Mémoire 2
TLB - Protection - Allocation - Memory Mapping

Programmation Système - R3.05

C. Raïevsky

IUT A Valence
Université Grenoble Alpes

Département Informatique

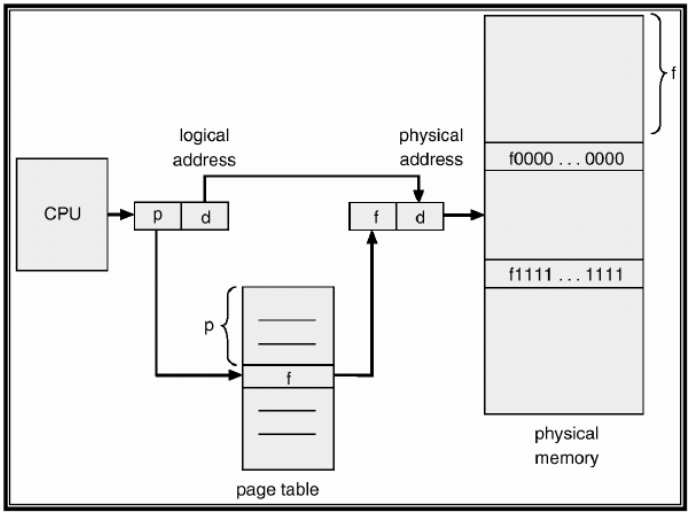
BUT Informatiques 2ème année

2023

Plan

Table des pages – Rappel

Représentation simpliste



Source : courses.teresco.org/cs432_f02/lectures/11-memory/11-memory.html

Translation Lookaside Buffer – TLB

Problème :

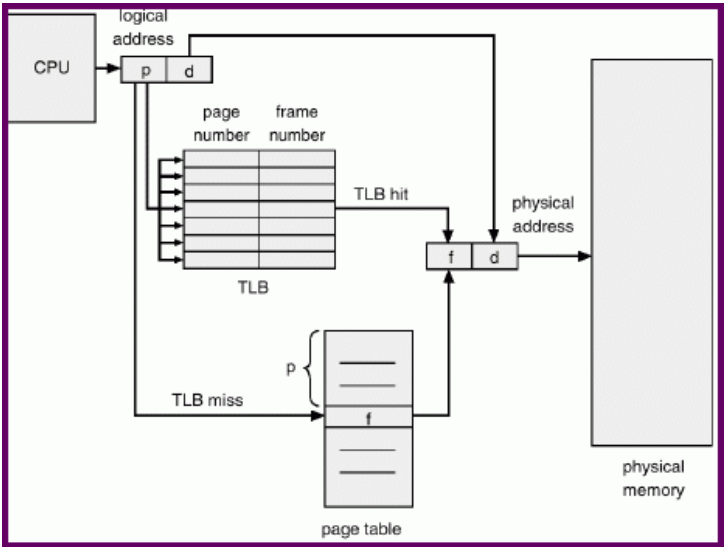
Chaque accès mémoire ⇒ un accès à la table des pages

- Or la table des pages est en mémoire
- On double le nombre d'accès mémoire ⇒ très coûteux

Solution : *Translation Lookaside Buffer – TLB*

- Cache stockant les accès récent à la table des pages
- Accélère drastiquement les accès mémoire consécutifs aux pages chargées, ce qui est fréquent
- Cache potentiellement fourni par le matériel (donc ultra rapide)

Translation Lookaside Buffer – TLB



Et le noyau ?

Le noyau

- Est-ce que le noyau utilise des adresses virtuelles ?

Et le noyau ?

Le noyau

- Est-ce que le noyau utilise des adresses virtuelles ?

- Le noyau s'exécute en mode d'adressage physique
- Une plage d'adresse lui est réservée
- Il ne passe pas par une table de pages

Adressage Mémoire

Caractéristiques Générales

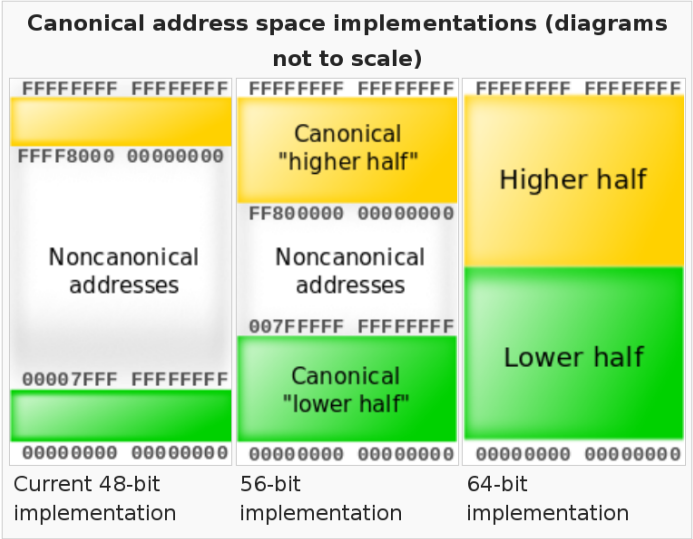
Taille typique des pages : 4Ko

Architecture	défaut	"Huge Pages"
x86_64	4Ko	2Mo ou 1Go
ARM	4Ko	1Mo ou 16Mo
32-bit x86	4Ko	4Mo
Sparc	8Ko	64Ko, 4Mo, 256Mo, 2Go

Taille de l'espace d'adressage virtuel

- Systèmes 32bits : maximum 4Go
- Systèmes 64bits :
  - Maximum théorique : 18446744073709551616 octets
    $\rightsquigarrow$ 1 milliard de Go
  - Implémentation actuelle (x86_64, 48bits) :
    140737488355328 octets $\rightsquigarrow$ 256000 Go
  - Ça devrait suffire pour un moment...

Espace d'adressage virtuel pour l'architecture x86_64



Protection - Exemple

Exemple

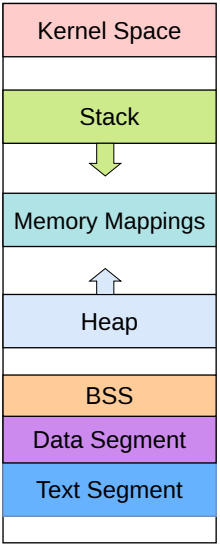
- Pourquoi est-ce que le code ci-dessous provoque une *segmentation fault* ?

```
int* pi = NULL;
*pi = 42;
```

Protection - Sécurité

Accès restreint à :

- Text segment (ro)
- Data segment
- BSS segment
- Heap (Tas, free-store)
- Stack (Pile)
- Memory Mapping segment



Protection - Sécurité

Un processus ne peut accéder qu'à certaine parties de son espace d'adressage

- Text segment (ro)
- Data segment
- BSS segment
- Heap (Tas, free-store)
- Stack (Pile)
- Memory Mapping segment

Tout accès en dehors de ces zones ⇒ **Segmentation Fault**

Une exception : appel de fonction → agrandissement de la pile

Cohérence assurée par les tables des pages

- Une page physique n'est associée qu'à un processus à la fois
- Exception : possibilité de partager explicitement des zones

Outils pour examiner la mémoire d'un processus

Avant lancement du processus :

- nm
- objdump
- readelf

Une fois le processus lancé :

- pmap
- /proc/n°process
- ps pstree

Démo :

`hello.c`

Allocation dynamique de mémoire (`malloc`)

Allocation et libération d'espaces mémoires

- Séquence non connue à la compilation
- Il existe toujours une séquence qui met en défaut l'algorithme d'allocation

Exercice : bloc mémoire libre de 100 unités, séquence d'allocations suivante :

- 5 allocations de 20 unités
- Libération des deuxième et quatrième blocs alloués
- Allocation d'un bloc de 30 unités

Fragmentation

Comme pour le chargement de pages en mémoire physique

- Impact significatif sur les performances
- Plusieurs algorithmes
- Pas de solution parfaite dans tous les cas

Algorithmes d'allocation dynamique de mémoire

Principaux algorithmes

- *First Fit*
- *Best Fit*
- *Worst Fit*

Sous linux : *Buddy* + *Slab*

Allocation dynamique de mémoire : *Buddy* (copain)

Fonctionnement

Représentation des blocs libres :

- N listes de blocs libres
- Chaque liste comporte des blocs de taille 2^N

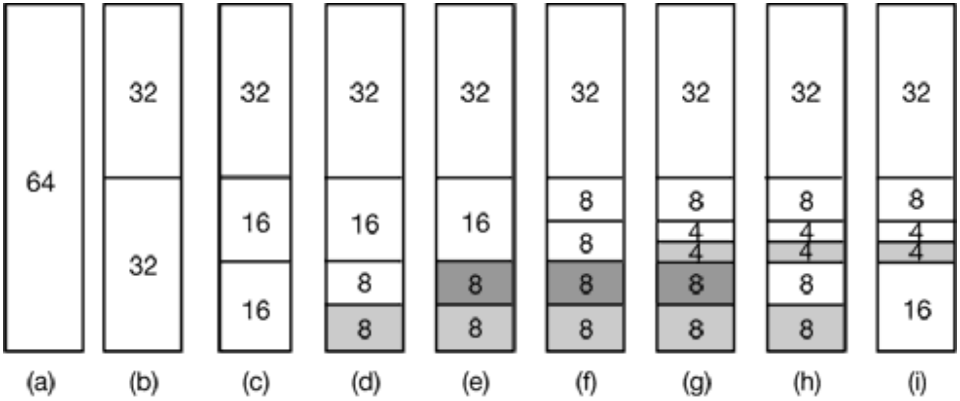
Allocation de k unités :

- Recherche du premier bloc libre de taille $\geq$ k
- Tant que le bloc choisi peu être coupé en deux et rester $\geq$ k
  - Découpage du bloc en deux "*buddies*" de même taille
- Renvoi du bloc choisi

Libération :

- On marque le bloc correspondant comme libre
- Si on a un "*buddy*" contigu, on fusionne

Allocation dynamique de mémoire : *Buddy*



Séquence d'allocations – libérations :

$8 \rightarrow 8 \rightarrow 4$

deuxième bloc de 8 libéré

Avantages et inconvénients de l'algorithme *buddy*

Avantages

- Tend à garder la mémoire physique libre par bloc
- Allocation et libération rapide (pas de parcours longs)
- Adapté aux requêtes de taille 2^N
  - Ce qui est souvent le cas (pages mémoire notamment)

Inconvénients

- Forte fragmentation interne
  - Une demande de 65 unités cause l'allocation de 128...

Solution : *slabs*

Slab allocation

- Mécanisme ajouté par dessus le *buddy allocator*
- Gestion d'un pool de "*slab*" (bloc)
- Chaque *slab* a une taille fréquemment utilisée dans le noyau

Les caches mémoire

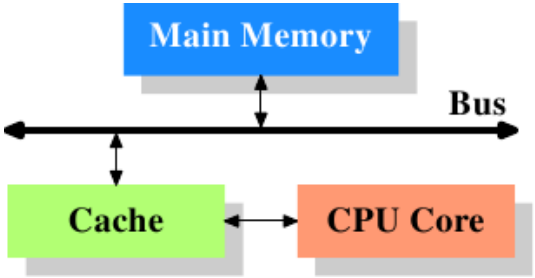
Logiciels & Matériels

Deux principaux types de cache pour la gestion de la mémoire :

- Cache de pages → Noyau
- Caches matériels (L1, L2, L3) → Intégrés au CPU

Caches matériels – Principe Général

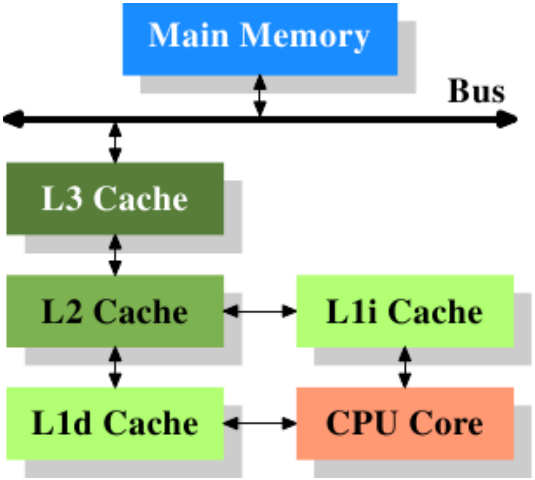
- Tout accès à la mémoire centrale passe par un cache.
- Une donnée est tout d'abord recherchée dans le cache.
- Si elle n'y est pas, elle est chargée depuis la mémoire centrale dans le cache.



Ulrich Drepper "What every programmer should know about memory"
lwn.net/Articles/252125/

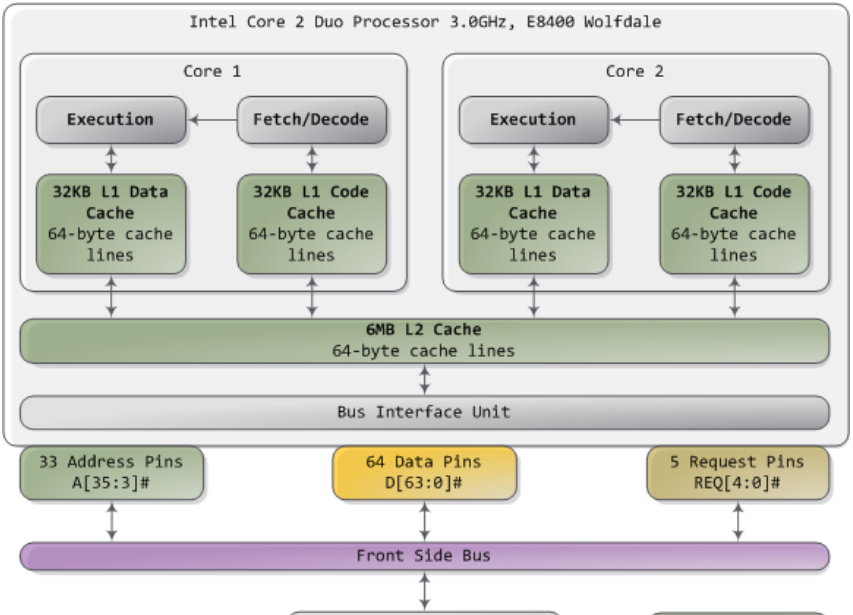
Caches matériels – Hiérarchie

Il existe une **hiérarchie** de caches matériels



Certains sont dédiés aux instructions

Caches matériels – Exemple (simplifié)



Cache de pages – *Page Cache*

Ensemble de pages gardées en mémoire

- Gérées par l'OS
- Pour optimiser les accès successifs
  - En lecture et écriture
  - Par le même processus ou pas

Toutes les opérations d'entrée-sortie sur des fichiers passent par le cache de page du noyau.

Cache de pages – *Page Cache*

Lors d'une écriture

- L'opération est réalisée sur la page du cache
- propagée au support
  - à intervalle régulier
  - par un thread dédié à chaque support
  - ou lors d'un `sync`

Outils

- `free -m`
- `/proc/meminfo` ("Dirty" notamment)

Comportement lors d'un `read`

```
ssize_t read(int fd, void *buf, size_t count);
```

Séquence lors d'un `read` dans un fichier

- Requête par le programme d'une portion de fichier
- Pages correspondant au fichier dans le cache de pages ?
- Non $\rightarrow$ Chargement d'une page du fichier vers le cache de pages
- Copie d'une portion de la page chargée vers le buffer utilisateur

Après cette séquence :

Deux copies de la portion du fichier sont en mémoire physique !!

Solution : *Memory Mapped Files*

Memory Mapped Files

Mise en correspondance d'un espace mémoire avec le contenu d'un fichier

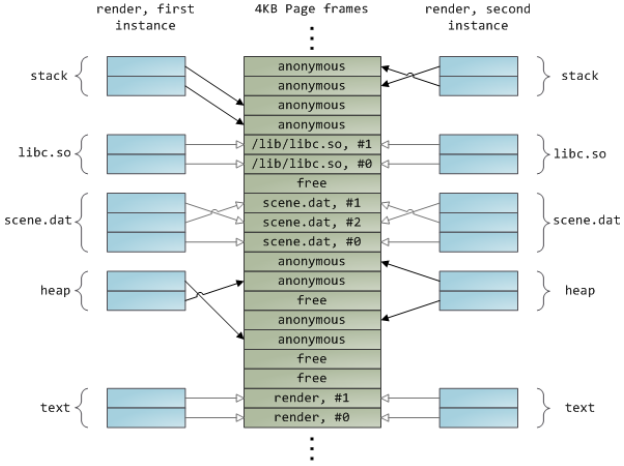
Remplace les opérations de *read* et *write*

- Le contenu du fichier est directement accessible
- Pas de copie dans un tampon
- Chargement uniquement des pages effectivement accédées
- Utilisation des mécanismes du noyau :
  - défaut de page
  - remplacement de pages

Utilisation transparente et efficace du **cache de pages**

Exemple

Deux instances d'un programme ("render") qui travaillent avec un même fichier ("scene.dat").



Memory Mapped Files – Remarques

- Très utilisé pour les librairies
- De plus en plus pertinent avec un espace d'adressage virtuel de 64bit
- Peuvent être partagés $\rightarrow$ IPC

Pour aller plus loin

- Gestion de la mémoire par le noyau : Gustavo Duarte's blog "*how-the-kernel-manages-your-memory*"
- Allocation dynamique mémoire :
  - www.ibm.com/developerworks/library/l-memory/
  - Transparents de l'université Columbia
- Cache de pages : Gustavo Duarte's blog "*page-cache-the-affair-between-memory-and-files*"
- Document très complet sur la mémoire : "*What every programmer should know about memory*", Ulrich Drepper
  - Part1
  - Part2