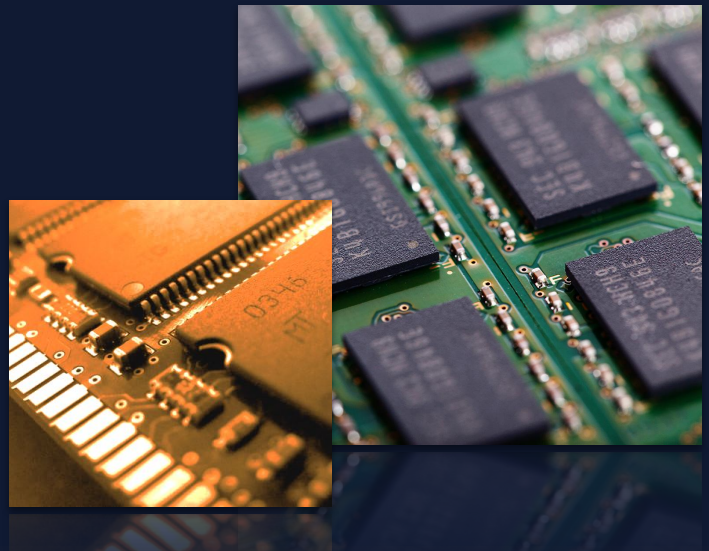
ISART DIGITAL

PARIS

ISART DIGITAL PARIS

Utilisation de la mémoire



Sommaire

- Définition
- Hiérarchie de la mémoire
- Alignement mémoire
- Le format d’un exécutable
- La Stack et le Heap
- Malloc

Utilisation de la mémoire

Définition

Un **ordinateur** a deux **caractéristiques principales** :

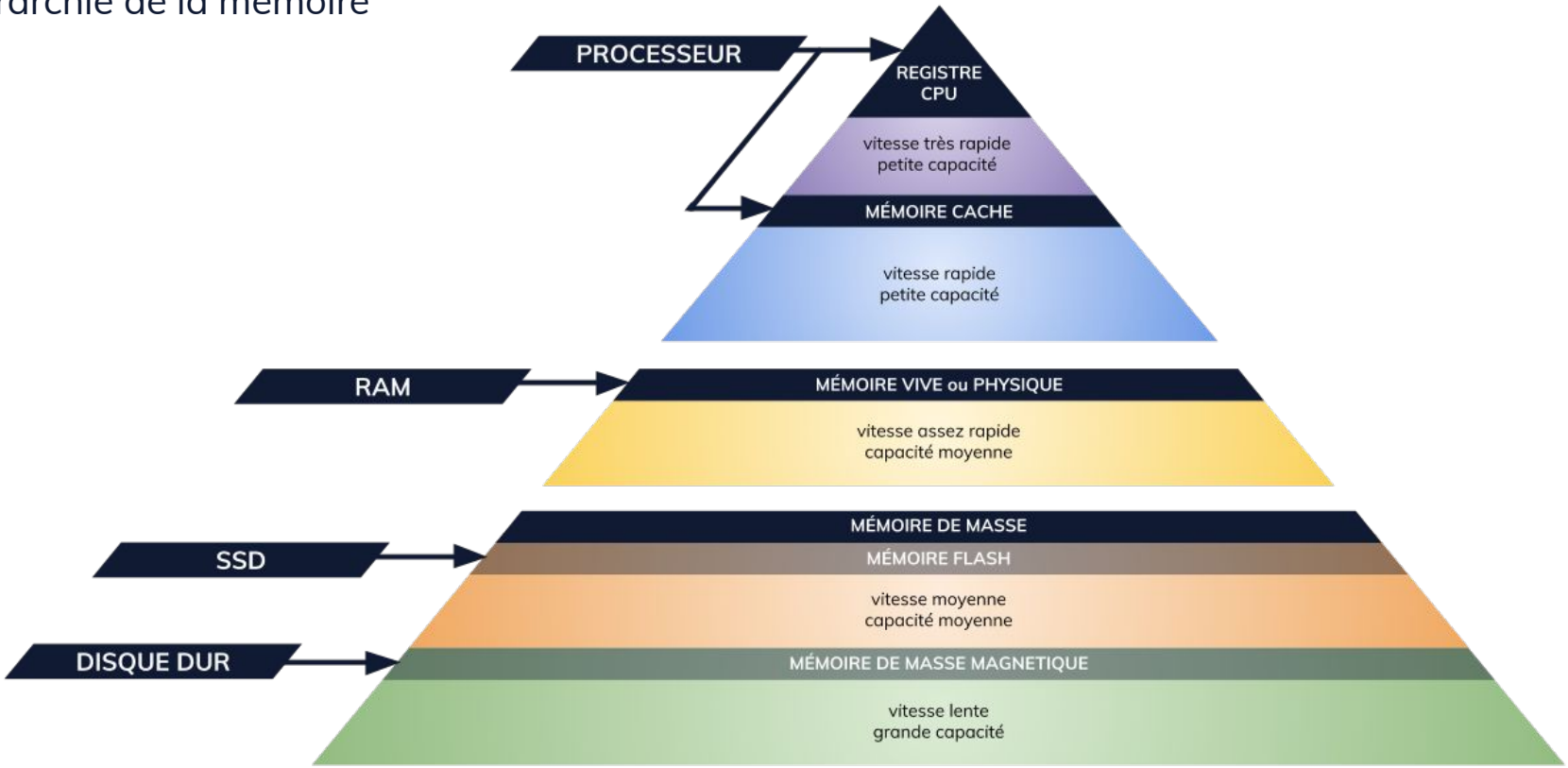
- La **vitesse de traitement**
- La **capacité de mémorisation**

La **mémoire informatique** est un composant essentiel permettant de stocker l'information.

Il y a différentes catégories de mémoire, influençant la vitesse d'accès et la capacité.

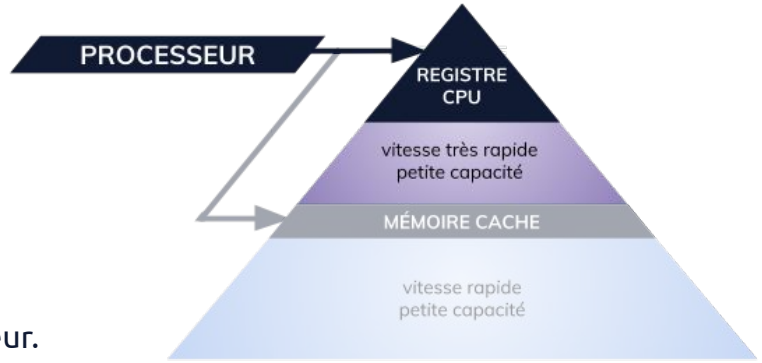
Utilisation de la mémoire

Hiérarchie de la mémoire



Utilisation de la mémoire

Hiérarchie de la mémoire



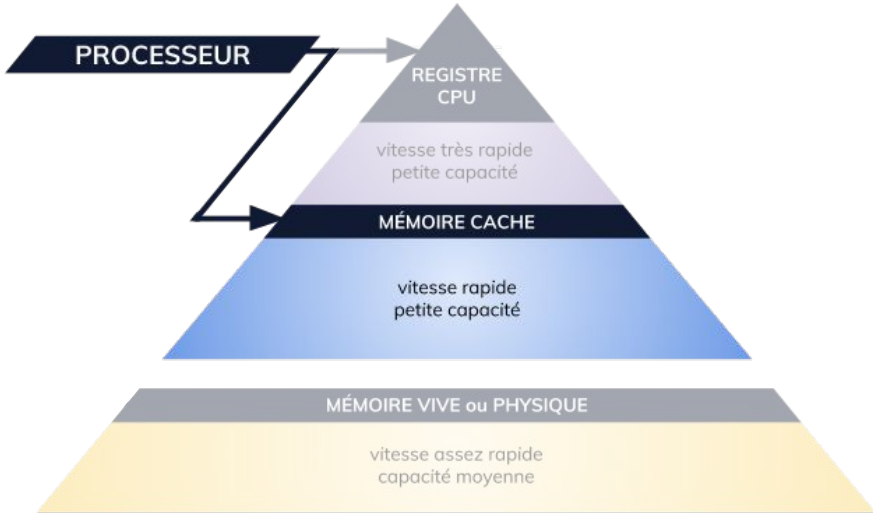
Le Registre CPU est la **mémoire la plus rapide** de l'ordinateur.

L'**accès** aux données est donc **très rapide** mais la **place disponible** est très **limitée**.

Elle sert principalement à **stocker des valeurs opérandes** *(arguments / paramètres)* et leurs résultats intermédiaires.

Utilisation de la mémoire

Hiérarchie de la mémoire



La Mémoire cache sert de **tampon** entre la **Mémoire Vive** et le **registre CPU** afin d'optimiser les temps d'accès aux informations.

L'**accès** aux données est donc **rapide** mais la **place disponible** est très **limitée**.

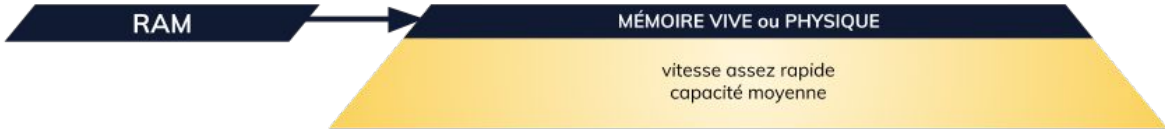
Le processeur y **stocke les informations** dont il a **le plus fréquemment besoin**.

Utilisation de la mémoire

Hiérarchie de la mémoire

La Mémoire vive ou **physique** est le composant principal pour stocker les informations utilisé par le Processeur.

L'**accès** aux données est **assez rapide** et la **place disponible** est **moyenne**.



Lorsque l'on souhaite **exécuter** un **programme**, celui-ci est préalablement **chargé** dans la **Mémoire Vive**.

Utilisation de la mémoire

Hiérarchie de la mémoire

La Mémoire de masse flash possède les mêmes caractéristiques que la mémoire vive, cependant, les **informations** qui y sont **stockées perdurent** même **après l'arrêt de l'ordinateur**.
(mémoire non volatile)

L'**accès** aux données et la **place disponible** est **moyenne**.

Sa **consommation** est **faible** par rapport au disque dur et les données stockées ne risque pas d'effacement mécanique.
(rayure, choc, etc.)

Utilisation de la mémoire

Hiérarchie de la mémoire

La Mémoire de masse magnétique est une mémoire également **non volatile**.

L'**accès** aux données est lente mais la **place disponible** est **grande**.

La **structure mécanique** du disque dur est **fragile** et peut progressivement **entraîner** des **pertes de données**.
(il faut éviter de le déplacer lorsqu'il est allumé)

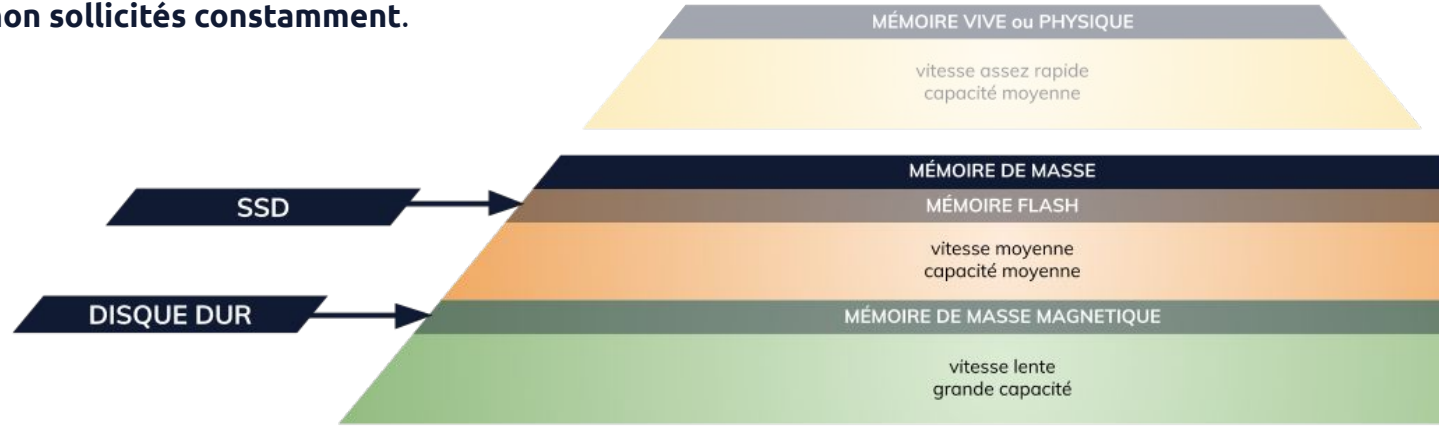
Utilisation de la mémoire

Hiérarchie de la mémoire

Il existe également un **mécanisme de Mémoire virtuelle** permettant d'**utiliser**

la **Mémoire de masse** comme **extension** de la **mémoire vive**.

Elle se concrétise par un **fichier d'échanges** *(fichier swap)*,
contenant des **données non sollicités constamment**.



Utilisation de la mémoire

Alignement de la mémoire

```
// 32 bits values
typedef struct Foo
{
   unsigned int   mUInt;  // 4 bytes
   float          mFloat; // 4 bytes
   int            mSInt;  // 4 bytes
   char           mChar;  // 1 byte
} Foo;
```

Il est important de se rappeler que la **déclaration** d'une **structure n'alloue pas de mémoire !**

La **mémoire** est **allouée** au moment de l'**instanciation** d'une variable du type de la structure.

RAPPEL

1 byte = 1 octet
= 8 bits

4 bytes = 4 x 8 bits
= 32 bits

Utilisation de la mémoire

Alignement de la mémoire

```
struct inefficientPacking
{
    unsigned int    mUI;    // 4 bytes
    float           mF;     // 4 bytes
    unsigned char   mByte;  // 1 byte
    int             mI;     // 4 bytes
    bool            mBool;  // 1 byte
    char*           mP;     // 4 bytes
}
```

Que se passe-t-il lorsque les petits membres sont entrecoupés de membres plus importants ?

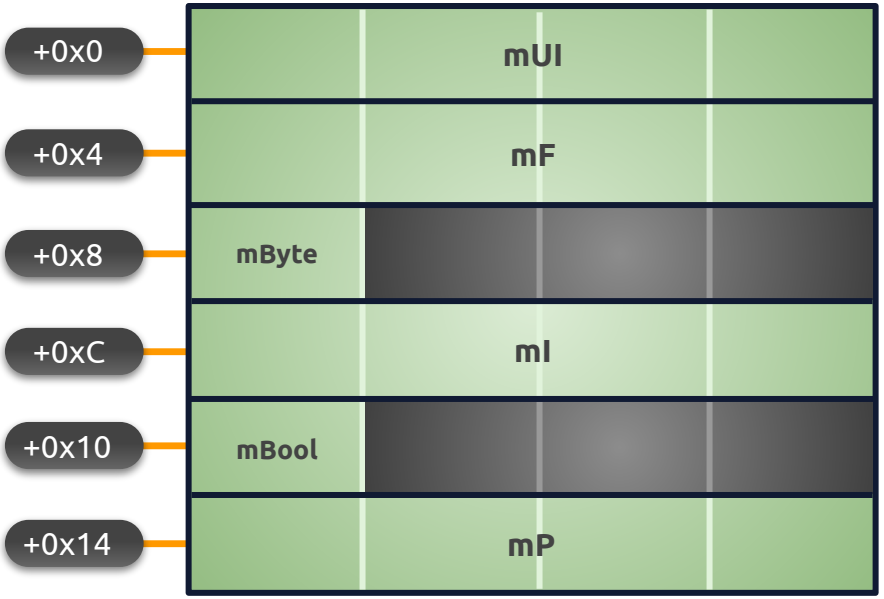
Utilisation de la mémoire

Alignement mémoire

Par défaut le **compilateur** va laisser des **espaces vides** dans l'**alignement mémoire**.

Cela permet au **CPU** de **lire/écrire** dans la **mémoire** de façon **plus performante**.

L'**alignement naturel** d'une **donnée** est **déterminé** par l'**adresse mémoire** où elle peut être **affecté**.
Celle-ci doit être un **multiple de sa taille**.
(généralement une puissance de 2)



Utilisation de la mémoire

Alignement mémoire

Ainsi on peut remarquer que la taille d'une structure sera différente en fonction de l'organisation de celle-ci :

```
// 8 bits values
typedef struct s1
{
   char        c1; // 1 byte
   char        c2; // 1 byte
   short int   si; // 2 bytes
   float       f;  // 4 bytes
} t1;
```

La structure t1 prendra 8 bytes.


```
// 12 bits values
typedef struct s2
{
   char        c1; // 1 byte
   float       f;  // 4 bytes
   short int   si; // 2 bytes
   char        c2; // 1 byte
} t2;
```

La structure t2 prendra 12 bytes.

Utilisation de la mémoire

Alignement mémoire

Pour résumer :

- Une donnée avec une **taille de 1 octet** réside à **n'importe quelle adresse mémoire**.
- Une donnée avec une **taille de 2 octets** réside aux **adresses paires**. *(0x0, 0x2, 0x4, etc)*
- Une donnée avec une **taille de 4 octets** réside aux **adresses multiples de 4**. *(0x0, 0x4, 0x8, etc)*
- Une donnée avec une **taille de X octets** réside aux **adresses multiples de X**.
- L'alignement d'une structure dans son ensemble est égal à la plus grande exigence d'alignement parmi ses membres.

On peut utiliser la directive **#pragma pack** afin de spécifier alignement que l'on souhaite.

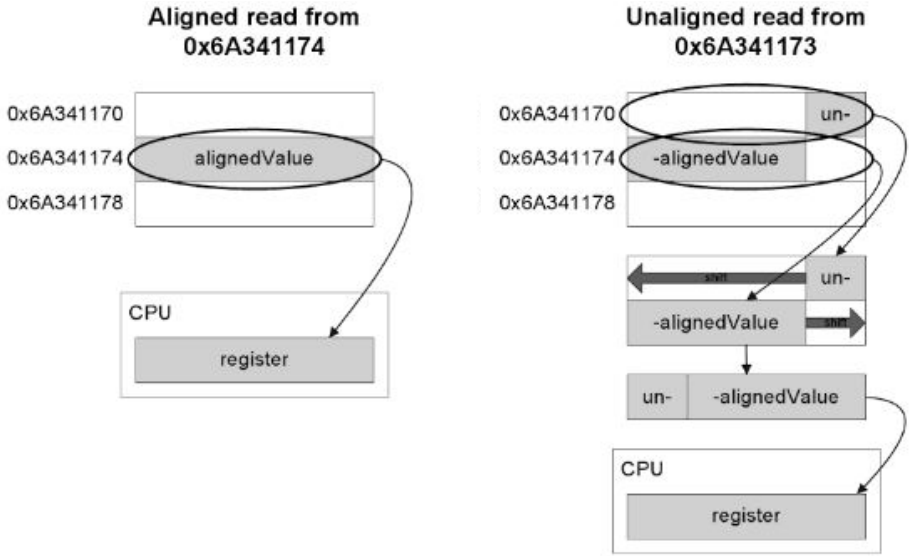
Utilisation de la mémoire

Alignement mémoire

L'alignement mémoire est **très important** pour les **performances** d'**écriture** et de **lecture** du CPU.

- Lire un int *(4-octet)* à l'adresse *0x6A341174* est facile, le contrôleur mémoire obtient la valeur instantanément.

- Lire un int *(4-octet)* à l'adresse *0x6A341173* est plus complexe.
  Le contrôleur mémoire va commencer à lire à l'adresse *0x6A341170* pour avoir une partie du int, puis va se déplacer à l'adresse *0x6A341174* pour avoir la fin.



Utilisation de la mémoire

Le format d'un exécutable

ELF *(Executable and Linkable Format)* est un **format de fichier binaire** utilisé pour l'enregistrement de **code compilé**.

(objets, exécutables, bibliothèques de fonctions)

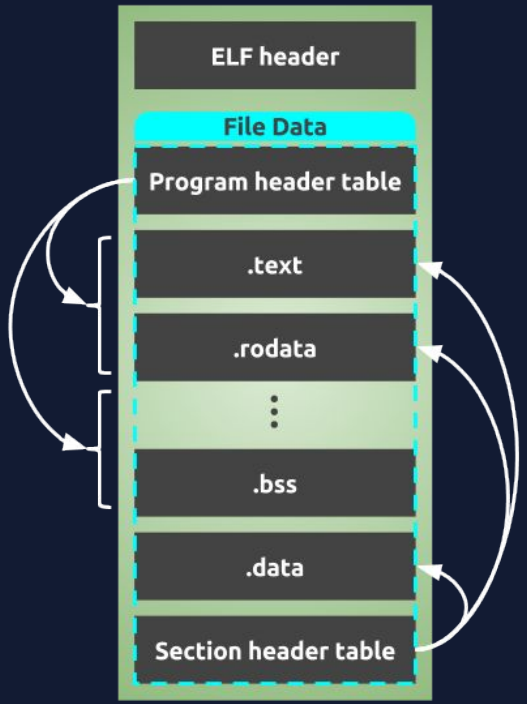
Il est créé par le compilateur. *(gcc, g++, etc)*

Il est **utilisé** sur la plupart des **systèmes d'exploitation UNIX**.

Windows utilise le format PE. *(Portable Executable)*

Mac OS X utilise le format Mach-O.

Il est composé de plusieurs parties dont les deux principales sont le **ELF header** et le **File Data**.

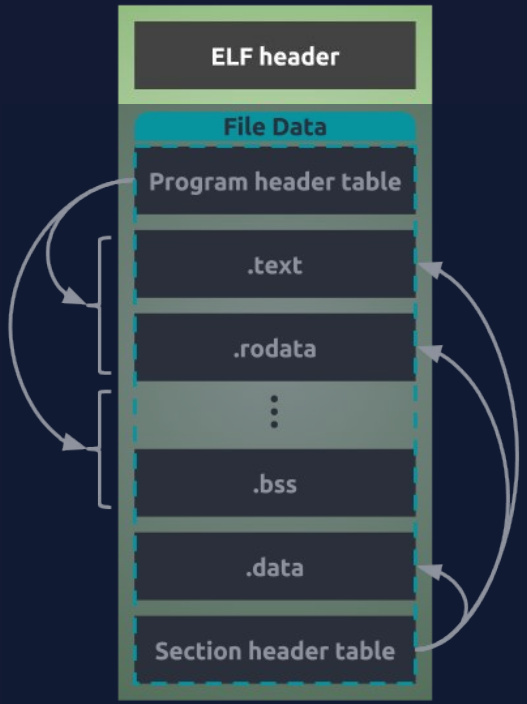


Utilisation de la mémoire

Le format d'un exécutable

Le **ELF Header** permet de **définir** si le **programme** utilise des adresses **32 ou 64 bits**.

Il **contient** les **paramètres** pour **exécuter** le **programme**, ainsi que le **nombre** de **sections** du **fichier**.



Utilisation de la mémoire

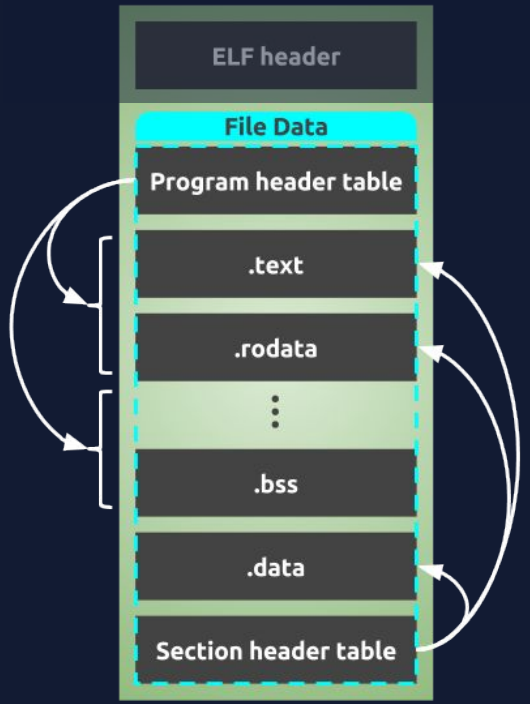
Le format d'un exécutable

Le **File Data** englobe les autres parties du fichier binaire :

- Le **Program header table**,
  - Décrit et localise les différents **segments** de code à charger *(informations nécessaires au moment de l'exécution du fichier)*
  - Les localise avec leur adresse

- Le segment **.text** contient le **code machine** pour toutes les **fonctions** définies par le programme

- Le segment **.rodata** contient
  - l'ensemble des **variables** en **read-only** *(const)*
  - cependant, la plupart des const int sont insérées dans le code machine



Utilisation de la mémoire

Le format d'un exécutable

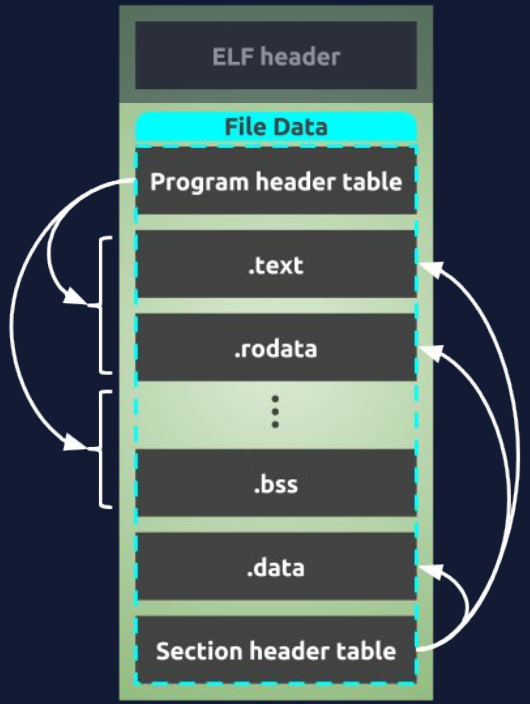
Le **File Data** englobe les autres parties du fichier binaire :

- Le segment **.bss** contient, les **variables globales** et **static** **non initialisées**, **initialisées à 0** ou n'ayant **pas d'initialisation explicite**
  ex : float gUninitializedGlobal;

- Le segment **.data** contient,
  - les **variables globales** et **static initialisées**
  - une **zone** en **lecture seul** et une **zone** en **lecture-écriture**

  ex : float gInitializedGlobal = 2.0f;

- Le **Section header table**,
  - **Répertorie** les différentes **segments** du fichier binaire
    (données importantes pour la liaison et la relocalisation)



Utilisation de la mémoire

La Stack et le Heap

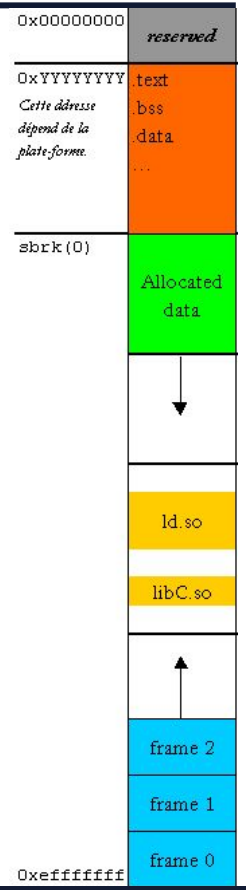
Au sein d'une **application**, la **mémoire disponible** est découpée en **2 grandes zones** :

- La **Stack** (pile) est **gérée automatiquement** par le **compilateur**

  Au moment de la création d'une variable , le compilateur va automatiquement
  lui réserver de la mémoire dans la stack.
  Elle sera vidée automatiquement, le plus rapidement possible, ce qui permet une bien meilleur optimisation.

  Son utilisation est à privilégier !

- La **Heap** (tas) doit être **explicitement demandé** par le **programmeur** *(via l'**allocation dynamique**)*



Utilisation de la mémoire

La Stack et le Heap

La **Stack** n'a pas une taille infinie, le **compilateur** y **libère** de la **mémoire dès que possible**.

Elle **se compose** de **frames**, chacune correspond à une **fonction**.

Chaque **frame** contient :

- Les **variables locales**
- L'**adresse de retour**
- Une **sauvegarde** des **paramètres** passée **en entrée**
- Une **copie** de **la zone du registre CPU** utilisé *(complète ou partielle)*

Une **frame** est donc **valide tant** que l'**on ne sort pas** de la **fonction** auquelle elle est **associée**.
(le cas contraire entraîne une erreur de segmentation)

Utilisation de la mémoire

La Stack et le Heap

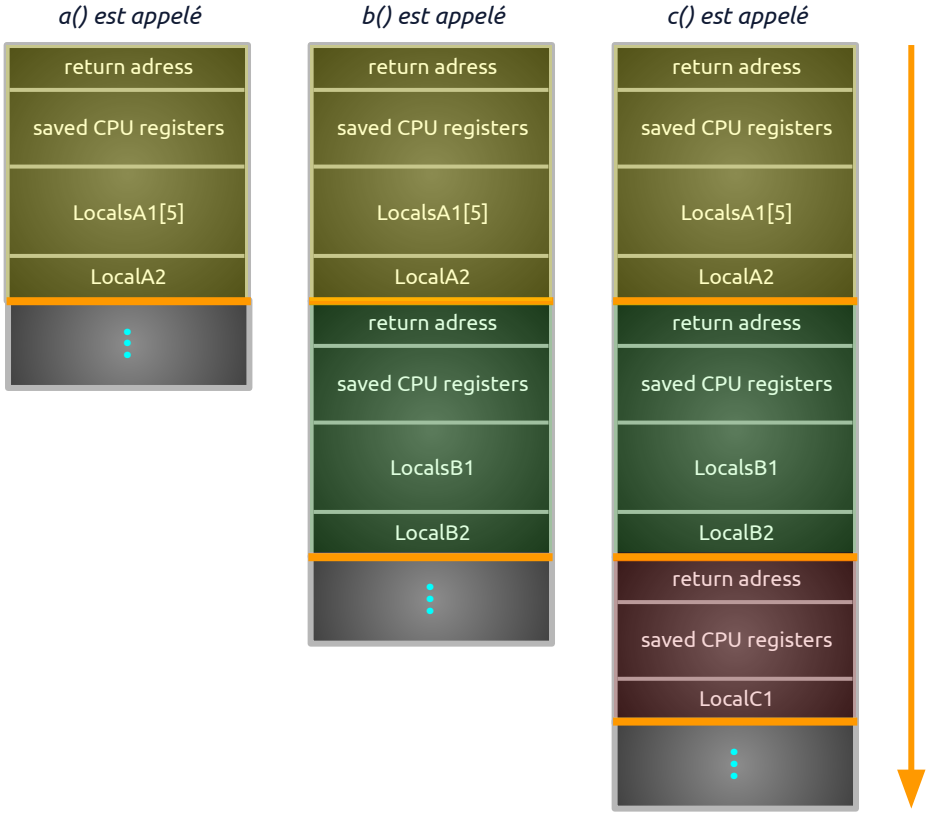
Exemple de frames au sein de la **Stack** :

```
void    c()
{
    unsigned int       localC1;
}

float   b()
{
    float      localB1;
    float      localB2;
    c();

    return localB1;
}

void    a()
{
    unsigned int       localsA1[5];
    float              localA2 = b();
}
```



Utilisation de la mémoire

La Stack et le Heap

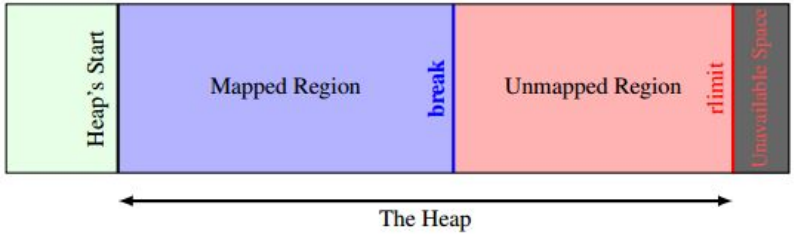
Le **Heap** n'a pas une taille infinie, c'est au **développeur** de bien penser à y **libérer** de **la mémoire dès que possible**.

Il permet d'**allouer** de la **mémoire manuellement** en **maîtrisant** sa **durée de vie**.

Il est utilisé par les fonctions **malloc/free** pour **allouer/libérer** de la mémoire dynamiquement.

<u>Le Heap est divisé en plusieurs parties :</u>

- Le **Heap's start** → Là ou il commence
- La **Unmapped Region** → L'espace disponible
- La **Mapped Region** → La mémoire déjà utilisé *(précédentes allocations)*
- Le **break** → Là ou il se termine actuellement
- Le **rlimit** → Jusqu'où il peut aller



Utilisation de la mémoire

La Stack et le Heap

Exemple de programme utilisant :

La Stack

```
typedef struct Foo Foo;

// As a local variable on the program stack.
void programStackExample
{
        Foo             localFoo;
}

// As a global, file-static or function-static.
Foo                     gFoo;
static Foo              sFoo;

void segmentsExample
{
        static Foo      sLocalFoo;
}
```

Une fois la struct/class déclarée, on peut l'allouée.

Ici, on ne l'alloue pas dynamiquement.

Donc placé dans le Data Segment ou BSS segment.

Ici, on fait une allocation dynamique.

Le pointeur lui-même peut être global, static.

Il contient l'adresse de la data elle-même.

Le Heap

```
typedef struct Foo Foo;

Foo*    gpFoo = NULL;

void heapFunction()
{
    // Allocate a Foo instance from the heap
    gpFoo = malloc(sizeof(Foo));

    //local pointer to Foo
    Foo* pAnotherFoo =  malloc(sizeof(Foo));

    // Allocate a pointer to a Foo
    // from the heap
    Foo** ppFoo = malloc(sizeof(Foo*));

    (*ppFoo) = pAnotherFoo;
}
```

Utilisation de la mémoire

Malloc

Rappel de la fonction Malloc:

- Elle **permet allouer** un **nombre d'octets en mémoire** dans le **Heap**.
- Elle **retourne** un **pointeur sur l'espace alloué**, **ou NULL** si elle échoue.
- **Aucun autre** appel de **malloc ne peut écraser** cette **mémoire précédemment alloué tant que** vous ne l'avez **pas free**.
- **Vous devez free votre mémoire** dès que vous n'en avez **plus besoin** !

La fonction **Malloc** fait appelle à **trois** *(éventuelles)* **fonctions** :

- brk()
- sbrk()
- mmap()

Utilisation de la mémoire

Malloc

La fonction **malloc joue avec la limite break** du **Heap**.

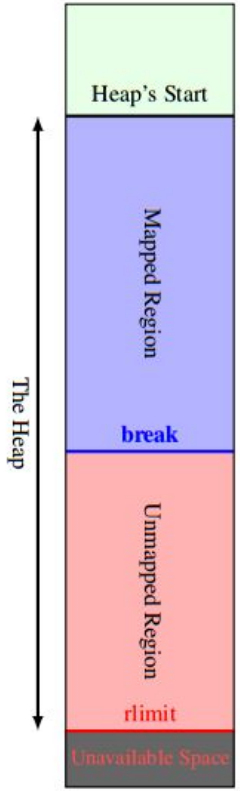
Lorsqu'on l'appelle, la fonction **vérifie** si de l'**espace mémoire** est **disponible** dans la **Mapped Region**.

Si ce n'est pas le cas, elle **déplace** le segment **break** afin d'**agrandir** la **Mapped Region**.

Quand on appelle la fonction **free**, celle-ci **rend disponible** la **zone mémoire** en question dans la **Mapped Region**.

Pour **déplacer** le segment **break**, on utilise **deux fonctions** de la librairie **unistd.h**:

- `int brk(const void* addr);`
- `void* sbrk(intptr_t incr);`



Utilisation de la mémoire

Malloc

- La fonction **brk** permet de **positionner** le segment **break** à l'**adresse** reçu **en paramètre**.

  Elle **retourne 0** lorsqu'elle **réussie**, **-1** en cas d'**échec**.

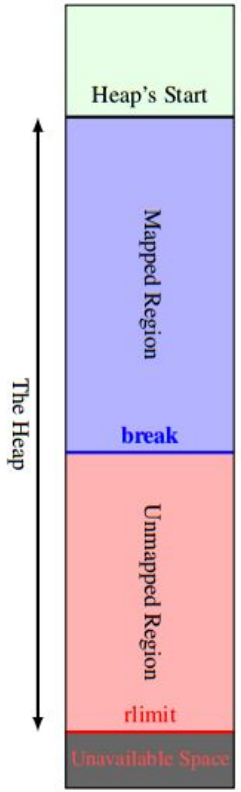
  Exemple : `brk(addr_data);`

- La fonction **sbrk** permet de **déplacer** le segment **break** du **nombre d'octet** reçu en paramètre.

  Elle **retourne** un **pointeur** sur l'**adresse de break**.

  Exemple :
  ```
  void*   newZone = sbrk(sizeof(int));
  ```

  Obtenir emplacement courant de break :
  ```
  void*   currentLimit = sbrk(0);
  ```



Utilisation de la mémoire

Malloc

- La fonction **mmap**, quand à elle, permet d'**allouer** une **grande quantité de mémoire**, directement **par page**.
- La fonction **munmap** permet alors de **libérer** ces **grandes quantités de mémoires allouées**.

Accéder à une **adresse après** la limite **break** provoque une **erreur de bus**.

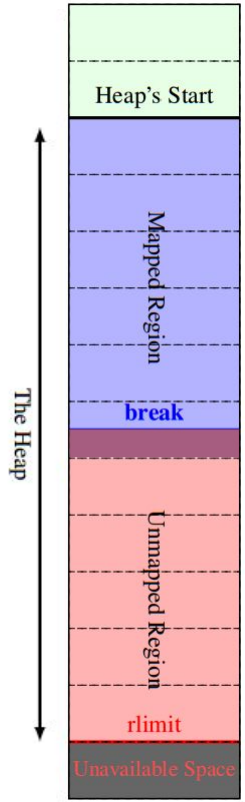
La **mémoire physique** et la **mémoire virtuelle** sont **découpés en pages** *(frames pour la mémoire physique)* de taille fixes *(la plupart du temps)*.

La **taille** d'une **page** est **beaucoup plus grande** qu'un **octet**. *(4096 octets en général)*

Le segment **break** ne s'arrête pas forcément à la limite d'une page !

Remarque :

L'**espace entre** le segment **break** et la **fin de la page** est **accessible** **mais** **on ne peut pas savoir** quand-est-ce que l'on **atteint** la **fin** de la **page**.



Utilisation de la mémoire

Malloc

Lorsqu'on **appelle** la fonction **free**, on **rend disponible la mémoire précédemment alloué**.

Ainsi lorsqu'on **appelle** la fonction **malloc**, il est logique qu'elle **parcourt**
ses **précédentes allocations** pour **vérifier** si
l'**une d'entre elle peut être remplacée** par la nouvelle.

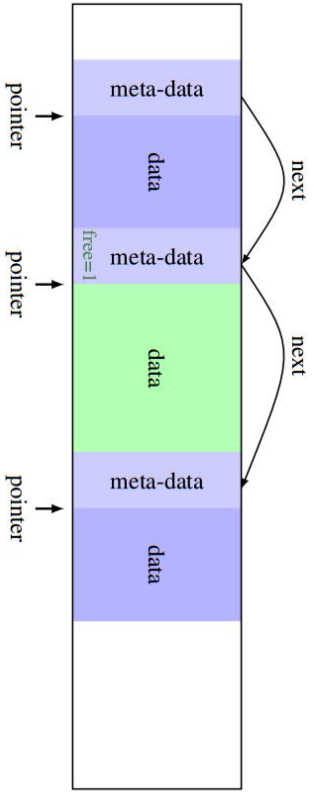
Cela permet d'**éviter** d'**agrandir** la **Mapped Region** inutilement.

Pour mettre en place ce **système**, on utilise une **liste chaînée** de **meta-data**.

Ces **meta-datas** sont une structure contenant chacun *(au minimum)* les **informations** sur :

- Les données allouées *(les concernant)*
- La disponibilité du block *(si on peut l'écraser)*
- Un pointeur sur le block suivant

L'**espace mémoire occupé** par ces **meta-datas n'est pas transmise** par la fonction **malloc**,
uniquement celui des données allouées.



ISART DIGITAL

www.isartdigital.com

60 bd Richard Lenoir 75011 Paris
+33 1 48 07 58 48