

# Sistemi Di Elaborazione e Trasmissione dell'Informazione

## (1)Lezione 1 (21/09/20):

Lezione del 21/09/2020 – Giovanni Chiola  
Link al [video](#) su YT

### RFC

I protocolli di rete vengono identificati secondo la notazione RFC – numero (request for comments). Gli [RFC](#) sono quindi il modo in cui vengono definiti i protocolli internet. Sono documenti di tipo testuale che raccolgono tutte le specifiche di un certo protocollo e altri dettagli di basso livello; se cerco qualcosa su un protocollo è molto utile andare a guardare all'interno del suo RFC.

### Struttura della rete

La rete internet è l'unione, molto complessa, di vari **strati o livelli**.

Esistono 2 approcci per rappresentare la struttura della rete:

- Approccio **ISO/OSI** stratificato in 7 livelli, modello ad oggi non utilizzato. [A Chiola non piace perché il modello non è aderente alla realtà – non parlarne all'esame]
- Approccio Pratico: conforme alla realtà: **Internet** stratificato in 5 livelli

Livello	Nome	Protocollo
5	<b>Applicativo</b>	HTTP (uno dei tanti)
4	<b>Trasporto</b>	TCP/UDP
3	<b>Rete</b>	IP (v4 o v6)
2	<b>Datalink</b>	Ethernet
1	<b>Fisico</b>	-

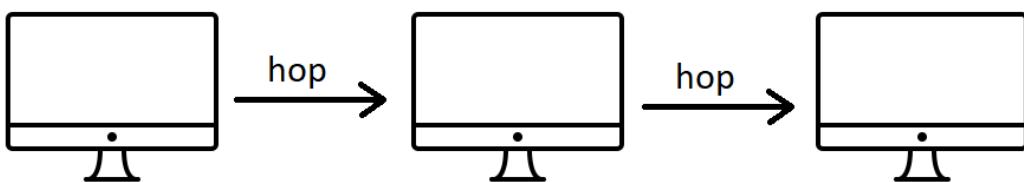
### Invio di un messaggio nella rete

L'invio di un messaggio attraverso la rete avviene tramite 2 dispositivi:

- Host (**Sender**)  
Scrive il messaggio nella RAM – trasferisce il messaggio tramite DMA nella NIC – invia il messaggio in rete.
- Host (**Receiver**)  
Riceve il messaggio attraverso la NIC – lo memorizza nella NIC – la DMA sposta il messaggio in RAM – il Sistema Operativo legge da RAM

Il passaggio del messaggio tramite internet avviene secondo la tecnica del **store & forward**, i dispositivi sono così connessi a coppie.

Per giungere al destinatario può essere necessario passare per Host intermedi; l'instradamento dei messaggi avviene per salti successivi **multi hop**. Tutto questo viene realizzato nel livello 3 “rete”.



## Protocollo IP (Livello 3)

Questo protocollo definisce la comunicazione tra 2 host attraverso il loro indirizzo IP. In pratica il protocollo IP contribuisce alla costruzione di una rete che mette in comunicazione delle sottoreti locali: una rete di reti.

Il protocollo **IP**, insieme ai protocolli del livello di trasporto (TCP/UDP), è il fondamento su cui si basa il funzionamento della rete internet.

Ad ogni PC in rete è assegnato un indirizzo **IP** che ne permette l'identificazione. Ad oggi vengono utilizzati le versioni **IP v4** e **IP v6** [A causa della mancanza di indirizzi si sta effettuando una transizione verso la versione 6]

- **IP v4** riserva **32 bit** per l'indirizzamento (*4 MLD di indirizzi*)
- **IP v6** riserva **128 bit** per l'indirizzamento (*ovvero centinaia di bilioni di indirizzi*)

## Protocolli TCP/UDP e Porte (Livello 4)

La comunicazione a livello di Trasporto avviene tramite protocollo **TCP (stream)** o **UDP (datagram)**.

Il livello di trasporto **stabilisce il modo per indirizzare la singola applicazione all'interno di una macchina**; su un **host multiprogrammato** è possibile far girare contemporaneamente più applicazioni: ognuna di queste può usare il suo protocollo di comunicazione a livello di trasporto, per inviare e ricevere messaggi.

L'indirizzo IP quindi non basta, perché su una singola macchina host possono girare contemporaneamente diverse applicazioni; per identificarle viene utilizzato un ulteriore indirizzo chiamato **porta di comunicazione**, di 16 bit, a cui si collegano le applicazioni.

Vengono quindi utilizzati meccanismi di **multiplexing** e **demultiplexing** che si occupano di raccogliere tutti i messaggi e mandarli in rete nel primo caso, oppure indirizzarli verso l'operazione corretta nel secondo.

In questo modo tutte le applicazioni che girano su un calcolatore possono accedere individualmente alla rete.

Ricapitolando: abbiamo due livelli di indirizzamento:

- a livello di rete si utilizza l'indirizzo IP dell'host per recapitare il messaggio
- una volta arrivato a destinazione, il messaggio deve essere indirizzato alla giusta applicazione tramite le porte di comunicazione; ogni applicazione ha la sua porta.

## Socket (Livello 5)

L'**interfaccia di programmazione per scrivere delle applicazioni** è detta socket: attraverso esso è possibile accedere a tutte le primitive di comunicazione disponibili sulla macchina.

Si accede così ad un'unica versione **virtuale** della rete, che implementa tutte le sue varie funzionalità.

I socket rappresentano quindi delle interfacce necessarie per gestire la rete a livello applicativo.

A questo livello vengono utilizzati svariati protocolli che offrono svariate funzionalità, un esempio è il protocollo **HTTP**.

## (2)Lezione 3 (24/09/20)

### Approfondimento sulla Struttura della Rete Internet

Per gestire la complessità della rete la si stratifica in 5 livelli:

Livello	Nome	Protocollo	In breve
5	<b>Applicativo</b>	HTTP (uno dei tanti)	Applicazioni
4	<b>Trasporto</b>	TCP/UDP	Connessione tra applicazioni
3	<b>Rete</b>	IP (v4 o v6)	Instradamento Multi-Hop
2	<b>Datalink</b>	Ethernet	Interazione mittente-destinatario
1	<b>Fisico</b>	-	Tipo di Canale

## Livello 1: Fisico

Viene specificato il funzionamento della codifica sul canale, ad esempio: canale radio, fibra ottica, ecc...

## Livello 2: Datalink

A livello datalink viene implementata la comunicazione a livello logico e fisico tra il mittente ed il destinatario.

Il protocollo principale del data link è il protocollo Ethernet IEEE 802.3.

Questo livello ed è necessario alla realizzazione dei livelli successivi.

## Livello 3: Rete

La funzionalità principale del livello di rete è di fornire **instradamento**. Sulla base dell'indirizzo del destinatario viene calcolato un percorso che dovrà essere seguito dai pacchetti per arrivare al device destinatario.

A livello di rete abbiamo il protocollo IPv4 e IPv6 i quali differiscono solo per il modo in cui viene codificato l'indirizzo:

- IPv4 su 32 bit
- IPv6 su 128 bit

## Livello 4: Trasporto

La funzionalità principale del livello di trasporto è quella di stabilire una **connessione** tra applicazioni che girano in una certa macchina.

Oltre all'indirizzo che viene specificato al livello di rete, al livello di trasporto, viene specificato il **numero di porta**. Questo numero di porta è **associato ad un processo in fase di esecuzione sulla macchina** e in questo modo tra tutte le applicazioni attive si può scegliere l'applicazione giusta.

A livello di Trasporto abbiamo due protocolli principali, il protocollo TCP e il protocollo UDP.

## Livello 5: Applicativo

Al livello applicativo troviamo tutto il resto, **sono in esso implementate le funzionalità del sistema e della rete; si può scegliere se mandare messaggi piuttosto che scaricare risorse da remoto o upload, streaming, audio, video.**

Vengono in questo livello implementati svariati protocolli dalle differenti mansioni, quali per esempio HTTP (web) o SMTP (mail); tutto questo si basa sulle funzionalità offerte dagli altri livelli.

L'interfaccia di programmazione è l'interfaccia del **socket** grazie alla quale è possibile accedere a tutte le primitive di comunicazione disponibili su quella macchina. In questo modo si accede a una versione virtuale della rete che implementa tutte le capacità dei nostri protocolli di comunicazione.

## Livello 4: Modalità Datagram (Protocollo UDP):

La modalità Datagram si basa su un host mittente, un host destinatario e la rete internet, che attraverso dispositivi chiamati **router** definisce il percorso che devono fare i messaggi (o più propriamente **datagrammi**) per arrivare a destinazione.

UDP funziona quindi attraverso router e datagrammi:

- I **router** implementano l'instradamento di rete; attraverso essi si definisce un percorso calcolato dal livello IP che andranno a percorrere i pacchetti.

La modalità di comunicazione prevede l'invio di messaggi di una certa lunghezza specificata dal mittente.

- Il protocollo UDP sta per User **Datagram** Protocol; i **datagrammi** sono quindi parte fondamentale di esso e li possiamo immaginare come delle lettere da scrivere nel quale abbiamo una intestazione (**header**) che contiene l'indirizzo della porta del destinatario e altre informazioni utili al trasferimento, seguita dal contenuto del messaggio (**payload**).

## Esempio di Comunicazione

Abbiamo un host mittente (A) e un host destinatario (B);

A e B si devono mettere d'accordo in modo da eseguire correttamente le primitive `send()` e `receive()`; a questo punto la comunicazione si divide in 3 passaggi principali:

- 1) Host A esegue una `send()`, la quale termina quando è stato scritto il messaggio nei buffer di trasmissione.

Durante la `send` A deve **specificare l'indirizzo dell'area di memoria che contiene il datagramma che deve essere inviato**. Ci saranno due parametri principali, **il puntatore al primo byte** più un'indicazione alla **lunghezza** espressa in byte del datagramma stesso.

I datagrammi possono essere di lunghezza variabile. Per indicare la lunghezza dei datagrammi si usa un'indicazione numerica su 16 bit quindi possiamo avere **datagrammi di lunghezza compresa tra 0 e 65535 byte**.

- 2) I datagrammi del protocollo UDP, sono quelli implementati a livello di rete dal protocollo IP. UDP non deve fare praticamente nulla, poiché la spedizione dei datagrammi è implementata dal livello 3.

Il datagramma passa quindi dal buffer TX della NIC del mittente fino al buffer RX del primo router; a questo punto il datagramma salta quindi di router in router attraverso vari **hop**: un passaggio delicato dove si rischia di perdere il messaggio.

Una caratteristica importante di questa modalità di comunicazione di tipo datagram è che **non viene garantito il recapito di tutti i messaggi**, quindi alcuni potrebbero andare persi.

**Per questo il protocollo UDP è considerato poco affidabile.**

Se va tutto bene, il messaggio arriva intatto all'host di destinazione.

- 3) Host B esegue quindi una *receive()*, che termina quando riceve il messaggio che si aspettava.

B nella *receive()* **deve specificare un'area di memoria**, indicare un puntatore e una dimensione, passare il puntatore e la dimensione come parametro alla primitiva e **sapere quanto è lungo il messaggio inviato da A**, affinché sia possibile predisporre una quantità di memoria adeguata.

Se la possibile perdita di datagrammi non può essere tollerata, o è necessario inviare messaggi di dimensione superiore a  $2^{16}$ , allora conviene passare alla modalità stream.

## Livello 4: Modalità stream (protocollo TCP):

Risolve il problema dell'affidabilità e della dimensione del messaggio creando un **canale virtuale e bidirezionale di comunicazione**.

Invece che mandare singoli messaggi è come se venisse stabilito un tubo bidirezionale nel quale possono passare informazioni da entrambe le direzioni.

Una volta stabilita questa connessione A può mandare a B una **quantità arbitraria di byte** e viceversa.

I byte mandati da A sono ricevuti da B nell'ordine in cui sono stati mandati; se ad un certo punto A vuole mandare più byte di quelli ricevibili da B, la parte di essi non ancora memorizzata resta sul canale di comunicazione fino a che B non svuota il suo buffer di ricezione.

Se invece B ad un certo punto cerca di ricevere un numero di byte maggiore rispetto a quelli inviati da A, entra in gioco il meccanismo di blocco della *receive()*, che fa in modo che B aspetti fino alla completa ricezione di essi.

## Apertura di connessione TCP attraverso 3-way handshake:

Per poter stabilire questa connessione si deve avere una fase preliminare di apertura di questa connessione chiamata **3-way handshake**.

E' necessario che A e B si comportino in modo diverso, e che **instaurino una relazione di tipo Client-Server**, infatti:

- Uno dei due si deve comportare come **server**, entità passiva;
- l'altro si deve comportare come **client**, entità attiva.

Non importa quale dei due assuma S e C; il client per convenzione è quello che invia il primo messaggio; se tutto va nel verso giusto, il 3-Way-Handshake funziona a grandi linee attraverso 3 passaggi:

- 1) Il client prende quindi l'iniziativa e manda un primo messaggio, in cui esplica la sua volontà di aprire la connessione,
- 2) Il server quando arriva la richiesta di connessione risponde comunicando anch'egli la sua volontà di aprirla.
- 3) Quando il client ha ricevuto una risposta da parte del server, manda un ulteriore messaggio dicendo che la connessione è stata aperta ed è pronta ad essere utilizzata.

### Header TCP:

Abbiamo una parte di intestazione formata da:

- Numero di porta sorgente (16 bit)
- Numero di destinazione (16 bit)
- Numero di **sequenza** (32 bit)
- Numero di **acknowledgment** (32 bit)

Gli ultimi due numeri permettono la realizzazione del livello di astrazione delle stream di byte e quindi permettono di passare da datagram a stream, una volta fatto il 3-way handshake.

Gli ultimi 16 bit sono divisi in:

- una parte che indica la lunghezza dell'header,
- dei bit inutilizzati
- 6 bit dedicati ai **flag**, che servono per gestire tutte le varie situazioni della comunicazione.

I flag sono:

- 1) SYN: Synchronization,
- 2) ACK: Acknowledgement,
- 3) URG: Urgent,
- 4) PSH: Push,
- 5) RST: Reset,
- 6) FIN: Finish.

Numero di Porta Sorgente	Numero di Porta Destinazione				
Numero di Sequenza					
Numero di Acknowledge					
S	A	U	P	R	F
Y	C	R	S	S	I
N	K	G	H	T	N
Lunghezza Header	Receive Window				

I numeri di sequenza e acknowledge sono inoltre e soprattutto dei controlli di sicurezza del 3-way-handshake; sono codici numerici generati casualmente (da 0 a  $2^{32} - 1$ ) che vengono

scambiati da i due host in questione, che assicurano che la comunicazione sia **riservata**, e **forniscono sicurezza e affidabilità**.

In poche parole ci assicurano che nessuno si intrometta nella comunicazione.

La modalità stream permette infatti di mandare sequenze arbitrarie di byte con la certezza che queste vengano ricevute, quindi è considerata **più affidabile**.

## (3)Lezione 5 (28/09/20)

### 3-Way Handshake nel Dettaglio

Il 3-Way-Handshake tra due Host H1 e H2, rispettivamente Client e Server, avviene in 3 passaggi:

#### 1) C → S, ovvero H1→H2

H1 manda un messaggio attivando il flag **syn** per richiedere l'apertura della connessione.

Nel numero di sequenza viene inserito un numero generato casualmente, che chiameremo *x*.

Vengono riempiti i campi riguardanti i numeri di porta, mentre il numero di acknowledge è per ora vuoto.

#### 2) S → C, ovvero H2→H1

Nel caso H2 non voglia aprire la connessione, attiva il flag RST e la comunicazione si ferma, e il 3-WH non va a buon fine.

Se invece H2 volesse anch'egli aprire la connessione, risponde attivando:

- il flag **syn**, che comunica la sua volontà di aprire la connessione.
- il flag **ack**, che comunica che il messaggio di H1 è stato ricevuto correttamente.

H2 inserisce inoltre:

- nel numero di sequenza un numero generato casualmente che chiameremo *y*.
- nel numero di acknowledge andrà ad inserire *x+1*, ovvero:

$$\text{numero di acknowledge} = \text{numero di sequenza della controparte} + \text{numero di byte ricevuti}$$

Numero di sequenza e numero di acknowledge assumono quindi anche il significato di **contatori**, che tengono traccia rispettivamente del numero di byte inviati e ricevuti.

#### 3) C → S, ovvero H1→H2

H1 riceve il messaggio e manda ad H2 un **feedback positivo** che indica la conferma che la connessione è stata stabilita con successo.

Lo fa attivando il flag **ack**, mentre il flag **syn** non serve più e viene quindi spento.

H1 inserisce inoltre:

- nel numero di sequenza il suo  $x+1$ .
- nel numero di acknowledge andrà ad inserire  $y+1$ .

Grazie a questo processo la comunicazione viene mantenuta **privata e sicura grazie ad i numeri di sequenza, riservati ad i due host.**

Dal terzo datagramma in poi i due host possono cominciare a scambiarsi messaggi: ogni byte inviato incrementa di uno acknowledge del ricevente; un esempio di ulteriore comunicazione potrebbe essere:

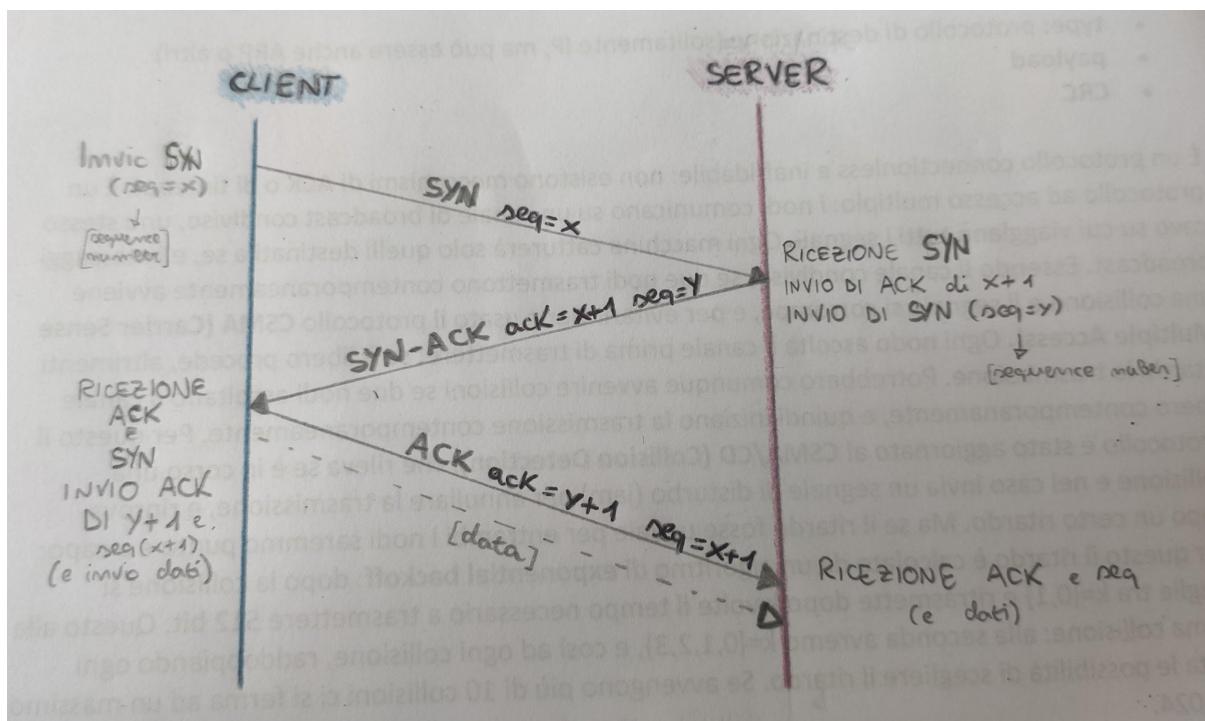
**Sempre nel messaggio 3, ovvero C -> S, ovvero H1->H2**

Oltre al flag **ack** ed i numeri di sequenza e acknowledge, H1 inserisce inoltre nel **payload** il messaggio "ciao". Vengono quindi mandati 5 byte ad H2.

#### 4) **S -> C, ovvero H2->H1**

H2 riceve il messaggio e manda:

- il flag **ack** che comunica che ha ricevuto il messaggio;
- nel numero di sequenza  $y+1$ .
- nel numero di acknowledge  $x+6$ , ovvero  $x+1+5$ .



## Timeout: Buffer e Round-Trip Time

Abbiamo due buffer: **TX buffer e RX buffer.**

La capacità di ogni buffer è di **2048 byte o 1024 parole a 16 bit**.

Nel **TX buffer** viene memorizzato il contenuto del messaggio che deve essere mandato, nel **RX buffer** invece viene memorizzato il contenuto del messaggio che viene ricevuto. Entrambi i buffer vengono mappati nello stesso spazio degli indirizzi. Si accede al buffer **TX con transazioni di scrittura**, mentre si accede al buffer **RX con transazioni di lettura**. Non è possibile leggere dal buffer TX o scrivere nel buffer RX.

Quando uno dei due device, dopo aver mandato un messaggio, non riceve una risposta, deve rimandare il messaggio.

Vi è un buffer che misura il tempo trascorso dall'invio del messaggio e si chiama **Timeout TX buffer** (timeout del buffer di trasmissione).

Il device che manda il messaggio lo salva quindi sul Time out TX buffer ed al raggiungimento di una certa soglia di timeout, se non è stata ricevuta risposta, lo rimanda.

La soglia di timeout dipende dalla distanza dei due host e dalle caratteristiche fisiche di essi; queste variabili vengono considerate attraverso il calcolo di una metrica detta Round-Trip-Time:

Infatti, se l'invio di un messaggio va a buon fine viene calcolato il tempo trascorso tra l'invio del messaggio e la risposta da parte dell'altro device, che viene detto **round-trip time**.

$$RTT = (t_4 - t_1) - (t_3 - t_2)$$

Per stimare la soglia di timeout quindi:

- 1) All'inizio parto da valori molto alti.
- 2) Poi calcolo e faccio una media dei RTT e **pongo la soglia di timeout leggermente maggiore a questa media**.

Quindi:

- 1) Se si perde il messaggio lo reinvio dopo il timeout.
- 2) Inoltre, grazie ai numeri di sequenza e di acknowledge posso gestire anche se il messaggio di acknowledge viene perso;  
In questo caso il mittente che non riceve la conferma della ricezione del messaggio pensa che quest'ultimo sia andato perso, e rinvia quindi il messaggio, come nel caso uno.

**Il ricevente riceve quindi un duplicato, che però riconosce grazie al numero di sequenza:** non considera quindi il payload ma rimanda semplicemente l'acknowledge.

## Piggybacking e Algoritmo di Nagle

Il piggybacking è una tecnica ottimizzazione effettuata dal S.O. dell'host mittente.

Il device che riceve il primo messaggio può decidere di aspettare un breve lasso di tempo prima di mandare la risposta in modo da **mandare un ACK# cumulativo** nel caso ricevesse un altro messaggio subito dopo la ricezione del primo.

Questo processo viene chiamato **piggybacking**, e permette di utilizzare un unico datagramma per rispondere a diversi messaggi.

Esso funziona tramite l'algoritmo di Nagle (**nagle's algorithm**), che consiste nel posticipare l'invio di messaggi brevi in modo da mandarli insieme e ottimizzare l'utilizzo degli ACK.

E' importante notare che il piggybacking potrebbe essere un rallentamento se si inviano solo messaggi brevi; è quindi possibile disattivarlo se necessario, attraverso una chiamata di sistema.

## Banda e Latenza

Sono due concetti fondamentali per apprendere le reti:

$$\text{BANDA} = \frac{\#BYTE}{tempo(s)} \quad \text{velocità di trasmissione dei dati.}$$

$$\text{LATENZA} = tempo(s) \quad \text{tempo trascorso per inviare 1 byte da un device all'altro.}$$

## Vantaggi e svantaggi della modalità di tipo stream (Protocollo TCP)

Il protocollo TCP consiste nel creare un canale di comunicazione tra 2 device costituito come da due tubi: uno per mandare i dati e l'altro per riceverli.

In questo modo dopo il 3-WH non devo più inserire la porta di destinazione.

Un altro lato positivo è che è una **modalità sicura** ovvero non c'è il rischio di perdere i dati che si inviano.

Il lato negativo è che prima di potersi scambiare i dati **bisogna aprire la connessione** utilizzando il protocollo 3-way handshake e questa operazione richiede molto tempo, tempo che non dipende dalla qualità della connessione dei due device ma dalla loro distanza.

Un altro svantaggio è che **bisogna assegnare ai due device il ruolo di server e di client**. Inoltre bisogna anche **chiudere la connessione**.

## Chiusura di una connessione di tipo stream

Vi sono 3 possibili modi per chiudere un canale:

- 1) Il flag **FIN** permette di stabilire in modo indipendente se i due dispositivi hanno ancora bisogno della connessione.  
Quando uno dei due dispositivi vuole chiudere la connessione manda un segnale FIN e l'altro dovrà rispondere con un FIN-ACK.  
Se in un momento uno dei due dispositivi chiude la connessione questa diventa unidirezionale e quel dispositivo può solo ricevere messaggi.
- 2) Un altro modo per chiudere la connessione è quello di mandare un messaggio con attiva la flag **RST** o reset, come quello che si manda quando si rifiuta di aprire la connessione; questo messaggio chiude il canale direttamente e in ambo le direzioni.
- 3) Il canale si può anche chiudere automaticamente quando i due dispositivi smettono di scambiarsi dati.

## Vantaggi e svantaggi della modalità di tipo datagram (Protocollo UDP)

La qualità del servizio dipende dal protocollo IP; in generale UDP risulta più **semplice** e **veloce** di TCP.

Devo, al contrario di TCP, sempre specificare numero di porta del destinatario.

Rispetto alla modalità stream la modalità datagram è **meno affidabile**, infatti potrebbe capitare che il messaggio inviato non venga recapitato oppure mandando più messaggi questi potrebbero arrivare in ordine sparso.

Questa inaffidabilità deriva dal fatto che i messaggi vengono instradati indipendentemente, senza un canale e senza il meccanismo degli ACK.

Il protocollo UDP utilizza un **multiplexer** per direzionare i pacchetti in **uscita dalle porte**, ovvero numeri che identificano i processi in esecuzione, **alla NIC**; Utilizza anche un **demultiplexer** per dirigere i pacchetti **in entrata dalla NIC alle varie porte**.

## UDP vs. TCP: Recap

UDP	TCP
Semplice e Veloce	Lento: Deve aprire (3-WH) e chiudere la connessione
Inaffidabile	Affidabile e Sicuro
Nessuna Relazione tra host	Host in relazione C-S
Devo sempre inserire IP e porta dei due host e l'instradamento può variare	Dopo il 3-WH posso evitare di inserire porta e IP e il percorso dei messaggi è sempre sul canale di comunicazione

## Socket (Livello 5)

La virtualizzazione della rete avviene attraverso i **socket** ovvero interfacce che ci permettono di accedere alle primitive di comunicazione, creando una connessione tra le applicazioni.

Un socket è una sorta di **porta di comunicazione che permette il passaggio di informazioni da un'applicazione alla rete e viceversa**.

Esso deve essere messo in comunicazione con i dispositivi fisici della macchina, in particolare la NIC; in questo modo l'informazione che arriva dalla rete passa attraverso il socket dell'applicazione di destinazione e giunge alla NIC, **permettendo quindi di far passare informazioni dal livello applicativo a quello del sistema operativo** (e viceversa).

Vi sono due tipi di socket:

- **Stream socket**: orientati alla connessione (*connection-oriented*), basati su protocolli affidabili come TCP o SCTP.  
**I socket possono essere visti come una coppia di file per supportare la comunicazione bidirezionale: un file in lettura ed uno in scrittura.**  
Dopo il 3-WH, l'idea è quella di mappare lo stream sulla rete.  
**L'host A può effettuare una serie di operazioni di scrittura sul socket; le informazioni viaggeranno sullo stream attraverso i protocolli TCP/IP e giungeranno all'host B, che potrà leggerle attraverso operazioni di lettura sul socket.**  
**Le operazioni di lettura e scrittura sono supportate dai buffer di trasmissione e ricezione.**

I socket di tipo stream essendo basati su protocolli a livello di trasporto come **TCP**, garantiscono una comunicazione **affidabile, full-duplex, orientata alla connessione**, e con un **flusso di byte** di lunghezza variabile.

- **Datagram socket**: non orientati alla connessione (*connectionless*), basati sul protocollo veloce ma inaffidabile UDP.

## Organizzazione dell'Interfaccia di Programmazione

L'interfaccia di programmazione viene organizzata attraverso varie chiamate di sistema che svolgono una parte delle operazioni legate alla ricezione, alla trasmissione e, in caso di un socket TCP, anche all'apertura della connessione.

La system-call **socket()** sul client **lo inizializza**, aprendo il file di comunicazione nell'applicazione; la chiamata restituisce un intero che corrisponde al file descriptor associato ad esso.

Differenziamo ora le due modalità;

Socket di tipo Stream

Il passo successivo consiste nello stabilire una connessione, ponendo in una relazione Client-Server i due Host.

### Lato Client:

Il client esegue la funzione **connect()**, che **permette di mandare un messaggio SYN verso il Server, ovvero di richiedere l'apertura della connessione.**

### Lato Server:

Il server procede anch'esso alla **creazione di un socket di tipo stream, attraverso la funzione socket()**; il socket agirà in modalità server, aspettando una richiesta di apertura

della connessione da parte del client; per fare ciò sono necessari una serie di passi intermedi:

- Dopo aver creato il socket **il server chiama la funzione *bind()* che definisce il numero di porta sulla quale avverrà la comunicazione e quindi sulla quale il client dovrà fare la sua *connect()*.**  
Attraverso la *bind()* il server può comunicare al S.O. il suo numero di porta, permettendo così il demultiplexing delle informazioni che giungono dalla rete.
- A questo punto **il server esegue la funzione *listen()* per informare il S.O. che è disposto ad accettare connessioni sulla porta indicata dalla *bind()*.**
- Dopo la ricezione del messaggio SYN da parte del client, comunicata al socket dal S.O., **il server, se vuole aprire la connessione, deve utilizzare la funzione *accept()* per inviare un messaggio SYN/ACK e aprire la connessione.**

La *accept()* è una funzione **bloccante**, ferma l'applicazione, che può ripartire solo dopo aver aperto la connessione.

Essa restituisce un intero corrispondente ad un file descriptor che identifica un altro socket.

Il server utilizza quindi  $n+1$  socket:

- uno (il primo) su cui sono state effettuate *bind()* e *listen()* e che si occupa di ricevere e gestire le richieste di connessione e di creare gli altri socket,
- gli altri  $n$  socket sono per le  $n$  connessioni con gli  $n$  client collegati al server; essi rispondono alle eventuali *send*, *receive*, *write* e *read*.  
Questi  $n$  socket mantengono comunque il numero di porta del socket principale.

## Socket di tipo Datagram

Rispetto alla versione di tipo stream i socket rimangono invariati, una non c'è più l'idea di poter andare a leggere e scrivere sopra come se fossero file.

All'interno dei socket di tipo datagram è possibile scrivere una lettera con degli header in cui i **viene specificato l'indirizzo del destinatario**.

Per fare questo vengono utilizzate delle primitive di comunicazione diverse rispetto a quelle della modalità stream, dette ***send\_to()* e *recvfrom()***.

La *send\_to()*, rispetto alle primitive *send()* e *write()* della modalità stream, prevede che venga specificato l'indirizzo del destinatario.

Quando ricevo un messaggio attraverso la *recvfrom()* viene specificato **l'indirizzo del mittente**.

Per poter avviare la comunicazione non è necessario stabilisce dei ruoli, ma è comunque **necessario che all'inizio uno dei due host conosca l'indirizzo dell'altro**.

Dopo il primo messaggio, attraverso la *recvfrom()* anche il secondo host conoscerà l'indirizzo del primo, a cui potrà inviare messaggi.

Per realizzare una comunicazione di tipo datagram davvero innanzitutto utilizzare la syscall `socket()` per la **creazione dei socket**. Questa system call **restituisce un intero che corrisponde ad un file descriptor** (i socket sono file descriptor).

Una volta creati i socket **almeno uno dei due deve richiamare la primitiva `bind()`** per poter informare il sistema operativo della connessione che si vuole stabilire. Attraverso questa syscall viene assegnato il numero di porta.

Tra i due host deve esistere una convenzione: **uno dei due deve aver comunicato all'altro in precedenza il numero di porta che si vuole utilizzare**.

L'altro host può così procedere all'invio del messaggio, **senza necessariamente effettuare una `bind()`**: inviando un messaggio attraverso la `sendto()` il sistema operativo **assegna automaticamente un numero di porta all'applicazione**.

## (4)Lezione 8 (06/10/2020)

### DNS:

Il DNS (domain name system) è un sistema dei nomi di dominio, ovvero complesso di server molto grande che si occupa di tradurre gli **indirizzi web in IPv4**.

I client si connettono a dei server locali in modalità datagram **UDP sulla porta 53**, mandano un datagramma contenente la stringa da tradurre e il server risponde con la sua traduzione.

### Distribuzione:

I server si distribuiscono ad albero:

- La radice, il **Root Nameserver**: risponde alle richieste di risoluzione dei nomi riguardanti il *namespace* del dominio principale (detto *root*, radice). Il suo compito è quello di **reindirizzare le richieste relative a ciascun dominio di primo livello (top-level domain) ai nameserver propri di quel TLD**.
- **Top-Level Domain**: è l'ultima parte del nome di dominio internet. Corrisponde alla sigla alfanumerica che segue il 'punto' più a destra dell'[URL](#), per esempio: l'indirizzo Internet di Pornhub è *pornhub.com* quindi la parte dell'indirizzo web che ricade entro il dominio di primo livello è *com*. **Ogni TLD gestisce un dominio**.
- **DNS autoritativi**: sono i DNS che contengono i dati specifici del nome del dominio, rispondono alle richieste per quel dominio e ne forniscono i dati relativi (web, mail, ftp, ecc.). Proprio perché i DNS autoritativi **contengono i dati specifici del nostro dominio, sono gestiti dalla società che ci ha fornito il nome di dominio**.

Il client si connette al server locale, il quale si connette con un altro server il quale a sua volta si connette con un altro e così fino ad arrivare alla radice.

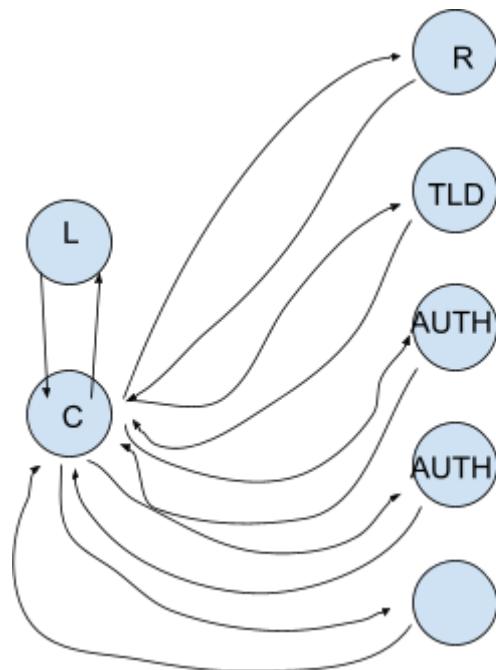
### Algoritmi di ricerca:

Vi sono due algoritmi di ricerca: ricorsivo e iterativo.

In quello **ricorsivo** il server locale contatta il Root nameserver, il quale avendo l'elenco di tutti i Top-level domain, va a contattare quello che gestisce il dominio che il client sta cercando. Questo poi andrà a contattare il server autoritativo.

**L'algoritmo ricorsivo non è molto efficiente.**

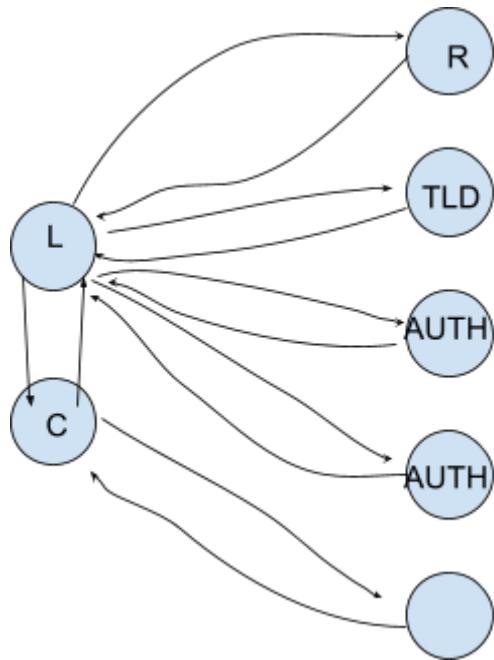
Nell'algoritmo **iterativo** invece chi sa di più cerca di dare subito una risposta: il cliente contatta il server locale il quale risponde con l'IP del root. Il client contatta il Root il quale risponderà con l'IP del TLD e così via fino ad arrivare alla pagina che si sta cercando.



In pratica **non si usa né l'algoritmo ricorsivo né quello iterativo ma vengono combinati: la ricerca dei nomi è iterativa mentre il server locale è ricorsivo.**

Il client contatta il server locale, questo contatta il root, riceve la risposta e contatta il TLD e così via;

Praticamente **il server locale si comporta come un intermediario, in questo modo il client non deve inoltrare troppe richieste perché lo fa il server locale.**



I vantaggi di questa combinazione sono che si crea **meno carico sulla gerarchia dei server** dal momento che essi inoltrano subito la risposta, non c'è ritardo e non devono memorizzare nulla.

**Dal momento che si utilizzano datagrammi per la comunicazione può capitare che vengano persi alcuni messaggi, per ovviare a ciò i server locali devono mantenere a mente le richieste mandate e ancora senza risposta.** Dopo una certa quantità di tempo se non hanno ancora ricevuto una risposta rimandano la richiesta (**timeout**).

Il server locale alla fine di tutto questo procedimento conoscerà la risposta alla richiesta fatta dal client e può tenerla memorizzata nella sua **cache** in modo che se un altro client manda la stessa richiesta potrà ricevere immediatamente una risposta.

Gli elementi in cache possiedono un **TTL** (time to leave), un tempo dopo il quale vengono cancellati dalla cache.

## Header DNS

I messaggi che si scambiano i server sono decodificati in una struttura:

- **Identificatore**: identifica univocamente una richiesta; andremo a vedere la sua utilità contro il DNS Cache Poisoning.
- **Richieste**: una stringa alfanumerica con cui vengono identificati gli host sulla rete, ovvero: una serie di caratteri separati da punti che vanno ad identificare quindi TLD e server autoritativi.  
*ex. www.servizionline.unige.it*
- **Risposte (non Autoritative)**: Risposte contenenti l'indirizzo IP cercato **acquisito dalla cache** ed il suo TTL.

ID	FLAG
#Richieste	#Risposte
#Risposte Autoritative	#Risposte Addizionali
Richieste	
Risposte Normali	
Risposte Autoritative	
Risposte Addizionali	

- **Risposte Autoritative:** Risposte contenenti l'indirizzo IP acquisito attraverso l'algoritmo di ricerca ricorsivo/iterativo, **senza l'utilizzo della cache**, con il suo TTL.
- **Risposte Addizionali:** Indirizzo IP del server autoritativo: necessario per verificare la validità di una risposta autoritativa; non sempre viene eseguita questa verifica.

## Esempio di Funzionamento

- 1) Client cerca prima nella sua **cache locale**, in un file di associazioni statiche tra indirizzi web e IP, chiamato `/etc/hosts`.  
Se non trova niente, allora richiede un indirizzo IP corrispondente ad un indirizzo web al server locale.
- 2) Il server lascia intatto l'indirizzo web richiesto all'interno del suo datagramma; dato che è stateless questo passaggio è fondamentale per ricordare e inoltrare la richiesta iniziale.
- 3) A questo punto il server cerca la risposta nella sua cache, e può:
  - trovare in cache la risposta pronta e fornire una risposta non autoritativa;
  - o, se non ha la risposta in cache, attivare il meccanismo iterativo e interrogare la gerarchia dei server.

## Dettagli sul DNS

Ci sono **13 copie del Root nameserver**, ciascuna con un indirizzo IP diverso, in modo da **distribuire uniformemente il carico di richieste** e non sovraccaricarne uno solo.

Oltre a queste 13 vi sono delle altre copie: infatti, sono i **provider** che si occupano di distribuire le richieste ai Root, ed ogni provider ha il suo nameserver (**nameserver e root nameserver sono due cose diverse**).

In questo modo ogni provider diminuisce il più possibile la distanza del nameserver rispetto agli host, ed in questo modo diminuisce i RTT e rende la connessione più efficiente.

Anche i client possono operare in modalità **stateless** ovvero non memorizzano la richiesta che hanno fatto la quale verrà riportata dal server locale assieme alla risposta ad essa.

## DNS Cache Poisoning

Un utente malevolo potrebbe intercettare la richiesta mandata da un cliente in modo da modificare la risposta da parte del server locale.

Per fare ciò però l'utente malintenzionato **deve conoscere sia l'ID della richiesta del client** **il che suo numero di porta, entrambi generati casualmente ad ogni richiesta**, i quali sono codificati in 16 bit ciascuno e questo vuole dire che esiste una possibilità su  $2^{32}$  di imbroggiare la combinazione giusta.

Se il teppista riuscisse a fare ciò **nella cache** del server locale **verrebbe memorizzata la risposta falsa** e tutti gli utente connessi a quel server locale che andranno a mandare la stessa richiesta del client che è stato intercettato riceveranno quella finta risposta.  
I browser possiedono una memoria nella quale sono salvati gli indirizzi IP più famosi in modo da non mandare una richiesta al server locale nel caso in cui l'utente cerchi di accedervi.

## (5)Lezione 9 (08/10/2020)

### Ancora sul DNS

Il protocollo DNS **utilizza principalmente** come livello di trasporto **il protocollo UDP**. Solo in rari casi e con una modalità di funzionamento diversa utilizza connessioni di tipo stream.

In entrambi i casi, a livello applicativo dobbiamo programmare queste applicazioni attraverso **la primitiva dei socket**;

Per esempio: poniamo di avere un'applicazione che gira su un certo host H1 e un'altra applicazione che gira sull'host H2; in mezzo ad H1 e H2 c'è la rete, in qualche modo H1 e H2 sono connessi alla rete e quindi questo schema è valido indipendentemente da che modalità usiamo, dobbiamo comunque aprire un **socket** sull'applicazione dentro H1 e H2 e stabilire in qualche modo la connessione virtuale tra il socket e la scheda di interconnessione di rete.

### Comunicazione di tipo Datagram

Chiunque può mandare datagrammi a chiunque è connesso alla rete, ma necessita di conoscere l'indirizzo IP del destinatario e il numero di porta dell'applicazione.

Ogni datagramma è una lettera che viene inviata sulla rete; il rischio principale è il fatto che essi possano **perdersi nella rete** e non arrivare a destinazione.

Nella comunicazione di tipo datagram:

- 1) Vengono utilizzate due primitive diverse:
  - **send\_to()**: per l'invio di messaggi; deve specificare l'indirizzo del destinatario,
  - **receive\_from()**: per la ricezione di messaggi; ci dice l'indirizzo del mittente (IP e numero di porta)
- 2) Per avviare la connessione serve che uno conosca l'indirizzo dell'altro e vengono usate le system call:
  - **socket()**: creazione di punti di ricezione datagrammi, i socket restituisce un file descriptor,
  - **bind()**: assegna il numero di porta.

Il numero di porta del server è **stabilito a priori**; ad esempio, nel protocollo DNS, il numero di porta dei server, definito da IANA, è 53.

Esempio di comunicazione:

- 1) A conosce a priori IP e porta del server; crea il suo socket con la primitiva `socket()` e il suo numero di porta con la primitiva `bind()`.
- 2) A fa una `send_to()` a B, inviandogli un messaggio, contenente anche il suo indirizzo IP e il suo numero di porta.
- 3) B tramite la funzione `receive_from()` riceve il messaggio e identifica il mittente tramite IP e numero di porta.

I datagrammi IP sono caratterizzati da una **dimensione massima di  $2^{16}$  Byte**, questo pone **un limite sulla dimensione massima che possono avere i messaggi**.

## Comunicazione di tipo Stream

Il protocollo DNS prevede la possibilità di usare anche **il protocollo di trasporto TCP per superare questo limite**. Questo processo è **molto costoso** perché bisogna per forza fare il 3 way handshake all'apertura delle connessioni.

### Apertura della Connessione

Poniamo di avere un server ed uno o più client:

- 1) Ognuno dei **client** deve richiamare la primitiva `socket()` in modo da consentire la comunicazione con il sistema operativo.
- 2) Anche **server** deve aprire un **socket** per consentire la comunicazione con il SO.

Quando si crea un socket bisogna decidere a priori se è un socket di tipo stream o di tipo datagram, in questo caso la comunicazione è TCP quindi sarà di tipo stream.

- 3) Dal lato **client** bisogna conoscere indirizzo IP e numero di porta, in modo da poter chiamare la funzione `connect()` che da avvio al messaggio *syn* per stabilire la connessione con il 3-way handshake.
- 4) Dal lato **server** bisogna fare la `bind()`, creando la porta in relazione al socket.
- 5) Il server esegue poi una `listen()` che gli permette di ricevere il messaggio *syn*;
- 6) la `accept()` infine, permette al server di mandare un messaggio *syn/ack* al client, che permette di avviare una connessione.

Il primo client manda un messaggio di tipo *syn* sulla porta concordata; attraverso le primitive `socket()`, `listen()` e `accept()` il server si crea un nuovo socket e lo connette con il client C1.

A questo punto il client C2 può chiedere una connessione mandando un messaggio *syn*, e ripetendo il procedimento il server può crearsi un altro socket connesso con il client C2 (e così via).

Avremo quindi **un socket che resta in ascolto e si occupa di stabilire le connessioni** con i nuovi host; si aggiungono ad esso un numero di socket che hanno una connessione in corso con i rispettivi host.

Per esempio, per gestire la comunicazione stream con 3 client il server avrà aperto 4 socket,

in modo da poter gestire le connessioni usando socket diversi. Il server può quindi comunicare con tanti client, ciascuno di essi individuato da un *socket*.

## Prestazioni di una Rete

Le prestazioni di una rete **non sono facili da calcolare**: nell'invio di un messaggio da A a B sono molte le strade possibili nella rete che gli permettono di arrivare a destinazione attraverso il processo multi-hop.

I due host utilizzano le due seguenti funzioni:

- 1) Host A chiama la funzione **Write(buf, n)** con:
  - *buf* il buffer di partenza del messaggio;
  - *n* il numero di byte da inviare.
- 2) Host B chiama la funzione **Read(buf, n)** con:
  - *buf* il buffer di arrivo del messaggio;
  - *n* il numero di byte da ricevere.

Se queste due funzioni sono poste in **modalità bloccante**, vuol dire che **la loro esecuzione blocca il resto dell'applicazione**.

## Calcolo del Tempo tra Partenza e Arrivo del Messaggio

Per calcolare il tempo tra partenza e arrivo di messaggi da un Host A ad un Host B utilizziamo un'applicazione chiamata **Ping Pong**; essa ci permette di inviare più volte lo stesso messaggio per stimare al meglio il risultato.

Gli host possono quindi calcolare al meglio il tempo trascorso tra:

- la *write()* di A,
- la *read()* di B.

Possiamo quindi calcolare:

$$\begin{aligned}t_{andata} &= t_{read_B} - t_{write_A} \\t_{ritorno} &= t_{read_A} - t_{write_B}\end{aligned}$$

E mettere assieme questi due dati nella misurazione del **Round-Trip-Time**:

$$RTT = t_{andata} + t_{ritorno} = (t_{read_A} - t_{write_A}) + (t_{read_B} - t_{write_B})$$

Possiamo inoltre **mettere in relazione tempo e dimensione del messaggio** grazie al modello **banda-latenza**.

## Modello Banda-Latenza

Il modello B-L descrive la seguente legge, che calcola il **delay nella comunicazione tra A e B**:

$$D = L_0 + \frac{N_{byte}}{B} \quad \text{con } L = \text{Latenza e } B = \text{Banda}$$

Per calcolare Latenza e Banda devo usare l'applicazione Ping-Pong per studiare questi 2 casi:

- caso di  $N_{byte}$  minimi, in ogni caso più piccoli possibile ad esempio mando solo l'header di un datagramma.
- caso di  $N_{byte}$  massimi (più grandi possibile);

E creare quindi un sistema a due equazioni e due incognite:

$$\begin{cases} D_{max} = L_0 + \frac{N_{byte_{max}}}{B} \\ D_{min} = L_0 + \frac{N_{byte_{min}}}{B} \end{cases}$$

Con  $D_{max}$  e  $D_{min}$  noti.

Posso dunque calcolare la Banda e la Latenza.

## Lezione 11 (11/10/2020)

### Protocollo NTP

Il protocollo NTP (Network Time Protocol) è un **protocollo fondamentale per la sincronizzazione degli orologi di una rete**.

Sincronizzare gli orologi è fondamentale per meccanismi che comparano i tempi tra più macchine come il **caching**.

L'NTP è un protocollo client-server appartenente al livello applicativo ed è in ascolto sulla porta **UDP 123**.

In pratica **gli orologi sono contatori** che si incrementano alla frequenza di 1Hz (un incremento al secondo).

Il contatore era originariamente **diviso in due sezioni**:

- Una contava i **secondi**:
  - In unix a 32 bit con segno (negativo prima dell'epoca)
  - In UTC a 32 bit senza segno

- Una contava i **nanosecondi**: a 32 bit senza segno sia in unix che in UTC.

**Il valore 0 del contatore**, ovvero la data e l'orario di partenza di esso è detto **epoca**:

- Per Unix: **01/01/1970**
- Per UTC: **01/01/1900**

Il problema che sorge da queste prime affermazioni è che il numero di bit ristretto fa sì che **entrambe queste versioni vadano in overflow**:

- **Unix dopo il 2038**
- **UTC dopo il 2036**

Ovvero: **non ci sono abbastanza bit per contare i secondi dopo queste due date**.

Si è passati quindi a **64 bit**:

- i secondi possono quindi essere contati **per miliardi di anni**;
- l'altro contatore non conta più i nanosecondi ma i **picosecondi**; il maggior numero di bit riesce quindi a dare più precisione al contatore.

## Funzionamento del Protocollo

Bisogna effettuare una **sincronizzazione** dell'orologio di un client rispetto ad un server.

Siccome NTP sfrutta UDP, la **poca affidabilità dei messaggi viene risolta interrogando più server**; si va quindi ad aumentare la ridondanza e a verificare quale sia il server più affidabile.

I server sono classificabili in base alla loro precisione grazie al loro **strato**, ovvero alla loro vicinanza rispetto ad un **orologio atomico**:

- Un server con al suo interno un orologio atomico è detto di **strato 0**,
- Un server collegato ad uno di strato 0 è detto di **strato 1**,
- Un server collegato ad uno di strato  $n$  diventa di strato  $n+1$ ,
- Siccome vi sono 4 bit allocati per lo strato, il massimo  $n$  possibile è 15, e per convenzione identifica un server **non sincronizzato**.

Una volta sincronizzato il client al server di strato  $n$ , esso diventa quindi di strato  $n+1$ .

La sincronizzazione può portare dei **problemi alle applicazioni che stavano usando l'orologio nel momento di cambio dell'ora**.

Per risolvere questo aspetto, al cambio dell'ora, agisco non sul tempo ma **sulla frequenza**, in pratica:

- aumento la frequenza se l'orologio è indietro,
- la diminuisco se l'orologio è avanti.

## Scelta del Server

Abbiamo detto che, siccome NTP sfrutta UDP, la poca affidabilità dei messaggi viene risolta interrogando più server; per trovare il server più affidabile si procede in questo modo:

- 1) Vengono interrogati quanti più server possibili per quanti più round possibili.
- 2) Dopo  $n$  round cerco di stabilire quale sia il server più affidabile, in base al numero di risposte che ho ricevuto e che non sono andate perse.
- 3) Una volta scelto il server e dopo la sincronizzazione con esso, continuo a fare interrogazioni (con minor frequenza) per verificare che sia ancora il server più affidabile.

Nella scelta del server tengo anche in considerazione:

- Lo **strato** (la precisione) di esso,
- Il **round-trip-time**: se ho un RTT alto vuol dire che il server è più lontano e/o lento nella trasmissione del messaggio,
- La **variazione dei round-trip-time**: se un server ha RTT che oscilla di molto vuol dire che cambia spesso in stradamento ed è quindi poco affidabile.

Via via diminuisco l'interrogazione ai server, ovvero la frequenza dei round, poiché ho già un orologio preciso, mi basta verificare ogni tanto che resti il migliore tra quelli possibili.

Arrivo quindi ad avere una precisione nell'ordine **dei millisecondi**.

## Sicurezza nel NTP

La mancata sincronizzazione dell'orologio di un dispositivo può portare a varie problematiche di sicurezza: un esempio è legato ai **certificati digitali**.

In pratica un certificato digitale è un'attestazione di proprietà di una chiave pubblica crittografata, su cui si basano molti meccanismi della rete.

In pratica, questi certificati hanno una data ed un'ora di scadenza; se portiamo indietro il nostro orologio allora potremmo rischiare di accettare certificati ormai scaduti.

Inoltre il protocollo NTP, derivando dall'UDP, soffre di **spoofing**, ovvero la falsificazione della propria identità.

Il protocollo NTP è quindi ad esempio sfruttato per attacchi **DDos**: in pratica, un host malevolo potrebbe usare l'IP dell'host vittima ed inoltrare con quell'indirizzo moltissime richieste NTP a svariati server; in questo modo, l'host vittima riceverà tantissime risposte da i vari server che impediranno il corretto funzionamento della macchina.

Per questo esiste anche un protocollo chiamato **NTPsec**, dove sec sta per *security*; è quindi una versione di NTP più sicura.

## Esempio di Uso di NTP

Un esempio di uso del NTP è il **sistema GPS**.

Esso è basato sulla misurazione dei tempi di orologi atomici contenuti all'interno di satelliti; in pratica tramite l'interrogazione di più satelliti, si può capire una determinata posizione.

## SNTP

Esiste una versione semplificata di NTP, ovvero il protocollo **SNTP** (Simplified Network Time Protocol); questa versione porta ad **un minore dispendio di memoria e batteria, ma anche ad una minore precisione degli orologi**: viene ad esempio utilizzata nei dispositivi mobile.

# (6)Lezione 12 (13/10/20)

## Laboratorio Ping-Pong

### Consegna:

Implementazione di un server "pong" e di due client "ping" nelle due versioni UDP e TCP.

Per compilare da shell lanciare "make" dalla directory che contiene il Makefile.

Nelle directory sono anche presenti dei file CMakeList.txt, che permettono di compilare tramite CMake, o aprire il tutto come "progetto" di CLion (<https://www.jetbrains.com/clion/>).

### Esecuzione:

Usiamo il protocollo **TCP** quindi il client dovrà creare il socket di tipo stream.

In seguito si usa la funzione *connect()* con la porta 1491.

Possiamo inizialmente testare il nostro programma *pong\_server* sulla nostra stessa macchina e se funziona possiamo connetterci al server dell'università (**webdev.dibris.unige.it**) con IPv4 130.251.61.30

Per poter lavorare in locale si usa il nome */localhost* a cui corrisponde l'IP 127.0.0.1

All'interno di *bin* ci sono tre eseguibili di nome:

- gc\_pong\_server
- gc\_tcp\_ping
- gc\_udp\_ping

Il client chiede l'apertura della connessione con la *connect()*, avviene la **Three-way handshake**, si specifica la dimensione del file e il numero delle ripetizioni e se server da l'OK si può partire.

Il primo messaggio dovrà contenere un numero progressivo che corrisponde al numero dei messaggi inviati (prima volta 1 poi 2 ecc...). Il server legge il numero di ripetizioni da fare e manda indietro il messaggio il numero di volte che gli abbiamo detto.

A ogni ripetizione si fa partire un timer che misura il tempo di trasmissione:

- $t_1$  = il client inizia ad inviare il file
- $t_2$  = il server riceve tutto il file
- $t_3$  = il server ha terminato di inviare tutto il file
- $t_4$  = il client ha finito di ricevere tutto il file

Nel momento in cui abbiamo tutti i valori di  $t$  possiamo calcolare il **Round Trip Time: ( $t_4 - t_1$ ) - ( $t_3 - t_2$ )**.

Nel *pong\_server* c'è il file completo, le parti che ci mancano in *tcp\_ping* mancano le possiamo reperire da *pong\_server*.

In seguito bisognerà compilare anche il file che contiene le statistiche di **RTT** e poi quello che contiene le parti ausiliarie per l'**UDP**.

infine dovremo usare degli script bash che serviranno per produrre grafici riguardanti il tempo di **throughput**.

Per passare il numero di porta e l'indirizzo IP alla *connect()* si usa la funzione *getaddrinfo()* che trova attraverso il DNS tutta la roba che ci serve.

La parte del completamento del server è **opzionale**.

## (7)Lezione 13 (15/10/2020)

### Protocollo FTP

Il protocollo FTP (File Transfer Protocol) è un **protocollo** di livello **applicativo** per la **trasmissione di dati tra host** basato su TCP e con architettura di tipo client-server.

FTP è **uno dei primi protocolli** definiti della Rete Internet e ha subito una lunga evoluzione negli anni:

La prima specifica risale al 1971 (RFC-114);

Prima del 1980 sfruttava NTP; la prima versione che sfrutta TCP risale al 1980 ed è spiegata nel RFC 765.

L'attuale specifica fa riferimento all'**RFC-959**.

Originariamente erano state definite definite insieme al protocollo due modalità: una **attiva** ed una **passiva**.

### Modalità Passiva

Abbiamo a disposizione un client ed un server.

Una delle connessioni che si instaura tra essi si chiama **connessione di controllo**: client e server entrano in comunicazione sulla **porta 20**, il client manda dei comandi sotto forma di stringhe ed il server li esegue.

I due comandi principali sono **GET** (download) e **PUT** (upload).

I comandi sono codificati in ASCII, attraverso 4 caratteri maiuscole e i più importanti si occupano del trasferimento di file da server a client e viceversa.

**Per effettuare un trasferimento** si apre un'**altra connessione** sulla porta **TCP/21** e si mandano i **dati** codificati come interi a 8 bit.

I file possono essere trasferiti sequenzializzandoli (mando i byte uno dietro all'altro).

Per utilizzare il protocollo FTP **un dispositivo deve essere autenticato anche sul server**, non solo sul client.

**In questo modo si risolve il problema dei permessi** (i permessi non sono più legati alla macchina ma all'account). Per poter identificare il file devo indicare un [pathname](#) per il file, uno per la directory sorgente e uno per quella di destinazione.

Quando faccio il login sul server **ho a disposizione due directory: una locale e una in remoto**, stessa cosa per la shell.

La directory remota si cambia con *cd* mentre quella locale con *lcd* (local change directory). Se volessimo listare i file presenti in una directory remota potremmo farlo col comando *dir*, il quale risultato verrà recapitato al client attraverso la connessione dati, come d'altronde viene fatto per tutti i comandi eseguiti in remoto.

## Modalità Attiva (Default)

La **modalità attiva** fa gestire la connessione dati al server, il quale si occupa di stabilire la connessione col client.

**Il server apre la connessione dati mentre il client apre la connessione controllo.** Per aprire la connessione dati **il server deve conoscere l'indirizzo IP e il n° di porta del client** i quali sono **contenuti nella richiesta della connessione controllo** da parte del client sulla porta 20.

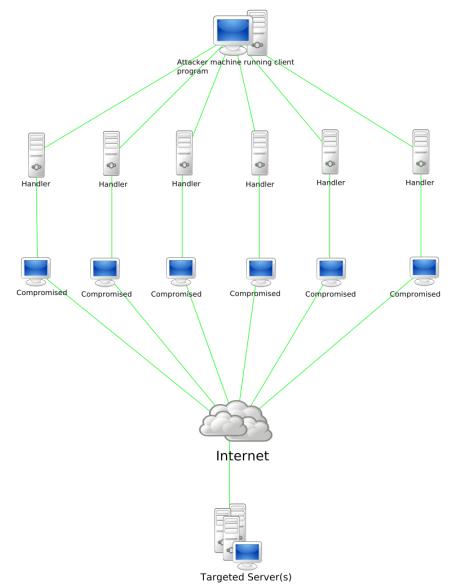
La modalità attiva è ormai **obsoleta**;

Infatti con questa modalità vi erano **problematiche di sicurezza**:

- Un utente avrebbe potuto mandare al server un indirizzo IP diverso dal suo in modo da far scaricare dal server uno o più file su un altro dispositivo potendo dunque compiere un **attacco DDoS**.

Un attacco [DDoS](#) consiste nel intasare la connessione del dispositivo vittima facendogli scaricare un numero altissimo di file da diversi server.

Un'altra problematica poteva essere che un server contattasse un client senza nessuna autorizzazione; quest'ultima problematica è stata poi risolta con l'introduzione dei **Firewall**.



Passare da una modalità all'altra

Per passare dalla modalità attiva a quella passiva bisogna dare il comando **passv**;

## Difetti e Alternative al FTP

Il protocollo FTP risulta svantaggioso perché:

- 1) deve gestire due connessioni (l'HTTP solo una),
- 2) obbliga un utente ad autenticarsi (HTTP è invece libero da questo vincolo),
- 3) l'autenticazione utilizza una password non cifrata, che comporta un rischio di sicurezza.

L'alternativa al protocollo FTP è quindi l'**HTTP**, usato maggiormente per il download, che usa una sola connessione per i comandi e dati e non usa l'autenticazione sul server.

Nel protocollo FTP l'autenticazione avviene usando username e password ma il protocollo non ha **nessun metodo di criptatura** dei datagrammi rendendo le credenziali vulnerabili allo *sniffing*.

Per questo vengono introdotte connessioni sicure **SFTP** che cifrano i dati scambiati; questo protocollo viene creato combinando l'FTP al protocollo **TLS** (Transport Layer Security), che permette la cifratura dei socket e quindi delle connessioni.

Dato che il protocollo FTP prevede anche la possibilità di fare upload nei server vi è il rischio che qualcuno carichi file dannosi ma, dal momento che i client vi accedono tramite un account, se un utente fa il bishero viene bloccato.

## Anonymous FTP

FTP anonimo è ciò che più si avvicina al protocollo HTTP.

Il protocollo FTP prevede infatti anche una **modalità anonima**: Anonymous FTP che consiste nel definire un account anonimo al quale si può accedere senza password. Veniva consigliato agli utenti che usavano Anonymous usare come password il proprio indirizzo di posta elettronica in modo che chi voleva poteva riconoscere.

Un esempio di uso del **download anonimo** è quello da **risorse open-source**.

Per l'upload invece era possibile utilizzare l'account anonimo solo se questo non avveniva in contemporanea al download.

L'**upload anonimo** poteva essere utilizzato dagli utenti per **segnalare dei bug agli sviluppatori**.

L'**upload e il download in modalità anonima era vietato** perché c'era il rischio che il server venisse usato per diffondere file malevoli, quindi in modalità anonima o si poteva scaricare o caricare, non entrambi contemporaneamente.

## Richieste e Risposte FTP

I comandi sono codificati in stringhe di tre o quattro caratteri mentre le **risposte** sono **codificate in numeri seguiti da una breve frase** scritta in linguaggio leggibile alle persone. Una risposta poteva essere **200 OK**:  
**OK** era leggibile dell'utente e **200** era leggibile dalla macchina.

Col **Telnet** ho la possibilità di scrivere su tastiera, caricare sul server e ricevere messaggi da esso. Aprendo la connessione sulla porta 20 posso mandare comandi al server, questo era utile agli sviluppatori per **verificare il suo funzionamento**.

Attualmente la versione FTP originale non si usa più ma si usano le versioni aggiornate, le quali cifrano le password.

Ci sono altri protocolli di tipo applicativo, andiamoli a vedere brevemente.

## SMTP (E-Mail)

**SMTP** consente l'invio dei messaggi di posta elettronica e si basa sullo stabilire una connessione in **modalità di trasferimento asincrono**, infatti possiamo **mandare messaggi anche quando il destinatario non è disponibile**.

La comunicazione tra due dispositivi è mediata dalla presenza di server nei quali vi è una **mailbox** dove vengono inseriti i messaggi in modo che il destinatario vi possa accedere e controllare la propria posta.

Se sono arrivati dei messaggi il protocollo **SMTP** si occupa del trasferimento di essi dal mittente alla mailbox del destinatario.

Ci possono essere degli altri **server intermedi** che memorizzano il messaggio prima di recapitarlo nella mailbox e servono per tollerare la non presenza del server con la **mailbox** del destinatario.

Quando quest'ultimo tornerà online andrà a pescare il messaggio memorizzato nel server mediano.

## Struttura dei Messaggi

I dati che passano sulla rete sono codificati in ASCII a 7 bit.

I messaggi sono composti da un header e un body entrambi codificati tramite ASCII.

L'header è codificato dando una serie di nomi predefiniti, per esempio: *subject*, *from*, *to*, *date*.

Il campo **to** è l'unico **obbligatorio**, gli altri sono facoltativi.

Tutto ciò viene codificato in righe di testo di massimo 72 caratteri seguiti da un a capo e dato che i diversi sistemi hanno diversi **terminatore di riga** quello dell'e-mail è: **<CR><LF>**

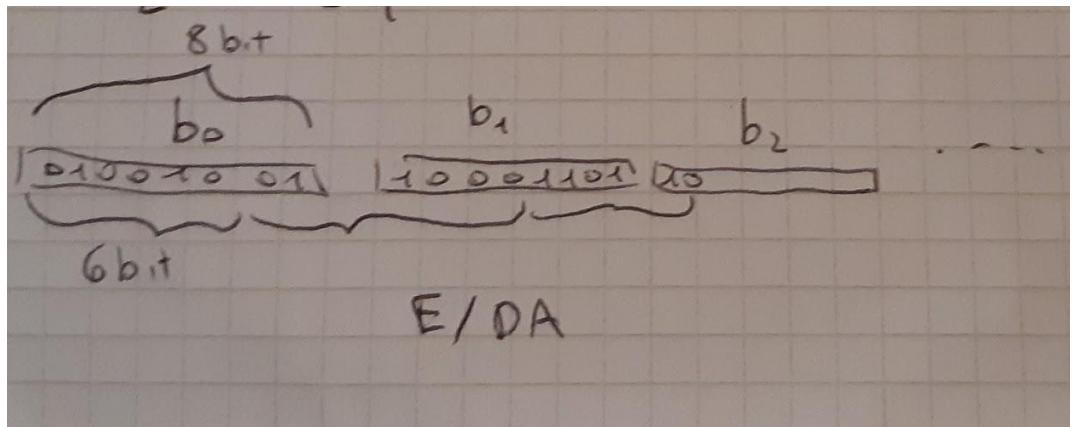
Per **terminare l'header** si lascia una riga vuota **<CR><LF><CR><LF>**

Per **terminare il body** si usa un punto e una riga vuota **<CR><LF> . <CR><LF>**

Le lettere accentate sono codificate attraverso la **codifica MIME** che non usa caratteri ascii a 7 bit e si usa anche **per mandare come allegati foto e video**.

MIME ha come base il codice [radix64](#) ovvero un alfabeto di 64 simboli che mi permette di scrivere i numeri in base 64.

I caratteri sono un sottoinsieme degli ascii per esempio: 0, 1, 2....; a, b, c....; A, B, C....; +, /....



## (8)Lezione 15 (19/10/2020)

SMTP (protocollo usato per l'invio della posta elettronica)

Il *Simple Mail Transfer Protocol* è un protocollo standard per la trasmissione di email, inizialmente proposto nella RFC 778 nel 1981.

È un protocollo a **livello applicativo**, basato sul protocollo di trasporto **TCP** e su quello di rete **IP**; il servizio viene fornito sulla **porta 25**.

L'idea del protocollo è di mandare messaggi in maniera **asincrona**, posso quindi mandare messaggi anche quando il ricevente non c'è; essa si basa su un server centrale, dove al suo interno risiede una **mailbox**, dove permangono i messaggi che un host ha ricevuto.

### Esempio di Comunicazione

La comunicazione avviene quindi in 3 passi:

- 1) Abbiamo un'applicazione chiamata MUA, che dialoga con un server MTA.

L'idea è quella che il server si prenda in carico l'invio del messaggio da parte dell'applicazione;

- 2) Se il server con la mailbox è al momento non disponibile, il messaggio può essere trasportato e memorizzato temporaneamente da un **server intermedio**; questo passaggio permette di **tollerare l'assenza del server con la mailbox**;

- 3) Una volta giunto a destinazione, il messaggio viene memorizzato sotto forma di file nella mailbox server.

## Caratteristiche del SMTP

SMTP è uno dei primi protocolli sviluppati, ha quindi un **trattamento particolare da parte del DNS**.

Il DNS contiene record chiamati **record MX** nel quale viene specificato per ciascun dominio qual è il server di posta elettronica da contattare; **attraverso un'interrogazione DNS il server SMTP conosce il destinatario**. Il destinatario viene determinato prendendo l'indirizzo del destinatario (*nome@indirizzohost*), in particolare la parte dopo la chiocciola; viene quindi fatta un'interrogazione DNS su quel dominio e ricavato qual'è il server di posta da contattare per poter raggiungere quel destinatario.

Questo protocollo viene definito come una qualità di servizio **best effort**; è una via di mezzo in termini di affidabilità: si prende cura della possibilità di errori, cerca di rimediare senza però dare la completa affidabilità. La cosa peggiore che può succedere è che cada la connessione; utilizza una quantità limitata di risorse per avere una quantità limitata di errori.

L'idea di best effort è di **mantenere in memoria il messaggio** da inviare **finché non abbiamo la conferma di ricezione da parte del destinatario**; se un server deve mandare un messaggio ad un'altro server, se la trasmissione va a buon fine ovvero il server destinatario risponde “250 ok”, allora esso si è preso carico del messaggio e il server mittente può cancellarlo.

## Codifica dei Dati

**Il protocollo supporta direttamente caratteri ASCII a 7 bit.**

Il body può contenere righe al massimo di 72 caratteri prima di andare a capo (per evitare problemi di visualizzazione su stampanti e monitor).

Codifica dei dati in UTF-8 significa che si può usare l'8 bit, attraverso questa estensione si permette **l'uso del protocollo di posta elettronica a paesi che non usano il modo di scrittura americana**.

Per quanto riguarda la possibilità di **aggiungere allegati** esiste **MIME** estensione del protocollo che permette di codificare formati di file arbitrari in modo da trasformarli in sequenze di caratteri che sono un sottoinsieme dei caratteri ASCII. Il sottoinsieme dei caratteri usati è il **radix-64** e la codifica consiste nel trasformare terne di byte successivi in quaterne rappresentate su 6 bit. L'idea è prendere 24 bit e trasformarli in 4 caratteri radix-64.

Questo perchè radix lavora 6 bit alla volta; per rendere questa codifica compatibile all'invio del messaggio, bisogna fare in modo che sia un multiplo di 3 byte; se non lo è bisogna aggiungere un **padding** (aggiungere zeri alla fine) per fare in modo che lo sia.

In SMTP è previsto anche l'uso della codifica **UUencode**: un sistema che permette di convertire file in formato binario (quali archivi compressi, disegni o programmi eseguibili) in

un formato ASCII contenente solo caratteri compresi tra 32 e 95. Questa codifica esclude i caratteri di controllo e i caratteri a 8 bit, **permettendo di trasmettere con sicurezza il file come allegato ad un messaggio e-mail.**

Tradizionalmente la codifica è eseguita da un programma chiamato *uuencode*, e la decodifica (che riporta il file nel formato originario) da *uudecode*.

## Sicurezza legata ai messaggi di posta elettronica

Facciamo un esempio esplicativo:

- A vuole mandare un messaggio a B:
  - A si collega a un server di posta elettronica, specifica il messaggio usando il protocollo SMTP e poco alla volta il messaggio viene recapitato a B.
- a questo punto B per poter leggere il messaggio deve aprire una sessione di lavoro sulla sua macchina, in modo da poter accedere alla **mailbox**, che è un **file che contiene tutti i messaggi ricevuti**;
  - per poter accedere ai messaggi B si deve **autenticare** specificando username e password **garantendo** in questo modo anche la **privacy** di A, in quanto i messaggi possono essere letti solo da lui.

## Esempio di Comunicazione, nel Dettaglio

Andando più nel dettaglio di un esempio di comunicazione:

- "A" tramite il suo MUA si connette a un server SMTP, aprendo una connessione nella **porta 25**.
- Una volta stabilita la connessione la prima cosa che fa il server è quella di mandare un messaggio di tipo **220 OK** con il quale **comunica** qual'è il **proprio indirizzo e che tipo di server è**.
- Il client dopo aver ricevuto la risposta positiva dal server è previsto che si presenti anche lui, mandando il comando **HELO** e **passando il suo indirizzo**.
- Il server risponde con **250** e reinviando l'indirizzo che ha dato il client.

A questo punto il client può mandare una serie di **comandi** che sono quelli **per spedire un messaggio** ovvero:

- indirizzo destinatario: *bob@unige.it*.
- **RCPT TO** che specifica al client deve mettere anche il proprio indirizzo di posta elettronica con il comando **MAIL FROM: alice@unito.it**
- Server riceve e risponde con 250 OK

L'indirizzo del mittente serve al protocollo SMTP per poter inviare eventualmente notifiche in caso di problemi; una volta acquisito, il protocollo ha gli indirizzi che gli servono e può procedere allo scambio di messaggi.

- Con il comando **DATA** il client richiede al server il permesso di poter mandare il messaggio.
- Il server dopo aver ricevuto il RCPT TO, il MAIL FROM e il DATA manda una risposta **354**; dopo di essa il server si aspetta di ricevere i byte fino alla fine del messaggio, e lo comunica con la stringa: *end data with <CR><LF>.<CR><LF>*
- Dopo aver ricevuto 354 *end data with*, inizia il vero e proprio invio del messaggio; il client comincia a mandare sequenza di byte codice ASCII 7 bit, iniziando con l'Header, costituito da due campi:
  - **from:** *alice@unito.it* (scrivere qui qualcosa di diverso significa fare spoofing dell'utente)
  - **to:** *bob@unige.it* (scrivere qui qualcosa di diverso aggiunge un hop alla comunicazione perché viene)

**Chi riceve il messaggio non ha modo di sapere se il *from* è corretto:** al di là della problematica di mancanza di garanzia di autenticità del mittente, questa debolezza del protocollo SMTP ha avuto un effetto devastante per quanto riguarda le problematiche riguardo lo **spoofing** e lo **spam**, ovvero lo sfruttare le caratteristiche di posta elettronica per ottenere vantaggi a scapito degli utenti.

## Spam

**Per un messaggio singolo pagano sia il mittente che i destinatario.**

**Se mando un messaggio a 10mila destinatari diversi pago come se avessi mandato un messaggio solo**, la moltiplicazione per 10mila la pagano i 10mila destinatari.

Il servizio di posta elettronica, pur essendo gratuito, è quindi pagato anche da chi riceve; normalmente ricevere un messaggio di posta elettronica non ha un grosso costo ma se si diffonde l'abitudine di mandare 10mila di messaggi per un solo mittente questo si traduce in uno **spreco enorme di risorse**.

Per evitare lo spam l'idea è stata di aggiungere dei **filtri**; prima di andare a inserire il messaggio nella mailbox del destinatario ci si chiede quindi se quello può essere spam oppure no, tramite lo sviluppo di vari algoritmi.

## Open Relay

**Un server senza filtri**, ovvero che accetta di mandare messaggi senza accettare la loro provenienza **viene chiamato open relay**;

Un server SMTP open relay **permette quindi a chiunque sulla rete di inviare e-mail attraverso di esso**; questo tipo di server è da evitare, quindi si aggiunge un meccanismo di autenticazione da parte del mittente e di tracciamento degli indirizzi IP, tramite estensioni del protocollo; in questo modo una qualsiasi azione malevola e di spam può essere tracciata.

Viene implementato un comando **EHLO** che mette in gioco un **meccanismo di autenticazione** che chiede all'utente di identificarsi usando username e password.

Nel caso posso anche inserire una **copia dati della procedura preliminare** insieme al messaggio: in essa non uso gli indirizzi mail ma gli indirizzi IP, in modo che non possano essere contraffatti.

L'utente può finire *onload* ed essere registrato e mantenuto nel tempo; in questo modo se egli manda messaggi attraverso spam si può agire su di esso disabilitandolo.

Al giorno d'oggi, in ogni caso, non ci potremmo connettere ad un server a caso ma dovremmo comunque connetterci ad un server SMTP.

## Protocolli per la ricezione dei messaggi

L'idea è quella di poter consentire a "B" di poter accedere alla sua mailbox senza dover effettuare login sul server di posta nella quale è stata depositata.

Se B usa una macchina diversa può connettersi via rete e ha bisogno di un protocollo che gli permetta di scambiare informazioni senza ricorrere a una shell sulla macchina.

I due protocolli introdotti sono il protocollo **POP** e **IMAP**; le versioni in uso sono la versione POP3 e IMAP4.

### POP

Pop è un protocollo **semplice** e **offline**.

Prendiamo l'esempio precedente:

POP effettua una copia della mailbox sulla macchina di "B":

- POP3 si connette alla macchina, va a vedere i messaggi nella mailbox e li copia sulla macchina locale di "B", qui ci possono essere due modi di funzionamento:
  - 1) Copia dalla mailbox alla macchina: sistema **più affidabile, ma spreco di spazio** perchè ho bisogno di memoria sul server e sul local.
  - 2) Sposta dalla mailbox alla macchina (effettua una copia e la cancella): meno spazio occupato, ma **meno efficienza e rapidità**.

In attacchi di tipo spam se non si provvede a una cancellazione periodica si esaurisce lo spazio e non si possono più ricevere mail.

Un altro svantaggio di POP3 sono i problemi legati al poter accedere con più dispositivi sulla stessa mailbox. Se io volessi accedere alla mia posta elettronica usando due dispositivi diversi, non posso fare l'operazione di cancellazione, sono costretto a mantenere l'originale sul server, altrimenti non posso accedere alle stesse informazioni con l'altro dispositivo; devo quindi sempre tenere traccia di quali messaggi sono sui vari dispositivi.

**Devo quindi essere sempre sincronizzato tra server e macchina** e non posso implementare un meccanismo di caching (come nell'IMAP).

Questo protocollo è usato quando non posso connettermi ad un server; il suo impiego principale è nei dispositivi **mobile**.

## IMAP

IMAP è un protocollo **sicuro e online**.

Esso **da la possibilità a più dispositivi HW di accedere alla stessa mailbox**; IMAP infatti mantiene la mailbox sul server ed effettua copie parziali usando una **tecnica di caching** sulle singole macchine.

Il protocollo permette di interrogare il server chiedendo quali sono i messaggi; dopo di chè se voglio vedere un messaggio, esso viene copiato e mantenuto in locale nella macchina ma l'originale viene mantenuto sul server.

Quando si esaurisce lo spazio nella macchina, usando una tecnica di caching, viene eliminato il messaggio letto con minor frequenza per lasciare spazio a un nuovo messaggio; questo **risolve in modo efficiente il problema di gestione dello spazio** e della comunicazione su client e server.

**IMAP risulta quindi molto più flessibile di POP.**

Nel caso in cui non ho la possibilità di connettermi al server però POP risulta vantaggioso rispetto a IMAP.

## Webmail

La versione di accesso alla posta elettronica attualmente in uso è chiamata webmail; essa è simile a IMAP ma **usa l'HTTP**.

Il protocollo HTTP viene sfruttato per accedere via rete a un server (HTTP) che a sua volta si connette al server di posta elettronica per far vedere tramite richiesta (HTTP) il contenuto della mailbox presente su di esso.

**La posta rimane quindi all'interno del server della mailbox.**

# (9)Lezione 17 (22/10/2020)

## Laboratorio UDP Ping

Dobbiamo contattare il server in modalità UDP:

Il server deve aprire una socket di tipo UDP, fare la *bind* sulla porta e rispondere al client con il numero di porta da utilizzare. Anche il client deve aprire un socket di tipo UDP e a questo punto server e client potranno cominciare a scambiarsi dati.

La difficoltà risiede nel fatto che nel server vengono utilizzate delle porte effimere (decise da IANA) che sono più alte di 55 mila. I firewall dei server unicamente tagliano fuori gli

UDP ma grazie all'intervento del Mastro Don Chiola abbiamo a disposizione un server che ci permette tale connessione.

Ci serviranno due socket diversi: uno di tipo **stream** e uno di tipo **datagram**. In modalità datagram sia il ping che il pong server devono essere preparati alla possibilità che alcuni messaggi vengano persi e quindi implementare un **timeout**, *application level time out*, che conti il tempo che trascorre nell'attesa della ricezione del messaggio.

Se il messaggio non è arrivato si aspetta un breve lasso di tempo e si ripete il ciclo fino a quando non arriva. Quindi in ogni ciclo continuerò a chiedermi se è arrivato il messaggio e misurerò l'istante di tempo in cui termina la *receive*. Se  $t4 - t2$  è minore del timeout ripeto il ciclo, se è maggiore vuol dire che il messaggio è andato perso.

La modalità bloccante / non bloccante vale sia per la ricezione che per la trasmissione. Dal momento che alcune system call sono bloccanti (non possono essere eseguite altre operazioni in contemporanea con loro) vengono messe in attesa fino a quando non serviranno nuovamente. I socket che andremo ad utilizzare non saranno bloccanti quindi se non riusciranno a ricevere un messaggio restituiranno l'errore -1. Per tale motivi ci converrà utilizzare la variabile *errno* che ci informerà in quale errore ci siamo imbattuti.

**Ewoldblock** e **again** sono delle costanti che ci dicono a che punto si blocca la nostra funzione.

*send\_to(cher \*buffer, int len)*

- len = dimensione del buffer
- buffer = indirizzo del buffer allocato in memoria. Quando ricevo un messaggio lo copio nel buffer di ricezione, quando devo mandarlo lo copio in quello di trasmissione.

Quando la *send\_to* è terminata posso modificare l'area di memoria del buffer . Se non c'è abbastanza spazio per memorizzare il messaggio ricevuto, nella modalità normale si blocca tutto, in quella modificata si termina con un errore. Questo errore prima o poi se ne andrà, basta ripetere il ciclo

## (10)Lezione 19 (26/10/20)

### Controlli nel Protocollo TCP

TCP ha due funzionalità di controllo per assicurare la sua affidabilità:

- il **controllo di flusso**;
- il **controllo di congestione**.

Controllo di flusso:

Il controllo di flusso **serve per ridurre le perdite dei messaggi** agendo in base all'RX buffer dell'host ricevente ed evitando il suo overflow.

Prendendo in considerazione una rete multi-hop abbiamo: il mittente host, i router e il destinatario host.

Ogni router riceve il messaggio, va a vedere nell'header l'indirizzo di destinazione, calcola il percorso e inoltra il messaggio al router successivo.

Come sappiamo i messaggi possono venire persi per errori di trasmissione sul canale. Quando il messaggio viene ricevuto si verifica la sua integrità e se non è integro viene scartato.

Attualmente la probabilità di errore è bassa grazie alle connessioni odierne e la causa più comune di perdita di pacchetti deriva dalla **mancanza di spazio sul buffer di ricezione**.

Con il controllo di flusso si evita che il messaggio sia troppo grande rispetto al buffer di ricezione del destinatario.

Bisogna quindi evitare l'**overflow** del buffer di ricezione **rallentando l'invio del messaggio dal mittente, per lasciare più tempo al destinatario per svuotare il buffer**.

Tutto ciò è possibile grazie alla struttura dell'header TCP.

Header TCP e Receive Window

Source port	Destination port		
seq # (4 byte)			
ack # (4 byte)			
Lunghezza intestazione (2 byte)	Bit non utilizzati	Flag (2 Byte)	Finestra di ricezione (16 Byte)

Ricapitolando, le fasi principali in una connessione TCP sono:

- Prima sia apre la connessione attraverso 3-way handshake;
- con il primo messaggio **syn** la finestra di ricezione ha come valore la dimensione del buffer di ricezione;
- quando il server risponde mette nella *receive window* lo spazio disponibile nel buffer.
- Server e client prima di mandare i messaggi **allocano spazio nei buffer** e quando si invia un datagramma si tiene conto della dimensione dei buffer del ricevente.
- **Se il destinatario non ha abbastanza spazio libero per memorizzare il datagramma il mandante manda solo la parte di messaggio che può essere salvata** e quando riceverà la comunicazione che si è liberato dello spazio manderà il resto del messaggio.  
Lo spazio va liberato con la syscall *recv()*.
- I messaggi rimangono nel buffer per un lasso di tempo preciso e poi vengono cancellati per fare spazio ai prossimi.

### Deadlock

Però c'è un problema: quando la comunicazione è unidirezionale (perché uno dei due canali è stato chiuso dopo *fin*) e A ha mandato la parte del suo datagramma a B occupando tutto il

buffer, in seguito B non potrà rispondere ad A per comunicargli che ha svuotato il buffer e che quindi è pronto per ricevere il resto del datagramma.

Questa situazione è detta di **deadlock**.

Per ovviare a ciò vi è un **timeout** allo scadere del quale A prova a mandare il resto del messaggio; se B è riuscito a memorizzarlo manderà un **ack** indietro.

Quando la *receive window* è 0 il mittente manda **un solo byte** per verificare se il destinatario ha liberato dello spazio, in modo da minimizzare il più possibile lo spreco di tempo.

### Controllo di congestione TCP

Il controllo di flusso **serve per ridurre le perdite dei messaggi** agendo in base all'RX buffer dei **router intermedi** ed evitando il loro overflow.

Esso **serve quindi per evitare di perdere messaggi se non vi è abbastanza spazio libero sui loro buffer di ricezione**.

Ogni router ha un buffer di ricezione e uno di trasmissione e se quello di ricezione si riempie quel router non potrà più inoltrare i datagrammi.

Nel protocollo IP in passato non era implementato un controllo di flusso infatti se si perdeva un datagramma veniva mandato indietro un messaggio **ICMP**, che notificava il mittente della perdita.

Per colmare questo problema è stato creato il controllo di congestione a livello di trasporto.

### Implementazione

Viene definita una **congestion window** nell'header che **indica lo spazio libero minimo del buffer di ricezione dei router** tra quelli che dovrà attraversare il datagramma per arrivare a destinazione.

Per stimare questo valore vi è un algoritmo che inizia con una fase di **slow start**: si parte inviando messaggi di piccole dimensioni e via a via che vado avanti le aumento.

Se A riceve un **ack** vorrà dire che la **dimensione del messaggio che ha mandato è minore di quella dello spazio disponibile sul buffer di ricezione del router con buffer più piccolo**.

Si procede così aumentando esponenzialmente (1, 2, 4, 8...) la dimensione del messaggio fino a quando non si è mandato tutto il datagramma oppure non si ha più ricevuto l'ack da parte di B.

Una volta che **sappiamo il peso massimo sostenibile conosciamo anche la *congestion window***.

Ad esempio: se non ricevessimo l'ack dopo aver mandato un datagramma di dimensione = 8 sapremmo che la congestion window è compresa tra 4 e 8;

a questo punto possiamo andare avanti un byte alla volta **ripartendo da capo**, quindi 1,2,3,4... per andare a trovare preciso la *congestion window*.

Bisogna tenere conto del fatto di due difetti:

- **non essendo gli unici ad utilizzare la rete potrebbe essere che un router ha poco spazio sul proprio buffer perchè sta venendo utilizzato da qualcun altro.**
- gli stessi router possono essere usati contemporaneamente da più connessioni (anche UDP): i buffer di ricezione possono essere riempiti anche da altre connessioni.

Il protocollo TCP ha sia il controllo di flusso che di congestione e **per determinare la dimensione massima del messaggio si prende il valore minimo tra la *congestion* e la *receive window*.**

## MTU e Frammentazione

La **MTU** (maximum transfer unit) è un vincolo sulla dimensione massima dei datagrammi.

Essa varia in base al protocollo: ad esempio, Ethernet ha una MTU di 1500 byte, IP di  $2^{16}$  byte.

Quindi nel protocollo IP la **dimensione complessiva massima di un datagramma è di  $2^{16}$  byte, mentre la dimensione massima di ogni singolo pacchetto inviato è di 1500 byte.**

Se A volesse mandare un datagramma di 15000 byte a B **non** potrebbe mandare tutto il datagramma assieme poiché quello supera la soglia massima di peso per singolo messaggio (1500 byte): il messaggio verrà quindi diviso in 10 parti, attraverso una **frammentazione**, che verranno poi mandate singolarmente.

**B manderà poi un singolo ack alla ricezione del datagramma completo** e non uno per ogni pacchetto ricevuto; questo comporta che se anche solo una delle 10 parti non viene recapitata B non manderà indietro l'ack.

## (11)Lezione 21 (29/10/2020)

### Protocollo IP

Lo scopo principale di IP è il **routing**, ovvero l'instradamento dei pacchetti sulla rete attraverso i router intermedi; lo scopo dei router è quindi determinare il percorso per spostare i datagrammi da una macchina all'altra.

In un collegamento tra due Host ci possono essere molti instradamenti possibili.

A grandi linee abbiamo: il mittente che si collega al primo router, il destinatario che si collega al router di frontiera e lo scopo è quello di trovare una serie di **router intermedi** che possono far da ponte per poter spostare un datagramma da mittente a destinatario.

I router sono dispositivi fisici simili a un calcolatore, la differenza principale da un computer è la possibilità di avere **tante interfacce di comunicazione**, avendo così una possibilità di scegliere dove instradare il nostro datagramma.

## Datagrammi IP e Architettura dei Router

Nell'intestazione abbiamo, tra le altre informazioni, l'**indirizzo del destinatario**.

Il router riceve il datagramma e lo immagazzina in un buffer di ricezione (RX); a questo punto va a vedere le intestazioni del datagramma IP, trova l'indirizzo del destinatario e in base ad esso **sceglie su quali canale inoltrare il messaggio**.

### Router Connesso a più Router

Supponiamo di avere un router che ha la possibilità di connettersi con altri 8 router, **ci sono 8 canali che permettono di ricevere e inviare datagrammi in direzioni diverse**.

**Ad ogni canale fisico corrisponde un buffer di ricezione e uno di trasmissione.**

Vi è però un vincolo di velocità di funzionamento: questa **operazione di routing, deve essere effettuata in un tempo ridotto** per evitare che si accumulino ed evitare che si perdano datagrammi.

Tra i due buffer, di trasmissione e di ricezione presenti nel router, ci deve essere una **tabella che associa un IP con un numero di porta in uscita**; dopo di che si può fare una ricerca associativa su questa tabella, per ottenere come risultato il numero di porta su cui instradare le informazioni.

Questa tabella è detta di inoltro (**forwarding table**) e si interpone tra i due buffer per associare IP e numero di porta.

Il problema che sorge è che gli indirizzi sono rispettivamente di 32 bit per IPv4 e 128 bit per IPv6; **è quindi impossibile creare una tabella di inoltro così grande**.

La soluzione è prendere l'IPv4 composto da 32 bit ma andare a vedere soltanto una piccola parte di bit: per esempio, i primi 8 bit; in questo modo avrà una tabella associativa di  $2^8$  celle.

In caso di indirizzi molto vicini tra loro è meglio avere una grossa quantità di bit per fare il controllo associativo.

## Indirizzo IP

Internet è una **rete di reti**, una parte dell'indirizzo deve indicare una rete locale a cui il router fa riferimento; in essa ci sono tanti host ognuno con un IP diverso, quindi la seconda parte deve andare a specificare l'host in quella sottorete.

Gli indirizzi IP sono quindi divisi in due parti :

- **Network:** corrisponde **agli indirizzi di rete** (parte utile da esaminare da parte del router)
- **Host:** corrisponde **agli indirizzi dell'host di quella rete** (ultimo router tiene conto del numero di host per poter inviare messaggio)

Network	Host
Netmask	

Esiste inoltre una **Netmask**, che è un numero delle stesse dimensioni dell'indirizzo IP e definisce la **dimensione della sottorete**;

E' costituita da un insieme di:

- Uni: che identificano la sottorete.
- Zeri: che identificano la parte destinata agli host.

ex. 111111100000000000000000000000000000000      *8 bit per la subnet, 24 per gli host*

All'interno della tabella possono essere memorizzate **porzioni di indirizzi di quantità arbitraria**.

**Più piccolo è il prefisso** (ovvero i bit dedicati al network) **peggiore è la qualità di instradamento**.

**All'aumentare del prefisso**, la qualità di instradamento migliora, ma **le tabelle di forwarding devo essere più grandi**.

Bisogna trovare il **giusto equilibrio**; se siamo vicini all'host di destinazione scriviamo tanti bit per l'indirizzo; se siamo molto lontani possiamo utilizzare un indirizzo più corto.

Saranno rari i casi in cui i 32 bit saranno significativi per ottenere l'instradamento corretto; ci basiamo quindi sulle reti locali per avere tabelle ridotte in modo da essere gestite in modo efficiente.

## Tabella di Inoltro (Forwarding Table)

Esse servono per associare indirizzo IP e numero di porta; la loro costruzione prevede il coinvolgimento di **algoritmi di routing** e di un protocollo ausiliario ad IP detto **ICMP**.

Bisogna avere inoltre un buon **supporto hardware**: una memoria associativa che supporti celle di lunghezza variabile, bus, reti di interconnessione; questi servono per creare la tabella e collegarla ai buffer dei router.

Gli algoritmi di instradamento servono per scegliere i percorsi dei pacchetti; essi però non vengono eseguiti quando arriva il pacchetto, sarebbe una perdita di tempo enorme; questo algoritmo funziona **offline**: prima che arrivi il pacchetto, l'algoritmo calcola dove viene smistato; ciò permette una maggiore velocità.

Questi algoritmi devono essere **eseguiti continuamente**, in modo da continuare a cercare strade più vantaggiose; da questo deriva il fatto che se noi mandiamo una serie di datagrammi allo stesso destinatario non è detto che seguano tutti la stessa strada.

Il router oltre a smistare i pacchetti, comunica con altri router per poter **scambiare informazioni utili per il calcolo di instradamenti**; la comunicazione tra router avviene grazie al **protocollo ICMP**.

## Protocollo ICMP

Questo protocollo serve ai router per poter **dialogare tra di loro e per creare tabelle di routing**; l'IP serve poi agli utenti per mandare datagrammi da mittente a destinatario usando la tabella di forwarding.

ICMP è un protocollo ausiliario: segnala errori, debugga la rete e **usa, per poter inviare messaggi a lunga distanza, gli IP e l'instradamento IP**.

IP e ICMP sono stati sviluppati assieme quasi come se fossero un protocollo unico: **ICMP non può fare a meno di IP e viceversa**.

## Algoritmo di routing

Lo scopo è quello di arrivare a costruire delle **associazioni tra**:

- **Prefisso Indirizzo IP (Network)**
- **Numero di Porta**

**La qualità dell'instradamento dipende dal numero di bit che costituiscono il prefisso dell'indirizzo** di cui teniamo conto: se abbiamo dei prefissi composti da pochi bit la qualità sarà scarsa, man mano che andiamo ad aumentare i bit del prefisso, otteniamo una precisione sempre maggiore di instradamento.

Se ho 2 indirizzi, uno con prefisso da 4 bit e l'altro da 24 bit, verrà data la **priorità** al prefisso da 24 bit perché ci da una maggiore precisione.

Un router, contattando altri router, può capire quali sono gli instradamenti migliori attraverso due approcci diversi:

- 1) **Link state:** Approccio di tipo **Globale**:

**ogni router comunica con tutti gli altri router della rete** e dice quali sono le sue connessioni; viene quindi creato un grafo dove i nodi sono tutti i dispositivi connessi alla rete; questo permette a tutti i dispositivi di avere informazioni precise su tutti gli altri;

A questo punto si può sfruttare l'**algoritmo di Dijkstra** per cercare il percorso minimo tra due nodi di un grafo.

## 2) **Distance Vector:** Approccio di tipo **Locale**:

**ogni router comunica coi suoi vicini:** se un router è connesso con 3 vicini dice a questi 3 tutte le sue informazioni. La stessa cosa la fanno gli altri 3 router distribuendo le loro informazioni con chi sono connessi: il messaggio si propaga e dopo un periodo di tempo abbastanza breve tutti riescono ad avere delle stime abbastanza buone sulla rete.

Ogni router misura la distanza che lo separa dai router adiacenti; a partire da quella crea una **tabella** dove associa **ogni destinazione conosciuta** con **la stima della distanza e il primo passo da eseguire verso essa**.

In questo modo riusciamo rapidamente a trovare una serie di indirizzi di macchine raggiungibili in pochi hop; ho quindi delle informazioni **approssimative** su quale porta di uscita è meglio mandare i messaggi.

Sulle reti di piccole dimensioni, un approccio globale (Link State) risulta essere più vantaggioso; esso risulta essere **molto preciso ma pesante** in termini di tempo.

Sulle reti di grande dimensioni ha più senso usare un approccio locale (Distance Vector); esso è **approssimativo ma molto veloce**.

## (12)Lezione 23 (02/11/20)

### Laboratorio:

A questo punto dovremmo avere sia il client TCP che UDP. Quando il client ping si connette con il server pong c'è una fase di convenevoli, in cui si scambiano il numero di ripetizioni e la dimensione del file, e poi si misura il round trip time.

$$RTT = (t4 - t1) - (t3 - t2)$$

Per avere una precisione migliore conviene fare le misurazioni più volte. I risultati che otteniamo sono un valore medio ( $\frac{\text{somma}}{\text{ripetizioni}}$ ) e un valore mediano (ottenuto ordinando i

valori in modo crescente e prendendo quello che sta in mezzo).

Questi valori non sono troppo significativi poiché la connessione potrebbe essere in un momento che è sovraccarica e quindi i valori potrebbero essere sballati.

Il modello *banda-latenza* da un'indicazione di massima, ad esempio il ritardo può essere approssimato con:

$$D(N) = L0 + \frac{N}{B}$$

D = ritardo (metà del RTT)

(N) = N min o N max (numero di byte)

L<sub>0</sub> = costante ottenuta misurando il comportamento della rete con messaggi piccoli

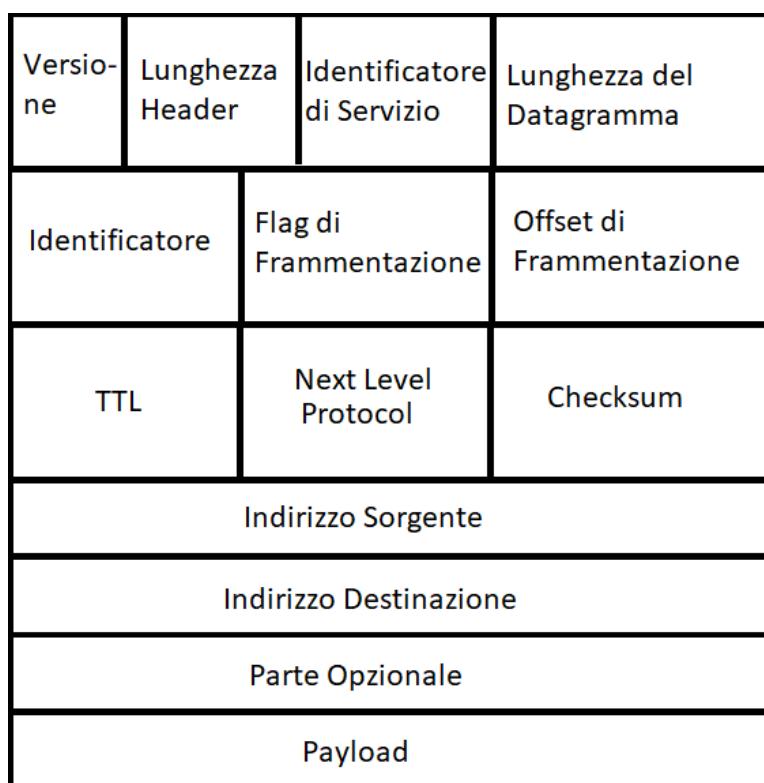
B = costante ottenuta misurando il comportamento della rete con messaggi lunghi

Con l'operazione GNUPLOT calcolo i grafici prendendo in ingresso dei valori contenuti in altri file. conviene utilizzare una scala di tipo algoritmico in modo da sfruttare al meglio lo spazio evidenziando quello che succede con valori piccoli e l'andamento asintotico con i valori grandi.

Per fare i calcoli conviene usare la calcolatrice della bash "BC" dove si può scegliere la precisione.

Grazie al MAKEFILE possiamo riprendere la misurazione da dove avevamo interrotto ma non controlla se i file contengono degli errori.

## Header IPv4



L'header IPv4 ha dimensioni variabili, identificate dal campo **Lunghezza dell'Header**.

L'**identificatore di servizio** è composto da una serie di **bit che identificano il tipo di datagramma** (non è usato molto spesso). Questi bit servivano all'host mittente per specificare il modo e in particolare la precedenza con cui l'host ricevente doveva trattare il datagramma. Ad esempio un host poteva scegliere una bassa latenza, mentre un altro preferire un'alta affidabilità. Nella pratica questo uso di questo campo non ha preso largamente piede.

Dopo molte sperimentazioni e ricerche, recentemente questi 8 bit sono stati ridefiniti ed hanno funzioni necessarie per le nuove tecnologie basate sullo streaming dei dati in tempo reale.

Il **Next Level Protocol** indica se utilizziamo il protocollo **TCP o UDP**.

Il **Checksum** è un valore che viene creato nel momento in cui il mittente manda il datagramma e cambia ad ogni hop, anche se rimane sempre calcolabile dai router intermedi tramite un algoritmo che si basa sugli altri campi dell'header; se non corrisponde tra mittente e destinazione vuol dire che probabilmente c'è stato qualche errore.

Questo numero è costituito da 16 bit, la probabilità quindi di non individuare un possibile errore è:

$$\frac{1}{2^{16}} \text{ probabilità relativamente alta per l'informatica}$$

Nel protocollo Ethernet viene quindi inserito un altro controllo chiamato **CRC32**, che funziona in modo analogo ma usa 32 bit; la probabilità di non rilevare l'errore è quindi di:

$$\frac{1}{2^{32}} \text{ probabilità molto bassa}$$

L'**indirizzo sorgente** serve per inoltrare al mittente, nel caso ci fossero, i messaggi di errore mentre l'**indirizzo di destinazione** serve chiaramente ai router per inoltrare il messaggio per farlo arrivare al destinatario.

In IPv4 i bit meno significativi dell'indirizzo destinazione indicano l'indirizzo della sottorete mentre quelli più significativi indicano il numero di host sotto quella rete.

Il **TTL** serve per evitare che errori nella trasmissione facciano girare messaggi all'infinito; esso contiene un numero che indica i salti massimi per cui il datagramma rimane memorizzato nel router prima di essere cancellato; il campo riservato al TTL è di 8 bit, quindi un datagramma può fare al massimo dagli 0 ai 255 salti a seconda di come viene impostato il TTL.

Dopo aver cancellato il datagramma viene mandato al mittente un messaggio di errore.

La **lunghezza del datagramma** è:

- al minimo 20 byte (la lunghezza minima dell'header)
- al massimo  $2^{16}$  byte.

Il datagramma potrebbe quindi essere maggiore alla MTU (1500 byte): in questo caso si ricorre alla **frammentazione**.

Per andare ad analizzare gli altri campi dobbiamo parlare proprio della **frammentazione**;

## Frammentazione

Il processo consiste, ad esempio, nel ricevere un datagramma in ingresso e mandarne due in uscita; chi riceve deve necessariamente **riassemblare** i frammenti ricevuti.

Per ogni frammento vi è un **identificatore**, che serve per riconoscere i pezzi in cui è stato diviso il datagramma e rimetterli insieme.

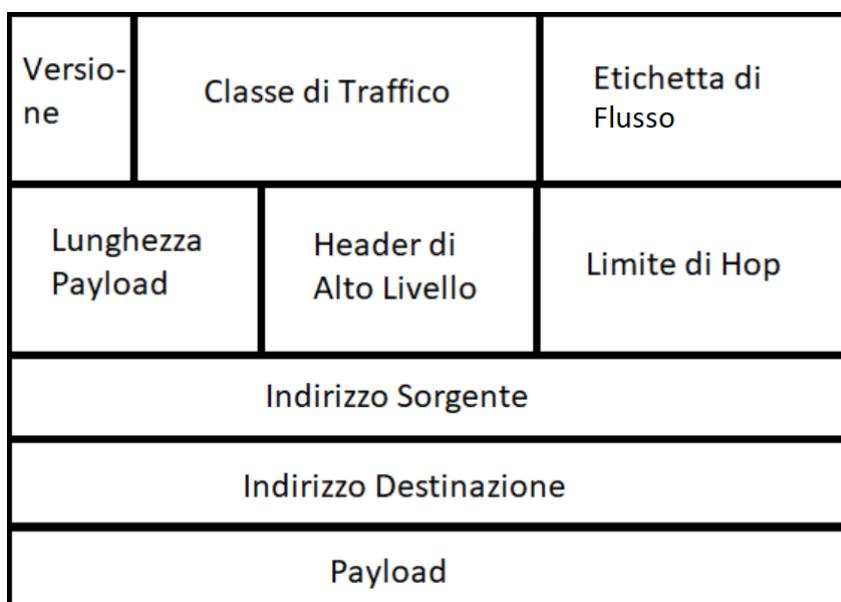
L'ordine con cui dovrà riassemblare i pezzi è specificato dall'**offset di frammentazione**: il primo frammento ha offset 0, il secondo 1 e così via, l'ultimo ha offset 1499 (la MTU-1).

Se il **flag frammentazione** è uguale a 1 vuol dire che quel datagramma è stato frammentato; i frammenti hanno dimensione minima variabile che però deve essere un multiplo di 8 byte.

La frammentazione e soprattutto il riassemblaggio hanno **costo molto alto** perché bisogna mantenere i frammenti di messaggi sul buffer RX finché non arriva il messaggio completo e ovviamente finchè un router non ha terminato tale operazione non può inoltrare il messaggio al router successivo.

## Lezione 25 (05/11/2020)

### Header IPv6



La versione è ovviamente la 6, a questo campo sono riservati 4 bit: **0110**.

La **classe di traffico** è simile a l'identificatore di servizio dell'header IPv4. Permette di gestire le code by priority assegnando ad ogni pacchetto una classe di priorità rispetto ad altri pacchetti provenienti dalla stessa sorgente. Viene usata nel controllo della congestione.

L'**etichetta di flusso** indica i datagrammi dello stesso flusso.

L'header IPv6 **ha dimensioni fisse (40 byte)**, mentre possono variare quelle del payload, identificate dal campo Lunghezza Payload.

L'**header di alto livello** può contenere sotto header encapsulati; questo campo include anche option di IPv4; questo sopperisce anche al fatto che un header abbia lunghezza fissa e non variabile.

In IPv6 il TTL è stato sostituito dal **hop limit**, un valore che **viene decrementato ad ogni hop** e una volta che arriva a 0 cancella il messaggio.

L'indirizzo Sorgente e quello di Destinazione sono di 128 bit; **sono talmente grandi che con datagrammi IPv6 non necessito di NAT e Port Forwarding**.

### Altri Dettagli

La frammentazione non è più gestita dai router ma dal mittente, quindi vengono eliminati tutti i campi ad essa riguardanti.

Il checksum è eliminato perché il controllo di integrità è gestito dal livello datalink attraverso il CRC-32, che controlla anche il payload del messaggio (oltre che l'header) e con maggiore precisione.

**Ci possono essere piu header in un datagramma:** i router lavorano sul primo header IPv6 scalando HOP limits; una volta arrivati al receiver vengono elaborati anche gli altri header che sono quindi di tipo **end-to-end**: i router non guardano gli header interni e in questo modo si aumenta la privacy.

Se l'header di alto livello contiene:

- IPv6, vuol dire che nel suo payload vi è un altro header;
- TCP o UDP, vuol dire che nel suo payload vi è il messaggio da trasportare e viene indicato attraverso quale protocollo.

In questo momento della storia dell'informatica siamo in un periodo di transizione da IPv4 a IPv6, quindi:

- alcune macchine supportano IPv4,
- alcune macchine supportano IPv6.

Poniamo il caso che i due host che comunicano siano IPv6 ma un router intermedio sia IPv4: se un router supporta solo IPv4 e riceve un datagramma IPv6, lo **incapsula** in un datagramma IPv4, mettendolo nel suo payload; a questo punto, se il ricevente supporta IPv6 lo "scarta", rimuovendo l'header IPv4; questo processo è detto **tunneling**.

Se invece, uno dei due Host è IPv4, l'altro si deve adattare.

Per passare da IPv4 a IPv6 basta aggiungere una serie di zero davanti all'indirizzo fino al completamento di esso, un'abbreviazione di tale operazione è l'inserimento di una coppia di ":" prima dell'indirizzo IPv4.

esempio: 192.168.1.22 -----> ::192.168.1.22

In questo modo IPv6 risulta **retrocompatibile** con IPv4, in un processo [IPv4-mapped address](#).

## Indirizzi IP

Ogni header IP abbiamo visto che contiene:

- indirizzo sorgente: identifica il mittente,
  - indirizzo destinazione: serve per instradare il messaggio.

Gli indirizzi IP sono divisi in:

Subnet	Host
--------	------

Nei **vecchi protocolli** Subnet e Host erano divise in **classi**:

Classe A	1 byte per la Subnet 3 byte per l'Host
Classe B	2 byte per la Subnet 2 byte per l'Host
Classe C	3 byte per la Subnet 1 byte per l'Host

Nei nuovi protocolli viene utilizzata la **netmask**, che ha gli stessi bit dell'indirizzo e:

- la parte composta da **uni** corrisponde e identifica la **subnet**,
  - la parte composta da **zero** corrisponde e identifica l'**host**.

## Indirizzi Privati

Il protocollo IP, specialmente nella versione 4, non poteva supportare l'assegnazione di indirizzi IP per tutti gli host delle reti private.

Esistono quindi degli **indirizzi privati**, utilizzabili in tutte le LAN, che poi attraverso meccanismi di **NAT e Port Forwarding** vengono convertiti per non creare conflitti sulla rete.

Gli indirizzi privati sono usati nelle reti locali private.

Essi si dividono in 3 classi:

Classe A	10.x.x.x (255.0.0.0)
Classe B	172.16.x.x (255.240.0.0)
Classe C	192.168.x.x (255.255.0.0)

Questi indirizzi servono per gestire le reti locali e gli utenti possono accedervi senza richiedere il permesso al provider (IANA).

## NAT

**Gli indirizzi privati non sono instradabili dai router**, se un router si vede arrivare un indirizzo che inizia con 10 (indirizzo di classe A) o con 192.168 (indirizzo di classe B) **non lo instrada** all'esterno; questo perché in giro per il mondo ci **possono essere device con gli stessi indirizzi privati** sotto diverse reti locali.

Per mandare messaggi da reti locali a altre reti locali si utilizza il **NAT**, una tecnica che **traduce indirizzi privati** (quelli che abbiamo nelle reti locali) **in pubblici** prima di essere instradati nella rete.

Il dispositivo che esegue il NAT ha due interfacce di rete quindi ci permette di comunicare dall'interno della subnet sione ambo i lati.

**Il dispositivo che effettua la conversione è un router (detto di frontiera) con un indirizzo detto Default Gateway e appunto trasforma un indirizzo IP privato in uno pubblico.**

Esempio:

La rete utilizza un indirizzo privato 10.20.30.40 e uno pubblico 12.12.12.12

Se A, all'interno della rete locale, volesse comunicare con B che si trova all'esterno dovrà mandare un datagramma che avrà come indirizzo sorgente l'indirizzo privato di A (10.10.10.10) e come indirizzo destinazione l'indirizzo pubblico di B (11.11.11.11).

A questo punto il NAT andrà a tradurre l'indirizzo sorgente in quello pubblico ovvero quello del NAT (12.12.12.12) mentre manterrà invariato quello di destinazione.

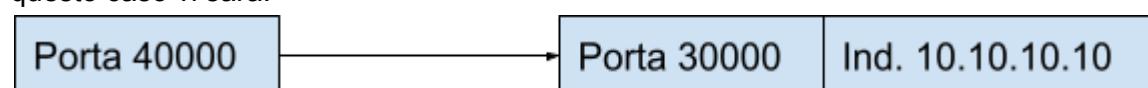
Se B volesse rispondere metterà nel datagramma l'indirizzo destinazione del NAT di A e quando il messaggio arriverà a destinazione il NAT di A tradurrà l'indirizzo destinazione del datagramma in quello di A ovvero 10.10.10.10.

Il NAT non traduce solo gli indirizzi IP ma **anche i numeri di porta**, i quali si trovano nel payload del datagramma.

**Per ricordarsi le traduzioni il NAT crea una tabella indicizzata con i numeri di porta che ha creato.**

Se A usa la porta 30000 e B la 5432 il NAT, leggendo il payload, non modificherà la porta destinazione ma solo quella sorgente, per esempio in 40000. Quando il messaggio verrà recapitato a B esso avrà come indirizzo sorgente quello del NAT di A (12.12.12.12) e come porta sorgente la 40000.

All'interno del NAT ci sono delle tabelle che memorizzano le conversioni che avvengono, in questo caso vi sarà:



Tutto questo processo serve per **ridurre il numero di indirizzi presenti sulla rete, perché a più macchine sotto la stessa rete locale corrisponde un unico indirizzo pubblico** (quello del NAT).

Vi è però una controindicazione: i numeri di porta sono codificati su 16 bit quindi il NAT non potrà avere più di  $2^{16}$  connessioni differenti; il problema è che molte porte sono predefinite e già occupate: le porte effimere (quindi utilizzabili) sono circa  $2^{15}$ .

Anche il NAT ha una cache.

## Port Forwarding

**Non è possibile inviare un messaggio ad una macchina che non abbia precedentemente comunicato con il router.**

Questo perché non conosciamo il suo indirizzo e il suo numero di porta, ed essi non sono presenti nella tabella del NAT.

Il NAT da solo può quindi funzionare solo lato client, perché il numero di porta è associato dinamicamente.

Per inserire un server all'interno di una sottorete bisogna utilizzare **il port forwarding**; esso è un'applicazione del NAT, che redirige una comunicazione da parte di una combinazione di indirizzo IP e un numero di porta a un'altra; questa redirezione avviene in un dispositivo apposito, che effettua la conversione mentre i pacchetti passano attraverso esso. Questo permette a dispositivi esterni di connettersi a uno specifico dispositivo della rete locale, a seconda della porta usata per la connessione.

Questa tecnica è usata solitamente per far sì che i servizi di un host residente in una rete privata possano essere disponibili agli host di una rete pubblica, attraverso la conversione degli indirizzi IP e del numero di porta.

Se i due dispositivi sono sotto una rete privata uno dovrà assumere il ruolo di client e l'altro di server; quest'ultimo dovrà fare la *bind()* prima di poter ricevere dei messaggi e qua entra in gioco **il port forwarding**.

Questo meccanismo è l'equivalente del NAT ma dal lato del server; quest'ultimo è configurato staticamente in modo che il numero di porta sia prestabilito.

Il port forwarding prevede quindi un **associazione statica** tra indirizzo privato e numero di porta pubblico; quest'ultimo deve essere prestabilito.

Esempio:

A ha un indirizzo pubblico 11.11.11.11 e uno privato 192.168.11.11

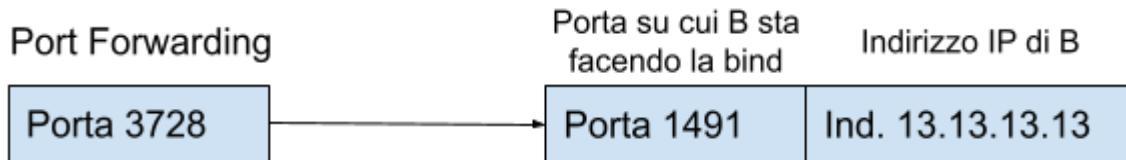
A si vuole connettere a B quindi B dovrà fare la *bind()* su una porta, per esempio la 1491.

A conosce l'indirizzo del port forwarding e il suo numero di porta ma non conosce quello di B quindi quando A manderà il messaggio a B sulla porta del port forwarding questo dispositivo tradurrà, oltre all'indirizzo IP, anche la porta in quella di B; questa associazione viene stabilita in precedenza e gli recapiterà il messaggio.

In breve:

B fa la `bind()` su una porta, il dispositivo di port forwarding associa la sua porta a quella di B.  
A vuole contattare B e manda un messaggio contenente come indirizzo e porta di destinazione quelli del dispositivo di port forwarding.

Il messaggio arriva a tale dispositivo il quale traduce porta e indirizzo IP in quelli di B il quale sta facendo la bind sulla suddetta porta.



## Sicurezza dell'Indirizzamento Privato

**Il NAT** permette di non ricevere connessioni a meno che non sia stata fatta una richiesta: **evita quindi connessioni indesiderate.**

Il port forwarding permette di mettere a disposizione i servizi di una macchina privata su una rete pubblica: evita quindi di esporre direttamente la macchina alla rete, **mascherandola** attraverso il router.

## (13)Lezione 27 (09/11/20)

### Protocollo ICMP

ICMP è un protocollo di livello 3 e serve per **aggiungere funzionalità relative al funzionamento dei router.**

Insieme ad IP cerca di **segnalare le condizioni di rete**: si occupa di trasmettere informazioni riguardanti malfunzionamenti, informazioni di controllo o messaggi tra i vari componenti di una rete di calcolatori.

IP è un protocollo **best-effort** e quindi, senza usare troppe risorse, cerca di segnalare, grazie ad ICMP, la maggior parte degli errori possibili.

ICMP è stato descritto inizialmente nel RFC 792, versione sviluppata in parallelo a IPv4;

Nell'RFC 2463 è descritta invece la versione sviluppata in parallelo a IPv6.

ICMP usa le funzionalità del protocollo IP quindi se il protocollo IP è di livello 3, ICMP dovrebbe essere al livello 4 ma siccome è un protocollo ausiliario, **complementare rispetto a IP**, viene identificato come di livello 3.

### Header e Richieste/Risposte ICMP

Prevede un header nella quale sono codificati due numeri interi:

1) **Tipo** (type)

2) **Codice** (code)

Dopo di che il datagramma viene completato con i primi 8 byte.

Type	Code	Comportamento
0	0	Risposta di tipo ping
3 [Errori]	0	Rete non raggiungibile
3 [Errori]	1	Host non raggiungibile
3 [Errori]	2	Protocollo non raggiungibile
3 [Errori]	3	Porta non raggiungibile
4	0	Ausiliario al Controllo di Congestione (quasi mai usato)
8	0	Richiesta di ping
9,10	0	Utilizzate dai router per le tabelle di instradamento
11	0	TTL scaduto

12	0	Segnala un errore nell'header IP
----	---	----------------------------------

Una delle applicazioni principali, che serve ad esempio per fare debugging nella rete, è **ping**.

L'applicazione ping sostanzialmente permette di **inviare dei messaggi da un host H1 a un host H2** (H1 invia il messaggio, H2 risponde e H1 così è in grado di calcolare l'RTT); permette in questo modo di **vedere se è possibile instaurare una connessione tra i due host e di calcolarne il RTT**.

Questi messaggi che vengono scambiati non sono però messaggi appartenenti ad un flusso TCP e non sono neanche datagrammi di tipo UDP; in questo caso sono direttamente **messaggi di tipo ICMP**.

Funzionamento di Ping:

- 1) Il messaggio che viene inviato è il messaggio ICMP *echo request* (corrispondente al tipo 8, code 0);
- 2) la risposta è di tipo ICMP *echo reply* (type:0 code:0).
- 3) L'applicazione mi permette di verificare che ci sia una connessione di rete utilizzabile tra H1 e H2 e misurare il RTT.

Per avere una misurazione più accurata del RTT vengono inviati una successione di ping attraverso messaggi ICMP *echo request* e volendo è possibile specificare una dimensione per il datagramma.

La caratteristica di questa implementazione dell'applicazione **ping** è proprio quella di non arrivare al livello 4, rimanere al livello 3 e quindi **non specificare nessuna porta**.

IP e ICMP risultano quindi **indispensabili l'uno per l'altro**: IP non può funzionare senza ICMP e viceversa.

## Traceroute

Traceroute è un software **di debugging della rete**.

**Serve per avere un'idea di quale possa essere un instradamento multi-hop dei datagrammi sulla rete e verificare il funzionamento e l'affidabilità di esso**; questo instradamento viene ricavato generando situazioni di errore e sfruttando ICMP che segnala la presenza di questi errori.

Implementazione Traceroute

Poniamo di avere: Host mittente A, Host destinazione B, rete di Hop con diverse possibilità di instradamento.

Viene preparato un datagramma UDP inserendo come destinatario l'IP della macchina su cui si vuole arrivare e una porta UDP presa a caso.

Viene inserito un valore molto basso per il TTL, per esempio 1; il nostro datagramma effettua il primo hop e arriva al primo router, il TTL viene decrementato, si scarta il messaggio e mi arriva una segnalazione di errore ICMP con *type 11*: questo datagramma sarà un datagramma IP e conterrà nel campo Source IP **l'indirizzo del mittente del primo router**; ricevendo la risposta ICMP questa applicazione è in grado di ricavare quindi l'IP del primo router.

Una volta ricevuto il messaggio di errore si incrementa di 1 il TTL e si invia nuovamente il messaggio (con TTL=2): il nostro messaggio effettua il secondo hop e sarà il secondo router a rispondere con il messaggio ICMP *type 11*, code 0.

In questo modo verrà riconosciuto qual è l'IP del secondo router, aumenta il TTL (TTL=3) il messaggio arriverà al 3 router, e così via.

**In questo modo si crea un elenco di tutti i router.**

Quando arriviamo all'ultimo router otteniamo per l'ultima volta il messaggio ICMP; aumentando ancora di uno il TTL il messaggio arriverà fino all'host di destinazione; siccome abbiamo scelto una porta casuale riceveremo un messaggio con *type 3, code 3* ovvero *porta non raggiungibile*, e capiremo che siamo arrivati a destinazione.

Questa tecnica ha il difetto che non c'è nulla che garantisca che **durante l'esecuzione non cambi l'instradamento**, ci potrebbe essere un aggiornamento delle tabelle di forwarding e si potrebbero eseguire due strade diverse.

## Data Link: Ethernet

Il livello di data link si occupa di **mettere in comunicazione due o più host (o un router) all'interno di una rete locale**.

Per rete locale si intende una rete di piccole dimensioni e che possa occupare una piccola area geografica (ad esempio un palazzo o una stanza).

Il protocollo principale a livello data link è **Ethernet**.

In passato sono stati sviluppati moltissimi protocolli diversi a livello di Data Link ma alla fine degli anni '70 si è affermato lo standard Ethernet IEEE 802.3.

Le reti di tipo **wireless**, oggi giorno estremamente diffuse, sono descritte nel IEEE 802.11.

Le **primissime versioni** di Ethernet facevano uso di un **cavo coassiale**: un cavo con un filo metallico interno, un dielettrico e una calza metallica esterna che funziona anche da schermatura per evitare le interferenze elettromagnetiche.

La comunicazione **avveniva in banda base** ovvero non c'era un sistema di modulazione: i **segnali venivano mandati sul filo e i singoli bit venivano codificati direttamente sotto forma di differenza di potenziale sul filo elettrico**.

Si trattava di un cavo di diametro di 2,5 cm; l'idea era di far correre questo cavo su una superficie estesa (**corridoio**) con la possibilità di **connettere nei punti desiderati, altri dispositivi** per poter collegare host.

Le connessioni avvenivano con una specie di **pinza** che **bucava il cavo** e una vite andava a toccare il cavo comportando la connessione.

Con una connessione di questo genere c'è la possibilità di inviare lo stesso messaggio a tutti quelli connessi a questo cavo: la comunicazione principale che viene realizzata sul cavo Ethernet è quindi di tipo **broadcast**.

## Indirizzi MAC

Supponiamo che H1 voglia mandare un messaggio a H3; se H3 è connesso riesce a ricevere il messaggio così come H2.

Per far ricevere solo ad H3 il messaggio si introduce un **meccanismo di indirizzamento** che è valido solo all'interno di questa rete locale; si usano gli **indirizzi MAC**.

Questi indirizzi sono definiti come configurazioni binarie su 6 byte (48 bit di indirizzo); l'idea è stata quella di poter garantire l'**unicità degli indirizzi** senza preoccuparsi dei dispositivi; ogni dispositivo ha il suo indirizzo MAC unico, per ottenere questa unicità i 6 byte sono stati suddivisi in 2 parti:

Costruttori	Dispositivi
-------------	-------------

- 1) 3 byte per i **costruttori**: identificano la ditta di produzione.
- 2) 3 byte per i dispositivi di quel determinato costruttore.

Questi indirizzi sono a loro volta incapsulati in un **frame ethernet**:

Trama (preamble)	MAC Sorgente	MAC Destinazione	Payload	CRC-32
------------------	--------------	------------------	---------	--------

- la **Trama** identifica l'inizio di un messaggio da parte di un host: nel canale passano infatti bit in continuazione, un dispositivo si "sveglia" solo quando riconosce una trama di un messaggio e procede quindi ad analizzarlo.
- MAC Sorgente e MAC Destinazione sono costituiti da 6 byte come descritto in precedenza.
- Il payload contiene il contenuto del messaggio.
- Il **CRC-32** è di 4 byte e permette di rilevare se sono presenti errori di trasmissione; in pratica, il ricevente calcola il CRC mediante un algoritmo e lo confronta con quello ricevuto in questo campo.

**È compito del singolo host di filtrare i messaggi sulla base degli indirizzi di destinazione**, infatti tutti i messaggi passano sul cavo e sono potenzialmente visibili da tutti.

Sta al singolo Host quello di andare a prendere soltanto i messaggi destinati al proprio indirizzo MAC.

Normalmente è suo interesse prendere soltanto i messaggi destinati a lui e non concentrarsi sugli altri; tuttavia ci sono dei casi dove può essere utile **vedere tutto il traffico** sulla rete; per questo motivo le NIC hanno una modalità di funzionamento detta **promiscua** che ignora il MAC address di destinazione e permette agli host di vedere tutti i messaggi che transitano sul canale.

## Protocollo CSMA/CD

- 1) **Coordinare i vari host tra di loro**, il canale è di tipo broadcast e quindi se un certo host sta inviando sul canale di comunicazione in un determinato momento, gli altri host non possono usarlo.

Per ovviare a questo problema sono stati sviluppati vari protocolli, quello a cui facciamo riferimento è il **CSMA/CD: Carrier Sense Multiple Access with Collision Detection**, ovvero **accesso multiplo tramite rilevamento della portante con rilevamento delle collisioni**).

Gli host che sono connessi al nostro canale di comunicazione sono indipendenti uno dall'altro e quindi in qualsiasi momento ogni host può decidere di mandare un messaggio.

**Carrier Sense (CS):** Per mandare il messaggio è necessario che non ci siano altre connessioni in corso, quindi l'host **monitora** cosa sta succedendo nel canale e, solo quando c'è un momento di silenzio, manda il suo messaggio.

Supponiamo di avere una situazione in cui H2 ha cominciato a trasmettere e sta mandando il messaggio ad H3, H1 vuole mandare un messaggio ad H4 ma non può farlo perché il canale è occupato da H2, quindi H1 aspetta la sequenza finale del messaggio di H2, dopo di che la connessione finisce e H1 può iniziare a trasmettere.

H1 prende il canale di trasmissione e inizia a trasmettere, però anche se gli altri host stanno facendo il **Carrier Sense** esiste la possibilità che ci sia qualche altro host, per esempio H3, che voglia usare il cavo, ad esempio per rispondere al messaggio ricevuto.

Per esempio, H1 e H3 vedono che il canale è libero:

- H1 inizia a mandare il messaggio verso H4
- H3 comincia contemporaneamente a dare la sua risposta ad H2

Per un po' di tempo nessuno dei due riconosce la presenza di questa **collisione**, a causa della **lunghezza del canale**.

Bisogna quindi ovviare, in caso, all'inconveniente della collisione, ovvero la sovrapposizione di due o più messaggi sullo stesso canale broadcast;

Per prima cosa però dobbiamo identificare la collisione: **Collision Detection (CD)**; il protocollo CD rileva se vi è una collisione in corso.

A questo punto la collisione viene risolta tramite il protocollo **Multiple Access (MA)**, esso funziona in 3 fasi:

- 1) **Jamming**: mandare una sequenza di bit particolare che serve per rendere evidente la presenza della collisione e assicurarsi che anche l'altro host ne sia a conoscenza; deve durare il tempo necessario per propagarlo fino all'altra estremità del cavo,
- 2) **STOP: Interruzione della trasmissione**: liberazione del canale,
- 3) **Ritrasmissione**, tramite un **meccanismo di espansione esponenziale dei tempi di attesa**:
  - la prima volta si attende una unità di tempo e poi si decide se inviare subito il messaggio o aspettare; se si prende la decisione sbagliata (probabilità di  $\frac{1}{2}$ ) e si genera un'altra collisione allora si raddoppia l'unità di tempo da aspettare, e così via crescendo esponenzialmente; **man mano che si va avanti diminuisce la probabilità di collisione, fino ad arrivare ad essere praticamente impossibile**. I messaggi vengono mantenuti nei buffer di trasmissione finché non arrivano a destinazione.

Un'unità di tempo sono i secondi che ci mette un messaggio a percorrere il cavo di connessione.

Quindi, ricapitolando:

- facendo **CS evita la maggior parte collisioni**: se vede che qualche host sta già trasmettendo non va a trasmettere ulteriori informazioni.
- Aspetta che il canale sia libero, dopo di che parte a trasmettere ma durante **l'intervallo di vulnerabilità**, qualche host potrebbe in contemporanea trasmettere e quindi la collisione risulta inevitabile.
- La collisione viene appunto riconosciuta dal protocollo **CD** e gestita grazie al **MA**, un protocollo di trasmissione con intervalli di tempo di ritardo via via crescenti con **limite massimo di  $2^{10}$** .

## Dispositivi Ethernet

Le prime versioni di ethernet con cavo coassiale viaggiavano a una velocità di 10 Mb/s; nel tempo la velocità è cresciuta grazie all'adattamento di nuove tecnologie.

### HUB

L'HUB è ripetitore e amplificatore di segnale: permette di non bucare i cavi, usando spinotti di connessione, il che aumenta di molta **l'affidabilità**, evitando che il cavo si rompa.

La connessione non è quindi più di tipo bus ma ha una struttura a stella.

Si passa inoltre da cavo coassiale ad un cavo detto **doppino telefonico**, che aumenta la velocità di connessione, che passa a 100 Mb/s.

## Switch

L'ultima versione di ethernet (attualmente in uso) ha un modo di funzionamento completamente diverso, basato sul dispositivo chiamato **switch**; la connessione di tipo switch è concettualmente simile a connessione mediante HUB, anche se cambia sia a livello fisico che di funzionamento.

Un dispositivo di tipo switch ha un numero massimo di connessioni possibili.

Differenze con L'HUB:

- 1) **Full Duplex: due doppini telefonici;** 2 fili che servono ad inviare e 2 che servono per ricevere, costruiscono un canale fisico chiamato full duplex, ovvero che permette di inviare e ricevere contemporaneamente.
- 2) Dentro lo switch c'è qualcosa al livello di datalink e il funzionamento dello switch è di tipo **store & forward**: l'idea è che ciascuna delle porte di connessione abbia dei buffer di ricezione e dei buffer di trasmissione.

La trasmissione avviene così:

- Host mittente manda la trama Ethernet allo switch,
- lo switch memorizza la trama all'interno del buffer di ricezione,
- dopo di che va a vedere il Mac address di destinazione del messaggio e sposta questo messaggio nel buffer di trasmissione della porta connessa all'host di destinazione.

Caratteristiche di una Rete basata su Switch:

- Velocità stessa di un HUB: **100 Mb/s**, ma questi Mb/s sono **raggiungibile da ogni singolo host**, mentre nell'HUB vengono spartiti tra tutti i dispositivi collegati.
- Posso inoltre, **sia inviare (upload) che ricevere (download) alla velocità di 100Mb/s ciascuno.**
- **Sparisce l'idea di avere un canale di tipo broadcast e di avere collisioni.**
- L'unico svantaggio di fare store & forward è che **si aumenta la latenza**: prima che il messaggio mandato da h1 possa essere ricevuto da h3, deve essere prima completamente ricevuto dallo switch, smistato e ritrasmesso allo switch; il tempo di ricezione e ritrasmissione sullo switch si sommano ai tempi di propagazione sui canali fisici.

# (14)Lezione 30 (16/11/20)

## Ancora su Ethernet

Ricapitolando: i MAC address sono numeri codificati su 6 byte.

I messaggi mandati dagli host sono codificati tramite un **ethernet frame**, costituito da vari campi:

Trama (preambolo)	MAC Sorgente	MAC Destinazione	Payload	CRC-32
-------------------	--------------	------------------	---------	--------

## Switch: Algoritmo di Autoapprendimento

Lo switch deve inoltrare il messaggio su una porta in uscita in cui è collegato l'host destinatario, per farlo va a vedere il **frame**, legge l'indirizzo del destinatario e inoltra il messaggio alla porta giusta.

Per fare ciò lo switch deve essere a conoscenza di tutti gli indirizzi degli host collegati e qua entra in gioco un algoritmo:

Questo **algoritmo di autoapprendimento** costruisce una tabella di corrispondenza che **assegna ad ogni porta un indirizzo MAC**; esso è in continua esecuzione perché in ogni momento potrebbero venire attaccati nuovi dispositivi.

L'algoritmo ha 2 fasi:

- 1) **Fase di Apprendimento (Dinamico):** L'algoritmo parte all'accensione dello switch e all'inizio avrà una tabella vuota, quando viene collegato un dispositivo esso non viene registrato subito ma solo nel momento in cui manda un messaggio; lo switch va quindi a leggere il messaggio, vede l'indirizzo MAC sorgente e lo associa a quella porta della tabella.

Se un host non ha ancora mandato un messaggio e ne deve ricevere uno lo switch non sa il suo indirizzo MAC; viene quindi mandato quel messaggio in **broadcast** su tutte le uscite e quando l'host destinatario risponderà il suo MAC verrà associato alla sua porta.

Nella fase iniziale (appena acceso) lo switch sarà più lento perché dal momento che non conosce gli indirizzi MAC di nessun dispositivo utilizzerà il broadcast per ogni messaggio.

Il suo funzionamento e la sua velocità in questa fase sono comparabili a quelli dell'hub; in realtà lo switch è leggermente più lento a causa della latenza maggiorata.

- 2) **Fase Statica:** Una volta costruite tutte le associazioni, si entra nella fase statica, dove non c'è necessità di mandare messaggi broadcast ma possono essere inviati messaggi **punto a punto**.

La fase iniziale risulta quindi **lenta**, a causa dell'utilizzo di messaggi broadcast; quando si entra nella fase statica invece, si può apprezzare la velocità della connessione tramite switch, che a questo punto è molto elevata.

Da notare è il fatto che è possibile che tanti host siano associati alla stessa porta, ad esempio in caso di molteplici switch collegati fra loro; questo complica di molto le cose, ma non ci interessa.

Lo switch possiede inoltre una modalità **promiscua**, ma bisogna essere admin per attivarla; in essa, un host può ricevere tutti i messaggi passanti per quello switch.

Per utilizzare questa modalità, sempre se si è amministratori di rete, si fa in modo che il mio host mandi messaggi con MAC diversi dal mio: ad esempio, se voglio ricevere i messaggi destinati all'host H1 mando un messaggio contenente come MAC sorgente quello di H1 e lo switch registrerà sulla porta collegata al mio host il MAC di H1 inoltrandomi tutti i messaggi destinati a lui.

Questa operazione può essere fatta da un malware di tipo **sniffing** o anche con attacchi di tipo **spoofing** ovvero intasando lo switch con tantissimi MAC e quindi riempiendo la tabella per rallentare tutto.

## Rete Locale (LAN): Livelli 2 e 3

Tutti gli host sotto la stessa LAN (Rete Locale) hanno la prima parte dell'indirizzo IP uguale (quello della subnet per l'esattezza), ad esempio: 192.168.1.33 e 192.168.1.13 sono indirizzi IP di dispositivi sotto la stessa LAN.

Con il **protocollo ARP** (Address Resolution Protocol) il router associa l'indirizzo IP a un MAC.

### Protocollo ARP

L'ARP è un protocollo di **livello applicativo**, che usa **UDP/IP** al fine di permettere una comunicazione di tipo **broadcast**.

Essa è fondamentale per il meccanismo che ha il fine ultimo di **associare un indirizzo MAC ad un indirizzo IP**.

Se un messaggio uscente da un host contiene come indirizzo destinazione un indirizzo che inizia con gli stessi numeri di quello della *subnet* sotto cui è l'host vuol dire che **quel messaggio è destinato ad un dispositivo nella stessa sottorete**.

L'indirizzo di broadcast a livello IP è costituito nella prima parte dell'indirizzo della **subnet**, seguita da **tutti 1** (al posto dell'indirizzo del destinatario):

Host#	
IP Broadcast:	Subnet
	111....

L'IP broadcast servirà al livello di rete delle richieste ARP.

A livello ethernet esiste un **indirizzo broadcast** che è una combinazione di 6 byte che viene interpretata come "questo messaggio è destinato a tutti gli host sotto questa rete locale".

Questo indirizzo è costituito da sole F (siamo in esadecimale):

FF:FF:FF:FF:FF:FF

Le richieste e le risposte ARP, rispettivamente chiamate **arp request** e **arp reply** sono incapsulate in **frame**, costituiti da:

Preamble	MAC Source	MAC Destination	ARP	Payload	CRC-32
----------	------------	-----------------	-----	---------	--------

Ai campi analizzati in precedenza si aggiunge quindi il campo ARP, diviso in 28 byte:

I 28 byte per l'ARP Request/Reply sono strutturati in questo modo:

- i campi: hardware type, protocol type, hardware len e protocol len, utili alla buona riuscita della richiesta o risposta ma che non ci interessano.
- **ARP operation**: specifica se si tratta di una richiesta ARP (valore 1), risposta ARP (valore 2).

Una richiesta ARP funziona in 3 passi:

- 1) un host che manda una *arp request* a tutti gli host sotto quella rete, facendo una richiesta di tipo broadcast;  
Quindi per effettuare una *ARP request* preparo un datagramma UDP contenente come payload l'indirizzo IP di cui voglio conoscere il MAC.
- 2) tutti gli host che non hanno quell'indirizzo IP **non rispondono**; risponde solo chi possiede quell'indirizzo IP, con un *arp reply* contenente come MAC sorgente quello che stavo cercando;  
La risposta avrà come **MAC sorgente quello della macchina che mi risponde**, come MAC destinazione il mio e come payload il MAC che cercavo.
- 3) posso quindi associare il suo indirizzo IP al suo indirizzo MAC.

**Questa operazione consuma molte risorse;**

Per cercare di fare meno broadcast possibile il protocollo ARP ha una propria **cache**, dove vengono memorizzate le associazioni tra IP e MAC.

L'ARP *request* mette quindi **in comunicazione il secondo con il terzo livello**.

Ethernet ↔ IP

Cache Poisoning e Man in the Middle

Il protocollo ARP è **senza stato (stateless)**, non memorizza quindi le richieste che ha inviato o le risposte che ha ricevuto; questo lo rende vulnerabile a vari attacchi alla sicurezza.

Abbiamo capito che l'*ARP Request* serve per sapere il MAC di un indirizzo IP e questo protocollo ha una **cache** che memorizza i MAC degli IP; essa è vulnerabile al **cache poisoning** ovvero si mandano delle *ARP reply* con informazioni sbagliate che verranno inserite all'interno della cache.

A questo punto si è vulnerabili ad attacchi di tipo **Man in the Middle** ovvero se E vuole intromettersi nella comunicazione tra A e B può mandare ad A una risposta dove dice che l'IP di B corrisponde al MAC di E e può mandare una *reply* ARP a B dove dice che l'IP di A corrisponde a quello di E.

A questo punto i messaggi destinati ad A e B verranno inoltrati dallo switch a E che volendo può modificarli e rimandarli ai destinatari.

L'attacco Man in the Middle può essere:

- **passivo**: l'host malevolo legge i messaggi senza modificarli e poi li inoltre inalterati,
- **attivo**: l'host malevolo legge e modifica il contenuto dei messaggi fra i due host.

Praticamente il cache poisoning **falsifica le corrispondenze IP e MAC** e basta che un solo host della sottorete sia malevolo per rischiare questo attacco; **non ci sono modalità di difesa per questo attacco**.

Il protocollo ARP si può però disattivare in qualsiasi momento rendendo statico l'assegnamento dei MAC agli IP.

## Protocollo DHCP: Richiesta di un Indirizzo IP

Un protocollo simile, di livello applicativo, è il **DHCP** (Dynamic Host Configuration Protocol) molto utile per i dispositivi che possono accendersi e spegnersi e quindi scollegarsi e ricollegarsi alla rete (ovvero non i server).

Questo protocollo **serves per assegnare un indirizzo IP ad un dispositivo che si sta collegando alla rete** e presuppone che ci sia almeno un server DHCP all'interno della LAN.

Dopo esserci connessi alla rete il primo passo è quello di chiedere al server di assegnarci un indirizzo IP:

Per fare la richiesta si usa il protocollo **UDP/IP** (porta 68 client e 67 server) che però ha bisogno di un indirizzo IP sorgente e un indirizzo IP destinazione (quello del server) ma in quel momento non conosciamo nessuno dei due; ci si autoassegna quindi un indirizzo **IP fasullo**, ad esempio *0.0.0.0*, e si manda la richiesta in **broadcast**.

Per mandare la richiesta in broadcast dobbiamo conoscere l'indirizzo MAC sorgente e destinazione: l'indirizzo MAC sorgente sarà il mio mentre quello destinazione è quello broadcast ovvero *FF:FF:FF:FF:FF:FF*.

Questi indirizzi li mandiamo racchiusi in un frame ethernet.

Il mio messaggio viene recapitato dallo switch a tutti gli host ma solo i server che hanno fatto la *bind* su quella porta lo riceveranno.

I server DHCP nella rete locale hanno una **lista di indirizzi IP disponibili** a chiunque voglia connettersi e risponderanno alla richiesta con uno di quelli.

Chiaramente i server per rispondere alla richiesta dovranno mandare un messaggio al nostro MAC e indirizzo IP fasullo (0.0.0.0 in questo caso).

L'host attende quindi lo scadere di un **timeout** e va a ad osservare le risposte collezionate dai vari server; l'host manda quindi un **messaggio di accettazione** al server che gli notifica l'accettazione quell'indirizzo IP.

L'accettazione viene comunicata in **broadcast**, come la richiesta, in modo da comunicare a tutti i server DHCP sotto quella rete il mio indirizzo IP, facendo sì che essi lo **cancellino** dalla loro lista di indirizzi possibili; in questo modo inoltre, se più server DHCP mi hanno dato la disponibilità di un indirizzo IP, comunico quello che ho scelto.

Il server DHCP che ha fornito l'indirizzo comunica quindi all'host le informazioni principali sulla rete e indica una **scadenza** di questo indirizzo IP; una volta scaduto l'indirizzo torna ad essere disponibile e l'host dovrà richiederne un altro; non è detto che gli venga riassegnato lo stesso, a meno che non si proceda con un'assegnazione statica di esso.

Ricapitolando:

1. Richiesta broadcast
2. Risposta
3. Accettazione broadcast
4. Conferma

Una volta che ricevuto quell'indirizzo IP lo posso utilizzare normalmente all'interno della rete e utilizzerò il protocollo ARP per conoscere i MAC degli altri dispositivi sotto quella rete.

Generalmente l'indirizzo IP ha una **scadenza** dopo la quale dovrò ripetere tutto il processo per riceverne un altro; tutto questo sarà da ripetere anche in caso di disconnessione del dispositivo dalla rete o spegnimento di esso.

Gli indirizzi IP di dispositivi non più sotto quella rete possono essere **riassegnati** ad altri dispositivi, stessa cosa per l'indirizzo IP pubblico, essi ritornano quindi a far parte della lista degli indirizzi disponibili.

Questa è detta **configurazione dinamica**.

Volendo posso anche assegnare **staticamente** gli indirizzi IP.

## (15)Lezione 35 (30/11/2020)

### Protocollo HTTP

Abbiamo visto tre versioni del protocollo HTTP:

- 1.0 (del 1996), descritta nel RFC 1945,
- 1.1 (del 1999), descritta negli RFC 2068 e 2616,
- 2 (del 2015), descritta nel RFC 7540; quest'ultima è quella attualmente in uso.

Il protocollo HTTP è un protocollo di tipo **client-server**:

Il client, solitamente il browser, apre la connessione **TCP sulla porta 80** per mettersi in comunicazione con il server; a questo punto si effettuano una serie di richieste e risposte codificate in ASCII, così strutturate:

Richiesta (Client):

Metodo	Risorsa	Protocollo
	Host	
	User Agent	
	Cookie	
	If-Modified-Since	
	connection_close	
	...altri opzioni (arbitrarie)	

La prima riga contiene **l'indicazione di:**

- **un metodo,**
- **una risorsa,**
- **un protocollo da utilizzare.**

Un **metodo** è il tipo di operazione che si vuole effettuare, esempio: *get, head, put, delete, patch, option, trace, connect*.

La **risorsa** di riferisce al *path* su cui vogliamo agire; ad esempio *GET index.html 1.0* richiede la risorsa *index.html*;

I **protocolli** sono ad esempio quelli descritti in precedenza; uno di essi va specificato come ultimo campo della riga; essi permettono l'upload e il download dai server, per esempio il metodo **get** corrisponde al **download di una risorsa** identificata da un *pathname*, il metodo **post** invece corrisponde **all'upload di informazioni** dal client al server; il *post* è quasi sempre disattivato nei server.

La prima riga termina con il **terminatore di riga <CR><LF>**.

Alla prima riga possono seguire delle **righe opzionali** che contengono delle opzioni, per esempio:

- 1) **Host:** permette di **impostare il nome dell'host a cui ci si vuole riferire**; questa opzione può essere utile quando viene utilizzata la funzionalità virtual host ovvero la possibilità di mettere sullo stesso server diversi domini e quindi potendo decidere attraverso il nome a quale dominio riferirmi.

- 2) **User-Agent**: specifica il tipo di client che sta effettuando la richiesta (nome del browser, versione eccetera) per permettere al server di utilizzare il metodo più consone per rispondere.
- 3) **Cookie**,
- 4) **If-Modified-Since**,
- 5) **connection\_close** nella 1.0.

Il numero delle opzioni è **arbitrario** e ogni opzione deve essere terminata dai caratteri \n\r ovvero <CR><LF> (terminatori di stringa).

Per dire che non ci sono altre opzioni si inserisce una **riga vuota**, che identifica anche la fine del messaggio di richiesta.

Risposta (server):

Stato	
Server	Content-Type
Payload	

E' composta da una prima riga che contiene lo **stato**, codificato in forma numerica, a cui è associata una stringa leggibile ad occhio umano; i codici principali sono:

- 200 che significa “**ok**”,
- 301 che significa “**moved permanently**”, ovvero che le risorse non sono più su quel sito,
- 304 che significa “**not modified**” nel caso in cui il client possedesse già la versione più recente della risorsa richiesta,
- 400 che significa “**bad request**” ovvero richiesta mal formata,
- 404 “**not found**” ovvero pagina non trovata,
- 500 che significa “**internal server error**” ovvero errori interni del server,
- 505 che significa “**HTTP method not implemented**” ovvero che quel server non è in grado di rispondere a tale richiesta.

Dopo di che si indicano le varie opzioni, le più comuni sono:

- 1) **Server**: mi comunica il tipo di server che sta rispondendo.
- 2) **Content-Type** che indica il tipo di file allegato con la risposta.

A questo punto abbiamo una **riga vuota** e poi il body che contiene il contenuto del file richiesto.

La codifica ASCII si limita alla parte header, non al body.

## Versione 1.0

E' la versione più **semplice** del protocollo, che si complica poi nelle versione successive; questa versione ha delle connessioni **non permanenti** (dopo aver effettuato la richiesta e ricevuta la risposta viene **chiusa la connessione**).

Tra le righe opzionali infatti è presente **connection\_close** che chiude la connessione al termine di una comunicazione.

Questa modalità semplifica l'implementazione del protocollo e:

- **evita di intasare il server con tante connessioni aperte**
- **ma è poco efficiente**, perché per ricevere un file devo aspettare 2 round trip time: uno per la 3 way-handshake e uno per la risposta.

Dopodichè bisognerà rifare il 3 way-handshake perché la connessione è stata chiusa.

## Versione 1.1

Dal momento che la versione 1.0 non era efficiente si è passati alla 1.1 rendendo la connessione permanente; **la connessione rimane quindi aperte fino allo scadere di un timeout**, che parte quando i due host non si comunicano più niente.

Nella versione 1.1 vi è anche la **richiesta in pipeline** ovvero: dopo aver aperto la connessione il client può mandare la prima richiesta, e ricevuta una risposta **può mandare in successione altre richieste senza aspettare ogni volta la risposta del server**.

In questo modo si utilizza:

- **1 RTT per aprire la connessione,**
- **1 RTT per ogni file nella pagina.**

Il server dovrà rispondere alle richieste del client **nello stesso ordine in cui sono state poste**.

Quando il client pone la prima richiesta deve aspettare una risposta da parte del server prima di poter effettuare le altre richieste in successione. Questo è l'unico momento in cui non si può fare pipeline perché **la prima richiesta indica la quantità di file da scaricare sul client**.

Un limite della pipeline è il **vincolo di dover mantenere lo stesso ordine per le richieste e le risposte** e quindi se la prima richiesta richiede il download di un file di grandi dimensioni (come una foto del mio pene) il client dovrà aspettare di aver scaricato tutto il primo file prima di poter scaricare i seguenti. Per ovviare a questo problema si è passati alla versione 2.0.

## Versione 2.0

La versione 2.0 prevede l'utilizzo di un sistema di **multiplexing** attraverso l'uso di **identificatori** in modo tale da **associare ad ogni richiesta una risposta** avente lo stesso identificatore, togliendo in questo modo il vincolo di rispettare l'ordine richiesta-risposta; in questo modo richieste e risposte possono essere inoltrate nel modo più efficiente possibile.

In questa versione abbiamo anche altre ottimizzazioni come la **compressione degli header** utilizzando un algoritmo simile a quello degli ZIP.

## Meccanismo dei cookie

Essendo HTTP un protocollo **senza stato** (il server si dimentica, subito dopo aver risposto, la domanda che gli era stata posta e chi gliel'aveva posta) vi sono delle **problematiche in ambito di autenticazione**, per esempio se facessimo l'accesso su un sito con *username* e *password* ogni volta che ricaricheremmo la pagina dovremmo nuovamente fare il login.

Per questo motivo sono stati introdotti i **cookie**: nell'header della risposta da parte del server vi può essere una stringa *set-cookie* contenente il nome, il valore (*uid*) e la data di scadenza di esso. Questa stringa assegna al client un cookie che esso memorizzerà e rimanderà al server ad ogni futura richiesta in modo da essere riconosciuto da quest'ultimo attraverso l'*uid* salvato nel cookie.

I cookie possono essere quindi utilizzati per:

- **accedere ad aree riservate:** una volta effettuata l'autenticazione, viene mandato al client un cookie, che egli invierà al server nei prossimi accessi, per evitare di doversi autenticare nuovamente;
- **meccanismi di tracciamento:** il server imposta il cookie con un identificatore univoco, in modo da poter ricostruire la sequenza delle richieste del client

## Proxy

Vi è anche un **meccanismo di caching** per velocizzare l'accesso alle pagine web che consiste nel connettersi al server con un [\*\*proxy\*\*](#) che **mantiene delle copie dei file mandati dal server**.

Il proxy si interpone nella comunicazione tra client e server:

- si comporta come server nei confronti del client
- si comporta come client nei confronti del server

Il client contatta il proxy, questo inoltra la richiesta al server, esso risponde al proxy, il quale **memorizza** la risposta, e la inoltra al client.

Il proxy immagazzina quindi **copie temporanee** di file del server.

In questo modo se il client richiedesse nuovamente una risorsa già chiesta in precedenza il proxy gliela rimanderebbe indietro senza dover contattare il server velocizzando così la procedura e alleviando il carico di lavoro del server.

Questa procedura si può fare anche nelle reti locali utilizzando una macchina locale come **caching proxy**.

Vi è però un problema di consistenza infatti la copia dei file contenuta nel proxy potrebbe non essere aggiornata rispetto a quella del server e quindi è stato introdotto l'**header last-modified** nella risposta da parte del server;

Questo header specifica la data in formato [\*\*GMT\*\*](#) che indica l'ultima modifica di quel file sul server; in questo modo il client può fare una **richiesta condizionale** aggiungendo l'header

**if-modified-since** e, se la sua versione del file è aggiornata, gli verrà risposto con 304 “*not modified*” senza nessun file allegato.

Il momento in cui mandare un *conditional get* viene deciso dal browser web e volendo si potrebbe anche disattivare questa opzione togliendo così il meccanismo di caching.

Il proxy può essere di tipo:

- 1) **trasparente**: viene inserito a livello di gestione della rete e può essere utilizzato per **filtrare il traffico** in entrata e uscita decidendo quali richieste o risposte far passare e quali bloccare.
- 2) **esplicito**: il client decide esplicitamente di utilizzare un proxy per aumentare l’efficienza della comunicazione.

## (16)Lezione 38 (07/12/2020)

### IncApache

in questo laboratorio bisogna implementare un server web che risponde alle richieste HTTP. Il nostro server verrà contattato da un certo numero di client, sarà in attesa su una porta (quella di default è 8000), riceverà come primo argomento la directory dove trovare i file, per esempio *www-root*, e verranno fatte delle richieste, ad esempio *get-index* ecc.

Una problematica di sicurezza è la possibilità che il servizio venga utilizzato per prendere dati dal PC che ospita il server e quindi bisogna evitarlo system call chroot che cambia la root del file system visibile da quel processo.

Bisogna tenere in considerazione che la system call funge solo se l’user id del processo è zero ma dopo averlo fatto bisogna riportarlo ad uno altrimenti l’utente potrebbe rifare la chroot e cambiare di nuovo la directory.

L’eseguibile dovrà avere permesso del proprietario “s” e per gli altri dovrà avere “x”.

Per poter rispondere alla richiesta HTTP devo poter produrre un header con mime type.

L’applicazione che si occupa di ottenere il mime type è in */bin* ma avendo fatto la *chroot()* non possiamo accedervi perciò attraverso una *fork()* duplichiamo il processo prima della chroot e poi attraverso una pipe li mettiamo in comunicazione.

### Accesso concorrente tra più client

Più client devono potersi connettere e un client può fare più richieste di connessione quindi per fare ciò si usano una serie di thread, ognuno dedicato alla gestione di una connessione. uno dei thread deve eseguire la system call *accept()* ma non mentre anche un altro lo sta facendo nello stesso momento. Questa mutua esclusione viene ottenuta con le strutture di sincronizzazione mutex ovvero dei contatori che contano i thread che stanno eseguendo codice protetto dalla mutua esclusione.

Ogni thread per entrare in quella muta esclusione deve effettuare una *mutex-lock()* fare l’*accept()* e poi la *mutex-unlock()*.

Più richieste corrispondono a più thread che lavorano in parallelo.

Il main si occupa di fare la *chroot()*, le *bind()*, le *fork()*, le *pipe()* ecc. I *to be done 4.0* sono obbligatori mentre i *to be done 4.1* sono facoltativi.

*incapache\_thread.c* implementa la gestione dei thread, l'*incapache\_http* invece si occupa del riconoscimento delle richieste da parte dei client e l'invio delle risorse.

Le funzionalità richieste sono il riconoscimento dei metodi GET e HEAD, non il POST.

Il metodo GET prevede di mandare la risorsa richiesta.

Non bisogna terminare l'esecuzione del server per via di un errore della richiesta da parte del client.

Se la richiesta va bene rispondiamo con *200-ok* altrimenti con un errore. Per esempio 300 o 500 a seconda del tipo di errore.

Importanti sono le funzionalità, implementate con gli header opzionali di HTTP, *conditional\_get* e il meccanismo dei cookie.

Il *conditional\_get* prevede delle richieste con un flag *modified\_since* il quale contiene una data che verrà confrontata con quella dell'ultima modifica (ottenuta attraverso la system call *stat()*) del file e se il file è aggiornato si risponde *304-not\_modified* e non si manda il file altrimenti si risponde con *200-ok* e lo si manda.

La manipolazione delle date è gestita in *incapace\_aux.c* quindi bisogna farlo prima della parte del conditional get altrimenti non funziona un belino.

Il server verifica che la richiesta non contenga cookie e se è così vuol dire che quello è un nuovo client.

Il meccanismo dei cookie richiede la collaborazione dei client la infatti prima richiesta da parte di un client non conterrà cookie, il server se ne deve accorgere e attraverso una set cookie deve fare in modo che quel client a d ogni successiva richiesta mandi anche un cookie.

Ad ogni client, identificato dal cookie, corrisponde un numero contenuto in un array che aumenta ad ogni richiesta mandata da quel medesimo client.

Nella relazione del laboratorio spiegare come abbiamo fatto il debugging.

## (17)Lezione 40 (14/12/2020)

Seconda Parte del laboratorio.

# Lagorio

## (1)Lezione 2 (22/09/20):

### Introduzione ai Sistemi Operativi

#### Obiettivo del Corso

L'obiettivo di questa parte del corso è lo studio dei **Sistemi Operativi**.

I **S.O. si basano su** 3 principi fondamentali:

**Virtualizzazione, Concorrenza e Persistenza.**

Andremo ad analizzare a fondo questi 3 aspetti, in particolar modo la virtualizzazione.

## Registri

Piccole porzioni di memoria.

A differenza delle celle di RAM essi sono molti meno e sono contenuti nella CPU.

Inoltre, essi non hanno un indirizzo ma un **nome** (ex.  $R_o$ ).

Queste caratteristiche (nome e vicinanza con la CPU) rendono i registri molto più veloci della RAM.

## Programmi

Un programma, per funzionare, necessita del funzionamento di molte componenti di un computer, ma in particolare di una: la **CPU**.

Essa infatti permette di eseguire tutte le istruzioni di un programma in 3 fasi:

- 1) **Fetch**: guarda all'interno di un registro detto **EIP** (extended instruction pointer), da cui acquisisce l'istruzione da eseguire.
- 2) **Decode**: decodifica la stringa di bit dell'istruzione.
- 3) **Execute**: esegue l'istruzione.
- 4) Ritorno alla fetch con **controllo** che non ci siano stati **interrupt** (interruzioni) derivanti dal mondo esterno.

## Sistema Operativo

Il S.O. si trova nel livello 3 della gerarchia di astrazione di un calcolatore, e si interpone tra i livelli software (L4 Librerie, L5 Applicazioni) e i livelli hardware (da L2 a L-2)

E' costituito principalmente dal **Kernel (Nucleo)** e da altri software necessari al funzionamento (ambiente grafico, shell, editor di testo, ...).

Il kernel è quindi una parte importante ma non costituisce la totalità del S.O., ad esempio:

*Linux è un Kernel; Ubuntu Linux è un S.O.*

## Kernel

Come abbiamo già detto è il nucleo del S.O., e si occupa di varie mansioni:

- Fa da **ponte tra hardware e software**, rende possibile la comunicazione tra livelli software (L4, L5) e i livelli hardware (da L2 a L-2)
- **Gestisce e virtualizza le risorse**
- **Gestisce i processi** (programmi in esecuzione) garantendone l'**efficienza** e l'**equità** tra essi a livello di sfruttamento della CPU

Il kernel riesce a svolgere queste mansioni grazie alle sue componenti:

## Device Driver

Parti di Software, scaricabili e installabili dalla rete o da disco, che consentono ad un dispositivo hardware (come ad esempio un dispositivo I/O) di comunicare con i livelli più astratti.

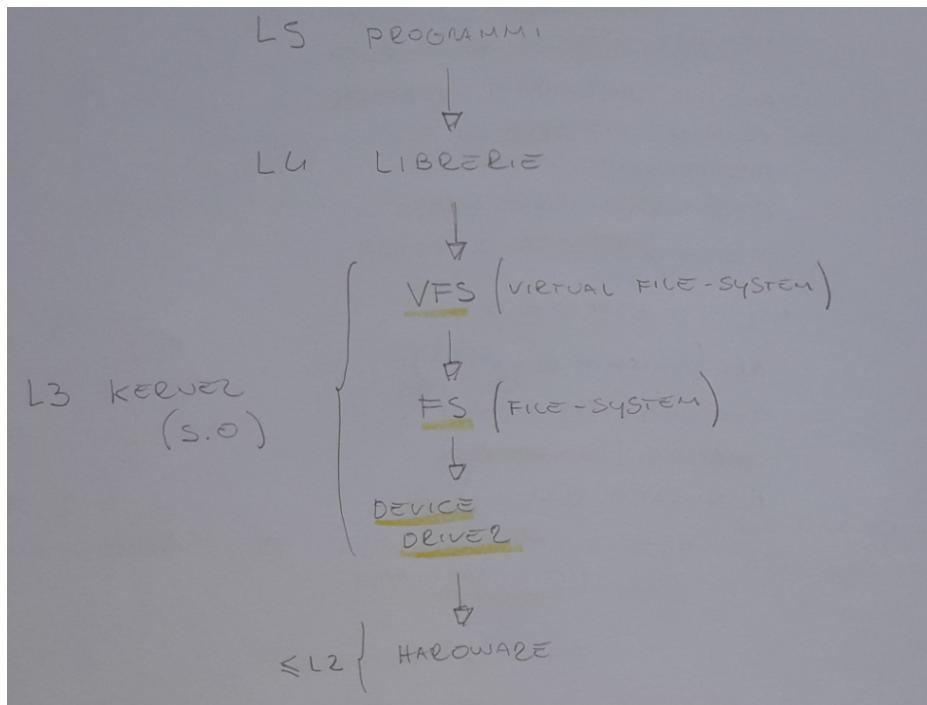
## File-System

Struttura dati che contribuisce alla gestione delle risorse stabilendo un ordine sui dischi.

## VFS: Virtual File-System

Stabilisce la comunicazione e l'ordine tra i vari file-system.

Schema Riassuntivo:



## (2)Lezione 4 (25/09/20):

### Macchina Virtuale e Chiamate di Sistema

Per ogni processo il S.O. crea una **macchina virtuale**, con la propria CPU virtuale e tutte le altre sue componenti.

Alcune istruzioni sono eseguite dalla CPU ma non tutte sono disponibili, esistono infatti istruzioni **speciali/privilegiate** dette **API (chiamate di sistema)**.

### Virtualizzazione

La virtualizzazione oltre a minimizzare l'**overhead** ovvero il quantitativo di risorse non primarie utilizzate per eseguire un processo, fornisce anche **protezione e isolamento tra processi diversi**.

Il sistema operativo praticamente **per ogni processo crea una macchina virtuale** in modo tale che quando viene eseguito è come se avesse a disposizione una propria CPU ed un proprio quantitativo di RAM.

## Il Kernel

Il kernel è il nucleo del sistema operativo, esso può occuparsi anche dello **scheduling** e del **multitasking**.

Un kernel deve essere **affidabile e privo di bug** altrimenti comprometterebbe sia la sicurezza che l'efficienza della macchina.

## System-Call

Vi sono due modalità di esecuzione dei processi:

- **Modalità Kernel** (anche detta Sistema)
- **Modalità Utente** (anche detta User)

Le operazioni di I/O si possono fare solo in modalità sistema e per accedere alla modalità sistema bisogna fare una **System call**.

Una system call può essere effettuata ad esempio dagli interrupt o dalle trap.

Ci possono essere quattro tipi di system call che agiscono su file:

- [Open](#) (apre un file)
- [Write](#) (scrive su un file)
- [Close](#) (chiude un file)
- [Read](#) (legge da un file)

## Interrupt e Trap

Le interrupt sono generalmente segnali dall'esterno (ad esempio un tasto che viene premuto sulla tastiera o una periferica che viene inserita nel computer).

Le trap sono simili ma avvengono in modo sincrono (a run-time), ad esempio una divisione per 0 o quando si prova ad andare a scrivere in parti di memoria su cui non siamo autorizzati.

Sugli Intel [x86](#) abbiamo 4 **modalità di privilegi**: ring 0, 1, 2 e 3.

Ring 0 è il sistema e Ring 3 è l'utente.

In un sistema ci possono essere **più interrupt handler ognuno destinato a diversi interrupt**, magari con diverse priorità; ad esempio un interrupt per la tastiera, uno per le periferiche, e così via.

Quando arriva un interrupt si passa in modalità privilegiata, la CPU smette di fare quello che stava facendo e si concentra sull'interruzione.

L'interrupt handler non deve andare a modificare i registri della CPU in modo da non creare problemi al programma che si stava eseguendo prima dell'interrupt.

## Driver

I **driver** sono software che uniformano l'interfaccia tra i diversi tipi di hardware e software; altrimenti per ogni tipo di CPU, per esempio, dovremmo avere un sistema operativo in grado di farla funzionare bene.

I driver girano all'interno del kernel.

## Unix

Nasce negli anni sessanta come concorrente di Multix. E' un sistema **scritto in C** e assembler ed è **semplice e potente**.

Il codice sorgente è accessibile da tutti ed per un periodo è stato fornito alle università.

Altri sistemi Unix-Like sono: FreeBDS, NetBDS e xv6, il quale è un sistema sviluppato dal MIT.

## (3)Lezione 6 (29/09/2020)

### Shell e Terminali

#### Shell

La Shell è un **interprete di comandi** che permette, grazie alla sua interfaccia, di **sfruttare le capacità del kernel**.

Un esempio di shell open-source comunemente utilizzata è la **Bash** (Bourne-Again Shell).

Su di essa vengono eseguite svariate istruzioni, un esempio è l'istruzione *man* che permette di visitare una pagine del manuale.

ex. *man man* mi permette di leggere le istruzioni sull'utilizzo del manuale

Per interagire con la shell viene utilizzato un **terminale**.

#### Terminale

Un terminale è un dispositivo hardware, elettronico o elettromeccanico, che viene usato per inserire dati in ingresso ad un calcolatore e riceverli in uscita per la loro visualizzazione.

#### Storia del Terminale

I primi terminali nascono nel 19esimo secolo, in relazione ai vecchi telegrafi, chiamati **Teletype**, da cui deriva proprio il nome di terminale, e soprattutto la sua abbreviazione (**tty**). Negli anni '60 e '70 del 20esimo secolo, in seguito all'avvento del computer e del sistema operativo **Unix**, i terminali si trasformano in un **file**, con ad esso collegati vari **file speciali**. I terminali in Linux sono **Virtual Console**, ovvero **terminali virtuali (tty)** che interagiscono attraverso la tastiera e lo schermo.

#### Pseudo-Terminali (pty)

All'interno degli ambienti desktop (interfacce grafiche) si può utilizzare un **emulatore di terminale**; esso funziona grazie ad un collegamento tra un terminale virtuale ed un'istanza della shell.

Questo collegamento tra tty (terminale) e shell avviene grazie ad un **pty o pseudo-terminale**.

I due dispositivi vengono posti in una relazione di tipo **Master/Slave**, dove:

- Il master è il nuovo terminale virtuale;
- Lo slave è la shell, che interagisce con il terminale virtuale e fa sembrare all'utente di interagire con un vero e proprio terminale.

## File Descriptor

**E' un numero intero non negativo che corrisponde ad un file aperto da un processo.**

Il processo accede quindi al file aperto utilizzando il file descriptor.

Ogni processo usa, di base, 3 file descriptor:

- Standard Input (Stdin, cin)
- Standard Output (stdout, cout)
- Standard Error (stderr, cerr)

Una shell interattiva permette il funzionamento del terminale leggendo da standard input e scrivendo su standard output, che corrispondono a `/dev/tty`.

## Directory Principali

/	Radice
<code>/bin e /sbin</code>	Comandi essenziali e di amministrazione
<code>/boot</code>	file del boot di sistema
<code>/etc</code>	file di configurazione di sistema
<code>/home e /root</code>	home degli utenti (non root) e root
<code>/lib*</code>	librerie
<code>/media e /mnt</code>	mount-pointer per media rimovibili e FS
<code>/proc e /sys</code>	file system virtuali che danno un interfaccia grafica alle strutture dati del kernel
<code>/tmp</code>	file temporanei, spesso un ramdisk nei sistemi moderni
<code>/usr</code>	gerarchia secondaria, che può essere condivisa in sola lettura da più host

# (4)Lezione 7 (05/10/2020)

## Cosa fa la Shell

- in modalità iterativa, stampa un **prompt**, ovvero una sequenza di caratteri che indica che è in attesa di comandi.  
ex. `llc@ubuntu:~/Scrivania$`  
Posso anche cambiare questo prompt per ottenerne uno più informativo, come ad esempio il *liquid prompt*.
- legge l'input lo spezza in **token**, ovvero parole e operatori.
- espande gli **alias**, ovvero un comando di shell che permette di definire altri comandi.
- fa il **parsing** (analizza i token) in comandi semplici e composti
- esegue varie **espansioni** (`{}` ~ \$ \*?[]).

- esegue le eventuali **redirezioni** dell'I/O.
- **esegue il “comando”**, tipicamente un eseguibile/script esterno (ma può essere una funzione o un comando built-in)
- tipicamente, ne aspetta la terminazione
- ricomincia

I comandi più importanti:

- ***man***: apre una pagina di manuale:  
ex. *man mkdir* apre la pagina di manuale che spiega il comando *mkdir*
- ***help*** spesso, usare l'opzione *-h* o *--help*; essi, digitati dopo un comando, ne spiegano il funzionamento;  
ex. *cat --help* spiega il funzionamento di *cat*
- ***type***: permette di vedere se un certo comando è built-in oppure è esterno.

## Escaping e Quoting

Alcuni caratteri hanno significati speciali (per esempio, lo spazio) e devono essere **messi tra virgolette o preceduti da un backslash** per essere utilizzati:

- i **singoli apici** permettono l'inserimento di qualsiasi carattere, a parte gli apici stessi
- le **doppie virgolette** trattano in maniera speciale \$ ' e \
- la forma \$'...’ permette di espandere i \... alla ANSI-C;  
per esempio: *echo \$'ciao\nmondo\x21'* diventa *ciao<newline>mondo!*  
(\n→ <newline>, \x21→!)

## Tab-completion e History

- **tab-completion**: completa i comandi che sto scrivendo  
ex. cd /Scriv + tab → cd /Scrivania  
N.B. per inserire un tab devo usare ctrl+V tab
- **freccia su e giù**: scorrono la cronologia dei comandi;
- ***history***: mostra la cronologia dei comandi;
- **!!** è una stringa che al momento dell'esecuzione del comando viene rimpiazzata dal comando precedente, esempio:

```
some_commands
echo !!
```

E' l'equivalente di scrivere:

```
echo some_commands
```

- ***!n***: è una stringa che al momento dell'esecuzione del comando viene rimpiazzata dal comando numero ***n*** della cronologia.
- ***!str*** è una stringa che al momento dell'esecuzione del comando viene rimpiazzata dal comando ***str*** della cronologia.
- ***ctrl+R* (o Break)** ferma l'esecuzione del comando.

## Variabili

- **creare/aggiornare:** `name=value`.  
attenzione: senza spazi, infatti se scrivo `A = 10`, esso cerca di invocare A, passando come argomenti “=” e “10”.
- **leggere:** `$varname` o  `${varname}`  
la seconda forma è necessaria per accedere agli array, per esempio:  
 `${BASH_VERSINFO[*]}, ${BASH_VERSINFO[0]}, ${BASH_VERSINFO[1]}, ...`
  - sono possibili varie **espansioni/sostituzioni**, per esempio:  `${varname :value}`  `${varname :=value}`  `${varname /pattern /string}`  `${varname :offset}` e  `${varname :offset :len}`  `${#varname} ...`
- **\$\$** corrisponde al **PID** (numero che indica univocamente un processo) della bash.

## Variabili d'ambiente

- per specificare una **variabile d'ambiente** devo usare il comando:  
`export name[=value]`
- le variabili d'ambiente sono usate per specificare impostazioni; per esempio:
  - **PATH** specifica alla shell dove cercare i comandi (quando il nome non contiene /)
  - **MANPAGER** indica a man quale programma utilizzare per visualizzare le pagine di manuale
  - **PS1** configura il prompt principale di bash
- ogni processo ha la sua copia delle variabili d'ambiente.

# (5)Lezione 10 (09/10/2020)

Comandi:

Espansioni:

Ogni volta che si scrive su una riga di comando e si preme il tasto invio, bash esegue diverse **elaborazioni** sul testo prima di eseguire il comando.

Abbiamo visto in un paio di casi come una sequenza di un solo carattere, per esempio “~”, può avere diversi significati per la shell.

Il processo che permette che questo succeda è chiamato **espansione**; con essa, si digita qualcosa che si espande in qualcos'altro prima che la shell agisca su di esso.

Per esempio con il comando **echo**, il quale dovrebbe stampare i suoi argomenti di testo sullo standard output, se gli viene passato il **carattere ~**, lo espande e ci stampa in output il nome della directory di home.

Questo perché la tilde si espande nel comando “dimmi il nome della directory home”.

## Altri esempi di espansioni:

\$var, \${var}, . . .	Il dollaro viene usato per espandere le variabili.
\$cmd e ‘cmd’	Redirigono input e output per un comando che legge un file
\$((expr))	
<(cmd) e >(cmd)	

Le espansioni che non sono avvenute all'interno di doppie virgolette (e contengono spazi, tab o newline) vengono spezzate in parole separate.

**Il pattern matching** è l'azione di controllo della presenza di un certo motivo ([pattern](#)) all'interno di un oggetto composito.

Il pattern-matching sui nomi di file con **wildcard** (\* ? [...]): i nomi che iniziano con punto sono “nascosti”, per convenzione

**\*\* espande ricorsivamente nelle directory**, se l'opzione globstar è abilitata (di default non lo è)

**Il globstar** corrisponde al pattern jolly e restituisce i nomi di file e directory che corrispondono, quindi sostituisce il pattern jolly nel comando con gli elementi corrispondenti.

## Redirezioni:

La **redirezione** è la deviazione dei canali standard (*standard input*, *standard output* e *standard error*) di un dato comando verso destinazioni (o da sorgenti, nel caso dello *standard input*) che sono diverse da quelle predefinite.



Scorciatoia: &>fname equivalente a >fname 2>&1

Due file speciali che possono tornare comodi:

`/dev/null` serve per fare l'output senza vedere gli errori  
`/dev/zero`

per approfondimenti vedere `NULL(4)`, così come il comando `yes`

## Scripting:

Lo scripting indica l'esecuzione di programmi:

- **Semplici:** una sequenza di parole, separate da blank
  - **Pipeline:** sequenza di comandi, separati da | o |&
  - **Liste:** sequenze di pipeline, separati da ;, &, &&, o ||, optionalmente terminate da ;, &, o newline

- o possono essere racchiuse fra ( e ), o { e }, per applicare un'unica redirezione a tutti i comandi nella lista

Esempio:

```
echo mondo; echo pippo > file
```

Nel terminale:	<i>mondo</i>
Nel file:	<i>pippo</i>

## Exit status

L'Exit status è il valore che restituisce il processo quando termina:

- Ogni comando restituisce un exit status, che finisce in \$? .
  - o Si può uscire dalla bash con exit n
- Per convenzione, 0 OK, non-0 errore
- Nei contesti "booleani" 0 → True, non-0 → False.
- && e || possono essere usati per comporre pipelines

## Segnali:

Sono interruzioni software a un processo, che notificano un evento asincrono;

I comandi possono essere tipicamente interrotti premendo Ctrl+C, quando un terminale è in modalità canonica, coi settaggi di default, Ctrl+C corrisponde a inviare al comando un segnale **SIGINT (2)** che tecnicamente viene inviato al gruppo di processi in foreground.

Per inviare segnali, si può usare il comando kill che di default invia il segnale di terminazione **SIGTERM (15)** il quale termina il processo fra i vari segnali.

**SIGKILL (9)** termina il processo e non può essere catturato, bloccato o ignorato.

L'exit-status di un processo terminato per un segnale s è (s + 128).

**SIGSEGV (11)**, che corrisponde al segmentation fault, usa 139.

Per visualizzare l'elenco dei segnali usare: *Signac*

## Funzioni

La bash è un linguaggio di programmazione (un po' strano) per cui si possono definire funzioni tramite la parola chiave **function**:

**function name { cmd-list }**

Dando il nome della funzione e tra graffe, una serie di comandi.

Gli argomenti sono \$1,\$2,...

per vedere il numero di argomenti \$#.

Il comando return può essere usato come in C/Python.

local può essere usato per dichiarare variabili locali.

## Condizioni

Come confrontare tra loro dei valori tramite comandi:

- test
- [

questa sintassi permette di implementare un contatore di espressioni fuori dalla shell.

- [ 1 -eq 2 ] si chiede se 1 è uguale a 2
- [ 1 = 2 ]
- [ 1 == 2 ]
- [ -f /etc/passwd ] verificare se un certo file esiste
- [ -w /etc/passwd ] verificare se un certo file è scrivibile
- test 1 -ne 2

versione moderna (bash built-in): [[ ... ]]

si possono usare < e > per confrontare stringhe, attenzione ai tipi:

- [ 10 < 2 ] vs [[ 10 < 2 ]] vs [[ 10 -lt 2 ]]

le stringhe vuote non sono un problema.

## Costruttori condizionali e cicli

### IF:

```
if
if tst-commands; then
    consequent-commands;
[elif more-tst-commands; then
    more-consequents;]
[else alternate-consequents;]
fi
```

### FOR:

- for name in words; do commands; done
- for (( expr1 ; expr2 ; expr3 )) ; do commands ; done

### WHILE:

```
while tst-commands; do consequent-commands; done
```

## Introduzioni al Job Control

Meccanismo che permette di gestire più lavori con uno stesso terminale, sospendendo/riprendendo l'esecuzione di gruppi di processi, chiamati anche job :

- importante coi terminali "veri", meno con gli emulatori
- i job sono raggruppati in sessioni, una per terminale
- per ogni pipeline che inserite nella shell, viene creato un job

## Sessioni

Un singolo comando può far partire più processi, ad esempio usando la pipe “|” possiamo concatenare due processi in modo da redirigere lo standard output del primo sullo standard input del secondo. L'insieme dei processi che sono fatti partire da un unico comando viene chiamato *process group* o *job*.

Una sessione invece è un insieme più grande di processi, che comprende di solito tutti i processi fatti partire da un unico login. Il primo di questi viene chiamato *session leader*. Quando il processo che controlla il terminale (nel caso precedente è la shell) viene terminato manda a tutti gli appartenenti alla sessione un segnale (SIGHUP) che, se non è ignorato, causa la morte dei processi.

Normalmente si ha un solo session leader per login, e questo è la shell. Il processo che ha l'uso del terminale in un dato momento si dice essere in *foreground*, mentre gli altri sono in *background*.

Quando apriamo un terminale c'è una sessione associata e dentro la sessione possiamo avere tanti gruppi di processi, gruppi di processi e job sono sinonimi.

Ogni volta che lanciamo un comando dalla shell, quello che succede è che viene creato un gruppo di processi.

Quando noi diamo il Ctrl+C, esso arriva al gruppo di processi che sta in primo piano.

Quando apriamo l'emulatore di terminale, lui:

- crea un pty e gestisce la parte master
- crea un nuovo processo *p*, che:
  - crea una nuova sessione; *p* diventa **session-leader**
  - apre lo slave, che diventa il terminale di controllo della sessione; *p* è detto anche **controlling process**
  - esegue la shell

quando chiudiamo la finestra corrispondente, viene “disconnesso” :

- il kernel manda il segnale SIGHUP al session-leader, la shell
- la shell, a sua volta, lo manda a tutti i job che gestisce
- di default, SIGHUP termina i processi

ogni terminale può avere:

- un job in foreground che può: leggere dal terminale, ricevere segnali se l'utente usa ^C, ^Z, ...
- tanti job in background: una pipeline terminata da & viene eseguita in background; \$! è il PID del job

^Z oppure *bg [job\_spec]* mandano un processo da foreground a background invece con *fg [job\_spec]* si fa l'operazione inversa.

Il comando **jobs** elenca i job attivi.

Con alcuni comandi built-in (es. kill, wait, disown) i job possono essere identificati con %n; l'ultimo fermato quando era in foreground, o fatto direttamente partire in background, è %%

## (6)Lezione 14 (15/10/2020)

### File e Processi

#### System call

**Un processo non può interagire direttamente con l'HW.**

Può utilizzare una chiamata di sistema = system call, AKA syscall “chiamata controllata” all'interno del kernel.

Per effettuare una syscall è necessario usare **assembly**, ma la libreria C offre delle funzioni wrapper.

#### Funzioni Wrapper

Queste funzioni (chiamabili da) C **preparano argomenti/#-syscall secondo le convenzioni del sistema**.

A seconda del S.O.:

- Linux: parametri caricati in specifici registri
- Xv6: parametri sullo stack (utente)
- in entrambi: # syscall in EAX eseguono una trap (Linux x86 = INT 0x80, Xv6 = INT 0x40)

Il kernel:

- controlla la validità di #/argomenti
- esegue la richiesta e scrive il risultato in un registro
- ritorna alla modalità utente, tramite un'istruzione speciale (**IRET**)

La funzione wrapper controlla il risultato in caso di errore imposta *errno* e restituisce un codice di errore, tipicamente -1 altrimenti, restituisce il risultato al chiamante

**Una syscall è molto più costosa di una semplice chiamata a funzione.**

#### Esito

Controllate sempre il valore di ritorno di funzioni/syscall

In caso di errore, *errno* indica la ragione del fallimento potete considerarla una specie di “variabile globale” intera ogni thread ha la sua in caso di successo, non è detto che venga azzerata/modificata

Vedete anche `errno(3)`, `perror(3)` e `strerror(3)`

## Tipo di Dato

Varie informazioni, per esempio i PID, vengono memorizzate in `int/long/. . .` a seconda del sistema

Scrivete codice portabile usando gli alias definiti dagli header; per esempio, `getpid` vi restituisce un `pid_t`

Questo crea qualche problema quando si vogliono stampare con `printf` e funzioni simili

Un approccio è usare un cast a un tipo sicuramente più grande

- Per esempio, `long` o `long long`
- Dal C99 in poi è possibile usare `intmax_t` (e `uintmax_t`), stampandoli con `%jd` e `%ju`, definiti in `stdint.h`

## File Descriptor

Tutto l'I/O avviene tramite i file descriptors, interi non negativi che identificano un file aperto. "file" in senso abbastanza generale, non solo file regolari, anche connessioni di rete, pipe, dispositivi, . . . ogni processo ha i suoi.

Tre sono "speciali" (solo per convenzione)

- 0 `STDIN_FILENO` ↔ `stdin/cin`
- 1 `STDOUT_FILENO` ↔ `stdout/cout`
- 2 `STDERR_FILENO` ↔ `stderr/cerr`

## Open

[`int open\(const char \*pathname, int flags\[, mode\_t mode\]\);`](#)

Flags bitmask che indica se si vuole aprire ad esempio in lettura o scrittura;

I flag si possono recuperare e (qualcuno) modificare con `fcntl(2)`

La maggior parte dei flag riguarda il file lettura/scrittura `O_APPEND` `O_NONBLOCK`, ma esistono (per ora uno: `FD_CLOEXEC`) flag associati al fd mode: è utilizzato solo quando viene creato un file ovvero, flags contiene `O_CREAT` o `O_TMPFILE`; può essere comodo specificarlo in ottale.

Viene sempre restituito il più piccolo file descriptor disponibile o `-1`, in caso di errore.

## Esercizio 1

Scrivete un programma che crea un file (regolare) chiamato pippo

Che sia leggibile solo dal proprietario (non scrivibile da nessuno, proprietario compreso)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
    int err1 = open("pippo", O_CREAT, 0400);

    if(err1 == -1)
        printf("errore1\n");

    int err2 = creat("pippococa", 0400);
    if(err2 == -1)
        printf("errore2\n");
}
```

Cosa succede se il file esiste già? Non mi permette di sovrascriverlo.

La syscall va a buon fine? La open va a buon fine e ritorna 3, la creat da errore (ritorna -1)

Il contenuto del file esistente viene preservato? Sí

Potete fare in modo che un file già esistente non venga ri-creato per errore? Sí, mettendo i permessi solo di lettura.

Modificate quel file con un editor di testo (potete farlo? come?) No, si può solo aprirlo in lettura.

## (7)Lezione 16 (19/10/2020)

Makefile, Valgrind e Address Sanitizer

### Makefile

Alternativa a:      `gcc *`      che compila tutto.

*Makefile* fa in modo che se cambio un file, in fase di compilazione **compilo solo i file che dipendono dal file cambiato**.

*esempio di funzionamento:*

*pippo: pippo.o* *da il nome pippo alla serie di comandi*  
*stabilisce che pippo ha priorità su pippo.o*

*pippo.o pippo.c* *stabilisce che pippo.o ha priorità su pippo.c*

*make* *avvia la compilazione*

*N.B.:* in questo processo non posso nominare i comandi con nomi di file che esistono già; per esempio, non posso usare il nome *clean* perché esiste già; se voglio usarlo a tutti i costi devo usare il comando *.phony clean*

## Valgrind

Comando che permette di compilare un programma **controllando** sulla **gestione della memoria dinamica** di esso.

Segnala quindi errori sulle funzioni *malloc()*, *realloc()*, *calloc()* e *free()*.

Segnala anche eventuali *memory leak*.

*Specifiche:*

(<http://valgrind.org>)

Installazione: apt install valgrind

Quick-start: compilate con i simboli di debug (-ggdb),  
poi valgrind [-leak-check=full]

## Address Sanitizer

Segnala molti errori che non vengono riconosciuti dal compilatore.

Rispetto a valgrind è molto più efficace ed **efficiente**.

Per utilizzarlo bisogna compilare con: *-fsanitize=address -ggdb*

## Debugger (*gdb*)

Ha varie funzionalità:

- quando un programma si schianta, indica dove esso si è fermato.
- permette di bloccare il programma in determinati punti (*breakpoints*) e di scorrerlo lentamente, osservando le variabili presenti in esso.

Comandi:

<b>gdb nomefile</b>	lancia il debugger
<b>break nomefile numerodilinea</b>	inserisce un breakpoint
<b>print variabile</b>	stampa una variabile in quel punto
<b>s</b>	va avanti di un passo
<b>n</b>	va avanti di un istruzione (se è una funzione la esegue)
<b>watch variabile</b>	tiene d'occhio una variabile

## System Call *read*, *write* e *close*

- *ssize\_t read(int fd, void \*buf, size\_t count);*
- *ssize\_t write(int fd, const void \*buf, size\_t count);*
- *int close(int fd);*  
Può fallire (per esempio quando il *fd* è già chiuso o per errori specifici di alcuni *FS*);  
In ogni caso, il file descriptor viene rilasciato → ritentare la chiusura può portare a oscure race-condition in programmi multithread

N.B.: *read* e *write* possono restituire un valore più piccolo di *count* (ma > 0); questo non è un errore. Per esempio, da terminale (di default) si legge solo una linea. 0 indica EOF (end of file).

**strace** comando che traccia le syscall eseguite da un programma

Scrivete un “cat dei poveri”, ovvero un programma che se invocato senza parametri legge da standard-input:

Altrimenti, dal/dai file specificati da riga di comando scrivendo tutto quello che riesce a leggere su standard-output.

Debugging:

Per i casi più complessi serve *gdb*, ma spesso basta *strace*

## Esercizio 2

La nostra Soluzione

Cosa succede se specificate: file che non esistono? *fopen* ritorna null

File che esistono ma di cui non avete il permesso di lettura? *fopen* ritorna null

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char ** argv)
{
    // se non ci sono argomenti
    if(argc <= 1)
    {
        // legge da stdin
        const int N = 1024;
        char buffer[N];

        printf( "Scrivi qualcosa: ");
        scanf("%s", buffer);

        printf( "\nHai scritto: %s \n", buffer);
    }
}
```

```

// altrimenti
else
{
    // dal/dai file specificati da riga di comando
    // scrivendo tutto quello che riesce a leggere su standard-output

    for(int i=1; i<argc; i++)
    {
        FILE *f = (fopen(argv[i], "r"));

        // controllo che non ritorni un errore
        if(f == NULL)
        {
            printf("Errore, il file non esiste\n");
            exit(1);
        }

        char s[1024];
        while(!feof(f))
        {
            *s = 0;
            fgets(s, sizeof s, f);
            printf("%s\n", s);
        }

        fclose(f);
    }
}

```

### Soluzione del Prof.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

static const int DEFAULT_BUFFER_SIZE = 1024;
static const int MAX_SIZE = 8192;

static void my_close(int fd)
{
    if (close(fd)) perror("my_close");
}

static int cat_fd(const int fd, const char * const pathname);

static int cat_file(const char *const pathname)
{
    const int fd = open(pathname, O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "Impossible to open file: %s (%s)\n",
                pathname, strerror(errno));
        return -1;
    }
    return cat_fd(fd, pathname);
}

```

```

static int write_all(char *buffer, ssize_t count)
{
    ssize_t n_total_written = 0;
    ssize_t offset = 0;
    while (n_total_written < count) {
        const ssize_t n_written = write(STDOUT_FILENO, buffer +
            offset, count - n_total_written);
        if (n_written < 0) {
            fprintf(stderr, "Cannot write to stdout.\n");
            return -1;
        }
        n_total_written += n_written;
        offset += n_written;
    }
    return count;
}

static off_t calculate_buffer_size(int fd)
{
    struct stat st;
    if (fstat(fd, &st) < 0)
        return -1;
    off_t buf_size = (st.st_mode&S_IFMT)==S_IFREG ? st.st_size :
        DEFAULT_BUFFER_SIZE;
    if (buf_size > MAX_SIZE)
        buf_size = MAX_SIZE;
    return buf_size;
}

static int cat_fd(const int fd, const char * const pathname)
{
    off_t buf_size = calculate_buffer_size(fd);
    if (buf_size < 0) {
        fprintf(stderr, "Cannot read metadata of %s (%s)\n",
            pathname, strerror(errno));
        my_close(fd);
        return -1;
    }
    char * const buffer = malloc(buf_size);
    if (buffer == NULL) {
        fprintf(stderr, "Cannot allocate the buffer.\n");
        my_close(fd);
        return -1;
    }
    for(;;) {
        const ssize_t n_read = read(fd, buffer, buf_size);
        if (n_read < 0) {
            if (pathname)
                fprintf(stderr, "Impossible to read from
file: %s\n", pathname);
            else
                fprintf(stderr, "Cannot read from
stdin?!?!?\n");
        }
        cleanup_and_fail:
            free(buffer);
            my_close(fd);
            return -1;
        }
        if (n_read == 0)
            break;
        if (write_all(buffer, n_read) < 0)
            goto cleanup_and_fail;
    }
    my_close(fd);
}

```

```

        free(buffer);
        printf("\n");
        return 0;
    }

int main(int argc, char *argv[])
{
    if (argc < 2)
        cat_fd(STDIN_FILENO, 0);
    else for (int i = 1; i < argc; ++i)
        cat_file(argv[i]);
}

```

## (8)Lezione 18 (23/10/2020)

### File offset

- A ogni **file aperto** è associato un **file offset / “pointer”** che indica in che posizione leggere / scrivere (all'interno del file)
  - usato per i file regolari
  - non ha senso andare avanti / indietro su un socket, dispositivo, etc
- Dopo la open l'offset è 0
- Viene spostato da *read()* e *write()*
  - Oppure da una syscall apposta: *off\_t lseek(int fd, off\_t offset, int whence)*; **La posizione e' calcolata aggiungendo offset** (che puo' assumere anche valori negativi) **a whence**.  
Dove whence può essere: SEEK\_SET, SEEK\_CUR o SEEK\_END  
**Che può essere: dall'inizio dalla posizione corrente dalla fine**
- Può essere spostato oltre la fine del file
  - questo NON cambia la dimensione del file
  - un file può avere “buchi” che corrispondono (logicamente) a byte a 0
    - → huge\_file / xxd ...

### Metadati di un file

I **metadati** sono quei dati che **descrivono altri dati**, in particolare *in riferimento ai documenti digitali*.

Per recuperare i metadati di un file:

- *int stat (const char \*pathname, struct stat \*statbuf);*  
**stat()** è una chiamata di sistema Unix che **restituisce dati utili sull'inode di un file**.
- *int lstat(const char \*pathname, struct stat \*statbuf);*
  - è identica a **stat()** tranne il fatto che quando **pathname** individua un link simbolico vengono acquisite le sue informazioni, piuttosto di quelle del file che è referenziato.

- `int fstat (int fd, struct stat *statbuf);`
  - riempie la `stat` con metadati

```
struct stat {
    dev_t      st_dev;      // major (12 bits) + minor (20 bits)
    ino_t      st_ino;      // Inode number
    mode_t     st_mode;     // File type and mode
    nlink_t    st_nlink;    // Number of hard links
    uid_t      st_uid;      // User ID of owner
    gid_t      st_gid;      // Group ID of owner
    dev_t      st_rdev;     // Device ID (if special file)
    off_t      st_size;     // Total size, in bytes
    blksize_t  st_blksize;  // Block size for filesystem I/O
    blkcnt_t   st_blocks;   // Number of 512B blocks allocated
    struct timespec  st_atim;    // Time of last access
    struct timespec  st_mtim;    // Time of last modification
    struct timespec  st_ctim;    // Time of last status change
}
```

## Processo di Compilazione e Linking

gcc/clang sono programmi che “pilotano” l’intero processo di compilazione lanciando: preprocessore, compilatore, assemblatore e linker

- **Compilatori e assemblatori producono file oggetto/rilocabili:**

`*.c + *.h — (cpp)+cc1 → *.s — as → *.o`

- Id, il **link editor** (AKA linker statico), **mette assieme file-oggetto/librerie, eseguendo rilocazione e risoluzione dei simboli:**

`*.o (+ *.a + *.so + script.id) — ld → a.out`

Tutto questo pilotato da un linguaggio, di cui non ci occupiamo.

### Differenze Linking Statico e Dinamico

Infine, il linking può essere:

- **statico**
  - Id fa tutto il lavoro, creando **programmi autocontenuti**
    - i “pezzi” di librerie necessari vengono copiati dentro al programma
    - gli eseguibili funzionano su “tutti” le macchine (con stesso HW/s.o.)
    - su **Xv6 è così** per semplicità
- **(statico +) dinamico**
  - Id (link editor) prepara l’eseguibile, **“annotando” le dipendenze esterne**
  - fra cui, il suo interprete, ovvero, il linker dinamico: ld.so
  - quest’ultimo **mette assieme i pezzi mancanti a tempo di esecuzione**
  - **default sui sistemi moderni**

- **fa risparmiare spazio**, sia su disco, sia in memoria
- fisica facilita l'aggiornamento (ma anche la “distruzione globale”)
- può essere problematico portare l'eseguibile da una macchina a un'altra

## Fasi della compilazione:

Sia gcc che g++ processano file di input attraverso uno o più dei seguenti passi:

1. preprocessing
  - a. rimozione dei commenti
  - b. interpretazioni di speciali direttive per il preprocessore denotate da "#" come: #include e #define
2. compilation
 

traduzione del codice sorgente ricevuto dal preprocessore in codice assembly
3. assembly
 

creazione del codice oggetto
4. linking
 

combinazione delle funzioni definite in altri file sorgenti o definite in librerie con la funzione main() per creare il file eseguibile.

## Cosa contiene un eseguibile?

Gli eseguibili, così come i file rilocabili (.o) e le librerie possono contenere:

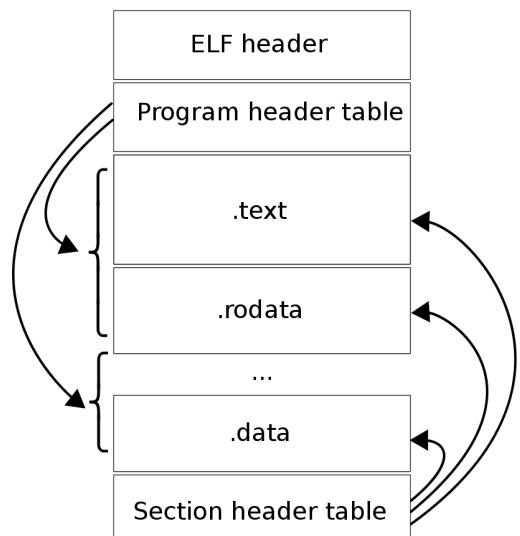
- **Codice macchina** → sezione .text
- **Dati e dati a sola lettura** → .data e .rodata
- **Metadati**
  - architettura (Intel/ARM/. . . , 32/64 bits, little/big endian, . . . )
  - entry-point; no, non è il main
  - quanto spazio riservare per variabili globali non inizializzate → .bss

In un eseguibile le sezioni vengono raggruppate in **segmenti**, i “pezzi” che vengono mappati in memoria dal s.o./linker-dinamico per l'esecuzione.

Attenzione: questi segmenti NON necessariamente corrispondono ai segmenti della CPU/MMU. Infatti, non è così in Linux, Windows e Xv6

## Executable and Linkable Format

- ELF header, obbligatorio, all'inizio → elf(5)
- Section header table, se presente ci indica dove trovare le informazioni per il linking; per esempio, le sezioni .text, .data, . . . , la tabella dei simboli, le informazioni di rilocazione, . . .
- **Program header table**, se presente, descrive i **segmenti**, che contribuiscono a costruire lo **spazio di indirizzamento** di un processo



## (9)Lezione 20 (30/10/20)

Lezione pratica di esercizi, nessun appunto.

## (10)Lezione 22.1 (30/10/2020)

### Spazi di indirizzamento

I processi usano indirizzi logici/virtuali che vengono tradotti in indirizzi fisici dalla **MMU**, in questo processo entrano in gioco:

- **Segmentazione**
- **Paginazione**

Per far girare i programmi non ci basta mappare codice e dati da un ELF, servono anche :

- **Stack** -> variabili locali/temporanee, parametri, indirizzi di ritorno; per ragioni “storiche”, cresce verso indirizzi decrescenti
- **Heap** -> memoria dinamica
- **Kernel** -> non accessibile in modalità utente

Il codice di programmi e, grazie al linking-dinamico, le librerie possono essere condivisi (e mappati a indirizzi diversi) in processi diversi.

Un thread e un processo sono due cose differenti:

- Un Thread è un flusso di esecuzione.
- Fork crea un processo:

`pid_t getpid(void)`

`pid_t getppid(void)`

I processi formano un albero, con radice *init* (PID=1); su Ubuntu in realtà è *systemd*, ma lo chiameremo lo stesso *init*.

### PID

I PID vengono riutilizzati, quindi “identificano” i processi **solo in uno specifico momento**.

Il kernel espone le informazioni tramite lo **pseudo-filesystem** `/proc`:

- una directory per ogni processo
- vari file, la maggior parte a sola lettura
- per esempio, *maps* mostra lo spazio di indirizzamento

Ogni processo ha una:

- **directory root** (radice), che viene usata per tutti i percorsi assoluti (=che iniziano con `/`) non è necessariamente la root del file-system, ne parleremo più avanti bisogna avere dei privilegi per cambiarla
- directory di lavoro, che viene usata per tutti i percorsi relativi; vedere `getcwd(2)` o, estensione GNU,
- `get_current_dir_name(3)` restituiscono quella corrente mentre `chdir(2)` e `fchdir(2)`, la modificano

Le syscall principali per gestire i processi sono solo quattro:

- **fork**, crea un nuovo processo
- **\_exit** (standard C: `exit`), termina il processo chiamante quando non ci saranno ambiguità, diremo semplicemente `exit`
- **wait**, aspetta la terminazione di un processo figlio non è vero al 100%, ne parliamo dopo
- **execve**, esegue un nuovo programma nel processo chiamante, sostituendo l'intero spazio di indirizzamento; spesso invocata dopo `fork`; pensate, ad esempio, alla shell
  - chiamata semplicemente `exec` o `exec*`, per indicare l'intera famiglia di funzioni per eseguire programmi

## Fork

`pid_t fork(void)` “clona” un processo

`init` è speciale.

In Linux esistono `clone/[__]clone2/clone3`, ma non ne parliamo

Fork, creando un nuovo processo, detto “**figlio**” del processo chiamante, esso è una copia quasi identica del processo chiamante:

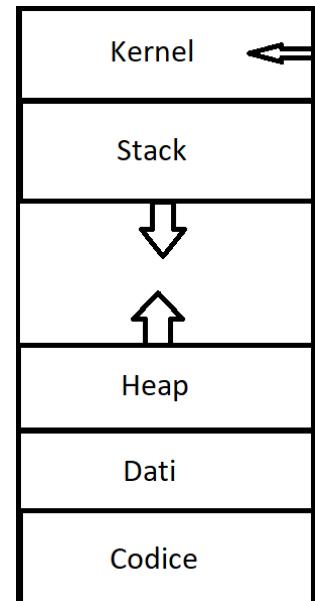
- **cambiano** chiaramente **PID, PPID**; però
- i **FD** (flag compresi) vengono **duplicati**
- **l'intero spazio di indirizzamento viene copiato**
  - nei sistemi moderni, copia logica grazie al copy-on-write
    - in passato, “trucchi sporchi” tipo `vfork` per evitare la copia
  - in caso di successo, **ritorna due volte**, restituendo
    - il PID del figlio al padre
    - 0 al figlio

# (11)Lezione 22.2 (30/10/2020)

Cosa ci serve per far “girare” i programmi?

- mappare **codice e dati**, da un [ELF](#).
- **stack** — variabili locali/temporanee, parametri, indirizzi di ritorno;  
Per ragioni “storiche”, cresce verso indirizzi decrescenti:  
(al suo crescere gli indirizzi diminuiscono)
- **heap** — memoria dinamica: *malloc*, *free*, *realloc*, *calloc*; non serve davvero un altro segmento, può crescere il segmento dati.
- il **kernel**, non accessibile in modalità utente

Il codice di programmi è, grazie al linking-dinamico, librerie può essere condiviso (e mappato a indirizzi diversi) in processi diversi.



Utilità della Memoria Virtuale

Se tre programmi fanno uso, ad esempio, della libreria C, essa può' essere mappata in memoria una sola volta ed essere utilizzata da tutti e tre simultaneamente.

## PID

`pid_t getpid(void)` prende il PID di un processo  
`pid_t getppid(void)` prende il PID del genitore (*parent*) di un processo

I processi formano un albero, con radice ***init*** (PID=1)  
(su Ubuntu in realtà è *systemd*, ma lo chiameremo lo stesso *init*)

N.B.: I PID vengono **riutilizzati**, quindi “identificano” i processi **solo in uno specifico momento**.

Il kernel espone le informazioni tramite lo pseudo-filesystem **/proc**  
ex.     `ls /proc`

- vi è una directory per ogni processo
- la maggior parte dei file sono a sola lettura, per esempio:  
`maps`        mostra lo spazio di indirizzamento
- per più informazioni vedere   `man proc(5)`

## Directory di lavoro

Ogni processo ha:

- una **directory root (radice)**:

- viene usata per tutti i percorsi **assoluti** (che iniziano con /)
- non è necessariamente la root del file-system
- bisogna avere dei **privilegi** per cambiarla
- una **directory di lavoro**, che viene usata per tutti i percorsi **relativi**;
  - `getcwd(2)` o, estensione GNU, `get_current_dir_name(3)` restituiscono quella corrente
  - `chdir(2)` e `fchdir(2)`, la modificano

## API per la gestione dei processi

Le syscall principali per gestire i processi sono solo quattro:

- **`fork`**, crea un nuovo processo
- **`_exit`** (standard C: `exit`), termina il processo chiamante; quando non ci saranno ambiguità, diremo semplicemente `exit`
- **`wait`**: tranne in casi particolari, aspetta la terminazione di un processo figlio
- **`execve`**, esegue un nuovo programma nel processo chiamante, sostituendo l'intero spazio di indirizzamento; spesso, invocata dopo `fork` (ad esempio, nella shell) chiamata semplicemente `exec` o `exec*`, per indicare l'intera famiglia di funzioni per eseguire programmi

## Fork

**`pid_t fork(void)`**      “clona” un processo

- **`init`** è speciale: è l'unico processo non creato da una `fork`
- in Linux esistono funzioni simili come `clone/clone2/clone3`, ma non sono `posix`
- il processo nuovo è detto “**figlio**” del processo chiamante
  - è la copia quasi identica del processo chiamante:
    - **cambiano** chiaramente: **PID**, PPID;
    - i **FD** (flag compresi) vengono duplicati e **rimangono invariati**
    - l'intero **spazio di indirizzamento viene copiato** nei sistemi moderni, copia logica grazie al *copy-on-write* in passato, “trucchi sporchi” tipo `vfork` per evitare la copia
- in caso di successo, **ritorna due volte**:
  - restituendo il PID del figlio al padre
  - restituendo 0 al figlio
  - Normalmente gdb continua a seguire solo uno dei due processi; scegliete quale con: `set follow-fork-mode [child | parent]`

# (12)Lezione 24 (03/11/2020)

## Exit

<code>void exit(int status)</code>	libreria C
<code>void _exit(int status)</code>	syscall (in Linux, exit_group(2))

La prima:

- **chiama le funzioni** registrate con `atexit(3)` e `on_exit(3)`
- **svuota i buffer** di I/O
- **elimina file temporanei** creati con `tmpfile(3)`

In entrambi i casi:

- **vengono chiuse/rilasciate le risorse del processo** (file descriptor, memory-mapping, System-V IPC objects, . . .)
- il processo termina con **exit-status** uguale a (`status & 0xff`) (dalla shell lo recuperate con `$?`)
- Ciascun figlio del processo corrente viene ereditato dal processo 1 (init, il padre di tutti i processi), sopravvivendo alla scomparsa del processo parent.

Restituire `n` dal main corrisponde a `exit(n);`

Lo standard C definisce le costanti `EXIT_SUCCESS` (=0) ed `EXIT_FAILURE` (=1).

## Wait

La funzione **wait()** sospende il processo corrente finché un figlio (child) termina o finché il processo corrente riceve un segnale di terminazione o un segnale che sia gestito da una funzione.

Quando un child termina il processo, senza che il parent abbia atteso la sua terminazione attraverso la funzione di **wait()**, allora il child assume lo stato di "zombie" ossia di processo "defunto".

Se il processo corrente esegue la funzione di **wait()**, mentre ci sono uno o piu' child in stato di zombie, allora la funzione ritorna immediatamente e ciascuna risorsa del child viene liberata.

<code>pid_t wait(int *wstatus)</code>	attende un "cambio di stato" in un figlio
---------------------------------------	---

Esso può essere determinato da:

- **terminazione**
- **stop/ripartenza, tramite segnali**

Se `wait` va a buon fine, allora `wstatus != 0`; inoltre:

- se ritorna `WIFEXITED(*wstatus)`, allora potete recuperare lo exit-status con `WEXITSTATUS(*wstatus)`
- se ritorna `WIFSIGNALED(*wstatus)`, allora potete recuperare il segnale con `WTERMSIG(*wstatus)`

In caso di successo `wait()` ritorna il process ID del child che termina.

Torna -1 in caso di errore e `errno` viene settato in modo appropriato.

## Zombie

**Un processo terminato, non aspettato dal padre, è uno zombie.**

Il sistema rilascia alcune risorse, ma deve continuare a memorizzare alcune informazioni, per esempio, PID ed exit-status.

Quando un processo termina, viene inviato al padre il segnale SIGCHLD:

- di default è ignorato
- può essere catturato per “aspettare” i figli; siccome i segnali (non *realtime*) non si accodano, potrebbero esserci più figli terminati per un singolo segnale.

## Exec

```
int execve(const char *pathname, char *const argv[], char *const envp[]);  
libc: int execl(const char *pathname, const char *arg, ... /* (char *) NULL */);  
int execvp(const char *file, const char *arg, ... /* (char *) NULL */);  
int execle(const char *pathname, const char *arg, ... /* (char *) NULL, (char *) NULL, char *const envp[] */);  
int execv(const char *pathname, char *const argv[]);  
int execvp(const char *file, char *const argv[]);  
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

**Exec sostituisce i segmenti codice e dati del processo correntemente in esecuzione nello stato di utente con quelli di un altro programma contenuto in un file eseguibile specificato, ma lascia invariati:**

- **PID, PPID**
- **file descriptor** (a meno di FD\_CLOEXEC)

Quando eseguiamo un programma il sistema (kernel+linker) prepara:

- codice, mappato r-x (.text)
- dati a sola lettura, mappati r-- (.rodata)
- dati, mappati rw- (.data, .bss e heap)
- stack, mappato rwxn (fondo allo stack (quindi, a indirizzi più grandi) il kernel copia: le variabili d'ambiente, gli argomenti della riga di comando e (altra "roba"))
- (il kernel, "invisibile")

Poi, in caso di linking dinamico fa la stessa cosa, ricorsivamente, per codice e dati delle librerie necessarie (+linking vero e proprio); quando viene creato un thread, viene anche allocato un nuovo stack.

Nei sistemi moderni, codice/dati vengono portati in RAM **on-demand**.

### **Se la syscall/funzione ritorna al chiamante, qualcosa è andato storto!**

Se l'eseguibile ha i bit set-user-ID/set-group-ID abilitato, potrebbero succedere comportamenti inaspettati (ne parleremo in un altro momento).

## (13)Lezione 26 (06/11/2020)

Dup

***int dup(int oldfd);***

- il nome è fuorviante, non duplica nel vero senso della parola, ma **crea un FD equivalente a oldfd**.
  - a livello kernel, vi è una struttura dati dove sono indiciati i FD; in essa, dopo la *dup*, **entrambi i FD "punteranno" allo stesso file aperto**.
  - **I FD potranno essere usati in modo intercambiabile: hanno gli stessi offset, flag, inode** (struttura dati che rappresenta un file nel file-system).
    - Per esempio è possibile effettuare una *lseek()* su un file descriptor e ritrovarsi la posizione modificata su entrambi i file descriptor.
- restituisce il più piccolo FD disponibile (come *open*): ex. se ho utilizzato solo gli indici 0,1,2 predefiniti mi restituirà il FD numero 3.
- non vengono copiati i flag del FD (al contrario di quelli del file)
 

*n.b. flag di FD e di file sono ben differenti: i flag di un file sono ad esempio come è stato aperto (lettura, scrittura..), l'FD ha un unico flag:*

  - FD\_CLOEXEC (ovvero l'unico flag del FD), non sarà attivo per quello nuovo; **This flag specifies that the file descriptor should be closed when an exec function is invoked**  
*per maggiori informazioni vedere *fcntl(2)**

***int dup2(int oldfd, int newfd);***

- come *dup*, ma usa *newfd* come indice del nuovo FD;
  - chiudendolo se necessario (solo le *oldfd* è valido)
- se *oldfd* è uguale a *newfd* non fa nulla

p.s. Linux offre anche una *dup3*

**N.B.** la *fork* utilizza *dup* poiché il figlio possederà una copia dei file descriptor del padre.

Relazione fra file descriptor e file aperti: POSIX

Ogni processo ha la sua tabella di file descriptor, all'interno della struttura dati **process control block**, una struttura dati più grossa che contiene tutte le informazioni per gestire un processo.

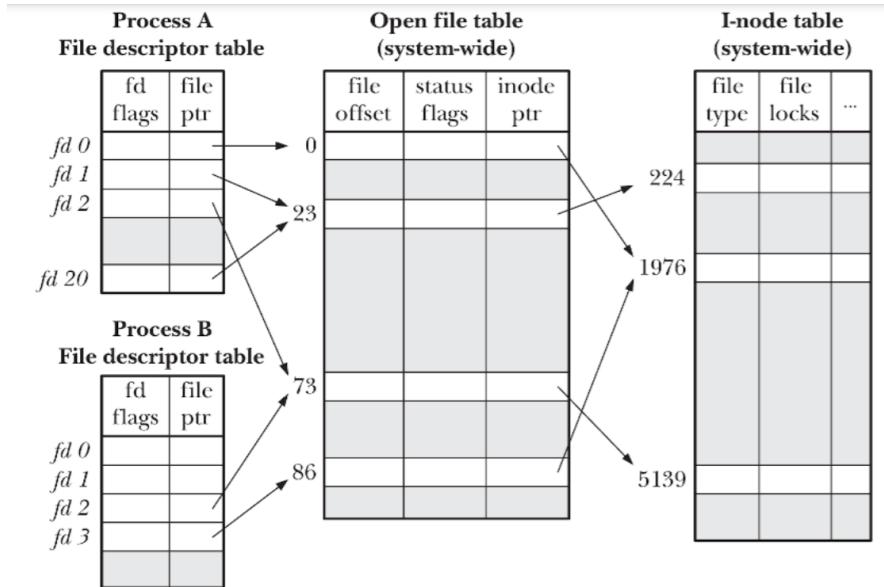


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Un FD punta ad il corrispettivo file aperto, che a sua volta punta all'inode corrispondente.

Tabella dei File Descriptor

Per ogni file descriptor vi sono associati:

- i **flag del FD**,
- il **puntatore al file aperto corrispondente**.

Tabella dei File Aperti

Per ogni file aperto vi sono associati:

- l'**offset**,
- i **flag del file**,
- il **puntatore all'inode** (nella tabella degli inode).

Nota bene:

- 1) Più FD possono “puntare” lo stesso file aperto, ad esempio dopo una *dup* o una *dup2*.
- 2) Più *open* possono aprire lo stesso file, quindi diversi FD possono avere lo stesso *inode*

- 3) FD di processi diversi possono puntare lo stesso file aperto, ad esempio dopo *fork*
    - anche FD diversi di processi diversi possono puntare lo stesso file aperto
  - 4) Quindi, è ben diverso aprire due volte lo stesso file e duplicare un FD
    - anche in assenza di [race conditions](#), basti pensare a *read*, *write* e *lseek*

## Pipe

*int pipe(int pipefd[2])*

- crea un **canale unidirezionale** (anonimo)
  - il canale è un *byte-stream*: non c'è un concetto di messaggio e/o corrispondenza fra numero di scritture e letture
  - tipicamente usato per far **comunicare processi**
  - La funzione pipe() crea una coppia di file descriptor, utilizzabili per una 'pipe'. I file descriptor vengono posti nell'array puntato da filedes.
    - filedes[0] e' il file descriptor utilizzabile per la lettura. filedes[1] e' il file descriptor utilizzabile per la scrittura
    - quello che viene scritto nella *pipe* finisce in un **buffer del kernel**
    - quando il buffer è pieno, le *write* vengono messe in attesa, o falliscono se flag O\_NONBLOCK è abilitato
      - se tutti i FD di lettura sono stati chiusi, *write* provoca SIGPIPE (è inutile scrivere su quel canale se nessuno potrà mai leggere)
    - quando il buffer è vuoto, le *read* vengono messe in attesa, o falliscono se flag O\_NONBLOCK è abilitato
      - se tutti i FD di scrittura sono stati chiusi, *read* restituisce 0, come EOF

p.s.: su man vedere pipe(2) e pipe(7)

Collegamento con le *pipe* della shell:

Per ogni pipe sulla riga di comando la bash chiama la syscall *pipe(..)* per creare un buffer all'interno del kernel;

Lo standard output della stringa alla sinistra della pipe nella shell viene rediretto alla stringa di destra della pipe:

**ex.** `ls -l | wc -l` lo standard output di `ls` viene usato come input di `wc`

(14)Lezione 28 (10/11/20)

Xv6

## Accensione della Macchina

Dai manuali Intel [i386, int19] scopriamo che:

- “Real-address mode is in effect after a signal on the RESET pin” e “mode appears as a high-speed 8086”: la Modalità Real-Address subentra dopo un segnale dato dal tasto RESET del calcolatore.
- “The starting location is 0FFFFFFF0H”: la locazione di partenza (BIOS) detto anche 232 – 16 è sull’indirizzo 0FFFFFFF0H.

A quell’indirizzo **deve essere presente il firmware**, in ROM o in EPROM, **che**

- 1. testa/inizializza l’HW**
- 2. cerca un dispositivo da cui poter fare il boot** ovvero, cerca nei suoi dischi un dispositivo a blocchi il cui primo settore contenga la “firma” 0x55 0xAA agli offset 510 e 511
- 3. carica il** primo settore del boot, detto **boot block**, **all’indirizzo fisico 0x7c00**
- 4. lo esegue**

Vedere anche [https://wiki.osdev.org/Boot\\_Sequence](https://wiki.osdev.org/Boot_Sequence)

Perché 0x7c00?

Correva l’anno 1981, il DOS 1.0 richiedeva almeno 32 KB.

Gli sviluppatori del BIOS per il PC 5150 di IBM hanno pensato di **lasciare più spazio possibile per il S.O.** all’interno di quei 32 KB.

Hanno scelto l’ultimo KB (sicuramente) disponibile: 0x7c00

- ovvero 1KB, (32768-1024)
- **512 byte per il boot sector**, altri **512 per dati/stack**
- dopo il boot quella zona non serve più; da allora, per compatibilità, continua a essere quello.

## Il Resto della Memoria

Sempre per ragioni storiche, in RAM:

- la parte 0-640KB, AKA 0-0xA0000, è detta **memoria convenzionale**.
- la parte 640KB-1MB, AKA 0xA0000-0x100000, è detta **memoria alta** (UMA: Upper Memory Area) ed è riservata alle periferiche per esempio, per la VGA:
  - 0xA0000 for EGA/VGA graphics modes (64 KB)
  - 0xB0000 for monochrome text mode (32 KB)
  - 0xB8000 for color text mode and CGA-compatible modes (32 KB)
- la memoria sopra al MB è detta **memoria estesa**

Nei PC più moderni i registri dell’APIC (“Advanced Programmable Interrupt Controller”) sono mappati a 0xFEE00xxx.

## QEMU (Emulatore)

Installiamo Xv6 attraverso un nostro fork.

In teoria, potremmo usare Xv6 su un PC “vero” ma:

- Dovremmo utilizzare un pc vecchio perché non ci sono i driver per gestire HW moderno.

- In ogni caso, è molto più facile e comodo usare un emulatore, principalmente per il debugging, che su una macchina fisica non è possibile.

**QEmu crea un “PC virtuale”,** su cui possiamo fare i test.

Per avviare il PC virtuale e Xv6 usiamo `./run` (che equivale a `make qemu-nox`).

Al momento, concentriamoci su:

- La variabile di ambiente **CPUS**, che ci permette di impostare il numero di processori che vogliamo (default: 1).
- Due file che vengono usati come dischi:
  - **xv6.img**, ovvero il **disco di avvio, boot+kernel**
  - **fs.img**, ovvero il **disco con il root file-system**

## Boot Block

In Xv6 i sorgenti del boot block, ovvero del primo settore del boot, sono:

- `kernel/bootasm.S`
- `kernel/bootmain.c`

Nota: fra tutti e due devono stare in 510 byte

Il makefile:

1. li mette assieme e **“firma”** il boot block
2. **crea l’immagine di un disco**, xv6.img, che contiene:
  - a. boot block, come primo blocco
  - b. kernel, dal secondo blocco in poi

## Cscope

E’ uno strumento per “navigare” sorgenti C/C++; scritto negli anni ’80 ai Bell Labs, al tempo del PDP-11.

Esso svolge svariate funzioni:

- permette di **cercare**:
  - i riferimenti a un simbolo
  - le funzioni chiamate-da o che-chiamano una funzione
  - e altre cose
- può essere usato **da terminale**: **cscope -d** — EOF (ctrl+d) per uscire
- o da vim , [http://cscope.sourceforge.net/cscope\\_vim\\_tutorial.html](http://cscope.sourceforge.net/cscope_vim_tutorial.html)
- vim -t simbolo o, da dentro vim:
  - :ta simbolo o :cs find g simbolo — vai alla definizione globale
  - :cs find s simbolo — trova tutti i riferimenti
  - ctrl+] — va alla definizione del simbolo sotto al cursore
  - ctrl+T — torna indietro

- `ctrl+\ c` — chiamate alla funzione sotto al cursore (calls) zione della **modalità protetta a 32 bit**
- `ctrl+\ d` — funzioni chiamate da quella sotto al cursore (called)
- `ctrl+\ s` — tutti i riferimenti al simbolo sotto al cursore (called)
- usate la variabile d'ambiente `CSCOPE_DB`, come suggerisce il makefile

## Bootasm.S

- 1) Partenza a 16 bit, modalità “reale”:
  - **Segmentata**, ma non come probabilmente vi aspettate:  
 $(segment << 4) + offset$   
 Trucco per indirizzare un MB con un processore a 16 bit (che quindi potrebbe indirizzare solo  $2^{16} \text{ bit} = 64kB$ )
- 2) Viene disabilitato il [gate A20](#), hack storico che possiamo ignorare
- 3) Prepara
  - **LGDT: caricamento della tabella dei segmenti**
    - Non ci interessano, ma non possiamo farne a meno:  
 indirizzo virtuale → **segmentazione** → lineare → **paginazione** → fisico
  - Prepariamo segmento nullo, codice e dati; per tutti: base 0 e limite  $+\infty$

→ [x86\\_translation\\_and\\_registers.pdf](#): Mostra come funziona la gestione della memoria su x86

- 4) Passaggio alla modalità protetta tramite **abilitazione di PE** (Portable Executable, *credo sia un eseguibile*) in **CR0** (*credo sia un registro*)
- 5) **Salto (detto “far”) su start32 che ci porta davvero ad una modalità 32 bit**
  - Shift di tre bit perché i tre bit meno significativi del selector sono TI (table indicator) e RPL (Requestor’s Privilege Level)
  - **Setup di uno stack** per il codice C (niente heap, niente libc)
- 6) **Salto al codice C, bootmain**

## (15)Lezione 29 (13/11/2020)

### Bootmain.c

- 1) Controllo che il kernel sia un ELF (**sanity check**)
- 2) **Caricamento dei segmenti** dell’ELF, da 1MB in su
  - Carica i segmenti parlando direttamente con il controller del disco IDE
- 3) **Divisione dello spazio di indirizzamento**; non stiamo ancora usando la paginazione, quindi parliamo di indirizzi fisici:
  - a) il kernel viene caricato all’inizio della memoria estesa EXTMEM (1MB) = `0x100000`
  - b) finite le inizializzazioni, lo spazio di indirizzamento sarà diviso in due:
    - i primi due giga per la parte utente

- gli altri due per il kernel (parte alta della memoria)
- c) il kernel sarà a KERNBASE (2G) + EXTMEM (1M) = 0x80100000
- 4) Si salta all'entry-point dell'ELF: `_start`  
il codice di `_start` è in `entry.S`

→ `kernel/bootmain.c`

## Paginazione su x86

Con 32 bit e pagine da 4kB (ne' grandi ne' piccole)

- se si usasse una sola tabella delle pagine, sarebbe da 4MB, perché:
  - $2^{12}$  per l'offset
  - $2^{20}$  per le pagine = 1M pagine
  - servono 4 byte per ogni pagina della tabella: 20 bit indirizzo + flags
  - $4B \times 1M = 4MB$
- **4MB contigui per ogni processo sono tanti**, quindi:
- **Soluzione**: tabelle multilivello;
  - In i386, tabelle a:
    - **2 livelli** (chiamate **directory** e **pagina**) con **pagine da 4kB**
    - 1 livello, con pagine da 4MB

Con i 64-bit, approccio analogo: pagine da 4kB, 2MB o 4MB, con tabelle da 1 a 4 livelli.

## entry.S

**Abilita la paginazione** inizialmente **con pagine da 4 MB** (molto grandi) durante la fase di boot, infatti il kernel viene mappato in una pagina da 4 MB da 1 MB in poi.

Dopo di chè si passa a pagine da 4 kB.

Anche XV6 possiede pagine multilivello

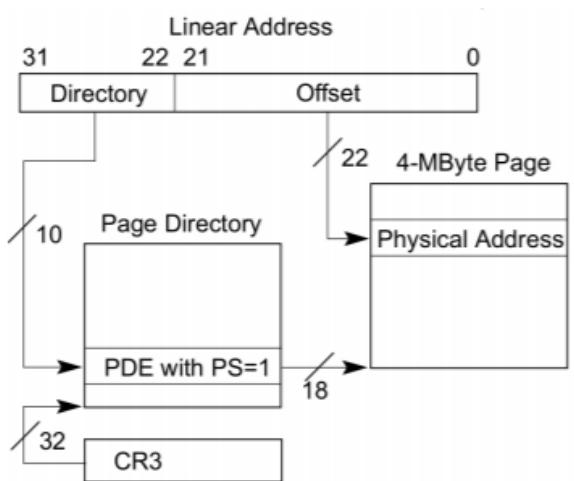
- con page-table `entrypgdir`, definita in `main.c`
- **i primi 4 MB sono mappati contemporaneamente a**
  - 0 e
  - KERNBASE = 2GB
- **prepara uno stack**
- **salta al main**, in C

→ `kernel/entry.S`

## (16)Lezione 31 (17/11/2020)

### Prima tabella utilizzata

La PD (Page Directory) è di 4k, allineata a 4k, e contiene 1024 PDE.



Ogni PDE controlla zone di 4MB, allineate a 4MB.  
Si indirizzano potenzialmente 40 bit.

## main.c

Viene avviato dopo una lunga riga di **inizializzazioni**, tra cui:

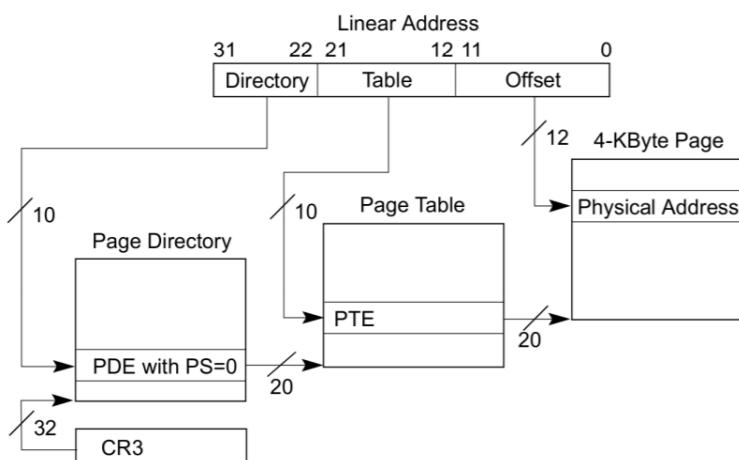
- *kinit1* — prepara l'allocatore iniziale di pagine
- *kvmalloc* — prepara la page-table del kernel

E:

- **crea init**
- **lancia lo scheduler**

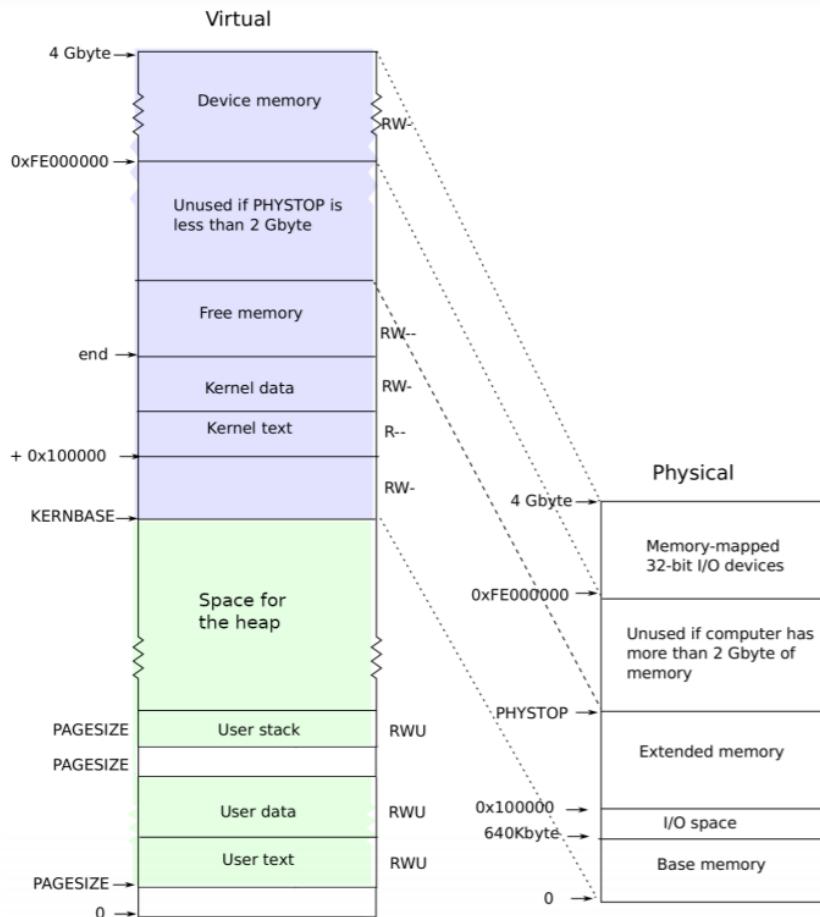
→*main.c*

## Paginazione su x86: pagine da 4kB



La PD è di 4k, allineata a 4k,  
e contiene 1024 PDE  
Ogni PDE punta a una PT,  
che controlla 4MB  
Le PG sono di 4k, allineate a  
4k, e contengono 1024 PTE  
Ogni PTE punta a un page  
frame (di 4k)

## Spazio di indirizzamento



## System call e libreria C

Syscall e funzioni disponibili → *user/user.h*

- 22 syscall e 12 di libreria

Per i casi più semplici basta un solo include: **#include "user/user.h"**

La “libc” è una micro-libreria, estremamente limitata e non standard

- **exit** non ha parametri
- **printf** usa direttamente i file descriptor ; questo evita inizializzazioni prima del main, che corrisponde all’entry-point dell’ELF
- per questa ragione, *main* deve uscire con *exit()*

Wrapper delle syscall: funzione che viene chiamata per processare la chiamata e mandarla al kernel → si trova in *user/usys.S*

Lato kernel, la syscall s è implementata da *sys\_s*

## Primo Programma

1. dentro a user creiamo *ciao.c*:

```
#include "user/user.h"
int main() {
    printf(1, "Ciao SETI!\n");
    exit();
}
```
2. nel Makefile aggiungiamo *\$U\_ciao* a *UPROGS*
3. *make* — crea il file *user/\_ciao*
  - a. che verrà “automagicamente” incluso in *fs.img*
  - b. è un ELF, ma girerà solo su Xv6, con Linux si schianta

## Interrupt e System Call

### Concetti generali.

La faccenda, su x86, è un po’ complicata: esistono 256 vettori di interrupt/trap

- l>IDT (Interrupt Descriptor Table) “punta” ai vari handler

**Nella risposta a un'interruzione bisogna** interrompere il programma e gestirla, ma anche **ricordarsi dove è stata interrotta l'esecuzione**, in modo simile a una CALL, ma:

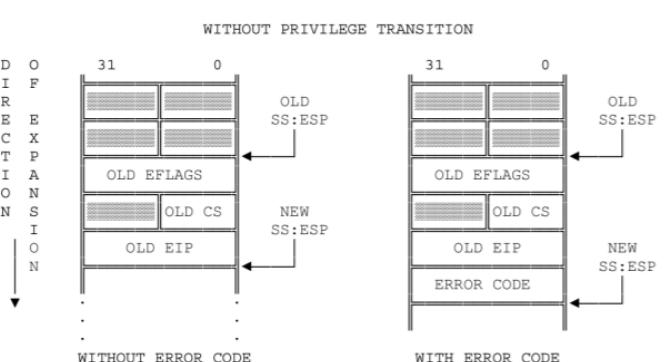
- in modalità utente, il registro ESP potrebbe avere un valore qualsiasi quindi, **ogni processo ha due stack**:
  - uno per la modalità **utente** (il normale ESP)
  - uno che si attiva quando si passa in modalità **kernel**, salvato in un registro speciale (è più complicato di così ma non ci interessa; per i curiosi: è nello State Task Segment, “puntato” da TR)
- **se l'interruzione arriva in modalità utente, la CPU passa allo stack kernel e salva il valore dello stack utente sullo stack kernel;** in questo modo quando ha finito col kernel ripristina lo stack di prima.
- vengono salvati i flag: EFLAGS e CS:EIP (instruction pointer)
- **qualche interrupt ha anche un codice di errore associato**, per riconoscere meglio il tipo di interruzione.

## Interrupt senza Cambio di Privilegio

Lo stack non cambia.

Questo interrupt può essere con o senza codice di errore, in caso viene aggiunto il campo “Error Code”.

La CPU salva flag e instruction pointer.

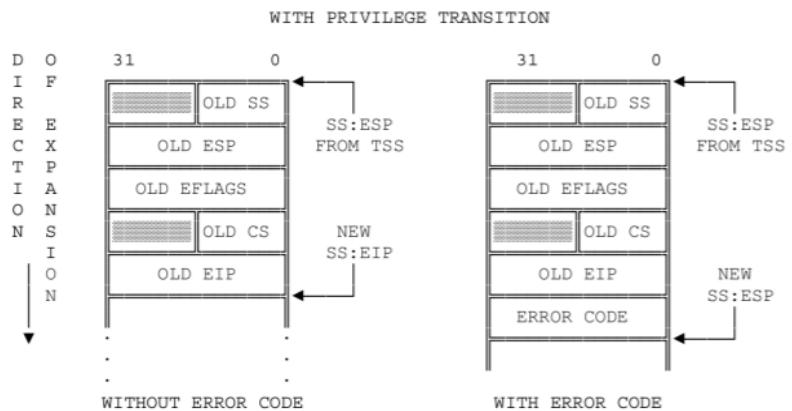


## Interrupt con Cambio di Privilegio

Lo stack cambia da Utente a Kernel o viceversa.

Questo interrupt può essere con o senza codice di errore, in caso viene aggiunto il campo "Error Code".

La CPU salva flag e instruction pointer.



## Interrupt in Xv6

Xv6 uniforma la gestione degli errori:

il file *kernel/vector.S*, contiene tutti gli handler agli interrupt.

Esso è generato da *utils/vector.pl*:

1. per uniformità, fa in modo che ci sia **sempre un error-code**
2. fa *push* del numero dell'interrupt
3. salta a ***alltraps*** → *kernel/trapasm.S*

quest'ultima:

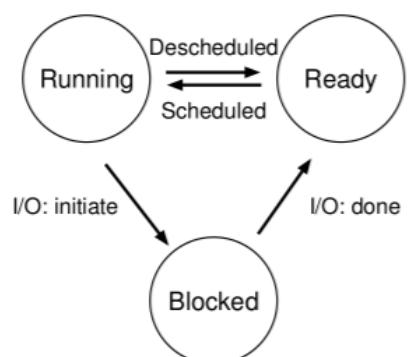
1. costruisce un trap-frame
  - a. *struct trapframe* definita in *kernel/x86.h*
  - b. contiene il numero di trap, l'error-code e (copia di) tutti i registri
2. e lo passa a trap:
  - a. che usa *trapno* per decidere il da farsi
  - b. se *T\_SYSCALL* (se c'è una syscall, quindi un cambio di privilegio)
    - i. salva il trap-frame nel PCB
    - ii. chiama syscall → *kernel/syscall.c*
3. al ritorno, ripristina i registri ed esce dalla gestione interrupt

Nota: il ritorno potrebbe essere dopo molto tempo, nel frattempo potrebbe quindi venir attivato un processo diverso --> *context-switch*:

Il processo che sta ritornando è messo in attesa e viene avviato un altro processo.

## Stati di un processo

Su Xv6 gli stati sono elencati in:



```

enum procstate {
    UNUSED,
    EMBRYO /* AKA: INIT */,
    SLEEPING,
    RUNNABLE /* AKA: READY */,
    RUNNING,
    ZOMBIE
};

```

Trucco per ricordarseli: reSURREZione

In altri Unix-like, es. Linux, le cose possono essere più complicate (due diversi tipi di sleep, anche stato uno stopped, . . . )

## (17)Lezione 32 (20/11/2020)

### Kernel debug

Per debuggare un kernel si utilizza il **remote debugging** (si usa un'altra macchina per debuggarlo); noi abbiamo la fortuna in Xv6 di farlo molto più semplicemente, dal terminale, attraverso il file **.gdbinit**; esso:

- carica i simboli del kernel in automatico: *symbol-file kernel/kernel*
- può essere modificato, permette infatti di modificare il vostro *~/.gdbinit* per far caricare quello nella directory corrente (nel caso, gdb vi dirà anche come)

In due finestre separate lancio:

1. in una *./debug*
2. in una *gdb*

da gdb inserisco l'istruzione **c (continue)**

- Siccome QEMU può emulare più processori virtuali ed ognuno di essi è un thread in gdb.
- Usiamo un solo processore (default CPUS=1) e ignoriamo, almeno per ora, la cosa per vedere la tabella delle pagine, da QEMU:
  - **ctrl+a c** → interrompe la macchina Xv6 ed entra nella console di QEMU.
  - **info mem** → Acquisisco informazioni sulla memoria (più in particolare le pagine)
  - **ctrl+a c** → esco dalla console di QEMU

### Lista processi

Dalla console di Xv6 il comando **ctrl+P** mostra, per ogni processo:

1. PID
2. PPID
3. stato
4. nome
5. **back trace**, ovvero l'**elenco degli EIP (instruction pointer)** sulla pila di chiamate nel kernel; da qui possiamo eseguire i comandi:
  - **info symbol** → vi aiuta a “decodificare” gli indirizzi

- *disassemble /s* (funzione) → mostra assembler+C

Per ricordarselo: **Padre Pio Non Sei Bello**

## Debug di programmi utente

A fine debug, se abbiamo già caricato dei simboli da programmi utente, possiamo eliminarli tutti e impostare quelli del kernel:

- *symbol-file* levo i simboli
- *symbol-file kernel/kernel* imposto quelli del kernel

Per poi aggiungere i simboli con *add-symbol-file*; per esempio:

- *add-symbol-file*
- *user/\_ls b f\_type*
- *c*

**In questo modo viene visualizzato anche il mio codice c, e posso seguirlo riga per riga.**

Possiamo inoltre **seguire una system-call anche all'interno del kernel**; posso quindi andare dentro all'esecuzione di una syscall e vedere da più vicino come funziona.

- Una delle delle syscall più semplice da osservare è *getpid*.

## Limited Direct Execution

Per virtualizzare la CPU si utilizza il **time sharing**; ovvero, **ogni processo riceve l'uso esclusivo della CPU per un po', poi si passa a un altro processo e così via**

Problemi:

- **Efficienza**: come fare tutto questo senza perdere troppo tempo nel passaggio da un processo all'altro?
- **Controllo**: come garantire che un processo rilasci l'uso della CPU?
  - Inoltre, come già discusso, dobbiamo impedire che un processo esegua operazioni "delicate", per esempio, parlare direttamente con l'HW.

Andiamo ad analizzare meglio questi problemi.

## Meccanismi per lo scambio di processi

Approccio:

1. **cooperativo**: il SO si "fida" dei processi
  - Ogni tanto i processi eseguono una syscall, anche se non ne hanno bisogno (sistemi di questo tipo tipicamente prevedono una *syscall ad-hoc: yield*)
  - Se non lo fanno, per malizia o per un bug, si passa all'approccio non cooperativo.
2. **non cooperativo**: il SO si riprende il controllo "a forza", tramite un **timer interrupt**.
  - Per esempio, in Xv6 in kernel/trap.c troviamo:

```

void trap(struct trapframe *tf) {
/* ... */
if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 +
IRQ_TIMER)
yield();
/* ... */

```

**yield()** viene chiamata da **alltraps** (definita in *kernel/trapasm.S*)

A questo punto il kernel può far partire, tramite un context-switch, un altro processo pronto (*ready*), vediamo meglio nel prossimo paragrafo.

## Salvare e ripristinare il contesto

Quando il SO riprende il controllo, deve decidere se far continuare l'esecuzione del processo corrente o meno.

La parte di kernel che prende questa decisione è lo **scheduler**

- Ci sono diverse **politiche** di scheduling, ne parleremo

Per cambiare processo, si fa un cambio di **contesto (context switch)**; l'idea è semplice:

1. si salvano i registri del processo che si sta eseguendo
2. si caricano i registri del processo che vogliamo mandare in esecuzione
  - nota: questo include anche, per esempio, il registro che "punta" alla tabella delle pagine

L'implementazione non è così semplice, perché deve tenere conto di tutti i dettagli; la funzione che implementa il context-switch è **swtch**.

## Ottimizzazione del contesto

Quando **swtch** verrà chiamata, i registri utente saranno già stati salvati nel trap-frame (nello stack kernel) del processo corrispondente, gli unici registri che devo salvare sono dunque quelli del kernel; quindi basta **scambiare i registri** che:

- **non vengono automaticamente gestiti dal compilatore** in seguito a una chiamata di funzione (EAX, ECX ed EDX)
- **non sono già salvati altrove**; per esempio il PCB contiene:
  - pde\_t \*pgdir;* indirizzo della tabella delle pagine
  - char \*kstack;* indirizzo dello stack kernel
- **e cambiano da un processo all'altro**
  - per esempio, i registri di segmento sono uguali all'interno del kernel

Il contesto si riduce a **edi, esi, ebx, ebp, eip**:

```

struct context { /* definita in kernel/proc.h */
    uint edi;
    uint esi;
    uint ebx;

```

```

    uint ebp;
    uint eip; /* inserito "automaticamente" da CALL */
}; /* esp usato per puntare a questa struttura */

```

nel PCB abbiamo:

```

struct proc {
    //...
    struct trapframe *tf;
    struct context *context; // valore caricato in esp da swtch
    //...
};

```

## Context-Switch

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
	<b>timer interrupt</b> save regs(A) to k-stack(A) move to kernel mode jump to trap handler	...
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) <b>return-from-trap (into B)</b>	restore regs(B) from k-stack(B) move to user mode jump to B's PC	Process B ...

Che tradotto in codice:

```

# void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

```

**swtch:**

```

    mov eax, [esp+4]
    mov edx, [esp+8]

    # Save old callee-saved registers
    push ebp
    push ebx
    push esi

```

```

push edi

# Switch stacks
mov [eax], esp
mov esp, edx

# Load new callee-saved registers
pop edi
pop esi
pop ebx
pop ebp ret

```

## Recap avvio:

- Accendiamo la macchina
- Partiamo in modalità **real address** dalla locazione di partenza ovvero il bios
- A quell'indirizzo c'è il **firmware** che si occupa di cercare il **boot** attraverso la sua firma e caricarne il primo settore ovvero il **boot block**
- Il boot block è diviso in:
  - *kernel/bootasm.S*
  - *kernel/bootmain.c*
- Si parte da *bootasm* per passare dalla modalità real address a 16 bit ad una modalità protetta a 32 bit
- *bootasm.S* carica anche la tabella dei segmenti
- Si passa a *bootmain.c* che dopo aver effettuato un **sanity check** carica i segmenti e divide lo spazio di indirizzamento
- Si passa a *entry.s* che abilita la paginazione e chiama il main (*main.c*), quest'ultimo fa una serie di inizializzazioni, crea l'init e lancia lo scheduler

-> perché usiamo Xv6? come lo usiamo? come facciamo a debuggare? cosa possiamo vedere durante il debug? come è virtualizzata la CPU? come passiamo da un processo all'altro? in cosa consiste il context-switch?

## (18)Lezione 33 (24/11/2020)

### Scheduling

#### CPU Scheduling

I processi che girano in un sistema sono detti **workload** (carico di lavoro).  
Nel contesto dello scheduling i processi sono spesso chiamati **job**.

#### Assunzioni

Per semplificare partiamo con assunzioni irrealistiche, che elimineremo man mano.

1. Ciascun job dura lo stesso tempo,
2. Tutti i job arrivano allo stesso momento (li conosciamo tutti all'inizio),
3. Una volta iniziato un job lo si porta fino in fondo (senza interruzioni),
4. Tutti i job usano solo CPU, no I/O (la più irrealistica),
5. Il tempo di ciascun job è noto a priori.

## Metriche

Per misurare la “bontà” di un algoritmo di scheduling, rispetto a un altro, abbiamo bisogno di **metriche**.

Inizialmente ci concentreremo sulle **performance** ma un altro aspetto da tenere in considerazione è la **fairness** (ovvero quando tutte le entità possono usare una determinata risorsa) (il contrario è **starvation**):

Definiamo il Turn-around Time, che misura le performance, come tempo di completamento sottratto al tempo di arrivo:

$$T_{turnaround} = T_{completion} - T_{arrival} \text{ con le assunzioni iniziali, } T_{arrival} = 0 \text{ quindi:}$$

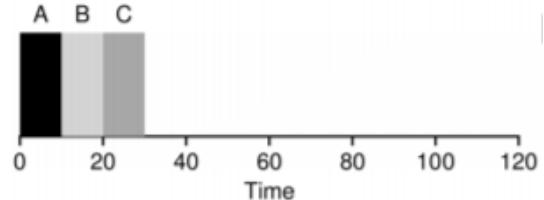
$$T_{turnaround} = T_{completion}$$

Calcoliamo vari Turn-around Time medi, attraverso l'uso di vari algoritmi, più o meno efficienti:

### Algoritmo FIFO (First In First Out)

A, B e C durano 10:

$$T_{turnaround} = (10 + 20 + 30)/3 = 20$$

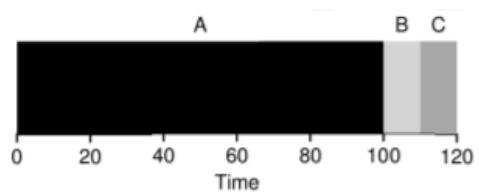


### Effetto convoglio

Rilassiamo quindi la prima assunzione, cioè che tutti i job durano lo stesso tempo: ci sono workload più brutti di altri? Vediamolo:

Se A dura 100, B e C 10, il tempo medio diventa:

$$T_{turnaround} = (100 + 110 + 120)/3 = 110$$



Per via dell'**effetto convoglio**.

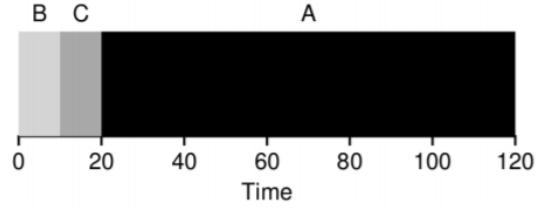
*Se sono dal macellaio e devo prendere solo un etto di prosciutto, ma quello davanti ordina mezzo negozio, devo comunque aspettare un sacco, anche se devo fare poco.*

## Shortest-Job First

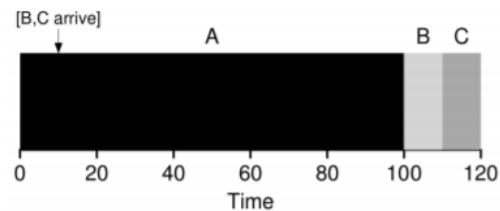
$$T_{turnaround} = (10 + 20 + 120)/3 = 50$$

Con l'assunzione che tutti i job arrivano allo stesso momento, si può provare che **SJF è ottimo**.

Funziona bene anche se togliamo questa assunzione?



Se A (che ricordiamo dura 100) arriva all'istante 0, e B e C (che durano 10) arrivano all'istante 10:

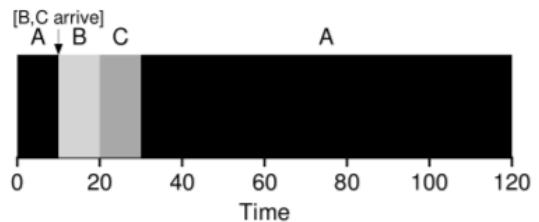


$$T_{turnaround} = (100 + (110 - 10) + (120 - 10))/3 \approx 103$$

Cosa si potrebbe fare?

## Shortest Time-To-Completion First

Rilassando l'assunzione di non interrompere i job, l'algoritmo **STCF è ottimo**.



$$T_{turnaround} = (120 + (20 - 10) + (30 - 10))/3 = 50$$

$$\text{poiché ricordiamo } T_{turnaround} = T_{completion} - T_{arrival}$$

Ancora poco realistica:

- non considera il context-switch tra job,
- l'assunzione 5 continua ad essere molto irrealistica, non sappiamo quasi mai quanto duri un job,
- questo algoritmo inoltre non è **fair** (continuo a eseguire prima processi corti e rischiamo di non finire o addirittura iniziare mai un processo lungo); per risolvere questo uso il response time.

## Response time

a.k.a. vedere che succede roba:

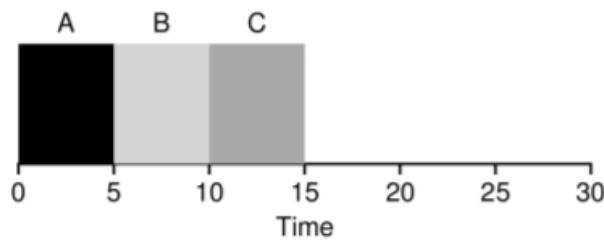
quando il response time è alto, possono accadere cose brutte:

ex. quando sciaccco sul telefono ma non mi piggia la clickata

Nei [sistemi batch](#) il turnaround può bastare, ma nei sistemi interattivi un'altra metrica è molto importante.

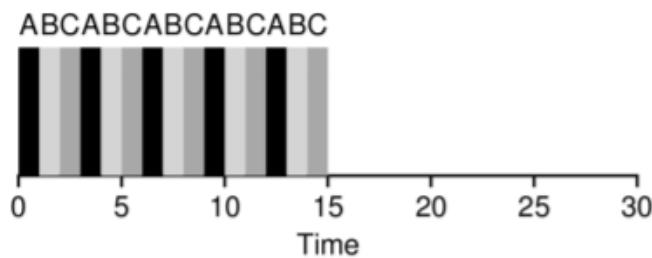
$$\text{Response time: } T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$$

Negli algoritmi visti, il response time non è buono; pensate a cosa succederebbe se arrivassero  $n$  job della stessa lunghezza: l'ultimo deve aspettare la terminazione degli altri ( $n - 1$ ) prima di essere mandato in esecuzione.



## Round-Robin

Nel Round-Robin ogni job ottiene una “fetta/quanto di tempo” e poi si passa al prossimo job:  
più le fette sono piccole, più è breve il tempo di risposta  
se però ho fettine troppo corte ho tantissimi context-switch



Supponiamo che A, B e C arrivino al tempo 0 e durano 5 secondi; il tempo medio sarebbe:

$$T_{\text{response}} = (0 + 1 + 2)/3 = 1 \text{ per il RR,}$$

mentre sarebbe:

$$T_{\text{response}} = (0 + 5 + 10)/3 = 5 \text{ per SJF/STCF}$$

Si può ottimizzare il response-time restringendo il quanto di tempo, ma bisogna tenere in considerazione l'overhead del context-switch.

**Il round-robin si comporta malissimo rispetto al Turn-Around Time.**

L'esecuzione viene “diluita” e il tempo di completamento si allunga.

- in generale gli algoritmi di scheduling "equi" (fair) evitano la starvation ma "penalizzano" i job corti.

## Assunzioni (non ancora eliminate)

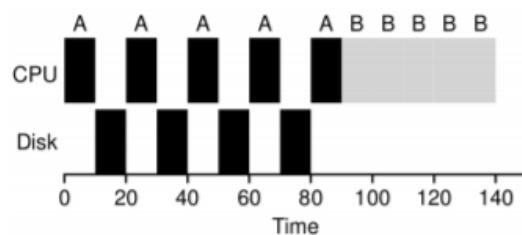
Abbiamo ancora due assunzioni da eliminare:

4. tutti i job usano solo CPU, no I/O .
5. il tempo di ciascun job è noto a priori.

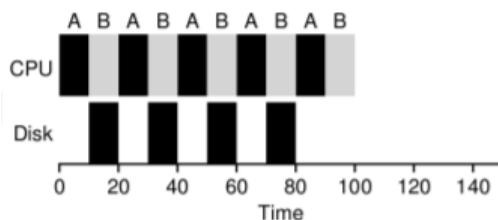
## I/O

Come abbiamo, già discusso, **sarebbe stupido tenere allocata la CPU per un processo che attende l'I/O:**

Cerco di evitare questo:



Quindi la cpu viene divisa solo tra i processi *ready*, mentre aspetto l'input:



## (19)Lezione 34 (27/11/2020)

### I/O Bound e CPU Bound

Un programma può essere:

- **I/O Bound:** sfrutta molto più I/O che CPU.
- **CPU Bound:** sfrutta molto più CPU che I/O.

I programmi CPU Bound utilizzano molto la CPU e quindi sono da considerarsi svantaggiosi e da schedulare il più tardi possibile.

I programmi I/O (detti **interattivi**), al contrario, sono molto meno dispendiosi per la CPU; devono quindi avere priorità maggiore e vanno schedulati prima; questo avviene nell'algoritmo MLFQ:

## Multi-level Feedback Queue

Il MLFQ tenta di ottimizzare:

- sia il **turnaround time**, facendo eseguire prima i job corti
- sia il **response time**

In genere uno scheduler non sa a priori le caratteristiche di un processo, come fa, quindi, a "impararle"?

### MLFQ: idee

MLFQ implementa **diverse code, ognuna con una diversa priorità**; a seconda di come si comporta un processo, gli viene assegnata una priorità.

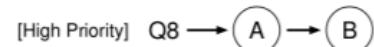
Se due processi hanno la **stessa priorità** vengono gestiti con il **Round-Robin**.

**La priorità** di ogni processo **varia dinamicamente** in base al comportamento osservato. L'idea è usare la storia di un processo per predirne il comportamento futuro.

### MLFQ: regole

Descriviamo l'algoritmo con delle regole, partiamo da:

1. Se  $p(A) > p(B)$ , gira A (e non gira B)
2. Se  $p(A) = p(B)$ , A e B in Round-Robin



Q7

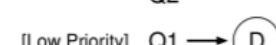
Q6

Q5

Q4

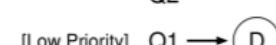
Q3

Q2



Con solo le prime due regole, finché A e B non terminano, ci sarebbe una condizione di **starvation** per C e D.

Devo definire altre regole per evitarlo:



### MLFQ: nuove regole, primo tentativo

1. Se  $p(A) > p(B)$ , gira A (e non gira B);
2. Se  $p(A) = p(B)$ , A e B in RR;  
*in condizione iniziale entrano tutti con la stessa priorità e quindi in RR*
3. Un nuovo job entra con la priorità massima.
4. Quando è in modalità Round-Robin, se un job usa tutto il suo quanto di tempo (il 100%) sfruttando la CPU e mai in I/O, allora la sua priorità viene ridotta (altrimenti rimane alla stessa priorità).  
*sta usando troppo la CPU, allora lo declassiamo*

Problemi di queste regole

1. **Starvation:** se continuano ad arrivare job, quelli di bassa priorità non verranno mai eseguiti;
  - Una volta che si è perso priorità, non la si guadagna più (anche se si usa pochissima CPU)
2. Si può “**barare**”: se un processo rilascia la CPU al 99% del suo quanto, non perde priorità.

MLFQ: nuove regole, secondo tentativo

1. Se  $p(A) > p(B)$ , gira A (e non gira B)
2. Se  $p(A) = p(B)$ , A e B in RR
3. Un nuovo job entra con la priorità massima
4. Quando è in modalità Round-Robin, se un job usa tutto il suo quanto di tempo (il 100%) sfruttando la CPU e mai in I/O, allora la sua priorità viene ridotta (altrimenti rimane alla stessa priorità).
5. **Ogni tot secondi, spostiamo tutti i job alla priorità più alta.**  
*resetiamo le priorità e torniamo in RR, evitando starvation*

Migliorie e Problemi di queste regole

- Non c’è più starvation e se un processo diventa I/O-bound, rimane ad alta priorità.
- Si può sempre “barare”, *se qualcuno usa il 99% del suo quanto sfruttando la CPU, risulta comunque buono (quando non lo è).*

MLFQ: regole finali

1. Se  $p(A) > p(B)$ , gira A (e non gira B).
2. Se  $p(A) = p(B)$ , A e B in RR.
3. Un nuovo job entra con la priorità massima.
4. **Quando un job usa un tempo fissato  $t$  a una certa priorità  $x$  (considerando la somma dei tempi usati nella coda  $x$ ), allora la sua priorità viene ridotta.**  
*Negli algoritmi la soglia era del 100%, e si rischiava che qualcuno barasse, ora la soglia è fissata e decidiamo noi quando un processo deve essere declassato.*
5. Ogni s secondi, spostiamo tutti i job alla priorità più alta.

Migliorie e Problemi di queste regole

Con queste regole le cose più o meno funzionano;

C’è ancora però un problema:

Vi sono dei parametri da scegliere:

- Quante code?
- Quanto vale  $t$ ?
- Ogni quanto si resetta tutto?

La “soluzione” può essere lasciar scegliere i parametri a un admin.

## Proportional Share

E' un approccio completamente diverso rispetto a MLFQ.

Gli scheduler proportional-share, invece di cercare di ottimizzare i tempi di workaround o response, si basano su un concetto semplice: **ogni processo dovrebbe avere la sua percentuale di tempo di CPU.**

### Nice

Esiste un valore **nice**, che quantifica la priorità e l'importanza di un processo:

- è 0 di default,
- va da -20 a +19,
- può essere modificato con le syscall *nice(1)* e *nice(2)*.

E che è a sua volta contenuto nella classe di **ionice** che è il valore di nice riguardante le operazioni di input/output:

- è 2 di default,
- va da 0 a 3,
- può essere modificata con le syscall *ionice(1)* e *ioprio\_set(2)*.

### CFS (Completely Fair Scheduler)

Assegna maggiore priorità a chi usa meno la CPU.

CFS conteggia il tempo usando **virtual runtime: vruntime**

Nel caso più semplice, il *vruntime* è **proporzionale** al tempo vero.

Quando c'è da scegliere un processo da mandare in esecuzione, **CFS seleziona quello con il vruntime più piccolo;**

- per quanto? come sempre c'è da considerare l'overhead del context-switch.

CFS usa tantissimi parametri, uno è **sched\_latency**:

Ogni *sched\_latency* si chiede: quanti processi ho che devono essere eseguiti? e divide lo sched in n fettine con n=numero processi pronti

Se ci sono n processi pronti, si manda in esecuzione quello col *vruntime* più basso per un tempo  $\frac{sched\_latency}{n}$

- purché non sia inferiore a un altro parametro: **min\_granularity**  
*min\_granularity mi serve per non fare fettine troppo piccole, che sarebbe troppo svantaggioso per via del context-switch.*

Si applica inoltre un **fattore di scala**, per tenere conto del valore **nice** (priorità) di ogni processo, per cui:

*la fettina che si prende può essere un po' più grossa o piccola in base alla priorità (al valore di nice) del processo.*

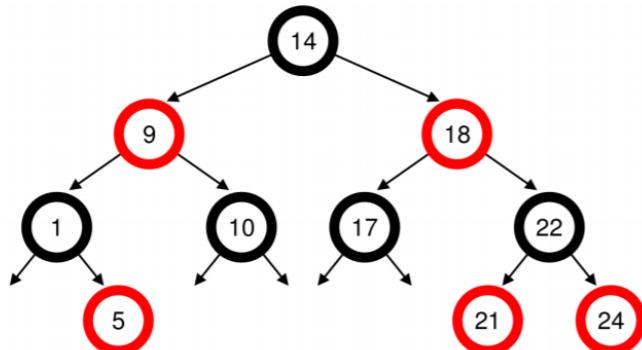
$$Slice_k = \max(sched\_latency \cdot \frac{w_k}{\sum_{i=0}^{n-1} w_i}, min\_granularity)$$

$$vruntime += \frac{runtime_k}{w_k}$$

CFS: Efficienza, alberi rossi/neri

**CFS deve sempre andarsi a prendere il *vruntime* più piccolo.**

Per trovare **efficientemente** il minimo/aggiornare il *vruntime* dei processi pronti, essi vengono tenuti in un albero binario bilanciato di tipo rosso/nero:



CFS: I/O

Bisogna considerare cosa succede quando un processo viene messo in attesa:

Quando un processo torna *ready* se lasciassimo inalterato il *vruntime* (a 0) potrebbe **monopolizzare** a lungo la CPU.

Quindi, un processo che diventa *ready* (dopo sleep o init) si prende un ***vruntime uguale al minimo degli altri***;

- si evita starvation al costo di non essere fair con i processi frequentemente per breve tempo (non è veramente *completely fair*).

## (20)Lezione 36 (01/12/2020)

### Thread

Introduzione ai thread

Il kernel virtualizza le risorse, inclusa la CPU (ogni processo pensa di averla tutta per sé).

Inoltre, i **thread** permettono di avere più flussi di esecuzione in un processo, sempre all'interno dello stesso spazio di indirizzamento.

Possiamo ad esempio scambiare puntatori tra thread, quindi un thread è ben diverso da un processo.

- in un certo senso, **creare un thread significa creare una “CPU virtuale”**

### Differenze e Similitudini tra Thread e Processi

- la differenza principale tra thread e processi è la condivisione (o meno) dello spazio di indirizzamento;
  - inoltre, **il context-switch tra thread costa meno** del context-switch tra processi, dobbiamo cambiare meno grazie alla condivisione tra thread diversi.
- in ogni caso, **i thread vanno schedulati** (come i processi):
  - Linux non fa differenza: entrambi vengono schedulati **allo stesso modo**;

Ogni thread **ha il proprio stack, mentre condivide codice e dati** con gli altri thread dello stesso processo

- In verità, tecnicamente, gli stack sono nello stesso spazio di indirizzamento, e quindi un thread può accedere anche allo stack di un altro thread (cosa che però va evitata).

Approfondimenti sui Thread: [Slide 1](#), [Slide 2](#).

### Perché usare i thread?

1. Sfruttare il **parallelismo**.  
*suddividersi il lavoro per lavorare su zone diverse dello stesso problema; se devo fare 800 operazioni e ho 8 thread posso farne fare 100 ad ognuno di essi.*
2. **Non bloccarsi per fare I/O** (non è l'unico modo, ma facilita le cose).  
*leggere da I/O mentre facciamo qualcos'altro.*

### Creazione di thread

Considerando le API POSIX, la funzione per creare un thread è **pthread**:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

```
// Compile and link with -pthread.
```

Attenzione: **in caso di successo restituisce 0**, ma

- **in caso di errore, restituisce direttamente il codice di errore** (NON lo scrive in errno)

## Pagine di manuale

Su Ubuntu dovete installare il package *manpages-posix-dev* per rendere disponibili tutte le pagine di manuale corrispondenti ai *pthread*.

## Terminazione e attesa

Potete uscire, con un certo valore, da un thread facendo un return dalla funzione iniziale, oppure chiamando:

```
void pthread_exit(void *retval);
```

E attenderne la terminazione con:

```
int pthread_join(pthread_t thread, void **retval); //serve per sincronizzare, un thread aspetta la terminazione di un altro thread. garantisce mutua esclusione
```

Vediamo un esempio → *threads.v0.c*

In esso vi sono due thread che incrementano un *counter*, fino ad arrivare ad un momento in cui si rompe:

Grazie a [Compiler Explorer](#) possiamo capire facilmente la ragione del comportamento bizzarro:

The screenshot shows two windows from the Compiler Explorer tool. On the left is the C source code editor with the file 'threads.v0.c' containing the following code:

```
1 volatile int counter = 0;
2 int loops;
3
4 void *worker(void *arg)
5 {
6     for(int i = 0; i < loops; i++) {
7         counter++;
8     }
9 }
```

On the right is the assembly output window titled 'x86-64 gcc 10.2 (Editor #1, Compiler #1) C'. The assembly code generated for the 'worker' function is:

```
1 counter:
2     .zero    4
3 loops:
4     .zero    4
5 worker:
6     push    rbp
7     mov     rbp, rsp
8     mov     QWORD PTR [rbp-24], rdi
9     mov     DWORD PTR [rbp-4], 0
10    jmp    .L2
11    .L3:
12    mov     eax, DWORD PTR counter[rip]
13    add     eax, 1
14    mov     DWORD PTR counter[rip], eax
15    add     DWORD PTR [rbp-4], 1
16    .L2:
17    mov     eax, DWORD PTR loops[rip]
18    cmp     DWORD PTR [rbp-4], eax
19    jl     .L3
20    nop
21    pop    rbp
22    ret
```

Il counter è una **sezione critica**! Esso è infatti formato da 3 operazioni: *mov*, *add*, *mov*; può succedere che un thread sia interrotto tra queste 3 istruzioni, e faccia casino.

## Sezioni critiche e race-condition

Una **sezione critica** è un frammento di codice che accede a una **risorsa condivisa**.

Quando si hanno più flussi di esecuzione, si parla di **race condition** (alle volte detta *data race*); il risultato finale dipende dalla temporizzazione o dall'ordine con cui vengono schedulati

- La computazione è non-deterministica

- Per esempio, **si ha una race-condition quando più thread eseguono (più o meno allo stesso tempo) una sezione critica**

Per evitare questi problemi, serve **sincronizzazione** fra i thread, alle volte, anche di processi diversi.

Per effettuarla servono le **primitive di sincronizzazione**;

Ne esistono molte ma presentiamo qui solo le primitive per la **mutua esclusione**.

## (21)Lezione 37 (04/12/2020)

### Locks

*Un robo che permette di sincronizzare due thread rendendo atomiche le operazioni; permette di entrare in una sezione critica un thread alla volta.*

*E' come se il thread prendesse una bandierina (lock) e poi la poggiasse quando ha finito (unlock) e per entrare nella sezione critica un thread deve avere quell'unica bandierina.*

Il problema dell'esempio era la mancanza di **atomicità** nell'incrementare il contatore.

Una soluzione è introdurre dei **lock**, implementati dai **mutex** in *pthread*:

Un lock:

- si dichiara con ***pthread\_mutex\_t*** e si può inizializzare tramite:
  - assegnazione della costante ***PTHREAD\_MUTEX\_INITIALIZER***
  - ***pthread\_mutex\_init***
- può essere acquisito (preso/tenuto), da un thread alla volta, tramite ***pthread\_mutex\_lock***
- va rilasciato, il prima possibile, tramite ***pthread\_mutex\_unlock***.

I lock pagano la sincronizzazione dei thread con un molto **maggior costo in termini di tempo** (circa 10 volte più grande).

### Quanti lock?

Se un programma usa più strutture dati (condivise) diverse, probabilmente meglio **avere più lock**; questo perché **aumenta l'efficienza** (anche se può **complicare la gestione**);

Non bisogna ovviamente esagerare; per esempio, un lock per ogni nodo di una lista è (tipicamente) troppo: questo perché bisogna considerare anche l'overhead.

Per esempio, in Xv6 :

In *kernel/proc.c*:

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

In *kernel/file.c*:

```

struct {
    struct spinlock lock;
    struct file file[NFILE];
} ftable;

```

## Implementazione di lock

Come possiamo costruire dei lock? Come possiamo confrontare implementazioni diverse?

Vanno considerate:

- **mutua esclusione** (quello che deve fare un lock);
- **fairness/starvation**;(non tenere troppo tempo le risorse e poi lasciarle agli altri)
- **performance**.(prestazioni)

Consideriamo quindi varie possibilità;

### Disabilitare le interruzioni

Quando andiamo ad usare i lock disabilitiamo lo scheduler con i suoi interrupt, evitando context-switch:

```

void lock() {
    DisableInterrupts();
}

void unlock() {
    EnableInterrupts();
}

```

**Ha senso? Vari problemi:**

- istruzioni privilegiate
- vogliamo davvero permettere a un processo di disabilitare le interruzioni e monopolizzare la CPU?
- nei sistemi multi-processore/core non funziona: due core possono eseguire la stessa sezione critica

Proviamo un'altra strada;

Una semplice struttura dati

Una struct con un unico campo che ci dice se il lock è disponibile (0) oppure non è disponibile (1):

```

typedef struct __lock_t {
    int flag;
} lock_t;

```

```

void init(lock_t *mutex) { // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}

```

Problema: non funziona

Supponiamo che flag sia 0; se abbiamo due thread ed essi vengono interrotti in mezzo al codice finiscono ad avere lo stesso valore di flag, e non garantire mutua esclusione.

Thread 1	Thread 2
call lock ()	
while (flag == 1)	
<b>interrupt: switch to Thread 2</b>	
	call lock ()
	while (flag == 1)
	flag = 1;
	<b>interrupt: switch to Thread 1</b>
flag = 1; // set flag to 1 (too!)	

Figure 28.2: Trace: No Mutual Exclusion

Serve supporto hardware

Diversi processori forniscono delle primitive atomiche, come ad esempio:

- Test-and-Set (usa i flag ma in modo atomico);
- **scambi atomici** (scrive in un indirizzo di memoria un valore e ci restituisce il valore che c'era prima, in modo atomico).

Su x86 abbiamo l'istruzione XCHG che corrisponde a:

```

/* pseudo-code (nel x86, istruzione XCHG) */
int AtomicExchange(int *ptr, int new) {
    int old = *ptr;
    *ptr = new;
    return old;
}

```

Con queste istruzioni si possono implementare degli **spin-lock**:

Spin-lock: implementazione

```
typedef struct __lock_t {
    int is_locked;
} lock_t;

void init(lock_t *lock) {
    lock->is_locked = 0;
}

void lock(lock_t *lock) {
    // finché ci restituisce uno non facciamo niente, quando ci restituisce 0 entriamo
    while (AtomicExchange(&lock->is_locked, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->is_locked = 0;
}
```

Questo funziona?

Spin-lock: analisi

**E' un meccanismo che entra in gioco quando si vuole acquisire un lock per garantire mutua-esclusione attraverso l'utilizzo di supporto hardware, per evitare che un thread venga deschedulato mentre fa *lock()* e due thread entrino quindi nella stessa sezione critica.**

**Non è fair;** un thread in attesa non ha qualche garanzia di ottenere, prima o poi, il lock, quindi è a rischio starvation.

Per quanto riguarda le performance dobbiamo distinguere rispetto al numero di core:

- singolo core: se un thread che ha il lock viene deschedulato, gli altri sprecano tempo e CPU.
- core multipli: funziona discretamente bene.

Ha senso aspettare in un loop (consumando CPU)? se si aspetta poco sì, ma i context-switch costano;

Negli anni, sono state fatte varie proposte di approcci ibridi, ovvero "lock in due fasi":

Implica l'esecuzione di un po' di spin, poi eventualmente sleep.

*ad esempio: cicliamo 1000 volte e poi se non siamo riusciti ad entrare andiamo in sleep e veniamo svegliati dal kernel quando si è liberato il lock.*

# (22)Lezione 39 (11/12/2020)

## Inversione delle priorità

**Scheduling e (spin-)lock possono interagire in modi inaspettati.**

*Nei sistemi complessi singole componenti possono funzionare benissimo da sole ma in modo inaspettato se combinate assieme.*

Supponete di avere un scheduler a priorità e due thread:

- T1 bassa priorità, T2 alta priorità
- supponiamo che T2 sia in attesa di qualcosa, va in esecuzione T1.
- T1 acquisisce un certo lock L
- T2 ha finito l'I/O e torna ready
- lo scheduler deschedula T1 e manda in esecuzione T2, T1 però continua ad avere il lock.
- T2 va in spin-wait per il lock L, T1 non va in esecuzione perché aspetta T2
- Game Over

Ci sono modi per risolvere il problema, per esempio “**ereditare la priorità**” di un thread in attesa (se maggiore di quella corrente); come è stato fatto per il NASA Pathfinder.

*T2 è in attesa di T1, T2 eredita la priorità di T1, sbloccando la situazione; una volta finita questa situazione T2 riacquisisce la sua vecchia priorità.*

## NASA Pathfinder

Vi era un watchdog che controllava che il sistema fosse responsivo (rispondesse spesso); si verificava il problema descritto prima e il watchdog continuava a resettare il robot; si è risolto adottando l’ereditarietà della priorità.



## Tipi di bug dei Thread

Del codice corretto in un mondo single-threaded, può avere molti problemi quando si passa a un mondo multi-threaded; essi sono inoltre difficilmente debuggabili.

Vediamo qualche esempio:

Violazione dell'atomicità

```
struct foo *p; void f()
{
    ...
    if (p!=NULL) {
        p->bar = 3;
    }
}

struct foo *p; void f() {
    ...
    if (p!=NULL) { p->bar = 3; // boom; p==0 }
}
```

In un programma single-thread non ci sarebbero problemi.

In un programma multi-thread se un thread fosse nell'if, un altro thread potrebbe porre p a NULL; bisogna usare i mutex o altre primitive di sincronizzazione.

Violazione dell'ordine

```
void init()
{
    ...
    glob_thread = create_thread(thread_func, ...);
    ...
}

void thread_func(...)
{
    ...
    foo = glob_thread->bar; // boom: glob_thread uninitialized
    ...
}
```

Vi è un problema legato al multi-thread: vi sono due thread ed entrambi possono andare in esecuzione; vi è un ordine di esecuzione di essi per cui *glob\_thread* risulta non inizializzato. Quando abbiamo piu thread **non è detto che essi seguano l'ordine di scheduling** che noi vogliamo.

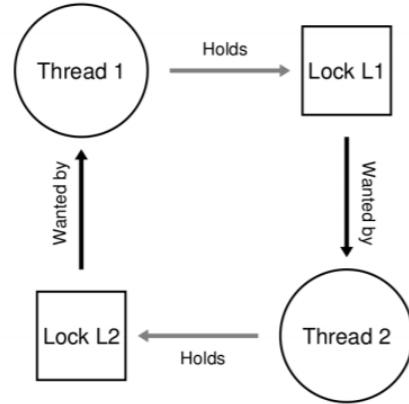
## Deadlock

Consideriamo due thread, con i loro lock:

*Thread-A lock(m1);  
lock(m2);*

*Thread-B lock(m2);  
lock(m1);*

Può succedere che A prenda m1 e poi venga deschedulato;  
B a quel punto prende m2 e poi cerca di prendersi m1; intanto A possiede m1 e vorrebbe m2.  
Questa situazione di stallo è chiamata **attesa circolare**.



Condizioni necessarie per il deadlock

Condizioni per il deadlock (devono essere tutte vere poiché si verifichi un deadlock):

- **mutual exclusion**: i thread hanno controllo esclusivo (non possiamo rimuoverla).
- **hold-and-wait**: i thread mantengono le risorse acquisite mentre ne richiedono/aspettano altre (ci si può organizzare per rimuovere questa condizione, ma è difficile e inefficiente implementare il contrario).
- **no preemption**: le risorse non possono essere tolte (può essere rimossa uccidendo un processo a caso che si trova in circular-wait).
- **circular wait**: ci deve essere un ciclo nelle attese (può essere rimossa).

Prevenire il deadlock

**Sol.1:** Imporre un ordine sui lock e acquisirli in ordine (no circular wait); è la tecnica più semplice ed utilizzata.

**Sol.2:** Fare in modo che i lock vengano sempre acquisiti tutti assieme, atomicamente (no hold-and-wait).

## Sicurezza

### Introduzione

Questa prima parte di slide è basata su: [slide 1](#), [slide 2](#).

Il s.o. è alla base di tutte le applicazioni e costituisce le fondamenta di ogni applicativo:

- Non possiamo costruire applicazioni sicure se l'S.O. non lo è;
- Non possiamo realizzare s.o. sicuri se l'HW non lo è.

Cosa vuol dire “sicuro”? Ne parlerete a Computer Security ma, ad altissimo livello, si parla di tre obiettivi:

- |                          |                           |
|--------------------------|---------------------------|
| - <b>Confidentiality</b> | Confidenzialità/seretezza |
| - <b>Integrity</b>       | Integrità                 |
| - <b>Availability</b>    | Disponibilità             |

In generale, potremmo riassumere con **condivisione controllata**; per esempio, potremmo voler fare leggere un nostro file ad alcuni utenti ma non ad altri.

## Contesto

Prima di eseguire una syscall il **kernel** deve valutare se la richiesta:

1. è “sensata” (#-syscall e parametri validi)
2. rispetta la **politica di sicurezza**;

Terminologia:

- l'entità che fa la richiesta è chiamata **principal** o **subject**;
- le richieste sono relative a una risorsa, l'**object** e, tipicamente, una **modalità di accesso** (per esempio, lettura o scrittura).

Nel valutare se eseguire, o meno, bisogna considerare il **contesto**, principalmente, chi fa la richiesta; per esempio in alcuni casi tutti i processi avviati da un utente girano con gli stessi permessi, ma non sempre:

- Windows/Linux si comportano diversamente rispetto ad Android, per esempio.

Un utente non parla **direttamente** con il kernel; servono principalmente (ma non solo):

- un'associazione fra utenti e processi;
- un modo per verificare l'identità degli utenti.

## AAA

<b>Authentication</b>	Autenticazione: verifica dell'identità ( <i>username e password</i> );
<b>Authorization</b>	Autorizzazione: applicazione di una politica di sicurezza; decidere se accettare o rifiutare le richieste;
<b>Accounting</b>	Contabilità: <i>logging</i> (scrivere su un database chi sta agendo sulle risorse) e gestione del consumo di risorse.

## Identificazione e autenticazione

Argomento molto vasto; caso più comune, nei sistemi Unix-like:

- gli utenti si identificano con un **username**, e si autenticano con una **password**:
  - file */etc/passwd* e */etc/shadow*.
- il kernel identifica ogni utente con un numero intero: **UID**.
  - analogamente, i gruppi con un **GID**; vedere */etc/group*.
  - da riga di comando: **id**.

Ci concentreremo sugli UID, il discorso per i GID è analogo:

- **I'UID zero corrisponde a root**, l'amministratore di sistema;
  - attenzione: non confondete la radice del FS con l'utente amministratore
- tradizionalmente, i processi si dividono in:
  - **privilegiati** con UID = 0,
  - **non-privilegiati** con UID != 0.
- le Linux capabilities permettono una maggiore granularità, si veda *capabilities(7)*; ai fini di SETI, le ignoriamo per semplicità.

## Autorizzazione

*Decidere se un subject può accedere ad una determinata risorsa.*

Ci sono due approcci generali all'autorizzazione:

- **access control list**  
*tipo un buttafuori in una festa privata, fa entrare solo chi è in lista*
  - per ogni object: lista delle coppie subject/access
  - Unix: versione “ottimizzata” (=ridotta) di ACL in 9 bit
    - r, w, x per proprietario, gruppo, altri
- **capabilities** (attenzione: non c'entrano nulla con le Linux capabilities)  
*dare la copia delle chiavi a chi deve entrare*
  - per ogni object/access ci sono delle “chiavi”, che ne permettono l'uso

L'idea generale è semplice, ma poi vanno considerati molti dettagli, tra cui:

- dove memorizzare queste informazioni/quanto spazio serve,
- cosa bisogna aggiornare quando aggiungiamo/eliminiamo utenti.

## DAC vs MAC

Indipendentemente dall'uso di ACL/capabilities, chi decide chi può/non-può accedere a una risorsa?

- Il proprietario, si parla di **DAC: discretionary access control**.
- Una qualche autorità impone le regole (e.s. ambito militare), si parla di **MAC: mandatory access control**.

I sistemi più comuni seguono il DAC, anche se non al 100%:

- Per esempio, tradizionalmente *root* può leggere qualsiasi file.
- Esistono varianti MAC di Linux; [esempio](#) (contiene sopra al DAC anche dei controlli MAC).

## Principio del minimo privilegio

In ogni momento, ciascuna entità dovrebbe avere il minimo privilegio che gli permetta di eseguire i suoi compiti legittimi.

Detto diversamente, nessun processo dovrebbe mai poter accedere a più risorse del minimo indispensabile per il suo corretto funzionamento.

In generale, è importante considerare il **ruolo** (in quel momento).

Lo stesso utente può avere diversi ruoli in Unix:

- pensate a *login(1)* deve poter leggere *etc/shadow*, accedere alla *home directory*, e altro, ma quando poi viene loggato un utente senza privilegi, i privilegi di login vengono rilasciati.
- esistono dei comandi per aumentare i privilegi:
  - storicamente: *su(1)* (super-user)
  - oggi: *sudo(1)* (do something in super-user)

Passiamo a *root* per, ad esempio, installare un pacchetto e poi torniamo ai privilegi precedenti;

Come funzionano *su* e *sudo*?

## UIDs

Ogni processo ha tre UID:

- **real UID** il proprietario del processo, ovvero, **chi può usare kill** su quel processo.
- **effective UID** identità usata per **determinare i permessi di accesso a risorse condivise**; per esempio ai file.
- **saved UID**
- (solo Linux: FS UID, lo ignoriamo; si veda *credentials(7)*)

**AI login, tutti e tre coincidono.**

Quindi il real UID è la tua **vera identità** e l'effective UID è **quello a cui guarda il SO quando deve decidere se permetterti o no l'accesso ad una risorsa**. Quando si effettua la syscall *sudo* il tuo effective UID viene cambiato in quello di root, finita l'operazione viene resettato al valore contenuto in real UID.

Tramite *setuid(2)* un processo può modificare l'effective UID, facendolo diventare come il real o il saved:

- se il chiamante è privilegiato, può modificare tutti e tre come vuole

## Set-UserID

Un nuovo processo “eredita” gli id del parent; di solito, *execve* non cambia gli id del processo chiamante, tranne in un caso: se il file eseguibile ha il **bit set-userid** (set user ID on execution) abilitato, allora:

- **Effective UID e Saved UID** diventano quelli del proprietario del file

Questo solitamente accade per i file di proprietà di root e si vede perché tra i permessi che vediamo con */s -l* c'è una **s**.

Ciò significa che quando eseguiamo questo comando **diventiamo automaticamente root**.

Per esempio: supponiamo che un processo venga lanciato con effective UID=0 (con bit *set-userid*). Se il programma è scritto bene, quando fa azioni che non richiedono privilegi da root, lo cambia impostandolo uguale a quel o real. Quando invece deve svolgere azioni che richiedono il privilegio, può reimpostare l'effective a zero "copiandolo" dal saved.

Un esempio d'uso è quando un utente vuole cambiare la sua password, chiamando il comando *passwd*.

Il proprietario di *passwd* è il root e marcato con il bit *set-uid*: viene quindi garantito all'utente l'accesso temporaneo a quella risorsa.

Si può impostare il bit-setuid tramite il comando: *chmod u+s myfile*

Esiste inoltre un bit analogo per i gruppi: *set-gid*

## Chroot jails e namespace

*chroot(2)* è una syscall che permette di modificare il significato di "/" nella risoluzione dei percorsi assoluti, ovvero di limitare la visibilità del FS ad una determinata directory:

- la modifica è per il processo e tutti i (futuri) figli
- è necessario che il processo sia privilegiato
- nota: i FD aperti non vengono toccati

permette di creare le cosiddette **chroot jails**:

- **limitando la visibilità del file-system a una directory** (e sotto-directory)
- **è un'applicazione del principio del minimo-privilegio** ma
- se il processo rimane privilegiato e/o la syscall non è correttamente associata a una *chdir(2)*, è possibile "uscire di prigione"

Per esempio, anche in caso di bug, in incApache un client non può "sbirciare" fuori dalla www-root;

Il meccanismo "antenato" per creare isolamento sono i **namespace**, limitati però al file-system;

## Container

Il meccanismo moderno per creare isolamento sono i **container** (usati, per esempio, dal famoso Docker); NON fanno parte del programma di SETI, ma per chi vuole approfondire, consiglio i seguenti video:

Breve panoramica: [How Docker Works - Intro to Namespaces by LiveOverflow](#).

Containers unplugged: [Linux namespaces by Michael Kerrisk](#).

Fuel Containers unplugged: [understanding user namespaces by Michael Kerrisk](#).

[Understanding and Working with the Cgroups Interface by Michael Anderson](#).

# (23)Lezione 41 (18/12/2020)

## Persistenza

In informatica, il concetto di persistenza si riferisce alla caratteristica dei dati di un programma di sopravvivere all'esecuzione del programma stesso che li ha creati: senza questa capacità questi infatti verrebbero salvati solo in memoria Ram venendo dunque persi allo spegnimento del computer.

## Supporti di memoria

Chiamiamo comunemente "dischi" i dispositivi a blocchi di memoria secondaria, ovvero:

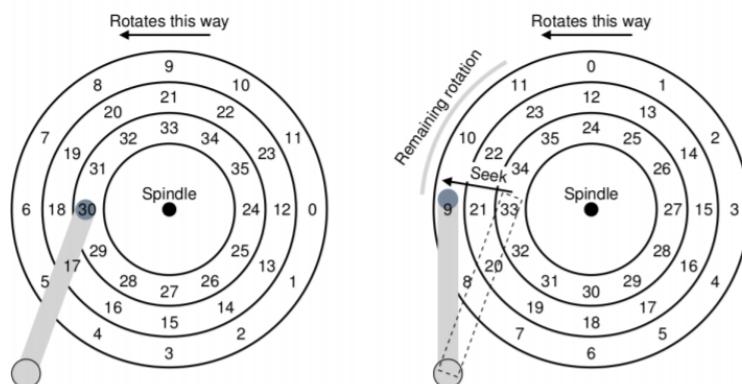
- **dischi magnetici** (hard/floppy disk di vario tipo)
  - Bisogna formattarli a livello hardware: la formattazione a basso livello prepara questa struttura logica sui dischi magnetici (da decenni sono venduti "pre-formattati").
- **dischi ottici** ad esempio DVD e Bluray;
- **"pennette" USB**;
- **SSD**: Solid-state Storage Device/Drive/Disk.

Dal punto di vista logico/astratto, sono tutti una **sequenza di settori**, tipicamente da 512 byte l'uno.

## Dischi

Una volta erano dischi, cioè oggetti piatti e di forma circolare.

Accedere a dati "vicini" costava meno che accedere a dati "lontani", in un fenomeno detto **ritardo rotazionale**: il tempo per spostare la testina da una traccia all'altra; *se sono sul blocco 19 posso spostarmi al 20 in pochissimo ma per spostarmi al 18 devo rifare tutto il giro*.



## Interfaccia

Un moderno SSD non è rotondo, non ruota, e funziona in modo completamente diverso a livello hardware.

Alcune convenzioni, **la terminologia e l'interfaccia sono sostanzialmente rimaste**

- un s.o. potrebbe usare un SSD, che esporta un'interfaccia “da HD”, senza saperlo (la cosa non è ottimale, ovviamente)

L'interfaccia semplificata che considereremo è: **un disco ha n settori/blocchi, che possiamo indirizzare con “indirizzi” da 0 a (n – 1).**

- notare che la “granularità” di accesso è il **settore**: 512 byte.  
*possiamo quindi leggere o scrivere solo interi blocchi, se dobbiamo cambiare 1 byte, il sistema operativo si porta dietro anche gli altri 511.*

## Buffer cache

L'I/O su disco è ordini di grandezza più costoso dell'accesso in RAM.

I sistemi Unix-like utilizzano quindi una **(unified) buffer cache**.

Una volta le cache erano 2 e si dividevano i compiti:

- **buffer-cache** per i blocchi disco acceduti tramite *read/write*.
- **page-cache** per i file mappati in memoria tramite la sys-call *mmap()*, che, a sua volta, si appoggiava sulla buffer-cache.

Su Linux, per forzare la scrittura dei dati/metadati in cache si può usare il comando *sync(2)*.

La presenza di memorie cache è la ragione per cui è importante “**espellere**” (=smontare) i **driver** prima di scollarli!

## Partizioni

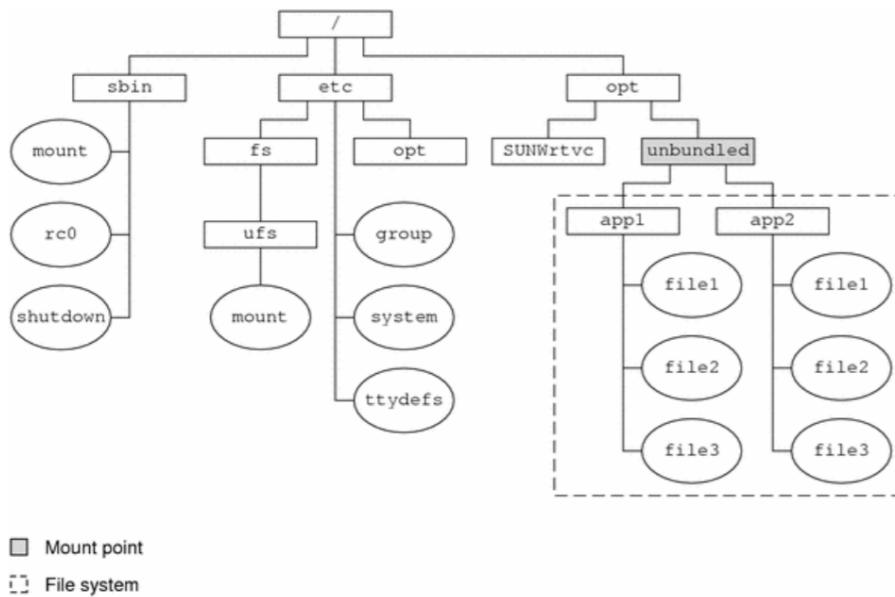
Per varie ragioni (per esempio installare sistemi operativi diversi) si può voler **partizionare** un disco:

- Ogni **partizione** si può usare come un dispositivo a blocchi a sé stante.
- Sono due gli standard usualmente utilizzati per partizionare:
  - **Master Boot Record** (MBR) che presenta: un boot sector che fornisce informazioni sulle 4 partizioni “primarie” massime (ce ne possono essere altre estese).
  - **GUID Partition Table** (GPT) partizioni illimitate, identificate da un UUID (Universally Unique Identifier).
- Nei sistemi Unix-like **file speciali a caratteri o blocchi** corrispondono a dispositivi di I/O, identificati da un **major** e un **minor number**.
  - per esempio, in Linux */dev/sda* corrisponde a un intero disco, mentre */dev/sda1*, */dev/sda2* e così via le singole partizioni;

- i dispositivi possono essere creati con *mknod()*, da root ma per essere utilizzati, vanno **montati**, tramite *mount()* (ma per essere montati vanno prima formattati).

## Mounting e Unmounting

In un sistema Unix c'è un solo file system; *mount()* "aggancia" l'albero di file e directory di un dispositivo all'albero globale, su un **mount-point**.



In questo modo su Linux esiste solo un unico File-System, mentre ad esempio su Windows esiste un file system per ogni disco: C: è il principale, D: E: F: sono quelli esterni (A: e B: erano per i floppy).

## Gestione file speciali

**Con l'aumentare dei dispositivi supportati dal kernel e l'arrivo di dispositivi hot-plug il sistema mostrava i suoi limiti:**

Necessitava di migliaia di file sotto */dev* e inoltre era complicato gestire cosa succedesse se un disco venisse (fisicamente) spostato, per esempio, in */etc/fstab*

Era quindi complicato gestire pendrive USB e altri dispositivi hot-plug.

Oggi, le partizioni si possono identificare in modo più stabile tramite **UUID**; si vedano *blkid(1)* e *fstab(5)*.

Oggi il kernel solleva eventi quando vengono collegati o scollegati dispositivi.

Processi utente, es. *udev(7)*, possono monitorare questi eventi e:

- **creare/rimuovere file speciali** sotto */dev*, che oggi è un *tmpfs*
- **caricare o scaricare moduli kernel**
- **notificare il file-manager**

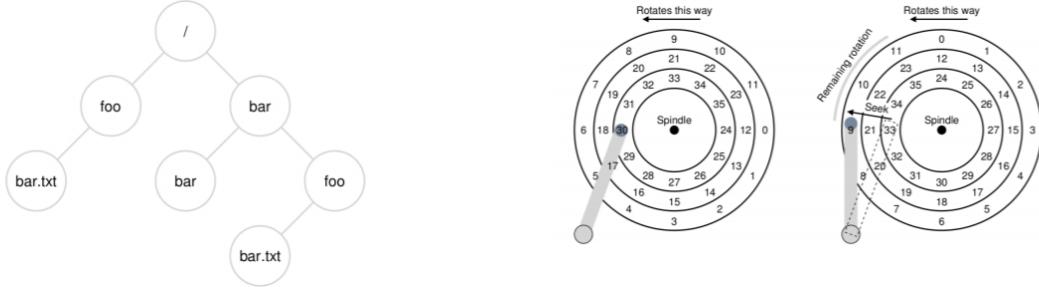
"qualcuno" (*udiskd*, *Nautilus*, *Thunar*, . . . ) può **montare automaticamente** il dispositivo appena collegato, tipicamente sotto */media/username /label*

## File e directory

Naturalmente, gli utenti di un s.o. usano le astrazioni di:

- **file** una sequenza di byte
- **directory** contenitori logici di file e directory

Problema: come implementare le astrazioni di file/directory su un dispositivo a blocchi?



## Implementazione

Un **file-system** è una struttura dati, che risiede su un dispositivo a blocchi e **gestisce dati e metadati**;

Noi consideriamo una versione semplificata di FS “alla Unix”, che il libro chiama **vsfs** (Very Simple File-System).

**Formattare**, rispetto a un certo formato, significa preparare questa struttura dati sul disco:

- Esistono vari formati (FAT, FAT32, NTFS, ext2/3/4); per formattare i vari formati si usano le istruzioni di tipo *mkfs*;

Non dimentichiamoci che **sono necessarie anche altre strutture dati**, per gestire l'accesso ai file da parte dei processi utente.

Per esempio, quando un processo usa *open()*, tra le altre cose il kernel deve:

1. recuperare l'inode (struttura su disco) corrispondente al percorso specificato come stringa
2. allocare una struttura che corrisponde al file aperto, che “punta” a (1)
3. allocare un FD nel PCB del processo, che “punta” a (2).

Relazione fra file descriptor e file aperti: POSIX

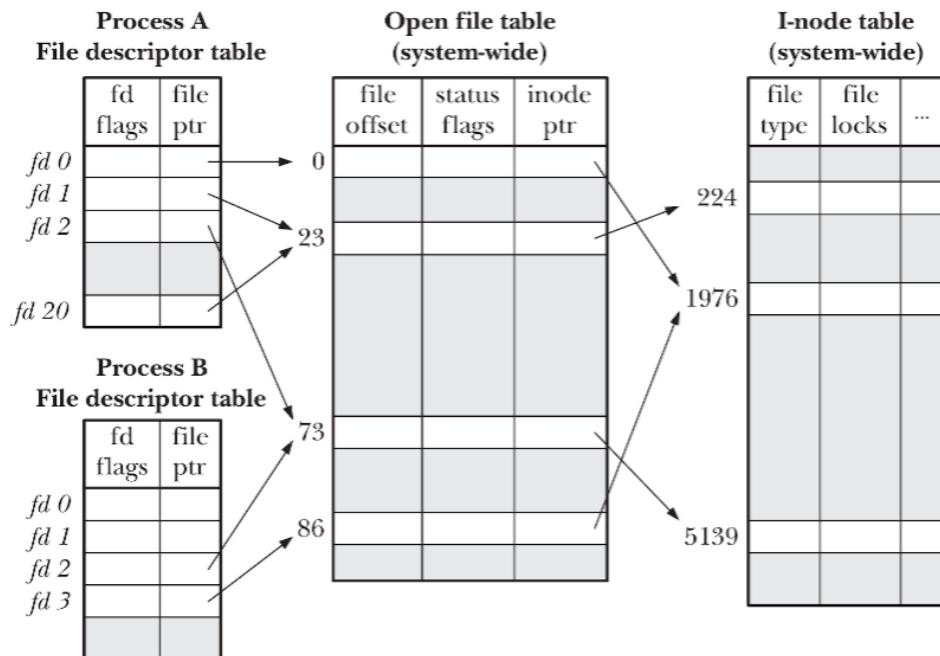


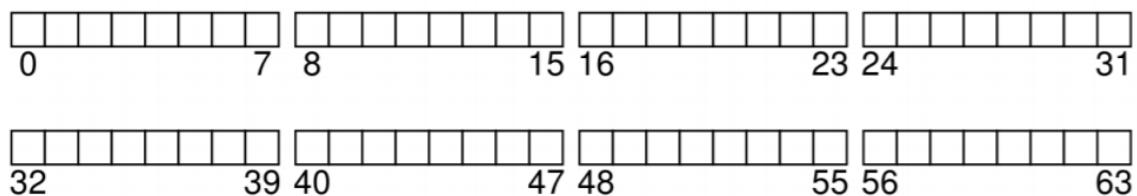
Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Non esiste solo il lato disco: ci sono alcune strutture che stanno su disco (l'inode per esempio), ma ci sono delle strutture che sono già in RAM.

## Organizzazione

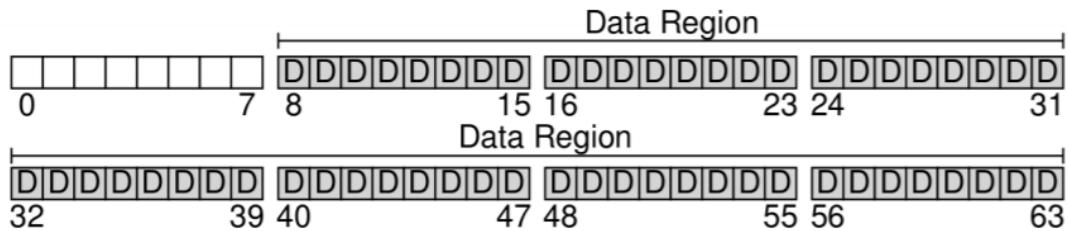
Continuando l'esempio del libro, assumiamo di avere un disco (per semplicità), o partizione, di **64 blocchi da 4k**.

Spesso i settori da 512 byte all'interno dei blocchi si raggruppano in blocchi logici, i **cluster**.



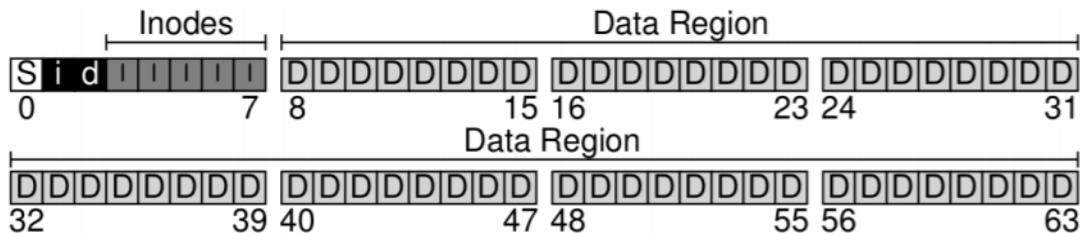
## Dati e Metadati

Non possiamo usare tutto il disco per i dati.



dobbiamo memorizzare i metadati sui file; in particolare:

- per ogni file corrisponde un **inode**;
- gli inode sono contenuti nella **tavella degli inode** quindi, un numero massimo di file che possono stare sul FS (siccome ogni file corrisponde ad un inode);
- per ogni inode corrisponde un **inode bitmap**: un modo efficiente per sapere se l'inode usato o meno; analogamente, per ogni blocco dati vi sono dei **data bitmap**;
- infine, esiste un **superblocco** per identificare il tipo di file-system e le sue caratteristiche (numero di inode, numero di blocchi dati, ...);



## inode

Ogni inode contiene i metadati relativi a un file (**ma non il nome**), come ad esempio:

- **tipo** di file, che può essere:
  - **regular file**
  - **directory**
  - **symbolic link**
  - **FIFO**
  - **socket**: non quelli internet, *che abbiamo usato nei laboratori*, ma quelli locali
  - **character device**
  - **block device**
- **UID/GID di proprietario e gruppo**
- **dimensione** in byte
- maschera dei bit relativi ai **permessi**
- **date**, ad esempio di creazione o modifica
- **numero di (hard) link**
- “**puntatori**” ai **blocchi dati**

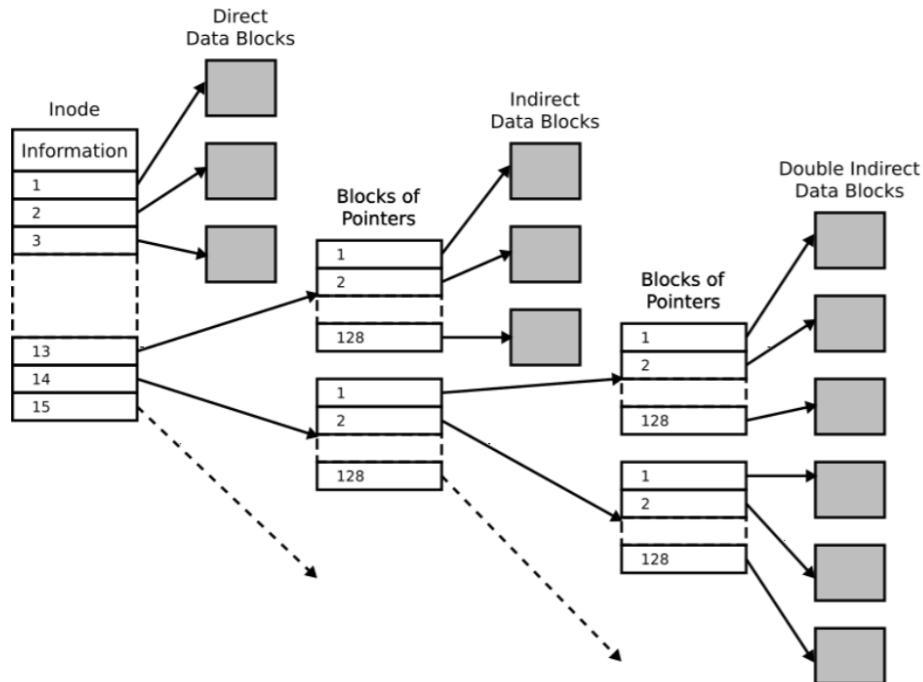
## Puntatori ai blocchi dati

Un file può essere più o meno lungo e può quindi usare uno svariato numero di blocchi dati; la tabella inode, al contrario, ha dimensione fissa;

Questa dicotomia può essere risolta in modo efficiente tramite l'uso di puntatori a blocchi dati: per i primi blocchi, abbiamo un array che ci portano al corrispettivo blocco dati, ed accedo **in modo efficiente**.

I puntatori possono accedere **ad altri array**, a cui accedo in modo meno efficiente.

Questo da il giusto equilibrio tra **efficienza e grandezza dei file**.



## Directory, link e cancellazione

Le directory sono file, che contengono associazioni tra nome e numero di inode:

$$\text{nome} \rightarrow \#\text{inode}$$

Queste associazioni sono **hard link** ed, in particolare, contengono:

- . (inode della directory corrente)
- .. (inode della parent)

Tramite il comando *In(1)* si possono creare:

- (**hard link**: aggiungere un nome a un file esistente (*a questo punto più nomi corrispondono allo stesso inode, il cui contenuto è sempre lo stesso*).  
*Un hard link da quindi un nome diverso ad un file che esiste già.*)
- **link simbolici (symbolic link)**: file (che quindi hanno il loro inode associato al nome), il cui contenuto corrisponde a un percorso, **non necessariamente esistente**.

le syscall sono due: *link(2)* e *symlink(2)*

**È possibile creare hard-link solo all'interno dello stesso FS**, questo perché il numero di inode ha senso solo nello stesso FS; su un altro FS, con un'altra tabella di inode, quell'inode avrebbe un senso diverso.

`rm(1)` si appoggia a `unlink(2)` ed elimina un link, ma **non è detto che elimini il file**; questo dipende dal numero di hard-link:

- anche se il numero è 0, finché ci sono FD aperti il file corrispondente viene “tenuto in vita”
- “trucco” spesso usato per i file temporanei

## Permessi

Per le directory:

- r** posso leggere? Ovvero, listare i file contenuti
- w** posso modificarne il contenuto? Creare/cancellare/rinominare/. . . nomi contenuti
- x** posso “entrare”/accedere?

Quindi, per creare/eliminare nomi da una directory *d*, dovete avere il permesso di scrittura su *d*;

Quindi, per esempio, possiamo cancellare file read-only di root se la directory è nostra.

## Risoluzione dei percorsi — path\_resolution(7)

1. Nel PCB ci sono **inode di root r** e **directory corrente c**  
Se il percorso inizia con “/”?
  - a. allora *d* = *r*
  - b. altrimenti, *d* = *c*
2. per ogni componente (separata da /) non-finale, che chiamiamo *n*:
  - a. si hanno i **permessi** di ricerca in *d*?
    - i. no → EACCES
    - ii. sì → vai avanti
  - b. si **cerca n in d**:
    - i. non si trova → ENOENT
    - ii. altrimenti si recupera inode corrispondente *i*
  - c. *i* è una **directory**?
    - i. Se sì, si riparte da *li*: *d* = *i*
    - ii. Se no, vado avanti
  - d. *i* è un **link-simbolico**? Si risolve a partire da *d*:
    - i. Il risultato non è una directory? → ENOTDIR
    - ii. Se lo è, *d* 0 , si continua da *li*: *d* = *d* 0 — un contatore evita loop infiniti
    - e. se non è né una directory, né un link simbolico, fallisco: ENOTDIR;
3. **per la componente finale non si pretende che sia una directory**;
4. . e .. hanno **significato speciale**:
  - a. Anche se il sottostante FS non li memorizza esplicitamente
  - b. Ma non si può salire sopra la radice: *./≡/*

## Consistenza del file-system

Se ci sono varie parti della struttura dati da aggiornare (e.g. blocco dati, bitmap e puntatori ai blocchi), **se manca la corrente a metà è un grande problema**. C'è un ordine di operazioni migliore di altri che mira a evitare il più possibile complicazioni.

## Fsck (file system consistency check)

Per controllare l'integrità di un file system (smontato, non in uso) uso ***fsck(1)***:

- controlla che il superblocco sia "ragionevole"
- consistenza bitmap blocchi liberi e puntatori ai file
  - più inode puntano a uno stesso blocco?
    - attraverso ***fix*** si può duplicare il blocco, dando a ognuno la sua copia;
  - Uso ***fix*** anche quando:
    - un blocco puntato risulta libero
    - un blocco non puntato risulta usato

La ***fsck*** controlla anche:

- stato degli inode (e.g. tipo valido)
- link count
  - e gestisce il recupero di file senza nome
- puntatori fuori dal range dei blocchi
- directory
  - verifica che ognuna abbia il suo . e ..
  - verifica se qualcuna è collegata più di una volta nell'albero

## Journaling

Ovviamente, il controllo di integrità è **molto lento**.

- sempre di più al crescere della dimensione dischi

Si effettua un'ottimizzazione: un file system smontato "in modo pulito" non viene controllato a ogni ***mount()***.

Essa viene implementato, nell'approccio moderno tramite il **Journaling (AKA write-ahead logging)** che:

- rende atomiche le (sequenze di) operazioni: prima di fare le modifiche, esse vengono segnate in un log; se c'è crash vengono riapplicate al ***mount()***.

**Roba Varia:**

La write può andare a modificare la bitmap dei dati quando finiamo i blocchi in cui è memorizzato il file e quindi bisogna allocarne di nuovi.

PATH specifica alla shell dove cercare gli eseguibili, è una variabile d'ambiente.

slow start, adaptive increase