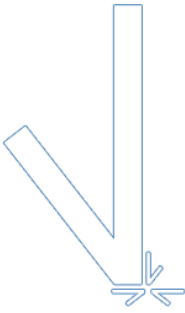


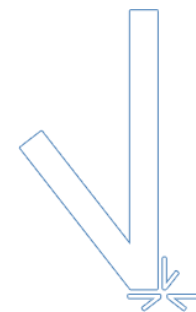
Systemes d'Exploitation



Objectif :

Comprendre comment un système
d'exploitation fonctionne et
comment l'utiliser

Systemes d'Exploitation



Noyau et Pilotes

(cours précédent : Systemes d'Exploitation – Sommaire et Introduction)

I. Systèmes d'Exploitation

I.1. Sommaire



- I. **Système d'Exploitation**
 - 1. Sommaire
 - 2. Introduction
 - 3. **Noyau et Pilotes**
 - a. Noyau
 - b. Pilotes
 - 4. Utilisateurs et Sessions
 - 5. Système de Fichier
 - 6. Permissions et Droits
 - 7. Shell et Utilitaires
 - 8. Gestion de la Mémoire
 - 9. Programmes et Processus
 - 10. Variables d'Environnement
 - 11. Scripts Shell
 - 12. Gestion des Paquets

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Noyau

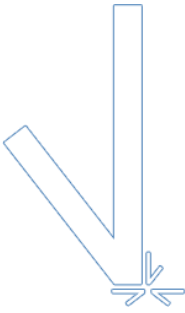


Noyau

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Noyau - fondamentaux



🔒 Le **noyau** est **invisible** pour l'utilisateur.

🔗 Le noyau **permet** aux autres programmes d'être **exécutés**.

🖨 Le noyau gère les **événements produits au niveau de la couche matérielle**. Ces événements sont appelés **Interruptions**.

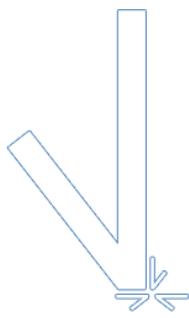
📄 Le noyau gère les **événements produits au niveau de la couche logicielle**. Ces événements sont appelés **Appels Système**.


🏠 Le noyau gère les **accès aux ressources matérielles** ou **logicielles**.

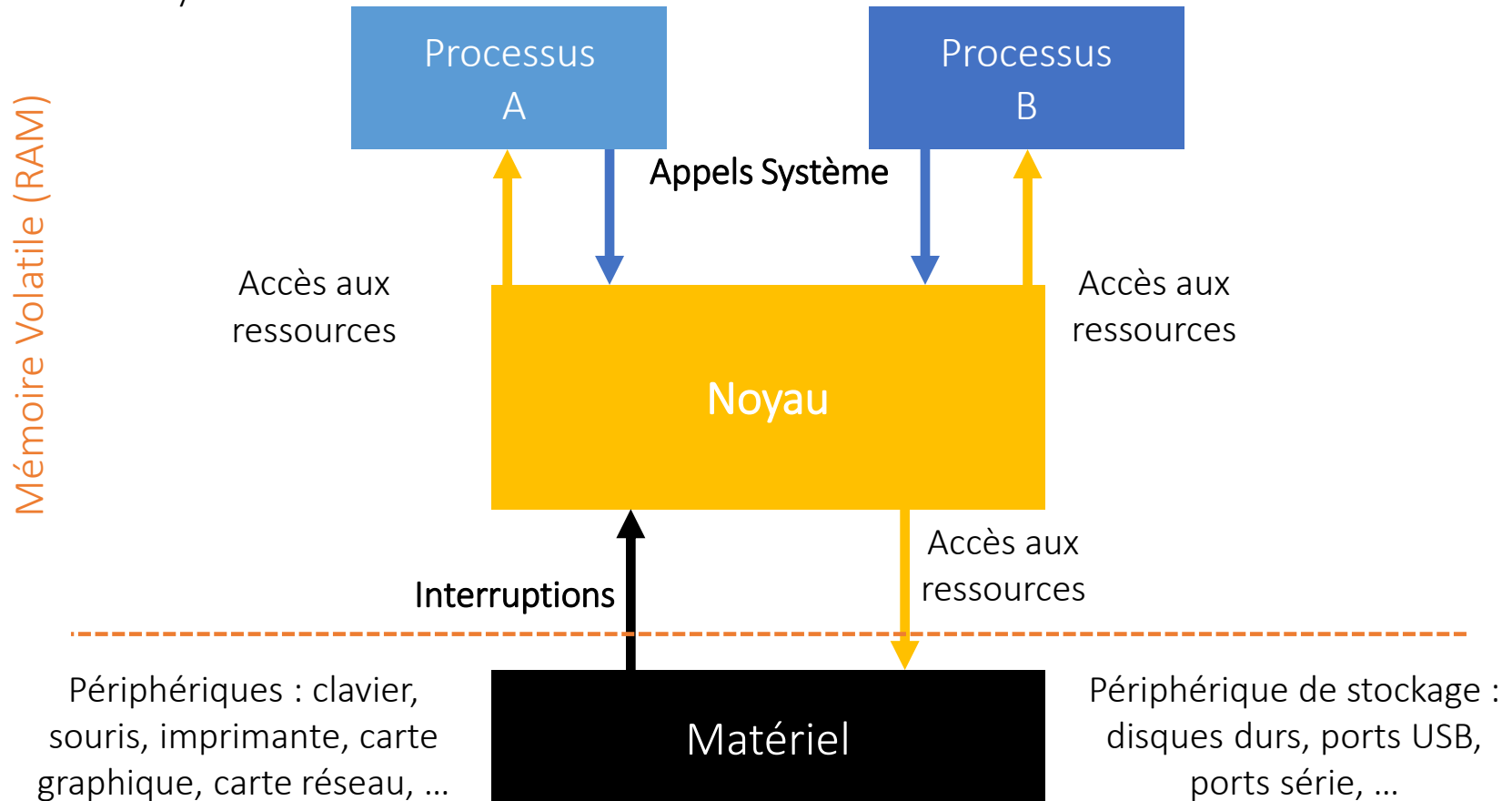
I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Noyau - fondamentaux



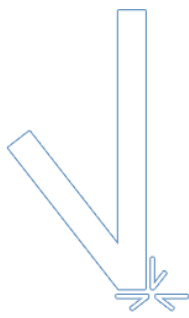
 Rôle du Noyau :



I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Noyau – langages de programmation



IOIO
IOIO Le **noyau** est chargé en mémoire sous la forme de **code machine**. Il s'agit d'un code de bas niveau qui interagit avec le matériel et qui est donc spécifique au matériel.

IOIO
IOIO Les codes machine x86 (pour les architectures matérielles 32 bits) et le x64 (pour les architectures matérielles 64 bits) sont utilisés sur les matériels de type PC compatible.

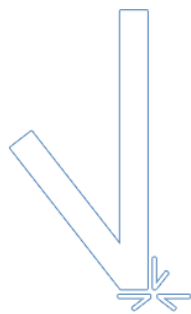
```
00000000 0000 0001 0001 1010 0010 0001 0004 0128
00000010 0000 0016 0000 0028 0000 0010 0000 0020
00000020 0000 0001 0004 0000 0000 0000 0000 0000
00000030 0000 0000 0000 0010 0000 0000 0000 0204
00000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
00000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
00000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
00000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
00000080 8888 8888 8888 8888 288e be88 8888 8888
00000090 3b83 5788 8888 8888 7667 778e 8828 8888
000000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
000000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
000000c0 8a18 880c e841 c988 b328 6871 688e 958b
000000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eac
000000e0 3d86 dcbb 5cbb 8888 8888 8888 8888 8888
000000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Code machine brut

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Noyau – langages de programmation



❏ Pour créer du code machine, un programme doit être écrit en **langage ASseMbleur** (ASM) et transformé en code machine en utilisant un programme **Assembleur**.

❏ Quand on écrit un programme qui interagit avec le noyau, **les appels système** au noyau doivent être programmés en **langage ASseMbleur** (ASM).

❏ Les langages Assembleurs sont **verbeux**, très **orientés bas niveau** (il nécessitent de maîtriser très précisément les spécificités du matériel sous-jacent), et sont **spécifiques au matériel**.

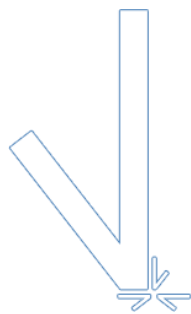
100 101 102 103 104 105 106 107 00000030 B9FFFFFFF 108 109 110 00000035 41 111 00000036 003C0000 112 113 114 0000003A 75F9 115 116 117 118 119 120 121 122 0000003C C3	<pre>----- ; zstr_count: ; Counts a zero-terminated ASCII string to determine its size ; in: eax = start address of the zero terminated string ; out: ecx = count = the length of the string zstr_count: ; Entry point mov ecx, -1 ; Init the loop counter, pre-decrement ; to compensate for the increment .loop: inc ecx ; Add 1 to the loop counter cmp byte [eax + ecx], 0 ; Compare the value at the string's ; [starting memory address Plus the ; loop offset], to zero jne .loop ; If the memory value is not zero, ; then jump to the label called '.loop', ; otherwise continue to the next line .done: ; We don't do a final increment, ; because even though the count is base 1, ; we do not include the zero terminator in the ; string's length ret ; Return to the calling program</pre>
--	---

Code machine et langage Assembleur côte à côte

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel – librairies standard



- Quand un programme est écrit en langage **C**, il doit être **compilé**. La **Compilation** est un processus à travers lequel un programme C est **transformé en assembleur**, puis **transformé en code machine**.
- Dans un **programme C**, le programmeur utilise des **librairies standard** (écrite en langage C) fournies avec le système d'exploitation pour produire des **appels système**.
- Les librairies standard exposent une API C au programmeur (des variables et des fonctions en C). Quand une librairie standard est compilée avec le programme qui l'utilise, elle produit le code machine pour des appels système tels qu'attendu par le noyau.
- Ces librairies sont **fournies** avec le système d'exploitation. Elles sont prévues pour être importées et utilisées à l'**intérieur** de programmes écrit en C.

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel – le standard POSIX



- Les Systèmes d'Exploitation (dérivés d'Unix) fournissent des bibliothèques standard qui respectent le **standard POSIX** (Portable Operating System Interface).
- POSIX est un standard défini par l'IEEE (Institute of Electrical and Electronics Engineers) sous la référence [IEEE 1003](#).
- Le standard POSIX décrit aussi d'autres **composants du système d'exploitation**. Par exemple, l'interface en ligne de commande UNIX (le *shell*) fait partie du standard POSIX.

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel – le standard POSIX

- Exemple :

1. Programme en C faisant appel à la librairie POSIX `stdio.h`

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n"); // appel système
    return 0;
}
```

2. Ce programme est transformé en code assembleur par le compilateur (par exemple avec gcc : `gcc -march=x86-64 -o output program.c`) (voir à droite)
3. Puis le compilateur transforme le code assembleur en code machine (voir à droite)
4. Enfin le code machine obtenu est lié avec le code machine provenant de la librairie POSIX pour former l'exécutable final

2. Compilation : Code Assembleur Intel 64

```
section .data
    message db "Hello, World!", 0      ; La chaîne de caractères,
terminée par un zéro

section .text
    global _start                      ; Point d'entrée du programme

extern printf                          ; Déclaration de la fonction
printf

_start:
    ; Préparer les arguments pour printf
    mov rdi, message                  ; Le premier argument à passer
    est l'adresse de la chaîne
    xor rax, rax                      ; printf nécessite que rax soit
mis à zéro (appel de fonction variadique)

    ; Appeler printf
    call printf

    ; Sortie du programme
    mov rax, 60                      ; L'appel système pour "exit"
    est 60
    xor rdi, rdi                      ; Code de sortie 0
    syscall                          ; Effectuer l'appel système
```

3. Assemblage : Code machine Intel 64 obtenu (« objet »)

```
48 bf 00 00 00 00 00 00 00 00 48 31 c0 e8
00 00 00 00 b8 3c 00 00 00 48 31 ff 0f 05
```

4. Linking : Ajout d'objet provenant de librairies (ici POSIX)

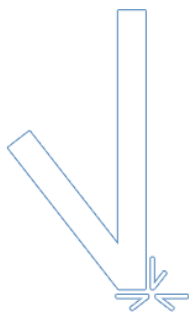
```
bf 00 20 40 00 48 31 c0 e8 f3 ff ff ff b8
3c 00 00 00 48 31 ff 0f 05
```

Ci-dessus le binaire exécutable pour un système Intel 64 / AMD64

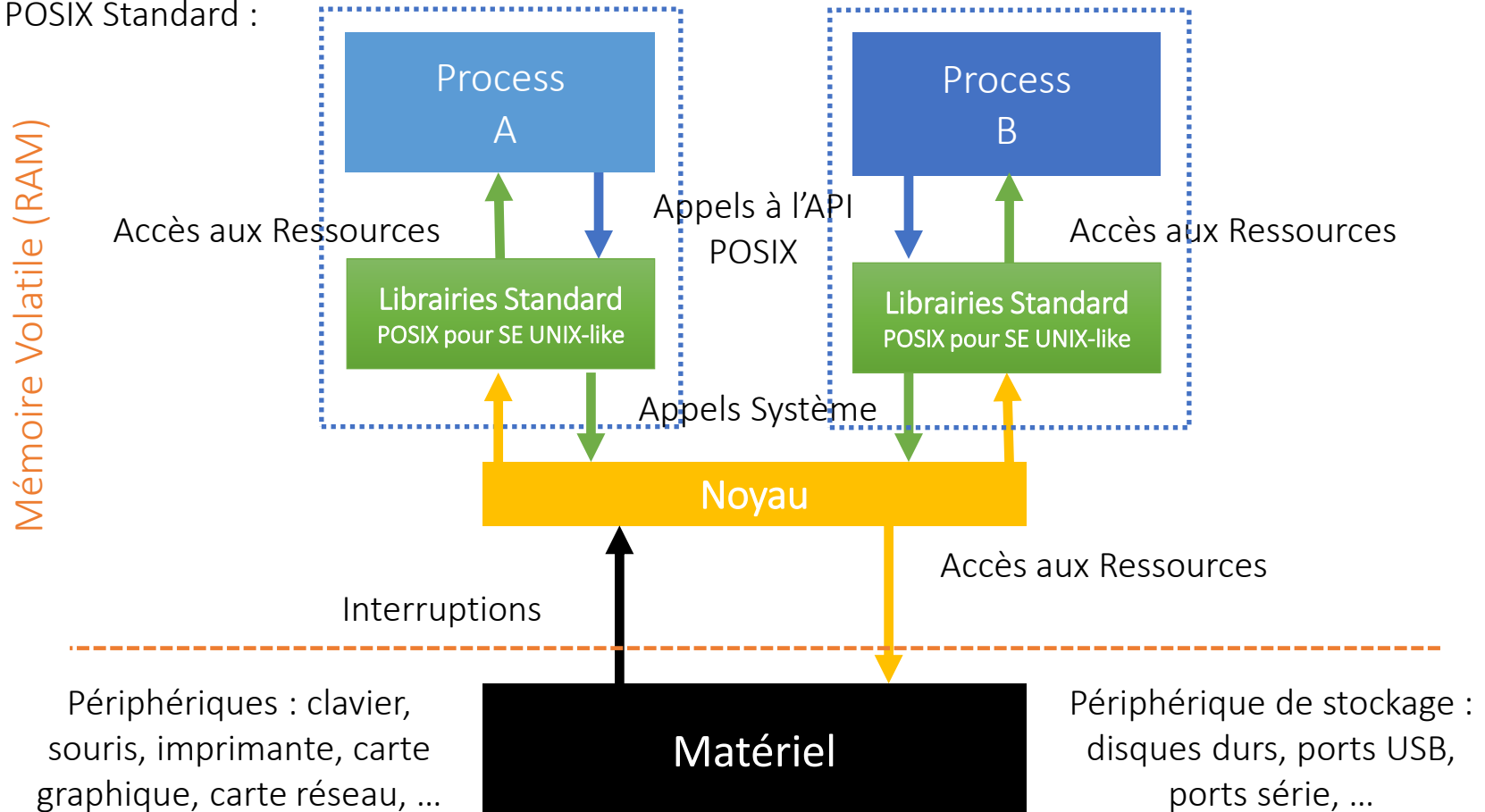
I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel – le standard POSIX



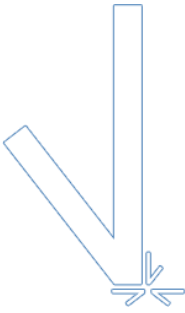
- POSIX Standard :



I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel – 2 catégories



- Il existe **2 catégories de noyau** pour les systèmes UNIX-like.
- 1^{ère} catégorie : [Microkernel](#) :
 - Gère la communication entre les programmes chargés en mémoire volatile (Inter Process Communication):

↔ **IPC : Inter Process Communication**

- Fournit une interface en lecture/écriture pour la mémoire volatile. Il s'agit de la Mémoire Virtuelle. Elle permet aux processus d'ignorer la localisation **physique** réelle des données qui sont stockées:

≡ **Virtual Memory**

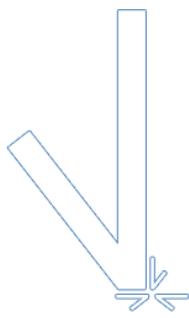
- Alloue du temps d'exécution au processus en mémoire:

🕒 **Scheduling**

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel – 2 catégories



- 2^{ème} catégorie : **Noyau Monolithique** :

- A les mêmes caractéristiques qu'un Microkernel :

⚖ **IPC : Inter Process Communication, Virtual Memory and Scheduling**

- Contient des pilotes de périphériques sous la forme de modules du noyau. Un module de noyau est un extrait de code de noyau qui gère les événements relatifs à un périphérique système ou des appels systèmes produits par des processus.

🔧 **Device Drivers**

- Alloue du temps CPU pour l'exécution de « stacks » (ensemble d'instructions spécifiques) au sein d'un processus :

🕒 **Dispatcher**

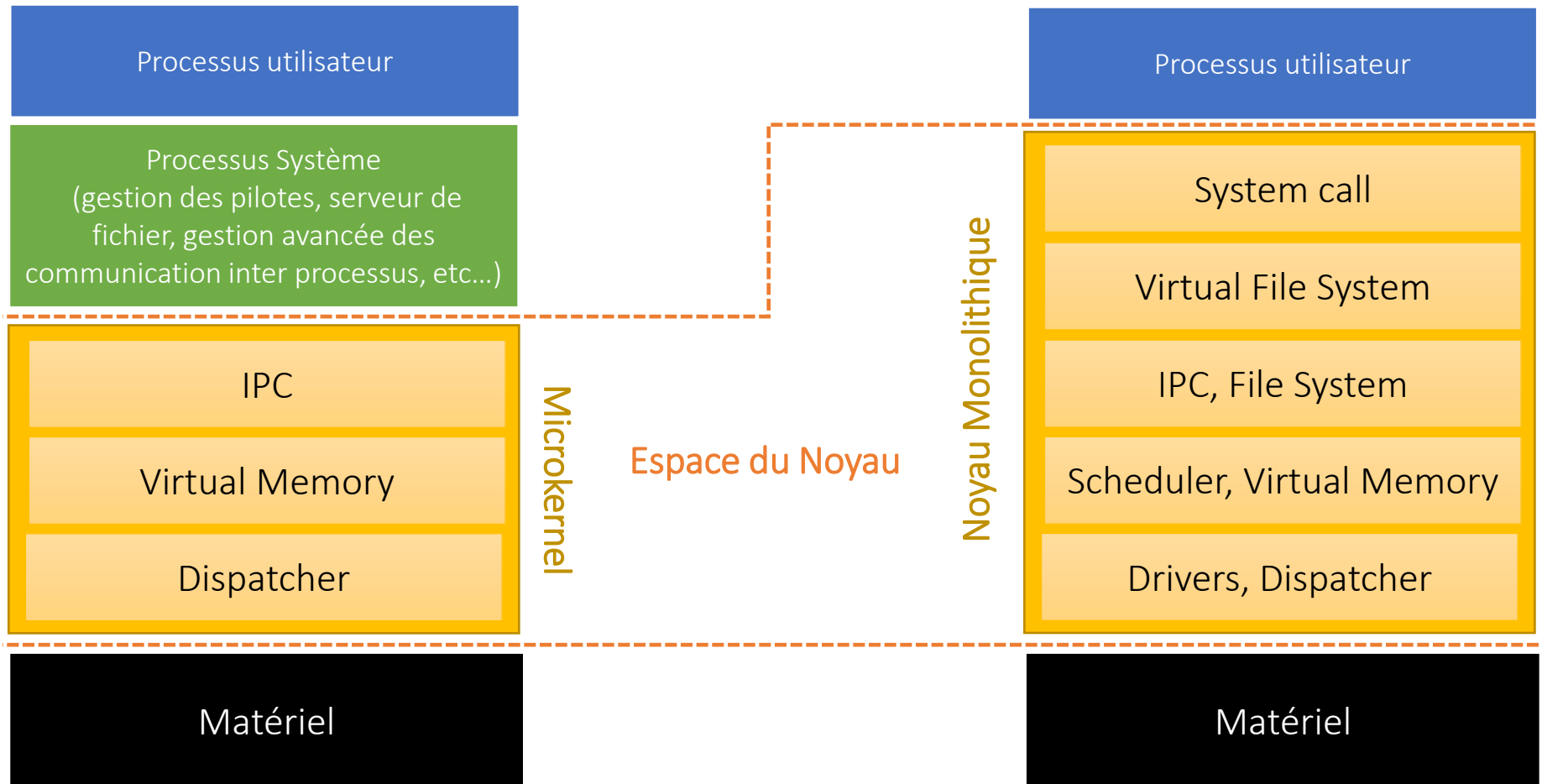
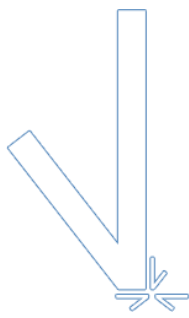
- Fourni une interface en lecture/écriture pour n'importe quel type de stockage (mémoire volatile ou non volatile) sous la forme d'un système de fichier hiérarchique . Cette interface est appelée Système de fichier virtuel:

📁 **File System, Virtual File System**

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

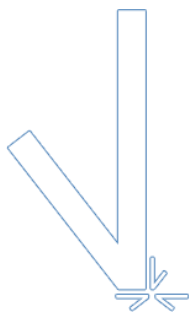
I.3.a Kernel – 2 catégories



I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.a Kernel - conclusion

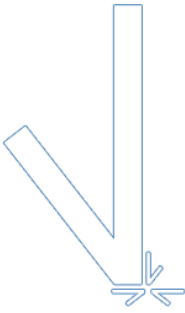


- Certains Systèmes d'Exploitation UNIX-like à **microkernel** :
 - [Minix](#).
 - ...
- Certains Systèmes d'Exploitation UNIX-like à **noyau monolithique** :
 - Les distributions Linux.
 - ...
- Pour **créer** un **noyau** à partir de rien, on peut programmer en :
 - **Assembleur** (en [langage assembleur x64 pour les CPU 64 bit](#));
 - **C/C++** en utilisant des bibliothèques dédiées qui produiront du code assembleur ([le site Internet OSDev.org est un bon début pour démarrer la programmation d'un système](#)).
- Nos cours seront centrés sur les Systèmes d'Exploitation à **noyau monolithique**.

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.c Pilotes





Pilotes

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.c Pilotes - fondamentaux

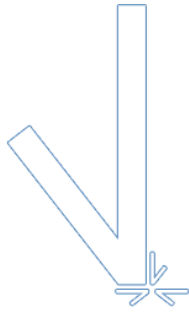


- Les pilotes de périphériques sont des blocs de code qui font :
 -  **Partie** du noyau;
 -  Ou qui sont **insérés** dynamiquement dans un noyau en cours d'exécution sous la forme de « **modules** ».
- Les pilotes ajoutent au noyau :
 - La gestion d'**Interruptions** entre des **périphériques matériels** et le **noyau**.
 - La gestion d'**Appels Systèmes** entre des applications de l'**espace utilisateur** et le **noyau**.
- Les pilotes sont :
 - Fournis par le constructeur du périphérique matériel ou développés par des membres de la communauté open-source contribuant au système d'exploitation;
 - Sous licence propriétaire ou open source.

I. Systèmes d'Exploitation

I.3. Noyau et Pilotes

I.3.c Pilotes – commandes utiles



- Sur Linux, la **liste** des **modules** insérés dynamiquement peut être consulté sur le CLI en utilisant la commande :

lsmod

- De **nouveaux modules** peuvent être ajoutés avec la commande suivante :

modprobe <file>

(modprobe suivi du chemin vers le fichier du module)

- On peut obtenir des **informations** concernant un module avec la commande :

modinfo <module name>

(modinfo suivi du nom du module tel que listé par lsmod)

- D'**autres** outils peuvent être utilisés pour configurer les modules ([comme indiqué ici](#)).



Utilisateurs et Sessions

(voir cours suivant: Systèmes d'Exploitation – Utilisateurs et sessions)