Programmation système

Ressource R3.05 - Mémoire

monnerat@u-pec.fr

IUT de Fontainebleau

Sommaire

Principes

Principes

Introduction

Types de mémoire

Type	Accès	Vitesse	Persistance	Utilisation
Registre	L/E	*****	non	temporaire
SRAM	L/E	****	non	buffer
DRAM	L/E	***	non	mémoire centrale
ROM	L	**	oui	mémoire de démarrage (BIOS)
SWAP	L/E	*	oui	mémoire virtuelle

Idéalement :

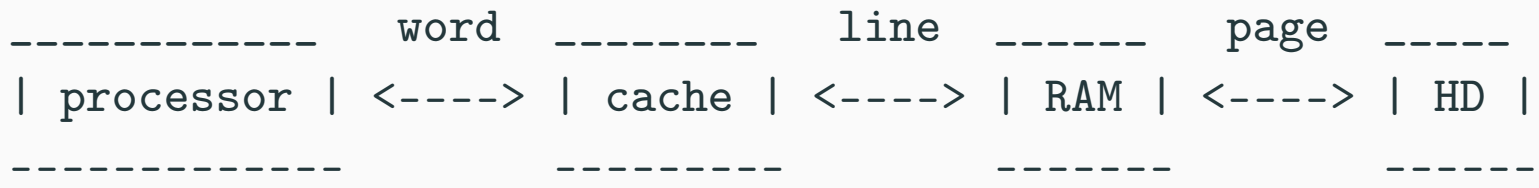
- Rapide,
- Bon marché,
- Grande capacité.

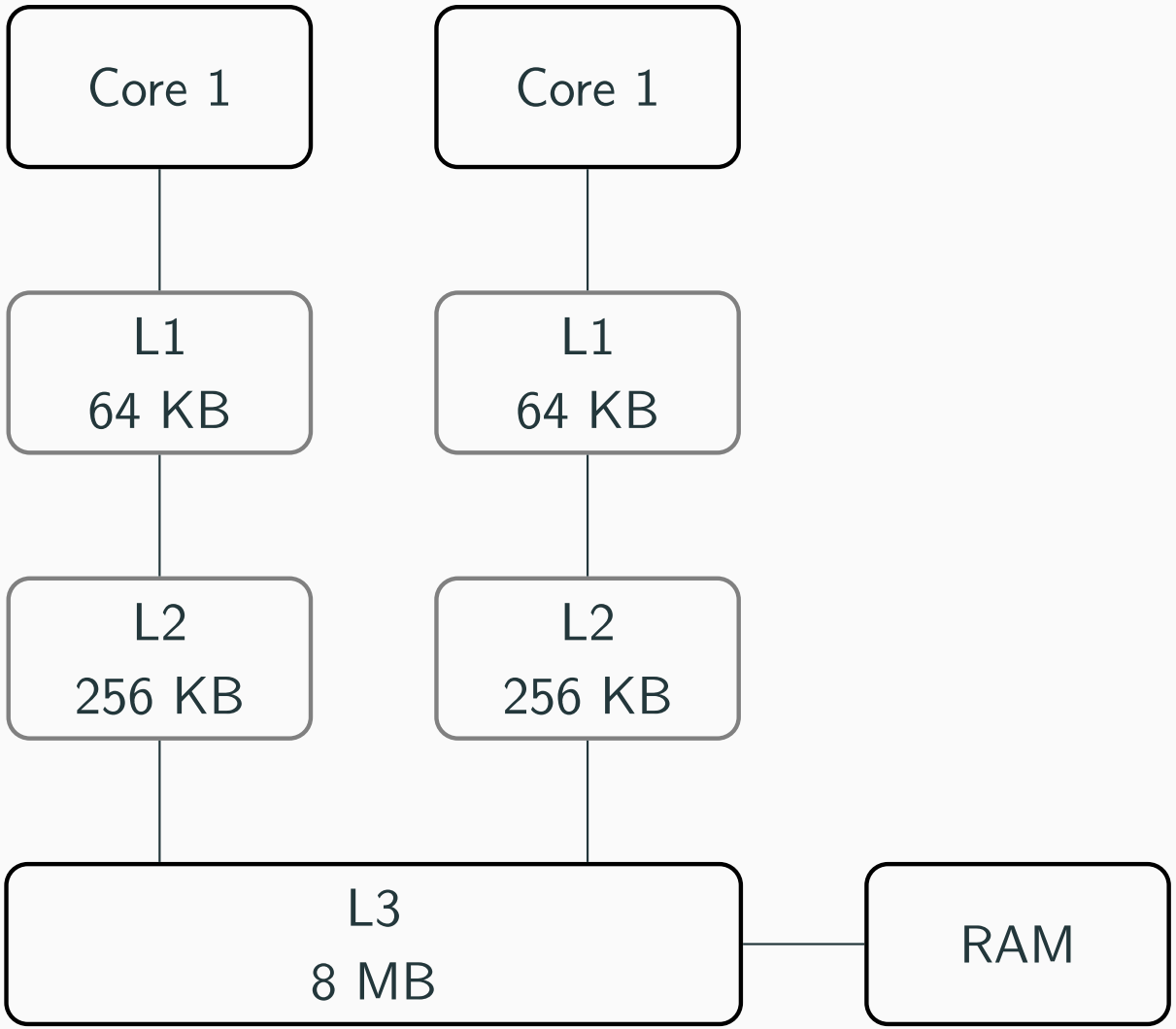
Ça n'existe pas !

Hiérarchie

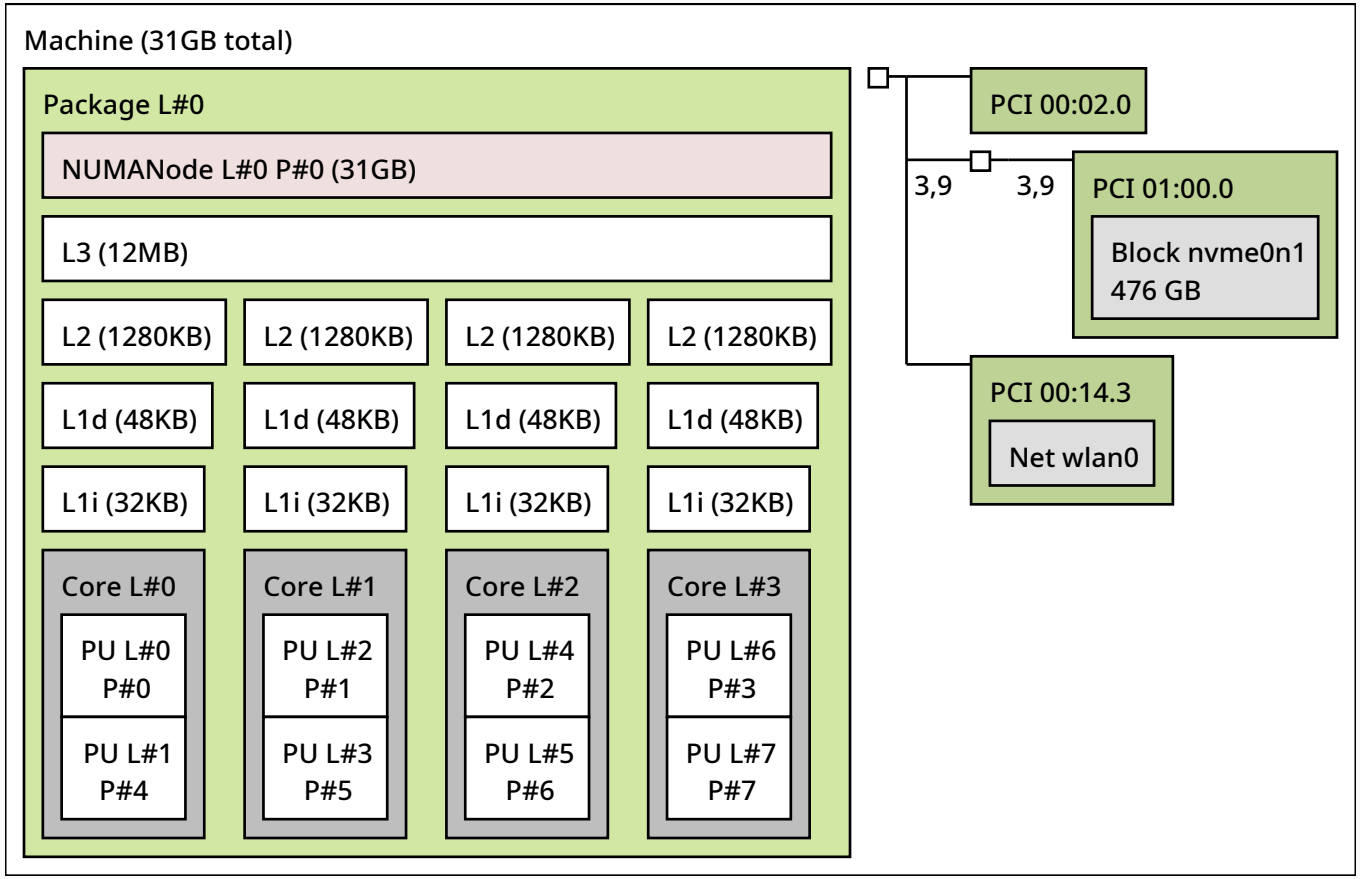
- registres du processeur,
- mémoire cache
- mémoire principale (main memory, RAM),
- mémoire de masse (disk memory,HD).

Les échanges entre ces éléments se font par blocs d'octets qui portent les noms suivants (dans la terminologie conventionnelle) :




Core 1
Core 1
L1
64 KB
L1
64 KB
L2
256 KB
L2
256 KB
L3
8 MB
RAM

Sur ma machine :



```
[denis@portabledenis listings]$ getconf -a |grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  1310720
LEVEL2_CACHE_ASSOC                 20
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  12582912
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
```

Principes

Mémoire cache

Mémoire cache

Idée : on duplique une partie de la mémoire centrale dans la mémoire cache.

But : diminuer les temps d'accès (la mémoire cache est plus rapide mais plus couteuse).

Deux principes :

- *localité spatiale* : l'accès à une donnée à une adresse x sera suivi par des accès à une zone proche de x.
  $\Rightarrow$ quand on accède à une donnée, on amène un voisinage de cette donnée dans le cache.
- *localité temporelle* : l'accès à une zone mémoire peut se répéter.
  $\Rightarrow$ on garde dans le cache une donnée accédée récemment.

Mesure de l'efficacité

Hit La donnée a été trouvé dans le bon niveau de mémoire.

Miss La donnée n'a pas été trouvé ⇒ Il faut chercher dans le niveau suivant.

$$\begin{aligned}\text{Hit Rate} &= \#\textit{hits}/\#\textit{memory accesses} \\ &= 1 - \text{Miss Rate}\end{aligned}$$

$$\begin{aligned}\text{Miss Rate} &= \#\textit{misses}/\#\textit{memory accesses} \\ &= 1 - \text{Hit Rate}\end{aligned}$$

Organisation

- Le cache est organisé par ligne. Chaque ligne contient une portion des données de la mémoire, et une étiquette (tag) qui est l'adresse de ces données dans la mémoire.
- Lors de l'accès à la mémoire, on compare l'adresse avec les étiquettes du cache. On parle de succès ou de défaut de cache suivant le cas.

Index	Dirty	Tag	Data
0	0	`362F`	`4F ...`
1	0	`F3E0`	`00 ...`
2	1	`980A`	`56 ...`
3	0	`51DE`	`1A ...`

Un élément crucial de l'efficacité du cache est de retrouver rapidement si des données à une adresse mémoire sont déjà dans le cache.

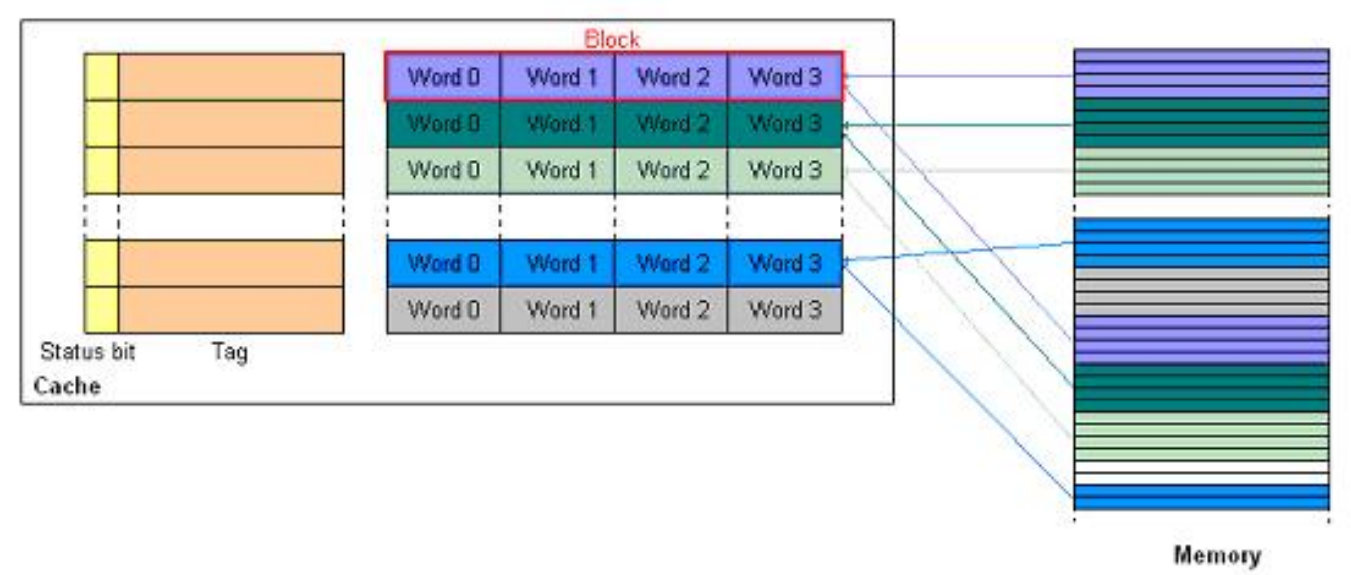
Différents types de cache

Pour accélérer la recherche, le nombre de lignes où peuvent être rangées les données à une adresse mémoire est souvent réduit. Ce nombre représentent l'associativité du cache.

3 types principalement suivant la fonction de rangement (*mapping*) d'un bloc de la mémoire dans une ligne du cache :

- les mémoires caches directes (direct mapped cache)
  associativité $= 1$
- les mémoires caches complètement associatives (fully associative cache)
  associativité = nombre de ligne du cache ;
- les mémoires caches N-associatives (N-way set associative cache)
  associativité $= N$.

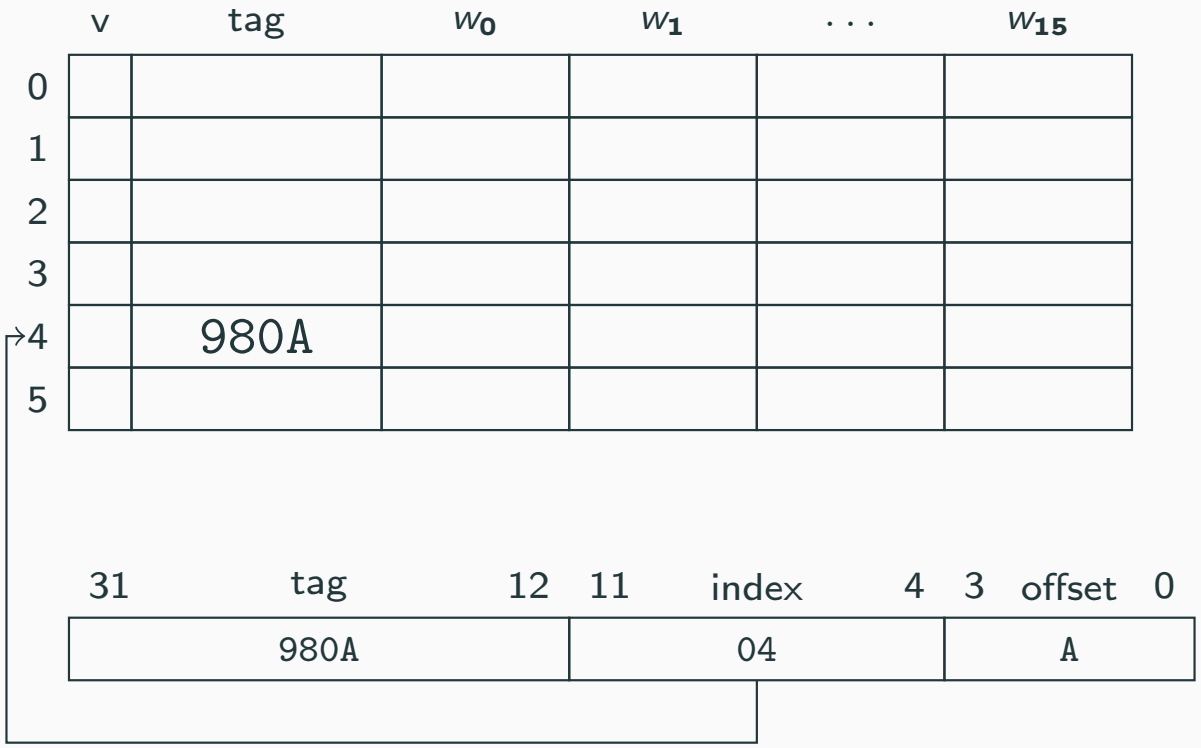
Caches directes



- Une ligne de la mémoire ne peut aller que dans une ligne du cache. On utilise généralement les bits de poids faible de l'adresse (modulo).
- Dans le cache, les bits utilisés pour déterminer la ligne n'ont pas besoin d'être stockés dans le tag.
- La recherche est rapide, mais risque de collisions.

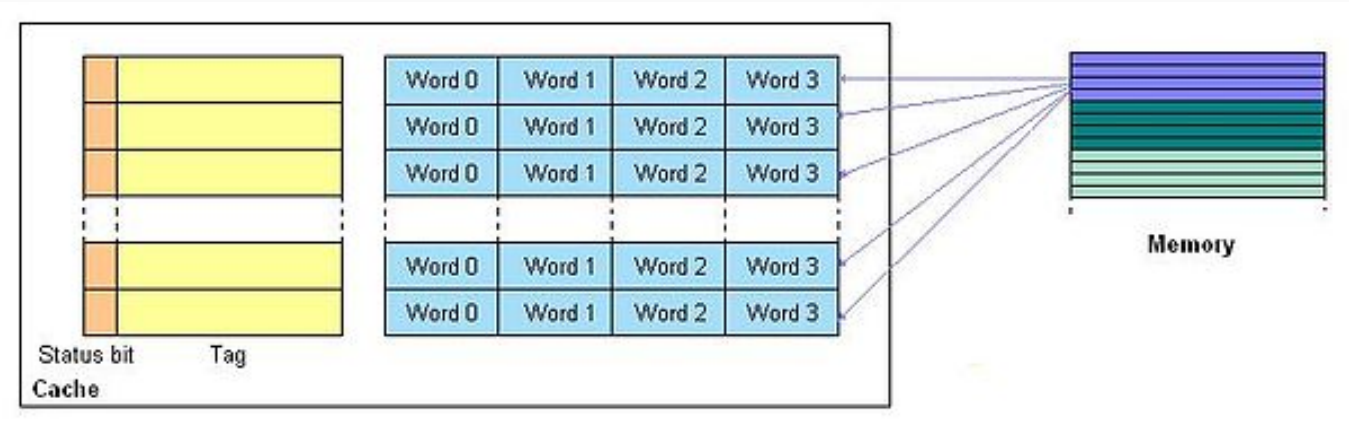
Un exemple concret

Un cache de 16 Ko (256 lignes de 16 mots), adresse mémoire sur 32 bits.



- `index` (8 bits) fournit le numero de la ligne dans le cache : $2^8 = 256$.
- `tag` (20 bits) est stocké dans la ligne pour savoir si elle correspond à la ligne en mémoire.
- `block offset` (4 bits) permet d'accéder à un des 16 mots d'une ligne.
- `v` est le bit de validité de la ligne.

Cache complétement associatif

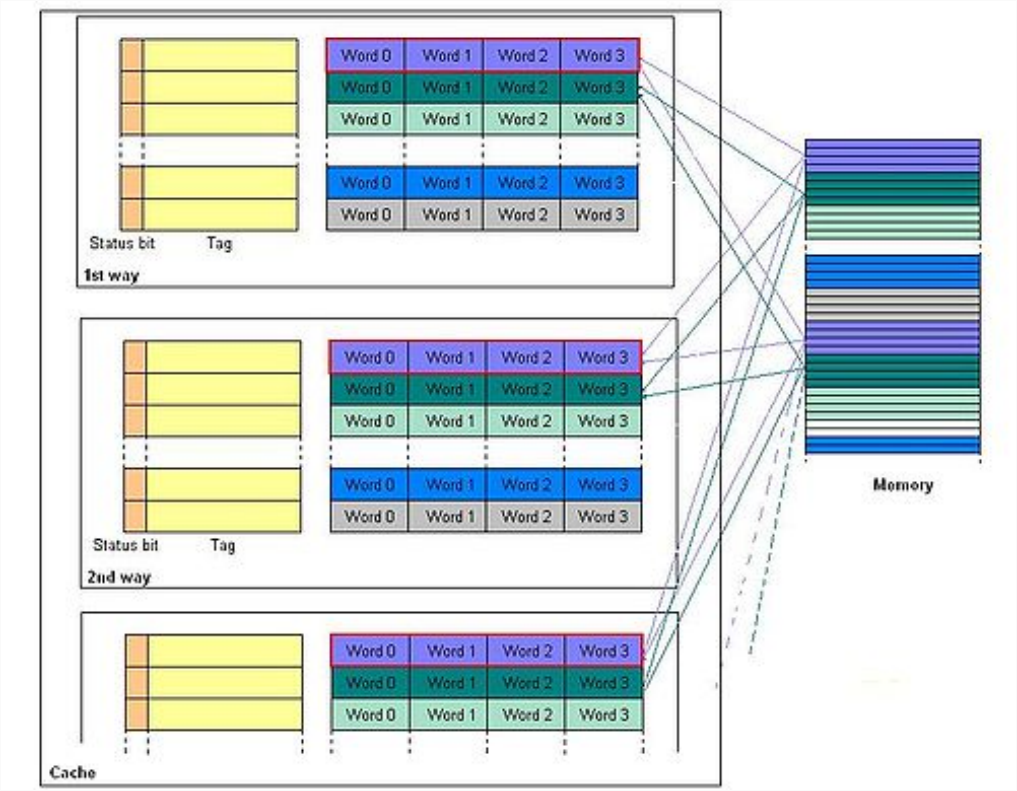


- Une ligne de la mémoire peut aller dans n'importe quelle ligne cache.
- On découpe l'adresse mémoire en un couple (tag, $offset$)

Cache N-associatif

Compromis entre direct et associatif.

La mémoire cache est divisée en ensembles associatifs (sets) de `N` lignes de cache.



Chaque adresse peut aller dans un ensemble de `N` lignes.

Résumé sur l'organisation

Paramètres

- C : capacité.
- *lsize* : taille des lignes
- Nombre de lignes : $L = C/lsize$
- N : nombre de lignes par ensemble
- Nombre d'ensembles : L/N

Organisation	Nombre de voies	Nombre d'ensembles
Direct	1	L
N-associatif	$1 < N < L$	L/N
complétement associatif	L	1

Écriture dans la mémoire

2 stratégies :

- write-through (écriture immédiate)
  la donnée est écrite à la fois dans le cache et dans la mémoire principale. La mémoire principale et le cache ont à tout moment une valeur identique, simplifiant ainsi de nombreux protocoles de cohérence.
- write back (écriture différée)
  la ligne est écrite (si nécesssaire) dans la mémoire lorsqu'elle est remplacée.

En cas de défaut de cache en écriture, 2 stratégies :

- write allocate
  on alloue une ligne dans le cache.
- no write allocate
  on n'alloue pas de ligne, et on écrit seulement dans la mémoire.

Principes

Mémoire virtuelle/MMU

Besoins vis à vis du SE :

- Mémoire partagée par un nombre variable de processus.
- Plusieurs "simultanément" en mémoire.
- Usage dynamique.

Contraintes

- Sécurité : mémoire lisible pas le propriétaire uniquement.
- Intégrité : mémoire non modifiable par un autre processus.
- Disponibilité : le SE doit satisfaire au mieux la demande.

Solution classique : Mémoire virtuelle et pagination.

- Donner l'illusion d'une mémoire plus grande, sans le coût associé à la dram.
- La mémoire principale (DRAM) joue le rôle de cache pour le disque dur.

Mémoire virtuelle

Chaque processus utilise des adresses virtuelles :

- L'espace virtuel est stocké sur le disque.
- Un sous-ensemble est en mémoire principale (DRAM) $\Rightarrow$ espace d'adressage $\geq$ taille de la mémoire physique
- Traduction adresses virtuelles $\rightarrow$ adresses physiques.
- Les données non présentes en mémoire principale sont chargées depuis le disque.

Chaque processus a sa propre correspondance virtuel $\rightarrow$ physique :

- 2 processus peuvent utiliser la même adresse virtuelle pour des données différentes $\Rightarrow$ simplification de l'Application Binary Interface (ABI).
- Cloisonnement et protection.

Analogie entre mémoire cache et mémoire virtuelle

Cache	Mémoire virtuelle
ligne	page
taille ligne	taille page
offset ligne	offset page
Miss	defaut de page
tag	virtual page number

MMU

- Adresse : (*page*, *offset*)
- Mémoire réelle en frame (cadres) de même taille que les pages logiques.
- MMU (*Memory Management Unit*) : adresse virtuelle $\rightarrow$ adresse réelle

Il doit :

- savoir quelles pages virtuelles sont en mémoire et où.
- prévenir le SE des défauts de page.

TLB : Translations Lookaside Buffers (cache de translation des adresses).

Calcul d'adresses

al : adresse logique

ap : adresse physique

tp : taille d'une page

tc[] : table de correspondance

numéro de page → numéro de cadre

- no_page = al div tp
- décalage = al mod tp // en anglais : offset
- ap = tc[no_page]*tp + décalage

`/proc/pid/pagemap`

Dans la "vraie vie", la table de page est hiérarchique (arbre).

Exemple tp = 10 et tc =

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
21	22	23	24	02	03	04	36	34	35	43	44	45	46	10	xx

al = 123

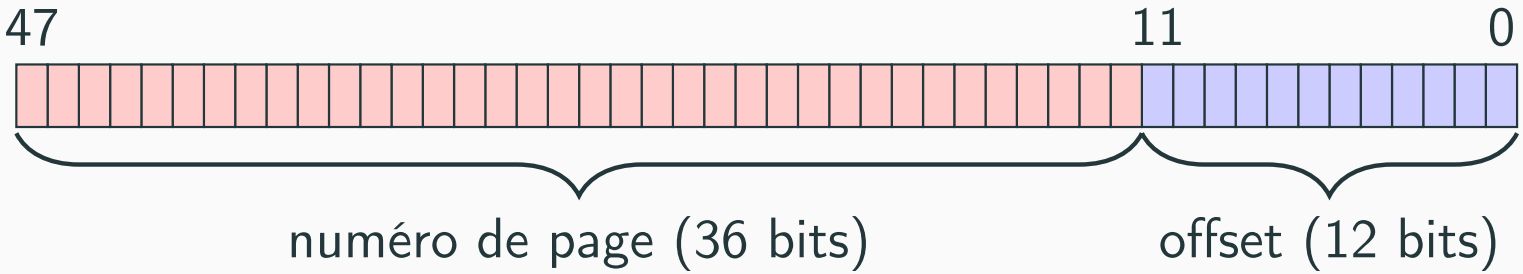
page 12 (120 à 129)

tc[12]=45 (450 à 459)

ap = 453

Remarques :

- La taille des pages est une puissance de 2 : 4 ko en général.
- L'adresse virtuelle est découpé en 2 morceaux : numero de page / offset



Exemple de mémoire virtuelle

Système :

- mémoire virtuelle $2Go = 2^{31}$ octets
- mémoire physique : $128Mo = 2^{27}$ octets
- taille page : $4Ko = 2^{12}$ octets

Organisation :

- Adresse virtuelle : 31 bits
- Adresse physique : 27 bits
- Page offset : 12 bits
- # pages virtuelles $= 2^{31}/2^{12} = 2^{19}$ (VPN = 19 bits)
- # pages physiques $= 2^{27}/2^{12} = 2^{15}$ (PPN = 15 bits)

Table des pages

La table des pages est un tableau : le numéro du bloc correspondant à la page n est à la n-ième case du tableau. Chaque entrée peut contenir des flags :

- R,W,X : indiquent si la page est accessible en lecture / écriture / exécution
- p (present) : indique si la page est présente en RAM ou sur l'espace d'échange (swap) Drapeaux renseignés par le MMU / utilisés (et parfois modifiés) par le SE
- a (accessed) : indique si la page a été lue
- d (dirty) : indique si la page été modifiée
- etc.

Exemples

Exemple 1 : adresse physique de l'adresse virtuelle `0x5F20` ?

p(resent)	Page Frame Number
0	
1	`0x0000`
1	`0x7FFE`
0	
0	
	⋮
0	
0	
1	`0x0001`
0	
0	
1	`0x7FFF`
0	
0	

Exemples

Exemple 1 : adresse physique de l'adresse virtuelle `0x5F20` ?

- VPN = 5
- entrée 5 de la table de page donne la page physique 1
- adresse physique = `0x1F20`

p(resent)	Page Frame Number
0	
1	`0x0000`
1	`0x7FFE`
0	
0	
	⋮
0	
0	
1	`0x0001`
0	
0	
1	`0x7FFF`
0	
0	

Exemples

Exemple 2 : adresse physique de l'adresse virtuelle `0x73E0` ?

p(resent)	Page Frame Number
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Exemples

Exemple 2 : adresse physique de l'adresse virtuelle `0x73E0` ?

- VPN = 7
- entrée 7 n'est pas présente
- défaut de page

p(resent)	Page Frame Number
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Principes

Stratégie de remplacement de pages

Défaut de pages

En cas de défaut de page, le MMU émet une requête d'interruption, et c'est le SE qui choisit une page à remplacer.

Algorithmes

- Algorithme optimale
- choix aléatoire.
- FIFO (First In First Out)
- Second Chance
- LRU (Least Recently Used)
- NRU (Not Recently Used).
- etc.

Optimale

À chaque défaut de page, on choisit celle qui sera re-utilisé le plus tard.

- Il faut connaître l'avenir :-)
- Utilité théorique pour évaluer a posteriori les autres algorithmes.

Exemple : 3 pages en caches, accès avec la séquence

7 , 0 , 1 , 2 , 0 , 3 , 0 , 4 , 2 , 3 , 0 , 3 , 2 , 1 , 2 , 0 , 1 , 7 , 0 , 1

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
c_0	7	7	7	2		2		2			2			2				7		
c_1		0	0	0		0		4			0			0				0		
c_2			1	1		3		3			3			1				1		

9 défauts

Fifo

À chaque défaut, on choisit la plus en ancienne en mémoire. (On espère qu'elle est celle qui a le moins de chance d'être réutilisé)

Exemple : 3 pages en caches, accès avec la séquence

7 , 0 , 1 , 2 , 0 , 3 , 0 , 4 , 2 , 3 , 0 , 3 , 2 , 1 , 2 , 0 , 1 , 7 , 0 , 1

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
c_0	7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
c_1		0	0	0		3	3	3	2	2	2			1	1			1	0	0
c_2			1	1		1	0	0	0	3	3			3	2			2	2	1

15 défauts

Défauts

- La plus ancienne est peut-être la plus utilisée.
- Anomalie de Belady.

Seconde Chance

Amélioration de Fifo.

On ajoute à chaque page un bit pour savoir si la page est en cours d'utilisation par un processus.

Avant de retirer une page, on lui donne une seconde chance :

- si le bit est à 0, on la retire.
- sinon, on met le bit à 0 (deuxième chance, la page est mise en queue), et on cherche une autre page.

NRU : Not Recently used

On utilise 2 bits pour chaque page :

- R : la page a été lue.
- M : la page a été modifiée.

Au lancement, toutes les pages ont R et M à 0. Régulièrement, R est repassé à 0 pour toutes les pages.

Lors d'un défaut de page, le système classe les pages dans 4 ensembles :

- R=0 et M=0 (ni utiliséé ni modifiée)
- R=0 et M=1
- R=1 et M=0
- R=1 et M=1

NRU remplace une page au hasard dans le premier ensemble non vide.

Least Recently Used

Solution naïve : tri dynamique des pages par ordre d'accès (couteux).

Autre solution algorithmique :

- Matrice triangulaire $N \times N$,
- Hachage et liste doublement chaînée.

Linux

Linux

Mémoire virtuelle du processus

- Chaque processus est exécuté comme s'il avait toute la mémoire.
- Un processus peut accéder uniquement à sa mémoire. Il existe quand même des mécanismes de partage.
- La mémoire virtuelle est décomposée en segments (facilite le partage et la protection)

Processus

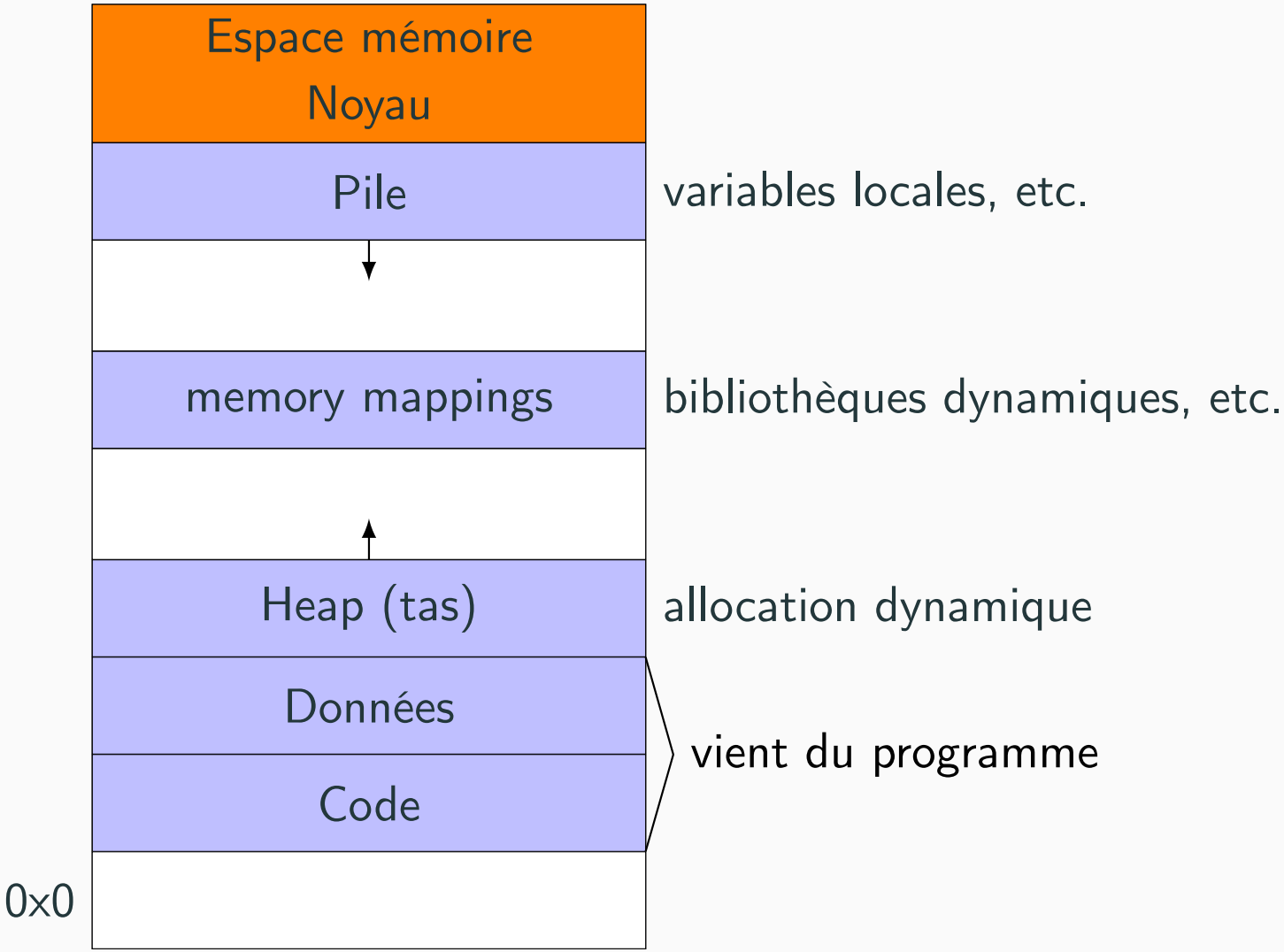
Image dynamique de l'exécution d'un programme.

- Un programme (fichier exécutable - format ELF) est statique. Il fournit :
  - Le code (instructions) - Code Segment
  - Les données - Data Segment
- Son exécution par le SE est dynamique ⇒ Processus

Durant toute sa vie, l'état d'un processus comprend :

- mémoire : code, data, tas, pile.
- contexte d'exécution : registres, compteur ordinal, sommet de pile, etc.
- Process control Bloc (PCB) : état vis à vis du SE.

Mémoire du processus



cat /proc/pid/maps

```
558d6b6fa000-558d6b6fb000 r--p 00000000 08:04 541510                     a.out
558d6b6fb000-558d6b6fc000 r-xp 00001000 08:04 541510                     a.out
558d6b6fc000-558d6b6fd000 r--p 00002000 08:04 541510                     a.out
558d6b6fd000-558d6b6fe000 r--p 00002000 08:04 541510                     a.out
558d6b6fe000-558d6b6ff000 rw-p 00003000 08:04 541510                     a.out
558d6ce5f000-558d6ce80000 rw-p 00000000 00:00 0                          [heap]
7f0fc83ed000-7f0fc83ef000 rw-p 00000000 00:00 0
7f0fc83ef000-7f0fc8415000 r--p 00000000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc8415000-7f0fc8562000 r-xp 00026000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc8562000-7f0fc85ae000 r--p 00173000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85ae000-7f0fc85b1000 r--p 001be000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85b1000-7f0fc85b4000 rw-p 001c1000 08:03 1969089                    /usr/lib/libc-2.32.so
7f0fc85b4000-7f0fc85ba000 rw-p 00000000 00:00 0
7f0fc8600000-7f0fc8602000 r--p 00000000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc8602000-7f0fc8623000 r-xp 00002000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc8623000-7f0fc862c000 r--p 00023000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc862c000-7f0fc862d000 r--p 0002b000 08:03 1969032                    /usr/lib/ld-2.32.so
7f0fc862d000-7f0fc862f000 rw-p 0002c000 08:03 1969032                    /usr/lib/ld-2.32.so
7fffbe4a3000-7fffbe4c4000 rw-p 00000000 00:00 0                          [stack]
7fffbe59b000-7fffbe59f000 r--p 00000000 00:00 0                          [vvar]
7fffbe59f000-7fffbe5a1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Il s'agit d'adresses virtuelles !

Mémoire virtuelle vs mémoire physique

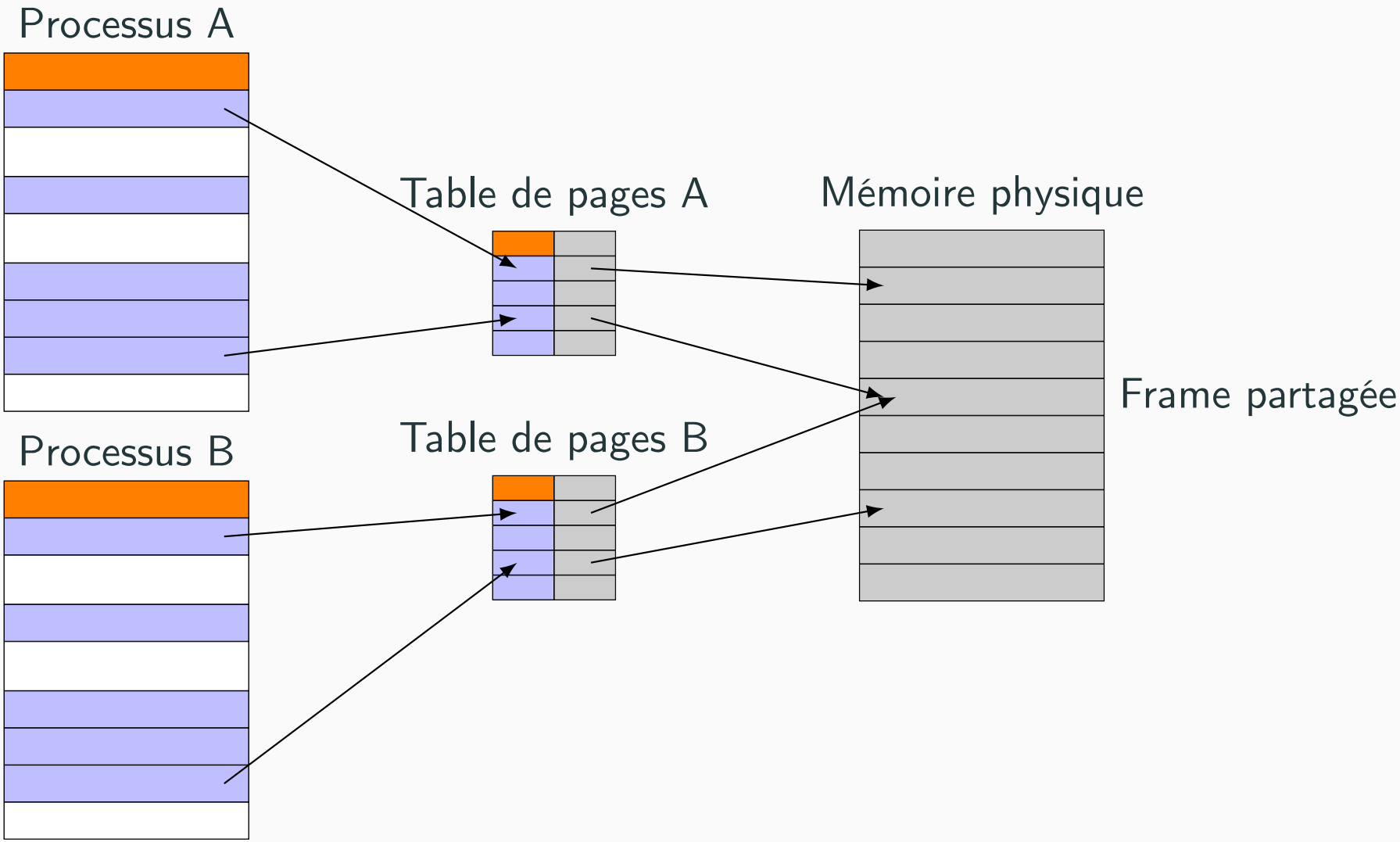


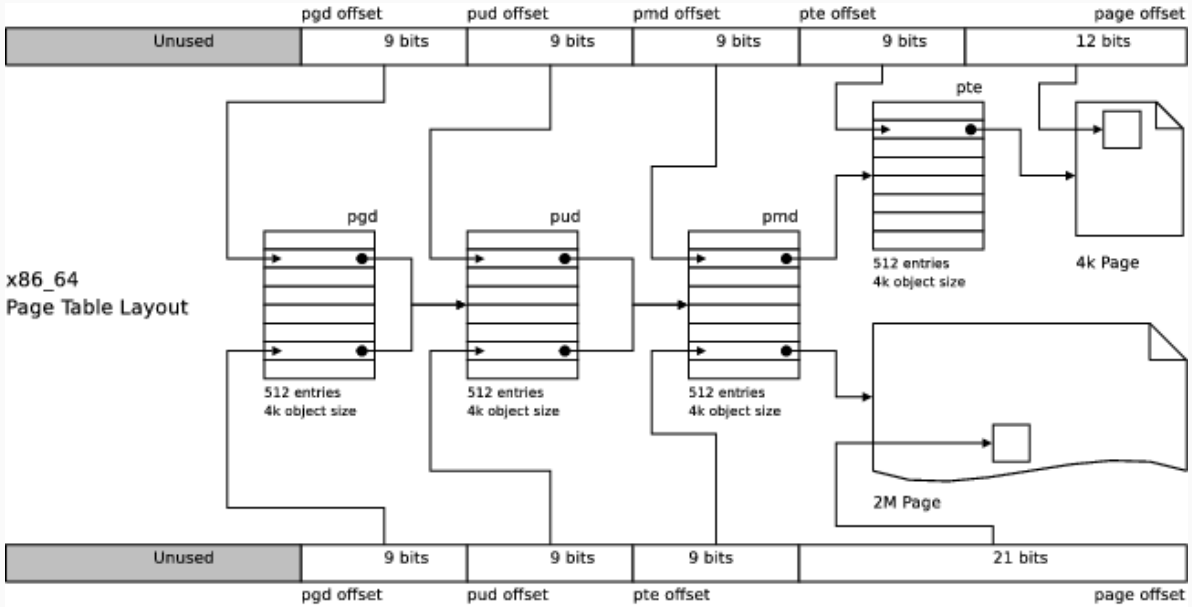
Table de pages

- Chaque processus à sa propre table.
- La même adresse virtuelle peut être utilisée dans 2 processus différents et mappée avec des adresses physiques différentes. (il y a par exemple sur x86 un registre (`cr3`) qui pointe sur le repertoire de page actif)
- Utilisation d'un cache pour les translations (Translation Lookaside Buffer).

Cas de l'architecture x86_64

Quatre niveaux (arbre) pour des pages de 4 Ko : Page Global Directory, Page Upper Directory, Page Mid-level Directory, Page Table Entry

Bits utilisés	PGD	PUD	PMD	PTE
x86_64	39-47	30-38	21-29	12-20



Chaque processus possède sa propre Page Global Directory (adresse physique stockée dans le registre cr3).

Bits de contrôle sur les pages

bit	function
`_PAGE_PRESENT`	Page is resident in memory and not swapped out
`_PAGE_PROTNONE`	Page is resident but not accessable
`_PAGE_RW`	Set if the page may be written to
`_PAGE_USER`	Set if the page is accessible from user space
`_PAGE_DIRTY`	Set if the page is written to
`_PAGE_ACCESSED`	Set if the page is accessed

Linux

Découpage de l'espace d'adressage

Le code

- Chargement en mémoire de l'éxecutable (Linux : format ELF)

Les données

- Variables globales
- Variables locales static

```c
#include <stdio.h>

int i=3;
int j;
void count(){
  static int cpt = 0;
  cpt++;
}
```

La pile

- Variables locales, paramètres de fonctions.
- Allouées et desallouées "automatiquement" (pas de `free`)

```
int * f(){
  int i;
  return &i; /* illegal ! */
}
```

- Allocation sur la pile.

```
void stack_alloc(int n){
int * arr = alloca(n*sizeof(int));
return;
}
```

Le tas

- Gestion dynamique de la mémoire.
- `malloc`, `calloc`, `free`, etc.

```
void heap_alloc(int n){

int * arr1,*arr2;
arr1 = (int*) malloc(n*sizeof(int));
arr1 = (int*) calloc(n,sizeof(int));



free(arr1);
free(arr2);
}
```

Durée de vie d'une variable

```
char ch1 = 'A';
const char ch2 = 'B';

void f(void) {
    char ch3 = 'C';
    char* ch4 = (char*)malloc(sizeof(char));
}
```

- *statique* : durée de vie du processus (ch1, ch2).
- *automatique* : le compilateur alloue/désalloue automatiquement en fonction de la portée (ch3,ch4).
- *dynamique* : le programmeur alloue/désalloue lui-même (*ch4).

Gestion du tas

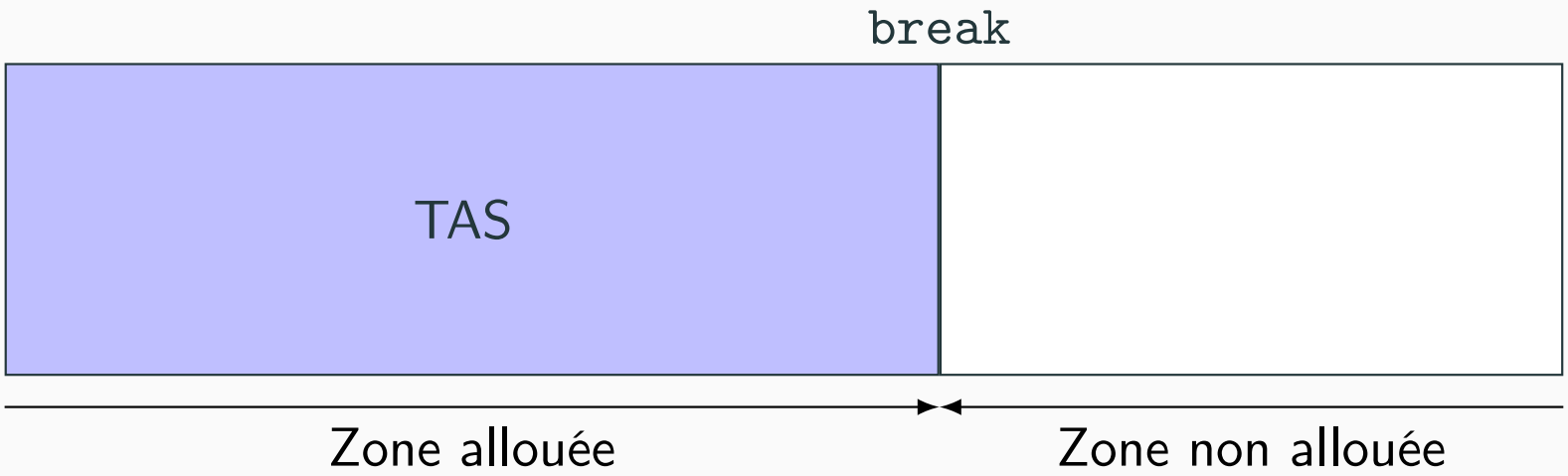
Changement de sa taille

- Déplacement absolu

```c
int brk(void *addr);
```

- Déplacement relatif

```c
void *sbrk(intptr_t increment);
```



mmap

```
void *mmap(void *addr, size_t length, int prot, int flags,
                    int fd, off_t offset);
```

`mmap()` est le moyen le plus standard d'allouer de grande quantité de mémoire en espace user.

- Bien que souvent utilisé pour des fichiers, le flag `MAP_ANONYMOUS` permet d'allouer de la mémoire au processus.
- `MAP_SHARED` permet de partager des pages avec d'autres processus.

La taille demandée est alignée sur la taille des pages.

L'implantation des fonctions `malloc/calloc` peuvent utiliser soit `brk` ou `mmap`

`mallopt` et le paramètre `M_MMAP_THRESHOLD` permet de contrôler le comportement.

Linux

Alignement

Interêt : rendre la partie martérielle plus simple et rapide.

Exemple :

Imaginons un cache (cpu) avec des lignes de 128 octets. Ces lignes auront des adresses systématiment alignées.

Les adresses `127,128,129,130` vivent dans 2 lignes différentes :

- `[0,127]`
- `[128,255]`

Un entier (4 octets) avec une adresse alignée sur 4
`[4n,4n+1,4n+2,4n+3]` est toujours dans le même ligne !

Alignement

x86-64 Linux. `T` type primitif

Type	Taille	Adresse	(alignof(Type))
`char` (signed , unsigned)	1	/	1
`short` (unsigned short)	2	Multiple de 2	2
`int` (unsigned int)	4	Multiple de 4	4
`long` (unsigned long)	8	Multiple de 8	8
`float`	4	Multiple de 4	4
`double`	8	Multiple de 8	8
`long double`	16	Multiple de 16	16
`T*`	8	Multiple de 8	8

alignof(T) == sizeof(T)

`malloc()`

- renvoie un pointeur générique, qui doit pouvoir être casté vers `T*` pour n'importe quel type `T`.
- dans la pratique, `malloc` renvoie une adresse alignée sur 16.

Cas des structures

- l'adresse du premier membre est l'adresse de la structure.
- les membres sont rangés dans l'ordre avec les contraintes d'alignement propre à leur type (padding souvent nécessaire).
- l'alignement de la structure est égale à l'alignement maximal de ses membres (le ppcm, mais il s'agit toute de puissance de 2).

On peut allouer en alignant (sur la taille du cache par exemple) avec la fonction

```
int posix_memalign(void **memptr,
                   size_t alignment, size_t size);
```