

Appels de fonction

Ensimag 1A Apprentissage

Matthieu Moy

Matthieu.Moy@imag.fr

2013

Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Attention



La manière d'utiliser la pile présentée ici n'est pas la seule possible (différente de celle utilisée à l'an dernier en particulier). Cette convention est sans doute la plus simple, mais n'est **pas** compatible avec la dernière version de Mac OS X.

Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Appels de fonctions, retour de fonction : le but

```
void sous_prog() {  
    /* ... */  
    return; /* Revient apres l'appel de sous_prog */  
}  
  
void sous_prog2() {  
    /* ... */  
} /* Retour apres l'appel de sous_prog a la fin de la fonction */  
  
int main(void) {  
    sous_prog(); /* Saute au debut de sous_prog */  
    sous_prog2(); /* Idem pour sous_prog2 */  
}
```

Première tentative : jump

```
.text
        .globl main
main:
        jmp sous_prog
apres_sous_prog:

        jmp sous_prog2
apres_sous_prog2:

        ret

sous_prog:
        movl $42, %eax
        movl $0, %ecx
        jmp apres_sous_prog

sous_prog2:
        movl $42, %edx
        movl %eax, %ecx
        jmp apres_sous_prog2
```

Première tentative : jump

```
.text
        .globl main
main:
        jmp sous_prog
apres_sous_prog:

        jmp sous_prog2
apres_sous_prog2:

        ret

sous_prog:
        movl $42, %eax
        movl $0, %ecx
        jmp apres_sous_prog

sous_prog2:
        movl $42, %edx
        movl %eax, %ecx
        jmp apres_sous_prog2
```

Question



Où est le problème ?

Deuxième tentative : un registre pour l'adresse de retour

```
.text
    .globl main
main:
    movl $apres_sous_prog, %edx
    jmp sous_prog
apres_sous_prog:

    movl $apres_sous_prog2, %edx
    jmp sous_prog
apres_sous_prog2:

    ret /* Fin du main. */

sous_prog:
    movl $42, %eax
    movl $0, %ecx
    jmp *%edx
```


Deuxième tentative : un registre pour l'adresse de retour

```
.text
    .globl main
main:
    movl $apres_sous_prog, %edx
    jmp sous_prog
apres_sous_prog:

    movl $apres_sous_prog2, %edx
    jmp sous_prog
apres_sous_prog2:

    ret /* Fin du main. */

sous_prog:
    movl $42, %eax
    movl $0, %ecx
    jmp *%edx
```

Question



Où est la limitation ?

Deuxième tentative : un registre pour l'adresse de retour

```
.text
    .globl main
main:
    movl $apres_sous_prog, %edx
    jmp sous_prog
apres_sous_prog:

    movl $apres_sous_prog2, %edx
    jmp sous_prog
apres_sous_prog2:

    ret /* Fin du main. */

sous_prog:
    movl $42, %eax
    movl $0, %ecx
    jmp *%edx
```

Question



Et les fonctions
récursives ?

Et les fonctions récursives ?

```
.text
    .globl main
main:
    movl $apres_sous_prog, %edx
    jmp sous_prog
apres_sous_prog:

    ret /* Fin du main. */

sous_prog:
    movl $42, %eax

    /* Ecrase %edx :-( */
    movl $apres_sous_prog2, %edx
    jmp sous_prog
apres_sous_prog2:

    movl $0, %ecx
    jmp *%edx
```

La solution en assembleur Pentium

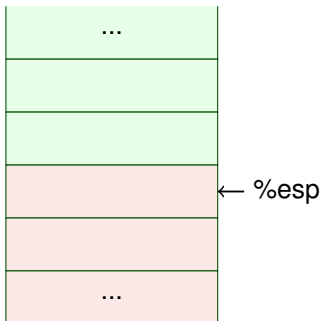
- Instruction `call` *etiquette* :
 - ▶ Empile l'adresse suivant le `call` dans la **pile**
 - ▶ Saute à l'*etiquette*
- Instruction `ret` :
 - ▶ Dépile l'adresse de retour
 - ▶ Saute à cette adresse

```
.text
    .globl main
main:
    call sous_prog
    ret

sous_prog:
    movl $42, %eax
    call sous_prog
    // Il faudrait aussi une condition d'arret ...
    movl $0, %ecx
    ret
```

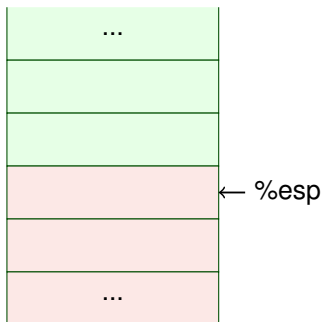
La pile : instruction `call`

Avant `call`

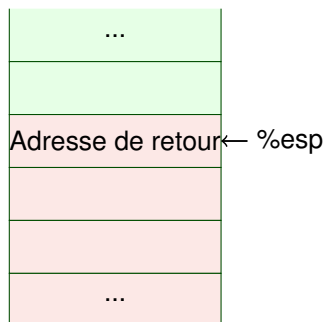


La pile : instruction `call`

Avant `call`



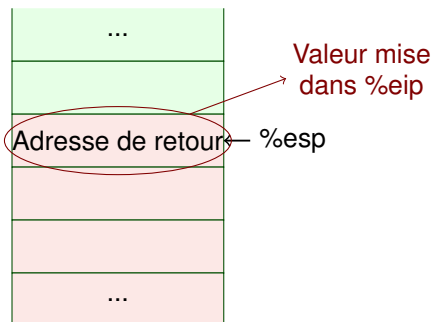
Après `call`



Adresse de retour = adresse suivant le `call`

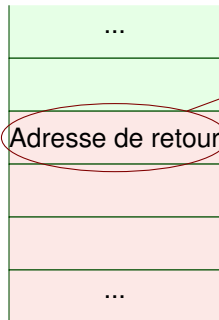
La pile : instruction `ret`

Avant `ret`



La pile : instruction `ret`

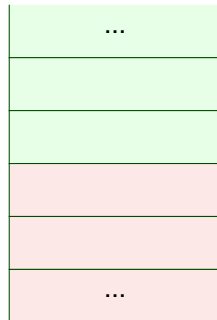
Avant `ret`



Valeur mise
dans `%eip`

Adresse de retour ← `%esp`

Après `ret`



← `%esp`

call et ret : Exemple

```

.text
.globl main
main:
    call sous_prog
    movl $0x1234, %eax
    ret

sous_prog:
    movl $42, %ecx
    ret

(gdb) x/5i $eip
0x8048354 <main>:      call 0x804835f <sous_prog>
0x8048359 <main+5>:    mov  $0x1234,%eax
0x804835e <main+10>:   ret
0x804835f <sous_prog>: mov  $0x2a,%ecx
0x8048364 <sous_prog+5>: ret
(gdb) x/4x $esp
0xbffffec5c:      0xb7e9d455      0x00000001
                  0xbfffece4      0xbfffecec
(gdb) step
9
    movl $42, %ecx
(gdb) x/4x $esp
0xbffffec58:      0x08048359      0xb7e9d455
                  0x00000001      0xbfffece4
(gdb) next
(gdb) next
5
    movl $0x12345678, %eax
(gdb) x/4x $esp
0xbffffec5c:      0xb7e9d455      0x00000001
                  0xbfffece4      0xbfffecec
(gdb) print $eip
$1 = (void (*)( )) 0x8048359 <main+5>

```

Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Paramètres : tentative (ratée) sans utiliser la pile ...

```
int f(int N) {                                f:
    /* utilisation de N */                    /* utilisation de %eax */
}                                              ret

int main(void) {                               main:
    f(5);                                     movl $5, %eax
}                                              call f
                                              ret
```

- Ne marchera pas si `f` est récursive !
- Pose problème dès qu'on a plusieurs appels de fonctions imbriqués
- \Rightarrow on ne va pas faire comme ça ...

Contexte d'exécution d'une procédure

Contexte d'exécution = ensemble des variables accessibles par une procédure

- Variables globales
- Variables locales
- Paramètres (\approx variables locales initialisées par l'appelant)

Contexte d'exécution d'une procédure

Contexte d'exécution = ensemble des variables accessibles par une procédure

- Variables globales

⇒ Existent en 1 et 1 seul exemplaire. Gestion facile avec des étiquettes

- Variables locales

- Paramètres (\approx variables locales initialisées par l'appelant)

Contexte d'exécution d'une procédure

Contexte d'exécution = ensemble des variables accessibles par une procédure

- Variables globales
 - ⇒ Existent en 1 et 1 seul exemplaire. Gestion facile avec des étiquettes
- Variables locales
- Paramètres (\approx variables locales initialisées par l'appelant)
 - ⇒ Existent seulement quand la fonction est appelée

Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales**
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Variable locale \neq Variable globale

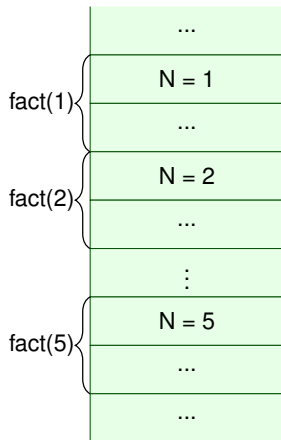
```
int fact(int N) {  
    int res;  
    if (N <= 1) {  
        res = 1;  
    } else {  
        res = N;  
        res = res * fact(N - 1);  
    }  
    return res;  
}
```

- `fact(5)` appelle `fact(4)`
qui appelle `fact(3)` ... \Rightarrow il y
a plusieurs valeurs de `N` en
même temps en mémoire.

Variable locale \neq Variable globale

```
int fact(int N) {  
    int res;  
    if (N <= 1) {  
        res = 1;  
    } else {  
        res = N;  
        res = res * fact(N - 1);  
    }  
    return res;  
}
```

- **fact(5) appelle fact(4)**
qui appelle fact(3) ... \Rightarrow il y
a plusieurs valeurs de N en
même temps en mémoire.



Adressage des variables locales

- Adressage absolu :
⇒ impossible, l'adresse n'est pas fixe
- Adressage relatif à `%esp` :
⇒ possible, mais pénible : `%esp` change souvent de valeur ...
- Solution retenue : Adressage par rapport au pointeur de base `%ebp`
 - ▶ `%ebp` est positionné en entrée de fonction
 - ▶ ... et restauré en sortie de fonction

Gestion du pointeur %ebp : appel de fonction

```
main:
```

```
    // ...
```

```
    call f
```

```
    // ...
```

```
f:
```

```
    // Corps de f
```

```
    // loc2 = loc1
```

fonction
f

fonction
main

← %esp

```
ret
```

Gestion du pointeur %ebp : appel de fonction

main:

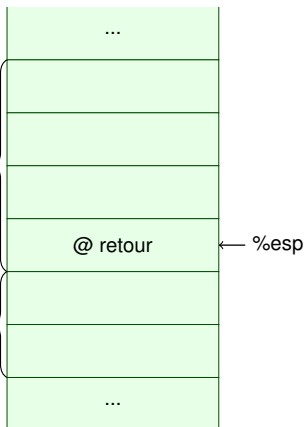
```
// ...  
call f  
// ...
```

f:

```
// Corps de f  
// loc2 = loc1
```

fonction
f

fonction
main



ret

Gestion du pointeur %ebp : appel de fonction

```
main:
```

```
    // ...
```

```
    call f
```

```
    // ...
```

```
f:  pushl %ebp
```

```
    // Corps de f
```

```
    // loc2 = loc1
```

```
    popl %ebp
```

```
    ret
```

fonction
f

Sauvegarde %ebp

← %esp

@ retour

fonction
main

Gestion du pointeur %ebp : appel de fonction

```
main:
```

```
// ...
```

```
call f
```

```
// ...
```

```
f:  pushl %ebp
```

```
    movl %esp, %ebp
```

```
// Corps de f
```

```
// loc2 = loc1
```

```
popl %ebp
```

```
ret
```

fonction
f

Sauvegarde %ebp

%ebp
← %esp

@ retour

fonction
main

Gestion du pointeur %ebp : appel de fonction

```
main:
```

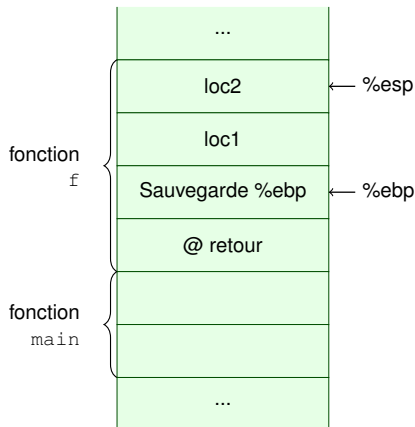
```
// ...
```

```
call f
```

```
// ...
```

```
f:  pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    // Corps de f
    // loc2 = loc1
```

```
    movl %ebp, %esp
    popl %ebp
    ret
```



Gestion du pointeur %ebp : appel de fonction

```
main:
```

```
// ...
```

```
call f
```

```
// ...
```

```
f:  pushl %ebp
```

```
    movl %esp, %ebp
```

```
    subl $8, %esp
```

```
    // Corps de f
```

```
    // loc2 = loc1
```

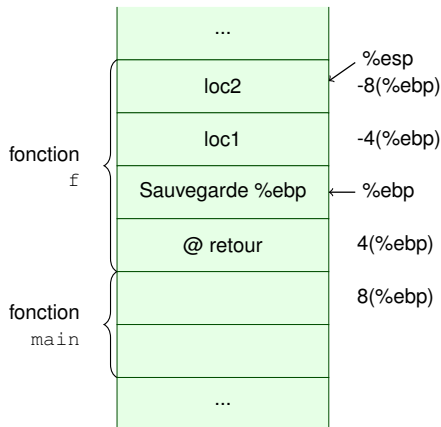
```
    movl -4(%ebp), %eax
```

```
    movl %eax, -8(%ebp)
```

```
    movl %ebp, %esp
```

```
    popl %ebp
```

```
    ret
```



Gestion du pointeur %ebp : retour de fonction

```
main:
```

```
// ...
```

```
call f
```

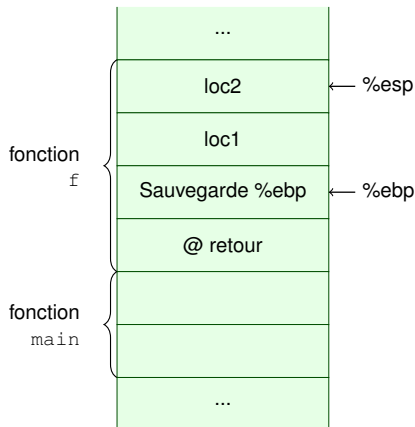
```
// ...
```

```
f:  pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    // Corps de f
    // loc2 = loc1
    movl -4(%ebp), %eax
    movl %eax, -8(%ebp)
```

```
    movl %ebp, %esp
```

```
    popl %ebp
```

```
    ret
```



Gestion du pointeur %ebp : retour de fonction

```
main:
```

```
// ...
```

```
call f
```

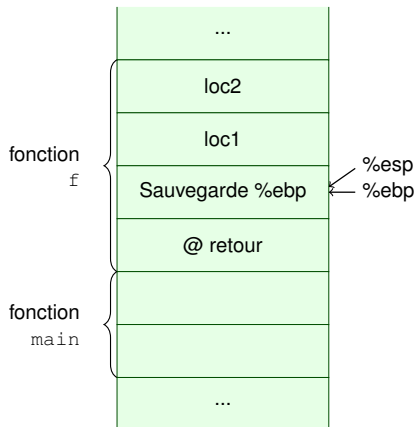
```
// ...
```

```
f:  pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    // Corps de f
    // loc2 = loc1
    movl -4(%ebp), %eax
    movl %eax, -8(%ebp)
```

```
movl %ebp, %esp
```

```
popl %ebp
```

```
ret
```

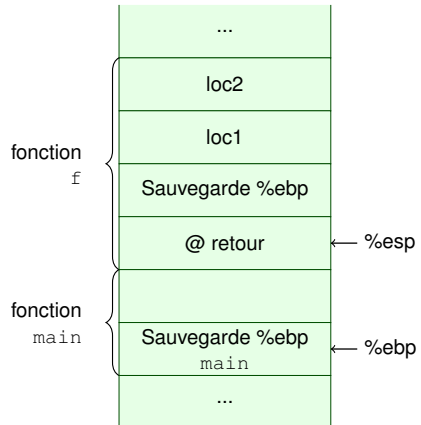


Gestion du pointeur %ebp : retour de fonction

main:

```
// ...  
call f  
// ...
```

```
f:  pushl %ebp  
    movl %esp, %ebp  
    subl $8, %esp  
    // Corps de f  
    // loc2 = loc1  
    movl -4(%ebp), %eax  
    movl %eax, -8(%ebp)  
  
    movl %ebp, %esp  
    popl %ebp  
    ret
```

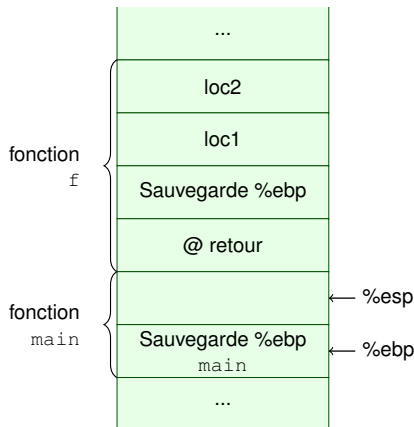


Gestion du pointeur %ebp : retour de fonction

main:

```
// ...  
call f  
// ...
```

```
f:  pushl %ebp  
    movl %esp, %ebp  
    subl $8, %esp  
    // Corps de f  
    // loc2 = loc1  
    movl -4(%ebp), %eax  
    movl %eax, -8(%ebp)  
  
    movl %ebp, %esp  
    popl %ebp  
  
ret
```



Entrée et sortie de fonction : syntaxe alternative

```
f:  pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
```

```
f:  enter $8, $0
```

```
// Corps de f
// ...
```



```
// Corps de f
// ...
```

```
movl %ebp, %esp
popl %ebp
ret
```

```
leave
```

```
ret
```



enter n'est pas géré par valgrind ...

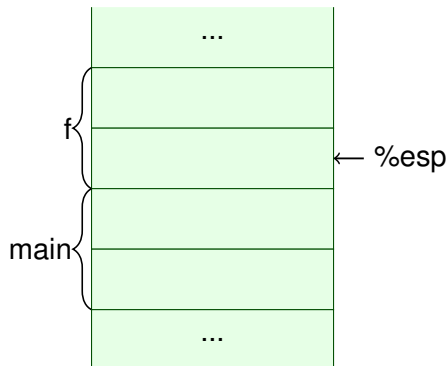
Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 **Passage de paramètre**
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Paramètres passés sur la pile : l'idée

```
int f(int N) {  
    /* ... */  
}
```

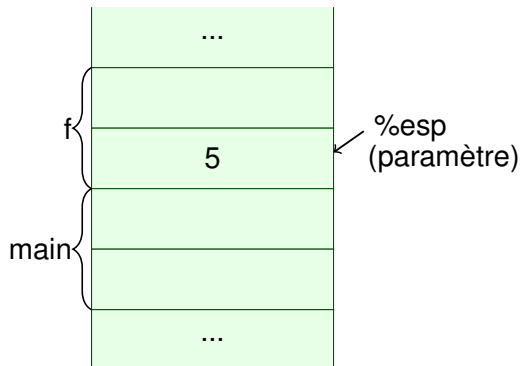
```
int main(void) {  
    f(5);  
}
```



Paramètres passés sur la pile : l'idée

```
int f(int N) {  
    /* ... */  
}
```

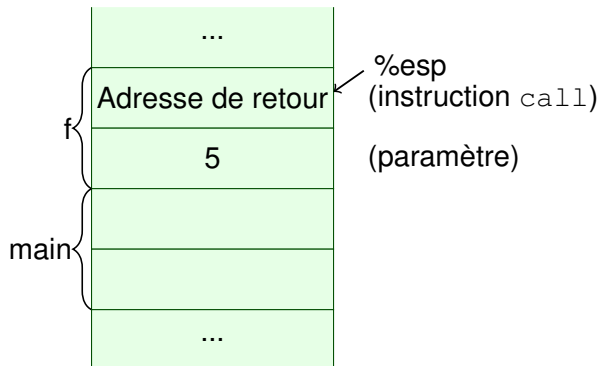
```
int main(void) {  
    f(5);  
}
```



Paramètres passés sur la pile : l'idée

```
int f(int N) {  
    /* ... */  
}
```

```
int main(void) {  
    f(5);  
}
```



Paramètres passés sur la pile : comment ?

- Solution retenue : empilement des paramètres avec `push` :

```
pushl param3
pushl param2
pushl param1
call fonction
addl $12, %esp // on remet le pointeur de pile
               // au bon endroit
...
```

- Alternative (pas dans ce cours) : pré-allocation de la place pour les paramètres, puis `movl param1, 0(%esp), ...`

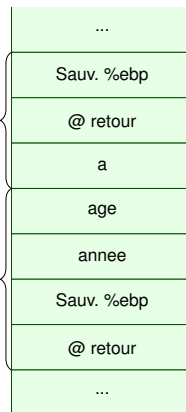
Passage de paramètres : exemple

```
unsigned calcule_age(unsigned a) {  
    return 2013 - a;  
}
```

```
int main(void) {  
    unsigned annee, age;  
    printf("Annee de naissance ?");  
    scanf("%u", &annee);  
    age = calcule_age(annee);  
    printf("Age : %u ans.\n", age);  
    return 0;  
}
```

calcule_age

main

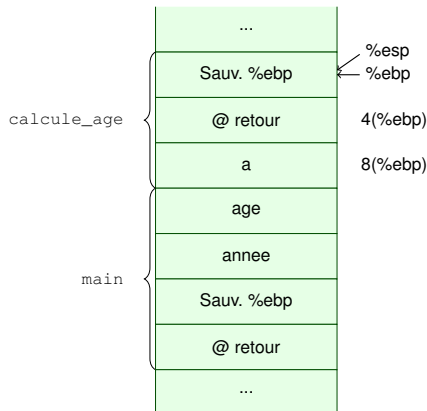


Passage de paramètres : exemple

```
unsigned calcule_age(unsigned a) {  
    return 2013 - a;  
}
```



```
calcule_age:  
    pushl %ebp  
    movl %esp, %ebp  
    // Pas de variable locale  
  
    movl $2013, %eax  
    subl 8(%ebp), %eax  
  
    // Valeur de retour dans %eax  
    // (par convention)  
    leave  
    ret
```

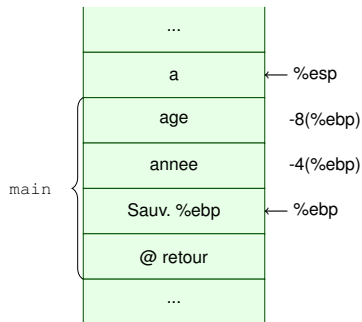


Passage de paramètres : exemple

```
int main(void) {
    unsigned annee, age;
    // ...
    age = calcule_age(annee);
    // ...
}
```



```
main: pushl %ebp
      movl %esp, %ebp
      // 2 variables locales => 8 octets
      subl $8, %esp
      // ...
      // age = calcule_age(annee)
      pushl -4(%ebp)
      call calcule_age
      // un push de 4 octets a depiler
      addl $4, %esp
      // Valeur de retour
      // dans %eax
      movl %eax, -8(%ebp)
      // ...
      leave
      ret
```

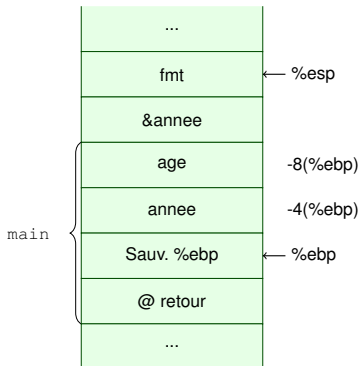


Passage de paramètres par adresse

```
int main(void) {
    unsigned annee, age;
    // ...
    scanf("%u", &annee);
    // ...
}
```



```
main: pushl %ebp
      movl %esp, %ebp
      // 2 variables locales => 8
      subl $8, %esp
      // ...
      // scanf("%u", &annee);
      movl %ebp, %eax // ou bien :
      addl $-4, %eax // leal -4(%ebp), %eax
      pushl %eax
      pushl $fmt_u
      call scanf
      // 2 push de 4 octets a depiler
      addl $8, %esp
      // ...
      leave
      ret
```

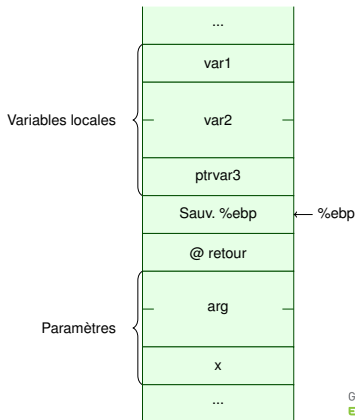


Paramètres de taille \neq 32 bits

- Paramètres plus petits que 32 bits (char, short int, ...) : on arrondit à 32 bits (i.e. 4 octets)
- Paramètres plus grands : on calcule la taille (et on arrondit au multiple de 4 octets supérieur si besoin)

```
struct pair {  
    int first;  
    int second;  
};
```

```
void f(struct pair arg, int x) {  
    int var1;  
    pair var2;  
    pair *ptrvar3;  
    // ...  
}
```



Demo ...

GDB sur le code de 6-age.S :

```
#ifndef __APPLE__
#define printf _printf
#define scanf _scanf
#define main _main
#endif

.text
calcul_age:
    pushl %ebp
    movl %esp, %ebp
    // Pas de variable locale

    movl $2013, %eax
    subl $(%ebp), %eax

    leave
    ret

.globl main
main:
    pushl %ebp
    movl %esp, %ebp
    // de la place pour :
    // 2 variables locales => 8 octets
    subl $8, %esp

    // Corps de main()

    // printf("Année de naissance ? ");
    pushl $fmt_annee
    call printf
    addl $4, %esp

    // scanf("%i", &annee);
    movl %ebp, %eax
    addl $-4, %eax
    pushl %eax
    pushl $fmt_u
    call scanf
    addl $8, %esp

    // age = calcul_age(annee)
    pushl $-4(%ebp)
    call calcul_age
    addl $4, %esp
    // Valeur de retour
    // dans %eax
    movl %eax, -8(%ebp)

    pushl -8(%ebp)
    pushl $fmt_age
    call printf
    addl $8, %esp

    leave
    ret

.data
fmt_annee: .asciz "Année de naissance ? "
fmt_age:  .asciz "Age : %u ans.\n"
fmt_u:    .asciz "%u"
```


Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres**
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Sauvegarde/restauration des registres

- Problème : Registre = variable globale

```
movl $4, %ecx  
call f           // Peut utiliser %ecx  
movl %ecx, ...   // %ecx a ete modifie
```

- Sauvegarde possible :

- ▶ Par l'appelant préalablement à l'appel :
⇒ Restauration faite au retour, chez l'appelant
- ▶ Par l'appelé :
⇒ Restauration avant le retour, chez l'appelé

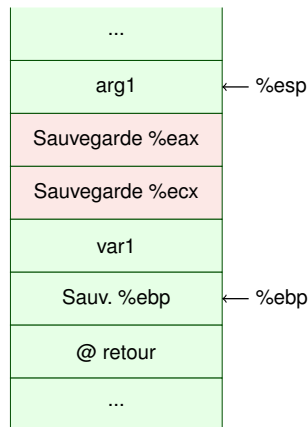
Sauvegarde/restauration des registres par l'appelant

- Sauvegarde dans la pile avant l'appel :

```
pushl %ecx  
pushl %eax
```

- Restauration après l'appel :

```
popl %eax  
popl %ecx
```



Sauvegarde/restauration des registres par l'appelé

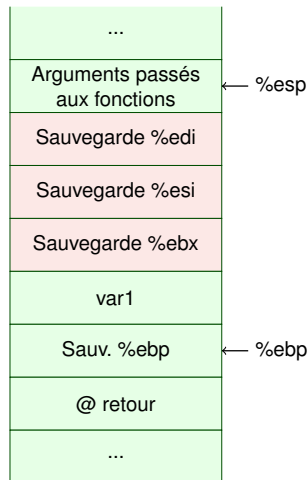
- Sauvegarde dans la pile en début de fonction :

```
f: pushl %ebp
    movl %esp, %ebp
    // Sauvegarde des registres
    subl $taille, %esp
    pushl %ebx
    pushl %esi
    pushl %edi
```

- Restauration en fin de fonction :

```
// Restauration
popl %edi
popl %esi
popl %ebx

movl %ebp, %esp // ou
popl %ebp      // leave
ret
```



Sauvegarde des registres

Sauvegarde par l'appelant/appelé : Que choisir ?

- Il faut que l'appelant et l'appelé aient la même convention !
- On peut avoir une convention différente par registre :
 - ▶ Registres « **scratch** » (volatiles) \Rightarrow l'appelé n'est pas tenu de sauvegarder. L'appelant sauvegarde si besoin.
 - ▶ Registres « **non scratch** » \Rightarrow l'appelé doit sauvegarder les registres dont il se sert.

Sauvegarde des registres

Sauvegarde par l'appelant/appelé : Que choisir ?

- Il faut que l'appelant et l'appelé aient la même convention !
- On peut avoir une convention différente par registre :
 - ▶ Registres « **scratch** » (volatiles) \Rightarrow l'appelé n'est pas tenu de sauvegarder. L'appelant sauvegarde si besoin.
 - ▶ Registres « **non scratch** » \Rightarrow l'appelé doit sauvegarder les registres dont il se sert.
- Conseils :
 - ▶ Utiliser les registres « scratch » comme des temporaires pendant l'évaluation d'une expression (i.e. quelques lignes du programme assembleur)
 - ▶ Utiliser les registres « non-scratch » pour conserver des valeurs comme des variables locales, mais ne pas oublier de les sauvegarder.

Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons**
- 7 Récapitulatif
- 8 Kinesthetic learning

Conventions de liaisons : définition

- **Convention de liaison** (ABI, Application Binary Interface) = conventions de programmation imposées par le système aux applications qui l'utilisent.
- **Peuvent imposer :**
 - ▶ un certain nombre d'appels au système et la façon de les réaliser
 - ▶ les adresses mémoires utilisables par un programme
 - ▶ des conventions d'utilisation des registres
 - ▶ des conventions d'utilisation de la pile

Conventions de liaison Intel 32 bits Unix

- Le format d'un bloc de pile associé à un appel est conforme à ce que nous avons présenté.
- L'allocation et la libération des paramètres est faite par l'appelant
- Les paramètres d'une procédure sont empilés de la droite vers la gauche : $f(p_1, p_2, \dots, p_n) \Rightarrow$ on empile d'abord p_n .
- Paramètre par référence : adresse de la variable effective sur 4 octets
- Paramètre par valeur, de type simple (entier, pointeur) : on empile la valeur effective sur 4 octets.
- Pas de passage de paramètres par registre

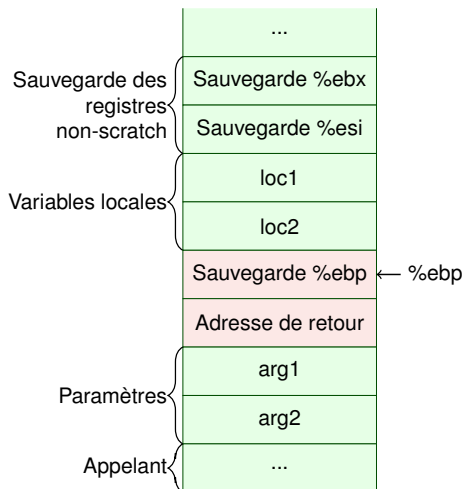
Gestion des Registres

- `%ebx`, `%edi`, `%esi` : registres généraux « non-scratch ».
⇒ On sauvegarde si on utilise
- `%ebp` et `%esp` : registres « non-scratch » également, mais utilisation bien particulière.
- Les autres (`%eax`, `%ecx`, `%edx`, ...) sont « scratch ».
⇒ On fait ce qu'on veut avec, mais un `call` peut les modifier
- `%eax` contient le résultat d'une fonction.

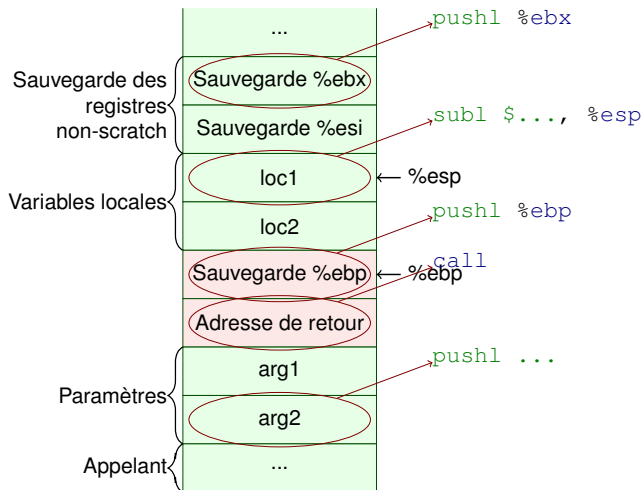
Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif**
- 8 Kinesthetic learning

Un cadre de pile typique



Qui a fait quoi avec la pile ?



Sommaire

- 1 Appels et retour de fonction
- 2 Pile, variables locales, paramètres
- 3 Variables locales
- 4 Passage de paramètre
- 5 Gestion des registres
- 6 Conventions de liaisons
- 7 Récapitulatif
- 8 Kinesthetic learning

Programme à exécuter

```

.data
fmt_d: .asciz "%d\n"

.text
.globl main
// int main() {

main:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp

    // int a = 42;
    movl $42, -4(%ebp)

    // increment(&a);
    movl %ebp, %eax
    addl $-4, %eax
    pushl %eax
    call increment
    addl $4, %esp

    // printf("%d\n", a);
    pushl -4(%ebp)
    pushl $fmt_d
    call printf
    addl $8, %esp

    // return 0;
    xorl %eax, %eax
    leave
    ret

```

```

// void increment(int *x)
increment:
    pushl %ebp
    movl %esp, %ebp

    // (*x)++;
    movl 8(%ebp), %eax
    incl (%eax)

    leave
    ret

```

```

#include <stdio.h>

void increment(int *x) {
    (*x)++;
}

int main() {
    int a = 42;
    increment(&a);
    printf("%d\n", a);
    return 0;
}

```

Pile d'exécution

