



# Assembleur 101

x86/x64

EPITA · APPING2 S7

<b>Auteur(s)</b>	Hugo Sibony
------------------	-------------

<b>Date</b>	24/12/2025
-------------	------------

# Table des matières

1	C'est quoi l'assembleur ? .....	3
1.1	Exemple simple .....	3
1.2	Structure d'un fichier assembleur .....	3
2	Les registres : ta mémoire rapide .....	4
2.1	Registres 64-bit (amd64) .....	4
2.2	Registres 32-bit (i386) .....	4
2.3	Sous-parties des registres .....	5
3	Les deux syntaxes : AT&T vs Intel .....	6
3.1	Tableau comparatif .....	6
3.2	Exemples côte à côte .....	6
3.3	Suffixes de taille (AT&T) .....	7
3.4	Template Intel avec GAS .....	7
4	Boucles et conditions .....	8
4.1	Les flags du CPU .....	8
4.2	Mettre un registre à zéro .....	8
4.3	Comparer deux valeurs : <code>cmp</code> .....	8
4.4	Tester si un registre est zéro : <code>test</code> .....	8
4.5	Sauts conditionnels .....	9
4.6	Exemple : condition simple .....	9
4.7	Exemple : boucle while .....	9
4.8	Exemple : parcourir une chaîne .....	10
4.9	Exemple : boucle for avec tableau .....	11
4.10	Astuce : <code>lea</code> pour l'arithmétique .....	11
5	La division .....	12
5.1	Division 64-bit .....	12
5.2	Division signée .....	12
6	La pile : comment ça marche .....	13
6.1	Opérations de base .....	13
6.2	Visualisation de la pile après un <code>CALL</code> .....	13
7	Stack Frame et alignement .....	14
7.1	Pourquoi 16 octets ? .....	14
7.1.1	C'est quoi SSE ? .....	14
7.1.2	Pourquoi l'alignement ? .....	14
7.2	Le mécanisme .....	14
7.3	Comment réaligner .....	14
7.3.1	Solution 1 : un seul push .....	15
7.3.2	Solution 2 : <code>sub direct</code> .....	15
7.3.3	Solution 3 : frame pointer .....	15
7.4	Résumé visuel .....	15
7.5	Stack frame complet .....	16
7.6	Erreur classique .....	16
8	L'ABI : le contrat entre fonctions .....	17
8.1	ABI i386 (32-bit) .....	17

8.2	ABI amd64 (64-bit) .....	17
8.3	Sauvegarder les registres : quand et comment ? .....	17
9	Appeler des fonctions C depuis l'assembleur .....	18
9.1	Recette .....	18
9.2	Alignement de la pile .....	18
9.3	@PLT et @GOTPCREL .....	18
9.4	lea vs mov .....	19
9.5	Exemple : appeler puts .....	19
9.6	printf avec plusieurs arguments .....	19
9.7	Fonctions fichiers de la libc .....	19
10	Les syscalls : parler au noyau Linux .....	20
10.1	Différence avec les fonctions C .....	20
10.2	Structure d'un syscall .....	20
10.3	Numéros et arguments .....	20
10.4	File descriptors standards .....	21
10.5	Flags pour open() .....	21
10.6	Flags pour mmap() .....	21
10.7	mmap() pour allouer de la mémoire .....	22
10.8	Gestion des erreurs .....	22
11	Debugger avec GDB .....	23
11.1	Lancer GDB .....	23
11.2	Exemple de session .....	23
12	Aide-mémoire .....	24
12.1	Conventions d'appel .....	24
12.2	Callee-saved (à sauvegarder) .....	24
12.3	Syscalls .....	24
12.4	Templates .....	24

# 1 C'est quoi l'assembleur ?

L'assembleur est le langage le plus proche du processeur. Chaque instruction correspond directement à une opération du CPU.

## 1.1 Exemple simple

En C :

```
int x = 5;
x = x + 3;
```

En assembleur, on manipule directement les registres du CPU :

```
movl $5, %eax    # Met 5 dans le registre EAX
addl $3, %eax    # Ajoute 3 à EAX → EAX contient maintenant 8
```

## 1.2 Structure d'un fichier assembleur

```
.section .data          # Section des données (variables globales)
message: .asciz "Hello" # Une chaîne de caractères

.section .text          # Section du code
.globl ma_fonction      # Rend le symbole visible pour le linker

ma_fonction:            # Étiquette = début de la fonction
    # ton code ici
    ret                # Retour à l'appelant
```

### ★ Directives importantes

Directive	Rôle
.section .text	Code exécutable
.section .data	Données initialisées (modifiables)
.section .rodata	Données en lecture seule (constantes, chaînes)
.globl nom	Exporte le symbole (obligatoire pour que C trouve ta fonction !)
.ascii "..."	Chaîne sans <code>\0</code> final
.asciz "..."	Chaîne avec <code>\0</code> final
.long valeur	Valeur 32-bit
.quad valeur	Valeur 64-bit

## 2 Les registres : ta mémoire rapide

Les registres sont des « variables » directement dans le CPU. Ils sont ultra-rapides.

### 2.1 Registres 64-bit (amd64)

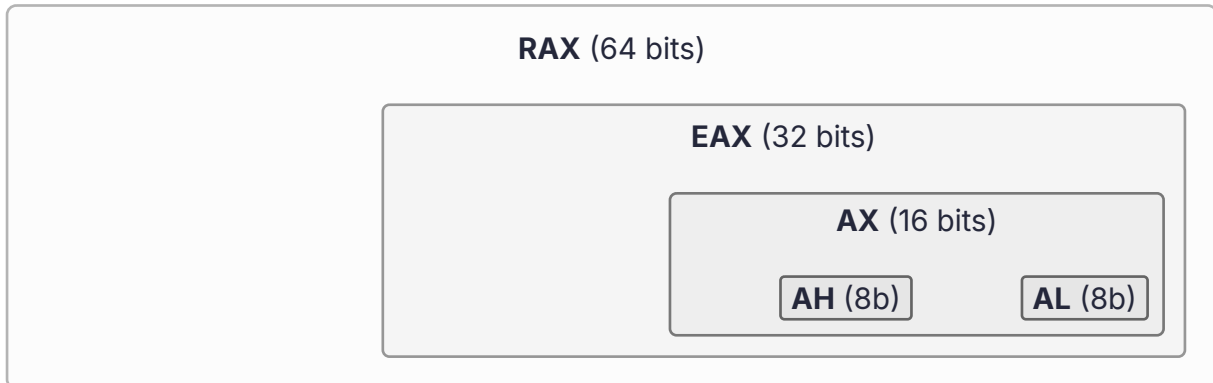
Registre	Rôle
RAX	Valeur de retour des fonctions
RBX	Usage général (À SAUVEGARDER !)
RCX	4ème argument (historiquement : compteur pour les boucles en i386)
RDX	3ème argument / utilisé par les divisions
RSI	2ème argument
RDI	1er argument
RSP	Pointeur de pile (NE JAMAIS PERDRE !)
RBP	Pointeur de base (À SAUVEGARDER !)
R8	5ème argument
R9	6ème argument
R10-R11	Temporaires
R12-R15	Usage général (À SAUVEGARDER !)

### 2.2 Registres 32-bit (i386)

Registre	Rôle
EAX	Valeur de retour
EBX	Usage général (À SAUVEGARDER !)
ECX	Compteur (pour les boucles)
EDX	Données / utilisé par les divisions
ESI	Source (À SAUVEGARDER !)
EDI	Destination (À SAUVEGARDER !)
ESP	Pointeur de pile
EBP	Pointeur de base (À SAUVEGARDER !)

## 2.3 Sous-parties des registres

Un registre 64-bit peut être accédé par morceaux :



### ! Important

**ATTENTION :** Écrire dans EAX met les 32 bits hauts de RAX à zéro automatiquement !

```
movq $0xFFFFFFFFFFFFFFFF, %rax # RAX = 0xFFFFFFFFFFFFFFFF
movl $1, %eax                  # RAX = 0x0000000000000001 (pas 0xFFFFFFFF00000001!)
```

## 3 Les deux syntaxes : AT&T vs Intel

Tu dois savoir écrire dans les deux syntaxes. Voici les différences :

### 3.1 Tableau comparatif

Aspect	AT&T	Intel
Ordre	source, destination	destination, source
Registres	<code>%rax</code>	<code>rax</code>
Immédiats	<code>\$42</code>	<code>42</code>
Mémoire	<code>8(%rsp)</code>	<code>[rsp + 8]</code>
Taille	Suffixe : <code>movl</code> , <code>movq</code>	Mot-clé : <code>DWORD</code> , <code>QWORD</code>

### 3.2 Exemples côte à côte

Mettre une valeur dans un registre :

AT&T

```
movl $42, %eax
movq $100, %rax
```

Intel

```
mov eax, 42
mov rax, 100
```

Copier un registre dans un autre :

AT&T

```
movq %rdi, %rax
```

Intel

```
mov rax, rdi
```

Lire depuis la mémoire (pile) :

AT&T

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

Intel

```
mov eax, [esp + 4]
```

Écrire en mémoire :

AT&T

```
movq %rax, (%rdi)
movq %rax, 8(%rdi)
```

Intel

```
mov [rdi], rax
mov [rdi + 8], rax
```

### 3.3 Suffixes de taille (AT&T)

Suffixe	Taille	Registre exemple
b	1 octet (byte)	AL , BL , CL , R8B
w	2 octets (word)	AX , BX , CX , R8W
l	4 octets (long/dword)	EAX , EBX , ECX , R8D
q	8 octets (quad)	RAX , RBX , RCX , R8

### 3.4 Template Intel avec GAS

Pour utiliser la syntaxe Intel avec l'assembleur GNU (GAS) :

```
.intel_syntax noprefix    # Active Intel, désactive les préfixes % et $

.text
.globl ma_fonction
ma_fonction:
    # ton code en syntaxe Intel ici
    ret
```



## 4 Boucles et conditions

### 4.1 Les flags du CPU

Le CPU possède des « flags » (drapeaux) mis à jour par certaines instructions :

- **ZF** (Zero Flag) : mis à 1 si le résultat est zéro
- **SF** (Sign Flag) : mis à 1 si le résultat est négatif

Les instructions de saut ( `jz` , `jnz` , etc.) lisent ces flags pour décider si elles sautent.

### 4.2 Mettre un registre à zéro

```
xorq %rax, %rax      # RAX = RAX XOR RAX = 0
```

#### ★ Astuce

Pourquoi `xor` et pas `movq $0, %rax` ? Plus court et plus rapide.

### 4.3 Comparer deux valeurs : `cmp`

```
cmpq %rsi, %rdi      # Calcule RDI - RSI, met les flags, jette le résultat
```

Après `cmpq %rsi, %rdi` :

- Si `RDI == RSI` → `ZF = 1`
- Si `RDI < RSI` → `SF = 1` (pour nombres signés)

### 4.4 Tester si un registre est zéro : `test`

```
testq %rdi, %rdi     # Calcule RDI AND RDI, met les flags, jette le résultat
```

`RDI AND RDI` = `RDI`. Donc :

- Si `RDI == 0` → résultat = 0 → **ZF = 1**
- Si `RDI != 0` → résultat != 0 → **ZF = 0**

C'est la façon standard de tester si un registre vaut zéro.

## 4.5 Sauts conditionnels

Instruction	Condition (après <code>cmp b, a</code> ou <code>test</code> )
<code>jz</code> / <code>jz</code>	<code>a == b</code> (ou <code>résultat == 0</code> )
<code>jne</code> / <code>jnz</code>	<code>a != b</code> (ou <code>résultat != 0</code> )
<code>jl</code> / <code>jg</code>	<code>a &lt; b</code> / <code>a &gt; b</code> (signé)
<code>jle</code> / <code>jge</code>	<code>a &lt;= b</code> / <code>a &gt;= b</code> (signé)
<code>jb</code> / <code>ja</code>	<code>a &lt; b</code> / <code>a &gt; b</code> (non-signé)
<code>jbe</code> / <code>jae</code>	<code>a &lt;= b</code> / <code>a &gt;= b</code> (non-signé)
<code>js</code>	résultat négatif
<code>jmp</code>	toujours

### ★ Astuce

**Signé vs non-signé :** `jl` / `jg` pour int, `jb` / `ja` pour unsigned

## 4.6 Exemple : condition simple

En C :

```
if (x == 0) return -1;
return x;
```

En assembleur :

```
testq %rdi, %rdi      # RDI AND RDI → ZF=1 si RDI==0
jnz not_zero          # saute si ZF=0 (donc si RDI ≠ 0)
movq $-1, %rax
ret
not_zero:
movq %rdi, %rax
ret
```

## 4.7 Exemple : boucle while

En C :

```
int sum = 0;
while (n > 0) { sum += n; n--; }
return sum;
```

En assembleur :

```

    xorq %rax, %rax          # sum = 0 (xor avec soi-même = 0)
loop:
    testq %rdi, %rdi         # n == 0 ?
    jz done                  # si ZF=1 (n==0), termine
    addq %rdi, %rax          # sum += n
    decq %rdi                # n--
    jmp loop
done:
    ret

```

## 4.8 Exemple : parcourir une chaîne

En C :

```

int len = 0;
while (*s != '\0') { len++; s++; }
return len;

```

En assembleur :

```

    xorq %rax, %rax          # len = 0
loop:
    movzbl (%rdi), %ecx       # charge 1 octet à l'adresse RDI dans ECX
    testb %cl, %cl           # octet == 0 ?
    jz done
    incq %rax                 # len++
    incq %rdi                 # s++ (passe au caractère suivant)
    jmp loop
done:
    ret

```

### i Information

`movzbl (%rdi), %ecx` = « Move Zero-extend Byte to Long »

- Lit 1 octet à l'adresse RDI
- L'étend à 32 bits en ajoutant des zéros (zero-extend)
- Le met dans ECX

**Équivalent Intel :** `movzx ecx, BYTE PTR [rdi]`

## 4.9 Exemple : boucle for avec tableau

En C :

```
int sum = 0;
for (int i = 0; i < len; i++) { sum += arr[i]; }
return sum;
```

En assembleur :

```
xorl %eax, %eax      # sum = 0
xorl %ecx, %ecx      # i = 0
loop:
  cmpl %esi, %ecx     # compare i avec len
  jge done           # si i ≥ len, termine
  movl (%rdi,%rcx,4), %edx # edx = arr[i]
  addl %edx, %eax     # sum += arr[i]
  incl %ecx           # i++
  jmp loop
done:
  ret
```

### ★ Adressage indexé

$(\%rdi, \%rcx, 4) = \text{adresse } RDI + RCX * 4$

- RDI = adresse de base du tableau
- RCX = index
- 4 = taille d'un élément (4 octets pour `int`)

## 4.10 Astuce : lea pour l'arithmétique

`lea` calcule une adresse sans accéder à la mémoire :

AT&T

```
leaq (%rdi,%rsi), %rax # addition
leaq (%rdi,%rdi,4), %rax # * 5
leaq 1(%rdi), %rax     # + 1
```

Intel

```
lea rax, [rdi + rsi]
lea rax, [rdi + rdi*4]
lea rax, [rdi + 1]
```

### ★ Multiplication par 10 (utile pour atoi)

`result = result * 10 + digit`

```
leaq (%rax,%rax,4), %rax # rax *= 5
leaq (%rdx,%rax,2), %rax # rax = digit + rax*2 (donc digit + ancien*10)
```

## 5 La division

La division en assembleur est un peu particulière car elle utilise une paire de registres.

### 5.1 Division 64-bit

```
xorq %rdx, %rdx      # OBLIGATOIRE! Met RDX à 0
divq %rbx             # Divise RDX:RAX par RBX
                     # RAX = quotient
                     # RDX = reste
```

#### ! Important

`xorq %rdx, %rdx` est **obligatoire** avant `divq`. L'instruction divise le nombre 128-bit formé par RDX:RAX. Si RDX contient des valeurs aléatoires, le résultat sera faux.

### 5.2 Division signée

Pour les nombres signés, utilise `idivq` et `cqto` (sign-extend RAX into RDX:RAX) :

```
cqto                 # Étend le signe de RAX dans RDX
idivq %rbx           # Division signée
```

## 6 La pile : comment ça marche

La pile est une zone mémoire qui **grandit vers le bas** (adresses décroissantes).

### 6.1 Opérations de base

```
# PUSH: Empile une valeur
pushq %rax
# Équivalent à:
# 1. RSP = RSP - 8      (la pile grandit vers le bas)
# 2. Écrit RAX à l'adresse RSP

# POP: Dépile une valeur
popq %rax
# Équivalent à:
# 1. Lit la valeur à l'adresse RSP dans RAX
# 2. RSP = RSP + 8      (la pile rétrécit)
```

### 6.2 Visualisation de la pile après un CALL

Quand le C appelle ta fonction `add(3, 4)` en 32-bit :

```
Adresses hautes
|
v
+-----+
|  4    | ← ESP + 8  (2ème argument)
+-----+
|  3    | ← ESP + 4  (1er argument)
+-----+
| RetAddr | ← ESP      (adresse de retour, poussée par CALL)
+-----+
^
|
Adresses basses
```

#### ! Important

**C'est pour ça que le 1er argument est à ESP+4 et pas ESP+0 !** L'instruction CALL pousse automatiquement l'adresse de retour.

# 7 Stack Frame et alignement

## 7.1 Pourquoi 16 octets ?

L'ABI exige : `RSP % 16 == 0` avant chaque `call`.

### 7.1.1 C'est quoi SSE ?

SSE (Streaming SIMD Extensions) = instructions qui traitent **plusieurs données en parallèle**.

Le CPU a des registres spéciaux de 128 bits (16 octets) : `XMM0` à `XMM15`.

```
# Registres classiques : 1 opération à la fois
addq %rax, %rbx      # 1 addition 64-bit

# Registres SSE : 4 opérations en parallèle
addps %xmm0, %xmm1   # 4 additions 32-bit en même temps
```

La libc utilise SSE pour optimiser `memcpy`, `strlen`, `printf`, etc. C'est plus rapide.

### 7.1.2 Pourquoi l'alignement ?

Les instructions SSE qui accèdent à la mémoire (`movaps`, `movdqa`) exigent une adresse divisible par 16. C'est une contrainte hardware : le bus mémoire charge 16 octets alignés plus efficacement.

```
movaps (%rsp), %xmm0  # Charge 16 octets depuis RSP dans XMM0
                     # Si RSP % 16 != 0 → SIGSEGV immédiat
```

Si ta pile est mal alignée quand tu appelles `printf` → `printf` utilise `movaps` → crash.

## 7.2 Le mécanisme

Suivons RSP pas à pas avec des vraies valeurs :

1. Quelqu'un appelle ta fonction  
`RSP = 0x7fff0010` (multiple de 16 ✓)
2. L'instruction `CALL` s'exécute
  - Pousse l'adresse de retour (8 octets)
  - `RSP = 0x7fff0010 - 8 = 0x7fff0008`
3. Tu entres dans ta fonction  
`RSP = 0x7fff0008 → 0x7fff0008 % 16 = 8 × DÉALIGNÉ`

**Problème :** À l'entrée de ta fonction, RSP est TOUJOURS désaligné de 8.

## 7.3 Comment réaligner

Tu dois soustraire 8 octets (ou un nombre impair de fois 8) pour revenir à un multiple de 16.

### 7.3.1 Solution 1 : un seul push

```
func:
    pushq %rbx          # RSP -= 8
    # RSP était 0x7fff0008, maintenant 0x7fff0000 ✓

    call printf@PLT     # OK, RSP aligné

    popq %rbx
    ret
```

Calcul :  $0x7fff0008 - 8 = 0x7fff0000 \rightarrow 0x7fff0000 \% 16 = 0 \checkmark$

### 7.3.2 Solution 2 : sub direct

```
func:
    subq $8, %rsp       # Même effet qu'un push

    call printf@PLT

    addq $8, %rsp
    ret
```

### 7.3.3 Solution 3 : frame pointer

```
func:
    pushq %rbp          # RSP -= 8 → aligné
    movq %rsp, %rbp
    subq $32, %rsp      # 32 octets de variables locales (multiple de 16)

    # Variables : -8(%rbp), -16(%rbp), -24(%rbp), -32(%rbp)

    leave               # RSP = RBP, pop RBP
    ret
```

Pourquoi `sub $32` et pas `sub $24` si j'ai 3 variables de 8 octets ?

Après `push %rbp` :  $RSP = 0x7fff0000$  (aligné)  
 Après `sub $24` :  $RSP = 0x7fff0000 - 24 = 0x7ffefff8$   
 $0x7ffefff8 \% 16 = 8 \times$

Après `sub $32` :  $RSP = 0x7fff0000 - 32 = 0x7ffefff0$   
 $0x7ffefff0 \% 16 = 0 \checkmark$

**Règle :** après le `push %rbp`, alloue toujours un multiple de 16.

## 7.4 Résumé visuel

	RSP % 16	
Avant call	0	✓ aligné
Après call (entrée)	8	× désaligné
Après 1 push	0	✓ aligné
Après 2 push	8	× désaligné
Après 3 push	0	✓ aligné

**Pattern :** nombre impair de push → aligné. Nombre pair → désaligné.



## 7.5 Stack frame complet

+24	argument 7	(si > 6 args)
+16	argument 8	
+8	adresse retour	← poussée par CALL
0	ancien RBP	← RBP pointe ici
-8	variable 1	
-16	variable 2	← RSP après sub \$16

Accès : `movq -8(%rbp), %rax` pour lire variable 1.

## 7.6 Erreur classique

```
# FAUX - va segfault
func:
    call printf@PLT    # RSP = 8 mod 16 → CRASH
    ret

# CORRECT
func:
    pushq %rbx
    call printf@PLT    # RSP = 0 mod 16 → OK
    popq %rbx
    ret
```

### ! Important

**Debug :** Dans GDB, avant un call suspect : `p $rsp % 16`

Si ça affiche 8 → ta pile est mal alignée.

## 8 L'ABI : le contrat entre fonctions

L'ABI (Application Binary Interface) définit les règles pour que les fonctions puissent s'appeler correctement.

### 8.1 ABI i386 (32-bit)

Catégorie	Valeur
Arguments	Sur la pile : [ESP+4] , [ESP+8] , [ESP+12] ...
Retour	EAX
Libres	EAX, ECX, EDX
À sauvegarder	EBX, ESI, EDI, EBP

### 8.2 ABI amd64 (64-bit)

Catégorie	Valeur
Arguments	RDI, RSI, RDX, RCX, R8, R9 (dans cet ordre)
Retour	RAX
Libres	RAX, RCX, RDX, RSI, RDI, R8-R11
À sauvegarder	RBX, RBP, R12-R15
Pile	RSP multiple de 16 avant <code>call</code>

### 8.3 Sauvegarder les registres : quand et comment ?

Si ta fonction utilise un registre « callee-saved » (comme RBX), tu DOIS :

1. Le sauvegarder au début (avec `push`)
2. Le restaurer à la fin (avec `pop`)

```
ma_fonction:
    pushq %rbx           # Sauvegarde RBX

    # ... tu peux utiliser RBX ici ...

    popq %rbx           # Restaure RBX
    ret
```

Si tu utilises uniquement des registres « caller-saved » (RAX, RDI, RSI...), pas besoin de les sauvegarder.

## 9 Appeler des fonctions C depuis l'assembleur

### 9.1 Recette

1. Mettre les arguments dans RDI, RSI, RDX, RCX, R8, R9
2. Aligner la pile (RSP doit être multiple de 16 avant `call` )
3. `call fonction@PLT` → résultat dans RAX

### 9.2 Alignement de la pile

À l'entrée de ta fonction,  $RSP = 8 \bmod 16$  (l'adresse de retour a été pushée).

**Solution simple :** `pushq %rbx` avant le `call` (ajoute 8 octets → RSP multiple de 16).

**Solution avec frame pointer (recommandée pour fonctions complexes) :**

**AT&T :**

```
ma_fonction:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp

    # ... ton code ...
    # Variables locales: -8(%rbp), -16(%rbp)

    leave
    ret
```

**Intel :**

```
ma_fonction:
    push rbp
    mov rbp, rsp
    sub rsp, 16

    ; ... ton code ...
    ; Variables locales: [rbp-8], [rbp-16]

    leave
    ret
```

#### ★ Astuce

`leave` = restaure RSP depuis RBP puis dépile RBP. Pratique pour nettoyer le stack frame.

### 9.3 @PLT et @GOTPCREL

Quand tu appelles des fonctions de la libc ou accèdes à des variables globales externes :

```
# Appeler une fonction externe (printf, puts, getline, etc.)
call printf@PLT          # @PLT = Procedure Linkage Table

# Accéder à une variable globale externe (stdin, stdout, stderr)
movq stdin@GOTPCREL(%rip), %rdx # Charge l'ADRESSE de stdin depuis la GOT
movq (%rdx), %rdx             # Charge la VALEUR de stdin (le FILE*)
```

#### i Information

**Pourquoi @PLT ?** Le linker dynamique résout l'adresse de la fonction au runtime.

**Pourquoi @GOTPCREL ?** Les variables globales externes sont dans la Global Offset Table.

## 9.4 lea vs mov

```
lea msg(%rip), %rdi    # RDI = ADRESSE de msg (pour passer une chaîne)
movq var(%rip), %rdi   # RDI = CONTENU de var (pour lire une variable)
```

## 9.5 Exemple : appeler puts

```
.section .rodata
msg: .asciz "Hello"

.section .text
.globl exemple
exemple:
    pushq %rbx           # Aligne la pile
    leaq msg(%rip), %rdi  # arg1 = adresse de msg
    call puts@PLT
    popq %rbx
    ret
```

## 9.6 printf avec plusieurs arguments

```
leaq format(%rip), %rdi    # arg1: format string
movq %rcx, %rsi            # arg2: valeur à afficher
xorl %eax, %eax            # RAX = 0 (pas d'arguments flottants)
call printf@PLT
```

### ! Important

`xorl %eax, %eax` est obligatoire avant printf/scanf (fonctions variadiques).

## 9.7 Fonctions fichiers de la libc

Fonctions pour manipuler des fichiers :

Fonction	Prototype	Description
<code>fopen</code>	<code>FILE *fopen(char *path, char *mode)</code>	Ouvre un fichier, retourne NULL si erreur
<code>fclose</code>	<code>int fclose(FILE *stream)</code>	Ferme un fichier
<code>fgetc</code>	<code>int fgetc(FILE *stream)</code>	Lit un caractère, retourne -1 (EOF) si fin
<code>fputc</code>	<code>int fputc(int c, FILE *stream)</code>	Écrit un caractère
<code>putchar</code>	<code>int putchar(int c)</code>	Écrit un caractère sur stdout

### ★ Astuce

**Modes fopen :** "r" = lecture, "w" = écriture (crée/écrase)

# 10 Les syscalls : parler au noyau Linux

## 10.1 Différence avec les fonctions C

### ! Important

**ATTENTION** : Pour les syscalls, le 4ème argument est dans **R10**, pas RCX !  
(L'instruction `syscall` utilise RCX en interne pour sauvegarder RIP)

	Fonctions C	Syscalls
4ème arg	RCX	<b>R10</b>
Appel	<code>call func</code>	<code>syscall</code>
Erreur	Variable	RAX négatif

## 10.2 Structure d'un syscall

```

movq $NUMERO, %rax    # Numéro du syscall
movq arg1, %rdi        # 1er argument
movq arg2, %rsi        # 2ème argument
movq arg3, %rdx        # 3ème argument
movq arg4, %r10        # 4ème argument (ATTENTION: R10, pas RCX!)
movq arg5, %r8         # 5ème argument
movq arg6, %r9         # 6ème argument
syscall                # Exécute le syscall
# Résultat dans RAX (négatif = erreur)

```

## 10.3 Numéros et arguments

#	Nom	Arguments
0	read	fd, buffer, count → nb octets lus
1	write	fd, buffer, count → nb octets écrits
2	open	path, flags, mode → fd
3	close	fd
9	mmap	addr, len, prot, flags, fd, offset → ptr

## 10.4 File descriptors standards

Nom	Valeur	Description
stdin	0	entrée standard
stdout	1	sortie standard
stderr	2	erreur standard

## 10.5 Flags pour open()

Constante	Valeur	Description
O_RDONLY	0	Lecture seule
O_WRONLY	1	Écriture seule
O_RDWR	2	Lecture et écriture
O_CREAT	0x40	Créer si n'existe pas
O_TRUNC	0x200	Tronquer si existe

### ★ Astuce

Pour ouvrir en écriture avec création : `O_WRONLY | O_CREAT | O_TRUNC = 0x241`

## 10.6 Flags pour mmap()

Constante	Valeur
PROT_READ	1
PROT_WRITE	2
MAP_PRIVATE	0x02
MAP_ANONYMOUS	0x20

### ★ Astuce

Pour allouer de la mémoire : `prot = 3 , flags = 0x22 , fd = -1 , offset = 0`

## 10.7 mmap() pour allouer de la mémoire

```
movq $9, %rax          # syscall mmap
xorq %rdi, %rdi         # addr = NULL
movq $SIZE, %rsi        # len = taille voulue
movq $3, %rdx           # prot = PROT_READ | PROT_WRITE
movq $0x22, %r10        # flags = MAP_PRIVATE | MAP_ANONYMOUS
movq $-1, %r8           # fd = -1
xorq %r9, %r9           # offset = 0
syscall
# RAX = pointeur vers la mémoire allouée (ou négatif si erreur)
```

## 10.8 Gestion des erreurs

Les syscalls retournent une valeur négative en cas d'erreur :

```
syscall
testq %rax, %rax        # SF=1 si RAX négatif
js erreur              # Jump if Sign (si négatif)
```

# 11 Debugger avec GDB

## 11.1 Lancer GDB

```
gdb ./mon_programme
```

Commande	Description
<code>break add</code>	Breakpoint sur la fonction add
<code>run</code>	Lancer
<code>stepi</code>	1 instruction (entre dans les calls)
<code>nexti</code>	1 instruction (saute les calls)
<code>continue</code>	Continuer
<code>info registers</code>	Voir les registres
<code>p/x \$rax</code>	Afficher RAX en hexa
<code>x/s \$rdi</code>	Afficher la chaîne pointée par RDI
<code>x/10x \$rsp</code>	10 valeurs hexa depuis RSP

## 11.2 Exemple de session

```
$ gdb ./mon_programme
(gdb) break add          ← point d'arrêt sur ta fonction
(gdb) run                ← lance le programme
Breakpoint 1, add ()
(gdb) info registers rdi rsi ← regarde les arguments
rdi                0x3
rsi                0x4
(gdb) stepi             ← exécute 1 instruction
(gdb) info registers rax ← vérifie le résultat
(gdb) continue          ← continue l'exécution
```



## 12 Aide-mémoire

### 12.1 Conventions d'appel

Type	Arguments	Retour
Fonctions C (amd64)	RDI, RSI, RDX, RCX, R8, R9	RAX
Syscalls (amd64)	RDI, RSI, RDX, <b>R10</b> , R8, R9 (numéro dans RAX)	RAX
i386	[ESP+4], [ESP+8], ...	EAX

### 12.2 Callee-saved (à sauvegarder)

Arch	Registres
amd64	RBX, RBP, R12-R15
i386	EBX, ESI, EDI, EBP

### 12.3 Syscalls

#	Nom	Args
0	read	fd, buf, count
1	write	fd, buf, count
2	open	path, flags, mode
3	close	fd
9	mmap	addr, len, prot, flags, fd, off

### 12.4 Templates

#### amd64 AT&T

```
.text
.globl func
func:
    ret
```

#### amd64 Intel

```
.intel_syntax noprefix
.text
.globl func
func:
    ret
```

#### i386

```
.text
.globl func
func:
    ret
```