

Appels de fonction

Ensimag 1A Apprentissage

Matthieu Moy

Matthieu.Moy@imag.fr

2012



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.



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 ?



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
```



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 f() {
    /* ... */
    return; /* Revient apres l'appel de f */
}

void g() {
    /* ... */
}

int main(void) {
    f(); /* Saute au debut de f */
    g(); /* Idem pour g */
}
```



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_prog2
apres_sous_prog2:
    ret /* Fin du main. */

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

Question



Où est la limitation ?

Question



Et les fonctions récursives ?



La solution en assembleur Pentium

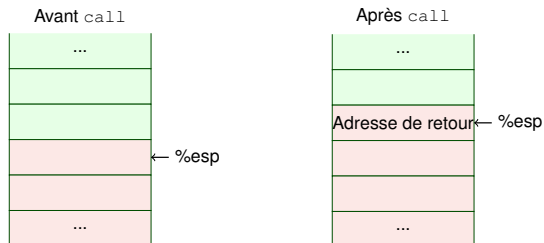
- Instruction **call** *etiquette* :
 - Empile l'adresse suivant le call dans la **pile**
 - Saute à l'*etiquette*
- Instruction **ret** :
 - Dépile l'adresse de retour
 - Saute à cette adresse

```
.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
```



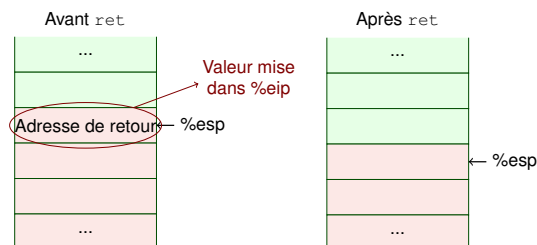
La pile : instruction call



Adresse de retour = adresse suivant le call



La pile : instruction ret



call et ret : Exemple

```
(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
0xbfffec5c: 0xb7e9d455 0x00000001
0xbfffec5d: 0xbfffec54 0xbfffec5c
(gdb) step
9
(gdb) x/4x $esp
0xbfffec58: 0x08048359 0xb7e9d455
0xbfffec59: 0x00000001 0xbfffec58
(gdb) next
(gdb) next
5
(gdb) x/4x $esp
0xbfffec5c: 0xb7e9d455 0x00000001
0xbfffec5d: 0xbfffec54 0xbfffec5c
(gdb) print $eip
$1 = (void (*)()) 0x8048359 <main+5>
```



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

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

main:
    movl $5, %eax
    call f
    ret
```

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



Contexte d'exécution d'une procédure

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

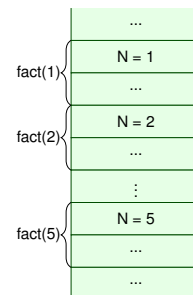
- Variables globales
⇒ Existent en 1 et 1 seul exemplaire. Gestion facile avec des étiquettes
- Variables locales
- Paramètres (≈ variables locales initialisées par l'appelant)
⇒ Existent seulement quand la fonction est appelée



Variable locale ≠ 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) ... ⇒ 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

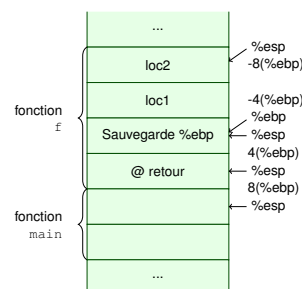


Gestion du pointeur %ebp : appel de fonction

```
main:
    // ...
    call f
    // ...

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

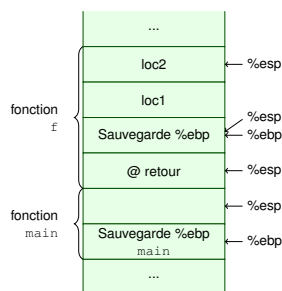
    movl %ebp, %esp
    popl %ebp
    ret
```



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

// Corps de f
// ...

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

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



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

leave
ret
```



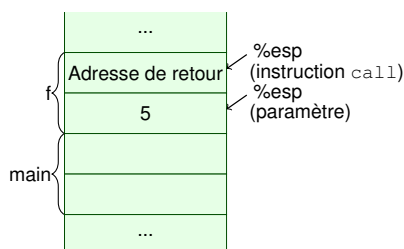
enter n'est pas géré par valgrind ...



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
...
```

- Alternative : pré-allocation de la place pour les paramètres, puis

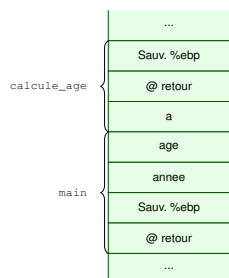
```
movl param1, 0(%esp), ...
```



Passage de paramètres : exemple

```
unsigned calculer_age(unsigned a) {
return 2012 - a;
}

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



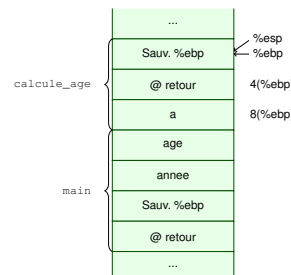
Passage de paramètres : exemple

```
unsigned calculer_age(unsigned a) {
return 2012 - a;
}

calculer_age:
pushl %ebp
movl %esp, %ebp
// Pas de variable locale

movl $2012, %eax
subl 8(%ebp), %eax

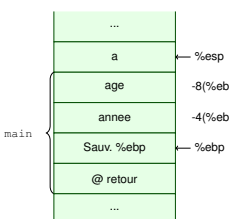
// Valeur de retour dans %eax
// (par convention)
leave
ret
```



Passage de paramètres : exemple

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

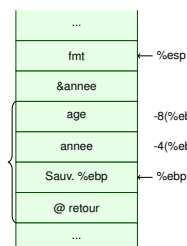
main: pushl %ebp
      movl %esp, %ebp
      // 2 variables locales => 8 octets
      subl $8, %esp
      // ...
      // age = calculer_age(annee)
      pushl -4(%ebp)
      call calculer_age
      // un push de 4 octets a depiler
      addl $4, %esp
      // Valeur de retour
      movl %eax, -8(%ebp)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

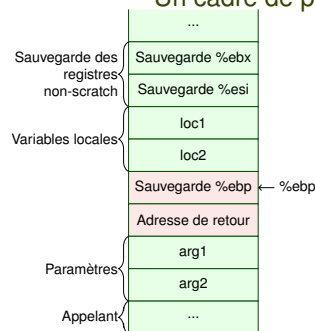


Gestion des Registres

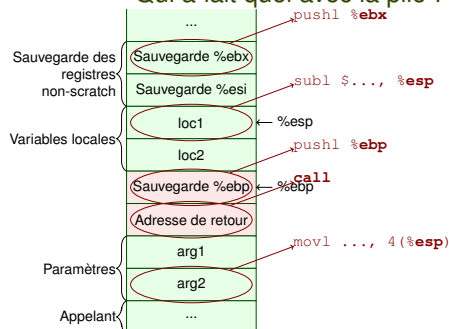
- `%ebx`, `%edi`, `%esi` : registres généraux « non-scratch ».
⇒ On sauvegarde si on utilise
- `%ebp` et `%esp` : registres « non-scratch » également, mais utilisation bien particulière.
- Les autres (`%eax`, `%ecx`, `%edx`, ...) sont « scratch ».
⇒ On fait ce qu'on veut avec, mais un `call` peut les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



Un cadre de pile typique



Qui a fait quoi avec la pile ?



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

