Gestion de la mémoire d'un processus

Exemple du langage C

Credits

- Ce cours de Marc de Falco

Credits

- Ce cours de Marc de Falco
- Wikipedia

Credits

- Ce cours de Marc de Falco
- Wikipedia
- Cette mise au point de Stackoverflow

Credits

- Ce cours de Marc de Falco
- Wikipedia
- Cette mise au point de Stackoverflow
- Cette présentation YouTube de Gwendal le Vaillant

Portée

- La *portée* d'un identificateur i d'un programme P est la zone du texte de P dans laquelle on peut faire référence à i sans erreur à la compilation.

Portée

- La *portée* d'un identificateur i d'un programme P est la zone du texte de P dans laquelle on peut faire référence à i sans erreur à la compilation.
- Exemple

```
int a = 1;

int f (int x)
{
    int y = x + a;
    return y;
}

int g()
{
    int z = 3;
    return z + f(z);
}

```

Portée des identificateurs

- Dans l'exemple précédent :

identificateur	**portée**
a	1-13
x	3-7
y	5-7
f	4-13
g	9-13
z	11-13

Portée des identificateurs

- Dans l'exemple précédent :

identificateur	portée
a	1-13
x	3-7
y	5-7
f	4-13
g	9-13
z	11-13

- Un identificateur peut être utilisé dès la ligne de sa déclaration : `int x = x` est en théorie possible mais pas recommandé.

Portée des identificateurs

- Dans l'exemple précédent :

identificateur	**portée**
a	1-13
x	3-7
y	5-7
f	4-13
g	9-13
z	11-13

- Un identificateur peut être utilisé dès la ligne de sa déclaration : `int x = x` est en théorie possible mais pas recommandé.
- La portée des identificateurs ne désignant pas des fonctions inclut la ligne de déclaration (donc la portée de x démarre à L3).

Portée des identificateurs

- Dans l'exemple précédent :

identificateur	portée
a	1-13
x	3-7
y	5-7
f	4-13
g	9-13
z	11-13

- Un identificateur peut être utilisé dès la ligne de sa déclaration : `int x = x` est en théorie possible mais pas recommandé.
- La portée des identificateurs ne désignant pas des fonctions inclut la ligne de déclaration (donc la portée de x démarre à L3).
- Pour les identificateurs de fonctions, le corps de la fonction est dans la portée pour permettre la récursion.

Portée des identificateurs

- Dans l'exemple précédent :

identificateur	portée
a	1-13
x	3-7
y	5-7
f	4-13
g	9-13
z	11-13

- Un identificateur peut être utilisé dès la ligne de sa déclaration : `int x = x` est en théorie possible mais pas recommandé.
- La portée des identificateurs ne désignant pas des fonctions inclut la ligne de déclaration (donc la portée de x démarre à L3).
- Pour les identificateurs de fonctions, le corps de la fonction est dans la portée pour permettre la récursion.
- Portées des variables globales : toutes les lignes après la déclaration.

Portée

Variables locales de même nom

- Deux variables locales ont même nom :

```
int f()
{
    int i = 3;
    return i;
}

int g()
{
    int i = 5;
    return i+1;
}

```

Portée

Variables locales de même nom

- Deux variables locales ont même nom :

```
int f()
{
    int i = 3;
    return i;
}

int g()
{
    int i = 5;
    return i+1;
}

```

- Deux variables peuvent avoir le même nom mais dans ce cas elles doivent avoirs des portées différentes.

Portée

Variables locales de même nom

- Deux variables locales ont même nom :

```
1  int f()
2  {
3      int i = 3;
4      return i;
5  }
6
7  int g()
8  {
9      int i = 5;
10     return i+1;
11 }
12
```

- Deux variables peuvent avoir le même nom mais dans ce cas elles doivent avoirs des portées différentes.
- L'identificateur `i` a pour portée les lignes 3-5 et 9-11 mais il désigne deux variables qui ont des portées disjointes `f.i` et `g.i`.

Portées imbriquées

- Soit la situation suivante :

```
{
 int i=3;
  {
   int i=5;
    printf("%d", i); // affiche 5
  }
 printf("%d",i); // affiche 3
}

```

Portées imbriquées

- Soit la situation suivante :

```
{
 int  i=3;
  {
   int  i=5;
    printf ("%d", i);  //  affiche  5
  }
 printf ("%d",i);  //  affiche  3
}

```

- La portée de la première variable `i` contient celle de la seconde.

Portées imbriquées

- Soit la situation suivante :

```
{
 int  i=3;
  {
   int  i=5;
    printf("%d", i); // affiche 5
  }
 printf("%d",i); // affiche 3
}

```

- La portée de la première variable `i` contient celle de la seconde.
- Lors du premier `printf`, seule la seconde variable (celle valant 5) est visible, la première existe toujours (elle occupe un espace mémoire) mais elle n'est pas accessible.

Portées imbriquées

- Soit la situation suivante :

```
1 {
2  int i=3;
3   {
4    int i=5;
5     printf("%d", i); // affiche 5
6   }
7  printf("%d",i); // affiche 3
8 }
9
```

- La portée de la première variable `i` contient celle de la seconde.
- Lors du premier `printf`, seule la seconde variable (celle valant 5) est visible, la première existe toujours (elle occupe un espace mémoire) mais elle n'est pas accessible.
- Après la ligne 6, la seconde variable est détruite (elle n'occupe plus d'espace mémoire) et il ne reste plus que la première.

Durée de vie

- La *durée de vie* d'une variable est la période de l'exécution du programme durant laquelle la variable est présente en mémoire. C'est une notion souvent (mais pas toujours) équivalente à celle de *portée*.

Durée de vie

- La *durée de vie* d'une variable est la période de l'exécution du programme durant laquelle la variable est présente en mémoire. C'est une notion souvent (mais pas toujours) équivalente à celle de *portée*.
- Pour une variable globale, la durée de vie est l'intégralité du programme.

Durée de vie

- La *durée de vie* d'une variable est la période de l'exécution du programme durant laquelle la variable est présente en mémoire. C'est une notion souvent (mais pas toujours) équivalente à celle de *portée*.
- Pour une variable globale, la durée de vie est l'intégralité du programme.
- La durée de vie d'une variable locale non statique est la portée de l'identificateur qui lui est associé. L'emplacement mémoire des données de cette variable est alloué au début sa portée et libéré à la fin automatiquement.

Durée de vie

- La *durée de vie* d'une variable est la période de l'exécution du programme durant laquelle la variable est présente en mémoire. C'est une notion souvent (mais pas toujours) équivalente à celle de *portée*.
- Pour une variable globale, la durée de vie est l'intégralité du programme.
- La durée de vie d'une variable locale non statique est la portée de l'identificateur qui lui est associé. L'emplacement mémoire des données de cette variable est alloué au début sa portée et libéré à la fin automatiquement.
- Les variables locales (déclarées comme `static`) ont une durée de vie supérieure à leur portée.

Durée de vie

- La *durée de vie* d'une variable est la période de l'exécution du programme durant laquelle la variable est présente en mémoire. C'est une notion souvent (mais pas toujours) équivalente à celle de *portée*.
- Pour une variable globale, la durée de vie est l'intégralité du programme.
- La durée de vie d'une variable locale non statique est la portée de l'identificateur qui lui est associé. L'emplacement mémoire des données de cette variable est alloué au début sa portée et libéré à la fin automatiquement.
- Les variables locales (déclarées comme `static`) ont une durée de vie supérieure à leur portée.
- Les variables locales allouées dynamiquement (par `malloc`) peuvent exister en dehors du bloc où elles ont été déclarées : leur durée de vie peut être supérieure à leur portée (si elles ne sont pas libérées dans le bloc où a lieu l'allocation).

Les zones de la mémoire virtuelle d'un processus

- Chaque fois qu'un programme C est exécuté, de la mémoire est allouée dans la RAM pour l'exécution. Cette mémoire est utilisée pour stocker le code machine, les variables du programme, les constantes etc...

Les zones de la mémoire virtuelle d'un processus

- Chaque fois qu'un programme C est exécuté, de la mémoire est allouée dans la RAM pour l'exécution. Cette mémoire est utilisée pour stocker le code machine, les variables du programme, les constantes etc...
- Les différents types de variables : locales, globales, statiques sont enregistrées dans différentes zones (ou *segments de mémoire*) de la mémoire virtuelle du processus (*i.e.* programme en cours d'exécution).

Les zones de la mémoire virtuelle d'un processus

- Chaque fois qu'un programme C est exécuté, de la mémoire est allouée dans la RAM pour l'exécution. Cette mémoire est utilisée pour stocker le code machine, les variables du programme, les constantes etc...
- Les différents types de variables : locales, globales, statiques sont enregistrées dans différentes zones (ou *segments de mémoire*) de la mémoire virtuelle du processus (*i.e.* programme en cours d'exécution).
- Les 4 segments de mémoire pour un programme sont : le *segment de code* (ou de *texte*), le *segment de données* (lui-même divisé en C en le segment des variables non initialisées, celui des variables initialisées, celui des constantes -en lecture seule-) le *segment de pile* et enfin le *segment de tas*.

Retour sur la RAM

- La mémoire d'un ordinateur contient à la fois les programmes à exécuter (les différentes instructions) et les données que ces programme doivent manipuler.

Retour sur la RAM

- La mémoire d'un ordinateur contient à la fois les programmes à exécuter (les différentes instructions) et les données que ces programme doivent manipuler.
- Les octets stockés dans la mémoire ne sont pas étiquetés comme étant du code ou des données.

Retour sur la RAM

- La mémoire d'un ordinateur contient à la fois les programmes à exécuter (les différentes instructions) et les données que ces programme doivent manipuler.
- Les octets stockés dans la mémoire ne sont pas étiquetés comme étant du code ou des données.
- C'est le *contexte* qui détermine si on les considère comme du code (à exécuter) ou des données (à manipuler).

Retour sur la RAM

- La mémoire d'un ordinateur contient à la fois les programmes à exécuter (les différentes instructions) et les données que ces programme doivent manipuler.
- Les octets stockés dans la mémoire ne sont pas étiquetés comme étant du code ou des données.
- C'est le *contexte* qui détermine si on les considère comme du code (à exécuter) ou des données (à manipuler).
- Grossièrement, la mémoire peut-être vue comme un tableau de cases mémoires élémentaires appelées *mots mémoires*.
  La taille de ces mots varie selon les machines de 8)à 64 bits.

Retour sur la RAM

- La mémoire d'un ordinateur contient à la fois les programmes à exécuter (les différentes instructions) et les données que ces programme doivent manipuler.
- Les octets stockés dans la mémoire ne sont pas étiquetés comme étant du code ou des données.
- C'est le *contexte* qui détermine si on les considère comme du code (à exécuter) ou des données (à manipuler).
- Grossièrement, la mémoire peut-être vue comme un tableau de cases mémoires élémentaires appelées *mots mémoires*.
  La taille de ces mots varie selon les machines de 8)à 64 bits.
- Traditionnellement, on représente ce tableau mémoire verticalement avec les premières adresses de la mémoire en bas du schéma.

Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.

Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.
- Les données sont réparties en trois segments :

Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.
- Les données sont réparties en trois segments :
  - Les constantes peuvent se trouver dans le segment de texte ou parfois dans le segment .rodata.

Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.
- Les données sont réparties en trois segments :
  - Les constantes peuvent se trouver dans le segment de texte ou parfois dans le segment .rodata.
  - les segments .bss ou .data contiennent les variables globales ou statiques. Selon que les variables globales sont initialisées ou non non, elle vont dans l'un ou l'autre de ces segments de données.

Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.
- Les données sont réparties en trois segments :
  - Les constantes peuvent se trouver dans le segment de texte ou parfois dans le segment .rodata.
  - les segments .bss ou .data contiennent les variables globales ou statiques. Selon que les variables globales sont initialisées ou non non, elle vont dans l'un ou l'autre de ces segments de données.
  - Les tailles de ces zones sont fixes car connues à la compilation.

Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.
- Les données sont réparties en trois segments :
  - Les constantes peuvent se trouver dans le segment de texte ou parfois dans le segment .rodata.
  - les segments .bss ou .data contiennent les variables globales ou statiques. Selon que les variables globales sont initialisées ou non non, elle vont dans l'un ou l'autre de ces segments de données.
  - Les tailles de ces zones sont fixes car connues à la compilation.
- Le *segment de pile* contient l'espace mémoire alloué dynamiquement par un programme sans intervention explicite du programmeur. La pile est utilisée au moment de l'*appel de fonction* pour stocker paramètres et variables locales de ces fonctions.

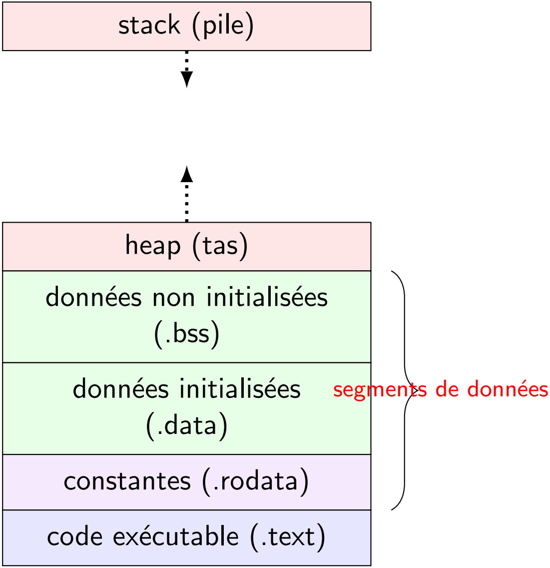
Découpage en segments

Schématiquement, l'espace mémoire d'un processus est découpé en 4 parties appelées *segments de mémoire*

- Le *segment de code* contient les instructions du programme en langage machine.
- Les données sont réparties en trois segments :
    - Les constantes peuvent se trouver dans le segment de texte ou parfois dans le segment .rodata.
    - les segments .bss ou .data contiennent les variables globales ou statiques. Selon que les variables globales sont initialisées ou non non, elle vont dans l'un ou l'autre de ces segments de données.
    - Les tailles de ces zones sont fixes car connues à la compilation.
- Le *segment de pile* contient l'espace mémoire alloué dynamiquement par un programme sans intervention explicite du programmeur. La pile est utilisée au moment de l'*appel de fonction* pour stocker paramètres et variables locales de ces fonctions.
- Le *segment de tas* contient l'espace mémoire alloué dynamiquement par un programme avec intervention explicite du programmeur

Une bonne explication se trouve sur cette page LinkedIn notamment pour les stacks frame

Organisation de la mémoire en C♡



- Les segments de tas et de pile sont souvent représentés progressant l'un vers l'autre :
  - la pile avec des adresses décroissantes,
  - le tas avec des adresses croissantes...

  ou l'inverse !
- La taille des segments .text, .rodata, .data et .bss est déterminée à la compilation et ne change pas à l'exécution.

Accès aux mots de la mémoire

En première approximation, la mémoire d'un ordinateur peut être vue comme un tableau de case mémoires élémentaires appelées `mots mémoires` et de taille entre 8 et 64 bits.

- L'adresse d'un mot mémoire s'obtient en additionnant l'adresse du début du segment memoire où il se trouve (code, données, tas, pile), qu'on appelle *base*, et la position de ce mot dans le segment, qu'on appelle *déplacement*.

≪ Informatique, cours et exercices corrigés, MPI/MPII ≫

Accès aux mots de la mémoire

En première approximation, la mémoire d'un ordinateur peut être vue comme un tableau de case mémoires élémentaires appelées `mots mémoires` et de taille entre 8 et 64 bits.

- L'adresse d'un mot mémoire s'obtient en additionnant l'adresse du début du segment memoire où il se trouve (code, données, tas, pile), qu'on appelle *base*, et la position de ce mot dans le segment, qu'on appelle *déplacement*.
- Cet adressage dit *relatif* à une base, permet à un programme de s'exécuter de la même façon dans n'importe quelle partie de la mémoire d'un ordinateur : pas besoin de changer les instructions d'accès en lecture ou écriture, l'OS fera automatiquement les translations nécessaires.

≪ Informatique, cours et exercices corrigés, MPI/MPII ≫

Le segment de code

Appelé aussi *segment de texte*.

- C'est la zone de la mémoire du processus qui contient les instructions à exécuter.

Le segment de code

Appelé aussi *segment de texte*.

- C'est la zone de la mémoire du processus qui contient les instructions à exécuter.
- Souvent en lecture seule pour éviter d'être accidentellement réécrit.

Le segment de code

Appelé aussi *segment de texte*.

- C'est la zone de la mémoire du processus qui contient les instructions à exécuter.
- Souvent en lecture seule pour éviter d'être accidentellement réécrit.
- Placé dans les adresses inférieures pour que des dépassement de pile ou de tas n'écrivent pas dedans.

Le segment de code

Appelé aussi *segment de texte*.

- C'est la zone de la mémoire du processus qui contient les instructions à exécuter.
- Souvent en lecture seule pour éviter d'être accidentellement réécrit.
- Placé dans les adresses inférieures pour que des dépassement de pile ou de tas n'écrivent pas dedans.
- Certaines implémentation de C y placent aussi les chaînes de caractères en lecture seule comme le « Hello » de `printf("Hello")`.

Le segment de code

Appelé aussi *segment de texte*.

- C'est la zone de la mémoire du processus qui contient les instructions à exécuter.
- Souvent en lecture seule pour éviter d'être accidentellement réécrit.
- Placé dans les adresses inférieures pour que des dépassement de pile ou de tas n'écrivent pas dedans.
- Certaines implémentation de C y placent aussi les chaînes de caractères en lecture seule comme le « Hello » de `printf("Hello")`.
- Le *segment rodata* contient les constantes (et peut contenir les constantes chaînes).

```c
// va en segment rodata
const int x = 3;

```

Les segment de données .data et .bss

Souvent représentés en-dessous du tas. En tout cas jamais entre pile et tas.

Segment de données initialisées ou segment `.data`. Contient les variables initialisées globales (y compris des pointeurs) et les variables locales statiques initialisées.

Les segment de données .data et .bss

Souvent représentés en-dessous du tas. En tout cas jamais entre pile et tas.

Segment de données initialisées
: ou segment `.data`. Contient les variables initialisées globales (y compris des pointeurs) et les variables locales statiques initialisées.

Segment des données non initialisées
: ou segment `.bss` (pour « block starting symbol ») contient les variables globales et variables locales statiques non initialisées OU les variables ou pointeurs initialisés à 0 ou NULL.

Exemple

```
char s[] = "hello world";
int debug=1;
const char* string = "hello world";// ptr sur type const char
float x;

int main(){
  return 0;
}
```

- La chaîne de caractères `s` comme la variable entière `debug` sont dans le segment de données (.data).

Exemple

```
char s[] = "hello world";
int debug=1;
const char* string = "hello world";// ptr sur type const char
float x;

int main(){
  return 0;
}
```

- La chaîne de caractères `s` comme la variable entière `debug` sont dans le segment de données (.data).
- Le pointeur de caractère `string` est aussi dans le segment de données mais la chaîne de caractères littérale `"hello world"` est dans la zone en lecture seule (.rodata).

Exemple

```c
char s[] = "hello world";
int debug=1;
const char* string = "hello world";// ptr sur type const char
float x;

int main(){
  return 0;
}

```

- La chaîne de caractères `s` comme la variable entière `debug` sont dans le segment de données (.data).
- Le pointeur de caractère `string` est aussi dans le segment de données mais la chaîne de caractères littérale `"hello world"` est dans la zone en lecture seule (.rodata).
- La variable globale `x` non initialisée est dans le segment `.bss`.

Exemple

```c
char s[] = "hello world";
int debug=1;
const char* string = "hello world";// ptr sur type const char
float x;

int main(){
  return 0;
}

```

- La chaîne de caractères `s` comme la variable entière `debug` sont dans le segment de données (.data).
- Le pointeur de caractère `string` est aussi dans le segment de données mais la chaîne de caractères littérale `"hello world"` est dans la zone en lecture seule (.rodata).
- La variable globale `x` non initialisée est dans le segment `.bss`.
- La fonction `main` est dans le segment de texte `.text`

BSS : compléments

- Le segment BSS ne prend aucune place dans le fichier objet. C'est pour cela qu'on ne trouve pas les variables stockées dans le BSS avec un utilitaire comme `objdump`.

BSS : compléments

- Le segment BSS ne prend aucune place dans le fichier objet. C'est pour cela qu'on ne trouve pas les variables stockées dans le BSS avec un utilitaire comme `objdump`.
- C'est le *chargeur de programme* qui, au moment du chargement, initialise la mémoire allouée pour le segment BSS.

BSS : compléments

- Le segment BSS ne prend aucune place dans le fichier objet. C'est pour cela qu'on ne trouve pas les variables stockées dans le BSS avec un utilitaire comme `objdump`.
- C'est le *chargeur de programme* qui, au moment du chargement, initialise la mémoire allouée pour le segment BSS.
- L'information sur la place que prendra le segment BSS à l'exécution est, elle, stockée dans le fichier objet.

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;
  - les arguments passés à la fonction ;

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;
  - les arguments passés à la fonction ;
  - un espace de stockage pour la valeur de retour ;

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;
  - les arguments passés à la fonction ;
  - un espace de stockage pour la valeur de retour ;
  - l'adresse (dite *de retour*) d'où on a appelé la fonction.

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;
  - les arguments passés à la fonction ;
  - un espace de stockage pour la valeur de retour ;
  - l'adresse (dite *de retour*) d'où on a appelé la fonction.
- Les variables stockées dans la pile pour une fonction sont détruites dès que l'exécution de la fonction termine. Allocation et libération sont automatiques.

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;
  - les arguments passés à la fonction ;
  - un espace de stockage pour la valeur de retour ;
  - l'adresse (dite *de retour*) d'où on a appelé la fonction.
- Les variables stockées dans la pile pour une fonction sont détruites dès que l'exécution de la fonction termine. Allocation et libération sont automatiques.
- Il y a une limite (8 Mio sous Linux) à la taille de la pile qui dépend de plusieurs facteurs comme le langage de programmation, de l'OS, du traitement multithread et de la quantité de RAM disponible.

La pile

- Le segment de pile est utilisé pour stocker les arguments ou les variables qui sont créées dans une fonction . On y empile les *stackframe* qui contiennent :
  - les variables locales à la fonction considérée (y compris les pointeurs locaux) ;
  - les arguments passés à la fonction ;
  - un espace de stockage pour la valeur de retour ;
  - l'adresse (dite *de retour*) d'où on a appelé la fonction.
- Les variables stockées dans la pile pour une fonction sont détruites dès que l'exécution de la fonction termine. Allocation et libération sont <u>automatiques</u>.
- Il y a une limite (8 Mio sous Linux) à la taille de la pile qui dépend de plusieurs facteurs comme le langage de programmation, de l'OS, du traitement multithread et de la quantité de RAM disponible.
- Alors que les noms des variables globales sont bien connus dans le segment de données, les noms des variables locales et des arguments ne sont pas des informations stockées dans la pile.

Stack overflow

- Un *dépassement de pile* (d'exécution) (pour *stack overflow*) est un bug causé par un processus qui, lors de l'écriture dans la pile d'exécution, écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus.

Stack overflow

- Un *dépassement de pile* (d'exécution) (pour *stack overflow*) est un bug causé par un processus qui, lors de l'écriture dans la pile d'exécution, écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus.
- Récursivité infinie

```
void a()
{
    a(); // nb appels imbriqués non borné
}

int main()
{
    a();// trop gd nb appels imbriqués
    return 0;
}// Erreur de segmentation (core dumped)

```

La pile devient pleine et une erreur de dépassement de pile est soulevée

Stack frame

- Chaque fonction pousse dans la pile des informations comme la valeur des arguments, la valeur des variables locales et l'adresse de retour

Stack frame

- Chaque fonction pousse dans la pile des informations comme la valeur des arguments, la valeur des variables locales et l'adresse de retour
- Cet ensemble d'informations associées à la fonction est une *trame de pile* (pour *stack frame* en anglais).

Stack frame

- Chaque fonction pousse dans la pile des informations comme la valeur des arguments, la valeur des variables locales et l'adresse de retour
- Cet ensemble d'informations associées à la fonction est une *trame de pile* (pour *stack frame* en anglais).
- Une stack frame contient au moins un espace pour stocker la valeur retournée par la fonction.

Stack frame

- Chaque fonction pousse dans la pile des informations comme la valeur des arguments, la valeur des variables locales et l'adresse de retour
- Cet ensemble d'informations associées à la fonction est une *trame de pile* (pour *stack frame* en anglais).
- Une stack frame contient au moins un espace pour stocker la valeur retournée par la fonction.
- La pile est donc organisée par empilement/dépilement de stack frame.

Stack frame

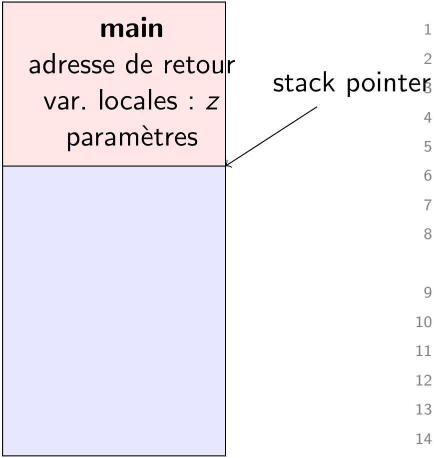
- Chaque fonction pousse dans la pile des informations comme la valeur des arguments, la valeur des variables locales et l'adresse de retour
- Cet ensemble d'informations associées à la fonction est une *trame de pile* (pour *stack frame* en anglais).
- Une stack frame contient au moins un espace pour stocker la valeur retournée par la fonction.
- La pile est donc organisée par empilement/dépilement de stack frame.
- Un registre du processeur, le *pointeur de pile* (Stack Pointer), pointe sur le sommet de la pile et est mis à jour après chaque empilement.

Stack frame

- Chaque fonction pousse dans la pile des informations comme la valeur des arguments, la valeur des variables locales et l'adresse de retour
- Cet ensemble d'informations associées à la fonction est une *trame de pile* (pour *stack frame* en anglais).
- Une stack frame contient au moins un espace pour stocker la valeur retournée par la fonction.
- La pile est donc organisée par empilement/dépilement de stack frame.
- Un registre du processeur, le *pointeur de pile* (Stack Pointer), pointe sur le sommet de la pile et est mis à jour après chaque empilement.
- Dans chaque stack frame, les informations sont elles-mêmes stockées sous forme de pile. Il y a donc aussi un *pointeur de trame* (Frame Pointer) qui pointe sur le sommet de la trame.

Stack

A la base de la pile, on trouve la stack frame de `main` avec ses arguments et variables.



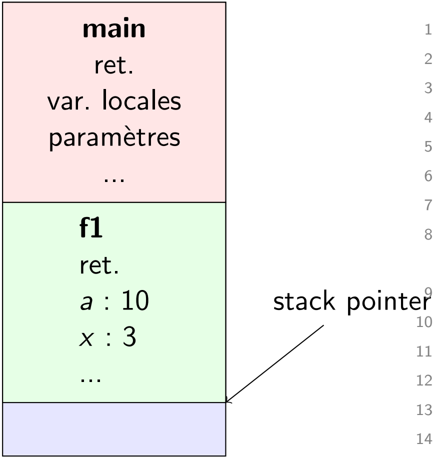

```c
void f2(int c){
  return c+3;
}


void f1(int x){
  int a = 10;// local
  return f2(a+x); // adresse retour
}


void main(){
  int z = f1(3);
}
```

Stack

A l'appel `f1(3)` , une autre frame stack est ajoutée à la pile.

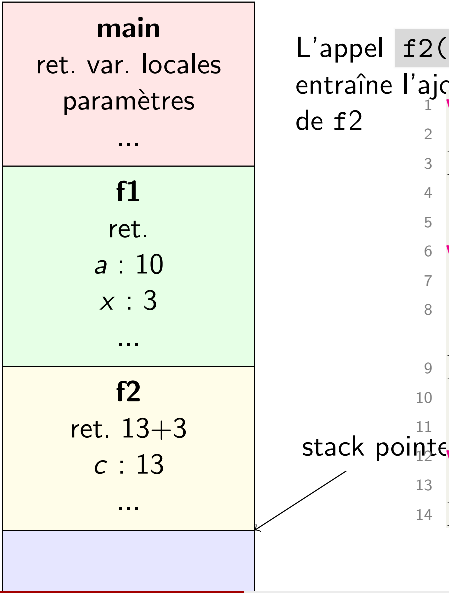



```c
void f2(int c){
  return c+3;
}


void f1(int x){
  int a = 10;// local
  return f2(a+x); // adresse
      retour
}


void main(){
  int z = f1(3);
}
```

Stack : appel `f2(10+3)`



L'appel `f2(a+x)` entraîne l'ajout de la stack frame de `f2`

```
void f2(int c){
  return c+3;
}


void f1(int x){
  int a = 10;// local
  return f2(a+x); // adresse
      retour
}


void main(){
  int z = f1(3);
}
```

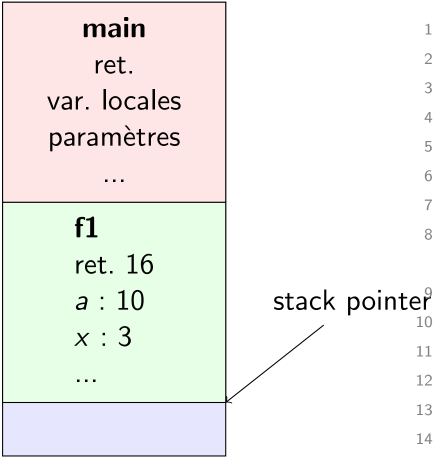
Dépilement

- Après l'exécution de `f2(a+x)`, le programme dépile la stack frame de `f2` et revient à l'instruction écrite en L4.

Dépilement

- Après l'exécution de `f2(a+x)`, le programme dépile la stack frame de `f2` et revient à l'instruction écrite en L4.
- L'exécution de `f1(3)` termine et le programme revient en L12.

Dépilement de la trame de `f2`

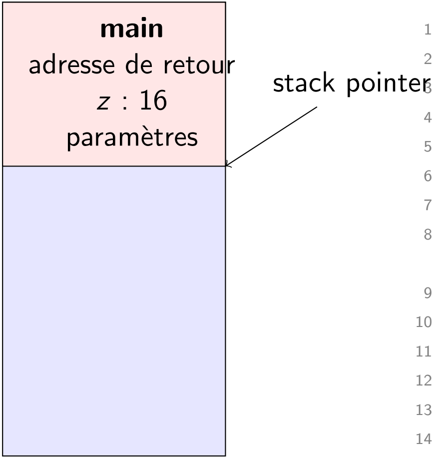



```c
void f2(int c){
  return c+3;
}


void f1(int x){
  int a = 10;// local
  return f2(a+x); // adresse
      retour
}


void main(){
  int z = f1(3);
}
```

Dépilement de la trame de `f1`




```c
void f2(int c){
  return c+3;
}


void f1(int x){
  int a = 10;// local
  return f2(a+x); // adresse retour
}


void main(){
  int z = f1(3);
}
```

Le segment de tas

- Ce segment est conçu pour supporter l'allocation dynamique de mémoire typiquement par `malloc, calloc, realloc`.

Le segment de tas

- Ce segment est conçu pour supporter l'allocation dynamique de mémoire typiquement par `malloc, calloc, realloc`.
- La taille des variables sur le tas n'est en principe pas limitée sinon par la taille de la RAM.

Le segment de tas

- Ce segment est conçu pour supporter l'allocation dynamique de mémoire typiquement par `malloc, calloc, realloc`.
- La taille des variables sur le tas n'est en principe pas limitée sinon par la taille de la RAM.
- Les variables créées sur le tas sont accessible de n'importe où dans le programme : leur portée est globale.

Le segment de tas

- Ce segment est conçu pour supporter l'allocation dynamique de mémoire typiquement par `malloc, calloc, realloc`.
- La taille des variables sur le tas n'est en principe pas limitée sinon par la taille de la RAM.
- Les variables créées sur le tas sont accessible de n'importe où dans le programme : leur portée est globale.
- Les variables sur le tas peuvent être redimensionnées.

Le segment de tas

- Ce segment est conçu pour supporter l'allocation dynamique de mémoire typiquement par `malloc, calloc, realloc`.
- La taille des variables sur le tas n'est en principe pas limitée sinon par la taille de la RAM.
- Les variables créées sur le tas sont accessible de n'importe où dans le programme : leur portée est globale.
- Les variables sur le tas peuvent être redimensionnées.
- Tout emplacement alloué doit être libéré avec `free`. Une règle simple : celui qui alloue est celui qui libère.

Erreurs de libération

Dans ce programme, on veut libérer un objet (l'adresse de `a`) qui n'est pas sur le tas.

```c
#include <stdlib.h>

int main()//exemple  marc de falco
{
    int a;
    free(&a);
    return 0;
}
```

A la compilation avec `-Wall`, on obtient un avertissement `attempt to free a non-heap object 'a'`.
A l'exécution, on obtient une erreur

```
munmap_chunk(): invalid pointer
Abandon (core dumped)
```

Erreurs de libération

Dans le programme suivant, pas d'erreur de compilation car on appelle `free` avec un objet qui est sur le tas. Cependant cet objet n'a pas été explicitement alloué par un `malloc`. On obtient une erreur d'exécution.

```c
#include <stdlib.h>

int main()// exemple marc de falco
{
    char *a = malloc(2);
    free(&(a[1]));
    return 0;
}
```

```
free(): invalid pointer
Abandon (core dumped)
```

Double libération

On libère deux fois le même objet qui n'a fait l'objet que d'une allocation. On obtient une erreur de double libération.

```c
#include <stdlib.h>

int main()// exemple marc de falco
{
    char *a = malloc(2);
    free(a);
    free(a);// double libération
    return 0;
}
```

```
free(): double free detected in tcache 2
Abandon (core dumped)
```

Croissance et décroissance des adresses

- La pile est souvent située dans la partie haute de la mémoire (les adresses élevées). Les adresses utilisées sont alors décroissantes

Croissance et décroissance des adresses

- La pile est souvent située dans la partie haute de la mémoire (les adresses élevées). Les adresses utilisées sont alors décroissantes
- Un registre du processeur, le « pointeur de pile » (stack pointer) suit la progression du sommet de la pile. Il est mis à jour chaque fois qu'une nouvelle valeur est empilée (donc à chaque nouvel appel de fonction).

Croissance et décroissance des adresses

- La pile est souvent située dans la partie haute de la mémoire (les adresses élevées). Les adresses utilisées sont alors décroissantes
- Un registre du processeur, le « pointeur de pile » (stack pointer) suit la progression du sommet de la pile. Il est mis à jour chaque fois qu'une nouvelle valeur est empilée (donc à chaque nouvel appel de fonction).
- Le tas démarre souvent à la fin du segment BSS (les variables globales non initialisées). Et les adresses utilisées successives grossissent alors.

Croissance et décroissance des adresses

- La pile est souvent située dans la partie haute de la mémoire (les adresses élevées). Les adresses utilisées sont alors décroissantes
- Un registre du processeur, le « pointeur de pile » (stack pointer) suit la progression du sommet de la pile. Il est mis à jour chaque fois qu'une nouvelle valeur est empilée (donc à chaque nouvel appel de fonction).
- Le tas démarre souvent à la fin du segment BSS (les variables globales non initialisées). Et les adresses utilisées successives grossissent alors.
- Suivant les implémentations, les adresses du tas peuvent décroître et celles de la pile croître.

Croissance et décroissance des adresses

- La pile est souvent située dans la partie haute de la mémoire (les adresses élevées). Les adresses utilisées sont alors décroissantes
- Un registre du processeur, le « pointeur de pile » (stack pointer) suit la progression du sommet de la pile. Il est mis à jour chaque fois qu'une nouvelle valeur est empilée (donc à chaque nouvel appel de fonction).
- Le tas démarre souvent à la fin du segment BSS (les variables globales non initialisées). Et les adresses utilisées successives grossissent alors.
- Suivant les implémentations, les adresses du tas peuvent décroître et celles de la pile croître.
- Quoi qu'il en soit, traditionnellement, les adresses d'une zone sont présentées comme progressant vers l'autre zone.

Croissance et décroissance des adresses

- La pile est souvent située dans la partie haute de la mémoire (les adresses élevées). Les adresses utilisées sont alors décroissantes
- Un registre du processeur, le « pointeur de pile » (stack pointer) suit la progression du sommet de la pile. Il est mis à jour chaque fois qu'une nouvelle valeur est empilée (donc à chaque nouvel appel de fonction).
- Le tas démarre souvent à la fin du segment BSS (les variables globales non initialisées). Et les adresses utilisées successives grossissent alors.
- Suivant les implémentations, les adresses du tas peuvent décroître et celles de la pile croître.
- Quoi qu'il en soit, traditionnellement, les adresses d'une zone sont présentées comme progressant vers l'autre zone.
- Quand les pointeurs de la pile et du tas se rejoignent, c'est le signe qu'il n'y a plus de mémoire disponible pour le programme.

Croissance et décroissance des adresses

Considérons la fonction récursive `f` :

```
void f(int k){// exemple inès klimann
  int n = 3;
  int *p = malloc(sizeof(int));

   if(k==0) return;
   printf("k = %d, &n : %p, p : %p\n", k, &n, p);
  f(k-1);
  free(p);
}

int main(){
  f(5);
}
```

- la variable locale `n` est dans la pile

Croissance et décroissance des adresses

Considérons la fonction récursive `f` :

```
void f(int k){// exemple inès klimann
  int n = 3;
  int *p = malloc(sizeof(int));

  if (k==0) return;
  printf ("k = %d, &n : %p, p : %p\n", k, &n, p);
  f(k-1);
  free(p);
}

int main(){
  f(5);
}
```

- la variable locale `n` est dans la pile
- la variable `p` est dans le tas.

Croissance et décroissance des adresses dans la pile et le tas

Résultat :

```
k = 5, &n : 0x7ffde641144c, p : 0x55aab027e260
k = 4, &n : 0x7ffde641140c, p : 0x55aab027e690
k = 3, &n : 0x7ffde64113cc, p : 0x55aab027e6b0
k = 2, &n : 0x7ffde641138c, p : 0x55aab027e6d0
k = 1, &n : 0x7ffde641134c, p : 0x55aab027e6f0
```

Les adresses de la pile (`&n`) décroissent

$$44c, 40c, 3cc, 38c, 34c$$

celle du tas (`p`) augmentent

$$60, 90, b0, d0, f0$$

et les adresses de piles sont plus grandes que celles de tas

$$0x7ffde641134c > 0x55aab027e6f0$$

Segments et protection

- Lorsqu'un programme s'exécute, il possède un environnement mémoire constitué de plusieurs segments. Certains d'entre eux sont en lecture seule.

Segments et protection

- Lorsqu'un programme s'exécute, il possède un environnement mémoire constitué de plusieurs segments. Certains d'entre eux sont en lecture seule.
- Cette division de la mémoire allouée au programme permet de protéger la mémoire. L'unité (physique) de gestion de la mémoire sait quelles zones sont accessibles au programme et quels sont les droits de ces zones.

Segments et protection

- Lorsqu'un programme s'exécute, il possède un environnement mémoire constitué de plusieurs segments. Certains d'entre eux sont en lecture seule.
- Cette division de la mémoire allouée au programme permet de protéger la mémoire. L'unité (physique) de gestion de la mémoire sait quelles zones sont accessibles au programme et quels sont les droits de ces zones.
- En cas d'accès anormal, le matériel provoque une erreur qui remonte au système d'exploitation, lequel provoque alors l'arrêt du programme avec l'erreur `Segmentation fault` (erreur de segmentation).

Exemple de `Segmentation fault`

La fonction `scanf` cherche à récupérer un entier sur l'entrée standard et stocke cette valeur à l'adresse `x`.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *x;// non initialisée
     printf("entrer un nombre : ");
    scanf("%d", x);
    return (EXIT_SUCCESS);
}//Erreur de segmentation (core dumped)
```

Comme `x` contient n'importe quoi, le programme tente sans doute d'écrire dans une zone interdite pour lui.

Exemple de `Segmentation fault`

Lecture à l'adresse zéro : cela provoque toujours une erreur.

```c
int main()// exemple marc de falco
{
    int *a = 0;

    //return a[0];
    return *a;
}//Erreur de segmentation (core dumped)
```

Exemple de `Segmentation fault`

Le code de la fonction `main` est dans le segment de code, lequel est read-only. Le pointeur `a` désigne le début du corps de `main`. Essayer d'écrire 0 (ou n'importe quoi d'autre) à cette adresse, provoque une erreur.

```c
int main()// exemple marc de falco
{
    int *a = (int*) (&main);
    // a pointe sur le corps de la fonction main

    *a = 0;

    return 0;
}
```

Exemple de `Segmentation fault`

Dans cet exemple, on crée une chaîne de caractère `"hello"`. Elle est placée dans la section `.rodata` qui est read-only.

```c
int main(void)
{
  char *s = "hello";// emplacement d'adresse statique
  *s = 'H'; // ou s[0]='H';
}//seg fault
```

Le pointeur `s` pointe sur la première position de `"hello"`. En utilisant ce pointeur, on veut mettre un `'H'` dans cette zone mémoire protégée d'où le segmentation fault.

Eviter `Segmentation fault`

Au lieu d'utiliser une constante chaîne, laquelle est de classe d'allocation statique (rodata), on peut préférer utiliser un tableau de caractères. Alors la variable locale (à `main`) `s` est dans la pile et elle est donc modifiable.

```c
int main(){

  char s[] = "hello";// emplacement d'adresse dynamique
  s[0] = 'H';
  return 0;
}
```

Mémoire cache

- Dans le modèle de Von Neumann, il y constamment des échanges entre CPU et RAM, soit pour charger la prochaine instruction à exécuter, soit pour récupérer les données sur lesquelles agit l'instruction courante.

Mémoire cache

- Dans le modèle de Von Neumann, il y constamment des échanges entre CPU et RAM, soit pour charger la prochaine instruction à exécuter, soit pour récupérer les données sur lesquelles agit l'instruction courante.
- Cependant, comme les micro-processeurs sont beaucoup plus rapides que la mémoire, ils passeraient leur temps à attendre les données (on parle de *goulet d'étranglement*).

Mémoire cache

- Dans le modèle de Von Neumann, il y constamment des échanges entre CPU et RAM, soit pour charger la prochaine instruction à exécuter, soit pour récupérer les données sur lesquelles agit l'instruction courante.
- Cependant, comme les micro-processeurs sont beaucoup plus rapides que la mémoire, ils passeraient leur temps à attendre les données (on parle de *goulet d'étranglement*).
- C'est pour cela qu'on été introduites les *mémoires caches*. Pour gagner du temps, la mémoire cache est chargée avec les données provenant de la RAM lorsqu'une instruction en a besoin.

Mémoire cache

- Dans le modèle de Von Neumann, il y constamment des échanges entre CPU et RAM, soit pour charger la prochaine instruction à exécuter, soit pour récupérer les données sur lesquelles agit l'instruction courante.
- Cependant, comme les micro-processeurs sont beaucoup plus rapides que la mémoire, ils passeraient leur temps à attendre les données (on parle de *goulet d'étranglement*).
- C'est pour cela qu'on été introduites les *mémoires caches*. Pour gagner du temps, la mémoire cache est chargée avec les données provenant de la RAM lorsqu'une instruction en a besoin.
- L'idée, c'est que si on a eu besoin une fois d'une donnée, on risque d'en avoir besoin plusieurs fois. On gagne ainsi du temps sur les accès ultérieurs à cette donnée.

Cycle d'exécution d'une instruction

- Dans la mémoire sont stockées de la même façon un *opcode* (instructions en langage machine qui spécifie l'opération à traiter) et les données qu'elle manipule : sous forme d'octets.

Cycle d'exécution d'une instruction

- Dans la mémoire sont stockées de la même façon un *opcode* (instructions en langage machine qui spécifie l'opération à traiter) et les données qu'elle manipule : sous forme d'octets.
- Pour progresser dans l'exécution du programme, l'unité de contrôle de l'ordinateur réalise de manière continue un cycle appelé *Cycle d'exécution d'une instruction*.

Cycle d'exécution d'une instruction

- Dans la mémoire sont stockées de la même façon un *opcode* (instructions en langage machine qui spécifie l'opération à traiter) et les données qu'elle manipule : sous forme d'octets.
- Pour progresser dans l'exécution du programme, l'unité de contrôle de l'ordinateur réalise de manière continue un cycle appelé *Cycle d'exécution d'une instruction*.
- La durée de ce cycle est imposée par une *horloge globale*. L'horloge permet d'assurer que les données sont valides au cycle d'horloge suivant, c'est à dire que les calculs sont terminés et les résultats stabilisés.

Cycle d'exécution d'une instruction

Registres IP et IR

- Le registre IP (≪ Instruction Pointer ≫ ou ≪ compteur ordinal ≫) est un registre qui s'incrémente sans cesse. Il contient en permanence l'adresse de la prochaine instruction à réaliser.
  Faire un saut dans un programme, revient à inscrire dans le registre IP l'adresse de l'instruction ou le programme doit se rendre.
  L'incrémentation reprend alors à partir de cette nouvelle valeur.

Cycle d'exécution d'une instruction

Registres IP et IR

- Le registre IP (≪ Instruction Pointer ≫ ou ≪ compteur ordinal ≫) est un registre qui s'incrémente sans cesse. Il contient en permanence l'adresse de la prochaine instruction à réaliser.
  Faire un saut dans un programme, revient à inscrire dans le registre IP l'adresse de l'instruction ou le programme doit se rendre.
  L'incrémentation reprend alors à partir de cette nouvelle valeur.
- Le registre d'instruction (IR) contient l'instruction en cours d'exécution. Ce registre est chargé par l'Unité de Contrôle au début du cycle d'exécution avec l'opcode de l'instruction dont l'adresse est donnée par le registre IR.

Cycle d'exécution d'une instruction

Déroulement

1. Recherche puis Chargement. L'UC récupère le mot machine contenu à l'adresse mémoire indiqué par le registre IP et le met dans le registre IR.

Cycle d'exécution d'une instruction

Déroulement

1. Recherche puis Chargement. L'UC récupère le mot machine contenu à l'adresse mémoire indiqué par le registre IP et le met dans le registre IR.
2. Décodage. La suite de bits du mot contenu dans le registre IR est décodée pour connaître :

   Cette étape, complexe, peut donc nécessiter de lire d'autres mots machines.

Cycle d'exécution d'une instruction

Déroulement

1. Recherche puis Chargement. L'UC récupère le mot machine contenu à l'adresse mémoire indiqué par le registre IP et le met dans le registre IR.
2. Décodage. La suite de bits du mot contenu dans le registre IR est décodée pour connaître :
    - l'opération à exécuter,

   Cette étape, complexe, peut donc nécessiter de lire d'autres mots machines.

Cycle d'exécution d'une instruction

Déroulement

1. Recherche puis Chargement. L'UC récupère le mot machine contenu à l'adresse mémoire indiqué par le registre IP et le met dans le registre IR.
2. Décodage. La suite de bits du mot contenu dans le registre IR est décodée pour connaître :
    - l'opération à exécuter,
    - les données (*opérandes*) sur lesquelles elle agit (elles peuvent être en RAM, dans le cache, ou dans des registres).

   Cette étape, complexe, peut donc nécessiter de lire d'autres mots machines.

Cycle d'exécution d'une instruction

Déroulement

1. Recherche puis Chargement. L'UC récupère le mot machine contenu à l'adresse mémoire indiqué par le registre IP et le met dans le registre IR.
2. Décodage. La suite de bits du mot contenu dans le registre IR est décodée pour connaître :
   - l'opération à exécuter,
   - les données (*opérandes*) sur lesquelles elle agit (elles peuvent être en RAM, dans le cache, ou dans des registres).

   Cette étape, complexe, peut donc nécessiter de lire d'autres mots machines.
3. Exécution. Elle peut être réalisée soit par l'UAL (cas d'une opération arithmétique ou logique) soit par l'Unité de Contrôle (cas d'une opération de branchement). Dans ce dernier cas, le registre IP est modifié et non plus seulement incrémenté.

Cycle d'exécution d'une instruction

Déroulement

1. Recherche puis Chargement. L'UC récupère le mot machine contenu à l'adresse mémoire indiqué par le registre IP et le met dans le registre IR.
2. Décodage. La suite de bits du mot contenu dans le registre IR est décodée pour connaître :
    - l'opération à exécuter,
    - les données (*opérandes*) sur lesquelles elle agit (elles peuvent être en RAM, dans le cache, ou dans des registres).

    Cette étape, complexe, peut donc nécessiter de lire d'autres mots machines.
3. Exécution. Elle peut être réalisée soit par l'UAL (cas d'une opération arithmétique ou logique) soit par l'Unité de Contrôle (cas d'une opération de branchement). Dans ce dernier cas, le registre IP est modifié et non plus seulement incrémenté.
4. Ecriture du résultat. Puis recommencer.