



# Exokernel Teoria ed implementazioni

- PROFESSORE: GIUSEPPE CATTANEO
- SISTEMI OPERATIVI AVANZATI A.A 2022/2023
- STUDENTE: ANTONIO ALLOCCA



# OS tradizionali

L'OS è un'interfaccia tra le applicazioni e le risorse fisiche. Gli OS nascondono le informazioni delle risorse dietro ad astrazioni ad alto livello come

- ▶ processi
- ▶ file
- ▶ spazio di indirizzamento
- ▶ comunicazione tra processi



# Problemi degli OS tradizionali

## Os tradizionali limitano:

- Performance
- Flessibilità
- Funzionalità delle applicazioni
- Implementazione di astrazioni del sistema operativo come
  - Comunicazione tra processi (IPC)
  - Virtual memory (VM)

# Idea di base

Le applicazioni sono molto diverse le une dalle altre.

Un'interfaccia con lo scopo di supportare **ogni** applicazione deve tenere conto di tutte le sue possibili necessità.

L'implementazione di tale interfaccia deve **soddisfare** tutti i **tradeoff** e **anticipare** tutti i modi in cui l'interfaccia è usata.

Nasce la necessità di avere **un'interfaccia specifica** che soddisfi le necessità dello sviluppatore.

Ci sono stati diversi approcci per l'estendibilità:

- ▶ migliori microkernel (per fornire controllo più a basso livello)
- ▶ virtual machines (nasconde le informazioni)
- ▶ scaricare codice untrusted all'interno del kernel (ad esempio in SPIN)

L'obiettivo principale di un exokernel è anche quello di risolvere il dibattito tra **struttura monolitica contro microkernel**



# OS con exokernel

---

Un OS con exokernel fornisce una gestione a livello applicativo delle risorse fisiche.

---


Nell'architettura exokernel un piccolo kernel esporta le risorse hardware tramite un'interfaccia di basso livello ad untrusted LibOS

# LibOS

Non tutte le applicazioni necessitano di comunicare direttamente con il kernel.

Molti programmi possono essere collegati a librerie che nascondo le risorse a basso livello con astrazioni dei tradizionali OS.

A differenza delle tradizionali implementazioni di tali astrazioni, le implementazioni delle librerie possono essere modificate o sostituite a piacimento.



**LibOS** che lavora sull'interfaccia **exokernel** implementa **astrazioni** ad alto livello e chi scrive le applicazioni può selezionare le librerie o implementarle da sé.



Le **nuove** implementazioni del LibOS sono incorporate tramite **re-linking** degli eseguibili.



# Exokernel

Exokernel separa la **protezione** delle risorse dal **management**. Ci sono 3 tecniche per fornire delle risorse hardware in modo sicuro:

1. Secure bindings
2. Visible resource revocation
  - ▶ resource revocation protocol
3. Abort protocol
  - ▶ rompe i secure bindings con la forza se non cooperativi

# Exokernel e LibOS

Un'applicazione che usa direttamente un'interfaccia dell'**exokernel** non sarà portabile in quanto include informazioni legate all'hardware (**Hardware specific**)

Un'applicazione che usa **LibOS** che implementa delle **interfacce standard** sarà portabile per tutti i sistemi che implementano la stessa interfaccia



# Retrocompatibilità

Come in un sistema a **microkernel**, un exokernel può fornire retrocompatibilità in 3 modi:

1. Emulazione binaria del sistema operativo e dei suoi programmi
2. Implementando il suo HW abstraction Layer sull'exokernel
3. re-implementando le astrazioni dell'OS sull'exokernel

# Principi di design

Un exokernel specifica i dettagli dell'interfaccia che libos utilizza per interagire con le risorse

- ▶ Securely expose hardware: sicurezza con primitive a basso livello per permettere l'accesso diretto alle risorse
- ▷ Expose allocation
- ▷ Expose Names
- ▷ Expose Revocation.(resource revocation protocol)



# secure bindings



Meccanismo di protezione che disaccoppia l'autorizzazione dall'utilizzo della risorsa

Miglioramento delle performance in due modi:

1. controlli di sicurezza che prendono parte ad un secure binding sono espresse con operazioni semplici
2. controlla l'autorizzazione sono nel momento in cui viene effettuato il bind

"Simply put, a secure binding allows the kernel to protect resources without understanding them"





# Implementazione dei secure bindings

Necessità di primitive sia hw che sw per esprimere controlli della protezione



3 modi per implementare i secure bindings:

Hardware

Software

Download application  
code

# Multiplexing della memoria fisica

Quando un LibOS alloca una pagina della memoria fisica l'exokernel crea un secure binding per quella pagina registrando l'owner e i permessi di lettura e scrittura specificati da LibOS. L'owner di una pagina può modificare i permessi e deallocarla.

Per controllare i permessi di accesso a una determinata pagina di memoria viene usata una TLB

Per rompere un secure binding un exokernel deve cambiare i permessi associati e marcare le risorse come libere.





# Multiplexing della rete

Si devono conoscere i protocolli utilizzati per interpretare il contenuto dei messaggi che si ricevono e per identificare il destinatario.

Il demultiplexing può essere fornito sia a livello HW che SW

- ▶ HW: circuito viruale in celle ATM(Asynchronous Transfer Mode) per collegare flussi ad applicazioni
- ▶ SW: packet filters (esempio di secure binding in cui il codice è scaricato nel kernel)

Per quanto riguarda i messaggi in uscita, si può condividere l'interfaccia di rete.

# Downloading code

Alternativa all'implementazione dei Secure Binding

Scaricare codice direttamente all'interno del kernel porta a 2 vantaggi principali:

1. Eliminazione del kernel crossing
2. Il tempo di esecuzione può essere stimato e delimitato

# ASH

**ASH** (Application-specific Safe Handlers) sono un esempio di codice scaricato.

Essi possono essere scaricati nel kernel per partecipare all'elaborazione **dei messaggi**. Un **ASH è associato ad un packet filter** e viene eseguito quando un pacchetto viene ricevuto

Usando gli ASH la latenza del roundtrip è ridotta al minimo in quanto la risposta viene inviata al momento dell'arrivo del messaggio invece di essere accumulata fin quando l'applicazione è schedulata.



# Visible resource revocation

Quando una risorsa viene collegata ad una applicazione si deve poter pure rilasciare la risorsa e **rompere il secure binding**.

Tale revocazione può essere sia **visibile** che **invisibile**

- ▶ I tradizionali OS lo fanno in modo **invisibile**
- ▶ L'exokernel utilizza la revocazione **visibile** delle risorse per permettere al LibOS di salvare lo stato del processore solo quando richiesto

# Revocation and physical naming



Un exokernel deve esportare **nomi fisici** siccome sono efficienti e non c'è bisogno di tradurre i **nomi virtuali** in fisici.

# Abort protocol

Se viene effettuata una **richiesta di revoca di una risorsa** e ciò non avviene, allora l'exokernel definisce una seconda fase del protocollo di revoca in cui **viene imposto un rilascio delle risorse** in un tempo stabilito.

Questo comportamento definisce l'**Abort Protocol**

**difficoltà:** stabilire dei bounds in tempi reali per l'esecuzione dei programmi

Per **gestire le risorse** che devono essere revocate e quelle no, il LibOS utilizza una lista che prende il nome di **repression vector**

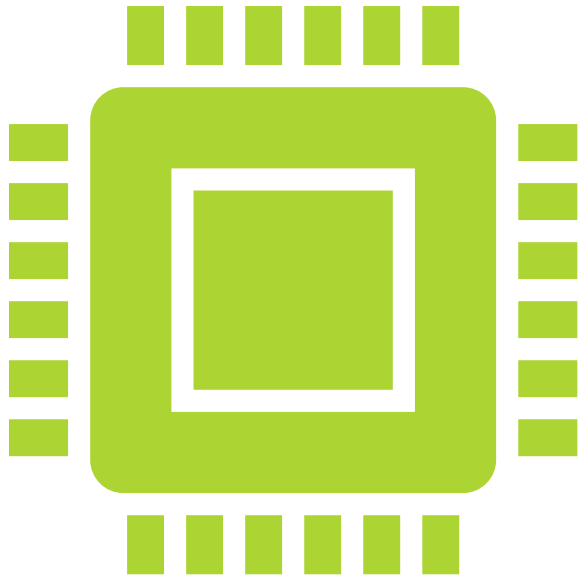


# Metodologie sperimentali

Sono stati implementati due sistemi software che seguono l'architettura ad Exokernel

- ▶ Aegis: exokernel
- ▶ ExOS: LibOS

# Aegis e ExOS



Sono entrambi implementati su stazioni DEC basate su MIPS

- ▶ Aegis esporta: processore, memoria fisica, TLB, eccezioni, interrupt, interfaccia di rete che usa un sistema di packet filter
- ▶ ExOS implementa: processi, memoria virtuale, eccezioni ad alto livello, IPC, protocolli di rete (IP, UDP, ARP/RARP)

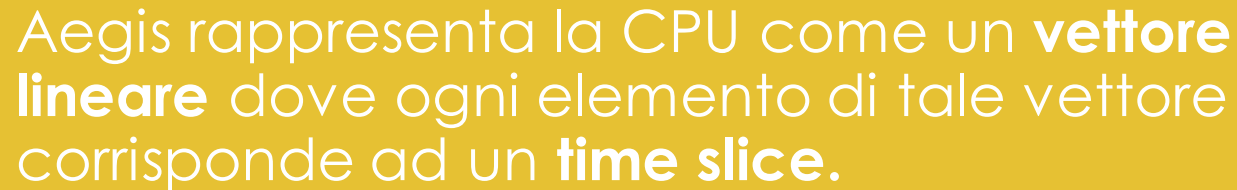


# Confronto con sistemi Ultrix

Il confronto con i **sistemi Ultrix** viene fatto per mettere in evidenza che **l'overhead** dei sistemi moderni può essere **rimosso** da **software specializzato**

# Time slice del processore

Aegis rappresenta la CPU come un **vettore lineare** dove ogni elemento di tale vettore corrisponde ad un **time slice**.



Scheduling in **round robin**



è presente una proprietà che prende il nome di **position** che rappresenta un upperbound di quando le azioni devono essere eseguite.



# Processor environment

Il processor environment di Aegis è una struttura che immagazzina informazioni per inviare degli eventi alle applicazioni.

Ci sono 4 tipi di eventi:

1. Exception context
2. Interrupt context
3. Protected Entry context
4. Addressing context

# System call in Aegis

Aegis ha due percorsi per le System call:

1. Senza stack
2. Con stack

Con questa soluzione il demultiplexing di system call composte è molto più rapido rispetto ad Ultrix





# Eccezioni

Per eseguire le eccezioni, Aegis compie vari passi:

1. Salva 3 scratch registers
2. Carica
  1. exeption program counter
  2. l'ultimo indirizzo virtuale tradotto tramite TLB
  3. causa dell'eccezione
3. **Jump** al program counter dell'applicazione con i permessi opportunamente modificati

Dopo aver processato l'eccezione, l'applicazione può immediatamente riprendere l'esecuzione senza passare per il kernel

# Address translation

Supportare la memoria virtuale a livello applicativo comporta 2 problemi

Bootstrapping del virtual naming system

Efficienza



Aegis supporta il **bootstrapping** tramite un piccolo numero di mapping "**garantiti**".



Ovviamente se il mapping dovesse fallire, sarà lanciata **un'eccezione** che sarà gestita automaticamente da Aegis.



# Mapping



# Protected control transfers

Per permettere la **comunicazione tra processi in modo efficiente** vengono usati i protected control transfers

Sostituzione del **program counter** con un valore concordato che denota la **time slice del processor environment** e installa gli elementi del processor context

Essi possono essere:

- ▶ Sincroni
- ▶ Asincroni

Entrambi garantiscono due importanti proprietà:

- ▶ Atomicità delle operazioni
- ▶ Aegis non sovrascriverà i registri (consistenza)

# Dynamic Packet Filter **DPF**

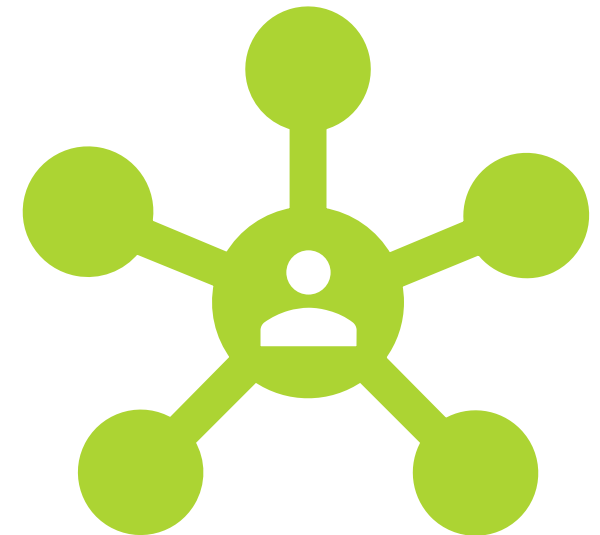
Il **sottosistema network** di Aegis deve fornire un efficiente **demultiplexing** dei messaggi e la loro gestione.

Nei tradizionali OS i packet filters sono **interpretati** quindi computazionalmente onerosi

in Aegis ciò viene fatto tramite **DPF** che permettono la **generazione automatica di codice eseguibile**

ciò viene fatto in due modi:

1. **Compilando** i packet filters quando sono installati nel kernel
2. Utilizzando dei **filtraggi prestabiliti** per ottimizzare il codice eseguibile in modo aggressivo



# Aegis efficiente





## Ulteriori implementazioni



sistema  
**Xok/ExOS**

**Xok** exokernel  
per intelx86



**Cheetah** HTTP/1.0 Server

# Supporto del kernel per astrazioni protette

Gli exokernel devono garantire la protezione fornendo anche un **controllo sugli accessi** ad oggetti definiti ad alto livello (tipo i file)

**Xok** fornisce questa protezione tramite 3 tecniche

1. Controllo degli accessi su tutte le risorse alla stessa maniera
2. Astrazioni software per collegare insieme risorse hardware
3. Alcune applicazioni dello Xok permettono il download di codice

# Descrizione del problema

il requisito più difficile da ottenere in modo **efficiente** è di **ottenere i permessi** che un utente possiede su un particolare **blocco** del disco.

- ▶ Disc-block-level multiplexing (associa ad ogni blocco un'ACL)
  - ▶ causa un enorme overhead di tempo in quanto si deve accedere al disco due volte per ogni blocco (lettura e controllo dei permessi)
- ▶ Self-descriptive metadata (ogni blocco aveva una parte iniziale che andava a descrivere il blocco in sé, metadati application-specific)
  - ▶ **overhead** di spazio e complessità di modifica in quanto le strutture dati nel file system non hanno un formato universale
- ▶ Template-based description
  - ▶ ci si è resi conto che effettivamente non è possibile avere un formato universale ma è altrettanto vero che le strutture dati su disco sono poche quindi è possibile descriverle usando dei template

# XN

È un sistema di **storage stabile** per **exokernel**

Fornisce un **accesso** ad uno storage stabile a livello di blocchi sul disco

Bisogna prevenire che un **utente malevolo** si impossessi di blocchi sul disco che non gli appartengono

XN utilizza **UDF** (Untrusted Deterministic Functions) che sono funzioni per la traduzione di **metadati** specifici per ogni tipo di file.





# UDF (Untrusted Deterministic Functions)

UDF analizza e **traduce i metadati** in un modo più semplice per il kernel.

In questo modo il kernel può analizzare in modo **efficiente** i metadati **senza conoscere il layout** con il quale sono scritti.

Gli UDF sono immagazzinati in strutture chiamate **templates** in cui ad ogni template corrisponde un particolare formato di metadato

# XN - Requisiti

1. Non consentire accessi non autorizzati, per velocità utilizza i **secure binings** per effettuare i controlli sugli accessi al momento del **bind** e non per ogni accesso
2. XN deve determinare senza ambiguità quali permessi un utente ha su ogni blocco (**UDF**)
3. Deve garantire che gli aggiornamenti del disco non permettono, in caso di **crash**, ad un LibOS di accedere a memoria che **non è stata allocata**

# XN – Proprietà

L'**integrità** del disco è garantita dal rispetto delle regole dettate da **Ganger** e **Patt**

Questo permette a più **LibOS** di poter coesistere sullo stesso sistema

l'interfaccia XN è utile anche per poter **interfacciare diversi filesystem** contemporaneamente (log-structured file systems, RAID, e memory-based file systems)

# C-FFS Lib File System

XN fornisce la protezione base per garantire l'**integrità** ma i file system devono poter garantire anche altri tipi di requisiti, ad esempio il file system di Unix deve poter garantire l'unicità del nome di un file all'interno della directory

Questo tipo di requisiti può essere fornito tramite 3 modi:

1. Nel kernel
2. In un server
3. Programmazione sicura



C-FFS scarica i metodi direttamente all'interno del kernel per controllare i suoi requisiti.

In C-FFS ci sono 4 principali meccanismi di protezione:

Access Control  
(uid,gid...)

Aggiornamenti well-formed: UNIX-specific file semantics

Atomicità (dati sempre reperibili e scrittura abilitata solo quando i metadati sono consistenti)

Aggiornamenti impliciti (ad es. la modifica di un file comporta l'aggiornamento della data in cui sono stati modificati l'ultima volta)

# HTTP Server

è stato dimostrato che un controllo a livello applicativo può dar vita anche ad **HTTP Server altamente efficienti** come **Cheetah**

**Cheetah** utilizza una versione modificata di:

- ▶ File system
- ▶ Implementazione di TCP

in base alle proprietà del traffico HTTP

Cheetah è 8 volte più veloce del migliore server HTTP basato su **Unix**.

disk subsystem: XN

# Cheetah HTTP/1.0 Server

L'architettura ad exokernel è ottimale per la costruzione di server ad alte performance.

I server sono basati su I/O e per questo è stata sviluppata una libreria XIO per un **I/O estendibile** ed un'applicazione che la utilizza ovvero **Cheetah HTTP Server**

Il compito di un server HTTP/1.0 è semplice

1. Il client richiede una risorse
2. Il server la trova
3. La restituisce al client



# Ottimizzazioni

## Merged File Cache e Pool di ritrasmissione

- Evita
  - che la cpu debba accedere sempre in memoria
  - pool di ritrasmissione TCP
- trasmette i file tramite cache ed utilizza checksum pre-computate

## Knowledge-based Packet Merging.

- vengono tenute conto le transazioni di stato che avvengono per ogni richiesta in modo da ridurre il numero di I/O.

## HTML-based File Grouping

- alloca i file contenuti in un documento HTML in blocchi adiacenti ad esso, solo quanto possibile

Grazie per  
l'attenzione

