L3 : Système

Gilles Menez

Université de Nice – Sophia-Antipolis
Département d'Informatique
email : menez@unice.fr
www : www.i3s.unice.fr/~menez

September 13, 2024: V 1.0

Le Web et quelques bons livres . . .

J.P. Braquelaire
Méthodologie de la programmation en Langage C - Principes et Applications.
Masson, 1995, 2e édition.

Rifflet J.M
La programmation sous Unix.
Ediscience, 1993, 3e édition.

Rémy Card, Eric Dumas, Franck Mével
Programmation Linux 2.0.
Eyrolles, 1997.

Daniel P.Bovet and Marco Cesati
Understanding the Linux Kernel.
Third edition - O'Reilly, 2006.

Table des Matières (Cours 1)

Table des Matières (Cours 2)

Pourquoi ce cours ? I

La "programmation" ... est un acte complexe qui mobilise plusieurs "intervenants" :

- la machine matérielle,
- le système d'exploitation,
- le langage,
- les usages (et les utilisateurs),
- le code,
- ...

Croire ou oublier que ces "intervenants" n'ont pas d'interactions est une grave erreur qui peut déboucher par exemple sur des dysfonctionnements, ou sur de mauvaises performances ou encore sur des failles de sécurité.

Pourquoi ce cours ? II

Il y a des interactions entre la conception, l'exécution et l'utilisation.

Par exemple :

- La machine matérielle contient des périphériques (RAM, DISK, ...) qui sont bien lents au regard des performances et du coût d'un processeur.

  Pour améliorer ce "fait de la nature" (donc incontournable), un grand nombre de "caches" et de buffers ont été introduit au niveau matériel et logiciel pour factoriser les accés.

  Mais la gestion de ces caches est souvent "asynchrones", et on peut facilement perdre ou dupliquer de l'information sur le flux d' E/S si on n'y prend pas garde.

Pourquoi ce cours ? III

- L'usage des machines a très rapidement évolué vers la multiprogrammation : plusieurs programmes "en même temps".

  Mais comment cela est-il possible avec une architecture de machine mono processeur telle celle de Von Neumann ?

  La réponse réside dans la notion de processus. Mais qu'est ce que cette notion recouvre exactement ? sur les droits du code exécuté, la mémoire, la conception, . . . **d'un programme qui doit partager les ressources de la machine.**
- La programmation a depuis toujours souhaité permettre du "dynamisme" et la possibilité par exemple, avec la récursivité, d'un nombre non restreint d'appels de fonctions.

  Pour réaliser cela, une organisation spécifique de la mémoire a été adoptée. **Cette organisation comporte une pile d'exécution qui pourrait bien mettre en danger la sécurité de vos programmes . . .**

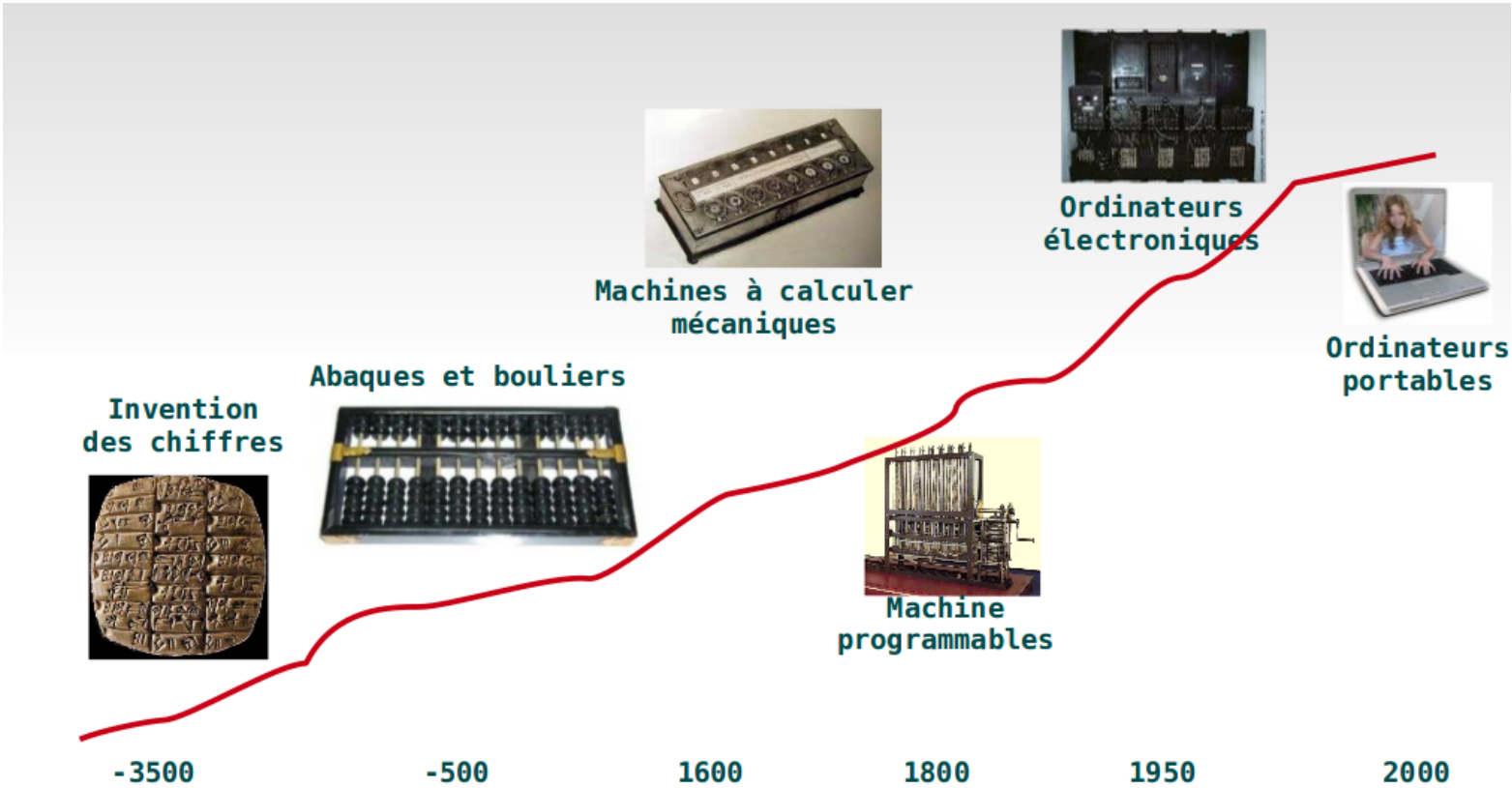
Pourquoi ce cours ? IV

L'objectif du cours de vous apprendre à faire ces liens entre les choses de la programmation et à avoir une vision plus globale que le exercice de programmation que vous produisez.

Du coup, cela ne peut pas être en Java ou en Python ou en . . . "tous ces excellents, mais complexes, langages" qui reposent sur une machine logicielle qui tend à masquer ces interactions.

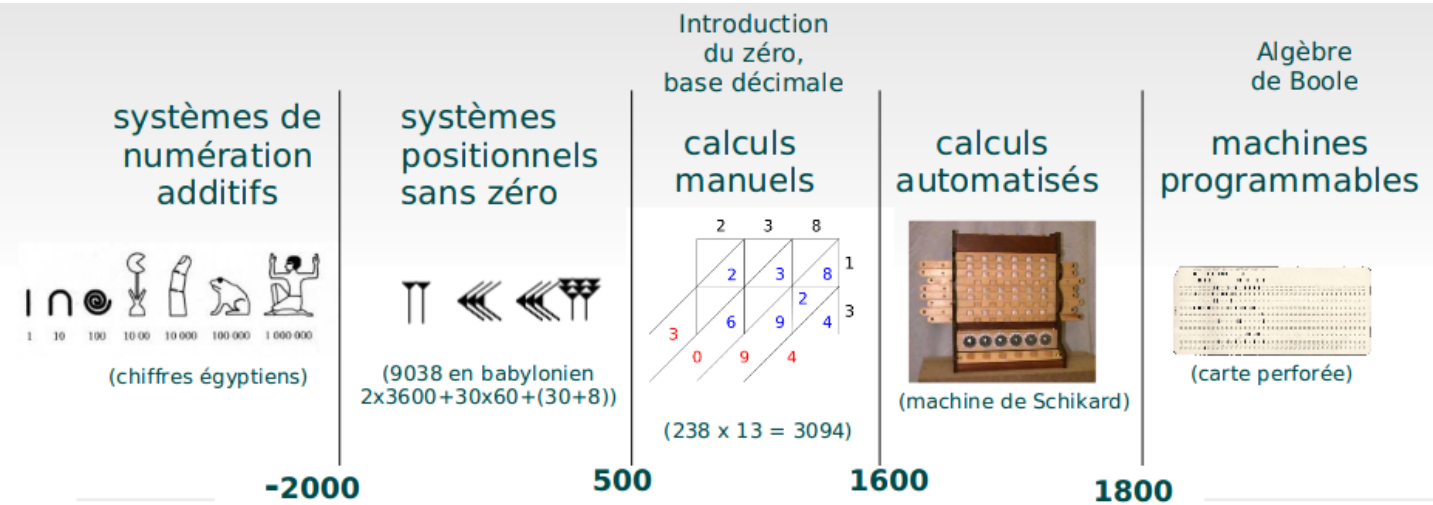
- On utilisera le langage C !

Science du Calcul : Plus de 5000 ans d'existence !



. . . pour aboutir à "**un modèle de machine programmable**".

Les étapes de l'évolution

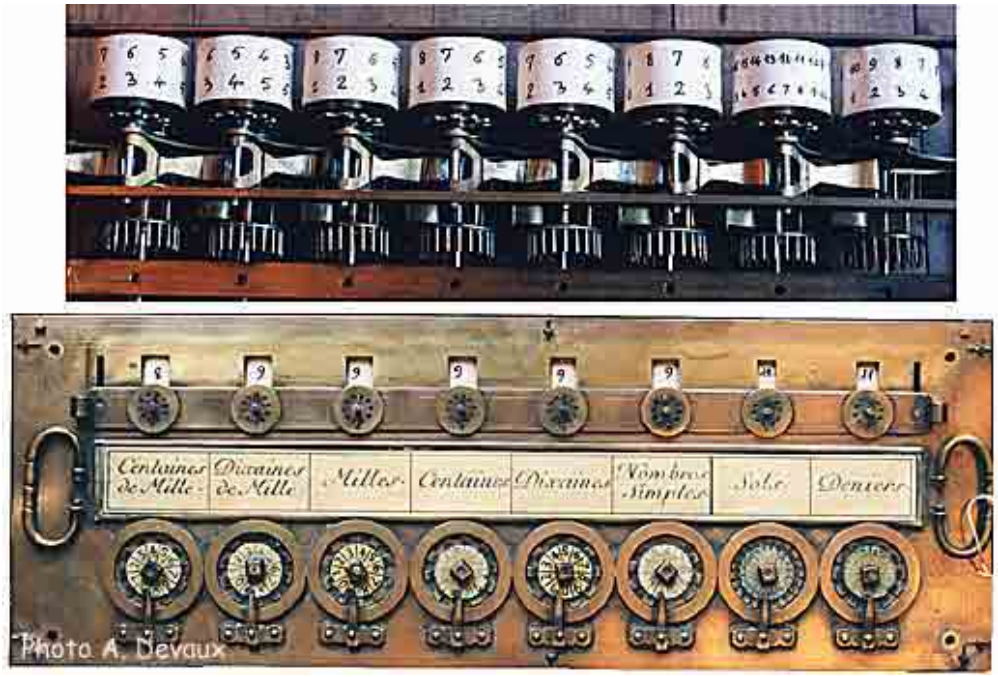


Rendre les traitements de plus en plus "automatiques" :

- Les mathématiques : les nombres et les calculs,
- Automatisation des calculs,
- Enchaînement des calculs,
- Programmation des calculs,
- ...
- Informatique : "traitement automatique de l'information"

Les machines de calcul mécaniques

C'est en 1642, que Blaise Pascal conçut l'idée de la "Pascaline", voulant soulager la tâche de son père (surintendant de la Haute-Normandie), qui devait remettre en ordre les recettes fiscales de cette province.



Elle permettait :

- d'**additionner et de soustraire** deux nombres d'une façon directe,
- et de faire des **multiplications et des divisions** par répétitions.

Cette période est riche de telles machines qui visent à automatiser les opérations arithmétiques.

Mais pour l'instant . . . **pas d'enchaînement des opérations !**

Les machines programmables

Le métier à tisser de Joseph-Marie Jacquard (1752-1834) est le plus célèbre de ces automates . . . à bandes perforées.

➢ La première machine programmable ?

Le "programme" détermine l'enchaînement des opérations.

- Un programme très **séquentiel** . . . sans ruptures (fonctions, etc)
- Dans un premier temps ces machines, ont fort à faire, pour exécuter un seul programme !

Machines modernes

Au sortir de la guerre de 1945, il y une forte effervescence autour de machines électroniques : tubes puis transistors.

Ces machines se développent sur la base d'un modèle, dit de Von Neumann.

Ce modèle définit

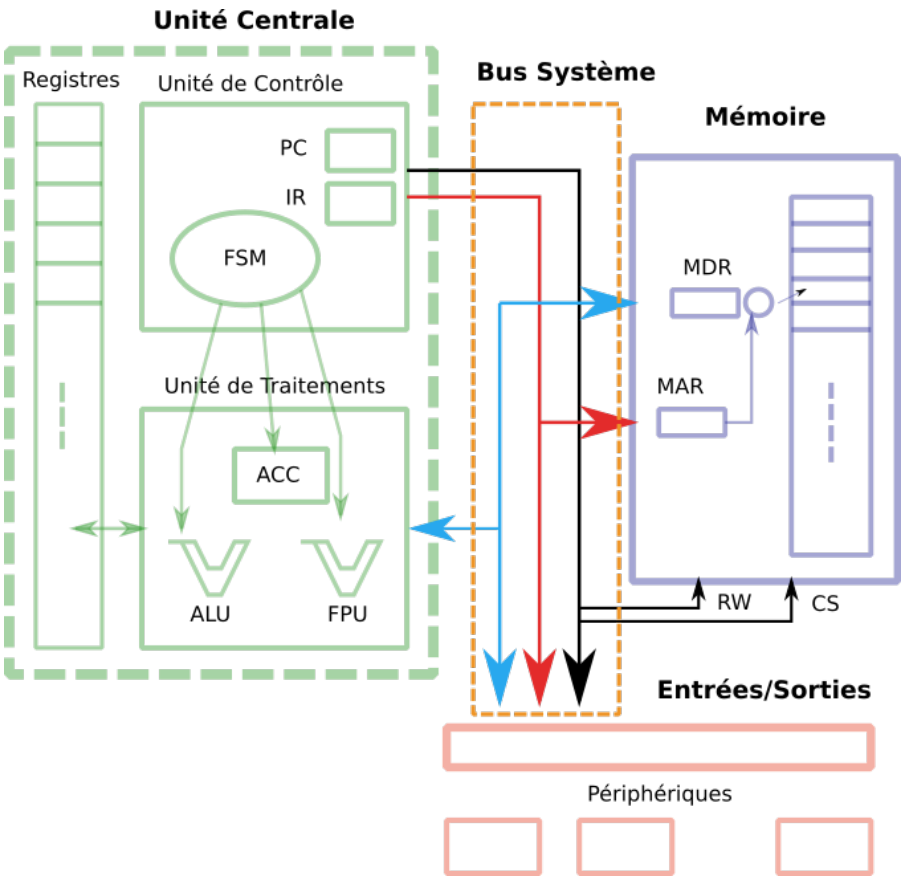
- ➠ les éléments de la machine : rôle et connexion
- ➠ un "fonctionnement" !

  Avec notamment la présence d'une mémoire qui contiendrait le programme et remplacerait les câblages des calculateurs "d'avant".

Encore aujourd'hui, ce modèle est celui des machines/des microprocesseurs.

Modèle de Von Neumann

Le modèle de Von Neumann propose de réaliser une machine sur la base de l'existence de différents **blocs fonctionnels** et de leurs utilisations "codifiées" (par le modèle).

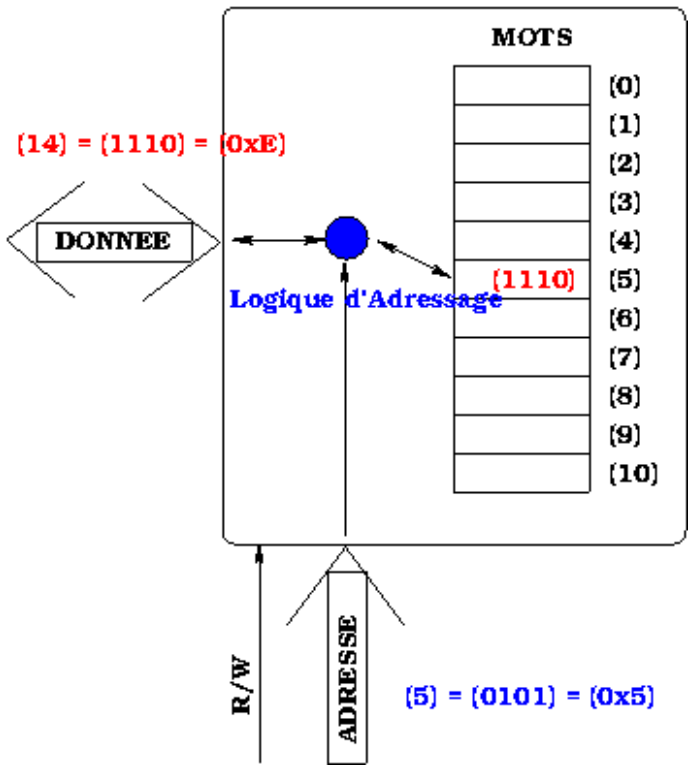


Ces blocs sont :

1. La mémoire (interne à la machine)
2. L'unité centrale
3. Le bus système

La mémoire

Le premier des blocs fonctionnels est la **mémoire**.



En première approximation, la mémoire dont il question ici est assimilable à une "bibliothèque" : une juxtaposition d'étagères.

Chaque étagère est désignée par un numéro, son **adresse**.

Chaque étagère contient une information adressable, un **mot**.

- Ce mot contient une séquence de bits qui représentent **soit une instruction, soit la valeur d'une variable ou d'une constante**.

Les mots sont acheminés dans la mémoire par le bus de données :

- Quand on récupère une information sur une étagère, c'est une lecture (en mémoire).
- Quand on place une information sur une étagère, c'est une écriture (en mémoire).

L'adresse spécifiée par le bus d'adresses permet de désigner le mot concerné par l'opération de lecture ou d'écriture.

Caractérisation de la mémoire

Différents grandeurs caractérisent les capacités de la mémoire :

- La **largeur du bus d'adresses** fait référence au nombre de fils et donc de façon équivalente au nombre de bits qui constituent une adresse.

  De ce nombre, que l'on désignera par **na**, on déduit la **capacité d'adressage** de la machine, soit le nombre d'adresses différentes "exprimables" avec **na** bits.

  Sans surprise, **na** bits permettront de différencier 2^{na} mots. Ceci donne la taille maximale de la mémoire utilisable.

  - Donc avec un bus d'adresse de 32 bits on pourra accéder à $2^{32} = 4$ Giga mots.

  La dernière génération de processeurs Intel (Alder Lake) est capable de manipuler une mémoire de 512 Giga octets (39 bits) (et même plus sur certaines versions).

  Notons que les mécanismes d'adressage de telles machines sont bien moins simples que celui décrit jusqu'à présent. Mais comme l'idée n'est pas de faire un cours sur les architectures de machines, nous nous contenterons du modèle "pour les débutants".

- Le bus de données est comme une autoroute avec **nd** voies possibles.

  64 bits est la **largeur du bus de données** des machines de bureau modernes.

  - Cette largeur conditionne le domaine de valeurs numériques manipulables et donc la précision des calculs.

  Mais lors d'un accès à la mémoire toutes ces voies ne sont pas forcément utilisées.

  Aujourd'hui, les architectures de machines basées sur l'utilisation des processeurs Intel permettent d'accéder à la mémoire avec une "granularité" qui va de l'octet (8 bits) au mot de 64 bits en passant par 16 et 32 bits.

  Ceci signifie qu'une adresse désigne un octet puisque l'on peut y accéder. Mais rien n'empêche de récupérer "en même temps" plusieurs octets qui suivent.

L'unité centrale

Le second bloc fonctionnel est l'**Unité centrale** (Central Processing Unit : CPU).

- Ce bloc fonctionnel correspond "grosso modo" au (micro)processeur que l'on trouve sur les cartes mères des machines modernes.

Il se compose

- d'une **unité de traitements** qui effectue les calculs.
- de l'**unité de contrôle** qui "pilote" le comportement "codifié" de tous les éléments de l'architecture.
- et de **registres**.

  Certains de ces registres sont des éléments de mémorisation (propres à l'UC) des résultats intermédiaires.

  D'autres registres sont dédiés au contrôle comme par exemple le "**Program Counter**" (PC) qui permet de savoir quelle est l'adresse de la prochaine instruction à traiter et qui sera placée dans l'"**Instruction Register**"(IR) pour être analysée.

  Le "PC" permet au programme de "progresser correctement" dans le traitement des instructions.

L'unité de Contrôle

L'**unité de contrôle** est un bloc fonctionnel de l'unité centrale.

Il "déroule en continu" **un automate** (FSM : "Finite State Machine") qui réalise à l'infini et séquentiellement les opérations suivantes :

1. **"Fetch Instruction"** : "Aller chercher la prochaine instruction dans la mémoire selon la valeur de PC".

   Cette instruction "courante" est placée dans le registre CIR ou IR (Current Instruction Register).

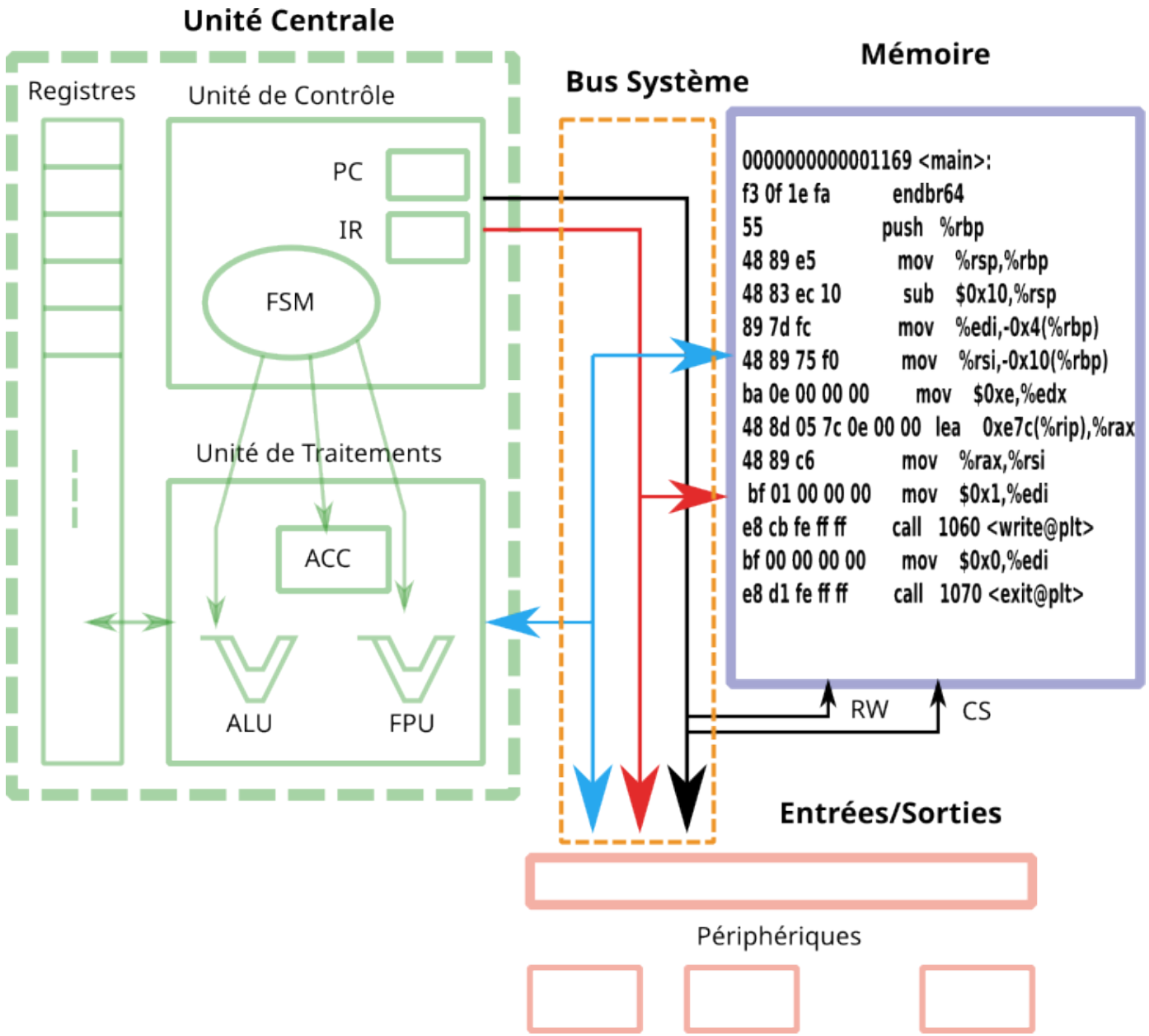
2. **"Decode Instruction"** : "Comprendre cette instruction".

   Comprendre l'instruction, c'est comprendre les traitements qu'elle induit. Potentiellement, ce traitement peut nécessiter l'accès préalable à des opérandes en mémoire.
   - **Operand Address** : "Evaluer l'adresse d'un opérande" pour les instructions qui nécessitent un accès mémoire.
   - **Fetch Operands** : "Aller chercher" en mémoire ou dans les registres les opérandes.

3. **"Execute Instruction"** : "Exécuter l'instruction".

   C'est cette opération qui active et exploite l'**unité de traitement** selon l'instruction qui vient d'être définie.

4. **"Store Result"** : "Stocker" le résultat dans un registre (Accumulateur) ou en mémoire.


Unité Centrale
Registres
Unité de Contrôle
PC
IR
FSM
Unité de Traitements
ACC
ALU
FPU
Bus Système
Mémoire
0000000000001169 <main>:
f3 0f 1e fa endbr64
55 push %rbp
48 89 e5 mov %rsp,%rbp
48 83 ec 10 sub $0x10,%rsp
89 7d fc mov %edi,-0x4(%rbp)
48 89 75 f0 mov %rsi,-0x10(%rbp)
ba 0e 00 00 00 mov $0xe,%edx
48 8d 05 7c 0e 00 00 lea 0xe7c(%rip),%rax
48 89 c6 mov %rax,%rsi
bf 01 00 00 00 mov $0x1,%edi
e8 cb fe ff ff call 1060 <write@plt>
bf 00 00 00 00 mov $0x0,%edi
e8 d1 fe ff ff call 1070 <exit@plt>
RW
CS
Entrées/Sorties
Périphériques

Le bus système

L'unité centrale communique avec la mémoire et avec le "monde extérieur" (E/S) grâce à un "**System Bus**" qui regroupe trois bus :

1. Bus de données /"Data Bus",
2. Bus d'adresse /"Address Bus",
3. Bus de controle /"Control Bus".

L'unicité de la voie d'accés ("Data Bus") par laquelle vont transiter données **et** instructions est **caractéristique du modèle de Von Neumann**.

Les accès au "monde extérieur" correspondent aux "**Entrées/Sorties** ".

- On y trouve les périphériques de stockages (disques, bandes ...), les interfaces de communications (réseaux), ...

Les points importants

Ce qu'il faut bien retenir :

- La définition d'une machine **programmable** aboutit au modèle de Von Neumann.
- Son architecture : 1 mémoire, 1 bus, 1 UCentrale (contrôle + calcul)
- Ce schéma induit **une mémoire unique contenant données (variables/constantes) et instructions** et donc un transfert "permanent" des instructions et des données entre la mémoire et l'UC.

RMQ sur la suite :

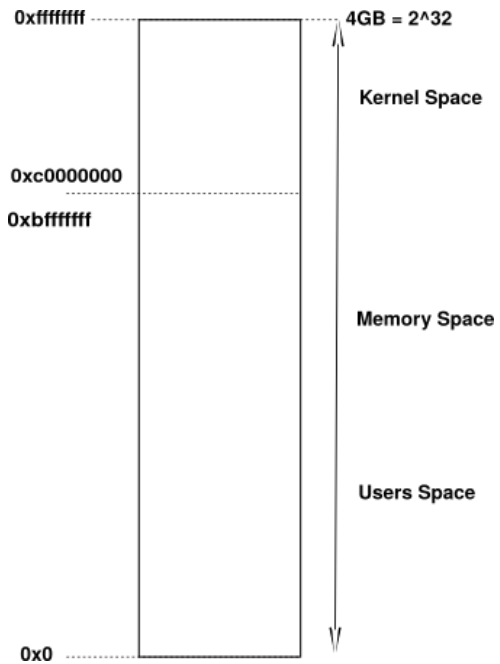
Il n'y a qu'une seule mémoire . . . ce qui pourrait devenir problématique en terme de sécurité/confidentialité si il y a avait plusieurs programmes !

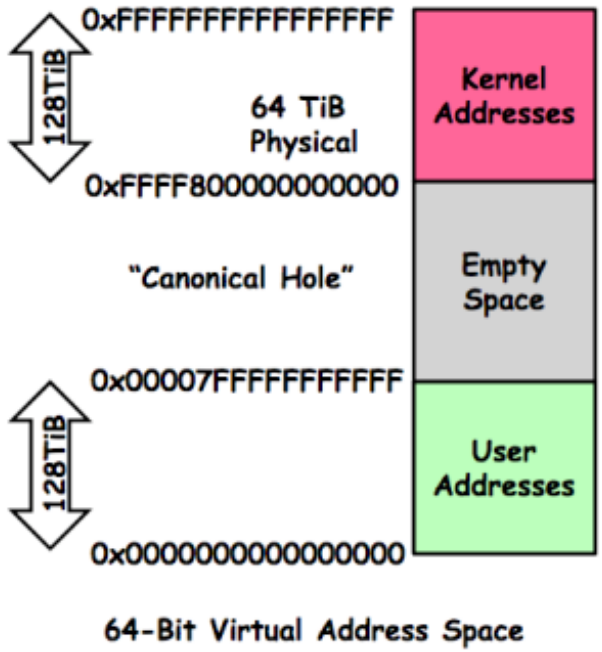
- Comment organiser la cohabitation de plusieurs programmes dans la machine et notamment en mémoire ?

Espace d'adressage d'un programme/processus "User"

L'`espace d'adressage virtuel` d'un programme (/processus) est le modèle qu'il se fait de la mémoire de la machine :

- Un bloc continu de 4GB (si machine 32 bits)
- qui a été splitté pour protéger le noyau . . . on y reviendra !





Quelques contraintes !

Faire d'un programme un processus n'est pas aussi simple que de "transporter **en vrac**" le code et les données du disque (où se trouve le fichier exécutable) vers la mémoire (où résidera le processus).

Il faut tenir compte de quelques exigences/difficultés :

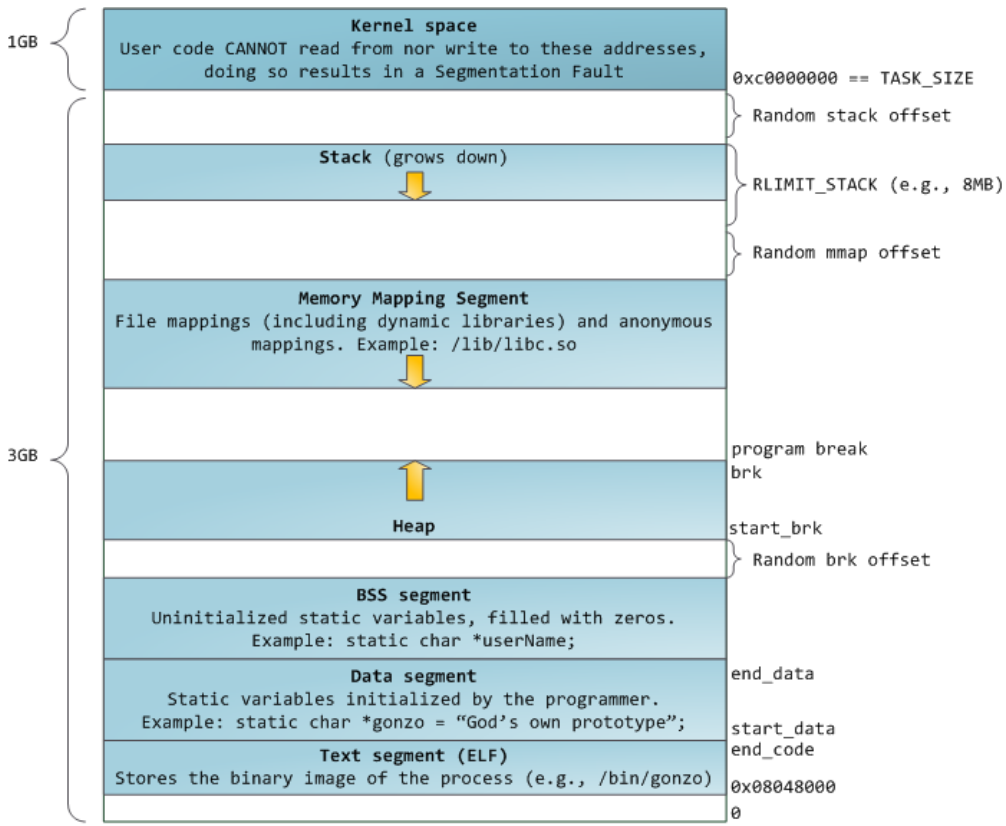
- Un programme définit des instructions ET des données, il va donc falloir organiser la mémoire pour les retrouver facilement,
- Un programme fait souvent appel à des bibliothèques dont le code peut être partagé par plusieurs processus . . . histoire de ne pas multiplier les même codes en mémoire (libc par exemple)
- L'exécution d'un programme peut être influencé au lancement par des arguments en ligne de commande (argv,argv) et il peut aussi exploiter des variables d'environnement (HOME, PATH, . . .).

  Comment les trouver !?
- La mémoire d'un processus peut varier en taille : appel de fonction et récursivité, allocation dans le tas (malloc, . . .)

  Comment gérer ce dynamisme ?

Segments du Processus

Pour répondre à ces exigences, l'espace d'adressage linéaire d'un **programme Unix est traditionnellement partitionné**, d'un point de vue logique, en plusieurs intervalles d'adresses linéaires appelés **segments** (rien à voir avec les segments d'Intel pour les microprocesseurs 80x86) :



https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

- La pile (/stack) : On y trouve les variables automatiques (= dynamiques) des fonctions : adresse de retour, paramètres, variables locales.
- Le tas (/heap) : Permet une gestion de la durée de vie des variables qui ne soit pas dictée par le compilateur mais bien par le programmeur.
- Memory mapping segment : pour faire des bibliothèques partagées, ...
- Initialized data segment : ce segment contient les données initialisées.

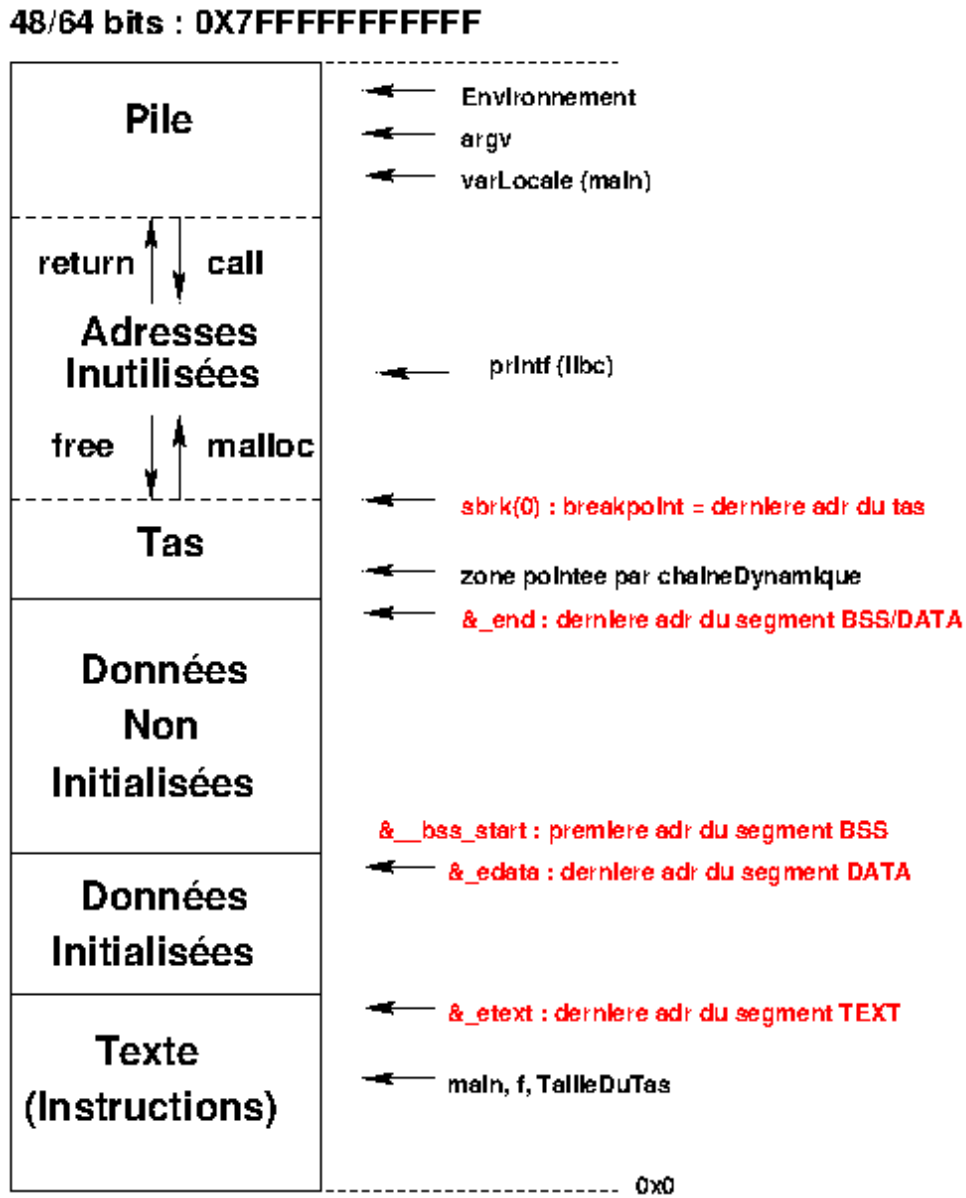
  Les variables statiques et les variables globales dont les valeurs sont stockées dans le fichier exécutable (car le programme doit connaître leurs valeurs au démarrage).
- Uninitialized data segment (bss) : Les données non-initialisées, toutes les variables globales dont les valeurs initiales ne sont pas stockées dans le fichier exécutable (parce que le programme définit les valeurs avant de les référencer).
- Text segment : le texte du programme (/text) c'est à dire les codes des instructions des fonctions.

La machine participe à la mise en oeuvre de cette organisation, MAIS c'est le compilateur qui est à la manoeuvre !

```c
/** Voir les segments par la programmation  **/
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
extern int _end;     /* ld adds a symbol which marks
                        the end of all segments in elf */
extern int _etext;   /* Fin du segment de code     */
extern int _edata;   /* Fin du segment de donnees */
extern int __bss_start; /* Debut du segment BSS  */
extern char **environ;  /* Environnement          */
#define FORMAT "%-17s: %p\n"
/*-------- Variable statiques --------------*/
int nonInitialisee;      /* Variable in BSS  */
int initialisee = 2;     /* Variable in DATA */
/*-------- Taille Du Tas ---------------------*/
long TailleDuTas (void){
return (long) sbrk (0) - (long) & _end;}
/*-------- Une fonction ----------------------*/
int f(int varParam){
  int varLocalef;     /* Variable dans la pile */
  printf ("--> f sur la pile :\n");
  printf (FORMAT, "'varLocale'", & varLocalef);
  printf (FORMAT, "'varParam'", & varParam);
}
/*-------- Main ----------------------------*/
int main(int argc, char *argv[]){
  const int TAILLE = 8192;
  int varLocale;          /* Var dans la pile */
```

```
  char* chaineDynamique; /* Var dans la pile : mais valeur dans le tas ! */
  if ((chaineDynamique = (char *)malloc(TAILLE)) == NULL)
    perror("Pas pu allouer 'chaineDynamique'");

  /* Affichage de la localisation des segments induit par le code ci dessus :  */
  printf ("--> Segments memoire:\n\n");
  printf (FORMAT, "'environ [0]'", environ [0]);
  printf (FORMAT, "'argv [0]'", argv [0]);
  printf (FORMAT, "'environ'", environ);
  printf (FORMAT, "'argv'", argv);
  printf (FORMAT, "'varLocale'", & varLocale);
  printf (FORMAT, "'printf'", printf);
  printf (FORMAT, "'chaineDynamique'", chaineDynamique);
  printf (FORMAT, "'_end'", & _end);
  printf (FORMAT, "'__bss_start'", & __bss_start);
  printf (FORMAT, "'nonInitialisee'", & nonInitialisee);
  printf (FORMAT, "'_edata'", & _edata);
  printf (FORMAT, "'initialisee'", & initialisee);
  printf (FORMAT, "'_etext'", & _etext);
  printf (FORMAT, "'main'", main);
  printf (FORMAT, "'f'", f);
  printf (FORMAT, "'TailleDuTas'", TailleDuTas);
  printf ("\n");

  f(varLocale);
}
```

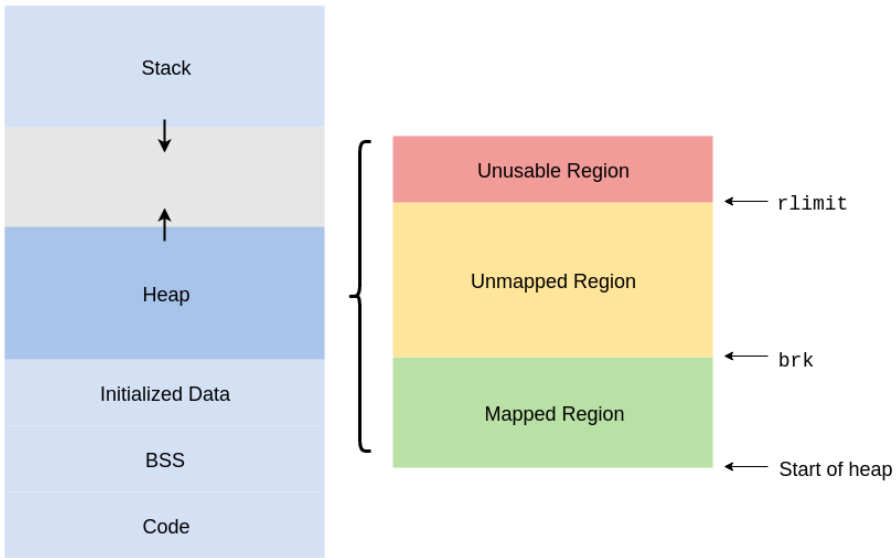
```
-> Segments memoire:

'environ [0]'     : 0x7fff80ffaa60 (HIGH)
'argv [0]'        : 0x7fff80ffaa58
'environ'         : 0x7fff80ff8e68
'varLocale'       : 0x7fff80ff8d5c
'argv'            : 0x7fff80ff8e58
--> f sur la pile :
'varLocale'       : 0x7fff80ff8d2c
'varParam'        : 0x7fff80ff8d1c


'printf'          : 0x7f9302a0b190
Shared lib


'chaineDynamique': 0x56167396f010

'_end'            : 0x561672f49068




'nonInitialisee' : 0x561672f49064
'__bss_start'    : 0x561672f49054

'_edata'          : 0x561672f49054
'initialisee'     : 0x561672f49050

'_etext'          : 0x561672d47bed
'main'            : 0x561672d47908
'f'               : 0x561672d478b0
'TailleDuTas'     : 0x561672d47890 (LOW)
```



Le point de rupture (breakpoint)

Le point de rupture (breakpoint = sbrk(0)) d'un processus est défini comme :

- La "break value" (point de rupture) est l'adresse du premier mot hors (au delà) du segment des données (data). Ce qui correspond au "sommet (actuel) du tas".



On ne peut théoriquement pas accéder à des adresses supérieures à ce point de rupture puisqu'elle n'ont pas encore été "mappées" (on va y revenir)

- En fait, avec les mécanismes de pagination (accès par page) de la mémoire, on peut le faire (sans générer systématiquement de core) MAIS c'est logiquement incorrect (et très non portable).

Tab of Tab : gestion des pointeurs et segments

La manipulation de tableaux de tableaux (les chaînes de caractères) doit être acquise !

```c
/* Fichier : taboftab.c  */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*-------------------------------------*/
void afficher(char *TofT[]){
  int i = 0;
  while (TofT[i] != NULL){
      printf("%s\n",TofT[i]);
      i++;
    }
}
/*-------------------------------------*/
int main(){  // exec sous https://pythontutor.com/c.html#mode=display
  char *T[6]; // Creation dans la pile
  T[0] = strdup("Zero"); // Creation dans le tas
  T[1] = strdup("Un");
  T[2] = strdup("Deux");
  T[3] = NULL; // Stop

  afficher(T); // Une fonction => la pile !

  free(T[0]); // Gestion du tas
  free(T[1]);
  free(T[2]);
  return 0;
}

```

Sous pythontutor

https://pythontutor.com/render.html#mode=display

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

C (C17 + GNU extensions)
known limitations

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 /*-------------------------------------*/
 5 void afficher(char *TofT[]){
 6   int i = 0;
 7
 8   while (TofT[i] != NULL){
 9       printf("%s\n",TofT[i]);
10       i++;
11     }
12 }
13 /*-------------------------------------*/
14 int main(){
15   char *T[6];
16   T[0] = strdup("Zero");
17   T[1] = strdup("Un");
18   T[2] = strdup("Deux");
19   T[3] = NULL; // Stop
20
```

Edit this code

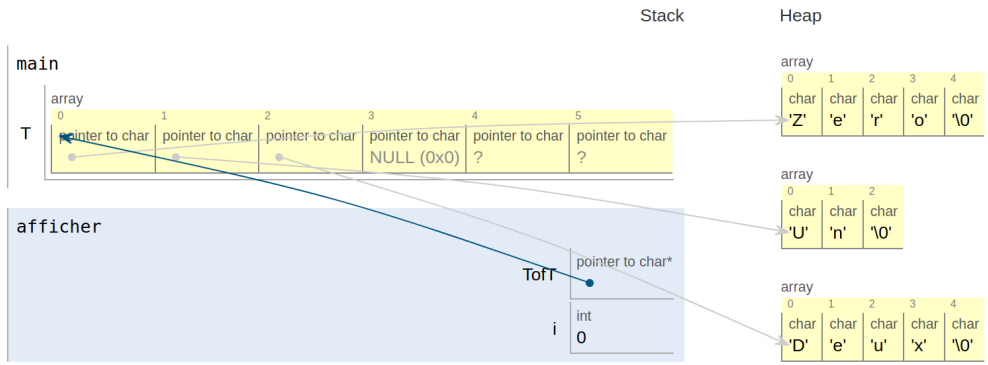
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 11 of 24

Print output (drag lower right corner to resize)

```
Zero
```



Note: ? refers to an uninitialized value

C/C++ details: none [default view]

Fonction et mémoire

L'"appel de fonction" permet d'exécuter le code factorisé dans la fonction.

Syntaxiquement, un appel de fonction est réalisé lorsqu'un nom de fonction est suivi d'une parenthèse ouvrante :

- ➢ cet appel peut être l'opérande d'une expression :

$$i = j + pow(i, j);$$

- ➢ ou une instruction à part entière :

printf("toto");

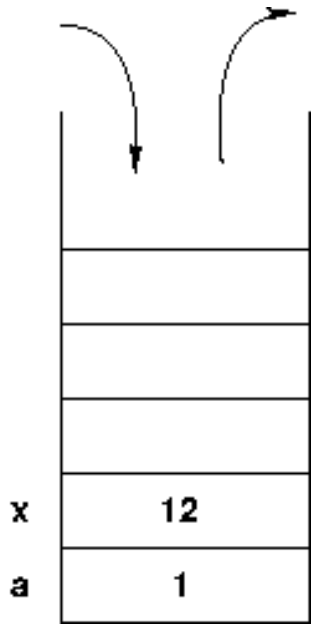
Remarque :

A l'exécution, en C, il n'y a **aucun contrôle dynamique** ni sur le nombre, ni sur le type des paramètres ...

- ➢ d'où l'intérêt du prototypage qui permet un contrôle statique (à la compilation).

Pile d'exécution

Le concept de fonction est **très lié** à la notion de pile !



Puisque la mémoire est une "étagère", un tableau.

Rien n'empêche de gérer ce tableau comme une pile :

- ➢ On crée et on supprime dans la mémoire des éléments, par le sommet de la pile.

C autorise la récursivité :

Un langage, qui ne mettrait pas en place au niveau de la gestion mémoire ce modèle d'exécution, ne pourrait pas gérer de récursivité !

C'est le seul moyen d'obtenir une "profondeur" dynamique d'appels de fonctions.

Un tel appel entraîne :

```
int f (int x){
  int y = x+1;
  return y;
}
int main(void){
  int a = 1;
  a = f(3*4);
}
```

① l'évaluation des paramètres d'appel :

➢ l'ordre d'évaluation est **non normalisé** !

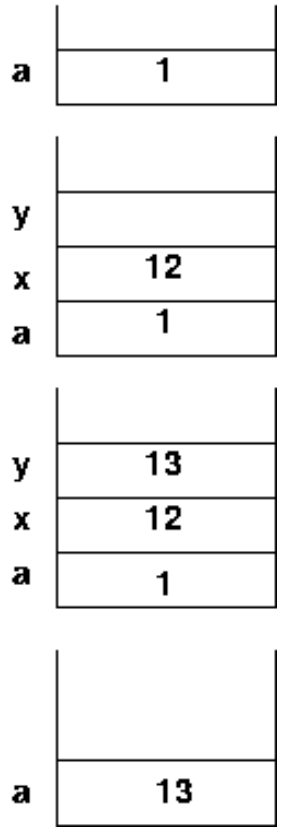
② pour chaque paramètre, il y a création sur la pile d'exécution d'une variable **locale à l'appel** qui sera supprimée au retour de l'appel.

Cette variable est initialisée à la valeur de l'expression correspondante (par sa position) qui constitue le paramètre effectif :

➢ c'est la **TRANSMISSION PAR VALEUR** des paramètres.

Exemple de passage de paramètres **par valeur**

```
int f (int x){
  int y = x+1;
  return y;
}
int main(void){
  int a = 1;
  a = f(3*4);
}
```



A l'appel de `f` par `main`,

① $3 * 4$ est évalué, donnant 12,

② x et y sont créés sur la pile,

③ et x est initialisé avec 12,

④ les instructions de la fonction sont exécutées : y vaut 13,

⑤ la valeur de y est celle renvoyée par la fonction

⑥ et a est affectée avec cette valeur.

⑦ La fonction prenant fin, x et y son retirées de la pile.

L'existence (i.e. la présence) de x et de y sur la pile est liée au fait que la fonction f est en cours d'exécution !

➢ Variables automatiques / dynamiques

Buffer overflow

Dans ce suit qui nous allons montrer que ce mécanisme de pile, si utile aux fonctions, peut être une source de piratage.

Nous abordons ici la notion de débordement de tampon au niveau de la pile d'exécution.

- un débordement de la mémoire tampon de la pile se produit lorsqu'**un programme écrit à une adresse mémoire sur la pile d'appels du programme en dehors de la structure de données prévue**, qui est généralement une mémoire tampon de longueur fixe

---

Il s'agit d'une vulnérabilité très ancienne et pourtant très répandue encore . . . et vous allez comprendre pourquoi !

L'exemple qui suit va nous servir de base de travail :

```
1   /*
2   Fichier "bof.c" : Simplification du use case de
3   https://zestedesavoir.com/articles/100/introduction-aux-buffer-overflows/
4   et à compiler avec la commande : gcc bof.c -fno-stack-protector */
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <string.h>
8   #include <unistd.h>
9
10  #define BUFSIZE 12
11  /*======================*/
12  int main(int argc, char *argv[]){
13    int access_granted = 0;
14    char password[BUFSIZE] = "hello"; /* Le vrai mot de passe */
15    char proposal[BUFSIZE];                  /* Mot de passe propose */
16
17    printf("Enter the password to get the access granted! ");
18    scanf("%s", proposal);
19
20    if (strcmp(proposal,password) == 0)
21      access_granted = 1;
22
23    printf("%p\n",&access_granted);
24    printf("%p\n",proposal);
25    printf("%p\n",&proposal[strlen(proposal)]);
26
27    printf("%d\n",access_granted);
28    /* => Core dumped en cas de stack smashing detected */
29    if(access_granted) { /* Si l'autorisation est donnée */
30      char *newargv[] = {NULL};
31      char *newenviron[] = {NULL};
32      printf("Access granted ! \n"); /* On execute un nouveau Shell */
33      execve("/bin/sh", newargv, newenviron);
34    } else { printf("ACCESS DENIED !\n");
35    }
36    exit(EXIT_SUCCESS);
37  }
```

Test du programme

```
~/EnseignementsCurrent/Sys_Prog_Cours/Tps...
menez $ ls

a.out  bof.c  stack_over.c
menez $ gcc bof.c -fno-stack-protector
menez $ ./a.out
Enter the password to get the access granted! hello
Access granted !
$ exit
menez $ ./a.out
Enter the password to get the access granted! ciao
ACCESS DENIED !
menez $ ./a.out
Enter the password to get the access granted! xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Access granted !
$
```

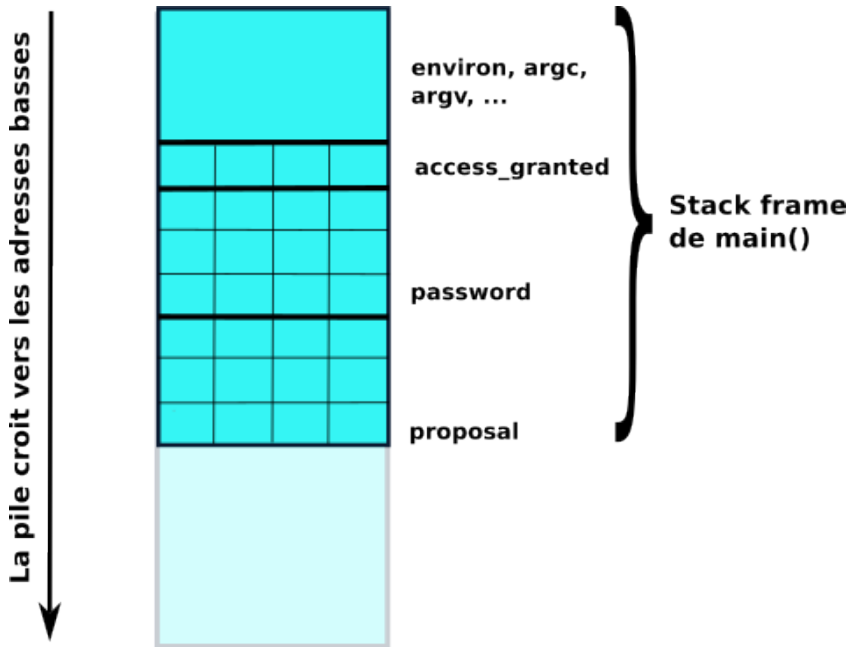
Sur la figure, vous constatez qu'on arrive à accéder en fournissant un mot de passe FAUX mais suffisamment long !

- Nous avons effectué un "buffer overflow" et nous avons débordé sur la variable "access_granted" qui valait désormais autre chose que 0 au moment du test exploitant l'autorisation d'accès.

La fonction scanf

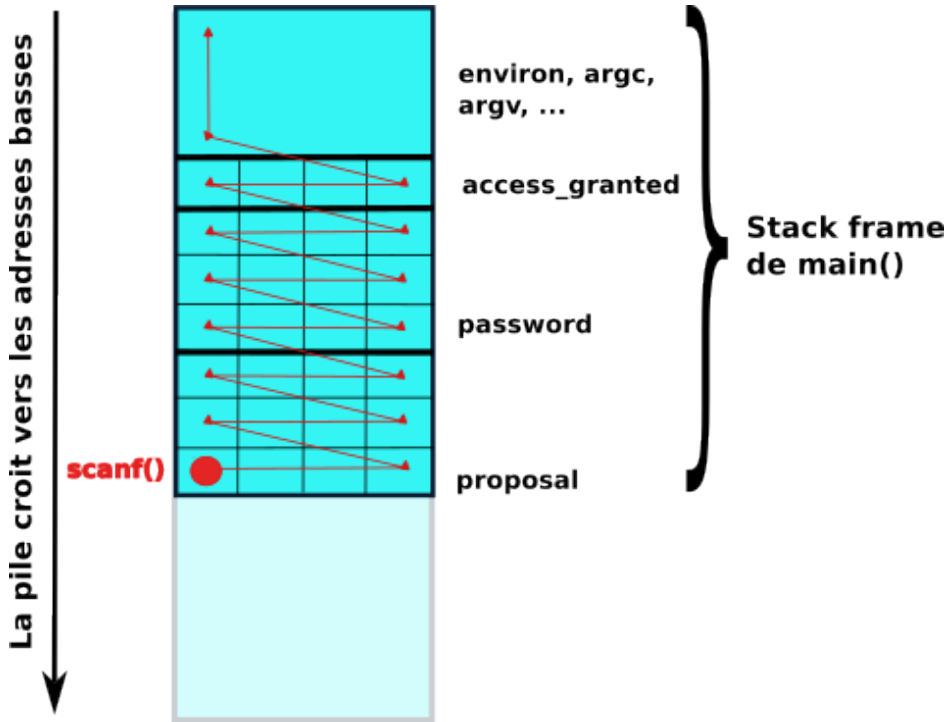
Pour réaliser ce débordement, nous avons utilisé la fonction scanf (dont la mauvaise réputation n'est plus à faire en terme de sécurité !)

- Avant l'appel à scanf, l'accès n'est pas autorisé puisque la valeur vaut 0
- La pile d'exécution est conforme au schéma suivant :



- Et scanf est appelé !

scanf() place les octets tapés au clavier dans la zone d'adresse commencant en "proposal" :



Le problème c'est qu'il n'y a pas de limite/borne !

Si on "pousse", on vient écraser les contenus dans la pile notamment celui de la variable access_granted qui aurait du rester à 0 pour signifier la "fermeture".

"Stack smashing"

https://en.wikipedia.org/wiki/Stack_buffer_overflow

Un débordement de la mémoire tampon de la pile peut être provoqué délibérément dans le cadre d'une attaque connue sous le nom de "**stack smashing**" (écrasement de la pile).

- Si le programme concerné s'exécute avec des privilèges spéciaux ou accepte des données provenant d'hôtes réseau non fiables (par exemple, un serveur web), le bogue constitue alors **une vulnérabilité potentielle en matière de sécurité**.
- Si la mémoire tampon de la pile est remplie de données fournies par un utilisateur non autorisé, celui-ci peut corrompre la pile de manière à injecter un code exécutable dans le programme en cours d'exécution et prendre le contrôle du processus.

Il s'agit de l'une des méthodes les plus anciennes et les plus fiables permettant aux attaquants d'obtenir un accès non autorisé à un ordinateur.

- N'exécutez que les codes dont vous êtes sûrs !

Les fonctions "à surveiller"

Un certain nombre de situations peuvent entraîner un débordement de mémoire tampon, comme l'utilisation de types et de fonctions non sûrs, la copie ou l'accès non sécurisé à la mémoire tampon, etc.

cf : https://www.cyberpunk.rs/buffer-overflow-linux-gdb

Par exemple, les fonctions (C/C++) suivantes sont "naturellement" nuisibles et peuvent engendrer des vulnérabilités :

- fgets, gets,
- getws,
- sprintf,
- strcat,
- strcpy, strncpy,
- scanf,
- memcpy,
- memmove

Toujours sur la pile

On vient de montrer comment le mécanisme de gestion des variables dynamiques (local data storage) dans la pile pouvait être corrompu.

Mais l'écrasement n'est pas une fin en soi, le pirate aimerait pouvoir faire exécuter un code qui n'était pas prévu !

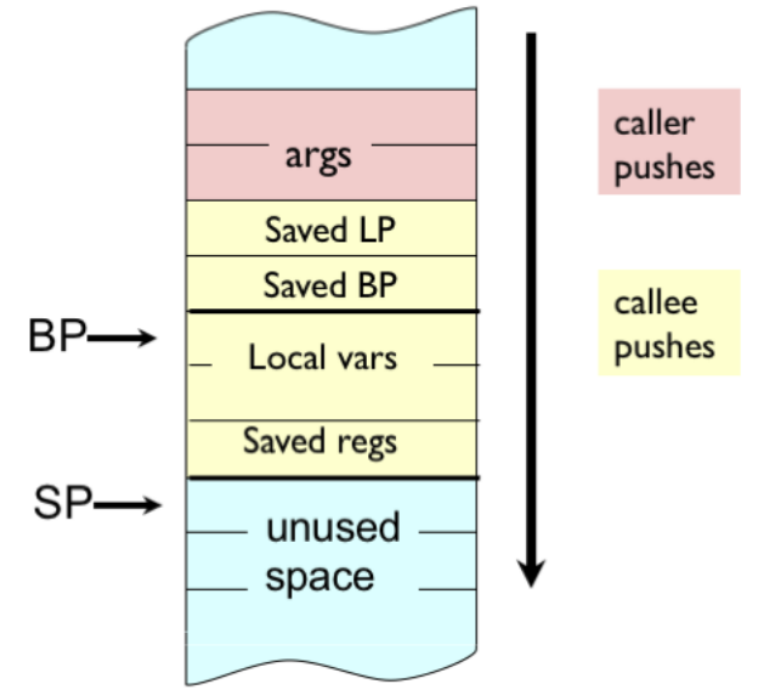
- et pour cela, **il faut arriver à modifier le flot d'instructions** pour aller exécuter le bout de code qui permettra le crack.

Ca tombe bien la pile est aussi utilisée par bien d'autres mécanismes (cf "Functions of the call stack dans https://en.wikipedia.org/wiki/Call_stack) et notamment celui de la gestion des appels de fonctions.

Jouer avec le mécanisme d'appel

La pile d'exécution est composée de "stack frames" (also called "activation records" or "activation frames").

- Chaque appel de fonction induit le "push" d'une stack frame.



Une "stack frame" contient les informations liées à "cette" instanciation / activation / appel de la fonction.

Classiquement :

- les valeurs de ces paramètres (déjà vu)
- les variables locales de l'instance courante (déjà vu),
- mais aussi une "adresse de base",
- **et l'adresse de retour** !

https://computationstructures.org/lectures/stacks/stacks.html

On regarde le stack frame ...

Je ne souhaite pas rentrer trop loin dans l'assembleur et la programmation du microprocesseur (surtout que c'est dépendant du microprocesseur utilisé) mais ces informations ont un intérêt certain ... pour un pirate.

L'**adresse de base** intervient pour adresser les variables "locales" (y compris les paramètres) de la fonction.

- Chaque variable locale a une adresse mais cette adresse ne peut pas être déterminée statiquement puisqu'il peut y avoir plusieurs instances de la fonction sur la pile.

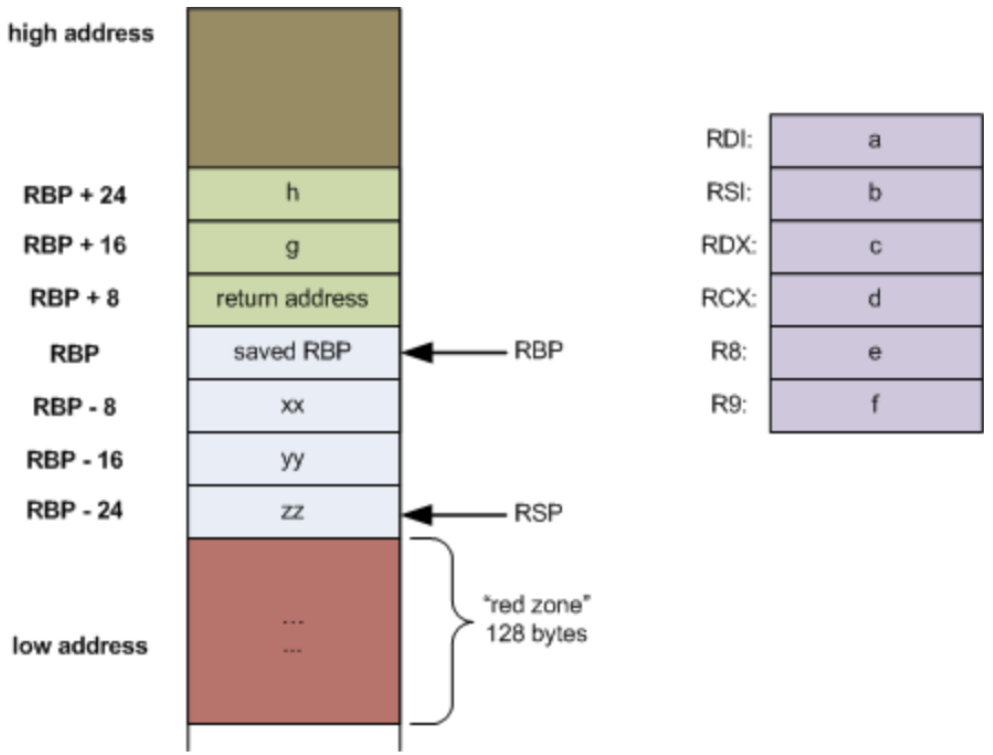
L'idée est de désigner les adresses des objets locaux de l'instance de la fonction par des déplacements (constantes positives ou négatives) par rapport à une base qui est définie dynamiquement lorsque l'instance de la fonction est placée sur la pile.

- Dans les architectures Intel 32 bits, cette base doit être placée dans le registre EBP (Base Pointer) lorsque l'instance est active.

  Pour les 64 bits, le registre change mais le principe reste.

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```

This is the stack frame:



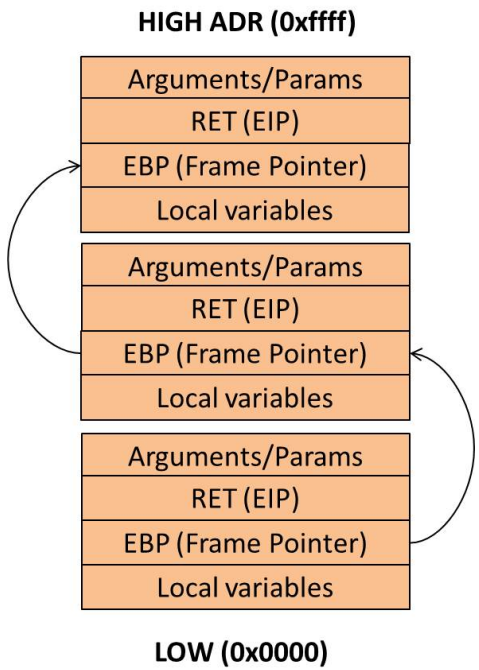
https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/

Plusieurs stack frames

Lorsque une fonction appelle une autre fonction, l'appel provoque une rupture du flot d'instruction :

- au lieu d'exécuter l'instruction à l'adresse suivante, on va "ailleurs" (là ou se trouve le code de la fonction appel).

Mais **la notion d'appel suppose un retour** . . . sur l'instruction juste après l'appel.



Pour organiser le retour, le compilateur prévoit de placer dans la pile l'adresse de l'instruction à exécuter juste après le return de la fonction appelée.

Cette adresse est contenue dans le registre EIP (Instruction Pointer) au moment du de l'exécution de l'appel.

Vulnérabilité !

Si on arrive à modifier cette valeur de l'adresse de retour dans la pile, on change la nature du programme et on peut essayer de lui faire faire ce que l'on veut !

https://www.cyberpunk.rs/stack-structure-overview-gdb

Tout est bon . . . à cracker

La pile n'est pas la seule cible possible d'attaques.

La technique du buffer overflow peut s'appliquer à la pile/stack mais aussi au tas/heap (bref à la mémoire !) :

https://en.wikipedia.org/wiki/Buffer_overflow

Shellcode

Un Shellcode est défini comme un ensemble d'instructions injecté dans un programme.

- A l'origine il s'agit d'un code qui permet de lancer un Shell root.
  On rappelle qu'un Shell est un interpréteur de commandes ;-)

Aujourd'hui, le terme désigne plus largement **une technique dont l'objectif est de faire exécuter un code qui ne faisait pas partie du programme originel**.

Si vous êtes capable de faire cela, qui plus est sur un programme qui dispose de droits spéciaux (root) ...

- c'est le jackpot car vous êtes le maître de la machine !

Le shellcode est utilisé pour manipuler directement les registres de la machine aussi le Shellcode est généralement écrit en assembleur et traduit en "hexa/codes operations".

Il est difficile d'injecter un code écrit dans un langage de haut niveau et inutile puisque c'est un code petit.

L'exemple qui suit montre ce qu'est un Shellcode :

```
1   /*
2     Shellcode : Hello World
3     Compile with : gcc -Wall -z execstack hello.c -o hello
4    */
5   #include <stdio.h>
6   #include <string.h>
7
8   int main(int argc, char **argv) {
9
10    // Bytecode from  https://gist.github.com/procinger/a65c8bde824a10294a4a6966de5a47b4
11    char code[] = "\xeb\x20\x48\x31\xc0\x48\x31\xff\x48\x31\xf6"
12      "\x48\x31\xd2\xb0\x01\x40\xb7\x01\x5e\xb2\x0c"
13      "\x0f\x05\x48\x31\xc0\xb0\x3c\x40\xb7\x00\x0f\x05"
14      "\xe8\xdb\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f"
15      "\x72\x6c\x64\x21";
16
17    int (*func)();                    // function pointer
18    func = (int (*)()) code;    // make a func from our bytecode
19
20    (int)(*func)();                   // execute the function code[]
21
22    return 1;
23  }
```

```c
/*
  Shellcode : Hello World
  Compile with : gcc -Wall -z execstack hello.c -o hello
 */
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {

  // Bytecode from  https://gist.github.com/procinger/a65c8bde824a10294a4a6966de5a47b4
  char code[] = "\xeb\x20\x48\x31\xc0\x48\x31\xff\x48\x31\xf6"
    "\x48\x31\xd2\xb0\x01\x40\xb7\x01\x5e\xb2\x0c"
    "\x0f\x05\x48\x31\xc0\xb0\x3c\x40\xb7\x00\x0f\x05"
    "\xe8\xdb\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f"
    "\x72\x6c\x64\x21";

  int (*func)();                    // function pointer
  func = (int (*)()) code;    // make a func from our bytecode

  (int)(*func)();                   // execute the function code[]

  return 1;
}
```

On distingue trois étapes/parties :

1. Le code qui sera injecté . . . écrit en "code machine",

   On passe par du code machine parce que les codes "compilés" suivent des règles de "bonne conduite". Là on fait ce que l'on veut ou presque.

2. Le positionnement d'un pointeur de fonction à l'adresse du début du code,

3. Et enfin l'appel à ce code via le pointeur de fonction,

(Attaquez par) les appels systèmes

On reviendra sur les "appels systèmes" mais ce que l'on peut déjà dire c'est qu'ils sont **à la base** de toutes les actions que l'on peut demander au système d'exploitation et plus précisément à son noyau (la partie que l'on ne voit jamais ;-).

Ces appels permettent notamment d'accéder aux fichiers, aux processus et à la mémoire qui sont des cibles de choix pour "cracker" le système.

- C'est pour cette raison que les Shellcodes utilisent souvent des appels systèmes.

* Le terme "appel système" est en réalité une interruption et pas un appel sur la pile !

Taille du Shellcode

Un "bon" shellcode doit aussi simple et aussi compact, que possible.

- Plus le plus le shellcode est petit, plus il sera utile d'un point de vue générique.

Pour injecter un Shellcode dans un programme sur disque on va chercher à ne pas modifier la taille du fichier et c'est donc plus facile à faire si on a peu de code à insérer.

Idem à l'exécution, il faut que l'empreinte mémoire reste la plus proche possible de l'originel.

- En général, un Shellcode occupe quelques dizaines d'octets mais cela suffit largement pour réaliser un recouvrement d'image du processus et ainsi modifier complètement le comportement du processus.

Réaliser son Shellcode

On voit cela dans un futur TP !