KVM

- Kvm -Détails
- Stream audio
- Virsh - Commande
- virt-install
- VM - passthrough pci gpu
- Gestion réseau
- Kvm FS

Kvm -Détails

KVM : focus sur l'implémentation d'un hyperviseur dans Linux

Magazine

Marque

GNU/Linux Magazine

HS n°https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-087

Numéro

87

|

Mois de parution

novembre 2016

|

Auteurs

Thierry Philippe

https://connect.ed-diamond.com/themes/connect/images/creative-commons.png

Domaines

Système

Résumé

Cet article décrit l'implémentation noyau de KVM, ses interactions avec le matériel et les contraintes associées sur les architectures Intel et ARM. Il a également pour but de décrire ce qui n'est pas du ressort du module, et quels rôles ont réellement Qemu, le noyau et la libvirt dans l'exécution d'une machine virtuelle.

Body

Cet article a pour but de décrire comment est implémenté **KVM**, ses interactions avec le matériel avec l'écosystème logiciel de l'équipement. Le but est de montrer les concepts derrière l'implémentation de KVM à la fois sur x86, mais également sur les cibles embarquées. Les éléments de gestion ne sont pas traités, au profit des couches basses du logiciel et des API associées.

1. Architecture macroscopique de KVM

Pour commencer, je vous propose quelques définitions de termes employés dans cet article :

- KVM : *Kernel-based virtual machine*. KVM est une implémentation logicielle d'un hyperviseur *bare-metal* (classiquement nommé « type 1 » **[1][2]**) ;

- Qemu : Qemu est un logiciel d'émulation (Qemu pour *Quick Emulator*), en charge d'émuler une machine physique (séquence de boot, périphériques, etc.) ;

- hyperviseur, VMM : un hyperviseur (ou VMM, *Virtual Machine Monitor*) est une fonction logicielle et/ou matérielle en charge de créer et d'exécuter des machines virtuelles ;

- **U-boot** : U-Boot est un *bootloader* Open source, implémenté par **DENX** software. Il se substitue, dans les architectures embarquées (ARM, PowerPC, etc.) à une partie du BIOS (qui n'existe pas dans ces dernières) et à **GRUB** ;

- *Full-virtualization* : virtualisation complète, permettant d'exécuter un environnement logiciel complet sans modification. Peut se faire de deux manières :

** à l'aide du matériel, on parle alors de HVM (*Hardware-assisted Virtualization*). Ce dernier est alors apte à différencier un contexte virtuel d'un contexte natif, et apporte des abstractions au niveau du matériel pour permettre l'exécution d'un système d'exploitation sans modification de ses instructions ;

** sans aide du matériel. Cas historique, plus difficile à réaliser et coûteux (utilisation de mécanismes de *« binary translation »* pour détecter toute utilisation d'instructions invalides en

avance de phase). L'arrivée du HVM a permis de fortement simplifier l'exécution non modifiée d'environnements logiciels ;

- para-virtualisation : virtualisation avec l'aide du logiciel virtualisé. Ce dernier est modifié pour faire des requêtes explicites à l'hyperviseur à certains moments de son exécution. Diverses fonctions peuvent être para-virtualisées :

** le processeur (la MMU, accès à divers registres d'états, etc.) ;

** les périphériques (exemple de **VirtIO**, que nous verrons plus loin).

- OS invité : système d'exploitation exécuté dans une machine virtuelle ;

- OS hôte : système d'exploitation, lorsqu'il existe, s'exécutant en dehors du contexte virtuel.

1.1 Le module KVM

Le module KVM est le composant logiciel en charge de gérer les fonctions matérielles de gestion de la virtualisation ainsi que les interactions avec les composantes d'hypervision du noyau Linux dont il n'est pas le gestionnaire (comme l'ordonnancement des machines virtuelles). Selon les architectures matérielles, il peut être le seul composant logiciel à s'exécuter avec les droits d'accès aux API de virtualisation matérielle (c'est le cas sur ARM par exemple). Sa volumétrie de code est relativement petite au regard du noyau Linux dans son ensemble (de l'ordre de 50 000 lignes de code pour le support de l'ensemble des extensions x86, de l'ordre de 15 000 pour le support ARM sur un noyau 4.7.0).

1.2 Le rôle de Qemu

Qemu est un émulateur... et continue de l'être y compris dans le cadre de KVM. Son but est d'émuler un certain nombre de fonctions que le module KVM n'est pas capable de gérer par lui-même :

- la séquence de boot du matériel virtualisé : exécution d'un BIOS ou d'un X-loader, nécessaire pour pouvoir transmettre un certain nombre de structures de données en mémoire au bootloader (Grub dans le cas x86, U-boot sur ARM) ;

- l'émulation potentielle de divers périphériques (contrôleurs Ethernet, contrôleurs de disque et ainsi de suite) ;

- les communications inter-VMs ou, avec le noyau de l'OS hôte ou avec des périphériques physiques *remappés*, via des *threads* d'I/O, en utilisant des IPC standards.

Globalement, Qemu gère donc les séquences de démarrage/extinction et les I/O lorsque celles-ci ne correspondent pas à des accès directs à un périphérique physique ou *virtualisable*

matériellement. Bien que la présence du processus **qemu** au-dessus du noyau de l'hôte donne l'impression que le noyau de la machine virtuelle s'exécute dans un contexte applicatif, il s'agit d'un faux-ami. Qemu peut être vu comme un élément à positionner comme une fonction d'émulation de certaines I/O positionnées à côté du contexte effectif de la machine virtuelle. Il se comporte alors comme une fonction de proxy d'I/O pour l'OS invité.

Il faut bien voir que Qemu n'est pas strictement interconnecté à KVM. Le module KVM fournit une API permettant à toute solution applicative apte à émuler le démarrage d'une machine physique et à *proxifier* des entrées-sorties de venir se positionner en lieu et place de Qemu.

1.3 Et la libvirt dans tout ça ?

La **libvirt** apporte le plan d'administration, de contrôle et de management local de l'hyperviseur. Elle permet de gérer la création, le déplacement, la destruction et plus généralement le cycle de vie des machines virtuelles. Elle n'est pas spécifique à KVM et n'entre pas en jeu dans l'exécution effective de la machine virtuelle. La libvirt reste hors du sujet de cet article, mais il faut globalement la voir comme un plan d'administration, de supervision et de contrôle. Elle fait également le lien avec la gestion des réseaux virtuels (e.g. via **OpenVSwitch**) et est l'une des briques de contrôle, de supervision et d'administration nécessaires aux architectures *Cloud* type **OpenStack**.

2. KVM et les extensions matérielles d'aide à la virtualisation

2.1 Petit historique du HVM

La virtualisation aidée du matériel (HVM) est arrivée globalement en même temps sur Intel (VT-x) et AMD (initialement appelé AMD Pacifica, aujourd'hui AMD-V). À ses débuts, les mécanismes matériels d'aide à la virtualisation ont entraîné un certain rejet de la part des fournisseurs de solutions de virtualisation, **VMWare** en tête **[3]**, qui la jugeait trop rigide et moins performante que ses propres implémentations logicielles.

Néanmoins, et VMWare l'a tout à fait accepté depuis, les évolutions successives et les nouvelles fonctions d'aide à la virtualisation portées par le matériel ont permis de simplifier les implémentations logicielles des hyperviseurs, au profit d'un accroissement de complexité dans le matériel.

Historiquement les premiers, AMD et Intel ne se sont jamais mis d'accord sur une API commune, ce qui a impliqué pour les fournisseurs d'hyperviseurs (KVM compris) d'implémenter le support des deux API en parallèle.

De nos jours, la richesse et la modularité de l'API matérielle d'aide à la virtualisation génèrent une certaine complexité dans l'implémentation de l'hyperviseur, ce dernier devant vérifier pour chaque fonction d'aide à la virtualisation si celle-ci est supportée par le matériel ou s'il doit intégrer une gestion logicielle de la fonction. C'est le cas des EPT ou des *Shadow Page-Tables* Intel, décrites plus loin.

2.2 Étude des API Intel

2.2.1 L'API VT de virtualisation du cœur processeur

L'API de virtualisation VT-x, permettant la virtualisation de l'exécution du logiciel sur un cœur processeur, a été pour la première fois implémentée en 2005 dans les Pentium 4 modèles 662 et 672. Ce support est visible sous le nom **vmx** dans le fichier **/proc/cpuinfo** sous GNU/Linux :

$ cat /proc/cpuinfo |grep vmx | head -1

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl **vmx** est tm2 ssse3 fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt

La principale modification est la création d'un mode supplémentaire dans le « ring » **0** (terminologie spécifique à x86), que l'on trouve classiquement dans la littérature sous le nom de mode **vmx-root**, ou mode hyperviseur.

Pour rappel, une architecture x86 standard possède quatre niveaux d'exécution, allant de **0** à **3** (du plus élevé au plus faible en termes de droits d'accès). Classiquement sous GNU/Linux, le noyau s'exécute en ring **0**, les processus applicatifs en ring **3**. L'apparition du mode **vmx-root** permet de séparer le niveau d'exécution du noyau en deux modes :

- un mode dans lequel les instructions de gestion de la virtualisation sont accessibles : le mode **vmx-root** ;

- un mode dans lequel les instructions de gestion de la virtualisation ne sont pas accessibles et où le matériel considère que le logiciel est cloisonné dans un environnement virtuel distinct du mode **vmx-root** : le mode **vmx non-root**.

Au démarrage de l'équipement physique, lorsque le mode VMX est activé par le BIOS ou l'UEFI, ce dernier passe dans le mode **vmx-root**, ce qui permet au noyau Linux de pouvoir créer des machines virtuelles. Le support du mode **vmx-root** est cependant désactivable dans le BIOS ou l'UEFI, désactivant tout simplement le support des extensions matérielles de virtualisation des cœurs processeurs, ainsi que la capacité à instancier un mode **vmx non-root** (via l'instruction **VMXON**).

Lorsque le noyau de l'OS invité s'exécute dans un contexte **vmx non-root**, il s'exécute alors avec des droits amoindris, mais néanmoins suffisants pour permettre à un noyau d'un système d'exploitation non modifié de pouvoir s'exécuter sans générer un grand nombre de *traps* liés à son exécution nominale (configuration de la MMU, etc.) - voir figure 1. Ce contexte limite cependant ses droits au contrôle du contexte de la machine virtuelle dans laquelle il s'exécute exclusivement, sans pour autant impacter l'état de la machine physique. Ce dernier point reste la pierre angulaire de toute solution de virtualisation : apporter un mécanisme (logiciel, matériel) permettant de cloisonner les états de la machine physique et de chacune des machines virtuelles de manière performante et sûre.

intel_rings
Image not found or type unknown

Fig. 1 : Nouvelle découpe des niveaux d'exécution sur les architectures Intel x86 avec l'apparition de VT-x. Le mode VMX root permet d'héberger le VMM, le mode VMX non-root permet d'héberger le noyau du système d'exploitation de la machine virtuelle sans nécessiter de modification ni provoquer un trop grand nombre de traps liés à son exécution.

Le support des contextes d'exécution des machines virtuelles (sous la forme de blocs de mémoire en charge de conserver les contextes des machines virtuelles et quelques informations associées) est fait via des structures nommées VMCS : *Virtual Machine Control Structure*.

Une VMCS est associée à une machine virtuelle. Elle héberge principalement un duplicata d'un sous-ensemble des registres processeur, principalement les registres d'état, permettant de garder en mémoire un certain nombre d'informations sur son contexte lorsque la machine virtuelle est préemptée, pour pouvoir être ensuite rechargée. On y trouve donc typiquement les registres de contrôle, de debug, les registres de pointage des tables de descripteurs GDTR et IDTR, les registres MSR, et ainsi de suite. La VMCS héberge également le contexte du VMM, ceci afin de sauvegarder/recharger rapidement le contexte hyperviseur lorsque la machine virtuelle est préemptée (par exemple, lorsque le logiciel s'exécutant dans le contexte virtuel exécute une instruction nécessitant une intervention de l'hyperviseur).

L'automate à état (simplifié) lié à l'utilisation de VT-x est présenté en figure 2.

vtx_automaton
Image not found or type unknown

Fig. 2 : Automate à état simplifié des contextes processeurs sur architecture Intel x86 avec le support des extensions VMX (VT-d).

En 2008, Intel ajoute le support de l'aide à la virtualisation de la gestion mémoire et de la MMU : les *Extended Page Tables* (EPT). Ce support est nécessaire pour pouvoir exécuter un processeur

virtuel (i.e. un contexte de machine virtuelle, suite à un **VM_ENTER**, confère plus loin) en mode réel (en mode sans MMU, mode dans lequel une machine virtuelle va s'initialiser puis configurer la MMU pour entrer en mode protégé (avec support des pages mémoires et de la segmentation). Le support complet du mode réel dans un contexte processeur virtuel a été intégré en 2010.

Les EPT permettent ainsi à la MMU de différencier la vision des adresses virtuelles et physiques de l'hôte de celles vues par l'OS invité. L'OS invité peut alors configurer la MMU sans pour autant mettre en danger la configuration de la mémoire virtuelle associée à l'espace d'adressage de la machine virtuelle elle-même. Globalement, on peut voir l'EPT comme « Inception » : la gestion de tables de pages (celles des processus de l'OS invité) à l'intérieur d'une table de pages (celle de l'espace d'adressage de la machine virtuelle) **[4]**.

2.2.2 L'API VT de cloisonnement des périphériques (I/OMMU)

Rappel sur l'architecture Intel x86

Historiquement, les architectures Intel possèdent deux *bridges*, en séquence, permettant de relier le processeur aux périphériques, appelés « North-bridge » et « South-bridge ».

Le North-bridge est le plus proche du processeur, il interconnecte les périphériques impliquant un grand nombre d'interactions avec les cœurs (typiquement les contrôleurs mémoire ou le contrôleur graphique via un bus PCI-express dédié).

Le South-Bridge, plus éloigné, interconnecte les autres périphériques (contrôleurs PCI-express complémentaires, contrôleurs USB, etc.).

L'aide à la virtualisation des périphériques physiques, nommée VT-d dans les processeurs Intel le supportant, a pour but d'apporter une fonction de filtrage et de cloisonnement mémoire des périphériques. L'I/OMMU a été positionné au niveau du North-bridge, en aval des contrôleurs mémoire, mais en amont des périphériques à accès rapide (reliés au North-Bridge). Cela permet de filtrer l'accès de tout périphérique qu'il soit relié au North-bridge ou au South-bridge à la mémoire, en implémentant une méthode d'abstraction de la mémoire semblable à de la pagination mémoire, mais dont les destinataires sont les périphériques.

L'I/OMMU est sous le contrôle du processeur en mode **vmx-root**. Cela permet à l'hyperviseur, lorsqu'il alloue un périphérique physique à une machine virtuelle (via un *remapping* des accès au périphérique à cette dernière), de s'assurer que le périphérique ne peut accéder qu'à l'espace mémoire correspondant à la machine virtuelle.

En effet, si l'OS invité envoie un ordre de recopie mémoire au périphérique, il peut lui demander de recopier une zone mémoire entre deux emplacements de la RAM, (typiquement si le périphérique est ou possède un contrôleur DMA). Le périphérique étant alors autonome pour la recopie, la MMU et l'hyperviseur ne peuvent à eux seuls limiter ses accès. Le périphérique est alors capable d'aller lire (voire écrire) dans l'espace mémoire de l'hyperviseur, de l'hôte, ou d'une autre machine

virtuelle sans aucun filtrage. L'I/OMMU apporte ce filtrage en permettant à l'hyperviseur de configurer la fenêtre mémoire à laquelle le périphérique a accès.

Cette configuration est assez semblable à la configuration de la MMU pour le processeur : via une abstraction de la mémoire physique via la définition de pages mémoire virtuelles, on ne montre aux périphériques qu'un sous-ensemble de la mémoire ne correspondant qu'aux espaces mémoire physiques auxquels il a l'autorisation d'accéder.

Donner un accès à un périphérique physique à une machine virtuelle est quelque chose d'assez fréquent, classiquement pour des raisons de performances. Cela implique cependant, sans aide de leur part, de les dédier à une machine virtuelle donnée. Cependant, ces dernières années, certains périphériques commencent à supporter nativement la virtualisation, comme c'est le cas des contrôleurs Ethernet compatibles SR-IOV dont nous parlons plus loin dans le cas de KVM.

2.2.3 Petit détour par la « Nested-vitualization »

Le principe de la « Nested Virtualization » est d'exécuter un hyperviseur (*Virtual Machine Monitor*)... par-dessus un autre hyperviseur. On parle alors de VMM L1 (celui du dessus) et de VMM L0 (celui du dessous, qui s'exécute en mode hyperviseur). Ce besoin est apparu plus récemment suite à des besoins plus particuliers d'applications nécessitant une exécution dans des contextes virtualisés, eux-mêmes exécutés en concurrence avec d'autres contextes sur une même plateforme matérielle.

Le problème principal de cette architecture est l'impact sur les performances liées aux *traps* que génère le VMM L1 lorsqu'il traite les contextes des machines virtuelles qu'il gère (**vmenter**, **vmexit**, etc.), impliquant une exécution du VMM L0 qui prendra alors en charge l'exécution effective des traitements hyperviseurs, avant de rendre la main au VMM L1. Dans ses processeurs récents (Haswell), Intel a enrichi son implémentation matérielle de support à la virtualisation pour permettre la gestion des *Shadow VMCS*. Le principe est de gérer des VMCS pour les machines virtuelles de niveau L2 (gérée par le VMM L1) sans impliquer un *trap* vers le VMM de niveau L0. Le gain est de l'ordre de 40 %. L'implémentation dans KVM du support des *Shadows VMCS* a été introduite dans les noyaux 3.10 en avril 2013. Vous pouvez voir en figure 3 une comparaison des interactions entre machines virtuelles avec et sans support des *Shadows VMCS*.

nested_virt
Image not found or type unknown

Fig. 3 : Comparaison des interactions entre machines virtuelles (guest), VMM L1 et VMM L0 avec et sans support des Shadows VMCS. Seuls les ordonnancements de machines virtuelles (VM_EXIT/VM_RESUME) impliquent un passage par le VMM L0 lorsque les Shadows VMCS sont supportées par le matériel.

2.3 Et sur ARM alors ?

L'écosystème ARM

ARM est une société anglaise ayant connu une forte croissance ces dernières années avec le marché des smartphones et des objets connectés. ARM ne crée pas de processeur, mais des *designs*, que d'autres (Samsung, AllWinner, etc.) se chargent d'intégrer à des SoC (*System on Chip*), en y ajoutant divers composants matériels supplémentaires.

La famille Cortex-A de ARM est la plus puissante, apte à exécuter des OS riches type GNU/Linux. Les dernières versions du jeu d'instruction sont l'ARMv7 (32 bits) et ARMv8 (64 bits). Les cœurs ARM aptes à gérer la virtualisation sont les Cortex A7 (ARMv7a), A15 (ARMv7a), A53 (ARMv8), A57 (ARMv8) et A72 (ARMv8).

Globalement, vous avez probablement plus de composants ARM chez vous que d'x86, parfois sans le savoir (passerelles multimédias, smartphones, tablettes, téléviseurs, montres connectées, NAS, etc.).

L'arrivée du support de la virtualisation matérielle sur ARM est plus récente (ARMv7a, présenté par ARM en 2010). La virtualisation du cœur processeur, correspondant à l'apparition d'un nouveau mode d'exécution, appelé mode hyperviseur (**HYP mode** dans la littérature) a nécessité un certain travail de la part de l'équipe KVM :

- une plus forte dissociation entre le module KVM et le noyau Linux ; seul le module KVM devant s'exécuter en mode hyperviseur, le noyau Linux restant en mode superviseur (à l'inverse du mode **vmx-root** Intel) ;

- une modification de la chaîne de boot du noyau. Inutile sous x86, le noyau, sur les architectures ARM, doit démarrer en mode hyperviseur, charger le module KVM, puis passer en mode superviseur.

La première version de KVM avec le support de la virtualisation ARM (sur base ARMv7a) est sortie en 2014 **[5]**.

Sur ARM, la séparation noyau/KVM est plus flagrante **[6]**, bien que les deux continuent bien sûr d'interagir pour l'ordonnancement des machines virtuelles (le module KVM n'ordonnance pas à proprement parler) ou pour la gestion des I/O.

Les spécificités de l'implémentation de la virtualisation du cœur processeur sur ARM ont nécessité de *patcher* également le bootloader (classiquement, U-boot) pour que ce dernier démarre l'OS en mode hyperviseur et non plus en mode superviseur. En effet, il n'y a aucun retour arrière possible. Le patch d'U-boot date initialement de 2013, pour le support des extensions de virtualisation ARMv7a (Cortex A15, puis A7, comme la CubieBoard 2). Le patch a été complété pour l'ARMv8 par la suite.

Avec l'arrivée des Cortex A53/A57 et A72, ARM a déployé un mécanisme appelé SMMU (*System-MMU*) dont le but est d'apporter un cloisonnement des périphériques comme le fait l'I/OMMU sous Intel. La différence majeure est dans sa philosophie :

- Intel positionne l'I/OMMU en coupure entre la mémoire et les périphériques ;

- ARM (tout comme certains PowerPC) positionne la SMMU de manière distribuée : chaque contrôleur pouvant être initiateur sur l'*Interconnect* (apte à communiquer directement avec d'autres périphériques sans passage par la MMU et sans ordre initié par le processeur) est cloisonné derrière une instance de SMMU, qui sera en charge de faire de la translation d'adresse et de la gestion de droits. Sous certains PowerPC (typiquement chez NXP), le PAMU (*Peripherical Access Management Unit*) a le même but.

Nous parlons ici des extensions de virtualisations ARM. Il ne s'agit pas de la capacité *TrustZone*, beaucoup plus ancienne et dont le but n'est pas le même.

2.4 Les autres architectures

KVM supporte également l'architecture PowerPC, du moins pour les SoC qui supportent les extensions de virtualisation matérielle (par exemple, certains SoC de NXP). La communauté open source y est cependant moins active, car ce marché est plus confidentiel, limité principalement à certains secteurs de l'industrie comme les télécoms.

2.5 Ok, mais plus concrètement... comment KVM gère des machines virtuelles ?

Nous avons vu la manière dont KVM interagit avec le matériel pour construire les éléments structurels en charge de différencier les contextes virtuels des contextes de l'hôte. Néanmoins, il y a encore du travail logiciel à réaliser.

2.5.1 La création d'une machine virtuelle

Une machine virtuelle est vue comme un processus de l'hôte... ou presque. Elle possède un espace d'adressage (au sens de la MMU), mais avec une structure particulière. Là où un processus de l'hôte est classiquement découpé en deux types de *mapping* mémoire (*user* et *kernel*, pour la gestion des *traps* et des appels systèmes), l'espace d'adressage d'une VM possède un *mapping* mémoire taggué... « guest ». L'OS hôte n'est alors pas en charge d'en gérer le contenu **[7]**. Lorsque nous parlons de l'espace d'adressage de la machine virtuelle, il faut bien voir qu'il ne s'agit pas de l'espace d'adressage du processus Qemu qui lui est apparenté, mais bien de la machine virtuelle elle-même. Il s'agit bien de deux espaces d'adressages différents, même si un certain *remapping* mémoire est fait entre les deux. En effet, le processus Qemu est lui un processus normal de l'OS hôte, et doit donc avoir un *mapping user* et *kernel*, comme tous les autres.

La création effective de la machine virtuelle se fait au travers du fichier spécial **/dev/kvm**, au travers duquel des ordres liés à la création/suppression de machines virtuelles peuvent être envoyés. Ces ordres sont envoyés classiquement par qemu (et non directement par la libvirt),

lorsque ce dernier est configuré pour travailler avec KVM. La séquence est la suivante :

- Qemu fait alors une demande explicite de création de machine virtuelle au module KVM ;

- ce dernier initialise les éléments matériels (structure VMCS, EPT si présent, etc.) ;

- le module KVM crée également les contextes logiciels (structures de données du noyau, etc.) et interagit avec les éléments du *kernel* comme l'ordonnanceur ou le gestionnaire mémoire pour créer l'espace d'adressage de la machine virtuelle et l'instancier ;

- la VM démarre suite à un appel à l'instruction **VMENTER**. La séquence de *boot* est gérée par Qemu, qui fournit par ailleurs une image BIOS (cas x86 uniquement). Globalement, Qemu sera en charge de traiter toute I/O ou signal lié au matériel émulé. Cependant, ces traitements ne se font pas par un passage direct entre la machine virtuelle et Qemu : toute sortie de la machine virtuelle (**VMEXIT**) implique un passage par le VMM (le module KVM), qui est alors en charge de déterminer si la sortie est liée à une action à effectuer de la part de Qemu ou non (il peut s'agir d'un *trap* impliquant une action du module KVM) - voir figure 4.

Fig. 4 : Architecture logicielle générale d'un hôte KVM hébergeant une machine virtuelle et interactions.

2.5.2 La destruction d'une machine virtuelle

La destruction est assez logique, car elle correspond à la suppression du processus Qemu en charge de gérer les I/O émulées de la machine virtuelle ainsi que du contexte logiciel et matériel associé à la machine virtuelle dans le noyau Linux et dans le matériel.

Cependant, on utilise classiquement les capacités de Qemu à envoyer un message ACPI (ou équivalent) à l'invité avant sa destruction afin de lui permettre de « s'éteindre » proprement. C'est la différence entre le « shutdown » et le « destroy » dans la configuration XML de la libvirt. Cet ordre n'est pas géré par KVM, mais par la libvirt par interaction avec Qemu. KVM se charge lui de supprimer la machine virtuelle, pas de gérer le logiciel qui s'y trouve cloisonné.

2.5.3 L'ordonnancement d'une machine virtuelle

Le module KVM utilise les capacités du noyau Linux pour ordonnancer les machines virtuelles. Cela lui permet de gérer l'affinité CPU, les cgroups et ainsi de suite. Il est alors possible d'exploiter des capacités évoluées d'ordonnancement, paramétrable par exemple dans la configuration d'un domaine libvirt, pour spécifier le quantum CPU autorisé pour chaque VM, l'affinité ou divers autres éléments d'ordonnancement et d'optimisation.

2.5.4 Mais alors... Les processus root de l'hôte voient et ont accès aux machines virtuelles ?

En réalité la réponse est non. Ils voient et ont potentiellement accès au processus Qemu, mais pas à l'espace d'adressage de la machine virtuelle (celui taggué « guest »). Celui-là, seul le noyau Linux est capable de le voir.

3. Le modèle Virtio

L'API Virtio est une implémentation dont le but est de fournir une abstraction de plusieurs familles de périphériques au travers d'une API unifiée, afin d'optimiser les performances d'accès par rapport à l'émulation d'un périphérique. Il ne s'agit plus ici d'émuler un composant, mais réellement d'employer volontairement une API logicielle pour instancier des périphériques divers comme de type bloc ou de type réseau (**virtio-block** et **virtio-net**). L'usage de l'API Virtio implique bien sûr de modifier le noyau de l'OS invité. Il doit en effet volontairement utiliser un périphérique de type Virtio. Il s'agit donc ici de paravirtualisation des I/O. Attention cependant : KVM, bien qu'apte à supporter la paravirtualisation des I/O, ne sait pas paravirtualiser le cœur processeur comme le permet par exemple Xen : la présence des extensions matérielles d'aide à la virtualisation du cœur processeur sont nécessaires.

L'usage de composant virtio permet de créer des périphériques virtuels en considérant que tout mécanisme de communication avec un périphérique d'I/O revient au final à un échange de *buffers*. On retrouve donc un ensemble limité de fonctions permettant de gérer la transmission et la réception de *buffers*, ainsi que des fonctions de gestion d'événements. Son implémentation est donc basée sur une abstraction :

- les drivers virtio spécialisés (driver Ethernet, de disque, etc.) ;

- un backend « générique » de traitement des *buffers*.

Côté Qemu, les mécanismes de traitements des buffers doivent être également traités, ce qui implique donc également la présence de code spécifique pour récupérer dans l'*iothread* (voir figure 4) les *buffers* transmis par la machine virtuelle ou devant lui être remis. À cela, il faut également déterminer vers quelle destination doivent être émises ces données. Il s'agit là des *backend-drivers* . Il en existe un grand nombre :

- à destination d'un fichier pour un périphérique en mode bloc géré sous forme d'un fichier ;

- à destination de la pile réseau du noyau Linux (cas d'un périphérique virtio-net interconnecté à un *bridge* par exemple) ;

- à destination d'un autre processus *userspace* (typiquement le cas de DPDK **[8]**) ;

- ...

Il faut bien voir que ces mécanismes impliquent l'exécution du processus qemu et de son ou ses *iothreads*, et donc classiquement une préemption de la machine virtuelle. À l'inverse, le *remapping*

direct de périphériques physiques ou de Virtual Functions SR-IOV évite ces préemptions.

4. KVM et les périphériques SR-IOV

SR-IOV (*Single-Root I/O Virtualization*) définit un ensemble de méthodes pour les périphériques PCIe afin de les rendre compatibles avec les architectures virtualisées. Il s'agit d'une implémentation matérielle et donc d'un support que doit posséder le périphérique. Le périphérique est alors apte à instancier des contextes « virtuels » de lui-même (appelés VF, *Virtual Functions*), en plus d'un contexte physique (PF, *Physical Function*) sous contrôle de l'hôte. Un périphérique peut alors être utilisé par plusieurs machines virtuelles en appliquant un *remapping* des VF dans les machines virtuelles tout en assurant un cloisonnement entre ses dernières.

Bien que la spécification SR-IOV décrit qu'un périphérique est apte à instancier jusqu'à 128 VF, dans les faits on retrouve plutôt de l'ordre de 8 VF par PF. Par exemple, les contrôleurs Ethernet Intel modernes comme les cartes i240 (driver **igb**) supportent cette fonction.

La configuration des VF doit être faite par l'hôte et n'est pas du ressort du module KVM. Une fois configurée, chaque VF est vue comme un net *device* (au sens Linux), ce qui permet par la suite de l'assigner de manière tout à fait naturelle aux machines virtuelles. Chaque VF possède son propre contexte et ses registres de configuration, paramétrables par la machine virtuelle, avec des limitations pour ne pas impacter l'état du contrôleur physique (typiquement une modification des paramètres d'auto-négociation pouvant entrer en collision avec une autre VF, celle-ci étant reliée au même contrôleur Phy et au même câble Ethernet).

5. KVM et la sécurité

La présence de plusieurs machines virtuelles sur une même machine physique implique un grand nombre d'impacts en terme de sécurité dont on ne parlera pas ici. Cependant, il faut bien voir qu'il est important :

1. de cloisonner les machines virtuelles entre elles ;

2. de cloisonner l'ensemble des fonctions de virtualisation (libvirt, qemu, etc., selon la richesse des fonctions présentes) du reste des fonctions logicielles de l'hôte ;

3. de protéger le noyau Linux, qui héberge entre autres le module KVM.

Les machines virtuelles, pour interagir avec l'environnement logiciel qui les porte, s'appuient sur le processus **qemu**. Ce dernier doit donc impérativement être restreint pour limiter le risque de VM-Escape (capacité, pour une fonction logicielle virtualisée, de sortir de son contexte virtualisé). La libvirt apporte ainsi des éléments d'aide à la sécurisation en simplifiant le cloisonnement des machines virtuelles via un driver SELinux et via les cgroups. Le but est de limiter les accès du

processus Qemu, en charge de gérer les I/O de la machine virtuelle.

Lorsque des périphériques physiques sont *remappés* dans une machine virtuelle, la présence d'une I/OMMU (ou d'une SMMU pour ARM, PAMU pour PowerPC) est impérative. Cette fonction doit être configurée de manière rigoureuse.

Classiquement, l'OS hôte est dédié à la virtualisation. Il peut néanmoins héberger des services divers avec des droits étendus, et il est important de s'assurer que les fonctions n'ayant pas nécessité d'accéder aux fonctions de gestion de la virtualisation restent contraintes. Les LSM (**AppArmor**, **SELinux**, etc.) permettent, comme pour tout système GNU/Linux de mettre en place de telles entraves. La présence d'un OS hôte en plus d'une fonction d'hyperviseur accroît le risque sécuritaire. Des entités comme l'ANSSI ont défini des documents de bonnes pratiques pour les serveurs et les systèmes de virtualisation, qu'il ne faut pas hésiter à étudier **[9][10]**.

Conclusion

Nous avons pu voir dans cet article que l'hyperviseur KVM est basé sur une architecture faisant intervenir de multiples entités logicielles comme matérielles, ce qui rend sa compréhension et sa maîtrise difficile. De plus, l'hétérogénéité et la modularité des architectures matérielles rendent l'implémentation d'un hyperviseur performant et portable malaisé malgré l'aide que le matériel peut apporter.

Cet article a mis l'accent sur les différentes API et différents chemins de données cachés aux utilisateurs de la technologie KVM, afin de mieux appréhender la complexité sous-jacente. Il ne faut cependant pas perdre de vue que les technologies logicielles décrites dans cet article ne sont qu'un composant de l'écosystème logiciel qui fait aujourd'hui la force de KVM et qui intègre des fonctions de plus haut niveau comme **OpenStack**, permettant d'aboutir à des infrastructures de virtualisation de grande envergure et répondant à des contraintes métier que l'hyperviseur seul ne permet pas de résoudre.

Références

[1] KVM Myths - Uncovering the Truth about the Open Source Hypervisor :
https://www.ibm.com/developerworks/community/blogs/ibmvirtualization/entry/kvm_myths_uncovering_the_truth_about_the_open_source_hypervisor?lang=en

[2] Red Hat, *« Top-secret KVM, lessons learned from an ICD 503 deployment »* :
http://people.red-hat.com/jamisonm/usaf/KVM_security.pdf

[3] VMWare, *« Understanding Fullvirtualization, paravirtualization and hardware assist »* :
https://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[4] *« EPT in KVM »* : **https://zhongshugu.wordpress.com/2010/06/17/ept-in-kvm/**

[5] *kvm : the Linux Virtual Machine Monitor*

[6] *KVM/ARM : The Design and Implementation of the Linux ARM Hypervisor*

[7] Intel, *« Intel® 64 and IA-32 Architectures Software Developer's Manual »*, Volume 3B - *« System Programming Guide, Part 2 »*.

[8] DPDK, kit de développement pour application de traitement réseau haute performance : **http://www.dpdk.org/**

[9] *« Bonnes pratiques de configuration d'un poste GNU/Linux »* : **http://www.ssi.gouv.fr/entreprise/guide/recommandations-de-securite-relatives-a-un-systeme-gnulinux/**

[10] *« Bonnes pratiques de sécurisation d'un système de virtualisation »* : **http://www.ssi.gouv.fr/entreprise/guide/problematiques-de-securite-associees-a-la-virtualisation-des-systemes-dinformation/**

Stream audio

Install scream sur la vm windows:

https://github.com/duncanthrax/scream/releases

Scream sur le réseau

Install scream sur le Linux

Et lancement de scream sur l'interface réseau:

scream -i virbr0

Pour limiter l'utilisation du réseau et éviter les saccade, dans la machine windows 16bit 44100hz qualité cd:

https://github.com/duncanthrax/scream/raw/master/doc/sampling-rate.png

---

Scream sur un device mem

Possibilité de crée une interface audio vituelle dans KVM pour ne pas utiliser le réseau:

```
<device>
...
 <shmem name='scream-ivshmem'>
   <model type='ivshmem-plain'/>
   <size unit='M'>2</size>
   <address type='pci' domain='0x0000' bus='0x00' slot='0x11' function='0x0'/>
 </shmem>
 ...
</device>
```

Clé registre à ajouter dans un shell:

REG ADD HKLM\SYSTEM\CurrentControlSet\Services\Scream\Options /v UseIVSHMEM /t REG_DWORD /d 2

Drivers windows ivshmen, c'est le drivers pci ram device qu'il faut mettre à jour:

https://fedorapeople.org/groups/virt/virtio-win/direct-downloads/upstream-virtio/

Et d'aller lire le contenu:

scream -m /dev/shm/scream-ivshmem

Virsh - Commande

Utilisation de la commande Virsh

Nous allons voir ici l'Utilisation de la commande **Virsh** suite à l'installation de notre **Xen** ou encore **KVM**. Celle-ci est intégrée à la librairie **libvirt** et permet donc la gestion des machines virtuelles.

Virsh

Bien sûr, il est possible d'utiliser l'outil graphique **Virt-manager** mais l'outil en ligne de commande nous permettra surtout l'écriture de script par la suite.

Pour commencer, dans son mode le plus simple, **Virsh** s'exécute en mode interactif :

```
[root@localhost ~]# virsh
Bienvenue dans virsh, le terminal de virtualisation interactif.

Taper :  « help » pour l'aide ou « help » avec la commande
        « quit » pour quitter

virsh #
```

Ensuite, pour lister toutes les machines actives, il faudra effectuer la commande suivante :

```
# virsh list
```

Mais aussi, vous pouvez ajouter l'option **–inactive** pour les machines inactives et **–all** pour lister tout sans faire de différence :

```
# virsh list --inactive
# virsh list --all
```

Puis, pour démarrer, arrêter ou encore redémarrer une machine, nous pourrons utiliser les commandes suivantes :

```
# virsh start VmName
# virsh shutdown VmName
# virsh reboot VmName
```

D’autre part, pour forcer l’arrêt d’une VM :

```
# virsh destroy VmName
```

Et si vous voulez mettre en pause une VM :

```
# virsh suspend VmName
# virsh resume VmName (pour la reprise d'une VM précédemment mise en pause)
```

Aussi, comme vu dans les articles précédents sur Xen et KVM, vous pouvez vous connecter via :

```
# virsh console
```

Mais encore, pour afficher les informations d’une machine virtuelle :

```
# virsh dominfo VmName
ID :           2
Nom :          VmName
UUID :         df7ad48b-f752-4f5d-be4b-84407c6fe5f9
Type de SE :   hvm
État :         en cours d'exécution
CPU :          1
Temps CPU :    19,2s
Mémoire Max : 1048576 KiB
Mémoire utilisée : 1048576 KiB
Persistent:     yes
Démarrage automatique : disable
Managed save:   no
Security model: selinux
Security DOI:   0
Security label: system_u:system_r:svirt_t:s0:c758,c991 (enforcing)
```

Vous pouvez aussi afficher le fichier de configuration **xml** de la machine virtuelle :

```
# virsh dumpxml VmName
```

Puis modifier cette configuration ici :

```
# virsh edit VmName
```

De plus, vous pouvez voir la liste des pools et les noms des volumes utilisés par celui-ci :

```
# virsh pool-list
Nom                  État      Démarrage automatique
-------------------------------------------
 images               actif      yes
# virsh vol-list images (images est ici le nom du pool pour la commande  virsh vol-list NomDuPool)
 Nom                  Chemin
-------------------------------------------------------------------------------
 VmDebianTest.img     /var/lib/libvirt/images/VmDebianTest.img
 VmDebianTestKVM.img  /var/lib/libvirt/images/VmDebianTestKVM.img
 VmFedo29.img         /var/lib/libvirt/images/VmFedo29.img
 VmFedora31.img       /var/lib/libvirt/images/VmFedora31.img
```

Petit aparté, pour manipuler les images disques, il faudra vous aider de **qemu-img** :

```
Informations sur une image:
qemu-img info fichier.img

Étendre la taille de l'image:
qemu-img resize fichier.img +tailleG
```

Ensuite, du côté réseau, il est possible de lister la liste des réseaux virtuels :

```
# virsh net-list --all (ici nous listons les réseaux actifs et inactifs)
```

Et il est possible d'avoir un peu plus d'informations sur un réseau en particulier via la commande suivante :

```
# virsh net-info NomDuReseau
```

Par ailleurs, vous pouvez aussi activer ou désactiver un réseau virtuel avec les commandes suivantes :

```
# virsh net-start NomDuReseau
# virsh net-destroy NomDuReseau
```

Virsh permet également de créer et gérer des **snapshot**. Pour la création et suppression d'un **snapshot**, il faudra utiliser les commandes suivantes :

```
# virsh snapshot-create VmName
# virsh snapshot-delete VmName
```

Pour lister les différents **snapshot** d'une VM :

```
# virsh snapshot-list VmName
```

De plus, vous pouvez revenir au **snapshot** créé précédemment via cette commande :

```
# virsh snapshot-revert VmName
```

Pour finir, celle-ci vous permettra d’avoir des informations à propos de la **node** :

```
# virsh nodeinfo
modèle de CPU :    x86_64
CPU :              2
Fréquence de la CPU : 4008 MHz
socket(s) CPU :    1
Coeur(s) par emplacements : 2
Thread(s) par coeur : 1
cellule(s) NUMA :  1
Taille mémoire :   3848800 KiB
```

virt-install

Install debian view serial console:

```
virt-install --name debianprod --ram 4000 --disk /mnt/data/vm_prod/debian_garfi-fr.qcow2 --vcpus 4 --network network=default --location /mnt/data/vm_prod/debian-testing-amd64-netinst.iso --extra-args="auto console=ttyS0,115200n8 serial" --graphics none --console pty,target_type=serial
```

VM - passthrough pci gpu

Préparation:

- activer IOMMU bios
- installer qemu/virt-manager/libvirt/ovmf
- paramètre kernel: intel_iommu=on ou amd_iommu=on

Configuration modules/vfio:

- script bash pour voir le groupe iommu de la carte:
  - **#!/bin/bash**
    shopt -s nullglob
    for d in /sys/kernel/iommu_groups/*/devices/*; do
    n=${d#*/iommu_groups/*}; n=${n%%/*}
    printf 'IOMMU Group %s ' "$n"
    lspci -nns "${d##*/}"
    done;

- Configuration vfio:

  - **cat /etc/modprobe.d/vfio.conf**

    options vfio-pci ids=10de:1c81,10de:0fb9 disable_vga=1
  - Ou bien niveau noyau: vfio-pci.ids=1002:67e3,1002:aae0

- Chargement des module:
  - **cat /etc/modules-load.d/modules.conf**
    # List of modules to load at boot
    vfio-pci ids=1002:67e3,1002:67e3
    vfio
    vfio_iommu_type1
    vfio_pci

vhost-net

Configuration kvm/virtmanger

- configurer bios uefi/q35
- ajouter périphérie pci (GPU/HDMI SOUND)
- Affichage VNC, (pas spice)
- Vidéo VGA (pas qxl)

Gestion réseau

Forward port:

```
virt-xml $DOMAIN --edit --confirm --qemu-commandline '-net'
virt-xml $DOMAIN --edit --confirm --qemu-commandline 'user,hostfwd=::$PORT-:$PORT'
```

XML:

```
<qemu:commandline>
<qemu:arg value='-net'/>
<qemu:arg value='user,hostfwd=::$PORT-:$PORT'/>
</qemu:commandline>
```

Acces relay par iptables

```
iptables -I FORWARD -m conntrack -d $SOUS_RESEAU_VM/24 --ctstate NEW,RELATED,ESTABLISHED -j ACCEPT
```

```
iptables -t nat -A PREROUTING -p tcp -d $IP_PUBLIC --dport 32400 -j DNAT --to-destination $IP_VM:$PORT_VM
```

Kvm FS

Réduire taille vm à la réel:

```
apt-get install libguestfs-tools


virt-sparsify --in-place file.qcow2
```

Augmenter taille

```
qemu-img resize file.qcow2 +100G (shutdown previous)
```