IX- SE AVANCÉS

HYPERTHREADING / MULTIPROCESSEUR

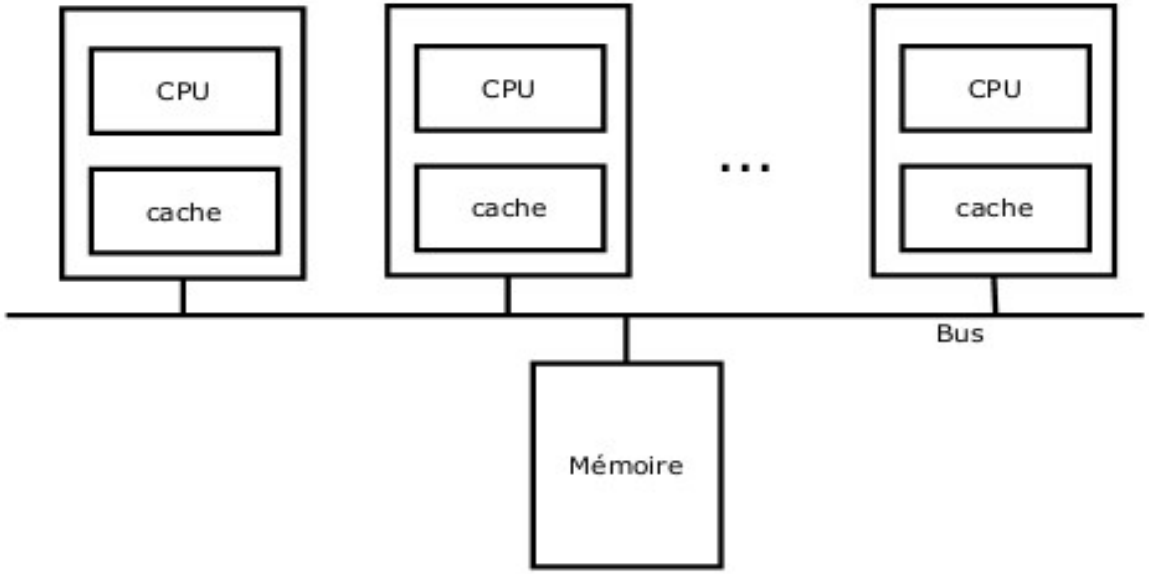
- **L'hyperthreading** consiste à dupliquer H fois les registres du processeur => quand on passe d'un processus à un autre on n'a plus à sauvegarder/restituer le contexte machine mais seulement à changer de registres (plus rapide).
- Le système d'exploitation voit alors le processeur comme **H processeurs** mais l'exécution n'est pas parallèle (seulement plus rapide).
- **Le Multi-cœur** consiste à placer K cœurs dans la même puce. Ils partagent les mémoires caches de niveau 2 et 3 mais ont chacun leur CPU et leurs caches de niveau 1.
- Le système d'exploitation gère alors les K cœurs et l'exécution est parallèle

HYPERTHREADING / MULTIPROCESSEUR

- Multiprocesseur consiste à placer N processeurs (puces) dans la même machine.
- Chaque processeur pouvant être multi-cœur et chaque cœur pouvant faire de l'hyperthreading.
- Si on a N processeurs chacun ayant K cœurs implémentant l'hyperthreading (H duplications des registres) alors le système d'exploitation voit **N*K*H processeurs**.

ARCHITECTURES À MÉMOIRE COMMUNE

- **Principe** : Les processeurs partagent la même mémoire. Chacun a ses caches propres.



PROBLÈMES LIÉS À LA MÉMOIRE COMMUNE ET AUX CACHES

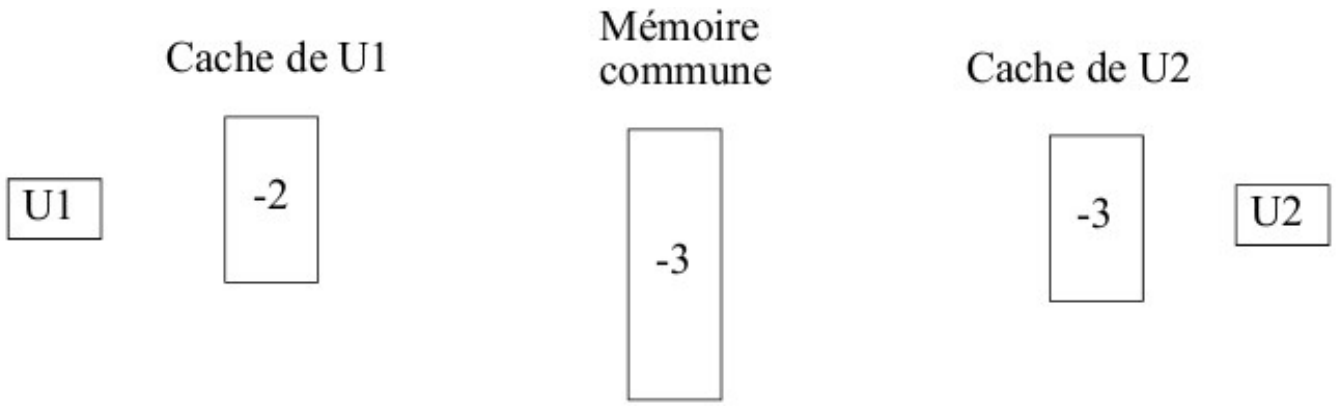
- Un processus (ou thread) P1 s'exécute sur le processeur U1
- Un second processus (ou thread) P2 s'exécute sur le processeur U2
- Ces 2 processus partagent une variable V qui est dans la mémoire commune de la machine
  - P1 modifie V par exemple : si (V<0) V ➔V+1
  - P2 utilise V par exemple : A←V/2
  - U1 et U2 ont des caches de niveau 1 propres.

PROBLÈMES LIÉS À LA MÉMOIRE COMMUNE ET AUX CACHES

- Au moment où P1 lit V pour savoir s'il est <0 , cette variable est mise dans le cache de U1 si elle n'y était pas déjà. Donc quand P1 modifie V (exécute V←V+1) elle est dans le cache de U1
- Au moment où P2 utilise V, cette variable est mise dans le cache de U2 si elle n'y était pas déjà
- Plusieurs cas peuvent se produire :

PROBLÈMES LIÉS À LA MÉMOIRE COMMUNE ET AUX CACHES

- 1er cas : V est déjà dans le cache de U2

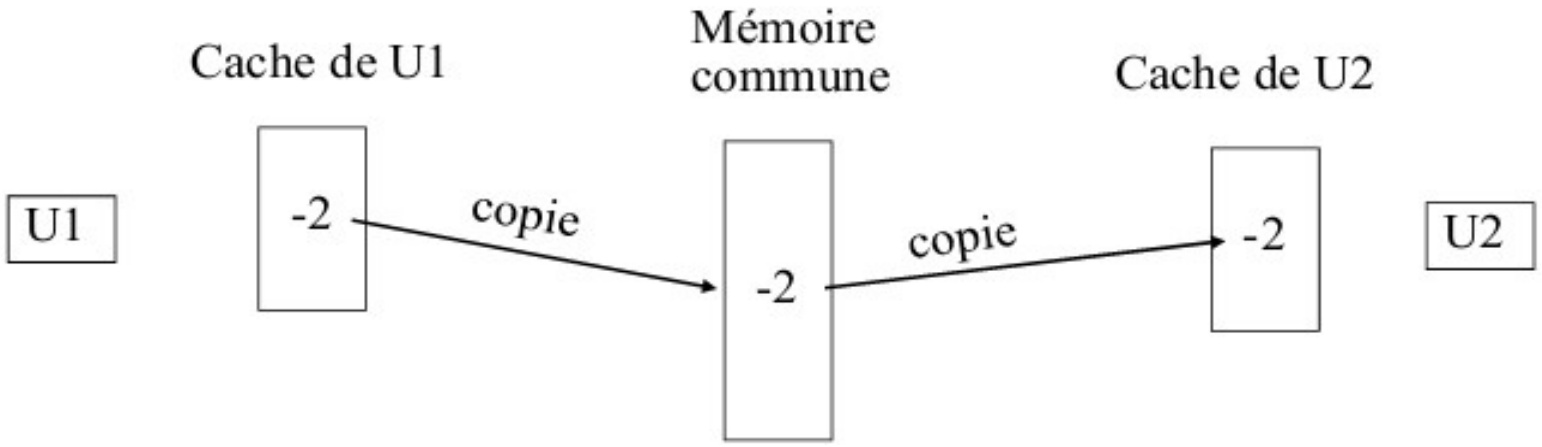


- V étant <0 P1 modifie V
- Quand P2 exécute A←V/2 il obtient A= -1,5

→ **C'est FAUX**

SOLUTION POSSIBLE AUX PROBLÈMES LIÉS À LA MÉMOIRE COMMUNE ET AUX CACHES

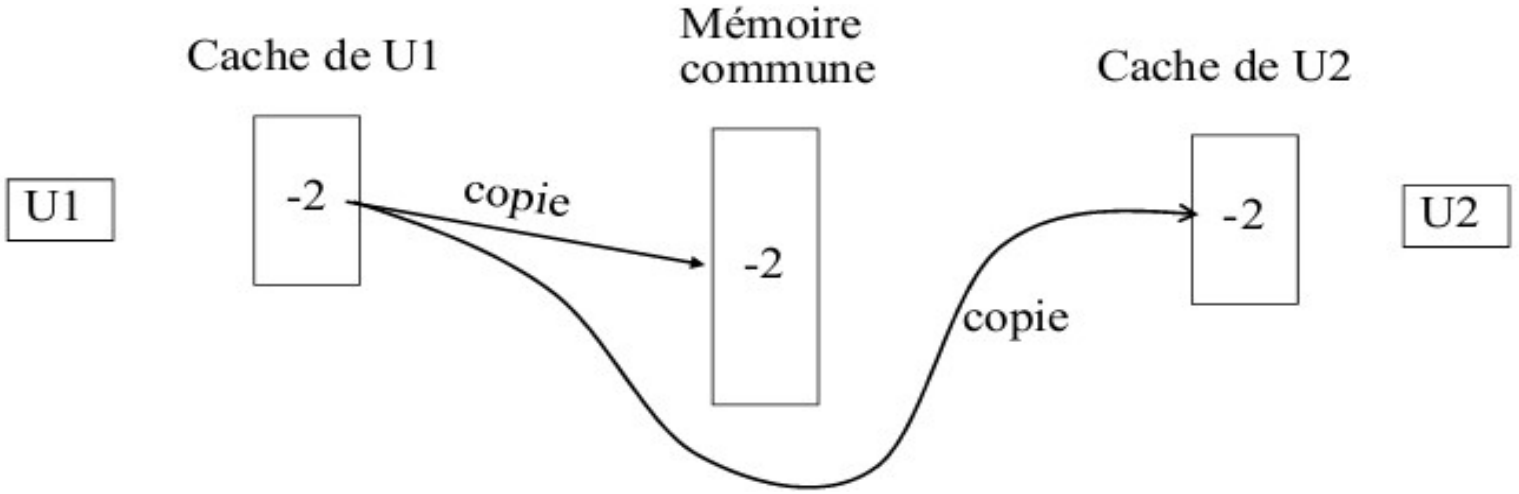
- Synchronisation par la mémoire



• V étant <0 alors P1 modifie V
• Comme V est partagé, la copie en mémoire du cache de U1 est faite
• Le bloc contenant V dans le cache de U2 est invalidé
• Quand P2 veut exécuter A←V/2 une copie est faite dans son cache depuis la mémoire. Le calcul de P2 donne bien A= -1

SOLUTION POSSIBLE AUX PROBLÈMES LIÉS À LA MÉMOIRE COMMUNE ET AUX CACHES

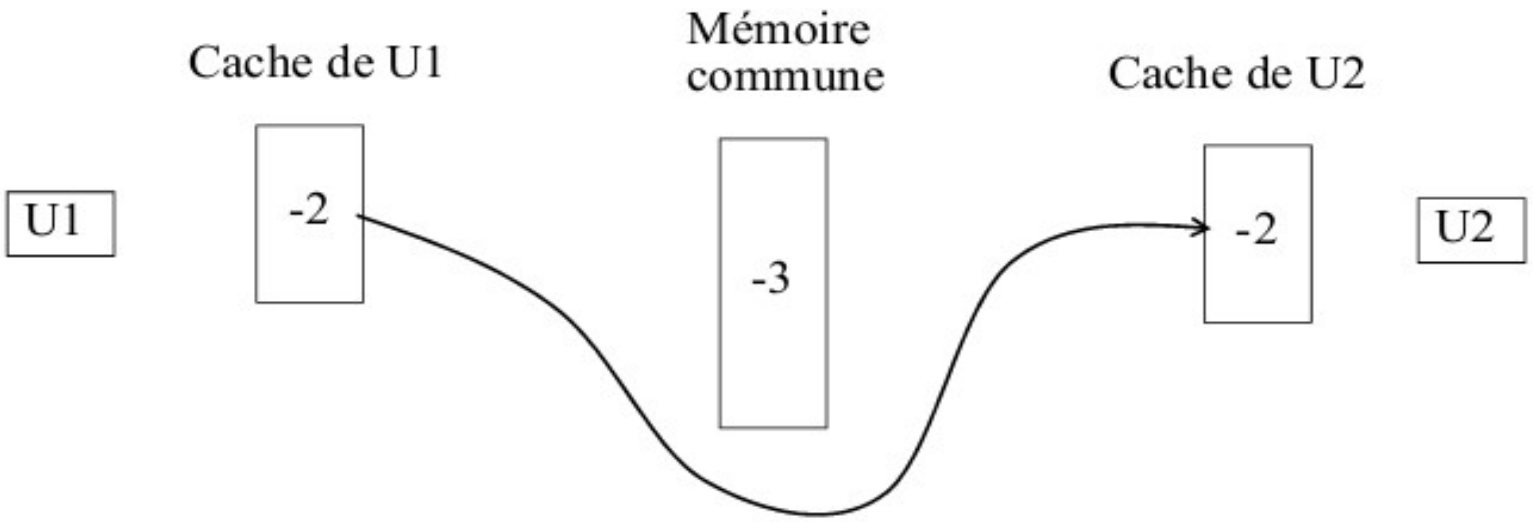
- Synchronisation par la mémoire (version optimisée)



• V étant <0 alors P1 modifie V
• Comme V est partagé, la copie en mémoire du cache de U1 est faite et le bloc contenant V dans le cache de U2 est mis à jour.
• Le calcul de P2 donne bien A= -1

SOLUTION POSSIBLE AUX PROBLÈMES LIÉS À LA MÉMOIRE COMMUNE ET AUX CACHES

- Synchronisation des caches



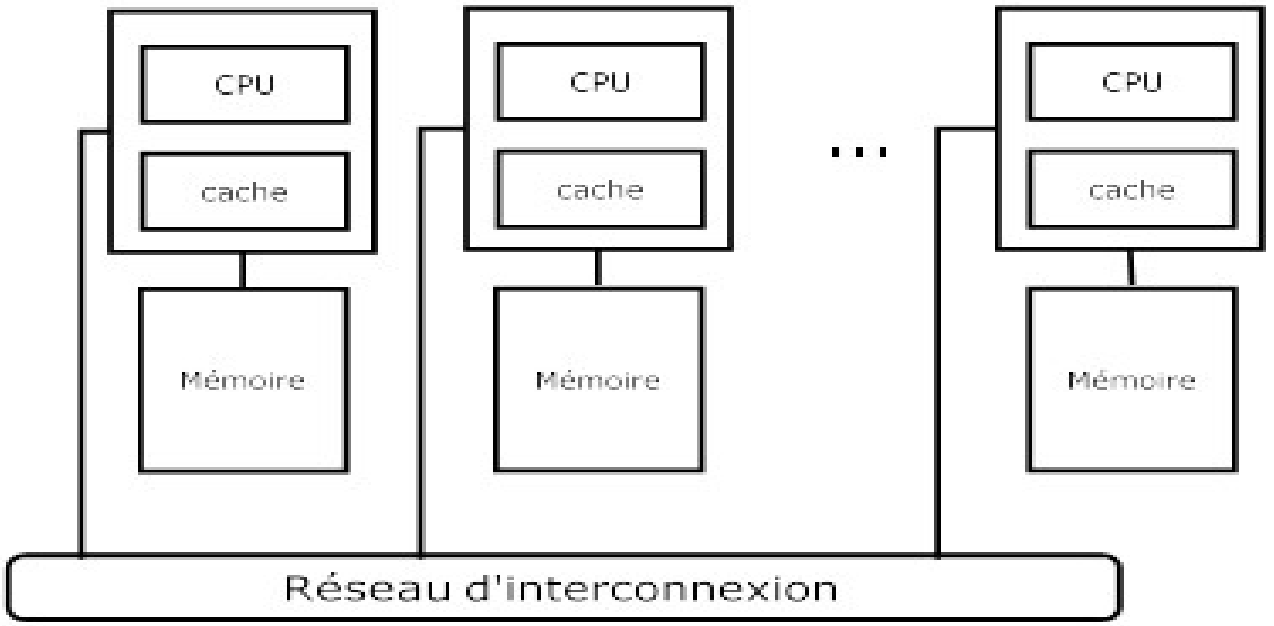
- V étant <0 alors P1 modifie V
- Comme V est partagé la modification du cache de U1 est reportée sur celui de U2
- Le calcul de P2 donne bien A= -1

SYNCHRONISATION DES CACHES

- Un gestionnaire des caches qui sait ce que contient chaque cache :
  - Lors d'un transfert mémoire → cache il vérifie si la mémoire doit être mise à jour avant.
  - Lors d'utilisation simultanée de variables il synchronise les caches (copie d'un vers N)
- Optimisation :
  - Lors des transferts de données entre cache et mémoire on peut récupérer ces données sur le bus pour effectuer le transfert vers les caches concernées => mise à jour simultanée de plusieurs caches => gain de temps.

ARCHITECTURES À MÉMOIRES SÉPARÉES (CLUSTERS)

- Principe : Chaque processeur dispose de sa propre mémoire.
- Ils sont interconnectés par un réseau rapide (échange de données).



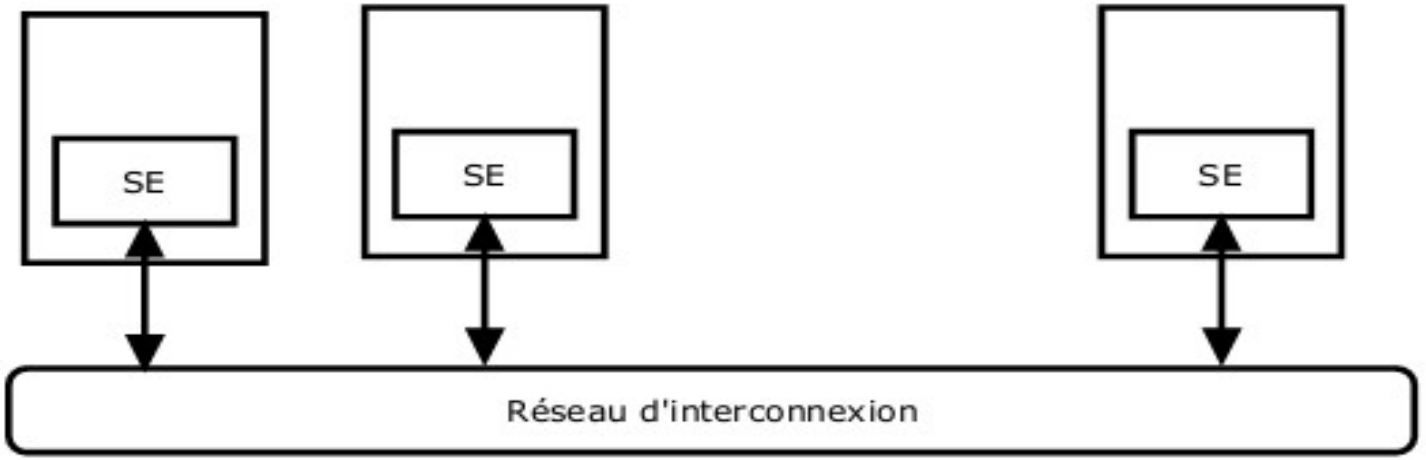
PROBLÈMES LIÉS AUX MÉMOIRES SÉPARÉES

- Distribution initiale des données
  - Certaines données doivent être dupliquées dans les mémoires des processeurs (comme le fait fork)
- Transferts de données par réseau
  - Les processeurs échangent des données par le réseau d'interconnexion (comme on le fait par un pipe)
- Synchronisation des données
  - Il faut s'assurer que les données modifiées en un point et utilisées en un autre soient à jour.
- Communication entre processus
  - Par le réseau d'interconnexion
    - Communication directe ou indirecte
    - Synchrone ou asynchrone
  - Possibilité d'appel de fonctions à distance (RPC)

SYSTÈMES MULTIPROCESSEURS

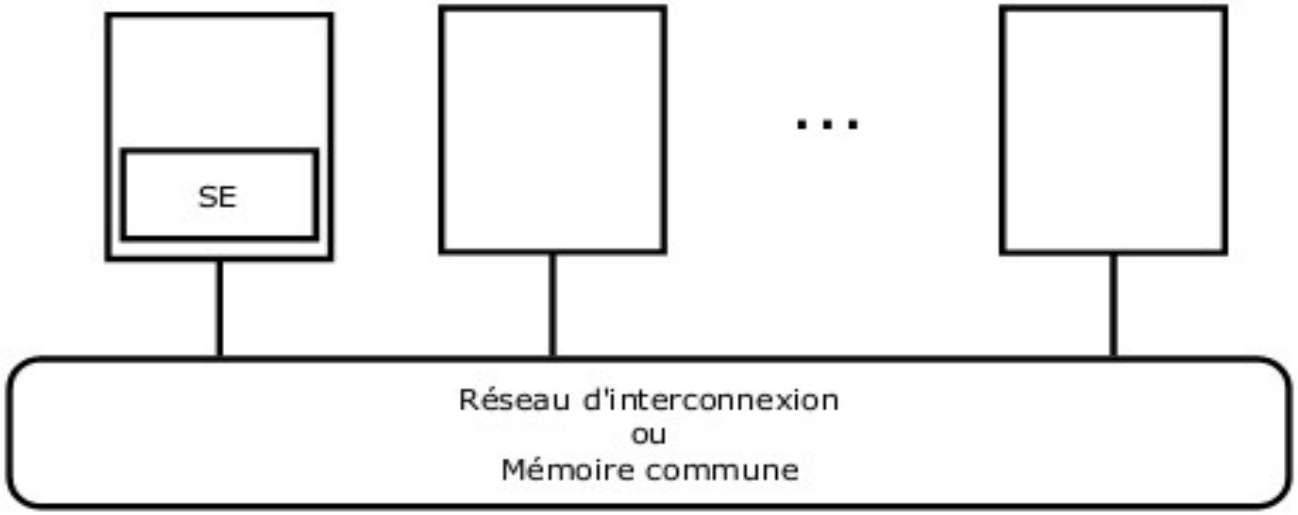
- Il existe 3 grands types de solutions :

1. **Chaque processeur a son SE** mais dans ce cas il n'y a pas vraiment de coopération possible entre processus sauf à faire comme sur un réseau classique



SYSTÈMES MULTIPROCESSEURS

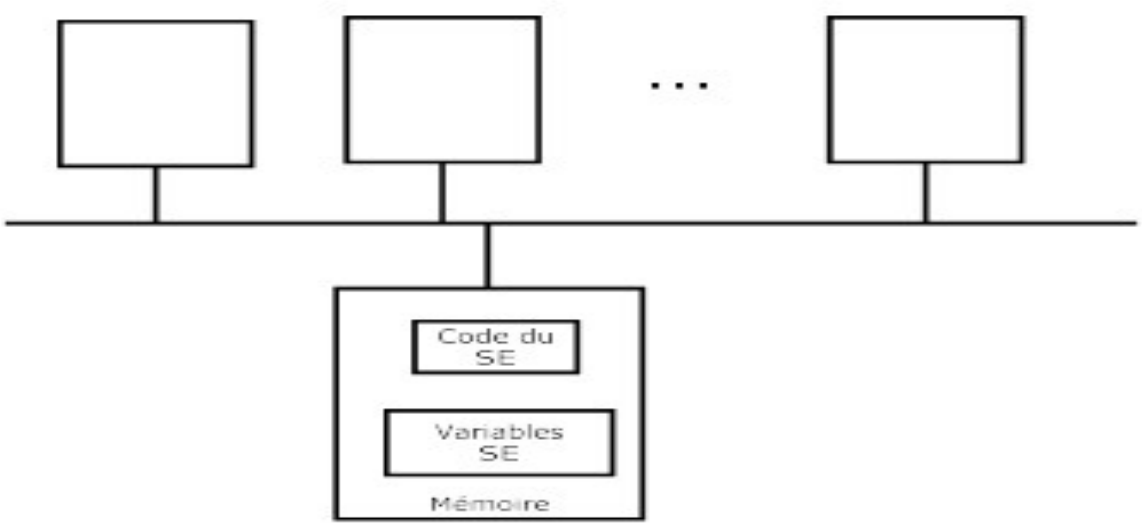
2. **Maître/Esclave** : 1 processeur a le SE, les autres exécutent les processus utilisateurs. A chaque nouveau processus créé le CPU maître choisit le CPU sur lequel il va l'implanter en essayant d'équilibrer la charge.



SYSTÈMES MULTIPROCESSEURS

3. **Symétrique** : Il n'y a qu'un SE en mémoire mais tous les CPU peuvent l'exécuter (cette mémoire est partagée). On protège alors le code du SE par un MUTEX pour qu'un seul CPU l'exécute à la fois

Pour optimiser on met des MUTEX différents selon les parties du SE (par ex un CPU peut exécuter l'ordonnanceur pendant qu'un autre exécute une E/S) => le SE est divisé en sections critiques mais il faut éviter les interblocages



ORDONNANCEMENT DES PROCESSUS

- Le système d'exploitation distribue les processus sur les différents processeurs :
  - Equilibrer les charges des processeurs
- Le problème est celui de l'ordonnancement des processus : Quand un processus démarre ou redémarre sur quel processeur faut-il le mettre ?
- A priori sur le moins chargé mais …

ORDONNANCEMENT DES PROCESSUS

- **Cas d'un processus non lié** (c-à-d n'ayant aucun lien avec d'autres)
- Choisir le processeur le moins chargé a un inconvénient qui est que si un processus A a tourné sur le processeur U1 puis est repris par le processeur U2, il y a perte de temps du fait que U1 avait ses pages en mémoire alors que U2 n'a rien.
- On peut éviter ça en choisissant qu'un processus soit toujours exécuté sur le même CPU choisi à la création du processus mais la charge n'est plus aussi bien répartie.

ORDONNANCEMENT DES PROCESSUS

- **Cas des processus liés :** Ce sont des processus qui échangent beaucoup entre eux (père-fils ou threads d'un même processus).
- Pour que les échanges se fassent de façon efficace il est préférable que :
  - ces processus tournent sur des CPU différents
  - ces processus tournent en même temps
- **Explication** :
  - – S'ils tournent sur le même CPU il n'y a pas de vrai parallélisme
  - – S'ils tournent sur des CPU différents mais pas au même moment les échanges sont différés
- **Solution** : créer des groupes de processus qui sont rendus actifs en même temps chacun sur un CPU différent et ont des quota de temps synchrones.

SYSTÈMES À MÉMOIRE COMMUNE

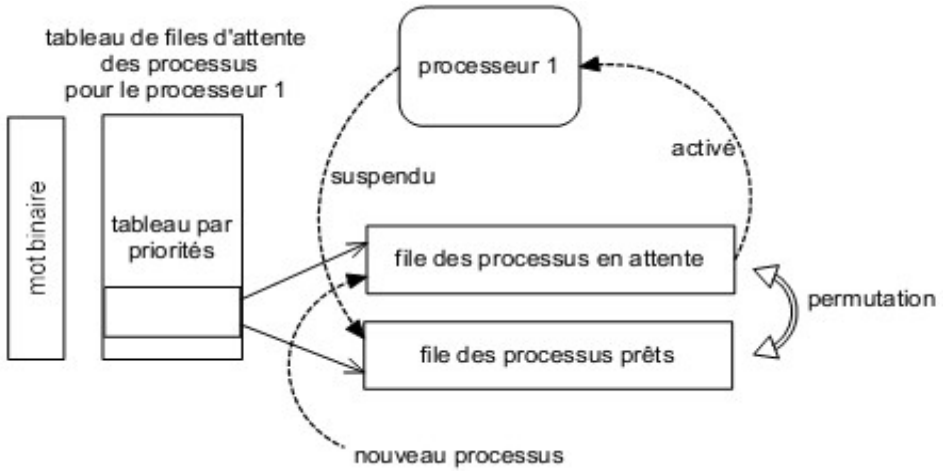
- Le système d'exploitation distribue les processus sur les différents processeurs en essayant d'équilibrer les charges des processeurs mais en respectant les règles d'ordonnancement suivantes :
  - Un processus qui a été suspendu reprendra de préférence sur le même processeur pour éviter de recharger la totalité des pages.
  - Un nouveau processus (fork) sera assigné au processeur le moins chargé.
  - Un processus surchargé (exec) pourra être transféré sur un autre processeur moins chargé.

SYSTÈMES À MÉMOIRE COMMUNE (CAS DE LINUX 2.6)

- L'ordonnanceur gère un tableau de files d'attente de processus par processeur (une entrée par priorité, 140 niveaux de priorité).
- Chaque entrée de ce tableau désigne une file des processus en attente et une file des processus prêts.
- Un mot binaire indique quelles sont les entrées du tableau qui désignent des files de processus en attente non vides. Par exemple si on a des processus en priorité 2 et en priorité 4 ce mot aura les bits 2 et 4 à 1 (les autres à 0).
- Ceci permet de savoir rapidement dans quelle entrée du tableau aller chercher (celui de plus forte priorité non vide donc le 1er bit à un de ce mot binaire).
- Les processus en attente de même priorité sont exécutés dans l'ordre (parcours de la file des processus en attente).

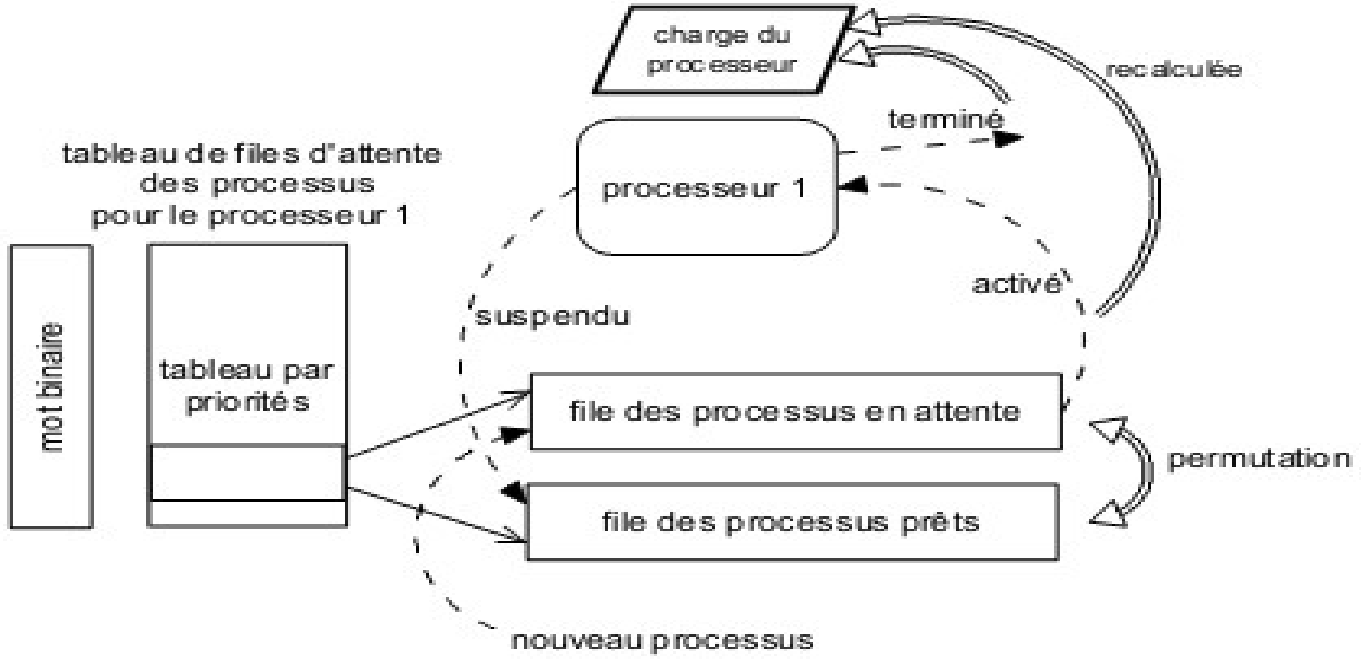
SYSTÈMES À MÉMOIRE COMMUNE (CAS DE LINUX 2.6)

- Quand un processus termine son quota de temps :
  - Un nouveau quota lui est affecté
  - Il est déplacé de la file des processus en attente vers celle des processus prêts de même priorité
  - S'il était le dernier processus en attente de cette priorité, la file des processus prêts et celle des processus en attente sont permutées.



SYSTÈMES À MÉMOIRE COMMUNE (CAS DE LINUX 2.6)

- Quand un processus démarre :
  - Si le processus est nouveau il est affecté au processeur le moins chargé
  - La charge du processeur sur lequel il est lancé est recalculée.



ORDONNANCEMENT SUR UN CLUSTER

- Chaque CPU fait son propre ordonnancement. Le problème est alors de choisir sur quel CPU on implante un nouveau processus.
- On essaye d'équilibrer la charge et de minimiser les communications entre CPUs.

1. Si on connaît d'avance les besoins des processus en terme de :
   - temps CPU
   - mémoire
   - communication
- On peut faire cela de façon statique (c'est le cas pour un gros programme qui tourne sur un cluster)

ORDONNANCEMENT SUR UN CLUSTER

2. Sinon on peut le faire dynamiquement de 3 façons :

- 1. Quand un processus en crée un autre, le CPU sur lequel il est le garde ou cherche un CPU peu chargé qui veuille le prendre
- 2. Quand un processus se termine sur un CPU celui-ci consulte les autres CPU pour voir s'ils sont plus chargés que lui, si c'est le cas il leur demande un processus à exécuter
- NB : on peut combiner ces 2 méthodes.
- 3. Les CPUs maintiennent dans un fichier commun leur état d'occupation. Quand un processus est créé sur un CPU celui-ci consulte ce fichier pour voir quel CPU est le moins chargé.

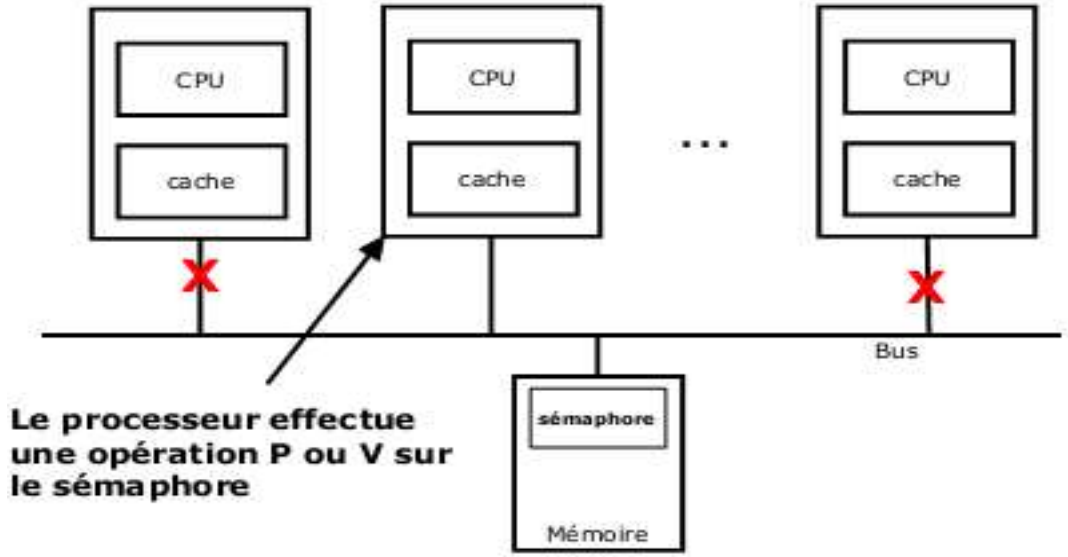
PROBLÈME DE SYNCHRONISATION

- Sur un monoprocesseur on utilise des opérations atomiques comme P et V en bloquant les interruptions
- Sur un multiprocesseur cela ne suffit pas car les autres CPU continuent à fonctionner
- La solution dépend de l'architecture :
  - Mémoire commune
  - Mémoires distribuées et réseau d'interconnexion

SYNCHRONISATION AVEC MÉMOIRE COMMUNE

- Le sémaphore est en mémoire et n'est pas copié dans les caches (variable protégée)
- La mémoire est verrouillée pendant une opération atomique (de type TSL) pour éviter qu'un autre processeur y accède.

- Les processeurs possèdent une ligne physique de verrouillage de la mémoire.
- Les instructions accédant à la mémoire peuvent provoquer une activation de cette ligne (préfixe lock sur les processeurs Intel).



SYNCHRONISATION AVEC MÉMOIRES DISTRIBUÉES

- Le sémaphore S est dans la mémoire de l'un des processeurs.
- Les opérations P et V sur ce sémaphore S sont exécutées par ce processeur seulement.
- Elles lui sont demandées par réseau (appel de fonction à distance) par les autres processeurs
- Lors d'une opération P le processus demandeur est bloqué jusqu'à ce que l'opération soit terminée. A la fin de l'opération, selon la valeur de S, soit il reste bloqué sur le sémaphore S soit il continue.

SYNCHRONISATION AVEC MÉMOIRES DISTRIBUÉES

- Lors d'une opération V, le processus demandeur est bloqué jusqu'à ce que l'opération soit terminée.
- A la fin de l'opération le processus demandeur est débloqué et l'un des processus bloqués sur le sémaphore S (sur l'une des machines) est également débloqué.
- On utilise un protocole réseau (synchrone) et des primitives bloquantes (send et receive) pour ces opérations.

GESTION DES MÉMOIRES DISTRIBUÉES DANS UN CLUSTER

- Sur les clusters on n'a pas de mémoire partagée mais le SE peut faire comme si.
- Chaque CPU utilise des adresses virtuelles qui correspondent à des pages mémoire présentes ou pas. Dans un système classique une page non présente => le SE va la chercher sur disque.
- Dans un cluster le SE peut aller la chercher sur un autre CPU en demandant au SE de celui-ci de la lui envoyer => l'adresse virtuelle correspond à adresser toutes les mémoires de tous les CPU du cluster (ou au moins une partie de chacune)
- Quand un CPU modifie une page ainsi partagée le SE envoie un message aux autres SE pour qu'il marquent cette page comme non présente chez eux => au prochain accès ils referont un transfert.

PARALLÉLISME

- Le système d'exploitation distribue la charge entre processeurs au niveau des processus donc :
- Si une application est mono-processus elle ne bénéficie du multiprocesseur que parce que les autres processus (système, autres utilisateurs …) s'exécutent sur d'autres processeurs.
- Pour bénéficier de la puissance d'un multiprocesseur, les applications doivent être écrites en processus parallèles.
- MAIS :
  - En général les langages de programmation ne sont pas adaptés à cela.
  - Les programmeurs n'ont pas toujours ces habitudes.
  - Un bon programmeur doit les avoir!

LANGAGES ET PARALLÉLISME

Plusieurs approches :

- Détection automatique du parallélisme par le compilateur : limité au parallélisme d'exécution des instructions dans les processeurs (pas de parallélisme de tâches).
- Création de processus ou de threads par le programmeur
  - fork et pthread_create en C ou C++
  - Classe Thread et interface Runnable en java
  - Type Task en ADA

LANGAGES ET PARALLÉLISME

- Parallélisation du traitement de données
  - PLINQ (Parallel Langage-Integrated Query) de Microsoft permet de traiter en parallèle les éléments d'une collection
- Parallélisation de structures de contrôle (langages spécialisés ou extensions de langages classiques)
  - OCCAML et OpenMP (Open Multi-Processing) permettent de paralléliser l'exécution de blocs de code, de boucles et d'alternatives

LES SYSTÈMES TEMPS RÉEL

- Un système d'exploitation en temps réel est la catégorie d'un système d'exploitation qui est utilisé pour gérer des applications en temps réel. C'est-à-dire que pour de tel système l'on a besoin d'une réponse correcte et dans un temps très court.
- Ce genre de système possède des spécificités autres que celles déjà vues.
- Un système en temps réel est généralement un système à but spécifique, de petite taille, à peu de frais produites en masse et avec les exigences de calendrier précis.
- Un système temps réel est conçu pour un seul but et vous ne pouvez pas l'utiliser pour d'autres tâches que celle spécifiée lors de sa conception.
- La plupart des systèmes en temps réel sont utilisés dans des environnements exigus qui place également une limitation de CPU et de mémoire pour ces systèmes.

LES SYSTÈMES TEMPS RÉEL

- L'exigence de temps, une caractéristique qui définit un système en temps réel qui permet de réagir aux événements en temps opportun. En général, un système en temps réel a trois exigences principales :

1. Prévisibilité: le comportement de synchronisation du système d'exploitation doit être prévisible
2. Gestion: le système d'exploitation doit gérer le calendrier et la programmation car il peut être au courant des délais de tâches. En outre, le système d'exploitation doit fournir des services de temps précis avec une résolution élevée.
3. Vitesse: Le système d'exploitation doit être rapide dans toutes les réponses, ce qui est important dans la pratique.

LES SYSTÈMES TEMPS RÉEL

- Un système temps réel offre au programmeur :
  - Les mécanismes de synchronisation (sémaphores)
  - Les mécanismes de communication (boîte à lettres, signaux …)
  - La possibilité de définir des tâches
  - La possibilité de gérer les interruptions
- L'objectif principal est la prise en compte des besoins d'urgence, d'importance et de réactivité. La seule contrainte est que tout soit exécuté à temps. C'est donc essentiellement un problème d'ordonnancement des tâches.

LES SYSTÈMES TEMPS RÉEL

- Principes d'ordonnancement des tâches :
  - Choix de l'ordonnancement
    - Statique (à l'avance) / Dynamique (au moment)
  - Priorités
    - Statiques / Dynamiques
  - Algorithmes
    - Préemptifs / Non préemptifs

LES SYSTÈMES TEMPS RÉEL

- Types de tâches :
  - Normales / Urgentes
  - Dépendantes / Indépendantes
  - Répétitives (périodiques / sporadiques)
  - Non répétitives
- **NB** : les tâches répétitives et les tâches urgentes sont critiques

SYSTÈMES EMBARQUÉS

- Ce sont des systèmes temps réels autonome, conçus pour des problèmes spécifiques, pour lesquels le matériel et l'application sont intimement liés.
- **Lorsqu'il n'y a pas de SE**, il y a un moniteur qui :
  - Initialise le matériel (registres du processeur et des contrôleurs de périphériques)
  - Contient les pilotes de périphériques
  - Offre une API minimale
  - Peut communiquer (USB ou JTAG) avec une autre machine qui :
    - Permet d'éditer et compiler les applications
    - Permet de les télécharger sur le système embarqué
    - Permet de les tester (traces, points d'arrêt …)
    - Parfois permet aussi la simulation
- Tout le reste est fait par l'application

SYSTÈMES EMBARQUÉS

- Lorsqu'il y a un SE, **deux approches** :
- Systèmes légers (ce sont des SE multi applications) :
  - 1. SE existant épuré pour ne garder que ce qui est nécessaire
    - Linux (Android)
    - IOS (Apple) dérivé de Mac OS X
    - Window pour mobiles (Microsoft)
  - 2. SE développé pour l'embarqué
    - Symbian (Nokia) , RIM (blackberry) …
    - Squawk (sunspots) est une machine virtuelle java
    - Système modulaire (c'est un SE mono application) :
- le SE est divisé en modules (ordonnanceur, horloge …) et, selon ce dont a besoin l'application, on n'installe que certains modules.
  - RTOS (système temps réel sous forme de bibliothèque)
  - Tiny OS (associé au langage de programmation NesC)

FIN