https://www.coincoin.fr.eu.org/?KVM-evaluation

KVM evaluation

- 1- Blog-Notes - Au boulot -

Date de mise en ligne : lundi 12 septembre 2011

Virtualisation Matérielle

Tout d'abord KVM n'est utilisable que sur des processeurs implémentant un mode de virtualisation matérielle : - VT (Virtualization Technology) chez Intel - SVM (Secure Virtual Machine) chez AMD

Le principe de ce mode de fonctionnement est le suivant :

Par défaut, un processeur dispose de 4 niveaux d'exécution appelés « rings ».

Sur une archi x86 32 bits, le système d'exploitation fonctionne sur le ring 0 et dispose du plus haut niveau de contrôle à l'inverse des applications qui tournent sur le ring 3.

Les couches de niveau supérieures ne peuvent pas modifier le comportement des couches inférieures, seul l'inverse est vrai. (Un OS arrête une appli, pas l'inverse)

Les rings 1 et 2 sont des niveaux intermédiaires, peu utilisé et donc supprimé sur les archis x86-64.

Le passage en mode privilégié est standard sur des archis x86, ce sont nos chers syscalls.

Exemple :

On fait un « open » en userland, une interruption (int 0x80 sous archi x86 pour ne pas la citer) est envoyée au kernel qui passe dans la fonction « trampoline » codée en assembleur (fichier syscalls.S si je ne m'abuse) et qui permet de switcher le processeur en mode privilégié, le fameux ring 0.

A partir de là on est en kernel mode, le processeur est au ring 0 et tout est possible.

Revenons à nos moutons :

Lorsque l'on virtualise, on utilise un OS privilégié : l'hyperviseur. Il contrôle les hôtes et les ressources physiques et utilise le ring 0.

L'hyperviseur gère les guests qui doivent donc, pour respecter la hiérarchie de contrôle, utiliser les rings 1 ou 2.

Le problème est qu'un OS est conçu pour ne s'exécuter que sur le ring 0, il vérifie qu'il réside au bon endroit mémoire dédié aux OS et certaines instructions ne s'exécutent que si elles viennent du ring 0 ou y vont. (et heureusement…)

Pour pallier à ce problème, on utilise donc la virtualisation.

La paravirtualisation (XEN) consiste à modifier les OS invités pour permettre un mode d'exécution autre que le ring

0.

Lorsque l'on ne peut pas modifier le code de l'OS (…), on procède à des translations binaires (VMware, QEMU) pour faire croire à l'OS qu'il occupe un emplacement mémoire valide ainsi que le niveau d'exécution processeur ring 0. Evidemment, ce dernier mode est plus gourmand en ressource.

Les technologies de virtualisations matérielles consistent à créer un nouveau mode d'exécution (VMX chez Intel). On a donc un niveau racine < ring 0, et un niveau normal correspondant aux rings 0-3 décrits précédemment. L'hyperviseur s'exécute en mode racine et les hôtes en ring 0. Pour ceux qui s'imaginent déjà que nous nous sommes complètement affranchis de la translation binaire, je suis désolé mais ce comportement est encore nécessaire pour certaines fonctions.

Lorsque l'on se penche sur les sources de KVM, les développeurs mettent en avant un code concis, clair et permettant de profiter des améliorations du kernel 2.6 sans effets de bords possible. C'est précisément parce que les technologies VT et SVM existent que cela est possible.

Maintenant que les guests OS disposent d'un espace d'exécution dédié ainsi que d'un niveau d'exécution qui ne les perturbe pas trop, il va falloir partager un peu les ressources, l'hyperviseur est généralement dédié à cela.

Naturellement un guest OS se croit seul au monde, et l'hyperviseur se doit de faire du context switching d'OS pour fournir des « time slice » en fonction des besoins de chacun. Pour rendre possible tout cela, il faut stocker une image du système, sauvegarder les registres, les caches etc… et remettre tout ce petit monde en activité régulièrement.

Les technologies VT et SVM disposent d'un jeu d'instructions dédiées à ces taches (VM Exit, VM Launch, VM Resume…), d'un jeu de signaux spécialisés dans l'évènementiel (interruptions hardware, Idle Time) et chaque hôte se voit allouer un segment mémoire de quelques Ko de son espace d'adressage pour stocker les données d'état. Cet espace est évidemment lu avant chaque mise en activité et écrit à chaque context switch d'OS.

Exemple :

Le processeur utilisé alors par un guest OS vaque à ses occupations, reçoit un signal connu et référencé dans son handler (table de 32 bits), il lance VM Exit et passe la main à l'hyperviseur. Ce dernier lance son processus de sauvegarde d'état puis effectue un éventuel context switch par le biais de VM Resume puis VM Launch.

L'hyperviseur dispose ainsi d'une granularité de contrôle importante et c'est très peu consommateur en ressource.

Fonctionnement de KVM

Nous disions donc que KVM nécessite le support des fonctionnalités processeurs de virtualisation matérielle.

Le fonctionnement de KVM est basé sur un hyperviseur. Par abus de langage, le terme d'hyperviseur est assimilé à l'OS privilégié qui gère les guests OS, c'est inexact. Pour être précis, l'OS privilégié exécute des tâches critiques sur demande de l'hyperviseur qui est une entité fonctionnelle dans le kernel. Cette entité étant basée sur diverses technologies matérielles et algorithmes.

Dans le modèle KVM, l'hyperviseur est donc basé sur : - Les technologies de virtualisations matérielles - Le scheduler kernel - La gestion de la mémoire du kernel - Un développement propre au projet permettant, entre autres, d'organiser et ordonnancer tout cela.

L'hyperviseur gère le scheduling des taches au sens large du terme, la mémoire et délègue les IO à un hôte privilégié.

Par défaut, un OS Linux dispose de 2 modes de fonctionnement : User Mode et Kernel Mode.

KVM ajoute un troisième mode appelé Guest Mode qui dispose de ses propres User Mode et Kernel Mode.

Les taches sont décomposées de la façon suivante : - Les OS invités sont exécutés en Guest Mode, on y traite toutes les taches hormis les IO. - En Kernel Mode on ordonnance les OS invités et les requêtes d'IO de ces derniers. - En User Mode on exécute les requêtes d'IO des OS invités. (c'est là ou cela devient amusant…)

On remarquera qu'à aucun moment, dans le modèle décrit, il n'est nécessaire de créer un scheduler de taches, un système de gestion de mémoire, un patch pour des drivers hardware. C'est précisément ici que se trouve la valeur ajoutée de KVM. Il est natif au code du kernel Linux, ne nécessite pas de maintenance pour les drivers hardware et bénéficie de toutes les améliorations kernel. (Pour rappel, à partir des kernels 2.6, le scheduler de tache est en O(1) et sera encore amélioré, il serait dommage de ne pas en profiter….)

Utilisation de KVM et résultats de tests

Pour revenir à des choses un peu basique, KVM est implémenté à partir des versions de kernel 2.6.20 et se décompose en : - Un module kernel « kvm », implémentant l'API globale de scheduling, de gestion de mémoire, le Guest Mode et l' « intelligence » de l'outil - Un module « kvm-intel » ou « kvm-amd » comprenant les traitements et instructions propres au processeur utilisé. - Un device driver /dev/kvm (le minor number est alloué dynamiquement, pas vraiment pratique pour industrialiser…) permettant de contrôler KVM et les guests OS.

Ensuite nous utilisons l'outil userland QEMU, légèrement modifié pour prendre en compte /dev/kvm ainsi que le module d'accélération matériel kqemu. QEMU s'occupe de gérer l'émulation hardware, on a parlé d'IO tout à l'heure, c'est par ici que cela se passe.

QEMU est utilisé pour gérer les OS invités à savoir : - le démarrage - l'arrêt - le monitoring - les espaces disques (le guest OS est-il sur un ensemble de partitions LVM2 ? dans un fichier monté en loopback ? sur un CDROM ? dans un fichier ISO ?) - les interfaces réseaux (natif, VLAN …) - l'accès au guest OS via VNC entre autres - l'activation / désactivation des options de boot comme l'ACPI ou l'USB - le copy-on-write des images disques - etc etc…

Son utilisation n'est pas du tout conviviale et l'industrialisation de l'outil ne révèlerait pas straightforward.

On boot sur un kernel 2.6.21, on charge les modules kvm, kvm-intel (on aura au préalable activé le VT dans le BIOS), kqemu (module d'accélération matérielle).

On crée les devices /dev/kvm et /dev/kqemu avec les minor qui vont bien.

On démarre un Guest OS par le biais de qemu :

qemu -boot c -hda /dev/mapper/vgâ€”ld0-pouet_kvm -hdb /dev/mapper/vgâ€”ld0-pouet_work -k fr -vnc 0:0 -daemonize -m 1024 -net nic -kernel-kqemu

A partir de ce moment, QEMU vérifie que nous avons monté une shm d'une taille suffisante pour le guest OS (1024 dans notre exemple), ensuite il mmap() la mémoire physique de l'hôte invité puis appelle le module kernel kvm (par le biais /dev/kvm) pour qu'il lance le Guest Mode.

Un Guest OS est naît.

Si l'on se penche maintenant sur le cheminement suivit par les processus d'IO, cela se déroule comme suit :

Un OS invité envoi une interruption d'IO. (Guest/User Mode puis Guest/Kernel Mode) L'hyperviseur (j'ai bien dit hyperviseur…) reçoit la requête et la transfert à l'OS privilégié (shm quand tu nous tiens…). En reprenant notre exemple, l'OS privilégié sera l'OS depuis lequel nous avons lancé la commande « qemu » et crée le guest OS.

Le scheduler de l'hyperviseur procède à un context switch d'OS pour donner la main à l'OS privilégié. On arrive ici en User Mode puisque, aux yeux de l'OS privilégié, c'est le processus userland QEMU qui a demandé la requête d'IO.

On appelle le gestionnaire d'interruption de l'hôte privilégié, qui envoi la requête au scheduler d'IO. (Kernel Mode) L'hôte privilégié procède à un context switch classique. (Kernel Mode) La requête d'IO est effectuée. (Kernel Mode) Ici c'est une requête d'IO classique. Pour résumer : Syscall -Â» VFS -Â» récupération de l'inode (fonction nameilookup dans namei.c si mes souvenirs sont corrects) -Â» switch sur le FS qui va bien -Â» remplissage d'une structure bioread après avoir récupérer le dinode (j'ai bien dit dinode), le driver hardware fait son boulot en flushant le buffer et on repart en sens inverse.

Un signal de fin d'IO est envoyé à l'hôte privilégié, on revient en User Mode pour repartir de suite en Kernel Mode, puis l'hyperviseur procède à un context switch d'OS pour revenir au guest OS initiateur de la demande d'IO. L'hyperviseur relance le guest OS. (Vous vous souvenez, VM Resume tout ça…)

L'avantage, lorsque l'on utilise KVM est qu'aux yeux d'un hôte privilégié, les guests OS sont des processus classiques. Les outils d'administrations standard fonctionnent donc sans problème (ps », « kill », « top »).

Les tests comparatifs ont été réalisés entre 1 OS privilégié et 1 guest OS : G5 doté d'un CPU 3.00GHz, 1 Go RAM, partitions LVM2

(ces résultats sont une moyenne de 3 benchs).

Tests IO :


```
 iozone -s 1k -r 1k -t 200 -i0 -i1 -o



                      OS privilégié / OS invité
Ecriture                          20
Ré-écriture                      100
Lecture                           12
Re Lecture                       13
```

Tests CPU :

```
nbench
```

Je ne mets pas les résultats de chaque test ici, ce n'est pas pertinent dans la mesure ou la seule chose à retenir est que nous perdons, dans le pire des cas 3% de CPU entre l'OS privilégié est le Guest OS.

Les résultats des tests CPU et IO sont rassurants puisqu'ils viennent confirmer le mode de fonctionnement décrit précédemment.

En revanche, les résultats des tests d'IO sont très en dessous de notre seuil de tolérance.

Conclusion

Le concept KVM est sympa et très prometteur.

La majeure partie des points négatifs vient de QEMU, les IO sont mauvaises, l'administration des hôtes n'est pas limpide, pas de support 64 bits pour le moment, tous les hardwares ne sont pas supportés (disk et réseau pour ne citer qu'eux). Il n'est pas simple d'implémenter une archi Front/Middle/Back avec les VLANS qui vont bien.

Dans le cadre d'une industrialisation il faudrait créer un nouveau processus de génération de guest OS, un wrapper d'administration facilement utilisable ainsi qu'un outil de monitoring propre pour la prod.

Bref, avis défavorable en ce qui me concerne mais il faut garder un œil dessus car s'ils se mettent à faire un peu d'IO et qu'ils codent un outil d'admin/monitoring convivial cela peut être vraiment sympa.