Cloud Computing

Appunti su parte A e B del prof. Vallati

Autore: Gabriele Frassi

Università di Pisa
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Artificial Intelligence and Data Engineering

prof. Carlo Vallati, ing. Carlo Puliafito

1 [A] INTRODUCTION AND FOUNDATIONS CONCEPTS

1.1 Evolution towards Cloud Computing

1.1.1 Introduzione

Poniamo alcuni aspetti storici che ci permettono di capire come siamo arrivati al Cloud Computing e come è possibile definire il Cloud Computing.

- Abbiamo una definizione del National Institute of Standards and Technology (NIST) che pone quanto segue: "Cloud Computing is a model for enabling **ubiquitous**, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort of service provider interactions".

  L'aggettivo chiave in questa definizione è *ubiquitous*: il Cloud Computing è ubiquitous, è presente in una molteplicità di applicazioni da noi utilizzate e l'utente medio non se ne rende conto (la posta elettronica, i contatti, la galleria – tutti quei servizi che permettono il passaggio di tali dati da un dispositivo a un altro)

- Il Cloud Computing non è un'idea recente: si ha un'introduzione teorica negli anni 60 ad opera di McCarthy (per intenderci la stessa persona che ha introdotto il concetto di "Artificial intelligence"): "computation may someday be organized as public utility', like water, electricity, gas, etc...".

  L'idea interessante è l'idea che le risorse computazionali di una macchina possano essere offerte come una public utility (società che offrono servizi come gas, elettricità, acqua… l'utente consuma e paga in proporzione a quanto consumato). Concetto utopistico: all'epoca le risorse computazionali disponibili erano insufficienti per poter attuare una cosa del genere. Il concetto è stato recuperato dopo più di quarant'anni e reso possibile grazie alle innovazioni emerse nel tempo (distributed computing, parallel computing, etc…)

1.1.2 Primo passaggio: *mainframes*

I primi dispositivi sono i cosiddetti mainframe, macchine di grandi dimensioni diffuse a partire dagli anni settanta.

- Presenti solo nelle grandi compagnie (prezzi elevatissimi, fuori dalla portata dell'utente consumatore e costosi anche per una compagnia).
- Abbiamo un unico dispositivo che svolge le tasks di molteplici utenti: questi forniscono l'input alla macchina (e ottengono l'output) per mezzo di un *dumb terminal* (*dumb* in quanto l'unica cosa che fa è trasferire gli input-output tra utente e mainframe)
- Approccio centralizzato che contribuisce alla formazione di bottlenecks. L'utente si mette in coda e attende un lungo periodo prima di avere l'esito della sua task (potrebbe anche ritornare nei giorni successivi).

1.1.3 Secondo passaggio: rivoluzione dei Personal Computer

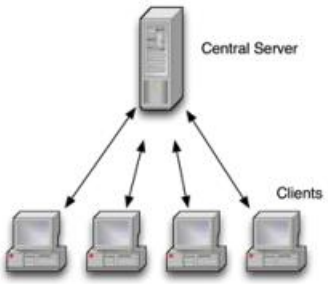
Verso la fine degli anni settanta si introducono i personal computer (PC), dispositivi meno costosi dei mainframes. Il loro costo rimane ancora elevato, se confrontato ad oggi, ma il calo ne permette l'ingresso nelle case dei consumatori.

- La potenza non è la stessa dei mainframes (inizialmente)
- Ogni utente utilizza un suo PC.
- Man mano che si va avanti i sistemi mainframes perdono importanza e vengono sostituiti dai PC.

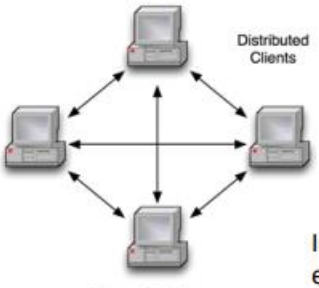
1.1.4 Terzo passaggio: reti di PC e applicazioni distribuite

Grandi società iniziano ad acquistare una molteplicità di PC, che vengono posti in più stanze. Ogni PC ha capacità computazionali proprie e lavora in autonomia. Emerge la necessità di far dialogare tra loro questi dispositivi.

- Nascono le reti di PC, nella forma di Local Area Network (LAN) e Wide Area Network (WAN).
- La bandwidth inizialmente è molto bassa.
- Nasce cosi il concetto di applicazione distribuita e i due paradigmi che già conosciamo:
    - *paradigma client-server* (dove abbiamo un server che offre servizi e una molteplicità di client che richiedono ai server tali servizi) e
    - *paradigma peer-to-peer* (la rete è caratterizzata da peer, nodi che possono svolgere sia il ruolo del client che quello del server)



Client / Server

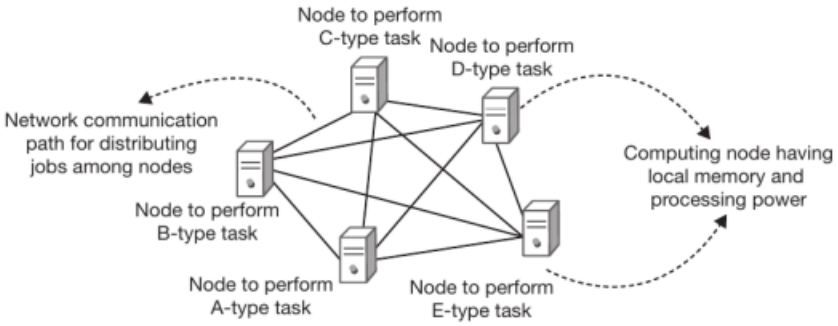


Peer to Peer

Interaction was limited to exchange of small data due to network limitations

1.1.5 Quarto passaggio: Parallel processing

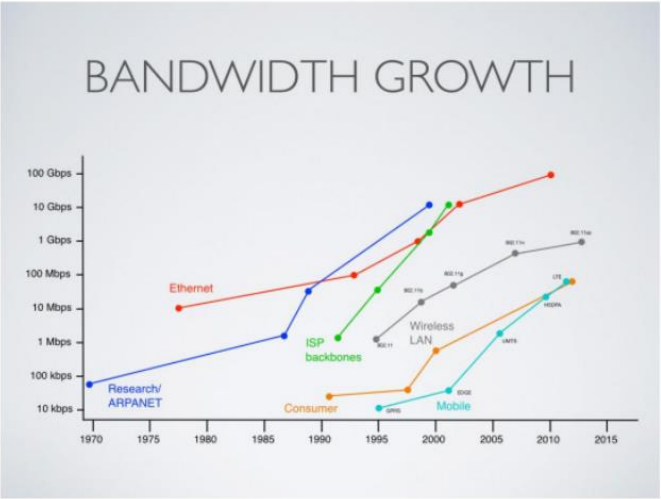
Si introduce a partire dagli anni 80 il concetto di parallel processing. Inizialmente si ritiene che l'unico modo possibile per migliorare la performance dei computer sia realizzare processori più potenti. Successivamente nasce l'idea di far lavorare insieme una molteplicità di processori (che possono stare sulla stessa macchina oppure distribuiti su macchine diverse) sulla stessa applicazione.



- Una task viene divisa in subtasks.
- Ogni subtask è assegnata a un sistema con risorse fisiche (in primis processore e memoria).
- I sistemi dialogano tra di loro per scambiarsi gli esiti delle subtasks e costruire cosi l'esito della task.

La generalizzazione del parallel processing è detta ***distributed computing systems***: "*a collection of processors interconnected via communication network to execute a complex task, impossible to be executed on a single PC*".

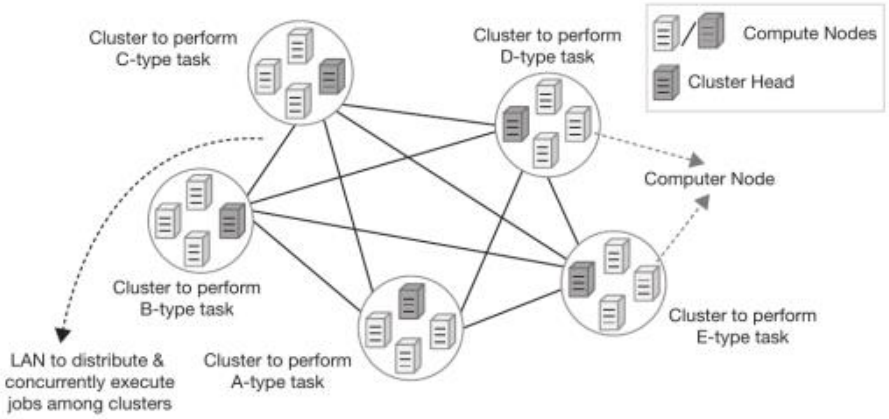
Questa cosa è resa possibile dall'aumento della bandwidth negli anni 80 (che arriva a 100Mbps per le LAN e 64Kbps delle WAN).



1.1.6 Quinto passaggio: *Cluster computing*

Un problema del distributed computing systems detto prima è che se uno dei nodi presenti nella rete fallisce rispetto alla subtask assegnata allora fallisce l'intera task. Altro aspetto è ridurre l'overhead dovuto alla comunicazione. Vogliamo cercare di affrontare queste problematiche e la prima idea è il *cluster computing*, dove la rete viene divisa in clusters.

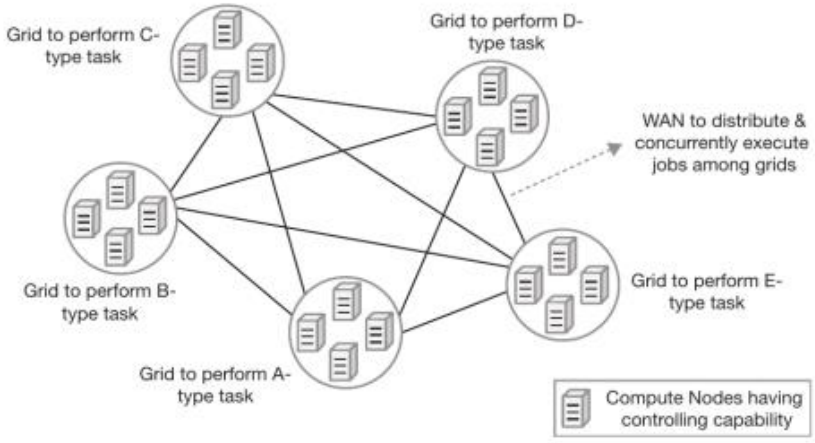
- Ciascun cluster è caratterizzato da nodi connessi per mezzo di una LAN comune. I nodi appartenenti al cluster svolgono la stessa task o task simili.
- Ogni cluster è caratterizzato da un cluster head che ha due compiti: ricevere le tasks e distribuirle tra i nodi appartenenti al cluster.



La presenza di molteplici nodi in un cluster permette l'introduzione di *ridondanza* e quindi un miglioramento della *reliability*, ma si introduce un approccio gerarchico e conseguentemente emerge il problema principe degli approcci gerarchici: il *single point of failure* (se viene meno il cluster head allora viene meno l'intero cluster).

1.1.7 Sesto passaggio: Grid computing

Si risolve il *single point of failure* introducendo a partire dagli anni 90 il *grid computing*. La rete è clusterizzata come prima, ma non si ha un cluster head e la responsabilità di coordinazione è distribuita su tutti i nodi che costituiscono il cluster. La soluzione è interessante e ci piace in quanto scalabile: se desideriamo aumentare la potenza computazionale è sufficiente incrementare il numero di dispositivi presenti all'interno di ciascun cluster.



Rimangono dei problemi:

- il sistema rimane *non fault-tolerant*, se fallisce un nodo fallisce l'intera task;
- non è possibile un *real-time scaling*, cioè alterare le risorse computazionali a infrastruttura alimentata (l'unico modo è spegnere il cluster, introdurre i nuovi nodi e riavviare l'infrastruttura);
- I dispositivi sono caratterizzati da architetture hardware eterogenee e non è facile farle interagire (problema principale: *code adaptation, code portability*).

1.1.8 Game changers: Hardware virtualization e World Wide Web

Il game changer, colui che permette l'emergere del cloud computing è la ***hardware virtualization***! Eseguiamo lo stesso codice su architetture differenti grazie all'introduzione di un *abstraction layer*!

- Si tiene conto della diversità tra architetture hardware
- Si permette real-time scalability.

Definiamo ***virtualization techniques*** (ci ritorneremo): "a layer of software over the hardware system that could simulate the whole physical system environment"

Un'altra killer application è il ***World Wide Web,*** che non solo permette la diffusione di contenuti ma anche la possibilità di realizzare interfacce grafiche semplici per gli utenti (grazie ai protocolli su cui si basa il WWW): si parla di *Web Based Technologies*!

1.1.9 Service Oriented Architecture

Le applicazioni sono diventate sempre più complesse negli anni, sia in contenuto che in dimensione. È necessario superare gli schemi classici e a tal proposito si introduce il paradigma noto come *Service Oriented Architecture* (SOA):

- metodo di sviluppo dove il software si basa su componenti dette *servizi*, che interagiscono l'una con l'altra;
- realizzazione modulare dove i servizi sono usati per realizzare applicazioni complesse, servizi che possono essere cambiati in maniera flessibile.

Perché ci piace questa cosa? Con moduli già realizzati lo sviluppo di applicazioni è più veloce.

1.1.10 Recap e date importanti

Ricapitoliamo. Abbiamo:

- ***Scalable computing infrastructure made of heterogeneous resources***
  Infrastruttura scalabile caratterizzata da risorse eterogene.
- ***Distributed computing environments empowered by high bandwidth networks***
  Infrastruttura distribuita dove l'elevata bandwidth permette scambio di una mole significativa di informazioni tra nodi.
- ***Collaborative environment across the world via web services***
  Web Services che favoriscono la collaborazione tra utenti con interfacce di facile utilizzo, anche a grandi distanze.
- ***Flexible applications via SOA***
  Applicazioni flessibili realizzate col paradigma Service Oriented Architecture (SOA).

Grazie alla virtualizzazione si introduce utility computing model, basato su un *pay per use model* e su *resources on demand*: vedremo entrambi i concetti. L'Utility computing evolverà in Cloud Computing.

Alcune date importanti.

- Il primo servizio di Cloud Computing su larga scala nasce nel 2006 con Amazon con Elastic Cloud Computing (EC2) e Simple Storage Service (S3).
- Non è il primo servizio di Cloud Computing in assoluto: il primo viene lanciato nel 1999 da salesforce.com
- Nel 2009 anche Microsoft entra nel settore con Microsoft Azure.
- Il termine Cloud Computing viene usato per la prima volta (nella concezione attuale) dall'amministratore delegato di Google (Erich Schmidt) nel 2006, durante una conferenza.

1.2 Technology foundation concepts

1.2.1 Definizione rigorosa di Cloud Computing (inizio)

Introdotti alcuni passaggi storici e alcune idee di base sul Cloud Computing possiamo introdurre una definizione più rigorosa di Cloud Computing.

- **A cosa fa riferimento il Cloud Computing?**
  Il Cloud Computing è una parola usata con grande frequenza e molto spesso in maniera inappropriata. In generale si fa riferimento a due cose:
  - ai ***cloud services*** (pezzi di codice eseguiti su *cloud infrastructures*)
  - a ***cloud infrastructures*** (spazi dedicati per ospitare *cloud services*) …

  Detta in altre parole: cloud computing viene usato per fare riferimento sia alle applicazioni sia all'infrastruttura dove vengono eseguite le applicazioni.

- **Definizione rigorosa di Cloud Computing.**
  Introduciamo la definizione rigorosa: *"cloud computing is a distinct IT environment designed for the purpose of remotely providing ~~scalable and measured~~ IT resources that are accessible via the Internet"*
  - Quali sono le risorse citate? Es: RAM e CPU
  - Chi offre queste risorse? I nodi che costituiscono la rete
  - Come vengono offerti questi servizi? Attraverso la rete Internet

Chiariremo più avanti le parole *scalable* e *measured*, per il momento ignoriamo!

1.2.2 Cloud services e paradigma Everything as a Service (XaaS)

Le risorse fornite dalla cloud infrastructure sono utilizzate per implementare cloud services. Un cloud service offre un servizio puro (*a pure service*) oppure è un'applicazione a tutti gli effetti.

- Un'applicazione è tipicamente accessibile dagli utenti (esseri umani) per mezzo di pagine web visitate con browser.
- Un *pure service* è "un pezzo di codice" utilizzato da altre applicazioni, che accedono ai servizi per mezzo di un'interfaccia API.

- ***Paradigma Everything as a Service – XaaS***.
  Un paradigma è *Everything as a Service*: i cloud services implementano una vasta gamma di funzioni e tutto viene offerto nella forma di servizio. L'accesso ai servizi avviene ovviamente per mezzo della connessione internet.

Cloud File Storage Service

Files are uploaded to the cloud, from there they can be downloaded on other devices or shared with other users. The service is accessed via web interface or via dedicated application

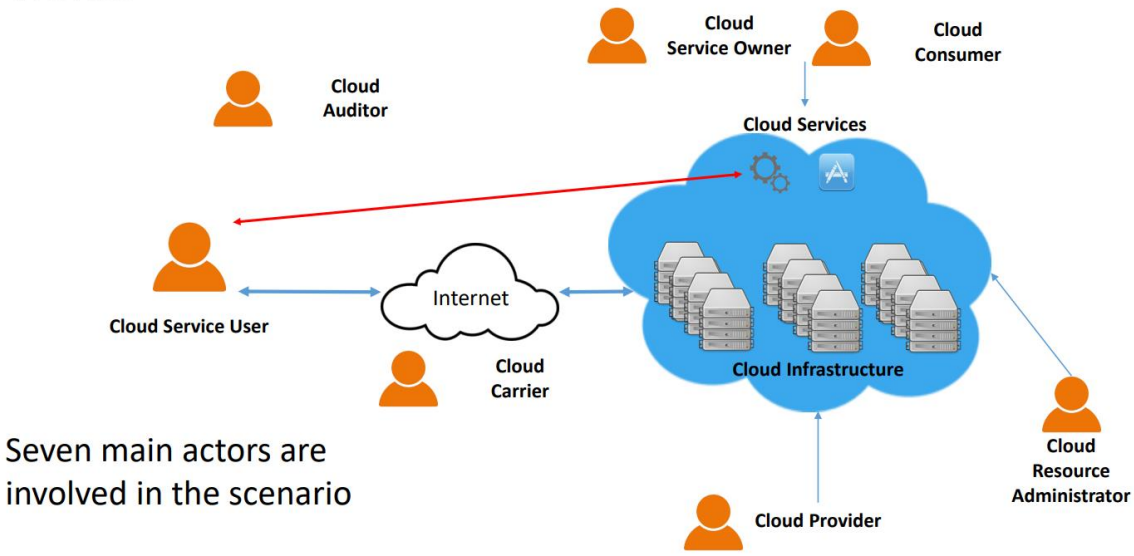
Cloud ERP Service

The ERP is deployed in the cloud. Employees access it through a web interface or a dedicated client (e.g. a smartphone application or a PC application) from anywhere

1.2.3 Utility-based model

1.2.3.1 Ruoli in un'infrastruttura Cloud

Enunciamo i protagonisti di un'infrastruttura Cloud!



Cloud Provider	Solitamente una compagnia con un business model. Organizzazione che fornisce gli investimenti per creare una *IT infrastructure*, la crea e la mantiene.
Cloud consumer	Organizzazione (o persona) che ha un contratto col Cloud Provider e utilizza le IT resources fornite dal Cloud provider per creare servizi.
Cloud Service Owner	Organizzazione o persona che crea un cloud service sfruttando le risorse offerte dall'infrastruttura del Cloud Provider. Attenzione! Molto spesso Cloud Provider e Cloud Service Owner coincidono, nel senso che chi realizza l'infrastruttura è anche colui che offre il servizio: si pensi a Google, ad esempio al servizio di posta elettronica Gmail.
Cloud Service User	L'utente finale, colui che utilizza il Cloud Service offerto dal Cloud Service Owner
Cloud Carrier	Il soggetto che fornisce la connettività tra utenti e cloud provider (l'ISP)
Si segnalano due ulteriori ruoli, secondari:	
Cloud Resource Administrator	Organizzazione o persona responsabile per l'amministrazione delle risorse disponibili sul cloud. Solitamente il soggetto fa riferimento al Cloud Provider, ma nulla vieta che faccia riferimento al Cloud Consumer
Cloud Auditor	Entità terza (non è utente, non è service owner, non è service provider) che valuta in maniera indipendente la qualità del servizio in termini di sicurezza e performance.

Nelle pagine relative al *Cloud NIST Model* è introdotto un ulteriore ruolo: il *Service Cloud Broker*.

1.2.3.2 Definizione rigorosa di Cloud Computing (conclusione)

Chiariamo i termini "scalable" e "measured" nella definizione di Cloud Computing.

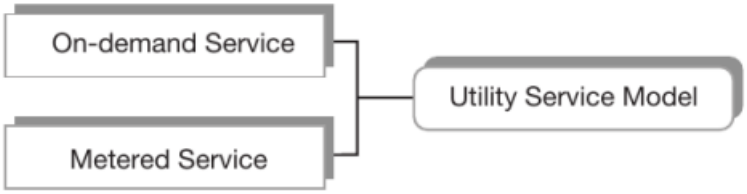
*"cloud computing is a distinct IT environment designed for the purpose of remotely providing **scalable and measured** IT resources that are accessible via the Internet"*

La presenza di questi due aggettivi, indispensabile per avere il Cloud Computing, porta a due cose:

- l'adozione di uno *utility-based model*
- la possibilità di allocare dinamicamente e in tempi brevi le risorse computazionali (il cloud provider alloca al cloud service owner).

Cosa intendiamo con questi due concetti?

- Con ***utility-based model*** intendiamo un modello dove le risorse computazionali sono offerte secondo un modello *pay-per-use*:
    - il Cloud Consumer richiede al Cloud Provider certe risorse: processore, RAM, bandwidth, etc…
    - l'uso delle risorse è misurato ("usage of resources is metered");
    - il Cloud Consumer paga in proporzione alle risorse effettivamente utilizzate ("they pay only what they get".

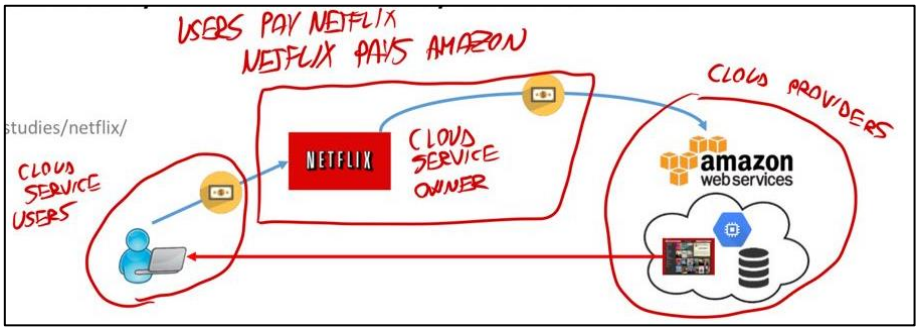


- Con ***allocazione dinamica*** intendiamo la possibilità per il cloud provider di allocare/istanziare in tempi brevi le risorse, mettendole a disposizione del cloud consumer. Tale allocazione può derivare da una richiesta esplicita umana, ma anche su richiesta di applicazioni. In ogni caso tale allocazione deve avvenire senza l'intervento umano.

1.2.3.3 Business model

Per comprendere i vantaggi dietro lo utility-based model dobbiamo analizzarne il Business Model. Esso è semplice: un cloud provider (Amazon, Google, etc.) vende *IT resources* a compagnie. Queste compagnie utilizzano le risorse acquistate per creare servizi/applicazioni per uso interno o rivolte ai loro customers.

Un esempio noto è Netflix!



- L'utente finale, il Cloud Service User, usufruire dei servizi offerti dalla piattaforma Netflix.
- Netflix, che è il Cloud Service Owner, ottiene le risorse necessarie per tenere in piedi il servizio con Amazon Web Services (Amazon è il Cloud Provider)

L'alternativa è la creazione di un'infrastruttura *in-house*, cosa che vedremo avere i suoi svantaggi.

1.2.4 Multi-tenancy

Dobbiamo ricordarsi che le risorse computazionali sono vendute a clienti differenti, non solo uno! Un'architettura tradizionale prevede la presenza di una molteplicità di utenti e distingue l'utente standard dall'amministratore (che installa e configura il sistema). Emergono due problemi significativi:

- Il numero di utenti è estremamente elevato e ciascuno ha i suoi requisiti, è impossibile per un amministratore gestire tutti questi requisiti;
- tra coloro che fanno uso di un'infrastruttura cloud fornita da un Cloud provider non ci sono solo Cloud Service User, ma anche Cloud Service Owner / Cloud provider (soggetti che non vogliono usare l'infrastruttura come meri utenti, desiderano anche controllare le risorse fornite);
- non è molto conveniente per il Cloud provider assegnare un server fisico a un unico soggetto.

Introduciamo di conseguenza un'architettura *multi-tenant* (dove *tenant* sta per inquilino).

- Inquilino? L'inquino paga un affitto per stare in un'abitazione, ma ha pieno controllo di questa: decide chi entra e chi esce dall'abitazione, può dipingere i muri, …
- I *tenants* non sono semplici utenti, sono amministratori del loro sistema e pertanto devono essere in grado di controllare il sistema in tutti i suoi aspetti.
- L'infrastruttura è congeniata in maniera tale che il cloud consumer abbia l'impressione di avere pieno controllo delle risorse assegnate

La virtualizzazione permette la creazione di un architettura multi-tenant!

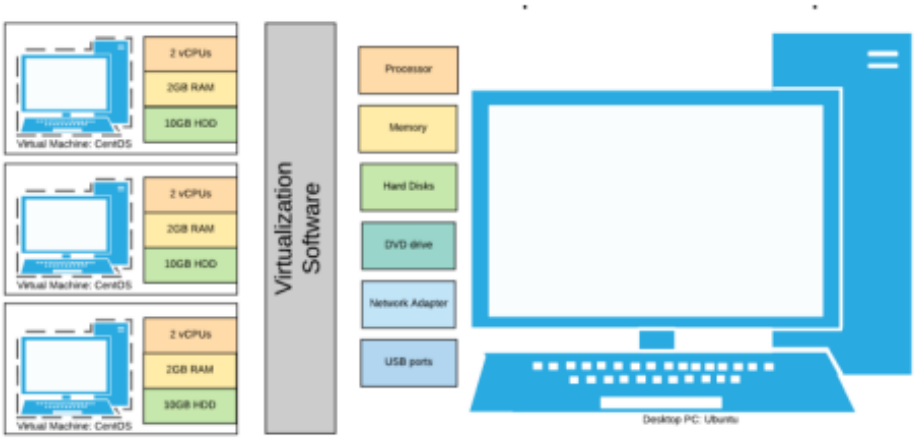
1.2.5 Virtualization

1.2.5.1 Concetto

La virtualizzazione permette la realizzazione di un'architettura multi-tenant.

"Virtualization is a broad concept, it refers to every technique that allows to create a virtualized version of something"

con *something* intendiamo IT resources (CPU, RAM, hard disk, server, …). Si prevede la creazione di molteplici copie virtuali delle risorse reali. Il cloud consumer accede unicamente alle risorse virtuali e ha l'impressione di averne il pieno controllo!

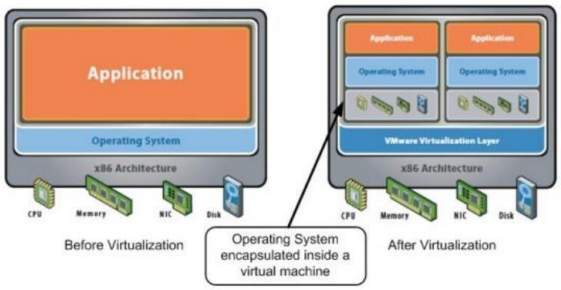


1.2.5.2 Hardware Virtualization

La forma di virtualization più popolare è la *hardware virtualization*, che permette la creazione di una rappresentazione virtuale di tutte le risorse di una macchina fisica (*a server creating a virtual server*).

- L'ambiente virtuale creato con la virtualizzazione è detto ***virtual machine*** (VM) e al suo interno è possibile eseguire un ***full operating system***.
- Il sistema operativo ospitato è detto ***guest operating system***.

L'esempio più noto (non relativo al Cloud Computing) è VirtualBox, tipicamente usato per eseguire un sistema operativo diverso da quello installato sul dispositivo.



- Hardware Virtualization allows multiple Operating Systems to run on the same hardware.
- They access the virtualized version of each resource

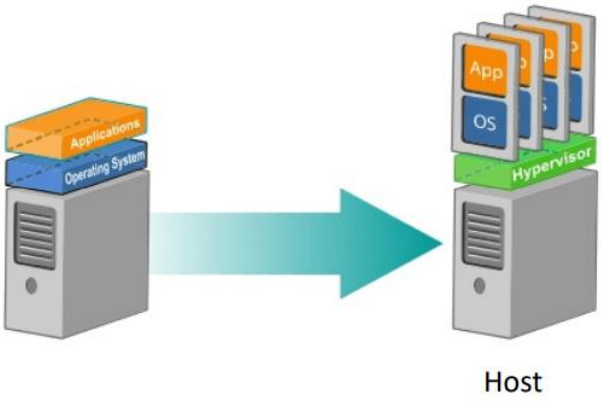
Virtualization is often used in commercial PCs to run a different OS than the one the PC has (e.g. Linux on Windows)

1.2.5.3 Hypervisor (o Virtual Machine Manager)

La virtualizzazione, in questo caso la hardware virtualization, è realizzata introducendo *un virtualization layer*: ciò che rende possibile l'introduzione di questo layer è la componente software nota come ***hypervisor***!

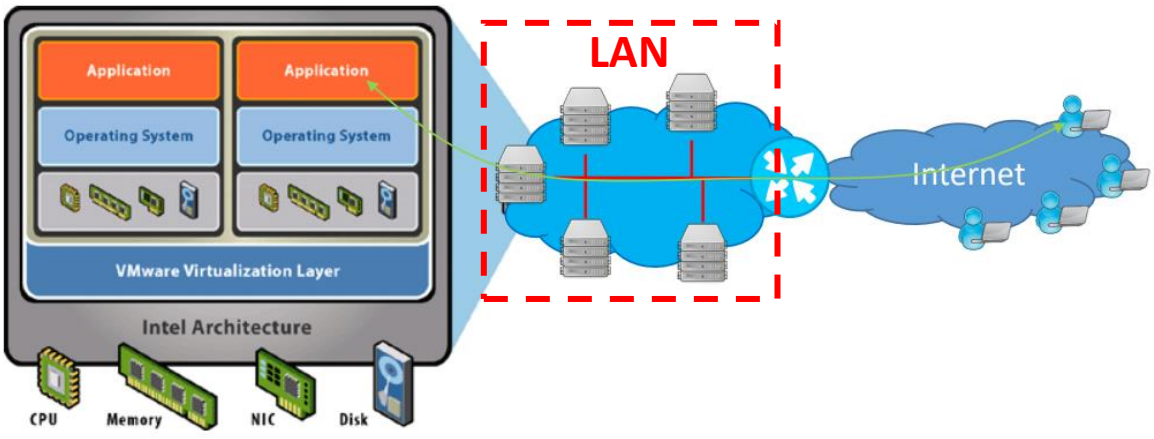
Con esso intendiamo l'insieme di tutte le tecniche di virtualizzazione richieste per ottenere un ambiente hardware virtuale (caratterizzato da risorse virtuali) in cui viene eseguito un sistema operativo.

- Il sistema operativo eseguito in questo ambiente virtuale è detto ***guest operating system***.
- La macchina fisica su cui viene eseguito l'hypervisor è detta ***host***.
- Il sistema operativo che ospita l'hypervisor è detto ***host operating system***.



1.2.5.4 Cloud Virtualized Infrastructure

In un datacenter ogni server è dotato di *hypervisor* per l'esecuzione di macchine virtuali, macchine create dinamicamente a seconda delle richieste dei cloud consumers (*on-demand dynamic provisioning*).

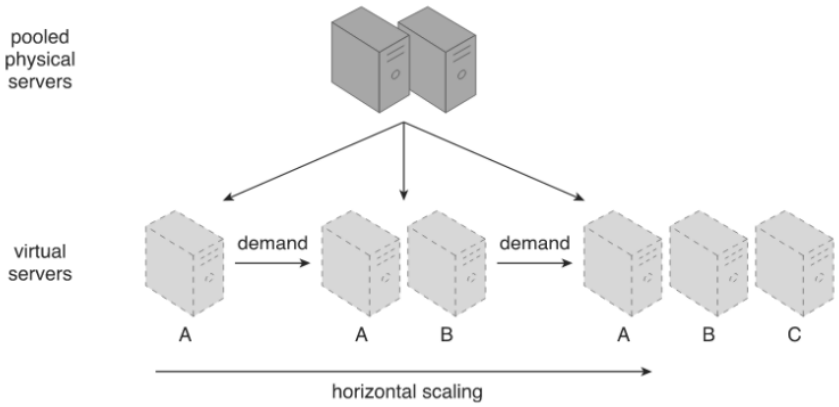


- La macchina virtuale è creata su uno dei server fisici del datacenter.
- I servers fisici del datacenter sono collegati tra loro per mezzo di una LAN ad alta velocità.
- Ogni macchina virtuale può comunicare con le altre macchine virtuali create nel datacenter, grazie alla rete LAN.
- Ogni macchina virtuale può comunicare con hosts esterni al datacenter, nell'ottica di offrire i propri cloud services: prima attraverso la LAN del datacenter e infine con la connessione Internet.
- Semplifica la *measurability of resources*: possibile fare ciò aggiungendo poche righe di codice nell'hypervisor. Maggiore overhead e misurazione meno precisa in *traditional computing*.

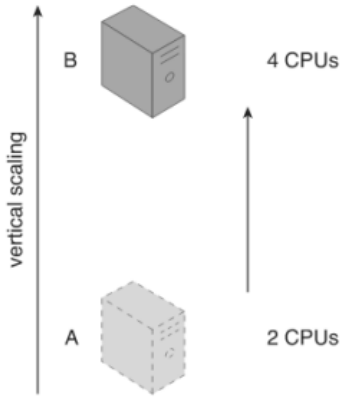
1.2.5.5 Vertical e horizontal scaling

Possibili due meccanismi per la scalabilità: *vertical scaling* e *horizontal scaling*.

- ***Horizontal scaling.***
  Allocazione (*scaling out*) o rilascio (*scaling in*) di nuove risorse svolto con l'allocazione/deallocazione di nuove macchine virtuali.
  - **Esempio 1**: inizialmente viene allocata una sola macchina virtuale, se questa non offre risorse computazionali sufficienti si aumentano tali risorse creando nuove macchine virtuali.
  - **Esempio 2**: il carico sull'applicazione si alleggerisce, il numero di risorse necessarie è calato. Possiamo deallocare macchine virtuali.



- ***Vertical scaling***.
  Una macchina virtuale è sostituita (o riconfigurata) con un'altra macchina virtuale aventi risorse maggiori. Si parla di *scaling up* quando le risorse aumentano, *scaling down* quando avviene il contrario.



Facciamo un confronto

Horizontal Scaling	Vertical Scaling
less expensive (through commodity hardware components)	more expensive (specialized servers)
IT resources instantly available	IT resources normally instantly available
resource replication and automated scaling	additional setup is normally needed
additional IT resources needed	no additional IT resources needed
not limited by hardware capacity	limited by maximum hardware capacity

- La *vertical scalability* è la più semplice in quanto l'applicazione può essere eseguita su una singola macchina senza necessità di alterarne la logica, tuttavia ci sono dei problemi: le risorse hardware

allocabili non sono infinite, sono limitate dalle risorse del server fisico che ospita la macchina virtuale. Tutto questo, inoltre, è molto più costoso per il cloud provider (*specialized servers*).
- La horizontal scalability è la più economica: otteniamo maggiori risorse per mezzo di commodity hardware components. Questa scalability richiede la modifica della logica dell'applicazione, che dovrà essere *cloud native*.
- Le risorse maggiori offerte con lo scaling sono immediatamente disponibili in caso di *horizontal scaling*, mentre la riconfigurazione in vertical scaling potrebbe provocare *downtime*.

1.2.5.6 Tipologie di virtualizzazione

Esistono diverse tipologie di virtualizzazione, scegliamo quella più appropriata in base ai requisiti che l'applicazione deve soddisfare:

- ***Full virtualization***
  Quella vista fino ad ora, realizzazione di un server virtuale completo, con tutte le sue componenti.
- ***Para-virtualization***
- ***Operating system virtualization***

Le differenze principali tra queste tipologie stanno nel livello di astrazione hardware e il livello di overhead presenti.

1.3 Benefit analysis and business models examples

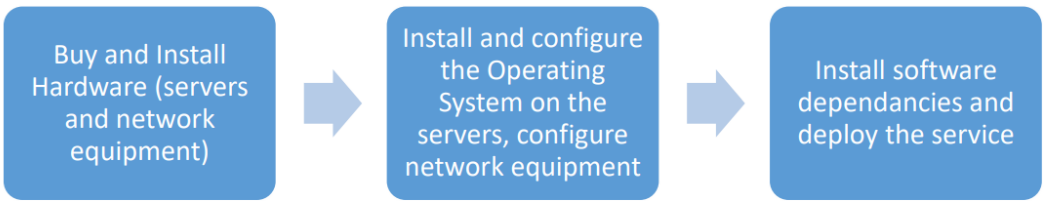
Cerchiamo di comprendere meglio il business model alla base del Cloud Computing.

- Cambia radicalmente il modo in cui i sistemi sono progettati e resi disponibili agli utenti.
- Molteplici vantaggi per le imprese a differenza dei paradigmi convenzionali.

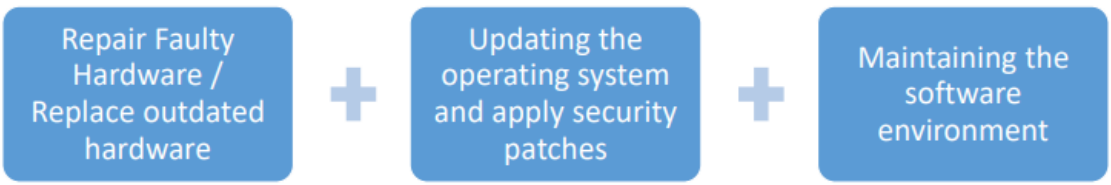
1.3.1 Convential computing paradigm

1.3.1.1 Costi

Soluzioni tradizionali sono implementate dalle imprese per mezzo di design e deployment di un'intera infrastruttura, realizzata *in-house*. Questo significa investire pesantemente per la creazione di questa infrastruttura (hardware e software), con costi che vanno dall'acquisto delle componenti che costituiscono l'infrastruttura al pagamento delle persone che hanno progettato e configurato il tutto.



I costi non si hanno solo per creare l'infrastruttura, ma anche per mantenerla a seguito della sua creazione (costi fissi per personale e riparazioni).



1.3.1.2 Scale the system

Le attività possono crescere in modo veloce e in maniera completamente inaspettata.

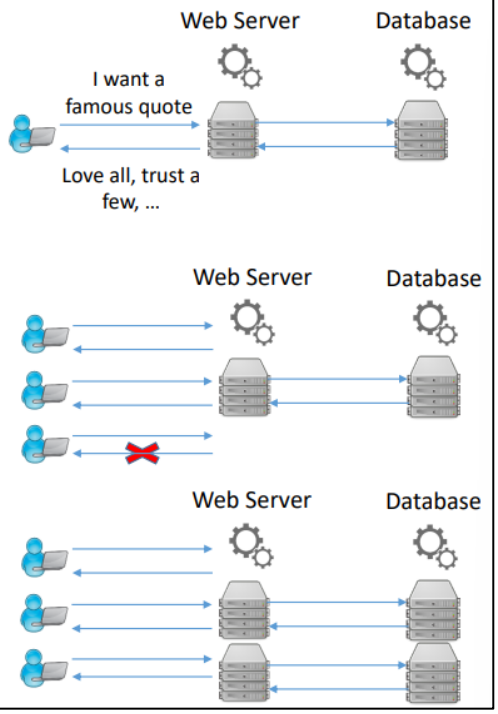
- Se le risorse della nostra infrastruttura sono insufficienti è necessario fare *scaling*: costi importanti per aggiornare l'hardware e introdurre così maggiori risorse. L'infrastruttura è adesso in grado di gestire il maggiore carico.
- Dopo un po' le attività calano drasticamente, ritornando ai livelli precedenti: il carico si riduce, ma le risorse acquisite non sono utilizzate (spreco, soprattutto se il calo di attività è permanente).

The system cannot scale down, acquired resources cannot be uninstalled and sold

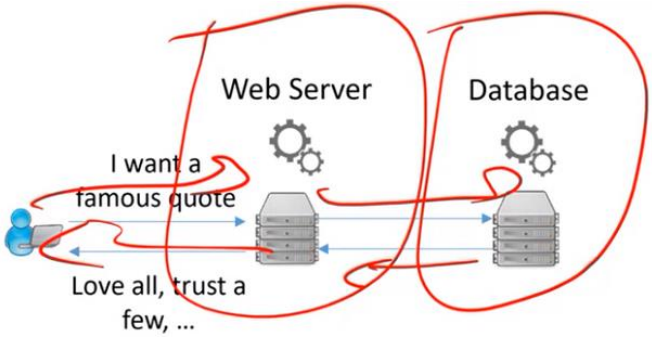
Consideriamo il seguente esempio: “Famous Quotes as a Service”

Example

- SmartQuotes ltd is a small company that provides to its customers a ‘Famous Quotes as a Service’
- Through a website customers can request for personalized quotes based on quotes from famous people
- Using the conventional computing paradigm, the company has to:
  - Hire 2 computer engineers to manage its small infrastructure
  - Buy two servers, one to host a database and one to host the web server that replies to the requests for quotes
- As the business expands, the number of requests from the customers increases, the current infrastructure cannot handle the load: responses are delayed or lost
- To handle the load, the company has to buy 2 more servers

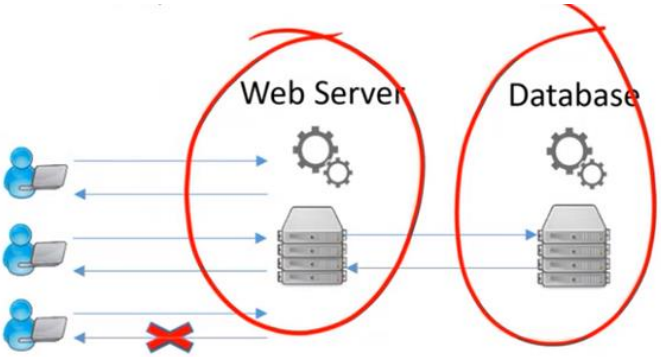


- **Cosa serve per sviluppare questa idea di business?**
  Un webserver per l’interfaccia utente e un database dove sono mantenute le citazioni. L’utente dialoga col webserver, questo si rivolge al database per gestire le richieste.



  Oltre a questo, si assumono due *computer engineers* per gestire l’infrastruttura.

- Supponiamo che un influencer parli di questo servizio su Instagram (Chiara Ferragni, *not the most appropriate example at the moment* cit.). Si passa da dieci utenti a diecimila utenti.
- Il webserver e il database non sono in grado di gestire il maggior numero di richiesto: maggiore lentezza nel fornice le citazioni, in alcuni casi le richieste non vengono portate a conclusione.



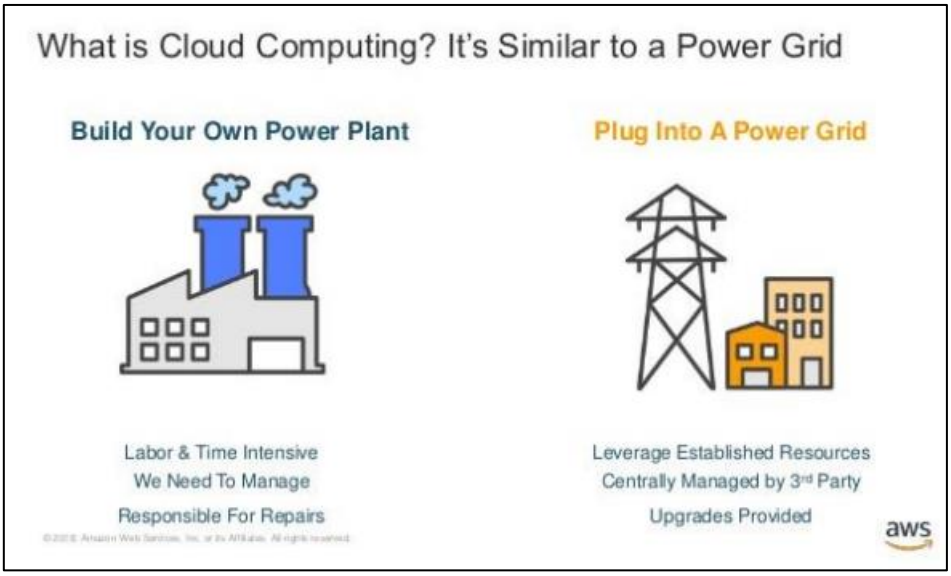
- Costi per acquistare un ulteriore *webserver* e un ulteriore database.

1.3.2 Vantaggi del Cloud Computing

Il Cloud Computing model è *utility-based*: il cloud consumer paga al cloud provider una cifra proporzionale alle risorse utilizzate (*pay per use model* grazie alla misurazione delle risorse utilizzate – *metering*). I costi sono sempre significativi, ma proporzionali all'utenza che accede ai nostri servizi (e quindi costi sostenibili).

Queste cose ci permettono di avere i seguenti vantaggi

- ***dynamic scalability***
  - La scalabilità aiuta nell'aggiunta di nuove risorse all'infrastruttura virtuale nel giro di pochi minuti (*more computing resources on-demand*).
  - L'allocazione di servizi avviene in maniera automatizzata: la richiesta può essere posta dall'essere umano, ma ancora più interessante è la possibilità che una certa applicazione richieda autonomamente ulteriori risorse.
    - Esempio: l'ammontare di richieste supera un certo *threshold*, viene lanciata una richiesta di maggiori risorse al provider in maniera del tutto automatica.
    - Esempio 2: *user base* ridotta, si chiede la provider la riduzione delle risorse.
- ***No fixed costs***
  - Non gestiamo direttamente un'infrastruttura: non c'è il costo iniziale per la creazione dell'infrastruttura e non ci sono i costi di manutenzione.
  - Non è necessario assumere personale specializzato per mantenere l'infrastruttura.
- ***Cost reduction***
  - Le risorse sono offerte a costi decisamente più bassi di quelli di un'infrastruttura in-house.
  - Tre ragioni:
    - ***Large***
      Il cloud provider acquista hardware in grandi quantità (bulks), creando così una grande infrastruttura. Il prezzo è più basso quando si acquistano grandi quantità.
    - ***Shared***
      Molteplici utenti utilizzano la stessa infrastruttura (*tenants*), l'uso delle risorse è massimizzato (*no resources are wasted*).
    - ***Fixed costs are shared***
      Il cloud provider assume personale per gestire l'infrastruttura, ma lavorano su un'infrastruttura utilizzata da più utenti (ergo il costo è distribuito tra gli utenti).



[Utility market] Abiti in casa e hai bisogno di corrente elettrica:
ti colleghi alla rete elettrica o costruisci una centrale elettrica?

1.3.3 Vantaggi indiretti del Cloud Computing

Vi sono ulteriori vantaggi:

- ***Minimal Management Responsibility***
  Gestire un'infrastruttura comporta grandi responsabilità, anche legali. Si pensi anche alla sicurezza delle informazioni memorizzate sull'infrastruttura. Il cloud pone la quasi totalità di queste responsabilità in capo al Cloud provider. Il personale assunto dal cloud provider si dedica a gestire tali responsabilità.

- ***Higher Quality of Service***.
  Il Cloud provider ha personale che si dedica interamente al mantenimento dell'infrastruttura, a differenza delle *traditional computing enterprises*. L'impegno non è solo nel mantenere l'infrastruttura, ma anche nel garantire la più alta qualità di servizio possibile (in termini di performance: response, bandwidth, cpu availability…) con l'adozione di *best practices*.

- ***Proprietà tipiche (che riecheggiano frequentemente)***
  - ***Continuous availability***.
    Le infrastrutture sono progettate per funzionare 24h su 24, 7 giorni su 7, grazie a *redundancy, failure recovery, failure predictions*…
  - ***Reliability***.
    Reliability significa che il sistema non solo è disponibile, ma funziona correttamente: *load balancing, backup practices, recovery procedures* garantiscono un recupero veloce del sistema in caso di *hardware failures*.

- ***Minimal software management***.
  L'infrastruttura richiede il pagamento di licenze per i software utilizzati (Esempio: Windows Server, se pensiamo al sistema operativo). Nel caso del cloud la maggior parte di queste licenze sono in capo al cloud provider. Non è solo una questione di costi, ma anche di gestione degli aggiornamenti: anche questo è un problema del cloud provider.

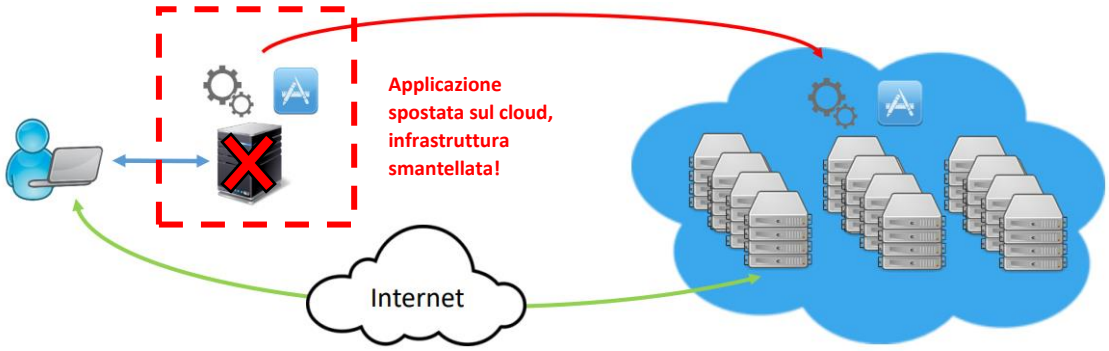
- ***Location independent***.
  I cloud services sono accessibili da qualunque parte e in qualunque momento con Internet.

- ***Companies are focused on their business***. Le imprese non devono più pensare al mantenimento dell'infrastruttura, possono porre il focus sulla loro idea di impresa.

1.3.4 Cloudification

1.3.4.1 Concetto

I vantaggi del Cloud Computing hanno dato luogo al fenomeno della cloudification



"moving application and services from local computing deployments to somewhere into the Cloud"

Assistiamo a un numero significativo di imprese che hanno smantellato la propria infrastruttura *in-house* per passare al Cloud. Non abbiamo bisogno di cambiare una singola riga di codice dato che la macchina virtuale assomiglia in tutto e per tutto alla macchina fisica.

1.3.4.2 Use-case: grandi imprese (Expedia)

Grandi imprese hanno ridotto i costi fissi grazie al passaggio al Cloud. Expedia inizialmente aveva una sua infrastruttura: nel 2017 ha deciso di spostare l'80% dei suoi servizi su AWS.

- Ha ridotto i costi
- Ha minimizzato la *response latency* da 700 ms a 50 ms grazie all'infrastruttura di AWS!

Da Wikipedia. *Expedia Inc. is an online travel agency owned by Expedia Group, based in Seattle.[1] The website and mobile app can be used to book airline tickets, hotel reservations, car rentals, cruise ships, and vacation packages*.

1.3.4.3 Use-Case: startups (Airbnb)

La categoria che ha beneficiato maggiormente dei vantaggi del Cloud Computing è quella delle startup, che hanno potuto porre il focus sulla loro idea di business senza front-up costs (soldi che vengono investiti prima del lancio del servizio, costi elevati se si decide di fare un'infrastruttura da zero).

Esempio: Airbnb, che ha basato fin da subito la propria infrastruttura su AWS.

- ***Airbnb:*** The San Francisco-based Airbnb began operation in 2008
- When the company was launched its entire infrastructure was based on AWS
- AWS scalability and flexibility allowed the company to grow considerably in short time, currently it has hundreds of employees across the globe supporting property rentals in nearly 25,000 cities in 192 countries
- https://aws.amazon.com/solutions/case-studies/airbnb/

1.3.4.4 Use-case: Sporadic High Performance Computing (Pixar)

Consideriamo un'altra situazione: compagnie che hanno bisogno di svolgere *very high performance computations sporadicamente*. Il Cloud Computing permette di fare questo senza la necessità di costruire un'infrastruttura in house: si acquistano risorse in grande quantità e limitatamente al periodo in cui queste sono necessarie.

Esempio: Pixar.

- ***Pixar:*** rendering an animation movie needs for each frame between 10 and 100 hours of CPU time. Multiply that by between 24 and 60 frames per second, and then multiply again for a circa ninety-minute movie, render a movie on a single machine could hit between 10 and 100 million processing hours (or between 100 and 1000 years)
- Use cloud resources only for the time needed to render the movie reduces costs and time needed for the process
- https://www.fasthosts.co.uk/blog/cloud/cloud-pixar-and-hollywood-computing

Operazioni complesse che vengono concluse in un paio di settimane!

1.3.4.5 Use-case: global-scale services (Google)

Con global-scale services intendiamo applicazioni/servizi che potenzialmente possono essere richiesti da chiunque nel mondo.

- Pensiamo a un motore di ricerca, poniamo questo servizio in un'unica posizione (anche con molteplici servers).
- Con milioni di utenti che tentano di accedere a quel servizio il collo di bottiglia diventa Internet: tutte le richieste del mondo vengono redirette in un'unica posizione.
- La soluzione è semplice: la compagnia che offre il servizio replica il proprio servizio in posizioni diverse del mondo.

I cloud providers offrono soluzione per questo problema: hanno molteplici datacenters sparsi per il mondo, offrono ai cloud consumer soluzioni ad-hoc per la replica di servizi.

Esempio: Google.

- **Google Search**, the most popular web search service (91.63% share of the market)
- *Google Search handles about 15 exabytes of data, according to some estimates. That's 15 billion gigabytes, basically more than 30 million household HDDs*
- *Google handles more than 65,000 search queries a SECOND! That's 3.9 million a minute, 234 million an hour, 5.6 billion a day, and more than 2 TRILLION a year*
- 16% of the searches are is new search terms never seen before. That means that about 896 million unique keywords are searched for every day!
- The most searched term on Google is 'facebook' with a monthly search volume of 2520000000
- *Do you think it is feasible the implementation of such service on a single location?*

2018 DATA

1.3.4.6 Use-case: data collection (General Electric Power)

Abbiamo sempre una global-scale application, ma il data-flow è invertito. L'applicazione deve raccogliere dati da molteplici dispositivi sparsi in tutto il mondo.

Esempio: General Electric Power.

- GE Power is the largest energy company in the world. In 2018, General Electric power plants produced one-third of the world's electricity
- GE Power has sensors embedded in their equipment (gas/steam turbines, electric generators, etc) to collect telemetry data to monitor equipment behavior to optimize their settings and detect malfunctioning
- A cloud-based data collection infrastructure allows GE Power to collect 500,000 data records per second, and scale to support the ingestion of 20 billion sensor-data tags from devices installed all over the world
- https://aws.amazon.com/solutions/case-studies/ge-power/

1.3.4.7 Use-case: content distribution (King)

Inverso del caso precedente: non abbiamo raccolta di dati da dispositivi, ma distribuzione di dati sui dispositivi (tanti dispositivi)! Questa cosa deve avvenire velocemente e in modo efficiente.

- Quanto distribuito può essere eterogeneo: video, musica, pagine web, *software updates* (sull'ultimo si pensi a Windows, alla gestione degli aggiornamenti di sistema o di Office di recente).
- Alcuni contenuti potrebbero richiedere una certa latenza o una certa bandwidth: bisogna tenere conto anche di questo.

Esempio: King (creatore di Candy Crush).

- King is a web gaming company funded in 2003. Now they produce web or mobile app games (Candy Crush)
- The company has the need to deliver game content to a global user base, with different network connectivity, from fast new networks (4G) to old mobile user networks (e.g. 2G or 3G). Regardless of location and technology, players need fresh content to use the game
- They exploit cloud providers for global content distribution in order to have an optimal and scalable global reach
- https://aws.amazon.com/blogs/aws/king-using-amazon-cloudfront-to-deliver-mobile-games-to-over-200-countries/

1.3.5 Challenges/Risks

Cloud Computing non è esente da svantaggi, se confrontato col *traditional computing*. Questi svantaggi suscitano scetticismo in molte imprese, che vedono nella *Cloudification* dei loro servizi un rischio.

- ***Network costs***.
  Cloud Computing richiede accesso costante alla rete, LAN e Internet: questo è un costo! Situazione migliorata rispetto al passato.

- ***Network Bandwidth (principale limite tra gli utenti convinti dal Cloud)***.
  It's still an issue (cit.). Se l'applicazione deve gestire grandi quantità di dati e la bandwidth non è adeguata allora la rete potrebbe fare collo di bottiglia.

- ***Data security (principale limite tra gli utenti meno convinti dal Cloud)***.
  Una delle principali ragioni per cui alcune imprese decidono di non passare al Cloud è sicuramente la paura di trasmettere dati (*precious data*, cit.) sulla rete Internet (non sicura) e di ospitarli su un'infrastruttura non propria. Sul primo punto può essere utile adottare forme di *encryption*, sul secondo *trust the cloud* (*high quality standards*, il cloud provider non accede ai dati che ospita, gli altri utenti presso il cloud provider non possono accedere ai nostri dati).

- ***Limited portability***.
  La standardizzazione in Cloud Computing è limitata, come vedremo nelle pagine successive. Grandi players, ciascuno con la propria infrastruttura proprietaria e il desiderio di prevalere rispetto agli altri competitors. Se creiamo un servizio su AWS sarà difficile eseguire lo stesso servizio su altre piattaforme senza modifiche di codice.

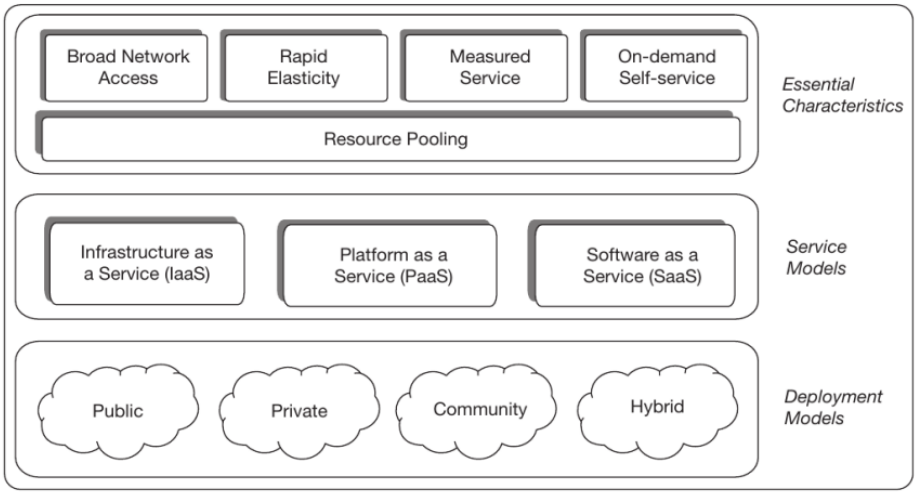
- ***Legal issues***.
  I cloud providers pongono i loro datacenters nei luoghi più convenienti. Potrebbe succedere che i dati del cloud consumer, soggetto avente sede in un certo stato, siano ospitati in un datacenter di un altro stato. Le leggi di alcuni stati potrebbero imporre delle restrizioni su alcuni *use-cases*, ad esempio situazioni in cui vengono trattati dati sensibili (esempio: *medical records*).

1.4 Cloud NIST model, service types and deployment models

La standardizzazione del Cloud Computing è limitata. Due aspetti: tecnologia recente creata from scratch; competizione tra le compagnie che per prime hanno investito nel Cloud Computing, conseguentemente rivalità in cui ciascuna compagnia cerca di far affermare in maniera aggressiva la propria architettura. L'interoperabilità rimane un problema.

1.4.1 Introduzione al NIST Model (Cloud Computing Reference Architecture)

Nonostante ciò il National Institute of Standard and Technology (NIST) ha elaborato un modello che fornisce tutti gli aspetti principali di un architettura Cloud e che risulta in buona parte implementato nelle architetture cloud più diffuse. Attenzione: definisce solo il set di elementi che costituiscono l'architettura, non come questa viene implementata!

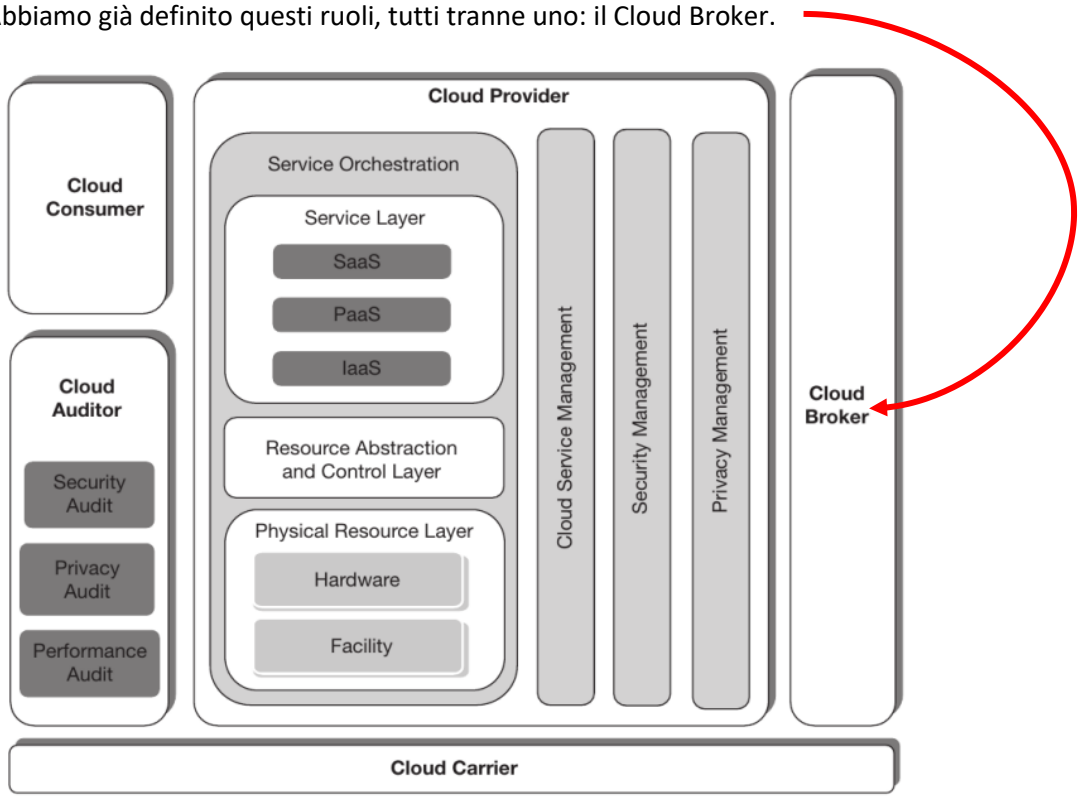


Il modello pone i seguenti aspetti:

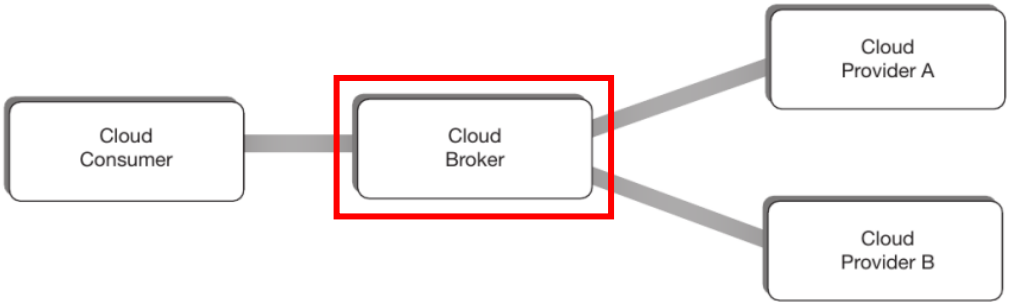
- **Essential characteristic**, caratteristiche obbligatorie per ogni cloud computing system
- **Service delivery models**, i modi con cui il cloud providers offre servizi
- **Deployment models**, i modi in cui l'infrastruttura cloud è offerta agli utenti.

1.4.2 Attori: quelli già visti e il *Cloud Broker*

Il modello NIST pone una serie di attori: Cloud Consumer, Cloud Provider, Cloud Auditor, Cloud Carrier e Cloud Broker. Abbiamo già definito questi ruoli, tutti tranne uno: il Cloud Broker.



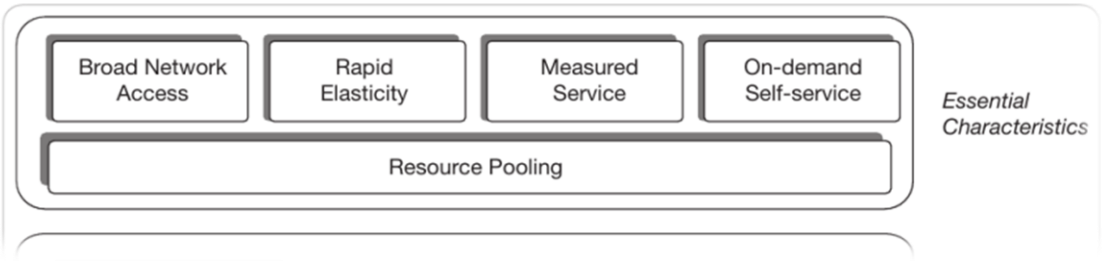
Il Cloud Broker è un soggetto terzo che si pone, quando presente, tra cloud provider e cloud consumer (una sorta di proxy). *"A cloud broker manages the delivery of cloud services from different providers to the consumer and negotiates the relationship".*



Il Cloud Broker utilizza questa sua posizione per ottenere profitto: acquista risorse dai Cloud Providers e le rivende ai Cloud Consumers. Il Cloud Consumer acquisisce risorse a costi più bassi (che si rivolge a più provider allo stesso tempo).

1.4.3 Essential characteristics

Il NIST model definisce cinque caratteristiche obbligatorie per una infrastruttura cloud:



- ***Broad Network Access***
  L'infrastruttura deve essere accessibile da qualunque punto
- ***Rapid Elasticity***
  L'allocazione e la deallocazione delle risorse deve avvenire rapidamente
- ***Measured Service***
  Le risorse sono misurate e il pagamento è proporzionale al loro uso effettivo (*pay-per-use model*).
- ***On-Demand Self-Service***
  Le risorse sono allocate su richiesta (che può essere umana o invocata da un'applicazione).
- ***Resource Pooling***
  Le risorse devono essere in grandi quantità. Devono soddisfare più utenti in contemporanea e questi devono avere l'impressione di avere a disposizione infinite risorse!

1.4.4 Deployment models

Un'infrastruttura cloud può essere creata e resa disponibile in modi diversi. I modelli variano in:

- collocazione delle infrastrutture;
- entità che gestisce l'infrastruttura rispetto all'organizzazione che utilizza le risorse, il Cloud Service Provider;
- i "boundaries" tra Cloud Provider e Cloud Service Provider.

1.4.4.1 Public cloud e Private cloud

I due principali modelli sono il public cloud e il private cloud

- ***Public cloud***.
  Il public cloud è la configurazione vista fino ad ora, la più popolare. Il Cloud Provider crea l'infrastruttura e la rende disponibile a qualunque Cloud Customer interessato (che ovviamente paga per accedere a tali servizi). I customers accedono ai servizi in remoto. In questa configurazione la multi-tenancy è promossa ai massimi livelli: l'infrastruttura è utilizzata da un numero di utenti molto elevato. Questo e la possibilità per il Cloud Provider di porre i suoi sforzi unicamente sulla manutenzione dell'infrastruttura permette la massimizzazione dei ricavi.
- ***Private cloud***.
  Il private cloud è una configurazione alternativa dove una compagnia che crea un'infrastruttura per uso interno: soggetti esterni alla compagnia non possono accedere!
  - Ritorna un problema già enunciato precedentemente: la compagnia deve creare e mantenere un'infrastruttura, i costi sono maggiori. Ritornano problemi, ma godiamo dei vantaggi del Cloud Computing (*virtualization* in testa).
  - Configurazione adottata quando la compagnia ha bisogno di un controllo completo dell'infrastruttura, o quando i requisiti sono molto specifici (es: dati sensibili).

La seguente tabella riepiloga le differenze

Private Cloud	*Public Cloud*
It can be both of types of on-premises and off-premises.	There cannot be any on-premises public cloud deployment.
On-premises private cloud can be delivered over the private network.	It can only be delivered over public network.
It does not support multi-tenancy feature for unrelated and external tenants.	It demonstrates multi-tenancy capability with its full ability.
The resources are for exclusive use of one consumer (generally an organization).	The resources are shared among multiple consumers.
A private cloud facility is accessible to a restricted number of people.	This facility is accessible to anyone.
This is for organizational use.	It can be used both by organization and the user.

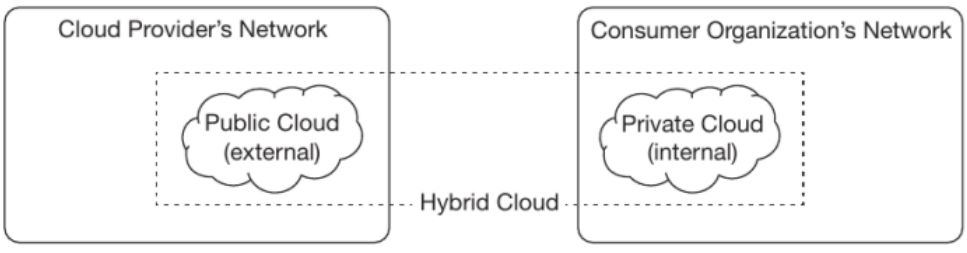
1.4.4.2 A form of generalized private cloud: community cloud

Una via di mezzo è *il community cloud*, una configurazione che prevede un consorzio di compagnie (attive nello stesso settore) che decidono di fare un investimento congiunto per creare un'infrastruttura.

- Tale infrastruttura è accessibile ai soli soggetti che fanno parte della community.
- Possibile applicare un *pay-per-use model*.

1.4.4.3 Hybrid cloud

Si parla di hybrid cloud quando *public e private deployments* risultano combinati. Infrastruttura caratterizzata da due componenti: una pubblica e una privata. In generale si tende a minimizzare la seconda utilizzandola solo per lo stretto necessario (esempio: servizi che trattano dati sensibili, oppure servizi con *low latency requirements*).



1.4.4.4 Scelta del modello più appropriato

Come scegliamo il modello più appropriato?

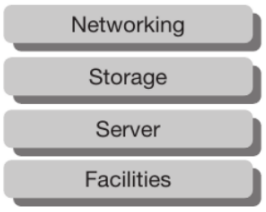
- Non c'è una risposta giusta o sbagliata, dipende dalle necessità della compagnia. Bisogna considerare tutti i vantaggi e svantaggi per arrivare a una decisione.
- **Regola generale**: public cloud deployment è la soluzione migliore per uso generale (reliable, simple, cheap), se non vi sono particolari requisiti.
- Altre questioni di cui tenere conto: *privacy concerns* (quando trattiamo dati sensibili) e costi derivanti dalla creazione di un'infrastruttura *in-house*.

1.4.5 Service Delivery models

1.4.5.1 Introduzione

Come vengono offerti i servizi ai Cloud Consumers? Introduciamo tre modelli, differenti su due fronti: tipologia di servizio offerto; modo in cui il servizio viene offerto al Cloud Consumer.

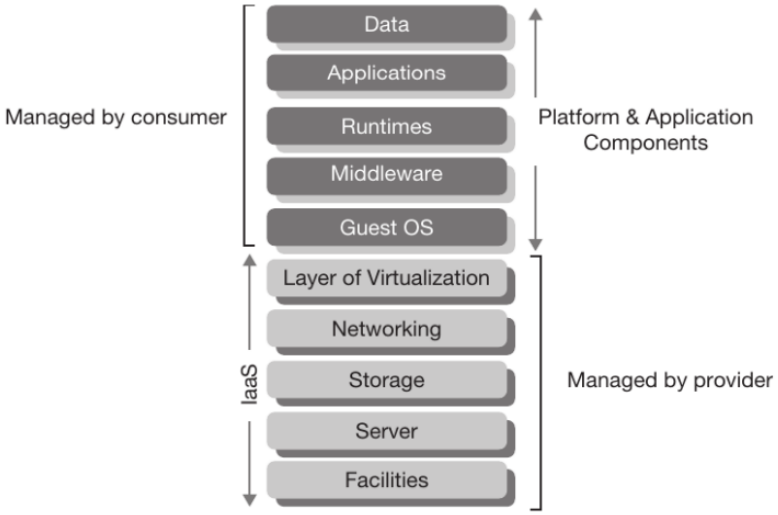
- Differenza significativa tra i tre modelli è il livello di astrazione fornito: maggiore è il livello di astrazione maggiori sono gli aspetti che il Cloud Provider deve gestire.
- Un modello convenzionale (non legato al Cloud Computing) prevede che il Cloud Consumer gestisca **<u>tutti</u>** i seguenti aspetti: creazione e gestione di un'infrastruttura fisica; installazione del sistema operativo; installazione delle applicazioni necessarie; sviluppo e pubblicazione dei servizi; gestione dei dati.



Cosa succede con i Service Delivery Models?

1.4.5.2 Infrastructure/Hardware as a Service (IaaS) – Minima astrazione

Il modello visto fino ad ora, il più popolare: il Cloud Provider fornisce Virtual Machines (con tutte le risorse hardware necessarie)! Il Cloud Provider si occupa unicamente delle componenti fisiche/hardware dell'infrastruttura, il resto è in capo al Cloud Consumer: installazione del sistema operativo; installazione e configurazione delle applicazioni necessarie; sviluppo e pubblicazione dei servizi; gestione dei dati.

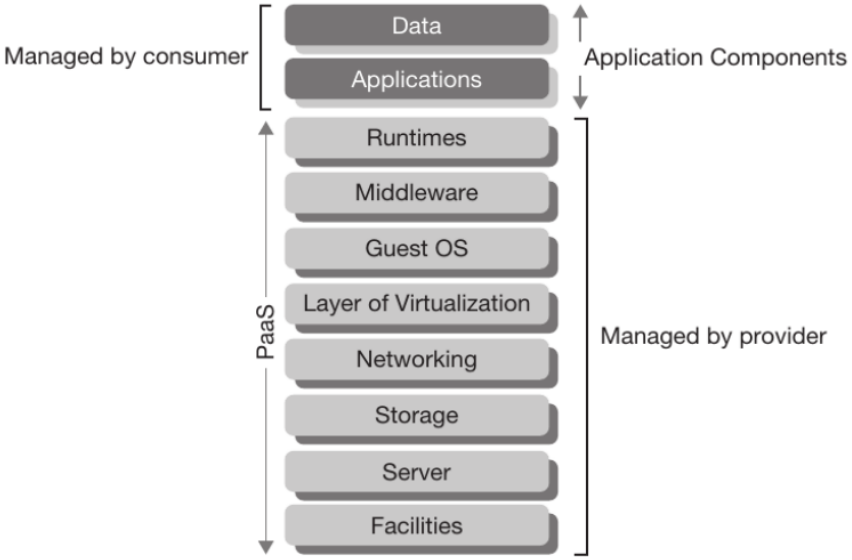


I vantaggi per i Cloud Consumers sono significativi, in particolare non vi sono più i costi per costruire e gestire un'infrastruttura fisica (ci pensa il Cloud Provider!). Esempio: Amazon Web Service.

1.4.5.3 Platform as a Service (PaaS)

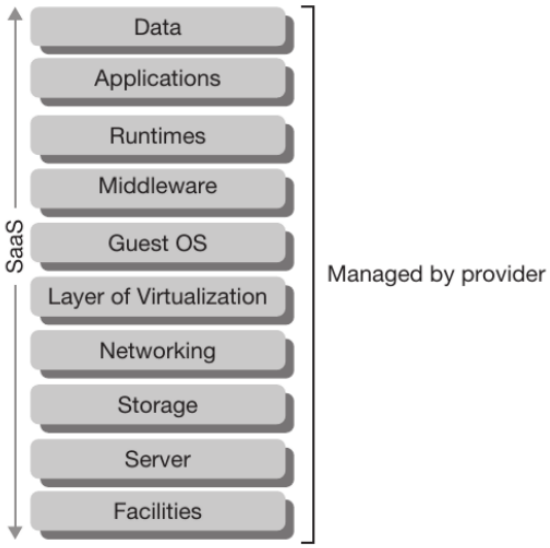
Livello maggiore di astrazione, non si ha accesso a una macchina virtuale ma ad una piattaforma! L'unica cosa che rimane in capo al Cloud Consumer sono la logica dell'applicazione e la gestione dei dati!

- Semplificazione per il Cloud Consumer, ma viene meno la flessibilità tipica del modello precedente: le applicazioni devono essere sviluppate utilizzando le API del sistema!
- Consegue che le applicazioni progettate in una certa piattaforma possano essere eseguite in un'altra piattaforma.
- Esempi: Google App Engine, Microsoft Azure Platform, Force.com.



1.4.5.4 Software as a Service (SaaS) – Massima astrazione

Massimo livello di astrazione, il Cloud Provider si occupa di tutti gli aspetti (anche il licensing) e fornisce un'applicazione già pronta e sviluppata al Cloud Consumer. Tipicamente l'applicazione è offerta all'utente per mezzo di un'interfaccia Web.



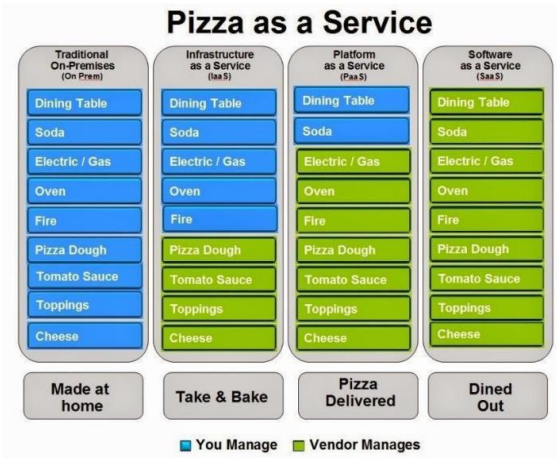
Nota: PaaS e SaaS possono essere implementate in cima a un'infrastruttura IaaS

- Creazione di una infrastruttura PaaS a partire da un'infrastruttura IaaS: creazione di macchine virtuali e installazione di una software platform su di esse.
- Creazione di un sistema SaaS a partire da un'infrastruttura PaaS: la piattaforma PaaS è usata per creare un software che viene offerto come servizio (SaaS).

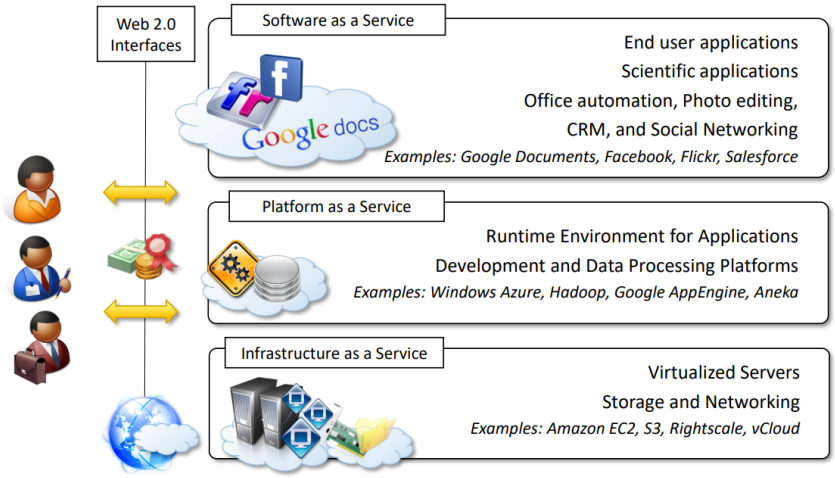
1.4.5.5 Esempi

Per comprendere meglio i concetti visti prendiamo l'esempio di una pizzeria!

- **Infrastructure as a Service**: IaaS provides virtual machines, virtual storage, virtual infrastructure, and other hardware assets as resources that clients can provision
- **Platform as a Service**: PaaS provides virtual machines, operating systems, applications, services, development frameworks, transactions, and control structures
- **Software as a Service**: SaaS is a complete operating environment with applications, management, and the user interface



Seguono alcuni esempi



1.4.5.6 Ulteriori modelli di Cloud Services

Menzioniamo ulteriori esempi di tipologie.

- ***Storage as a Service***.
  Paghiamo per aver spazio dove caricare files (IaaS).
- ***Database as a Service***.
  Paghiamo per istanziare un database e quindi connettere l'applicazione allo stesso. Ci interessa solo la logica dell'applicazione (PaaS)!
- ***Backup as a Service***.
  Paghiamo affinchè vengano svolti backup automatici dei nostri dati.
- ***Desktop as a Service***.
  Paghiamo non solo per avere una macchina virtuale, ma anche per avere un ambiente desktop a cui possiamo accedere. Si parla di *virtual desktop* o *hosted desktop*.

- **[Non presente nelle diapositive, detto a voce]** ***functions as a service***
  Ulteriore modello è *functions as a service*, un'evoluzione di *infrastructure as a service*: l'idea è che il Cloud Provider fornisce un ambiente simile a Platform as a Service, dove gli sviluppatori scrivono funzioni e queste vengono eseguite su un'infrastruttura remota (e lo sviluppatore non deve gestire tutto). La differenza rispetto a *Platform as a Service* è che queste funzioni non sono continuamente eseguite: sono invocate quando triggerate da eventi. Molto utile nell'Internet of Things!

2 [B] VIRTUALIZATION TECHNOLOGIES

2.1 Introduction to virtualization and basic concepts

Abbiamo capito dalle spiegazioni precedenti che la virtualizzazione è la *killer application*. Cerchiamo di capire meglio in cosa consiste la virtualizzazione.

2.1.1 Recap introduttivo

Abbiamo già detto che la virtualizzazione consiste nella rappresentazione di risorse computazionali fisiche in una forma simulata, per mezzo dell'aggiunta di uon strato di software (*abstraction layer*, "*this software, referred as virtualization layer, is installed over the physical machine to provide a virtual form of the hardware*"). Otteniamo risorse virtuali che:

- appaiono e si comportano come quelle reali;
- possono essere usate per rispondere alle necessità degli utenti.

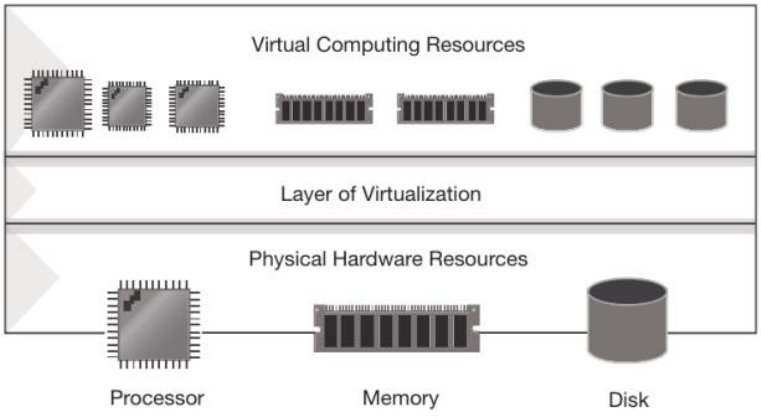
La novità significativa è la creazione di un ambiente completamente diverso da quello classico:

- nell'ambiente classico le applicazioni dialogano direttamente con le risorse fisiche, ergo devono essere consapevoli dell'hardware
- nell'ambiente creato con la virtualizzazione questo non è più necessario!

2.1.2 *Resource virtualization*

Si parla di virtualizzazione di risorse: quali sono le risorse che effettivamente possiamo virtualizzare? Processore, memoria RAM, disco rigido, periferiche (tastiera, mouse, stampante, rete, etc.)

- La *resource virtualization* può essere immaginata come una composizione di tecniche di virtualizzazione, ciascuna dedicata a virtualizzare una specifica risorsa (da quelle obbligatorie come la CPU a quelle opzionali come una periferica).



- **Aspetto importante sui *basic computing devices***.
  Nel caso di "basic computing devices" (CPU, RAM, disco rigido) la virtualizzazione è possibile soltanto quando sono disponibili le corrispondenti risorse fisiche. Non è possibile creare virtualizzazione di CPU, RAM e disco rigido senza avere CPU, RAM e disco rigido reali ("*can function only when a physical resource empowers the virtual representation*").

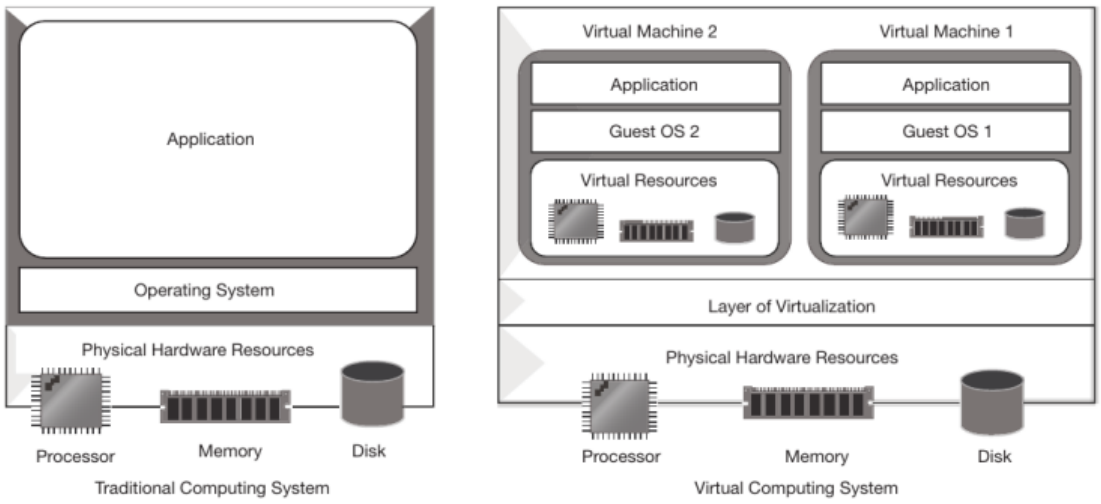
- **E per quanto riguarda la virtualizzazione delle periferiche?**
  Le periferiche non necessitano di risorse fisiche per essere virtualizzate: posso simulare una rete (*Virtual Network Adapter*) senza necessità di una connessione vera e propria; posso simulare una tastiera senza necessità di una tastiera reale…

- **Trasformazione delle risorse fisiche: uguaglianza con risorse virtuali non necessaria**.
  Altro aspetto: il virtualization layer trasforma le risorse fisiche! La rappresentazione virtuale "*may or may not be equal to the actual physical component, in architecture, quantity or quality*".
  - Prendiamo un esempio dove abbiamo un processore a 64 bit: è possibile creare un processore virtuale a 32bit a partire da quello fisico a 64bit.
  - Possibile creare tre processori virtuali a partire da un processore fisico. Anche qua il processore fisico non coincide con la rappresentazione virtuale (soprattutto in termini di performance).

2.1.3 Server virtualization

La *server virtualization* consiste nella creazione di una macchina virtuale su una macchina fisica: l'ultima è detta *host system*, la macchina virtuale *guest system*. Si utilizzano le tecniche di virtualizzazione che vanno a costituire il *virtualization layer*.

- Viene meno la relazione 1:1 tra computer fisico e sistema operativo: abbiamo una molteplicità di sistemi operativi eseguiti sulla stessa macchina.
- Ciascun guest systems rimane indipendente dagli altri e accede alle risorse hardware dell'host system. Lo fa in maniera indiretta, considerata la presenza del *virtualization layer*.
- Le applicazioni sono eseguite "*on top of the virtualized environments*", nel *guest system*.



2.1.4 Hypervisor: recap e tipologie di implementazione (type 1 e type 2)

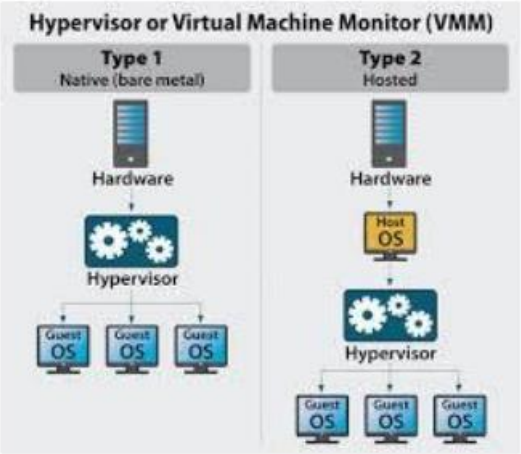
Ciò che rende possibile la virtualizzazione è il *virtualization layer*.

- La componente software del *virtualization layer* è detta *Hypervisor* (o *Virtual Machine Monitor*).
- Il virtualization layer, per mezzo dell'hypervisor, crea la rappresentazione virtuale delle risorse di base (CPU, RAM, disco, …) risorse che saranno utilizzate per creare le macchine virtuali.
- L'*hypervisor* contribuisce ad astrarre software e hardware ai livelli inferiori, coi vantaggi detti prima.

Esistono due approcci nell'implementazione dell'hypervisor

- ***Hosted approach*** (*Type 2*).
  Abbiamo un server col suo hardware: sul server è installato un sistema operativo (*host OS*). L'*hypervisor* è installato come software sul sistema operativo: implementa le tecniche di virtualizzazione e permette la creazione delle macchine virtuali. Sulle macchine virtuali possono essere eseguiti sistemi operativi (*guest OS*).
  - **Esempi**: VMWare e VirtualBox.

- **Bare Metal Approach** (*Type 1*).
  Abbiamo un server col suo hardware: sul server è installato direttamente l'hypervisor, senza sistemi operativi (host OS) nel mezzo. L'hypervisor in questo caso non è un semplice software, ma si comporta come un sistema operativo (devono essere implementate funzionalità che tipicamente sono implementate nel sistema operativo - memory management, CPU management, ….).
  - **Esempi**: VMWare ESX, Microsoft Hyper-V**.**



Quali vantaggi e svantaggi presentano queste due tipologie di implementazione?

- ***Hosted approach***.
  - **Vantaggi**: l'host OS gestisce le interazioni con l'hardware, semplificando drasticamente l'implementazione dell'hypervisor; maggiore compatibilità con diverse tipologie di hardware.
  - **Svantaggi**: non si ha accesso diretto alle risorse, questo degrada le prestazioni delle macchine virtuali.
- ***Bare Metal approach***.
  - **Vantaggi**: controllo pieno e diretto dell'hardware, l'overhead è minimizzato e le prestazioni sono migliori.
  - **Svantaggi**: l'hypervisor ha una limitata compatibilità hardware (supportato da un set di CPU, periferiche, architetture … - compatibilità favorita per componenti utilizzate nel "mercato" dei datacenters).

2.1.5 Tecniche di virtualizzazione basate su hypervisor

Le tecniche di virtualizzazione basate su hypervisor possono essere classificate in tre categorie:

- *full virtualization* (o *native virtualization*)
- *para-virtualization*
- *hardware-assisted virtualization*

Vedremo alcuni esempi di tecniche non basate su hypervisor (Es: *operating system virtualization*)

2.1.5.1 Full virtualization (o native virtualization)

Nella *full virtualization* l'hypervisor permette l'esecuzione di un sistema operativo come guest OS senza necessità di cambiamenti: questo ha la sensazione di essere eseguito per mezzo di risorse fisiche e non è consapevole della sua esecuzione in un ambiente virtuale.

- Esecuzione di sistemi operativi non modificati, inclusi sistemi proprietari: Windows, Linux, …
- Nella full virtualization l'hypervisor è responsabile della creazione della rappresentazione virtuale delle risorse e gestisce le interazioni tra guest OS e hardware.
- Il guest OS è eseguito nella macchina virtuale ed è completamente isolato rispetto alle risorse fisiche (*underlying hardware*) grazie all'hypervisor. Nel caso di virtualization type 2 l'hypervisor gestisce direttamente l'interazione, nel caso di virtualization type 1 l'hypervisor gestisce il tutto indirettamente interloquendo con l'host OS.

2.1.5.2 Para-virtualization

Nella full virtualization tutte le attività legate alla virtualizzazione sono svolte dall'hypervisor (instruction translation, interazione con l'hardware …). Queste attività potrebbero incidere in negativo sulle prestazioni delle macchine virtuali: per questo la *para-virtualization* prevede che il *guest OS* svolga una porzione di queste attività.

- I sistemi operativi tradizionali non svolgono queste attività: questo significa che non sono compatibili con la para-virtualization. Necessarie delle modifiche (*porting*, interlocuzione con API dell'hypervisor…)
- I guest OS sono consapevoli della loro esecuzione in un ambiente virtuale. Non solo questo: il guest OS deve conoscere anche l'hypervisor con cui dovrà interloquire (le modifiche potrebbero essere necessarie per rendere compatibile il guest OS con l'hypervisor).
- **Vantaggi**.
    - Riduzione dell'overhead. Il Guest OS può comunicare direttamente con l'hypervisor senza necessità per l'hypervisor di creare rappresentazioni virtuali delle risorse fisiche (no translations, no emulations …).
    - Nella full virtualization il virtualization layer offre device driver, nella para-virtualization i device drivers sono questione del guest OS. Semplificazione nell'implementazione dell'hypervisor.
- **Svantaggi**.
    - Versioni non modificate dei sistemi operativi non sono compatibili con la para-virtualization, come già anticipato. Facile fare modifiche con sistemi operativi open-source, decisamente più problematico con sistemi proprietari (dipende dal proprietario).
    - Minore sicurezza: il guest OS non è completamente separato dall'*underlying hardware* (a differenza del guest OS nella full virtualization).

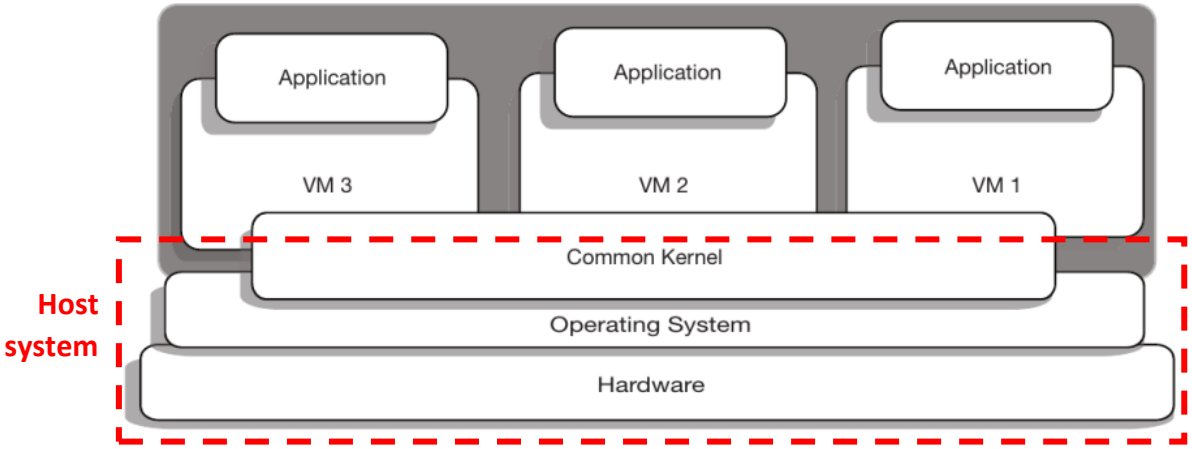
2.1.5.3 Hardware assisted virtualization

Adottata con le componenti hardware più recenti: particolare tipologia di *full virtualization* che sfrutta il supporto nativo della virtualizzazione da parte dell'hardware.

- Intel e AMD includono il supporto alla virtualizzazione nei loro processori dal 2006.
- I framework più noti sono *AMD-Virtualization* (AMD-V) e *Intel Virtualization Technology* (Intel-VT) e permettono:
    - minimizzazione dell'overhead dovuto all'esecuzione del guest OS sulla macchina virtuale;
    - esecuzione di particolari istruzioni privilegiate da parte del guest OS, istruzioni che saranno gestite direttamente dalla CPU senza passare dall'hypervisor (niente *binary translation* o *para-virtualization*).
- Possibile solo con specifiche combinazioni di componenti hardware.

2.1.6 Operating system level virtualization (no hypervisor)

La virtualizzazione a livello di sistema operativo (operating system level virtualization) funziona in maniera diversa rispetto alle precedenti tecniche di virtualizzazioni introdotte.



- Non è previsto un hypervisor, il ruolo dell'hypervisor è assunto dal kernel del sistema operativo installato sulla macchina fisica. Questo permette una drastica minimizzazione dell'overhead.
- Sulla macchina fisica è installato un sistema operativo (host OS), che è caratterizzato da un kernel. Tale kernel è modificato per supportare l'esecuzione di molteplici macchine virtuali "*on top of the same kernel*". Il kernel è condiviso tra tutte le macchine virtuali (molteplici istanze di server virtuali su una singola istanza di un OS kernel)!

- **Vantaggi**.
  Very light: molteplici server virtuali con un solo kernel. Il minimo overhead permette a un singolo sistema fisico di supportare un numero di server virtuali significativamente più alto rispetto al numero di server virtuali supportati con la full virtualization.
- **Svantaggi**.
  Tutte le macchine virtuali devono utilizzare lo stesso sistema operativo, dato che il kernel è condiviso. Possibili problemi di sicurezza: in caso di bug il kernel potrebbe consentire a una macchina virtuale non benevola accessi a porzioni di memoria di altre macchine virtuali.

2.1.7 Non solo server virtualization: network and storage virtualizations

Non siamo solo interessati alla virtualizzazione di una macchina o di un server.

- ***Network Virtualization***.
  Insieme delle tecniche che possono essere sfruttate per creare una rete virtuale (*virtual network*). Questa rete permette la comunicazione tra le macchine virtuali. La rete virtuale può sfruttare l'infrastruttura di rete reale per la trasmissione di dati (non obbligatorio perché questa rete può essere creata interamente in software).

- **Storage Virtualization**.
  Nel *traditional computing system* memoria significa collegamento diretto alla macchina locale dove il sistema/applicazione è eseguita (e.g. local hard drives). In un sistema virtualizzato anche lo *storage* è virtualizzato, in modo da supportare lo storage delle macchine virtuali. I dati memorizzati in storage virtuali saranno memorizzati in una qualche memoria fisica.

2.1.8 Emulazione: cosa si intende e che differenza c'è con la virtualizzazione

Introduciamo il concetto di emulazione: intendiamo *the act of making a system imitating another one*, attraverso il quale l'architettura può supportare un set di istruzioni relativo a un'altra macchina. Questo concetto nasce prima della virtualizzazione per permettere la simulazione in software di una particolare architettura.

- **Perché?**
  Si vuole eseguire un sistema operativo/programma scritto e compilato per una specifica architettura in un'altra architettura!
  - All'epoca l'incompatibilità di un sistema operativo/applicazione rispetto alle molteplici architettura esistenti era un problema serio.
  - Con l'adozione su larga scala dell'architettura Intel il problema si risolve.
  - Il problema sta riemergendo negli ultimi anni: dispositivi diversi da pc con architetture diverse da Intel (smartwatch, smartphone…), volontà di alcuni rivenditori di PC di abbandonare l'architettura di Intel (esempio: Apple, passaggio ad ARM).
- Il software di emulazione converte *binary data* per l'esecuzione su una macchina in una equivalente *binary form* per esecuzione su un'altra macchina.
- L'emulazione può essere implementata in due modi:
  - ***Binary translation***
    Il codice viene ricompilato e convertito nella sua interezza in una forma compatibile con l'architettura dove si vuole compiere l'emulazione.
  - ***Interpretation***
    Non si ha una ricompilazione dell'intero codice: si esegue il codice originario, l'emulatore

interpreta le istruzioni una ad una dopo le relative operazioni di lettura. Più facile da implementare, ma maggiore lentezza nell'esecuzione del codice.

Questa cosa non sembra fin troppo diversa dalla virtualizzazione, ma c'è una differenza sostanziale:

- ***Regular emulation***
  L'emulazione consiste nella simulazione in software di un'architettura completamente diversa, basata su un set di istruzioni diverso;
- ***Regular virtualization***
  Una delle assunzioni fatte è che l'instruction set usato nel sistema virtuale e l'instruction set fornito dall'hardware fisico è lo stesso! Questo significa che il software è in grado di funzionare autonomamente per un tempo significativo – tempo durante il quale non devono avvenire interpretazioni/traduzioni).

Si può immaginare la virtualizzazione, in un certo senso, come *una versione semplificata dell'emulazione*: qualcosa che è significativamente più veloce dell'emulazione e vicina al tempo di esecuzione del sistema operativo (se non eseguito in un ambiente virtuale).

emulation-based virtualization. L'emulazione permette la creazione dell'hardware in software, ergo permette la creazione di un ambiente virtuale dove un Guest OS viene eseguito *on top of a host* con un'architettura diversa (Esempio personale: esecuzione di un sistema operativo a 32 bit, creo a partire da una CPU a 64 bit una rappresentazione virtuale a 32 bit). Si parla di *emulation-based virtualization*.

Perché parliamo di emulazione? Perché in alcuni casi la virtualizzazione potrebbe richiedere forme di emulazione / traduzione. Lo vedremo nelle spiegazioni future.

2.1.9 Virtualization advantages and downsizes

Concludiamo con vantaggi e svantaggi della virtualizzazione.

- **Vantaggi**
  - ***Server consolidation***.
    Processo attraverso cui eseguiamo una molteplicità di macchine virtuali sullo stesso hardware. La cosa ci piace perché si migliora l'utilizzazione delle risorse hardware (e si riduce gli sprechi) rispetto agli approcci tradizionali.
  - ***Reducing hardware costs***.
    Se utilizziamo meglio le risorse hardware esistenti allora siamo in grado di ridurne i costi.
  - **Simplifying the system administration**.
    La separazione di hardware e software con il virtualization layer favorisce la semplificazione dell'amministrazione del sistema operativo. Si distingue nettamente la gestione del guest OS dalla gestione dell'host system.
    - **Simplifies system installation**.
      Installazione semplificata: possibile clonando al volo un'altra macchina virtuale.
    - ***Zero downtime maintenance***.
      Le macchine virtuali possono essere spostate da un server fisico a un altro con downtime minimo o addirittura nullo. Supponiamo che sia necessario fare la manutenzione di un particolare server fisico: possiamo spostare la macchina virtuale su un altro server fisico funzionante e lavorare sul primo senza provocare la sospensione del servizio.
    - ***Fault tolerance*.**
      Possibile creazione periodica di backup delle macchine virtuali.
  - ***Improves security***.
    Ogni macchina virtuale è isolata dalle altre grazie al virtualization layer.
- **Svantaggi**
  - ***Single point of failure***.
    Ogni macchina fisica è un single point of failure: è vero che eseguire molteplici macchine

virtuali su una singola macchina fisica permette la server consolidation, ma è anche vero che se la macchina fisica ha problemi questo si ripercuote su tutte le macchine virtuali.

- ***Lower performances***.
  Riduzione della performance nelle macchine virtuali. È stato dimostrato che un server virtuale può raggiungere l'85%/90% delle performance di un server fisico. Ricordiamo che la macchina virtuale non accede direttamente all'hardware.

2.2 Multiprogramming recap

2.2.1 Requisiti della virtualizzazione e legame con la multiprogrammazione

La virtualizzazione pone dei requisiti:

- ***Equivalence***, un sistema operativo eseguito su una macchina virtuale sotto l'hypervisor dovrebbe esibire un comportamento nei fatti identico a quello mostrato dallo stesso sistema operativo quando eseguito su una macchina fisica equivalente.
- ***Resource control***
    - L'hypervisor deve avere il pieno controllo delle risorse fisiche
    - Il sistema operativo eseguito sulla macchina virtuale deve avere il pieno controllo delle risorse virtualizzate.
- ***Efficiency***, una frazione rilevante delle istruzioni eseguite sulla macchina virtuale deve avvenire senza l'intervento dell'hypervisor, in modo da minimizzare l'overhead.

I requisiti appena introdotti sono gli stessi della multiprogrammazione, concetto già affrontato nel corso di Calcolatori elettronici (triennale in Ingegneria informatica di Pisa).

2.2.2 Multiprogrammazione

2.2.2.1 Concetto

"We emulate a machine with more processors (and other peripherals than we have in the reale hardware"

La multiprogrammazione permette, in sostanza, la creazione di un ambiente dove più applicazioni "sono eseguite in parallelo" nonostante la presenza di un unico processore!

- Ogni processo ha il suo processore virtuale (virtual processor, VCPU). Ogni processore virtuale è riservato a un particolare processo.
- Si ha un'operazione di multiplexing dove i vari processori virtuali possono eseguire le istruzioni dei processi attraverso il processore fisico.
- Caratteristica fondamentale: il processo deve avere l'impressione di un pieno controllo del processore fisico, al punto tale da credere di essere l'unico processo eseguito su di esso.

2.2.2.2 Implementazione

La multiprogrammazione è implementata dal sistema operativo per mezzo di strutture dati. Ciascuna di esse contiene al suo interno lo stato di un processore virtuale (i suoi registri).

- Una macchina con un solo processore (in generale è cosi) emula un processore virtuale alla volta):
    - per prima cosa lo stato del processore virtuale, memorizzato nella relativa struttura dati, viene caricato sul processore reale (valori dei registri);
    - a questo punto le istruzioni del processo assegnato al processore virtuale vengono eseguite;
    - a un certo punto l'esecuzione va in pausa (può succedere per svariate ragioni) e la struttura dati contenente lo stato del processore virtuale viene aggiornata con gli attuali valori dei registri;
    - viene scelto un nuovo processore virtuale da eseguire e si ritorna al primo punto

  Con un'esecuzione frequente delle CPU virtuali diamo ai processi l'impressione che questi siano eseguiti "continuously" (caratteristica fondamentale rispettata)

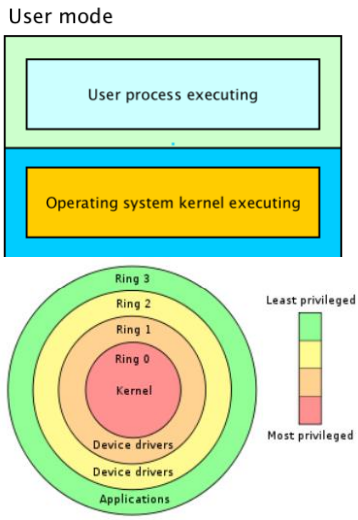
- Il passaggio da un processo a un altro (che per quanto detto prima equivale al passaggio da una VCPU a un'altra) è detto *context switch*. È implementato all'interno del sistema operativo ed è determinato da un timer (lo scheduler) o un'interruzione (che viene lanciata quando l'input di un hardware diventa disponibile per dei processi). *"which implements the operations of loading/unloading the host processor registers from/to memory"*.

- La multiprogrammazione pone ulteriori requisiti lato hardware:

- meccanismo delle interruzioni (esempio un timer periodico per triggerare un periodico context-switch);
- un meccanismo di protezione che impedisca a processi non autorizzati di eseguire specifiche istruzioni (dette istruzioni privilegiate) e accedere a specifici settori della memoria RAM (si vuole impedire la manipolazione delle strutture dati alla base della multiprogrammazione da parte di processi non autorizzati);
- un meccanismo automatizzato che permetta il load/unload dello stato dei registri (questo non è necessario, ma fare una cosa per mezzo di un'unica istruzione è sicuramente più veloce del farlo con più istruzioni – è il CISC vs RISC).

- Il meccanismo di protezione prevede la presenza di tre ulteriori registri:
  - registro usr/sys, con flag che specifica se la CPU si trova in system level o user level;
  - registro sys-mem, che contiene lo starting address dell'area di memoria dove sono memorizzati gli stati dei VCPU;
  - registro ret, che contiene l'indirizzo di memoria su cui porsi al termine dell'esecuzione in system level.

- Per l'attuazione dei requisiti della multiprogrammazione si introducono livelli di privilegio. Tipicamente si hanno due livelli di privilegio: system level e user level.
  - Processi eseguiti in system level possono eseguire tutte le istruzioni, incluse quelle privilegiate, e avere accesso all'intera memoria RAM. Processi eseguiti in user level non possono eseguire istruzioni privilegiate e avere accesso a specifiche sezioni della memoria RAM.
  - I processi relativi alle applicazioni eseguite dagli utenti sono lanciati con livello di privilegio user level.
  - Le routine legate alla gestione del context switch sono eseguite con livello di privilegio system level.

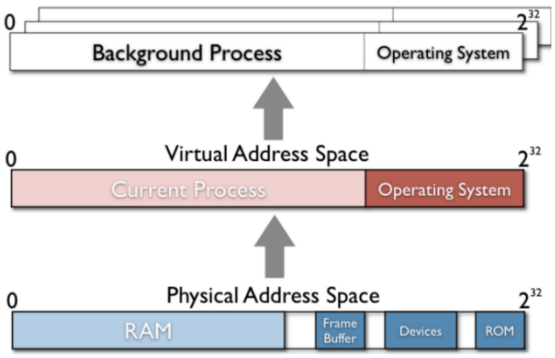


Prendiamo come esempio il context switch triggerato dal timer, cosa fa il sistema operativo?
  - la routine viene eseguita in system level;
  - lo stato della CPU dell'host viene posto nell'apposita struttura dati (VCPU);
  - la routine seleziona la nuova la nuova VCPU da eseguire e carica il suo stato sulla CPU dell'host;
  - la routine esegue una istruzione speciale che fa JUMP sull'indirizzo memorizzato nel ret register e ripristina il livello di privilegio user level.

Le cose dette nel secondo e nel terzo punto sono possibili in CPU recenti per mezzo di un set di istruzioni implementate direttamente in hardware.

2.2.3 Memoria virtuale

2.2.3.1 Concetto

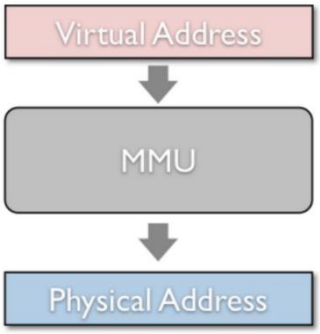


Si introduce il concetto di memoria virtuale (virtual memory). Non abbiamo più solo una memoria reale con indirizzi fisici, abbiamo un ulteriore spazio di indirizzi virtuali.

- Ogni processo ha un suo Virtual Address Space. Gli indirizzi posti nelle istruzioni delle applicazioni sono indirizzi virtuali.
- Si stabiliscono corrispondenze tra gli indirizzi fisici e gli indirizzi virtuali.
- Il processo ha l'impressione di un "contiguous address space", ma ciò non è vero: i dati possono essere memorizzati nella RAM in maniera non contigua.
- Il processo ha l'impressione di avere accesso all'intera memoria fisica.

2.2.3.2 Implementazione

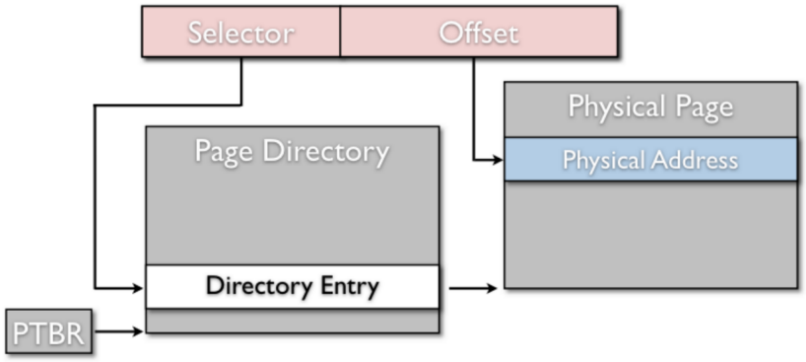
L'implementazione della memoria virtuale avviene introducendo due meccanismi:



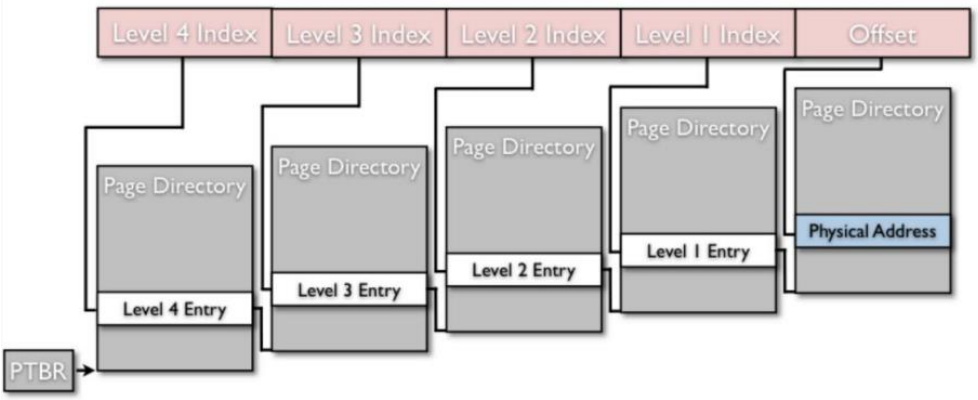
- ***Virtual address spaces management***, viene attuata dal sistema operativo che gestisce i Virtual Address Space creati per ogni applicazione;
- ***Address translation***, traduzione degli indirizzi virtuali in indirizzi fisici, attuata per mezzo di una componente hardware chiamata Memory Management Unit (MMU).

La MMU gestisce la traduzione degli indirizzi ricorrendo alla Page Table.

- La Page Table è una tabella memorizzata in memoria che contiene le corrispondenze tra indirizzi virtuali e indirizzi fisici. È gestita dal sistema operativo.
- La CPU possiede un Page Table Base Register (PTBR) che ha come valore l'indirizzo fisico del primo byte della Page table.
- La memoria fisica è divisa in pagine di uguali dimensioni (4KB ciascuna).
- La Page Table è organizzata in Page directories, strutture dati che contengono le informazioni utili alla traduzione di un subset degli indirizzi virtuali nei corrispondenti indirizzi fisici.



- Tipicamente le Page Tables sono organizzate in più livelli: è necessario accedere a più page directories per poter tradurre l'intero indirizzo virtuale in un indirizzo fisico. Su questa organizzazione a livelli incide moltissimo l'architettura hardware e il sistema operativo.



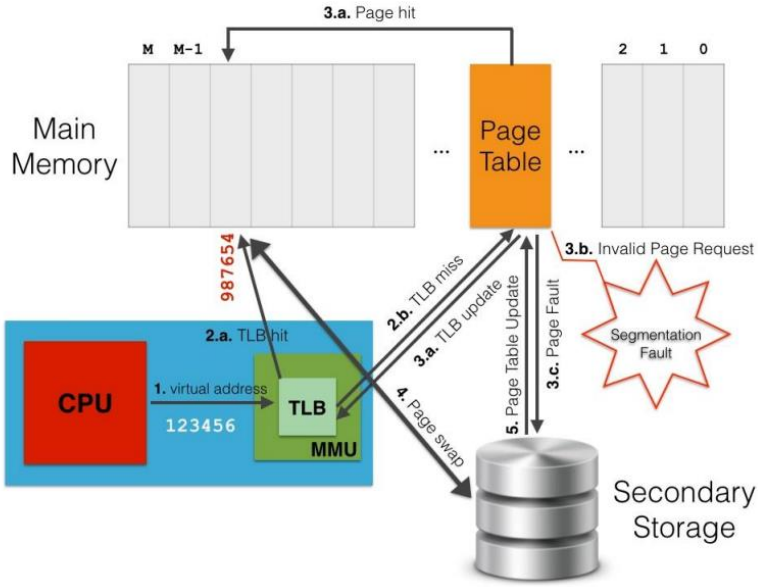
- Tipicamente si velocizzano le operazioni della MMU sfruttando una memoria cache: il Translation Lookaside Buffer (TLB).

Recap differenza MMU e Page Table ad opera di ChatGPT
La **Memory Management Unit (MMU)** e la **Page Table** sono entrambi componenti cruciali per la gestione della memoria virtuale nei sistemi operativi, ma svolgono ruoli differenti: 1. **Memory Management Unit (MMU)**: È un componente hardware del processore che gestisce le traduzioni degli indirizzi virtuali in indirizzi fisici. La MMU utilizza la Page Table per effettuare questa traduzione in tempo reale durante l'esecuzione di un programma. 2. **Page Table**: È una struttura dati utilizzata dal sistema operativo per mappare gli indirizzi virtuali a quelli fisici. La tabella delle pagine è memorizzata in memoria e contiene le informazioni necessarie per la MMU per tradurre gli indirizzi. In sintesi, la **MMU** esegue la traduzione degli indirizzi utilizzando la **Page Table** come riferimento per le traduzioni.

2.2.3.3 Segmentation fault e page fault

È possibile espandere il *Virtual Address Space* oltre la dimensione della RAM utilizzando memorie secondarie (ad esempio un hard disk).

- Le istruzioni eseguite sono prese sempre dalla RAM, ergo ogni istruzione da eseguire deve essere spostata nella RAM se non presente.
- La memoria fisica è divisa in pagine, ai sensi di quanto detto sul Page Table. La RAM ospita al suo interno pagine, se queste non sono utilizzate (nel senso che le sue istruzioni non vengono eseguite) non è necessario tenerle in memoria RAM.
- Ogni volta che un programma tenta di accedere a una pagina memorizzata in memoria secondaria viene invocato un page fault: compito del sistema operativo è recuperare la pagina e porla in memoria RAM. Viene aggiornata, conseguentemente, la Page Table.



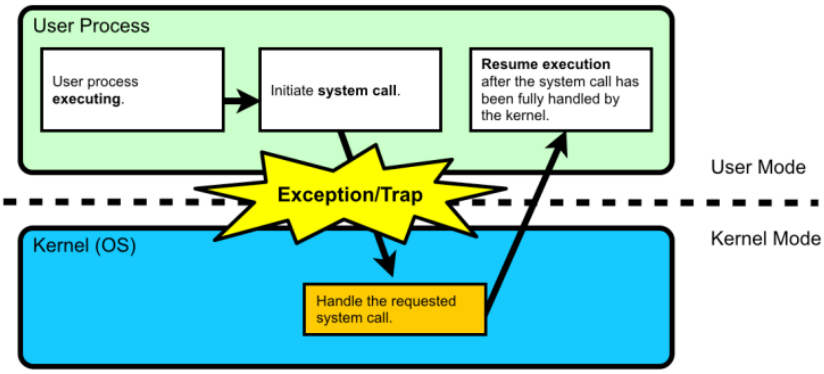
Riepiloghiamo i passaggi:

- la CPU pone un indirizzo virtuale;
- si cercano corrispondenze tra pagine nel TLB, se c'è corrispondenza abbiamo finito;
- se non c'è corrispondenza si accede al page table per individuare la corrispondenza, se c'è si aggiorna la TLB e abbiamo finito;
- se dalla lettura della Page Table si deduce che la richiesta non è valida (esempio, accesso ad area di memoria privilegiata) si invoca un *segmentation fault*;
- se dalla lettura della Page Table si deduce che la richiesta è valida, ma che la pagina non è presente in RAM allora si invoca il Page fault e si fa quanto detto prima (*page swap*, se la RAM è piena devo decidere quale pagina spostare dalla RAM alla memoria secondaria per liberare spazio).

2.2.4 Interruzioni ed eccezioni

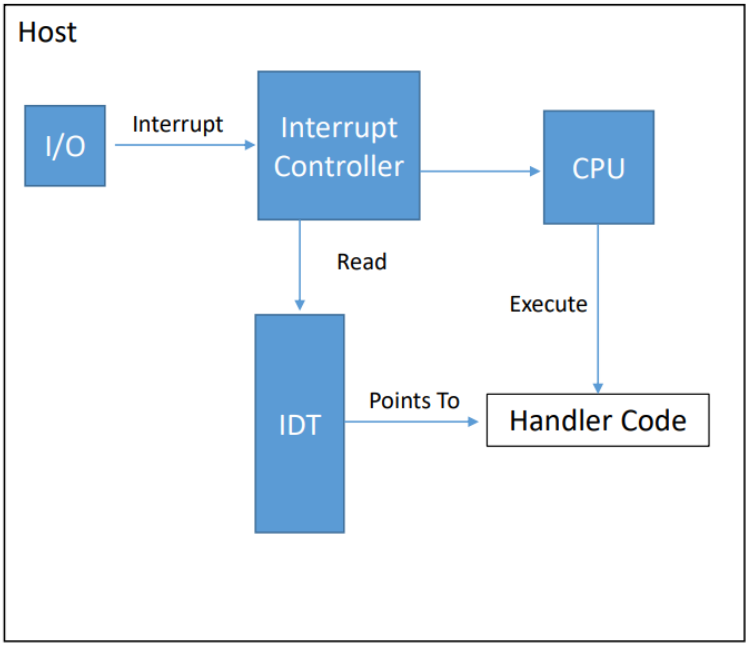
Durante l'esecuzione di un programma potrebbero emergere eventi che necessitano di attenzione immediata. I meccanismi che permettono di avvertire il sistema di ciò sono interruzioni ed eccezioni.

- Le eccezioni permettono di notificare il sistema di eventi interni, ad esempio errori del programma (divisione per zero, cattivo indirizzo, page fault…). Possono essere invocate anche per mezzo dell'istruzione INT. L'istruzione INT permette l'implementazione delle primitive di sistema, le API che possono essere utilizzate dagli utenti nei programmi.
- Le interruzioni sono evocate da componenti hardware esterne alla CPU e notificano il sistema di eventi esterni (timer, pressione di un tasto della tastiera…).
- Alle eccezioni e interruzioni sono associati handler, cioè funzioni eseguite a livello sistema a seguito dell'invocazione delle relative eccezioni/interruzioni. Al termine dell'esecuzione dell'handler si ritorna a livello utente e viene ripreso quanto lasciato in sospeso.



Le operazioni descritte sono gestite per mezzo di una Interrupt Descriptor Table (IDV), una tabella che permette di associare gli handler (posti in memoria) a specifiche interruzioni/eccezioni. “The table is populated by the operating system at bootstrap”. Ricapitoliamo prendendo ad esempio un’interruzione:

- Una periferica invoca un’interruzione notificando ciò a un apposito controllore delle interruzioni;
- Il controllore legge nella IDT l’indirizzo dove recuperare l’handler da eseguite;
- Il controllore avverte la CPU, che inizia ad eseguire l’handler a livello sistema.



2.3 Full virtualization Techniques – trap and emulate approach

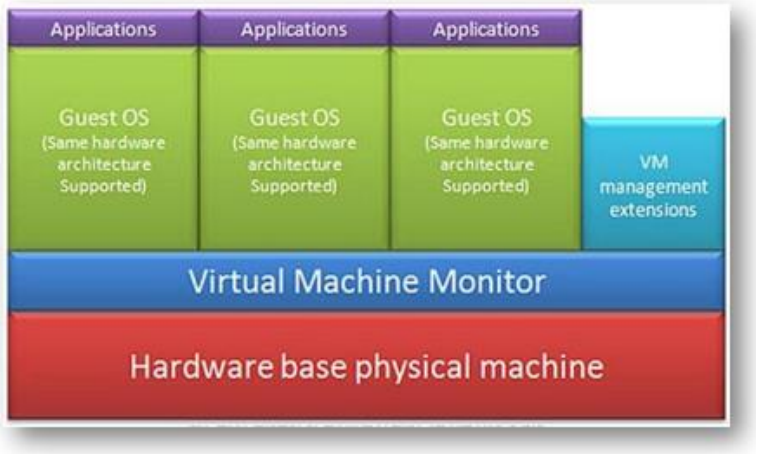
2.3.1 Introduzione: multiprogrammazione e hypervisor

La multiprogrammazione già include alcune tecniche di virtualizzazione che potrebbero essere utilizzate per implementare *System Level virtualization*.

- La multiprogrammazione è pensata in maniera tale che il processo abbia l'impressione di avere pieno controllo su CPU e RAM.
- Allo stesso modo l'*Hypervisor* è pensato in maniera tale che la macchina virtuale abbia l'impressione di avere pieno controllo delle risorse fisiche (processori, memoria, periferiche…), quando in realtà le risorse fisiche sono gestite dall'hypervisor stesso.

L'hypervisor ha un ruolo simile a quello dell'OS Kernel, ma c'è una differenza significativa: dentro la macchina virtuale non vogliamo eseguire un semplice programma, ma un sistema operativo (un sistema multiprogrammato – vuole accesso alle *page tables*, alla *Interrupt Descriptor Table*, …)! Dire "il sistema operativo deve avere l'impressione di avere pieno controllo" significa che questo deve avere tale impressione anche sui meccanismi di multiprogrammazione.

Si introduce un set di meccanismi attraverso cui l'hypervisor fornisce una rappresentazione virtuale del sistema operativo, rappresentazione che include l'accesso ai meccanismi di multiprogrammazione.



Se l'architettura della macchina virtuale e quella dell'host OS è la stessa allora la maggioranza delle funzionalità che ci si aspetta da una macchina virtuale sono già implementate in hardware!

- Si implementa un *context switch* tra esecuzioni di macchine virtuali (e non tra applicazioni).
- Quando la macchina virtuale viene caricata vogliamo che questa venga eseguita "*on top of the physical hardware*", cioè minimizzare emulazioni e in generale interventi dell'hypervisor (detta in altre parole: minimizzazione dell'overhead).

- **L'emulazione sarà presente in ogni caso**.
  È necessario introdurre dei limiti nell'esecuzione della macchina virtuale: se questa vuole eseguire istruzioni che non può eseguire (si pensi alle istruzioni privilegiate…) l'hypervisor deve assumere il controllo del sistema ed emulare queste funzionalità.

2.3.2 Primo step: virtualizing CPU

Il primo step è la virtualizzazione della CPU. Le tecniche adottate sono quelle della multiprogrammazione.

- **Idea di base**.
  La VMM deve emulare l'intero processore: le funzionalità a livello utente e anche quelle a livello sistema (supporto alla multiprogrammazione).

- Il guest OS è eseguito interamente in *user space*. L'hypervisor è eseguito in *system/kernel space*.

- **Come è possibile eseguire un intero guest OS in user space? Non ci sono le istruzioni privilegiate?**
  Senza virtualizzazione l'esecuzione di un'istruzione privilegiata a livello utente è un *security threat*
  - Ogni volta che una macchina virtuale è posta in esecuzione lo stato della macchina virtuale è caricato nell'host processor (caricamento dei valori dei registri). Fatto questo il codice viene eseguito sul processore fino a quando non si individuerà qualcosa che non può essere eseguito direttamente dalla macchina virtuale.
  - Viene triggerato un *context switch* dalla macchina virtuale all'hypervisor ogni volta che un'istruzione privilegiata viene eseguita all'interno della macchina virtuale. L'hypervisor code viene eseguito: deve emulare l'istruzione che ha triggerato il context switch, istruzione che non può essere eseguita in user/space. Fatto questo l'hypervisor restituisce il controllo alla macchina virtuale e l'esecuzione del flusso di istruzioni ripresa.

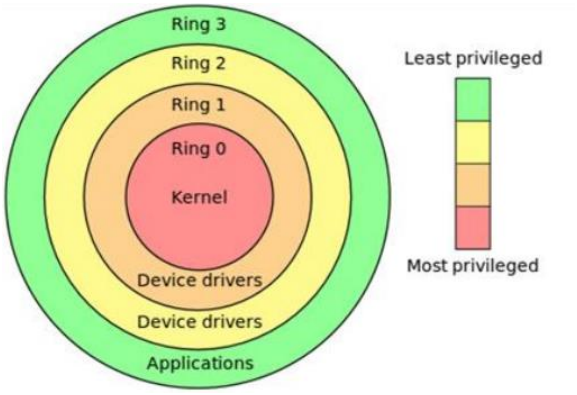
  Quanto detto ha un costo significativo (*wasted time*, ma noi vogliamo minimizzare il più possibile l'intervento dell'hypervisor).

- **Presenza di molteplici macchine virtuali?**
  In presenza di molteplici macchine virtuali l'hypervisor è congeniato in maniera che un interrupt venga lanciato periodicamente (timer) per eseguire un *virtual machine scheduler*.

L'approccio *trap and emulate* si articola in due passaggi:

- ***trap***
  Ogni volta che viene lanciata un'istruzione privilegiata l'hardware solleva un'eccezione (*trap*).
- ***emulate***
  L'hypervisor recupera il controllo del processore ed emula l'esecuzione dell'istruzione privilegiata eseguendo il *trap code*, cioè l'handler associato all'eccezione. Per emulare si usa *binary translation*.

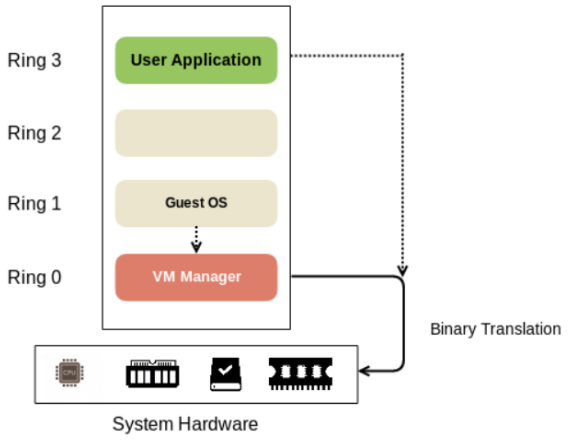
Al termine dell'emulazione il controllo viene restituito al *guest OS code*. Per rendere chiara la questione ricordiamo che il processore può essere eseguito in una tra molteplici modalità, in generale quattro:



- ***Ring 0***. Equivalente della *kernel/system mode* (*core functionalities of the operating system*).
- ***Ring 1 e Ring 2***.
  Situazione intermedia tra kernel/system mode e user mode, livelli dove si eseguono i device drivers. L'idea non è casuale: si preferisce dare un accesso privilegiato non completo in quanto i device drivers sono prodotti da soggetti terzi.
- ***Ring 3***. Equivalente della *user mode*.

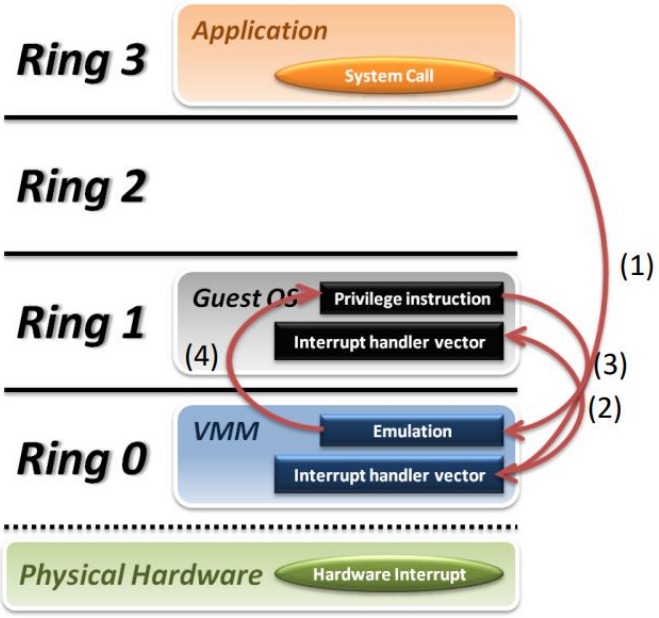
L'approccio *trap and emulate* prevede:

- esecuzione dell'hypervisor in Ring 0;
- esecuzione del Guest OS in Ring 1;
- esecuzione dell'applicazione sviluppata dall'utente in Ring 3.



Il Guest OS potrebbe eseguire istruzioni privilegiate che non sono parte del set di istruzioni fornite al device driver. In quel caso l'eccezione è sollevata e provoca l'esecuzione dell'hypervisor. Esempi: I/O instructions, accesso alle Page Tables…

Cosa succede quando un'applicazione eseguita in Ring 3 invoca una System Call?

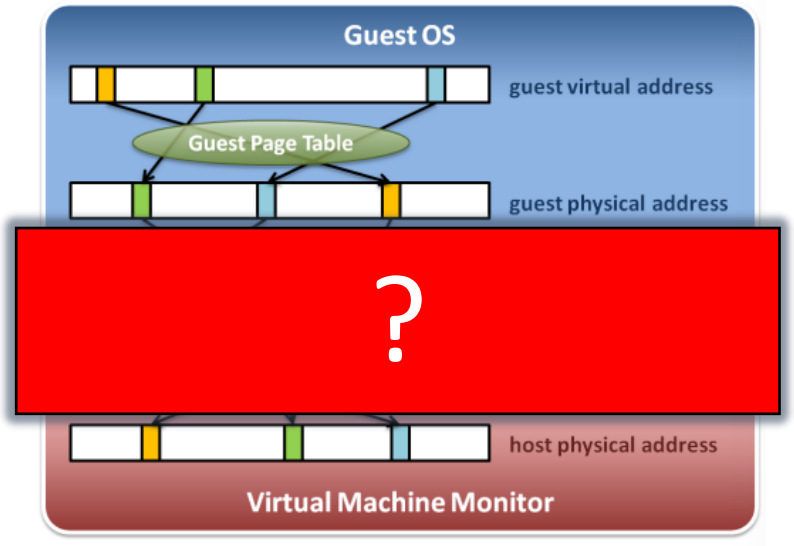


- Esecuzione di una *system call* significa lancio dell'istruzione
  `INT system_call_identifier`
  L'istruzione INT implica l'esecuzione di un handler/function implementata nel sistema operativo. In un ambiente virtualizzato il processo si allunga.
- INT non può essere eseguita in Ring 3. Context switch da Ring 3 a Ring 0 (*hypervisor*) e invocazione di un handler con identificativo `system_call_identifier`. L'indirizzo della prima istruzione viene recuperato nell'*Interrupt Descriptor Table* **dell'hypervisor**.
- L'handler eseguito in Ring 0 permetterà l'esecuzione dell'handler vero e proprio: quello del Guest OS! L'handler vero e proprio è una funzione che viene recuperata nell'*Interrupt Descriptor Table* **del Guest OS**. Context switch da Ring 0 a Ring 1.
- Ogni volta che si incontra un'istruzione privilegiata durante l'esecuzione dell'handler vero e proprio si ha context switch da Ring 1 a Ring 0: l'esecuzione avviene in Ring 0 (emulata dall'hypervisor in modo sicuro). Al termine viene restituito il controllo al *Guest OS* (context switch da Ring 0 a Ring 1).
- Al termine dell'esecuzione della *system call* il controllo verrà restituito all'applicazione in Ring 3.

2.3.3 Secondo step: virtualizing Physical Memory

2.3.3.1 Introduzione

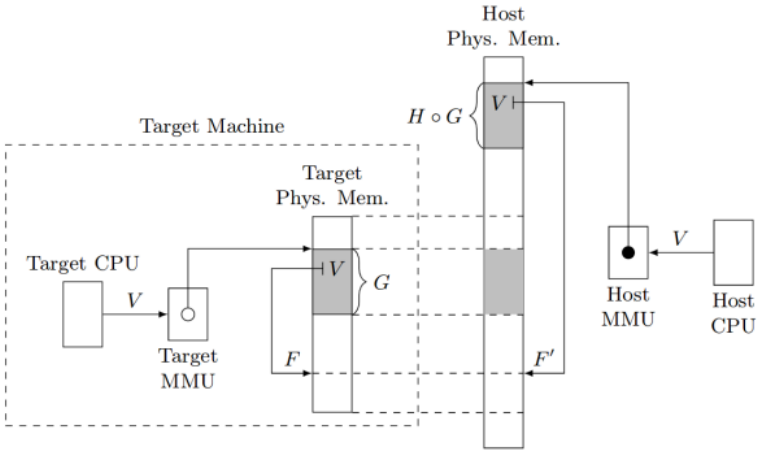
La memoria fisica di una Virtual Machine è implementata utilizzando una sezione della memoria fisica dell'host. Anche qua si fa ricorso ai meccanismi offerti dalla multiprogrammazione: la macchina virtuale deve avere l'impressione di avere controllo di tali meccanismi (memoria virtuale), ergo deve avere l'impressione di un pieno controllo sulla memoria fisica.



- **Pieno controllo dell'hypervisor**.
    - L'hypervisor è l'unico che ha pieno controllo dell'intera memoria fisica (RAM, indirizzi fisici…).
    - Ogni volta che viene creata una nuova macchina virtuale l'hypervisor riserva una porzione di memoria fisica a tale macchina virtuale.
    - L'*hypervisor* gestisce i *page fault* (recupero delle informazioni quando queste sono poste in memorie secondarie).
- L'hypervisor deve gestire il mapping tra la RAM vera e la RAM virtuale associata alla macchina virtuale. All'interno del Guest OS permane la distinzione tra indirizzi virtuali (guest virtual addresses, utilizzati dall'applicazione) e indirizzi fisici (*guest physical addresses*): la novità sta nel fatto che quelli che gli indirizzi fisici di un Guest OS sono a loro volta indirizzi virtuali!
- La traduzione dei *guest virtual addresses* in *guest physical addresses* era e rimane una competenza del Guest OS.
- È necessario definire dei meccanismi all'interno dell'hypervisor che permetteranno la traduzione dei *guest physical addresses* in *host physical addresses*.

2.3.3.2 Virtual MMU e brute force method

Implementiamo tale meccanismo creando una *Virtual MMU*

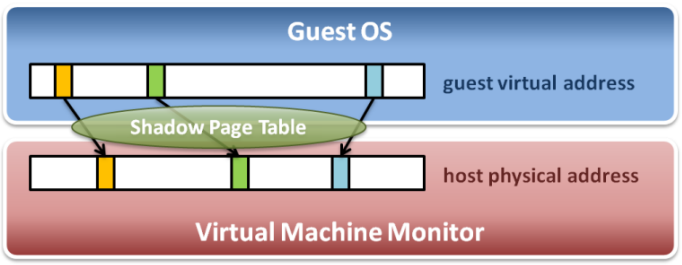


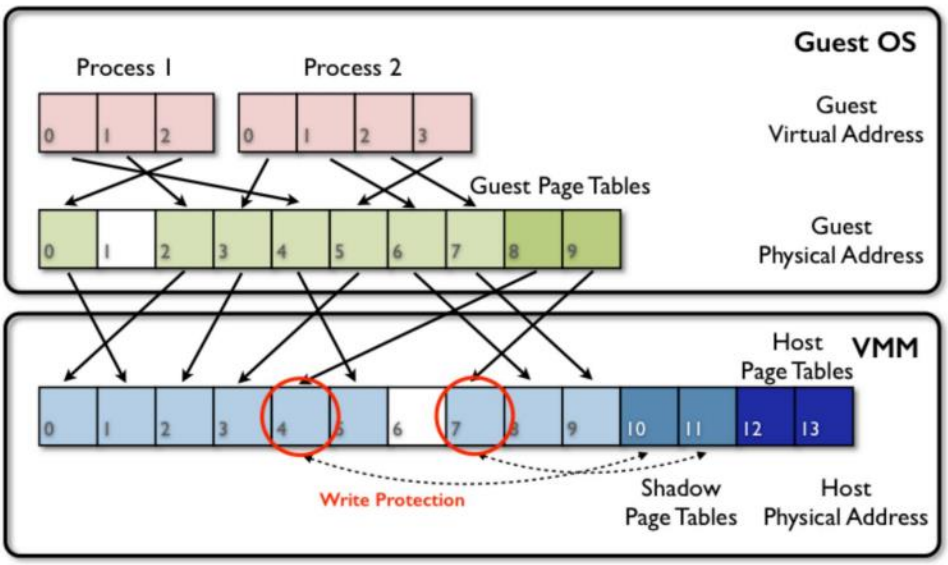
- Normalmente abbiamo una *MMU* (hardware) e una *Page Table* (software) gestita da un sistema operativo. È definita una funzione G che traduce gli indirizzi fisici in indirizzi virtuali.
- Nel caso della virtualizzazione la funzione G è gestita dal Guest OS per mezzo della sua *Page Table*, tuttavia ciò non è sufficiente in quanto il Guest OS non dialoga direttamente con l'hardware (detta in altre parole: l'indirizzo ottenuto non è ancora un indirizzo fisico).

Si vuole implementare una Virtual MMU che traduce gli indirizzi virtuali posti nel Guest OS in indirizzi fisici utilizzati a livello hypervisor per dialogare con l'hardware.

- La funzione definita è $H \circ G$, dove H è una funzione gestita dall'hypervisor.
- La funzione H permette la traduzione dei *guest physical addresses* in *host physical addresses*: questo significa che la funzione composizione $H \circ G$ permette la traduzione dei *guest virtual addresses* in *host physical addresses*!

L'implementazione della *Virtual MMU* avviene in maniera trasparente (dal punto di vista della VM). Per mezzo di un approccio forza bruta (*brute force*) l'hypervisor modifica la Page Table di una macchina virtuale con lo scopo di introdurre un ulteriore layer: la *Shadow Page Table*.





- Il kernel del Guest OS crea le *Page Tables*, permettendo la traduzione da *Guest Virtual Addresses* a *Guest Physical Addresses*.
- L'hypervisor modifica silenziosamente le Page Tables delle *Virtual Machines* introducendo ulteriori layers. Questi layers ulteriori permettono la traduzione da *Guest Physical Addresses* a *Host Physical Addresses*.
    - **Velocità nell'esecuzione dell'applicazione sul Guest OS**.
      La combinazione di queste Page Tables da luogo a Page Tables che traducono *Guest Virtual Addresses* in *Host Virtual Addresses*. Queste Page Tables sono lette runtime dalla MMU (zero overhead, non si hanno context switches).
    - **Passaggi quando il Guest OS modifica le Page Tables: overhead**!
      Ogni modifica alle Page Tables del Guest OS è intercettata dall'hypervisor (assume il controllo dato che le tabelle sono *write-protect*): analizza le modifiche applicate e modifica, di conseguenza, le *Shadow Page Tables*. I momenti precisi in cui avviene il context switch (con *trap and emulate approach*, ovviamente) sono i seguenti:
        - modifica delle entries di una page table;
        - cambio del valore posto nel registro PTBR, cosa che avviene quando il Guest OS vuole interamente sostituire la page table)

      Se si manifesta il secondo evento la tabella sarà analizzata per intero in modo da aggiornare la Shadow Page Table.

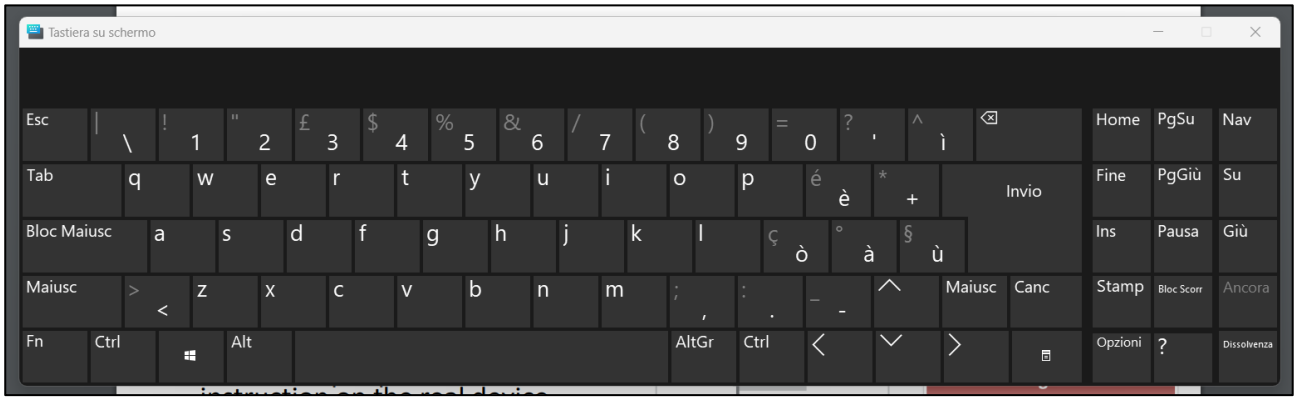
2.3.4 Terzo step: virtualizing I/O devices

2.3.4.1 Introduzione

Solitamente le istruzioni I/O sono istruzioni privilegiate, ergo non possono essere eseguite in *user space*.

- La macchina virtuale è eseguita su un server fisico caratterizzato da *physical devices*.
    - L'hypervisor è l'unico che può dialogare direttamente con tali *devices*.
    - L'hypervisor fornisce alle macchine virtuali una rappresentazione virtuale di questi *devices*. L'interazione col device virtuale può risultare in un'interazione col device fisico.
    - L'implementazione della rappresentazione virtuale (detta *device model*) avviene per mezzo di un set di strutture dati.
- Ogni volta che il *Guest OS* tenta un'interazione con un *device* si ha il lancio di una *system call*.
    - Con i passaggi tipici dell'approccio *trap and emulate* si ha l'emulazione dell'istruzione da parte dell'hypervisor.
    - Se l'emulazione del dispositivo virtuale richiede un'interazione col vero I/O device allora l'hypervisor gestirà le interazioni col dispositivo reale.

- Due possibili metodi per emulare l'I/O:
    - emulazione completa, rappresentazione virtuale del dispositivo senza interazione con un dispositivo fisico vero e proprio
        - *Virtual Printer*
          Quando il file viene convertito in PDF
        - *Virtual Keyboard*
          Un esempio di Virtual Keyboard può essere l'applicazione "Tastiera su schermo", ma la cosa può riguardare situazioni più complesse (Esempio: emulazione della tastiera leggendo un file che indica la combinazioni di tasti da premere, il punto è che la lettura da file permette di emulare l'azione di un utente che non è fisicamente a interagire con la tastiera fisica del server).

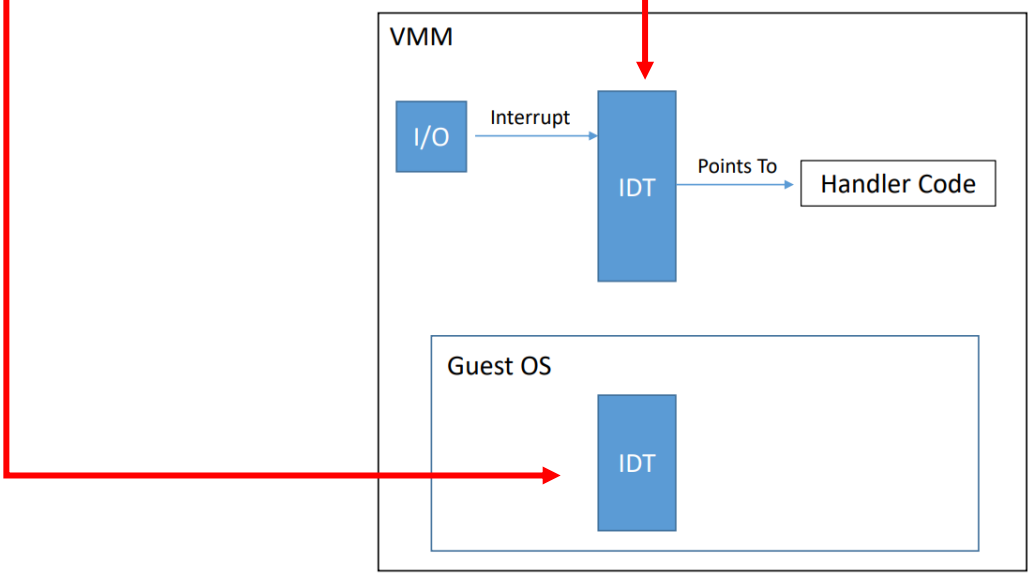


    - rappresentazione virtuale del dispositivo con possibile interazione col dispositivo fisico
        - *Scheda video e macchina virtuale su VirtualBox*
          Quando la macchina virtuale è in foreground e questa desidera proiettare qualcosa (ad esempio un video) viene inviata una richiesta alla rappresentazione virtuale (che nei fatti è una richiesta di emulazione della scheda video all'hypervisor). L'*hypervisor* inoltra la richiesta alla scheda video vera e propria.

2.3.4.2 Interrupt management

Come gestiamo le interruzioni provenienti dai dispositivi fisici[1]? Abbiamo detto che ogni cosa deve avvenire in maniera trasparente rispetto al Guest OS. Si distinguono due tipologie di *Interrupt Descriptor Table*:

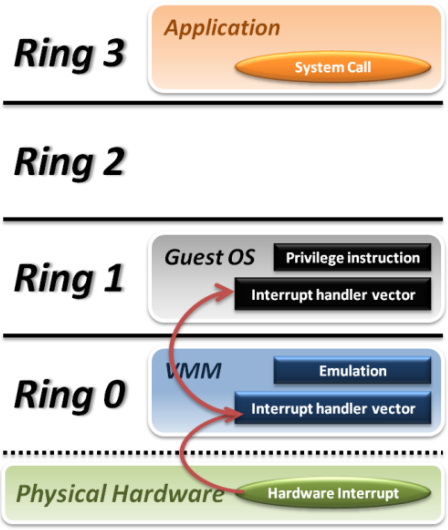
- ***Host CPU IDT***, relativa all'hypervisor;
- ***VM IDT***, relativa alla macchina virtuale e gestita dal Guest OS (che alloca gli handler e pone i relativi indirizzi nella tabella).



[1] Il contenuto di questa sezione è stato scritto a seguito di ricevimento col prof. Vallati

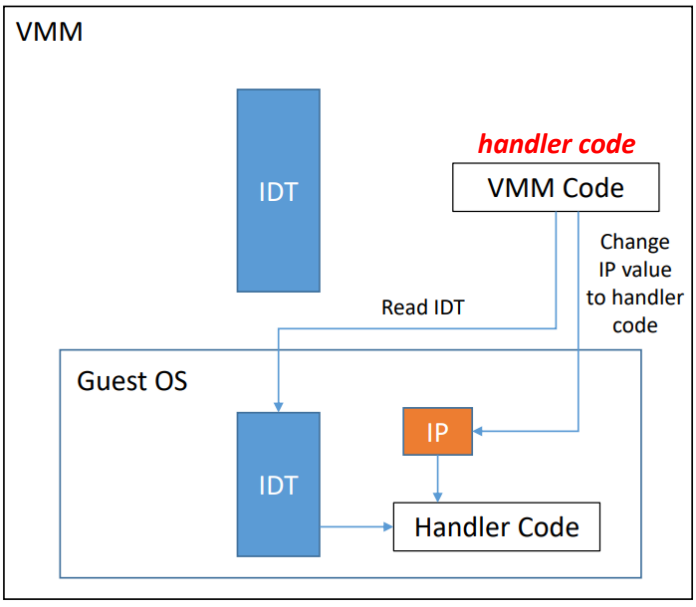
Le interruzioni lanciate da periferiche sono gestite utilizzando la *Interrupt Descriptor Table* dell'hypervisor (il cui indirizzo è puntato dal registro PTBR).

- L'host (lato hardware) visita l'*Interrupt Descriptor Table*, legge l'indirizzo dell'handler code che gestisce l'interruzione e altera l'*Instruction Pointer* ponendo tale indirizzo.



- **Interruzioni non relative a una particolare VM**.
  Se l'interruzione lanciata da una periferica non è relativa a un particolare Guest OS (Esempio: alimentatore che lancia interruzione per segnalare un aumento significativo della propria temperatura) allora l'*handler code* è eseguito interamente a livello hypervisor senza cambi di contesto nel mezzo.

- **Interruzioni relative a una particolare VM**.
  Se l'interruzione lanciata da una periferica è relativa a un particolare Guest OS allora è necessario che l'hypervisor emuli la ricezione di tale interruzione nel Guest OS.
  - Il passaggio dall'IDT dell'hypervisor e l'esecuzione dell'handler code ci sono sempre.
  - L'handler code eseguito nell'hypervisor deve emulare l'interruzione sul Guest OS, coi seguenti passaggi (normalmente fatti in hardware, in questo caso in software):
    - lettura dell'Interrupt Descriptor Table del Guest OS;
    - aggiornamento dello stato del sistema (si pone l'indirizzo letto nell'IDT nell'*Instruction Pointer*).
  - A seguito di questi passaggi viene restituito il controllo al Guest OS (*context-switch*), dove avverrà l'esecuzione dell'*handler code*. Rimane quanto detto sulla virtualizzazione della CPU: se durante l'esecuzione dell'*handler code* vengono invocate istruzioni privilegiate allora avremo molteplici *context-switches* (e non solo quello citato prima).



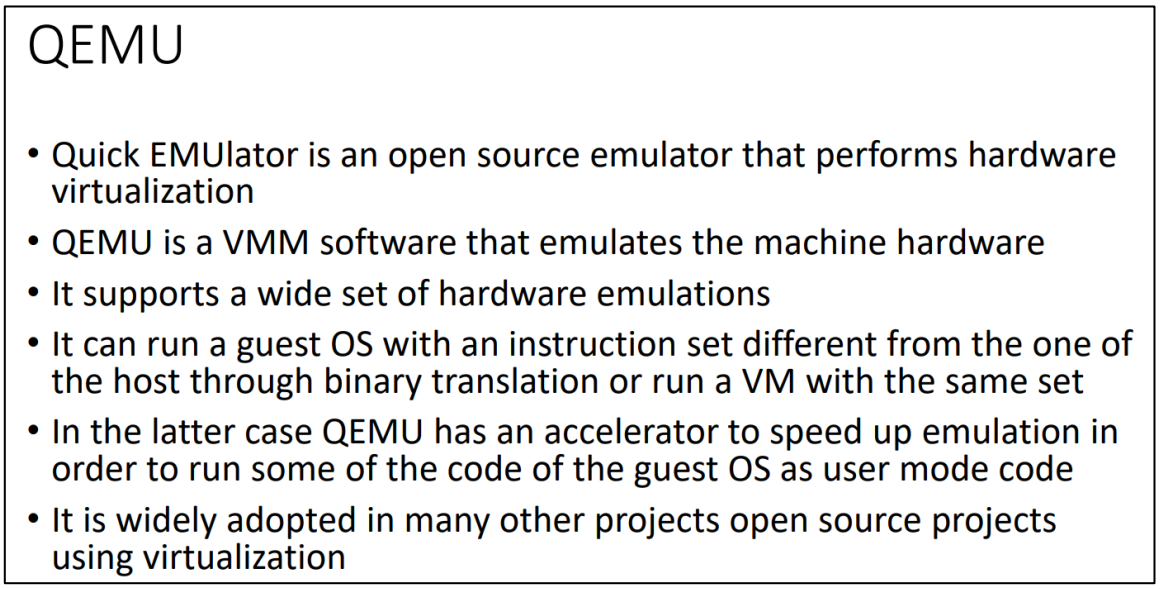
- **Cosa succede nel caso di periferiche completamente emulate (interruzione nata dal Guest OS)?**
  Si gestisce col *trap and emulate approach* spiegato nelle pagine precedenti.

- Nel caso in cui la CPU virtuale disattivi le interruzioni queste dovranno attendere una futura riattivazione: a quel punto le interruzioni verranno gestite nei modi detti.

2.3.5 QEMU

QEMU è acronimo di *Quick EMUlator*: consiste in un emulatore open source che svolge hardware virtualization. È piuttosto popolare perché può emulare diverse architetture: ARM, Intel, Spark…

- Si ha emulazione a tutti gli effetti quando l'instruction set dell'host è diverso da quello del guest.
- Possibile implementare la virtualizzazione qualora l'instruction set sia lo stesso.
- In quest'ultimo caso la velocità di QEMU migliora significativamente. *It can exploit hardware acceleration*.



2.4 Full virtualization techniques - Hardware assisted virtualization

2.4.1 Introduzione

Abbiamo già detto che la virtualizzazione in assenza di supporto hardware comporta la presenza di un overhead significativo.

- Guest OS eseguito interamente in user space, con le tecniche viste prima. *context switches* ed *emulations* incidono negativamente sulla performance.
- Vogliamo introdurre delle modifiche per ridurre il tempo di esecuzione di istruzioni in kernel space, ridurre il tempo necessario per l'emulazione di un'istruzione privilegiata.
- La virtualizzazione *hardware assisted* viene introdotta nel 2006 da AMD e Intel per mezzo di estensioni. L'obiettivo è minimizzare l'overhead. Ci interessa soprattutto la Virtual Machine eXtension a cura di Intel (VMX).

2.4.2 Novità

Elenchiamo le principali novità introdotte da questa tipologia di virtualizzazione

2.4.2.1 New operating modes: root e non-root

In generale abbiamo user mode e kernel mode. VMX introduce due ulteriori modalità, combinabili alle precedenti: *root mode* e *non-root mode*. Questo significa che si hanno quattro possibili modalità di esecuzione, le seguenti:

- *root/system*
- *root/user*
- *non-root/system*
- *non-root/user*

Le prime due sono pensate per l'esecuzione dell'hypervisor, le ultime due per l'esecuzione del codice all'interno della macchina virtuale (Guest OS e applicazioni). Lo scopo è introdurre limitazioni *hardware-controlled* (e non *software controlled* come visto fino ad ora) alle attività dei Guest OS e minimizzare la necessità di software emulation.

2.4.2.2 Maggiore flessibilità: lista di istruzioni privilegiate eseguibili direttamente dalla VM

La *hardware assisted virtualization* introduce la possibilità per l'hypervisor di indicare runtime un set di istruzioni privilegiate che la macchina virtuale può eseguire autonomamente. Questa è una cosa positiva: significa che l'istruzione potrà essere eseguita senza necessità di context-switches ed emulazioni.

- Ogni volta che viene lanciata un'istruzione privilegiata si controlla se questa può essere eseguita autonomamente dalla VM e quindi in *non-root mode*.
- Se l'istruzione privilegiata non rientra nell'elenco detto il controllo passa all'hypervisor e l'esecuzione avviene col *trap and emulate approach* già visto nelle pagine precedenti.

2.4.2.3 Introduzione di nuove istruzioni assembler per gestire il context-switch

Si ha l'introduzione di due istruzioni in assembler per automatizzare alcune delle operazioni richieste durante un context-switch tra hypervisor e macchina virtuale. Tali istruzioni sono consentite in root/system mode e permettono di mettere la macchina virtuale in esecuzione:

- *VMLAUNCH* è eseguito dall'hypervisor quando si vuole eseguire una macchina virtuale per la prima volta;
- *VMRESUME* è eseguito dall'hypervisor quando si vuole rimettere in esecuzione una macchina virtuale precedentemente messa in pausa;
- *VMEXIT* è eseguito dall'hypervisor quando vuole mettere in pausa la macchina virtuale in esecuzione.

L'esecuzione di queste istruzioni permette un miglioramento dell'efficienza.

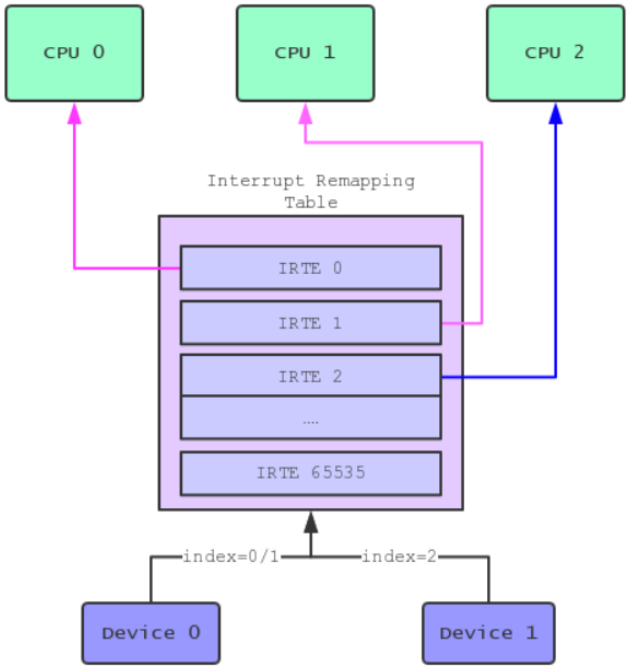
2.4.2.4 Interrupt Remapping Table (IRT)

Tutte queste cose permettono non solo l'esecuzione di istruzioni privilegiate da parte della macchina virtuale, **ma anche l'esecuzione di *interrupts handler***! Un esempio in cui questo può essere cosa interessante è nella virtualizzazione degli I/O devices: se si ha un'emulazione completa (non si contatta il device fisico) potrebbe essere vantaggioso far eseguire l'interrupt handler direttamente al Guest OS.

Come gestiamo questa cosa?

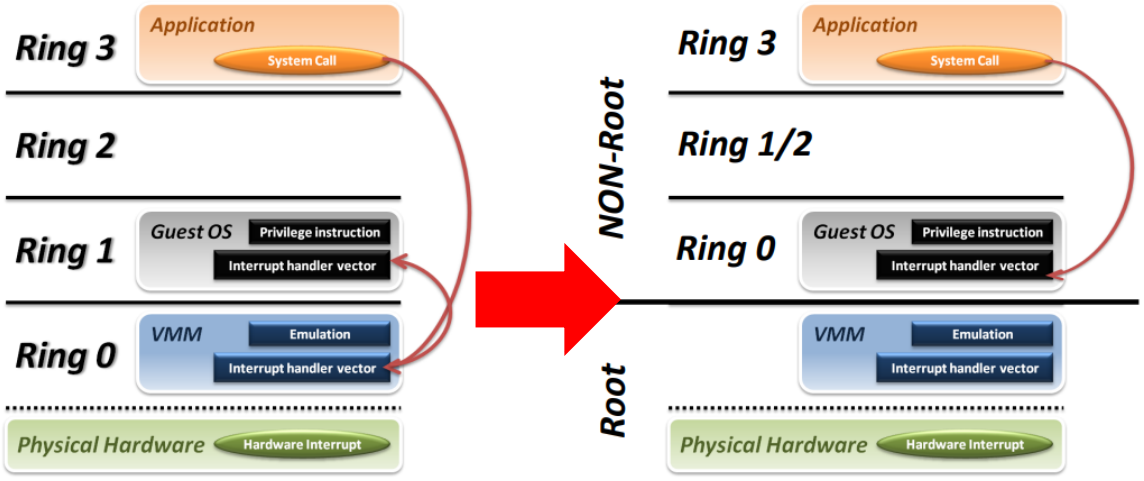
- Abbiamo visto precedentemente che sono presenti due tipologie di *Interrupt Descriptor Table*: una relativa all'hypervisor e una relativa al Guest OS. La seconda IDT è completamente separata dall'hardware (*completely detatched from hardware*), nel senso che la PTBR punta alla prima IDT.

- Si introduce un'ulteriore struttura dati: la *Interrupt Remapping Table* (IRT).
    - La struttura è simile alla *Interrupt Descriptor Table*: abbiamo un numero di entrate pari al numero di possibili tipi di interrupt lanciabili.
    - Ogni entrata della IRT punta a un'entrata di una Interrupt Descriptor Table, entrata che a sua volta punta all'*handler code* da eseguire.
    - La struttura dati è aggiornata dall'hypervisor, che decide se la IRT deve puntare:
        - a un'entrata della IDT del Guest OS (nel caso in cui vada bene l'esecuzione di un handler da parte della macchina virtuale);
        - a un'entrata della IDT dell'hypervisor (se vuole mantenere il controllo).



- Ogni volta che viene lanciato un interrupt:
    - si visita la IRT e si legge la corrispondente entrata;
    - si legge l'entrata della IDT puntata dalla IRT;
    - si esegue l'handler code puntato dall'entrata della IDT.
- Quanto detto ci piace perché avviene tutto in hardware senza necessità di coinvolgere l'hypervisor.

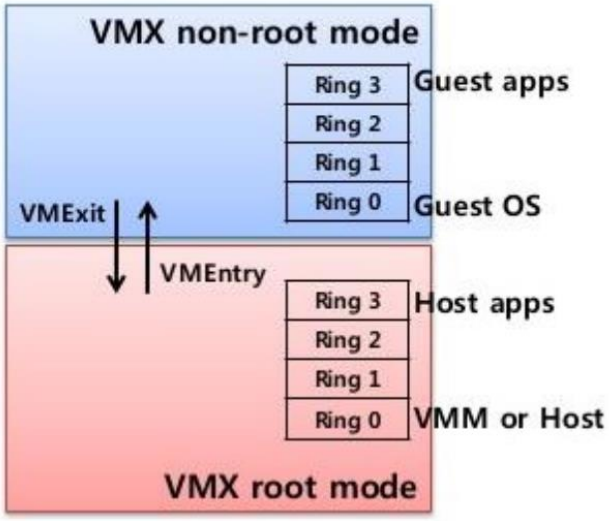
Cerchiamo di capire meglio riprendendo l'esecuzione della INT: cosa cambia?



- **Figura a sinistra**.
  Senza hardware support abbiamo sicuramente una serie di *context switches* e l'esecuzione dell'hypervisor (con operazioni da svolgere manualmente come la modifica dei valori dei registri e/o il lancio dell'esecuzione del codice in kernel space).
- **Figura a destra**.
  Con hardware support e quindi root mode e non-root mode possiamo eseguire l'istruzione INT (così come l'istruzione IRET) senza coinvolgere l'hypervisor e senza svolgere emulazione delle istruzioni. Il codice della system call invocata con INT viene eseguito rimanendo in Ring 0, non si hanno i molteplici *context switches* tra Ring 1 e Ring 0 visti precedentemente).

2.4.2.5 Utilità della root/user mode

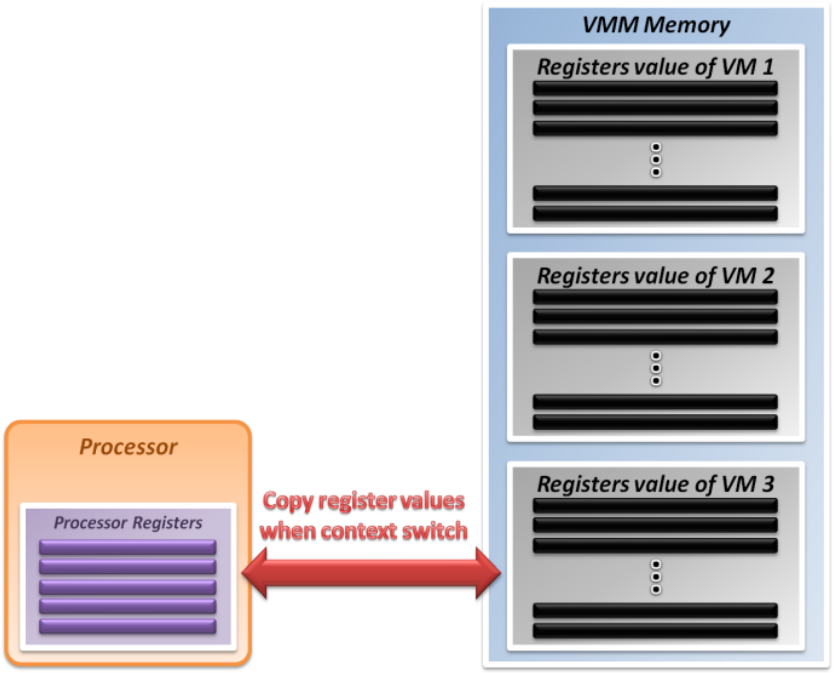
A cosa serve una root/user mode?



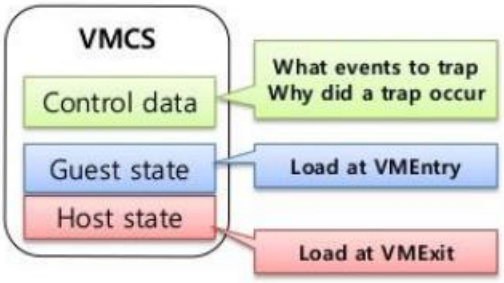
Ricordiamo che esistono due possibili approcci nell'implementazione dell'hypervisor: *Hosted approach* e *Bare Metal Approach*. Nell'hosted approach abbiamo un host OS e quindi un hypervisor eseguito *on top of the operating system*: in tale caso l'host OS è eseguito in root/user mode.

2.4.2.6 Virtual Machine Control Structure

Introduciamo la *Virtual Machine Control Structure* (VMCS), una struttura dati presente nell'hypervisor con lo scopo di mantenere tutte le informazioni riguardanti lo stato delle macchine virtuali. Non si ha solo una struttura dati, ma anche un set di istruzioni che possono essere utilizzate per manipolare queste strutture dati (via hardware).



La struttura memorizza le seguenti informazioni:



- ***Guest state***
  Lo stato dei processori virtuali associati alle macchine virtuali.
  - Con l'esecuzione di VMLAUNCH lo stato del processore virtuale è caricato nel processore fisico.
  - Con l'esecuzione di VMEXIT lo stato presente nel processore fisico viene memorizzato nella struttura dati.

  Tra le informazioni si pone l'Instruction Poiner (IP), cioè la prima istruzione da eseguire a seguito di VMLAUNCH.

- ***Host state***
  Lo stato dei processori fisici prima dell'esecuzione di VMLAUNCH. Contiene lo stato dell'hypervisor, eseguito prima del lancio della macchina virtuale.
  - Con l'esecuzione di VMLAUNCH lo stato presente nel processore fisico viene memorizzato nella struttura dati.
  - Con l'esecuzione di VMEXIT lo stato memorizzato nella struttura dati è caricato nel processore fisico.

  Tra le informazioni si pone l'indirizzo posto nell'Instruction Pointer a seguito di VMEXIT, con l'obiettivo di eseguire l'hypervisor.

- ***VM Execution control***
  Informazioni di controllo: cosa può fare e cosa non può fare la macchina virtuale in non-root mode. Nel caso di istruzioni non consentite si ha VMEXIT e gestione con trap and emulate approach.
  - Presenti flags per *interrupt and I/O management*.
  - Un flag determina se alcune istruzioni critiche devono provocare VM Exit o possono essere eseguite dalla macchina virtuale.
  - Un insieme di flags determina se le operazioni I/O sono permesse nella macchina virtuale o se è necessaria una VMEXIT.
  - Un flag determina cosa deve succedere quando la CPU riceve un *external interrupt* mentre si trova in non-root mode (se la coa deve essere gestita direttamente dalla macchina virtuale o se è necessaria una VMEXIT)

- ***VM enter control***
  Si determina un set di operazioni (con campi e flags) che il processore deve fare ogni volta che si ha un lancio di VMLAUNCH o VMRESUME (*behaviors of the root to non-root transition*).
  - Campi che possono essere settati dall'hypervisor per chiedere alla CPU di emulare un fake interrupt, con tutte le azioni normalmente eseguite in presenza di un'interruzione:
    - Salvataggio dello stato della macchina virtuale in esecuzione nel Guest State
    - Visita dell'IDT per determinare l'indirizzo dell'interrupt handler da eseguire
    - Modifica dei registri della CPU per eseguire l'interrupt handler

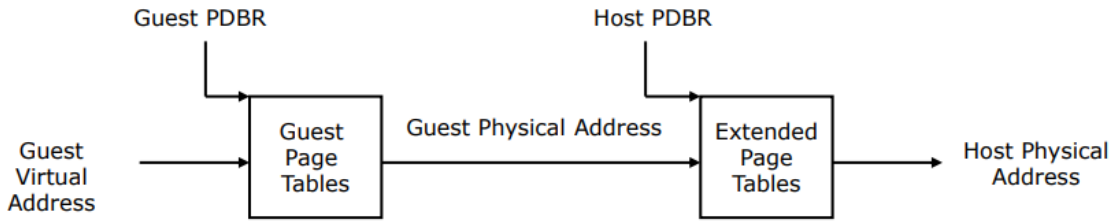
    Tutte questi passaggi sono svolti in hardware minimizzando l'overhead!

- ***VM Exit control***
  Si determina un set di operazioni (con campi e flags) che il processore deve fare ogni volta che si ha un lancio di VMEXIT (*behaviors of the non-root to root transition*).

- **VM Exit reason**
  Informazioni sulle motivazioni che hanno portato all'ultima esecuzione di VMEXIT.
  - Flag/Fields impostati automaticamente in hardware per segnalare le ragioni della VMEXIT, senza necessità di dover leggere l'intero guest state nell'apposita struttura dati.

2.4.3 Virtualization of RAM

Si parla anche in questo caso di *Extended Page Tables*: l'hardware che implementa la host MMU è modificato per avere due registri PTBR:



- uno relativo alla page table del Guest OS;
- uno relativo alla page table dell'hypervisor.

Come prima si applica la funzione $H \circ G$: la prima page table implementa G, la seconda H!

- **Vantaggi**.
  La presenza di due PTBR permette una traduzione in hardware senza necessità di coinvolgere l'hypervisor (niente VMEXIT).
  Svantaggi.
  L'estensione aumenta i costi per tradurre un indirizzo, dato che l'hardware deve compiere un maggior numero di accessi in memoria (*for each translation 24 additional memory accesses are required*). Per ridurre il conseguente overhead viene implementata una memoria cache a sostengo della page table.

2.4.4 Virtualization of I/O: hardware passthrough

2.4.4.1 Introduzione

Sfruttiamo la virtualizzazione *hardware assisted* per un'esecuzione efficiente di istruzioni I/O.

- Sappiamo che le operazioni di I/O comportano diversi VMEXIT.
- Due tipologie di operazioni di I/O:
  - Operazione per triggerare una funzione / fornire dati a una periferica
  - Periferica che fornisce dati all'applicazione / al sistema operativo.
- In generale l'I/O è gestito con emulazione: l'hypervisor crea ed espone rappresentazioni virtuali delle periferiche. La VM interagisce unicamente con la rappresentazione virtuale.
  - Parole chiave: emulazione e *context-switches*, operazioni che comportano spreco di tempo.
  - La cosa potrebbe essere accettabile per periferiche non particolarmente veloci o raramente invocate.

- **Come miglioriamo la situazione?**
  Diamo all'*hypervisor* la possibilità di dare accesso diretto a una periferica a una macchina virtuale! L'approccio detto è noto come *passthrough* (nel senso di "*passare l'hypervisor e toccare direttamente le periferiche*").
  - **Conseguenza**: la macchina virtuale ha accesso esclusivo alla periferica! Non può essere utilizzata da altre macchine virtuali o addirittura dall'hypervisor stesso.
  - Non si hanno traduzioni, ergo l'overhead è ridotto drasticamente se non eliminato

Esempi in cui l'approccio risulta sensato sono l'interfaccia di rete e la GPU.

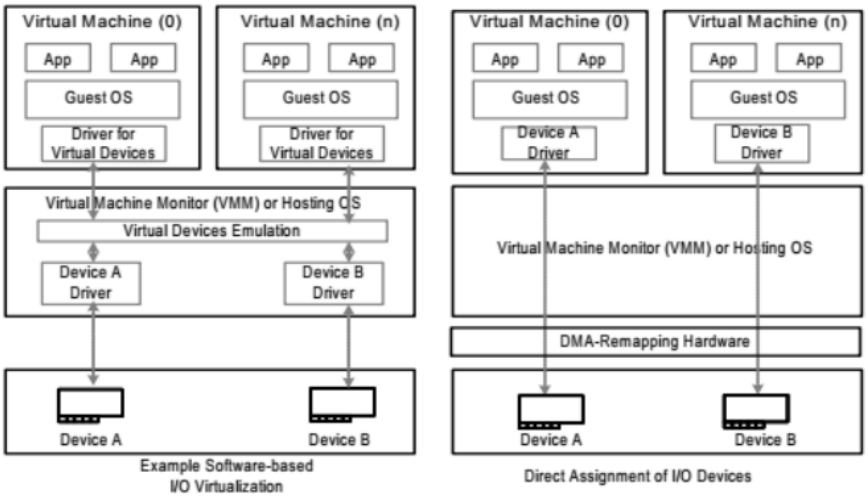
- L'interfaccia di rete è unica sulla macchina fisica e costituisce un collo di bottiglia. Già questo può dare luogo a rallentamenti, vorremo evitarne ulteriori.
- La GPU è complessa da virtualizzare.

L'implementazione dell *passthrough* richiede hardware support, precisamente abbiamo bisogno di due funzionalità:

- **Gestione degli spazi di memoria degli I/O (*I/O mapping*)**.
  Mapping dello spazio di memoria fisico dell'I/O nello spazio di memoria virtuale della macchina virtuale, in modo tale che il Guest OS possa scrivere direttamente sui registri della periferica.
- **Gestione delle interruzioni (*Interrupts*)**.
  La periferica può generare interruzioni, la macchina virtuale può gestire tali interruzioni eseguendo i propri *handler code*.

2.4.4.2 I/O mapping

Ragioniamo sulla prima funzionalità. Abbiamo detto che vogliamo mappare gli indirizzi fisici relativi alla periferica negli indirizzi virtuali assegnati a una particolare macchina virtuale.



- Vogliamo svolgere operazioni di lettura e scrittura su questi indirizzi senza necessità di intervento da parte dell'*hypervisor*.
- Allo stesso tempo è nostro interesse che operazioni di lettura e scrittura su indirizzi relativi ad altre periferiche vengano intercettate, in modo tale da restituire il controllo all'*hypervisor*.

- L'architettura Intel include un ***I/O bitmap*** nella struttura dati VMCS. Il bit in questione indica all'hardware se porzioni indirizzi nello spazio di I/O devono essere gestito in modo classico (emulazione) o per mezzo di hardware *passthrough*. In quest'ultimo caso il processore permette le operazioni di lettura e scrittura per questi indirizzi.

- **Ricapitoliamo i passaggi**.
  - Siamo, ovviamente, in *non-root mode*. La CPU vuole eseguire un'operazione di lettura o scrittura su un indirizzo relativo all'I/O.
  - La CPU verifica nell'I/O bitmap relativo all'indirizzo su cui il programma vuole leggere o scrivere.
    - Se il bit è settato l'operazione è eseguita.
    - Se il bit non è settato viene triggerato un *context-switch* e l'esecuzione viene emulata dall'hypervisor.

2.4.4.3 Interrupts

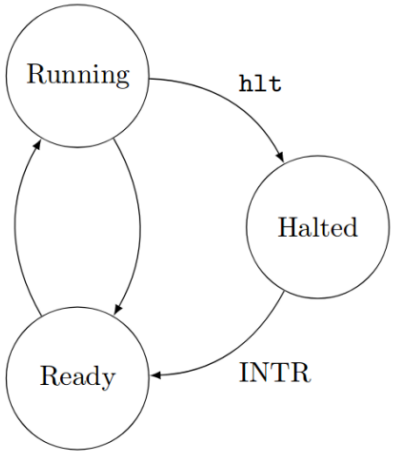
Abbiamo bisogno di un meccanismo che supporti due tipologie di interruzioni:

- Interrupt non relativo a un *passthrough device*, gestione da parte dell'hypervisor (si ha un VMEXIT, gestione di default);
- Interrupt relativo a un *passthrough device*, gestione diretta da parte della macchina virtuale senza VMEXIT.

Intel VMX offre tale supporto: l'hardware visiona direttamente l'*Interrupt Descriptor Table* del Guest OS e aggiorna lo stato del processore per eseguire il relativo *handler code*. Il Guest OS gestisce la IDT senza il coinvolgimento dell'hypervisor. Implementiamo questa cosa introduciamo la ***Interrupt Remapping Table***, di cui abbiamo già parlato nelle pagine precedenti.

Problema. Cosa succede se la macchina virtuale non è in esecuzione? Il *passthrough system* deve gestire anche questo caso.

- ***VM states***.
  La macchina virtuale può trovarsi in uno dei seguenti stati:
  - ***running***, la VM sta eseguendo codice;
  - ***ready***, la VM è pronta per eseguire codice (altre macchine virtuali stanno lavorando, oppure sta lavorando l'hypervisor);
  - ***halted,*** stato in cui si trova la VM a seguito di invocazione dell'istruzione hlt (shutdown della macchina). La macchina non può essere invocata dall'hypervisor, salvo riavvio ella stessa (da un external input, che può essere un tasto premuto in un'interfaccia di gestione o la ricezione di un interrupt)



- ***Interrupts management***.
  In presenza di un passed-through device dobbiamo gestire le relative interruzioni in un certo modo.
  - ***Running state***.
    Gestione diretta da parte del processore, cioè andare direttamente all'IDT del Guest OS senza coinvolgimento dell'hypervisor.
  - ***Ready state***.
    Il processore sta eseguendo altre macchine virtuali. L'interrupt viene mantenuto in memoria e recuperato quando il processore riprenderà l'esecuzione della macchina virtuale coinvolta.
  - ***Halted state***.
    L'interrupt deve essere memorizzato e lo stato della macchina virtuale deve essere cambiato in *ready state*.

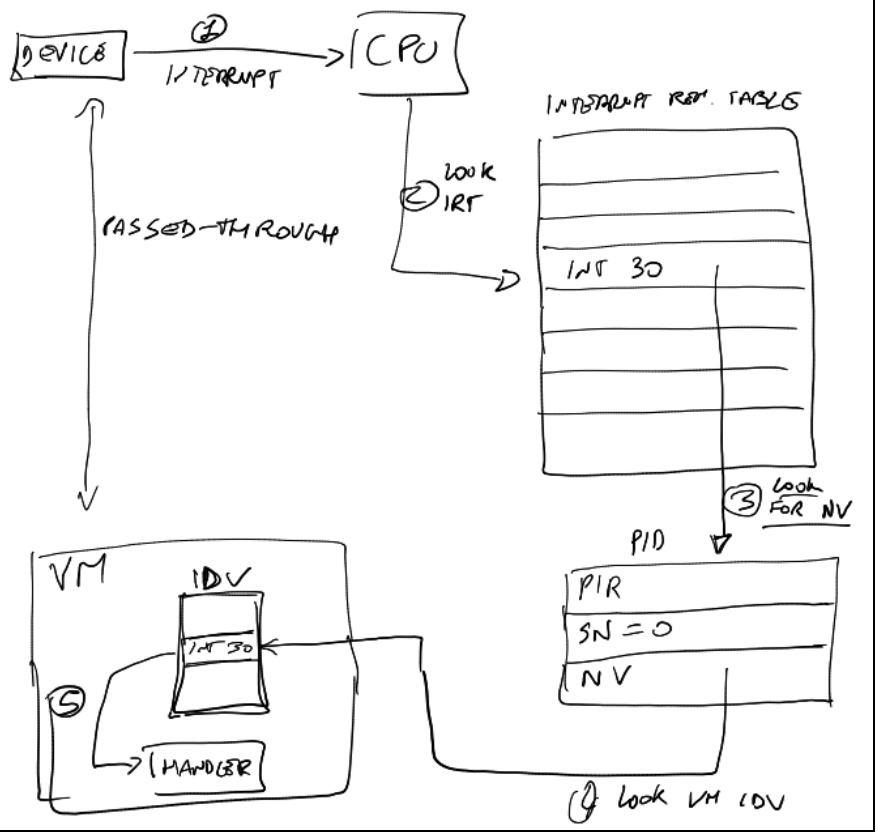
  Come possiamo implementare i passaggi da adottare in caso di *ready state* e *halted state*?
  - Il salvataggio dell'interruzione avviene con un meccanismo noto come *posted interrupt mechanism*. Si permette la memorizzazione delle interruzioni e la notifica alle macchine virtuali ad esecuzione ripresa, il tutto senza il coinvolgimento dell'hypervisor.
  - Per ogni macchina virtuale si considera il ***Posted Interrupt Descriptor*** (PID).
  - Le entrate della Interrupt Remapping Table relative a interruzioni che devono essere gestite dal Guest OS conterranno il PID e non il puntatore alla relativa IDT (o all'handler code).
  - Un *Posted Interrupt Descriptor* è una struttura dati caratterizzata dai seguenti campi:
    - ***Posted Interrupt Request*** (PIR)
      Componente che prende nota di tutte le interruzioni pendenti.
    - ***Suppress Notification flag*** (SN)
      Flag che determina se è necessario notificare o meno la CPU (0 se deve essere

notificata, 1 se non deve essere notificata) a seguito di inserimento di un'interruzione nel PIR.

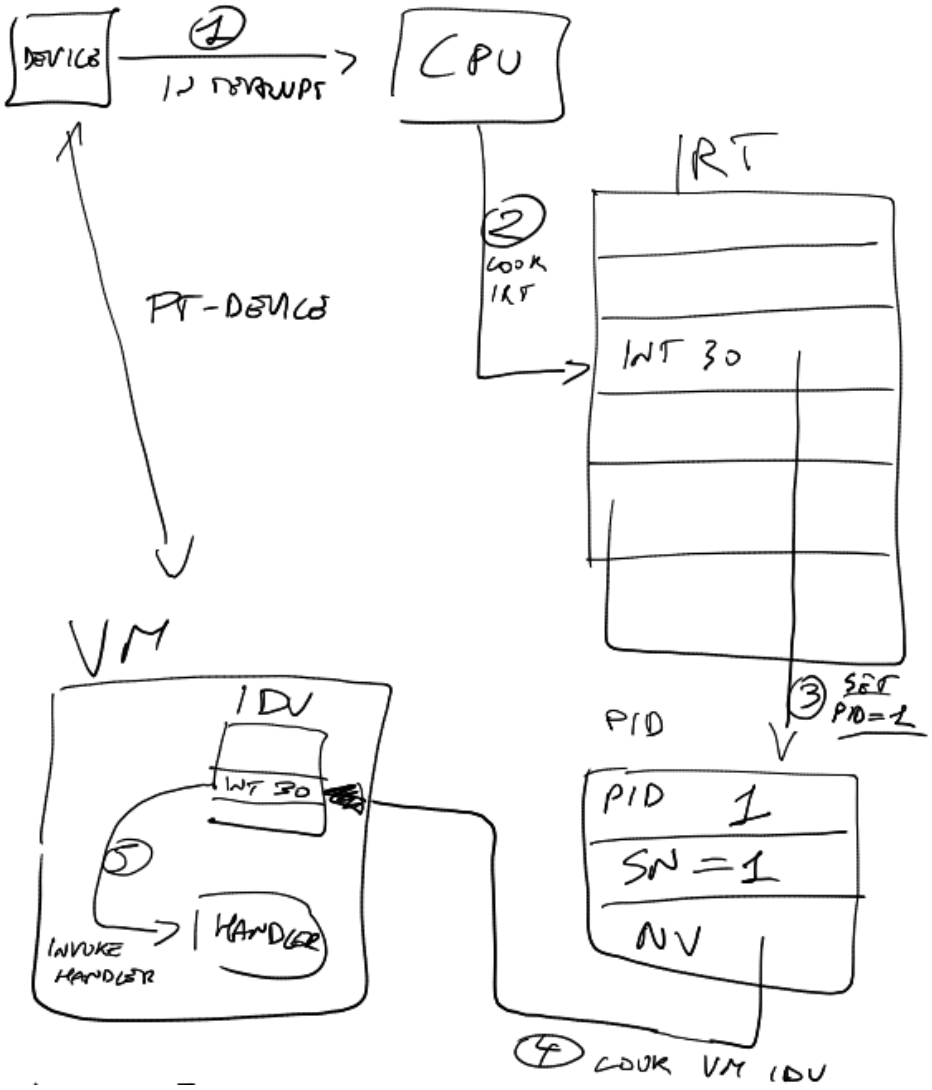
- ***Notification Vector*** (NV)
  Puntatore all'Interrupt Descriptor Table, utilizzato per notificare la VCPU della macchina virtuale.

Ricapitoliamo i passaggi già accennati e introduciamone di nuovi.

- L'Interrupt Controller deve gestire un'interruzione.
- Visita la IRT, se nella relativa entrata si fa riferimento a un'istanza del *Post Interrupt Descriptor* ci si muove nel seguente modo:
  - Si imposta il relativo bit nel Posted Interrupt Request (PIR)
  - Se SN è uguale ad 1 non si fa altro, non devono essere trasmesse notifiche alla CPU.
  - Se SN è uguale a 0 e la VM è in *running state* si utilizza il Notification Vector per notificare il processore virtuale, quello relativo alla macchina virtuale.
  - Se SN è uguale a 0 e la VM è in *halted state* viene triggerato il risveglio della VM.
- A seguito di cambiamenti nello stato della macchina virtuale l'hypervisor deve aggiornare il PID.
  - **Se la VM è posta in *running state*.**
    L'hypervisor pone SN uguale a 0 e aggiorna il relativo Notification Vector in modo che questo punti alla IDT della VM (*Active Notification Vector*). Si deve anche verificare che non vi siano bit settati nel PIR, e in caso affermativo sistemare il *Notification Vector* in modo tale da notificare il processore (che gestisce interruzioni ricevute durante stati precedenti).
  - **Se la VM è posta in *ready state***. L'hypervisor pone SN uguale a 1
  - **Se la VM è posta in *halted state***. L'hypervisor pone SN uguale a 0 e aggiorna il relativo *Notification Vector* ponendo un *Wakup Notification Vector* che permette di triggerare il risveglio della macchina virtuale.

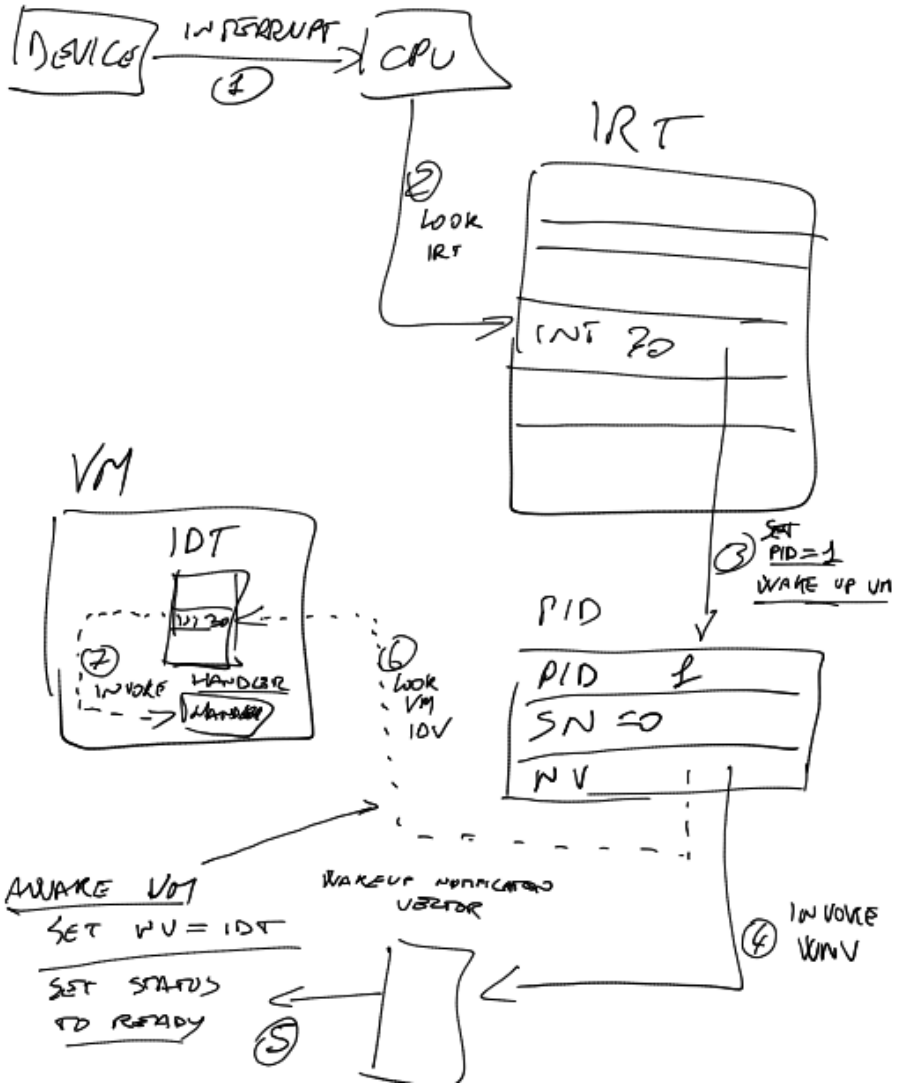
Ricezione delle interruzioni	
Macchina virtuale coinvolta nell'interruzione si trova in running state	
- Un device genera un interrupt: questo è stato ricevuto dalla CPU. La macchina virtuale coinvolta è in running state. - L'hardware visita la IRT, che punta al PID. Troviamo che SN è uguale a 0. - Si utilizza il *Notification Vector*, che punta a un'entrata dell'IDT della VM. - L'handler code puntato dall'entrata dell'IDT viene eseguito	

Macchina virtuale coinvolta nell'interruzione si trova in ready state



- Un device genera un interrupt: questo è stato ricevuto dalla CPU. Questa sta eseguendo una macchina virtuale diversa da quella coinvolta nell'interrupt. L'interrupt è relativo a un passed-through device.
- L'hardware visita la IRT, che punta al PID. Troviamo che SN è uguale a 1, questo significa che gli interrupts non possono essere gestiti direttamente.
- Il PIR viene posto uguale ad 1. **A questo punto ci si ferma**.
- Quando la macchina virtuale passa da *ready state* a *running state* la gestione dell'interruzione viene ripresa.
- Alla ripresa viene trovato il PIR settato. Si utilizza il Notification Vector, che punta a un'entrata dell'IDT della VM.

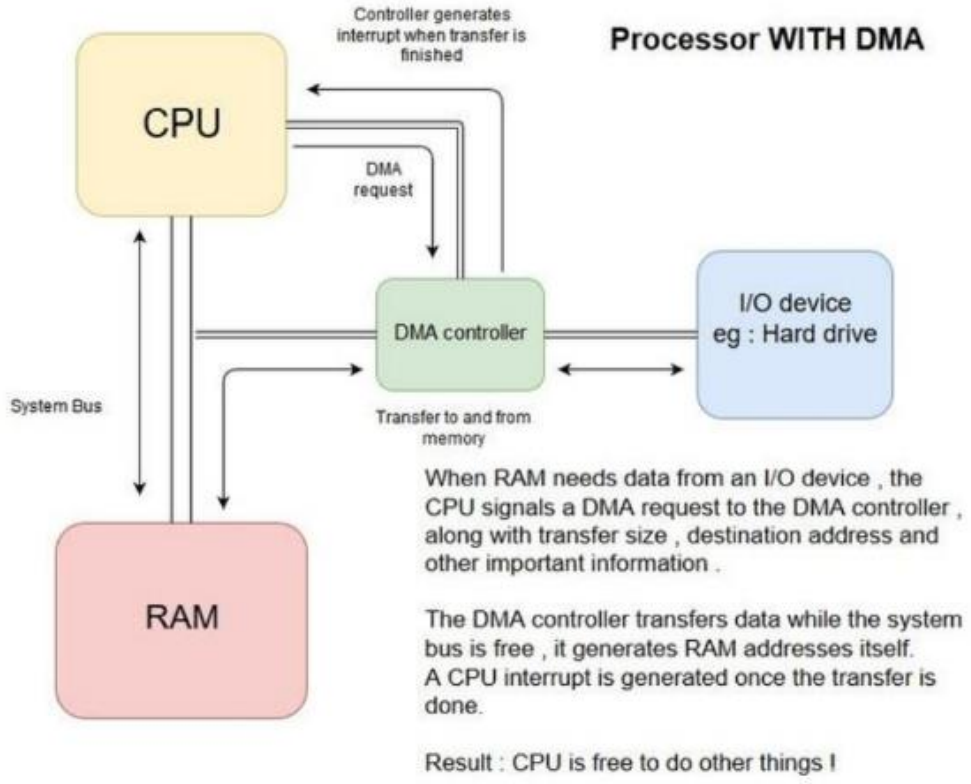
Macchina virtuale coinvolta nell'interruzione si trova in halted state



- Un device genera un interrupt: questo è stato ricevuto dalla CPU. Questa sta eseguendo una particolare macchina virtuale, diversa da quella coinvolta nell'interrupt. L'interrupt è relativo a un passed-through device.
- L'hardware visita la IRT, che punta al PID. Troviamo che SN è uguale a 0, questo significa che un processo verrà triggerato immediatamente.
- Il PIR viene posto uguale ad 1.
- Dal Notification Vector si vede che la macchina è in *halted state*: non si punta all'Interrupt Descriptor Table della macchina virtuale, ma ad un'altra struttura dati che permette di triggerare un *awakening procedure*: la macchina virtuale viene posta in *ready state*, il Notification Vector viene modificato affinchè punti all'Interrupt Descriptor Table della macchina virtuale.
- Si gestisce il passaggio da *ready state* a *running state* nei modi detti precedentemente.

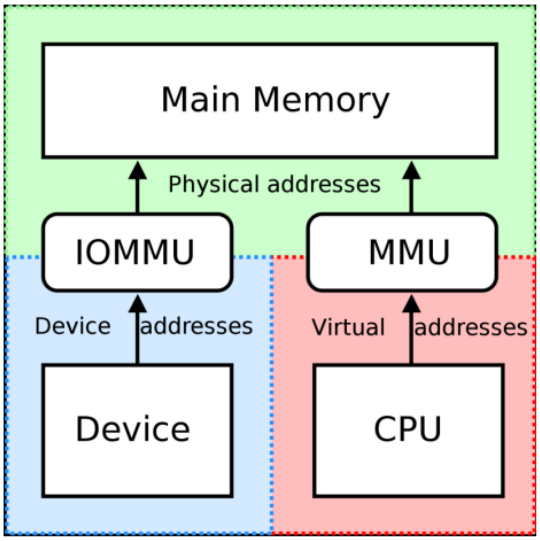
2.4.5 Dispositivi DMA

I dispositivi *Direct Memory Access* (DMA) sono periferiche in grado di leggere i dati direttamente dalla memoria RAM, senza interpellare la CPU.



- Necessario *hardware-support* per poter realizzare questa cosa in lettura e scrittura: *DMA controller*.
- Necessari *device drivers* che configurino il DMA: registri che specificano le porzioni di RAM dove è possibile svolgere operazioni di lettura e scrittura.

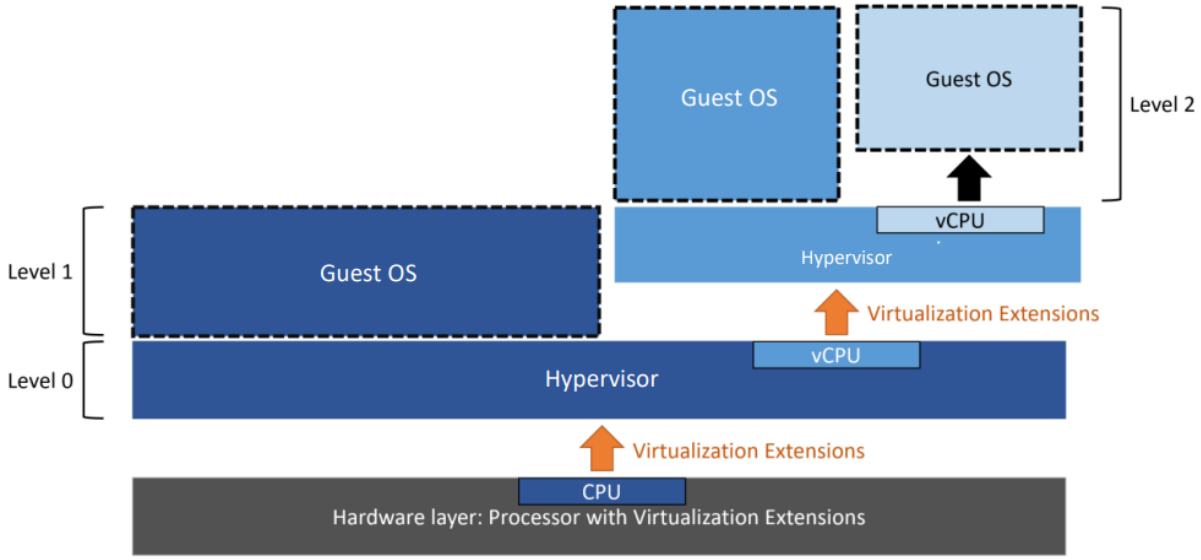
L'I/O device interagisce direttamente con la memoria, ma noi ragioniamo per indirizzi virtuali: è necessario tradurli in indirizzi fisici!! Introduciamo a tal proposito una IOMMU per svolgere tale traduzione: abbiamo un circuito dedicato esclusivamente ai *DMA devices*.



- La IOMMU lavora in parallelo alla MMU.
- Il numero di livelli di traduzione è sempre lo stesso, i passaggi legati alla virtualizzazione non cambiano.

2.4.6 Nested virtualization

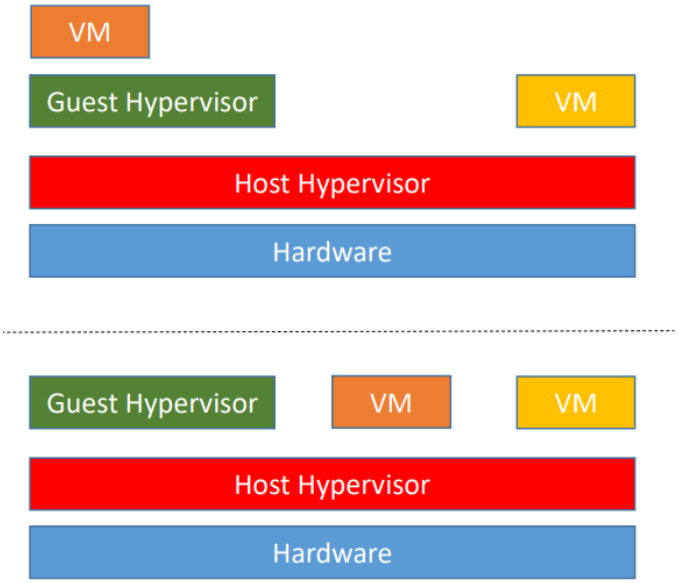
Domanda particolare: è possibile eseguire una macchina virtuale all'interno di una macchina virtuale? Sì, basta scaricare un guest hypervisor all'interno della macchina virtuale.



L'overhead è significativo e soprattutto non è possibile fare ricorso alla *hardware-assisted virtualization*: si deve usare per forza il *trap and emulate approach*.

Come possiamo gestire l'overhead del guest Hypervisor?

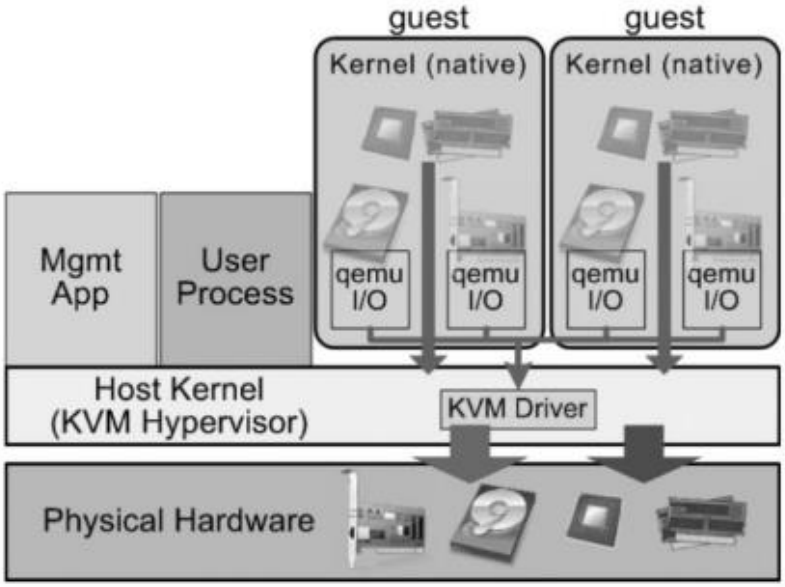
- Supponiamo che l'Host Hypervisor e il Guest Hypervisor coincidono in software: se riusciamo a farli comunicare direttamente siamo in grado di migliorare le prestazioni del sistema.
- L'Host hypervisor è supportato dall'hardware ed è veloce.
- Il Guest Hypervisor parla con l'Host Hypervisor e chiede a questo di svolgere per lui le operazioni.



2.4.7 Kernel-Based Virtual Machine (KVM)

KVM, acronimo di *Kernel-Based Virtual Machine*, è un hypervisor che implementa la full virtualization, basato sul kernel di Linux.

- Può eseguire differenti macchine virtuali sulla stessa macchina.
- Non è eseguito in Linux come un normale programma: è posto nel Kernel (si veda la figura).
- È garantito l'isolamento delle macchine virtuali.
- Si rispettano le proprietà alla base della virtualizzazione, in particolare il dynamic provisioning delle risorse.
- Utilizzato frequentemente insieme a QEMU.



2.5 Paravirtualization and Operating System Level Virtualization

2.5.1 Introduzione

Abbiamo visto che

- La *full virtualization* richiede la virtualizzazione dell'intera architettura. Overhead significativo (si pensi a *trap and emulate approach*)
- La rappresentazione virtuale dell'hardware viene esposta alla macchina virtuale, il sistema operativo può essere eseguito senza modifiche che tengano conto della virtualizzazione.
- Il numero di *context-switches* è significativo in assenza di *hardware-support*.
- L'emulazione è svolta frequentemente (per tutte quelle istruzioni che non possono essere eseguite direttamente dal *Guest OS*)

Introduciamo due ulteriori forme di virtualizzazione:

- la *paravirtualization* (obsoleta);
- la *Operating System Level virtualization* (utilizzata).

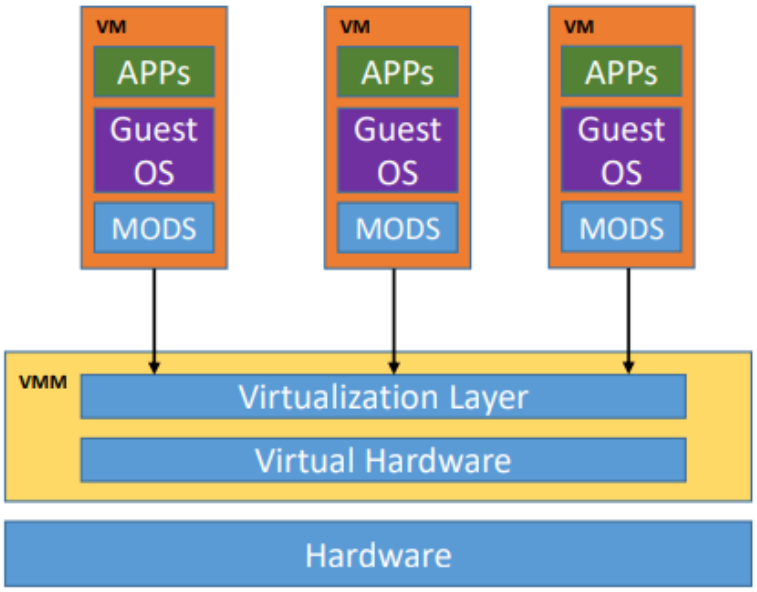
Entrambe hanno lo stesso scopo: ridurre l'overhead rispetto alla *full virtualization* (dove si hanno molteplici invocazioni di istruzioni privilegiate e molteplici *context-switches*).

2.5.2 Paravirtualization

2.5.2.1 Introduzione

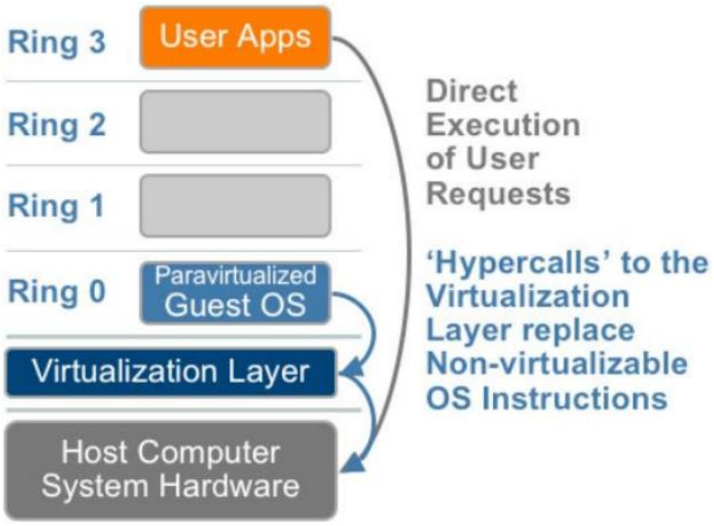
La *paravirtualization* è una forma di virtualizzazione.

- Essa parte dall'idea che i sistemi operativi eseguiti dentro una macchina virtuale possano essere modificati: il sistema diventa consapevole di essere eseguito all'interno della Virtual Machine e l'implementazione dell'hypervisor tiene conto di tali modifiche.
- Le modifiche al *Guest OS* avvengono nel *kernel*: non si interviene sulle applicazioni normalmente eseguite all'interno del sistema operativo.



- Ovviamente è presente l'hardware.
- Ovviamente è presente l'*hypervisor* (VMM), che però risulta diviso in due componenti:
  - *Virtual hardware* (core component)
  - *Virtualization layer*, che espone certe API.
- Ogni macchina virtuale è caratterizzata da applicazioni in esecuzione, il Guest OS e le modifiche (MODS) al Guest OS: tali modifiche permettono l'implementazione di funzionalità precedentemente gestite con *context-switches* ed *emulations* per mezzo dell'invocazione delle API fornite dall'*hypervisor*.

- Ricordarsi che non parliamo di *hardware-assisted virtualization*: i livelli di privilegio sono due, *kernel* e *user*.
- Si ha sempre emulazione di istruzioni privilegiate da parte dell'hypervisor, ma con una differenza significativa: si ha un solo *context-switch* (quando si invocano le API) e non molteplici *context-switches* come nel *trap and emulate approach*.



- Le applicazioni sono sempre eseguite in Ring 3, il Guest OS è sempre eseguito in Ring 0. Il Guest OS è ovviamente modificato tenendo conto della *paravirtualization*.
- Si implementano le ***hypercalls***, APIs che l'hypervisor espone alle macchine virtuali e che possono essere invocate dalle stesse. Si modifica il Guest OS per invocare le *hypercalls* al posto delle istruzioni privilegiate.
- Il codice delle *hypercalls* è eseguito pienamente in system mode (a differenza del trap and emulate approach): può eseguire istruzioni privilegiate e interagire direttamente con l'hardware fisico.

- **Vantaggi**
  - Migliora decisamente le performance della virtualizzazione.
  - Non richiede hardware-support.
- **Svantaggi**
  - Richiede modifiche al Guest OS.
  - Sistemi operativi proprietari potrebbero non supportare la *paravirtualization*.

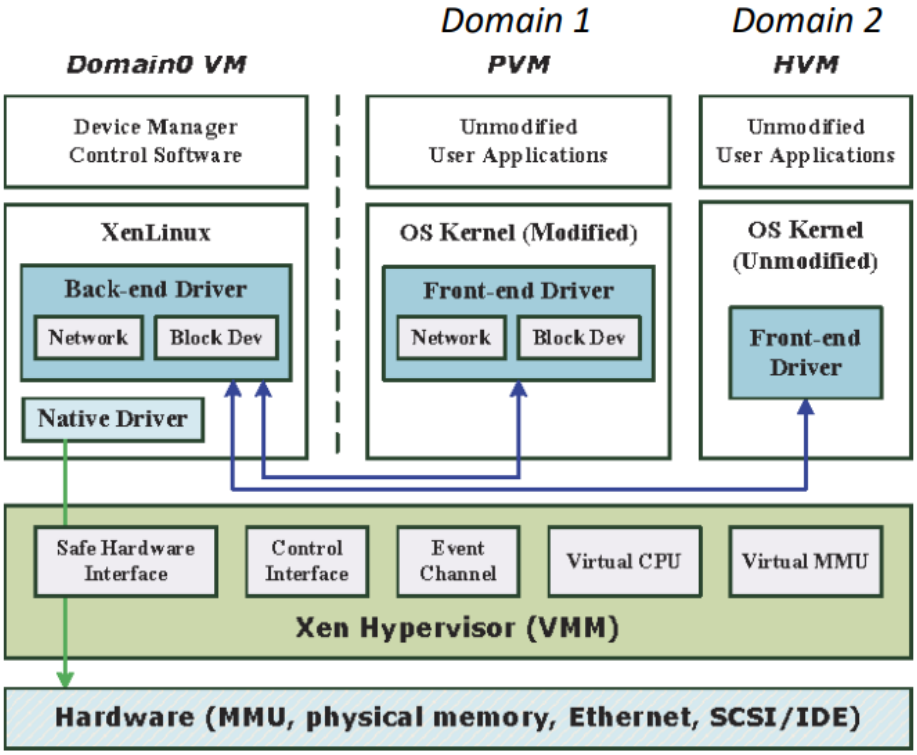
All'inizio certi sistemi operativi (compatibili con la *paravirtualization*) erano molto popolari, ma con l'avvento dell'*hardware-support virtualization* la *full virtualization* ha recuperato lo svantaggio e tali sistemi sono caduti in disuso.

2.5.2.2 Xen Operating System

Un esempio di sistema operativo è Xen

- Prima versione rilasciata nel 2003, prima dell'introduzione dell'*hardware-assisted virtualization* nel 2006 con Intel e AMD.
- Si modifica il kernel di Linux per invocare *hypercalls*.
- Molto popolare per la sua efficienza nei primi anni dopo il lancio.
- Presente anche oggi, ma non più popolare come una volta: oggi supporta sia *paravirtualization* che *full virtualization via hardware-support*.

Architettura. Alcune osservazioni sull'architettura del Sistema Operativo



- Ovviamente è presente un *hypervisor*
- Sono eseguite on top of the *hypervisor* molteplici macchine virtuali, che nello XEN gergo sono note come *Domains*. Su ciascuna può essere eseguito un Linux OS.
- La gestione dei *domains* è eseguita da una terza macchina virtuale denominata *Domain 0* (*a curious approach* – cit.). Questa ha accesso alle API che permettono la creazione e la distruzione di domains. Possibile installare un Linux OS anche sul *Domain 0*.
- I domains non comunicano direttamente con l'*hypervisor*, ma passano da *Domain 0*.

2.5.3 Operating System Virtualization

2.5.3.1 Premessa: lightweight virtualization, perché?

Non siamo sempre interessanti all'intero set di meccanismi offerti dalla virtualizzazione.

- **Esempio**: condivisione sicura di risorse hardware tra diverse applicazioni eseguite sullo stesso sistema (*application isolation*). Non abbiamo bisogno di un'intera macchina virtuale, ma solo di *isolated environments* per l'esecuzione di applicazioni.
- L'overhead non è solo legato all'*hypervisor*, ma anche all'esecuzione del *Guest OS* sulla VM.
    - L'overhead dell'*hypervisor* non ci spaventa grazie alla *paravirtualization* (le cose vanno già molto più veloci rispetto alla *full virtualization*)
    - Non ci piace l'overhead del sistema operativo se non ci serve l'intera macchina virtuale.

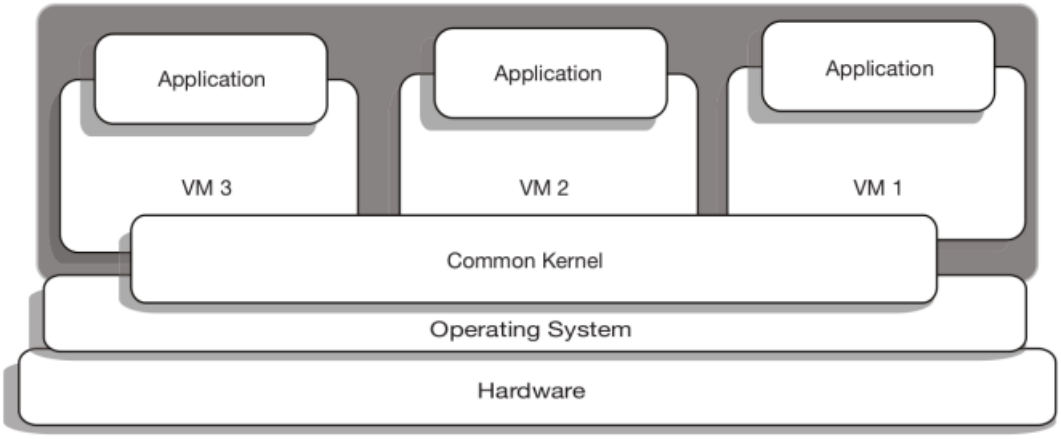
  Se manteniamo l'isolamento per mezzo di differenti macchine virtuali poniamo in ciascuna di queste un sistema operativo: in molti casi tutte le macchine virtuali hanno installato lo stesso sistema operativo con la stessa configurazione!!

- L'idea è di creare un ambiente che garantisca solo l'*isolation*, col kernel condiviso tra *isolated environments*: ***lightweight virtualization***.

  Goal: "*create a lightweight execution environment for services and applications that do not require the virtualization of an entire system and still guarantees the same advantages of virtualization*".
    - Isolamento
    - *Self-contained environment* (con cui non può interagire nessun altro environment, ad esempio se ci sono più environments sulla stessa macchina)
    - Istanziamento dinamico (come le macchine virtuali)

2.5.3.2 Introduzione alla Operating System Virtualization

Si parla di *Operating System Virtualization*

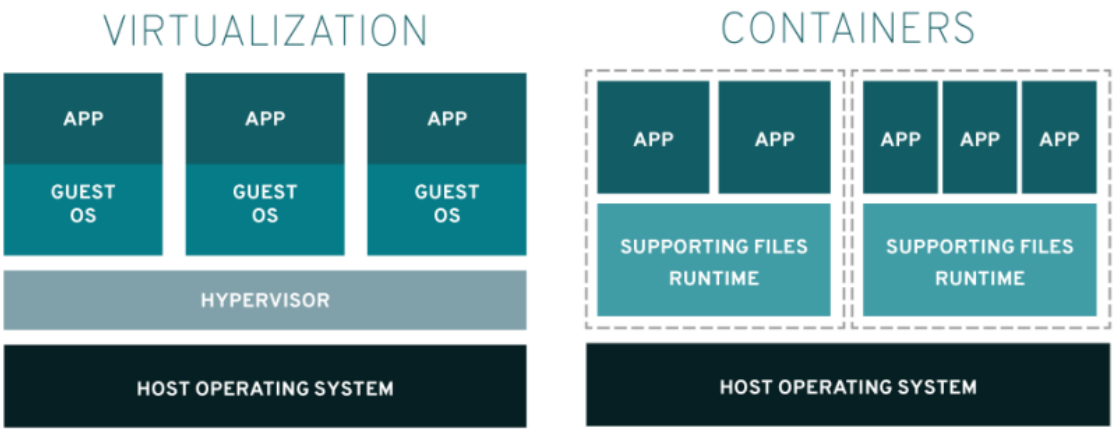


- Questa virtualizzazione è *operating system based*: si modifica il kernel del sistema operativo per introdurre nuove funzionalità, in particolare la possibilità di creare *virtualized environments* per l'esecuzione di applicazioni.
- L'hypervisor è rimosso e il suo ruolo viene trasferito al sistema operativo, in particolare al kernel del sistema (che è comune a tutte le macchine virtuali).
- Si garantisce l'isolamento all'interno di un singolo sistema operativo, le macchine possono essere istanziate dinamicamente e non possono comunicare tra di loro dinamicamente.

- **Esempi**: *FreeBSD Jails, Linux Containers* (molto popolare).
- **Vantaggi**
  - Riduzione significativa dell'overhead dovuto all'esecuzione del sistema operativo (*OS virtualization is lighter in overhead*).
  - Il sistema operativo può gestire un numero maggiore di macchine virtuali rispetto alla *full virtualization*, a parità di risorse.
- **Svantaggi**
  - Le macchine virtuali condividono lo stesso kernel: attenzione a *security issues*.
  - Non tutti i sistemi operativi esistenti sono compatibili con questa forma di virtualizzazione.

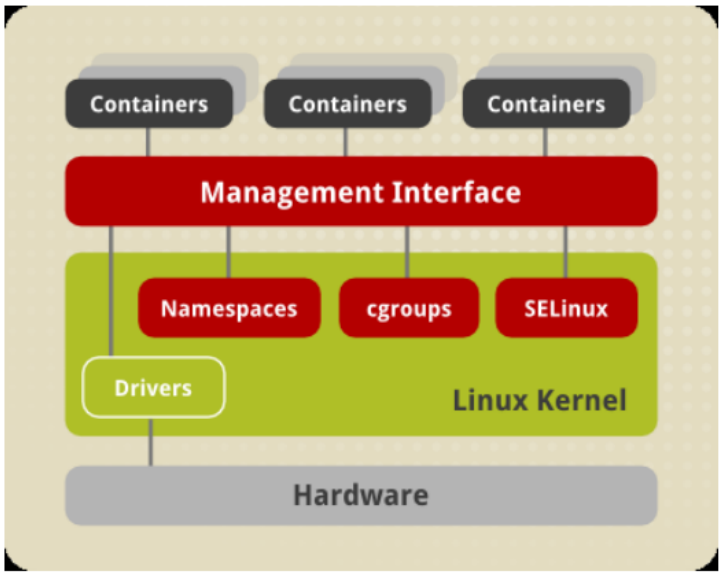
2.5.3.3 Containers in Linux Operating Systems

Il *container* consiste in un meccanismo per l'isolamento dei processi.



- Questi hanno l'idea di essere gli unici processi in esecuzione sulla macchina (*isolated environment*).
  - Applicazioni eseguite in containers diversi non possono comunicare tra di loro.
  - Le applicazioni possono accedere solo a un subset delle risorse a disposizione del Sistema Operativo, quelle che sono state assegnate ai relativi containers.
- Non sono macchine virtuali: non c'è un guest kernel, non viene seguito un Guest OS al loro interno! L'unico sistema eseguito è l'Host OS.
- L'esecuzione in containers non comporta una degradazione della performance dell'applicazione!

Il principale esempio è costituito dai *Linux Containers* (utilizzati in *Docker*)!



- Si utilizzano due features del kernel di Linux: *namespaces* e *control groups*.
- Le due features garantiscono la realizzazione di un *isolated environment* per ciascuna applicazione.
    - Nessuna interazione con altre applicazioni.
    - Il container può accedere soltanto alle risorse a cui ha effettivamente diritto di accedere.

- **Ruolo di *namespaces***.
  Meccanismo che permette di "segregare" risorse del sistema, nel senso che vengono nascoste a particolari processi.
    - Estensione della funzione `chroot()` di UNIX, che permette di indicare la porzione del file system in cui l'applicazione viene segregata: questo rende la cartella scelta la root del file system del processo. Era pensato per processi considerati non affidabili: se possono accedere solo a una porzione del file system (quella da noi indicata) non possono fare danni.
    - Esistono molteplici *namespaces*, non è più una mera questione di *file system*:
        - *network namespaces*, si garantisce l'accesso solo a un subset di interfacce di rete;
        - *pid namespaces*, si limitano le interazioni a un subset di processi.

  Garantita l'*isolation*.

- **Ruolo dei *control groups***.
  I control groups garantiscono un maggiore controllo sulle quantità di risorse assegnate.
    - Se usiamo solo i *namespaces* il processo potrebbe abusare delle risorse di sistema: occupazione elevata della memoria, uso della CPU per un tempo elevato, occupazione significativa della *network bandwidth*.
    - Si indica per un gruppo di processi il set di risorse che sono rese disponibili ponendo dei limiti al loro utilizzo (*resource usage*). Si garantisce che i limiti stabiliti non vengano violati.
        - Limite all'occupazione della memoria RAM
        - Utilizzazione della CPU
        - ….
    - Caratteristiche:
        - Sono creati dall'utente root
        - Ogni processo appartiene a un control group.
        - Tutti i processi figli creati da processi del control group ereditano l'appartenenza a quel control group.