ReDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Systèmes d'exploitation (SYE)
Introduction

Profs Daniel Rossier, Alberto Dassatti, Salvatore Valenza

Version 3.1 (2017-2018)

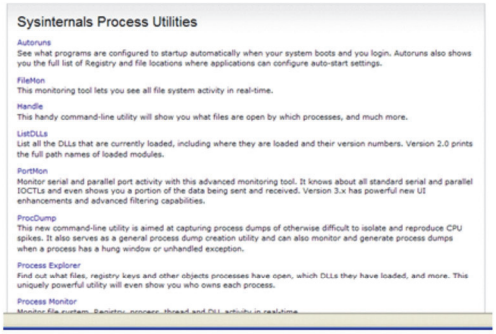
Contact & Infos diverses

- Contact
  - daniel.rossier@heig-vd.ch, alberto.dassatti@heig-vd.ch
  - **skype**: rossierd, alberto.dassatti, salvovalenza
- **Institut REDS**
  - *Reconfigurable Embedded Digital Systems*
  - http://www.reds.ch

- Bureau **D. Rossier → A21, A. Dassatti → A11, S. Valenza (Cisco)**
- ***http://www.linkedin.com***

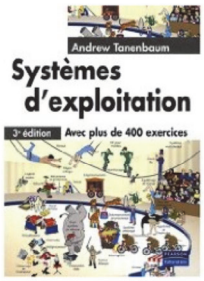
Liens utiles

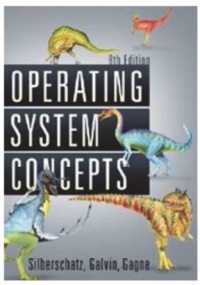
- *http://www.sysinternals.com*
- *http://www.kernel.org*
- *http://osxdaily.com*
- *http://www.osnews.com*



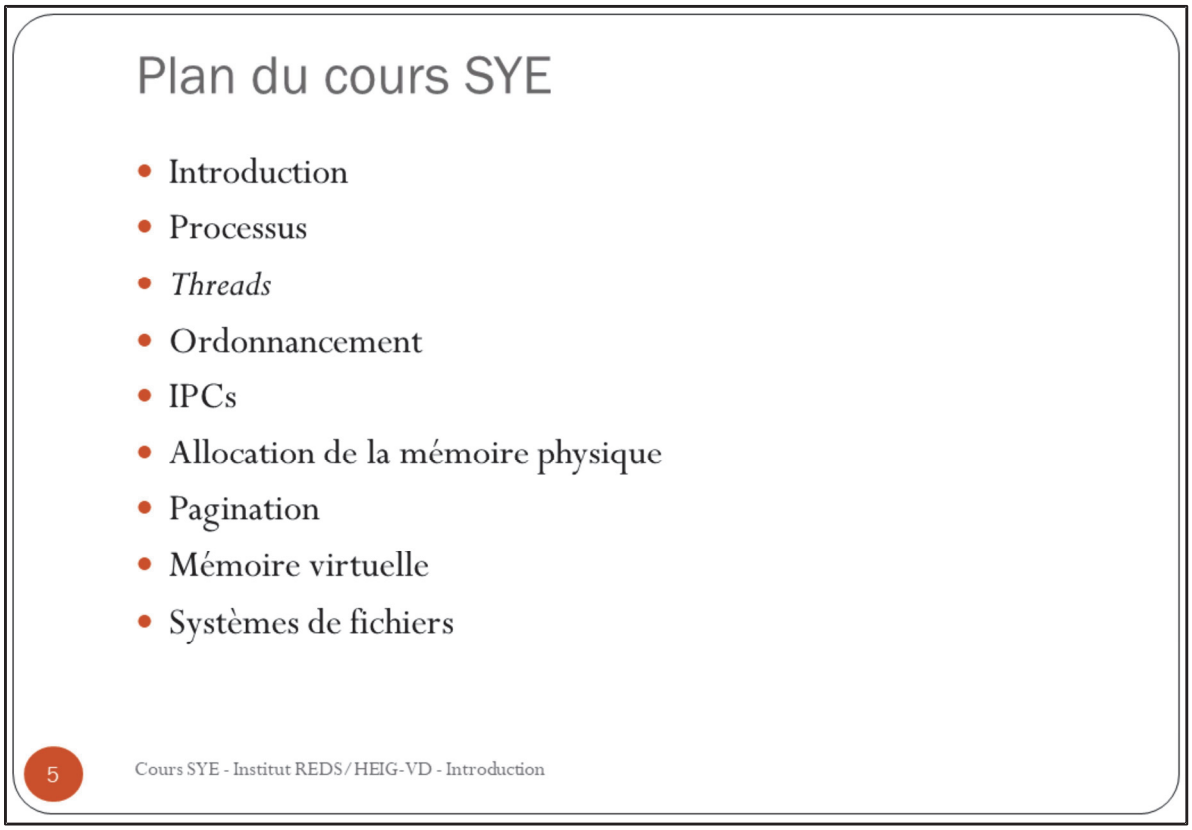
Bibliographie

- A. Silberschatz et al.: "Operating Systems Concepts", 8th edition, Wiley
- Andrew Tanenbaum: "Systèmes d'exploitation", 3ème édition





Les deux ouvrages mentionnés contiennent un très bon aperçu des différentes parties d'un système d'exploitation. L'ouvrage de *Tanenbaum* est quelque peu plus détaillé. L'un ou l'autre ouvrage est vivement recommandé pour toutes celles et ceux qui sont intéressés par les concepts détaillés d'un système d'exploitation. Toutefois, il ne constitue pas une obligation pour le cours SYE.



Le cours SYE comporte neuf chapitres traitant les aspects généraux fondamentaux de tout système d'exploitation. Un rappel introductif sur l'architecture matérielle permettra de comprendre les interactions matérielles-logicielle. Les briques de base telles que les processus et les *threads* seront ensuite étudiées, ainsi que les différentes politiques d'ordonnancement.

Les mécanismes IPCs (*Inter-Process Communication*) seront examinés et approfondis au travers de différents laboratoires.

Les techniques d'allocation et de gestion mémoire sont cruciales dans tout OS. C'est pourquoi, une attention particulière sera portée sur les techniques de traduction d'adresse et de pagination, ainsi que sur la notion de mémoire virtuelle.

Finalement, les mécanismes fondamentaux implémentés dans les systèmes de fichiers seront également examinés.

Plan (Introduction)

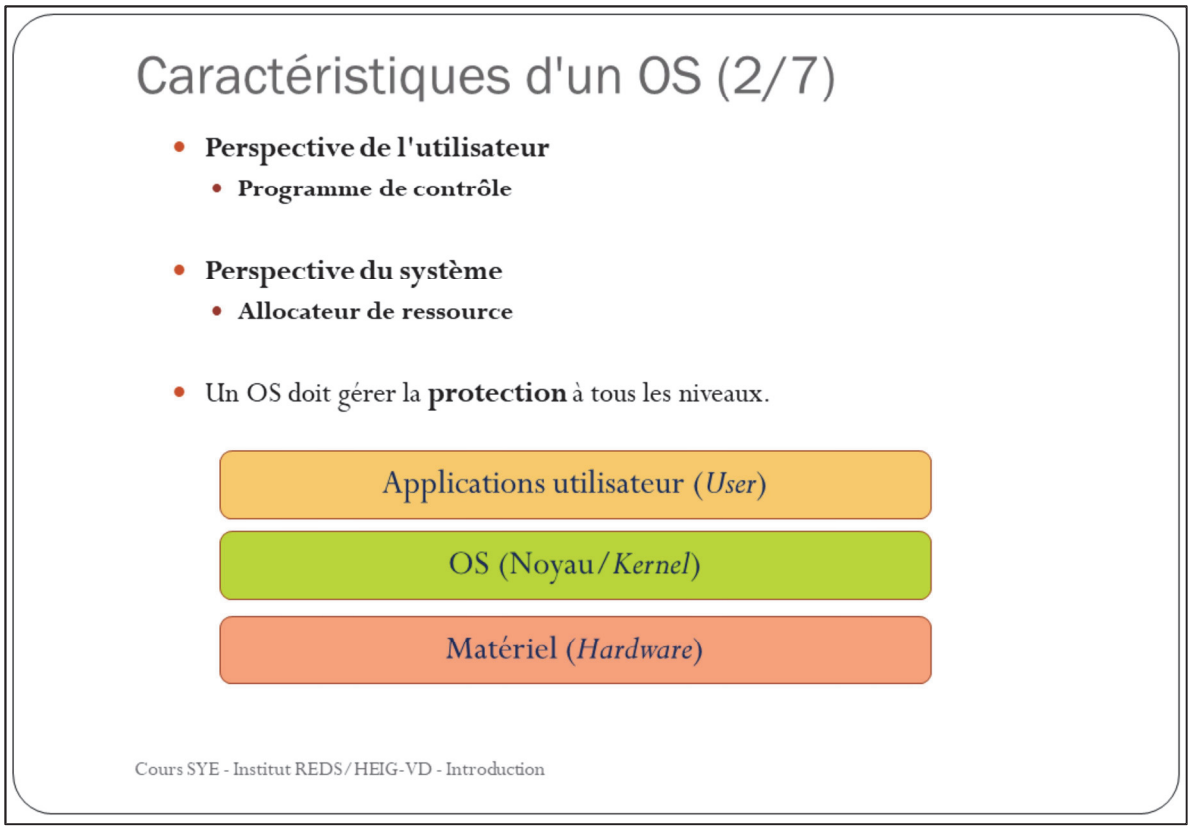
- Caractéristiques d'un OS
- Architecture matérielle (rappel)
- Architecture d'un OS
- Appels systèmes
- Construction d'une image binaire

Il est relativement difficile de donner une définition précise d'un système d'exploitation, tant son utilisation dénote des composants différents selon les domaines d'application et fournisseurs.

"Un **système d'exploitation** est l'ensemble de programmes central d'un appareil informatique qui sert d'interface entre le matériel et les logiciels applicatifs." (*source: Wikipedia*)

Or, lorsque l'on parle de Windows 10, on fait bien entendu allusion à un système d'exploitation, mais ce dernier comprend également plusieurs applications (pas seulement les composants nécessaires aux interactions machine-application).
Sous Unix/Linux/Mac-OS-X, un système d'exploitation comprend également un compilateur C.

Dans ce cours, nous nous concentrerons sur la partie dite "**noyau**" d'un OS, à savoir l'ensemble des composants qui gèrent les interactions entre le matériel et les applications utilisateur, ainsi que la gestion complète des applications tournant sur la machine. L'ensemble de ces différentes fonctionnalités de type *noyau* est décomposé en ***sous-systèmes***.

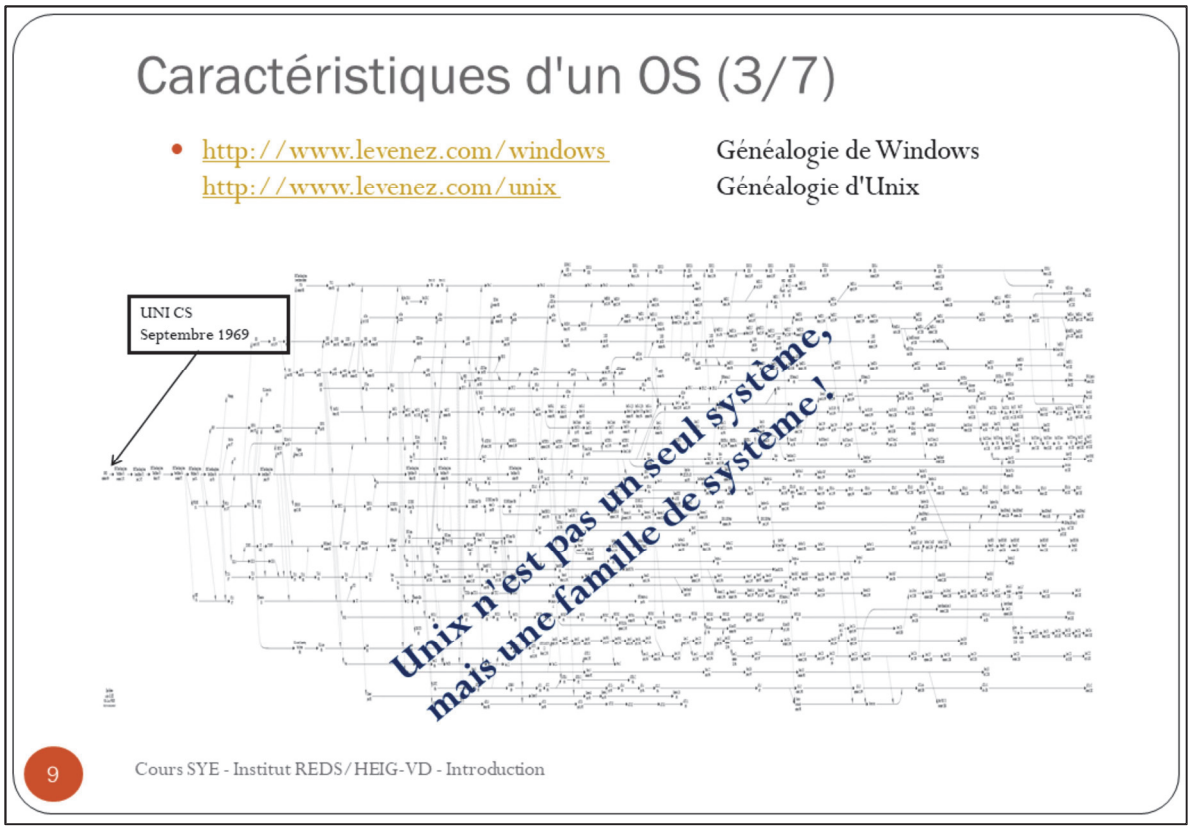


L'architecture d'un système classique tournant un système d'exploitation (OS) peut se découper en 3 couches principales (du bas vers le haut):

- Le **matériel** (*Hardware*)
- Le **noyau** de l'OS
- Les **applications** (tournant sur le noyau).

On comprend aisément que le noyau de l'OS aura la responsabilité de sécuriser les accès au matériel tout en gérant différents ***contextes d'exécution*** au niveau des applications. Cette architecture met en évidence la **criticité** du code qui s'exécute au niveau du noyau (le noyau d'un OS n'est rien d'autre que du code logiciel qui s'exécute à un moment ou à un autre). Nous verrons par la suite comment le processeur – du point de vue matériel – peut protéger certaines instructions afin d'aider l'OS dans sa tâche de supervision.

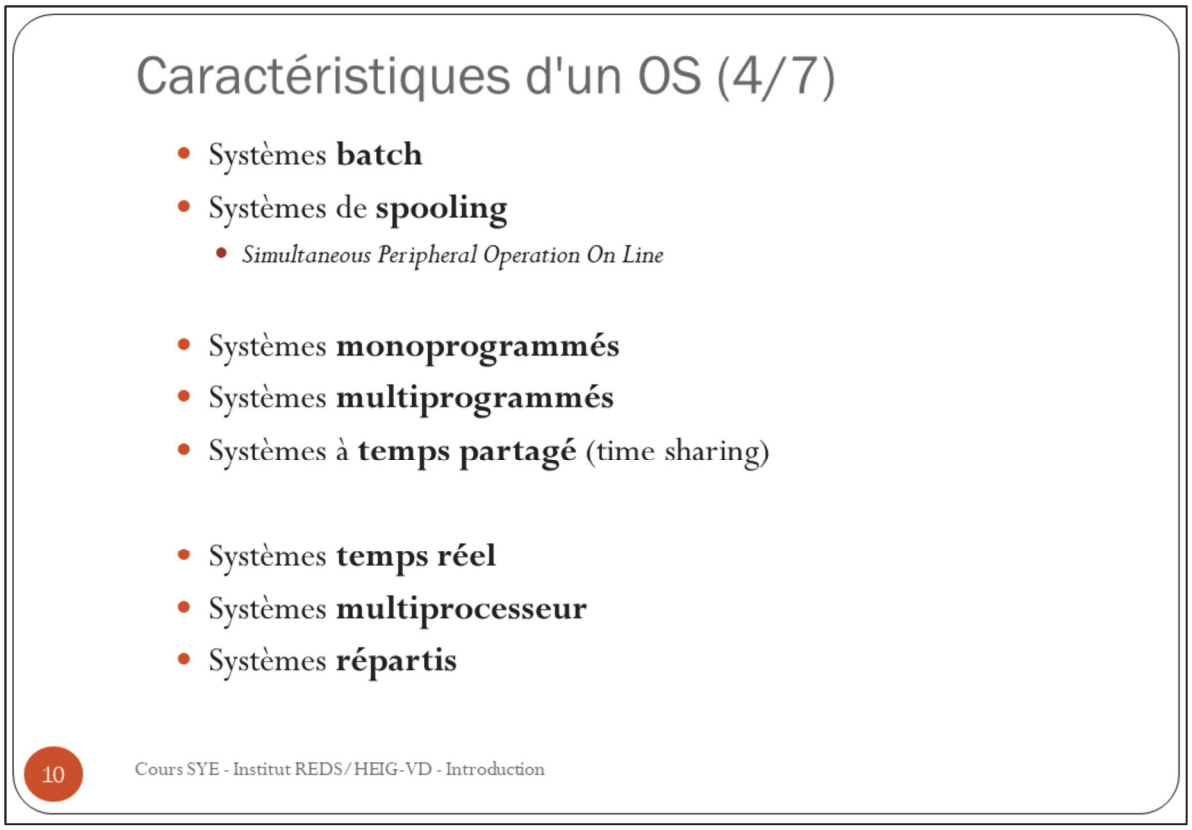
Dans ce cous, nous étudierons en détail les interfaces entre ces différentes couches, c-à-d la barrière qu'il y a entre les applications "utilisateur" et le noyau, ainsi qu'entre le noyau et le matériel.



Aujourd'hui, il existe une multitude de systèmes d'exploitation. On reconnait toutefois trois grandes familles: les OS provenant de *Microsoft,* ceux provenant de *Apple*, et ceux développés dans le cadre des logiciels libres (*Linux* pour n'en cité qu'un seul).

Les OS se déclinent ensuite en différentes variantes en fonction du domaine d'application: OS temps-réel *strict* pour les applications critiques, OS embarqués destinés à être déployés sur des petits systèmes et tournant des applications dédiées, OS de type GPOS (*General Purpose* OS) à utilité générale (PC, laptop, etc.), OS de type exécutif (l'OS est fortement lié avec l'application), etc.

Tous ces types d'OS respectent néanmoins les mêmes concepts fondamentaux liés aux architectures d'OS que nous étudierons dans ce cours.

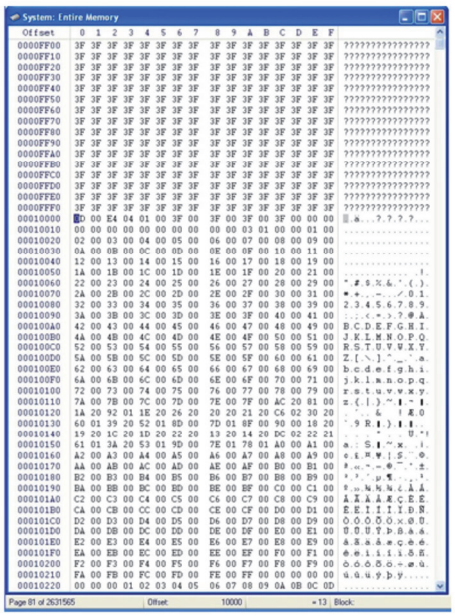


On peut trouver différents types d'OS en fonction de la catégorie des systèmes. Ces derniers peuvent présenter des contraintes bien particulières.

- Systèmes *batch*: un ensemble de commandes/programmes est exécuté automatiquement par le processeur (*Control Process Unit* - **CPU**).
- Systèmes de *spooling (Simultaneous Peripheral Operation On Line*): les entrées-sorties (I/Os) sont gérées automatiquement (par un *spooler*) et de ce fait, l'application principale n'a pas besoin d'attendre la terminaison de requêtes I/O (exemple typique: impression de documents).
- Systèmes **monoprogrammés**: un seul programme (utilisateur) peut être contenu en mémoire vive (en plus de l'OS).
- Systèmes **multiprogrammés**: plusieurs programmes (utilisateur) peuvent cohabiter en mémoire vive (RAM).
- Systèmes à **temps partagé**: le processeur est alloué aux différentes applications de manière équitable. Dans le cas des systèmes monoprogrammés à temps partagé, une opération de ***swapping*** est alors nécessaire: le programme en cours d'exécution est déchargé sur une mémoire secondaire (*swap-out*) et cède la place à un autre programme donnant lieu à un transfert de la mémoire secondaire vers la RAM (*swap-in*).
- Systèmes temps-réel: doit absolument garantir les temps de réponse dans des délais déterminés (on parle alors de systèmes déterministes).
- Systèmes à multiprocesseur: L'OS et ses applications peuvent tourner sur plusieurs processeurs (ou *cœurs* de processeur).

Caractéristiques d'un OS (5/7)

- **Espace d'adressage**
  - Espace *linéaire*
- **L'octet** (*byte*) représente la plus petite quantité de mémoire adressable.
  - Système *byte*-adressable
- **Une adresse pour un octet**
- Un registre peut contenir plusieurs bytes (registre 32 bits, 64 bits, etc.)



Avant d'examiner plus en détails les différentes catégories de système, il est nécessaire de comprendre la notion *d'**espace d'adressage***.

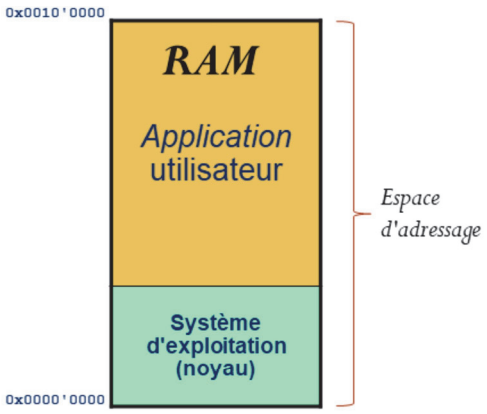
L'espace d'adressage représente l'ensemble des octets (*bytes*) que le processeur peut référencer (adresser). L'octet est la plus petite entité adressable par le processeur (accéder les bits de l'octet est bien entendu possible, mais cela se fait au moyen d'instructions arithmétiques/logiques). On comprend également que chaque octet peut être accéder individuellement et indépendamment. C'est pourquoi, chaque octet est *référençable* avec une adresse (une adresse détermine exactement l'emplacement d'un octet). On dit que le système est *byte*-adressable.

En revanche, le processeur peut transférer plus d'un octet à la fois (par exemple 4 octets, ce qui représentent 32 bits). Dans le plan mémoire (ou espace d'adressage), cette quantité de données correspond donc à quatre adresses.

La quantité de *bytes* pouvant être adressée dépend directement de la taille des registres du processeur; en effet, une instruction d'accès à la mémoire nécessite le stockage de l'adresse dans un registre en guise d'opérande (paramètre). Sur une architecture 32 bits, le registre pourra ainsi stocker une valeur d'adresse *représentable* sur 32 bits. La quantité de *bytes* adressable avec cette taille sera donc de **2^{32} octets (4 Go)**, les adresses allant de **0x0 à 0xffffffff (2^{32}-1)**.
Par la suite, nous verrons qu'un espace d'adressage ne permet pas seulement d'accéder la mémoire vive (RAM), mais également toute sorte de mémoires.

Caractéristiques d'un OS (6/7)

- **Monoprogrammation**
  - Un seul programme utilisateur chargé à une **adresse fixe**
  - L'OS est toujours résident en mémoire
    - Peu de fonctionnalités sont requises
    - Système léger
    - Exemple: *iOS (iPhone OS) < 4.0*

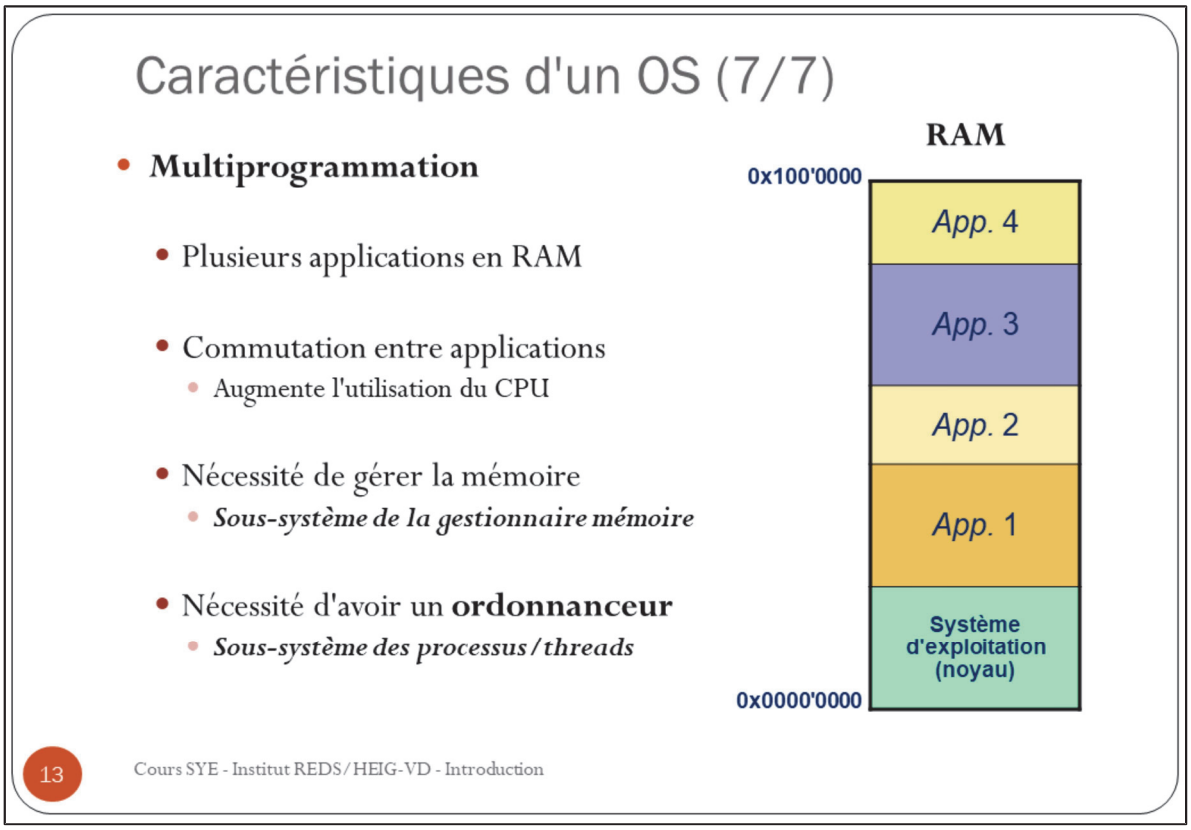


12 Cours SYE - Institut REDS/HEIG-VD - Introduction

Le système le plus simple consiste à faire tourner une seule application sur l'OS. C'est le cas de la *monoprogrammation*. Un système monoprogrammé est très souvent mis en place comme première solution dans des nouveaux systèmes, du fait de sa simplicité, de sa faible empreinte mémoire et de ses performances. L'adresse de chargement d'une application est **toujours fixe**; c'est dire que le compilateur peut produire un code binaire non relogeable contenant des adresses *absolues*.

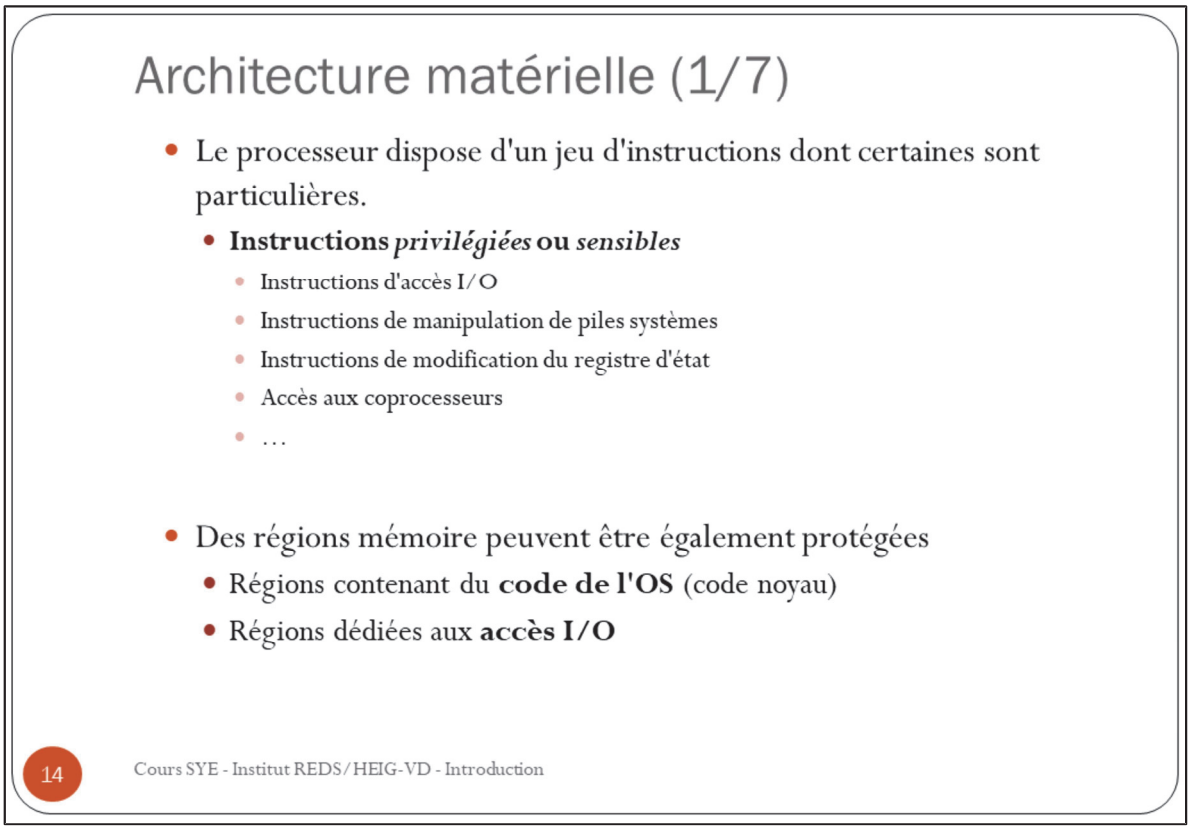
Malheureusement, il limite l'utilisation du système à une seule application et la commutation entre tâches nécessite le remplacement de la tâche courante par une autre.

La commutation entre applications peut aussi nécessiter une technique de *swapping* sur stockage secondaire afin de préserver l'état courant de l'application. L'échange d'applications entre mémoire secondaire et RAM (*swap-out / swap-in*) peut ralentir considérablement le système.



La multiprogrammation permet d'obtenir un taux d'utilisation CPU bien plus élevé que la monoprogrammation car plusieurs applications peuvent résider en RAM. Dès lors, la commutation entre applications dans le cas d'un système à temps partagé peut s'effectuer rapidement sans l'intervention d'une mémoire secondaire (nécessitant plusieurs requêtes I/O). En revanche, cette approche complique la gestion au niveau de l'OS qui doit charger les applications en mémoire lors du démarrage; en effet, il faut trouver un emplacement disponible et donc déterminer **l'adresse de base** du programme (processus). Typiquement, le chargement du code binaire d'un processus peut nécessiter un **réajustement des adresses** en fonction de son adresse de base. On parle dans ce cas de code *relogeable*.

De plus, l'OS doit gérer la commutation entre applications selon une politique d'ordonnancement.



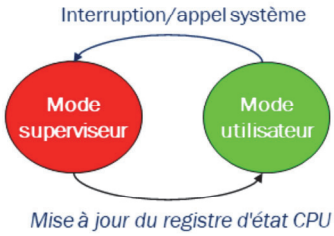
Un des rôles fondamental d'un système d'exploitation est celui de garantir la sécurité au niveau applicatif, donc au niveau de l'utilisateur. En effet, une application ne doit pas pouvoir interférer d'une manière illicite avec d'autres applications en cours d'exécution. De même, une application ne peut pas accéder le matériel à sa guise au détriment du noyau de l'OS.

C'est pourquoi, les microprocesseurs/microcontrôleurs modernes offrent un *support matériel* afin de gérer au mieux la protection durant l'exécution du code. En particulier, les ***instructions sensibles*** *(ou privilégiées)* ne peuvent être exécutées **que par le noyau** et non pas par les applications. Le processeur dispose ainsi de plusieurs modes d'exécution autorisant ou non l'exécution d'instructions sensibles.

Le processeur permet également de différencier les accès mémoire en fonction des adresses; certaines zones mémoires, dédiées à la gestion de périphériques par exemple, ne peuvent pas non plus être accéder par n'importe quelle application.

Architecture matérielle (2/7)

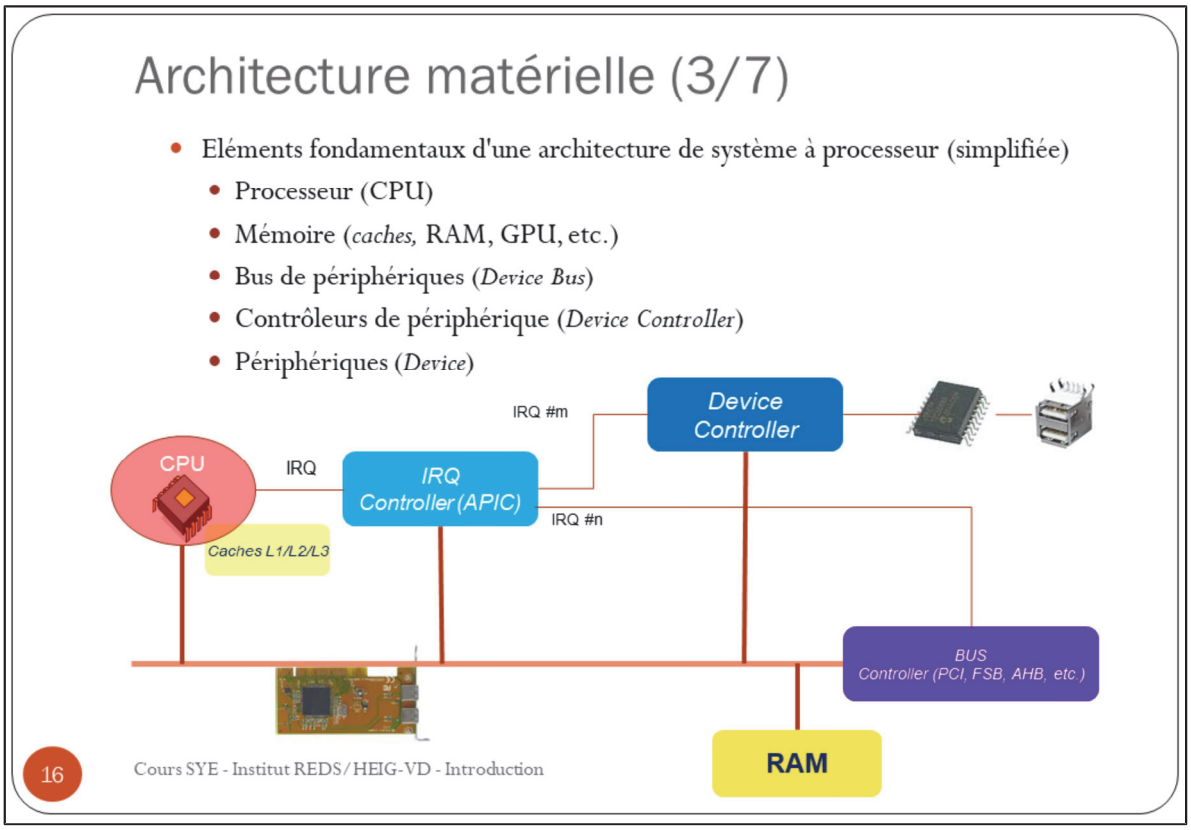
- Le processeur dispose de différents modes d'exécution.
  - Mode **utilisateur** (***USER***)
  - Mode **privilégié** ou **noyau** (***KERNEL***)
    - Plusieurs modes différents (*rings* sous *Pentium*)



D'une manière générale, les processeurs offrent la possibilité de différencier l'exécution de code s'exécutant au niveau du noyau d'un OS, de l'exécution de code s'exécutant au niveau d'une application (utilisateur). Il s'agit du mode *utilisateur (**user**)* et du mode *superviseur/noyau (**kernel**)*.

Au démarrage d'un système, le CPU est en mode noyau.

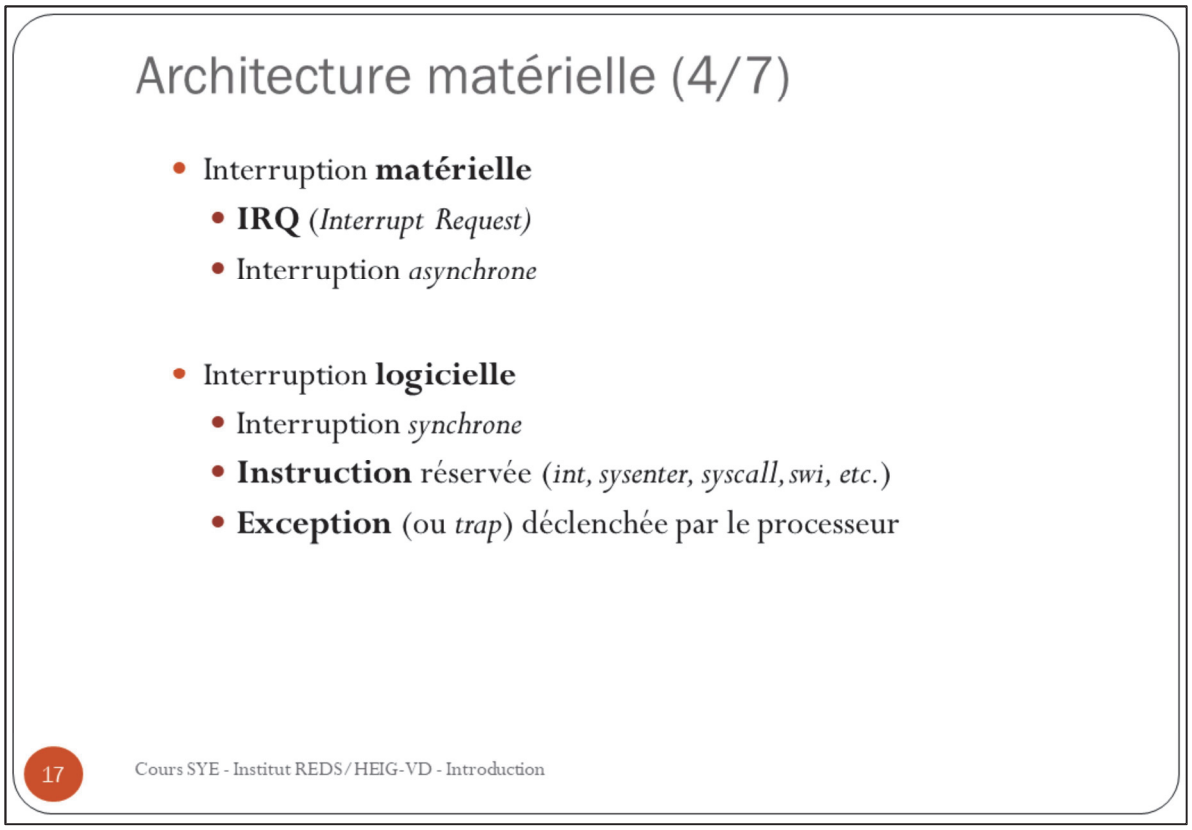
Le passage du **mode *kernel* en mode *user*** s'effectue en modifiant le registre d'état du processeur, ce qui nécessite l'exécution d'une instruction sensible. En revanche, le passage du **mode *user* en mode *kernel*** ne peut se faire que lors d'une interruption, ***matérielle*** ou ***logicielle*** (*exception/trap*).



Du point de vue d'un OS, l'interface avec le matériel consiste essentiellement à traiter **deux types de requêtes**:

- Les requêtes du processeur vers "l'extérieur" (c-à-d les périphériques résidant autour du processeur): cela inclus les accès en écriture/lecture vers la RAM, l'envoi de données vers un contrôleur de périphérique via le bus sur lequel le périphérique est interconnecté. Ces requêtes sont de nature synchrones car elles sont *initiées* par le processeur et sont exécutées immédiatement.

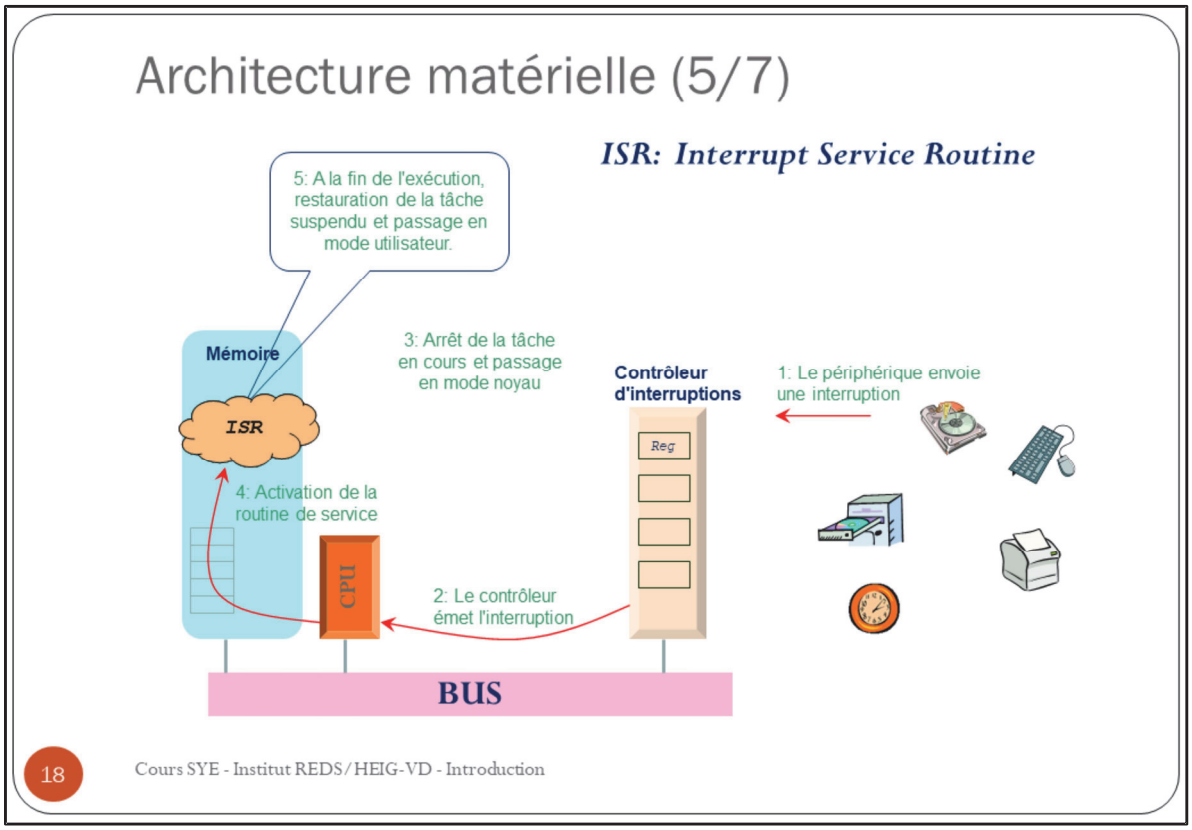
- Les requêtes en provenance de "l'extérieur" vers le processeur: ce sont typiquement les interruptions matérielles. Ce sont les contrôleurs de périphérique qui, via une ligne d'interruption physiquement dédiée, vont provoquer **l'interruption** du programme en cours d'exécution et ***dérouter*** l'exécution du code "ailleurs" dans la mémoire, comme nous le verrons plus tard. Ces requêtes sont toujours de nature *asynchrones* par définition.



La communication d'un périphérique (ou d'un co-processeur) vers le CPU s'effectue via des ***interruptions matérielles*** qui surviennent de manière **asynchrones**. En effet, un périphérique peut sans autre effectuer des opérations sans l'intervention du processeur. Dès que celui-ci souhaite informer le CPU (par exemple lorsqu'il a terminé de traiter une requête), il sollicite une ligne électrique dédiée entre lui et le CPU (en passant par le contrôleur d'interruption).

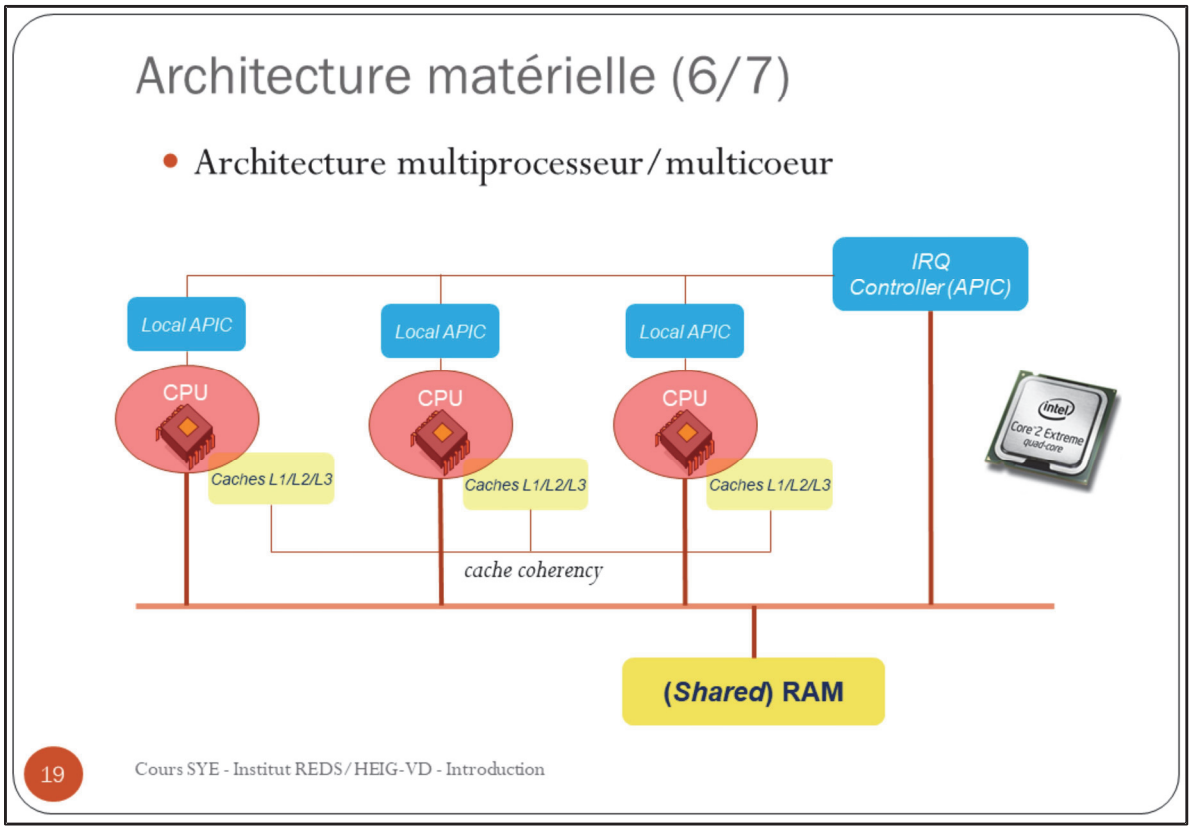
Il existe un autre type d'interruption: **l'interruption logicielle**. Dans ce cas, c'est l'exécution d'une instruction par le processeur qui entraîne une *exception* (aussi appelée *trap* en anglais) conduisant le processeur à se comporter "comme si" une interruption matérielle survenait. Une instruction effectuant un accès mémoire à une adresse invalide, l'exécution d'une instruction qui n'en est pas une ou encore une division par zéro entraînent ce type d'interruption. C'est le cas aussi avec des instructions particulières telles que *int, sysenter, etc.* Il ne s'agit **pas toujours** d'un problème à l'exécution. A l'inverse d'une interruption matérielle, une interruption logicielle est de type **synchrone**.

Dans les deux cas (interruption matérielle ou logicielle), le traitement au niveau du processeur est sensiblement le même: lors d'une interruption, une fonction particulière appelée ***routine de service*** *(ou **ISR** pour Interrupt Service Routine)* est immédiatement exécutée provoquant ainsi une *interruption apparente* du code en cours d'exécution. La routine de service associée à une interruption est une fonction appartenant au noyau de l'OS.



Chaque interruption matérielle ou logicielle est associée à une routine de service se trouvant en mémoire, chargée au démarrage de l'OS.

La figure ci-dessus illustre le mécanisme intervenant lors d'une interruption matérielle: chaque périphérique dispose de sa propre ligne d'interruption; en principe, cette ligne n'est pas reliée directement au processeur mais à un *contrôleur d'interruption* (1) permettant ainsi une gestion des interruptions plus efficace et indépendante du processeur; à ce niveau, il est possible par exemple de prioriser les interruptions si plusieurs arrivent en même temps, de les masquer/démasquer, etc.
Le contrôleur d'interruption propage ensuite l'interruption au CPU (2). Celui-ci doit aussitôt interrompre l'exécution en cours (il termine l'instruction) afin de poursuivre l'exécution de la routine de service, après avoir passé en mode noyau (3). De ce fait, il doit connaître l'emplacement, donc l'adresse, de cette routine. L'emplacement de cette routine peut varier d'un OS à l'autre; c'est pourquoi, le processeur utilise une indirection – appelée dans ce contexte ***vecteur d'interruption*** – qui associe à un numéro (ou *index*) l'adresse de la routine (4). Puis, la routine de service (ISR) s'exécute; elle effectue le traitement immédiat correspondant au vecteur d'interruption (dont acquitter interruption par exemple). La terminaison de la routine de service se caractérise par le retour en mode *user* puis la poursuite du code interrompu au moment de l'interruption (5).



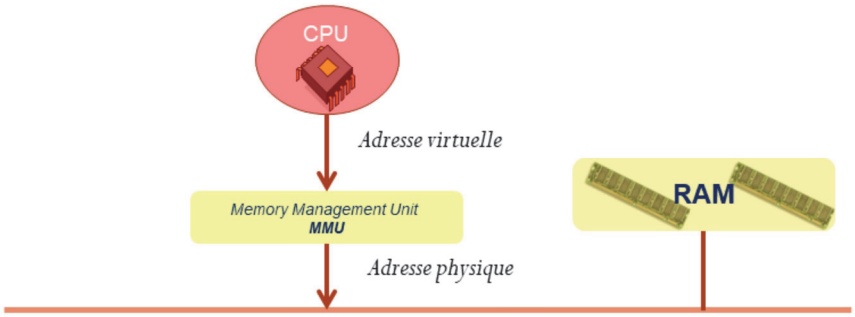
Aujourd'hui, la plupart des PCs/Laptops, *smartphones* et autres dispositifs embarqués sont très souvent équipés de processeurs multicœur (*multicore)*. La présence de plusieurs *cœurs de calcul* permet de répartir l'exécution d'applications (voire de parties d'applications) entre ces unités de traitement et donc d'accélérer considérablement les temps de réponse.

Le modèle le plus utilisé est basé sur une architecture de type SMP (*Symmetric Multi-Processing*) qui consiste à répartir la mémoire (RAM) entre les différents cœurs de calcul. Cela donne à chaque cœur la possibilité d'exécuter un code *commun* (que ce soit du code d'OS ou du code d'application). En revanche, chaque cœur dispose de ses propres antémémoires (mémoires *cache* ou *caches*).
Selon l'évolution de cette architecture, on trouve différents modèles avec un ou plusieurs caches individuels ou partagés. L'utilisation de *caches* individuels nécessitent une gestion de la cohérence entre les *caches* de telle manière que si l'un des processeurs modifie le contenu du cache, la donnée éventuellement présente dans un autre *cache* doit être également mis à jour.

Le traitement des interruptions matérielles (IRQ) fait également l'objet d'une adaptation au niveau de l'architecture avec la présence d'un contrôleur d'interruption local à chaque cœur de calcul. Selon l'OS et le modèle de contrôleur d'interruption, chaque IRQ sera alors *dispatchée* sur un ou plusieurs cœurs. En principe, le contrôleur global s'arrange pour *équilibrer* la charge des interruptions entre les cœurs de calcul.

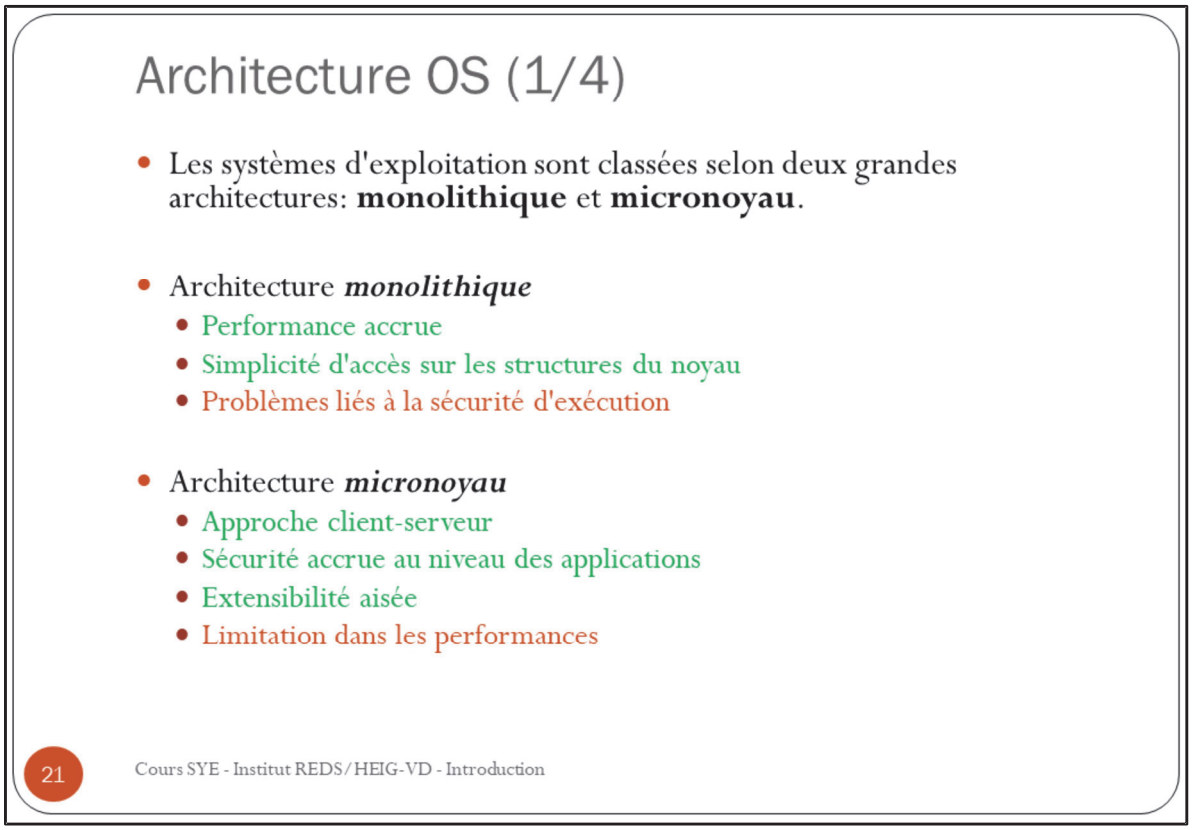
Architecture matérielle (7/7)

- Notion d'espace d'adressage
  - Un espace d'adressage est l'ensemble des adresses que le processeur peut accéder.
  - L'accès aux espaces d'adressage est sous contrôle d'une unité matérielle spécifique: la **MMU** (*Memory Management Unit*)
- La MMU traduit une adresse virtuelle en une adresse physique



Le processeur exécute une suite d'instructions dont beaucoup effectue des manipulations sur la mémoire. On se rappelle que le programme lui-même se trouve en mémoire, et par conséquent l'utilisation d'adresses mémoire intervient déjà dès l'opération de chargement d'instruction (opération *fetch*). Lors de la présence d'une MMU, les adresses *utilisées* par le processeur ne sont pas les adresses physiques (ou réelles), mais bien des adresses virtuelles. La MMU doit être donc configurée de telle sorte à ce que l'espace d'adressage *présenté* au processeur corresponde aux besoins de l'OS.

La notion d'espace d'adressage et l'utilisation d'une MMU sont fondamentales dans un OS: elles permettent l'introduction de la notion de **virtualisation**. Cette notion sera étudiée dans le chapitre dédié à la gestion mémoire dans un OS.

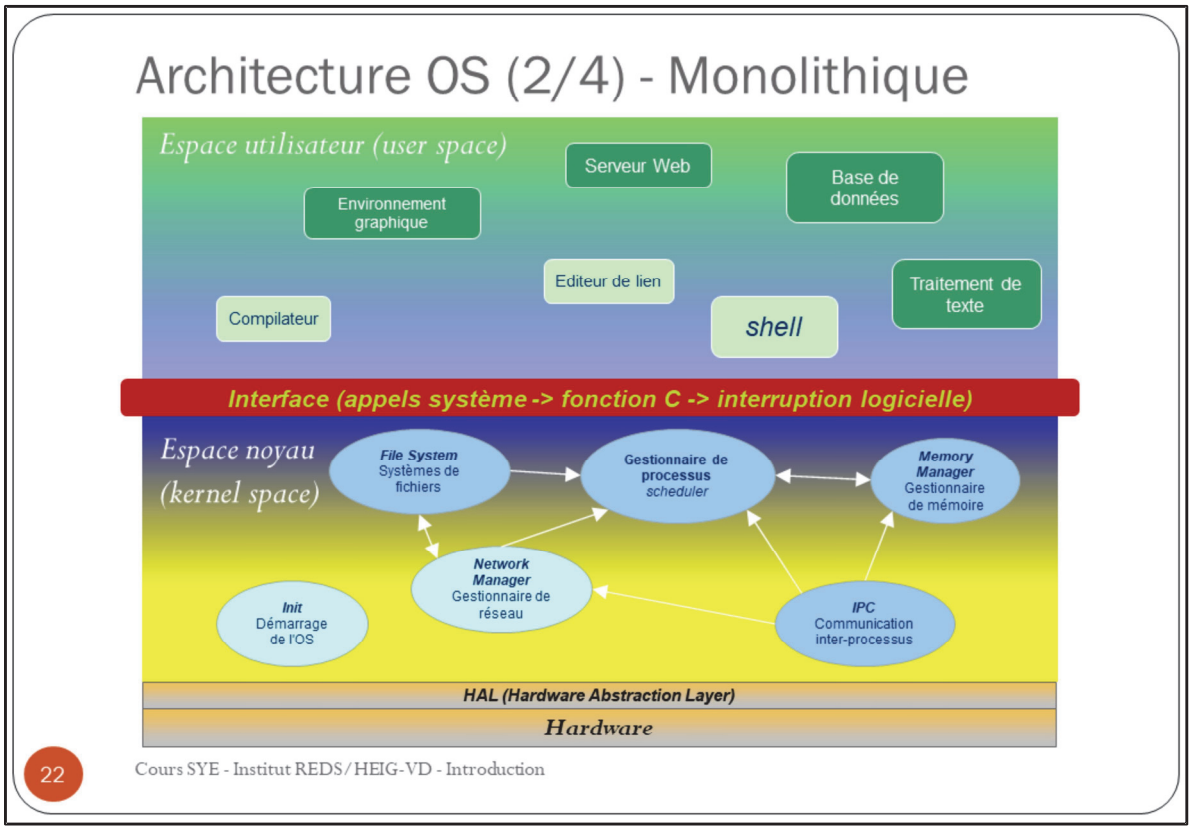


Un système d'exploitation (OS) peut se définir selon deux perspectives différentes: la perspective utilisateur, dans laquelle l'OS doit fournir une interface conviviale, performante et sécurisée aux applications utilisateur, et la perspective système, dans laquelle l'OS doit *piloter* le matériel en général.

Il va de soi que l'OS joue un rôle essentiel dans la sécurité et la robustesse du système, à tous les niveaux, ainsi que dans les aspects liés aux performances. C'est pourquoi, les systèmes d'exploitation n'ont sans cesse évolués pour accroître la qualité de ces différents paramètres. Bien souvent d'ailleurs, les OS doivent faire face à des compromis entre les aspects de sécurité et de performance.

Les architectures modernes d'OS reposent sur une notion fondamentale, celle des **espaces utilisateur (*user space*) et espaces noyau (*kernel space*)**. Ces deux espaces sont intimement liés à l'architecture matérielle du processeur sur lequel l'OS tourne. En particulier, les deux espaces sont liés aux modes d'exécution d'un processeur (deux modes fondamentaux: mode *user* et mode *kernel*. De plus, l'accès mémoire est largement conditionnée par les capacités du processeur à gérer les systèmes paginés/segmentés à travers la MMU (*Memory Management Unit*).

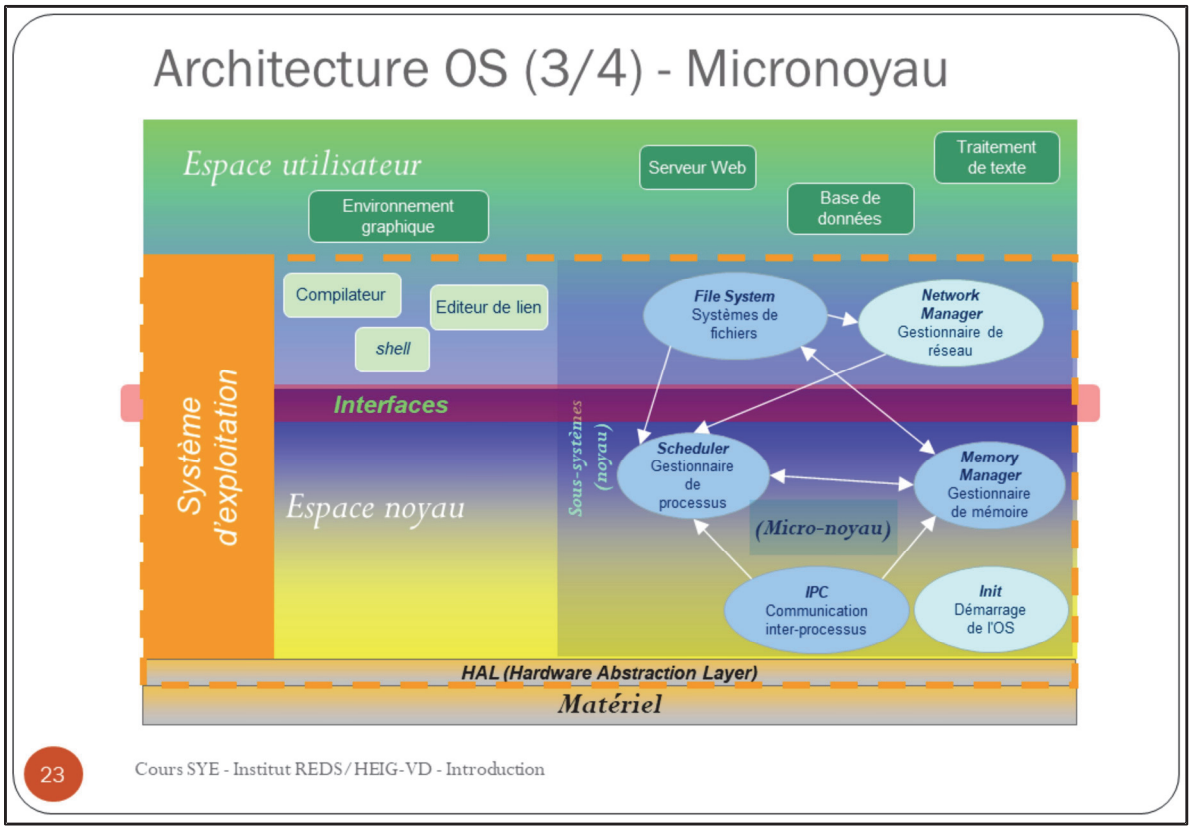
Une architecture monolithique consiste à placer tous les sous-systèmes (y compris les *drivers*) dans l'espace noyau. Une architecture micronoyau place un minimum de sous-système dans l'espace noyau, les autres sont relégués dans l'espace utilisateur, et "communiquent" au travers de mécanismes appropriés.



L'architecture monolithique d'un OS est la plus utilisée, car elle reste très attractive au niveau de ses performances. En effet, tous les sous-systèmes sont localisés dans l'espace noyau. Cela inclut généralement les pilotes de périphériques (ou *drivers*). Ainsi, l'accès aux structures noyau est facile, mais comprend des risques inhérents à la sécurité. Avec cette approche, il est possible de planter l'OS et par conséquent, toutes les applications en cours d'exécution.

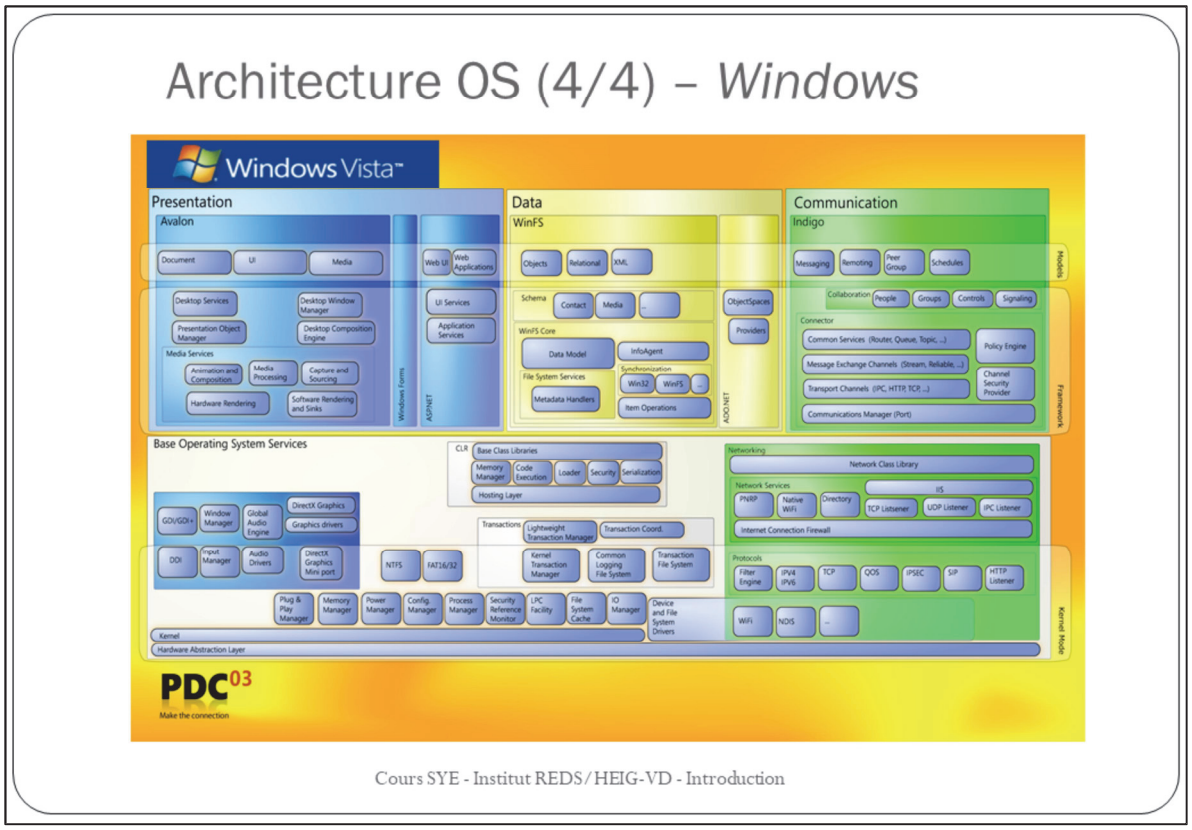
En revanche, le fait que peu de couches d'abstraction soient présentes permet à de tels OS d'offrir d'excellentes performances. Souvent, les applications temps-réels critiques (temps-réel dur) tournent également dans l'espace noyau, afin d'être le plus réactif possible aux événements extérieurs.

La séparation entre espace utilisateur et espace noyau est possible grâce à l'utilisation des appels système (*syscalls*). Les appels système sont particuliers: ils nécessite l'utilisation d'une instruction machine dédiée. Ils seront traités en détail dans les chapitres suivants.



L'architecture micronoyau quant à elle est beaucoup plus sécurisée que l'architecture monolithique, puisqu'un minimum de fonctionnalités tourne dans l'espace noyau. Ainsi, il est beaucoup plus difficile de planter l'ensemble du système. Comme nous le verrons plus tard, la notion de *processus* introduit un compartimentage des applications avec une séparation très nette des espaces d'adressage, évitant ainsi qu'une application n'interfère avec les autres. Aussi, un processus ne pourra pas accéder le noyau de n'importe quelle façon.

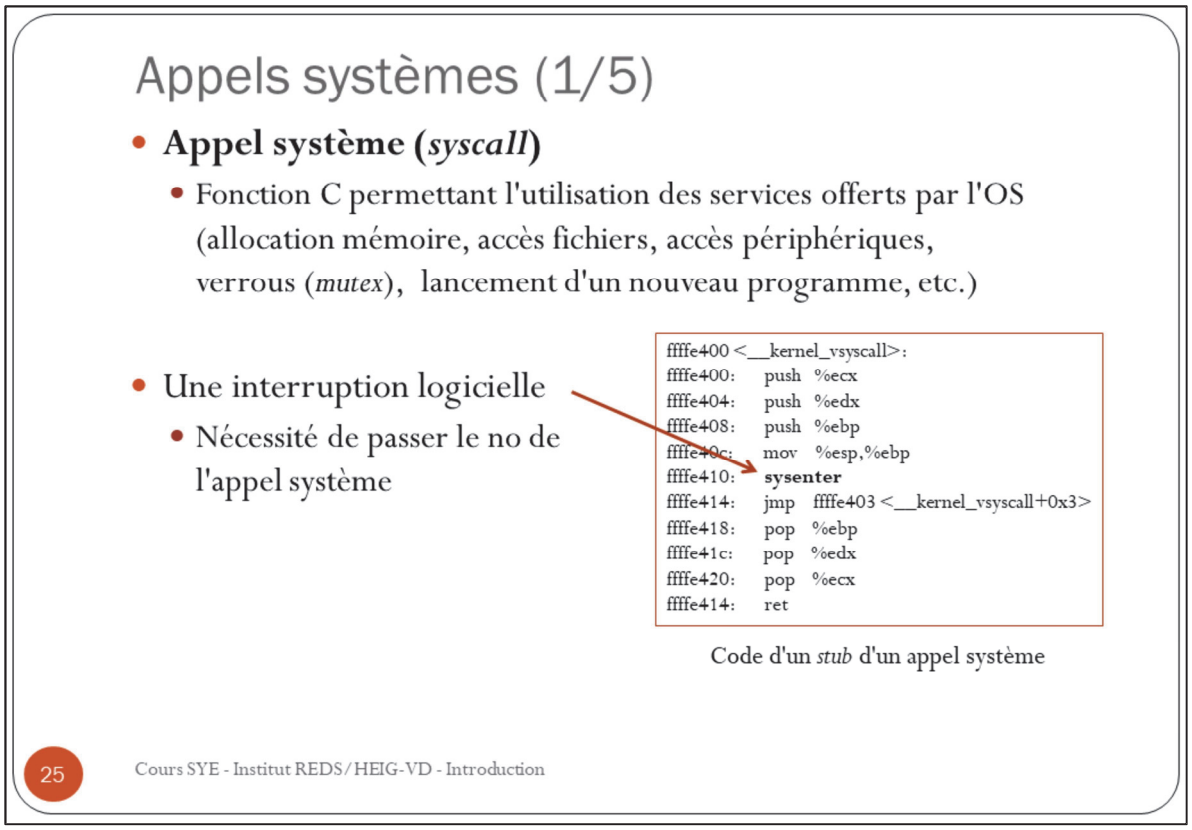
Comme les sous-systèmes n'ont pas directement accès aux structures du noyau, la communication entre eux s'effectuent au même titre qu'un processus utilisateur. Cette communication, en revanche, devrait être basée strictement sur une approche *client-serveur,* une telle approche permettant une spécification très claire de l'interface au niveau des sous-systèmes, ainsi qu'une définition sans ambiguïté du protocole de communication.



La figure ci-dessus illustre un exemple d'architecture d'un OS (*Windows Vista*). Il est intéressant de voir la découpe "espace utilisateur / espace noyau".

L'espace noyau est caractérisé sur dans cette architecture par la zone *kernel mode* laissant penser que l'architecture de *Windows* est de type micronoyau. Or, on trouve néanmoins le système de fichiers (NTFS) et d'autres sous-systèmes qui, dans une pure architecture micronoyau, devrait apparaître dans l'espace utilisateur. C'est dire que *Windows* n'est pas complètement de type micronoyau, mais que les ingénieurs ont dû trouver des compromis entre une architecture monolithique offrant des performances élevées et une architecture micronoyau permettant d'accroître la sécurité du noyau.

On parle alors d'architecture *hybride*; c'est ce type d'architecture que l'on retrouvera également avec *Mac OS X* par exemple.

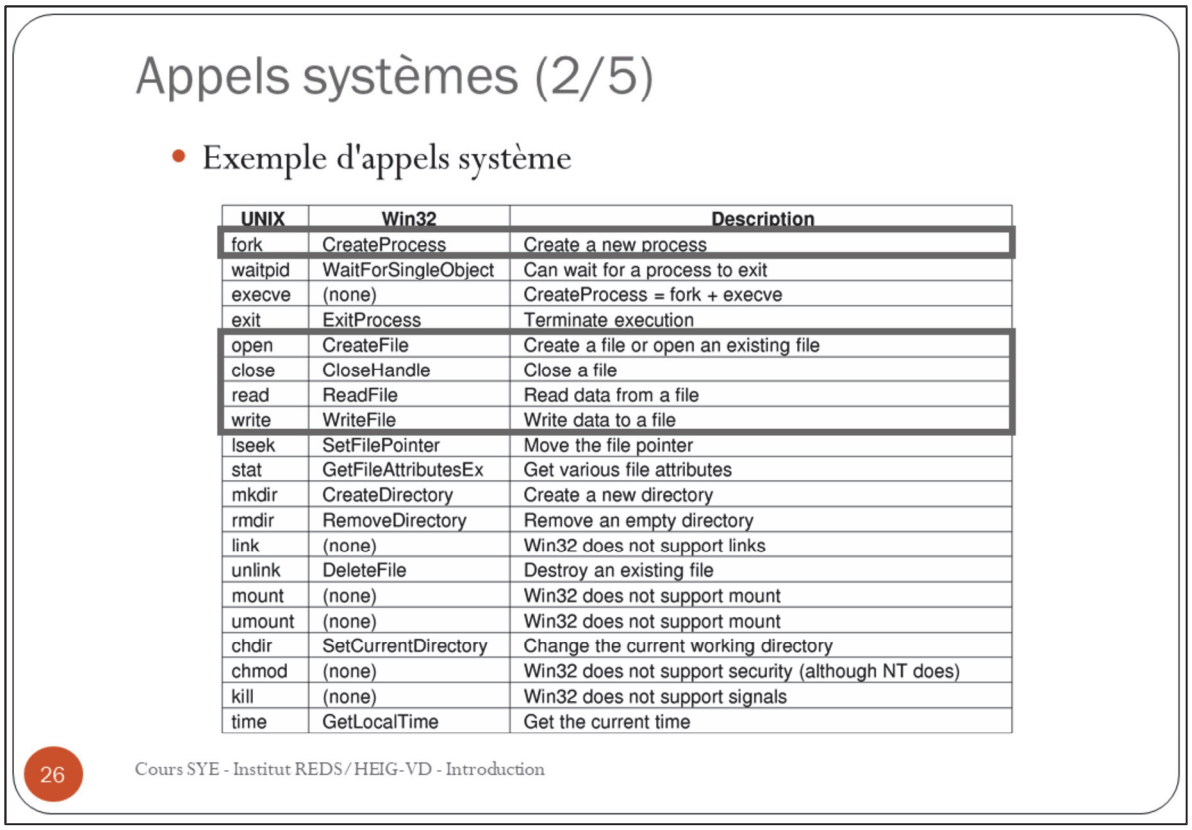


L'appel système permet de *franchir* la barrière logique entre l'espace utilisateur et l'espace noyau. Autrement dit, dès que cette barrière est franchie, le processeur exécute à nouveau du code noyau (appartenant à l'OS). C'est pourquoi, le passage du code *utilisateur* a du code *noyau* nécessite le basculement du processeur en mode noyau, et donc l'utilisation obligatoire d'une interruption logicielle, autrement dit, d'une instruction machine particulière.

L'instruction d'interruption logicielle sur les processeurs *Intel* est l'instruction *int n*, où *n* correspond au numéro de l'interruption.

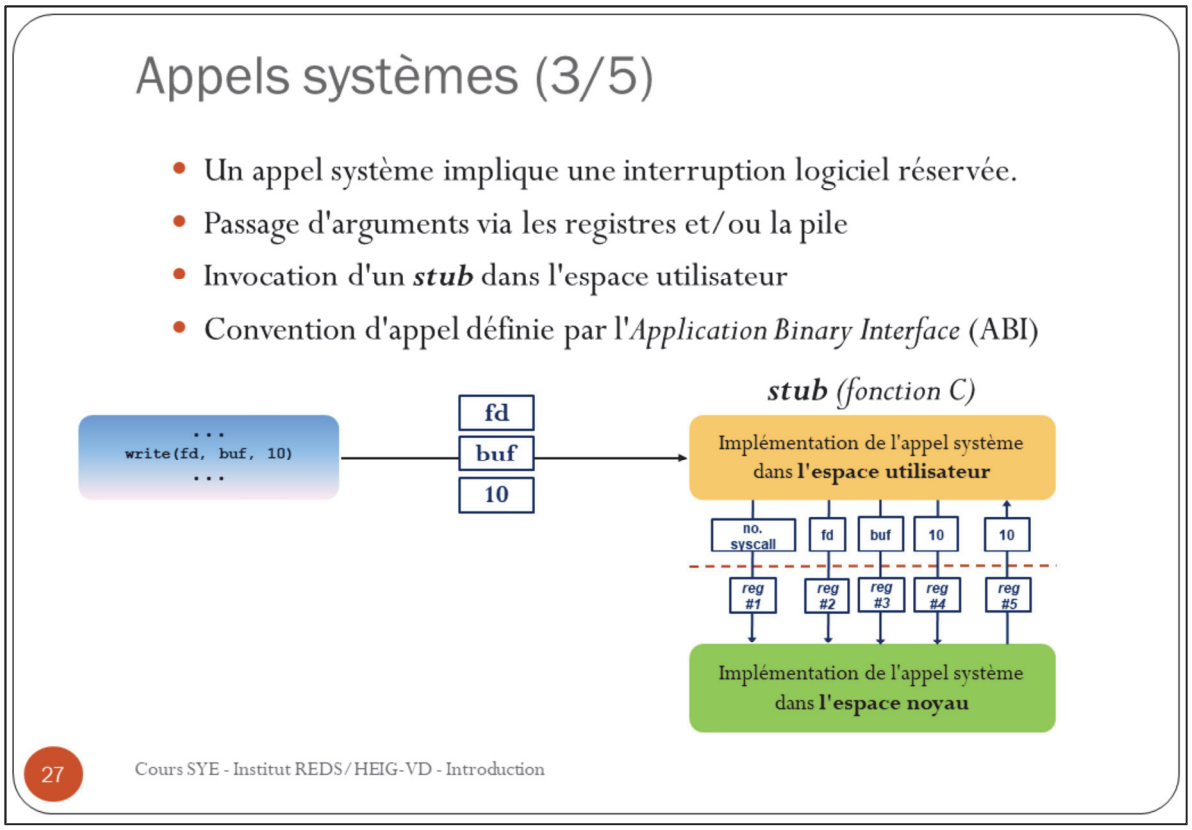
Comme nous l'avons vu précédemment, la plupart de ces interruptions sont utilisées par le processeur comme déroutement de l'exécution lors d'erreurs dans le traitement d'une instruction (division par 0, mauvais alignement, accès mémoire interdit, etc.). Bien que réservé avant tout au processeur, ces interruptions peuvent être également déclenchées (volontairement) par le programme.

Une des interruption logicielles les plus utilisées par les programmes utilisateur sera naturellement l'interruption dédiée aux appels systèmes. Sur les processeurs *Intel*, il peut s'agir de l'instruction *sysenter* ou *int 0x80*. Sur une architecture ARM, il s'agit de l'instruction *swi*, sur MIPS, il s'agit de l'instruction *syscall*.



Concrètement, du point de vue du développeur, un appel système correspond à une fonction C. Cette fonction prépare les arguments sur la pile (ou directement dans des registres), met le numéro de l'appel système sur la pile (ou dans un registre), exécute l'instruction déclenchant l'interruption logicielle, et gère le retour de la routine de service.

Dans les autres langages, il est possible d'utiliser directement des appels système grâce aux possibilités d'insérer du code C permettant l'appel des fonctions correspondantes. En Java par exemple, c'est le mécanisme *Java Native Interface* (JNI) qui permet de faire appel à une fonction écrite en C; avec *JNI,* il est possible de transmettre des données entre du code Java et du code C.

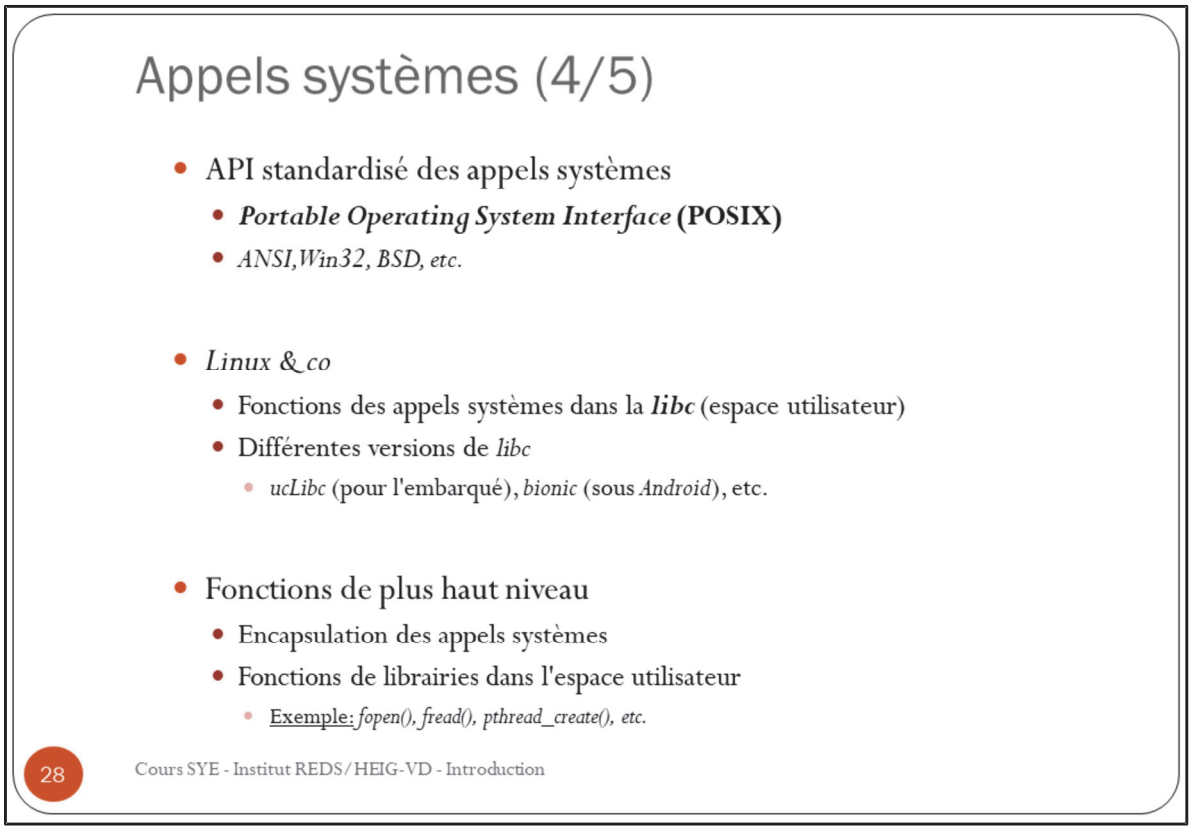


Comme nous l'avons évoqué, l'implémentation d'un appel système se trouve dans le noyau de l'OS, et dépend donc fortement de celui-ci. Afin que l'application puisse utiliser ces fonctionnalités du noyau, il faut donc un code permettant de générer l'interruption logicielle associée aux appels système. Ce code correspond à la partie de l'implémentation de l'appel système au niveau de l'espace utilisateur et s'appelle habituellement ***stub*** (souche/socle) de l'appel système.

Le *stub* d'un appel système est petit: il consiste en un code d'entrée, de l'interruption logicielle (*int 21h, sysenter, swi, etc.*) et du code de sortie.

Le code d'entrée permet de stocker le numéro de l'appel système correspondant, ainsi que ses éventuels **arguments**, soit dans des registres, soit dans une zone mémoire (pile, autre) dont l'adresse est passée dans un registre, et tout ceci selon une convention qui doit être établie entre l'espace utilisateur et l'espace noyau. Cette convention entre les librairies et le noyau, aussi connue sous le nom de ***Application Binary Interface*** (ABI), permet au noyau de récupérer de manière univoque le numéro du *syscall* ainsi que ses arguments.
Lors de l'interruption logicielle, le code du noyau "prend la main" et exécute l'appel système correspondant au numéro indiqué par le code d'entrée.

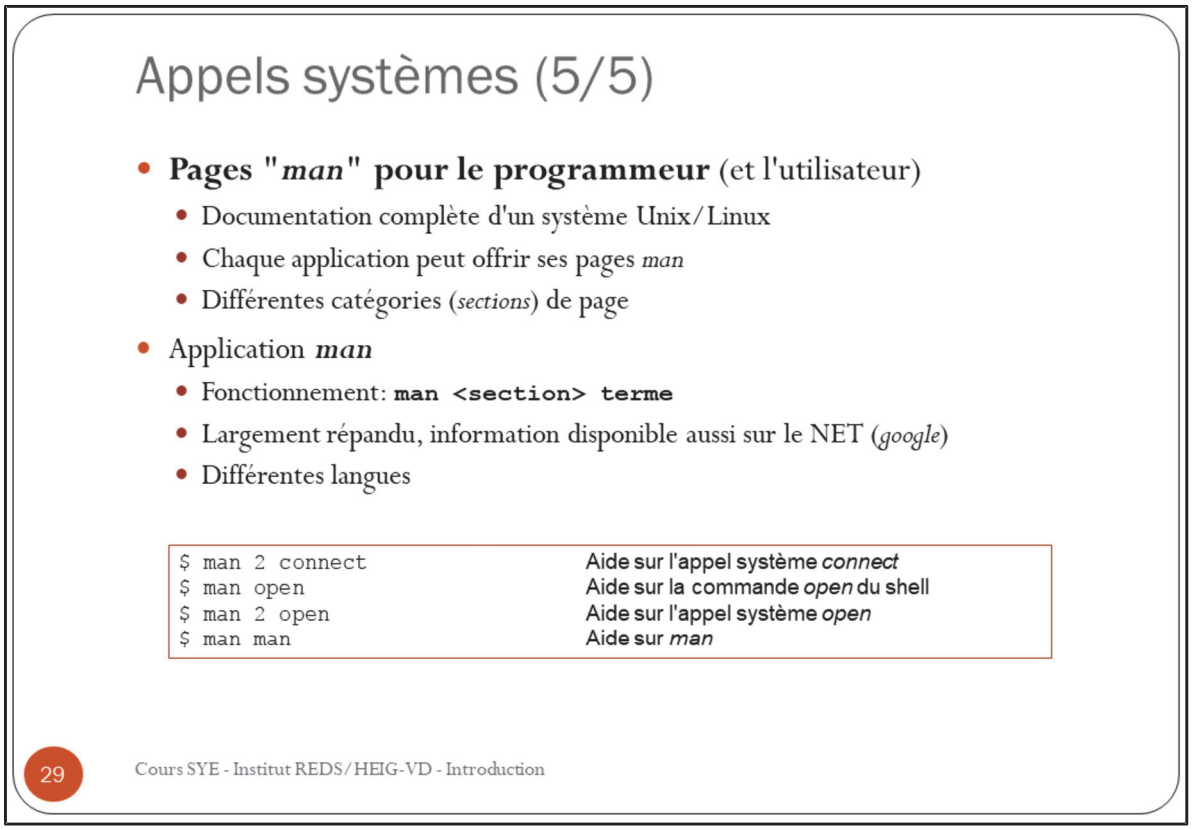
A la fin de l'appel système, le noyau peut déposer une valeur de retour dans un registre (ou en mémoire), valeur qui sera traitée par le code de sortie de l'appel système dans l'espace utilisateur.



Les appels systèmes (en tant que fonctions C) sont normalement standardisés selon différentes normes: la plus connue est la norme POSIX (*Portable Operating System Interface*). Sous Windows, les appels système sont implémentés dans la librairie *Win32* décrite dans le référentiel MSDN. POSIX est également supportée, mais dans une moindre mesure. Sous *Linux,* les appels système sont implémentés dans la fameuse ***libc*** de l'espace utilisateur. C'est donc dans cette librairie que nous retrouverons l'implémentation des *stubs* des appels système.

D'autres normes relatives aux APIs des appels systèmes sont: *ANSI, SVr4, X/OPEN, BSD*.

D'une manière générale, les appels système sont relativement coûteux car ils entraînent une interaction entre espace utilisateur et espace noyau, et la gestion de cette interaction nécessite des changements d'état du processeur. C'est pourquoi, dans la mesure du possible, il est plus judicieux de recourir à l'utilisation de fonctions de plus haut niveau qui offrent des fonctionnalités similaires, mais dont le traitement est optimisé dans l'espace utilisateur et le recours aux appels système est minimisé. Cela est possible, par exemple lors d'un *read()*, en gérant des *buffers* dans l'espace utilisateur. Ces fonctions sont regroupés dans différentes librairies de l'espace utilisateur.

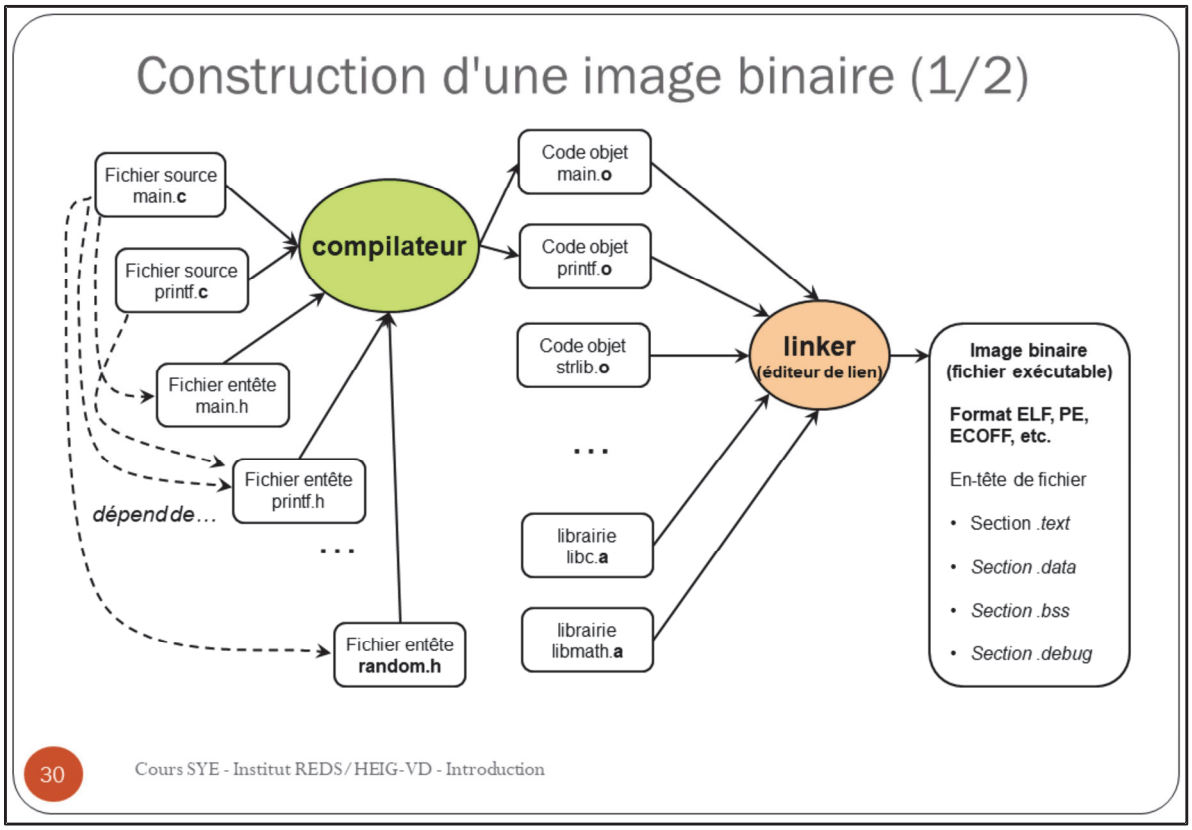


La description complète des appels systèmes et des autres fonctions (ainsi que l'ensemble des commandes et toute application d'ailleurs) peut être obtenu grâce aux pages ***man***. L'application *man* est très connue et permet facilement - la plupart du temps - d'obtenir les informations que l'on recherche. Facile d'utilisation, l'application ***man*** permet d'accéder à ces pages organisée en section.

Les sections sont les suivantes:

1 - Commandes utilisateur
2 - **Appels système**
3 - **Fonctions de bibliothèque**
4 - Fichiers spéciaux
5 - Formats de fichier
6 - Jeux
7 - Divers
8 - Administration système
9 - Interface du noyau *Linux*

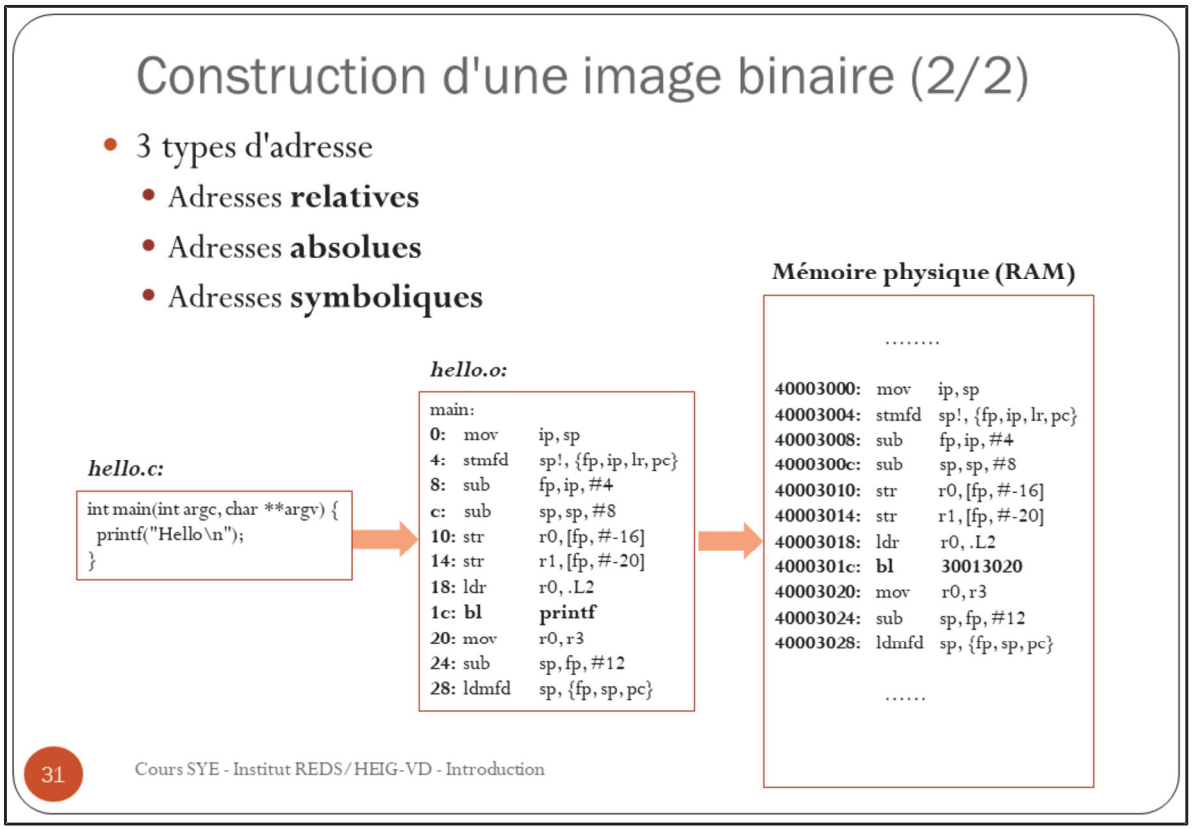
```
MAN(1)                                  Manual pager utils                                  MAN(1)

NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-C  file]  [-d]  [-D]  [--warnings[=warnings]]  [-R  encoding]  [-L  locale]  [-m  system[,...]]  [-M  path] [-S list] [-e extension] [-i|-I]
       [--regex|--wildcard] [--names-only] [-a] [-u] [--no-subpages] [-P pager] [-r prompt] [-7] [-E encoding] [--no-hyphenation] [--no-justification] [-p
       string] [-t] [-T[device]] [-H[browser]] [-X[dpi]] [-Z] [[section] page[.section] ...] ...
       man -k [apropos options] regexp ...
       man -K [-w|-W] [-S list] [-i|-I] [--regex] [section] term ...
       man -f [whatis options] page ...
       man  -l  [-C  file]  [-d]  [-D]  [--warnings[=warnings]]  [-R  encoding]  [-L  locale]  [-P  pager] [-r prompt] [-7] [-E encoding] [-p string] [-t]
       [-T[device]] [-H[browser]] [-X[dpi]] [-Z] file ...
       man -w|-W [-C file] [-d] [-D] page ...
       man -c [-C file] [-d] [-D] page ...
       man [-?V]

DESCRIPTION
       man is the system's manual pager.  Each page argument given to man is normally the name of a program, utility or function.  The manual page associ-
       ated with each of these arguments is then found and displayed.  A section, if provided, will direct man to look only in that section of the manual.
       The default action is to search in all of the available sections following a pre-defined order ("1 n l 8 3 2 3posix 3pm 3perl 3am 5 4  9  6  7"  by
       default,  unless overridden by the SECTION directive in /etc/manpath.config), and to show only the first page found, even if page exists in several
       sections.
```



La chaîne de compilation permet de construire une application sous forme d'un fichier binaire exécutable, appelé aussi ***image binaire*.** L'image binaire est donc le résultat d'une compilation et d'une opération de *linkage* ou *édition des liens*.

Le compilateur *traduit* le code source (C, C++, Java, assembleur, etc.) en un fichier binaire intermédiaire, appelé *fichier objet* (ou ***code objet***). Le code objet contient les instructions machines sur lequel le code est censé s'exécuter. Il s'agit donc d'un fichier binaire, mais ne peut pas encore être exécuté en tant que tel. En effet, le code contient des instructions qui effectuent des sauts à des fonctions qui sont définies *ailleurs* que dans le code compilé. Par exemple, si le code source fait appel à la fonction *printf()*, cette dernière n'aura pas son implémentation dans le code source: elle est stockée dans une *librairie*, typiquement fournie avec le compilateur.

C'est l'opération de *linkage* qui se chargera de récupérer le code des fonctions manquantes dans les librairies correspondantes, et de former ainsi l'image binaire finale.



L'emplacement du code et des données dans un fichier ou en mémoire nécessite l'introduction de trois types d'adresse:

a) les adresses **relatives**

L'emplacement mémoire est déterminé sur la base d'un *offset* (positif ou négatif). Typiquement, le compilateur ne connait pas *à priori* l'emplacement des instructions ou des données dans la mémoire physique (RAM). Une instruction machine peut également utiliser une adresse relative; c'est le processeur qui déterminera la *vraie* adresse (par rapport à la position courante par exemple).

b) les adresses **absolues**

Ces adresses représentent l'emplacement "final" dans la mémoire. Le compilateur peut générer un code binaire contenant des adresses absolues. Ces adresses ne seront plus changées au chargement du code en mémoire.

c) les adresses **symboliques**

Ce type d'adresse est utilisé par le compilateur lorsque celui-ci n'a pas la *visibilité* d'une instruction ou d'une donnée (elle peut se trouver dans un autre fichier ou résider en mémoire physique). La référence est donc représentée par un nom symbolique qui sera traité ultérieurement par l'éditeur de lien (*linker*).

Références

- A. Silberschatz et al.:
  "**Operating Systems Concepts**", 8th edition, Wiley

- Andrew Tanenbaum
  "**Systèmes d'exploitation**", 3ème édition