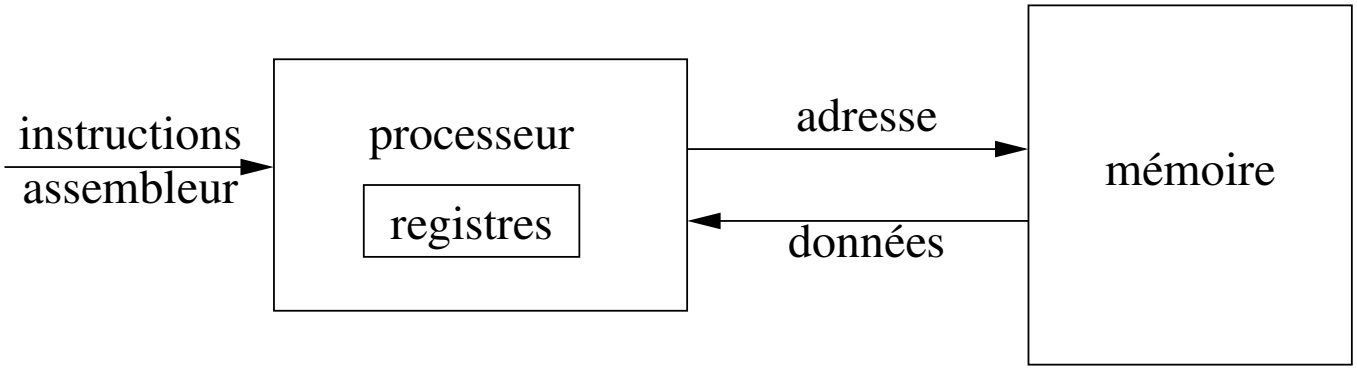
Architecture et Système

Stefan Schwoon

Cours L3, 2019/20, ENS Cachan

Vue abstraite d'un ordinateur



Ceci est la machine abstraite proposée au programmeur.

Jusqu'aux années 70/80 : réalisation plus au moins directe de cette architecture.

Depuis : gestion de plus en plus complexe derrière les scènes
(micro-architecture)

Gestion mémoire/processeur avancée

Besoins d'un système d'exploitation :

- partage du processeur entre différents processus/utilisateurs
- protection et partage mémoire
- gestion de privilèges (mode noyau/mode utilisateur)

Optimisations :

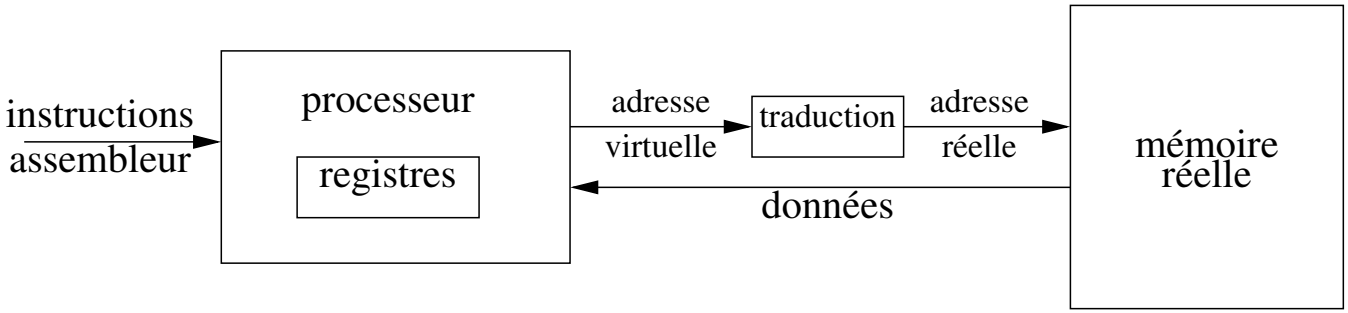
- parallélisation (pour accélérer le calcul)
- mémoire rapide (cache)
- réordonnancement d'instructions / prediction de branches
- réduction la consommation énergétique

Remarques :

Les détails techniques varient pas mal entre les différents ordinateurs/systèmes.

Quelques détails de l'exposition suivante sont valables pour un processeur Intel i7 et les versions courantes de Linux, mais les principes sont applicables à tous les systèmes courants.

Mémoire virtuelle



Les adresses fournies par les instructions assembleur sont des adresses *virtuelles*.

Une unité dédiée (Memory management unit, MMU) les traduit vers des adresses physiques.

La traduction est telle qu'un processus n'a accès qu'à la mémoire qui lui est propre (au moins, en théorie. . .).

Traduction mémoire virtuelle $\leftrightarrow$ réelle

Exemple : traduction sur un processeur Intel 64-bit

Espace virtuel :

adressage avec 48 bits (donc théroriquement 256 Tera)

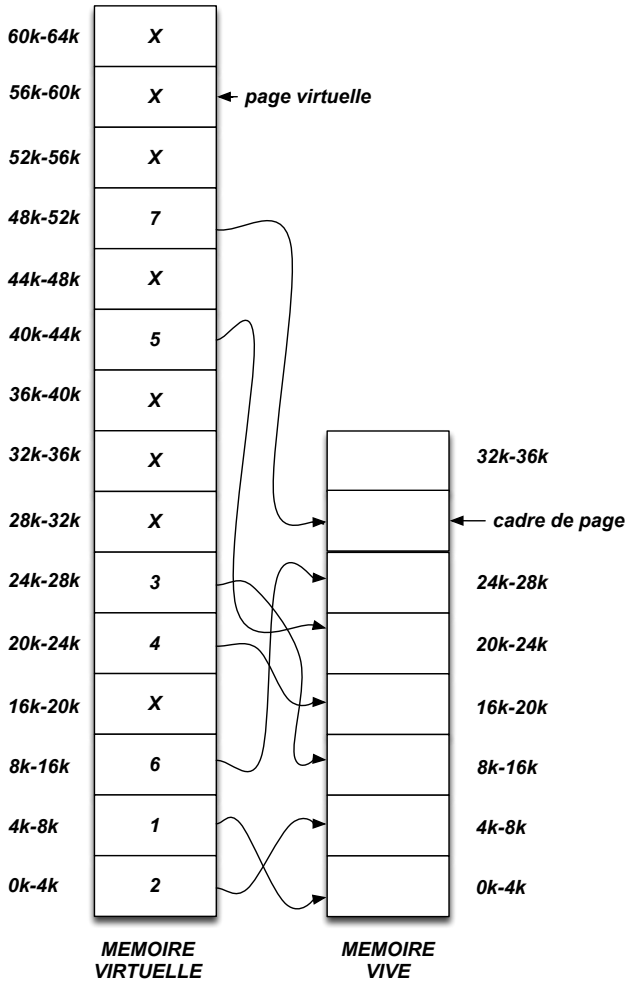
découpé en *pages* (bloc de mémoire contiguë)

taille d'une page : 1 GB, 2 MB, 4KB, selon le cas

une adresse se découpe donc en:
– numéro de page (les bits les plus significatifs) – écart/offset (les bits les moins significatifs)

Visualisation

Une mémoire virtuelle traduit donc le numéro d'une page virtuelle vers une page réelle, en gardant l'écart:



Propriétés de la traduction:

partielle, injective

réalisée à l'aide d'un *tableau de pages* propre à chaque processus, créé et maintenu par le noyau

Un registre dédié (CR3) contient le pointer vers le tableau du processus actuel. Lorsque le système bascule entre deux processus, c'est ce registre qui change de valeur.

Remarques

Lors de l'exécution du processus, les accès mémoire se font indépendamment du noyau, le matériel s'en occupe.

Certains appel système (p.ex. `malloc`, `brk`) modifient le tableau de traduction.

Traitement d'un accès mémoire illégal :

- le processeur déclenche une interruption ;
- le système rattrape cette exception et envoye un signal (`SIGSEV`) au processus ;
- le signal termine le processus (sauf s'il le rattrape).

Détails de la pagination

Traduction réalisée à l'aide d'un *trie* (tableau à plusieurs niveaux).

Une partie des pages réelles (de taille 4 KB) est utilisée pour stocker les tableaux.

> 4 KB contiennent $512 = 2^9$ adresses à 64 bits.
>
> Dans le tableau de premier niveau on utilise les 9 bits les plus significatifs pour obtenir l'adresse d'un bloc de niveau 2.
>
> Après 4 niveaux ($4 \times 9 = 36$) on obtient l'adresse d'un bloc de 4 KB ($4096 = 2^{12}$).

Les 16 bits non-utilisés stockent des informations sur les *priviléges*:

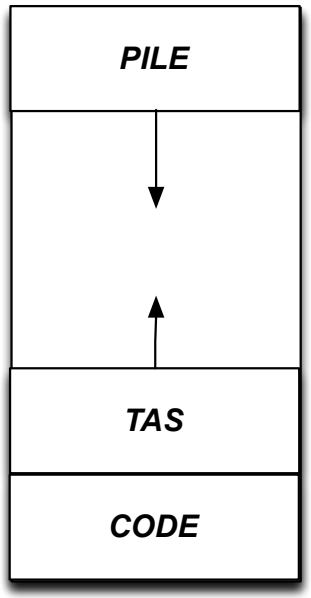
> accès en écriture, page exécutable, accès privilégié (en mode noyau seulement), ...

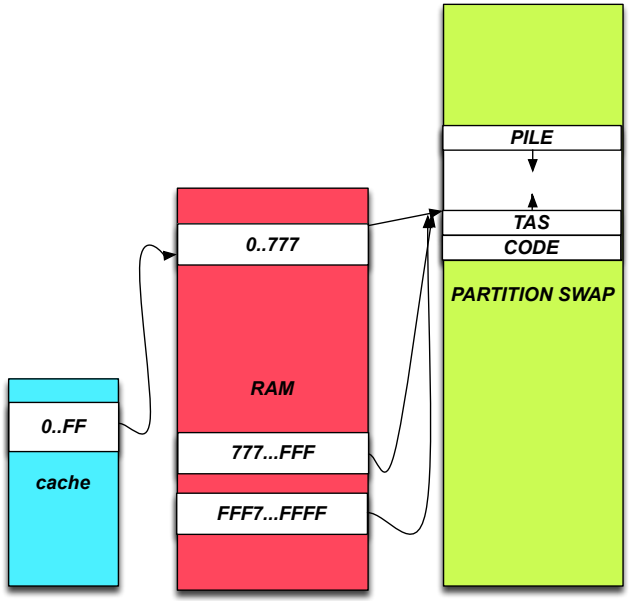
Organisation de la mémoire virtuelle

Sous Linux, un processus vit dans la moitié basse de la mémoire virtuelle (bit 47 = 0).

La partie haute est reservé au noyau.

Organisation en code, tas et pile:





Gestion du tas

Une grande partie de l'espace virtuelle ne correspond à aucune page réelle (et les accès à ces adresses déclenchent une violation de segment).

Un processus peut élargir la taille de son tas avec `brk`.

`malloc` / `free` : Fonctions en mode utilisateur qui organisent le tas (font appel à `brk` si nécessaire).

Ces fonctions utilisent un partie du tas pour organiser les allocations.

Les bogues (débordement de mémoire) peuvent corrompre cette information, ménant à des effets secondaires non prévisibles.

Translation lookaside buffer (TLB)

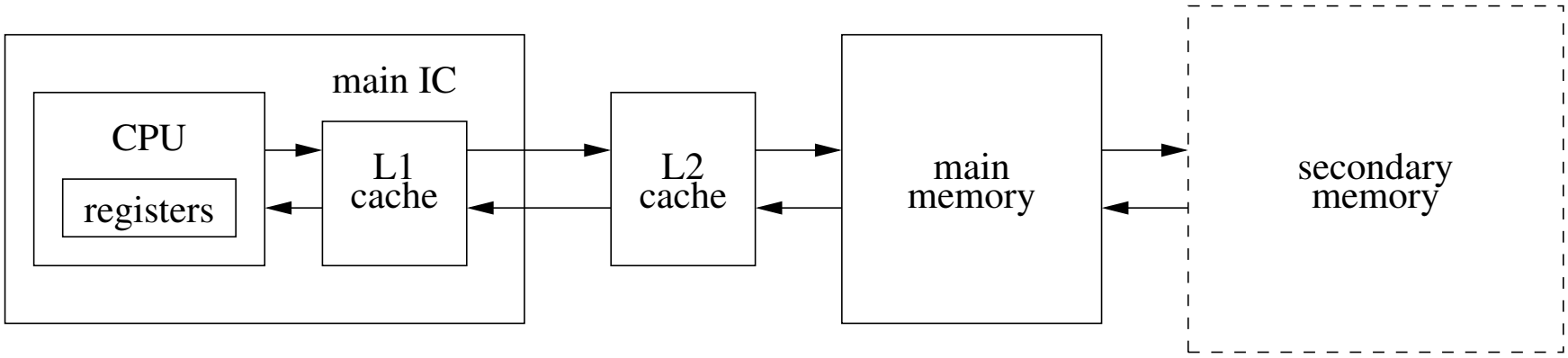
Avec le principe évoqué précédemment, chaque accès mémoire virtuel se traduit en cinq accès réels $\rightarrow$ inefficace

Du coup, on mémorise les pages utilisées le plus souvent dans le TLB (mémoire associative mais limitée).

Attention, une même adresse virtuelle correspond à plusieurs adresses réelles, en fonction du processus:

> Certains processeurs récents offrent un régistre stockant l'identifiant du processus actuel, pris en compte par le TLB.
>
> Dans d'autres processuers, le TLB doit être invalidé quand on bascule vers un autre processus.

Les caches



Idée: garder les données utilisées souvents dans une mémoire spéciale rapide, mais de taille limitée

Facteurs limitants : coût, espace physique limité

Cache à plusieurs niveaux, les cache distants sont plus lents mais aussi plus grands

Transparent : Le programmeur accède à une adresse virtuelle, le contenu se trouve dans la mémoire principale ou dans l'un des caches.

Une adresse dans le cache L1 est fourni trés rapidement (1 ou 2 cycles).

Pour récupérer une adresse dans la mémoire principale, il faut une centaine de cycles !

Les accès à la mémoire secondaire (va-et-vient, swap space) nécessitent l'intervention du système (par le biais d'une interruption, similaire au mécanisme SIGSEGV, mais transparent pour le processus concerné).

Réalisation d'un cache

Grosso modo, le cache fonctionne comme une petite mémoire virtuelle :

étant donné une adresse du mémoire principale (qui elle est très grande), il renvoie une adresse dans une espace plus limité (celui du cache);

si une adresse demandé n'est pas dans le cache, il faut la récupérer et remplacer une adresse qui reside actuellement dans le cache.

Functionnement en pratique : découpage de la mémoire en *cache lines* (similaire aux pages, mais de taille moins importante, p.ex. 64 octets pour le L1 au i7).

Correspondence entre mémoire et cache gérée par une *mémoire associatif*.

Réordonnancement d'instructions

Un processeur se trouvent face à des instructions de durée variable :

raisons inhérentes (instructions complexes vs simples)

délais dus aux accès mémoire (et périphérique)

Pour accélérer le calcul et mieux utiliser les résources, les processeurs font de nombreuses optimisations derrière les scènes :

(découpage d'instructions d'assembleur en micro-instructions)

réordonnancement d'instructions considérées comme indépendantes (read before write dans l'Intel, voir l'Algorithme de Peterson)

repartition entre plusieurs unités d'exécution en parallèle

Exécution spéculative

Supposons que le processeur doit exécuter un branchement conditionnel, alors que le résultat de la condition n'est pas encore connue (p.ex. en raison des accès mémoire),

Il peut donc commencer à exécuter l'un des branches (ou les deux, en fonctions des ressources disponibles) jusqu'à ce que la condition a été évaluée. Les résultats d'un mauvaise branche sera invalidée par la suite.

Pour savoir quelle branche devrait être exécutée, le processeur possède une unité pour la *prédiction des branches*.