Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

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

Systèmes d'exploitation et Environnements d'Exécution Embarqués (SEEE)
Introduction

Module d'approfondissement MSE

Prof. Daniel Rossier

Version 1.9 (2017)

REDS
Reconfigurable & embedded Digital Systems

Contact & Infos diverses

- Contact
  - daniel.rossier@heig-vd.ch
  - skype: rossierd
- Institut REDS
  - *Reconfigurable Embedded Digital Systems*
  - http://www.reds.ch
- Bureau A21
- Assistant : Magali Fröhlich (magali.frohlich@heig-vd.ch)
  - Bureau A23

Le développement d'application sur des systèmes embarqués nécessite une connaissance approfondie des architectures matérielles et des microcontrôleurs, ainsi que les mécanismes logiciels dits de "bas niveau". La maîtrise des interactions logicielles-matérielles est cruciale. L'environnement d'exécution dans lequel tourne les applications doit souvent faire l'objet d'une attention particulière, puisqu'il doit être capable de *gérer les interactions matérielles*, respecter des contraintes temps-réel, assurer la sécurité inter-application, garantir *l'isolement des applications*, etc.

Les couches logicielles apparaissant traditionnellement au niveau d'un système d'exploitation doivent également fournir les abstractions nécessaires aux applications afin de faciliter avant tout le développement de celles-ci et de gérer la complexité matérielle.

C'est pourquoi ce module d'approfondissement propose de traiter certains aspects liés à l'informatique de bas niveau (appelé aussi informatique technique) dans un contexte de systèmes embarqués, en examinant le cœur (ou noyau) de l'OS *Linux,* très largement répandu pour ce type de développement.
Dans ce contexte, les trois techniques particulières suivantes seront étudiées en approfondissement:

- L'émulation logicielle au niveau processeur et périphérique
- Le fonctionnement de pilotes périphériques (*drivers*)
- La virtualisation embarquée

Déroulement du module SEEE (1/3)

- Matériel du cours et laboratoires
  - http://reds.heig-vd.ch
  - http://reds.heig-vd.ch/formations/master/SEEE

- **Evaluations**
  - Un travail écrit (examen intermédiaire) → note TE
  - 3 laboratoires évalués (rapports & code) → moyenne labo
  - Moyenne semestre = (note TE + moyenne labo) / 2

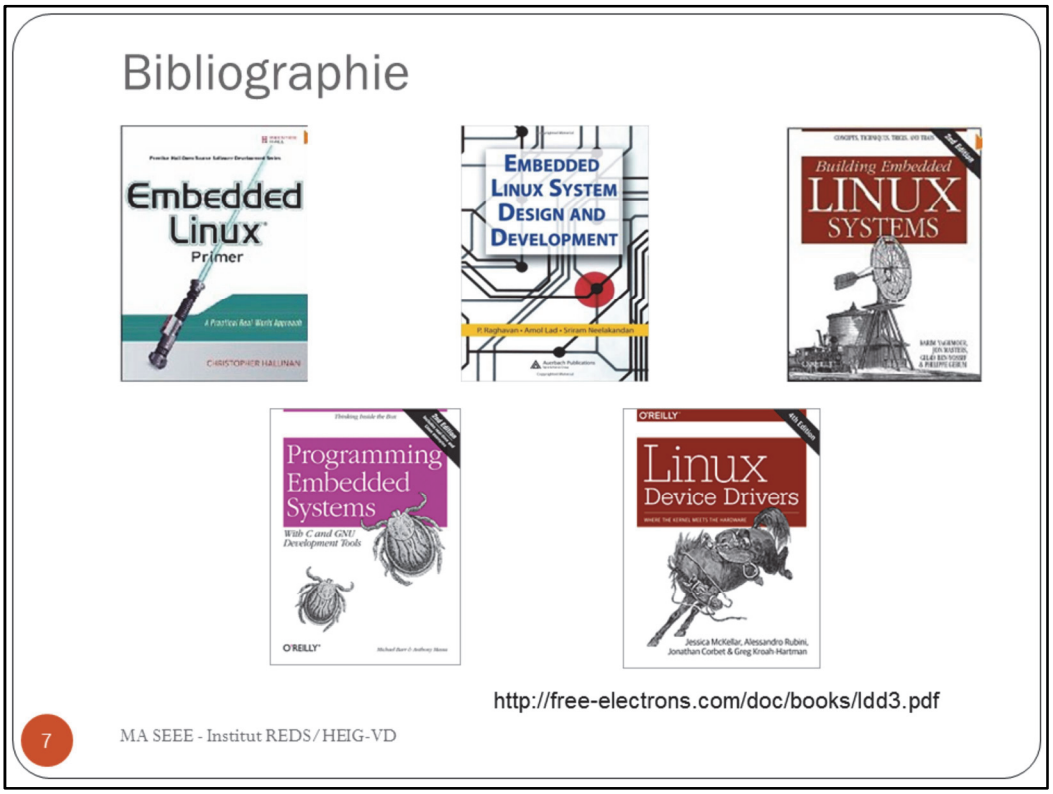
- **Examen final** (50 %)
  - Note finale = (moyenne labo + examen final) / 2

Déroulement du module SEEE (2/3)

- Le module SEEE s'accompagnera de différents laboratoires.
  - Prise en main de la plate-forme REPTAR
  - Emulation de périphérique REPTAR
  - *Driver* dans cible émulé et physique
  - Virtualisation sur cible émulé et physique
- Les laboratoires s'effectuent en binôme.

Déroulement du module SEEE (3/3)

- Introduction
- Emulation
- *Drivers*
- Virtualisation



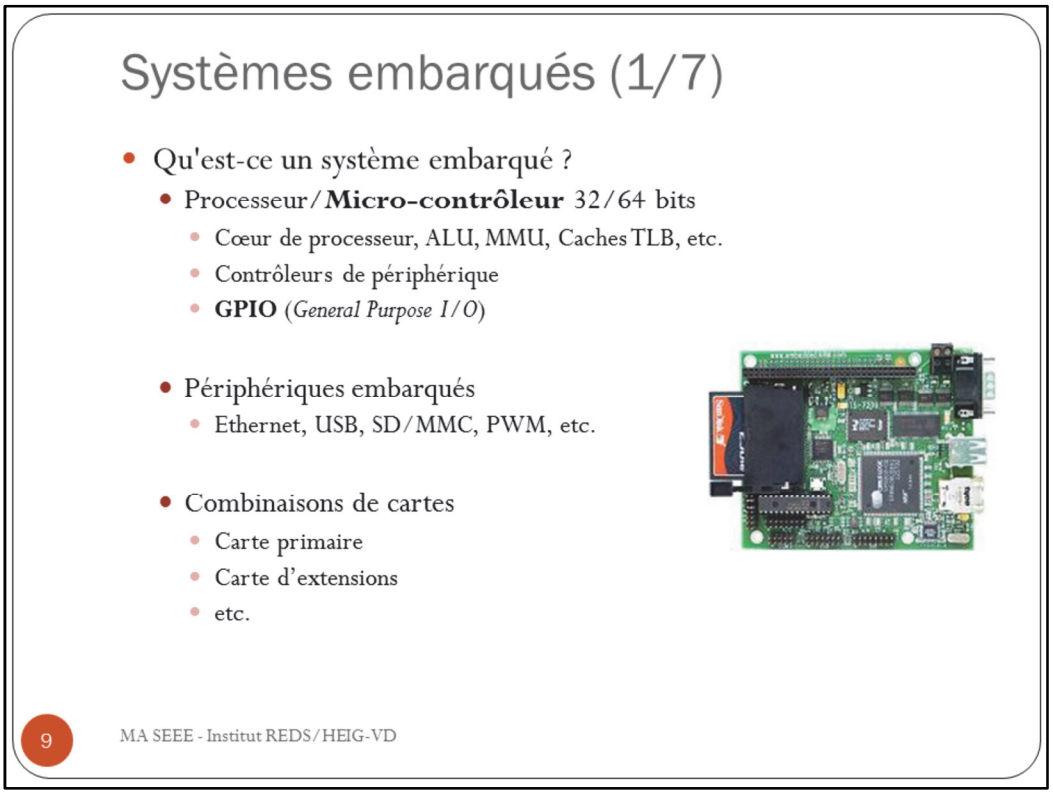
Les livres ci-dessus sont largement répandus et utilisés dans le développement des systèmes embarqués. Des versions téléchargeables peuvent être facilement trouvées sur Internet.

Les liens suivants sont également importants :

- http://free-electrons.com
- http://stackoverflow.com
- http://www.osnews.com

Plan (Introduction)

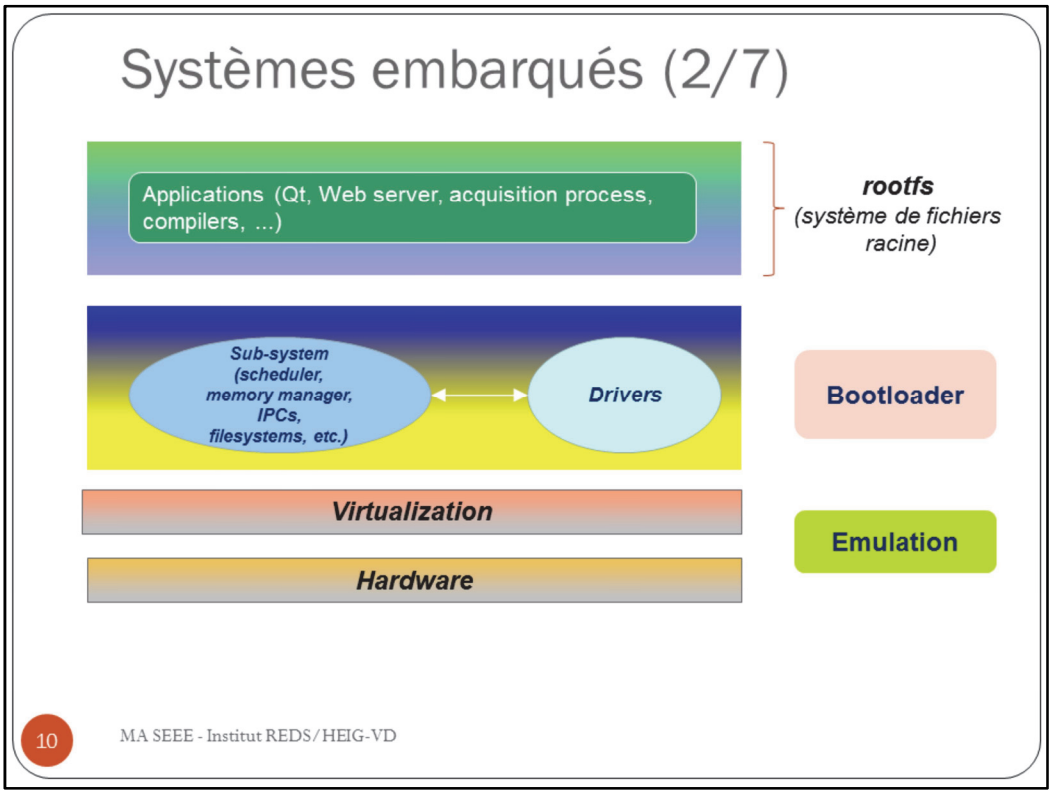
- Systèmes embarqués
  - Matériel et logiciel
  - Adressage I/O, plan mémoire
- Plate-forme REPTAR
- Architectures OS
- Appels systèmes
- Espaces d'adressage
- Processus & *threads*



Un système embarqué (on parle parfois de système enfoui) est un système électronique, piloté par un logiciel, qui est complètement intégré au système qu'il contrôle. On peut aussi définir un système embarqué comme un système électronique soumis à diverses contraintes.

Les systèmes embarqués sont présents partout : rien que sur vous, il y en a généralement au moins deux. Mais qu'est-ce qu'un système embarqué? En deux mots, c'est un dispositif à base de processeur ***dédié à un tâche précise,*** par exemple une montre ou un téléphone portable. On en trouve aussi dans les voitures (plusieurs dizaines par voiture en moyenne), et ils vont du plus simple (montres) aux systèmes de pilotage automatique de fusées, en passant par les *set-top-box* (décodeurs TV numériques), les consoles de jeu, les appareils médicaux, les tables de mixages, etc.

Aujourd'hui, une grande majorité de plates-formes embarquées fonctionne avec des microcontrôleurs 32 bits (bientôt 64 bits). Ces microcontrôleurs offrent de nombreux périphériques qui sont, pour la plupart, *mappés en mémoire*. Il s'agit donc de connaître le *plan mémoire physique* du microcontrôleur afin de pouvoir lire et écrire des valeurs dans les contrôleurs de périphériques.



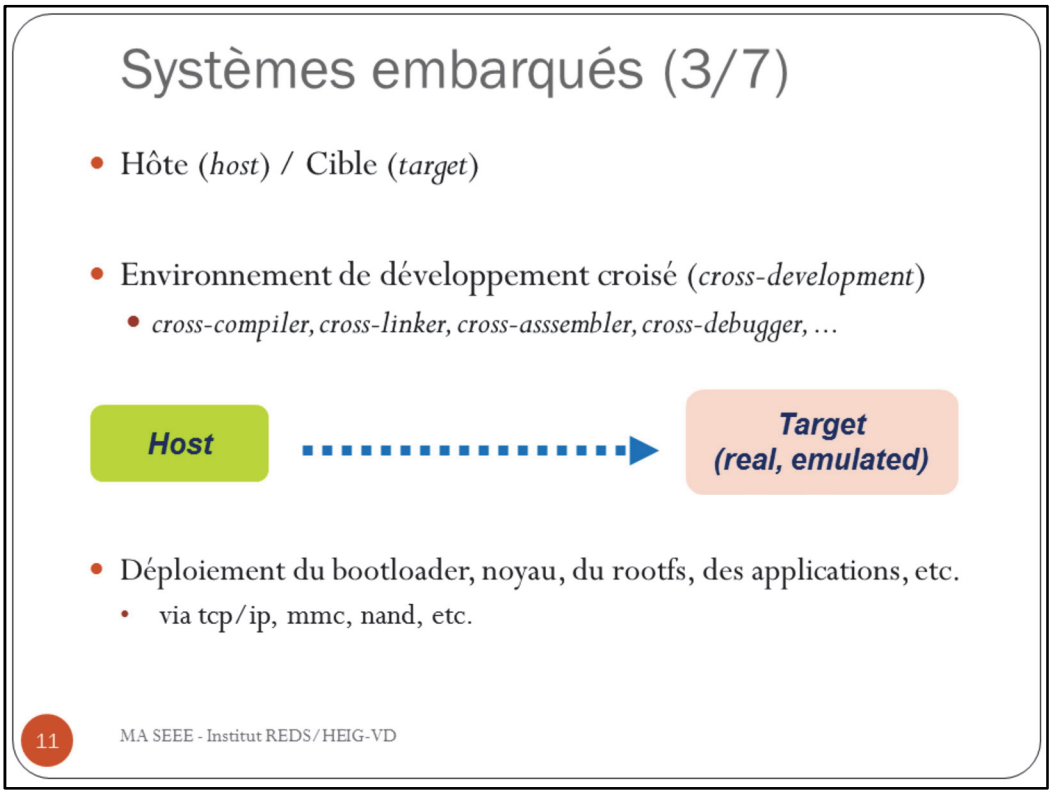
Un système embarqué comporte un ensemble d'applications dédiées composé éventuellement d'un système d'exploitation, voire d'un environnement virtualisé.

La matériel est constitué d'un microcontrôleur (processeur avec un ensemble de contrôleurs de périphériques) et des périphériques embarqués. Tout ce matériel peut être émulé de manière logicielle permettant ainsi de tester et de mettre au point les fonctionnalités du système **avant** de disposer de la *vraie* plate-forme.

Le *bootloader* (ou moniteur) permet l'amorçage du système. En principe, il permet de rechercher le système d'exploitation stocké dans une mémoire de stockage embarquée et de le démarrer.

Il permet également de donner un accès au matériel, au moyen d'une interface série (*UART)* typiquement, afin de pouvoir effectuer des tests, de transférer des applications et le système d'exploitation, de démarrer les applications (ou l'OS), etc.

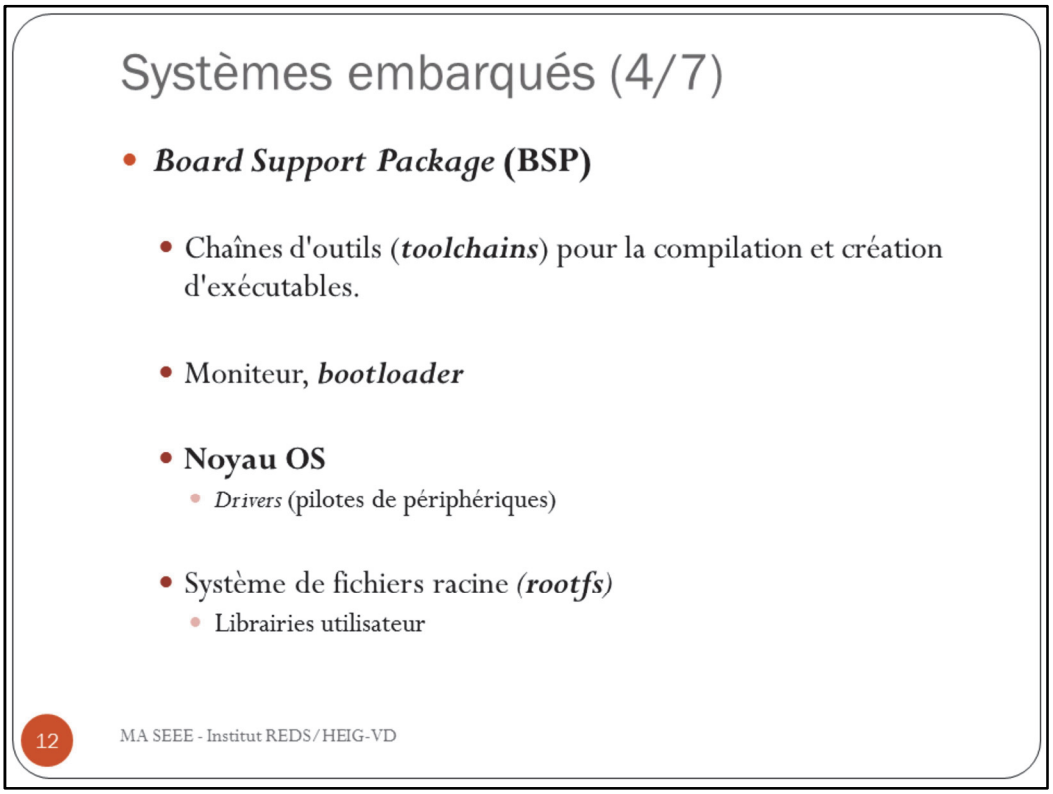
Les aspects liés à l'émulation, *drivers* et virtualisation seront traités dans les chapitres suivants.



Le développement d'applications embarqués commence sur un PC de développement appelé *machine hôte (host)*. L'environnement de développement croisé comprend un compilateur permettant de générer un binaire exécutable sur le processeur de la plate-forme embarquée, appelée plate-forme *cible (target)*.

Une fois les binaires générées sur la machine hôte, ceux-ci seront transférés sur la cible via un canal de communication classique (*tcp/ip*) utilisant l'interface *Ethernet*, ou via une mémoire de stockage (mémoire *flash* ou carte *SD/MMC*).

La mise au point d'une application peut s'effectuer grâce à un *debugger* croisé (*cross-debugger*) capable de se connecter à distance via *tcp/ip* selon une architecture client-serveur traditionnelle. Il est à noter que si l'environnement matériel est émulé, la technique de *debugging* est identique, la plate-forme cible étant local sur la machine hôte dans ce cas (adresse *tcp/ip* de type *localhost*).

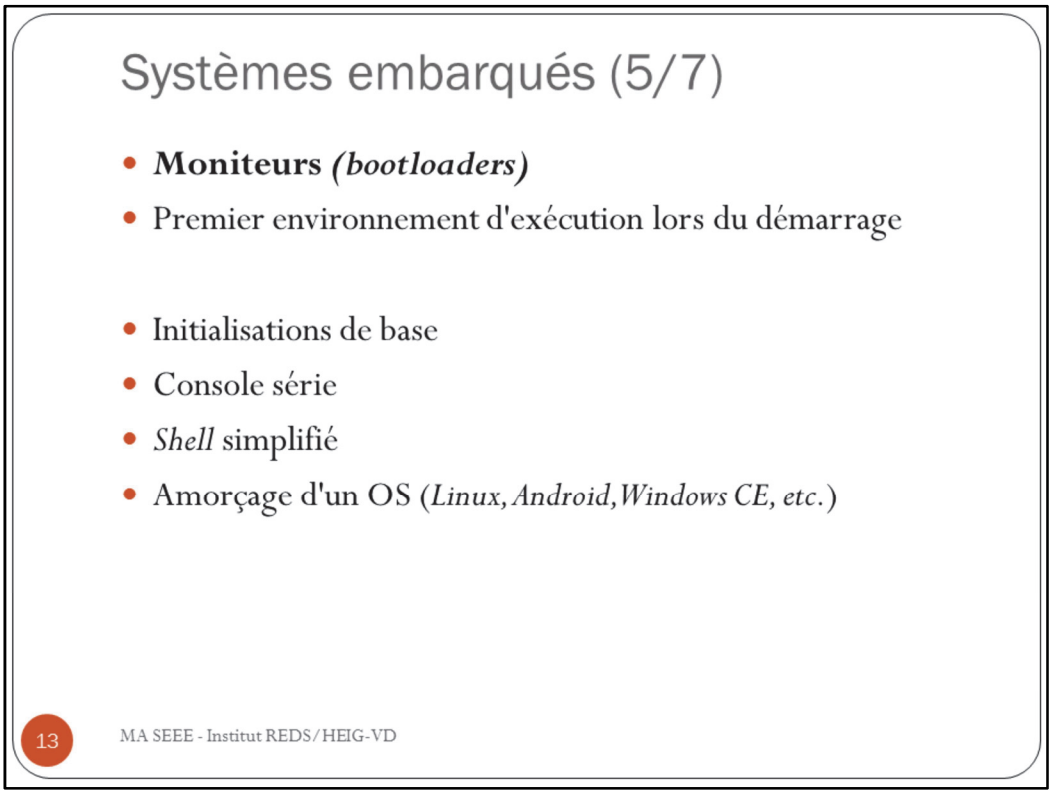


Le BSP *(Board Support Package*) contient les composants logiciels destinés à **développer** des applications embarquées destinées à être déployer sur une plate-forme cible.

Dans ce but, un *BSP* contient les outils nécessaires au développement croisé, à savoir une ou plusieurs chaînes d'outils (*toolchain*) comprenant le compilateur, *linker*, compilateur d'assemblage, *debugger*, etc. capables de traiter le binaire exécutable sur le processeur cible.

De plus, le *BSP* comprend éventuellement un ou plusieurs systèmes d'exploitation (*Linux, Android, Windows Embedded, etc.*) ainsi que les *drivers* des périphériques embarqués spécifiques à la plate-forme cible.

Finalement, le *BSP* comprend le système de fichiers racine contenant les fichiers de base et de configuration requis par le système d'exploitation ainsi que tout l'environnement applicatif inclus les *librairies*.

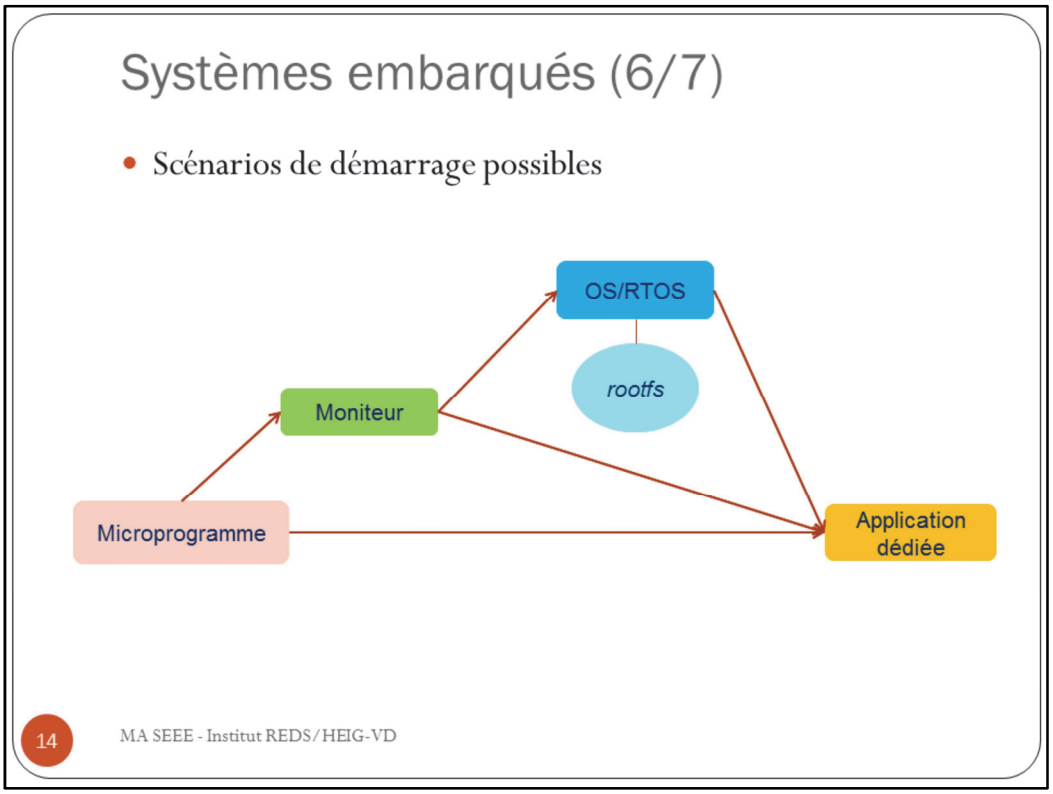


Le moniteur (ou *bootloader* ou encore chargeur d'amorçage) est le premier environnement d'exécution disponible sur un système embarqué.

Généralement, lors de la mise sous tension, un processeur ou microcontrôleur effectue une séquence de *bootstrap* matériel dont le code est implémenté sous forme de circuit matériel, et constitue le ***microprogramme*** initial. Ce microprogramme n'est généralement pas modifiable. Il peut se résumer à une instruction de saut à une adresse particulière, ou être plus conséquent dans le cas où il "cherche" à démarrer l'application suivante (moniteur ou autre) dans un espace de stockage défini ou via une connexion réseau par exemple. De plus ce microprogramme peut effectuer certaines vérifications liées à la sécurité d'utilisation (authentification).

Ce microprogramme donne ensuite la main au moniteur généralement sauvegardé en mémoire flash. Le moniteur travaille généralement avec un seul espace d'adressage physique, celui de la plate-forme. Il s'exécute généralement dans le mode superviseur (*kernel*) permettant ainsi d'accéder facilement le matériel. On peut donc aisément écrire/lire à n'importe quel emplacement mémoire, ce qui peut s'avérer très pratique dans les phases de mise au point et de tests.

Le moniteur peut ensuite démarrer une application ou directement l'OS. Il n'est en général pas possible de revenir sur le moniteur, celui-ci pouvant avoir été aussi effacé de la mémoire afin de récupérer la place qu'il occupait.

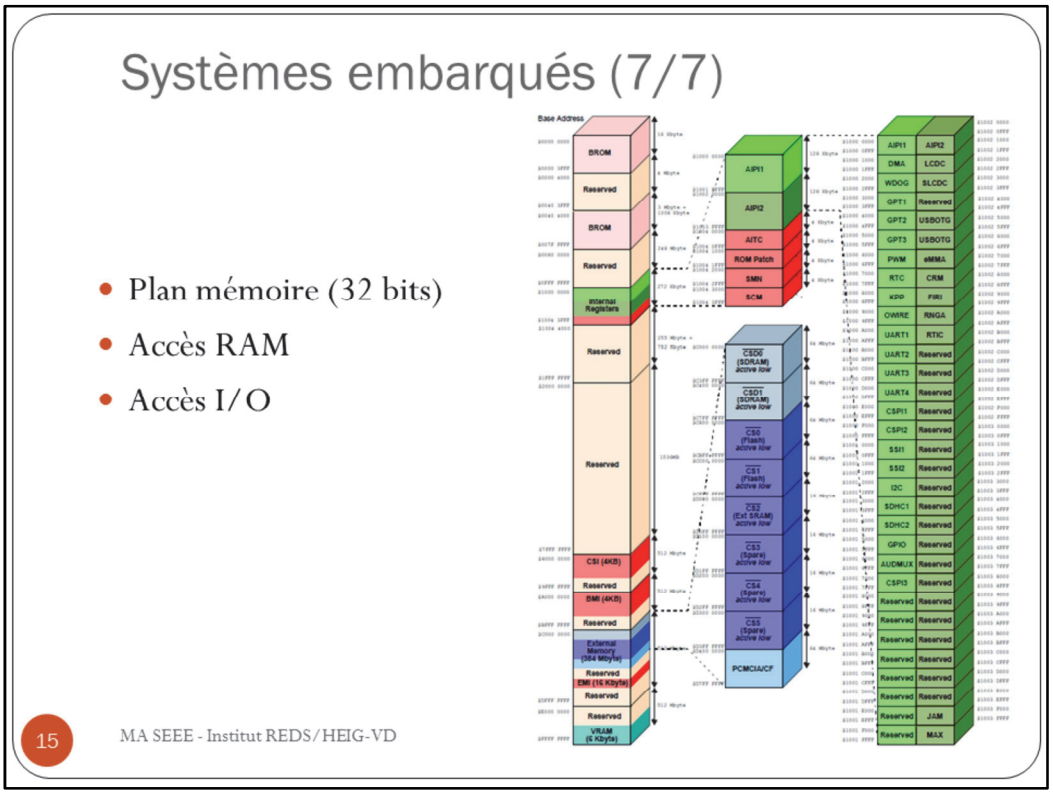


Les composants logiciels qui interviennent lors du démarrage d'une application embarquée sont multiples et différents scénarios peuvent exister.

Il est possible par exemple de démarrer une application *standalone* sans OS. L'application bénéficie des initialisations effectuées par le moniteur (mémoire, horloges, CPU, etc.).

Le moniteur peut transmettre à l'OS des informations relatives à la plate-forme sur les caractéristiques de la mémoire, des supports de stockage, du système de fichiers racine *(rootfs)* à monter, ou encore sur d'autres paramètres.

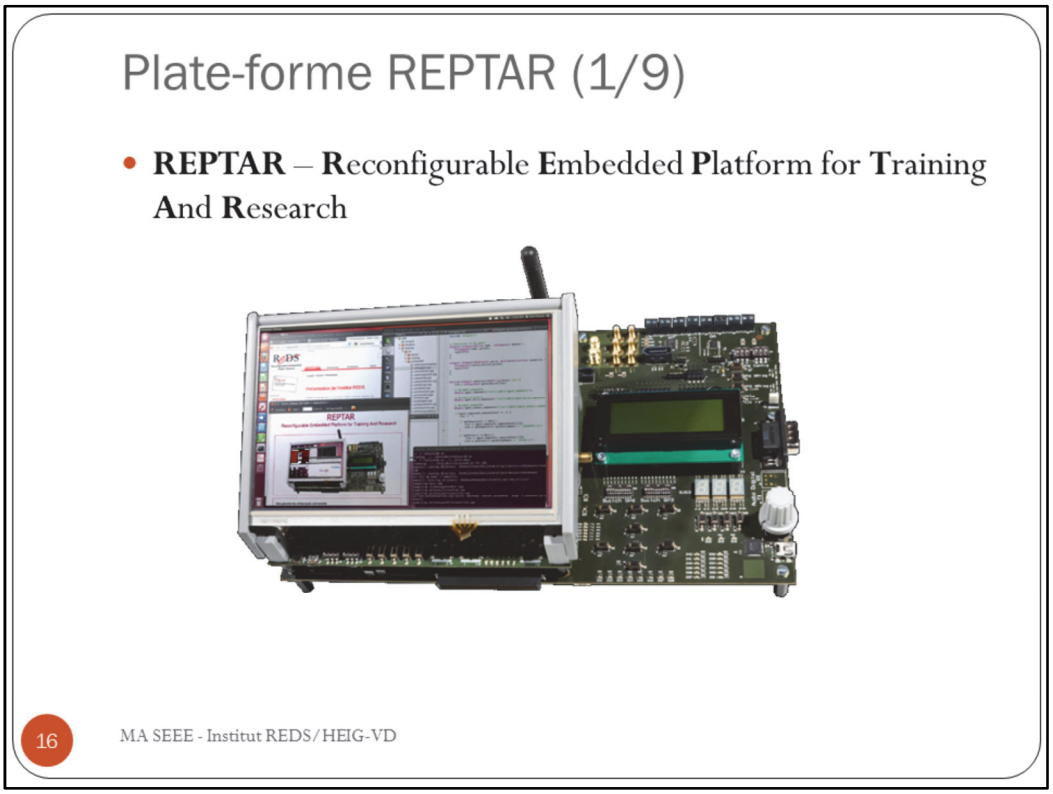
Le *rootfs* peut être stocké sur une mémoire flash ou sur une *sdcard*.



Comme le montre la figure ci-dessus, chaque périphérique dispose d'une région mémoire permettant l'accès aux registres d'entrée-sortie (I/O ainsi qu'aux différentes zones mémoire manipulées par les périphériques).

L'espace d'adressage physique dépend directement de la largeur de l'adresse. Une architecture 32 bits permettra ainsi d'adresser 2^{32} octets (*bytes*), soit environ 4 Go. Les microcontrôleurs n'ont généralement pas d'extension d'adressage physique, comme c'est le cas avec les processeurs Pentium (PAE, pour *Physical Address Extension*) ou AMD par exemple, leur permettant ainsi d'adresser plus que 4 Go. Il est à relever que cet espace d'adressage doit comprendre l'accès à la RAM elle-même.

Le plan mémoire dépend directement du processeur; par conséquent, le développeur devra adapter le *software* afin que celui-ci dispose des bonnes adresses pour accéder les bons périphériques. Cette adaptation est un exemple concret des différences existantes entre BSP (*Board Support Package*) des plates-formes embarquées.

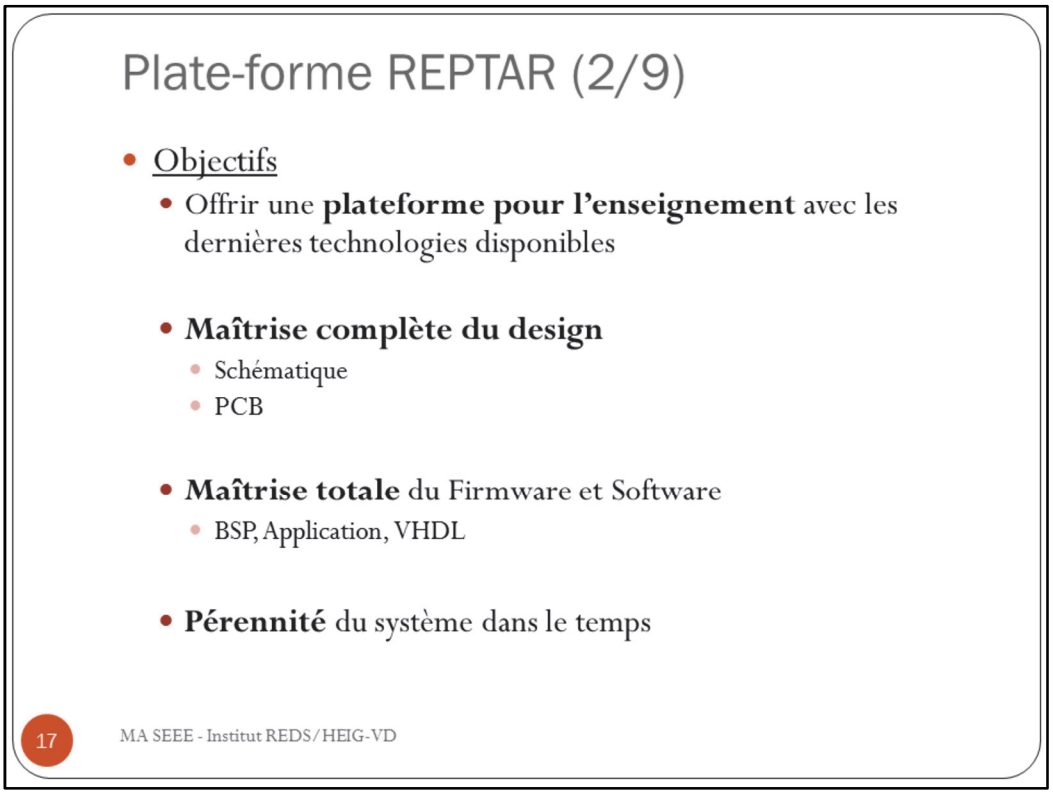


REPTAR est une plate-forme universelle destinée à l'enseignement et à la recherche (prototypes, démonstrateurs, etc.). Sa conception est modulaire et utilise des composants de dernières générations (processeur, périphériques de communication, écran tactile, etc.)

REPTAR est basé sur un système modulaire composé d'une **carte CPU** (*TI DM-3730 ARM* 32 bits à base d'un cœur *Cortex-A8* avec *DSP* inclus) ainsi qu'une **carte *FPGA*** (*Xilinx Spartan-3* et *Spartan-6*)

Les caractéristiques principales sont les suivantes:

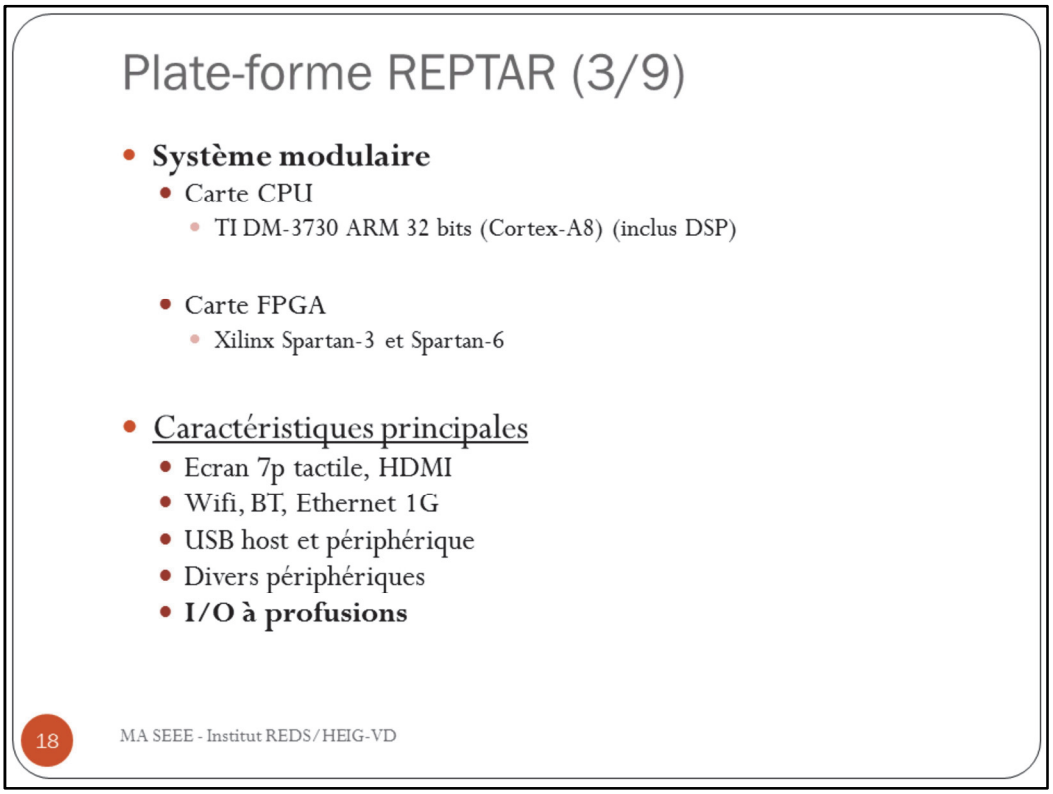
- Ecran 7p tactile, HDMI
- Wifi, BT, Ethernet 1G
- USB host et périphérique
- Divers périphériques
- Entrées-sorties (*I/O*) à profusion



REPTAR est basé sur un système modulaire composé d'une **carte CPU** (TI DM-3730 ARM 32 bits à base d'un cœur Cortex-A8 avec DSP inclus), ainsi qu'une **carte FPGA** (Xilinx Spartan-3 et Spartan-6)

Les caractéristiques principales sont les suivantes:

- Ecran 7p tactile, HDMI
- Wifi, BT, Ethernet 1G
- USB host et périphérique
- Divers périphériques
- I/O à profusion ...



Les principales fonctionnalités de la carte REPTAR sont les suivantes:

- GPIO multiples (*led*, boutons, *switch*, *buzzer*, *Encoder*)
- Convertisseurs AD/DA

- Liens hautes vitesses (*SATA*, *PCIe*)

- Connecteurs d'extensions (FMC, 80p)

- Display (tactile), *LCD*

- Stockage (*Flash*, *SD, DDR*)

- Communication (*Ethernet*, *Wifi*, *GSM, GPRS, BT, USB*)
- Système de positionnement (accéléromètre, *GPS*)
- Interfaces série (*SPI, I2C, UART, CAN*)

- Audio, Vidéo, (*Mic, Line, digital, hdmi*)

Plate-forme REPTAR (4/9)

- Architecture fonctionnelle
  - Modularité **matérielle** et **fonctionnelle**
    - Carte **FPGA autonome**
    - Carte **CPU autonome**

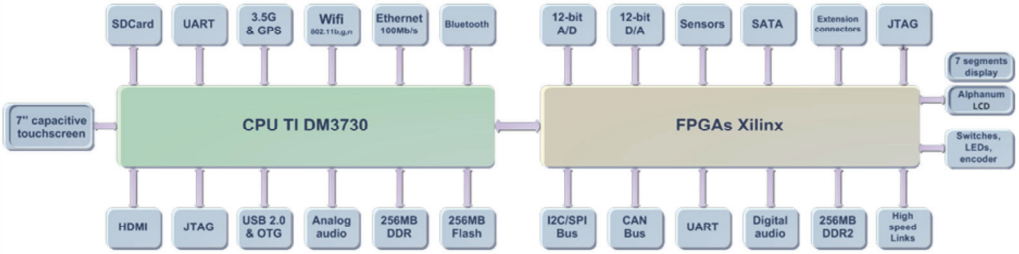
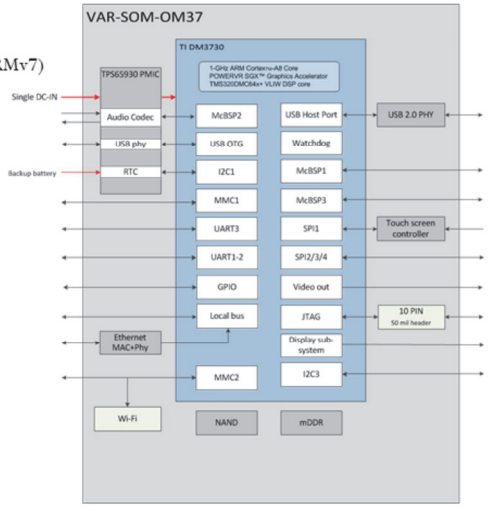


Plate-forme REPTAR (5/9)

- **Module CPU**
  - Module original: **Variscite** ***VAR-SOM-OM37***
  - Microcontrôleur **DM 3730** à base d'un **Cortex-A8** (ARMv7) (compatible OMAP 3)
  - Fréquence CPU de 1 GHz
  - **RAM** DDR 256 MB (64-512 MB)
  - Mémoire **flash** de 256 MB (256-512 MB)
  - Interface **SDcard**
  - Interface pour écran TFT / **Sortie HDMI**
  - Interface **écran tactile capacitif**
  - Interfaces **UART/I2C/SPI**
  - Interface **Ethernet 100Mbit**
  - DAC/ADC **16 bits** linéaire audio stereo
  - MIC et Line In & Out
  - Interfaces USB 2.0 Host + OTG
  - **Support pour JTAG**



Sous-système Mémoire

- Mémoire externe **DDR2 SDRAM** (connectée à la Spartan6)
  - 256 MB
  - 800 MHz

- Mémoire **flash** parallèle (sur module CPU)
  - 256 MB

- Mémoire **DDR SDRAM** (sur module CPU)
  - 256 MB
  - 400 MHz

- Interface **SDCard** présente sur le module CPU
  - Il est possible de démarrer le moniteur sur la *sdcard* (*switch* matériel)

Plate-forme REPTAR (6/9)

- **Carte CPU**

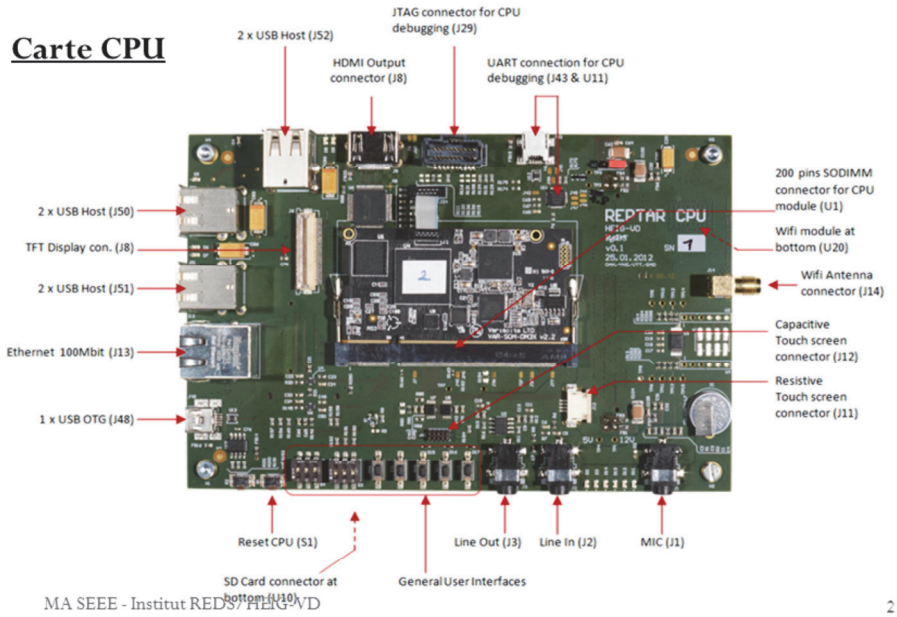
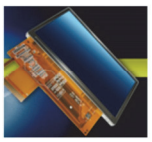


Plate-forme REPTAR (7/9)

- Dispositifs d'affichage
  - Ecran **tactile capacitif 7"**
    - 800 x 480
    - I2C pour la surface tactile
    - Possibilité de connecter un écran résistif à la place du capacitif



  - Affichage **LCD**
    - 4 x 20 lignes
  - Affichage **7-segments**

  - **HDMI**
    - Sortie simultanée sur écran tactile & HDMI

Plate-forme REPTAR (8/9)

- **Circuits programmables**
  - FPGA utilisée pour la **configuration**
  - Xilinx Spartan3 AN XC3S200AN-5FTG256C
    - Pilotage du chargement du bitstream dans la Spartan 6
    - Arbitreur de la chaîne JTAG

  - Xilinx PlatformFlash XCF32P
    - Stockage jusqu'à **4 bitstreams compressés** pour la Spartan 6

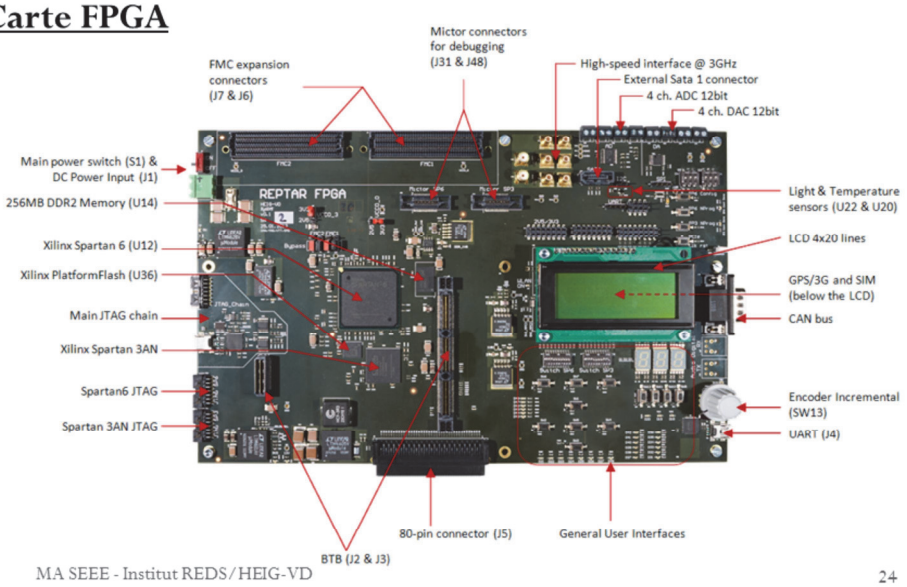
MA SEEE - Institut REDS/HEIG-VD 23

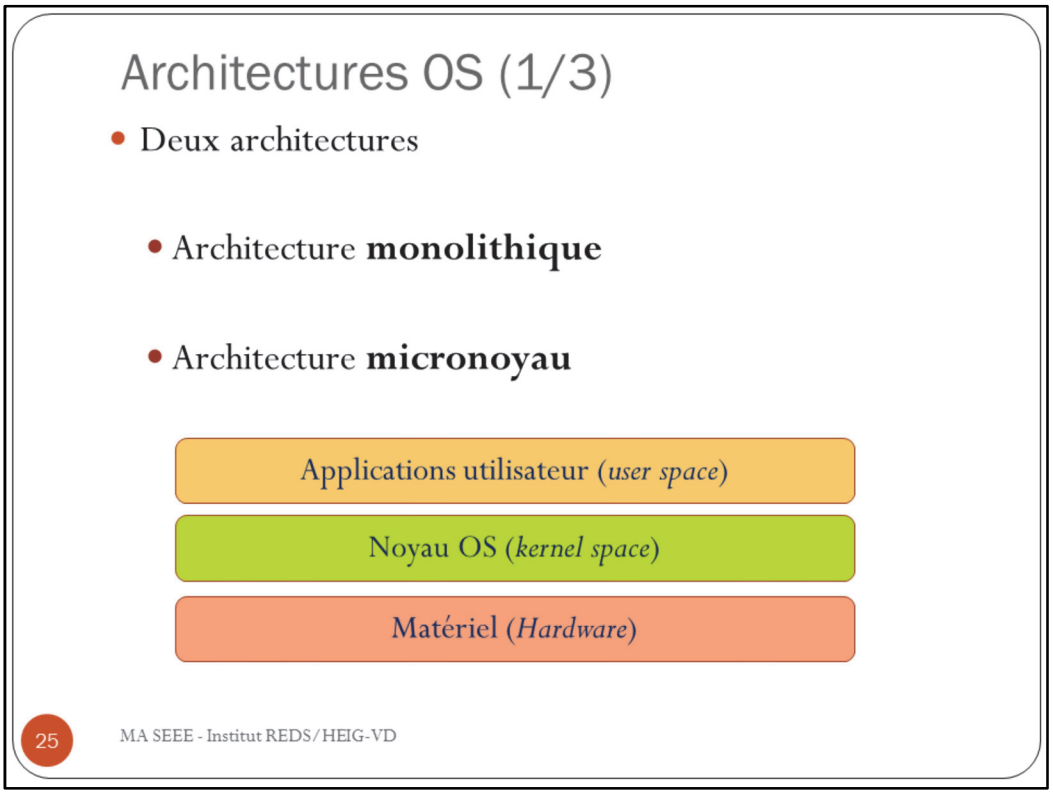
Circuits programmables

- FPGA Xilinx Spartan 6
- XC6SLX150TFGG900-3
- 147’443 éléments logiques
- 184’304 Flip-flop
- 1’335 Kbit de RAM distribuée
- 4’824 Kbit de blocs RAM
- 4 MCB
- 8 Transceivers (GTP) @ 3GHz
- Support pour JTAG

Plate-forme REPTAR (9/9)

- **Carte FPGA**

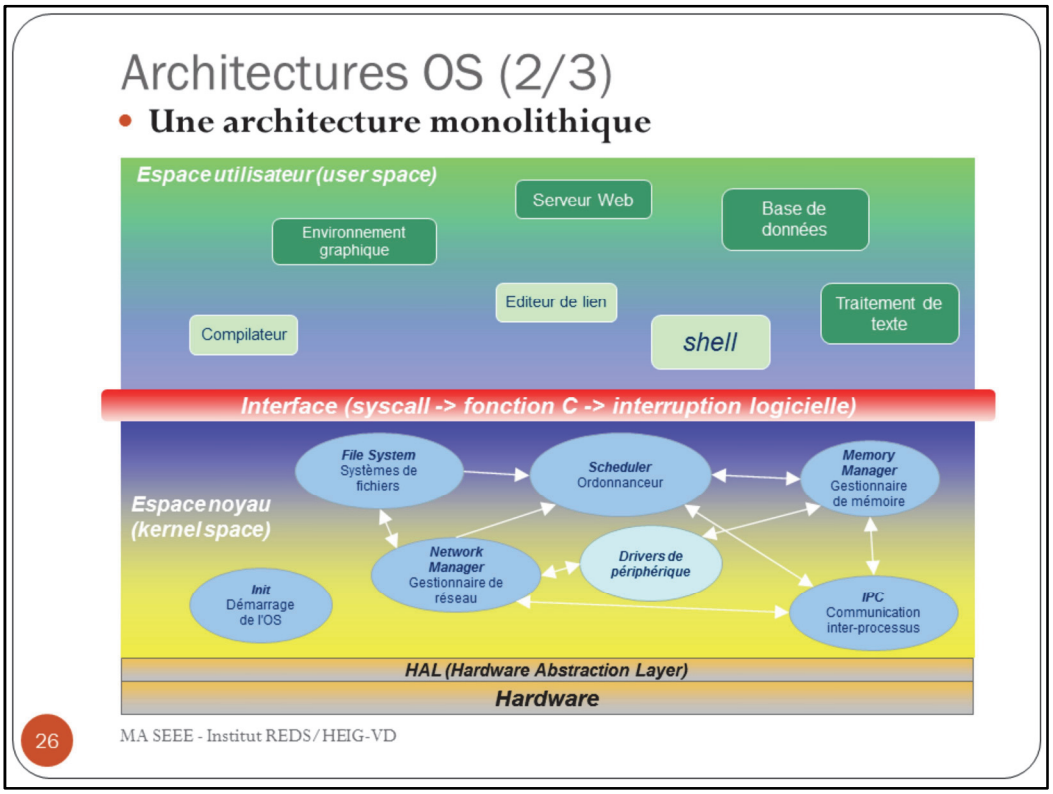




Un système d'exploitation (OS) peut se définir selon deux perspectives différentes: la perspective utilisateur, dans laquelle l'OS doit fournir une interface conviviale, performante et sécurisée aux applications utilisateur, et la perspective système, dans laquelle l'OS doit *piloter* le matériel en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

Dans l'approche monolithique, tous les sous-systèmes (fonctionnalités), y compris les pilotes de périphériques *(drivers),* sont implémentés dans le noyau de l'OS *(espace noyau).* Les applications et librairies (utilisateur et systèmes) ne font pas partie du noyau et résident dans *l'espace utilisateur*.

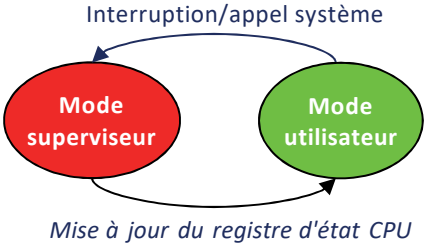
L'architecture micronoyau quant à elle consiste à placer un minimum de fonctionnalités dans le noyau et à placer le reste dans l'espace utilisateur au même niveau que les applications (cela est aussi valable pour les *drivers*). Avec cette approche, la communication entre les composants de l'espace utilisateur s'effectue au travers de mécanismes de type IPC (*Inter-Process Communication*) selon un modèle client-serveur bien établi. Le système est plus sûr et plus robuste, mais les performances sont moindres (plus d'interaction entre les deux espaces).

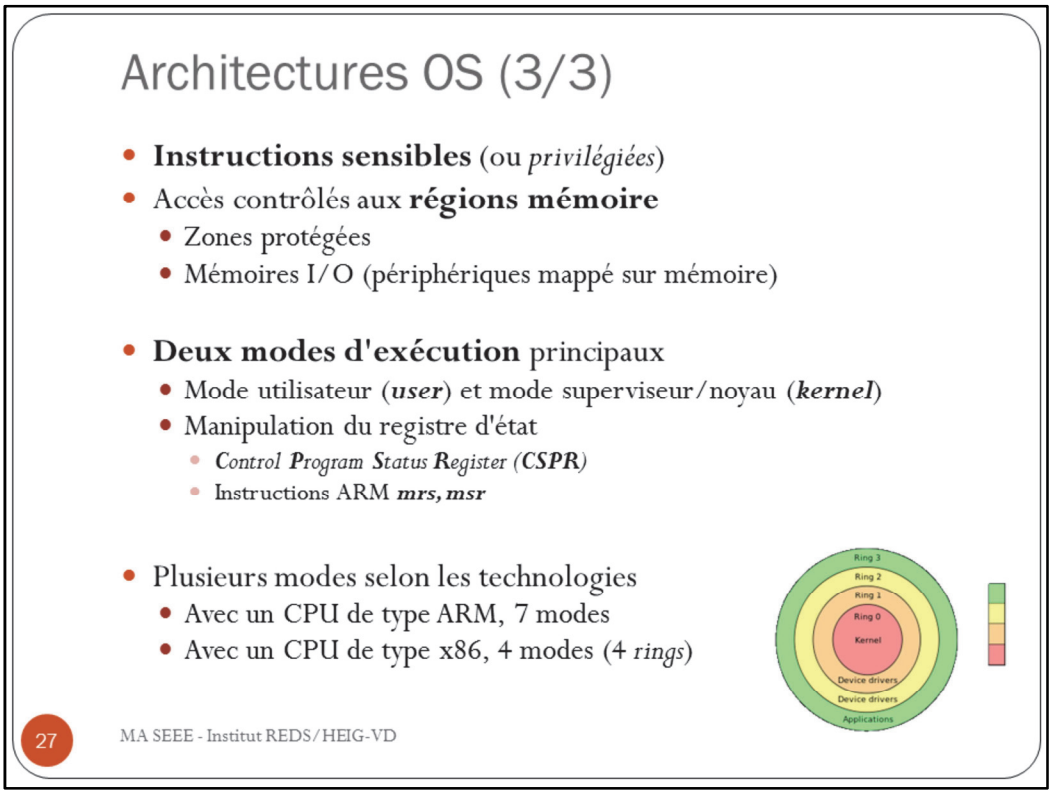


Pour des raisons de performance essentiellement, l'architecture monolithique est la plus utilisée. Elle reste très attractive au niveau de ses performances. Lors du développement noyau, l'accès aux structures noyau est relativement aisé, mais rend le système moins sûr (exploitation de failles de sécurité).

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 distinction entre espace utilisateur et espace noyau réside dans l'utilisation de modes d'exécution différents au niveau du processeur (CPU) lui-même. Le passage du mode noyau s'effectue en utilisant une instruction sensible dédiée (modification du registre d'état) alors que le passage du mode utilisateur au mode noyau se fera automatiquement lors d'une interruption (matérielle ou logicielle).



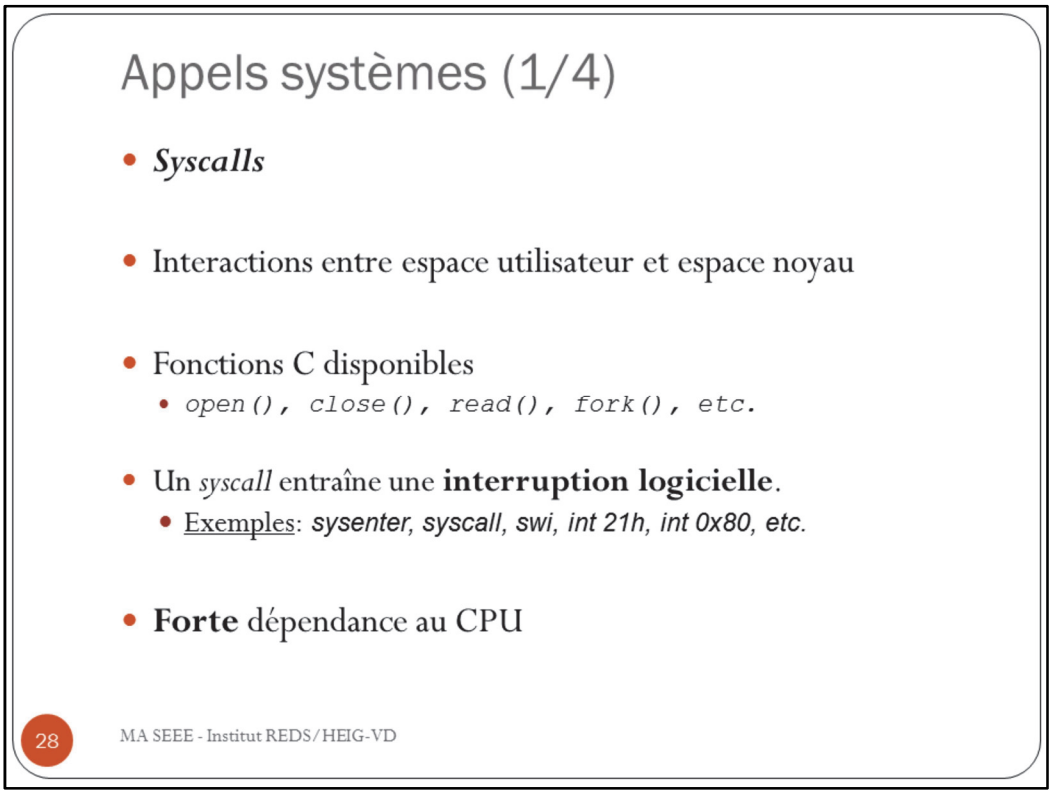


Les modes d'exécution utilisateur (*user*) et noyau (*kernel*) constituent l'un des deux mécanismes fondamentaux pour garantir la protection au niveau de l'exécution d'un OS, et donc d'une application. L'autre mécanisme étant la traduction dynamique d'adresse, mise en œuvre par la MMU comme il sera étudié plus tard dans ce cours.

Le mode utilisateur permet de concrétiser la notion **d'*espace utilisateur*** (ou ***user space***) dans lequel tourne les applications sur le système d'exploitation. Comme tout code s'exécutant sur un processeur, une application est donc une suite d'instructions machines (résultat de la compilation d'un code source). Par conséquent, en mode utilisateur, seul les instructions **non**-privilégiées pourront s'exécutées. Il n'est donc pas possible d'exécuter "n'importe quoi" sur un processeur à partir d'un code source.

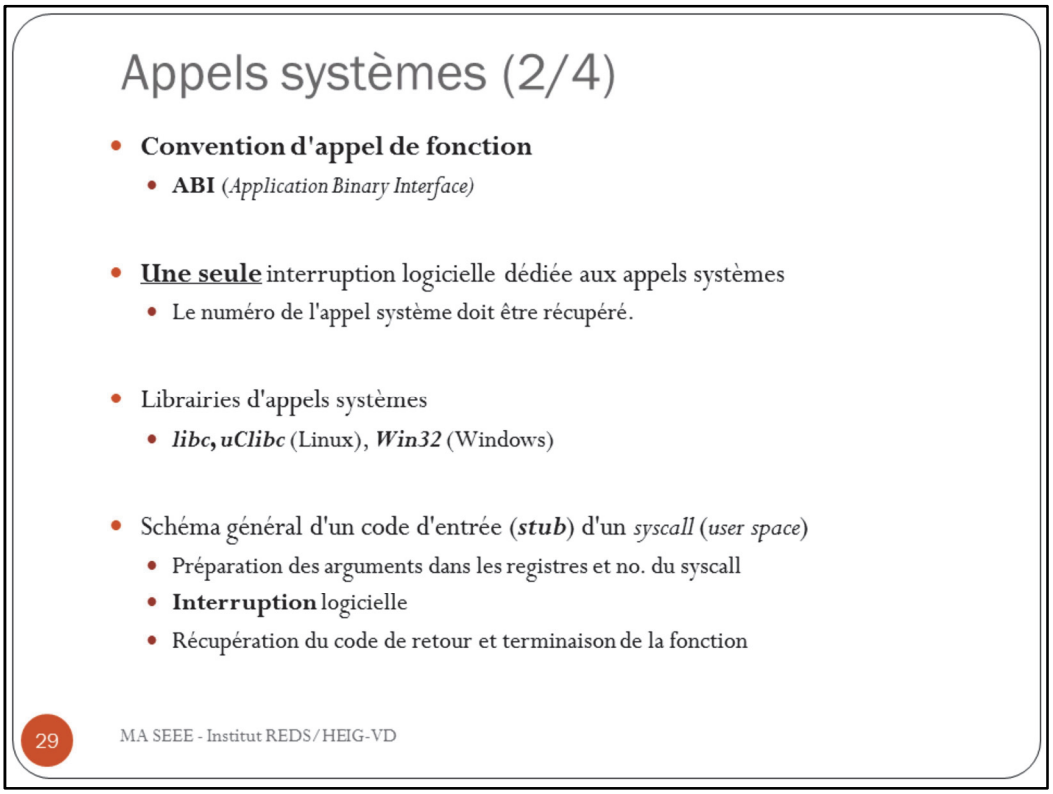
Le mode noyau, quant à lui, permettra l'exécution de ces instructions sensibles. Ce mode caractérise ainsi ***l'espace noyau*** (ou *kernel space*) dans lequel tourne le code du noyau de l'OS. Cette approche permet au code du noyau d'accéder à toutes les instructions, et par voie de conséquence à l'ensemble du matériel et de la mémoire.

Le mode d'exécution dans lequel le processeur se trouve à un certain temps est dicté par un ensemble de bits du registre d'état (PSR). Le passage du mode *user* au mode *kernel* n'est possible que sur l'arrivée d'une interruption matérielle ou l'exécution d'une interruption logicielle. En revanche, le passage inverse s'effectue en modifiant le registre d'état.



L'appel système (ou ***syscall***) est le mécanisme central permettant aux applications de l'espace utilisateur d'accéder aux différents services du noyau; l'implémentation des appels systèmes se trouvent donc dans l'espace noyau. Par conséquent, il est nécessaire que le processeur puisse exécuter des instructions sensibles dans le code noyau, ce qui nécessite un *changement de mode*. Cela intervient typiquement lors d'une interruption matérielle, il en sera de même pour **interruption logicielle**: cette dernière est caractérisé par l'exécution d'une instruction spécialement prévue à cet effet (sur ARM, il s'agit de l'instruction *SWI*).

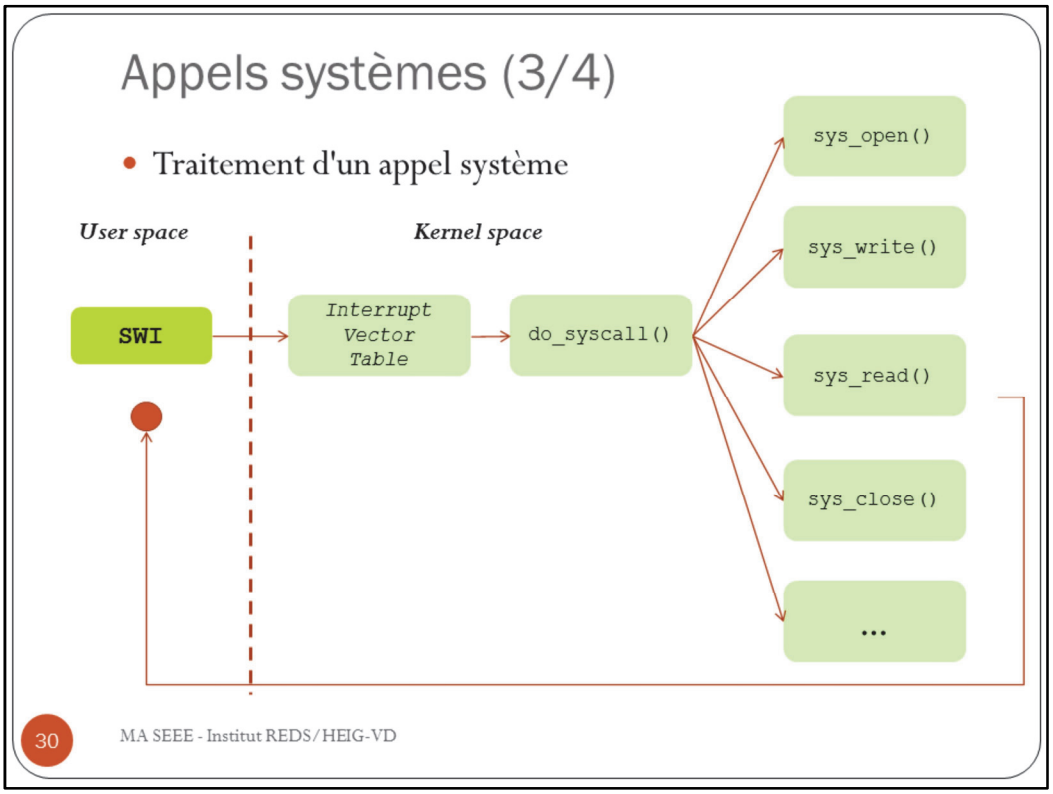
A la différence d'une interruption matérielle qui peut survenir n'importe quand, l'interruption logicielle sera provoquée par l'exécution d'une instruction; ce comportement donne un aspect déterministe à l'exécution du programme et l'on parlera ainsi *d'interruption synchrone*.



L'utilisation d'appel système exige des **conventions d'appel** de fonction entre l'espace utilisateur et l'espace noyau. Ces conventions constituent l'*ABI* (*Application Binary Interface*) définis au niveau de la *toolchain* et du noyau. l'*ABI* détermine quels registres sont réservés pour stocker les informations relatives à l'appel système, à savoir le **numéro** du *syscall,* les **arguments**, la **valeur de retour**.

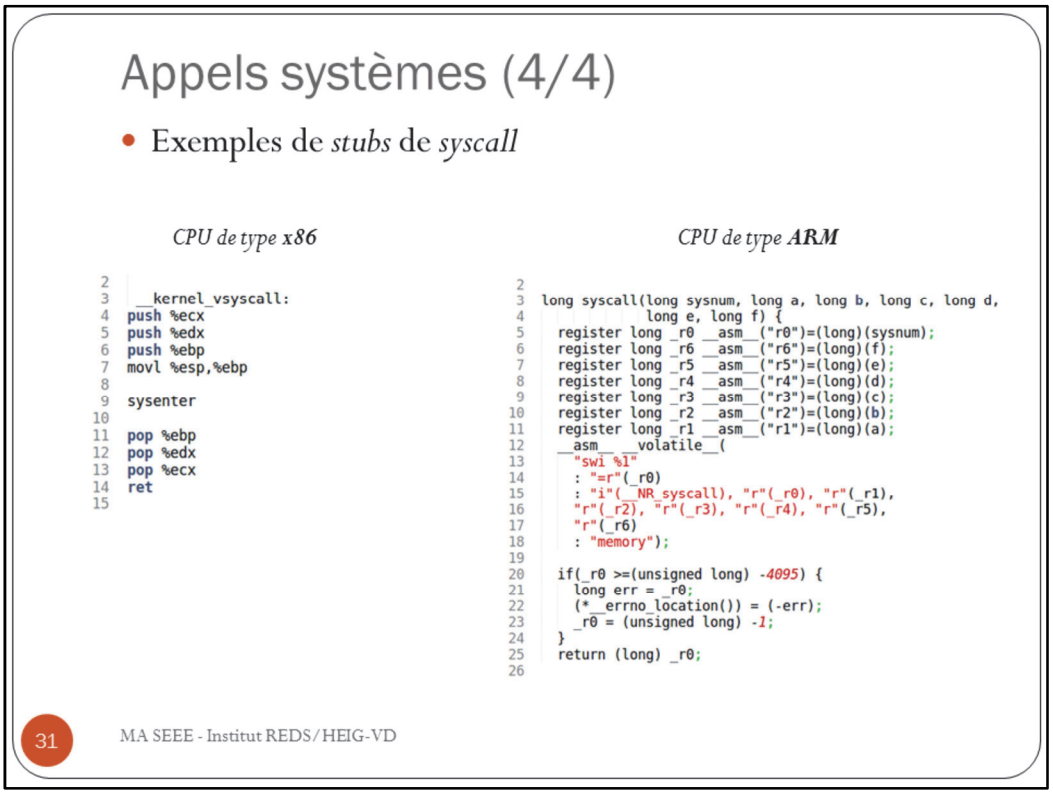
Les conventions dépendent de l'architecture matérielle et de l'OS et sont souvent une source d'incompatibilité lors de portage d'applications. C'est pourquoi, les appels systèmes sont généralement spécifiés selon une norme bien établie (typiquement la norme POSIX), et sont associés à une librairie contenant l'implémentation au niveau de l'espace utilisateur.

Le code d'un appel système côté utilisateur est sensiblement le même pour l'ensemble des *syscalls*. Seul le numéro de l'appel système change.



Le schéma ci-dessus illustre le fonctionnement d'un appel système implémenté dans un OS. L'instruction ARM *swi* provoque le basculement du processeur en mode noyau et branche l'exécution sur la fonction associée au vecteur d'interruption correspondant. Ce mécanisme est similaire à celui mis en place lors du traitement d'une interruption matérielle.

Il est à noter que le code d'un appel système au niveau noyau s'exécute dans le contexte du processus et du *thread* en cours d'exécution. C'est un peu comme si le code "appartenait" au processus, sauf qu'il est exécuté dans l'espace noyau. Des changements de contexte au niveau *thread* peuvent intervenir durant l'exécution d'un *syscall,* l'exécution de celui-ci sera donc suspendu jusqu'au réveil du *thread*.

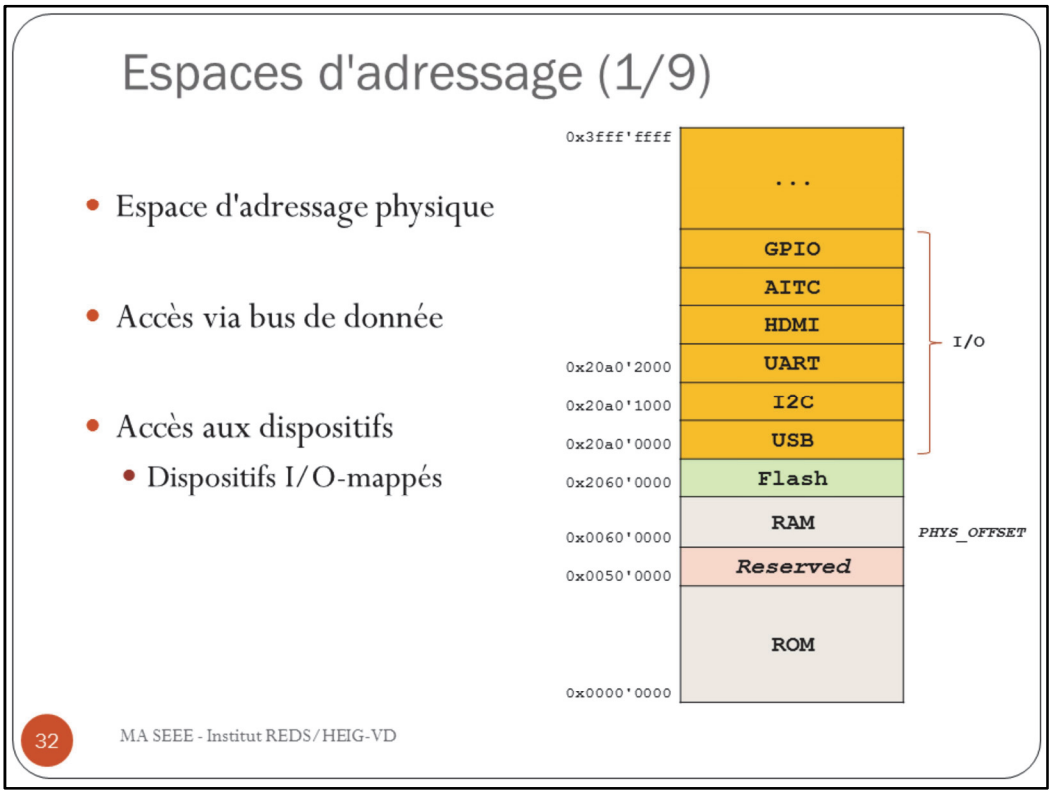


Les deux exemples ci-dessus correspondent au *stub* (code générique) associé à l'appel système exécuté dans l'espace utilisateur.

Les arguments de l'appel système sont placés soit dans les registres (plus rapide) soit sur la pile utilisateur.

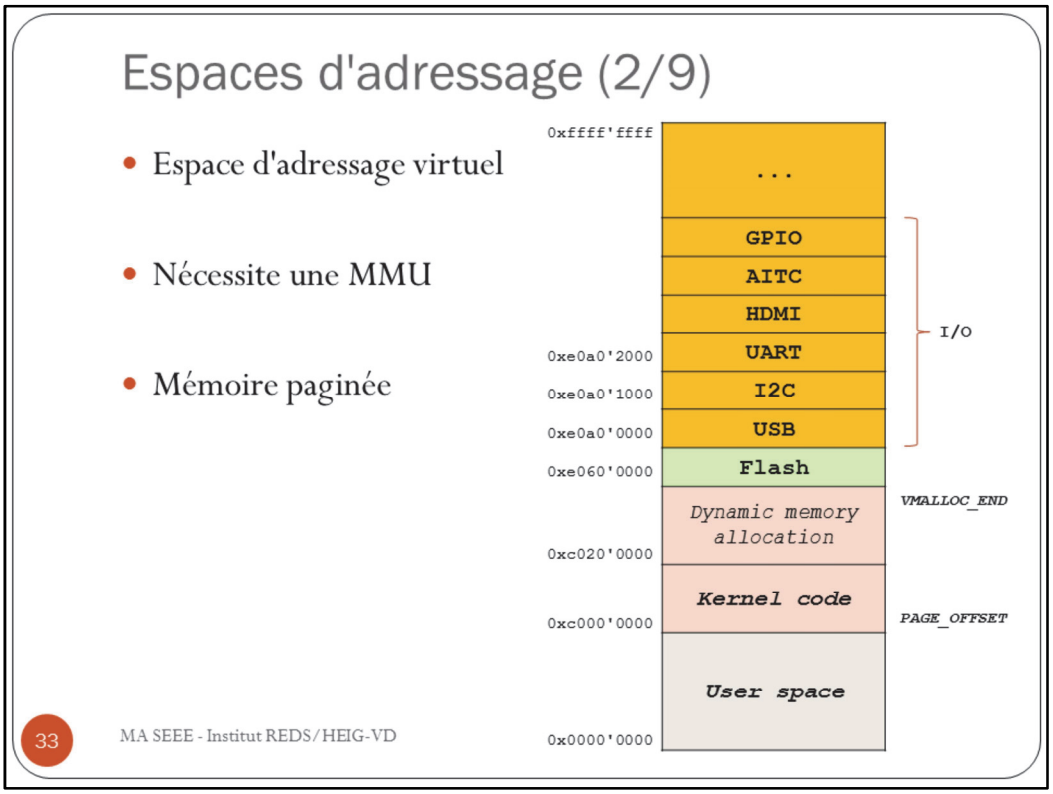
Etant donné que le code (utilisateur) d'un *syscall* nécessite l'utilisation d'une instruction particulière pour l'interruption logicielle, il est généralement écrit en assembleur. La version *ARM* montre la possibilité du langage C à "supporter" du code assembleur de bas niveau avec l'utilisation de directive particulière (`__asm__ { … }`), alors que la version *x86* sera compilé directement à partir de l'assembleur (le code objet issu d'un code C peut être *linké* avec un code objet issu d'un code assembleur.

En outre, la variable globale ***errno*** est particulière; visible et déclarée "externe" dans les entêtes de fichier de la librairie standard *(stdlib.h, stdio.h, etc.*), elle permet de récupérer un code d'erreur après l'exécution d'un appel système. Ainsi, en cas d'erreur, l'appel système retourne typiquement une valeur négatif, et *errno* contient le code de l'erreur associée. Elle renseigne sur les détails de l'erreur et peut être utilisée directement avec la fonction *perror("")* permettant l'affichage d'un texte prédéfini associé au code d'erreur correspondant. Pour plus de détails à ce sujet, les pages *man* fournissent l'aide nécessaire.



L'espace d'adressage physique contient l'ensemble des données qui peuvent être accédées par le processeur en lecture et/ou écriture. Une adresse permet d'accéder individuellement chaque octet (*byte*) de la mémoire. Dans ce contexte, l'espace mémoire ne se réfère pas exclusivement à la mémoire RAM (ou ROM), mais également aux périphériques (contrôleurs de périphériques, mémoire *flash*, mémoire graphique, etc.)

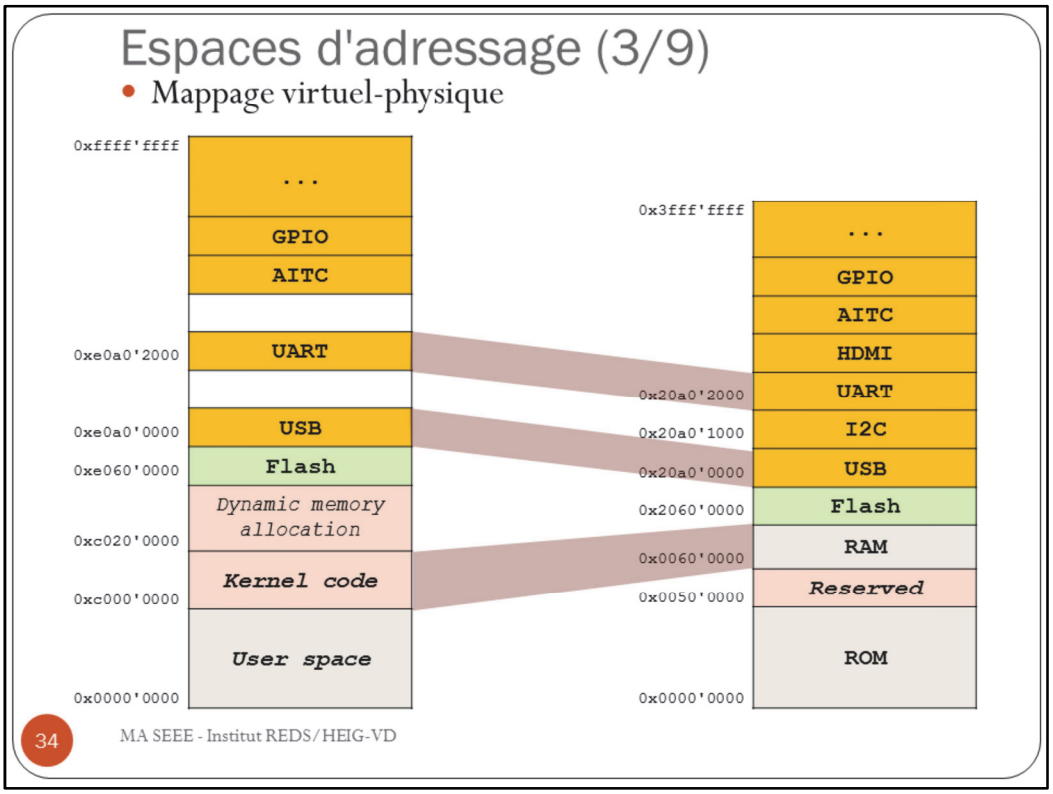
Dans un système embarqué, le plan mémoire (*memory layout*) permet de connaître les différentes plages d'adresses relatives aux différents périphériques, y compris la RAM. Sur une architecture 32 bits, les adresses sont également sur 32 bits puisque les transferts CPU<->mémoire se font à l'aide de registres de cette taille.



Les processeurs (et microcontrôleurs) 32 bits sont quasiment tous équipés d'une MMU (*Memory Management Unit*). Cette unité matérielle permet de traduire une adresse virtuelle vers une adresse physique grâce à l'utilisation d'une table de correspondance appelée ***table de pages***. Pour des raisons d'efficacité, la table de pages ne contient pas une équivalence adresse virtuelle → adresse physique, mais fait correspondre une page mémoire virtuelle vers une page physique: une page est une zone de plusieurs octets contigus; **sa taille est généralement de 4 Ko**.

En réalité, pour des raisons d'optimisation, une traduction de page ne s'effectue pas à l'aide d'une seule table de pages, mais de deux tables (pagination à deux niveaux), voire de trois ou quatre sur des architectures 64 bits. Chaque entrée de table (*PTE* ou *Page Table Entry*) – ou ligne de la table – contient au minimum un numéro de page physique correspondant au numéro de page virtuelle représenté par son index (cf plus loin dans ce chapitre).

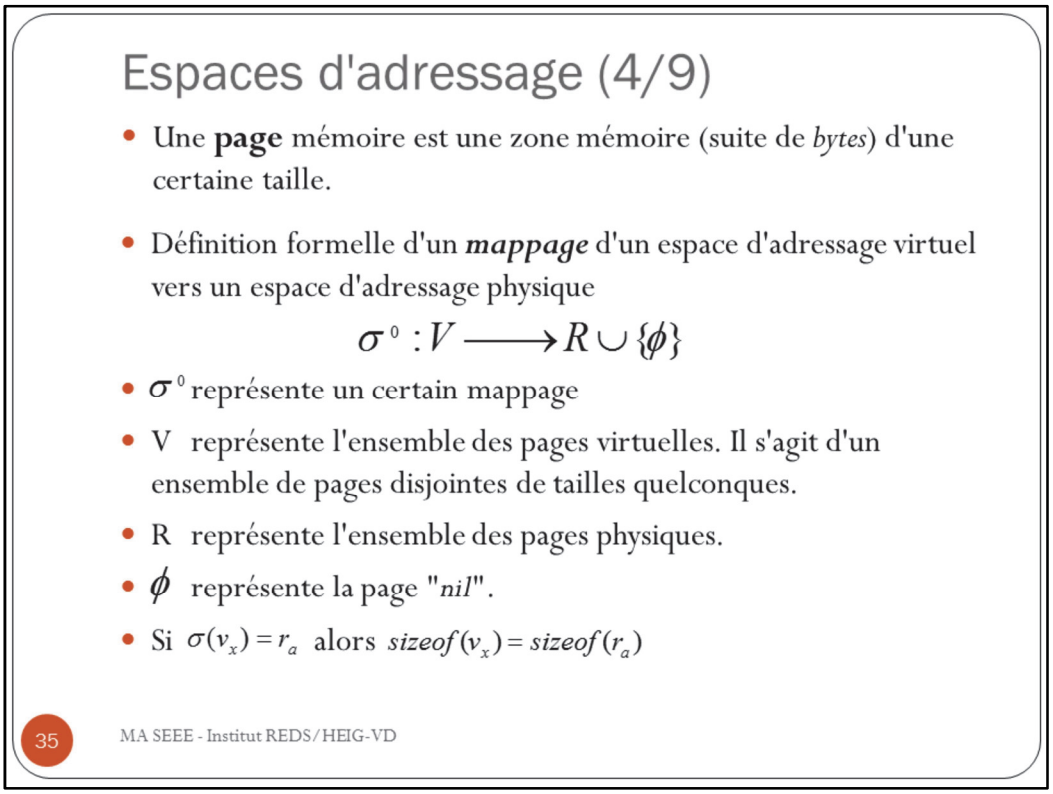
Avec une pagination à deux niveaux, l'adresse virtuelle est décomposée en trois parties: l'index de niveau 1 (bits de poids fort la MMU doit connaître l'emplacement de la table de premier niveau afin qu'elle puisse ensuite récupérer la PTE correspondante
quelle table de premier niveau connaît l'adresse physique de la table de 1er niveau, puis la table de second niveau contient les PTEs indiquant la page physique contenant la table de second niveau. Cette dernière contient les PTEs indiquant la page physique contenant la donnée référencée.



Un mappage entre espace d'adressage virtuel et physique peut se faire à différentes granularités. S'il est vrai que les pages de l'espace utilisateur font toujours 4 Ko, le mappage de zones mémoire dans l'espace noyau peut s'effectuer avec des tailles plus grandes (1 Mo par exemple). Typiquement, le code exécutable du noyau ne va pas changer de taille et sera toujours chargé en RAM. Il est donc possible de le mapper avec des pages de 1 Mo.

Le mappage d'un espace virtuel peut changer au fil du temps, typiquement lors de changement de contexte mémoire (donc de processus). C'est pourquoi, lors d'une phase de développement et plus particulièrement lors de mise au point, il faut s'interroger sur l'espace virtuel en cours d'utilisation: celui-ci peut varier rapidement.

Un changement de contexte mémoire consiste à reconfigurer la MMU et à lui attribuer une table de page de premier niveau différente. Une fois la MMU reconfigurée, l'espace d'adressage peut changer radicalement et une mauvaise compréhension de ce mécanisme peut conduire rapidement à des plantées liées à la manipulation de mauvaises adresses.



Le formalisme utilisé pour définir la construction d'un espace d'adressage s'inspire des travaux de *Jochen Liedtke* (cf bibliographie).

Un mappage de pages virtuelles vers des pages physiques se définit par la fonction σ. Dans notre cas, l'ensemble V représente l'ensemble de toutes les pages virtuelles supportées par une architecture donnée (typiquement 32 bits, soit $\frac{2^{32}}{2^{12}}$ pages pour une taille de page de 4 Ko).

La définition formelle de σ conduit aux propriétés suivantes:

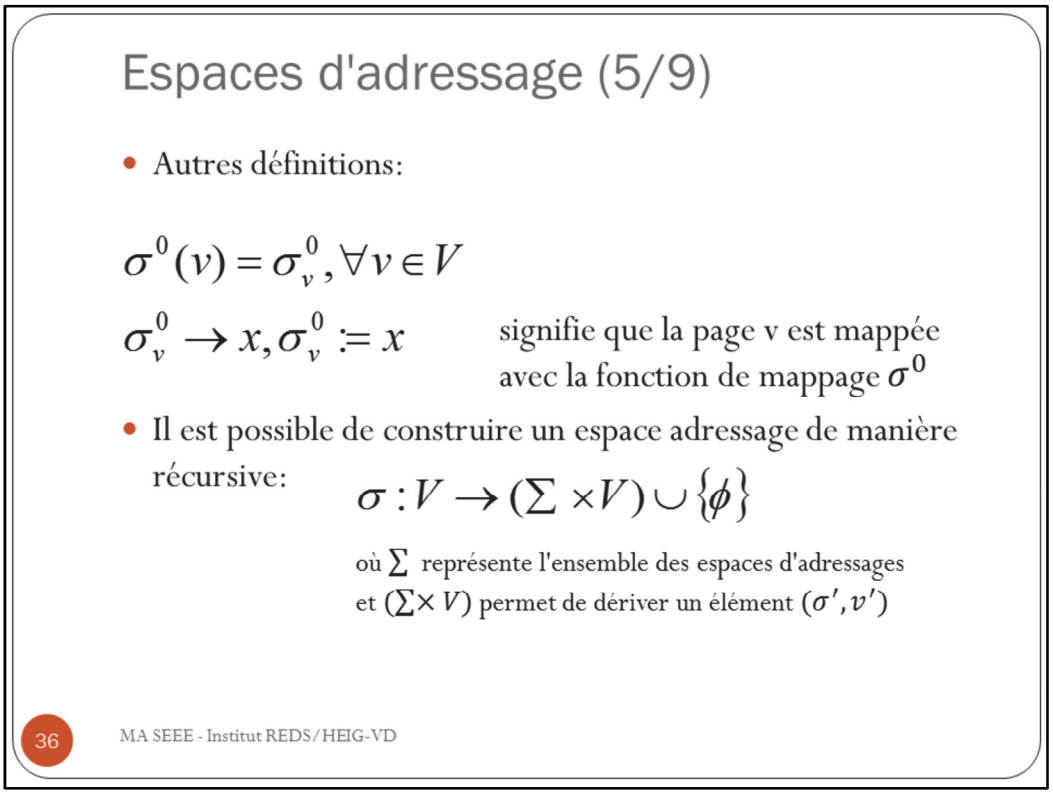
1) Toutes les pages virtuelles ne sont pas mappées vers une page physique (mappage sur ∅). Dans ce cas, on dit qu'il n'existe pas de mappage pour de telles pages virtuelles.

2) Plusieurs pages virtuelles peuvent être mappées vers une même page physique, par exemple $\sigma(v_x) = \sigma(v_y) = r_a$

3) Sur un même ensemble V, il peut exister plusieurs fonctions σ représentant ainsi des espaces d'adressage différents:

$$\sigma_m(v_x) = r_a$$
$$\sigma_m(v_y) = r_b$$
$$\sigma_n(v_x) = r_c$$
$$\sigma_n(v_y) = r_b$$

Par conséquent, avec deux mappages différents, $r_a \neq r_b$ pour la page v_x alors qu'il est identique pour v_y !!



Dans la pratique, la dernière observation met bien en évidence qu'il est possible de préserver un mappage lors d'un changement de contexte, ce qui est très utile pour le partage de code (cas des librairies partagées) et bien entendu le code noyau, commun à tous les processus.

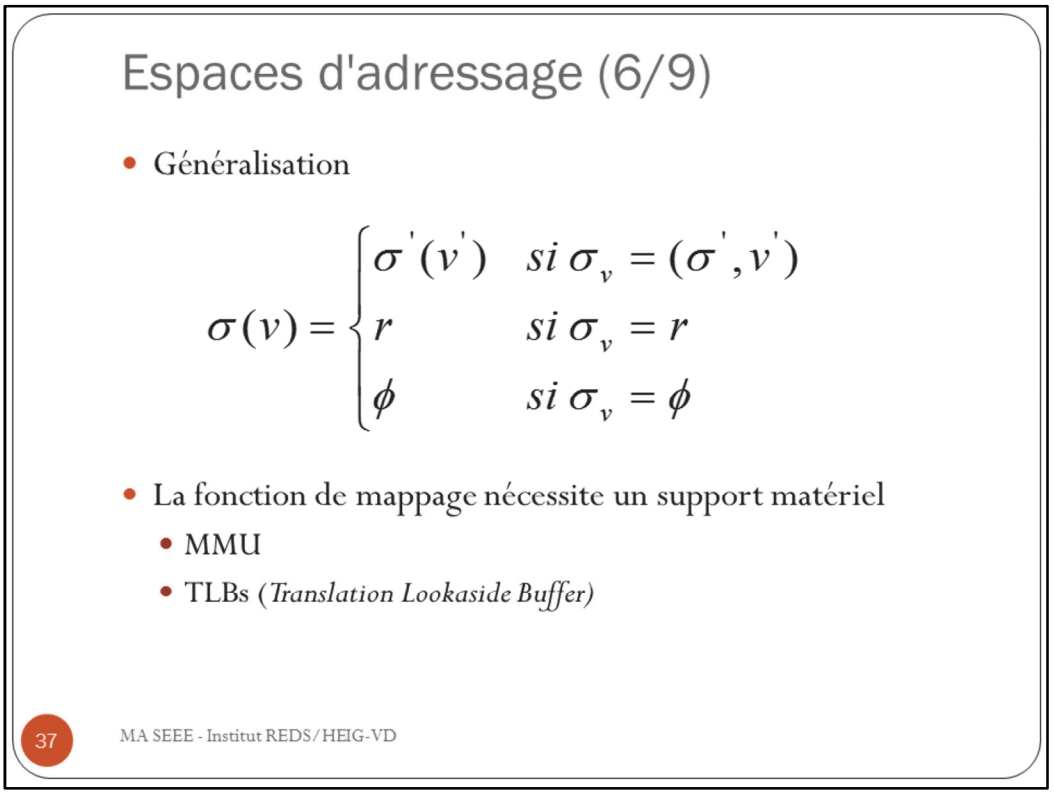
La définition de la fonction de mappage σ permet également de construire des espaces d'adressages de manière récursive. En effet, si la fonction de mappage effectue une traduction vers des pages physiques, elle peut également le faire sur un ensemble de pages virtuelles. Par exemple, si $\forall v \in V, \sigma^0(v) = r, r \in R$, on peut construire une nouvelle fonction σ^1 où $\forall v^1 \in V^1, \sigma^1(v^1) = v, v \in V$.
Dès lors, on a $\sigma^0(\sigma^1(v^1)) = r, avec\ r \in R$.

Ce mécanisme récursif est très puissant et permet de créer des espaces sécurisés qui seront étudiés dans le chapitre de la virtualisation.

Par ailleurs, on observe qu'il serait théoriquement possible de construire récursivement un mappage avec plusieurs fonctions de mappages sous-jacentes. Par exemple:

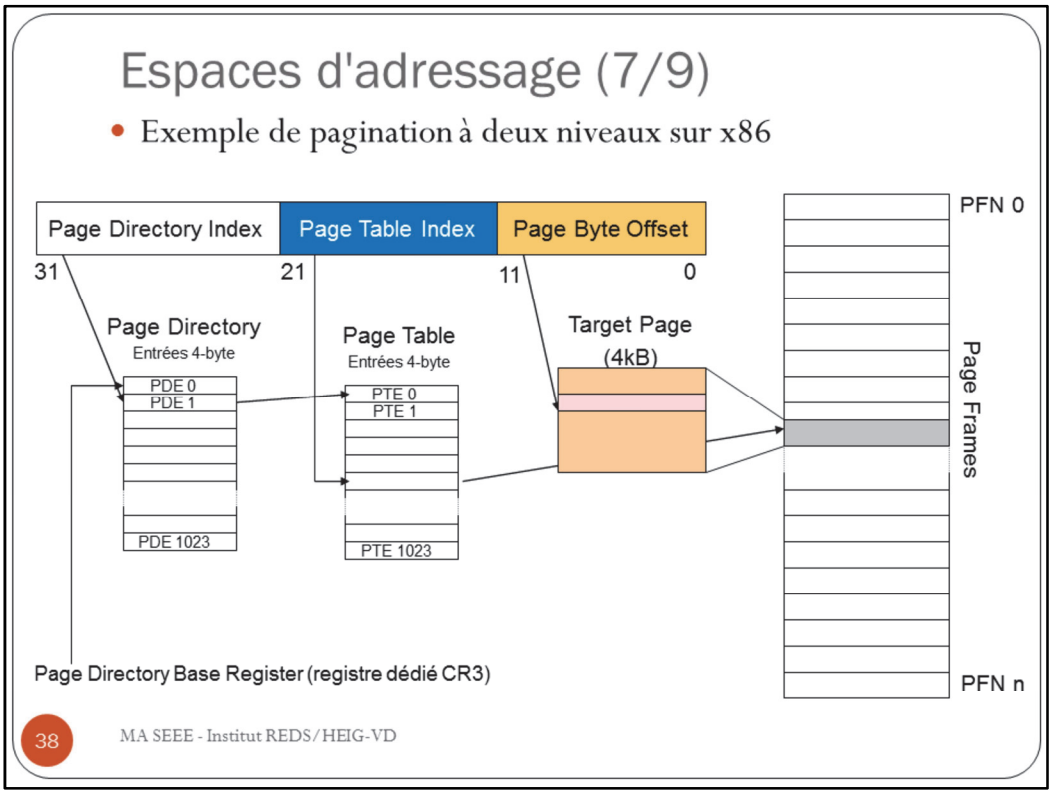
$$\sigma^2(v_x^2) = (\sigma^1, v^1) \text{ et } \sigma^2(v_y^2) = (\sigma^0, v^0)$$

Ce cas est également utile dans le contexte de la virtualisation, mais peut être difficile à gérer au niveau matériel.



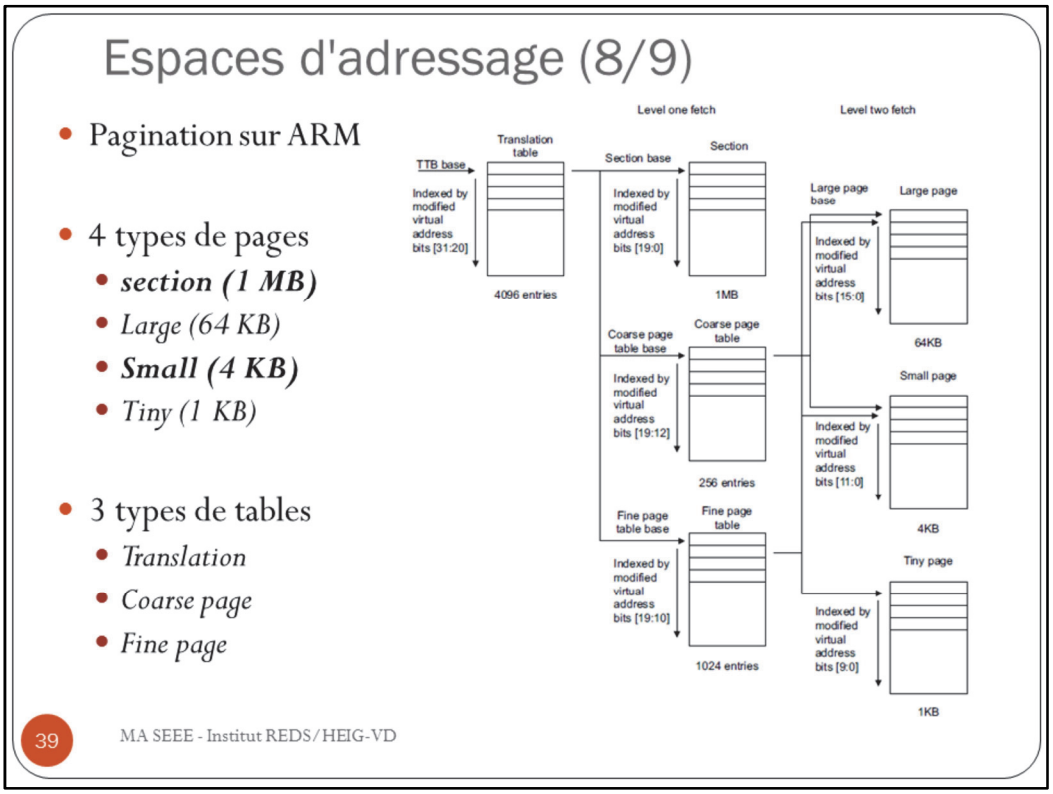
La généralisation de la fonction de mappage est définie ci-dessus. On remarque la possibilité de ne "rien" mapper à tous les niveaux des espaces d'adressage.

La plupart des microcontrôleurs 32 bits supporte un niveau de mappage (page virtuelles, pages physiques). Il est toutefois possible d'implémenter de manière logicielle des fonctions de mappages récursives, mais cela demande de fréquents ré-ajustage au niveau des tables de pages et peut impacter de manière significative sur les performances du système. Les prochaines génération de microcontrôleurs permettront de supporter un niveau supplémentaire de traduction grâce aux architectures de virtualisation à l'instar du cœur ARM Cortex-A15.



Avec une pagination à deux niveaux, l'adresse virtuelle est décomposée en trois parties: l'index de niveau 1 (ou *Page Directory Index* dans l'exemple), l'index de niveau 2 (ou *Page Table Index* dans l'exemple), ainsi que l'*offset* (*ou Page Byte Offset* dans l'exemple).

La MMU connait l'emplacement de la table de premier niveau (*Page Directory* dans l'exemple) grâce à l'adresse physique de son emplacement. Elle peut donc extraire la PTE correspondante au premier niveau. Cette dernière contient le numéro de page physique de la table de second niveau (*Page Table* dans l'exemple). L'index de second niveau permet de récupérer ainsi la PTE correspondante qui contiendra le numéro de page physique *(Target Page)* de la donnée.

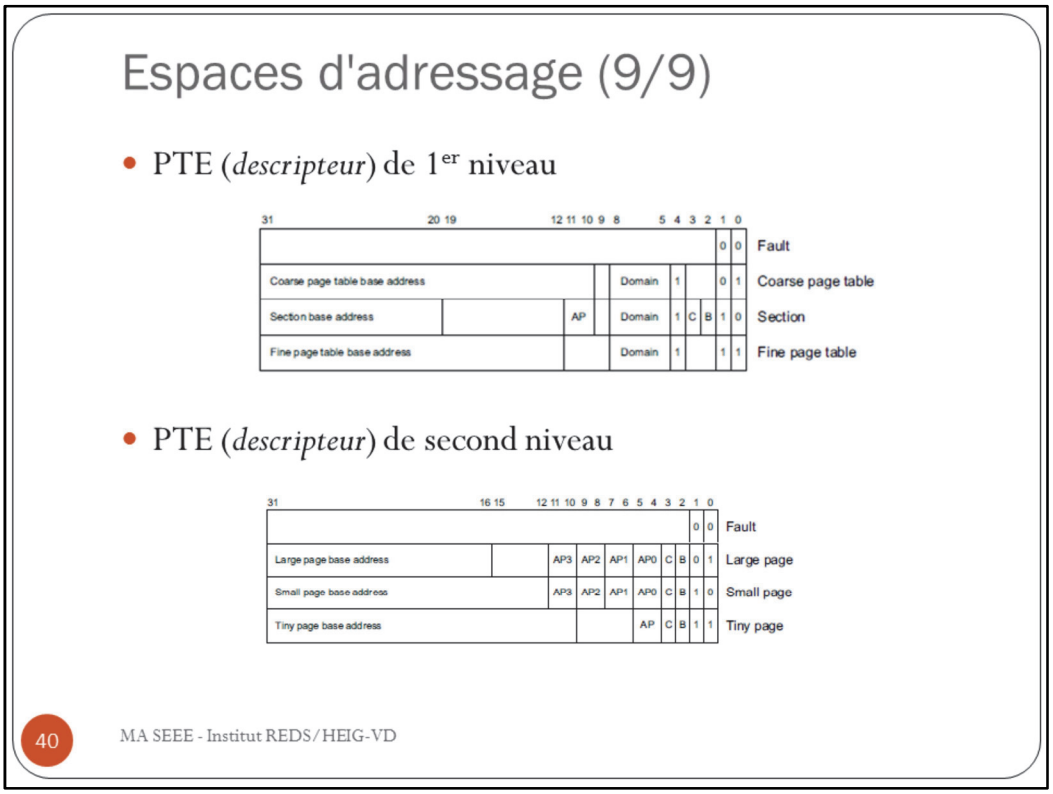


Un cœur ARM supporte différentes granularités de mappage. Il est possible de réaliser un mappage de page de 1 Mo (appelé *section*), de 64 Ko (appelé *large page*), de 4 Ko (appelé *small page*) et de 1 Ko (appelé *tiny page*).

Dans un noyau Linux, on utilise fréquemment les mappages de section (zones contrôlées par le noyau, zones I/O) ainsi que les pages de 4 Ko (zones utilisateur et certaines zones noyau).

La structure de la table de page de premier niveau est commune aux différents types de mappage, à savoir une table de 16 Ko contenant 4096 entrées (PTEs) d'une taille de 4 octets (32 bits) chacune. Les bits d'une PTE de premier niveau permet de déterminer le type de mappage.

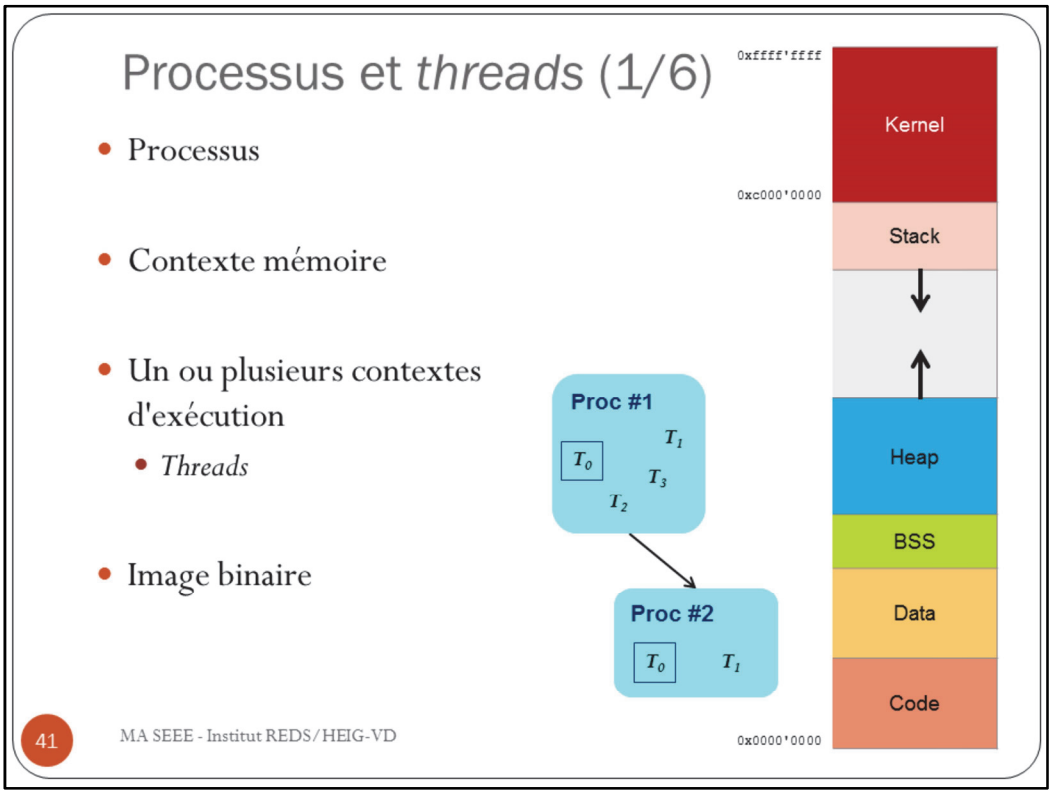
La structure des tables de page de second niveau quant à elle diffère en fonction du type de mappage. Lorsque l'on mappe des sections, l'offset est représenté par les 20 bits de poids faible de l'adresse, et les 12 bits de poids fort sont utilisés pour déterminer le numéro de page (section). Par conséquent, seul 12 bits sont nécessaires au niveau de la PTE pour la récupération de la page physique. C'est ce que l'on peut constater sur la description de la PTE sur la page suivante.



Sur ARM, la PTE de premier et second niveau – appelée *descripteur* dans la terminologie ARM - montre une structure différente en fonction du type de mappage. Lors d'un mappage de section, seul une PTE de premier niveau est nécessaire: les 12 bits de poids fort contiennent la *section base address.* Les bits de poids faible de l'adresse représentent l'offset et sont au nombre de 20 (2^{20} = 1 Mo).

Dans le cas d'un mappage sur 64 Ko ou de 16 Ko, les bits de la PTE de premier niveau sont utilisés cette fois-ci pour l'adresse d'une table de second niveau, à savoir 22 bits de poids fort pour l'adresse d'une table de second niveau de type "*coarse page table*", et 20 bits pour celle de type "*fine page table*".

Par conséquent, et selon la description du schéma de la page précédente, une page de 4 Ko peut être référencée par une table de second niveau de type "*fine page table*", table qui occupe elle-même une page de 4 Ko complète, ou par une table de type "*coarse page table*", table plus petite de 1 Ko avec moins d'entrées. C'est dire qu'avec ce type de table, on peut utiliser 4 tables de niveau 2 à l'intérieur d'une page de 4 Ko, ce qui est réduit considérablement l'utilisation de la RAM pour des structures de données gérées par le noyau uniquement. Toutefois, ce cas de figure s'avère intéressant si l'espace d'adressage virtuel est accédé de manière très éparse (comme des accès aléatoires ou des périphériques mappés de manière très segmentés).

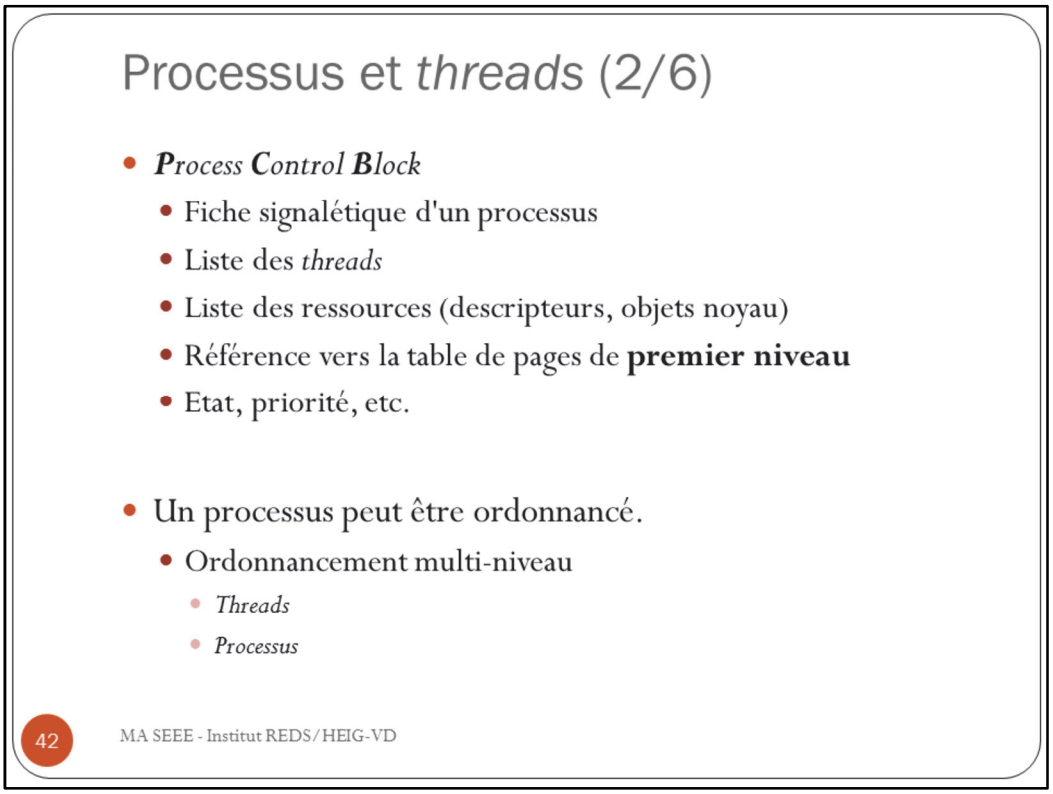


Un processus est caractérisé par un contexte mémoire défini par un espace d'adressage virtuel. Ce dernier est organisé d'une manière claire et découpé en différentes zones appelées ***section***. Les sections les plus communes sont la section *text* (code du processus), *data* (variables globales/statiques initialisées, constantes), *BSS* (*Block Started by Symbol*) contenant les variables non initialisées où l'allocation mémoire s'effectue lors du chargement), le tas (*heap*) permettant les allocations dynamiques, et la pile (*stack)* permettant le stockage des variables locales ainsi que le passage éventuel d'arguments lors d'appels de fonction et une adresse de retour.

Chaque processus dispose de son propre espace d'adressage virtuel, ce qui a l'avantage d'isoler complètement les processus entre eux et garantir ainsi une sécurité accrue: par conséquent, un processus **ne peut jamais interférer** directement avec un autre.

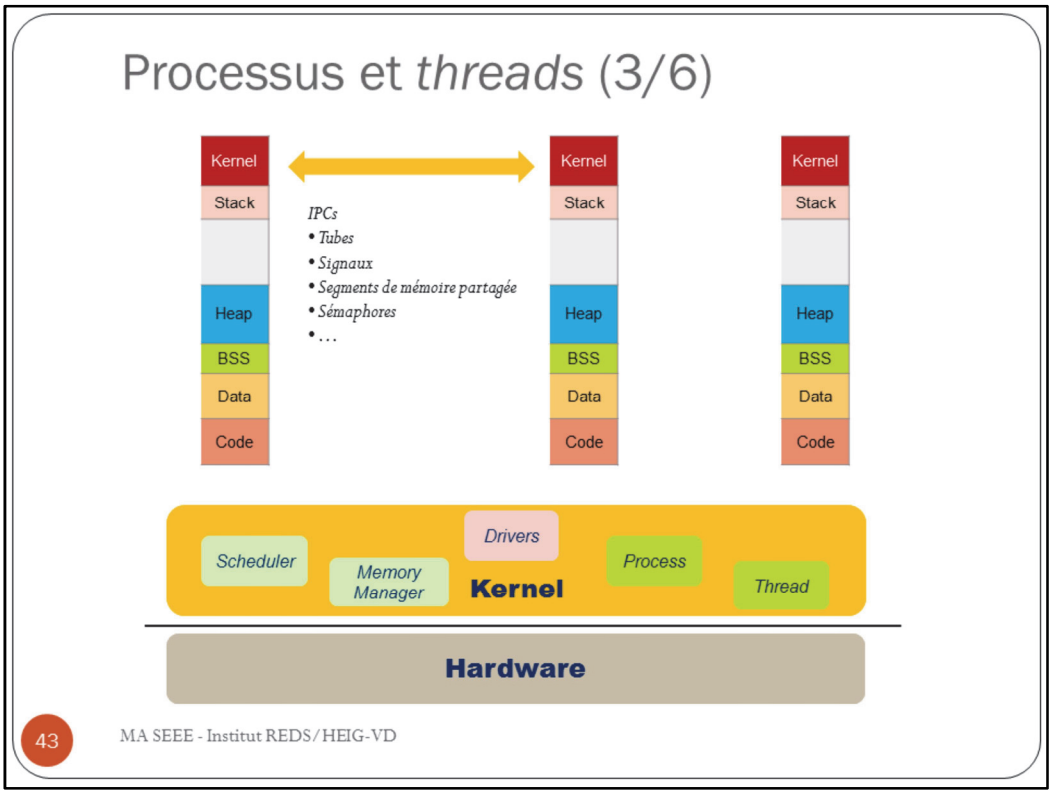
L'espace d'adressage (virtuel) d'un processus est découpé en deux grandes parties: la partie *utilisateur* (*espace utilisateur*) et la partie *noyau* (*espace noyau*). La taille de ces deux zones est habituellement de 3 Go pour la partie *user* et de 1 Go pour la partie *kernel*. Ce découpage est aussi décrit sous la forme (***3G:1G***). Il existe cependant d'autres découpages (***2G:2G*** ou ***4G:4G***).

La communication entre les processus s'effectue à l'aide mécanismes IPC (*Inter-Process Communication*) tels que tubes, signaux, segments de mémoire partagée, etc. Tous ces objets sont sous contrôle du noyau et constituent des ressources disponible au niveau du processus.



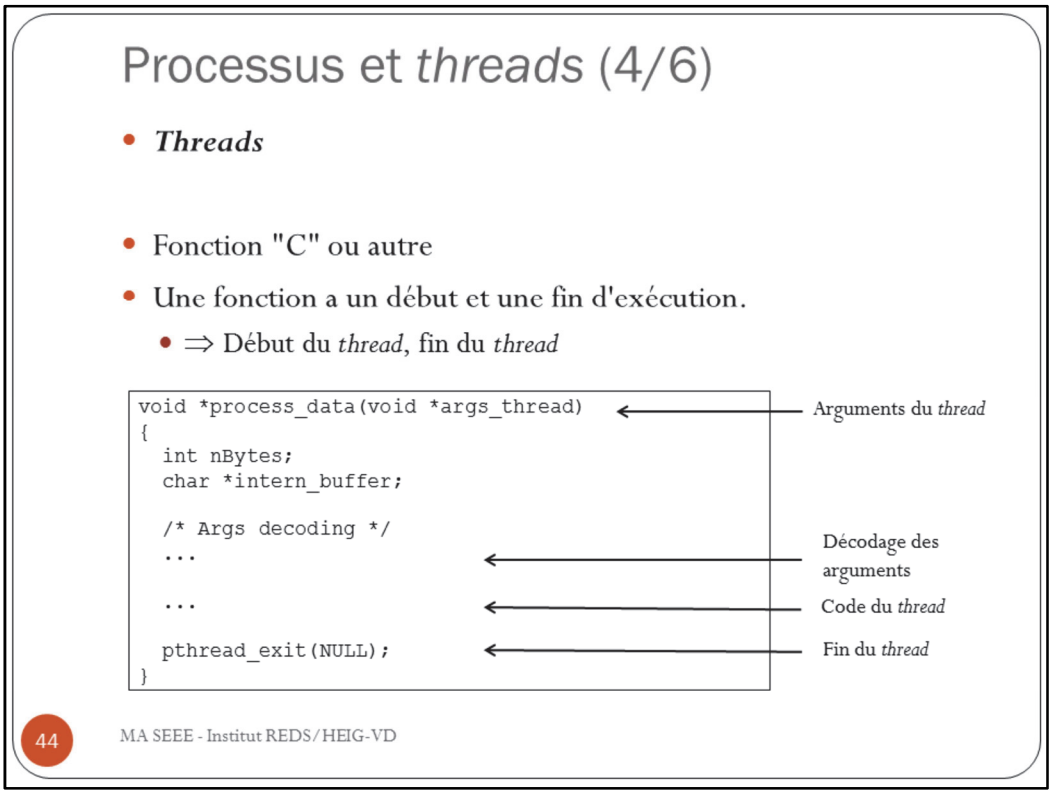
Le *Process Control Block* (PCB) permet de préserver toutes les informations relatives à un processus utilisées par le noyau. Cette structure de donnée se trouve donc dans l'espace noyau.

Un changement de contexte entre deux processus consiste à stocker l'état des registres du processeur (et des coprocesseurs) et de changer ces valeurs avec celles stockées dans un autre PCB. Par ailleurs, il est indispensable de changer le contexte mémoire, c-à-d de reconfigurer la MMU afin qu'elle pointe vers la table de premier niveau correspond au processus nouvellement élu.



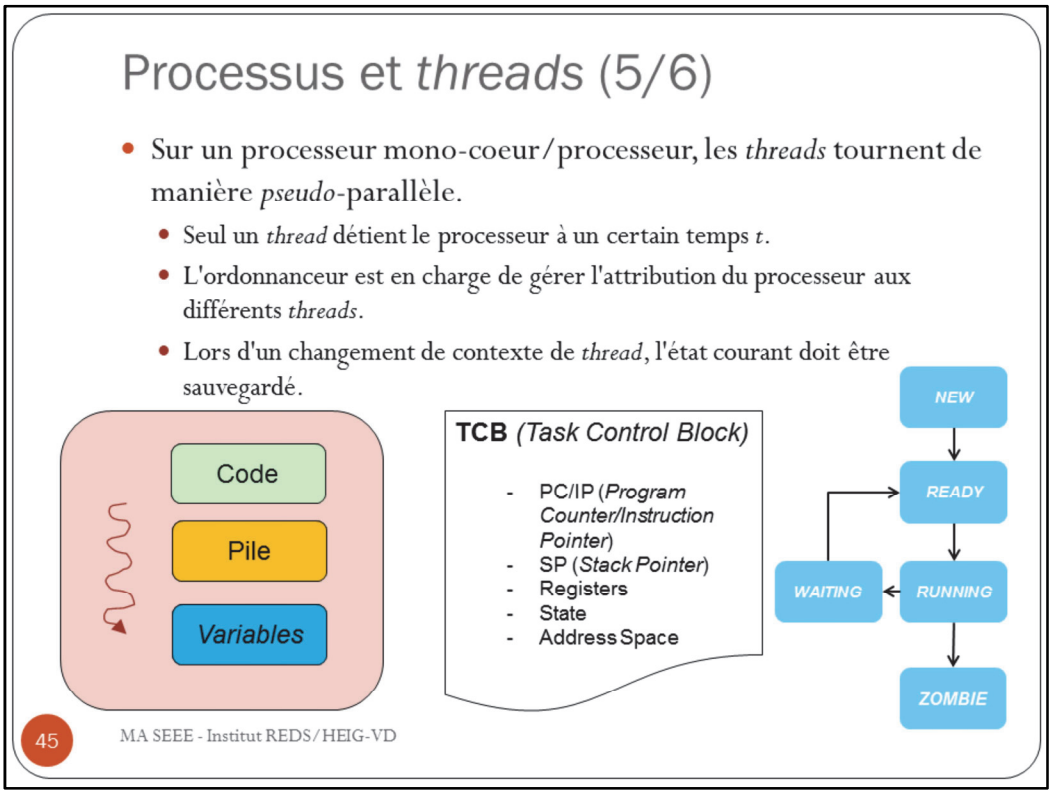
La plupart des noyaux d'OS supporte la notion de processus et de *threads.* Sous *Linux*, la notion de *threads* uniquement est utilisée, un processus étant représenté par le *thread* principal qui contient une référence vers le contexte mémoire. Sous Windows, les deux notions sont bien séparées dans le noyau.

L'ordonnancement peut s'effectuer au niveau du processus et/ou des *threads*. La politique d'ordonnancement détermine les critères qui sont examinés pour déterminer le prochain processus/*thread* à exécuter.



"Les *threads* se distinguent du multiprocessus plus classique par le fait que deux processus sont typiquement indépendants et peuvent interagir uniquement à travers une API fournie par le système telle que IPC. D'un autre côté les *threads* partagent une information sur l'état du processus, des zones mémoire ainsi que d'autres ressources. Puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux *threads* est moins coûteuse que la commutation de contexte entre deux processus. On peut y voir un avantage de la programmation utilisant des *threads* multiples." (*source: Wikipedia*)

Il est à noter que la programmation par *thread* nécessite souvent l'utilisation de verrous (*mutex,* sémaphore, etc.) afin de protéger l'accès concurrent aux variables.

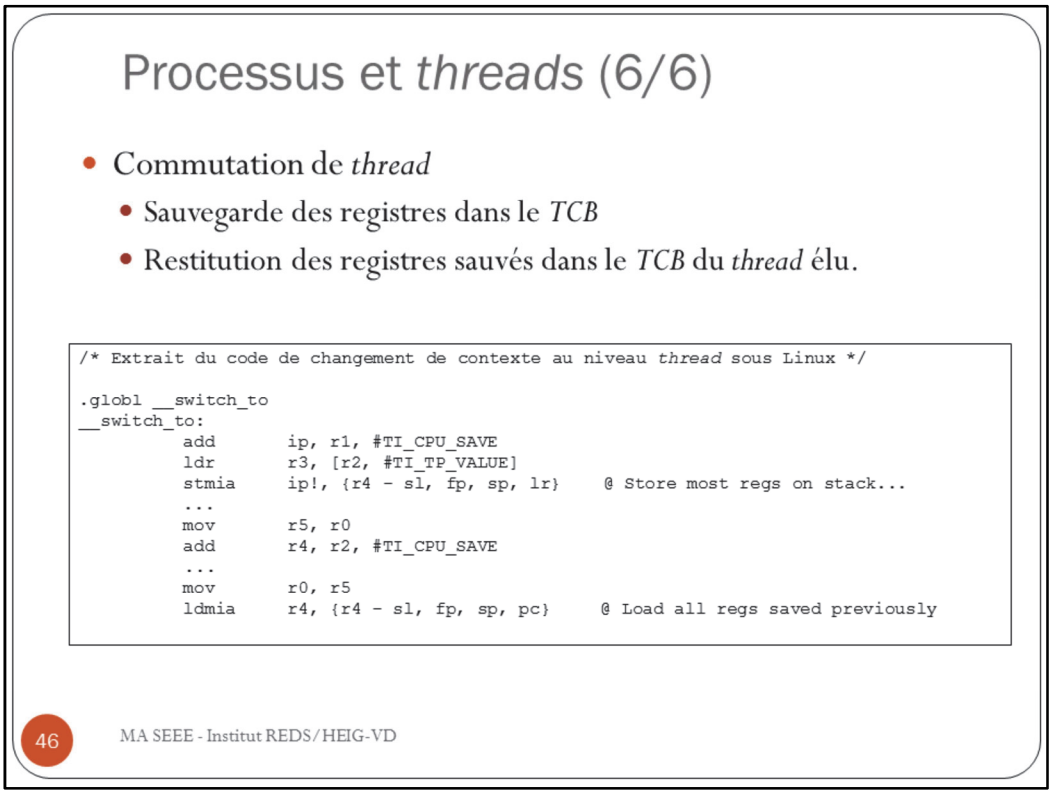


Pour s'exécuter correctement, une fonction doit disposer d'une **pile** d'exécution (arguments, variables locales, imbrication, sauvegarde temporaire, etc.). La fonction peut également avoir recours à des objets globaux (variables globales, objets partagés tels que *mutex* ou sémaphore, etc.).

Par analogie, un *thread* disposera donc de sa propre pile et de ses registres de travail. Les registres d'un *thread* correspondent à ceux utilisés par le processeur lorsque le *thread* est dans l'état actif (*running*). Lorsque le *thread* est inactif, les registres-machine sont **copiés** dans des variables propres aux *threads* (registres virtuels).

On distingue entre *thread* utilisateur et *thread* noyau. Un *thread* utilisateur tourne dans l'espace utilisateur, i.e. lorsque le processeur est en mode utilisateur. Un *thread* noyau est créé, géré et détruit dans le noyau, i.e. lorsque le processeur est en mode noyau.

Il existe des librairies de *threads* dans l'espace utilisateur permettant un ordonnancement particulier sans faire appel à un ordonnancement dans le noyau. Dans ce cas, ce dernier n'a pas forcément connaissance de l'existence de *threads* dans l'espace utilisateur. Cette approche correspond à un mappage de *thread* utilisateur-noyau `<n:1>`. Dans la majorité des cas, un *thread* utilisateur correspond à un *thread* noyau, le mappage étant de type `<1:1>`.



Le code ci-dessus illustre bien le changement de contexte entre *thread* (sous *ARM Linux*). La fonction `__switch_to()` prend trois arguments (*`prev`*, *`thread_info(prev)`*, *`thread_info(next)`*) stockés respectivement dans les registres `r0`, `r1` et `r2`.

L'instruction `stmia` permet la sauvegarde de registres multiples (à l'endroit du TCB prévu à cet effet) et l'instruction `ldmia` restitue les valeurs de registre du prochain *thread* à exécuter. On remarque que le registre *`pc`* récupère l'adresse de la prochaine instruction à exécuter du prochain *thread*, là où celui-ci avait été interrompu précédemment.

Références

- Linux LWM, http://lwm.net
- Linux Device Drivers (3rd Edition), Jonathan Corbet, Alessandro Rubini & Greg Kroah-Hartman
  - **Version online**: http://lwn.net/images/pdf/LDD3
- Jochen Liedtke: On u-Kernel Construction, *SOSP '95 Proceedings of the fifteenth ACM symposium on Operating systems principles*
- Pierre Ficheux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)
- ARM Architecture Reference Manual, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html
- Intel® 64 and IA-32 Architectures Software Developer's Manual, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf
- http://wiki.qemu.org/Documentation/GettingStartedDevelopers#Getting_to_know_the_code
- Eclipse C/C++ Indexer, http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/concepts/cdt_c_indexer.htm