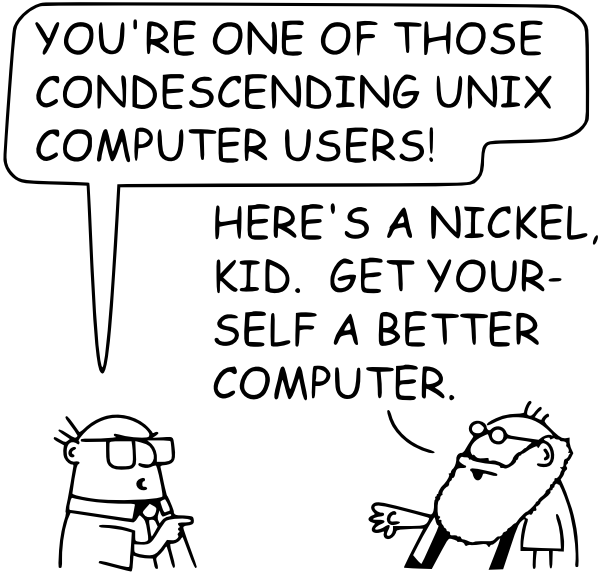
Faculté des Sciences & Techniques

Université de Limoges

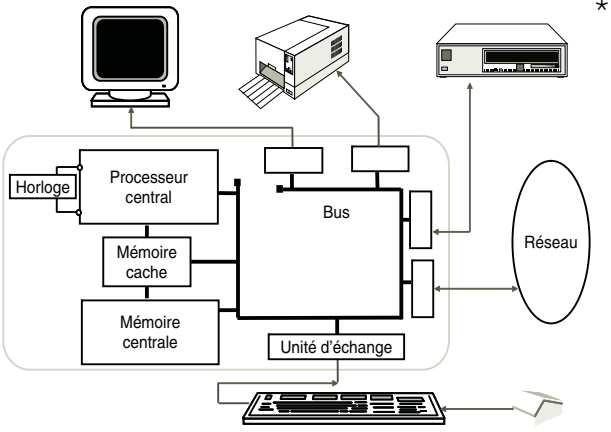
Licence 3ème année




# Système d'exploitation — P-F. Bonnefoi

*Version du 7 septembre 2022*

Table des matières



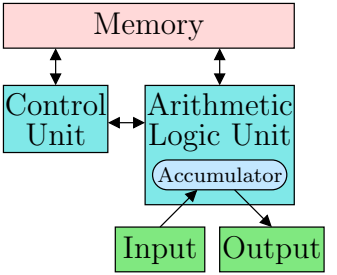

- une *mémoire centrale* :
  - architecture «Von Neuman» : programme et données résident dans la même mémoire.
    *La lecture ou l'écriture dans la mémoire concerne une donnée ou une instruction.*
  - architecture «Harvard» : programme et données sont dans des mémoires différentes ;
    *Il est possible de charger simultanément une instruction et une donnée.*

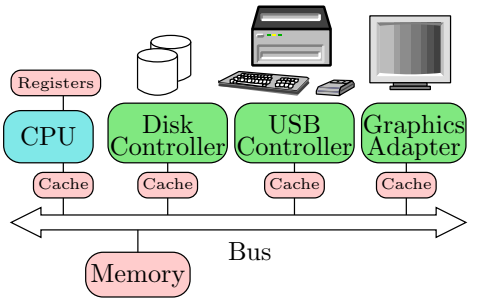
  L'architecture d'un ordinateur est de type «von Neumann».
  Des matériels embarqués, comme des modules Arduino ou des DSP, «*Digital Signal Processor*», utilisent une architecture de type «Harvard».

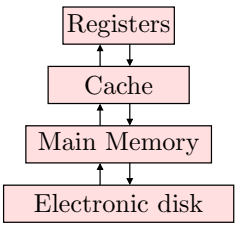
- un *processeur central* ou CPU, «*Central Processing Unit*» qui réalise le traitement des informations logées en mémoire centrale :
  - le processeur permet l'exécution d'un programme ;
  - chaque processeur dispose d'un langage de programmation composé d'**instructions machine** spécifiques ;
  - **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.

# Architecture d'un ordinateur : plus de détails

- le processeur est la combinaison :
  - d'une « unité arithmétique et logique », «ALU», «*Arithmetic Logic Unit*» ;
  - d'une « unité de contrôle » : contrôle du bus servant à échanger données et instructions ;
- la mémoire n'est pas d'**accès uniforme**, c-à-d que la vitesse d'accès n'est pas toujours la même :
  - les registres : cases mémoires intégrées au processeur ;
  - le cache : zone mémoire **tampon** entre la mémoire centrale et le processeur :
    - ⋆ permet d'éviter l'accès à la mémoire centrale pour des données déjà accédées et mémorisées dans le cache ;
    - ⋆ évite d'utiliser le bus de données ;
  - le disque : permet de « *simuler* » de la mémoire.

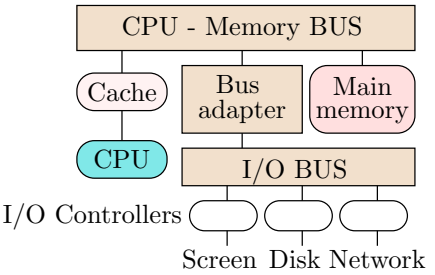


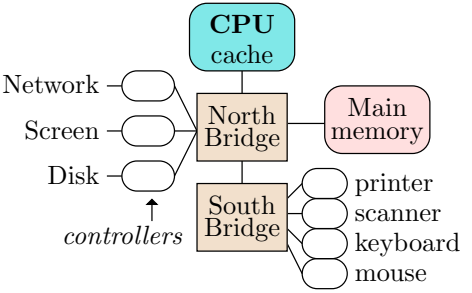



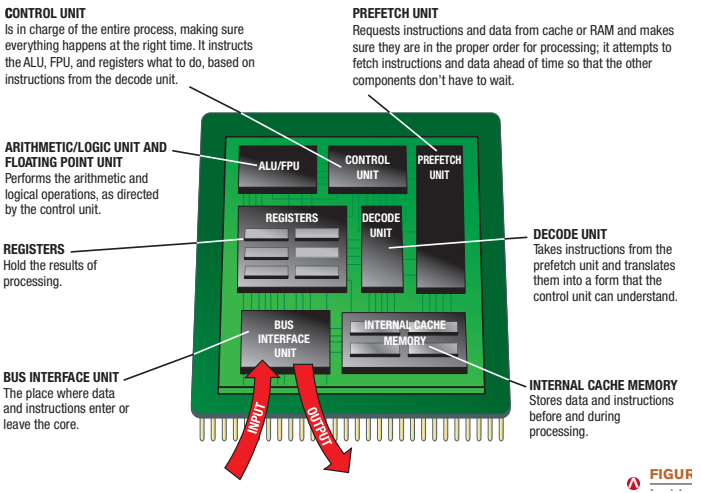

- la vitesse entre ces différentes mémoires est **très différentes** :
- la taille de ces memoires est très différente :
  - ⋆ registres < $1ko$
  - ⋆ cache : qqs $Mo$ ;
  - ⋆ mémoire centrale : qqs $Go$ ;
  - ⋆ le stockage sur disque : qqs $To$.

"Archaic" design




Current design




- «FPU», «*Floating Point Unit*» : gère les calculs sur nombre à virgule flottante ;
- «Prefetch Unit» : charger les données et instructions en amont de leur traitement pour accélerer le travail du processeur ;
- «Decode Unit» : analyse les instructions pour le processeur.

# Exemple le processeur 8bits 6502

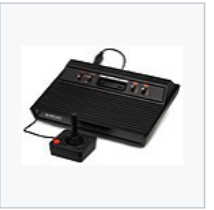


- processeur développé par Chuck Peddle pour la société MOS Technology ;
- introduit en 1975 ;
- très populaire :

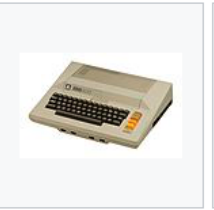
Apple IIe

Commodore PET

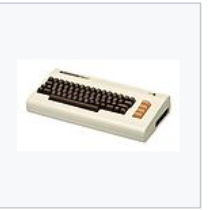
BBC Micro



Atari 2600



Atari 800



Commodore VIC-20

Commodore 64

Family Computer

Ohio Scientific Challenger 4P

Tamagotchi digital pet[53]

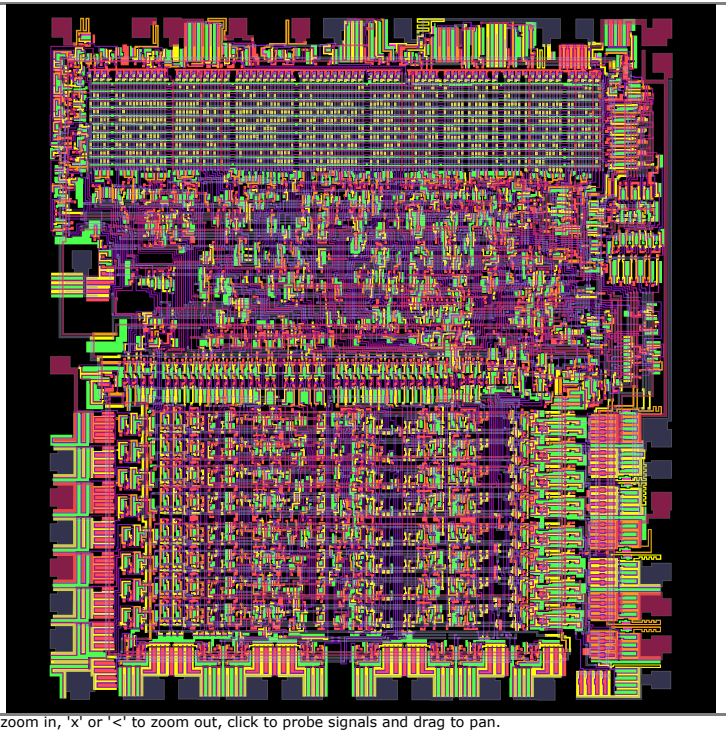
Atari Lynx

- 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)

Find: Clear Highlighting Animate during simulation:

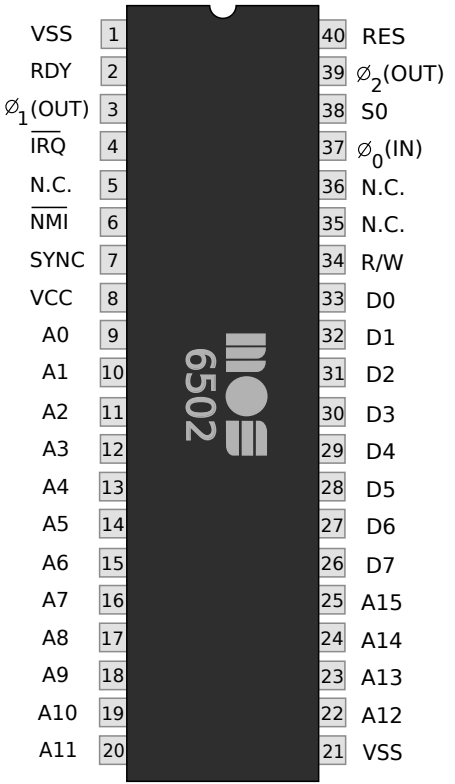
Hide Chip Layout Link to this location



```
halfcyc:372 phi0:0 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 0
0010: e8 88 e6 0f 38 69 02 60 00 00 00 0
0020: 00 00 00 00 00 00 00 00 00 00 00 0
0030: 00 00 00 00 00 00 00 00 00 00 00 0
0040: 00 00 00 00 00 00 00 00 00 00 00 0
0050: 00 00 00 00 00 00 00 00 00 00 00 0
0060: 00 00 00 00 00 00 00 00 00 00 00 0
0070: 00 00 00 00 00 00 00 00 00 00 00 0
0080: 00 00 00 00 00 00 00 00 00 00 00 0
0090: 00 00 00 00 00 00 00 00 00 00 00 0
00a0: 00 00 00 00 00 00 00 00 00 00 00 0
00b0: 00 00 00 00 00 00 00 00 00 00 00 0
00c0: 00 00 00 00 00 00 00 00 00 00 00 0
00d0: 00 00 00 00 00 00 00 00 00 00 00 0
00e0: 00 00 00 00 00 00 00 00 00 00 00 0
00f0: 00 00 00 00 00 00 00 00 00 00 00 0
0100: 00 00 00 00 00 00 00 00 00 00 00 0
0110: 00 00 00 00 00 00 00 00 00 00 00 0
0120: 00 00 00 00 00 00 00 00 00 00 00 0
0130: 00 00 00 00 00 00 00 00 00 00 00 0
0140: 00 00 00 00 00 00 00 00 00 00 00 0
0150: 00 00 00 00 00 00 00 00 00 00 00 0
0160: 00 00 00 00 00 00 00 00 00 00 00 0
0170: 00 00 00 00 00 00 00 00 00 00 00 0
0180: 00 00 00 00 00 00 00 00 00 00 00 0
0190: 00 00 00 00 00 00 00 00 00 00 00 0
01a0: 00 00 00 00 00 00 00 00 00 00 00 0
01b0: 00 00 00 00 00 00 00 00 00 00 00 0
01c0: 00 00 00 00 00 00 00 00 00 00 00 0

@AABBCCDDEEFFG

Trace more  Trace less  Trace these too:
Clear Log

cycle ab db rw Fetch pc a x y s p
```

**http://www.visual6502.org/JSSim/expert.html**

# Le 6502 : connexion avec l'extérieur




- Accès à la mémoire :
  - $A_0$, ..., $A_{15}$ : 16 bits d'adresse ;
  - $D_0$, ..., $D_7$ : 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 ;

- constituée de circuits élémentaires : bits, «*binary digits*», contenant «0» ou «1» ;
- toute information, données et instructions, est stockée sous forme d'une suite de bits ;
- découpée en cellules mémoires : les «mots mémoires» ou «*words*» ;
  - un mot est constitué d'une suite de bits définissant sa taille ;
  - mot de 1 bit, mot de 4 bits (quartet ou «*nibble*»), 8 bits (octet ou «*byte*»), 16 bits, 32 bits, 64bits.
  - chaque mot est repéré dans la mémoire par une **adresse**, c-à-d un numéro qui identifie le mot mémoire.
  - un mot est un **contenant** accessible par son **adresse**.

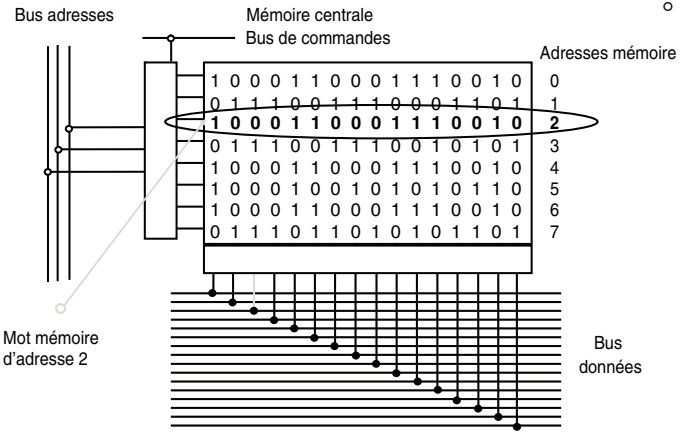
**Accès en écriture ou en lecture**

On utilise 3 bus :

- ▷ de données ;
- ▷ de commandes ;
- ▷ d'adresse.

L'utilisation d'un bus ressemble à l'utilisation d'un tuyau d'eau : une fois connecté, il échange son contenu :

- soit plein pour indiquer la valeur «1» ;
- soit vide pour indiquer la valeur «0».




- sur le schéma :
  - la mémoire a une capacité de 8 mots de 16 bits ;
  - la sélection d'une case mémoire se fait à l'aide du «bus d'adresse» où l'on code l'adresse de cette case mémoire :
    - pour accèder à la case d'adresse 3 :
      on met «011» sur le bus d'adresse ;
    - suivant la valeur du bus de commande : ;
    - «0» pour lire : on récupére le contenu du mot sur le bus de données ;
    - «1» pour écrire : on stocke le contenu du bus de données dans le mot.

# Valeur et représentation : illustration en C

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 decimale : %d\n",caractere);
 printf("Variable valeur_16_bit\n");
 printf("  elle tient sur %ld octets\n",sizeof(short int));
 printf("  sous forme decimale : %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 decimale : %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 decimale : %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
gnee);
}
```

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 decimale : 65
Variable valeur_16_bit
  elle tient sur 2 octets
  sous forme decimale : 513
  sous forme hexadecimale : 201
  Premier octet de la variable
valeur_16_bit
    sous forme decimale : 1
    sous forme hexadecimale : 1
  Second octet de la variable
valeur_16_bit
    sous forme decimale : 2
    sous forme hexadecimale : 2
  BigEndian ou LittleEndian ?
Variable valeur_signee en
entier : -43
Variable valeur_non_signee en
entier : 213
  signee ffffffd5 et non signee
d5
```

*Quelle est la taille par défaut d'un* `int` *?*

*4 octets d'après le dernier affichage* `ffffffd5`

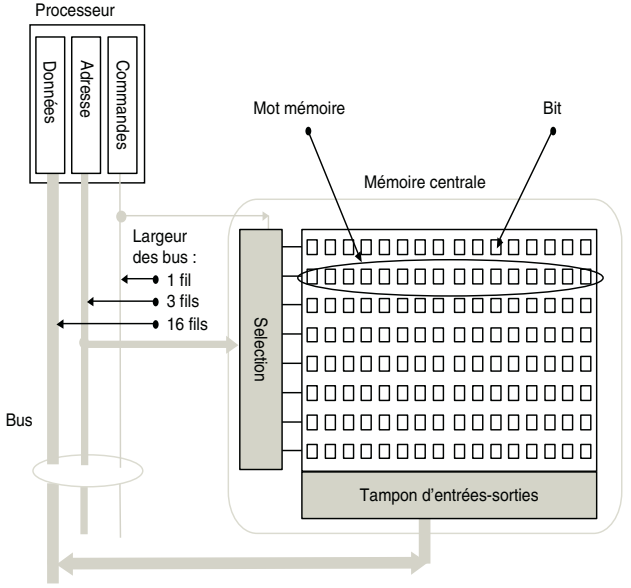
❶⇒crée un pointeur d'octet et le faire pointer sur le premier octet de l'entier qui en compte deux ;

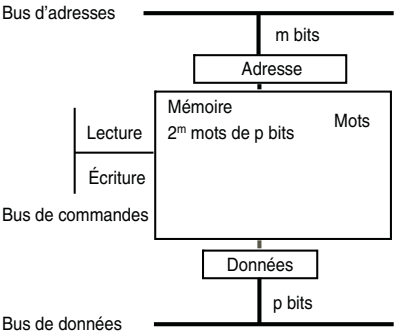
❷⇒affiche la valeur pointée, c-a-d la valeur du premier octet ;

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 *`).

# Le bus de communication

La largeur du bus d'adresse définit la **capacité d'adressage** du processeur.

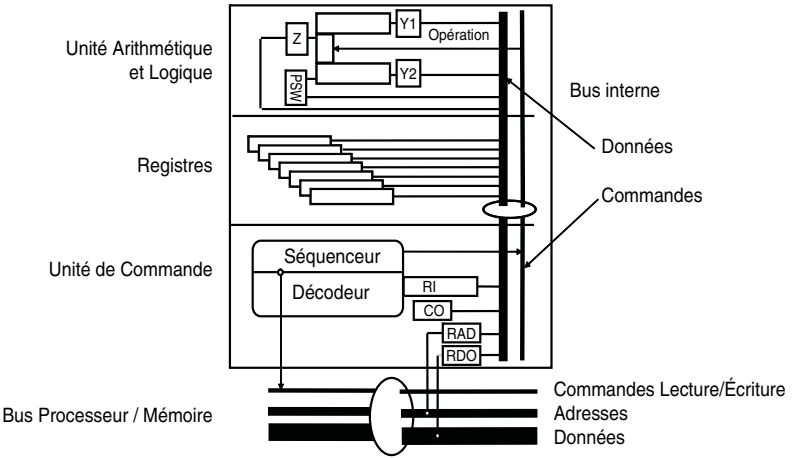



| | |
|---|---|
| Fonctions : | stockage programme et données |
| Unités de stockage : | Bit, Octet, Mot. |
| Adressage : | Mot |
| Temps d'accès. | |
| Technologies. | |
| Coûts. | |

- le bus d'adresses a une largeur de $mbits$ ;
- le bus de données à une largeur de $pbits$ ;
- la capacité de stockage en bits de cette mémoire est de $2^m$ mots de $pbits$.

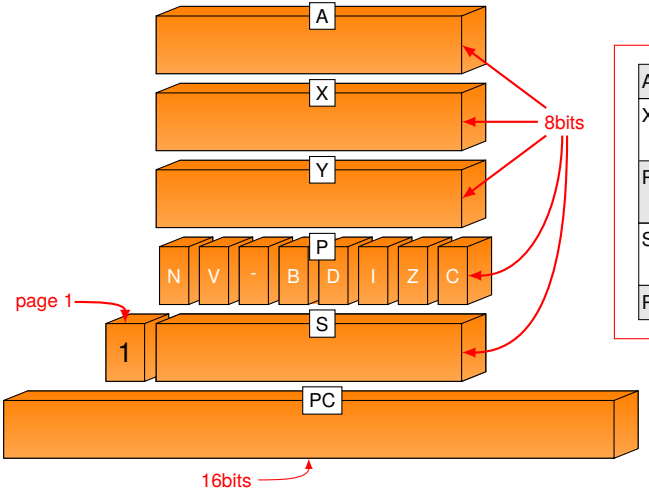
*Le bus de commande indique qu'elle type d'opération on désire réaliser (ici, 1 bit indique lecture ou écriture).*




- les registres : zones de mémorisation internes au processeur :
  - nombre et taille varient suivant le type du processeur ;
  - de type données : contient le contenu d'un mot mémoire ;
  - de type adresse : contient l'adresse d'un mot mémoire ;
  - spécifique : fonction précise (adresse de la pile par exemple) ;
  - générique : sert à stocker des résultats intermédiaires (calculs de l'UAL) ;

- l'Unité Arithmétique et logique :
  - réalise tous les calculs possibles du processeur : addition , soustraction, multiplication, comparaison *etc* (pour les calculs flottants on utilise une FPU) ;
  - possède des registres d'entrées $E_1$ et $E_2$ et un registre de sortie $S$ ;
  - possède un registre d'état, le «PSW», «Program Status Word» : qui donne l'état du calcul réalisé : dépassement de capacité, valeur à zéro *etc.*
- l'unité de commande : exécute les instructions machine à partir des registres et de l'UAL
  - possède des registres : CO, «Compteur ordinal», RI, «Registre d'instruction», RAD, «Registre d'ADresse», RDO, «Registre de DOnnées» ;
  - le CO est un registre d'adresse qui contient l'adresse de la **prochaine instruction** à exécuter et il avance à chaque chargement d'instruction.
  - le décodeur : sert à analyser l'instruction du RI et contôle le séquenceur qui manipule les $\mu$ instructions.

# Le processeur 6502

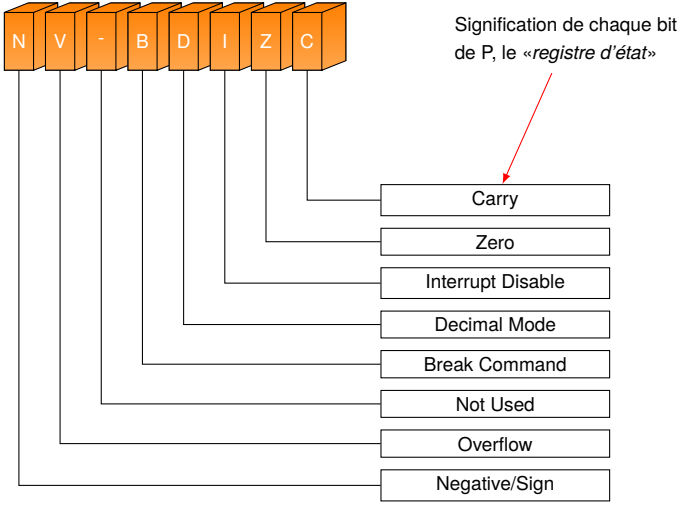



Registres

| | | |
|---|---|---|
| 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 |

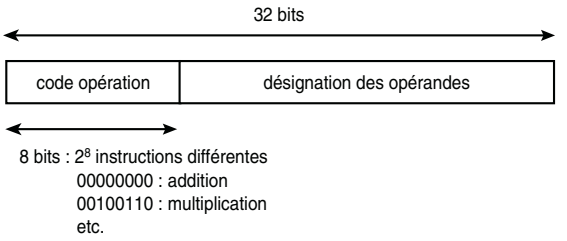
Explications du registre d'état

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



Une instruction machine :

- «code opération» : type d'opération à effectuer ;
- «opérandes» : données sur lesquelles l'opération définie par le code doit être réalisée :
  - soit un mot mémoire ;
  - soit un registre du processeur ;
  - soit une valeur immédiate ;




Une instruction du langage d'asssemblage :

| étiquette | code opération | désignation des opérandes |
|---|---|---|

- «étiquette» : *non obligatoire*, indique l'adresse de l'instruction machine ;
- code opération : mnémonique correspondant à une instruction machine ;
- opérandes : notation humaine désignant les opérandes de l'instruction machine ;

l'instruction en langage d'assemblage

| étiquette | code opération | opérandes |
|---|---|---|
| boucle : | ADD | Rg2 R0, R1 |

correspond à l'instruction machine

| adresse | code opération | opérandes |
|---|---|---|
| 01110110 | 00000000 | 111 0000 0001 |

*L'utilisation d'étiquette dans un programme en langage d'assemblage permet de définir la notion de saut : passage du CO d'une adresse mémoire à une autre, avec éventuellement un retour à la valeur du CO antérieure.*

| 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 ⟹ 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*

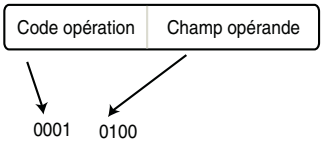
| Ins | description | mode adressage | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | immédiat | absolu | page zéro | indexé X | indexé Y | implicite | relatif |
| ADC | ajoute un octet avec le bit de retenu dans l'accumulateur | 69 | 6D | 65 | 7D | 79 | | |
| BEQ | «*Branch if EQual*», saut 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 | | 4C | | | | | |
| 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 | | |
| NOP | ne fait rien | | | | | | EA | |

# Le processeur : exécution d'un programme, exemple

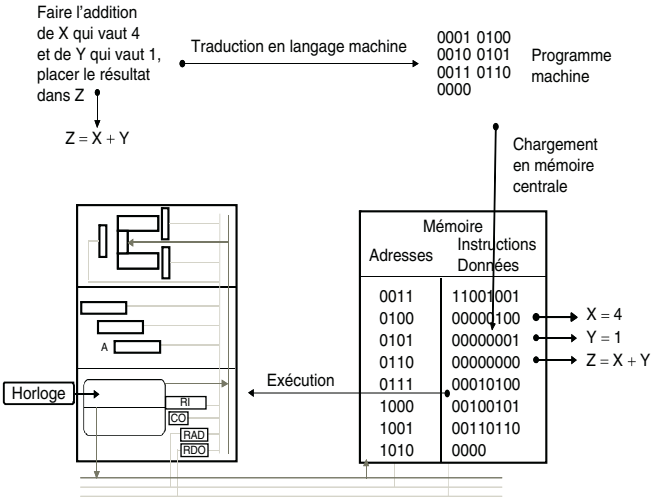
**Problème** : réaliser l'addition de X, valant 4 avec Y, valant 1, et placer le résultat dans Z.

C-à-d `Z = X + Y` avec `X=4` et `Y=1`.

- les mots sont sur 8 bits, les instructions et les données sont codées sur un mot mémoire ;
- X : `00000100`, Y : `00000001`
- le code opération est sur 4bits, le champs opérande sur 4 bits ;
- une adresse implicite est utilisée dans une instruction :le registre Accumulateur ou «A» : l'addition porte sur la donnée spécifiée et le contenu de A, le résultat étant placé dans A.




| | | |
|---|---|---|
| 0001 | 0100 | Charger le contenu du mot mémoire d'adresse 0100 dans le registre A |
| 0010 | 0101 | Additionner le contenu du registre A avec le contenu du mot mémoire d'adresse 0101 et placer le résultat dans le registre A |
| 0011 | 0110 | Placer le contenu du registre A dans le mot mémoire d'adresse 0110 |
| 0000 | | Stopper le programme |




- X est stocké à l'adresse `0100` et Y à l'adresse `0101`, le résultat Z à l'adresse `0110` ;
- le programme machine est chargé à l'adresse `0111` ;
- le CO est chargé avec l'adresse `0111`.

**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 définie 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 contenu 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
```

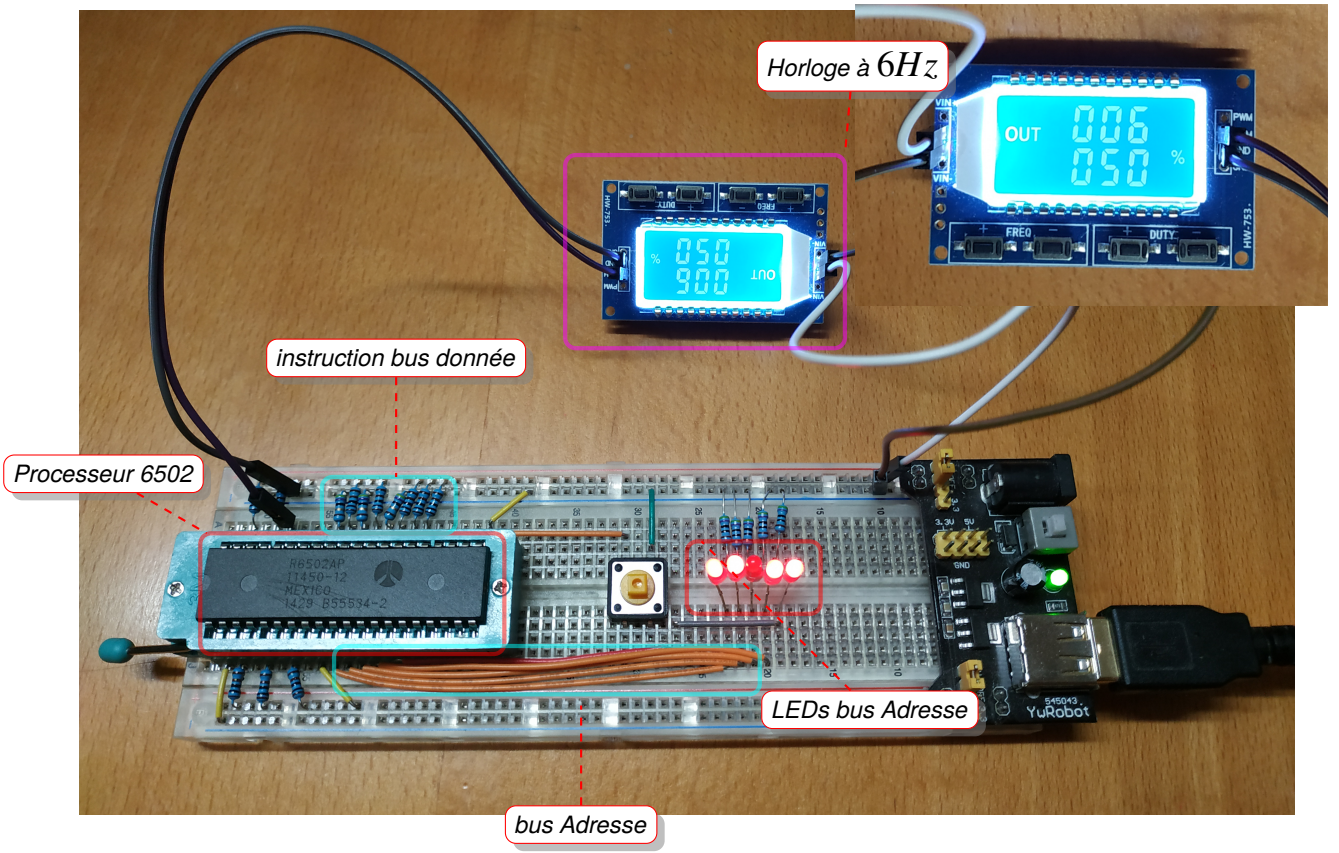
```
0600: ad 2e 06 8d 00 02 69 01 85 00 ad 3c 06 69 01 85
0610: 01 a2 01 a0 01 bd 2e 06 59 3c 06 9d 00 02 e8 e4
0620: 00 f0 0a c8 c4 01 d0 ed a0 01 4c 15 06 00 0d 68
0630: 65 6c 6c 6f 20 62 6f 6e 6a 6f 75 72 09 74 6f 70
0640: 73 65 63 72 65 74
```

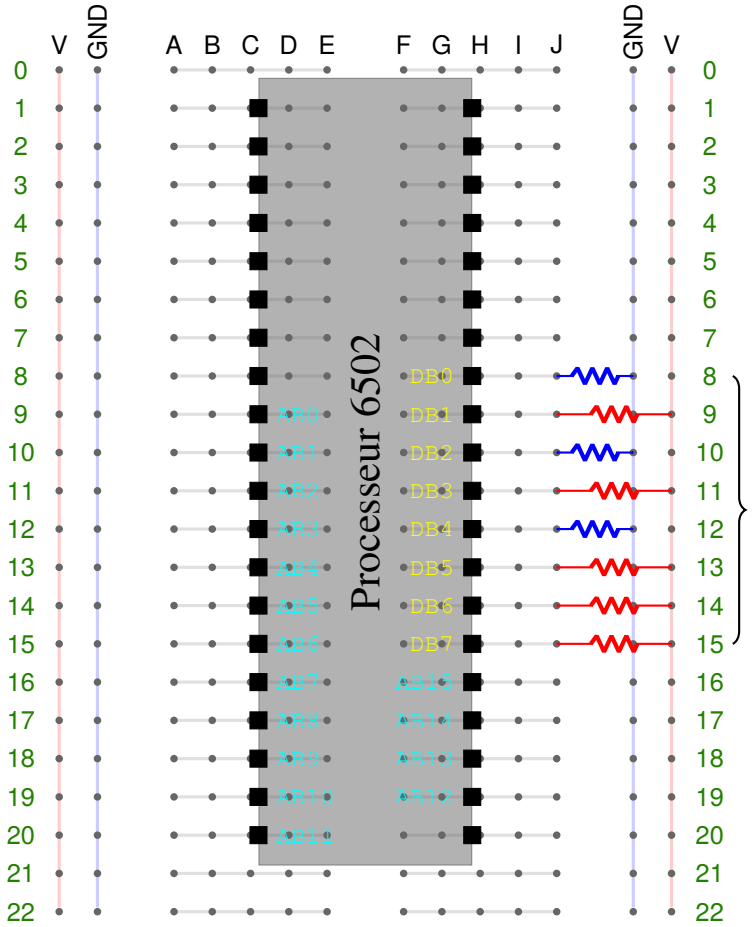
On note que :

| | |
|---|---|
| `$062e` | adresse de la chaîne saisie |
| `$063c` | adresse de la chaîne mdp |
| `$062d` | adresse de l'instruction `brk` |
| `$062e` | le désassembleur trouve des instructions dans le contenu de la chaîne saisie<br>$\Longrightarrow$ Interprétation automatique erronée |
| `$063e`<br>`$0643` | Interprétation automatique impossible, il n'y a pas d'instruction reconnue |



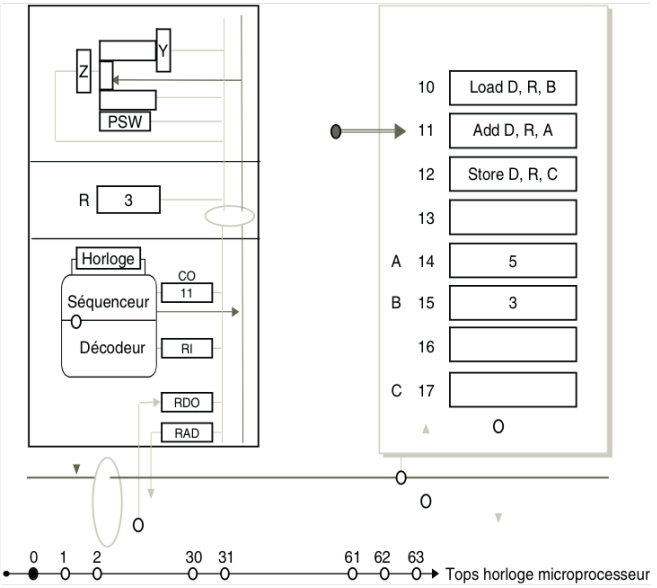
# 6502 sur «*breadboard*»






⟹11101010 de `DB7` à `DB0`
ce qui donne en hexa $\{1110\}_2\{1010\}_2 = \{E\}_{16}\{A\}_{16}$

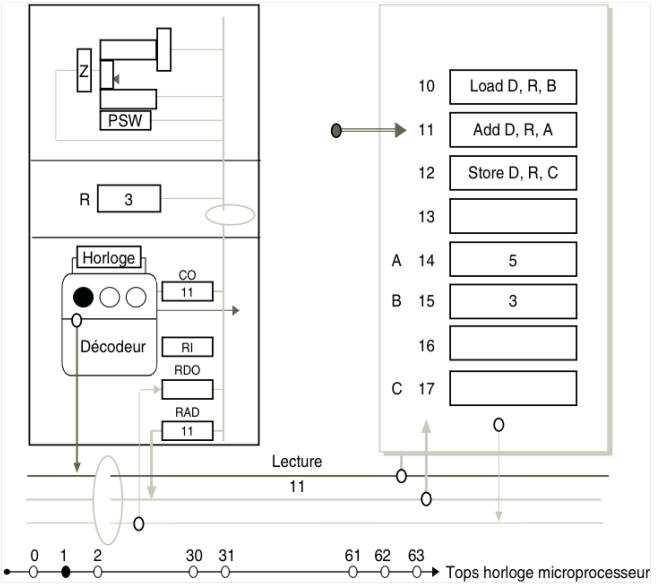
`EA` en instruction 6052 ⟹ `NOP` pour «*No OPeration*»
Ce qui veut dire que le processeur ne fait rien
Il passe seulement à l'instruction suivante
⟹il incrémente le CO, «*Compteur ordinal*», ou «*instruction counter*»
⟹l'adresse est incrémentée sur le bus d'adresse `AB0` à `AB15`




*exécution* ❶

CO pointe sur 11, le RAD passe sur l'instruction et la charge dans RI

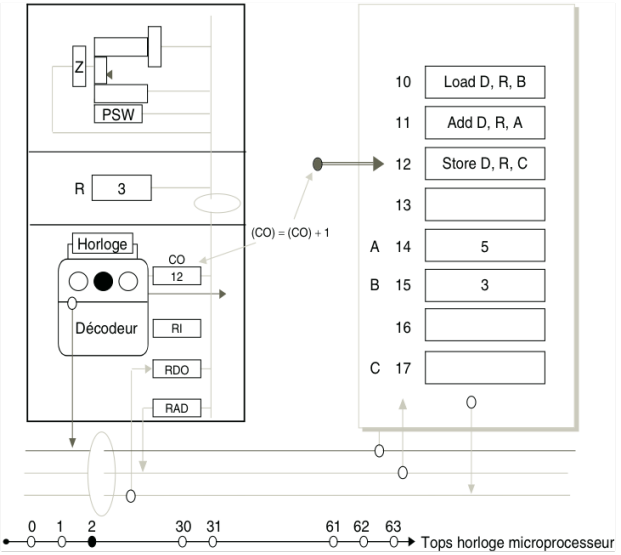
*Le registre R, contient la valeur de B (3), résultat de l'instruction 10.*




*exécution* ❷

Le résultat du passage sur l'instruction 11 :

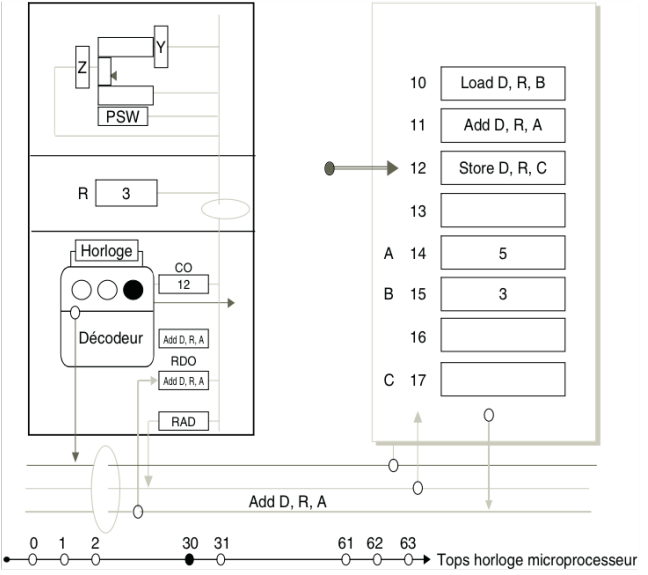
- positionner l'adresse de l'instruction sur le bus d'adresse ;
- charger l'instruction «`Add D,R,A`» depuis le bus de données dans RI ;




*exécution* ❸

CO est incrémenté sur 12.

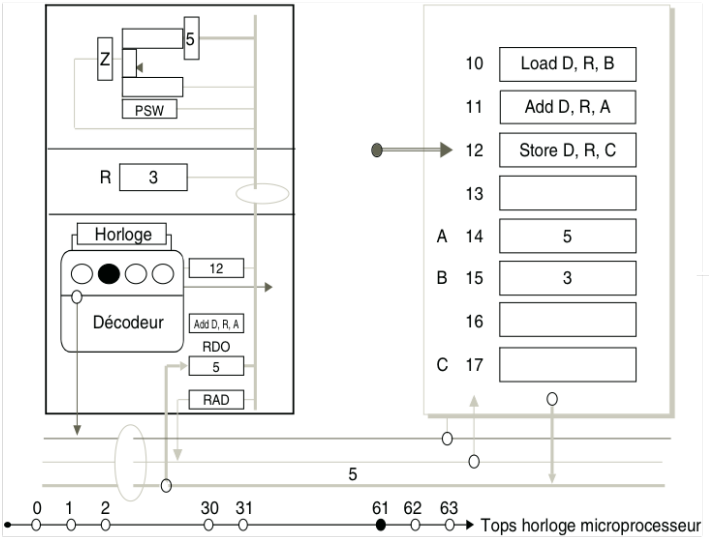
La mémoire est accédée suivant le contenu du registre d'adresse (11).




*exécution* ❹

L'instruction est retournée sur le bus de données pour être chargée dans RI.

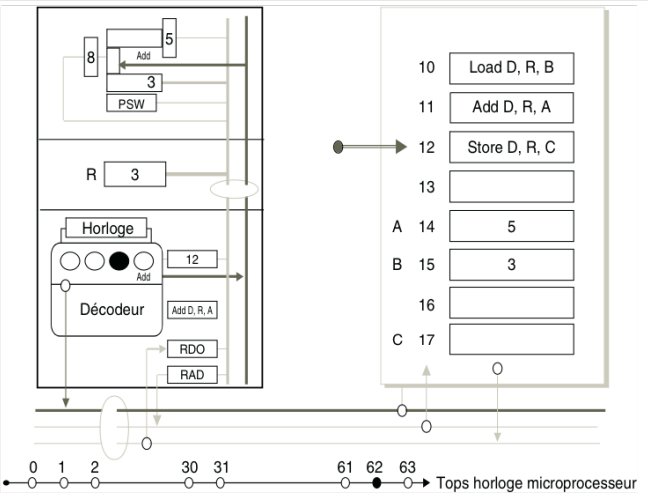
Le décodeur va commencer à interpréter l'instruction 11 chargée dans le RI (tops d'horloge).




*exécution* ❺

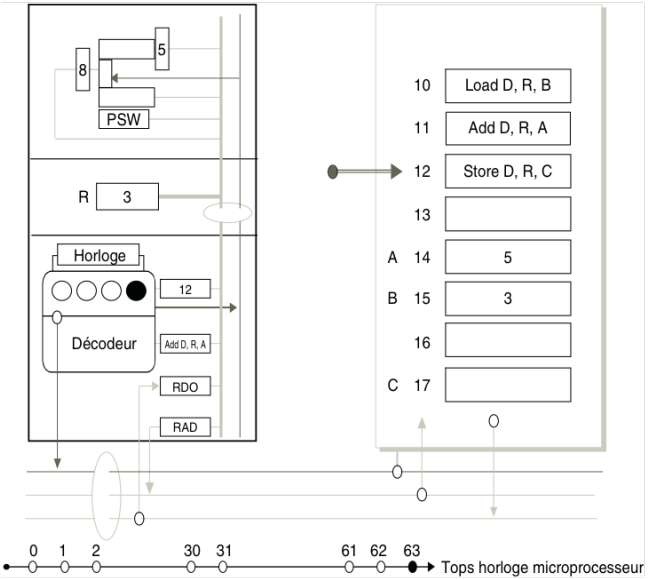
L'instruction demande l'accès à la variable A :

- chargement dans le registe d'adresse de 14 ;
- échange sur le bus de données de la valeur de A qui est 5 ;
- placement de la valeur 5 dans l'ALU.




*exécution* ❻

La valeur de R, 3, est placée dans l'ALU.
Le décodeur sélectionne l'opération «Add» de l'ALU.
Le résultat de l'addition, 8, est disponible.

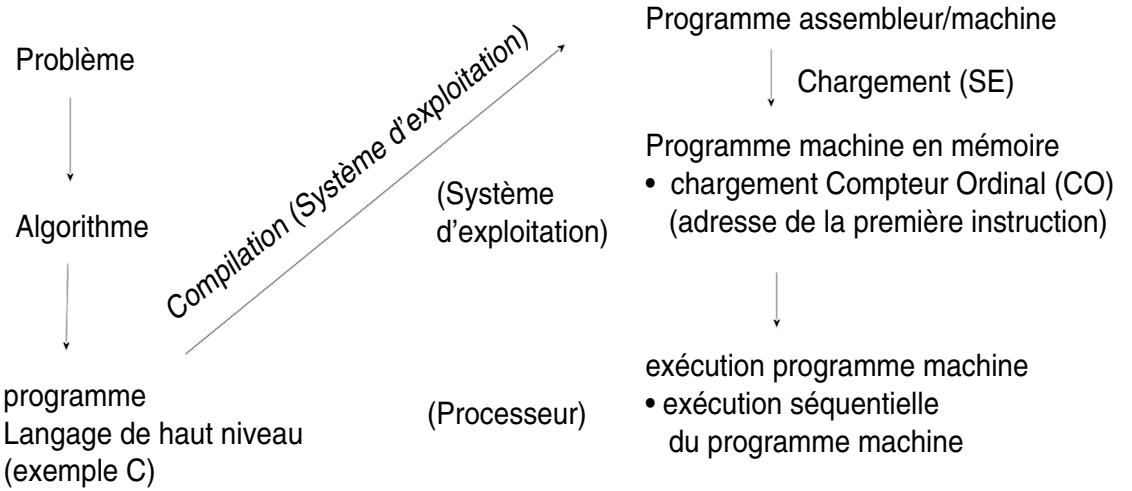


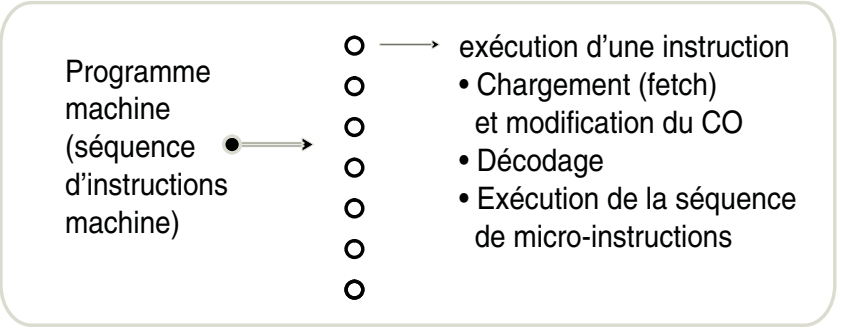

**Exécution ❼**

Le décodage de l'instruction 11 est terminée : le décodeur est arrivé à la fin de son cycle.

Le registre d'état, PSW contient un état qui dépend du déroulement de l'instruction de l'ALU (en particulier, s'il y a eu dépassement de capacité lors de l'addition, mais aussi de signe, de nullité).

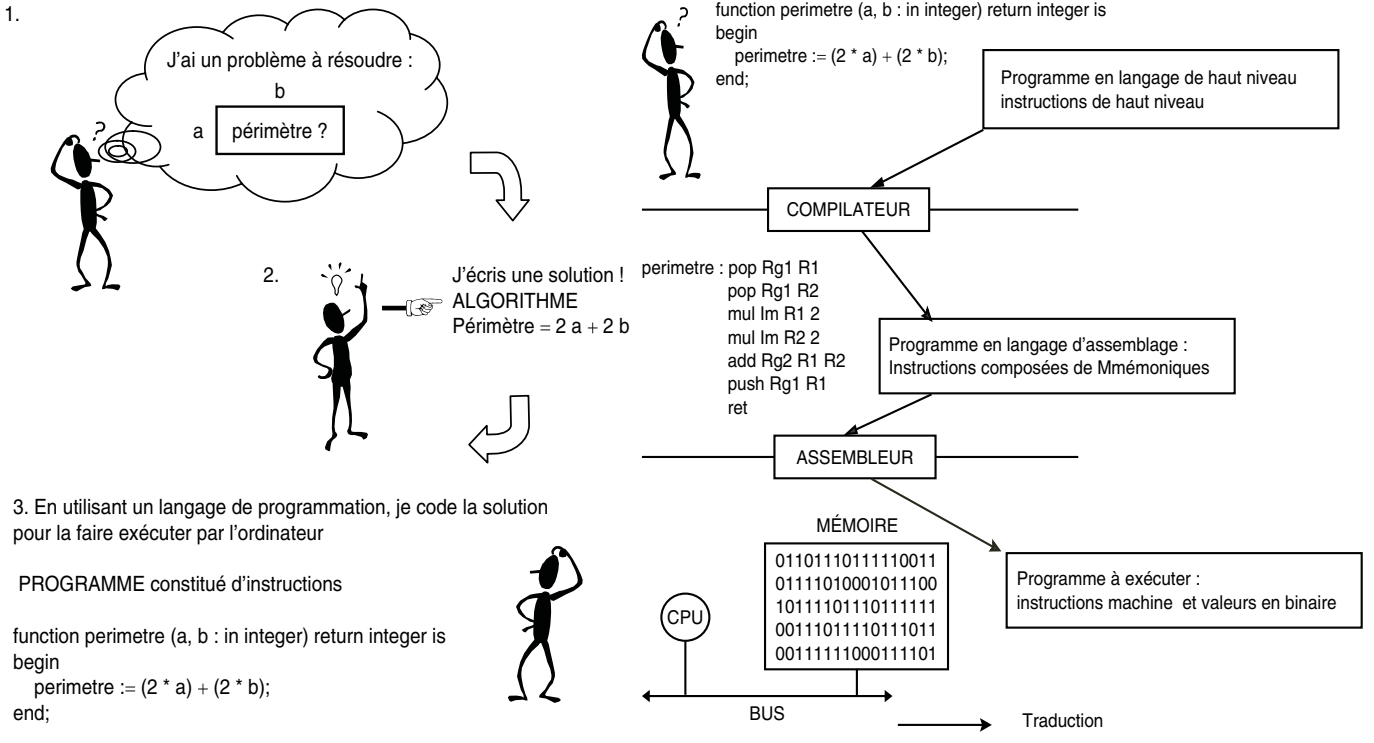
Le nombre de tops d'horloge qui a été nécessaire est important et varie avec la nature de l'instruction : plus important si on a besoin d'accéder à la mémoire centrale.



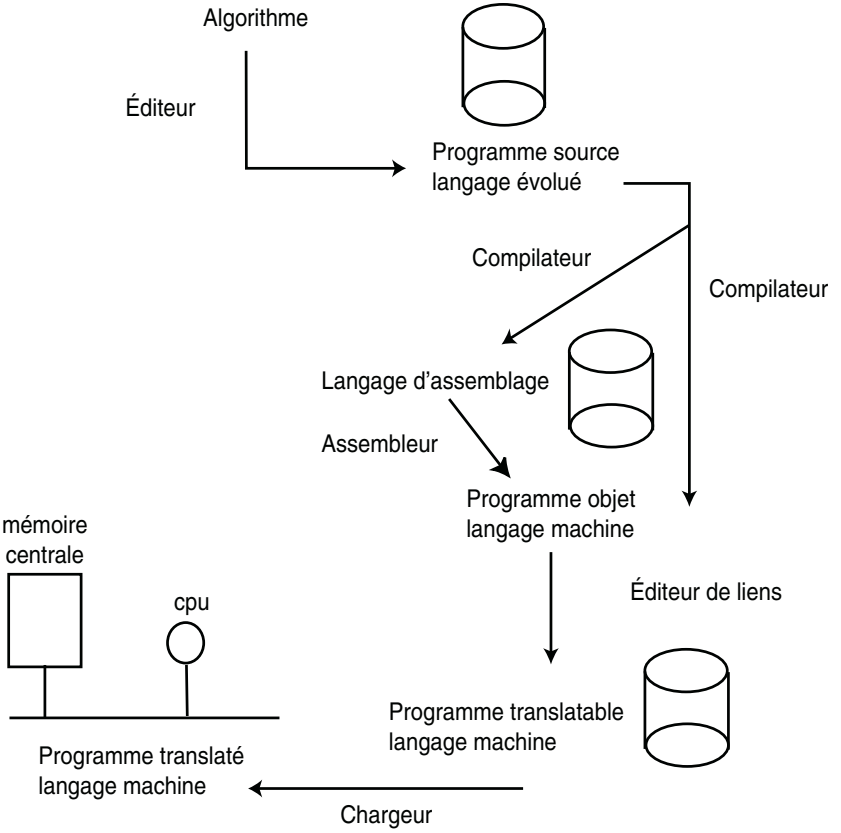

*Un des rôles du Système d'Exploitation, ou «OS», «operating system» est de chargé le programme dans la machine.*

- 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.

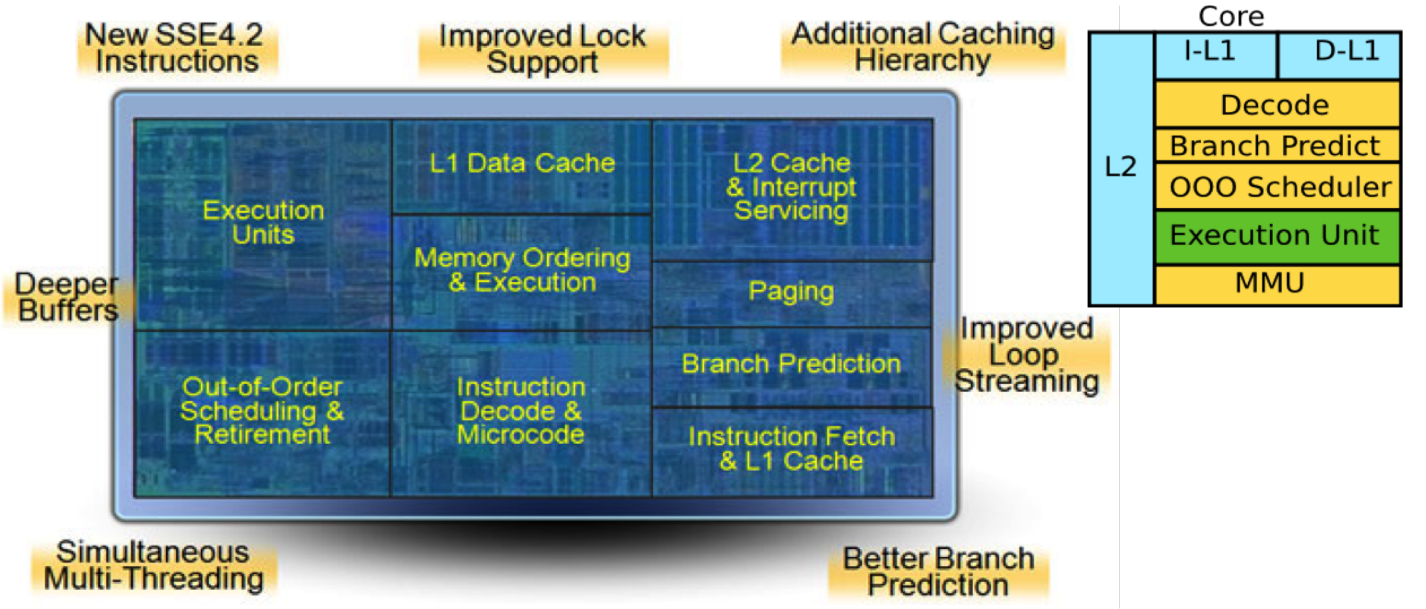



*Les «mnémoniques» sont des appellations mémorisables par un humain, correspondant aux instructions machine.*

# Du problème à sa résolution : point de vue utilisateur



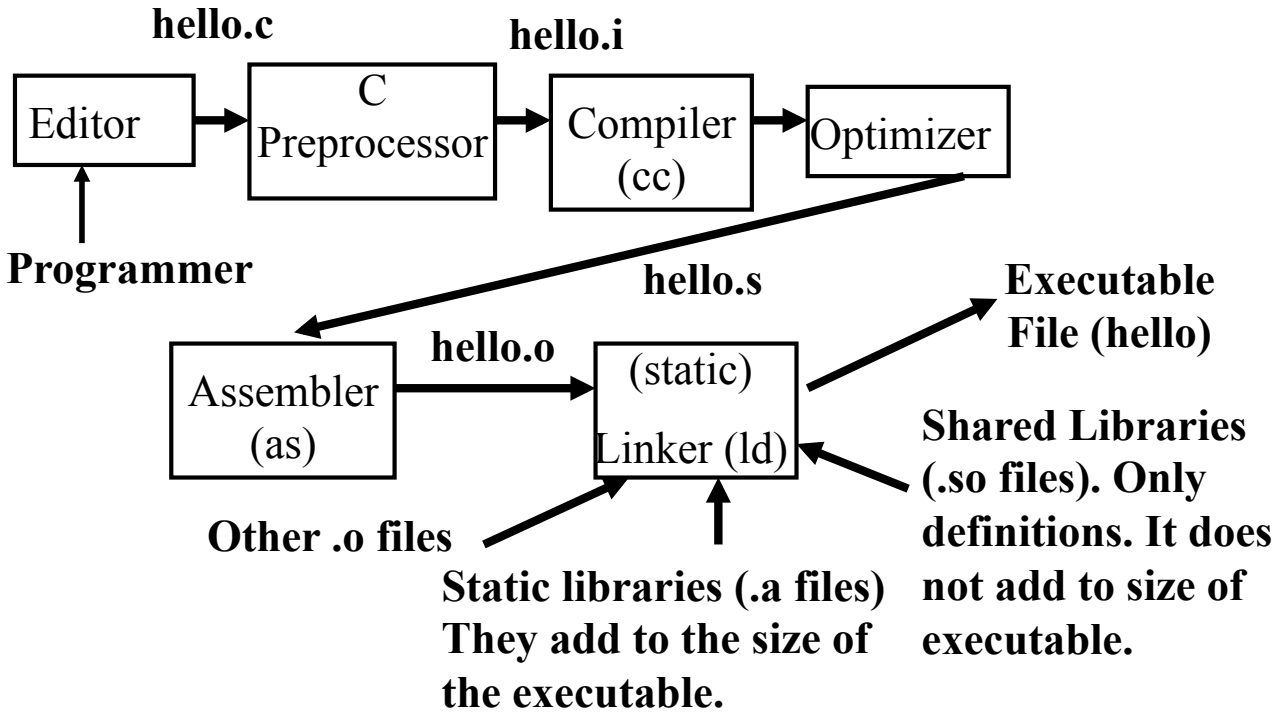
# Un processeur moderne




Moins de 10%de la surface sert à l'exécution réelle

*«OOO» : Out of Order*

**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 assembling)
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.*



# Compiler du C vers le 6502



```
#include <stdio.h>

void main()
{
  printf("Hello\n");
}
```

```
;
; File generated by cc65 v 2.18 - Git df07e23f
;
   .fopt       compiler,"cc65 v 2.18 - Git df07e23f"
   .setcpu      "6502"
   .smart      on
   .autoimport   on
   .case       on
   .debuginfo   off
   .importzp   sp, sreg, regsave, regbank
   .importzp   tmp1, tmp2, tmp3, tmp4, ptr1, ptr2, ptr3, ptr4
   .macpack   longbranch
   .forceimport   __STARTUP__
   .import      _printf
   .export      _main

.segment   "RODATA"

S0001:
   .byte   $48,$65,$6C,$6C,$6F,$0A,$00

; ---------------------------------------------------------------
; void __near__ main (void)
; ---------------------------------------------------------------

.segment   "CODE"

.proc   _main: near

.segment   "CODE"

   lda     #<(S0001)
   ldx     #>(S0001)
   jsr     pushax
   ldy     #$02
   jsr     _printf
   rts

.endproc
```

```
#include <stdio.h>

void main()
{
  int i=0;
  for (i=0;i<5;i++)
    printf("%d\n",i);
}
```

```
   .fopt       compiler,"cc65 v 2.18 - Git df07e23f"
   .setcpu       "6502"
   .smart      on
   .autoimport    on
   .case       on
   .debuginfo    off
   .importzp    sp, sreg, regsave, regbank
   .importzp    tmp1, tmp2, tmp3, tmp4, ptr1, ptr2, ptr3, ptr4
   .macpack    longbranch
   .forceimport    __STARTUP__
   .import      _printf
   .export      _main
.segment    "RODATA"
S0001:
   .byte    $25,$64,$0A,$00
```

```
.segment    "CODE"
.proc    _main: near
.segment    "CODE"

   ldx      #$00
   lda      #$00
   jsr      pushax
   ldx      #$00
   lda      #$00
   ldy      #$00
   jsr      staxysp
L0003:    ldy      #$01
   jsr      ldaxysp
   cmp      #$05
   txa
   sbc      #$00
   bvc      L000A
   eor      #$80
L000A:    asl      a
   lda      #$00
   ldx      #$00
   rol      a
   jne      L0006
   jmp      L0004
```

```
L0006:    lda      #<(S0001)
   ldx      #>(S0001)
   jsr      pushax
   ldy      #$03
   jsr      ldaxysp
   jsr      pushax
   ldy      #$04
   jsr      _printf
   ldy      #$01
   jsr      ldaxysp
   sta      regsave
   stx      regsave+1
   jsr      incax1
   ldy      #$00
   jsr      staxysp
   lda      regsave
   ldx      regsave+1
   jmp      L0003
L0004:    jsr      incsp2
   rts

.endproc
```

- `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
```



- `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

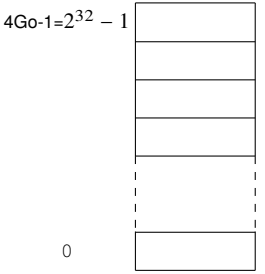
```
$ nm hello
080484bc R _IO_stdin_used
         w _Jv_RegisterClasses
0804a014 A __bss_start
0804a00c D __data_start
08048470 t __do_global_ctors_aux
08048350 t __do_global_dtors_aux
08049f14 d __init_array_end
08049f14 d __init_array_start
08048460 T __libc_csu_fini
080483f0 T __libc_csu_init
         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.*



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    return 0;
7 }
```

*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_word.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_word.c
$ size hello_world
   text      data       bss       dec       hex   filename
 668942      3316      5892    678150     a5906   hello_world
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int global_init_data = 30;
int global_noinit_data;

void function_prints(void)
{ char c;
  int local_data = 1;
  int *dynamic_data = (int *)calloc(1,sizeof(int));

  printf("Pid of the process is = %d\n", getpid());
  printf("Addresses which fall into:\n");
  printf("1) Data  segment = %p\n", &global_init_data);
  printf("2) BSS   segment = %p\n", &global_noinit_data);
  printf("3) Code  segment = %p\n", &function_prints);
  printf("4) Stack segment = %p\n", &local_data);
  printf("5) Heap  segment = %p\n", dynamic_data);
  scanf("%c", &c);
}
```

```c
int main()
{
  function_prints();
  return 0;
}
```

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 = 0x8d76008
```

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
08d76000-08d97000 rw-p 00000000 00:00 0        [heap]
bfe28000-bfe49000 rw-p 00000000 00:00 0        [stack]
```

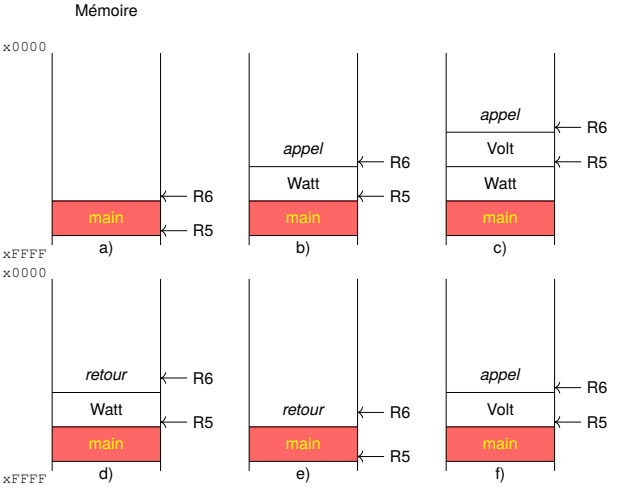
xterm

```
$ nm -f sysv segments
Symbols from segments:
Name                  Value   Class   Type     Size      Section
main                |08048558|   T  |     FUNC|00000012||.text
global_init_data    |0804a020|   D  |   OBJECT|00000004||.data
global_noinit_data  |0804a02c|   B  |   OBJECT|00000004||.bss
function_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» ;

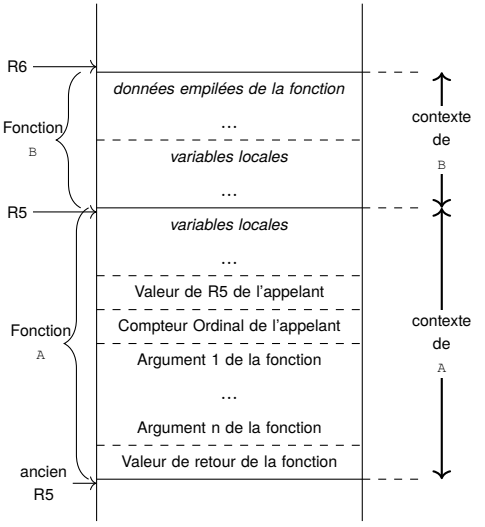
c. le registre R5 prend la valeur de R6 et pointe sur le «contexte» associé à l'exécution de la fonction, R6 sur le nouveau sommet qui va être décalé pour faire de la place aux arguments, valeur de retour et adresse de retour (empilement) ;

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 ;

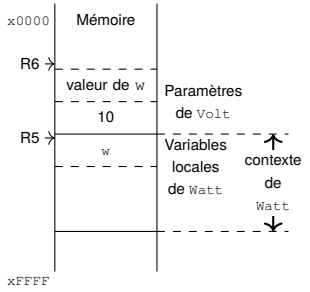
f. nettoyage de la pile pour l'exécution d'une autre fonction (dépilement valeur retour et arguments précédents) ;

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 «déboguage» 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.*

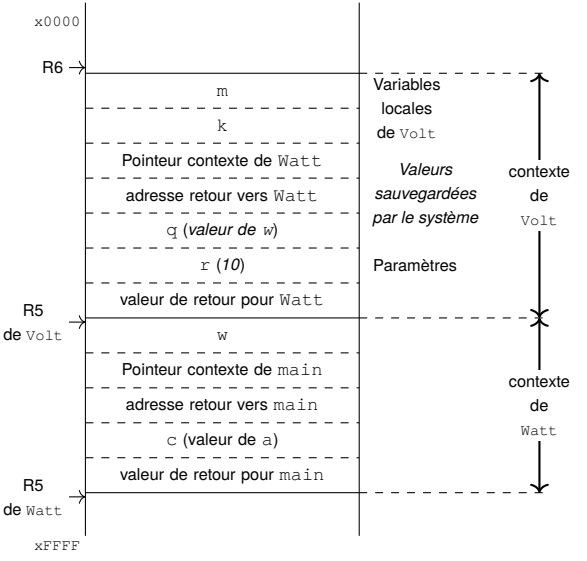



Lors de l'appel de la fonction «Volt», par la fonction «Watt», la valeur contenue dans la variable locale «w» de «Watt» est transmise en tant que paramètre à la fonction «Volt».

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

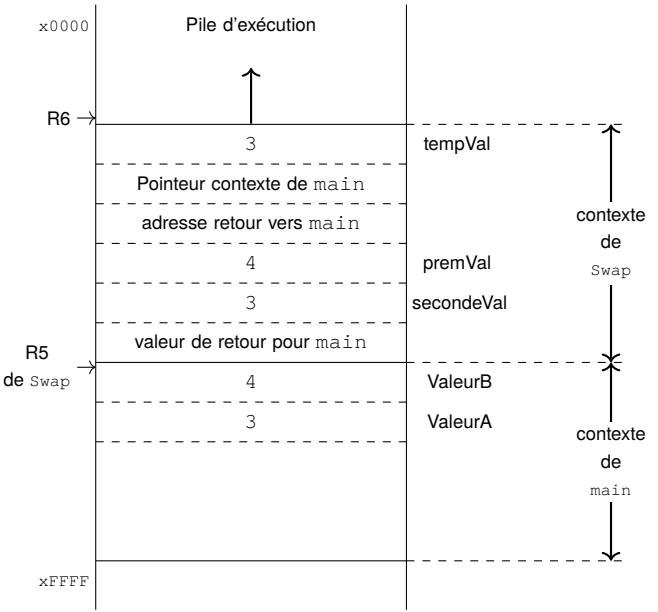
Dans le contexte de la fonction «Volt» :

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



**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   return;
22 }
```



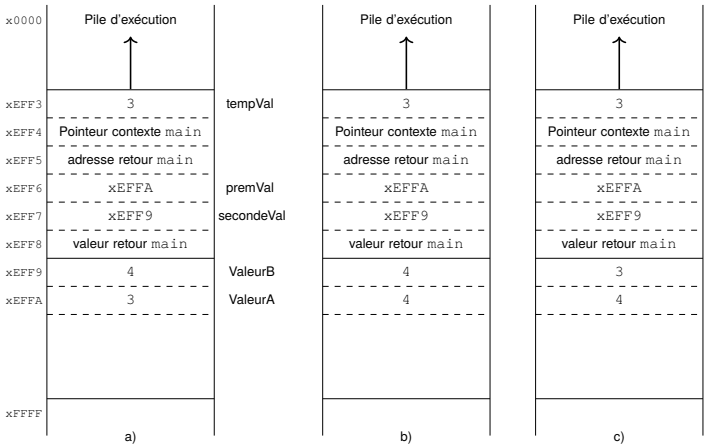

*L'état de la pile avant le retour de la fonction*

Que fait le programme ?

**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 `0xEFF9`;
- «`premVal`» et «`secondeVal`» sont des **pointeurs** :
  - «`premVal`» contient l'adresse de «`valeurA`», `0xEFFA`;
  - «`secondeVal`» contient l'adresse de «`valeurB`», `0xEFF9`;

- Définition de variables locales :

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

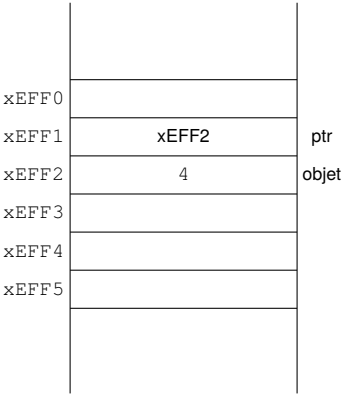
- Assignons des valeurs à ces variables :

```
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 1ère case est l'adresse du tableau…

**Attention**

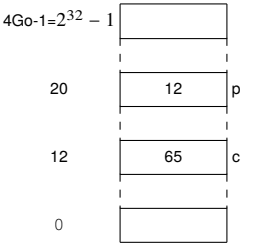
Il ne faut pas donner, comme valeur de retour d'une fonction, un **pointeur sur une variable locale** à cette fonction !
*Lors du retour de la fonction, le contenu de la pile est réutilisé, et la valeur retournée est écrasée.*



- 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**

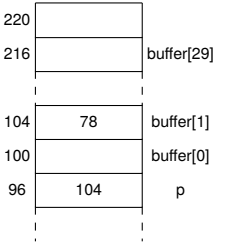
1. affecter une valeur numérique à un pointeur :

```
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 :

a. on mémorise où on en est dans le code que l'on exécute : sauvegarde de la valeur du **compteur ordinal** ;
b. on charge dans le compteur ordinal, l'**adresse de la première instruction** du code de la fonction ;
c. on **restaure l'ancienne valeur du compteur ordinal** lorsque l'on retourne de la fonction ;
d. 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.

# Programme *vs* Processus

**Programme** entité purement statique associée à la suite des instructions qui la composent.

**Processus** entité purement dynamique associée à la suite des actions réalisées par un programme.

- ◇ La notion de processus introduit implicitement le concept et le fonctionnement des *systèmes multiprogrammé*.
- ◇ Un processus est une abstraction de données définies par 2 parties : un **état** et un **comportement**.

À chaque étape d'exécution du programme :

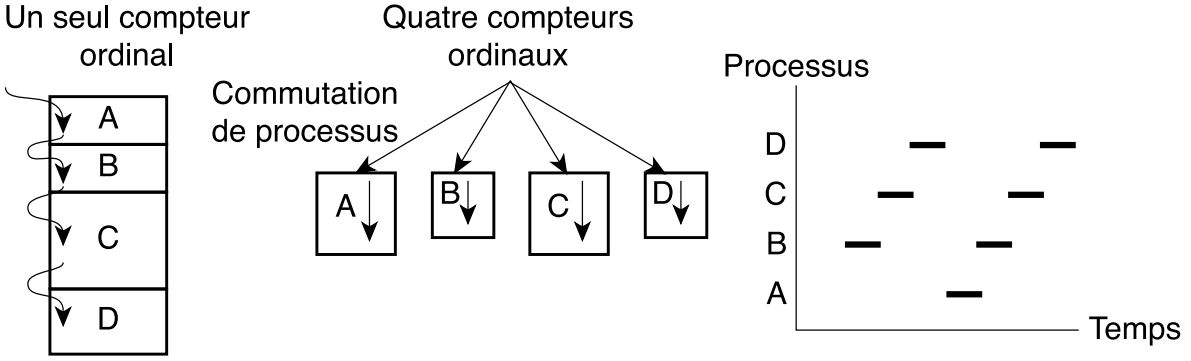
- ▷ le contenu des registres du compteur ordinal évolue ;
- ▷ le contenu de la mémoire centrale peut être modifié : écriture ou lecture.

*On appelle **processus** l'image de l'état du processeur et de la mémoire au cours de l'exécution d'un programme.*

Le programme est statique et le processus représente la dynamique de son exécution.

**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».

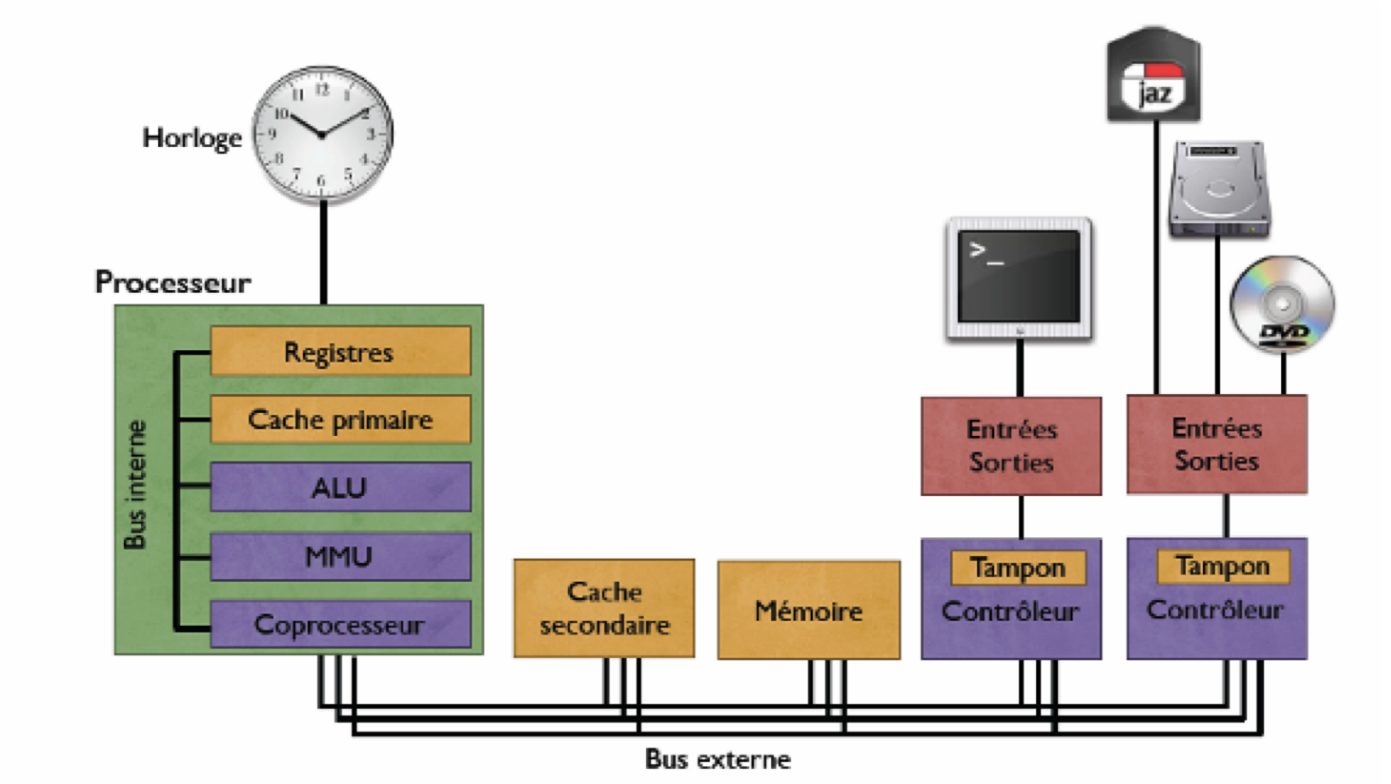
**Comment passer règulièrement et automatiquement d'un processus à l'autre ?**

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

# Représentation fonctionnelle d'un ordinateur



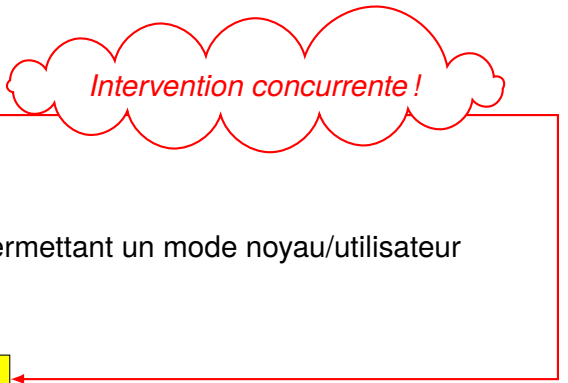
On ajoute dans cette représentation fonctionnelle, une horloge et des contrôleurs de périphériques



# Représentation fonctionnelle d'un ordinateur

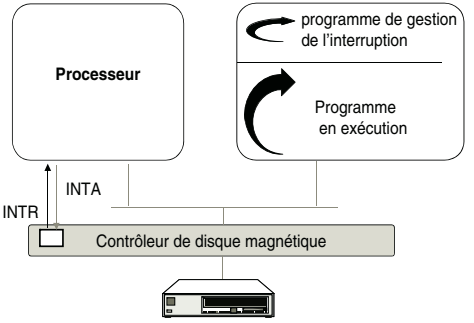
Les composants indispensables d'un système capable de supporter Unix :

- La mémoire principale
- La mémoire cache
- Les contrôleurs de périphériques
- Les bus de communication
- Une horloge
- Un processeur permettant un mode noyau/utilisateur
- Les interruptions



- permet d'arrêter l'exécution du programme en cours afin d'exécuter une tâche jugée plus urgente :
  «*Un périphérique peut signaler un événement important (fin de papier dans l'imprimante, dépassement de température signalée par une sonde placée dans un four...) au processeur qui, s'il accepte cette interruption, exécute un programme de service (dit programme d'interruption) traitant l'événement.*
  *À la fin du programme d'interruption le programme interrompu reprend son exécution normale.*»

- il en existe trois sortes provenant :
  - **d'un périphérique** : interruptions externes qui permettent à un périphérique d'avertir le processeur ;




  - **d'un programme en cours d'exécution** : interruptions logicielles internes nommées **appels systèmes**.
    Il s'agit de permettre à un programme en cours d'exécution de se dérouter vers un programme du système d'exploitation qui doit gérer une tâche particulière.
    *Par exemple les instructions d'entrées/sorties, permettant les échanges d'informations entre le processeur et les périphériques, sont traitées de cette manière :*
    *un ordre d'entrées/sorties est un appel au système (une interruption logicielle) qui interrompt le programme en cours au profit du programme spécifique (driver ou pilote) de gestion d'un périphérique*.

  - **du processeur** : traiter des événements exceptionnels de type division par zéro, dépassement de capacité

Principe

**commutation de contexte** provoquée par un **signal géré par le matériel**.

Ce **signal** est lui-même un événement qui peut être :

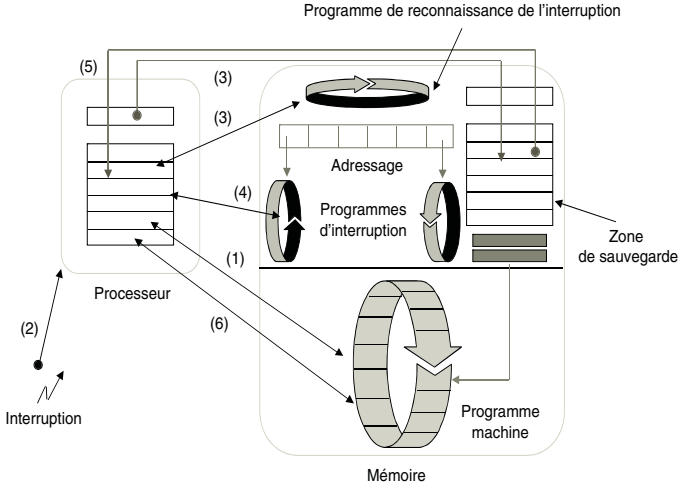
- ◇ interne au processus et donc résultant de son exécution ;
- ◇ extérieur indépendant de cette exécution.

Exemple

- ◇ L'événement concerne 1 périphérique d'entrée / sortie.
- ◇ L'événement concerne 1 utilisateur ou un opérateur.

Le signal :

- ▷ réalise un **changement de contexte processeur** lorsqu'il est matériel ;
- ▷ modifie un indicateur qui est **consulté régulièrement** par le système d'exploitation en vue de déterminer la cause de l'interruption lorsqu'il est logiciel.

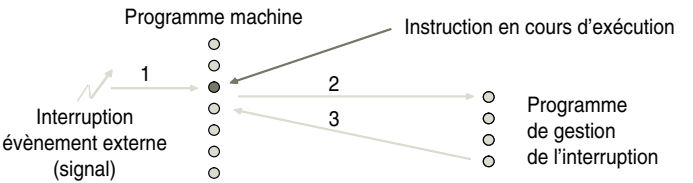



1. le programme utilisateur dispose du processeur. C'est lui qui est en cours d'exécution. Il dispose des registres, de l'unité arithmétique et logique, de l'unité de commande. Le compteur ordinal CO contient l'adresse de la prochaine instruction à exécuter ;
2. une interruption est déclenchée par un périphérique ;
3. sauvegarde du contexte matériel d'exécution du programme utilisateur et du compteur ordinal CO.
   Le programme de reconnaissance s'exécute et lit le numéro de l'interruption.
   Le numéro de l'interruption permet l'identification de l'adresse du programme de traitement ;
4. le compteur ordinal CO est chargé avec l'adresse du programme de traitement et celui-ci s'exécute ;
5. le contexte d'exécution du programme utilisateur est rechargé dans le processeur à la fin du programme d'interruption ;
6. le programme utilisateur reprend son exécution.

En version simplifiée :

Matériel — Prise en compte par le processeur d'événements externes (exemple : les périphériques positionnent un signal qui est reçu par le processeur).

Logiciel



**Gestion logicielle de l'horloge**

L'horloge est gérée par un «pilote d'horloge» pour fournir les services suivants :

- mettre à jour l'heure courante ;
- empêcher les processus de dépasser le temps qui leur est alloué ;
- comptabiliser l'allocation du processeur ;
- traiter l'appel système `alarm()` des processus utilisateur ;
- fournir des compteurs de garde au système lui-même ;
- fournir diverses informations au système (tracé d'exécution, statistiques).

**Différentes horloges**

⋆ **horloge «temps réel»** : maintenir la date et l'heure «humaine».

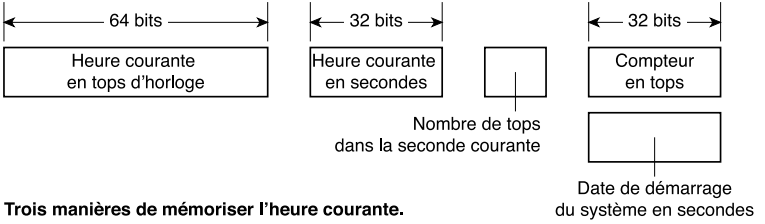
▷ une date de référence : le 1$^{er}$ janvier 1970 ;

▷ un compteur exprimant le nombre de «tops» depuis la date de référence ;

▷ capacité du compteur : dépend du nombre de bits du compteur et de la fréquence de l'horloge ;

*Exemple : un compteur sur 32bits, avec une horloge à $60Hz$ déborde en 2 ans…*

**Solutions :**




**Trois manières de mémoriser l'heure courante.**

◇ compteur sur 64bits : augmente la complexité du calcul qui doit être réalisé de nombreuses fois par seconde ;

◇ compteur sur 32 bits par seconde + compteur supplémentaire «tops → seconde»

***Solution retenue pour Unix : dépassement de capacité en 2038 !***

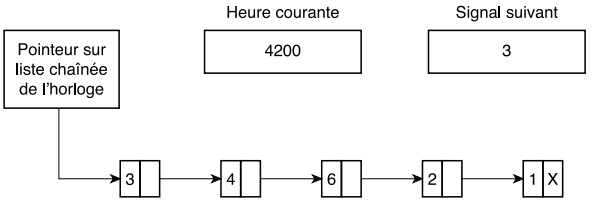
◇ compteur de tops à partir de l'heure du démarrage récupérée automatique (utilisateur ou réseau).

**Autres services**

- **Ordonnanceur** : garantie qu'un processus ne s'exécute pas plus logntemps que son «quantum de temps» :
  - lorsqu'un processus démarre **un compteur lui est associé** avec un nombre de «tops» d'horloge ;
  - à chaque interruption de l'horloge **ce compteur est décrémenté** ;
  - lorsque le compteur est à zéro, **l'ordonnanceur est rappelé** et choisi **un nouveau processus**.
- **Temps d'allocation d'un processus** : comptabiliser le temps alloué à chacun des processus :
  - on incrémente un compteur associé au processus à chaque top d'horloge ;
  - on distingue le temps passé en mode noyau et en mode utilisateur au service de ce processus : *on indique également le temps réel attendu par l'utilisateur devant son écran.*

```
pef@darkstar:~$ time ls
Segments  segments.c

real  0m0.010s
user  0m0.004s
sys 0m0.004s
```

- **Alarmes** : chaque processus peut demander d'être alerté après un certain temps ou à une certaine heure.



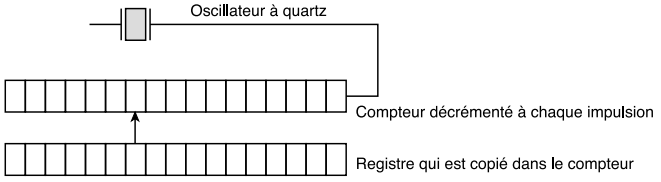

**Simulation de plusieurs compteurs à partir d'une horloge unique**

Optimisation : une seule horloge + une liste chaînée d'alarmes qui mémorise les délais entre chacune de ces alarmes : *sur la figure les signaux vont être envoyés pour* 4203, 4207, 4213, 4215 *et* 4216

**Quels usages ?**

- conserver la date et l'heure courante : fournir cette date à l'utilisateur, estampiller les fichiers *etc.*
- gérer les alertes : alertes pour le noyau ou les programmes utilisateur ;
- générer les «**tops d'horloges**» : nécessaire au fonctionnement du système d'exploitaiton.

**Comment ça marche ?**




**Une horloge programmable.**

L'oscillateur à quartz permet une meilleur qualité de signal périodique, que celui de $50Hz$ ou $60Hz$ du courant électrique : plus régulier, compris entre $5Mhz$ et $100MHz$.

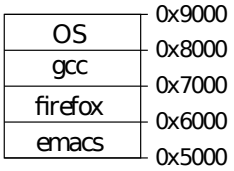
La qualité de l'horloge n'est pas toujours très bonne, et il vaut mieux synchroniser **régulièrement** l'heure et la date du système avec un serveur «NTP», «*Network Time Protocol*» afin d'éviter des écarts.

**Fonctionnement :**

- au moment de l'initialisation de l'horloge, une valeur est copiée dans le compteur ;
- chaque impulsion du crystal décrémente le compteur ;
- lorsque le compteur tombe à zéro, une **interruption** se produit.
  Deux modes de fonctionnement :
  *non répétitif* : lorsque l'interruption se produit, l'horloge reste arrêtée.
  *à répétition* : lors de chaque interruption, la valeur du compteur est automatiquement rechargé : le processus se répète indéfiniment.
  Ces interruptions périodiques sont appelées les «**tops d'horloge**».

**Comment partager la mémoire entre différents processus ?**

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




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

**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` ?

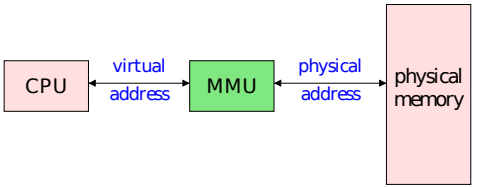
**Contraintes**

- ⋆ **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 resources** :
  - ◇ un programmeur développe un programme avec l'idée que l'ordinateur dispose de suffisament 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 :

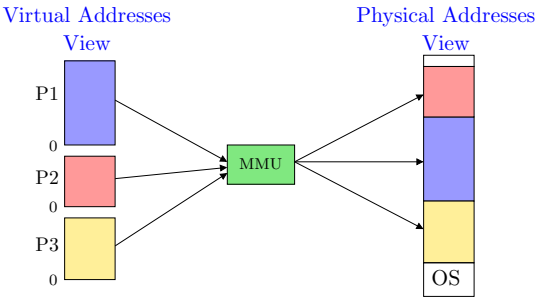
    «MMU», «Memory Management Unit» :



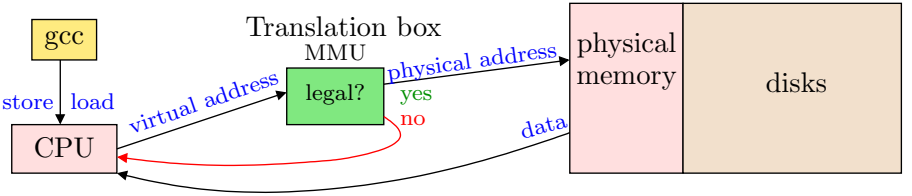

  - ▷ paramétrable uniquement avec des instructions privilégiées du «mode noyau» ;
  - ▷ traduit des adresses virtuelles vers des adresses physiques ;
  - ▷ la MMU fait partie du processeur.

- 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 mémorie de l'OS.
  - ◇ a sa mémoire qui commence à l'adresse zéro ;

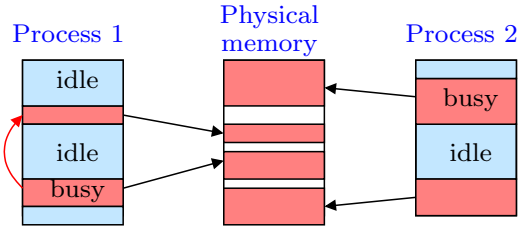


- protéger la mémoire d'un processus
  - **empêcher** qu'un processus chamboule la mémoire d'un autre processus




*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 ;

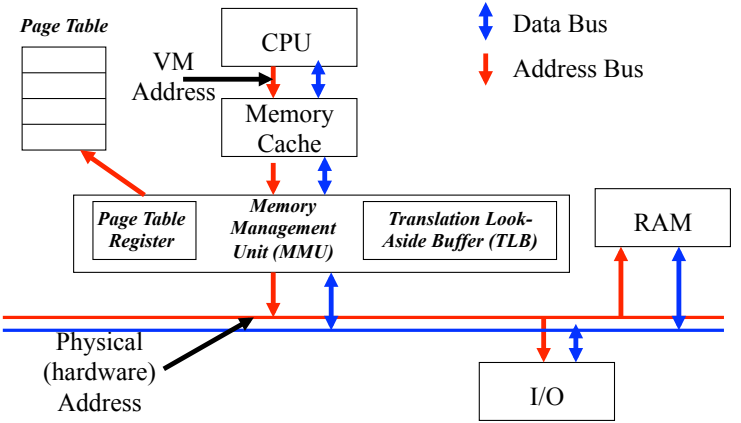
  ⟹ Utiliser la **segmentation**
- comment permettre aux processus d'augmenter la mémoire allouée ?

  ⟹ 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 ?

  ⟹ Paramètrer la MMU avec une **table de traduction d'adresse** associée à chaque processus.

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

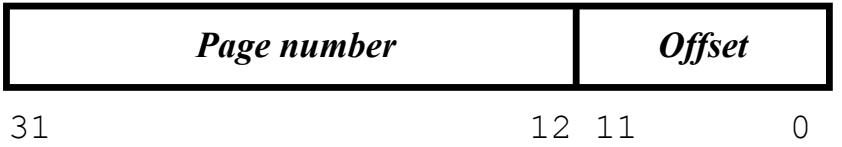
- le concept de page introduit une «indirection» supplémentaire pour l'accès à la mémoire :
  - le processus utilise une adresse virtuelle pour désigner un emplacement mémoire ;
  - pour accéder à cet emplacement mémoire, il faut connaître son adresse physique ;
  - pour connaître cette adresse physique, 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 ;



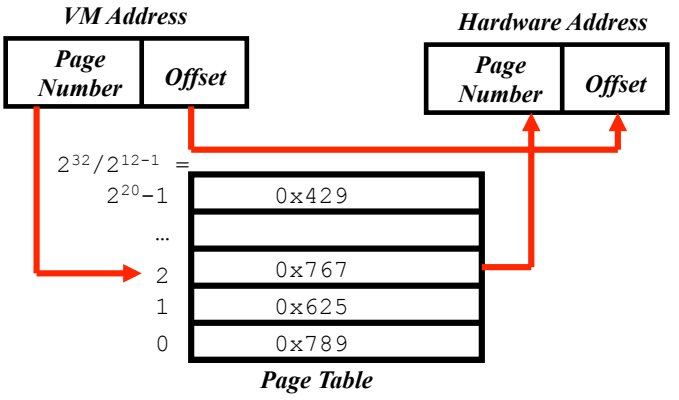

- la MMU possède un registre référençant la table courante utilisée pour réaliser la traduction virtuelle ⟶ physique ;
- chaque processus possède sa propre table (il utilise des adresses physiques différentes des autres processus) ;
- le registre, «Page Table Register», est modifié à chaque changement 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.

# Le paging : comment se passe la traduction d'adresse ?

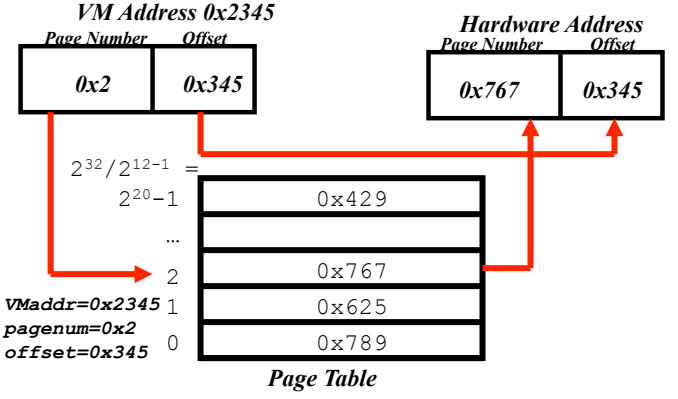
- une adresse virtuelle est décomposée en deux parties :
  - le numéro de page sur 20 bits ;
  - le décalage ou «*offset*» sur 12 bits, soit une taille de page de 4096 octets ;




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




Ce qui donne :

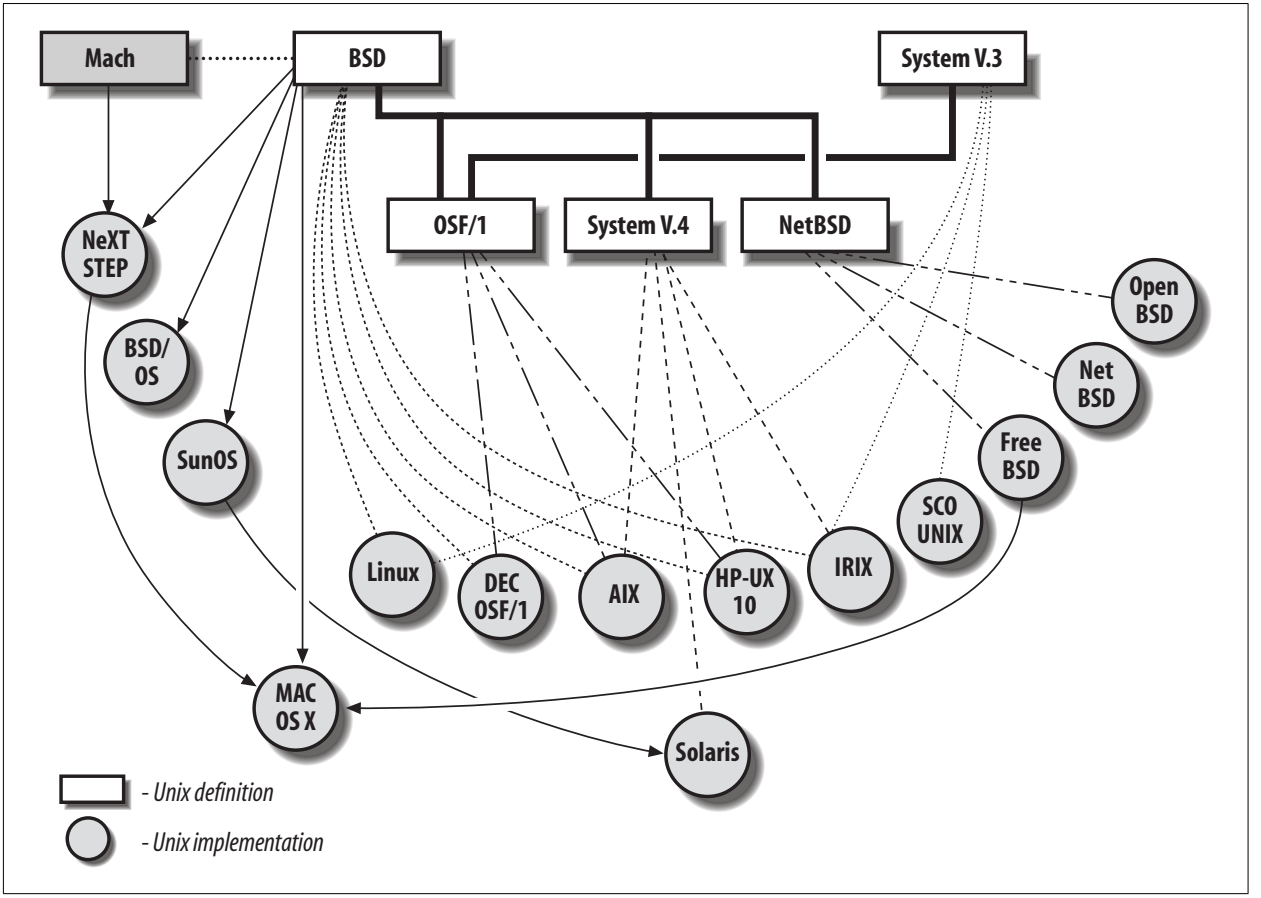


Plan de la partie

- 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 : les différentes variantes





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 :

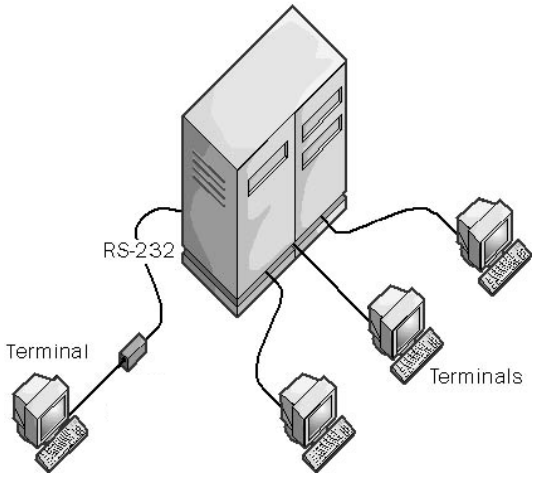
- un mode **noyau** (ou système ou superviseur)
  *le processeur peut exécuter toutes les instructions disponibles du système.*
- un mode **utilisateur**.
  *certaines instructions lui sont interdites, pour des raisons de sécurité du système*.

**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**.

**Un ordinateur central avec des accès partagés**



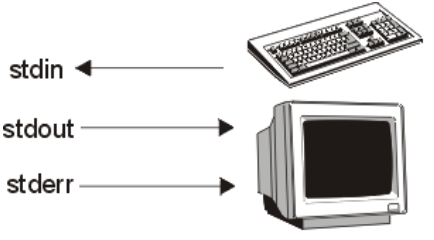

L'ordinateur est utilisé au travers de terminaux :

– un terminal correspond à un écran et un clavier :

– une liaison série, «RS-232», permet d'échanger du texte dans les deux sens (terminal ⟺ 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**

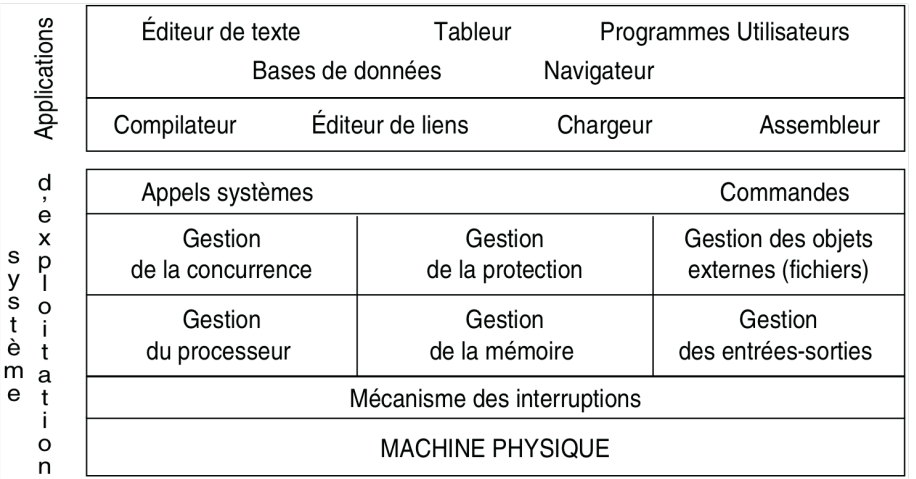
La **programmation concurrente** est liée à la naissance des SE et à l'invention des contrôleurs de périphériques (device controllers) :

- 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.




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 (pagination/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).

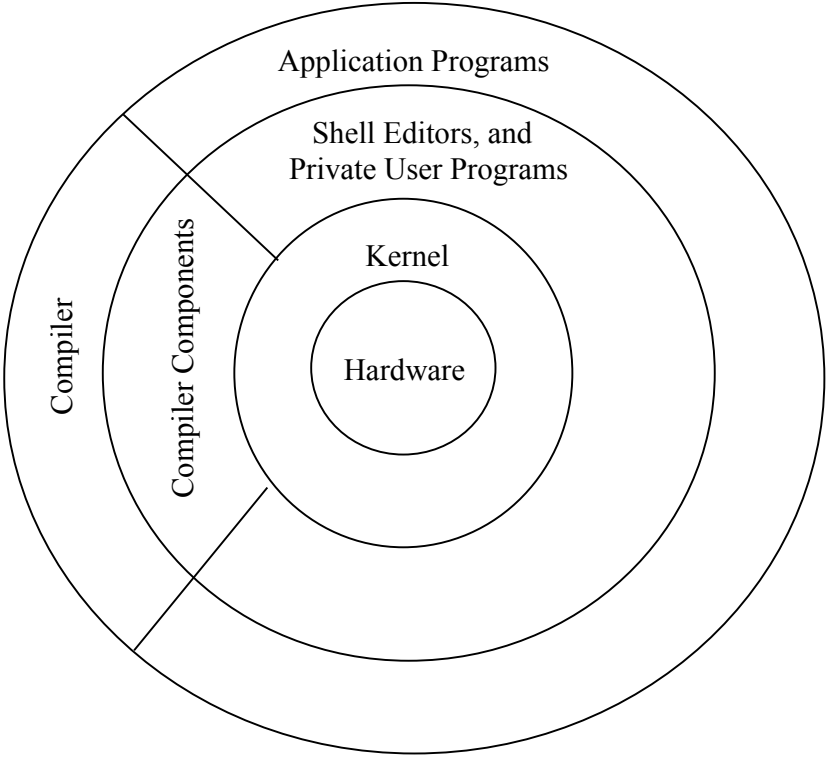
  *C'est le pilote d'E/S (driver) qui assure cette liaison.*
- **Gestion de la concurrence** : plusieurs programmes coexistent en mémoire centrale :
  - assurer les besoins de communication pour échanger des données ;
  - synchroniser l'accès aux données partagées afin de maintenir la cohérence de ces 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.



# La virtualisation de la machine vue par Unix

- 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»*.



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**

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.

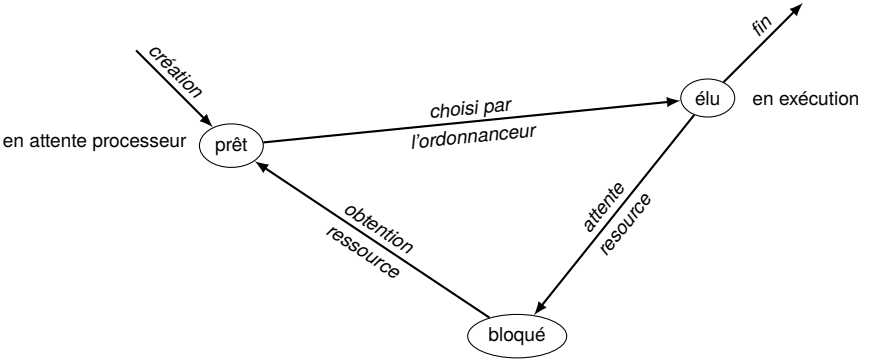
Un processus est :

- 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 :

- lorsque le processus obtient le processeur et s'exécute, il est dans l'état **élu**.
- 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** :

- 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) ;
- 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.

- lorsque le processus est passé dans l'état bloqué, le processeur a été **alloué** à un autre processus.
- 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 crée dans l'état **prêt** .
- ☐ Un processus se termine toujours à partir de **l'état élu** (sauf anomalie).



Un processus peut en créer un autre :

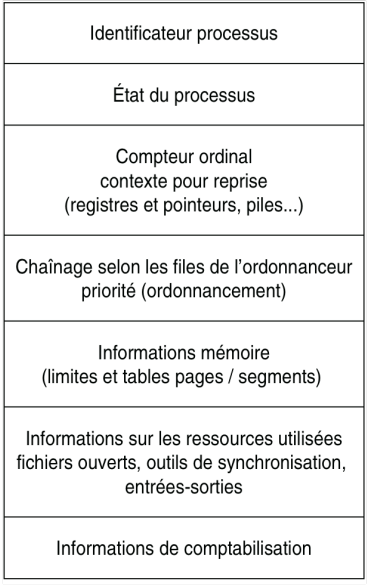
- ▷ le premier est appelé **père**
- ▷ le second **fils**.
- ▷ le fils peut à son tour créer d'autres processus : il devient le père de ses processus...
  Construction d'une arborescence de processus ou la racine correspond au processus ancêtre.

**PCB ou «*Process Control Bloc*»**

Le système d'exploitation crée un PCB, une structure de description du processus associée au nouveau programme exécutable qui contient les informations suivantes :

- ⋆ identificateur unique du processus (PID de type entier)
- ⋆ état courant du processus (élu, prêt, bloqué)
- ⋆ contexte processeur du processus : la valeur du CO, la valeur des autres registres du processeur
- ⋆ contexte mémoire : ce sont des informations mémoire qui permettent de trouver le code et les données du processus en mémoire centrale
- ⋆ informations diverses de comptabilisation pour les statistiques sur les performances système
- ⋆ informations liées à l'ordonnancement du processus.

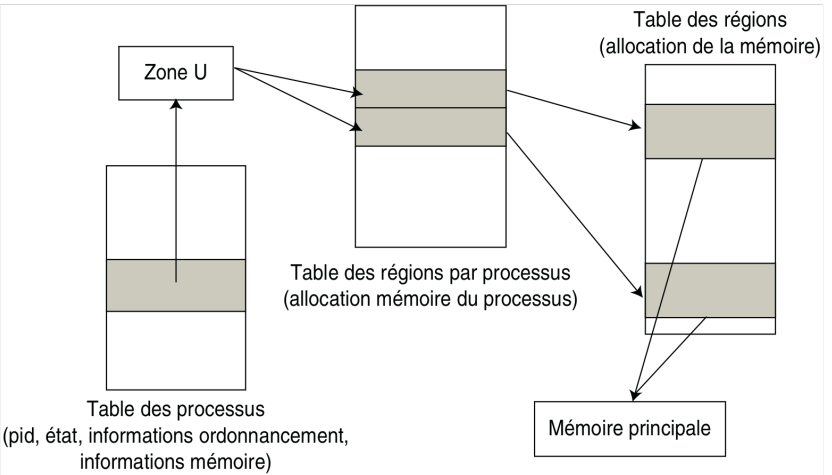
*Le PCB permet la sauvegarde et la restauration du contexte mémoire et du contexte processeur lors des opérations de commutations de contexte (commutation d'un processus à un autre pour le processeur).*

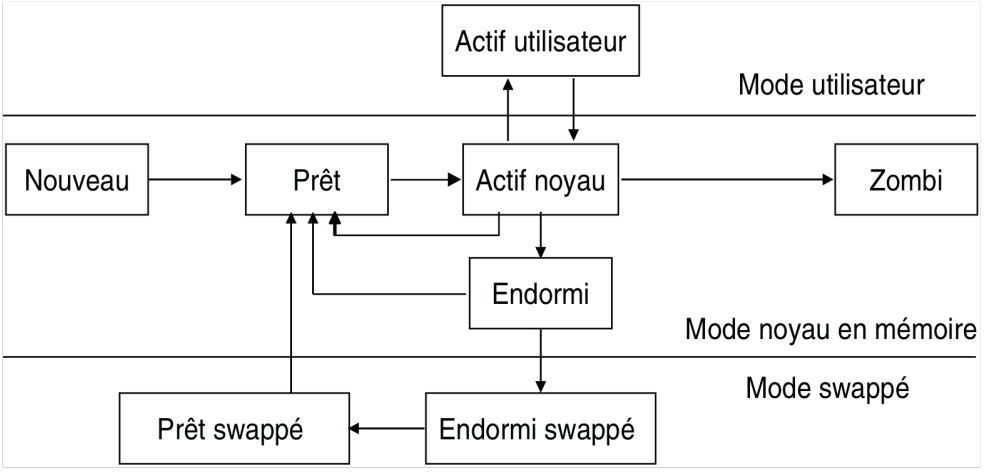


Le bloc de contrôle du processus est divisé en deux parties, chaque processus dispose :

– d'une entrée dans une table générale du système, la table des processus.
*cette entrée contient les informations sur le processus qui sont toujours utiles au système quel que soit l'état du processus : l'identificateur du processus (pid), l'état du processus, les informations d'ordonnancement, les informations mémoire, c'est-à-dire l'adresse des régions mémoire allouées au processus.*

– d'une seconde structure : la Zone U. *Cette Zone U contient d'autres informations concernant le processus, mais ce sont des informations qui peuvent être temporairement «swappées» sur le disque*






Cycle entre trois modes au cours de l'exécution d'un processus :

- le mode utilisateur qui est le mode normal d'exécution,
- le mode noyau en mémoire qui est le mode dans lequel se trouve un processus prêt ou bloqué (endormi)
- le mode swappé qui est le mode dans lequel se trouve un processus bloqué (endormi) déchargé de la mémoire centrale dans la zone de swap sur le disque dur.

*Le système Unix décharge de la mémoire centrale les processus endormis depuis trop longtemps (ils sont alors dans l'état Endormi swappé).*

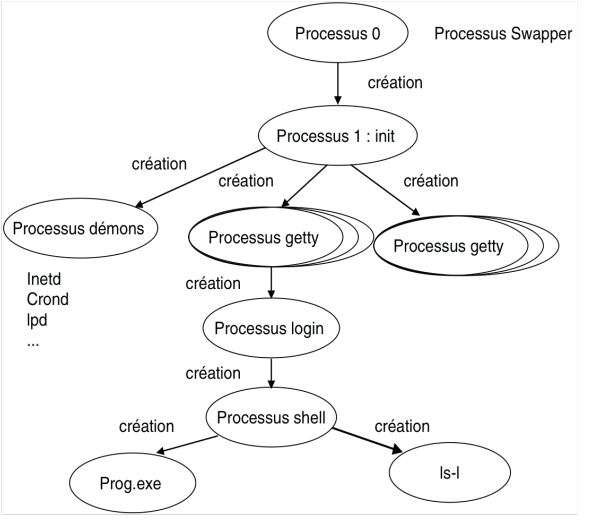
*Ces processus réintègrent la mémoire centrale lorsqu'ils redeviennent prêts (transition de l'état prêt swappé vers l'état prêt).*

Un processus qui se termine passe dans un état dit zombi. Il y reste tant que son PCB n'est pas entièrement détruit par le système.

Le système Unix est entièrement construit à partir de la notion de processus :

- au démarrage du système, un premier processus est crée : le processus 0.
- le processus 0 crée à son tour un autre processus, le processus 1 ou init.
- le processus `init` lit le fichier `/etc/inittab` et crée chacun des deux types de processus décrits dans ce fichier :

  Les processus démons (dont le nom est suffixé par un "d") qui sont des processus système responsables d'une fonction (inetd surveille le réseau, lpd gère les imprimantes, crond gère un échéancier, *etc.*)
- les processus getty qui surveillent les terminaux




***Attention : les dernières versions de Linux utilisent systemd.***

Lorsqu'un utilisateur vient se loguer sur un terminal :

▷ un processus login est crée qui lit le nom de l'utilisateur et son mot de passe.

▷ Ce processus login vérifie la validité de ces informations en utilisant le fichier `/etc/passwd` décrivant les comptes.

▷ Si les informations sont valides, le processus login crée un processus shell, c-à-d un interpréteur de commandes. *Cet interpréteur de commandes exécute alors les commandes et programmes de l'utilisateur en créant à chaque fois un nouveau processus.*

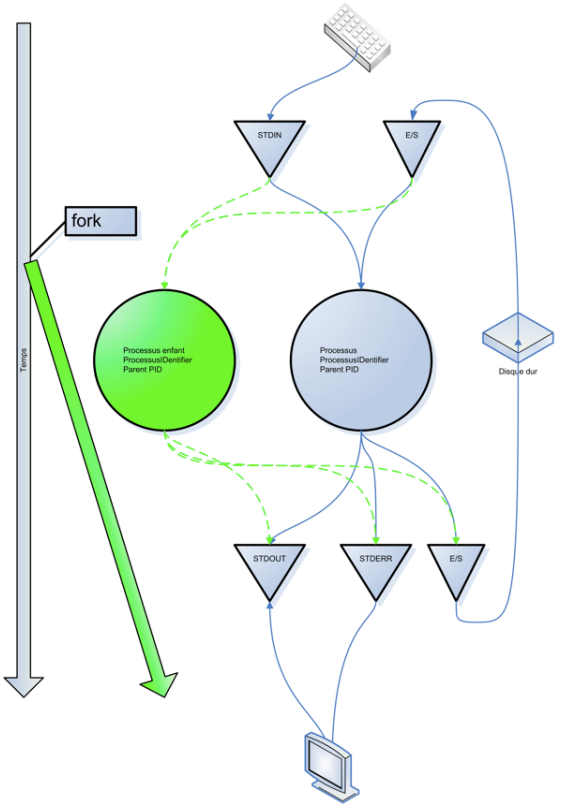
Les processus sont créés au travers d'appels sytème :

⋆ la primitive `fork()` crée un processus fils **clone du processus père** appelant.

⋆ la primitive `exec()` termine la création du processus en fournissant le code qui est propre à ce nouveau processus.

**Création**

a. «clonage» du processus père.
   *Le premier processus père : le Shell.*
b. le nouveau processus partage ses E/S avec son père :
   - ◇ stdin, stdout, stderr ;
     *la lecture de l'entrée se fait depuis le shell, les sorties se font dans le shell*
   - ◇ tous les fichiers ouverts
     *les fichiers disques, les tubes de communication*
   - ◇
c. Possède le même code que son père :
   - ◇ différence de valeur de retour du fork :
     - ⋆ 0 pour le processus fils ;
     - ⋆ pid du fils pour le processus père ;
   - ◇ recouvrement du code par un autre avec «`exec()`



# Destruction du processus

Le processus fils peut se terminer de différentes facons :

- Il se termine normalement après l'exécution de la dernière instruction du code qui lui est associé.
- Il peut exécuter une instruction d'auto-destruction. Exemple : primitive `exit(...)`.
- Un processus peut-être détruit par un autre processus. Exemple : primitive `kill(...)`.

La **relation hiérarchique** entre un processus et ses descendants est utilisée essentiellement pour contrôler la destruction des processus.

xterm

```
pef@samus:~$ pstree -A
init-+-acpid
     |-avahi-daemon---avahi-daemon
     |-cgmanager
     |-cron
     |-dhclient
     |-2*[dnsmasq]
     |-freeradius---5*[{freeradius}]
     |-6*[getty]
     |-irqbalance
     |-libvirtd---10*[{libvirtd}]
     |-logger
     |-mysqld_safe---mysqld---19*[{mysqld}]
     |-rsyslogd---3*[{rsyslogd}]
     |-sendmail-mta
     |-sshd---sshd---sshd---bash---pstree
     |-systemd-logind
```

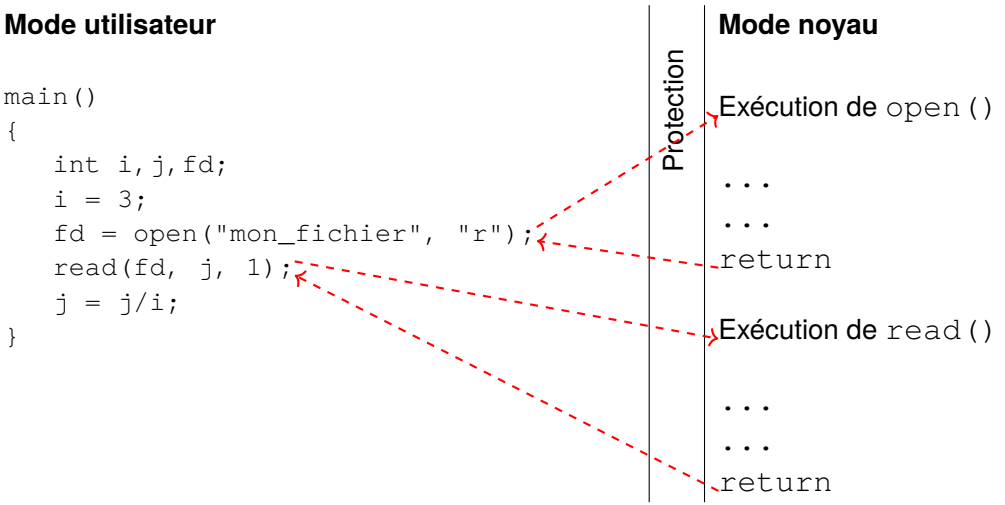
La destruction d'un processus entraîne :

- la libération des ressources qui lui avait été affectée.
- son PCB est effacé : il disparaît de la table et des files d'attente du système (il peut également rester en «processus zombi» jusqu'à ce que le processus père consulte les statistiques d'exécution lorsque le processus a été lancé avec un *wait*).

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 appelables 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**




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");
```

Protection

**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 chaine a afficher
        mov edx,helloSize       ; Taille de la chaine
        int 80h                 ; Execution 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");
```

Protection

**Mode noyau**

Exécution de `open()`

```
...
...
return
```

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

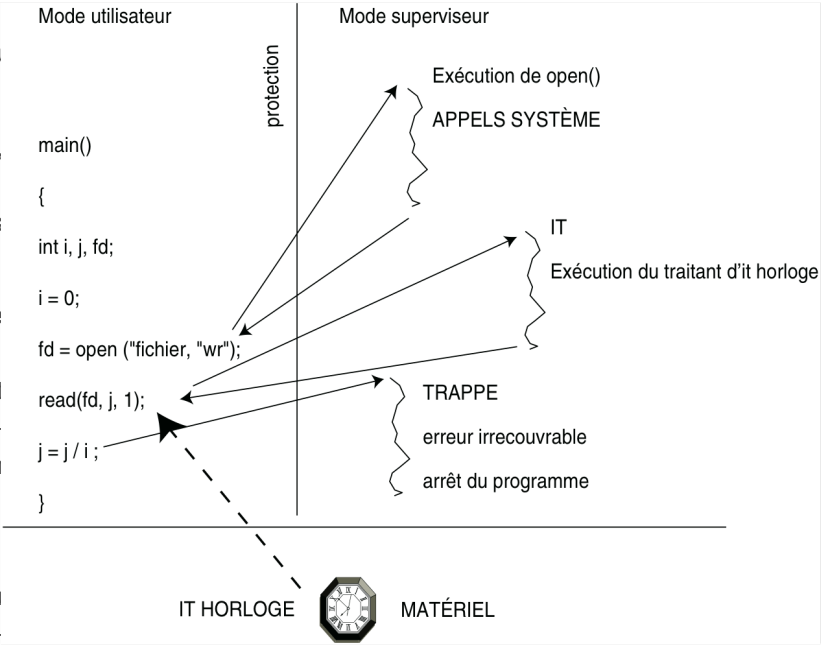
À la fin de l'**appel système** :

▷ passage du **mode noyau** au **mode utilisateur** ;

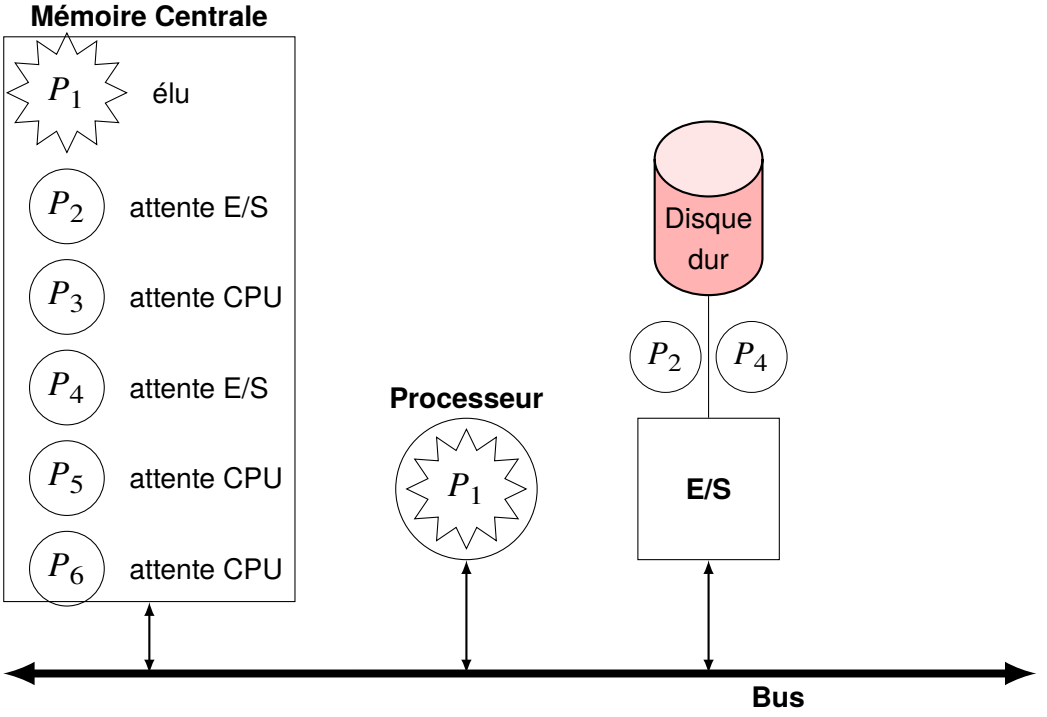
⟹ **commutation de contexte** : restauration du contexte utilisateur.

**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**.



# Préemption : commutation par l'horloge

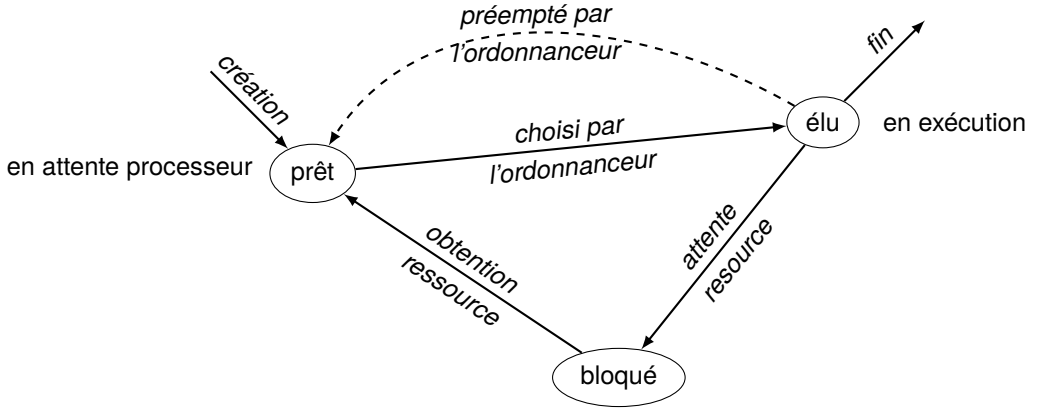


Dans ce schéma, plusieurs processus sont présents en mémoire centrale.

- P1 est élu et s'exécute sur le processeur.
- P2 et P4 sont dans l'état bloqué car ils attentent tous les deux une fin d'entrée- sortie avec le disque géréé par l'unité d'échange (UE).
- P3, P5 et P6 quant à eux sont dans l'état prêt : ils pourraient s'exécuter (ils ont à leur disposition toutes les ressources nécessaires) mais ils ne le peuvent pas car le processeur est occupé par P1.

Lorsque P1 quittera le processeur parce qu'il a terminé son exécution, les trois processus P3, P5 et P6 auront tous les trois le droit d'obtenir le processeur.
Mais le processeur ne peut être alloué qu'à un seul processus à la fois : il faudra donc choisir entre P3, P5 et P6.

C'est le rôle de **l'ordonnancement** qui **élira** un des trois processus.




**L'automate** montre les transitions pouvant exister entre l'**état prêt** (état d'attente du processeur) et l'**état élu** (état d'occupation du processeur).

Passage de :

- ▷ l'**état prêt** vers l'**état élu** constitue l'opération **d'élection** : allocation du processeur à un des processus prêts.
- ▷ l'**état élu** vers l'**état prêt** : **réquisition** du processeur, c-à-d que le processeur est retiré au processus élu alors que celui-ci dispose de toutes les ressources nécessaires à la poursuite de son exécution.

Cette réquisition porte le nom de **préemption**.

**Ordonnancement préemptif ou non préemptif**

- □ **non préemptif** ou **coopératif** : la transition de l'état élu vers l'état prêt est **interdite** : un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque.
- □ **préemptif** : la transition de l'état élu vers l'état prêt est **autorisée** : un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

### Obtention du PID, «*Processus IDentifier*»

```
#include <unistd.h>

pid_t getpid(void) // retourne le pid du processus appelant
pid_t getppid(void) // retourne le pid du père du processus appelant
```

### Obtenir des informations sur les processus

La commande `ps` : retourne la liste des processus avec leur caractéristiques (pid, ppid, état, terminal, durée d'exécution, commande associée...)

xterm

```
pef@cube:~$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1000  3889  3880  20   0  22840  4988 poll_s Ss+  pts/0      0:00 bash
0  1000 25566 25565  20   0  23120  5888 poll_s Ss+  pts/2      0:01 -bash
0  1000 29150 25566  20   0  54772  9300 signal T    pts/2      0:06 vi build_architecture
0  1000 30199 30198  20   0  22772  5252 wait   Ss   pts/1      0:00 -bash
0  1000 30772 30770  20   0  22900  5672 poll_s Ss+  pts/3      0:00 -bash
0  1000 32705 30199  20   0  31344  1580 -      R+   pts/1      0:00 ps l
```

### Envoyer un signal à un processus

xterm

```
pef@cube:~$ kill -9 29150
```

*Le signal nº9 correspond à `SIGKILL` : il termine de immédiatement le processus donné par son PID.*

xterm

```
pef@cube:~$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1000  3889  3880  20   0  22840  4988 poll_s Ss+  pts/0      0:00 bash
0  1000 25566 25565  20   0  23120  5888 poll_s Ss+  pts/2      0:01 -bash
0  1000 30199 30198  20   0  22776  5256 wait   Ss   pts/1      0:00 -bash
0  1000 30772 30770  20   0  22900  5672 poll_s Ss+  pts/3      0:00 -bash
0  1000 32733 30199  20   0  31344  1604 -      R+   pts/1      0:00 ps l
```



**Création de processus**

```
#include <unistd.h>

pid_t fork(void)
```

La primitive `fork()` permet la **création dynamique** d'un nouveau processus qui s'exécute de manière concurrente avec le processus qui l'a créé :

- Tout processus Unix/Linux hormis le processus 0 est crée à l 'aide de cette primitive.
- Le processus créateur (le père) crée un processus fils par un appel à la primitive `fork()` qui est une **copie exacte** de lui-même (code et données)

**Déroulement de l'appel système `fork`**

**MODE UTILISATEUR**

PROCESSUS PID 10279

```
1 int main (void)
2 {
3   pid_t ret;
4   int i, j;

5   for(i=0; i<8; i++)
6     i = i + j;

7   ret = fork();
8 }
```

**MODE NOYAU**

Exécution de l'appel système fork :

Si les ressources noyau sont disponibles :

- allouer une entrée de la table des processus au nouveau processus
- allouer un **pid unique** au nouveau processus
- dupliquer le contexte du processus parent (code, données, pile)
- retourner :
  - ◇ 0 au processus fils
  - ◇ le pid du processus crée à son père

**MODE UTILISATEUR**

PROCESSUS PID 10279

```
1 int main (void)
2 {
3   pid_t ret;
4   int i, j;

5   for(i=0; i<8; i++)
6     i = i + j;

7   ret = fork();
8 }
```

PROCESSUS PID 10280

```
1 int main (void)
2 {
3   pid_t ret;
4   int i, j;

5   for(i=0; i<8; i++)
6     i = i + j;

7   ret = fork();
8 }
```

10280

0

**MODE NOYAU**

Exécution de l'appel système fork :

Si les ressources noyau sont disponibles :

- allouer...
- allouer...
- dupliquer...
- retourner :
  - ◇ 0 au processus fils
  - ◇ le pid du processus crée à son père

Chaque processus père et fils reprend son exécution après le `fork()` :

- le code et les données étant strictement identiques, il est nécessaire de disposer d 'un mécanisme pour différencier le comportement des deux processus après le `fork()` ;
- on utilise pour cela le code retour du `fork()` qui est différent chez le fils (toujours 0) et le père (pid du fils crée).

PROCESSUS PID 10279

```
1  int main (void)
2  {
3    pid_t ret;
4    int i, j;

5    for(i=0; i<8; i++)
6      i = i + j;

7    ret = fork();
8    if (ret == 0)
9      printf("Je suis le fils");
10   else
11   {
12     printf("Je suis le pere");
13     printf("pid de mon fils %d",ret);
14   }
15 }
```

| Pid du fils | 10280 |
|---|---|
| getpid | 10279 |
| getppid | pid du shell |

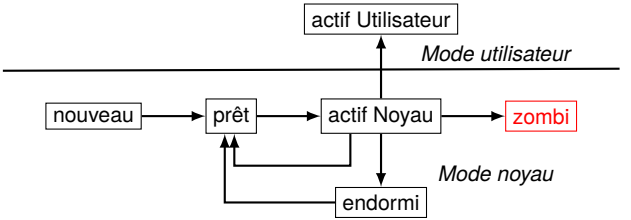
PROCESSUS PID 10280

```
1  int main (void)
2  {
3    pid_t ret;
4    int i, j;

5    for(i=0; i<8; i++)
6      i = i + j;

7    ret = fork();
8    if (ret == 0)
9      printf("Je suis le fils");
10   else
11   {
12     printf("Je suis le pere");
13     printf("pid de mon fils %d",ret);
14   }
15 }
```

| getpid | 10280 |
|---|---|
| getppid | 10279 |

```
1 #include <stdlib.h>
2
3 void exit (int valeur);
4 pid_t wait (int *status);
```

un appel à la primitive `exit()` provoque la terminaison du processus effectuant l'appel avec un code retour valeur. *Par défaut, en C, la sortie du dernier bloc d'instructions tient lieu d'exit.*




▷ un processus qui se termine passe dans l'état **Zombi** tant que son père n'a pas pris en compte sa terminaison ;

▷ le processus père «récupère» la terminaison de ses fils par un appel à la primitive `wait()`

PROCESSUS PID 10279

```
int main (void)
{
   pid_t ret;
   int i, j;

   for(i=0; i<8; i++)
      i = i + j;

   ret = fork();
   if (ret == 0)
   {
      printf("Je suis le fils");
      exit();
   }
   else
   {
      printf("Je suis le pere");
      printf("pid de mon fils %d",ret);
      wait();
   }
}
```

PROCESSUS PID 10280

```
int main (void)
{
   pid_t ret;
   int i, j;

   for(i=0; i<8; i++)
      i = i + j;

   ret = fork();
   if (ret == 0)
   {
      printf("Je suis le fils");
      exit();
   }
   else
   {
      printf("Je suis le pere");
      printf("pid de mon fils %d",ret);
      wait();
   }
}
```

synchronisation