

Architecture des ordinateurs

Cours 6

Responsable de l'UE : Karine Heydemann

Contact : prenom.nom@lip6.fr

Plan du cours 6

- 1 Introduction
- 2 Données de type tableau ou enregistrement
- 3 Variables locales et pile d'exécution
- 4 Optimisation des accès mémoires et des variables locales
- 5 Tableaux et enregistrements en variables locales

Variables d'un programme C

Représentation et implantation

- Une variable correspond à un emplacement mémoire
- La taille de cet emplacement dépend du type de la variable
- Deux grandes familles : types élémentaires et types structurés
- Les types élémentaires correspondent aux types C : `int`, `unsigned int`, `short`, `unsigned short`, `char`, `unsigned char`, `char*`, `int*`, `void*`,...
- Le contenu de l'emplacement contient le codage de la valeur de la variable : c'est un mot binaire, c'est son utilisation dans le code qui lui donne une interprétation.
- Les types structurés correspondent à des variables comportant plusieurs données élémentaires :
 - ⇒ l'objet de ce cours est de présenter l'implantation mémoire des tableaux et des enregistrements.
- Les variables globales sont implantées dans le segment de données (`.data` avec des directives dédiées). Quid des variables locales ?
 - ⇒ Ce cours présente l'allocation des variables locales et la notion de pile d'exécution.

Structure de données de type tableau

Tableau

- Ensemble fini d'éléments d'un même type
- Ensemble ordonné : un indice donne la position d'un élément dans le tableau

Accès à un élément

- Opération d'indexation $T[i]$
- $T[i]$ désigne l'élément d'indice i , c'est le $i+1$ ième.

Exemples

- `int tab[5] = {1,2,3,4,5};`
- `char chaine1[5]={ 'a', 'r', 'c', 'h', 'i' };`
- `char chaine2[] = "ARCHI";`

Implantation de données de type tableau

Implantation mémoire

- Un tableau est alloué en mémoire de façon contiguë : les éléments sont rangés les uns à la suite des autres

Zone mémoire allouée à un tableau

- `adresse_debut` : l'adresse de base du tableau, c'est l'adresse du premier élément (indice 0)
- `taille_elem` : taille (en octets) d'un élément
- `nb_elem` : nombre d'éléments du tableau
- la taille de la mémoire allouée est $\text{nb_elem} * \text{taille_elem}$

Adresse d'un élément du tableau

- L'adresse `&T[i]` de l'élément `i` est $\text{adresse_debut} + i * \text{taille_elem}$.
Equivalent à $\text{\&T[0]} + i * \text{taille_elem}$.

Implantation de données de type tableau

Accès à un élément

- L'adresse de l'élément i ($\&T[i]$) est `adresse_debut + i*taille_elem`.
Equivalent à `&T[0]+i*taille_elem`.
- Accéder à l'élément $T[i]$ c'est déréférencer la case d'adresse `&T[i]`
- Correspond toujours un accès mémoire :
 - `a = ...T[i]...`; requiert un accès en lecture à l'élément $T[i]$, donc à l'adresse `adresse_debut + i*taille_elem`
 - `T[i] = ...;` requiert un accès en écriture à l'élément $T[i]$, donc à l'adresse `adresse_debut + i*taille_elem`

Exemples de données globales de type tableau

- Soit le programme C :

```
/* variables globales */  
...  
int tab[5] = {5, 17, 23, 41, 123};  
char chaine[] = "mips";
```

- Directives de déclaration des tableaux

```
tab: .word 5, 17, 23, 41, 123  
chaine: .asciiz "mips"
```

- Hypothèse : `tab` est implanté à l'adresse `0x1001000c`
- L'adresse de `chaine` est $0x1001000c + 5 \times 4 = 0x1001000c + 0x14 = 0x10010020$
- Remarque : l'adresse de `tab` est nécessairement un multiple de 4

Exemples de données globales de type tableau

- Soit le programme C :

```
/* variables globales */  
...  
int tab[5] = {5, 17, 23, 41, 123};  
char chaine[] = "mips";
```

- Directives de déclaration des tableaux

```
tab: .word 5, 17, 23, 41, 123  
chaine: .asciiz "mips"
```

- tab est implanté à l'adresse 0x1001000c et chaine à 0x10010020
- Représentation de la mémoire avec vue par mot

adresse (mot)	0x1001000c	0x10010010	0x10010014	0x10010018
contenu (hexa)	0x00000005	0x00000011	0x00000017	0x00000029
contenu (données)	5	17	23	41
adresse (mot)	0x1001001c	0x10010020	0x10010024	0x10010028
contenu (hexa)	0x0000007b	0x7370696d	0x00000000	0x00000000
contenu (données)	123	's','p','i','m'	?, ?, ?, '\0'	?, ?, ?, ?

Exemples de données globales de type tableau

- Directives de déclaration des tableaux

```
tab: .word 5, 17, 23, 41, 123
chaine: .asciiz "mips"
```

- tab est implanté à l'adresse 0x1001000c et chaine à 0x10010020
- Représentation de la mémoire avec vue par mot

adresse (mot)	0x1001000c	0x10010010	0x10010014	0x10010018
contenu (hexa)	0x00000005	0x00000011	0x00000017	0x00000029
contenu (données)	5	17	23	41
adresse (mot)	0x1001001c	0x10010020	0x10010024	0x10010028
contenu (hexa)	0x0000007b	0x7370696d	0x00000000	0x00000000
contenu (données)	123	's','p','i','m'	?, ?, ?, '\0'	?, ?, ?, ?

- $\&\text{tab}[2] = \&\text{tab}[0] + 2 \times 4 = 0\text{x}1001000\text{c} + 8 = 0\text{x}10010014$
- $\&\text{ch}[3] = \&\text{ch}[0] + 3 \times 1 = 0\text{x}10010020 + 3 = 0\text{x}10010023$
- $\text{tab}[2] = \text{contenu de l'élément d'indice 2} = \text{mot à l'adresse } 0\text{x}10010014 = 23$
- $\text{ch}[3] = \text{contenu de l'élément d'indice 3} = \text{octet à l'adresse } 0\text{x}10010023 = \text{'s'}$

Structure de données de type enregistrement

- Données comportant plusieurs champs de type potentiellement différent :

```
struct _ms_type {int id; char t1[2]; int t2[2]};
```

- Accès aux champs par notation pointée / déréférencement si pointeur vers un enregistrement :

```
struct _ms_type {int id; char t1[2]; int t2[2]};  
struct _ms_type ms1 = {12, {1,2}, {11,12}};  
struct _ms_type ms2;  
struct _ms_type *ms_ref;
```

```
void main() {  
    ...  
    ms_ref = &ms2;           // adresse de ms2 dans ms_ref  
    ms_ref->id = ms1.id + 1;  
    ms_ref->t1[1] = ms1.t1[1] + 2;  
    ...  
}
```

Implantation de données de type enregistrement

- Les données d'un enregistrement sont allouées en mémoire les unes à la suite des autres
- Respect des contraintes d'alignement de chaque champ en fonction de sa taille en octets

```
struct _ms_type {int id; char t1[2]; int t2[2];};  
struct _ms_type ms1 = {12, {1,2}, {11,12}};  
struct _ms_type ms2;  
struct _ms_type *ms_ref;
```

⇒ Une donnée de type `struct ms_type` requiert 14 octets de données + 2 octets nécessaires pour respecter la contrainte d'alignement du champ `t2` : 16 octets

```
.data  
ms1:      .word 12      # ms1.id  
          .byte 1, 2    # ms1.t1  
          .word 11, 12  # ms1.t2  
          .align 2  
ms2:      .space 16     # 16 octets  
          .align 2  
ms_ref:   .space 4      # pointeur = adresse = 4 octets
```

Adresse d'implantation d'enregistrements

- L'adresse du 1er champ est l'adresse de la structure
- L'adresse du champ i ($i > 1$) d'une structure est égale à l'adresse du champ $i-1$ + taille du champ $i-1$ + contrainte d'alignement du champ i
- Exemple de calcul des adresses d'implantation

```
.data
ms1:      .word 12      # @ms1.id = 0x10010000
          .byte 1, 2    # @ms1.t1 = 0x10010004
          .word 11, 12  # @ms1.t2 = 0x10010004 + 2 + align=2
                      #          = 0x10010008

ms2:      .align 2
          .space 16     # @ms2      = 0x10010008 + 4*2
                      #          = 0x10010010
                      #          = @ms2.id
                      # @ms2.t1 = @ms2.id + 4
                      #          = 0x10010014
                      # @ms2.t2 = @ms2.t1 + 2 + align=2
                      #          = 0x10010018

ms_ref:   .align 2
          .space 4      # @ms_ref = 0x10010010 + 16
                      #          = 0x10010020
```

Exemple de code avec tableau et enregistrement

Code C

```
struct _ms_type {
    int id;
    char t1[2];
    int t2[2];
};
...
void main() {
    ...
    /*adresse de ms2 dans
      ms_ref*/
    ms_ref = &ms2;
    ms_ref->id =
        ms1.id + 1;
    ms_ref->t1[1] =
        ms1.t1[1] + 2;
    ...
}
```

Code ASM

```
.data
ms1:    .word 12      # @ms1.id = 0x10010000
        .byte 1, 2   # @ms1.t1 = 0x10010004
        .word 11, 12 # @ms1.t2 = 0x10010008
ms2:    .align 2
        .space 16    # @ms2 = 0x10010010
ms_ref: .align 2
        .space 4     # @ms_ref = 0x10010020

.text
lui $8, 0x1001
ori $9, $8, 16 # $8 = &ms2
sw $9, 32($8)  # écriture ms_ref
lw $9, 0($8)   # lecture ms1.id
addiu $9, $9, 1 # ms1.id + 1
lw $10, 32($8) # lecture ms_ref
sw $9, 0($10)  # ecriture ms_ref->id
lb $9, 5($8)   # lecture ms1.t1[1]
addiu $9, $9, 2 # ms1.tab[1] + 2
lw $10, 32($8) # lecture ms_ref
sb $9, 5($10)  # ecriture ms_ref->t1[1]
...

```

Variables locales

- Variable = emplacement mémoire
- Variable locale = à portée limitée, existe uniquement dans la fonction ou le bloc contenant sa déclaration
- A l'exécution, plusieurs instances d'une même variable possible : c'est le cas lors de l'exécution de fonctions récursives
- Chaque exécution d'une fonction a besoin d'emplacements mémoire dédiés pour ses variables locales (entre autres) : ces emplacements font partie du **contexte d'exécution** de la fonction

Segments mémoires d'implantation

- Les variables globales sont visibles, donc accessibles par toutes les fonctions : allocation statique dans le segment de données (via des directives dans `.data`), leur initialisation est réalisée lors du chargement mémoire du code binaire (par le loader)
- Les variables locales sont visibles par une seule instance de fonction (ou bloc), accessibles que par cette fonction : leur allocation est **dynamique**, réalisée tout comme leur initialisation par la fonction à l'exécution, et ce dans un autre segment mémoire appelé **segment de pile**.

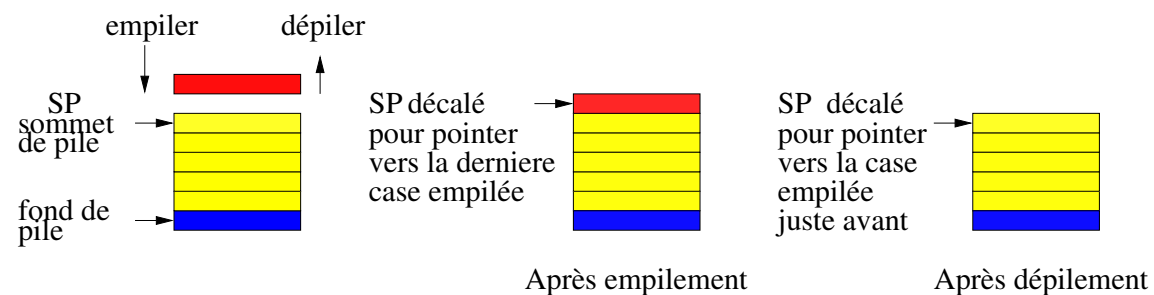
Imbrication des appels de fonction et contextes

- En général, un programme principal (qui est une fonction) appelle des fonctions qui elles-même peuvent appeler des fonctions qui etc.
- Ordre de création/destruction et durée de vie des contextes : quand une fonction f appelle une fonction g , le contexte de g est créé et co-existe avec celui de f jusqu'à la fin de l'exécution de g . Le contexte courant est celui de la fonction qui s'exécute : celui de f puis de g puis de nouveau f .
- Profondeur d'appel à un moment donné = nombre d'appels imbriqués
- Le nombre maximal de contextes co-existant correspond à la profondeur d'appel maximale possible : ce n'est pas déterminable statiquement dans le cas général (cas des fonctions récursives).
- Besoin d'un mécanisme d'allocation mémoire de contextes
 - 1 dynamique : co-existence d'un nombre a priori quelconque de contextes
 - 2 mémorisant l'imbrication des appels de fonction en cours : détermination aisée du contexte courant au retour d'une fonction
- La structure de pile est adaptée pour cela

Notion de pile

Pile

- Structure de données abstraite permettant de stocker/récupérer des informations
- Gestion LIFO (Last In First Out) : la structure de pile conserve l'ordre d'allocation
- Opérations sur la pile :
 - Empiler : allouer un emplacement au sommet de pile + le remplir
 - Dépiler : récupérer le contenu du sommet de pile si besoin + désallouer la mémoire en sommet de pile
 - Tester si la pile est vide ou pleine



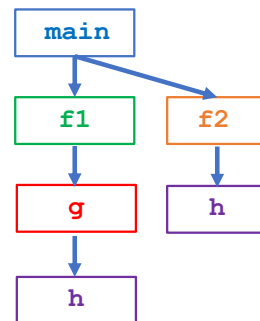
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Grappe d'appels

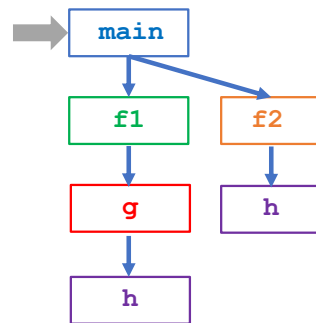


Contexte de fonctions et structure de pile

→

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



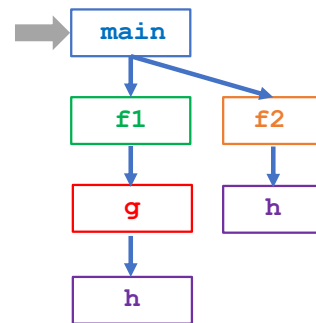
A l'exécution

Contexte
main

Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Grappe d'appels



A l'exécution

Contexte
main

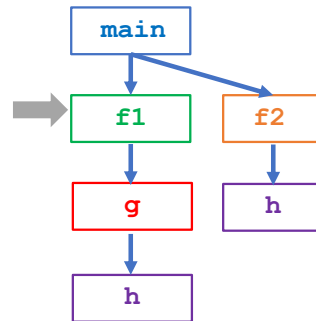
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}
```

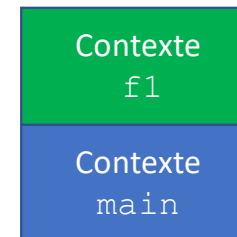
→

```
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



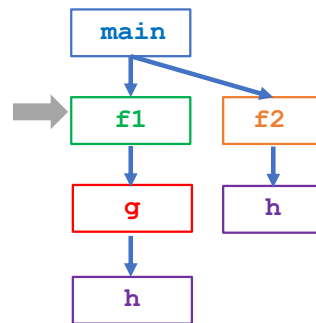
A l'exécution



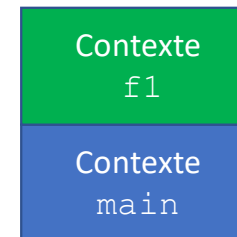
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



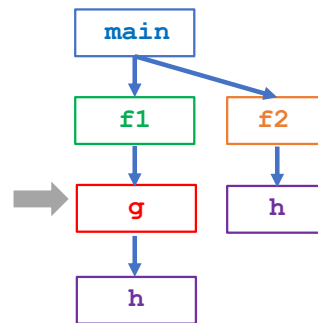
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
→ void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



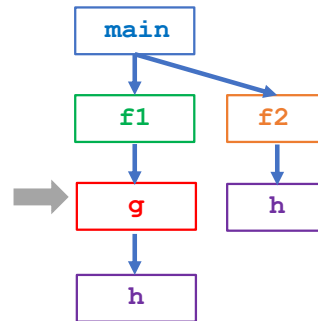
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



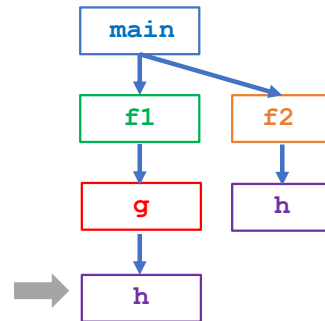
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



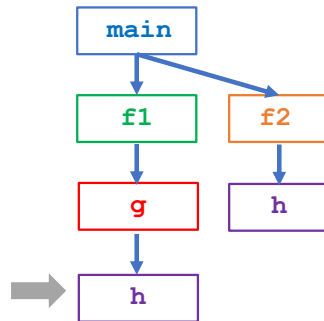
A l'exécution



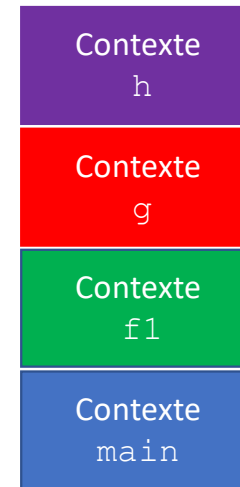
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



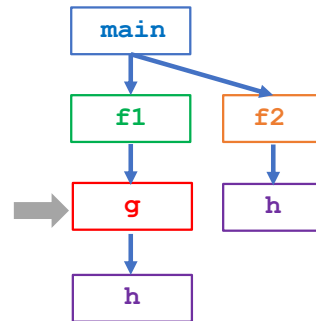
A l'exécution



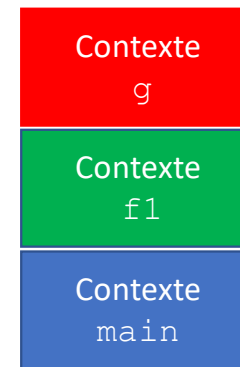
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



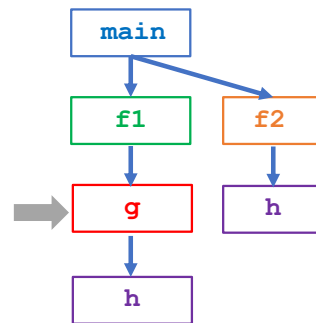
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



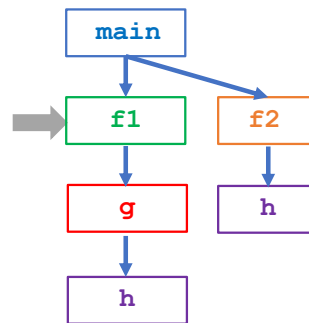
A l'exécution



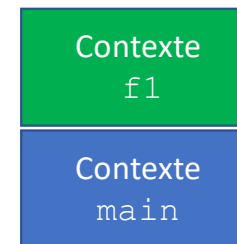
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



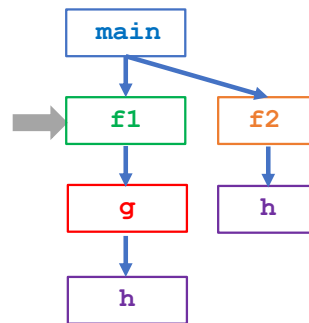
A l'exécution



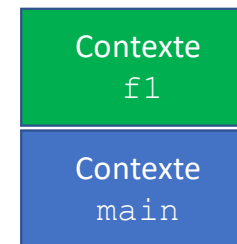
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



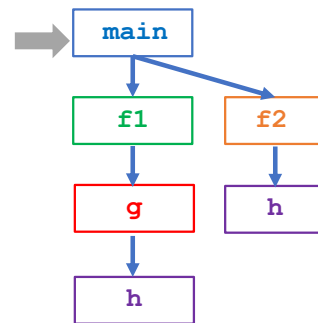
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Grappe d'appels



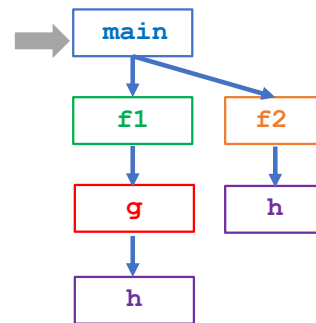
A l'exécution

Contexte
main

Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphes d'appels



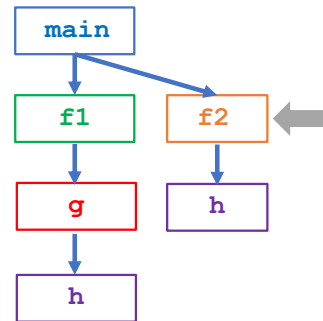
A l'exécution

Contexte
main

Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



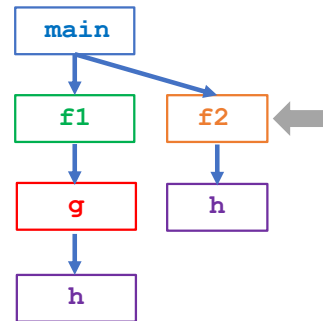
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



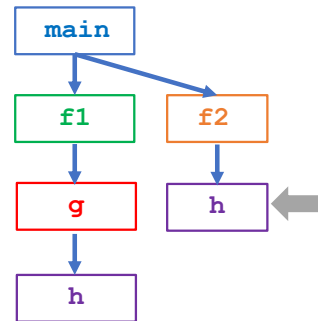
A l'exécution



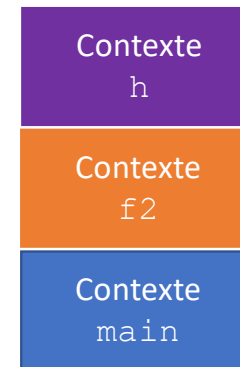
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



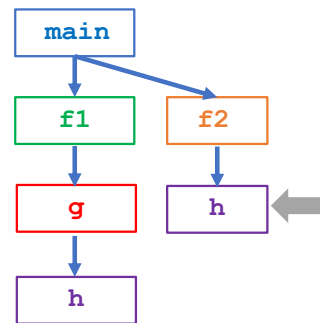
A l'exécution



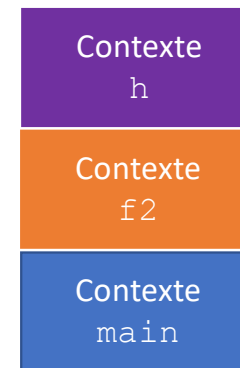
Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



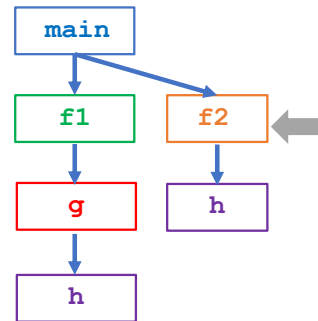
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



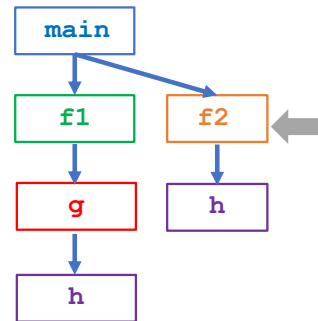
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Graphe d'appels



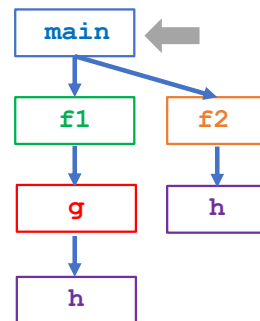
A l'exécution



Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Grappe d'appels



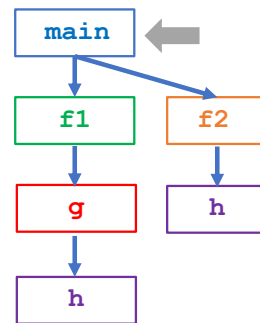
A l'exécution

Contexte
main

Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

Grappe d'appels



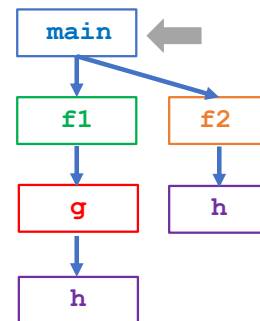
A l'exécution

Contexte
main

Contexte de fonctions et structure de pile

```
void main() {  
    ...  
    f1();  
    ...  
    f2();  
    ...  
    exit();  
}  
void f1() {  
    ...  
    g();  
    return;  
}  
void g() {  
    ...  
    h();  
    return;  
}  
void h() {  
    ...  
    return;  
}  
void f2() {  
    ...  
    h();  
    return;  
}
```

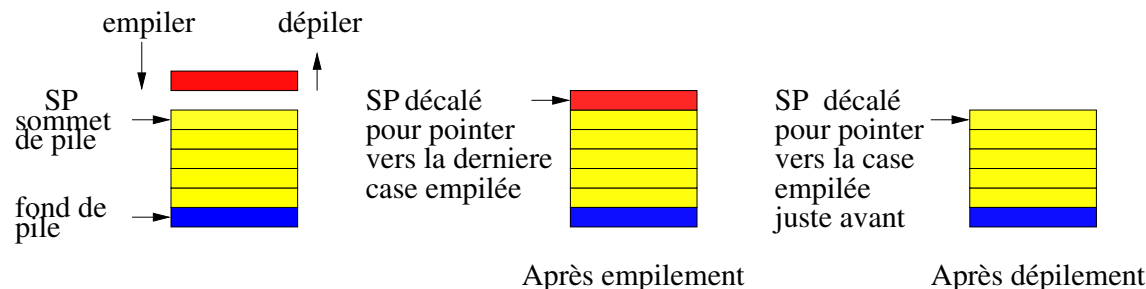
Grappe d'appels



A l'exécution

Segment de pile

- Le segment de pile est un segment en mémoire : ensemble d'adresses contiguës accessibles par le processeur
 - On y accède comme au segment de données, avec des instructions de transfert mémoire et une adresse de base : celle du sommet de pile contenue dans un registre dédié (\$29 en MIPS).
 - Le segment de pile est utilisé **comme une pile** en réalisant les opérations “empiler” et “dépiler” dynamiquement avec des instructions du processeur
 - Allouer de la mémoire au sommet de pile \equiv modifier l'adresse du sommet de pile
 - Désallouer de la mémoire au sommet de pile \equiv modifier l'adresse du sommet de pile
 - Accès au contenu la pile : lecture ou écriture de données dans la pile \equiv `load` ou `store`.
- Attention** : pas le droit d'accéder à un emplacement non alloué !



Implantation de la pile en Mips

Organisation de la pile

- De manière générale, dépend de la machine cible
- En Mips, le registre \$29 contient l'adresse du sommet de la pile, et le sommet de la pile est la dernière case occupée (et non la première libre)
- Ce registre est appelé le “pointeur de pile” (SP, Stack Pointer)
- En Mips, la pile grandit vers les adresses décroissantes
 - Allouer dans la pile revient à **décrémenter \$29**
 - Désallouer dans la pile revient à **incrémenter \$29**
- \$29 doit toujours contenir l'adresse d'un mot du segment de pile (soit une adresse multiple de 4)
 - Au chargement il contient l'adresse du fond de pile (0x7FFFFFFC dans la convention Mips)

Allocation et désallocation d'un contexte

Prologue d'une fonction et du programme principal

- Premières instructions exécutées par la fonction / le programme principal pour allouer les emplacements mémoire du contexte et réaliser les opérations nécessaires à son remplissage
- Allocation \equiv décrémentation de \$29 de x octets, avec x satisfaisant 2 contraintes
 - 1ère contrainte : $x \geq$ place nécessaire pour les emplacements de toutes les variables locales en respectant les contraintes d'alignement imposées par leur taille (cf. allocation variables globales).
 - 2ème contrainte : alignement du pointeur de pile, x est un multiple de 4
- Initialisation des variables locales par des écritures mémoire à leur emplacement sur la pile
 - Dans ce cours : l'emplacement de la variable déclarée en premier se trouve au sommet de la pile

Exemple (incomplet) avec variables locales

Code C

```
void main() {  
    int a = 5;  
    int b = 9;  
    int c;  
    c = a + b;  
    exit();  
}
```

Code ASM

```
.text  
#prologue du main  
addiu $29, $29, -12 # allocation de 3 mots  
# initialisation des variables  
ori $8, $0, 5 # $8 <- 5  
sw $8, 0($29) # a <- 5  
ori $8, $0, 9 # $8 <- 9  
sw $8, 4($29) # b <- 9  
  
# corps du main  
  
# exit  
ori $2, $0, 10  
syscall
```

- 3 variables locales de type `int` : 3 mots alloués sur la pile dans le prologue
- Première variable déclarée au sommet de pile : `a` est à l'adresse `$29`, `b` à l'adresse `($29 + 4)`, `c` à l'adresse `($29 + 8)`
- Initialisation de `a` et `b`

Désallocation d'un contexte

Epilogue d'une fonction et du programme principal

- Dernières instructions exécutées par la fonction / programme principal avant de retourner à la fonction appelante / finir le programme
- Récupération quand nécessaire d'informations mises dans le contexte : rien à faire pour les variables locales
- Désallocation du contexte : incrémentation de \$29 de x (même quantité que l'allocation dans le prologue)

Exemple (incomplet) avec variables locales

Code C

```
void main() {  
    int a = 5;  
    int b = 9;  
    int c;  
    c = a + b;  
    exit();  
}
```

Code ASM

```
.text  
#prologue du main  
addiu $29, $29, -12 # allocation de 3 mots  
# initialisation des variables  
ori  $8, $0, 5  # $8 <- 5  
sw   $8, 0($29) # a <- 5  
ori  $8, $0, 9  # $8 <- 9  
sw   $8, 4($29) # b <- 9  
  
# corps du main  
  
# epilogue du main  
addiu $29, $29, 12 # désallocation des 3 mots  
# exit  
ori  $2, $0, 10  
syscall
```

- 3 variables locales de type `int` : 3 mots alloués sur la pile dans le prologue, 3 mots désalloués dans l'épilogue

Contexte d'exécution d'une fonction

Contexte d'exécution

- Zone mémoire contiguë dans la pile contenant différentes informations relatives à une exécution d'une fonction
- Notamment des emplacements mémoires pour les variables locales de la fonction
- Mais aussi des emplacements pour
 - la sauvegarde des registres persistants (voir prochain cours)
 - les paramètres à passer aux fonctions appelées par la fonction (voir prochain cours)
 - la sauvegarde temporaire de calculs intermédiaires lorsque l'on manque de registre pour faire les calculs (dépend du nombre de registres généraux et des calculs réalisés par la fonction – on ne rencontrera pas ce cas dans ce cours)

Traduction assembleur des accès aux variables d'un programme C

Traduction littérale

- Chaque lecture d'une variable `v`, d'un élément de tableau `tab[i]`, ou champ de structure `ms.x` correspond à un `load` à l'emplacement mémoire de `v`, de `tab[i]`^a ou de `ms.x`
- Chaque écriture d'une variable `v`, d'un élément de tableau `tab[i]`, ou champ de structure `ms.x` correspond à un `store` en mémoire à l'emplacement correspondant
- Vrai pour les variables globales ET les variables locales
- Une variable n'est **pas** associée à un registre : il peut y avoir beaucoup de variables (des centaines) alors qu'il n'y aura toujours que 32 registres
 - Les registres ne sont utilisés que de façon très temporaire
 - Différents `load` de la même variable peuvent avoir des registres de destination différents
 - Un même registre peut contenir les valeurs de variables différentes au cours de l'exécution d'une fonction

a. NB : `i` est une variable locale, il faut lire sa valeur en mémoire pour calculer l'adresse de `tab[i]` !

Exemple complet avec variables locales

Code C

```
void main() {  
    int a = 5;  
    int b = 9;  
    int c;  
    c = a + b;  
    exit();  
}
```

Code ASM

```
.text  
#prologue du main  
addiu $29, $29, -12 # allocation de 3 mots  
# initialisation des variables  
ori $8, $0, 5 # $8 <- 5  
sw $8, 0($29) # a <- 5  
ori $8, $0, 9 # $8 <- 9  
sw $8, 4($29) # b <- 9  
# corps du main  
lw $8, 0($29) # $8 <- a (lecture de a)  
lw $9, 4($29) # $9 <- b (lecture de b)  
addu $8, $8, $9 # $8 <- 14 (a + b)  
sw $8, 8($29) # c <- 14 (écriture de c)  
# epilogue du main  
addiu $29, $29, 12 # désallocation des 3 mots  
# exit  
ori $2, $0, 10  
syscall
```

- Lecture de `a` et `b` sur la pile (adresses `$29` et `($29+4)`), écriture de la valeur de `c` sur la pile (adresse `($29+8)`)

Simulation de l'exécution de l'exemple

```
.data

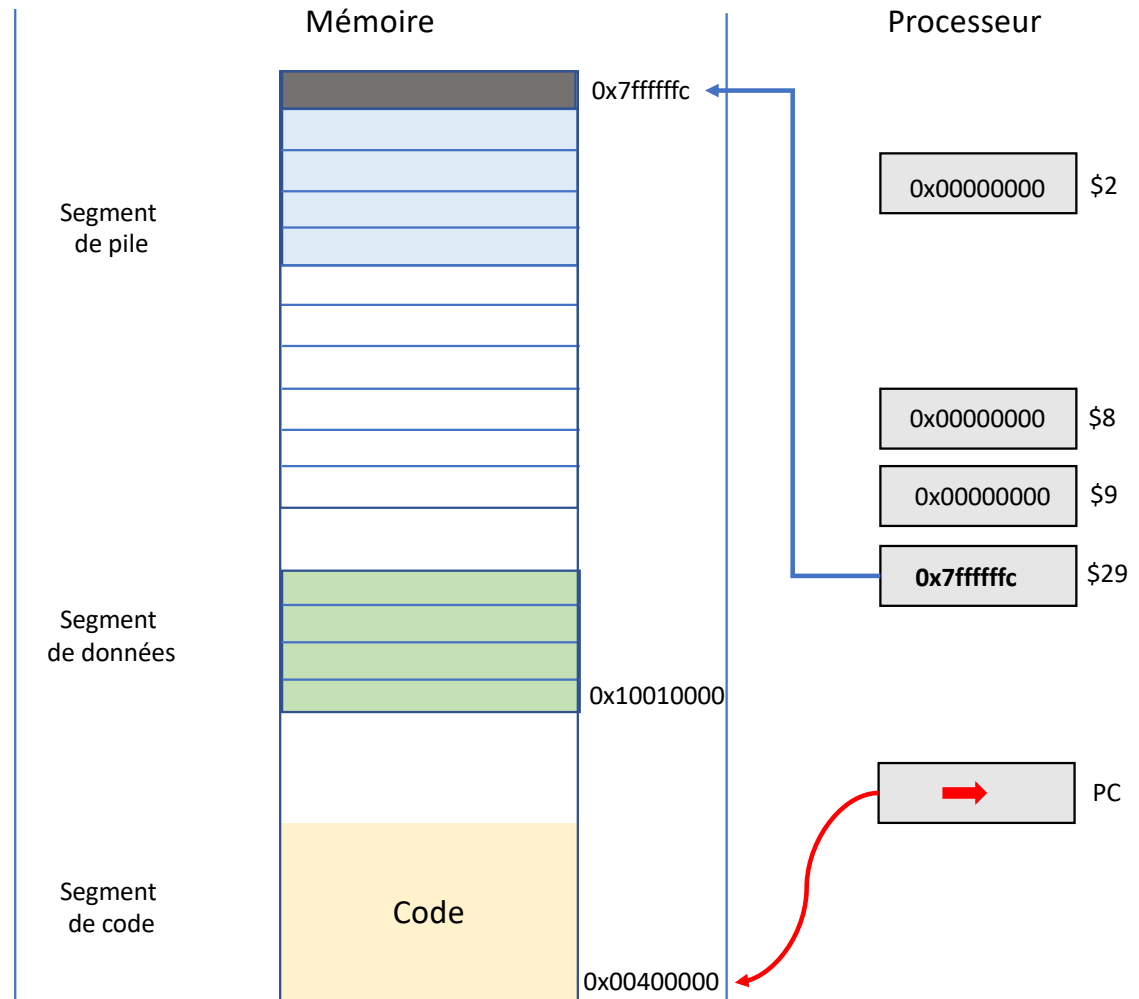
.text
#prologue du main
➔ addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori  $8, $0, 5    # $8 <- 5
sw   $8, 0($29)   # a <- 5
ori  $8, $0, 9    # $8 <- 9
sw   $8, 4($29)   # b <- 9

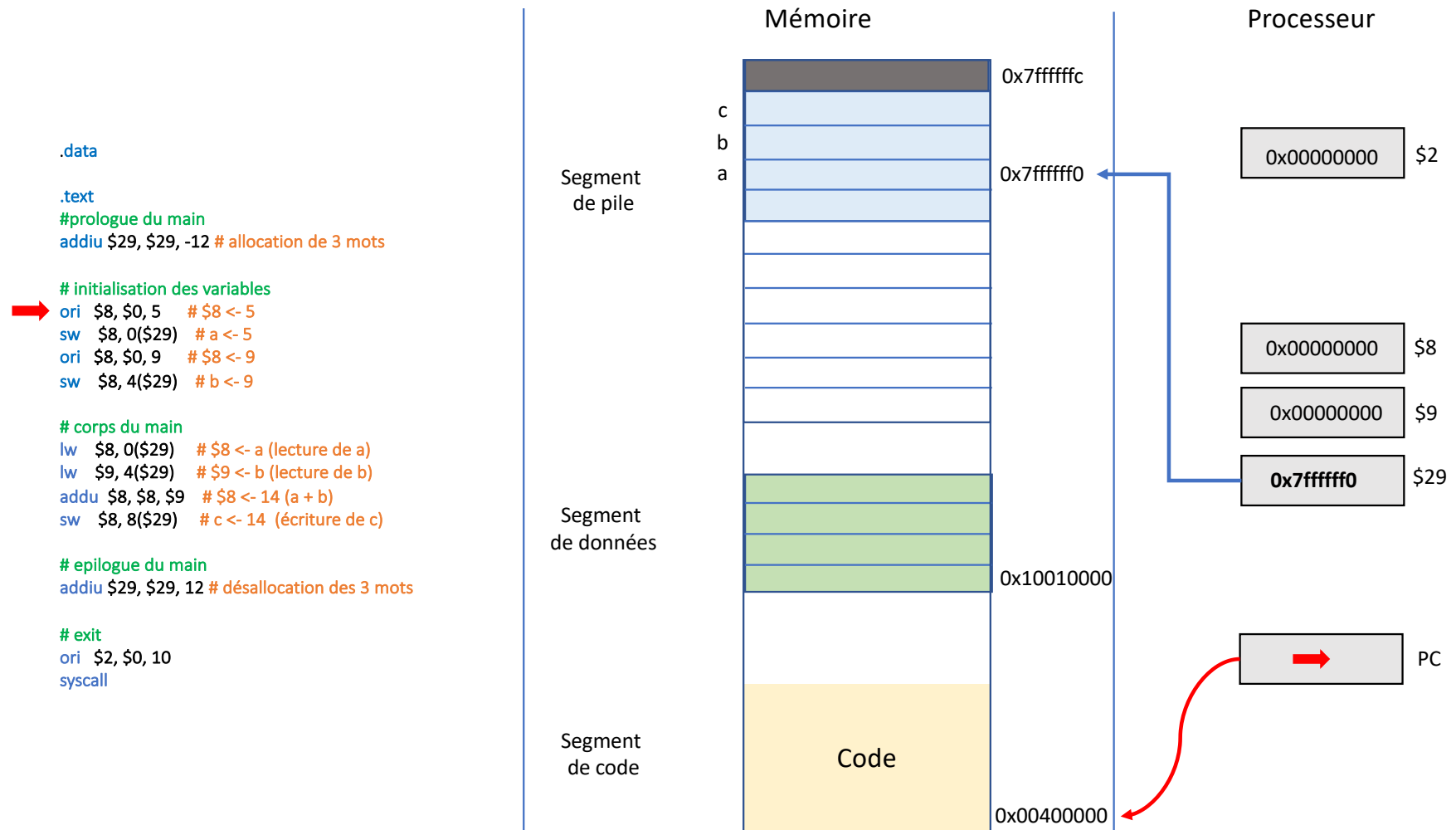
# corps du main
lw   $8, 0($29)   # $8 <- a (lecture de a)
lw   $9, 4($29)   # $9 <- b (lecture de b)
addu $8, $8, $9   # $8 <- 14 (a + b)
sw   $8, 8($29)   # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

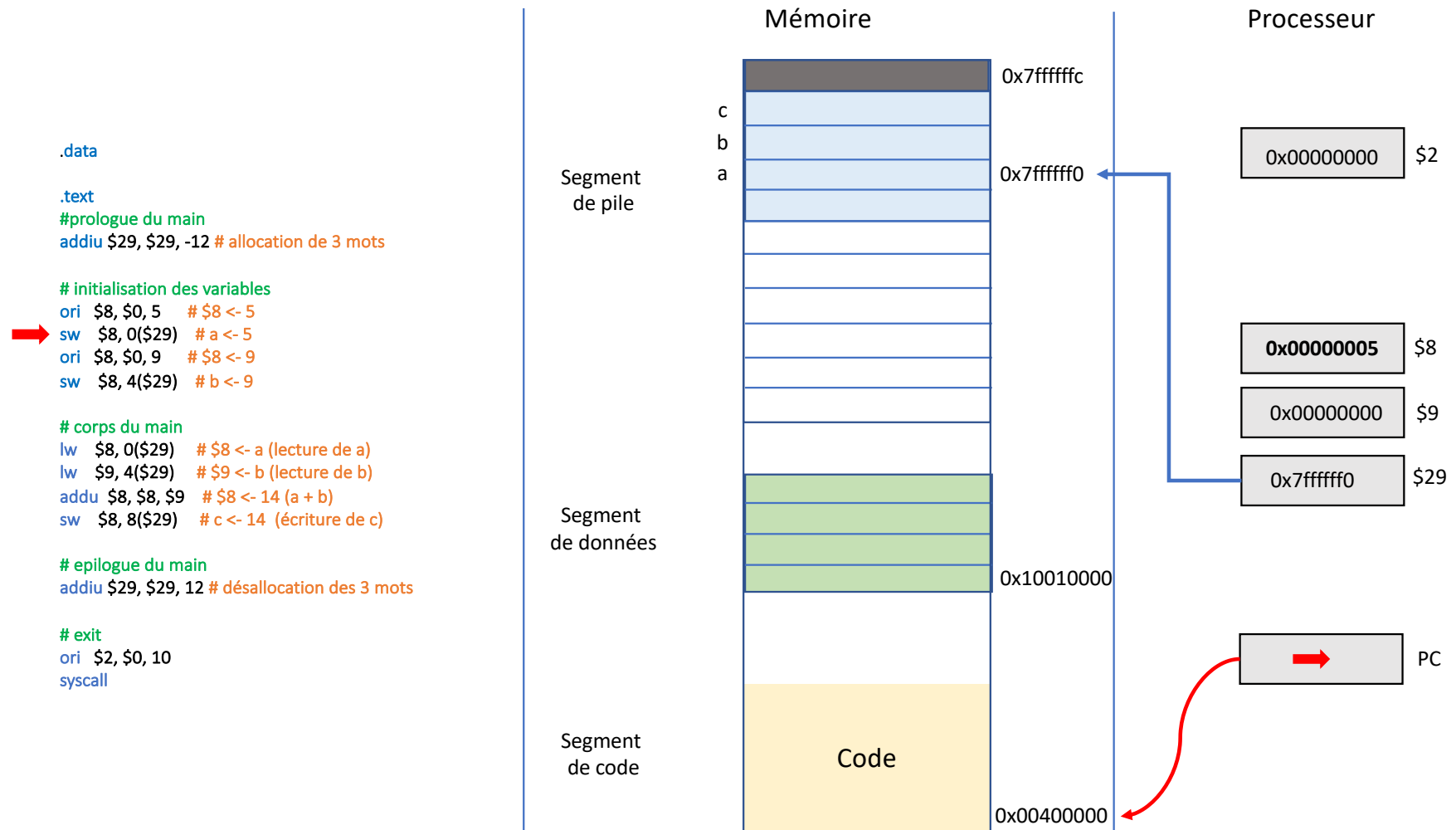
# exit
ori  $2, $0, 10
syscall
```



Simulation de l'exécution de l'exemple



Simulation de l'exécution de l'exemple



Simulation de l'exécution de l'exemple

```

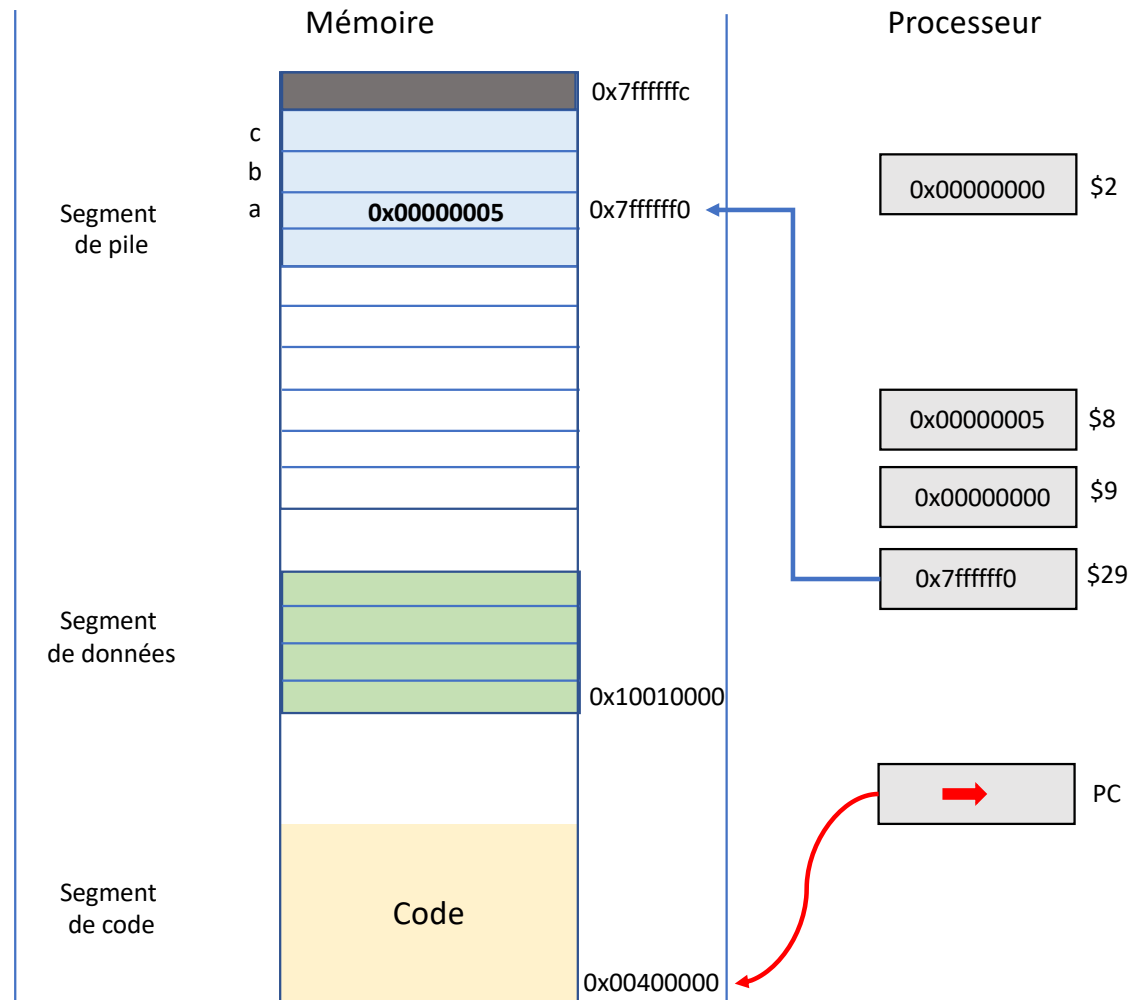
.data
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
→ ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple

```

.data

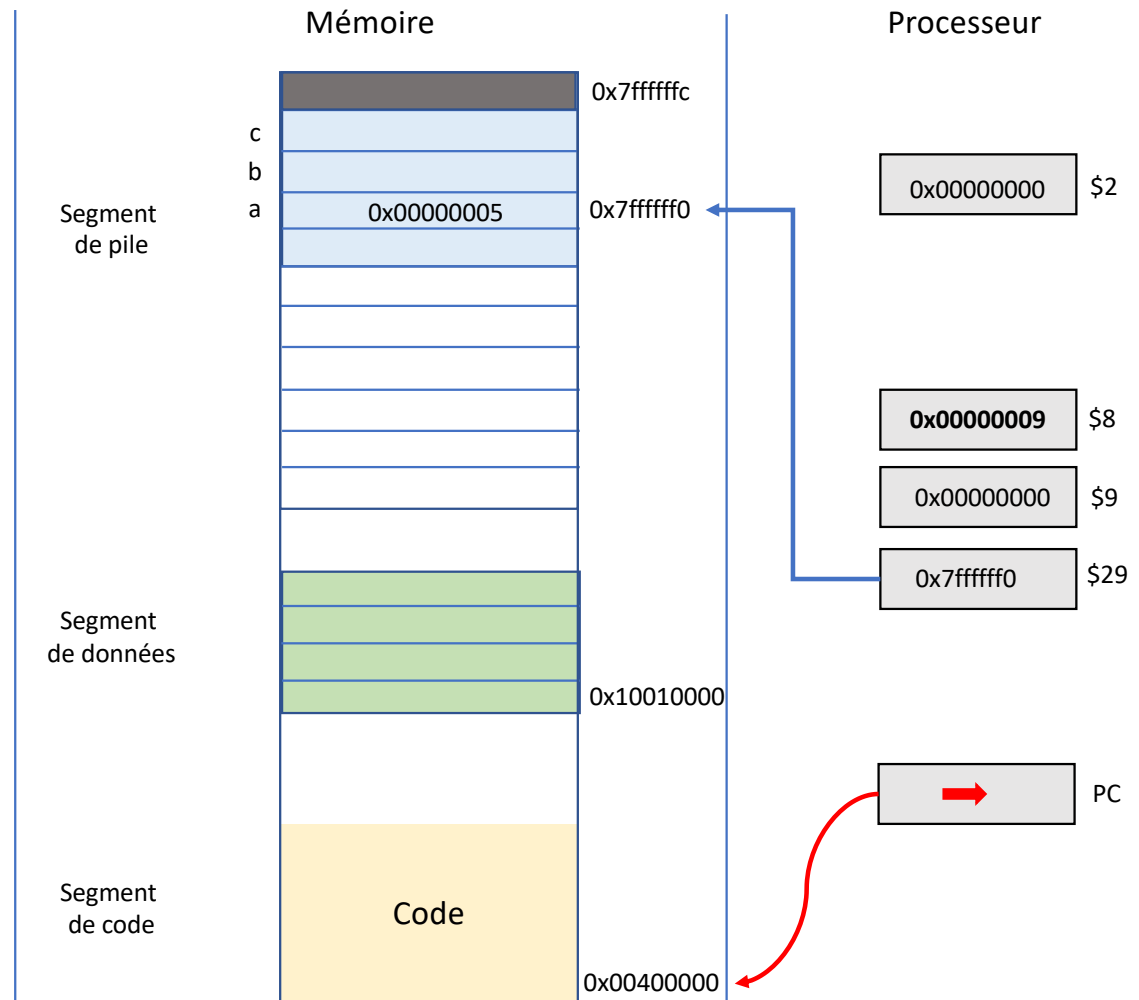
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
→ sw $8, 4($29) # b <- 9

# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple

```

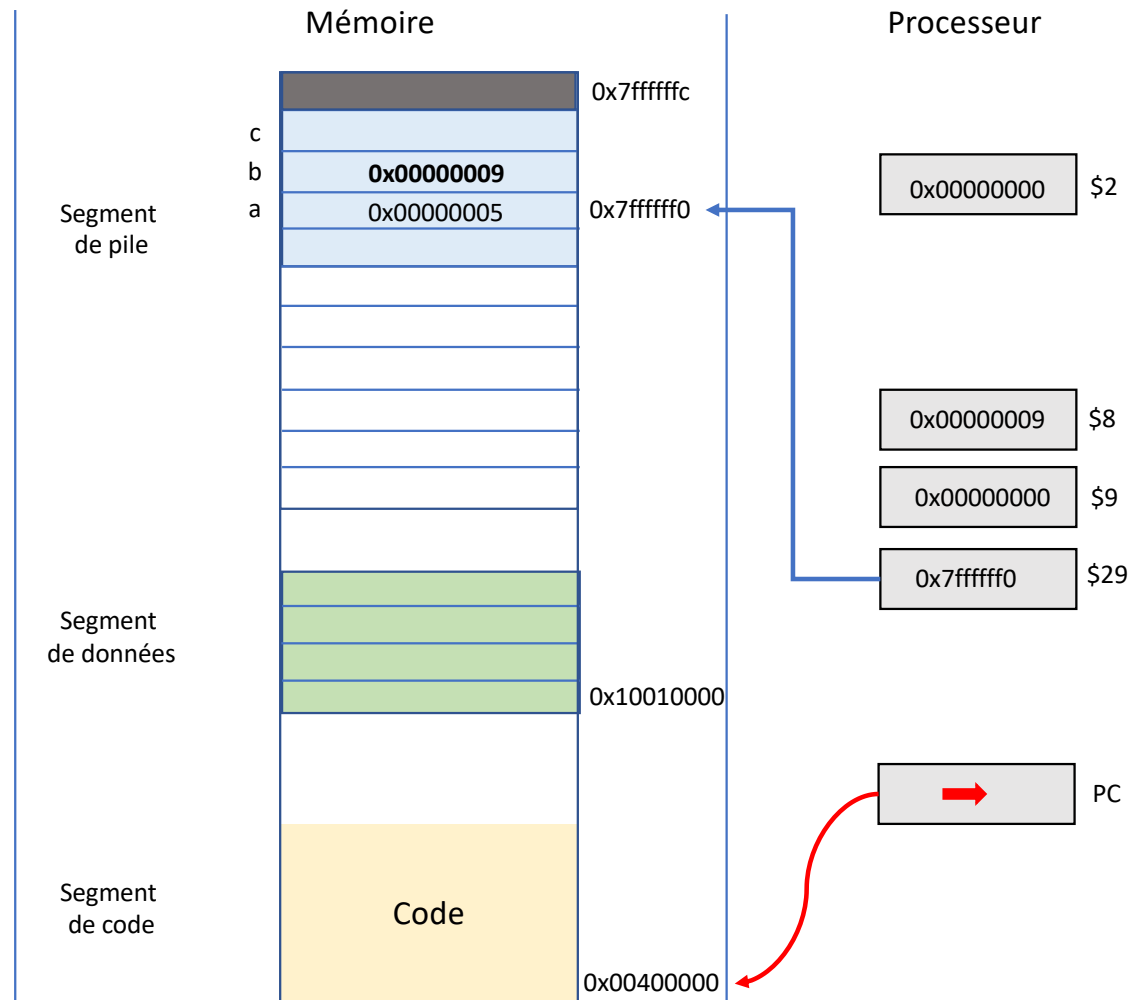
.data
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

# corps du main
→ lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple

```

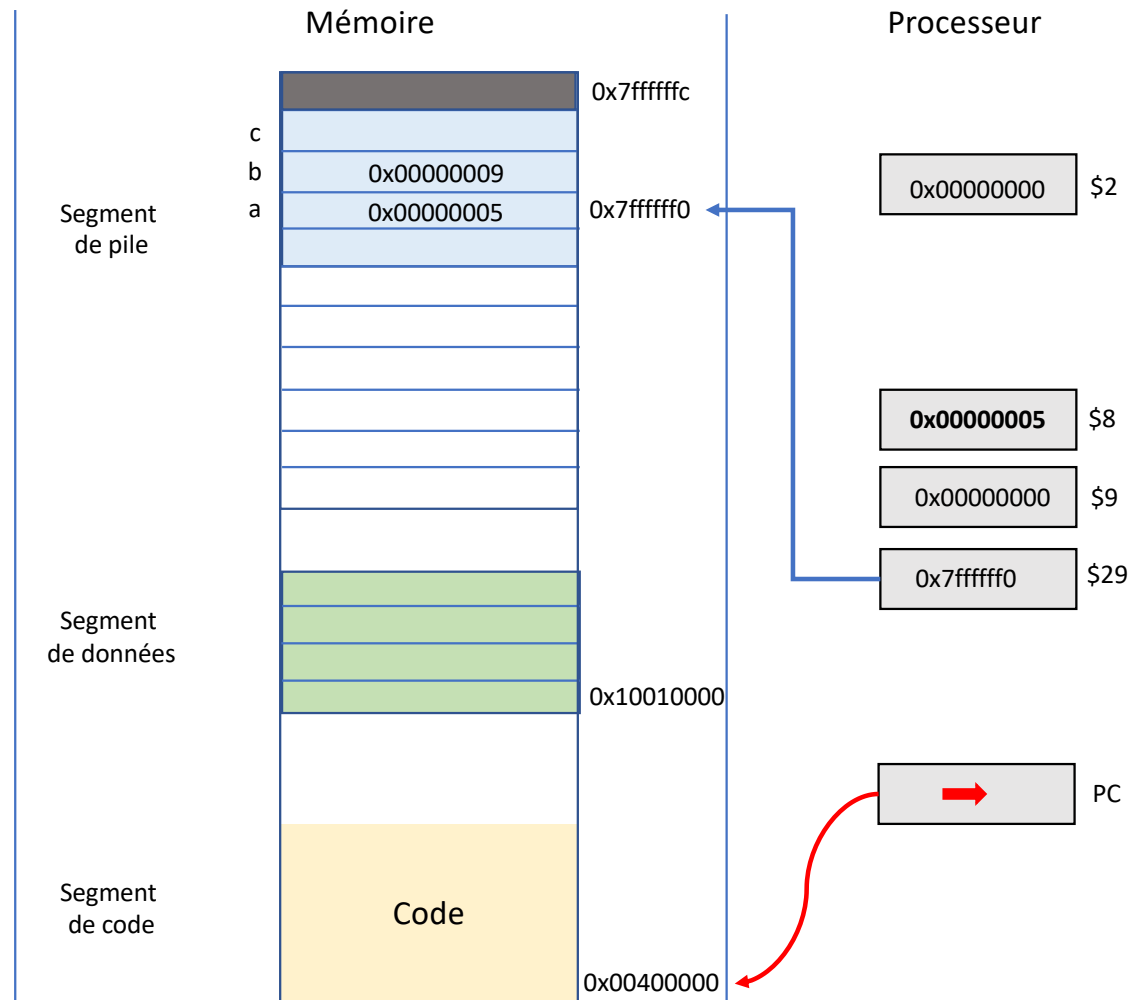
.data
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple

```

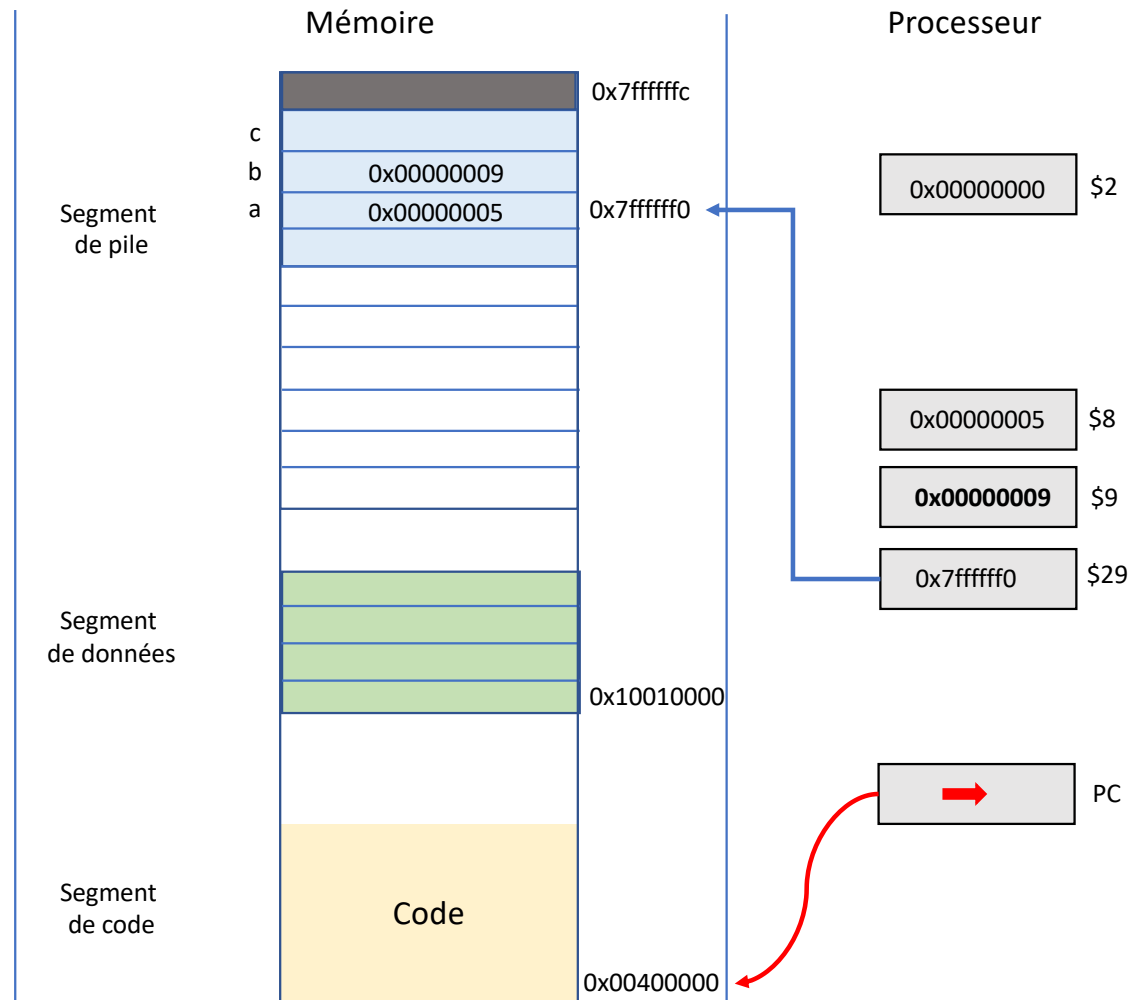
.data
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
➔ addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple

```

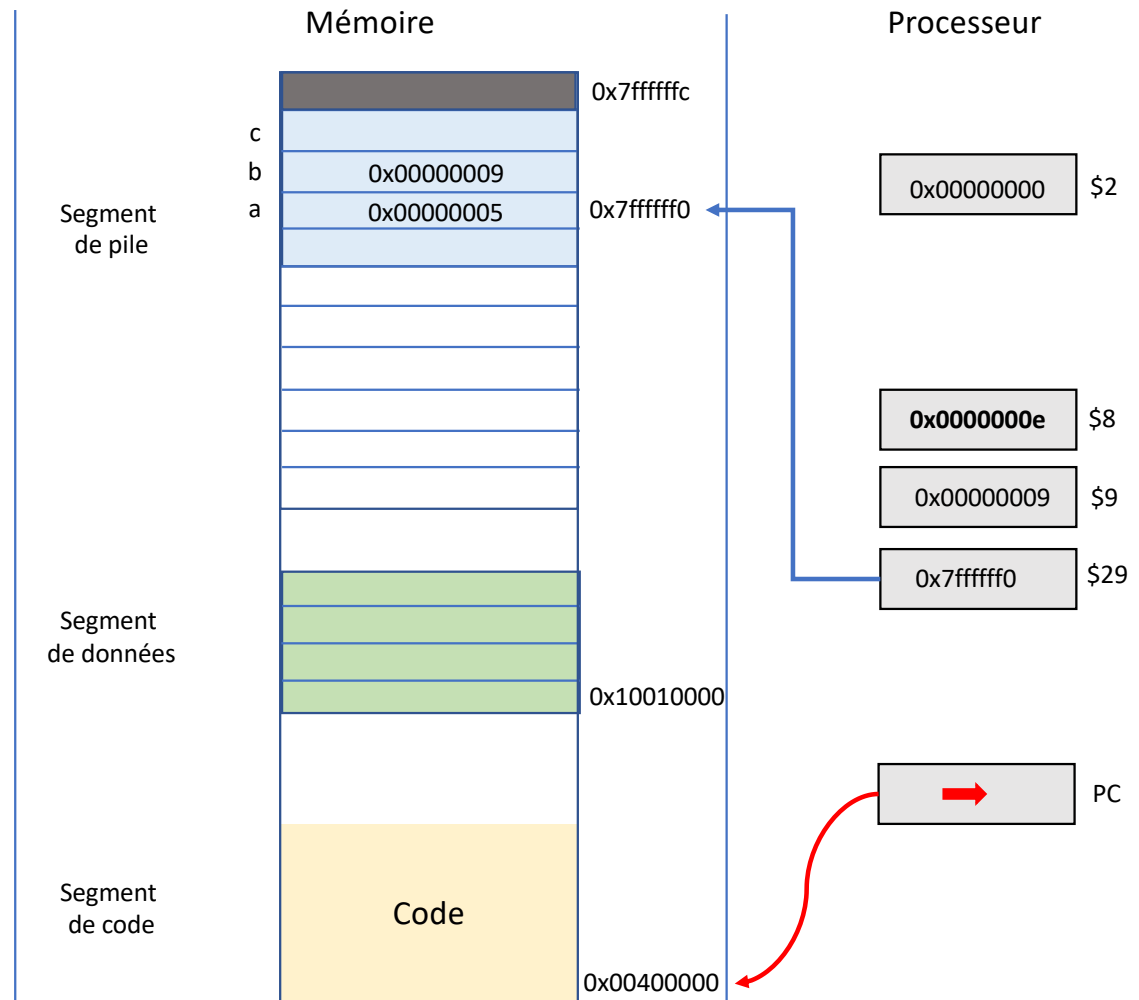
.data
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
➔ sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple

```

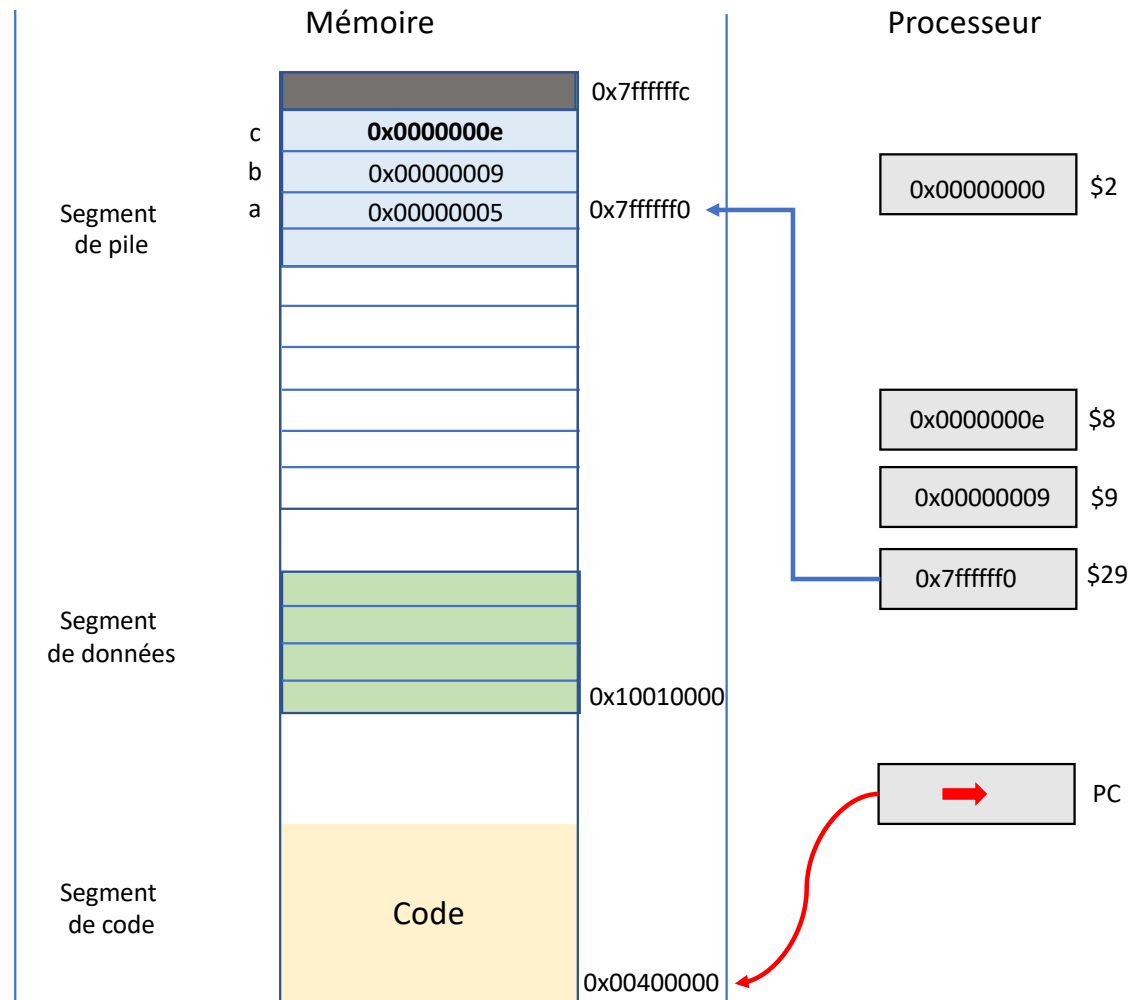
.data
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

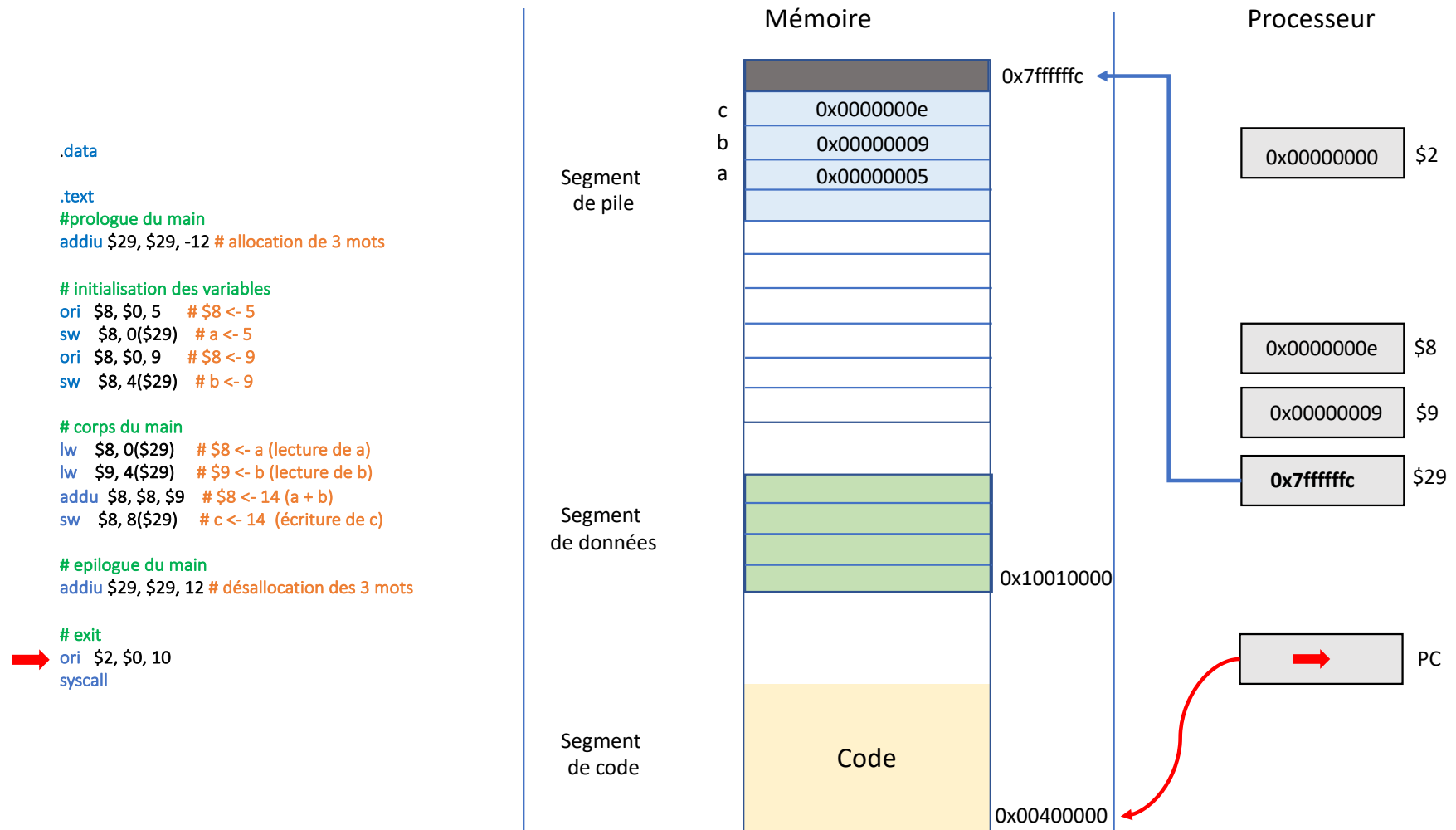
# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# epilogue du main
➔ addiu $29, $29, 12 # désallocation des 3 mots

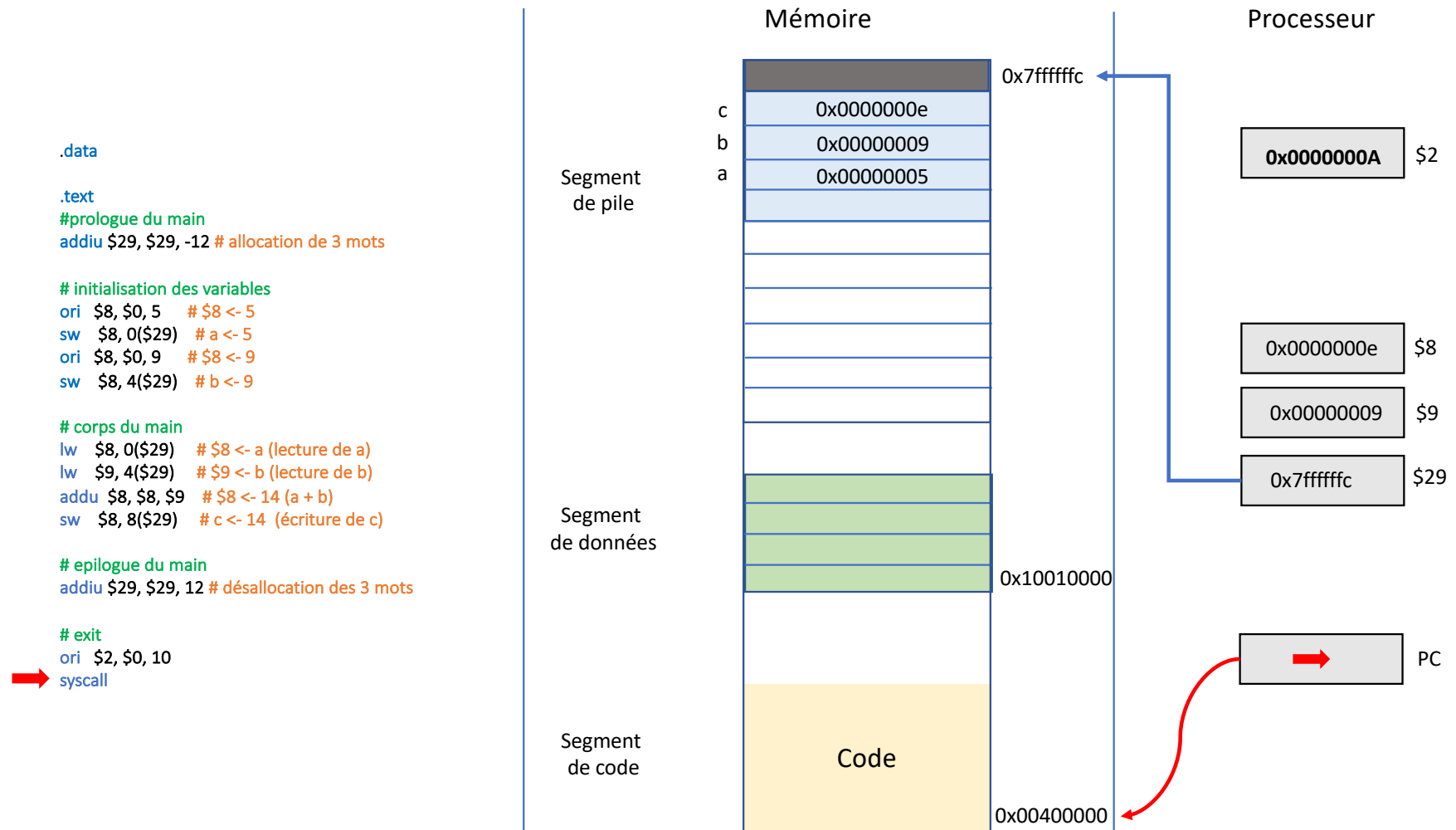
# exit
ori $2, $0, 10
syscall
    
```



Simulation de l'exécution de l'exemple



Simulation de l'exécution de l'exemple



Simulation de l'exécution de l'exemple

```
.data

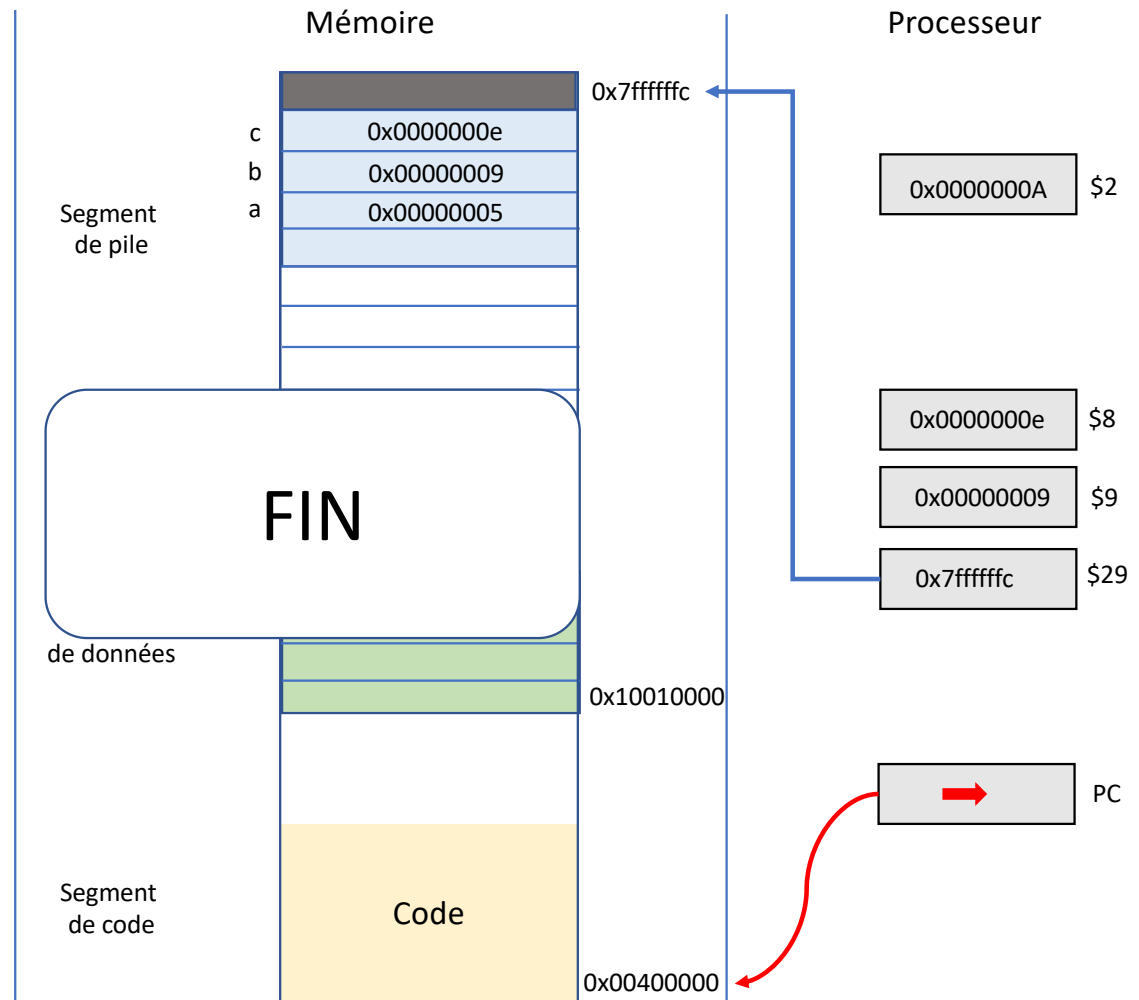
.text
#prologue du main
addiu $29, $29, -12 # allocation de 3 mots

# initialisation des variables
ori $8, $0, 5 # $8 <- 5
sw $8, 0($29) # a <- 5
ori $8, $0, 9 # $8 <- 9
sw $8, 4($29) # b <- 9

# corps du main
lw $8, 0($29) # $8 <- a (lecture de a)
lw $9, 4($29) # $9 <- b (lecture de b)
addu $8, $8, $9 # $8 <- 14 (a + b)
sw $8, 8($29) # c <- 14 (écriture de c)

# épilogue du main
addiu $29, $29, 12 # désallocation des 3 mots

# exit
ori $2, $0, 10
syscall
```



Optimisation des accès aux variables

- On peut *optimiser* le code assembleur pour réduire le nombre d'instructions du code et donc réduire sa taille et son temps d'exécution
- Optimisation manuelle ou *via* des options spécifiques du compilateur
- Optimisation possible : réduction des accès mémoire aux variables
- Cas des variables locales
 - On peut (parfois) faire l'association d'une **variable locale** (i.e. dont l'emplacement mémoire est en pile) avec un registre et éliminer les accès à l'emplacement mémoire de la variable locale, mais il faut garder en tête que c'est une **optimisation**, et non une approche générale
 - Mais même dans ce cas, on doit allouer sur pile la place correspondant à la variable locale optimisée
- Cas des variables globales
 - On peut éviter de relire une variable globale en mémoire lorsque sa valeur est déjà dans un registre et qu'on est sûr qu'elle n'a pas changé
 - On n'a pas le droit d'éliminer les écritures en mémoire des variables globales
- Autre optimisation : élimination de calculs redondants, par exemple une adresse calculée (`&tab[i]`), en conservant la valeur dans un registre

Exemple avec variables locales optimisées en registre

Code C

```
void main() {  
    int a = 5;  
    int b = 9;  
    int c;  
    c = a + b;  
    exit();  
}
```

Code ASM

```
addiu $29, $29, -12 # alloue 3 mots  
ori $8, $0, 5 # a <- 5  
ori $9, $0, 9 # b <- 9  
addu $10, $8, $9 # c <- 14  
addiu $29, $29, 12 # désallocation  
ori $2, $0, 10  
syscall
```

- a est optimisée dans le registre \$8, b est optimisée dans le registre \$9, c est optimisée dans le registre \$10

Exemple de programme complet

Programme C : passer une chaine en majuscule

```
char str[] = "helloworld";

int main() {
    int i = 0;
    while (str[i] != '\0') {
        str[i] = str[i] - 0x20;
        i = i + 1;
    }
    return 0;
}
```

Exemple de programme complet

```
.data
str: .asciiz "helloworld"

.text
# Prologue du main
addiu $29, $29, -4 # 1 variable locale : i
sw    $0, 0($29)   # i = 0

# Corps du main
loop:
# Condition de la boucle while
lui   $8, 0x1001    # $8 = 0x10010000 = @str
lw    $9, 0($29)    # lecture i
addu  $9, $8, $9    # $9 = &str[i] = @str + i
lb    $9, 0($9)     # $9 = str[i]
beq   $9, $0, finloop # test str[i] == 0
# Corps du while
```

Exemple de programme complet

```
# Corps du while
# lecture str[i]
lui $8, 0x1001          # $8 = 0x10010000 = @str
lw $9, 0($29)           # lecture i
addu $9, $8, $9         # $9 = &str[i] = @str + i
lb $9, 0($9)            # $9 = str[i]
addiu $9, $9, -0x20     # str[i] - 0x20
# écriture str[i]
lui $8, 0x1001          # $8 = 0x10010000 = @str
lw $10, 0($29)          # lecture i
addu $10, $8, $10       # $10 = &str[i] = @str + i
sb $9, 0($10)           # écriture str[i]
# i = i + 1
lw $9, 0($29)           # lecture i
addiu $9, $9, 1         # i + 1
sw $9, 0($29)           # écriture i
# Retour au début de la boucle
j loop

finloop:
# Epilogue du main
addiu $29, $29, 4       # Désallocation dans la pile
ori $2, $0, 10
syscall                # exit
```

Exemple de programme complet

Remarques sur le programme Mips

- Un peu long, mais à peu de chose près, c'est exactement le code généré par gcc (sans optimisation)
- Traduction facile à automatiser : c'est ça qu'il faut savoir faire
- Néanmoins, on peut largement optimiser ce code (à la main ou avec une option type -O2)
- Exemples d'optimisations à suivre :
 - 1 optimisation de la variable `i` en registre + élimination d'accès mémoire et calculs d'adresse redondants
 - 2 optimisation qui élimine entièrement l'utilisation de la variable `i` **mais pas son emplacement en pile !**
- Attention :
 - Vous devez savoir écrire des programmes traduits littéralement et comprendre les optimisations possibles des variables locales
 - Optimisation seulement une fois que la version de base est maîtrisée !
- L'optimisation de code relève du domaine de la compilation (= 👍)

Exemple avec optimisation des accès mémoire

```
.data
str: .asciiz "helloworld"

.text
addiu $29, $29, -4    # 1 variable locale : i
ori $8, $0, 0         # i <- 0, $8 <- valeur de i
# Corps du main
lui $9, 0x1001        # $9 = 0x10010000 = @str
loop:
# Condition de la boucle while
addu $10, $9, $8      # $10 <- &(str[i]) = @str + i
lb $11, 0($10)        # $11 <- str[i]
beq $11, $0, finloop  # test str[i] == 0
# Corps du while
addiu $11, $11, -0x20 # $11 <- str[i]-0x20 ($11 déjà contient str[i])

sb $11, 0($10)        # écriture str[i] ($10 déjà contient &(str[i]))
addiu $8, $8, 1       # incrémentation valeur de i
# Retour au début de la boucle
j loop
finloop:
addiu $29, $29, 4     # Désallocation dans la pile
ori $2, $0, 10
syscall              # exit
```

Exemple sans utilisation de la variable locale `i`

```
.data
str: .asciiz "helloworld"

.text
# Prologue du main
addiu $29, $29, -4    # 1 variable locale : i

# Corps du main
lui $8, 0x1001        # $8 = &(str[0]) = str
loop:
lb $9, 0($8)          # $9 = element courant
beq $9, $0, fin       # test element courant == 0

addiu $9, $9, -0x20    # $9 = element courant - 0x20
sb $9, 0($8)          # écriture élément courant
addiu $8, $8, 1        # $8 <- adresse element suivant
                     # incrémentation de l'adresse

j loop
fin:
# Epilogue du main
addiu $29, $29, 4
ori $2, $0, 10
syscall
```

Tableaux et enregistrements en variables locales

- Ce sont des variables locales comme les autres, il faut donc :
 - allouer sur la pile l'espace nécessaire pour ces variables en suivant les consignes,
 - initialiser, en cohérence avec le code source, **chaque** élément des tableaux, **chaque** champ des structures avec des instructions d'écriture mémoire
- Le code du programme principal est ensuite la traduction du code C :
 - quand on doit accéder à une variable locale de type tableau ou enregistrement on sait où elle est rangée en pile, son adresse est relative au sommet de pile ;
 - pour simplifier ces accès, on met en général l'adresse du tableau ou de la structure dans un registre qu'on utilise ensuite lors des accès aux éléments/champs.

Exemple de programme complet

Programme C : passer une chaine en majuscule

```
int main() {  
    int i = 0;  
    char str[] = "abc";  
  
    while (str[i] != '\0') {  
        str[i] = str[i] - 0x20;  
        i = i + 1;  
    }  
    return 0;  
}
```

Tableau local et optimisation des accès mémoire

```
.data
.text
addiu $29, $29, -8    # 1 variable locale : i + ch (4 octets)
ori $8, $0, 0x61      # $8 <- 'a'
sb $8, 4($29)         # str[0] <- 'a'
ori $8, $0, 0x62      # $8 <- 'b'
sb $8, 5($29)         # str[1] <- 'b'
ori $8, $0, 0x63      # $8 <- 'c'
sb $8, 6($29)         # str[3] <- 'c'
ori $8, $0, 0         # $8 <- '\0'
sb $8, 7($29)         # str[3] <- '\0'
ori $9, $0, 0         # i <- 0, $9 <- valeur de i
# Corps du main
addiu $8, $29, 4      # $8 = @str
loop:
# Condition de la boucle while
addu $10, $8, $9      # $10 <- &str[i] = @str + i
lb $11, 0($10)        # $11 <- str[i]
beq $11, $0, finloop  # test str[i] == 0
# Corps du while
addiu $11, $11, -0x20  # $11 <- str[i]-0x20 ($11 contient déjà str[i])
sb $11, 0($10)        # écriture str[i] ($10 contient déjà &str[i])
addiu $8, $8, 1       # incrémentation valeur de i
j loop               # Retour au début de la boucle
finloop:
addiu $29, $29, -8    # Désallocation dans la pile
ori $2, $0, 10
syscall              # exit
```

Conclusion

On a vu

- L'implantation mémoire de données structurées (tableau et enregistrement)
- La notion de contexte d'exécution d'une fonction
- Le segment de pile et son utilisation pour allouer les variables locales du programme principal
- La traduction littérale d'un programme C en assembleur
- Les optimisations possibles
- Les données structurées en variable locale

Vous devez savoir :

- traduire des codes C manipulant des données structurées, ayant des variables locales. Traduction littérale (sans optimisation) puis avec optimisation.

Prochain cours : fonctions et convention d'appels.