8

SYSTÈMES À PROCESSEURS MULTIPLES

Toute communication entre des composants électroniques (ou optiques) se résume finalement à l'envoi de messages - des chaînes de bits bien définies - entre eux. Les différences résident à l'échelle de temps, l'échelle de distance et l'organisation logique impliquée. À l'extrême, on trouve les multiprocesseurs à mémoire partagée, dans lesquels entre deux et environ 1 000 unités centrales communiquent via une mémoire partagée. Dans ce modèle, chaque CPU a un accès égal à l'ensemble de la mémoire physique et peut lire et écrire des mots individuels à l'aide des instructions LOAD et STORE. L'accès à un mot de mémoire prend généralement entre 1 et 10 nano sec. Comme nous le verrons, il est désormais courant de placer plus d'un cœur de traitement sur une seule puce de processeur, les cœurs partageant l'accès à la mémoire physique.

la mémoire principale (et parfois même le partage des caches). En d'autres termes, le modèle de multiordinateurs à mémoire partagée peut être mis en œuvre en utilisant des unités centrales de traitement physiquement séparées, des cœurs multiples sur une seule unité centrale de traitement, ou une combinaison de ces éléments. Si ce modèle, illustré à la Fig. 8-l(a), semble simple, sa mise en œuvre réelle ne l'est pas vraiment et implique généralement un passage considérable de messages sous les couvertures, comme nous l'expliquerons bientôt. Toutefois, ce passage de messages est invisible pour les programmeurs.

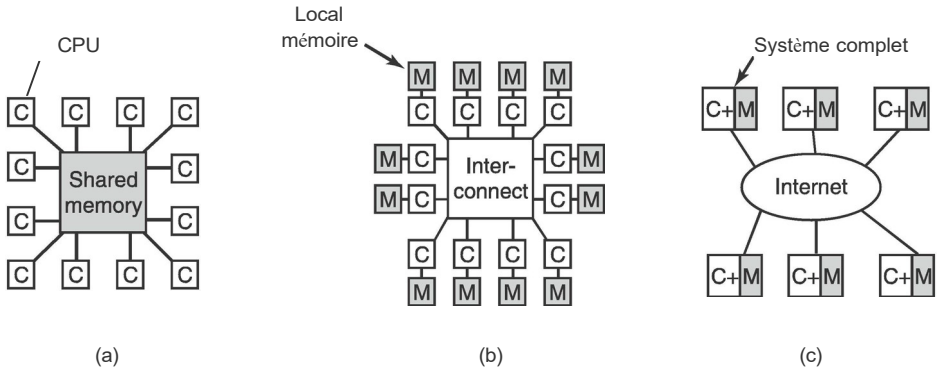


Figure 8-1. (a) Un multiprocesseur à mémoire partagée. (b) Un multiordinateur à passage de messages. (c) Un système distribué à grande échelle.

Vient ensuite le système de la Fig. 8-l(b) dans lequel les paires CPU-mémoire sont reliées par une interconnexion à grande vitesse. Ce type de système est appelé un multi-ordinateur à messages. Chaque mémoire est locale à une seule unité centrale et ne peut être accédée que par cette unité. Les CPU communiquent en envoyant des messages de plusieurs mots sur l'interconnexion. Avec une bonne interconnexion, un message court peut être envoyé en 10--50 *µsec,* mais toujours beaucoup plus long que le temps d'accès à la mémoire de la Fig. 8-l(a). Il n'y a pas de mémoire globale partagée dans cette conception. Les multi-ordinateurs (c'est-à-dire les systèmes à passage de messages) sont beaucoup plus faciles à construire que les multiprocesseurs (à mémoire partagée), mais ils sont plus difficiles à programmer. Ainsi, chaque genre a ses fans.

Le troisième modèle, illustré à la figure 8-l(c), relie des systèmes informatiques complets par un réseau étendu, tel qu'Internet, pour former un système distribué. Chacun d'eux possède sa propre mémoire et les systèmes communiquent par passage de méssage. La seule différence réelle entre la Fig. 8-l(b) et la Fig. 8-l(c) est que dans cette dernière, des ordinateurs complets sont utilisés et les délais de transmission des messages sont souvent de 10--100 msec. Ce long délai oblige ces systèmes **faiblement couplés** à être utilisés de manière différente des systèmes **étroitement couplés** de la figure 8-l(b). Les trois types de systèmes diffèrent par leurs délais d'environ trois ordres de grandeur. C'est la différence entre un jour et trois ans.

Ce chapitre comporte trois grandes sections, correspondant à chacun des trois modèles de la Fig. 8-1. Pour chaque modèle abordé dans ce chapitre, nous commençons par une brève description de la situation.

Introduction au matériel concerné. Nous passons ensuite au logiciel, en particulier aux questions relatives au système d'exploitation pour ce type de système. Comme nous le verrons, dans chaque cas, différents problèmes sont présents et différentes approches sont nécessaires.

8.1 MULTIPROCESSEURS

Un **multiprocesseur à mémoire partagée** (ou simplement multiprocesseur dans la suite du texte) est un système informatique dans lequel deux CPU ou plus partagent un accès complet à une RAM commune. Un programme exécuté sur l'une des unités centrales voit un espace de mémoire virtuelle normal (généralement paginé). La seule propriété inhabituelle de ce système est que l'unité centrale peut écrire une valeur dans un mot de mémoire, puis relire le mot et obtenir une valeur différente (parce qu'une autre unité centrale l'a modifiée). Lorsqu'elle est organisée correctement, cette propriété constitue la base de la communication entre processeurs : un processeur écrit des données dans la mémoire et un autre les lit.

Dans la plupart des cas, les systèmes d'exploitation multiprocesseurs sont des systèmes d'exploitation normaux. Ils traitent les appels système, gèrent la mémoire, fournissent un système de fichiers et gèrent les périphériques 1/0. Néanmoins, il existe certains domaines dans lesquels ils possèdent des caractéristiques uniques. Il s'agit notamment de la synchronisation des processus, de la gestion des ressources et de l'ordonnancement. Ci-dessous, nous allons d'abord jeter un bref coup d'oeil au matériel multiprocesseur, puis nous passerons aux problèmes de ces systèmes d'exploitation.

8.1.1 Matériel multiprocesseur

Bien que tous les multiprocesseurs aient la propriété que chaque CPU puisse adresser toute la mémoire, certains multiprocesseurs ont la propriété supplémentaire que chaque mot de mémoire peut être lu aussi rapidement que tous les autres mots de mémoire. Ces machines sont appelées multiprocesseurs **UMA (Uniform Memory Access)**. En revanche, les multiprocesseurs **NUMA (Nonuniform Memory Access)** ne possèdent pas cette propriété. Nous verrons plus loin pourquoi cette différence existe. Nous examinerons d'abord les multiprocesseurs UMA, puis les multiprocesseurs NUMA.

Multiprocesseurs UMA avec architectures à base de bus

Les multiprocesseurs les plus simples sont basés sur un seul bus, comme l'illustre la figure 8-2(a). Deux UC ou plus et un ou plusieurs modules de mémoire utilisent tous le même bus pour communiquer. Lorsqu'une unité centrale veut lire un mot de mémoire, elle vérifie d'abord si le bus est occupé. Si le bus est inoccupé, l'unité centrale de traitement place l'adresse du mot qu'elle souhaite sur le bus, active quelques signaux de commande et attend que la mémoire place le mot souhaité sur le bus.

Si le bus est occupé lorsqu'une unité centrale veut lire ou écrire dans la mémoire, l'unité centrale attend simplement que le bus devienne inactif. C'est là que réside le problème de cette conception. Avec deux ou trois CPU, la contention pour le bus sera gérable ; avec 32 ou 64, elle sera insupportable. Le système sera totalement limité par la bande passante du bus, et la plupart des CPU seront inactives la plupart du temps.

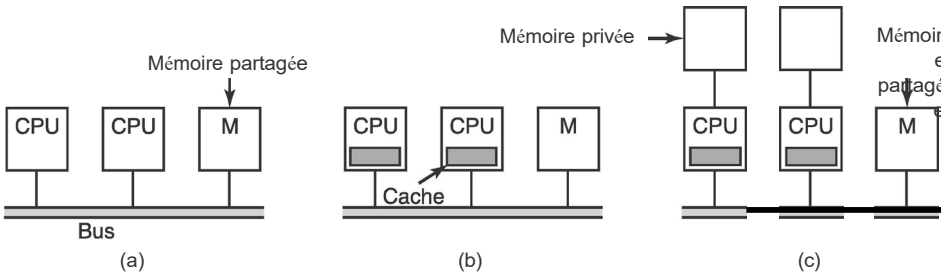


Figure 8-2. Trois multiprocesseurs à base de bus. (a) Sans mise en cache. (b) Avec mise en cache. (c) Avec mise en cache et mémoires privées.

La solution à ce problème consiste à ajouter un cache à chaque unité centrale, comme illustré à la Fig. 8-2(b). Le cache peut se trouver à l'intérieur de la puce du processeur, à côté de la puce du processeur, sur la carte du processeur, ou une combinaison des trois. Étant donné que de nombreuses lectures peuvent maintenant être satisfaites à partir du cache local, il y aura beaucoup moins de trafic sur le bus et le système pourra prendre en charge davantage de CPU. En général, la mise en cache ne se fait pas sur la base d'un mot individuel mais sur la base de blocs de 32 ou 64 octets. Lorsqu'un mot est référencé, son bloc entier, appelé **ligne de cache,** est récupéré dans le cache de l'unité centrale qui le touche.

Chaque bloc de cache est marqué comme étant soit en lecture seule (auquel cas il peut être présent dans plusieurs caches en même temps), soit en lecture-écriture (auquel cas il ne peut être présent dans aucun autre cache). Si une unité centrale tente d'écrire un mot qui se trouve dans un ou plusieurs caches distants, le matériel du bus détecte l'écriture et envoie un signal sur le bus pour informer tous les autres caches de l'écriture. Si les autres caches ont une copie "propre", c'est-à-dire une copie exacte de ce qui se trouve en mémoire, ils peuvent simplement écarter leurs copies et laisser l'auteur récupérer le bloc de cache en mémoire avant de le modifier. Si une autre antémémoire possède une copie "sale" (c'est-à-dire modifiée), elle doit soit la réécrire en mémoire avant que l'écriture puisse avoir lieu, soit la transférer directement à l'auteur via le bus. Cet ensemble de règles est appelé **protocole de cohérence du cache** et n'est qu'un exemple parmi d'autres.

Une autre possibilité encore est la conception de la figure 8-2(c), dans laquelle chaque unité centrale dispose non seulement d'un cache, mais aussi d'une mémoire locale privée à laquelle elle accède par un bus (privé) dédié. Pour utiliser cette configuration de manière optimale, le compilateur doit placer tout le texte du programme, les chaînes de caractères, les constantes et autres données en lecture seule, les piles et les variables locales dans les mémoires privées. La mémoire partagée n'est alors utilisée que pour les variables partagées accessibles en écriture. Dans la plupart des cas, ce placement judicieux réduira considérablement le trafic du bus, mais il nécessite une coopération active du compilateur.

Multiprocesseurs UMA utilisant des commutateurs Crossbar

Même avec la meilleure mise en cache, l'utilisation d'un seul bus limite la taille d'un multiprocesseur UMA à environ 16 ou 32 CPU. Pour aller au-delà, un autre type de réseau d'interconnexion est nécessaire. Le circuit le plus simple pour connecter n CPU à k

La mémoire est le **commutateur crossbar,** illustré à la Fig. 8-3. Les commutateurs crossbar sont utilisés depuis des décennies dans les centraux téléphoniques pour connecter un groupe de lignes entrantes à un ensemble de lignes sortantes de manière arbitraire.

À chaque intersection d'une ligne horizontale (entrante) et verticale (sortante) se trouve un **point de croisement. Un** point de croisement est un petit interrupteur électronique qui peut être ouvert ou fermé électriquement, selon que les lignes horizontales et verticales doivent être connectées ou non. Sur la figure 8-3(a), nous voyons trois points de croisement fermés simultanément, ce qui permet de connecter les paires (CPU, mémoire) (010, 000), (101, 101) et (110, 010) en même temps. De nombreuses autres combinaisons sont également possibles. En fait, le nombre de combinaisons est égal au nombre de façons différentes dont huit tours peuvent être placées en toute sécurité sur un échiquier.

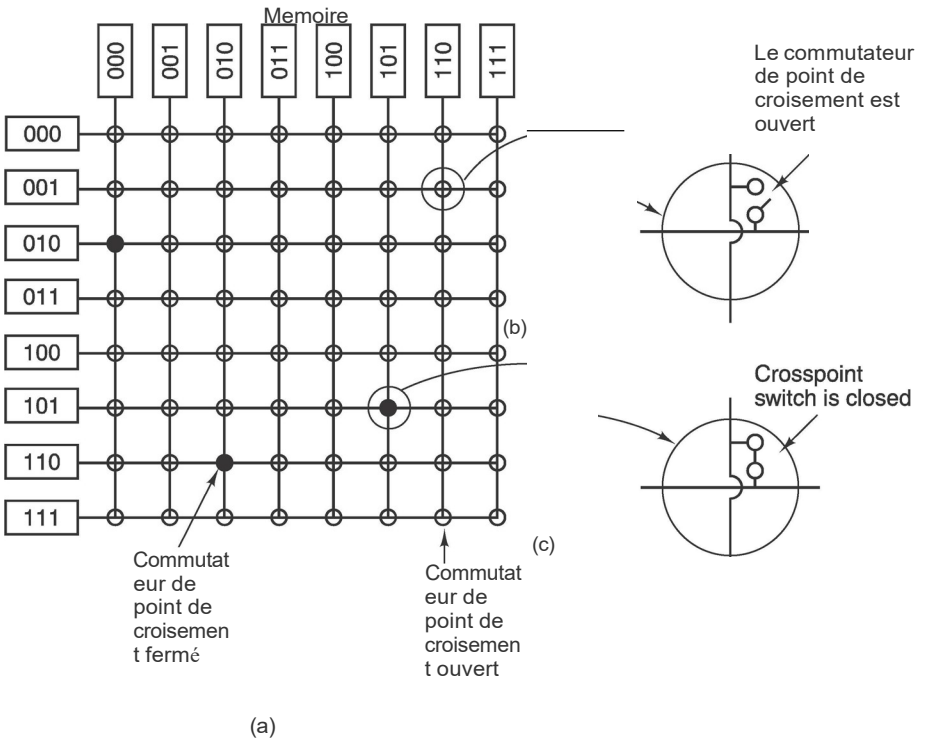


Figure 8-3. (a) Un commutateur à barres croisées 8 x 8. (b) Un point de croisement ouvert. (c) Un point de croisement fermé.

L'une des plus belles propriétés du commutateur crossbar est qu'il s'agit d'un **réseau non bloquant,** ce qui signifie qu'aucune unité centrale ne se voit refuser la connexion dont elle a besoin parce qu'un point de croisement ou une ligne est déjà occupé (en supposant que le module de mémoire lui-même soit disponible). Toutes les interconnexions ne possèdent pas cette belle propriété. En outre, aucune planification préalable n'est nécessaire. Même si sept connexions arbitraires sont déjà établies, il est toujours possible de connecter l'unité centrale restante à la mémoire restante.

La contestation de la mémoire est toujours possible, bien sûr, si deux CPU veulent accéder au même module en même temps. Néanmoins, en partitionnant la mémoire en *n* unités, la contention est réduite d'un facteur *n par* rapport au modèle de la Fig. 8-2.

L'une des pires propriétés du commutateur à barres croisées est le fait que le nombre de points de croisement croît comme n^2. Avec 1000 CPU et 1000 modules de mémoire, nous avons besoin d'un million de points de croisement. Un commutateur crossbar de cette taille n'est pas réalisable. Néanmoins, pour les systèmes de taille moyenne, une conception crossbar est réalisable.

Multiprocesseurs UMA utilisant des réseaux de commutation à plusieurs étages

Une conception de multiprocesseur complètement différente est basée sur l'humble commutateur 2 x 2 illustré à la Fig. 8-4(a). Ce commutateur possède deux entrées et deux sorties. Les messages arrivant sur l'une des lignes d'entrée peuvent être commutés sur l'une des lignes de sortie. Pour nos besoins, les messages peuvent contenir jusqu'à quatre parties, comme le montre la Fig. 8-4(b). Le champ *Module* indique la mémoire à utiliser. L'*adresse* spécifie une adresse dans un module. Le champ *Opcode* indique l'opération, telle que READ ou WRITE. Enfin, le champ *Value,* facultatif, peut contenir un opérande, tel qu'un mot de 32 bits à écrire lors d'un WRITE. Le commutateur inspecte le champ *Module* et l'utilise pour déterminer si le message doit être envoyé sur *X* ou sur *Y.*

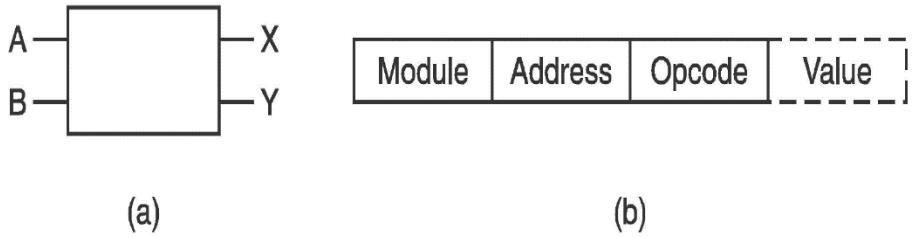


Figure 8-4. (a) Un commutateur 2 x 2 avec deux lignes d'entrée, *A* et *B,* et deux lignes de sortie, *X* et *Y.* (b) Un format de message.

Nos commutateurs 2 x 2 peuvent être agencés de nombreuses façons pour construire de plus grands **réseaux de commutation à plusieurs étages** (Adams et al., 1987 ; Garofalakis et Stergiou, 2013 ; et Kumar et Reddy, 1987). L'une des possibilités est le **réseau oméga de** classe bovine, sans fioritures, illustré à la figure 8-5. Ici, nous avons connecté huit CPU à huit mémoires à l'aide de 12 commutateurs. Plus généralement, pour *n* CPU et *n* mémoires, nous aurions besoin de $\log_2 n$ étages, avec *n/2* commutateurs par étage, pour un total de $(n/2) \log_2 n$ commutateurs, ce qui est bien mieux que n^2 points de croisement, surtout pour les grandes valeurs de *n.*

Le schéma de câblage du réseau oméga est souvent appelé "**mélange parfait",** car le mélange des signaux à chaque étape ressemble à un jeu de cartes coupé en deux, puis mélangé carte par carte. Pour voir comment le réseau oméga fonctionne, supposons que l'unité centrale 011 souhaite lire un mot du module de mémoire 110. L'unité centrale envoie un message READ au commutateur 1D contenant la valeur 110 dans le champ *Module*. Le commutateur prend le premier bit (c'est-à-dire le plus à gauche) de 110 et l'utilise pour le routage. Un AO achemine vers la sortie supérieure et un 1 achemine vers la sortie inférieure. Comme ce bit est un 1, le message est acheminé vers 2D via la sortie inférieure.

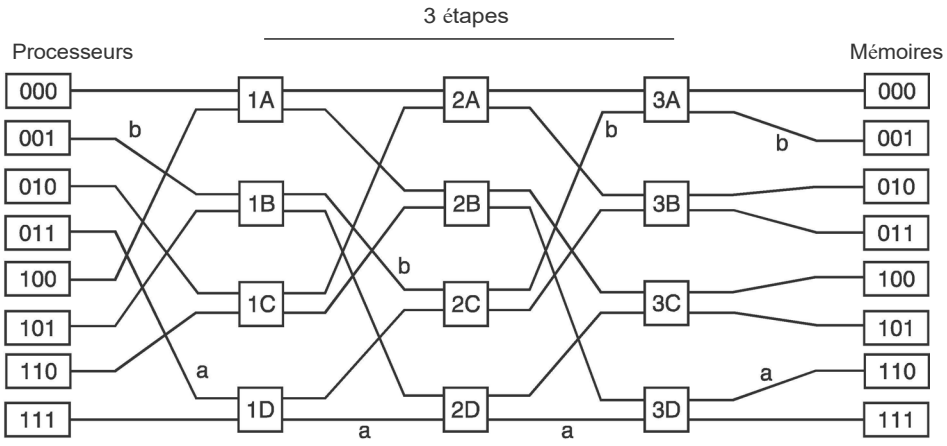


Figure 8-5. Un réseau de commutation oméga.

Tous les commutateurs du deuxième étage, y compris 2D, utilisent le deuxième bit pour le routage. Celui-ci étant également un 1, le message est maintenant acheminé par la sortie inférieure vers 3D. Ici, le troisième bit est testé et se révèle être un 0. Par conséquent, le message sort par la sortie supérieure et arrive à la mémoire 110, comme souhaité. Le chemin suivi par ce message est marqué sur la Fig. 8-5 par la lettre *a*.

Lorsque le message se déplace sur le réseau de commutation, les bits situés à l'extrémité gauche du numéro de module ne sont plus nécessaires. Ils peuvent être utilisés à bon escient en y enregistrant le numéro de la ligne entrante, afin que la réponse puisse retrouver son chemin. Pour le chemin *a, les* lignes entrantes sont respectivement O (entrée supérieure vers ID), 1 (entrée inférieure vers 2D) et 1 (entrée inférieure vers 3D). La réponse est renvoyée en utilisant 011, en la lisant seulement de droite à gauche cette fois.

Au même moment, l'unité centrale 001 souhaite écrire un mot dans le module de mémoire 001. Un processus analogue se produit ici, le message étant acheminé via les sorties supérieure, supérieure et inférieure, respectivement, marquées par la lettre *b.* Lorsqu'il arrive, son champ *Module* indique 001, représentant le chemin qu'il a emprunté. Comme ces deux requêtes n'utilisent pas les mêmes commutateurs, lignes ou modules de mémoire, elles peuvent se dérouler en parallèle.

Considérons maintenant ce qui se passerait si l'unité centrale 000 voulait simultanément accéder au module de mémoire 000. Sa demande entrerait en conflit avec celle de l'unité centrale 001 au niveau du commutateur 3A. L'un des deux devrait alors attendre. Contrairement au commutateur crossbar, le réseau oméga est un **réseau de blocage.** Tous les ensembles de demandes ne peuvent pas être traités simultanément. Des conflits peuvent se produire pour l'utilisation d'un fil ou d'un commutateur, ainsi qu'entre les demandes *à la* mémoire et les réponses de la mémoire.

Comme il est hautement souhaitable de répartir les références mémoire uniformément entre les modules, une technique courante consiste à utiliser les bits de poids faible comme numéro de module. Considérons, par exemple, un espace d'adressage orienté octet pour un ordinateur qui

accède généralement à des mots complets de 32 bits. Les 2 bits de poids faible sont généralement 00, mais les 3 bits suivants sont répartis uniformément. En utilisant ces 3 bits comme numéro de module, les mots consécutifs seront dans des modules consécutifs. Un système de mémoire dans lequel des mots consécutifs se trouvent dans des modules différents est dit **entrelacé.** Les mémoires entrelacées maximisent le parallélisme car la plupart des références mémoire sont des adresses consécutives. Il est également possible de concevoir des réseaux de commutation non bloquants et offrant plusieurs chemins de chaque unité centrale à chaque module de mémoire pour mieux répartir le trafic.

Multiprocesseurs NUMA

Les multiprocesseurs UMA à bus unique sont généralement limités à quelques dizaines de processeurs, et les multiprocesseurs à barres croisées ou à commutation nécessitent beaucoup de matériel (coûteux) et ne sont pas beaucoup plus grands. Pour dépasser les 100 unités centrales, il faut faire des concessions. En général, il s'agit de l'idée que tous les modules de mémoire ont le même temps d'accès. Cette concession conduit à l'idée des multiprocesseurs NUMA, comme mentionné ci-dessus. Comme leurs cousins UMA, ils fournissent un espace d'adressage unique pour toutes les unités centrales, mais contrairement aux machines UMA, l'accès aux modules de mémoire locaux est plus rapide que l'accès aux modules distants. Ainsi, tous les programmes UMA fonctionneront sans changement sur les machines NUMA, mais les performances seront moins bonnes que sur une machine UMA.

Les machines NUMA présentent trois caractéristiques essentielles qu'elles possèdent toutes et qui, ensemble, les distinguent des autres multiprocesseurs :

1. Il existe un seul espace d'adressage visible par tous les processeurs.
2. L'accès à la mémoire distante se fait via les instructions LOAD et STORE.
3. L'accès à la mémoire distante est plus lent que l'accès à la mémoire locale.

Lorsque le temps d'accès à la mémoire distante n'est pas caché (parce qu'il n'y a pas de cache), le système est appelé **NC-NUMA (Non Cache-coherent NUMA).** Lorsque les caches sont cohérents, le système est appelé **CC-NUMA (Cache-Coherent NUMA).**

Une approche populaire pour construire de grands multiprocesseurs CC-NUMA est le **multiprocesseur à base de répertoires. L**'idée est de maintenir une base de données indiquant où se trouve chaque ligne de cache et quel est son état. Lorsqu'une ligne de cache est référencée, la base de données est interrogée pour savoir où elle se trouve et si elle est propre ou sale. Comme cette base de données est interrogée à chaque instruction qui touche la mémoire, elle doit être conservée dans un matériel spécialisé extrêmement rapide qui peut répondre en une fraction de cycle de bus. Pour rendre l'idée d'un multiprocesseur basé sur un répertoire un peu plus concrète,

Considérons, à titre d'exemple simple (hypothétique), un système de 256 nœuds, chaque nœud étant constitué d'un CPU et de 16 Mo de RAM connectés au CPU par un bus local. La mémoire totale est de 2^{32} octets et elle est divisée en 2^{26} lignes de cache de 64 octets chacune. La mémoire est allouée statiquement entre les noeuds, avec 0-16M dans le noeud 0, 16M-32M dans le noeud 1, etc. Les nœuds sont reliés par un réseau d'interconnexion,

comme le montre la Fig. 8-6(a). Chaque nœud détient également les entrées de répertoire pour les 2^{18} lignes de cache de 64 octets qui composent sa mémoire de 2^{24} octets. Pour l'instant, nous supposerons qu'une ligne peut être conservée dans un seul cache au maximum.

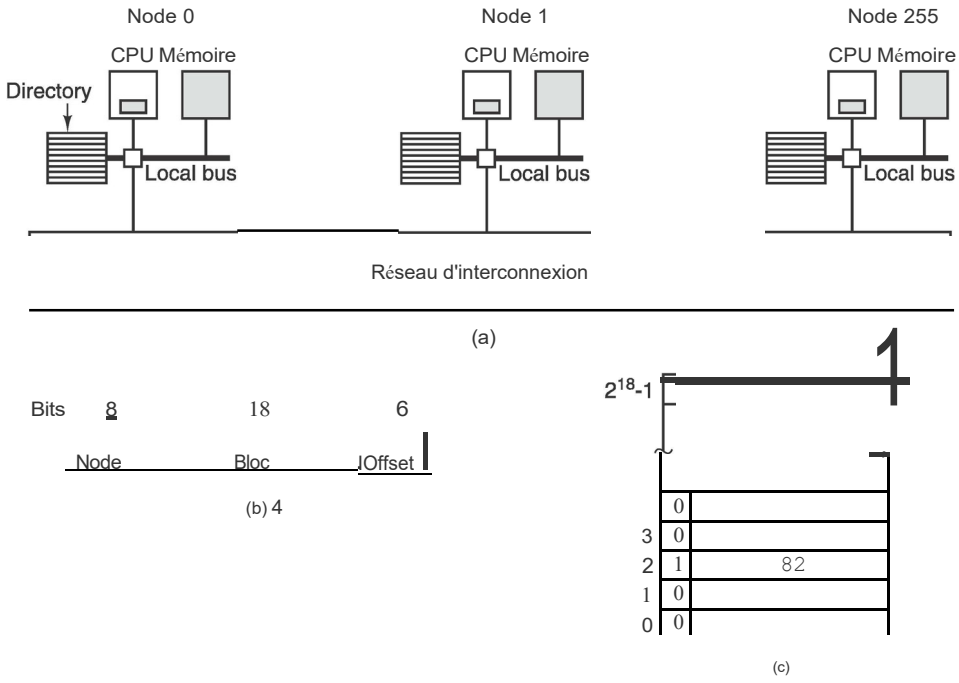


Figure 8-6. (a) Un multiprocesseur à 256 nœuds basé sur un répertoire. (b) Division d'une adresse mémoire de 32 bits en champs. (c) Le répertoire au nœud 36.

Pour voir comment le répertoire fonctionne, traçons une instruction LOAD du CPU 20 qui fait référence à une ligne mise en cache. Tout d'abord, le processeur émettant l'instruction la présente à son MMU, qui la traduit en une adresse physique, disons 0x24000108. Le MMU divise cette adresse en trois parties comme le montre la Fig. 8-6(b). En décimal, les trois parties sont le noeud 36, la ligne 4 et l'offset 8. La MMU voit que le mot de mémoire référencé provient du noeud 36 et non du noeud 20, elle envoie donc un message de requête via le réseau d'interconnexion au noeud d'origine de la ligne, 36, pour demander si sa ligne 4 est mise en cache, et si oui, où.

Lorsque la demande arrive au noeud 36 par le réseau d'interconnexion, elle est acheminée vers le matériel du répertoire. Le matériel indexe dans sa table de 2^{18} entrées, une pour chacune de ses lignes de cache, et extrait l'entrée 4. D'après la Fig. 8-6(c), nous voyons que la ligne n'est pas mise en cache, donc le matériel lance une extraction de la ligne 4 à partir de la RAM locale et, après son arrivée, la renvoie au noeud 20. Il met alors à jour l'entrée 4 du répertoire pour indiquer que la ligne est maintenant mise en cache au noeud 20.

Considérons maintenant une deuxième requête, cette fois-ci concernant la ligne 2 du nœud 36. D'après la Fig. 8-6(c), nous voyons que cette ligne est mise en cache au nœud 82. À ce stade, le matériel pourrait mettre à jour l'entrée 2 du répertoire pour indiquer que la ligne se trouve maintenant au nœud 20, puis envoyer un message au nœud 82 pour lui demander de transmettre la ligne au nœud 20 et d'invalider son cache. Notez que même un soi-disant "multiprocesseur à mémoire partagée" a beaucoup de messages qui passent sous le capot.

En guise d'aparté, calculons la quantité de mémoire occupée par les répertoires. Chaque nœud dispose de 16 Mo de RAM et de 2^{18} entrées de 9 bits pour garder la trace de cette RAM. Ainsi, la surcharge du répertoire est d'environ 9×2^{18} bits divisés par 16 Mo ou environ 1,76 %, ce qui est généralement acceptable (bien qu'il doive s'agir de mémoire à haute vitesse, ce qui augmente son coût, bien sûr). Même avec des lignes de cache de 32 octets, l'excédent ne serait que de 4 %. Avec des lignes de cache de 128 octets, il serait inférieur à 1 %.

Une limitation évidente de cette conception est qu'une ligne ne peut être mise en cache que sur un seul nœud. Pour permettre aux lignes d'être mises en cache sur plusieurs nœuds, nous aurions besoin d'un moyen de les localiser toutes, par exemple, pour les invalider ou les mettre à jour lors d'une écriture. Sur de nombreux processeurs multicœurs, une entrée de répertoire consiste donc en un *vecteur* binaire avec un bit par cœur. Un "1" indique que la ligne de cache est présente sur le cœur, et un "O" qu'elle ne l'est pas. De plus, chaque entrée de répertoire contient généralement quelques bits supplémentaires. Par conséquent, le coût de la mémoire du répertoire augmente considérablement.

Puces multicœurs

À mesure que la technologie de fabrication des puces s'améliore, les transistors deviennent de plus en plus petits et il est possible d'en mettre de plus en plus sur une puce. Cette observation empirique est souvent appelée **loi de Moore,** du nom du cofondateur d'Intel, Gordon Moore, qui l'a remarquée le premier. En 1971, l'Intel 8080 contenait un

transistors, tandis que les processeurs Xeon Nehalem-EX comptent plus de 2 milliards de transistors.

Une question évidente est : "Que faire de tous ces transistors ?" Comme nous l'avons évoqué dans la Sec. 1.3.1, une option consiste à ajouter des mégaoctets de cache à la puce. Cette option est sérieuse, et les puces avec 4-32 Mo de cache sur puce sont courantes. Mais à un moment donné, augmenter la taille du cache peut faire passer le taux de réussite de 99 % à 99,5 % seulement, ce qui n'améliore pas beaucoup les performances des applications.

L'autre option consiste à placer deux ou plusieurs processeurs complets, généralement appelés **cœurs,** sur la même puce (techniquement, sur le même **dé).** Les puces à double cœur, à quadruple cœur et à octa cœur sont déjà courantes et vous pouvez même acheter des puces avec des centaines de cœurs. Pour l'instant, aucune hypothèse n'est faite sur le fait que tous les cœurs sont identiques ou répartis sur la puce. Par exemple, l'Intel Xeon 2651 possède 12 cœurs physiques hyper threadés, ce qui donne 24 cœurs virtuels. Chacun des 12 cœurs physiques dispose de 32 Ko de cache d'instructions LI et de 32 Ko de cache de données LI. Chacun dispose également de 256 Ko de cache L2. Enfin, les 12 cœurs partagent 30 Mo de cache L3.

Si les processeurs peuvent ou non partager des caches (voir, par exemple, la figure 1-8), ils partagent toujours la mémoire principale, et cette mémoire est cohérente dans le sens où il y a toujours une valeur unique pour chaque mot de mémoire. Des circuits matériels spéciaux rendent

s'assurer que si un mot est présent dans deux ou plusieurs caches et que l'une des unités centrales modifie ce mot, il est automatiquement et atomiquement supprimé de tous les caches afin de maintenir la cohérence. Ce processus est connu sous le

Le résultat de cette conception est que les puces multicœurs ne sont que de très sors. En fait, les puces multicœurs sont parfois appelées **CMP (Chip MultiProcessors).** D'un point de vue logiciel, les CMP ne sont pas vraiment très différents des bus.
ou des multiprocesseurs à base de multiprocesseurs qui utilisent des réseaux Cependant, il y a quelques différences. Pour commencer, sur un multiprocesseur basé sur un bus, les processeurs ont leur comme dans la et aussi comme dans la de Fig. 8-2(b)
Fig. l-8(b). La conception du cache partagé de la figure l-8(a), qu'Intel utilise dans un grand nombre de ses processeurs, n'existe pas dans d'autres multiprocesseurs. Un cache L2 ou L3 partagé peut affecter les performances si une quantité de mémoire cache cette conception permet au chasseur de cache de prendre D'un autre côté, le Le cache partagé permet également à un cœur gourmand de nuire aux autres

La tolérance aux pannes est un domaine dans lequel les CMP diffèrent de leurs cousins plus grands. Soyez
Les UC étant étroitement connectées, les défaillances des composants partagés peuvent mettre hors service plusieurs UC à la fois, ce qui est peu probable avec les multiprocesseurs traditionnels.

Outre les puces multicœurs symétriques, où tous les cœurs sont identiques, une autre catégorie courante de puce multicœur est le **système sur puce (SoC).** Ces puces possèdent un ou plusieurs processeurs principaux, mais aussi des cœurs à usage spécifique, tels que des décodeurs vidéo et audio, des cryptoprocesseurs, des interfaces réseau, et bien d'autres encore, ce qui donne un système informatique complet sur une puce.

Puces multicœurs

Multicore signifie simplement "plus d'un cœur", mais lorsque le nombre de cœurs dépasse largement la portée du comptage des doigts, nous utilisons un autre nom. Les **puces manycore** sont des multicœurs qui contiennent des dizaines, des centaines, voire des milliers de cœurs. Bien qu'il n'y ait pas de seuil précis au-delà duquel un multicœur devient un manycore, on peut dire que vous avez probablement un manycore si vous ne vous souciez plus de perdre un ou deux cœurs.

Les cartes additionnelles d'accélérateur comme le Xeon Phi d'Intel possèdent plus de 60 cœurs x86. D'autres fournisseurs ont déjà franchi la barre des 100 cœurs avec différents types de cœurs. Un millier de cœurs à usage général sont peut-être en passe d'arriver. Il n'est pas facile d'imaginer ce que l'on peut faire avec un millier de cœurs, et encore moins comment les programmer.

Un autre problème avec un nombre vraiment important de cœurs est que les mécanismes nécessaires pour maintenir la cohérence de leurs caches deviennent très compliqués et très coûteux. De nombreux ingénieurs craignent que la cohérence des caches ne puisse pas s'étendre à plusieurs centaines de cœurs. Certains préconisent même de l'abandonner complètement. Ils craignent que le coût des protocoles de cohérence dans le matériel soit si élevé que tous ces nouveaux cœurs brillants n'amélioreront pas beaucoup les performances car le processeur sera trop occupé à maintenir les caches dans un état cohérent. Pire encore, il devrait dépenser beaucoup trop de mémoire sur le répertoire (rapide) pour le faire. C'est ce qu'on appelle le **mur de cohérence.**

Prenons, par exemple, notre solution de cohérence du cache basée sur un répertoire, présentée ci-dessus. Si chaque entrée du répertoire contient un vecteur binaire pour indiquer quels cœurs contiennent une ligne de cache particulière, l'entrée du répertoire pour un processeur avec 1024 cœurs aura une longueur d'au moins 128 octets. Comme les lignes de cache elles-mêmes sont rarement plus grandes que 128 octets, cela conduit à une situation délicate où l'entrée du répertoire est plus grande que la ligne de cache qu'elle suit. Ce n'est probablement pas ce que nous voulons.

Certains ingénieurs affirment que le seul modèle de programmation qui a prouvé sa capacité à s'adapter à un très grand nombre de processeurs est celui qui utilise le passage de messages et la mémoire distribuée - et c'est ce à quoi nous devons nous attendre dans les futures puces manycore. Des processeurs expérimentaux tels que le SCC à 48 cœurs d'Intel ont déjà abandonné la cohérence du cache et fourni un support matériel pour un passage de messages plus rapide. D'autre part, d'autres processeurs assurent toujours la cohérence, même avec un nombre élevé de cœurs. Des modèles hybrides sont également possibles. Par exemple, une puce de 1024 cœurs peut être partitionnée en 64 îlots de 16 cœurs cohérents avec le cache chacun, tout en abandonnant la cohérence du cache entre les îlots.

Les milliers de cœurs ne sont même plus si spéciaux. Le nombre de cœurs le plus courant aujourd'hui, les processeurs graphiques, se retrouvent dans pratiquement tous les systèmes informatiques qui ne sont pas intégrés et qui possèdent un écran. Un **GPU** est un processeur doté d'une mémoire dédiée et, littéralement, de milliers de cœurs minuscules. Par rapport aux processeurs à usage général, les GPU consacrent une plus grande partie de leur budget de transistors aux circuits qui effectuent les calculs et moins aux caches et à la logique de contrôle. Ils sont très performants pour de nombreux petits calculs effectués en parallèle, comme le rendu des polygones dans les applications graphiques. Ils ne sont pas aussi performants pour les tâches en série. Ils sont également difficiles à programmer. Si les GPU peuvent être utiles pour les systèmes d'exploitation (par exemple, pour le cryptage ou le traitement du trafic réseau), il est peu probable qu'une grande partie du système d'exploitation lui-même fonctionne sur les GPU.

D'autres tâches informatiques *sont* de plus en plus prises en charge par les GPU, notamment les tâches exigeantes en termes de calcul qui sont courantes dans l'informatique scientifique. Le terme utilisé pour le traitement polyvalent sur les GPU est - vous l'avez deviné - **GPGPU.** Heureusement, la programmation efficace des GPU est extrêmement difficile et nécessite des langages de programmation spéciaux comme **OpenGL** ou **CUDA**, le langage propriétaire de NVIDIA**.** Une différence importante entre la programmation des GPU et celle des processeurs universels est que les GPU sont essentiellement des machines à "instruction unique et données multiples", ce qui signifie qu'un grand nombre de cœurs exécutent exactement la même instruction mais sur des données différentes. Ce modèle de programmation est idéal pour le parallélisme des données, mais pas toujours pratique pour d'autres styles de programmation (comme le parallélisme des tâches).

Multicores hétérogènes

Certaines puces intègrent un GPU et un certain nombre de cœurs à usage général sur la même puce. De même, de nombreux SoC contiennent des cœurs à usage général en plus d'un ou plusieurs processeurs à usage spécifique. Les systèmes qui intègrent plusieurs races différentes

de processeurs dans une seule puce sont collectivement connus sous le nom de processeurs **multicœurs hétérogènes**. Un exemple de processeur multicœur hétérogène est la gamme de processeurs de réseau IXP introduite par Intel en 2000 et régulièrement mise à jour avec les dernières technologies. Les processeurs de réseau contiennent généralement un seul noyau de contrôle général (par exemple, un processeur ARM exécutant Linux) et plusieurs dizaines de processeurs de flux hautement spécialisés qui sont vraiment bons pour traiter les paquets de travail du réseau et pas grand-chose d'autre. Ils sont couramment utilisés dans les équipements de réseau, tels que les routeurs et les pare-feu. Pour acheminer les paquets du réseau, vous n'avez probablement pas besoin d'opérations à virgule flottante, c'est pourquoi, dans la plupart des modèles, les processeurs de flux n'ont pas du tout d'unité à virgule flottante. D'autre part, les réseaux à haut débit dépendent fortement d'un accès rapide à la mémoire (pour lire les données des paquets) et les processeurs de flux disposent d'un matériel spécial pour rendre cela possible.

Dans les exemples précédents, les systèmes étaient clairement hétérogènes. Les processeurs de flux et les processeurs de contrôle sur les IXP sont des bêtes complètement différentes avec des jeux d'instructions différents. Il en va de même pour le GPU et les cœurs à usage général. Cependant, il est également possible d'introduire de l'hétérogénéité tout en conservant le même jeu d'instructions. Par exemple, un CPU peut avoir un petit nombre de "gros" cœurs, avec des pipelines profonds et éventuellement des vitesses d'horloge élevées, et un plus grand nombre de "petits" cœurs plus simples, moins puissants et fonctionnant peut-être à des fréquences plus basses. Les cœurs puissants sont nécessaires pour exécuter un code nécessitant un traitement séquentiel rapide, tandis que les petits cœurs sont utiles pour les tâches qui peuvent être exécutées efficacement en parallèle. Un exemple d'architecture hétérogène de ce type est la famille de processeurs big.LITTLE d'ARM.

Programmation avec plusieurs cœurs

Comme cela s'est souvent produit dans le passé, le matériel a une longueur d'avance sur le logiciel. Si les puces multicœurs sont désormais disponibles, notre capacité à écrire des applications pour elles ne l'est pas. Les langages de programmation actuels sont mal adaptés à l'écriture de programmes hautement parallèles et les bons compilateurs et outils de débogage sont rares sur le terrain. Peu de programmeurs ont une expérience de la programmation parallèle et la plupart ne savent pas comment diviser un travail en plusieurs paquets qui peuvent être exécutés en parallèle. La synchronisation, l'élimination des conditions de course et l'évitement des blocages sont des choses dont les rêves les plus terribles sont faits, mais malheureusement les performances en souffrent terriblement si elles ne sont pas bien gérées. Les sémaphores ne sont pas la solution.

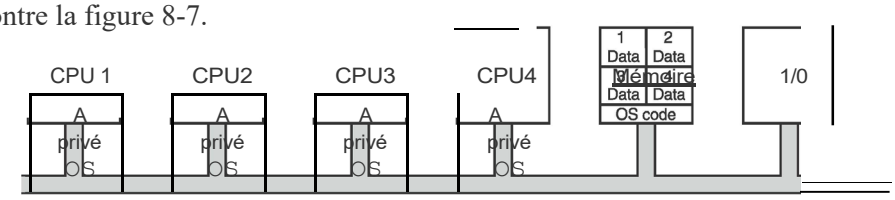
Au-delà de ces problèmes de démarrage, il est loin d'être évident de savoir quel type d'application a réellement besoin de centaines, voire de milliers, de cœurs, en particulier dans les environnements domestiques. Dans les grandes fermes de serveurs, par contre, il y a souvent beaucoup de travail pour un grand nombre de cœurs. Par exemple, un serveur populaire peut facilement utiliser un cœur différent pour chaque demande du client. De la même manière, les fournisseurs de cloud computing évoqués dans le chapitre précédent peuvent absorber les cœurs pour fournir un grand nombre de machines virtuelles à louer à des clients à la recherche de puissance de calcul à la demande.

8.1.2 Types de systèmes d'exploitation multiprocesseurs

Passons maintenant du matériel multiprocesseur aux logiciels multiprocesseurs, en particulier aux systèmes d'exploitation multiprocesseurs. Différentes approches sont possibles. Nous en étudierons trois ci-dessous. Notez que toutes s'appliquent aussi bien aux systèmes multicœurs qu'aux systèmes à processeurs discrets.

Chaque CPU a son propre système d'exploitation

La façon la plus simple d'organiser un système d'exploitation multiprocesseur est de diviser statiquement la mémoire en autant de partitions qu'il y a de CPU et de donner à chaque CPU sa propre mémoire privée et sa propre copie privée du système d'exploitation. En fait, les *n* UC fonctionnent alors comme *n* ordinateurs indépendants. Une optimisation évidente consiste à permettre à toutes les unités centrales de partager le code du système d'exploitation et de faire des copies privées des seules structures de données du système d'exploitation, comme le montre la figure 8-7.



'--sus

Figure 8-7. Partitionnement de la mémoire du multiprocesseur entre quatre CPU, mais partageant une une seule copie du code du système d'exploitation. Les cases marquées Data sont les données privées du système d'exploitation pour chaque CPU.

Ce schéma est encore meilleur que celui qui consiste à avoir *n* ordinateurs séparés, car il permet à toutes les machines de partager un ensemble de disques et d'autres périphériques 1/0, et il permet également de partager la mémoire de manière flexible. Par exemple, même en cas d'allocation statique de la mémoire, une unité centrale de traitement peut se voir attribuer une portion extra-large de la mémoire afin de pouvoir traiter efficacement les gros programmes. En outre, les processus peuvent communiquer efficacement entre eux en permettant à un producteur d'écrire des données directement dans la mémoire et à un consommateur de les récupérer à l'endroit où le producteur les a écrites. Cependant, du point de vue des systèmes d'exploitation, le fait que chaque CPU ait son propre système d'exploitation est aussi primitif que possible.

Il convient de mentionner quatre aspects de cette conception qui ne sont peut-être pas évidents. Tout d'abord, lorsqu'un processus fait un appel système, celui-ci est attrapé et traité sur sa propre unité centrale en utilisant les structures de données dans les tables de ce système d'exploitation.

Deuxièmement, puisque chaque système d'exploitation possède ses propres tables, il possède également son propre ensemble de processus qu'il ordonnance lui-même. Il n'y a pas de partage des processus. Si un utilisateur se connecte au CPU 1, tous ses processus s'exécutent sur le CPU 1. Par conséquent, il peut arriver que l'unité centrale 1 soit inactive alors que l'unité centrale 2 est chargée de travail.

Troisièmement, il n'y a pas de partage des pages physiques. Il peut arriver que l'unité centrale l ait des pages en réserve alors que l'unité centrale 2 effectue une pagination continue. Il n'y a aucun moyen pour l'unité centrale 2 d'emprunter des pages à l'unité centrale 1 puisque l'allocation de mémoire est fixe.

Quatrièmement, et c'est le pire, si le système d'exploitation maintient un cache tampon des blocs de disque récemment utilisés, chaque système d'exploitation le fait indépendamment des autres. Il peut donc arriver qu'un certain bloc disque soit présent et sale dans plusieurs caches tampons en même temps, ce qui entraîne des résultats incohérents. La seule façon d'éviter ce problème est d'éliminer les caches tampons. Ce n'est pas difficile à faire, mais cela nuit considérablement aux performances.

Pour ces raisons, ce modèle est rarement utilisé dans les systèmes de production, bien qu'il ait été utilisé aux débuts des multiprocesseurs, lorsque l'objectif était de porter les systèmes d'exploitation existants sur un nouveau multiprocesseur aussi rapidement que possible. Dans le domaine de la recherche, ce modèle fait un retour en force, mais avec toutes sortes de variantes. Il y a quelque chose à dire pour garder les systèmes d'exploitation complètement séparés. Si tout l'état de chaque processeur est conservé localement, il n'y a que peu ou pas de partage pouvant entraîner des problèmes de cohérence ou de verrouillage. À l'inverse, si plusieurs processeurs doivent accéder et modifier la même table de processus, le verrouillage devient rapidement compliqué (et crucial pour les performances). Nous en dirons plus à ce sujet lorsque nous aborderons le modèle de multiprocesseur symétrique ci-dessous.

Multiprocesseurs maître-esclave

Un deuxième modèle est illustré à la Fig. 8-8. Ici, une copie du système d'exploitation et de ses tables est présente sur l'unité centrale 1 et sur aucune des autres. Tous les appels système sont redirigés vers l'unité centrale l pour y être traités. Le CPU l peut également exécuter des processus utilisateur s'il reste du temps CPU. Ce modèle est appelé **maître-esclave** car l'unité centrale 1 est le maître et toutes les autres sont des esclaves.

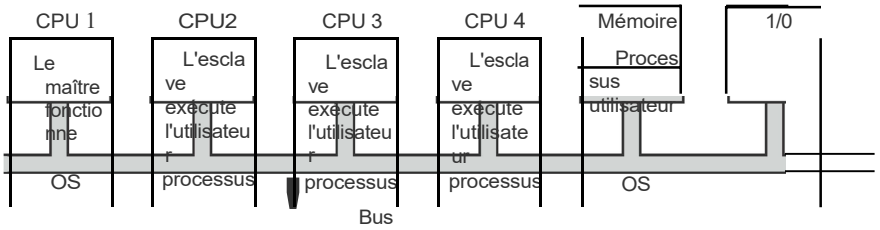


Figure 8-8. Un modèle de multiprocesseur maître-

Le modèle maître-esclave résout la plupart des problèmes du premier modèle. Il existe une structure de données unique (par exemple, une liste ou un ensemble de listes hiérarchisées) qui garde la trace des processus prêts. Lorsqu'une unité centrale est inactive, elle demande au système d'exploitation de l'unité centrale l un processus à exécuter et on lui en attribue un. Ainsi, il ne peut jamais arriver qu'une unité centrale soit

inactif alors qu'un autre est surchargé. De même, les pages peuvent être réparties entre tous les processus de manière dynamique et il n'y a qu'un seul cache tampon, de sorte que les incohérences ne se produisent jamais.

Le problème de ce modèle est qu'avec de nombreuses unités centrales, le maître deviendra un goulot d'étranglement. Après tout, il doit gérer tous les appels système de toutes les unités centrales. Si, par exemple, 10 % du temps est consacré à la gestion des appels système, alors 10 unités centrales satureront pratiquement le maître, et avec 20 unités centrales, il sera complètement surchargé. Ce modèle est donc simple et utilisable pour les petits multiprocesseurs, mais il échoue pour les grands.

Multiprocesseurs symétriques

Notre troisième modèle, le **SMP (Symmetric MultiProcessor),** élimine cette asymétrie. Il n'y a qu'une seule copie du système d'exploitation en mémoire, mais n'importe quelle unité centrale peut l'exécuter. Lorsqu'un appel système est effectué, l'unité centrale sur laquelle l'appel système a été effectué se connecte au noyau et traite l'appel système. Le modèle SMP est illustré à la Fig. 8-9.

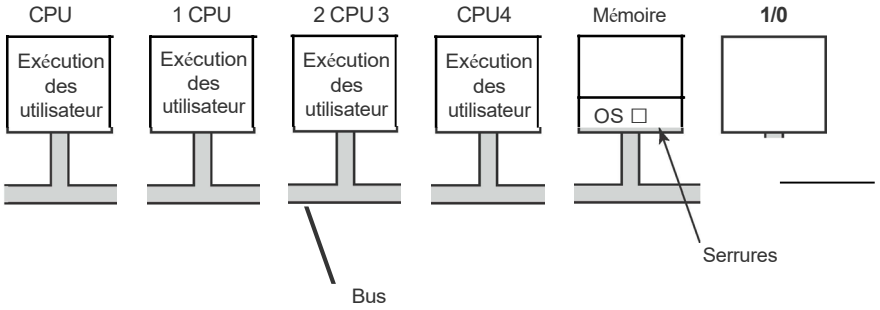


Figure 8-9. Le modèle de multiprocesseur SMP.

Ce modèle équilibre les processus et la mémoire de manière dynamique, puisqu'il n'y a qu'un seul ensemble de tables de système d'exploitation. Il élimine également le goulot d'étranglement du processeur maître, puisqu'il n'y a pas de maître, mais il introduit ses propres problèmes. En particulier, si deux CPU ou plus exécutent du code de système d'exploitation en même temps, il peut en résulter un désastre. Imaginez que deux CPU choisissent simultanément le même processus à exécuter ou réclament la même page de mémoire libre. La façon la plus simple de contourner ces problèmes est d'associer un mutex (c'est-à-dire un verrou) au système d'exploitation, faisant de l'ensemble du système une grande région critique. Lorsqu'une unité centrale veut exécuter le code du système d'exploitation, elle doit d'abord acquérir le mutex. Si le mutex est verrouillé, il attend simplement. De cette façon, n'importe quelle unité centrale peut exécuter le système d'exploitation, mais seulement une à la fois. Cette approche est parfois appelée "**big kernel lock".**

Ce modèle fonctionne, mais il est presque aussi mauvais que le modèle maître-esclave. Encore une fois, il faut supposer que 10 % du temps d'exécution est passé dans le système d'exploitation. Avec 20 processeurs, il y aura de longues files d'attente de processeurs attendant d'entrer. Heureusement, il est facile de l'imager. De nombreuses parties du système d'exploitation sont indépendantes les unes des autres. Pour

Par exemple, il n'y a aucun problème à ce qu'une unité centrale exécute l'ordonnanceur pendant qu'une autre unité centrale gère un appel au système de fichiers et qu'une troisième traite un défaut de page.

Cette observation conduit à diviser le système d'exploitation en plusieurs régions critiques indépendantes qui n'interagissent pas entre elles. Chaque région critique est protégée par son propre mutex, de sorte qu'une seule unité centrale à la fois peut l'exécuter. De cette façon, il est possible d'obtenir un parallélisme beaucoup plus important. Cependant, il peut arriver que certaines tables, comme la table des processus, soient utilisées par plusieurs régions critiques. Par exemple, la table des processus est nécessaire pour l'ordonnancement, mais aussi pour l'appel système **fork** et pour la gestion des signaux. Chaque table qui peut être utilisée par plusieurs régions critiques a besoin de son propre mutex. De cette façon, chaque région critique ne peut être exécutée que par une seule unité centrale à la fois et chaque table critique ne peut être accédée que par une seule unité centrale à la fois.

La plupart des multiprocesseurs de modem utilisent cet arrangement. La difficulté d'écrire le système d'exploitation d'une telle machine ne réside pas dans le fait que le code réel soit si différent d'un système d'exploitation ordinaire. Il ne l'est pas. La partie la plus difficile est de le diviser en régions critiques qui peuvent être exécutées simultanément par différentes unités centrales sans interférer les unes avec les autres, même pas de manière subtile et indirecte. De plus, chaque table utilisée par deux ou plusieurs régions critiques doit être protégée séparément par un mutex et tout code utilisant la table doit utiliser le mutex correctement.

En outre, il faut faire très attention pour éviter les blocages. Si deux régions critiques ont toutes deux besoin de la table *A* et de la table *B,* et que l'une d'entre elles réclame d'abord *la table A* et l'autre la *table B*, un blocage se produira tôt ou tard et personne ne saura pourquoi. En théorie, on pourrait attribuer des valeurs entières à toutes les tables et demander à toutes les régions critiques d'acquérir les tables par ordre croissant. Cette stratégie permet d'éviter les blocages, mais elle exige du programmeur qu'il réfléchisse très soigneusement aux tables dont chaque région critique a besoin et qu'il fasse les demandes dans le bon ordre.

Au fur et à mesure que le code évolue, une région critique peut avoir besoin d'une nouvelle table dont elle n'avait pas besoin auparavant. Si le programmeur est nouveau et ne comprend pas toute la logique du système, il sera tenté de s'emparer du mutex de la table au moment où il en a besoin et de le libérer lorsqu'il n'en a plus besoin. Aussi raisonnable que cela puisse paraître, cela peut conduire à des blocages, que l'utilisateur percevra comme un blocage du système. Il n'est pas facile de bien faire les choses et il est très difficile de les maintenir pendant des années en changeant de programmeur.

8.1.3 Synchronisation des multiprocesseurs

Les processeurs d'un multiprocesseur ont souvent besoin de se synchroniser. Nous venons de voir le cas où les régions critiques et les tables du noyau doivent être protégées par des mutex. Examinons maintenant de près comment cette synchronisation fonctionne réellement dans un multiprocesseur. C'est loin d'être trivial, comme nous le verrons bientôt.

Pour commencer, des primitives de synchronisation appropriées sont vraiment nécessaires. Si un processus sur une machine uniprocesseur (une seule unité centrale) fait un appel système qui nécessite l'accès à une table critique du noyau, le code du noyau peut simplement désactiver les interruptions avant que

touchant la table. Il peut alors effectuer son travail en sachant qu'il pourra terminer sans qu'aucun autre processus ne s'infiltre et ne touche la table avant qu'il n'ait terminé. Sur un multiprocesseur, la désactivation des interruptions n'affecte que l'unité centrale de traitement qui procède à la désactivation. Les autres CPU continuent à fonctionner et peuvent toujours toucher la table critique. Par conséquent, un protocole mutex approprié doit être utilisé et respecté par toutes les unités centrales pour garantir le fonctionnement de l'exclusion mutuelle.

Le cœur de tout protocole mutex pratique est une instruction spéciale qui permet d'inspecter et de fixer un mot mémoire en une seule opération indivisible. Nous avons vu comment l'instruction TSL (Test and Set Lock) était utilisée à la Fig. 2-25 pour implémenter les régions critiques. Comme nous l'avons vu précédemment, cette instruction permet de lire un mot mémoire et de le stocker dans un registre. Simultanément, elle écrit un 1 (ou une autre valeur non nulle) dans le mot mémoire. Bien entendu, il faut deux cycles de bus pour effectuer la lecture et l'écriture en mémoire. Sur un uniprocesseur, tant que l'instruction ne peut pas être interrompue à mi-chemin, la TSL fonctionne toujours comme prévu.

Imaginez maintenant ce qui pourrait se passer sur un multiprocesseur. Sur la figure 8-10, nous voyons la chronologie la plus défavorable, dans laquelle le mot mémoire 1000, utilisé comme verrou, est initialement à 0. À l'étape 1, l'unité centrale 1 lit le mot et obtient un 0. À l'étape 2, avant que l'unité centrale 1 ait la possibilité de réécrire le mot à 1, l'unité centrale 2 entre en jeu et lit également le mot à 0. À l'étape 3, l'unité centrale 1 écrit un 1 dans le mot. A l'étape 4, le CPU 2 écrit également un 1 dans le mot. Les deux CPU ont reçu un 0 en retour de l'instruction TSL, donc les deux ont maintenant accès à la région critique et l'exclusion mutuelle échoue.

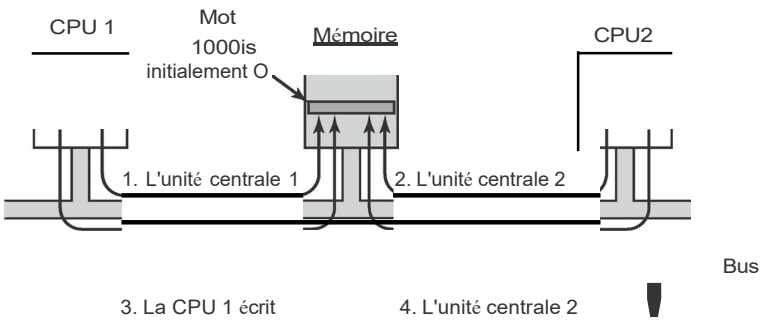


Figure 8-10. L'instruction TSL peut échouer si le bus ne peut pas être verrouillé. Ces quatre étapes montrent une séquence d'événements où l'échec est démontré.

Pour éviter ce problème, l'instruction TSL doit d'abord verrouiller le bus, empêchant les autres CPU d'y accéder, puis effectuer les deux accès à la mémoire, et enfin déverrouiller le bus. En général, le verrouillage du bus est effectué en demandant le bus à l'aide du protocole de demande de bus habituel, puis en activant (c'est-à-dire en mettant à une valeur logique de 1) une ligne de bus spéciale jusqu'à ce que les *deux* cycles soient terminés. Tant que cette ligne spéciale est activée, aucune autre unité centrale n'a accès au bus. Cette instruction ne peut être mise en oeuvre que sur un bus qui dispose des lignes nécessaires et du protocole (matériel) pour les utiliser. Les bus modernes disposent tous de ces facilités, mais sur les anciens bus qui n'en disposaient pas, il n'était pas possible d'utiliser cette instruction.

d'implémenter correctement le TSL. C'est pourquoi le protocole de Peterson a été inventé : pour synchroniser entièrement en logiciel (Peterson, 1981).

Si TSL est correctement implémenté et utilisé, il garantit que l'exclusion mutuelle peut fonctionner. Cependant, cette méthode d'exclusion mutuelle utilise un **verrou tournant,** car l'unité centrale requérante reste assise dans une boucle serrée qui teste le verrou aussi vite qu'elle le peut. Non seulement cela fait perdre du temps à l'unité centrale (ou aux unités centrales) demandeuse, mais cela peut également imposer une charge massive sur le bus ou la mémoire, ralentissant sérieusement toutes les autres unités centrales qui essaient de faire leur travail normal.

À première vue, il pourrait sembler que la présence de la mise en cache devrait éliminer le problème de contention du bus, mais ce n'est pas le cas. En théorie, une fois que l'unité centrale requérante a lu le mot de verrouillage, elle devrait en obtenir une copie dans son cache. Tant qu'aucune autre UC ne tente d'utiliser le verrou, l'UC requérante devrait pouvoir épuiser son cache. Lorsque l'unité centrale propriétaire du verrou écrit un O sur celui-ci pour le libérer, le protocole de cache invalide automatiquement toutes les copies de celui-ci dans les caches distants, ce qui nécessite de récupérer à nouveau la valeur correcte.

Le problème est que les caches fonctionnent par blocs de 32 ou 64 octets. En général, les mots entourant le verrou sont nécessaires à l'unité centrale qui détient le verrou. Comme l'instruction TSL est une écriture (car elle modifie le verrou), elle a besoin d'un accès exclusif au bloc de cache contenant le verrou. Par conséquent, chaque TSL invalide le bloc dans le cache du détenteur du verrou et récupère une copie privée et exclusive pour l'unité centrale requérante. Dès que le détenteur du verrou touche un mot adjacent au verrou, le bloc de cache est déplacé vers sa machine. Par conséquent, l'ensemble du bloc de cache contenant le verrou est constamment déplacé entre le détenteur du verrou et le demandeur du verrou, ce qui génère encore plus de trafic sur le bus que les lectures individuelles du mot verrou.

Si nous pouvions nous débarrasser de toutes les écritures induites par le TSL du côté de la demande, nous pourrions réduire de manière appréciable le "thrashing" du cache. Cet objectif peut être atteint en demandant au CPU demandeur de faire d'abord une lecture pure pour voir si le verrou est libre. Ce n'est que si le verrou semble être libre qu'il effectue un TSL pour l'acquérir. Le résultat de ce petit changement est que la plupart des interrogations sont maintenant des lectures au lieu d'écritures. Si l'unité centrale détenant le verrou ne fait que lire les variables du même bloc de cache, elles peuvent chacune avoir une copie du bloc de cache en mode lecture seule partagée, éliminant ainsi tous les transferts de blocs de cache.

Lorsque le verrou est finalement libéré, le propriétaire effectue une écriture, qui nécessite un accès exclusif, invalidant ainsi toutes les copies dans les caches distants. Lors de la prochaine lecture par l'unité centrale requérante, le bloc de cache sera rechargé. Notez que si deux CPU ou plus se disputent le même verrou, il se peut que les deux voient qu'il est libre simultanément, et que les deux fassent un TSL simultanément pour l'acquérir. Un seul d'entre eux réussira, il n'y a donc pas de condition de course ici car l'acquisition réelle est faite par l'instruction TSL, et elle est atomique. Le fait de voir que le verrou est libre et d'essayer de le saisir immédiatement avec une TSL ne garantit pas que vous l'obteniez. Quelqu'un d'autre pourrait gagner, mais pour l'exactitude de l'algorithme, cela n'a pas d'importance de savoir qui l'obtient. La réussite de la lecture pure est simplement un indice que c'est le bon moment pour essayer d'acquérir le verrou, mais ce n'est pas une garantie que l'acquisition réussira.

Une autre façon de réduire le trafic du bus est d'utiliser l'algorithme binaire exponentiel backoff bien connu d'Ethernet (Anderson, 1990). Au lieu d'interroger en continu, comme sur la Fig. 2-25, une boucle de retard peut être insérée entre les interrogations. Initialement, le délai est d'une instruction. Si le verrou est toujours occupé, le délai est doublé à deux instructions, puis quatre instructions, et ainsi de suite jusqu'à un certain maximum. Un maximum faible donne une réponse rapide lorsque le verrou est libéré, mais gaspille plus de cycles de bus pour la mise en cache. Un maximum élevé réduit la destruction de la mémoire cache au prix de ne pas remarquer que le verrou est libéré si rapidement. Le backoff exponentiel binaire peut être utilisé avec ou sans les lectures pures précédant l'instruction TSL.

Une idée encore meilleure est de donner à chaque CPU souhaitant acquérir le mutex sa propre variable de verrouillage privée à tester, comme illustré à la Fig. 8-11 (Mellor-Crummey et Scott, 1991). La variable doit résider dans un bloc de cache inutilisé pour éviter les conflits. L'algorithme fonctionne en faisant en sorte qu'une unité centrale de traitement qui ne parvient pas à acquérir le verrou alloue une variable de verrou et s'attache à la fin d'une liste d'unités centrales de traitement en attente du verrou. Lorsque le détenteur actuel du verrou sort de la région critique, il libère le verrou privé que le premier CPU de la liste teste (dans son propre cache). Cette unité centrale entre alors dans la région critique. Lorsqu'elle a terminé, elle libère le verrou que son successeur utilise, et ainsi de suite. Bien que le protocole soit quelque peu compliqué (pour éviter que deux CPU ne s'attachent simultanément à la fin de la liste), il est efficace et sans famine. Pour tous les détails, les lecteurs sont invités à consulter l'article.

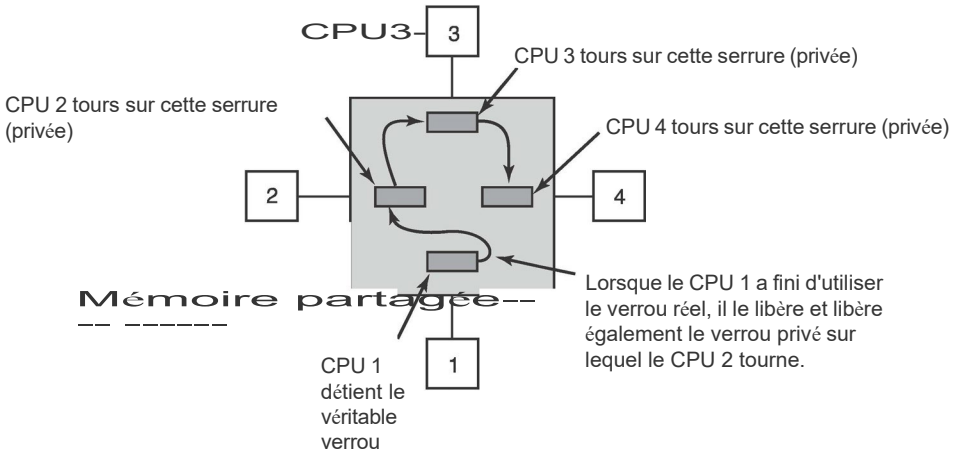


Figure 8-11. Utilisation de verrous multiples pour éviter le thrashing du cache.

Spinning vs. Switching

Jusqu'à présent, nous avons supposé qu'une CPU ayant besoin d'un mutex verrouillé l'attendait simplement, en l'interrogeant de manière continue, intermittente ou en s'attachant à une liste de CPU en attente. Parfois, il n'y a pas d'autre solution pour l'unité centrale requérante que d'attendre. Par exemple, supposons qu'une CPU soit inactive et ait besoin d'accéder au mutex partagé

pour choisir un processus à exécuter. Si la liste des processus prêts à l'emploi est verrouillée, l'unité centrale ne peut pas simplement décider de suspendre ses activités et d'exécuter un autre processus, car cela nécessiterait de lire la liste des processus prêts à l'emploi. Il *doit* attendre jusqu'à ce qu'il puisse acquérir la liste des processus prêts.

Cependant, dans d'autres cas, il existe un choix. Par exemple, si un thread d'une unité centrale doit accéder au cache tampon du système de fichiers et que celui-ci est actuellement verrouillé, l'unité centrale peut décider de passer à un autre thread au lieu d'attendre. La question de savoir s'il faut faire tourner ou faire un changement de thread a fait l'objet de nombreuses recherches, dont certaines seront abordées ci-dessous. Notez que ce problème ne se pose pas sur un uniprocesseur car la rotation n'a pas beaucoup de sens lorsqu'il n'y a pas d'autre CPU pour libérer le verrou. Si un thread tente d'acquérir un verrou et échoue, il est toujours bloqué pour donner au propriétaire du verrou une chance de s'exécuter et de libérer le verrou.

Si l'on suppose que la rotation et le changement de thread sont deux options possibles, le compromis est le suivant. La rotation gaspille directement des cycles CPU. Tester un verrou de manière répétée n'est pas un travail productif. La commutation, cependant, gaspille également des cycles de CPU, puisque l'état du thread actuel doit être sauvegardé, le verrou de la liste prête doit être demandé, un thread doit être sélectionné, son état doit être chargé et il doit être lancé. En outre, le cache du CPU contiendra tous les blocs erronés, de sorte que de nombreux et coûteux manques de cache se produiront lorsque le nouveau thread commencera à fonctionner. Des erreurs de TLB sont également probables. Finalement, un retour au thread d'origine doit avoir lieu, ce qui entraîne de nouvelles pertes de cache. Les cycles passés à effectuer ces deux changements de contexte, plus toutes les pertes de cache, sont gaspillés.

Si l'on sait que les mutex sont généralement maintenus pendant, disons, 50 µsec et qu'il faut 1 msec pour basculer du thread actuel et 1 msec pour revenir plus tard, il est plus efficace de simplement faire tourner le mutex. D'un autre côté, si le mutex moyen est maintenu pendant 10 msec, cela vaut la peine d'effectuer les deux commutations de contexte. Le problème est que les régions critiques peuvent varier considérablement dans leur durée, alors quelle approche est la meilleure ?

Un premier modèle consiste à toujours tourner. Une deuxième conception consiste à toujours commuter. Mais une troisième conception consiste à prendre une décision séparée chaque fois qu'un mutex verrouillé est rencontré. Au moment où la décision doit être prise, on ne sait pas s'il est préférable de tourner ou de commuter, mais pour tout système donné, il est possible de faire une trace de toute l'activité et de l'analyser ultérieurement hors ligne. On peut alors dire rétrospectivement quelle décision était la meilleure et combien de temps a été perdu dans le meilleur des cas. Cet algorithme rétrospectif devient alors une référence par rapport à laquelle les algorithmes réalisables peuvent être mesurés.

Ce problème est étudié par les chercheurs depuis des décennies (Ousterhout, 1982). La plupart des travaux utilisent un modèle dans lequel un thread ne parvenant pas à acquérir un mutex tourne pendant une certaine période de temps. Si ce seuil est dépassé, il commute. Dans certains cas, le seuil est fixe, typiquement l'overhead connu pour le passage à un autre thread puis le retour. Dans d'autres cas, il est dynamique, en fonction de l'historique observé du mutex en attente.

Les meilleurs résultats sont obtenus lorsque le système garde la trace des derniers temps de rotation observés et suppose que celui-ci sera similaire aux précédents. Par exemple, en supposant que le temps de changement de contexte est à nouveau de 1 ms, un thread tournera pendant une durée de

maximum de 2 msec, mais observez combien de temps il a réellement tourné. S'il ne parvient pas à acquérir un verrou et qu'il constate que lors des trois exécutions précédentes, il a attendu en moyenne 200 μsec, il devrait tourner pendant 2 msec avant de commuter. Cependant, s'il voit qu'il a tourné pendant toute la durée de 2 msec à chacune des tentatives précédentes, il devrait commuter immédiatement et ne pas tourner du tout.

Certains processeurs de modem, dont le x86, proposent des instructions spéciales pour rendre l'attente plus efficace en termes de réduction de la consommation d'énergie. Par exemple, les instructions **MONITOR/MWAIT** sur x86 permettent à un programme de se bloquer jusqu'à ce qu'un autre processeur modifie les données dans une zone mémoire préalablement définie. Plus précisément, l'instruction **MONITOR** définit une plage d'adresses qui doit être surveillée pour les écritures. L'instruction **MWAIT** bloque ensuite le thread jusqu'à ce que quelqu'un écrive dans cette zone. En fait, le thread tourne, mais sans brûler inutilement de nombreux cycles.

8.1.4 Ordonnancement des multiprocesseurs

Avant d'examiner comment l'ordonnancement est effectué sur les multiprocesseurs, il est nécessaire de déterminer *ce qui est* ordonnancé. Autrefois, lorsque tous les processus étaient monofilaires, les processus étaient ordonnancés - il n'y avait rien d'autre à ordonnancer. Tous les systèmes d'exploitation modernes prennent en charge les processus multithreads, ce qui rend l'ordonnancement plus compliqué.

Il est important de savoir si les threads sont des threads du noyau ou des threads de l'utilisateur. Si le threading est effectué par une bibliothèque de l'espace utilisateur et que le noyau ne sait rien des threads, l'ordonnancement se fait sur une base par processus comme cela a toujours été le cas. Si le noyau ne sait même pas que les threads existent, il peut difficilement les ordonnancer.

Avec les threads du noyau, la situation est différente. Ici, le noyau connaît tous les threads et peut choisir parmi les threads appartenant à un processus. Dans ces systèmes, la tendance est que le noyau choisisse un thread à exécuter, le processus auquel il appartient n'ayant qu'un petit rôle (voire aucun) dans l'algorithme de sélection des threads. Nous parlerons ci-dessous de l'ordonnancement des threads, mais bien sûr, dans un système avec des processus monofilaires ou des threads implémentés dans l'espace utilisateur, ce sont les processus qui sont ordonnancés.

Le processus par rapport au thread n'est pas le seul problème d'ordonnancement. Sur un uniprocesseur, l'ordonnancement est unidimensionnel. La seule question à laquelle il faut répondre (de manière répétée) est : "Quel thread doit être **exécuté** ensuite ?" Sur un multiprocesseur, l'ordonnancement a deux dimensions. L'ordonnanceur doit décider quel thread doit être exécuté et sur quelle unité centrale il doit l'exécuter. Cette dimension supplémentaire complique considérablement l'ordonnancement sur les multiprocesseurs. Un autre facteur de complication est que, dans certains systèmes, tous les threads ne sont pas liés, ils appartiennent à des processus différents et n'ont rien à voir les uns avec les autres. Dans d'autres, ils se présentent en groupes, appartenant tous à la même application et travaillant ensemble. Un exemple de la première situation est un système serveur dans lequel des utilisateurs indépendants lancent des processus indépendants. Les threads des différents processus ne sont pas liés et chacun d'entre eux peut être planifié sans tenir compte des autres.

Un exemple de cette dernière situation se produit régulièrement dans les environnements de développement de programmes. Les grands systèmes sont souvent constitués d'un certain nombre de fichiers d'en-tête contenant des macros, des définitions de types et des déclarations de variables qui sont utilisées par les fichiers de code proprement dits. Lorsqu'un fichier d'en-tête est modifié, tous les fichiers de code qui l'incluent doivent être recompilés. Le programme *make* est couramment utilisé pour gérer le développement. Lorsque *make* est invoqué, il lance la compilation des seuls fichiers de code qui doivent être recompilés en raison des modifications apportées aux fichiers d'en-tête ou de code. Les fichiers d'objets qui sont encore valides ne sont pas régénérés.

La version originale de *make* effectuait son travail séquentiellement, mais les nouvelles versions signées pour les multiprocesseurs peuvent lancer toutes les compilations en même temps. Si 10 compilations sont nécessaires, cela n'a pas de sens de programmer 9 d'entre elles pour qu'elles s'exécutent immédiatement et de laisser la dernière s'exécuter beaucoup plus tard, car l'utilisateur ne percevra pas le travail comme terminé avant que la dernière ne soit terminée. Dans ce cas, il est logique de considérer les threads effectuant les compilations comme un groupe et d'en tenir compte lors de leur ordonnancement.

De plus en plus, il est parfois utile de programmer des threads qui communiquent de manière intensive, par exemple dans un mode producteur-consommateur, non seulement en même temps, mais aussi à proximité les uns des autres dans l'espace. Par exemple, ils peuvent bénéficier du partage des caches. De même, dans les architectures NUMA, il peut être utile qu'ils accèdent à la mémoire qui est proche.

Partage du temps

Examinons d'abord le cas de l'ordonnancement de threads indépendants ; nous verrons ensuite comment ordonnancer des threads liés. L'algorithme d'ordonnancement le plus simple pour traiter les threads indépendants consiste à disposer d'une structure de données unique à l'échelle du système pour les threads prêts à fonctionner, éventuellement une simple liste, mais plus probablement un ensemble de listes pour les threads ayant des priorités différentes, comme le montre la figure 8-12(a). Ici, les 16 CPU sont toutes occupées et un ensemble de 14 threads prioritaires attendent de s'exécuter. La première unité centrale à terminer son travail en cours (ou à voir son thread bloqué) est l'unité centrale 4, qui verrouille alors les files d'attente d'ordonnancement et sélectionne le thread de plus haute priorité, *A,* comme le montre la figure 8-12(b). Ensuite, l'unité centrale 12 devient inactive et choisit le fil *B,* comme illustré à la figure 8-12(c). Tant que les threads ne sont pas du tout liés, cette méthode d'ordonnancement est un choix raisonnable et très simple à mettre en œuvre efficacement.

Le fait de disposer d'une structure de données d'ordonnancement unique utilisée par toutes les unités centrales permet de répartir dans le temps le processus de planification de l'ordonnancement.

comme ils le seraient dans un système uniprocesseur. Elle permet également un équilibrage automatique de la charge, car il ne peut jamais arriver qu'une unité centrale soit inactive alors que les autres sont surchargées. Les deux inconvénients de cette approche sont la contention potentielle de la structure de données d'ordonnancement lorsque le nombre de CPU augmente et le surcoût habituel lié au changement de contexte lorsqu'un thread bloque pour 1/0.

Il est également possible qu'un changement de contexte se produise lorsque le quantum d'un thread expire. Sur un multiprocesseur, cela présente certaines propriétés qui n'existent pas sur un uniprocesseur. Supposons que le thread

détienne un spin lock lorsque son quantum expire. Les autres CPUs qui attendent le spin lock perdent leur temps à tourner jusqu'à ce que le quantum expire.

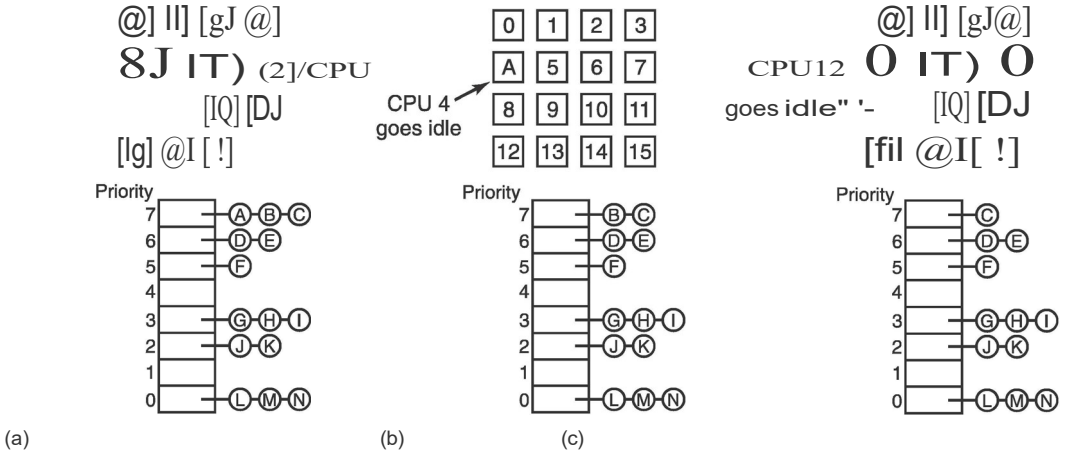


Figure 8-12. Utilisation d'une structure de données unique pour l'ordonnancement d'un multiprocesseur.

Le thread est à nouveau programmé et libère le verrou. Sur un uniprocesseur, les verrous tournants sont rarement utilisés, donc si un processus est suspendu alors qu'il détient un mutex, et qu'un autre thread démarre et tente d'acquérir le mutex, il sera immédiatement bloqué, donc peu de temps est perdu.

Pour contourner cette anomalie, certains systèmes utilisent un **ordonnancement intelligent,** dans lequel un thread qui acquiert un verrou de rotation place un drapeau à l'échelle du processus pour montrer qu'il possède actuellement un verrou de rotation (Zahorjan et al., 1991). Lorsqu'il libère le verrou, il efface l'indicateur. L'ordonnanceur n'arrête donc pas un thread détenant un verrou de spin, mais lui donne un peu plus de temps pour terminer sa région critique et libérer le verrou.

Un autre problème qui joue un rôle dans l'ordonnancement est le fait que si tous les processeurs sont égaux, certains le sont davantage. En particulier, lorsque le thread *A* a été exécuté pendant une longue période sur le CPU *k,* le cache du CPU k sera rempli des blocs de A. Si A doit être exécuté à nouveau prochainement, il peut être plus performant s'il est exécuté sur le CPU k parce que k est plus performant. Si *A* doit s'exécuter à nouveau prochainement, il peut être plus performant s'il est exécuté sur le CPU *k,* car le cache de k peut encore contenir certains des blocs de A. Le fait d'avoir des blocs de cache préchargés augmentera le taux d'accès au cache et donc la vitesse du thread. En outre, la TLB peut également contenir les bonnes pages, ce qui réduit les défauts de la TLB.

Certains multiprocesseurs tiennent compte de cet effet et utilisent ce qu'on appelle l'**ordonnancement par affinité (**Vaswani et Zahorjan, 1991). L'idée de base est de faire un effort sérieux pour qu'un thread s'exécute sur le même CPU que celui sur lequel il s'est exécuté la dernière fois. Une façon de créer cette affinité est d'utiliser un **algorithme d'ordonnancement à deux niveaux.** Lorsqu'un thread est créé, il est affecté à un CPU, par exemple en fonction de celui qui a la charge la plus faible à ce moment-là. Cette affectation des threads aux CPU constitue le niveau supérieur de l'algorithme. En conséquence de cette politique, chaque CPU acquiert sa propre collection de threads.

L'ordonnancement réel des threads est le niveau inférieur de l'algorithme. Il est effectué par chaque CPU séparément, en utilisant des priorités ou d'autres moyens. En essayant de

En gardant un thread sur le même CPU pendant toute sa durée de vie, l'affinité du cache est maximisée. Cependant, si une unité centrale n'a pas de threads à exécuter, elle en prend un d'une autre unité centrale plutôt que de rester inactive.

L'ordonnancement à deux niveaux présente trois avantages. Tout d'abord, il distribue la charge de manière à peu près égale sur les processeurs disponibles. Deuxièmement, l'affinité avec le cache est exploitée dans la mesure du possible. Troisièmement, en donnant à chaque CPU sa propre liste de disponibilité, la contention pour les listes de disponibilité est minimisée car les tentatives d'utiliser la liste de disponibilité d'un autre CPU sont relativement peu fréquentes.

Partage de l'espace

L'autre approche générale de l'ordonnancement multiprocesseur peut être utilisée lorsque les threads sont liés les uns aux autres d'une certaine manière. Nous avons mentionné plus haut l'exemple de la *fabrication* en parallèle comme l'un des cas. Il arrive aussi souvent qu'un seul processus ait plusieurs threads qui travaillent ensemble. Par exemple, si les threads d'un processus communiquent beaucoup, il est utile de les faire fonctionner en même temps. La planification de plusieurs threads en même temps sur plusieurs CPU est appelée **partage d'espace.**

L'algorithme de partage d'espace le plus simple fonctionne comme suit. Supposons qu'un groupe entier de threads apparentés soit créé en une seule fois. Au moment de sa création, l'ordonnanceur vérifie s'il y a autant de CPU libres que de threads. Si c'est le cas, chaque thread reçoit sa propre unité centrale dédiée (c'est-à-dire non multiprogrammée) et ils démarrent tous. S'il n'y a pas assez de CPU, aucun des threads n'est lancé jusqu'à ce que suffisamment de CPUs soient disponibles. Chaque thread conserve son CPU jusqu'à ce qu'il se termine, auquel cas le CPU est remis dans le pool de CPUs disponibles. Si un thread se bloque sur 1/0, il continue à retenir le CPU, qui est simplement inactif jusqu'à ce que le thread se réveille. Lorsque le lot suivant de threads apparaît, le même algorithme est appliqué.

À tout moment, l'ensemble des processeurs est divisé de manière statique en un certain nombre de partitions, chacune d'entre elles exécutant les threads d'un processus. Sur la figure 8-13, nous avons des partitions de taille 4, 6, 8 et 12 CPU, avec 2 CPU non assignés, par exemple. Au fil du temps, le nombre et la taille des partitions changent au fur et à mesure que de nouveaux threads sont créés et que les anciens se terminent.

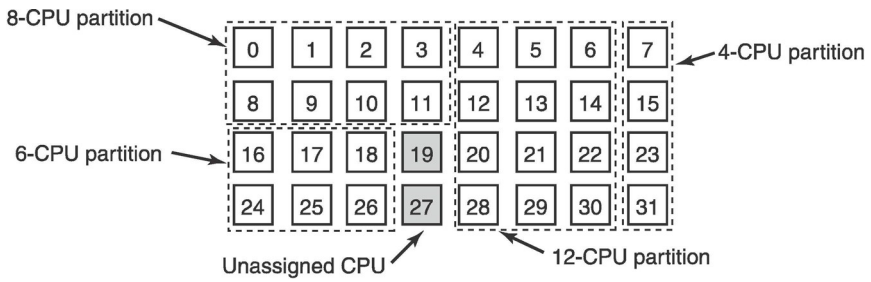


Figure 8-13. Un ensemble de 32 CPU divisé en quatre partitions, avec deux CPU disponibles.

Périodiquement, des décisions d'ordonnancement doivent être prises. Dans les systèmes uniprocesseurs, l'algorithme de la tâche la plus courte en premier est bien connu pour l'ordonnancement des lots. L'algorithme analogue pour un multiprocesseur consiste à choisir le processus nécessitant le plus petit nombre de cycles d'unité centrale, c'est-à-dire le fil dont le nombre d'unités centrales x le temps d'exécution est le plus petit des deux.
les candidats. Cependant, dans la pratique, cette information est rarement disponible, de sorte que l'algorithme est difficile à réaliser. En fait, des études ont montré que, dans la pratique, il est difficile de battre le principe du "premier arrivé, premier servi" (Krueger et al., 1994).

Dans ce modèle de partitionnement simple, un thread demande simplement un certain nombre de CPU et soit il les obtient tous, soit il doit attendre qu'ils soient disponibles. Une autre approche consiste à ce que les threads gèrent activement le degré de parallélisme. Une méthode pour gérer le parallélisme est d'avoir un serveur central qui garde la trace des threads qui sont en cours d'exécution et qui veulent s'exécuter, ainsi que de leurs besoins minimum et maximum en CPU (Tucker et Gupta, 1989). Périodiquement, chaque application interroge le serveur central pour savoir combien de CPU elle peut utiliser. Elle ajuste alors le nombre de threads à la hausse ou à la baisse en fonction de ce qui est disponible.

Par exemple, un serveur Web peut avoir 5, 10, 20, ou tout autre nombre de threads fonctionnant en parallèle. Si le nombre de threads est actuellement de 10, que la demande de CPU augmente soudainement et qu'on lui demande de passer à 5, lorsque les 5 threads suivants auront terminé leur travail, on leur demandera de se retirer au lieu de leur confier un nouveau travail. Ce schéma permet à la taille des partitions de varier dynamiquement pour s'adapter à la charge de travail actuelle, mieux que le système fixe de la figure 8-13.

Programmation des gangs

Un avantage évident du partage de l'espace est l'élimination de la multiprogrammation, qui élimine le surcoût du changement de contexte. Cependant, un inconvénient tout aussi évident est le temps perdu lorsqu'un processeur se bloque et n'a rien à faire jusqu'à ce qu'il soit à nouveau prêt. Par conséquent, les gens ont cherché des algorithmes qui permettent de planifier à la fois le temps et l'espace, en particulier pour les threads qui créent plusieurs threads, qui ont généralement besoin de communiquer entre eux.

Pour voir le type de problème qui peut se produire lorsque les threads d'un processus sont ordonnancés de manière dépendante, considérons un système avec les threads A_0 et A_1 appartenant au processus A et les threads B_0 et B_1 appartenant au processus B. Les threads A_0 et B_0 sont en temps partagé sur l'unité centrale O ; les threads A_1 et B_1 sont en temps partagé sur l'unité centrale 1. Les threads A_0 et A_1 doivent communiquer souvent. Le schéma de communication est le suivant : A_0 envoie un message à A_1 , et A_1 renvoie ensuite une réponse à A_0 , suivie d'une autre séquence de ce type, courante dans les situations client-serveur. Supposons que la chance veuille que A_0 et B_1 commencent en premier, comme le montre la Fig. 8-14.

Dans la tranche de temps 0, A_0 envoie une demande à A_1 , mais A_1 ne la reçoit pas avant de s'exécuter dans la tranche de temps 1 à partir de 100 msec. Il envoie la réponse immédiatement, mais A_0 ne reçoit pas la réponse avant qu'il ne s'exécute à nouveau à 200 msec. Le résultat net est une séquence demande-réponse toutes les 200 ms. Ce n'est pas une très bonne performance.

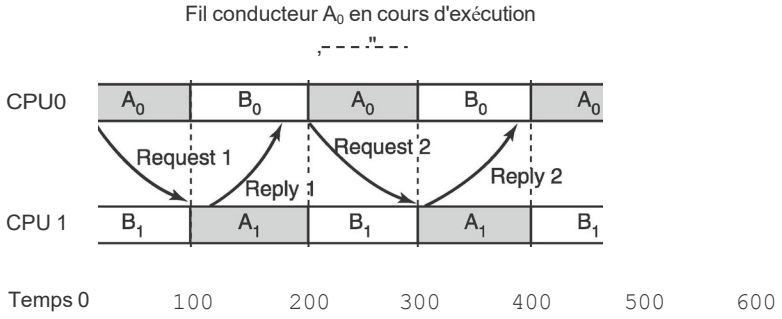


Figure 8-14. Communication entre deux threads appartenant au thread *A* qui sont déphasés.

La solution à ce problème est l'**ordonnancement collectif,** qui est une excroissance du **co-ordonnancement (**Ousterhout, 1982). L'ordonnancement collectif comporte trois parties :

1. Les groupes de fils apparentés sont programmés comme une unité, un gang.
2. Tous les membres d'un groupe s'exécutent en même temps sur différents processeurs en temps partagé.
3. Tous les membres du gang commencent et finissent leurs tranches de temps ensemble.

L'astuce qui fait fonctionner l'ordonnancement collectif est que tous les processeurs sont ordonnancés de manière synchrone. Cela signifie que le temps est divisé en quanta discrets, comme sur la figure 8-14. Au début de chaque nouveau quantum, *toutes les* unités centrales sont réordonnancées, et un nouveau thread est lancé sur chacune d'elles. Au début du quantum suivant, un autre événement d'ordonnancement se produit. Entre les deux, aucun ordonnancement n'est effectué. Si un thread se bloque, son CPU reste inactif jusqu'à la fin du quantum.

La figure 8-15 donne un exemple du fonctionnement de l'ordonnancement collectif. Nous avons ici un multiprocesseur avec six CPU utilisées par cinq processus, de *A* à *E,* avec un total de 24 threads prêts. Pendant le créneau 0, les threads A_0 et A_6 sont programmés et exécutés. Pendant le créneau 1, les threads B_0 , B_1 , B_2 , C_0 , C_1 , et C_2 sont programmés et exécutés. Pendant le slot 2, les cinq threads de D et E_0 s'exécutent. Les six autres threads appartenant au thread *E* s'exécutent dans le créneau 3. Puis le cycle recommence, le slot 4 étant identique au slot O et ainsi de suite.

L'idée de l'ordonnancement collectif est de faire en sorte que tous les fils d'un processus s'exécutent ensemble, au même moment, sur des CPU différents, de sorte que si l'un d'entre eux envoie une requête à un autre, il recevra le message presque immédiatement et sera en mesure de répondre presque immédiatement. Dans la Fig. 8-15, comme tous les threads *A* fonctionnent ensemble, pendant un quantum, ils peuvent envoyer et recevoir un très grand nombre de messages dans un même quantum, ce qui élimine le problème de la Fig. 8-14.

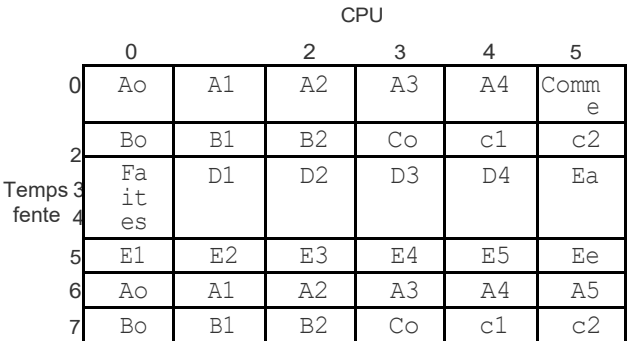


Figure 8-15. Planification des groupes.