Appels de fonction

Ensimag 1A Apprentissage

Matthieu Moy

Matthieu.Moy@imag.fr

2010

Sommaire

Attention

⚠ La manière d'utiliser la pile présentée ici est subtilement différente de celle présentée dans le polycopié. L'intérêt de cette nouvelle manière est qu'elle est compatible avec la dernière version de Mac OS X.

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 */
}
```


Première tentative : jump

```
#ifdef __APPLE__
#define main _main
#endif

.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
```

Question

Où est le problème ?

Deuxième tentative : un registre pour l'adresse de retour

```
#ifdef __APPLE__
#define main _main
#endif

.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 ?

Question

Et les fonctions récursives ?

Et les fonctions récursives ?

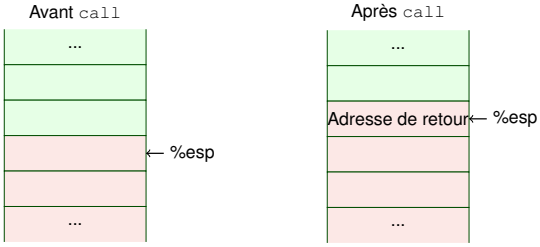
```
#ifdef __APPLE__
#define main _main
#endif

.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

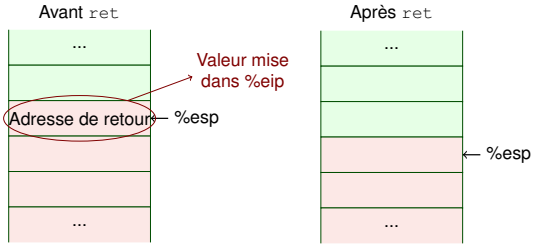
```
.text
        .globl main
main:
        call sous_prog
        ret

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

La pile : instruction `call`



Adresse de retour = adresse suivant le `call`

La pile : instruction `ret`



`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
0xbfffec5c:     0xb7e9d455      0x00000001
                0xbfffece4      0xbfffecec
(gdb) step
9               movl $42, %ecx
(gdb) x/4x $esp
0xbfffec58:     0x08048359      0xb7e9d455
                0x00000001      0xbfffece4
(gdb) next
(gdb) next
5               movl $0x12345678, %eax
(gdb) x/4x $esp
0xbfffec5c:     0xb7e9d455      0x00000001
                0xbfffece4      0xbfffecec
(gdb) print $eip
$1 = (void (*)()) 0x8048359 <main+5>
```


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

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

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

```
f:
    /* utilisation de %eax */
    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
- ⇒ on ne va pas faire comme ça ...

Contexte d'exécution d'une procédure

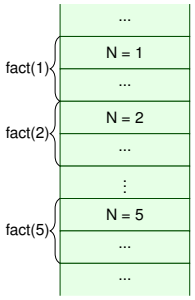
Contexte d'exécution = ensemble des variables accessibles par une procédure

- 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.

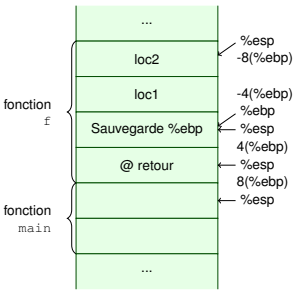


Adressage des variables locales

- 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

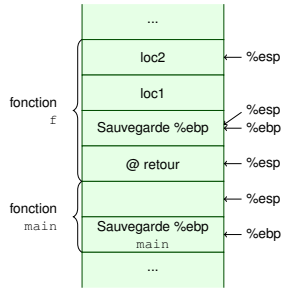
Gestion du pointeur %ebp : appel de fonction

```
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

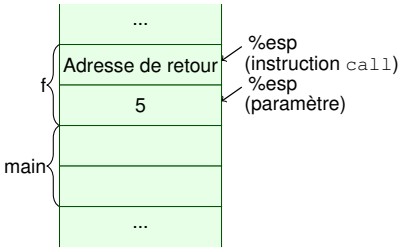
```
f:  pushl %ebp
    movl %esp, %ebp
    subl $8, %esp

    // Corps de f
    // ...

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

⇔

```
f:  enter $8, $0

    // Corps de f
    // ...

    leave

    ret
```


Paramètres passés sur la pile : l'idée

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

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

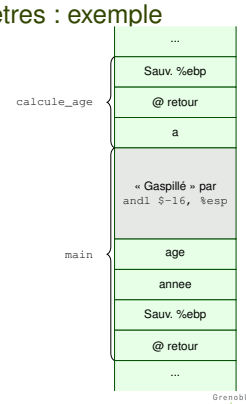


Paramètres passés sur la pile : comment ?

- Empilement des paramètres avec `push` :
  - Faisable sur certains systèmes
  - Mais une règle impose : « %esp doit être multiple de 16 avant d'appeler `call` »
- Solution retenue : adressage relatif à %esp
  - On fait de la place : `subl $..., %esp`
  - On s'assure de `%esp` est multiple de 16 : `andl $-16, %esp`
  - On empile les paramètres (on suppose des paramètres de taille 4 ici) :
    - `movl` *param1*`, 0(%esp)`
    - `movl` *param2*`, 4(%esp)`
    - `movl` *param3*`, 8(%esp)`
    - ...

Passage de paramètres : exemple

```
unsigned calcule_age(unsigned a) {
  return 2010 - 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;
}
```

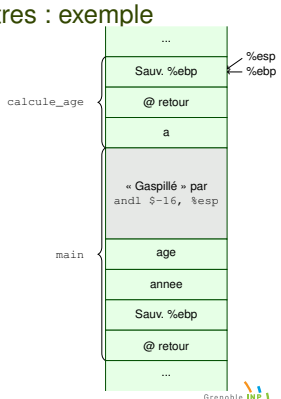


Passage de paramètres : exemple

```
unsigned calcule_age(unsigned a) {
  return 2010 - a;
}
```

⇓

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

    movl $2010, %eax
    subl 8(%ebp), %eax

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

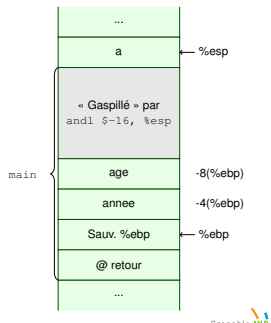


Passage de paramètres : exemple

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

⇓

```
main: pushl %ebp
    movl %esp, %ebp
    // 2 variables locales => 8
    // 2 parametres max => 8
    subl $(8+8), %esp
    // Rendre %esp multiple de 16
    andl $-16, %esp
    // ...
    // age = calcule_age(annee)
    movl -4(%ebp), %eax
    movl %eax, 0(%esp)
    call calcule_age
    // Valeur de retour
    // dans %eax
    movl %eax, -8(%ebp)
    // ...
    leave
    ret
```

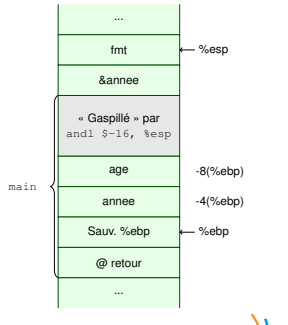


Passage de paramètres par adresse

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

⇓

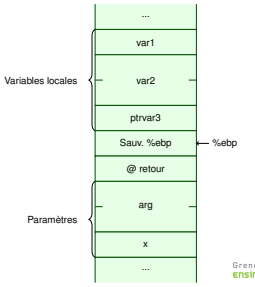
```
main: pushl %ebp
    movl %esp, %ebp
    // 2 variables locales => 8
    // 2 parametres max => 8
    subl $(8+8), %esp
    // Rendre %esp multiple de 16
    andl $-16, %esp
    // ...
    // scanf("%u", &annee);
    movl $fmt_u, 0(%esp)
    movl %ebp, %eax // ou bien :
    addl $-4, %eax // leal -4(%ebp),%eax
     movl %eax, 4(%esp)
    call scanf
    // ...
    leave
    ret
```



Parametres de taille $\neq$ 32 bits

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

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

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



Demo ...

GDB sur le code de `6-age.S` :

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

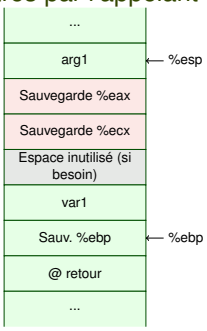
- Sauvegarde dans la pile avant l'appel :

```
movl %eax, 4(%esp)
movl %ecx, 8(%esp)
```

⚠ Il faut compter cette place dans le `subl $..., %esp` (ou `enter $..., $0`) en début de fonction

- Restauration après l'appel :

```
movl 4(%esp), %eax
movl 8(%esp), %ecx
```



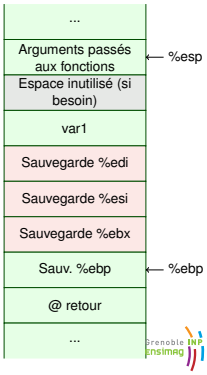
Sauvegarde/restauration des registres par l'appelé

- Sauvegarde dans la pile en début de fonction :

```
f: pushl %ebp
   movl %esp, %ebp
   // Sauvegarde des registres
   pushl %ebx
   pushl %esi
   pushl %edi
   subl $taille, %esp
   // Rendre %esp multiple de 16
   andl $-16, %esp
```

- Restauration en fin de fonction :

```
   // Restauration
   movl -12(%ebp), %edi
   movl -8(%ebp), %esi
   movl -4(%ebp), %ebx

   movl %ebp, %esp
   popl %ebp
   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) ⇒ l'appelé n'est pas tenu de sauvegarder. L'appelant sauvegarde si besoin.
  - Registres « non scratch » ⇒ 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.

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

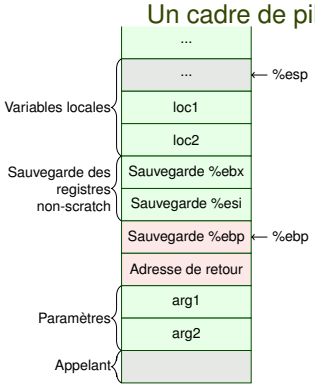
Conventions de liaison Intel 32 bits Unix

- Le format d'un bloc de pile associé à un appel est conforme à ce que nous avons présenté.
- La libération des paramètres est faite par l'appelant et pas par l'appelé
- Les paramètres d'une procédure sont empilés de la droite vers la gauche f(p1, p2, ... , pn) on empile d'abord pn.
- 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.

Un cadre de pile typique



Qui a fait quoi avec la pile ?