UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 1

Concepts de base et rappels

Gabriel Girard[1]

Sherbrooke

6 janvier 2023

[1] Gabriel.Girard@usherooke.ca

Table des matières

Chapitre 1

Concepts de base et rappels

Ce chapitre est inspiré de [27, 30, 31].

1.1 Rappel : les noyaux et micro-noyaux des systèmes d'exploitation

Cette section constitue un rappel de concepts de base reliés aux systèmes d'exploitation [1].

1.1.1 Structure d'un système d'exploitation

Résumons ici la structure d'un système d'exploitation. Un système d'exploitation est composé de plusieurs couches. La figure 1.1 présente un aperçu très grossier de ces couches. La couche supérieure est l'interface utilisateur [2]. Cette interface peut être graphique ou sous forme de console. Généralement, ces interfaces sont implantées par des processus. Avec Linux, les environnements graphiques populaires sont Gnome, KDE, MATE, XFCE, LXQT et Cinnamon. Sur ce même système, il existe aussi plusieurs environnements de type console tel que «sh», «csh», «Bash», «KornShell», «Tcsh», «Fish» et «Z shell». Avec Windows, il n'y a pas vraiment de choix d'environnement graphique. Il existe toutefois plusieurs applications de type console sur Windows tel que «PowerShell», «Cmder», «Terminus» «Zoc» et plusieurs autres [26, 8].

La seconde couche contient toutes les bibliothèques utiles pour les applications. Elle comprend des bibliothèques statiques et dynamiques (DLL sur Windows et «.so» sur Linux).

Enfin, la dernière couche avant le matériel lui-même est le noyau du système. Il est important de ne pas confondre l'environnement système dans lequel nous travaillons avec le noyau lui-même. Ainsi Windows 10 est un environnement et le noyau de ce système est Windows NT. Linux est un noyau et les environnements sont les distributions de Linux telles que Android, Ubuntu, Debian, CentOS, Fedora, RedHat, Manjaro, Mint, Tizen, WebOS, Roku, etc. MacOS X et IOS sont des environnements basés sur le noyau XNU (dérivé du micro-noyau MACH).

1. Ces concepts ont été abordés en profondeur dans le cours de système d'exploitation (IFT 320).
2. Aussi appelée interface personne-machine.

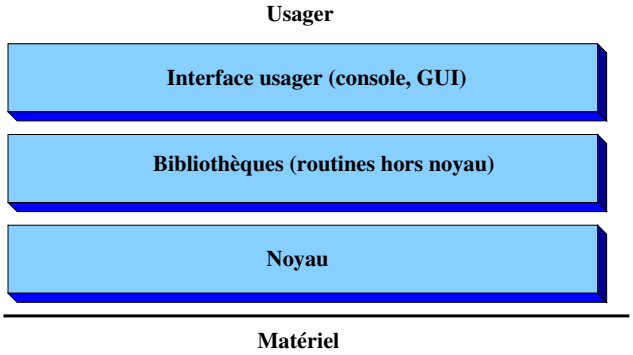


Figure 1.1 – Structure d'un système d'exploitation

1.1.2 Le concept de noyau

Nous débutons notre révision par une étude du concept noyau car c'est le premier programme parallèle. En effet, il doit gérer l'exécution simultanée de multiples applications avec tous les problèmes de communication et de synchronisation que cela implique. De plus, c'est aussi lui qui implante les outils nécessaires à la programmation parallèle tels que les processus, les fils d'exécution, les sémaphores, les «mutex», les verrous, les «sockets», les messages, ...).

Le noyau est le cœur du système d'exploitation. Il représente certes une mince partie de tout le système mais c'est la plus utilisée. Pour cette raison, le noyau réside (en grande partie) généralement en mémoire centrale tandis que le reste du système d'exploitation voyage entre les mémoires secondaire et centrale selon les besoins.

Le noyau fournit normalement les fonctions suivantes :

- la gestion des interruptions ;
  Selon le matériel, le noyau doit gérer divers types d'interruptions. Parmi ceux-ci on retrouve :
  - ▶ les interruptions pour les entrées/sorties ;
  - ▶ les interruptions de l'horloge ;
  - ▶ les interruptions pour les erreurs (bus error, segmentation fault, ...) ;
  - ▶ les interruptions logiques (emt, trap, svc, ...)
- la gestion des processus ;
  Pour la gestion des processus, le système fournit des fonctions de création, de destruction et plusieurs autres que nous abordons à la section 1.2.
- la gestion de l'UCT et du temps (IFT 320) ;
- la gestion de la mémoire (IFT 320) ;
- la gestion de la synchronisation ;
- la gestion de la communication ;
- la gestion des fichiers ;
  Dans certain cas, ce gestionnaire peut être retiré du noyau et intégré dans un processus.
- la gestion des activités d'entrées et de sorties ;
  Cette fonction se retrouve seulement sur certains systèmes particuliers.

- la gestion des codes d'accès des usagers ;

  Cette fonction se retrouve seulement sur certains types de systèmes.

Il est important de faire la distinction entre les notions de primitives et de fonctions [3]. Les primitives fournies par le système composent ce qu'on appelle son API (Application Programming Interface) et permettent d'accéder aux fonctions du système. Ainsi, pour certaines fonctions, le noyau implante des primitives telles que :

★ l'obtention de la date (gestion du temps)

★ l'allocation et la libération de la mémoire (gestion de la mémoire) ;

★ la création et la destruction de processus (gestion des processus) ;

★ les primitives de synchronisation (sémaphores, wait, signal, post) (gestion de la synchronisation) ;

★ les primitives de communication (sockets, messages, send, receive) (gestion de la communication) ;

  La communication inter-processus s'implante généralement grâce à :

  - la communication par messages (send, receive) ;
  - la communication par mémoire commune (pas de primitives) ;

Certaines fonctions du noyau ne fournissent aucune primitive et ne sont donc pas directement accessibles via l'API du système. C'est le cas de la gestion des blocs de contrôle (descripteurs) et de la gestion des interruptions.

Dans le passé, certains concepteurs ont choisi, pour des raisons de fiabilité et de modularité, de retirer des fonctions particulières du noyau créant, de ce fait, le concept de «**micro-noyau**».

1.1.3 Les micro-noyaux

Les noyaux contenant toutes les fonctionnalités mentionnées dans la section précédente sont appelés ou dits «**noyaux monolithiques**». Linux est un noyau monolithique.

Pour des raisons de flexibilité, de fiabilité et de modularité, les micro-noyaux ont été introduits. Selon ce concept, on retire le maximum de fonctionnalités du noyau pour les localiser plutôt dans des processus (des serveurs). Ceux-ci se retrouvent dans l'espace usager. Cependant, certaines fonctionnalités, comme la gestion des processus et des communications, ne peuvent pas être retirées du noyau, contrairement à la gestion des fichiers, des codes utilisateurs et de la mémoire (pagination et *swapping*).

Les avantages théoriques de ce type d'organisation sont une modularité, une flexibilité et une fiabilité accrue. En retirant une partie du code du noyau, on y minimise les erreurs, le rendant ainsi plus robuste et fiable. Certaines études ont montré qu'il y avait, en moyenne, entre deux et dix erreurs par 1 000 lignes de codes. Cela signifie qu'un noyau monolithique contenant 5 millions de lignes de code pourrait contenir entre 10 000 et 50 000 erreurs.

Le noyau de Linux contenait, en 2018, 25 millions de lignes de code impliquant donc qu'il s'y «cache» entre 50 000 et 250 000 erreurs. Quant à lui, le noyau de Windows, avec environ 50 millions de lignes de code, contiendrait entre 100 000 et 500 000 erreurs. Notons que les premiers micro-noyaux contenaient environ 50 000 de codes.

Les fonctions qui se retrouvent à l'extérieur du noyau sont aisément remplaçables ou réplicables. Cela facilite la maintenance et donne de la flexibilité. Il serait par exemple possible d'implanter

3. Cette distinction a été traitée au cours IFT 320

deux systèmes de fichiers distincts (deux processus distincts). En revanche, la communication entre les différents processus risque rapidement de devenir un goulot d'étranglement. En effet, cela génère un nombre considérable d'appels au système et provoquera éventuellement un ralentissement de ce dernier ainsi que des applications qu'il traite.

Les systèmes Ameoba et Mach ont été les premiers micro-noyaux. En voici de plus récents :

- ★ Zircon (Magenta) du nouveau système Fushia de Google [9, 60] ;
- ★ L4 du système Gnu Hurd [15, 42] ;
- ★ Qnx (système d'exploitation temps réel) [67, 3] ;
- ★ XNU le micro-noyau «enrichi» de MacOS X qui est dérivé du micro-noyau Mach [74, 19] ;
- ★ K42 le micro-noyau du système Symbian [69, 70].

Les figures 1.2, 1.3 et 1.4 illustrent quelques architectures pour les noyaux et micro-noyaux.

Pour une information plus complète sur les micro-noyaux, se référer à Wikipedia [51, 47] et à des manuels de système d'exploitation.

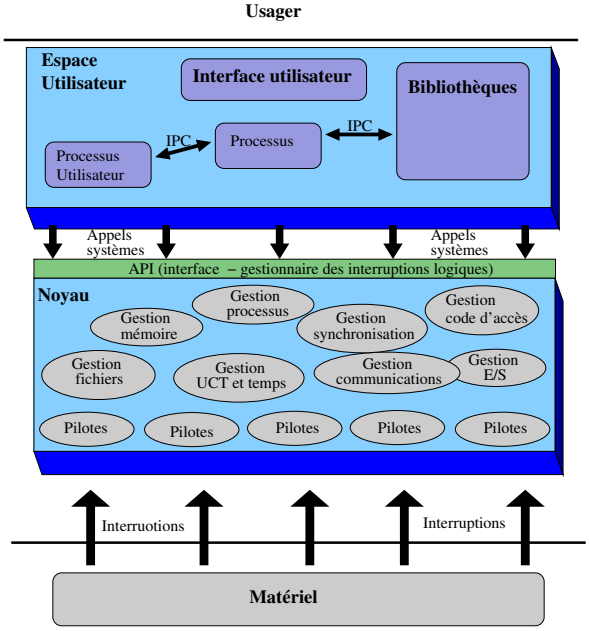


Figure 1.2 – Structure d'un noyau monolithique (Unix)

1.1.4 Implantation du noyau

Le noyau d'un système d'exploitation se doit d'être fiable. Pour atteindre cet objectif, il se doit d'être extrêmement bien isolé de toutes les influences externes non prévues. Il doit aussi se protéger de tous risques d'erreurs internes tels que les problèmes de synchronisation et d'interblocages.

Pour éviter toutes influences externes, l'unique moyen d'accéder au noyau est de s'y rendre via les primitives qu'il fournit, ce qu'on appelle communément l'API. Des exemples d'API de noyaux de systèmes d'exploitation : Win32 (Win64) [79, 80], Unix [16], Posix [77, 78] et Linux [75, 21, 22].

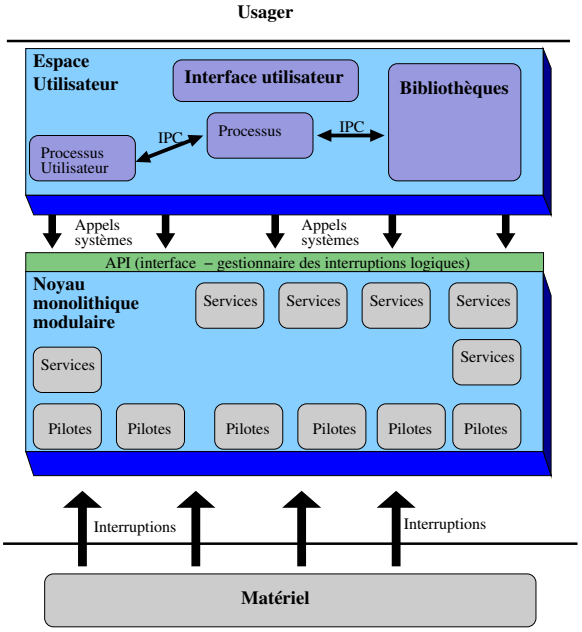


Figure 1.3 – Structure d'un noyau monolithique modulaire (Linux, BSD)

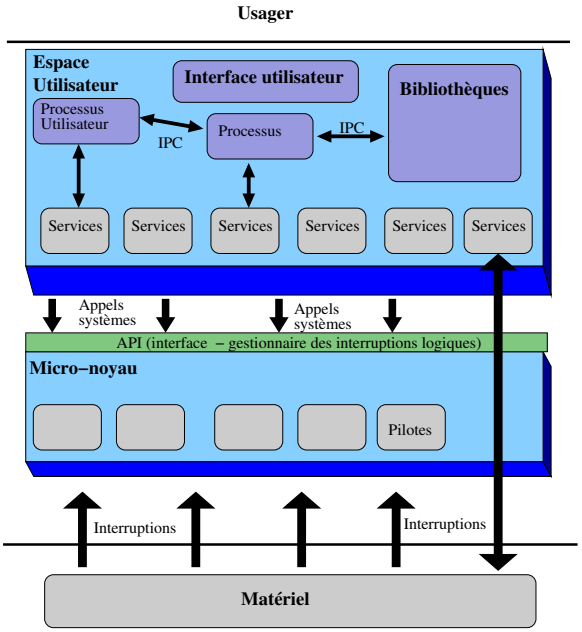


Figure 1.4 – Structure d'un micro-noyau

Ces primitives, sur la plupart des machines, sont implantées grâce aux interruptions logiques (EMT, SVC, ...) et accessibles par l'intermédiaire d'appels au système. Le gestionnaire des interruptions logiques reçoit alors chaque demande qu'il valide et redirige vers la fonction désirée. Aucun accès direct au noyau n'est possible. Ceci garantit sa protection contre toutes influences externes.

À l'interne, plusieurs actions sont prises pour rendre le système plus fiable. Rappelons que les systèmes d'exploitation doivent régulièrement traiter plusieurs événements en parallèle, comme par exemple devoir traiter plusieurs interruptions simultanément. Pour éviter toutes corruptions des données et bien accomplir ses tâches, le noyau doit parfois désactiver (ou masquer) les interruptions. Cela lui permet de faire un traitement sans risquer de se «faire déranger». En effet, certaines interruptions pourraient survenir alors que le noyau exécute une tâche importante (comme le traitement d'une autre interruption) et produire ainsi des effets néfastes sur son bon fonctionnement. On évite donc, par exemple, de débuter le traitement d'une interruption pendant que l'on en traite une autre. On assure ainsi une forme de synchronisation.

Une question qui vient alors à l'esprit, à savoir pendant combien de temps peut-on désactiver les interruptions sans risquer de diminuer la concurrence et de perdre éventuellement des événements possiblement importants. Une période de désactivation trop longue provoquera nécessairement des problèmes difficiles à corriger.

Pour augmenter encore plus la fiabilité, on utilise fréquemment une approche de conception hiérarchique. Le noyau est alors conçu telle une série de machines virtuelles superposées qui s'ajoutent à la machine physique pour en faciliter l'usage et en augmenter les capacités. Ainsi,

- Niveau 0 $\rightarrow$ à la base de la hiérarchie, on retrouve la machine physique ;
- Niveau 1 $\rightarrow$ au niveau directement supérieur, réside une fonction du noyau (généralement la gestion de l'UCT). Au-dessus de ce niveau, on obtient une machine virtuelle avec plusieurs UCTs virtuelles.
- Niveau 2 $\rightarrow$ au niveau suivant, on installe une autre couche (comme la gestion de la synchronisation) qui utilise la machine virtuelle existante pour obtenir une autre machine virtuelle encore plus sophistiquée.
- ...
- Niveau N $\rightarrow$ dernière couche pour compléter le noyau, habituellement le gestionnaire des interruptions logiques qui fournit l'API.

Cette approche demande une analyse poussée car la couche de niveau i ne peut qu'adresser ses demandes de services qu'aux couches des niveaux inférieurs à i. La figure 1.5 présente un exemple de noyau hiérarchique. Dans celui-ci, le noyau est constitué de cinq couches. Le niveau 1 est celui de la gestion des processus, le niveau 2 celui de la gestion mémoire, etc. Notons que les niveaux 1 et 3 concernent tous les deux la gestion des processus. Ce gestionnaire est séparé en deux parties car d'une part, la création d'un processus requiert l'utilisation du gestionnaire de la mémoire pour allouer de l'espace au processus et d'autre part, le gestionnaire de la mémoire a besoin du gestionnaire des processus pour se synchroniser et possiblement se bloquer dans le cas où, par exemple, il n'y aurait pas suffisamment de mémoire pour compléter une requête. On remarque également dans cet exemple que certains services sont localisés à l'extérieur du noyau, tels que la gestion de entrées/sorties, et que l'approche hiérarchique est aussi appliquée sur ceux-ci. La figure 1.6 fournit un second exemple de conception hiérarchique.

Ce type de conception a fait ses preuves quant à la mise au point des logiciels. Il permet de mieux prouver son bon fonctionnement. Cela est surtout dû au sens unique des demandes de services qui permet en particulier d'éliminer les interblocages.

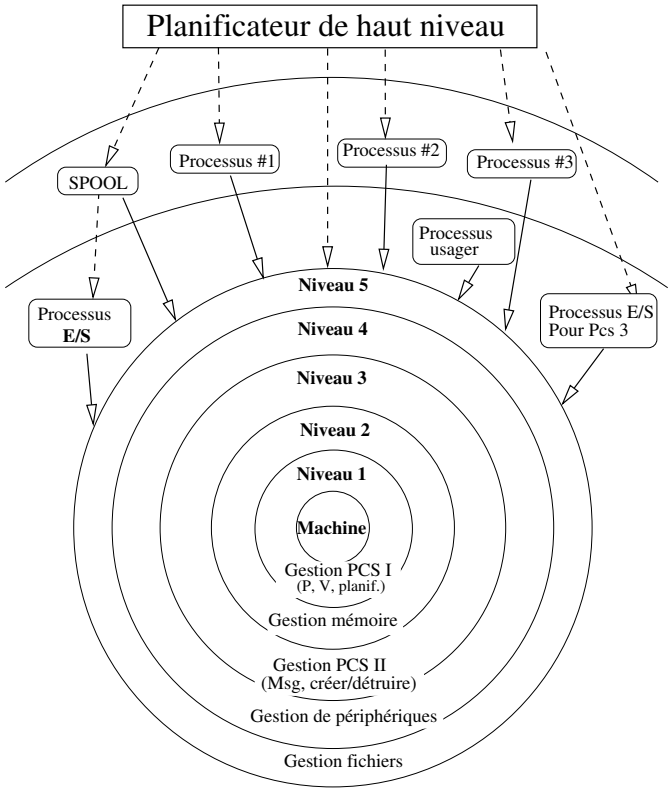


Figure 1.5 – Exemple de noyau avec une structure hiérarchique

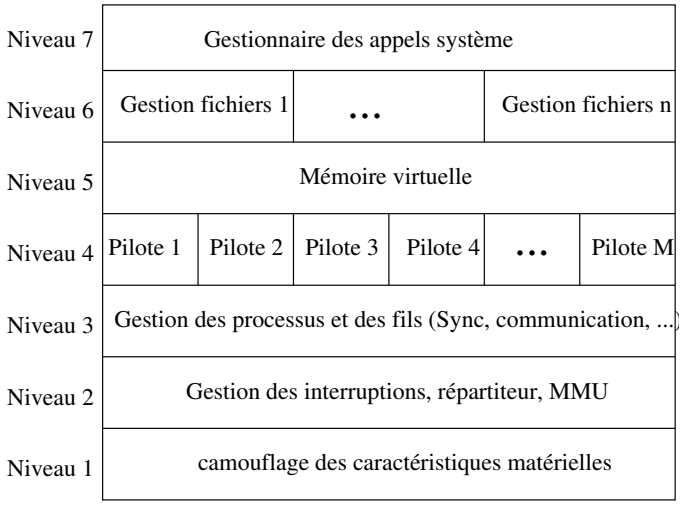


Figure 1.6 – Second exemple de noyau avec une structure hiérarchique

1.2 Rappel : les processus

Au cours «Système d'exploitation» (IFT320), le concept de processus a été présenté. Définir précisément ce concept est relativement ardu car sa signification varie selon les auteurs ou les environnements. Ainsi, le concept de processus auquel réfère Windows est différent de celui utilisé par Linux ou Unix.

Pour les besoins de ce cours, nous définissons un processus comme étant un programme en exécution qui est connu des systèmes d'exploitation ou du noyau. Le processus est donc identifié par un descripteur à l'intérieur du noyau (bloc de contrôle ou PCB). C'est cette définition que Unix et Linux empruntent. Le plus important est de ne surtout pas confondre programme et processus.

Afin de faciliter la tâche du gestionnaire de l'UCT (planificateur court terme), on associe aussi un état à un processus. Parmi les états possibles on retrouve :

- l'état «prêt» ;
- l'état «bloqué» ou «en attente» ;
- l'état «en exécution».

Ce sont les principaux états pouvant être associés à un processus. Il en existe plusieurs autres qui varient d'un système à l'autre.

Un processus est donc une entité dynamique qui naît et qui meurt. Il peut naître au démarrage du système (comme le processus «init» de Unix) ou pendant son opération, et il peut mourir soit lors de l'arrêt du système ou pendant son opération. Cependant, un processus ne fait pas que naître et mourir. En effet, plusieurs opérations existent afin de manipuler le processus et ce, de différentes façons.

Un processus est aussi fréquemment considéré comme un type abstrait de données (souvent appelé «`class`» dans les langages). Comme vous le savez, un type abstrait de données encapsule les données avec les opérations qui les manipulent. Pour un processus, on retrouve donc :

- Données
  - ▶ Descripteur (PCB)
    - ✓ l'identificateur (pid) ;
    - ✓ l'état ;
    - ✓ le compteur ordinal (pc) ;
    - ✓ les registres ;
    - ✓ etc.
  - ▶ Espaces d'adresses
    - ✓ le code ;
    - ✓ la pile (données locales) ;
    - ✓ les données statiques, globales ou constantes ;
    - ✓ les données dynamique (heap ou amas).

  La figure 1.7 illustre l'organisation des données d'un processus.
- Les opérations
  - ▶ Création/destruction
  - ▶ Arrêt/démarrage d'un processus ;

- Suspension/reprise de l'exécution d'un processus ;
- Blocage/réveil d'un processus (synchronisation, E/S) ;

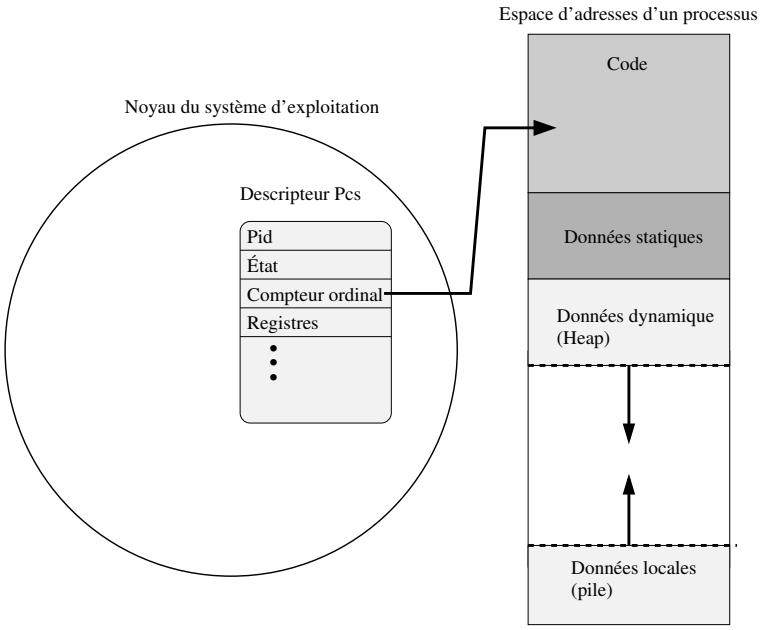


Figure 1.7 – Structure d'un processus

1.2.1 Création et destruction de processus

La création et la destruction constituent les deux principales opérations effectuées sur les processus. La création d'un processus implique un certain travail soit :

- la création du descripteur (bloc de contrôle ou PCB)
  Cette opération crée le descripteur et initialise toute l'information qu'il doit contenir telle qu'un identificateur, un état initial, etc.
- La création de l'espace d'adresses
  Cela comprend l'allocation de l'espace et le chargement du programme lui-même (code et données initiales) à partir d'un fichier sur disque vers la mémoire centrale.

Hiérarchie de processus

Un processus est généralement créé par un autre processus. Cela introduit donc une relation parent/enfant entre les différents processus. La seule exception à ce fait est lorsqu'un processus est créé au démarrage du système. C'est l'unique situation où un processus n'a aucun parent.

La relation parent/enfant est tellement fréquente que les diverses relations entre les processus ont été spécifiées. Soit un processus P, sa descendance est définie de la façon suivante :

- un processus Q créé par P appartient à la descendance de P ;
- tout processus créé par un processus Q appartenant à la descendance de P appartient à la descendance de P ;
- il n'existe aucune autre façon de définir un processus appartenant à la descendance de P.

De façon plus spécifique, le processus P s'appelle le parent et le processus Q, l'enfant. La relation entre P et sa descendance est représentée par l'arbre de création de la figure 1.8

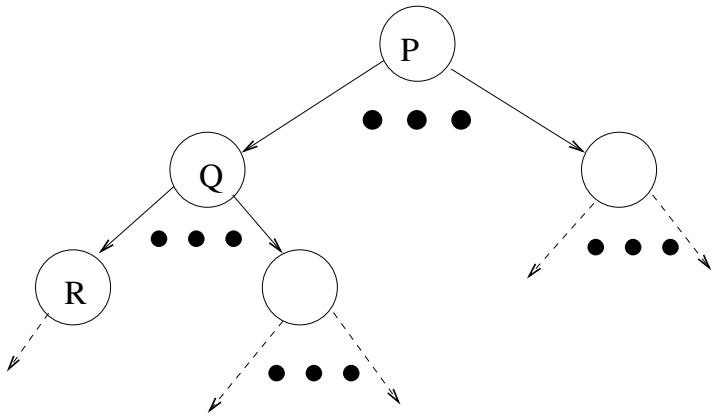


Figure 1.8 – Relation entre les processus

Options lors de la création d'un processus

Selon les options ou les paramètres spécifiés, la création d'un processus produira différents résultats :

- Au niveau de l'exécution ;
  - ▶ Exécution synchrone ;

    Le processus créateur attendra que le processus qu'il a démarré se termine avant de poursuivre son exécution.

    Par exemple, sous les systèmes Unix, chaque commande entrée à la console crée un nouveau processus. Normalement, par défaut, tant que la commande n'est pas terminée, la console est «gelée». C'est donc une création de processus synchrone.
  - ▶ Exécution asynchrone ;

    Le processus créateur poursuit son exécution concurremment avec le ou les processus qu'il a créés.

    Encore une fois, sous les systèmes Unix, chaque commande entrée à la console qui est suivie d'un «&» démarre le processus de façon asynchrone. La console est donc libérée et il est possible d'exécuter plusieurs commandes en parallèle.
- Au niveau du partage ;

  Comme le processus qui vient d'être créé doit accomplir un certain travail pour son parent, il est donc utile que les deux puissent se partager de l'information. Voici les différentes situations

possibles :

- ▶ Partage complet ;
  Le parent et son enfant partagent toutes leurs données (variables et constantes).
- ▶ Partage partiel ;
  Le parent et l'enfant ne partagent seulement qu'un sous-ensemble de leurs données (variables et constantes) ;
  C'est le cas du système Unix où un parent et son enfant ne partagent seulement que la table des fichiers ouverts.
- ▶ Aucun partage ;
  Il est aussi envisageable que le parent et l'enfant ne partagent aucune donnée. Dans ce cas, il devront communiquer par l'intermédiaire de messages.

- Au niveau de la protection ;
  Lors de la création, l'allocation d'un pouvoir initial à l'enfant constitue une opération délicate. Si l'enfant possède plus de pouvoir que son parent, cela risque de générer un problème de sécurité. Par contre, si l'enfant n'a pas assez de pouvoir, il lui sera difficile de remplir la tâche pour laquelle il a été créé.
- Au niveau de la communication ;
  Lorsque les processus doivent communiquer par messages, il est fréquent d'utiliser les relations parents/enfants pour limiter les communications, par exemple, permettre à un enfant de communiquer seulement avec son parent ou seulement avec les autres enfants de son parent.

Destruction de processus

La destruction d'un processus survient lors de trois circonstances distinctes :

- fin normale
  Cette destruction est provoquée par le processus lui-même à la fin de son exécution ;
  Soit cela est due à une fin normale du programme (`exit(0)` ou `return`) ou à une fin anormale suite à une erreur détectée et traitée par le programme (`exit(valeur)`) ;
- fin anormale ;
  Cette destruction est provoquée par le système d'exploitation lors de la capture d'une erreur irrécupérable («*segmentation fault*», «*bus error*», ...).
- fin provoquée ;
  Cette destruction est provoquée par un autre processus autorisé tel son parent. Elle se fait sous Unix avec la commande «`kill`».

Lorsqu'un processus P est détruit, on doit prévoir ce qui adviendra de sa descendance. Il y a deux possibilités :

- poursuite de l'exécution ;
  Dans ce cas, on permet à tous les processus Q appartenant à la descendance d'un processus P de poursuivre leur exécution.
- fin de l'exécution ;
  Ici, on détruit tous les processus Q appartenant à la descendance d'un processus P.

Certains systèmes sont configurés pour appliquer l'une ou l'autre de ces possibilités.

Lors de sa destruction, on se doit aussi de faire disparaître toutes traces du processus, soit détruire son descripteur et libérer son espace d'adresses.

Étude de cas : Unix

Sous le système d'exploitation Unix, la création d'un processus est réalisée à l'aide de deux commandes distinctes. La première étape de la création d'un processus consiste à appeler la commande «`Fork`». Cette commande est conçue pour créer un nouveau processus, soit un clone du parent. Ceci signifie que le code et les données sont copiés dans le nouvel espace d'adresse attribué à l'enfant. De plus, son compteur ordinal pointe au même endroit dans le code que celui du parent. Le parent poursuit son exécution à l'instruction suivante (celle qui suit le «`Fork`») et l'enfant démarre son exécution au même endroit (comme s'il venait d'appeler la fonction «`Fork`»). Ce qui permet de distinguer l'un de l'autre, c'est le code de retour de la fonction. Le parent reçoit l'identificateur de son enfant (`pid`) et l'enfant reçoit un code spécifique (généralement 0). Le programme 1.1 présente le code permettant de distinguer le parent de l'enfant grâce au code de retour.

```
1 pcsid = fork();
2 if (pcsid == -1)
3     // erreur lors de la création....
4 else if (pcsid == 0)
5    // code pour processus nouvellement créé (enfant qui exécute le code du parent)
6 else // code du parent
```

Programme 1.1 – Exemple d'utilisation du Fork en Unix

La seconde étape de la création d'un nouveau processus consiste à charger le code à exécuter. Lorsqu'un processus détecte qu'il est un enfant nouvellement créé, il doit charger le code nécessaire à l'accomplissement de sa tâche. On se rappelle qu'au départ le nouveau processus exécute le code de son parent. Pour charger un nouveau programme, il utilise une des différentes versions de la commande «`exec`», soit :

- `exec, execl, execv, execve` et `execle` ;

  Toutes ces commandes permettent de charger un exécutable à partir d'un fichier. La différence entre celles-ci se situe au niveau des paramètres. Le programme 1.2 présente un exemple d'utilisation de la commande «`execl`».

- `execlp` et `execvp` ;

  On réfère à ces deux commandes lorsque le programme à charger n'est pas un «exécutable» mais un «script» (shell script par exemple).

```
1 pcsid = fork();
2 if (pcsid == -1)
3 {   // erreur lors de la création....      }
4 else if (pcsid == 0)
5 {  // c'est l'enfant... Chargement du nouveau programme
6    execl(fichier, fichier,  arg1, arg2, arg3, (char *)0)
7 else // code du parent
```

Programme 1.2 – Exemple d'utilisation du Fork/exec en Unix

Lorsqu'un processus exécute une commande `fork`, le parent poursuit son exécution en parallèle avec son enfant. Le parent et l'enfant partagent la table des fichiers ouverts mais ne partagent pas les données (pile, données statiques, données dynamiques, ...). Toutefois, comme le code associé aux

processus est en lecture seulement, il est possible de le partager par l'intermédiaire de la mémoire virtuelle.

Étude de cas : Windows

Windows fournit la commande «`CreateProcess`» qui crée un nouveau processus. Celle-ci est cependant fournie dans une bibliothèque. Au niveau du noyau, cette commande appelle deux primitives du noyau, «`NTCreatProcess`» et «`NTCreateThread`»

Nous verrons dans la section suivante que Windows fait une distinction, plus claire et concise que Unix, entre le concept de processus (l'environnement contenant l'information sur le processus) et celui de son exécution (qui elle, est associée au fil d'exécution).

1.2.2 Autres opérations

Arrêt/Démarrage

Ces commandes permettent d'arrêter l'exécution d'un processus et de tout réinitialiser afin de reprendre l'exécution au début.

Suspension/Reprise (suspend/resume)

Ces deux opérations permettent d'arrêter l'exécution d'un processus mais sans rien détruire afin de reprendre l'exécution exactement au même endroit ultérieurement. Elles introduisent en général un nouvel état pour un processus.

Ainsi, sous les systèmes Unix, il est possible de suspendre un processus à partir d'une console grâce à la commande «`crtl-z`». La reprise de son exécution se fait grâce aux commandes «`bg`» (background) ou «`fg`» (foreground).

Notons que le système d'exploitation a la possibilité de faire appel à ces commandes pour contrôler la charge de travail. Ainsi, si celle-ci est trop lourde, le système suspend temporairement l'exécution de plusieurs processus. Leur exécution reprendra plus tard lorsque la charge sera moins importante.

1.3 Les fils d'exécution (thread)

Le concept de «fil d'exécution» est équivalent à une création de processus dans lequel l'enfant partage tout l'espace d'adresse de son parent. Les figures 1.9 et 1.11 illustrent le principe des fils d'exécution à l'intérieur d'un même espace d'adresses.

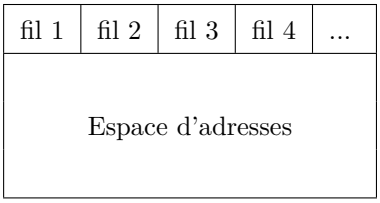


Figure 1.9 – Architectures multi-fils

On qualifie aussi les fils d'exécution de «processus légers» car leur création s'avère beaucoup plus rapide que celle d'un processus normal. En effet, comme ils s'exécutent dans un espace d'adresses existant, aucune opération d'allocation d'espace ni de chargement de code n'est requise. Il suffit de leur allouer un bloc de contrôle (descripteur) et quelques autres ressources indispensables telles que la pile et l'amas (données dynamiques). La figure 1.10 compare les concepts de processus et de fils. Le cercle représente un «espace d'adresses» et le «fil» une exécution à l'intérieur de cet espace d'adresses. La partie (a) de la figure présente un processus standard tel que défini et utilisé par Unix et Linux, i.e. un processus avec un seul fil d'exécution. La partie (b) illustre un environnement comprenant plusieurs processus standards (avec un seul fil d'exécution). Les figures 1.10(c) et 1.10(d) présentent respectivement un processus et plusieurs processus admettant de multiples fils d'exécution.

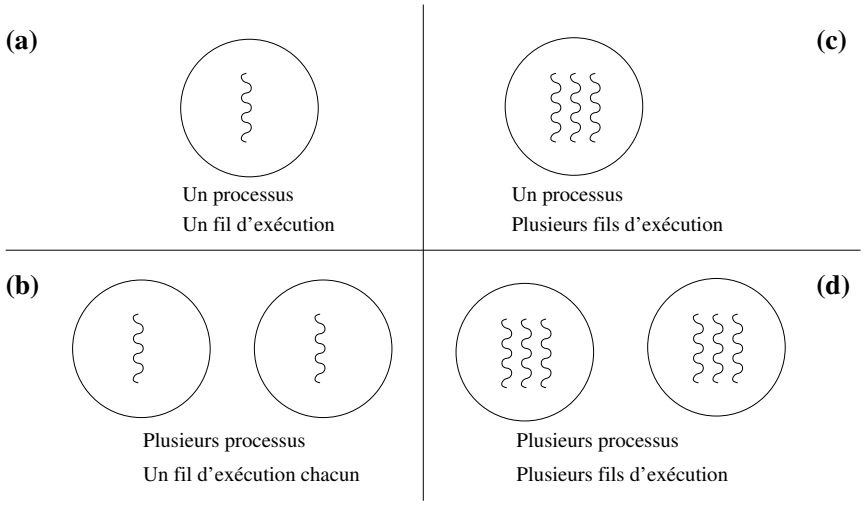


Figure 1.10 – Comparaison entre processus mono-fil et multi-fils

La figure 1.11 présente une certaine organisation d'un système multi-fils. Cette organisation ressemble à celle des systèmes Solaris et Windows. En effet dans Windows, un processus ne représente qu'un espace d'adresses. Il faut créer un premier fil pour démarrer l'exécution. Dans le système Linux, chaque fil possède son propre PCB.

Un fil d'exécution est donc un point de contrôle dans le système d'exploitation ou une unité de base d'utilisation de l'UCT. La figure 1.12 indique ce qui est partagé ou non entre les différents fils d'un processus.

Pour éviter toute confusion, nous identifions l'environnement d'exécution d'un fil par le terme «tâche». Ainsi une tâche avec un seul fil d'exécution est équivalente à un processus tel que défini précédemment.

Voici certains avantages que possède un fil d'exécution par rapport à un processus :

- La création d'un fil est beaucoup plus rapide que celle d'un processus ;

  Le tableau 1.13 fournit quelques exemples de temps de démarrage de fils et de processus [7] [4].

4. Je n'ai pu trouver de références récentes à ce sujet. Toutefois les proportions sont tout de même réalistes et représentatives. Lorsque des références plus récentes seront identifiées, le tableau sera mis à jour.

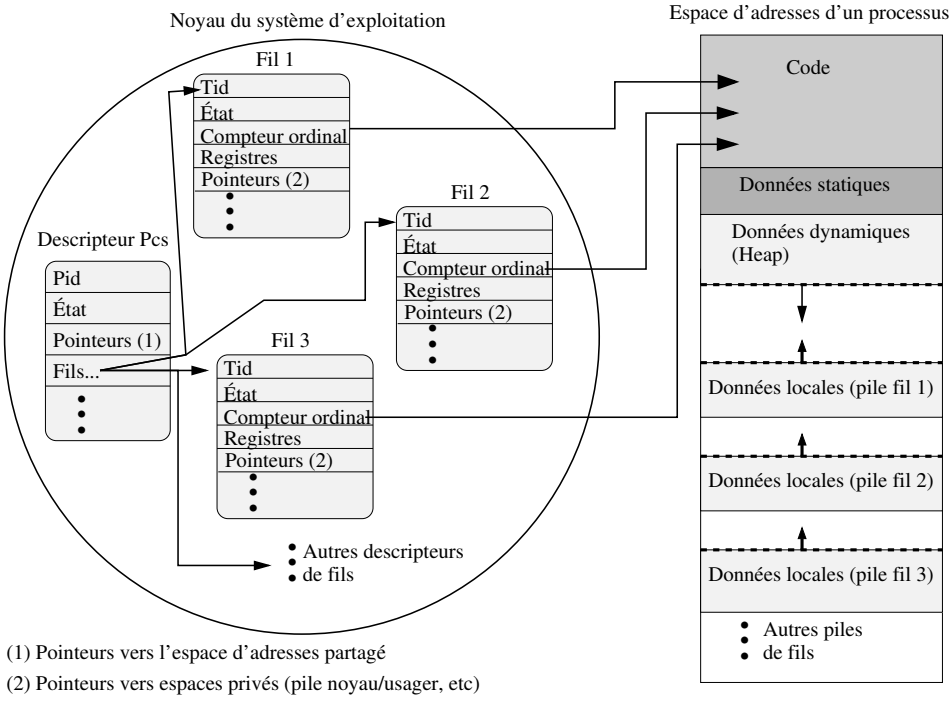


Figure 1.11 – Structure possible d'un système multi-fils

Partagé entre les fils	Non partagé entre les fils
Code (espace d'adresses)	Registres
Données statiques (espace d'adresses)	Données locales (piles : espace d'adresses)
Ressources (fichiers, sockets, ...)	Données locales (pile : noyau)

Figure 1.12 – Ressources partagées ou non entre les fils d'exécution

On remarque que le démarrage des fils est parfois plus de 10 fois plus rapide.

Architecture	Système d'exploitation	Temps démarrage fil	Temps démarrage processus
Power 3 (375 MHz)	Unix	7,46	61,94
Power 4 (1,5 GHz)	Unix	1,49	44,08
Power 5 (1,9 GHz)	Unix	1,13	50,66
Intel Xeon (2,4 GHz)	Unix	1,70	23,81
Intel Itanium (1,4GHz)	Unix	2,10	23,61

Figure 1.13 – Architectures multi-fils

- La répartition de l'UCT (changement de contexte) entre des fils est aussi plus rapide que celle entre des processus. En effet, comme les fils se partagent la même mémoire, il y aura moins de changements au niveau de la mémoire virtuelle (faute de pages, TLB, ...).
- Le partage d'information est beaucoup plus efficace entre les fils d'un même processus qu'entre deux processus distincts.
- ...

Toutefois, les fils d'exécution s'avèrent (parfois) plus difficiles à programmer car justement ils se partagent des données. Les risques d'erreur dus à des problèmes de synchronisation sont plus élevés.

Les fils d'exécution implantent, dans les serveurs, l'équivalent de la réentrance au niveau des fonctions. La réentrance est une caractéristique (voir l'encadré) qui permet à une fonction, routine ou programme, d'être exécutée simultanément par plusieurs processus. Le code d'une fonction ré-entrante, une fois chargé en mémoire, peut donc être partagé et utilisé par plusieurs processus simultanément. Un fonction non ré-entrante ne peut pas être partagée entre plusieurs processus.

Définition : Réentrance (inspiré de [53, 52])

On dit qu'un fonction est ré-entrante si elle ne se modifie pas elle-même. En fait, la réentrance est la propriété qui rend possible son utilisation simultanée par plusieurs fils d'exécution ou processus. Cela permet d'éviter la duplication en mémoire d'un programme utilisé simultanément par plusieurs applications.
La réentrance s'atteint particulièrement en séparant clairement le code et les données (segment de code et segment de données). Pour plus d'informations, voir l'annexe A.

1.3.1 Les différents types de fils

Il existe deux types de fils d'exécution : les fils de niveau noyau et les fils de niveau usager. Les «processus légers» (*Light Weight Processes - LWPs*) est une autre expression pour identifier les fils de niveau noyaux. Toutefois, certains auteurs font une distinction entre les LWPs et les fils de niveau noyau. Dans ce cas, les fils de niveau noyau sont gérés et réservés au système d'exploitation et les LWPs sont des intermédiaires, assurant le lien entre les fils de niveau usager et les fils de niveau noyau. C'est le cas par exemple du système d'exploitation Solaris. **Pour les besoins du cours, nous ne ferons aucune distinction entre les fils de niveau noyau et les LWPs.**

Ainsi, les fils de niveau noyau sont des fils gérés par le système d'exploitation. Pour les créer, il faut effectuer un appel système. Ce dernier crée alors un descripteur pour chacun des fils et les gère de façon indépendante comme pour les processus (ils ont leur état, leur tranche de temps, etc). De même, leur destruction requiert un appel système et leur planification force une intervention du système.

Les fils de niveau usager ne font aucunement appel au système d'exploitation. Ils sont gérés par des bibliothèques intégrées dans l'application. La bibliothèque fournit alors une interface pour manipuler les fils (création/destruction/...) et s'occupe de la répartition du temps UCT entre les fils. Aucun appel au système d'exploitation n'est requis pour cette gestion. La figure 1.14 présente une organisation possible pour les fils de niveau usager.

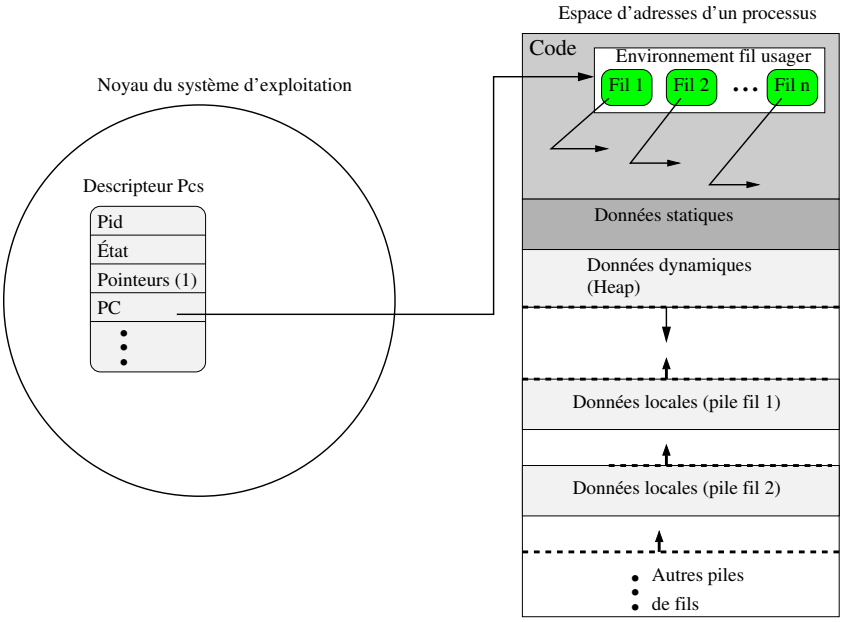


Figure 1.14 – Structure pour les fils de niveau usager

Énumérons ici les avantages et inconvénients des fils de niveau usager par rapport aux fils de niveau noyau :

- les opérations de création et de destruction sont très rapides car elles n'impliquent aucun appel au système ;
- les changements de contexte entre les fils sont aussi très rapides car ils n'impliquent aucune intervention du système d'exploitation ;
- une opération bloquante d'un fil bloque le processus en entier y compris tous les autres fils de niveau usager.

Modèle de correspondance : fils usager vs fils noyau

Plusieurs modèles permettent de faire correspondre les fils de niveau usager aux fils de niveau noyau.

1. Plusieurs à un ;

   Le premier modèle et le plus simple, est celui dans lequel tous les fils de niveau usager d'une tâche sont rattachés à un seul fil de niveau noyau. C'est le modèle de fil de niveau usager «pur» tel que présenté par la figure 1.15. Ici, si un fil de niveau usager se bloque, tous les autres fils de la même tâche sont aussi bloqués. Remarquons que, dans ce mode de fonctionnement, il est impossible de profiter des multiples cœurs ou processeurs d'un ordinateur moderne. C'est pourquoi, peu de systèmes supportent ce choix aujourd'hui.

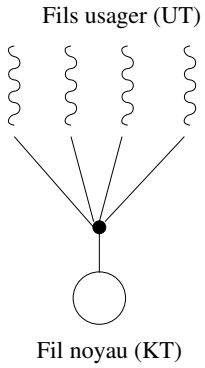


Figure 1.15 – Modèle de correspondance «plusieurs à un» pour les fils d'exécution.

   Le modèle de fils «plusieurs à un» est souvent appelé «Green Threads» [46]. Un exemple de bibliothèque fonctionnant selon ce principe fut la «GNU Portable Thread» [45, 14].

2. Un à un

   Dans ce second modèle, chaque fil de niveau usager est rattaché à un fil de niveau noyau. La performance de celui-ci est moindre que le précédent car chaque création implique un appel au système d'exploitation. Toutefois lorsqu'un fil usager se bloque, les autres fils sont en mesure de poursuivre leur exécution. De plus, ces fils profitent des architectures multi-cœurs. La figure 1.16 illustre ce mode de fonctionnement implanté par Linux et Windows de 95 à XP.

3. Plusieurs à plusieurs

   Dans ce troisième modèle, voir la figure 1.17, la correspondance est établie entre plusieurs fils de niveau usager à un nombre égal ou inférieur de fils de niveau noyau. Le nombre de fils de niveau usager est généralement plus élevé que le nombre de fils de niveau noyau. Un nouveau fil de niveau noyau est créé au besoin pour éviter qu'un blocage au niveau usager n'affecte tous les fils de ce niveau. C'est le système d'exploitation qui décide du nombre de fils de niveau noyau. Ce modèle profite des avantages des modèles 1 et 2 sans souffrir de leurs inconvénients. Ainsi, les utilisateurs ne sont pas limités sur le nombre de fils usagers, un fil qui se bloque ne bloque pas tous les autres et il est possible de profiter des architectures multi-cœurs.

4. Deux niveaux

   Le modèle à deux niveau constitue une variation des modèles précédents combinant le «plusieurs à plusieurs» et le «un à un». Comme le montre la figure 1.18, il est possible de rattacher

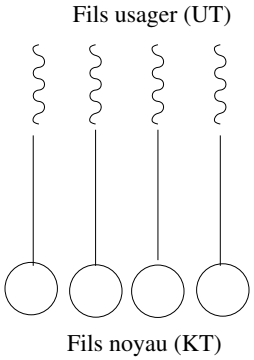


Figure 1.16 – Modèle de correspondance un à un pour les fils d'exécution.

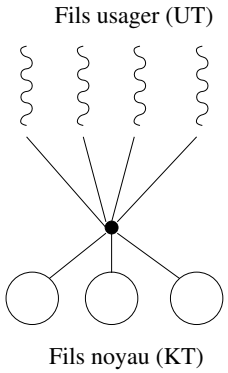


Figure 1.17 – Modèle de correspondance «plusieurs à plusieurs» pour les fils d'exécution.

plusieurs fils usagers à plusieurs fils noyau, mais aussi de rattacher exclusivement un fil de niveau usager à un fil de niveau noyau (bounded thread). Cela s'avère utile dans certaines situations où la performance d'un fil est requise.

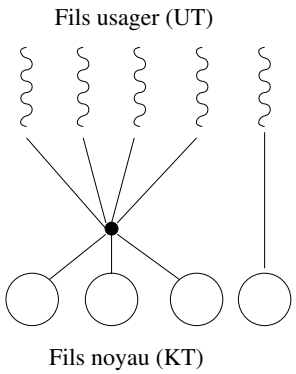


Figure 1.18 – Modèle de correspondance à deux niveaux (two-tier) pour les fils d'exécution.

Ce modèle est (ou était) implanté par les systèmes d'exploitation Windows (7 à 10), Solaris (jusqu'à la version 9), IRIX, HP-UX, Tru64 Unix.

Il faut savoir que le vocabulaire qui identifie les fils de niveau usager et noyau varie d'un système à l'autre. Le tableau 1.19 introduit la terminologie propre à certains systèmes.

Système d'exploitation	Fils de niveau usager	Fils de niveau noyau
Solaris	Thread	Light weigth process (LWP)
Posix	Thread	Light weigth process (LWP)
Linux	Thread	Light weigth process (LWP)
Windows	Light weigth process (LWP)/ Fibre	Thread

Figure 1.19 – Ressources partagées ou non entre les fils d'exécution

1.4 Exemples de bibliothèques de fil d'exécution

La plupart des systèmes d'exploitation offrent une interface (API) pour manipuler les fils d'exécution. Celle-ci s'accède soit directement par l'interface fournie par le système, soit par l'intermédiaire de bibliothèques.

1.4.1 Posix

À venir...

1.4.2 Windows

Sous Windows, un processus représente un environnement d'exécution. Celui-ci peut contenir un ou plusieurs fils d'exécution.

À venir...

1.4.3 Linux

À venir...

1.4.4 Solaris

À venir...

1.5 Résumé sur les fils

Le tableau 1.20 met en évidence les différences, avantages et inconvénients des processus versus les fils d'exécution.

Base de comparaison	Processus	Fil d'exécution
Définition	C'est un programme en exécution (processus lourd)	C'est un «segment» de processus en exécution (processus léger)
Temps pour changement de contexte	Plus long pour un processus	Plus court pour un fil
Partage de mémoire	Aucun partage par défaut (isolement complet)	Partage quasi toute la mémoire
Temps de communication	Plus long si aucun partage de mémoire	Plus court en général
Blocage	Le blocage d'un processus ne bloque pas les autres	Le blocage d'un fil usager bloque tous les autres fils. Blocage d'un fil noyau ne bloque pas les autres.
Utilisation de ressources	Consomme plus de ressources	Consomme moins de ressources
Dépendance	Indépendance complète	Dépendance entre eux
Partage de code et de données	Aucun partage	Partage le code, les données statiques et dynamiques et les fichiers
Traitement par le système	Séparation complète	Aucun pour fil usager. Distinct pour fil noyau.
Temps de création	Plus long	Plus court
Temps pour terminaison	Plus long	Plus court

Figure 1.20 – Ressources partagées ou non entre les fils d'exécution

Pour plus d'informations sur les fils d'exécution, se référer aux documents suivants [1, 2, 4, 13, 17, 23, 32, 54] et à d'autres sites sur le Web.

1.6 Les types de parallélismes au niveau matériel

Rappelons que tous les ordinateurs modernes sont basés sur des modèles architecturaux mis au point par Turing et Von Neuman.

Alan Turing : père de l'informatique

Les contributions de Alan Turing [57] au domaine de l'informatique sont énormes. Il est considéré par plusieurs comme le père de l'informatique et de l'intelligence artificielle. D'ailleurs le prix le plus prestigieux en informatique porte son nom, le «Turing Award». Ses principales contributions :

- la machine de Turing, le modèle théorique de tous les ordinateurs modernes ;
- la conception d'un des premiers ordinateurs (la Bombe) ;
- le bris du code allemand basé sur Énigma pendant la seconde guerre mondiale ;
- les fondements de l'intelligence artificielle.

John Von Neumann

John von Neumann [63] a apporté d'importantes contributions en mécanique quantique, en analyse fonctionnelle, en théorie des ensembles, en informatique, en sciences économiques et dans maints autres domaines des mathématiques et de la physique. En informatique, il a contribué, en 1945, au développement d'un des premiers ordinateurs, l'EDVAC, qui implantait la notion de programmes et données emmagasinés en mémoire. Ce type d'architecture, reproduit dans tous les ordinateurs modernes, est appelé «l'architecture de Von Neumann».
Von Neumann attribuait lui-même cet architecture à Alan Turing,

La plupart des ordinateurs modernes font appel à des mécanismes variés dans le but de «gagner en vitesse». Parmi ces mécanismes, on retrouve diverses formes de parallélisme :

- le parallélisme au niveau des instructions ;

  À ce niveau, les concepts de pipeline, superscalaire et de VLIW permettent d'exécuter simultanément plusieurs instructions.

- le parallélisme au niveau des fils d'exécution (thread) ;

  Ce concept implante des processeurs virtuels en multiplexant les processeurs matériels. Cette approche exige un support du système d'exploitation car celui-ci doit planifier autant de fils sur les processeurs réels que sur les virtuels.

- Le parallélisme de type multi-cœurs ;

  La plupart des processeurs modernes ont plusieurs cœurs. Même les processeurs des téléphones cellulaires contiennent de quatre à huit cœurs. Ce sont de réels processeurs qui requièrent un support du système d'exploitation pour la gestion et la planification.

- le parallélisme de type multi-processeurs ;

Ce concept est similaire à celui des multi-cœurs sauf qu'ici, les processeurs se retrouvent sur des puces distinctes. Le support du système d'exploitation est requis au même titre que pour le multi-cœurs.

Voyons en détails le fonctionnement de ces différents concepts.

1.6.1 Parallélisme au niveau des instructions (Instruction Level Parallelism - ILP)

À ce niveau, on essaie d'extraire le parallélisme des instructions d'un même programme. On exécute donc simultanément plusieurs instructions d'un même programme lorsque cela est possible. Il existe trois approches pour y parvenir, soit le pipeline, le super-scalaire ou le VLIW.

Le pipeline

Le concept de pipeline [5] consiste à diviser l'exécution des instructions en étapes et permet ainsi d'en traiter plusieurs en même temps, mais chacune à une étape différente. Un exemple de division simple consiste à séparer l'exécution d'une instruction en quatre étapes, soit celle du chargement, celle du décodage, celle de l'exécution proprement dite et finalement celle de l'écriture des résultats. En procédant de cette façon, on traite théoriquement jusqu'à quatre instructions simultanément, augmentant ainsi la vitesse d'un ordinateur d'un facteur de quatre (toujours théoriquement).

La figure 1.21 illustre le temps requis pour exécuter une instruction sans le concept de pipeline. Elle occupe seule le processeur du début à la fin. La figure 1.22 illustre un pipeline qui traite simultanément quatre instructions (le traitement est divisé en quatre étapes). On remarque que pour un temps donné, quatre instructions sont exécutées concurremment plutôt qu'une seule.

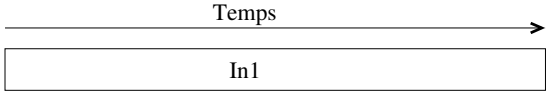


Figure 1.21 – Exécution d'une instruction sans pipeline

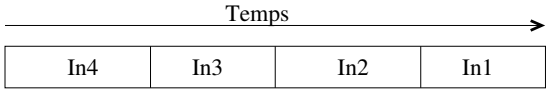


Figure 1.22 – Exécution d'instructions avec pipeline

Ce gain est toutefois théorique car des instructions dites hasardeuses risquent de forcer le pipeline à se vider, réduisant de ce fait le gain espéré en performance. Les principales instructions hasardeuses sont les instructions de branchement, pour lesquelles il est impossible de toujours prédire correctement le résultat, et les instructions qui accèdent à la mémoire centrale («`load/store`») entraînant de longues attentes afin d'obtenir l'information. En réalité, il est très probable que le pipeline soit vide de 20% à 30% du temps.

5. Ce concept a déjà été présenté dans le cours de programmation système

Le super-scalaire

Ce concept introduit la notion «d'unités fonctionnelles» qui consiste à diviser un unique processeur en unités fonctionnelles spécialisées qui travaillent en parallèle. On retrouve dans la plupart des architectures des unités fonctionnelles spécialisées :

- pour le traitement des nombres entiers ;

  Ces unités servent à exécuter les instructions qui manipulent des nombres entiers. Parmi celles-ci, on retrouve principalement les opérations arithmétiques sur les nombres entiers (`add`, `sub`, `mul`, ...). Certaines architectures possèdent plusieurs unités fonctionnelles entières.

- pour le traitement des nombres en virgule flottante ;

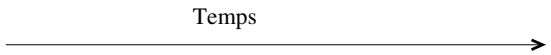
  Ces unités servent à exécuter les instructions qui manipulent des nombres réels représentés par des nombres en virgule flottante. Parmi celles-ci, on retrouve principalement les opérations arithmétiques sur les nombres en virgule flottante (`addf`, `subf`, `mulf`, ...). Certaines architectures contiennent plusieurs unités fonctionnelles de type virgule flottante.

- pour le traitement des instructions de chargement et de stockage (`load`, `store`, ...) ;

- pour le traitement des instructions de branchement (`ba`, `beq`, `ble`, ...).

Il est donc possible d'exécuter simultanément, à l'intérieur d'un même processeur, des instructions de différents types (entier, réel, branchement et chargement) lorsque le processeur possèdent plusieurs unités fonctionnelles, et aussi des instructions de même type (instructions sur les entiers par exemple) dans le cas où le processeur possède plusieurs unités fonctionnelles identiques.

De plus, comme chaque unité fonctionnelle possède éventuellement son propre pipeline, cela augmente de façon substantielle le nombre d'instructions traitées en un temps donné. La figure 1.23 présente une architecture comprenant quatre unités fonctionnelles ayant chacune un pipeline de profondeur quatre (4 étapes). Cette architecture a donc la capacité théorique d'exécuter jusqu'à 16 instructions simultanément.



UF1	In13	In9	In5	In1
UF2	In14	In10	In6	In2
UF3	In15	In11	In7	In3
UF4	In16	In12	In8	In4

Figure 1.23 – Exécution optimale d'instructions avec super-scalaire et pipeline

Le défi de cette architecture, en plus d'optimiser les opérations du pipeline, consiste à identifier suffisamment d'instructions à exécuter en parallèle. En effet, cela représente un défi car ces instructions doivent à la fois être indépendantes (aucun partage de données ou de registres), ne pas être hasardeuses (pipeline) et, de plus, elles doivent appartenir au même programme. En réalité, étant donné ces exigences, les unités fonctionnelles ne sont réellement utilisées qu'à un faible pourcentage de leur capacité, comme constaté à la figure 1.24.

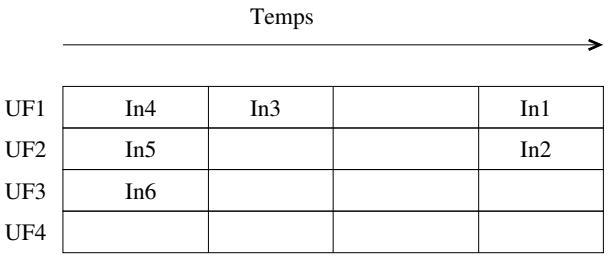


Figure 1.24 – Exécution d'instructions avec super-scalaire et pipeline

Notons ici un autre inconvénient de ce type d'architecture qui traite plusieurs instructions en parallèle : c'est le risque que les instructions soient exécutées dans le désordre (un ordre qui ne respecte pas l'ordre indiqué dans le programme). Nous aborderons plus loin des conséquences d'un tel ré-ordonnancement.

Il est important de se souvenir que l'approche ILP permet d'obtenir du parallélisme au niveau des instructions d'un même programme. Par exemple, le programme 1.3 contient trois instructions qui peuvent s'exécuter en parallèle car elles ne partagent aucune donnée et concernent trois unités fonctionnelles distinctes. En revanche, le programme 1.4 ne contient aucune instruction indépendante et ne peut donc profiter au maximum des multiples unités fonctionnelles.

```
1 ADD  R1, R2, R3
2 FADD F1, F2, F3
3 BA test
```

Programme 1.3 – Exemple de programme contenant des instructions indépendantes

```
1 ADD  R1, R2, R3
2 SUB  R3, R4, R5
3 CONV R4, F1
4 FMUL F1, F2, F3
5 CMP  F3, F6
6 BEQ  test
```

Programme 1.4 – Exemple de programme contenant des instructions avec dépendances

1.6.2 Parallélisme au niveau des fils d'exécution (Thread Level Parallelism : TLP)

Tandis que l'approche ILP tente d'exploiter le parallélisme au niveau d'un même programme, le parallélisme au niveau des fils [64, 68, 71, 36], lui, tente d'exploiter le parallélisme en utilisant des instructions provenant de plusieurs programmes. Les instructions extraites de programmes distincts, n'ayant théoriquement aucune dépendance, devraient avoir la capacité de s'exécuter en parallèle pour ainsi augmenter le rendement des unités fonctionnelles et diminuer le temps d'exécution moyen de différents processus. En cela, le TLP est similaire au concept de multi-tâches avec réquisition implanté dans les systèmes d'exploitation.

Au final, le parallélisme au niveau des fils d'exécution possède le potentiel d'être beaucoup plus efficace que celui au niveau des instructions (ILP). Il implique cependant des ajouts importants au

niveau du matériel. En effet, le processeur doit se souvenir du contexte «matériel» d'exécution de plusieurs programmes, appelés fils d'exécution (ceux qui s'exécutent simultanément). Cela implique l'ajout :

- de compteurs ordinaux (PC) (un pour chaque fil) ;
- de registres d'état (un pour chaque fil) ;
- de registres pointeurs de piles (SP) (un pour chaque fil) ;
- d'ensembles de registres généraux (un pour chaque fil) ;
- d'ensembles de registres pour la mémoire virtuelle (un pour chaque fil)
- de TLBs (une pour chaque fil) ;
- de mécanismes de synchronisation (verrous) ;
- d'un mécanisme rapide de changement de contexte matériel ;
- etc.

Il existe au moins trois implantations distinctes pour le multi-fils matériel. Les deux premières décrites ici sont basées sur le multi-fils temporel [71] tandis que la troisième l'est sur le multi-fils simultané (SMT) [68].

1. le multi-fils finement intercalé (aussi appelé «Barrel multi-threading» [71, 58]) ;

   Selon cette première implantation, le matériel change de fil à chaque instruction ou à chaque cycle de l'horloge. Ce changement est généralement basé sur un algorithme de type «tourniquet» (round-robin) qui choisit des instructions de chacun des fils alternativement (un fil par cycle) afin de lancer leur exécution. Si un fil est bloqué en attente d'une demande en mémoire ou autre, il saute son tour. La figure 1.25 donne un exemple d'un tel fonctionnement. Ainsi au premier cycle (bleu) on exécute toutes les instructions possibles pour le fil 1. Au second cycle (jaune), on fait de même pour le fil 2, etc.

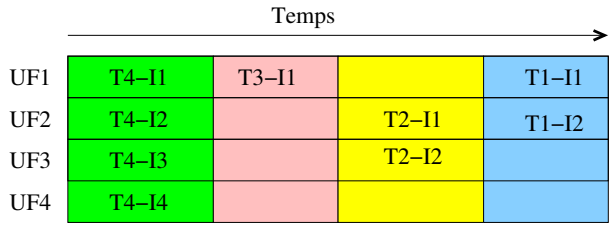


Figure 1.25 – Exécution optimale d'instructions avec super-scalaire et pipeline

   L'avantage d'un algorithme de type tourniquet est de cacher les latences courtes (branchement) et longues (`load/store`). Comme les instructions d'un fil particulier ne s'exécutent pas une à la suite de l'autre et qu'il y a un certain nombre de cycles entre les deux, le délai est suffisamment long pour que le matériel puisse charger les données de la mémoire ou résoudre la destination d'un branchement.

   Le multi-fils finement intercalé possède toutefois un certain nombre d'inconvénients induits par le tourniquet. Ainsi l'exécution de chaque fil se fait beaucoup plus lentement, car même si un fil est prêt à s'exécuter sans blocage, il sera tout de même ralenti par l'exécution des instructions des autres fils. Par exemple, lors du traitement de quatre fils simultanément,

l'exécution des fils sera ralenti d'un facteur de quatre (on exécute les instructions d'un fil particulier seulement 1 cycle sur 4).

Le processeur est aussi plus complexe car il doit être en mesure de changer de fil à chaque cycle, donc très rapidement et très souvent.

Le SPARC Niagara [24, 73, 41, 55, 56] est un processeur basé sur cette approche. La première version, l'UltraSparc T1, présentée en 2005, comportait quatre cœurs supportant chacun quatre fils, donc 16 cœurs virtuels, pouvant exécuter jusqu'à 16 programmes simultanément. La seconde édition, l'UltraSparc T2, lancé en 2007, comportait huit cœurs supportant chacun huit fils, donc 64 cœurs virtuels offrant la possibilité d'exécuter jusqu'à 64 programmes simultanément.

Ce type d'architecture est particulièrement utile et efficace dans des environnements qui requièrent peu de calculs, beaucoup d'entrées/sorties et où la vitesse de chaque transaction est moins importante que le nombre de transactions traitées (rendement) telles que les serveurs WEB, les serveurs de courrier électronique et les serveurs de base de données.

2. le multi-fils grossièrement intercalé ;

   Dans ce cas-ci, comme l'illustre la figure 1.26, le matériel traite simultanément deux fils d'exécution. De plus, il change de fil seulement lorsqu'un fil bloque sur des attentes longues (accès à la mémoire par des instructions de type `Load/Store`). Ainsi aux trois premiers cycles (bleu), le processeur exécute toutes les instructions possibles pour le fil 1 (T1). Au dernier cycle (vert), comme le fil 1 (T1) est bloqué, le processeur traite des instructions du second fil (T2).

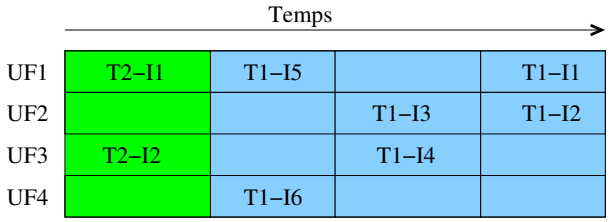


Figure 1.26 – Multi-fils grossièrement intercalé

Le multi-fils grossièrement intercalé présente l'avantage de ne pas exiger de changement de contexte à chaque cycle. Il n'est donc pas nécessaire d'implanter une unité ultra rapide de changement de contexte. De plus, un fil prêt à exécuter ne sera pas ralenti par le tourniquet obligatoire.

Son principal inconvénient réside dans le fait qu'elle ne cache pas entièrement les blocages courts (branchements). De plus, elle se limite généralement à deux fils d'exécution par processeur. Ce type de multi-fils n'est plus vraiment utilisé de nos jours.

3. le multi-fils simultané (Simultaneous Multi-Threading ou SMT [68]) ;

   Cette troisième approche est vraiment la plus complexe car elle exploite simultanément le ILP et le TLP. Elle recherche dans tous les fils, des instructions pour occuper au maximum toutes les unités fonctionnelles à chaque cycle du pipeline. Elle vise donc à lancer l'exécution

de plusieurs instructions provenant de fils différents à chaque cycle comme illustré par la figure 1.27. En comparaison, les deux premières approches lançaient l'exécution de plusieurs instructions du même fil à chaque cycle.

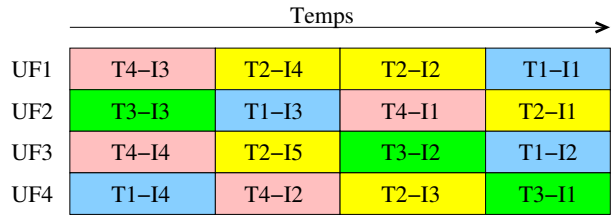


Figure 1.27 – Multi-fils simultané

Le SMT a le potentiel d'être la plus performante mais elle est aussi la plus complexe à implanter.

Les processeurs de la compagnie Intel, utilisés dans la plupart des ordinateurs, les processeurs ARM [18] et les processeurs de la série POWER d'IBM [65, 66] implantent le multi-fils simultané. Intel implante un SMT supportant deux fils matériel et qualifie ce procédé de «hyper-threading» [61, 62, 20, 28]. C'est pour cette raison que les statistiques sur le taux d'utilisation du processeur de votre ordinateur concernent le double du nombre de cœurs physiques, soit huit cœurs virtuels sur un processeur ayant physiquement seulement quatre cœurs. Le hyper-threading donne généralement un gain de 20% à 30% de performance.

1.6.3 Parallélisme au niveau des processus (multi-cœurs, multi-processeurs, multi-ordinateurs)

La plupart des ordinateurs modernes fournissent plusieurs unités de calcul relativement indépendantes que ce soit de multiples cœurs, processeurs ou même ordinateurs entiers (grappes de calcul). Clarifions d'abord quelques éléments de vocabulaire.

- Carte mère

  La carte mère d'un ordinateur est un matériel informatique (composé de circuits imprimés et de ports de connexion) servant à interconnecter toutes les composantes d'un micro-ordinateur. Autrement dit, tel qu'illustré à la figure 1.28, la carte mère est le support sur lequel tous les composants de votre ordinateur viennent se brancher tels qu'un disque dur (SSD ou classique), un processeur, une carte graphique, des barrettes de mémoire, etc. On peut ainsi l'assimiler à son système nerveux. La carte mère est installée à l'intérieur de l'ordinateur. La figure 1.29 présente diverses cartes mères disponibles sur le marché pour les stations de travail (a), les portables (b) et les cellulaires (c).

- Le processeur

  Le principal élément que l'on branche sur la carte mère est une puce, comme l'illustre la figure 1.30, que nous appellerons, pour les besoins du cours, le processeur. Il porte aussi le nom de «Unité Centrale de Traitement» (UCT ou CPU). Son rôle est de contrôler tous les éléments branchés sur la carte mère ainsi que de traiter et d'exécuter tous les programmes.

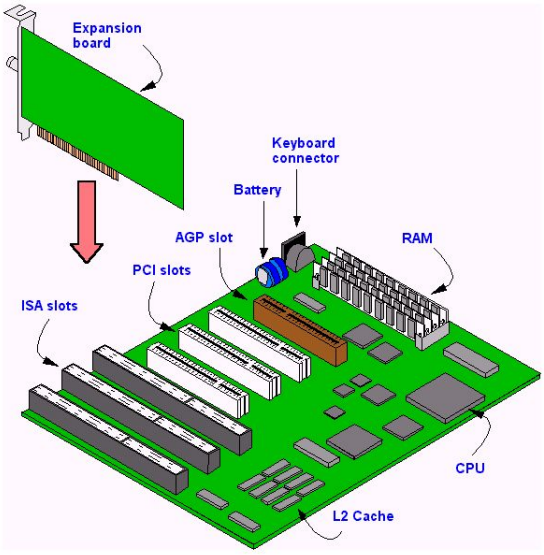


Figure 1.28 – Une carte mère est une pièce servant à interconnecter les composantes d'un ordinateur. Repris de «The Free Dictionary» [6]

(a) (b) (c)

Figure 1.29 – Exemples de cartes mères. Reprise de «Wikipedia» [76]

Figure 1.30 – Exemples de puces/processeurs. Reprise de «FuturaTech» [10]

Les systèmes multi-cœurs (Chip-level Multiprocessing - CMP)

Aujourd'hui, les ordinateurs sont équipés de puces (ou processeurs) qui contiennent plusieurs cœurs de calcul. Il existe deux types de processeurs multi-cœurs, les «généraux» et les «spécialisés».

1. Les multi-cœurs «généraux» ;

   Dans cette architecture, chaque cœur implante une unité de calcul complète capable d'exécuter des programmes différents à un rythme différent. La plupart des processeurs modernes possèdent plusieurs cœurs. Le tableau 1.31 présente le nombre de cœurs associés à plusieurs de ces processeurs. Notons que cette approche peut s'avérer moins rapide que l'approche «multi-processeurs». Ce fait est surtout dû aux grands nombres de ressources partagées entre les cœurs (bus, cache, ...).

   Pour plus d'informations, voir l'article sur le Threadripper 7000[25]

2. Les multi-cœurs «spécialisés» ;

   Le recours à cette architecture implique que tous les cœurs exécuteront simultanément la même fonction, ce qu'on appelle SIMD (Single Instruction Multiple Data). On retrouve cette organisation principalement dans les cartes graphiques (Nvidia et ATI/AMD). Comme les cœurs sont spécialisés et parfaitement synchronisés, on a la possibilité d'en intégrer un grand nombre (plus de 1 0000). La tableau 1.32 présente quelques exemples d'architecture multi-cœurs spécialisés.

Les multi-processeurs (System-Level multiprocessing - SMP)

Une architecture multi-processeurs implique la présence de plusieurs puces (processeurs) sur une même carte mère ou de plusieurs cartes mères dans le même ordinateur. Chacun de ces processeurs possèdent potentiellement plusieurs cœurs. Selon leur localisation, ils partagent le bus et la mémoire centrale. Chaque processeur possède donc son propre bus interne et sa propre mémoire cache, ce qui n'est pas le cas des multiples cœurs intégrés dans un unique processeur.

Marque	Modèle	Cœurs physiques	Coeurs virtuels (multi-fils)
ARM	Cortex	1-4	?
	Snapdragon (Qualcomm)	1-8	1-8
	M1 (Apple)	8	8
	Denver (Nvidia)	2	2
	Carmel (Nvidia)	2-8	2-8
	X-Gene (Applied Micro)	64	64
	Exynos (Samsung)	2-8	2-8
	ThunderX (Cavium)	8-96	32-384 ?
	Altra (Ampere)	80	80
Intel	I9	6-10	12-20
	I7	4-8	8-16
	I5	4-6	8-12
	Xeon	2-14	4-28
AMD	Ryzen Threadripper	8-96 *	16-192
	Ryzen	4-16	8-32
Sparc	T2-T4 (SUN)	8	64
	T3-T5 (SUN)	8	128
	UltraSparc (SUN)	2	32
	Sparc64 XII (Fujitsu)	8	96
	Sparc M8 (Oracle)	8	256
POWER	7 (IBM)	8	256
	8 (IBM)	6-12	48-96
	9 (IBM)	12-24	96-192

*Pour plus d'informations, voir l'article sur le Threadripper 7000 [25].

Figure 1.31 – Exemple de processeurs multi-cœurs et multi-fils

Marque	Modèle	Cœurs physiques
NVIDIA GeForce	RTX 3090	10496
	RTX 3080	8074
	RTX 2080 TI	4352
	RTX 2080 Super	3072
	RTX 2080	2944
	RTX 2070	2560
AMD Radeon	RX 6900	5120
	RX 6800 XT	4608
	RX 6800 XT	4608
	RX 6800	3840
	RX 5700	2560
	VII	3840

Figure 1.32 – Exemple de processeurs multi-cœurs

La différence entre multi-processeurs et multi-cœurs se situe dans le fait que ces derniers se retrouvent sur la même puce tandis que les premiers sur la même carte (différentes puces). La figure 1.33 présente une architecture supportant de multiples processeurs, de multiples cœurs et le multi-fils au niveau matériel. Par exemple, l'architecture POWER 7 d'IBM supporte jusqu'à quatre fils d'exécution par cœur, huit cœurs par processeur et 32 processeurs sur une carte mère (socket). Elle a donc la capacité d'exécuter jusqu'à 1024 programmes simultanément sur une seule machine.

L'architecture multi-processeurs possède ses avantages et ses inconvénients par rapport aux multi-cœurs. Le partage de ressources entre les cœurs rend la communication entre eux plus rapide. Néanmoins, ce même partage de ressources, principalement la cache, rend l'exécution de processus distincts plus lente sur un système multi-cœurs que sur un système multi-processeurs. En revanche, un multiprocesseur s'avère généralement plus fiable car un processeur qui tombe en panne n'affectera pas les autres. Le coût d'un système multi-processeurs est aussi plus élevé.

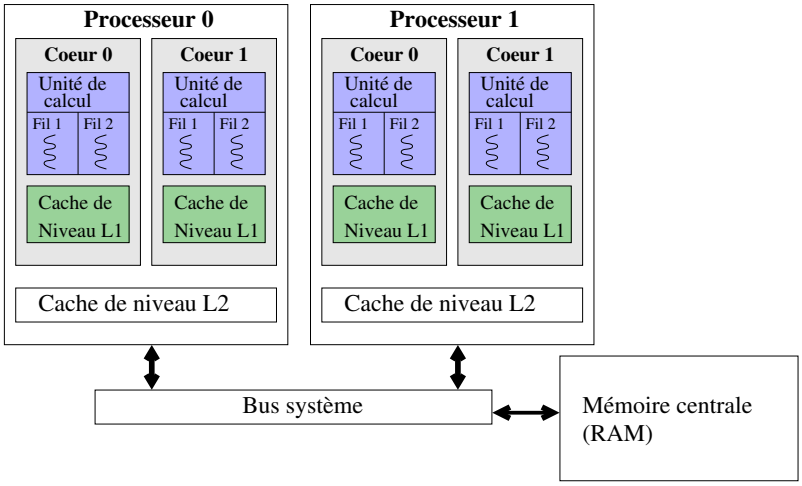


Figure 1.33 – Architecture supportant le multi-processeurs, le multi-cœurs et le multi-fils.

Les multi-ordinateurs

Cette architecture relie tout simplement entre eux plusieurs ordinateurs. Chaque ordinateur ayant ses propres ressources (mémoire, bus, ...), l'exécution de tâches indépendantes y est d'autant plus efficace. Toutefois la communication entre les ordinateurs est beaucoup plus lente que celle entre les processeurs sur une même carte mère. Les grappes de calcul (cluster) et les centres de données font un usage intensif de ce type d'architecture. Évidemment chacun des ordinateurs présents dans une grappe contient généralement plusieurs processeurs, qui à leur tour contiennent plusieurs cœurs, qui eux enfin supportent le multi-fils. Cette configuration procure une puissance de calcul impressionnante.

Ainsi une grappe théorique de 10 000 ordinateurs comprenant chacun huit processeurs de 16 cœurs chacun, exécutera en parallèle 1 280 000 programmes, ceci excluant le multi-fils qui ajoute des cœurs virtuels (mais souvent au dépend de la vitesse pure). Ainsi, la grappe Sunway TaihuLight,

première au top 500 de 2016, comprenait 10 649 600 cœurs. La grappe Tianhe-2A, seconde à ce même classement, ne comprenait que 4 981 760 cœurs.

La mémoire cache

Un composant important qui limite de façon significative la vitesse d'exécution d'un programme est la mémoire centrale. En effet, le temps requis pour accéder aux données et instructions localisées en mémoire centrale est significativement plus long que le temps requis pour traiter une instruction. L'accès à une donnée en mémoire centrale risque donc de bloquer l'exécution d'une instruction pour une période de temps significative. De même, l'accès à une instruction en mémoire centrale bloque la progression d'un processus.

Pour remédier à ce problème, les ordinateurs modernes utilisent de la mémoire cache. La mémoire cache est une mémoire rapide qui emmagasine temporairement les données provenant de la mémoire centrale. Elle est beaucoup plus rapide que cette dernière et affecte ainsi la performance des programmes séquentiels et parallèles. En effet, éviter les accès à la mémoire centrale offre un gain de performance significatif.

Toutefois le fait de lire et d'écrire des données directement dans la cache, génère parfois des incohérences au niveau des données. Celles-ci sont cependant peu fréquentes et contrôlées partiellement par le protocole de gestion de la cache [6]. Le programme 1.5 implante un programme qui démontre le problème d'incohérence de données dans la cache. Le résultats à la fin de l'exécution ne devrait jamais donner `r1 == 0` et `r2 == 0`, car cela impliquerait un non respect dans l'ordre d'exécution des instructions ou une incohérence au niveau de la cache. Le résultat présenté à la figure 1.34 montre bien que cette «impossibilité» se produit régulièrement. Pour régler le problème, il suffit, en C++, de définir les variables `X` et `Y` comme étant atomique (`atomic<int> X; atomic<int> Y;`).

La mémoire cache, dans plusieurs architectures, est séparée en deux parties : la cache d'instructions et la cache de données. Comme les instructions ne sont pas susceptibles d'être modifiées (constantes) mais que les données, elles, le sont, cette séparation (ou non) a des conséquences importantes quant au mode de gestion appliquée à chacune de ces deux parties. Ainsi la cache d'instructions, étant accessible en lecture seulement, n'exige aucun effort pour maintenir la cohérence. En revanche, les données pouvant être modifiées dynamiquement, la cache de données requiert l'usage d'algorithmes sophistiqués pour le maintien de la cohérence.

Comme la mémoire cache est coûteuse et que l'espace pour la placer sur la puce (ou sur la carte mère) est limité, plutôt que de fournir une seule mémoire cache (avec ses parties de données et de code), on en fournit plusieurs. Étant donné ses mémoires possèdent des caractéristiques différentes en terme de vitesse d'accès, de séparation données/instructions et de localisation (sur la puce ou sur le carte mère), elles sont classées en niveaux. Plusieurs architectures possèdent trois niveaux de cache :

1. La cache de niveau L1 ;

   La mémoire cache de ce niveau est intégrée sur la puce du processeur et est séparée en cache de données et d'instructions. Il existe généralement une cache de niveau L1 par cœur de calcul. Comme la mémoire cache de niveau L1 offre un temps d'accès extrêmement rapide mais est très coûteuse, sa taille est restreinte.

2. La cache de niveau L2 ;

6. Plusieurs protocoles de gestion de cache seront présentés ultérieurement.

```
#include <pthread.h>
#include <semaphore.h>
#include <iostream>
using namespace std;
//-------------------------------------
// Sémaphores pour synchroniser le début des fils
sem_t beginSema1;
sem_t beginSema2;
sem_t endSema;

int X, Y;  // Variables partagées, modifiées et copiées
int r1, r2; // Recoivent des copies de X et Y
//-------------------------------------------------------
void *fonction1(void *param){  // Fil 1
    srand (time(NULL));
    for (;;) {
        sem_wait(&beginSema1);        // Attend le signal du depart
        while ( rand() % 8 != 0) {}  // Attente aleatoire
        // On effectue les affectations
        X = 1; r1 = Y;
        sem_post(&endSema);  // Affectations completees
    }}
/-------------------------------------------------------------------
void *fonction2(void *param) { // Fil 2
    srand (time(NULL));
    for (;;) {
        sem_wait(&beginSema2);       // Attend le signal du depart
        while (rand() % 8 != 0) {}  // Attente aleatoire
        // On effectue les affectations
        Y = 1;  r2 = X;
        sem_post(&endSema);  // Affectations completees
    }}
//---------------------------------------------------------------------
// Démarrage des deux fils en parallèle
int main()  {
    // Initialise les semaphores
    sem_init(&beginSema1, 0, 0);
    sem_init(&beginSema2, 0, 0);
    sem_init(&endSema, 0, 0);
    // Demarre les fils
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, fonction1, NULL);
    pthread_create(&thread2, NULL, fonction2, NULL);
    // On repete a l'infinie
    int nb = 0;
    for (int i = 1; i < 10000000; i++)  {
        // Replace X et Y a 0
        X = 0;  Y = 0;
        // Signal du depart
        sem_post(&beginSema1);   // Débloque fil 1
        sem_post(&beginSema2);   // Débloque fil 2
        // Attend la fin des affectations
        sem_wait(&endSema);     // Attend résulta du fil 1
        sem_wait(&endSema);     // Attend résulta du fil 1
        // Verifie pour un resultat incoherent
        if (r1 == 0 && r2 == 0) { // Ce résultat est impossible normalement!!!
            nb++;
            cout<<nb<<" resultats erronnees apres "<<i<< " iterations"<<endl;
        } }
    return 0;
}
```

Programme 1.5 – Programme montrant les incohérences de la cache.

```
$ ./incoherence-cache
1 resultats erronnees apres 160869 iterations
2 resultats erronnees apres 176620 iterations
3 resultats erronnees apres 194356 iterations
4 resultats erronnees apres 415984 iterations
5 resultats erronnees apres 514700 iterations
6 resultats erronnees apres 515847 iterations
7 resultats erronnees apres 831586 iterations
8 resultats erronnees apres 883595 iterations
9 resultats erronnees apres 1055347 iterations
10 resultats erronnees apres 1080760 iterations
11 resultats erronnees apres 1109733 iterations
12 resultats erronnees apres 1624721 iterations
13 resultats erronnees apres 1720216 iterations
14 resultats erronnees apres 1953886 iterations
15 resultats erronnees apres 2077454 iterations
16 resultats erronnees apres 2224320 iterations
17 resultats erronnees apres 2243363 iterations
18 resultats erronnees apres 2266532 iterations
19 resultats erronnees apres 2270996 iterations
...
```

Figure 1.34 – Résultat de l'exécution du programme 1.5

La mémoire cache de niveau L2 présente un temps d'accès plus lent que celle de niveau L1 et, par conséquence, est moins coûteuse. La cache à ce niveau est donc de plus grande dimension. Malgré sa taille plus importante, elle demeure localisée directement sur la puce, mais, selon les implantations, elle est soit privée à chaque cœur, soit partagée par plusieurs cœurs. Elle n'est généralement pas séparée en cache de données et d'instructions.

3. La cache de niveau L3 ;

   La mémoire cache de ce niveau offre un temps d'accès encore plus lent que celle de niveau L2. Sa dimension est donc encore plus importante. Cette mémoire n'est généralement pas située sur la puce elle-même, mais sur la carte mère. Elle est alors partagée par tous les cœurs. Elle n'est pas, en général, séparée en cache de données et d'instructions.

La figure 1.35 présente un exemple de mémoire cache à trois niveaux dont seulement le 3^e est partagé par tous les cœurs, les deux premiers étant privés (accessible par un seul cœur). La figure 1.36 explicite une organisation dans laquelle seulement la cache de niveau L1 est privée. La cache de niveau L2 est partagée par des pairs de cœurs et la cache de niveau L3 est commune à tous.

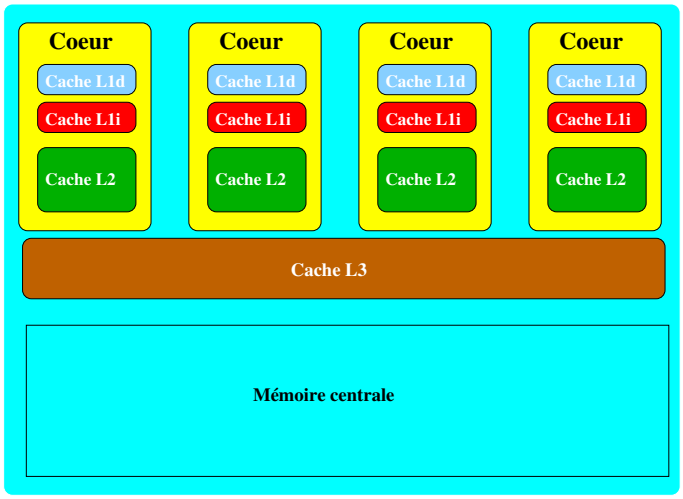


Figure 1.35 – Architecture comportant trois niveaux ce cache.

Pour plus d'informations sur la mémoire cache, voir l'annexe B.

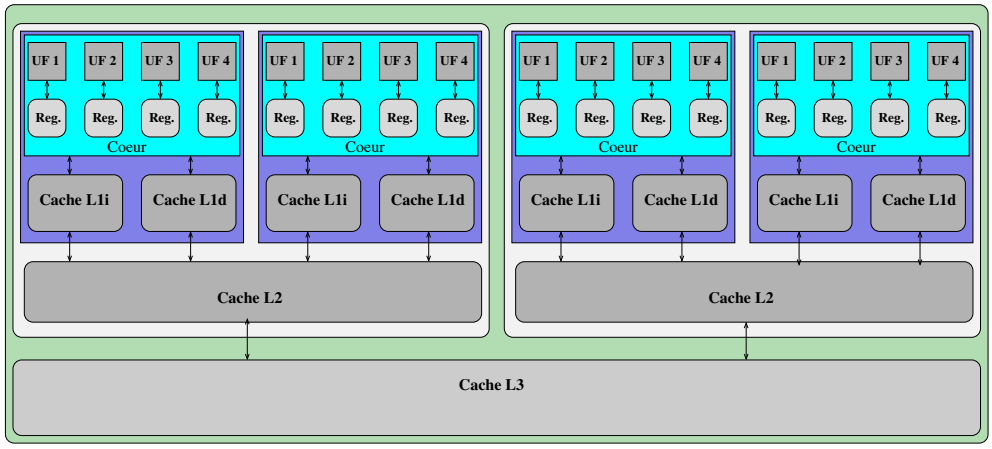


Figure 1.36 – Architecture comportant trois niveaux ce cache.

Annexe A

La réentrance

On dit d'une fonction qu'elle est ré-entrante si elle ne se modifie pas elle-même. En fait, la réentrance [53, 52, 11, 34, 33, 12, 5] est la propriété qui permet à une fonction d'être utilisable simultanément par plusieurs fils d'exécution ou processus. Ceci évite la duplication en mémoire d'un programme exécuté simultanément par plusieurs applications. Notons que ce type de fonction sert aussi au niveau des interruptions matérielles et de la récursivité.

Pour obtenir du code ré-entrant, les règles suivantes s'appliquent :

- Le code ré-entrant ne doit jamais contenir de données à même son code (intégré dans le code) ;

  On réfère ici aux données «statiques» ou «globales» qui se doivent d'être fournies par l'appelant ou remplacées par des variables locales. Ces dernières ne posent pas de problème car les piles les contenant, sont associées à chaque processus individuellement.

  La séparation des segments de code et de données faite par les compilateurs et les systèmes assure la réentrance. Les langages modernes fournissent tous cette séparation. De même, en langage d'assemblage, si le code et les données sont correctement identifiés par rapport à leur segment [1], le code sera ré-entrant.

- Du code ré-entrant ne doit pas se modifier lui-même ;

  Une fonction ré-entrante ne doit pas modifier son propre code. Elle doit donc être conçue en ce sens. De plus, le système peut renforcer cette exigence en chargeant le code dans un segment ou une page accessible en lecture seulement. La plupart des langages modernes empêchent les fonctions de modifier leur propre code.

- Du code ré-entrant ne doit jamais faire appel à une fonction non ré-entrante.

Le noyau d'un système d'exploitation comporte souvent des parties non ré-entrantes afin d'éviter des complications fâcheuses (incohérence de données critiques, perte de performances). L'écriture de modules destinés à une exécution dans l'espace noyau reste, pour cette raison, délicate.

La réentrance des fonctions n'est pas forcément garantie par tous les langages de programmation. Ainsi, une fonction écrite en Ada sera toujours ré-entrante alors qu'une fonction écrite en C ne l'est pas par défaut.

1. Les segments «`.text`» pour le code, «`.bss`» (block storage section) pour les données non-initialisées, «`.data`» pour les données initialisées et «`.rodata`» (read-only) pour les constantes.

La non réentrance d'une fonction est rarement un problème dans le cas de la programmation mono-tâche, mais s'avère souvent désastreuse en programmation concurrente.

Il ne faut pas confondre «réentrance» et «sûre pour le multi-fils» («*thread-safe*»). Pour rendre une fonction «sûre pour le multi-fils», la synchronisation lors de l'accès à certaines données est requise.

Sûre pour le multi-fils [72]

Une entité (une section de code (fonction) ou une structure de données) est dite «sûre pour le multi-fils» s'il est possible pour plusieurs fils de l'accéder simultanément sans nuire à leur bon fontionnement et si elle n'empêche pas ceux-ci de satisfaire leur spécification à cause d'interactions inappropriées.
Ainsi une fonction ou une structure de données est dite sûre pour le multi-fils si elle garantie qu'il n'y aura aucune condition de course lorsqu'accédée par plusieurs fils simultanément.
Il existe plusieurs approches pour garantir cette propriété. L'une d'elles consiste à ne pas partager d'information par le biais de variables locales, d'objets immuables et de fonctions réentrantes. La seconde approche consiste se servir de la synchronisation. Cette dernière approche est utile lorsque le partage d'information est inévitable. Dans cette situation, le recours à l'exclusion mutuelle et aux opérations atomiques est requise.

A.1 Exemple 1 : fonction non ré-entrante, ni «sûre pour le multi-fils»

Le programme A.1 implante une fonction non ré-entrante et non «sûre pour le multi-fils», puisqu'elle emploie une variable globale et qu'elle ne synchronise pas les accès. Ainsi, si une interruption se produit après la ligne 6, la variable «`tmp`» risque d'être corrompue. Étudions la séquence illustrée à la figure A.1. On remarque qu'à la fin de l'exécution, la variable «`y`» du processus P1 ne contient plus la valeur attendue.

```
1  int tmp;
2  void swap(int* x, int* y)
3  {
4      tmp = *x;
5      *x = *y;  /* Interruption après cette ligne */
6      *y = tmp;
7  }
```

Programme A.1 – Fonction non ré-entrante et non «sûre pour le multi-fils»

A.2 Exemple 2 : fonction ré-entrante et «sûre pour le multi-fils»

Le programme A.2 présente une fonction ré-entrante et sûre pour le multi-fils, puisqu'elle recourt seulement à des variables locales. Peu importe la séquence d'exécution des divers processus, la

Processus	ligne	x	y	tmp
P1	4	1	2	1
P1	5	2	2	1
P2	4	4	5	4
P2	5	5	5	4
P2	6	5	4	4
P1	6	2	4	4

Figure A.1 – Ressources partagées ou non entre les fils d'exécution

fonction retournera toujours le résultat escompté. La séquence de la figure A.2 est la même que celle de l'exemple précédant. Dû à la présence d'une variable «`tmp`» distincte pour chaque processus, le résultat sera toujours valide.

```
1  void swap(int* x, int* y)
2  {
3      int tmp;
4      tmp = *x;
5      *x = *y;
6      *y = tmp;
7  }
```

Programme A.2 – Fonction ré-entrante et «sûre pour le multi-fils»

Processus	ligne	x	y	tmp (P1)	tmp (P2)
P1	4	1	2	1	
P1	5	2	2	1	
P2	4	4	5		4
P2	5	5	5		4
P2	6	5	4		4
P1	6	2	1	1	

Figure A.2 – Ressources partagées ou non entre les fils d'exécution

A.3 Exemple 3 : fonction ré-entrante et non «sûre pour le multi-fils».

Il est possible qu'une fonction soit ré-entrante mais non «sûre pour le multi-fils». Le programme A.3, extrait de Wikipedia [52], illustre cette situation. La fonction est ré-entrante mais non «sûre pour le multi-fils» car, d'une part, la synchronisation n'est pas assurée et, d'autre part, la variable globale est sujette à des modifications par plusieurs processus simultanément. Une discussion sur

Stackoverflow [11] met en évidence la réentrance de la fonction si toutefois, la définition de la réentrance admet qu'une fonction termine toujours avant l'appel suivant. Dans ce cas spécifique, la réentrance garantie que chacune des exécutions «complètes» de la fonction ne produise aucun effet de bord. Si la définition de la réentrance n'inclue pas «l'entièreté» de l'exécution de chaque instance de la fonction, cette dernière n'est pas ré-entrante puisqu'une interruption pourrait se produire pendant son exécution.

```
1  int tmp;
2
3  void swap(int* x, int* y)
4  {
5      int s;
6      s = tmp;
7      tmp = *x;
8      *x = *y;
9      *y = tmp;
10     tmp = s;
11 }
```

Programme A.3 – Fonction ré-entrante et non «sûre pour le multi-fils»

A.4 Exemple 4 : fonction non ré-entrante et sûre pour le multi-fils

Le programme A.4 implante une fonction non ré-entrante mais «sûre pour le multi-fils». C'est la présence d'une variable «`tmp` globale» mais «locale» à chaque fil d'exécution qui permet à la fonction de devenir «sûre pour le multi-fils». Ainsi, étant donné qu'il n'y a aucun partage de variable entre deux fils d'exécution différents, la séquence d'exécution sera la même que celle du tableau A.2.

La réentrance n'est toutefois pas garantie comme le démontre la séquence d'exécution de la figure A.3. Supposons une fonction (exécutée par le processus P1) interrompue par un événement matériel. Si la routine de traitement de cette interruption (INT) fait appel à cette même fonction, alors le processus P1 et la routine d'interruption (INT) partageront la même variable `tmp`. Notons qu'alors le risque de corruption est présent.

```
1  _Thread_local int tmp;
2
3  void swap(int* x, int* y)
4  {
5      tmp = *x;
6      *x = *y;    /* Interruption se produit à cette étape */
7      *y = tmp;
8  }
```

Programme A.4 – Fonction non ré-entrante et «sûre pour le multi-fils»

Processus	ligne	x	y	tmp
P1	5	1	2	1
P1	6	2	2	1
INT	5	4	5	4
INT	6	5	5	4
INT	7	5	4	4
P1	7	2	4	4

Figure A.3 – Ressources partagées ou non entre les fils d'exécution

Annexe B

La mémoire cache

Pour augmenter la performance des systèmes en général et des systèmes multi-processeurs en particulier, on utilise de la mémoire cache. On associe donc à chaque processeur ou cœur sa propre mémoire cache.

Rappelons qu'une mémoire cache est une mémoire beaucoup plus rapide mais aussi beaucoup plus petite que la mémoire centrale.

Plusieurs facteurs sont à considérer lors de l'utilisation d'une mémoire cache. Le premier est la gestion de la mémoire elle-même, soit le remplacement des entrées dans la cache lorsqu'elle est pleine. Puis, dû au fait que plusieurs mémoires cache implique potentiellement plusieurs copies de la même donnée, il faut également gérer le problème de cohérence des données emmagasinées dans les multiples mémoires cache. Finalement, lors du développement de programmes parallèles, il est parfois judicieux d'optimiser les accès aux données en fonction de la mémoire cache de l'ordinateur. En effet, dans certaines situations, des gains significatifs en temps de calcul sont faits en s'ajustant au fonctionnement de la mémoire cache.

B.1 Rappel de l'architecture d'un ordinateur

Un ordinateur est constitué des composants suivants :

- Le processeur (UCT, ou CPU)

  Le processeur est le cerveau de l'ordinateur. Il se divise en cœurs qui eux, se subdivisent en unités fonctionnelles, chacune pouvant exécuter une instruction d'un certain type. Ainsi, il existe des unités fonctionnelles pour traiter les instructions sur les nombres entiers, les instructions sur les nombres en virgule flottante, les instructions de chargement et d'emmagasinage des données, et les instructions de branchement.

  Le processeur est cadencé par une horloge qui, soumise à un courant électrique, envoie des impulsions. La fréquence d'horloge, appelée également cycle, correspond au nombre d'impulsions par seconde et s'exprime en Hertz (Hz). Ainsi, un ordinateur à 200 MHz possède une horloge envoyant 200 000 000 impulsions par seconde. À chaque impulsion de l'horloge le processeur exécute une action qui peut être une instruction ou une partie d'instruction (si le processeur possède un pipeline).

La puissance d'un processeur s'exprime souvent par le nombre d'instructions qu'il peut exécuter par seconde. Cette puissance est parfois caractérisé par les unités MIPS (Millions d'instructions par seconde) et BIPS (Milliards d'instructions par seconde). Même si ces mesures s'avèrent de très mauvais indicateurs de la vitesse d'un ordinateur, elles sont encore en usage.

Le processeur possède aussi une petite mémoire locale, appelée registre, qui sert à emmagasiner temporairement les données provenant de la mémoire.

- La mémoire centrale (ou mémoire vive)

  La mémoire centrale est une mémoire de très grande taille dans laquelle on charge le code des programmes et ses données en vue de leur exécution par le processeur. Le problème de la mémoire centrale est sa vitesse d'accès qui, proportionnellement à la vitesse de l'ordinateur, est excessivement lente. Les accès en mémoire centrale représentent donc un goulot d'étranglement pour le processeur. Une hiérarchie de mémoires, composées des registres et de divers niveaux de cache, est nécessaire pour réduire le coût des accès à la mémoire centrale.

- La mémoire cache

  La mémoire cache est une mémoire à accès rapide qui emmagasine temporairement les données provenant de la mémoire centrale afin d'accélérer le temps d'accès aux données par le processeur. La mémoire cache se distingue des registres par sa vitesse d'accès (plus lente) et sa taille (plus grande). Elle se distingue de la mémoire centrale aussi par sa vitesse d'accès (plus rapide) et sa taille (plus petite).

La mémoire cache est l'élément le plus susceptible d'affecter la performance d'un programme parallèle. Nous allons donc en discuter en détails afin de déterminer les adaptations possibles d'un programme parallèle à la mémoire cache.

Même si notre intérêt ici ne concerne que la mémoire cache matérielle, située entre la mémoire centrale de l'ordinateur et le processeur, il est important de noter que des mécanismes de mémoires cache peuvent être introduits entre tous les producteurs et les consommateurs de données fonctionnant de façon asynchrone. Par exemple, de la mémoire cache est présente entre un disque dur et la mémoire centrale ainsi qu'entre le réseau et l'espace applicatif. Le concept de mémoire cache existe aussi au niveau logiciel. Ainsi, sur un système de fichiers, la mémoire centrale sert de cache pour le fichier normalement situé sur le disque. De même, sur un système de fichier réseau, une cache logicielle accélère les accès aux contenus des fichiers distants.

B.2 Mémoire cache : définition

La mémoire cache (également appelée antémémoire) est une mémoire rapide permettant de réduire les délais d'attente lors d'accès à des informations normalement situées en mémoire centrale (ou sur un autre composant de mémoire plus lent). Elle enregistre temporairement, lors d'un premier accès, des copies de données provenant de la mémoire centrale afin de diminuer le temps d'un nouvel accès (en lecture ou en écriture) à ces données. Dans le cas d'écritures, afin d'avoir des informations à jour et éviter les incohérences, il faut que les données soient éventuellement recopiées en mémoire centrale. Pour cela, il existe plusieurs protocoles dont les principaux sont : write-through, write-back et write-around.

La mémoire cache, étant souvent significativement plus rapide et aussi énormément plus coûteuse, est conséquemment d'une dimension beaucoup plus restreinte que la mémoire à laquelle elle sert d'intermédiaire.

Techniquement, il est avantageux de gérer séparément les données non modifiables (code) de celles modifiables (données). Les processeurs ont fréquemment des caches distinctes pour emmagasiner le code et les données.

Le ordinateurs récents possèdent plusieurs niveaux de cache :

1. le premier niveau (L1) ;

   La mémoire cache du premier niveau est intégrée à la puce du processeur et se subdivise en cache de données (L1d) et cache d'instructions (L1i). Elle est très rapide ($\simeq$ 1 ns) mais de taille limitée (32KB à 96KB).

2. le second niveau (L2) ;

   À ce niveau, la mémoire cache se situe généralement sur la même puce que le processeur [1]. Elle est de plus grande taille (256KB à 2 MB) mais moins rapide ($\simeq$ 3 ns) que la cache de niveau L1. Selon les architectures, elle est soit privée aux cœurs, soit partagée entre eux. Elle n'est pas séparée en cache de données et cache d'instructions.

3. le troisième niveau (L3) ;

   La mémoire cache de ce niveau est habituellement située sur la même carte mère que le processeur et rarement sur la même puce. Elle est plus grande (4MB à 8MB) et plus lente ($\simeq$ 12 ns) que la cache L2. Sa vitesse est généralement de l'ordre de 2 fois plus rapide que la vitesse d'accès à la mémoire centrale. Elle n'est pas séparée en cache de données et d'instructions.

La mémoire cache est d'autant plus utile dès lors que l'algorithme à exécuter demande des accès répétitifs à de petites zones de la mémoire (comme c'est le cas dans une boucle). De plus, si le processeur est en mesure de prédire ses besoins futurs, il peut alimenter à l'avance le cache, ce qu'on appelle «préchargement» (*prefetch*).

B.3 Accès

Lorsque le processeur désire lire ou écrire une donnée en mémoire, il vérifie d'abord sa disponibilité dans la cache. Si elle est disponible la lecture ou l'écriture se fait immédiatement dans la cache. Sinon, elle y est copiée pour des accès ultérieurs.

Les deux principes permettant de garantir l'efficacité de la mémoire cache sont ceux de la localité spatiale ou temporelle.

1. Le principe de localité spatiale indique qu'un accès à une donnée située à une certaine adresse sera probablement suivi par l'accès à une autre donnée située non loin de cette même adresse. C'est le comportement typique d'une exécution séquentielle d'un programme ou de l'exécution d'une boucle.

2. Le principe de localité temporelle indique qu'un accès à une zone de mémoire à un instant donné se reproduira inévitablement dans la suite immédiate de l'exécution programme. C'est le cas par exemple des boucles.

1. Elle peut parfois être sur une puce séparée.

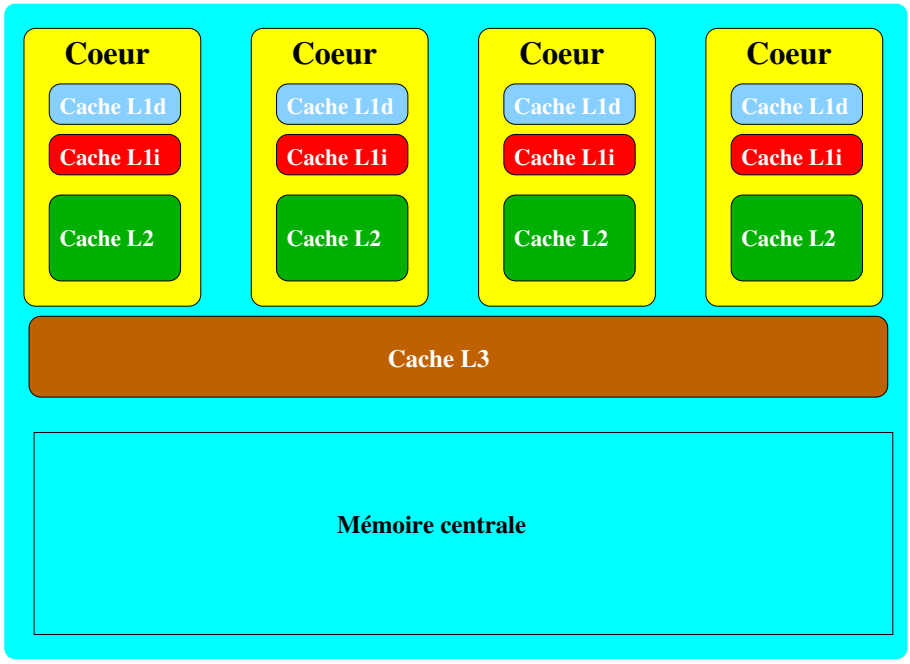


Figure B.1 – Structure de la mémoire cache

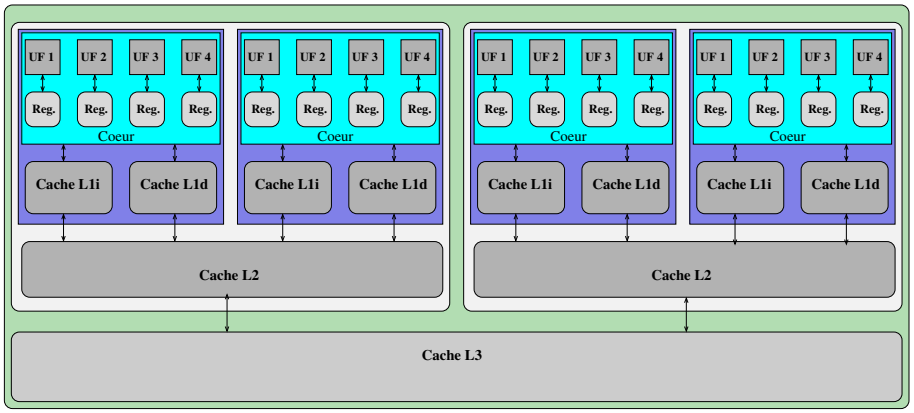


Figure B.2 – Structure de la mémoire cache

Certains types de calcul dans des programmes ne respectent pas ces principes. Ainsi, un programme effectuant un calcul matriciel et qui accéderait aux éléments de la matrice par colonnes, irait à l'encontre de localité spatiale.

B.4 Principe de fonctionnement

La mémoire cache est décomposée en lignes et en mots. Les transferts de données entre la mémoire centrale et la cache se fait par blocs de taille fixe appelées lignes. Au moment où une ligne est copiée de la mémoire centrale vers la cache, une entrée, contenant les données ainsi que sa localisation (tag), est créée. Lorsque le processeur accède la mémoire, l'adresse sert d'indice pour la recherche dans les lignes de la cache.

Deux concepts importants déterminent l'efficacité de la mémoire cache : la technique pour rechercher une entrée et la politique de remplacement de lignes lorsque la cache est pleine.

B.4.1 Algorithme de recherche

Correspondance directe ou pré-établie (Direct-mapped)

Selon ce mode de correspondance, chaque ligne de la mémoire centrale ne peut être enregistrée qu'à une seule adresse de la mémoire cache. Par exemple, son adresse dans la cache peut être associée au modulo de son adresse (ligne de la cache = adresse en mémoire % Nombre de lignes dans la cache).

Selon cette approche, illustrée par la figure B.3, une ligne de la cache est associée à plusieurs adresses en mémoire, ce qui provoque plusieurs collisions et rend cette approche peu efficace.

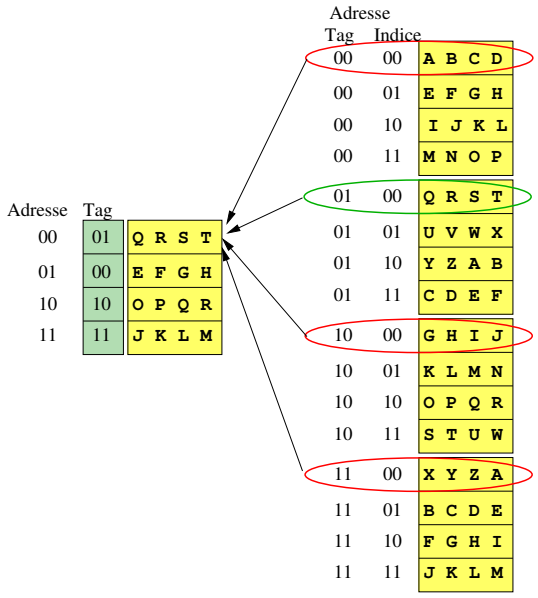


Figure B.3 – Correspondance directe

B.4.2 Entièrement associative

Selon ce mode de correspondance, chacune des lignes de la mémoire peut être associée à n'importe laquelle des adresses dans la mémoire cache. Cette méthode requiert cependant un circuit logique plus complexe car les possibilités de placements sont nombreuses (et aléatoires) ce qui nécessite que l'adresse recherchée soit comparée à toutes les entrées de la cache à chaque accès. Cette méthode s'avère trop complexe et n'est utilisée que pour des mémoires cache de très petite taille.

La figure B.4 montre comment les adresses peuvent être associées aux lignes de la mémoire cache.

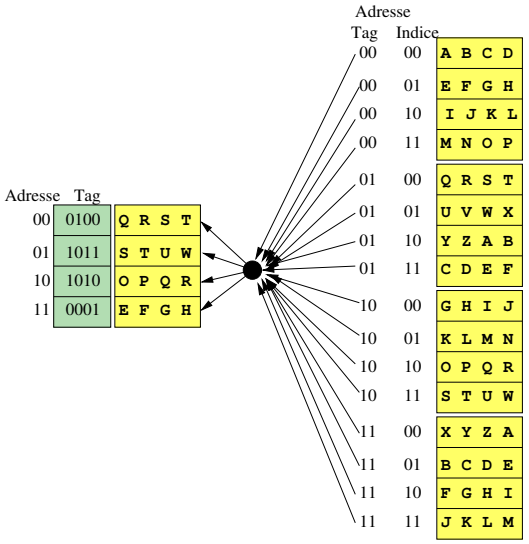


Figure B.4 – Correspondance associative

B.4.3 N-voies associatives

Cette méthode constitue un intermédiaire entre la correspondance directe et entièrement associative. La mémoire cache est divisée en N ensembles de M lignes. Une ligne de la mémoire est alors associée à une seule et unique ligne (direct) mais à l'intérieur de n'importe lequel des ensembles (entièrement associatif). Cette façon de faire permet d'éviter de nombreux défauts de cache conflictuels. À l'intérieur d'un ensemble, on utilise la correspondance directe, alors que, l'on utilise l'associativité pour la correspondance aux ensembles. En général, la sélection de l'ensemble est effectuée par l'équation : «$Ensemble = Adresse\ mémoire\ \%\ Nombre\ d'ensembles$». L'association peut aussi se faire à l'inverse, i.e. association pleine aux ensembles et directe aux lignes de l'ensemble. Les figures B.5 et B.6 réfèrent à ces deux types d'association pour $N = 2$.

B.5 Remplacement

Selon le type d'association privilégié, il faut pouvoir remplacer une page lorsque la cache est pleine. Plusieurs algorithmes existent pour traiter cette situation. Ils sont très similaires à ceux

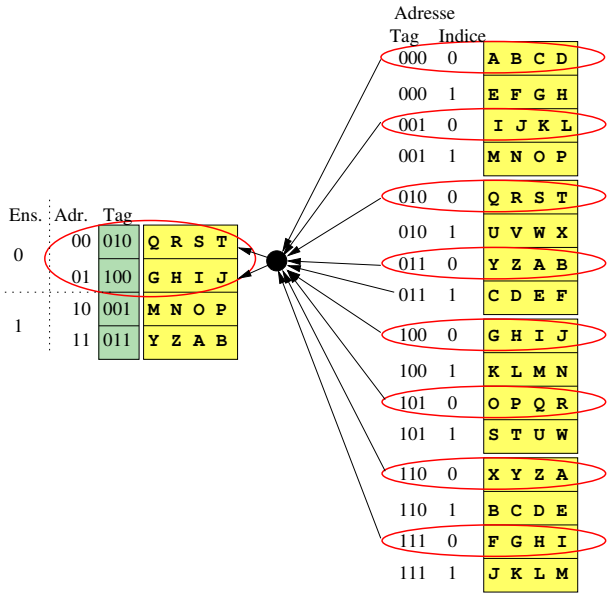


Figure B.5 – Structure de la mémoire cache

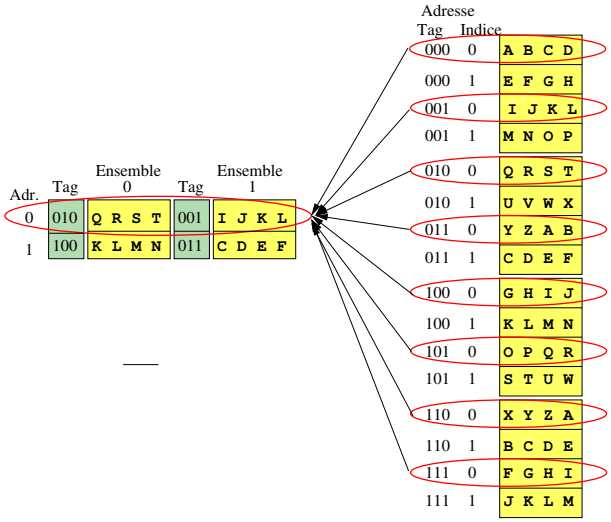


Figure B.6 – Structure de la mémoire cache

utilisés pour gérer la mémoire virtuelle.

Les mémoires caches de type N voies associatives et complètement associatives impliquent une correspondance entre différentes lignes de la mémoire centrale et celles du même ensemble dans la cache. Ainsi, lorsque l'ensemble de lignes de la mémoire cache, correspondant à l'endroit où une ligne de la mémoire centrale doit être chargée, est rempli, il faut désigner la ligne à effacer au profit cette nouvelle ligne. Le but de l'algorithme de remplacement des lignes de cache est de choisir cette ligne de manière optimale. Ces algorithmes doivent être implantés par le matériel afin d'être extrêmement rapides et ne pas ralentir le processeur.

La majorité des algorithmes reposent sur le principe de localité pour tenter de prévoir le futur à partir du passé. Parmi les algorithmes de remplacement des lignes de mémoire cache les plus répandus, citons :

- les choix aléatoires pour la simplicité de la création de l'algorithme ;
- FIFO (First In, First Out) pour sa simplicité de conception ;
- LRU (Least Recently Used) qui mémorise la liste des derniers éléments accédés.

B.6 Politique d'écriture dans la mémoire de niveau supérieur

Lorsqu'une donnée se situe dans le cache, le système possède au moins deux copies de cette donnée : l'une dans la mémoire centrale et l'autre dans la mémoire cache. Si la donnée est modifiée dans la cache, plusieurs politiques de propagation en mémoire centrale existent :

- écriture immédiate (write-through) ;

  Selon cette politique, toutes les données écrites dans la cache sont immédiatement recopiées en mémoire centrale. Cette dernière et la mémoire cache ont à tout moment une valeur identique, simplifiant ainsi de nombreux protocoles de cohérence, comme nous le verrons plus tard. Toutefois, ce procédé augmente de façon significative le trafic sur le bus.

- écriture différée (write-back) ;

  Selon cette politique, l'information n'est écrite dans la mémoire centrale que lorsque la ligne disparaît de la cache (invalidée par d'autres processeurs, évincée pour écrire une autre ligne...). Cette technique est la plus répandue lorsqu'une haute performance est requise car elle permet d'éviter de nombreuses écritures inutiles en mémoire centrale. Pour ne pas avoir à réécrire des informations qui n'ont pas été modifiées (et ainsi éviter d'encombrer inutilement le bus), chaque ligne de la mémoire cache est pourvue d'un bit indiquant la modification (bit dirty). Lorsque la ligne est modifiée dans le cache, ce bit est positionné à 1, indiquant ainsi qu'il faudra éventuellement réécrire la donnée dans la mémoire principale. L'écriture différée nécessite bien entendu des précautions particulières lorsqu'on l'utilise.

B.7 Cohérence des données en mémoire

Dans les situations où plusieurs processeurs possèdent sa propre cache locale, il est fort possible qu'il existe plusieurs copies (en mémoire centrale et en cache) de la même donnée. Vu cette possibilité

(plusieurs copies d'une donnée à des endroits variés), lorsque la donnée est modifiée, il faut que ce changement soit connu partout (mémoire centrale et les autres caches).On parle alors de cohérence. La cohérence au niveau de la mémoire cache définie le comportement des lectures et des écritures visant une seule adresse.

Pour conserver la cohérence de la cache, il est nécessaire de :

- propager les écritures ;

  Tout changement aux données dans n'importe quelle mémoire cache doit être propagé à toutes les copies.

- sérialisation de la transaction ;

  Les lectures/écritures à une même adresse mémoire doivent être vues dans le même ordre par tous les processeurs.

Théoriquement, la cohérence doit être maintenue pour chaque instruction de type «load/store», mais en pratique cela se fait au niveau des lignes de la cache.

Une mémoire est dite cohérente si les conditions suivantes sont satisfaites :

1. Une lecture $L_P(x)$ à l'adresse x par un processeur P qui suit une écriture $E_P(x)$ par P à la même adresse x, sans aucune autre écriture sur x par aucun autre processeur entre cette lecture et cette écriture, doit toujours retourner la valeur écrite par $E_P(x)$.

2. Une lecture $L_P(x)$ par une processeur P qui suit une écriture $E_Q(x)$ par un autre processeur Q retourne la valeur écrite par Q si les deux opérations sont «suffisamment» décalées dans le temps et si aucune autre écriture sur x ne se produit entre ces deux opérations.

3. Toutes les écritures à la même adresse doivent être sérialisées, i.e. deux écritures à la même adresse par des processeurs distincts sont vues dans le même ordre par tous les processeurs. Ceci assure qu'une lecture ne peut voir une ancienne valeur.

La première propriété préserve l'ordre du programme, la seconde définie ce que signifie «avoir une vue cohérente de la mémoire» et la troisième, assure que toutes les écritures sont vues dans le bon ordre par tous.

La clé pour implanter un protocole garantissant la cohérence de cache est de suivre l'état de chaque bloc de données. Il existe deux approches, celle basée sur un répertoire et celle dite «de cohérence par espionnage» :

- l'approche basée sur un répertoire ;

  Selon cette approche, l'état de chaque bloc de mémoire physique (ligne) (localisation, nombre de copies, ...) est conservé en un seul endroit, appelé le répertoire. Ce répertoire pourrait être distribué pour des raisons de mise à l'échelle. La communication avec le répertoire se fait par des demandes «point-à-point» sur le réseau de communication.

  Un protocole de cohérence par répertoire (directory-based) emmagasine les informations d'emplacement et d'état pour chaque portion de mémoire de la taille d'une ligne de cache. Il existe plusieurs moyens pour répartir ces données, soit en utilisant une unique mémoire, soit en distribuant celle-ci, soit enfin en utilisant une organisation hiérarchique, chaque nœud possédant les informations sur les lignes de cache présentes chez ses fils.

L'approche par répertoire tend à imposer une longue latence. En effet, chaque opération nécessite trois messages. Toutefois, elle utilise beaucoup moins de bande passante que l'approche par espionnage puisque les messages sont «point à point» et non diffusés. Pour cette raison, les systèmes ayant plusieurs processeurs (plus de 64) préfèrent utiliser cette approche.

- Cohérence par espionnage (snoop-based)

  Chaque mémoire cache possédant une copie d'un bloc de données possède aussi une copie de l'état du bloc contenant toutes les informations pertinentes sur la situation de partage du bloc. Toutefois contrairement à l'approche par répertoire aucun état centralisé n'est conservé. Cette approche nécessite l'usage d'un médium de communication fonctionnant par diffusion (bus ou *switch*). Chaque processeur surveille le médium de communication afin de déterminer s'ils ont une copie des données (lignes) accédées. Il invalide la ligne correspondante dans sa cache lors d'une écriture par un autre processeur. Il est nécessaire que toutes les demandes soient diffusées sur le médium en permanence. Cela surcharge rapidement le médium de communication et fonctionne correctement seulement pour des multiprocesseurs de petites tailles (moins de 64 processeurs). De façon générale, l'approche par espionnage est généralement plus rapide que l'approche basée sur un répertoire, si la bande passante est disponible puisque toutes les transactions sont des requêtes/réponses vues par tous les processeurs. L'inconvénient de cette approche est qu'elle ne supporte pas la mise à l'échelle. Chaque demande doit être une diffusion à tous les processeurs du système, ce qui signifie que si le nombre de processeurs dans le système augmente, alors la taille du bus et la bande passante doivent aussi augmenter.

Deux approches sont applicables pour maintenir la cohérence des informations et ce, quel que soit l'approche choisie, par répertoire ou espionnage :

- l'approche par invalidation

  Pour assurer le cohérence des données, cette approche vérifie d'abord l'existence d'un accès exclusif à l'information par le processeur, avant toute modification. Ce protocole, appelé « *écriture avec invalidation*», invalide toutes les copies en cache lorsqu'une écriture est observée. C'est le protocole le plus courant pour l'approche par espionnage autant que pour celle par répertoire. Ce procédé assure l'absence de toute copie lisible ou modifiable en cache. Ainsi, toutes les lectures subséquentes seront tenues de se rendre en mémoire centrale pour obtenir l'information.

  Par exemple, lorsqu'on utilise l'approche par espionnage, une mémoire cache invalidera sa copie dès qu'une écriture sera observée sur une donnée qu'elle possède.

  Une écriture par invalidation implique que le processeur passe par le bus pour effectuer l'invalidation. Pour se faire, il doit d'abord obtenir l'accès au bus pour ensuite diffuser l'adresse à invalider sur le bus. Comme tous les processeurs espionnent le bus en permanence, ils reçoivent ce message et vérifient si l'adresse est dans leur cache. Si elle s'y trouve, il l'invalide. Si deux processeurs tentent d'écrire à la même adresse en même temps, leur tentative sera sérialisée par le système de réservation du bus.

- Diffusion d'écriture

  L'alternative à l'invalidation est la diffusion de l'écriture ou le recourt à un mécanisme de mise à jour. Dans les deux cas, toutes les copies en cache sont mises à jour simultanément. Cette approche exige davantage de bande passante. De plus, lorsque de multiples mises à jour se

produisent à la même adresse, plusieurs d'entre elles se révèlent inutiles. Toutefois la latence est moins grande en écriture et en lecture.

Le traitement de la lecture elle-même détermine aussi la cohérence. En effet, lors d'une lecture, il est nécessaire de localiser la copie la plus récente de la donnée. Si elle se situe dans la cache, la donnée est retournée immédiatement, sinon il faut la localiser. Si la cache utilise le système d'écriture immédiate, la donnée la plus récente réside alors en mémoire centrale. Si le système utilise l'écriture différée, la valeur la plus récente peut se situer dans une mémoire cache plutôt qu'en mémoire centrale. L'espionnage sert alors à retracer la donnée. Comme tous les processeurs espionnent les adresses envoyées sur le bus, si l'un d'entre eux détient une copie à jour (modifiée - copie sale) de la donnée en cache, il la fournit (la ligne de données) en réponse à la demande de lecture et provoque l'annulation de cette dernière en mémoire centrale.

B.7.1 Exemple de protocoles

De multiples protocoles permettent d'assurer la cohérence de la mémoire cache[59, 40, 39, 44, 48, 43, 50, 49, 38, 29, 35, 37]. Les plus populaires sont basés sur l'espionnage avec écriture différée et par invalidation tels que MSI, MESI, MESIF (Intel), MOESI (AMD), MERSIF. Certains sont plutôt basés sur l'espionnage avec diffusion d'écriture comme le protocole Dragon[59]. Le protocole AMBA 4 ACE de ARM[38] supportent quant à lui l'approche par espionnage et celle par répertoire.

Dans cette section nous étudions le fonctionnement du protocole MSI basé sur l'espionnage avec écriture différée et par invalidation. Les protocoles MESI, MOSI. MOSIF et MOESI, pour ne citer qu'eux, sont des améliorations du protocole MSI.

Le procotole MSI

À venir...

Bibliographie

[1] Ted Baker : Operating systems : Notes - threads. https://www.cs.fsu.edu/~baker/opsys/notes/threads.html, 2015.

[2] John Bell : Courses notes : Operating systems - threads. https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/, 2013.

[3] Blackberry : Qnx. https://blackberry.qnx.com/en, 2021.

[4] Kristi Castro : Difference between process and thread. https://www.tutorialspoint.com/difference-between-process-and-thread, 2020.

[5] Chemeng : Writing reentrant and thread-safe code. https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/aixprggd/genprogc/writing_reentrant_thread_safe_code.htm, 2018.

[6] The Free Dictionnary : motherboard. https://encyclopedia2.thefreedictionary.com/motherboard, 2019.

[7] Itai Dinur, Danny Hendler et Iakobashvili Robert : Syllabus de cours - processes abd scheduling. https://www.cs.bgu.ac.il/~os182/wiki.files/Processes_18.pdf, 2018.

[8] Alexander Fox : 7 of the best terminal emulators for windows 10. https://www.maketecheasier.com/best-terminal-emulators-for-windows/, 2020.

[9] Fuchsia : Zircon. https://fuchsia.dev/fuchsia-src/concepts/kernel, 2022.

[10] FuturaTech : Cpu : qu'est-ce que c'est ? https://www.futura-sciences.com/tech/definitions/informatique-cpu-5741/, 2022.

[11] Jack Ganssle : Introduction to reentrancy. https://en.wikipedia.org/wiki/Reentrancy_(computing), 2001.

[12] Geeksforgeeks : Reentrant function. https://www.geeksforgeeks.org/reentrant-function/, 2018.

[13] Geeksforgeeks : Difference between process and thread. https://www.geeksforgeeks.org/difference-between-process-and-thread/, 2020.

[14] GNU : Gnu pth - the gnu portable threads. https://www.gnu.org/software/pth/, 2006.

[15] Gnu : Gnu hurd/ history/ porting the hurd to another microkernel. https://www.gnu.org/software/hurd/history/port_to_another_microkernel.html, 2015.

[16] The Open Group : The single unix specification interface tables. https://unix.org/apis.html, 2022.

[17] Guru99 : Process vs thread : What's the difference ? https://www.guru99.com/difference-between-process-and-thread.html, 2020.

[18] Gareth Halfacree : Arm launches first smt-capable cortex core. https://bit-tech.net/news/tech/cpus/arm-launches-first-smt-capable-cortex-core/1/, 2018.

[19] Howstuffworks : How mac os x works. https://computer.howstuffworks.com/macs/mac-os-x2.htm, 2021.

[20] Intel : Qu'est-ce que la technologie hyper-threading ? https://www.intel.ca/content/www/ca/fr/gaming/resources/hyper-threading.html, 2021.

[21] Kernel.org : The linux kernel api. https://www.kernel.org/doc/htmldocs/kernel-api/, 2022.

[22] Kernel.org : The linux kernel api. https://docs.kernel.org/core-api/kernel-api.html, 2022.

[23] Gregory Kesden : Courses notes : Lecture 4 (threads). https://www.andrew.cmu.edu/user/gkesden/ucsd/classes/sp17/cse120-a/applications/ln/lecture4.html, 2017.

[24] P. Kongetira, K. Aingaran et Kunle Olukotun : Niagara : A 32-way multithreaded sparc processor. *Micro, IEEE*, 25:21 – 29, 04 2005.

[25] Aurélien Lagny : Quelques processeurs amd threadripper 7000 storm peak en 96 cores et 192 threads en ballade. https://www.cowcotland.com/news/84120/quelques-processeurs-amd-threadripper-7000.html, 2022.

[26] Adnan Mostafa : The best free standalone terminals for windows 10 (2020). https://dev.to/adnanmostafa/the-best-free-standalone-terminals-for-windows-2019-kmj, 2020.

[27] James L. Peterson et Abraham. Silberschatz : *Operating system concepts / James L. Peterson, Abraham Silberschatz.* Addison-Wesley Pub. Co Reading, Mass, 1983.

[28] Diwas Poudel : The secret of cpu hyperthreading in depth. https://ourtechroom.com/tech/secret-of-cpu-hyperthreading/, 2021.

[29] A. P. Shanthi : Cache coherence. http://www.cs.umd.edu/~meesh/411/CA-online/chapter/cache-coherence-i/index.html, 2018.

[30] Abraham Silberschatz, Greg Gagne et Peter Baer Galvin : *Operating System Concepts.* Wiley, 6 édition, 2002.

[31] Abraham Silberschatz, Peter Baer Galvin et Greg Gagne : *Operating System Concepts, 10th Edition.* Wiley, 2018.

[32] StackOverflow : Threads vs processes in linux. https://stackoverflow.com/questions/807506/threads-vs-processes-in-linux, 2009.

[33] StackOverflow : What exactly is a reentrant function ? https://stackoverflow.com/questions/2799023/what-exactly-is-a-reentrant-function, 2011.

[34] StackOverflow : Why is this code reentrant but not thread-safe. https://stackoverflow.com/questions/9116598/why-is-this-code-reentrant-but-not-thread-safe, 2014.

[35] StackOverflow : Which cache-coherence-protocol does intel and amd use ? https://stackoverflow.com/questions/31876808/which-cache-coherence-protocol-does-intel-and-amd-use, 2015.

[36] StackOverflow : Is hyperthreading / smt a flawed concept ? https://stackoverflow.com/questions/23078766/is-hyperthreading-smt-a-flawed-concept, 2018.

[37] StackOverflow : What is the point of mesi on intel 64 and ia-32. https://stackoverflow.com/questions/49843709/what-is-the-point-of-mesi-on-intel-64-and-ia-32, 2018.

[38] Chris Thompson : Using vip for cache coherency hardware implementations. https://www.techdesignforums.com/practice/technique/amba4-ace-cache-coherency-vip/, Février 2013.

[39] Wikipedia : Cpu cache. https://en.wikipedia.org/wiki/CPU_cache, Juillet 2016.

[40] Wikipedia : Mémoire cache. https://fr.wikipedia.org/wiki/MÃŀmoire_cache, mars 2016.

[41] Wikipedia : Ultrasparc t1. https://fr.wikipedia.org/wiki/UltraSPARC_T1, 2018.

[42] Wikipedia : Gnu hurd. https://fr.wikipedia.org/wiki/GNU_Hurd, 2019.

[43] Wikipedia : Mesif protocol. https://en.wikipedia.org/wiki/MESIF_protocol, Septembre 2019.

[44] Wikipedia : Cache coherence. https://en.wikipedia.org/wiki/Cache_coherence\#cite_note-10, Juillet 2020.

[45] Wikipedia : Gnu portable threads. https://en.wikipedia.org/wiki/GNU_Portable_Threads, 2020.

[46] Wikipedia : Green threads. https://en.wikipedia.org/wiki/Green_threads, 2020.

[47] Wikipedia : Kernel (operating system). https://en.wikipedia.org/wiki/Kernel_(operating_system), 2020.

[48] Wikipedia : Mesi protocol. https://en.wikipedia.org/wiki/MESI_protocol, Juillet 2020.

[49] Wikipedia : Moesi protocol. https://en.wikipedia.org/wiki/MOESI_protocol, Juillet 2020.

[50] Wikipedia : Msi protocol. https://en.wikipedia.org/wiki/MSI_protocol, Juillet 2020.

[51] Wikipedia : Noyau de système d'exploitation. https://fr.wikipedia.org/wiki/Noyau_de_syst%C3%A8me_d%27exploitation, 2020.

[52] Wikipedia : Reentrancy (computing). https://en.wikipedia.org/wiki/Reentrancy_(computing), 2020.

[53] Wikipedia : Réentrance. https://fr.wikipedia.org/wiki/R%C3%A9entrance, 2020.

[54] Wikipedia : Thread (informatique). https://fr.wikipedia.org/wiki/Thread_(informatique), 2020.

[55] Wikipedia : Ultrasparc t1. https://en.wikipedia.org/wiki/UltraSPARC_T2, 2020.

[56] Wikipedia : Ultrasparc t2. https://fr.wikipedia.org/wiki/UltraSPARC_T2, 2020.

[57] Wikipedia : Alan turing. https://fr.wikipedia.org/wiki/Alan_Turing, 2021.

[58] Wikipedia : Barrel processor. https://en.wikipedia.org/wiki/Barrel_processor, 2021.

[59] Wikipedia : Dragon protocol. https://en.wikipedia.org/wiki/Dragon_protocol, Janvier 2021.

[60] Wikipedia : Google fuchsia. https://fr.wikipedia.org/wiki/Google_Fuchsia, 2021.

[61] Wikipedia : Hyper-threading. https://fr.wikipedia.org/wiki/Hyper-threading, 2021.

[62] Wikipedia : Hyper-threading. https://en.wikipedia.org/wiki/Hyper-threading, 2021.

[63] Wikipedia : John von neumann. https://fr.wikipedia.org/wiki/John_von_Neumann, 2021.

[64] Wikipedia : Multithreading (computer architecture). https://en.wikipedia.org/wiki/Multithreading_(computer_architecture), 2021.

[65] Wikipedia : Power8. https://en.wikipedia.org/wiki/POWER8, 2021.

[66] Wikipedia : Power9. https://en.wikipedia.org/wiki/POWER9, 2021.

[67] Wikipedia : Qnx. https://en.wikipedia.org/wiki/QNX, 2021.

[68] Wikipedia : Simultaneous multithreading. https://en.wikipedia.org/wiki/Simultaneous_multithreading#Modern_commercial_implementations, 2021.

[69] Wikipedia : Symbian. https://en.wikipedia.org/wiki/Symbian, 2021.

[70] Wikipedia : Symbian. https://en.wikipedia.org/wiki/K42, 2021.

[71] Wikipedia : Temporal multithreading. https://en.wikipedia.org/wiki/Temporal_multithreading, 2021.

[72] Wikipedia : Thread safety. https://en.wikipedia.org/wiki/Thread_safety, 2021.

[73] Wikipedia : Ultrasparc t1. https://en.wikipedia.org/wiki/UltraSPARC_T1, 2021.

[74] Wikipedia : Xnu. https://en.wikipedia.org/wiki/XNU, 2021.

[75] Wikipedia : Linux kernel interfaces. https://en.wikipedia.org/wiki/Linux_kernel_interfaces, 2022.

[76] Wikipedia : Motherboard. https://en.wikipedia.org/wiki/Motherboard, 2022.

[77] Wikipedia : Posix. https://fr.wikipedia.org/wiki/POSIX, 2022.

[78] Wikipedia : Posix. https://en.wikipedia.org/wiki/POSIX, 2022.

[79] Wikipedia : Windows api. https://fr.wikipedia.org/wiki/Windows_API, 2022.

[80] Wikipedia : Windows api. https://en.wikipedia.org/wiki/Windows_API, 2022.