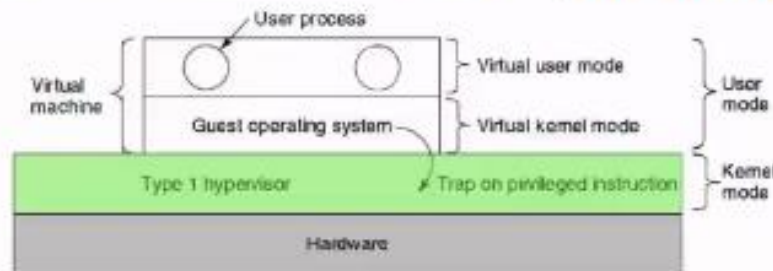


Virtualizzazione

- Nel 2005 Intel e AMD introducono la virtualizzazione sulle loro CPU e rendono possibile il monitor delle macchine virtuali tipo 1.
- Sulle CPU Intel Core 2 la tecnologia è chiamata **Virtualization Technology** (o **VT**) mentre sulle macchine AMD Pacific CPU è denominata **Secure Virtual Machine** (o **SVM**).
- Sebbene entrambe si ispirano al funzionamento dell'IBM/370 hanno delle piccole differenze: le macchine virtuali sono eseguite all'interno di contenitori fintantoché il SO guest non provoca un'eccezione che genera una trap per l'hypervisor.

Hypervisor di tipo 1

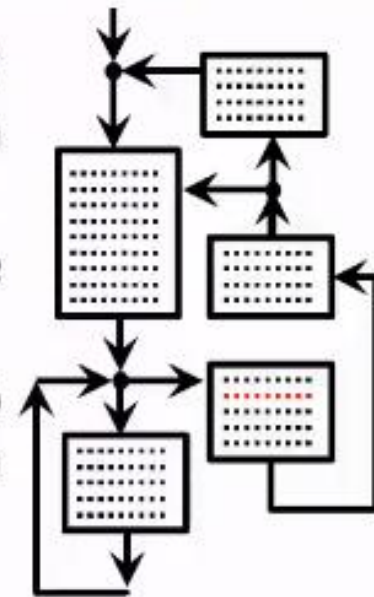
- La macchina virtuale è eseguita come un processo utente in modalità utente, non può eseguire istruzioni sensibili anche se pensa di essere in modalità kernel (**modalità virtuale del kernel**).




- Quando il SO guest esegue una istruzione sensibile:
 - Se la CPU non ha la VT la virtualizzazione non è possibile;
 - Altrimenti avviene una trap nel kernel e l'hypervisor può vedere se l'istruzione è stata inviata da:
 - una VM del SO guest, in questo caso la esegue.
 - un programma utente, in questo caso emula il comportamento dell'hardware reale.

Hypervisor di tipo 2

- **VMware** è un hypervisor di tipo 2 ed è eseguito come programma utente su un qualsiasi SO host.
- Quando si avvia per la prima volta, si comporta come un computer senza SO e cerca di installare il SO guest nel suo disco virtuale.
- Per eseguire un programma si utilizza una tecnica nota come **traduzione binaria**:
- esegue la scansione del codice alla ricerca dei **blocchi base** cioè quei blocchi che terminano con istruzioni di cambio flusso (es. JMP, CALL, trap,...).
- Ricerca nei blocchi le istruzioni sensibili e le traduce con una procedura **VMware**.
- Il blocco è messo nella cache di **VMware** e può essere eseguito alla velocità della macchina fisica (istruzioni tradotte a parte).



Confronto tra hypervisor

- Negli hypervisor tipo 2 tutte le istruzioni sensibili sono sostituite da chiamate a procedure che ne emulano il comportamento. Il SO guest non invierà mai nessuna istruzione sensibile alla macchina fisica.
- Le performance delle CPU con VT sono enormemente superiori rispetto a quelle senza VT ?
- Purtroppo l'approccio "trap-and-emulate" adottato dagli hardware VT genera troppi trap ed un eccessivo overhead di gestione, mentre la traduzione delle istruzioni sensibili è più efficiente.
 alcuni hypervisor di tipo 1 effettuano la traduzione binaria (anche se non serve) comportandosi come quelli di tipo 2.

Paravirtualizzazione

- Gli hypervisor tipo 1 e 2 funzionano senza modifiche al SO guest, ma con performance non eccellenti.
- Un diverso approccio prevede la modifica del codice sorgente del SO guest: invece di eseguire istruzioni sensibili si effettuano chiamate di procedure definite dall'hypervisor.
- Quindi l'hypervisor definisce una interfaccia, cioè delle **API (Application Program Interface)**, che i sistemi operativi guest possono attivare.
- Questo trasforma di fatto l'hypervisor in un microkernel e il SO guest modificato viene detto **paravirtualizzato**.
- Le performance ovviamente migliorano poiché le trap si trasformano in system call.

Conclusioni

- Affrontate le problematiche dello scheduling nelle architetture multiprocessori.
- Introdotto le architetture di multicomputer.
- Approfondite le problematiche dei sistemi operativi nei multicomputer.
- Analizzato il concetto di macchina virtuale.
- Approfondito il funzionamento degli attuali hypervisor.

Multiprocessori NUMA

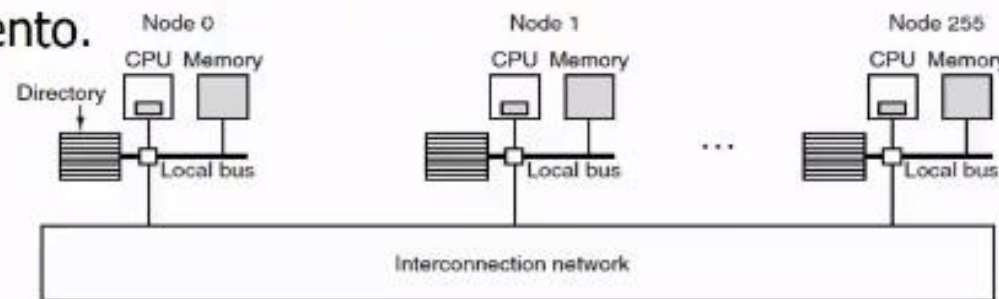
- Con i processori UMA a singolo bus si può arrivare a connettere insieme fino a 12 CPU, con una rete di interconnessione, a causa dei costi dell'hardware, si può arrivare a connettere insieme meno di 100 CPU.
- Per eccedere questo valore occorre introdurre una nuova architettura in cui si rinuncia a qualcosa: che i tempi di accessi della memoria siano uniformi.
- In un processore NUMA il tempo di accesso ai moduli di memoria locale è minore rispetto a quello dei moduli remoti.
- Tutti i programmi che girano su multiprocessori UMA continuano a girare sulle macchine NUMA, ma naturalmente hanno performance degradate.

Multiprocessori NUMA

- Le macchine NUMA hanno tre caratteristiche chiavi:
 - 1) C'è un unico spazio di indirizzamento visibile a tutte le CPU.
 - 2) L'accesso alla memoria remota è attraverso istruzioni di LOAD e STORE.
 - 3) L'accesso alla memoria remota è più lento dell'accesso rispetto alla memoria locale.
- Ci sono due tipi di macchine NUMA
 - 1) **No Cache** - Nonuniform Memory Access (NC-NUMA).
 - 2) **Cache Coherent** - Nonuniform Memory Access (CC-NUMA).

Multiprocessori NUMA

- L'approccio più comune utilizzato per costruire grandi multiprocessori CC-NUMA è il multiprocessore basato sulle directory (**directory-based multiprocessor**).
- L'idea di fondo è di mantenere una base dati delle linee di memoria ed il loro stato. Quando è referenziata una linea di cache, viene interrogata la base dati per sapere dove si trova e se è "pulita" o "sporca".
- Poiché ogni istruzione che accede alla memoria interroga questo database, occorre che sia gestito su un hardware specifico velocissimo.
- L'unione di tutte le memorie dei nodi costituisce l'intero spazio di indirizzamento.



Tipi di sistemi operativi multiprocessore

- Ci sono tre differenti approcci per i sistemi operativi multiprocessori:

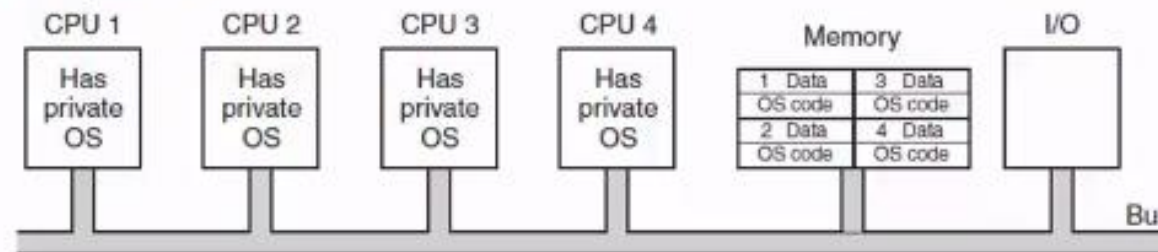
1) Ogni CPU ha un proprio sistema operativo

2) Multiprocessori master-slave

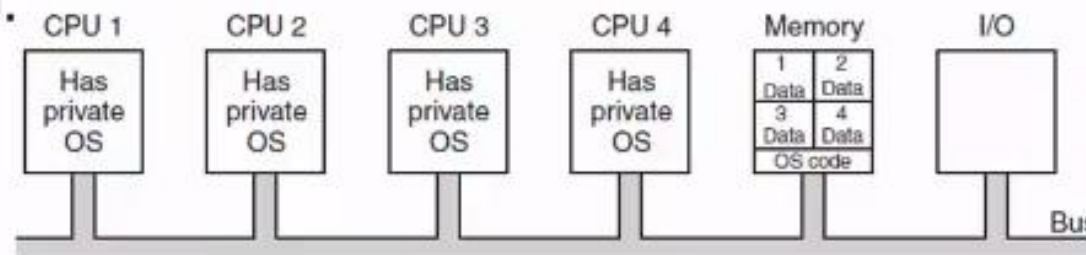
3) Multiprocessori simmetrici

Ogni CPU ha un proprio sistema operativo

- La soluzione più semplice è di **dividere staticamente** la memoria in tante partizioni quante sono le CPU.
- Ogni CPU ha un proprio sistema operativo e una a propria memoria privata.



- Le CPU operano con computer indipendenti, una possibile ottimizzazione è attraverso la condivisione del sistema operativo tra CPU.

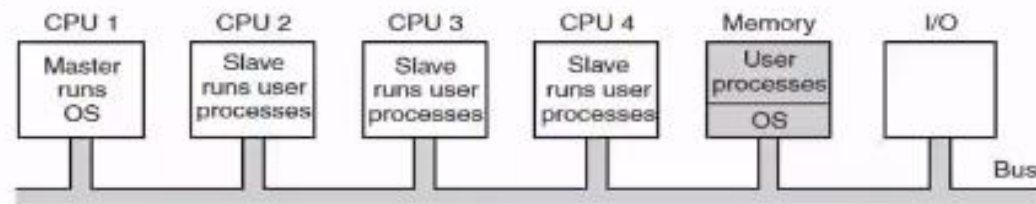


Ogni CPU ha un proprio sistema operativo

- Problematiche:
 - 1) Quando un processo esegue una system call, questa è gestita dalla CPU che l'ha lanciato utilizzando le strutture dati presenti nelle tabelle di quel sistema operativo.
 - 2) Non c'è condivisione di processi, ogni CPU gestisce indipendentemente i propri processi. Quindi non è possibile il bilanciamento del carico (una CPU potrebbe essere scarica mentre un'altra satura).
 - 3) Sistemi operativi indipendenti non condividono pagine di memoria. Una CPU può eseguire molto swap a causa della carenza di spazio libero mentre un'altra può disporre di pagine libere.
 - 4) Se un sistema operativo mantiene un buffer cache dei blocchi del disco usati recentemente indipendentemente dagli altri questo può condurre a letture inconsistenti (causate da letture sporche nei vari buffer).

Multiprocessori master-slave

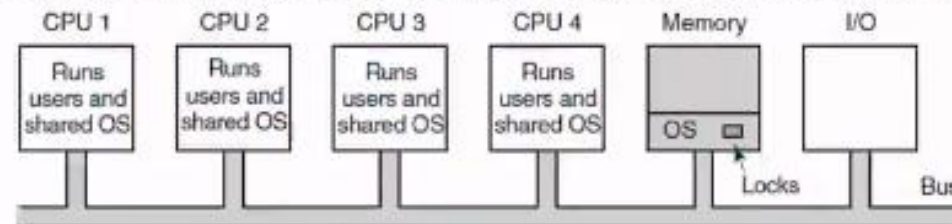
- Il sistema operativo e le sue tabelle è presente solo sulla CPU 1 (**master**). Tutti le *system call* sono redirette sulla CPU 1 per essere processate



- Il modello master-slave resolve la maggior parte dei problemi del modello precedente:
 - C'è una singola struttura dati che tiene traccia dei processi pronti.
 - La CPU master può bilanciare il carico sulle varie CPU ma può anche diventarne il collo di bottiglia.
 - Le pagine sono allocate dinamicamente tra i processi e c'è una sola buffer cache che evita le inconsistenze.
- Proprio perché la CPU master può diventare critica, il modello funziona con un numero ridotto di CPU.

Multiprocessori simmetrici

- Il **MultiProcessore Simmetrico (SMP)** elimina l'asimmetria del modello master-slave: ogni CPU può diventare master.
- Una copia del sistema operativo è in memoria e ciascuna CPU può eseguirla. Quando si fa una system call, la CPU che riceve la chiamata esegue un trap nel kernel e processa la richiesta.



- Questo modello bilancia processi e memoria dinamicamente.
- Elimina il collo di bottiglia e introduce nuovi problemi:
 - La sincronizzazione delle CPU dovuta alla condivisione del SO.
 - Evitare i deadlock (potrebbe essere impossibile risolverli)
 - La gestione è critica perché dipende dall'esperienza del programmatore nel contesto del sistema.

La sincronizzazione dei multiprocessori

- Le CPU in un multiprocessore hanno bisogno di sincronizzarsi: le regioni critiche del kernel e le tabelle devono essere protette da mutex.
- Se si disabilitano gli interrupt di una CPU, le altre CPU continuano a funzionare normalmente e possono accedere ad una regione critica.
- In un mono processore l'istruzione TSL (Test e Set Lock), impiegata nei mutex, permette ad una parola di memoria di essere controllata e impostata in una sola operazione indivisibile.
- Con più CPU, l'istruzione TSL deve prima bloccare il bus, evitando l'accesso di altre CPU, poi accedere alla memoria e quindi sbloccare il bus.

La sincronizzazione dei multiprocessori

- Se la TSL è realizzata correttamente, la CPU che fa richiesta del bus cerca di verificarne la disponibilità in un ciclo rapidissimo in modo da non rallentare le altre CPU (**spin lock**).
- Un approccio alternativo è lo switch dei processi, una CPU invece che attendere (spinning) passa il controllo ad un altro processo
- Non esiste la soluzione ideale, un approccio ibrido potrebbe essere di attendere per un certo tempo e poi fare un cambio di processo oppure memorizzare i tempi medi di attesa e vedere se il tempo attuale rispetta i tempi medi o è fuori norma (a meno di un certo errore)

Conclusioni

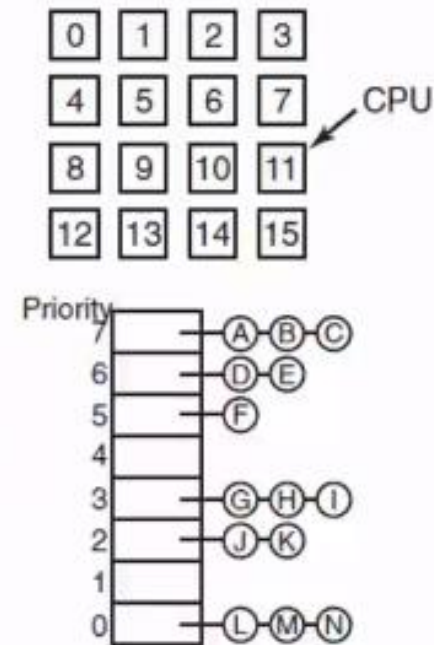
- È stata analizzata la modalità con la quale si realizza il parallelismo all'interno di un singolo chip.
- Introdotte le diverse architetture di calcolo parallelo.
- Studiati i diversi hardware dei multiprocessori.
- Introdotto i tipi di sistemi operativi utilizzati dai multiprocessori.

Scheduling nei multiprocessori

- Su un monoprocesore lo scheduling è mono-dimensionale:
 - L'unica domanda da porsi è: "Qual è il prossimo thread da mandare in esecuzione?"
- Su un multiprocessore, lo scheduling ha è bidimensionale:
 - Lo scheduler deve decider quale thread mandare in esecuzione e su quale CPU.
- Un altro fattore di complicazione è che in alcuni sistemi non tutti i thread sono correlati o lo sono in gruppi
- Esistono vari algoritmi di scheduling per ciascuna situazione:
 - 1) **Timesharing.**
 - 2) **Space sharing.**
 - 3) **Gang scheduling.**

Scheduling a condivisione di tempo (Timesharing)

- Lo scheduling a condivisione di tempo (**Timesharing**) è utilizzato quando i thread sono indipendenti.
- L'algoritmo utilizza un vettore di liste di thread (tutti nello stato "ready") a diverse priorità di esecuzione.
- Il fatto di avere un'unica struttura dati utilizzata da tutte le CPU ripartisce il tempo tra le CPU come se fossero in un sistema monoprocesso.



Scheduling a condivisione di tempo (Timesharing)

- Supponiamo che un thread termini il suo quantum di tempo mentre mantiene uno spin lock, le altre CPU dovranno attendere il rilascio della risorsa che avverrà quando il thread tornerà attivo e rilascerà il lock.
 - In alcuni sistemi quando ciò accade il thread, che esegue un lock, attiva un flag speciale di processo che permette allo scheduler di non bloccare il thread ma di concedergli un extra-quantum (**smart scheduling**).
- Alcuni multiprocessori utilizzano lo scheduling per affinità (**affinity scheduling**) cioè tentano di far eseguire lo stesso thread alla medesima CPU
 - La CPU che potrebbe avere nella propria cache dei blocchi già caricati e questo migliora di molto le performace!

Scheduling a condivisione di tempo (Timesharing)

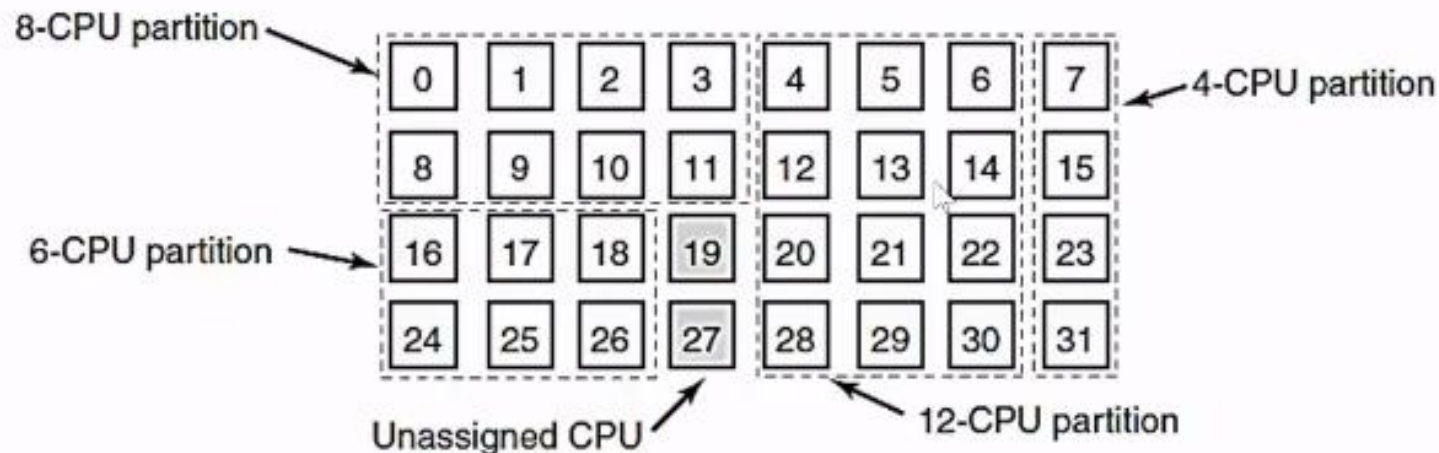
- In questi casi è conveniente utilizzare uno **scheduling a due livelli**:
 - 1) Quando è creato un thread viene assegnato alla CPU più scarica (top-level scheduling).
 - 2) Lo scheduling reale è fatto separatamente da ogni CPU (bottom-level scheduling) utilizzando l'affinità della cache e le priorità del thread.
- Lo scheduling a due livelli presenta vari vantaggi:
 - 1) distribuisce il carico in modo equo fra le CPU disponibili.
 - 2) l'utilizzo dell'affinità della cache migliora le performance.
 - 3) Si minimizzano le dispute sulle liste di thread poiché ogni CPU tenta di riusare i propri thread.

Scheduling a condivisione di spazio (space sharing)

- Se accade che i thread sono interrelati si può utilizzare lo scheduling a condivisione di spazio (**space sharing**).
- Lo scheduling di molti thread nello stesso tempo su diverse CPU è chiamato **space sharing scheduling**.
- Supponendo di voler creare con un intero gruppo di thread correlati, in un solo colpo. Lo scheduler controlla se ci sono tante CPU libere quanti sono i thread del gruppo, se:
 - accade, ad ogni thread è assegnata una CPU dedicata ed è avviato.
 - Non accade, nessun thread è avviato fintantoché non ci sono CPU disponibili.

Scheduling a condivisione di spazio (space sharing)

- Ad ogni istante di tempo, l'insieme delle CPU è partizionato staticamente, ciascun gruppo delle quali esegue il gruppo di thread correlati.



Gang scheduling

- Un vantaggio evidente dello scheduling a condivisione di spazio è l'eliminazione della multiprogrammazione che elimina anche l'overhead dovuto agli scambi di contesto.
- Uno svantaggio è il tempo sprecato quando si blocca una CPU perché non c'è nulla da fare .
- Conseguentemente si sono cercati algoritmi che tentassero lo scheduling in tempo e spazio.
- Una soluzione alternativa è il **gang scheduling** che si compone di tre passaggi:
 - 1) I gruppi di thread correlati sono schedulati come una unica squadra (o **gang**).
 - 2) Tutti i membri della gang girano simultaneamente su differenti CPU in timesharing.
 - 3) Tutti i thread della gang iniziano e terminano la loro porzione di tempo contemporaneamente

Gang scheduling

- Il tempo è diviso in *quantum* distinti e le CPU sono schedulate in sincronia
- All'inizio di ciascun nuovo *quantum* tutte le CPU sono rischedulate.
- Se un thread si blocca, la sua CPU rimane idle fino allo scadere del *quantum*.

		CPU					
		0	1	2	3	4	5
Time slot	0	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
	1	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
	2	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀
	3	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆
	4	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
	5	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
	6	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀
	7	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆

Multicomputer

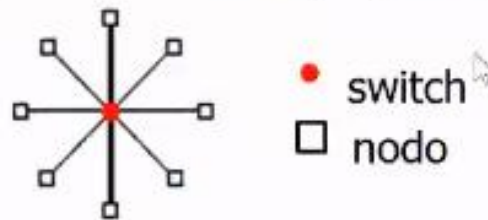
- I multiprocessori offrono un modello semplice di comunicazione: tutte le CPU condividono una memoria comune.
- I thread possono scrivere messaggi in memoria che poi possono, a loro volta, essere letti da altri thread.
- La sincronizzazione può essere fatta utilizzando mutex, semafori o monitor.
- I multiprocessori di grandi dimensioni sono difficili da costruire e perciò risultano costosi.
- Per ovviare a questi problemi sono nati i **multicomputer** che sono fortemente accoppiati ma non condividono memoria.
- Ogni computer ha una propria memoria.
- Questi sistemi sono anche chiamati **Cluster di Computers** or **Cluster di Workstations** (COW).

Multicomputer

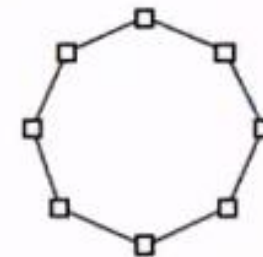
- I multicomputer sono facili da costruire perché il componente base è un PC con una scheda di rete con alte performance.
- Il segreto di ottenere alte performance in un multicomputer è nel progetto della rete di interconnessione e delle schede di rete.
- I messaggi sono spediti in un tempo nell'ordine dei μsec , mille volte di meno rispetto ad un accesso in memoria (ordine dei ns).
- Il progetto è quindi più semplice da realizzare ed allo stesso tempo economico.

Hardware dei multicomputer

- Il nodo base di un multicomputer consiste di una CPU, una memoria e una interfaccia di rete (talvolta anche un hard disk).
- Ogni nodo ha una interfaccia di rete con uno o due cavi (o fibra) che lo connette agli altri nodi oppure agli switch.
- In un piccolo sistema, ci potrebbe essere uno switch attraverso il quale sono connessi tutti i nodi (**topologia a stella**):

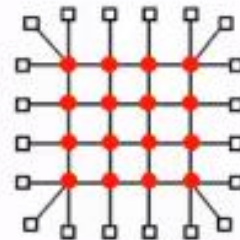


- Un'altra rete di interconnessione è la topologia ad anello: ogni nodo è connesso ad altri due nodi in ordine per formare un anello circolare (non sono necessary gli switch!).

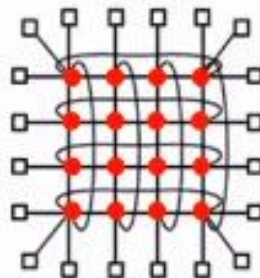


Hardware dei multicomputer

- La griglia (**grid**) o maglia (**mesh**) è una struttura bidimensionale utilizzata in molti sistemi complessi. La sua forma regolare la rende altamente scalabile. Il percorso più lungo tra due nodi, **diametro**, aumenta come la radice quadrata del numero dei nodi.



- Il **doppio toro** è una variante della griglia con i nodi estremi che si congiungono, è più tollerante ai guasti della maglia ma con un diametro più piccolo.



Hardware dei multicomputer

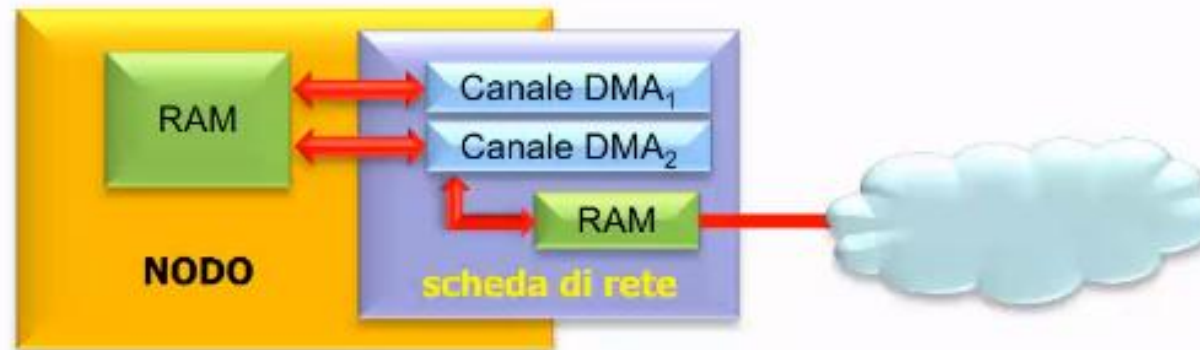
- Nei multicomputer sono usati due tipi di schemi di switching:
 - 1) **Store-and-forward packet switching** (connection-less).
 - 2) **Circuit switching** (connection oriented).

Store-and-forward packet switching

- Ogni messaggio è suddiviso in **pacchetti** che sono poi inseriti nella rete. Il pacchetto raggiunge il nodo destinatario attraverso delle politiche di instradamento che dipendono da vari fattori (i.e. traffico dati, priorità,...).
 - È flessibile ed efficiente ma ha il problema dell'incremento dei tempi di latenza lungo la rete di interconnessione.

Interfacce di rete

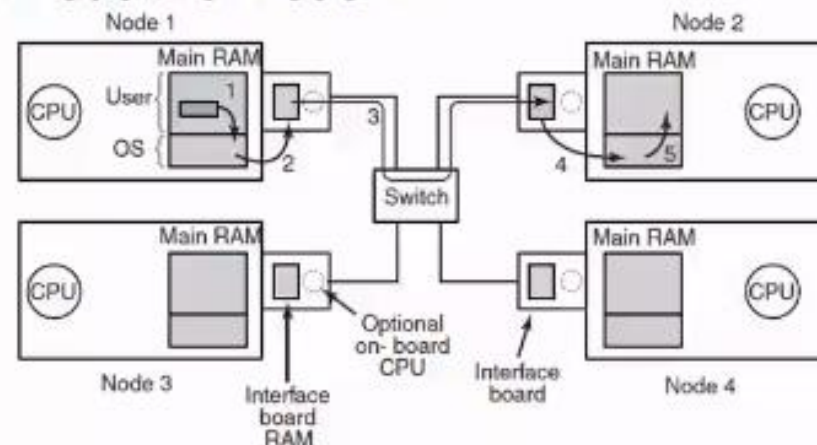
- In un multicomputer tutti i nodi hanno (almeno) una scheda di rete che permette di collegare il nodo alla rete di interconnessione, l'interfaccia contiene una RAM per memorizzare i pacchetti che entrano ed escono dal nodo.



- La scheda d'interfaccia può avere uno o più canali DMA oppure una CPU completa (**processori di rete**).
- I canali DMA possono copiare i pacchetti tra la scheda d'interfaccia e la RAM principale ad alta velocità, richiedendo i trasferimenti dei blocchi sul bus di sistema.



Software di comunicazione a basso livello

- Il principale nemico delle prestazioni dei multicomputer è eccessiva copia di pacchetti.
- Nell'esempio di sotto sono necessarie 5 copie prima di spedire il pacchetto dal nodo 1 al nodo 2.



- Per evitare questo problema molti multicomputer mappano la scheda d'interfaccia nello spazio utente permettendo così al processo utente di inserire i dati direttamente nella scheda di rete (senza coinvolgere il kernel)

Software di comunicazione a basso livello

- Questa soluzione pone due problematiche:
 - 1) La competizione dei processi concorrenti sullo stesso nodo che vogliono spedire pacchetti.
 meglio un solo processo per nodo.
 - 2) La condivisione della scheda di rete tra il kernel, che magari vuole accedere ad un file system remoto, e il processo utente.
 meglio utilizzare due schede di rete per ciascuna funzione.

Software di comunicazione a livello utente

- I processi sulle diverse CPU di un multicomputer comunicano attraverso lo scambio di messaggi.
- Il sistema operativo fornisce le primitive che consentono ai processi utente di inviare (**send**) o ricevere messaggi (**receive**).

send (*destination*, &*message_pointer*);

Spedisce il messaggio indirizzato da *message_pointer* ad un processo identificato da *destination*.

receive (*address* , &*message_pointer*);

In un multicomputer statico il numero di CPU è noto a priori, quindi il campo *address* si compone di:

- 1) Identificativo della CPU.
- 2) Identificativo del processo o della porta sulla CPU selezionata.

Chiamate bloccanti e non bloccanti

- Le primitive **send()** e **receive()** possono essere **bloccanti** (chiamate sincrone) o **non bloccanti** (chiamate asincrone).
- Se una **send()** è **non bloccante**, essa restituisce il controllo al chiamante immediatamente dopo la sua esecuzione e prima che il messaggio venga realmente spedito. Nel caso contrario il chiamante rimane in attesa finchè il messaggio non è inviato.
- La scelta tra bloccanti o non dipende dal progettista del sistema sebbene in pochi sistemi siano entrambe disponibili.
- All'evidente vantaggio di performance offerto dalle chiamate non bloccanti contrasta un **serio svantaggio**: il processo che ha spedito il messaggio (attraverso la **send()**) non può accedere al buffer fino a che il buffer non è vuotato e il messaggio spedito.
- In più non conosce quando la trasmissione sarà terminata.

Chiamate bloccanti e non bloccanti

- Esistono 3 possibili soluzioni:
 - 1) Il **kernel copia il messaggio in un buffer interno**, quindi ogni messaggio è copiato dallo spazio utente a quello del kernel (eccesso di copie).
 - 2) Al termine della spedizione il mittente riceve un **interrupt** in modo che possa riutilizzare il buffer (la gestione degli interrupt a livello utente è complessa e introduce ulteriori *race condition*).
 - 3) Fare una **copia del buffer** nel caso in cui venga riscritto da una nuova spedizione **modificando la sua pagina come read-only** finché il messaggio non è completamente inviato (eccesso di copie).

Chiamate bloccanti e non bloccanti

- Il processo mittente può eseguire quindi:
 - 1) Una spedizione bloccante e mantenere bloccata la CPU.
 - 2) Una spedizione non bloccante con copia (la CPU spreca tempo solo per eseguire la copia)
 - 3) Una spedizione non bloccante con interrupt (programmazione difficile ed insidiosa)
 - 4) Una spedizione non bloccante con copia su scrittura (la CPU spreca tempo anche per la copia di fine del processo oltre le scritture richieste)
- In un sistema multi-thread la prima scelta è la migliore: mentre il thread che esegue la **send()** è bloccato, gli altri continuano a lavorare.

Chiamate bloccanti e non bloccanti

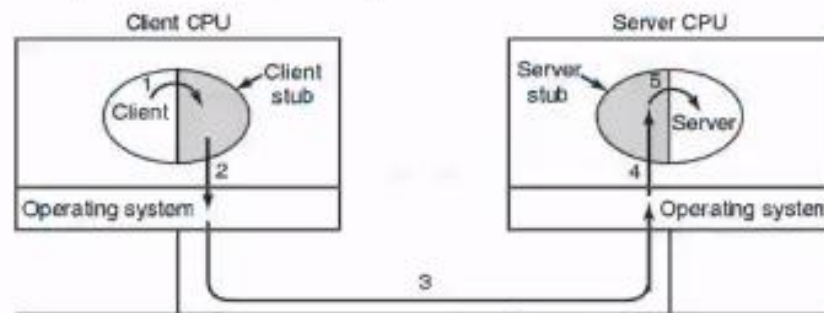
- Il processo destinatario può utilizzare una **receive()** non bloccante: indica semplicemente al kernel dov'è il buffer.
- L'arrivo di un messaggio può essere gestito:
 - tramite interrupt, ma sono difficili da programmare e molto lenti.
 - Richiamando una procedura **poll()** che restituisce se ci sono messaggi in attesa di essere letti
 - Attraverso la creazione automatica di un thread (chiamato appunto **thread pop-up**) che finito il suo compito muore spontaneamente.
 - Attraverso un interrupt che attiva nella ISR il codice di gestione (questo schema è una versione ibrida dei precedenti che sfrutta l'idea del thread pop-up senza creare alcun thread – migliorando così le performance - e si chiama **messaggi attivi**)

Remote Procedure Call (RPC)

- Sebbene il modello a scambio di messaggi è un sistema conveniente per in un sistema operativo multicomputer, tutte le comunicazioni, e quindi i programmi, utilizzano l'I/O.
- Un approccio differente è quello che consente ai programmi di chiamare procedure che si trovano su le altre CPU del multicomputer in modo indipendente dall'I/O
- Questa tecnica è detta **RPC (Remote Procedure Call)** ed è alla base del software per multicomputer.
- Per chiamare una procedura remota il programma client deve avere una piccola procedura chiamata **client stub**.
- Analogamente, il programma server è legato ad una procedura chiamata **server stub**.

Passaggi durante una RPC

- 1) Il programma client chiama il client stub (locale).
- 2) Il client stub confeziona un messaggio e chiede al sistema operativo di spedirlo (**marshaling**).
- 3) Il kernel spedisce il messaggio dal nodo client (sorgente) a quello server (destinazione).



- 4) Il kernel del nodo destinazione passa il messaggio al al server stub.
- 5) Il server stub chiama la procedura invocata dal nodo sorgente.

Insidie nell'uso delle RPC

- 1) Non è possibile utilizzare i puntatori come parametri poiché una chiamata a procedura remota risiede su un altro nodo (quindi un differente spazio di indirizzi).
- 2) Quando si utilizzano array come parametri è necessario specificare le dimensioni, a differenza di ciò che accade con i linguaggi debolmente tipizzati (es. il linguaggio C) in cui non è necessario farlo.
- 3) Non è sempre possibile dedurre il tipo dai parametri quindi attivare procedure/metodi che siano **polimorfe** come ad esempio la *printf* (che può avere più parametri).
- 4) Non è possibile utilizzare le variabili globali, in quanto non esiste un contesot comune.

Insidie nell'uso delle RPC

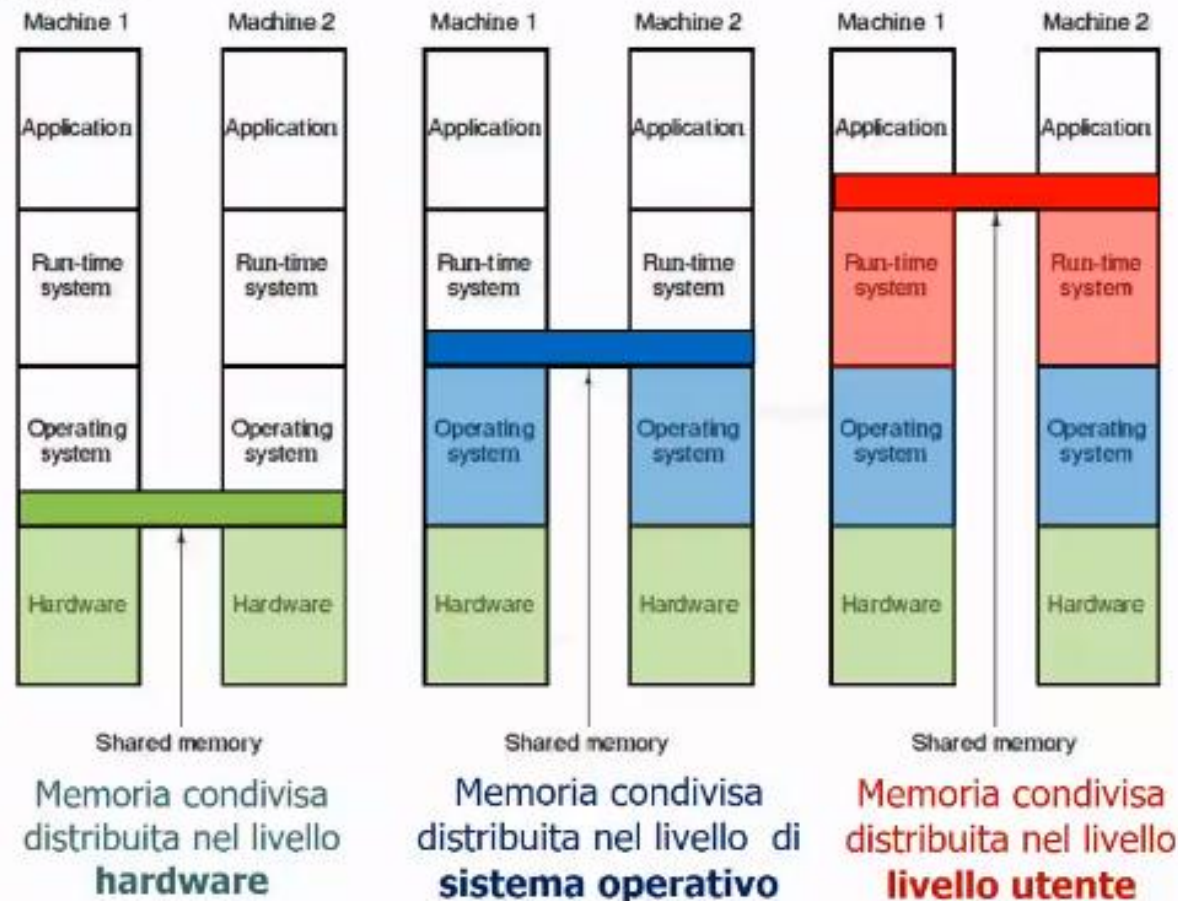
- 1) Non è possibile utilizzare i puntatori come parametri poiché una chiamata a procedura remota risiede su un altro nodo (quindi un differente spazio di indirizzi).
- 2) Quando si utilizzano array come parametri è necessario specificare le dimensioni, a differenza di ciò che accade con i linguaggi debolmente tipizzati (es. il linguaggio C) in cui non è necessario farlo.
- 3) Non è sempre possibile dedurre il tipo dai parametri quindi attivare procedure/metodi che siano **polimorfe** come ad esempio la *printf* (che può avere più parametri).
- 4) Non è possibile utilizzare le variabili globali, in quanto non esiste un contesto comune.

Distributed shared memory

- Molti programmatori preferiscono utilizzare il modello di memoria condivisa anche sui multicomputer.
- Attraverso la tecnica del **Distributed Shared Memory (DSM)** è possibile fornire una visione astratta della memoria.
- Ciascuna macchina ha la propria memoria virtuale
- Quando una CPU legge o scrive un dato su una pagina che non ha avviene una *trap* del SO:
 - Prima il SO localizza la pagina.
 - chiede poi l'unmap della pagina alla CPU che la possiede e se la fa spedire nella rete di interconnessione.
 - Una volta ricevuta la pagina, viene rimappata e l'istruzione che ha generato l'errore riavviata.

Memoria condivisa distribuita (DSM)

- La memoria condivisa distribuita si può collocare a diversi livelli dello stack:



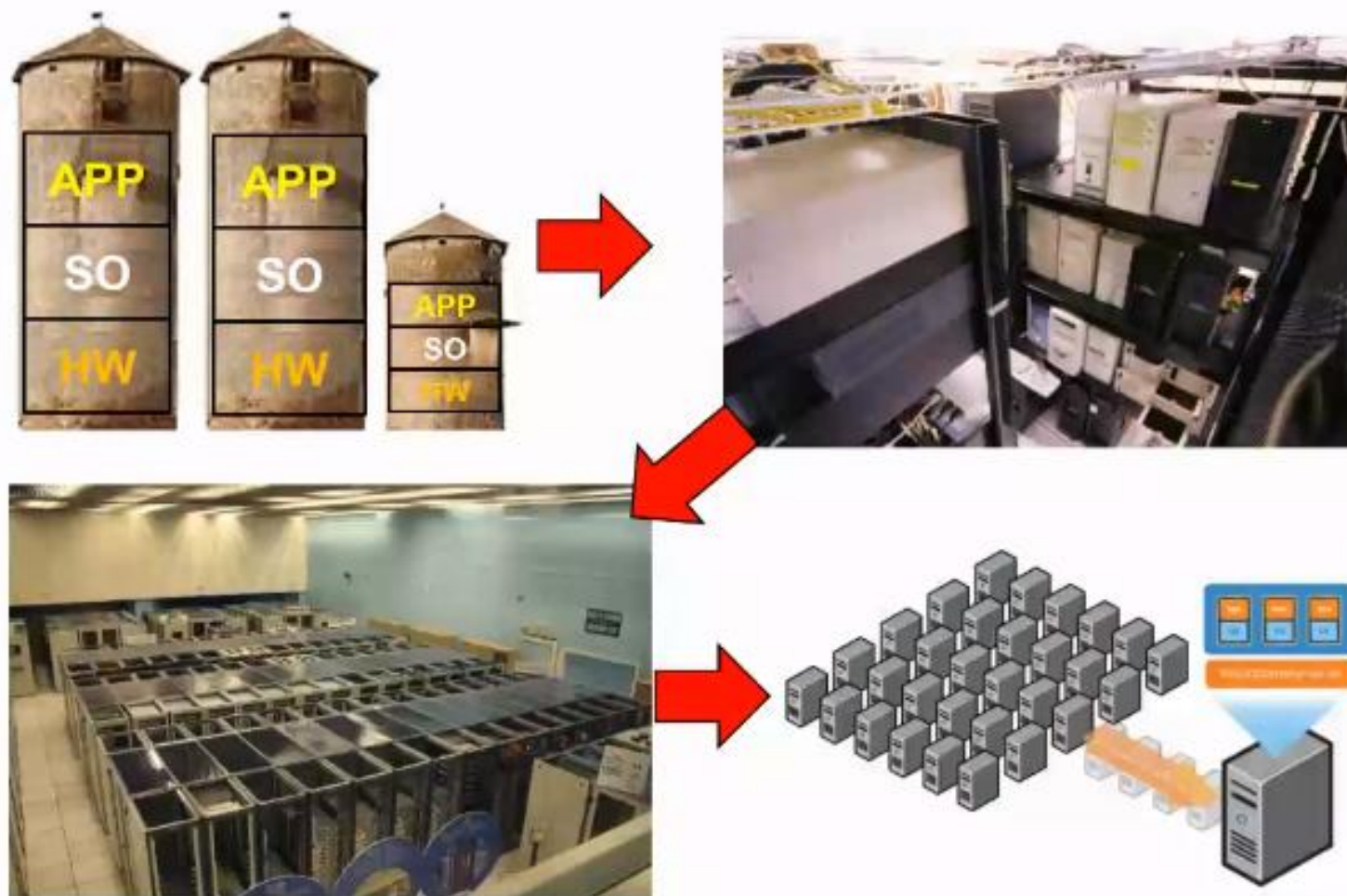
Scheduling sui multicomputer

- Su un multiprocessore i processi risiedono nella stessa memoria: tutti i processi sono potenzialmente candidati all'esecuzione.
- Su un multicomputer ogni nodo ha una propria memoria e un proprio insieme di processi.
- Lo scheduling su un multicomputer è simile a quello su un multiprocessore, ma non tutti gli algoritmi considerati per i multiprocessori sono applicabili ai multicomputer.
- L'algoritmo più semplice di tenere un unico elenco dei processi pronti, sui multiprocessori non funziona poiché ogni processo può essere eseguito solo sulla CPU corretta.
- Quando si crea un nuovo processo occorre scegliere dove eseguirlo anche in relazione al bilanciamento del carico del sistema.

Bilanciamento del carico

- Una volta che un processo è assegnato ad un nodo, funziona qualsiasi algoritmo di scheduling locale (a meno che sia utilizzato il gang scheduling).
- Diversamente dai multiprocessori, in cui tutti i processi condividono la stessa memoria e possono essere schedulati su qualsiasi CPU, nei multicomputer la scelta di quale processo spedire su un nodo è cruciale.
- Gli algoritmi e le euristiche per l'assegnazione di un processo al nodo sono conosciuti come gli **algoritmi di allocazione dei processi**.
 - Essi differiscono per i requisiti (CPU, memoria, traffico dati,...) e per l'obiettivo.

Virtualizzazione



Virtualizzazione

- La **virtualizzazione** (o tecnologia delle macchine virtuali) è una tecnologia che ha più di 40 anni!
- Un singolo computer di ospitare più macchine virtuali, ciascuna con un proprio sistema operativo.
- Un sistema virtualizzato mantiene l'affidabilità di un sistema multicomputer ad un costo ridotto ed una maggiore semplificazione della manutenibilità.
- Un guasto sul server che gestisce le macchine virtuali produce un danno catastrofico rispetto a quello di un nodo di un multicomputer.
 - La probabilità di guasti di natura hardware è enormemente più bassa rispetto a quelli di natura software!

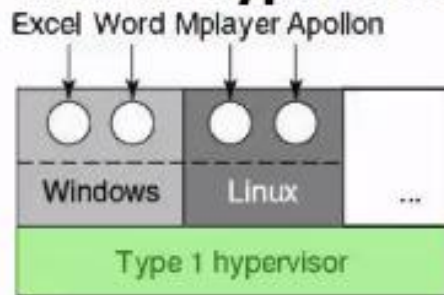


Vantaggi della virtualizzazione

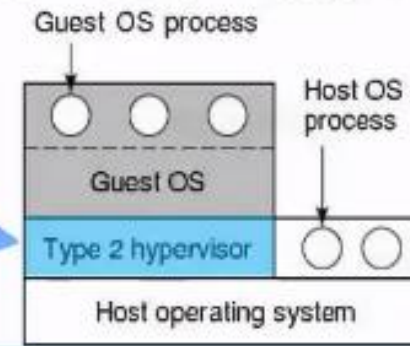
- Forte isolamento tra le macchine virtuali: un malfunzionamento su una macchina virtuale non inficia il comportamento delle altre.
- La riduzione delle macchine fisiche riduce lo spazio, il consumo di energia, la produzione di calore quindi l'energia per il raffreddamento.
- La creazione di *checkpoint* e la migrazione di macchine virtuali è più semplice rispetto ad un ambiente tradizionale.
- Si possono far girare applicazioni legacy su ambienti obsoleti che non funzionerebbero con gli attuali hardware.
- I programmatori possono effettuare il test delle applicazioni su differenti SO senza disporre di hardware fisici e SO.

Virtualizzazione

- Esistono due differenti approcci per il monitor delle macchine virtuali: **hypervisor** di **tipo 1** e di **tipo 2**.



L'hypervisor è nel SO della macchina reale (**SO host**) e gira in modo kernel, il suo compito è di supportare le macchine virtuali (**SO guest**) così come accade per i thread e i processi.



È un programma utente che interpreta le istruzioni delle VM e le traduce sul SO della macchina reale: sono analizzati frammenti di codice insieme con lo scopo di incrementarne le performance.

È la soluzione che permette la virtualizzazione alle CPU Intel senza tecnologia VT (es. 386, Pentium,...).

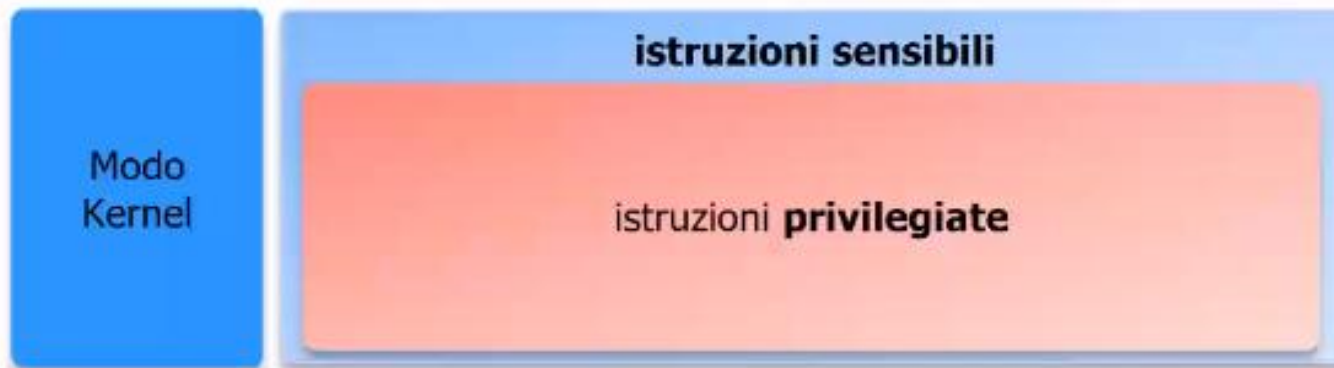
Virtualizzazione

- Ogni CPU possiede il proprio set di istruzioni



Virtualizzazione

- Una macchina è virtualizzabile (tipo 1) se e solo se tutte le istruzioni sensibili sono privilegiate



- A differenza del IBM/370 che è virtualizzabile (tipo 1), l'Intel 386 possiede delle istruzioni non privilegiate che sono ignorate se eseguite in modalità utente (come ad esempio la POPF).
- Quindi l'Intel 386 ed i suoi successori fino al 2005 (tecnologia VT) non possono essere virtualizzati con un hypervisor di tipo 1.