La Memoire
(Chapitre 3)

Francesco Bronzino
ArchiSys

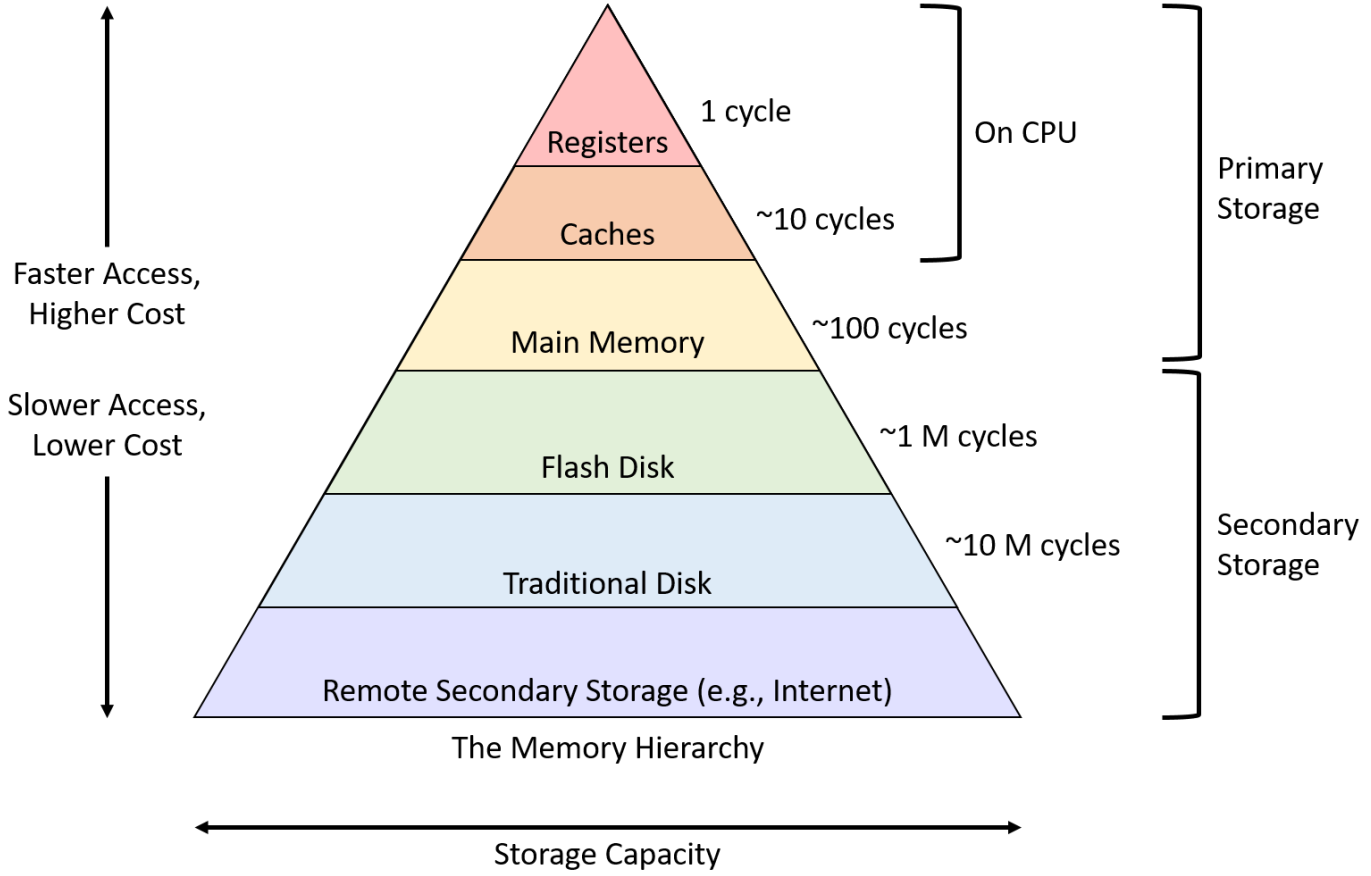
Sujets

1. La mémoire dans les systèmes à processeur
2. Approches historiques
3. La mémoire virtuelle
4. Remplacement des pages
5. Disposition de la mémoire
6. Loader, bibliothèques et pages partagées
7. Gestion de la mémoire dans Bash

La mémoire dans les systèmes à processeur

La mémoire dans les systèmes à processeur

Les systèmes à processeur possèdent de nombreuses *mémoires*.



La mémoire dans les systèmes à processeur

La tâche du système d'exploitation est de gérer l'utilisation de la mémoire par les processus, avec les objectifs suivants :

- Haute performance : utiliser la mémoire aussi vite que possible.
- Isolation entre les processus : éviter les problèmes de sécurité et de stabilité.
- Simplicité de programmation : on souhaite que le système d'exploitation soit *transparent* pour le programmeur.

Approches historiques

Au départ, il n'y avait pas de système d'exploitation : l'ordinateur exécutait un programme à la fois.

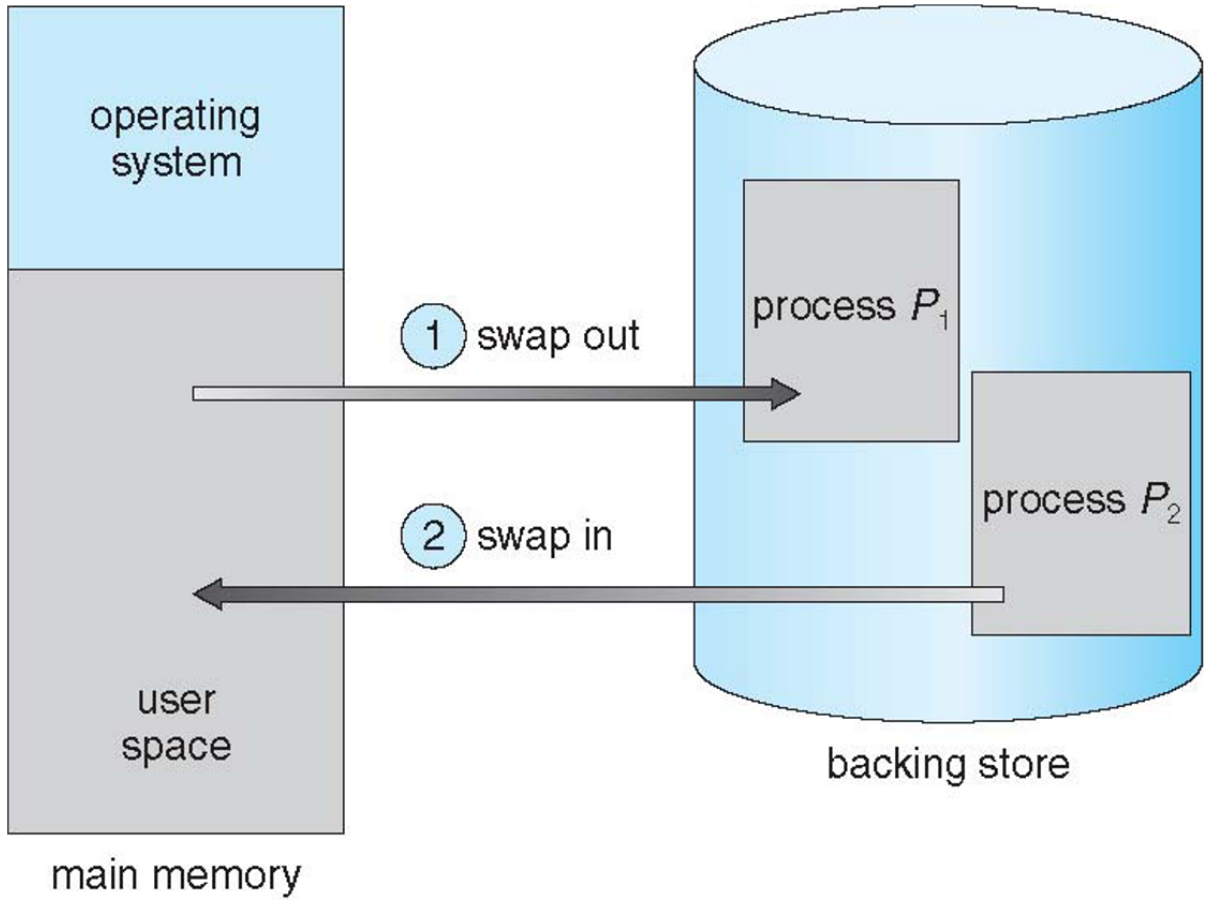
- Un programme pouvait accéder à n'importe quel emplacement de la mémoire

Dans le tout premier système d'exploitation, c'était toujours le cas.

- Le système d'exploitation avait la tâche de remplacer toute la mémoire à chaque *Context Switching*.

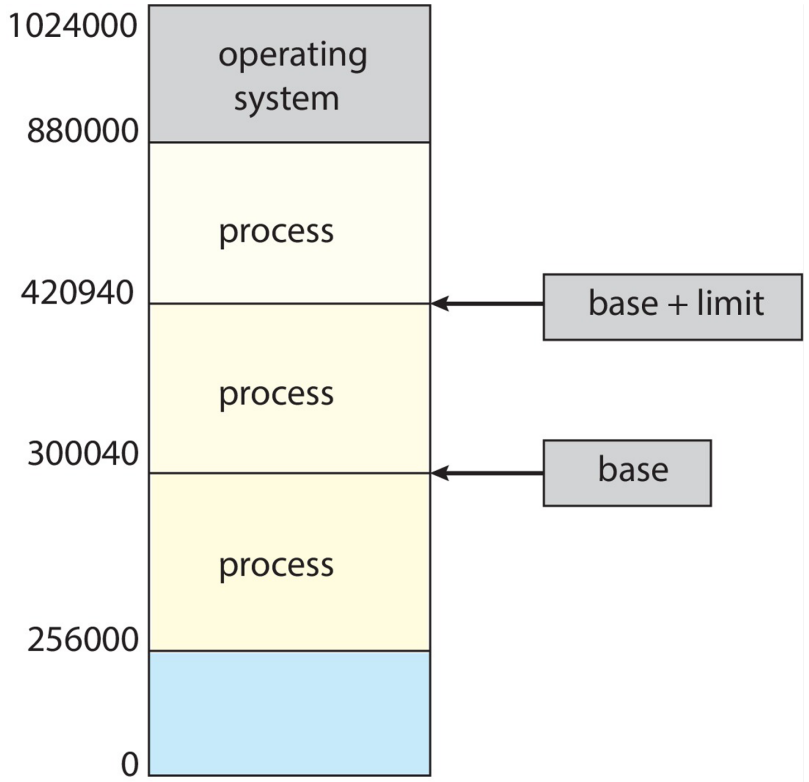
Approche a remplacement

L'OS remplace complètement la mémoire principale du programme **en cours d'exécution**.



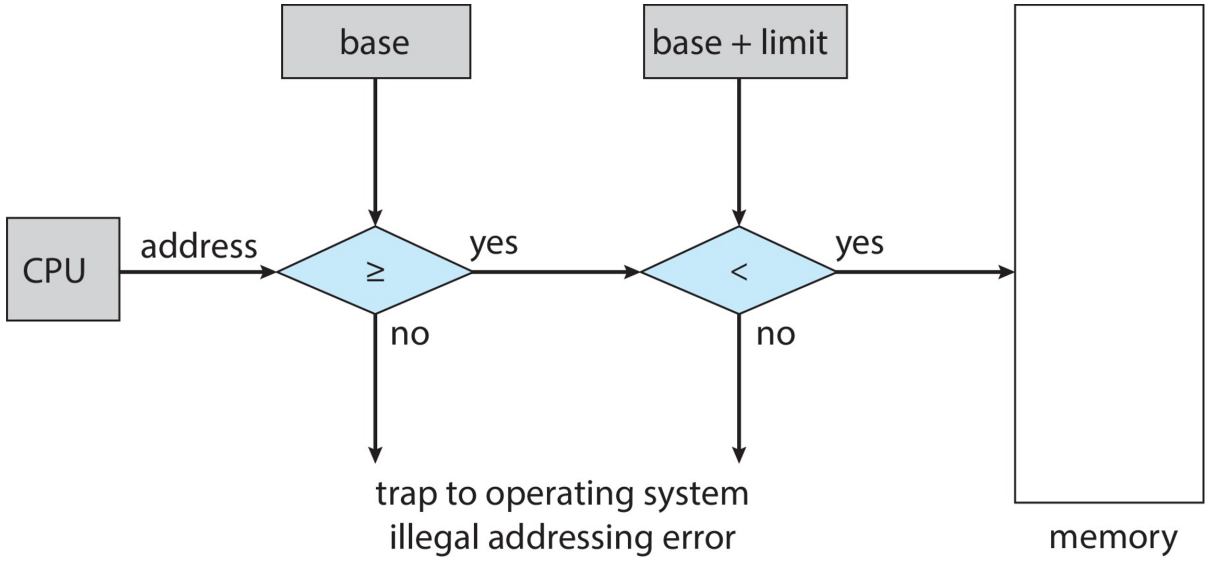
Approche Base et Limite

- Plusieurs processus partagent la mémoire
- Ils ne sont autorisés à accéder qu'à une seule zone de mémoire



Approche Base et Limite

- Le CPU possède des registres *Base* et *Limite*, définis par le système d'exploitation.
- Il ne permet aux processus que de sortir les adresses autorisées



Approche Base et Limite

Avantages :

- Permet des processus multiples
- Un processus ne peut pas accéder à la mémoire des autres.

Inconvénients :

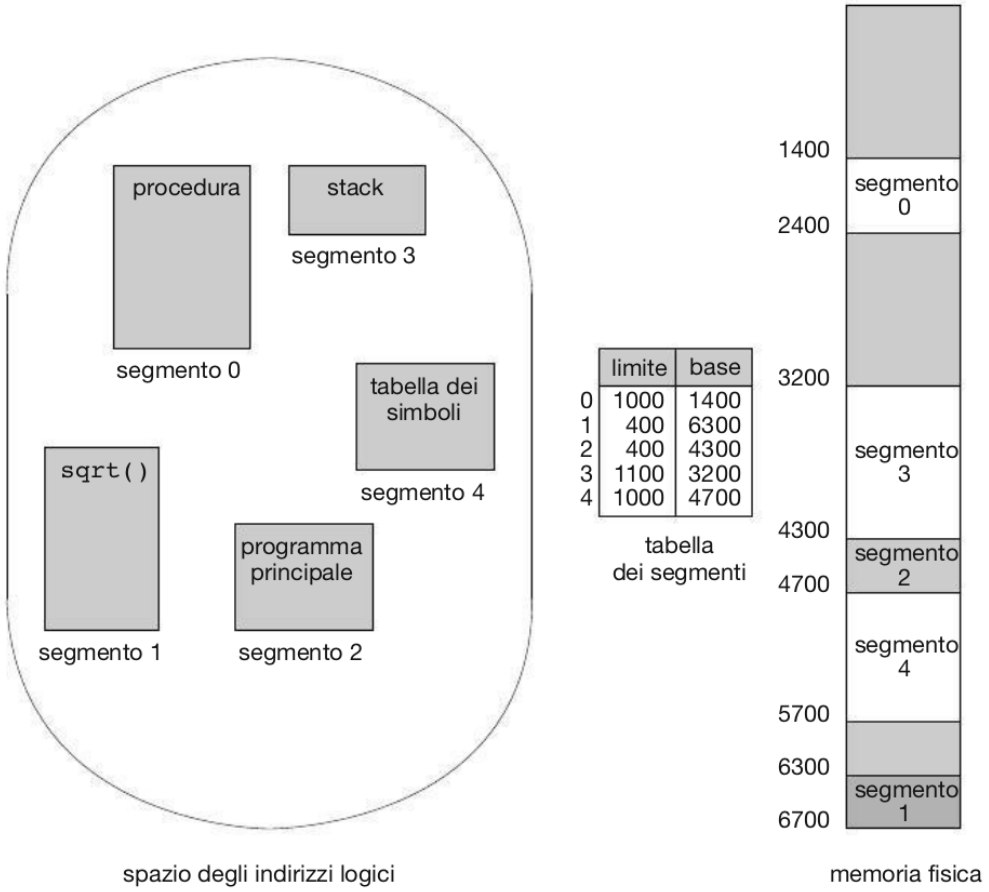
- Allocation contiguë : risque de gaspillage de mémoire
- Peu de flexibilité

Approche de la mémoire segmentée

Comme Base + Limite, mais chaque processus a plus de **segments** disponibles.
Habituellement, chaque segment a des objectifs différents :

- Segment de code
- Segment de données (variables globales et constantes)
- Segment de Stack (variables de fonction)

Approche de la mémoire segmentée



Approche de la mémoire segmentée

Avantages :

- Assez flexible

Inconvénients :

- Les segments de longueurs différentes sont difficiles à gérer.
- Introduit la fragmentation comme dans *Base* + *Limite*.

Elle a cependant été largement utilisée dans les années 1980 et 1990.

Approche par pagination

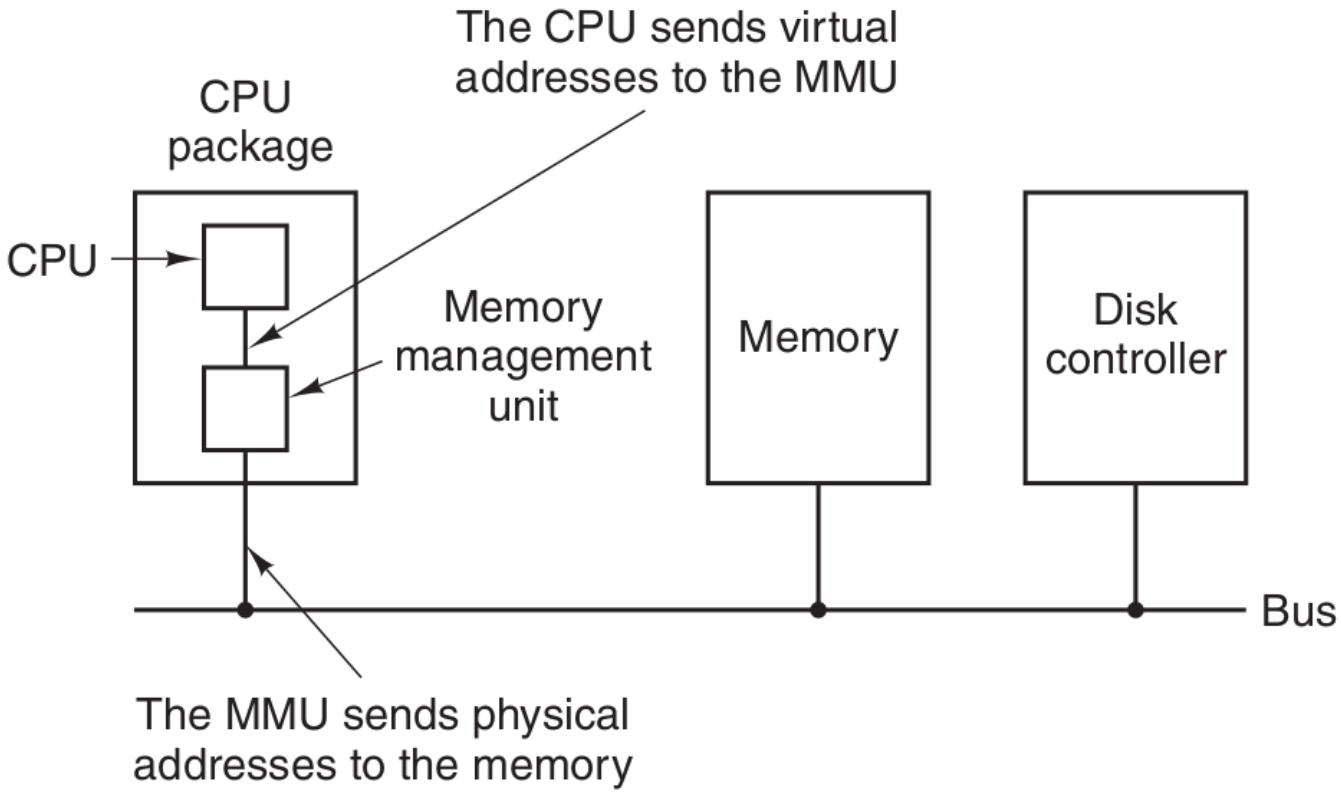
Un processus produit des **adresses virtuelles**.

- Un module matériel appelé **Unité de gestion de la mémoire** les traduit en **Adresses physiques**.

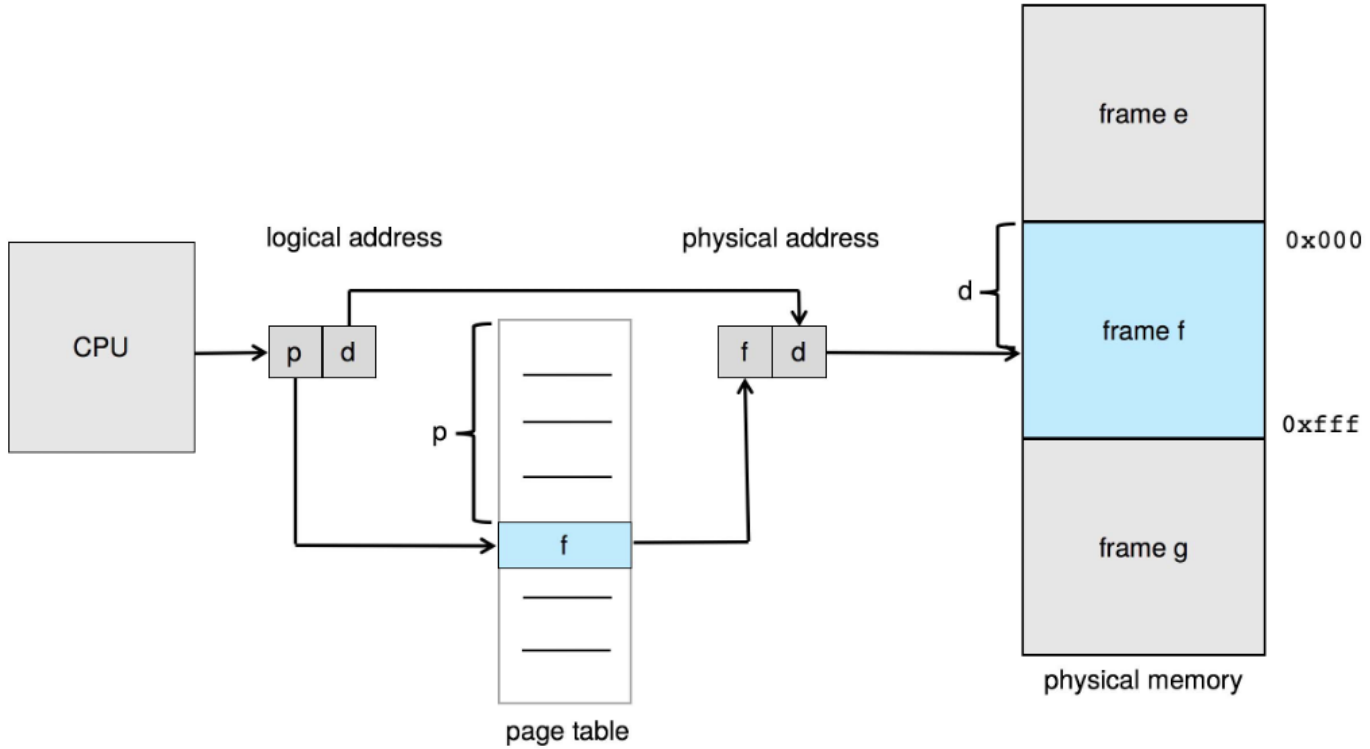
L'espace d'adressage virtuel est divisé en blocs de longueur fixe appelés **pages**.

- Un tableau fait correspondre la position des pages de l'espace virtuel à l'espace physique

Approche de la pagination

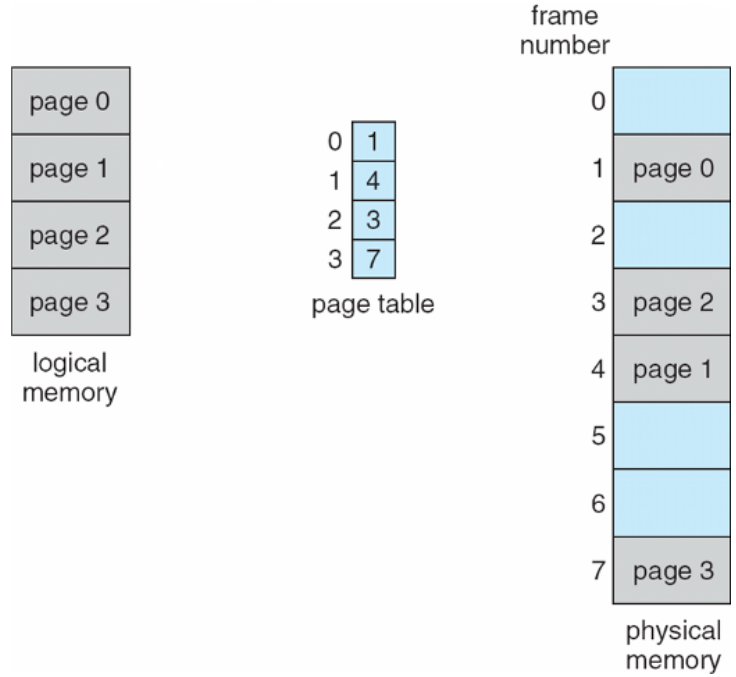


Approche de la pagination



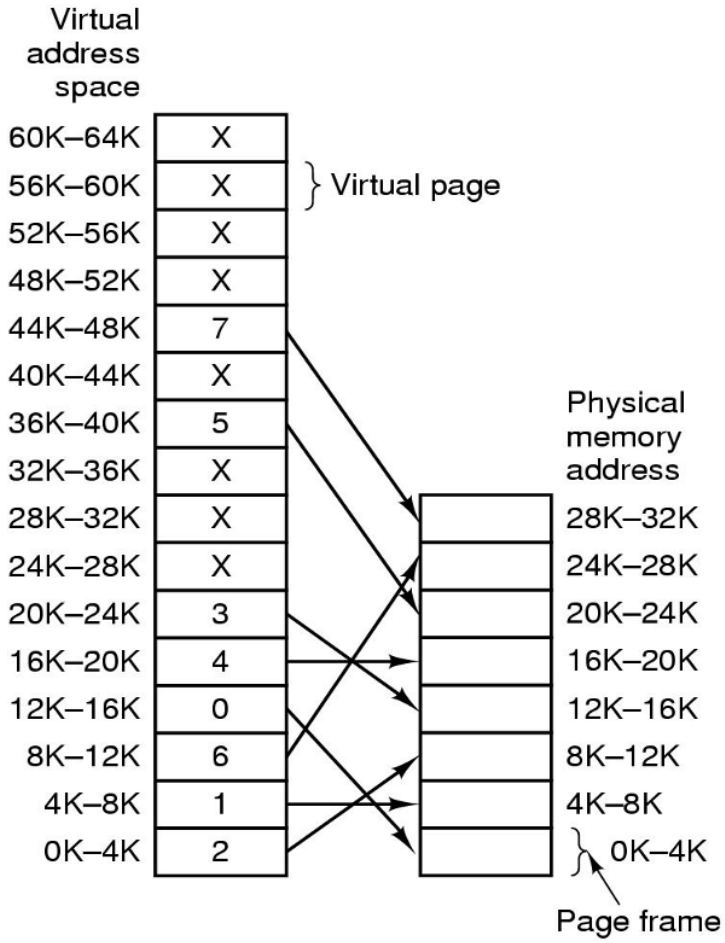
Approche de la pagination

- Le processus voit un espace virtuel qui est divisé en pages de taille fixe.
- La table des pages montre comment elles ont été allouées dans la mémoire physique.
- La MMU effectue la traduction
- Le système d'exploitation configure/programme la MMU.



Approche de la pagination

- En général, l'espace d'adressage virtuel est plus grand que l'espace d'adressage physique.
- **Exemple:** Le programme peut émettre des adresses sur 1024 pages, mais il n'y en a que 80 en mémoire.
- Le programmeur n'a pas besoin de connaître la quantité de mémoire dont dispose le système.



Approche par pagination

Avantages :

- Pas de fragmentation
- Flexible

Inconvénients :

- Nécessite un matériel rapide pour supporter l'ensemble du processus.

C'est le **standard *de-facto***.

- Utilisé dans tous les processeurs et systèmes d'exploitation modernes

Mémoire virtuelle

La mémoire virtuelle

C'est l'effet naturel de la mémoire paginée

- Le processus voit un espace d'adressage virtuel, mappé sur la mémoire physique.

Architecture X86-64 :

- Adresses virtuelles sur 64bit, mais seulement 48 utilisées
  - Mémoire virtuelle de ??? B

La mémoire virtuelle

C'est l'effet naturel de la mémoire paginée

- Le processus voit un espace d'adressage virtuel, mappé sur la mémoire physique.

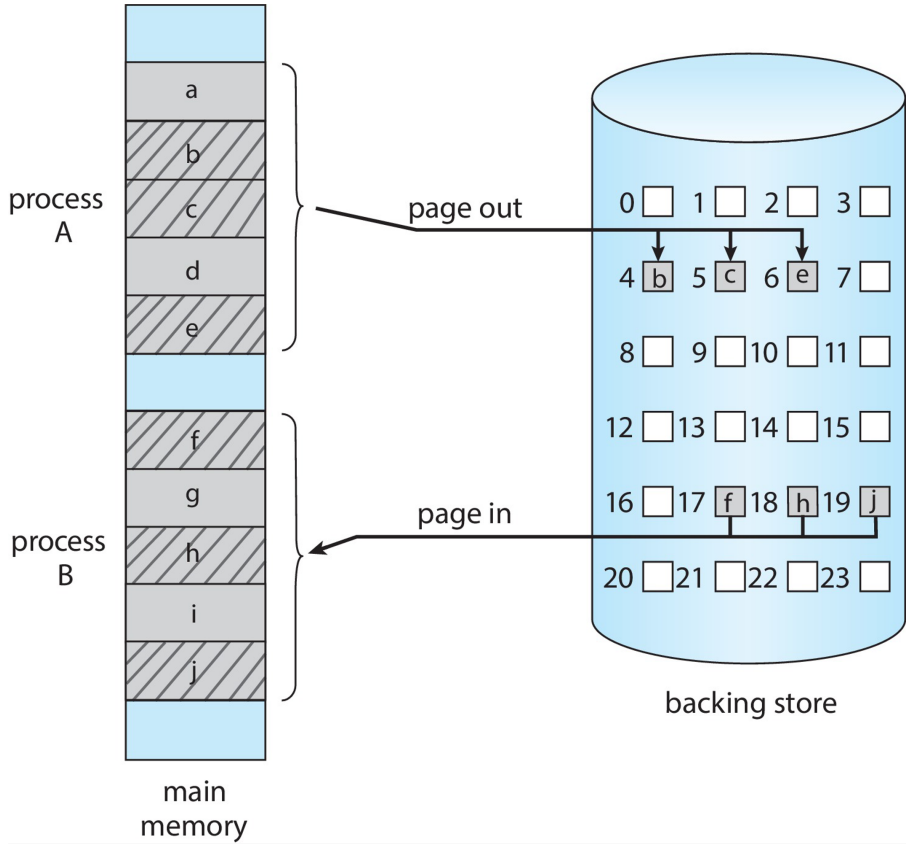
Architecture X86-64 :

- Adresses virtuelles sur 64bit, mais seulement 48 utilisées
  - Mémoire virtuelle de 256TB
- Pages de 4KB
  - Décalage de 12 bits
- MMU traduit en adresses de 48 bits
  - Mais la mémoire physique est toujours **beaucoup** plus petite

Mémoire virtuelle

Ce qui n'est pas en mémoire est mis sur le disque.

L'espace disque qui contient les pages qui ne sont pas en mémoire est appelé **Swap**.

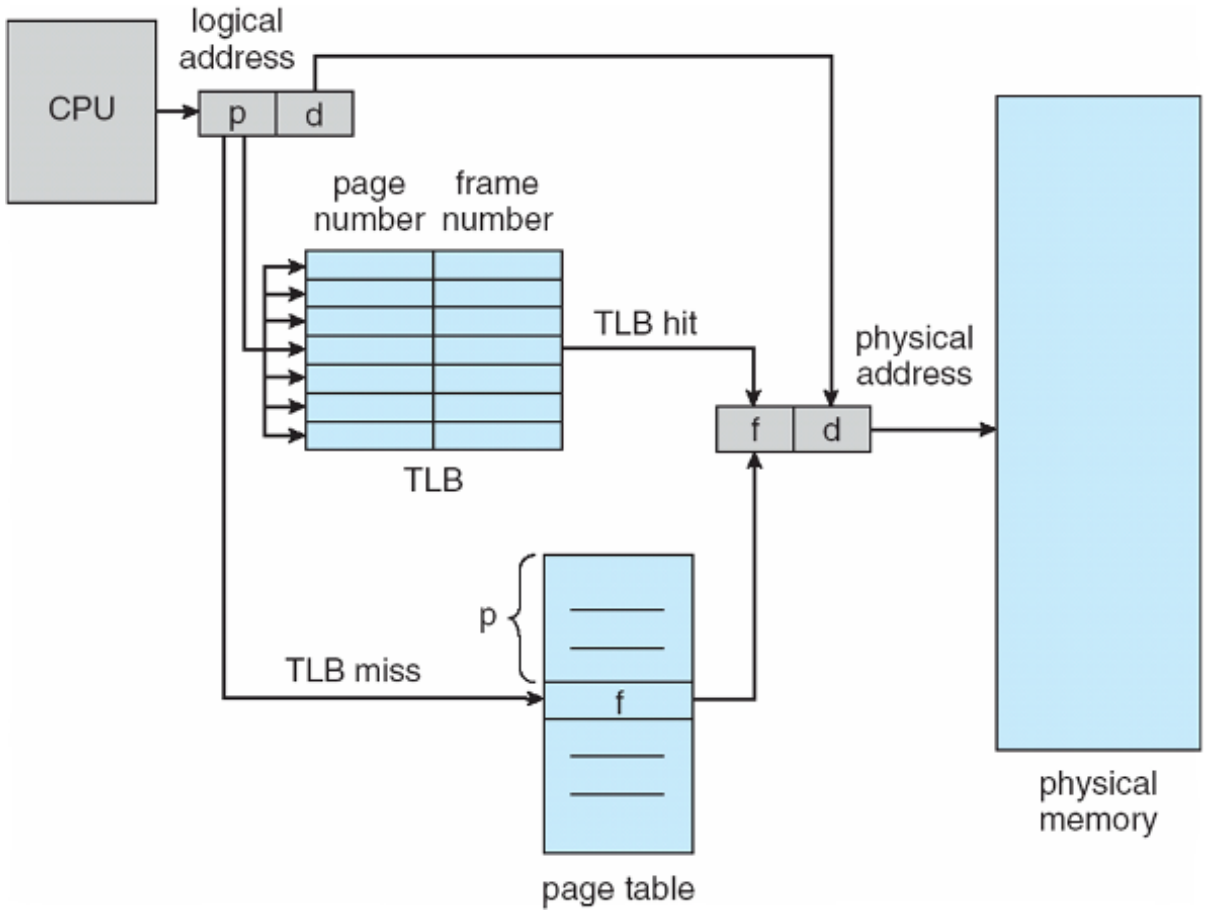


Mémoire virtuelle

Si le processus sort une adresse de page qui n'est pas en mémoire, un **Page Fault** se produit :

- La MMU avertit le SE
- L'OS abandonne le processus
- L'OS charge la page (ou la crée) à partir du disque
- Le système d'exploitation réinitialise la MMU
- Le processus reprend

La mémoire virtuelle



Remplacement des pages

Remplacement des pages

La mémoire physique est toujours plus petite que la mémoire virtuelle

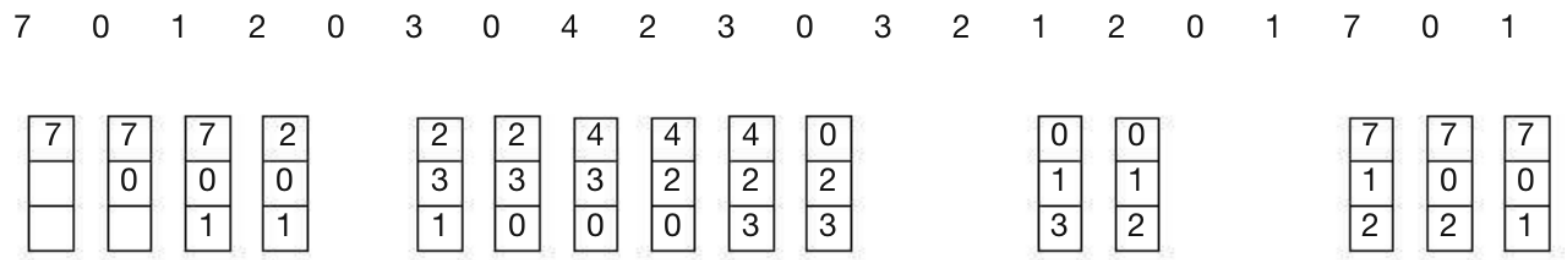
Si elle est pleine de pages utilisées par des processus actifs, le système d'exploitation doit choisir les pages à supprimer et à enregistrer sur le disque.

Il existe plusieurs algorithmes de remplacement pour faire cela de manière intelligente

Des idées ?

Algorithme FIFO

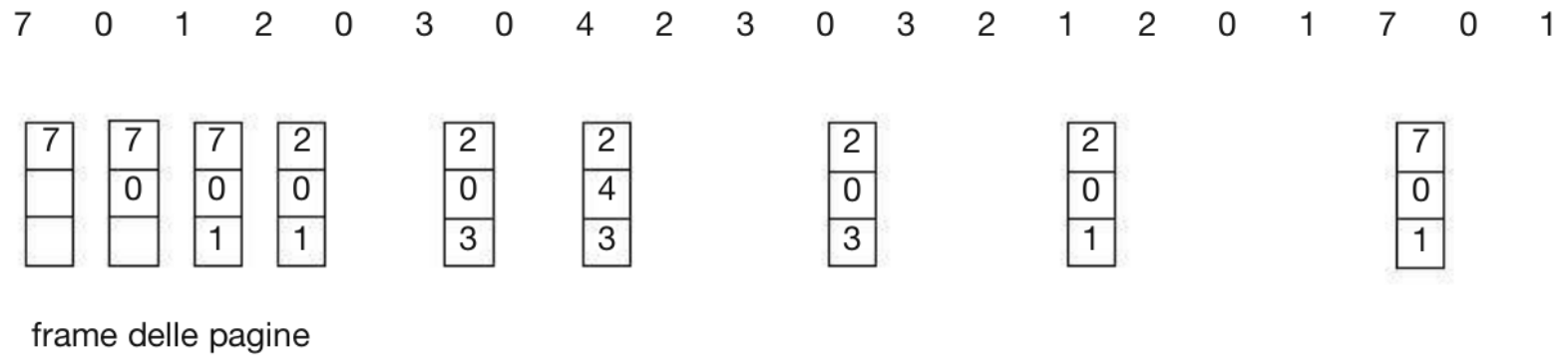
Suppression de la page la plus longtemps chargée de la mémoire



Simple, mais inefficace

Algorithme optimale

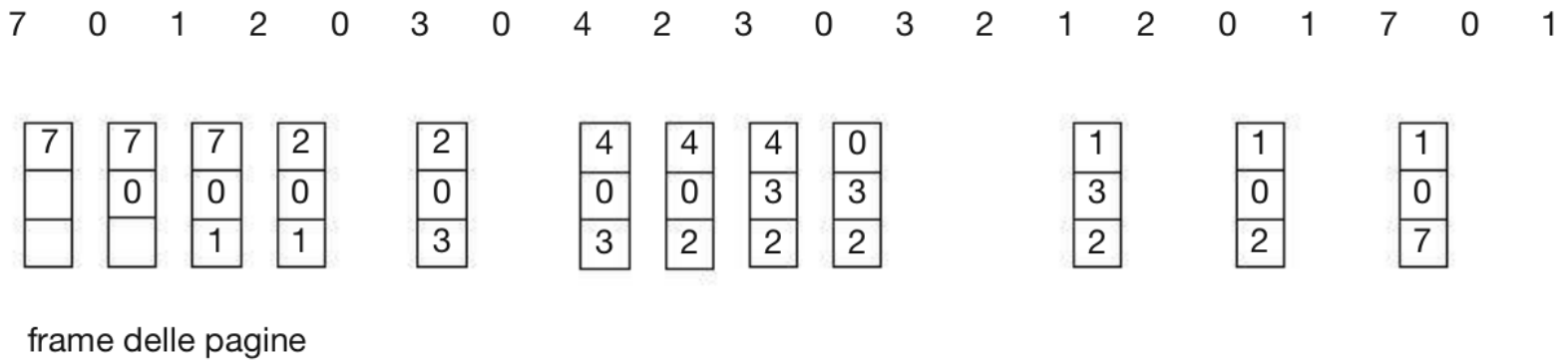
Je supprime la page dont je n'aurai plus besoin **dans le futur**.



Excellent, mais impossible de prédire l'avenir

Algorithme Least Recently Used

Je supprime la page qui n'a pas été utilisée depuis le plus longtemps



Simple, efficace.

Nécessite la coopération de la MMU pour garder la trace des accès.

Utilisé presque toujours (avec des variantes)

Disposition de la mémoire

Disposition de la mémoire

Un processus peut accéder à n'importe quel emplacement mémoire dans l'espace d'adressage virtuel.

- L'espace d'adressage virtuel est divisé en pages
- Si la page est en mémoire, le MMU la traduit en une adresse physique.
- Si la page n'est pas en mémoire, le système d'exploitation la crée/libère à partir du disque.

Disposition de la mémoire

Un programme qui accède à des adresses "aléatoires" n'est pas efficace.

- L'utilisation de la page et de la mémoire serait fortement pénalisée

Historiquement, les gens ont commencé à utiliser les adresses à partir de l'extrémité "basse" :

- Vous commencez à utiliser l'adresse `00 00 00 00`, puis `00 00 00 01`.
- Ainsi, vous remplissez complètement une page, puis vous commencez à en utiliser une autre.

Il existe différentes conventions, selon l'architecture de l'ordinateur et le système d'exploitation. On voit **Linux**

Disposition de la mémoire

Actuellement, les adresses hautes et basses sont utilisées.

- La mémoire peut croître dans deux directions
- Je peux avoir deux zones de mémoire qui croissent selon les besoins du programmeur

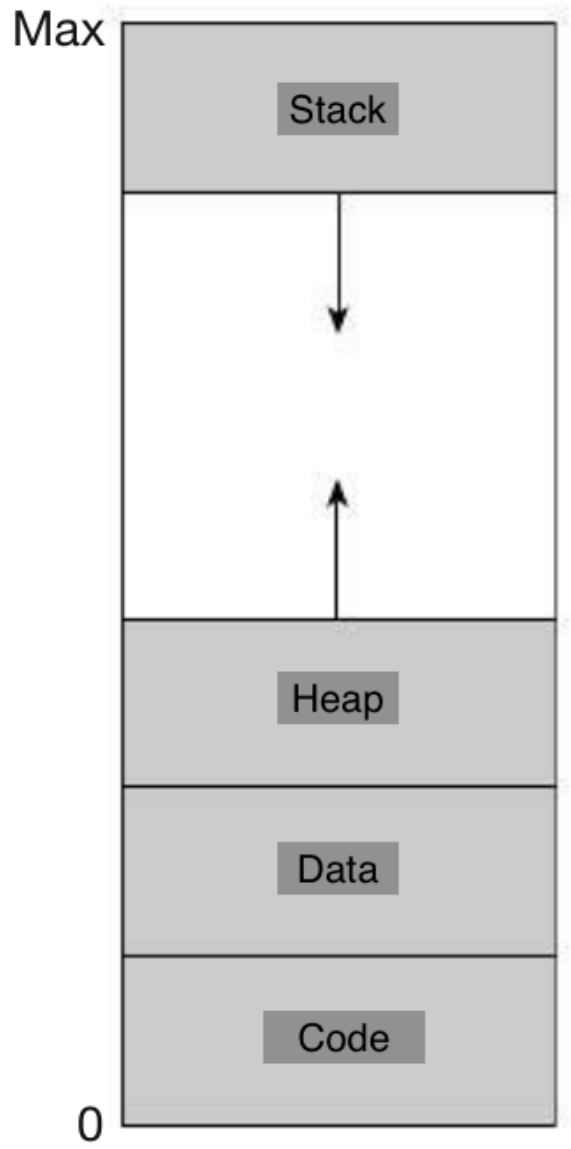
1. **Heap:** croît de bas en haut. Utilisée par le programmeur pour allouer de la mémoire quand il en a besoin.
2. **Stack:** grandit de haut en bas. Utilisée par le compilateur pour placer les variables des fonctions

Disposition de la mémoire

Un processus peut accéder à n'importe quel emplacement mémoire.
Par convention et pour des raisons de performance, il est préférable de commencer par les extrêmes.

Il existe 4 emplacements mémoire :

- Code
- Données
- Stack
- Heap

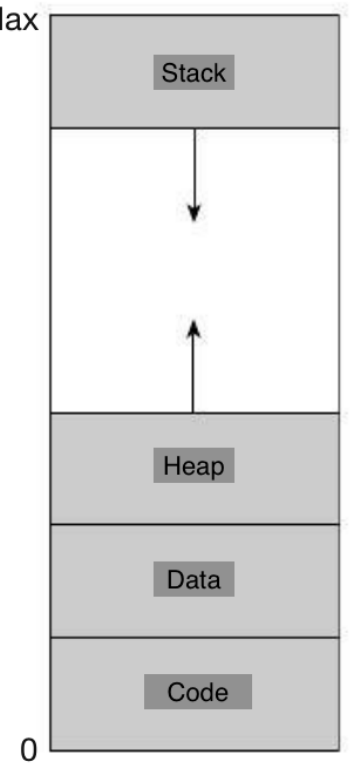


Code

Le système d'exploitation copie le code du programme depuis le disque vers les adresses les plus basses.

- Le code doit être en mémoire
- Le registre du *compteur de programme* du CPU pointe vers une adresse dans cette plage.

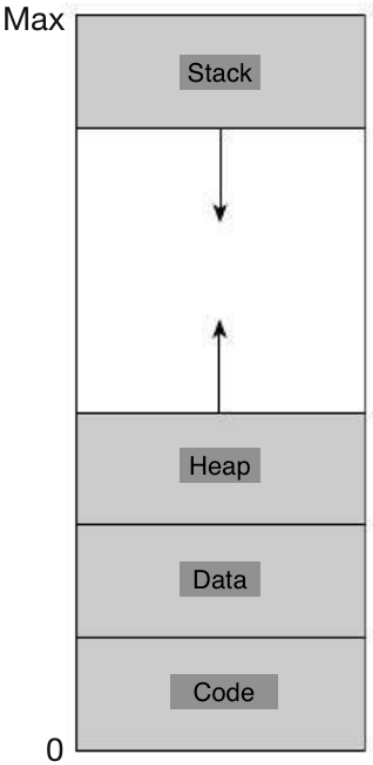
Cette partie de la mémoire est *Lecture seule* : un programme ne peut pas se modifier lui-même.



Données

Les adresses suivants sont utilisées pour les variables globales.

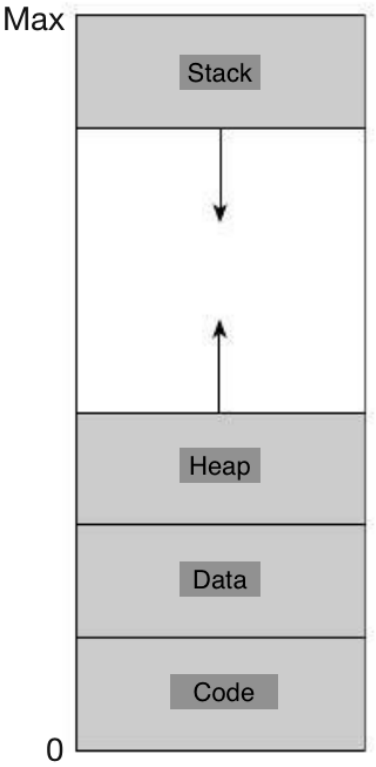
- Le compilateur utilise ces adresses pour les variables globales
- Les variables globales *initialisées* sont remplies directement par le système d'exploitation lorsque le processus est lancé.
- Les autres contiennent toutes `'\0'`.



Heap

Utilisé pour la **mémoire dynamique**.

- Le programmeur peut avoir besoin d'une mémoire dont la taille n'est pas prévue au stade de la programmation
- Gérée via les fonctions de la bibliothèque *malloc* et *free*.



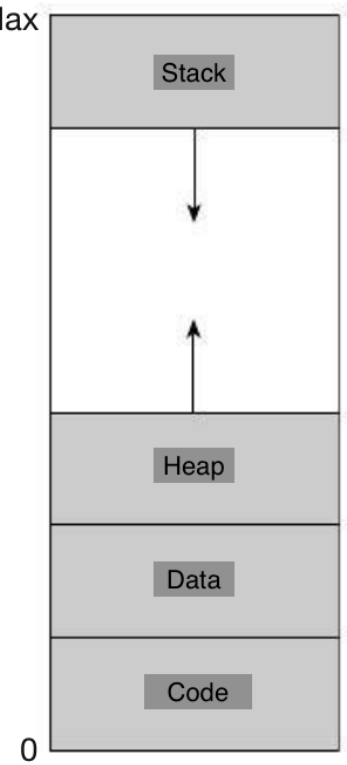
Stack

Utilisé pour les variables liées aux fonctions : arguments et variables internes.

- Comme il est dit, cette zone est traitée comme s'il s'agissait d'une **Stack**.

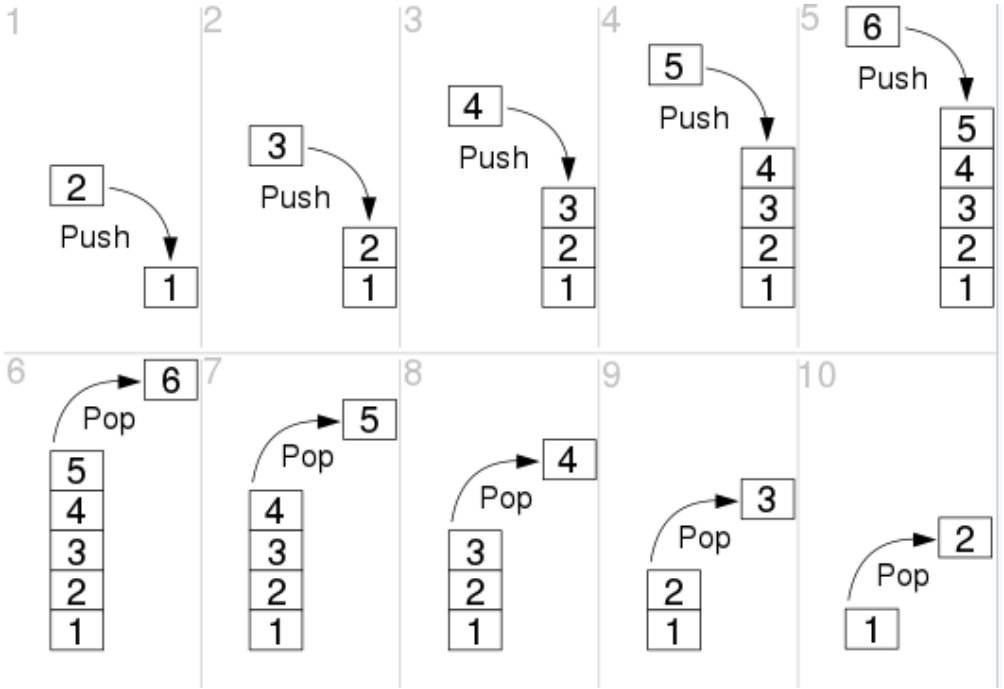
Les données le sont :

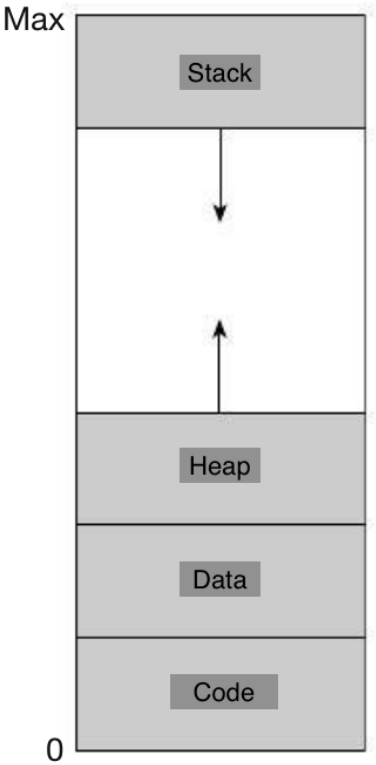
- Empilées pour être ajoutées : **Push.**
- Retirées de la pile lorsqu'elles doivent être utilisées : **Pop.**



Stack

Les données sont ajoutées et retirées du sommet de la pile.



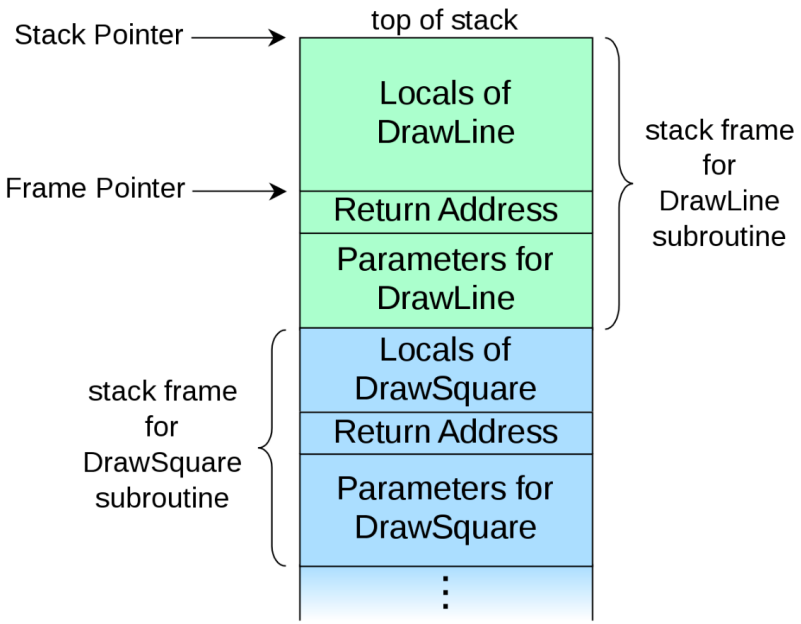


Stack

Concept pratique pour la gestion des fonctions
Lorsqu'une fonction est appelée, un bloc est ajouté à la pile qui la contient (**Push**) :

- Adresse de retour
- Paramètres
- Variables globales

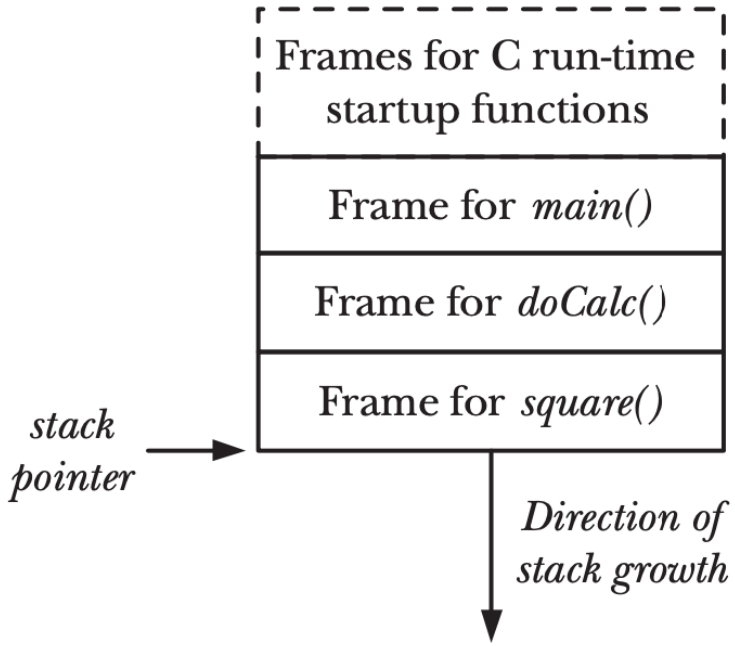
Lorsque la fonction revient, le bloc est supprimé (**Pop**).



Stack

Chaque bloc est appelé un **Stack Frame**.

- Créé lorsque la fonction est invoquée
- Supprimé lorsque la fonction revient



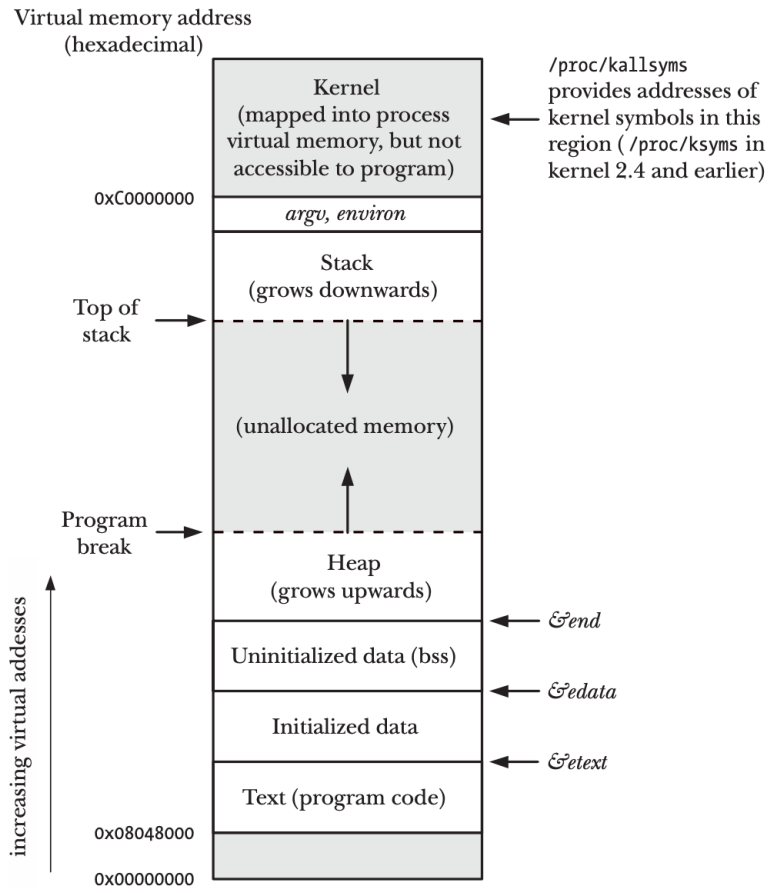
Sous Linux

La disposition vue précédemment est générique.

Sous Linux, on spécifie que :

- Un segment pour les variables globales initialisées et un autre pour les non initialisées.
- `argc` et `argv` dans les adresses hautes
- **Program Break** : adresse maximale pour la partie basse.
- **Top of Stack** : adresse minimale pour la partie haute

.



Chargeurs, bibliothèques et pages partagées

Chargeurs, bibliothèques et pages partagées

Le **Loader** est le composant du système d'exploitation qui démarre les processus. Ses tâches sont :

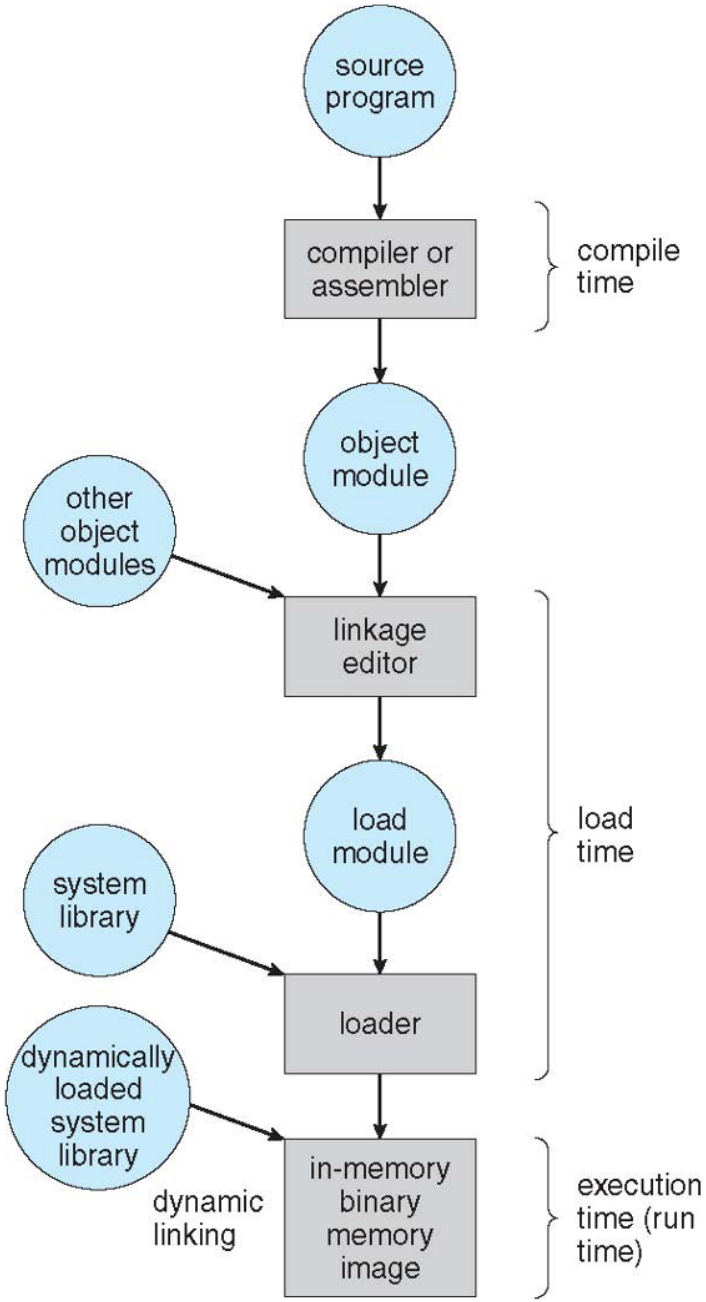
- Vérifier que l'utilisateur a les permissions.
- Copier le code du programme et les variables globales sont initialisées en mémoire.
- Charger les bibliothèques partagées
- Valider `argc` et `argv`.
- Lancez le processus depuis `main` en paramétrant le planificateur du système d'exploitation

Chargeurs, bibliothèques et pages partagées

Le compilateur crée le code machine

Le chargeur le charge et commence l'exécution

- Il doit également charger les bibliothèques du système, si elles sont utilisées.



Chargeurs, bibliothèques et pages partagées

Les programmes peuvent utiliser des **bibliothèques partagées** :

- Proposées par le système d'exploitation pour faciliter l'appel système.
- Installées par l'utilisateur à des fins particulières (par exemple, la trigonométrie).

Les bibliothèques sont du code qui s'exécute en *mode utilisateur*.

- Elles n'ont aucun privilège sur le code utilisateur.
- Elles **ne font pas** partie du noyau.

Chargeurs, bibliothèques et pages partagées

Les **librairies** partagées sont du code exécutable dans les dossiers système par défaut.

Dans Linux :

- `/lib`
- `/usr/lib`
- Répertoires listés dans le fichier `/etc/ld.so.conf`.

Sous Windows :

- `C:\Windows\SYSTEM32`
- Répertoire courant

Chargeurs, bibliothèques et pages partagées

Les exécutables sous Linux sont au format ELF.

- En plus du code, ils contiennent la liste des bibliothèques système qu'ils vont utiliser
- Contenus dans un emplacement prédéfini

Les bibliothèques et les pages partagées sont identifiées par leur nom et leur version.

Loader, bibliothèques et pages partagées

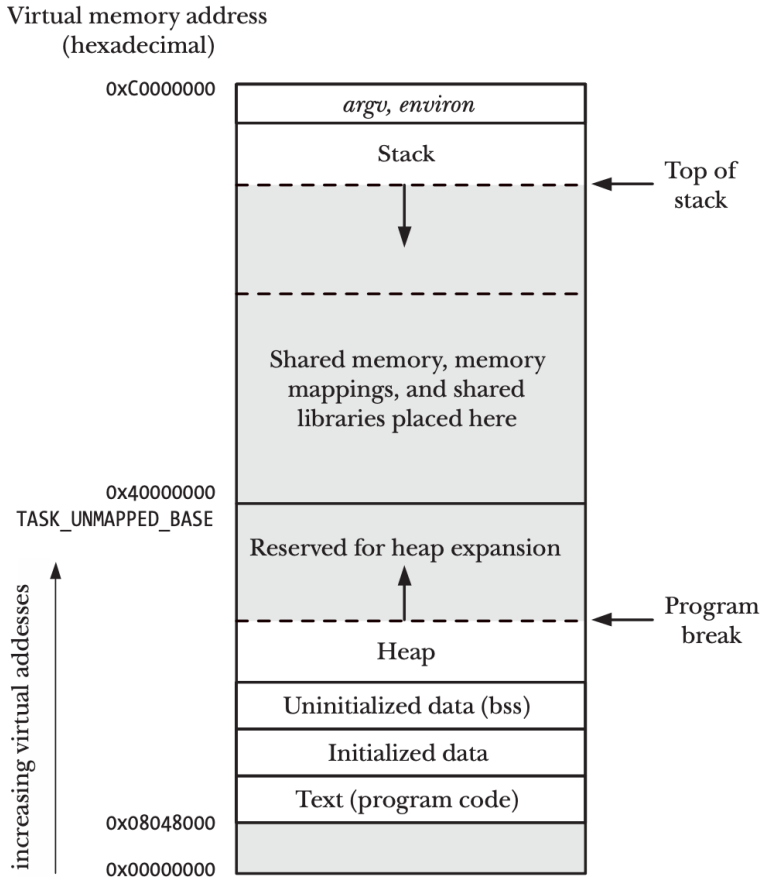
Les bibliothèques sont chargées depuis le Loader dans des adresses intermédiaires

Si plusieurs processus utilisent la même bibliothèque, la page est partagée

Loader, bibliothèques et pages partagées

La mémoire partagée entre processus fonctionne de la même manière

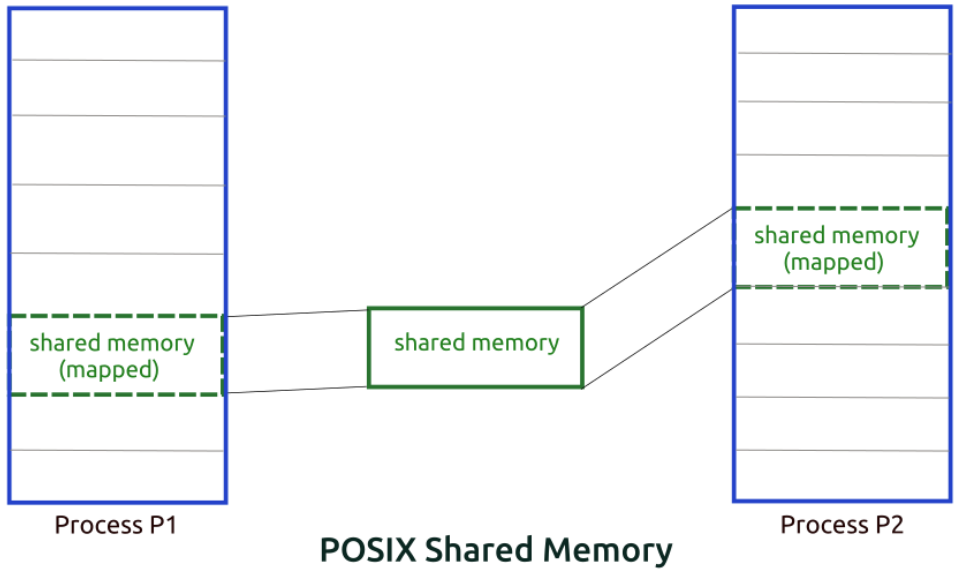
- Les adresses entre le tas et la pile sont utilisées pour tout ce qui est partagé



Chargeurs, bibliothèques et pages partagés

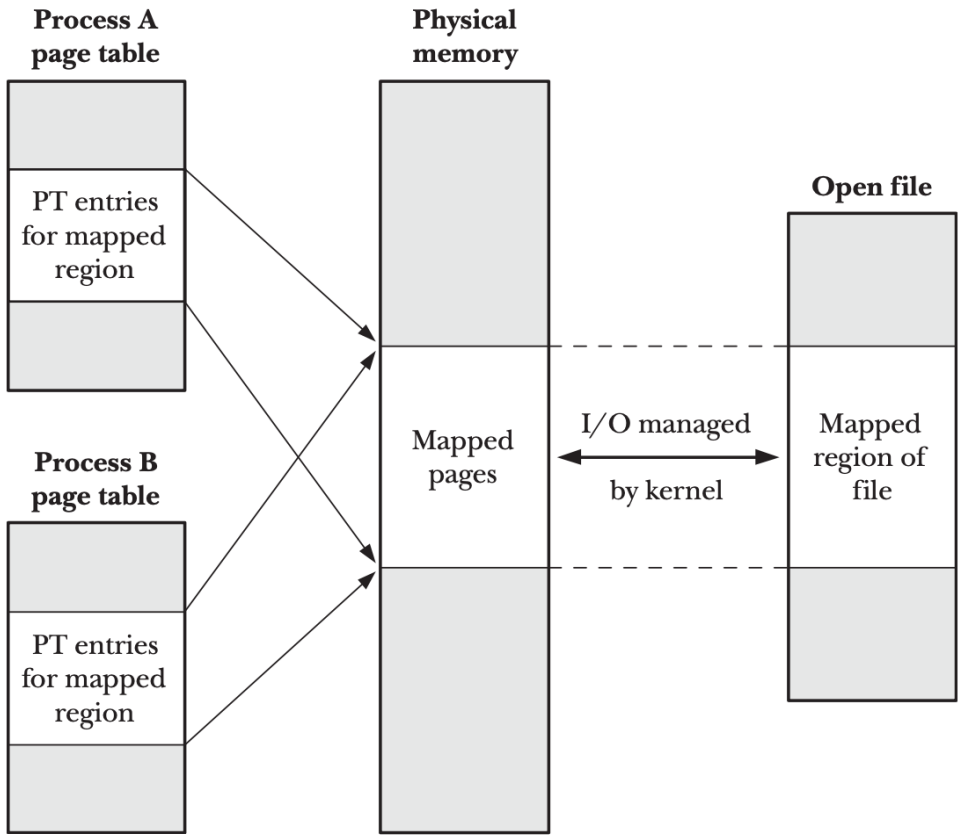
Tout ceci est possible grâce à la MMU

- Le système d'exploitation configure la MMU pour mettre en œuvre ce schéma



Chargeurs, bibliothèques et pages partagées

Dans le cas des `mmaps` avec persistance des fichiers, le noyau se charge d'aligner la zone de mémoire partagée sur le disque



La gestion de la mémoire dans Bash

Gestion de la mémoire en Bash

- `free` : montre combien de mémoire est disponible/utilisée/libre sur la machine
- `top` : montre diverses informations sur les processus
  - Colonne `RES` : *Resident Set Size* : combien de pages de l'espace virtuel d'un processus sont chargées dans la mémoire physique
  - Colonne `VIRT` : *Virtual Set Size* : combien de pages ont été utilisées par le processus dans son historique.
  - Colonne `%MEM` : `RES/total` : combien de la mémoire physique de la machine contient de pages du processus.

Gestion de la mémoire en Bash

- `cat /proc/meminfo` : affiche des informations détaillées sur la mémoire de la machine
- `ldd [executable]` : montre quelles bibliothèques partagées il requiert

Limites de la mémoire statique

Les variables globales sont allouées dans le segment **data**.

Le loader initialise la valeur

```
int a = 40 ; /* Initialisé par le loader*/
int main(){...}
```

Si elle n'est pas spécifiée, la variable est initialisée à 0.

```
int a ; /* Initialisé à 0 */
int main(){...}
```

Limites de la mémoire statique

Les variables de fonction sont allouées dans le **stack**.

Aucune valeur n'est initialisée ! Elles peuvent contenir des données arbitraires

```
int  f(int a, int b){
     int  s = a + b ;
     return  s ;
}
```

Les arguments `a` et `b`, la variable `s`, et la valeur de retour sont sur le stack

Limites de mémoire statique

Variables statiques :

Les variables dans une fonction avec le mot-clé `static` sont allouées dans le segment de données et non dans le stack.

Elles sont initialisées par le loader.

Elles conservent leur valeur après la fin de la fonction.

```c
#include<stdio.h>
int fun(){
   static int count = 0 ; /* Initialisé UNE FOIS par le loader au démarrage du processus */
   count++ ;
   return count ;
}

int main(){
   printf("%d ", fun()) ;
   printf("%d ", fun()) ;
   return 0 ;
}
```

Imprime `1 2`

Limites de la mémoire statique

Dans certains cas, le programmeur ne sait pas combien de données il doit charger en mémoire

- Lecture d'une structure de données à partir d'un fichier
- Entrée utilisateur de longueur variable

Avec ce que nous avons vu, en C les tableaux ont des longueurs fixes, connues au moment de la compilation

```
#define N 50
int v [N] ;
```

Limites de la mémoire statique

En C, vous ne pouvez pas créer des vecteurs dont la longueur n'est pas connue du compilateur.
Le code suivant est **incorrect**

```
scanf("%d", &n) ;
int v[n] ;
```

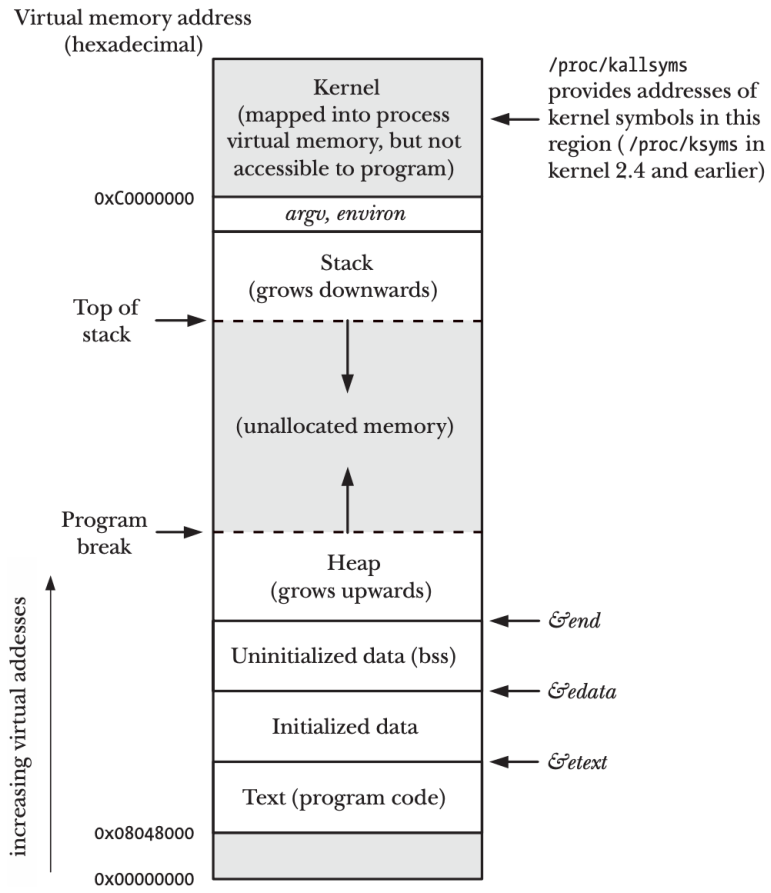
Cela conduirait les programmeurs à **surdimensionner** les vecteurs (s'il n'y avait pas de mémoire dynamique)

Mémoire dynamique

En C, il est possible d'utiliser la **mémoire dynamique** pour créer des structures de données dont la taille n'est pas connue à la compilation.

Utilisation correcte : création de vecteurs de longueur arbitraire et décidée au *moment de l'exécution*.

Fonctionnalité : les adresses virtuelles sont utilisées dans le segment **Heap**. Il peut croître au cours de l'exécution du programme



Mémoire dynamique

Dans Linux

Pour utiliser la mémoire dynamique, les **fonctions de bibliothèque** sont utilisées pour allouer ou libérer des blocs de mémoire.

Les fonctions de bibliothèque utilisent l'appel système `sbrk` qui informe le système d'exploitation que le processus émettra des adresses virtuelles dans des zones précédemment inutilisées.

- En pratique, il informe le système d'exploitation que l'heap s'agrandit et que le processus accèdera à des pages supplémentaires de mémoire virtuelle.

Mémoire dynamique

Toutes les fonctions de la bibliothèque pour la mémoire dynamique sont *Thread Safe*.

- Elles peuvent être invoquées en parallèle par plusieurs *threads*.
- En interne, elles maintiennent et utilisent des *mutex* pour réguler l'accès aux structures de données.
- Gardez ces phrases en mémoire 😃

La fonction `malloc`

```
#include <stdlib.h>
void  *malloc(size_t size) ;
```

Alloue `size` octets de mémoire et renvoie le pointeur vers la mémoire allouée.
La mémoire n'est pas initialisée, elle peut contenir n'importe quelle valeur.
Si l'allocation échoue (par exemple, s'il n'y a pas de mémoire), elle renvoie `NULL`.

La fonction `malloc`

Utilisation

La fonction `malloc` requiert `size` en octets. Vous devez utiliser l'opérateur `sizeof` pour connaître la taille du type de variable à allouer.

La valeur de retour est `void *`, qui est un pointeur sans type.

Pour utiliser la mémoire allouée, vous l'affectez à un pointeur du type désiré

```
/* Nous voulons allouer un vecteur de flottants*/
float * v ;
/* La longueur est déterminée au moment de l'exécution */
scanf("%d", &n) ;
/* Les octets à allouer sont n blocs, chacun étant aussi long qu'un flottant */
v = malloc(n * sizeof(float)) ;  /* Un void* est alloué à un float */
v[0] = 12.2 ;  /* Arithmétique des pointeurs */
```

La funzione `calloc`

```
#include <stdlib.h>
void  *calloc(size_t nmemb, size_t size);
```

Similaire à `malloc` mais elloue de la mémoire pour un vecteur de `nmemb` éléments chacun de `size` octets et renvoie le pointeur.
La mémoire **est initialisée** à 0$.

Observation: Contrairement à `malloc`, `calloc` reçoit `size` et `nmemb` et fait la multiplication en interne.

La funzione `realloc`

```
#include <stdlib.h>
void  *realloc(void *ptr, size_t size);
```

Modifie la taille de la zone mémoire pointée de `ptr` à `size` octets.
La valeur de retour est le pointeur sur la zone étendue

- S'il en résulte un rétrécissement de la zone mémoire, les données excédentaires sont perdues.
- S'il en résulte une augmentation, la zone supplémentaire n'est pas initialisée.

Note: `ptr` doit avoir été obtenu avec `malloc` ou `realloc`.

Note: si possible, `realloc` étend la zone de mémoire actuelle, et la valeur de retour est égale à `ptr`.
Si ce n'est pas possible, les données sont copiées dans une nouvelle région, dont l'adresse est renvoyée

La funzione `free`

```c
#include <stdlib.h>
void  free(void *ptr);
```

Désalloue (ou libère) la zone de mémoire indiquée par `ptr`.
`ptr` doit avoir été obtenu avec `malloc` ou `realloc`.

Si vous essayez de libérer une zone de mémoire plus d'une fois, le comportement est indéfini.

La fonction `free`

Toutes les zones de mémoire doivent être désallouées via la fonction `free`.
Si cela n'est pas fait, la mémoire est libérée à la fin du processus

Important :.
Ne pas désallouer la mémoire est toujours une erreur !

Pourquoi ?

La fonction `free`

Toutes les zones de mémoire doivent être désallouées via la fonction `free`.
Si cela n'est pas fait, la mémoire est libérée à la fin du processus

Important :.
Ne pas désallouer la mémoire est toujours une erreur !
Dans les programmes qui doivent fonctionner pendant longtemps, la mémoire qui n'est pas désallouée provoque un **Memory Leak**.
À un moment donné, toute la mémoire du système est allouée !

La fonction `free`

Common Errors:

La valeur de retour de `malloc` n'est pas assignée à un pointeur

```
// Incorrect
float v = malloc(5*sizeof(float))
float v [10] = malloc(5*sizeof(float)) ;
// Corrigé
float * v = malloc(5*sizeof(float)) ;
```

Créer un vecteur dont la taille n'est pas connue à la compilation

```
// Incorrect
float v [n] ;
// Corrigé
float * v = malloc(n*sizeof(float)) ;
```

Fonctionnement interne

Les fonctions `malloc` `calloc` `realloc` `free` sont des fonctions de bibliothèque qui utilisent l'appel système `sbrk`.

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

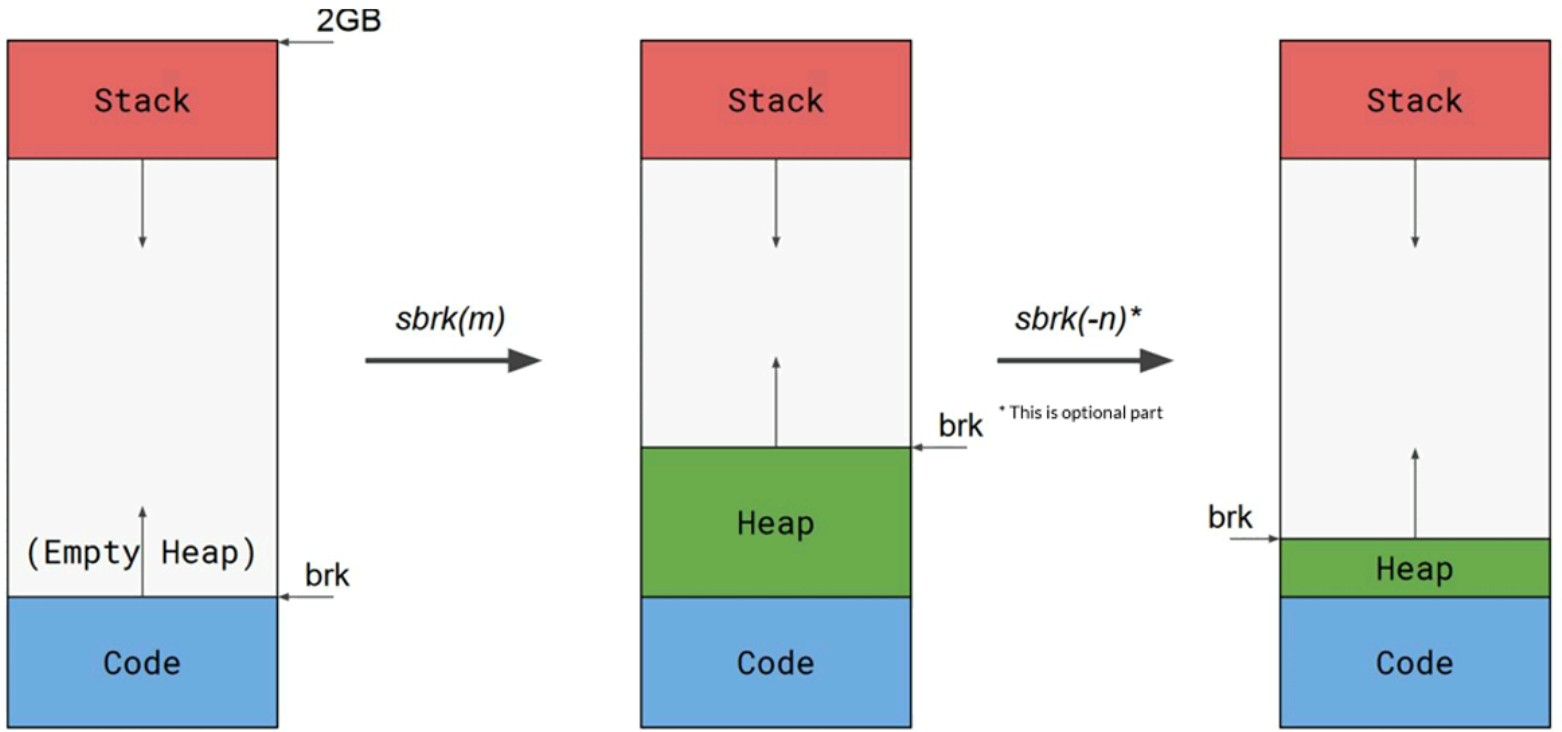
Il augmente par `incrément` le *data segment*, entendu comme l'union du segment de code, des données et de l'heap.

En pratique, cela informe le système d'exploitation que l'heap est en train de s'étendre.

- Si nécessaire, le système d'exploitation configurera la MMU pour qu'elle puisse accueillir des pages supplémentaires

Fonctionnement interne

L'appel à `sbrk` est en soi suffisant pour utiliser des adresses virtuelles plus élevées



Fonctionnement interne

Il serait difficile pour le programmeur de gérer la mémoire dynamique en utilisant uniquement `sbrk`.

- Il devrait garder une trace de chaque allocation et désallocation
- Il devrait disposer d'une technique pour réutiliser les *trous* laissés libres par une désallocation.
  - Lors d'une nouvelle allocation
- Invoquer `sbrk` à chaque allocation est inefficace
  - Un appel système est lent (implique un changement de contexte)

Les fonctions de la bibliothèque gèrent tout cela pour le programmeur.

- Utiliser des structures de données appropriées

Fonctionnement interne

La fonction moderne `malloc` provient d'une proposition de Doug Lea, professeur à l'Université de l'Etat de New York à Oswego.

En interne, elle utilise une **linked list** pour garder une trace des zones occupées.

Note : *heap* a deux significations !

1. Une structure de données qui met en œuvre une file d'attente prioritaire par le biais d'un arbre.
    - Permet de trouver facilement le maximum d'un ensemble de nombres.
    - Rapide à mettre à jour
2. La zone de mémoire virtuelle où la mémoire dynamique est allouée

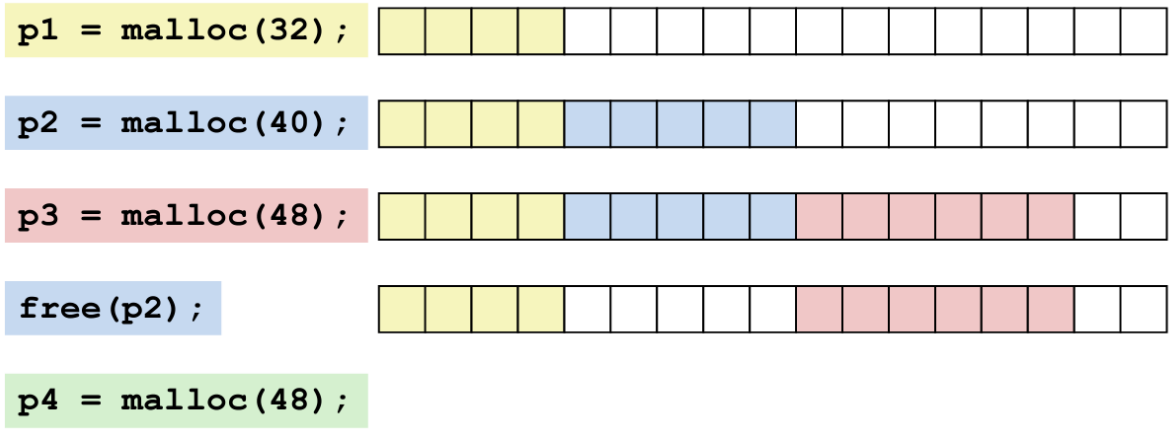
Fonctionnement interne

Le `malloc` gère des blocs de taille variable

- Il n'y a pas de discrétisation ou d'utilisation de blocs de taille fixe
- Conduit à la **fragmentation externe** : mémoire gaspillée parce qu'il s'agit d'une zone contiguë trop petite pour être allouée.

Fonctionnement interne

Il est possible d'arriver à des situations comme celle-ci :



`malloc(48)` pourrait être contourné si la mémoire libre était contiguë

Fonctionnement interne

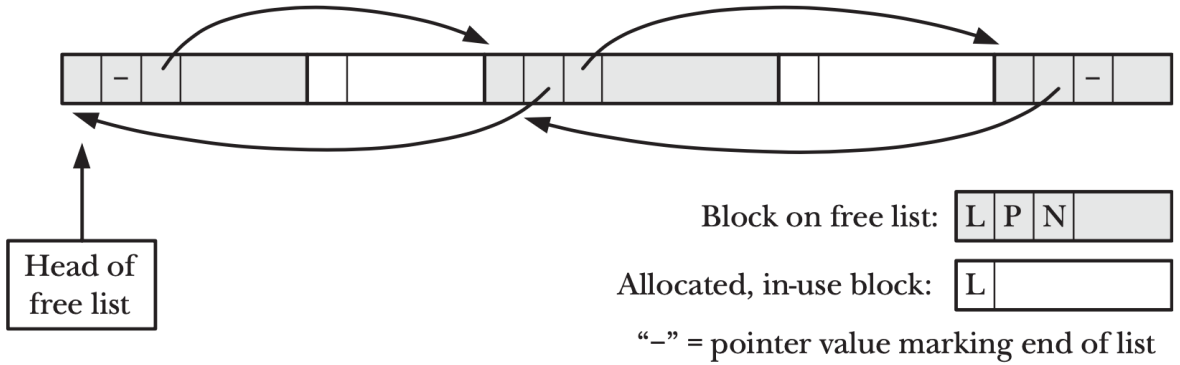
La fonction `malloc` gère indépendamment plusieurs zones de mémoire, appelées **Arenas**.

- Les structures de données sont répliquées
- Il est donc plus efficace de l'utiliser dans des contextes multithreads.
  - Les fonctions `malloc`, etc., sont sûres pour les threads.
- Évite à plusieurs threads d'être ralentis par l'attente du relâchement d'un verrou.
  - Les verrous sont nécessaires, mais l'utilisation de plusieurs structures réduit leur impact.

Fonctionnement interne

Une zone de mémoire gérée par `malloc` est administrée par une **liste liée**.

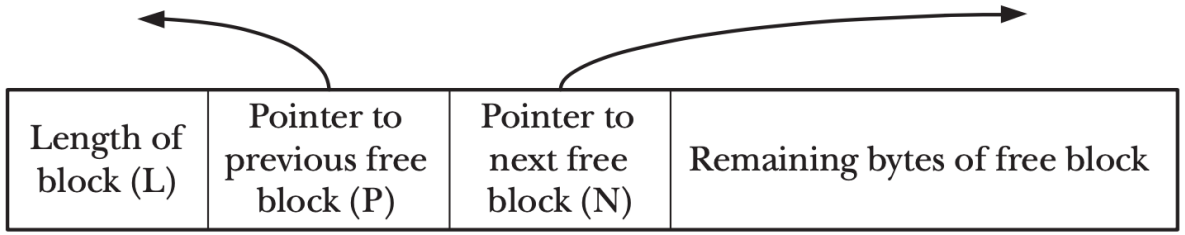
- Les segments encore libres sont une **liste doublement liée**.
- Les zones allouées sont momentanément retirées de la liste



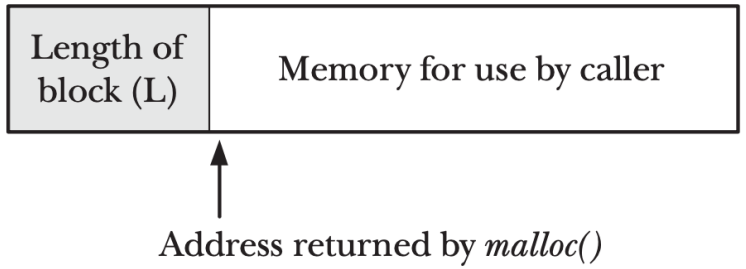
Fonctionnement interne

Chaque zone libre ou allouée possède une `struct` dans les premiers octets qui donne des informations sur elle et les blocs adjacents

- **Zone libre**



- **Zone allouée**



Indications sur le fonctionnement interne

Dans le cas où `malloc` doit allouer de grandes régions de mémoire (typiquement $> 128\ kB$), il utilise l'appel système `mmap` pour allouer une région de mémoire.

- `malloc` demande une région de type `MAP_ANONYMOUS`. Elle ne doit être partagée avec personne !
- Le système d'exploitation crée une ou plusieurs pages pour le processus
- Il les place à l'endroit de son choix dans l'espace d'adressage virtuel.