**Virtualizzazione**: tecnologia per creare una vista logica (versione virtuale di un sistema Hw o SW).

Cosa voglio fare:

* Eseguire diversi servizi in un ambiente sicuro, affidabile e isolato
* Eseguire applicazioni su diversi SO
* Eseguire l'ereditarietà di SO non più supportati
* Testare SW sviluppati in diversi sistemi operativi

Come? Ho bisogno di diverse macchine fisiche.

Virtualizzazione: separazione di una richiesta di un servizio dalla sua consegna fisica.

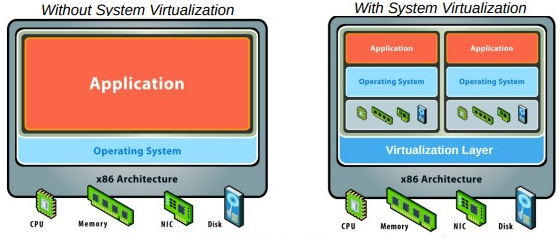
-->Accesso a più memoria di quanta ne abbia a disposizione fisicamente.

**Virtualizzazione di sistema:** permette di astrarre le risorse HW di un computer in diversi ambienti di esecuzione. Come se eseguissi simultaneamente diversi SO sulle stesse risorse HW.

Ogni SO ha l'illusione che lui abbia le risorse HW (o una porzione), ma hanno solo le risorse virtuali.

**Virtual machine**: ambiente di esecuzione che consiste in un SO con le sue applicazioni come le risorse virtuali che pensa di avere.

**Livello di virtualizzazione**: strato SW tra Hw e VM che partiziona, condivide e gestisce le risorse tra le varie VM.



//DIFFERENZA CON EMULAZIONE: essa usa un emulatore per runnare il SO, si finge un altro diverso (con la virtualizzazione è due copie di sé stesso), l'esecuzione è fatta sul SW. Non molto efficiente\\

Componenti della virtualizzazione:

* **Host**; macchina fisica che runna il VMS
* **Host SO**: SO che runna sull'Host
* **Hypervisor** (virtual machin manager): crea e runna delle VMs provvedendo a loro una piattaforma hardware virtuale equivalente a uno degli host. Dà l'illusione che ogni VM controlli l'host.
* **VM/guest**: entità che runna e controlla il VMM
* **SO guest**: SO che runna il guest.

**VM:** costrutto software che mima le caratteristiche di un computer fisico è configurato con risorse HW virtuali (numero di processori, RAM, memoria, porte network...).

Una volta creato, il VM può essere alimentato a livello fisico, caricato con SO e applicazioni e utilizzato come un computer fisico ma vede solo le risorse con cui è configurato, non quelle fisiche dell'host.

**SO guest**: accessi alle risorse HW virtuali che sono presentate dal hypervisor che gestisce le risorse per evitare accessi a risorse altrui e controlla la corretta esecuzione delle operazioni I/O delle operazioni su dispositivi fisici condivisi.

Le VM sono fatte di files, una VM è formata da pochi file:

* **File di configurazione**: descrive gli attributi del VM (processori allocati, RAM, I/O che gli hanno accesso, memoria a cui ha accesso...)
* **Files disco virtuali**: conserva il contenuto dei dravi HD della VM.

**Freeze dei VM in esecuzione**: sospendo VM creando dei snapshot che possono "ripartire" più tardi da dov'era arrivato. Uno snapshot può anche essere usato per creare un clone della VM o spostare una VM da una macchina all'altra senza modificare lo stato (cold migration).

**Templating**: strumento offerto dalle piattaforme di virtualizzazione per creare nuovi VM.

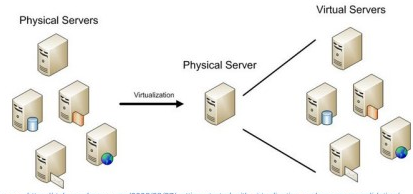
VM template: VM incompleto che non può essere alimentato, definisce configurazione delle piattaforme virtuali e ha il SO guest installato. Lo user o sistema di virtualizzazione modificando le impostazioni di configurazione che identificano il VM possono crearne un nuovo eseguibile.

**Migrazione live:** meccanismo, offerto da molte piattaforme di virtualizzazione, per spostatre un guest in esecuzione da un host all'altro senza interrompere le sue operazione o le connessioni attive. Migrazioni di: memoria, immagazzinamento, connessioni rete...

Approccio per migrazione memoria VM: **migrazione della memoria pre-copia**

* Fase pre-copia: le pagine vengono copiate sull'host di destinazione mentre il VM sta eseguendo sull'host originale. Se le pagine già copiate vengono modificate-->dirty.
* Fare stop-&-copy: VM in pausa e le pagine dirty vengono copiate sull'host di destinazione. Operazione svolta senza rallentamenti notevoli e interruzioni (seamless live migration).

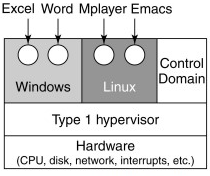
**Benefici:**

* provvisoning rapido: clonando una VM otteniamo una copia perfetta del server in pochi minuti, altrimenti posso usare i templates.
* Sicurezza ed isolazione performance: contenimento risorse arbitrario del VMM così ogni VM ottiene la quantità giusta. Un problema nel VM non affligge altri VM eseguiti nella stessa infrastruttura fisica
* Sviluppo e test rapidi: riuso dei sistemi di pre-configurazione, facilita il lavoro in team
* Scalamento virtuale delle risorse: se un VM ha bisogno di risorse HW, alloco altre risorse fisiche al VM. Se invece un VM ha più risorse, possono essere ridotte. Posso aggiungerne/rimuoverle senza spegnere il VM (resource hot-plugging).
* Consolidamento server: concentro tanti servers (come VM) su un PC per aumentarne l'efficienza.

**HYPERVISOR:**

Proprietà che un VMM deve soddisfare:

* Equivalenza (fidelity): il comportamento di un programma dev'essere uguale quando eseguito su VM e quando eseguito su HW.
* Efficienza: la maggior parte del codice eseguito in VM dovrebbe runnare direttamente nel livelllo HW sottostante senza intervento del VMM.
* Sicurezza: VMM deve avere il controllo completo delle risorse per la stabilità. Un programma Guest non può accedere alle risorse non allocate al suo VM. Il VMM può toglere risorse al VM.

Tipi di hypervisor:

* Tipo 1: bare-metal hyp.: strato SW che esegue sul HW sottostante.

+)Più leggero di un SW tradizionale.

E' come un SO speciale che invece di usare system call supporta la creazione, esecuzione e gestione dei VM.

Ha il pieno controllo della macchina.

ES: Microsoft Hyper-V, VMware ESXi, Xen...

Controlla tutte le risorse fisiche del livello HW sottostante (usando i drivers).

Supporta esecuzione VM e implementa i servizi core del SO.

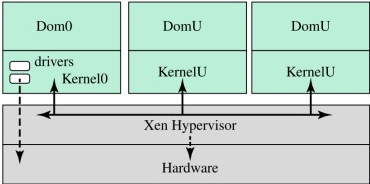
#Architettura 1: hypervisor implementa e provvede a tutte le funzionalità per eseguire e gestire i VMs. Funzionalità del SO ma ottimizzate per VM.

Gli hypervisor funzionano solo su piattaforme conosciute per cui hanno i drivers (di solito vengono eseguiti su server aziendali con poca differenza).

**VMKERNEL**: cuore del hypervisor, svolge le operazioni di virtualizzazione

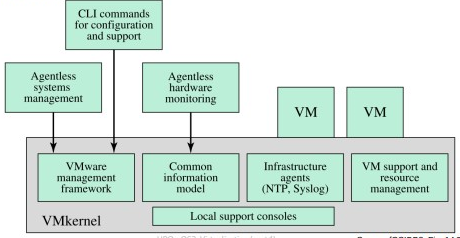
#Architettura 2: hypervisor si affida ad un SO esistente.

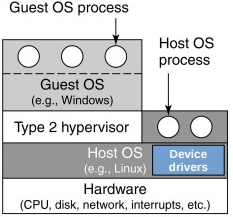
Hypervisor eseguito direttamente su HW.

un VM di servizio provvede ai servizi SO e gestione del hypervisor:

* Esegue un SO tradizionale
* Accesso privilegiato al HW
* fornisce servizi di base
* Crash della VM: critico

ES: Xen: servizio privilegiato = Dom0 (unico di solito esegue Linux come SO guest),

 gli altri sono non privilegiati = DomU. Non include drivers dispositivo o interfaccia utente, vengono messi dal SO guest di Dom0.

* Tipo 2: hosted hypervisor: un atro programma viene eseguito e gestito dall'host SO.

SO non consapevole della virtualizzazione in corso.

Può essere eseguito contemporaneamente ad altri programmi sul SO host.

Sfrutta meccanismi e drivers dell'host.

ES: Linux KVM, Oracle VirtualBox, Parallels desktop for Mac, Microsoft Windows virtual PC...

|  |  |
| --- | --- |
| **Tipo 1: bare-metal hypervisor** | **Tipo 2: hosted hypervisor** |
| Softwares eseguito nel livello HW fisico sottostante | Un altro programma eseguito e gestito dal SO Host |
| Host fisico-> controllo diretto risorse fisiche del host | Tra l'hypervisor e le risorse c'è un SO, si affida al SO per gestire interazioni HW |
| Prestazioni migliori e garanzie (nessuno controllo degli host del livello inferiore) | Prestazioni peggiori e meno garanzia (controllo del livello host SO inferiore) |
| Serve un server dedicato alla virtualizzazione | Non serve dedicare un server intero alla virtualizzazione (no privilegi delle radici) |
| Istallazione complessa | Istallazione semplice |

**Virtualizzazione CPU**: ogni VM è assegnata 1/+ CPU virtuali (vCPU).

Ogni vCPU non esegue istruzioni ma rappresenta lo stato della CPU come il SO guest che sia.

Per ogni VM, VMM mantiene 1/+ vCPU che rappresentano lo stato corrente della CPU.

Ogni vCPU è associata a 1/+ CPU fisiche (pCPU) che eseguono le istruzioni.

Context switch VM->pCPU le informazioni della vCPU vengono usate per caricare i giusto contesto.

Il VMM:

* divide il tempo di pCPU (time-sharing) tra le vCPU delle VM.
* fa da arbitro nelle contese e fa uno scheduling del tempo di pCPU.
* Intercetta le istruzioni SO guest->vCPU -->schedula il tempo sul pCPU host -->Manda richieste di esecuzione -->return del risultato al SO guest -->se guest occupa la pCPU per troppo tempo: VMM riprende il controllo e lo assegna ad un altro guest (**machine switch**).

La VMM durante un machine-switch deve:

* salvare lo stato della macchina del VM in esecuzione
* ripristinare lo stato macchina della VM da eseguire
* Ricominciare l'esecuzione della VM da eseguire

La VM come esegue le istruzioni?

#Tecnica 1: **interpretazione/emulazione**

VMM interpreta ogni istruzione del guest in SW riproducendo il comportamento delle istruzioni originali con gli stessi input e output.

La VMM mantiene ogni stato VM nel SW.

+)Implementazione semplice +)Portabile +)Completamente isolato: non esegue direttamente istruzioni del guest sul host +)Debug semplice -)Emulazione complessa -)Lento

Posso eseguire ogni istruzione direttamente nel HW?

No, solitamente il SO viene eseguito in kernel per poter eseguire ogni istruzione, ma con la virtualizzazione l'hypervisor è in kernel e il SO guest pensa di essere in kernel (kernel virtuale) ma è in modalità user per l'isolamento.

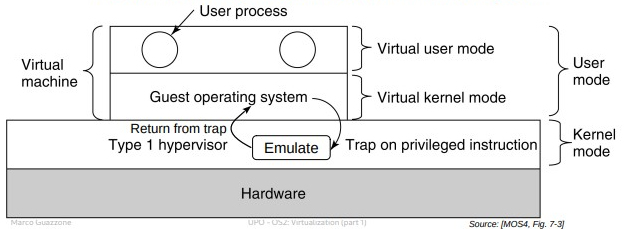
#Tecnica 2: **trap-&-emulate:**

La VMM affida la diretta esecuzione alla CPU.

Esecuzione istruzioni non privilegiate-> ok in HW. Istruzioni privilegiate-> Trap.

La VMM ottiene il controllo ed esegue/emula le istruzioni privilegiate che sono state provate dal SO guest ->quando finisce il controllo torna al VM.

Emulazione di istruzioni privilegiate deve evitare risultati inaspettati (es: disattivazione interrupt).



Istruzioni: privilegiate (trap se mod user), non privilegiate (non trap) e sensibili.

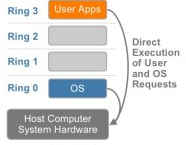
Istruzioni sensibili:

* **control sensitive**: prova a cambiare la configurazione delle risorse nel sistema (aggiornamento mapping da virtuale a fisico, comunicazione con dispositivi...)
* **behavior sensitive:** si comporta in modi diversi in base alla configurazione delle risorse.

Posso creare un "trap & emulate" solo se le istruzioni sensibili sono anche privilegiate.

Ogni istruzione sensibile deve andare in trap se eseguita in user mode.

E' utilizzabile nelle architetture x86?

Solo dal 2005 in poi: il VMM doveva avere più privilegi del SO e c'erano 17 istruzioni sensibili senza trap.

**Protection ring:** 4 livelli di privilegio per accedere al HW. Il codice eseguito in un anello privilegiato può leggere/scrivere in un anello meno privilegiato ma le chiamati di funzione tra anelli possono accadere solo attraverso meccanismi HW rinforzati. Le istruzioni privilegiate sono solo dell'anello 0.

Problema 1: il Guest pensa di essere al ring 0 ma non lo è.

Problema 2: VMM deve runnare ad un livello inferiore del guest, se il guest fosse nello 0, la VMM dove sarebbe?

Sol: VMM runna nel ring 0 e il guest SO in uno >0.

* Schema 0/1/3: VMM in ring 0> SO guest in 1 >proccesso user in 3
* Schema 0/3/3: VMM in ring 0> SO guest e processo user in ring 3 (costoso e poco usato)

Approcci per architettura non virtualizzabile:

**#Tecnica 3: Dynamic binary translation**

Se un programma guest di livello utente esegue una istruzione, la VMM usa esecuzione diretta.

Se guest esegue: istruzioni privilegiate->trap&emulate; sensibili->sostituisce con un'altra equivalente non sensibile.

VMM legge le istruzioni del SO guest dinamicamente lavorando sui blocchi.

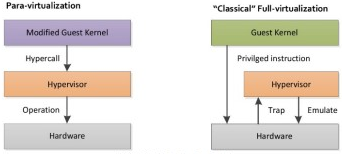
Per ogni blocco: prima dell'esecuzione scansiono per vedere se ci sono istr. sensibili, poi esegue il blocco tradotto.

Consuma tempo->ottimizzazioni:

1. considero solo il codice guest nel kernel: l'unico da analizzare e tradurre (semplice)
2. caching dei blocchi di base: metto in cache i blocchi tradotti in modo da non doverli tradurre in futuro.

**#Tecnica 4:Paravirtualizzazione (Virtualizzazione assistita dal SO)** rinuncia all'obiettivo si eseguire SO non modificati nel guest.

Modifico il SO guest per invocare direttamente l'hypervisor per eseguire un'istruzione privilegiata o sensibile. Esegue un sistema modificato.

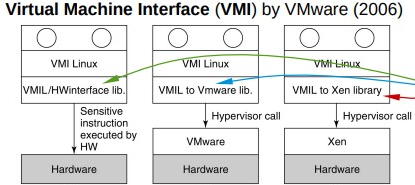
[FULL VIRTUALIZATION: il SO guest è completamente astratto dai livelli inferiori HW. Il guest pensa di essere in Kernell.]

Differenza para/full virtualization: nella paravirt. il SO guest sa di non essere in Kernel e collabora con VMM per rendere virtualizzate le architetture HW.

HYPERCALL: interfaccia virtuale che sostituisce istruzioni privilegiate o sensibili con SO guest modificato per ottenere risorse HW.

Costo del cambiamento del codice guest SO per la paravirtualizzazione?

Costo in crescita negli anni.

+)Implementazione semplice +)Meno costi di virtualizzazione

-)Poco portabile (dipende da VMM e hypercall)

-)SO non modificabili

-)Difficile mantenere il passo con i nuovi rilasci kernel (poca mantenibilità) ->parzialmente risolto introducendo l'interfaccia di virtualizzazione standard per interazioni guest-host.

**#Tecnica 5: Virtualizzazione HW-assistita**

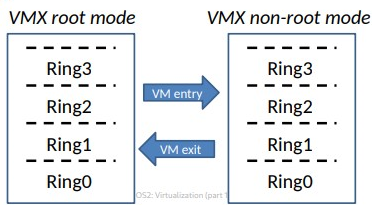
Full virtualization + trap&emulate (con esecuzione diretta).

Non modifca i SO, usa T&E + esecuzione diretta x eseguire istruzioni guest.

ES: Intel VTx: virtualization technology.

VTx non cambia la semantica delle istruzioni x86 preesistenti, duplica lo stato di architettura visibile del processore e introduce 2 modalità di esecuzione oguna con il supporto di 4 aneli di privilegio.

a)**Modalità VMX root** (host mode): eseguendola in ring 0 dà accesso diretto a HW e permette esecuzione delle istruzioni privilegiate. VMM e host SO runnano così.

b)**Modalità VMX non root**: alcune istruzioni non possono essere eseguite direttamente in HW perché causano trap -> passo controllo alla VMM.

Non serve più il ring deprivilieging.

Virtual machine control structure:

struttura dati per memorizzare lo stato di una VM, include

guest area e host area.

2 nuove transizioni:

a)**VM entry**: transizione mod. root-->non root. Salva

l'host e carica la guest state area,

b)**VM exit**: transizione mod non root->root. Salva la guest (e altre info come la causa) e carica la host-state area. Cause: istruzioni non eseguibili in non-root mode, interrupt I/O, altre istruzioni e eccezioni... Quando un'istruzione causa una exit transition, il controllo viene trasferito alla VMM, quando ha finito anche lui, tramita una VM entry il controllo passa al guest.

+)Sviluppo più semplice della paravirtualizzazione

-)Tante transazioni->spreco.

Hypervisor tipo 2: di solito hanno + moduli kernel in esecuzione sul SO host che abilitano un supervisore per eseguire istruzioni privilegiate.

**RAM**

La VM ha bisogno, oltre alla CPU, anche della RAM, le partizioni VMM condividono lo spazio di memoria fisica tra le VM.

ES: host con 32GB RAM. Host SO + Hypervisor: 1GB; VM1: 16GB, VM2:8GB, VM3:4GB, avanza 1 GB

I SO guest pensano di avere pieno controllo della RAM, la memoria è uno spazio di indirizzi fisici rappresentato come un array di pagine contigue a cui ad ognuna di esse sè stessa o un processo.

**Memoria fisica guest:** RAM visibile al SO guest.

**Host di mem fisica**: RAM dell'host.

Ogni **indirizzo fisico guest** dev'essere mappato in un host/macchina fisica reale dal VMM

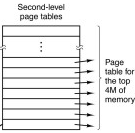
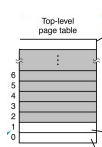
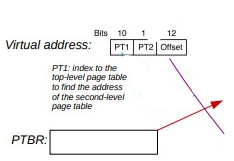
Come ogni SO usa la memoria virtuale per astrarre->illusione di avere un proprio spazio di indirizzamento virtuali.

Indirizzo virtuale: indirizzo di memoria dello spazio di indirizzi virtuale

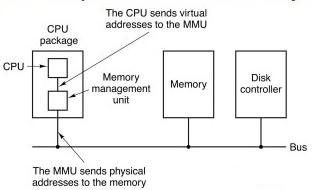
Indirizzo fisico: indirizzo di memoria dello spazio di indirizzi fisici.

Memoria virtuale: mappare pagine nello spazio di indirizzi virtuali in pagine dello spazio di indirizzi fisici. Questo mapping è definitio da tabelle di pagine che mappano i numeri di pagine.

Page-table base register (PTBR): registro di controllo CPU che punta alla top level page.



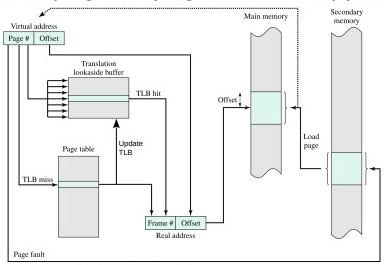
PTBR-->top level page table -->second level page table.

Indirizzo virtuale: PT1...PTn -->puntano dalla top level e alla second level con page frame

Offset-->Page frame.

La traduzione tra indirizzo virtuale e fisico è fatta completamente in HW della MMU, le page table vengono attraversate dalla MMU e la RAM non sa nulla della MMU e lavora con indirizzi già tradotti.

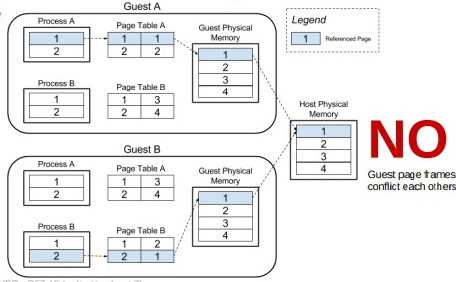
**Translation lookaside buffer (TLB):** piccola cache di mapping da pagine virtuali a pagine fisiche.

Il SO usa la memoria virtuale per i suoi processi e mappa gli indirizzi virtuali guest in indirizzi fisici guest attraverso la guest page table.

Guest virtual mem: mem virtuale visibile ad ogni processo utente del SO Guest.

Guest physical mem: mem visibile al guest come presentata dal VMM.

Guest page tables: tabella delle pagine per processo mantenute dal SO guest per mappare indir guest da virtuali a fisici.

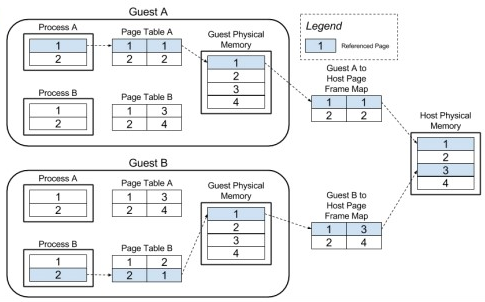
Host di mem fisica: memoria del sistema host.

Come mappo una pagina fisica guest in una pagina fisica host?

1. **num frame guest = nume frame host**, non funziona perché frame guest di diverse VM potrebbero mappare alla stessa pagina.
2. Per ogni VM: VMM ha una tab pagine per mappare frame guest in frame host.

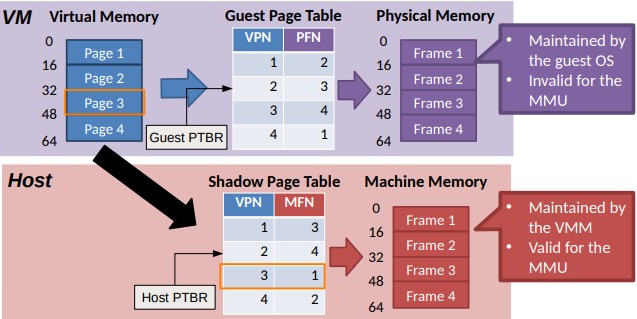
Le tabelle sono salvate nella RAM host fisica.

Funzionante ma lento (1 livello extra e accesso alla guest page table)



1. **Shadow paging** (full virtualization): per ogni processo di ogni VM, la VMM crea e mantiene una shadow page table che mappa: pag guest virtuali->pag host frame (fisico).

Il SO Guest non è a conoscenza della sPT.

Evita il mapping intermedio, mappa direttamente da guest virtuale a host fisico

VMM forza la cpu ad usare le shadow page table del guest in esecuzione per tradurre l'indirizzo e deve aggiornarle ad ogni cambiamento del guest sulla guest page table.

Come mi accorgo dei cambiamenti?

a)**Approccio write-protecting the gpt**: proteggo la gPT dalle scritture rendendo le page table delle read-only. Le modifiche non vengono accettate->trap al VMM->VMM applica cambiamenti a sPT e gPT e restituisce il controllo al SO guest.

b)**Virtual TLB**: il VMM concede l'accesso alla gPT senza intercettata da VMM e la sPT non viene aggiornata. Tentativo di accesso->manca nella sPT->page fault-> controllo passa alla VMM che decide se aggiungere i mapping mancanti alla sPT e eseguire le istruzioni.

Non posso rimuove un mapping->intercettata da VMM che se ne prende carico.

a e b)Tanti page fault causati dal guest che accedono a pagine fuori dalla RAM e page fault indotti dall'hypervisor legati alla sincronia tra gPT e sPT.

VMM deve intercettare e analizzare questi page fault:

Page fault guest->ritornano al guest; page fault dell'hypervisor->gestiti da VMM.

Costi dei page fault:

* Context switch: Guest->Hypervisor (salvo info sul fault, salvo stato CPU, carico stato hypervisor).
* Hypervisor gestisce il fault
* Context switch: Hypervisor->guest

1. **Paravirtualizzazione**: il SO guest è consapevole della virtualizzazione e finito il cambiamento del gPT informa il VMM.

Per ogni cambio: cambio gPT, hypercal che avvisa il VMM del cambiamento (hypercall meno costosa di un page fault). Ottimizzazione: non una hypercall a cambiamento, una per un gruppo di cambiamenti (quando cambia tutta la tabella).

1. **Nested paging** (traduzione indirizzi di secondo livello): sfrutta il supporto alla virtualizzazione fornito dall'HW a gPT e nPT.

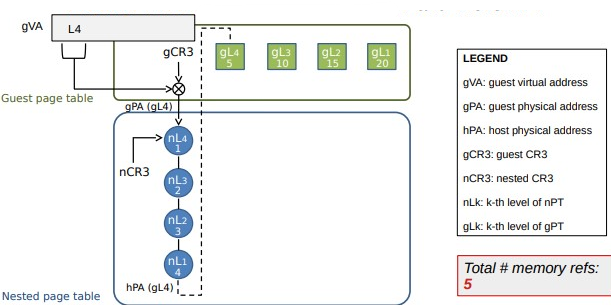
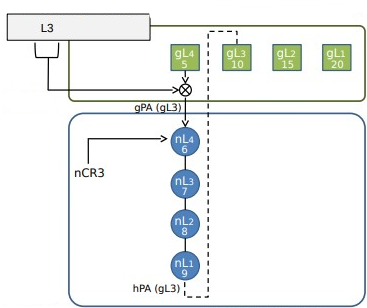
Nested page table: (incirizzamento rapido virtualizzato) dell'architettura AMD.

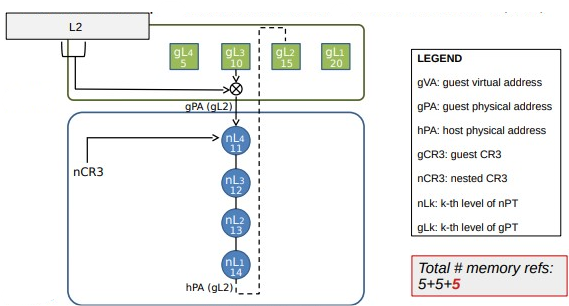
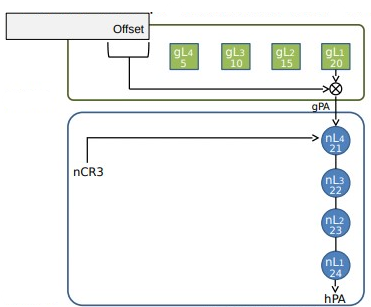
Extended page table di Intel.

Elimina il bisogno dello shadow paging (che è lento).

Cìè un registro Cr3 per i guest (gCR3) accessibile in modalità non-root e uno annidato (nCR3) accessibile solo in modalità root e puntano rispettivamente a gPT e nPT.

il SO guest non è a conoscenza della nPT e ha pieno controllo sulla gPT per cui nessuna VM exit può causare page-fault introdotti dal guest.

Se un guest prova a referenziare memoria con un indirizzo guest virtuale, un cammino 2d viene eseguito usando gpt e npt per tradurre indirizzo virtuale guest in fisico host.



Traduzione ind. guest virtuale->host fisico è costosa: con gPT a m-livelli e nPT n-livelli: (n+1)\*m+n accessi.

Mod. non-root: una entry TLB mappa una pag virtuale guest a una pag fisica host, ogni miss è molto costoso, se sono tanti->peggio della shadow paging.

Sono state proposte molte ottimizzazioni per ridurre gli accessi.

**Allocare e reclamare memoria**:

**Memory hot-plugging**: aggiungere e rimuover banchi di Ram virtuale durante l'esecuzione. Deve essere supportata da SO guest.

**Memory overcommitment**: la RAM totale allocata ai VM eccede la capacità del sistema host. (ES:32GB fisici e 3VM da 16GB l'una). Devo inserire meccanismi di gestione dinamica, alloca e reclama memoria da/a VM in esecuzione tramite:

* **Page sharing**: VMM conidivide pagine fisiche host identiche tra vari VM. Tecnica copy-on-write per gestire i cambiamenti: le modifiche non sono visibili agli altri. La VMM rimpiazza la pagina condivisa con una copia nuova che verrà assegnata alla VM così da essere immune ai cambiamenti. VMM controlla nTS->vedo le pagine usate. Pagine condivise sono in read-only.
* **Ballooning:** un modulo ballon è caricato in ogni VM come uno pseudo driver dispositivo che comunica con la VMM. Alloco pagine->gonfia->richiesta mem.

Dealloco pagine->sgonfia->può allocare altra memoria.