S54MA2M7 : Informatique 2
Archi des ordis, mémoire, débogage

Luigi Santocanale
LIS, Aix-Marseille Université

26 février 2019

Plan

Sources

Les cours de Jean-Michel Adam :

http://imss-www.upmf-grenoble.fr/~adamj/doclicence/A-StructureGeneraleOrdinateur.pdf
http://imss-www.upmf-grenoble.fr/~adamj/doclicence/0-FonctionnementUniteCentrale.ppt
http://imss-www.upmf-grenoble.fr/prevert/SpecialiteIHS/Documents/6-GestionMemoire-4dpp.pdf

Le cours de Jean-Marc Talbot :

http://pageperso.lis-lab.fr/~jean-marc.talbot/Teaching/Archi/

Le cours de Sylvain Montagny :

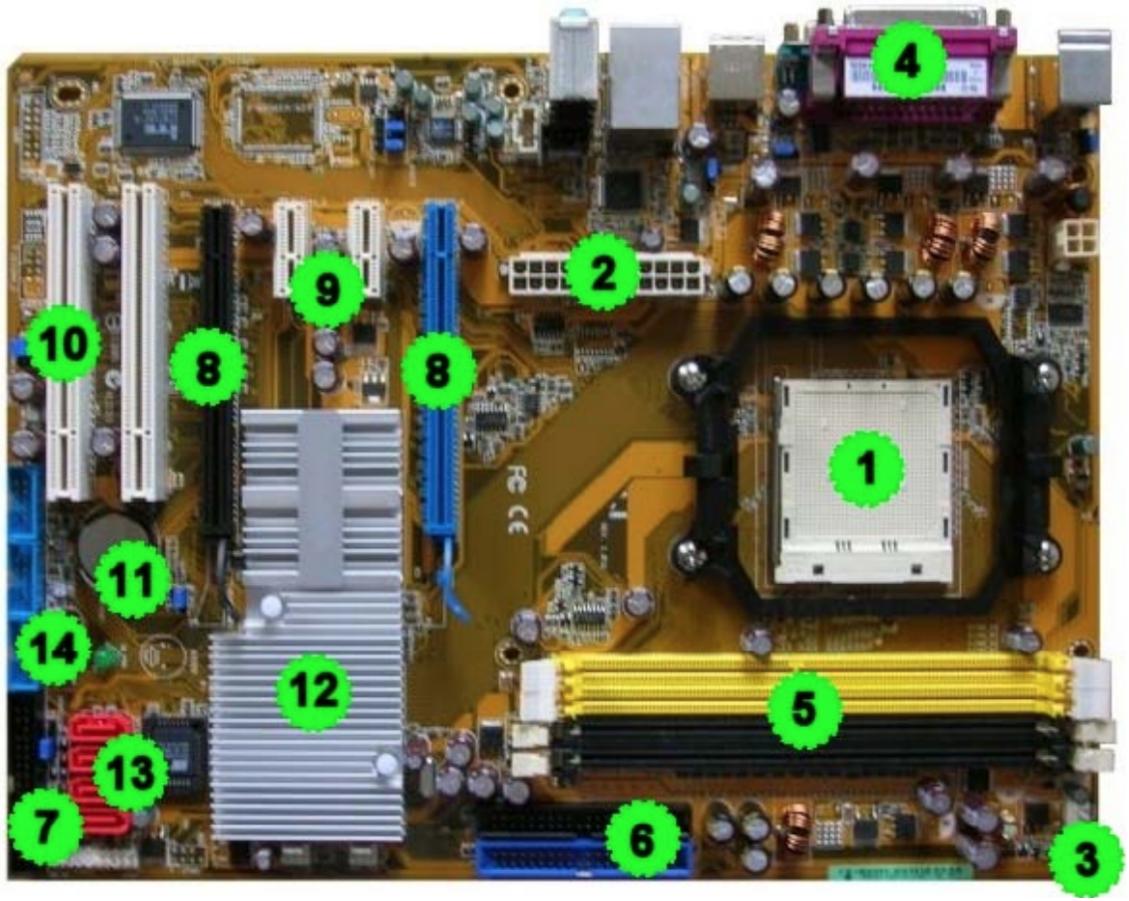
https://les-electroniciens.com/sites/default/files/cours/cours-architecture_des_ordinateurs.pdf

Le cours de Walker White :

https://www.cs.cornell.edu/courses/cs1110/2015fa/lectures/09-24-15/presentation-10.pdf

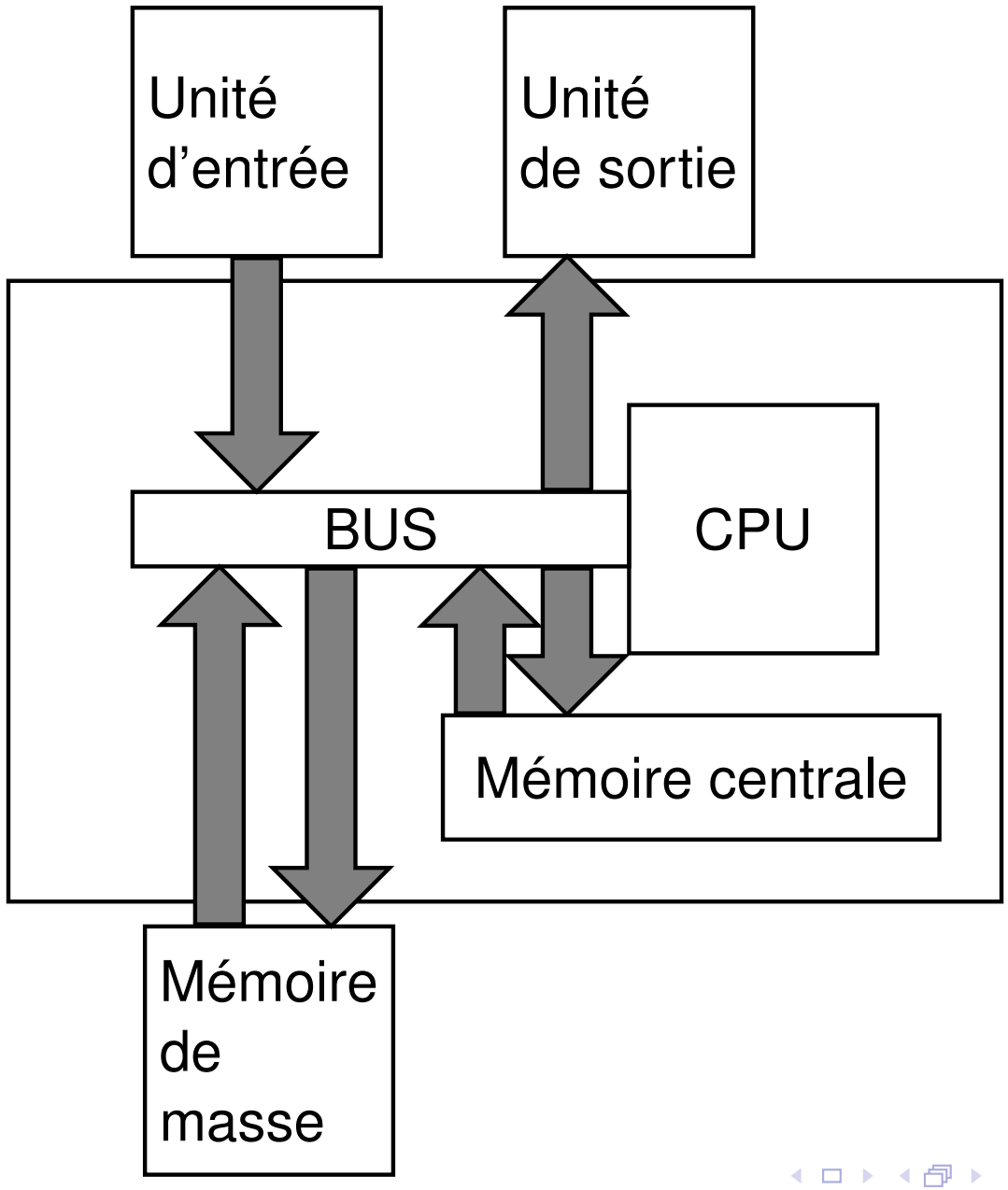
Plan

La carte mère



1. Socket ou slot du processeur
2. Prise de l'alimentation électrique de la carte mère
3. Prise de l'alimentation électrique du ventilateur du processeur
4. Ports situés à l'arrière
5. Slot(s) mémoire (ici pour DDR2-SDRAM sur 240 broches)
6. Port IDE
7. Port Floppy
8. Port PCI Express 16x
9. Port PCI Express 1x
10. Port PCI
11. Pile
12. Chipset
13. Ports Serial ATA
14. Ports USB interne

Schéma d'un ordinateur

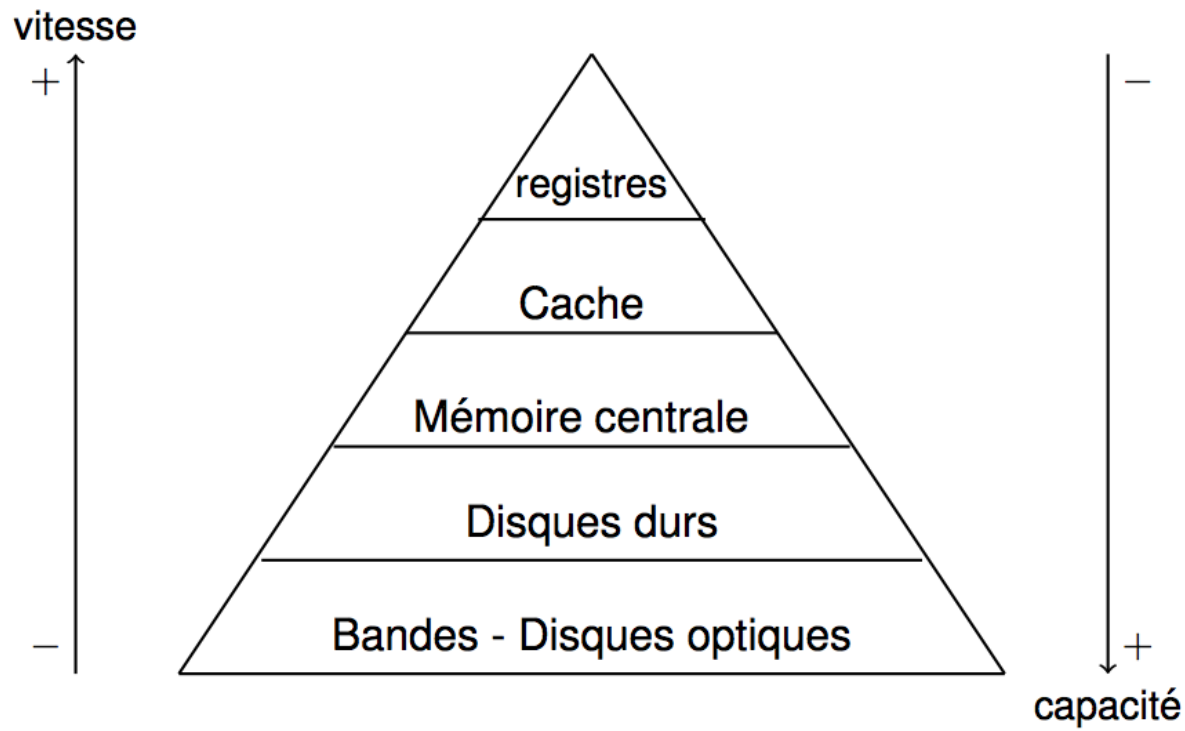


Les différentes mémoires

On distingue :

- dispositifs de stockage dans la CPU :
  - registres ;
  - mémoire cache ;
- mémoire centrale :
  - mémoire vive (RAM, Random Access Memory), volatile ;
  - mémoire morte (ROM, Read-Only Memory), non volatile ;
- mémoire de masse, non volatile, sur périphérique de stockage :
  - bande magnétique,
  - disque dur,
  - solid-state drive, disque optique (CD, DVD, Blu-ray),
  - disque magnéto-optique,
  - mémoire flash.

Types de mémoires (I)

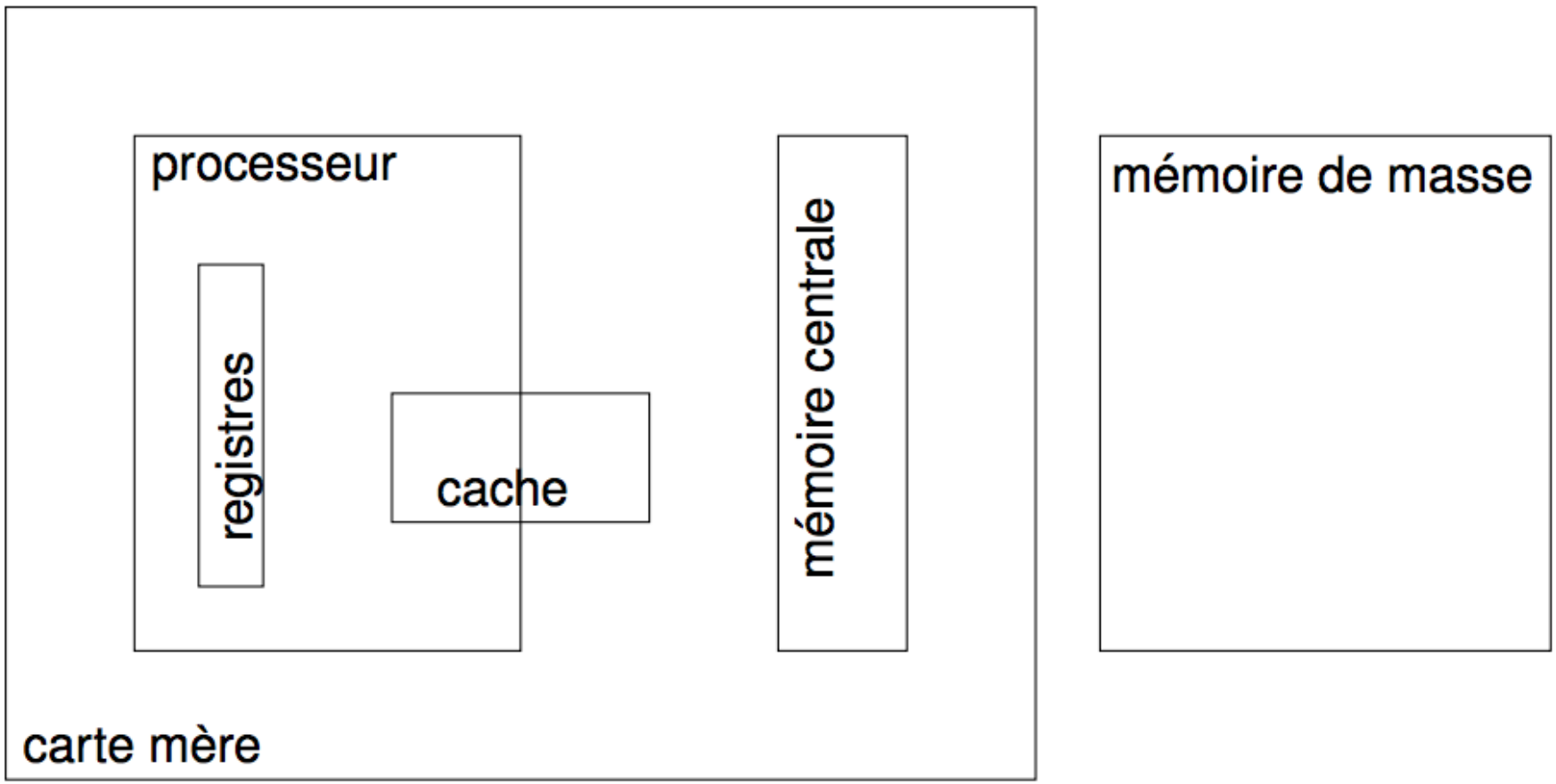


Performances des mémoires

	vitesse (temps d'accès)	vitesse (débit)	capacité
régistres	< 1 ns	> 50 Go/s	< 100 octets
cache	2 - 5 ns	5 - 20 Go/s	100 Ko - 1 Mo
mémoire centrale	20 ns	1 Go/s	256 Mo - 4 Go
disque dur	1-10 ms	300 Mo/s	50 Go - 500 Go

- temps d'accès :
  temps qui sépare une demande de lecture/écriture et sa réalisation.
- débit (ou bande passante) :
  nombre de bits maximum transmis par seconde.

Memoires : localisations



Le processeur (CPU)

- Unité Centrale de Traitement, UCT.
  En anglais : Central Processing Unit, CPU.

- Deux éléments principaux :
  - unité de commande (control unit),
  - unité de traitement (processing unit).

L'unité de commande I

- Permet de séquencer le déroulement des instructions.
- Effectue la recherche en mémoire de l'instruction et le décodage de l'instruction codée sous forme binaire.
- Pilote l'exécution de l'instruction.

Les composantes de l'unité de commande :

1. Compteur de programme (PC : Programme Counter) ou Compteur Ordinal (CO) :
   - registre contenant l'adresse de la prochaine instruction à exécuter ;
   - initialisé avec l'adresse de la première instruction du programme.

L'unité de commande II

2. Registre d'instruction, et décodeur d'instruction :
   - chacune des instructions à exécuter est transféré depuis la mémoire dans ce registre ;
   - puis est décodée par le décodeur.
3. Séquenceur (ou bloc logique de commande) :
   - organise l'exécution des instructions au rythme d'une horloge
   - élabore tous les signaux de synchronisation internes ou externes (bus de commande) du microprocesseur
     en fonction de l'instruction qu'il a à exécuter.

L'unité de traitement I

Circuits assurant les traitements nécessaires à l'exécution des instructions.

Composantes de l'unité de traitement :

1. Les accumulateurs :
   registres de travail qui servent à stocker une opérande au début d'une opération arithmétique et le résultat à la fin de l'opération.
2. L'Unité Arithmétique et Logique (UAL) :
   assure les fonctions logiques (ET, OU, Comparaison, Décalage, etc...) ou arithmétique (Addition, soustraction...).
3. Le registre d'état :
   - composé généralement de 8 bits
   - chaque bit est appellé indicateur d'état ; son contenu dépend du résultat de la dernière opération effectuée par l'UAL.

Le fonctionnement basique d'une opération de calcul

1. Charger une instruction depuis la mémoire
2. Charger les opérandes depuis la mémoire
3. Effectuer les calculs
4. Stocker le résultat en mémoire

Voir la démo :

`http://imss-www.upmf-grenoble.fr/~adamj/doclicence/0-FonctionnementUniteCentrale.ppt`

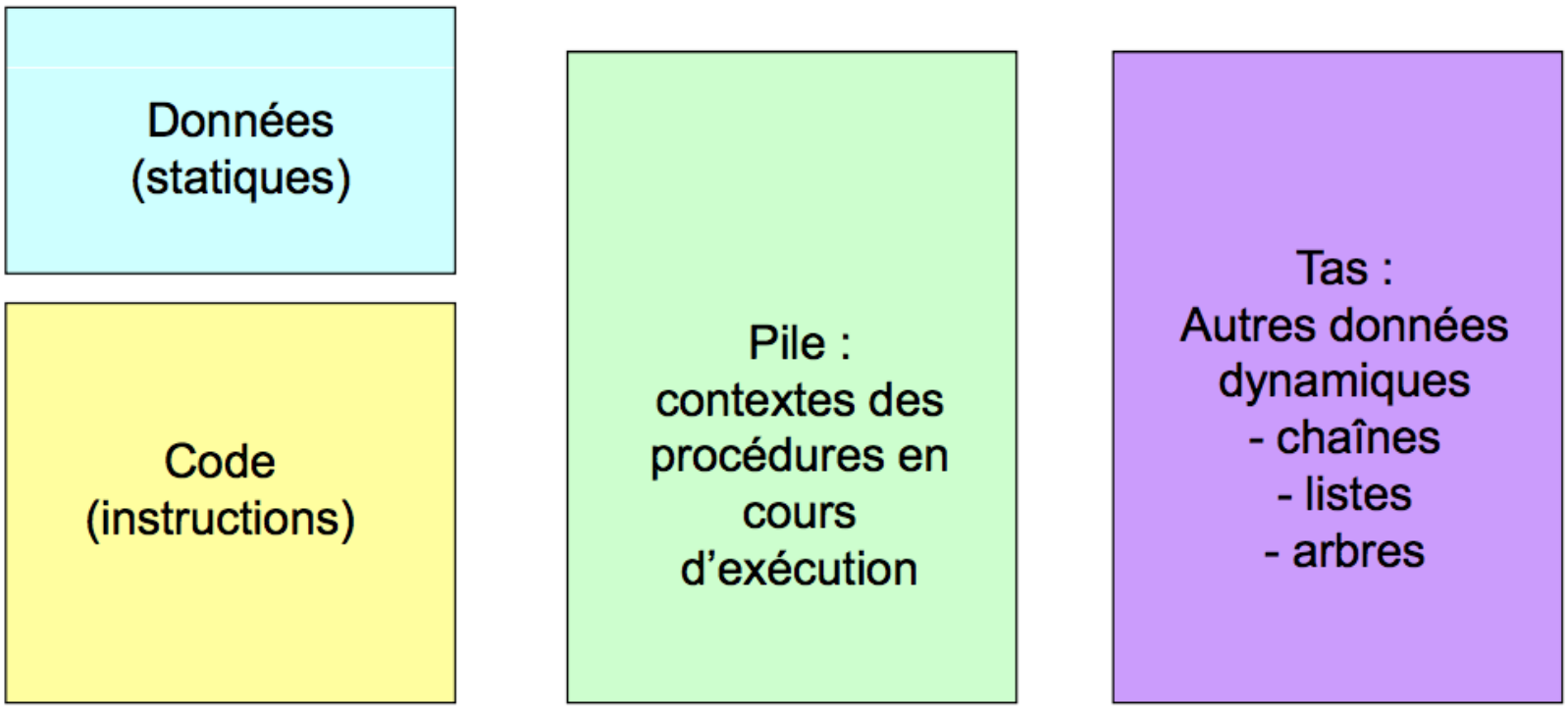
Plan

La mémoire d'un programme en cours d'exécution

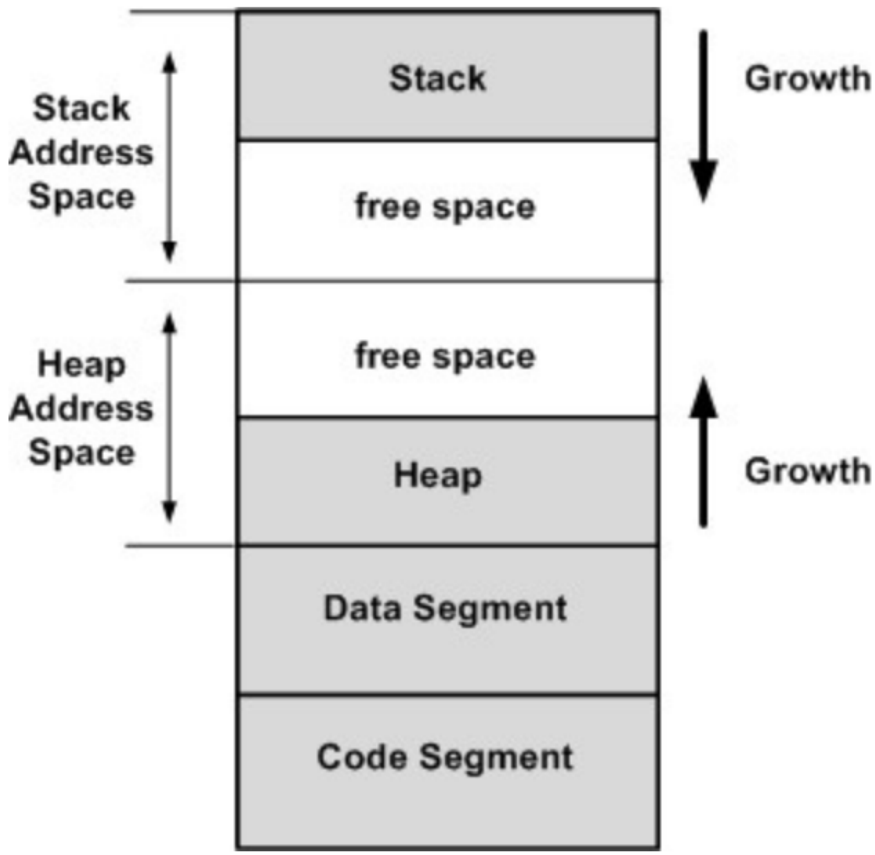
Un programme en cours d'exécution en mémoire centrale est composé de plusieurs parties, appelés « segments ».
Typiquement :

- Code (Text) : contient les instructions du programme
- Données (Data) : contient les informations statiques manipulées par le programme.
  Données statiques : informations dont on connaît la taille à la compilation (c'est-à-dire avant l'exécution du programme).
  Exemples :
  - les variables globales,
  - les chaînes de caractères constantes.
- Pile (Stack) : permet l'évaluation du programme à l'exécution.
  Y sont représentées les données crées dynamiquement à l'appel d'une procédure/fonction.
- Tas (Heap) : utilisé pour représenter les autres données dynamiques dont la durée de vie n'est pas liée à l'exécution des procédures.

Schéma d'exécution d'un programme



Organisation typique (e.g. en C) des 4 segments



Accès aux mots d'un segment

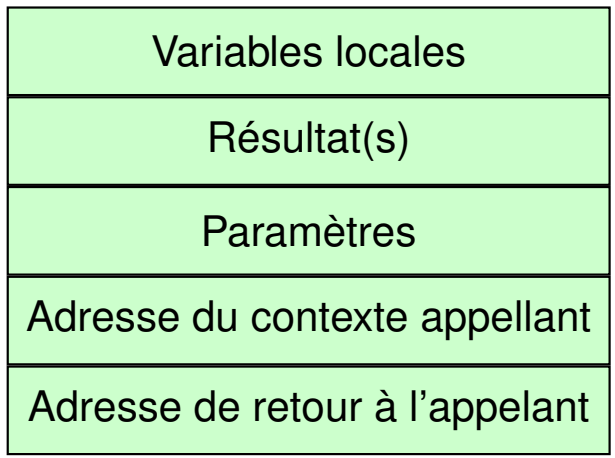
- L'adresse d'un mot est obtenue à partir :
  - de l'adresse du début du segment (la « base »)
  - de la position du mot dans le segment (le « déplacement »)
- Pour accéder à un mot le système effectue un calcul pour obtenir l'adresse effective en mémoire de chaque mot :

$$\text{AE} = \text{base} + \text{déplacement}$$

- Cette technique permet de disposer de code « relogeable », c'est-à-dire déplaçable en mémoire.
- Pour déplacer un segment, il suffit de modifier sa base pour pouvoir l'exploiter après son déplacement.

Gestion de la pile

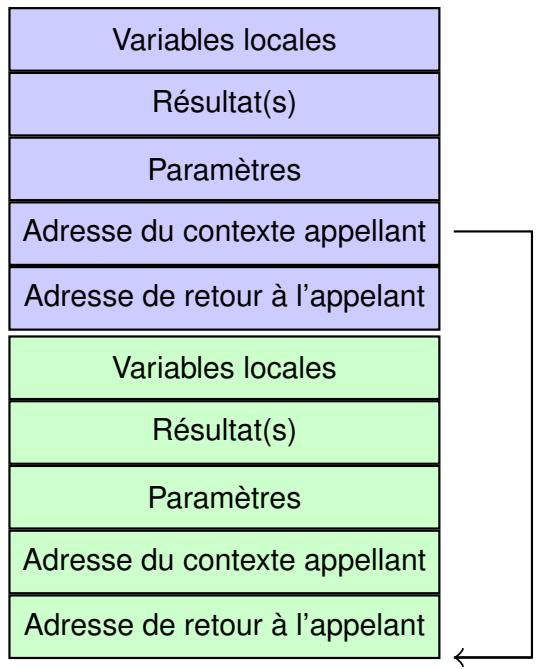
- La pile est utilisée pour empiler les contextes des procédures en cours d'exécution.
- Un contexte est structuré de la manière suivante :



- En anglais : le contexte est le « stack frame » (cadre de pile) ou « call frame » (cadre d'appel).

Gestion de la pile

- La durée de vie des informations du contexte correspond à l'exécution de la procédure qui lui correspond
- Adressage : toutes les informations sont désignées par rapport à l'adresse de début du contexte.



Appel de procédure/fonction, retour de l'appel

Appel de fonction :

- Création d'un nouveau contexte sur la pile :
  - Empiler l'adresse de retour
  - Empiler l'adresse du contexte de l'appelant
  - Empiler les paramètres
  - Créer la place pour stocker les résultat final de la fonction
  - Créer la place pour stocker les variables locales
- donner la main à la fonction appelée.

Retour d'une fonction :

- Copier le résultat de la fonction sur un registre
- Dépiler le dernier contexte
- donner la main à la fonction appelant

Gestion du tas

- L'organisation beaucoup plus anarchique :
  - Le tas est une réserve d'emplacements mémoire mise à la disposition du programme pour répondre aux demandes d'allocation dynamique (`malloc`, `new`, création d'un objet) ou de libération de la mémoire (`free`, `delete`)
  - Le tas est composé de segments de mémoire qui sont soit occupés, soit libres.
  - Les segments libres sont chaînés entre eux.

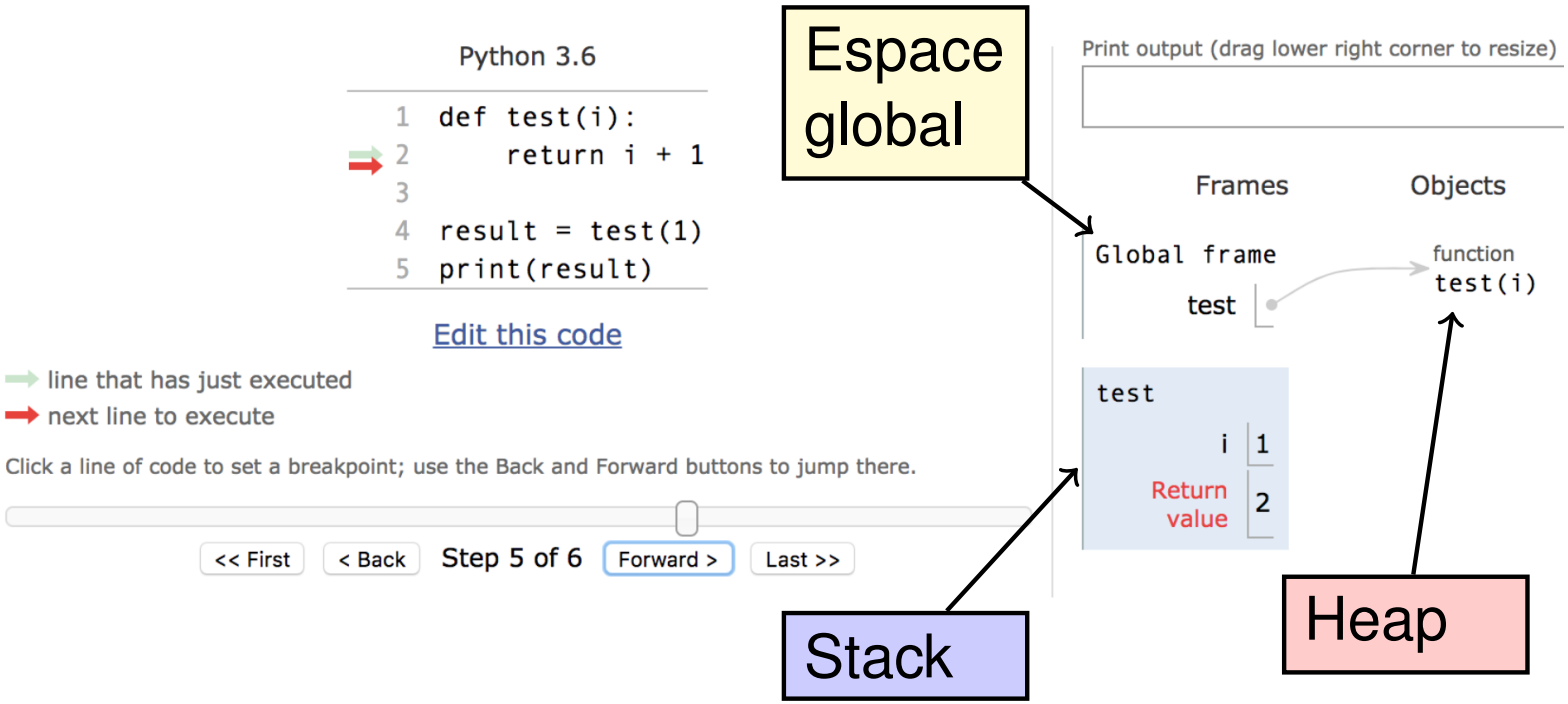
Gestion du tas (II)

- A chaque demande de mémoire, le gestionnaire du tas choisit le segment libre dans lequel il va prendre la mémoire à donner au programme.
  Différentes stratégies de choix du segment existent.
- Dans certaines situations plus aucun segment n'est suffisamment grand pour répondre à la demande.
- Il faut alors mettre en œuvre un algorithme « ramasse-miettes » (garbage collector) pour restructurer le tas.
  Ceci va nécessairement engendrer le déplacement de données en mémoire, ce qui va nécessiter de mettre à jour les pointeurs sur ces objets.

La mémoire en Python

- Espace globale :
  - Variables globales
  - Modules et fonctions
  - Dure jusqu'à quitter Python
- Pile des appels, pile des contextes, « Call Frames »
  - Variables locales à l'appel à fonction
  - Se termine lors du retour de l'appel à fonction
- Tas (Heap space) :
  - On y stocke les repertoires (« folders »)
  - On y accède de façon indirecte.

Python tutor



Lien :

`http://pythontutor.com/visualize.html#mode=edit`

Fonctions et espace globale

- La définition d'une fonction :
  - Crée une variable globale, même nom de la fonction.
  - Crée un répertoire contenant le corps de la fonction.
  - Assigne l'id du repetoire à la variable.
- Variable vs. appel de fonction :

```
>>> to_centigrade
<fun to_centigrade at 0x100498de8>
>>> to_centigrade(32)
0.0
```

Modules et espace globale

- Importer un module :
  - Crée une variable globale (avec le même nom du module).
  - Ajoute le contenu du module dans un répertoire :
    - les variables du module ;
    - les fonctions du module.
  - Assigne l'id du répertoire à la variable.

```
>>> import math
>>> math
<module 'math' from
 ↪  '/anaconda3/...math.cpython-37m-darwin.so'>
```

- Le mot clé `from` dépose le contenu dans l'espace globale.

```
>>> from test import fonction
>>> fonction
<function fonction at 0x10c58a158>
```

Contextes (Call Frames)

1. Créer un contexte pour l'appel.
2. Assigner les valeurs des arguments aux paramètres (dans le contexte).
3. Exécuter le corps de la fonction :
   - pour les variables, regarder dans le contexte ;
   - si pas trouvé, chercher une variable globale avec le même nom.
4. A la fin de l'exécution du corps de la fonction : éliminer le contexte pour cet appel.

Plan

Sources

https://alexandre-laurent.developpez.com/articles/debogage-application/

http://fsincere.free.fr/isn/python/cours_python_debugger.php

https://python-scientific-lecture-notes.developpez.com/tutoriel/notes-cours/python-debugging-code/

Avant le débogage

- Assertions.
- Tests unitaires.
- Analyse statique :
  - Analyse de types dans un langage comme C ;
  - Avertissements du compilateur.
- En **Python** (typage dynamique) :
  - `https://www.python.org/dev/peps/pep-0008/pep8`,
  - `https://www.pylint.org/pylint`,
  - `https://pypi.org/project/pyflakes/pyflakes`

Fonctionnalités d'un déboguer

- Inspecter la pile (des appels).
- Voir les valeurs des variables.
- Exécuter le programme pas à pas.
- Placer un (ou plusieurs) points d'arrêt :
  - s'arrêter à une ligne spécifique du programme,
  - le débogueur s'arrêtera avant d'exécuter cette ligne.

Débogage post-mortem

- Votre programme se plante.
  Par exemple, en **Python**, une exception est levée et pas interceptée.
- Objectif : comprendre pourquoi se plante.
- Vous redémarrez votre programme, avec même conditions initiales, avec un déboguer.

Inspection de la pile d'appels

Inspection de la pile d'appels :

- Hypothèse : les bibliotèques utilisés marchent bien ! ! !
  Le plantage est du à une mauvaise utilisation d'une fonction de la biliothèque bibliothèque.
- Après plantage, l'inspection de la pile permet de repérer cette mauvaise utilisation.

Exemple débogage pm (I)

Module `fib.py` :

```
def fib(n):
    if n < 0:
        raise ValueError
    elif n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
from fib import fib

def allFibs(n):
    print(fib(n), end=' ')
    if n >= 0:
        allFibs(n-1)

if __name__ == '__main__':
    allFibs(3)
```

```
1 > ipython
2 In [1]: %run stack_search.py
3 3 2 1 1 ValueError
```

Exemple débogage pm (II)

```
In [2] %debug
ipdb> where
...
/.../stack_search.py(4)allFibs()
      2
      3 def allFibs(n):
----> 4     print(fib(n), end=' ')
      5     if n >= 0:
      6         allFibs(n-1)
      7

/.../fib.py(3)fib()
      1 def fib(n):
      2     if n < 0:
----> 3         raise ValueError
      4     elif n == 0 or n == 1:
      5         return 1

ipdb> p n
-1
ipdb> up
ipdb> p n
0
ipdb> quit

In [3]:
```

Commandes `pdb`

%debug :	depuis `ipython` on démarre de débogueur
where :	affiche la pile des appels
p n :	affiche la valeur de la variable *n* (globale, ou contenue dans le contexte courant).
up :	se déplace vers le frame appelant.
quit :	quitte le déboguer.

Documentation de `pdb` (débogueur de Python) :

`https://docs.python.org/3/library/pdb.html`

Débogage pas à pas

- On soupçonne l'existence d'un bogue dans votre code, même si votre programme ne se plante.
- Objectif : identifier la bogue (et la corriger).

Exemple débogage pas à pas (I)

```
1  def init_listes(size):
2      """Buggy code : it is supposed
3      to produce a list of singletons lists"""
4
5      listes = [[]] * size
6      i=0
7      for liste in listes:
8          liste.append(i)
9          i+=1
10     return listes
11
12 if __name__ == '__main__':
13     listes = init_listes(10)
14     print(listes)
```

```
In [1]: %run err_listes.py
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1,
 ↪  2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2,
 ↪  3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3,
 ↪  4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [0, 1, 2, 3, 4,
 ↪  5, 6, 7, 8, 9], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]

In [2]:  %run -d err_listes.py
```

Exemple débogage pas à pas (II)

```
ipdb> step
> /.../err_listes.py(12)<module>()
     10     return listes
     11
---> 12 if __name__ == '__main__':
     13     listes = init_listes(10)
     14     print(listes)

ipdb> step ...

> /.../err_listes.py(8)init_listes()
      7     for liste in listes:
      8         liste.append(i)
----> 9         i+=1
     10     return listes
     11

ipdb> p i
0
ipdb> p listes
[[0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]

ipdb> quit
```

Exemple débogage pas à pas (III)

```
In [3]:  %run -d err_listes.py

ipdb> break 9
Breakpoint 3 at /.../err_listes.py:9
ipdb> break
Num Type         Disp Enb   Where
3   breakpoint   keep yes   at /.../err_listes.py:9

ipdb> continue
> /.../err_listes.py(9)init_listes()
      7     for liste in listes:
      8         liste.append(i)
3---> 9         i+=1
     10     return listes
     11

ipdb> p listes
[[0], [0], [0], [0], [0], [0], [0], [0], [0], [0]]
ipdb>
```

Commandes `pdb`

step : exécute une ligne, en rentrant dans les fonctions (en traversant les contextes),

next : exécute une ligne, sans rentrer dans les fonctions,

break *n* : ajoute un point d'arrêt à ligne *n*,

break `nomfonction` : ajoute un point d'arrêt à la première ligne de la fonction `nomfonction`,

break : liste de tous les points d'arrêt,

continue : continue l'exécution du programme jusqu'au prochain point d'arrêt.

Avec Spyder

démarrer le débogage du fichier,

exécuter la prochaine ligne (next),

se déplacer dans une fonction (step),

continuer jusqu'au prochain `return` (commande `pdb` : return),

continuer jusqu'au prochain point d'arrêt (continue),

arrêter le débogage (quit),

Pour placer un point d'arrêt : double-click sur le numéro de ligne :

```
 1 def init_listes(size):
 2     """Buggy code : it is supposed
 3     to produce a list of singletons lists"""
 4
 5     listes = [[]] * size
 6     i=0
 7     for liste in listes:
 8         liste.append(i)
 9         i+=1
10     return listes
11
12 if __name__ == '__main__':
13     listes = init_listes(10)
14     print(listes)
15
```