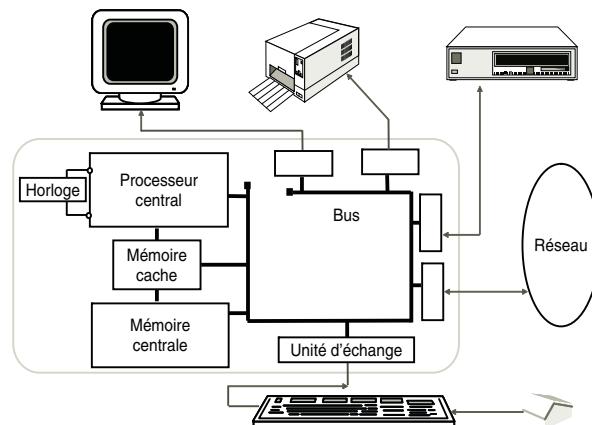


## Table des matières

1	Qu'est-ce qu'un ordinateur ? .....	3
	Codage de l'information .....	24
	Du problème à sa résolution .....	36
	Compilation d'un programme C : le travail du préprocesseur .....	40
2	La segmentation : les différentes parties d'un processus .....	44
	La pile : utilisation au travers d'appels de fonction .....	47
3	La pile : appel de fonction et passage de paramètres .....	50
	L'arithmétique des pointeurs .....	52
	La multiprogrammation .....	55
4	Gestion de la mémoire entre différents processus .....	56
	Gestion de la mémoire : les solutions fournies par la mémoire virtuelle .....	57
	Le «paging» .....	60
5	Présentation d'Unix .....	62
	La virtualisation de la machine vue par Unix .....	69
	Mode utilisateur et mode noyau .....	71
6	Les Processus Unix .....	72
7	Notion de programme et de processus .....	73
	Commutation de contexte .....	76



# 1 Qu'est-ce qu'un ordinateur ?



- \* une *mémoire centrale* :
    - ◊ architecture «Von Neuman» : programme et données résident dans la même mémoire.  
*La lecture ou l'écriture dans la mémoire concerne une donnée ou une instruction.*
    - ◊ architecture «Harvard» : programme et données sont dans des mémoires différentes ;  
*Il est possible de charger simultanément une instruction et une donnée.*
- L'architecture d'un ordinateur est de type «von Neumann». Des matériels embarqués, comme des modules Arduino ou des DSP, «Digital Signal Processor», utilisent une architecture de type «Harvard».

- \* un *processeur central* ou CPU, «Central Processing Unit» qui réalise le traitement des informations logées en mémoire centrale :
  - ◊ le processeur permet l'exécution d'un programme ;
  - ◊ chaque processeur dispose d'un langage de programmation composé d'**instructions machine** spécifiques ;
  - ◊ **résoudre un problème** : exprimer ce problème en une suite d'instructions machines ;
  - ◊ la solution à ce problème est **spécifique à chaque processeur** ;
  - ◊ le programme machine et les données manipulées par ces instructions machine sont placés dans la mémoire centrale.
- \* des *unités de contrôle* de périphériques et des périphériques :
  - ◊ périphériques d'entrée : clavier, souris, etc ;
  - ◊ périphériques de sortie : écran, imprimante, etc ;
  - ◊ périphérique d'entrée/sortie : disques durs, carte réseau, etc.
- \* un *bus de communication* entre ces différents composants.



Mais un bit, c'est quoi au juste ?  
Qu'est-ce que l'adressage ?



# Qu'est-ce qu'un bit, «*binary digit*» ?

5

Un **bit** représente un système à **deux états** possibles :

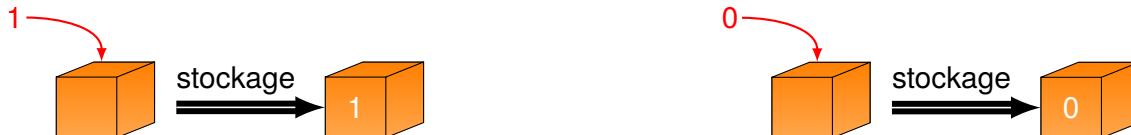
- «allumé»,  ou «éteint»,  ou «éteint», , d'où le symbole présent sur les interrupteurs poussoir :  ou 
- «Vrai» ou «Faux» ;
- un **voltage** «*bas*»  ou un voltage «*haut*»  ou de sens sud-nord 

## Qu'est-ce qu'un bit de mémoire dans un ordinateur ?

«Un bit est juste un emplacement de stockage d'électricité :

- ▷ *s'il n'y a pas de charge électrique alors le bit est 0 ;*
- ▷ *s'il y a une charge électrique*  *alors le bit est 1*

*La seule chose que l'ordinateur peut mémoriser est si le bit est à 1 ou 0.»*



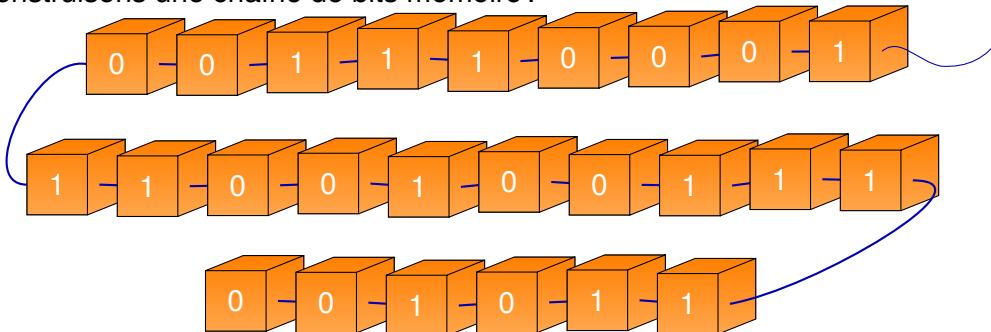
Chaque bit de mémoire correspond à une case dans laquelle on peut stocker un bit de données, soit la valeur 1, soit la valeur 0.



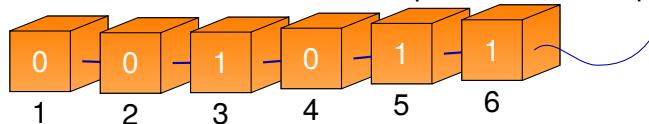
# Comment accéder à la valeur de chaque bit de mémoire ?

6

Construisons une chaîne de bits mémoire :

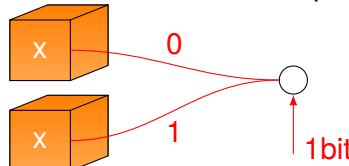


Comment accéder à une case précise ?  $\Rightarrow$  On pourrait numérotter les cases !



Mais, si le numéro de la case doit être manipulé par l'ordinateur, il doit **être mémorisé** dans des cases !

Combien de bits faut-il pour numérotter toutes les cases mémoires ?

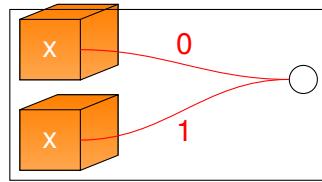


Un bit de «comptage» peut compter deux bits de mémoire... Si on veut 100 cases mémoires, il faut 50 cases pour les compter ? Non c'est un peu plus compliqué que cela...

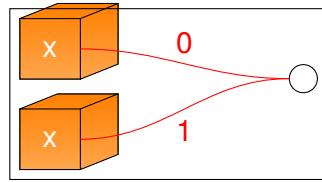


## Numérotation des cases

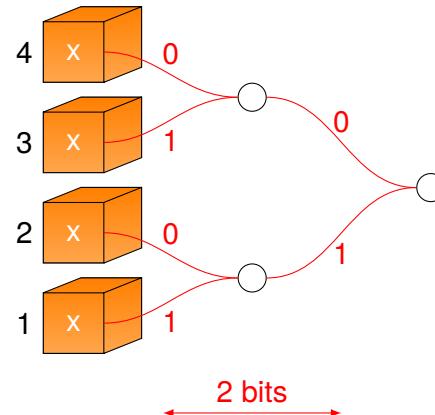
7



+



=



Numéro case	bits n
1	00
2	10
3	01
4	11

On remarque que si on lit la combinaison des bits n de droite à gauche, on arrive directement au bit désiré :

Numéro case	bits n
1	00
2	01
3	10
4	11

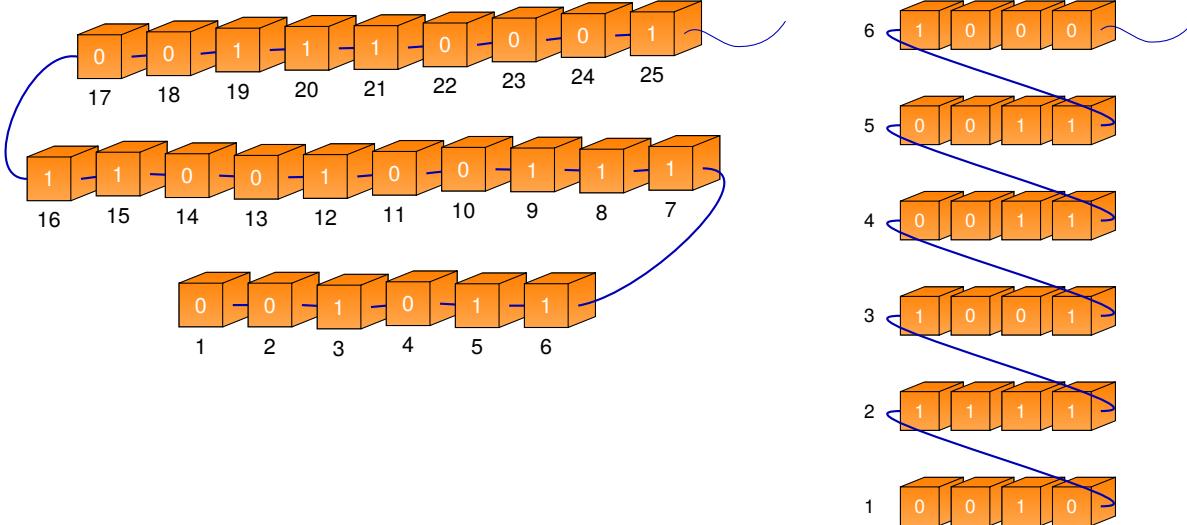
- ⇒ Pour 4 cases, ou 4 bits de données, il faut 2 bits de numérotation.
  - ⇒ Pour 8 cases, il faut 3 bits de numérotation.
  - ⇒ Pour 16 cases, il faut 4 bits de numérotation.
  - ⇒ Pour 32 cases, ou 32 bits de données, il faut 5 bits de numérotation.
- Il faut  $\log_2(N)$  bits de numérotation pour N cases ⇒ pas très rentable !



# Comment diminuer la numérotation ?

8

On associe les bits mémoires par groupe de taille donnée, puis on compte ces groupes :



Ici, on a regroupé par groupe de 4 bits, appelé «quartet» ou «nibble».

Il faut :

- ▷ 1 bit de numérotation pour deux quartets, ou 8 bits mémoire ;
- ▷ 2 bits de numérotation pour 4 quartets, ou 16 bits de mémoire ;
- ▷ 3 bits de numérotation pour 8 quartets, ou 32 bits de mémoire ;

C'est mieux !



# Quelle est la bonne taille de regroupement des bits de mémoire ?

9

Il faut trouver un compromis entre :

- le nombre de bits nécessaire à la **numérotation** de ces groupes ;
- l'intérêt qu'un groupe peut représenter pour **coder** de l'information.

Quel est l'intérêt d'un groupe ?  $\Rightarrow$  *le nombre de séquences de bits différentes qu'il peut exprimer.*

Avec un quartet ou «*nibble*», combien de **séquences différentes** de bits peut-on exprimer ?

$$2 * 2 * 2 * 2 = 2^4 = 16$$

Et si on voulait associer un caractère de l'alphabet à chacune de ces séquences ? On ne pourrait pas exprimer l'alphabet entier !

séq.	lettre	séq.	lettre	séq.	lettre	séq.	lettre
0000	a	1000	e	1111	i	1010	m
0001	b	1001	f	0110	j	1011	n
0010	c	1100	g	0101	k	1101	o
0100	d	1110	h	1001	l	0010	p

$\Rightarrow$  Il faut trouver un **meilleur compromis** !

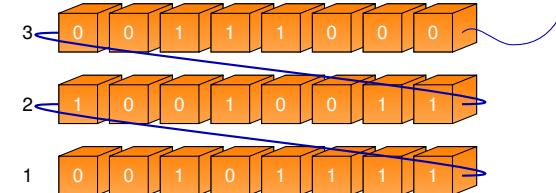
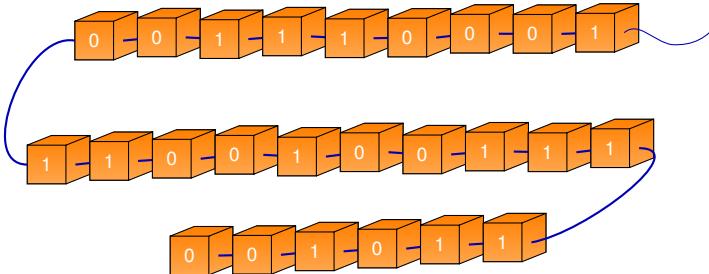
- $\triangleright 2^5 = 32 \Rightarrow$  on peut exprimer un alphabet mais il reste juste 6 caractères pour la ponctuation...
- $\triangleright 2^6 = 64 \Rightarrow$  un alphabet + 10 chiffres + caractères accentués + ponctuation, encore un peu juste
- $\triangleright 2^7 = 128 \Rightarrow$  plutôt satisfaisant.
- $\triangleright 2^8 = 256 \Rightarrow$  le choix qui a été fait.



## Regroupement par octet ou «byte»

10

On regroupe les bits de mémoires par groupe de 8, appelé octet ou «byte» :



Il faut :

- ▷ 1 bit de numérotation pour 2 octets ou 16 bits de mémoire ;
- ▷ 2 bit de numérotation pour 4 octets ou 32 bits de mémoire ;
- ▷ 3 bits de numérotation pour 8 octets ou 64 bits de mémoire ;
- ▷ 4 bits de numérotation pour 16 octets ou 128 bits de mémoire ;
- ▷ 8 bits de numérotation pour 256 octets ou 1024 bits de mémoire ;
- ▷ 16 bits de numérotation pour 65536 octets ou 64 Kilo-octets de mémoires ;

### Attention

En informatique, «kilo» représente 1024 lorsqu'il s'agit de manipulation dans l'ordinateur.

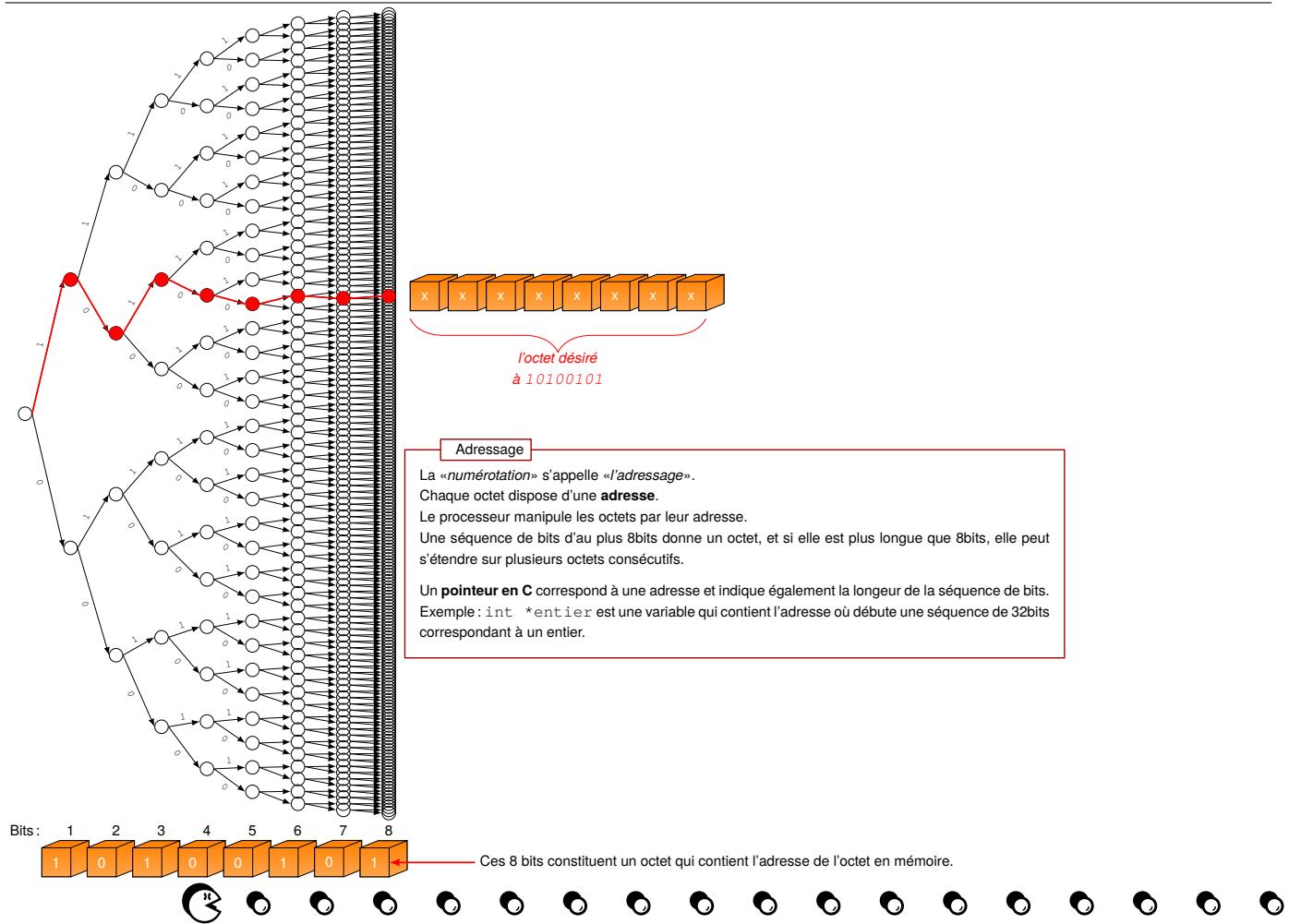
Parce que  $2^{10} = 1024$ .

Dans le SI, «*système international*», «kilo» signifie 1000 et **kibi** signifie 1024.

C'est l'unité utilisée pour les débits réseaux :  $20 \text{ Mbits} \Rightarrow 10^6 \text{ bits}$  ou la capacité de stockage.

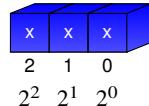


## Exemple : Numérotation sur un octet, ou «*adressage sur 8bits*»

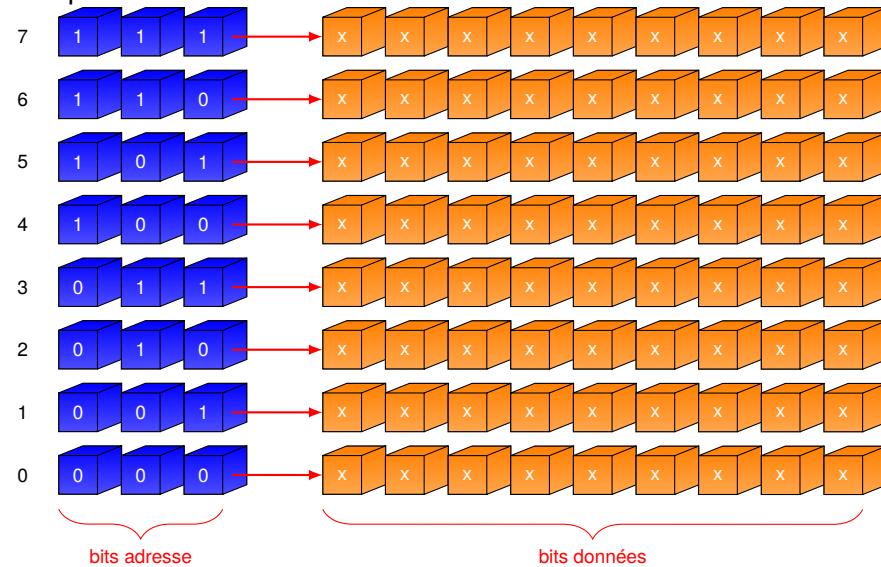


## Exemple : adressage sur 3 bits

- il y a  $2^3 = 8$  octets adressables ;
- chaque adresse peut être codée sur 3 bits :



Ce qui donne une mémoire de **8 octets** ou 64 bits :



Du coup, l'adressage est toujours le même ?  
Non...notion de «page»

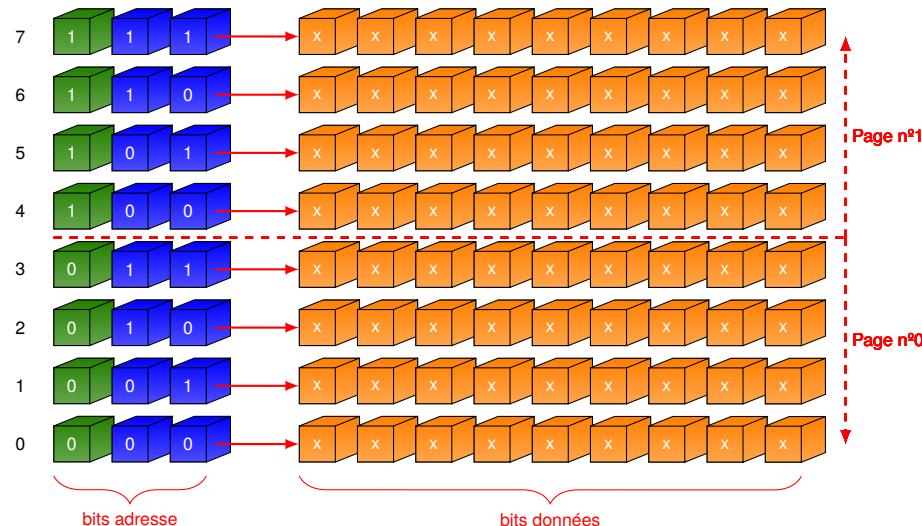


### Exemple d'adressage sur 3 bits avec un bit de page et 2bits d'adresse dans la page

- ▷ une «page» correspond à tous les octets adressables avec un préfixe d'adresse donné ;



- ▷ il est possible de numérotter les pages :



adresses de la page 0 :	0 00	0 01	0 10	0 11
adresses de la page 1 :	1 00	1 01	1 10	1 11

On parle aussi de l'«adresse x dans la page n».

Exemple : l'adresse 10 dans la page 1 correspond à l'adresse complète 110



# Exemple le processeur 8bits 6502



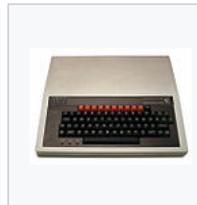
- processeur développé par Chuck Peddle pour la société MOS Technology ;
- introduit en 1975 ;
- très populaire :



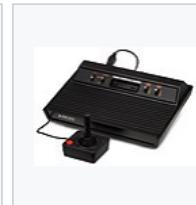
Apple IIe



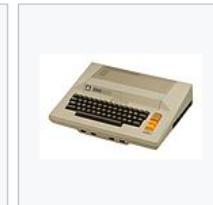
Commodore PET



BBC Micro



Atari 2600



Atari 800



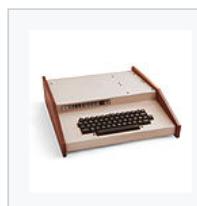
Commodore VIC-20



Commodore 64



Family Computer



Ohio Scientific Challenger 4P

Tamagotchi digital pet<sup>[53]</sup>

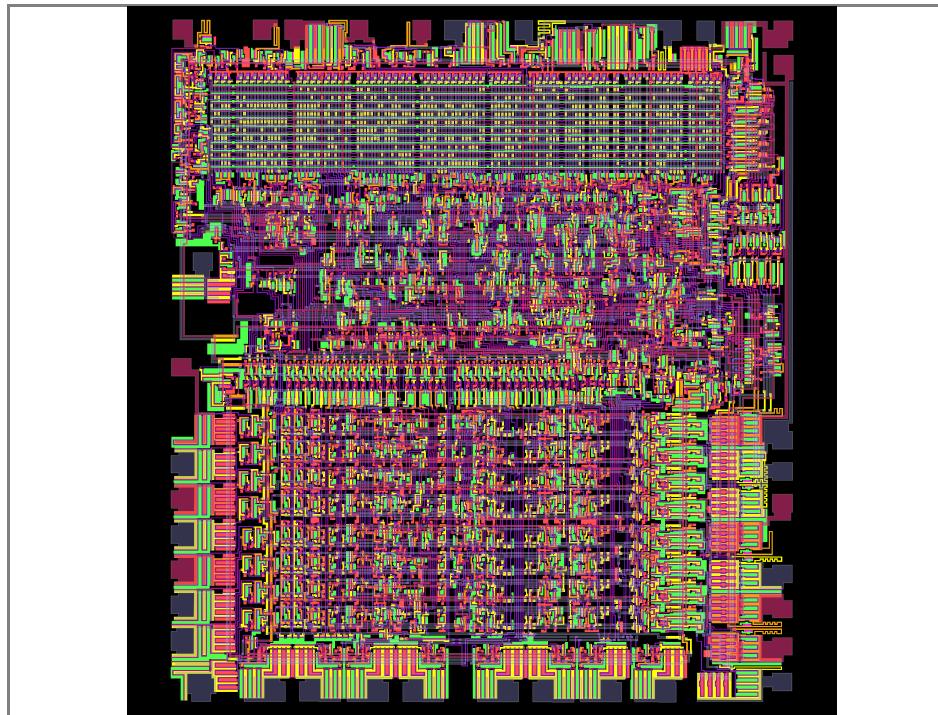
Atari Lynx

- toujours en vente et utilisé dans les **systèmes embarqués** ;
- processeur 8bits, avec un bus d'adresse sur 16bits et «little-endian», cadencé de 1 à 2 MHz



# Le processeur 6502 : représentation visuelle

[FAQ](#) [Blog](#) [Links](#) [Source](#) [easy6502 assembler](#) [mass:werk disassembler](#)



Use 'z' or '>' to zoom in, 'x' or '<' to zoom out, click to probe signals and drag to pan.

Show:  (diffusion)  (grounded diffusion)  (powered diffusion)  (polysilicon)  (metal)  (protection)



```
halfcyc:372 phi:0:0 AB:0015 D:69 RnW:1
PC:0015 A:12 X:07 Y:f9 SP:fb nv-BdIzc
Hz: 3.3 Exec: SEC(T0+T2)
```

```
0000: a9 00 20 10 00 4c 02 00 00 00 00 00 00 00 00 00
0010: e8 88 e6 0f 38 69 02 60 00 00 00 00 00 00 00 00
0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

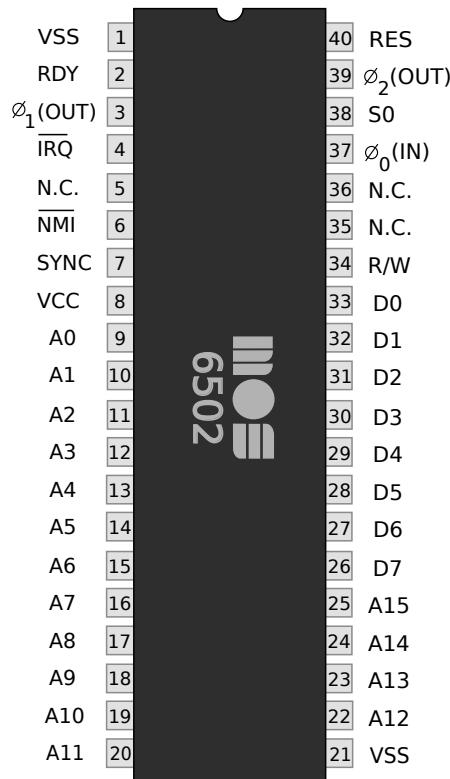
@AABBCCDDEEFFG

[

cycle ab db rw Fetch pc a x y s p

<http://www.visual6502.org/JSSim/expert.html>

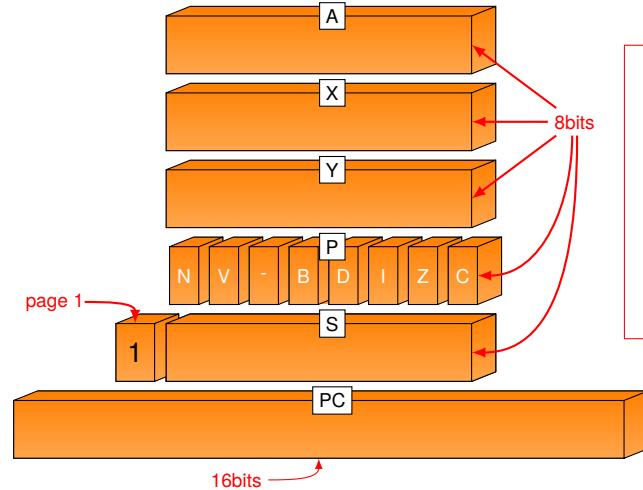




- ▷ Accès à la mémoire :
  - $A_0, \dots, A_{15}$  : 16 bits d'adresse ;
  - $D_0, \dots, D_7$  : 8 bits de données ;
  - $R/W$ : indique si c'est une opération de lecture ou d'écriture ;
- ▷ Interactions avec l'extérieur :
  - *Sync* : signal d'horloge : ryhtme le travail du processeur ;
  - *NMI* : «*Non Maskable Interruption*» : signal d'interruption ;
  - *RES* : «*reset*», réinitialise l'état du processeur et, si maintenue, le bloque ;

# Le processeur 6502

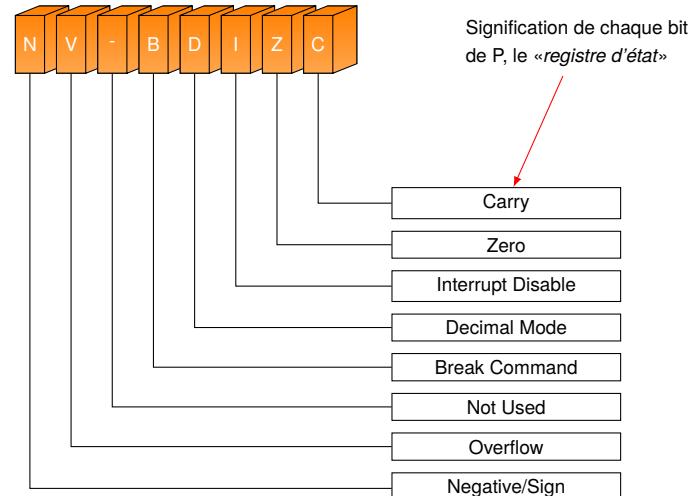
19



Registres	
A «Accumulator»	stockage depuis ou vers l'ALU
X & Y «Index register»	utilisés dans certaines instructions pour calculer une adresse par décalage : adresse+X
P «Processor status register»	chaque bit indique 1 état suite à l'exécution de l'instruction : nombre positif, nul, etc.
S «Stack pointer»	contient l'adresse du dernier octet dans la pile le bit de préfixe à 1 place la pile sur la seconde page
PC «Program counter»	indique l'adresse de la prochaine instruction à exécuter

## Explications du registre d'état

Carry	indique un bit de retenu après opération de l'ALU (9 <sup>ème</sup> bit...)
Zero	la valeur de X, Y ou A est devenue zéro
oOverflow	dépassement de capacité lors d'opération sur des nombres signés
Negative	vrai si le bit de rang 7 est à 1



## Quelques instructions du 6502

20

mode	opérande
immédiat	la donnée
absolu	n'importe quelle adresse
page zéro	un octet correspondant au second octet d'adresse, le premier est fixé à zéro
indexé X	adresse+registre X
indexé Y	adresse+registre Y
implicite	pas d'opérande
relatif	un octet relatif en complément à deux, de -128 à 127

Chaque instruction est **codée sur un octet** en fonction du mode choisi et possible.

*Exemple : l'instruction ADC donne l'octet 69 si la valeur à additionner est donnée en paramètre (mode immédiat).*

*69 01 signifie additionner la valeur 1 dans l'accumulateur.*

*Certaines instructions modifient le registre d'état P :*

*Exemple pour faire un saut sur la condition que X soit égal à zéro :*

*CPX \$0 ; compare la valeur du registre X avec 0  $\Rightarrow$  positionne le bit Z qui devient nul si les deux valeurs sont identiques (on fait une soustraction en fait)*

*BEQ 0A ; test la valeur du bit Z : 1 donne vrai et 0 donne faux*

Ins	description	mode adressage					
		immédiat	absolu	page zéro	indexé X	indexé Y	implicite
ADC	ajoute un octet avec le bit de retenu dans l'accumulateur	69	6D	65	7D	79	
BEQ	«Branch if Equal», saut vers une adresse si vrai						F0
BNE	«Branch if Not Equal», saut vers une adresse si faux						D0
CPX	compare avec le registre X	E0	EC	E4			
INX	Incrémente la valeur dans le registre X						E8
INY	Incrémente la valeur dans le registre Y						C8
JMP	«JuMP», saut						
JSR	«Jump to SubRoutine», saut vers un sous-programme	20					
LDA	charge un octet dans le registre A	A9	AD	A5	BD	B9	
LDX	charge un octet dans le registre X	A2	AE	A6		BE	
LDY	charge un octet dans le registre Y	A0	AC	A4	BC		
RTS	«ReTurn from Subroutine», retour d'un sous-programme						60
STA	stocke l'accumulateur à une adresse donnée	8D	85	9D	99		



# Additionner deux nombres en assembleur 6502

21

## Sur 8bits



Le programme assembleur :

```
LDA adresse1 ; charge le nombre stocké à l'adresse 1 dans l'accumulateur  
ADC adresse2 ; additionne le nombre stocké à l'adresse 2 à l'accumulateur  
STA adresse3 ; stocke le contenu de l'accumulateur à l'adresse 3  
RTS ; retourne
```

Les mnémoniques :

AD	adresse1
6D	adresse2
8D	adresse3
60	

## Sur 16bits

Le premier nombre sur deux octets 

W	W1
---	----

Le second nombre sur deux octets 

X	X1
---	----

**Attention** : on est en «LittleEndian»,  
c-à-d avec inversion des octets de la valeur sur 16bits.

	octets		1 <sup>er</sup>	2 <sup>nd</sup>	car
	307	51			
Premier nombre	307	51	1		$307 = 1 * 256 + 51$
Second nombre	764	252	2		$764 = 2 * 256 + 252$

Le programme assembleur :

```
CLC  
LDA adresse W  
ADC adresse X  
STA adresse Y  
LDA adresse W1  
ADC adresse X1  
STA adresse Y1  
LDA #&0  
ADC #&0  
STA adresse Z  
RTS
```

Les mnémoniques :

18	
AD	adresse W
6D	adresse X
8D	adresse Y
AD	adresse W1
6D	adresse X1
8D	adresse Y1
A9	00
69	00
8D	adresse Z
60	

⇒ on utilise le bit de retenue...



## Application d'un xor d'un texte avec un mot de passe

Le programme calcule  $saisie_i \oplus mdp_i$  pour chaque caractère  $i$  de *saisie* et de *mdp*.

```
1 define sortie $200 ; on définit l'adresse de sortie à 0200
2
3 LDA saisie ; on lit la taille de la chaîne saisie
4 STA sortie ; on la reporte dans la chaîne de sortie
5 ADC #$1 ; on incrémente la valeur pour la comparaison utilisée pour arrêter la boucle
6 STA $0 ; on la stocke dans la page zéro
7 LDA mdp ; on lit la taille de la chaîne mdp
8 ADC #$1 ; on incrémente la valeur utilisée pour réinitialiser l'utilisation du mdp
9 STA $1 ; on la stocke dans la page zéro
10
11 LDX #$1 ; on charge la valeur 1 dans le registre X
12 LDY #$1 ; on charge la valeur 1 dans le registre Y
13
14 boucle: ; on définit une étiquette
15     LDA saisie,X ; on charge dans l'accumulateur la valeur à l'adresse saisie+X
16     EOR mdp,Y ; on réalise un xor entre le registre A et la valeur à l'adresse mdp+Y
17     STA sortie,X ; on stocke le résultat à l'adresse sortie+X
18     INX ; on incrémente la valeur contenue dans le registre X
19     CPX $0 ; on compare la valeur de la taille de la chaîne saisie
20     BEQ fin ; si elle est identique, on a fini et on mets l'adresse fin dans le registre PC
21     INY ; on incrémente la valeur contenue dans le registre Y
22     CPY $1 ; on compare avec la valeur de la taille de la chaîne mdp
23     BNE boucle ; si elle n'est pas égale on recommence la boucle en sautant à l'adresse boucle
24     LDY #$1 ; sinon on réinitialise le registre Y à 1
25     JMP boucle ; et on effectue un saut à l'adresse boucle
26 fin: ; étiquette
27     BRK ; instruction d'arrêt
28
29 saisie:
30     dcb 5,$68,$65,$6c,$6c,$6f ;hello
31 mdp:
32     dcb $9,$74,$6f,$70,$73,$65,$63,$72,$65,$74;topsecret
```



# Utilisation du désassembleur

Address	Hexdump	Dissassembly
\$0600	ad 2e 06	LDA \$062e
\$0603	8d 00 02	STA \$0200
\$0606	69 01	ADC #\$01
\$0608	85 00	STA \$00
\$060a	ad 3c 06	LDA \$063c
\$060d	69 01	ADC #\$01
\$060f	85 01	STA \$01
\$0611	a2 01	LDX #\$01
\$0613	a0 01	LDY #\$01
\$0615	bd 2e 06	LDA \$062e, X
\$0618	59 3c 06	EOR \$063c, Y
\$061b	9d 00 02	STA \$0200, X
\$061e	e8	INX
\$061f	e4 00	CPX \$00
\$0621	f0 0a	BEQ \$062d
\$0623	c8	INY
\$0624	c4 01	CPY \$01
\$0626	d0 ed	BNE \$0615
\$0628	a0 01	LDY #\$01
\$062a	4c 15 06	JMP \$0615
\$062d	00	BRK
\$062e	0d 68 65	ORA \$6568
\$0631	6c 6c 6f	JMP (\$6f6c)
\$0634	20 62 6f	JSR \$6f62
\$0637	6e 6a 6f	ROR \$6f6a
\$063a	75 42	ADC \$42, X
\$063c	09 74	ORA #\$74
\$063e	6f	???
\$063f	70 73	BVS \$06b4
\$0641	65 63	ADC \$63
\$0643	72	???
\$0644	65 74	ADC \$74

0600:	ad 2e 06 8d 00 02 69 01 85 00 ad 3c 06 69 01 85
0610:	01 a2 01 a0 01 bd 2e 06 59 3c 06 9d 00 02 e8 e4
0620:	00 f0 0a c8 c4 01 d0 ed a0 01 4c 15 06 00 0d 68
0630:	65 6c 6c 6f 20 62 6f 6e 6a 6f 75 72 09 74 6f 70
0640:	73 65 63 72 65 74

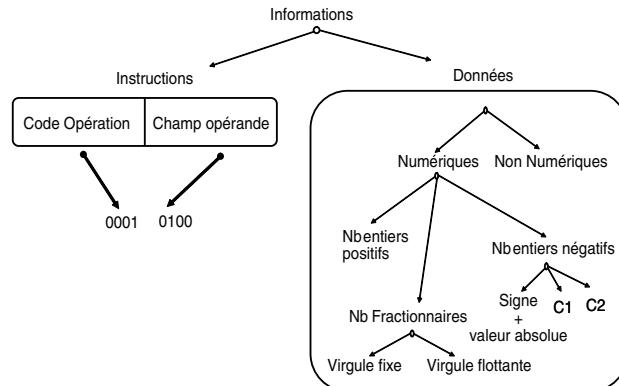
On note que :

062e	adresse de la chaîne saisie
063c	adresse de la chaîne mdp
062d	adresse de l'instruction brk
062e	le désassembleur trouve des instructions dans le contenu de la chaîne saisie ⇒ Interprétation automatique erronée
063e	Interprétation automatique impossible,
0643	il n'y a pas d'instruction reconnue



# Codage de l'information

Les différentes natures d'information :



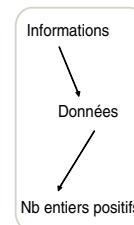
- données et instructions sont codées sur un ou plusieurs mots mémoires ;
- instructions machine :
  - ◊ propre à chaque processeur ;
  - ◊ «code opération» : sélectionne l'instruction machine, sa taille dépend du «jeu d'instructions» du processeur ;
  - ◊ «champ opérande» : indique l'adresse ou la donnée que manipule l'instruction

Bit	0	1	Alphabet binaire
x	2 combinaisons	0	1
xx	4 combinaisons	0	0
		0	1
		1	0
		1	1
xxx	8 combinaisons	0	0 0
		0	0 1
		0	1 0
		0	1 1
		1	0 0
		1	0 1
		1	1 0
		1	1 1

n bits donnent  $2^n$  combinaisons différentes

n	$2^n$
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Le codage binaire :



Octet 01100101  
Position du bit → 7 6 5 4 3 2 1 0  
0 1 1 0 0 1 0 1

Valeur de l'octet :  
 $0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$   
 Soit 101 en base 10

La valeur maximale d'un entier sur p bits est  $2^p - 1$



# Codage de l'information : la table ASCII

25

La table «ASCII», «American Standard Code for Information Interchange» a été définie en 1960.

\* comporte :

- ◊ sur 7 bits : US-ASCII ;
- ◊ des caractères de contrôle : valeur de 0 à 31, contrôle réseau et impression ;
- ◊ des caractères alphabétiques, numériques et ponctuation : valeur de 32 à 127 ;

		b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>		0	0	0	1	0	1	1	0	1	0	1	1	0	1	
		Bits				Column →				Row ↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D
		0	0	0	0	0	0	0	NUL	DLE	SP	0	@	P	‘	p	‘	’	’	’	’	’	’	’
0	0	0	0	1	1	0	0	SOH	DC1	!	1	1	A	Q	a	q	‘	’	’	’	’	’	’	’
0	0	1	0	0	2	0	0	STX	DC2	”	2	2	B	R	b	r	‘	’	’	’	’	’	’	’
0	0	1	1	3	ETX	DC3	0	#	3	C	S	c	s	‘	’	’	’	’	’	’	’	’	’	’
0	1	0	0	4	EOT	DC4	0	\$	4	D	T	d	t	‘	’	’	’	’	’	’	’	’	’	’
0	1	0	1	5	ENQ	NAK	0	%	5	E	U	e	u	‘	’	’	’	’	’	’	’	’	’	’
0	1	1	0	6	ACK	SYN	0	&	6	F	V	f	v	‘	’	’	’	’	’	’	’	’	’	’
0	1	1	1	7	BEL	ETB	0	‘	7	G	W	g	w	‘	’	’	’	’	’	’	’	’	’	’
1	0	0	0	8	BS	CAN	0	(	8	H	X	h	x	‘	’	’	’	’	’	’	’	’	’	’
1	0	0	1	9	HT	EM	0	)	9	I	Y	i	y	‘	’	’	’	’	’	’	’	’	’	’
1	0	1	0	10	LF	SUB	0	*	:	J	Z	j	z	‘	’	’	’	’	’	’	’	’	’	’
1	0	1	1	11	VT	ESC	0	+	:	K	[	k	{	‘	’	’	’	’	’	’	’	’	’	’
1	1	0	0	12	FF	FC	0	,	<	L	\	l		‘	’	’	’	’	’	’	’	’	’	
1	1	0	1	13	CR	GS	0	-	=	M	]	m	}	‘	’	’	’	’	’	’	’	’	’	’
1	1	1	0	14	SO	RS	0	.	>	N	^	n	~	‘	’	’	’	’	’	’	’	’	’	
1	1	1	1	15	SI	US	0	/	?	O	_	o	DEL	‘	’	’	’	’	’	’	’	’	’	’

\* étendue sur 8 bits :

- ◊ caractères accentués de 128 à 255 ;

128	ő	144	é	160	á	176	í	192	ł	208	ł	224	ó	240	í
129	ű	145	æ	161	í	177	ő	193	ł	209	ł	225	ö	241	å
130	é	146	œ	162	ó	178	ő	194	ł	210	ł	226	ł	242	ę
131	ᾶ	147	ð	163	ú	179	ł	195	ł	211	ł	227	ñ	243	đ
132	ᾶ	148	ð	164	ñ	180	ł	196	ł	212	ł	228	ñ	244	đ
133	ᾶ	149	ð	165	ñ	181	ł	197	ł	213	ł	229	ø	245	ž
134	ᾶ	150	û	166	ׁ	182	ׁ	198	ׁ	214	ׁ	230	ׁ	246	ׁ
135	݂	151	݂	167	݂	183	݂	199	݂	215	݂	231	݂	247	݂
136	݂	152	݂	168	݂	184	݂	200	݂	216	݂	232	݂	248	݂
137	݂	153	݂	169	݂	185	݂	201	݂	217	݂	233	݂	249	݂
138	݂	154	݂	170	݂	186	݂	202	݂	218	݂	234	݂	250	݂
139	݂	155	݂	171	݂	187	݂	203	݂	219	݂	235	݂	251	݂
140	݂	156	݂	172	݂	188	݂	204	݂	220	݂	236	݂	252	݂
141	݂	157	݂	173	݂	189	݂	205	݂	221	݂	237	݂	253	݂
142	݂	158	݂	174	݂	190	݂	206	݂	222	݂	238	݂	254	݂
143	݂	159	݂	175	݂	191	݂	207	݂	223	݂	239	݂	255	݂

Source : [www.LookupTables.com](http://www.LookupTables.com)



Soit le programme suivant et le résultat de son exécution :

```
#include <stdio.h>

int main()
{
    char caractere = 'A';
    short int valeur_16_bit = 513;
    ❶ char *un_octet = (char *) &valeur_16_bit;
    char valeur_signee = 213;
    unsigned char valeur_non_signee = 213;
    printf("Variable caractere\n");
    printf(" sous forme de caractere : %c\n", caractere);
    printf(" de valeur hexadecimale : %x\n", caractere);
    printf(" de valeur decimal : %d\n", caractere);
    printf("Variable valeur_16_bit\n");
    printf(" elle tient sur %ld octets\n", sizeof(short int));
    printf(" sous forme decimal : %d\n", valeur_16_bit);
    printf(" sous forme hexadecimale : %x\n", valeur_16_bit);
    ❷ printf(" Premier octet de la variable valeur_16_bit\n");
    printf(" sous forme decimal : %d\n", *un_octet);
    printf(" sous forme hexadecimale : %x\n", *un_octet);
    ❸ printf(" Second octet de la variable valeur_16_bit\n");
    printf(" sous forme decimal : %d\n", *(un_octet+1));
    printf(" sous forme hexadecimale : %x\n", *(un_octet+1));
    printf(" BigEndian ou LittleEndian ?\n");
    printf("Variable valeur_signee en entier : %d\n", valeur_signee);
    printf("Variable valeur_non_signee en entier : %d\n", valeur_non_signee);
    printf(" signee %x et non signee %x\n", valeur_signee, valeur_non_si
gnnee);
}
```

```

xterm
pef@darkstar:~/tmp$ gcc -o tester_format res_sys_format.c
pef@darkstar:~/tmp$ ./tester_format
Variable caractere
sous forme de caractere : A
de valeur hexadecimale : 41
de valeur decimal : 65
Variable valeur_16_bit
elle tient sur 2 octets
sous forme decimal : 513
sous forme hexadecimale : 201
Premier octet de la variable valeur_16_bit
    sous forme decimal : 1
    sous forme hexadecimale : 1
Second octet de la variable valeur_16_bit
    sous forme decimal : 2
    sous forme hexadecimale : 2
BigEndian ou LittleEndian ?
Variable valeur_signee en entier : -43
Variable valeur_non_signee en entier : 213
    signee fffffd5 et non signee
d5

```

*Quelle est la taille par défaut d'un int ?  
4 octets d'après le dernier affichage  
fffffd5*

- ❶ ⇒ crée un pointeur d'octet et le faire pointer sur le premier octet de l'entier qui en compte deux ;
- ❷ ⇒ affiche la valeur pointée, c-a-d la valeur du premier octet ;
- ❸ ⇒ affiche celle du second octet en utilisant «l'arithmétique des pointeurs» : décalage de l'adresse de une fois la taille du type pointé, c-à-d, ici, de un octet (type du pointeur `char *`).



## L'octet

La valeur 70 représentée en binaire  $\Rightarrow$

0	1	0	0	0	1	1	0
128	64	32	16	8	4	2	1

Cet octet peut ensuite être associé avec un caractère au travers de la table «ASCII» ou du codage UTF-8.

## Les entiers relatifs

Suivant la taille de la représentation machine :

- on associe le bit de «poids fort», c-à-d associé à la puissance de 2 la plus grande, au signe ;
- on choisit un codage facilitant l'arithmétique binaire ;
- on peut représenter des valeurs comprises entre  $-2^{\text{nbre\_bits}-1}$  et  $2^{\text{nbre\_bits}-1} - 1$

Exemple sur une représentation suivant une taille de 8bits,  $\text{nbre\_bits} = 8$ :

0	1	0	0	0	1	1	0	= 70
0	1	1	1	1	1	1	1	= 127
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	= -1
1	1	1	1	1	1	1	0	= -2
-128	64	32	16	8	4	2	1	

- \* on ne code qu'une seule fois la valeur 0 :  
*il n'y a pas +0 et -0*
- \* les opérations arithmétiques sont simplifiées :  
 $3 - 4 = 3 + (-128 + 124) = -128 + 127 = -1$

-128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	0	0

1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	1	

La conversion d'un entier négatif en sa représentation correspond à inverser les bits de sa valeur absolue et d'ajouter 1 au résultat ou à représenter  $2^{\text{nbre\_bits}} - |x|$ . Cette opération s'appelle «complément à  $2^n$ ».



## Réels

Un réel est manipulé par la machine sous une forme approchée appelée «nombre à virgule flottante» ou «*float*» constitué de :

- ▷ un signe ;
- ▷ une mantisse, c-à-d un nombre fixe de bits significatifs ;
- ▷ un exposant suivant une base 2, 10 ou 16 (hexadécimal) ;

Exemple :

- \* en base 10 : 13,625 ou  $13625 * 10^{-3}$ , la mantisse est 13625 ;
- \* en base 2 :

Puissance de 2	3	2	1	0	,	-1	-2	-3
valeur	8	4	2	1	,	0,5	0,25	0,125
bit	1	1	0	1	,	1	0	1

$$\begin{aligned}1101,101_2 &= 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3} \\&= 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 + 1 * 0,5 + 0 * 0,25 + 1 * 0,125 \\&= 8 + 4 + 0 + 1 + 0,5 + 0 + 0,125 \\&= 13,625_{10}\end{aligned}$$

Ce qui donne  $1101101 * 2^{-3}$ , la mantisse est 1101101.



```
xterm
>>> float.hex(1.72)
'0x1.b851eb851eb85p+0'

>>> float.hex(0.3)
'0x1.333333333333p-2'

>>> float.hex(0.1)
'0x1.999999999999ap-4'
```

Sous Python :

- ▷ `float.hex()` : donne la représentation hexadécimal du nombre à virgule ;
- ▷ `float.fromhex()` : retrouve le nombre à virgule à partir de sa représentation hexadécimale.

D'après l'exemple de la doc Python :

`0x3.a7p10` est égal à  $(3 + 10./16 + 7./16^{*}2) * 2.0^{*}10$ , soit la valeur 3740.0

Où en notation humaine :

$$(3+10*\frac{1}{16}+7*\frac{1}{16^2})*2^{10} = (3+10*\frac{1}{16}+7*\frac{1}{16^2})*2^4*2^4*2^2 = (3+10*\frac{1}{16}+7*\frac{1}{16^2})*16*16*4 = \\ (3 * 16 + 10 + 7 * \frac{1}{16}) * 16 * 4 = (3 * 16 * 16 + 10 * 16 + 7) * 4 = (768 + 160 + 7) * 4 = 3740$$

On notera que 0.1 est égale à :

$$(1 + 9 * \frac{1}{16} + 9 * \frac{1}{16^2} + 9 * \frac{1}{16^3} + ... 9 * \frac{1}{16^{12}} + 10 * \frac{1}{16^{13}}) * 2^{-4}$$

Ce qui conduit à une approximation de cette valeur :

0,100000000000000055511151231257827021181583404541015625



- on associe un bit pour exprimer si le nombre est positif ou négatif : le bit de «poids fort» situé à gauche ;
- on fixe le nombre de bits alloués à la représentation de l'exposant :
  - ◊ on simplifie le codage de cet exposant, en exprimant différemment son signe :
    - \* on décide de ne pas pas exprimer le signe par l'utilisation d'un bit, mais en le déduisant par une opération de «décalage» :  $\text{exposant\_décalé} = \text{exposant} + \text{décalage}$
    - Ainsi, pour un exposant variant de +100 à -100, si on choisit un décalage de 100, la valeur de l'exposant décalé ira de 0 à 200.*
    - \* on calcule l'espace de valeurs possibles suivant le nombre fixe de bits associés à l'exposant ;  
Exemple pour un exposant sur 8 bits :  $2^{\text{nbre\_bits\_exposant}} = 256$
    - \* on choisit une valeur particulière de cet espace,  $2^{\text{nbre\_bits\_exposant}} - 1$ , pour exprimer :
      - ▷ la valeur infini :  $\infty$  avec une mantisse nulle ;
      - ▷ une valeur d'erreur : *NaN*, «not a number» avec une mantisse non nulle ;
    - \* on utilise la valeur zéro de l'exposant pour exprimer :
      - ▷ le nombre 0 : exposant et mantisse nuls ;
      - ▷ des nombres «dénormalisés» : exposant nul mais mantisse non nulle ;
    - \* on divise cet espace de valeurs en deux pour exprimer au choix, un exposant positif ou négatif ;  
Exemple pour un exposant sur 8 bits :  $2^{\text{nbre\_bits\_exposant}-1} = 128$
    - \* on détermine la valeur du décalage compte tenu de la taille de l'espace de valeurs restant :  
 $\text{décalage} = 2^{\text{nbre\_bits\_exposant}-1} - 1$   
Exemple pour un exposant sur 8 bits :  $\text{décalage} = 2^7 - 1 = 127$   
l'exposant peut varier alors de :
      - ▷ +127, ce qui donne  $\text{exposant\_décalé} = 127 + 127 = 254$  la valeur maximale possible ;
      - ▷ à -126, ce qui donne  $\text{exposant\_décalé} = -126 + 127 = 1$  la valeur minimale possible ;



### La norme IEEE754 facilite le traitement par un ordinateur

Les comparaisons entre deux nombres à virgule flottante est facilité : elle est similaire, dans la plupart des cas, à celle réalisée sur des entiers.

### Représentation des valeurs et stockage de la mantisse

Les nombres représentés suivant cette norme doivent obligatoire avoir un bit à 1 au début de leur représentation :  $0,00101_2 = 1,01_2 * 2^{-3}$  Ce qui permet :

- de «supprimer» du stockage ce premier bit à 1 (il devient implicite) ;
- d'augmenter de 1 bit la capacité de représentation.

### Calcul de la capacité de représentation numérique

La valeur est calculée suivant la formule suivante :

$$(-1)^{\text{bit\_signe}} * \text{mantisse} * 2^{\text{exposant-décalage}}$$

On distingue deux «précisions» différentes pour un codage sur 32 bits ou 64 bits :

▷ simple précision :

- valeur la plus proche de zéro :  $\pm 2^{-126} \cong \pm 1,175494351 * 10^{-38}$
- valeur la plus éloignée de zéro :  $\pm (2 - 2^{23}) * 2^{127} \cong 3,4028235 * 10^{38}$

▷ double précision :

- valeur la plus proche de zéro :  $\pm 2^{?1022} \cong \pm 2,2250738585072020 * 10^{-308}$
- valeur la plus éloignée de zéro :  $\pm (2^{-1024} - 2^{971}) \cong \pm 1,7976931348623157 * 10^{308}$

	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^{\text{bit\_signe}} * \text{mantisse} * 2^{\text{exposant}-127}$	24 bits	7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^{\text{bit\_signe}} * \text{mantisse} * 2^{\text{exposant}-1023}$	53 bits	16



### Les nombres «dénormalisés»

Les nombres en représentation «dénormalisés» :

- permettent d'exprimer des valeurs plus petites que celles exprimables dans la représentation normalisée ;
- ne disposent pas de «bit implicite» à 1 en début de la représentation contrairement à la version normalisée ;
- ne possèdent pas autant de chiffres significatifs que la représentation normalisé ;
- permettent une perte graduelle de précision quand les résultats de calculs sont trop proches de zéro pour être représentés par la version normalisée.

L'exposant de ces nombres est à zéro ce qui donne un exposant de -126 en simple précision et de -1022 en double précision.

Ce qui donne :

- ▷ en simple précision, plus petit nombre différent de zéro et plus grand nombre négatif différent de zéro :  $\pm 2^{-1074} \approx \pm 2^{-149} \approx \pm 1,4012985 * 10^{-45}$  ;
- ▷ en double précision, plus petit nombre différent de zéro et plus grand nombre négatif différent de zéro :  $\pm 2^{-1074} \approx \pm 4,9406564584124654 * 10^{324}$  ;

### Les valeurs $\infty$ et $NaN$

$x \div 0 = \pm\infty$
$0 \div 0 = NaN$
$x \div \infty = \pm 0$
$\infty \div \infty = NaN$
$(+\infty) + (+\infty) = +\infty$
$(+\infty) - (+\infty) = NaN$



## La capacité des valeurs numériques

Les valeurs entières ne sont pas limitées en dimension.

Calcul de la clé du numéro de sécurité sociale sur 13 chiffres :

```
In [17]: numero_secu=1700187001001
```

```
In [18]: 97-(numero_secu%97)
```

```
Out[18]: 73
```

Les nombres à virgule flottante sont limités suivant la «taille des mots» processeur :

- ▷ pour 32 bits la limite est  $\pm 3.4 * 10^{38}$  ;
- ▷ pour 64 bits la limite est  $\pm 1.8 * 10^{308}$  ;

```
In [12]: 9*1.8E307
```

```
Out[12]: 1.62e+308
```

```
In [13]: 10*1.8E307
```

```
Out[13]: inf
```



## Les règles d'associativité et de commutativité

Soit l'équation :  $\frac{(x-x+y)}{y} = \frac{((x-x)+y)}{y} = \frac{(x-(x+y))}{y}$

Calcul sous Python :

- ▷ Pour des valeurs  $x$  et  $y$  proches :

```
>>> x=10000000.0
>>> y=9999999.0
>>> print (((x-x)+y)/y)
1.0
>>> print ((x+(y-x))/y)
1.0
```

- ▷ Pour des valeurs  $x$  et  $y$  éloignées :

```
>>> x=10000000000000.0
>>> y=0.000000000001
>>> print (((x-x)+y)/y)
1.0
>>> print ((x+(y-x))/y)
0.0
```



## Erreurs de calcul :

```
In [106]:  
v=-1.0  
for i in range(0,201):  
    print (math.acos(v))  
    v = v + 0.1
```

```
ValueError      Traceback (most recent call last)
<ipython-input-106-e13af1a75e60> in <module>()
      1 v=-1.0
      2 for i in range(0,201):
----> 3     print acos(v)
      4     v = v + 0.1
```

ValueError: math domain error

3.14159265359

2.69056584179

•

0.643501108793

0.451026811796

2.10734242554e-08

Une autre version :

$$v = -100$$

```
for i in range(0, 201) :
```

```
print (math.acos(v/100.0))  
v = v + 1
```

## Pourquoi ?

```
In [3]: format(0.01, '.30f')
Out[3]: '0.01000000000000000208166817117'
```

```
In [4]: sum([0.01]*100)  
Out[4]: 1.0000000000000007
```

```
In [5]: round(1.0000000000000007, 13)  
Out[5]: 1.0
```

## Explications :

- \* «`0x1.999999999999ap-4`» est un nombre flottant normalisé en double précision exprimé en hexadécimal ;
  - \* «`0x`» : indique la notation hexadécimale ;
  - \* «`1.`» : correspond au bit implicite ;
  - \* «`99 99 99 99 99 99 a`» : la mantisse sur 52 bits ;
  - \* «`p-4`» : correspond à l'exposant  $2^{-4}$

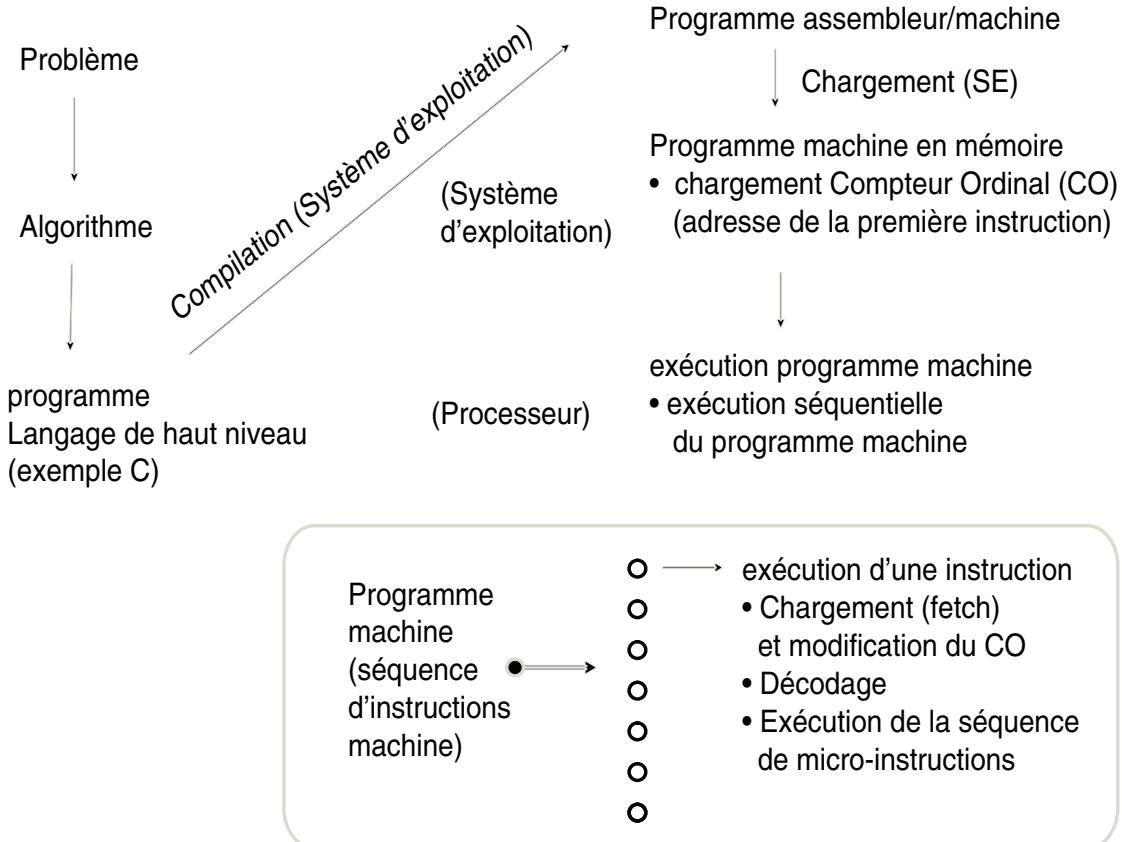
En binaire, cela donne :

```
In [3]: from decimal import Decimal
```

```
In [4]: Decimal(0.1)
```

```
Out[4]: Decimal('0.10000000000000005551115123...')
```

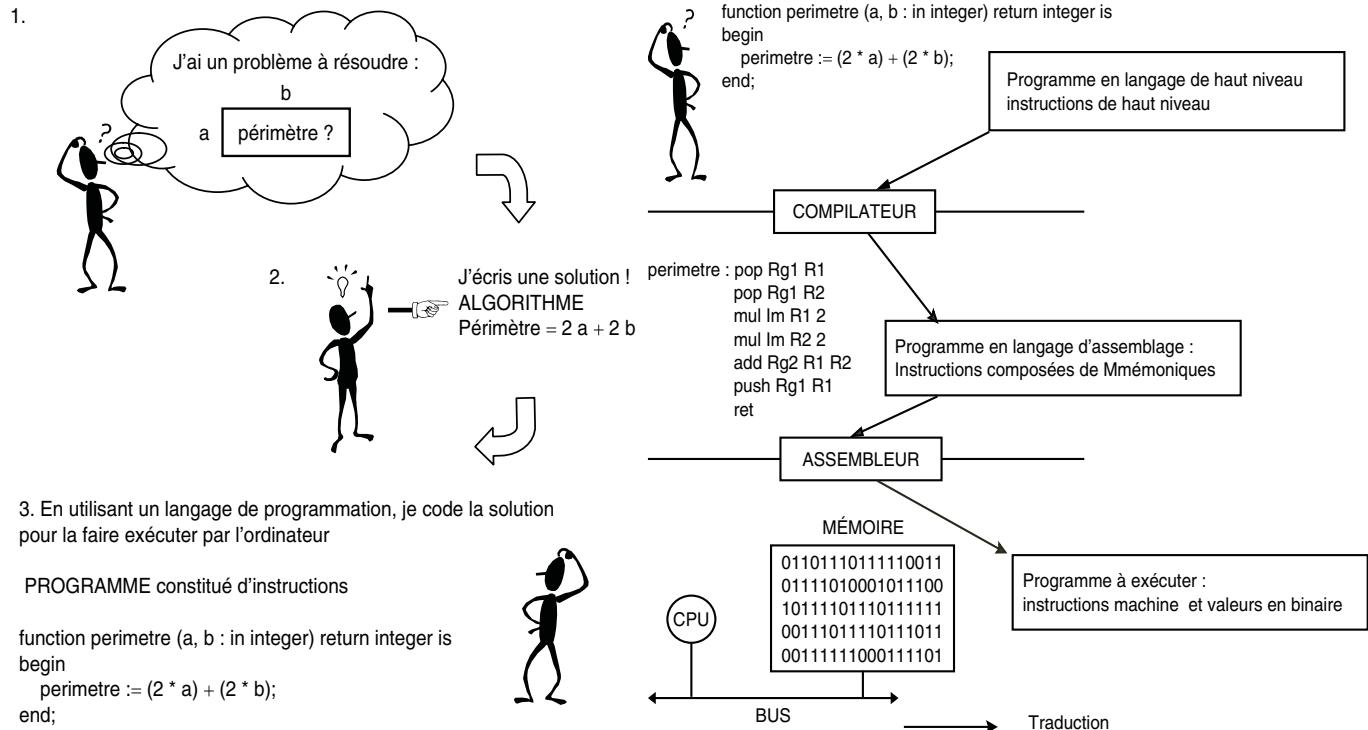




*Un des rôles du Système d'Exploitation, ou «OS», «operating system» est de chargé le programme dans la machine.*



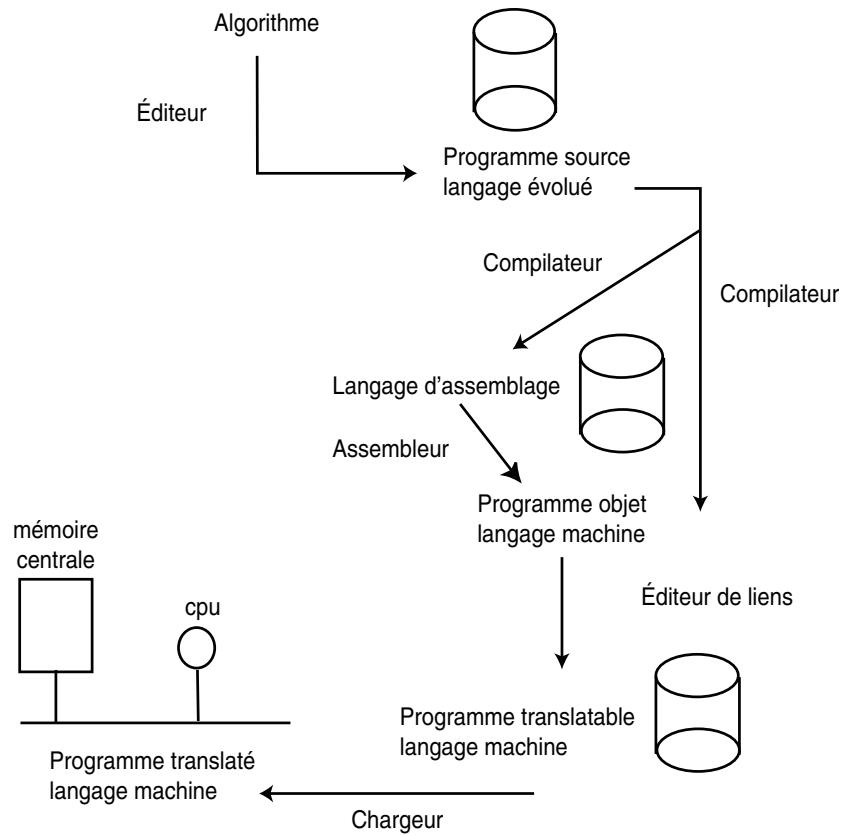
- le problème est exprimé dans un langage de «haut niveau», tel que C, Java, Python etc.
- un compilateur traduit le «source» du langage de haut niveau choisi, en programme en langage d'assemblage ;
- ce langage d'assemblage est traduit en assembleur correspondant au programme exécutable par la machine.



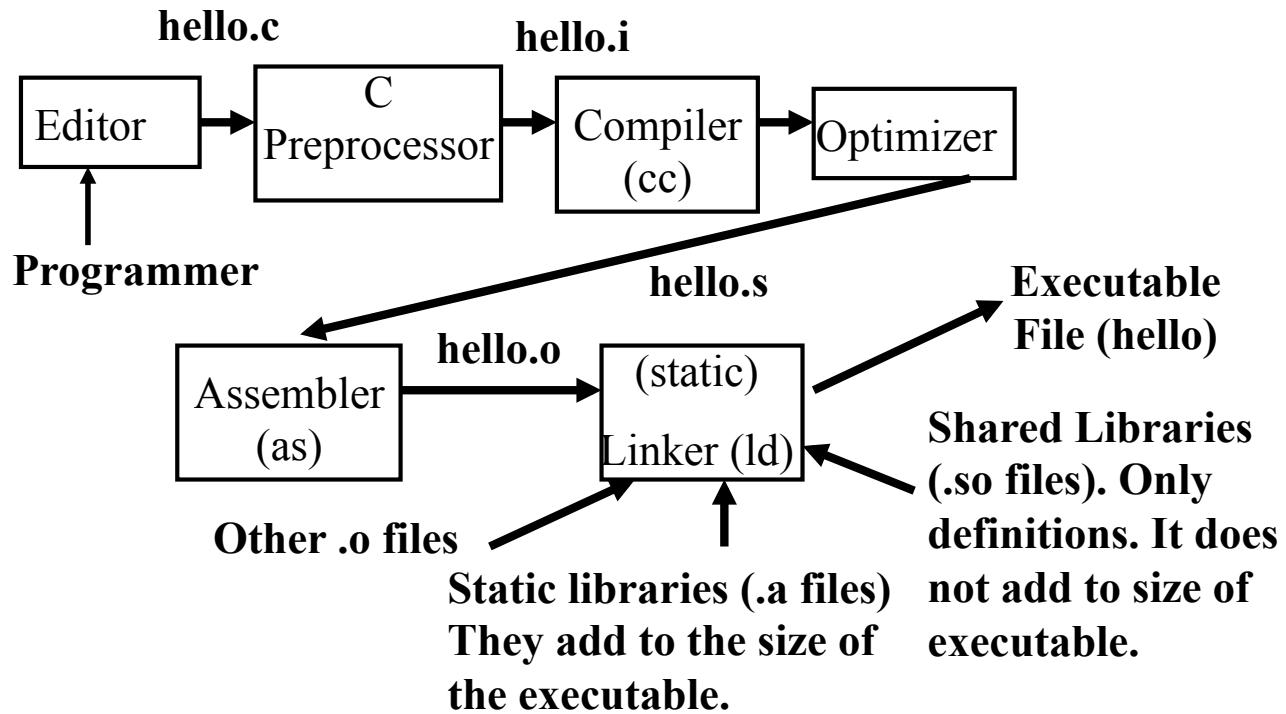
Les «mnémoniques» sont des appellations mémorisables par un humain, correspondant aux instructions machine.

# Du problème à sa résolution : point de vue utilisateur

38



## Les différentes étapes et les différents outils



**Exemple****Le programme suivant :**

```
1 #include <stdio.h>
2 main()
3 {
4     printf("Hello\n");
5 }
```

**Donne après le travail du préprocesseur :**

```
1 $ gcc -E hello.c > hello.i
2         (-E stops compiler after running preprocessor)
3 hello.i:
4     /* Expanded /usr/include/stdio.h */
5     typedef void *__va_list;
6     typedef struct __FILE __FILE;
7     typedef int     ssize_t;
8     struct FILE {...};
9     extern int fprintf(FILE *, const char *, ...);
10    extern int fscanf(FILE *, const char *, ...);
11    extern int printf(const char *, ...);
12 /* and more */
13 main()
14 {
15     printf("Hello\n");
16 }
```

*Le préprocesseur rajoute le contenu des fichiers d'en-tête indiqués dans les directives de compilation «#include» :*

- `#include <...>`: indique qu'il faut rechercher le fichier dans ceux du système, par exemple dans le répertoire `/usr/include`
- `#include "..."`: indique de rechercher le fichier dans le répertoire courant.



```
1 $ gcc -S hello.c      (-S stops compiler after assembly)          ↪
2 hello.s:
3     .file    "hello.c"
4     .section .rodata
5 .LC0:
6     .string   "Hello"
7     .text
8     .globl   main
9     .type    main, @function
10 main:
11 .LFB0:
12     .cfi_startproc
13     pushl   %ebp
14     .cfi_def_cfa_offset 8
15     .cfi_offset 5, -8
16     movl   %esp, %ebp
17     .cfi_def_cfa_register 5
18     andl   $-16, %esp
19     subl   $16, %esp
20     movl   $.LC0, (%esp)
21     call    puts
22     leave
23     .cfi_restore 5
24     .cfi_def_cfa 4, 4
25     ret
26     .cfi_endproc
```

```
28 .LFE0:
29     .size    main, .-main
30     .ident   "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5)"
31     .section .note.GNU-stack, "",@progbits
```

On accède au langage d'assemblage généré par le compilateur.



- gcc -c hello.c crée un «objet», hello.o, c-à-d un fichier contenant les instructions exécutables directement par le processeur;
- l'objet hello.o possède des «symboles», c-à-d des références vers des fonctions, indéfinis comme l'appel à la fonction printf;
- la fonction principale, «main», possède une valeur relative à l'objet qui la contient, «hello.o»

La commande «nm» permet d'afficher les symboles définies dans l'objet :

```
└── xterm ━━━━━━  
$ nm hello.o  
00000000 T main  
          U puts
```

Où l'on constate qu'un PC est «Little endian» :

```
└── xterm ━━━━━━  
$ readelf -h hello.o  
ELF Header:  
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  
  Class: ELF32  
  Data: 2's complement, little endian  
  Version: 1 (current)  
  OS/ABI: UNIX - System V  
  ABI Version: 0  
  Type: REL (Relocatable file)  
  Machine: Intel 80386
```



## Compilation d'un programme : après le «linking»

- gcc -o hello hello.c crée l'exécutable «hello»;
- printf ne possède pas de valeur tant que le programme n'est pas chargé en mémoire (devenu un processus).

xterm

```
$ nm hello
080484bc R _IO_stdin_used
          w _Jv_RegisterClasses
0804a014 A __bss_start
0804a00c D __data_start
08048470 t __do_global_ctors_aux
08048350 t __do_global_dtors_aux
08049f14 d __init_array_end
08049f14 d __init_array_start
08048460 T __libc_csu_fini
080483f0 T __libc_csu_init
          U __libc_start_main@@GLIBC_2.0
0804a014 A _edata
0804a01c A _end
0804849c T _fini
080484b8 R _fp_hw
080482b0 T _init
08048320 T _start
0804a014 b completed.6159
0804a00c W data_start
0804a018 b dtor_idx.6161
080483b0 t frame_dummy
080483d4 T main
          U puts@@GLIBC_2.0
```

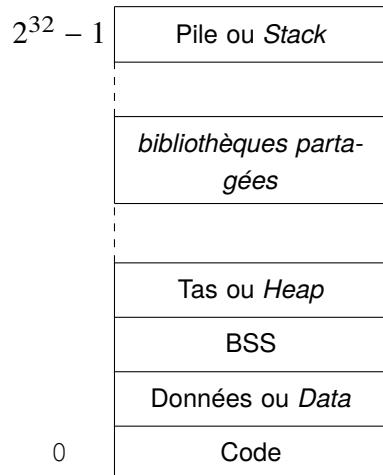
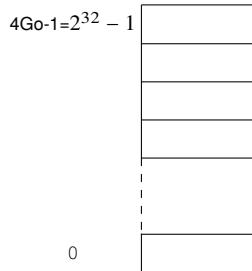
D'après la documentation de la commande «nm» :

- ▷ "A", The symbol's value is absolute, and will not be changed by further linking.
- ▷ "B" "b", The symbol is in the uninitialized data section (known as BSS).
- ▷ "D" "d", The symbol is in the initialized data section.
- ▷ "R" "r", The symbol is in a read only data section.
- ▷ "T" "t", The symbol is in the text (code) section.
- ▷ "U" The symbol is undefined.
- ▷ "W" "w" The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the symbol is determined in a system-specific manner without error. On some systems, uppercase indicates that a default value has been specified.



## 2 La segmentation : les différentes parties d'un processus

Un programme sur un adressage sur 32 bits voit la mémoire de l'adresse 0 à  $2^{32} - 1$ , c-à-d 4Go:



### Les segments :

- \* chaque section possède des droits d'accès différents : read/write/execute
- \* **Code** : les instructions du programme (appelé parfois «*Text*») ;
- \* **Data** : les variables globales initialisées ;
- \* **Bss** : les variables globales non initialisées (elles seront initialisées à 0) ;
- \* **Heap** : mémoire retournée lors d'allocation dynamique avec les fonctions «*malloc/calloc/new*» : *elle croît vers le haut* ;
- \* **Stack** : stocke les variables locales et les adresses de retour : *elle croît vers le bas* ;
- \* **Shared libraries** : les bibliothèques partagées, c-à-d le code des fonctions partagées par plusieurs processus (fonctions d'entrée/sortie, mathématiques, etc.).



## Le fichier «hello\_world.c» :

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello world\n");
6
7     return 0;
8 }
```

## La présence de différents segments :

```
xterm
$ objdump -h hello_world

hello_world:      file format elf32-i386

Sections:
Idx Name      Size    VMA      LMA      File off  Algn
 5 .text      0007ae2c  08048300  08048300  00000300  2**4
              CONTENTS, ALLOC, LOAD, READONLY, CODE
 25 .data     00000c40  080ed060  080ed060  000a4060  2**5
              CONTENTS, ALLOC, LOAD, DATA
 26 .bss      000016d4  080edca0  080edca0  000a4ca0  2**5
              ALLOC
```

## La taille des différents segments présents dans l'exécutable :

```
xterm
$ gcc -o hello_world hello_world.c
$ size hello_world
      text      data       bss      dec      hex   filename
    1131        256         8     1395      573  hello_world
```

## L'option «-static» permet d'inclure les fonctions de bibliothèques dans l'exécutable :

```
xterm
$ gcc -static -o hello_world hello_world.c
$ size hello_world
      text      data       bss      dec      hex   filename
  668942      3316      5892    678150    a5906  hello_world
```



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int our_init_data = 30;
7 int our_noinit_data;
8
9 void our_prints(void)
10{
11    int our_local_data = 1;
12    char c;
13    int *our_dynamic_data = (int *)calloc(1,sizeof(int));
14
15    printf("Pid of the process is = %d\n", getpid());
16    printf("Addresses which fall into:\n");
17    printf("(1) Data segment = %p\n", &our_init_data);
18    printf("(2) BSS segment = %p\n", &our_noinit_data);
19    printf("(3) Code segment = %p\n", &our_prints);
20    printf("(4) Stack segment = %p\n", &our_local_data);
21    printf("(5) Heap segment = %p\n", our_dynamic_data);
22    scanf("%c", &c);
23}

```

```

24 int main()
25{
26    our_prints();
27    return 0;
28}

```

xterm

```
$ ./segments
Pid of the process is = 4697
Addresses which fall into:
1) Data segment = 0x804a020
2) BSS segment = 0x804a02c
3) Code segment = 0x8048494
4) Stack segment = 0xbfe47454
5) Heap segment = 0xd76008
```

xterm

```
$ more /proc/4697/maps
08048000-08049000 r-xp 00000000 08:01 424814 ./segments
08049000-0804a000 r--p 00000000 08:01 424814 ./segments
0804a000-0804b000 rw-p 00001000 08:01 424814 ./segments
0xd76000-0xd97000 rw-p 00000000 00:00 0 [heap]
bfe28000-bfe49000 rw-p 00000000 00:00 0 [stack]
```

xterm

```
$ nm -f sysv segments
Symbols from segments:
Name          Value  Class  Type    Size   Section
main          08048558| T | FUNC|00000012||.text
our_init_data 0804a020| D | OBJECT|00000004||.data
our_noinit_data 0804a02c| B | OBJECT|00000004||.bss
our_prints    08048494| T | FUNC|000000c4||.text
```

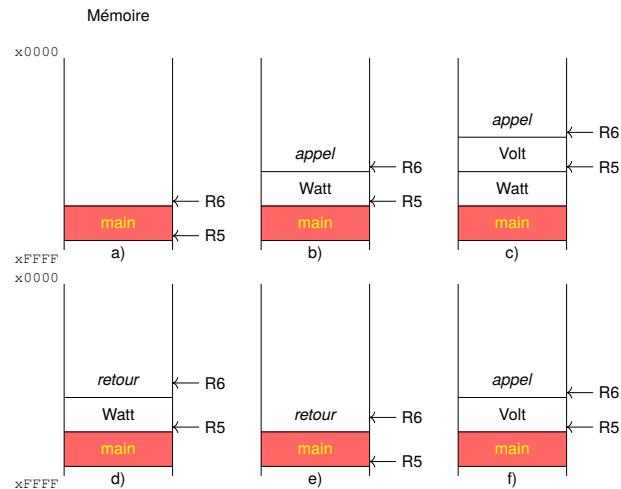
On remarque que sur la sortie du fichier «maps» :

- \* le segment code possède les droits d'exécution ;
- \* les segments bss, data possèdent les droits de lecture/écriture mais **pas d'exécution**.



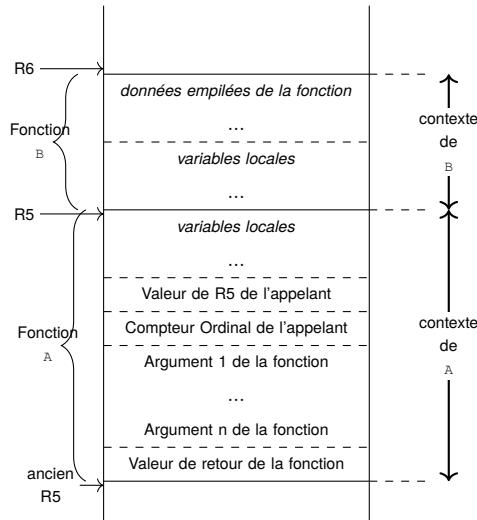
```

1 int main ()
2 {
3     int a = 23;
4     int b = 14;
5     ...
6
7     b = Watt(a); /* appel de Watt */
8     b = Volt(a,b); /* puis de Volt */
9     ...
10 }
11
12 int Watt(int c)
13 {
14     int w = 5;
15     ...
16     w = Volt(w,10); /* appel Volt */
17     ...
18     return w;
19 }
20
21 int Volt(int q, int r)
22 {
23     int k = 3;
24     int m = 6;
25     ...
26     return k+m;
27 }
```



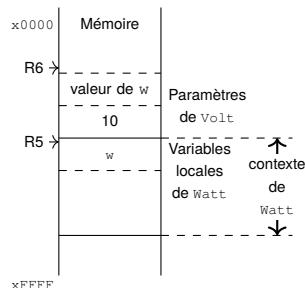
- a. l'état de la pile au démarrage du programme :
  - ◊ registre «R5», «frame pointer» : contient l'adresse du début du contexte courant (appelé «frame»);
  - ◊ registre «R6», «Top of Stack (TOS) pointer» : contient l'adresse du sommet de la pile qui varie par empilement/dépilement;
- b. appel de la fonction «Watt»;
- c. le registre R5 prend la valeur de R6 et pointe sur le «contexte» associé à l'exécution de la fonction, R6 sur le nouveau sommet qui va être décalé pour faire de la place aux arguments, valeur de retour et adresse de retour (empilement);
- d. le processeur saute à l'adresse du code de la fonction appelée;
- e. lors du retour de la fonction, le processeur dépile l'adresse de retour et le registre R6 pointe sur la valeur de retour de la fonction;
- f. nettoyage de la pile pour l'exécution d'une autre fonction (dépilement valeur retour et arguments précédents);
- g. **⇒écrasement** de l'utilisation de la pile de l'ancien appel «Watt», par le nouvel de appel «Volt».





- soit l'état de la pile d'exécution après l'appel de la fonction A :
  - ◊ l'exécution de la fonction A crée un «contexte A» dans la pile :
  - \* la valeur de retour de la fonction A ;
  - \* les arguments : «*Argument 1 à n de la fonction*» ;
  - \* la valeur du «PC» ou «CO» de l'appelant de la fonction A ;
  - Il est nécessaire de mémoriser l'ancienne valeur du CO, puisqu'il sert maintenant à exécuter le code de la fonction A.*
  - \* la valeur du registre de R5 lors de l'exécution de l'appelant : obligatoire pour pouvoir remettre l'appelant dans un état correct lors du retour de la fonction ;
  - \* les variables locales à la fonction ;
  - \* les registres R5 et R6 contiennent maintenant des valeurs relatives au contexte de A ;

- ensuite, la fonction A appelle la fonction B :
    - ◊ l'appel de la fonction B crée le contexte de B dans la pile :
      - \* on empile les données de la fonction en rapport avec A (valeur de retour et arguments) ;
      - \* on empile le CO de l'appelant, c-à-d l'adresse de l'instruction de la fonction A à laquelle il faudra retourner (A ayant appelé B). On mets l'adresse du code de la fonction B dans le CO, ce qui force l'exécution de la fonction B.
  - lors de l'exécution de B, on empile la valeur de R5 pour pouvoir la restaurer et on empile les variables locales de B ;
  - cela crée une «liste chaînée» des différents **appels imbriqués** : il est possible de remonter dans les contextes d'appels en examinant la pile. *Les outils de «débogage» permettent de visualiser et analyser le contenu de la pile d'appel.*
- Le mécanisme «d'exception» présent dans le langage C++, Java, Python etc. permet en cas d'erreur de remonter de contexte de fonction en contexte de fonction appelante à la recherche d'une fonction capable de traiter cette erreur.*

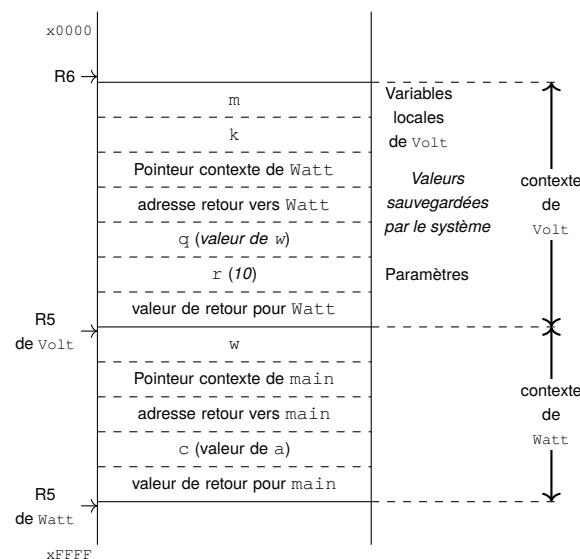


Lors de l'appel de la fonction «Volt», par la fonction «Watt», la valeur contenue dans la variable locale «*w*» de «Watt» est transmise en tant que paramètre à la fonction «Volt».

*Cet appel correspond à la ligne 16 du code source.*

Dans le contexte de la fonction «Volt» :

- o la variable locale «*q*» reçoit la valeur du paramètre transmis, c-à-d la valeur de «*w*» de la fonction «Watt» ;
- o la variable locale «*r*» reçoit la valeur du paramètre transmis, c-à-d la valeur directe 10 ;
- o pour son travail, elle utilise deux variables locales «*k*» et «*m*» ;
- o des sauvegardes des adresses concernant «Watt» :
  - ◊ de retour d'exécution ;
  - ◊ de **restauration de contexte**.

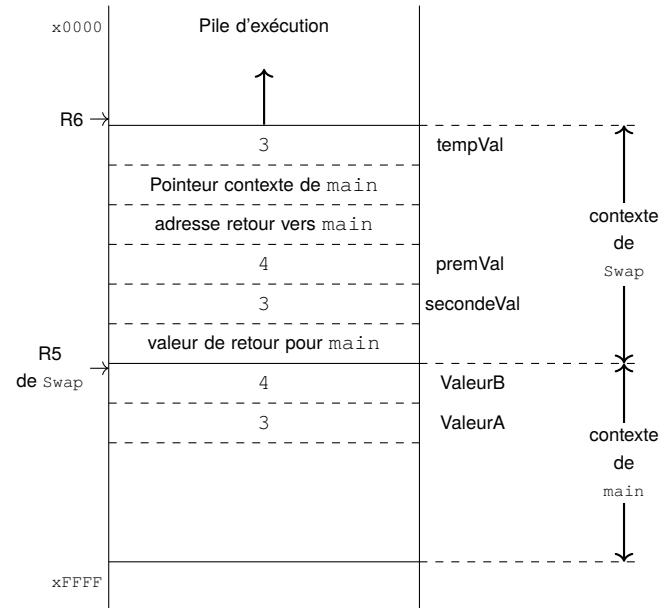


### 3 La pile : appel de fonction et passage de paramètres

50

#### Passage de paramètres par valeur

```
1 #include <stdio.h>
2 void Swap(int PremVal, int secondeVal);
3
4 int main()
5 {
6     int valeurA = 3;
7     int valeurB = 4;
8
9     Swap(valeurA, valeurB);
10    return valeurA; /* retourne 3 */
11 }
12
13 void Swap(int premVal, int secondeVal)
14 {
15     int tempVal; /* conserve PremVal */
16     /* lors de l'échange */
17
18     tempVal = premVal;
19     premVal = secondeVal;
20     secondeVal = tempVal;
21
22 }
```



L'état de la pile avant le retour de la fonction

Que fait le programme ?

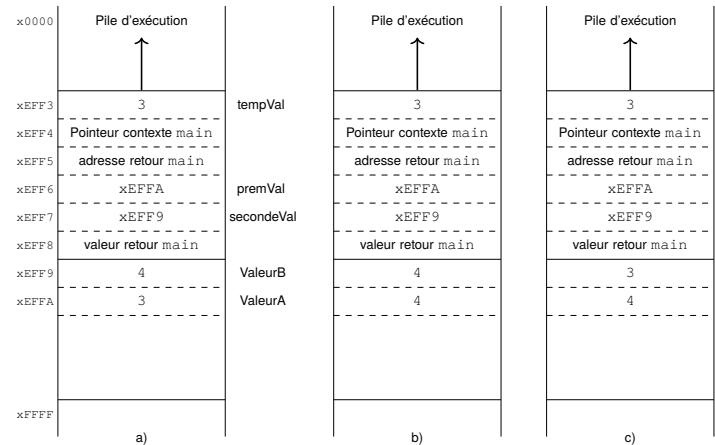
# La pile : appel de fonction et passage de paramètres

## Passage de paramètres par référence

```

1 #include <stdio.h>
2
3 void NewSwap(int *premVal,
4             int *secondeVal);
5
6
7 int main()
8 {
9     int valeurA = 3;
10    int valeurB = 4;
11
12    NewSwap(&valeurA, &valeurB);
13    return valeurA; /* retourne 4 */
14 }
15
16
17 void NewSwap(int *PremVal,
18             int *secondeVal)
19 {
20     int tempVal; /* conserve premVal*/
21     /* lors de l'échange */
22
23     tempVal = *PremVal;
24     *PremVal = *secondeVal;
25     *secondeVal = tempVal;
26     return;
27 }
```

Que fait le programme ?



- ▷ «valeurA» est à l'adresse 0xEFFA;
- ▷ «valeurB» est à l'adresse 0xFF9;
- ▷ «premVal» et «secondeVal» sont des pointeurs :
- ◊ «premVal» contient l'adresse de «valeurA», 0xEFFA;
- ◊ «secondeVal» contient l'adresse de «valeurB», 0xFF9;

- \* Définition de variables locales :

```
int objet;  
int *ptr;
```

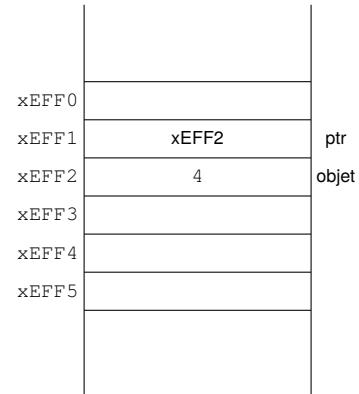
- \* Assignons des valeurs à ces variables :

```
objet = 4;  
ptr = &objet;  
*ptr = *ptr + 1;  
ptr++;
```

- \* La troisième instruction est équivalente à :

```
objet = objet + 1
```

- \* Question : *Quelles sont les valeurs finales de «objet» et «ptr» ?*



## Les tableaux en C

Un tableau est une adresse d'une zone mémoire dont la taille est égale au produit de la taille d'une case \* par le nombre de cases. La notation d'accès indexée permet de faire varier le pointeur d'une case à une autre :

```
&tableau[0] ~ tableau
```

L'adresse de la 1<sup>ère</sup> case est l'adresse du tableau...

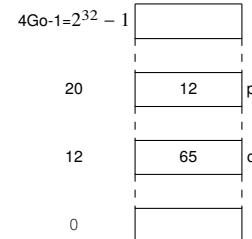
Attention

Il ne faut pas donner, comme valeur de retour d'une fonction, un **pointeur sur une variable locale** à cette fonction !  
*Lors du retour de la fonction, le contenu de la pile est réutilisé, et la valeur renournée est écrasée.*

- \* un pointeur est une variable qui contient une adresse mémoire ;
- \* dans une architecture 32bit, la taille d'un pointeur est de 4 octets, quelque soit le type du pointeur ;

```
1 char c = 'a'; /* valeur 65 en ASCII */
2 char *p = &c;
```

*La variable p étant de type pointeur, elle occupe les cases mémoires 20, 21, 22 et 23, soient 4 cases à partir de la première.*



### Les différentes utilisations d'un pointeur

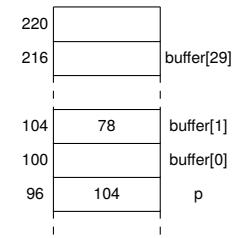
1. affecter une valeur numérique à un pointeur :

```
1 char *p = (char *) 0x2050;
2 *p = 5; /* stocker 5 à l'adresse 0x2050 */
```

2. Obtenir l'adresse mémoire d'une autre variable :

```
1 int *p;
2 int buff[30];
3 p = &buff[1];
4 *p = 78;
```

*Ici, une adresse et un entier tiennent sur 32 bits ou 4 octets.*



3. allouer de la mémoire sur le tas :

```
1 int *p;
2 p = (int *) calloc(1, sizeof(int));
```



### Appel de fonction, variables et segments

Lors d'un appel d'une fonction :

- on mémorise où on en est dans le code que l'on exécute : sauvegarde de la valeur du **compteur ordinal** ;
- on charge dans le compteur ordinal, l'**adresse de la première instruction** du code de la fonction ;
- on **restaure l'ancienne valeur du compteur ordinal** lorsque l'on retourne de la fonction ;
- la sauvegarde du compteur ordinal et sa restauration se fait à l'aide de la «**pile d'exécution**» ;

On étend cette utilisation de la pile aux variables manipulées par la fonction :

- passage des arguments de la fonction : ils sont mis sur la pile ;
- récupération du résultat de la fonction lors du retour : il est laissé sur la pile ;
- définition des variables locales utilisées dans la fonction : dans la pile ;
- allocation d'espace mémoire disponible pendant tout le programme : utilisation du tas.

#### Contexte processeur

Un contexte est l'ensemble des informations nécessaires à l'exécution d'un programme par le processeur :

- la valeur du compteur ordinal ;
- les valeurs des registres généraux ;
- les valeurs des registres de segments : qui pointent sur la base et le sommet de la pile, sur le tas, sur les segments de code et de données ;
- la valeur du ou des registres d'état.

Ses informations peuvent être sauvegardées et restaurées : elles permettent de passer de l'exécution d'un programme à un autre.

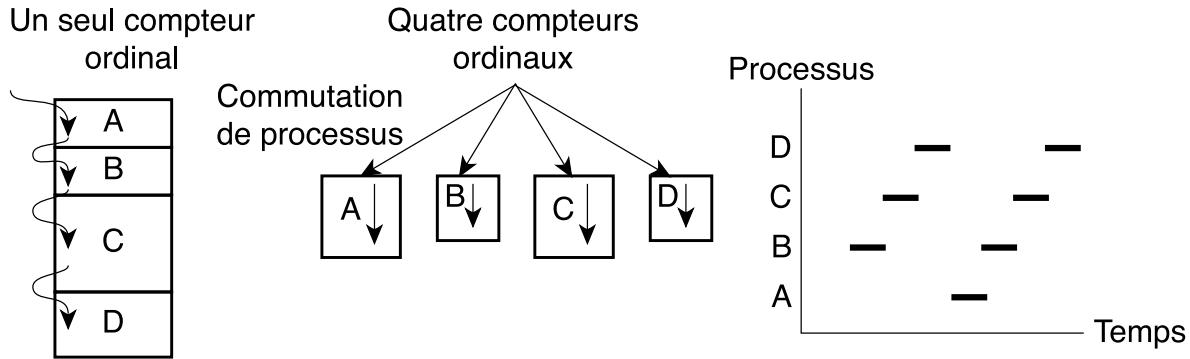
Un **programme qui s'exécute** est l'association de :

- ▷ son **contexte** d'exécution ;
- ▷ le **contenu utilisé** de chacun de ses segments.



## Comment exécuter plusieurs programmes en «multi-tâche»

- charger différents programmes en mémoires :
  - ◊ répartir la mémoire entre les différents programmes ;
  - ◊ traduire les adresses etc.
- passer, «switcher», de l'exécution d'un programme à l'autre :
  - ◊ associer un contexte du processeur à chaque programme → notion de **processus** ;
  - ◊ passer d'un processus à un autre : sauvegarde un contexte et restaurer un autre contexte ;
  - ◊ passer d'un «CO» associé à un processus à un «CO» associé à un autre processus :



## Comment organiser la mémoire entre les différents processus ?

En utilisation le concept de «pagination».

## Comment passer régulièrement et automatiquement d'un processus à l'autre ?

En utilisant le concept «d'horloge» et d'«interruption».



### Comment partager la mémoire entre différents processus ?

Exemple : les programmes suivants se partagent la mémoire de l'ordinateur :

OS	0x9000	1. l'OS, «operating system», de l'adresse 0x8000 à 0x9000 ;
gcc	0x8000	2. le compilateur <code>gcc</code> de l'adresse 0x7000 à 0x8000 ;
firefox	0x7000	3. le navigateur <code>firefox</code> de l'adresse 0x6000 à 0x7000 ;
emacs	0x6000	4. l'éditeur de texte <code>emacs</code> de l'adresse 0x5000 à 0x6000 ;
	0x5000	

### Problématiques

- *emacs voudrait plus de mémoire que celle qui lui est allouée ;*
- *firefox voudrait plus de mémoire que la mémoire totale de la machine ;*
- *gcc, à la suite d'une erreur, écrit dans la mémoire 0x6500 ;*
- *emacs n'utilise pas toute la mémoire qui lui a été allouée.*

**Autre question :** comment adapter le code d'`emacs` à son emplacement à l'adresse 0x5000 ?

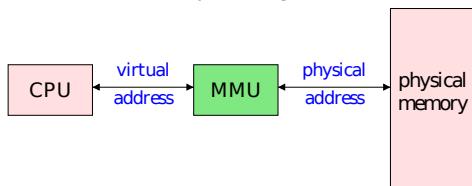
### Contraintes

- \* **Protection :**
  - un «bug» dans un processus ne doit pas corrompre la mémoire dans un autre processus ;
  - empêcher qu'un processus A ne vienne corrompre la mémoire d'un processus B ;
  - empêcher A d'observer la mémoire du processus B.
- \* **Transparence :**
  - un processus ne doit pas nécessiter un emplacement mémoire particulier ;
  - un processus nécessite de larges quantités de mémoires contigües : pour ses structures de données, pour son fonctionnement (pile), etc.
- \* **Gestion automatique des ressources :**
  - un programmeur développe un programme avec l'idée que l'ordinateur dispose de suffisamment de mémoire ;
  - la somme de la taille de tous les processus est souvent supérieure à celle de la mémoire physique.



## La mémoire virtuelle

- donner à chaque processus son propre espace mémoire «virtuel» :
  - \* un processus utilise **uniquement** de la mémoire référencée par des adresses **virtuelles ou logiques** ;
  - \* la mémoire utilise des adresses **physiques ou réelles** ;
- ◊ traduire chaque opération d'accès mémoire en chargement et en stockage, «load» & «store» ;
- ◊ masquer au processus quelle mémoire physique il utilise ;
- ◊ utiliser un **composant** dédié à la gestion de la mémoire pour accélérer le traitement : «MMU», «Memory Management Unit» :

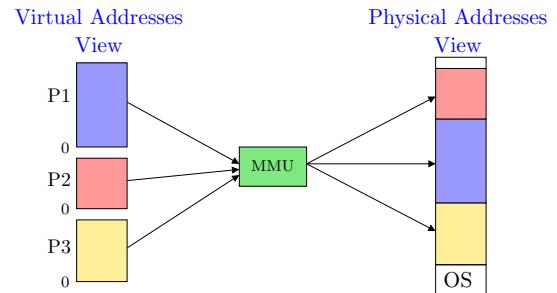


- > paramétrable uniquement avec des instructions privilégiées du «mode noyau» ;
- > traduit des adresses virtuelles vers des adresses physiques ;
- > la MMU fait partie du processeur.

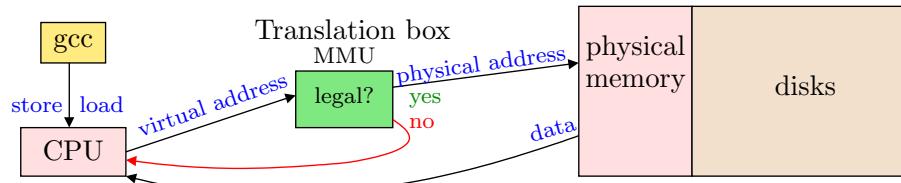
- crée une vue «par processus» de la mémoire :

Chaque processus :

- ◊ a l'impression d'être seul en mémoire :
  - \* ne voit pas la mémoire des autres processus ;
  - \* ne voit pas la mémoire de l'OS.
- ◊ a sa mémoire qui commence à l'adresse zéro ;

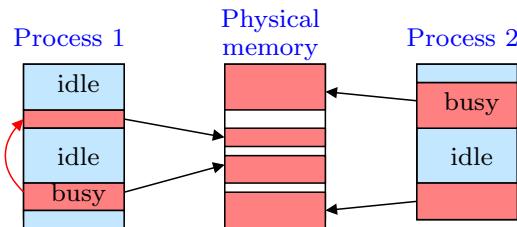


- protéger la mémoire d'un processus
  - ◊ empêcher qu'un processus chamboule la mémoire d'un autre processus



*La tentative d'accès à la MMU pour la reconfigurer/l'utiliser en lieu et place de l'OS provoque l'arrêt du processus : bascule du mode utilisateur vers le mode noyau ⇒ l'OS rejette le processus.*

- permettre à un processus de voir plus de mémoire que celle existante :
  - ◊ reloger certaines zones mémoires sur disque :



- \* les parties «idle», inactives sont placées sur le disque jusqu'à ce qu'elles soient utilisées ;
- \* les processus peuvent utiliser la mémoire libérée ;
- \* **Quand un processus n'utilise pas une mémoire, l'allouer à un autre processus.**
- \* il faut identifier quelle mémoire est utilisée ou va être utilisée et la «manipuler» :
  - ▷ par octets ? gestion trop lourde
  - ▷ par zone mémoire ? oui, on introduit le concept de «page mémoire».

La taille d'une page sous Linux :

```
pef@darkstar:~$ getconf PAGESIZE  
4096
```

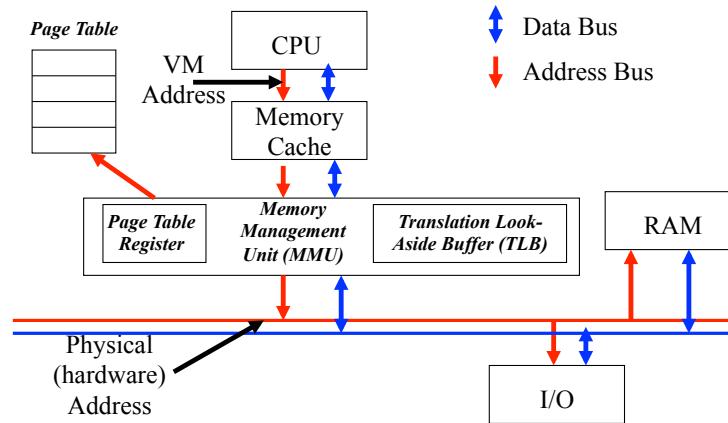
## Problèmes restant à traiter

- comment identifier les différents besoins d'allocation du processus ?
  - ◊ mémoire qui restera allouée toute la durée d'exécution du processus ?
  - ◊ mémoire ponctuellement allouée ?
- ⇒ Utiliser la **segmentation**
- comment distinguer les différentes parties d'un processus ?
  - ◊ identifier la partie code, la partie données ;
  - ◊ identifier la création de données ;
- ⇒ Utiliser la **segmentation**
- comment permettre aux processus d'augmenter la mémoire allouée ?
  - ⇒ Permettre à la MMU de décharger/recharger une page depuis le disque dur en détectant un «défaut de page», c-à-d l'accès à une adresse virtuelle dont la page contenant l'adresse physique associée n'est pas en mémoire.
- comment gérer plusieurs processus ?
  - ⇒ Paramétrer la MMU avec une **table de traduction d'adresse** associée à chaque processus.



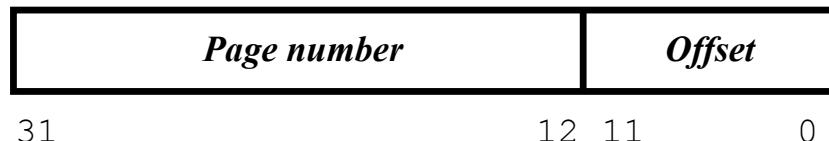
## Comment intégrer le concept de «page» dans le «processus» ?

- le concept de page introduit une «indirection» supplémentaire pour l'accès à la mémoire :
  - le processus utilise une adresse virtuelle pour désigner un emplacement mémoire ;
  - pour accéder à cet emplacement mémoire, il faut connaître son adresse physique ;
  - pour connaître cette adresse physique, il faut la rechercher dans une table de correspondance ;
  - pour connaître l'emplacement dans cette table, on se sert de l'adresse virtuelle.
- cette indirection (passer par une table intermédiaire) est gérée par la MMU : pas de ralentissement ;

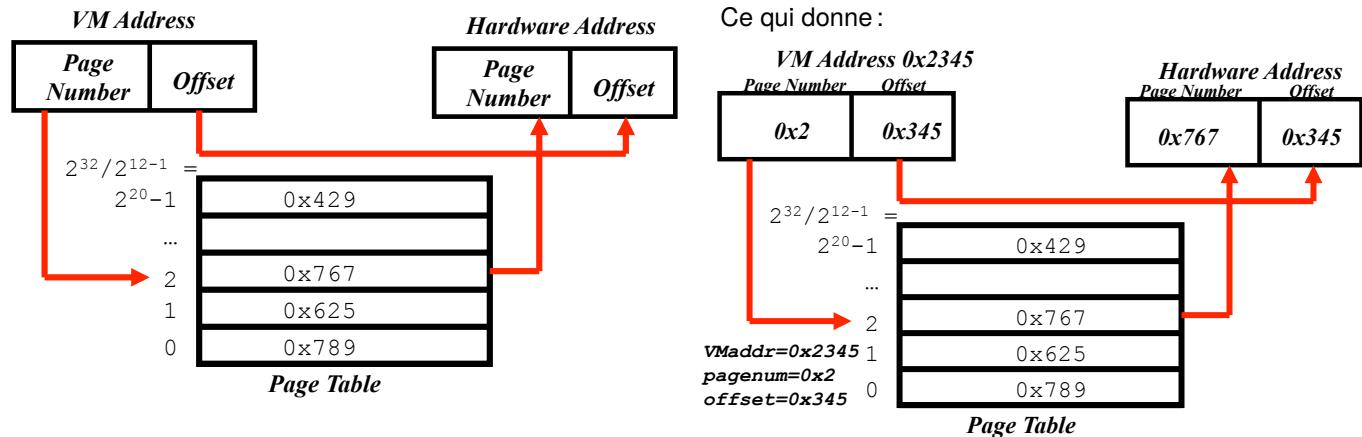


- la MMU possède un registre référençant la table courante utilisée pour réaliser la traduction virtuelle → physique ;
- chaque processus possède sa propre table (il utilise des adresses physiques différentes des autres processus) ;
- le registre, «Page Table Register», est modifié à chaque changement de contexte lié au changement de processus ;
- la table de page contient les informations des zones mémoires valides pour un processus ;
- pour accélérer la consultation de la table, on stocke les traductions les plus récentes dans le TLB.

- une adresse virtuelle est décomposée en deux parties :
  - le numéro de page sur 20 bits ;
  - le décalage ou «offset» sur 12 bits, soit une taille de page de 4096 octets ;



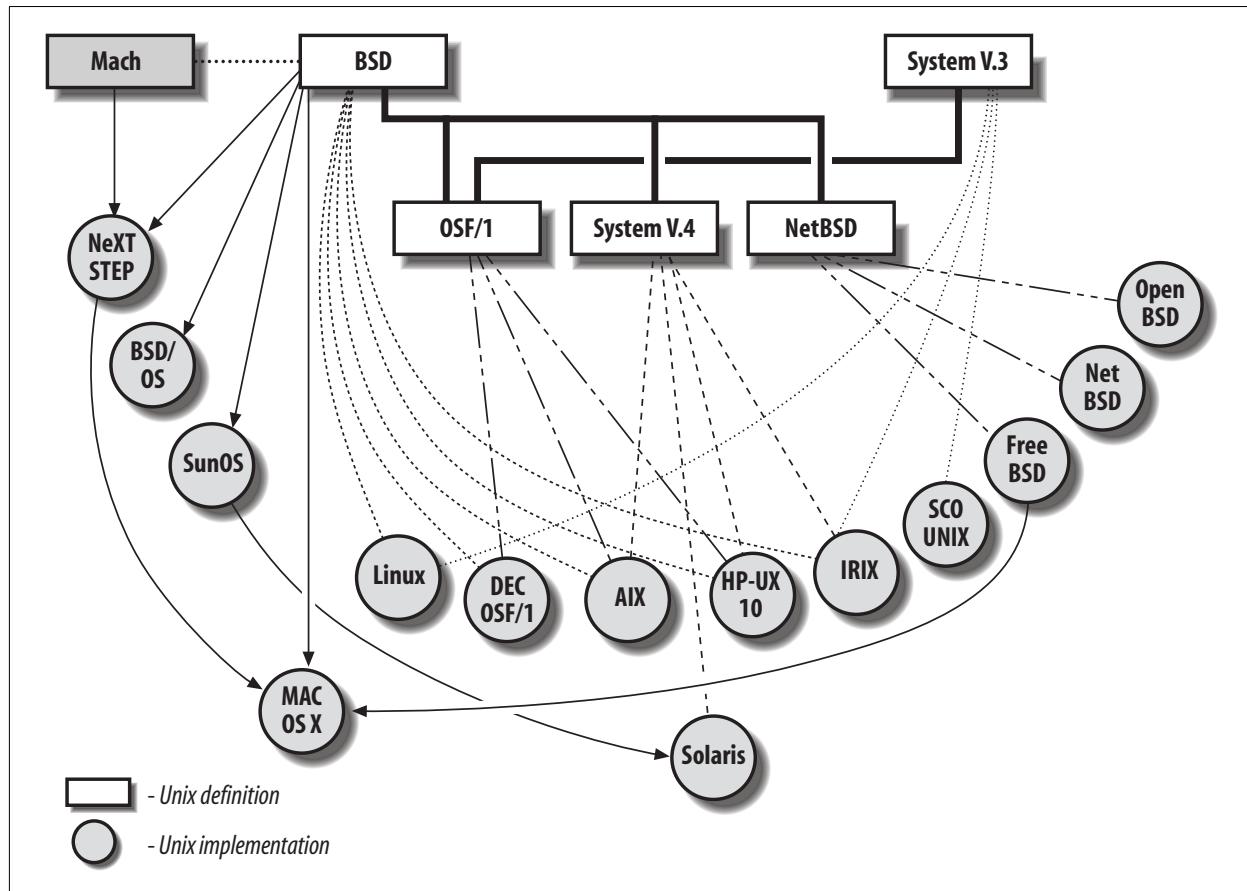
- seul le numéro de page est traduit, le décalage reste le même ;
- exemple : on veut traduire l'adresse virtuelle 0x2345 :
  - 0x2345 donne  $\boxed{2}$   $\boxed{345}$ , les trois derniers chiffres hexadécimaux correspondant au décalage sur 12 bits,
  - le numéro de page est 0x2.



### Plan de la partie

- les contraintes historiques qui amènent à la création d'Unix :
  - ◊ les systèmes d'exploitation multi-utilisateurs ;
  - ◊ le concept de «*time sharing*» ;
  - ◊ la création du langage C: «*un assembleur portable*» ;
- les fondamentaux d'Unix ;
- le partage de la machine physique ;
- les interruptions ;
- mode noyau/mode utilisateur ;
- les fonctions d'un système d'exploitation ;





Unix est un système supportant :

- Plusieurs utilisateurs ;
- Plusieurs programmes :
  - ◊ Partage du temps d'utilisation ;
  - ◊ Protection de la mémoire ;
  - ◊ Partage des ressources ;
  - ◊ Accès à distance.
- Problèmes :
  - ◊ Équité ;
  - ◊ Gestion du matériel ;
  - ◊ Droits d'accès...

Unix est un système nécessitant un processeur récent disposant d'au moins **deux modes de fonctionnement** pour lui permettre de contrôler les accès aux ressources de la machine :

- un mode **noyau** (ou système ou superviseur)  
*le processeur peut exécuter toutes les instructions disponibles du système.*
- un mode **utilisateur**.  
*certaines instructions lui sont interdites, pour des raisons de sécurité du système.*

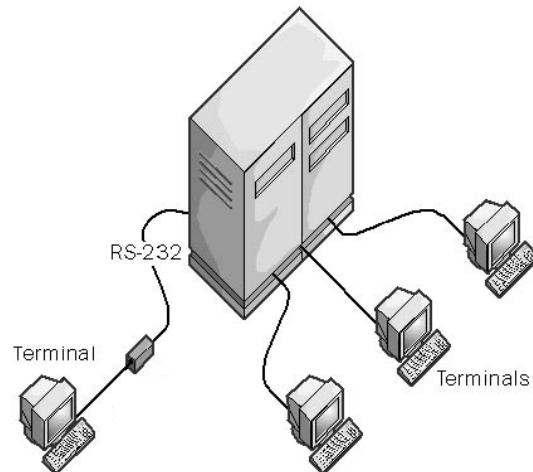
#### Protection fondamentale

Les programmes sont exécutés en mode utilisateur :

- ◊ obligation de faire appel au système d'exploitation pour les opérations à risque qui nécessite de passer en mode noyau.  
*On appelle ces opérations des **appels système** (exemple : opérations de gestion de fichier pour modification des données)*
- ◊ **Tous les accès aux ressources et aux données sont contrôlés par le système d'exploitation.**

## L'origine : le «mainframe»

### Un ordinateur central avec des accès partagés



L'ordinateur est utilisé au travers de terminaux :

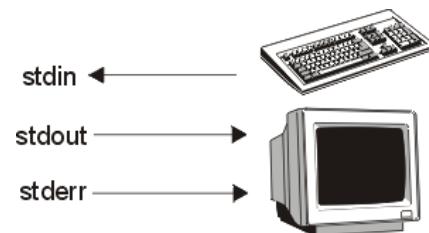
- un terminal correspond à un écran et un clavier :



- une liaison série, «RS-232», permet d'échanger du texte dans les deux sens (terminal  $\Leftrightarrow$  mainframe).

Par convention, on distingue 3 canaux d'échange essentiels :

- «stdin» ou «0» : le canal d'entrée standard ;
- «stdout» ou «1» : le canal de sortie standard ;
- «stderr» ou «2» : le canal standard d'erreur.



La gestion du partage de la machine physique et des ressources matérielles doit permettre de répondre aux questions suivantes :

- Partage du **processeur unique** : parmi tous les programmes chargés en mémoire centrale, lequel doit s'exécuter ?
- Partage de la **mémoire centrale** : comment allouer la mémoire centrale aux différents programmes.
  - ◊ Comment assurer la protection entre plusieurs programmes utilisateurs ?
  - ◊ Comment protéger le système d'exploitation des programmes utilisateurs ?
  - ◊ *Par protection, on entend ici veiller à ce qu'un programme donné n'accède pas à une plage mémoire allouée à un autre programme.*
- Partage des **périphériques** ;

## Périphérique et concurrence

La **programmation concurrente** est liée à la naissance des SE et à l'invention des contrôleurs de périphériques (device controllers) :

- ▷ fonctionnent indépendamment du processeur central ;
- ▷ permettent d'effectuer des opérations d'E/S en concurrence d'un programme exécuté par le processeur central ;
- ▷ Le contrôleur communique avec le processeur central par l'intermédiaire d'une **interruption**, un signal matériel, qui le déroute de l'exécution de la séquence d'instructions courante pour exécuter une séquence d'instructions différente.

## Problème ?

L'intégration de contrôleur de périphérique pose le problème que certaines parties d'un programme peuvent s'exécuter dans un ordre imprévisible ! *Si un programme est en train de modifier la valeur d'une variable, une interruption peut arriver et peut conduire à ce qu'une autre partie du programme essaie de changer la valeur de cette même variable !*



Le système d'exploitation se présente comme une couche logicielle placée entre la machine matérielle et les applications.

système d'exploitation	Éditeur de texte Bases de données	Tableur Navigateur	Programmes Utilisateurs
	Compilateur	Éditeur de liens	Chargeur Assembleur
	Appels systèmes		
	Gestion de la concurrence	Gestion de la protection	Gestion des objets externes (fichiers)
Gestion du processeur			
Gestion de la mémoire			
Gestion des entrées-sorties			
Mécanisme des interruptions			
MACHINE PHYSIQUE			

Il s'interface avec :

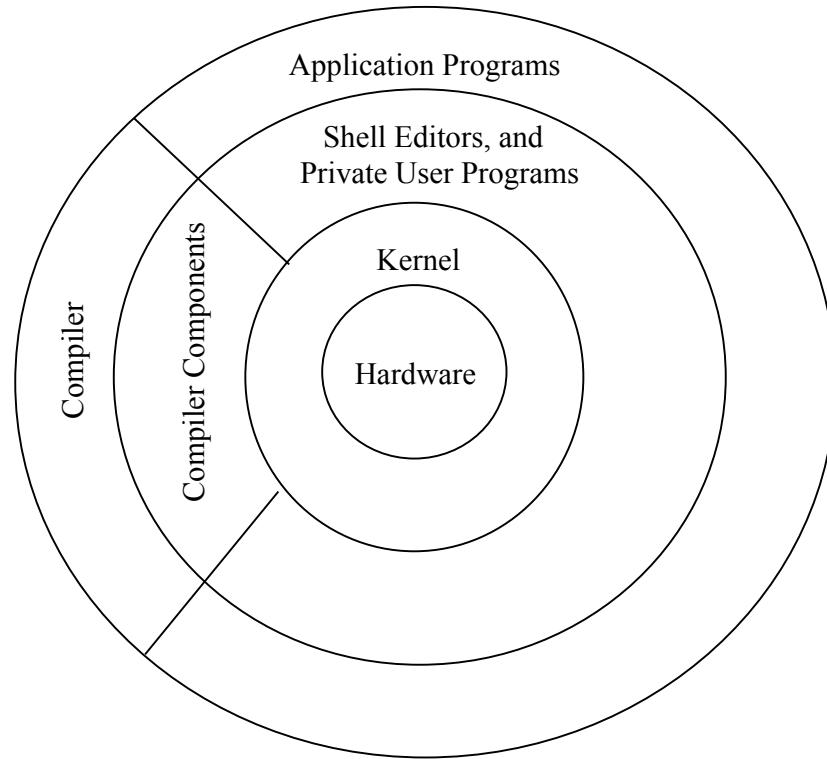
- la couche matérielle, notamment par le biais du mécanisme des interruptions.
- les applications par le biais des primitives qu'il offre : appels système et commandes.



- **Gestion du processeur** : allouer le processeur aux différents programmes pouvant s'exécuter.  
*Cette allocation se fait suivant un algorithme d'ordonnancement qui planifie de l'exécution des programmes.*
- **Gestion de la mémoire** : allouer la mémoire centrale entre les différents programmes pouvant s'exécuter (pagina-tion/segmentation).  
*Mécanisme de **mémoire virtuelle** : traduction entre adresse logique et physique, chargement uniquement des parties de code et de données utiles à l'exécution d'un programme.*
- **Gestion des E/S** :
  - ◊ accès aux périphériques,
  - ◊ faire la liaison entre appels de haut niveau des programmes (exemple : `getchar()`) et les opérations de bas niveau du système d'exploitation responsable du périphérique (exemple : le clavier).*C'est le pilote d'E/S (driver) qui assure cette liaison.*
- **Gestion de la concurrence** : plusieurs programmes coexistent en mémoire centrale :
  - ◊ assurer les besoins de communication pour échanger des données ;
  - ◊ synchroniser l'accès aux données partagées afin de maintenir la cohérence de ces données.*Le système offre des outils de communication et de synchronisation entre les programmes.*
- **Gestion du système de fichier** : stockage des données sur mémoire de masse (disque dur, mémoire flash, SSD, etc.)  
*Le système d'exploitation gère un **format d'organisation** (`ext4` sous Linux), un **système de protection** (journalisation des modifications) et un **format de fichier** (`inodes`).*
- **Gestion de la protection** : mécanisme de garantie que les ressources (CPU, mémoire, fichiers) ne peuvent être utilisées que par les programmes disposant des droits nécessaires.
- Gestion des **comptes** et des **droits** ,
- **Protection du système d'exploitation** : séparation du système d'exploitation en mode noyau et des programmes en mode utilisateur.



- au centre se trouve le «hardware»
- sur le bord se trouve les applications utilisateurs ;
- entre les deux : le système d'exploitation ;
- *le compilateur sert de «médiateur».*



## Linux = Noyau monolithique

Quatre grandes fonctions nécessaires :

- Gestion des processus**

*Administre l'exécution de programmes.*

- Gestion de la mémoire**

- Gestion des fichiers : VFS**

*Interface avec gestionnaire de fichiers spécifiques  
(Unix, DOS, floppy, CD-ROM...)*

- Gestion du réseau**

*Interface avec gestionnaires de protocoles*

## Linux = Noyau modulaire

Ajout de fonctionnalités supplémentaires :

- Modules statiques**

*Ajout uniquement à la compilation du noyau :  
le système chargera de manière inconditionnelle les modules de ce type.*

- Modules dynamiques**

*Ajout «au cours de la vie» du système, chargement arbitraire, en temps voulu.*



## Le mode noyau

- \* lorsque le processeur est dans ce mode :
  - ◊ il peut exécuter n'importe quelle instruction possible ;
  - ◊ il peut modifier le contenu de n'importe quelle case mémoire ;
  - ◊ il peut accéder et modifier n'importe quel registre du processeur ou d'un périphérique ;
  - ◊ il a le **contrôle total** de l'ordinateur ;
- \* le système d'exploitation s'exécute en mode noyau.

## Le mode utilisateur

- \* lorsque le processeur est dans ce mode :
  - ◊ il n'exécute qu'un sous-ensemble limité d'instructions ;
  - ◊ il ne peut modifier que les zones mémoires affectés à l'exécution du programme ;
  - ◊ il ne peut accéder qu'à un sous-ensemble limité des registres du processeur et il ne peut accéder aux registres des périphériques ;
  - ◊ il a un accès limité aux ressources de l'ordinateur ;
- \* les programmes utilisateurs s'exécute en mode utilisateur

## Mode utilisateur et mode noyau

- \* lorsque l'ordinateur démarre, il charge le système d'exploitation en mode noyau ;
- \* en mode noyau, le système d'exploitation configure les vecteurs d'interruption et initialise les périphériques ;
- \* ensuite, il démarre le premier **processus** et bascule en mode utilisateur ;
- \* il exécute en mode utilisateur les processus en tâche de fond, «*background*», appelés «démons» ;
- \* à la connexion d'un utilisateur, il démarre un **shell** ou un gestionnaire de fenêtre.



## Plan

### **La notion de processus**

- notion de programme et de processus ;
- les états d'un processus ;
- les ressources ;

### **Les opérations sur un processus**

- Création et gestion par l'OS ;
- Cycle de vie ;
- Les processus et l'organisation de l'OS ;
- La terminaison d'un processus ;

### **Le processus & les interruptions**

- appel système ;
- préemption ;

### **Notion de «threads»**

### **Ordonnancement**

## 7 Notion de programme et de processus

73

Un **programme** est une entité purement statique associée à la suite des instructions qui la composent (le ou les fichiers stockés sur le disque dur).

Un **processus** est un programme en cours d'exécution auquel est associé :

- un environnement processeur (CO, PSW, RSP, registres généraux) ;
- un environnement mémoire appelés contexte du processus.

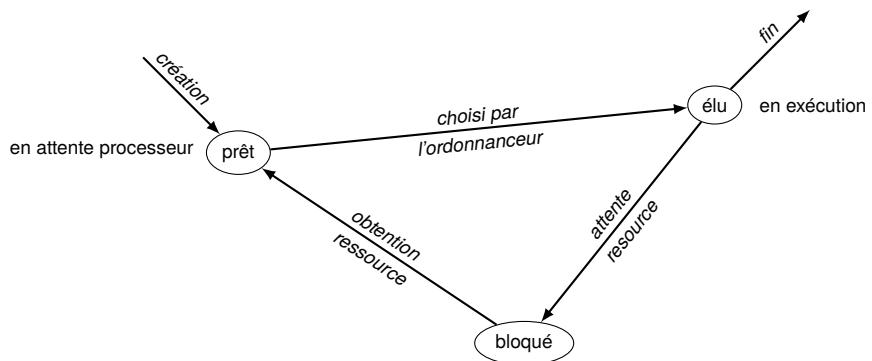
Un processus est :

- ▷ l'«instance dynamique» d'un programme ;
- ▷ le **fil d'exécution**, thread, de celui-ci dans un espace d'adressage protégé (ensemble des instructions et des données accessibles en mémoire).

### État d'un processus

C'est le système d'exploitation qui détermine et modifie l'état d'un processus sous l'effet des événements :

- le choix de l'**ordre d'exécution** des processus est appelé ordonnancement ;
- l'algorithme chargé de faire ce choix est appelé **ordonnanceur**.



## État élu

Lors de son exécution, un processus est caractérisé par un état :

- o lorsque le processus obtient le processeur et s'exécute, il est dans l'état **élu**.
- o l'état **élu** est l'état d'exécution du processus.

## État bloqué

Lors de cette exécution, le processus peut demander à **accéder à une ressource** (réalisation d'une entrée/sortie, accès à une variable protégée) qui n'est **pas immédiatement disponible** :

- o le processus **ne peut pas poursuivre** son exécution tant qu'il n'a pas obtenu la ressource (par exemple, le processus doit attendre la fin de l'entrée-sortie qui lui délivre les données sur lesquelles il réalise les calculs suivants dans son code) ;
- o le processus *quitte* alors le processeur et passe dans l'état **bloqué** : l'état bloqué est donc l'état d'attente d'une ressource autre que le processeur.

## État prêt

Lorsque le processus a enfin obtenu la ressource qu'il attendait, celui-ci peut potentiellement reprendre son exécution. Cependant, dans le cadre de **systèmes multiprogrammés**, il y a plusieurs programmes en mémoire centrale, et donc plusieurs processus.

- o lorsque le processus est passé dans l'état bloqué, le processeur a été **alloué** à un autre processus.
- o le processeur n'est donc pas forcément libre : le processus passe alors dans l'état **prêt**.

L'état «Prêt» est l'état **d'attente** du processeur.



## Changement d'état

- ▷ Le passage de l'état prêt vers l'état élu constitue l'opération **d'élection** .
- ▷ Le passage de l'état élu vers l'état bloqué est l'opération de **blocage** .
- ▷ Le passage de l'état bloqué vers l'état prêt est l'opération de **déblocage** .

### Remarques

- Un processus est toujours créée dans l'état **prêt** .
- Un processus se termine toujours à partir de **l'état élu** (sauf anomalie).



Les **appels systèmes** constituent l'interface du système d'exploitation et sont les **points d'entrées** permettant l'exécution d'une fonction du système :

- les **appels système** sont directement appelleables depuis un programme.
- les **commandes** permettent d'appeler les fonctions du système depuis le prompt de l'interpréteur de commande (shell, «*Command Line Interface*»)

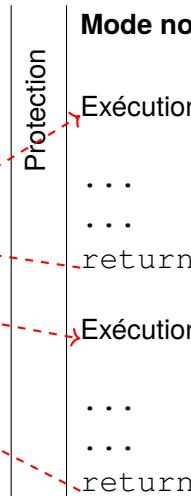
## Déroulement d'un appel système

### Mode utilisateur

```
main()
{
    int i, j, fd;
    i = 3;
    fd = open("mon_fichier", "r");
    read(fd, j, 1);
    j = j/i;
}
```

### Mode noyau

```
Exécution de open ()
...
return
Exécution de read ()
...
return
```



Lors de l'exécution d'un appel système, le programme utilisateur passe du mode d'exécution utilisateur au mode d'exécution **superviseur** ou **privilégié**, appelé «**mode noyau**».



**Mode utilisateur**

```
main()
{
    int i, j, fd;
    i = 3;
    fd = open("mon_fichier", "r");
```

**Mode noyau**

Exécution de `open()`

...  
...

1 sauvegarde CO, PSW utilisateur  
 2 chargement CO ← adresse de la fonction `open`  
 3 chargement PSW ← mode superviseur

- **appel système** : demande d'exécution d'une fonction du système d'exploitation ;
- passage du **mode utilisateur** au **mode noyau** :
  - ◊ déclenchement d'une **interruption logicielle** visible dans le code assembleur du programme ;
  - ◊ passage dans un mode d'**exécution privilégié** qui est le mode d'exécution du système d'exploitation (mode noyau ou superviseur).

*Accès à un plus grand nombre d'instructions machine que le mode utilisateur (permet l'exécution des instructions de masquage et démasquage des interruptions interdites en mode utilisateur).*

  - ◊ **sauvegarde du contexte** utilisateur ;

⇒ **commutation de contexte**

**Exemple Assembleur dans le système Linux**

Assembleur x86, sous Linux, écrit pour le compilateur NASM

```
section .data
    helloMsg: db 'Hello world!',10
    helloSize: equ $-helloMsg
section .text
    global start
start:
    mov eax,4          ; Appel système "write" (sys write)
    mov ebx,1          ; File descriptor, 1 pour STDOUT (sortie standard)
    mov ecx,helloMsg  ; Adresse de la chaîne à afficher
    mov edx,helloSize ; Taille de la chaîne
    int 80h            ; Exécution de l'appel système
    ; Sortie du programme
    mov eax,1          ; Appel système "exit"
    mov ebx,0          ; Code de retour
    int 80h
```

Appel au noyau de LINUX (int 80h)



**Mode utilisateur**

```
main()
{
    int i, j, fd;
    i = 3;
    fd = open("mon_fichier", "r");
```

**Mode noyau**

```
Exécution de open ()
...
...
return
```

Protection

```
1 restauration du contexte utilisateur
2 chargement CO ← CO sauvegardé
3 chargement PSW ← PSW sauvegardé
```

À la fin de l'**appel système**:

- ▷ passage du **mode noyau** au **mode utilisateur**;
- ⇒ **commutation de contexte**: restauration du contexte utilisateur.



## Trois causes de passage du mode utilisateur au mode noyau

- appel d'une fonction du système :**  
demande explicite de passage en mode noyau
- exécution d'une opération illicite**  
(division par 0, instruction machine interdite, violation mémoire...): trappe.  
L'exécution du programme utilisateur est alors arrêtée.
- interruption par le matériel et le système d'exploitation :**  
le programme utilisateur est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en mode superviseur.
- interruption par l'horloge :**  
c'est une interruption matérielle qui permet au système d'exploitation de passer de l'exécution d'un processus à un autre dans le cadre de la «multiprogrammation».   
Cette interruption ne peut être ignorée par le processus.  
On appelle ce mécanisme : la **préemption**.

