de l'activation préemptée (ou de la pile spécifique au processeur en cas d'information `New()`) résout efficacement ces besoins répétés de piles tout en permettant de simplifier l'implémentation réentrante des *upcalls*. La partie 5.2 explique plus en détail tous les mécanismes proposés par notre implémentation pour faciliter son utilisation par les bibliothèques de processus légers utilisateur.

4.4.2.3 Une maîtrise complète de l'ordonnancement

Contrairement au modèle original, l'état complet des processus légers n'accompagne plus les informations `Préemption()` et `Unblock()`. Comme nous allons le voir, cela permet de maîtriser complètement l'ordonnancement.

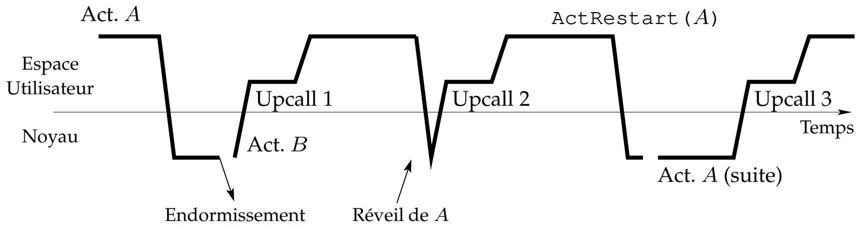
Dans le modèle original, l'information `Préemption()`, accompagnée de l'état d'un processus léger, est transmise dans deux situations. La première est l'indisponibilité de l'un des processeurs utilisés par l'application ; remarquons que cette information n'a plus de sens dans notre modèle puisque le système n'a pas le droit de modifier le nombre de processeurs dévolus à l'application. La seconde situation survient à la préemption d'une activation pour exécuter un *upcall*, comme, par exemple, lorsqu'une activation bloquée se réveille : il est nécessaire de préempter une autre activation en cours d'exécution afin d'informer, via un *upcall*, du réveil survenu (voir Figure 4.6(g)). Dans notre modèle, la préemption d'une activation se fait de façon implicite, en déroutant temporairement l'activation (et son processus léger) en exécution sur le processeur où s'exécute l'*upcall*. L'exécution du processus léger préempté reprendra normalement à la terminaison du traitement de l'*upcall*. Ainsi, la transmission via l'*upcall* de l'état de l'activation préemptée est inutile.

L'état du processus léger accompagnant l'information `Unblock()` est, quant à lui, remplacé par la notification du numéro de l'activation qui s'est réveillée et par un appel système (`ActRestart()`) permettant par la suite de redémarrer cette activation. Cela permet de contrôler en permanence l'ordonnancement des activations (*i.e.* des processus légers utilisateur qui s'y exécutent), y compris lorsqu'elles sont en mode noyau. La Figure 4.8 illustre ce comportement ; on compte le même nombre de changements de contexte dans les deux modèles (Figure 4.7). De plus, notre modèle ne donne pas la main immédiatement à l'activation réveillée. En fait, le modèle original ne garantit pas que l'activation réveillée va obtenir rapidement la main (cela dépend entièrement du système d'exploitation). Selon notre modèle, l'activation en exécution va immédiatement exécuter un *upcall* et l'application pourra éventuellement réordonnancer au plus tôt l'activation réveillée.

Ainsi, notre modèle autorise le contrôle complet de l'ordonnancement des processus légers de l'application, y compris lorsqu'ils s'exécutent en mode noyau.

4.4.3 Discussion

Nous tenons à évoquer ici deux aspects qui pourraient être perçus comme des points faibles de notre approche, à savoir l'absence de notification des préemptions et l'utilisation des ressources noyau due à la reprise différée des activations réveillées.



Voici le déroulement, selon notre modèle, d'un appel système bloquant (à comparer avec la Figure 4.7 page 80).

***Upcall* 1 {`Block(`A`)` $+$ `New(`B`)`} :** un appel système bloquant survient, une activation (B) est créée ;

***Upcall* 2 {`Unblock(`A`)`} :** l'application ne reçoit que l'identifiant de l'activation A (l'état complet de son processus léger, encore en mode noyau, n'est pas transmis). De plus, l'activation A n'est pas redémarrée par le système d'exploitation. L'activation B est utilisée pour faire cet *upcall* (pas de nouvelle activation crée ici) ;

`ActRestart(`A`)` : au moment opportun, l'application demande au système d'exploitation de redémarrer (en supprimant l'activation courante B) l'activation A qui va alors finir son appel système en mode noyau avant de revenir en mode utilisateur ;

***Upcall* 3 {`Restarted`} :** l'application est informée de la reprise de l'activation A en mode utilisateur. Fondamentalement, cet *upcall* n'est pas réellement nécessaire, mais il facilite l'implémentation en permettant la mise à jour de variables internes à la bibliothèque de processus légers.

Les *upcalls* générés ici sont légers (pas besoin de fournir l'état complet d'un processus léger) et l'application garde en permanence le contrôle de son ordonnancement.

FIG. 4.8 – Déroulement d'un appel système bloquant avec le nouveau modèle

Préemption et attente active. Sans notification de préemption, l'application ne peut pas déterminer si elle utilise ou non réellement un processeur. Cela a généralement peu de conséquences pour les applications, puisque celles-ci utilisent les processus légers comme abstraction pour le parallélisme et ne se préoccupent donc pas des processeurs physiques. Tel n'est pas le cas de la bibliothèque de processus légers puisqu'elle est responsable de l'ordonnancement des processus légers sur les processeurs physiques. Sur machine monoprocesseur, signaler la préemption est inutile : soit l'application a la main et peut s'exécuter, soit elle ne l'a pas et ne peut donc pas être prévenue. En revanche, sur une machine multiprocesseur, une bibliothèque de processus légers a besoin de se synchroniser à faible coût entre les divers flots d'exécution réellement parallèles. Comme il serait beaucoup trop coûteux de solliciter systématiquement l'aide du système d'exploitation, elle utilise généralement des mécanismes d'attente active (*spinlock*). Ces *spinlock* protègent habituellement de courtes parties de la bibliothèque devant s'exécuter de manière exclusive. Or, il peut arriver qu'un flot d'exécution cherche à entrer dans une telle section alors que le *spinlock* est possédé par un autre flot d'exécution. Si ce dernier est en cours d'exécution, il sort rapidement de la section critique et libère le *spinlock*. En revanche, si le flot est préempté dans la section critique, le flot gaspille en attente active la fin de son quantum. Le modèle original des *Scheduler Activations* permet de repérer cette préemption et d'ordonnancer en priorité le processus léger possédant le *spinlock*.

Nous avons cependant considéré que la complexité requise par la gestion de la préemption était trop importante en regard des gains espérés. Tout d'abord, notre cible est le calcul hautes performances où les machines sont dédiées aux applications de calcul. Aussi la préemption de manière durable d'une activation de l'application pour un autre processus est-elle considérée comme un événement rare. Qui plus est, plusieurs techniques permettent de réduire cet inconvénient. Après avoir tenté d'obtenir le *spinlock* pendant quelques cycles, le processus peut rendre la main à l'ordonnanceur système (fonction `sched_yield()`). On peut alors espérer que l'activation qui avait été préemptée au profit d'une autre application reprenne la main pour libérer par la suite le *spinlock*. Enfin, il est possible d'utiliser le mécanisme des FUTEX [Rus02b] introduit dans les noyaux LINUX récents. Ce mécanisme permet d'implémenter des verrous dont le code reste en espace utilisateur s'il n'y a pas de contention, mais utilise le système d'exploitation pour suspendre le flot d'exécution en cas de contention. En cas de besoin, nous pourrions spécialiser l'outil d'attente active de notre bibliothèque à l'aide de ces FUTEX.

Consommation des ressources système. Le fait de ne plus terminer les appels systèmes séance tenante (et donc ne plus libérer l'activation lors de l'*upcall* `Unblock()`) peut faire craindre une consommation excessive de ressources systèmes par notre modèle, d'autant plus que c'est à l'application d'appeler `ActRestart()` pour redémarrer l'activation.

Dans notre modèle, le blocage d'une activation n'entraîne la création d'une autre que si aucune activation n'est prête à être redémarrée. Cela signifie qu'une nouvelle activation n'est crée que si un processus est libre et que toutes les autres activations sont soit ordonnancées, soit bloquées dans le noyau, c'est-à-dire dans les mêmes cas que le modèle original. Notre modèle ne crée donc pas plus d'activations que le modèle original mais plutôt légèrement moins car la plupart des *upcalls* du modèle original entraînent la création d'une activation. Par contre, les activations y sont présentes plus longuement dans la machine puisqu'il faut parfois attendre longuement avant que l'application ne réordonnance une activation signalée `Unblock()`. Ceci nous a semblé raisonnable par rapport à la possibilité de contrôler parfaitement l'ordonnancement des activations, même en mode noyau.

4.4.4 Bilan

Grâce au modèle des *Scheduler Activations*, une bibliothèque de processus légers de niveau utilisateur peut exécuter des appels systèmes bloquants sans se paralyser. L'adaptation que nous avons faite de ce modèle nous permet un contrôle applicatif précis et complet sur l'ordonnancement des processus légers utilisateur, y compris lorsqu'ils s'exécutent en mode noyau (appels systèmes bloquants). Un processus léger utilisateur peut désormais utiliser un appel système bloquant pour attendre une interruption générée par une carte réseau ; le contrôle fin de l'ordonnancement permettra à l'application de réagir promptement à l'arrivée d'un message si elle le désire.

Le modèle des *Scheduler Activations* n'est pas encore très répandu dans la plupart des systèmes, sans doute en raison des difficultés à faire coopérer efficacement le noyau et la bibliothèque utilisateur. Un support spécifique est nécessaire de la part du système d'exploitation. Nous avons implémenté nos travaux dans le noyau LINUX sur architecture X86, ce qui fait que cette plate-forme est la seule à l'heure actuelle où MARCEL peut librement utiliser des appels systèmes bloquants.

4.5 Conclusion

Directement ou via des supports exécutifs, la plupart des applications de calcul hautes performances font usage de processus légers. Pour répondre à leurs besoins, nous avons décider de concevoir une bibliothèque portable, réactive et efficace sur des architectures matérielles variées, une bibliothèque de processus légers garantissant la *portabilité des performances*.

Cette garantie est obtenue par l'emploi des techniques de spécialisation (afin de bénéficier des meilleurs mécanismes de l'architecture ciblée) et de modularisation (afin d'éviter le surcoût dû à des fonctionnalités inutiles et d'assurer l'extensibilité de notre bibliothèque) ainsi que par le développement des services suivants :

Un service pour réagir aux événements d'entrée/sortie. Dans le domaine du calcul hautes performances, une bonne réactivité du support exécutif aux messages réseaux et, plus généralement, aux entrées/sorties constitue un atout certain. Cependant, la diversité des mécanismes de détection d'événements et leurs contraintes quant à leur utilisation en environnement multithreadé rendent difficile pour l'application l'écriture de mécanismes de détection efficaces en toutes circonstances. À cette fin, nous avons conçu un *serveur d'événements* qui, d'une part, rend l'application indépendante des moyens de détection réellement utilisés et d'autre part, sélectionne automatiquement la méthode de détection la plus adaptée au système ciblé. L'implémentation de ce serveur d'événements au sein de la bibliothèque de processus légers permet en outre de garantir la fréquence de scrutation lorsque cette technique est employée, garantissant également une borne de réactivité.

Des extensions système pour accroître le contrôle sur l'ordonnancement. Afin de permettre l'utilisation d'appels systèmes bloquants par des processus légers de niveau utilisateur, nous avons revisité et implémenté le modèle des *Scheduler Activations*. Ainsi, il devient possible d'utiliser les appels systèmes bloquants pour exploiter les interruptions matérielles générées, par exemple, par les cartes réseaux.

Un service pour observer. La complexité des ordonnancements produits par notre bibliothèque rend indispensable l'obtention de traces lorsque l'on désire comprendre finement le déroulement de l'application. L'absence d'outils adaptés nous a conduits à développer une solution originale. Ce service, bien qu'intégré lui aussi dans notre bibliothèque, est plus indépendant afin de permettre l'observation de l'ensemble de l'application, y compris le cœur de la bibliothèque. C'est pourquoi il est présenté en détail dans le chapitre 6 page 107.

Chapitre 5

Eléments d'implémentation

Sommaire

Après avoir présenté les fondements de notre bibliothèque de processus légers et les mécanismes de prise de traces, nous exposons dans ce chapitre les difficultés rencontrées et les solutions adoptées lors de l'implémentation de nos travaux.

Ainsi, nous nous focaliserons sur l'implémentation des mécanismes qui ont demandé quelques réflexions à savoir ceux concernant la synchronisation interne, le serveur d'événements, les activations et ceux de l'exclusion mutuelle nécessaire au recueil des traces. Nous présenterons ensuite plusieurs aspects du développement lié à la notion de généricité et de spécialisation de notre bibliothèque.

5.1 Synchronisation

```
mutex_lock(mutex) {
  while (is_mutex_hold(mutex)) {
    record_want_mutex(mutex, self);
    sleep();
  }
  set_mutex_hold(mutex);
}
```

```
mutex_unlock(mutex) {
  unset_mutex_hold(mutex);
  next=recorded_want_mutex(mutex);
  if (next!=NULL) {
    wake_up(next);
  }
}
```

Ceci n'est pas une implémentation correcte d'un `mutex` *!*

De nombreuses erreurs peuvent survenir si deux processus légers exécutent simultanément ou de manière entrelacée ces deux fonctions. Par exemple, imaginons qu'un processus léger A veuille le verrou détenu par le processus léger B. Si B libère le verrou entre le moment où A teste s'il est pris (`is_mutex_hold(mutex)`* renvoie vrai) et le moment où il s'enregistre dans la liste d'attente (*`record_want_mutex(mutex, self)`*), alors B ne détectera pas de processus en attente (A n'étant pas encore enregistré), donc ne réveillera aucun processus et A s'endormira alors pour l'éternité.*

FIG. 5.1 – Implémentation d'un verrou sans synchronisation interne

L'un des problèmes les plus ardus à propos de l'écriture de programmes parallèles concerne la synchronisation des flots d'exécution, et le développement d'une bibliothèque de processus légers n'y échappe évidemment pas. Il convient cependant de distinguer les mécanismes de synchronisation interne et externe. Les mécanismes de synchronisation externe sont les mécanismes classiques tels que les verrous, conditions, sémaphores, barrières, etc. Ces mécanismes permettent aux programmeurs de synchroniser les processus légers et reposent sur les mécanismes de synchronisation interne, de plus bas niveau. Prenons l'exemple des verrous : une certaine synchronisation est nécessaire afin de tester si le verrou est libre ou non et de passer la main à un autre processus léger le cas échéant. La Figure 5.1 montre un exemple d'implémentation de verrou non synchronisé et donc incorrect.

Dans un premier temps, nous allons présenter et justifier les mécanismes de synchronisation bas niveau que nous avons introduits. Nous examinerons ensuite le serveur d'événements d'entrées/sorties qui, pour être efficace, nécessite une synchronisation assez fine. Enfin, nous présenterons le mécanisme de synchronisation utilisé pour garantir une prise de trace cohérente entre processus légers.

5.1.1 Les mécanismes de synchronisation interne

Le but des primitives de synchronisation interne à une bibliothèque de processus légers est de fournir des moyens d'accéder de manière cohérente à des données partagées entre plusieurs flots d'exécution. Ces opérations sont cruciales car elles sont souvent invoquées et peuvent être la cause de forts ralentissements (goulots d'étranglement).

5.1.1.1 Discussion autour des contraintes du modèle

Notre bibliothèque ayant vocation à s'adapter efficacement à des architectures différentes, les mécanismes de synchronisation choisis devront pouvoir être implémentés *toujours* efficacement.

Comme évoqué auparavant (cf. section 3.2.3.1), une bibliothèque de processus légers de niveau purement utilisateur sans préemption, ce qui correspond au paradigme des coroutines, peut être synchronisée très facilement. Par exemple, le code de la Figure 5.1 est alors correct : les fonctions présentées ne peuvent plus être exécutées de manière simultanée ou entrelacée puisque tous les changements de contexte doivent être écrits explicitement dans le code. De manière plus générale, une synchronisation pour une bibliothèque de niveau utilisateur est généralement moins coûteuse que lorsqu'il faut prendre en compte d'autres flots d'exécution concurrents.

Le modèle de synchronisation interne doit donc respecter plusieurs contraintes :

être assez souple pour permettre de traiter les schémas de synchronisation ciblés ;

être indépendant de l'architecture pour pouvoir être optimisé efficacement sur tous les types de plate-forme ;

s'intégrer avec l'ordonnanceur efficacement mais sans dépendance.

Le dernier point, l'intégration avec l'ordonnanceur, mérite quelques éclaircissements. La synchronisation interne ne fait pas intervenir directement l'ordonnanceur, *i.e.* les processus légers ne sont pas suspendus par l'ordonnanceur en cas de contention sur une ressource. Par contre, les fonctions utilisant la synchronisation interne doivent être capables de faire endormir un processus léger ou d'en réveiller afin d'implémenter les primitives de synchronisation externe. Or, ces changements d'état doivent généralement être réalisés de manière synchrone avec d'autres informations, comme l'état du verrou (libre ou pris) pour reprendre l'exemple de la Figure 5.1. Un objectif de notre bibliothèque est d'êre très flexible, avec un ordonnanceur indépendant. C'est pourquoi il est crucial que l'interface de l'ordonnanceur permette d'endormir et réveiller des processus légers de manière synchrone avec d'autres opérations sans que ces autres opérations soient incluses dans l'ordonnanceur. Cela signifie que l'ordonnanceur ne peut pas avoir une interface avec uniquement les fonctions `sleep()` pour s'endormir en passant la main et `wake_up(thread)` pour réveiller un processus léger.

Enfin, le modèle de synchronisation doit permettre la gestion d'événements asynchrones. Certains événements, comme les demandes de changement de contexte ou encore les *upcalls* (lorsque les activations sont présentes), peuvent survenir à tout instant. Il faut alors être capable de les traiter immédiatement ou d' offrir des mécanismes pour retarder légèrement leur traitement.

5.1.1.2 Le modèle proposé

Nous présentons ici le modèle de synchronisation bas niveau que nous avons implémenté en nous inspirant de celui du noyau LINUX : Les LWP et les processus légers d'une bibliothèque mixte peuvent être conceptuellement assimilés aux processeurs physiques et au processus d'un système d'exploitation malgré certaines différences[1].

[1] Le noyau LINUX est assuré d'avoir toujours le même nombre de processeurs à sa disposition, processeurs qui sont en permanence en activité. Dans une bibliothèque mixte, le nombre de LWP est susceptible de varier plus facilement, et surtout ils peuvent perdre la main s'ils sont en surnombre par rapport au nombre de pro-

Sections critiques. Nous proposons tout d'abord de permettre la déclaration de sections critiques. Ces sections ne peuvent être exécutées que par un seul flot d'exécution simultanément. En revanche, contrairement à ce qui se passerait avec un verrou (*mutex*) classique, un processus léger dans une telle section critique ne peut ni passer la main (il doit quitter la section critique auparavant), ni perdre la main (il reste ordonnancé sur son LWP courant jusqu'à ce qu'il sorte de la section critique).

Interruptions logicielles. Nous proposons également des interruptions logicielles (appelées SOFTIRQ) qui peuvent être déclenchées à tout instant, y compris dans un traitant de signal asynchrone ou un *upcall*. Elles sont ensuite traitées, en exécutant une fonction spécifique à l'interruption générée, le plus tôt possible. En effet, le code peut interdire temporairement le traitement de ces interruptions *sur le* LWP *courant*. Rien n'interdit leur traitement sur un autre LWP. Comme dans le cas des sections critiques, un code interdisant le traitement de ces interruptions ne peut ni céder la main, ni donner la main.

Interface avec l'ordonnanceur. Les ordonnanceurs de notre bibliothèque doivent offrir trois primitives pour s'interfacer avec le code de synchronisation bas niveau :

`set_state(`*`état`*`)` qui permet de changer l'état du processus léger faisant l'appel. Cette fonction sera principalement utilisée pour passer de l'état `RUNNING` à `BLOCKED`. À noter que cette fonction ne fait que changer l'état logique du processus léger sans modifier l'ordonnancement courant ;

`schedule()` qui appelle l'ordonnanceur proprement dit. Il prendra les décisions qui s'imposent en fonction de l'ensemble du système, de sa politique interne et de l'état du processus léger appelant : ce dernier perdra nécessairement la main s'il est dans l'état `BLOCKED`. Puisque `schedule()` est susceptible de faire perdre la main au processus léger appelant, il est interdit de se trouver dans une section critique au moment de l'appel ;

`wake_up(thread)` qui permet de réveiller (passer dans l'état `RUNNING`) un processus léger s'il est dans l'état `BLOCKED`. Notons que `wake_up()` n'est pas un point où l'ordonnancement est modifié. Un processus léger réveillé devra attendre qu'un appel à `schedule()` soit effectué pour avoir une possibilité d'être ordonnancé à nouveau.

Le fait de séparer le changement d'état (`set_state(`*`état`*`)`) de la demande de prise en compte de cet état par l'ordonnanceur (`schedule()`) permet d'écrire des codes de synchronisation corrects tout en gardant pour l'ordonnanceur une interface réduite et indépendante des mécanismes de synchronisation proprement dits. La Figure 5.2 montre comment passer la main de manière cohérente avec un état (celui d'un verrou) protégé par une section critique.

---

cesseurs physiques ou plus simplement si le système les suspend pour donner la main à un démon ou un autre processus.

Une autre différence concerne la communication interprocesseur. Le noyau utilise des interruptions matérielles spécifiques. Une bibliothèque ne peut reposer que sur les mécanismes de signaux qui sont généralement très coûteux et difficiles à mettre en œuvre correctement. Ce genre de communication inter-LWP est en général évité.

```
mutex_lock(mutex) {
  enter_crit_section(mutex.crit);
  while (is_mutex_hold(mutex)) {
    record_want_mutex(mutex, self);
    set_state(BLOCKED);
    exit_crit_section(mutex.crit);
    schedule();
    enter_crit_section(mutex.crit);
  }
  set_mutex_hold(mutex);
  exit_crit_section(mutex.crit);
}
```

```
mutex_unlock(mutex) {
  enter_crit_section(mutex.crit);
  unset_mutex_hold(mutex);
  next=recorded_want_mutex(mutex);
  if (next!=NULL) {
    wake_up(next);
  }
  exit_crit_section(mutex.crit);
}
```

Ce pseudo-code montre comment implémenter correctement des verrous à l'aide des primitives de synchronisation bas niveau.

Toutes les opérations sont protégées par une section critique, sauf l'appel à `schedule()`*. Ce code reste cependant correct. Imaginons, comme dans la figure précédente, qu'un processus léger A veuille le verrou détenu par le processus léger B. Si B libère le verrou entre le moment où A quitte la section critique et celui où il appelle* `schedule()`*, alors B va bien voir A enregistré, il va le réveiller,* i.e. *le remettre dans l'état* `RUNNIG`*, et lorsque A va appeler* `schedule()`*, l'ordonnanceur ne va pas retirer A de la liste des processus légers prêts puisqu'il n'est plus dans l'état* `BLOCKED`*.*

FIG. 5.2 – Implémentation d'un verrou avec synchronisation interne

Des variables spécifiques. Les synchronisations inter-LWP qui font intervenir plusieurs flots indépendants peuvent être coûteuses (voir ci-dessous). Lorsque cela est possible, il peut être intéressant de les remplacer par des variables spécifiques aux LWP dont l'accès ne demande alors aucune protection : ces variables sont dupliquées en autant d'exemplaires que la bibliothèque a lancé de LWP. Bien entendu, un code utilisant ces variables spécifiques ne doit pas pouvoir changer de LWP au cours de son exécution. Il doit être assuré de garder la main sur le même LWP, ce qui est le cas lorsque ce code est en section critique, mais cela peut aussi être demandé spécifiquement : il suffit d'interdire la préemption et de ne pas appeler de fonction susceptible de passer la main, comme `schedule()` en particulier.

5.1.1.3 Une implémentation adaptée à l'architecture

Les divers mécanismes présentés ci-dessus peuvent être implémentés de manière très différente suivant les architectures considérées. Nous présentons ici quelques unes des implémentations les plus représentatives.

Sections critiques sur architecture monoprocesseur. Dans ce modèle, il n'y a qu'un seul flot d'exécution (un seul LWP) disponible. Pour s'assurer d'une cohérence correcte, il suffit d'interdire la préemption durant la section critique. Le processus léger courant ne pouvant plus passer la main, l'exécution du code de la section critique ne peut pas être entrelacée avec le code d'un autre processus léger. L'entrée et la sortie d'une section critique se ramènent, dans ce cas précis, à la simple modification d'une variable[2].

[2] L'utilisation de primitives de cohérence mémoire peut être également nécessaire sur certaines architectures pour assurer que la mémoire se trouve dans un état cohérent en cas d'interruption par un signal ou un *upcall*.

Sections critiques sur architecture multiprocesseur. Dans ce cas, désactiver la préemption sur le LWP local permet de régler les problèmes de réentrance sur ce LWP (comme dans le cas précédent). Mais ce n'est plus suffisant : plusieurs flots peuvent tenter d'entrer dans la section critique simultanément. Il convient de les séquentialiser correctement.

Dans l'implémentation courante, nous avons choisi pour cela d'utiliser de l'attente active (*spinlock*). En effet, s'il y a contention, nous savons que le processus léger dans la section critique est actuellement ordonnancé dans un autre LWP : comme indiqué ci-dessus, la préemption est désactivée sur les LWP demandant à entrer dans la section critique. Le danger avec l'attente active est d'attendre en consommant du temps processeur alors que le propriétaire de la section critique n'est pas réellement ordonnancé. Nous considérons que, comme notre bibliothèque se destine au calcul hautes performances, la machine est dédiée à l'application. Et comme nous créons autant de LWP que de processeurs, nous pouvons supposer qu'ils sont tous réellement ordonnancés en parallèle sur des processeurs physiques. L'attente active est alors effectivement un moyen efficace de gérer l'accès simultané aux sections critiques[3].

On peut toutefois remarquer que si notre hypothèse[4] se révélait fausse, alors, il suffirait de modifier notre implémentation des sections critiques pour adapter notre bibliothèque. Plusieurs solutions seraient alors envisageables comme, par exemple, insérer des appels à `sched_yield()` dans la boucle d'attente active pour tenter de donner la main au LWP qui est dans la section critique. On pourrait aussi utiliser le mécanisme des FUTEX introduit récemment dans le noyau LINUX qui permet de synchroniser des flots d'exécution gérés par le noyau en restant en mode utilisateur lorsqu'il n'y a pas de contention.

5.1.1.4 Bilan

À l'instar des diverses composantes de notre bibliothèque, la synchronisation bas niveau propose des fonctions pourvues d'une sémantique bien précise. Cela permet aux autres composants de les utiliser efficacement sans avoir à s'interroger sur leur implémentation réelle. Ces fonctions sont instanciées de la manière la plus efficace possible pour l'architecture ciblée lors de la compilation, ce qui assure une efficacité optimale en toutes circonstances. Le code des autres composants reste portable tout en étant correctement optimisé.

5.1.2 Le serveur d'événements d'entrées/sorties

Nous avons présenté à la section 4.3 le serveur d'événements qui permet d'obtenir la meilleure réactivité possible suivant le système donné. Dans le cas où il faut utiliser des méthodes actives (scrutation), notre bibliothèque est chargée d'invoquer régulièrement les fonctions de scrutation. Nous présentons ici les problèmes de synchronisation relatifs à l'appel de ces fonctions.

[3] L'attente active est aussi utilisée par le noyau LINUX sur machines SMP. Ce modèle de synchronisation semble réellement performant comparé à d'autres, ainsi que le rapportent les conversations sur les listes de discussion du développement du noyau (`http://www.kerneltraffic.org/kernel-traffic/kt20041019_278.html#3`).

[4] Le fait de considérer que tous nos LWP sont en permanence ordonnancés chacun sur un processeur physique peut devenir incorrect si notre bibliothèque est utilisée sur un système SMP avec plusieurs applications tournant simultanément.

5.1.2.1 Les difficultés

Pour éviter des changements de contexte inutiles, les fonctions de scrutation sont invoquées directement par notre bibliothèque, et non pas par l'intermédiaire d'un processus léger dédié. Cette décision de conception a pour conséquence d'interdire l'utilisation de mécanismes de synchronisation de haut niveau (verrous, sémaphores, etc.) pour séquentialiser les appels aux fonctions de scrutation puisqu'en cas de contention, il n'y aurait aucun processus léger à suspendre.

Notre première implémentation utilisait les sections critiques. La scrutation et les demandes adressées au serveur d'événements se protègent avec une section critique. Ce système a très bien fonctionné sur architecture monoprocesseur, mais plusieurs problèmes sont apparus sur multiprocesseur :

- lorsque plusieurs LWP n'ont aucun processus léger disponible, ils exécutent une boucle qui appelle les fonctions de scrutation. On espère ainsi qu'un événement va survenir qui conduira au réveil d'un processus léger. Dans cette situation, les différents LWP sans travail attendent en continu le *spinlock* les autorisant à scruter ;
- plus gênant : lorsqu'un seul LWP était sans travail, il monopolisait le *spinlock* en bouclant sur une scrutation en permanence, aussi les autres LWP demandant épisodiquement une scrutation étaient-ils souvent momentanément bloqués sur le *spinlock*.

Pour résoudre ces difficultés, nous avons introduit de nouveaux objets de synchronisation plus adaptés à nos besoins.

5.1.2.2 La solution retenue

Nous avons choisi d'implémenter dans notre bibliothèque le concept de *tasklet* présent dans le noyau LINUX. L'idée est, pour chaque *tasklet*, de définir une tâche, *i.e.* une fonction, dont l'invocation possède plusieurs propriétés :

- si l'on invoque une *tasklet*, alors sa fonction sera exécutée au moins une fois sur un LWP après cet appel ;
- si, lors d'une invocation, l'exécution de la fonction d'une *tasklet* est déjà prévue mais n'est pas encore commencée, alors la fonction ne sera exécutée qu'une seule fois ;
- la fonction d'une *tasklet* s'exécute de manière exclusive vis-à-vis d'elle-même (mais pas des autres *tasklets*).

Ces propriétés sont très utiles sur architecture multiprocesseur. Si l'on place les fonctions de scrutation dans une *tasklet*, alors, il suffit d'invoquer cette *tasklet* pour effectuer une scrutation. Si l'on invoque plusieurs fois la *tasklet*, de manière simultanée à son exécution ou pas, alors, d'une part, il n'y a pas de contention (la *tasklet* est marquée pour exécution ultérieure sans contention possible), et d'autre part, la *tasklet* est exécutée au plus une fois après la dernière invocation (plusieurs invocations simultanées n'exécuteront la *tasklet* qu'une seule fois).

Enfin, pour assurer une fréquence de scrutation régulière, notre serveur d'événements utilise un *timer* qui se contente d'invoquer la *tasklet* correspondante. Encore une fois, l'utilisation d'une *tasklet* permet de garantir qu'une fonction de scrutation n'est pas exécutée simultanément sur deux LWP quelles que soient les raisons de déclencher une scrutation (horloge, LWP sans tâche, etc.)

5.1.2.3 La synchronisation du serveur d'événements avec le code utilisateur

L'utilisation d'une *tasklet* pour exécuter les fonctions de scrutation assure que ces fonctions, définies par l'utilisateur à l'initialisation, sont sérialisées. Cependant, pour permettre à l'utilisateur un emploi correct de nos mécanismes, il reste encore à fournir des outils permettant de synchroniser les fonctions de rappel et le reste du code, en particulier la soumission des requêtes.

On obtient cette synchronisation en fournissant deux fonctions :

```
int marcel_ev_lock(marcel_ev_server_t server);
int marcel_ev_unlock(marcel_ev_server_t server);
```

Ces fonctions permettent de désactiver (puis de réactiver) la *tasklet* correspondant au serveur considéré. Une fois `marcel_ev_lock()` appelé, on est assuré qu'aucune fonction de scrutation n'est en cours et qu'aucune scrutation ne sera démarrée par notre bibliothèque.

Si, après avoir bloqué les scrutations, on appelle une fonction d'attente d'événements, alors la demande est prise en compte puis le système rétablit les scrutations en attendant l'événement. Cela permet à l'utilisateur de programmer le scénario suivant :

1. un processus léger bloque les scrutations ;
2. il modifie une structure de données partagée avec les fonctions de scrutation, par exemple une structure précisant un type d'information ;
3. il attend un événement, ce qui rétablit les scrutations et endort ce processus léger ;
4. la bibliothèque invoque les fonctions de scrutation de l'utilisateur. Ces fonctions sont sérialisées par la *tasklet*, elles peuvent donc accéder sans problème à la structure de données partagée ;
5. une fonction de scrutation signale l'événement attendu par ce processus léger ;
6. le processus léger en attente d'événements est réveillé. Les scrutations sont à nouveau bloquées.
7. le processus léger peut consulter et/ou modifier la structure partagée (la fonction de scrutation a pu stocker des informations pertinentes sur l'événement survenu) ;
8. il rétablit les scrutations pour que les autres processus légers en attente d'événements puissent les recevoir.

5.1.2.4 Bilan

Les besoins particuliers de synchronisation pour le serveur d'événements de notre bibliothèque a conduit à la conception et l'implémentation de mécanismes ad-hoc. Comme précédemment, le travail réalisé s'est d'abord focalisé sur la définition des propriétés que l'on désirait ; cela a conduit à la définition du concept de *tasklet* et à l'introduction de deux nouvelles fonctions pour l'utilisateur. Il s'agit là du travail le plus ardu. Plusieurs tentatives ont été nécessaires pour isoler correctement les concepts clés et les meilleures interfaces.

Ensuite, ces objets ont été implémentés, basés en grande partie sur les outils de synchronisation présentés précédemment (sections critiques, SOFTIRQ, etc.). Ceci leur a assuré une optimisation automatique à la compilation, alors que leur code est écrit de manière complètement portable pour les différentes architectures supportées par notre bibliothèque.

5.2 Activations : gérer les ressources le plus efficacement possible

Ainsi que nous l'avons exposé dans la section 4.4, nous avons modifié le modèle des *Scheduler Activations* afin de le rendre plus efficace dans le contexte du calcul hautes performances. Les principaux bénéfices se traduisent par une diminution du coût des *upcalls* (l'état complet des processus légers n'a pas à être transmis) et de leur nombre (les préemptions dues aux autres processus ne sont plus signalées). Nous voulons détailler ici le déroulement d'un *upcall* afin d'expliciter son coût. D'autre part, nous détaillerons également la gestion des piles utilisateur nécessaires aux *upcalls*.

5.2.1 *Upcall* et signaux

Conceptuellement, un *upcall* et un signal sont deux entités similaires. Tous deux sont générés par le noyau pour signaler un événement à l'application. Le code applicatif est alors dérouté pour exécuter le code de l'*upcall* ou le traitant du signal avant de revenir à son exécution antérieure[5].

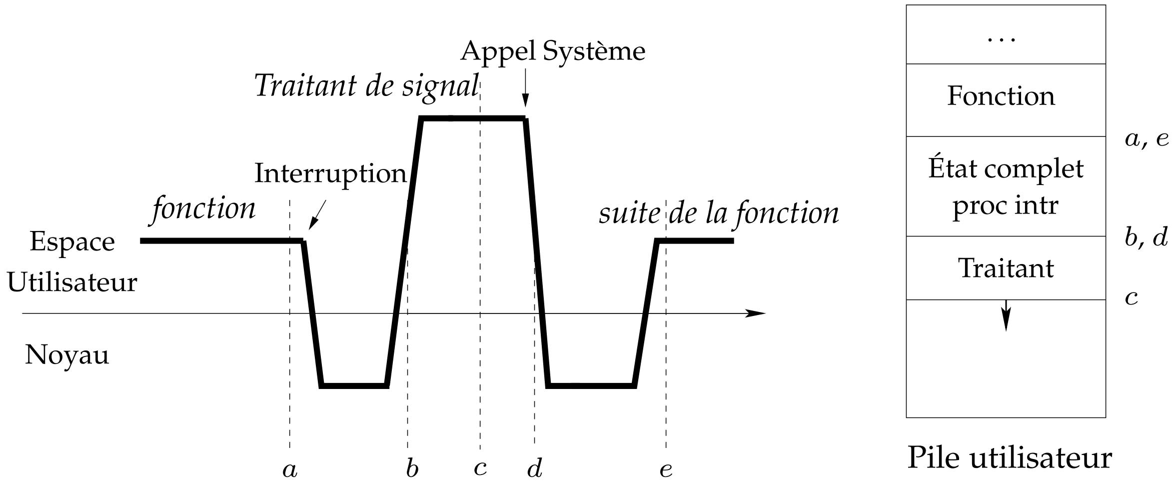
Nous aurions pu utiliser le mécanisme des signaux du système d'exploitation pour réaliser nos *upcalls*. Par exemple, SOLARIS signale à la bibliothèque utilisateur que tous les LWP sont bloqués en lui envoyant un signal de numéro spécifique. Cependant, nos besoins ne s'accordent pas très bien avec l'interface système des signaux : nous devons transmettre à l'application des informations qui sont absentes des signaux traditionnels. En outre, le mécanisme des masques de signaux pour bloquer leur arrivée n'est pas réellement pertinent avec les *upcalls* [6]. Ceci explique que notre implémentation dans le noyau LINUX propose un mécanisme proche de celui des signaux, mais distinct, pour réaliser les *upcalls*. En outre, comme nous allons le voir, la connaissance du contexte dans lequel les *upcalls* sont émis nous a permis parfois d'optimiser leur déroulement.

Lorsqu'un traitant (signal ou *upcall*) doit être exécuté de manière asynchrone vis-à-vis de l'application, le noyau doit sauver l'ensemble de l'état du processeur avant d'exécuter le traitant et cet état doit ensuite être restauré. En effet, l'application peut avoir placé le processeur dans un état quelconque et ce dernier doit être remis dans cet état. À cette fin, le noyau LINUX utilise un appel système spécifique à la fin des traitants de signaux. La description précise de la procédure est détaillée dans la Figure 5.3.

Dans le cas où le déroutement intervient au retour d'un appel système, les conventions d'appels font que le compilateur s'attend à ce que la plupart des registres soient modifiés. Le code applicatif a donc déjà sauvegardé la plupart des informations qui l'intéressent. Aussi, lorsque le noyau génère son *upcall*, il ne doit fournir qu'une petite partie de l'état du processeur. En outre, il est alors facile au code « trampoline » de restaurer directement cet état au lieu de demander au système de le faire. On évite ainsi une double transition espace utilisateur/noyau. Ce déroulement est illustré sur la Figure 5.4. Il peut être utilisé, par exemple, lors de chaque *upcall* `restart` qui est généré au redémarrage d'une activation après un appel système bloquant.

[5]Le cas d'un *upcall* avec l'information `New()` est légèrement différent : puisqu'il signale un nouveau flot d'exécution disponible, l'*upcall* ne doit pas terminer mais plutôt basculer vers l'exécution d'un processus léger.

[6]Parfois, il est effectivement nécessaire de bloquer les *upcalls*, mais utiliser des appels systèmes `sigsetmask()` pour cela serait trop coûteux ; le bénéfice d'une bibliothèque de processus légers de niveau utilisateur serait perdu avec toutes ces transitions noyau/espace utilisateur.



Un signal peut survenir à tout instant, quel que soit l'état du processeur (registre de code condition ou unité flottante en cours d'utilisation, etc.)

a : *l'application est interrompue au lieu d'une fonction ;*

b : *le système se rend compte qu'il doit délivrer un signal. L'état complet du processeur est placé sur la pile de l'application puis le processeur est « réinitialisé » ;*

c : *le traitant du signal s'exécute, la pile est descendue plus bas ;*

d : *un appel système est exécuté à la fin du traitant. Le noyau trouve l'ancien état du processeur sur la pile ;*

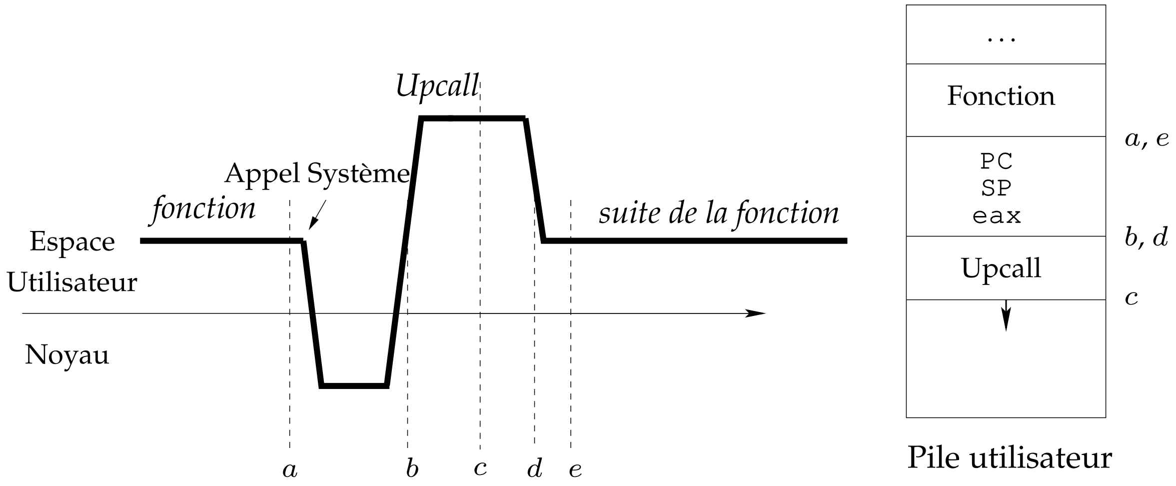
e : *le processeur est restauré dans son ancien état, la pile reprend sa valeur au moment du déroutement.*

FIG. *5.3 – Déroulement d'un signal*

Tous les *upcalls* ne peuvent bien sûr pas bénéficier de cette amélioration : ceux intervenant de manière asynchrone, par exemple pour signaler la fin d'un appel système bloquant d'une autre activation, ne sont pas concernés. Mais pour les autres, cette amélioration remplace une coûteuse sauvegarde de l'état du processeur et sa restauration (appel système) par la sauvegarde de quelques registres (3 sur architectures X86) et quelques instructions assembleur pour les restaurer.

5.2.2 Gestion de la réentrance et des piles avec les activations

L'une des difficultés des *Scheduler Activations* qui n'a pas encore été évoquée concerne la gestion des piles utilisées lors de l'exécution des *upcalls*. En effet, un *upcall*, aussi bref soit-il, a besoin d'une pile en espace utilisateur afin de stocker ses variables locales, les arguments des fonctions appelées, etc. La difficulté provient de la dualité du responsable de son allocation, le noyau, et du responsable de sa libération, la bibliothèque utilisateur lorsqu'elle ordonnance un de ses processus légers dans l'activation. Les signaux n'ont pas ce problème car le noyau peut utiliser la pile courante le temps du traitant : le flot d'exécution est dérouté, suspendu pendant le traitement du signal. Avec les activations, ce schéma n'existe plus : de nouveaux flots d'exécution sont créés lorsqu'une activation en fonctionnement bloque, alors que d'autres sont supprimés lors des préemptions dans le modèle original. Le noyau doit donc proposer une pile pour l'exécution de chaque *upcall*.



Lors d'un appel système, l'état du processeur peut être modifié. L'application doit sauver la plupart des registres qui l'intéressent avant l'appel système. Un upcall *s'exécutant là peut utiliser les registres du processeur sans créer de conflit.*

a : *l'application exécute un appel système. Elle a donc sauvé les registres du processeur qui l'intéressent (sur la pile par exemple) ;*

b : *le système se rend compte qu'il doit délivrer un* upcall*. Seuls les quelques registres qui doivent être restaurés au retour des appels systèmes sont sauvés sur la pile ;*

c : *la pile est descendue plus bas, l'*upcall *s'exécute ;*

d : *le code de l'*upcall *restaure lui-même les quelques registres nécessaires avant de se placer au lieu du retour de l'appel système ;*

e : *la fonction continue après l'appel système et replace dans les registres du processeur les valeurs qu'elle souhaite.*

On économise ainsi un appel système, donc deux transitions espace utilisateur/noyau, et les données à sauver sur la pile sont beaucoup moins volumineuses.

FIG. 5.4 – *Déroulement d'un* upcall *synchrone*

L'article original présentant ce modèle [ABLL91] n'aborde pas ce problème. En revanche, l'adaptation de ce modèle sur le système NETBSD [Wil02] utilise pour sa part tout un pool de piles que l'application fournit au système. Le noyau utilise alors ces piles une à une à chaque *upcall*, l'application se chargeant de les rendre au système pour réutilisation avec un appel système. Le coût de cet appel est amorti : l'application n'effectue cet appel que lorsqu'elle a suffisamment de piles à remettre dans le pool.

Nous avons choisi une autre approche, tirant parti de notre décision d'abandonner les notifications de préemption. Dans notre implémentation, le noyau rencontre deux situations :

– soit il s'agit d'une nouvelle activation et dans ce cas, le noyau utilise une pile propre au processeur correspondant. À cet effet, lors de l'initialisation, l'application fournit au noyau une pile par processeur ;
– soit il s'agit d'un *upcall* au sein d'une activation existante et dans ce cas, le noyau utilise la pile de l'activation interrompue. On rencontre cette situation quand il s'agit du redémarrage d'une activation qui avait été bloquée ou bien lorsqu'un *upcall* est généré pour

signaler un événement asynchrone comme le réveil d'une autre activation bloquée.
Le danger, surtout pour le premier cas, est alors de réutiliser la pile alors que l'*upcall* n'est pas terminé. Cela peut arriver si le code de l'*upcall* bloque lui-même. La solution que nous proposons est l'utilisation d'une variable partagée par processeur entre l'application et le noyau. Cette variable est mise à 1 par le noyau avant de commencer un *upcall* et à 0 par l'application lorsque l'*upcall* est fini. En outre, si une activation bloque pendant que cette variable est positionnée, alors le noyau ignore ce blocage. Plus précisément, le flot d'exécution est bien bloqué dans le noyau, mais aucun *upcall* n'est généré pour prévenir ni du blocage, ni du redémarrage.

Cette approche permet également de définir des sections critiques sans *upcall* : il suffit que l'application positionne cette variable. Ceci est particulièrement utile pour les portions de code où l'on ne souhaite aucun changement de contexte, comme le cœur de l'ordonnanceur par exemple. Enfin, cette approche est particulièrement peu coûteuse puisque qu'elle ne génère aucune transition espace utilisateur/noyau. Le noyau doit juste consulter une fois cette variable lorsqu'une activation bloque pour savoir s'il doit ou non tenir compte du blocage.

5.2.3 Bilan

L'analyse précise du contexte des *upcalls* a permis quelques optimisations de leur exécution. La restauration de l'état du processus léger que l'*upcall* déroute peut parfois être grandement simplifiée, conduisant ainsi à des gains substantiels (cf. les mesures présentées dans le chapitre suivant). L'étude de l'enchaînement des *upcalls* au sein d'une application a également permis de proposer une implémentation gérant efficacement les piles utilisateur : contrairement à l'implémentation du système NetBSD, notre modèle n'a besoin que d'un nombre fixe de piles (égale au nombre de processeurs de la machine) pour fonctionner correctement. Il permet en outre à l'application de définir des sections critiques vis-à-vis des *upcalls* pour un coût négligeable (lecture et écriture d'une variable). Les choix que nous avons faits au niveau de notre modèle ont permis cette implémentation efficace.

5.3 Un code caméléon

La multiplicité des choix offerts à l'utilisateur de notre bibliothèque a nécessité la mise en place d'une architecture logicielle et le développement d'outils adaptés qui sont présentés dans un premier temps. Nous montrons ensuite comment le code de la bibliothèque reste en grande majorité identique quels que soient les choix effectués, ceci, afin d'éviter au maximum la duplication de code. Enfin, cette adaptabilité du code est mise en évidence avec le problème de la réentrance des bibliothèques extérieures vis-à-vis de la nôtre.

5.3.1 Une multitude d'options

La flexibilité de notre bibliothèque, qui doit pouvoir facilement s'adapter et répondre aux besoins des applications ou environnement de programmation, constitue un point clé de notre approche. Cela nous interdit l'approche traditionnelle consistant à proposer une bibliothèque prête à compiler de la même façon dans tous les cas.

Il existe, certes, des mécanismes permettant de faire certains choix lors de la compilation. Ainsi, avec les outils `automake/autoconf`, il est possible de spécifier des options qui seront prises en compte lors de la compilation. Cependant, ces mécanismes ne nous ont pas semblé suffisamment souples pour nos besoins. En effet, notre bibliothèque requiert des spécialisations à plusieurs niveaux.

Compilation de la bibliothèque : il s'agit de n'inclure dans la bibliothèque compilée que les fonctionnalités souhaitées.

Compilation du logiciel client : le logiciel client souhaitant utiliser notre bibliothèque peut avoir besoin d'options spécifiques lors de sa compilation ou de son édition de lien. Ainsi, pour bénéficier d'un traçage automatique de ses fonctions avec nos outils de traces, il est nécessaire d'activer une option spécifique du compilateur.

Exécution du logiciel client : suivant les choix effectués pour notre bibliothèque, le logiciel client ne se lancera pas nécessairement de la même façon. Par exemple, certains choix imposent le préchargement de bibliothèques dynamiques spécifiques.

Nos mécanismes doivent également permettre de gérer l'ensemble des choix de notre plate-forme applicative, c'est-à-dire notre environnement de programmation PM^2. Cet environnement comporte de nombreux composants dont les deux plus importants sont sans doute MARCEL, bibliothèque de processus légers, et MADELEINE, bibliothèque de communications.

Flavor. Pour répondre à tous ces besoins, nous avons développé la notion de *flavor*. À une *flavor*, correspond l'ensemble des options choisies pour notre environnement de programmation. Ces choix comprennent principalement :

les modules à inclure : PM^2 comprend de nombreux modules. Celui qui correspond à la gestion des processus légers est MARCEL ;

les options des modules : chaque module possède un certain nombre d'options qui peuvent être sélectionnées ou non. Pour MARCEL, cela correspond aux fonctionnalités que l'on désire ou non inclure ;

des options globales : ce sont des options relatives à la *flavor* dans son ensemble tels que son nom, son répertoire de compilation, etc.

Par la suite, sauf exception, je ne considérerai que les choix relatifs à la bibliothèque de processus légers, bien que les discussions qui seront développées pourraient s'appliquer à l'ensemble de l'environnement PM^2.

Plusieurs acteurs sont concernés par l'aspect caméléon de notre environnement. Chacun d'entre eux a cependant des besoins différents explicités ci-dessous.

5.3.1.1 Pour le développeur

Il s'agit ici de définir les besoins des développeurs de l'environnement de programmation PM^2, c'est-à-dire principalement de notre équipe de recherche ainsi que des contributeurs extérieurs qui se sont intéressés à notre projet ;

Compilations multiples. De par la grande variété de choix offerte, le développeur est très souvent amené à comparer différentes *flavors*. Il convient donc que différentes *flavors* puissent être facilement compilées simultanément sur la même machine. Cela s'obtient en plaçant tous les fichiers produits par une compilation dans un répertoire spécifique à la *flavor*, les répertoires contenant les sources de notre environnement n'étant jamais modifiés lors du processus de compilation.

Modification de la palette de choix offerte. Le développeur doit être capable de modifier facilement la palette des choix qui sont offerts à l'utilisateur. Par exemple, il ne faut pas que l'ajout d'un choix nécessite l'adaptation de l'ensemble des programmes qui manipulent les *flavors*. Ceci est obtenu par une architecture en couches indépendantes les unes des autres. On trouve ainsi :

la définition des options : les options sont décrites grâce à quelques fichiers ;

la gestion des options : les options (et les modules) sont manipulés par l'intermédiaire du programme `pm2-module` qui est le seul à connaître la structure des fichiers définissant les options ;

la gestion des *flavors* les *flavors* sont créées ou modifiées avec le programme `pm2-flavor`. Les choix d'une *flavor* sont rassemblés avec le programme `pm2-module` et stockés dans un fichier propre à la *flavor*. Seuls les programmes `pm2-flavor` et `pm2-config` qui, lui, permet d'obtenir les informations spécifiques à une *flavor* manipulent directement le fichier de définition de la *flavor*, `pm2-config` se contentant d'un accès en lecture.

5.3.1.2 Pour les applications externes

Après avoir exposé la facilité avec laquelle un développeur peut ajouter ou modifier une option, voyons comment une application ou un environnement utilisant notre bibliothèque peut s'interfacer. Il s'agit ici de fournir un moyen pour compiler et lier un programme avec notre environnement qui puisse être utilisé facilement depuis la chaîne de développement d'un projet plus large, comme PADICOTM ou MPI-CH par exemple.

La solution retenue est de proposer des outils en ligne de commande répondant aux besoins de ces environnements. Les trois principaux sont présentés ici :

`pm2-flavor`: ce programme permet de définir une *flavor* à partir d'une liste d'options passée en argument. Typiquement, ce programme sera appelé lors de l'étape de configuration de l'application ou de l'environnement utilisant notre bibliothèque. Une dizaine de *flavors* correspondant aux choix classiques sont automatiquement crées lors de l'installation de notre suite logicielle. Si on les utilise, il n'est pas nécessaire, alors, d'en créer une nouvelle avec `pm2-flavor` ;

`pm2-config`: ce programme permet de récupérer les options de compilation et de liens à utiliser. Un programme `prog.c` utilisant notre bibliothèque pourra être compilé par les commandes suivantes :

```
gcc `pm2-config --flavor=nom_flavor --cflags` -c prog.c
gcc `pm2-config --flavor=nom_flavor --libs` prog.o -o prog
```

`pm2load`: ce programme lance l'exécutable en chargeant au préalable les bibliothèques dynamiques nécessaires si besoin est. Pour reprendre l'exemple précédent, on aurait alors :

```
pm2load -f nom_flavor prog arg1 arg2 ...
```

`pm2load` a diverses options qui permettent par exemple de lancer l'exécutable au sein de `gdb`.

Pour éviter de spécifier le nom de la *flavor* en permanence, il est possible de positionner la variable d'environnement `PM2_FLAVOR`.

5.3.1.3 Pour l'utilisateur

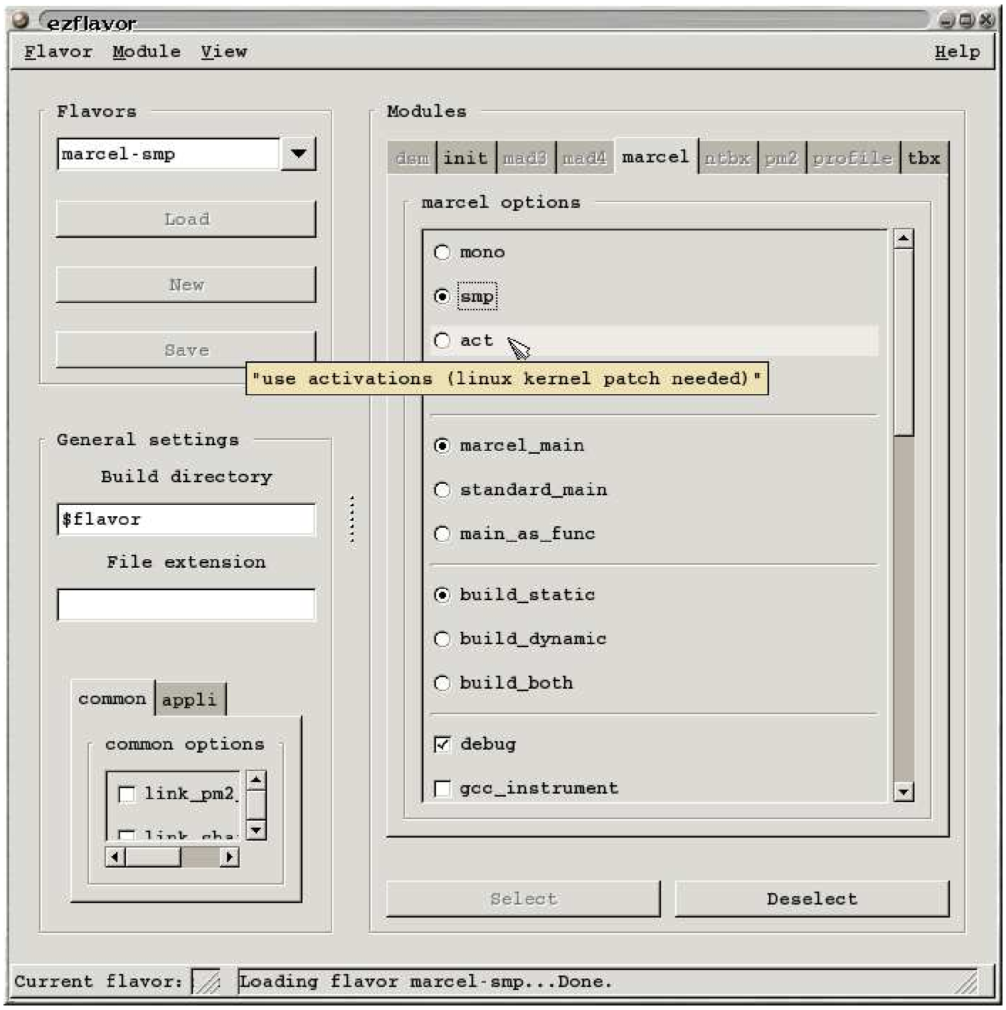


FIG. 5.5 – `ezflavor`, un moyen simple de faire des choix

Les utilisateurs humains (par opposition aux environnements automatisés) utilisant notre bibliothèque peuvent, bien entendu, se servir des outils présentés ci-dessus. Cependant, créer une *flavor* n'est pas très aisé, surtout si l'on ne connaît pas par cœur l'ensemble des options disponibles. C'est pourquoi nous avons développé des outils spécifiques pour faciliter la création de nouvelles *flavors* adaptées tout particulièrement à un projet particulier.

Il existe deux programmes répondant à ce besoin : le premier, `pm2-config-flavor`, interactif, en ligne de commande et permettant de construire pas à pas sa *flavor* et le second, `ezflavor`, graphique. Tous les deux s'assurent que les contraintes de dépendance sont correctes avant de sauver une nouvelle *flavor*. La Figure 5.5 présente une capture d'écran de l'application `ezflavor` où l'on peut voir que l'utilisateur n'a qu'à cliquer pour sélectionner ses choix. Des bulles d'aide apparaissent pour expliquer ces choix au passage de la souris.

Il est à noter que ces deux programmes récupèrent à leur démarrage l'ensemble des informations sur les modules et les options avec le programme `pm2-modules`. Ainsi, lorsque l'on rajoute les fichiers d'une nouvelle option, il n'est pas nécessaire de recompiler et encore moins de modifier ces deux programmes pour que la nouvelle option apparaisse : relancer le programme suffit.

5.3.2 Un unique code

Notre environnement de développement permet de proposer un vaste choix de fonctionnalités que l'on peut ou non inclure dans la bibliothèque à compiler. Nos outils permettent de sélectionner ces options de manière conviviale, de compiler facilement les programmes une fois ces options choisies et, pour le développeur, de rajouter aisément de nouvelles options.

La plupart des options qui correspondent directement à une fonctionnalité font en sorte de définir un symbole pour le préprocesseur. Cela permet au code de la bibliothèque d'être compilé de manière conditionnelle en fonction de ce symbole :

```
  if ((thread->state == READY)
#ifdef HAS_SEVERAL_LWPS
      && (thread->lwps_allowed & (1 << current_lwp->number))
#endif /* HAS_SEVERAL_LWPS */
  ) {
    /* On donne la main à ce processus léger */
    ...
  }
```

Cependant, la multiplicité des options proposées rendrait un tel code difficile à lire et à maintenir. C'est pourquoi une autre organisation a été retenue. Le code propre à une fonctionnalité est placé dans une macro ou une fonction « en ligne ». Seul ce code est déclaré de manière conditionnelle. Le reste de la bibliothèque utilise toujours cette macro ou fonction « en ligne » qui, dans le cas où la fonctionnalité n'a pas été sélectionnée, ne correspond à aucun code et sera donc éliminée par le compilateur. Ainsi, le code précédent peut être réécrit de la façon suivante :

```
  if ((thread->state == READY) && can_schedule(thread, current_lwp)) {
    /* On donne la main à ce processus léger */
    ...
  }
```

avec, dans un fichier en-tête :

```
#ifdef HAS_SEVERAL_LWPS
#  define can_schedule(thread,lwp) \
         (thread->lwps_allowed & (1 << lwp->number))
#else /* HAS_SEVERAL_LWPS */
#  define can_schedule(thread,lwp) 1
#endif /* HAS_SEVERAL_LWPS */
```

Cela permet d'écrire des fonctions beaucoup plus lisibles car non encombrées de nombreuses directives de préprocesseur emboîtées.

Dans le même souci de permettre un développement du code de la bibliothèque plus homogène, nous avons cherché à réduire au maximum la duplication de code. Cet aspect est particulièrement visible lorsque l'on s'intéresse aux diverses personnalités proposées. Une même fonction peut avoir plusieurs noms différents suivant la personnalité par laquelle on y accède. Par exemple, la fonction qui permet à un processus de demander à passer la main à un autre processus peut s'appeler `marcel_yield`, `pmarcel_yield` ou `pthread_yield` : la première correspond à la personnalité la plus optimisée de MARCEL (mais ne suivant pas toujours la norme POSIX), la seconde correspond à la personnalité respectant la norme POSIX mais sans entrer en conflit avec son API, la dernière correspondant à la personnalité respectant totalement la norme POSIX, API comprise.

Dans chacun de ces cas, écrire trois fois le même corps de fonction aurait conduit, à terme, à l'apparition de petites divergences (corrections faites dans une version et pas dans une autre, etc.) qui auraient induit un code difficile à maintenir et à faire évoluer correctement. C'est pourquoi nous avons, là encore, mis à contribution le préprocesseur et les fonctionnalités de l'éditeur de liens afin de ne définir la fonction qu'une seule fois dans le code source, les outils de compilation se chargeant de définir des *alias* comme il se doit. Pour notre exemple, les fonctions `yield` apparaissent dans le code source sous l'unique forme :

```
DEF_MARCEL_POSIX(int, yield, (void))
{
        ... /* code de la fonction */
}
DEF_PTHREAD(yield)
```

Suivant les interfaces demandées dans la *flavor* en cours de compilation, les *alias* corrects seront générés grâce aux macros et aux fonctionnalités du compilateur.

Bilan. Afin de rendre l'écriture et la relecture du code plus faciles malgré les nombreuses options disponibles, nous avons utilisé au maximum les outils et les fonctionnalités du préprocesseur, du compilateur et du lieur, de manière à avoir un code propre sans redondance. Grâce à l'emploi de macros et de fonctions « en ligne », notre code se spécialise comme demandé dans la *flavor* à la compilation, sans que ces différences polluent le fichier source. La maintenance du code complet de la bibliothèque en est ainsi grandement facilitée.

5.3.3 Réentrance vis-à-vis des bibliothèques extérieures

Nous allons montrer ici comment s'applique la spécialisation du code sur un exemple précis, à savoir la protection des appels de bibliothèques extérieures vis-à-vis de MARCEL.

Nous avons déjà discuté à la section 4.2.4.2 des différents cas que l'on peut rencontrer pour les bibliothèques extérieures vis-à-vis d'une bibliothèque de processus légers. Dans cette partie, nous ne considérerons que le cas le plus courant de nos jours, à savoir une bibliothèque qui a déjà été compilée de manière à être réentrante vis-à-vis des processus légers offerts par la bibliothèque du système. C'est le cas, par exemple sous LINUX, de la bibliothèque C (`glibc`) et donc des fonctions `printf()`, `malloc()`, `free()`, etc.

5.3.3.1 De nombreuses situations différentes à gérer

Un programme voulant utiliser MARCEL comme bibliothèque de processus légers va devoir protéger ses appels aux fonctions de la `glibc`. Suivant les fonctionnalités et les personnalités qui auront été incluses dans la *flavor* utilisée par le programme, différentes actions sont nécessaires. Nous allons maintenant détailler les différents cas.

Personnalité POSIX incluse. C'est le cas le plus simple. MARCEL fournit alors l'ABI attendue par les bibliothèques du système et remplace complètement la bibliothèque de processus légers d'origine. Comme les bibliothèques extérieures sont réentrantes par rapport à celle du système, elles le deviennent automatiquement vis-à-vis de MARCEL. Malheureusement, cette solution ne s'applique pas dans tous les cas puisqu'il n'est pas toujours possible de proposer une personnalité POSIX. Certains systèmes n'offrent pas la possibilité de remplacer facilement leur bibliothèque de processus légers. D'autre part, il est très coûteux en temps de développement de suivre régulièrement leurs évolutions. C'est pourquoi, pour l'instant, MARCEL ne possède la personnalité POSIX que sur le système LINUX pour architecture X86. Étendre ce résultat à d'autre architectures sous LINUX ne pose pas de difficultés majeures (sinon trouver le temps nécessaire à ce travail). Étendre ce résultat à d'autres systèmes que LINUX nécessite une étude précise de l'interface de la bibliothèque de processus légers de ce système.

Dans les cas suivants, nous considérerons que la personnalité POSIX n'est pas disponible.

Support pour machines monoprocesseur. MARCEL est alors une bibliothèque de processus légers de niveau purement utilisateur. On peut distinguer deux situations, selon que MARCEL a été compilé avec ou sans préemption automatique. Sans préemption, comme dans le cas précédent, il n'y a rien à faire : MARCEL ne peut passer la main d'un processus léger à un autre qu'à l'occasion d'un appel direct à l'une de ses fonctions. Le processus léger ne peut donc pas être à cet instant au milieu d'une fonction de la bibliothèque extérieure.

Si la *flavor* utilisée contient le support pour la préemption, il faut éviter qu'un processus léger soit interrompu pendant un appel à une fonction de la bibliothèque extérieure. On risquerait alors de donner la main à un autre processus léger qui exécuterait la même fonction simultanément, sans que l'accès aux variables partagées soit protégé. Une solution est alors de désactiver la préemption avant l'appel de la fonction de la bibliothèque extérieure et de la réactiver au retour.

Support pour machines multiprocesseurs grâce à la `libpthread`. Dans ce cas, MARCEL est une bibliothèque mixte qui utilise les processus légers de niveau noyau du système pour obtenir des flots d'exécution parallèles (LWP). Deux processus légers utilisateur sur deux LWP différents (donc dans deux processus légers systèmes différents) peuvent exécuter en même temps des fonctions de la bibliothèque extérieure puisque celle-ci est réentrante vis-à-vis des processus légers systèmes. En revanche, comme dans le cas précédent, il faut éviter qu'un processus léger utilisateur ne passe la main à un autre au milieu d'une telle fonction. C'est pourquoi, là encore, une solution est de désactiver la préemption sur un LWP, le temps de l'appel à la fonction de la bibliothèque extérieure.

Support pour machines multiprocesseurs sans la `libpthread`. MARCEL est alors une bibliothèque mixte qui utilise des moyens spécifiques au système pour obtenir des flots d'exécution parallèles, comme par exemple l'appel système `clone()` sous LINUX. Dans ce cas, deux processus légers utilisateur dans deux LWP ne doivent pas exécuter simultanément une fonction de la bibliothèque extérieure : aucune protection n'est en place. Dans ce cas, une solution est de séquentialiser tous les accès à la bibliothèque extérieure en les protégeant par un verrou.

5.3.3.2 Une solution unique pour l'application

La flexibilité de notre bibliothèque a pour conséquence, dans cette situation, de multiplier les actions nécessaires pour que le code de l'application soit correct. Certes, il serait possible de se placer dans le cas le plus contraignant (le dernier évoqué) et de toujours se protéger avec un verrou. Cependant, cela serait beaucoup moins efficace dans tous les autres cas évoqués, en particulier dans celui où la personnalité POSIX est disponible puisque tous les processus légers peuvent appeler la bibliothèque extérieure simultanément.

La bonne solution est, là encore, de faire exprimer par l'application le service qu'elle désire, à savoir se protéger correctement pour effectuer un appel à une fonction réentrante vis-à-vis des processus légers système. Ce service sera ensuite correctement instancié par MARCEL de la manière la plus efficace possible en fonction des particularités de la *flavor* courante. Dans ce cas, cela se traduit par la mise à disposition de l'application de deux fonctions :

```
void marcel_extlib_protect();
void marcel_extlib_unprotect();
```

Ces fonctions doivent être utilisées par le programme avant et après tout appel à une fonction d'une bibliothèque extérieure devant être protégée pour être réentrante vis-à-vis de MARCEL. Suivant les fonctionnalités incluses dans la *flavor*, ces fonctions seront ignorées, désactiveront temporairement la préemption ou définiront une section critique grâce à un verrou.

Le programme exprime son besoin, notre bibliothèque fournit une solution efficace et adaptée pour répondre à ce besoin. Le développeur du programme n'est pas concerné par la mise en œuvre réelle de cette solution.

5.4 Conclusion

Ce chapitre, à travers l'étude de quelques aspects précis de notre implémentation, a permis d'illustrer les objectifs de développement que l'on s'était fixés. La description de mécanismes de synchronisation a fourni un exemple de la spécialisation automatique de notre bibliothèque en fonction de l'architecture ciblée. Nous avons également vu, à travers l'étude détaillée de nos mécanismes des *Scheduler Activations*, comment leur conception soignée nous permet d'améliorer toujours davantage l'efficacité de notre code.

Le développement d'un logiciel tel que le nôtre est une tâche ardue. Les problématiques sont très voisines de celles que l'on rencontre dans le développement du noyau LINUX. Ainsi, au niveau de l'environnement de développement, les outils d'aide à la mise au point ne sont pas toujours disponibles : le « déboggueur » `gdb` s'interface très difficilement avec notre bibliothèque à deux niveaux. Même l'utilisation de la fonction `printf()` doit parfois être proscrite pour ne pas générer de nouveaux problèmes, entre autres dans les sections critiques.

Sur le plan conceptuel, nos travaux ont également nécessité des efforts importants. Pour obtenir la modularité et les interfaces neutres de notre bibliothèque, de nombreux cycles de développement ont été nécessaires : proposition, implémentation, expérience et retour des utilisateurs. Je tiens d'ailleurs à signaler que l'utilisation de notre environnement par plusieurs projets extérieurs a été très bénéfique, à la fois pour améliorer la stabilité grâce aux signalements d'erreurs, et aussi pour le retour d'expérience sur l'adéquation des services fournis aux besoins réels. De longues discussions animées ont parfois été nécessaires avant de trouver une solution satisfaisante à toutes les parties.

Chapitre 6

Mécanismes de prise de trace

Sommaire

Déterminer les goulots d'étranglement, vérifier le bon comportement d'une application dans des circonstances très précises ou encore comprendre la portée réelle d'une optimisation nécessitent une analyse extrêmement fine de l'exécution d'une application.

Nous avons présenté, dans le chapitre précédent, l'architecture et les mécanismes que nous avons retenus pour obtenir une bibliothèque de processus légers dont l'efficacité est portable d'un système à un autre. Cela s'est traduit par la proposition d'une bibliothèque caméléon pouvant être de niveau utilisateur ou mixte en fonction de la machine ciblée (mono- ou multiprocesseur), et utilisant ou non des *Scheduler Activations* suivant si cette fonctionnalité est offerte par le système. En raison de tous ces mécanismes, l'ordonnancement final des processus légers de l'application peut donc se révéler complexe à prévoir.

L'objectif du travail présenté dans ce chapitre est la réalisation d'un environnement permettant d'analyser finement le comportement précis d'un programme multithreadé, même lorsque les mécanismes d'ordonnancement sont complexes. Cet environnement permettra, par exemple, de déterminer le temps CPU individuellement consommé par chaque processus léger de l'application et sur quel(s) processeur(s) il s'est exécuté. Pour parvenir à nos fins, nous devons concilier plusieurs impératifs :

Être très peu intrusif. Non seulement le surcoût doit être faible afin de modifier le moins possible les performances du programme, mais aussi pour ne pas influer sur l'ordonnancement *normal* des processus légers de l'application (par exemple en augmentant fortement le temps d'exécution des sections critiques, en créant de nouveaux points de changement de contexte pour l'ordonnanceur, etc.)

Savoir gérer une grande quantité d'informations. La connaissance du comportement précis d'une application multithreadée nécessite la récolte de beaucoup d'informations (décisions d'ordonnancement, entrée et sortie des processus légers pour les fonctions qu'ils exécutent, etc.) Il convient d'être attentif à la masse de données générées, de l'ordre du Mo/s, afin que l'application n'en souffre pas (consommation mémoire, etc.)

Maîtriser les ordonnancements complexes. Les techniques développées devront permettre de mettre en évidence les ordonnancements réels des processus légers de l'application sur les plates-formes complexes (SMP avec bibliothèque de processus légers à deux niveaux). À cette fin, il apparaît clairement que l'on aura besoin d'informations de l'ordonnanceur utilisateur (en particulier les changements de contexte des processus légers utilisateur) mais aussi d'informations du système d'exploitation (quand et sur quel processeur s'exécutent les LWP).

Après un rappel des techniques de recueil de performances, nous présentons notre propre technique qui repose sur l'utilisation d'une trace noyau et d'une trace utilisateur pour observer l'exécution des applications multithreadées ; nous présentons ensuite notre environnement de profilage et détaillons, en quelques points, les techniques que nous avons mises en œuvre pour garantir la qualité des mesures obtenues.

6.1 Techniques pour le recueil des performances

Pour analyser l'exécution d'une application, on utilise des outils de *profilage* ; ces outils peuvent être séparés en deux catégories suivant le point de vue (noyau ou applicatif) qu'ils adoptent pour observer l'exécution de l'application.

Le point de vue *noyau* est obtenu essentiellement par une instrumentation légère du système d'exploitation. Cette instrumentation est souvent intégrée au code source du noyau par le programmeur (LINUX TRACE TOOLKIT [YD00], FAST KERNEL TRACE [RC01]), mais elle peut l'être aussi dynamiquement (KERNINST [TM99]). Ces outils permettent une analyse très fine des applications : dans [RC01], une analyse des performances de la pile TCP est menée, afin de comprendre le manque d'efficacité relatif d'un circuit de calcul des sommes de contrôle.

Le point de vue *applicatif* est obtenu par l'instrumentation du programme exécutable, souvent au moyen d'outils intervenant à la génération de code. De nos jours, toutes les plates-formes de développement sont pourvues de ce type d'outils ; parmi ceux-ci, on peut citer les utilitaires GNU `gcov` et `gprof` [GKM82] ou encore l'outil d'INTEL `VTune` qui est capable de tirer partie des compteurs de performance des processeurs INTEL et de suivre le cheminement des processus légers ordonnancés par le système d'exploitation.

6.1.1 Un exemple de base : `gprof`

L'outil `gprof` [GKM82] fournit pour chaque fonction (ou ligne) du programme son nombre exact d'appels et une estimation du temps pris par cette exécution. Ce profilage se fait en trois étapes :

1. instrumention du programme cible à la compilation : un appel à une fonction de comptage est inséré au début de chaque fonction afin d'incrémenter le compteur attaché à l'arc `(fonction appelée, adresse de retour);`
2. recueil des informations lors de l'exécution : outre les informations nécessaires au graphe d'appel, la valeur du compteur ordinal est enregistrée par échantillonnage dans un « histogramme » (tous les centièmes de seconde par exemple, ou mieux, en variant de façon aléatoire la fréquence d'échantillonnage) ;
3. synthèse *post-mortem* des informations : le graphe d'appel est reconstitué grâce aux compteurs attachés aux arcs et la durée d'exécution des fonctions est estimée grâce à l'histogramme du compteur ordinal.

Ainsi `gprof` exploite les deux techniques de base du profilage des performances : l'enregistrement d'une trace d'événements et l'échantillonnage du compteur ordinal. Contrairement au traçage d'événements, la technique d'échantillonnage ne nécessite pas l'instrumentation du code et elle est moins intrusive ; cependant, les durées d'exécution ainsi estimées reflètent rarement la réalité. Par exemple, l'estimation de la durée d'exécution d'une fonction est calculée par `gprof` une fois pour toutes, indépendamment de la valeur des paramètres effectifs ou du chemin d'appel.

6.1.2 Éléments matériels pour l'évaluation des performances

La seule mesure du temps ne peut mettre en évidence des contre-performances d'origine matérielle (tels des défauts de cache répétés), c'est pourquoi tous les processeurs modernes intègrent des *compteurs de performances*.

À titre d'exemple, le PENTIUM IV dispose d'un compteur de cycles sur 64 bits et de 18 registres compteurs de performances où 45 types d'événements peuvent y être dénombrés [Int03]. Ces compteurs de performances sont spécialisés (4 concernent plutôt le bus et la mémoire, 4 concernent l'horloge et les sauts, 4 concernent les flottants et 6 concernent le pipeline). Chaque registre dispose d'un compteur (40 bits) et de bits de contrôle qui servent (1) à définir l'ensemble des événements à compter et (2) à fixer le nombre d'occurrences à atteindre pour déclencher une interruption. Il est ainsi possible de baser l'échantillonnage du compteur ordinal, non plus seulement sur le temps, mais aussi sur le décompte d'événements déterminés (tous les 1000 défauts de cache de niveau L2, par exemple) ; cette technique est mise en œuvre par le profileur `VTune`.

Si de nombreuses bibliothèques spécialisées facilitent l'accès aux compteurs de performances, les bibliothèques `PAPI` [BDG+00] et `PCL` [BZ98] se dégagent en ce sens qu'elles proposent des API portables exploitant les compteurs de performances des processeurs actuels. `PAPI` propose en outre des fonctionnalités plus complexes comme le multiplexage temporel des compteurs (mise à disposition de compteurs virtuels au détriment de la précision des mesures) et la création d'histogrammes basés sur des compteurs de performances.

Notons enfin que l'enregistrement d'un registre de performance coûte avec `PAPI` de l'ordre de 1300 cycles sur PENTIUM [Muc02] et devrait être bientôt ramené à 230 cycles (le coût d'accès à un registre de performance étant de l'ordre de 100 cycles).

6.1.3 Instrumentation des applications réparties

À la base, l'analyse des applications réparties réclame l'instrumentation des applications et de leur support d'exécution afin de relever non seulement les valeurs des compteurs de performances mais aussi de conserver la trace de certains événements (entrée/sortie de procédure, changement de contexte de processus légers, réception/envoi de message, etc.)

Dans sa thèse [She01], Shende définit des techniques pour relier des observations de bas niveau aux requêtes exprimées à un niveau plus abstrait et propose une stratégie d'instrumentation multi-niveaux. Shende montre ainsi que l'analyse d'une application complexe peut nécessiter des interventions aux niveaux suivants : code source, préprocesseur, compilateur, bibliothèque, support d'exécution et machine virtuelle. Shende a validé son approche en s'appuyant sur la suite logicielle `Tau` [SMC+98] qui propose une API et un ensemble d'outils portables pour l'analyse des programmes parallèles écrits en C, C++, Fortran ou Java. Notons que `Tau` est relativement peu intrusif puisque, d'après [MS04], le coût d'un enregistrement est de l'ordre de 1400 cycles (500*ns* sur un PENTIUM IV cadencé à 2,8 GHz). Cependant, en environnement multithreadé, Shende [She01] utilise des verrous de haut niveau, ce que nous désirons absolument éviter.

Notons que l'analyse des programmes MPI a retenu une attention soutenue, sans doute parce que, outre sa popularité dans le milieu des hautes performances, MPI comporte une API de profilage standardisée (PMPI). Ainsi, depuis quelques années déjà, on dispose d'environnements de profilage MPI performants ; le lecteur trouvera dans [MCLD01] un panorama des outils de profilage MPI tels que `Tau` et `Vampir` de la société PALLAS / INTEL. Parmi ces environnements, on peut distinguer le projet KOJAK [MW03] qui propose une analyse automatique des performances des programmes MPI et OPENMP. Ainsi KOJAK intègre :

- des bibliothèques d'accès aux compteurs de performances (`PAPI` et `PCL`) ;
- un logiciel d'instrumentation de programmes et de support d'exécution (`Tau`) ;
- une bibliothèque de stockage et de manipulation de traces (`EPILOG`) ;
- un outil d'analyse automatique des performances piloté par un langage de spécification des propriétés de performance (`EXPERT`, [WM03]) ;
- un outil de visualisation, celui de l'environnement de profilage `Vampir`, par exemple.

La neutralité de l'observation et l'interprétation des performances obtenues constituent donc deux problèmes qui prennent une dimension supérieure dans le cadre des applications parallèles ou réparties. Par exemple, dans [MS04], Malony et Shende proposent et étudient une technique de compensation à la volée des erreurs d'estimation introduites par la prise de traces et d'échantillons. Ils montrent que leur technique est adaptée aux programmes séquentiels et améliore la précision des mesures des programmes parallèles. Toutefois, ils montrent que cette technique ne permet pas d'estimer l'effet intrusif de la prise d'une mesure sur un processus autre que celui qui la réalise. Enfin, l'observation fine d'une exécution d'un programme parallèle implique l'utilisation de mécanismes de synchronisation (pour la simple sauvegarde des mesures, par exemple) qui peuvent modifier radicalement le comportement parallèle du programme et, au bout du compte, dénaturer l'expérience.

6.2 Des traces noyau à un environnement de profilage multithread

6.2.1 Les Fast Kernel Traces

FKT (FAST KERNEL TRACE) [RC01] est une méthode permettant d'obtenir une trace du flot d'exécution du système d'exploitation LINUX sur machines multiprocesseurs. Cette trace peut être aussi précise que voulue par le programmeur noyau. La méthode consiste à placer des macro-commandes spéciales directement dans le code source du noyau. Ainsi, contrairement à des outils d'instrumentation dynamique comme `KernInst`, l'instrumentation du noyau ne peut se faire à la volée : la modification d'un point d'enregistrement nécessite la modification du code source et donc la recompilation du noyau et son redémarrage. Néanmoins, l'enregistrement de la trace dans un tampon peut être démarré ou interrompu à tout instant depuis l'espace utilisateur. Le mécanisme de sauvegarde de ce tampon est très optimisé[1] [Thi03].

```
#define FKT_PROBE2(KEYMASK,CODE,P1,P2)                \
  do                                                  \
      if( KEYMASK & fkt_active )                      \
         fkt_header( ((unsigned int)(CODE)),          \
            (unsigned int)(P1), (unsigned int)(P2) );\
         while(0)
```

FIG. 6.1 – Définition simplifiée d'une macro-commande FKT

Détaillons les arguments de la macro-commande présentée dans la Figure 6.1 :

– Le premier argument, `KEYMASK`, allié à la variable globale du noyau `fkt_active`, permet de valider ou non l'enregistrement de l'événement lors de l'exécution. La variable globale `fkt_active` peut être positionnée depuis l'espace utilisateur grâce à un nouvel appel système. Notons qu'il suffit d'utiliser directement la fonction `fkt_header` pour éviter un test inutile.
– Le deuxième argument, `CODE`, doit être caractéristique de l'événement à enregistrer : il servira à repérer les différentes occurrences de l'événement dans la trace collectée. Quelques événements sont prédéfinis, comme celui marquant les changements de contexte, les autres codes étant à la disposition du programmeur.
– Le programmeur a ensuite, à sa disposition, jusqu'à 5 arguments entiers supplémentaires (ici 2).

La fonction `fkt_header` enregistre l'événement en utilisant la meilleure estampille disponible puisque FKT fait appel au registre compteur de cycles (le *time-stamp counter register* du PENTIUM) qui fournit sur 64 bits le nombre de cycles écoulés depuis la mise sous tension du processeur ; cette horloge est suffisamment dimensionnée puisque sa période est supérieure à 136 années (2^{32} s) pour un processeur cadencé à 4 GHz (2^{32} Hz).

Ainsi, FKT est un système de trace noyau peu intrusif (une mesure consomme entre 80 et 230 cycles) et flexible puisque l'utilisateur a la capacité de placer et de prendre ses mesures où et quand il le désire. Il devra être étendu pour atteindre notre objectif.

[1]Ce tampon est en effet confondu avec la projection en mémoire du fichier associé (son *buffer-cache*) et les pages de ce dernier, gérées comme un buffer tournant, sont automatiquement sauvegardées et évincées par le gestionnaire des pages (la *TLB*), évitant ainsi toute recopie et même tout gaspillage mémoire.

Nous reviendrons plus en détail sur FKT par la suite, cependant le lecteur pourra se référer à [Rus02a] pour la description et l'analyse complètes de ce système.

6.2.2 Les contraintes issues de l'ordonnancement à deux niveaux

Afin de tracer précisément le comportement d'un programme multithreadé, nous devons être capables de déterminer, à tout instant, quel processus léger s'exécute sur quel processeur physique. Cette simple requête mène aux constats suivants :

Les informations du noyau seules ne suffisent pas. Le noyau, n'ayant pas connaissance des processus légers utilisateur des applications utilisant une bibliothèque de processus légers à deux niveaux, est dans l'incapacité de donner leur ordonnancement au sein des LWP.

Les informations de l'application seules ne suffisent pas. Supposons qu'une exécution d'un programme utilise quelques processus légers utilisateur s'exécutant au sein de trois LWP sur une machine biprocesseur. S'il est possible de déterminer *sans information du noyau* quel LWP exécute quel processus léger utilisateur[2], on ne peut cependant pas déterminer la date des changements de contexte du LWP. Plus problématique encore, sans information du noyau, on ne peut pas connaître les dates de préemption des trois LWP par le système d'exploitation. Autrement dit, la préemption d'un processus léger utilisateur due à la préemption du LWP sous-jacent ne laisse pas de trace dans l'espace utilisateur ; dans ces conditions, il est généralement impossible d'obtenir des mesures exactes sur un processus léger donné, comme par exemple, son temps d'occupation des CPU.

La communication espace noyau / espace utilisateur est trop coûteuse. Une première solution serait de créer de nouveaux appels systèmes pour, par exemple, connaître le processeur exécutant l'enregistrement de l'événement ou encore pour que l'application informe le noyau de ses changements de contexte utilisateur. Mais cette solution est trop intrusive : outre le temps d'exécution prohibitif d'un appel système (cf. les micro benchmarks de la section 7.3.3), cette solution aurait également pour effet de fournir à l'ordonnanceur l'occasion de prendre des décisions d'ordonnancement bien plus fréquemment que lors d'une exécution non instrumentée. Une seconde solution serait l'utilisation par le noyau d'*upcalls*, c'est-à-dire que le noyau force l'application à appeler une fonction, comme le fait le mécanisme des signaux, pour transmettre ses événements à l'application. Toutefois, cette seconde solution est coûteuse car elle nécessite la sauvegarde de l'ensemble des registres.

6.2.3 Notre proposition

La solution retenue est de laisser FKT générer des événements relatifs au noyau et de développer un système similaire, appelé FAST USER TRACE (FUT), pour la génération des traces de l'application. Les deux points clés de cette solution sont :
– les traces noyau et applicatives sont nécessaires à l'obtention d'une analyse complète de l'exécution de l'application ;
– les traces noyau et applicatives partagent la même référence de temps afin de pouvoir reconstruire l'ordonnancement complet des processus légers de l'application sur la machine.

[2] Il suffit de connaître le premier processus léger utilisateur sur chaque LWP puis de suivre les changements de contexte utilisateur.

Une fois les deux traces fusionnées en une unique *supertrace*, on dispose pour chaque événement, en plus de son code, de ses paramètres, de son estampille et de l'identité du processeur, de celle du LWP et celle du processus léger utilisateur à l'origine de cet événement. En conciliant ainsi les points de vue noyau et utilisateur, cet environnement de profilage permet non seulement de reconstituer le déroulement d'une application multithreadée complexe mais aussi de replacer ce déroulement dans son contexte d'exécution, les événements majeurs du noyau pouvant être aussi tracés et analysés.

Sur le plan pratique, FUT est très similaire à FKT à la fois dans son principe de fonctionnement et dans son utilisation. Seuls quelques champs de la trace enregistrée ont été adaptés pour une utilisation en espace utilisateur ; une description du format des traces est détaillée dans l'annexe B.

6.2.4 La problématique de la synchronisation

Notre mécanisme de prise de traces nécessite l'enregistrement des informations dans un tampon, que ce soit dans le noyau (FKT) ou en espace utilisateur (FUT). Dans le noyau, il peut être envisageable d'avoir un tampon par processeur : il y a généralement un nombre limité de processeurs. En revanche, en espace utilisateur, utiliser un tampon par processus léger conduirait à consommer trop de mémoire lorsque l'application utilise de nombreux processus légers (plusieurs milliers pour certaines applications). Il est alors nécessaire de partager un tampon entre plusieurs processus légers. Cet accès partagé doit être le plus efficace possible pour éviter de perturber ou de ralentir l'application observée.

L'utilisation de mécanismes d'exclusion mutuelle de haut niveau (verrous, sémaphores etc.) est proscrite pour plusieurs raisons :

1. ces fonctions sont trop coûteuses ;
2. elles introduisent de nouveaux points d'ordonnancement dans l'application qui seraient absents sans le mécanisme de traces. Or, on ne veut pas modifier l'ordonnancement du programme observé ;
3. elles ne sont pas utilisables partout, en particulier dans les sections critiques ou dans les événements asynchrones comme les traitants de signal. Des parties de l'application seraient alors inobservables avec nos mécanismes.

Une solution aurait été d'utiliser nos mécanismes de synchronisation interne, comme les sections critiques présentées précédemment (cf. section 5.1.1 page 88). Cependant, l'objection 3 s'appliquerait en partie : il serait par exemple difficile d'observer nos mécanismes de synchronisation interne si l'on se servait d'eux pour prendre nos traces. C'est pourquoi nous nous sommes orientés vers une synchronisation légère, adaptée à nos besoins, mais indépendante de tous nos autres travaux.

En collaboration avec R. Russell, nous avons résolu ce problème en utilisant l'instruction machine atomique `cmpxchgl` (décrite par la Figure 6.2). Après avoir obtenu l'adresse du premier emplacement libre dans le tampon de la trace, l'adresse du prochain emplacement libre est calculée en fonction de la taille de l'événement à écrire ; cette nouvelle valeur remplace de manière atomique l'ancienne adresse. La procédure est itérée en cas d'échec, c'est-à-dire si un autre processus léger a réservé l'emplacement pour écrire une trace pendant notre calcul de la nouvelle adresse.

Par conséquent, les événements enregistrés dans la trace peuvent être temporellement désordonnés : un processus léger peut récolter un événement, mais se «faire doubler» au

```
int cmpxchgl(int* ref, int oldvalue, int newvalue) {
        if (*ref != oldvalue)
                return 0;
        *ref=newvalue;
        return 1;
}
```

L'instruction processeur `cmpxchgl` *exécute ce code de manière atomique*

FIG. 6.2 – *Pseudo code de l'instruction* `cmpxchgl`

moment de l'enregistrer. Charge est laissée à l'utilitaire `supertrace` de réordonner temporellement les événements lors de la fusion des traces noyau et utilisateur.

Ce mécanisme simple de synchronisation assure une cohérence dans les enregistrements du tampon. Complètement indépendante de tous les autres mécanismes de synchronisation de notre bibliothèque et de l'application, cette technique permet de récolter des traces à n'importe quel endroit du code du programme. Elle simplifie également une éventuelle réutilisation de ces travaux pour tracer d'autres bibliothèques de processus légers comme NPTL.

6.3 Description de notre environnement

Après avoir présenté notre plate-forme d'implémentation, nous détaillons l'interface d'instrumentation des applications puis nous nous intéresserons au traitement des traces obtenues.

6.3.1 Plate-forme d'implémentation

Nous avons implémenté notre environnement de profilage sur l'architecture INTEL x86 à la fois dans le noyau LINUX et dans MARCEL, la bibliothèque de processus légers purement utilisateur ou à deux niveaux de l'environnement PM^2.

Pour avoir le support de FKT, on instrumente le noyau en un ensemble de points en lui appliquant un *patch*. Ainsi, des événements sont mémorisés lors des changements de contexte, lors des entrées et sorties d'interruption matérielle (IRQ, etc.) ou logicielle (appels systèmes, etc.).

L'instrumentation de la bibliothèque de processus légers MARCEL permet de mémoriser les événements nécessaires à la reconstruction de l'ordonnancement global (changement de contexte des processus légers utilisateur, création et terminaison des LWP). Cette instrumentation permet aussi l'analyse de toutes les fonctions de la bibliothèque MARCEL. De cette façon, on peut, par exemple, tracer précisément les performances de la bibliothèque et/ou voir les causes de préemption des processus légers (temps écoulé, attente de verrou, etc.) L'instrumentation de la bibliothèque MARCEL n'a pas demandé beaucoup de travail et pourrait être très facilement reproduite dans toute autre bibliothèque de processus légers.

6.3.2 Outils d'instrumentation

Nous avons repris l'interface de FKT. Ainsi, l'enregistrement d'événements repose principalement sur l'utilisation des macros `FUT_PROBE`x`()` et quelques types d'événements sont prédéfinis, comme ceux marquant le début ou la fin des processus légers.

Notons aussi que nous avons développé un outil de prétraitement. Celui-ci attribue automatiquement un code d'entrée et un code sortie à chaque fonction. Pour instrumenter un programme, il suffit alors d'insérer les macros `PROF_IN()` et `PROF_OUT()` au début et à la fin des fonctions à profiler. L'insertion de ces macros peut être faite manuellement ou automatiquement : le compilateur `gcc` dispose de fonctionnalités d'instrumentation des fonctions compilées et ce, au moyen de deux appels de fonctions supplémentaires.

6.3.3 Analyse des traces

Une fois les traces utilisateur et noyau générées, celles-ci sont fusionnées dans une *supertrace* au moyen de l'outil `supertrace`. Dans cette supertrace, les événements sont interclassés temporellement et chaque événement est relié au processeur, au LWP et au processus léger utilisateur qui l'ont enregistré. Toutefois, certains événements, comme ceux enregistrés dans les routines d'interruptions, ne peuvent être associés à un processus léger utilisateur.

L'analyse de cette supertrace peut être menée par un utilitaire en ligne de commande, appelé `sigmund`. Il permet d'appliquer à la supertrace un ou plusieurs filtres. On peut, par exemple, extraire les événements concernant un processeur, un LWP ou un processus léger utilisateur ; on peut aussi sélectionner les événements produits entre deux dates ou durant l'exécution d'une fonction spécifiée, etc.

Des informations telles que la mesure du temps actif, du temps d'exécution réel des processus légers ou des fonctions sélectionnées peuvent être extraites des événements ainsi sélectionnés. Il est alors possible de profiler précisément les fonctions d'un processus multithreadé, même avec une charge importante sur le système.

6.3.4 Exemples

```
int fonction_instrumentee(int foo, int bar) {
  int res;

  PROF_IN();
  if (foo) {
    FUT_PROBE2(code_perso, foo, bar);
    ...
    PROF_OUT();
    return res;
  }
  ...
  PROF_OUT();
  return 0;
}
```

FIG. 6.3 – Exemple d'instrumentation de code

```
$> sigmund --trace-file supertrace.log --thread 15 \
   --event CONTEXT_SWITCH --list-events
type  date_tick  pid cpu thr   code name                 param(s)
[...]
USER  97615576  7137   1   7  23014 USER_CONTEXT_SWITCH  15
USER  97757052  7137   1  15  23014 USER_CONTEXT_SWITCH  8
USER  98006248  7136   0   6  23014 USER_CONTEXT_SWITCH  15
KERN  98139183  7136   0  15  23014 KERN_CONTEXT_SWITCH  6152
KERN  98638163  2352   2   ?  23014 KERN_CONTEXT_SWITCH  7136
USER  99060185  7136   2  15  23014 USER_CONTEXT_SWITCH  7
[...]
$> sigmund --trace-file supertrace.log --thread 15 --active-time
Temps actif total = 130193845 cycles
```

FIG. 6.4 – Interrogation de la supertrace avec sigmund

La Figure 6.3 montre comment instrumenter une fonction dans le code de l'application à l'aide des macros `PROF_IN()`/`PROF_OUT()` pour que la fonction génère un événement à chaque entrée et à chaque sortie.

La macro `FUT_PROBE2(code_perso, foo, bar)` permet de générer un événement dans la trace dont le code sera `code_perso`. Cet événement aura deux arguments qui lui seront associés (`foo` et `bar`).

La Figure 6.4 montre deux exemples d'informations fournies par notre environnement. Le programme instrumenté est un programme multithreadé exécuté sur une machine biprocesseur XEON SMT (soit 4 processeurs logiques). La bibliothèque de processus légers utilisée est une bibliothèque à 2 niveaux utilisant 4 LWP noyau pour propulser ses processus légers utilisateur. On essaie d'obtenir des informations concernant l'ordonnancement d'un processus léger particulier (le numéro 15 sur cet exemple).

La première requête sert à connaître les événements de la trace relatifs aux changements de contexte concernant le processus léger sélectionné. On peut voir ici quelles sont les informations enregistrées dans la supertrace pour chaque événement, à savoir :

type	le type d'événement (noyau ou utilisateur) ;
date_tick	sa date ;
`pid`	le LWP s'exécutant à cet instant ;
`cpu`	le processeur qui a enregistré la trace ;
`thr`	le processus léger utilisateur s'exécutant à cet instant (s'il est profilé) ;
`code`	le code de l'événement ;
`name`	le nom de l'événement ;
`param(s)`	les paramètres éventuels associés à l'événement.

Sur cet exemple, on voit que le processus léger utilisateur 15 a été ordonnancé par le LWP 7137 sur le processeur 1. Puis, il a été ordonnancé par un autre LWP (le 7136) sur le processeur 0. Ce LWP a été préempté par le noyau (qui a donné la main à une autre application) pour être réordonnancé sur le processeur 2 un peu plus tard. Le processus léger utilisateur a ensuite passé la main.

Grâce à ce type d'analyse, on peut voir quelle a été l'affinité d'un processus léger donné avec les processeurs physiques. Sur cet exemple, on voit que le processus léger 15 n'a pas eu d'affinité marquée avec un LWP particulier et qu'un LWP lui-même a été déplacé par le système.

La seconde requête permet de connaître le temps CPU total consommé par un processus léger utilisateur qui a été ordonnancé par différents LWP sur différents processeurs.

6.3.5 Vers une présentation graphique de la trace

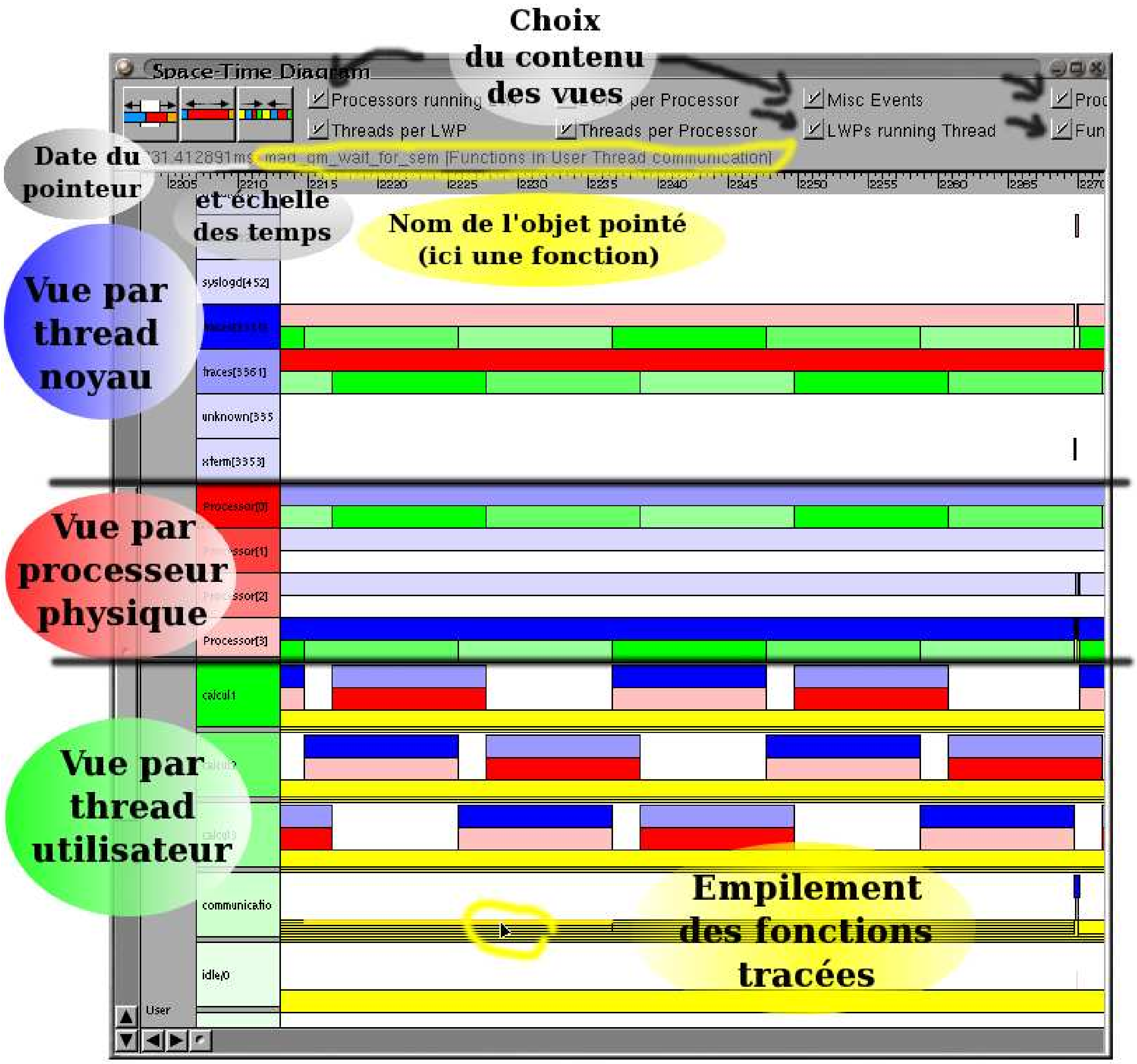


FIG. 6.5 – Visualisation d'une trace avec PAJÉ

Une fois la supertrace calculée, nous disposons de l'ordonnancement effectif des processus légers de l'application sur la machine ainsi que des événements enregistrés qui leur sont associés. Le programme `sigmund` permet alors d'interroger cette supertrace et d'en extraire les informations désirées. Cependant, une analyse globale du comportement du programme est humainement difficile à mener en mode texte ; une représentation graphique peut s'avérer mieux adaptée.

Plutôt que de développer notre propre outil de présentation graphique, nous avons préféré utiliser les travaux effectués dans ce domaine. Notre choix s'est porté vers PAJÉ [dKdOS00] qui est un outil de visualisation interactif présentant le flot d'exécution d'applications parallèles où un nombre, potentiellement grand, de processus légers s'exécutent concurremment[3]. PAJÉ est facile à interfacer car il a été conçu comme un outil générique : la structure des flots d'exécution à représenter n'est pas définie dans le logiciel mais elle est décrite dans le fichier de données lui-même. Ainsi l'écriture d'un filtre a suffi pour la conversion de la supertrace au format accepté par le logiciel PAJÉ.

Sur la Figure 6.5, on peut observer un extrait du déroulement d'un programme multithreadé. Sur cet exemple, la machine a quatre processeurs physiques (il s'agit d'un biprocesseur XEON SMT). L'application observée (nommée `traces`) utilise deux LWP pour ordonnancer ses quatre processus légers utilisateur : trois sont destinés aux calculs d'un produit de matrice (nommés `calcul1`, `calcul2` et `calcul3`) et le quatrième assure des communications. Les informations sont représentées dans trois zones matérialisant les points de vue processus léger utilisateur, LWP et processeur[4]. Détaillons ces différentes vues :

Informations sur chaque LWP :

1. identité du processeur physique utilisé (lorsque ce LWP est ordonnancé par le système d'exploitation) ;
2. identité du processus léger utilisateur qu'il ordonnance (si le LWP exécute l'application observée) ;

Informations sur chaque processeur :

1. identité du LWP exécuté ;
2. identité du processus léger utilisateur exécuté (s'il s'agit de l'exécution de l'application observée) ;

Informations sur chaque processus léger utilisateur :

1. identité du LWP propulseur (lorsqu'il est sélectionné par l'ordonnanceur de l'application) ;
2. identité du processeur physique lorsqu'il est réellement exécuté (il doit alors être aussi ordonnancé par un LWP) ;
3. emboîtement des appels de fonction et des événements tracés dans l'application.

Il est facile d'observer sur la représentation graphique que le système d'exploitation laisse bien la main aux deux LWP de l'application ; ceux-ci s'exécutent de façon continue, chacun sur un processeur de la machine (les processeurs 0 et 3). En revanche, pour les processus légers utilisateur, on observe que le processus léger de communication est rarement pris en charge par un LWP : il est en fait bloqué sur un sémaphore (information révélée en

[3] PAJÉ a été initialement conçu pour suivre l'état des processus légers de niveau utilisateur de l'environnement de programmation distribué ATHAPASCAN [BIG97].

[4] Si tous les LWP et tous les processeurs du système sont représentés, seuls les processus légers utilisateur de l'application espionnée sont observables.

positionnant la souris sur la dernière fonction appelée par ce processus léger). Le graphique montre tout aussi clairement que les processus légers utilisateur de calcul passent d'un LWP à l'autre à chaque quantum. Cela induit un va-et-vient d'un processeur à l'autre, ce qui peut être préjudiciable à l'application quant à l'utilisation du cache processeur par les processus légers utilisateur.

Sans ces traces à deux niveaux, il aurait plus été difficile de se rendre compte de la mauvaise utilisation du cache par les processus légers utilisateur. On « voit » ainsi que l'ordonnanceur des processus légers de niveau utilisateur ne tient pas compte des affinités processus légers utilisateur/LWP. Si l'on peut facilement améliorer l'exécution de ce programme[5], le problème de fond, lui, est non trivial : il s'agit d'améliorer le dialogue entre la machine et l'application.

6.4 Conclusion

Les ordonnanceurs multiniveaux, comme celui que l'on a développé, permettent de tirer pleinement partie des architectures de type SMP, deux processus légers d'un même processus pouvant s'exécuter réellement en parallèle, tout en conservant une grande efficacité sur le plan de la gestion des processus légers (création, terminaison, etc.) Ce progrès complexifie notablement le recueil et l'analyse des performances des applications multithreadées, analyse cruciale dans un contexte hautes performances. Les logiciels que nous avons développés et présentés dans ce chapitre constituent une réponse à ce besoin : les outils proposés permettent d'instrumenter rapidement et précisément l'application cible afin d'en analyser finement l'exécution.

Pour bénéficier de cet environnement de profilage, il est nécessaire d'utiliser un système d'exploitation et une bibliothèque de processus légers adaptés. Notons que les modifications nécessaires pour supporter notre environnement de profilage sont très localisées ; les bibliothèques de processus légers comme NGPT [ADH+02], NPTL [DM03] ou LINUXTHREAD [Ler96] peuvent donc être adaptées sans difficultés majeures.

Dans l'état actuel, notre environnement reconstruit l'ordonnancement des processus légers et peut enregistrer les appels de fonction ainsi que des événements définis par l'utilisateur. Il serait intéressant de généraliser et diversifier les mesures. D'une part, on pourrait utiliser les compteurs matériels de performances et ce, pourquoi pas, au travers d'une API telle que PAPI. D'autre part, des types d'événements caractéristiques relatifs au multithreading pourraient être récoltés et exploités. Par exemple, il pourrait être utile de suivre l'état d'un verrou pour savoir quand et combien de processus légers cherchent à l'acquérir. Un goulot d'étranglement autour d'un tel verrou serait alors facile à mettre en évidence.

[5] Il suffit de faire coopérer non pas trois, mais deux ou quatre processus légers au calcul du résultat.

Chapitre 7

Évaluation

Sommaire

L'ensemble des concepts, outils et mécanismes décrits dans les chapitres précédents ont fait l'objet d'une implémentation au sein de la plate-forme de développement pour applications parallèles distribuées PM^2. Plus précisément, la bibliothèque de processus légers MARCEL a été complètement réécrite afin de vérifier expérimentalement les nouveaux concepts et les approches proposées.

7.1 Que tester, que mesurer ?

Lorsque l'on désire comparer des bibliothèques de communication, les débits atteints et les temps de latence sont les principaux paramètres évalués. La communauté a reconnu la pertinence de ces paramètres assez simples à recueillir[1]. Il existe également des jeux de benchmarks reconnus pour certaines bibliothèques comme, par exemple, MPI. Il s'agit alors de mesurer la performance globale de la bibliothèque dans le contexte d'une application test censée reproduire des comportements typiques.

Dans le cas de bibliothèques de processus légers, il n'existe pas (encore) de benchmarks complets reconnus par la communauté. Cela est probablement dû en partie à l'extrême variabilité des fonctionnalités offertes par les bibliothèques de processus légers : préemption ou non, niveau d'ordonnancement, gestion des appels systèmes bloquants, etc. C'est pourquoi nous avons créé un ensemble de programmes permettant de tester et comparer divers aspects des bibliothèques de processus légers qui nous paraissaient importants.

7.1.1 Description de la plate-forme matérielle

Pour comparer de manière équitable un ensemble de bibliothèques, il convient de fixer le maximum de paramètres. En l'occurrence, nous avons choisi l'architecture `x86` et le système LINUX. Ce choix permet, d'une part, de bénéficier d'une vaste gamme de bibliothèques différentes à tester, et d'autre part, de pouvoir comparer l'ensemble de nos développements puisque les activations n'ont été implémentées, pour l'instant, que sur ce système.

Les machines utilisées correspondent à deux configurations classiques dans les grappes pour calcul hautes performances. Nous effectuerons d'abord les tests sur une machine monoprocesseur contenant un processeur Intel® Pentium® 4 cadencé à 1.8 GHz avec 512 kilo-octets de mémoire cache et 512 Méga-octets de mémoire vive. Cela correspond à une architecture matérielle très simple au niveau des flots d'exécution : un seul est disponible à tout instant. La seconde machine type est un biprocesseur SMT contenant deux processeurs Intel® Xeon™ cadencés à 2.66 GHz avec 512 kilo octets de mémoire cache et 1 giga octet de mémoire vive. L'architecture matérielle propose donc ici quatre flots d'exécution parallèles.

7.1.2 Fonctionnalités évaluées

7.1.2.1 Opérations représentatives des bibliothèques de processus légers

Dans un premier temps, nous nous intéressons à un ensemble d'opérations correspondant aux principales actions demandées à une bibliothèque de processus légers, c'est-à-dire ce qui concerne la création et la synchronisation des flots d'exécution concurrents.

Changement de contexte : Il s'agit de mesurer le temps nécessaire pour passer la main volontairement d'un processus léger à un autre. Le résultat est obtenu en lançant deux processus légers qui se passent la main un certain nombre de fois. Il faut s'assurer (par exemple avec de l'attente active) que les deux processus légers sont prêts avant de démarrer le test (pour ne pas compter le temps de création des processus légers).

Création de processus légers : Nous évaluons le coût de création d'un processus léger. Comme des bibliothèques retardent certains coûts relatifs à la création jusqu'au moment du premier ordonnancement du processus léger créé, nous mesurons non pas le

[1]Encore que leur définition peut varier. Par exemple, pour le débit, il peut s'agir soit du demi-temps d'un aller retour, soit du temps d'une seule communication à sens unique (en pratique, en fait alors un grand nombre de communication à sens unique puis une très courte dans l'autre sens pour arrêter le chronomètre). Suivant les bibliothèques, les résultats ne sont pas nécessairement identiques.

temps d'exécution de la fonction `create()`, mais le temps entre l'appel de cette fonction et le début de l'exécution du processus léger. Nous laissons ce dernier se terminer immédiatement et nous mesurons également le temps de sa terminaison (le processus léger principal effectue un `join()` juste après avoir créé le processus léger.

Primitives de synchronisation bloquantes : Nous mesurons le temps nécessaire pour bloquer un processus léger et de passer la main au suivant. Ce test est très semblable au précédent, mais il oblige l'ordonnanceur à manipuler les listes de processus légers prêts et bloqués. On obtient ces mesures, là encore, en lançant deux processeurs légers qui se bloquent et se débloquent réciproquement avec des sémaphores.

Primitives de synchronisation non bloquantes : Il s'agit ici de mesurer le temps nécessaire pour acquérir un verrou lorsqu'il n'y a pas de contention dessus.

7.1.2.2 Programmes synthétiques

Après avoir mesuré des opérations isolées, nous essayons à travers quelques programmes de créer des situations permettant d'évaluer des qualités (ou des défauts) des bibliothèques.

Utilisation des ressources matérielles : Nous désirons ici observer le comportement de la bibliothèque en présence d'une application ayant de nombreux travaux indépendants à effectuer. Dans ce test, nous calculons un produit de matrice en parallèle. Le test est effectué avec un nombre variable de processus légers. La taille globale du calcul reste en revanche inchangée. Une bibliothèque exploitant des machines multiprocesseurs pourra s'attendre à une accélération linéaire sur cet exemple. L'application est nommée `prodmat`.

Tâches parallèles dépendantes, synchronisation interne : Ce programme permet d'utiliser de façon intensive les fonctionnalités des bibliothèques de processus légers. En effet, il utilise de façon intensive les primitives de synchronisation entre processus légers en calculant une somme par la méthode *diviser pour régner* (le programme génère un arbre de processus légers). Ce programme permet également d'évaluer le nombre de processus légers que peut créer la bibliothèque. C'est une bonne indication des ressources nécessaires aux processus légers. Ce programme est nommé `sumtime`.

7.1.3 Bibliothèques évaluées

Tous les tests précédemment décrits ont été exécutés sur une grande variété de bibliothèques citées ci-après.

Bibliothèques disponibles sur la plate-forme. Quatre bibliothèques de processus légers de LINUX sont évaluées, à savoir :

LINUXTHREAD : l'ancienne bibliothèque de référence de LINUX ;

NPTL : la nouvelle bibliothèque standard de LINUX utilisant les nouveaux services du noyau ;

GNUPTH : la bibliothèque du projet GNU ;

NGPT : la bibliothèque développée momentanément par IBM pour remplacer LINUXTHREAD et abandonnée suite à l'apparition de NPTL.

Une description plus complète de ces bibliothèques est disponible dans la partie 3.4. Nous nous bornerons à remarquer ici que tous les types de bibliothèques sont présents : noyau (2), utilisateur et mixte.

Parallélisation sans processus légers. Outre les bibliothèques de processus légers, certaines mesures sont pertinentes rapportées aux processus lourds. C'est le cas pour les changements de contexte ainsi que pour les temps de création. Dans ces deux cas, des mesures ont été effectuées en utilisant des processus lourds (`fork()`) ainsi que des flots indépendants au sein du processus avec la primitive `clone()` [2]

La bibliothèque MARCEL. Pour notre bibliothèque MARCEL, nous avons testé deux versions :

`marcel-mono`: bibliothèque de niveau purement utilisateur ;

`marcel-smp`: bibliothèque à deux niveaux ;

Nous avons construit ici des versions optimisées (pas de génération de traces ou de messages de déverminage), ce qui correspond aux choix des utilisateurs de notre environnement lors de leurs développements.

7.2 Résultat des expériences

7.2.1 Opérations de base sur les processus légers

Bibliothèque	Changements de contexte	Création seule	Création avec terminaison	Synchronisation bloquante (sémaphore)	Synchronisation non bloquante (mutex)
UNIX (`fork()`)	4,5	-	154	-	-
`clone()`	2,9	23	48	-	-
LINUXTHREAD	3,0	84	125	12,1	0,25
NPTL	2,9	6,6	17	5,4	0,19
GNUPTH	5,1	81	113	NA[3]	0,09
NGPT	3,3	90	127	1,1	0,40
`marcel-mono`	0,39	1,8	4,3	0,61	0,035
`marcel-smp`	1,1	3,6	9,5	0,79	0,46

Les données sont exprimées en μs. Ces tests ne concernant que des opérations de base, ils ont été réalisés sur la machine monoprocesseur (les résultats offrent le même profil sur la machine multiprocesseur).

TAB. 7.1 – *Coût des opérations de base des processus légers*

Le tableau 7.1 présente les temps d'exécution de diverses fonctions de base des bibliothèques de processus légers. Les résultats obtenus sont cohérents entre eux. Plusieurs remarques peuvent être faites :

[2] Il s'agit de la primitive système utilisée par les bibliothèques noyau ou mixte pour obtenir des flots parallèles au sein d'un exécutable.

[3] GNUPTH ne propose pas d'implémentation des sémaphores.

– notre bibliothèque offre d'excellentes performances tant en mode utilisateur que mixte. Le surcoût entre les deux versions s'explique par les synchronisations internes plus coûteuses. Dans le cas de la version `marcel-smp`, les mesures obtenues pour la synchronisation non bloquante sont décevantes, sans justification *a priori*. Il s'agit probablement d'un défaut dans l'implémentation qui a échappé pour l'instant à nos investigations. Nous espérons le corriger rapidement ;
– les bibliothèques utilisateur et mixte GNUPTH et NGPT ont des performances nettement inférieures aux nôtres. Cela est dû à la conformité stricte à la norme POSIX de ces bibliothèques qui exécutent fréquemment des appels systèmes pour bloquer les signaux. On se rend compte à cette occasion que ces surcoûts annulent pour certaines opérations le bénéfice espéré en restant en espace utilisateur par rapport aux bibliothèques noyau. Ainsi, un changement de contexte est 1.7 fois plus coûteux pour GNUPTH par rapport à LINUXTHREAD bien que la première soit une bibliothèque de niveau utilisateur et la seconde de niveau noyau. Les appels à `setprocmask()` de GNUPTH diminuent fortement ses performances ;
– les deux bibliothèques noyau ont des performances vraiment distinctes. L'ancienne bibliothèque LINUXTHREAD paye le coût d'une synchronisation par échange de signaux. La nouvelle bibliothèque NPTL utilise tous les nouveaux supports du noyau LINUX. On voit bien ici tout l'intérêt d'un support adéquat dans le noyau vis-à-vis des performances.

7.2.2 Applications synthétiques

	Nombre de processus légers							
Bibliothèque	1	2	3	4	5	6	7	8
LINUXTHREAD	10,4	5,1	3,4	2,3	3,0	3,1	2,6	2,3
NPTL	10,5	5,2	3,5	2,3	3,0	3,2	2,7	2,5
GNUPTH	10,5	10,5	10,5	10,5	10,5	10,5	10,5	10,5
NGPT	10,9	5,5	3,4	2,6	4,0	3,1	2,8	2,3
`marcel-mono`	10,4	10,4	10,4	10,4	10,4	10,4	10,4	10,4
`marcel-smp`	10,8	5,3	3,6	2,3	2,4	2,4	2,4	2,4

Les données sont exprimées en secondes.
Ces tests ont été réalisés sur la machine multiprocesseur (les résultats sont constants sur la machine monoprocesseur).

TAB. 7.2 – *Passage à l'échelle d'une application régulière sur machine multiprocesseur*

Le tableau 7.2 présente les temps d'exécution d'un calcul de produit de matrice : le domaine de calcul est divisé en un nombre prédéterminé de sous-domaines, chacun pris en charge par un processus léger. Plusieurs commentaires peuvent être faits :
– comme prévisible, les bibliothèques de niveau utilisateur sont incapables de bénéficier d'une accélération sur une machine multiprocesseur ;
– les bibliothèques de niveau noyau ont un meilleur comportement lorsqu'il y a le même nombre de processus légers que de processeurs, ou à défaut lorsque ce nombre en est un multiple. Cela simplifie probablement l'équilibrage de charge du système ;
– ce n'est pas visible dans le tableau, mais toutes ces mesures sont très stables, à l'exception de celles de la bibliothèque NGPT. Cela provient du fait que, contrairement

à notre bibliothèque `marcel-smp`, les quatre LWP créés par NGPT ne sont pas fixés chacun sur un processeur physique. En cas de mauvais ordonnancement du système (plusieurs LWP en alternance sur le même processeur physique), les performances se dégradent très vite, atteignant même les temps des bibliothèques utilisateur dans les pires cas.

	Borne de la somme à calculer				
Bibliothèque	100	500	1 000	10 000	100 000
LINUXTHREAD	15,1	406	1570	OutOfMem	OutOfMem
NPTL	7,5	BUG	BUG	BUG	BUG
GNUPTH	44,5	800	3700	OutOfMem	OutOfMem
NGPT	13,4	33,5	64,8	705	7486
`marcel-mono`	1,9	10,3	20,9	155	OutOfMem
`marcel-smp`	2,1	13,6	26,8	236	OutOfMem

OutOfMem : pas assez de mémoire

BUG : erreur interne à la bibliothèque

Les données sont exprimées en millisecondes.
Ces tests ont été réalisés sur la machine multiprocesseur.

TAB. 7.3 – *`sumtime` : un programme de stress pour les bibliothèques*

Le programme `sumtime` permet d'aller aux limites[4] des bibliothèques de processus légers non seulement parce qu'il utilise énormément de synchronisation en permanence, mais aussi parce qu'il crée un très grand nombre de processus légers. Les mesures présentées dans le tableau 7.3 démontrent encore l'intérêt des bibliothèques de processus légers utilisateur : les versions avec MARCEL sont beaucoup plus rapides.

7.3 Coûts et gains des nouveaux services offerts

7.3.1 Le modèle des *Scheduler Activations*

Les *upcalls*. Comme expliqué à la section 5.2.1, l'exécution d'un *upcall* après un appel système ne nécessite la sauvegarde que de quelques registres sur la pile et non pas l'état complet du processeur. De plus, une fois le code de l'*upcall* exécuté, on retourne directement dans l'application sans passer par le mode noyau. Nous souhaitons mettre en évidence le gain que ces deux optimisations apportent par rapport au déroulement d'un signal classique.

Obtenir uniquement le temps du déroutement n'est pas chose aisée. C'est pourquoi nous mesurons un chemin d'exécution connu sur lequel nous rajoutons, si nous le désirons, un *upcall* ou un signal. L'important n'est donc pas la valeur absolue, mais plutôt la différence avec la situation de référence. Dans le tableau 7.4, on peut comparer les temps d'exécution lorsque le noyau préempte l'application en exécutant un *upcall* ou un traitant de signal et lorsqu'il n'exécute que son chemin normal. Ces mesures ont été prises sur la machine monoprocesseur.

[4]Ce programme semble même révéler un problème dans la bibliothèque NPTL qui fera l'objet d'un rapport de bug auprès de ses développeurs.

On remarquera que les surcoûts aussi bien des *upcalls* que des traitants de signaux sont assez faibles. Le gain entre un *upcall* et un traitant de signal s'explique par la moindre quantité de données à sauver et la non nécessité d'un appel système pour restaurer le contexte d'exécution.

	Durée	Surcoût
Retour standard (référence)	$13,1\,\mu$s	$0\ (+0\,\%)$
Upcall	$14,1\,\mu$s	$+1,0\,\mu$s $(+7,63\,\%)$
Traitant de signal	$16,5\,\mu$s	$+3,4\,\mu$s $(+25,95\,\%)$

TAB. 7.4 – Surcoût des *upcalls*

`Surcoût des upcalls`	$1,0\,\mu$s
`getpid()`	$0,92\,\mu$s
`act_cntl(RESTART_UNBLOCKED, id)`	$2,17\,\mu$s

TAB. 7.5 – Coûts globaux du redémarrage d'une activation

Redémarrage d'une activation disponible. En présentant notre modèle à la section 4.4, nous avons expliqué qu'un appel système est nécessaire pour relancer une activation prête à repartir. Cet appel système (`act_cntl(ACT_CNTL_RESTART_UNBLOCKED, id)`) supprime l'activation courante et redémarre l'activation prête sur le LWP local. Dans le tableau 7.5, on peut comparer le temps d'un simple appel système (`getpid()` au temps nécessaire pour redémarrer une activation. Pour ce faire, nous mesurons le temps depuis le début de l'appel système jusqu'au redémarrage effectif de l'activation prête ; ce temps inclut celui de l'appel système et celui du surcoût de l'*upcall* au redémarrage de l'activation. Par conséquent, le surcoût dû à l'arrêt de la précédente activation et le basculement vers l'activation réveillée n'est que de $0.25\,\mu$s.

7.3.2 Le serveur d'événements

7.3.2.1 Mesure de la réactivité

Il s'agit ici d'évaluer la capacité d'une application à réagir rapidement lorsqu'un message réseau survient, quelle que soit sa charge de calcul. Pour ce faire, un programme synthétique lance un certain nombre de processus légers exécutant une tâche de calcul quelconque pendant qu'un autre processus léger attend des messages et renvoie ces derniers sitôt réceptionnés. Nous mesurons le temps nécessaire à ce programme pour renvoyer en écho un message réseau reçu en fonction du nombre de processus légers effectuant du calcul (voir Table 7.6).

En désactivant le serveur d'événements de notre bibliothèque, sans support de la part de l'ordonnanceur, le processus léger à l'écoute du réseau teste si un événement est survenu chaque fois qu'il est ordonnancé. Si aucun événement n'est survenu, alors il repasse la main immédiatement. Si n processus légers sont en cours d'exécution, un événement réseau peut rester non détecté pendant une durée pouvant aller jusqu'à n quanta de temps. Le quantum de la bibliothèque est classiquement de 10 ms, donc $10*n/2$ ms sont nécessaires en moyenne pour réagir comme le montre la première ligne de la Table 7.6.

	Nombre de processus légers de calcul				
Version de l'ordonnanceur	Aucun	1	2	5	10
Sans support particulier (ms)	0, 130	5, 011	10, 022	25, 010	50, 010
Avec serveur d'événements (ms)	0, 134	4, 846	4, 837	5, 471	4, 848
+ activations (ms)	0, 451	0, 453	0, 452	0, 457	0, 453

TAB. 7.6 – Temps de réaction à une requête d'entrée/sortie en fonction du nombre de processus légers de calcul.

Avec le serveur d'événements, le processus léger réseau délègue sa scrutation à l'ordonnanceur utilisateur. L'ordonnanceur peut alors contrôler le délai entre chaque scrutation, et ceci quel que soit le nombre de processus légers en exécution. Le temps de réponse aux requêtes réseaux devient plus ou moins constant. En moyenne, il doit être d'un demi-quantum de temps, soit 5 ms, comme observé dans les résultats.

Utiliser des appels systèmes bloquants fournit de meilleures performances : on observe un temps constant de 450 μs quel que soit le nombre de processus légers en exécution. Cependant, le support des activations est nécessaire pour les gérer correctement. Les appels systèmes bloquants introduisent un surcoût significatif qui apparaît ici lorsqu'aucun processus léger de calcul n'est lancé ou, pour une application réelle, lorsqu'ils sont tous bloqués (colonne 1 de la Table 7.6).

7.3.2.2 Serveur multirequête et agrégation

Dans le cas où l'on doit effectuer de la scrutation, il est important d'optimiser le plus possible les scrutations qui peuvent être assez fréquentes. Dans cette expérience, nous allons mesurer les perturbations induites sur une tâche de calcul par la scrutation due à d'autres processus légers.

Dans notre expérience, un seul processus léger de calcul effectue du travail (calcul et primitives de synchronisation), alors que d'autres processus légers sont en attente de messages sur l'interface réseau. Tous les processus légers en attente utilisent le même serveur d'événements pour les requêtes. Nous observons le temps nécessaire pour effectuer la tâche alors que des événements surviennent au hasard (voir la Table 7.7 et la Figure 7.1).

	Nombre de processus légers en attente							
Version de l'ordonnanceur	1	2	3	4	5	6	7	8
Sans support particulier (ms)	80, 3	101, 3	119, 0	137, 2	156, 6	175, 7	195, 2	215, 7
Avec serveur d'événements (ms)	81.2	84.0	84.0	84.7	86.4	87.9	89.6	91.6

TAB. 7.7 – Temps de calcul d'une tâche en fonction du nombre de processus légers en attente d'entrées/sorties.

Ceci montre qu'agréger les requêtes de détection d'événements dans l'ordonnanceur augmente de manière significative les performances. Sans agrégation, le temps d'exécution de la tâche principale augmente de manière importante avec le nombre de processus légers en attente d'entrées/sorties. Avec l'agrégation, ce temps devient presque constant ; la légère

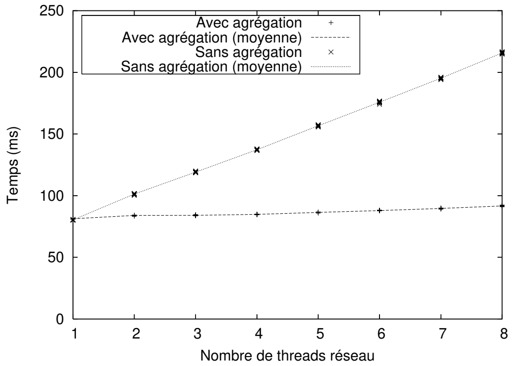


FIG. 7.1 – Temps de calcul d'une tâche en fonction du nombre de processus légers en attente d'événements.

augmentation est due au temps nécessaire pour agréger les requêtes.

7.3.3 Surcoût induit par l'enregistrement des événements

L'enregistrement des traces doit être le plus rapide possible afin de réduire au mieux les perturbations engendrées par les mesures. Pour ce faire, les événements estampillés sont écrits dans des tampons (un par trace) alloués lors de l'initialisation.

Fonction/Macro		Nombre de cycles
Macro	`PROF_IN`	260
Appel système	`getpid()`	1900
Fonction standard d'E/S tamponnée	`printf("test")`	672

TAB. 7.8 – Micro benchmarks

Le Tableau 7.8 permet de comparer les différents coûts (appel système, appel de bibliothèque d'entrée/sortie, enregistrement d'un événement). Il apparaît clairement qu'un appel système est beaucoup plus intrusif qu'un appel de fonction. On observe aussi que l'enregistrement d'un événement dans le tampon dédié est deux à trois fois plus efficace que l'écriture d'une chaîne constante par la fonction `printf`.

Nous avons ensuite cherché à évaluer le surcoût induit par la prise des traces ainsi que la taille des traces générées. Ces deux quantités sont très fluctuantes et dépendent fortement du niveau d'instrumentation du code. On peut en effet se contenter de tracer uniquement l'ordonnancement des processus légers sur le système ou bien, à l'extrême, on peut aussi tracer l'ensemble des fonctions de l'application cible ainsi que celles des bibliothèques utilisées, ce qui permet alors d'obtenir une vue très précise du déroulement du programme.

Le Tableau 7.9 montre le surcoût dû au profilage lors de l'exécution de deux applications types. On compare l'exécution normale (application et bibliothèques compilées sans code d'instrumentation), l'exécution profilée où l'on n'enregistre que les changements de contexte et l'exécution profilée où on enregistre tous les appels de fonction et tous les appels système. Dans les deux derniers cas, il s'agit du même binaire ; seule une option au lancement permet de choisir le masque actif pour l'enregistrement des événements. Ces mesures

	temps d'exécution	nombre d'événements enregistrés	(taille)	débit (Mo/s)
Programme `Sumtime`				
non profilé	220 ms	-		-
profilé	269 ms (+22%)	119 577	(3 529 Mo)	10,2
profilé (complet)	426 ms (+93%)	837 325	(14 421 Mo)	32
Programme `ProdMat`				
non profilé	8.1 s	-		-
profilé	8.1 s (+0%)	2 618	(0,086 Mo)	0,010
profilé complet	8.2 s (+1%)	695 068	(11,801 Mo)	1,456

TAB. 7.9 – Performance pour des applications types

ont été obtenues sur une machine sous LINUX 2.6.4 équipée de processeurs XEON SMT cadencés à 2,8 GHz (quatre processeurs exploités par LINUX). La bibliothèque de processus légers utilisée est la bibliothèque de processus légers à deux niveaux MARCEL qui, dans les versions profilées des programmes, est entièrement instrumentée (toutes ses fonctions sont instrumentées, pas uniquement les changements de contexte). Sur chaque ligne du tableau, on trouve le temps d'exécution du programme, le nombre d'événements enregistrés et la taille de la trace générée, et enfin le débit en écriture dans le fichier de trace au cours de l'exécution de l'application.

La première application (`Sumtime`) est le test intensif de la bibliothèque de processus légers que nous avons présenté auparavant. Ici, beaucoup d'événements sont enregistrés, y compris lorsqu'on se contente d'enregistrer uniquement les changements de contexte car les processus légers se synchronisent (donc se bloquent et passent la main) en permanence. Lorsque l'on n'enregistre que les changements de contexte, on observe une pénalité de 22%. On a alors le déroulement exact de l'ordonnancement des processus légers de l'application. Si, de plus, on désire connaître dans quelle fonction ils se trouvent à tout instant, la pénalité passe à 93%. Ces chiffres élevés s'expliquent par le fait que l'application n'a pas de code de calcul propre : toutes ses fonctions sont très courtes. L'application génère par conséquent un nombre d'événements très important. Il s'agit là d'un pire cas pour notre environnement.

Le second programme (`ProdMat`) est celui de calcul de multiplication de matrices utilisé ci-dessus pour observer le passage à l'échelle des applications. Le déroulement interne des boucles de calcul ne génère pas d'événement particulier ; on désire seulement recueillir l'ordonnancement des processus légers et les temps de calcul de chacune de leurs tâches. On observe alors que le surcoût dû à l'enregistrement d'événements devient négligeable devant les autres coûts de l'application : les perturbations dues aux démons sur le système engendrent des variations du temps d'exécution plus importantes que celles dues à l'instrumentation. Ce type d'observation est plus représentatif de ce que l'on souhaite observer lorsque l'on veut analyser une application multithreadée. Et rien n'empêche d'avoir des portions de code instrumentées de façon plus complète que d'autres, ou encore de modifier le masque des événements enregistrés une fois que l'on a déjà ciblé en partie les périodes les plus intéressantes ou les plus critiques.

7.4 Réalisations logicielles s'appuyant sur MARCEL

Notre bibliothèque MARCEL a été et continue d'être choisie par des projets extérieurs comme élément de base pour le support du multithreading. Nous souhaitons présenter ici quelques uns de ces projets et les raisons de leur support de MARCEL.

7.4.1 HYPERION

L'équipe dirigée par Luc BOUGÉ dans laquelle j'ai été accueilli pour effectuer ma thèse a monté un projet de collaboration NSF/Inria avec le groupe de Phil HATCHER de l'université du New Hampshire. Cette collaboration qui a duré quatre ans, de 1998 à 2001, s'articulait autour de l'utilisation du multithreading distribué pour le support d'applications parallèles développées avec des langages de haut niveau. Elle a conduit à la réalisation d'une machine Java parallèle distribuée qui s'appuyait sur l'ancienne version de MARCEL pour le multithreading.

Le cœur de ce projet, nommé Hyperion [ABH+01b], consiste à founir un environnement distribué capable d'exécuter efficacement des programmes Java au format « *bytecode* » sur des grappes de PC. L'environnement Hyperion s'appuie sur PM^2 en lui déléguant la gestion du multithreading (inhérent aux applications Java) et la gestion des communications [ABH+00a, ABH+00b, KHBB01]. Reposant sur l'utilisation d'une mémoire virtuelle distribuée, cet environnement a mis en évidence la nécessité de pouvoir fournir des services réactifs. Même si des processus légers Java étaient prêts à s'exécuter, répondre aux messages de la mémoire virtuelle partagée devait être prioritaire pour ne pas bloquer l'exécution des autres nœuds.

Ce projet a depuis été abandonné, avant même de pouvoir bénéficier des services du nouveau MARCEL. Il a toutefois été un moteur important de mes réflexions lors de l'élaboration des premières extensions à MARCEL pour offrir des services garantissant une certaine réactivité.

7.4.2 MPICH-MAD

Message Passing Interface Chameleon (MPICH) est une implémentation libre du standard MPI [Mes95], développée au ARGONNE NATIONAL LABORATORY. De nombreuses réalisations existantes – y compris commerciales – de MPI dérivent de MPICH. L'une d'entre elle, MPICH-MAD [AMN01], a choisi d'utiliser MARCEL et MADELEINE comme support exécutif sous-jacent.

Le but de cette adaptation était de permettre l'exploitation efficace, avec des programmes MPI, des grappes de station à réseaux rapides éventuellement multiples. L'introduction des processus légers dans le support exécutif était requis par la bibliothèque de communication pour pouvoir réémettre des messages d'un réseau à un autre sur les machines passerelles indépendamment de l'application MPI. Le faible coût de création a permis de les utiliser également lors de chaque communication asynchrone. Cette version de MPICH peut enfin envoyer des messages asynchrones qui progressent réellement pendant les phases de calcul de l'application.

Les services de MARCEL, en particulier ceux concernant le serveur d'événements, sont utilisés à plusieurs niveaux. D'une part, MADELEINE en fait grandement usage chaque fois que des réseaux rapides doivent être scrutés en espace utilisateur (protocoles GM et BIP pour MYRINET, réseaux SCI, etc.). D'autre part, MPICH-MAD utilise directement le serveur d'événements de MARCEL pour scruter les échanges effectuées par mémoire partagée. Les excellentes performances obtenues par cet environnement témoignent de l'utilité et de la qualité de MARCEL et de ses services.

7.4.3 PADICOTM

PADICOTM [DPP01] a été développé à Rennes dans l'équipe PARIS. Il s'agit d'une plate-forme de communication adaptée aux grilles et permettant aux applications d'utiliser le paradigme de communication le plus adapté pour elles, indépendamment du support réel sous-jacent.

Comme précédemment, pour exploiter les grappes, PADICOTM a choisi comme supports exécutifs MARCEL pour les processus légers et MADELEINE pour les communications. Le choix de MARCEL s'explique par l'intégration réussie qu'il entretient avec la bibliothèque de communication MADELEINE ainsi que par ses excellentes performances intrinsèques.

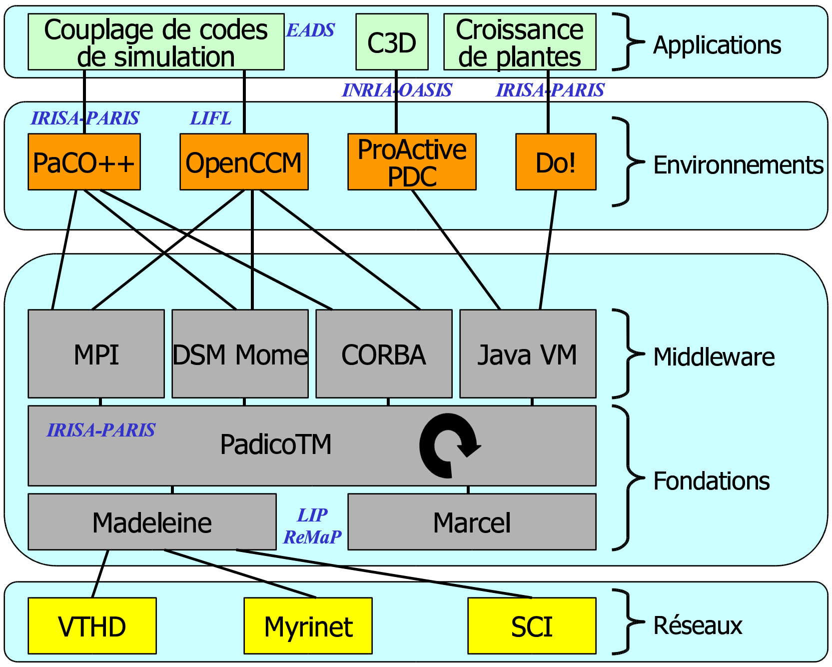


FIG. 7.2 – La plate-forme Padico dans le projet GRID-RMI

PADICOTM est le projet qui a mis en évidence la nécessité de proposer une personnalité POSIX BINAIRE. En effet, PADICOTM permet la cohabitation de plusieurs plates-formes logiciels conséquentes telles que MPICH, CORBA, etc. (voir la Figure 7.2). Assurer une compatibilité POSIX au niveau de MARCEL permet d'adapter plus rapidement ces environnements à PADICOTM, sans empêcher PADICOTM ou MADELEINE d'utiliser les autres personnalités de MARCEL pour accéder aux services évolués. Par exemple, PADICOTM accède directement au serveur d'événements de MARCEL afin de gérer les communications TCP.

Les performances obtenues par PADICOTM en débit et latence sur de nombreux réseaux rapides démontrent une fois encore l'adéquation des services proposés par MARCEL aux besoins des environnements de programmation ainsi que la qualité de l'implémentation proposée.

7.5 Conclusion

Les chapitres précédents ont décrit en détail la structure et les fonctionnalités d'une bibliothèque de processus légers que nous avons implémentée. Cette bibliothèque propose des services originaux dédiés à la réactivité , de plus, elle est structurée autour de nombreux composants pour lui permettre de s'adapter facilement à des besoins applicatifs et des architectures matérielles variés. Ce chapitre a permis de montrer que, malgré la complexité logicielle, l'efficacité de notre bibliothèque est au rendez-vous. Elle exploite les machines mono- ou multiprocesseurs beaucoup plus efficacement que les bibliothèques standards disponibles sur ces systèmes. L'extension développée dans le noyau LINUX, les *Scheduler Activations*, lui offre même une propriété jusque-là refusée aux bibliothèques de processus légers de niveau utilisateur : les appels systèmes bloquants peuvent survenir à tout instant sans craindre qu'ils suspendent toute l'application.

Enfin, plus encore que ses performances intrinsèques, les services dédiés à la réactivité ont été à l'origine de son adoption comme support exécutif dans plusieurs projets extérieurs. L'élément central, le serveur d'événements, lui permet une excellente intégration avec la bibliothèque de communication pour réseaux hautes performances MADELEINE. Le couple MARCEL/MADELEINE devient alors un support exécutif particulièrement intéressant pour les environnements ciblant les grappes. Pour permettre une adoption aisée de MARCEL, il nous reste à compléter le développement de la personnalité POSIX *binaire* : elle assure une compatibilité automatique des applications et des bibliothèques déjà développées. Mais, pour l'instant, seules les principales fonctionnalités (création, synchronisation, etc.) sont proposées dans cette personnalité ; il reste à offrir les fonctionnalités moins utilisées comme celles relatives aux interactions avec les signaux. Mes travaux actuels au CEA/DAM portent justement sur cet aspect.

Chapitre 8

Conclusion et perspectives

Avec la diversification des plates-formes matérielles due à l'apparition de processeurs multithreadés et au développement des machines multiprocesseurs à hiérarchie mémoire (NUMA), l'utilisation de processus légers dans le calcul hautes performances a connu un essor considérable ces dernières années. Ce succès est également dû au développement d'environnements complexes pour la programmation parallèle qui utilisent de plus en plus les processus légers pour exécuter des tâches indépendantes de l'application principale et pour gérer l'asynchronisme des communications.

8.1 Contribution

Très vite, l'utilisation conjointe de processus légers et de bibliothèques de communication en environnement hautes performances fait apparaître des difficultés. Ces deux composants ne se marient pas très bien, même s'ils sont performants indépendamment l'un de l'autre. Des propriétés comme la *réactivité* des applications peuvent grandement souffrir de l'ignorance réciproque de ces deux composants.

Concernant les processus légers, on peut constater que l'interface de la majorité des bibliothèques découle directement de la norme POSIX correspondante. Or cette interface est très insuffisante pour les besoins des supports exécutifs. L'ordonnancement ne peut pas être dirigé, la plate-forme matérielle n'est pas correctement abstraite : il est difficile de développer une application utilisant efficacement des processus légers quelle que soit la plate-forme matérielle sous-jacente. Par exemple, la norme POSIX ne permet pas de distinguer les processeurs en fonction de leur affinité mémoire sur une machine NUMA. Un support exécutif aura alors énormément de difficulté à prendre en compte cette topologie mémoire pour ordonnancer ses processus légers efficacement.

Les travaux exposés dans ce document découlent du constat d'une offre inadaptée des bibliothèques de processus légers vis-à-vis des supports exécutifs pour le calcul parallèle hautes performances. Après avoir analysé les bibliothèques existantes, nous avons conçu des mécanismes et des extensions permettant de répondre à nos besoins spécifiques. Ces travaux ont été intégrés dans la bibliothèque MARCEL afin de les évaluer.

MARCEL est une bibliothèque *caméléon* dont l'objectif premier est d'assurer la *portabilité des performances*. Pour ce faire, elle a été conçue de manière très modulaire s'adaptant, d'une part, à l'architecture et au système sous-jacent (bibliothèque de niveau utilisateur ou mixte selon que la machine est mono- ou multiprocesseur par exemple) et d'autre part, à l'application en n'intégrant que les fonctionnalités qui lui sont nécessaires pour ne pas payer le coût des inutiles.

Pour répondre au problème de la réactivité, nous avons conçu pour MARCEL un *serveur d'événements*. Il permet à l'application (ou aux bibliothèques extérieures) de proposer tout un ensemble de méthodes de détection d'événements. Notre serveur choisit alors la plus adaptée au contexte, en fonction du type de bibliothèque de processus légers et des fonctionnalités du système disponibles. Cette sélection est transparente pour l'application à qui il est fourni une *interface uniforme* pour attendre des événements. Intégrer le serveur à la bibliothèque de processus légers a plusieurs avantages : le choix de la méthode de détection adaptée est facilité par la connaissance des propriétés de la bibliothèque de processus légers elle-même. En cas de scrutation, l'ordonnanceur est capable de garantir une fréquence régulière indépendante du nombre de processus légers prêts et sans nécessiter de changements de contexte inutiles. Enfin, les requêtes de même type peuvent éventuellement être agrégées et réduire ainsi le coût de la détection d'événements.

Pour plus de souplesse et d'efficacité, les processus légers de notre bibliothèque sont de niveau utilisateur. Ceci est problématique vis-à-vis des appels systèmes bloquants qui paralysent l'application entière au lieu du seul processus léger incriminé. Pour résoudre cette difficulté, nous avons implémenté le modèle des *Scheduler Activations* d'Anderson après l'avoir adapté et amélioré dans notre contexte d'utilisation (calcul hautes performances).

Enfin, pour pouvoir suivre le déroulement exact et précis de nos applications, en particulier pour connaître l'ordonnancement exact de nos processus légers sur les processeurs physiques de la machine, nous avons conçu et implémenté des mécanismes de traces. Grâce à la récolte d'une trace noyau et d'une trace applicative, nous sommes en mesure de reconstituer l'exécution précise de notre application.

L'évaluation de notre bibliothèque prouve la validité de notre approche et de nos mécanismes. Ces performances globales sont excellentes comparées à celles des bibliothèques fournies en standard sur les systèmes. De plus, MARCEL peut fonctionner sur de nombreuses architectures. Enfin, notre serveur d'événements nous permet effectivement de garantir une borne de réactivité. Ces performances, surtout en présence de communications, sont directement à l'origine du choix de MARCEL comme support exécutif d'environnement de programmation de plus haut niveau comme MPICH-MAD, une version de MPI multithreadée, ou PADICOTM, un environnement permettant de supporter simultanément plusieurs *middleware* comme MPI, CORBA et JAVA.

8.2 Perspectives

Ces résultats très positifs de MARCEL ouvrent de nombreuses perspectives intéressantes pour l'avenir. Nous en exposons quelques-unes ci-après.

À court terme, une collaboration avec BULL et le CEA va permettre le portage de plusieurs de nos outils vers l'architecture ITANIUM. En effet, si MARCEL fonctionne déjà sur cette plate-forme, tous les outils n'ont pas encore été portés. Ainsi, ni l'interface binaire (personnalité POSIX *binaire*) ni les activations n'ont été encore adaptées. Concernant le mécanisme de trace, la présence d'un registre compteur de cycles accessible en mode utilisateur va permettre le portage de FKT et FUT vers cette architecture.

Ensuite, nos travaux sur la prise de trace peuvent être approfondis. Nous avons une infrastructure nous permettant de récolter des traces et capable de dire *a posteriori* où ces traces ont eu lieu exactement. Il s'agit désormais de l'exploiter en récoltant les événements les plus pertinents pour comprendre le déroulement de l'application. Toute la difficulté consiste justement à définir ce que sont ces événements pertinents. Une collaboration avec le projet de trace de NPTL [1] est déjà amorcée. Gageons qu'elle se révélera bénéfique pour les deux parties.

À moyen terme, il est possible d'utiliser notre bibliothèque comme fondement pour de nouveaux travaux. La flexibilité de MARCEL et son architecture modulaire peuvent désormais être exploitées. Un axe particulièrement intéressant concerne les architectures de type NUMA. La hiérarchisation de leur mémoire montre la nécessité, dans cette situation, de gérer de manière conjointe l'ordonnancement et l'allocation de mémoire. Il est en effet important pour les performances d'ordonnancer au maximum des processus légers sur des processeurs où leur pile et leurs données seront accessibles à faible coût. À l'heure actuelle, l'application a la possibilité de gérer ces placements manuellement afin d'obtenir un résultat correct. Cependant, cela va à l'encontre de l'objectif de portabilité. L'application ne devrait pas avoir à gérer elle-même le placement de ses données et de ses processus légers sur la plate-forme matérielle. Il convient donc de réfléchir aux diverses possibilités permettant à l'application d'exprimer des propriétés sur ses processus légers et ses données afin que la bibliothèque prenne de bonnes décisions d'ordonnancement et de placement. Ces propriétés devraient exprimer plutôt des objectifs sémantiques (à définir) qui seraient donc indépendants de la plate-forme matérielle mais traduits en terme d'ordonnancement et de placement de données par la bibliothèque. Ces travaux sont au centre de la thèse de Samuel THIBAULT.

À plus long terme, nous pourrons nous pencher sur la problématique de traçage d'applications réparties. Actuellement, les traces que nous récoltons et analysons proviennent d'une même machine où tous les événements sont datés par une horloge précise et synchrone. Cependant, le calcul parallèle hautes performances utilise généralement un ensemble de machines, chacune avec son horloge. Il serait intéressant de réfléchir aux possibilités d'étendre nos mécanismes pour tracer des expériences réparties. Reconstruire une trace précise sur une machine donnée nécessite d'intervenir dans l'ordonnanceur pour enregistrer tous les changements de contexte. Peut-être qu'en environnement réparti, il sera nécessaire d'intervenir dans d'autres composants comme les communications pour dater les messages en transit.

Pour finir, je dirai que les travaux pour exploiter correctement le multithreading ne font que commencer. La norme POSIX autour de laquelle un consensus s'était dégagé il y a quelques années montre ses limitations avec les architectures hiérarchiques actuelles. Elle n'est pas adaptée pour interfacer efficacement les processus légers avec d'autres composants comme les communications ni assez souple pour permettre une virtualisation efficace des plates-formes matérielles récentes. Et ce, d'autant plus que le matériel fournit de plus en plus de fonctionnalités. Des nouveaux processeurs SMT incluent désormais une notion de priorité matérielle pour permettre de gérer au niveau logiciel les conflits d'accès aux unités de calcul. De façon plus générale, je pense que c'est une erreur de concevoir le multithreading comme un aspect indépendant des applications. Cette thèse a mis en évidence la nécessité d'intégrer la problématique des communications dans la bibliothèque de processus légers pour obtenir une réactivité portable sur les différentes architectures. Je pense qu'il faudra faire de même avec la mémoire, surtout avec le développement des architectures hiérarchiques à plusieurs niveaux (Les NUMA de multicore SMT devraient bientôt apparaître). De nos jours, le multithreading est devenu incontournable et il est nécessaire de

[1] `http://nptl.bullopensource.org/home.php`

le prendre en compte au plus tôt s'il l'on souhaite qu'il reste efficace et portable. Beaucoup de réalisations devront être repensées pour intégrer correctement cette notion et un travail conséquent au niveau des interfaces sera nécessaire.

Chapitre 9

Bibliographie

[ABH+00a] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. Compiling multithreaded Java bytecode for distributed execution. In *Euro-Par 2000: Parallel Processing*, volume 1900 of *Lect. Notes in Comp. Science*, pages 1039–1052, Munchen, Germany, August 2000. Springer-Verlag. Selected as a Distinguished Paper.

[ABH+00b] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. Implementing Java consistency using a generic, multithreaded DSM runtime system. In *Parallel and Distributed Processing. Proc. Intl Workshop on Java for Parallel and Distributed Computing*, volume 1800 of *Lect. Notes in Comp. Science*, pages 560–567, Cancun, Mexico, May 2000. Held in conjunction with IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag.

[ABH+01a] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, March 2001.

[ABH+01b] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, March 2001.

[ABLL91] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Efficient kernel support for the user-level managment of parallelism. In *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP 91)*, volume 10, pages 95–105, October 1991.

[ADH+02] Bill Abt, Saurabh Desai, David P. Howell, Inaky Perez Gonzalez, and Dave McCracken. Next Generation POSIX Threading Project. `http://www-124.ibm.com/developerworks/oss/pthreads/`, 2002.

[AM03] Olivier Aumage and Guillaume Mercier. MPICH/MADIII : a Cluster of Clusters Enabled MPI Implementation. In *proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, pages 26–33, Tokyo, Japan, May 2003.

[AMN01] Olivier Aumage, Guillaume Mercier, and Raymond Namyst. MPICH/Madeleine: a true multi-protocol MPI for high-performance networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, April 2001. IEEE.

[BB00] S. Benkner and T. Brandes. Efficient parallel programming on scalable shared memory systems with high performance fortran. *Concurrency and Computation: Practice and Experience*, 2000. Special Issue of HPF Users Group Meeting, 2000, Tokyo, Japan.

[BCC+96] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. In *Proceedings of Supercomputing '96*. IEEE Computer Society Press, November 1996.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[BGPP97] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime: efficiency for irregular problems. In C. Lengauer et al., editors, *EURO-PAR'97 Parallel Processing*, volume 1300 of *Lect. Notes in Comp. Science*, pages 591–600. Springer, August 1997.

[BGR97] Thomas Beilsel, Edgar Gabriel, and Mickael Resch. An extension to MPI for distributed computing on MPP's. In Marian Buback, Jack Dongarra, and Jerzy Wasniewski, editors, *EuroPVM/MPI '97: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 75–83, Cracow, Pologne, novembre 1997. Springer Verlag.

[BIG97] J. Briat and B. Plateau I. Ginzburg, M. Pasin. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Euro-Par '97 Conference*, volume 1300 of *Lecture Notes in Computer Science*, pages 590–599, Passau, Germany, août 1997. Springer Verlag.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principales and Practice of Parallel Programming (PPOPP'95)*, pages 207–216, Santa Barbara, California, July 1995. `http://theory.lcs.mit.edu/pub/cilk/focs94.ps.Z`.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computation by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994. `http://theory.lcs.mit.edu/pub/cilk/focs94.ps.Z`.

[BZ98] Rudolf Berrendorf and Heinz Ziegler. Pcl - the performance counter library: A common interface to access hardware performance counters on microprocessors. Technical report, julie, 1998. `http://www.fz-juelich.de/zam/docs/autoren98/berrendorf3.html`.

[Cal99] G. G. H. Calcalheiro. *Athapascan-1 : Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle*. Phd thesis, INPG-IMAG, Grenoble, November 1999.

[CDD+95] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and Whaley. R.C. LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. Technical Report 95, UT, 1995.

[CDL+02] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[Che82] David R. Cheriton. *The Thoth System*. Elsevier Science Inc., New York, NY, USA, 1982. ISBN:0444007016.

[Cor04] Jonathan Corbet. Scheduling domains. `http://lwn.net/Articles/80911`, April 2004.

[CRCM95] B. Le Cun, C. Roucairol, VD. Cung, and T. Mautor. BOB: A unified platform for implementing branch-and-bound like algorithms. Technical Report 95-16, PRISM, Versailles, 1995.

[daC84] Robert daCosta. The History of ada. *Defense Science Magazine*, March 1984.

[Dan02] Vincent Danjean. Réactivité aux événements d'entrées/sorties dans les environnements multithreads. In *Actes des Rencontres francophones du parallélisme (RenPar 14)*, Hammamet, Tunisie, April 2002.

[Dar99] Alain Darte. *De l'organisation des calculs dans les codes répétitifs*. Habilitation à diriger des recherches, Université Claude Bernard de Lyon, pour des travaux effectués à l'école normale supérieure de Lyon, March 1999. `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/HDR/HDR1999/HDR1999-03.ps.Z`.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London New York, 1968.

[dKdOS00] J. Chassin de Kergommeaux and B. de Oliveira Stein. Pajé: an extensible environment for visualizing multi-threaded programs executions. In *Proceedings of the 6th International EuroPar Conference*. EuroPar2000, 2000.

[DM03] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`, January 2003.

[DNR00] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating Kernel Activations in a Multithreaded Runtime System on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Lect. Notes in Comp. Science, Cancun, Mexico, May 2000. Springer-Verlag.

[Dor99] M. Doreille. *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. Phd thesis, INPG-IMAG, Grenoble, December 1999.

[DPP01] A. Denis, C. Pérez, and T. Priol. Towards high performance corba and mpi middlewares for grid computing. In *In Proc of the 2nd International Workshop on Grid Computing*, Denver, Colorado, USA, November 2001. To appear.

[DPP02] Alexandre Denis, Christian Pérez, and Thierry Priol. PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 144–151, Berlin, Germany, May 2002. IEEE Computer Society.

[Eng99] Ralf S. Engelschall. GNU Portable Threads (Pth). `http://www.gnu.org/software/pth/`, 1999.

[Eng00] Ralf S. Engelschall. Portable Multithreading — The Signal Stack Trick for User-Space Thread Creation. In *USENIX Annual Technical Conference*, pages 239–250, San Diego, California, USA, June 2000. `http://www.usenix.org/publications/library/proceedings/usenix2000/genera`

[FGIS97] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede: Easy parallel programming in available distributed environments. *Software Practice and Experience*, 27(8):929–965, Août 1997.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal on Supercomputer Applications*, 11(2):115–128, 1997.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998. `http://theory.lcs.mit.edu/pub/cilk/cilk5.ps.gz`.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. In *IEEE Transactions on Computers*, volume 9 of *C-21*, pages 948–960, September 1972.

[Gal99] F. Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. Phd thesis, INPG-IMAG, Grenoble, September 1999.

[Gin97] I. Ginzburg. *Athapascan-0b: Intégration efficace et portable de multiprogrammation légère et de communications*. Thèse de doctorat, Institut National Polytechnique de Grenoble, LMC, Sep 1997.

[GKM82] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *SIGPLAN Notices*, 1982.

[GRBK98] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed Computing in a Heterogeneous Computing Environment. In Vassil Alexandrov and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Sciences, pages 180–188. Springer, 1998.

[HCM94] M. Haines, D. Cronk, and P. Mehrotra. On the design of chant: A talking threads package. In *Proc. of Supercomputing'94*, pages 350–359, Washington, November 1994.

[Hoa74] C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *Comm. ACM*, 17(10):549–557, October 1974.

[HRR00] P. Hénon, P. Ramet, and J. Roman. Pastix: A parallel sparse direct solver based on a static scheduling for mixed 1d/2d blocks distributions. In *Proceedings of Irregular'2000*, pages 519–525. Springer Verlag, 2000.

[Int03] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 3 : System Programming Guide*, 2003.

[Joh95] K. L. Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, Laboratory for Computer Science, MIT, Cambridge, USA, December 1995. Technical Report MIT/LCS/TR-674.

[KHBB01] T. Kielmann, P. Hatcher, L. Bougé, and H. Bal. Enabling Java for high-performance computing exploiting Distributed Shared Memory and Remote Method Invocation. *Communications of the ACM*, 44(10):110–117, Octobre 2001. Special issue on Java for High Performance Computing.

[L$^+$66] B. W. Lampson et al. A user machine in a time-sharing system. *Proc. IEEE*, 54(12):1744–1766, December 1966.

[Ler96] Xavier Leroy. The LinuxThreads Library. `http://pauillac.inria.fr/~xleroy/linuxthreads/`, 1996.

[Lib01] Davide Libenzi. Improving (network) I/O performance. `http://www.xmailserver.org/linux-patches/nio-improve.html`, 2001.

[LMD04] J.L. Lawall, G. Muller, and H. Duchesne. Language Design for Implementing Process Scheduling Hierarchies. In *Symposium on Partial Evaluation and Program Manipulation*, pages 80–91, Verona, Italy, August 2004. ACM SIGPLAN.

[LRBB96] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating Polling, Interrupts, and Thread Management. In *Proc. 6th Symp. on the Frontiers of Massively Parallel Computing (Frontiers '96)*, pages 13–22, Annapolis, MD, October 1996.

[MCC$^+$95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[McC02] Dave McCracken. POSIX Threads and the Linux Kernel. In *Ottawa Linux Symposium*, pages 330–337, Ottawa, Ontario, Canada, 2002.

[MCLD01] S. Moore, D. Cronk, K. London, and J. Dongarra. Review of Performance Analysis Tools for MPI Parallel Programs. In Springer Verlag, editor, *8th European PVM/MPI Users' Group Meeting*, number 2131 in Lect. Notes in Comp. Science, pages 241–248, 2001.

[Mes95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Accessible à `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[MS04] Allen D. Malony and Sameer S. Shende. Overhead Compensation in Performance Profiling. In *Proc. Europar 2004 Conference*. LNCS, 2004. à paraître.

[Muc02] Phil Mucci. Dynaprof and PAPI: A Tool for Dynamic Runtime Instrumentation and Performance Analysis. ScicomP 6, `http://icl.cs.utk.edu/papi`, August 2002.

[Mue93] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, 1993. `http://moss.csc.ncsu.edu/~mueller/pthreads`.

[MW03] B. Mohr and F. Wolf. Kojak - a tool set for automatic performance analysis of parallel applications. In *Proc. of the European Conference on Parallel Computing (EuroPar)*, LNCS 2790, pages 1301–1304. Springer, August 2003.

[Nam01] Raymond Namyst. *Contribution à la conception de supports exécutifs multithreads performants*. Habilitation à diriger des recherches, Université Claude Bernard de Lyon, pour des travaux effectués à l'école normale supérieure de Lyon, December 2001. `http://dept-info.labri.fr/~namyst/runtime/biblio/Namyst/NamystHDR.pdf`.

[NM95] Raymond Namyst and Jean-François Méhaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.

[Ope99a] *OpenMP Fortran Application Program Interface*, Novembre 1999. `http://www.openmp.org`.

[Ope99b] *OpenMP Fortran Application Program Interface*, November 1999. `http://www.openmp.org`.

[PB03] Luciano Porto Barreto. *Conception aisée et robuste d'ordonnanceurs de processus au moyen d'un langage dédié*. PhD thesis, Université de Rennes 1, June 2003.

[Pro93] Christopher Provenzano. MIT Pthreads. `http://www.humanfactor.com/pthreads/mit-pt` 1993.

[RAA+90] M. Rozier, V. Abrossimov, F. Armand, J. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating System. Technical report, Chorus systèmes, Bolton Landing, NY, USA, 1990.

[RC01] Robert D. Russell and Mrinalini Chavan. Fast Kernel Tracing: a Performance Evaluation Tool for Linux. In *Proc. 19th IASTED International Conference on Applied Informatics (AI 2001)*. IASTED, February 2001.

[Rov86] P. R. Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 37(8):46–57, November 1986.

[Rus02a] Robert D. Russell. FKT: Fast Kernel Tracing. Technical Report 00-02, University of New Hampshire, 2002.

[Rus02b] Rusty Russell. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. *Linux Symposium*, pages 479–495, June 2002.

[Sal66] Jerome Howard Saltzer. Traffic Control in a Multiplexed Computer System. Technical report mac-tr-30 (thesis), Cambridge, Massachusetts Institue of Technology, 1966, 1966.

[SDGM94] V. Sunderam, J. Dongarra, A. Geist, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20(4):531–547, April 1994.

[She01] S.S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.

[SMC+98] S. Shende, A. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[Thi03] Samuel Thibault. Developping a software tool for precise kernel measurements. Master's thesis, University of New Hampshire, 2003.

[Thi04] Samuel Thibault. *Un ordonnanceur flexible pour machines multiprocesseurs hiérarchisées*. Dea, École normale supérieure de Lyon, July 2004. `http://perso.ens-lyon.fr/samuel.thibault/stage/3/rapport.ps.gz`.

[TM99] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.

[TMvR86] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *IEEE, Proc. Sixth Int'l Conf on Distributed Computing Systems*, pages 558–563, 1986. `www.cs.vu.nl/pub/amoeba/`.

[URŠ03] Theo Ungerer, Borut Robič, and Jurij Šilc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys (CSUR)*, 35(1):29–63, March 2003. ISBN:0360-0300.

[vBCZ+03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles table of contents*, pages 268–281. ACM Press, New York, NY, USA, 2003. ISBN:1-58113-757-5.

[VLR+03] Geoffroy Vallée, Renaud Lottiaux, Louis Rilling, Jean-Yves Berthou, Ivan Dutka-Malhen, , and Christine Morin. A Case for Single System Image Cluster Operating Systems: Kerrighed Approach. *Parallel Processing Letters*, 13(2), June 2003.

[vRBM96] R. van Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[Wil02] Nathan J. Williams. An Implementation of Scheduler Activations on the NetBSD Operating System. In *USENIX Annual Technical Conference*, pages 99–108, June 2002.

[Wir77] N. Wirth. Modula: A Language for Modular Multiprogramming. *Software - Practice and Experience*, 7(1):3–35, January-February 1977.

[WM03] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *J. Syst. Archit.*, 49(10-11):421–439, 2003.

[YD00] Karim Yaghmour and Michel R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceeding of the 2000 USENIX Annual Technical Conference*, June 2000.

Chapitre 10

Liste des publications

Conférences internationales avec publication des actes et comité de lecture

[A] Luc Bougé, Vincent Danjean, and Raymond Namyst. Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API. In B. Monien and R. Feldman, editors, *Parallel Processing : 8th International Euro-Par Conference (Euro-Par 2002)*, volume 2400 of *Lect. Notes in Comp. Science*, pages 605–614, Paderborn, Germany, août 2002. Held in conjunction with ACM and IFIP, Springer-Verlag Heidelberg. 10 pages.

[B] Vincent Danjean and Raymond Namyst. Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events. In *Proceedings of the 2003 International Conference on High Performance Computing (HiPC '03)*, volume 2913 of *Lect. Notes in Comp. Science*, pages 490–499, Hyderabad, India, décembre 2003. Held in conjunction with IEEE Computer Society and ACM, Springer-Verlag. 10 pages.

[C] Vincent Danjean, Raymond Namyst, and Robert Russell. Linux kernel activations to support multithreading. In *Proc. 18th IASTED International Conference on Applied Informatics (AI 2000)*, pages 718–723, Innsbruck, Austria, février 2000. IASTED.

Colloques internationaux avec publication des actes et comité de lecture

[D] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, Cancun, Mexico, mai 2000. Held in conjunction with IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag.

Revues nationales avec comité de lecture

[E] Vincent Danjean and Pierre-André Wacrenier. Mécanismes de traces efficaces pour programmes multithreadés. *TSI*, 2005. To appear, version longue de [H].

Conférences nationales avec publication des actes et comité de lecture

[F] Vincent Danjean. LinuxActivations : un support système performant pour les applications de calcul multithreads. In *Actes des Rencontres francophones du parallélisme (RenPar 12)*, pages 87–92, LIB, Univ. Besançon, juin 2000. Version abrégée de [M].

[G] Vincent Danjean. Réactivité aux événements d'entrées/sorties dans les environnements multithreads. In *Actes des Rencontres francophones du parallélisme (RenPar 14)*, Hammamet, Tunisie, avril 2002.

[H] Vincent Danjean. Mécanismes de traces efficaces pour programmes multithreadés. In Michel Auguin, Françoise Baude, Dominique Lavenier, and Michel Riveill, editors, *Actes de RenPar'15, CFSE'3, SympAAA'2003*, pages 92–100, La Colle sur Loup, France, octobre 2003. INRIA. ISBN 2-7261-1264-1.

Rapports de recherche

[I] Luc Bougé, Vincent Danjean, and Raymond Namyst. Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API. Research Report RR-4471, INRIA, Rhone-Alpes / Rennes, France, juin 2002. Published as [A].

[J] Vincent Danjean. Introduction d'un ordonnanceur de processus légers mixte dans PM2 pour l'exploitation efficace des architectures multiprocesseurs. Rapport de stage MIM1, Magistère d'informatique et modélisation (MIM), ENS Lyon, LIP, ENS Lyon, septembre 1998. Stage effectué sous la direction de Luc Bougé et Raymond Namyst.

[K] Vincent Danjean. Extending the Linux kernel with activations for better support of multithreaded programs and integration in PM2. Rapport de stage MIM2, Magistère d'informatique et modélisation (MIM), ENS Lyon, Dept. Comp. Science, Univ. New Hampshire, Durham, NH, USA, septembre 1999. Stage effectué sous la direction de Robert Russell et Phil Hatcher, en liaison avec Raymond Namyst et Luc Bougé. Publié comme [C].

[L] Vincent Danjean. Environnement multithreads distribués: traitement efficace et réactif des communications. Rapport de stage de DEA, DEA d'informatique fondamentale, Univ. Claude Bernard, Lyon 1, France, juin 2000.

[M] Vincent Danjean. Linux Activations : un support système performant pour les applications de calcul multithreads. Research Report RR2000-14, LIP, ENS Lyon, Lyon, France, mars 2000. Publié comme [F].

Manuels

[N] the PM2 Team. *Getting Started with PM2*, 2001. 59 pages, `http://www.pm2.org`.

Annexe A

Interface complète du serveur d’événements

```
/****************************************************************
 * Certains commentaires sont étiquetés par U/S/C/I signifiant
 * User : utile aux threads applicatifs pour exploiter ces mécanismes
 * Start : utile pour l'initialisation
 * Call-backs : utile pour le concepteur de call-backs
 * Internal : ne devrait pas être utilisé hors de Marcel
 *
 * Ce ne sont que des indications, pas des contraintes
 ****************************************************************/

/****************************************************************
 * Récupère l'adresse d'une structure englobante
 * Il faut donner :
 * - l'adresse du champ interne
 * - le type de la structure englobante
 * - le nom du champ interne dans la structure englobante
 */
#define struct_up(ptr, type, member) \
        tbx_container_of(ptr, type, member)


/****************************************************************
 * Les types abstaits pour le serveur et les événements
 */
/* Le serveur définit les call-backs et les paramètres de scrutation à
 * utiliser. Il va éventuellement regrouper les ressources
 * enregistrées (en cas de scrutation active par exemple)
 */
typedef struct marcel_ev_server *marcel_ev_server_t;

/* Une requête définit une entité qui pourra recevoir un événement que
 * l'on pourra attendre.  Diverses requêtes d'un serveur pourront être
 * groupées (afin de déterminer plus rapidement l'ensemble des états
 * de chacune des requêtes à chaque scrutation)
 */
typedef struct marcel_ev_req *marcel_ev_req_t;

/* Un thread peut attendre l'arrivée effective d'un événement
```

```
 * correspondant à une requête. Le thread est déschedulé tant que la
 * requête n'est pas prête (que l'événement n'est pas reçu).
 */
typedef struct marcel_ev_wait *marcel_ev_wait_t;

/*************************************************************
 * Initialisation d'un serveur
 */

/* Initialisation statique
 * var : la variable constante marcel_ev_server_t
 * name: une chaîne de caractères pour identifier ce serveur dans les
 *          messages de debug
 */
#define MARCEL_EV_SERVER_DEFINE(var, name) \
   struct marcel_ev_server ma_s_#var = MARCEL_EV_SERVER_INIT(ma_s_#var, name); \
   const MARCEL_EV_SERVER_DECLARE(var) = &ma_s_#var

/* Idem, mais dynamique */
#ifndef __cplusplus
inline static void marcel_ev_server_init(marcel_ev_server_t server, char* name);
#endif

/* Enregistrement des call-backs utilisables */
inline static int marcel_ev_server_add_callback(marcel_ev_server_t server,
                                                marcel_ev_op_t op,
                                                marcel_ev_callback_t *func);
/* Réglage des paramètres de scrutation */
int marcel_ev_server_set_poll_settings(marcel_ev_server_t server,
                                       unsigned poll_points,
                                       unsigned poll_frequency);

/* Démarrage du serveur
 * Il devient possible d'attendre des événements
 */
int marcel_ev_server_start(marcel_ev_server_t server);

/* Arrêt du serveur
 * Il est nécessaire de le réinitialiser pour le redémarrer
 */
int marcel_ev_server_stop(marcel_ev_server_t server);

/*************************************************************
 * Fonctions à l'usage des threads applicatifs
 */

/* Attribut pouvant être attaché aux événements */
enum {
        /* Désactive la requête lorsque survient l'occurrence
         * suivante de l'événement */
        MARCEL_EV_ATTR_ONE_SHOT=1,
        /* Ne réveille pas les threads en attente d'événements du
         * serveur (ie marcel_ev_server_wait())*/
```

```
        MARCEL_EV_ATTR_NO_WAKE_SERVER=2,
};

/* Un raccourci pratique des fonctions suivantes, utile si l'on ne
 * soumet la requête qu'une seule fois. Les opérations suivantes sont
 * effectuées : initialisation, soumission et attente d'un
 * événement avec ONE_SHOT positionné */
int marcel_ev_wait(marcel_ev_server_t server, marcel_ev_req_t req,
                   marcel_ev_wait_t wait, marcel_time_t timeout);

/* Initialisation d'un événement
 * (à appeler en premier si l'on utilise autre chose que marcel_ev_wait) */
int marcel_ev_req_init(marcel_ev_req_t req);

/* Ajout d'un attribut spécifique à une requête */
int marcel_ev_req_attr_set(marcel_ev_req_t req, int attr);

/* Soumission d'une requête (le serveur PEUT commencer à scruter si cela
 * lui convient) */
int marcel_ev_req_submit(marcel_ev_server_t server, marcel_ev_req_t req);

/* Abandon d'une requête et retour des threads en attente sur cette
 * requête (avec le code de retour fourni) */
int marcel_ev_req_cancel(marcel_ev_req_t req, int ret_code);

/* Attente bloquante d'un événement sur une requête déjà enregistrée */
int marcel_ev_req_wait(marcel_ev_req_t req, marcel_ev_wait_t wait,
                       marcel_time_t timeout);

/* Attente bloquante d'un événement sur une quelconque requête du serveur */
int marcel_ev_server_wait(marcel_ev_server_t server, marcel_time_t timeout);

/* Renvoie une requête survenue n'ayant pas l'attribut NO_WAKE_SERVER
 * (utile au retour de server_wait())
 * À l'abandon d'une requête (wait, req_cancel ou ONE_SHOT) la requête
 * est également retirée de cette file (donc n'est plus consultable) */
marcel_ev_req_t marcel_ev_get_success_req(marcel_ev_server_t server);

/* Exclusion mutuelle pour un serveur d'événements
 *
 * - les call-backs sont TOUJOURS appelés à l'intérieur de cette
 *   exclusion mutuelle.
 * - les call-backs BLOCK_ONE|ALL doivent relâcher puis reprendre ce
 *   lock avant et après l'appel système bloquant avec les deux fonctions
 *   prévues pour (marcel_ev_callback_*).
 * - les fonctions précédentes peuvent être appelées avec ou sans ce
 *   lock
 * - le lock est relâché automatiquement par les fonctions d'attente
 *   (marcel_ev_*wait*())

 * - les call-backs et les réveils des threads en attente sur les
 *   événements signalés par le call-back sont atomiques (vis-à-vis de
 *   ce lock)
```

```
 * – si le lock est pris avant les fonctions d'attente (ev_wait_*),
 *     la mise en attente est atomique vis–à–vis des call–backs
 *     ie: un call–back signalant l'événement attendu réveillera cette
 *     attente
 * – si un événement à la propriété ONE_SHOT, le désenregistrement est
 *     atomic vis–à–vis du call–back qui a généré l'événement.
 */
int marcel_ev_lock(marcel_ev_server_t server);
int marcel_ev_unlock(marcel_ev_server_t server);
/* Pour BLOCK_ONE et BLOCK_ALL avant et après l'appel système bloquant */
int marcel_ev_callback_will_block(marcel_ev_server_t server);
int marcel_ev_callback_has_blocked(marcel_ev_server_t server);

/* Appel forcé de la fonction de scrutation */
void marcel_ev_poll_force(marcel_ev_server_t server);
/* Idem, mais on est sûr que l'appel a été fait quand on retourne */
void marcel_ev_poll_force_sync(marcel_ev_server_t server);

/****************************************************************
 * Les constantes pour le polling
 * Elles peuvent être ORed
 */
#define MARCEL_EV_POLL_AT_TIMER_SIG   1
#define MARCEL_EV_POLL_AT_YIELD       2
#define MARCEL_EV_POLL_AT_LIB_ENTRY   4
#define MARCEL_EV_POLL_AT_IDLE        8

/****************************************************************
 * Les différents opérations call–backs possibles
 */
typedef enum {
        /* id, XX, req to poll, nb_req_grouped, flags */
        MARCEL_EV_FUNCTYPE_POLL_POLLONE,
        /* id, XX, NA, nb_ev_grouped, NA */
        MARCEL_EV_FUNCTYPE_POLL_GROUP,
        /* id, XX, NA, nb_ev_grouped, NA */
        MARCEL_EV_FUNCTYPE_POLL_POLLANY,
        /* Les suivants ne sont pas encore utilisés... */
        /* id, XX, ev to block, NA, NA */
        MARCEL_EV_FUNCTYPE_BLOCK_WAITONE,
        MARCEL_EV_FUNCTYPE_BLOCK_WAITONE_TIMEOUT,
        MARCEL_EV_FUNCTYPE_BLOCK_GROUP,
        MARCEL_EV_FUNCTYPE_BLOCK_WAITANY,
        MARCEL_EV_FUNCTYPE_BLOCK_WAITANY_TIMEOUT,
        MARCEL_EV_FUNCTYPE_UNBLOCK_WAITONE,
        MARCEL_EV_FUNCTYPE_UNBLOCK_WAITANY,
        /* PRIVATE */
        MA_EV_FUNCTYPE_SIZE
} marcel_ev_op_t;

/****************************************************************
 * Le prototype des call–back
 * id : le serveur
```

```
 * op : l'opération (call–back) demandée
 * req : pour *(POLL|WAIT)ONE* : la requête à tester en particulier
 * nb_req : pour POLL_* : le nombre de requêtes groupées
 * option : flags dépendant de l'opération
 *   – pour POLL_POLLONE :
 *       + EV_IS_GROUPED : si la requête est déjà groupée
 *       + EV_ITER : si POLL_POLLONE est appelée sur toutes les requêtes
 *                   en attente (ie POLL_POLLANY n'est pas disponible)
 *
 * La valeur de retour est pour l'instant ignorée.
 */
typedef int (marcel_ev_callback_t)(marcel_ev_server_t server,
                                   marcel_ev_op_t op,
                                   marcel_ev_req_t req,
                                   int nb_ev, int option);

typedef marcel_ev_callback_t *marcel_ev_pcallback_t;

/****************************************************************
 * Les flags des call–backs (voir ci–dessus)
 */
enum {
        /* Pour POLL_POLLONE */
        MARCEL_EV_OPT_REQ_IS_GROUPED=1,
        MARCEL_EV_OPT_REQ_ITER=2,
};

/****************************************************************
 * Itérateurs pour les call–backs
 */

/***************************************************************
 * Itérateur pour les requêtes enregistrées d'un serveur
 */

/* Itérateur avec le type de base
   marcel_ev_req_t req : itérateur
   marcel_ev_server_t server : serveur
*/
#define FOREACH_REQ_REGISTERED_BASE(req, server) \
  list_for_each_entry((req), &(server)->list_req_registered, chain_req_registered)

/* Idem mais protégé (usage interne) */
#define FOREACH_REQ_REGISTERED_BASE_SAFE(req, tmp, server) \
  list_for_each_entry_safe((req), (tmp), &(server)->list_req_registered, chain_req_registered)

/* Itérateur avec un type utilisateur
   [User Type] req : pointeur sur structure contenant un struct marcel_req
                     (itérateur)
   marcel_ev_server_t server : serveur
   member : nom de struct marcel_ev dans la structure pointée par req
*/
#define FOREACH_REQ_REGISTERED(req, server, member) \
```

```
    list_for_each_entry((req), &(server)->list_req_registered, member.chain_req_registered)

/**************************************************************
 * Itérateur pour les requêtes groupées de la scrutation (polling)
 */

/* Itérateur avec le type de base
     marcel_ev_req_t req : itérateur
     marcel_ev_server_t server : serveur
*/
#define FOREACH_REQ_POLL_BASE(req, server) \
    list_for_each_entry((req), &(server)->list_req_poll_grouped, chain_req_grouped)

/* Idem mais protégé (usage interne) */
#define FOREACH_REQ_POLL_BASE_SAFE(req, tmp, server) \
    list_for_each_entry_safe((req), (tmp), &(server)->list_req_poll_grouped, chain_req_grouped)

/* Itérateur avec un type utilisateur
     [User Type] req : pointeur sur structure contenant un struct marcel_req
                       (itérateur)
     marcel_ev_server_t server : serveur
     member : nom de struct marcel_ev dans la structure pointée par req
*/
#define FOREACH_REQ_POLL(req, server, member) \
    list_for_each_entry((req), &(server)->list_req_poll_grouped, member.chain_req_grouped)


/**************************************************************
 * Itérateur pour les requêtes groupées de l'attente bloquante
 */

/* Itérateur avec le type de base
     marcel_ev_req_t req : itérateur
     marcel_ev_server_t server : serveur
*/
#define FOREACH_REQ_BLOCK_BASE(req, server) \
    list_for_each_entry((req), &(server)->list_req_block, chain_req)

/* Idem mais protégé (usage interne) */
#define FOREACH_REQ_BLOCK_BASE_SAFE(req, tmp, server) \
    list_for_each_entry_safe((req), (tmp), &(server)->list_req_block, chain_req)

/* Itérateur avec un type utilisateur
     [User Type] req : pointeur sur structure contenant un struct marcel_req
                       (itérateur)
     marcel_ev_server_t server : serveur
     member : nom de struct marcel_ev dans la structure pointée par req
*/
#define FOREACH_REQ_BLOCK(req, server, member) \
    list_for_each_entry((req), &(server)->list_req_block, member.chain_req)


/**************************************************************
```

```
 * Itérateur pour les attentes d'événements rattachés à une requête
 */

/* Itérateur avec le type de base
     marcel_ev_wait_t wait : itérateur
     marcel_ev_req_t req : requête
*/
#define FOREACH_WAIT_BASE(wait, req) \
   list_for_each_entry((wait), &(req)->list_wait, chain_wait)

/* Idem mais protégé (usage interne) */
#define FOREACH_WAIT_BASE_SAFE(wait, tmp, req) \
   list_for_each_entry_safe((wait), (tmp), &(req)->list_wait, chain_wait)

/* Itérateur avec un type utilisateur
     [User Type] req : pointeur sur structure contenant un struct marcel_req
                       (itérateur)
     marcel_ev_server_t server : serveur
     member : nom de struct marcel_ev dans la structure pointée par req
*/
#define FOREACH_WAIT(wait, req, member) \
   list_for_each_entry((wait), &(req)->list_wait, member.chain_wait)



/* Macro utilisable dans les call-backs pour indiquer qu'une requête
     a reçu un événement.

     * la requête sera fournie à marcel_ev_get_success_req() (tant
     qu'elle n'est pas désenregistrée)
     * les threads encore en attente d'un événement lié à cette requête seront
     réveillés et renverront le code 0

     marcel_ev_req_t req : requête
*/
#define MARCEL_EV_REQ_SUCCESS(req) \
   do { \
            list_del(&(req)->chain_req_ready); \
            list_add(&(req)->chain_req_ready, &(req)->server->list_req_ready); \
   } while(0)



/* Macro utilisable dans les call-backs pour réveiller un thread en attente
     en lui fournissant le code de retour
     marcel_ev_wait_t wait : l'événement à reveiller
*/
#define MARCEL_EV_WAIT_SUCCESS(wait, code) \
   do { \
            list_del(&(wait)->chain_wait); \
            (wait)->ret_code=(code); \
   } while(0)
```

Annexe B

Le format des traces

Lorsque l'on désire observer le comportement d'une application avec nos outils, son exécution génère tout d'abord des traces noyau et utilisateur. Si l'on souhaite avoir le comportement à grain très fin de l'application, il faudra relever de nombreuses mesures. Il est donc important que ces mesures soient stockées de manière relativement compacte en évitant d'enregistrer de l'information inutile ou redondante.Plus l'empreinte d'une trace sera faible, plus nous pourrons en enregistrer.

La date de l'événement est obtenue à partir du registre compteur de cycles qui fournit une valeur sur 64 bits. Concernant son enregistrement, en mode noyau, seuls les 32 bits de poids faibles sont sauvegardés : il y a en effet suffisamment d'événements noyau (ne serait-ce que les interruptions de l'horloge) pour que l'on soit assuré d'avoir moins de 2^{32} cycles (de l'ordre de la seconde) entre chacun des événements noyau et de pouvoir reconstituer à coup sûr l'ordonnancement des événements noyau. Cet argument ne tient plus en mode utilisateur, car on ne peut faire aucune hypothèse générale sur la prise de mesure d'un processus léger utilisateur. Il est donc nécessaire d'enregistrer les 64 bits du registre.

offset	taille	description
0	4	32-bits bas du cycle d'horloge
4	2	pid du thread noyau
6	2	#CPU (en SMP)
8	1	taille de l'entrée (en octet)
9	3	code
12	4	paramètre 1 (si présent)
16	4	paramètre 2 (si présent)
20	4	paramètre 3 (si présent)
24	4	paramètre 4 (si présent)
28	4	paramètre 5 (si présent)

TAB. B.1 – Format d'une trace d'un événement noyau

Les tableaux B.1 et B.2 décrivent le format des événements ; on y précise les champs requis (partie supérieure) où on retrouve l'estampillage et les champs optionnels (partie inférieure). On peut observer que les enregistrements des plus petits événements (sans argument) n'occupent que 12 ou 16 octets dans la trace, autorisant leur mémorisation en grand nombre.

offset	taille	description
0	8	64-bits du cycle d'horloge
8	1	taille de l'entrée (en octet)
9	3	code
12	4	TID du thread (en SMP)
16	4	paramètre 2 (si présent)
20	4	paramètre 3 (si présent)
24	4	paramètre 4 (si présent)
28	4	paramètre 5 (si présent)

TAB. B.2 – Format d'une trace d'un événement utilisateur

-=-=-=-