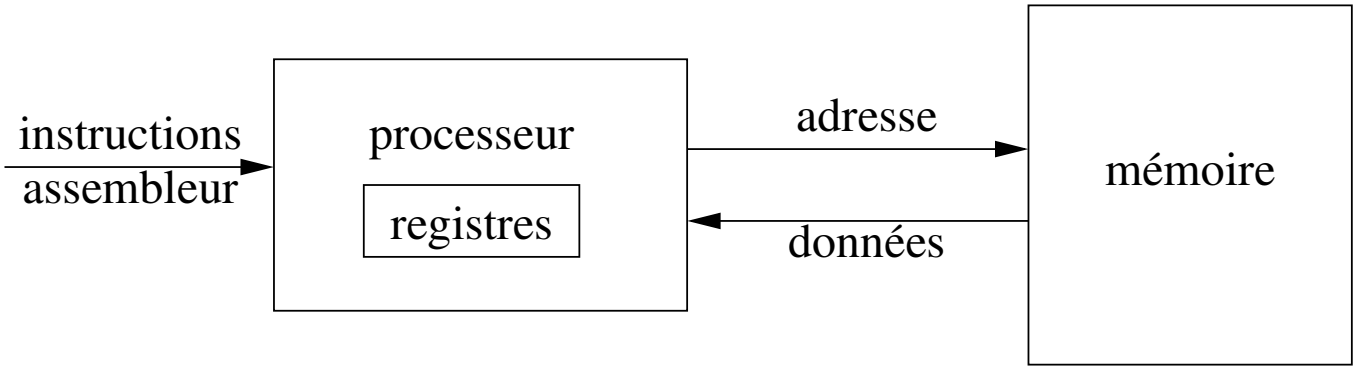
Architecture et Système

Stefan Schwoon

Cours L3, 2024/2025, ENS Paris-Saclay

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
(mémoire virtuelle, micro-architecture)

Gestion mémoire/processeur avancée

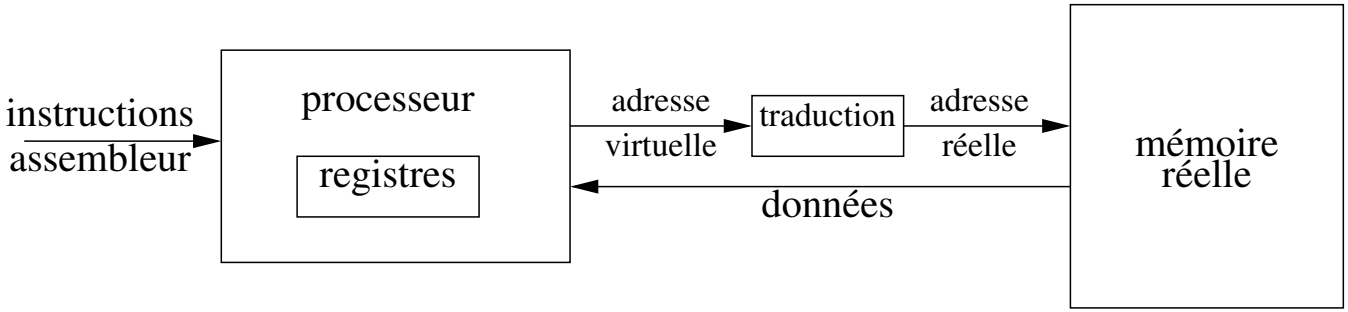
Besoins d'un système d'exploitation :

- partage du processeur entre différents tâches/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 de la consommation énergétique

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.

Du coup, un processus n'a accès qu'à la mémoire qui lui appartient.

Traduction mémoire virtuelle $\leftrightarrow$ réelle

Exemple : traduction sur un processeur Intel 64-bit

Espace virtuel :

adressage avec 48 bits (donc théroriquement 256 Tera)

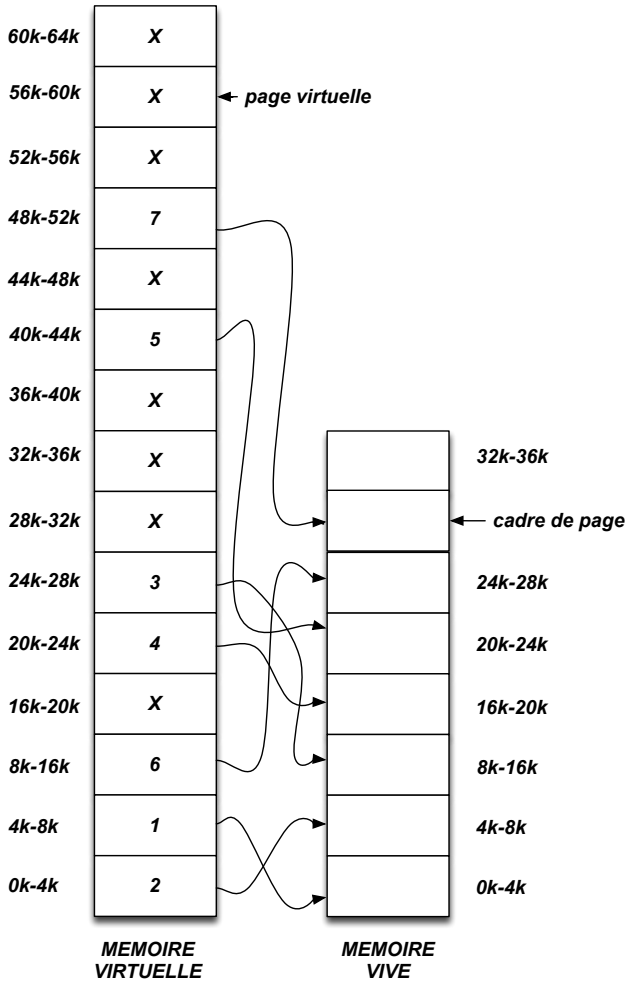
découpé en *pages* (bloc de mémoire contiguë)

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 (p.ex. CR3 dans les Intel i7) 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 appels système (p.ex. `malloc`, `brk`) modifient le tableau de traduction.

Traitement d'un accès mémoire illégal (à une adresse non affectée) :

- le processeur déclenche une interruption ;
- le système rattrape cette exception et envoye un signal (`SIGSEGV`) 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 un premier temps, on découpe l'espace virtuel en $512 = 2^9$ blocs de 512 GB ($= 2^{39}$ octets).

Un premier tableau est alors utilisé pour indiquer si un tel bloc est soit entièrement inutilisé, soit l'adresse d'une autre page qui décrit son découpage.

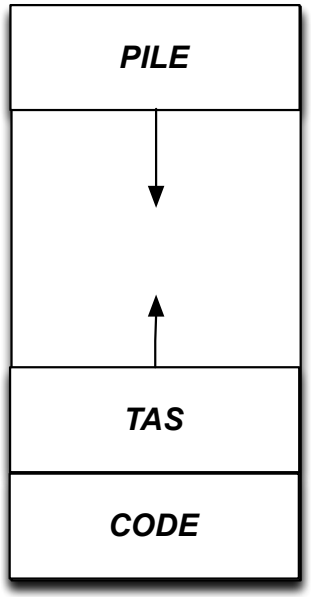
Tout bloc de 512 GB est découpé en 512 blocs de 1 GB ($= 2^{30}$ octets), et de suite. Après 4 niveaux ($4 \times 9 = 36$) on obtient l'adresse d'un bloc de 4 KB ($4096 = 2^{12}$).

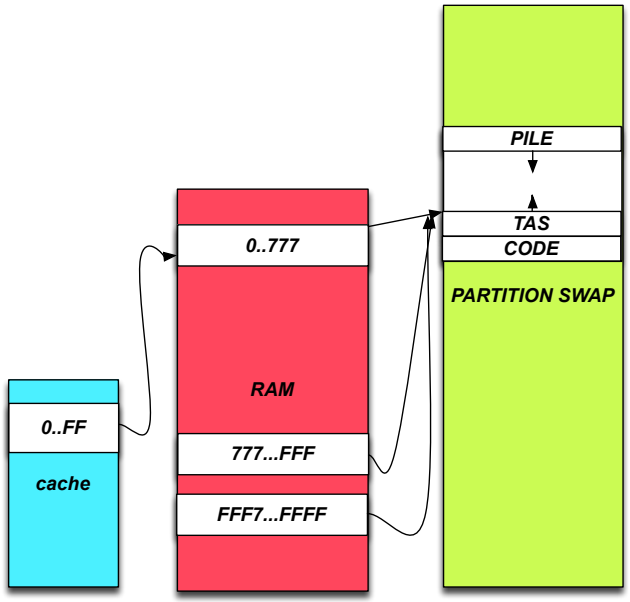
Organisation de la mémoire virtuelle

Les 16 bits non-utilisés stockent des informations sur les *permissions*: accès en écriture, page exécutable, accès privilégié (en mode noyau seulement), . . .

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

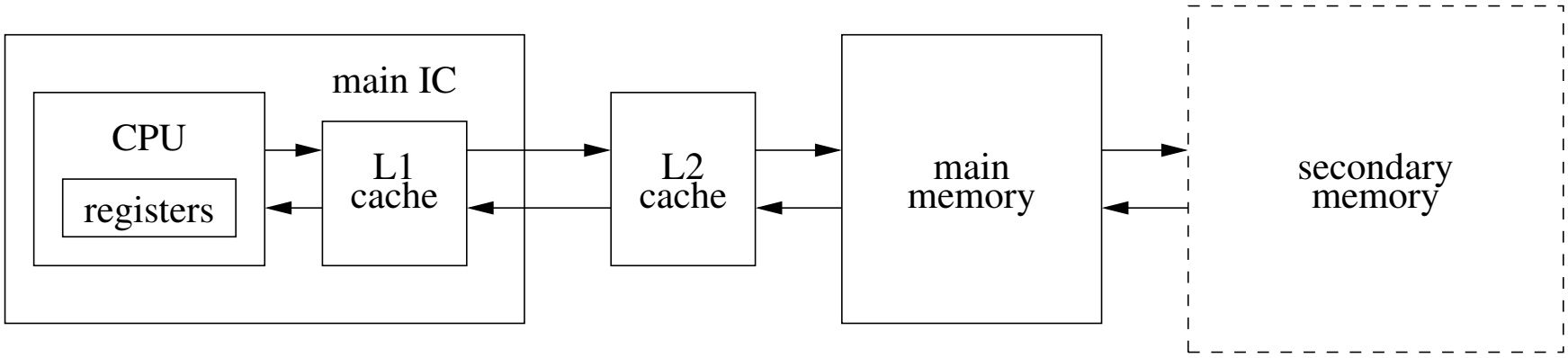
Du coup, on mémorise les pages utilisées le plus souvent dans le TLB (mémoire associative mais limitée).

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

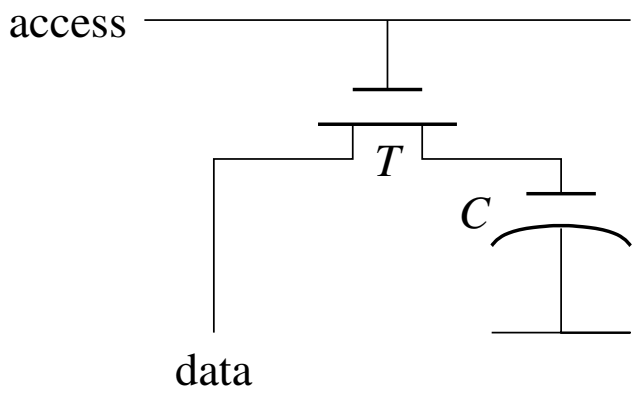
Facteurs limitants : coût, espace physique limité

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.

DRAM vs SRAM

DRAM (*dynamic random-access memory*)



réalisation compacte (p.ex. un seul transistor + capaciteur)

interprétation: capaciteur chargé = “1”

lecture déstructive, fuite dans les capaciteurs $\rightarrow$ récharge périodique

compact mais lent, typiquement utilisé pour mémoire principale

DRAM vs SRAM

SRAM (*static random-access memory*)

p.ex. réalisation par une bascule D

moins compact mais plus rapide, utilisé pour les caches

p.ex. L1 = intégré dans le CPU, L2 = dans un autre chip

Une adresse dans le cache L1 est fournie trés rapidement ($\sim$ 4 cycles).

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

Mémoire secondaire

(aussi appellée va-et-vient, *swap space*)

zones de mémoire virtuelles stockées (temporairement) sur disque dur

mécanisme spécial de pagination:

- une page virtuelle peut être marquée comme “residant en mémoire virtuelle”
- lors d’un accès à une telle page, le processeur déclenche une interruption
- l’interruption est captée par le noyau qui libère une page mémoire réelle pour y stocker la page virtuelle, puis change la table de pagination
- le processeur résume son travail et délivre le résultat
- transparent pour le processus, mais très lent

Réalisation d'un cache

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

on découpe la mémoire en petites pages (appellées *cache line*),
p.ex. 64 octets pour le L1 au i7

le cache stocke un nombre limité de lignes

lors d'un accès mémoire, on teste si la ligne de cache est stocké au cache (*)

si oui, on utilise le cache

sinon, on récupère la ligne depuis la mémoire principale et on vire une autre ligne du cache.

(*) test par *mémoire associatif*, similaire au TLB

Algorithmes de cache

On suppose qu'il existe un nombre fixe de créneaux dans le cache.

S'il intervient un accès mémoire dans une page du cache, on renvoie le résultat immédiatement.

Sinon, on récupère la page depuis une couche de mémoire plus lente. On sélectionne un créneau dans le cache que l'on juge le moins utile et remplace la page dedans. (page fault).

Politique de sélection FIFO : remplacer la page la plus ancienne.
Cette idée souffre de anomalie de Belady: une augmentation du cache peut augmenter le nombre de fautes de page !

Exemple: séquence 3 2 1 0 3 2 4 3 2 1 0 4 avec trois ou quatre créneaux

Seconde chance / Horloge

Seconde chance: Modification /improvement de FIFO: On gère un bit de ”référence” dans chaque créneau qui est mis à 1 quand sa page est utilisée.

Lors d’une faute de page : éliminer le créneau le plus ancien si son bit est 0. Sinon, on le met à zéro et déplace le créneau à la fin de la liste.

Si tous les créneaux ont été utilisés, on élimine le plus ancien après tout. thrown out.

Algo d’horloge: Implémentation de la seconde chance avec une liste circulaire.

LRU / NRU

Least recently used: On garde le temps du dernier accès pour tout créneau, lors d'une faute de page on élimine celle qui n'a pas été référencée le plus longtemps.

On obtient certains résultats d'optimalité, mais cher algorithmiquement.

Not recently used: Gérer deux bits par créneau (référencé/modifié); le bit de référence est remis à 0 périodiquement.

Quatre classes de pages: 00 = non référencé/non modifié, 01 = non référencé mais modifié etc; lors d'une faute de page on choisit un créneau de la classe minimale disponible.

Compromis entre optimalité et performance (par rapport à LRU).