

Séance 3

Système d'exploitation multiprocesseurs



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons
Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Rappels

- Backup et archivage pour protection et exigence légale
 - Exigence matérielle et conditions de stockage
 - Schedule de backup et techniques de restauration
- Gestion des crashes et restauration des données
 - Check de consistance des données et reconstruction
 - Problèmes de mise à jour des métadonnées
- Utilisation d'un système de fichiers journalisé

Principe de la journalisation et gestion des logs

Objectifs

- Ordinateurs avec **plusieurs processeurs** ou cœurs
 - Organisation d'ordinateurs multiprocesseur symétrique (SMP)
 - Systèmes avec processeurs multicœurs
 - Multithreading et SMP/multicœurs
- Caractéristiques d'**OS multiprocesseur** (MPOS)

Fonctionnalités et points d'attention au niveau du design
- Techniques d'**ordonnancement** spécifiques aux multicœurs
 - Définition des différents niveaux de granularité
 - Ordonnancement des processus et des threads

ERSIN

Multicore



5 CORE SOLDER

Multi{processseur, cœur}

Machine séquentielle

- Ordinateur initialement vu comme une **machine séquentielle**
 - Programmeur écrit algorithme comme séquence d'instructions
 - Exécution d'une instruction à la fois et séquentiellement
 - Chaque instruction est composée d'une séquence d'opérations
- **Vue simpliste** ne colle pas complètement à la réalité
 - Plusieurs signaux en même temps niveau micro-opérations
 - Pipelining avec au moins fetch et execute en même temps

Parallélisme

- Nouveauté et **chute du prix** du hardware des ordinateurs

Designers cherchent opportunités d'ajout de plus de parallélisme

- Amélioration tant des **performances** que de la fiabilité

Plus de calculs en même temps, différents ou les mêmes

- **Trois approches** répandues pour produire du parallélisme

- Multiprocesseur symétrique (SMP)
- Processeurs multi-cœurs
- Cluster d'ordinateurs

Multiprocesseur symétrique (1)

- Ordinateur autonome avec plusieurs caractéristiques
 - 1 Deux ou plus processeurs similaires avec capacité comparable
 - 2 Processeurs connectés par bus et partage mémoire principale
 - 3 Accès aux E/S partagé par canaux identiques ou séparés
 - 4 Tous les processeurs réalisent les mêmes fonctions
 - 5 Système contrôlé par un seul système d'exploitation intégré

- Présence d'une **grande coopération** entre les processeurs

Interactions inter-processus sur base d'éléments de données

Multiprocesseur symétrique (2)

- **Quatre avantages** principaux par rapport aux uni-processeurs
 - Augmentation des performances si calculs parallélisables
 - Plus haute disponibilité notamment si panne d'un processeur
 - Croissance incrémentale du système par ajout de processeurs
 - Production plusieurs configurations d'un système par vendeurs
- Il s'agit de **bénéfices potentiels** et pas garantis

L'OS doit offrir des services et outils pour exploiter le parallélisme
- Présence plusieurs processeurs **transparente** pour l'utilisateur

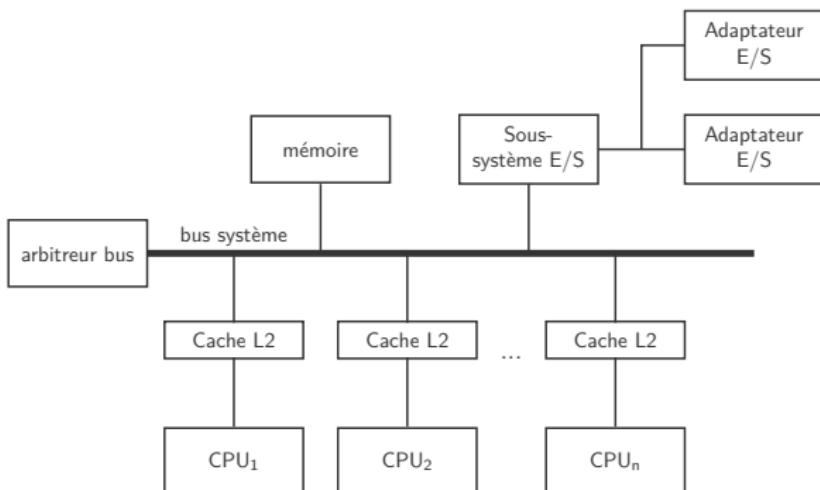
Ordonnancement, synchronisation entre processeurs géré par l'OS

Organisation générale SMP

- Plusieurs processeurs partageant mémoire et accès E/S

Cache L1, unité de contrôle, ALU et registres par CPU

- Problèmes de cohérence de cache avec la cache L1



Multithreading et SMP

- Multithread  et SMP sont deux aménagements indépendants
 - Séparation processus application et kernel avec threads
 - Deux processus en parallèle avec SMP même sans threads
- Utilisation combinée des threads et du SMP

Les deux techniques se complémentent très bien

Ordinateur multicœurs

- Un ou plusieurs processeurs combinés sur une **même chip**
 - Multiprocesseur on-chip, avec processeurs appelés cœurs
 - Contient registre, ALU, hardware pipeline, cache L1...
- Par exemple, **Intel Core i7-990X** contient six processeurs x86
 - Des caches L1 d'instructions et de données, chacune 32 Ko
 - Une cache L2 de 256 Ko pour chaque cœur également
 - Une cache L3 de 12 Mo partagée entre tous les cœurs

Multithreading et multicœurs

- Support d'une application avec **plusieurs threads**
 - Application intense au niveau de l'utilisation processeur
 - Difficultés de design et de performances
- Mesure du **bénéfice potentiel** sur la performance (Amdahl)

$$\text{Speedup} = \frac{\text{temps sur un seul processeur}}{\text{temps sur } N \text{ processeurs parallèles}} = \frac{1}{(1 - f) + \frac{f}{N}}$$

Où

- N processeurs parallèles
- Fraction f du temps d'exécution infiniment parallélisable

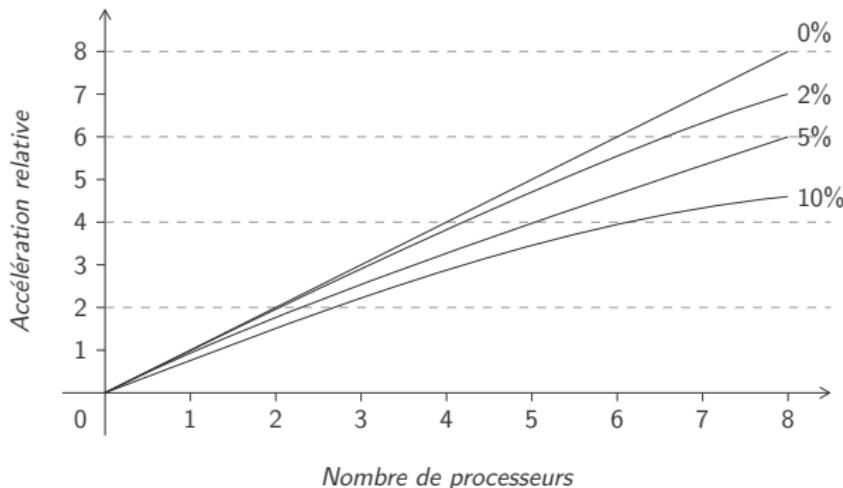


Performance

- Même un peu de séquentiel a un **impact sur les performances**

Avec seulement $f = 0.9$, bénéfice de 4.7 avec 8 processeurs

- En réalité, ne pas oublier les **overheads de communication**



Classes d'application

- Applications nativement **multithreadées**

Peu de processus avec des threads (Lotus Domino, Siebel CRM)

- Applications composées de **plusieurs processus**

Plusieurs processus single-thread (DB Oracle, SAP, PeopleSoft)

- **Applications Java** avec sa machine virtuelle très threadée

Serveurs d'application, web avec Tomcat, ou encore J2EE

- Applications **multi-instances**

Avec de la virtualisation pour les parties qui doivent être isolées

Valve Game Software

- Compagnie **Valve** de divertissement et technologie
 - Développement de plusieurs jeux très populaires
 - Moteur graphique d'animations Source
- Source redéveloppé pour exploiter les **processeurs multicœurs**
 - Threading gros grain par module (rendering, AI, physique...)
 - Threading fin grain pour tâche similaire (même action sur liste)
 - Threading hybride mélangeant les deux approches précédentes

OS multiprocesseur



Exécution sur un SMP

- **Self-scheduling** automatique par les processeurs depuis un pool
 - Le kernel peut être exécuté sur n'importe quel processeur
 - Design du kernel sous forme de plusieurs processus
- Augmentation de la **complexité du multiprocessor OS** (MPOS)
 - Gestion du partage des ressources comme structure de données
 - Coordination des actions comme accès à un périphérique
 - Résoudre et synchroniser les demandes de ressources

Fonctionnalité MPOS (1)

- Fonctionnalité **multiprogrammation** sur système uniprocesseur
 - Utilisation transparente du système pour l'utilisateur*
- Ajout de fonctionnalités spécifiques aux **multiprocesseurs**
 - Possibilité de mieux exploiter le hardware*
- **Cinq points d'attention** pour le design d'un MPOS
 - Processus ou threads **concurrents** simultanément
 - Code kernel réentrant car plusieurs processeurs peuvent l'exécuter*
 - Chaque processeur fait de l'**ordonnancement**
 - Difficile de forcer une politique d'ordonnancement*

Fonctionnalité MPOS (2)

- **Cinq points d'attention** pour le design d'un MPOS
 - Importance de la **synchronisation** des exécutions parallèles
Espace mémoire et E/S peuvent être partagés
 - Gestion de la **mémoire** pour exploiter le parallélisme
Coordination du paging sur les différents processeurs
 - **Dégénération progressive** en cas de panne d'un processeur
Restructuration dynamique des tables de gestion
- **Extension** des solutions uni-processeur
Quelques éléments néanmoins spécifiques aux MPOS

Système multicoeurs

- Éléments de design supplémentaires pour gérer **les multicoeurs**
De plus en plus de cœurs et caches dédiée/partagée
- Gérer intelligemment et efficacement les **ressources on-chip**
Matcher parallélisme many-core inhérent avec celui d'application
- **Potentiel de parallélisme** se trouve à trois niveaux
 - Parallélisme d'instructions hardware dans chaque cœur
 - Multiprogrammation ou multithreading dans les processeurs
 - Une application peut avoir plusieurs processus/threads

Parallélisme d'application

- Subdivision d'une application en **plusieurs tâches** parallèles
 - Le choix de la découpe incombe au développeur
 - Un code peut ou doit être exécuté en asynchrone ou parallèle
- **Deux interventions** possibles pour le parallélisme
 - Compilateur et feature langage de programmation pour design
 - Support par OS pour allocation ressources aux tâches parallèles
- Par exemple **Grand Central Dispatch** (GCD) de macOS et iOS
 - Mécanisme de type pool de threads pour tâches en parallèle
 - Fonction anonyme (bloc) étend le langage pour parallélisme

Machine virtuelle

- OS pour multicœur fonctionne comme un **hyperviseur**

Chaque cœur est vu comme une machine virtuelle

- L'OS affecte un **processeur virtuel** à des applications
 - Gestion par l'application des ressources utilisées
 - Similaire aux deux processeurs virtuels kernel et user mode
 - Moins d'overhead switching processus en multiprogrammation

Ordonnanceur multiprocesseur



Système multiprocesseurs

- Difficultés de design avec **systèmes multiprocesseurs**

Considérations différentes au niveau des processus ou threads

- **Trois catégories** de systèmes avec plusieurs processeurs

- Multiprocesseur peu couplé/distribué ou cluster
- Processeur avec fonction spécialisée (processeur E/S...)
- Couplage fort avec mémoire partagée et contrôle intégré

Granularité (1)

- Caractérisation des multiprocesseurs par leur **granularité**

Granularité ou fréquence de la synchronisation, entre processus

- **Pas de synchronisation** explicite entre processus indépendants

Diminution temps de réponse, dans environnement time-sharing

Taille (grain)	Description	Intervalle synch (instructions)
Fin	Inhérent dans flux d'instructions simples	< 20
Moyen	Parallélisme tâche au sein d'une application	20 – 200
Gros	Processus concurrent en multiprogrammation	200 – 2000
Très gros	Calcul distribué sur nœuds dans un réseau	2000 – 1M
Indépendant	Processus sans liens	pas applicable

Granularité (2)

- Parallélisme avec très gros et **gros grain**
 - Synchronisation à très haut niveau des processus
 - Passage de uniprocesseur à multiprocesseurs très facile
- Parallélisme avec un **grain moyen**
 - Plusieurs threads à l'intérieur d'un même processus
 - Parallélisme potentiel à définir par le programmeur
 - Haut niveau de coordination et d'interaction inter-threads
- Parallélisme avec un **fin grain**

Utilisation très complexe du parallélisme

Ordonnancement (1)

- **Trois problèmes** liés pour ordonnancement multiprocesseurs
 - L'affectation de processus sur des processeurs
 - L'utilisation de la multiprogrammation sur un processeur
 - Le dispatching effectif d'un processus
- Affectation avec architecture multiprocesseurs **uniforme**
 - Traiter processeurs comme pool de ressources
 - Affectation statique d'un processus sur le même processeur
Problème potentiel d'avoir des processeur pas utilisés
 - Une seule file globale et affectation dynamique
L'information de contexte doit être dans une mémoire accessible

Ordonnancement (2)

- Nécessité de faire de la **multiprogrammation** par processeur
 - Un processus va passer du temps bloqué à attendre des E/S
 - Évident pour parallélisme gros grain et processus indépendants
 - Avoir plusieurs threads en même temps pour grain moyen
- Choix du processus à exécuter et **dispatching sur le processeur**
 - Choix plus simples parfois plus efficaces en multiprocesseur
 - Utilisation des priorités et de l'historique moins importantes

Ordonnancement de processus

- Quelques files (voire une) pour tous les processeurs
 - Utilisation éventuelle de la priorité pour les files
 - Choix d'un processeur d'exécution parmi un pool
- Diminution de l'influence de l'algorithme d'ordonnancement

Simplement utiliser FCFS avec affectation statique suffit



Ordonnancement de threads

- Exécution concurrente de **plusieurs threads** d'un processus

Coopération, exécution concurrente dans même espace d'adresses
- Plus de **complexité** due à la présence de plusieurs processeurs
 - Avec un seul processeur, switching de processus = thread
 - Véritable parallélisme si présence de plusieurs processeurs
 - Impact non négligeable si grande interaction entre threads
- **Quatre approches** pour scheduler des threads

Partage de charge, gang, processeur dédié, dynamique

Partage de charge (1)

- Une file globale de threads avec processeur idle qui y choisit
 - Partage de la charge entre les processeurs
 - Pas besoin d'avoir un ordonnanceur centralisé
 - Plusieurs organisations possibles pour la file (priorité...)
- Trois versions principales du partage de charge
 - FCFS place les threads d'un processus dans la file
 - Processus avec plus petit nombre de threads comme priorité
 - Ajout de la préemption à la solution précédente

Partage de charge (2)

- Plusieurs **désavantages** au partage de charges
 - File centrale à partager et protéger par mutex
 - Thread préempté redémarre sur autre processeur (! cache)
 - Peu de liens entre les threads d'un même processus
- Algorithme le plus souvent utilisé pour **sa simplicité**
 - Mach utilise deux files (globale et locale par processeur)
 - Possibilité de bounder un thread à un processeur donné

Gang scheduling



- Ordonnancer directement un **groupe de processus**
 - Intérêt d'envoyer en même temps processus liés et coordonnés
 - Diminution de l'overhead de l'exécution du scheduler
- Plusieurs techniques d'**ordonnancement de groupes**
 - Coscheduling sur multiprocesseurs Cm* groupe des tâches
 - Gang scheduling pour plusieurs threads d'un même processus
 - Très utilisé pour grain moyen ou fin où threads importants
 - Minimiser les switches de processus en activant des threads
 - Plus efficace que FCFS avec load sharing

Processeur dédié

- Dédier un **groupe de processeurs** pour une application
 - Jusqu'à ce que l'application finisse son exécution
 - Pas de multiprogrammation au niveau des processeurs
- **Deux arguments** majeurs en faveur de cette technique
 - Utilisation des processeurs moins critique si très grand nombre
 - Diminuer le switch de processus est plus bénéfique globalement
- Problème similaire à celui de la **gestion de la mémoire**
 - Combien de processeurs faut-il assigner à chaque processus ?
 - Combien de cadres faut-il affecter à chaque processus ?

Dynamic scheduling (1)

- Outils du langage et système pour **changer nombre de threads**

Permet à l'OS d'ajuster la charge pour améliorer utilisation

- **Approche hybride** avec gestion par l'OS et l'application

- L'OS partitionne les processeurs entre les jobs
- Chaque job affecte des tâches sur des threads
- L'application choisit threads à arrêter quand processus stoppe

Dynamic scheduling (2)

- Lorsqu'un job fait une **demande** pour un/des processeur/s
 - 1 Si assez de processeurs idle, utilisés pour la requête
 - 2 Si nouveau job, il reçoit un processeur
Retiré à un autre job qui a actuellement plusieurs processeurs
 - 3 Portion requête peut être en suspens
Jusqu'à disponibilité processeur ou rétraction demande du job
- Lorsqu'un **processeur est libéré** (thread ou job)
 - 4 Scan de la file de requêtes non satisfaites
- Globalement **mieux** que gang ou processeur dédié

Ordonnancement multicœurs de threads

- **Ordonnancement multicœurs** traité comme multiprocesseurs
Garder les processeurs occupés avec une répartition de charge
- Il faut néanmoins **minimiser les accès mémoire** off-chip
 - Exploitation des caches locales et du principe de localité
 - Complexité lorsque mix de caches locales et partagées

Crédits

- <https://www.flickr.com/photos/76345608@N00/4607426544>
- <https://www.flickr.com/photos/16038409@N02/3897997969>
- <https://www.flickr.com/photos/qchen/439420443>