

## Chapitre 3

# Appels de fonctions dans les langages à blocs

### Sommaire

---

<b>3.1 Fonctions, pile, récursion, blocs d'activation</b> . . . . .	<b>1</b>
3.1.1 Le problème . . . . .	1
3.1.2 La pile . . . . .	2
3.1.3 Exemple . . . . .	3
<b>3.2 Description détaillée des étapes d'un appel de fonction</b> . . . . .	<b>6</b>
3.2.1 Utilisation des registres \$fp et \$gp . . . . .	6
3.2.2 Séquence d'appel d'une fonction . . . . .	6
3.2.3 Variantes . . . . .	7
3.2.4 Exemple . . . . .	8

---

## 3.1 Fonctions, pile, récursion, blocs d'activation

### 3.1.1 Le problème

#### Les appels de fonctions et la récursion...

On a vu comment écrire en assembleur une boucle qui calcule la factorielle d'un entier lu sur la console, et comment imprimer le résultat.

Pour cela, il a suffi d'utiliser les appels système, quelques registres et des sauts conditionnels très simples.

Pour écrire des fonctions, il ne suffit plus d'utiliser les registres :

- chaque appel de fonction (`jal`) met l'adresse de retour dans le registre `$ra` (31), et si on effectue des appels de fonction imbriqués sans sauver ce registre, on perd les adresses de retour de tous les appels sauf le dernier.
- si l'on autorise les fonctions récursives, chaque instance de la fonction récursive exécute le même code, et donc utilise forcément les mêmes registres, donc si on ne sauve pas ces registres quelque part, on perd la valeur qu'ils avaient dans tous les appels, sauf le dernier

- si l'on autorise dans le langage source des définitions de fonctions imbriquées, on peut alors accéder dans une fonction profonde à des données locales à une fonction qui l'englobe, et donc ces données (on dit qu'elles *échappent*) ne peuvent être maintenue dans des registres

### ... posent un problème...

On peut voir clairement le problème posé par les deux premiers points en programmant la fonction factorielle en assembleur de façon récursive (sur le site du cours, vous avez la version naïve complètement erronée, nous allons la corriger en cours jusqu'à arriver à la version correcte).

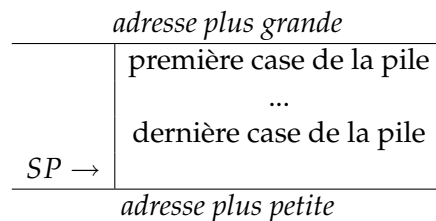
### 3.1.2 La pile

La réponse la plus immédiate à ce problème, pour une large famille de langages source, est l'utilisation d'une *pile* (*stack* en anglais).

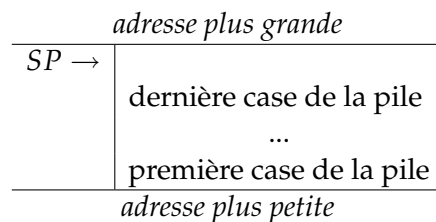
Il s'agit d'une zone contigüe de mémoire gérée de façon LIFO (Last In First Out), avec un pointeur *SP* (Stack Pointer = pointeur de pile) qui indique la limite entre la mémoire appartenant à la pile, et la mémoire libre. N.B. : pour compiler des langages fonctionnels comme OCaml, une pile ne suffira plus.

L'organisation de la pile varie de machine en machine.

Sur certaines machines, la pile grandit vers le bas (Pentium, Sparc, Mips).



Mais sur d'autres elle grandit vers le haut (HPPA)...



Aussi, le registre *SP* pointe sur une case qui sépare la partie utilisée de celle libre de la pile, mais est-ce que cette case fait partie de la partie libre ou utilisée ?

C'est une convention qui dépend de la machine cible.

**Important :** les « conventions » sont là pour permettre à des codes objets produits par des compilateurs différents de pouvoir interagir correctement.

Pour le MIPS, \$sp pointe au dernier mot utilisé.

## Push/Pop

Sur la pile on peut sauvegarder des données (ex : celles qui doivent être préservées lors des appels des fonctions) :

**sauvegarde** « push » de la donnée

**CISC** `pushl %ebp`

**RISC** le choix de `$sp` est juste *une convention*

`sub $sp $sp 4`

`sw $3 ($sp)`

**restauration** « pop » de la donnée

**CISC** `popl %ebp`

**RISC** le choix de `$sp` est juste *une convention*

`lw $3 ($sp)`

`add $sp $sp 4`

### 3.1.3 Exemple

#### La factorielle récursive : version naïve incorrecte

```
# Version de la factorielle avec récursion complètement incorrecte
.data
str1: .ascii "Entrez un entier :"
str2: .ascii "Sa factorielle est "
.text
main: li $v0, 4          # system call code for print_str
      la $a0, str1      # address of string to print
      syscall          # print the string

      li $v0, 5          # system call code for read_int
      syscall          # read int, result in $v0

      move $a0,$v0      # prepare parameter for calling fact
      jal fact         # call fact, on return the result is in $v0

sortie: li $v0, 4        # system call code for print_str
        la $a0, str2    # address of string to print
        syscall        # print the string

        li $v0 1        # system call code for print_int
        move $a0 $3      # integer to print
        syscall        # print the integer

        li $v0 10       # on sort proprement du programme
        syscall        #
```

```

fact:  bgt $a0 1 recur  # si le paramètre est > 0, appel récursif,
      # sinon retourne 1
      li  $3 1          # fact(0) = 1
      jr  $ra          # retourne (adresse de retour dans $ra)

recur: move $t0 $a0     # sauve le paramètre
      subi $a0 $a0 1   # n-1
      jal fact         # appel récursif, le résultat est dans $v0
      mul $3 $t0 $3    # multiplie par le paramètre sauvé
      jr  $ra          # retourne (adresse de retour dans $ra)

```

### La factorielle récursive : on préserve \$ra

```

      # Version du factoriel avec récursion
      .data
str1:  .ascii "Entrez un entier :"
str2:  .ascii "Son factoriel est "
      .text
main:  li  $v0, 4        # system call code for print_str
      la  $a0, str1     # address of string to print
      syscall          # print the string

      li  $v0, 5        # system call code for read_int
      syscall          # read int, result in $v0

      move $a0,$v0     # prépare parameter for calling fact
      jal fact         # call fact, on return the result is in $3

sortie: li  $v0, 4      # system call code for print_str
      la  $a0, str2     # address of string to print
      syscall          # print the string

      li  $v0 1        # system call code for print_int
      move $a0 $3      # integer to print
      syscall          # print the integer

      li  $v0 10       # on sort proprement du programme
      syscall          #

fact:  bgt $a0 1 recur  # si le paramètre est > 0, appel récursif, sinon retourne 1
      li  $3 1          # fact(0) = 1
      jr  $ra          # retourne (adresse de retour dans $ra)

recur: sub  $sp $sp 4   # place pour sauver l'adresse de retour
      sw  $ra ($sp)    # sauve adresse de retour
      move $s0 $a0     # sauve le paramètre

```

```

sub $a0 $a0 1    # n-1
jal fact        # appel récursif, le résultat est dans $3
mul $3 $s0 $3   # multiplie par le paramètre sauve
lw  $ra ($sp)   # restaure adresse de retour
add $sp $sp 4   # libere place pour l'adresse de retour

jr $ra          # retourne (adresse de retour dans $ra)

```

### La factorielle récursive : la bonne version

```

# Version de la factorielle avec récursion
.data
str1: .asciiz "Entrez un entier :"
str2: .asciiz "La factorielle est "
.text
main: li $v0, 4      # system call code for print_str
      la $a0, str1   # address of string to print
      syscall        # print the string

      li $v0, 5      # system call code for read_int
      syscall        # read int, result in $v0

      move $a0,$v0   # prépare parameter for calling fact
      jal fact       # call fact, on return the result is in $3

sortie: li $v0, 4    # system call code for print_str
        la $a0, str2 # address of string to print
        syscall      # print the string

        li $v0 1     # system call code for print_int
        move $a0 $3  # integer to print
        syscall      # print the integer

        li $v0 10    # on sort proprement du programme
        syscall      #

fact:  bgt $a0 1 recur # si le paramètre est > 0, appel récursif, sinon retourne 1
      li  $3 1        # fact(0) = 1
      jr  $ra        # retourne (adresse de retour dans $ra)

recur: sub $sp $sp 8   # place pour sauver l'adresse de retour ET le paramètre
      sw  $ra ($sp)   # sauve adresse de retour
      sw  $a0 4($sp)  # sauve le paramètre
      sub $a0 $a0 1   # n-1
      jal fact        # appel récursif, le résultat est dans $3
      lw  $a0 4($sp)  # restaure le paramètre

```

```

mul $3 $a0 $3    # multiplie par le paramètre
lw  $ra ($sp)   # restaure adresse de retour
add $sp $sp 8    # libere place pour l'adresse de retour
jr  $ra         # retourne (adresse de retour dans $ra)

```

## 3.2 Description détaillée des étapes d'un appel de fonction

### 3.2.1 Utilisation des registres \$fp et \$gp

Dans un programme d'un langage de haut niveau, on peut retrouver, outre les paramètres des fonctions, aussi deux autres types de variables :

**variables globales** visibles partout dans le programme,

```

#include "stdio.h"
int i = 3;
int j;
void stepup(int x) {j+=i*x;}
void main()
{
    j=0; stepup(2); stepup(4)
}

```

Elles sont rangées tout au fond de la pile ou bien sur le tas. On peut y accéder avec une étiquette, ou par décalage (offset) par rapport au pointeur \$gp (exemple : `lw $t0, 8($gp)`). L'avantage de cette dernière solution est qu'elle n'utilise qu'une seule instruction pour un « load » alors que l'utilisation d'une adresse 32 bits nécessite 2 instructions machine sur un RISC (même si l'on n'écrit qu'une seule pseudo-instruction).

**variables locales** visibles par la fonction qui les définit, et éventuellement par les sous-fonctions de celle-ci

```

#include "stdio.h"
void stepup(int x) {int inc=3; return x+inc;}
void main()
{ int j=0;
  j=stepup(j);
}

```

elles sont rangées sur la pile dans une zone contenant tout l'espace nécessaire pour mémoriser les données propres à la fonction : cette zone porte le nom de *bloc* et elle est délimitée par les deux pointeurs \$sp et \$fp, même si l'on peut faire sans \$fp, comme on verra plus loin.

### 3.2.2 Séquence d'appel d'une fonction

Quand une fonction *f* appelle une fonction *g* :

- l'appelant (*f*) sauve les temporaires *t\** qu'il veut préserver et place les paramètres de *g* à la place réservée pour cela dans son bloc

- l’appelé (*g*) alloue son bloc sur la pile, sauve `$fp`, `$ra` si nécessaire, éventuellement sauve les temporaires *s\** et initialise ses variables (prologue)
- le code de l’appelé (*g*) est exécuté
- l’appelé (*g*) restaure si nécessaire les temporaires *s\**, `$fp`, `$ra`, désalloue son bloc, et retourne en exécutant `jr $ra` (épilogue)
- l’appelant (*f*) dépile les paramètres et restaure éventuellement les registres *t\**

### Appel d’une fonction, création d’un bloc sur la pile

Voyons plus en détail comment se déroule un appel de fonction *f*, en supposant que chaque paramètre et variable occupe exactement un *mot* de mémoire. Il y a une partie du travail qui est faite par l’appelant :

- l’appelant met en place les *m* paramètres de la fonction appelée *f* dans l’espace réservé à cet effet du cadre de pile ;
- l’appelant appelle la fonction *f* (instruction assembleur `jal f`).

Et une partie du travail qui est faite par l’appelé :

**prologue** La fonction appelée *f* « empile son bloc » sur la pile

- *f* alloue son bloc (de taille `framesize` mots)
 

```
subu $sp $sp framesize
```

 sauvegarde l’ancienne valeur de FP à une certaine position `frameoffset` dans le bloc...
 

```
sw $fp framesize-frameoffset($sp)
```

 et bouge `$fp` pour pointer à l’ancien `$sp`

```
addi $fp $sp framesize
```

- *f* a reçu par l’appelant dans un registre spécial *ret* (`$ra` sur MIPS) l’adresse de retour<sup>1</sup>. Elle peut le sauver dans son bloc, si nécessaire.

**calcul** on exécute le corps de *f*, le résultat est dans un registre spécial *res* (`$v0` ou `$v1` sur MIPS)

**épilogue** *f* « dépile » son bloc et retourne le contrôle à l’appelant

- *f* désalloue son bloc et restaure les valeurs de *SP* et *FP*, et éventuellement `$ra`...
 

```
lw $fp framesize-frameoffset($sp)
```

```
addu $sp $sp framesize
```

 et saute à l’adresse de retour :

```
jr $ra
```

### 3.2.3 Variantes

**Peut-on se passer de `$fp` ?**

Oui..., mais dans ce cas :

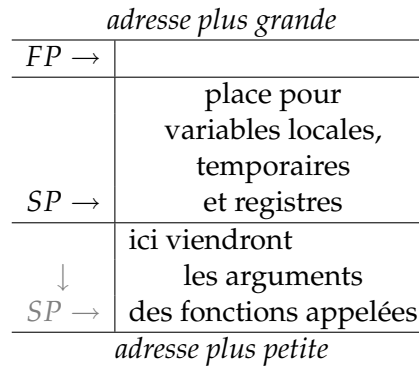
- on accède aux variables locales par décalage par rapport à `$sp`
- mais alors il ne faut pas changer `$sp`, pendant l’exécution de la fonction, pour que ce décalage soit fixe !
- en particulier, il faut allouer déjà au moment de la création du bloc d’une fonction *f* la place pour tous les paramètres que l’on pourrait avoir à empiler pour appeler d’autres fonctions dans le corps de *f*, sinon `$sp` changerait !

<sup>1</sup>Ceci est bien mieux que retrouver l’adresse *ret* sur la pile, pourquoi ?

Cependant, même dans ce cas, on peut continuer à utiliser `$fp`, si on le souhaite.

### IMPORTANT : convention d'appel pour le projet

Dans le projet, vous devez organiser votre bloc (*frame*) comme suit :



La taille du bloc varie.

### 3.2.4 Exemple

#### Un exemple : la fonction fibonacci

Vous pouvez voir tous ces concepts en oeuvre en écrivant l'équivalent en assembleur de la fonction C fibonacci qui calcule

$$\begin{aligned} \text{fibonacci}(0) &= 1 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n+2) &= \text{fibonacci}(n+1) + \text{fibonacci}(n) \end{aligned}$$

Voici le code de la fonction en C :

```
#include "stdio.h"
int fibonacci(int n) {
    int temp;
    if (n==0) {return 1;};
    if (n==1) {return 1;};
    temp=fibonacci(n-1)+fibonacci(n-2);
    return temp;
}

void main(){
    printf("fibonacci(3)=%d\n", fibonacci(3));
    exit(0);
}
```

Et la même en assembleur MARS (d'après gcc) :

```
.data
$LC0:
```

```

        .align      2
        .ascii     "fibonacci(4)="

        .text
        .globl    main
        .globl    fibonacci

main:
    subi        $sp,$sp,40
    sw          $31,32($sp)
    sw          $fp,28($sp)
    sw          $28,24($sp)
    move        $fp,$sp
    li          $4,4                # 0x4
    jal         fibonacci           # rî½sultat dans $v0 = $2
    move        $t0,$v0            # sauvegarde rî½sultat dans $t0
    la          $4,$L3             # $4 = $a0
    li          $v0,4
    syscall
    move        $a0,$t0
    li          $v0,1
    syscall
    li          $v0,10
    syscall
    move        $sp,$fp
    lw          $31,32($sp)
    lw          $fp,28($sp)
    addi        $sp,$sp,40
    jr          $31

fibonacci:
    subi        $sp,$sp,48
    sw          $31,44($sp)
    sw          $fp,40($sp)
    sw          $28,36($sp)
    sw          $16,32($sp)
    move        $fp,$sp
    sw          $4,48($fp)
    lw          $2,48($fp)
    bne        $2,$0,$L3
    li          $2,1                # 0x1
    j          $L2

$L3:
    lw          $2,48($fp)
    li          $3,1                # 0x1

```

```

        bne        $2,$3,$L4
        li         $2,1                # 0x1
        j         $L2

$L4:
        lw         $3,48($fp)
        addi       $2,$3,-1
        move       $4,$2
        jal        fibonacci
        move       $16,$2
        lw         $3,48($fp)
        addi       $2,$3,-2
        move       $4,$2
        jal        fibonacci
        addu       $3,$16,$2
        sw         $3,24($fp)
        lw         $3,24($fp)
        move       $2,$3
        j         $L2

$L2:
        move       $sp,$fp            # ij1/2ilogue
        lw         $31,44($sp)
        lw         $fp,40($sp)
        lw         $16,32($sp)
        addi       $sp,$sp,48
        jr         $31

```