

Fonctions, pile, récursion et blocs d'activation

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 sauvegarder ce registre, on perd les adresses de retour de tous les appels sauf le dernier.
- ▶ si on s'autorise des 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 on s'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 maintenues 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).

... nécessitent une pile

La réponse la plus immédiate à ce problème, pour une large famille de langages source, est l'utilisation d'une *pile*¹.

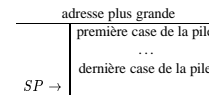
Il s'agit d'une zone contiguë de mémoire gérée de façon LIFO², avec un pointeur `SP`³ qui indique la limite entre la mémoire appartenante à la pile, et la mémoire libre. N.B.: pour compiler des langages fonctionnels comme OCaml, une pile ne suffira plus

L'organisation de la pile varie de machine à machine. Sur certaines machines, la pile grandit vers le bas (Pentium, Sparc, Mips).

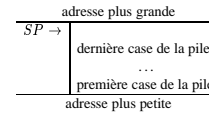
¹stack en anglais

²Last In First Out

³Stack Pointer = pointeur de pile



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 intégrer correctement.

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

Push/Pop

Sur la pile on peut sauvegarder des données⁴:

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`

⁴ex: celles qui doivent être préservées lors des appels des fonctions

La factorielle récursive: version naïve incorrecte

```
# Version du factoriel avec recursion completement incorrecte
.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 # prepare 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    #

ifact: bgt $a0 1 recur # si le parametre est > 0, appel recursif,
# sinon retourne 1
      li $3 1      # fact(0) = 1
      j $ra       # retourne (adresse de retour dans $ra)

recur: move $t0 $a0  # sauve le parametre
      subi $a0 $a0 1 # n-1
      jal fact     # appel recursif, le resultat est dans $3
      mul $3 $t0 $3 # multiplie par le parametre sauve
      j $ra       # retourne (adresse de retour dans $ra)
```

La factorielle récursive: on preserve \$ra

```
# Version du factoriel avec recursion
.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 # prepare 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 parametre est > 0, appel recursif, sinon r
      li $3 1      # fact(0) = 1
      j $ra       # retourne (adresse de retour dans $ra)

recur: sub $sp $sp 4 # place pour sauve l'adresse de retour
      sw $ra ($sp) # sauve adresse de retour
      move $t0 $a0 # sauve le parametre
      sub $a0 $a0 1 # n-1
      jal fact     # appel recursif, le resultat est dans $3
      mul $3 $t0 $3 # multiplie par le parametre sauve
      lw $ra ($sp) # restaure adresse de retour
      add $sp $sp 4 # libere place pour l'adresse de retour

      j $ra       # retourne (adresse de retour dans $ra)
```

La factorielle récursive: la bonne version

```
# Version du factoriel avec recursion
.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  # prepare 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 parametre est > 0, appel recursif, sinon r
        li $3 1      # fact(0) = 1
        j $ra       # retourne (adresse de retour dans $ra)

recur:  sub $sp $sp 8  # place pour sauver l'adresse de retour ET le para
        sw $ra 4($sp) # sauve adresse de retour
        sw $a0 ($sp)  # sauve le parametre
        sub $a0 $a0 1 # n-1
        jal fact     # appel recursif, le resultat est dans $3
        lw $a0 ($sp) # restaure le parametre
        mul $3 $a0 $3 # multiplie par le parametre
        lw $ra 4($sp) # restaure adresse de retour
        add $sp $sp 8 # libere place pour l'adresse de retour
        j $ra       # retourne (adresse de retour dans $ra)

```

Les autres pointeurs: \$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()
{

```

5

```

    j=0; stepup(2); stepup(4)
}

```

elles sont stockées tout au fond de la pile.
On peut y accéder avec une étiquette..., ou par offset au pointeur \$gp.

variables locales visibles par la fonction qui les définit, et éventuellement par les sous-fonctions⁵ 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 stockées 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 on pourrait à la limite faire à moins de \$fp.

Séquence d'appel d'une fonction

- ▶ l'appelant sauve les temporaires t* et empile les paramètres
- ▶ l'appelé alloue son bloc sur la pile, sauve \$fp, \$ra éventuellement sauve les temporaires s* et initialise ses variables (prologue)
- ▶ le code de l'appelé est exécuté
- ▶ l'appelé restaure les temporaires s*, \$fp, \$ra, désalloue son bloc, et retourne (épilogue)
- ▶ l'appelant dépile les paramètres et restaure les registres t*

Appel d'une fonction, création d'un bloc sur la pile

Voyons comment peut se dérouler un appel de fonction *f*, en supposant que chaque paramètre et variable occupe exactement une case mémoire.

Il y a une partie du travail qui est fait par l'appelant:

- ▶ l'appelant mets en place les *m* paramètres actuels de la fonction appelée *f* (c'est bien des "empilements", avec \$sp qui décroît...)
- ▶ l'appelant appelle la fonction *f* (instruction assembleur jal *f*)

Et une partie du travail qui est fait par l'appelé...

⁵on reverra ça en détail plus avant

6

Appel d'une fonction, création d'un bloc sur la pile

prologue *f* "empile son bloc" sur la pile

- ▶ l'appelé, *f*, sauvegarde la valeur de FP...

```

sub $sp $sp 4
sw $fp ($sp)

```

et alloue son bloc (de taille *K* mots)

```

add $fp $sp 4
sub $sp $sp K*4

```

- ▶ *f* a reçu par l'appelant dans un registre spécial *ret* l'adresse de retour⁶. Elle peut le sauver dans son frame, si nécessaire.

calcul on exécute le corps de *f*, le résultat est dans un registre spécial *res*

épilogue *f* "dépile" son bloc et retourne le contrôle à l'appelant

- ▶ *f* désalloue son bloc et restaure les valeurs de *SP* et *FP*, ...

```

add $sp $sp K*4
lw $fp ($sp)
add $sp $sp 4

```

et saute à l'adresse de retour:

```

j $ra

```

Peut-on se passer de \$fp?

Oui ..., mais c'est compliqué:

- ▶ on accède aux variables locales par décalage par rapport à \$sp
- ▶ mais ce décalage n'est pas fixe!
- ▶ en particulier, quand on empile les paramètres d'une fonction, \$sp change!

Donc, on préfère utiliser \$fp, par simplicité.

Un exemple: la fonction fibonacci

On peut voir tout ces concept en oeuvre quand on écrit la fonction `fibonacci`, qui calcule

```

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n + 2) = fibonacci(n + 1) + fibonacci(n)

```

⁶Ceci est bien mieux que retrouver l'adresse *ret* sur la pile, pourquoi?

7

```

#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);
}

```

8