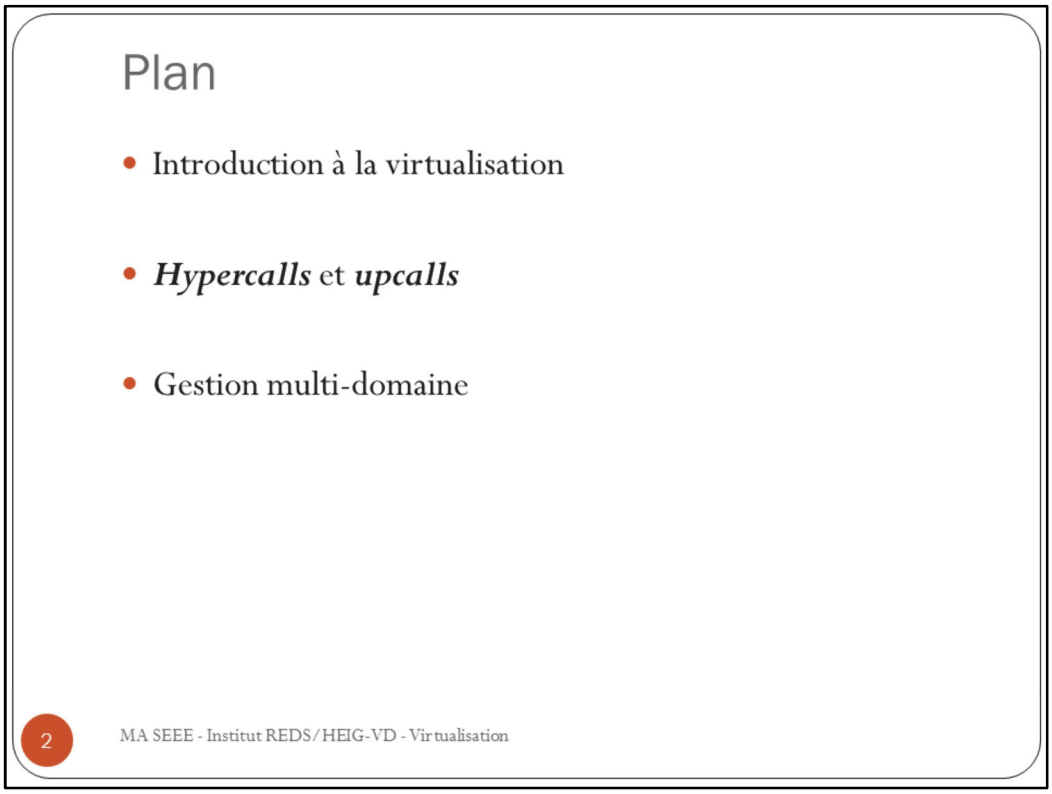
# Systèmes d'exploitation et Environnements d'Exécution Embarqués (SEEE)
## *Virtualisation*

Module d'approfondissement MSE

Prof. Daniel Rossier

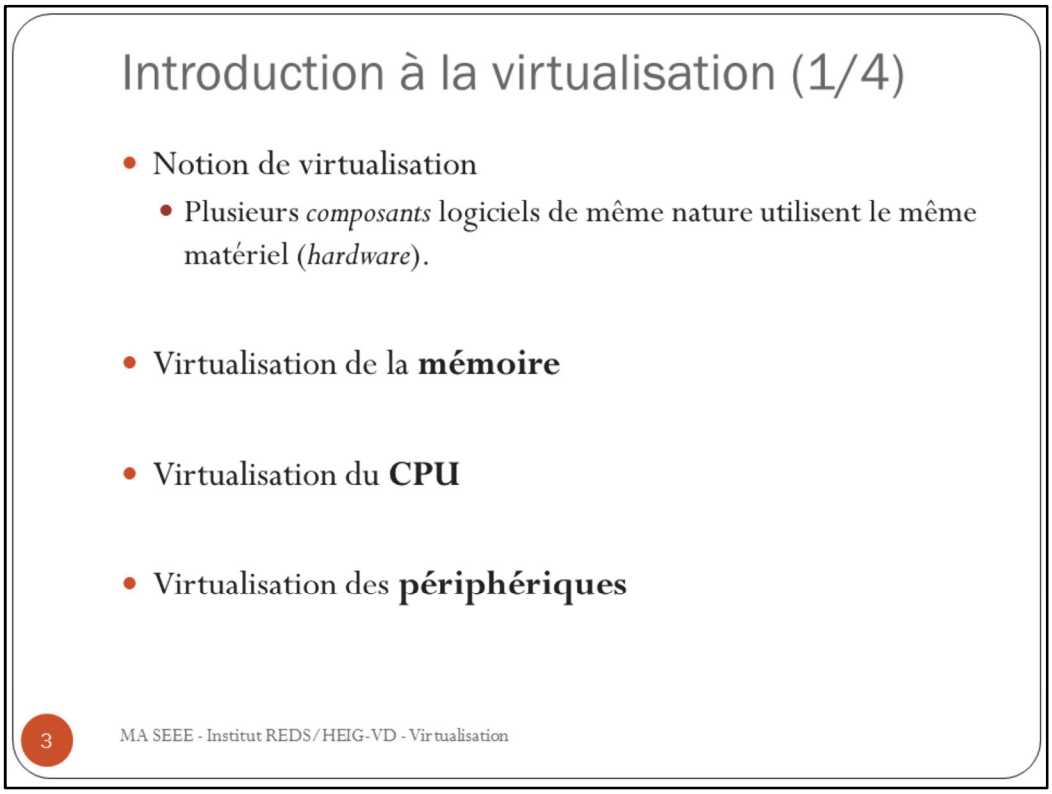
Version 1.5 (2017)




Les mécanismes de virtualisation ont été introduits il y a fort longtemps dans les années 60 par IBM avec leur système CP/CMS. La motivation première de la virtualisation est la possibilité ***d'isoler*** des environnements d'exécution logiciels de manière sécurisée. C'est le cas, par exemple, avec plusieurs processus s'exécutant sur un OS. Par analogie, cette approche peut s'étendre au niveau d'un OS complet: plusieurs OS – dit ***OS invités*** – peuvent tourner sur un même CPU de manière isolée, sous contrôle d'un **moniteur** (logiciel) s'exécutant avec des droits privilégiés.

Les avantages de la virtualisation sont multiples. On peut citer:

- **L'utilisation optimale des ressources** d'un parc de machines
- L'installation, **déploiement et migration facile** des machines virtuelles d'une machine physique à une autre
- L'économie sur le matériel par **mutualisation** (consommation électrique, entretien physique, surveillance, support, compatibilité matérielle, etc.)
- L'installation, tests, développements, cassage et possibilité de recommencer **sans casser le système d'exploitation hôte**
- La **sécurisation** et/ou **isolation** d'un réseau
- **L'isolation des différents utilisateurs** simultanés d'une même machine (utilisation de type site central)
- L'allocation dynamique de la puissance de calcul **en fonction des besoins** de chaque application à un instant donné



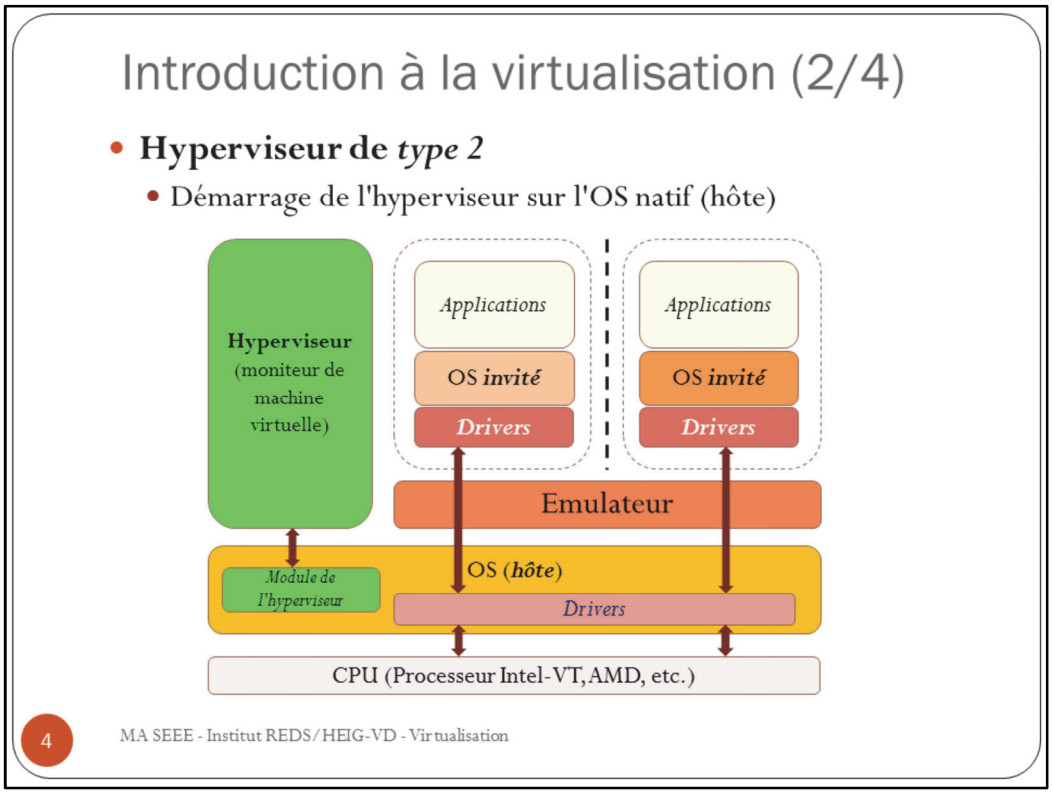

Aujourd'hui, la plupart des processeurs 32/64 bits possèdent un support matériel pour assister les tâches de virtualisation; alors que ce support est largement répandu dans les processeurs qui équipent les PCs/laptops et serveurs, ce n'est que récemment que les microcontrôleurs en sont équipés, avec notamment la nouvelle génération de microcontrôleurs basés sur la famille *Cortex-A15.*

Le concept de virtualisation dans le monde des systèmes embarqués est très attractif, en particulier pour des aspects liés à la **sécurisation** (**isolation** des environnements d'exécution), à la capacité de gérer du matériel **hétérogène**, à la sécurisation d'un **environnement critique** (temps-réel strict), et bien d'autres encore.

Dans ce contexte, la virtualisation permet d'offrir une abstraction matérielle aux systèmes d'exploitation. Dans les systèmes embarqués, on ne cherchera pas à faire tourner "beaucoup" d'OS invités, mais deux suffiront dans la plupart des scénarios.

La virtualisation intervient à différents niveaux: la **mémoire** (chaque OS possède une région de mémoire physique dédiée), le **CPU** (il est alloué aux différents OS de manière équitable), et les **périphériques** (ils sont partagés entre les OS invités).

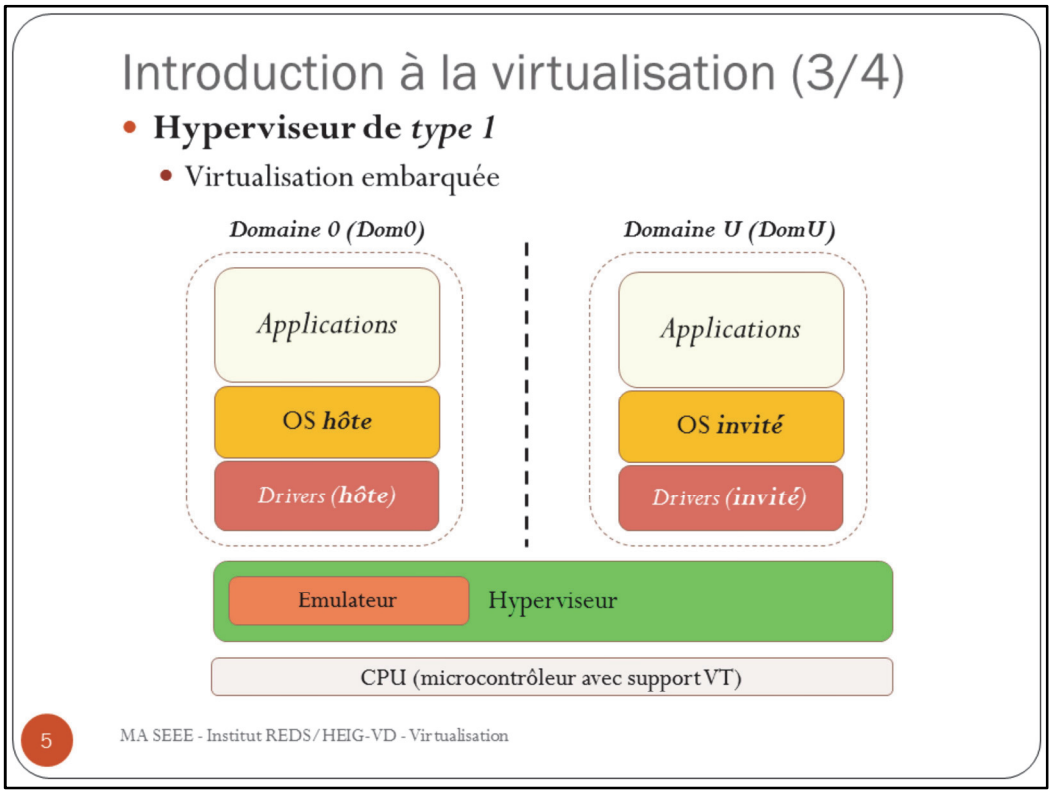
La virtualisation repose principalement sur deux mécanismes fondamentaux: la protection de certaines instructions (**instructions privilégiées**) et **l'utilisation d'une MMU**.




Le moniteur de machine virtuel est généralement appelé ***hyperviseur***. Il doit pouvoir tourner dans un mode privilégié afin de pouvoir arbitrer les OS invités et gérer au mieux les interactions avec le matériel.

On distingue deux types d'hyperviseur: les hyperviseurs de **type 1** qui tournent *directement* sur le matériel en mode privilégié, et les hyperviseurs de **type 2** qui tournent sur un OS hôte et n'interagissent pas directement sur le matériel. Il va de soi que les hyperviseurs de type 2 sont moins efficaces que ceux du type 1 car l'OS invité apparaît au troisième niveau (OS hôte – hyperviseur – OS invité). Une approche hybride consiste à modifier le noyau de l'OS hôte afin de se rapprocher du matériel et de gagner en privilèges. De tels exemples sont évidemment *VirtualBox, Parallels, VMWare, etc.*

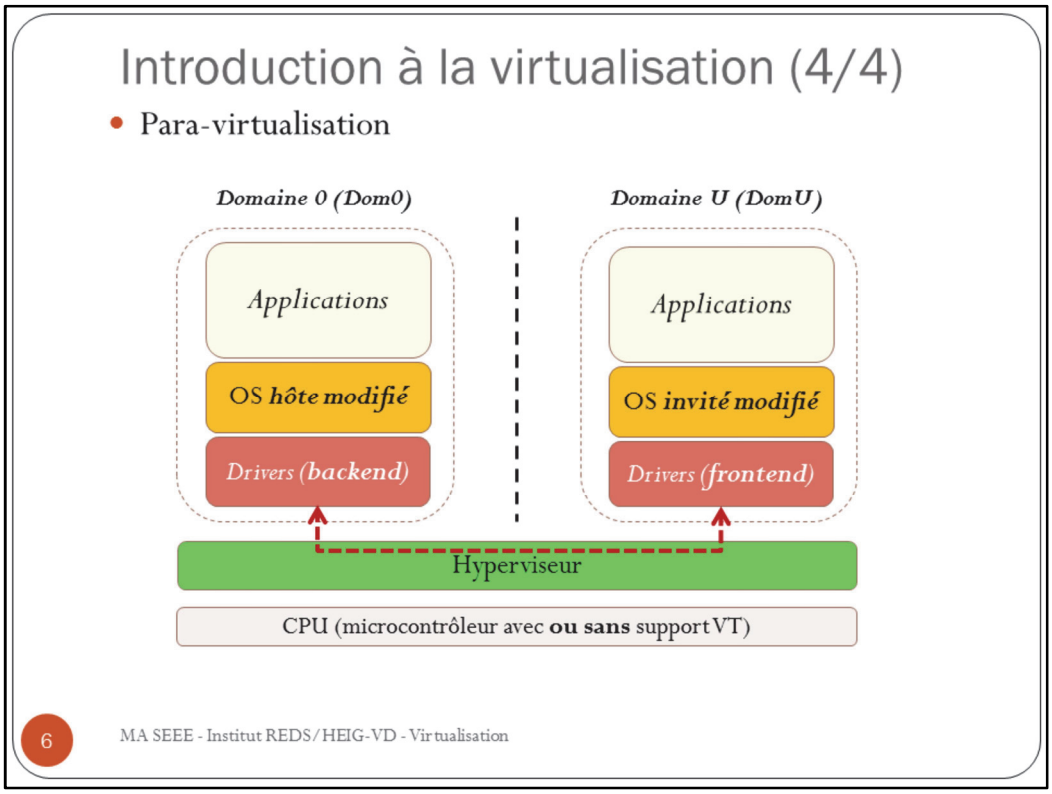
Si plusieurs OS invités peuvent tourner sur la machine hôte, les *drivers* de l'OS ne peuvent pas sans autre accéder au matériel "comme s'ils étaient seuls". On imagine facilement les problèmes de conflits et de concurrence que cela pourrait entraîner. C'est pourquoi, afin que le code des *drivers* puissent être utilisé sans autre, il est nécessaire d'***émuler*** les périphériques en question. L'émulateur (qui fait normalement partie de l'hyperviseur dans l'approche de type 2) doit donc *traiter* les instructions des *drivers*. Mais alors, comment faire la différence entre de telles instructions et les autres instructions de l'OS invité?
C'est à ce niveau qu'interviennent les deux premiers types de virtualisation: la **virtualisation complète** et la **virtualisation assistée** (matériellement).




La **virtualisation complète** (ou *full virtualisation*) consiste à faire tourner l'OS invité (et par conséquent toutes ses applications "utilisateur") dans un environnement émulé; **toutes les instructions** sont interceptées et traitées par l'émulateur. Ce sont les débuts de la virtualisation, alors que les processeurs n'avaient aucune assistance matérielle pour la virtualisation. Cette technique est très lente et ne convient pas pour des systèmes embarqués.

La virtualisation assistée matériellement (*accelerated virtualization* ou encore *Hardware Virtual Machine (HVM))* permet au code invité de s'exécuter de manière natif et de déclencher une interruption logicielle synchrone lorsque certaines instructions s'exécutent (par exemple, une instruction de lecture/écriture à une adresse I/O). Ainsi le contrôle peut être donné à l'hyperviseur qui peut, à son tour, donner la main à l'émulateur afin que celui-ci puisse gérer les interactions avec le *vrai* matériel. L'émulateur gère une *copie* du périphérique concerné; le périphérique est ainsi ***virtualisé*** et l'OS invité dispose d'une interface adéquate.

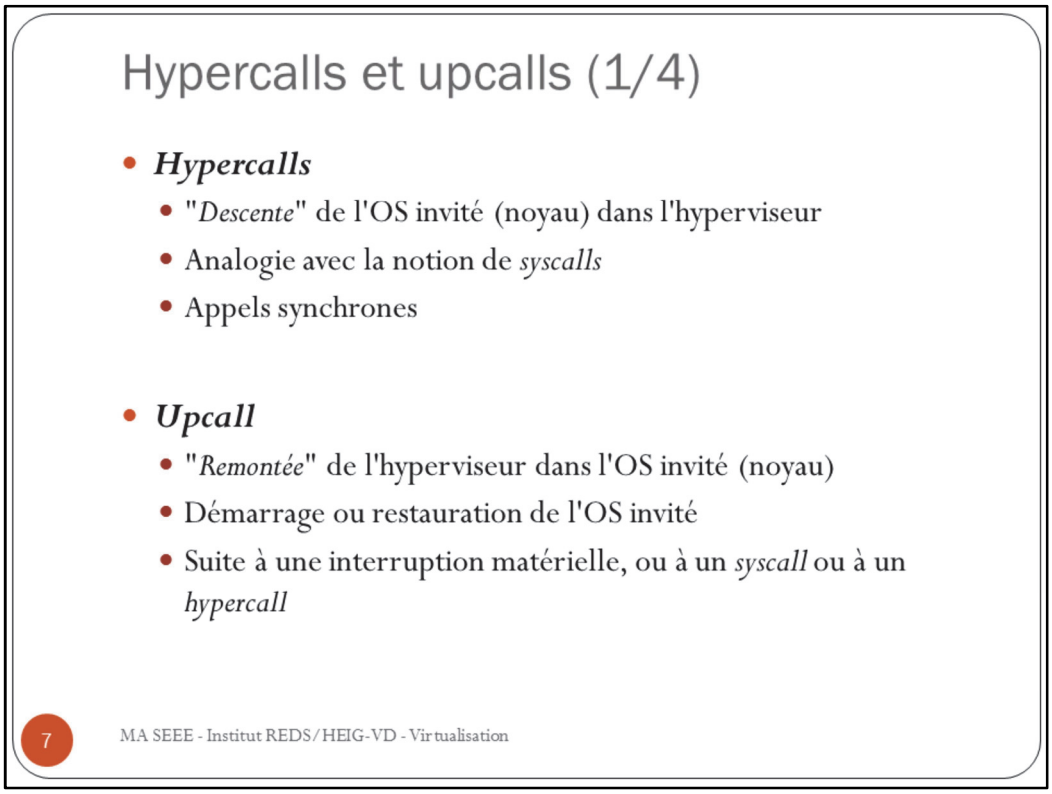
Le schéma ci-dessus présente une architecture de virtualisation de **type 1**. C'est l'environnement "classique" pour un système embarqué. L'hyperviseur est démarré *(booté)* en premier et tourne de manière natif. Deux OS – appelés aussi ***domaines*** – démarrent: *dom0* représente généralement l'OS hôte et contient les *drivers* originaux pilotant les périphériques locaux de la plate-forme. Un second domaine, appelé ***domU***, correspond au second OS invité.




L'architecture précédente implique par conséquent que le microcontrôleur puisse assister la virtualisation, car sinon une couche complète d'émulation (*full virtualisation*) est requise et ne se prête pas pour un tel système.

Pour les processeurs n'ayant aucun support pour la virtualisation, une troisième technique est possible: la ***para-virtualisation***. Dans ce cas-ci, l'OS hôte et invité doivent subir des modifications, ainsi que les *drivers*.
En effet, comme il n'est pas possible de rediriger l'exécution via le processeur, l'OS doit avoir une certaine connaissance de l'hyperviseur afin de pouvoir interagir directement avec lui. De même, l'hyperviseur peut appeler des fonctions de type *callback* implémentées complètement dans l'OS (hôte ou invité). Ces mécanismes seront détaillés plus tard.

Ainsi, la para-virtualisation offre un très bon compromis pour le support de la virtualisation dans les systèmes embarqués, avec un hyperviseur ayant une empreinte mémoire réduite et deux systèmes d'exploitation. Il existe aujourd'hui bon nombre de *frameworks* de virtualisation pour un OS et un RTOS, ou pour deux OS avec des mécanismes avancés de sécurité; la plupart de ces *frameworks* sont commerciaux.
*EmbX (anciennement EmbeddedXEN)* est un environnement *open source* pour faire tourner deux OS de type *Linux* sur un système embarqué de type ARM, basé sur XEN et développé au sein de l'institut REDS *(http://sourceforge.net/projects/embeddedxen).*

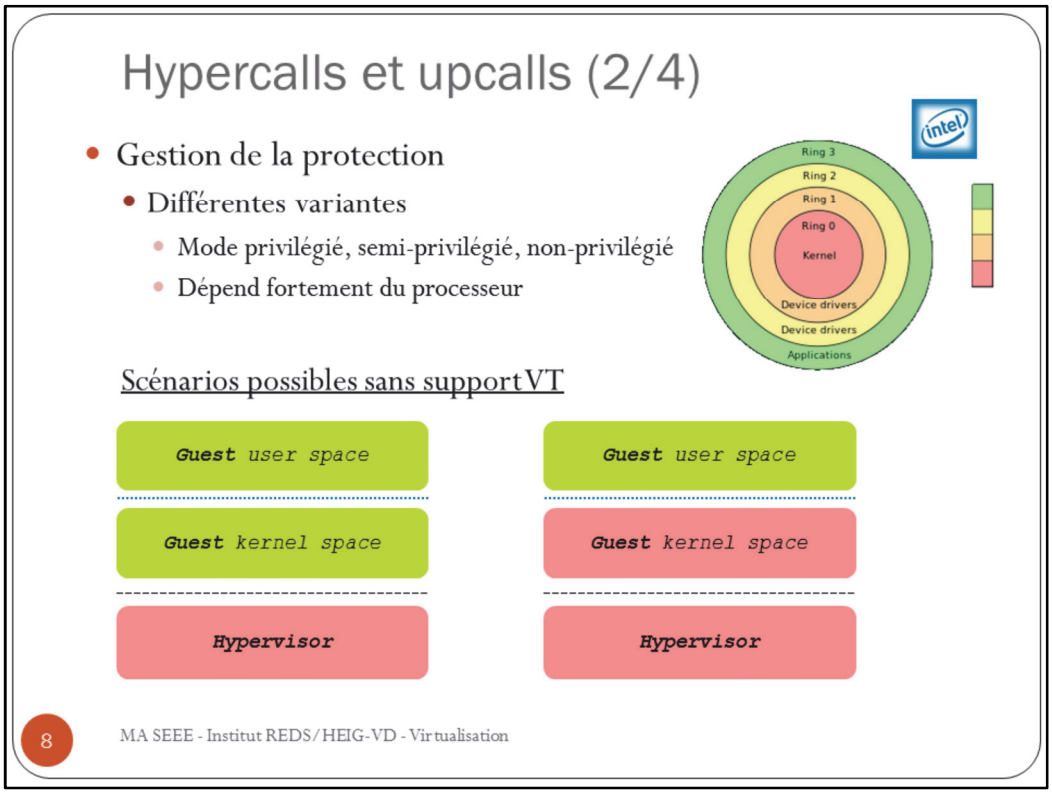



Par la suite, nous utiliserons le terme "*invité*" pour désigner indifféremment l'OS hôte (*dom0*) et l'OS invité (*domU*).

Nous savons qu'un code s'exécutant dans l'espace utilisateur doit avoir recours aux appels systèmes (*syscalls*) afin de pouvoir accéder aux services du noyau et aux périphériques. Dans un environnement virtualisé, une couche logicielle s'insère entre le matériel et le noyau invité; ce dernier peut également avoir recours aux services de l'hyperviseur, et un mécanisme similaire aux appels systèmes existe entre le noyau invité et l'hyperviseur: l'***hypercall***.

A l'instar d'un *syscall*, un *hypercall* peut être implémenté sous forme d'une interruption logicielle ou d'un saut vers une routine appartenant à l'hyperviseur. Nous verrons plus loin que l'une ou l'autre approche dépendra des caractéristiques du microcontrôleur.

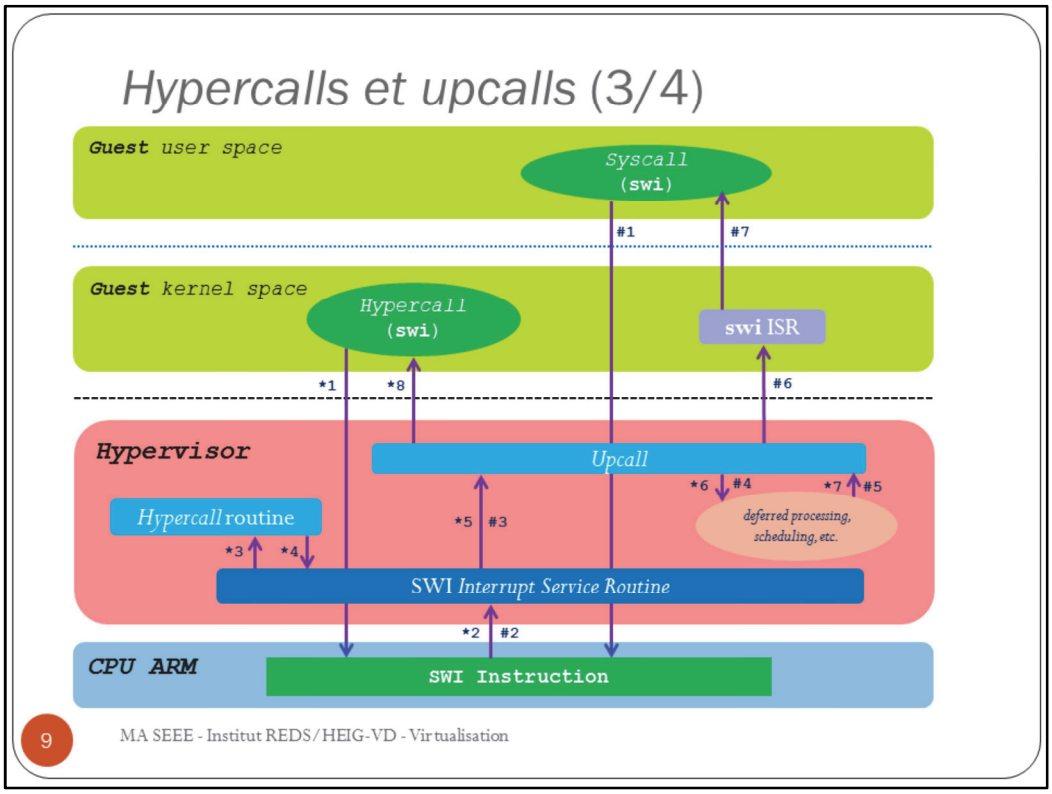
Au démarrage du système, l'hyperviseur s'exécute. Il effectue des initialisations basiques et prépare les deux OS invités à leur exécution. Lorsque un OS invité s'exécute, il n'est possible de retourner dans l'hyperviseur que dans deux situations (identiques au scénario avec un processus et le noyau d'OS): soit une interruption matérielle survient, soit le code effectue un *hypercall*. Lorsque le code de l'hyperviseur termine son exécution, il redonne la main à l'un des OS invité (il peut y avoir changement de contexte entre les domaines). Le chemin d'exécution effectuant la transition entre l'hyperviseur et l'OS invité est appelé ***upcall***.



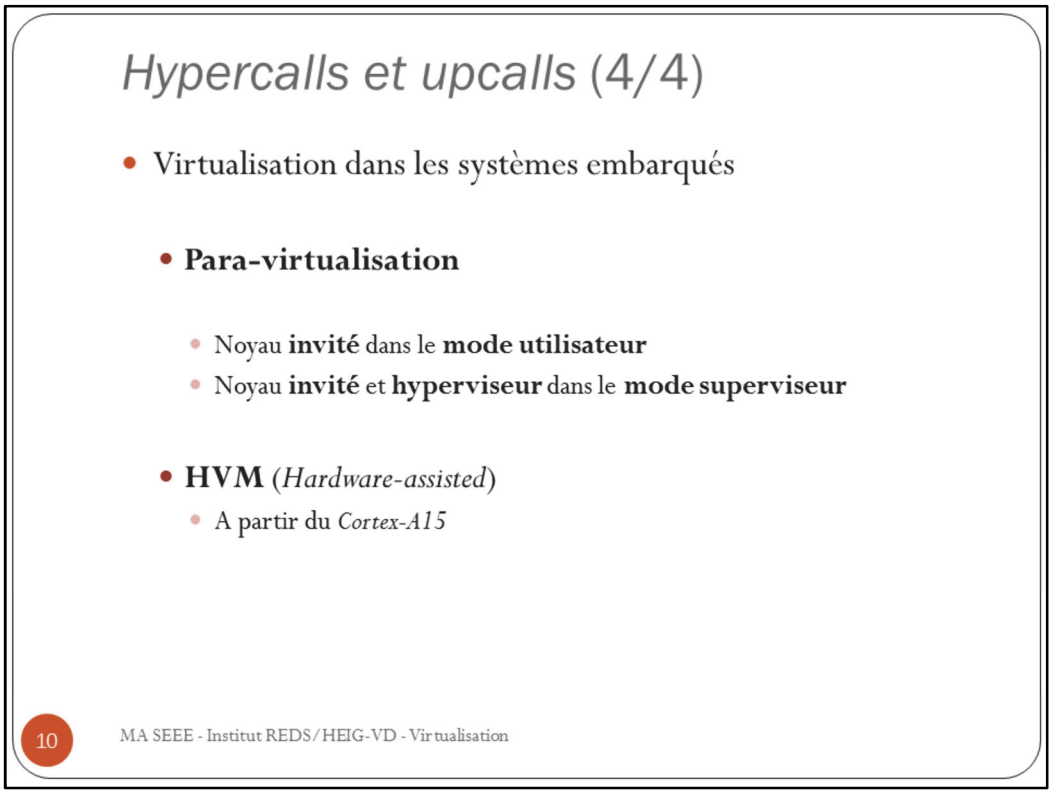

L'isolation entre OS ne peut être entièrement garantie (sécurisée) que si leurs noyaux s'exécutent **dans un mode différent** de celui de l'hyperviseur; s'ils disposent des mêmes privilèges que l'hyperviseur, les noyaux invités peuvent potentiellement **interférer entre eux** et corrompre l'exécution de l'hyperviseur.

C'est pourquoi les processeurs de dernière génération, pour autant qu'ils supportent la virtualisation, possèdent plusieurs modes d'exécution avec des niveaux de privilèges différents. Les processeurs *Intel Pentium* définissent la notion de *ring* équivalent à celle de mode d'exécution. L'hyperviseur peut ainsi tourner dans le mode le plus privilégié, le noyau invité dans un mode semi-privilégié et l'espace utilisateur dans le mode le moins privilégié.

Bien que les systèmes plus récents à base de CPU ARM supportent la virtualisation (*Cortex-A15, A7, etc.*), un nombre élevé de systèmes embarqués ne sont pas dotés de ce support. Dans ce cas, il serait possible de laisser le noyau invité s'exécuter en mode utilisateur, et d'utiliser la même instruction d'interruption logiciel des *syscalls* (*SWI*) au niveau du noyau, en définissant de nouveaux *syscalls* qui correspondraient alors aux *hypercalls*. Cette approche est présentée sur la page suivante. L'alternative consiste à faire tourner le noyau invité dans le mode noyau (identique à celui de l'hyperviseur). L'*hypercall* est implémenté sous forme d'un *trampoline*, c-à-d un saut dans une fonction implémentée dans l'hyperviseur depuis le noyau invité. L'adresse du *callback* est alors transmise à l'OS invité lors de son démarrage.
Cette approche est celle utilisée dans *EmbX*.



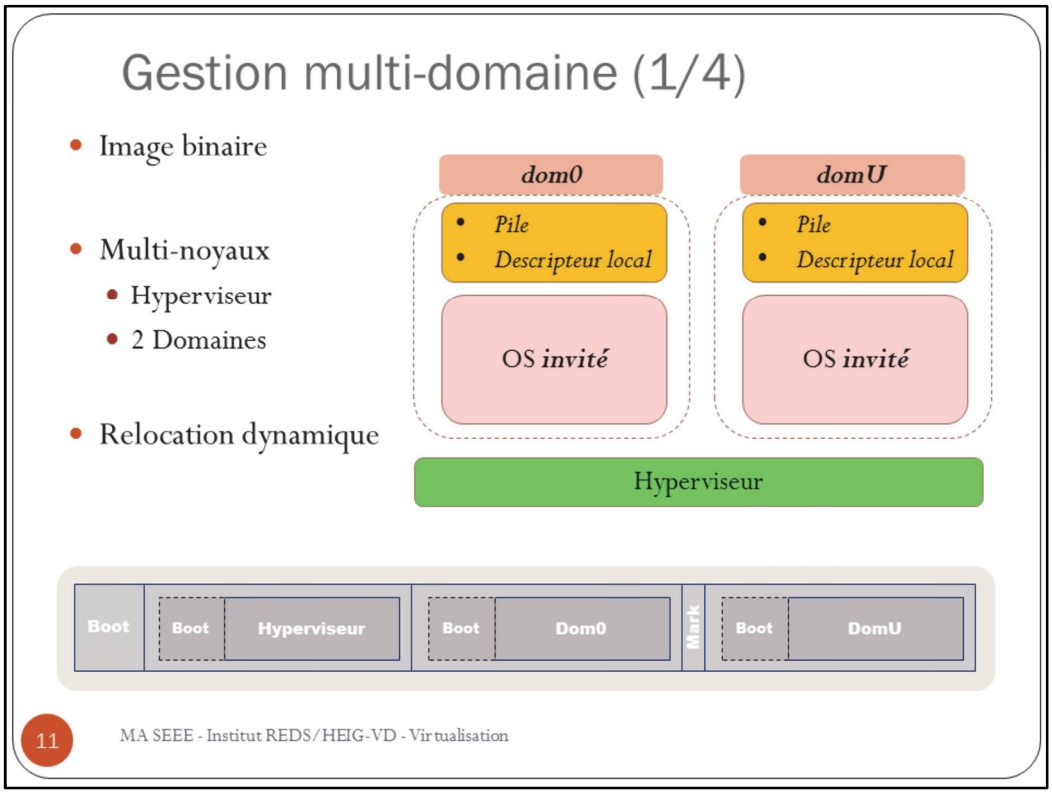

Examinons en détail le fonctionnement d'un *hypercall* et d'un *upcall* sur ARM avec un environnement para-virtualisé dans lequel le noyau invité tourne dans un mode non-privilégié. Lorsque le noyau invité invoque un *hypercall*, il utilise l'instruction *SWI* provoquant une exception (interruption logicielle) (**1*) conduisant à l'exécution de la routine de service correspondante (**2*). L'ISR implémentée dans l'hyperviseur récupère le numéro de l'appel système et détecte que celui-ci correspond à un *hypercall*. La routine associée à cet *hypercall* est alors exécutée (**3*). A son retour (**4*), l'ISR peut effectuer un *upcall* (**5*) en vue de restaurer l'exécution du noyau invité. Mais avant de redonner définitivement la main au noyau invité, il est possible d'effectuer certaines opérations dans le contexte de l'interruption (**6*) comme l'exécution de tâches différées *(softirq)* ou activer l'ordonnanceur de domaine. Lorsque ces tâches annexes sont terminées (**7*), l'hyperviseur peut terminer proprement le *syscall* invoqué dans le noyau invité (**8*).
Si le *syscall* est invoqué depuis l'espace utilisateur de l'OS invité (cas *nominal*), le mécanisme d'invocation de l'ISR est identique (#1, #2) et la routine détecte simplement que le numéro de *syscall* ne correspond pas à un *hypercall*. Dans ce cas, l'hyperviseur exécute directement la remontée via l'*upcall* (#3) en effectuant les différentes actions décrites dans le paragraphe précédent (#4, #5). Lorsque l'invité poursuit son exécution (#6), ce sera l'ISR du noyau associée à l'interruption logicielle des appels systèmes qui sera exécutée, correspondant bien au comportement "normal" sans virtualisation.




L'architecture précédente peut toutefois conduire à une utilisation intensive d'*hypercalls* et amener des problèmes de performance. De plus, certains microcontrôleurs n'autorisent pas à effectuer des accès I/O en mode utilisateur.

En résumé, l'utilisation de la virtualisation dans les systèmes embarqués dépend encore fortement des capacités matérielles des microcontrôleurs. Actuellement, c'est une approche basée principalement sur la para-virtualisation avec un noyau invité tournant dans le mode privilégié, au même titre que l'hyperviseur, qui est prédominante: le surcoût (*overhead*) induit par l'hyperviseur est limité au minimum, les performances sont meilleures, les modifications de l'OS invité au niveau de la para-virtualisation sont minimales et les accès matériels (I/O) peuvent se faire sans autre. En revanche, cette approche souffre d'une sécurité accrue au niveau de l'isolation des domaines.
Néanmoins, il est possible de trouver des *frameworks* de virtualisation pour ARM où le noyau invité tourne en mode utilisateur, et qui utilisent la notion de *domaine d'accès* ainsi que les mécanismes *TrustZone* implémentés au niveau du microcontrôleur afin de contrôler plus finement la sécurité des accès mémoire.

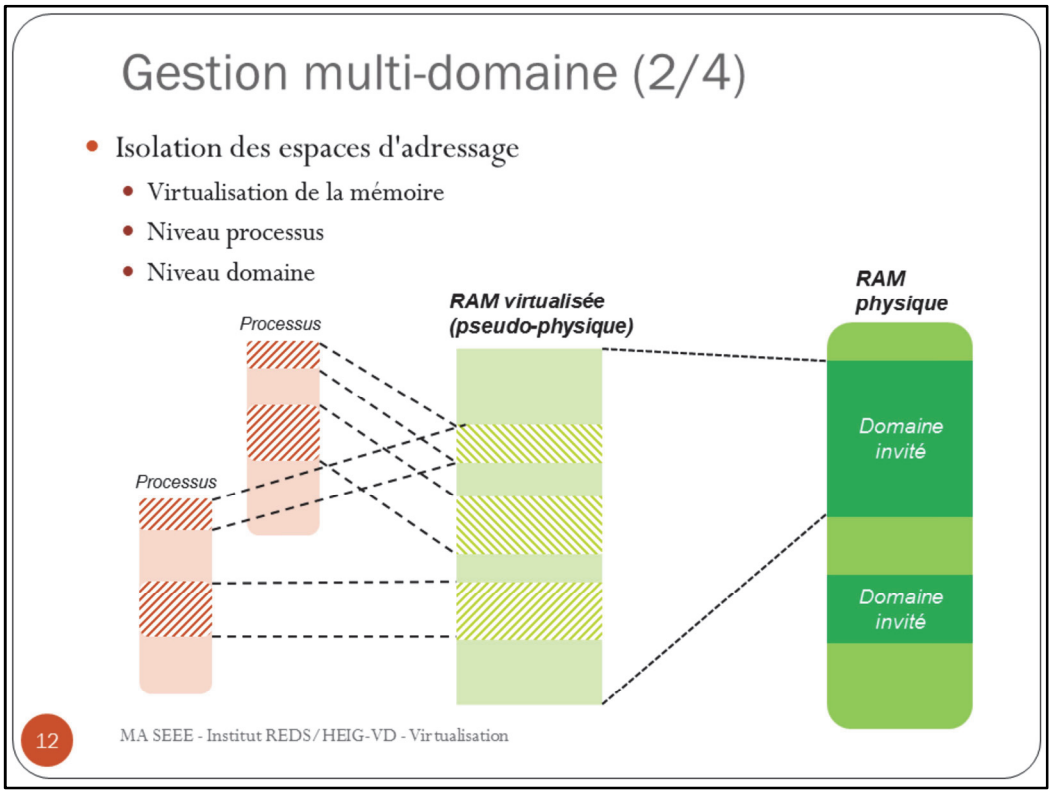
La nouvelle génération de microcontrôleurs ARM basés sur le *Cortex-A15* supporte pleinement la virtualisation au niveau CPU et permettra d'envisager l'utilisation d'une approche HVM.




Dans les systèmes virtualisés de type 1, l'hyperviseur et les OS invités sont en principe stockés dans des fichiers (images binaires) différents. L'hyperviseur démarre, puis nécessite le démarrage d'un premier OS invité afin de fournir une console de gestion à l'utilisateur, console à partir de laquelle il pourra démarrer les autres OS invités (appelés aussi machines virtuelles).

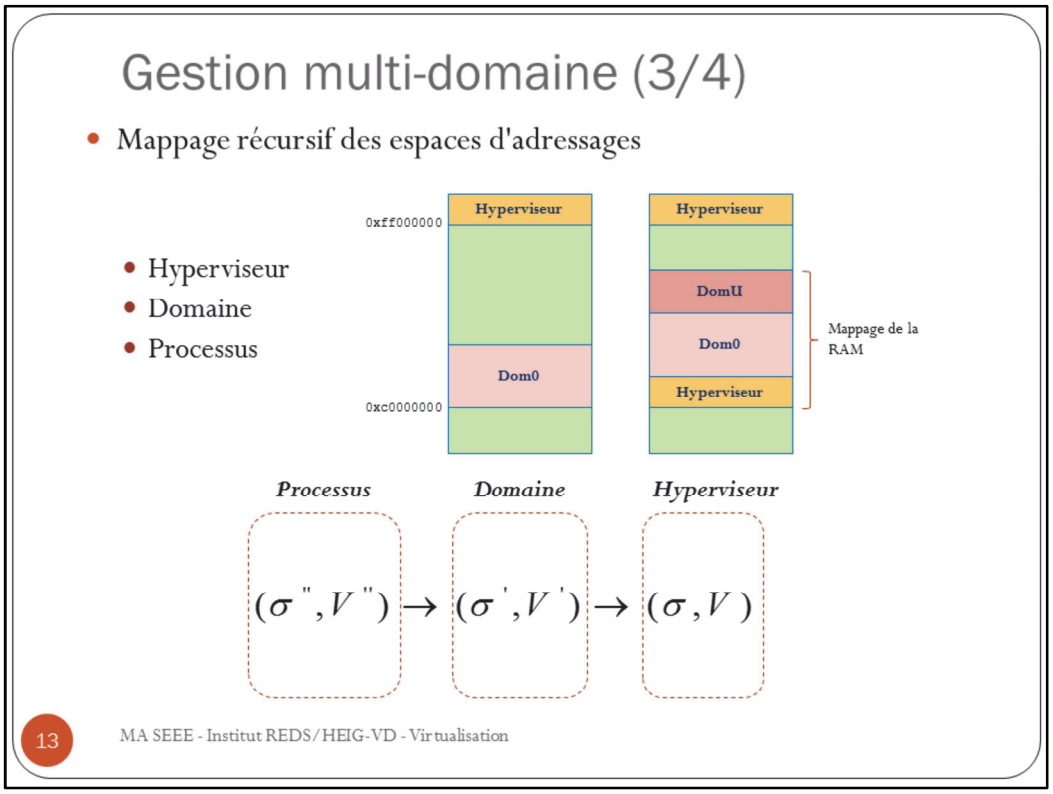
Avec *EmbX*, l'hyperviseur est les deux OS invités sont stockés dans un seul fichier. Comme le montre la figure ci-dessus, un premier code *bootstrap* décompresse l'image, puis passe le relai au code de *bootstrap* de l'hyperviseur. Celui-ci effectue un *parsing* de l'image binaire afin d'y extraire les deux domaines. Il gère leur relocation et lance à son tour le code de *bootstrap* du domaine, après avoir initialisé la pile et le descripteur local lié au domaine. Le descripteur contient des informations gérées par l'hyperviseur et relatives au domaine. On y trouve notamment un masque de bits utilisé lors d'interruptions afin de permettre à l'OS de les traiter via ses propres vecteurs d'interruption; ce mécanisme permet de ***virtualiser*** les interruptions matérielles.

L'image binaire telle que représentée ci-dessus correspond à un seul fichier pouvant être transféré sur la cible.




L'isolation des espaces d'adressage appartenant à l'hyperviseur et aux domaines invités peut être garantie à l'aide d'un mappage récursif de l'espace physique: un premier mappage permet la virtualisation de la RAM au niveau du domaine. En effet, le domaine (OS) doit pouvoir disposer d'une zone (généralement contiguë) de mémoire physique. S'il est seul à tourner, cette zone représente la RAM complète. Dans le cas de la virtualisation, **la RAM est *virtualisée*** et seule une portion est allouée au domaine. Ainsi, un premier mappage assurera l'isolation du domaine par rapport aux autres. Les adresses physiques devient alors des adresses virtuelles et cet espace est appelé ***pseudo-physique***. Les adresses pseudo-physiques sont des adresses virtuelles du point de vue de l'hyperviseur (et du CPU), mais des adresses physiques du point de vue du noyau invité.

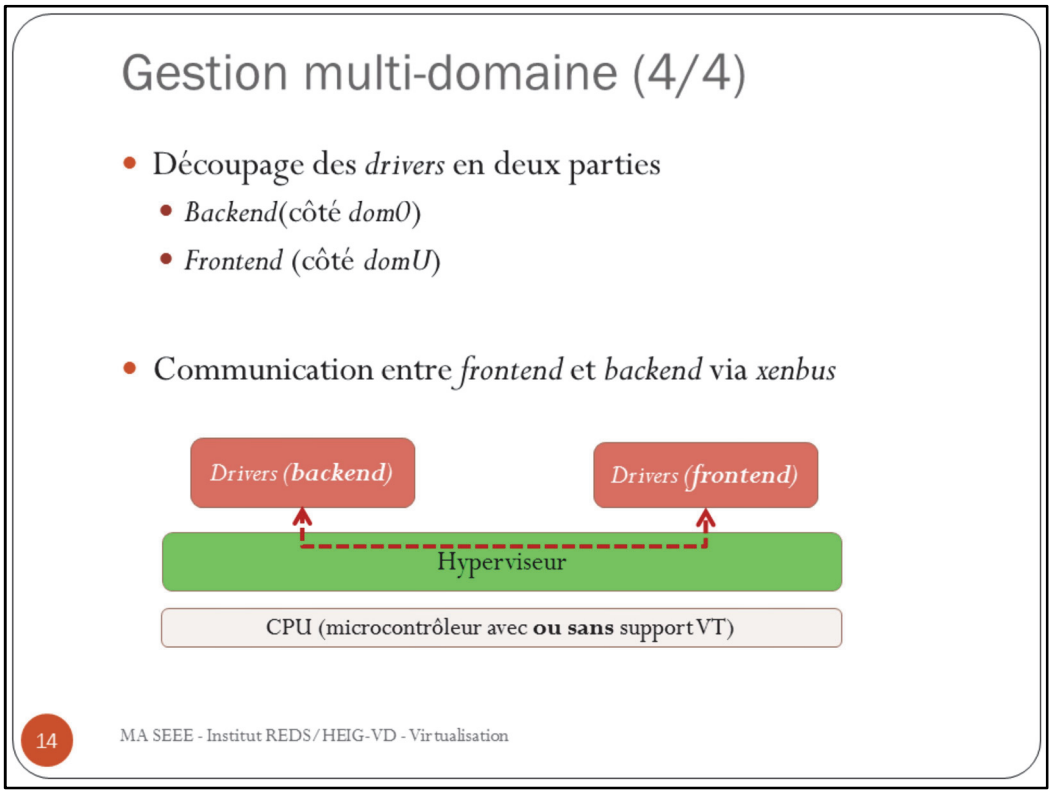
Un second mappage permet ensuite au noyau invité de gérer l'isolation locale inter-processus. Si les tables de pages de ce mappage sont gérées par le noyau invité, l'hyperviseur gère les tables de pages du premier mappage. Le support matériel pour ce double mappage – au niveau de la MMU – est primordiale au niveau des performances. Sans un tel support, l'hyperviseur doit alors gérer une table de translation pseudo-physique/physique et gérer les accès aux tables de pages de manière adéquate, **y compris les tables de pages utilisées par le noyau invité** (mappage des processus notamment). Une approche basée sur la para-virtualisation nécessite ainsi l'utilisation d'*hypercalls* à chaque fois que le noyau invité effectue un changement de contexte mémoire (changement de processus) ou, d'une manière générale, effectue des opérations au niveaux des tables de pages.




Une première fonction de mappage $\sigma$ est initialisée lors du démarrage de l'hyperviseur; l'espace d'adressage virtuel associé contient le mappage (linéaire) de l'hyperviseur ainsi qu'un mappage (linéaire) de toute la mémoire physique (RAM) qui permettra d'implanter les domaines invités. Cette fonction de mappage donne ainsi à l'hyperviseur une "visibilité" sur les différentes domaines.

Dès qu'un domaine prend la main, une nouvelle fonction de mappage ($\sigma'$) permet de décrire la RAM virtualisée allouée au domaine; il s'agit de l'espace d'adressage pseudo-physique. Un changement de contexte entre domaines implique par conséquent un changement de cette fonction de mappage.

Finalement, la fonction de mappage $\sigma''$est celle construite et gérée par le noyau invité permettant l'isolation entre les processus (elle change lors d'un changement de contexte entre processus).




L'utilisation de périphériques communs par les domaines (carte *Ethernet*, *framebuffer*, etc.) nécessite une virtualisation des interfaces matérielles. En effet, les accès en provenance de différentes sources doivent être coordonnés, multiplexés/démultiplexés, synchronisés, protégés, etc.; le périphérique réel, lui, est unique. Par conséquent, les *drivers* de chaque domaine, dans un contexte **non émulé, para-virtualisé**, doivent également subir des adaptations. L'hyperviseur XEN est l'un des premiers hyperviseurs à avoir proposé un découpage des *drivers* tel que présenté ci-dessus, avec une approche para-virtualisée.

Les *drivers* appartenant au domaine *dom0* garde le contrôle sur leurs périphériques; ils gèrent les accès concurrents en provenance des sources multiples. L'une des sources est les accès en provenance de l'autre domaine (*domU*). Ce dernier implémente également une version modifiée de ses *drivers* originaux de telle sorte à ce que tous les accès aux périphériques soient *virtualisés*, c-à-d qu'il n'y ait aucun accès direct à ceux-ci : les requêtes et réponses vers et des périphériques s'effectuent via un bus logiciel entre les deux domaines permettant le transfert de messages. Ce bus est appelé ***xenbus***.

Le *driver* modifié du côté *dom0* correspond au ***backend*** alors que le *driver* côté *domU* correspond au ***frontend***. *Frontend* et *backend* interagissent via *xenbus*.
Ainsi, la *para-virtualisation* des *drivers* consistent à les *"splitter"* en deux: un *frontend* qui exposera une interface virtualisée dans *domU,* et le *backend* qui gardera le contrôle avec le(s) périphérique(s) réel(s).

# Références


- Sebastian Biemüller. **Hardware-Supported Virtualization for the L4 Microkernel**, Universty of Karlsruhe, 2006
- Dong-Guen Kim et *al.* **Design of the Operating System Virtualization on L4 Microkernel**, Fourth International Conference on Networked Computing and Advanced Information Management, 2008
- Paul Barham et *al.* **Xen and the Art of Virtualization**, technical paper from University of Cambridge Computer Laboratory, 2003
- Daniel Rossier. **EmbeddedXEN: A Revisited Architecture of the XEN hypervisor to support ARM-based embedded virtualization**. White paper, June 2012, available at http://en.wikipedia.org/wiki/EmbeddedXEN