Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

TITOLO ITALIANO

TITOLO INGLESE

NOME CANDIDATO

Relatore: *Relatore*
Correlatore: *Correlatore*

Anno Accademico 2014-2015

Nome Candidato: *Titolo italiano,* Corso di Laurea in Informatica, © Anno Accademico 2014-2015

INDICE

ELENCO DELLE FIGURE

"Inserire citazione"
— Inserire autore citazione

1

INTRODUZIONE

In questa introduzione sarà presente una prima parte che andrà a dare le conoscenze di base minime per comprendere cosa sia un sistema operativo e una piccola classificazione, dopodiché seguiranno la descrizione di alcuni concetti che sono fondamentali per comprendere il resto dell'elaborato.

1.1 COS'È UN SISTEMA OPERATIVO

Un sistema operativo (SO) è un insieme di software che gestisce le risorse hardware e software di un sistema di elaborazione fornendo servizi agli applicativi utente.
In un computer quindi il sistema operativo fornisce l'unica interfaccia diretta con l'hardware e in quanto tale ha un accesso esclusivo con il massimo dei privilegi detto *kernel mode*. Questo comporta che una vulnerabilità all'interno del sistema operativo può portare a gravi conseguenze per l'integrità e la sicurezza del sistema, inoltre qualche malintenzionato potrebbe approfittare di questo bug per trarne profitto. Uno degli obiettivi principali di un SO è quindi quello di garantire la sicurezza, ulteriore scopo è l'efficienza: un buon sistema operativo deve saper sfruttare al meglio tutte le risorse che ha a disposizione, dalla gestione della memoria per sfruttare al meglio lo spazio alla schedulazione dei processi per ottimizzare i tempi di esecuzione. Come ultimo obiettivo, ma non per questo meno rilevante, deve rendere il più semplice possibile l'utilizzo del dispositivo su cui è installato. Dalla definizione di SO data a inizio capitolo possiamo isolare una specifica parte di codice che è quella che permette al software di interfacciarsi con l'hardware, quindi l'accesso e la gestione delle risorse di un dispositivo, questa specifica parte si chiama *kernel* che come suggerisci il nome (nocciolo dall'inglese) rappresenta la parte centrale di un sistema operativo su cui tutto il resto si appoggia.

1.2 MICROKERNEL E KERNEL MONOLITICI

Esistono vari modelli strutturali per i sistemi operativi: monolitici, modulari, a livelli, microkernel ed ibridi, ad oggi i più diffusi sono gli ibridi, che combinano i vari modelli tra di loro, ma che in gran parte si basano su sistemi monolitici i quali consistono di un unico file binario statico al cui interno sono definite tutte le funzionalità del kernel e che viene eseguito in un unico spazio di indirizzi, questo comporta dei vantaggi:

- efficienza → motivo principale per cui la maggior parte dei sistemi operativi ancora si basano su kernel in gran parte monolitici, lavorando nello stesso spazio di indirizzi e gestendo tutto attraverso chiamate a procedura il SO risulterà molto reattivo e performante

- semplicità → in quanto non ha una vera e propria strutturazione ma il codice è tutto in un unico file binario risulta chiaramente più semplice da progettare anche se poi l'implementazione risulta difficile

d'altra parte ha anche degli svantaggi:

- inserimento di un nuovo servizio → questo richiede la ricompilazione del kernel, quindi non permette l'inserimento di un nuovo servizio a runtime (problema risolto nei modelli ibridi)

- dimensione → dovendo gestire tutte le principali funzionalità del sistema operativo il kernel sarà composto da milioni di righe di codice sorgente (MSLOC - linux ha circa 20MSLOC) e questo porta direttamente al successivo grosso svantaggio

- sicurezza → maggiore è il numero di righe di codice maggiore sarà il numero di possibili bug, essendo tutto il codice eseguito nello stesso spazio di indirizzi un bug rischia di far bloccare l'intero sistema anche se il problema è molto piccolo e isolato a una minima funzione del kernel.

All'estremo opposto troviamo i *microkernel* che sono composti da un kernel ridotto al minimo indispensabile, che comprende la gestione della memoria, dei processi e della CPU, le comunicazioni tra processi (IPC) e l'hardware di basso livello, mentre tutto il resto deve essere gestito da server (deamon) che operano sopra al kernel, quindi in spazi di indirizzi separati.
I microkernel sono spesso usati in sistemi embedded in applicazioni

mission critical di automazione robotica o di medicina, a causa del fatto che i componenti del sistema risiedono in aree di memoria separate, private e protette.
Anche questo modello ha dei vantaggi:

- flessibilità → l'inserimento di un nuovo servizio avviene al di sopra del kernel quindi in qualsiasi momento è possibili aggiungere o togliere servizi senza dover modificare il kernel.

- sicurezza → minore quantità di codice eseguita in kernel mode (quindi minore quantità di bug e minore superficie attaccabile) maggiore è la sicurezza del sistema, inoltre i servizi sono lavorano in uno spazio di indirizzi differente da quello del kernel di conseguenza se un server (su cui viene eseguito un servizio) smette di funzionare tutto il resto del sistema continua a funzionare normalmente e si potrà procedere a riavviare quel singolo servizio

- semplicità → essendo il codice composto da giusto qualche decina di migliaia di righe di codice (KSLOC) risulta molto più facile da scrivere

e dall'altro lato ha un grande svantaggio:

- efficienza → dato che ogni servizio gira a livello utente l'utilizzo di uno qualsiasi di questi richiede il ricorso a chiamate di sistema che rallentano fortemente l'esecuzione di ogni operazione, motivo principale per cui ancora oggi i sistemi operativi si basano in gran parte su sistemi monolitici.

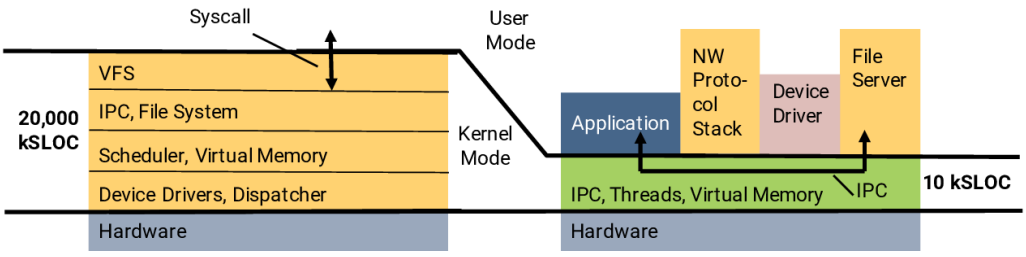

Figura 1: Kernel monolitici (sinistra) VS microkernel (destra)

1.3 HYPERVISOR

Un *hypervisor*, chiamato anche virtual machine monitor (VMM), è un tipo di sotware/firmware che permette di creare ed eseguire macchine virtuali. Un computer sul quale un hypervisor esegue una o più macchine virtuali prende il nome di host machine mentre le singole macchina virtuali prenderanno il nome di guest machine, su ognuna di queste è possibile far girare un sistema operativo diverso che eseguirà la maggior parte delle istruzioni direttamente sulle risorse hardware virtualizzate rese disponibili dall'hypervisor.

2

SEL4

seL4 fa parte della famiglia dei microkernel L4 che risalgono alla prima metà degli anni '90 creato da Jochen Liedtke per sopperire alle scarse performance dei primi sistemi operativi basati su microkernel, ad oggi fa parte del Trustworthy System.
Come descritto poco sopra nell'introduzione, seL4 essendo un microkernel, ha un numero di righe di codice sorgente estremamente piccolo e questo è sufficiente per determinare che non è un sistema operativo ma soltanto un microkernel, infatti non fornisce nessun dei servizi che siamo solitamente abituati a trovare su un comune SO, "è solo un sottile involucro attorno all'hardware" [1], tutti i servizi devono essere eseguiti in modalità utente e questi dovranno essere importati ad esempio da sistemi operativi open-source come Linux (oppure scritti da zero). data questa sua definiamola "incapacità" nel fornire servizi all'utente seL4 è anche un *hypervisor*, quindi è possibile eseguire macchine virtuali sulle quali far girare un comune SO che fornirà i servizi non presenti in seL4.

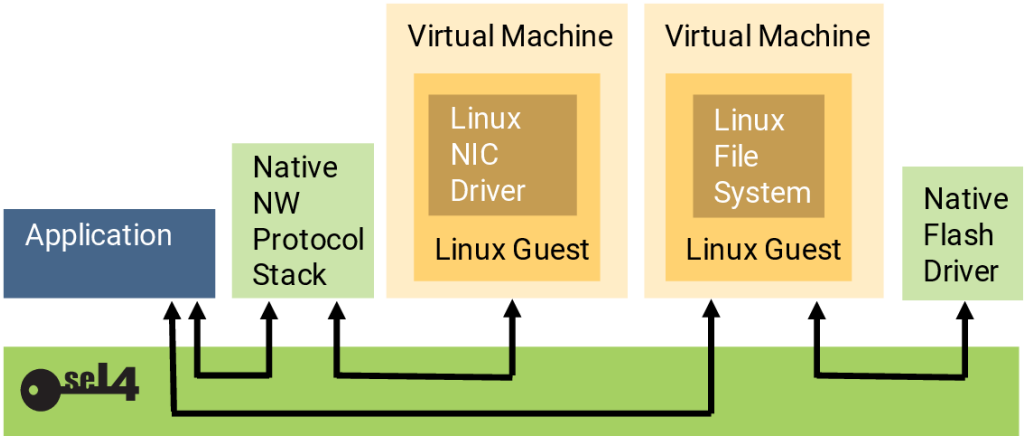

Figura 2: Virtualizzazione del SO Linux per l'intergrazione dei servizi di networking e file system

Un esempio pratico può essere quello mostrato in figura 2, in cui è raffigurato seL4, una generica applicazione e due macchine virtuali (VM) sulle quali viene eseguita una versione ridotta al minimo di Linux, che quindi avrà poco più oltre al servizio che dovrà eseguire. Queste due VM forniranno all'applicazione il servizio di networking ed il file system per la gestione della memoria secondaria (hard disk, supporti rimovibili ecc.), le comunicazioni tra le parti saranno gestite da un canale fornito dal microkernel ma le due macchine virtuali non avranno modo di comunicare tra di loro, non solo questo ma come si vede in figura anche le comunicazioni tra le varie parti e l'applicazione sono ben delineate e precise, nessun'altra comunicazione tra le varie parti è possibile al di fuori di quelle indicate dalle frecce.

2.1 CAPABILITY

Una *capability* è definita formalmente come un riferimento ad oggetto, nel nostro caso specifico possiamo definirla anche come un puntatore immutabile, cioè una capability farà sempre riferimento allo stesso oggetto. SeL4 è un sistema capability-based (basato sulle capability) questo significa che l'unico modo per eseguire un operazione è attraverso l'invocazione di una capability. Ad ognuna di esse, inoltre, sono associati dei diritti di accesso, quindi una capability è un incapsulamento di un riferimento ad oggetto con i diritti ad esso conferiti. Per dare una definizione meno formale possiamo pensare alle capability come a delle chiavi di accesso estremamente specifiche riguardo quale entità può accedere ad una particolare risorsa del sistema. Inoltre permettono di supportare il *principle of least privilege*, principio del privilegio minimo chiamato anche *principle of least authority* PoLA, questo principio implica che ogni modulo deve avere accesso solo ed esclusivamente alle risorse strettamente necessarie al suo scopo. In seL4 quindi i diritti dati ad un componente possono essere ristretti al minimo indispensabile per svolgere il loro lavoro come richiesto dal PoLA e chiaramente questo è un grosso punto a favore per quanto riguarda la sicurezza.
Nei sistemi operativi più comuni tipo Windows o Linux l'accesso alle risorse è gestito dalle *access-control list* (ACL), quindi, nel caso specifico di Linux, ad ogni file viene associato un set di bit che determinano quali operazioni (lettura, scrittura, esecuzione) possono essere eseguite su di esso dai vari utenti (proprietario, gruppo, altri) questo però implica che ogni file del sistema con lo stesso set di permessi è a disposizione di quello specifico utente, quindi se ci mettiamo nello scenario di voler

avviare un programma, di cui non siamo sicuri della sua attendibilità, su uno specifico file questo non è possibile perchè come può accedere a quel file può accedere anche a tutti gli altri che hanno le stesso ACL.
Con le capability questo scenario non si può presentare perchè il kernel consentirebbe un operazione se e solo se chi richiede di eseguire l'operazione ha la "giusta capability" per eseguire l'operazione su quel file.

2.1.1 *Proprietà delle capability*

Interposition → le capability hanno la proprietà di mettersi in mezzo tra chi crea una capability e l'effettivo accesso ad una risorsa: se un utente dà una capability ad un oggetto esso non è in grado di sapere cosa effettivamente sia quell'oggetto, può chiaramente utilizzarlo senza però sapere che tipo di oggetto sia. Delegation → le capability supportano la delegazione dei privilegi tra gli utenti: l'utente X ha un oggetto e vuole dare accesso ad esso anche all'utente Y; X può creare una nuova capability e darla ad Y senza conservare nessun riferimento all'utente X che l'ha creata, la nuova capability può anche avere meno diritti di accesso (esempio solo lettura invece di lettura e scrittura) e inoltre X in qualsiasi momento può revocare l'accesso ad Y in qualsiasi momento distruggendo la capability.

BIBLIOGRAFIA

[1] Gernot Heiser. The sel4 microkernel - an introduction. *The seL4 Foundation*, Revision 1.2, 2020. (Cited on page 11.)

[2] Wikipedia, l'enciclopedia libera. Kernel, 2023. ultima modifica 7 giu 2023.

[3] Wikipedia, l'enciclopedia libera. Operating system, 2023. ultima modifica 16 luglio 2023.

[4] Rosario Pugliese. Chiedere al prof come citare le sue slide.