

VIRTUALISATION

Exposer à un user qqch qui ne l'est pas VM⁺
→ mémoire, machine ...
⇒ Isolat°, m° performance que sans virtu.

On peut vouloir rajouter des comportements sur la couche virtualisée
ex: Copy on write

Si on essaie juste d'exec un kernel, au debut ça passe
mais dès qu'accès à certains registres (reg de contrôle) ça casse.

Instr sensible → lis ou écrit ds reg de config

Privilégiée → superviser Non privilégiée → user & supervisor

D'habitude, si priv en user → kill pgm
En virtu, on trap et on émule

En plus, on veut RAM (on virtualise la mem, déjà fait)
et des devices (on émule : en map en PROT NONE)

Mais ça sous entend que archi virtualisable
→ les instr sensibles sont privilégiées.

Spoiler : x86 n'est pas virtualisable → IDT : sidt
sidt n'est pas priv → on récupère @ hyperviseur et pas @ IDT

57 → 6 mois "On virtu x86 en un an"

58 → Ils réussissent o.o (fonderont + tard vmware)
⇒ lit les instr en avance et les patch
en gros, on met l'OS ds un debugger
pb sur code automodifiant.

Cambridge : refléchit ≠ ^{m+}. (< 2006)

Normal m+, on ne modifie pas ce qui on virtualise.

Faut sauter ce requirement → modif OS.

Comme on envie de virtu linux, code source dispo → cool
⇒ hypercall (= sys call qui modif linux) → Xem

7th, extension pr permettre virtu du x86 (nouveau jeu d'instr)
→ accelerat° par CPU (instr priv exec par CPU ds context VM
⇒ plus le boulot de l'hyperviseur)

Il reste des trucs à faire par l'hyperviseur → VM exit
(ex: devices)

Grâce au nouveau jeu d'instr x86 → on peut faire tourner VM de
tous les OS sous les modif.

Mémoire : nesting de paginat°:

phys → process=vm virt mem → virt mem process
ds VM.

charge mode)
↳ VMCS

Device : APIC. Au début complètement émulé.

Qd real device, on renvoie interrupt en passant par hyperviseur qui le renvoie avec APIC émulé.
On peut la remonter directement (sans VMexit)

Minimiser VMexit → + rapide

(VMexit ~ context switch : lent + coûte cher)

Arrivée des VM sur PC → modifié le comportement des gens à propos des machines.

VM : + simple à admin

- si sa casse, on peut juste détruire la VM et la relancer.
- arrivée du cloud
- automatisation de l'install de machine

Amazon : business sur le rent de machine virt. + auto de déploiement

Netflix : pas de serveur 0.0 → infra sur Amazon

Phys^e datacenter : google, fb et quelques autres seulement

Thuis il reste des pb face à l'automatisation → déploiement de soft

Container : on veut augmenter la densité des services.

au lieu de lancer plein d'OS virt pour avoir pleins de services → virt à plus haut niveau : namespace!

Sur Linux : namespace, par ex pour pid : plusieurs jeux de pid.

isolat^e + forte qu'entre process.

de un namespace on ne peut pas toucher aux pid des autres.

⇒ plusieurs userland isolés qui partagent le même noyau.

⇒ même kernel de 1 seul OS.

* si faille dans kernel, accès à tous les containers.

cgroup : grp de process avec dt spé / comportement spé

ex : limite nb cœur, % CPU, % mémoire, ressources.

security module Linux : label d'objets kernel, ...

seccomp : filtrer et tracer syscalls used by process
→ limite comportement du process.

noch namespace mais partage non. ex : keyring.

Tes 4 features servent à créer des containers.

Deployment de code facile/auto → décrire ds un file isolé
qui installe/lance le service et le lancer ailleurs

⇒ Docker pb : stateless. Accès disque ??

VMware

Hyper-viseur : kvm qemu workstation ESX hyper-v virtualbox
 bhyve vmm xen

(Relance des machines sur une machine).

Qd on virtualise, on fait tourner en user un code qqconque comme si de rien n'était.

ex: un OS

Emulat^o: reinterprete bytecode pour exec mais pas sur CPU

Qd instr mémé, on emule. Sinon on exec. → Virtualisat^o
 → perf acceptable / natif, fidélité (pas de modif du pgm)
 isolé ↳ même comportement

VMM : virtual machine monitor

Reg généraux et de contrôle
 Modes user et supervisor

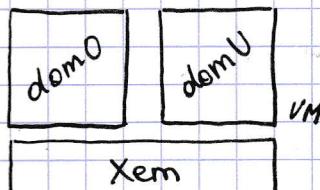
Pour être virtualisable, il faut que
 tous les instr sensibles soient
 privilégiées.

→ Trap & emulate

x86 → pas virtualisable :
 ↳ sgdt, efags

en 98, vmware (pas encore ce nom) réussit à virtu x86
 → lit en avance et patch

Xen → fait sauter un requirem^t ⇒ modif de Linux ac hypercall



dom0 : vm qui a le droit à toutes hypercall
 domU : vms

dom0 manage les domU (les lance, schedule,...)

1 partie du virtualiseur est accélérée par CPU

→ extension de virtualisat^o

VMX → vmx root & vmx non-root : vt-x

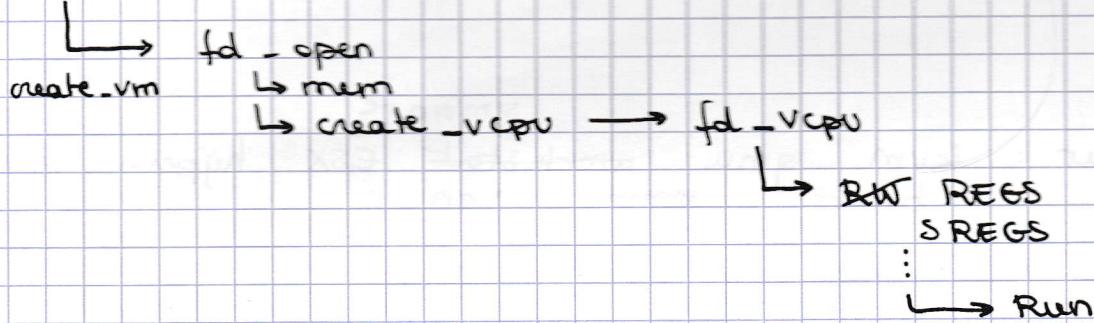
VMCS →

Il nous manque notre RAM et devices.

Ram → nesting sur paginat^o

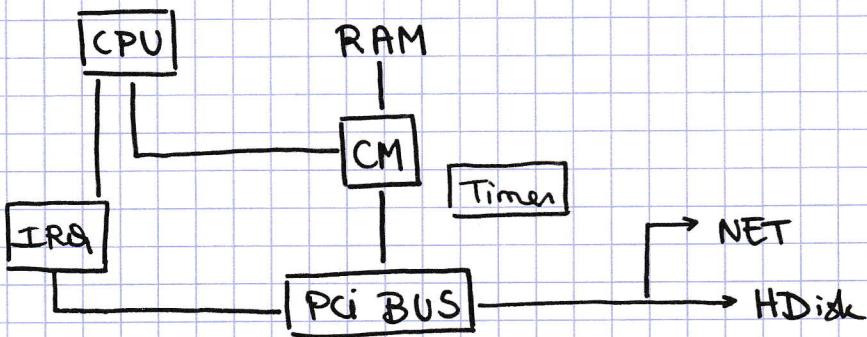
Device → zone mem IO, int mat, ports IO
 (cpt violat^o) (emule PIC (vmexit)
 charge mode) ↳ VMCS

/dev/kvm



Clients de /dev/kvm : qemu

Device à émuler : disque, carte réseau, contrôleur d'interruption, bus PCI, timer



OPERATING SYSTEM

Pourquoi avoir un file \$?

- + de place
- persistance
- partage de donnée

→ large storage

→ static

→ decoupled from processes

File \$: format & logique d'accès au stockage

Directory : liste de fichiers

Files : unité de stockage

Block : partie minimum d'un fichier / élément de base d'un fichier

Partition : partie du stockage

Qd construct° de format → data à stocker.

→ medium de stockage

Disk :

- + LBA
- + Block size (4kb, 2 kb, 512 bytes)
- + R/W/F (Read, Write, Flush)
- + Queue : Submit/Request, Completion → Ring buffer
 - ↳ potentiellement pls → pls processus en même temps

2 types de medium :

- HD (disque rotatif) vieux, lent, 1 submit queue
- Flash (ssd entre autre) pls requêtes à la fois.

Protocoles et controllers :

- + IDE (ATA) vieux, pas de queue, 2 nodes : accès synchrone / asynchrone
 - bus // (16 fils, 16 bits à la fois)
- + DMA : device qui écrit direct en mémoire et int qd finit
- + SCSI utilisé sur server. Bus série (1 fil, bit par bit à une certaine fréq)
on peut chaîner pls disques. → Rooteage de paquet
utilisé par que pour disque avant l'USB.
- + SATA / SAS au départ ATA en série, avec suff SCSI cmd.
- + NVME sur ssd → 500K requête IO/S.
en RAM → 4 M requête IO/S
64K queue de 64K requêtes

Ideal : mettre une requête dans la queue en 1 ItemIO

File \$: données + métadonnées.

1 fichier = 1 id unique (inode)

- name
- format
- type
- attributs selon file \$
 - date
 - proprio
 - ACL (Access control list)
 - archive
 - hidden

Types of files

- MSBOS : exec seulement certains types
- Mac OS : pleins d'inter
- Unix : pas de type.
(à part shared, blocked)

Struct dossier

- inode
- size_len
- name_len
- file_type
- name []

taille dir entry } les 2 en cas de changement de nom
du fichier.

accès à la data après la struct

nom du dossier → link
(pas ds inode)

→ pas besoin de taper ds tout le disque,
juste ds le dossier (ex: ls)

Taille max de nom de fichier ⇒ nb fichier ds dossier dépend
de nom de fichier

inode pas contraint par n° block

* block que inode

* inode que bouge

• inode alloués au début et stockés ds BTee.

Journalisat° : log des opérat°

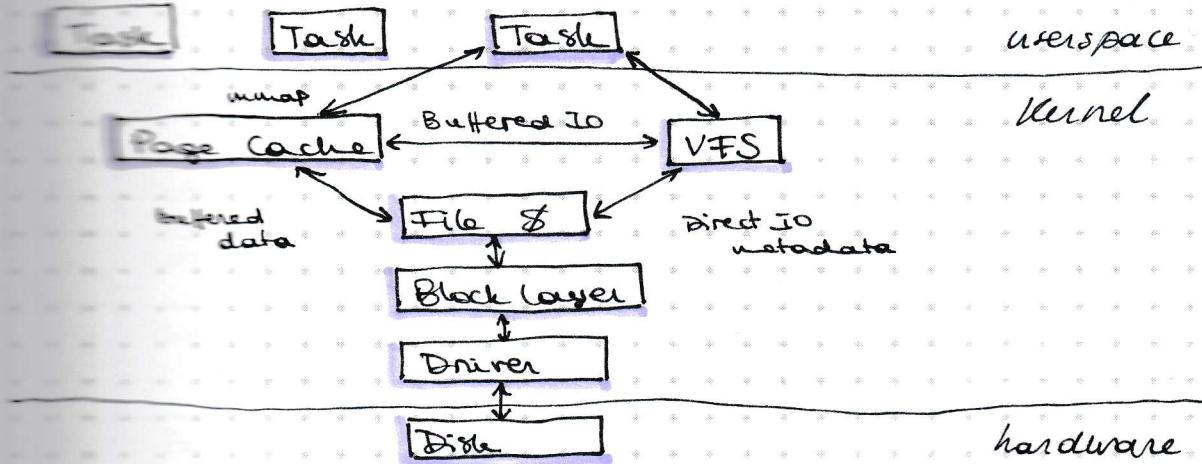
⇒ opérat° transactionnelles (set d'opé + commit ou rollback)

Page Cache

• inode : struct to address block of data

• dirent : gèle inode to filenames → directory entry.

Linux Kernel IO Architecture



⇒ IO :

- gd userland fait le caching direct (vm...)
- ne pas dévier + de mem au quest que celle qu'on lui a donné
- gest° de base de donnée (SGBD)

Se servir fichier etc → VFS. Etapes traduites sur le FS

Transferts ensuite sur block → r/w

Block layer : bcp de chose, scheduling de requête.

Buffered I/O : avec un cache. Si déjà data ds cache, on donne.
Sinon on refait tout le chemin de la requête.

Impact du cache important : presque tout est un fichier donc tout passe par lui.

Metadata fils en direct, seulement les datas dans le cache.

Cache très utile pour mmap!

au moment du sleep, page fault → il voit que mapping de fichier → vm_operations_struct : *fault

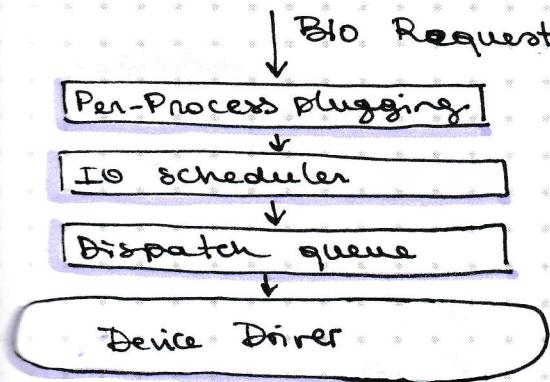
2 types de mapping : • private read → page cache write → duplique
• shared change → propagé → écrit dans page cache

Block layer:

→ requêtes IO : • start, r/w, taille
• hints (SYNC), flags (FUA, FLUSH)

Vie d'une requête

- créée dans block layer quand IO
- peut être delayed, merged (IO scheduler, multi queue handling)
- dispatch dans le bon driver
- driver signale qud IO finit



- NOOP : just pass requests into dispatch queue
- CFQ : Completely fair queueing
 - actif par défaut
 - sync > async
 - support IO priorities, cgroups, sync request idling...
- Deadline : read > write
 - reduce seeking : sort
 - dispatcher request at least after deadline expired.
→ le-tard possible