

Chapitre 3

Appels de fonctions dans les langages à blocs

Sommaire

3.1	Fonctions, pile, récursion, blocs d'activation	1
3.1.1	Le problème	1
3.1.2	La pile	2
3.1.3	Exemple	3
3.2	Description détaillée des étapes d'un appel de fonction	6
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 ($j + 1$) met l'adresse de retour dans le registre \$r₃₁ (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).

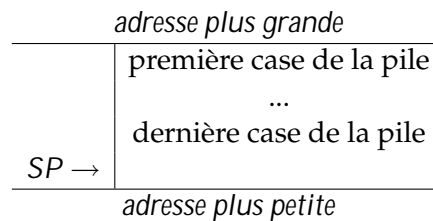
3.1.2 La pile

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

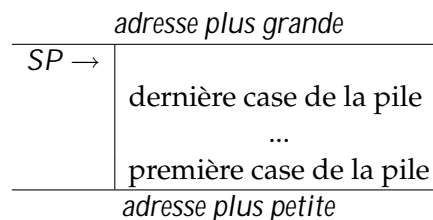
Il s'agit d'une zone contigüe de mémoire gérée de façon LIFO (Last In First Out), avec un pointeur SP (Stack Pointer = pointeur de pile) qui indique la limite entre la mémoire appartenant à la pile, et la mémoire libre. N.B. : pour compiler des langages fonctionnels comme OCaml, une pile ne suffira plus.

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)
dd $sp $sp 4
```

3.1.3 Exemple

La factorielle récursive : version naïve incorrecte

```
# Version de l f ctorielle vec récursion complètement incorrecte
.d t
str1: . sciiz "Entrez un entier :"
str2: . sciiz "S f ctorielle est "
.text
m in: li $v0, 4          # system c ll code for print_str
      l $0, str1         # ddress of string to print
      sysc ll           # print the string

      li $v0, 5          # system c ll code for re d_int
      sysc ll           # re d int, result in $v0

      move $0,$v0        # prep re p r meter for c lling f ct
      j l f ct          # c ll f ct, on return the result is in $v0

sortie: li $v0, 4         # system c ll code for print_str
        l $0, str2        # ddress of string to print
        sysc ll          # print the string

        li $v0 1          # system c ll code for print_int
        move $0 $3        # integer to print
        sysc ll          # print the integer

        li $v0 10         # on sort proprement du progr mme
        sysc ll          #
```

```

f ct:  bgt $ 0 1 recur  # si le p r mètre est > 0,  ppel récursif,
                                # sinon retourne 1
        li  $3 1        # f ct(0) = 1
        jr $r           # retourne ( dresse de retour d ns $r )

recur:  move $t0 $ 0      # s uve le p r mètre
        subi $ 0 $ 0 1   # n-1
        j l f ct        #  ppel récursif, le résult t est d ns $v0
        mul $3 $t0 $3    # multiplie p r le p r mètre s uvé
        jr $r           # retourne ( dresse de retour d ns $r )

```

La factorielle récursive : on préserve \$r

```

        # Version du f ctoriel  vec récursion
        .d t
str1:   . sciiz "Entrez un entier :"
str2:   . sciiz "Son f ctoriel est "
        .text
m in:   li $v0, 4        # system c ll code for print_str
        l  $ 0, str1     # ddress of string to print
        sysc ll         # print the string

        li $v0, 5        # system c ll code for re d_int
        sysc ll         # re d int, result in $v0

        move $ 0,$v0     # prép re p r meter for c lling f ct
        j l f ct        # c ll f ct, on return the result is in $3

sortie: li $v0, 4        # system c ll code for print_str
        l  $ 0, str2     # ddress of string to print
        sysc ll         # print the string

        li $v0 1        # system c ll code for print_int
        move $ 0 $3      # integer to print
        sysc ll         # print the integer

        li $v0 10       # on sort proprement du progr mme
        sysc ll         #

f ct:   bgt $ 0 1 recur  # si le p r mètre est > 0,  ppel récursif, sinon retourne 1
        li  $3 1        # f ct(0) = 1
        jr $r           # retourne ( dresse de retour d ns $r )

recur:  sub $sp $sp 4     # pl ce pour s uver l' dresse de retour
        sw  $r ($sp)     # s uve dresse de retour
        move $s0 $ 0     # s uve le p r mètre

```

```

sub $ 0 $ 0 1      # n-1
j l f ct           # ppel récursif, le résultat est d ns $3
mul $3 $s0 $3      # multiplie p r le p r mètre s uve
lw $r ($sp)        # rest ure dresse de retour
dd $sp $sp 4       # libere pl ce pour l' dresse de retour

jr $r              # retourne ( dresse de retour d ns $r )

```

La factorielle récursive : la bonne version

```

# Version de l f ctorielle vec récursion
.d t
str1: . sciiz "Entrez un entier :"
str2: . sciiz "L f ctorielle est "
.text
m in: li $v0, 4      # system c ll code for print_str
      l $ 0, str1     # ddress of string to print
      sysc ll        # print the string

      li $v0, 5      # system c ll code for re d_int
      sysc ll        # re d int, result in $v0

      move $ 0,$v0    # prép re p r meter for c lling f ct
      j l f ct       # c ll f ct, on return the result is in $3

sortie: li $v0, 4     # system c ll code for print_str
        l $ 0, str2   # ddress of string to print
        sysc ll      # print the string

        li $v0 1      # system c ll code for print_int
        move $ 0 $3    # integer to print
        sysc ll      # print the integer

        li $v0 10     # on sort proprement du progr mme
        sysc ll      #

f ct:  bgt $ 0 1 recur # si le p r mètre est > 0, ppel récursif, sinon retourne 1
      li $3 1         # f ct(0) = 1
      jr $r           # retourne ( dresse de retour d ns $r )

recur: sub $sp $sp 8   # pl ce pour s uver l' dresse de retour ET le p r mètre
      sw $r ($sp)     # s uve dresse de retour
      sw $ 0 4($sp)   # s uve le p r mètre
      sub $ 0 $ 0 1    # n-1
      j l f ct        # ppel récursif, le résultat est d ns $3
      lw $ 0 4($sp)   # rest ure le p r mètre

```

```

mul $3 $0 $3    # multiplie p r le p r mètre
lw  $r ($sp)    # rest ure dresse de retour
dd $sp $sp 8    # libere pl ce pour l' dresse de retour
jr $r           # retourne ( dresse de retour d ns $r )

```

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 $\$fp$, $\$r$ 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^* , $\$fp$, $\$r$, désalloue son bloc, et retourne en exécutant `jr $r` (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 `j 1 f`).

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

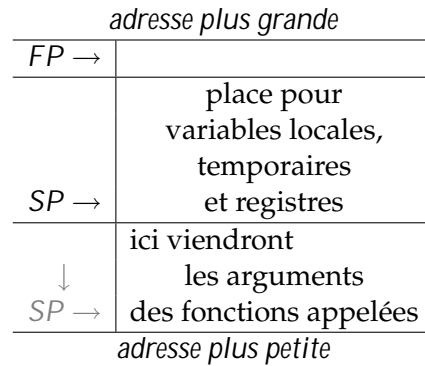
prologue La fonction appelée f « empile son bloc » sur la pile

- f alloue son bloc (de taille $framesize$ mots)

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 œuvre 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:
```



```

        . lign      2
        . sciiz      "fibon cci(4)="

        .text
        .globl      m in
        .globl      fibon cci

m in:
        subi        $sp,$sp,40
        sw          $31,32($sp)
        sw          $fp,28($sp)
        sw          $28,24($sp)
        move        $fp,$sp
        li          $4,4                # 0x4
        j l         fibon cci          #  $r_{i, \frac{1}{2}}$  sult t d ns $v0 = $2
        move        $t0,$v0           # s uveg rde  $r_{i, \frac{1}{2}}$  sult t d ns $t0
        l           $4,$LC0           # $4 = $ 0
        li          $v0, 4
        sysc ll
        move        $ 0,$t0
        li          $v0,1
        sysc ll
        li          $v0,10
        sysc ll
        move        $sp,$fp
        lw          $31,32($sp)
        lw          $fp,28($sp)
        ddi         $sp,$sp,40
        jr          $31
fibon cci:
        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)
        ddi      $2,$3,-1
        move     $4,$2
        j l      fibon_cci
        move     $16,$2
        lw       $3,48($fp)
        ddi      $2,$3,-2
        move     $4,$2
        j l      fibon_cci
        ddu      $3,$16,$2
        sw       $3,24($fp)
        lw       $3,24($fp)
        move     $2,$3
        j        $L2

$L2:
        move     $sp,$fp            # i21/2ilogue
        lw       $31,44($sp)
        lw       $fp,40($sp)
        lw       $16,32($sp)
        ddi      $sp,$sp,48
        jr       $31

```