ESEMPI DI SISTEMI e di FILE SYSTEM

Consideriamo alcuni sistemi operativi esistenti e le proprietà ed i modelli visibili a livello utente

Accent Mach

Amoeba

Chorus

V-kernel

DCE

Mach

Storicamente

Rochester **I**ntelligent **G**ateway (**RIG**)	1975
Accent	1981
Mach	**1987**

Microkernel con emulazione di UNIX (BSD, V e HP-UX)
Modello di accesso a workstation

obiettivi:
- **efficienza** della realizzazione
- **spazi di indirizzamenti** ampi e separati
- **accesso trasparente** alle risorse
- **eterogeneità** di architettura e di sistemi
- parallelismo di sistema ed applicazioni
- progetto di sistemi operativi con strategie utente

Si considerino le risorse create e gestite
- **processi**
- **thread** (nell'ambito di un processo)
- **porte**
- **oggetti memoria**
- **messaggi**

Il concetto di **porta** è nato con Accent:
ogni comunicazione, anche verso il kernel avviene tramite porte
Ogni processo ha la propria visione **locale** delle **porte**
Ogni accesso alle porte avviene tramite il **kernel** che può controllare la correttezza dell'operazione tramite la verifica delle capability per le porte

Comunicazione con porte

la comunicazione usa le porte ed accoda i messaggi su strutture di kernel apposite

Lo scambio di informazioni avviene tramite **messaggi** che possono contenere anche porte

le capability consentono le operazioni sulle porte:
ricezione / invio / invio-unico
Il kernel controlla la correttezza delle porte ed eventualmente invia eccezioni (porta eccezioni)
Invio-unico per la implementazione del cliente/servitore

*per potere realizzare la attesa multipla, un processo può definire un **set-di-porte**, da cui ricevere con una unica primitiva*

*sono previste operazioni **sincrone** ed **asincrone**, con **time-out***

Il messaggio contiene dati che possono anche non essere contigui *(vedi primitive scatter-gather per socket)* (tramite dati out-of-line)

Inoltre, si usa il meccanismo della **copy-on-write** per ottenere buone prestazioni

Il sistema ha il compito di **monitorare** le comunicazioni e di fornire informazioni (su richiesta) di uso delle risorse

Processi
I processi ospitano thread leggeri al loro interno e condividono una porta specifica
Ogni operazione significativa ha la propria porta
process port/ **thread** port
bootstrap port **exception** port
registered ports (per servizi well-known)

Il processo racchiude informazioni
di **memoria** (condivisa o privata dei thread)
di **scheduling** (anche il processore/i da utilizzare)
Non si prevede riallocazione

Le primitive di processo sono:
- **create**

(specifiche da un prototipo, anche eredità dal padre)
- **terminate**
- **suspend / resume**
- **getState / setState**
- **assign** (allocazione dei nuovi thread)

I thread hanno pesi diversi e sono del tipo **Cthread**
Le primitive di thread sono:
- **fork**
- **join**
- **detach**
- **exit**
- **yield**

La detach consente di non aspettare la terminazione del figlio: il thread viene immediatamente distrutto al completamento

I **thread utente** sono a livello applicativo
L'assegnamento dei thread ai thread del kernel (di basso livello) avviene secondo le specifiche dell'utente

si può intervenire sullo scheduling con un assegnamento opportuno: un thread importante viene assegnato da solo (applicativo e kernel) ad una CPU specifica

I thread hanno un sistema di priorità (**0-127**)
priorità corrente
priorità massima (bassa) -------- minima (alta)
Un thread può chiedere di alzare ed abbassare la priorità per **favorire** l'esecuzione di altri

Inoltre, per favorire thread che si vogliono riattivare, è possibile **dare spazio** a loro: questo suggerimento può migliorare molto le prestazioni

La **yield** ha anche lo stesso significato
scheduling
code differenziate per la stessa priorità
coda unica globale
In caso di nessun altro thread si riesegue il corrente
Anche un processo dummy in attesa

Per la sincronizzazione dei thread
semafori (mutex) e **variabili condizione**

Mach evolve nel senso di supporto veloce a politiche differenziate di comunicazione e di gruppo

Amoeba

Storia
Amoeba **1981** **1984**

Sistema Operativo distribuito
Modello a processor-pool
implementazione a **microkernel** con
servitori realizzati in spazio utente
ogni azione è delegata ad uno **strumento utente**
file server, directory server, replication server

obiettivi:
- accesso trasparente alle risorse globali
- eterogeneità di architettura e di sistemi
- parallelismo di sistema ed applicazioni
- progetto di applicazioni parallele

tutte le risorse create e gestite sono
oggetti come oggetti **passivi (puri dati)**
con interazione attraverso **RPC**

A basso livello
- **capability**
- **processi**
- **thread**
- **segmenti memoria**

Capability per oggetti

L'accesso ad ogni oggetto avviene attraverso *capability*
{Server_port, Object_ID, Diritti_Accesso, Check}
Ogni capability può anche essere **ristretta**
(prima di passarla ad altri)

Le primitive significative sugli oggetti sono:
- **garbage collector**
- **copy / destroy**
- **getParams / setParams**
- **info / status**
- **restrict**

Processi
Il processo viene creato dall'inizio con i propri segmenti necessari (e non dal padre) e può poi ampliare la memoria o restringerla dinamicamente

Il processo racchiude informazioni
di **memoria** (condivisa o privata dei thread)
di **scheduling** (anche il processore/i da utilizzare)
Non si prevede riallocazione, ma le informazioni di processo sono usate per la nuova allocazione nella valutazione del carico di ogni nodo

Ogni processo può aggiungere **segmenti di memoria**
la memoria come componenti contigui in uno spazio virtuale: i segmenti sono contigui anche fisicamente
Un processo esegue solo se tutti i suoi segmenti sono in memoria

Primitive di processo e thread

Le primitive di processo consentono di creare processi, dovunque nel sistema
- **create**
- **exec**
- **getLoad**
- **stun** (cioè blocca in modo normale o per emergenza)

Un processo nasce come un solo thread che poi ne può creare dinamicamente che condividono lo spazio del processo
I thread hanno priorità
lo scheduling è o **preemptive** o **run-to-completion**

Per la sincronizzazione dei thread
semafori, **mutex**, e **segnali**
- segnali solo tra thread dello stesso processo
- mutex per la mutua esclusione con rispetto della priorità dei thread ed anche la possibilità di avere time-out
- semafori con la possibilità di avere time-out

Comunicazione

la comunicazione si articola in:
RPC (a basso livello message passing punto-a-punto)
broadcast

RPC semantica at-most-once

con primitive di basso livello

trans **get-request**
put-reply

La capability per il cliente passa al server con la richiesta (anche possibile crittografare)

Comunicazione di Gruppo

si introducono gruppi chiusi: per arrivare basta richiedere una operazione ad un componente del gruppo

- **creategroup**
- **joingroup**
- **leavegroup**
- azioni al gruppo
  sendtogroup
  receivefromgroup
  recovery

Uso di broadcast atomico

Reliable Broadcast

Un nodo è considerato il gestore (**sequencer**)
In caso di fallimento, si procede con una fase di elezione ad eleggere un nuovo nodo sequencer

Ogni messaggio di broadcast viene inviato al sequencer che gli assegna un numero unico in sequenza (sbloccando il mittente)
La **numerazione unica** consente di evitare conflitti: ogni partecipante al gruppo considera i messaggi nello stesso ordine (hold-down)

Il kernel del mittente bloccato provvede alla ritrasmissione nel caso non si abbia risposta
Il sequencer che vede una richiesta a cui ha già risposto provvede a riinviarla
(si sfrutta il **piggybacking** dove possibile)

il sequencer tiene la **storia totale** dei messaggi
e in caso di guasto del sequncer la storia deve essere *implicitamente* od *esplicitamente* ricostruita

Comunicazione a basso livello

protocollo ad-hoc per ottenere efficienza nel message passing

FLIP protocol
Protocollo di trasporto con nomi unici legati al processo (**64** bit)

Chorus

Sistema Operativo Europeo - progetto CEE
Chorus (v. 1) 1979/80
Nasce come kernel basato sul modello ad attori
Versione multiprocessore (v. 2 e v.3)
Versione commerciale (v.4) **Chorus Systemes**

Microkernel
anche **emulazione di UNIX**
Chorus/Mix anche per il controllo processi

obiettivi:
- efficienza della realizzazione
- accesso trasparente alle risorse attraverso server
- eterogeneità di architettura e di sistemi
- parallelismo di sistema ed applicazioni
- progetto di sistemi operativi con strategie utente
- integrazione con tecniche **Object-Oriented**
- progetto di sistemi **real-time**

Il concetto di **attore**:
ogni entità di esecuzione è leggera e comunica usando lo scambio di messaggi

Attori

Ogni attore esegue nel proprio spazio con la possibilità di ospitare diversi flussi di esecuzione (**thread**)

Gli **attori** possono determinare servizi (e server)
che possono operare
sia a livello di kernel efficienza
sia a livello utente
senza che l'utente se ne accorga
Ogni attore viene ritorvato attraverso una propria porta

Il **sistema** è a **livelli** (dal basso):
- microkernel
- processi di kernel
- processi di sistema
- processi di utente

Risorse create e gestite
- **attori**
- **thread** (nell'ambito di un attore)
- **regioni**
- **porte e gruppi di porte**
- **messaggi**

ACCESSO AGLI OGGETTI

Uso di Identificatori unici

UID di 64 bit con nomi statici (Unique Identifier)
{TipoOggetto, Nodo, TimeStamp}

LID di 32 bit (Local Identifier) di contesto

Capability

L'accesso ad ogni oggetto avviene attraverso *capability*

{UID, Chiave}

Ogni parte a 64 bit

L'accesso agli oggetti avviene tramite
porte e gruppi di porte
che corrispondono a
attori e gruppi di attori

Creazione di attori

Le primitive di processo sono:

- **ActorCreate**

(specifica con eredità dal padre)

- **ActorDelete**
- **ActorStop**
- **ActorStart**

I thread hanno pesi diversi e le primitive di thread sono:

- **ThreadCreate**
- **ThreadSuspend/ Resume**
- **ThreadContext**
- **ThreadLoad** (salvare stato)
- **ThreadStore**

COMUNICAZIONE E SINCRONIZZAZIONE

Per la comunicazione, scambio di messaggi attraverso le porte (anche gruppi di processi)

Il messaggio è anche esso un oggetto:
header di 64 byte
corpo (< 64 Kbyte)

1) invio asincrono di messaggi (datagram)
senza garanzia di consegna
anche gruppi di multicast per la comunicazione
(multicast unreliable asincrono)

2) RPC (at-most-once)

Per la sincronizzazione dei thread (interna ad un attore)
semafori
mutex

Il sistema prevede poi:

- possibilità di eccezioni
- possibilità di migrazione degli attori
- possibilità di memoria condivisa distribuita

V-kernel

Sistema Operativo per File system di Stanford
Thoth 1972/1979
Nasce come kernel per ottenere prestazioni efficienti
David Cheriton

Kernel
come un **componente tipo bus**
in sistemi ampi e con molti processori eterogenei
File system **Andrews** efficienza e migrazione
Operating system **processi** per load sharing

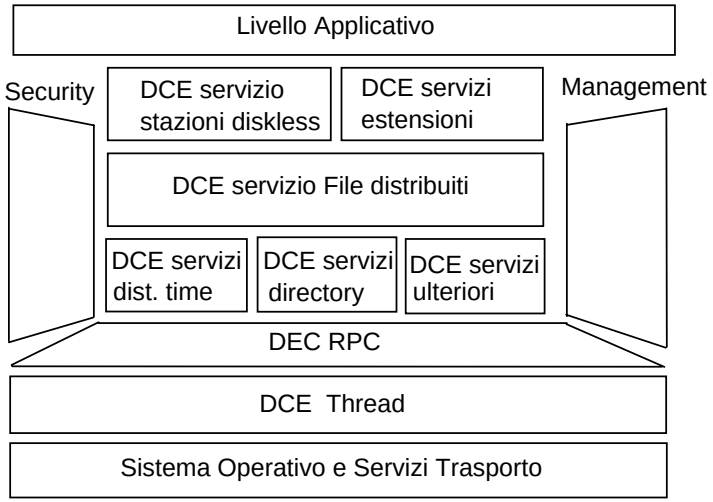
caratteristica e proprietà:
si definisce il **concetto di gruppo** e l'**insieme di azioni possibili** in un **gruppo**
concetto di server come **gestore**

obiettivi:
- **efficienza della realizzazione**
- accesso trasparente alle **risorse** attraverso **C/S**
- **eterogeneità** di architettura e di sistemi
- **parallelismo** di sistema ed applicazioni
- progetto di sistemi operativi con strategie utente
- progetto di sistemi **real-time**

Insieme di strumenti e meccanismi (tutti in **V**
VTMP V-kernel Transport Message Protocol
VICE & **VENUS** supporto al file system)
in evoluzione continua fino al 1993/1995

DISTRIBUTED COMPUTING ENVIRONMENT

(DCE) OSF



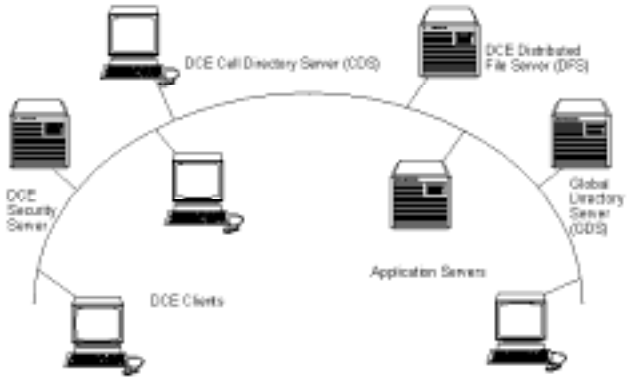
Strumenti DCE per programmazione distribuita
DCE **Remote Procedure Call**
DCE **Thread**

Thread per avere un elevato parallelismo di esecuzione a basso overhead
Se il sistema operativo sottostante prevede Thread, i DCE Thread sono superflui

Uso di **RPC** tradizionali od anche autenticate e sicure

DCE

il sistema si sviluppa fornendo una molteplicità di servizi, spesso con servitori dedicati



obiettivi:
- introduzione della località
- overhead limitato
- accesso trasparente alle risorse
- gestione sicurezza
- eterogeneità di architettura

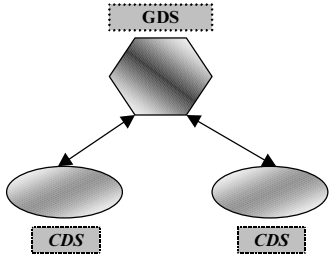
Risorse fondamentali sono
- **celle**
  località di visibilità di risorse ed oggetti comuni con politiche comuni di gestione
- **processi**
  clienti dimensioni limitate
  server grosse applicazioni
- **thread** con priorità

CELLE

Le **celle** sono degli insiemi di
gruppi di nodi e utenti

Il raggruppamento in **celle** viene fatto in base ad esigenze comuni di:
- sicurezza
- condivisione di risorse comuni
- limitare overhead
- problemi di amministrazione comune

Una cella mette insieme i servizi di:
- **directory di cella** (Cell Directory Service)
- **sicurezza di cella**
- **tempo di cella**



I servizi di cella si devono coordinare tra loro e globalmente per fornire servizi globali, soptrattutto di direttorio globale (GDS)
Il GDS si occupa dell ainterfaccia verso l'esterno e verso direttori X.500
I servizi di directory di cella possono anche essere replicati: una copia master e altre aggiornate o ad ogni cambiamento o periodicamente

Creazione entità

Si possono usare template per la creazione di ogni entità
processi, thread, mutex, variabili condizione

Le primitive di thread sono:
- **create**
- **exit**
- **join**
- **detach**

Lo **scheduling** dei thread è predefinito:
- FIFO
- round-robin (priorità alta prima)
- default
  quanto di tempo proporzionale alla priorità

Per la sincronizzazione si usano
mutex
variabili condizione

I **mutex** sono gestiti in modi diversi:
- fast mutex (vedi lock)
- recursive mutex (con possibilità di accesso dello stesso processo più volte)
- non-recursive mutex (errore al rientro)

Le **variabili condizione** rilasciano i mutex acquisiti

COMUNICAZIONE CON RPC

la comunicazione avviene tramite **RPC**
supporto per la generazione degli stub

Si usa
- un **linguaggio IDL** per la definizione di interfacce uniche
- un **name server** per ritrovare le entità nell'ambito di una cella e il coordinamento tra celle diverse

Tra celle diverse sono possibili anche sistemi di nomi del tutto standard come
- **DNS** in ambiente TCP/IP
- **X.500**, cioé un servizio di direttorio, in ambiente OSI

La identificazione del servizio avviene attraverso la generazione di numeri unici (**uuidgen**) associati al singolo servizio
uuidgen sono valori a **128** bit

Ogni richiesta può essere autenticata ed autorizzata correttamente
la security si ottiene attraverso l'uso di Kerberos associata all'uso di celle

SERVIZI

I **servizi** disponibili
- **Directory** Service
- **Distributed File** Service
- **Distributed Time** Service
- **Security** Service
- **Diskless Support** Service

Directory Service è un deposito centralizzato di informazioni sulle risorse disponibili
nome della risorsa e attributi

Distributed File Service (DFS) consente l'accesso e la condivisione di file su File Server

Diskless Support Service consente a un nodo privo di disco di operare

Distributed Time Service provvede alla sincronizzazione del tempo assoluto

Security Service assume normalmente "non fidato" il sistema operativo di ciascun host

SOTTOUNITÀ

Login Facility e Authentication Service

Kerberos come protocollo di autenticazione

ogni utente può ottenere gradi di sicurezza per le proprie risorse

Registry Service

si occupa del management degli account e del database dei soggetti autorizzati

ogni utente al login viene corredato dei diritti corretti

Access Control List (ACL) Facility

liste di utenti autorizzati all'uso di una determinata risorsa per ogni Application Server

ogni utente appartiene ad un gruppo per la verifica delle azioni (lista accessi per oggetto)

Privilege Service

sicurezza ulteriore nella procedura di accesso alle risorse

Privilege Attribute Certificate (PAC)

PAC consente di fare controlli sul livello di autorizzazione previsto per il richiedente

Servizio di autenticazione basato su chiavi segrete ed autorizzazioni per ogni servizio sulla base di chiavi di sessione

Si usano le celle come ambiti di verifica dell'accesso e ci si coordina con altre celle (e gestori) in caso di comunicazioni inter-cella