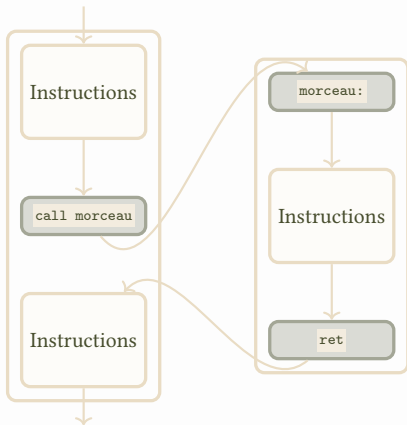


Instructions `call` / `ret`

Schématiquement, voici l'action produite sur l'exécution d'un programme par le couple `call` / `ret`



Attention : le retour à l'endroit du code attendu par l'instruction `ret` n'est correct que si l'état de la pile à l'étiquette `morceau` est le même que celui juste avant le `ret`.

Fonctions — conventions d'appel du C

L'écriture de fonctions respecte des conventions indiquant comment sont passés les arguments et comment la valeur de retour peut être récupérée.

Celle majoritairement employée sur l'architecture x86 est la conventions d'appel du C. Celle-ci consiste en le respect des points suivants :

1. les **arguments** d'une fonction sont passés, avant son appel, dans la **pile** en les empilant ;
2. le **résultat** d'une fonction est renvoyé en l'écrivant dans le registre `eax` ;
3. la **pile** doit être dans le **même état** avant l'appel et après l'appel d'une fonction.

Ceci signifie que l'état des pointeurs `esp` et `ebp` sont conservés et que le contenu de la pile qui suit l'adresse `esp` est également conservé ;

4. les **valeurs** des registres **différents de** `eax`, `ecx` et `edx` doivent être dans le **même état** avant l'appel et après l'appel de la fonction.

Fonctions — écriture et appel

L'**écriture** d'une fonction suit le squelette

```
NOM_FCT:
    push ebp
    mov ebp, esp

    INSTR

    pop ebp
    ret
```

Ici, `NOM_FCT` est une étiquette d'instruction qui fait d'office de nom pour la fonction. De plus, `INSTR` est un bloc d'instructions.

Il est primordial que `INSTR` **conserve l'état de la pile**.

L'**appel** d'une fonction se fait par

```
push ARG_N
...
push ARG_1

call NOM_FCT

add esp, 4 * N
```

Ici, `NOM_FCT` est le nom de la fonction à appeler. Elle admet `N` arguments qui sont **empilés du dernier au premier**.

Après l'appel, on incrémente `esp` pour dépiler d'un coup les `N` arguments de la fonction.

Fonctions — appel et état de la pile

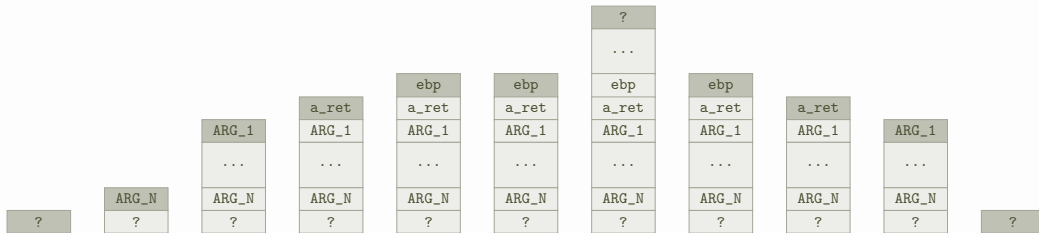
Analysons l'état de la pile, étape par étape, lors de l'appel d'une fonction.

```
; Appel de la fonction NOM_FCT.  
push ARG_N  
...  
push ARG_1  
call NOM_FCT  
ADD esp, 4 * N ; Adresse a_ret.
```

```
; Definition de la fonction NOM_FCT.  
NOM_FCT:  
    push ebp  
    mov ebp, esp  
    INSTR  
    pop ebp  
    ret
```

À partir du 6^e dessin de pile, **ebp** pointe vers une zone fixée de la mémoire.

Il s'agit de la zone de la pile située à deux cases au dessus de là où figure le 1^{er} argument.



Fonctions — préservation de l'état de la pile

L'état de la **pile** doit être **préservé** par le bloc d'instructions **INSTR**.

Cela signifie que l'état de la pile juste avant d'exécuter **INSTR** et son état juste après son exécution sont les mêmes. En d'autres termes,

- **esp** doit posséder la même valeur ;
- toutes les données de la pile d'adresses plus grandes que **esp** ne doivent pas être modifiées.

C'est bien le cas si

- **INSTR** contient autant de **push** que de **pop** ;
- à tout instant, il y a au moins autant de **push** que de **pop** qui ont été exécutés.

Il faut bien respecter ces deux conditions dans la pratique.

Fonctions — rôle de `ebp`

La **valeur** de `esp` est susceptible de **changer** dans `INSTR`. C'est pour cela que l'on **sauvegarde** sa valeur, à l'entrée de la fonction, dans `ebp`.

Le registre `ebp` sert ainsi, dans `INSTR`, à **accéder aux arguments**. En effet, l'adresse du 1^{er} argument est `ebp + 8`, celle du 2^e est `ebp + 12` et plus généralement, celle du `ie` argument est

$$\text{ebp} + 4 * (\text{i} + 1)$$

On **sauvegarde** et on **restaure** tout de même, par un `push ebp` et `pop ebp` l'état de `ebp` à l'entrée et à la sortie de la fonction.

Ce même mécanisme doit être utilisé pour sauvegarder/restaurer l'état des registres de travail `eax` (sauf si la fonction renvoie une valeur), `ebx`, `ecx` et `edx`.

Fonctions — exemple 1

– Exemple –

```
; Fonction qui renvoie la somme de deux entiers.  
; Arguments :  
; (1) une valeur entière signée sur 4 octets  
; (2) une valeur entière signée sur 4 octets.  
; Renvoi : la somme des deux arguments.
```

```
somme:  
    push ebp  
    mov ebp, esp  
  
    mov eax, [ebp + 8]  
    add eax, [ebp + 12]  
  
    pop ebp  
    ret
```

Pour calculer dans `eax` la somme de $(43)_{\text{dix}}$ et $(1996)_{\text{dix}}$, on procède par

```
push 1996  
push 43  
call somme  
add esp, 8
```

Rappel 1 : on empile les arguments dans l'ordre inverse de ce que la fonction attend.

Rappel 2 : on ajoute `8` à `esp` après l'appel pour dépiler d'un seul coup les deux arguments ($8 = 2 \times 4$).

Fonctions — exemple 2

– Exemple –

```
; Fonction qui affiche un caractere.  
; Arguments :  
; (1) valeur du caractere a afficher.  
; Renvoi : rien.
```

```
print_char:  
    ; Debut.  
    push ebp  
    mov ebp, esp
```

```
; Sauv. des registres.  
    push eax  
    push ebx  
    push ecx  
    push edx
```

```
; Affichage.  
    mov ebx, 1  
    mov ecx, ebp  
    add ecx, 8  
    mov edx, 1  
    mov eax, 4  
    int 0x80
```

```
; Rest. des registres.  
    pop edx  
    pop ecx  
    pop ebx  
    pop eax  
  
; Fin.  
    pop ebp  
    ret
```

Fonctions — exemple 2 bis

– Exemple –

```
; Fonction qui affiche un caractere.  
; Arguments :  
; (1) adresse du caractere a afficher.  
; Renvoi : rien.
```

```
print_char_2:
```

```
; Debut.  
push ebp  
mov ebp, esp
```

```
; Sauv. des registres.  
push eax  
push ebx  
push ecx  
push edx
```

```
; Affichage.  
mov ebx, 1  
mov ecx, [ebp + 8]  
mov edx, 1  
mov eax, 4  
int 0x80
```

```
; Rest. des registres.  
pop edx  
pop ecx  
pop ebx  
pop eax  
  
; Fin.  
pop ebp  
ret
```

Fonctions — exemples 2 et 2 bis

Les deux fonctions `print_char` et `print_char_2` ne s'utilisent pas de la même manière : la 1^{re} prend son argument **par valeur**, tandis que la 2^e prend son argument **par adresse**.

– Exemple –

Pour afficher p.ex. le caractère 'W', on appelle `print_char` par

```
push 'W'
call print_char
add esp, 4
```

La fonction `print_char_2` s'appelle par

```
push c1
call print_char_2
add esp, 4
```

où `c1` est l'adresse d'un caractère en mémoire. Celle-ci a été définie par exemple par `c1 : db 'W'` dans la section de données.

Fonctions — Déclaration C

En C, les deux fonctions `print_char` et `print_char_2` se différencient par leur prototype :

`void print_char(char c)` VS `void print_char2(char *c)`

En suivant la convention d'appel, la déclaration d'une fonction suffit à savoir comment l'utiliser.

– Exemple –

Pour appeler la fonction `nom_fun` de signature `int nom_func(int n, int *tab, char *str)`, il faut :

- Empiler l'adresse d'un `byte`
- Empiler l'adresse d'un `dword`
- Empiler la valeur d'un `dword`
- Faire un `call` sur l'étiquette de la fonction

La valeur renvoyée par la fonction se trouvera dans le registre `eax`.

Inversement, pour accéder au i -ième argument dans le code d'une fonction, on a la méthode `mov REG, [ebp + 4 * (i + 1)]`.

Chaînes de caractères

Une chaîne de caractères est une **suite contiguë d'octets** en mémoire se terminant par l'octet nul.

On accède à une chaîne de caractères via l'**adresse** `x` de son **1^{er} caractère**. Par abus de langage, on parle de « la chaîne `x` ». Les adresses des autres caractères s'obtiennent par incrémentation.

– Exemple –

La chaîne `x` de valeur `"nasm"` est représentée en mémoire par

Mémoire	'n'	'a'	's'	'm'	0
Adresses	<code>x</code>	<code>x + 1</code>	<code>x + 2</code>	<code>x + 3</code>	<code>x + 4</code>

La valeur de `[x]` est `'n'`, celle de `[x + 1]` est `'a'`, ..., celle de `[x + 4]` est `0`.

On **lit** le `ie` caractère d'une chaîne `x` en le rangeant dans `al` par `mov al, [x + i - 1]`.

On **écrase** le `ie` caractère d'une chaîne `x` en la valeur contenue dans `al` par `mov [x + i - 1], al`.

Toute fonction ayant un argument chaîne de caractère travaille à partir de l'adresse de cette chaîne. Il s'agit d'un **passage par adresse**.

Fonctions — exemple 3

– Exemple –

```
; Fonction qui affiche une chaine de carac.  
; Arguments :  
; (1) adresse de la chaine de carac.  
; Renvoi : nombre caracteres affichees.
```

```
print_string:
```

```
; Debut.  
push ebp  
mov ebp, esp
```

```
; Sauv. des registres.  
push ebx
```

```
; Adresse du debut de la chaine.  
mov ebx, [ebp + 8]
```

```
; Init. compteur  
mov eax, 0
```

```
; Boucle d'affichage.  
boucle:  
    cmp byte [ebx], 0  
    je fin_boucle
```

```
; Appel de print_char.  
push dword [ebx]  
call print_char  
add esp, 4
```

```
    add ebx, 1  
    add eax, 1  
    jmp boucle  
fin_boucle:
```

```
; Rest. des registres.  
pop ebx  
  
; Fin.  
pop ebp  
ret
```

Fonctions — exemple 4

– Exemple –

On souhaite écrire une fonction pour calculer **récurivement** le n^{e} nombre triangulaire $\text{triangle}(n)$ défini par $\text{triangle}(0) := 0$ et pour tout $n \geq 1$, $\text{triangle}(n) := n + \text{triangle}(n - 1)$.

```
; Fonction de calcul des nb triangulaires.  
; Arguments :  
;   (1) entier positif  
; Renvoi : le nb triangulaire de l'arg.
```

```
triangle:  
    ; Debut.  
    push ebp  
    mov ebp, esp  
  
    ; Sauv. des registres.  
    push ebx  
    push ecx  
    push edx  
  
    ; Sauv. de l'argument.  
    mov ebx, [ebp + 8]
```

```
    cmp ebx, 0  
    je cas_terminal  
cas_non_terminal:  
  
    ; Preparation appel recursif.  
    mov ecx, ebx  
    sub ecx, 1  
  
    ; Appel recursif.  
    push ecx  
    call triangle  
    add esp, 4  
  
    ; Calcul du resultat.  
    add eax, ebx  
  
    jmp fin
```

```
cas_terminal:  
    mov eax, 0  
  
fin:  
  
    ; Rest. des registres.  
    pop edx  
    pop ecx  
    pop ebx  
  
    ; Fin.  
    pop ebp  
    ret
```

Fonctions — exemple 5

– Exemple –

On souhaite écrire une fonction pour calculer **récurivement** le factorielle $\text{fact}(n)$ définie par $\text{fact}(0) := 1$ et pour tout $n \geq 1$, $\text{fact}(n) := n \times \text{fact}(n - 1)$.

```
; Fonction de calcul de la factorielle.  
; Arguments :  
;   (1) entier positif  
; Renvoi : la factorielle l'argument.
```

```
fact:  
    ; Debut.  
    push ebp  
    mov ebp, esp  
  
    ; Sauv. des registres.  
    push ebx  
    push ecx  
    push edx  
  
    ; Sauv. de l'argument.  
    mov ebx, [ebp + 8]
```

```
    cmp ebx, 0  
    je cas_terminal  
cas_non_terminal:  
  
    ; Preparation appel recursif.  
    mov ecx, ebx  
    sub ecx, 1  
  
    ; Appel recursif.  
    push ecx  
    call fact  
    add esp, 4  
  
    ; Calcul du resultat.  
    mul ebx  
  
    jmp fin
```

```
cas_terminal:  
    mov eax, 1  
  
fin:  
  
    ; Rest. des registres.  
    pop edx  
    pop ecx  
    pop ebx  
  
    ; Fin.  
    pop ebp  
    ret
```

Fonctions — exemple 6

– Exemple –

On souhaite écrire une fonction pour calculer **récurivement** le n^{e} nombre de Fibonacci $\text{fibonacci}(n)$ défini par $\text{fibonacci}(0) := 0$, $\text{fibonacci}(1) := 1$ et pour tout $n \geq 2$, $\text{fibonacci}(n) := \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$.

*; Fonction de calcul des nombres
de Fibonacci.*

; Arguments :

; (1) entier positif.

; Renvoi : le nombre de

; Fibonacci de l'arg.

fibonacci:

; Debut

push ebp

mov ebp, esp

; Sauv. des registres.

push ebx

push ecx

push edx

; Sauv. de l'argument.

mov ebx, [ebp + 8]

cmp ebx, 1

jle cas_terminaux

cas_non_terminal :

; Prepa. appel rec. 1.

mov ecx, ebx

sub ecx, 1

; Appel rec. 1.

push ecx

call fibo

add esp, 4

; Sauv. res. 1 pile.

push eax

; Prepa. appel rec. 2.

mov ecx, ebx

sub ecx, 2

; Appel rec. 2.

push ecx

call fibo

add esp, 4

; Calcul resultat.

pop ebx

add eax, ebx

jmp fin

cas_terminaux:

mov eax, ebx

fin:

; Rest. des registres.

pop edx

pop ecx

pop ebx

pop ebp

ret

Étiquettes locales

Dans un programme complet, il peut être difficile de gérer de nombreuses étiquettes de code.

Il existe pour cette raison la notion d'**étiquette locale**, dont la syntaxe de définition et de référence est

```
.ETIQ:
```

– Exemple –

```
etiq_globale:  
  .action:  
    INSTR_1  
    jmp .action
```

```
autre_etiq_globale:  
  .action:  
    INSTR_2  
    jmp .action
```

Ceci déclare plusieurs étiquettes de code, dont deux du même nom et locales, `.action`.

Le 1^{er} `jmp` saute à l'instruction correspondant au 1^{er} `.action`, tandis que le 2^e `jmp` saute à l'instruction correspondant au 2^e `action`.

Les noms absolus (`etiq_globale.action` et `autre_etiq_globale.action`) de ces étiquettes permettent de faire référence à l'étiquette réellement souhaitée si besoin est.

Programmation modulaire — modules

Il est possible de réaliser des projets en assembleur sur plusieurs fichiers, découpés en modules.

Un module est constitué

- d'un **fichier source** d'extension `.asm` ;
- d'un **fichier d'en-tête** d'extension `.inc`.

Seul le module qui contient la fonction principale `main` ne dispose pas de fichier d'en-tête.

Pour compiler un projet sur plusieurs fichiers, on se sert des commandes

```
nasm -f elf32 Main.asm
nasm -f elf32 M1.asm
...
nasm -f elf32 Mk.asm

ld -o Exec -melf_i386 -e main Main.o M1.o ... Mk.o
```

Programmation modulaire — étiquettes visibles

Un module (non principal) contient une collection de fonctions destinées à **être utilisées depuis l'extérieur**.

On autorise une étiquette d'instruction `ETIQ` à être visible depuis l'extérieur en ajoutant la ligne

```
global ETIQ
```

juste avant la définition de l'étiquette.

De plus, on renseigne dans le fichier d'en-tête l'existence de la fonction par

```
extern ETIQ
```

Il est d'usage de documenter à cet endroit la fonction.

Programmation modulaire — inclusion

Pour bénéficier des fonctions définies dans un module `M` dans un fichier `F.asm`, on invoque, au tout début de `F.asm`, la directive

```
%include "M.inc"
```

Uniquement les fonctions rendues visibles depuis l'extérieur de `M` peuvent être appelées dans `F.asm`.

– Exemple –

Voici un module `ES` et son utilisation dans `Main.asm` :

```
; ES.inc  
  
; Documentation...  
extern print_char  
  
; Documentation...  
extern print_string
```

```
; ES.asm  
  
...  
global print_char  
print_char:  
    ...  
  
global print_string  
print_string:  
    ...
```

```
; Main.asm  
  
%include "ES.inc"  
...  
call print_string  
...
```

Interaction avec le C

En suivant les conventions d'appel du C, cela permet à un code en assembleur d'utiliser et d'être utilisé par des fonctionnalités réalisées suivant les mêmes conventions et donc notamment des fonctionnalités réalisées en C.

Pour bénéficier des fonctions définies dans un module C `M` dans un fichier `F.asm`, il faut disposer d'un fichier `M.inc` qu'il faut éventuellement créer. Ce dernier comporte les mêmes symboles déclarés dans le `M.h`.

– Exemple –

Voici un module `ES` et son utilisation dans `Main.asm` :

<code>; ES.h</code>	<code>; ES.c</code>	<code>; ES.inc</code>
<code>// Documentation...</code>	<code>void print_char(char c)</code>	<code>; Documentation...</code>
<code>void print_char(char c);</code>	<code>{ ... }</code>	<code>extern print_char</code>
<code>// Documentation...</code>	<code>void print_string(char *s);</code>	<code>; Documentation...</code>
<code>void print_string(char *s){</code>	<code>... }</code>	<code>extern print_string</code>

```
; Main.asm
#include "ES.inc"
...
call print_string
...
```

Interaction avec le C

Pour bénéficier des fonctions définies dans un module assembleur `M` dans un fichier `F.c`, il faut disposer d'un fichier `M.h` qu'il faut éventuellement créer. Ce dernier comporte les mêmes symboles déclarés dans le `M.inc`.

– Exemple –

Voici un module `ES` et son utilisation dans `Main.c` :

```
; ES.inc
```

```
; Documentation...
```

```
extern print_char
```

```
; Documentation...
```

```
extern print_string
```

```
; ES.asm
```

```
...
```

```
global print_char
```

```
print_char:
```

```
...
```

```
global print_string
```

```
print_string:
```

```
...
```

```
; ES.h
```

```
// Documentation...
```

```
void print_char(char c);
```

```
// Documentation...
```

```
void print_string(char *s);
```

```
; Main.c
```

```
#include "ES.h"
```

```
...
```

```
print_string("Bonjour");
```

```
...
```

Interaction avec le C

Pour compiler un projet sur plusieurs fichiers, on se sert des commandes

```
nasm -f elf32 A1.asm
...
nasm -f elf32 Am.asm

cc -Wall -m32 -c -o .c.o C1.c
...
cc -Wall -m32 -c -o .c.o Cn.c
ld -o Exec -melf_i386 -e main A1.o ... Am.o C1.o ... Cn.o
```

Interaction avec le C

De nombreuses bibliothèques C dépendent de la bibliothèque standard du C, la `libc`.

Un code assembleur peut également profiter de cette bibliothèque. Le plus simple en ce cas est d'assembler le programme via le compilateur du C qui utilise `ld` avec les bonnes options pour lier dynamiquement l'exécutable produit avec la `libc`.

```
nasm -f elf32 Main.asm  
cc -m32 -o Exec Main.o
```

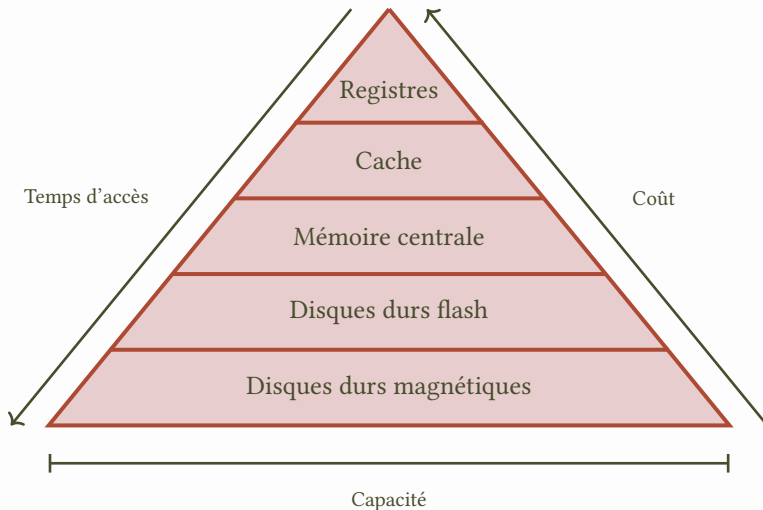
Lors de cette compilation, un fichier supplémentaire est introduit contenant le symbole `_start` (le point d'entrée par défaut) et contenant du code initialisant des composant de la bibliothèque standard du C.

Axe 3 : améliorations

Mémoires

Pipelines

Les trois dimensions de la mémoire



Les caractéristiques principales des mémoires — en plus des dimensions capacité, temps d'accès et coût — sont

- la **volatilité** (présence obligatoire ou non de courant électrique pour conserver les données mémorisées);
- le nombre de **réécritures** possibles;
- le **débit de lecture**;
- le **débit d'écriture**.

Mémoires

Voici les caractéristiques de quelques mémoires :

- **registre** : volatile, réécriture possible, temps d'accès de l'ordre de 1 ns ;
- **mémoire morte** (ROM, **Read Only Memory**) : non volatile, pas de réécriture possible, temps d'accès de l'ordre de 150 ns ;
- **mémoire vive** (RAM, **Random Access Memory**) : volatile, réécriture possible, débit de lecture/écriture de l'ordre de 8 Gio/s, temps d'accès de l'ordre de 10 ns ;
- **mémoire flash** : non volatile, réécriture possible (de l'ordre de 10^5 fois), débit de lecture/écriture de l'ordre de 500 Mio/s, temps d'accès de l'ordre de 0.1 ms ;
- **mémoire de masse magnétique** : non volatile, réécriture possible, débit de lecture/écriture de l'ordre de 100 Mio/s, temps d'accès de l'ordre de 10 ms.

Principes de localité

Problématique : comment optimiser les accès mémoire ?

On se base sur les deux principes raisonnables suivants.

Localité temporelle : si une zone de la mémoire a été considérée à un instant t donné, elle a une forte chance d'être reconsidérée à un instant t' proche de t .

La localité temporelle s'observe par exemple dans les **boucles** : la variable de contrôle de la boucle est régulièrement lue/modifiée.

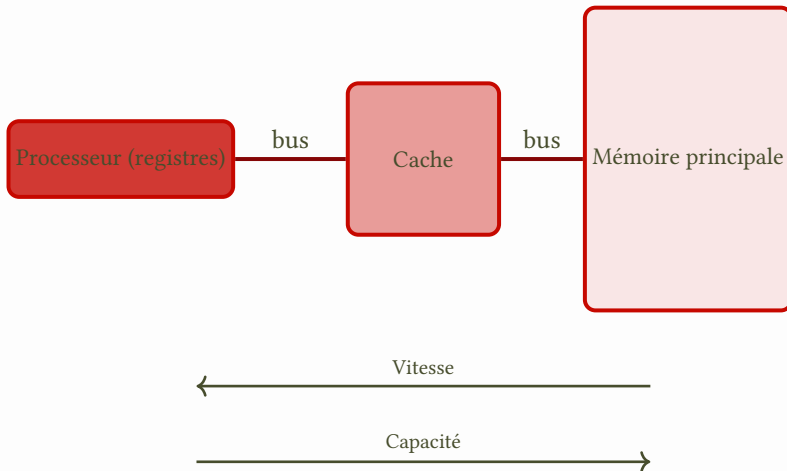
Localité spatiale : si une zone de la mémoire à une adresse x donnée a été considérée, les zones de la mémoire d'adresses x' avec x' proches de x ont une forte chance d'être considérées.

La localité spatiale s'observe dans la manipulation de **tableaux** ou encore de la **pile** : les données sont organisées de manière contiguë en mémoire.

Ces deux principes impliquent le fait qu'à un instant donné, un programme **n'accède qu'à une petite partie de son espace d'adressage**.

Organisation de la mémoire

La mémoire est organisée comme suit :



La mémoire cache dans l'organisation de la mémoire

La **mémoire cache** est une mémoire très rapide en lecture et en écriture (entre les registres et la mémoire centrale).

Elle est constituée de plusieurs **couches** : L1, L2, L3 où L1 a la plus petite capacité (de l'ordre de 128 Kio) et la plus grande vitesse, et L3 a la plus grande capacité (de l'ordre de 10 Mio) et la plus petite vitesse.

Il fonctionne de la manière suivante :

1. le processeur demande à lire une donnée en mémoire ;
2. la mémoire cache, couche par couche, est interrogée :
 - 2.1 si elle contient la donnée, elle la communique au processeur ;
 - 2.2 sinon, la mémoire principale est interrogée. La mémoire principale envoie la donnée vers la mémoire cache qui l'enregistre (pour optimiser une utilisation ultérieure) et la transmet au processeur.

Organisation du cache

La mémoire cache est organisée en **lignes**. Chaque ligne est en général constituée de 32 octets.



- V est un **bit de validité** : il informe si la ligne est utilisée.
- Indicateur permet de connaître l'**adresse** en mémoire principale des données représentées par la ligne.
- Mot_1, Mot_2, Mot_3 et Mot_4 contiennent des **données**.

La ligne est la plus petite donnée qui peut circuler entre la mémoire cache et la mémoire principale.

Le **mot** est la plus petite donnée qui peut circuler entre le processeur et la mémoire cache. Celui-ci est en général composé de 4 octets.

Stratégies de gestion de la mémoire cache

Invariant important : toute donnée représentée dans la mémoire cache est également présente dans la mémoire centrale. C'est la **propriété d'inclusion**.

Ceci implique que lorsqu'une donnée qui figure dans la mémoire cache est modifiée, il faut également modifier l'instance de la donnée située en mémoire principale.

Il existe deux stratégies pour cela :

1. **l'écriture simultanée** : lorsqu'une ligne du cache est modifiée, la mémoire principale est immédiatement mise à jour. Cette méthode est lente.
2. La **recopie** : lorsqu'une ligne du cache est modifiée, on active un drapeau qui la signale comme telle et la mémoire principale n'est mise à jour que lorsque nécessaire (juste avant de modifier à nouveau la ligne du cache en question).

Stratégies d'écriture dans la mémoire cache

Il existe plusieurs stratégies d'écriture dans la mémoire cache, plus ou moins complexes et plus ou moins rapides :

- l'organisation **à correspondance directe** : à toute donnée est associée une position dans la mémoire cache (par un calcul modulaire).

Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, ce dernier est écrasé.

- L'organisation **totale associative** : une donnée peut se retrouver à une place quelconque dans la mémoire cache.

Si une donnée doit être écrite dans la mémoire cache à un endroit déjà occupé, une position aléatoire est générée pour tenter de placer la nouvelle donnée.