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 régistres 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 on s'authorise des fonctions récursives, chaque instance de la fonction récursive exécute le même code, et donc utilise forcément les mêmes régistres, 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'authorise dans le langage source des définitions de fonctions imbriqués, on peut alors accéder dans une fontion 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 régistres

... posent un problème...

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

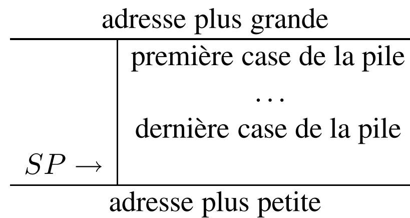
... nécéssitent une *pile*

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

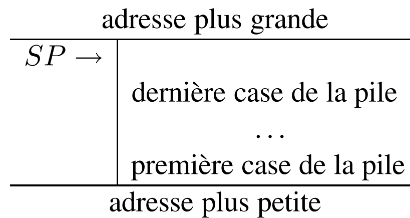
Il s'agit d'une zone contigüe de mémoire gérée de façon LIFO[2], avec un pointeur `SP`[3] qui indique la limite entre la mémoire appartenente à la pile, et la mémoire libre. N.B.: pour compiler des langages fonctionnels comme OCaml, une pile ne suffira plus

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

[1] *stack* en anglais
[2] Last In First Out
[3] Stack 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 dépende de la machine cible.

Important: les "conventions" sont là pour permettre à des codes objets produits par des compilateurs différents de pouvoir intéragir correctement.

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

Push/Pop

Sur la pile on peut sauvegarder des données[4]:

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
```

[4]ex: 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:   .asciiz "Entrez un entier :"
str2:   .asciiz "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 $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 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 $v0
        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:   .asciiz "Entrez un entier :"
str2:   .asciiz "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 sauver l'adresse de retour
         sw  $ra ($sp)      # sauve adresse de retour
         move $s0 $a0       # sauve le parametre
         sub $a0 $a0 1      # n-1
         jal fact           # appel recursif, le resultat est dans $3
         mul $3 $s0 $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 de la factorielle avec recursion
         .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       # 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  $ra ($sp)      # sauve adresse de retour
         sw  $a0 4($sp)     # sauve le parametre
         sub $a0 $a0 1      # n-1
         jal fact           # appel recursif, le resultat est dans $3
         lw  $a0 4($sp)     # restaure le parametre
         mul $3 $a0 $3      # multiplie par le parametre
         lw  $ra  ($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()
{
```

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

elles sont stoquées tout au fond de la pile.
On peut y acceder avec une étiquette…, ou par offset au pointeur `$gp`.

variables locales visibles par la fonction qui les définit, et eventuellement par les sous-fonctions[5] 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 stoquées sur la pile dans une zone contenant tout l'espace nécéssaire pour mémoriser les données propres à la fonction: cette zone porte le nom de *bloc* et ell'est délimitée par les deux pointeurs `$sp` et `$fp`, même si on peut faire à moins de `$fp`, comme on verra plus avant.

Séquence d'appel d'une fonction: généralités

Quand une fonction f appelle une fonction g:

- l'appelant (f) sauve les temporaries `t*` qu'il veut préserver et empile les paramètres pour g
- l'appelé (g) alloue son bloc sur la pile, sauve `$fp`, `$ra` si nécéssaire, éventuellement sauve les temporaires `s*` et initialise ses variables (prologue)
- le code de l'appelé (g) est executé
- l'appelé (g) restaure si nécéssaire les temporaires `s*`, `$fp`, `$ra`, desalloue son bloc, et retourne en exécutant `jr $ra` (epilogue)
- l'appelant (f) dépile les paramètres et restaure éventuellement les régistres `t*`

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

Voyons 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 mets en place les m paramètres actuels de la fonction appelée f (c'est bien des "empilements", avec `$sp` qui décroît…)

[5] on reverra ça en détail plus avant

- l'appelant appelle la fonction f (instruction assembleur `jal f`)

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

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

prologue f "empile son bloc" sur la pile

- l'appelé, f, *alloue son bloc* (de taille $framesize$ mots)

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

  sauvegarde l'ancienne valeur de FP à une certaine position dans le bloc...

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

  et bouge fp pour pointer à la première case du bloc

  ```
  addi $fp $sp framezise-4
  ```

- f a reçu par l'appelant dans un registre spécial ret l'adresse de retour[6]. Elle peut le sauver dans son bloc, si nécessaire.

calcul on exécute le corps de f, le résultat est dans un registre spécial res (le couple `$v0`, `$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:

  ```
  j $ra
  ```

Peut-on se passer de `$fp`?

Oui..., mais dans ce cas:

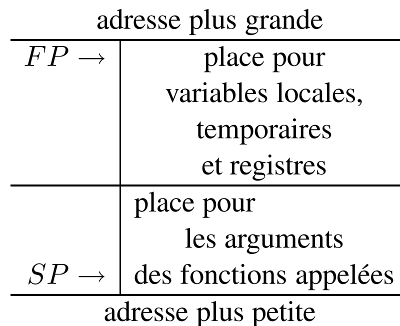
- on accède aux variables locales par décalage par rapport à `$sp`
- mais alors il ne faut pas changer `$sp`, pendant l'execution de la fonction, pour que ce décalage soit fixe!
- en particulier, il faut allouer déjà au moment de la création du bloc d'une fonction f la place pour tous les paramètres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

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

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

IMPORTANT: convention d'appel pour le projet

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



ensuite, SP *n'est plus modifié* dans tout le corps de la fonction!
Attention: par la suite FP sera traité un peu différemment dans votre compilateur, on y reviendra!

Un exemple: la fonction fibonacci

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

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

```
#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);
}
```