

# Reti e Sistemi Operativi

by Quack

# Reti e Sistemi Operativi

~Quack



Per la parte di Reti, vedrai la paperina a sinistra



Per la parte di Sistemi Operativi, vedrai la paperina a destra

## Reti di Telecomunicazioni

1. Introduzione
2. Livello applicazione
3. Livello trasporto
4. Livello reti: piano dei dati
5. Livello reti: piano del trasporto
6. Livello collegamento

## Sistemi Operativi

1. Struttura e Servizi
2. Processi e Thread
3. Scheduling CPU
4. Gestione memoria
5. File System
6. Sistemi di I/O

Reti



## Reti di Telecomunicazioni

~Quack

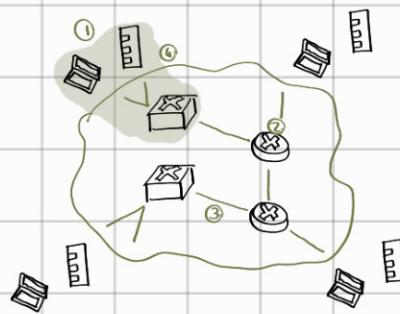
### Capitolo 1: Introduzione alle reti

1. Internet: miliardi di dispositivi connessi:

- host
- dispositivi ai margini della rete

2. Packet switches: spedisce pacchetti (parti di dati)

- router
- switch



3. Communication links: dove si muovono i pacchetti.

- fibra, rame
- satellite, onde radio
- banda: rateo di trasmissione

4. Network: collezione di dispositivi

ISP: Internet Service Provider

Protocollo: Controlla invio, ricevimento e gestione pacchetti.

Standard: Documenti che specificano le regole.

Protocollo: Definisce il formato, l'ordine del messaggio da mandare e ricevere tra le entità e prende azioni in base a ciò che riceve.

· Host: client e server.

· Reti di accesso: wired e wireless

· Network core: reti interconnesse tra routers. Decidono su dove mandare il pacchetto.

\* tra i 3 vedremo di più gli ultimi





## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Introduzione a host e router

Funzioni dell'host:

- riceve messaggio
- lo spezzetta in pacchetti di lunghezza  $L$  (bits)
- trasmette i pacchetti a un transmission rate  $R$  (bits/sec)
- packet transmission delay:  $L/R$

2 funzionalità principali:

- commutazione (forwarding): azione locale la quale mi informa su quale percorso devo inviare un pacchetto.

Ciò avviene tramite forwarding table

- instradamento (routing): azione globale che attraverso algoritmi indirizza il pacchetto fino alla destinazione.

forwarding table	
001	1
010	2
011	3



Store and forward: modalità dove prima che il pacchetto possa essere spedito su un link, devo averlo nello sua interezza



Queuing: Succede quando arrivano informazioni più velocemente di quanto posso servirli.

Il buffer contiene i pacchetti in coda. Se il buffer è pieno, si perdono i dati.



## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Introduzione a ritardi e flussi.

Packet delay: 4 tipi da sommare tra loro:

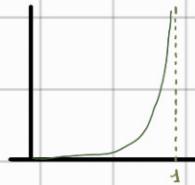
- trasmissione: dimensione pacchetto / rateo trasmissione.
- elaborazione: al nodo: check bit error, determinare output link. microsecondi.
- accodamento: livello di congestione sul router. Dipende da pacchetto  $\alpha$  pacchetto. millisecondi.
- propagazione: lunghezza link / velocità di propagazione.  $d_{prop}$ : d/s.

accodamento:

$$\left. \begin{array}{l} \text{a: media arrivo pacchetti} \\ \text{L: lunghezza pacchetto} \\ \text{B: rateo di trasmissione} \end{array} \right\} \text{intensità di traffico } \frac{L \cdot a}{B}$$

Se:

- $L \cdot a / B \rightarrow 0$ : poco delay.
- $L \cdot a / B \rightarrow \infty$ : aumenta la coda
- $L \cdot a / B > 1$ : arriva più lavoro di quello che zero!



Perdita di pacchetti.

throughput: n° di bit/unità di tempo necessari tra sorgente e ricevitore. Può essere:

- istantaneo: rateo ad un certo punto
- average: rateo medio.

Ese.  $R_s$  = Rete di accesso,  $R_c$  = rete di core.

$$R_s < R_c: \text{throughput} = R_s$$



$$R_s > R_c: \text{throughput} = R_c$$



Bisogna vedere chi è bottleneck (spesso  $R_s$ )



## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Introduzione a struttura Internet

Rete a commutazione di circuito: Alternativa al packet switching: non abbiamo pacchetti, ma abbiamo risorse dedicate.

Il circuito diventava inattivo se non vi è una chiamata. Non abbiamo il problema della coda, ma si ha uno spreco di risorse. Per questo è obsoleta nel mondo Internet, in quanto non sostiene troppi utenti.

Esempi: tradizionali telefoni.

Confronto packet vs circuit switching:

- packet migliore per "bursty data": condivisione risorse e non bisogna preparare la chiamata.
- packet ha possibili perdite e ritardi.

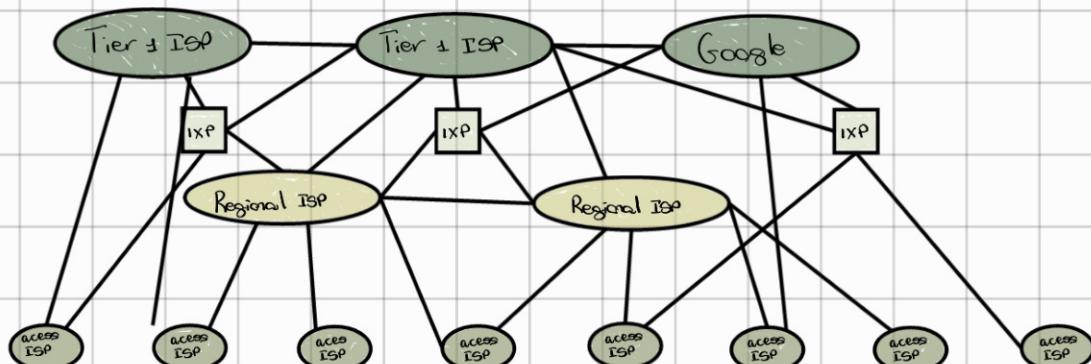
Struttura Internet: i vari access host si collegano a ISP globali che possono essere in competizione tra loro. I vari ISP globali sono collegati in due modi:

- peering link: gratis
- Internet exchange point: a pagamento.

Spesso però tra i punti di accesso locali si connettono prima quelli regionali, i quali a loro volta si collegano ad al più ISP globali, per avere più sicurezza di connettività.

I content provider sono grosse aziende con le proprie reti, che dovrebbero collegarsi direttamente agli access point, ma spesso non è possibile, quindi si "attaccano" agli ISP e ai globali.

Esempio:





## Reti di Telecomunicazioni

~Quack

### Capitolo 1: Introduzione ai protocolli a strati

**Layer Protocol:** Vogliamo trovare una struttura gerarchica delle reti, dato le varie parti che la compongono.

Avere ogni livello con un servizio preciso, sfruttando i servizi compiuti in precedenza dal livello inferiore.

Cambiando le procedure di un livello, non interessa gli altri livelli.

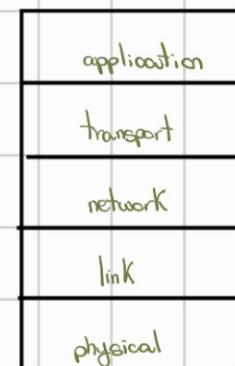
**Protocollo TCP/IP:** application: supporto applicazioni network (es. http, dns)

transport: processo di trasferimento dati (es. TCP, UDP)

network: indirizzamento pacchetti (es. IP)

link: trasferimento di dati tra nodi adiacenti. (es. Wi-Fi)

physical: trasferimento bit sul cavo.



L'overhead è la quantità di bit da aggiungere al messaggio tramite intestazioni.

Applicazioni scambiano  $n$  messaggi tra loro.

A livello transport avviene l'**incapsulamento** del messaggio  $M$  con l'intestazione  $H_1$ , creando un **segmento**.  $[H_1|M]$

A livello network il segmento diventa **datagram** con l'**incapsulamento + aggiunta** dell'intestazione  $H_n$ .  $[H_1|H_n|M]$

A livello link il datagram diventa **frame** con l'**incapsulamento + aggiunta** dell'intestazione  $H_l$ .  $[H_1|H_n|H_l|M]$



## Reti di Telecomunicazioni

~Quack

### Capitolo 2: Livello applicativo

DNS: Domain Name System: database distribuito associato ad ogni indirizzo IP un nome. (gerarchico)

Gli host comunicano con i server per risolvere nomi.

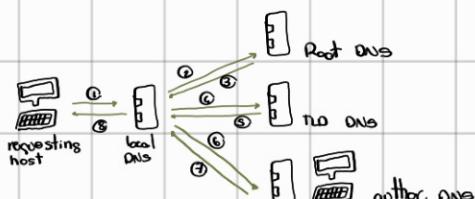
Struttura DNS:



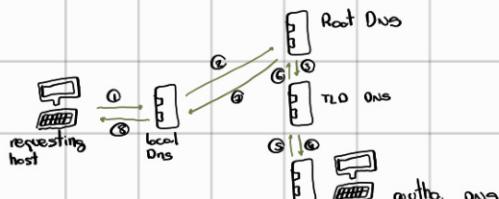
Iterated Query: Può contattare direttamente i server root, TLD e altri DNS server.

Recursive Query: Il server non può reindirizzare il client dns a un server dns diverso, quindi passa da root dns → TLD

Iterativi



Ricorsivi





## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Livello trasporto

Servizio: Fornisce comunicazione logica tra i processi applicativi di host differenti.

- Azioni:
  - sender: spezzetta e passa sotto a livello rete
  - receiver: riassembla e passa sopra al livello applicazione.

Protocolli: TCP, UDP

Socket: interfaccia tra liv. appl. e trasporto



Step:  
1) Sender manda application-layer message al socket

- 2) Determina i segmenti del messaggio, viene creato e lo passa al IP che lo passa al receiver
- 3) Riceve il messaggio e viene estratto.
- 4) Liv. applicativo demultiplexer il messaggio

TCP: Transmission Control Protocol: complesso, servizi importanti, come:

- consegna in ordine dei messaggi ed affidabile.
- congestione di controllo, adeguando le quantità di messaggi da inviare. (rete)
- controllo di flusso, evitando di sovraccaricare l'host destinatario (host)
- connection setup

UDP: User Datagram Protocol, non garantisce l'affidabilità di TCP. Ha solo 3 funzionalità:

- multiplexing: gestisce dati da socket multipli, aggiungendo header specifici utili in fase di demultiplexing.
- demultiplexing: quando arriva un segmento con l'header multiplexing, li indirizzo alla socket corretta.
- connectionless: non c'è bisogno di stabilire la connessione
- indipendenza pacchetti: ogni segmento è indipendente dagli altri

Entrambi i servizi non garantiscono un ritardo specifico o una banda minima.

Connectionless Demultiplexing(UDP): crea socket specifico host-local port. Specifico nel datagramma: ip add. dest e dest port #.

Connection Oriented Multiplexing(TCP): TCP socket identificati: source and dest ip address, source and dest port number.



## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Livello trasporto

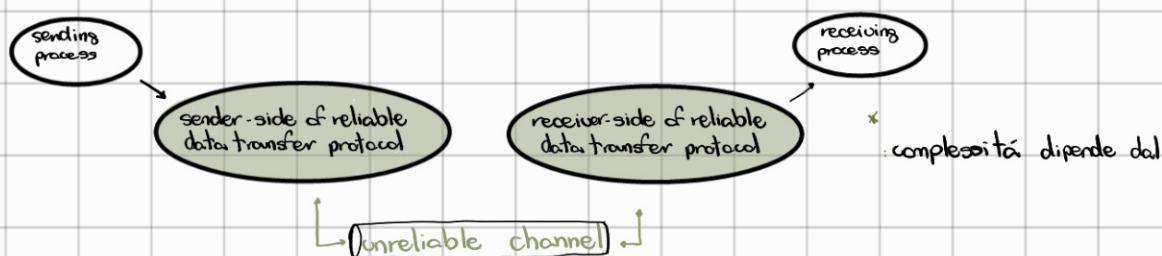
UDP segment header:



- checksum: trova errori

- length: lunghezza in bytes del segmento UDP.

Principio trasferimento affidabile:



positive acknowledge (ACK): ha ricevuto il messaggio correttamente.

negative acknowledge (NACK): ha ricevuto il messaggio non correttamente.

Il problema è che ACK/NACK possono essere anche loro corrotti.

Una soluzione è numerare ogni messaggio e passare mandarne una copia per controllare ed evita duplicati.

Aggiungendo un timer possiamo gestire le perdite. Implemento stop-and-wait. Dobbiamo calibrarlo perché:

- troppo corto: ritrasmissione inutile
- troppo lungo: stop di trasmissione lungo

Performance con ACK:  $V = \frac{L/R}{RTT + L/R}$  RTT = round trip time.

Un'alternativa a stop-and-wait sono le "scorrimenti di finestra", dove la finestra corrente non si chiude fino all'arrivo del primo ACK.

Se un segmento viene perso si possono usare diversi protocolli:

1. go-back-n: ritrasmette i segmenti. Non ho bisogno di un buffer in quanto duplicativi vengono scartati e gli ACK's sono accumulativi.
2. Selective repeat: si ritrasmette solo il segmento perso, serie in buffer per riordinare.

I due protocolli a schema a scorrimento non sono gli unici, TCP usa un ibrido tra i due.



## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Livello trasporto

Overview TCP:

- point-to-point: un sender, un receiver
- reliable: in-order byte
- full duplex data: comunica in 2 direzioni.  
si ha un maximum segment size.
- acks cumulativi
- connection-oriented: handshaking
- flow controlled: no overwhelmed receiver
- pipelining

TCP segment header



Dettagli

- sequence number: numero del 1° byte del segmento
- ack number: sen manda Seq=x e Ack=y, rec manda Seq=y e Ack=x+1.  
sen rimanda Seq=x+1 e Ack=y+1
- C,E: congestion notification
- RST, SYN, FIN: connection management.
- Receive window: flow control

Per settare bene il timer del TCP bisogna conoscere il Round Trip Time (RTT). Per stimarlo si può:

Sample RTT: misura il tempo che impiega il segmento precedente.

Estimated RTT: valore stimato + sample RTT con pesi diversi.

$$\text{Estimated RTT} = (1-\alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

TimeOut Interval: EstimatedRTT + 4\*DevRTT dove

DevRTT: Range di sicurezza:  $(1-\beta) * \text{DevRTT} + \beta * | \text{SampleRTT} - \text{EstimatedRTT} |$  (maggior è la differenza tra Sample e Estimated maggiore il margine)



## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Livello trasporto

TCP fast retransmit: Se il ricevitore riceve 3 ACKs uguali, rimanda il segmento unAcked perché probabilmente abbiamo perso un pacchetto.

TCP Sender: crea segmento con seq#, avvia 1° byte del segmento, start timer, controllo timeout e ACKs

TCP flow control: attraverso rwind (receiver window) riceviamo dati senza ricevere overflow nel buffer.



TCP Handshake: su 3 vie:

- Il sender inizia con un Seq=x e Synbit=1
- Il ricevitore risponde con Seq=y e Synbit=1 e manda ACKbit=1 e ACKnum=x+1
- Il sender capisce di avere un receiver e allora conferma con ACKbit=1 e ACKnum=y+1

TCP Closure:

- Sender manda FINbit=1. Da qui due possibilità:

Server non ha finito



Server ha finito

- ACK bit = 1 (ser)
- ...
- FINbit = 1 (rec)
- ACK bit = 1 (sen)

- ACKbit=1, RSTbit=1 (rec)
- ACKbit=1 (sen)

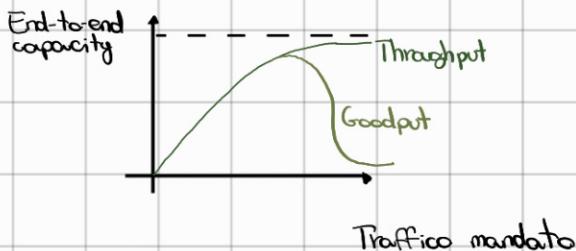


## Reti di Telecomunicazioni

~Quack

### Capitolo 3: Livello trasporto

Congestione: abbiamo troppi dati che la rete non regge! (+ flow control)



Ci sono due tipi di gestione della congestione:

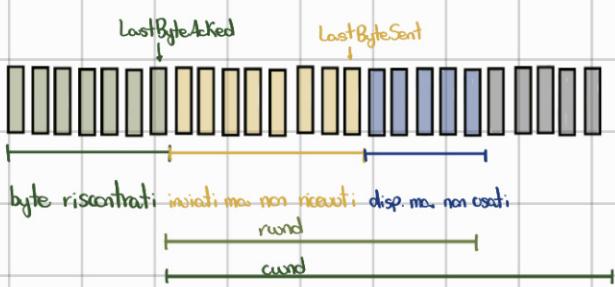
- End-to-end: no feedback, congestione causata da perdite di informazioni (tcp)
- Network assisted: raddrizza feedback dando il livello di congestione e esplicito un rate da mantenere.

Il sender può aumentare velocità di invio fino a perdita di pacchetti e poi diminuisce il rateo di trasmissione.

Additive Increase: increase by 1 maximum segment size (mss) every RTT until loss detected

Multiplicative Decrease: cut sendingrate in half at each loss

### Congestione e Controllo del Flusso



Quando cwnd < rwnd il controllo di congestione è dominante rispetto al flusso.

• Limite Trasmissione:  $\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{cwnd}, \text{rwnd}\}$

• cwnd è dinamicamente regolato in risposta alla congestione della rete.

• rwnd è regolato in base al valore ricevuto dal ricevitore



## Reti di Telecomunicazioni

~Quack

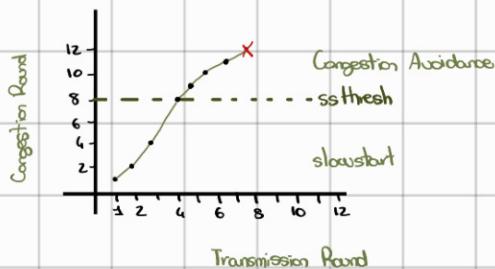
### Capitolo 3: Livello trasporto

**Slow Start:** Quando la connessione inizia, il flusso cresce esponenzialmente fino alla prima perdita.

- cwnd inizializzato a 1
- lo raddoppia ogni RoundTripTime (RTT)
- fatto aumentando cwnd per ogni ACK ricevuto

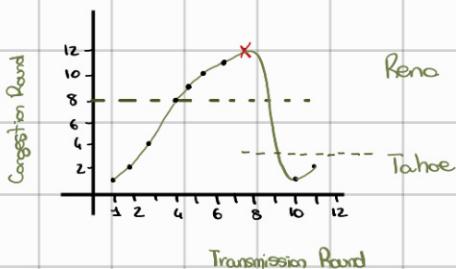
**Lineare:** Slow Start diventa lineare quando cwnd diventa  $\frac{1}{2}$  del suo valore prima dell'ultimo timeout.

Si implementa con la variabile ssthresh, che in caso di perdita, è settato a  $\frac{1}{2}$  cwnd prima della perdita.



**Congestion Avoidance:** Con TCP Reno si utilizzano fast recovery dopo il triplicato degli ACKs.

- $cwnd = ssthresh + 3 \text{ MSS}$
- uguali a TCP Tahoe in time out.



# Sistemi Operativi

## Teoria

# Sistemi Operativi

~Quack

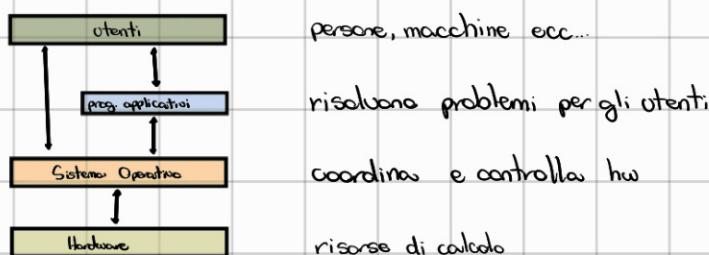


## Capitolo 1: Struttura e Servizi

Il SO è il primo programma che viene eseguito all'accensione di un computer. Ci permette di gestire il computer, installando ed eseguendo altri programmi (applicazioni) contemporaneamente e scambiare informazioni tra loro. Fornisce un ambiente omogeneo attraverso un insieme di regole, alla quale le applicazioni devono attenersi. Infine, mantiene ed organizza i nostri dati sotto forma di file e cartelle.

- Un SO è un insieme di programmi (software) che gestisce gli elementi fisici di un computer (hardware).
- Fornisce una piattaforma di sviluppo per i programmi applicativi che permette loro di condividere ed astrarre le risorse hardware.
- Agisce come intermediario tra utenti e computer.
- Protegge le risorse degli utenti, dai eventuali intrusi (virus).

### Componenti di un sistema di elaborazione (versione base)



Cosa richiede un dispositivo ad un SO?

- Server, mainframe: max performance, rendere facile la condivisione di risorse tra utenti.
- Laptop, pc, tablet: max facilità d'uso, produttività di chi lo usa.
- Dispositivi mobili: ottimizzare i consumi energetici e connettività.
- Sistemi embedded: funzionare senza intervento umano e reagire in tempo reale agli interrupt.

# Sistemi Operativi



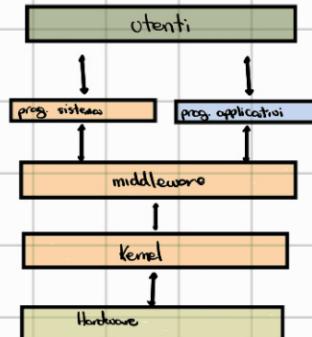
~Quack

## Capitolo 1: Struttura e Servizi

La maledizione della generalità è quando un so cerca di funzionare su diversi dispositivi con diversi scopi, ma purtroppo si avranno performance scarse.

### Struttura dei sistemi operativi

- Kernel: utilizza e crea le chiamate di sistema delle applicazioni.
- Middleware: Gestisce le chiamate con framework e librerie per renderle più "semplici" (API)
- Programmi di sistema: utilizzano i servizi offerti dal middleware e vengono creati dai utenti



### Principali servizi

- Controllo processi: caricano, eseguono e controllano le terminazioni dei programmi.
- Gestione File: leggere e scrivere files e directory
- Gestione dispositivi: permettono I/O per i programmi
- Comunicazione tra processi: scambio informazioni tra applicazioni
- Protezione e sicurezza: Controllano e difendono il sistema da attori esterni.
- Allocazione risorse: alloca le risorse hw in modo equo ed efficiente.
- Rilevamento errori: intraprende azioni di recupero quando si verificano errori.
- Logging: mantiene tracce di quali programmi usano quali risorse.

# Sistemi Operativi



~Quack

## Capitolo 1: Struttura e Servizi

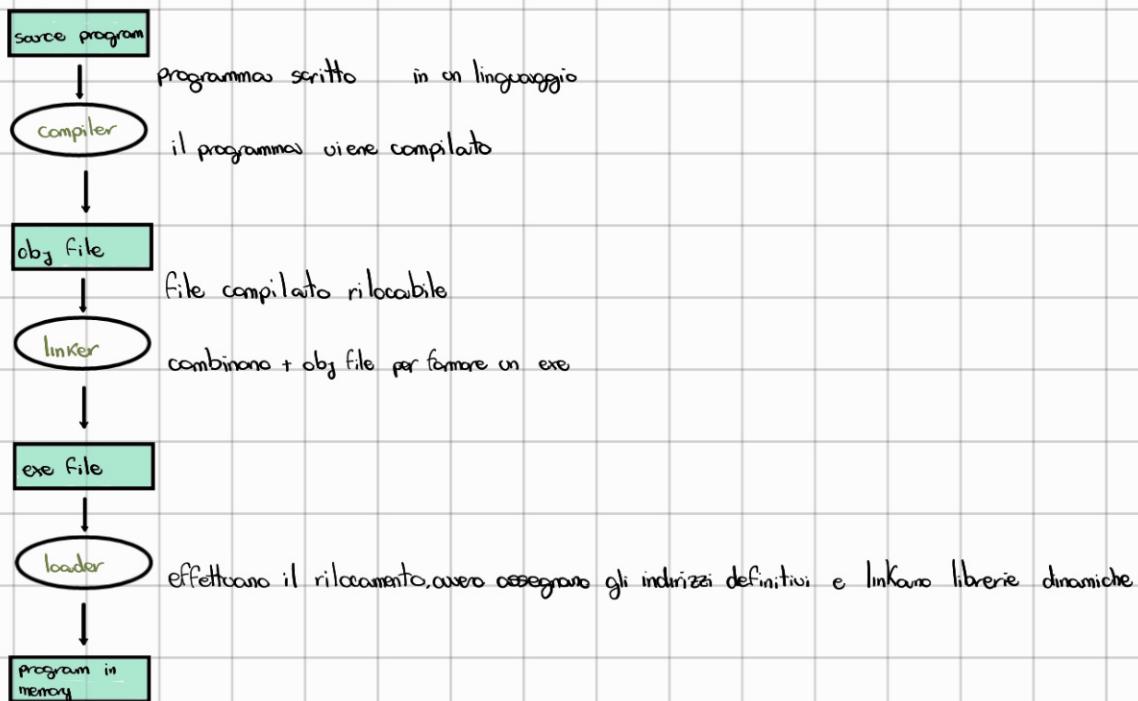
### Chiamate di sistema ed API

Le chiamate di sistema sono funzioni invocabili in un determinato linguaggio. (create da Kernel)

I programmi utilizzano le Application Program Interface (API) gestite dal middleware. Questo per:

- Sono standardizzate
- Sono stabili rispetto alle versioni del SO.
- Offrono servizi più ad alto livello e semplici.

### Loader e Linker



Nei sistemi ad-hoc non tutto il linking avviene al compile-time: le librerie dinamiche avvengono quando viene caricato il programma.

Questo permette di condividere le librerie tra vari programmi, riducendo la dimensione di questi ultimi.

Inoltre, se si modificasse la libreria, non bisognerebbe ricompilare tutti gli eseguibili.

# Sistemi Operativi

~Quack



## Capitolo 1: Strutture e Servizi

### Diverse modalità di funzionamento

- **mod utente:** Non si può accedere ai prog. applicativi del Kernel per proteggere quest'ultimo dai prog. in esecuzione
- **mod Kernel:** attraverso un validity bit si può attivare e si può accedere ai determinati luoghi di memoria critica.

Quando si passa da API a un programma di sistema si cambia modalità, con una system call.

Una chiamata di sistema non è semplice come una chiamata di funzione, si necessita di:

- Un numero che identifica quale chiamata va effettuata.
- Tutti i parametri necessari alla chiamata.

Viene generata un'eccezione che permette di passare il controllo a una subroutine ad un indirizzo preciso in memoria. (mod Kernel)

La system call interface salta a tale indirizzo e viene eseguita la routine.

Al ritorno si torna in user mode.

Essendo un'eccezione, il passaggio parametri è complesso. 3 modi:

- nei registri del processore: + rapido, - parametri di dim limitata
- indirizzo ad un blocco nella quale salvo i parametri: + Linux utilizza in combo con il se push dei parametri sullo stack e la syscall recupera i parametri + flessibile - tanto e macchinoso  
n.b.: ogni programma ha 2 stack: + kernel e + User. ↓  
dove recuperarlo da memoria

### Application binary interface (ABI)

Si vuole che con aggiornamenti del so, non bisogna ricompilare/linkare le app se le API restano uguali.

Si rende dinamica le API. Bisogna però non cambiare ABI, cioè le convenzioni tra binario dell'app e dell'API.

Se invece cambiano le API la questione è + complessa.

# Sistemi Operativi

~Quack



## Capitolo 1: Strutture e Servizi

### Eseguibili binari

3 modi per avere programmi portabili:

- scrivo in linguaggio portabile (Python) eseguibile = sorgente
- scrivo in linguaggio con ambiente run-time portabile (Java) eseguibile = bytecode
- scrivo in linguaggio con compilatore portabile e API standard eseguibile = binario

Nei primi due casi: l'exe è uno uguale per tutte le architetture

Nel terzo caso bisogna generare un exe distinto x architetture (anche x s.o.). Questo perché:

- diff. architettura hardware
- se archi uguali, per le API utilizzate
- se archi e API uguali, per formati binari usati
- se archi, API e formati uguali, per syscall implementate
- se archi, API, formati e syscall uguali, per ABI diverse

# Sistemi Operativi

~Quack



## Capitolo 1: Strutture e Servizi

### Programmi di Sistema

I prog. di sistema permettono l'utilizzo dei servizi del SO. Le tipologie sono:

- **Interfaccia utente**: permette interazione con utente. Può essere grafico (GUI), riga di comando (CLI) o touch screen
- **Gestione File**: creazione, cancellazione, modifica di directory
- **Modifica File**: editor testo, programmi per manipolazione file
- Visualizzazione e modifica info dello stato: data, ora, memoria ecc...
- **Caricamento ed esecuzione programmi**: loader assoluti e ricaricabili, linker, debugger
- **Ambienti di supporto**: compilatori, assemblatori, interpreti per linguaggi.
- **Comunicazione**: connessione tra utenti, programmi e sistemi.
- **Servizi in background**: Verifica stato dischi, scheduling di jobs, logging ecc.

**Interfacce utente**: l'interprete dei comandi permette input, esempio **shell**. Due modi per implementare un comando:

- **Built-in**: interprete esegue comando (Windows)
- **Prog. di sistema**: interprete manda in esecuzione il programma (Unix)

Le GUI sono basate sull'idea di scrivania, semplificano l'uso attraverso icone.

Le TUI sono usate nei dispositivi mobili, no puntatori, tastiere virtuali.

Gli utenti esperti utilizzano le CLI: + rapide, programmabili, accessi a tutto.

I sistemi Windows e Mac sono stati lungamente vincolati da GUI, con limitato cui (ora è cambiato)

L'implementazione di prog. di sistema avviene mediante API.

# Sistemi Operativi



~Quack

## Capitolo 1: Strutture e Servizi

### Sottosistemi Kernel

Sono basati sulle categorie dei servizi del kernel stesso

Servizi offerti:

- gestione processi e thread
- comunicazione tra processi e sincronizzazione
- gestione memoria
- gestione I/O
- file system

Il Kernel del SO è un programma:

- di dim. elevate e complesso
- deve operare velocemente
- Un bug può causare un crash del sistema di elaborazione

Viene progettato in queste possibilità:

- Struttura monolitica, + semplice e veloce, ma essendo unico è pesante e ad ogni modifica devo ricompilare (Windows)
- Struttura a strati, dove liv. n usav n-3 pochi lo usano in maniera pura, ma in alcune zone tipo file system
- Struttura microkernel, poche funzionalità nel Kernel e si mettono in user. (es. 5° windows)
  - + facile estensione so, pochi bug. - overhead, latenza
- Struttura a moduli, componenti dinamicamente caricabili che parlano attraverso GUI. È un ibrido tra microkernel e a strati (accelerata mai meno isolamento)
- Sistemi ibridi: quasi tutti i sistemi usano un mixto.

# Sistemi Operativi

~Quack



## Capitolo 1: Strutture e Servizi

### Politiche e meccanismi

Una politica dice quando una certa operazione viene effettuata

Un meccanismo spiega come una certa operazione è effettuata

Le politiche influenzano le caratteristiche percepite dal sistema di operazione.

I meccanismi sono più stabili, mentre le politiche cambiano spesso per evitare la maledizione delle generalità

# Sistemi Operativi

~Quack



## Capitolo 2: Processi e Thread

Il numero di programmi sono maggiori di quanti processori ha un PC solitamente. Il sistema mette a disposizione un'astrazione detta **processo**, un'entità che esegue un programma.

n.b: programma ≠ processo. un processo (attivo) esegue un programma (passivo). Un programma può essere eseguito da più processi.

L'obiettivo del SO è mantenere impegnate le CPU sui programmi dando l'illusione che ogni programma ha un processore dedicato.

Si introduce la **multi-programmazione** e **multitasking**.

- multi-programmazione: impedire ad un programma che non può proseguire l'esecuzione mantenere la CPU.
- multitasking: far sì che un programma interattivo reagisca ad input velocemente.

### Multi programmazione

Il SO mantiene in memoria i processi da eseguire. Se una CPU non è impegnata, le assegna un processo non in esecuzione.

Richiede che tutte le immagini dei processi siano in memoria.

Se processi sono troppi, si usa lo swapping.

Un'altra tecnica è la memoria virtuale dove le immagini non sono completamente in memoria.

Queste tecniche aumentano il grado di multi-programmazione

### Multitasking

È l'estensione del multi prog. dove la CPU viene sottratta periodicamente ed assegnata ad un altro programma, per mantenere parallela l'esecuzione, permettendo ai programmi batch (usare poco I/O)

# Sistemi Operativi



~Quack

## Capitolo 2: Processi e Thread

### Operazioni sui processi

I SO forniscono syscall con le quali i processi possono creare/terminano/manipolare altri processi.

La gerarchia dei processi è:

- Un processo padre genera figli
- Questi a loro volta possono essere padri, generando un albero di

La gerarchia è importante per la condivisione di risorse, esempi:

- Padre e figlio condividono tutto, o figlio riceve un sottoinsieme o nulla.

Stesso discorso per la creazione di spazio di indirizzi

- Figlio duplicato del padre oppure no e specifico cosa esegue il figlio.

E le coordinazioni:

- Padre sospeso se i figli non terminano, oppure in parallelo + attesa dei figli, o indipendenti,
  - Forza la terminazione dei figli, i padri possono richiedere di essere terminati.  
↓  
(effetto a cascata: anche i nipoti ecc (POSIX non ne fa uso))

# Sistemi Operativi

~Quack

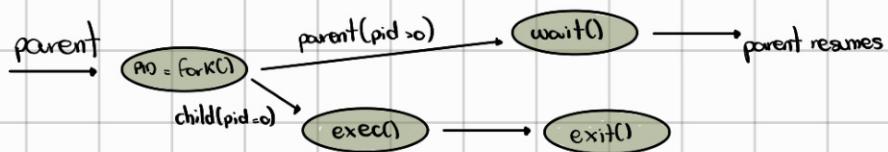


## Capitolo 2: Processi e Thread

Operazioni sui processi.

Ese: API POSIX:

- fork(), crea un figlio
- exec(), sostituzione esecuzione
- wait(), padre aspetta figlio:
  - ritorno PID del figlio terminato
  - codice ritorno figlio
- exit(), termina il processo che lo invoca:
  - 1) accetta un numero
  - 2) se elimina processo e recupera risorse
  - 3) restituisce codice ritorno (se ha invocato wait())
  - 4) Viene invocato implicitamente se esce dal main
- abort(), termina i figli.



Se un processo termina ma il padre non lo sta aspettando (non usa wait()) si dice processo zombie

Se un padre termina prima del figlio, esso è detto orfano

# Sistemi Operativi

~Quack



## Capitolo 2: Processi e Thread

**Struttura Processo:** Un processo è composto da:

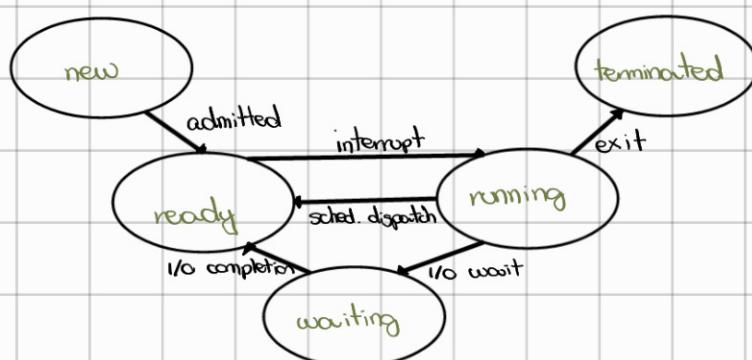
- Stato registri del processore che esegue il programma
- Stato Immagine, ovvero della regione di memoria centrale usata dal programma (processi distinti → immagini distinti)
- Stato processo
- Risorse del SO in uso dal programma (possono essere condivise tra processi)

**Immagine Processo:**

- Text Section: codice programma dim fissa
- Data Section: variabili globali dim fissa
- Stack: per le chiamate dim variabile
- Heap: memoria allocata dinamicamente dim variabile.

**Stati processo:**

- new: creato ma non può eseguire
- ready: attesa assegnazione a CPU
- running: eseguendo
- waiting: in attesa di qualche evento
- terminated: termina esecuzione



**Process Control Block**

- |                          |  |
|--------------------------|--|
| • Stato                  | • Gest. memoria utilizzata                   |
| • Numero: PID            | • Info su I/O utilizzato                     |
| • PC                     | • Info di scheduling: priorità, puntatori... |
| • Registri: il contenuto | • Info accounting: CPU usata, tempo...       |

# Sistemi Operativi

~Quack



## Capitolo 2: Processi e Thread

**Scheduling Processi:** Sceglie il prossimo processo da eseguire. Due code:

- **Ready Queue:** processi ready
- **Wait Queue:** processi in wait. A sua volta ha più code in base al tipo di evento.

**Context switch:** Passaggio tra un processo all'altro.

Se ne occupa il **Dispatcher** che:

- Salva il contesto del processo da interrompere
- Carica il contesto del processo da eseguire

Tempo di context switch = overhead. + complesso SO  $\rightarrow$  + complesso PGS  $\rightarrow$  + overhead. Alcuni processori minimizzano il tempo offrendo banchi di registri multipli.

**Comunicazione interprocesso:** Processi sono indipendenti o cooperare. Cooperano se il suo comportamento influenza o è influenzato da quello di altri. Possibili motivi:

- Condivisione informazioni.
- Accelerazione computazioni.
- Modularità e isolamento

Per poter comunicare tra loro servono le primitive di comunicazione interprocesso (IPC). Due modalità:

- Mem condivisa: comunicazione controllata dal processi. Problema di sincronizzazione, quindi necessitano altre primitive.
- Message Passing: comunicano con mediazione del SO. Permette sincronizzazione. Dopo aver stabilito un link di comunicazione, scambiano messaggi con due operazioni: send e receive.

# Sistemi Operativi

~Quack



## Capitolo 2: Processi e Thread

• **Pipe**: Sono canali di comunicazione tra processi. Possono avere varianti:

- Unidirezionale o bidirezionale.
- Se bidirezionale, half-duplex (uno scrive, l'altro legge) o full-duplex.
- Relazioni tra processi comunicanti.
- Usabili o meno in rete.

Ci sono diverse diverse tipi di pipe:

- **convenzionali**: unidirezionali, condivise con figlio tramite fork(), non usabili in rete. (chiamate anche anonymous)
- **named**: Bidirezionali, esistono anche dopo terminazione del padre.
- **Unix**: Half-duplex, solo su stessa macchina, dati byte-oriented
- **Windows**: Full-duplex, anche tra macchine diverse, message-oriented.

• **Molti threading**: Se un processo può avere molti processori, più istruzioni possono eseguire concorrentemente e quindi più percorsi (**thread**).

I thread condividono memoria globale, il code e le risorse ottenute, ma ciascuno ha il proprio stack.

• **Possibili Problemi nelle API per Threading**: le librerie di thread sono le API fornite al programmatore per creare e gestire thread.

I possibili problemi sono 4:

- Fork duplica solo il thread chiamante o tutti? Alcuni lo fanno due tipi di fork().
- Exec invocata su un thread che effetti ha sugli altri? Di solito termina tutti i thread del processo precedente in esecuzione.
- Quale thread riceve il segnale? Più soluzioni: Ogni thread, alcuni thread, un thread apposito.
- Cancellazione thread? Due approcci: Asincrona dove il thread termina immediatamente, e sincrona, dove si controlla periodicamente se il thread deve terminare.

# Sistemi Operativi

~Quack



## Capitolo 2: Processi e Thread

**Thread Local Storage, ns.**: Può essere comodo avere thread con dei dati locali, ad esempio quando il programmatore non ha controllo diretto sul momento di creazione del thread.

**Implementazione Thread**: Abbiamo due tipi di thread:

- **Livello utente**: sono quelli offerti dalle librerie di thread ai processi.
- **Livello Kernel**: sono utilizzati per strutturare il Kernel stesso in maniera concorrente. Sono utilizzati dalle librerie di thread per implementare quelli a liv. utente. Possono esserci varie strategie:
  - **Molti-a-uno**: usabili su ogni s.o., ma se utente blocca un thread blocca tutti i thread del processo e non sfrutta i più core presenti. Poco usata.
  - **Uno-a-uno**: Maggior grado concorrenza, parallelismo nei sistemi multicore, minori performance del molti-uno e altro stress al liv. Kernel. I più usati.
  - **Molti-a-molti**: combina i vantaggi ma difficile da implementare. Con il modello a due livelli puoi creare degli uno-a-uno con un thread a liv. Kernel.

**Thread Control Block**: analoghi ai PCB, il quale è collegato appunto al PCB dei thread Kernel utilizzati dal processo e viceversa.

**Lightweight process**: interfaccia da Kernel a librerie per usare i Kernel Thread. Associato staticamente a quest'ultimo.

LWP può mandare in esecuzione un thread utente.

**Attivazione Scheduler**: collaborazione tra librerie thread e kernel. Questo comunica alla libreria gli eventi tramite **upcall**.

# Sistemi Operativi

~Quack



## Capitolo 3: Scheduling CPU

### Principi generali dello scheduling

Lo scheduler CPU a breve termine seleziona un processo tra quelli in coda e lo assegna un core ad esso. Ciò può avvenire in diversi momenti:

- running → waiting
- running → ready
- waiting → ready
- termina

Nei casi 1 e 4 si dice senza prelazione (non preemptive) o cooperativo, dal momento che un processo rinuncia al core.

Nei casi 2 e 3 si dice con prelazione (preemptive), dal momento che un core viene sottratto dal Kernel ad un processo che lo sta usando. È più difficile da implementare.

Il dispatcher possiede il controllo da CPU al processo effettuando:

- context switch
- user mode
- jump nel punto corretto del processo.

La latenza di dispatch è il tempo che impiega passando da un processo all'altro

Per max. l'utilizzo della CPU, gli algoritmi di scheduling sfruttano l'esecuzione di un processo che è una sequenza di:

- burst CPU: seq. operazioni CPU
- burst I/O: attesa completamento I/O

# Sistemi Operativi

~Quack



## Capitolo 3: Scheduling CPU

### Programmas I/O bound

- burst CPU brevi elevati
- burst CPU lunghi ridotti
- tipico dei programmi interattivi

### Programmas CPU bound

- burst CPU brevi ridotti
- burst CPU lunghi elevati
- tipico dei programmi batch



n.b. unica differenza:

- I/O bound: max + or sx
- CPU bound: max + or dx

Criteri di scheduling: confrontano diversi algoritmi attraverso:

- Utilizzo CPU: % di tempo in cui CPU attivo (40% - 90%)      ottimale: max
- Throughput: n° di processi che vengono eseguiti in un lasso di tempo      ottimale: max
- Tempo completamento: per un'esecuzione di processo      ottimale: min
- Tempo attesa: tempo di ready queue del processo      ottimale: min
- Tempo risposta: tempo nei processi interattivi tra richiesta e risposta.      ottimale: min

guarderemo questi

# Sistemi Operativi

~Quack



## Capitolo 3: Scheduling CPU

Algoritmi di scheduling. Ci sono diversi algoritmi di scheduling.

• FCFS: vantaggio: implementazione semplice (coda FIFO) svantaggio: tempo medio attesa lungo

SJF: shortest job first, ovvero quello con CPU burst più breve. vantaggio: implementazione come FCFS ma con  $\bar{t}$  attesa minore.

svantaggio: non si sa chi ha CPU burst più breve

• SRTF: shortest remaining time first: utilizza prelazione per gestire i casi dove i processi non arrivano insieme: guarda tra quelli che arrivano chi ha burst CPU + veloce.

$\bar{t}$  attesa: istante terminazione processo - (tempo arrivo + burst)

$t_{compl. processo}$ : istante terminazione processo - tempo arrivo

• RR: round robin, scheduling circolare: ogni processo ha  $\times$  quanto di tempo di esecuzione. Dopo un in ready queue FIFO.

queue come buffer. Se ci sono  $n$  processi in ready queue e quanto tempo  $q$ :

- nessun p aspetta più di  $q(n-1)$
- ogni p ha  $\geq 1/n$  tempo di CPU

In base a  $q$  succede:

- $q$  elevato: RR tende ad FCFS
- $q$  basso: deve essere + lungo di latenza dispatch

Performance:

- RR ha tempo completamento + alto di SJF
- RR ha tempo risposta medio + basso di SJF
- tempo completamente medio non migliora con aumento di  $q$

# Sistemi Operativi



~ Quack

## Capitolo 3: Scheduling CPU

Scheduling prioritari: ogni p ha un' id. Si esegue quello con priorità più alta (Unix n° + basso, windows + alto)

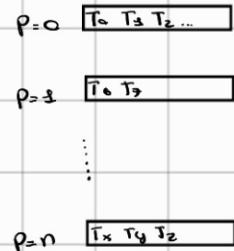
Può essere preemptive o no, possono esserci processi con priorità uguale, nel caso si usa RR.

Problema: attesa infinita starvation: un processo a bassa priorità rischia di non eseguire.

Soluzione: aging, aumento priorità ad aumento tempo attesa

## Code multilivello con retroazione

In PS (priority scheduling) usa ready queue separate.



La assegnazione ai processi avviene così:

- ↑  
• real-time
- system
- interactive
- batch

La priorità può variare: il processo di invecchiamento sposta un processo verso una coda con p maggiore.

Anche l'id. di CPU e I/O bound sono dinamiche.

# Sistemi Operativi

~ Quack



## Capitolo 3: Scheduling CPU

### Scheduling thread.

Se il kernel è multithread, le entità dello scheduler sono thread Kernel.

Per quelli user, possono intervenire due scheduler:

- da librerie dei thread
- da Kernel

Si parla di ambito della contesa di thread utente

- Nei modelli molti-uno e molti-molti, un user thread condivide un LWP(thread Kernel) del processo con altri user thread: **process-contention scope**
- Nei modelli uno-uno un thread user ha LWP non condiviso: **system-contention scope**.

Nel caso **POSIX** la contesa / condivisione LWP regolata dal scheduler di librerie thread, mentre lo scheduler del Kernel si occupa dei processori, tra tutti i Kernel.

# Sistemi Operativi