IVC

Infrastructures Virtuelles et Conteneurs

Legond-Aubry Fabrice

fabrice.legond-aubry@parisnanterre.fr

Infrastructures Virtuelles et Conteneurs

Principes Théoriques

Plan du Cours

Types de virtualisation

Virtualisation Pure

Conteneurs

Virtualisation

- **Virtualiser** : proposer, par l'intermédiaire d'une couche d'abstraction proche du matériel, une vue multiple d'un matériel unique, en sérialisant les appels vus concurrents de l'extérieur.
- Il existe plusieurs types de virtualisation qui dépendent de différents types de couche d'abstraction.

Virtualisation

Terminologie

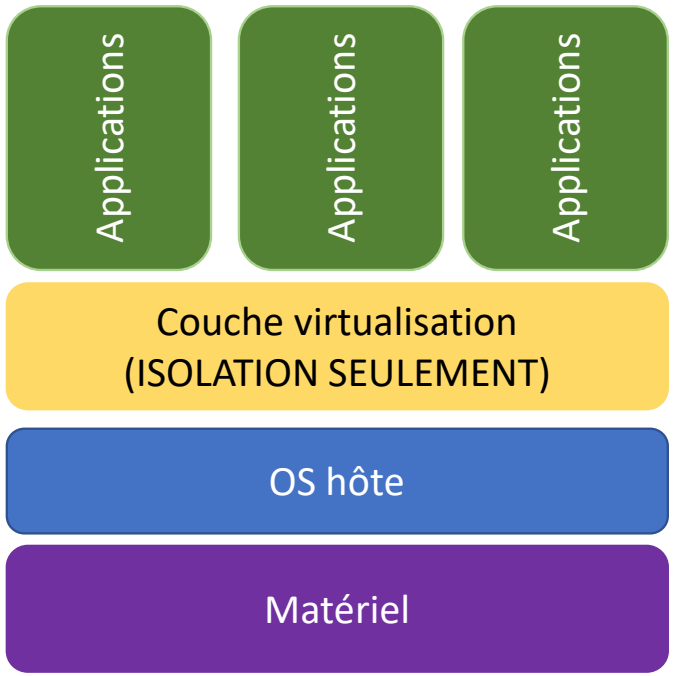
- Le système **hôte** (**host**) est l'OS principal de l'ordinateur.
- Le système **invité** (**guest**) est l'OS installé à l'intérieur d'une machine virtuelle.
- Une **machine virtuelle** (**VM**) est un ordinateur virtuel qui utilise un système invité. On utiliser parfois le terme de **Virtual Device (VD)** pour les systèmes embarqués emulés.
- Un ordinateur virtuel est aussi appelé serveur privé virtuel (**Virtual Private Server ou VPS**) ou environnement virtuel (Virtual Environment ou VE)
- **Hyperviseur** : Noyaux ultra léger permettant la gestion, le monitoring et le diagnostique des VM/VD

Virtualisation

Types de virtualisation #1 :

Virtualisation d'OS ou Isolateur (~conteneurs)

- Notion de conteneur
- Isole l'exécution des applications dans des contextes d'exécution.
- Généralisation de la notion de « contexte » Unix, plus isolation
  - ✓ des périphériques,
  - ✓ des systèmes de fichiers
  - ✓ Ajout d'un filtre entre le noyau et les applications
  - ✓ Isole les applications les une entre les autres
- Solution très performante et économique en mémoire
- Mais Partage du code noyau
  - ✓ isolation plus limité qu'avec une virtualisation complète
  - ✓ Uni-kernel (un seul OS – celui de l'hôte)
- Exemple :
  - ✓ chroot (changement de racine)
  - ✓ Linux Vserver, OpenVZ
  - ✓ (Virtuozzo), Docker, LXC (Cgroups), ...

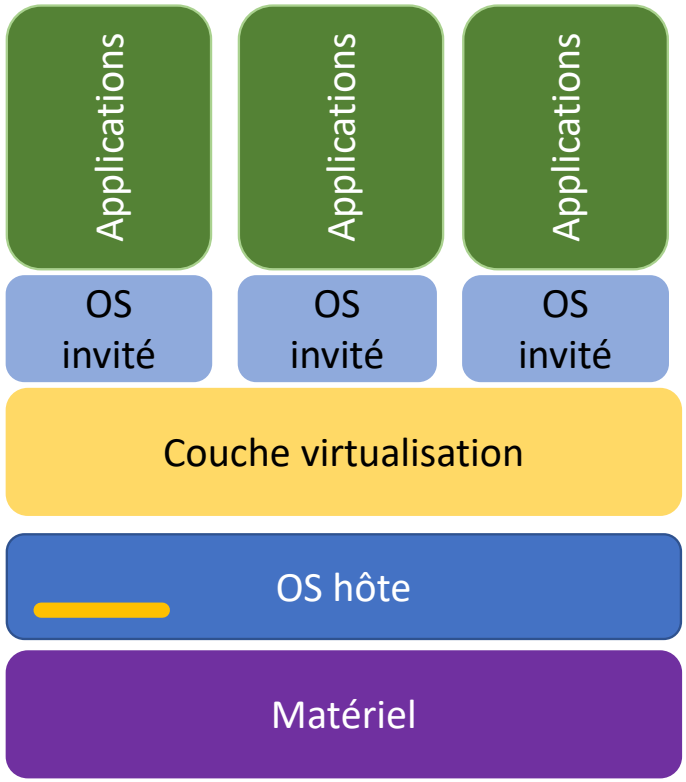


Virtualisation

Types de virtualisation #2 :

Hyperviseur avec Architecture hébergée

- Application installée sur l'OS invité
- Virtualise et/ou émule le matériel
  - ✓ Virtualisation complète d'une machine
  - ✓ Virtualisation "pure"
  - ✓ Multi-Kernel (ie Multi-Os) et multi-materiel
- Comparable à un émulateur mais accès « direct » au CPU, RAM, FS.
- Performances réduites si le CPU doit être émulé
- Bonne étanchéité entre les OS invités.
- Exemples : VirtualBox, QEMU, Vmware (workstation, fusion,player),
- Microsoft Virtual PC, Parallels desktop,…
- Peut emprunter aussi des technologies à la paravirtualisation (cf après) :
  - ✓ Cad Utilisation de drivers spécialisés dans les OS invités

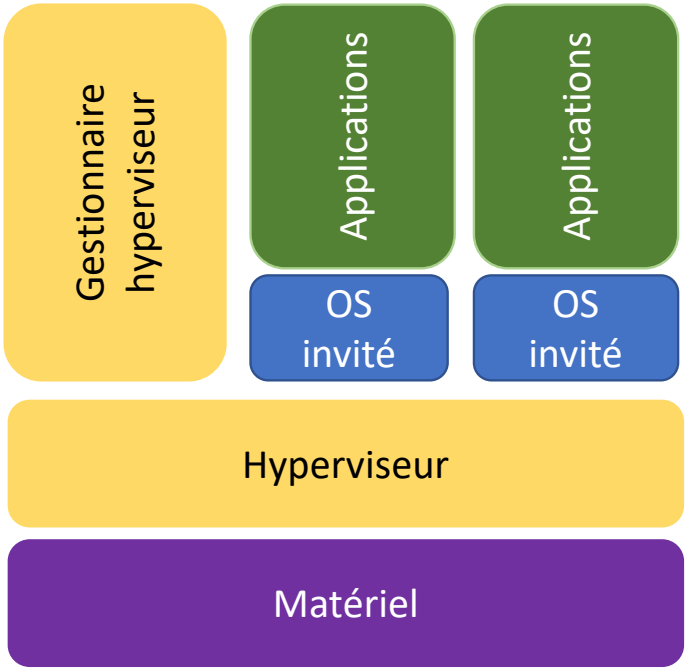


Virtualisation

Types de virtualisation #3 :

Hyperviseur complet

- Création d'un mini système pour gérer les systèmes hébergés
  - ✓ OS de gestion ("hôte") = hyperviseur
  - ✓ Noyau système léger et optimisé
  - ✓ Gère uniquement l'isolation et la répartition des ressources
  - ✓ Parfois l'hyperviseur peut émuler du matériel
- Outils de supervision
  - ✓ Gestionnaire d'hyperviseur
  - ✓ En général un mini-os lui aussi
- Permet l'exécution d'OS natifs (multi-kernel)
- Usage d'instructions dédiées à la virtualisation (sinon émulation).
- Ex: XEN, KVM, Vmware vSphere,…

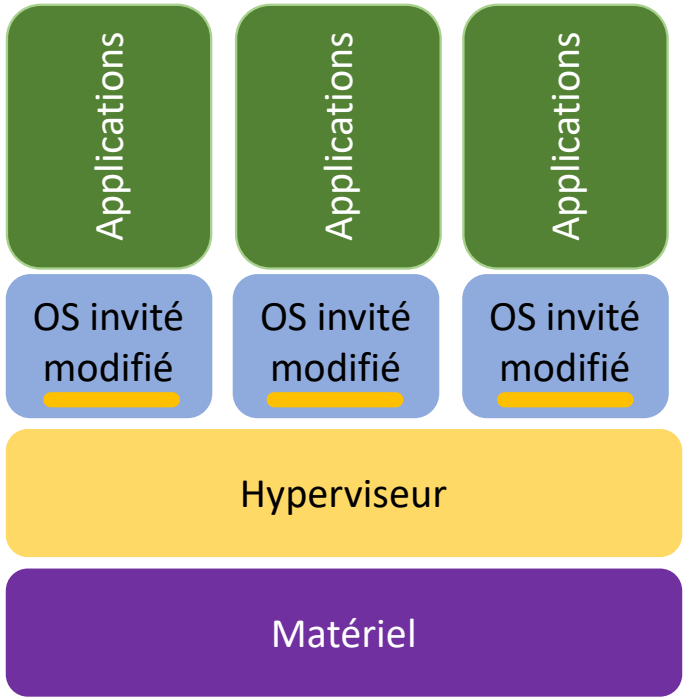


Virtualisation

Types de virtualisation #4 :

Paravirtualisation

- Technologie proche de l'hyperviseur complet
  - ✓ Noyau système allégé et optimisé
- Différence :
  - ✓ Noyau invités adaptés et optimisés (patch + drivers)
  - ✓ Multi-kernel
  - ✓ Utilisable sans les instructions spécifiques des CPUS (ex : VT-x ou AMD-v).
- Souvent impraticables pour les systèmes non libres.
- Exemples : Vmware Vsphere, XEN, Microsoft Hyper-V server, KVM,...

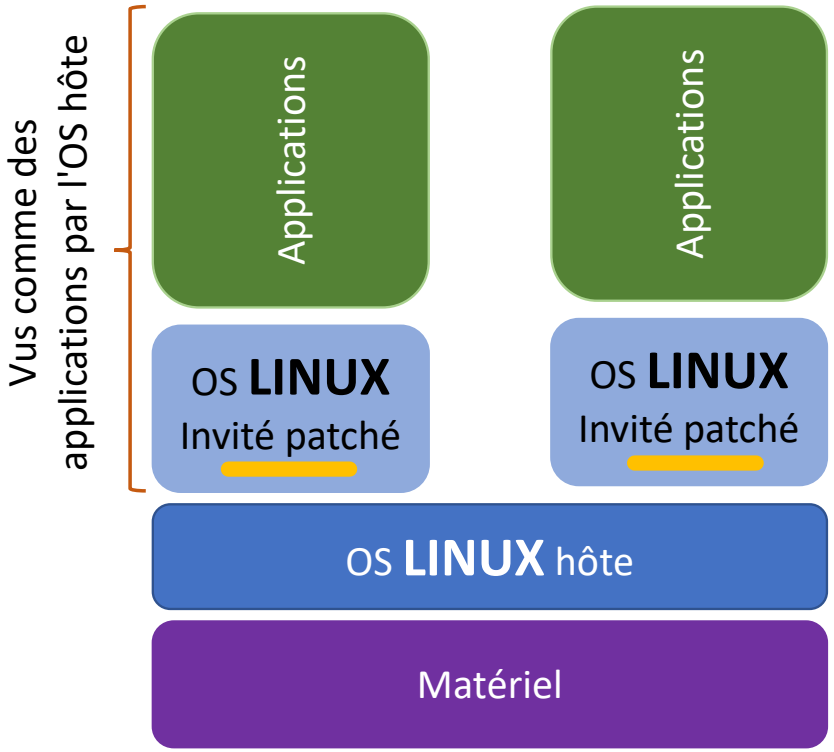


Virtualisation

Types de virtualisation #5 :

Noyau dans l'espace utilisateur

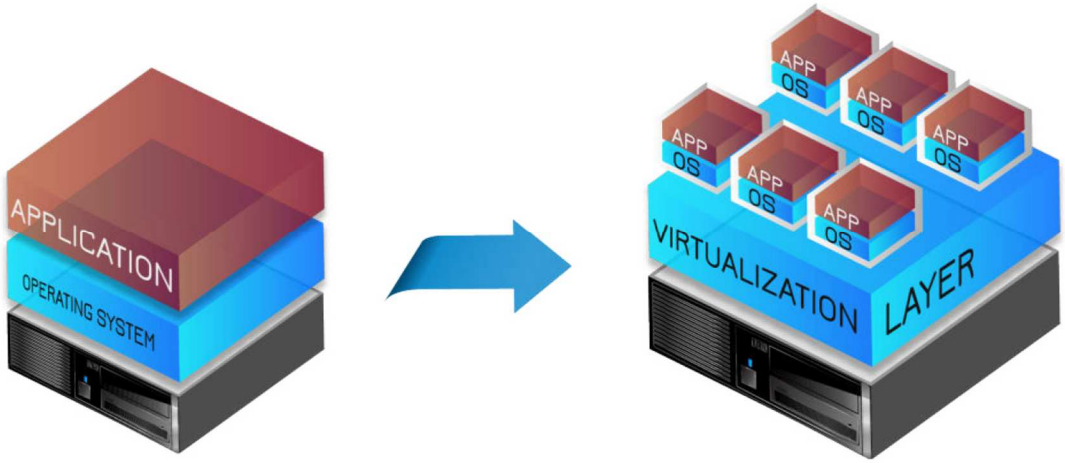
- Pour info : Une 5ème technologie
  - ✓ Hybride, partielle, abandonnée
- Noyau dans l'espace utilisateur
  - ✓ Un noyau linux exécuté comme une application dans le user-space du noyaux linux
- Très peu performant
  - ✓ Empilement de deux noyaux !
  - ✓ Multi-kernel (linux seulement)
  - ✓ Contrôle plus faible car pas de couche d'isolation
- Utile au développement noyau.
- Ex : UML (User Mode Linux) http://user-mode-linux.sourceforge.net/



Virtualisation

PRINCIPE

- Virtualisation pure (Virtualisation de type #2 à #4)
  - ✓ Création d'une machine complète virtuelle
  - ✓ Emulation de tous les éléments d'une machine
    - ➢ CPU, Mémoire
    - ➢ IO: Disque Dur, Carte Graphique, Clavier, Souris
- Possible car :
  - ✓ Puissance exponentielle des machines
  - ✓ Certaines applications n'ont pas besoin de beaucoup de ressources



Virtualisation

Principe de virtualisation "Pure"

- Virtualisation pure
  - ✓ Nécessite ce qu'on appelle un hyperviseur
    - ➢ Gestion des machines virtuelles et leur cohabitations
  - ✓ Nécessite donc l'exécution d'un autre noyau système (n'importe lequel) pour chaque VM créée
    - ➢ Windows, linux, osx, bsd
    - ➢ Surcoût important
  - ✓ Si on émule un matériel "invité" (émulé) n'est pas celui de l'hôte (matériel physique sous-jacent)
    - ➢ Il faut transformer les appels entrants et sortant
    - ➢ Surcoût important encore

Virtualisation

Principe de virtualisation "Pure"

- Virtualisation pure
  - ✓ Pour les VMs pures modernes, on définit (dans la VM invité) du matériel proche du matériel physique.
    - ➢ Evite / Minimise les transformations de codes
    - ➢ Ainsi par exemple, pour les systèmes Android émulé, on utilise des VM ayant le même type de CPU (même jeu d'instructions) que celui de l'ordinateur qui héberge la VMs
    - ➢ Permet des surcoûts minimes et des exécutions proche du code natif
    - ➢ ATTENTION: cela n'élimine pas le surcoût induit par l'exécution des noyaux des VMs
- Si le matériel est proche (similaire) entre le VMs et la machine hôte
  - ✓ Permet des surcoûts minimes et des exécutions proche du code natif
  - ✓ Utilisation des technologie de virtualisation dans le matériel
- Dans certains cas, on peut dédié un périphérique à un VMs
  - ✓ Exemple: une carte réseau

Virtualisation

Principe de virtualisation "Pure"

- Technologies de virtualisation pour le CPU
  - ✓ Doit souvent être activée dans le BIOS/UEFI
  - ✓ Nom: Intel VT-x, AMD-V, …
  - ✓ Les VMs sont donc "adaptées" et pensées pour l'OS sous jacent
    - ➢ Volonté plus sécuritaire : isolation
- Cette technologie permet (sans passer par le couche de virtualisation)
  - ✓ Déléguer automatiquement de portion de code "inoffensives" directement au CPU
  - ✓ D'accéder directement à la mémoire

Virtualisation

Principe de virtualisation "Pure"

- Technologie Intel VT-x / AMD-V
  - ✓ Jeu étendu d'instructions de virtualisation.
  - ✓ Un « super Bios » / UEFI fait l'interface avec la puce.
  - ✓ Permet la cohabitation de plusieurs noyaux bas niveaux simultanés
  - ✓ Simplifie la virtualisation logicielle (ex: sauvegarde de contexte, LAHF/SAHF)
- Technologie Bit NX/XD
  - ✓ NX (Non eXecutable) ou XD (eXecute Disable)
  - ✓ Bit spécial qui permet de marquer des zones mémoires comme non exécutable.
  - ✓ Améliore l'isolation des VM.

Virtualisation

Principe de virtualisation "Pure"

- Technologie de la gestion de la mémoire
  - ✓ Rappel de L3 système : Pour chaque processus, le système créer une carte mémoire linéaire et réparti la mémoire dans la RAM
  - ✓ Si on crée une VM, on lance un 2^{e} système (noyaux).
    - ➢ Pour chaque application du système virtualisé, on fait 2 translations d'adresse
  - ✓ EPT (Extended Page Table) permet d'éviter ces conversions en garantissant la sécurité (isolation)
- D'autres technologies de virtualisation existent pour d'autres matériels
  - ✓ Scalable I/O, GPU (kvmgt), USB, …

Virtualisation

Principe de virtualisation "Pure" : principaux défauts

- Un point de défaillance unique
  - ✓ Une machine physique, plusieurs VMs
- Un recours à des machines plus puissantes
  - ✓ Parfois plus coûteux. Ce défaut tend à disparaitre (multicore).
- Une dégradation des performances
  - ✓ Surcoût des virtualisation, surcoût noyau
- Une complexité accrue de l'analyse d'erreurs
  - ✓ Empilement de couche
- Parfois inadapté (Ex : I/O intense)
  - ✓ Les services de BDs sont plus impactés

Virtualisation

Principe de virtualisation "Pure" : principaux avantages (rappels)

- Sécurité accrue
  - ✓ Isolation, création d'image, mise en suspend
  - ✓ surveillance / audit (log)
  - ✓ Diagnostiques post-mortem (forensic) plus simple
- Usage plus optimal des ressources même sur une seule machine
  - ✓ Mutualisation, contrôle des ressources Mutualisation des services
- Création de machine "kit" qui peuvent se déployer facilement
  - ✓ Gestion de banque de services/OS/applications déployables

Virtualisation

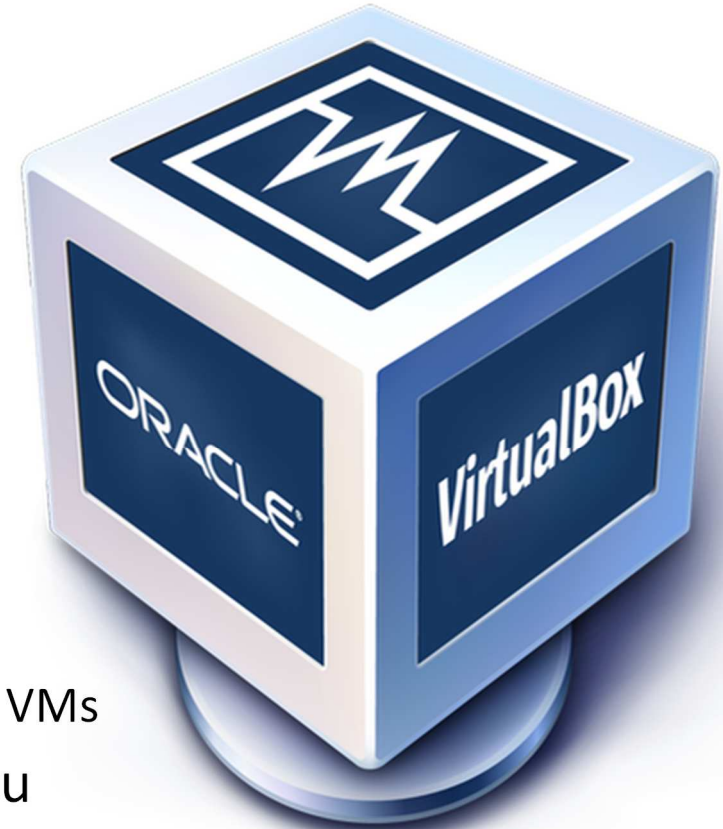
Virtualisation pure: QEMU

- Parmi les premiers logiciels de virtualisation
  - ✓ **OPEN SOURCE**
- Machine virtuelle complète
  - ✓ Techniquement très aboutie
  - ✓ Émulation complète de machine (x86, ARM, MIPS, …)
- L'usage du module kQemu pour une virtualisation accélérée.
- Émulation par recompilation sur un modèle « just-in-time » comme en java
- Gourmand en mémoire
  - ✓ Translation du code assembleur de l'invité vers l'hôte
  - ✓ Translation pour les différents modèles mémoires (invité ← → hôte)
- Sans accélération lent et charge l'hôte
  - ✓ Emulation d'une machine complète

Virtualisation

Virtualisation pure: VirtualBox

- Gratuit mais des problèmes de licences (Oracle)
- Machine virtuelle, émule un PC complet
  - ✓ Support des instructions de virtualisation
  - ✓ Limité aux architectures "intel" (x64, x86)
- Solution de virtualisation efficace
  - ✓ Plus rapide que qemu
- Gourmand en mémoire
- Simple à utiliser
  - ✓ Des efforts vers des outils de commandes en ligne
  - ✓ Scriptage, interfaçage vers des logiciels de gestion de VMs
- Ex : VirtualBox (oracle) et KVM reposent sur Qemu
  - ✓ https://www.virtualbox.org/wiki/Developer_FAQ



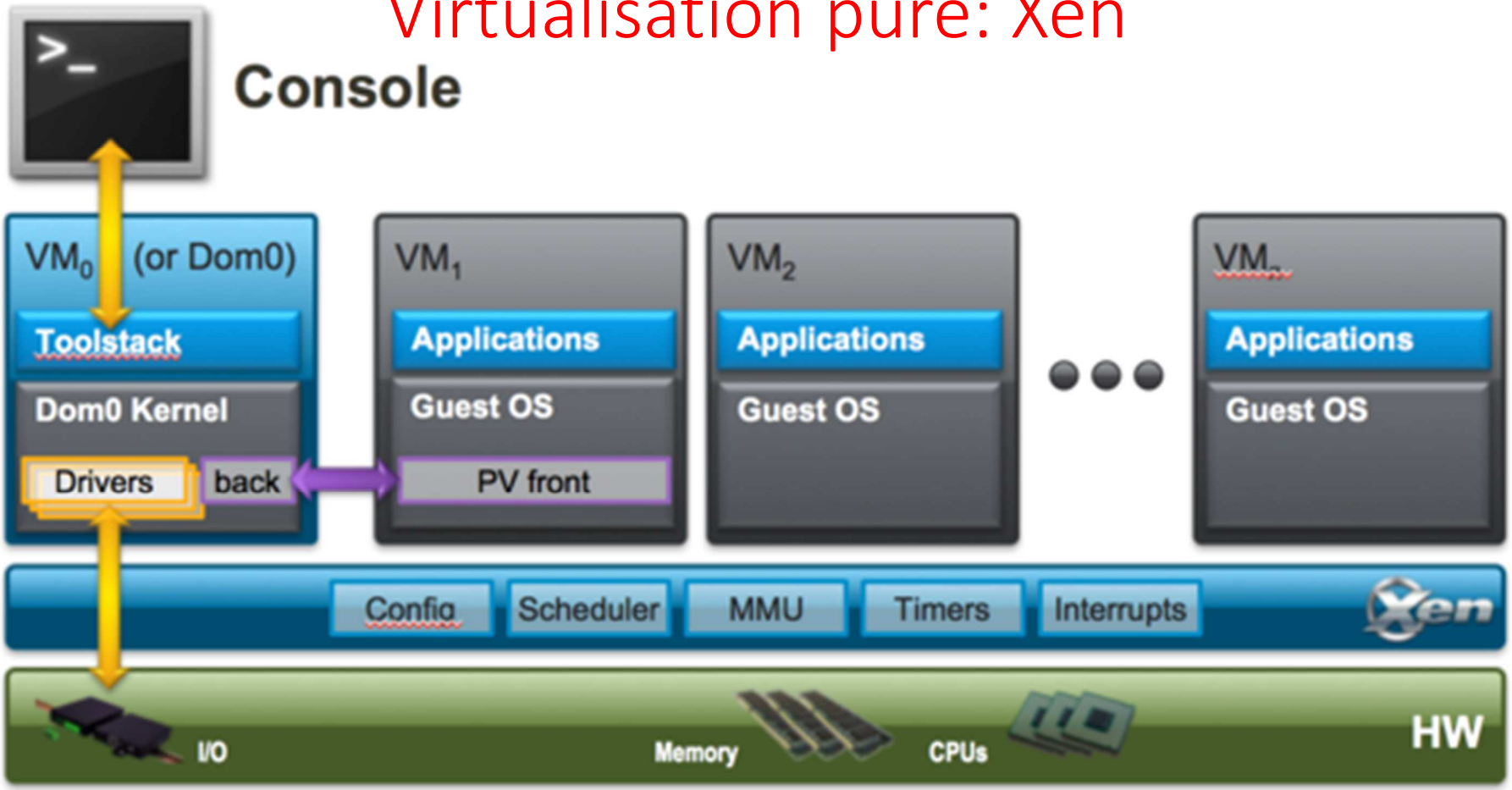
Virtualisation

Virtualisation pure: Xen

- Solution libre ancienne mais encore utilisée
- A base d'hyperviseur !!!
  - ✓ Pas de noyaux "linux"
- Vocabulaire :
  - ✓ OS privilégié : Dom0
  - ✓ OS invités : DomU
- Plusieurs type de DomU
  - ✓ DomU standard (paravirt.)
  - ✓ DomU HVM (hardware assisted)
- Deux modes d'usage :
  - ✓ Paravirtualisation : Noyau speçifique dans le DomU
    Très bonnes performances
  - ✓ Virtualisation matérielle
    Virtualisation transparente pour le système invité.
- Besoin d'un support dans le processeur (AMD-V ou Intel VT)

Virtualisation

Virtualisation pure: Xen

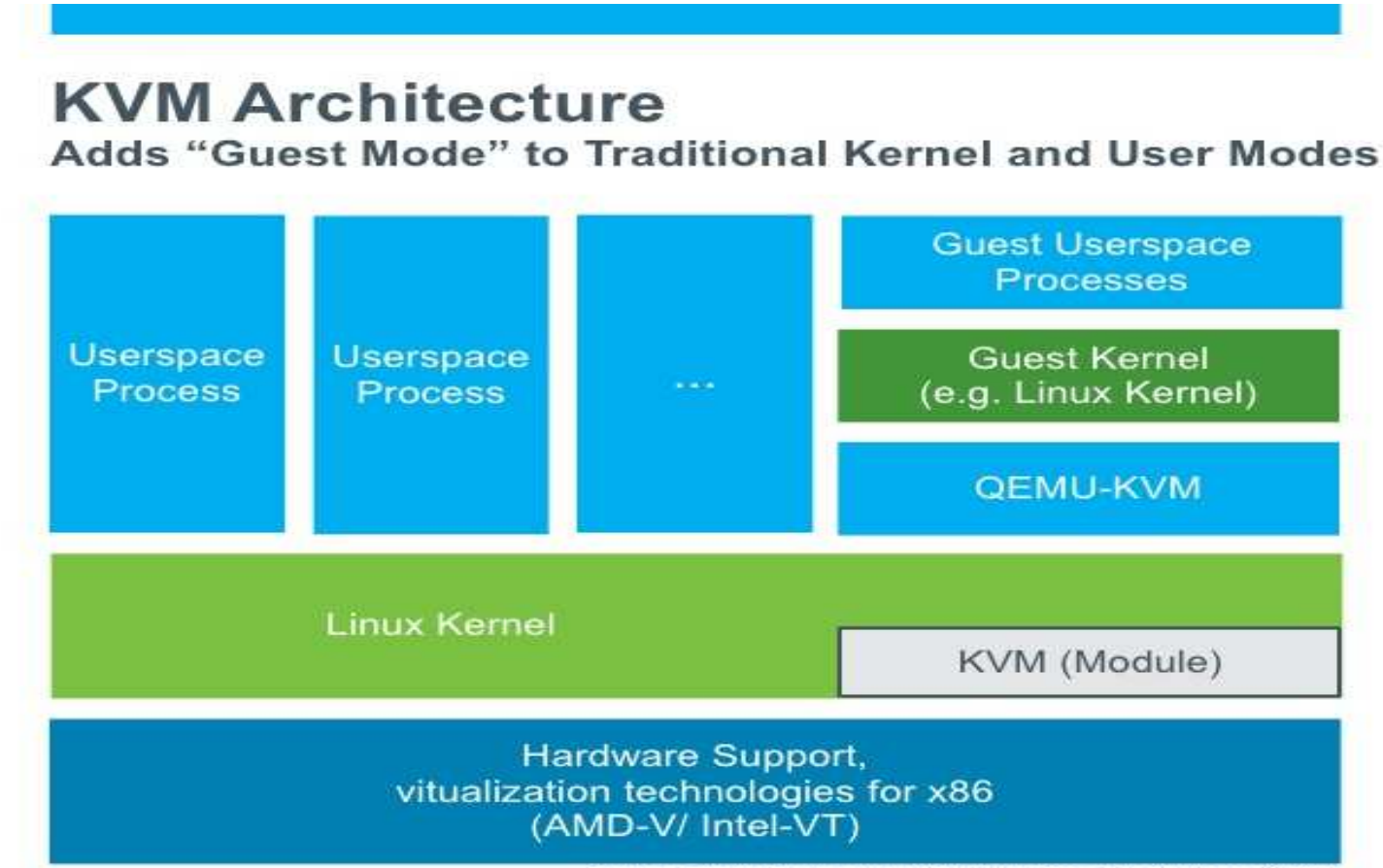


Virtualisation

Virtualisation pure: KVM

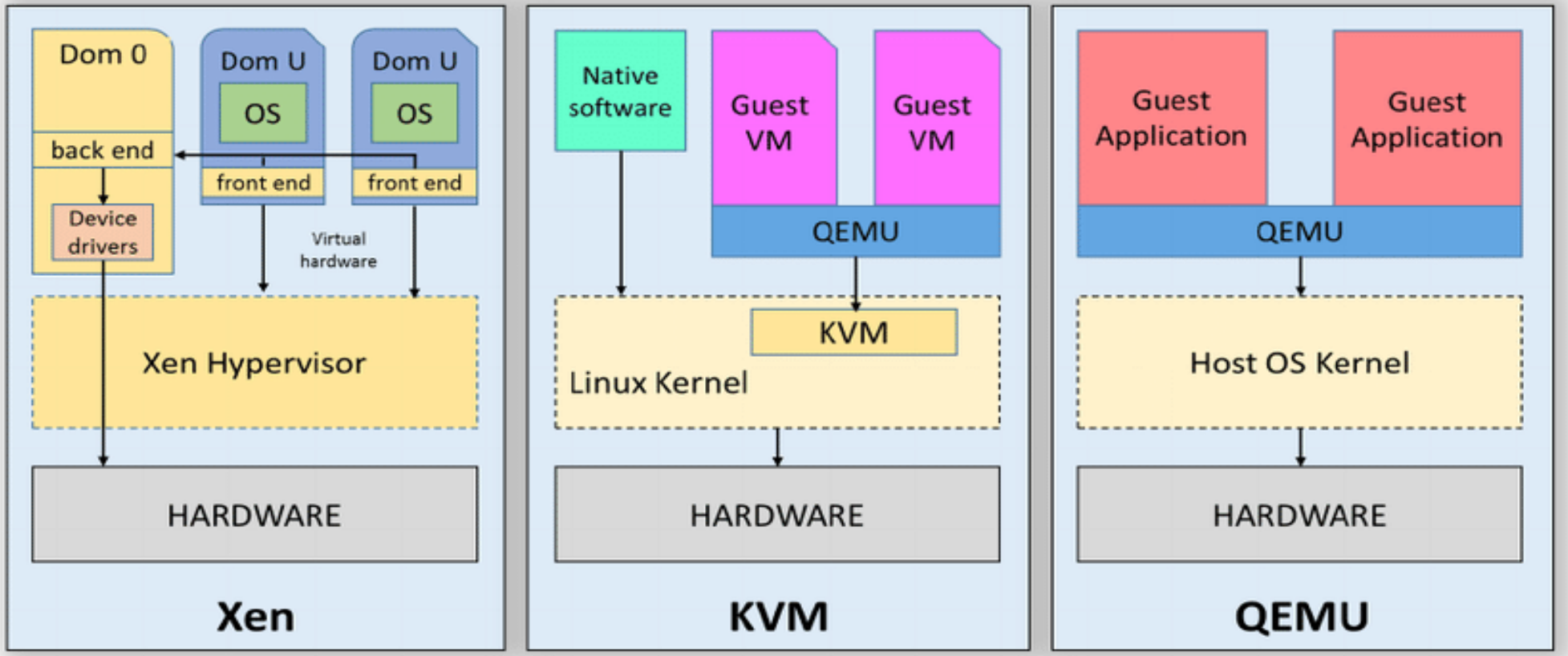
- Projet plus récent que Xen
  - ✓ Très populaire.
- Basé en partie sur QEMU (pour le supports des périphériques)
- KVM pour « **Kernel Virtual Machine** » est une technologie de virtualisation Open Source qui permet de transformer un système linux en un Hyperviseur.
- Entièrement intégré au noyau Linux
  - ✓ Facile à utiliser
  - ✓ Module noyau
- Support de la virtualisation dans les processeurs indispensable.
- Paravirtualisation (virtio) pour les performances.

Virtualisation



Virtualisation

Virtualisation pure: comparaison



Virtualisation

Virtualisation pure: Mécanique interne

- Créer une machine virtuelle impose au minimum :
  - ✓ D'émuler le CPU ou déléguer le code au CPU (utilisation de Intel-VTx ou AMD-V)
  - ✓ De gérer des disques virtuels
    - ➢ Encapsulation des disques dans de gros fichiers ou segments de fichiers
    - ➢ Dépend du FS en dessous
    - ➢ Nécessite la gestion des caches
    - ➢ Nécessite de créer des structures de fichiers permettant l'émulation
    - ➢ Exemple: VDI, vdmk, qcow, …
  - ✓ D'émuler la RAM
    - ➢ Gestion de la mémoire (double mapping)

Virtualisation

Virtualisation pure: Mécanique interne

- Créer une machine virtuelle impose au minimum :
  - ✓ D'émuler le CPU ou déléguer le code au CPU (utilisation de Intel-VTx ou AMD-V)
  - ✓ De gérer des disques virtuels
    - ➢ Encapsulation des disques dans de gros fichiers ou segments de fichiers
    - ➢ Dépend du FS en dessous
    - ➢ Nécessite la gestion des caches
    - ➢ Nécessite de créer des structures de fichiers permettant l'émulation
    - ➢ Exemple: VDI, vdmk, qcow, …
  - ✓ D'émuler la RAM
    - ➢ Gestion de la mémoire (double mapping)

Virtualisation

Virtualisation pure: VMM

- Gestion du CPU : VMM (en général dans l'hyperviseur)
  - ✓VMM = Virtual Machine Monitor
  - ✓Gère les morceaux de code à délégué au CPU
  - ✓Gère l'états des registres du CPU émulé (et du context)
  - ✓Gère l'adéquation en les ordonnanceurs émulés et celui du système hôte
  - ✓Isole/Transforme les appels

Virtualisation

Virtualisation pure: VMM

- La relation entre la machine émulé (guest) et l'hôte est la même qu'entre l'OS hôte et une application
  - ✓ Exécute le code émulé dans un environnement non privilégié
  - ✓ L'accès aux périphériques ne peut se faire directement
    - ➢ Emulation des appels aux périphériques
    - ➢ Interception des appels
  - ✓ Nécessite les transformations des appels/services noyaux
    - ➢ Ex: horloge, …

Virtualisation

Virtualisation pure: (IO)MMU

- Les CPU utilisent une MMU
  - ✓MMU = memory management unit
- Dans l'OS, la MMU gère la vision de la mémoire de chaque application vers la RAM
  - ✓Il existe une table de correspondance nommée la TLB
  - ✓TLB = Translation Lookaside Buffer

Virtualisation
Virtualisation pure: (IO)MMU

- MMU en virtualisation
  - ✓ Emule de la RAM
    - ➢ Valide à la fois pour les virtualisation à base d'hypersiveur natif (type #3) ou OS (type #2)
  - ✓ Pour hyperviseur type #2
    - ➢ Va transformer la vision applicative de la mémoire en RAM **vu par l'hyperviseur** en une vision RAM physique dans la VM
  - ✓ Isole les espaces de VMs entre elles et avec le système hôte
  - ✓ Permet l'indépendance avec le système hôte

Virtualisation
Virtualisation pure: (IO)MMU

- Problèmes :
  - ✓ Pour les hyperviseur de type #2 (hyperviseur OS)
    - ➢ les limitations imposées aux applications par le système s'imposent à l'hyperviseur
  - ✓ Les limitations imposées par la matériels s'imposent au matériel émulé
- Transforme donc les appels RAM des VM en
  - ✓ Mémoire applicative (hyperviseur #2)
    - ➢ Double translation (OS VM←→RAM VM←→ OS Hôte ←→RAM hôte)
  - ✓ En RAM native (hyperviseur #3)
    - ➢ Translation (OS VM ←→RAM VM(hyperviseur) ←→RAM hôte)

Virtualisation
Virtualisation pure: (IO)MMU

- Gestion de la mémoire : (IO)MMU
  - ✓ IO MMU gère le transfert des données entre les périphériques et les processus dans le VM
- Utilisation de la paravirtualisation
  - ✓ Drivers spécifiques dans la VMs
  - ✓ L'utilisation de module dans l'OS (pour hyperviseur #2) pour la paravirtualisation
- Paravirtualisation permet l'accès de la mémoire des VMs aux périphériques de la machine hôte
  - ✓ Elimine la nécessité de la copie de nombreux blocs mémoires
  - ✓ Economise les translations d'adresses
  - ✓ Pratique pour disques, réseaux, graphiques

Virtualisation

Virtualisation pure: accélération MMU + VMM

- Si
  - ✓ vos CPU émulés sont les mêmes que les CPUs (core) hôte
  - ✓ Le cpu hôte disposent d'instructions d'accélération
- L'accélération est importante
  - ✓ Le développement de la MMU et VMM est facilité
  - ✓ Minimise la mémoire nécessaire pour l'hyperviseur/moteur de VM
  - ✓ Accélère encore la transmission entre la VM et les périphériques
- Mais limite les VMs émulés
  - ✓ Il faut les technologies dans le silicium
  - ✓ AMD-V, Intel-vtx, Intel-vti (itanium)
  - ✓ ARM-VE(virtual extension)/LPAE(large physical addr. extension)

Virtualisation

Virtualisation pure: accélération MMU + VMM

- Instruction assez simples mais utiles
  - ✓ VMX Enter / VMX Exit pour entrée dans une section de code virtualisée
    - ➢ Le code est directement exécuté sur le CPU dans un contexte à droit limité
    - ➢ Contexte de gestion de mémoire par VM (sauvé/restauré)
  - ✓ Ajout d'une structure permettant de mémoire d'état d'une VM
    - ➢ Virtual Machine Control Structure (VMCS) per VM
  - ✓ Instructions de gestion de mémoire pour les VMs

Virtualisation

Virtualisation pure: Transfert de ressources

- Avantage: Une VM est une machine complète
  - ✓ Migration simple
  - ✓ Mise en pause simple
  - ✓ Isolation du hardware extrême

- Désavantage :
  - ✓ surcoût assez important même avec paravirtualisation et extensions CPU
  - ✓ Nécessite le moteur d'exécution sur les différents CPU hôte

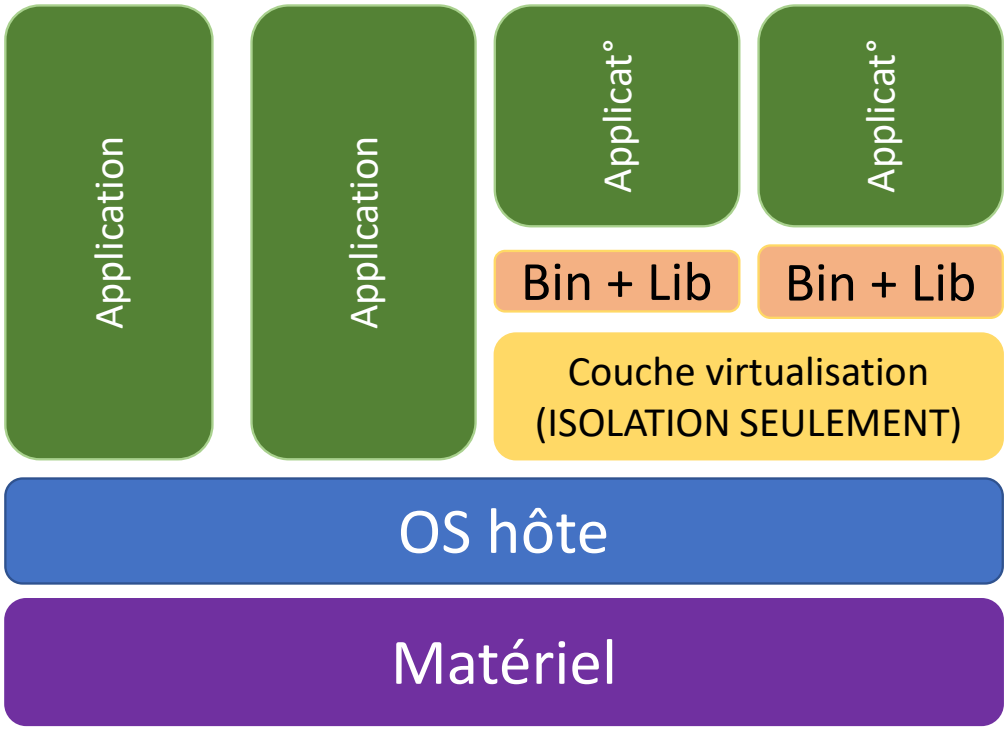
Conteneur

Principe

- Virtualisation de type #1
- Un conteneur à la différence d'une VM n'exécute pas d'OS invité ou de noyaux
  - ✓ Tout reste avec le noyaux système de l'hôte (uni-kernel)
  - ✓ Economise la mémoire
  - ✓ Economise le CPU consommé par tous les noyaux des machines
- Tout cela évite :
  - ✓ les problèmes de partages / accès des périphériques
  - ✓ les problèmes liés aux drivers et leur partage
- Mais impossible de déployer
  - ✓ Une application d'un OS différent
  - ✓ Une application pour un matériel différent
- **On renforce le contrôle de l'accès aux ressources**

Conteneur
Principe

- L'hyperviseur est remplacé par une couche d'isolation
- Rappel : On isole l'environnement, pas le matériel
- Attention cependant, en cas de faille noyau :
  - ✓ L'isolation peut être brisée
  - ✓ Si une attaque passe par un conteneur, il y a évasion
  - ✓ Il faut utiliser aussi d'autres protections du système
  - ✓ Par ex : SeLinux. Voir fin du module.

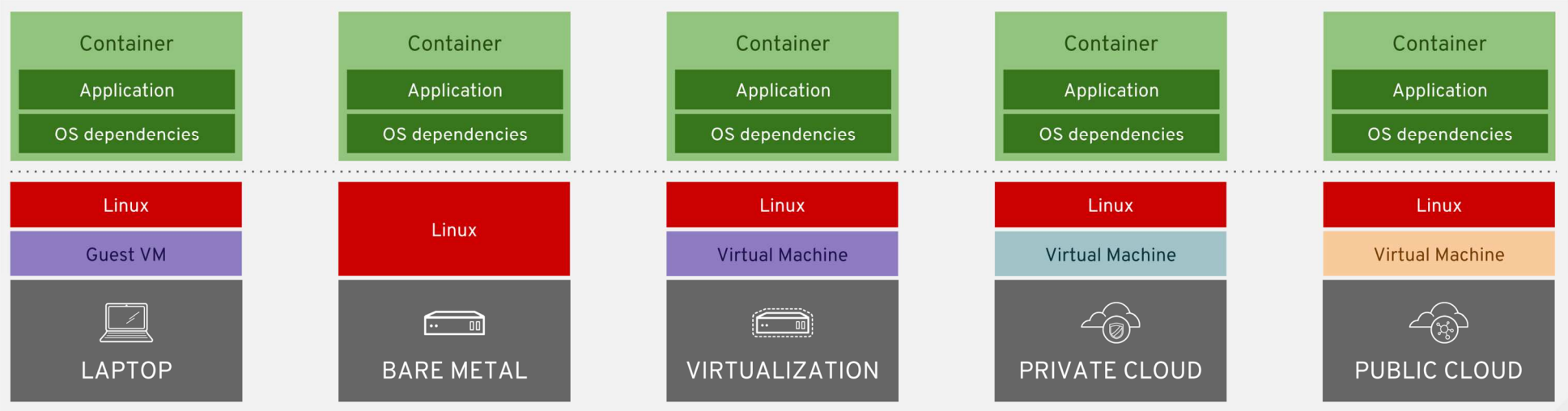


Conteneur
Principe

- Donc l'intérêt principale (pour ne pas dire presque l'unique) :
  - ✓ ISOLATION
  - ✓ Empaquetage d'une application avec toutes ses dépendances
    - ➢ Librairies, code, structure de FileSystem, configuration
    - ➢ Isolé (cad NE VOIT PAS) certaines librairies du système hôtes
  - ✓ Utilise juste le noyaux du système hôte (le contrôle des processus)
- Il peut y avoir des conteneurs fonctionnant :
  - ✓ sous root
  - ✓ sans root (Rootless)
  - ✓ À l'aide de module noyaux pour une meilleurs administration
- NOTE: DOCKER N'EXECUTE PAS UN CONTAINER, C'EST UN GESTIONNAIRE de conteneur

Conteneur
Principe

- Bonne portabilité :
  - ✓ Conteneur Linux + OS Linux Hôte → compatibilité quasi-certaine
  - ✓ Il peut y avoir des services noyaux différents qui pourraient poser pb
- Exemple (avec linux) :



Conteneur
Précurseurs

- Toutes les fonctionnalités des conteneurs repose sur des services noyaux
- Fonctionnalité du noyau Linux pour limiter, compter et isoler l'utilisation des ressources (processeur, mémoire, disque, etc.).
  - ✓ Limitation des ressources
  - ✓ Priorisation
  - ✓ Comptabilité
  - ✓ Isolation
  - ✓ Contrôle
  - ✓ Isolation par espace de nommage
  - ✓ Utilisation de la sécurité de l'OS
  - ✓ Droits utilisateurs, …

Conteneur
Précurseurs : ulimit

- base de l'isolation dans linux: **ulimit**
  - ✓ Configuration via bash (ulimit)
  - ✓ /etc/security/limit.conf
  - ✓ Module pam_limits
- contrôle de base des ressources
  - ✓ Présent dans différents OS
- On peut fixer les ressources en fonction de l'uid

```
#> ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63470
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 4096
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Conteneur
Précurseurs : chroot / pivot_root / mount

- Base de l'isolation dans linux: **chroot / pivot_root / mount**
- **Mount:** voir vos cours systèmes L3
- **Chroot** :
  - ✓ Exécute une commande en changeant la racine apparente du FS et pour TOUS SES ENFANTS
  - ✓ Les chemin relatifs sont calculés par rapport à la nouvelle racine
  - ✓ Les chemins de recherche sont calculés à partir de la nouvelle racine
  - ✓ Créer une cage pour la vue FS
- **Pivot_root** (en complement de chroot) :
  - ✓ Change la racine du FS pour le processus courant
  - ✓ On exécute pas un nouveau processus comme pour chroot, on change celui en cours
  - ✓ On utilise pivot_root en collaboration avec chroot

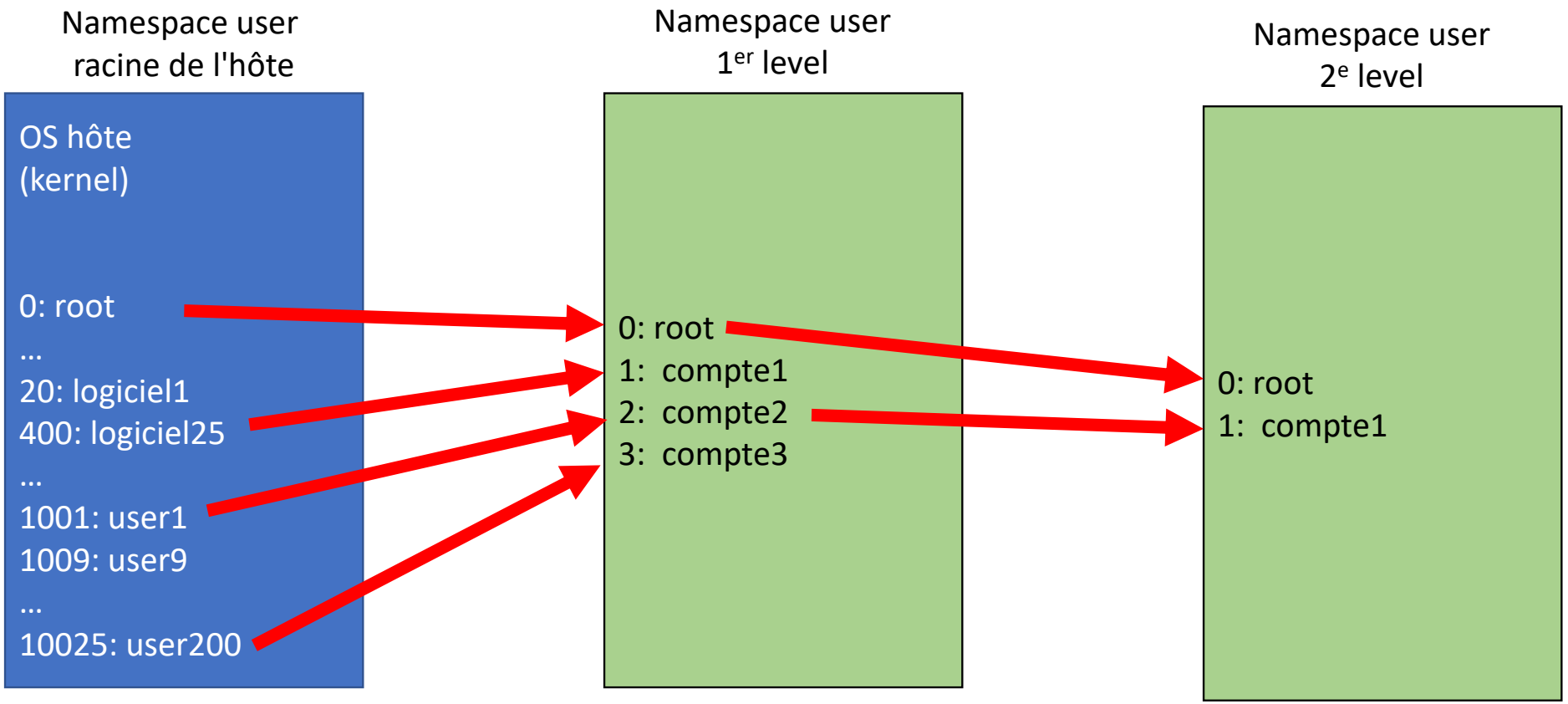
Conteneur
Précurseurs : chroot / pivot_root / mount

- Base de l'isolation dans linux: **chroot / pivot_root / mount**
- On peut s'évader du Chroot
- Très souvent utilisé pour la fabrication des noyaux pour le boot des OS
- Très souvent utilisé pour isoler certains services
  - ✓ historiquement DNS particulièrement attaqué
- **Attention :**
  - ✓ La commande chroot ne fournit aucun outil pour recréer la structure du FS dans l'environnement isolé
  - ✓ C'est à l'utilisateur de recréer / copier les fichiers (librairies) nécessaires à l'application
  - ✓ D'où la création d'outils pour automatiser cette phase (ex : Docker, …)

Conteneur
Précurseurs : kernel namespace

- Base de l'isolation dans linux: les **namespace**
- Linux kernel 3.8+
- Namespace :
  - ✓ Création POSSIBLE d'une vue spécifique des ressources pour chaque processus
  - ✓ Ressources (voir cous système) : IPC, PID (vue des processus), réseau, point de montage, id utilisateurs (UID/GID),
  - ✓ Mais aussi ressources de sécurité : firewall
  - ✓ Possibilité de migrer certaines ressources vers les namespaces (technique !)
    - ➢ Si prévu dans les drivers
- Le **namespace** le plus utile pour les conteneurs (mais pas le seul !) :
  - ✓ User namespace : mapping (correspondance) UID/GID entre un conteneur et le système hôte

Conteneur
Précurseurs : kernel namespace



Conteneur
Précurseurs : kernel namespace / user

- User namespace
  - ✓ Un processus non privilégié (root) peut avoir un accès ROOT limité
- Root dans un namespace a un UID 0 mais ce n'est pas le vrai root
  - ✓ Certains fichiers restent inaccessibles
  - ✓ Ne peut pas modifier le noyaux
    - ➢ Par ex. : interdiction d'insérer des modules noyaux LKM
  - ✓ Interdiction de rebooter la machine
- Pour permettre la correspondance (map) :
  - ✓ Utilisation d'outils newuidmap fournit par les "shadow-utils"
  - ✓ On peut voir les map dans /proc/xxx/uid_map
  - ✓ Fichiers dans /etc/sub[ug]id
  - ✓ On peut SETUID/SETGID des fichiers et des process

Conteneur
Précurseurs : kernel namespace / user

- **/etc/subuid**

```
# first: root uid inside container
# second: start of reserved uid space
# third: end of reserved uid space
1000:4000:5000
```

- **cat /proc/439/uid_map**

```
0          1000       1
1          40005000
```

Conteneur
Précurseurs : kernel namespace / user

- Les tables de correspondance (mapping) est compliqué à maintenir
  - ✓ Problèmes d'héritage des UID/GID pour les user namespace
  - ✓ Grand nombre d'UID/GID

- Alternative :
  - ✓ Un seul UID/GID par conteneur

- Limiter donc les privilèges par newuidmap/newgidmap

Conteneur

Précurseurs : kernel namespace / network

- Un utilisateur peut aussi créer un namespace réseau
  - ✓ Créer des règles iptables (firewall)
    - ➢ Elles seront appliquées aux processus qui sont liés à cet espace
  - ✓ Créer des socket linux (anonymes) isolées
  - ✓ Manipulations réseaux plus complexe
    - ➢ Capture des traces réseaux Tcpdump/wireshark
    - ➢ Virtualisation des réseaux au niveau > à IP –TCP/UDP
    - ➢ …
- Note:
  - ✓ Il est impossible de créer les liens entre les interfaces physiques et les namespace DANS LE CONTENEUR (pb de sécurité)
  - ✓ Cela **impose** donc :
    - ➢ la création AVANT le déploiement du namespace et de la configuration
    - ➢ La création du lien entre l'interface physique, et le namespace hors du conteneur
    - ➢ Exemple: outils slirp4netns
- Voir la partie sécurité de ce module

Conteneur
Précurseurs: cgroups

- Cgroup v1 : Introduit avec kernel 2.6.24
- Cgroup v2 : Introduit avec kernel 4.5
- Cgroup : système de fichier pour la réification du contrôle des ressources d'un ensemble de processus. Cad:
  - ✓ Chaque fichier permet de configurer les ressources pour un groupe de processus (un **slice** avec systemd)
  - ✓ Ecrire dans un fichier change la configuration du groupe
  - ✓ Lire dans un fichier indique la valeur de configuration en cours
- Système de fichier : /sys/fs/cgroup

Conteneur
Précurseurs: cgroups

- Il existe deux version (cgroup v1 et cgroup V2). Olala !!!!
  - ✓ Changement des les structures de fichiers (path)
  - ✓ Passage de ./resource/mon_cgroup vers ./mon_cgroup/resource (plus logique)
  - ✓ En général (ca peut varier en fonction des distribution), on a
    - ➢ /sys/fs/cgroup qui est la vision 1
    - ➢ /sys/fs/cgroup/unified qui est la version 2
    - ➢ Note: fedora 31+ monte la version 2 directement dans /sys/fs/cgroup
  - ✓ **Cgroups a été associé dans sa version 2 aux namespaces !!!**
- On peut associer des users ou des processus à des cgroup via encore une fois des modules pam.d (pam_cfgs)

Conteneur
Précurseurs: cgroups

- Contrôle :
  - ✓ Limitation des ressources par groupe (ulimit like)
  - ✓ Fixation limite sur Mémoire, CPU, accès I/O, Attribution de "core"
  - ✓ Contrôle accès périphérique
  - ✓ Priorisation : CPU, mémoire
  - ✓ Monitoring : CPU, mémoire
  - ✓ Dé/Gèle de processus, ajout de check points
  - ✓ Réseau: Injection, tag de paquets, priorisation
  - ✓ Gestion des namespaces pour le cgroup
- Pas toujours fournit / chargé par défaut mais quasi obligatoire pour les conteneurs
- Liens
  - ✓ https://www.kernel.org/doc/Documentation/cgroups/
  - ✓ https://docs.fedoraproject.org/en-US/Fedora/17/html/Resource_Management_Guide/ch01.html

Conteneur
Précurseurs: cgroups

- Rappel: cgroup v1 et v2 peuvent cohabité mais on **ne peut pas** contrôler un élément (mémoire, pids, cpu) à la fois dans la version 1 et la version 2

Conteneur
Précurseurs: pb de gestion de fichiers

- Les conteneurs ne sont pas toujours déployés par "root"
  - ✓ Pas d'accès au moyen de contrôles des uid/gid (devicemapper, …)
- Les fichiers posent beaucoup de problèmes
  - ✓ Le chroot n'est pas forcement suffisant
  - ✓ PB de partage de ressources entre les conteneurs
  - ✓ PB de partage de ressources entre un conteneur et l'hôte
  - ✓ Duplication de fichiers (grosse consommation d'espace)

Conteneur
Précurseurs: pb de gestion de fichiers

- Des "workaround" pas au top:
  - ✓ Certains système de fichiers autorisent des sous volumes (BTRFS)
  - ✓ Utilisation des liens de fichiers (BTRFS, XFS, …)
  - ✓ Module utilisateur de fichiers (fuse mount) – via "**fuse-overlayfs**"
- Fuse-overlayfs
  - ✓ Fuse → utilisable par les utilisateurs simples
  - ✓ Mutualisation des fichiers (dé-duplication)
  - ✓ Mapping uid/gid entre un niveau n et n+1 de conteneur
  - ✓ Pt négatif : Complexe
- L'overlay-fs va mentir sur le propriétaire (group/user) des fichiers
  - ✓ Evite chown/chmod/cp -r

Conteneur
Définition

- L'ensemble de ces technologies dans l'OS forment les briques du "moteur d'exécution" des conteneurs.
  - ✓ Outils linux: Mount, chroot, namespace, Cgroup, overlay-fs
  - ✓ Il existe des choses similaires pour les autres OS

- Constitution et définition d'un conteneur
  - ✓ Un fichier de configuration pour le déploiement
  - ✓ Une image AUTOSUFFISANTE (code compilé et avec ses librairies)
  - ✓ **Au final, un conteneur est une instance en exécution d'une image selon une configuration**

Conteneur
Définition

- Une image AUTOSUFFISANTE (code compilé et avec ses librairies)
  - ✓ Les libraires (dont certaines librairies considérées comme système)
    - ➢ Exemple : glibc, libssl, stdlib, …
  - ✓ Binaires de votre(vos) application(s)
    - ➢ Exemple : http, python, …
  - ✓ Et/ou packages par un gestionnaire
    - ➢ Exemple de gestionnaires : apt, urpmi, yum, …
    - ➢ Exemple de packages : rpm, deb, …

- Le contenu dépend des devs, de l'appli et de la politique de dev
  - ✓ 1 application, multiples applications coordonnées
  - ✓ Aucune application (data)
  - ✓ → nécessité de l'intégrer au workflow de développement

Conteneur
Définition

- Notion de "Configure once … run anything"
  - ✓ Installation répétable
  - ✓ Elimination des inconsistances entre dev, test, post-prod, prod/customer
  - ✓ Facilite les MAJ en séparant data avec l'environnement et le code

- Séparation des rôles
  - ✓ Dev se concentre sur le contenu
    - ➢ code, libs, package manager, liens vers data
  - ✓ DevOps se concentre sur l'extérieur, sur le contenant
    - ➢ log, accès et configuration réseau, monitoring, gestion (migration, start/stop, …)
  - ✓ L'ensemble forme le conteneur avec un membrane d'isolation

Conteneur
Définition: conteneur rootless

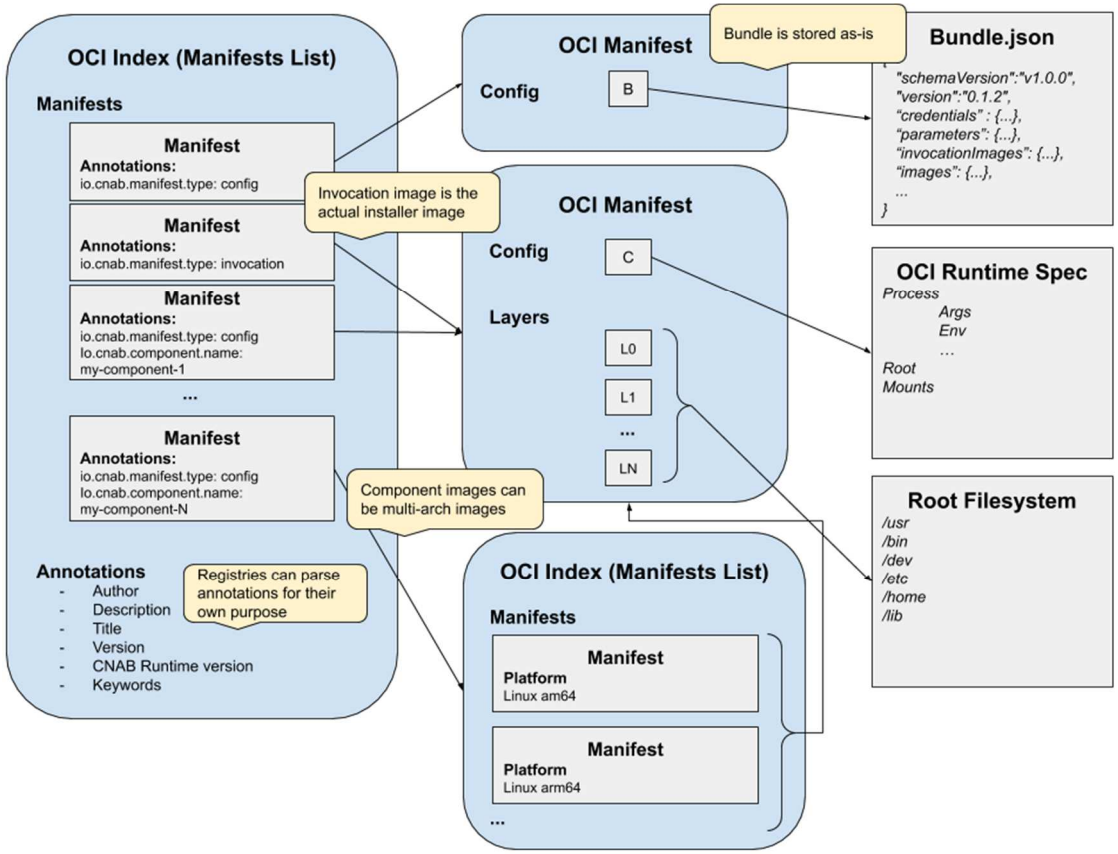
- **Les conteneurs "rootless"** se réfèrent à la possibilité pour un utilisateur non privilégié (non root) de créer et manipuler des conteneur et par l'intermédiaire d'outils
  - ✓ De manipuler les ordonnanceurs
  - ✓ De contrôler les ressources
  - ✓ Différence entre Docker (root) VS buildah/podman (rootless)

Conteneur
Définition : création d'image (OCI)

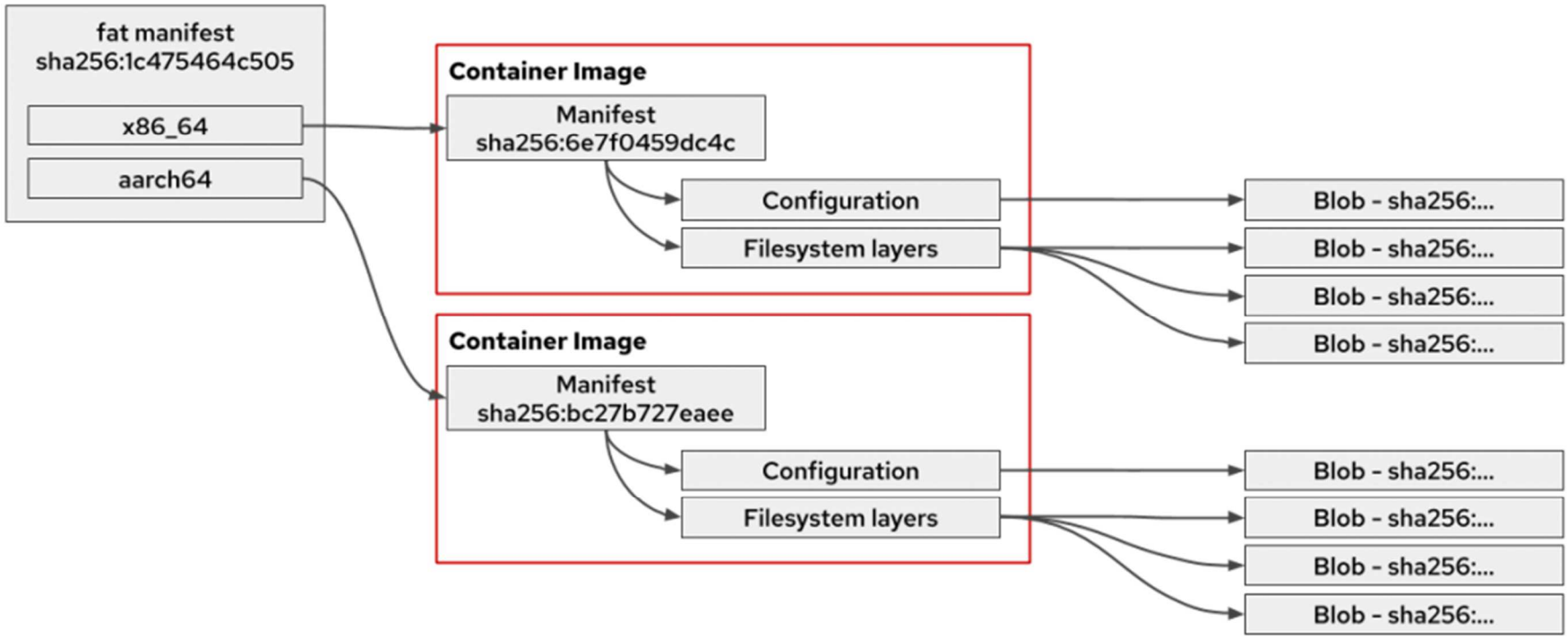
- Les images peuvent être créées en interne
  - ✓Il s'agit d'une étape supplémentaire à la compilation
  - ✓Création d'artéfact "image conteneur" dans le workflow
  - ✓Voir les OCI (open container initiative), l'intégration à gitlab/github
  - ✓https://opencontainers.org/
  - ✓Voir plus tard dans le module
- Notes : Des images d'applications existent
  - ✓**IL EXISTE DES "REPOSITORIES"** (dépôts)

Conteneur

Définition : création d'image (OCI)



Conteneur
Définition : création d'image (OCI)



Conteneur
Définition: image registry

- Il existe des "image registry" (repository) globales
  - ✓ https://hub.docker.com/
  - ✓ https://quay.io/repository/prometheus/node-exporter
  - ✓ https://jfrog.com/container-registry/
- Il existe des "image registry" par les acteurs majeurs du cloud computing
  - ✓ https://azure.microsoft.com/en-us/services/container-registry/ (ms)
  - ✓ https://gallery.ecr.aws/ (amazon)
  - ✓ https://eu.gcr.io (google)
- Il existe des "image registry" spécifiques
  - ✓ Fedora : https://registry.fedoraproject.org/
- Il existe des outils de "probing"/requêtage des OCI
  - ✓ https://github.com/containers/skopeo (voir aussi les autres outils)
- Il existe des "loueurs" d'architectures (cloud) pour les OCI

Rappel :Conteneur VS Machine Virtuelle

- synthèse virtualisation VS container

Outils (bilans)

- Outils d'exécution de VMs "Hypervisor on OS" (VM type #2)
  - ✓ VmWare Desktop, VirtualBox, VirtualPC (ancien), Qemu
- Outils d'exécution de VMs "Hypervisor on hardware" (VM type #3)
  - ✓ KVM, VmWare ESX, Xen
- Outils de gestion / Manipulation des VMs
  - ✓ Docker, Vagrant
- Outils de configuration automatique (scripté) d'OS / conteneur
  - ✓ Ansible, Chef, Puppet

Outils (bilans)

- Outils "d'exécution", de gestion / Manipulation des conteneurs
  - ✓ Docker (windows/linux/osx): conteneur rootless+rootful, daemonful
  - ✓ Singularity (linux/win+osx via vragant box): rootless+rootful, daemonless
  - ✓ podman + buildah (linux) : conteneur rootless, daemonless
  - ✓ LXC (linux/solaris/bsd) – conteneur "rootful+rootless", daemonful
  - ✓ Kubernetes (linux/osx/win) - conteneur "rootful+rootless (via usernetes)", daemonful
  - ✓ Cri-O (with Kubernetes) – API d'"exécution" des conteneurs sur le cloud

- Outils de gestion / Manipulation des conteneurs
  - ✓ Docker, Vagrant

Outils (bilans)

- Règles
- 2014-dec-03-midi-dev-docker-1412050735557-conversion-gate01