

# Processus - Exécution

Programmation Système — R3.05

C. Raïevsky



Département Informatique  
**BUT Informatiques 2<sup>ème</sup> année**

# Fonctions d'un Système d'Exploitation

Fonctions essentielles – Utiles pour les programmeurs

## Gestion de l'exécution des programmes

- Lancement
- Basculement entre processus – Ordonnancement
- Adressage mémoire
- Certains aspects de la sécurité

## Rôle de chef d'orchestre

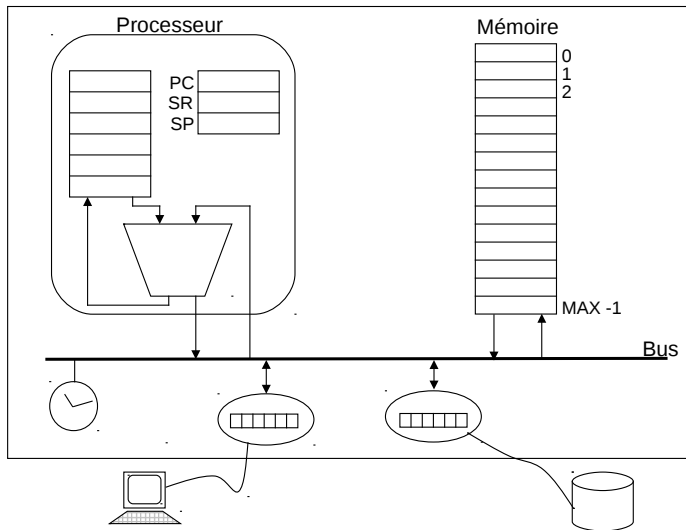
## Interaction avec le monde extérieur - Abstraction du matériel sous forme de ressources

- Système de fichier
- Réseau
- Horloges – Timers
- Pilotes matériels

## Offre au programmeur une interface uniforme et portable

# Architecture générale

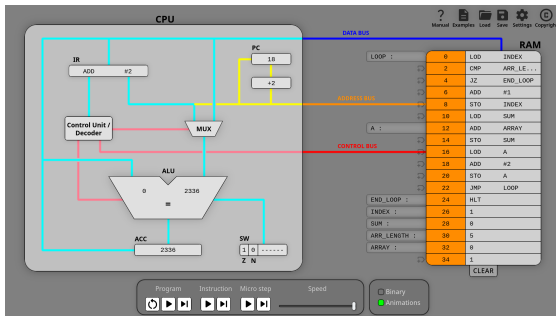
Von Neumann (Princeton)



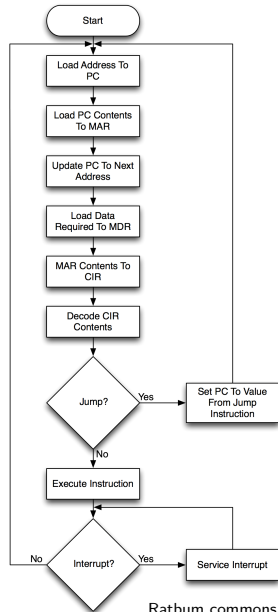
# Vision simpliste de l'exécution

À chaque pas le processeur :

- Récupère l'instruction courante,
- la décode,
- l'exécute.



Pour simuler : CPU Visual Simulator  
Un simple add



Ratbum commons.wikimedia.org

# Activité, contexte

## Notion d'activité

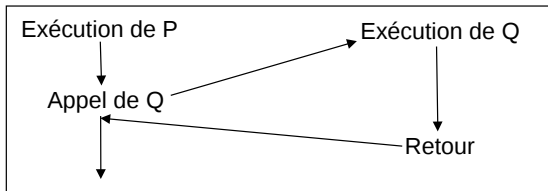
- On nomme **activité** l'exécution d'une procédure (ou fonction) sur un processeur
- Le *contexte* d'une activité est l'ensemble des informations accessibles au processeur au cours de l'exécution de l'activité (registres processeur, mémoire)
- Cadre d'exécution – *Execution frame*

## Commutation d'activité

- Passage d'une activité à une autre
- Sauvegarde du contexte de l'activité courante
- Création ou
- Restauration du contexte de la nouvelle activité

# Appel de fonction

Principe :



Séquence d'appel, par P :

- Préparation des paramètres transmis à Q
- Sauvegarde du contexte de P
- Remplacement du contexte de P par le contexte de Q
- Après retour, récupération des résultats transmis par Q

Retour :

- Préparation des résultats
- Restauration du contexte de P

# Appel de fonction à l'aide de la pile – *The Call Stack*

Pourquoi est-ce important ?

La pile est le support des **appels de fonctions**

→ **Essentiels** en programmation

Le fonctionnement d'un logiciel est basé sur l'appel successif de fonctions.

Compréhension de ce mécanisme essentiel pour :

- le débogage,
- l'analyse de performance
- savoir, d'une manière générale, ce qu'il est en train de se passer

# Stack Frame – Cadre d'exécution

Créé à chaque appel de fonction

## Contient :

- Les variables locales à la fonction
- Les arguments passés à la fonction
- Les informations nécessaire au retour

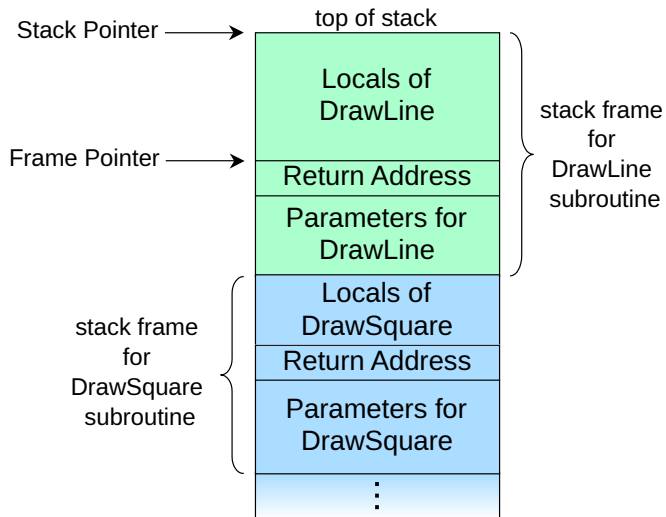
## Le contenu exact varie en fonction de :

- l'architecture du cpu
- la convention d'appel (*function call convention*)



# Stack Frame Concrètement

Dans l'adressage mémoire



## Dans le contexte de la fonction

- Toutes les adresses sont relatives au *Frame Pointer*

## Notamment

- Les variables locales
- Les paramètres

# Exemple d'appel de fonction

*intel x86, C-style function call*

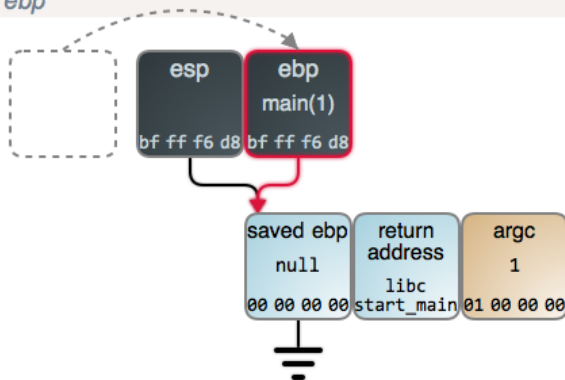
```
1
2 int add(int a, int b){
3     int result = a + b;
4     return result;
5 }
6
7 int main(int argc, char* argv[]){
8
9     int answer;
10    answer = add(40, 2);
11
12    return EXIT_SUCCESS;
13 }
```

# En mémoire

From : [duartes.org](http://duartes.org)'s *journey-to-the-stack*

Au tout début du main, à la ligne 8 :

3. `movl %esp, %ebp # copy esp to ebp`



**ebp** : Base Pointer = Frame Pointer

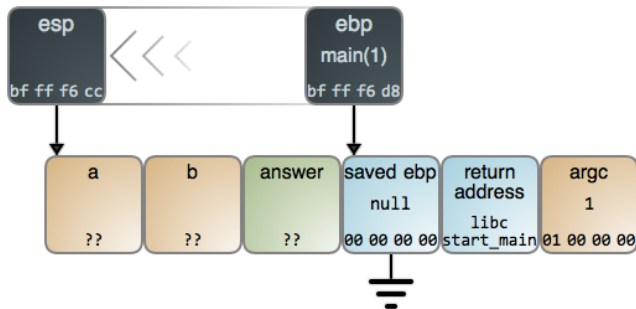
**esp** : Stack Pointer

# En mémoire

On agrandit la pile pour y stocker :

- La variable locale (answer)
- Les paramètres (a et b)

4. `subl $12, %esp # make room for stack data`



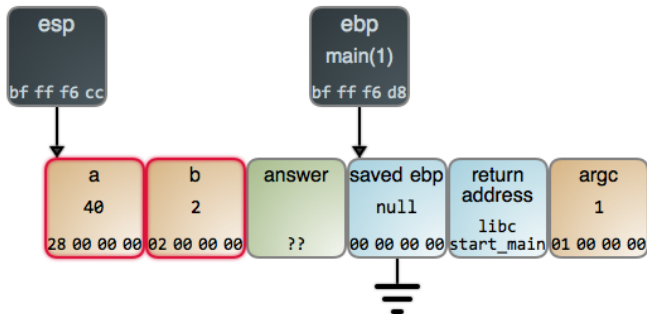
`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)

## En mémoire

On place les valeurs des paramètres sur la pile :

```
5.  movl $2, 4(%esp) # set b to 2
     movl $40, (%esp) # set a to 40
```



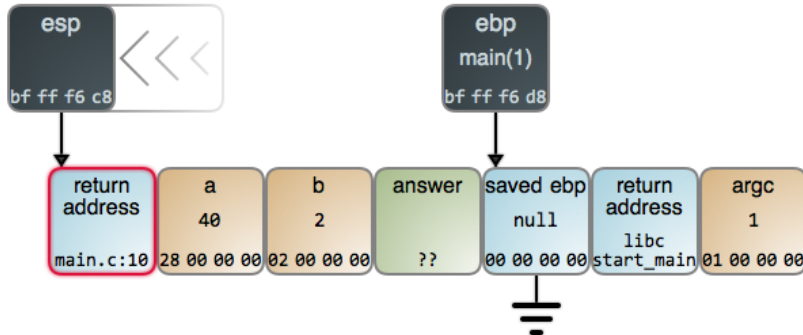
ebp : Base Pointer = Frame Pointer – esp : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)

## En mémoire

L'adresse de retour est empilée :

6. `call add` # push return address onto stack, jump into add



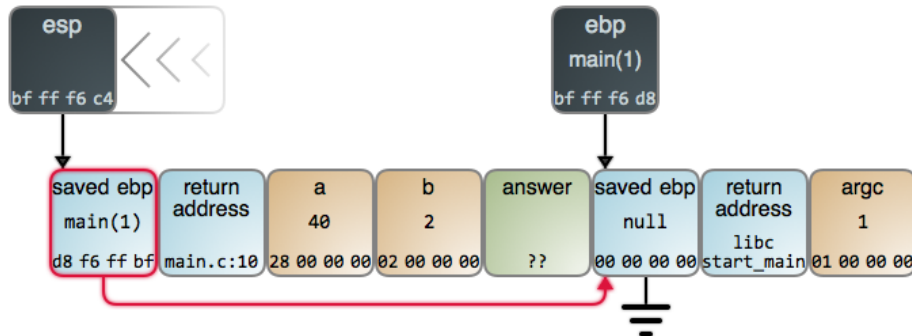
`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](https://duartes.org's-journey-to-the-stack)

## En mémoire

## On sauvegarde le contexte précédent

7. `pushl %ebp # save current ebp register value`



`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

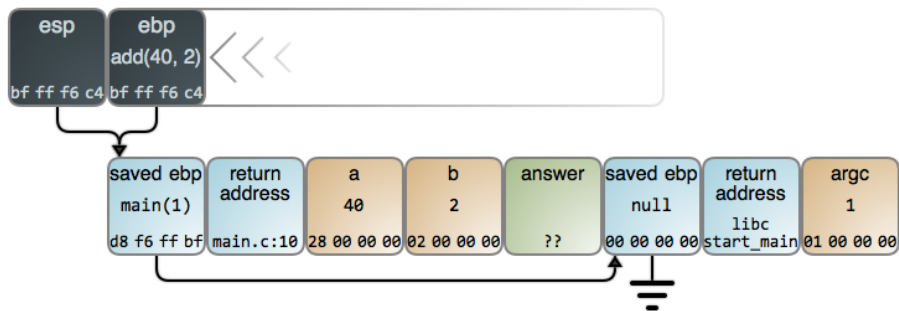
From : [duartes.org's journey-to-the-stack](https://duartes.org's-journey-to-the-stack)

## En mémoire

## Changement de contexte !!

On est maintenant dans le contexte de la fonction add

8. `movl %esp, %ebp # copy esp to ebp`



`ebp` : Base Pointer = Frame Pointer - `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)



# Exemple d'appel de fonction

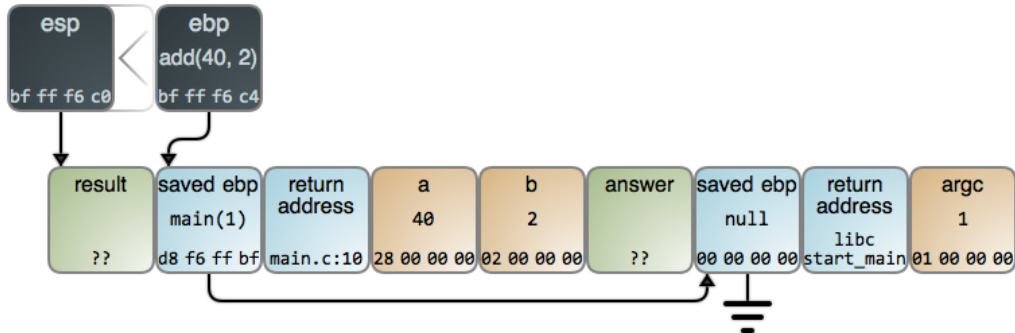
*intel x86, C-style function call*

```
1
2 int add(int a, int b){
3     int result = a + b;
4     return result;
5 }
6
7 int main(int argc, char* argv[]){
8
9     int answer;
10    answer = add(40, 2);
11
12    return EXIT_SUCCESS;
13 }
```

## En mémoire

On fait de la place pour la variable locale `result` (`!= answer`)

9. `subl $4, %esp # make room for result`



`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

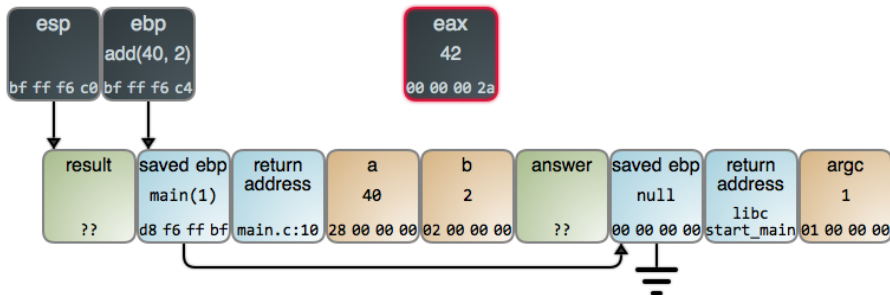
From : [duartes.org's journey-to-the-stack](https://duartes.org's-journey-to-the-stack)

## En mémoire

On effectue l'addition en stockant les paramètres dans des registres

```

movl 12(%ebp), %eax # move b to eax
10. movl 8(%ebp), %edx # move a to edx
    addl %edx, %eax   # add edx into eax. total is 42.
  
```



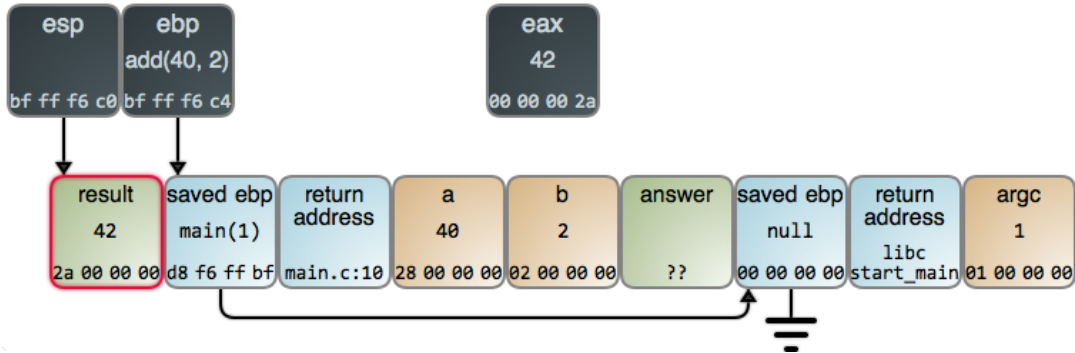
`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)

## En mémoire

On stocke le résultat dans la variable locale `result`

```
11.  movl %eax, -4(%ebp) # copy eax to result
```



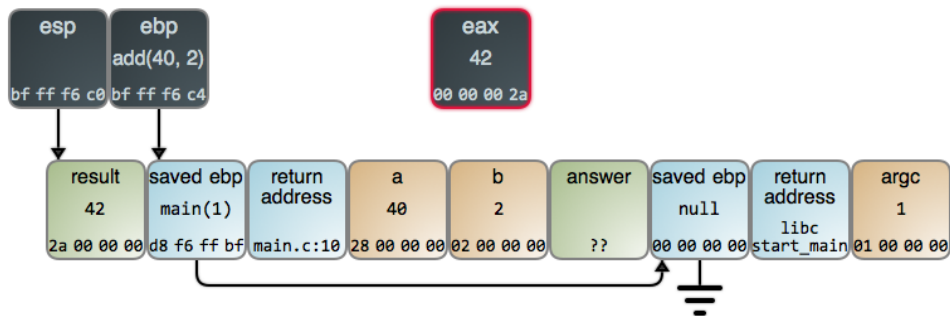
ebp : Base Pointer = Frame Pointer – esp : Stack Pointer

From : [duartes.org's journey-to-the-stack](https://duartes.org's-journey-to-the-stack)

## En mémoire

On stocke le résultat dans le registre `eax`, comme valeur de retour de la fonction

1. `movl -4(%ebp), %eax` # send result as the return value through `eax`



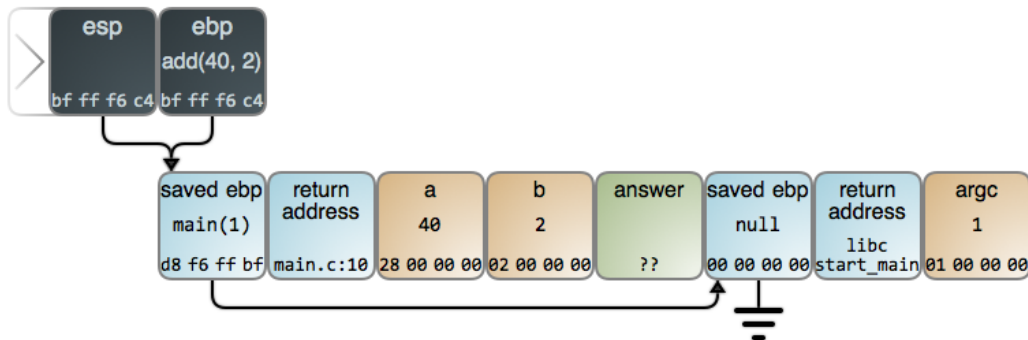
`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)

## En mémoire

On dépile la variable locale `result`

2. `leave` # part 1: copy `ebp` to `esp`



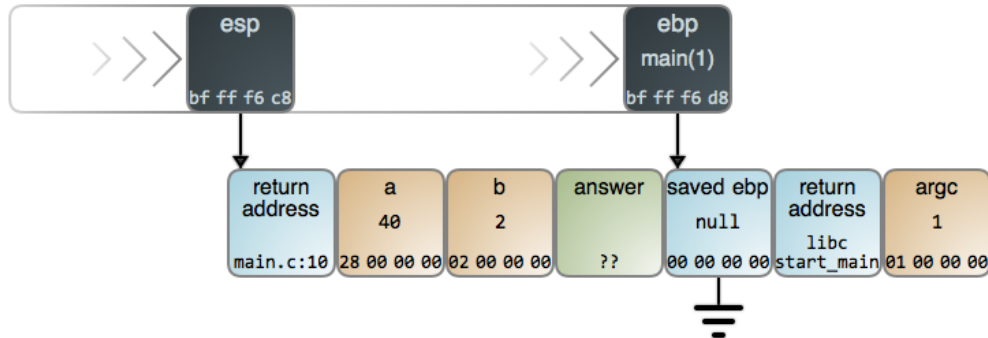
`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](#)

## En mémoire

On restaure le contexte de l'appelant

3. `leave` # part 2: pop into ebp



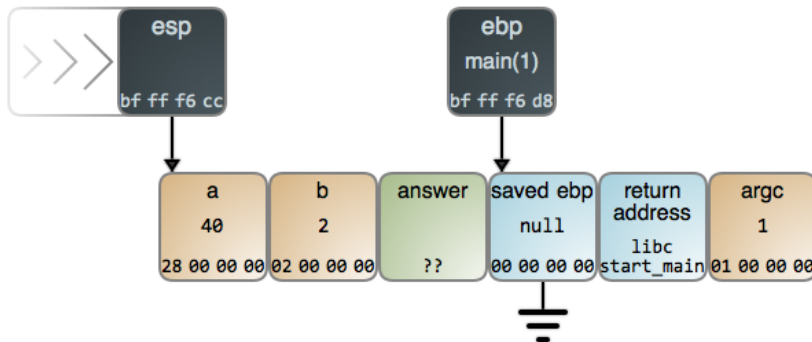
ebp : Base Pointer = Frame Pointer – esp : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)

# En mémoire

On dépile l'adresse de retour, qui contient l'adresse de la prochaine instruction à exécuter après le retour

## 4. `ret` # pop into eip (instruction pointer)

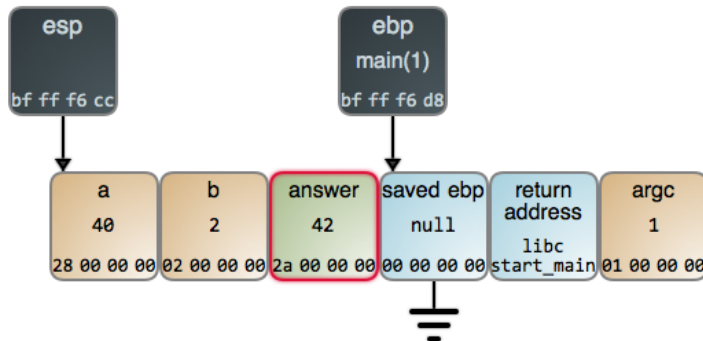




## En mémoire

On copie le résultat de la fonction dans la variable locale `answer`

5. `movl %eax, -4(%ebp) # copy eax to answer`



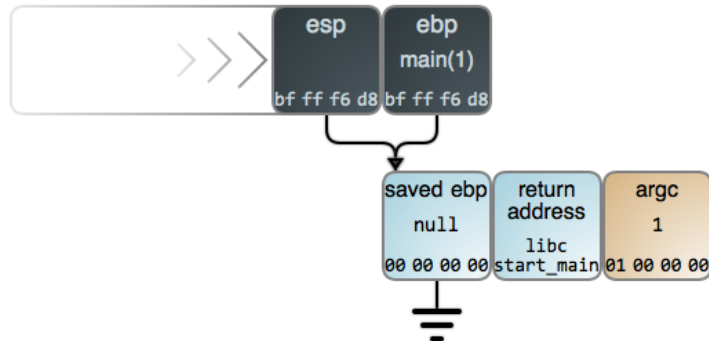
`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](https://duartes.org/journey-to-the-stack)

## En mémoire

On dépile les paramètres et la variable locale avant de sortir du main

6. `leave` # part 1: copy `ebp` to `esp`



`ebp` : Base Pointer = Frame Pointer – `esp` : Stack Pointer

From : [duartes.org's journey-to-the-stack](http://duartes.org's-journey-to-the-stack)

# Convention d'appel, optimisation

On sort du monde des bisounours

## Dans la réalité

- Beaucoup d'optimisations
- Différentes conventions
- Mécanismes de protection

## Par exemple :

- Passage des paramètres de la fonction via des registres CPU
- Remplacement de l'appel par du code "*en ligne*"
- Retour du résultat (potentiellement complexe) de la fonction sur la pile, le tas.