Le problème des Philosophes

La mémoire cache

Le problème des Philosophes : définition

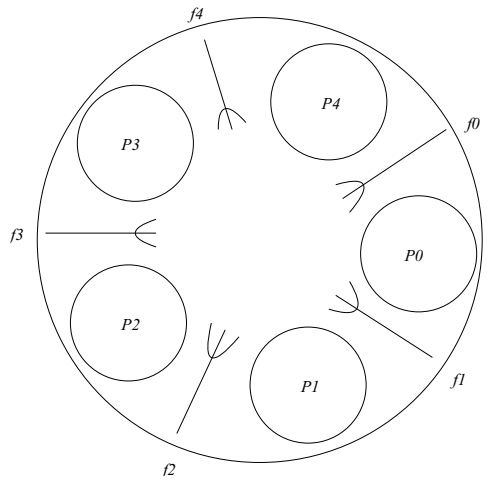
Il s'agit d'un problème factice largement utilisé pour illustrer les problème posés par le partage de ressources en programmation parallèle.

Ce problème est usuellement décrit comme suit.

- Un certain nombre de philosophes sont assis autour d'une table circulaire.
- Chaque philosophe partage son temps entre deux activités : penser et manger.
- Pour penser le philosophe n'a besoin de rien, pour manger il lui faut des couverts.

- Or, la table est dressée de façon très particulière : entre chaque paire d'assiettes ne se trouve qu'une fourchette.
- Un philosophe étant maladroit, il lui faut deux fourchettes pour manger : celle qui se trouve à sa droite et celle qui se trouve à sa gauche.
- Un philosophe ne peut donc manger en même temps qu'un de ses voisins : les fourchettes sont des ressources partagées pour lesquelles les philosophes sont en concurrence.
- Le problème est de régler l'accès à ces ressources partagées pour que tout se passe harmonieusement.

Le problème des Philosophes : illustration



Le problème des Philosophes : une première solution

- Cette première solution utilise un sémaphore pour modéliser chaque fourchette.
- Prendre une fourchette est alors réalisé par une opération `wait`, ce qui à pour effet de mettre le processus en attente si la fourchette n'est pas disponible.
- Libérer une fourchette se fait naturellement par une opération `signal`.


```
/* Définitions et initialisations globales*/
#define N = ? /* nombre de philosophes */
semaphore fork[N]; /* sémaphores correspondant
                      aux fourchettes */
int j; for (j=0, j < N, j++) fork[j]=1;
```

Chaque philosophe (`0` à `N-1`) correspond à un processus exécutant la procédure suivante, où `i` est le numéro du philosophe.

```
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]); wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]); signal(fork[(i+1)%N]);

  }
}
```


Le problème des Philosophes : une première solution - le deadlock

- La solution présentée souffre d'un deadlock possible.
- En effet, si chaque philosophe exécute `wait(fork[i])` avant qu'aucun des philosophes n'ait exécuté `wait(fork[(i+1)%N])`, chaque philosophe se retrouve avec une fourchette et est en attente sur la deuxième.
- Le problème est que chaque philosophe doit acquérir deux ressources, et le fait
  1. en deux étapes,
  2. dans un ordre qui peut mener à un blocage, et
  3. sans annulation possible,

Pour éviter les deadlocks, il faut éliminer un de ces trois éléments.

Le problème des Philosophes : une deuxième solution

Dans cette solution, on modifie l'ordre dans lequel le philosophe `N-1` prend ses fourchettes.

```
/* Philosophes 0 à N-2 */
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[i]);
    wait(fork[(i+1)%N]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1)%N]);

  }
}
```

```
/* Philosophe N-1*/
philospher(i)
int i;
{ while(true)
  { think();
    wait(fork[(i+1)%N]);
    wait(fork[i]);
    eat();
    signal(fork[(i+1)%N]);
    signal(fork[i]);
  }
}
```

Le problème des Philosophes : une deuxième solution - deadlock ?

Dans la deuxième solution, il n'y a pas de deadlock possible. Le principe appliqué est le suivant.

- Les ressources partagées sont ordonnées (`0` à `N-1` pour les fourchettes).
- Un processus qui doit acquérir plusieurs ressources les acquiert en respectant l'ordre global sur les ressources.
- Il n'y a alors pas de deadlock possible.
  - En effet, considérons le processus qui possède la ressource utilisée d'ordre le plus élevé.
  - Ce processus a déjà acquis toute les ressources d'ordre inférieur qui pourraient lui être nécessaires et donc ne peut pas être bloqué.

De plus, en supposant les sémaphore équitables, aucun processus ne pourra rester bloqué indéfiniment.

- En effet, si un processus attend une fourchette, l'équité des sémaphores fera qu'il en disposera dès qu'elle sera libérée.
- Pour qu'une fourchette ne soit jamais libérée, il faut que le processus qui la possède soit en attente d'une fourchette d'ordre plus élevé.
- Vu que la chaîne des attentes bloquantes nous fait monter dans l'ordre des fourchettes, elle doit s'interrompre lorsque l'on atteint la fourchette d'ordre le plus élevé. En effet, celle-ci est forcément la dernière acquise par le processus qui la possède et sera donc libérée.

Le problème des Philosophes : une troisième solution

Dans cette solution, l'acquisition des ressources se fait d'un bloc. Pour ce faire, nous utiliserons un moniteur centralisant la gestion des fourchettes.

- Ce moniteur utilise un tableau `f[]`, où l'on conserve le nombre de fourchettes disponibles pour chaque philosophe (0, 1 ou 2).
- Si le philosophe `i`, ne trouve pas deux fourchettes disponibles, il est mis en attente sur une file qui lui est propre `oktoeat[i]`. Cette file est implémentée par un sémaphore simple car elle ne contient jamais plus d'un processus. L'implémentation utilisée sera directe et n'emploiera pas la classe `Waitqueue`.
- Un tableau de booléens `waiting` indique pour chaque philosophe s'il est ou non en attente dans sa file `oktoeat`. A noter que un appel à `semNbWait` sur le sémaphore implémentant une file `oktoeat` ne donne pas la même information que `waiting`, car l'exclusion mutuelle est libérée avant l'appel à `SemWait`.

Le problème des Philosophes : Le moniteur des fourchettes - constructeur

```
public class ForkMonitor
 { private int nb;
   private int urcount, f[];
   private boolean waiting[];
   private Semaphore urgent, mutex, oktoeat[];

   public ForkMonitor(int N)
    { nb = N; urcount = 0;
      f = new int[nb];
      for (int i=0; i<nb ; i++) f[i] = 2;
      urgent = new SemaphoreFIFO(0);
      mutex = new SemaphoreFIFO(1);
      oktoeat = new Semaphore[nb];
      for (int i=0; i<nb ; i++)
        { oktoeat[i] = new Semaphore(0);
          waiting[i] = false;
        }
    }
```

Le problème des Philosophes : Le moniteur des fourchettes - prise des fourchettes

```
public void takeFork(int i)
  { mutex.semWait();
    if (f[i] != 2)
      { waiting[i] = true;
        if (urcount > 0) urgent.semSignal;
            else mutex.semSignal();
         oktoeat[i].semWait();
         waiting[i] = false;
      }
    f[(i+1) % nb]--;
    f[(i-1+nb) % nb]--;
    if (urcount > 0) urgent.semSignal();
      else mutex.semSignal();
  }
```


Le problème des Philosophes : Le moniteur des fourchettes - libération des fourchettes

```
  public void releaseFork(int i)
    { mutex.semWait();
      f[(i+1) % nb]++;
      f[(i-1+nb) % nb]++;
      if ((f[(i+1) % nb] == 2) && (waiting[(i+1) % nb])
        { urcount++;
          oktoeat[(i+1) % nb].semSignal();
          urgent.semWait(); urcount--;
        }
      if ((f[(i-1+nb) % nb] == 2) && (waiting[(i-1+nb) % nb]))
        { urcount++;
          oktoeat[(i-1+nb) % nb].semSignal();
          urgent.semWait(); urcount--;
        }
      if (urcount > 0) urgent.semSignal();
        else mutex.semSignal();
   }
}
```


Le problème des Philosophes : L'utilisation du moniteur des fourchettes

Une fois un objet correspondant aux fourchettes créé,

```
ForkMonitor F = new ForkMonitor(nb);
```

chaque philosophe exécute la procédure suivante, où l'argument est le numéro du philosophe.

```
philosophe(int i)
  { while(true)
      { think();
        F.takeFork(i);
        eat();
        F.releaseFork(i);
       }

  }
```

Dans cette solution, aucun deadlock n'est possible car si tous les processus sont en attente, aucune fourchette n'est utilisée, ce qui est contradictoire. Elle permet toutefois à deux philosophes d'en exclure un troisième.

La mémoire cache

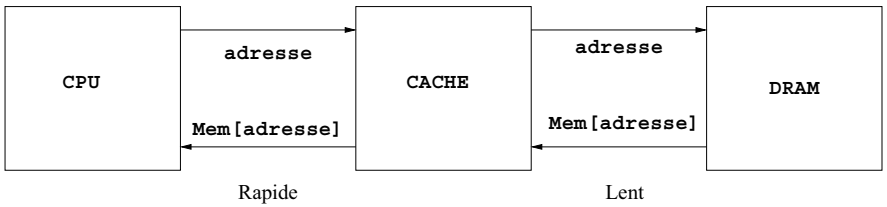
La mémoire cache : principe

- Dans la machine ULg03, l'accès à la mémoire DRAM se fait à la même vitesse que l'accès aux registres.
- Cette condition n'est vérifiée que si l'on ralentit fortement le fonctionnement de l'ensemble de la machine.
- Il y a toutefois une autre possibilité : maintenir dans une petite mémoire rapide une copie d'une partie de la mémoire DRAM. C'est ce qu'on appelle une *mémoire cache* (*cache memory*).
- On peut alors espérer que la plupart des accès se feront dans la mémoire cache et seront rapides. Quand une donnée nécessaire n'est pas dans la mémoire cache, il faut la transférer de la DRAM, ce qui est lent et implique de faire attendre le processeur.

La mémoire cache : caractère associatif

- Une mémoire de masse associe des valeurs à des adresses.
- La mémoire cache doit remplir la même fonction et donc associer des données, non pas à ses propres adresses, mais à des adresses de la mémoire principale.
- La mémoire cache doit donc contenir des paires (adresse, donnée) et permettre de très rapidement trouver une donnée à partir de son adresse.
- On parle donc de *mémoire associative*.
- Lorsque l'on recherche la donnée correspondant à une adresse dans une mémoire cache, elle est soit présente (*cache hit*), soit absente (*cache miss*).

La mémoire cache : illustration du principe

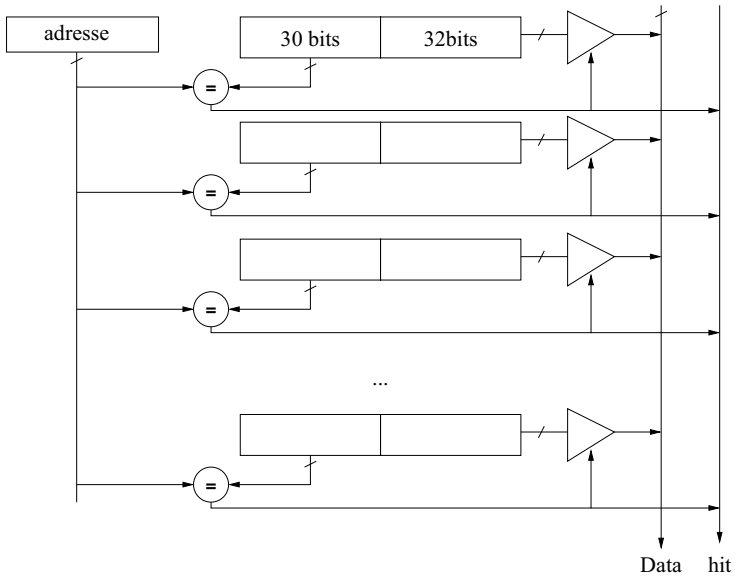


Lorsqu'une donnée recherchée n'est pas dans la cache, le processeur est bloqué en attendant que la donnée soit transférée de la DRAM vers la cache.

Pourquoi les mémoires caches sont elles intéressantes ?

- Si les accès à la RAM dynamique étaient tout à fait aléatoires, une mémoire cache ne donnerait pas de bons résultats. Heureusement, ces accès sont loin d'être aléatoires.
- En effet, on accède souvent aux mêmes adresses à des moments proches. C'est ce qui est appelé la *localité temporelle* (*temporal locality*).
- Lorsqu'on l'on a consulté une adresse donnée, il est fort probable que les adresses consultées suivantes en seront proche. C'est la *localité spatiale* (*spatial locality*).
- Les mémoires caches et leur gestion sont conçues pour exploiter ces deux localités.

La cache totalement associative

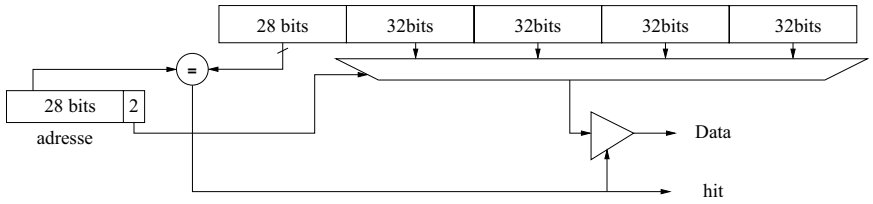


La cache totalement associative : caractéristiques

- Une adresse est associée à chaque mot mémoire conservé dans la cache. La place occupées par les adresses est donc aussi importante que celle occupée par les données.
- Il faut un comparateur par mot conservé ce qui limite la capacité possible de la cache.
- Une bonne politique de remplacement est *LRU* (Least Recently Used), mais l'implémenter est coûteux.
- On peut aussi associer à une adresse plus d'un mot. On parle alors de *caches en blocs* (*blocked caches*).

Caches en blocs

Le principe est le même que pour les caches associatives, mais un bloc de données de taille 2^B mots est associé à chaque adresse de $30 - B$ bits.



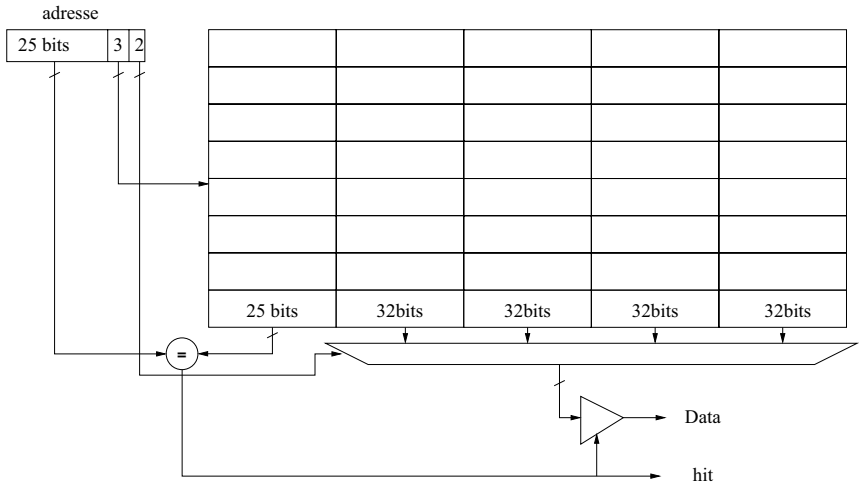
Quand il y a un "cache miss", il faut toutefois transférer un bloc entier de la mémoire, mais la localité spatiale rend cela souvent utile.

Dans la pratique, il y a une taille des blocs qui minimise le nombre de "cache miss".

Les caches à traduction directes

- Les *caches à traduction directe* (*direct mapped cache*) utilisent une partie de l'adresse mémoire comme adresse dans la cache. Le reste de l'adresse est mémorisé et comparé lors d'une recherche.
- Pour chaque adresse mémoire, il y a une seule adresse possible dans la cache. Une politique de remplacement n'est donc pas nécessaire.
- Par contre, il est impossible d'avoir simultanément dans la cache le contenu d'adresses correspondant à la même adresse dans la cache.
- Dans une cache à traduction directe, un seul comparateur en nécessaire. La cache peut être construite avec de la mémoire ordinaire.

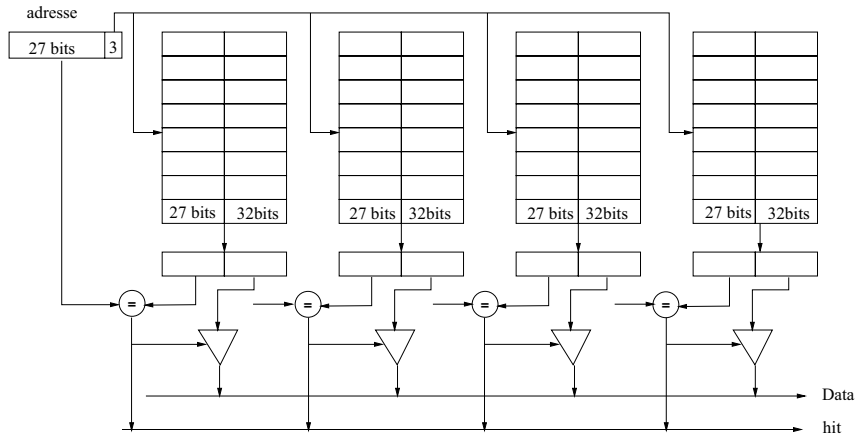
Une cache en blocs à traduction directe



Une solution intermédiaire : la cache associative par ensembles

- Un compromis entre les caches associatives et les caches à traduction directe sont les caches *associatives par ensembles* (*set associative*) qui combinent les deux techniques.
- On a en fait N caches à traduction directe, la sélection entre les caches à traduction directe se faisant comme dans une cache associative.
- Une cache associative par ensemble peut aussi être en blocs.
- D'un point de vue pratique $N = 8$ est souvent suffisant.
- Une politique de remplacement aléatoire donne presque d'aussi bons résultats que LRU pour les caches de taille suffisante.

Une cache associative par ensembles



La mémoire cache et les opérations d'écriture

- La plupart des opérations en mémoire sont des lectures. Que fait-on lors d'une modification ? Faut-il modifier la mémoire immédiatement ? Il y a plusieurs options possibles.
  - *write-through :* modification immédiate de la DRAM avec attente du CPU.
  - *write-back :* modification retardée.
- La modification retardée peut améliorer les performances, mais il faut indiquer pour chaque entrée de la cache si elle doit être écrite en mémoire ou non.
- Dans le cas des multiprocesseurs, le problème de la *cohérence des caches* (*cache coherence*) se pose.

La mémoire cache et la mémoire virtuelle

- Si on utilise la mémoire virtuelle, la cache peut être réalisée en termes d'adresses physiques ou d'adresses virtuelles.

- Une cache fonctionnant avec des adresses virtuelles évite de faire la traduction d'adresse tant que l'on reste dans la cache, mais nécessite que la cache soit invalidée à chaque changement de contexte.

- Une cache virtuelle pose problème si plusieurs adresses virtuelles peuvent correspondre à la même adresse physique (partage de données par un mécanisme d'alias).