RSO - Reti e Sistemi Operativi

Sara Angeretti

Ottobre 2023

Indice

Capitolo 1

Organizzazione del corso

1.1 Informazioni sul corso

1.1.1 Lezioni

Crediti: 4 crediti lezione (32 ore) + 4 crediti esercitazione (40 ore).
Due turni

- Turno 1: cognomi dalla A alla L
- Turno 2: cognomi dalla M alla Z

Corso in blended e-learning:

- Lezioni frontali in presenza in aula e da remoto (a causa di ristrutturazione delle aule, tenere sempre controllato il calendario)
- Esercitazioni in e-learning asincrono
- Materiale didattico attraverso il sito del corso (su elearning.unimib.it)
  - Slides e video registrazioni delle lezioni in presenza
  - Materiale video e testuale erogato in e-learning
  - Quiz di autovalutazione
  - Materiale di approfondimento (non oggetto di esame)
  - Indispensabile l'interazione attraverso i forum con docenti, esercitatori e compagni

1.1.2 Personale del corso

Docenti:

- Pietro Braione: docente per la parte di sistemi operativi e responsabile del corso
- Marco Savi: docente per la parte di reti

Esercitatori:

- Jacopo Maltagliati: esercitatore per la parte di sistemi operativi
- Samuele Redaelli: esercitatore per la parte di reti

Tutor:

- Samuele Redaelli: tutor e-learning

1.1.3 Modalità di svolgimento del corso

Le parti di reti e di sistemi operativi si svolgono in contemporanea.
Il calendario del corso, disponibile sul sito, riporta le date delle lezioni in presenza, in remoto a causa di assenza aula, e le date in cui saranno pubblicati i materiali per l'e-learning asincrono.
Il programma del corso (anch'esso disponibile sul sito) riporta le esatte sezioni dei libri di testo da studiare per ciascun argomento del corso, e la distribuzione degli argomenti sulle due prove in itinere.

1.1.4 Il sito del corso

Strumento indispensabile, dal momento che il corso è in blended e-learning!
Aperte le iscrizioni spontanee (iscrivetevi il prima possibile se non siete già iscritti).
Le notizie sul corso verranno comunicate attraverso il forum avvisi.
I materiali didattici vengono distribuiti attraverso il sito.
Sono a disposizione dei forum anonimi per interagire con docenti, esercitatori e tra di voi, allo scopo di discutere gli argomenti del corso e di chiarirvi i dubbi: usateli!

1.1.5 Appelli d'esame

Cinque (o sei?) appelli:

- Due prove in itinere, la prima a novembre 2023 e la seconda a gennaio 2024.
- Due appelli nella sessione di gennaio/febbraio 2024.
- Due appelli nella sessione di giugno/luglio 2024.
- Un appello (o due?) nella sessione di settembre 2024.
- Si può recuperare una (una sola) prova in itinere nel secondo appello della sessione di gennaio/febbraio 2024.

Modalità d'esame:

- Questionario online svolto su computer in laboratorio.
- Le domande possono essere sia teoriche che esercizi che richiedono calcoli, a scelta multipla oppure domande aperte, in qualunque combinazione.
- Ogni prova d'esame comprende sia domande di reti che domande di sistemi operativi: non è possibile sostenere separatamente le parti di reti e di sistemi operativi!
- Regolamento dettagliato di esame disponibile sulla pagina del corso.

1.1.6 Le tempistiche

Parte Sistemi Operativi

Durata Corso (4 CFU)

- 16 ore di didattica frontale
- 10 ore verranno erogate in presenza (aula), le restanti 6 ore online (sincrono)
- Più due ulteriori lezioni in remoto asincrono
- 20 ore di esercitazioni in blended e-learning
- Video e quiz di autovalutazione caricati sulla pagina Moodle secondo calendario didattico
- Possibile qualche incontro in remoto sincrono e un incontro in presenza fuori orario (per chi è interessato)
- Previste inoltre due sessioni di Q&A prima delle prove in itinere

Orario lezioni: vedi calendario sul sito (verranno comunicate di volta in volta così come se in presenza o da remoto).
Controllare sempre il calendario sul sito per sapere se c'è lezione e quando verranno pubblicati i video/quiz per le esercitazioni!

Parte Reti

Durata Corso (4 CFU)

- 16 ore di didattica frontale (in presenza o da remoto a seconda dei giorni)
- Previste inoltre due sessioni di Q&A prima delle prove in itinere
- 20 ore di esercitazioni in blended e-learning
- Video e quiz di autovalutazione caricati sulla pagina Moodle secondo calendario didattico

Orario lezioni: vedi calendario sul sito (verranno comunicate di volta in volta così come se in presenza o da remoto).
Controllare sempre il calendario sul sito per sapere se c'è lezione e quando verranno pubblicati i video/quiz per le esercitazioni!

1.2 Contatti

Parte Sistemi Operativi

Prof. Pietro Braione.
Ufficio: Edificio U14 (DISCo), secondo piano, stanza 2051.
Email: pietro.braione@unimib.it

- Inserire "[RSO]" prima dell'oggetto dell'email!

Telefono: 0264487915
Orario di ricevimento: appuntamento via email.
Team:

- Jacopo Maltagliati - Esercitatore (email: j.maltagliati@campus.unimib.it)
- Samuele Redaelli - Tutor (email: samuele.redaelli@unimib.it)

Parte Reti

Prof. Marco Savi
Ufficio: Edificio U14 (DISCo), Secondo Piano, Stanza 2035
Email: marco.savi@unimib.it

- Inserire "[RSO]" prima dell'oggetto dell'email!

Telefono: 0264487884
Orario di ricevimento: appuntamento via email
Team:

- Samuele Redaelli - Esercitatore (email: samuele.redaelli@unimib.it)
- Samuele Redaelli - Tutor (email: samuele.redaelli@unimib.it)

1.3 Obiettivi del corso

Parte Sistemi Operativi

Acquisire le conoscenze fondamentali relativi ai sistemi operativi:

- A cosa servono i servizi operativi? Perché sono necessari?
- Che servizi offrono ai programmi e agli utenti di un sistema?
- Come sono implementati i sistemi operativi e i servizi che offrono?

Particolarmente importanti sono le esercitazioni pratiche, nelle quali imparerete ad usare i servizi di un sistema operativo moderno (Linux).

- Necessario un computer laptop
- Sono argomento di esame!

Parte Reti

Acquisire le conoscenze fondamentali per comprendere l'architettura e i protocolli principali delle reti di telecomunicazioni basate sullo stack TCP/IP.

- Lo stack TCP/IP è alla base della quasi totalità dei servizi di comunicazione.

Al termine del corso avrete appreso i principi fondamentali del funzionamento di Internet.

1.4 Materiale e strumenti didattici

Parte Sistemi Operativi

Libro di riferimento:

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Sistemi Operativi - Concetti e Esempi, Decima edizione, Pearson, 2019.
- Versione in inglese: Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating Systems Concepts, 10th Edition, John Wiley and Sons, 2018.

Materiale online su Moodle.

- Slides e registrazioni video delle lezioni
- Quiz di autovalutazione
- Video delle esercitazioni

Forum tematico anonimo di discussione su Moodle (indispensabile per le esercitazioni!)

Parte Reti

Materiale online su Moodle

- Slides ufficiali del libro di riferimento (rivisitate, in inglese)
- Video sulla parte di esercitazioni

Libro di riferimento

- Jim Kurose, Keith Ross, Reti di calcolatori ed Internet - Un approccio top-down, Ottava edizione, Pearson, 2021
- Versione in inglese: Jim Kurose, Keith Ross, Computer Networking - A Top-Down Approach, 8th Edition, Pearson, 2021

Forum su Moodle per la parte di reti (specialmente utile per la parte di esercitazioni...)

1.5 Programma del corso

Parte Sistemi Operativi

Sistemi operativi: struttura e servizi
Servizi:

- Processi e thread: i servizi
- Gestione della memoria: i servizi
- File system: i servizi

Struttura:

- Interfaccia e struttura del kernel
- Processi e thread: la struttura
- Gestione della memoria: la struttura
- File system: la struttura

Parte Reti

Parti specifiche del libro (da Capitolo 1 a Capitolo 6)

- Capitolo 1: Introduzione alle reti di calcolatori e Internet
- Capitolo 2: Livello di applicazione [cenni]
- Capitolo 3: Livello di trasporto
- Capitolo 4: Livello di rete - Piano dei dati
- Capitolo 5: Livello di rete - Piano di trasporto
- Capitolo 6: Livello di collegamento e reti locali

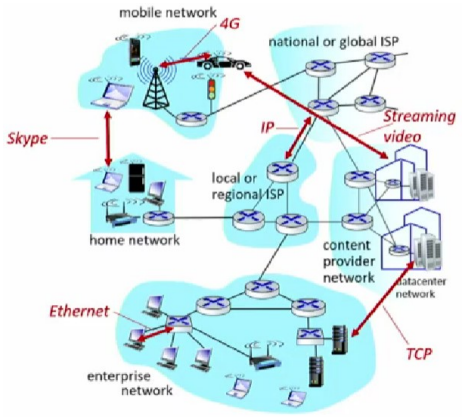
Capitolo 2

Introduzione alle reti

2.1 Cos'è Internet?

Due prospettive:

- Visione degli ingranaggi (dadi e bulloni), delle componenti della rete.
- Visione della rete come un'infrastruttura che fornisce servizi.



Nell'immagine che schematizza, sui bordi della rete troviamo un grandissimo numero di devices (cell, laptops, ...) che eseguono *applicazioni di rete* che richiedono uno scambio di dati e che la rete si occupa di collegare. Alcuni nodi di questa applicazione eseguono parte di un'applicazione e altri nodi interconnessi eseguono altre parti di applicazioni (sistemi distribuiti).

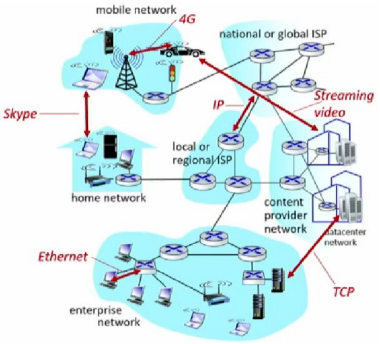
2.1.1 Visione nuts and bolts

hosts: dispositivi *end system* ovvero terminali.

packet switches o *commutatori di pacchetto*: sono *routers* e *switches*, dispositivi che si occupano di trasferire pacchetti (unità) di informazione.
I pacchetti sono unità in cui l'informazione viene scomposta ed etichettata per essere trasferita da un dispositivo all'altro.

Communication links o *collegamenti di comunicazione*: sono i canali che collegano i nodi della rete. Li disegnamo come righe che uniscono i vari nodi. Possiamo crearli usando diversi mezzi trasmissivi, filo di rame, onde radio, fibra ottica... Ognuno di questi link è definito da una **banda**, la quantità di informazione che il link può trasferire in un secondo.

Networks: Internet è una rete di reti, ovvero una rete di dispositivi che sono collegati tra loro e che a loro volta sono reti di dispositivi collegati tra loro. Quindi tutta la rete è una connessione globale di reti locali (es. la rete dell'ufficio, la rete domestica...)



Sempre più dispositivi oggi giorno si sono evoluti fino ad adattarsi all'uso di Internet, come ad esempio le auto, i frigoriferi, le lavatrici...

Quindi anche Internet deve evolversi e adattarsi per poter gestire un numero sempre maggiore di dispositivi con esigenze sempre più diverse.

Internet come rete di reti

Abbiamo detto che Internet è viibile appunto come una rete di reti, un *insieme interconnesso* di **ISPs** (Internet Service Providers), le organizzazioni che gestiscono la rete. A volte ISPs viene usato anche come sinonimo della rete vera e propria.
Gli ISPs comunicano fra loro tramite l'uso di **protocolli**. I protocolli nella rete sono ovunque, controllano il modo in cui i messaggi vengono inviati e ricevuti tra i dispositivi. Alcuni esempi sono HTTP (Web), streaming video, Skype, TCP, IP, WiFi, 4G, Ethernet...
Questi protocolli si basano su **standards**.
Un lavoro fondamentale a riguardo è svolto da **enti di standardizzazione** (il più famoso è IETF, *Internet Engineering Task Force*) che stilano documenti (RFC, *Request for Comments*) che spiegano come i protocolli devono essere implementati e come i dispositivi che implementano questi protocolli devono

comportarsi.
Gli standard sono fondamentali per la comunicazione univoca tra i dispositivi, senza di essi non ci sarebbe interoperabilità tra i dispositivi.

2.1.2 Visione come infrastruttura di servizi

Internet può essere visto come un'**infrastruttura** che fornisce **servizi** alle applicazioni distribuite.
È una vista importante perché la rete sottostante è fondamentale per quando dobbiamo per esempio programmare delle applicazioni o dei servizi: da questo pov è molto importante il concetto di socket, un'interfaccia che ci permette di interfacciarci alla rete senza necessariamente sapere come la rete sotto funziona.

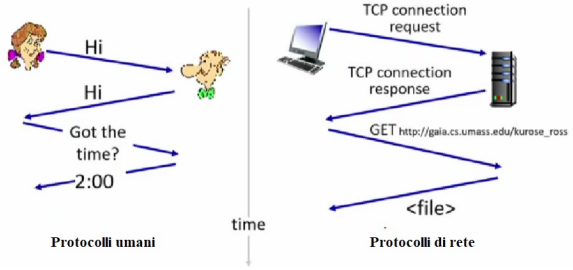
2.2 Protocolli

2.2.1 Cos'è un protocollo?

Protocolli umani	**Protocolli di rete**
I protocolli umani sono le regole che seguono gli esseri umani quando comunicano tra loro. Es.: incontro una persona, scambio di convenevoli, poi chiedo: "Che ore sono?" e in base alla risposta ho delle reazioni diverse. Altro es.: "Ho una domanda" durante una lezione, così che per non interrompere il professore, lui possa finire il suo discorso e poi rispondere alla domanda. Altro es.: un giro di tavolo per fare le presentazioni, seguo delle convenzioni di discorso.	Emulano il funzionamento dei protocolli umani. Tutte le comunicazioni di rete sono gestite da protocolli.

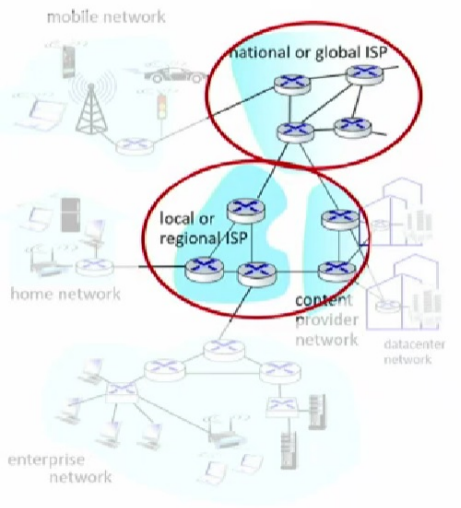
DEFINIZIONE

I protocolli di rete definiscono le regole per i messaggi inviati in rete. In particolare definiscono il *formato dei messaggi*, *l'ordine* in cui vengono *inviati* e *ricevuti* tra le entità di rete (host, commutatori, ...) e le *azioni da intraprendere* una volta che questi messaggi vengono inviati e ricevuti.



2.3 Da cosa è composta la rete?

Andiamo più nel dettaglio.



Network edge

Primo segmento della struttura generale della rete. Ci troviamo sui bordi della rete (quindi c'è un po' una diatriba sul fatto che appartengano o meno alla rete, in ogni caso sono molto importanti), dove si trovano gli **hosts** (client, server). Sono intesi in senso un po' lato, ovvero:

- I *client* sono intesi come dispositivi con una *bassa* capacità computazionale.
- I *server* sono intesi come dispositivi con una *alta* capacità computazionale. Infatti di solito si trovano nei data center.

Access networks

Addentrandoci ancora più nella rete, abbiamo le reti di accesso. Tipicamente hanno dei collegamenti di comunicazione che possono essere *wired* (cavi, stoppini, fibre ...) e *wireless* (4G, onde radio...). Sono molto importanti perché accolgono il traffico dagli utenti e lo portano verso la rete. O viceversa accolgono il traffico di dati da passare al client. Per accedere alla rete bisogna acquistare un accesso alla rete per mezzo di un provider, un ISP.
Sono importanti perché evolvono molto rapidamente nel tempo, sostengono sempre più il cambiamento e l'evoluzione del traffico generato dagli utenti a loro volta in costante evoluzione.
Non parleremo delle reti di accesso.

Network core

La rete di core è una rete con una maglia di dispositivi solitamente molto performanti (quindi in grado di gestire una gran quantità di informazione) e che si

trovano al centro della rete. Permettono di implementare quella rete di reti di cui abbiamo parlato prima.
Per mezzo delle reti di core garantisco che ogni utente che si trova in rete possa comunicare con ogni altro utente che si trova nella stessa rete. Ovviamente non è sempre concretamente possibile.
Internet è una rete connessa: da un nodo posso raggiungere un qualsiasi altro nodo. Questo è quello di cui si occupano le reti core: permettono di mettere in comunicazione reti più al limitare della rete di altre.

2.3.1 Network edge

L'host, abbiamo detto, ha il compito di inviare pacchetti di dati, prende un messaggio applicativo che deve inviare e lo scompone in pacchetti di lunghezza pari ad L (per semplicità ora li consideriamo tutti della stessa lunghezza, ma di solito hanno dimensioni di lunghezza variabile). Questi pacchetti vengono inviati dall'host in rete dalle reti di accesso che forniscono l'accesso ad un rate trasmissivo (banda, capacità...) pari a R (è in bit/s). Questo vuol dire che l'host può inviare pacchetti di lunghezza L a velocità R.

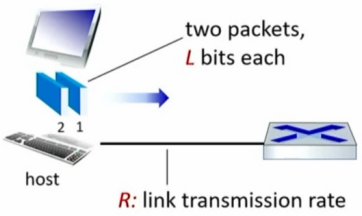
Ovviamente più è alta la banda, più è alta la velocità di trasmissione.

Ma come si calcola il **ritardo di trasmissione**?

DEFINIZIONE

Il *ritardo di trasmissione* è il ritardo che intercorre tra l'invio del primo bit di un pacchetto di L bit alla ricezione dall'altro lato del link dell'ultimo bit dello stesso pacchetto.

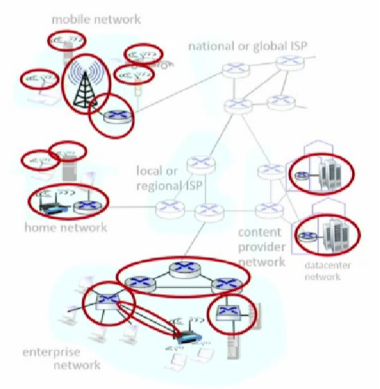
Essendo L la lunghezza del pacchetto e R la velocità di trasmissione, il *ritardo di trasmissione* sarà pari a L/R.



2.3.2 Access networks

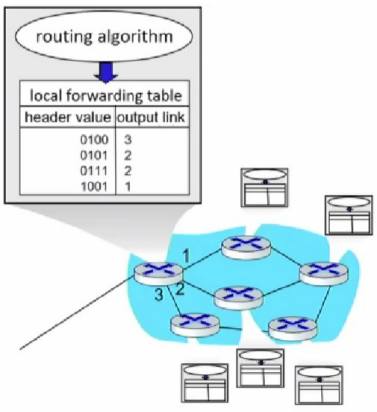
Abbiamo detto che Internet è un *packet switching network*, una rete a commutazione di pacchetto. L'unità informativa che viene scambiata in rete è il **pacchetto** che appunto viaggiano in rete, ricevuti e trasmessi da più dispositivi.
Il compito della rete è di inoltrare i pacchetti verso i router o i commutatori di pacchetto sul link corretto che va da sorgente a destinazione. Questo è il compito principale dei dispositivi della rete di core: creare un percorso (possibilmente il migliore) fra sorgente e destinazione.

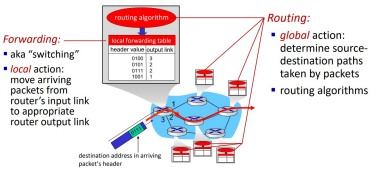
Sembra, almeno concettualmente, abbastanza banale come concetto. Ma la gerarchia della rete è studiata in modo che nodi sufficientemente vicini possano essere messi in contatto senza passare da router o commutatori.



2.3.3 Network core

Come sono fatti i nodi della rete di core:





Forwarding table: è una tabella che contiene le intestazioni dei vari pacchetti (dove c'è specificato l'indirizzo della destinazione e quale interfaccia ovvero l'uscita del link da cui il pacchetto deve uscire) da inviare verso una specifica interfaccia.
L'**inoltro** (forwarding) ha ovviamente valenza locale. Ogni singolo nodo consultando la propria tabella prende decisioni autonome riguardo l'inoltro.

Switching: commutazione, sinonimo di inoltro (forwarding). Le due terminologie hanno una valenza simile ma si riferiscono a due cose leggermente

diverse: l'*inoltro* è l'operazione di ricevere un messaggio e inoltrarlo verso un altro nodo, mentre la *commutazione* è l'operazione per trasferire il pacchetto da un'interfaccia di ingresso ad una di uscita di un router.
Il risultato è lo stesso, ma lo *switching* è più a livello interno del nodo ignorando ciò che sta all'esterno, il *forwarding* invece tiene in considerazione che io prendo e inoltro da un nodo un pacchetto che sposto poi attraverso 1+ link ad un altro nodo.

Routering: ogni singolo nodo prende decisioni localmente, ma io devo garantire che una volta che diversi nodi effettuano l'operazione di inoltro in serie il pacchetto arrivi da sorgente a destinazione senza intoppi. Parliamo di **routing**, instradamento, azione **globale** che ha l'obiettivo di trovare un percorso tra sorgente che invia il pacchetto a destinazione che lo riceve. In ognuno dei nodi avremo un **algoritmo di routing** che agisce in maniera distribuita; un algoritmo di routing comunica con un altro algoritmo di routing tramite protocolli di routing che permettono di popolare la **tabella di forwarding** in maniera corretta.

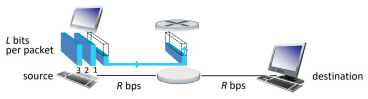
2.4 Modalità packet-switching

2.4.1 Store-and-forward

La rete internet abbiamo detto essere a commutazione di pacchetto e opera secondo la modalità store-and-forward.

Cos'è lo store-and-forward?

È quella modalità per cui un pacchetto prima di poter essere inviato attraverso un link (ad esempio verso un router), deve essere prima inviato nella sua interezza dal nodo da cui arriva. Questo è dovuto al fatto che ogni pacchetto è dotato di un'intestazione che se persa mi fa perdere anche informazione su cosa sia quel pacchetto.
Nell'immagine abbiamo una sorgente, un commutatore e una destinazione.



Se per dire inviassi un bit al commutatore e lui lo spedisse verso la destinazione senza aspettare il resto del pacchetto dalla sorgente, se il bit si perdesse non saprei a chi quel bit appartiene.

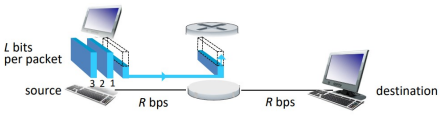
Perché lo store-and-forward è importante?

La modalità store-and-forward è importante quindi perché semplifica notevolmente come le reti debbano essere costruite, prima di inviare un pacchetto devo riceverlo tutto, ricostruirlo, e analizzare l'intestazione per sapere cosa contiene e che è legata a quello specifico pacchetto.

Problemi dello store-and-forward

Introduce dei ritardi, ovviamente, dovendo aspettare che un pacchetto sia ricevuto per intero prima di poterlo inoltrare. Questo ritardo vale $2\frac{L}{R}$ secondi a trasmettere attraverso un link un pacchetto di L bits ad una velocità R di trasmissione. Questo perché impiega $\frac{L}{R}$ secondi per trasmettere il pacchetto al commutatore e altri $\frac{L}{R}$ secondi per trasmettere il pacchetto dal commutatore alla destinazione.
Se invece non usassi questa modalità e ad esempio facessi passare un bit alla volta attraverso il commutatore, impiegherei meno tempo: agirebbe come se il commutatore non esistesse e i due link di velocità R fossero direttamente collegati tra loro, come un unico link più lungo, quindi la velocità complessiva sarebbe comunque R e il tempo di trasmissione **totale** sarebbe $\frac{L}{R}$ secondi. Ovviamente avrei meno ritardi ma altri problemi che vedremo in seguito.

2.4.2 Esempio di esercizio

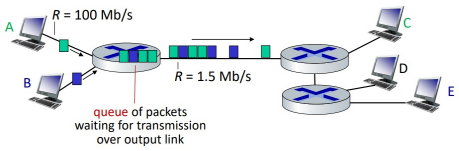


Se come in figura avessi tre pacchetti alla sorgente in attesa di essere inviati (P1, P2 e P3), quanto tempo (in multipli di $\frac{L}{R}$) passerebbe fra l'invio dalla sorgente del primo bit di P1 alla ricezione della destinazione dell'ultimo bit di P3?
La risposta è $4\frac{L}{R}$. Perché?

1. P1 viene inviato al commutatore in $\frac{L}{R}$ secondi.
2. P1 viene inviato alla destinazione e P2 viene inviato al commutatore in $\frac{L}{R}$ secondi.
3. P2 viene inviato alla destinazione e P3 viene inviato al commutatore in $\frac{L}{R}$ secondi.
4. P3 viene inviato alla destinazione in $\frac{L}{R}$ secondi.

2.4.3 Queueing

Abbiamo parlato di possibili problemi riscontrabili con lo store-and-forward. Uno dei principali svantaggi è che in una rete a commutazione di pacchetto si verifica quello che si chiama **accodamento di pacchetti** o **queueing**.

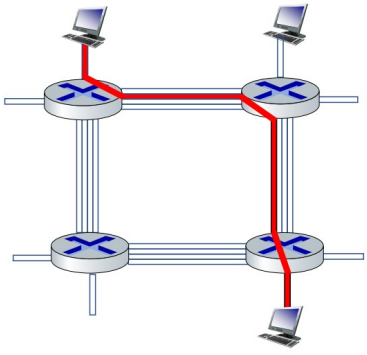


Nella maggior parte dei casi quello che può verificarsi è che il rate a cui io posso trasmettere in uscita sul mio link è più basso rispetto a quello con cui ricevo i pacchetti. Nell'immagine ho un R pari a $100^{Mb}/_{s}$ sui link in ingresso e un R

pari a $1.5^{Mb}/_{s}$ sui link in uscita, *molto* più basso. Arriverò ad un punto in cui non riesco a smaltire i miei pacchetti che si accumulano sul router ad un ritmo sufficientemente sostenuto da tenere il passo del ritmo con cui li ricevo. Allora comninciano ad accumularsi in *buffer*, in zone di memoria, e si crea una coda di pacchetti in attesa di essere trasmessi sul link in uscita.
Queste zone di memoria del commutatore hanno però una capacità limitata (ovviamente) e quando il rate del link in uscita è troppo inferiore rispetto al rate in ingresso al commutatore e quindi si accumulano i pacchetti, rischio di andare in ***buffer* overflow**: è rischioso perché quando lo raggiungo poi perdo i pacchetti che incuranti continuano ad arrivare, e che vengono scartati.
Questo problema è chiamato **il problema della perdita nelle reti a commutazione di pacchetto**.
Un altro problema è il **ritardo di accodamento**.

2.4.4 Alternativa alla modalità packet-switching: circuit-switching

Una rete a commutazione di circuito si basava su principi molto diversi da quelli visti finora.
Al giorno d'oggi non è più usata, ma è importante conoscerla perché è stata la prima modalità di funzionamento delle reti di telecomunicazione. Un esempio è la vecchia rete telefonica (fissa di casa).
Si hanno delle risorse dedicate esclusivamente alla chiamata o al circuito, quando si stabilisce una chiamata si stabilisce un circuito dedicato che rimane dedicato per tutta la durata della chiamata. Una certa quantità di banda viene riservata esclusivamente alla comunicazione: stabilendo un circuito end-to-end di risorse dedicate alla comunicazione.



Vantaggi: non ho i problemi tipo l'accodamento, non ho ritardi di accodamento, non ho perdita di pacchetti. Quindi migliori prestazioni.

Svantaggi: scarsa capacità, ho per esempio nell'immagine un numero massimo di 4 dispositivi collegabili. Quindi peggior utilizzo e scarsa condivisione delle risorse.

2.4.5 Packet-switching vs circuit-switching

Boh non c'è tanto da scrivere, spesso diceva che non lo vediamo nel corso quindi salterei.

2.5 Struttura della rete: reti di reti

Abbiamo detto che gli *hosts* sono connessi da *links* e *ISPs*.
Come più volte abbiamo visto, la rete ha una struttura di tipo **gerarchico**: diversi provider forniscono connettività di tipi diversi. Ci sono nel mondo milioni di reti che fanno da punto di accesso a diversi clients/hosts, e devono essere connesse tra loro.

Prima ipotesi

Creare un punto di accesso verso tutte le altre reti di accesso. Non è affatto fattibile!! Non scala: il numero di collegamenti diretti che io dovrei avere fra le reti di accesso è dell'ordine di $O(n^2)$.

Seconda ipotesi

Creare un rete di un ISP che connette tutte le reti di accesso. Questo è quello che succede oggi giorno. Si interconnette a tutte le reti di accesso e grazie ad una rete di commutatori permette la commutazione fra tutte quelle reti di accesso. Si crea una rete in cui ogni rete di accesso paga l'ISP per poter avere l'accesso ed essere messo in comunicazione con tutte le altre.

2.5.1 Problemi e soluzioni

Una rete così fatta dovrebbe essere geometricamente estesissima, e questo non è fattibile. Si va quindi a creare una rete di reti di ISP, geometricamente più limitate ma anche numerose. Così una rete può scegliere da chi comprare l'accesso.
Il problema che sorge ora è che se una rete di ISP non è in comunicazione con un'altra rete di ISP, come faccio a mettere in comunicazione due reti di accesso che sono collegate a due reti di ISP diverse?
Si vanno a creare allora dei link che le mettano in comunicazione, collegamenti:

peering link: link fra pari, tra un ISP di una rete di ISP e un altro di un'altra rete.
Vantaggioso ad entrambi.
Due tipi diversi:

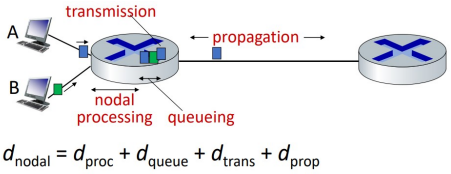
- **regional ISP:** collegamento fra **una** ISP e diverse reti di accesso a cui fornisce accesso.
- **Multi-homing:** collegamento fra **diverse** ISP e diverse reti di accesso a cui fornisce accesso, migliore perché nel caso cada la connessione con un ISP ci sono gli altri a mantenere attiva la connessione.

IXP: internet exchange point, interconnessioni fra ISP di reti diverse. Punti di contatto fra diversi ISPs.
Il MIX è quello di Milano e il più importante d'Italia.

Content provider network: esempio Google, straordinariamente geograficamente estesa in modo da prendere quanti più ISPs possibili. Ha un vantaggio per tutti, comprese le reti di accesso, hanno tutti accessi privilegiati.

2.5.2 Ritardi introdotti dalla packet-switching

4 tipi diversi di ritardo:



Il ritardo di queueing è l'unico che può variare notevolmente da pacchetto a pacchetto. Gli altri sono (all'incirca) costanti per pacchetti diversi. Uno che non abbiamo visto è il **ritardo di elaborazione al nodo**. Ritardo introdotto dal commutatore per effettuare alcune operazioni, tra cui la lettura della tabella di inoltro (*look up* la *table*) per recuperare le informazioni sul link di output. Ogni volta che un pacchetto arriva al commutatore, il commutatore deve analizzare l'intestazione del pacchetto per capire se ci sono stati problemi di trasmissione. Ci sono tutti dei codici/meccanismi per controllare ed eventualmente correggere questi eventuali errori (se non li correggo rischio di perdere il pacchetto, se li correggo lo posso salvare). Questo prende tempo, che diventa il **ritardo di elaborazione (d_{proc})**, questo tempo è dell'ordine di grandezza di 10^{-9} secondi. Un altro dato decisamente più importante (2-3 ordini di grandezza più grande, $10^{-6} - 10^{-3}$ secondi) è il **ritardo di accodamento (d_{queue})**. L'abbiamo già visto. È il tempo che un pacchetto aspetta sul mezzo di trasmissione di uscita (cavo) prima di essere trasmesso, dipende dal livello di congestione del router. Poi abbiamo il **ritardo di trasmissione ($d_{trans} = {}^{L}/_{R}$)**, anche questo già visto, che dipende dalla lunghezza del pacchetto ($L, bits$) e velocità di trasmissione (R, bps). Poi abbiamo il **ritardo di propagazione ($d_{prop} = {}^{d}/_{s}$)** tempo che un pacchetto impiega praticamente a fluire da un'estremità all'altra del pacchetto. Dipende dal mezzo di trasmissione, quindi lunghezza del cavo (d) e velocità di propagazione (s, circa pari alla velocità della luce, $\sim 2 \times 10^8 \frac{m}{sec}$).
A sommare tutte queste componenti, ottengo il tempo complessivo che il pacchetto ci mette a passare da sorgente a destinazione.

Recap:

Nome del ritardo	Sigla	Tempo
R. di elaborazione	d_{proc}	10^{-9} sec
R. di accodamento	d_{queue}	$10^{-6} - 10^{-3}$ sec
R. di trasmissione	d_{trans}	${}^{L}/_{R}$
R. di propagazione	d_{prop}	${}^{d}/_{s}$

Dove:

L:	lunghezza del pacchetto	$[bit]$
R:	velocità di trasmissione	$[{}^{bit}/_{sec}]$
d:	lunghezza del mezzo	$[m]$
s:	velocità di propagazione	$[{}^{m}/_{sec}] = \sim 2 \times 10^8 {}^{m}/_{sec}$

Una cosa fondamentale è la differenza fra *ritardo di trasmissione* e *ritardo di propagazione*: entrambi dipendono dal mezzo su cui il pacchetto viene trasferito, ma sono diversi. Il primo dipende dalla dimensione del pacchetto e il rate di trasmissione del commutatore; il secondo dalla lunghezza del mezzo propagativo (cavo, fibra ottica, whatever) e la sua capacità di propagazione, che è una velocità ma non la stessa dell'altro ritardo.

Es.

Immaginiamo di essere in tangenziale. Arrivo alla barriera e mi fermo al casello di boh Sesto, poi devo arrivare alla barriera di Legnano. Immaginiamo di avere un certo numero di macchine (i bit) che fanno parte di uno stesso pacchetto.
Per semplicità un solo casello. La differenza fra r. di trasmissione e r. di propagazione è: mettiamo di avere un casellante che fa passare 4 macchine al minuto, che portano ad un ritardo di trasmissione R_T (che ovviamente diminuisce all'aumentare della capacità del trasmittente). Se il casellante si sbriga, passano più macchine (più bit al minuto) e quindi il ritardo di trasmissione è più basso. Ovviamente c'è un limite fisico di velocità. Una volta che il casellante mi fa passare, io devo percorrere la tangenziale fino al casello successivo, che è un ritardo di propagazione R_P che dipende dalla lunghezza della tangenziale e dalla velocità massima che posso raggiungere.

2.5.3 Ritardo *end-to-end*

È il ritardo che intercorre per il processing fra sorgente e destinazione, introdotto da tutti i commutatori di pacchetto fra uno e l'altro e i ritardi introdotti dai sistemi periferici.
Abbiamo detto che è la somma delle 4 sorgenti di ritardo: $d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$.
Quando parliamo di *ping* però facciamo riferimento al tempo che ci mette un pacchetto a passare da sorgente a destinazione e di nuovo indietro da destinazione alla sorgente, prende il nome di ritardo *round-trip-time*.

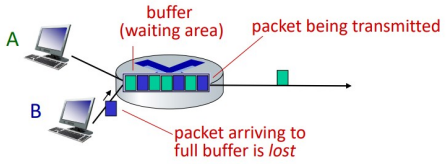
2.5.4 Di più sul ritardo di queueing

Il ritardo di queueing cambia da pacchetto a pacchetto.
Nel caso più generico possibile dipende dal tasso di **intensità di traffico**. Come si calcola? Con strumenti statistici. Abbiamo un rate medio di arrivo dei bit ($\bar{a}$), le lunghezze dei pacchetti (L) e la velocità di trasmissione del mezzo (R).
L'intensità di traffico è definita come $L \cdot \bar{a} / R$.
Se l'intensità di traffico tende a 0, allora il ritardo di queueing tende ad essere molto piccolo.
Se l'intensità di traffico tende a 1, allora il ritardo di queueing tende ad essere larghino.
Se l'intensità di traffico è maggiore di 1, allora il ritardo di queueing tende a infinito ed è ingestibile.
N.B.: stiamo parlando di valori medi!!!

2.5.5 Perdita di pacchetti

Più cause:

- capacità del commutatore satura (buffer overflow) che porta alla perdita di pacchetti inviati dopo che la capacità finita è stata riempita.
- problemi sul mezzo trasmissivo



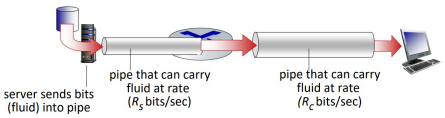
I pacchetti con errori di trasmissione vengono scartati. Potrebbe succedere che in qualche modo si è perso il pacchetto e decidere di ri-inviarlo ma non è scontato.

2.5.6 Throughput

Calcolato come una velocità, rate (bits/time unit) a cui i bits sono inviati da sorgente a destinazione (a volte chiamato ***end-to-end throughput***, anche se questi due termini sono abbastanza simili e quindi possono essere usati singolarmente) ed è:

istantaneo: rate ad un certo punto dato nel tempo

medio: rate in un periodo più esteso di tempo



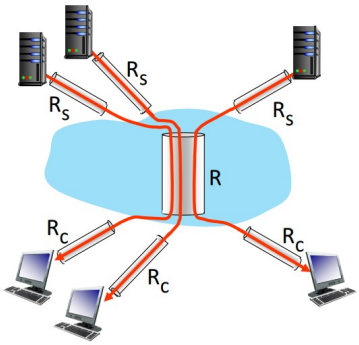
Domande:

$R_s < R_c$**:** quale è il throughput medio?
R_s perché il rate di trasmissione è più basso del rate di ricezione.
In questo caso non c'è accodamento sul commutatore.

$R_s > R_c$**:** quale è il throughput medio?
R_c perché il rate di trasmissione è più alto del rate di ricezione.
In questo caso c'è accodamento sul commutatore.

Parliamo di ***bottleneck link***, che è quel link sul percorso tra sorgente e destinazione (ovvero il percorso end-to-end) che vincola il throughput end-end. Nel primo caso è R_s, nel secondo è R_c.

Throughput: network scenario



Mettiamo di avere una rete come nella situazione in immagine. Abbiamo un certo numero (qua 10) di server, sopra nell'immagine, che acquistano un accesso alla rete per mezzo di un provider di una rete di accesso e hanno quindi link con velocità R_s. Immaginiamo vogliano comunicare con i client (diciamo ancora 10) che si trovano in basso, e che hanno accesso alla rete per mezzo di altri link con dimensione R_c.
Immaginiamo ora di modellare la rete di core come un link di dimensione R (oppure, posso dire che nella rete di core ho un link di dimensione R condiviso da tutte le connessioni in modo **fair**, ovvero equamente, quindi con rate $R/10$). Il throughput sarà dato dal minimo fra questi, ovvero $min(R_c, R_s, R/10)$.
Difficilmente $R/10$ sarà il bottleneck link: è decisamente sottodimensionato rispetto agli altri, tipicamente il bottleneck link sarà R_c o R_s.

Capitolo 3

Reti e livelli di rete

3.1 Architettura a strati

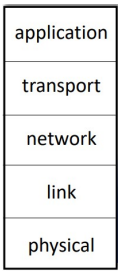
N.B: livelli e strati sono sinonimi qua. Partiamo dal presupposto che le reti hanno una struttura complessa. Esiste una possibilità di studiare e organizzare questa struttura? Un modo è quello di suddividere la complessa struttura in strati, ognuno dei quali si occupa di servizi diversi ed è in comunicazione solo con quelli ***immediatamente*** **adiacenti**.

Perché fare strati?

La stratificazione di un sistema permette di andare a definire un modello di riferimento e permette un'identificazione semplice delle relazioni dei vari pezzi del nostro sistema. La modellizzazione è vantaggiosa perché se vado a modificare l'implementazione di una componente, questo risulta essere trasparente a tutto il sistema e non va ad impattare sull'implementazione degli altri.

3.1.1 Architettura a strati dell'Internet

Anche nota come *Layered Internet protocol stack*, lo stack protocollare si legge dall'alto verso il basso.



applicativo: supporta applicazioni di rete.
Es.: HTTP (non vedremo), SMTP (protocollo per l'invio di mail), DNS (vedremo fra un attimo).

trasporto: trasferisce i dati tra processi diversi eseguiti (di solito) su macchine diverse.
Es.: TCP, UDP.

rete: instrada i datagrammi (pacchetti a livello di rete) da sorgente a destinazione.
Es.: IP, routing protocols.

link: trasferisce i dati tra nodi, elementi di rete, adiacenti.
Es.: Ethernet, 802.11 (WiFi).

fisico: trasferisce i bit su un canale fisico.
È il livello che non vedremo.

3.2 Servizi, stratificazione, incapsulamento

In che modo posso andare effettivamente ad implementare i vari servizi nei vari livelli?

3.2.1 A livello di sorgente

I livelli **applicativi** si scambiano messaggi per implementare specifici servizi.
Questi messaggi vengono trasportati sfruttando i servizi offerti dai livelli di **trasporto**. Questi livelli di *trasporto* hanno il compito di far comunicare diversi processi su macchine diverse e possono trasferire i messaggi (nel caso di TCP) in maniera affidabile, garantendo che il messaggio arrivi in maniera corretta a destinazione.

Incapsulamento

Per implementare i servizi del livello di trasporto si fa in questo modo:

- Prendo i bit del mio messaggio (M) e ci attacco un'intestazione chiamata ***Header*** (H_t), insieme di bit organizzati in **campi**.
- Il messaggio (M) prende il nome di ***payload*** (strato applicativo).
- Il messaggio così ottenuto ($H_t + M$), che prende il nome di ***segmento*** (strato di trasporto), viene passato al *livello di rete* che è immediatamente sotto.
- Lo strato di rete (*network*) prende il *segmento* e ci aggiunge un **suo header**, il messaggio diventa così $H_n + H_t + M$, prende il nome di ***datagram*** e viene passato al livello di link.
- Lo strato di link prende il *datagramma* e ci aggiunge un **suo header**, il messaggio diventa così $H_l + H_n + H_t + M$, prende il nome di ***frame*** *(trama)* e viene passato al livello fisico.

Così si crea la separazione fra i livelli, ogni livello vede i pochi bit passati dal livello precedente.
Alla fine arrivo ad avere un messaggio molto più lungo di quello originale, ma che contiene tutte le informazioni necessarie non ai fini del messaggio ma piuttosto per il funzionamento del sistema. Tutta la parte in eccesso (oltre al *payload* M) prende il nome di ***overhead*** ($H_l + H_n + H_t$). In breve, l'incapsulamento, prende e aggiunge l'intestazione (unica cosa che guarderò) senza preoccuparmi di ciò che ci sta prima (payload). Il succo è che ad ogni livello aggiungi un header.
Il messaggio non viene inviato finché non è stato completamente incapsulato. Deve attraversare tutto il percorso stratificato fino ad arrivare allo strato fisico e poi viene inviato al destinatario.

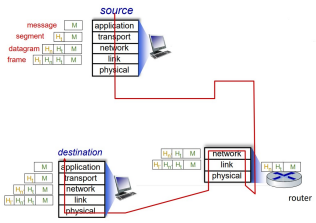
3.2.2 A livello di destinazione

Percorso inverso, partendo dal livello fisico e continuando fino ad arrivare al livello applicativo, si tolgono gli header e si ottiene il messaggio originale.

- Lo strato di link prende il **frame** e toglie il suo header H_l, il **frame** diventa quindi $H_n + H_t + M$ e viene passato allo strato di rete.
- Lo strato di rete prende il **datagram** e toglie il suo header H_n, il **datagram** diventa quindi $H_t + M$ e viene passato allo strato di trasporto.
- Lo strato di trasporto prende il **segmento** e toglie il suo header H_t, il **segmento** diventa quindi M ovvero il mio *payload* e viene passato allo strato applicativo.

A livello di commutatori

Negli *end-host* (sistemi periferici) ovviamente ci sono **tutti e 5** gli strati, a livello di commutatori invece è diverso: ci sono solo i primi 2 (switch) o 3 strati (router), ovvero il *livello fisico*, quello *di link* e quello *di rete*.
I router sono dotati di *commutatori di livello 3*.
Es. semplice:



In questo caso:

- il *frame* $(H_l + H_n + H_t + M)$ creato dal mittente passa al router, che abbiamo detto avere *solo strato fisico, di link, di rete.*
- Il router toglie l'header (H_l) del livello *di link* e inoltra il messaggio al livello *di rete.*
- Il livello *di rete* prende il messaggio $(H_n + H_t + M)$ e lo incapsula con un header diverso (che noi chiamiamo H'_n).
- Il *datagramma* così ottenuto $(H'_n + H_n + H_t + M)$ viene incapsulato in una nuova *trama* e inviato al destinatario.

Quindi in due punti diversi del percorso lo stesso messaggio potrebbe essere incapsulato in due header diversi, e questo succede perché magari in quei due punti del percorso vengono usati due protocolli internet diversi.

Capitolo 4

Livello applicativo

Vedremo ben poca roba, verrà approfondito meglio in altri corsi.

4.1 DNS: Domain Name System

Funzione importante del protocollo internet, implementato a livello applicativo. La sua complessità per questo motivo è limitata ai bordi della rete (a livello applicativo sono coinvolti sono gli end-users, clients e servers, non commutatori). Così come le persone hanno degli identificatori (quali nome, cognome, carta d'identità, codice fiscale, passaporto etc.), anche in rete avviene la stessa cosa. In particolare ci serve:

- host name (es. "www.unimib.it")
- indirizzo IP (indirizzo di 32 bit, tecnicamente associato all'interfaccia e non al server).

Quando scrivo e invio un host name, quello che concretamente faccio è generare una richiesta di DNS che mi porti a risolvere quell'host name. Questo è ciò che fa il DNS: permette di restituire sulla base dell'host name l'indirizzo IP su cui io posso reperire quello specifico contenuto a livello di interfaccia di rete del server.

4.1.1 Mapping tra indirizzi IP e host names

Per mezzo del **Domain Name System** (DNS) è possibile mappare gli indirizzi IP con gli host names.

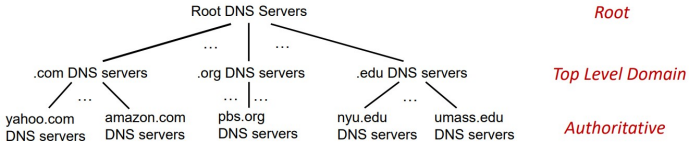
4.1.2 Cos'è un DNS?

È un database distribuito implementato seguendo una gerarchia e utilizzando diversi server che prendono il nome di ***DNS*** *(o* ***nameserver****).*
Il servizio di traslazione degli indirizzi non è facilmente implementabile tramite un unico server (che per ogni singolo host name restituisce l'indirizzo IP associato), non sarebbe scalabile (in internet ho una quantità gigantesca di host names, una singola entità non potrebbe possibilmente risolverli tutti).

Per questo è stata creata la gerarchia per distribuire questo carico su più nodi con diverse responsabilità.

4.1.3 Come funziona un DNS?

Struttura gerarchica dei DNS



Abbiamo:

root DNS servers: sono al top della gerarchia, gestiti da ICANN (Internet Corporation for Assigned Names and Numbers).
Sono una funzione internet importantissima. Contengono tutte le informazioni relative ai top-level domain (TLD) servers o i DNS servers che possono essere contattati.

top-level domain (TLD) DNS servers: gestiscono i domini di primo livello (es. .com, .org, .net, .it, .fr, .uk, .de, ...).
Due casi:

- organizzazioni grandi gestiscono e mantengono in modo diretto i propri server autoritativi (es.: Bicocca)
- organizzazioni piccole si appoggiano a provider di servizi DNS

authoritative DNS servers: organizzazioni che gestiscono i propri DNS (es. Bicocca).
Sono quelli che vanno a fare il mapping vero e proprio.

local DNS name servers: ISP, università, aziende, home network che forniscono accesso Internet ai client.

Local DNS name servers

Quando gli host fanno query DNS, queste vengono inoltrate a un **local DNS server**.

- i DNS locali, se la cache è vuota, creano l'associazione nome-indirizzo che viene storata e conservata per un limitato intervallo di tempo
- altrimenti (se non trova riscontro) interrogano la gerarchia e rispondono direttamente alle query che hanno già in cache, magari non con valori aggiornati ma che comunque non si discostano tanto ma che ne so senti la registrazione

Come funziona la risoluzione di DNS

iterativa: quella che vedremo
Il ruolo centrale ce l'ha il DNS locale che va a contattare gli altri della gerarchia e scopre quale server successivo da contattare per ottenere il vero indirizzo IP da cui ottenere l'informazione richiesta. Tutto trasparente per

il server ma non per il requesting host.
Meglio quella iterativa perché il DNS locale gestisce meglio.

ricorsiva: quella che non vedremo (no shit, CoP)
Si richiede più carico ai DNS all'interno della gerarchia e si toglie un po' di carico a livello del DNS locale.
Non è l'ideale perché vado ad aggiungere complessità ai messaggi che devono essere scambiati fra questi nodi e vista la grande richiesta di scambio di messaggi sulla rete è un casino.

Capitolo 5

Strato di trasporto

5.1 Servizi e protocolli di trasporto

Forniscono una comunicazione logica tra **processi applicativi** che risiedono su host diversi.

Cos'è una comunicazione logica? Fa sì che i processi che stanno comunicando lo facciano, si scambino cioè dati, come se fossero sullo stesso host (in realtà non lo sono). Di fatto *trasparente al livello superiore*. Come il livello applicativo, interpretato esclusivamente dagli host (utenti periferici), non dai dispositivi rete (switch, router, etc.).

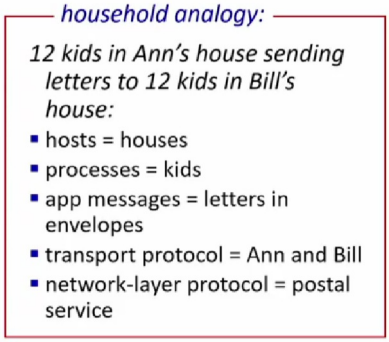
Come si comporta: spezzetta l'informazione in segmenti che trasporta poi a livello di rete.

Due tipi che vedremo: i due principali protocolli a livello di trasporto che sono disponibili per le applicazioni internet sono:

TCP: Transmission Control Protocol
Funzionalità aggiuntiva: *affidabilità*. I messaggi vengono inviati e se arrivano bene, se non arrivano vengono ritrasmessi.

UDP: User Datagram Protocol
Best-effort: I messaggi vengono inviati, se arrivano bene, se non arrivano va bene lo stesso.

Es. pratico:



5.1.1 Differenze livello di trasporto e livello di rete

l. di rete: comunicazione logica fra host

l. di trasporto: comunicazione logica fra processi

5.1.2 Socket

Quando si parla di protocolli a livello di trasporto si deve introdurre il concetto di ***socket***. Quando un processo deve comunicare con un altro processo, deve avere un'interfaccia di comunicazione. Questa interfaccia è il socket.

DEFINIZIONE

Una socket è un'interfaccia verso cui vengono spinti e ricevuti i messaggi che devono essere ricevuti da/inviati a un processo.

Permette di mettere in comunicazione i processi eseguiti a livello applicativo e il livello di trasporto.
N.B.: un processo può gestire più di una socket contemporaneamente.
Avremo bisogno di

- 1 socket per **sender**
- 1 socket per **receiver**

La pila protocollare da livello fisico a livello di trasporto è gestita dal *sistema operativo*. Lo sviluppatore può programmare quella parte di socket esposta verso il processo applicativo.
Per ricapitolare, il messaggio farà: `livello applicativo` → `socket` → `livello di trasporto` → `livello di rete` → `livello fisico` → `livello fisico` → `livello di rete` → `livello di trasporto` → `socket` → `livello applicativo`.

5.2 I due principali protolli Internet a livello di trasporto

5.2.1 TCP, Transmission Control Protocol

Si assicura che tutti i *segmenti* di informazione vengano consegnati *correttamente* e *in ordine*.
Garantisce la consegna affidabile e la connessione affidabile fra mittente e destinatario.
Offre altri servizi:

controllo della congestione: per evitare di contribuire alla congestione della rete fa capire al mittente di diminuire il numero di segmenti inviati.

controllo di flusso: non riguarda la rete ma il destinatario. Se il destinatario non è in grado di ricevere i segmenti, il mittente non li invia.

è connection-oriented: chiede che vengano istituite connessioni fra le parti prima di poter inviare i dati.

È molto importante per le *applicazioni elastiche*, che non sono in alcun modo tolleranti per quanto riguarda la perdita di dati ma molto più tolleranti per quanto riguarda i ritardi. TCP è perfetta.
Nessuno di questi due protocolli garantisce:

- un limite sui ritardi
- garanzie sulla banda (es. non posso dire "ho almeno 10 Mbit di banda per questa connessione").

TCP, principi della trasmissione affidabile dei dati

Cosa ci serve? Un canale di comunicazione affidabile. Ma nella realtà affidabile non lo è mai. A livello di trasporto bisogna implementare una tecnologia per rendere affidabile la comunicazione.
Servono servizi di feedback, il destinatario deve dirmi se gli sono arrivati i dati. Ho perso cose.

5.2.2 Reliable data transfer

Raffigurato c'è un diagramma da spiegare (inserisci qua screen). A sinistra ho l'asse dei tempi, a sinistra avrò `A` e a destra `B` e `A` comunica con `B`. Da `A` a `B` ho frecce che vanno in diagonale perché? Senti.
Consideriamo un modello di rete ideale, ovvero:

- non ci sono errori di bit
- perdite di segmenti/pacchetti
- ritardi di trasmissione dei segmenti/pacchetti

Così diventa superfluo il modello di reliable data transfer perché la rete è già ottimale e affidabile, nessun bisogno di meccanismi di feedback etc.
Non è così ovviamente in verità, quindi bisogna implementare un reliable data transfer.
Avendo errori, una cosa che posso fare è introdurre *due messaggi*:

positive acknowledgment: `ACK` (messaggio di conferma positiva)

negative acknowledgment `NACK` (messaggio di conferma negativa)

```
IF ack,
THEN M_(i+1)
ELSE IF nack
    THEN M_i
    ELSE ?
```

Non funziona, perché anche `ACK` e `NACK` possono essere soggetti ad errore.
Se i miei segmenti *fossero numerati*, riesco a capire se ho perso qualcosa e anche nel caso di duplicati.

```
IF ack
THEN M_(i+1)
ELSE M_i
```

Se vado a numerare anche gli *acknowledgments*, riesco anche ad evitare l'uso di `NACK`.
Nel momento in cu ricevo un ACK_i per il segmento `M_i`, so che tutti i segmenti precedenti sono stati ricevuti correttamente. Se ricevo un ACK_i per il segmento `M_i`, ovvero mando l'`ack` del segmento precedente, posso capire che il segmento `M_i` è stato perso.

```
IF ack
THEN M_(i+1)
ELSE M_i
```

Le reti però non introducono solo perdite di segmenti, ma anche ritardi.
La perdita può capitare al messaggio che sto inviando o all'ack che sto ricevendo.
Come gestisco situazioni in cui li perdo entrambi? Devo introdurre un *timer*.
Parliamo di protocolli ***stop-and-wait***.

Come settiamo questo timer? È una cosa complicata ma fondamentale. Se è troppo alto, se mando un time out spreco un sacco di tempo. Se è troppo breve, rischio di creare ritrasmissioni inutili, invece di usare il mio canale per comunicare normalmente lo riempio di messaggi inutili. Nelle slide un esempio di ciò, timer troppo breve.

Misurare la performance

Transmitting a segment at a time and waiting for its ack before a further transmission (stopand-wait) significantly limits performance.
Example: $RTT = 100ms, L = 1kbyte, R = 100Mbit/s \rightarrow U = 0,008$.

Sliding windows protocols

Sliding window protocols can transit up to W segments while waiting the ack of the first one Ci sarebbe da rivedere l'inizio di sta lezione.
Qua ho aumentando molto l'efficienza dei protocolli stop-and-wait. Se vado a dimensionare la finestra, posso avere un numero di segmenti che posso inviare prima di ricevere l'ack, quindi molta informazione utile.
Perché la trasmissione sia continua (ed evitare gli spazi vuoti) il tempo di invio del primo segmento + il tempo di ricezione dell'ack del primo segmento deve essere minore del tempo di trasmissione di W segmenti. Ovvero:

$$w \bullet \frac{L}{R} \geq RTT + \frac{L}{R}$$

$$W \geq \frac{RTT \bullet R}{L} + 1$$

Go-back-N protocol

Cosa ritrasmetto in caso di errore?

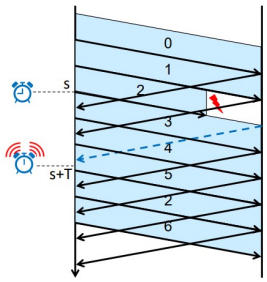
Go-back-N: ritrasmetto tutti i segmenti a partire da quello che ha avuto problemi.
Ma come funziona il Go-back-N?

Lato ricevitore: se ho problemi al segmento 2 (come da immagine), il segmento 3 lo scarto; poi dal segmento 4 mando ack negativi per dire che ho perso qualcosa prima e scarta i segmenti che riceve. Nell'immagine infatti vedo che dopo aver scartato il 5, inizia ad accettare il 2 quando viene ritrasmesso.

Lato mittente: quando riceve un ack negativo, ritrasmette tutti i segmenti a partire da quello che ha avuto problemi. Comunque riguarda la registrazione qua.

Non c'è bisogno di implementare un buffering al ricevitore.
Si dice che gli *ack* sono *cumulativi*. Nel momento in cui non ho un segmento mando un ack per l'ultimo segmento ricevuto e uno negativo per tutti i segmenti che non ho ricevuto.
Per come è fatto il protocollo, ho un solo timer, ogni volta che viene ricevuto correttamente un ack viene resettato e riportato a 0. Questo semplifica di molto l'implementazione.

Selective repeat: ritrasmetto solo il segmento che ha avuto problemi.



Gli ack **non** sono cumulativi. Quando scade il timer, ritrasmetto solo il segmento che ha portato al timeout, ovvero alla scadenza di quel timer.

5.2.3 TCP protocol

Overview

point-to-point: un mittente, un destinatario

affidabile, byte-stream in ordine: garantisce che i dati arrivino correttamente e in ordine, non ci sono "message boundaries"

full duplex data: flow di dati in entrambe le direzioni nella stessa connessione (MSS: maximum segment size)

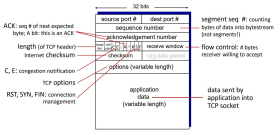
***acks* cumulativi:**

pipelining: TCP congestion and flow control set window size

connection-oriented: handshaking (exchange of control messages) initializes sender, receiver state before data exchange

flow controlled: sender will not overwhelm receiver

TCP, struttura di un segmento



source port #: + dest. port #: 16 bit + 16 bit

sequence number: 32 bit, numero di sequenza del primo byte di dati nel bytestream

acknowledgment number: 32 bit, numero di sequenza del prossimo byte di dati che il mittente si aspetta di ricevere
Sotto, non ho capito il nome, a sinistra del receive window. Noi non la vedremo, più opzioni con lunghezza variabile, dovrebbe contenere l'*header length* di 4 bit, lunghezza dell'header in parole da 32 bit, poi RST, SYN, FIN che si occupando di connection management

receive window: flow control, # bytes che il ricevente è disposto ad accettare

checksum: NOI NON VEDREMO, 16 bit, controllo degli errori

urgent pointer: NOI NON VEDREMO, 16 bit, se il flag URG è impostato, indica il byte successivo al byte urgente

data: NOI NON VEDREMO, 0 o più byte di dati

TCP numeri di sequenza e ACKs

Sequence numbers: byte stream "number" of first byte in segment's data

Acknowledgements: seq # of next byte expected from other side, cumulative ACK

Q: how receiver handles out-oforder segments?

A: TCP spec doesn't say, - up to implementor

TCP, round trip time e timeout

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque

pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

Il senso di tutto ciò è che RTT non è fisso ma bisogna cercare di settare un timeout sufficiente a permettere al pacchetto di arrivare e al ricevente di rispondere.

Q: how to estimate RTT?

- SampleRTT:measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimated RTT "smoother"
  - average several recent measurements (EstimatedRTT),not just current SampleRTT (quindi una media degli ultimi RTT misurati)

Come lo misuro?

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

senti la lezione per capire l'equazione.
Per il timeout, aggiungo un margine di sicurezza, tempo aggiuntivo per evitare la situazione in cui metto un timeout troppo breve e ritrasmetto inutilmente.

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

Dove DevRTT è il mio margine di sicurezza (perché 4* non lo sa nessuno, neanche il prof, credo non lo dicano proprio le specifiche).
L'intervallo varia anche in base al valore del RTT stimato. N.B.: quando apri una connessione e sei appena partito, non hai nessun RTT stimato, quindi di solito si dà il SampleRTT come valore all'EstimatedRTT.

TCP Sender (simplified)

data received from application create segment with seq # seq # is byte-stream number of first data byte in segment

start timer
- think of timer as for oldest unACKed segment
- expiration interval: TimeoutInterval

timeout
- retransmit segment that caused timeout
- restart timer

ACK received: if ACK acknowledges previously unACKed segments

- update what is known to be ACKed
- start timer if there are still unACKed segments

TCP: retransmission scenarios

Me lo sono persa

TCP: fast retransmit

Me lo sono persa

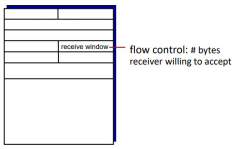
DEFINIZIONE

If sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq # (likely that unACKed segment lost, so don't wait for timeout).

Smallest seq # perché può capitare una perdita multipla.
importantissimo!!! Unico caso in cui non aspetto il timeout per ritrasmettere.

TCP controllo di flusso

Abbiamo detto che avviene nella parte del segmento TCP "receiver window", che si occupa di flow control (# bytes che il ricevente è in grado di accettare).



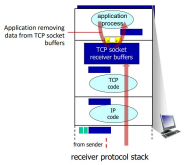
DEFINIZIONE

Flow control: receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast.

receive buffer: fixed amount of buffer space, allocated for a TCP socket

receiver advertises free buffer space by including value of RcvWindow in segment

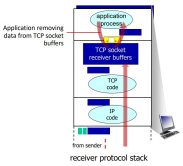
sender limits unACKed data to receiver's RcvWindow



- TCP receiver "advertises" free buffer space in *rwnd* (receive window) field in TCP header (many operating systems autoadjust RcvBuffer)
- sender limits amount of unACKed ("in-flight") data to received *rwnd*
- guarantees receive buffer will not overflow

TCP connection management

before exchanging data, sender/receiver "handshake":



- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

risorse (tipo variabili al centro dell'immagine) da allocare prima dell'handshake.

TCP 3-way handshake, per aprire una connessione

TCP chiudere una connessione

- client, server each close their side of connection

– send TCP segment with FINbit = 1

- respond to received FIN with ACK

– on receiving FIN, ACK can be combined with own FIN

TCP: controllo congestione

DEFINIZIONE

Congestion: informally, "*too many sources sending too much data too fast for network to handle*".

- manifestations: • long delays (queueing in router buffers) • packet loss (buffer overflow at routers) - different from flow control! NB: controllo di flusso esclusivo del ricevente, controllo di congestione è un problema di tutti i nodi della rete. When network gets closer to saturation of available resources, delay and loss percentage increase - If the transport layer

retransmits messages, the average number of messages retransmissions increases too - While throughput (i.e., messages traversing the network) is close to 100% of capacity, "goodput" experienced by the application decreases! Il goodput sono le informazioni che a me effettivamente interessano e droppa così nel grafico perché gran parte delle informazioni che io ricevo sono ritrasmissioni a me non utili.

Approaches towards congestion control

Non andremo a vederla più di tanto e comunque è vagamente fatto da quale parte del segmento TCP?
End-to-end congestion control: - no explicit feedback from network - congestion inferred from observed loss, delay Approaches towards congestion control data data ACKs ACKs - approach taken by TCP

Network-assisted congestion control: - routers provide direct feedback to sending/receiving hosts with flows passing through congested router - may indicate congestion level or explicitly set sending rate

Usa un approccio *Additive Increase Multiplicative Decrease*: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event.

Additive Increase
- increase sending rate by 1
- maximum segment size (MSS)
- every RTT until loss detected

Multiplicative Decrease
- cut sending rate in half at each loss event

Parliamo di *comportamento a dente di sega* di AIDM, vedi il grafico.

TCP sending behavior:

- roughly: send `mincwnd,rwnd` bytes, wait RTT for ACKS, then send more bytes
  - When `cwnd < rwnd`, congestion control is dominant w.r.t. flow control
- TCP sender limits transmission: $LastByteSent - LastByteAcked < mincwnd, rwnd$
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)
- `rwnd` is instead adjusted according to the received value by the receiver (implementing TCP flow control)

Tre fasi:

slow start: quando inizia la connessione, il rate di invio viene aumentato in maniera esponenziale. Nome un po' infelice (per citare il prof) visto che è tutto tranne che slow.

congestion avoindance: Q: when should the exponential increase switch to linear?

A: when cwnd gets to 1/2 of its value before last timeout

Implementation:
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of the value of **cwnd** just before loss event

  TCP Reno: it adopts the Fast Recovery mechanism after three duplicates acks
- set `cwnd` to `ssthresh + 3 MSS` (congestion avoidance)
- same as TCP Tahoe when timeout is reached

5.2.4 UDP, User Datagram Protocol

Non offre nessuno dei servizi di TCP. È un protocollo *best-effort*.
Decisamente meno affidabile di TCP per quanto riguarda perdita di dati e ordine di invio e ricezione.
Meglio per le *applicazioni interattive*, che sono tolleranti per quanto riguarda la perdita di dati ma non tolleranti per quanto riguarda i ritardi. UDP è perfetta.
Protocollo tipico: connectionless. Ogni segmento inoltre gestito in modo indipendente dagli altri. Non è così in TCP.
Ha dei vantaggi:

- non c'è necessità di stabilire una connessione (cosa che introduce ritardi)
- è più semplice (no controllo stato sender/receiver)
- header più piccolo
- non c'è controllo di flusso e di congestione

  Nessuno di questi due protocolli garantisce:
- un limite sui ritardi
- garanzie sulla banda (es. non posso dire "ho almeno 10 Mbit di banda per questa connessione").

UDP segment header

Senti lezione per questo pezzetto. E metti screen.

source port #: 16 bit?

dest. port #: 16 bit?

length: in bytes, lunghezza totale del segmento (header + dati).
16 bit?

checksum: controllo degli errori, wth is he saying, complemento a 1?!
16 bit?

5.2.5 Multiplazione e demultiplazione

Entrambi i protocolli hanno questi (?) servizi (?):

Multiplazione: l'operazione di multiplexing permette di gestire l'invio di più flussi di dati contemporaneamente verso più socket diverse, andando a inserire valori specifici negli header di trasporto. Operazione fondamentale per la *demultiplazione*.

Demultiplazione: comunicazione nella direzione opposta: nello stack protocollare vedo arrivare dati provenienti da processi (veramente socket) differenti. Il receiver ha il compito di andare a vedere (una volta tolti gli header) a quale protocollo mandare i dati.
Usa **indirizzi IP** (livello di *rete*) e **numero di porte** (livello di *trasporto*) per indirizzare i dati alle socket corrette.
L'host riceve l'IP da cui arriva il messaggio.

connectionless demultiplexing: UDP

- Quando creo la socket, devo assegnare un numero di porta #.
- Quando creo il datagram da inviare tramite UDP socket, devo specificare l'intirizzo IP del ricevente e il suo numero di porta #. Servono entrambe!! Questo perché:
  - due host diversi possono avere lo stesso numero di porta #.
  - un host può avere più socket aperte, ognuna con un numero di porta # diverso.
- Quando arriva il datagram, il livello di trasporto estrae l'indirizzo IP e il numero di porta # (li prende dall'header) e li usa per consegnare il datagram alla socket appropriata.

Sulle slide c'è un esempio molto chiaro: inserisci screen.

connection-oriented demultiplexing: TCP
Socket identificata da 4-tuple:

- source IP address.
- source port number.
- dest IP address.
- dest port number.

Sulle slide c'è un esempio molto chiaro: inserisci screen.
Socket strettamente associata alla coppia IP-porta: nell'esempio comunicando da C a B, avrò bisogno di due socket diverse.

Capitolo 6

Strato di rete

6.1 Strato di rete: servizi e protocolli

Obiettivo: prendere i segmenti dal livello di trasporto, incapsularli con un proprio header e inviarli al destinatario.
Ogni internet device (hosts, routers) ha i propri protocolli.
Interpretato da tutti i dispositivi che fanno parte della rete (il livello di trasporto era solo per gli end system).
A livello di routers deve:

- esaminare gli header in tutti i datagrammi ip che passano
- spostare i datagrammi da porta di ingresso a porta di uscita appropriata per trasferire il datagramma lungo il percorso

6.1.1 Funzioni

Forwarding : spostare i pacchetti da ingresso a uscita appropriata

Routing : determinare il percorso seguito dai pacchetti da sorgente a destinazione

6.1.2 Data plane, control plane

Data plane : determina come i pacchetti vengono inoltrati (forwarding)

- local, per-router function
- determines how datagram arriving on router input port is forwarded to router output port

Control plane : determina come i pacchetti vengono instradati (routing)

- network-wide logic, a livello di rete complessiva
- i routers devono comunicare fra loro in modo che il pacchetto segua un percorso end-to-end system
- due approcci principali:

traditional routing algorithms : implementati nei routers, in tutti i routers deve essere implementata una parte relativa al piano di controllo per creare un instradamento più sensato per il pacchetto. La stragrande maggioranza delle reti adotta questo approccio.

software-defined networking : il piano di controllo è separato dal piano di dati, il piano di controllo non è presente nei routers ma in un controller esterno che comunica con i routers. Non la vedremo in questo corso.

Per-router control plane

Individual routing algorithm components in each and every router interact in the control plane.
C'è una separazione tra il piano di controllo e il piano di dati.
A livello di piano di controllo c'è l'algoritmo di routing che agisce in modo distribuito su tutti i routers.
A livello di piano di dati c'è la local forwarding table che agisce in modo locale su ogni router.

Software-Defined Networking (SDN) control plane

Remote controller computes, installs forwarding tables in routers.
Praticamente un remote controller che agisce a livello globale, che comunica singolarmente con tutti i routers.

6.1.3 Network service model

Quale servizio viene offerto a livello di rete?
Abbiamo un tipo di servizio "best-effort", faccio del mio meglio ma se fallisco fa niente.
No guarantees on:

1. successful datagram delivery to destination
2. timing or order of delivery
3. bandwidth available to end-end flow

Overview dell'architettura dei routers

Abbiamo un certo # di porte (o interfacce, sono sinonimi) in input e una serie di porte in output (suddivisione logica, i ruoli sono scambiabili, ma vedremo meglio), separati da un box chiamato high-speed switching fabric (rete hardware molto performante), che ha il compito di commutare le porte di input in porte di output. Contiene una vera e propria rete implementata in hardware.
Perché in hardware? Qua si apre una digressione sulla grande differenza fra piano dati e piano di controllo. Il nostro obiettivo è operare il più velocemente possibile e per fare ciò mi servono implementazioni hardware, che hanno tempi dell'ordine dei nanosecondi. Il routing processor che opera a livello di piano di controllo è a livello di software, quindi ha tempi dell'ordine dei millisecondi.
Immagine.

L'importante di questa high-speed switching fabric è che **non sia bloccante**, ovvero che non ci siano situazioni in cui un pacchetto non può essere inoltrato perché la rete è occupata.
*Le **porte di input** sono formate da*:

- **strato fisico** : riceve stream di bit
- **strato di collegamento** : prende i bit e li raggruppa in frame
- **decentralized switching** :
    - using header field values, lookup output port using forwarding table in input port memory ("match plus action")
        - destination-based forwarding: forward based only on destination IP address (traditional)
    - goal: complete input port processing at 'line speed'
    - input port queuing: if datagrams arrive faster than forwarding rate into switch fabric

Se abbiamo un # di porte di input pari a N, e un # di porte di output pari a N, che ricevono i datagrammi con un rate R, se l'high-speed switching fabric ha un rate pari a $N \cdot R$ (idealmente) allora non ci sono problemi, altrimenti va ridimensionata la switching fabric in modo che le porte non diventino bottleneck.
Un problema grave che porta ad aumentare il tempo che un datagramma accodato ad una rete di accesso è: **Head-of-the-Line (HOL) blocking**.

If switch fabric slower than input ports combined –¿ queueing may occur at input queues (queueing delay and loss due to input buffer overflow!).
Head-of-the-Line (HOL) blocking: queued datagram at front of queue prevents others in queue from moving forward. Manca un'immagine.
*Le **porte di output** sono formate da*:

- **Buffering** required when datagrams arrive from fabric faster than link transmission rate. Drop policy: which datagrams to drop if no free buffers?
    - Questo porta ad una perdita di datagrammi.
- **Scheduling** discipline chooses among queued datagrams for transmission
    - Priority scheduling - who gets best performance

6.1.4 Intestazione protocollo IP

Slide 16.

- **Version** : 4 bit, indica la versione del protocollo IP.
- **Header length** : 4 bit, indica la lunghezza dell'header in parole da 32 bit.
- **Type of service** : 8 bit, indica il tipo di servizio richiesto.
  Non li vedremo, vengono usati per identificare datagrammi con priorità differente.

- **Total length** : 16 bit, indica la lunghezza totale del datagramma in byte.
- **16-bit identifier** : 16 bit, identifica il datagramma.
- **Flags** : 3 bit.
- **Fragment offset** : 13 bit.
  I due sopra si occupano di frammentare il messaggio: fino a poco tempo fa la frammentazione poteva essere fatta anche a livello di rete, ora a livello di trasporto. Non si usa più quindi il libro non lo tratta.
- **Time to live** : 8 bit, indica il numero di router che il datagramma può attraversare prima di essere scartato. Viene decrementato di uno ad ogni router.
  Molto importante perché evita che i datagrammi vadano in loop infinito nel caso in cui non arrivino a destinazione.
- **Upper layer** : 8 bit, indica il protocollo di livello superiore a cui il datagramma deve essere indirizzato una volta incapsulato.
- **Header checksum** : checksum effettuato solo sull'header (a livello di trasporto, effettuato sull'intero segmento).
- **Source IP address** : 32 bit, indirizzo IP del mittente.
- **Destination IP address** : 32 bit, indirizzo IP del destinatario.
- **Options (if any)** : 0 o più opzioni, non usate di frequente. Record route taken.
- **Payload data** : variable length, typically a TCP or UDP segment.

6.2 Indirizzamento IP

6.2.1 Introduzione

- **IP address** : 32-bit identifier associated with each host or router **interface**.
- **interface** : connection between host/router and physical link
  - routers typically have multiple interfaces
    - port and interface are synonyms in our context
  - *host* typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)

Dotted-decimal IP notation

223.1.1.1 = 11011111 00000001 00000001 00000001 da mettere in tabella

Connessione di interfacce

Q: how are interfaces actually connected?
A: we'll partially learn about that in chapter 6. For now: don't need to worry about how one interface is connected to another (with no router in between).

6.2.2 Subnets

- **What's a subnet** ?
    - device interfaces that can physically reach each other without passing through a router
- **IP addresses have structure** :
    - **subnet part** : devices in same subnet have common high order bits
      Anche chiamato **prefisso/prefix**.
    - **host part** : remaining low order bits
- **Recipe for defining subnets** :
    - detach each interface from its host or router, creating "islands" of isolated networks
    - each isolated network is called a subnet
    - what are the /24 subnet addresses?
      Servono a dire quanti bit sono dedicati alla parte di subnet (da sinistra) e quanti alla parte di host.
      A questo proposito parliamo di CIDR.
- **IP addressing: Classless (CIDR)** - CIDR: **C**lassless **I**nter**D**omain **R**outing.
    - subnet portion of address of arbitrary length
    - address format: `a.b.c.d/x`, where x is # bits in subnet portion of address

Ho perso: IP addressing: Classful, Special IP addresses, Destination-based forwarding Dei due esempi sotto (slide 24, destination-based forwarding), il primo matcha con l'interfaccia 0, il secondo con l'interfaccia 1, ma ancje con la 2. Come si fa? Si segue la regola ***longest prefix match***: vince quella che ha un maggior numero di bit che matchano (when looking for forwarding table entry for given destination address, use longest address prefix that matches destination address). In questo caso l'interfaccia 1. Questo è importante perché succede mooolto spesso.

6.2.3 IP addresses: how to get one?

Sorgono due importanti domande:

1.Q: How does a host get IP address within its network (host part of address)?

2.Q: How does a network get IP address for itself (network part of address)?

How does a host get IP address?

- Statically specified in config file of the OS (e.g., /etc/rc.config in UNIX)
- DHCP: Dynamic Host Configuration Protocol: dynamically get address from as server ("plug-and-play")

goal: host dynamically obtains IP address from network server when it "joins" the network
- can renew its lease on address in use
- allows reuse of addresses (only hold address while connected/on)

DHCP overview :
- host broadcasts **DHCP discover** msg [optional]
- DHCP server responds with **DHCP offer** msg [optional]
- host requests IP address: **DHCP request** msg
- DHCP server sends address: **DHCP ack** msg

DHCP can return more than just allocated IP address on subnet:
- name and IP address of the local DNS server
- subnet mask (indicating network versus host portion of address)

Q: how does network get subnet part of IP address?

A: gets allocated portion of its provider ISP's address space

Hierarchical addressing: route aggregation: hierarchical addressing allows efficient advertisement of routing information (route aggregation) ISP1 chiede tutti i pacchetti che cominciano per 200.23.16.0/20, ISP2 chiede tutti i pacchetti che cominciano per 199.31.0.0/16 etc.
A sua volta può spedire i pacchetti verso gli indirizzamenti corretti con una subnet mask di /23.
E se l'organizzazione 1 si spostasse da ISP1 a ISP2?
Organization 1 moves from ISP 1 to ISP 2. ISP 2 now advertises a more specific route to Organization 1.

Q: how does an ISP get block of addresses?

A: ICANN: Internet Corporation for Assigned Names and Numbers (`http://www.icann.org/`)

Q: are there enough 32-bit IP addresses?

- ICANN allocated last chunk of IPv4 addresses in 2011
- NAT (next) helps IPv4 address space exhaustion
- IPv6 has 128-bit address space

6.3 NAT: network address translation

NAT: all devices in local network share just one IPv4 address as far as outside world is concerned.
all datagrams leaving local network have same source NAT IP address: 138.76.29.7, but different source port numbers.
datagrams with source or destination in this network have 10.0.0.0/8 address for source, destination (as usual).
All devices in local network have 32-bit addresses in a "private" IP address space (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 prefixes) that can only be used in local network.
Advantages:

- just one "public" IP address needed from provider ISP for all devices
- can change addresses of host in local network without notifying outside world
- can change ISP without changing addresses of devices in local network
  è possibile cambiare l'ISP senza cambiare gli indirizzi IP della rete privata
- security: devices inside local net not directly addressable, visible by outside world
  utile dal pov della sicurezza perché tutti i dispositivi all'interno della rete privata (a destra del router dell'immagine della slide precedente) non sono direttamente visibili dall'esterno.

6.3.1 Implementation

NAT router must (transparently):

- outgoing datagrams: replace (source IP address, port #) of every outgoing datagram to (NAT IP address, new port #)
  - remote clients/servers will respond using (NAT IP address, new port #) as destination address
- remember (saved in a NAT translation table) every (source IP address, port #) to (NAT IP address, new port #) translation pair
- incoming datagrams: replace (NAT IP address, new port #) in destination fields of every incoming datagram with corresponding (source IP address, port #) stored in NAT table

Per ogni indirizzo sceglie una porta, come informazione inserisc eil destinatario. Quando questo datagramma raggiunge il router, questo lo traduce e lo inoltra al destinatario.
Se non presente, va istituita la NAT translation table.

Mentre lato LAN nel coordinamento non richiede la porta (di solito associato alla coppia indirizzo-porta) il NAT non può assegnare due pseudo porte randomiche a boh chi ci sta capendo niente.
La traslazione avviene tra indirizzo IP e porta in teoria, ma in pratica con la traslazione di indirizzo IP e traslazione della porta riesco a inoltrare questo (quale?) indirizzo.

6.3.2 NAT: osservazioni

NAT has been controversial:

- routers "should" only process up to layer 3
- address "shortage" should be solved by IPv6
- violates end-to-end argument (port # manipulation by network-layer device)

But NAT is here to stay:

- extensively used in home and institutional nets, 4G/5G cellular nets

Capitolo 7

Strato di Rete: Piano di Controllo

Inserisci slide 2.
Finora abbiamo visto la parte di data plane: Ora vedremo la parte di control plane e in particolare di algoritmi di instradamento e protocolli.

7.1 Algoritmi di instradamento e protocolli.

Permettono di determinare rotte/cammini/percorsi migliori (= costo migliore, meno congestionati, . . .) da sorgente a destinazione attraverso una rete di router che devono essere attraversati dai pacchetti.

7.1.1 Astrazione di grafi

Per parlare di algoritmi di routing dobbiamo parlare di astrazione di grafi. Nella slide: vediamo una rete di routers interconnessi da link.
Le interconnessioni fra i nodi A, B, C sono facili, mentre fra B, C, D vedo la presenza di un dispositivo di livello 2, uno switch, su cui torneremo.

Costi dei link

$c_{a,b}$: costo del link che unisce **direttamente** (senza passare da altri router o switch) a e b.
Cost defined by network operator: could always be 1, or inversely related to bandwidth, or inversely related to congestion.

7.1.2 Classificazione degli algoritmi di instradamento

• **Dinamici** : costi cambiano molto velocemente

• **Statici** : i router cambiano lentamente nel tempo

Ortogonalmente:

- **Globali** : c'è un continuo scambio con tutti i nodi della rete
  all routers have complete topology, link cost info ("link state" algorithms)
- **Decentralizzati** : c'è un continuo scambio solo con i propri vicini (localmente), guardando il cammino col costo più basso verso tutti i nodi della rete

Distance Vector (DV)

Si basano sull'algoritmo di Bellman-Ford, che si basa sull'equazione di Bellman-Ford: ipotizziamo di avere sorgente (nodo A) e destinazione (nodo X), e di voler trovarne la distanza. Sarà il cammino con costo minimo fra: link diretto da V a X e cammino tra V e X.

$$dist(A, X) = min_V dist(V, X) + c(A, V)$$

Esempio: So la distanza (cammino con costo *minimo*) fra due nodi, non il percorso effettivamente seguito. Per es. fra W e Z ho un link diretto ma con la strada alternativa non diretta mi costa di più.
L'algoritmo, key idea:

- from time-to-time, each node sends its own distance vector (DV) estimate to neighbors
- when x receives new DV estimate from any neighbor v, it updates its own DV using Bellman-Ford equation:

$$D_x(y) \leftarrow min_v c_{x,v} + D_v(y) for each node y in \mathbb{N}$$

- the estimate $D_x(y)$ converges to the actual least cost $d_x(y)$

State information diffusion:

Link State (LS)

I problemi degli algoritmi dei distance vector è risolta dai link state.
Nei DV vado a calcolare il vettore distanza di tutti i nodi ma lo comunico solo ai miei vicini.
Nei LS invece tutti i nodi verso tutti i nodi informano sullo stato dei link. Possono quindi eseguire l'algoritmo di Dijkstra per trovare il cammino con costo minimo.
Svantaggio: la mole di messaggi scambiati.
Distribution in selective flooding of the information related to the network topology

- Each router must broadcast its link state information to the other routers

Computation of the shortest path using Dijkstra's algorithm.
Mi calcola l'albero dei cammini minimi da un nodo specifico (dal nodo scelto ho indicazione dei cammini minimi per raggiungere un qualsiasi altro nodo).
L'algoritmo:

- **centralized:** network topology, link costs known to all nodes

- accomplished via "link state broadcast"
- all nodes have same info

computes least cost paths from one node ("source") to all other nodes

- gives forwarding table for that node

• **iterative:** after k iterations, know least cost path to k destinations

Notation:

•$c_{x,y}$**:** direct link cost from node x to y; $= \infty$ if not direct neighbors

•$D(v)$**:** current estimate of cost of least-cost-path from source to destination v

•$p(v)$**:** predecessor node along path from source to v

•N'**:** set of nodes whose leastcost-path definitively known

Manca slide 62.
Esempio: slide 63.

Distance Vector vs. Link State

Distance Vector	**Link State**
Simple and intuitive implementation	Complex implementation
DVs sent only to neighbors · Low number of exchanged messages	Selective Flooding · High number of exchanged messages
Slow convergence · And problems in achieving it	Fast convergence · And robust
Examples · RIP · IGRP · EIGRP	Examples · OSPF · IS-IS
BGP protocol uses an algorithms (Path Vector) based on similar principles of those of Distance Vector	

7.1.3 Internet approach to scalable routing

Routers are aggregated into regions known as "autonomous systems" (AS), a.k.a. "domains".

· **Intra-AS (a.k.a. "intra-domain"):** routing within same AS

- all routers in AS must run same intra-domain protocol
- routers in different ASes can run different intra-domain routing protocols

· **Inter-AS (a.k.a. "inter-domain"):** routing among ASes

- **gateway router**: it is at the "edge" of an AS, and has link(s) to router(s) in other ASes
- gateways routers perform inter-domain routing (as well as intra-domain routing)

Capitolo 8

Sistemi operativi: struttura e servizi

Argomenti

- A che servono i sistemi operativi?
- Requisiti per i sistemi operativi
- Struttura e servizi dei sistemi operativi
- Chiamate di sistema ed API
- I programmi di sistema

8.1 I sistemi operativi

Un s.o. è una certa quantità di software che viene dato assieme all'hardware perché quest'ultimo da solo non funziona. Per esempio quando compri un telefonino apri e ti trovi Android o iOS, mentre su un laptop abbiamo Windows, macOS e Linux.
Parliamo del modello teorico della macchina di Von Neumann. Questo modello si basa su cinque componenti fondamentali:

- Unità centrale di elaborazione (**CPU**), che si divide a sua volta in unità aritmetica e logica (ALU o unità di calcolo) e unità di controllo;
- Unità di **memoria**, intesa come memoria di lavoro o memoria principale (RAM, Random Access Memory);
- Unità di **input**, tramite la quale i dati vengono inseriti nel calcolatore per essere elaborati;
- Unità di **output**, necessaria affinché i dati elaborati possano essere restituiti all'operatore;
- **Bus**, un canale che collega tutti i componenti fra loro.

Un computer quando lo accendiamo inizia ad eseguire un programma. Se non c'è un programma da eseguire è solo un mucchio di ferraglia che si blocca e non fa niente. Perciò possiamo dire che un s.o. è il **primo** programma che viene eseguito all'accensione del computer e che permette di eseguire altri programmi.
È una cosa molto diversa dalle *applicazioni*, che sono quelle che "fanno le cose utili" cit prof. Infatti la prima cosa che facciamo quando per esempio compriamo un telefono è installare WhatsApp o altre app che ci servono e rendono comomda la vita. Un s.o. di per sé non è *"utile"*, lo diventa in relazione al funzionamento delle app che ci interessano.
Un s.o. fornisce un ambiente a finestre (laptop) o icone (smartphone) che permette di eseguire le applicazioni utili: le possiamo installare, lanciare, far eseguire (anche più di una contemporaneamente), interromperne l'esecuzione...
La macchina di Von Neumann esegue un programma alla volta, ma noi di solito su un computer eseguiamo più di un'applicazione alla volta (es. WebEx per la lezione e VSCode per gli appunti). Questa cosa ci è permessa dal s.o., che ci permette ci gestire n applicazioni contemporaneamente, anche più dei processori di cui dispone il mio computer. Es.: mettiamo di avere un computer con un processore a 32 core, io però posso eseguire anche centinaia di programmi contemporaneamente, non massimo 32.
Il s.o. crea un'ambiente in cui le applicazioni possono collaborare assieme. Se avessi più applicazioni che usano contemporaneamente lo stesso hardware (es. lo schermo)? Il s.o. si occupa di gestire le risorse hardware e di farle usare alle applicazioni. Il s.o. è il software *intermediario* tra le applicazioni e l'hardware.
Un'altra cosa che fa il s.o. è organizzare i nostri file in un sistema ordinato di file e cartelle (anche memorizzandoli su dispositivi secondari di memoria e storaging). Il s.o. si occupa di gestire i file e le cartelle, di crearli, cancellarli, rinominarli, spostarli...

Cos'è un sistema operativo?

DEFINIZIONE

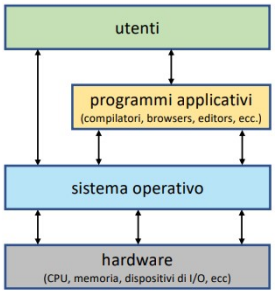
È un insieme di programmi (software) che gestiscono gli elementi fisici di un computer (hardware).

Fornisce una piattaforma di sviluppo per le applicazioni, che permette loro di condividere ed astrarre le risorse hardware.
Agisce da intermediario tra utenti e computer, permettendo agli utenti di controllare l'esecuzione dei programmi applicativi e l'assegnazione delle risorse hardware ad essi.
Protegge le risorse degli utenti (e dei loro programmi) dagli altri utenti (e dai loro programmi) e da eventuali attori esterni.
Un s.o. in primo luogo è una piattaforma di sviluppo, ossia **un insieme di funzionalità software** che i programmi applicativi possono usare.
Tali funzionalità permettono ai programmi di poter usare in maniera conveniente le risorse hardware, e di condividerle:

- Da un lato, il s.o. **astrae** le risorse hardware, presentando agli sviluppatori dei programmi applicativi una visione delle risorse hardware più facile da usare e più potente rispetto alle risorse hardware ≪native≫.

- Dall'altro, il s.o. **condivide** le risorse hardware tra molti programmi contemporaneamente in esecuzione, suddividendole tra i programmi in maniera equa ed efficiente e controllando che questi le usino correttamente.

Componenti di un sistema di elaborazione



Utenti: persone, macchine, altri computer...
Programmi applicativi: risolvono i problemi di calcolo degli utenti.
s.o.: coordina e controlla l'uso delle risorse hardware.
Hardware: risorse di calcolo (CPU, periferiche, memorie di massa...).

8.1.1 Requisiti per i sistemi operativi

Cosa si richiede ad un s.o.?

Oggigiorno i computer sono ovunque: vi sono molteplici tipologie di computer utilizzati in scenari applicativi diversi (i nostri laptop, i nostri smartphones, i computer detti "embedded" che non hanno lo scopo di interagire con persone ma sono "cyber-fisici", cioè che fanno parte di servizi ad es. quelli che controllano le automobili ad es. il sistema ABS che fa in modo che la ruota non si blocchi e quindi non slitti quando noi inchiodiamo e freniamo a fondo, etc...).
In quasi tutti i tipi di computer si tende ad installare un s.o. allo scopo di gestire l'hardware e semplificare la programmazione.
Ma ogni scenario applicativo in cui viene usato un computer richiede che il s.o. che vi viene installato abbia caratteristiche ben determinate (es. laptop molto diverso dai sistemi embedded). Che cosa si richiede ad un s.o. per supportare un certo scenario applicativo?
A seconda che sia:

Server, mainframe: massimizzare la performance, rendere equa la condivisione delle risorse tra molti utenti

Laptop, PC, tablet: massimizzare la facilità d'uso e la produttività della singola persona che lo usa

Dispositivi mobili: ottimizzare i consumi energetici e la connettività

Sistemi embedded: funzionare senza, o con minimo, intervento umano e reagire in tempo reale agli stimoli esterni (interrupt)

La maledizione della generalità

Nella storia (ed anche oggi) alcuni sistemi operativi sono stati utilizzati per scenari applicativi diversi,
Ad esempio, Linux è usato oggi nei server, nei computer desktop e nei dispositivi mobili (come parte di Android).
La **maledizione della generalità** afferma che, se un s.o. deve supportare un insieme di scenari applicativi troppo ampio, non sarà in grado di supportare nessuno di tali scenari particolarmente bene. Praticamente io quando cerco di dare più di un esame per sessione.
Questo si è visto con OS/360, il primo s.o. che doveva supportare una famiglia di computer diversi (la linea 360 dell'IBM).
Quella **maledizione della generalità** non avviene sempre e necessariamente, è un potenziale rischio; può essere tuttavia aggirata, ma non ho capito come.

8.2 Struttura dei sistemi operativi

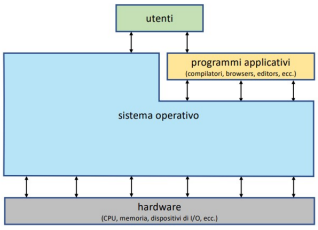
Non c'è una definizione universalmente accettata di quali programmi fanno parte di un s.o..
In generale però un s.o. comprende almeno:

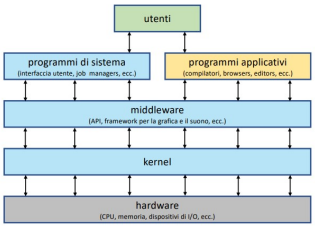
Kernel: il "programma sempre presente", che si "impadronisce" dell'hardware, lo gestisce, ed offre ai programmi i servizi per poterlo usare in maniera condivisa ed astratta.

Middleware: servizi di alto livello che astraggono ulteriormente i servizi del kernel e semplificano la programmazione di applicazioni (API, framework per grafica e per suono...)

Programmi di sistema: non sempre in esecuzione, offrono ulteriori funzionalità di supporto e di interazione utente con il sistema (gestione di jobs e processi, interfaccia utente...)

Alcuni sistemi operativi forniscono anche dei programmi applicativi (editor, word processor, fogli di calcolo...), ma non li considereremo parti del s.o. stesso.





8.3 Servizi dei sistemi operativi

Un s.o. offre un certo numero di **servizi**:

Per i programmi applicativi: perché possano eseguire sul sistema di elaborazione usando le risorse astratte esposte dal s.o..

Per gli utenti: per gestire l'esecuzione dei programmi e stabilire a quali risorse hardware i programmi (e gli altri utenti) hanno diritto.

Per garantire che il sistema di elaborazione funzioni in maniera efficiente.

Gli utenti però interagiscono con il s.o. attraverso i programmi di sistema...
...i quali utilizzano gli stessi servizi dei programmi applicativi...
Quindi, in definitiva, il s.o. ha bisogno di esporre i suoi servizi esclusivamente ai programmi (applicativi o di sistema).

8.3.1 Principali servizi:

Controllo processi: questi servizi permettono di caricare in memoria un programma, eseguirlo, identificare la sua terminazione e registrarne la condizione di terminazione (normale o erronea).

Gestione file: questi servizi permettono di leggere, scrivere, e manipolare files e directory

Gestione dispositivi: questi servizi permettono ai programmi di effettuare operazioni di input/output, ad esempio leggere da/scrivere su un terminale.

Comunicazione tra processi: i programmi in esecuzione possono collaborare tra di loro scambiandosi informazioni: questi servizi permettono ai programmi in esecuzione di comunicare.

Protezione e sicurezza: permette ai proprietari delle informazioni in un sistema multiutente o in rete di controllarne l'uso da parte di altri utenti e di difendere il sistema dagli accessi illegali.

Allocazione delle risorse: alloca le risorse hardware (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione in maniera equa ed efficiente.

Rilevamento errori: gli errori possono avvenire nell'hardware o nel software (es. divisione per zero); quando avvengono il s.o. deve intraprendere opportune azioni (recupero, terminazione del programma o segnalazione della condizione di errore al programma).

Logging: mantiene traccia di quali programmi usano quali risorse, allo scopo di contabilizzarle, ovvero fare sì che un programma o un utente non usi troppe risorse sottraendole ad altri programmi o utenti.

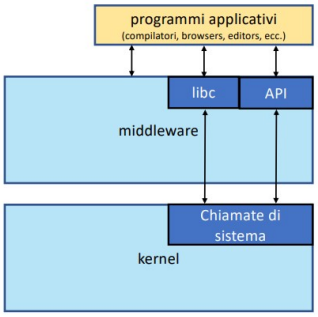
8.4 Chiamate di sistema ed API

8.4.1 Cosa sono?

kernel: programma sempre presente che è la parte centrale del s.o. e che permette di astrarre l'hardware e di fornire servizi ai programmi utente.

Middleware: fornisce le API, le chiamate di procedura che ci permettono di costruire i programmi.

Il kernel fornisce un insieme di servizi detti **chiamate di sistema**, funzioni invocabili in un determinato linguaggio di programmazione (ad es. C, C++...), anche se si tende a **non** invocare direttamente le chiamate di sistema. Ma si tende a frapporre uno strato (il *middleware*) tra il kernel e i programmi applicativi, in modo che questi ultimi chiamino le **API**, funzioni pensate appositamente per permettere ai programmi applicativi di usare i servizi offerti dal kernel in modo migliore rispetto alle chiamate di sistema.
Utili anche perché ogni linguaggio di programmazione ha una sua libreria, suoi oggetti...
Per evitare commistioni (mescolanze, contaminazioni...), si vanno a creare due strati distinti, uno per la libreria del L.d.P. e quello delle API.



Come visibile nell'immagine:

- Strato della libreria del L.d.P. può accedere alle chiamate di sistema.
- Strato della libreria delle API può accedere alle chiamate di sistema.
- Programmi applicativi possono accedere al middleware.
- Programmi applicativi possono accedere alla libreria del L.d.P..
- Programmi applicativi possono accedere alle API.

E basta.

8.4.2 Differenze fra chiamate di sistema ed API

API	Chiamate di sistema
Esposte dal **middleware**	Esposte dal **kernel**
Usano le c.d.s. nella loro implementazione	
Sono standardizzate (es. POSIX e Win32)	Non sono standardizzate (ogni kernel ha le sue)
Sono stabili	Possono variare al variare della versione del s.o.
Funzionalità più ad **alto livello** e più semplici da usare	Funzionalità più **elementari** e più complesse da usare

8.4.3 I programmi di sistema

I programmi di sistema sono programmi che usano le chiamate di sistema per offrire servizi di alto livello agli utenti. I più importanti:

Interfaccia utente (UI): permette agli utenti di interagire con il sistema stesso; può essere grafica (GUI) o a riga di comando(CLI). I sistemi mobili hanno un'interfaccia touch.

UI - l'interprete dei comandi:

Permette agli utenti di interagire con il sistema tramite **istruzioni**, comandi testuali.

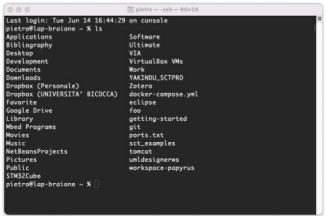
In molti sistemi operativi è possibile configurare quale interprete dei comandi usare, in quel caso è detto **shell**.

Ci sono due modi per implementare un comando:

▷ *Built-in:* l'interprete esegue direttamente il comando (tipico nell'interprete dei comandi di Windows).

▷ *Come programma di sistema:* l'interprete manda in esecuzione il programma (tipico delle shell Unix e Unix-like).

Spesso riconosce un vero e proprio l.d.p. con variabili, condizionali, cicli...

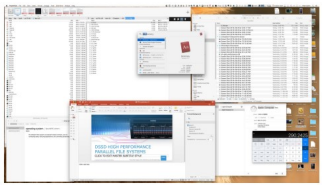


UI - le interfacce grafiche:

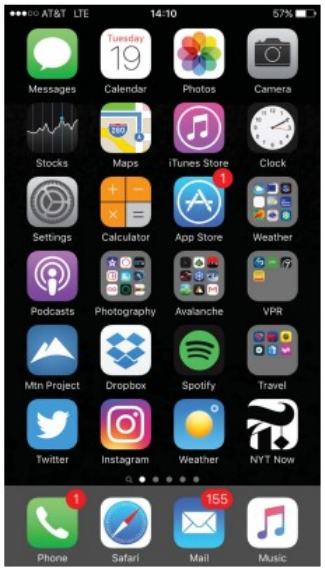
L'interfaccia grafica (GUI) è di solito basata sulla metafora della scrivania, delle icone e delle cartelle (le directory).

Nate dalla ricerca presso lo Xerox PARC lab negli anni '70, sono state popolarizzate poi da Apple con il Macintosh negli anni '80.

Su Linux le più popolari sono GNOME e KDE.



UI - le interfacce touch:
I dispositivi mobili richiedono interfacce di nuovo tipo.
Nessun dispositivo di puntamento (mouse)
Uso dei gesti (gestures)
Tastiere virtuali
Comandi vocali



Gestione file: creazione, modifica e cancellazione file e directory.

Modifica dei file: editor di testo, programmi per la manipolazione del contenuto dei file.
Il confine fra *programmi di sistema* e *programmi applicativi* è spesso labile.

Visualizzazione e modifica informazioni di stato: data, ora, memoria disponibile, processi, utenti. . . fino a informazioni complesse su prestazioni, accessi al sistema e debug. Alcuni sistemi implementano un registry, ovvero un database delle informazioni di configurazione.

Caricamento ed esecuzione dei programmi: loader assoluti e rilocabili, linker, debugger.

Ambienti di supporto alla programmazione: compilatori, assemblatori, debugger, interpreti per diversi linguaggi di programmazione.

Comunicazione: forniscono i meccanismi per creare connessioni tra utenti, programmi e sistemi; permettono di inviare messaggi agli schermi di un altro utente, di navigare il web, di inviare messaggi di posta elettronica, di accedere remotamente ad un altro computer, di trasferire file...

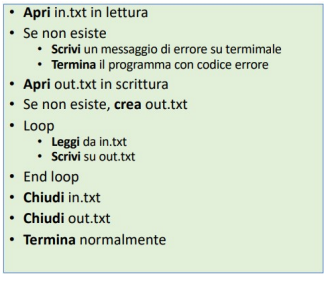
Servizi in background: lanciati all'avvio, alcuni terminano, altri continuano l'esecuzione fino allo shutdown. Forniscono servizi quali verifica dello stato dei dischi, scheduling di jobs, logging...
I servizi in background sono ad esempio i cosiddetti *demoni di sistema* nei sistemi Unix-like, sono programmi di sistema che di solito sono inattivi per la maggior parte del tempo ma attivati all'avvio del sistema e che rimangono in esecuzione fino allo spegnimento del sistema. Rimangono in attesa finché non si verifica un certo evento, compiono una certa operazione e si disattivano di nuovo in attesa del prossimo evento.
Utile per servizi temporali, ad esempio utilissimi per i backup, tipo quelli mensili.

L'implementazione dei programmi di sistema

I programmi di sistema sono implementati utilizzando le API, esattamente come i programmi applicativi.
Consideriamo ad esempio il comando cp delle shell dei sistemi operativi Unix-like:

- cp (copy a file) in.txt out.txt
- Copia il contenuto del file in.txt in un file out.txt
- Se il file out.txt esiste, il contenuto precedente viene cancellato, altrimenti out.txt viene creato

È implementato come programma di sistema tramite API (tutte POSIX).
Una possibile struttura del codice è riportata sulla destra (le invocazione delle API sono riportate in grassetto).



Capitolo 9

Processi e Threads: i servizi

Vedremo sostanzialmente come i s.o. gestiscono **processi** e **thread** e cosa si intende con questi due termini.

9.1 Argomenti

- Concetto di processo, operazioni e API POSIX
- Comunicazione interprocesso e le API POSIX
- Multithreading e le API POSIX

9.2 Concetto di *processo*

9.2.1 Cos'è un processo

Un s.o. ha come obiettivo il fatto di prendere un sistema di elaborazione e fare in modo che su di esso possano essere eseguiti più programmi in modo concorrente. Il loro numero può essere arbitrariamente elevato, di solito molto maggiore del numero di processori, di CPU del sistema. Il s.o. ci permette però di eseguire il n° di programmi che vogliamo indipendentemente dal n° di CPU che abbiamo. Come?
Il s.o. realizza e mette a disposizione una macchina astratta, un'astrazione detta ***processo****.*

DEFINIZIONE

Un **processo** è un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma.

All'interno del s.o. il processo è l'unità esecutiva del programma. Es.: su un pc da 4 CPU, 50 programmi in esecuzione vuole dire 50 processi.
Supporremo **per ora** che l'esecuzione di un processo sia **sequenziale** e quindi *non ci sia concorrenza.*

9.2.2 Programmi e processi

Notare la differenza tra *programma* e *processo*!

- Un programma è un'entità passiva (un insieme di istruzioni, tipicamente contenuto in un file sorgente o eseguibile)
- Un processo è un'entità attiva (è un *esecutore di un programma*, o un *programma in esecuzione*)

Uno *stesso programma* può dare origine a *diversi processi*:

- Diversi utenti eseguono lo stesso programma
- Uno stesso programma viene eseguito più volte, anche contemporaneamente, dallo stesso utente

9.2.3 Struttura di un processo

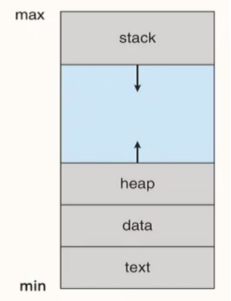
Un processo è composto da diverse parti:

- Lo stato dei registri del processore che esegue il programma, incluso il program counter.
- Lo stato della regione di memoria centrale usata dal programma, o **immagine** del processo.
- Lo stato del processo stesso.
- Le risorse del sistema operativo in uso al programma (files, semafori, regioni di memoria condivisa...).

Processi distinti hanno *immagini distinte* (*isolamento dei processi*).
Le risorse del sistema operativo invece possono essere condivise tra processi (a seconda del tipo di risorsa).

Immagine di un processo

L'intervallo di indirizzi di memoria *min...max* in cui è contenuta l'immagine di un processo è anche detto **spazio di indirizzamento** (*address space*) del processo. Ovviamente ogni processo lavora nel proprio spazio di indirizzamento. L'immagine di un processo è formata da:



- Lo **stack** delle chiamate, contenente parametri, variabili locali e indirizzo di ritorno delle varie procedure che vengono invocate durante l'esecuzione del programma.
- Lo **heap**, contenente la memoria allocata dinamicamente durante l'esecuzione.
- La **data section**, contenente le variabili globali.
- La (**text section**) contiene il codice macchina del programma.

Text e *data section* hanno dimensioni costanti per tutta
la vita del processo.
Stack e *heap* invece crescono/decrescono durante la vita del processo, smetteranno solo quando una delle due sezioni raggiungerà la fine della memoria allocata per il processo, ovvero collidono tra loro.

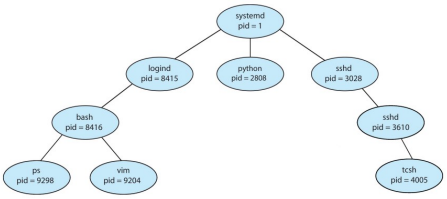
9.3 Operazioni sui processi

I sistemi operativi di solito forniscono delle chiamate di sistema con le quali un processo può creare/terminare/manipolare altri processi.
Dal momento che *solo un processo può creare un altro processo*, all'avvio il sistema operativo crea dei processi "primordiali" dai quali tutti i processi utente e di sistema vengono progressivamente creati.
Vedremo ora le operazioni fondamentali per:

- Creare processi
- Terminare processi

9.3.1 Creazione di processi

Chi crea il primo processo? Il s.o., in fase di booth. Il fatto che un processo crei un altro processo dà origine ad una **gerarchia**. Di solito nei sistemi operativi i processi sono organizzati in maniera gerarchica.
Un processo (padre) può creare altri processi (figli). Questi a loro volta possono essere padri di altri processi figli, creando un albero di processi.
Un albero di processi in Linux:



La relazione padre/figlio è di norma importante per le politiche di condivisione risorse e di coordinazione tra processi.
Possibili politiche di condivisione di risorse:

- Padre e figlio condividono tutte le risorse (**non** l'immagine, abbiamo detto che ogni processo ha la propria immagine) ...
- ... o un opportuno sottoinsieme ...
- ... o nessuna

Possibili politiche di creazione spazio di memoria/indirizzi:

- Il figlio è un duplicato del padre (stessa memoria e programma) ...
- ... oppure no, e bisogna specificare quale programma deve eseguire il figlio
  Questo succede per le API Posix e non per le Win32.

Possibili politiche di coordinazione padre/figli:

- Il padre è sospeso finché i figli non terminano ...
- ... oppure eseguono in maniera concorrente

9.3.2 Terminazione di processi

I processi di regola richiedono esplicitamente la propria terminazione al sistema operativo (ovvero invoca una API che di solito si chiama `exit`).
Un processo padre può attendere o meno la terminazione di un figlio.
Un processo padre può forzare la terminazione di un figlio. È sempre brutto terminare forzatamente un processo (facendolo quindi terminare in maniera asincrona, senza avere quindi il tempo di fare operazioni per uno shut-down ordinato) piuttosto che lasciarlo terminare spontaneamente, ma a volte si rende necessario. Possibili ragioni:

- Il figlio sta usando risorse in eccesso (tempo, memoria...).
- Le funzionalità del figlio non sono più richieste (ma è meglio terminarlo in maniera "ordinata" tramite IPC).
- Il padre termina prima che il figlio termini (in alcuni sistemi operativi).

Riguardo all'ultimo punto, alcuni sistemi operativi non permettono ai processi figli di esistere dopo la terminazione del padre:

- **Terminazione in cascata:** anche i nipoti, pronipoti... devono essere terminati (non c'è nelle API Posix).
- La terminazione viene iniziata dal sistema operativo.

API Posix per operazioni sui processi:

`fork()` crea un nuovo processo figlio; il figlio è un duplicato del padre ed esegue concorrentemente ad esso, ritorna al padre un numero identificatore (PID) del processo figlio e al figlio (che così capisce di essere il figlio) il PID 0.

`exec()` sostituisce il programma in esecuzione da un processo con un altro programma, che viene eseguito dall'inizio; viene tipicamente usata dopo una `fork()` dal figlio per iniziare ad eseguire un programma diverso da quello del padre.

`wait()` viene chiamata dal padre per attendere la fine dell'esecuzione di un figlio (qualsiasi); ritorna:

- Il PID del figlio che è terminato.
- Il codice di ritorno del figlio (passato come parametro alla `exit()`).

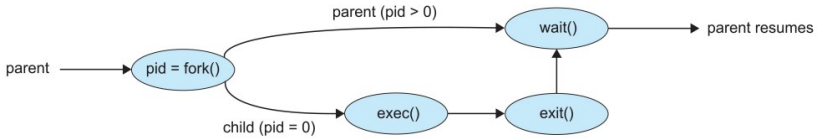
`exit()` fa terminare il processo che la invoca:

- Accetta come parametro un codice di ritorno numerico che dice come è andata
  (es. tipicamente se il figlio ritorna 0 allora è terminato normalmente, se invece è un numero $\neq 0$ c'è un errore di qualche tipo e il padre deve essere in grado di capire e gestire il codice di errore).

- Il sistema operativo elimina il processo e recupera le sue risorse.
- Quindi restituisce al processo padre il codice di ritorno (se ha invocato `wait()`, altrimenti lo memorizza per quando l'invocherà).
- Viene implicitamente invocata se il processo esce dalla funzione `main`.

`abort()` fa terminare forzatamente un processo figlio.

La sequenza `fork-exec` nelle API POSIX:



9.3.3 Processi *zombie* e *orfani*

Ricordiamo che abbiamo detto che nelle API Posix non c'è la terminazione a cascata.

P. zombie: Se un processo termina ma il suo padre non lo sta aspettando (non ha invocato `wait()`) il processo è detto essere ***zombie***. Praticamente il processo ritorna un codice di errore ma non c'è nessuno a prendere e gestire il codice di errore. Il processo viene allora riallocato ma il codice rimane, quindi il processo non è effettivamente morto. Per questo *zombie*.

P. orfano: Se un processo padre termina prima di un suo figlio, il figlio è detto essere ***orfano*** (non vi è terminazione a cascata).

9.4 Comunicazione interprocesso

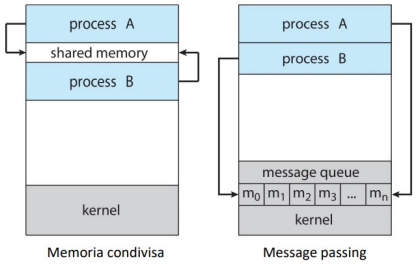
N.B.: quello di cui abbiamo parlato finora non vuol dire "1 processo = 1 applicazione". Es. Chrome, con più tab, **non** è un processo solo, *ogni* tab è un processo. Questo perché si è visto che la navigazione nei browser ha problemi di isolamento: es. metti che apro una tab con il sito di homebanking e una con un link che mi è stato mandato per posta (*LOL*). Che succede? Un bordello.
Allora si è pensato fosse meglio isolare ciò che succede in ogni tab, in modo che una pagina probabilmente compromessa non vada a disturbare le altre. Per fare ciò, si è pensato di applicare un processo distinto ad ogni tab del browser, in modo che ogni immagine a sé stante non possa andare a disturbare le altre.
Utile però sarebbe far comunicare in modo disciplinato fra un processo e l'altro.
Dunque, più processi possono essere indipendenti o cooperare.
Un processo coopera con uno o più altri processi se il suo comportamento "influenza" o "è influenzato da" il comportamento di questi ultimi.
Possibili motivi per avere più processi cooperanti:

- Condivisione informazioni
- Accelerazione computazioni
- Modularità ed isolamento (come in Chrome)

Per permettere ai processi di cooperare il sistema operativo deve mettere a disposizione delle altre API per la comunicazione interprocesso, noi vedremo alcune primitive di comunicazione interprocesso (IPC).
Tra cui queste due categorie:

Memoria condivisa (a sinistra): vuol dire che in qualche modo le due immagini dei processi condividono una certa regione di memoria.

Message passing (a destra): vuol dire che in qualche modo all'interno della memoria del s.o. viene messa una qualche coda di messaggi e quindi un processo può inserire dei messaggi nella coda e l'altro processo può prelevare i messaggi dalla coda.



9.4.1 IPC tramite memoria condivisa

Viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare.
Vedremo come si può implementare quando parleremo di gestione della memoria.
*La comunicazione è controllata dai **processi** che comunicano, **non** dal **s.o.**.*
Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di sincronizzarsi (un processo non deve leggere la memoria condivisa mentre l'altro la sta scrivendo).
Allo scopo i sistemi operativi mettono a disposizione ulteriori primitive per la sincronizzazione.

9.4.2 IPC tramite message passing

Permettono ai processi *sia* di comunicare *che* di sincronizzarsi.
I processi comunicano tra di loro senza condividere memoria, attraverso la mediazione del sistema operativo.
Questo mette a disposizione:

- Un'operazione `send(message)` con la quale un processo può inviare un messaggio ad un altro processo.
- Un'operazione `receive(message)` con la quale un processo può (mettersi in attesa fino a) ricevere un messaggio da un altro processo.

Per comunicare due processi devono:

- Stabilire un **link di comunicazione** tra di loro.
- Scambiarsi messaggi usando `send` e `receive`.

Pipe

Canali di comunicazione tra i processi (una forma di message passing).
Varianti:

- Unidirezionale o bidirezionale.
- (se bidirezionale) Half-duplex o full-duplex.
- Relazione tra i processi comunicanti (sono padre-figlio o no).
- Usabili o meno in rete.

Pipe convenzionali:

- Unidirezionali.
- Non accessibili al di fuori del processo creatore...
- ...quindi di solito condivise con un processo figlio attraverso una `fork()`.
- In Windows sono chiamate "*pipe anonime*".

Pipe convenzionali:

- Bidirezionali
- Esistono anche dopo la terminazione del processo che le ha create.
- Non richiedono una relazione padre-figlio tra i processi che le usano.

In Unix:

- Half-duplex
- Solo sulla stessa macchina
- Solo dati byte-oriented

In Windows:

- Full-duplex
- Anche tra macchine diverse
- Anche dati message-oriented

9.4.3 Notifiche con callback

In alcuni sistemi operativi (es. API POSIX e Win32) un processo può notificare un altro processo in maniera da causare l'esecuzione di un blocco di codice («callback»), similmente ad un interrupt.
Nei sistemi Unix-like (POSIX, Linux) tale notifiche vengono dette segnali, ed interrompono in maniera asincrona la computazione del processo corrente causando un salto brusco alla callback di gestione, al termine della quale la computazione ritorna al punto di interruzione.
Nelle API Win32 esiste un meccanismo simile, detto Asynchronous Procedure Call (APC), che però richiede che il ricevente si metta esplicitamente in uno stato di attesa, e che esponga un servizio che il mittente possa invocare.

9.5 Le API POSIX per la comunicazione interprocesso

9.5.1 Memoria condivisa in POSIX

- Un processo crea un segmento di memoria condivisa con la funzione `shm_open`:

```
int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Anche usato per aprire un segmento precedentemente creato
- Quindi imposta la dimensione del segmento con la funzione ftruncate:

```
ftruncate(shm_fd, 4096);
```

- Infine la funzione mmap mappa la memoria condivisa nello spazio di memoria del processo:

```
void *shm_ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

- Da questo momento si può usare il puntatore `shm_ptr` ritornato da `mmap` per leggere/scrivere la memoria condivisa

9.5.2 Pipe anonime in POSIX

- Vengono create con la funzione pipe, che ritorna due descrittori, uno per il punto di lettura e uno per il punto di scrittura:

```
int p_fd[2];
int res = pipe(p_fd);
```

- Le funzioni read e write permettono di leggere e scrivere:

```
ssize_t n_wr = write(p_fd[1], "Hello, World!", 14);
char buffer[256];
ssize_t n_rd = read(p_fd[0], buffer, sizeof(buffer) - 1);
```

  P.S.: di solito si usa la pipe quindi la `fork()` per comunicare tra processi padre e figlio.

- È possibile utilizzare la funzione `fdopen` per fare il wrapping di un punto della pipe in un file, ed utilizzare le funzioni C stdio con esso

9.5.3 Named pipes in POSIX

- Le named pipes vengono anche chiamate FIFO nei sistemi POSIX
- Per creare una FIFO si utilizza l'API `mkfifo`:

```
int res = mkfifo("/home/pietro/myfifo", 0640);
```

- La FIFO si utilizza come un normale file:

```
int fd = open("/home/pietro/myfifo", O_RDONLY);
char buffer[256];
ssize_t n_rd = read(fd, buffer, sizeof(buffer) - 1);
```

- Al termine dell'utilizzo, ricordarsi di chiudere:

```
close(fd);
```

- Per eliminare la FIFO, usare l'API unlink:

```
unlink("/home/pietro/myfifo");
```

9.5.4 Segnali POSIX

- Per inviare un segnale ad un processo utilizzare l'API kill:

```
int ok = kill(1000, SIGTERM); /* terminazione al processo 1000 */
```

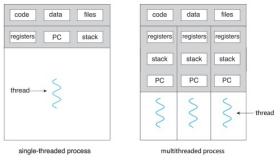
- Per registrare una callback per un determinato segnale utilizzare l'API sigaction:

```
struct sigaction act;
sigemptyset(&act.sa_mask); /* non bloccare altri segnali */
act.sa_flags = SA_SIGINFO; /* callback in act.sa_sigaction */
act.sa_sigaction = sigterm_handler; /* la callback */
int ok = sigaction(SIGTERM, &act, NULL);
```

9.6 Multithreading

Fino ad ora abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale (ossia, un singolo processore virtuale).
Se supponiamo che un processo possa avere molti processori virtuali, più istruzioni possono eseguire concorrentemente, e quindi il processo può avere più percorsi (thread) di esecuzione concorrenti.

9.6.1 Processi single e multithreaded



I thread di uno stesso processo condividono la memoria globale (data), la memoria contenente il codice (code) e le risorse ottenute dal sistema operativo (ad esempio i file aperti).
Ogni thread di uno stesso processo però deve avere un proprio stack, altrimenti le chiamate a subroutine di un thread interferirebbero con quelle di un altro thread concorrente.
Qua non è messo ma anche lo heap è condiviso.
Tipica domanda dell'esame: come funziona.

Librerie di thread

I thread si sfruttano tramite delle *librerie di thread*.
Sono le API fornite al programmatore per creare e gestire thread. Librerie più in uso:

- POSIX pthreads
- Windows threads

9.7 Le API POSIX per il multithreading

9.7.1 POSIX pthreads

Non sono un'implementazione, ma una specifica (POSIX standard IEEE 1003.1c).
Comune nei sistemi Unix e Unix-like (BSD, Linux, macOS).

9.7.2 Creare un nuovo thread

- All'inizio un processo viene creato con un singolo thread
- Per creare un nuovo thread si utilizza l'API `pthread_create`, per attendere la fine dell'esecuzione di un thread si utilizza l'API `pthread_join`:

```
void *thread_code(void *name) { ... }
...
pthread_id tid1, tid2;
int ok1 = pthread_create(&tid1, NULL, thread_code, "thread 1");
int ok2 = pthread_create(&tid2, NULL, thread_code, "thread 2");
...
void *ret1, *ret2;
ok1 = pthread_join(tid1, &ret1);
ok2 = pthread_join(tid2, &ret2);
...
```

9.7.3 Aspetti particolari nelle API per il multithreading

- Chiamate di sistema fork() ed exec()
- Gestione dei segnali

- Cancellazione dei thread
- Dati locali dei thread

9.7.4 Chiamate di sistema `fork()` ed `exec()`

Una `fork()` dovrebbe duplicare solo il thread chiamante o tutti i thread? Alcuni sistemi operativi Unix-like hanno due diverse `fork()`.
`exec()` invocata da un thread che effetto ha sugli altri thread? Di solito termina tutti i thread del processo precedentemente in esecuzione.

9.7.5 Gestione dei segnali

- Quando un processo è single-threaded, un segnale interrompe l'unico thread del processo
- Quando vi sono più thread, quale thread riceve il segnale?
- Possibili soluzioni:
  - Il thread a cui si applica il segnale (ad es. il segnale SIGSEGV viene inviato al thread che ha generato il segmentation fault)
  - Ogni thread del processo
  - Alcuni thread del processo
  - Un thread speciale del processo deputato esclusivamente alla ricezione dei segnali

9.7.6 Cancellazione dei thread

- L'operazione di cancellazione di un thread determina la terminazione prematura del thread
- Può essere invocata da un altro thread
- Due approcci (**tipicamente chiesta all'esame**):

  Cancellazione asincrona : il thread che riceve la cancellazione viene terminato immediatamente

  Cancellazione differita : un thread che supporta la cancellazione differita deve controllare periodicamente se esiste una richiesta di cancellazione pendente, e in tal caso terminare la propria esecuzione

- Vantaggi:

  Cancellazione differita : dal momento che un thread controlla il momento della propria cancellazione, può effettuare una terminazione ordinata

  Cancellazione asincrona : nessuna necessità di controllare periodicamente se ci sono richieste di cancellazione pendenti

9.7.7 Cancellazione nei POSIX pthreads

- Si può attivare/disattivare la cancellazione, ed avere sia cancellazione differita (default) che asincrona
- Se la cancellazione è inattiva, le richieste di cancellazione rimangono in attesa fino a quando (se) è attivata
- In caso di cancellazione differita, questa avviene solo quando l'esecuzione del thread raggiunge un punto di cancellazione (di solito una chiamata di sistema bloccante)
- Il thread può aggiungere un punto di cancellazione controllando l'esistenza di richieste di cancellazione con la funzione `pthread_testcancel()`

Esempio

```
void *thread_code(void *name) {
    int oldtype, oldstate;
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
    while (true) {
        pthread_testcancel();
        ... /* fa qualcosa di non interrompibile ma di durata finita */
    }
}
...
pthread_id tid;
int ok = pthread_create(&tid, NULL, thread_code, "thread 1");
... /* dopo un po' */
ok = pthread_cancel(tid);
void *ret;
ok = pthread_join(tid, &ret);
```

9.7.8 Dati locali dei thread

- In alcuni casi è utile assegnare ad un thread dei dati locali (thread local storage, TLS) non condivisi con gli altri thread dello stesso processo
- La TLS è diversa dalle variabili locali (ad es. è visibile a tutte le funzioni)
- Simile ai dati `static` del linguaggio C, ma unica per ciascun thread
- Utile quando il programma non ha un controllo diretto sul momento di creazione dei thread (es. quando si usano i thread pools)

Nei POSIX pthreads

- In POSIX i dati locali dei thread si possono creare utilizzando le funzioni `pthread_key_create()`, `pthread_getspecific()` e `pthread_setspecific()`
- `pthread_key_create()` crea un oggetto opaco di tipo `pthread_key_t`, che può essere usato da tutti i thread per identificare un dato locale

- `pthread_setspecific()` permette di associare ad una `pthread_key_t` un valore di tipo `void*`, `pthread_getspecific()` di richiamarlo data la chiave
- Ogni thread può associare ad una stessa chiave il proprio distinto valore locale (e successivamente richiamarlo)
  Di solito non chiesti all'esame.

Capitolo 10

Processi e thread: la struttura

Argomenti

- Multiprogrammazione e multitasking
- Implementazione dei processi
- Implementazione del multithreading
- Criteri di valutazione algoritmi di scheduling
- Principali algoritmi di scheduling
- Code multilivello con retroazione

10.1 Multiprogrammazione e multitasking

10.1.1 Multiprogrammazione e multitasking

Due obiettivi dei sistemi operativi:

Efficienza : mantenere impegnata la (o le) CPU il maggior tempo possibile nell'esecuzione dei programmi (se ci sono programmi da eseguire)

Reattività : dare l'illusione che ogni processo progredisca continuamente nella propria esecuzione, come se avesse una CPU dedicata; questo è particolarmente importante per i programmi interattivi, che devono reagire in tempi accettabili quando ricevono un input utente

Le due tecniche adottate nei sistemi operativi per ottenere questi due obiettivi sono la **multiprogrammazione** e il **multitasking** (o **time-sharing**)

- **Obiettivo della multiprogrammazione**: impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU
- **Obiettivo del multitasking**: far sì che un programma interattivo possa reagire agli input utente in un tempo accettabile

- Notare che la multiprogrammazione non è una tecnica rilevante per i sistemi puramente batch

10.1.2 Burst CPU e burst I/O

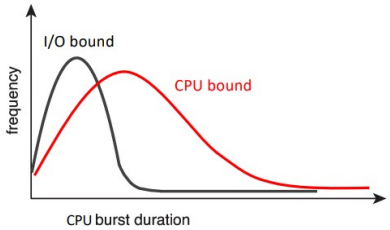
L'obiettivo della multiprogrammazione è massimizzare l'utilizzo della CPU.
Gli algoritmi di scheduling sfruttano il fatto che di norma l'esecuzione di un processo è una sequenza di:

- **Burst della CPU:** sequenza di operazioni di CPU
- **Burst dell'I/O:** attesa completamento operazione di I/O

10.1.3 Distribuzione delle durate dei burst della CPU

Programma con prevalenza di I/O (**I/O bound**):

- Elevato numero di burst CPU brevi
- Ridotto numero di burst CPU lunghi
- Tipico dei programmi interattivi



Programma con prevalenza di CPU (**CPU bound**):

- Elevato numero di burst CPU lunghi
- Ridotto numero di burst CPU brevi
- Tipico dei programmi batch

In entrambi i casi la curva della distribuzione ha la forma riportata accanto, ma:

- I/O bound: il massimo sta più a sinistra
- CPU bound: il massimo sta più a destra

10.1.4 Multiprogrammazione: l'implementazione

Il sistema operativo mantiene in memoria i processi da eseguire.
Quando una CPU non è impegnata ad eseguire un processo, il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU.
Quando un processo non può proseguire l'esecuzione (ad es. perché deve attendere il termine dell'input di dati che gli servono per proseguire), la sua CPU viene assegnata ad un altro processo non in esecuzione.
Come risultato, se i programmi da eseguire sono più delle CPU, queste saranno impegnate nell'esecuzione di qualche processo per la maggior parte del tempo.
Questo risolve il problema dell'efficienza: *non tenere le CPU ferme.*

10.1.5 Multitasking: l'implementazione

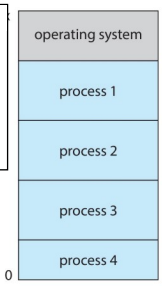
È un'*estensione della multiprogrammazione* per i sistemi interattivi.
La CPU viene "sottratta" periodicamente al programma in esecuzione ed assegnata ad un altro programma. Quindi non più solo quando il programma non è più in grado di proseguire ma anche ogni tot, così tutti i programmi hanno la possibilità di progredire. In questo modo tutti i programmi progrediscono in maniera continuativa nella propria esecuzione, anziché solo nei momenti in cui il programma che detiene la CPU si mette in attesa.
Questo fa sì che i programmi batch, che hanno burst CPU lunghi e pochi burst I/O (effettuano poco I/O (e quindi vanno poco in attesa)), non monopolizzino la CPU a discapito dei programmi interattivi.

10.1.6 Multiprogrammazione: la memoria

La multiprogrammazione richiede che tutte le immagini di tutti i processi siano in memoria perché questi possano essere eseguibili.
Se i processi sono troppi non possono essere contenuti tutti in memoria: la tecnica dello **swapping** può essere usata per spostare le immagini dentro/fuori dalla memoria.

DEFINIZIONE

Swapping significa che il s.o. può togliere la memoria (anche detta *immagine*) di un processo dalla memoria centrale e caricare l'immagine di un altro processo.



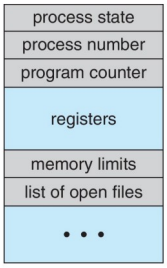
La **memoria virtuale** è un'ulteriore tecnica che permette, se l'immagine di un processo è troppo grande, di eseguire un processo la cui immagine non è completamente in memoria.
Queste tecniche aumentano il numero di processi che possono essere eseguiti in multiprogrammazione, ossia il grado di multiprogrammazione.
Es.: se in un istante sto eseguendo 100 processi, il grado di multiprogrammazione è 100. Se in un altro istante sto eseguendo 50 processi, in quell'istante il grado di multiprogrammazione sarà 50.

10.2 Implementazione dei processi

10.2.1 Process Control Block (PCB)

Detto anche task control block.
È la struttura dati del kernel che contiene tutte le informazioni relative ad un processo:

- Process state: ready, running...
- Process number (o PID): identifica il processo



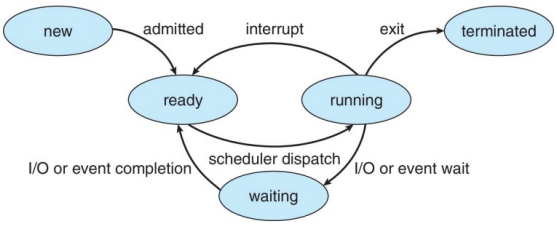
- Program counter: contenuto del registro "istruzione successiva"
- Registers: contenuto dei registri del processore
- Informazioni relative alla gestione della memoria: memoria allocata al processo
- Informazioni sull'I/O: dispositivi assegnati al processo, elenco file aperti...
- Informazioni di scheduling: priorità, puntatori a code di scheduling...
- Informazioni di accounting: CPU utilizzata, tempo trascorso...

10.2.2 Stati di un processo

Durante l'esecuzione, un processo cambia più volte stato.
Gli stati possibili di un processo sono:

- Nuovo (**new**): il processo è creato, ma non ancora ammesso all'esecuzione
- Pronto (**ready**): il processo può essere eseguito (è in attesa che gli sia assegnata una CPU)
- In esecuzione (**running**): le sue istruzioni vengono eseguite da qualche CPU
- In attesa (**waiting**): il processo non può essere eseguito perché è in attesa che si verifichi qualche evento (ad es. il completamento di un'operazione di I/O)
- Terminato (**terminated**): il processo ha terminato l'esecuzione

10.2.3 Diagramma di transizione di stato dei processi



N.B.: quel "`interrupt`" non è da confondersi con l'`interrupt` dei .

Il processo nello stato `new` è stato creato ma non ancora ammesso. QUando il numero tot di processi che il s.o. vede alternarsi negli stati `new`, `running` (in esecuzione, ha una CPU e il codice del processo viene eseguito) e `ready` (pronto per essere eseguito ma non ha ancora CPU) è sufficientemente basso, il processo viene ammesso.

Lo stato di `waiting`: va in attesa di uno specifico evento (ad esempio il completamento di un'operazione di I/O) e quando questo evento si verifica il processo torna nello stato `ready`, ma non gli viene immediatamente assegnata la CPU che potrebbe essere occupata da un altro processo.
Stato `exit`: viene invocata l'API `exit()` e il processo termina e il s.o. ne recupererà tutte le risorse.

10.2.4 Lo scheduler della CPU

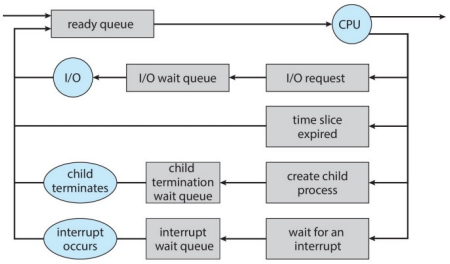
Lo **scheduler della CPU**, o **scheduler a breve termine** sceglie il prossimo processo da eseguire tra quelli in stato `ready` ed alloca un core libero ad esso.
Mantiene diverse code di processi:

Ready queue : processi residenti in memoria e in stato `ready`.

Wait queues : code (1+) per i processi che sono residenti in memoria e in stato `wait`; una coda diversa per ciascun diverso tipo di evento di attesa che mi fa entrare un processo in stato di `waiting`.

Durante la loro vita, i processi (i loro PCB) migrano da una coda all'altra a seconda dello stato del processo stesso.

10.2.5 Rappresentazione delle code di scheduling



- Ready + CPU = running.
- Quando il time slice di un processo termina, rimesso nella ready queue.
- Se un processo padre si mette in attesa di un processo figlio, il padre va in wait queue (visto che non gli serve CPU). Quando il processo figlio termina, quel processo padre viene ripescato da quella queue e rimesso nella ready queue.

10.2.6 Schemi di scheduling

Lo scheduler della CPU ha il compito di decidere a quale processo tra tutti quelli nella `ready queue` assegnare un core libero.
Tali **decisioni di scheduling** possono essere effettuati in diversi momenti, corrispondenti a cambi di stato dei processi:

1. Quando un processo passa da stato running a stato waiting
2. Quando un processo passa da stato running a stato ready

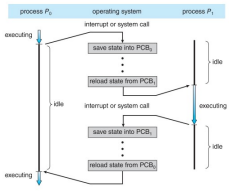
3. Quando un processo passa da stato waiting a stato ready
4. Quando un processo termina

Se il riassegnamento viene fatto solo nelle situazioni 1 e 4 lo schema di scheduling è detto senza prelazione (**nonpreemptive**) o cooperativo, dal momento che un core è sempre liberato da un processo che volontariamente rinuncia ad esso. Contrariamente a quello con prelazione, un processo può essere interrotto in qualsiasi momento. Viene "congelato" e bloccato in memoria fino a quando verrà ripreso e riprenderà a funzionare come se non fosse mai tato interrotto. Questa cosa è permessa dal **dispatcher**.
Altrimenti è detto con prelazione (**preemptive**), dal momento che un core può essere anche liberato perché un core viene forzatamente (quindi in modo asincrono) sottratto dal kernel ad un processo che lo sta usando. È tipo quello che succede quando introduciamo i thread asincroni (se sta svolgendo un'operazione es. di salvataggio dei dati, ma viene brutalmente interrotto, l'operazione ovviamente non va a buon fine).
Lo schema di scheduling preemptive è più complicato da implementare:

- Che succede se due processi condividono dati?
- Che succede se un processo sta eseguendo in modalità kernel (system call o IRQ)?

10.2.7 Commutazione di contesto e dispatcher

Quando il controllo della CPU deve passare da un processo ad un altro processo scelto dallo scheduler a breve termine avviene una **commutazione di contesto** (**context switch**).
La **commutazione di contesto** viene effettuata dal ***dispatcher***, che:



- Salva il contesto (stato, registri...) del processo da interrompere nel suo PCB, *process control block*.
- Carica il contesto del processo da eseguire dal suo PCB.
- Passa in modalità utente.
- Salta nel punto corretto del programma del processo selezionato (ossia, dove era stato precedentemente interrotto).

Notare che il dispatcher implementa un tipico *meccanismo*, mentre lo scheduler implementa una tipica *politica*.
Es.: program counter, ad ogni registro salvato viene incrementato, è da salvare alla fine quando ha finito di incrementare.
Q: se anche P1 avesse un'interrupt, cosa succede? Si torna a P0 o viene introdotto un altro processo?
A: dipende dalla politica di scheduling, ma in questo caso si vede chiaramente che abbiamo solo P0 e P1, quindi si torna a P0. Questa si chiama *round robin*

ovvero scheduling circolare, si passa dal primo al secondo al primo di nuovo al secondo ....
Garanzia: non ci sono processi a cui non do tempo e a tutti i processi do lo stesso tempo.

10.2.8 Latenza di dispatch

Essendo un meccanismo, si vuole che impieghi il meno tempo possibile.
La **latenza di dispatch** è il tempo impiegato dal dispatcher per fermare un processo ed avviarne un altro.
La latenza di dispatch è puro overhead: non viene eseguito alcun lavoro utile (il lavoro utile è l'esecuzione di un programma utente).
Più è complesso il sistema operativo, più è complesso il PCB (il contesto), maggiore è la latenza di dispatch.
Alcuni processori offrono supporto speciale per minimizzare la latenza di dispatch (es. banchi di registri multipli).
Qua finisce la parte su multiprogrammazione e multitasking.

10.3 Implementazione del multithreading

Molto più complesso, vedremo molto superficialmente.

10.3.1 Thread a livello utente e kernel

Kernel così complesso che anche a livello di kernel si opera in multithreading (giusto?).

Thread a livello utente: i thread disponibili nello spazio utente dei processi; sono quelli offerti dalle librerie di thread ai processi.

Thread a livello del kernel: i thread implementati nativamente dal kernel; sono utilizzati per strutturare il kernel (se stesso) in maniera concorrente. Utile per strutturare il kernel, aprendo ad una miriade di possibilità (e di errori), potendo sfruttare la presenza di più processori (e core) nel mio sistema. C'è la tentazione di offrirli anche a livello utente, ma non è una buona idea. Ci sono modelli di supporto per ovviare a questa cosa.

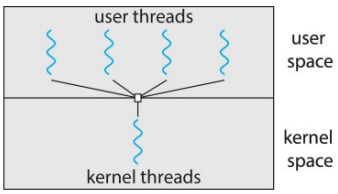
Modelli di supporto al multithreading

I thread a livello del kernel vengono utilizzati dalle librerie di thread per implementare i thread a livello utente di un certo processo.
A tale scopo possono essere adottate diverse strategie (modelli di multithreading):

Molti-a-uno: i thread a livello utente di un certo processo sono implementati su un solo thread a livello del kernel.

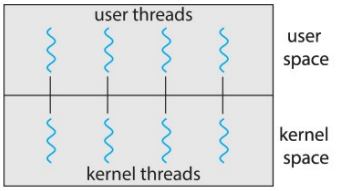
- Vantaggio: usabile su ogni sistema operativo (unica soluzione possibile se il sistema operativo non è multithreaded).
- Svantaggio: se un thread utente fa una chiamata di sistema bloccante blocca tutti i thread utente dello stesso processo.

- Altro svantaggio: non sfrutta la presenza di più core.
- Poco usato in pratica.
- Esempi:
  - Solaris Green Threads
  - GNU Portable Threads



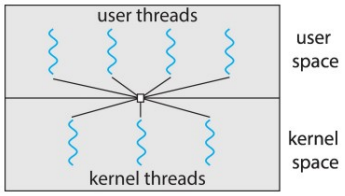
Uno-a-uno: ogni thread a livello utente è implementato su un singolo, distinto thread a livello del kernel.

- Vantaggio: permette un maggior grado di concorrenza.
- Vantaggio: permette di sfruttare il parallelismo nei sistemi multicore.
- Svantaggio: minore performance rispetto al modello molti-a-uno.
- Svantaggio: stress del kernel.
- Esempi:
  - Linux
  - Windows



Molti-a-molti: i thread a livello utente di un certo processo sono implementati su un insieme di thread a livello del kernel possibilmente inferiore di numero, e l'associazione thread utente / thread kernel è dinamica, stabilita da uno scheduler interno alla libreria di thread.

- Cerca di combinare i vantaggi dei modelli molti-a-uno e uno-a-uno.
- Svantaggio: complesso da implementare (la libreria di thread deve dinamicamente alternare l'esecuzione dei thread a livello utente sui thread a livello del kernel disponibili).
- **Modello a due livelli:** permette agli utenti di creare dei thread che hanno un mapping uno-a-uno con un thread a livello del kernel.
- Esempi:
  - Solaris
  - AIX



10.3.2 Thread control blocks

Visto che ormai ogni thread viene schedulato (lo scheduler non ha più solamente a che fare con i processi, deve avere l'idea che i thread sono composti da processi), serve un thread control block (TCB) legati al process control block (PCB) che già abbiamo visto.
Nei kernel multithreaded il kernel mantiene delle strutture dati analoghe ai PCB, i thread control blocks (TCB); un TCB memorizza il contesto di un thread e le sue informazioni di contabilizzazione.
Il PCB contiene solo le informazioni di contesto e contabilizzazione comuni (ad esempio lo spazio di memoria).
Il PCB è di norma collegato ai TCB dei thread kernel utilizzati dal processo, e viceversa i TCB dei thread kernel utilizzati da un processo sono collegati al PCB del processo.

10.4 Criteri di valutazione algoritmi di scheduling

10.4.1 Confrontare algoritmi di scheduling

Praticamente ogni sistema operativo ha i propri algoritmi di scheduling (quindi sono tantissimi).
Questo è indizio del fatto che non esiste una politica di scheduling "migliore" di tutte le altre.
Sì, ma... in quale senso possiamo dire che una politica di scheduling è "migliore" di un'altra?

10.4.2 Criteri di valutazione algoritmi di scheduling

- Misure che servono per confrontare le caratteristiche dei diversi algoritmi.
- (purtroppo non dipendono solo dall'algoritmo, ma anche dal carico)
- **Principali criteri**:

  Utilizzo della CPU: % di tempo in cui la CPU è attiva nell'esecuzione dei processi utente (dovrebbe essere tra il 40% e il 90%, in funzione del carico)

  Throughput: # di processi che completano l'esecuzione nell'unità di tempo (dipende dalla durata dei processi)

  Tempo di completamento: tempo necessario per completare l'esecuzione di un certo processo (dipende da molti fattori: durata del processo, carico totale, durata dell'I/O...)

Tempo di attesa: tempo trascorso dal processo nella ready queue (meglio del tempo di completamento, meno dipendente da durata del processo e dell'I/O)

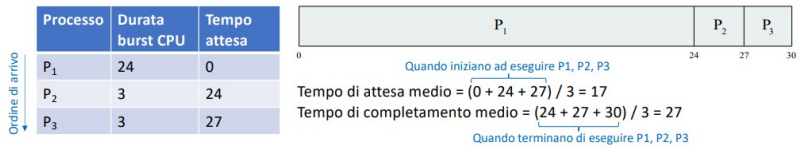
Tempo di risposta: negli ambienti interattivi, tempo trascorso tra l'arrivo di una richiesta al processo e la produzione della prima risposta, senza l'emissione di questa nell'output

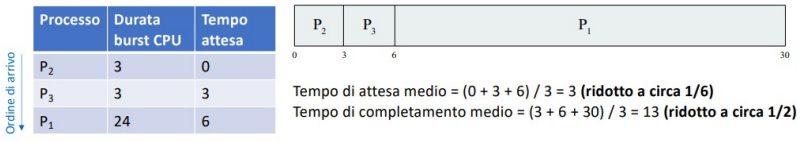
- A noi interessano essenzialmente tempo di completamento e di attesa, e su quelli svolgeremo i nostri esercizi.
- Cosa guardiamo:
  - Massimo utilizzo della CPU
  - Massimo throughput
  - Minimo tempo di completamento (medio)
  - Minimo tempo di attesa (medio)
  - Minimo tempo di risposta (medio)
- Naturalmente nessun algoritmo di scheduling può ottimizzare tutti i criteri contemporaneamente...

10.5 Principali algoritmi di scheduling

10.5.1 Scheduling in ordine di arrivo

Scheduling in ordine di arrivo, o **first-come-first-served** (FCFS): la CPU viene assegnata al primo processo che la richiede.
Vantaggio: implementazione molto semplice (coda FIFO, nessuna prelazione).
Svantaggio: tempo di attesa medio può essere lungo ("effetto convoglio").





Basta cambiare l'ordine di arrivo dei processi per cambiare completamente tutti i tempi.

10.5.2 Scheduling per brevità

Scheduling per brevità, o **shortest-job-first** (SJF): la CPU viene assegnata al processo che ha il successivo CPU burst più breve.
Vantaggi: implementazione quasi identica a FCFS, ma minimizza il tempo di attesa medio (è ottimale).
Svantaggio: di solito non si sa in anticipo qual è il processo che avrà il CPU burst più breve (quanto durerà il prossimo CPU burst?).
L'algoritmo **shortest-remaining-time-first** (SRTF) utilizza la prelazione per gestire il caso in cui i processi non arrivino tutti nello stesso istante: se nella `ready queue` arriva un processo con un burst più corto di quello running, quest'ultimo viene prelazionato dal nuovo processo.

10.5.3 Esempi

Scheduling per brevità

Saranno così gli esercizi d'esame. Dovremo saper calcolare i tempi di attesa medio e di completamento medio per gli algoritmi presentati.

Ordine di arrivo ↓

Processo	Durata burst CPU	Tempo attesa
P_1	6	0
P_2	8	6
P_3	7	14
P_4	3	21

Tempo di attesa medio $FCFS = \frac{0+6+14+21}{4} = 10.25$
Tempo di completamento medio $FCFS = \frac{6+14+21+24}{4} = 16.25$

Ordine di arrivo ↓

Processo	Durata burst CPU	Tempo attesa
P_1	6	0
P_2	8	6
P_3	7	14
P_4	3	21

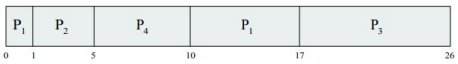
Tempo di attesa medio $SJF = \frac{0+3+9+16}{4} = 7$
Tempo di completamento medio $SJF = \frac{3+9+16+24}{4} = 13$

Shortest-remaining-time-first

- Con preemption e tempo di arrivo
- Il tempo di attesa di un processo è:
  `istante terminazione processo - (tempo di arrivo + durata burst)`
- Il tempo di completamento di un processo invece è:
  `istante terminazione processo - tempo di arrivo`

Processo	Tempo di arrivo	Durata burst CPU
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Determina l'ordine di arrivo



Tempo di attesa medio $SJF = \frac{(17-8)+(5-5)+(26-11)+(10-8)}{4} = 6.5$
Tempo di completamento medio $SJF = \frac{(17-0)+(5-1)+(26-2)+(10-3)}{4} = 13$

10.5.4 Scheduling circolare

Nello **scheduling circolare**, o **round-robin** (RR) ogni processo ottiene una piccola quantità fissata di tempo di CPU (**quanto di tempo**), di solito 10-100 millisecondi, per il quale può essere in esecuzione.
Trascorso tale tempo il processo in esecuzione viene interrotto e messo in fondo alla ready queue, che è gestita in maniera FIFO.
In tal modo la ready queue funziona essenzialmente come un buffer circolare, e i processi vengono scanditi dal primo all'ultimo, per poi ripartire dal primo nello stesso ordine.
Timer che genera un interrupt periodico con periodo q per effettuare la prelazione del processo corrente (passaggio del processo da stato running a ready).

Esempio

- Ricordiamo che il tempo di attesa di un processo è:
  `istante terminazione processo - (tempo di arrivo + durata burst)`
- Il tempo di completamento di un processo invece è:
  `istante terminazione processo - tempo di arrivo`
- In questo esempio il tempo di arrivo è 0 per tutti i processi

Processo	Durata burst CPU
P_1	24
P_2	3
P_3	3

Ordine di arrivo

$q = 4$
Tempo di attesa medio $= \frac{(30-24)+(7-3)+(10-3)}{3} = 5.67$
Tempo di completamento medio $= \frac{30+7+10}{3} = 15.67$

Scheduling circolare: caratteristiche

Se ci sono n processi nella `ready queue` e il quanto temporale è q:

- Nessun processo attende più di q (n - 1) unità di tempo nella ready queue prima di ridiventare running per un altro quanto di tempo (rispetto a SJF (shortest job first) e SRTF non c'è bisogno di sapere la durata del burst per avere un'idea di quale sia la durata di attesa nella ready queue)

- Ogni processo ottiene $1/n$ del tempo totale di CPU, in maniera perfettamente equa (rispetto a SJF (shortest job first) e SRTF vengono ottenute solo q unità di tempo per volta) del tempo della CPU.
  Va molto bene per i processi interattivi, che hanno bisogno di un tempo di risposta il più rapido possibile.

Comportamento in funzione del quanto di tempo, q, la parte programmabile di tutto lo scheduling:

- q troppo elevato: se q è di regola maggiore della durata di tutti i burst, RR degenera nel FCFS (diventa una coda dove il primo processo schedulato è il primo in ordine di arrivo). Non una buona idea, fa perdere lo scopo di avere un RR.

- q troppo basso: deve comunque essere molto più lungo della **latenza di dispatch**, tempo "mangiato" dal cambio di contesto che finora avevamo considerato come pari a zero e rappresentato da una lineetta, altrimenti questa si "mangia" un tempo comparabile al tempo di esecuzione dei processi utente e l'utilizzo della CPU diventa inaccettabilmente basso.
  L'utilizzo della CPU quindi non sarà più al 100% ma anzi se il quanto di tempo inizia a tendere alla latenza di dispatch, l'utilizzo della CPU tende al 50% tipo che non va assolutamente bene.

Performance:

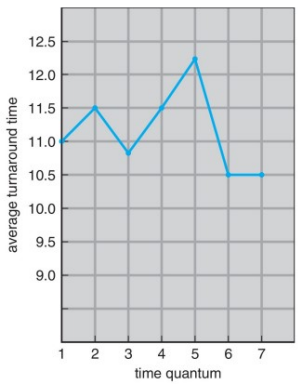
- Rispetto a SJF (shortest job first) tipicamente RR ha un tempo di completamento medio più alto (non una buona idea per i processi batch).

- Ma un tempo di risposta medio più basso (va bene per i processi interattivi).

- Il tempo di completamento medio non necessariamente migliora con l'aumento di q.

Scheduling circolare: tempo di completamento

Il tempo di completamento medio migliora se la maggioranza ($\tilde{8}0\%$) dei CPU burst è più breve di q.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Lì inizia ad essere un po' più simile a FCFS.

10.5.5 Scheduling con priorità

Nello scheduling con priorità ad ogni processo è associato un numero intero che indica la sua priorità.
Viene eseguito il processo con priorità più alta, gli altri aspettano (in Unix-like numero più basso = priorità più alta, in Windows il contrario).
Può essere preemptive o no.
Possono essere permessi più processi con *pari priorità* o no; in caso positivo occorre stabilire un secondo algoritmo di scheduling per arbitrare tra i processi a pari priorità (di solito si utilizza RR).
SJF è scheduling con priorità, dove la priorità è l'inverso della durata del CPU burst (burst breve priorità bassa, burst lungo priorità alta? ho capito giusto?).
Problema della **attesa indefinita** (**starvation**): un processo a priorità troppo bassa potrebbe non venir mai schedulato, sempre superato da altri processi a priorità maggiore. È un problema degli algoritmi di scheduling, l'assenza di garanzia che un processo prima o poi venga schedulato.
Soluzione: **invecchiamento** (**aging**), ossia aumento automatico di priorità di un processo al crescere del tempo di permanenza nella ready queue, in modo che anche i job a priorità più bassa prima o poi vengano schedulati.

Scheduling con priorità: esempio

Processo	Durata burst CPU	Priorità (UNIX)
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Tempo di attesa medio = $\frac{0+1+6+16+18}{5} = 8.2$
Tempo di completamento medio = $\frac{1+6+16+18+19}{5} = 12$

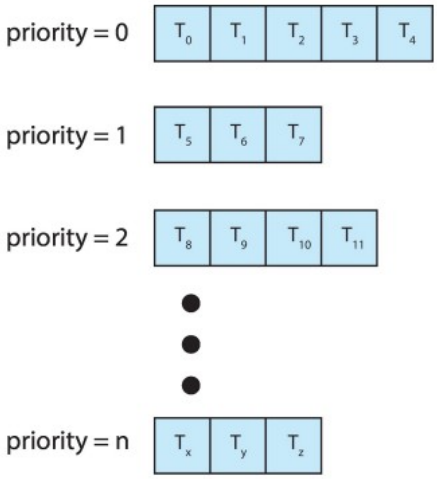
Scheduling con priorità + RR: esempio

$q = 2$ per processi con stessa priorità (l'esercizio **deve** specificare il quanto di tempo)
Calcolo sempre con le formuline viste nel SRTF:
Tempo di attesa medio = $\frac{(26-4)+(16-5)+(20-8)+(7-7)+(27-3)}{5} = 13.8$
Tempo di completamento medio = $\frac{26+16+20+7+27}{5} = 19.2$
Anche con il RR, finché c'è un processo da schedulare, occupa CPU e ne usa il 100%.
Tipicamente i processi con priorità pari a 1 hanno tempo di attesa pari a 0.

10.6 Code multilivello con retroazione

10.6.1 Code multilivello

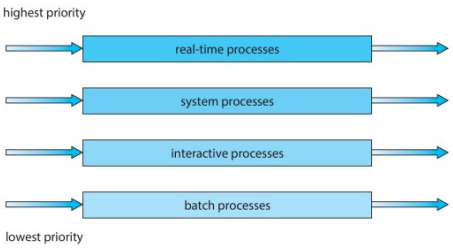
A volte un algoritmo solo non basta.
Lo scheduling con priorità usa (tante) ready queue code separate, una per ogni priorità in modo da pescare un processo ready dalla coda giusta.
Viene schedulato il processo nella coda non vuota con priorità maggiore. Se c'è qualcosa, RR. Se è vuota, vado in quella sotto. Se è vuota, vado in quella sotto... Finché non ne trovo una con un processo che viene quindi schedulato.



Priotità basata sul tipo di processo.

A volte sono s.o. embedded (ovvero interamente dedicato ad un solo computer con uno scopo, es. una macchina, un aereo, una lavatrice...). I processi sono fissi e non cambiano e non generano altri processi.
Deve essere lo scheduler a decidere quale processo eseguire. Nello scheduling con priorità in base a quale criterio possiamo assegnare le priorità ai processi?
Un criterio comunemente usato è quello di basare la priorità di un processo sul suo tipo:

- Priorità più alta ai processi interattivi o cyber-fisici che devono reagire rapidamente all'I/O (tipicamente I/O bound)
- Priorità più bassa ai processi che effettuano lunghe computazioni, o processi batch (tipicamente CPU bound)



Il problema è: come faccio a individuarli? Come faccio a capire quale processo è I/O bound (es. office per Windows) e quale CPU bound (es. grap per Unix-like)?
Ma non possiamo costringere chi scrive a catalogarli a mano oppure in momenti diversi alcuni processi possono comportarsi in modi diversi. Come faccio a metterlo nella coda giusta?
Soluzione: **code multilivello con retroazione**. In modo da avere processi più dinamici.

Code multilivello con retroazione

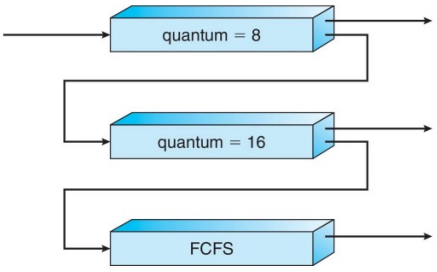
La priorità di un processo può variare dinamicamente: è sufficiente spostarlo da una ready queue ad una certa priorità ad un'altra.
L'invecchiamento è di solito implementato in questo modo (spostamento di un processo verso una coda a priorità più alta per evitare la starvation).
Ma anche la identificazione di un processo come I/O bound o CPU bound può essere effettuata dinamicamente, e determinare un cambio di priorità, stavolta con uno spostamento verso il basso.
Uno scheduler di questo tipo è detto con code multilivello con retroazione.
Introdotto nel sistema operativo CTSS (Corbatò et al., 1962).
Questo tipo di scheduler è adottato da moltissimi sistemi operativi moderni (Windows, macOS, FreeBSD, Solaris).

Esempio

Code:

- Q_0: RR con $q_0 = 8$ msec
- Q_1: RR con $q_1 = 16$ msec
- Q_2: RR con $q_2 = \infty$ (ossia FCFS)

Q_0 ha priorità alta, Q_1 intermedia, Q_2 bassa.
Alla creazione un processo entra in Q_0.
Se quando un processo diventa running non termina il suo burst entro il suo quanto di tempo, avviene preemption e viene messo all'inizio della coda immediatamente più bassa, altrimenti rimane nella sua coda.
Per evitare starvation l'invecchiamento sposta i processi in direzione opposta, verso le code più alte, se passano troppo tempo in una coda senza essere eseguiti.
Effetto ricercato: mantenere i processi I/O bound (con burst della CPU corti) nelle code ad alta priorità e quelli CPU bound (con burst della CPU lunghi) nelle code a bassa priorità.
NB: dall'alto verso il basso li faccio passare con l'invecchiamento.



Capitolo 11

Gestione della memoria: la struttura

Argomenti • Allocazione contigua e protezione con registro base e limite • MMU con registro di rilocazione • Svantaggi dell'allocazione contigua • Paginazione • Swapping • Grado di multiprogrammazione e utilizzazione dei processori • Memoria virtuale • Allocazione e sostituzione pagine • Thrashing

11.1 Allocazione contigua e protezione con registro base e limite

Ricapitoliamo un po' di background...

- Le aree di memoria che i processori possono usare direttamente per l'esecuzione dei programmi sono solo la memoria centrale ed i registri.
- Pertanto un programma deve essere portato dal disco in memoria centrale perché possa essere eseguito, e il programma deve avere anche sufficiente memoria centrale per memorizzare i risultati della computazione.
- La memoria centrale ≪vede≫ solo un flusso di indirizzi (richieste di lettura o scrittura) proveniente dal bus di sistema, e non è consapevole di chi ha generato tale flusso.
- Le operazioni sui registri richiedono un ciclo di clock (o anche meno).
- Viceversa, le operazioni sulla memoria richiedono molti cicli di clock, anche diverse centinaia, causando uno stallo (stall) del processore, durante il quale un core multithread può eseguire un altro thread hardware.
  Quando la CPU deve aspettare che il dato venga trasferito dalla memoria, non può computare (= stallo appunto).
  Le CPU moderne tuttavia sono multithreaded, quindi possono eseguire un altro thread hardware **in concorrenza** mentre aspettano che il dato venga trasferito dalla memoria.
- Per aumentare l'efficienza degli accessi in memoria vengono utilizzati diversi livelli di memorie cache tra processore e memoria centrale. L'accesso

alla memoria centrale quindi viene mediato, le memorie cache sono più veloci (pochissimi registri mooolto veloci) della memoria centrale, ma hanno una capacità minore, e contengono una copia dei dati in memoria.
Bene se il dato sta nella cache di primo livello.
Benino se il dato sta nella cache di secondo livello.
Male se il dato sta nella cache di terzo livello.
Malissimo se il dato sta nella memoria centrale.

Il problema dell'allocazione della memoria:

- Perché un processo possa andare in esecuzione la sua immagine (codice + dati statici + stack + heap) deve essere presente in memoria centrale.
- In un sistema multiprogrammato, più immagini di più processi stanno contemporaneamente nella memoria centrale.
- Il sistema operativo deve, pertanto, allocare porzioni di memoria centrale ai diversi processi in funzione delle necessità di tali processi.
- Questo pone diversi problemi:
  - Che **strategie di allocazione** possiamo adottare?
  - Come **proteggiamo** la memoria del sistema operativo dai processi, e quella di ogni processo da ogni altro processo?
  - Se un programma può essere caricato, in momenti diversi, in posizioni diverse della memoria, come possono le istruzioni del programma **referenziare** le locazioni di memoria usate dal programma?

Capitolo 12

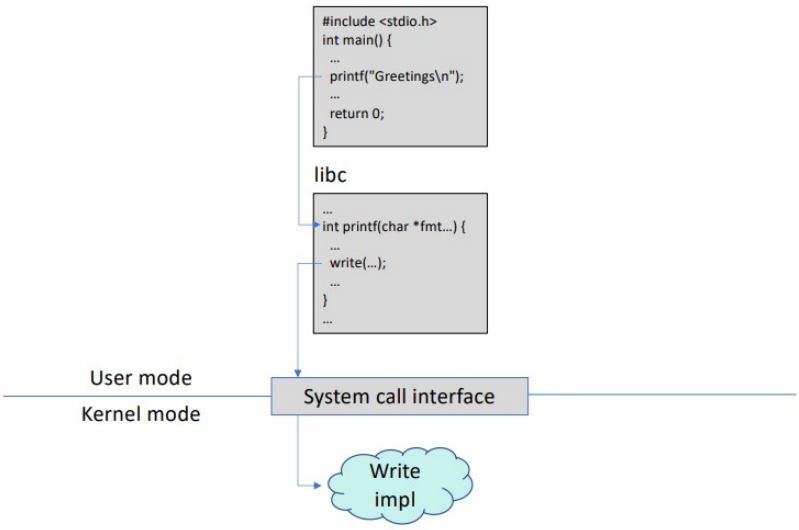
Interfaccia e struttura del kernel

Argomenti

- Implementazione di chiamate di sistema ed API
- Struttura del kernel
- Politiche e meccanismi

12.1 Implementazione di chiamate di sistema ed API

12.1.1 Implementazione API attraverso chiamata di sistema

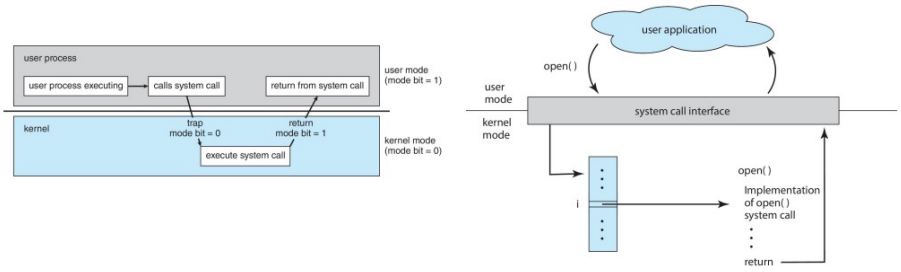


12.1.2 Duplice modalità di funzionamento

- Permette al sistema operativo di proteggere se stesso dai programmi in esecuzione
- La CPU può funzionare in modalità utente (user mode) o in modalità di sistema (kernel mode) impostando un opportuno bit di modalità
- Alcune istruzioni del processore sono privilegiate, ossia eseguibili solo in modalità di sistema: in particolare, in user mode la CPU non può accedere alla memoria del kernel

12.1.3 Implementazione delle chiamate di sistema

- Una chiamata di sistema non è semplice da implementare come una normale chiamata di funzione, perché occorre effettuare una transizione da modalità utente a modalità di sistema
- Prima vengono preparati i parametri necessari:
  - Un numero che identifica quale chiamata di sistema va effettuata...
  - ...più tutti i parametri necessari alla specifica chiamata di sistema
  - (Vedremo nella slide successiva come vengono effettivamente passati questi dati)
- Quindi viene invocata un'opportuna istruzione macchina che genera un'eccezione software; essa fa passare la CPU in modalità di sistema e trasferisce il controllo ad una subroutine ad un determinato indirizzo di memoria
- La subroutine (**system call interface**) legge il numero identificativo della chiamata di sistema, effettua un lookup da una tabella interna dell'indirizzo della routine che effettivamente implementa la chiamata di sistema, e salta a tale indirizzo
- La routine invocata legge i parametri ed esegue la funzionalità richiesta
- Al ritorno il processore passa di nuovo in modalità utente



12.1.4 Passaggio dei parametri alle chiamate di sistema

- Dal momento che l'invocazione delle chiamate di sistema passa per un'eccezione software, il passaggio di parametri è più complesso rispetto a quello di una normale chiamata di procedura
- Metodo più semplice: passare i parametri nei registri del processore
  - Vantaggio: rapido
  - Svantaggio: utile solo per pochi parametri i cui tipi di dati hanno dimensione limitata
- Altro metodo: passo in uno dei registri un indirizzo di memoria ad un blocco nel quale sono memorizzati i parametri
  - Usato da Linux in combinazione con il primo metodo
- Altro metodo: faccio push dei parametri sullo stack
  - Vantaggi: flessibile; simile ad una normale chiamata di procedura
  - Svantaggi: lento e macchinoso

12.1.5 La necessità del doppio stack

Notare che ogni thread ha di solito due stack:

- quello che viene utilizzato dal programma in modalità utente
- ed uno distinto che viene utilizzato quando il thread passa in modalità di sistema

Una chiamata di sistema, come prima cosa, imposta lo stack del thread corrente allo stack di sistema, e al termine della chiamata di sistema ripristina lo stack a quello utente.
Per quale motivo? Per sicurezza: dal momento che il processo potrebbe modificare a suo piacimento il registro stack pointer (che non è privilegiato) non è possibile fidarsi che questo punti ad uno stack «sano»: pertanto in modalità di sistema occorre usare uno stack sicuramente corretto.

12.1.6 Uso delle librerie dinamiche per le API

Si vuole far sì che, anche se il sistema operativo viene aggiornato, non vi sia bisogno di ricompilare/linkare le applicazioni qualora siano cambiate le chiamate di sistema, o l'implementazione delle API, purché l'interfaccia delle API resti la stessa.
Un vantaggio si ha realizzando le API e le librerie standard del linguaggio come librerie dinamiche.
Infatti se queste sono modificate (nell'implementazione, non nell'interfaccia), non occorre ricompilare tutti gli eseguibili per aggiornarli alla nuova versione delle librerie.

12.1.7 Application binary interface (ABI)

Occorre però un'ulteriore accortezza: oltre all'API non deve cambiare l'**application binary interface (ABI)**.
L'ABI è l'insieme delle convenzioni attraverso le quali il codice binario dell'applicazione si interfaccia con il codice binario della libreria dinamica delle API:

- Come si chiama internamente alla libreria la funzione da invocare (name mangling)?
- In che ordine i parametri vengono messi sullo stack delle chiamate?
- Come sono strutturati i tipi di dati? C'è padding? Qual è l'endianess?

12.1.8 La scarsa portabilità degli eseguibili binari

Come facciamo ad avere applicazioni portabili su diversi sistemi di elaborazione?
Tre possibili approcci:

- Scrivere l'applicazione in un linguaggio con un interprete portabile (es. Python, Ruby): l'eseguibile in tal caso è il sorgente
- Scrivere l'applicazione in un linguaggio con un ambiente runtime portabile (es. Java, .NET): l'eseguibile in tal caso è il bytecode
- Scrivere l'applicazione utilizzando un linguaggio con un compilatore portabile ed API standardizzate: l'eseguibile è il file binario compilato e linkato.

Nei primi due casi l'eseguibile è normalmente uno solo per tutte le architetture. Nel terzo caso invece occorre, di norma, generare un eseguibile distinto a variazioni anche minime del sistema di elaborazione (spesso anche solo al variare della versione del sistema operativo).
Come mai?

- Una prima banale ragione può essere la differenza nell'architettura hardware: ad esempio, un file binario prodotto per CPU ARM non può essere interpretato da un sistema di elaborazione con CPU x86-64, dal momento che le istruzioni macchina delle due CPU differiscono
- A parità di architettura hardware sistemi diversi possono supportare API diverse: ad esempio, Windows supporta le API Win32 e Win64 e non le API POSIX, supportate da Linux e macOS
- A parità di architettura e API sistemi diversi possono supportare diversi formati per i file binari: ad esempio, Linux riconosce il formato ELF, mentre macOS riconosce il formato MachO
- A parità di architettura, formato ed API può esservi differenza nelle chiamate di sistema che le implementano (se la libreria delle API è collegata staticamente)
- A parità di architettura, formato ed API, anche se la libreria delle API è collegata dinamicamente (o le chiamate di sistema sono le stesse), può esservi una differenza nell'ABI

- Solo quando tutti questi fattori sono identici un file binario è portabile da un sistema di elaborazione ad un altro

12.2 Struttura del kernel

12.2.1 Sottosistemi del kernel

Basati sulle categorie dei servizi offerti dal kernel stesso (e quindi sulle categorie delle chiamate di sistema). I principali sono:

- Gestione dei processi e dei thread
- Comunicazione tra processi e sincronizzazione
- Gestione della memoria
- Gestione dell'I/O
- File system

12.2.2 Organizzazione del kernel

Il kernel di un sistema operativo general-purpose è un programma

- Di dimensioni elevate e complesso
- Che deve operare molto rapidamente per non sottrarre tempo di elaborazione ai programmi applicativi
- Un cui malfunzionamento può provocare il crash dell'intero sistema di elaborazione

Si pone quindi il problema di come progettarlo in maniera da garantire rapidità e correttezza nonostante dimensioni e complessità. Alcune possibilità:

Struttura monolitica quella che tipicamente intendiamo quando parliamo di kernel

- Il sistema operativo Unix originale aveva una struttura monolitica, dove il kernel è un singolo file binario statico
- Il kernel forniva un elevato numero di funzionalità:
  - Scheduling CPU
  - File system
  - Gestione della memoria, swapping, memoria virtuale
  - Device drivers
  - ...
- Vantaggi: elevate prestazioni
- Svantaggi:
  - Complessità
  - Fragilità ai bug (crash di un singolo driver può causare crash dell'intero sistema)

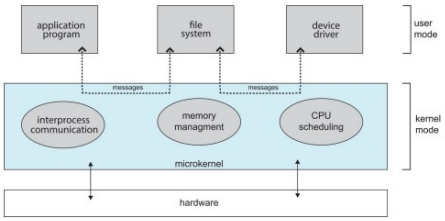
  - Necessità di ricompilare il kernel (e riavviare il sistema) se bisogna aggiornare/aggiungere una funzionalità, ad esempio il driver per una nuova periferica

Struttura a strati

- Negli approcci stratificati il sistema operativo è diviso in un insieme di livelli, o strati
- Lo strato più basso interagisce con l'hardware, lo strato `n-esimo` interagisce solo con lo strato `(n-1)-esimo`
- L'approccio offre due vantaggi:
  - Ogni strato può essere progettato e implementato indipendentemente dagli altri
  - È possibile verificare la correttezza di uno strato indipendentemente da quella degli altri
  - Ogni strato nasconde le funzionalità degli strati sottostanti e presenta allo strato soprastante una macchina dalle caratteristiche più astratte
- In realtà pochi sistemi operativi usano questo approccio in maniera pura:
  - È difficile definire esattamente quali funzionalità deve avere uno strato
  - Ogni strato introduce un overhead che peggiora le prestazioni
- È comunque conveniente strutturare alcune parti del sistema operativo a strati (es. file system o stack di rete)

Struttura a microkernel

- Il principale problema dei kernel monolitici è la loro complessità, e di conseguenza fragilità e inaffidabilità
- La struttura a microkernel sposta quanti più servizi possibile fuori dal kernel in programmi di sistema, mantenendo nel kernel l'insieme minimo di servizi indispensabili per implementare gli altri
- Il kernel è definito microkernel dal momento che ha dimensioni molto ridotte
- L'approccio è stato proposto negli anni 80 con il sistema operativo Mach
- Un microkernel offre pochi servizi, di solito lo scheduling dei processi, (una parte della) gestione della memoria e la comunicazione tra processi
- Gli altri servizi (es. filesystem e device drivers) vengono implementati a livello utente
- Per chiedere un servizio, un programma comunica con il programma di sistema che lo implementa attraverso le primitive di comunicazione offerte dal microkernel



- Vantaggi:
  - Facilità di estensione del sistema operativo: posso aggiungere un nuovo servizio senza dover modificare il kernel
  - Maggiore affidabilità: se un servizio va in crash non manda in crash il kernel; un kernel piccolo può essere reso più affidabile con meno sforzo
- Svantaggi:
  - Overhead: una tipica richiesta di servizio deve transitare dal processo richiedente al microkernel, al processo di sistema destinatario, e viceversa, con molti passaggi tra user e kernel mode, comunicazioni, cambi di contesto...
- I sistemi a microkernel puri vengono usati nelle applicazioni che richiedono elevata affidabilità (QNX neutrino, L4se)
- Altri sistemi inizialmente a microkernel si sono evoluti in sistemi ibridi (Windows NT, Darwin - kernel di macOS e iOS)

Struttura a moduli

- Il kernel è strutturato in componenti dinamicamente caricabili (moduli), che parlano tra di loro attraverso interfacce
- Quando il kernel ha bisogno di offrire un certo servizio, carica dinamicamente il modulo che lo implementa; quando il servizio non è più necessario, il kernel può scaricare il modulo
- Questo approccio ha alcune caratteristiche di quelli a strati e a microkernel, ma i moduli eseguono in modalità kernel, e quindi con minore overhead (ma anche con minore isolamento tra di loro)

Struttura ibrida

- In pratica pochi sistemi operativi adottano una struttura "pura": quasi tutti combinano i diversi approcci allo scopo di ottenere sistemi indirizzati alle prestazioni, sicurezza, usabilità...
- Esempio: Linux e Solaris sono monolitici per avere prestazioni elevate, ma supportano anche i moduli del kernel per poter caricare e scaricare dinamicamente funzionalità
- Altro esempio: Windows inizialmente aveva una struttura a microkernel, successivamente diversi servizi sono stati riportati nel kernel per migliorare le prestazioni. Ora è essenzialmente monolitico, pur conservando alcune caratteristiche della precedente architettura a microkernel. Inoltre supporta i moduli del kernel

12.3 Politiche e meccanismi

- Quando discutiamo come è realizzato il kernel è importante distinguere tra politiche e meccanismi
- Una *politica* dice quando una certa operazione viene effettuata; ad esempio: sotto che condizioni il kernel decide che è il momento di sospendere l'esecuzione di un programma per far riprendere l'esecuzione di un altro?
- Un *meccanismo* spiega come una certa operazione è effettuata; ad esempio: come fa il kernel a sospendere l'esecuzione di un programma in maniera che successivamente possa riprendere? Come fa a ripristinare l'esecuzione di un programma precedentemente sospeso?
- Le politiche impattano profondamente sulle caratteristiche percepite del sistema di elaborazione, i meccanismi no (se sono sufficientemente rapidi)
- I meccanismi sono più stabili delle politiche, che spesso cambiano in funzione delle caratteristiche percepite che vogliamo che il sistema di elaborazione abbia
- Avere politiche configurabili è utile per combattere la maledizione della generalità