Lo zoo dei sistemi operativi

- Sistemi operativi per mainframe/server
- Sistemi operativi per personal computer
- Sistemi operativi per palmari/smart-phone
- Sistemi operativi per sistemi integrati (embedded)
- Sistemi operativi real-time

mario di raimondo

Gli ambiti in cui si può sviluppare un sistema operativo sono molteplici:

- ***MAINFRAME***: Si distinguono perchè gestiscono risorse più abbondanti, dati enormi. Calcolatori che hanno capacità di gestire risorse con ordini di grandezza molto grandi in cui lo storage, la capacità di calcolo e la memoria sono grandi. Sono storicamente legati a **Job non interattivi** dove il sistema elabora grandi quantità di transazioni senza l'intervento costante dell'utente.
- ***SERVER***: Evoluzione dei mainframe in ottica di rete. Si sottolinea la necessità di gestire un sistema con capacità sopra la media rispetto ad un personal computer. Devono gestire servizi specifici (web,mail,database) forniti a più utenti, si crea multi utenza. Tali sistemi devono garantire non solo la multi utenza ma anche la reattività, un sistema che deve essere interattivo e reattivo rispetto all'utente.
- ***PERSONAL COMPUTER:*** Il SO per eccellenza che punta sull'**interattività**. Deve massimizzare l'esperienza utente. Parliamo di sistemi **multiprogrammati** con

interfacce grafiche (GUI) complesse. Qui l'algoritmo di scheduling è fondamentale: deve dare l'illusione all'utente che tutti i processi (browser, musica, editor di testo) stiano girando perfettamente in contemporanea.

In Sistemi interattivi rientra sia server che personal computer, algoritmi scheduling decidee quale scegliere per ogni processo, influenza il tipo di servizio , ci sono algoritmi pensati per ambiti interattivi. Sono multiprogrammati , multi utente.

esempi passati spesso hanno so comuni con alcune eccezioni, caratteristica di poter gestire le proprie risorse pe piu utenti e quindi più processi, i sistemi devono funzionare a prescinder da tutto

- ***PALMARI/ SMARTPHONE***: Simili a PC ma hanno dei vincoli. Caratterizzati da sistema interattivo(processi con interfaccia grafica GUI) multiprogrammato quasi sempre mono utente. La gestione da parte del sistema operativo nello sfruttamento delle risorse è necessario soprattutto quando incontriamo problemi derivanti anche da app (ES: risparmio energetico). Utilizzano sistemi di sicurezza e permessi molto rigidi per isolare un'app da un'altra. L'interattività esasperata da touch richiede tempi di risposta più rapidi, necessita di " un ulteriore grado di interattività ".
- ***SISTEMI EMBEDDED (INTEGRATI)***: Hanno analogie con i personal computer. Si tratta di sistemi che troviamo a casa (ES: tv, router, stampanti ecc). La struttura interna è quella di un calcolatore con la presenza di più processi , troviamo un SO simile a quello dei computer, in cui la differenza principale è che sono sistemi SEMI-CHIUSI: non sono progettati per installare qualsiasi app (come un PC), ma per eseguire un **set di processi predefinito** e prevedibile. Hanno anche risorse limitate da gestire con pochi KB o MB di memoria.
- ***SISTEMI REAL-TIME***: Qui non conta solo *cosa* fa il sistema, ma *quando* lo fa. La correttezza del sistema dipende dal rispetto dei tempi (deadline). Hanno un compito ben specifico (es. macchinari industriali ,robot, catena di lavoro/ assemblaggio) è un sistema multiprogrammato ma hanno peculiarità in termini di esigenze, già il nome ne specifica esigenza (reattivtà in tempo reale), può eserre considerato un sistema vicino all'interattività ma in realtà si basa sulle tempistiche di reazione dei processi (es. sensore che monitora posizione) per cui deve scattare una reazione ben precisa. I sistemi interattivi non sempre permettono di reagire nei momenti in cui serve, se questo accade su sistemi

real-time non si soddisfano le garanzie premesse e si può andare incontro a problemi non accettabili, **non ci sono tolleranze rispetto alla reattività**.

possiamo fare un a distinzione tra due tipi di di sistemi real-time:

TIPO	DESCRIZIONE	ESEMPIO
HARD REAL-TIME	Le deadline e le condizioni/reazioni sono imprescindibili.	airbag, controllo reattori
SOFT REAL-TIME	Le scadenze sono importanti ma dei ritardi occasionali possono essere tollerati.	lettore multimediale

A differenza dei sistemi desktop che usano lo scheduling (il SO toglie forza alla CPU), molti sistemi Real-Time usano un **approccio collaborativo**: il processo usa la CPU per il tempo strettamente necessario e poi la rilascia volontariamente.

Struttura di un sistema operativo

- Alcune **possibili strutture** per un SO:
  - Monolitici
  - A livelli (o a strati)
  - Microkernel
  - A Moduli
  - Macchine virtuali
- categorie **con intersezioni** (sistemi ibridi);
- **tassonomia** non per forza completa o condivisa.

mario di raimondo

STRUTTURA DI UN SISTEMA OPERATIVO

Un progetto software con una serie di procedure e parti che lo compongono è complesso, I sistemi operativi moderni sono programmi software molto complessi e grossi che richiedono una struttura rigorosa.

A volte si parte da una base cercando di adattarla ai nuovi scopi. In generale dal punto di vista di sviluppo mantenibilità e possibilità di aggiungere funzionalità senza bug diventa complicato da gestire a lungo termine.

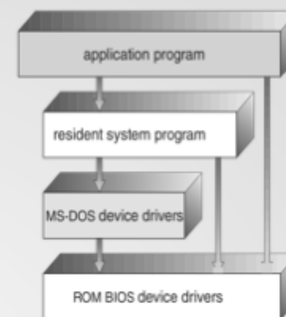
Alcune possibili strutture per un SO:

1. Monolitici
2. A livelli (o a strati)
3. Microkernel
4. A Moduli

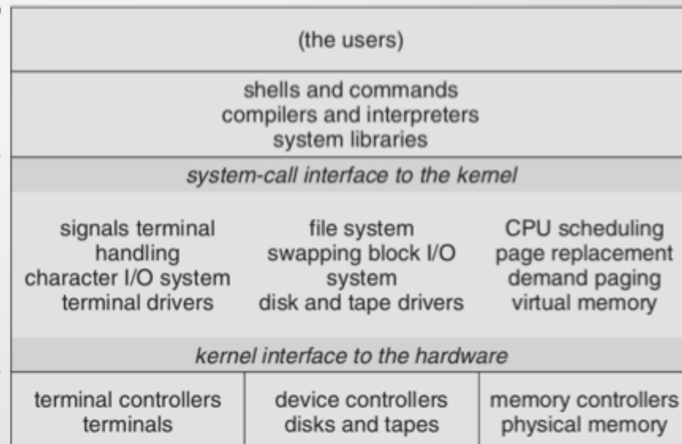
5. Macchine virtuali

Struttura monolitica

- **Monolitici:**
  - **nessun supporto** hardware;
    - problemi...;
    - esempi: MS-DOS, UNIX;
  - arrivò il supporto hardware alla **modalità kernel/utente**;
    - **unico kernel** con tutto dentro;
    - ogni componente può richiamare tutti gli altri
    - poco gestibile nel tempo.



kernel



mario di raimondo

SISTEMI MONOLITICI

Coincide con la struttura di primi SO,è un modello che privilegia le prestazioni a scapito dell'ordine e della manutenibilità. Possiamo intenderlo come un tutt'uno senza poter distinguere una sottoparte dalle altre.

Nei sistemi monolitici, il Kernel è un **unico, enorme file binario**. Dal punto di vista del codice:

- Non ci sono barriere: ogni funzione può chiamare qualsiasi altra funzione senza restrizioni.
- Le strutture dati (come la tabella dei processi o la gestione della memoria) sono condivise globalmente all'interno del kernel.

Questa mancanza di isolamento creava disorganizzazione sui primi sistemi (Un bug nel driver di una tastiera può corrompere la memoria del file system, causando un crash totale). La disorganizzazione portava a :

- **Difficoltà di Sviluppo:** Se si deve aggiungere una funzionalità, si rischia di rompere involontariamente qualcosa di apparentemente non correlato.

MS-DOS: Praticamente nessun supporto hardware per la separazione dei privilegi.

UNIX originale: Sebbene più strutturato, aveva "tutto dentro" il kernel (gestione file, driver, scheduling).

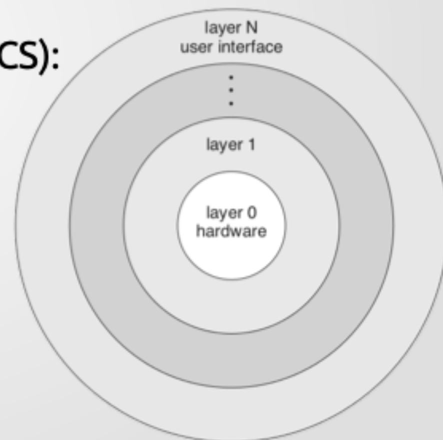
Inizialmente (es. MS-DOS), non c'era protezione hardware: un'applicazione poteva scrivere direttamente sui registri del disco. Con l'avvento di processori più avanzati, si è introdotta la separazione fisica:

- **User Mode:** Dove girano le app (browser, editor). Hanno privilegi limitati.
- **Kernel Mode:** Dove risiede il monolite. Ha controllo totale sull'hardware.
- **System Calls:** L'interfaccia delle *system-call* funge da unico "ponte" sicuro tra l'utente e il monolite.

PRO	CONTRO
Velocità: Non c'è overhead per passare messaggi tra componenti; le chiamate sono dirette.	**Instabilità:** Un errore in un punto qualsiasi fa crollare tutto il sistema.
Efficienza: Minor uso di memoria per la gestione della comunicazione interna.	**Rigidità:** Difficile da aggiornare o modificare senza ricompilare l'intero kernel.

Struttura a livelli (o a strati)

- Si utilizza una **gerarchia di livelli**;
- ogni livello implementa delle **funzionalità** impiegando quelle fornite da quello inferiore;
- **migliore progettazione**, più semplice da sviluppare e controllare (incapsulamento tipo OOP); suddivisione a livello progettuale;
- variante ad **anelli concentrici** (MULTICS):
  - **separazione forzata** dall'hardware;
- **problemi di prestazioni** dovuti alle chiamate nidificate e al relativo overhead.



mario di raimondo

STRUTTURA A LIVELLI (O A STRATI)

In questa struttura, il sistema operativo è visto come una serie di **macchine virtuali** sempre più sofisticate.

Un sistema a strati è un'astrazione in cui si lavora in ogni livello con delle esigenze ben precise. Usare gli strati permette di facilitare la fase di testing di ognuno di essi.

Un protocollo deve essere progettato con finalità precise come quella di risolvere problemi, deve basarsi anche sullo strato sottostante (ES: TCP si basa anche sullo strato sottostante). Perchè ogni livello nasconde la complessità di quello inferiore e offre servizi a quello superiore. (ES: un livello basso gestisce l'hardware fisico, ma lo strato superiore "vede" solo **CPU virtuali** (grazie al time-sharing). Lo strato superiore non sa *come* la CPU venga divisa, sa solo che ne ha una a disposizione.)

Creiamo una serie di strati dove il più in basso è quello che già conosciamo ovvero l' hardware, sopra troviamo SO che può essere formato da uno o piu strati. Gli strati interni a SO sono astrazioni che servono come servizio agli strati che gli stanno sopra, ma che devono basare i servizi sullo strato sottostante.

Questa suddivisione a strati permette di facilitare l'implementazione del SO.

Due vantaggi sono la modularità e il parallelismo:

- Puoi testare lo Strato 1 subito dopo averlo scritto. Se funziona, sai che eventuali bug futuri saranno nello Strato 2 o superiore.
- **Sviluppo in parallelo:** persone differenti possono lavorare su strati diversi, purché le interfacce (le "promesse" di cosa fa ogni strato) siano ben definite.

L'idea alla base del sistema a strati è che ogni livello offra un'interfaccia pulita a quello superiore, nascondendo i dettagli implementativi.

- **Esempio della Memoria Virtuale:** Lo strato della gestione memoria offre ai livelli superiori l'illusione di avere uno spazio di indirizzamento potenzialmente infinito.
- Se la RAM fisica finisce, il sistema sposta i dati meno usati sul disco (swapping). Gli strati superiori non "vedono" questo movimento; per loro, la memoria è semplicemente disponibile. Questa **trasparenza** permette di cambiare la tecnologia del disco (es. passare da HDD a SSD) senza dover riscrivere le applicazioni o gli strati più alti.

Non si possono disporre gli strati a piacimento; l'ordine è dettato dalle necessità operative. Lo strato del "Driver del Disco" deve trovarsi necessariamente **sotto** quello della memoria. Se un processo scrive su disco, la chiamata è spesso **bloccante**. Se lo strato di gestione della CPU (lo scheduler) fosse posizionato sopra quello del disco in modo errato, rischieremmo che l'intero sistema si fermi in attesa di un'operazione di I/O, rendendo impossibile gestire altri processi.

La struttura **ad anelli** è concettualmente simile a quella a strati, ma introduce una differenza fondamentale: la **protezione hardware**. Mentre nella stratificazione semplice l'ordine è solo organizzativo, qui l'architettura è rappresentata da cerchi concentrici dove l'hardware occupa l'anello centrale (il più protetto).

La differenza cruciale risiede nel momento in cui il codice viene eseguito:

- **Nei sistemi a strati classici:** L'organizzazione serve a progettare meglio il software, ma una volta in esecuzione, tutto il kernel gira spesso in un unico spazio di memoria. Se un bug colpisce uno strato, può corrompere i dati di tutti gli altri perché non c'è una barriera fisica: il sistema "crolla" proprio come nei sistemi monolitici.
- **Nel modello ad Anelli (es. MULTICS):** L'isolamento avviene a **runtime**. Ogni anello definisce un'"area di pertinenza" con privilegi decrescenti man mano che ci si allontana dal centro. Un bug in un anello esterno è confinato e non può accedere o danneggiare direttamente le strutture dati degli anelli più interni (più critici).

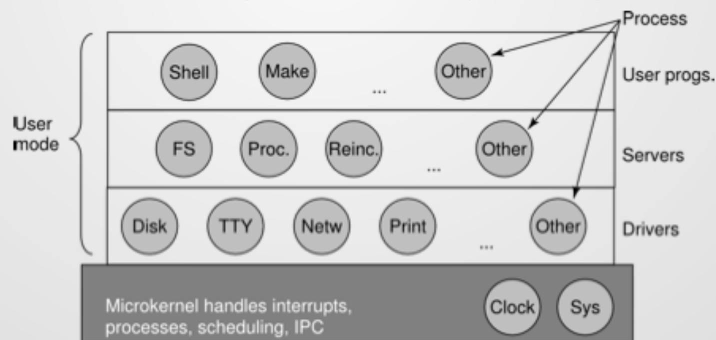
Per richiedere un servizio a un anello più interno (protetto), il processo deve generare una "trap" (un'eccezione controllata).Questo rende il sistema solido: è fisicamente impossibile per un pezzo di codice "saltare" senza un controllo preventivo dell'hardware.

Rimane il problema di definire esattamente l'ordine di precedenza.Ogni passaggio tra anelli richiede cicli di CPU per salvare lo stato, verificare i privilegi e cambiare i registri del processore. Questo **overhead** è molto più pesante rispetto a una semplice struttura a strati software, rendendo il sistema più lento.

Struttura a strati	Struttura ad anelli
Un bug può propagarsi a tutto il kernel.	Un bug viene isolato all'anello corrente.

Microkernel

- Uso di un **microkernel minimale** che si occupa dello scheduling, memoria e IPC;
- tutto il resto gestito da **moduli** (livello utente): filesystem, driver di dispositivi;
- comunicazione attraverso **messaggi**;
- miglior design (componenti piccoli) e migliore stabilità;
- esempi: MINIX 3, Mach, QNX, Mac OS (Darwin), Windows NT



mario di raimondo

MICROKERNEL

L'idea centrale è avere un **kernel più piccolo** rispetto alla concezione generale. Si mantiene nel "nucleo" solo ciò che è strettamente fondamentale per far funzionare la macchina.

Tirare fuori una funzionalità dal kernel significa spostarla all'esterno, trasformandola in **moduli o servizi**.

- Questi componenti sono isolati.
- Il loro codice e i loro servizi vengono eseguiti in **modalità utente** (user mode).
- All'interno del microkernel resta solo ciò che non si può assolutamente spostare, ovvero tutto ciò che lavora pesantemente a basso livello con l'hardware e che sta alla base dell'esistenza stessa dei processi.

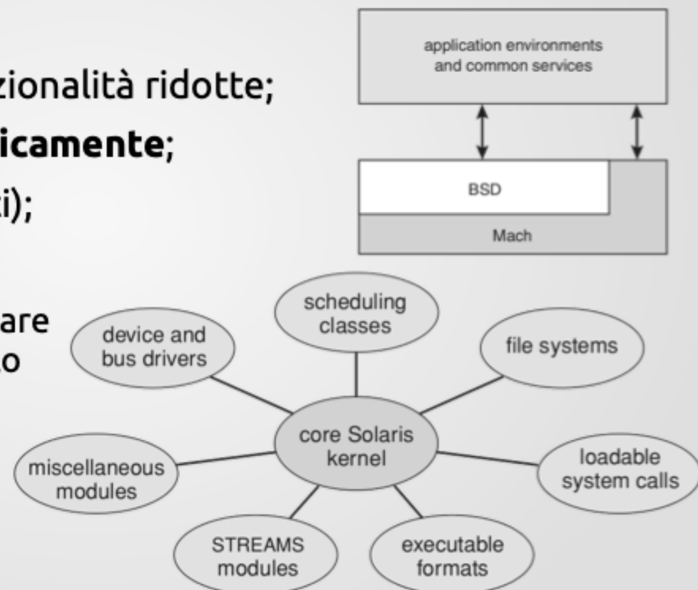
Interprocess Communication

Poiché i servizi sono frammentati e isolati, il compito principale del microkernel diventa la **comunicazione**. Il kernel fornisce un'infrastruttura di meccanismi che permette ai processi di comunicare tra loro e con il kernel stesso. Questo avviene tramite **richieste tra pari** o **messaggi** che vanno al kernel, permettendo a componenti separati di collaborare come se fossero un unico sistema.

Vantaggi	Contro
Tutto ciò che gira nel microkernel è "micro", quindi è molto più semplice da scrivere e gestire rispetto a un kernel normale (proprio come è più facile scrivere un singolo livello in un sistema a strati).	Far passare tutte le comunicazioni tra il microkernel e i vari servizi attraverso lo **scambio di messaggi** rappresenta una forma di **overhead**.
I singoli servizi sono istanziati dentro processi isolati. Ad esempio, si possono incapsulare i **driver** in processi utente separati. Se un driver fallisce, il resto del sistema rimane protetto.	Questo continuo scambio di dati rende il sistema meno scattante rispetto a un sistema monolitico dove tutto avviene internamente senza comunicazione esterna.

Struttura a Moduli

- Idea della **programmazione OO** applicata al kernel;
- **moduli** che implementano un qualche aspetto specifico;
  - filesystem, driver,...
- **kernel principale** a funzionalità ridotte;
- moduli **caricabili dinamicamente**;
- design pulito (ad oggetti);
- **efficiente**:
  - ogni modulo può invocare qualunque altro modulo direttamente;
  - niente messaggi;
- esempi: Solaris, Linux, Mac OS (ibrido).



mario di raimondo

se l'idea alla base di MINIX è quella del microkernel, un'altra idea usata da linux, solaris o macOS è quella della **struttura a moduli**. l'esempio più diffuso è linux.

Scrivere un Sistema Operativo in termini di **moduli**, alla base è simile al microkernel, ci sono parti inscindibili al kernel e poi una serie di funzionalità pensate in termini di moduli(molto vicino alla prog ad oggetti) ,questi moduli possono invocare i servizi offerti da altri moduli in modo libero ma organizzato vincolato da interfacce.

Un oggetto è qualcosa di simile ad un servizio: ha i suoi metodi, interfaccia con cui gli altri oggetti possono usare i servizi offerti

lo scheduler è l'algoritmo che sceglie quale istruzione devo effettuare; è a sua volta un modulo, posso avere tanti moduli quanti file system devo formattare.

Lo scheduler deve bilanciare obiettivi spesso in contrasto tra loro:

- **Massimizzare l'utilizzo della CPU:** Tenerla occupata il più possibile.
- **Equità (Fairness):** Evitare che un processo "rubi" tutta la potenza, lasciando gli altri a secco (fenomeno della *starvation*).
- **Tempo di risposta:** Garantire che il mouse e l'interfaccia reagiscano subito ai comandi dell'utente.
- **Throughput:** Completare il maggior numero di lavori nel minor tempo possibile.

Una delle caratteristiche fondamentali è caricamento on demand: ho un kernel minimale che manca di tante funzionalità e all’occorrenza metto le funzionalità che necessito. oppure posso caricare più scheduler o fare switch da uno all’altro.

Il cuore della differenza tra Microkernel e Moduli: **lo spazio di memoria e i privilegi.**

- Nel **Microkernel** (come Minix), i servizi (driver, file system) girano in **User Mode**. Per comunicare con il kernel devono usare il *Message Passing*, che è lento perché richiede continui cambi di contesto (context switch).
- Nel **Sistema a Moduli** (come Linux), i moduli vengono caricati e girano in **Kernel Mode**. Sono parte integrante del "corpo" del kernel.
- **Vantaggio:** La comunicazione tra moduli è una semplice chiamata a funzione (overhead zero).
- **Svantaggio:** un bug in un modulo ha privilegi totali e può tirare giù l'intero sistema (Kernel Panic). Se ce un bug in un modulo manda in crash in sistema. un driver (che è un modulo) se ha un bug scritto male manda in crash tutto il sistema. i sistemi linux sono molto piu stabili rispetto ad altri per questo. Crash sono abbastanza rari in linux.

Andiamo a sfruttare la programmazione modulare ad oggetti, i benefici del caricamento on demand e overhead 0. un modulo è perfettamente isolato e posso modificarlo senza dare problemi al resto del progetto.

il modulo viene caricato dall’esterno a causa di una richiesta, quindi questi moduli sono preesistenti caricati in fase di boot e usati quando vengono chiamati.

I sistemi moderni sono dei veri e propri **ibridi** che cercano di prendere il meglio da ogni mondo: la velocità del monolite, la pulizia del design ad oggetti e la sicurezza del microkernel.

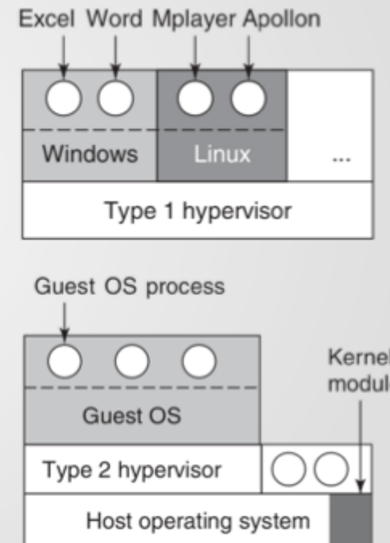
Linux nasce come kernel monolitico, ma la sua evoluzione lo ha reso estremamente sofisticato. Internamente, la sua forza è la **struttura a moduli**. Immagina il cuore di Linux come una base minima che può espandersi a caldo: se colleghi una nuova periferica, il kernel carica un "modulo" (un pezzetto di codice che gira con pieni privilegi hardware) senza dover riavviare. È un design simile alla **programmazione ad oggetti**: ogni modulo è isolato, ha la sua interfaccia e comunica con gli altri in modo efficiente (overhead zero).

Tuttavia, Linux "ruba" un'idea al microkernel per aumentare la stabilità: l'**esternalizzazione**. Alcune funzioni che un tempo stavano nel cuore del sistema oggi girano in **modalità utente**. Un esempio classico è la gestione della stampa o alcuni file system : se il driver della stampante va in crash, cade il servizio, ma il computer non si blocca. In questo senso, Linux si comporta "un po' come un microkernel".

macOS porta questo concetto ancora più in là. Il suo cuore è esplicitamente un ibrido. Utilizza una base microkernel (Mach) per gestire i compiti fondamentali, ma vi incolla sopra una parte monolitica per garantire le prestazioni. Come Linux, anche macOS usa i **moduli** (chiamati *Kernel Extensions* o *kexts*), che permettono di aggiungere funzionalità che girano con privilegi massimi, pur mantenendo una struttura interna molto ordinata.

Macchine virtuali

- L'estremizzazione del concetto di astrazione porta alla **virtualizzazione**;
- **Perchè?** Uso di più SO, VPS, isolamento dei servizi,...
- **Simulazione**, **paravirtualizzazione**.
- Viene tutto gestito dallo **Hypervisor**:
  - **Hypervisor di tipo 1**:
    - gira direttamente sull'hardware;
    - esempi: VMware ESX/ESXi, Microsoft Hyper-V hypervisor;
  - **Hypervisor di tipo 2**:
    - è un processo in un SO Host
    - esempi: VMware Workstation, VirtualBox.
- **Supporto hardware** per efficienza.



mario di raimondo

Macchine virtuali hanno analogie forti con le astrazioni, come nel caso delle CPU virtuali, un esempio di un qualcosa che non esiste ma viene implementato perche richiesto. Abbiamo memoria virtuale che completa il ruolo di una CPU virtuale. Se a questa astrazione includiamo pezzi via via, ottengo un vero e proprio calcolatore astratto e completo. Una macchina virtuale è un'astrazione. l'idea di creare queste macchine virtuali su un hardware reale è un astrazione completa che permette di far girare un software diverso in una macchina reale.

Non è un tema recente, nato insieme ai primi calcolatori, sono ritornate in auge da circa 10 anni e utilizzate per:

- istanziare una o più istanze virtuali, e dentro posso far girare un intero sistema operativo, mi permette un'istanza di linux su macchina virtuale ospitata in una macchina reale windows o viceversa.... le combinazioni sono varie.

PERCHE? Può essere un modo alternativo per usare più sistemi operativi contemporaneamente. All'occorrenza faccio partire quello che mi serve, uso

windows ma ho una macchina virtuale e lancio istanza di linux dentro windows.

Ci sono utilizzi più seri:

- per testare un programma su più sistemi operativi, è un modello di gestione dei servizi;
- sviluppo di sistemi operativi, gira tutto su macchine virtuali è più comodo;
- altri impieghi sul tema della sicurezza:

un’azienda che gestisce tutto dall’interno, ci sono delle esigenze: web server, file server, mail server ecc per far girare n servizi: o compro una macchina, installo SO e inizio a usarlo: non è detto sia il metodo migliore, infatti probabilmente non è la più sicura potrebbe esser facile attaccare il SO. Se alcuni dei servizi vengono esposti sulla rete esterna, andrei ad attaccare partendo da li.

Attaccare un servizio vuol dire che l’attaccante deve scoprire un bug che può sfruttare e porta ad interazioni con il sistema, o il web service crasha o l’attaccante cerca di prendere il controllo del servizio.

molti bug possono essere usati per prendere il comando del servizio, fanno un injection nel codice, sfruttano un bug. il web server gira sul server in questione, ad un certo punto il processo web server che dovrebbe eseguire solo il codice che ho generato, esegue anche codice che proviene dall’attaccante e quindi fa quello che vuole a suo favore. Ma se io attaccante prendo possesso di un web server, cerco di fare più danni possibile, ho spazio in base a quanto è stato bravo l’amministratore.

I servizi vengono eseguiti da utenti ordinari, cioe non amministratore di sistema perchè in ottica di attacco mi mette un minimo al sicuro. L’attaccante può fare danno nella misura di un semplice utente, ma è comunque un punto di accesso: può copiare tutti i file che il server forniva, permette di interagire con altri servizi che l’utente non può vedere o usufruire.

Potrebbe esserci anche un bug all’interno del kernel che può anche portare alla root escalation, se posso usufruire di un servizio di un utente, posso arrivare alla root: se sono di fronte ad un SO che ha un bug noto, posso usufruirne per attaccarlo e prendere tutti i servizi presenti sulla macchina.

Proprio per questo viene effettuato **isolamento dei servizi**, prendo macchine diverse e una fa web server, un'altra mail server ecc.... ma questo spreca tanta energia. Isolare servizi in macchine diverse, pur facendo root escalation, non può prendere possesso di tutti gli altri servizi, ma bisognava interagire con altri bug degli altri servizi.

Allora sfrutto l'isolamento cercando di non occupare tutto lo spazio, implemento più macchine virtuali su un unica macchina fisica, cosi istanzio tante macchine virtuali, collegandole tramite reti virtuali con latenza nulla, e potrei reimplementare modello di sicurezza ma su macchine virtuali. Hanno overhead, parte della RAM la perderei per questo, più sforzo configurativo.

Spesso simulare una macchina viene confuso con macchina virtuale.

La **virtualizzazione** viene ottenuta eseguendo il codice della macchina virtuale direttamente sulla CPU fisica.

Il codice della macchina virtuale LINUX vengono eseguite sulla CPU della macchina fisica, questo rappresenta un vincolo(software deve essere compatibile con CPU: arm arm, x86 ×86..) codice gira in modo nativo e a tutta velocità. La virtualizzazione diventa piu complicata quando si parla di periferiche di I/O.

Nella **simulazione** scenario diverso, il software dell'ambiente simulato usa istruzioni macchina non comprese dalla CPU.

Ci vuole un interprete, interpretazione del codice e aggiunge overhead e rallentamento, usa registri che in realtà non esitono.

una variante della virtualizzazione è la **paravirtualizzazione**, dove si viola uno dei principi della virtualizzazione, cioe che la macchina astratta deve essere compatibile e indistinguibile dalla macchina reale. Questo vincolo di far funzionare le cose in modo trasparente può costringere ad avere cose meno efficienti. Maggiori rallentamenti si hanno nel virtualizzare disco, scheda grafica, interfaccia di rete e qui la paravirtualizzazione permette di violare paradigma di astrazione perfetta dove crea ibrido. Esempio:

VirtualBox (sviluppato da Oracle) è un software di **virtualizzazione** gratuito e open source che ti permette di far girare un sistema operativo (es. Windows, Linux o macOS) dentro un altro sistema operativo, come se fosse una normale applicazione.

In termini tecnici, VirtualBox è un **Hypervisor di Tipo 2** (o *Hosted Hypervisor*).

Come si creano le macchine virtuali? usano Hypervisor due accezioni:

- **di tipo 1**: gira sull'hardware, non esiste SO intermedio che ospita le macchine virtuali, lo strato software gira sull'hypervisor di tipo 1. In pratica non esiste SO ospitante, queste erano le prime tipologie di SO virtuali. Si usa sui sistemi di alto livello: KVM (Kernel-based Virtual Machine) è un caso particolare e affascinante. È un modulo di Linux che trasforma il kernel Linux stesso in un Hypervisor di Tipo 1. Quindi sì, **Linux diventa l'hypervisor**. Viene considerato di Tipo 1 perché le VM girano insieme al kernel, gestite direttamente dalla CPU, senza passare per strati "utente" pesanti.
- **di tipo 2:** caso più diffuso, in cui ho hardware, SO ordinario e poi software installato che permette di creare macchine virtuali multiple. Non svolge tutto i suoi compiti in modalita utente, alcuni in modalita kernel. ha un piede nel kernel.

TIPO 2 è il caso che usiamo sui nostri PC (come VirtualBox o VMware Workstation). Qui abbiamo: **Hardware** → **SO Host** → **Hypervisor** → **VM**.

Osservazione "un piede nel kernel": Anche se installi VirtualBox come un normale programma (Modalità Utente), per poter funzionare deve installare dei **driver nel kernel** del sistema ospitante.

Perché? Perché un software in modalità utente non ha il permesso di gestire direttamente la CPU o la memoria RAM per isolare una VM. Ha bisogno di "un piede nel kernel" (moduli del kernel) per chiedere al processore di attivare le funzioni di virtualizzazione hardware.